EMPIRICAL EVALUATION OF OPTIMIZERS ALGORITHMS USING RECURRENT NEURAL NETWORKS FOR FORECASTING TASKS ON TIME-SERIES DATA

Andres Camilo Zuleta - 201755622

1.) INTRODUCTION

In the last years more and more problems have been solved using Machine Learning and Deep Learning techniques due to the develop of new, more complexes and more optimal architectures for Artificial Neural Networks (ANN) [3]. Also, the use of Deep Learning algorithms is becoming more popular thanks to their use of unsupervised learning rather than the classic supervised learning approach provided for common machine learning, which allows the ANN to discover by itself the relevant features of a specific data set instead of having them provided as an input [4]. Thanks to this feature a lot of very complex problems such as Human-Face-Recognition [5], Winning in Go [6] and predict values for Time-Series [3] are being solved with proficient results.

The objective of this paper is to compare empirically the performance of three of the most popular gradient optimizer algorithms for ANN applied to recurrent neural networks (RNN), a variant of typical ANNs and determine which optimizer seems to be the most suitable from an empirical perspective. To achieve that, first we will review some literature work to get insights about which of this algorithms seems to be the best one from a theoretical approach and why, then we will show the results of the experiments made to see which optimizers behave the best for the proposed data set and finally we will compare if our results match with the theory.

2. LITERATURE REVIEW

Nowadays massive amounts of data are produced and collected through the internet. This data can be since exchange currency rates until information about geo-physical activity, usually this data is archived and measured taking into account the past of the time and this is what we known as a Time Series. Analysis of Time-Series data have always been an important topic for fields like Economics or Engineering. At the beginning those analysis were conducted using statistical techniques such as ARIMA models (Autoregressive integrated moving average), classic ANNs or even Hybrid approached using both of them [3]. Those classic approach are being deprecated by the best results who can be achieved through the use of Deep learning techniques and advanced types of ANN such as Convolutional Neural Networks (CNN) or Recurrent Neural Networks (RNN). The last ones are pretty common on this kind of analysis because of their good behavior and good results, especially for forecasting tasks [3], reason why they are widely used in the analysis of this kind of data.

Additionally, the use of ANNs shown to usually obtain better and more reliable results than the ones provided from ARIMA, for example for financial data such as foreign currencies exchanges rates [1] or prediction of stock prices [7]. Furthermore, recent advances in the field of Deep Learning have develop new and better architectures that enhance the results obtained

by RNN such as the Long Short Term Memory or Gated Recurrent Unit Architectures [2] which outperforms other existent solutions for RNN.

The last mentioned of these networks, RNN, are going to be the focus of our study because we found it interesting because they are currently very state of the art. However, even though data scientist have now a powerful tool that helps them to obtain the desire goals, such as the capability of a model to predict values over the time, there are still a lot of factors to take into account who can improve the performance of a model, such as the selection of parameters and an appropriate optimizer. And even though advances have been made into the optimizers area, resulting into new and innovative algorithms such as the adaptive optimizers [9] the idea is to select two of these new algorithms and compare them with a classic optimizer to see how much have we advanced.

3. METHODS

Taking into account what we saw in the previous section we decided to select the following method to do our experiments: A RNN with a LSTM Architecture. And the chosen optimizers to experiment with will be the SDS, Adagrad and Adam.

Now, in order to facilitate the understanding of our experiment we need to briefly explain technically each one of the components that we are going to use.

3.1 RECURRENT NEURAL NETWORK

A recurrent neural network (RNN) is an extension of a conventional feedforward neural network, which is able to handle a variable-length sequence input. The RNN handles the variable-length sequence by having a recurrent hidden state whose activation at each time is dependent on that of the previous time. [2] Formally given a sequence $X = (x_1, x_2, \dots, x_t)$, the hidden state h_t is updated by:

$$h_t = \begin{cases} 0, & t = 0 \\ \Phi(h_{t-1}, x_t), & \text{otherwhise} \end{cases}$$

where Φ is a nonlinear function such as composition of a logistic sigmoid with an affine transformation. Optionally, the RNN may have an output $Y = (y_1, y_2, \dots, y_t)$. which may again be of variable length.

