Pázmány Péter Catholic University
Faculty of Information Technology and Bionics

Deep learning framework in TensorFlow for biomedical research

Csaba Botos

2017

A review submitted in partial satisfaction of the requirements of Guided Individual Study.

Advisor: PhD. István Z. Reguly

Abstract

Research in deep learning area is getting more attention, new fields where machine learning can be introduced are looking for efficient ways to test whether previously succesful architectures can be applied or not. To achieve a representative baseline and to allow fine tuning, it requires to maintain a stable work environment, yet flexible enough to try out new ideas. In this work we aimed to build such a general model, that is capable to integrate handcrafted features at different level of proces and automatically summarizes our experiments for later reconsideration while making the best use of our large computational capacity with parallel runs operating on both CPU and GPU.

Contents

1. Introduction

Motivation. Currently I am working in a research group, where my task is to design, implement and maintain the Deep Learning background of our experiments. At the time of writing we are working on a classification task, where a moderate set of labeled, different length, single channel ECG (Electro Cardio Gram) samples are provided. Our task is to research the main patterns which can indicate, or even predict the occurrence of a common cardiac failure, called atrium fibrillation. The greatest challenge in this task is the small number and the unbalanced distribution of entries between classes.

Focusing on recent success of different Deep Learning architectures, one can get easily intimidated to implement them for individual purposes. However many experiments are failing due to wrong choice of models, or not being able to track down the right hyperparameters. There is no theorem (yet) for answering whether a network will converge or not, even if it does so, it may fall in a local minima of the search space, that is not satisfactory. Following thumb rules [4, 1], can reduce our search space, still we have to be able to track our progress, and make every experiments reproduceable. Also if large computational capacity is present, it is still not trivial how to exploit our sources. To make the best use of our given background we decided to use a framework that is transparent in such a manner, that we don't have to rewrite our codebase if only CPUs are available, multiple GPUs or even cloud computing clusters are at dispose. For these reasons we chose TensorFlow, which my work was focused on and this writing is based on.

Across the many implementations I have met in open source projects [5, 2, 1] and online tutorials [15, 7, 14, 4], I tried to decide which of them would fit the best our needs. Thankfully the great camp of enthusiasts and support of the library has developed a site for gathering the most popular patterns to use when it comes to actually writing the tests [12].

The main goal of this writing is to describe the process step-by-step of developing the technical background of a succesful Deep Learning product, where I present what I learned from my personal experiences.

2. My commitments to the project

I aim to generalize this writing and be less specific about the exact project we are working on, so the following experiences can be transferred to other fields of otherwise non related studies. Basically our task was a simple classification problem for time samples of various length. My task was not just to try out famous Machine Learning concepts which *may* work, but to support our research group with the necessary backend of the experiments carried out, and to integrate different results in the final unified model.

Environment. For doing so I decided to use TensorFlow environment, in which I had previous experience, i.e. in building semantic analyzer for movie dialogs, image classifiers, and text generation for chatbots, image generation using Generative Adversarial Networks etc. For our project we started by applying different previously proven to be succesful feature extraction methods, and for introducing Deep Learning we wanted to improve our progress by building on top of these features. So in the first step I had to break down the recent DL architectures I wanted to use to basic modules. When reimplementing them I paid attention to make spaceholders or entry points for every possible external features my colleagues developed.

Entry points at different level of processing the input can be categorized as the following: - Variable length features (i.e. output of filters). - Fixed representation features (i.e. variability indices, frequency domain components). - Suggestions for classes (i.e. external classifier suggestion)

Also, to improve the model's efficiency I applied queues after different entry points to make mini-batch processing available, by filling these queues independently in mutliple threads, so the network trainer do not have to wait for these external sources to finish their processes to update the network's weights.

Data standardization. The biggest obstacles at first sight in using TensorFlow are the Tensors and the (control) Flow itself. In order to deploy a model we have to first assemble its computational graph, which will be compiled at run-time so it can be fed with real NumPy arrays and evaluated in the most efficient way relying on the TensorFlow backend.

