DeepLearningKit - an Open Source Deep Learning Framework for Apple's iOS, OS X and tvOS developed in Metal and Swift

Amund Tveit*
Memkite
amund@memkite.com

Torbjørn Morland*
Memkite
torb@memkite.com

Thomas Brox Røst*
Atbrox
thomas@atbrox.com

Abstract

In this paper we present DeepLearningKit - an open source framework that supports using pre- trained deep learning models (convolutional neural networks) for iOS, OS X and tvOS. DeepLearningKit is developed in Metal in order to utilize the GPU efficiently and Swift for integration with applications, e.g. iOS-based mobile apps on iPhone/iPad, tvOS-based apps for the big screen, or OS X desktop applications. The goal is to support using deep learning models trained with popular frameworks such as Caffe, Torch, TensorFlow, Theano, Pylearn, Deeplearning4J and Mocha. Given the massive GPU resources and time required to train Deep Learning models we suggest an App Store like model to distribute and download pretrained and reusable Deep Learning models.

1 Introduction

The Metal programming language is the recommended and most the efficient way of utilizing the GPU on Apple's iOS since 2014 [1, 2, 3, 4], and tvOS and OSX since 2015 [5, 6, 7]. This paper gives an overview of a Metal and Swift based Deep Learning library named **DeepLearningKit**, in particular parts of Metal convolutional neural network operators for the GPU. DeepLearningKit supports on-device Deep Learning on Apple's iOS, OS X and tvOS.

DeepLearningKit currently has shader functions for convolutional neural networks implemented in Metal and parallelized for the GPU - operators include: convolution, pooling, rectifier and softmax. In terms of deep learning model supported it has support for Min Lin's Caffe-trained Network In Network[8] (NIN - trained on CIFAR-10, CIFAR-100 and ImageNet data sets). We also have preliminary support running Theano[9] trained LeNet (trained on MNIST digit classification dataset). The reason we have chosen NIN is that the network is small compared to other deep convolutional neural networks, but at the same time provide very high classification accuracy on images, e.g. better than AlexNet. GoogleLeNet (winner of Imagenet 2014) uses a similar approach as NIN[**?**]. NIN can perhaps also be used in non-image domains, e.g speech recognition[10] or natural language processing[11]. In particular one could attempt to adapt Zhang and Lecun's encoding and 1D convolutional operators in "Text Understanding from Scratch"[12] and use it with NIN.

2 DeepLearningKit

This section gives describes how DeepLearningKit was built and with early empirical results using it.

*http://DeepLearningKit.org

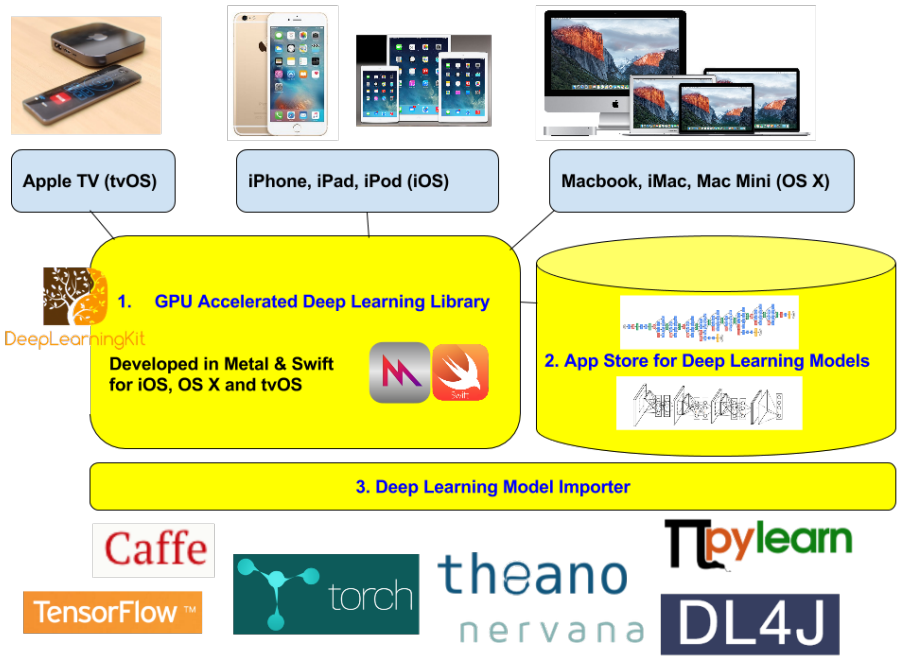

Figure 1: DeepLearningKit

2.1 Metal GPU Compute API in a Nutshell

All of the configuration of Metal happens in Swift (or Objective-C), and Metal itself is where to write the computational functions (shaders)