3.2 LONG-SHORT TERM MEMORY

As we saw, Vanilla RNNs work well for short term memory values as they can handle the state of previous outputs. However when it comes for long term memory they do not behave so well, especially if we want to make Deep RNN by stacking Recurrent Units. The problem is that the gradient of the weights starts to become too small or too large if the network is unfolded for too many time steps, ending in what is called the *vanishing gradients* problem [3].

In order to solve this problem we may use the Long-Short Term Memory architecture. This architecture was proposed by Hochreiter and Schmidhuber [12] and it overrides the Vanilla RNN by treating each node as a cell which now have its own state variable *C* that is modified by either one of three *Operation Gates* :

- *Forget Gate:* The idea of this gate it is to determine whether or not we are going to throw away the information from the previous cell state. This choice is made by a sigmoid layer that takes the output from the previous cell and outputs a value between 0 and 1 which is multiplied by the internal state as shows the following Formula.

$$f_t = \sigma(W_t \bullet [h_{t-1}, x_t] + b_t)$$

- *Input Gate:* The idea of this gate is to determine which new features and how much of them are we going to introduce into the internal state of the cell. In order to do that this gate is divided into three steps. First step works very similar to the forget gate and consists of a sigmoid layer with the previous output and current input as parameters, the second step is the value of the candidate layer that works in a hyperbolic tangent layer instead of a sigmoid function, finally the values returned by the first and second step are multiplied and that is the result of the input gate.

$$i_t = \sigma(W_i \bullet [h_{t-1}, x_t] + b_i)$$
$$\check{C}_t = tanh(W_C \bullet [h_{t-1}, x_t] + b_C)$$
$$C_t = f_t * C_{t-1} + i_t * \check{C}_t$$

- *Output Gate:* Finally we have this gate which purpose is to determine how much of the internal state is going to be passed to the output of the cell. This gate works in a similar way to the other gates, with a sigmoid layer to determine how much of the features to take into the output and then multiplying that result with an hyperbolic tangent function over the candidate layer features as shown below.

$$o_t = \sigma(W_o \bullet [h_{t-1}, x_t] + b_o)$$
$$h_t = o_t * \tanh(C_t)$$

3.3 STOCHASTIC GRADIENT DESCENT

Stochastic Gradient Descent (SGD) is one of the most commonly used learn algorithms for ANN. It consists of a variation of the classic Batch Gradient Descent algorithm, but instead of updating the parameters just one time, SGD performs a parameter update for each training example $x^{(i)}$ and label $y^{(i)}$ [9]. This algorithm requires a parameter θ as all the listed algorithms does, and also requires a learning rate ϵ_k. The way this algorithm works is explained in the following pseudo-code:

While stopping criterion not met **do**

- Sample a minibatch of m examples from training set $\{x^{(1)}, \dots, x^{(m)}\}$ with corresponding targets $y^{(i)}$.
- Compute gradient: $g = \frac{1}{m} \nabla_\theta \sum_i L(f(x^{(i)}; \theta), y^{(i)})$
- Apply update: $\theta = \theta - \epsilon_k g$

end while

However, because this algorithm depends on a vanilla minibatch process it have some problems such as choosing the proper learning rate or getting stuck on suboptimal local minimums. Even if is true that out there exists some alternative and improved versions of SGD that takes into account more components such as momentum, we decided to use the basic SGD algorithm to obtain a better contrast between the results obtained by this algorithm and the other ones.

3.4 ADAGRAD

Adagrad which stands for adaptive gradient is a learning algorithm that exactly what his name suggest. It adapts the learning rate to the parameters, performing larger updates for infrequent and smaller updates for frequent parameters. Consequently this algorithm is one of the best suited for sparse data [9]. The idea of Adagrad is to use different learning rates ϵ_k for every parameter θ_i at every step of time t. This algorithm need as required parameters a global learning rate ϵ (which will be adapted), an initial parameter θ, a Small constant δ of arbitrary value (literature recommends 10^{-7}) [13] and to initialize the accumulation variable r at 0 ($r = 0$). The way this algorithm works is explained in the following pseudo-code.