Knowing that any time we feed values in, and retreive values from Tensors the whole process has to switch back and forth between C++/CUDA backend and the Python interface, which in case of large arrays is computationally expensive, and possibly collapse the parallel processes to sequential evaluation causing unnecessary run-time overhead, making TensorFlow seemingly inefficient. In order to achieve best performance we have decrease the number of python API calls of Tensor evaluation, to let the backend order its computations in one run.

Special case of reducing calls is to discard the notorius use of placeholders [9, 11], and introduce Queues to the model.

TensorFlow comes with a handful of convenience functions and wrappers which can handle data loading and preprocessing in parallel threads, controlled by a native thread coordinator.

These operators can be built in the graph without actually loading the data, and provide Tensor outputs which instead of placeholders will feed the network automatically on evaluation, so we can forget the *feed dict*. By using queues we can ensure that the main training process is running continously by chopping minibatches off the queues' front while new samples are fed back to the other end, possibly on different devices and multiple threads. As a result the main process will not starve on input data, neither will be continously interrupted by Python calls to simply just transfer values from the API to the backend.

For the reasons above I prepared a *writer.py* script that takes our raw data files, reads them into memory and write them out in a single file in the suggested file format, *.TFRecord*. These files can be managed more easily, preserve the sufficient data in a compact way [13], and the most important part, they can be automatically transformed into Tensors.

To make our experiments standardized, we first separated a train and a test set from the original data in 80-20 ratio, and due to unbalanced entries class representation, I wrote and augmentation script to create an evenly sampled pre-training set with a disjunct train and validation set (80-20), in order to prevent the model from learning a single answer for every sample.

Digital Filter. Since our original approach was built upon modified Pan Tompkins filtering method [8]. When we refactored our code base to use TensorFlow, we approached a great obstacle: there was no previous work involving filters. While in SciPy we could reproduce the same results of the original MatLab code [3], the above mentioned problem of feeding external values instead of integrating the process in the computational graph was apparent again. We had to decide whether we pre-process all sample before writing the 'TFRecords' files keeping the heavy computing outside of our model, or implement the repeatedly used 'filtfilt' operation, and rewrite the method completely using abstract native TensorFlow Tensors and Op nodes.

Considering that we have to deploy the algorithm in real time inference application, we choose the latter. As a first step, my colleagues reimplemented the method using NumPy, that I used as a template to follow while writing a new basic TF operator called 'lfilt'. I paid attention to keep the TF policy which is that operators borrowed from SciPy and NumPy should be transparent by function call signature, and the output should behave the same way as in the original implementation.

- FeedBack filter components: $A = a_0, a_1, a_2 \dots a_M$, where $a_0 = 0$ by definition - FeedForward filter components: $B = b_0, b_1, b_2 \dots b_N$

```
# Lfilt operation definition at time step 't'
y(t) = x(t)*b(0) + x(t-1)*b(1) + x(t-2)*b(2) ... x(t-M)*b(N)
                  - y(t-1)*a(1) - y(t-2)*a(2) ... y(t-N)*a(M)
```

$$y_t = \sum_{i=0}^{N} x_{t-i} \cdot b_i - \sum_{i=1}^{M} y_{t-i} \cdot a_i$$

```
y(t) = x[t-N : t] * b[::-1] - y[t-M : t-1] * a[::-1]
```

for faster computation let s be $s := x * b$, where $*$ is the convolution operator. This can be evaluated previously at once since all values of x is known at computation time

```
y(t) = s[t] - y[y-t : t-1] * a[::-1]
```

As in the SciPy source code we can see [10], on lfilt calls both A and B are expected to include the 0^{th} parameter as well, which in the feedback operator's case will mean that in the exact computation we have to normalize both A and B by A_0 and leave out the 0^{th} component from A, since it is the corresponding constant of the current y_t value.

Visualization. We always have to keep track, of what we have tried before, and after dozens of runs of experiments it is only up to us what we logged, and how well can we remember. To make a decision on future steps in the project it is always useful if we can avoid crawling through a messy git history to find the actual train scripts we wrote, then start to extract the hyperparameters from the code - it uses up a tremendous time, and chances of missing something is very high.