Metal has a threading model similar to Vulkan (SPIR-V) as shown in figure 2 (source: [13]), with one or several command queues (MTLCommandQueue) that each store and run a sequence of command buffers (MTLCommandBuffer) that processes data on the GPU (MTLBuffer). Figure 3 shows a (partial) comparison between Metal/Swift API and C++/OpenCL (OpenCL can be used to generate Vulkan/SPIR-V code).

Each MTLCommandBuffer has a compute command encoder (MTLComputeCommandEncoder) that specifies how the command buffer should be executed on the GPU, e.g. number of threadgroups and threads per threadgroup (specified with MTLSize and dispatchThreadGroups()) as well as the Metal shader function to perform (MTLFunction) that is loaded from the Metal (source or binary) Library (MTLLibrary). The Metal shader function, e.g. convolution() or rectifier() is wrapped inside a compute pipeline descriptor (MTLComputePipeLineDescriptor) before it is inserted into the MTLCommandBuffer.

Since Convolutional Neural Networks is a set of layers (typically with matrix or vector calculations per layer), we represented each layer as a MTLCommandBuffer (with the approriate Metal shader) and inserted all of those into a MTLCommandQueue. Data (both on Swift and Metal side) was pre-allocated (e.g. MTLBuffer) before the calculation was started on the GPU.

2.2 GPU algorithms used in DeepLearningKit

Calculating convolution layers are the most computationally expensive part of deep convolutional neural networks, since it involves matrix multiplication - typically with the GEneral Matrix to Matrix Multiplication (GEMM) function parallelized for the GPU.

The approach for GPU based GEMM used in DeepLearningKit is similar to that of Nvidia's cuDNN CUDA based Deep Learning library, i.e. using an im2col() transformation followed by convolution() function, see Shaders.metal on github.com/deeplearningkit/deeplearningkit for implementation.

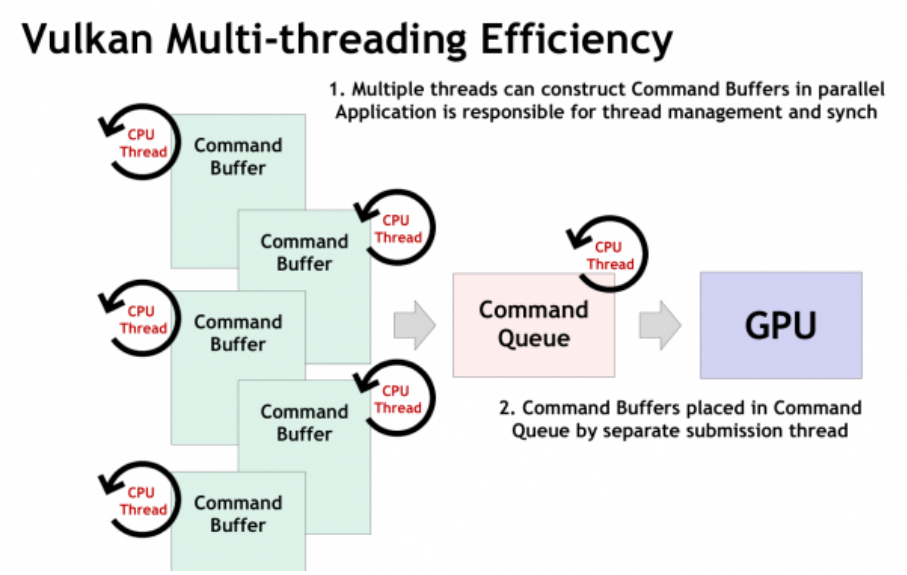

Figure 2: Metal - similar setup as Vulkan (SPIR-V)

