W E B Design

Benchmarking NVIDIA's Jetson Nano for Deep Learning Applications

Engineering Report

Revision A2 • August 02, 2019

W. Ekkehard Blanz

Contents

Revision History

Document Revision	Revision Date	Author	Revision Description
A1	July 31, 2019	W. Ekkehard Blanz	Initial Release
A2	August 02, 2019	W. Ekkehard Blanz	Moved code to github

ABSTRACT: Ever since Arduinos and Raspberry Pis have conquered the world of popular embedded computing, there has been an increasing wide-spread interest in low-cost small-form-factor Systems-on-a-Board, and, following Moore's law, those systems have become ever more powerful. One of the more recent additions in that area was NVIDIA's Jetson Nano. An ARM-based System-on-a-Chip paired with a powerful 128 core graphics processor, targeted specifically at Artificial Intelligence applications. This work reports the results of several benchmark experiments using this small computer and a standard Deep Learning software stack consisting of Keras, TensorFlow, and CUDA, and comparing the results with those obtained on other systems capable of interfacing with local hardware but without graphics processor support. The results must be called at least somewhat surprising.

1 Introduction and Scope

In April of 2019, NVIDIA started shipping its "Jetson Nano" product together with a Developer Kit that makes it palatable for the computer lab, as the kit provides all the peripherals, such as USB 3 connectors and Gigabit Ethernet port, that allow it to connect to other lab equipment. The Jetson Nano is a small from-factor, low-priced System-on-a-Board (SoB), the smallest of its Jetson series devices. Apart from its four-core ARM Cortex-A57 CPU, it contains a Graphics Processing Unit (GPU), which, with 128 CUDA cores, must be called rather powerful for a system this size. It is targeted at applications where AI devices are deployed "at the edge," i.e. off the cloud or powerful compute servers and close to where IoT applications meet actuators and sensors, from "smart cities to robots," as NVIDIA says in their advertising.

While the advertised intent to work at the edge and connect directly to end devices is clearly supported with a Raspberry-Pi-compatible 40-pin GPIO header and support for I^2C, NVIDIA does not make it equally obvious what the intended use for the 128 dedicated GPU cores is. But since they mention AI applications in their advertising, and NVIDIA's GPUs are used ubiquitously in Deep Learning and are widely supported by popular Deep Learning software packages such as TensorFlow, PyTorch, and Keras, it stands to reason that NVIDIA had Deep Learning applications in mind when they were talking about "AI." But Deep Learning usually means tens to hundreds of thousands if not millions of data points and almost equal amounts of trainable parameters, usually requiring a specialized server either in the cloud with an abundance of compute accelerators or at least a powerful local server equipped with a minimum of one of NVIDIA's high-end graphic cards. So can a Deep Learning environment be set up outside of the cloud and without high-end GPU support in general and on a very small system such as the Jetson Nano in particular? And how would one go about that? Which niche exactly can the Jetson Nano play in? Can it be used for the application of pre-trained models in recurring AI tasks only, or can one even reasonably expect to use the power of its small GPU to aid during training of the models as well? If so, which, if any, problem classes and sizes can one tackle with such a small system that is sold for under $ 100.00?

These are the questions that this reports tries to answer. Of course, it would be pointless to compare the Jetson Nano to a high end server with several TITAN RTXs or a TESLA K80. After all, such systems are not at the edge and their price range is orders of magnitude above that of the Jetson Nano. We will therefore compare the Jetson Nano to other platforms that are at the edge, from a simple desktop, to a laptop, down to a Raspberry Pi, and we will set up a functioning Deep Learning environment on all of those systems, including the Jetson Nane. There lies no particular significance in the choice of

the systems used in this report; they are simply the ones that were running some form of Linux and were available for experimentation at the time of writing.

This report is only covering the setup of the Deep Learning environment on the Jetson Nano and the execution of the benchmark experiments and, finally, their results. Any teaching of Deep Learning methods is way outside the scope of this document and we refer the interested reader to pertinent textbooks, such as [2]. We will also not go out of our way to carefully craft sophisticated benchmarks to compare these systems. We have simply chosen some examples out of an introductory book for Deep Learning applications written by François Chollet that teaches the use of Keras, which he also initially authored [1]. It would also go beyond the scope of this report to dive deep into the architectural reasons for particular performance differences. We will only point out those performance differences, give some superficial explanation, and let the reader chose which of his or her applications are best suited for the Jetson Nano.

Lastly, like all computer hardware benchmarks, this one too is not measuring some piece of hardware in isolation, but rather in the context of some software that may or may not take full advantage of the hardware under consideration. In particular in our case where we are mostly interested in performance boosts provided by a GPU, the results will only reflect the performance of the entire system, GPU, CPU, software, and all, not specifically the benefits of the GPU alone.

