Table of Contents

Part A

Part I

This part of the assignment requires a SoftMax classifier implementation using only low level TensorFlow methods. 3 main functions were implemented for the classifier, which is the forward pass function containing the SoftMax activation, cross entropy function which calculates the SoftMax loss for each iteration, and the calculate accuracy function which calculates the accuracy of the SoftMax classifier's predictions. The code and explanation for the functions are shown below:

Forward Pass

```
def forward_pass(x, weights, bias):
    A = tf.matmul(x, weights) + bias
    t = tf.math.exp(A)
    sumOfT = tf.reduce_sum(t, axis=1)
    sumOfT = tf.reshape(sumOfT, (x.shape[0], 1))
    H = t / sumOfT
    return H
```

In the forward pass function, the "A" variable represents the logits for the SoftMax activation formula. The logits are calculated by multiplying the training features matrix and the weights matrix, with the resulting matrix output be followed by the addition of the bias. The logits are then subjected to an exponential calculation, denoted by "t" (this would be the numerator for the SoftMax activation formula). Next, a sum of all values within "t" in each row will be calculated (denoted by "sumOfT"), this would then be the denominator of the SoftMax activation formula. With all the values in place, the function returns the result of the SoftMax activation value, which is "t" divided by the "sumOfT".

Cross Entropy

```
def cross_entropy(y, pred):
    entropy = -tf.reduce_sum(y * tf.math.log(pred + 1e-10), axis=1)
    loss = tf.reduce_mean(entropy)
    return loss
```

The SoftMax loss is calculated by multiplying the label values (which is either 0 or 1) with the natural log of the predicted values generated by the forward pass function element wise. In the implementation above, an extremely small float value was added to the predicted values to prevent the algorithm from performing a log(0), which yields an infinity value. After the multiplication operation, the result is the subjected to a summation reduction on a row-by-row basis. The summation results are then inverted negatively as most of the resulting values are negative, denoted

by “entropy” in the code snippet. The “entropy” was then passed into a mean reduction function to calculate the average loss of the prediction.

Calculating Accuracy

```
def calculate_accuracy(x, y, weights, bias):
    pred = forward_pass(x, weights, bias)
    predictions_correct = tf.cast(tf.equal(tf.math.argmax(pred, axis=1), tf.math.argmax(y, axis=1)), tf.float32)
    accuracy = tf.reduce_mean(predictions_correct)
    return accuracy
```

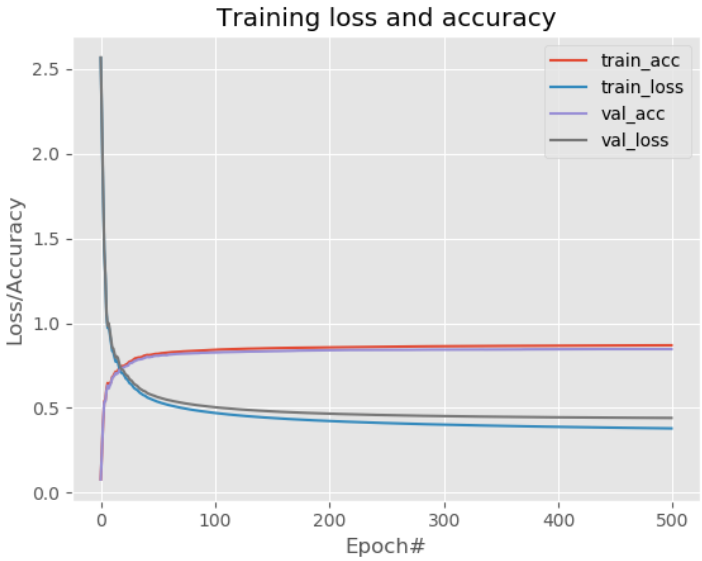
As the resulting prediction matrix from forward pass and the label matrix both have 10 columns (representing number of classes) and *m* number of instances, calculating the accuracy is simply obtaining the index of the maximum value of each row in both matrices and comparing said indexes. In the 2nd line in the function block above, the argmax of both prediction matrix and label matrix were compared and stored in the “predictions_correct” variable, leaving with a matrix with values of either 0 or 1 (indicating true or false) to denote which prediction instance is correct. The accuracy can then be calculated by a mean reduction on the “predictions_correct” variable.

