MSc Data Science - Coursework Report

Machine Learning - COIY065H7

Dhinta Jesslyn Foster
Student number: 13156097

dfoste06@mail.bbk.ac.uk / dhintajesslyn@hotmail.com
Department of Computer and Information Systems

Birkbeck, University of London

April 2020

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

A mathematical process called optimization allows one to minimize or maximize an objective function. Deep Learning models, e.g. Convolutional Neural Networks (CNNs), have an objective to minimize the loss at each epoch of training, or in other terms, to maximize model performance in order to have an optimal output result. Optimizers are used to minimize such loss in the training process, thereby helping to play a vital role in updating model parameters (the weights and biases) in the direction of the optimal target output at each stage of the network (Walia, 2017)

1.1 Deep Learning Optimization - WAME

One of the latest methods proposed to train neural networks is a gradient-based update rule, named "*Weight-wise Adaptive learning rates with Moving average Estimator*", henceforth known as "WAME" (Mosca et al, 2017). This optimization algorithm has been shown to improve learning speed compared to other common optimizers (e.g. Stochastic Gradient Descent) without a performance generalisation trade-off (ibid, 2017).

Algorithm 1 WAME

1: pick $\alpha, \eta_+, \eta_- \zeta_{min}, \zeta_{max}$
2: $\theta_{ij}(0) = 0, Z_{ij}(0) = 0, \zeta_{ij} = 1 | \forall i, j$
3: **for all** $t \in [0..T]$ **do**
4: **if** $\frac{\partial E(t)}{\partial \omega_{ij}} \cdot \frac{\partial E(t-1)}{\partial \omega_{ij}} > 0$ **then**
5: $\zeta_{ij}(t) = \min\{\zeta_{ij}(t-1) * \eta_+, \zeta_{max}\}$
6: **else if** $\frac{\partial E(t)}{\partial \omega_{ij}} \cdot \frac{\partial E(t-1)}{\partial \omega_{ij}} < 0$ **then**
7: $\zeta_{ij}(t) = \max\{\zeta_{ij}(t-1) * \eta_-, \zeta_{min}\ \}$
8: **end if**
9: $Z_{ij}(t) = \alpha Z_{ij}(t-1) + (1-\alpha)\zeta_{ij}(t)$
10: $\theta_{ij}(t) = \alpha \theta_{ij}(t-1) + (1-\alpha)(\frac{\partial E(t)}{\partial \omega_{ij}}(t))^2$
11: $\Delta_{ij}(t) = -\frac{\lambda}{Z_{ij}} \cdot \frac{\partial E(t)}{\partial \omega_{ij}} \frac{1}{\sqrt{\theta_{ij}(t)} + \mathbf{u}}$
12: $\omega_{ij}(t+1) = \omega_{ij}(t) + \Delta\omega_{ij}(t)$
13: **end for**

The algorithm is outlined above, with one known error fixed[1] in line 11. I have considered two amendments to improve the algorithm[2]:

- Line 11: Dividing the gradient by $\sqrt{\theta_{ij}(t)}$ rather than just $\theta_{ij}(t)$ has been shown to be more stable and improve performance by reducing effects in changes in weights (Tieleman and Hinton, 2012).
- Line 11: I have added a "Unit roundoff" (or "Machine Epsilon")[3], $\mathbf{u}$, to the divisor in order to avoid relative errors due to rounding, or errors due to division by zero.

WAME utilises methods from other optimization algorithms, namely RProp and RMSProp, however this algorithm also allows a dynamic learning rate which adapts to each weight in the model's network.

[1] As per the algorithm description, the step-factor Δ_{ij} should be calculated by a divison of λ_{ij} by the EWMA, (Z_{ij}, not a multiplication

[2] These amendments have been discussed with the author, Professor George Magoulas on 30 March 2020

[3] https://en.wikipedia.org/wiki/Machine_epsilon

ζ_{ij} is a per-weight acceleration factor which sits between $\{\zeta_{min}, \zeta_{max}\}$ to "avoid runaway effects" (Mosca et al, 2017). α is a decay rate and $\lambda(t)$ is the learning rate as per RMSProp. WAME also has extra hyperparameters: η_-, η_+.

1.2 Dataset - F-MNIST

The dataset chosen for this report is the Fashion-MNIST (F-MNIST) dataset, which consists of a training set of 60,000 examples and a test set of 10,000 articles (Zalando, 2017). Each piece of data is a 28x28 grey-scale image (784 pixels in total) with a label from 10 classes. The datasets classes are: [0: T-shirt/top, 1:Trouser, 2:Pullover, 3:Dress, 4:Coat, 5:Sandal, 6:Shirt, 7:Sneaker, 8:Bag, 9:Ankle Boot] Zalando (2017) aims to replace the MNIST data-set as the benchmark dataset for computer vision tasks with F-MNIST, given its complexity compared to the original MNIST dataset and that simpler convolutional networks are gaining high accuracy rates with this dataset. This dataset contains various images of fashion items and is more challenging than MNIST, which may provide different evaluations to this training algorithm.

