CS230 Project Final Report
Experimental Neural Net Framework

Student: *Konstantin Burlachenko* (bruziuz@stanford.edu, burlachenkok@gmail.com)
Project Mentor: *Steven Ziqiu Chen* (stevenzc@stanford.edu)

Abstract

In my work I created infrastructure in C++11 for making inference and learning for *Feed-Forward Neural Nets*. Project contains all need source code and does not contain any extra dependencies. You can take this project and compile for your target embedded device with **Windows** or any **Posix OS (like Linux)** and perform inference and learning on it.

Project can be builded if your CPU support SSE2 or AVX2 CPU instruction extensions. But even if you CPU do not contain such instructions and you have C++11 compiler then you can build this project.

If comparing Learning time for *Feed-Forward Neural* net with RELU(or RAMP) activation function and via using *RMSprop* as optimization algorithm then this work lead to **x4** faster computations.

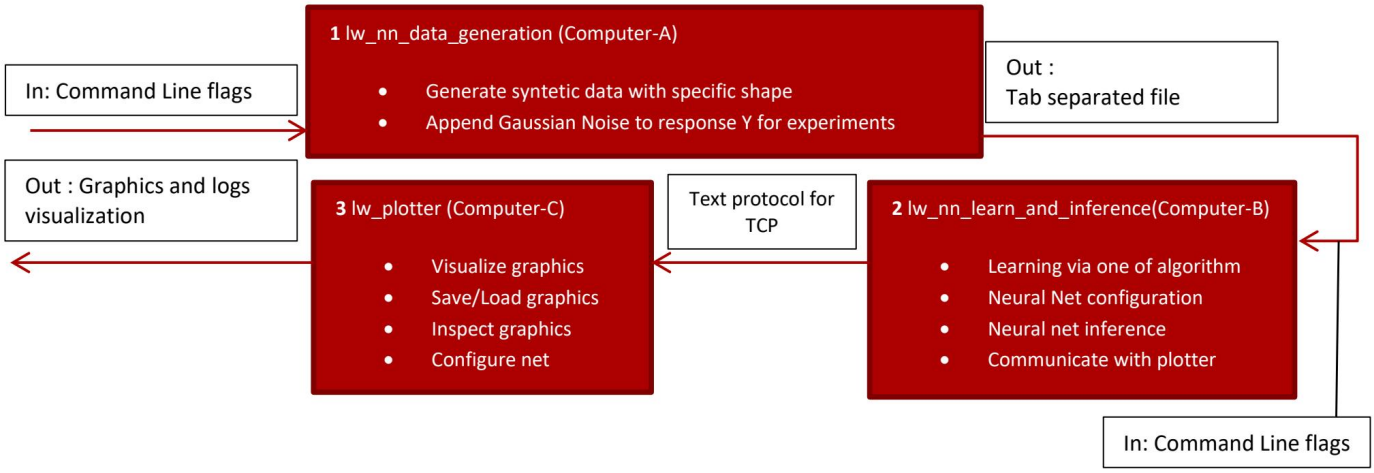
1. Introduction

In world of Machine Learning *Python* is most popular language for prototyping. This language has different benefits especially if compare with other scripting languages ([1]).

But in fact when algorithms are known and established it can be a case that *Python* (and any script language like *Matlab*) is not the best choice if concentrating more in some production quality software. One intermediate step from *Python* which eliminate some drawback of programming language is using *Cython*, but it's only intermediate step.

In my opinion once algorithm have been established and we need speed then it's time to use C++. Using of C++ lead to two important benefits – some errors compiler will catch during compilation and the result is optimized binary file contains instruction for target CPU, without any intermediate byte-code (which exist in *Python, Java, C#*). Cross-platform aspects of the programming environment in my opinion should lie in area of unifying interfaces to Operation System and Drivers. In my work I created infrastructure in C++11 for making *inference* and *learning* for deep neural nets.

