GPU Based Text Analytics

Technical Report

Karge, André
andre.karge@uni-weimar.de

Vogler, Benedikt S.
benedikt.vogler@uni-weimar.de

WS 2016/2017

Contents

1 Introduction

This is the documentation for the Project "GPU Based Text Analytics" of the Webis group. In this project we installed, configured and tested a new deep learning cluster for the group. The second part was to use the new cluster with deep learning software to get familiar with the use and detect issues. For that we first tested common deep learning applications and then tested ourselves on deep learning by writing a simple image classifier. Furthermore we started to prepare a sequence to sequence model which should learn the behaviour of the wikimedia markup parser.

2 GPU Cluster Setup

2.1 Used Hardware

At the beginning of the project we had to set up the new webis hardware. The hardware consisted of three Supermicro SuperServer 4028GR-TRT and a storage server SuperMicro-6048R-E1CR60N

SuperMicro SuperServer

- 2x Intel Xeon
- 1,5 TB RAM
- 8x Nvidia GTX 1080

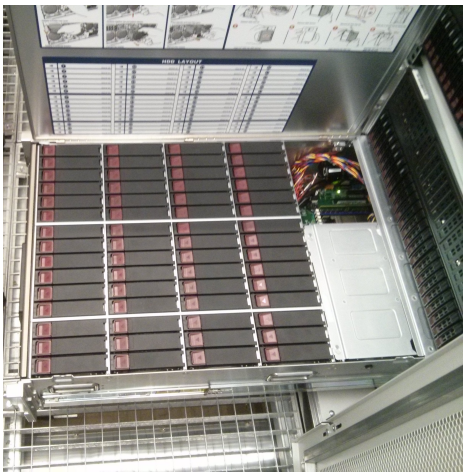
(a) view on motherboard (b) close up of nvidia gtx 1080

Figure 1: One instance of a SuperMicro SuperServer

SuperMicro-6048R-E1CR60N

- 60 x 4TB HDDs = 240 TB
- 20 cores at 2.2 GHz
- 0.1 TB RAM

The machines were mounted into empty racks in the server room beneath the Digital Bauhaus Lab. We had some issues concerning the cable arms: The server mounting rails were too short which is why we first could not mount the cable arms in the back of the machines. The operating system we installed was Ubuntu Server 16.04 LTS. After the installation of the operating system via USB-drive we made sure that all necessary drivers and dependencies were at hand in order to work with TensorFlow. The TensorFlow GPU version required the latest Nvidia GPU driver, CUDA and CuDNN. CuDNN is a library developed by Nvidia for faster deep neural network applications. It has been required to



(a) case opened, showing HDD slots

Figure 2: The SuperMicro-6048R-E1CR60N

register at Nvidia's website in order to get the installation binaries. Beside TensorFlow we provided installations of deep learning libraries like Keras and Theano. To provide a consistent installation on each server we wrote an install script which automates most parts of the installation process. The parts of the script are:

- all pre-required packages (e.g. python3)
- Nvidia proprietary driver
- cuda
- check_mk agent
- GPU plugin for check_mk
- daemon to prevent unload of GPU driver
- cuDNN
- TensorFlow
- other deep learning libraries
- benchmark tools

2.2 Hardware Installation

In each cluster machine there are eight 1TB SSD drives installed. The first drive was declared as system drive with root file system, boot sector and swap. The

partitions were created with the logical volume manager (lvm) which enables us to resize the partitions afterward as we like. The other drives were organised as a RAID. We choose RAID-0[1] since we didn't need redundancy but only the space. The mount point on every machine is at */mnt/raid/*. Instructions on how to set up a RAID0 bundle are in the README file of the project repository. The three cluster machines were named gammaweb01, gammaweb02, gammaweb03 and the storage server was named webis20.

2.3 Software Installation

2.3.1 CheckMK

As for every computer of the webis group the software *check_MK*[2] is used, to monitor the system status. *check_MK* sends periodically requests to a server to obtain and save various status information. The GPU information for *check_MK* can be read after installing a plugin. For Nvidia GPUs it is simple to extract all necessary information by using the provided command nvidia-smi. This command simply prints all information of all installed cards into the console. To use this information we had to use a script in order to prepare it to be understood by *check_MK*. We used a script by the Technische Universität Kaiserslautern. It contained a bug, which had been corrected by us and the authors were informed about the bug which they corrected. Every time the script was called, the graphics card driver was loaded and directly unloaded due to inactivity. This led to big resource utilization on the five-minutes checks. For reduction of this workload a system daemon helps to prevent the unloading of the graphics card driver.

2.3.2 TensorFlow and CISE

TensorFlow is an open sourced library developed by the Google Brain Team within Google's Machine Intelligence research organization[1]. At the core of TensorFlow data flow graphs are used. In praxis, this means that you first build a graph with operations in the nodes and then send the data though the graph which computes it. Nodes can be assigned to devices (CPU or GPU or even across machines) with one API.

