The 10h IEEE International Conference on Dependable Systems, Services and Technologies, DESSERT'2019
5-7 June, 2019, Leeds, United Kingdom

Non-Random Weight Initialisation in Deep Learning Networks for Repeatable Determinism

Richard N M Rudd-Orthner[1,2], Lyudmila Mihaylova[1]
[1] University of Sheffield, Sheffield, UK, {RNMRudd-Orthner1, L.S.Mihaylova}@sheffield.ac.uk
[2] MASS KSA (a Cohort plc company), Riyadh, KSA, {rruddorthner@mass.co.uk}

Abstract—**This research is examining the change in weight values of deep learning networks after learning. These research experiments require to make measurements and comparisons from a stable set of known weights and biases before and after learning is conducted, such that comparisons after learning are repeatable and the experiment is controlled. As such the current accepted schemes of random number initialisations of the weight values may need to be deterministic rather than stochastic to have little run to run varying effects, so that the weight value initialisations are not a varying contributor. This paper looks at the viability of non-random weight initialisation schemes, to be used in place of the random number weight initialisations of an established well understood test case. The viability of non-random weight initialisation schemes in neural networks may make a network more deterministic in learning sessions which is a desirable property in mission and safety critical systems. The paper will use a variety of schemes over number ranges and gradients and will achieve a 97.97% accuracy figure just 0.18% less than the original random number scheme at 98.05%. The paper may highlight that in this case it may be the number range and not the gradient that is effecting the achieved accuracy most dominantly, although there may be a coupling of number range with activation functions used. Unexpectedly in this paper, an effect of numerical instability will be discovered from run to run when run on a multi-core CPU. The paper will also show the enforcement of consistent deterministic results on an multi-core CPU by defining atomic critical code regions aiding repeatable Information Assurance (IA) in model fitting (or learning sessions).**

Keywords— Repeatable Deep Learning Networks, Real-Time Single Processor Affinity, Non Random Weight Initialization, Security and Information Assurance, Safety-Critical AI, Learning Session Determinism.

I. INTRODUCTION

Artificial Intelligence (AI) has the potential for growth in many areas, particularly the use of Deep Learning Networks and frameworks, but applications for Mission Critical and Safety critical software has additional challenges in security in terms of Information Assurance (IA). It may be argued that the application of Artificial Intelligence and Deep Learning Networks in particular have goals for replicating or challenging human abilities against a human performance baseline. Although, Mission Critical and Safety Critical software has goals of completeness, correctness and repeatability making it rigorous both in the development and in the deployed application performance, arguably to reach a performance that is "more than human", in that it reduces human error. With this consideration the application of deep learning networks has challenges when applied to Mission Critical and Safety Critical software in terms of gaining understanding and confidence for verification and validation of the machine learnt generalisation model, and that is a challenge for Information Assurance both in the formed generalisation model but also in the processes that formed that model. There has already been research in this area for a number of years with a number of papers, some of the most relevant to safety critical applications are in unmanned air vehicles [1], the automation of space missions [2] and also in space mission telecoms fault tolerance [3].

An advantage of the deep learning network approach, is that it has the ability to form generalisation models that can perform tasks that may be considered intractable by traditional approaches. However, without controls can form a solution that is not compliant to known understanding or real world physics. For mission critical and safety critical systems repeatability and determinism are desirable features for verification and validation, both for the processing to form the generalisation model, and when making a prediction with the model when deployed. As both repeatable and deterministic aspects are desirable attributes and also form an experimental control, one aspect that may make a disruption to this is the use of a random number initialisation state of weights before learning. This paper looks at different number ranges and gradients for a weight initialisation scheme to be used in place of a random number initialisation state. Three main non-random initialisation weight schemes are experimented with: constant value, uniform linear ramp and sinusoid. In each non-random scheme the number range and gradient are changed: the constant value scheme has no gradient and no number range, the linear ramp scheme has a constant gradient and controlled number range, and lastly the sinusoidal scheme has constant variations in the gradient with a controlled number range. The three schemes employed provide discriminations between number ranges used and the gradient slope of those values used in those schemes.

The research contribution that this paper is seeking to provide is to answer a research question, that is: "Are random number initialisation weight values required as the initial state before learning to have high accuracy in predictions or can a non random scheme also have comparable performance in those predictions".

This paper is a foundation environment for subsequent research experimental work that will examine changes and adaption to weight and bias matrixes before and after learning model fitting sessions. The foundation environment is using Anaconda Python, NumPy and Keras with TensorFlow deep machine learning framework accessed through the Jupyter Notebook web services environment.

This work was required to establish a repeatable result with a defined known initial state that has comparable performance to the existing random initialisation schemes used currently. The experiment's initial state before learning is to be defined, known and predictable such that it may be controlled and accounted for in results as a deterministic initial state that forms an experiment control. In the experiments a well understood problem that is published will be used. The MNIST dataset in TensorFlow with Keras and NumPy. This is an application of recognising hand written text characters and although in itself may not be a mission critical problem it is a mature reviewed solution.

II. Original Baseline Code Example

This example is familiar to researchers and is being used to demonstrate weight initialisations that are not using random number sequences.