AI (Artificial Intelligence) expand and go into different areas. In future people will create more complicated models then they do nowadays they will need such the most possible effective tools including tool for *deep neural nets*. At this phase of development I created Light Weight Experimental NN Framework with only need things. Experimental Light Neural Net Framework consist of several parts, but top level schema is the following:



2. Previous Works

There are exist several popular framework for Deep Learning. **Keras** is an open-source neural-network library (Written in Python). **TensorFlow** is a free and open-source software library for dataflow and differentiable programming across a range of tasks (Written in Python, C++, CUDA). Examples of other popular frameworks are: Caffe (UC Berkeley), Caffe2 (Facebook), Torch (NYU / Facebook), PyTorch (Facebook), Theano (U

Montreal), PaddlePaddle (Baidu), MXNet (Amazon), CNTK (Microsoft). Standard frameworks allow test new ideas and allow automatically compute gradients. Unfortunately this frameworks hide how they do what they do. In case of big data It is important because in that case you should think twice before perform any not-usefull memory copyinh or computations.

Another interesting project is *DartNet* https://pjreddie.com/darknet/. It's project from author of *YOLO* and it was written in C. But scale C language project is not an easy task.

3. Dataset and Features

Dataset is synthetically generated via using *lw_nn_data_generation* program [5]. Program can be launched as:
lw_nn_data_generation --dbg_plot_address 127.0.0.1 --dbg_plot_port 4545 -n 10 -m 2

It mean generate test data set with number of samples equal to 10 and number of features equal to 2. Formula which encode response is hardcoded in code and in current implementation it is:

$$y = 1^T x + 1^T \log(|x| + \mathbf{1}), x \in R^n$$

After having generated test set you can use several python scripts to inspect content and make some operations
https://bitbucket.org/bruziuz/comp-graph-eco/src/master/src/lw_neural_net_framework/tools/lw_nn_data_scripts/

Script name	Reason of existing
lw_nn_data_info.py	Report statistic information about data
lw_nn_holdout_cv.py	Split set into to disjoint sets for holdout cross-validation
lw_nn_read_top_n_samples.py	Read top n samples. First argument is filename, second is number of examples

4. Methods

Methods for making inference and learning implemented in **lw_nn_learn_and_inference** console program.

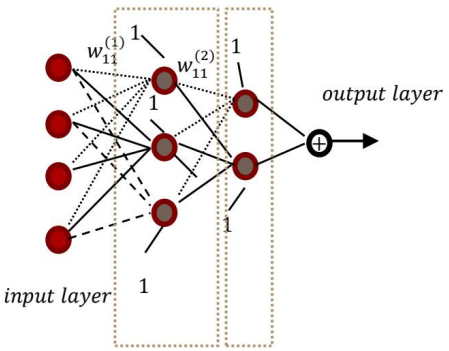
lw_nn_learn_and_inference learn --dbg_plot_address 127.0.0.1 --dbg_plot_port 4545 --in D:/train.txt --in_train_fraction 0.8

4.1 A bit history

Neural Nets approach in context of using backpropagation have been published in 1986 by Hinton, Rumelhart and Williams. This days Neural Networks come back, even in past they have difficult history.

4.2 A picture for multilayer feed-forward neural-net and Feed Forward

Classical multi-layer network pictorial can be represented as neurons connected in "*cascade style*"



$l = 1$ $l = 2$

$(1st\ layer)$ $(2nd\ layer)$

It's only picture. From equation point of view the Neural Net can be considered as a long nested composition of apply affine transformation for input $Z = Wx + b$ and after apply affine transformation apply some non-lineariry g. If talking more speicifically then :
$\boldsymbol{Z^{[1]} = W^{[1]}x + b^{[1]}}$
$\boldsymbol{A^{[1]} = g(Z^{[1]})}$
This two equations show how input $\boldsymbol{x}$ is maps into **A**, i.e. this setups mapping $\boldsymbol{X \rightarrow A^{[1]}}$. During apply this equation in layers after input layer then recurrent relation is the following:
$\boldsymbol{Z^{[l]} = W^{[l]}A^{[l-1]} + b^{[l]}}$
$\boldsymbol{A^{[l]} = g(Z^{[l]})}$

This elementary operations is sequentially applied in long chain list up to output layer where finally typically some final happens
$\boldsymbol{Out = A = g(Z)}$. Typically last g can be identity mapping for regression, softmax block for mulit-class classification, series of logisitic function for multi-class classification but when it is possible that several classes are represented in input $\boldsymbol{X}$.

