Total words count: 2614



Edward Hine (Student)

STUDENT NAME: EDWARD HINE

MSc DATA SCIENCE

**Emails:**ehine02@bbk.ac.uk

**Machine Learning- COIY065H7**

coursework report

April 27, 2021

Academic Declaration

I have read and understood the sections of plagiarism in the College Policy on assessment offences and confirm that the work is my own, with the work of others clearly acknowledged. I give my permission to submit my report to the plagiarism testing database that the College is using and test it using plagiarism detection software, search engines or meta-searching software.

Contents

[1 Introduction 2](#_Toc69982230)

[2 Methodology, Design, Technical Contribution 2](#_Toc69982231)

[2.1 Method & Design 2](#_Toc69982232)

[2.2 WAME Algorithm 3](#_Toc69982233)

[2.3 Data Preparation 4](#_Toc69982234)

[2.3.1 Columns & Features 4](#_Toc69982235)

[2.3.2 Missing Values 4](#_Toc69982236)

[2.3.3 Class Imbalance 4](#_Toc69982237)

[2.4 Parameters 5](#_Toc69982238)

[2.4.1 Network Architecture 5](#_Toc69982239)

[2.4.2 WAME Specific Parameters 6](#_Toc69982240)

[3 Experiments, Findings and Discussion 6](#_Toc69982241)

[4 Conclusions 7](#_Toc69982242)

[5 References 8](#_Toc69982243)

[6 Appendix 1- Additional Results 9](#_Toc69982244)

[7 Appendix 2- Code 10](#_Toc69982245)

# Introduction

Neural networks can be trained more efficiently through the application of optimisation techniques, specifically training algorithms. This report describes the evaluation of the WAME optimisation algorithm described in the paper ‘Training Convolutional Networks with Weight–wise Adaptive Learning Rates’[[1]](#footnote-2) by Mosca and Magoulas (M&M) applied to a given dataset against evaluations of other established optimisation techniques, namely Adam, RMSProp, Adagrad and Stochastic Gradient Descent.

The objective is to quantitatively assess the WAME optimisation performance through the technical implementation of the algorithm with a selection of parameters. The premise outlined in the referenced paper is that WAME results in improved average loss rates and is generally more accurate, albeit negligibly so, compared to results using the other optimisation algorithms mentioned above.

The dataset to be used for training the network is this: "Census Income" with the objective of predicting whether income exceeds $50K/yr based on this census data. The dataset comprises approximately 32,000 records of census data and 16,000 further records for evaluation of accuracy.

This report will:

* describe the optimisation methods evaluated using the dataset,
* detail the technical design and implementation of the WAME optimisation in Python,
* depict the graphical and tabulated output of the experiments conducted and
* draw conclusions based upon the same

# Methodology, Design, Technical Contribution

## Method & Design

The high-level design consists of three distinct but interfacing components:

* Dataset Parsing and Cleaning
* Novel WAME Implementation
* Testing Harness

I have implemented each component of the framework using Python built on top of several third-party libraries.

The dataset module is responsible for parsing the training and testing census dataset as described in the introduction. It makes use of Pandas to load the data from raw text input into a dataframe and perform transformations to appropriately format the data (see section 2.3 for details).

To compare the operation of various neural network training optimisation algorithms and configuration parameters, I have designed an automated testing framework with an implementation in Python. The framework accepts lists of activation functions, network widths (number of nodes in the hidden layer), and batch sizes as inputs. It then repeatedly trains multiple networks constructed with all the possible combinations of the supplied parameters, performs cross-validation and tests and writes each result to a file for analysis. As repeated runs have been performed the average performance may be calculated (see section 2.4). Additionally, the framework may also accept a list of optimisers which it can test using the identified optimal network parameters.

## WAME Algorithm

The WAME algorithm is itself implemented to be compatible with Tensorflow2.0[[2]](#footnote-3). This enables WAME to be compared with the other optimisers already implemented within the TF2 Keras[[3]](#footnote-4) framework. TF2 allows the programmer to construct a variety of network architectures into model objects. The network architecture model is flexible enough to enable different combinations of inputs, hidden layers, and activation functions to be configured and tested. Each network configuration is assembled separately and is subsequently compiled with different optimisers, loss functions and metrics so that the same network may be trained using a tailored training process. This separation means that many different combinations of architectures and training techniques can be compared.