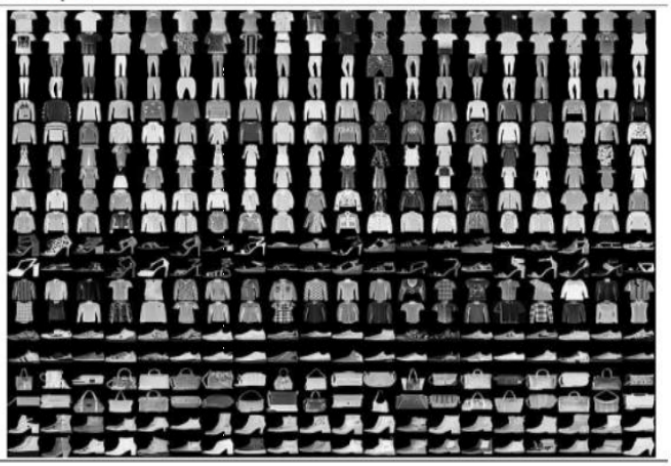
Label	Description	Examples
0	T-Shirt/Top	
1	Trouser	
2	Pullover	
3	Dress	
4	Coat	
5	Sandals	
6	Shirt	
7	Sneaker	
8	Bag	
9	Ankle boots	

Figure 1: Fashion-MNIST Example Images (Greeshma & Skreekumar 2019)

Previous Work

Since the publication of this dataset in 2017, Fashion-MNIST has been used in over 3,250 publications[4] utilising a number of different CNN architectures. Most recently, Kayad et al (2020) achieved a 98% accuracy using a LeNet-5 CNN architecture and provides a comprehensive review of previous works using this dataset.

1.3 Report Structure & Objectives

The aim of this assignment will implement the training algorithm described in section 1.1 using the dataset in 1.2. This report will firstly introduce the methodology, implementation and model design in Section 2, experimental results in Section 3 and concluding remarks and suggestions for future work in Section 4.

[4]Based on Google Scholar

2 Methodology and Design

2.1 Implementation & Software packages

The training algorithm will be implemented using Python 3 with suitable libraries imported for this assignment. I will mainly be using Keras as the deep learning framework as well as using "Google Colaboratory", which is a cloud-based Jupyter-style environment making available Google's GPU Instances without cost for faster model training.

In order to implement the custom optimizer, I utilise the Keras Backend API in order to call the "Optimizer" module and create a "WAME()" class. In Appendix II, I have detailed the code. In order to get the algorithm to work with the rest of the Keras framework, the class requires the following functions defined: __init__ , get_updates, get_config. Each function respectively, intialises the variables parsing through the optimization algorithm, gradient updates and configuration of variables at the next step of training. I follow similar implementations for other optimizers, such as RProp[5] and change the get_updates function to follow the steps of the algorithm outlined in Mosca et al (2017)[6].

2.2 Data Pre-processing

In order to load the dataset, I call the "load_data()" function from tensorflow.keras.datasets after importing the "fashion_mnist" dataset which allows automatic download from the cloud server. Two tuples, one for training and one for a test set are returned. The dataset already has a well-defined train and test set ready for use, however I first randomly sort both training and test datasets to remove any implicit sorting (Bhobé, 2019). The test dataset is then split into two: a test set and a cross-validation set (X_val/y_val), for model training and evaluation respectively. The evaluation of the final performance should be measured on a held-out data set that is never used in the training of the model. Some previous work (ibid, 2019) suggest an 80/20 split into testing and validation sets on the 10,000 testing images.

In order to prepare pixel data, scaling is required. Pixel values for each image in F-MNIST are unsigned integers in the range between black and white, or 0 and 255. Common practise suggests to scale each image between [0, 1] by converting each data point to a float, and dividing by 255 as CNN models tend to train faster on uniform and smaller scales.

2.3 Convolutional Neural Network (CNN) - Model Design

A Convolutional Neural Network is a common deep learning architecture utilising a multi-layer feed-forward artificial neural network, used predominantly in the field of image classification, first proposed by Yann LeCun in 1989. In the past 30 years, a number of different architectures and developments on the CNN model have been made. Similar to Mosca et al (2019), a simple two-convolutional layer architecture is initially implemented for feature extraction, as a simple design has been shown to be capable of achieving high levels of accuracy on the MNIST and CIFAR datasets.

[5] Available at the Toupee respository: https://github.com/nitbix/toupee

[6] Please refer to the appendix for the implemented code

Overview of the baseline model

For this assignment, I follow a simple CNN architecture with two hidden layers to find a baseline model. Firstly, for reproducibility of results, I set a seed. The first convolutional layer has 64 feature detectors of 5 x 5 to which we apply a 2 x 2 max pooling. The first max pool layer is the input to the second convolutional layer where we have another 64 filters, followed by a second 2x2 max pooling layer. Both use the relu activation. This baseline model then applies a 20% dropout then follows a dense layer of 128 units with a softmax activation function (given we want to see the probability of this model accurately classifying the correct image). This initial architecture design shows relatively good capability at extracting image features from the F-MNIST dataset with an initial test running through one single epoch reaching 80.2% accuracy. Similar to the MNIST dataset, there are ten classes but that being said, the F-MNIST dataset is slightly more complex given each image has more latent features. Therefore, a number of experiments must be run to evaluate how to achieve better accuracy using the WAME optimizer.

Baseline Architecture
64 Conv, 5 x 5 2x2 Max Pooling
64 Conv, 5x5 2x2 Max Pooling
Dense, 128 nodes 20% Dropout

3 Experiments, Findings and Discussion

3.1 Experiment design: Hyperparameter Tuning

In the previous section, some convergence was achieved using the WAME optimiser and baseline architcture. I run a number of experiments to improve the classification part of the model, searching through a number of hyperparameters to choose optimal values for the learning algorithm. There are number of approaches for hyper-parameter optimization - I choose to use a Random Search approach as this replaces searching through a manually specified subset of hyperparameters (such as a Grid Search) with randomly searching through combinations of proposed subset[7]. The scikit library offers a "RandomSearchCV" function which we utilise. In higher dimensions, it has been shown that Random Search outperfoms Grid search, especially when a small amount of hyperparameters affect the final performance of the model (Bergstra & Bengio, 2012). Furthermore, I choose Random over Grid search given that with higher dimension models, as the computational time taken is faster and is more efficient.

