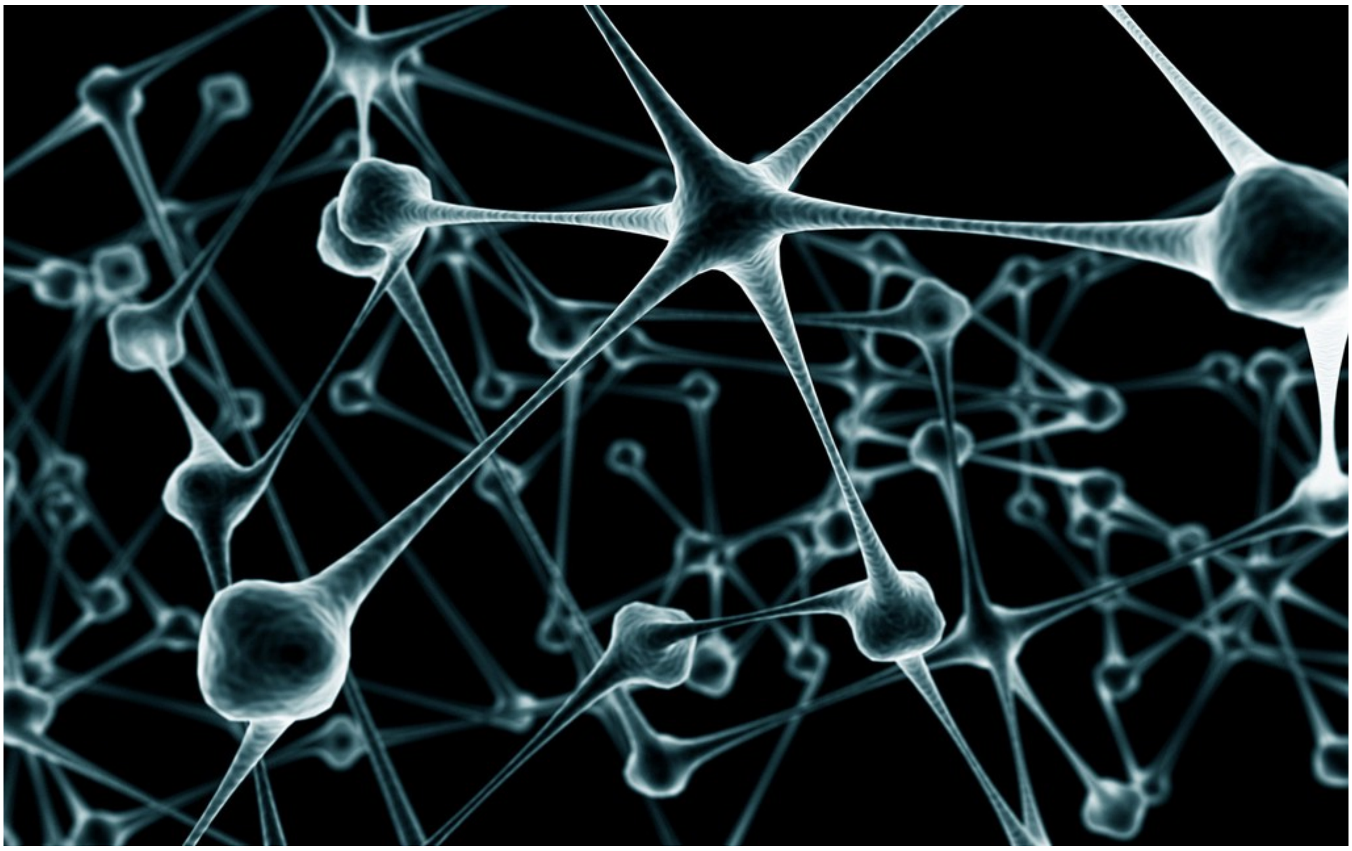
Deep Learning in Real Time — Inference Acceleration and Continuous Training

Synced
Aug 20, 2017 · 18 min read



Introduction

Deep learning is revolutionizing many areas of computer vision and natural language processing (NLP), infusing into increasingly more consumer and industrial products intelligence capabilities with the potential to impact the everyday experience of people and the standard processes of industry practices. On a high level, deep learning, similar to any automated system based on statistical machine learning techniques, works as a two-stage process.