#	Swift/Metal	C++/OpenCL	Description
1	MTLCreateSystemDefaultDevice() / MTLCopyAllDevices()	clGetDeviceIDs()	Get access to GPU
2	MTLDevice.newCommandQueue()	clCreateCommandQueue()	Prepare GPU command Queue (for command buffers MTLCommandBuffer on Metal)
3	MTLDevice.newDefaultLibrary()	clCreateProgramWithSource()	Get access to shader code
4	MTLDevice.newDefaultLibrary().newFunctionWithName() / MTLComputePipelineDescriptor()	clCreateKernel()	Initiate an actual shader
5	MTLDevice.newBufferWithBytes()	clCreateBuffer()	Create a GPU accessible memory buffer
6	MTLCommandBuffer.commit()	clEnqueueNDRangeKernel()	Start compute job to GPU
7	MTLCommandBuffer.waitUntilCompleted	clFinish()	Wait for GPU compute job to finish

Figure 3: Metal/Swift compared to C++/OpenCL

2.3 Experiences with PowerVR G6430/GT7600 on iPhone 5S/6S

The performance of DeepLearningKit from iPhone 5S (with PowerVR G6430 according to The iPhone 5S Review (AnandTech)) to iPhone 6S (with PowerVR GT7600 according to Apple iPhone 6S Plus vs. Samsung Galaxy S6 Edge+) we got 1 order of magnitude in improved performance. Calculation time to run through a 20 layer deep convolutional neural network model for image recognition went from approximately 2 seconds to less than 100 milliseconds. The network we used was NIN network trained on CIFAR-10. Based on XCode profiling we suspect that the Metal compute drivers for the GPU weren't fine tuned, so with lower level tools (e.g. for OpenCL/Vulkan SPIR-V) for tuning for the GPU we could probably improve performance quite a bit.

(Note that 100 milliseconds or in other words 0.1 seconds is what Jacob Nielsen stated is one of 3 important response times that a user feels a system reacts instantenously)

Based on XCode profiling with Instruments (figure 4) we suspect that our Metal/Swift code and perhaps the underlying Metal compute drivers still have potential to be improved (the time required to run through the network in a forward pass is about 93 milliseconds (ref lower right in figure), but Xcode instruments shows that the total duration for the GPU compute steps are about 0.5 milliseconds (537.62 microseconds, ref blue line at the bottom of figure 3), i.e. i.e. only approximately 1/200th of the total forward pass time (note: the entire forward pass does more than computation

on the GPU, the most likely time cost is memory allocation/copying and synchronization between GPU and CPU)

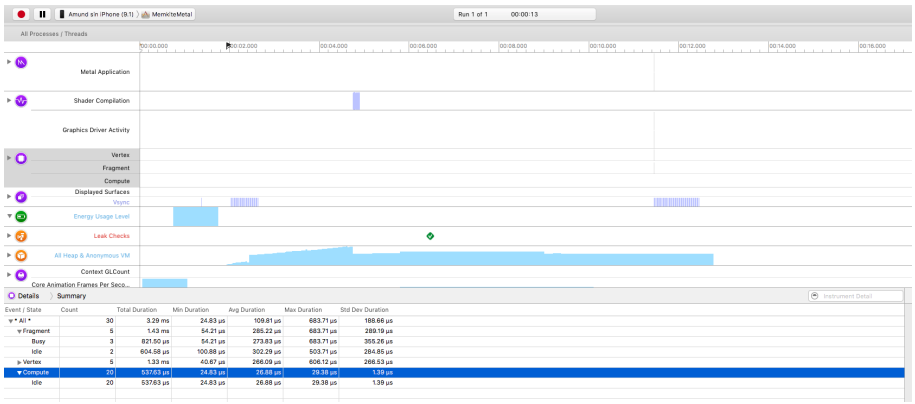

Figure 4: DeepLearningKit run - XCode Instruments Output - GPU Hardware

2.4 Effort needed to port from Metal/Swift to OpenCL/Vulkan Compute SPIR-V

Code needed to set up and run deep learning on the GPU, load/save data, and setup the deep learning pipeline (convolutional neural network)is done is done in Swift (for easy app integration on iOS, OS X and tvOS), but can be moved to a language of selection (e.g. Java on Android or C++/C on other devices). The Swift API for setting up Metal resembles the corresponding OpenCL C API as shown in Figure 3.