Within the classification part of the model, I look to vary the batch size in the dense layer as well as the dropout rate with the subsets outlined below. From the WAME optimizer itself, I choose to experiment varying the the learning rate, as Mosca et al (2017) have stated the values stated in the paper have been empirically established to be good.

To summarize, the following hyperparameter subsets are tuned over various training rounds:

- Batch size for the dense layer, [128, 256, 512, 1024]

[7] https://blog.usejournal.com/a-comparison-of-grid-search-and-randomized-search-using-scikit-learn-29823179bc85

- Dropout rate for the dense layer, [0.2, 0.4, 0.6, 0.8]
- Learning Rate [0.01, 0.001, 0.0001]

3.2 Results & Evaluation

Training Round 1-3

Each training round ran through 11 iterations of 5 epochs of random hyperparameter subsets of the batch size and dropout rate. Naturally, the higher the dropout rate, the better the generalisability of the model. To investigate the impact of the learning rate, I ran three different random search experiments - each time differing the learning rate.

Experiment	Learning Rate	Best Batch Size	Best Dropout Rate	Accuracy
#1	0.0001	512	20%	85.5%
#2	0.001	512	40%	88.8%
#3	0.01	256	40%	10.0%

The table above shows the results for the best hyperparameter in each experiment round, with a learning rate of 1e-3, batch size of 512 and a 40% dropout rate on the dense layer improving the accuracy. In each experiment, I have trained and tested the model using a validation approach and a simple 2-fold cross validation approach (due to time constraints in this assignment).

Training Round 4

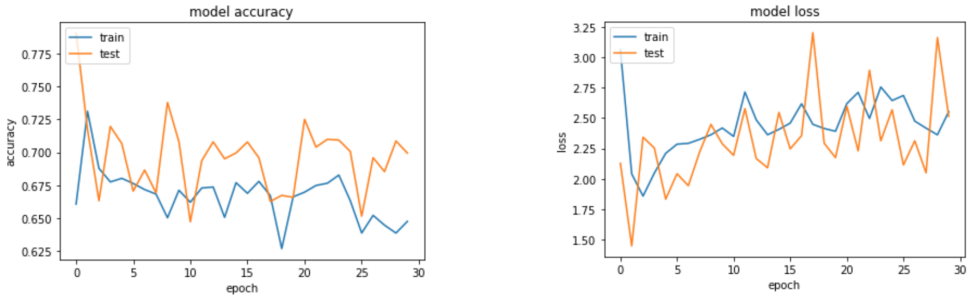
For the fourth training round I saved the best model produced in experiment 2 and run experiments on the test dataset over 30 epochs to gauge performance over increased training time. In order to evaluate this, I run a number of summary statistics on this model: F1, precision and recall.

Class	Precision	Recall	F1 Score
0	0.65	0.83	0.73
1	0.95	0.89	0.92
2	0.35	0.95	0.51
3	0.81	0.75	0.78
4	0.65	0.01	0.03
5	0.92	0.94	0.93
6	0.40	0.07	0.13
7	0.88	0.89	0.89
8	0.94	0.79	0.86
9	0.91	0.92	0.91

Precision is a measurement to determine of those predicted as a certain class, how many are actually relevant (i.e. true positives / actual results) and Recall refers to the proportion of relevant results classified by the model. The per-class recall shown in the above table shows quite a large variation per class with class 3 achieving the highest recall at 95%, yet class 6 achieved a disappointing 7% recall and class 4 missing pretty much altogether. That being said, overall the classifier performed relatively well over the other 8 classes. In order to rationalise the poor performance on class 4 and class 6, these depict greyscale images of coats and shirts respectively. As seen in the image in Section 1.2 and

generally speaking, there are wider variations in the types of these fashion items than might be expected with trousers or t-shirts with regards to the visible item shape within the image array.

The accuracy and loss results are presented in Figure 2. Theoretically, I expect accuracy to increase with number of epochs (i.e. less misclassification over time) and conversely loss to decrease. However, initially on the training dataset, accuracy increases for a small number of epochs and loss decreases, but then the curve continues in the opposite manner. Given the best epoch is suggested to be the 3rd out of only 30 epoch iterations, some overfitting could be indicated given the model was fitted with random hyperparameters.



(a) Accuracy (b) Loss

Figure 2: Experiment 4 - Accuracy vs Loss over 30 epochs

4 Conclusion

In this assignment, I implemented the WAME algorithm using Keras Backend, along with some adjustments to the proposed optimization algorithm in Mosca's paper to improve performance and any zero-division errors. Whilst using a simple random search methodology to tune hyperparameters, our simple model achieved a high of 88.8% classification accuracy.

Overall the given model was clearly not perfect, however there was some relatively good performance despite the simple initial CNN architecture design. Naturally, this accuracy rate does not compete with some of the more complex, deeper architectures that have achieved near-perfect performances. Automated hyperparameter optimization packages could be considered in the future to reduce any human bias in choosing the hyperparameter subsets (e.g. Keras Tuner) and to run further iterations when time permits. Given the behaviour of the accuracy and loss on our best given model was not as expected, perhaps tuning different hyperparameters at the smallest learning rate (1e-4) may have produced more accurate classifcation results.