Parallel computation is possible for all nodes in one layer because they all are completely independent during forward phase (and backward phase too).

4.3 Big picture why back-propagation is important

Neural Network can be considered as a big directed acyclic graph. Nodes in one level receive inputs only from nodes from previous level and affect directly only to nodes in the next layer. Purpose of backpropagation is not to recalculate already known intermediate Jacobians. In some sense I think it can be stated that Backpropagation is clever way to apply chain-rule. I have derived backpropagation rule in [7]. My derivation are not so well applicable because they has more theoretical characteristics and really bias(or intercept) term is not so well fitted into it. I come to decision during my work in software use recurrent relation which I studied during CS230 class ([8]) in first part dedicated for classical *Neural Nets*. The derivation is out of scope of this note.
Backpropagation allow to fastly compute gradient of the Loss function $L(F(\widehat{x;w}), y)$ with respect to w (most popular scenario, even it can be a case that you're interesting in gradient w.r.t. to ***x***) . In my work I implemented mini-batched variant of back-propagation algorithm.

4.4 About speedup first-order methods implemented in work

In Machine Learning there is a great trick to evaluate gradient not for complete loss function, which is objective to minimize, but evaluate it via compute it for each observation or for some subset (**mini-batch**). Whole pass over the whole data whatever strategy we selected called **"epoc"**. If during last **"epoc"** parameters has not changed then it means that algorithm converged. People in Neural Network talk about 400-500 epochs as a typical number.

One improvement about which ML community think about – is use convex combination with factor *alpha* to select convex combination with previous gradient and new. And at the beginning of each epoc make zero it. In fact there are a lot of schemas how to make smooth search direction first-order method which have been founded in are of mathematics called *Convex optimization*. Methods implemented in ENNF:

1) Stochastic Gradient Descend: $x^{k+1} = x^k - a_k g^k, a_k > 0$. I learned from EE364B slides of S.Boyd and J.Duchi [11]. In [10] S.Boyd mentioned that all Subgradient methods have been developed in USSR in 1960 and 1970.

2) Heavy Ball method: $x^{k+1} = x^k - a_k g^k + \beta_k (x^k - x^{k-1}), a_k > 0, \beta_k > 0$ subgradient with state. Boris Polyak refers to it as the **Heavy Ball method**. The role of second term is momentum.

3) Momentum: $s^k = (1-\beta)g^k + \beta s^{k-1}$

4) RmsProp: $\Sigma^k = decay(\Sigma^{k-1}) + (1 - decay)(\mathrm{g}^k)^2$ and $s^k = \frac{\mathrm{g}^k}{(\sqrt{\Sigma^k} + eps)}$

5) Adam. Adaptive Moment Estimation. This algorithm in fact combine of Momentum and RMSProp.

$m_1 = decay(m_{1,prev}) + (1 - decay)\mathrm{g}^k$ (Momentum for g^k)

$m_2 = decay(m_{2,prev}) + (1 - decay)(\mathrm{g}^k)^2$ (Momentum for $(\mathrm{g}^k)^2$)

$m_1^{unbias} = \frac{m_1}{(1-{\beta_1}^t)}$ (Bias correction) **and** $m_2^{unbias} = \frac{m_2}{(1-{\beta_2}^t)}$ (Bias correction) **and** $s^k = \frac{m_1^{unbias}}{(\sqrt{m_2^{unbias}}+eps)}$

4.5 About regularization technics imlemented in work

Regularization *has the following meaning in optimization community: "Regularization is a common scalarization technic for solve problem with two objectives"* ([11], p.306). This word have different flavor in machine learning. In ML this days it means any activity "*not perfect*" fit train data. In work I considered several regularization techics.