To make debugging, optimization, and workgroup meetings productive, I used TensorBoard, that is able to read native TensorFlow log or *summary* files and automatically turns them into spectacular graphs (see Figure 2.1 and 2.2), and quantitative metrics (see Figure **??**). The more explicit name and variable scopes we include in our code the more detailed the resulting graph and scorings will be.

By utilizing TensorBoard we boosted our progress, because we could easily determine if a training should be stopped because the network collapsed, track down which layer was causing

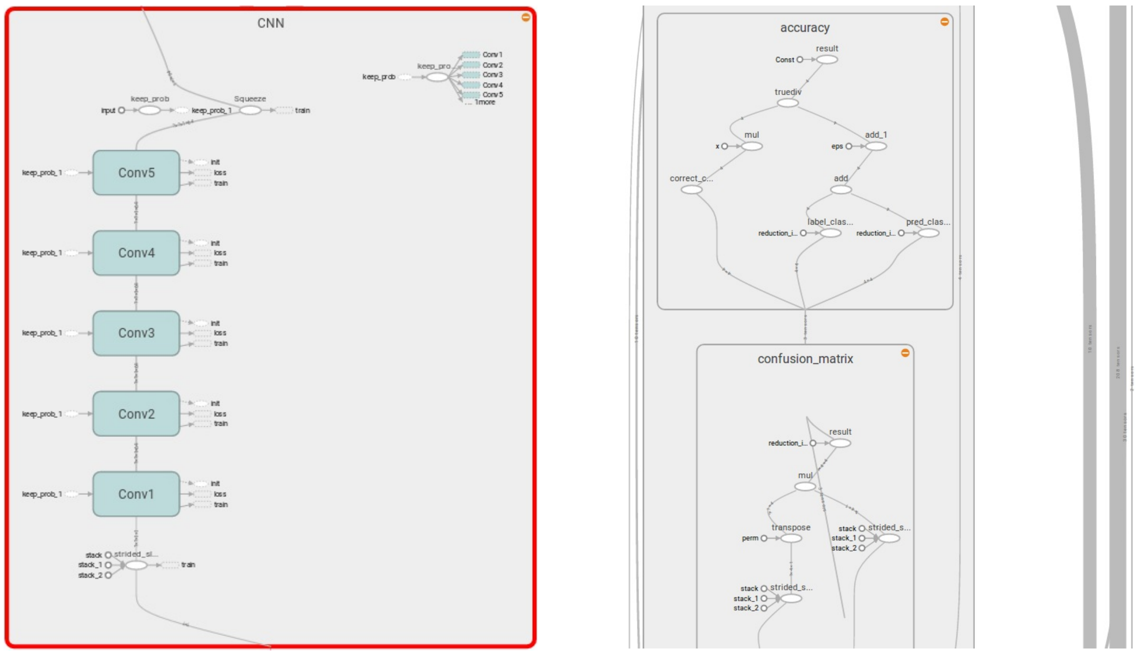


Figure 2.1: Left: Inner structure of a FCN. Right: The *Confusion Matrix* operator is build using elementary TensorFlow operations.

this failure by investigating the forward and backward pass of activations and gradients, the distribution of the models variables grouped by layers - all in all whether an experiment branch should be further investigated or not.

Fast prototyping. To make our experiments easily reproducible I refactored our network modules to be able to build themselves by just simply using a few high level hyperparameters, just like as number of internal layers, capacity, reduction factor (av/max pooling), etc. and made a template `.json` file which held these prototype variables, allowing us to use the same training script to build and train a wide variety of different architectures while saving and logging them in high detail to separate directories.

When servers with multiple GPUs are available, we can also boost our fine-tuning phase, with running trainings on parallel threads. By default a single TensorFlow session allocates the whole GPU source, even if the graph is not able to use but a single unit. So for parallel tests, I wrote a shell program with which we can launch the training with different parameters, automatically reserving the required devices to each training and restricting it from interfering in other processes.

