vectorflow: a minimalist neural-network library

optimized for sparse data and low-latency

Benoît Rostykus
Netflix
brostykus@netflix.com

Yves Raimond
Netflix
yraimond@netflix.com

ABSTRACT

We introduce vectorflow[1], a minimalist neural-network library for single machine environment written in D optimized for sparse data and low-latency. The library offers a Directed Acyclic Graph with allocation-free and copy-free forward and backward propagations, making it particularly suited for IO-bound, sparse or low-latency machine learning problems. Good runtime performance is achieved through a combination of software leverage (compile-time language and LLVM features) and distributed optimization care (lock-free stochastic gradient descent).

1 INTRODUCTION

In the last few years we have seen many innovations on Open Source Machine Learning software. Most of these innovations (e.g. Tensorflow[2][1] or PyTorch[3]) evolved from fairly specialized computational code for large dense problems such as image classification into general frameworks for neural-network-based models offering marginal support for sparse models. A few frameworks are specifically optimized for sparsity though, such as Vowpal Wabbit[4] and Amazon's DSSTNE[5].

At Netflix, our machine learning scientists deal with a wide variety of problems across a broad spectrum of areas: from personalizing TV and movie recommendations to optimizing encoding algorithms. A subset of our problems involves dealing with extremely sparse data; the total dimensionality of the problem at hand can easily reach tens of millions of features, even though every observation may only have a handful of non-zero entries. For these cases, we felt the need for a library that is specifically optimized for training shallow neural networks on sparse data in a single-machine, multi-core environment, as well as quickly iterating on their architecture. We wanted something small and easy to modify, so we built vectorflow, a minimalistic (6k lines of code) neural network library, and one of the many tools our machine learning scientists use.

2 DESIGN

Dense problems such as image classification are typically *compute-bound*, where computation cost (e.g. dense convolutions) outweighs data transfer and memory allocation costs. For such problems, pass-by-copy, mini-batch training and GPU computation are particularly appropriate. However training shallow networks on sparse data is typically *IO-bound*; there will be relatively few operations to run per row, so data transfers and memory allocations become the bottleneck. This fundamental difference led us to a number of design considerations and technology choices.

[1]https://www.github.com/Netflix/vectorflow
[2]https://www.tensorflow.org/
[3]http://pytorch.org/
[4]https://github.com/JohnLangford/vowpal_wabbit
[5]https://github.com/amzn/amazon-dsstne

2.1 Sparse-aware

Vectorflow avoids, wherever possible, copying or allocating any memory during both the forward and backward passes, with each layer referencing the data it needs from its parents and children. Matrix-vector operations have both sparse and dense implementations, the latter ones being SIMD-vectorized thanks to LLVM intrinsics surfaced at the language level[6]. Vectorflow also offers a way to run a sparse back-propagation when dealing with sparse output gradients, useful for high-dimensional output problems such as auto-encoders trained on large vocabularies.

2.2 IO agnostic

As mentioned above, shallow neural networks trained on sparse data are typically IO bound. By definition in this case the trainer will run only as fast as the IO layer. Vectorflow enforces very loose requirements on the underlying data schema (merely to provide an iterator of rows with a “features" attribute) so that one can write efficient data adapters based on the data source and avoid any pre-processing or data conversion steps while sticking with the same programming language. This allows to move the code to the data, not the opposite. Data is mapped to the DAG input nodes at compile-time through introspection so that there is no runtime cost for the neural-network to access the input data.

2.3 Single-machine

Distributed systems are hard to debug and introduce fixed costs such as job scheduling. Implementing distributed optimization of a novel machine learning technique is even harder. This is why we created an efficient solution in a single machine setting, lowering iteration time of modeling without sacrificing scalability for small to medium scale problems. We opted for generic asynchronous SGD solvers using Hogwild [3] as a lock-free strategy to distribute the load over the cores with no communication cost (at the exception of cores cache lines invalidation by the CPU prefetcher [2]). This works for most linear or shallow net models as long as the data is sufficiently sparse, and provides near-perfect linear scalability with the number of cores. As an example, a sparse logistic regression model trained using vectorflow on ~30M rows with ~500 non-zeros per row (out of 10k features) takes 4s per epoch on a `r4.8xlarge`[7] Amazon EC2 instance and uses 75% of the RAM bandwidth of the machine.