First, a deep neural network (DNN) model specifically designed for the problem domain and data available is trained, usually on a GPU or high-performance CPU cluster for anywhere from tens of hours to a few weeks. Then, it is deployed into a production environment where it takes in a continuous stream of input data and runs inference in real time, yielding output either directly used as the end result or further fed into downstream systems.

Either way, applications that have an ever stricter latency requirement, driverless cars, and search engines, for instance, demand lightning-fast deep learning inference, usually within tens of milliseconds for each sample. Thus, beyond the academia's typical focus on faster training, the industry is often more concerned with faster inference, bringing inference acceleration to the spotlight and core of many hardware and software solutions.

Another important aspect of production deep learning system is to tackle the distributional shifts in the input and output data over time. Like any statistical machine learning model, the validity and effectiveness of a deep neural network critically hinge on the assumption that the distribution of the input and output data does not change significantly over time, rendering the patterns and intricacies the model originally learned underperforming or even unusable.

However, such assumption rarely holds true in the real world, especially in domains such as information security, where fast-paced evolution of the underlying data generating mechanism is a norm (in the case of security, it is because both players, the defender and the adversary, are constantly striving to outmatch his opponent by changing his own strategies, thus exploiting the opponent's unguarded vulnerabilities).

Therefore, as the prospect of using deep learning to better solve many once unresolvable problems march into such domains, the problem of continuous training of a deep neural network and how to do it well without jeopardizing production quality guarantees is gaining greater attention among both Machine-Learning-as-a-Service (MLaaS) providers and application architects.

In this report, we will touch on some of the recent technologies, trends, and studies on deep neural network inference acceleration and continuous training in the context of production systems. The goal of this report is not to delve too deep into the technicalities of one or two specific techniques, but to survey the broader landscape of hardware and software solutions to these two important problems, offer our readers a starting point of study, and hopefully inspire more people with diverse expertise to join our discussion and exchange knowledge.

Part I — Inference Acceleration

Inference Acceleration — Hardware

Both DNN training and inference are computation-intensive processes but in very different ways. Training demands high throughput, thus is most often carried out by GPUs, given their massive parallelism, simple control flow, and energy efficiency. It is common to batch hundreds of training inputs (for example images in a computer vision task, sentence sequences in a NLP task or spectrograms in a speech recognition task) and perform forward or backward propagation on them as one unit of data simultaneously to amortize the cost of loading the network weights from GPU memory across many inputs.

For inference, however, the paramount performance goal is latency. To minimize the network's end-to-end response time, inference typically batches a much smaller number of inputs than training, as automated services relying on inference are required to respond in near real time.

For example, the latency requirement for the entire Google Autosuggest pipeline is less than 200 milliseconds, including Frontend, load balancing, query understanding and auto-complete suggestion powered by DNN, and a full search stack traversal to display what would the result be if users actually search for one of the auto-suggested queries. Mission-critical applications like driverless cars put an even stricter latency requirement on the pipeline, with DNN being just one component running on an in-car embedded processor when user's own safety is at stake.

While training is mostly dominated by GPUs, there are several players in the inference hardware market.

GPU

Nvidia features its cutting-edge Pascal-architecture Tesla P4, P40 and P100 GPU accelerators with a peak 33x higher throughput than a single-socket CPU server while at the same time maintain maximum 31x lower latency, reported by an Nvidia study which compared inference performance of AlexNet, GoogleNet, ResNet-152 and VGG-

19 on a CPU-only server (single Intel Xeon E5–2690 v4 @ 2.6GHz) versus a GPU server (same CPU with 1XP100 PCIe).