It turned out that TensorFlow binaries obtained through the Python Package Index (pip) installer do not support CPU instruction set extensions (CISE), which are available on our system. A CISE allows specific multiple calculations in one single calculation step, therefore improving speed. Since TensorFlow v.1.0 the program throws warnings on the launch when extensions are available on the system but they are not included in the binary:

```
W tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow library wasn't
    ↪  compiled to use AVX instructions, but these are available on your
    ↪ machine and could speed up CPU computations.
```

[1] https://en.wikipedia.org/wiki/Standard_RAID_levels#RAID_0

[2] https://mathias-kettner.de/check_mk.html

These CISE can be made available by compiling TensorFlow with some flags ("-msse4.2 -mavx -mavx2 -mfma -mfpmath=both"). Setting the flags in the configuration file before compilation removes some of the warnings in runtime, however some flags remain.

The flags noted above should automatically be used when using just the flag "-march=native" in the compilation configuration, however this did not remove any warnings.

According to an accepted question on stackoverflow[3] compiling with the following flags should compile it with every CISE:

```
bazel build -c opt --copt=-mavx --copt=-mavx2 --copt=-mfma --copt=-mfpmath=both
    ↪ --copt=-msse4.2 --config=cuda -k //tensorflow/tools/pip_package:
    ↪ build_pip_package
```

A first try failed, but after making sure that every other previously installed TensorFlow version was uninstalled every warning disappeared. Installed versions can be checked with the shell command `pip3 list | grep tensorflow`. The self compiled GPU version lists as *tensorflow* while the downloaded version is named *tensorflow-gpu*. The self-compiled version can be further tailored to our system. TensorFlow was compiled with CUDA support, jemalloc and python3. Included support for the GTX 1080 in our servers only needed the CUDA compute capability for 6.1[4]. A speed comparison showed no significant differences in speed between the self compiled version and the version obtained from Google. This is not surprising because we expect that only the CPU computations become faster. Due to our experiences with the training of models with the GPUs we assume that the bottleneck is in the GPU computation.

The self compiled binaries were installed on each of the three servers.

2.4 Tensorboard

TensorFlow includes a service called Tensorboard to view the models (the graphs) which result of the execution of your code. Tensorboard is a web client which visualizes (Fig. 3) stored training data of a specified folder. It can show progress during training in diagrams. By creating a subfolder for each training with different parameters the results can be compared easily. In the overview a list appears containing the names of all subfolders. A search function helps if the amount of trainings gets too big.

2.5 Workflow

Programming a neural net has the same workflow as traditional programming. You write code, optionally compile, and then test your program. To get access to the code on a server we used *sshfs* to have an updated copy of the working directory on local machines. Because most of the time we worked remotely on

[3] `http://stackoverflow.com/questions/41293077/how-to-compile-tensorflow-with-sse4-2-and-avx-instructions?noredirect=1&lq=1`

[4] `https://developer.nvidia.com/cuda-gpus`

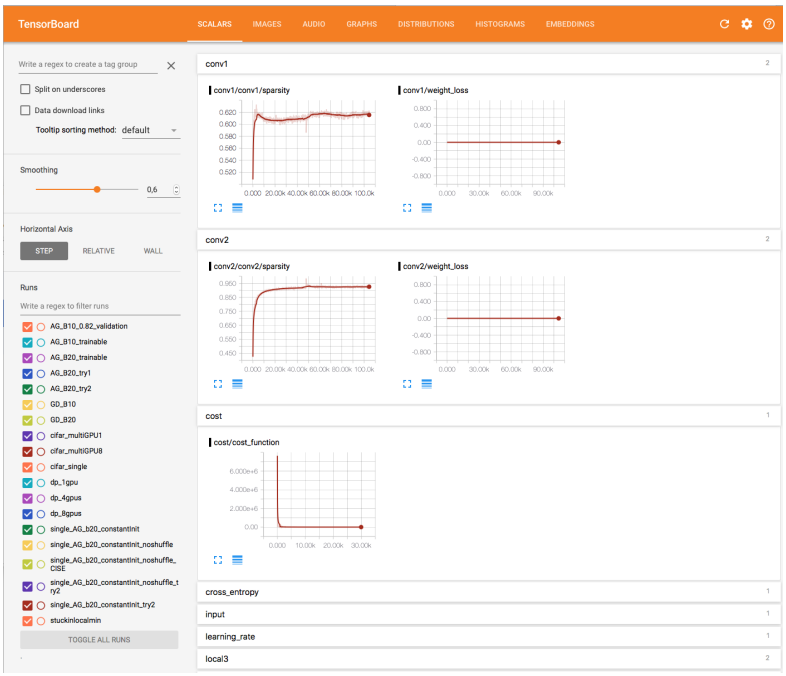


Figure 3: A sceenshot of the tensorboard interface

the server it sometimes happened that the ssh connection broke down causing all programs running on that connection to be killed. To keep running sessions alive, the use of *screen* is advised. *screen* continues to run when you disconnect a ssh connection and you can reconnect with *screen -r* any time. Before we used *screen* we lost several runs because they were interrupted and had to be restarted. On every machine an instance of Tensorboard ran. Every server can run one or more trainings at the same time if not all GPU memory is reserved. The results can only be compared by comparing different Tensorboard views. It is either possible to run several instances of tensorboard or, what we found easier, to copy the result into one folder which one tensorboard instance visualises. To simplify the process we propose to use only one instance of Tensorboard on each server in order to have one instance for all trainings happened on this machine. To understand the differences in training runs it helpful to require a name for a training or include the relevant parameters in the folder name automatically.