1. **Early stopping.** Dividing learn set into learn/dev set (90%/10%). Idea to stop when validation goes up. There is no guarantee that it will not came back, but it's very good heuristic to stop.
2. **Inverted Drop out.** During forward and backward propagate half of the network nodes *output* is setuped to zero. It's the same as this nodes are completely removed during phase. The schema is to *out* with some probability "p" some neurons in the whole computation graph.
3. **Extra regularization term for L2 norm square and L1 square in minimize objective**
   Because problem that we solve is non-convex then to be within a room of heuristics applicability of first order method for non-convex optimization problems we really can append anything which encode prior information and has (sub)gradient.

Experiments and Results

To evaluate success of the project I will use the several metrics. First metric is number of lines of code.

```
-------------------------------------------------------------------------------
Language                     files          blank        comment           code
-------------------------------------------------------------------------------
C/C++ Header                    55           1820           3354           8702
C++                             83           1450            271           7911
Python                          12            156            104            485
CMake                            9             92            101            388
YAML                             1              5             20             41
IDL                              1              9              0             36
DOS Batch                        2              5              0             19
Bourne Shell                     1              1              0              3
-------------------------------------------------------------------------------
SUM:                           164           3538           3850          17585
-------------------------------------------------------------------------------
```

Second to estimate result of work I created some specific architecture and compare learning time for my ADAM solver during learning with ADAM solver used in TensorFlow v1.13.1

For example learning during 100 epocs based on train data and script from:
https://bitbucket.org/bruziuz/comp-graph-eco/src/master/src/lw_neural_net_framework/tests/tf_baseline/tf_test.py
https://bitbucket.org/bruziuz/comp-graph-eco/src/master/src/lw_neural_net_framework/tests/tf_baseline/tf_test.py
Lead to following results:

Method / Approach	CPU TensorFlow v1.13.1	Experimental Neural Net Framework
ADAM optimizer	1.115 seconds	0.306
SGD optimizer	1.105 seconds	0.298

I tried to be extremely honest and really I did not take into account al initialization time for TensorFlow.
Also in TensorFlow there a calculation of cost for minibatch. For ENNF to mimic this I evaluated MSE over train set completely. Really there is a room for decrease numbers for ENNF even more.

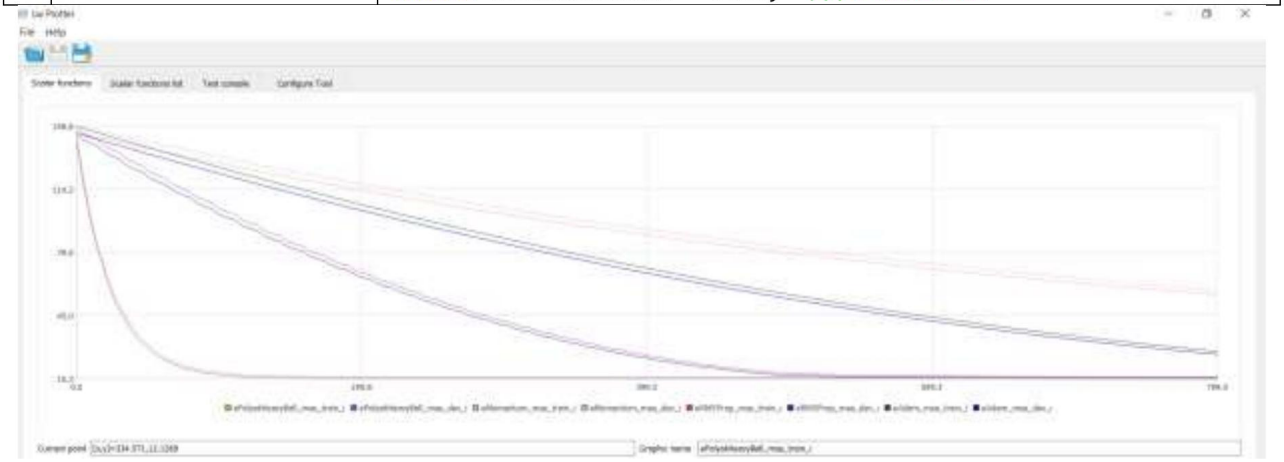
Convergence to the same result as cvxpy and sklearn gave as baseline. The last thing I would like to share my experimental evaluation for different optimization algorithms. From this particular experiment I launch for 50 epocs various optimization algorithm in the Neural Net with computation schema:
LINEAR -> RELU -> LINEAR -> RELU -> LINEAR -> RELU -> LINEAR. **Minibatch** size is 10.