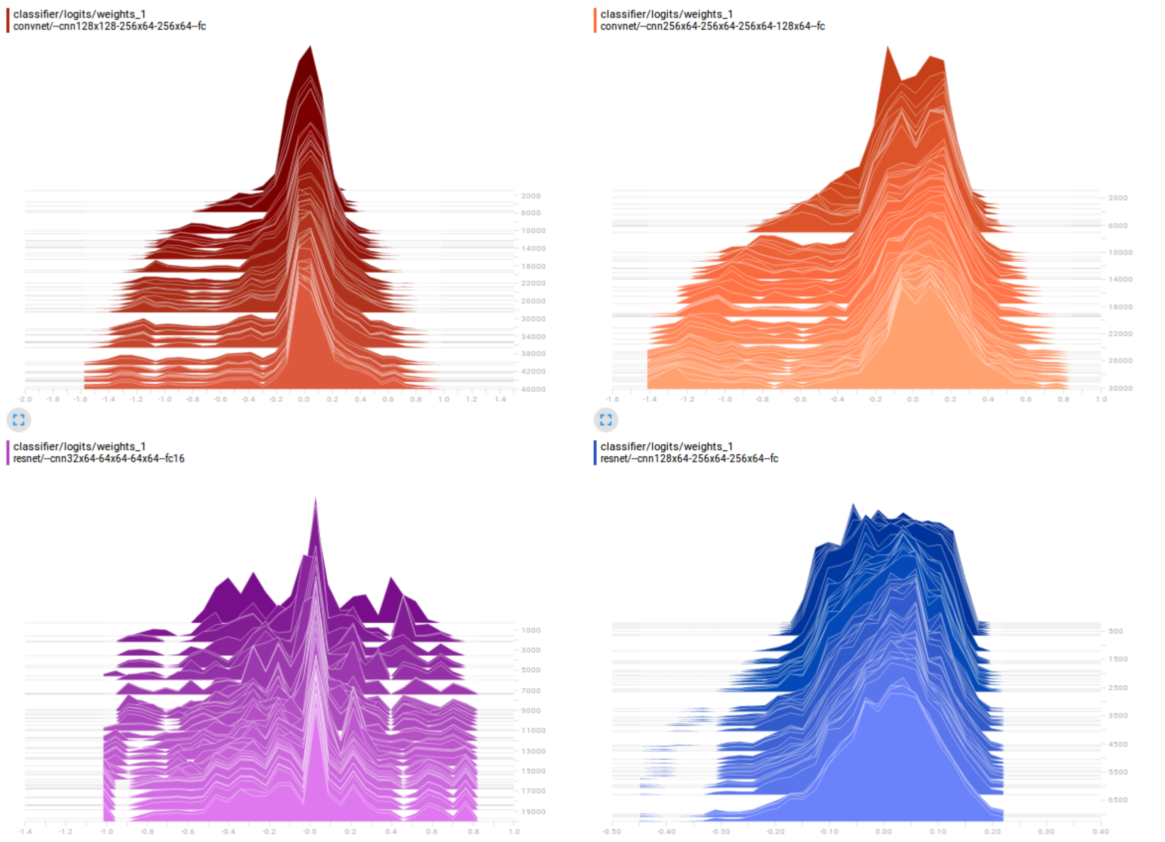
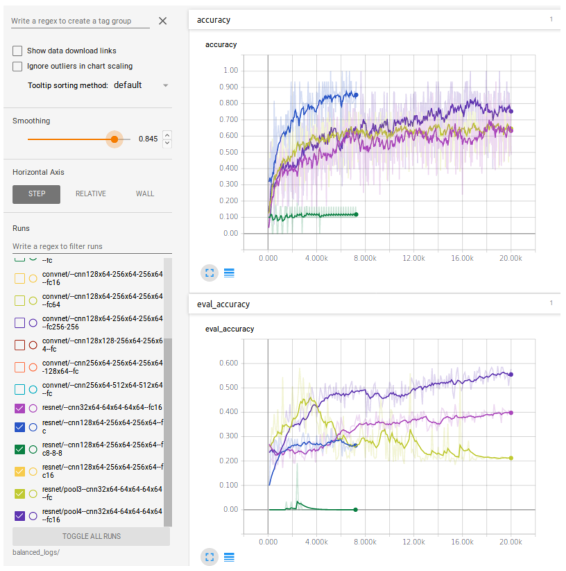


Figure 2.2: Using histograms we can keep track of our model's progress during the training session. We may look for local pikes in the distribution of the layers' weights, or biasing toward a specific mean through time.