2.6 Problems

2.6.1 Software Problems

Because the servers have no displays, using matplotlib can yield errors, since the plots are displayed on the display by default. This can be prevented by using

the backend "agg".

```
echo "backend␣:␣Agg" >> $HOME/.config/matplotlib/matplotlibrc
```

We had problems using TensorFlow due to the fact that the installation of Cuda doesn't link the *libcudart.so* file correctly. We solved this by simply linking the file to */usr/lib*.

Under some circumstances when using deep learning libraries an error message can occur:

```
ImportError: libcudart.so.8.0: cannot open shared object file: No such file or
    ↪ directory
```

This import problem can be fixed by sourcing the */etc/profiles* file with bash (*source /etc/profiles*). This loads the needed environmental variables.

2.6.2 Hardware Problems

We had the problem that gammaweb01 froze (becoming not responsive) from time to time with no further information on the cause. After some observation and inspecting logs it became clear the system had kernel panics. After changing CPU settings on the BIOS we thought the problem was solved. Later the machine again froze. We think that this may related to multiple competing deep learning libraries causing a kernel panic.

3 First Experiments with Deep Learning

3.1 Xor Example

A very simple example is to implement the learning of an logical xor gate. The learning data is a single table containing the mapping in a matrix:

$$x = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix} \; y = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

The first two columns are the input tuple and the third column is the expected output.
For the computation we need two hidden layers because "xor" needs an "and" and an "or" input. "and" and "or" are then learned in the first layer and "xor" is learned afterwards in the second layer. This example can be found in the *xor_example.py* file.

3.2 Neural-Style

The application *neural-style* is the implementation in TensorFlow [2] of the algorithm described in a paper[3]. Neural-Style transforms captured images using an extracted art style. It takes at least two images as input; namely, the style input image and the captured image which has to be transformed. The output of the tool is the transformed captured image.
Depending on the image dimensions, the algorithms processing time differs from 1 minutes up to 3 min.
Basic usage:

```
python3 neural_style.py --content ./input_image.jpg
  --styles ./style_image.jpg --output ./output_image.jpg
```

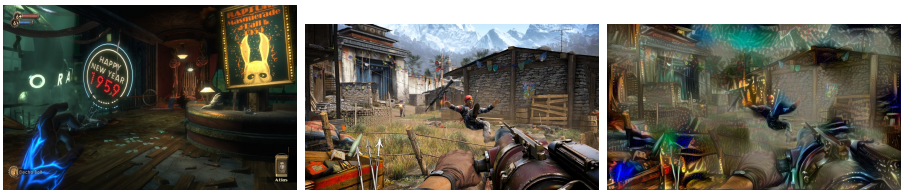
Here are some examples:

(a) style image (b) input image (c) output image

Figure 4: neural style example 1

(a) style image (b) input image (c) output image

Figure 5: neural style example 2



(a) style image (b) input image (c) output image

Figure 6: neural style example 3

3.3 Differences with Keras

Keras is another deep learning library. It is a high-level library, which can be used either on top of TensorFlow or Theano, another deep learning library. With Keras already developed software[5] could be tested on a server. We found out that using Keras it is possible to use the GPU, however only the first GPU can be used. Keras will become a part of TensorFlow with TensorFlow version 1.2 [6].

3.4 First Try at Multi-GPU Calculating

With distributed training it was shown that the training time can be lowered [4]. To observe the GPU load we used the provided GPU utility *nvidia-smi* (the same tool *check_MK* uses). This tool prints the current temperature, power consumption, GPU utilization and memory usage. By default TensorFlow uses just the GPU with the smallest ID. We observed that the load of this GPU was varying between 60-99% and the others weren't used for processing. To use the other GPU we changed our code to distribute the learning:

```
for i in range(FLAGS.num_gpus):
  with tf.device('/gpu:%d' % i):
```

After that, the load still was low but now all cards were used. With the change the execution time compared to single GPU was only 30%, so the overhead

[5] https://github.com/BSVogler/music-genre-recognition-pipeline
[6] https://blog.keras.io/introducing-keras-2.html

overweighted the speedup. We think that this followed our model design which was not designed to be distributed over multiple computation units.

4 Convolutional Neuronal Networks

Previous examples show the use-case of neural networks with convolution function. These networks are called convolutional neural networks (CNN). A CNN works by reducing the image dimensions and thereby using "kernels" to identify certain features. Those kernels have the positive property that they are invariant to transformation in contrast to the classical percpetron. In image tasks they work similar to edge-detection filters, but they are learned. CNN show good results in image and audio tasks and are therefore often used for such tasks. Of course the application of CNN is not limit to those tasks. Explaining CNN in depth is beyond the scope of this documentation. For the interested reader, we suggest Goodfellow's recent book [5].