The WAME algorithm has shown some ability to improve model accuracy rates in not only the F-MNIST dataset, but in others too (Mosca et al, 2017). That being said, fashion garments can be easily distorted and can be taken in different angles or settings (Kayad et al, 2020). In the future, I would experiment with deeper CNN architectures if higher levels of computing power and more time is available. For example comparing different deep architectures, such as ResNet, AlexNet, VGGNet and Inception Models using the WAME optimizer could show some interesting experimental results. Given that the spatial information of each instance within the dataset may vary quite drastically, this dataset could be applied to models that do not diminish such information within the MaxPooling layer and propose new optimisation techniques, entirely such as using Capsule Neural Networks (Hinton et al, 2017). In future, the WAME optimizer could be adapted to update each capsule node.

5 Bibliography

[1] Bergstra, J., Ca, J. and Ca, Y. (2012). Random Search for Hyper-Parameter Optimization Yoshua Bengio. Journal of Machine Learning Research, [online] 13, pp.281–305. Available at: http://jmlr.csail.mit.edu/papers/volume13/bergstra12a/bergstra12a.pdf.

[2] S. Bhatnagar, D. Ghosal and M. H. Kolekar, "Classification of fashion article images using convolutional neural networks," 2017 Fourth International Conference on Image Information Processing (ICIIP), Shimla, 2017, pp. 1-6, doi: 10.1109/ICIIP.2017.8313740.

[3] Bhobë, M. (2019). Classifying Fashion with a Keras CNN (achieving 94% accuracy) — Part 1. [online] Medium. Available at: https://medium.com/@mjbhobe/ classifying-fashion-with-a-keras-cnn-achieving-94-accuracy-part-1-1ffcb7e5f61a [Accessed 8 April 2020].

[4] Brownlee, J. (2016). Display Deep Learning Model Training History in Keras. [online] Machine Learning Mastery. Available at: https://machinelearningmastery.com/display-deep-learning-model-training-history-in-keras/.

[5] Brownlee, J. (2019). How to Calculate Precision, Recall, F1, and More for Deep Learning Models. [online] Machine Learning Mastery. Available at: https://machinelearningmastery.com/how-to-calculate-precision-recall-f1-and-more-for-deep-learning-models/.

[6] Brownlee, J. (2019). How to Develop a Deep CNN for Fashion-MNIST Clothing Classification. [online] Machine Learning Mastery. Available at: https://machinelearningmastery.com/how-to-develop-a-cnn-from-scratch-for-fashion-mnist-clothing-classification/.

[7] Brownlee, J. (2017). How to Use Metrics for Deep Learning with Keras in Python. [online] Machine Learning Mastery. Available at: https://machinelearningmastery.com/custom-metrics-deep-learning-keras-python/ [Accessed 10 May 2020].

[8] Caponetto, G. (2019). Random Search vs Grid Search for hyperparameter optimization. [online] Medium. Available at: https://towardsdatascience.com/random-search-vs-grid-search-for-hyperparameter-optimization-345e1422899d [Accessed 9 May 2020].

[9] Chollet, F. (2015) keras (2013) [online] Available at: https://github.com/charlespwd/project-title.

[10] Greeshma, K. Skreekumar, K. (2019) "Hyperparameter Optimization and Regularization on Fashion-MNIST Classification" International Journal of Recent Technology and Engineering. Blue Eyes Intelligence Engineering and Sciences Engineering and Sciences Publication - BEIESP, 8(2), pp. 3713–3719. doi: 10.35940/ijrte.b3092.078219.

[11] Hinton, G. (2012). Neural Networks for Machine Learning Lecture 6a Overview of mini–batch gradient descent. [online] Available at: https://www.cs.toronto.edu/ tijmen/csc321/slides/lecture$_s lides_l ec6.pdf$.

[12] Iazydjan, K. (2019). Hyper parameter tuning for Keras models with Scikit-Learn library. [online] Medium. Available at: https://medium.com/swlh/hyper-parameter-tuning-for-keras-models-with-scikit-learn-library-dba47cf41551 [Accessed 10 May 2020].

[13] M. Kayed, A. Anter and H. Mohamed, "Classification of Garments from Fashion MNIST Dataset Using CNN LeNet-5 Architecture," 2020 International Conference on Innovative Trends in Communication and Computer Engineering (ITCE), Aswan, Egypt, 2020, pp. 238-243, doi: 10.1109/ITCE48509.2020.9047776.

[14] MLGuy (2019). Adding Custom Loss and Optimizer in Keras. [online] Medium. Available at: https://medium.com/@mlguy/adding-custom-loss-and-optimizer-in-keras-e255764e1b7d [Accessed 1 April 2020].

[15] Mosca, A. and Magoulas, G. (2017). Training Convolutional Networks with Weight-wise Adaptive Learning Rates. [online] Available at: https://www.elen.ucl.ac.be/Proceedings/esann/esannpdf/es2017-50.pdf [Accessed 10 March 2020].

[16] Scikit-learn: Machine Learning in Python, Pedregosa et al., JMLR 12, pp. 2825-2830, 2011. from keras.datasets import fashion_mnist

[17] Anish Singh Walia (2017). Types of Optimization Algorithms used in Neural Networks and Ways to Optimize Gradient Descent. [online] Towards Data Science. Available at: https://towardsdatascience.com/types-of-optimization-algorithms-used-in-neural-networks-and-ways-to-optimize-gradient-95ae5d39529f.