To help developers better leverage its hardware, Nvidia's cuDNN library provides a series of inference optimizations for GPUs. In small batch scenarios, cuDNN improves on the problem of convolution algorithms not being able to parallelize enough threads to fully fill the GPU. The traditional algorithm such as precomputed implicit GEMM (generalized matrix-matrix product) is optimized for large output matrices, and its default parallelization strategy suffers from the problem of not being able to launch enough thread blocks provided that batch size is a multiplicative factor in one of the output matrix dimensions. Latest versions of cuDNN improved this algorithm by splitting in an additional dimension, which reduces the amount of computation per thread block and enables launching significantly more blocks, increasing GPU occupancy and performance.

Another major improvement is reduced precision floating point operations. In recent years, researchers have found that using lower precision floating point representations (FP16) for storage of layer activations and higher ones (FP32) for computation does not sacrifice classification accuracy, while in the meantime improves performance in bandwidth-limited situations and reduces overall memory footprint required to run the DNN. cuDNN introduced FP16 to convolution algorithms, delivering 2x performance boost compared to equivalent FP32 arithmetic.

Intel Processor Graphics

As AI usage in and via the cloud continues to grow quickly, the industry is witnessing a shift of the inference stage from high-end specialized servers outward to the edge of computing. This trend to terminal devices performing machine learning and deep learning locally versus solely relying on MLaaS is fueled by the necessity to reduce latency on a bandwidth-bottlenecked network, maintain > 99.999% service availability, spread costs across scaled-out devices rather than scaled-up servers, and address the heated issue of privacy and compliance.

Driven by the wave of smart appliances, driverless cars, automated manufacturing and virtual assistant on mobile platforms, Intel Processor Graphics has equipped its array of hardware products, including Intel HD Graphics, Intel Iris Graphics and Intel Iris Pro Graphics, with the ability to handle varying AI workloads. They are designed to strike a balance between inference acceleration and programming flexibility and are ubiquitous as a component of Intel SOCs from PCs to servers to embedded devices.

With a flexible Instruction Set Architecture allowing for rich data representations such as 16FP, 32FP, 16Int, 32Int and fast SIMD multiply-accumulate operations, these devices provide efficient memory block loading for optimized convolution and generalized matrix-matrix multiplication, both critical for fast and energy-efficient inference on edge devices.

To better utilize these hardware resources for inference, Intel's Deep Learning Deployment Toolkit performs static, compilation-time analysis on a trained DNN model to optimize execution on various endpoints. The Model Optimizer attempts horizontal and vertical layer fusion and redundant network branch pruning, before quantizing the network weights and feeding the reduced, quantized network to the Inference Engine, which further optimizes inference for target hardware with an emphasis on footprint reduction.

Given the Toolkit's decoupling from endpoint target devices, it can optimize inference for running on different hardware units including CPU, GPU and FPGA. For CPU inference acceleration it uses Intel's MKL-DNN plugin, and for GPU it leverages clDNN, a library of OpenCL kernels optimized for computer vision computation. For FPGA, it integrates with Intel's Deep Learning Inference Accelerator, a PCIe add-in card powered by Intel Arria 10 FPGA and optimized for popular CNN topologies including AlexNet, GoogleNet, CaffeNet, LeNet, VGG-16 and SqueezeNet.

FPGA

Generally speaking, with a flexible hardware configuration, FPGAs often provide better performance per watt of power consumption than GPUs, especially for sliding-window computations such as convolution and pooling. This makes them particularly attractive to industry users who ultimately care more about reducing costs for large scale applications and the ability to customize the inference architecture for a particular application.

Traditionally not competitive against GPUs on peak floating-point performance, the field of FPGA for DNN inference is advancing fast. The latest 14nm Intel Stratix 10 FPGA features over 5,000 floating-point units and over 28MB of on-chip RAM integrated with high-bandwidth memories, making it comparable to a Nvidia Titan X Pascal GPU (a report revealed Stratix 10 peaks at 9.2 TFLOPs for FP32, while Pascal Titan X offers 11 TFLOPs).