Results

The training operation on the implemented SoftMax classifier model was done with the following hyper-parameters:

learning rate = 0.01, iterations=500

The test dataset was treated as a validation data for each iteration and the accuracy/loss graph for the training is illustrated as below:



From the graph shown, it can be seen that after 500 iterations, the loss values for both training instances and validation instances converged nicely while showing little divergence from each other by the end. This means that minimal overfitting was observed in this model, which might be due to the simplicity of the model itself, containing only one SoftMax output layer. The accuracy for both training and validation are decent as well, albeit both hitting plateaus after 150 iterations. The final accuracy scores on the test dataset after 500 iterations yielded an accuracy rate of 84.68%.

Part II

In this section, two network architectures (network architecture A and B) were proposed to be built upon the code implemented in Part I. Network architecture A consists of 2 fully connected layers, a 300 neurons ReLU layer and a SoftMax output layer. Network architecture B would consist of 3 fully connected layers, a 300 neurons ReLU layer, a 100 neurons ReLU layer and a SoftMax output layer. The forward pass functions for each of the network architecture above will be shown below.

Forward Pass (Network Architecture A)

```
def forward_pass(x, weights1, bias1, weights2, bias2):
    # first layer (ReLu)
    A1 = tf.matmul(x, weights1) + bias1
    H1 = tf.maximum(A1, 0)

    # second layer (softmax)
    A2 = tf.matmul(H1, weights2) + bias2
    t = tf.math.exp(A2)
    sumOfT = tf.reduce_sum(t, axis=1)
    sumOfT = tf.reshape(sumOfT, (H1.shape[0], 1))
    H2 = t / sumOfT
    return H2
```

For the first ReLU layer, the logits were calculated exactly the same as before in the previous model. With the resulting logits, it is then subjected to a maximum function to suppress all negative values within the logits to 0, as defined in the ReLU activation formula. The results from the ReLU activation formula will subsequently be used for the logits calculation in the next layer (SoftMax), along with the second weights and bias matrices. From there, the implementation is identical to the SoftMax activation code in the previous model, and the resulting SoftMax activation matrix is returned for the use of cross entropy calculation. To summarize, the forward pass function now accepts 2 pairs of weights bias matrices, corresponding to each layer present in the network (2 layers), to be used for ReLU activation and SoftMax activation.

Forward Pass (Network Architecture B)

```
def forward_pass(x, weights1, bias1, weights2, bias2, weights3, bias3):
    # first layer (ReLu) with 300 neurons
    A1 = tf.matmul(x, weights1) + bias1
    H1 = tf.maximum(A1, 0)

    # second layer (ReLu) with 100 neurons
    A2 = tf.matmul(H1, weights2) + bias2
    H2 = tf.maximum(A2, 0)

    # second layer (softmax)
    A3 = tf.matmul(H2, weights3) + bias3
    t = tf.math.exp(A3)
    sumOfT = tf.reduce_sum(t, axis=1)
    sumOfT = tf.reshape(sumOfT, (H2.shape[0], 1))
    H3 = t / sumOfT
    return H3
```

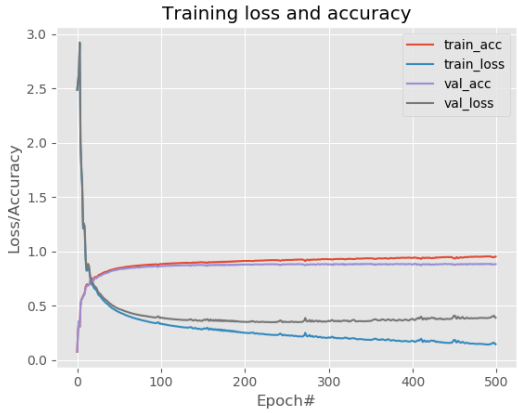
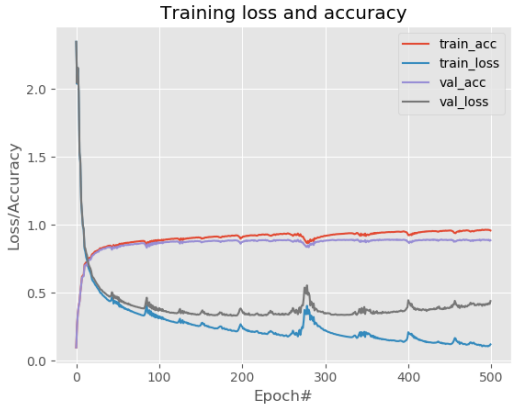
Even with multiple layers, the implementation remains largely similar, with the only difference in the forward pass between network architecture A and B being that the forward pass in network architecture B accepts an additional weights and bias matrix pair that corresponds to the addition of the third layer in the network.