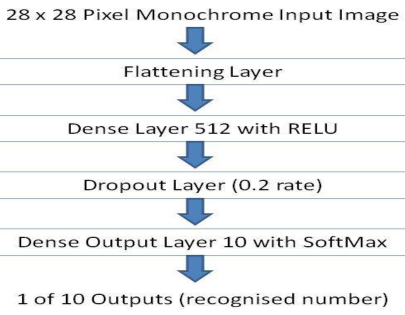

28 x 28 Pixel Monochrome Input Image
Flattening Layer
Dense Layer 512 with RELU
Dropout Layer (0.2 rate)
Dense Output Layer 10 with SoftMax
1 of 10 Outputs (recognised number)

Figure 1. Architectures of the Baseline Example.

The code specimen in Appendix A (A.1) is used as an open source example [4], it is known as the hello world of Neural Networks. When this code is run the output from the evaluate command provides both Loss and Accuracy figures, from five consecutive reset runs it is noted that the losses and accuracies vary from run to run in each learning session.

```
[0.06606189897235017, 0.9805]        [0.07090263138680603, 0.9788]
[0.06552187514795223, 0.9815]        [0.07510031864168705, 0.9769]
[0.06914228669836302, 0.979]
```

Figure 2: Hello World example results.

There is a variation in both the loss and the accuracy figures, which presumably means that there is some variation in the prediction performance depending on the model Fitting using Shuffles and the initialisation of weights. From the five runs the mean average loss is 0.069345802 and the mean average accuracy is 0.97934. The reasoning for this variation run to run may be considered to be due to the random number initialisation values present in the weight values and the shuffle ordering of the dataset before each learning session, Therefore setting the random number seed values before each learning session should make the random number sequence repeatable and therefore the accuracy and loss values results the same from each run to run.

III. Seeding the Random Number Generator

Running the same model with the no shuffle added to the fit command, and again using the evaluate command's loss and accuracy figures the results are gathered. At this point the number of epochs is reduced to one to reduce the runtime duration. Tensor Flow and NumPy random seeds have been set to form a baseline value from one epoch that should make the random number sequence defined. The code for seeding the random number generators [5] and the modified fit command are shown in Appendix A (A.2): The five run results are as follows:

```
[0.09898285598997027, 0.971]         [0.1022148139256984,  0.969]
[0.09992996315583587, 0.9699]        [0.09846471949797124, 0.9709]
[0.10279747271370143, 0.9693]
```

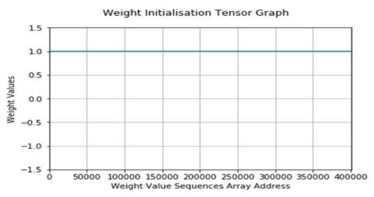
Figure 3: Single epoch random number seeded baseline results

From these five results we can see that the variation is still present, although arguably the mean accuracy may have lowered by a 0.9% with mean values for loss and accuracy of 0.100477965 and 0.97002. These values without the shuffle and with the single epoch form the comparison baseline for further measurement experiments. This might be an indicator that this variation run to run is not attributable to the initialisation of the weights alone. But changing the model construction code so that the random initialize values can be substituted with fixed values is shown in Appendix A (A.3):

Three initialisation schemes will be experimented with and these are: constant values, linear ramps and sinusoids, these will test different numerical aspects from values, gradients to number ranges.

IV. Constant Value Scheme

Now starting with a weight initialized array of 28*28*512 in the second layer and 512*10 weight values in the fourth layer that are all containing a constant value 1.0. The code in Appendix A (A.4) was used to initialise those weights, and was run five times and provided these results.



{+1.0…+1.0}

Figure 4: Constant Value (1.0) weight initialisation tensor

The five run results are as follows:

```
[14.490169499206543, 0.101]          [14.490169499206543, 0.101]
[14.490169499206543, 0.101]          [14.490169499206543, 0.101]
[14.490169499206543, 0.101]
```
Figure 5: Constant Value (1.0) results

This time the accuracy and loss values are deterministic, but the accuracy is very much lower, about 86% lower than the baseline perhaps indicating that learning is not occurring, also the loss is greater than the baseline values. But the values have no or very low variance run to run but perhaps are due to the loss in learning. So using the code in Appendix A (A.5) the initialized weight values are set to zeros instead, and gains the following results from the five runs.

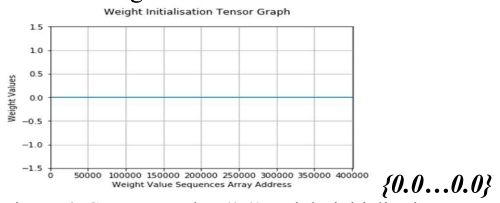


Figure 6: Constant Value (0.0) weight initialisation tensor

The five run results are as follows:

```
[2.3011608791351317, 0.1135]          [2.301160814285278,  0.1135]
[2.30116091003418,    0.1135]          [2.30116091003418,    0.1135]
[2.3011607849121094, 0.1135]
```
Figure 7: Constant Value (0.0) results

Again the accuracy values have no variance but the loss has a small variance run to run but is stable. Again the mean accuracy is much reduced and is about 86% lower than the baseline perhaps indicating that the loss in learning may be due to drop outs and saturations with those values. Looking into the Keras code [6] it is noted that the random initialisation value would have been between -0.05 and +0.05. So using a fixed constant value of +0.05 as an experiment the code in Appendix A (A-6) is used:

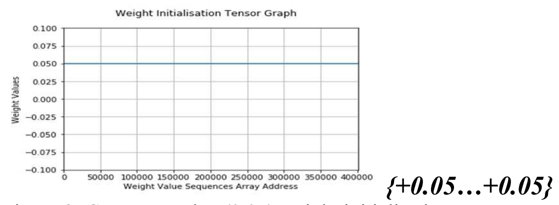


Figure 8: Constant Value (0.05) weight initialisation tensor

The five run results are as follows:

```
[1.8057675338745116, 0.2667]          [1.7881868183135987, 0.2766]
[1.7912575538635254, 0.2739]          [1.7899974334716797, 0.2739]
[1.7860924480438232, 0.2817]
```
Figure 9: Constant Value (0.05) results

At the +0.05 value the variances of the loss and accuracy in the five runs has increased again and the mean accuracy is about 70% less than the baseline mean, however, when an initialisation value of 1.0 was used there was no variance, this appears to change if the final layer's SoftMax activation function is replaced with ReLU and using the code in Appendix A (A-7) the five runs are repeated using ReLU:

The five run results are as follows:

```
    [2.3025851249694824, 0.0978]  [2.3025851249694824, 0.0978]
    [2.3025851249694824, 0.0978]  [2.3025851249694824, 0.0978]
    [2.3025851249694824, 0.0978]
```
Figure 10: Constant Value (0.05) results with ReLU in place of SoftMax

The variance in the loss and accuracy has diminished again. Although the accuracy is now 86% lower than the baseline, it is possible that the SoftMax function that uses summations of exponential numbers is having significant bit representation issues but the ReLU may be less vulnerable to that and has a consistent result. Possibly the run to run variations could be an effect of task scheduling and may be creating variations in the resultant numbers depending on if the CPU processor's internal 80bit extended precision floating point register [7] is interrupted. Upon which it may truncate the calculation to 32bits or 64bits when the value is stored by the task scheduler and cause a lower number of significant bits to be passed on to the next calculation when the task is next scheduled, However, this is not proven. But it is also noted that the ReLU is not allowing the learning to occur and has broken the model, so the SoftMax is put back in and more experiments are conducted using the other extreme of the original random number range, the -0.05 value is used as a constant value with the code in Appendix A (A-8) and the SoftMax put back into the final layer.

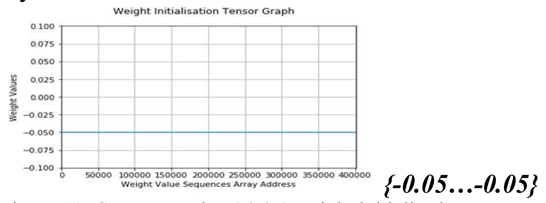


Figure 11: Constant Value (-0.05) weight initialisation tensor

The five run results are as follows:

```
[2.301160791397095,   0.1135]          [2.3011607082366945, 0.1135]
[2.3011608280181886, 0.1135]          [2.3011609703063964, 0.1135]
[2.301160668563843,   0.1135]
```
Figure 12: Constant Value (-0.05) results

Again the loss value has low variance run to run and the accuracy is deterministic but is a much lower accuracy. This compares with the constant zero value experiment although the +0.05 constant has a higher mean accuracy. This implies there is a sensitivity to the initialisation value and that the initial value as a potential to allow learning but also cause differences in the results depending on the value used. A summary table follows of the experiments with the single epoch baseline and the constant value weight initialisation schemes:

Experiment	Loss and Accuracy	Comment
Constant 1.0 values	[14.490169499206543, 0.101] [14.490169499206543, 0.101] [14.490169499206543, 0.101] [14.490169499206543, 0.101] [14.490169499206543, 0.101]	No Variances in Loss and Accuracy. also seen if the SoftMax was replaced with ReLU, 87% lower accuracy in this case.
Constant 0.0 values	[2.3011608791351317,0.1135] [2.301160814285278, 0.1135] [2.30116091003418, 0.1135] [2.30116091003418, 0.1135] [2.3011607849121094,0.1135]	Accuracy stable but Variances in Loss at the 7th significant place, 86% lower accuracy similar accuracy to the -0.05 experiment.
Constant + 0.05 values	[1.8057675338745116, 0.2667] [1.7881868183135987, 0.2766] [1.7912575538635254, 0.2739]	Variances in Loss at the 2nd significant place and Accuracy

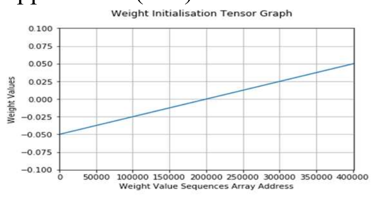
	[1.7899974334716797, 0.2739] [1.7860924480438232, 0.2817]	unstable, 70% lower accuracy. But is the highest accuracy of the constant value experiments.
Constant -0.05 values	[2.301160791397095, 0.1135] [2.3011607082366945,0.1135] [2.3011608280181886,0.1135] [2.3011609703063964,0.1135] [2.301160668563843, 0.1135]	Accuracy stable and Variances in Loss at the 7th significant place, 86% lower accuracy like the zero number experiment.