Two trends in DNN research are driving the adoption of FPGAs over GPUs: low precision data types and sparsity. Researchers have demonstrated accuracy improvements for 2-bit ternary and 1-bit binary DNNs, and have proposed techniques to induce neuron and weight sparsity as high as 90% by pruning, ReLU, ternarization, etc.

Together these techniques boost DNN algorithmic efficiency by orders of magnitude over dense FP32 architectures, conditioned on a proper implementation on hardware that can skip zeros efficiently. The monolithic, massively parallel GPU architecture falls short under these irregular parallelism patterns and custom data types, whereas FPGAs designed for extreme customizability shine. An Intel team recently reported 3x TOP/s and 4x GOP/s/Watt performance advantage of Stratix FPGA over Titan X GPU for low precision 6Int GEMM, 2x to 10x for 1-bit GEMM, and 4x for 85%-sparse GEMM.

TPU

Apart from general-purpose hardware repurposed or reprogrammed for DNN inference like CPU, GPU or FPGA, researchers and industrialists are investing heavily in ASICs (Application Specific Integrated Circuits) as well, believing that a dedicated chip design would yield ultimately superior performance for one single type of computational workload.

Google’s TPU, announced last year, is one such example. Powering a wide range of Google real time services including Search, Street View, Translate, Photos, and potentially driverless cars, TPU often delivers 15x to 30x faster inference than CPU or GPU, and even more per watt at a comparable cost level. Its outstanding inference performance originates from four, among others, major design optimizations: Int8 quantization, DNN-inference-specific CISC instruction set, massively parallel matrix processor, and minimal deterministic design.

Quantization greatly reduces the hardware footprint and energy consumption of the most important inference computation — matrix multiplication. Mobile and embedded system deployment also benefit from a reduced memory usage, which was reported by Google as more than 4x for common models like Inception. A TPU contains over 60,000 Int8 multipliers, almost 25x more than the number of FP32 multipliers on a common GPU.

TPU’s CISC instruction set focuses on directly representing and optimizing the major mathematical operations required for DNN inference — matrix multiplication and elementwise activation. The instruction set contains optimized CISC instructions for reading data block from memory, reading weight block from memory, matrix multiply or convolve the data and weight block and accumulate the intermediate results, apply hardwired activation functions elementwise, and write the result to memory.

Reconfigurable and reprogrammable, three hardware units, Matrix Multiplier Unit, Unified Buffer and Activation Unit, power the efficient execution of these specially designed instructions. The Matrix Multiplier Unit is a massively parallel matrix processor capable of running hundreds of thousands of matrix operations (multiplication and addition) in a single clock cycle, reusing both inputs (data block and weight block) for many different operations without storing them back to a register, and in this case, the 24MB SRAM Unified Buffer. During this massive matrix multiplication, all intermediate results are passed systemically between 64K ALUs without DRAM access, greatly reducing power consumption and increasing throughput.

Together these ASIC optimizations enable TPU to operate at near-peak chip throughput while sustaining a stringent latency guarantee for inference. Google reported that at 7ms per-prediction latency for a common MLP architecture, TPU offers 15x to 30x higher throughput than CPU and GPU, and for a common CNN architecture, TPU achieves peak 70x better performance than CPU.

Inference Acceleration — Algorithms

Algorithmically, one promising approach to reducing inference latency and DRAM footprint (hence power consumption) is model compression. A compressed model that can easily fit into on-chip SRAM cache rather than off-chip DRAM memory will facilitate the application of complex DNNs on mobile platforms or in driverless cars, where memory size, inference speed, and network bandwidth are all strictly constrained.

Fully connected layers are known to be overparametrized in most state-of-the-art DNN architectures. Much research has focused on compressing FC layers, either by bucketizing connection weights (pseudo)randomly using a hash function or by vector quantization. Network-in-Network proposed to replace FC layers with global average pooling, with an additional linear layer added at the top for better transferability.