Results

The training operation on the implemented network architecture A and B models were done with the following hyper-parameters:

learning rate = 0.01, iterations=500

The test dataset was treated as validation data for each iteration and the accuracy/loss graph for the training is illustrated as below:

	
Network Architecure A	Network Architecture B

Loss/Accuracy graphs for Network Architecture A and B

Models	**Accuracy (Test Set)**
Network Architecture A	88.17%
Network Architecture B	87.94%

Accuracy Scores on Test Datasets

It can be observed from the accuracy/loss graphs above that both architecture models exhibited some degree of overfitting during training based on the diverging loss patterns between the training loss and the validation loss. This might be due to the increased complexity of the model, now having more than 1 layer, thus increasing the chances of overfitting. There appears to be a more divergent accuracy pattern between training accuracy and validation accuracy for both models as well, when compared with the model from part I.

Besides, there appears to be increasingly volatile accuracy and loss patterns as the number of layers increase in a network. This can be observed as the loss patterns in network architecture B showed more obvious fluctuations as compared to network architecture A, which has lesser layers in the network. However, the resulting loss and accuracy after 500 iterations showed improvements over the single layer model in part I, with both models exceeding an accuracy rate of 87% on test datasets after 500 iterations.

Part III

In this section, the L1 and L2 regularisation methods were incorporated into the cross-entropy function of network architecture B.

Cross Entropy

```
def cross_entropy(y, pred, reg_rate, REG, weights1, weights2, weights3):
    entropy = -tf.reduce_sum(y * tf.math.log(pred + 1e-9), axis=1)
    loss = tf.reduce_mean(entropy)

    # regularization
    if REG == "L1":
        # L1 REGULARISATION
        l1 = reg_rate * (tf.reduce_sum(tf.math.abs(weights1)) + tf.reduce_sum(tf.math.abs(weights2)) + tf.reduce_sum(tf.math.abs(weights3)))
        loss = loss + l1
    elif REG == "L2":
        # L2 REGULARISATION
        l2 = reg_rate * (tf.reduce_sum(weights1 ** 2) + tf.reduce_sum(weights2 ** 2) + tf.reduce_sum(weights3 ** 2))
        loss = loss + l2

    return loss
```

For L1 regularisation, the 3 weights matrices are first subjected to the *abs* function to convert negative values to positive. This is because some negative weight values within the weight matrices would produce a final negative loss value. Once converted, each individual weight values within the weight matrices are summed together. The summed absolute weight values are then multiplied by the regularisation rate hyper-parameter to produce the regularisation cost. The regularisation cost will then be added to the normal loss score that was calculated as usual and returned to the main function as the final loss score.

Similar to L1 regularisation, L2 regularisation would be the sum of all squares of the weights within the 3 weight matrices. Due to squaring of weight values, initially negative weight values would become positive, therefore the values did not have to go through absolute value conversion. The summed squared weights would then be multiplied by the regularisation rate and added to the initial loss score to produce the final loss score.