The Deep Learning GPU code (e.g. shader functions with calculations of convolution etc) is written in Metal, a language that is a subset C++11 and also has its own (relatively few) additions compared to C++11. Porting the Metal code GPU code to OpenCL should be relatively straight forward since OpenCL is also a subset of C++, as an example see figures 5 and 6 for a rectifier function written in both Metal and OpenCL. Going from OpenCL to Vulkan SPIR-V can be done with compiler (figure **??**) for further profiling and optimization.

```
// Returns max(0, X[id])
kernel void rectifier_linear(device float* X [[ buffer(0)]],
                             uint id [[ thread_position_in_grid ]]) {
    X[id] = fmax(0.0, X[id]);
}
```

Figure 5: Rectifier Function in Metal

```
// Returns max(0, X[id])
__kernel void rectifier_linear(__global float X) {
    int id = get_global_id(0);
    if(id < count) {
        X[id] = fmax(0.0, X[id]);
    }
}
```

Figure 6: Rectifier Function in OpenCL

Threading model in Vulkan is close to 1-1 with Metal (figure 2), so that should not be an issue for porting

2.5 Roadmap for DeepLearningKit

Here follows a brief overview of things we are working on or is on our roadmap.

1. use FFT-based convolution - with precalculated convolution filters [14, 15]
2. use lower resolution on floating point - in order to increase performance and support larger models (for now it uses 32 bit float or complex numbers - i.e. 2x32 bit per complex number to prepare for FFT-based convolution) [16, 17]
3. avoid copying memory between CPU and GPU more than needed [18]
4. add support for other types of pre-trained networks than deep convolutional neural networks, e.g. recurring neural networks[19, 20]
5. look into more in-place calculations to save memory, i.e. supporting larger models
6. try to exploit larger parts of Metal API wrt memory layout, threadgroups to increase performance (this relates to 1.) [21, 22, 23, 24, 25]
7. Look into teacher-student deep networks or other compressed models for even smaller but still high quality models (recent research have shown AlexNet models being compressed from 240MB to 6.9MB), see the paper [A Deep Neural Network Compression Pipeline]
8. Look into algorithms for approximate matrix multiplication (i.e. convolution step speedup) to further increase speed (and reduce energy usage), interesting techniques include a) [Approximating matrix multiplication and low-rank approximation], [Fast Approximate Matrix Multiplication by Solving Linear Systems] and [Fast Monte-Carlo Algorithms for Approximate Matrix Multiplications].
9. Look into a broad set of Deep Learning applications, e.g. categories in figures 7 from the research bibliography at [http://Deeplearning.University]. It might be application specific optimizations that can be done, e.g. in the case of natural language processing with convolutional neural networks one uses 1D convolution instead of 2D (as in image classification).

3 App Store for Deep Learning Models

Given the immense asymmetry in time taken to train a Deep Learning Model versus time needed to use it (e.g. to do image recognition), it makes perfect sense to build a large repository of pre-trained models that can be (re)used several times. Since there are several popular tools used to train Deep Learning models (e.g. Caffe, Torch, Theano, DeepLearning4J, PyLearn and Nervana) were working on supporting importing pre-trained models in those tools into an app store for deep learning models (currently weve been primarily been working with Caffe CNN models).

The tweet in Figure 8 illustrates how much energy is required to train a Deep Network (per night), some Deep Learning Models can take weeks of training on GPUs like the Nvidia TitanX, or in other words piles of wood of energy. Using a model is quite different since it requires less energy than lighting match. See figures 9 and 10 for an illustration of this.

Deep Learning Models also typically have a (low) limit in the number of classes they can predict per model (e.g. in the ImageNet competition there are 1000 classes, CIFAR-100 100 classes and CIFAR-10 10 classes). This means that in order to create real-life applications one need to intelligently (and very rapid load them from SSD into GPU accessible RAM) switch between several Deep Learning Models, or if there is enough capacity one can run several models in parallel on the same GPU. Selecting an approriate Deep Learning model (i.e. which is the most likely to work well in a given context) is to our knowledge not a well-studied field of research, and in some ways it resembles the meta or universal search problem found in web search (e.g. cross-model ranking), but latency plays an even bigger part in the mobile on-device case (dont have time to run many models). We have some ideas for a meta model for selecting a model to use, which can use input like location, time of day, and camera history to predict which models might be most relevant.