[18] Xiao, H., Rasul, K. and Vollgraf, R. (2017). Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms. [online] Available at: https://arxiv.org/pdf/1708.07747.pdf [Accessed 10 March 2020].

6 Appendix

6.1 Appendix I - Raw Data & Results

Instructions for accessing the datasets

As discussed in Section 1.2, the Fashion MNIST dataset was initially published via Kaggle, a data science online community, which is available here: https://www.kaggle.com/zalando-research/fashionmnist

The original dataset was produced by Zalando Research (2017) and is available here: https://github.com/zalandoresearch/fashion-mnist

For reproducibility and purposes of this assignment, the Fashion MNIST dataset is readily available via Keras Datasets and can be imported with the following import:

```
from keras.datasets import fashion_mnist
```

Please consult the "results_df13156097.docx" uploaded file for the full results from the experiments.

6.2 Appendix II - Code

As per the assignment brief, the code for this report has been uploaded to Moodle named "finalcode_df13156097.py". I have attached a copy of the Python code output below for completeness, however for best results, I recommend to follow the python files as uploaded on an interactive web-based jupyter environment or via Google Colaboratory.

Machine Learning Coursework - Dhinta Foster (Student Number: 13156097)

Please note the results to this code are uploaded in a separate .docx document on Moodle

In [1]:
```
# Import relevant libraries

import numpy as np

from keras.datasets import fashion_mnist
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Dropout
from keras.layers import Flatten
from keras.layers.convolutional import Conv2D
from keras.layers.convolutional import MaxPooling2D
from keras.utils import np_utils
from keras import backend as K
```

```
/Users/dhinta/anaconda3/lib/python3.6/site-packages/h5py/__init__.py:36: FutureWarning: Conversion of the second argument of
  from ._conv import register_converters as _register_converters
Using TensorFlow backend.
```

In [ ]:
```
# F-MNIST Datasets - split into training and test

# fix random seed for reproducibility
np.random.seed(404)

# load data
(X_train, y_train), (X_test, y_test) = fashion_mnist.load_data()
```

```
X_train = X_train.reshape(60000, 28, 28, 1)
X_test = X_test.reshape(10000, 28, 28, 1)

# randomly sort X_test/y_test
indexes = np.arange(X_test.shape[0])
for _ in range(5): indexes = np.random.permutation(indexes)  # shuffle 5 times!
X_test = X_test[indexes]
y_test = y_test[indexes]
```

In [37]:
```
# Implement WAME optimizer using Keras backend.
# It has the same structure as the optimizers defined in Keras documentation,
# utilizing init, get_updates and get_config functions

from keras.optimizers import Optimizer
from keras import backend as K

if K.backend() == 'tensorflow':
    import tensorflow as tf

class WAME(Optimizer):
    """Weight{wise Adaptive learning rates with Moving average Estimator Optimizer. Mosca et al (2017).
       Available from:
       https://www.elen.ucl.ac.be/Proceedings/esann/esannpdf/es2017-50.pdf
    """
    def __init__(self, learning_rate = 0.0001, alpha=0.9, eta_plus = 1.2, eta_minus = 0.1,
                 zeta_min = 0.01, zeta_max = 100, epsilon = 1e-12, **kwargs):
        """Initialise WAME Optimizer with variable values as suggested from the Mosca et al (2017) paper"""
        super(WAME, self).__init__(**kwargs)
        self.learning_rate = K.variable(learning_rate)
        self.alpha = alpha
        self.eta_plus = eta_plus
        self.eta_minus = eta_minus
        self.zeta_min = zeta_min
        self.zeta_max = zeta_max
        self.epsilon = epsilon
        self.initial_decay = kwargs.pop('decay', 0.0)
        with K.name_scope(self.__class__.__name__):
            self.iterations = K.variable(0, dtype='int64', name='iterations')
            self.decay = K.variable(self.initial_decay, name='decay')

    @K.symbolic
    def get_updates(self, params, loss, contraints=None):
        self.updates = [K.update_add(self.iterations, 1)]
        grads        = self.get_gradients(loss, params)
        shapes       = [K.int_shape(p) for p in params]
        old_grads    = [K.zeros(shape) for shape in shapes]
        weights      = [K.zeros(shape) for shape in shapes]

        # Learning Rate
        learning_rate = self.learning_rate
        if self.initial_decay > 0:
            learning_rate *= (1. / (1. + self.decay * self.iterations))

        t = self.iterations + 1

        # Line 2 - initialise current weights

        zeta      = [K.ones(shape) for shape in shapes]
        Z         = [K.zeros(shape) for shape in shapes]
        theta     = [K.zeros(shape) for shape in shapes]

        for p, g, w, expMA, prevZ, prevTheta, old_g in zip(params, grads, weights, zeta, Z, theta, old_grads):
            change      = g * old_g
```

```
                pos_change  = K.greater(change,0.)
                neg_change  = K.less(change,0.)

                # Line 3-8: For all t in [1..t] do the following

                zeta_t      = K.switch(pos_change,
                                       K.minimum(expMA * self.eta_plus, self.zeta_max),
                                       K.switch(neg_change, K.maximum(expmA * self.eta_minus, self.zeta_min), expMA))
                zeta_clip   = K.clip(zeta_t, self.zeta_min, self.zeta_max)

                # Lines 9-12: Update weights for t with amendments as proposed for line 11

                Z_t         = (self.alpha * prevZ) + ((1 - self.alpha) * zeta_t)
                theta_t     = (self.alpha * prevTheta) + ((1 - self.alpha) * K.square(g))
                wChange     = - (learning_rate * (zeta_clip /zeta_t) * g) / K.sqrt(theta_t + self.epsilon)
                new_weight = w + wChange
                p_update    = p - w + new_weight

                self.updates.append(K.update(p,p_update))
                self.updates.append(K.update(w,new_weight))
                self.updates.append(K.update(expMA,zeta_t))
                self.updates.append(K.update(prevZ,Z_t))
                self.updates.append(K.update(prevTheta,theta_t))
            return self.updates

        def get_config(self):
            config = {'alpha': float(K.get_value(self.alpha)),
                      'learning_rate' : float(K.get_value(self.learning_rate)),
                      'eta_plus' : float(K.get_value(self.eta_plus)),
                      'eta_minus' : float(K.get_value(self.eta_minus)),
                      'zeta_min' : float(K.get_value(self.zeta_min)),
                      'zeta_max' : float(K.get_value(self.zeta_max))}
            base_config = super(WAME, self).get_config()
            return dict(list(base_config.items()) + list(config.items()))
```