Benchmarked on CPU, desktop GPU and mobile GPU, Deep Compression yields 30x to 50x more compact AlexNet and VGG-16 models that have 3x to 4x layerwise speedup and 3x to 7x higher energy efficiency, all without loss of accuracy on ImageNet. It accomplished so by a three-phase compression scheme: network pruning, trained quantization, and Huffman coding.

Pruning removes trained network connections with weights lower than a preset threshold, after which the DNN is retrained to adjust the weights of the remaining sparse connections. Positional index differences of the sparse structure, encoded in 8 bits for convolution layers and 5 bits for fully connected layers, are stored in CSR or CSC format. On average, pruning reduced 10x the number of parameters for AlexNet and VGG-16.

To further reduce the number of bits required to encode each weight, Deep Compression ties weights by a binning technique. For a layer with M input neurons and M output neurons, the M x M weights are quantized into M bins with all the weights in one bin sharing the same value, thus for each weight, only an index into a shared weights table needs to be stored. During backpropagation, gradients are grouped in a similar way, binned into M buckets, and then summed to update the shared weight centroids. Quantization by weight sharing effectively reduces the number of weights that need to be stored, and fine-tuning the shared weights reduces the number of update operations required. For a pruned AlexNet, whose convolutional layers were quantized to 8 bits and fully connected layers to 5 bits, its size was further compressed by 3x.

Huffman coding takes advantage of the fact that, after pruning and quantization, distributions of the weights, represented as sparse matrix indices, are heavily biased, usually with a small number of peaks clustered close to each other. Huffman coding on average compressed 20% — 30% of AlexNet and VGG-16 sizes.

Notably, targeting extremely latency- and power-bound applications running in real time on mobile platforms, Deep Compression was benchmarked with no batching at all (batch size = 1). Fully connected layers without weight sharing usually dominate the model size, contributing over 90% of storage required; they are the layers that benefit from Deep Compression the most. Trimming off over 95% of FC layer weight size, Deep Compression enables a fully trained DNN to be loaded into the SRAM of an embedded processor inside a driverless car, thus providing on-chip in-memory inference at low power consumption. A corresponding hardware architecture, the Efficient Inference Engine, was also proposed to perform inference on deep-compressed models. Loading compressed DNNs into SRAM gave EIE 120x energy saving and with sparsity, weight sharing, and skipping zero activations, 15x faster inference than GPU at a peak of 102 GOPS/s.

Part I — Summary

In Part I of this report, we introduced some of the recent hardware and algorithm technologies for DNN inference acceleration. In Part II, we will talk about DNN continuous training.

Part II — Continuous Learning

Introduction

In Part I of this report, we introduced some of the recent hardware and algorithm technologies for DNN inference acceleration. In Part II, we will talk about DNN continuous learning, based on a recent paper “Fine-Tuning Deep Neural Networks in Continuous Learning Scenarios” by Christoph Kading, Erik Rodner, Alexander Freytag and Joachim Denzler.

An important aspect of production deep learning system is to tackle the distributional shifts in the input and output data over time. Like any statistical machine learning model, the validity and effectiveness of a deep neural network critically hinge on the assumption that the distribution of the input and output data does not change significantly over time, rendering the patterns and intricacies the model originally learned underperforming or even unusable. However, such assumption rarely holds true in the real world, especially in domains such as information security, where fast-paced evolution of the underlying data generating mechanism is a norm (in the case of security, it is because both players, the defender and the adversary, are constantly striving to outmatch his opponent by changing his own strategies, thus exploiting the opponent’s unguarded vulnerabilities). Therefore, as the prospect of using deep learning to better solve many once unsolvable problems march into such domains, the problem of continuous learning of a deep neural network and how to do it well without jeopardizing production quality guarantees or raising resource consumption is gaining greater attention among both Machine-Learning-as-a-Service (MLaaS) providers and application architects.