4.1 Technical Term "Pooling"

A CNN is called as such when the network contains convolution layers. In fact, a convolution layer consists of two layers. One layer applies the kernel to the input, which is the convolution operation, named after the mathematical operation. Therefore, this layer inside the convolution layer itself is also called a "convolution layer", which can yield to confusion. The other layer is the "pooling layer".

We stumbled upon the question why the "pooling layers" are called as they are. In a paper of Y. LeCun in the year 1998, mentioning LeNet-5, the used word is "subsampling" [6]. The term "pooling" is used in more recent publications to this topic. It describes the general pooling function used for resampling (Chapter 9.3 [5]). Subsampling brings the benefit of reduced memory consumption as the dimensions decrease. For CNNs often the functions $max()$ or $average()$ are used.

4.2 Cat-Dog-Classifier

Instead of looking at finished solution to problems as before we now advanced exploratively. We wanted to implement a neural net on our own to discover the problems which can arise during the process and learn properly the details.

For that we searched for an appropriate dataset. We found a good website named *kaggle.com* with deep learning challenges and many datasets. The set of our choice[7] was a set of images either containing cats or dogs.

With this dataset we wanted to create a simple image classifier with the capability to distinguish between images containing cats or images containing dogs. Several examples helped us to create a CNN (Fig. 8) in TensorFlow solving the task. We now want to describe some problems we encountered building the model.

Due to the small amount of deep learning examples using convolution our rate of progress was very low at the beginning, but after the examination of some detailed implementations we were able to build our own network. Our

[7] https://www.kaggle.com/c/dogs-vs-cats

(a) cat example (b) dog example

Figure 7: Dataset items

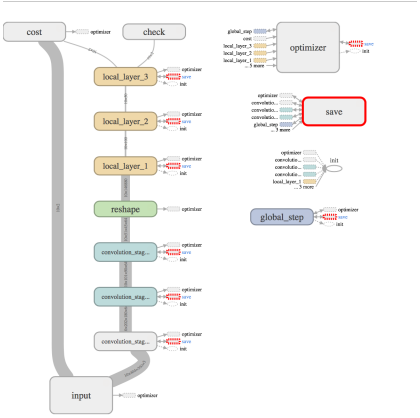


Figure 8: The graph of our developed net visualized with tensorboard.

net consisted of three convolutional stages, one reshape layer and two local layers. The typically used amount of convolution layers is two to three. The convolutional stages learned image kernels to detect specific features in images and the other fully connected layers resized the extracted information in order to get a resulting vector in the last layer with the correct prediction. The vector is in a special format called one-hot-encoded. Each scalar field of the vector is paired with an answer. In our case the vector is a 2x1 vector where one field describes the classification as a cat and the other as a dog.

Before filling the graph with data it must be properly initialized and the dimension must be known. We had some problems on calculating the correct dimensions for some layers, because when the net becomes deeper it becomes more difficult to keep an overview of how many dimensions are coming out of one layer and how many the next has to accept. The only dimension which can be unknown is the length of the input dimension. This variable depends on the number of items which are used in training.

To fill the data to the graph the complete dataset can either be loaded at once or by using batches. Most real-world examples are too big to be loaded at once. There are two ways to add the data to the graph in batches. One is to define variables and then fill the variables in a loop. A better approach is to use the included queue class.

The cost-function returned a result what made us question our choice for the cost function. The curve oscillated chaotic. After quite some time it did not seem to converge. However we did the simple mistake to not train the model long enough. After some hours the model starts to visibly converge.

After training you want to save the results. Saving and loading is needed for three use-cases: Validation, training continuation or production use. TensorFlow offers tools for saving at checkpoints and loading with the suffix ".ckpt.index". The model is saved in a file ending with ".ckpt.meta".

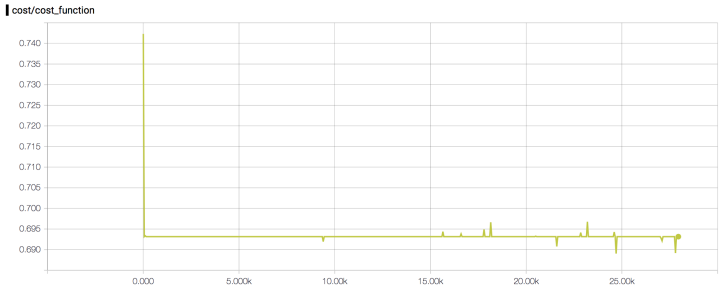
Saving is not that useful, if you can not load the model. There are two ways to restore a trained model: In theory one could either rebuild a model from source code and load the weights from a file or load a complete model including its weight. When model weights are loaded and the graph is not loaded or not matching the resulting error messages were not so helpful. In our case building the model and only loading the weights worked well. It is important to not initialize the variables, because this overwrites the loaded values with new starting values.

When the training is continued the curve in the display in tensorboard starts at a lower cost value, but at step zero again. This may be fixed by loading meta information.