Baseline Model

```
In [38]: # Implement baseline model with Keras Sequential

         def deep_cnn_model():

             # Model Layers
             model = Sequential()
             model.add(Conv2D(64, (5, 5),
                              input_shape=(28, 28, 1),
                              activation='relu'))
             model.add(MaxPooling2D(pool_size=(2, 2)))
             model.add(Conv2D(64, (5, 5), activation='relu'))
             model.add(MaxPooling2D(pool_size=(2, 2)))
             model.add(Dropout(0.2))
             model.add(Flatten())
             model.add(Dense(128, activation='softmax'))

             # Compile Model
             model.compile(loss='sparse_categorical_crossentropy',
                           optimizer= WAME(),
                           metrics=['accuracy'])

             return model

In [39]: # Build the model
         model = deep_cnn_model()
```

```
In [40]: # Fit the model - test on one epoch
         model.fit(X_train,
                   y_train,
                   validation_data=(X_test, y_test),
                   epochs=1,
                   batch_size=200)
```

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/1
60000/60000 [==============================] - 93s 2ms/step - loss: 2.0903 - accuracy: 0.6717 - val_loss: 0.5976 - val_accur
```

```
Out[40]: <keras.callbacks.callbacks.History at 0xb2d6e6cf8>
```

Experiment 1

```
In [ ]: # In order to do hyperparameter tuning, I will utilise more libraries

        import sklearn
        from sklearn.metrics import accuracy_score
        from sklearn.model_selection import RandomizedSearchCV
        from tensorflow.keras.wrappers.scikit_learn import KerasClassifier
        import matplotlib.pyplot as plt
```

```
In [ ]: # Create Keras models such that hyperparameters can be iterable

        def build_model(batch_size = 128, rate = 0.2):
            model = Sequential()
            model.add(Conv2D(64, (5, 5),
                             input_shape=(28, 28, 1),
                             activation='relu'))
            model.add(MaxPooling2D(pool_size=(2, 2)))
            model.add(Conv2D(64, (5, 5), activation='relu'))
            model.add(MaxPooling2D(pool_size=(2, 2)))
            model.add(Dropout(rate))
            model.add(Flatten())
            model.add(Dense(batch_size, activation='softmax'))
            model.compile(loss="sparse_categorical_crossentropy",
                          optimizer=WAME(),
                          metrics=["acc"])
            return model
```

```
In [ ]: # Returns summary of code

        model_default = build_model()
        model_default.summary()
```

```
In [ ]: # Hyperparameter subsets to experiment

        _batch_size=[128, 256, 512, 1024]
        _rate=[0.2, 0.4, 0.6, 0.8]


        params=dict(batch_size=_batch_size,
                    rate=_rate)

        print(params)
```

```
In [ ]: model = KerasClassifier(build_fn=build_model,epochs=10)

        np.random.seed(404) # set seed for reproducibility
```

```
        # Run random search model. Iterations over 5 epochs.
        rscv = RandomizedSearchCV(model, param_distributions=params, cv=2,     n_iter=5)

        rscv_results = rscv.fit(X_train,y_train)

        # print results

        print('Best score is: {} using {}'.format(rscv_results.best_score_,
        rscv_results.best_params_))
```

Experiment 2

```
In [ ]: # Implement WAME with different learning rate

        class WAME(Optimizer):
            """Weight{wise Adaptive learning rates with Moving average Estimator Optimizer. Mosca et al (2017).
                Available from:
               https://www.elen.ucl.ac.be/Proceedings/esann/esannpdf/es2017-50.pdf
            """
            def __init__(self, learning_rate = 0.001, alpha=0.9, eta_plus = 1.2, eta_minus = 0.1,
                         zeta_min = 0.01, zeta_max = 100, epsilon = 1e-12, **kwargs):
                """Initialise WAME Optimizer with variable values as suggested from the Mosca et al (2017) paper"""
                super(WAME, self).__init__(**kwargs)
                self.learning_rate = K.variable(learning_rate)
                self.alpha = alpha
                self.eta_plus = eta_plus
                self.eta_minus = eta_minus
                self.zeta_min = zeta_min
                self.zeta_max = zeta_max
                self.epsilon = epsilon
                self.initial_decay = kwargs.pop('decay', 0.0)
                with K.name_scope(self.__class__.__name__):
                    self.iterations = K.variable(0, dtype='int64', name='iterations')
                    self.decay = K.variable(self.initial_decay, name='decay')