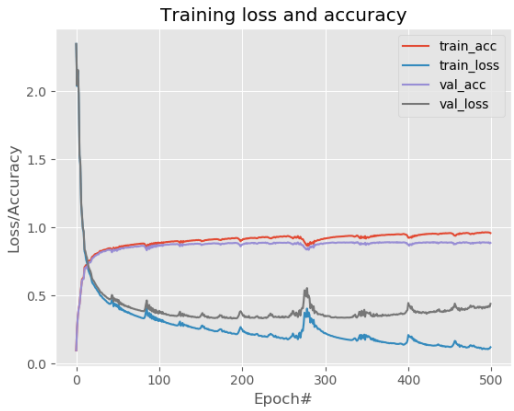
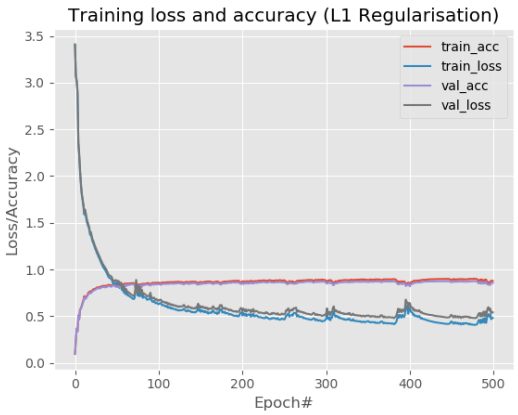
Results

The training operation on the implemented regularised models were done with the following hyper-parameters:

Regularisation rate = 0.0001, learning rate = 0.01, iterations=500

The test dataset was treated as validation data for each iteration and the accuracy/loss graph for the training is illustrated as below:

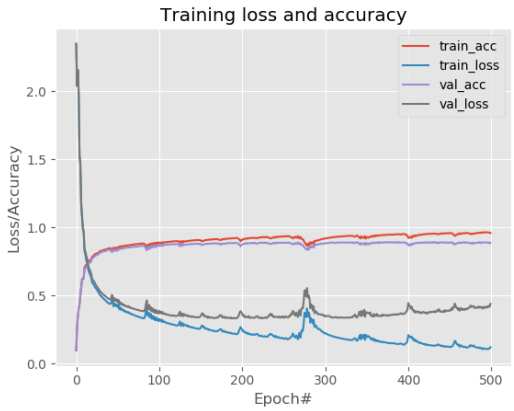
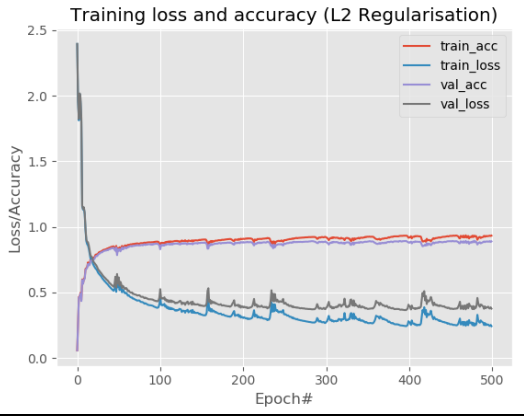
L1 Regularisation

	
Network Architecture B	Network Architecture B (with L1)

Models	**Accuracy (Test Set)**
Network Architecture B	87.94%
Network Architecture B (L1)	86.11%

From the table above comparing the accuracy/loss graphs between network architecture B and L1 regularisation enabled architecture, it is evident that there was a significant decrease in overfitting as the gap between the training pattern and validation pattern for both loss and accuracy have shrunk considerably in the L1 enabled architecture. Also, the performance table above also shows that there is only a slight decrease in accuracy for the L1 regularisation enabled model at 86.11% compared to 87.94% in network architecture B. This proves that the L1 regularisation method have successfully reduced the weights of the network and in turn, reduced overfitting.

L2 Regularisation

	
Network Architecture B	Network Architecture B (with L2)

Models	Accuracy (Test Set)
Network Architecture B	87.94%
Network Architecture B (L2)	88.83 %

Just like in L1 regularisation results, the accuracy/loss graphs above showed definite signs of reduced overfitting in the L2 enabled architecture when compared to network architecture B, as shown with the reduced gaps between validation and training loss patterns. From the performance table, it shows that the L2 regularisation enabled architecture model even outperforms network architecture B in accuracy with 88.83% accuracy rate.