Write a regex to create a tag group

☐ Show data download links
☐ Ignore outliers in chart scaling

Tooltip sorting method: default

Smoothing 0.845

Horizontal Axis: [✓ STEP] RELATIVE WALL

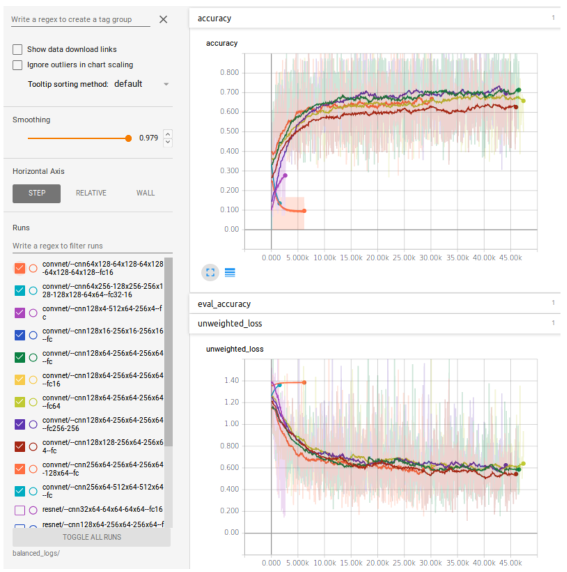
Runs

Write a regex to filter runs

--fc
☐ convnet/--cnn128x64-256x64-256x64--fc16
☐ convnet/--cnn128x64-256x64-256x64--fc64
☐ convnet/--cnn128x64-256x64-256x64--fc256-256
☐ convnet/--cnn128x128-256x64-256x64--fc
☐ convnet/--cnn256x64-256x64-256x64-128x64--fc
☐ convnet/--cnn256x64-512x64-512x64--fc
✓ resnet/--cnn32x64-64x64-64x64--fc16
✓ resnet/--cnn128x64-256x64-256x64--fc
✓ resnet/--cnn128x64-256x64-256x64--fc8-8-8
✓ resnet/--cnn128x64-256x64-256x64--fc16
✓ resnet/pool3--cnn32x64-64x64-64x64--fc
✓ resnet/pool4--cnn32x64-64x64-64x64--fc16

balanced_logs/

Figure 2.3: Performance test of Deep Residual networks



Write a regex to create a tag group

☐ Show data download links
☐ Ignore outliers in chart scaling

Tooltip sorting method: default

Smoothing 0.979

Horizontal Axis: [✓ STEP] RELATIVE WALL

Runs

Write a regex to filter runs

✓ convnet/--cnn64x128-64x128-64x128-64x128-64x128--fc16
✓ convnet/--cnn64x256-128x256-256x128-128x128-64x64--fc32-16
✓ convnet/--cnn128x4-512x64-256x4--fc
✓ convnet/--cnn128x16-256x16-256x16--fc
✓ convnet/--cnn128x64-256x64-256x64--fc
✓ convnet/--cnn128x64-256x64-256x64--fc16
✓ convnet/--cnn128x64-256x64-256x64--fc64
✓ convnet/--cnn128x64-256x64-256x64--fc256-256
✓ convnet/--cnn128x128-256x64-256x64--fc
✓ convnet/--cnn256x64-256x64-256x64-128x64--fc
✓ convnet/--cnn256x64-512x64-512x64--fc
☐ resnet/--cnn32x64-64x64-64x64--fc16
☐ resnet/--cnn128x64-256x64-256x64--f

balanced_logs/

Figure 2.4: Performance of Fully Convolutional networks

3. Benchmark of `filtfilt`

In order to verify that my implementation of the `filtfilt` operator in TensorFlow is better for large scale applications than the publicly available state-of-the art method found in `SciPy.signal module` [6], we attempted to measure the time required for evaluation. For benchmarking purposes we used random sequences sampled from standard normal distribution.

I have prepared multiple test runs, where we compare the two methods by evaluation time concerning the sample length, the number of filters applied and the number of samples.