In the second part of this report, we will formulate the continuous learning scenario and introduce an incremental fine-tuning approach. Then we present 3 important findings backed by several empirical studies. The purpose of this discussion is not to survey the broad landscape of continuous learning research, but to inspire more people with diverse expertise to join our discussion and exchange knowledge.

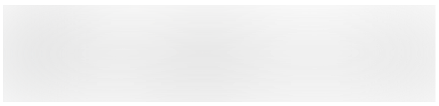
Part II — Continuous Learning

Continuous Learning Scenarios and Incremental Fine-Tuning

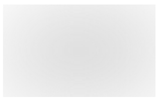
Originally, fine-tuning has been referring to the process of pretraining a DNN with a generative objective, followed by an additional training stage with a discriminative objective. Early works on pretraining and fine-tuning Deep Belief Networks and Deep Stacked Autoencoders all follow this approach. The expectation is that the generative training stage will guide the network towards learning a good hierarchical representation of the data domain, and the discriminative stage will take advantage of this representation and hopefully learn a better discriminator function more easily in the representation space.
More recently, researchers have been using fine-tuning to pretrain a sophisticated, state-of-the-art DNN on large general-purpose data sets like ImageNet, and then fine-tune the model on a smaller data set of interest. This helps alleviate the problem of insufficient labeled training data for some domains like medical diagnosis or geographical probing where obtaining labeled data is labor-intensive or expensive. The underlying assumption is that a reasonably good result on the large training data set already puts the network near a local optimum in the parameter space so that even a small amount of new data is able to quickly lead the network to an optimum.
From the perspective of continuous learning, both methods above are extreme cases where the network is only trained twice — initial pretraining and a one-time update. More general forms of continuous learning where training and updating becomes an iterative process pose the question of how an ongoing series of updates can be performed robustly and efficiently. Robustness is important to production systems because usually a real-time system does not tolerate a sudden drop in model performance, and the real-time nature of the system necessitates a high-efficiency requirement on both resource consumption and time was taken. For this reason, in the discussion below, we focus on scenarios where each update step only takes in a small amount (compared to the original full training data set) of new data, but an updated model is expected to be immediately available.

Continuous Learning Scenarios

In the most general form, entire training data sets evolve over time. However, for learning to be feasible, we confine ourselves to the case where input domain remains the same. This does not mean the data distribution over the input domain is constant — rather we allow for the broader case where distributions can vary but are always defined on the same domain. On the other hand, we assume the output domain can change. This is to accommodate the case where new labels emerge when the system keeps running for a relatively long time. This assumption holds true much more frequently than input domain changes because in most production systems, for example, autonomous vehicles, we cannot expect the system to be even remotely functional if the input videos or LIDAR images vary drastically (due to extreme weather, lighting, terrain or road conditions), but we do want the system to be able to accommodate a new type of unhittable object shows up and adapt to this new class of labels during continuous training. In that regard, we define the data in our continuous learning scenario as a sequence of data sets



indexed by time t. Our goal is to learn a network



time step t. Depending on how varies, there are two possible scenarios:

1. We don't receive new classes over time. Rather, our data set keeps growing…
2. We receive examples of new classes over time…

In each time step t, we need to update our network with the newly available information i.e. update set. Given our assumption of small update sets, to avoid underfitting, we warm-start optimization at time t with the converged from the last step. The validity of this technique hinges on the assumption that varies smoothly with respect to the expanding data set — this is why we are assuming small update steps and constant input domain. In reality, even if those assumptions can (partially) hold, we are still relying on one more expectation that, by initializing at a previous local optimum, the new optimization problem with a different parameter space landscape can quickly and stably converge to a new one. Most modern DNNs, however, are too complicated to guarantee this — their objectives are highly nonlinear and non-convex, sometimes with bad local condition numbers dominating the parameter space. Even so, warm-start is by far one of the best ways to cope with this, which is definitely better than starting over, and empirically shows the best promise. For scenario 2, at each time step, we potentially need to add additional neurons to the last output layer, along with connection weights with the preceding layer.