2 Materials and Methods

2.1 Benchmark Contenders

The systems we use in this benchmark study are, outside of the Jetson Nano, three more or less randomly collected systems of different size and performance. They were selected because they were all "at the edge," ran Linux, and were available to run the benchmark experiments, which took them out of service for quite some time in some cases, not because they could provide additional insight into how well different architectures fair on the benchmark tasks described in Section 2.4. We will refer to these systems by their host name they have on the LAN used for experimentation.

The first system, named "Gandalf," is an old custom-built deskside computer with a six core AMD Phenom 64-bit processor running at 1.7 GHz, and equipped with 8 GB of RAM. The second system, called "Bilbo," is an almost equally old Dell Inspiron 3521 laptop with an Intel Core i3 64-bit processor running at 1.9 GHz and 4 GB of RAM. The third we called "Frodo;" it is a Raspberry Pi 3 B with a Broadcom BCM2837 ARM 7

32-bit processor clocked at 1.2 GHz with 1 GB of RAM. The last one, of course, is the brand new NVIDIA Jetson Nano, named "Boromir" on our LAN, with an ARM Cortex MPCore 64-bit processor running at 1.4 GHz and 4 GB of RAM. This one also has 128 CUDA GPU cores, all other systems do not have any CUDA cores to accelerate the numerically intense calculations. All systems, except Boromir, are using single precision 32 bit floating point numbers for the benchmark tasks; Boromir uses only half precision, i.e. 16 bits, as this is all its CUDA cores can handle. Table 1 summarizes the systems used for benchmarking and their configuration.

Host name	Gandalf	Bilbo	Frodo	Boromir
Brand and Make	Custom Built	Dell Inspiron 3521	Raspberry Pi 3 B	NVIDIA Jetson Nano Developer Kit
Processor	AMD Phenom	Intel Core i3	Broadcom BCM2837	ARM Cortex-A57 MP Core
Architecture	x86 64-bits	x86 64-bits	ARM 7 32-bits	ARM 8 64-bits
Memory [GB]	8	4	1	4
CPU Cores	6	4	4	4
CPU Clock [GHz]	1.7	1.9	1.2	1.4
GPU Cores	0	0	0	128
Float bits	32	32	32	16
OS	Kubuntu 18.05	Kubuntu 18.04	Raspbian 9.9	Ubuntu 18.04
Kernel Version	4.15.0-55-gen.	4.15.0-55-gen.	4.19.42-v7+	4.9.140-tegra

Table 1: Systems competing in the benchmark study.

2.2 Setting Up the Jetson Nano Hardware

NVIDIA sells the Jetson Nano together with a Developer Kit that houses the Jetson Nano and provides 4 USB 3 connectors, a micro USB connector to supply power to the board, a Gigabit Ethernet connector, an HDMI 2.0 and an eDP 1.4 connector to attach up to two monitors, a camera connector that allows to attach a Raspberry-Pi-style camera, as well as a 40-pin GPIO connector, which again is compatible with that of the Raspberry Pi 3 and above. The Developer Kit also has a slot for a micro SD card and a 5.5×2.1 mm barrel jack for alternative 5 V power supply, which can be selected via a jumper. Neither micro SD card nor power supply nor jumper are provided and need to be purchased separately. The same is true for an optional fan and its mounting screws. The Developer

Kit comes in a box that doubles as a stand, which can be used to safely place the board in and protect it from accidental shorts that could occur if the board was placed directly on the lab table, thus eliminating the strong need for additional housing (see Figure 1).

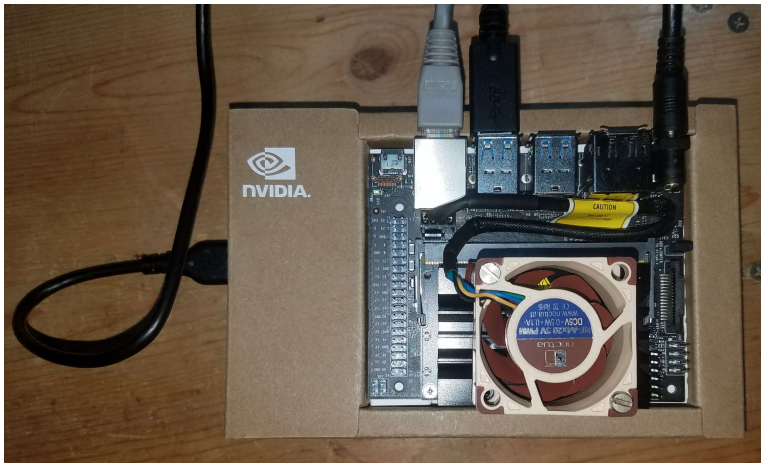

Figure 1: Top view of the Jetson Nano placed in its box with fan mounted to the heat sink and Ethernet, USB drive, and power connected.

First, the SD card image needs to be downloaded from NVIDIA's web page [3] and copied to a micro SD card. This step is fairly straightforward and works exactly as described on NVIDIA's web page.