Part B

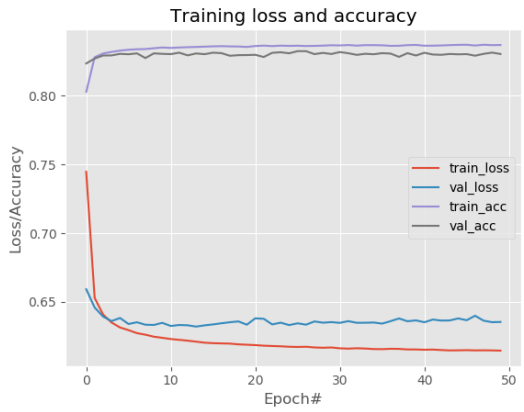
Task 1

Single Layer vs. 2-Layer Configuration

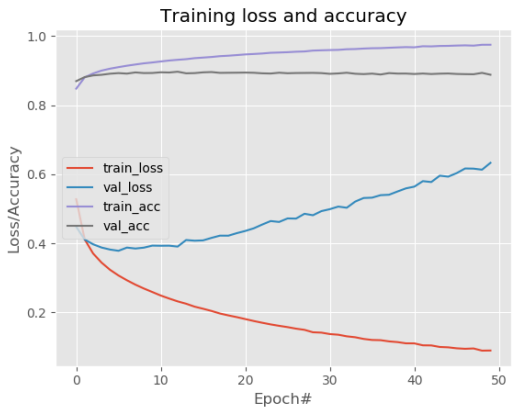
From the implementation results, it was observed that there was definitely an improvement in the 2 fully connected layer model (1 ReLU layer with 200 neurons and a SoftMax output layer) over the single SoftMax layer model. The training hyper-parameters are as follows:

Epochs = 50, validation_split = 0.1

The results can be observed with the training graphs of the two models below:



SoftMax only



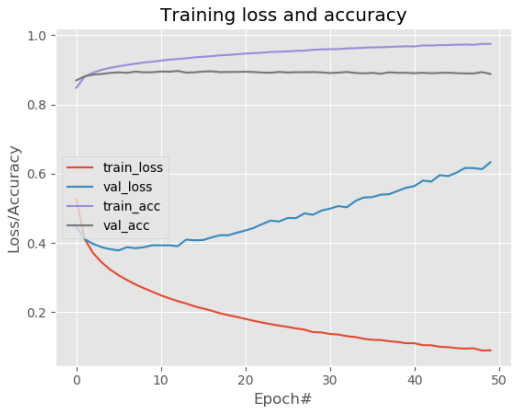
ReLU (200 neurons) + SoftMax

Model	Loss (Test Set)	Accuracy (Test Set)
SoftMax only	0.5466398373070885	85.59412%
ReLU (200 neurons) + SoftMax	0.5144767510332167	90.7%

The 2-layer model ended up with an accuracy rate of 90.7% while the SoftMax model returned a rate of 85.59%. A significant improvement in accuracy from the 2-layer model. The 2-layer model showed decreased loss value on the test dataset too. However, the 2-layer model showed extreme overfitting compared to the SoftMax model based on the loss graphs illustrated above.

Layer Configuration Comparison

To observe the impact of number of layers in a model have in a deep learning network, the 2-layer network from before was compared against a 3-layer network and a 4-layer network as specified in the assignment sheet. The resulting training graphs for all 3 networks will be illustrated below:

<table>
<tr>
<td> 
</td>
<td> 
</td>
</tr>
<tr>
<td>ReLU (200 neurons) + SoftMax</td>
<td>ReLU (400 neurons) + ReLU (200 neurons) + SoftMax</td>
</tr>
</table>

ReLU (600 neurons) + ReLU (400 neurons) + ReLU (200 neurons) + SoftMax

Model	Loss (Test Set)	Accuracy (Test Set)
ReLU (200 neurons) + SoftMax	0.5144767510332167	90.7%
ReLU (400 neurons) + ReLU (200 neurons) + SoftMax	0.7162836758600716	92.2%
ReLU (600 neurons) + ReLU (400 neurons) + ReLU (200 neurons) + SoftMax	0.7035305844043442	92.92353%