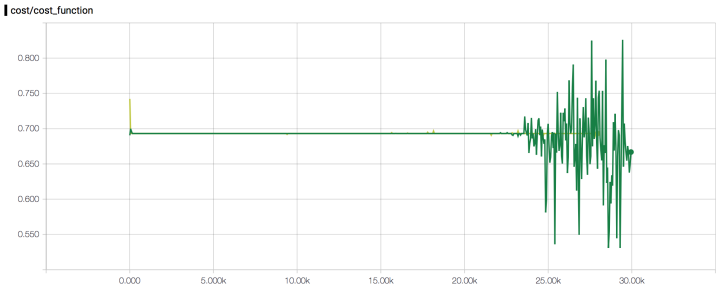
4.2.1 Results

The model converges to the cost value of almost zero. We wanted to verify or test the trained weights. Therefore a test set with different images showing cats and dogs is put into the trained net. The images are classified and the result is compared to the label then summed up to obtain a classification rate.

Sometimes the net did not learn anything new, i.e. it did not converge to a lower value in the cost function (Fig. 9a) . We think that this is due to a local minimum. In another training run it looks like it was again stuck in a local minimum, however after 22.000 steps the cost function started to get a growing amplitude. Usually the training ended at around 25.000 steps automatically. The condition to end the training is when no relevant progress is made for some time.



(a) The training does not converge.



(b) A training is first stuck.

Often the training shows a similar pattern at the start (visualized at Fig. 10). The cost function value is often the same, but some training keeps being stuck at this level.

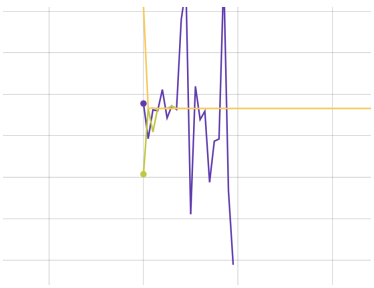


Figure 10: Training starts near a local minimum.

4.3 Second Try Multi-GPU Learning

As our first naive approach did not bring good results, we tried a more sophisticated one.

There exist two concepts of multi-GPU training. The first one is called replicated training (or called data-parallel) and the second is intra-model parallelism (or just called model-parallel)[8]. Replicated training trains several models in parallel. Intra-model parallelism splits the graph. For example one could split the calculations of a matrix multiplication on the GPUs. Because our model has not so long independent computation paths, we use the replicated training approach. An example for the standard problem "CIFAR10" can be found in the official model collection in the TensorFlow GitHub repository [9]. We used a distribution of jobs as the TensorFlow documentation suggests (Fig. 11).

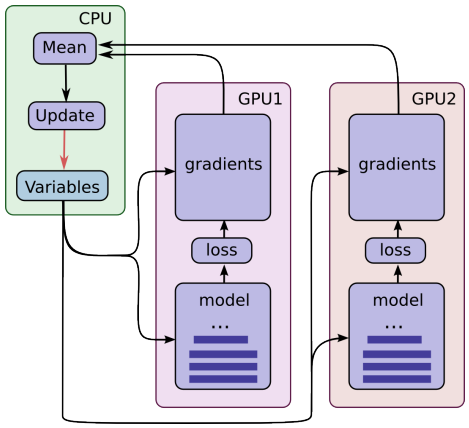


Figure 11: Model of distribution [7]

Our experiments with single GPU training had training times around 10 hours before the cost value were converging. We were wondering why the times were so high if they have been around the 4h mark previously. Resetting changes to the code to the state when we had one good training "run" did not bring back the fast training times. This hinted to a very large variance in training time ranging from 4 to 12 hours. With such a high variance it is hard to compare the performance impact of parameter changes.

4.3.1 Precision

With bigger batches, the training can be done faster. The batch size is limited by the amount of GPU memory and the size of one item. Especially convolutional networks are prone to huge memory consumption because of the high numbers

[8] http://stackoverflow.com/questions/37732196/tensorflow-difference-between-multi-gpus-and-distributed-tensorflow

[9] https://github.com/tensorflow/models/blob/master/tutorials/image/cifar10/cifar10_multi_gpu_train.py

of calculations. We tried reducing the memory by using only 16 bit precision floats as a possible way suggested by Dettmers:

> Another often overlooked choice is to change the data type that the convolution net uses. By switching from 32-bit to 16-bit you can easily halve the memory consumption without degrading classification performance. On P100 Tesla cards this will even give you a hefty speedup.

[8]. However by using this we were not able to increase the batch size, so the memory consumption stayed the same. This approach should allow to even further reduce to 8bit as described by Dettmers [9].

4.3.2 Measuring Performance

We compute the performance by comparing the progress in training (loss) relative to the time.

To compare the performance when changing some parameters we introduce the weights with a constant initialiser. This leads to a very high value in the loss function. Repeated results gave similar but still different results. Randomness in the run is still added by the random order of the training data in the queues. Hence, deactivating the shuffling should lead to deterministic behaviour. Of course running time of a process is influenced by the environment i.e. multitasking but the effect is only marginal. We could not eliminate the rest of the randomness, so we expect TensorFlow to introduce some randomness e.g. in the optimiser. For data parallel training deactivating the shuffling could lead to several models training the same input.

4.4 CIFAR10

CIFAR10 is a benchmark problem where images must be classified of ten categories. Google offers one example model with TensorFlow for multi-GPU training. We validated that data-parallel training with eight GPU is faster than data-parallel training with one card. We could also validate that using more cards is faster then using a non-data-parallel model.