The next thing one needs to do is to provide sufficient power to the board. The Jetson Nano has two pre-configured power modes, 5W and MAXN, and the kernel image comes preconfigured for MAXN mode, which will let the unit draw about 4 A versus the 1 A it will draw in 5W mode. One can switch between the modes via the command line tool `nvpmodel`. Note that the mentioned currents do not include any power that USB peripherals or an optional fan might require and that the USB 3 connectors are rated for up to 900 mA each—the power requirements for the fan with only 100 mA are almost negligible in this context. When the board is powered via the micro-USB connector, however, it can only draw up to 2 A over that connector. If that is not enough, and it will not be enough for many applications and/or when peripherals are attached via the USB 3 ports, the Jetson Nano will simply shut down. Therefore, a jumper should be installed at J48 (see Figure 2), and a sufficiently strong 5 V power supply should be connected to the barrel jack. We use a ALITOVE 5 V 15 A AC to DC Power Supply for our experiments.

Then, it may be a good idea to mount a 40 mm fan on top of the heat sink. It will draw a little extra power, but it will prevent the system from down-clocking in heavy

Figure 2: Jumper J48 placed on Jetson Nano Developer Kit board.

computational load situations. We use the NVIDIA-recommended Noctua NF-A4x20 5 V PWM, Premium Quiet Fan, which we connected to the PWM connector that is provided for this purpose (see Figure 3). One word of caution: the mounting holes in the heat sink of the Jetson Nano are not threaded. Therefore, either self-tapping M3 screws are to be used, as suggested by NVIDIA, or the holes need to be tapped before the fan can be installed with regular M3 machine screws.

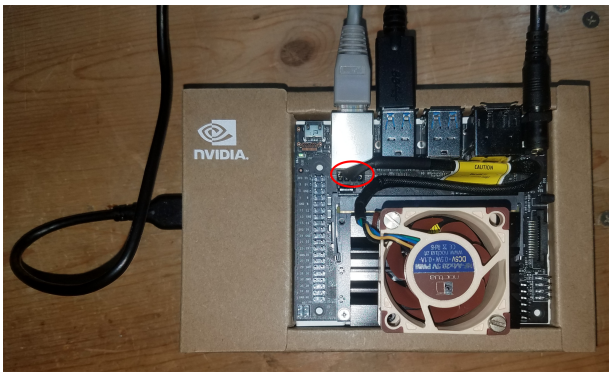

Figure 3: Fan connected to the PWM connector.

Further, even though many people do just that, it is not advisable to run any SoB, such as the Jetson Nano[1], from a micro SD card. These cards are much slower than USB drives, and they are not designed for the frequent access patterns associated with mass storage devices in Linux computers—they are *not* SSD drives! Therefore, it is a good idea

[1]Or the Raspberry Pi we used in the experiments, for that matter.

to use an external USB drive, either mechanical or SSD, connected to the Jetson Nano. Unfortunately, the kernel that was loaded onto the micro SD card does not support USB 3 drives. Therefore, a different kernel source tree needs to be downloaded. A good source for that is `JetsonHacksNano/rootonUSB` on github. This repository can be cloned via git or just downloaded as a zip file. It needs to be downloaded on the micro SD card of the Jetson Nano and then compiled there. This is made very simple with a shell script that is packaged with the kernel source that can be executed once we have navigated to the directory of the repository as

```
$ sudo ./buildKernel.sh
```

This will place the kernel image in /boot/Image, which means that if the Jetson Nano is rebooted, it will boot using that new kernel image.

Next we need to prepare the USB drive for use as a Linux disk. In our case, we have used an inexpensive mechanical 1 TB external USB drive, specifically the Toshiba Canvio Basic with a 480 Mb/s data rate. To keep the external USB drive together with the Jetson Nano as a unit, we have cut a slot in its cardboard box, big enough to accommodate the external USB drive (see Figure 4).

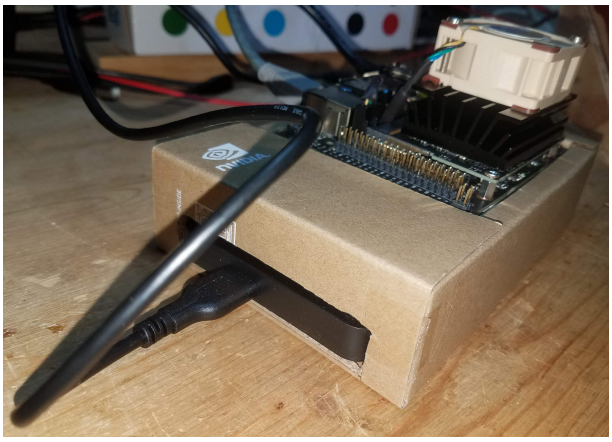

Figure 4: Side view of the Jetson Nano box showing slot for external USB drive.

We will use `gparted` to partition and format the USB drive, which therefore also needs to be installed on the Jetson Nano (see Section 2.3). We use about 64 GB for the root file system, 8 GB for swap space, and the rest of the disk space for `/home` and format the disk for the Ext4 file system.