While stopping criterion not met **do**

- Sample a minibatch of m examples from training set $\{x^{(1)}, \dots, x^{(m)}\}$ with corresponding targets $y^{(i)}$.
- Compute gradient: $g = \frac{1}{m} \nabla_\theta \sum_i L(f(x^{(i)}; \theta), y^{(i)})$
- Accumulate squared gradient: $r = r + g \odot g$
- Compute update: $\Delta\theta = -\frac{\epsilon_k}{\delta + \sqrt{r}} \odot g$
- Apply update: $\theta = \theta + \Delta\theta$

end while

One of Adagrad's main benefits is that it eliminates the need to manually tune the learning rate. However one of his main weakness is its accumulation of the squared gradients in the denominator. For the previous reason, sometimes the learning rate shrinks and eventually become infinitesimally small, causing the algorithm to no longer be able to acquire additional knowledge. Some algorithms like Adadelta [14], RMSProp or variation or these algorithms (such as ADAM which we will explain soon) tries to solve this problem.

3.5 ADAPTATIVE MOMENT ESTIMATION

Adaptive moment estimation or ADAM is another adaptive learning algorithm like Adagrad. Additionally ADAM extracts ideas from other algorithms like AdaDelta or RMSProp by storing an exponentially decaying average of past squared gradients and keeps ρ_1 and an exponentially decaying average of past gradients ρ_2 like momentum In order to use this algorithm we need to initialize the variables for Step size ϵ (literature suggest a value of 0.0001) , ρ_1 and ρ_2 in the interval [0, 1] (literature suggests 0.9 and 0.999 respectively) and finally and small constant δ with a suggested value of 10^{-8} . The algorithm is explained in the following pseudo-code.

While stopping criterion not met **do**

- Sample a minibatch of m examples from training set $\{x^{(1)}, \dots, x^{(m)}\}$ with corresponding targets $y^{(i)}$.
- Compute gradient: $g = \frac{1}{m} \nabla_\theta \sum_i L(f(x^{(i)}; \theta), y^{(i)})$; t = t + 1
- Update biased first moment estimate: s = $\rho_1 s$ + (1 - ρ_1)g
- Update biased second moment estimate: : u = ρ_2r + (1 - ρ_2)g ⊙ g
- Correct bias in first moment : š = $\frac{s}{1-\rho_1^t}$
- Correct bias in second moment û : = $\frac{u}{1-\rho_2^t}$
- Compute update: Δθ = $-\frac{\check{s}}{\delta+\sqrt{\hat{u}}}$
- Apply update: θ = θ + Δθ

end while

This algorithm is the most sophisticated and complex of all the above algorithms. In theory due to its adaptive behavior we should get good results in practice with spare data sets with only the default value for the learning rate. Also this algorithm should outperform the other algorithms. However that's what we are going to confirm in the next experiment.

4. EXPERIMENT SETUP

In order to prove which optimizer will bring us the best results we setup three different experiments for two different datasets as we will explain soon.

4.1 Dataset 1 – Retail trade values

The idea of this dataset was to test how the three optimizers behave on a data set with sparse data, (roughly more than 300 records) and their capability to predict values from it.

This first dataset consists of data from the Retail trade sales by the North American Industry Classification System (NAICS) from January of 1991 until January of 2017, this data was extracted from Statistics Bureau of Canada.

First we defined the data before January of 2011 as our training data or 'past' and any further data from this point to January of 2017 was defined as the test set as shown in fig (1).

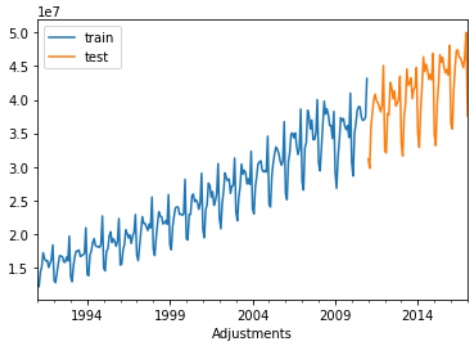

fig (1).

Additionally, this data was modified using the pandas library so the RNN used could manage a time window of up to (t – 12) in order to improve the performance, so each record has information of the complete previous year of retail prices, so we ended up with a sum of 463 parameters for the training set.