            @K.symbolic
            def get_updates(self, params, loss, contraints=None):
                self.updates = [K.update_add(self.iterations, 1)]
                grads        = self.get_gradients(loss, params)
                shapes       = [K.int_shape(p) for p in params]
                old_grads    = [K.zeros(shape) for shape in shapes]
                weights      = [K.zeros(shape) for shape in shapes]

                # Learning Rate
                learning_rate = self.learning_rate
                if self.initial_decay > 0:
                    learning_rate *= (1. / (1. + self.decay * self.iterations))

                t = self.iterations + 1

                # Line 2 - initialise current weights

                zeta      = [K.ones(shape) for shape in shapes]
                Z         = [K.zeros(shape) for shape in shapes]
                theta     = [K.zeros(shape) for shape in shapes]

                for p, g, w, expMA, prevZ, prevTheta, old_g in zip(params, grads, weights, zeta, Z, theta, old_grads):
                    change      = g * old_g
                    pos_change  = K.greater(change,0.)
                    neg_change  = K.less(change,0.)
```

```
                # Line 3-8: For all t in [1..t] do the following

                zeta_t      = K.switch(pos_change,
                                       K.minimum(expMA * self.eta_plus, self.zeta_max),
                                       K.switch(neg_change, K.maximum(expmA * self.eta_minus, self.zeta_min), expMA))
                zeta_clip   = K.clip(zeta_t, self.zeta_min, self.zeta_max)

                # Lines 9-12: Update weights for t with amendments as proposed for line 11

                Z_t         = (self.alpha * prevZ) + ((1 - self.alpha) * zeta_t)
                theta_t     = (self.alpha * prevTheta) + ((1 - self.alpha) * K.square(g))
                wChange     = - (learning_rate * (zeta_clip /zeta_t) * g) / K.sqrt(theta_t + self.epsilon)
                new_weight = w + wChange
                p_update    = p - w + new_weight

                self.updates.append(K.update(p,p_update))
                self.updates.append(K.update(w,new_weight))
                self.updates.append(K.update(expMA,zeta_t))
                self.updates.append(K.update(prevZ,Z_t))
                self.updates.append(K.update(prevTheta,theta_t))
            return self.updates

        def get_config(self):
            config = {'alpha': float(K.get_value(self.alpha)),
                      'learning_rate' : float(K.get_value(self.learning_rate)),
                      'eta_plus' : float(K.get_value(self.eta_plus)),
                      'eta_minus' : float(K.get_value(self.eta_minus)),
                      'zeta_min' : float(K.get_value(self.zeta_min)),
                      'zeta_max' : float(K.get_value(self.zeta_max))}
            base_config = super(WAME, self).get_config()
            return dict(list(base_config.items()) + list(config.items()))
```

In [ ]:
```
# Build iterable model

def build_model(batch_size = 128, rate = 0.2):
    model = Sequential()
    model.add(Conv2D(64, (5, 5),
                     input_shape=(28, 28, 1),
                     activation='relu'))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Conv2D(64, (5, 5), activation='relu'))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Dropout(rate))
    model.add(Flatten())
    model.add(Dense(batch_size, activation='softmax'))
    model.compile(loss="sparse_categorical_crossentropy",
                  optimizer=WAME(),
                  metrics=["acc"])
    return model

# Hyperparameters to test
_batch_size=[128, 256, 512, 1024]
_rate=[0.2, 0.4, 0.6, 0.8]


params=dict(batch_size=_batch_size,
            rate=_rate)

print(params)

# Create model & set seed

model = KerasClassifier(build_fn=build_model,epochs=10)
```

```
np.random.seed(404)

# Run randomised search

rscv = RandomizedSearchCV(model, param_distributions=params, cv=2,     n_iter=5)

rscv_results = rscv.fit(X_train,y_train)

# Print results

print('Best score is: {} using {}'.format(rscv_results.best_score_,
rscv_results.best_params_))
```

Experiment 3

```
In [ ]: # Implement WAME with different learning rate

class WAME(Optimizer):
    """Weight{wise Adaptive learning rates with Moving average Estimator Optimizer. Mosca et al (2017).
       Available from:
       https://www.elen.ucl.ac.be/Proceedings/esann/esannpdf/es2017-50.pdf
    """
    def __init__(self, learning_rate = 0.01, alpha=0.9, eta_plus = 1.2, eta_minus = 0.1,
                 zeta_min = 0.01, zeta_max = 100, epsilon = 1e-12, **kwargs):
        """Initialise WAME Optimizer with variable values as suggested from the Mosca et al (2017) paper"""
        super(WAME, self).__init__(**kwargs)
        self.learning_rate = K.variable(learning_rate)
        self.alpha = alpha
        self.eta_plus = eta_plus
        self.eta_minus = eta_minus
        self.zeta_min = zeta_min
        self.zeta_max = zeta_max
        self.epsilon = epsilon
        self.initial_decay = kwargs.pop('decay', 0.0)
        with K.name_scope(self.__class__.__name__):
            self.iterations = K.variable(0, dtype='int64', name='iterations')
            self.decay = K.variable(self.initial_decay, name='decay')

    @K.symbolic
    def get_updates(self, params, loss, contraints=None):
        self.updates = [K.update_add(self.iterations, 1)]
        grads        = self.get_gradients(loss, params)
        shapes       = [K.int_shape(p) for p in params]
        old_grads    = [K.zeros(shape) for shape in shapes]
        weights      = [K.zeros(shape) for shape in shapes]

        # Learning Rate
        learning_rate = self.learning_rate
        if self.initial_decay > 0:
            learning_rate *= (1. / (1. + self.decay * self.iterations))

        t = self.iterations + 1

        # Line 2 - initialise current weights

        zeta      = [K.ones(shape) for shape in shapes]
        Z         = [K.zeros(shape) for shape in shapes]
        theta     = [K.zeros(shape) for shape in shapes]

        for p, g, w, expMA, prevZ, prevTheta, old_g in zip(params, grads, weights, zeta, Z, theta, old_grads):
            change      = g * old_g
            pos_change  = K.greater(change,0.)
```

```
                neg_change  = K.less(change,0.)