With state-of-the-art compression techniques for Convolutional Neural Network the (groundbreaking) AlexNet model from 2012 can be compressed from 240MB to 6.9MB. This means that one could theoretically fit more than eighteen thousand AlexNet models on a 128 GB mobile device like the iPhone 6!

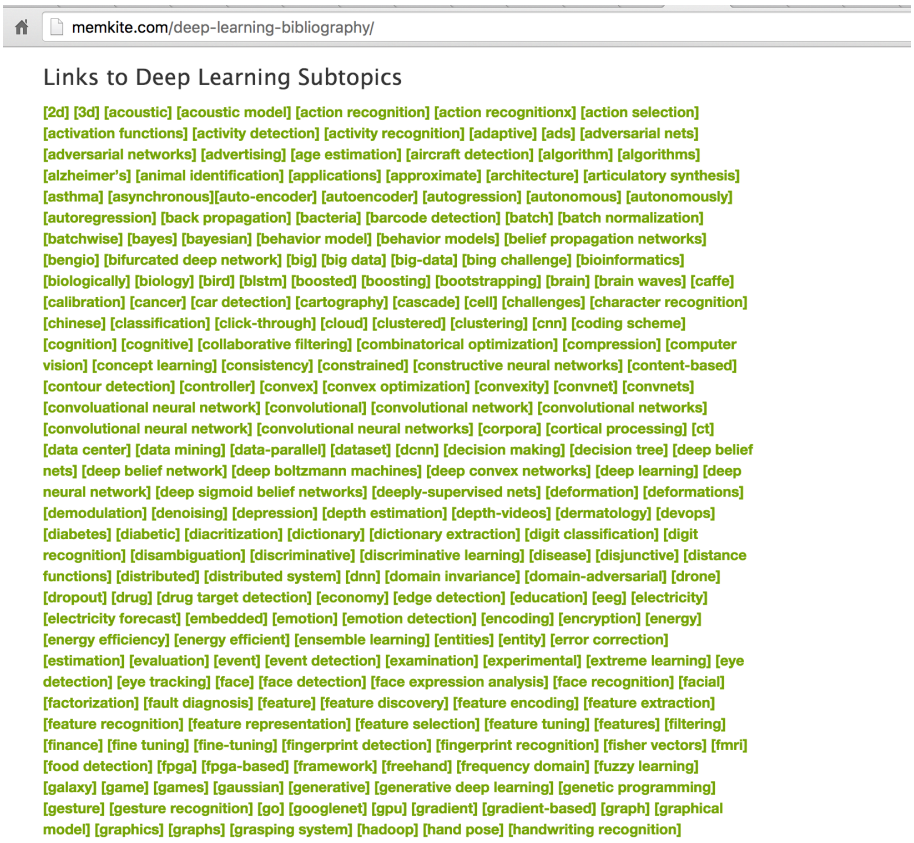

Figure 7: Memkite's http://Deeplearning.University - Sample Keywords

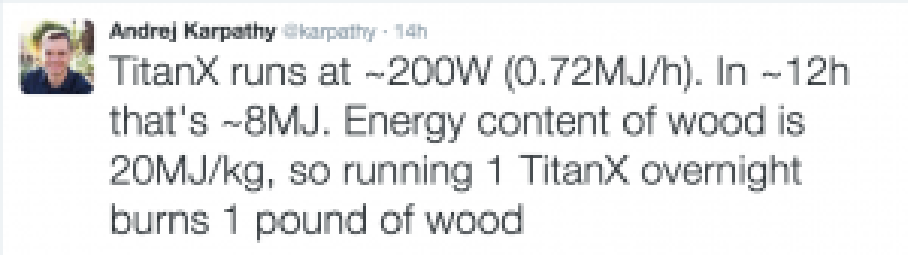

Figure 8: Tweet about deep learning training energy usage from Andrej Karpathy

4 Deep Learning Model Importer

Importing Deep Learning models into the model app store requires supporting the main Deep Learning tools. The most used ones in research are Torch and Caffe (with Google's TensorFlow growing very fast), and DeepLearningKit currently supports converting trained Caffe models to JSON and then importing into Swift/Metal (or OpenCL/Vulkan with porting) for the mobile app using Vadim Kantorov's caffemodel2json. Making support for importing convolutional neural network from other tools might require getting intimate insight into the tools, but since convolutional neural networks are quite similar of nature the complexity and effort for creating importers is not horrific. Proposing and supporting standards - e.g. for