The implementation of WAME is within a class (appropriately called WAME) that inherits from the TF2 OptimizerV2 base class. This is the parent class to the optimisers already implemented in the framework. During initialisation of the optimiser the several WAME hyperparameters are set to default values and the TF2 framework is set up to store and initialise the variables required during each iteration of the training process – namely the gradients (with initial tensor values=1), zetas (=1), betas (=0), and Z (=0). All of this initialisation is performed as described in the M&M’s WAME paper.

The TF2 framework calls the same method of the optimiser during each iteration of the training process; \_resource\_apply\_dense(). This method is where the core of the WAME algorithm is implemented. In M&M’s algorithm description, lines 4-12 correspond to the Python implementation as follows:

1. Multiply successive gradient values to determine sign
2. Update zeta according to
   1. Minimum of eta\_pos and eta\_max if positive sign
   2. Maximum of eta\_neg and eta\_min if negative sign
3. Update Z according to zeta and alpha
4. Update beta according to gradients and alpha
5. Update weights according to Z, beta and learning rate
6. Assign updated values to stored variables for next iteration

TF2 operates a graph-based execution model which offers improved performance and portability at the cost of increased complexity (especially when debugging). However, I believe the implementation provided is readable, understandable and corresponds well to the algorithm outlined in M&M’s paper.

## Data Preparation

### Columns & Features

The dataset consists of fifteen columns of census data including the target column salary. There are therefore fourteen potential features that can be used to train any model. By inspection it is apparent that some of these may be discarded:

* *education\_num* is an integer representing the ranking of *education\_lvl* column and is therefore duplicate
* *fnlwgt* is described as having a state specific interpretation in the data description, whereas the dataset is representative of all 51 states
* *capital\_loss* contains mostly zeros with what non-zero values there are evenly distributed between the two salary bands (i.e., it holds little predictive power)

Thus, eleven columns remain to be included as features. Three of these eleven (*age*, *hours\_pw* and *capital\_gain*) hold continuous numeric values and can be scaled in the range [-1,1] using a Scikit-Learn StandardScaler. The remaining eight are categorical and are each encoded as one-hot vectors[[4]](#footnote-5) in the input feature set using the Tensorflow2.0 API.

### Missing Values

Missing values are represented by “?” and are only present in columns *workclass*, *occupation* and *native-country*. The missing values are replaced by the modal categorical value in each case – “Private”, “Prof-specialty” and “United-States” respectively.

### Class Imbalance

There is a moderate imbalance between the prevalence of the two salary bands (class labels) within the training dataset[[5]](#footnote-6).



Imbalances like this can cause issues during training as mini batches will almost certainly contain fewer example records of the minority class, thus hindering the network’s ability to learn the classes’ characteristic features. There are several techniques available to mitigate against imbalances; in this instance I have chosen to implement an oversampling method which randomly samples (with replacement) and appends to the dataset duplicate records from the minority class until the classes are evenly distributed. This means that after oversampling the dataset grows to 49,440 records (2 x 24,720).

## Parameters

### Network Architecture

Taking advantage of the TF2 network model within the implemented test harness, it is possible to permute combinations of activation function, hidden layer width and batch size to perform repeated runs of training, cross-validation, and testing. By logging and analysing the corresponding results, the optimal configuration of the network model can be identified.

Figure 1 shows the results of an example test harness run, listing the average and maximum overall accuracy for different combinations of two activation functions (ReLu, Tanh), four hidden layer widths (7, 14, 28, 56 nodes), and four batch sizes (160, 320, 640, 1280). Each of the possible thirty-two combinations is run five times to calculate the average and maximum accuracy values. In this instance, the highlighted record (yellow) shows that by configuring a network with a single hidden layer of 56 nodes with ReLu activation functions and training batch size of 160 we could expect a respectable test accuracy of around 85% on average.



Figure 1 Network Architecture Testing Results

Beyond the above network parameters, my brief experimentation suggested that a single hidden layer was sufficient to achieve reasonable results – in fact, adding more layers tended to reduce testing accuracy, potentially overfitting the training data. Additionally, I found that applying a smaller learning rate (0.0001 with no decay) smoothed convergence during training, and that including a 30% dropout layer was appropriate to aid regularisation for testing.

### WAME Specific Parameters

I performed several ad-hoc experiments to compare different values of the eta (positive, negative, min and max clips) and alpha parameters of the WAME algorithm, but was unable to identify any adjustments that made significant improvements in efficiency or accuracy. Therefore, I have decided to fix the WAME specific parameters with the original values suggested in M&M’s paper.