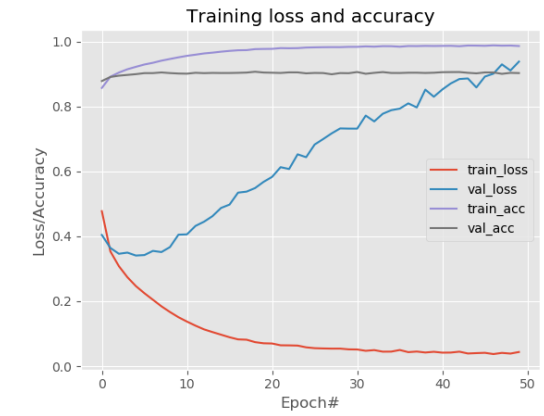
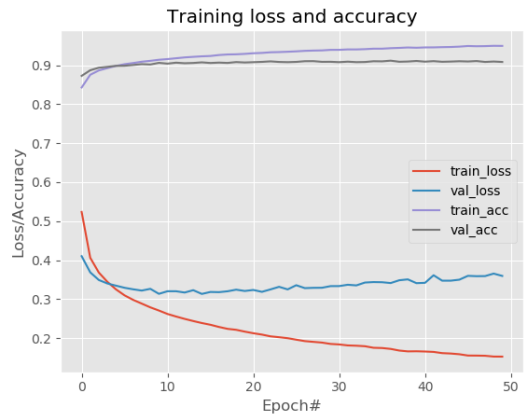
Based on the accuracy/loss graphs above, all 3 model configurations showed extreme signs of overfitting, especially the 3 and 4-layer models. Also, the loss scores of the 3 and 4-layer models were higher than the 2-layer model when tested with the test dataset. This might be due to the more severe overfitting experienced by the two deeper networks. However, the accuracy rates of the models did show a gradual increase as the layers in the network increases, as evident in the performance table.

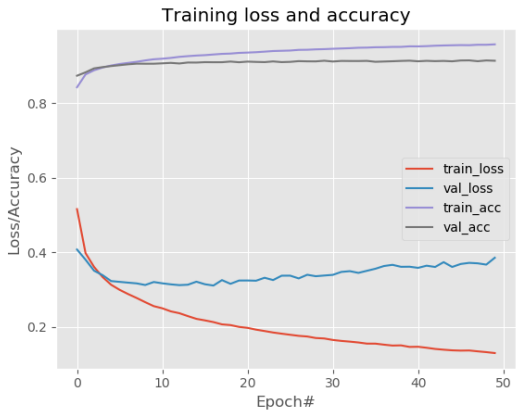
Task 2

In this task, dropout regularisation techniques were implemented on the two deepest networks from the previous task. The training hyper-parameters are as follows:

Dropout rate = 0.2, epochs = 50, validation split = 0.1

The resulting training graphs for the dropout implementations on the two networks will be compared with the original network's graph and illustrated below:

	
ReLU (400 neurons) + ReLU (200 neurons) + SoftMax	Dropout Enabled

	
ReLU (600 neurons) + ReLU (400 neurons) + ReLU (200 neurons) + SoftMax	Dropout Enabled

Model	Loss (Test Set)	Accuracy (Test Set)
ReLU (400 neurons) + ReLU (200 neurons) + SoftMax	0.7162836758600716	92.2%
(Dropout enabled) ReLU (400 neurons) + ReLU (200 neurons) + SoftMax	**0.28279827544727254**	**92.84118%**
ReLU (600 neurons) + ReLU (400 neurons) + ReLU (200 neurons) + SoftMax	0.7035305844043442	92.92353%
(Dropout enabled) ReLU (600 neurons) + ReLU (400 neurons) + ReLU (200 neurons) + SoftMax	**0.29868529496721796**	**93.22353%**

From the training graphs above, it can be observed that the dropout regularisation managed to reduce the overfitting phenomenon of the two networks by a considerable degree. This was concluded by observing the massive reduction in the gaps between the validation loss and the training loss. However, the overfitting phenomenon after dropout regularisation on the two networks have not been fully eliminated yet as the validation loss still showed slight gradual increases in pattern.

Major improvements in loss scores when tested on the test dataset was observed in both networks with dropout regularisations applied. The accuracy scores for both networks with dropout enabled increased slightly as well. This may be contributed to the reduction in loss increase of the models during training.

Part C: Research

The Adam optimization algorithm is an adaptive learning algorithm that attempts to provide specific rates of updates to each individual trainable parameter in a deep learning network such as weights and biases. It aims to deviate from the use of a single adjusted learning rate to update all training parameters, a trait we normally see used in stochastic gradient descent.