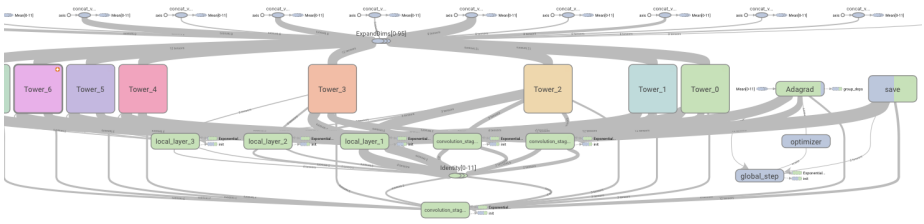


Figure 12: CIFAR10 model viewed in tensorboard.

5 Text Sequences

Text input can be viewed as a form of sequences and algorithms and neural networks for sequence labelling can be used. "In machine learning the term sequence labelling encompasses all tasks where sequences of data are transcribed with sequences of discrete labels." [10, p. 1] Classic examples are speech and handwriting recognition. In these examples the input data are images or audio streams. In this chapter we will look at text input. One important fact about most types of sequences is that the data is not independent, so sequences arise with certain statistic values.

5.1 Recurrent Neural Networks

We first start by using a Recurrent Neural Network (RNN), which is applied on character level. Previous results for using RNN on text gave surprisingly good results [11]. Karpathy used a RNN to predict the next characters to generate content based on a training set.

5.2 The Simpsons

We used this approach to train a model on the script of the well-known tv series "the Simpsons". The dataset was obtained via kaggle[10] and transformed to a format that fits our model. The input was then of the form:

```
1;13;Todd Flanders: Meri Kurimasu. I am Hotseiosha, a Japanese priest who acts
    ↪ like Santa Claus. I have eyes in the back of my head so children better
    ↪ behave when I'm nearby.;75000
1;14;Dewey Largo: And now, presenting Lisa Simpson, as Tawanga, the Santa Claus
    ↪ of the South Seas.;91000
1;15;Homer Simpson: Oh, it's Lisa. That's ours.;98000[...]
```

During training the model could already be queried to generate new content. The result evolved from noise with random letters to a sequence of only english-sounding words:

```
549;78;Weasel Presinger: Eah, um, you're too Islivest, hoop. You know, they have
    ↪ momenting at school can't say it. Can you have so pictures to fly407;96;
    ↪ Kirk Van Houten: I want Sweet, or I don't even votement fundary. (
    ↪ DRAMATICALLY) everyppuppets and our reach we gonna diched it up at this
    ↪ !;337;23;Seymour Skinner: (HAPPY BULLBO) Reviet five hundred I'm not su
    ↪ bit now, Moe. Apu! Food now. I love a blaxes did a letter carry into a
    ↪ remov27;207;Bart Simpson: Where are you good? I drive to keep the one:
    ↪ upoding Uhwan. I'll call them fun back at schools are already gojes. We'
    ↪ ve consu456;230;PRUFFOM BUILDING: Oh quiet, but I don't was awful the
    ↪ game would panic[...]
```

The missing line breaks in the output may be the result of different line ending characters (CRLF) or it may be needed to train a end-of-line sign like $\backslash n$. It also seems that each line has fixed sized length and sentences end abruptly.

[10] Various . csv files in a relational database fashion `https://www.kaggle.com/wcukierski/the-simpsons-by-the-data`

We are not sure if sequence to sequence (s2s) i.e. translations could be called sequence labelling as it may fulfil the definition of a pattern classifier [11]. Language translation allows more than one translation as the task of language translation is a label to label transformation with fuzzy and ambiguous meaning.

5.3 Learning a Parser

Transforming *Wiki markup* (the markup language used for wikipedia documents) to HTML is not properly possible with many tools. Only the php implementation of wikipedia[12] does it properly and robustly. Different implementations of the parser software emulate the behaviour of the original implementation. We want to find out if it possible to learn the Wiki markup to HTML parser.

5.3.1 Natural Language Sequence Model Architecture

We found a working model, namely the existing s2s model used in a tutorial for TensorFlow from Google. There exist some other s2s models for keras. Google's official documentation contains a chapter about a s2s model[12]. The model is an implementation of the described statistical machine translation system based on [4]. We were already familiar with TensorFlow and it allowed us to use the full calculating power of our hardware; therefore, we wanted to use this model as a foundation. At the time of this writing the referenced source code in the article of the needed API could not be found in the GitHub repository as the code was removed or replaced. We first want to outline how the model works and later compare how we can apply it to our parser problem. As an example the english language should be translated to French using statistical machine learning.

First, in a preprocessing step a vocabulary index is created. The index contains a list of vocabularies with fixed length. Secondly the text in both languages is tokenised by giving each word an id based on the index, thus limiting the number of tokens. The English to French example used a default vocabulary size of 40.000.

After the preprocessing the model is trained. The net learns by looking at the statistic values of the appearance of words, so the semantics on a character based level is not relevant.