# Experiments, Findings and Discussion

Previous experimentation using the test harness enabled me to determine a reasonable network configuration and training parameters to apply when testing the different optimisation algorithms. The following configuration was used:

* Input layer of eleven features
  + Three scaled numerical features
  + Eight categorical features encoded as one-hot vectors
* Hidden layer of 56 nodes with ReLU activation
* Training batch size of 160
* WAME specific parameters as described in M&M’s paper

With this consistent testing environment established, several instances of the network were trained using different optimisation algorithms to calculate comparative performance statistics. Five different optimisers were considered; WAME, RMSProp, Adam, Adagrad and Stochastic Gradient Descent, all using the implementations in the Tensorflow-Keras framework except for my novel implementation of WAME. The experiments were run repeatedly over a series of five runs. The average precision and recall results for networks trained with each optimiser are displayed in the following table.



Figure 2 Precision/Recall Results for networks trained with each optimiser

These results suggest WAME is very comparable to both Adam and RMSProp and that it outperforms both SGD and Adagrad. However, WAME is fundamentally designed to excel in the speed and efficiency with which it can converge to optimum parameters during training. To see whether this is the case, the training phase must be analysed.

By training networks with the five optimisers sequentially in the python test framework, several plots were generated for comparison of convergence rates in both training and cross validation. It can clearly be seen (Figure 3, below) that WAME outperforms the other methods, converging more quickly to a lower loss level and higher overall accuracy over the 20 training epochs in this example run (and consistently in my experiments). It is also clear that SGD and Adagrad may require much longer training epochs to approach the results of the momentum-based methods (if at all).

Graphical user interface, chart

Description automatically generated

Figure 3 Comparison of optimiser training and cross validation performance over 20 epochs

# Conclusions

This report has presented an analysis of the Census Income dataset, whereby neural networks have been trained as binary classifiers of salary bands based on the census records. Several different optimisers for training the network were compared to the implemented WAME method as described by M&M. I implemented a novel version of the algorithm described using the Tensorflow2/Keras framework. As the dataset was imbalanced, I performed oversampling of the minority class to normalise the data before training.

My experiments demonstrate that, on average, the WAME optimiser converges to a minimal loss more quickly than other comparable optimisers. It achieves this without any significant reduction in accuracy, and is, in fact, more accurate in some cases.

An extension to this experiment could be a review of the selection of the dataset features used as input to the models. However, my expectation would be that reduction of the feature set would not improve predictive ability given the intersectional nature of the data.

To improve the experimental design, I would consider trialling more combinations of network architectures with each optimiser. It is possible that certain optimisers perform better with architectures I did not analyse in these tests. It would also be interesting to apply similar tests to other datasets to investigate whether certain optimisers can become experts for specific datasets and/or problems - for example, multi-class classification, larger datasets and so on.

Additionally, given the time, I would be inclined to extend the test harness I implemented to make it more configurable and flexible, extending it into a general testing framework for neural network experiments of this kind.

# References

*Brownlee, J., 2019. A Gentle Introduction to Imbalanced Classification.*

*Available at* [*https://machinelearningmastery.com/what-is-imbalanced-classification/*](https://machinelearningmastery.com/what-is-imbalanced-classification/)

*Brownlee, J., 2015.Tactics to Combat Imbalanced Classes in your Machine Learning Dataset.*

[*https://machinelearningmastery.com/tactics-to-combat-imbalanced-classes-in-your-machine-learning-dataset/*](https://machinelearningmastery.com/tactics-to-combat-imbalanced-classes-in-your-machine-learning-dataset/)

*Chollet, F., 2018. Deep Learning with Python: Manning Publications*

*Geron, A., 2019. Hands-On Machine Learning with Scikit-Learn, Keras and TensorFlow: O’Reilly*

*Goodfellow, I., Bengio, Y., Courville, A., 2017. Deep Learning: The MIT Press*

*Mosca, A., Magoulas, G., 2017. Training Convolutional Networks with Weight-wise Adaptive Learning Rates. Available at* [*https://www.elen.ucl.ac.be/Proceedings/esann/esannpdf/es2017-50.pdf*](https://www.elen.ucl.ac.be/Proceedings/esann/esannpdf/es2017-50.pdf)

*Mosca, A., a.k.a. nitbix, 2017. Github Keras Fork. Available at*

[*https://github.com/nitbix/keras-oldfork/blob/master/keras/optimizers.py*](https://github.com/nitbix/keras-oldfork/blob/master/keras/optimizers.py)

*Tensorflow2.0 Tutorials, available online at* [*https://www.tensorflow.org/tutorials*](https://www.tensorflow.org/tutorials)