The Adam optimization algorithm achieves adaptive learning rates for each weight by estimating the first and second moments of a given gradient (Kingma and Ba, 2015), hence the name Adam (adaptive moment estimation). The Adam optimization algorithm utilizes exponentially moving averages (EMA) to compute the moment estimations. EMA is an algorithm that allows the tracking of the average value of a time series data. The formula for EMA is given below:

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t$$
$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2$$

Moving averages of gradient and squared gradient.

Where *g* denotes the gradient of the current time frame, *m* and *v* denotes first and second moments, and β denotes a new hyper-parameter of the algorithm to control the emphasis of the EMA put on newer moments. In the case of Adam, the β value for both *m* and *v* defaults to 0.9 and 0.999 respectively. The reason why EMA works for working out moving averages are due to the m_{t-1} portion of the algorithm, which can be expanded all the way from the current to the very beginning of a time series data. The illustration below demonstrates the expansion of the *m* attribute across several steps in a time series.

$$m_0 = 0$$
$$m_1 = \beta_1 m_0 + (1-\beta_1) g_1 = (1-\beta_1) g_1$$
$$m_2 = \beta_1 m_1 + (1-\beta_1) g_2 = \beta_1 (1-\beta_1) g_1 + (1-\beta_1) g_2$$
$$m_3 = \beta_1 m_2 + (1-\beta_1) g_3 = \beta_1^2 (1-\beta_1) g_1 + \beta_1 (1-\beta_1) g_2 + (1-\beta_1) g_3$$

As the expansion continues deeper, the gradients are multiplied by gradually smaller β values. Therefore, the less contribution the initial gradients contribute to the overall value. So, the EMA formula can essentially be simplified to:

$$(1-\beta_1) \sum_{i=0}^{t} \beta_1^{t-i} g_i$$

In Adam, it is assumed that the estimated moment parameters will be equal to the expected value of the estimator. With that assumption, the bias correction for the first moment estimator (Kingma and Ba, 2015) can be derived as such:

$$E[m_t] = E[(1-\beta_1)\sum_{i=1}^{t}\beta_1^{t-i}g_i]$$
$$= E[g_i](1-\beta_1)\sum_{i=1}^{t}\beta_1^{t-i} + \zeta$$
$$= E[g_i](1-\beta_1^t) + \zeta$$

Bias correction for the first momentum estimator

With the bias correction formula, the final bias corrected formulas for first and second moment estimation in Adam will be as follows:

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}$$
$$\hat{v}_t = \frac{v_t}{1-\beta_2^t}$$

Bias corrected estimators for the first and second moments.

Using the final first and second moment estimator formulas, the resulting moving average values can be used for individual adaptive rate scaling for individual parameters with the following formula:

$$w_t = w_{t-1} - \eta\frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

The above update rule formula is an example of weight update, but it can be used for other parameters as well such as bias. The *n* denotes the step size, while the *e* denotes the constant 10^{-8}. The pseudocode for the full Adam optimization algorithm (Kingma and Ba, 2015) will be illustrated below:

Algorithm 1: *Adam*, our proposed algorithm for stochastic optimization. See section 2 for details, and for a slightly more efficient (but less clear) order of computation. g_t^2 indicates the elementwise square $g_t \odot g_t$. Good default settings for the tested machine learning problems are $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$. All operations on vectors are element-wise. With β_1^t and β_2^t we denote β_1 and β_2 to the power t.

Require: α: Stepsize
Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates
Require: $f(\theta)$: Stochastic objective function with parameters θ
Require: θ_0: Initial parameter vector
  $m_0 \leftarrow 0$ (Initialize 1st moment vector)
  $v_0 \leftarrow 0$ (Initialize 2nd moment vector)
  $t \leftarrow 0$ (Initialize timestep)
  while θ_t not converged **do**
    $t \leftarrow t + 1$
    $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep t)
    $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
    $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
    $\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
    $\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
    $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$ (Update parameters)
  end while
  return θ_t (Resulting parameters)

Pseudocode for Adam Optimizer Algorithm

References

Kingma, D. and Ba, J., 2015. *Adam: A Method For Stochastic Optimization*. [online] arXiv.org. Available at: <https://arxiv.org/abs/1412.6980> [Accessed 4 April 2020].