Pure RNN lack the feature to save values over a long time as values vanish. To tackle this problems Long Short Term Memory (LSTM) cells were invented. Similar is a Gated Recurrent Unit (GRU), which "[...] was proposed by Cho et al. [2014] to make each recurrent unit to adaptively capture dependencies of

[11] "A pattern classifier $h : X \to Z$ is therefore a function mapping from vectors to labels"[10, p. 6]

[12] The source code for the parser is located at includes/parser/ in the repository at `https://github.com/wikimedia/mediawiki`. Counting the lines of code of the wikipedia parser with *cat ./includes/parser/* | wc -l* returns 16.809 lines.

different time scales. Similarly to the LSTM unit, the GRU has gating units that modulate the flow of information inside the unit, however, without having a separate memory cells."[13]

When translating one language to another usually the sentence lengths are variable. This problem is similar to the different image dimensions in convolutional neuronal networks. The article mentions some ways to deal with this problem.

- use a different subgraph for every length combination
- pad every sentence to the maximum length (results in bad performance on small input)
- Use buckets for a compromise between many subgraphs and padding

The model uses the last solution.

The described model currently has a bug which let it to fail with the following message:

```
ValueError: Attempt to reuse RNNCell <tensorflow.contrib.rnn.python.ops.
    ↪ core_rnn_cell_impl.GRUCell object at 0x7fe5b80c2f60> [...]
```

This is probably due to a recent change in the master branch. [13]

5.3.2 Natural Language Model to Parser Model

We now want to show how the natural language model can be transformed to a markdown parser model.

In order to train the parser two things have to be defined. First the corresponding input-output pairs have to be defined. A pair is one item for the training, in the following called a sentence. In the natural language model the data is separated until a dot appears or more generally by using a line break thus including single words. However, one item in the wiki source could lead to a nesting in the HTML output spanning several lines so defining a sentence line by line does not work. Hence, the lines in the HTML output are not statistically independent (e.g. opening a container with `<div>` will lead to a closing tag `</div>` some lines later). One could argue that sentences for language translation is neither independent but single natural language sentences can always be translated to a valid translation in another language. Because of the dependency we wish to use as much lines as possible up to one whole article. We empirically found out that wikitext does not have a level above a chapter in the HTML-tree, which is introduced with a heading. Therefore we want to use one chapter or subsection of an article as a sentence.

A sentence is made up of words. What is a word in this case? Words in most Indo-European languages are separated with a space or the end of a sentence. For markup code or programming languages this is more complicated. There are more delimiters and depending on the context the meaning of the delimiter

[13] A ticket on the GitHub repository is dealing with the error message:`https://github.com/tensorflow/tensorflow/issues/8191`

can change. Also each language has it's own delimiters. For some languages line breaks are used, for others a semicolon separates each command (e.g. C++). In markup there are no command-ending characters. A line break can introduce a heading when followed with equal signs or in a paragraph does not perform an action. Therefore we use a word separation at character level just as in the previous example with *the Simpsons*. In this case the use of a vocabulary index file and tokens are not needed and the files could directly be used without preprocessing.

Now if we assume that we have our dataset (we describe how we obtained this in the next chapter) we can train the parser.

In the previous model some simple change has to be made to work without using a delimiter. The code then has to be changed to not look for the space as a separator (translate.py). Getting the code for each character could be obtained with `[ord(x) for x in source]`.

Because of the mentioned bug, we tried a different model [14]. This model can also use buckets, which are disabled by default [15]. In this model we used a simple vocabulary file with common characters. When using an empty delimiter the input is separated byte-wise, so multibyte UTF-8 characters are split [16]. This leads to an error during training, because this creates invalid UTF-8 characters. We therefore changed our dataset for this model, so that a special character follows each character. This special character can then be used as a delimiter.

With this model the training could be started. During training, after some training steps, some sampling takes place, so it is possible to check if the training is working. For the sampling the saved model is loaded. During this process the model freezes.

5.3.3 The Wikipedia Dataset

In order to train a model, we first had to retrieve and pre-process the dataset. This is the point where the deep learning meets big data. For that we created a crawler which parsed a mediawiki xml dump in order to extract the markup text and download its corresponding html rendering. The first and naive approach to simply download the wikipedia html at full speed didn't stick to the wikipedia crawler conventions which led wikipedia to block our ip. One option to get the html representation of an article would have been to use all machines of webis on a subset of the wikidump and crawl with an appropriate delay. But we decided to set up a local wikipedia mirror on a server and crawl that.

For that we first followed a basic instruction[17] but this approach didn't work either. The mediawiki software could be installed and was configured so that a fast crawling could be possible (no caching needed and unicode normailization).

[14] `https://github.com/google/seq2seq`

[15] For a list of arguments see `https://google.github.io/seq2seq/training/#training-script-reference`

[16] `https://www.tensorflow.org/code/tensorflow/python/ops/string_ops.py`

[17] `https://github.com/entityclassifier-eu/entityclassifier-rest/wiki/How-to-setup-Wikipedia-mirror`

Because the suggested software *mwdumper.jar* was not updated for some time and seems to be not compatible with the current mediawiki software we used a different software. We used the included maintenance php script to import the xml file. The first run run worked till roughly revision id 50000. After fixing the thrown erros the whole was imported.