Figure 13: Constant Value summary table

It may appear that although there is an unexplained variance run to run and given that the random number generator had been seeded there is still an unexplained variation in results. It also may appear that using a constant number as an initialised value has a large impact on the resultant accuracy, and that some of that accuracy may be connected to the value used. It may be that the value 1.0 may have caused some arithmetic problems with number representations, and values between -0.05 and +0.05 as per the current random number initialisation range showed variances in results but very much lower accuracy. The values greater than zero had higher accuracy and lower loss but that may be because of the use of the ReLU in the second layer and a value 0.0 perhaps coursing drop outs, and that variation may be and effect of SoftMax in the final layer. These results may of course only pertain to this model and particularly if there explanation intuition is based on that models activation function architecture, but it shows a concern with 32bit number floating point significant bits, and noticing that the input data is positive image values in the scale 0 - 255 which is rescaled to 0 - 1 and the weight initialisation value of +0.05 is two significant places different. Where a five significant place difference is experienced in the computation this will begin to effect a 32bit calculation representation accuracy. Although the 32bit number accuracy is not conclusively proven, it is a concern and may be supported by the experiment that excluded the SoftMax activation function, as the SoftMax function divides a number by the sum of an exponential number. Alternatively this may be solved by using a 64bit number for the accumulator in the sum or the use of pre and post scaling of the number scales before and after the calculations. It should also be noted that results show a lower mean accuracy are shown with constant weight initialisation values used. Perhaps the learning is more uniformly effecting adjacent neurons by a similar amount and it could be expected by initial weight values that have no gradient or number range variation at the outset of learning and the next set of experiments should have a number range as a linear ramp to have a number range and fixed gradient.

V. Linear Ramp Scheme

Using a linear ramp between -0.05 and +0.05 as the initial values of the weights to provide areas of the neural network that will have different dominance towards an output from the outset of learning, and a gradient of values and number range in those initial weights may be higher performing. The initialisation weight value code is in Appendix A (A-9):



{-0.05…+0.05}

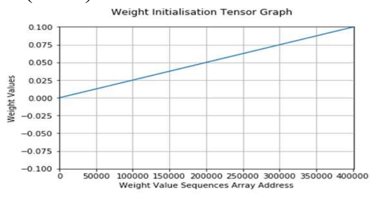
Figure 14: Linear Ramp (-0.05 to +0.05) weight initialisation tensor

The five run results are as follows:

[0.1565102383375168, 0.9523]	[0.15511292833536863, 0.9546]
[0.1435071627393365, 0.9561]	[0.15252709869667888, 0.9552]
[0.1440581636864692, 0.956]	

Figure 15: Linear Ramp (-0.05 to +0.05) results

It seems that a gradient and number range of values may be helpful and the accuracy is just 2% less than the baseline results. However, taking the -0.05 constant value case that was low performing and the higher performing +0.05 constant value the ramp will be modified from +/- 0.05 to be 0.0 to 0.1 to have a different number range but the same gradient. The initialisation code is in Appendix A (A.10):



{0.0…+0.1}

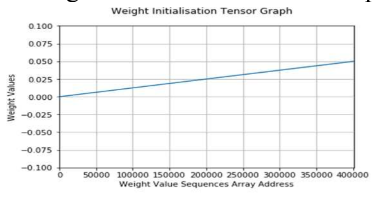
Figure 16: Linear Ramp (0.0 to 0.1) weight initialisation tensor

The five run results are as follows:

[0.24299218244701623, 0.9236]	[0.2762062883064151, 0.9141]
[0.2508674868822098, 0.9212]	[0.23018671630620957, 0.9275]
[0.2593486599966884, 0.9187]	

Figure 17: Linear Ramp (0.0 to 0.1) results

It seems that the results are similar but a little reduced then the ramp over 0 and is 5% lower than the baseline in accuracy. The gradient was unchanged but the number range was slid to positive numbers only, but that number range reached a number range greater than the original initialisation codes value range of the +0.05 value. In the next experiment the number range and gradient are changed to a ramp in the number range 0.0 to +0.05, and is using the initialisation code in Appendix A (A.11):



{0.0…+0.05}

Figure 18: Linear Ramp (0.0 to 0.05) weight initialisation tensor

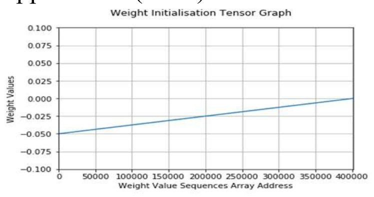
The five run results are as follows:

[0.2146800939079374, 0.934]	[0.21725718629658222, 0.9339]
[0.2175400296010077, 0.9317]	[0.21510434658303856, 0.9319]
[0.22056258716955782, 0.932]	

Figure 19: Linear Ramp (0.0 to 0.05) results

These results are very similar but it would be worth trying the negative value range -0.05 to 0.0 for completeness as there may be a difference between the

SoftMax and ReLU layer's activations and the benefit of each activation function needs from the initialisation values. The initialisation code for this experiment is in Appendix A (A-12):



{-0.05…0.0}

Figure 20: Linear Ramp (-0.05 to 0.0) weight initialisation tensor

The five run results are as follows:

[2.301160803604126, 0.1135]	[2.301160865020752, 0.1135]
[2.3011607265472414, 0.1135]	[2.301160787200928, 0.1135]
[2.301160820388794, 0.1135]	

Figure 21: Linear Ramp (-0.05 to 0.0) results

The results are very much lower almost like the negative values that were seen with the constant values perhaps suggesting that layer 2 ReLU activations may have drop outs. A summary table of the results follows:

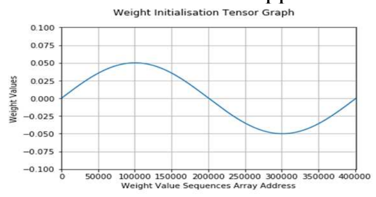
Experiment	Loss and Accuracy	Comment
Ramp -0.05 to +0.05	[0.1565102383375168, 0.9523] [0.15511292833536863,0.9546] [0.1435071627393365, 0.9561] [0.15252709869667888,0.9552] [0.1440581636864692, 0.956]	Variance in numbers, but only 2% lower accuracy from the baseline in this case.
Ramp 0.0 to +0.1	[0.24299218244701623,0.9236] [0.2762062883064151, 0.9141] [0.2508674868822098, 0.9212] [0.23018671630620957,0.9275] [0.2593486599966884, 0.9187]	Variance in numbers, 5% lower accuracy from the baseline in this case.
Ramp 0.0 to +0.05	[0.2146800939079374, 0.934] [0.21725718629658222,0.9339] [0.2175400296010077, 0.9317] [0.21510434658303856,0.9319] [0.22056258716955782,0.932]	Variance in numbers, 4% lower accuracy from the baseline in this case.
Ramp -0.05 to 0.0	[2.301160803604126, 0.1135] [2.301160865020752, 0.1135] [2.3011607265472414,0.1135] [2.301160787200928, 0.1135] [2.301160820388794, 0.1135]	No variances in the accuracy and in Loss at the 7th significant place, the Accuracy stable. 86% lower accuracy to the baseline but similar accuracy to the zero number experiment, perhaps 0 or negative numbers don't match the layer 2 ReLU.

Figure 22: Linear Ramp summary table

From these results negative number ranges seem to be low performing and positive values higher performing although the range between -0.05 to +0.05 was the highest performing in terms of accuracy. The gradient changed between the experiments with 0.0 to +0.05 and 0.0 to +0.1 but had little difference in results, but the number range and gradient were changed together. In the next set of experiments the gradient and number range are changed independently using a sinusoid.

VI. Sinusoidal Schemes

A moving gradient is used starting with the scale -0.05 to +0.05 in a sinusoidal form such that the number range is the same but the gradient is changing with respect to the linear ramp experiment of the same range. The initialisation code is in Appendix A (A-13).



0.05 sin (x)

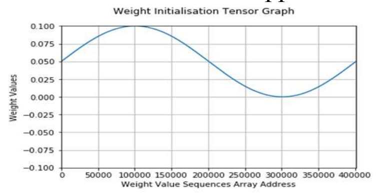
Figure 23: Sinusoid (-0.05 to 0.05) weight initialisation tensor

The five run results are as follows:

[0.14248031044751405, 0.9575]	[0.14802057081907988, 0.9561]
[0.15482748659588397, 0.9546]	[0.14560777206234635, 0.9565]
[0.15966865147389472, 0.9535]	

Figure 24: Sinusoid (-0.05 to 0.05) results

The results are similar to the linear ramp over the same number range which was also only 2% lower than the baseline, Using the positive figure experiment with the same sinusoidal pattern in the range 0 to 0.1 the initialisation code is in Appendix A (A-14):



0.05 sin (x) + 0.05

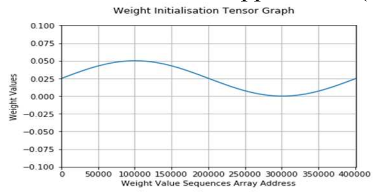
Figure 25: Sinusoid (0.0 to 0.1) weight initialisation tensor

The five run results are as follows:

[0.2765421679884195, 0.9159]	[0.27466120897680524, 0.916]
[0.274703558498621, 0.9164]	[0.2757590222135186, 0.9156]
[0.27346776156574487,0.9172]	

Figure 26: Sinusoid (0.0 to 0.1) results

The accuracy is 6% lower than the baseline. The next experiment also uses positive values, but only in the range 0.0 to +0.05 with the same sinusoidal form using the initialisation code in Appendix A (A-15).



0.025 sin(x)+ 0.025

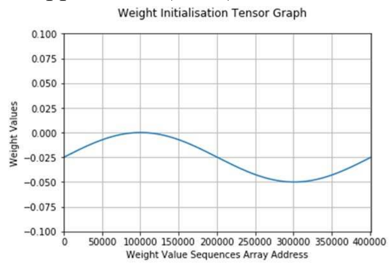
Figure 27: Sinusoid (0.0 to 0.05) weight initialisation tensor

The five run results are as follows:

[0.18291570566408336, 0.9453]	[0.19241121173687278, 0.942]
[0.18905000345483422, 0.9418]	[0.1834044139198959, 0.9446]
[0.1803453653005883, 0.9448]	

Figure 28: Sinusoid (0.0 to 0.05) results

Slightly lower results at almost 3% less than the baseline, but for completeness the sinusoidal range of -0.05 to 0.0 is provided below with the initialisation code in Appendix A (A-16).