Finally the model to use was a RNN with an LSTM architecture with 6 nodes, an input shape of (1,12) with the Mean Square Error (MSE) and 100 epochs per execution with capability of early stop to prevent overfitting as the loss function as shown in fig (2).

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lstm_1 (LSTM)                (None, 6)                 456
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 7
=================================================================
Total params: 463
Trainable params: 463
Non-trainable params: 0
_________________________________________________________________
```

fig (2).

This experiment consists of three different executions of the same model varying the used optimizer (SGD, Adam and Adagrad), store it respectively results, particularly execution time and loss value and finally compare them using a confidence interval of 95%. Due to time and computational constraints (experiments were made on a personal computer and each iteration of the experiment took about 15 minutes) the number of iterations were limited to 20.

4.2 Dataset 2 – Number of daily births in Quebec

The idea of this dataset was to test how the three optimizers behave when they are trained using a dataset with sufficient amount of data to learn from (approximately more than 5000 records).

This first dataset consists of the number of daily births in Quebec from January first of 1977 until December 31 of 1990. This data was extracted from Datamarket.com [15].

First we defined the data before January first of 1988 as our training data or 'past' and any further data from this point to December 31 of 1990 was defined as the test set as shown in fig (3).

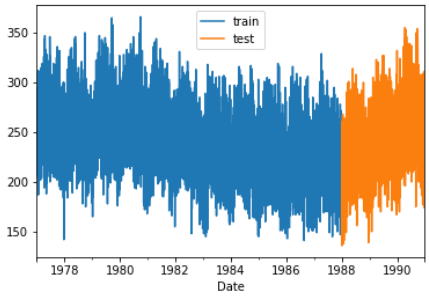

fig (3).

Additionally, this data was modified using the pandas library so the RNN used could manage a time window of up to (t – 30) in order to improve the performance.

Finally the model to use was a RNN with an LSTM architecture with 10 nodes, an input shape of (1,29) with the Mean Square Error (MSE) and 20 epochs per execution (because executions were taking around 2 – 3 hours with 100 epochs) with capability of early stop to prevent overfitting as the loss function as shown in fig (4).

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lstm_1 (LSTM)                (None, 10)                1600
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 11
=================================================================
Total params: 1,611
Trainable params: 1,611
Non-trainable params: 0
_________________________________________________________________
```

fig (4).

Similarly to the previous experiment, this one consists of three different executions of the same model varying the used optimizer and compare the results using a confidence interval of 95%. Due to time constraints the number of iterations were limited to 8.

5. RESULTS

5.1 Results for Dataset 1

Fort the first dataset we obtained the following results fig (5) when plotting the actual values of the test set (blue) vs the predicted values (orange).

fig (5).

This would give us the notion that Adam and Adagrad outperform SGD and have a better capability to predict values. However, sometimes for the same dataset we obtained results as shown in fig (6):

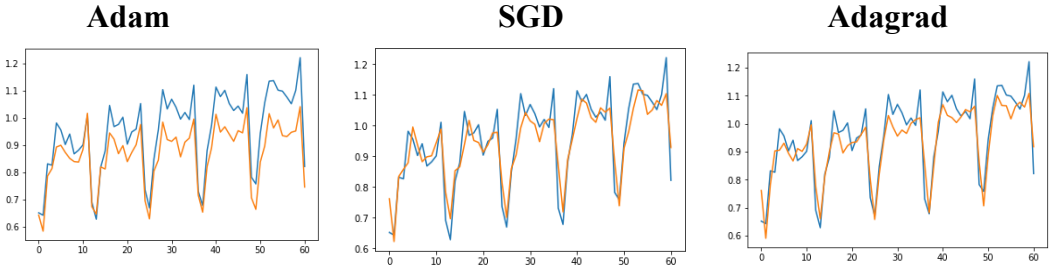

fig (6).