1. deep learning network description (i.e input to training stage)
2. input data formats (images, text, etc input to training stage)

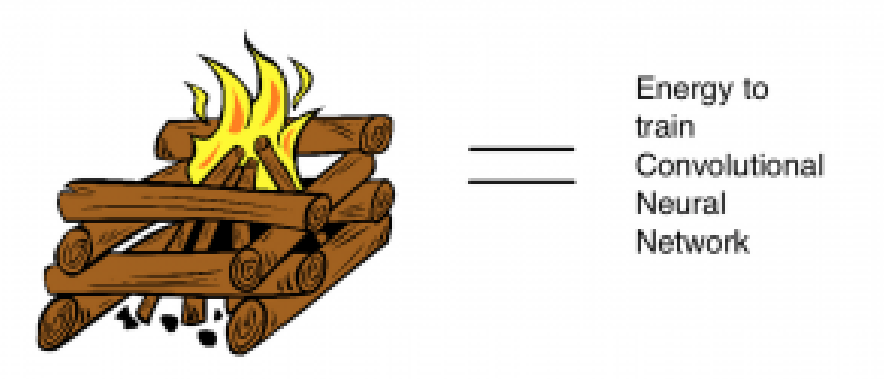

Figure 9: Energy needed to train CNN

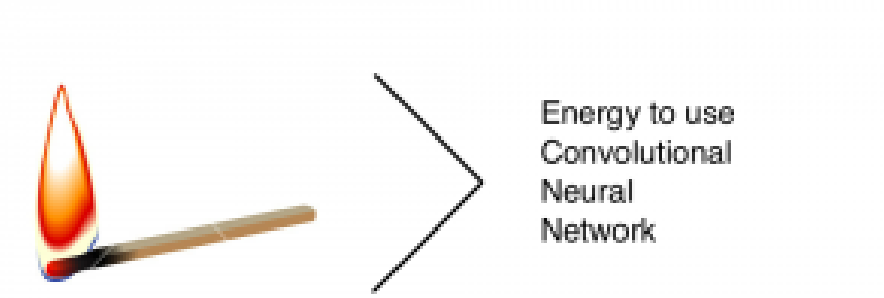

Figure 10: Energy needed to run CNN

3. trained networks (i.e. input to DeepLearningKit)

might be a longer term goal since this will make it easier to use pretrained models with OpenCL/Vulkan no matter which tool they are created in.

5 Conclusion

Have done a presentation of DeepLearningKit GPU accelerated Deep Learning for Metal/Swift. DeepLearningKit is published as open source code on github.com/deeplearningkit and documentation/tutorials can be found at DeepLearningKit.org. Resources related to this paper will be published on http://deeplearningkit.org/publications. A tutorial for using DeepLearningKit with iOS can found at http://deeplearningkit.org/tutorials-for-ios-os-x-and-tvos/tutorial-using-deeplearningkit-with-ios-for-iphone-and-ipad/

References

[1] Jeremy Sandmel. Working with Metal - Overview. Apple WWDC, published online - `https://developer.apple.com/videos/wwdc/2014/`, 2014.

[2] Richard Schreyer and Aaftab Munshi. Working with Metal - Fundamentals. Apple WWDC, published online - `https://developer.apple.com/videos/wwdc/2014/`, 2014.

[3] Gokhan Avkarogullari, Aaftab Munshi, and Serhat Tekin. Working with Metal - Advanced. Apple WWDC, published online - `https://developer.apple.com/videos/wwdc/2014/`, 2014.