0.025 sin (x) - 0.025

Figure 29: Sinusoid (-0.05 to 0.0) weight initialisation tensor

The five run results are as follows:

```
[2.301160641479492,  0.1135]        [2.301160855102539,  0.1135]
[2.301160636138916,  0.1135]        [2.3011608177185057, 0.1135]
[2.3011608085632322, 0.1135]
```

Figure 30: Sinusoid (-0.05 to 0.0) results

The summary of these experiments using sinusoidal patterns are shown below:

Experiment	Loss and Accuracy	Comment
sin -0.05 to +0.05	[0.14248031044751405,0.9575] [0.14802057081907988,0.9561] [0.15482748659588397,0.9546] [0.14560777206234635,0.9565] [0.15966865147389472,0.9535]	Same score as the same number range with the ramp experiment.
sin 0.0 to +1.0	[0.2765421679884195, 0.9159] [0.27466120897680524,0.916] [0.274703558498621, 0.9164] [0.2757590222135186, 0.9156] [0.27346776156574487,0.9172]	Almost the same score as the same number range with the ramp experiment but 1% lower at 6% lower than the baseline.
sin 0.0 to +0.05	[0.18291570566408336,0.9453] [0.19241121173687278,0.942] [0.18905000345483422,0.9418] [0.1834044139198959, 0.9446] [0.1803453653005883, 0.9448]	Almost the same score as the same number range with the ramp experiment but 1% higher at 3% lower than the baseline.
sin -0.05 to 0.0	[2.301160641479492, 0.1135] [2.301160855102539, 0.1135] [2.301160636138916, 0.1135] [2.3011608177185057, 0.1135] [2.3011608085632322, 0.1135]	This result also coincides with the ramp of the same number range.

Figure 31: Sinusoid summary table

Taking the highest score of the sinusoid number range of -0.05 to +0.05 and re-running with the five epochs and enabling the shuffle provides the following results as a comparison to the original code:

```
[0.06944945766797755, 0.9792]       [0.07057002582962159, 0.9793]
[0.0718287119449582,  0.9803]       [0.07078138581812382, 0.9809]
[0.07259123737227055, 0.9789]
```

Figure 32: 5 epoch and shuffle with high score sinusoid

In comparison with the original untouched code, the results are shown below and the variance run to run is similar and the accuracy is about the same as the baseline but is not using random weight initialisations:

```
[0.06606189897235017, 0.9805]       [0.07090263138680603, 0.9788]
[0.06552187514795223, 0.9815]       [0.07510031864168705, 0.9769]
[0.06914228669836302, 0.979]
```

Figure 33: Original code 5 epoch and shuffle with random init weights

However, although these results seam to show that an non-random initialisation state can be just as high performing in prediction accuracy as the random initialisation state, the run to run variation in results is masking the accuracy measurements and needs to be tackled to improve the measurement accuracies made for both the experiment and the baseline values.

VII. Tackling the Repeatability Run to Run

There is still a variance run to run in the results even using the seeded random numbers with non random weight value initialisations but taking into account the possibility of the scheduling causing variations in number representations. An experiment to try an invoke the real-time priority of the windows scheduler [8] with an affinity to one processor [9] as an attempt to deny or reduce interruption of the task thread and uses the code in Appendix A (A-17). However, the variation in the five runs is still present, and under investigation it seems to show that **real-time priority** is not being set and it is being set to **high priority** instead. It turns out that **you need to run Jupyter notebook in a cmd console as administrator** such that the real-time priority can be selected and then it becomes completely repeatable in each of the five runs. This supports the theory that task scheduling is interrupting and truncating calculations in the internal CPU 80bit extended precision floating point register [7], as now the python task is running on one processor uninterrupted. This provides an accurate repeatable figure for the highest scoring sinusoid scheme.

```
[0.07358874179359991, 0.9772]       [0.07358874179359991, 0.9772]
[0.07358874179359991, 0.9772]       [0.07358874179359991, 0.9772]
[0.07358874179359991, 0.9772]
```

Figure 34: high score sinusoid, on a single processor

Now that the runs are consistent the highest score number ramp scheme with a range of -0.05 to 0.05 is re-run with no variance in the results and they are similar suggesting that the initialisation is providing repeatability:

```
[0.0668453144104511, 0.9784]        [0.0668453144104511, 0.9784]
[0.0668453144104511, 0.9784]        [0.0668453144104511, 0.9784]
[0.0668453144104511, 0.9784]
```

Figure 35: high score linear ramp, on a single processor

It appears that the ramp is a very slightly better initialisation scheme then the sinusoid of the same number range dismissing the effect of initial varying gradients being of a benefit to the resultant accuracy and loss, at least in this case. Although repeatable results that a comparable score to the baseline is achieved the earlier concern of numerical stability of the SoftMax activation function is investigated.

VIII. Custom Number Scaled SoftMax Function

However, also an experiment of the SoftMax activation function used in the final layer, the code in Appendix A (A-18) is used to define a SoftMax with a rescaling for numerical stability [10] as was suggested as a possible concern earlier. But there is still no variance in the results and they are similar suggesting that the numerical stability is having a minor effect although this is the highest accuracy score yet, except for the original baseline, see below for the five results:

```
[0.06786308663240634, 0.9787]       [0.06786308663240634, 0.9787]
[0.06786308663240634, 0.9787]       [0.06786308663240634, 0.9787]
[0.06786308663240634, 0.9787]
```

Figure 36: high score sinusoid, on a single processor replaced SoftMax

A very marginal increase in accuracy, but now that the model can be run repeatedly, the original code is run in real-time priority with a single processor affinity and with the random number initialisation of the weights but seeded.

```
[0.061059941675240405, 0.9805]      [0.061059941675240405, 0.9805]
[0.061059941675240405, 0.9805]      [0.061059941675240405, 0.9805]
[0.061059941675240405, 0.9805]
```

Figure 37: Baseline, on a single processor

The baseline perfected value is 98.05% which is only 0.18% better than using the best linear ramp with a modified SoftMax, or just 0.21% better then the best linear ramp initialisation with the original SoftMax function and also just 0.33% better than using the best sinusoidal ramp weight initialisation.