Then the root file system needs to be copied over from the micro SD card to the USB drive. The `rootonUSB` repository has a convenience script for that, which we will use, supplying the volume name we gave to our USB drive in `gparted` as an argument

```
$ sudo ./copyRootToUSB.sh -v <volume name>
```

Once the kernel is compiled and placed on the micro SD card as well as the USB drive, we can "pivot the root," i.e. tell the Jetson Nano to just use the micro SD card as a bootloader and then switch to the USB drive. We do this by editing the file `extlinux.conf` in `/boot/extlinux` to look something like this

```
TIMEOUT 30
DEFAULT primary

MENU TITLE p3450-porg eMMC boot options

LABEL primary
      MENU LABEL primary kernel
      LINUX /boot/Image
      INITRD /boot/initrd
      APPEND ${cbootargs} rootfstype=ext4 root=/dev/sda1 rw rootwait

LABEL sdcard
      MENU LABEL primary kernel
      LINUX /boot/Image
      INITRD /boot/initrd
      APPEND ${cbootargs} rootfstype=ext4 root=/dev/mmcblk0p1 rw rootwait
```

With these changes, the Jetson Nano should be using the USB drive as a root file system after reboot.

For all benchmark applications that require access to files residing on the local disk during training or test, we store all such files on the external USB drive of the Jetson Nano and remotely mount this drive at the other systems and access the data on it via the Gigabit LAN from the other systems. This is to assure that the relatively slow disk does not unfairly disadvantage the Jetson Nano. For all benchmark experiments, we run the Jetson Nano as a "headless server," i.e. without keyboard, mouse, or monitor and use `xfreerdp` to log into it from other systems on the LAN, which requires the installation of `xrdp` (including enabling it using `systemctl`) on the Jetson Nano.

All Jetson Nano paraphernalia used in this study outside of standard cables, screws and jumpers, which are considered standard lab stock, were sourced through Amazon.com.

2.3 Software Setup

Of course, there are many different application domains one can imagine to showcase a dedicated GPU, amongst which graphics intense applications and visualizations certainly play a dominant role, and the software packaged with the Jetson Nano installation kernel contains several interesting graphical applications for that purpose. However, our focus is on the AI application niche that was targeted by NVIDIA for the Jetson Nano and its GPU, and the AI applications that stress the capabilities of any compute engine the furthest are certainly Deep Learning applications [2], [4]. That's why we focus on Deep Learning applications for this study exclusively.

The Deep Learning community has widely embraced Python with Jupyter notebooks as a working environment for online experimentation, in part because with that environment it makes no difference whether one is working in the cloud or locally. In that working environment, the Python package Keras is our tool of choice for Deep Learning experiments, mostly because of its ease of use. Keras works with several backends, TensorFlow, Theano, and CNTK, from which we selected TensorFlow as our backend of choice. TenesorFlow, in turn, uses several compute engine backends for standard numerical tasks, such as BLAS and LAPACK for computations that are performed on the regular CPU(s), as well as CUDA and cuDNN to unleash the power of GPUs to accelerate its most compute intensive tasks in cases where the system is equipped with appropriate GPUs.

To prepare for the experiments, some additional Python packages need to be installed on top of a regular Python 3 environment, which should probably include Jupyter, and, of course, pip. However, especially on the Jetson Nano, chances are that packages need to be built and therefore, the Unix build tools need to be installed in case they are not already present. All packages from distros on the Jetson Nano are downloaded using

```
$ sudo apt install <package name>
```

The build packages we get from the distro are

- build-essential
- cmake
- unzip
- pkg-config

In addition, as mentioned above, we need the numeric libraries BLAS and LAPACK also from distro

- libopenblas-dev
- liblapack-dev

Then we need to start installing Python packages specifically needed for Deep Learning. We strongly recommend to choose the packages from the particular distros wherever they are available over the generic Python packages. The Python packages we prefer to get from our distro are

- python3-setuptools
- python3-all-dev
- python3-numpy
- python3-scipy
- python3-matplotlib
- python3-libyaml-dev
- python3-cpuinfo
- python3-psutil

The last two packages are only needed to list system information in the log files.

Where Python packages cannot be obtained from distro, which, at the time of writing, is still often the case on the Jetson Nano, they need to be installed via pip, i.e.

```
$ sudo pip3 -H install <package name>
```

obviously here the package names don't use the "python3-" prefix[2]. On the Jetson Nano, pip will often need to compile the sources to build the packages, which may take a while.

Next, on the Jetson Nano and all machines with GPU support, we also need CUDA, and, optionally, cuDNN. Then we need to install TensorFlow, and, finally, Keras. We start with the packages that we only need where GPUs are present; on benchmark contenders without GPU, as is the case in our study in all systems outside of the Jetson Nano, we can skip the installation of CUDA and cuDNN. On the Jetson Nano, CUDA 10.0 comes preinstalled, but cuDNN was not available for this architecture at the time of writing. For instructions how to install CUDA and cuDNN on any other system with GPU support, we refer to the steps outlined in [1], which are very easy to follow.