# Appendix 1- Additional Results

Below are test results from repeated trials after training the network using only the WAME optimiser to ascertain stability. I ran these train/test cycles in addition to the results presented in section 3 of the report by adapting the test\_harness() method.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| run | loss | accuracy | precision | recall | auc |
| 1 | 0.398 | 0.818 | 0.803 | 0.844 | 0.900 |
| 2 | 0.400 | 0.814 | 0.820 | 0.805 | 0.900 |
| 3 | 0.400 | 0.814 | 0.820 | 0.805 | 0.900 |
| 4 | 0.393 | 0.818 | 0.820 | 0.814 | 0.904 |
| 5 | 0.409 | 0.809 | 0.816 | 0.798 | 0.895 |
| 6 | 0.412 | 0.804 | 0.818 | 0.782 | 0.895 |
| 7 | 0.410 | 0.807 | 0.819 | 0.790 | 0.896 |
| 8 | 0.424 | 0.796 | 0.860 | 0.708 | 0.904 |
| 9 | 0.417 | 0.802 | 0.858 | 0.724 | 0.906 |
| 10 | 0.390 | 0.823 | 0.818 | 0.830 | 0.906 |
| Average | 0.405 | 0.811 | 0.825 | 0.790 | 0.901 |

Also appended are example plots of the metrics which were generated during each training run:

Shape

Description automatically generated

# Appendix 2- Code

The code is uploaded to Moodle alongside this report in three files

* census\_data.py
  + get\_clean\_data() performs general data cleaning
  + get\_feature\_columns() generates the TF2 feature representations
  + df\_to\_dataset() transforms the raw dataframe to a TF2 dataset
* wame\_impl.py
  + \_resource\_apply\_dense() implements the core WAME algorithm
* testing.py
  + optimiser\_comparison() generates the main experimental results in section 3
  + test\_harness() generates the network architecture combination tests in section 2.4 and the WAME stability tests in Appendix 1

The core functionality of the novel WAME implementation is appended here for ease of reference.

def \_resource\_apply\_dense(self, grad\_t, vars, apply\_state):  
 # get the stored hyper parameters  
 hyper = apply\_state.get((vars.device, vars.dtype.base\_dtype))  
  
 # extract the stored state from the previous iteration  
 grad = self.get\_slot(vars, 'grad')  
 beta = self.get\_slot(vars, 'beta')  
 zeta = self.get\_slot(vars, 'zeta')  
 zedd = self.get\_slot(vars, 'zedd')  
  
 # multiply current gradient by stored gradient to check the sign  
 gt\_g = grad\_t \* grad  
  
 # set zeta for this iteration according to the gradient sign and capped eta values  
 zeta\_t = tf.where(tf.greater(gt\_g, tf.zeros\_like(gt\_g)),  
 tf.minimum(zeta \* hyper['eta\_pos'], hyper['eta\_max']),  
 tf.maximum(zeta \* hyper['eta\_neg'], hyper['eta\_min']))  
  
 # set Z for current iteration  
 zedd\_t = hyper['alpha'] \* zedd + hyper['one\_minus\_alpha'] \* zeta\_t  
 # set beta for current iteration  
 beta\_t = hyper['alpha'] \* beta + hyper['one\_minus\_alpha'] \* grad\_t \*\* 2  
 # weights update for this iteration  
 vars\_t = vars - (hyper['lr\_t'] / zedd\_t) \* grad\_t / (math\_ops.sqrt(beta\_t) + hyper['epsilon'])  
 # store current state for next iteration  
 grad.assign(grad\_t, use\_locking=self.\_use\_locking)  
 beta.assign(beta\_t, use\_locking=self.\_use\_locking)  
 zeta.assign(zeta\_t, use\_locking=self.\_use\_locking)  
 zedd.assign(zedd\_t, use\_locking=self.\_use\_locking)  
  
 # store updated weights and return  
 return state\_ops.assign(vars, vars\_t, use\_locking=self.\_use\_locking).op

1. Training Convolutional Networks with Weight–wise Adaptive Learning Rates Alan Mosca and George D. Magoulas (M&M) [↑](#footnote-ref-2)
2. <https://www.tensorflow.org/api_docs/python/tf> [↑](#footnote-ref-3)
3. <https://www.tensorflow.org/api_docs/python/tf/keras> [↑](#footnote-ref-4)
4. <https://www.tensorflow.org/tutorials/structured_data/feature_columns> [↑](#footnote-ref-5)
5. <https://machinelearningmastery.com/what-is-imbalanced-classification> [↑](#footnote-ref-6)