IX. Conclusion

In summary the initial original code has a accuracy of about 98.05% and using random numbers, conventional thoughts might be that the weight values and random numbers were responsible alone for the variations in successive results run to run. However, when the random seeds are set to a defined seed value the variation in the results continues run to run. The paper was able to establish a stable result making the processing deterministic but the exact cause of the variations in successive runs is not proven, but could be a numerical stability given calculations using 32bit or 64bit floating point maths but would need to be combined with another effect like task scheduling truncating the stored values between schedules. The solution to the variation run to run suggests that it may be task scheduling truncating a CPU internal 80bit extended precision register used for floating point maths used even with 32bit or 64bit calculations and will truncate a calculations result to 32bits or 64bits if interrupted by the task scheduler. It may be that this variation is only seem on multi core CPUs and GPUs may be immune.

The paper also tested a variety of initialisation schemes and they were experimented with and 0.18% less accuracy than the original code was achieved with a non-random number weight initialization pattern and in that case was a linear ramp in the numerical range -0.05 to +0.05 with a modified SoftMax function. But it should be noted that more optimal non-random schemes may exist but the paper has shown that random number initialisation is not an imperative requirement. It also may be that the number range could be optimised but we may expect that these may also couple with the activation function used in that layer. It is also possible that the 32bit or 64bit number representation may be contributing to a regularisation effect to help to not over fit a model by reducing significant bit resolution. It may follow that initialisation schemes could be set depending on the layer type, the activation function and the regularisation scheme used. The 80bit floating point representation could conceivably have benefits to achieved accuracy with an optimal number range but also has benefits to determinism in successive run results and that may have benefits to make a Deep Learning network capability accessible to mission and safety critical systems. However, the paper has demonstrated deterministic repeatable results in successive runs without random initialisations meaning that mission and safety critical applications may have the test and qualification determinism required by those applications and the test environment is viable for further experimentation control.

It has not been experimented with in this paper, but it is also possible that re-compiling the tensor flow backend with strict IEEE compliance could provide all cores hyper threading with run to run repeatability determinism. But this might not provide the 80bit extended precision benefits to accuracy in learning sessions and the SoftMax experiment might indicate that number representation is a feature that could affect regularisation. Thus the current implementation with the real-time single processor affinity provides the flexibility to define critical regions during learning and evaluation. But it was the research question of "Are random number initialisation weight values required as the initial state before learning to have high accuracy in predictions or can a non random scheme also have comparable performance in predictions" and the answer is that it is possible to use non random sequences and random numbers in this case it may not be an imperative requirement for accuracy performance. Although, it is also possible that coupling in those schemes may connect with: the deep learning architecture, the layer type and the activation function used. Also the absence of random numbers with the use of alternative non-random number schemes can be used to provide repeatable deterministic results from learning session to learning session, and that might be a support to mission and safety critical system's verification and validation obligations going forward.

X. References

[1] Ernest et al, N. (2019). Genetic Fuzzy based Artificial Intelligence for Unmanned Combat Aerial Vehicle Control in Simulated Air Combat Missions. [online] Research Gate. Available at: https://www.researchgate.net/profile/Nicholas_Ernest/publication/301944635_Genetic_Fuzzy_based_Artificial_Intelligence_for_Unmanned_Combat_Aerial_Vehicle_Control_in_Simulated_Air_Combat_Missions/links/576c4e5408ae193ef3a9a384/Genetic-Fuzzy-based-Artificial-Intelligence-for-Unmanned-Combat-Aerial-Vehicle-Control-in-Simulated-Air-Combat-Missions.pdf [Accessed 29 Jan. 2019].

[2] Freitas Jr. et al, R. (2019). Advanced Automation for Space Missions1. [online] Rfreitas.com. Available at: http://www.rfreitas.com/Astro/AASMJAS1982.htm [Accessed 29 Jan. 2019].

[3] Lawson, D. and James, M. (2019). SHARP: A multi-mission artificial intelligence system for spacecraft telemetry monitoring and diagnosis. [online] Ntrs.nasa.gov. Available at: https://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/19900009128.pdf [Accessed 29 Jan. 2019].

[4] Google, TensorFlow. (2019). TensorFlow. [online] TensorFlow. Available at: https://www.tensorflow.org/tutorials/ [Accessed 1 Jan. 2019].

[5] Brownlee, J. (2019). How to Get Reproducible Results with Keras. [online] Machine Learning Mastery. Available at: https://machinelearningmastery.com/reproducible-results-neural-networks-keras/ [Accessed 2 Jan. 2019].

[6] Allaire, J. (2019). rstudio/keras. [online] GitHub. Available at: https://github.com/rstudio/keras/blob/master/R/initializers.R [Accessed 2 Jan. 2019].

[7] Herf, M. (2000). stereopsis: know your FPU. [online] Stereopsis.com. Available at: http://stereopsis.com/FPU.html [Accessed 29 Jan. 2019].

[8] Niederberger, B. (2019). Set Process Priority In Windows « Python recipes « ActiveState Code. [online] Code.activestate.com. Available at: https://code.activestate.com/recipes/496767-set-process-priority-in-windows/ [Accessed 2 Jan. 2019].

[9] Shimao (2019). What process controls the CPU affinity of new python processes. [online] Super User. Available at:

https://superuser.com/questions/1273705/what-process-controls-the-cpu-affinity-of-new-python-processes [Accessed 2 Jan. 2019].