[4] Amund Tveit. GPGPU Performance of Swift/Metal vs Accelerate on iPhone 6/5S, iPad Air and iPad Mini. Memkite, published online `http://memkite.com/blog/2014/12/18/`

gpgpu-performance-of-swiftmetal-vs-accelerate-on-iphone-6-5s-ipad-air-and-ipad-mini/, 2014.

[5] Rav Dhiraj. What's New in Metal, Part 1. Apple WWDC, published online - https://developer.apple.com/videos/wwdc/2015/, 2015.

[6] Dan Omachi and Anna Tikhonova. What's New in Metal, Part 2. Apple WWDC, published online - https://developer.apple.com/videos/wwdc/2015/, 2015.

[7] Amund Tveit. Swift and Metal GPU Programming on OSX 10.11 / El Capitan. Memkite, published online http://memkite.com/blog/2015/06/10/swift-and-metal-gpu-programming-on-osx-10-11-el-capitan/, 2015.

[8] Min Lin, Qiang Chen, and Shuicheng Yan. Network in network. *CoRR*, abs/1312.4400, 2013.

[9] James Bergstra, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-Farley, and Yoshua Bengio. Theano: a CPU and GPU math expression compiler. In *Proceedings of the Python for Scientific Computing Conference (SciPy)*, June 2010. Oral Presentation.

[10] Amund Tveit. Deep Learning for Speech Recognition. published online http://memkite.com/blog/2015/02/11/deep-learning-for-speech-recognition/, 2015.

[11] Amund Tveit. Deep Learning for Natural Language Processing. published online http://memkite.com/blog/2015/01/29/deep-learning-for-natural-language-processing/, 2015.

[12] Xiang Zhang and Yann LeCun. Text understanding from scratch. *CoRR*, abs/1502.01710, 2015.

[13] Neil Trevett. Vulkan, spir-v and opencl 2.1. online, khronos.org, 2015.

[14] Nicolas Vasilache, Jeff Johnson, Michaël Mathieu, Soumith Chintala, Serkan Piantino, and Yann LeCun. Fast convolutional nets with fbfft: A GPU performance evaluation. *CoRR*, abs/1412.7580, 2014.

[15] Soumith Chintala. convnet-benchmarks. published online https://github.com/soumith/convnet-benchmarks/, 2015.

[16] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. Deep learning with limited numerical precision. *CoRR*, abs/1502.02551, 2015.

[17] Pete Warden. Why are Eight Bits Enough for Deep Neural Networks? published online http://petewarden.com/2015/05/23/why-are-eight-bits-enough-for-deep-neural-networks/, 2015.

[18] Amund Tveit. Example of Sharing Memory Between GPU and CPU with Swift and Metal for iOS8. Memkite, published online http://memkite.com/blog/2014/12/30/example-of-sharing-memory-between-gpu-and-cpu-with-swift-and-metal-for-ios8, 2015.

[19] Amund Tveit. Deep Learning University. published online http://Deeplearning.University, 2015.

[20] Amund Tveit. Deeplearning.University - Bibliographies from Lisa Labs (Yoshua Bengio's Lab). published online http://memkite.com/blog/2015/04/17/deeplearning-university-bibliographies-from-lisa-labs-yoshua-bengios-lab/, 2015.

[21] Apple Inc. Metal Framework Reference. published online https://developer.apple.com/library/ios/documentation/Metal/Reference/MetalFrameworkReference/index.html, 2015.

[22] Apple Inc. Metal Shading Language Guide. published online https://developer.apple.com/library/ios/documentation/Metal/Reference/MetalShadingLanguageGuide/Introduction/Introduction.html, 2015.

[23] Apple Inc. MetalKit Programming Guide. published online https://developer.apple.com/library/ios/documentation/Miscellaneous/Conceptual/MetalProgrammingGuide/Introduction/Introduction.html, 2015.

[24] Apple Inc. Metal Performance Shaders Framework Reference. published online https://developer.apple.com/library/prerelease/ios/documentation/MetalPerformanceShaders/Reference/MPSFrameworkReference/index.html, 2015.

[25] Apple Inc. MetalKit Framework Reference. published online https://developer.apple.com/library/prerelease/ios/documentation/MetalKit/Reference/MTKFrameworkReference/index.html, 2015.