[6]https://www.github.com/ldc-developers/ldc
[7]https://aws.amazon.com/ec2/instance-types/

	Model 1	**Model 2**	**Model 3**
training set size	5M	5M	10M
total input dimensionality	512	1M	1M
avg non-zero per row	7	16	16
# model parameters	513	1M	10M
# training cores	4	4	8

Table 1: Experiment set-up

	Model 1	**Model 2**	**Model 3**
download time	1.7s	2.5s	3.6s
pre-processing	12.1s	18.2s	36.1s
training time	1.6s	4.1s	1m25s
training AUC	0.67	0.73	0.75

Table 2: Results

2.4 Agility

We want our scientists to easily run and iterate on their models quickly and in total autonomy. We wrote vectorflow in D[8], a modern systems language with a very gentle learning curve. Thanks to its fast compilers and functional programming features, it offers a Python-like experience for beginners but with typically multiple orders of magnitude of performance gain at run-time, while enabling seasoned developers to leverage more advanced features, such as a templating engine, compile-time functionalities and lower-level features (C interface, in-line assembler, manual memory management, LLVM intrinsics, ...). Relying on a single language (as opposed to e.g. C and Python) also helps agility and flexibility; it enables experimentation with both low- (e.g. moving to a custom operation in an objective function) and high-level (e.g. modifying the input data) changes in a single model iteration. Vectorflow does not have any third-party dependencies, which eases its deployment. It also offers a templated callback-based API to easily plug-in custom loss functions for training, thus covering a wide range of supervised learning problems.

3 EXAMPLE APPLICATION

A few months after the project's inception, we've seen a wide variety of use cases for the library and multiple research projects and production systems are now using vectorflow for problems as diverse as causal inference, survival regression, density estimation or ranking algorithms for recommendation. It is also included in the default toolbox installed on basic instances used by Netflix machine learning practitioners.

As an example, we investigate the performance of the library on a marketing problem Netflix faces related to promoting our growing portfolio of original content. In this case, we want to perform weighted Maximum Likelihood Estimation with a survival exponential distribution [4]. To implement this, the custom callback function passed to vectorflow is:

[8]https://www.dlang.org

```
auto grad_surv = delegate float(float[] preds, ref Obs o,
     ref float[] grads)
{
  auto lab = o.label; // we can access any field of o
  double el = exp(preds[0]);
  float loss = void;
  if(!o.right_censor)
  { // uncensored part of the log-likelihood
    grads[0] = -(1.0 - lab * el);
    loss = -(pred - lab * el);
  }
  else
  { // censored part of the log-likelihood, gives us
       right-censoring point
    grads[0] = lab * el;
    loss = lab * el;
  }
  grads[0] *= o.weight; // MLE weighting
  return loss * o.weight; // optional: return loss value
       to monitor progress during training
}
```

Using this callback for training, we can easily compare 3 models:

- **Model 1:** linear model on a tiny set of sparse features ($\tilde{5}00$ parameters to learn);
- **Model 2:** linear model on a larger sparse set of features (1M parameters to learn);
- **Model 3:** shallow neural network on a sparse set of features (10M parameters to learn), trained on twice the data.

```
auto nn1 = NeuralNet()
  .stack(SparseData(512))
  .stack(Linear(1));
auto nn2 = NeuralNet()
  .stack(SparseData(1_048_576)) // 2^20 features
  .stack(Linear(1));
auto nn3 = NeuralNet()
  .stack(SparseData(1_048_576))
  .stack(Linear(10))
  .stack(SeLU())
  .stack(Linear(1));
```

The data source described in Table 1 is a Hive table stored on S3 using the columnar data format Parquet and we train directly against this data by streaming it to a `c4.4xlarge` instance and building in-memory the training set which we learn from. The results are presented in Table 2.

4 CONCLUSION

In this paper, we presented vectorflow – a minimalistic neural network library optimized for training shallow networks on sparse datasets. We described various design considerations which arise from this use-case, as well as an example application.

In the future, we plan to broaden the possible topologies supported beyond simple linear, polynomial or feedforward architectures, develop more specialized layers (such as recurrent cells) and explore new parallelism strategies while maintaining the "minimalist" philosophy of vectorflow.

REFERENCES

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Gregory S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian J. Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Józefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Gordon Murray, Chris Olah, Mike Schuster,

Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul A. Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda B. Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. *CoRR* abs/1603.04467 (2016). arXiv:1603.04467 http://arxiv.org/abs/1603.04467

[2] Christopher De Sa, Matthew Feldman, Christopher Ré, and Kunle Olukotun. 2017. Understanding and Optimizing Asynchronous Low-Precision Stochastic Gradient Descent. *SIGARCH Comput. Archit. News* 45, 2, 561–574. https://doi.org/10.1145/3140659.3080248

[3] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. 2011. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in neural information processing systems*. 693–701.

[4] G. Rodriguez. 2010. *Survival Models*. Technical Report. http://data.princeton.edu/wws509/notes/c7.pdf