                # Line 3-8: For all t in [1..t] do the following

                zeta_t      = K.switch(pos_change,
                                       K.minimum(expMA * self.eta_plus, self.zeta_max),
                                       K.switch(neg_change, K.maximum(expmA * self.eta_minus, self.zeta_min), expMA))
                zeta_clip   = K.clip(zeta_t, self.zeta_min, self.zeta_max)

                # Lines 9-12: Update weights for t with amendments as proposed for line 11

                Z_t         = (self.alpha * prevZ) + ((1 - self.alpha) * zeta_t)
                theta_t     = (self.alpha * prevTheta) + ((1 - self.alpha) * K.square(g))
                wChange     = - (learning_rate * (zeta_clip /zeta_t) * g) / K.sqrt(theta_t + self.epsilon)
                new_weight = w + wChange
                p_update    = p - w + new_weight

                self.updates.append(K.update(p,p_update))
                self.updates.append(K.update(w,new_weight))
                self.updates.append(K.update(expMA,zeta_t))
                self.updates.append(K.update(prevZ,Z_t))
                self.updates.append(K.update(prevTheta,theta_t))
            return self.updates

        def get_config(self):
            config = {'alpha': float(K.get_value(self.alpha)),
                      'learning_rate' : float(K.get_value(self.learning_rate)),
                      'eta_plus' : float(K.get_value(self.eta_plus)),
                      'eta_minus' : float(K.get_value(self.eta_minus)),
                      'zeta_min' : float(K.get_value(self.zeta_min)),
                      'zeta_max' : float(K.get_value(self.zeta_max))}
            base_config = super(WAME, self).get_config()
            return dict(list(base_config.items()) + list(config.items()))
```

In [ ]:
```
# Build iterable model

def build_model(batch_size = 128, rate = 0.2):
    model = Sequential()
    model.add(Conv2D(64, (5, 5),
                     input_shape=(28, 28, 1),
                     activation='relu'))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Conv2D(64, (5, 5), activation='relu'))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Dropout(rate))
    model.add(Flatten())
    model.add(Dense(batch_size, activation='softmax'))
    model.compile(loss="sparse_categorical_crossentropy",
                  optimizer=WAME(),
                  metrics=["acc"])
    return model

# Hyperparameters to test
_batch_size=[128, 256, 512, 1024]
_rate=[0.2, 0.4, 0.6, 0.8]


params=dict(batch_size=_batch_size,
            rate=_rate)

print(params)

# Create model & set seed
```

```
model = KerasClassifier(build_fn=build_model,epochs=10)

np.random.seed(404)

# Run randomised search

rscv = RandomizedSearchCV(model, param_distributions=params, cv=2,     n_iter=5)

rscv_results = rscv.fit(X_train,y_train)

# Print results

print('Best score is: {} using {}'.format(rscv_results.best_score_,
rscv_results.best_params_))
```

Experiment 4 - Best Model

```
In [ ]: # Define model layers with the best hyperparameters from experiments 1-3 and split into validation sets
        # Add additional metrics to calcuate to be able to create graphs later
        # Use WAME implementation with lr at 1e-3

        np.random.seed(404)

        from sklearn.model_selection import train_test_split
        X_train, X_val, y_train, y_val = train_test_split(X_train, y_train, test_size=0.2, random_state=1)

        def best_model():
            model_best = Sequential()
            model_best.add(Conv2D(64, (5, 5),
                                  input_shape=(28, 28, 1),
                                  activation='relu'))
            model_best.add(MaxPooling2D(pool_size=(2, 2)))
            model_best.add(Conv2D(64, (5, 5), activation='relu'))
            model_best.add(MaxPooling2D(pool_size=(2, 2)))
            model_best.add(Dropout(0.4))
            model_best.add(Flatten())
            model_best.add(Dense(512, activation='softmax'))
            model_best.compile(loss="sparse_categorical_crossentropy",
                               optimizer=WAME(),
                               metrics=['mse', 'mae', 'mape', 'cosine', 'acc'])
            return model_best

In [ ]: # Confirm summary of created model

        model_2 = best_model()
        model_2.summary()

In [ ]: # Re-run model over 30 epochs

        history = model_2.fit(X_train, y_train, epochs=30, batch_size=1, validation_data=(X_val, y_val))

In [ ]: from numpy import array
        from matplotlib import pyplot
        from sklearn.metrics import classification_report

        # summarize history for accuracy
        plt.plot(history.history['acc'])
        plt.plot(history.history['val_acc'])
        plt.title('model accuracy')
        plt.ylabel('accuracy')
        plt.xlabel('epoch')
```

```
plt.legend(['train', 'test'], loc='upper left')
plt.show()
# summarize history for loss
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()

# Print classification report

y_pred = model_2.predict(X_test, batch_size=10, verbose=1)
y_pred_bool = np.argmax(y_pred, axis=1)

print(classification_report(y_test, y_pred_bool))
```