Now it is time to install TensorFlow. Which package exactly to install depends on whether or not the system it is to be installed on has GPU support or not. Either way,

[2]We use Python 3 exclusively—never Python 2.

TensorFlow needs to be installed via pip, and the package name is either tensorflow or tensorflow-gpu, depending on whether or not GPU support is available.

We hit one minor roadblock when we installed TensorFlow on Gandalf, however. The old AMD Phenom has no AVX or SSE 4.2 instructions, which TensorFlow versions later than 1.5 requires. This forced us to roll back to TensorFlow version 1.5.0 on Gandalf only.

Lastly, we need to install Keras. This is very straightforward, as it should be installed directly with pip, and it installed without any problems on all of our systems.

2.4 The Benchmark Applications

We take all of our applications straight out of Chollet's book [1]. To facilitate experimentation, we wrote a small wrapper that calls the individual benchmark applications and logs their results in a log file that identifies not only the name of the application but also the parameters of the system that executed the application, such that the results are easily attributed to the system they were achieved with.

Table 2 shows the training and test samples sizes as well as the epochs and batch sizes used for the experiments. Table 3 shows the net architectures used for the same experiments. They are all the same as in [1].

	Train. Samp.	Test Samp.	Epochs	Batch Size
MNIST Digits 1D	60000	10000	5	128
IMDB	25000	25000	4	512
Reuters	8982	2246	9	512
MNIST Digits 2D	60000	10000	5	64
Dogs vs Cats	2000	1000	15	20
IMDB Embedded	25000	25000	10	32
MPI Weather	200000	120406	10	128
MPI Weather Conv.	200001	120407	10	128

Table 2: Benchmark tasks parameters.

In the following, we will give only a very brief description of the different experiments. Since teaching the idiosyncrasies of Deep Learning methods, especially for more than one-dimensional datasets is beyond the scope of this report, we refer the reader interested in any more details to [1] for the particular experiments and to [2] for a more in-depth

	Input Shape	Net Architecture	Parameters
MNIST Digits 1D	(784,)	(512, 10)	407050
IMDB	(10000,)	(16, 16, 1)	160305
Reuters	(10000,)	(64, 64, 46)	647214
MNIST Digits 2D	(28, 28, 1)	((26,26,32), (11,11,64), (3,3,64), 64, 10)	93322
Dogs vs Cats	(150, 150, 3)	((148,148,32), (72,72,64), (34,34,128), (15,15,128), (7,7,128), 512, 1)	3453121
IMDB Embedded	(50,)	((50,8), 400, 1)	80401
MPI Weather	(21,)	(32, 1)	5217
MPI Weather Conv.	(21,)	(32, 32, 32, 1)	14817

Table 3: Benchmark tasks network architectures.

coverage of Deep Learning networks. The code for all of these benchmarking applications can be found in github under `Ekkehard/DL-Benchmarks`.

2.4.1 MNIST Digits 1D This experiment uses a set of 70,000 images of handwritten digits, each stored as a 28 by 28 pixels gray-value image. The dataset comes pre-packaged with Keras and is split into 60,000 images as training set and 10,000 images as test set. The images have originally been assembled by NIST but have later been modified to better mix the sources of the digits in training and test set; they were also normalized in size and gray level intensity (hence the "M" in MNIST). In this experiment the pixels of the images are re-arranged to form simple vectors for a very easy one-dimensional Deep Learning experiment.

Here we use a net with only one hidden layer with 512 nodes and an output layer with (obviously) 10 nodes, one for each category. We train with 5 epochs and a batch size of 128.

2.4.2 IMDB Here we use 50,000 highly polarized reviews of movies from the Internet Movie DataBase which we want to classify into positive and negative reviews. Again, the data comes pre-packaged with Keras and is split into 25,000 training and 25,000 test samples with the same number of positive and negative reviews in each set. To make these texts palatable for neural networks, they need to be converted into vectors of real numbers. To do this, the texts are first converted into lists of integers where each integer represents a particular word but where we use only the 10,000 most frequently occurring words in the texts—the others simply don't encode at all. Then, following [1], we engage

in a rather wasteful encoding scheme that also eliminates the information stemming from the word sequence whereby we assign to each review a 10,000-dimensional vector that contains all zeros except for the positions that correspond to a word that is contained in the given review; at those positions they contain a one, even if the word occurred more than once.

We us a net with two hidden layers with 16 nodes each, and since this is a binary classification task only one node in the output layer. We train with 4 epochs and a batch size of 512.

2.4.3 Reuters In this experiment we use a database of 11,228 newswire reports from Reuters with wires pertaining to 46 different categories of news. This dataset too comes with Keras and is split into 8982 training samples and 2246 test samples. To convert the text into vectors of real numbers, we use the same method as described in Section 2.4.2.

We train a net with two hidden layers with 64 nodes each and use one node for each of the 46 categories in the output layer. We train with 9 epochs and a batch size of 512.