Three important questions remain. One, how many SGD steps are required at each update step? Two, how much data from previous and current step do we use during each update? Three, is this update procedure robust against labeling noise? All these questions are important for a production system because of the robustness and efficiency concerns we talked about previously.

Empirical Studies

In the experiments, we use BVLC AlexNet pretrained on ImageNet ILSVRC-2010 as an example network. We fix our learning rate at 0.001, L2 weight decay at 0.0005, momentum at 0.9 and update set size with examples from the same class. For evaluation, we use MS-COCO-full-v0.9 and Stanford40Actions, both small data sets compared to ImageNet because we want to evaluate continuous learning performance with small updates compared to initial pretraining. For MS-COCO, we use 15 classes, each containing 500 to 1000 examples; for each example, retain only ground truth bounding boxes with at least 256 pixels width and height. To evaluate, we randomly pick 10 classes to initially fine-tune the CNN. From the remaining data, choose 5 more randomly as novel data (new data coming in at each update step), and choose 100 examples randomly for each class. Classification accuracy is used as the measure at each update step. We bootstrap the experiments nine times to obtain an unbiased comparison.

Number of SGD Iterations Per Update Step

While keeping the SGD minibatch size fixed to 64, we vary the number of SGD iterations conducted at each update step. We represent the number of SGD iterations as a ratio relative to the total training data size i.e. number of epochs, to compensate for the growing training data size.

Surprisingly, classification accuracy remains robust to even much fewer SGD iterations per update, except for the case where only one tenth of data is used, likely because some classes are underrepresented in this small epoch. In fact, we can simply fix the number of SGD iterations to a small constant e.g. update set size, without significant performance degradation.

Ratio of Old and New Data Per Update Step

For incremental learning algorithms, the ratio of influence from old and new data is one of the most important hyperparameters. We sample each example i during SGD iterations with probability. At the extremes, means previous data are completely neglected and postpones the influence of new data to the next update step because that is when they end up.

We see that old data is necessary to prevent overfitting to the smaller update set. An interesting observation is, because new update set data is considered part of in the next step t, and yield comparable performance.

Robustness Against Labeling Noise

In a production scenario, labels for the new data are rarely noiseless — due to the real-time requirement, human judges often times have to make quick but inaccurate decisions, and disagreement rate between three to five judges can be as high as 75%. Thus, robustness against labeling noise is critical for a continuous learning scheme. We perform 10 epochs at each update step with a minibatch size of 64, and randomly pollute a fixed fraction of the newly available data by replacing them with examples from remaining classes while keeping their labels intact. As expected, labeling noise degrades accuracy, but our continuous learning scheme is relatively robust against it, sacrificing only 2% performance under a 10% noise.

Part II — Summary

In Part II of this report, we looked into a recent work on continuous learning and shed some light on the effectiveness and robustness of incremental fine-tuning. We hope this will inspire our readers to join our discussion on ML systems and exchange knowledge with people from different backgrounds.

References

[1] Nvidia developer blog https://devblogs.nvidia.com/parallelforall/inference-next-step-gpu-accelerated-deep-learning/
[2] Nvidia whitepaper http://www.nvidia.com/content/tegra/embedded-systems/pdf/jetson_tx1_whitepaper.pdf
[3] Intel clDNN Github repository https://github.com/01org/clDNN
[4] *In-Datacenter Performance Analysis of a Tensor Processing Unit*, Norman P. Jouppi et al., Apr 2017, https://arxiv.org/ftp/arxiv/papers/1704/1704.04760.pdf
[5] *Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding*, Song Han et al., Feb 2016,

https://arxiv.org/pdf/1510.00149.pdf
[6] *EIE: Efficient Inference Engine on Compressed Deep Neural Network*, Song Han et al., May 2016, https://arxiv.org/pdf/1602.01528.pdf

· · ·

Author: Yanchen Wang

Machine Learning Deep Learning Neural Networks Technology

About Help Legal