After the import of the wikidump the revision id (or in the URL called the *oldid*) did not match the ids in the xml dump any more. Therefore the extracted markup files have to be renamed to form matching pairs. This is another function the parser provides, using an unsorted list of the revision ids.

The wikipedia dataset has a speciality. It uses patterns. There are templates used, where the content is defined on a different page. One template in the form of wikitext as [[Category:*categoryname*]] will be the same for every page. Therefore, it would be best to replace the several output tokens compounding the template with a single one. This is another step in the preprocessing to avoid wasting resources on learning the wrong things.

On a regular office computer scanning and extracting the revision ids from the xml dump and sorting them by article length took 24 minutes and resulted in 16,526,000 revision ids. This is relatively quick, but not every operation on the dump is that fast. Dealing with such huge file sizes can take a very long time. Our script for extracting the markup from the dump and save it in separate files ran for more than two weeks straight. In the next chapter we will now see that this can cause another problem.

Concatenating Files Now we have a folder with a huge amount of files. Most models in this context require two big matching files with all the training data. To concatenate the files one encapsulated command can be used. For the shell *fish* (using bash is similar):

```
cat html/(sh -c "ls␣html␣|␣grep␣-E␣^[0-9]\{4\}.html") > concatenatedFile.html
```

or picking a range with

```
cat markup/(seq 0 9999).markup > concatenatedFile.txt
```

When the amount of files exceeds a limit an error message is thrown:

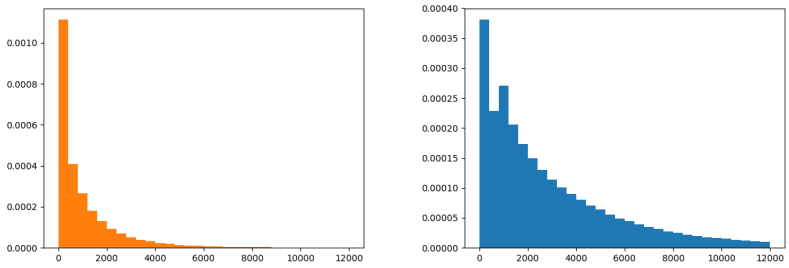
```
The total size of the argument and environment lists 167MB exceeds the operating
    ↪ system limit of 167MB.
```

Therefore the parser script has a method to perform this operation with an additional print-message showing the progress.

Before the training can be started the files must be split in a train and validation set. This is usually done with an 80/20 split. For this the commands `head` and `tail` in combination with `cat` can be used.

5.3.4 Buckets

As we described we may use buckets for our model. Before setting the buckets sizes we have to know the distribution of the tokens in the sentences our dataset.



(a) markup chapter length, 3.5 Mio. articles scanned

(b) HTML chapter length, 0.5 Mio. articles scanned

Figure 13: Distribution of wikipedia chapter length

Therefore we counted the number of characters in each sentence i.e. in each chapter (Fig. 13).

6 Future Work

We could not gain a real benefit by using multiple GPUs at the same for a one training. Finding the bottlenecks is not an easy task and often depends on the model. Finding heuristics or developing tools to identify the bottlenecks is something which can be further examined.

We successfully obtained a dataset for our planned training. Increasing the size must be accompanied with checks that the lines still match. The logical next step is to prototype the training with a model and improve upon. Either the issues with the models we tried can be resolved or a another model could be tested. Learning a C or Java compiler with applications compiled from a single file may be easier because the languages have a stricter grammar. Replacing repeating patterns is another pre-processing steps which we discussed and should be implemented.

References

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] Github repository neural-style. `https://github.com/anishathalye/neural-style`, 2016. urldate: 2016-11-25.

[3] Leon A. Gatys, Alexander S. Ecker, and Matthias Bethge. A neural algorithm of artistic style. 2015.

[4] Jianmin Chen, Rajat Monga, Samy Bengio, and Rafal Józefowicz. Revisiting distributed synchronous SGD. *CoRR*, abs/1604.00981, 2016.

[5] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[6] Yann Lecun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. In *Proceedings of the IEEE*, pages 2278–2324, 1998.

[7] Google Brain Team. Tensorflow parallelism. `https://www.tensorflow.org/images/Parallelism.png`, 2017. [Online; accessed April 9, 2017].

[8] Tim Dettmers. Which gpu(s) to get for deep learning: My experience and advice for using gpus in deep learning. `http://timdettmers.com/2014/08/14/which-gpu-for-deep-learning/`.

[9] Tim Dettmers. 8-bit approximations for parallelism in deep learning. *CoRR*, abs/1511.04561, 2015.

[10] Alex Graves. *Supervised Sequence Labelling with Recurrent Neural Networks*, volume 385 of *Studies in Computational Intelligence*. Springer, 2012.

[11] Andrej Karpathy. The unreasonable effectiveness of recurrent neural networks. `http://karpathy.github.io/2015/05/21/rnn-effectiveness/`, May 21, 2015.

[12] Sequence to sequence. `https://www.tensorflow.org/tutorials/seq2seq`.

[13] Junyoung Chung, Çaglar Gülçehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *CoRR*, abs/1412.3555, 2014.