2.4.4 MNIST Digits 2D In this experiment we use the same dataset as described in Section 2.4.1. However, this time we we will feed the images as two-dimensional images directly to the net and will not convert them into one-dimensional vectors first. To allow that, we will use what is called a convnet, i.e. a neural network that acts like a convolution kernel in image processing. Our convnet will start out with a 3 by 3 convolution kernel producing 32 "features" as the first hidden layer, still arranged in a two-dimensional pattern. We will then reduce the size of each of the 32 feature matrices by a 2 by 2 maximum non-linear filter, leaving us with 32 feature matrices of size 13 by 13[3]. Then, in the next hidden layer, we apply another convnet using a 3 by 3 window, this time producing 64 features. Again, we will reduce the size with a 2 by 2 maximum non-linear filter and apply yet another hidden convnet layer, producing 64 features with 3 by 3 kernels. We will finally "flatten" the "feature images" to regular vectors, apply a final hidden layer with 64 nodes feeding into the output layer with 10 nodes for the 10 categories of the digits.

We train with 5 epochs and a batch size of 64.

2.4.5 Dogs vs Cats Here again we use datasets of images, in this case even three-dimensional images since they are color images. This dataset did not come with Keras

[3]The convnet reduces the image size already as data points at the borders are not defined.

and had to be downloaded from Kaggle (`www.kaggle.com/c/dogs-vs-cats/data`). The full dataset contains 25,000 images of dogs and cats of which we randomly selected 2,000 images for training and 1,000 images for testing to reduce the problem size and hence the computation time to a more manageable level and since we were not interested in achieving the best possible classification result but rather in performance differences of different systems. We also instructed Keras to reshape all images to 150 by 150 pixels.

For the net architecture we chose again convnets as in Section 2.4.4 which we interspersed with 2 by 2 maximum filters. All in all, we used 5 hidden layers reducing the image or "feature map" size from 148 by 148 to 7 by 7 in four steps, but, at the same time, increasing the number of features from 32 to 128 and finally to 512. Since this again is a binary classification task, the output layer has only one node.

We train with 15 epochs and a batch size of 20.

2.4.6 IMDB Embedded This experiment uses the same dataset of movie reviews as described in Section 2.4.2. However, this time we use a word embedding scheme to encode the text into vectors of real numbers instead of a single vector with certain elements "switched on." This has the advantage of being overall less wasteful than what we have seen in Sections 2.4.2 and 2.4.3 even though it comes at the expense of an additional dimension for the embedding vector. In other words, at each training step, the subsequent layer will be presented with a number of embedding vectors that equals however many words in the text we want to look at, which in our case is only the first 50 words. Word embedding is done at the expense of higher computational effort when compared to Section 2.4.2, since this mapping will be part of what is learned from the training set and not programmed *a priori*.

In total our net consists only of the hidden embedding layer, embedding every text integer into an 8-dimensional vector and an output layer with a single node. We train with 10 epochs and a batch size of 32.

2.4.7 MPI Weather This dataset was downloaded from the Max Planck Institute for Biogeochemistry in Jena, Germany. We downloaded their weather data from 2009 to 2016 and tried to predict next days temperature from all the 21 weather-relevant measurements they collected every ten minutes over the last 5 days. Since this is a regression task and not a classification task, we have no classification accuracy to report. Since we are dealing with time series, we use a recurrent neural network (RNN) for this task to acknowledge the fact that successive measurement vectors are not independent of each other and that there is valuable information in the numerical progression from one measurement vector

to the subsequent ones. We will sub-sample the data every 6th step thus dealing with only one measurement vector per hour. For the prediction of the temperature 24 hours ahead, we will use the data from the last 5 days or 720 total observation vectors. Therefore, for each training step, we will present the data of five days worth of measurements as measurement vector and the data 24 hours from "now" as the target. We use a Gated Recurring Unit (GRU) with 32 nodes as first hidden layer and a single node output layer for the regression task.

We train with 10 epochs and a batch size of 128.

2.4.8 MPI Weather Convnet This last experiment uses the same dataset of weather data as described in Section 2.4.7. However, we now also apply the concept of convnets, albeit this time one-dimensional ones, not the two-dimensional ones we used for images. We use two hidden one-dimensional convolution layers with 5 taps each, both producing 32 "features" with one non-linear maximum finding unit with 3 taps between them. We then use the same GRU layer as described in Section 2.4.7, followed by a single node output layer.

Again, we train with 10 epochs and a batch size of 128.

3 Experimental Results

Here we will present the results of the benchmark experiments outlined in Section 2.4. Without considering Boromir, what we would expect is a superior performance of Gandalf and Bilbo, based on the number of cores, processor speed and bit-depth of the architecture. Frodo would come in last with a shallow 32-bit architecture, a slow clock speed of 1.2 GHz, and it may not even succeed at all experiments with its measly 1 GB of RAM (see Table 1). The big unknown, obviously, is Boromir. Without its GPU, it would be expected to rank somewhere between Frodo and Bilbo or Gandalf. But, as mentioned, that is without its GPU. The big question is how much of a performance boost we can expect from Boromir's GPU, and if we do see such a performance increase, in which application areas will they be most prominent.