Single sample - Single filter with different sample lengths

In the first comparison the TensorFlow implementation performs poorly, we had to apply different y label ticks. The reason for this result, is that the implementation uses GPU acceleration via TensorFlow CUDA kernels, and the mobilization of the data (copying into the graphic memory, and copying the result out from it) takes more time than actually executing the operation. Moreover, currently the TensorFlow backend does not support mutable `tf.Variables` inside native iterative loops `tf.while_loop`, meaning that we cannot modify specific values by indexing an immutable `tf.Tensor`. In other words, `tf.Tensor` is the only interface for the inner body of the while loop that can communicate with external graph nodes. Technically this yields, that the fixed size array of values cannot be allocated before the execution of an iteration, because the resulting space can only be handled by an interface that does not support element modification. As a transient solution we had to include array concatenation, so in every step t the array of previous outputs $[y_1, y_{t-1}]$ has to be concatenated to the current y_t output value, resulting in memory expensive operations.

Batch of samples - Single filter run, with same length samples

When multiple samples are available at run time, we can speed up the process, by running the same filter process in parallel. Here we normalized the evaluation time per sample per

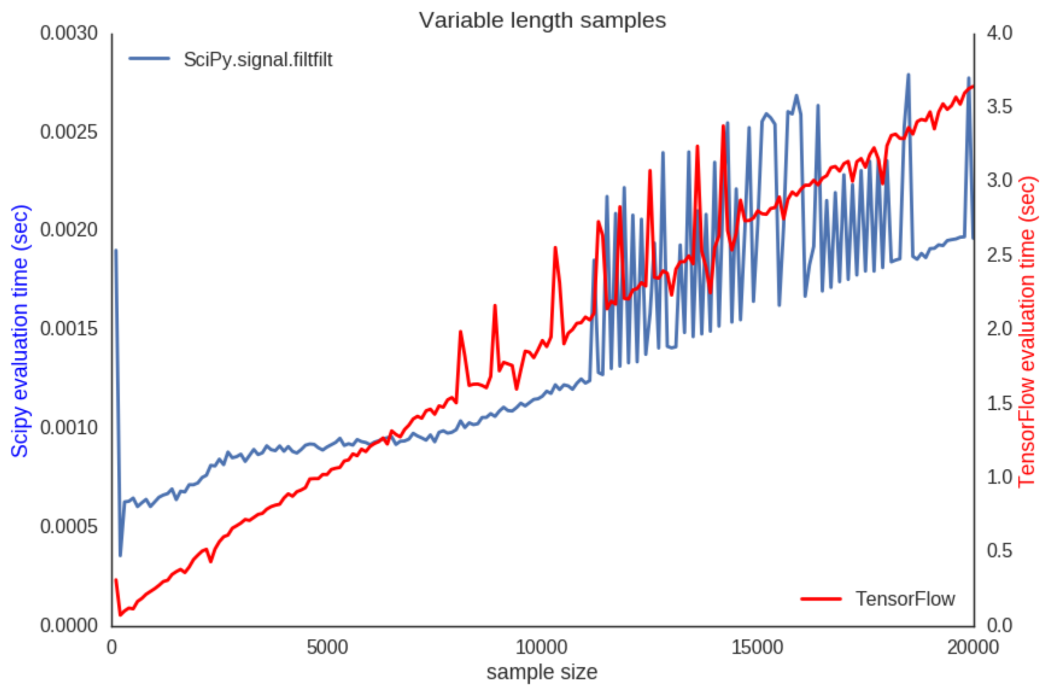


Figure 3.1: Time of evaluating a single sample with a single filter, at different sequence lengths. Later the measurement depicted with the blue line will be used as a baseline representing SciPy's performance on the given task. Notice that for we introduced a new axis for the TensorFlow implementation's time scale, because on single samples the performance of the two methods were not comparable.

filter of the SciPy method using the statistics of the previous experiment, since the method allows only processing in sequence, and after reproduced the standard SciPy baseline, to use as a reference point in further benchmarks, e.g. Figure 3.2. As we can see in Figure 3.3, the TensorFlow implementation runs in constant time at low batch size, and later starts to increase linearly

Time per sample

Finally, we normalized the computation time with regards to the batch size, as a result we have a curve that tells that with fixed sample length at given batch size how much time does it take to evaluate a single entry. For a detailed benchmark see Figure 3.4.