Sometimes the performance of SGD would randomly be excellent, even surpassing Adam or Adagrad, making the selection of the best algorithm among those three a hard one only by seeing those graphics. However, when observing the results obtained from the confidence interval for Total Execution Time and Final Lost for a confidence of 95% fig (7) we can clearly observe that Adam is the best algorithm, not only because is the one that usually can give us the minor final lost but also because its execution time is always inferior compared to the other two algorithms. On the other hand Adagrad also performs well and even have a more stable interval of values than Adam (which have a greater set of high values for the lost), but its execution time is longer, more than double the time it takes to Adam to execute. For those reasons we considered that for the case of this small dataset the best Optimizer would be Adam.

Total Execution Time Confidence Interval		
Adam	Adagrad	SGD
[66,2846372 ; 80,0153628]	[165,8620736 ; 195,1379264]	[121,6130053 ; 167,0869947]

Final Lost Confidence Interval		
Adam	Adagrad	SGD
[0,00147616 ; 0,00244384]	[0,001891231; 0,002128769]	[0,004356035; 0,010503965]

fig (7).

Finally we want to say that the previously strange behavior for the SGD algorithm which apparent to be a random one, is not. When consulting the experiments results table, attached to this document, we can find that whenever the final lost was high when using the SGD algorithm we also present low execution times, the reason behind that is because an early stop caused because a non-improvement of the model. This can be caused because SGD algorithm got stucked on a suboptimal minimum during that execution which is one of the main troubles this algorithm have as we explain before in previous sections. So basically SGD is an algorithm that does not performs very well for an sparse data set, while Adam and Adagrad performs well you could go with Adam for lowest values of the lost and quick executions times or Adagrad for more stable results of the lost but slowest execution times.

5.2 Results for Dataset 2

In a similar way to the previous experiment we plotted on of the iterations to see the behaviour of our algorithms fig (8)..

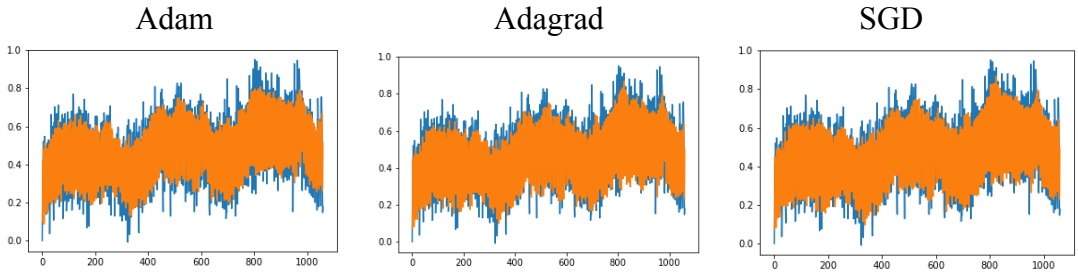

fig (8).

That behavior remained very similar for each one of the iterations, which were limited to 8 due to time constraints (each epoch was taking around 1 minute and the algorithms were using all the available epochs without early stopping). For that reason using a visual approach to determine which algorithm was performing the better was difficult. However, the results for the confidence interval for this experiment helped us to determine that fig (9).

Total Execution Time Confidence Interval		
Adam	Adagrad	SGD
[648,3219161 ; 752,1780839]	[638,4178273 ; 691,0821727]	[624,8858569 ; 694,6141431]
Final Lost Confidence Interval		
Adam	Adagrad	SGD
[0,008451374 ; 0,008598626]	[0,008805018 ; 0,008869982]	[0,008923029 ; 0,009026971]

fig (9).

In this case it is clear that Adam is the algorithm that performs obtaining the best results for the lost function. However, in contrast to the previous experiment, in this case Adam

is also the algorithm that performs with the slowest execution time among all of them. Also we can see that SGD and Adagrad obtain very similar results in both the lost function and execution time, maybe Adagrad does it better for results on the lost function and SGD is slightly better in execution time. In reality, in this case the performance of both the three algorithm is very similar and there is not an universal law to determine which of them would do it better, if minimizing the lost function without caring about the time is your goal then you should go for Adam, if you want decent results in a relatively low amount of time then you should go with SGD and if you want a balance between both time and lost then you should pick Adagrad. Also we have to take into account that the performance of this algorithms can be better if we let them execute more epochs, the main objective of this experiment was to do a contrast between the behavior of the algorithm when they use a large dataset rather than compare their behavior between them.