To gain a comprehensive and yet compact representation of the experimental results, we have collected them in three different tables, one to show the time it took to train the (for all systems identical) net architectures, one to show the time it took to execute the test set, and one where we listed the classification accuracy on the test set for all systems. We included this last table to see whether the reduced bit depth of the floating point

numbers we used on Boromir impacted the accuracy of the results. All numbers were obtained by running each experiment five times and then taking the median of those five values. We chose the median over the average because Linux is not a real-time operating system and we would expect some outliers when the CPU was occupied doing some system tasks (which we did in fact observe).

First, Table 4 shows the training times for all the eight different experiments on the four different computer systems. There is no entry for the IMDB experiment run on Frodo as it ran out of memory on this experiment.

	Training Time [s]			
	Gandalf	Bilbo	Frodo	Boromir
MNIST Digits 1D	33.7	29.3	255.8	63.2
IMDB	5.9	8.1	-	24.9
Reuters	8.9	9.7	85.4	21.1
MNIST Digits 2D	250.8	293.0	2378.1	263.5
Dogs vs Cats	1463.5	2147.9	15105.3	550.2
IMDB Embedded	29.3	29.6	118.3	125.4
MPI Weather	1154.8	901.2	6634.0	6570.4
MPI Weather Conv.	730.5	622.7	4593.8	3027.8

Table 4: Training times.

The first experiment with handwritten characters treated as one-dimensional vectors shows the expected result without any noticeable boost from Boromir's GPU. The same is true for the experiment with the classification of IMDB movie reviews and the Reuter news wires.

The first small surprise might be the classification of the MNIST database when we treat the images of the handwritten characters no longer as one-dimensional vectors but as two-dimensional images. The performance of Frodo is, as expected, very poor compared to that of Gandalf or Bilbo, but in this case, Boromir's performance is very comparable to that of the thus far leading contenders Gandalf and Bilbo and, in fact, beats Bilbo.

The biggest surprise by far, however, stems from the "dogs vs cats" experiment. First, it appears as if Gandalf could, for the first time, take advantage of its 6 cores over the 4 cores of Bilbo[4]. The performance ratio of Gandalf vs Bilbo is roughly that of 6 vs 4 cores directly. The performance of Frodo is poor for this huge experiment, as expected

[4]We did notice good use of all available cores while running the experiments.

for such a small system. However, based on what we have seen so far, the performance of Boromir must be called jaw dropping! 550.2 s vs 1463.5 s on Gandalf! That is almost a factor of three of performance improvement over the much bigger system Gandalf and about a factor of 4 improvement over Bilbo, which we consider quite remarkable.

The results of the experiment with the embedding layer to tackle the Internet Movie Database classification task is, at the other hand, also surprising, just in the opposite direction. The performance of Boromir is right in the area of Frodo, indicating that Boromir could not take any advantage of its GPU at all for this experiment.

The results of the experiments using the MPI weather time series is no different from the IMDB experiment with the embedding layer. The performances of Frodo and Boromir are virtually identical with a slight advantage for Boromir only when using the convnet, but still more than three times worse than Gandalf.

Next we will have a look at the times it took to run the test sets as outlined in Table 5

	Test Time [s]			
	Gandalf	Bilbo	Frodo	Boromir
MNIST Digits 1D	0.8	0.7	7.1	1.7
IMDB	3.3	2.1	-	8.0
Reuters	0.8	0.3	1.6	0.8
MNIST Digits 2D	3.8	3.1	26.8	5.8
Dogs vs Cats	19.9	64.2	165.6	23.9
IMDB Embedded	1.1	0.8	2.9	3.9
MPI Weather	54.8	47.0	374.3	331.6
MPI Weather Conv.	37.8	38.8	319.4	134.4

Table 5: Test times.

After having seen the results in Table 4, these results come as no big surprise. The performance improvement of Boromir's GPU over the other contenders in the "Dogs vs Cats" experiment is not as pronounced as it was during training, and "only" a little more than a factor of 2 better than Bilbo and even worse than Gandalf, where again Gandalf is showing a benefit from its larger number of cores over Bilbo. The other results are in the range where we would have expected them to be.

Lastly, we consult Table 6 to see if the reduced bit depth of Boromir's floating point number scheme has any negative impact on the classification accuracy. We note that the experiments using the MPI weather data show no classification accuracy as they were no classification tasks.

	Classification Accuracy on Test Data			
	Gandalf	Bilbo	Frodo	Boromir
MNIST Digits 1D	98.0%	97.9%	97.9%	98.0%
IMDB	88.3%	88.3%	-	88.3%
Reuters	79.0%	78.9%	79.1%	79.0%
MNIST Digits 2D	99.1%	99.1%	99.1%	99.2%
Dogs vs Cats	72.3%	72.0%	72.1%	71.9%
IMDB Embedded	81.9%	81.9%	81.9%	80.9%

Table 6: Classification accuracy on test data.