Input layer contain 30 neurons, output layer contains 1 neuron. Use loss is Mean Square Error Quartatic loss. There are 3 hidden layers and number in neuron in this layers are: 5, 5, 3. In my particular example.

#	Method	Params
1	Polyak Heavy Ball method	`double kAlphaPolyak  = 0.01;  ///< Polyak Heavy Ball` `double kBetaPolyak  = 0.0001;  ///< Polyak Heavy Ball`
2	RMSPRop method	`double kAlphaRmsProp = 0.01;  ///< RmsProp` `double kBetaRmsProp = 0.95;  ///< RmsProp`

3	ADAM method	`double kAlphaAdam = 0.01;  ///< ADAM`
		`double kBeta1Adam = 0.9;  ///< ADAM`
		`double kBeta2Adam = 0.99;  ///< ADAM`

4	Momentum method	`double kAlphaMomentum = 0.01; ///< Momentum` `double kBetaMomentum = 0.95;  ///< Momentum`



Conclusion:

1. Most of smooth methods are developed for speed-up convex optimization problems. It's rather hard to say how method will behave in some specific circumstances
2. Because Theory is developed behind this methods can give guarantees how method behaves for convex optimization problems, it's really can say nothing in specific circumstances how method behaves
3. In this method Heavy Ball Boris Polyak's method was lucky

Conclusion and Future Work

The need elements of software have been successfully implemented, code is documented, and big part of code is covered by unit-tests. I think work can be estimated as completed successfully.

Future directions of work

1. Append ability to insert arbitrarily expression in *lw_nn_data_generation* for have ability in automatic way test how good Deep Neural Net can approximate some functions with specific algebraic form.
2. Append support of Convolution Neural Nets
3. Append support of showing image in plotter tool.
4. Append support of visualizing configured Neural Net via **graphviz**
5. Try use YOLO: Real-Time Object Detection to have more fast algorithms for training and inference
6. Append NVIDIA CUDA support. I started make strps in this direction, but really there are still a lot of work
7. My current vision that in future instead having several methods and select which to use a-priori it can be better to have several methods and switch from them once method was stucked for several iterations.

References

[1] Table with compare some programming langugages
https://sites.google.com/site/burlachenkok/compare_programming_laguages

[2] Cython documentations https://cython.readthedocs.io/en/latest/index.html

[3] Instruction for final project http://cs230.stanford.edu/files/formatting-instructions-cs230.pdf

[4] CS230 requirements for project http://cs230.stanford.edu/project/

[5] Tool to generate syntetic dataset https://bitbucket.org/bruziuz/comp-graph-eco/src/master/src/lw_neural_net_framework/tools/lw_nn_data_generation/CMakeLists.txt

[6] Scripts perform some operations on data https://bitbucket.org/bruziuz/comp-graph-eco/src/master/src/lw_neural_net_framework/tools/lw_nn_data_scripts/

[7] Backpopagation derivations https://bitbucket.org/bruziuz/comp-graph-eco/src/master/docs/lw_framework_theory_working_document.docx

[8] CS230 class from Stanford University http://cs230.stanford.edu/syllabus/

[9] Notes about Stochastics Gradient Descend from S.P.Boyd and J.C.Duchi

http://web.stanford.edu/class/ee364b/lectures/stoch_subgrad_notes.pdf

[10] Subgradient Descend for Convex Optimization from S.P.Boyd

http://web.stanford.edu/class/ee364b/lectures/subgrad_method_notes.pdf

[11] Convex Optimization, S.Boyd and L.Vandeberghe

http://stanford.edu/~boyd/cvxbook/bv_cvxbook.pdf

[12] Repository with whole project including documentation and source code

https://bitbucket.org/bruziuz/comp-graph-eco/src/master/