6. CONCLUSIONS

We have done a series of empirical experiments, and taking into account the previous results then we can say that when talking about optimizers algorithm for RNN models there is no such thing as a holy grail or a direct perfect answer of which optimizer select. However, there is such an intuition that we can use in order to determine which optimizers definitively we should not choose in determinate scenarios and which optimizers would probably allow us to achieve the desired results.

First you should consider the dataset you are going to analyze if you are working with sparse data then optimizers that do not Adapt their learning rate like SGD are going to have a poor performance. In those cases you should pick an Adaptive learning algorithm like Adagrad or ADAM, also you will gain the benefit to do not have to select the correct learning rate, which is a relief.

Now if your dataset is a large one, then there is not an immediately correct answer. However we have seen that adaptive algorithms tends to have a better convergence and produce better results even though not always are the fastest ones. Additionally if we want to do not worry about assigning a proper learn rate then an Adaptive algorithm is definitively the best solution. So in general Adaptive algorithms are optimizers that are versatile and usually outperform other existent solutions, yet a classic non adaptive algorithm can also be used if we know very well our dataset.

REFERENCES:

[1] Foreign currency Exchange rates prediction using CGP and Recurrent Neural networks

[2] Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling, 2014. Extracted from: https://arxiv.org/pdf/1412.3555.pdf

[3] Deep Learning for Time-Series Analysis John Gamboa , 2017.
Extracted from : https://arxiv.org/pdf/1701.01887.pdf

[4] Unsupervised Learning, 2009 Extracted from: http://www.gatsby.ucl.ac.uk/~dayan/papers/dun99b.pdf

[5] DeepFace: Closing the Gap to Human-Level Performance in Face Verification Extracted, 2014 from: https://www.cs.toronto.edu/~ranzato/publications/taigman_cvpr14.pdf

[6] Mastering the game of Go with deep neural networks and tree search, 2015
Extracted from: https://storage.googleapis.com/deepmind-media/alphago/AlphaGoNaturePaper.pdf

[7] Export forecasting, 2015
Extracted from: https://link.springer.com/chapter/10.1007/978-3-319-27284-9_16

[8] An overview of gradient descent algorithms, 2017
Extracted from: https://arxiv.org/pdf/1609.04747.pdf

[9] Types of Optimization Algorithms used in Neural Networks and Ways to Optimize Gradient Descent, 2017
Extracted from: https://towardsdatascience.com/types-of-optimization-algorithms-used-in-neural-networks-and-ways-to-optimize-gradient-95ae5d39529f

[10] An overview and comparative analysis of Recurrent Neural Networks for Short Term Load Forecasting, 2017. Extracted from: https://arxiv.org/pdf/1705.04378.pdf

[11] A Long-Short Term Memory Recurrent Neural Network Based Reinforcement Learning Controller for Office Heating Ventilation and Air Conditioning Systems, 2017 Extracted from: http://www.mdpi.com/2227-9717/5/3/46

[12] Long Short-Term Memory, 2001 Extracted from: http://web.eecs.utk.edu/~itamar/courses/ECE-692/Bobby_paper1.pdf

[13] Duchi, J., Hazan, E., & Singer, Y. (2011). Adaptive Subgradient Methods for Online Learning and Stochastic Optimization. Journal of Machine Learning Research, 12, 2121–

2159. Extracted from: http://jmlr.org/papers/v12/duchi11a.html

[14] ADADELTA: An Adaptive Learning Rate Method, 2012.

Extracted from: http://arxiv.org/abs/1212.5701

[15] Number of daily births in Quebec, Jan. 01 '77 - Dec. 31 '90 (Hipel & McLeod, 1994) extracted on January 24, 2018 from: https://datamarket.com/data/set/235j/number-of-daily-births-in-quebec-jan-01-1977-to-dec-31-1990#!ds=235j&display=line

[16] Table 080-0020 Retail trade, sales by the North American Industry Classification System (NAICS) extracted on November 18, 2017 from: https://www80.statcan.gc.ca/wes-esw3/page1-eng.htm