In short, there seems to be no systemic performance degradation due to the reduced bit depth to half precision floating point when using Boromir. Most of the variations in the classification performance are assumed to be attributable to the random initialization of the nets.

4 Discussion

We set out to investigate whether the recently released Jetson Nano Developer Kit from NVIDIA is just another toy in our SoB collection or whether it might be capable of doing more serious work in the context of Deep Learning, not only for running pre-trained nets, but also during training. We therefore ran the majority of the experiments described in [1] on this board and on several other systems "on the edge," including a deskside computer, a laptop, and a Raspberry Pi. We note that this benchmark not only measures the isolated performance of the Jetson Nano but also that of the entire Deep Learning software stack including Keras, TensorFlow, and CUDA. In almost all cases there were no big surprises: The Jetson Nano performed where we might have expected it to perform without its GPU—somewhere between the Raspberry Pi and the complete systems.

With one notable exception.

When we ran computer vision classification experiments, the Jetson Nano either performed at par with the much bigger systems for the smaller 28 by 28 pixels image size problem, or outperformed them by a good margin for the bigger 150 by 150 pixels image size problem.

The performance of the Jetson Nano in all but the computer vision problems are

easily explained by the architecture of its CPU and its computational "resources" such as number of cores, CPU frequency, and installed memory. Therefore, the performance boost when solving computer vision classification problems can only be attributed to its GPU and its efficient use by the software we used. This is good and bad news. The good news is obviously that there is definitely something to be gained from a 128 core GPU in systems as small as the Jetson Nano even when compared to much bigger conventional computer systems. And this not only holds for applying pre-trained nets to new data, but also, and primarily, for training Deep Learning networks. The bad news is that there seems to be no noticeable benefit from this GPU in all of the other experiments. The open question is why that is.

Without diving deep into Keras and/or TensorFlow and CUDA, we can only speculate, as carefully examining all of that code would be way beyond the scope of this report. However, from what we have seen in these experiments, it appears as if the Deep Learning software stack we used did not take full advantage of the 128 GPU cores in most application areas but that the GPU is well capable of giving us advantages over conventional non-GPU systems when used properly. We have therefore reasons to hope that additional performance gains should be possible if one were to better exploit the benefits of GPUs in areas outside of computer vision. Therefore, diving deeper into software areas outside of computer vision, in particular in the area of time series analysis with convnets, shall be an area of our future investigations.

A Acronyms and Abbreviations

While important but lesser known acronyms are defined at first use, this appendix lists all acronyms and non-standard-English abbreviations used in this report and their respective meaning.

AI Artificial Intelligence

ARM Advanced RISC Machine (instruction set architecture)

AVX Advanced Vector eXtensions (x86 instruction set extension)

BLAS Basic Linear Algebra Subprograms (software library)

CNTK Cognitive Network ToolKit (now Microsoft Cognitive Toolkit)

convnet Convolution Network (NN performing a convolution task)

CPU Central Processing Unit

CUDA Compute Unified Device Architecture (NVIDIA GPU architecture and library)

cuDNN CUDA Deep Neural Network (NVIDIA GPU software library)

eDP embedded Display Port (lesser used alternative to HDMI for monitors)

GPIO General Purpose Input and Output (signal and power supply pins)

GPU Graphics Processing Unit

GRU Gated Recurrent Unit (one way of implementing RNNs)

HDMI High Definition Multimedia Interface

IMDB Internet Movie Database (database containing movie reviews)

IoT Internet of Things

I^2C Inter-Integrated Circuit (electronic communication protocol)

LAN Local Area Network

LAPACK Linear Algebra PACKage (software library)

MNIST Modified NIST (modification refers to database of handwritten characters)

MPI Max Planck Institute

NIST National Institute of Standards and Technology

NN Neural Network

PWM Pulse-Width Modulation (here: converts digital output to analog fan speed)

RISC Reduced Instruction Set Computer (reduced cycles-per-instruction architecture)

RNN Recurrent Neural Network (NN with "state")

SD Secure Digital (memory card format for portable devices)

SoB System on a Board

SoC System on a Chip

SSE Streaming SIMD Extensions (x86 instruction set extension)

SIMD Single Instruction, Multiple Data (computer architecture paradigm)

SSD Solid-State Drive (electronic storage device without moving parts)

USB Universal Serial Bus

x86 family of instruction set architectures based on Intel's 8086 chip

References

[1] François Chollet. *Deep Learning with Python*. Manning Publications Co., Shelter Island, New York, 2018.

[2] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. The MIT Press, Cambridge, Massachusetts, 2016.

[3] NVIDIA. Jetson nano homepage. `https://developer.nvidia.com/embedded/learn/get-started-jetson-nano-devkit`.

[4] Sebastian Raschka. *Python Machine Learning*. Packt Publishing Ltd., Birmingham, United Kingdom, 2016.