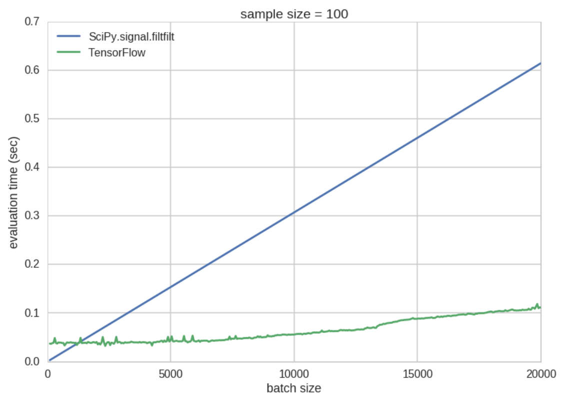


Figure 3.2: Comparison to the standardized SciPy baseline with regards to the baseline. The baseline is computed from the normalized average of the different sample length runs from the variable length single sample experiment. We show that with sample length 100, the current implementation only functions better when evaluation above 1500 sample at once.

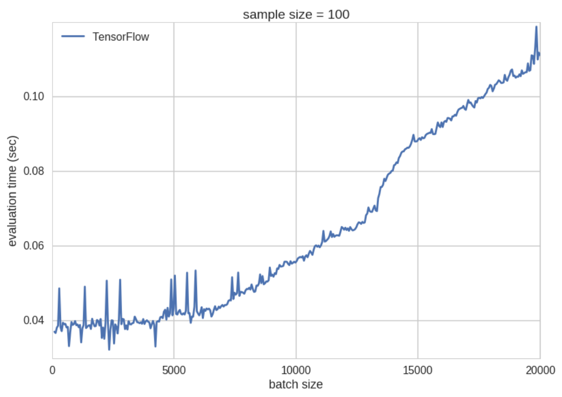


Figure 3.3: Here only the TensorFlow implementation's evaluation time is depicted without the baseline. It reveals that under 4000 samples the process is nearly constant in time complexity. Above 2000 samples it starts to behave linearly, but it is important to mention that the slope of the fitting linear curve is significantly less steeper than the SciPy baseline.

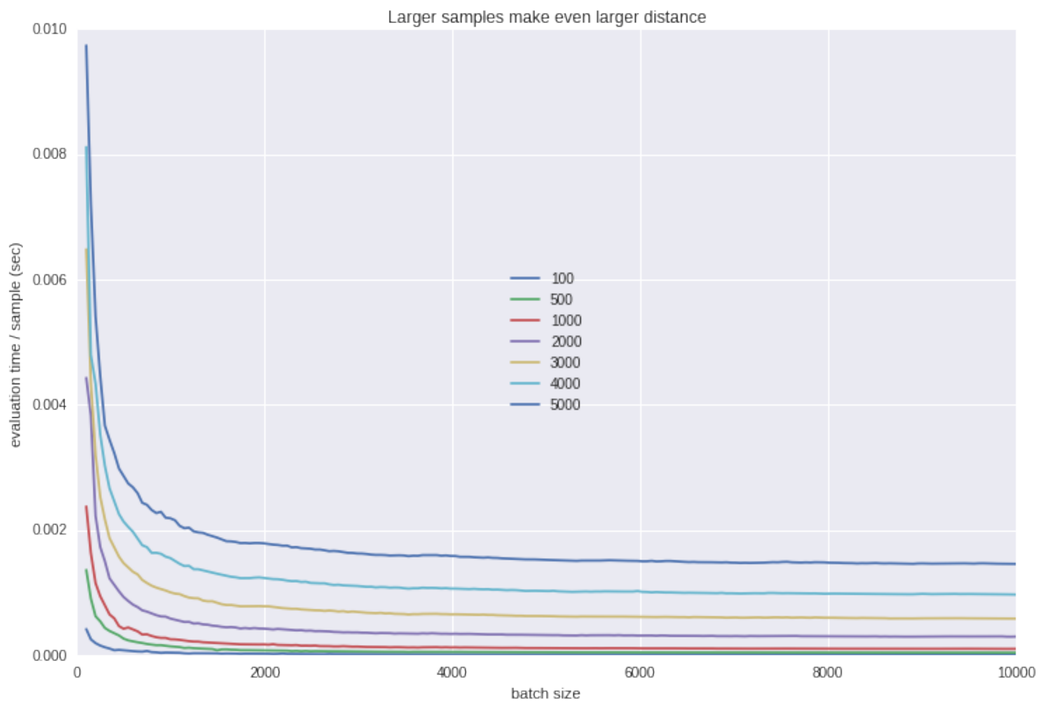


Figure 3.4: This curve tells that with fixed sample length (different curves on the figure) at given batch size how much time does it take to evaluate a single entry.

4. Future

Inference. When a reliable result is ready, with the saved weights and parameters we can transfer it to a simplified model, where we discard the auto differentation and back-propagation function in order to speed up the evaluation. In our situation, in inference time we are not allowed to use batch processing. The current solution requires to compile a specific model and load it's weights from previous training session, to infer a single sample - then every object is destroyed until the next inference call is made. To solve this problem, we intend to create a template that can do the necessary steps mentioned above with any kind of our model, after that it should run in the background and stay in idle state (rather than exiting the process) until a new sample is provided. We believe that the mentioned solution will radically reduce our inference time.

Filter. My next plan is improving the `lfilt` and the `filtfilt` operation by implementing on lower level, considering contribution to the open source library.

Bibliography

[1] Martin Abadi et al. *TensorFlow: A system for large-scale machine learning.* Tech. rep. arXiv preprint. Google Brain, 2016. URL: `https://arxiv.org/abs/1605.08695`.

[2] Ronan Collobert, Koray Kavukcuoglu, and Clément Farabet. "Torch7: A matlab-like environment for machine learning". In: *BigLearn, NIPS Workshop.* EPFL-CONF-192376. 2011.

[3] *Complete Pan Tompkins Implementation ECG QRS detector - File Exchange - MATLAB Central.* URL: `https://www.mathworks.com/matlabcentral/fileexchange/45840-complete-pan-tompkins-implementation-ecg-qrs-detector` (visited on 05/09/2017).

[4] *CS231n Convolutional Neural Networks for Visual Recognition.* URL: `http://cs231n.github.io/` (visited on 05/08/2017).

[5] Yangqing Jia et al. "Caffe: Convolutional architecture for fast feature embedding". In: *Proceedings of the 22nd ACM international conference on Multimedia.* ACM. 2014, pp. 675–678.

[6] Eric Jones, Travis Oliphant, and Pearu Peterson. "{SciPy}: Open source scientific tools for {Python}". In: (2001). URL: `http://www.scipy.org` (visited on 05/09/2017).

[7] *Learning TensorFlow ::* URL: `http://learningtensorflow.com/index.html` (visited on 05/08/2017).

[8] Jiapu Pan and Willis J Tompkins. "A real-time QRS detection algorithm". In: *IEEE transactions on biomedical engineering* 3 (1985), pp. 230–236.

[9] *Performance Guide.* URL: `https://www.tensorflow.org/performance/performance_guide` (visited on 05/08/2017).

[10] *scipy/scipy.* URL: `https://github.com/scipy/scipy` (visited on 05/09/2017).

[11] *TensorFlow: How to optimise your input pipeline with queues and multi-threading.* URL: `https://blog.metaflow.fr/tensorflow-how-to-optimise-your-input-pipeline-with-queues-and-multi-threading-e7c3874157e0` (visited on 05/08/2017).

[12] *TensorFlow^{TM} Patterns*. URL: http://www.tensorflowpatterns.org/ (visited on 05/08/2017).

[13] *Tfrecords Guide*. URL: http://warmspringwinds.github.io/tensorflow/tf-slim/2016/12/21/tfrecords-guide/ (visited on 05/09/2017).

[14] *Tutorials*. URL: https://www.tensorflow.org/tutorials/ (visited on 05/08/2017).

[15] *WildML – AI, Deep Learning, NLP*. URL: http://www.wildml.com/ (visited on 05/08/2017).