[10] Alvas (2019). How to implement the Softmax function in Python. [online] Stack Overflow. Available at: https://stackoverflow.com/questions/34968722/how-to-implement-the-softmax-function-in-python [Accessed 2 Jan. 2019].

XI. Appendix A

The code has been included for repeatability and to allow other researchers to overcome the numerical instability of task scheduling.

A.1 Original Code specimen [4] Experiment 1

```
import tensorflow as tf

mnist = tf.keras.datasets.mnist (x_train, y_train), (x_test, y_test) =
mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0

model = tf.keras.models.Sequential ([
  tf.keras.layers.Flatten(),
  tf.keras.layers.Dense(512, activation=tf.nn.relu),
  tf.keras.layers.Dropout(0.2),
  tf.keras.layers.Dense(10, activation=tf.nn.softmax) ])

model.compile(optimizer='adam', loss='sparse_categorical_crossentropy',
metrics=['accuracy'])

model.fit(x_train, y_train, epochs=5)
model.evaluate(x_test, y_test)
```

A.2 Random Number Seeding code change [5] Experiment 2

```
from numpy.random import seed
from tensorflow import set_random_seed
seed(1)
set_random_seed(2)
...
model.fit(x_train, y_train, epochs=1 , shuffle=False)
```

A.3 Modification for Init code insertion Experiments 2-20

```
model = tf.keras.models.Sequential ([
      tf.keras.layers.Flatten(),
      tf.keras.layers.Dense(512,
kernel_initializer=tf.constant_initializer (initval1), bias_initializer =
'zeros', activation = tf.nn.relu),
      tf.keras.layers.Dropout(0.2),
      tf.keras.layers.Dense(10, kernel_initializer=tf.constant_initializer
(initval2), bias_initializer = 'zeros', activation = tf.nn.softmax) ])
```

A.4 Constant value Initialisation code Experiment 3

```
initval1 = np.ones(28*28*512)
initval2 = np.ones(512*10)
```

A.5 Constant value Initialisation code Experiment 4

```
initval1 = np.zeros(28*28*512)
initval2 = np.zeros(512*10)
```

A.6 Constant value Initialisation code Experiment 5

```
initval1 = np.zeros(28*28*512) + 0.05
initval2 = np.zeros(512*10) + 0.05
```

A.7 Code change for ReLU Experiment 6

```
model = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(),
  tf.keras.layers.Dense(512, kernel_initializer = tf.constant_initializer
(initval1), bias_initializer = 'zeros', activation=tf.nn.relu),
  tf.keras.layers.Dropout(0.2),
  tf.keras.layers.Dense(10, kernel_initializer = tf.constant_initializer
(initval2), bias_initializer = 'zeros', activation=tf.nn.relu) ])
```

A.8 Constant value Initialisation code Experiment 7

```
initval1=np.zeros(28*28*512) - 0.05
initval2=np.zeros(512*10) - 0.05
```

A.9 Ramp Initialisation code Experiment 8

```
initval1=np.arange(0,28*28*512,1) /(28*28*512-1)*0.1-0.05
initval2=np.arange(0,512*10,1)/(512*10-1)*0.1-0.05
```

A.10 Ramp Initialisation code Experiment 9

```
initval1=np.arange(0,28*28*512,1) /(28*28*512-1)*0.1
initval2=np.arange(0,512*10,1)/(512*10-1)*0.1
```

A.11 Ramp Initialisation code Experiment 10

```
initval1=np.arange(0,28*28*512,1) /(28*28*512-1)*0.05
initval2=np.arange(0,512*10,1)/(512*10-1)*0.05
```

A.12 Ramp Initialisation code Experiment 11

```
initval1=np.arange(0,28*28*512,1) /(28*28*512-1)*0.05-0.05
initval2=np.arange(0,512*10,1)/(512*10-1)*0.05-0.05
```

A.13 Sinusoid Initialisation code Experiment 12

```
initval1=np.sin(np.arange(0,28*28*512,1)/(28*28*512-1)*np.pi*2)*0.05
initval2=np.sin(np.arange(0,512*10,1) /(512*10-1)*np.pi*2)*0.05
```

A.14 Sinusoid Initialisation code Experiment 13

```
initval1=np.sin(np.arange(0,28*28*512,1)/(28*28*512-1)*np.pi*2)*0.05
+0.05
initval2=np.sin(np.arange(0,512*10,1)/(512*10-1)*np.pi*2)*0.05+0.05
```

A.15 Sinusoid Initialisation code Experiment 14

```
initval1=np.sin(np.arange(0,28*28*512,1)/(28*28*512-
1)*np.pi*2)*0.025+0.025
initval2=np.sin(np.arange(0,512*10,1)/(512*10-
1)*np.pi*2)*0.025+0.025
```

A.16 Sinusoid Initialisation code Experiment 15

```
initval1=np.sin(np.arange(0,28*28*512,1)/(28*28*512-1)*np.pi*2)*
0.025-0.025
initval2=np.sin(np.arange(0,512*10,1)/(512*10-1)*np.pi*2)*0.025-0.025
```

A.17 Real-Time single Core Affinity code [8] [9] Experiments 17 - 20

```
import psutil
psutil.Process().cpu_affinity([0,0,0,0])

def setpriority(pid=None,priority=1):
   import win32api,win32process,win32con
   priorityclasses = [win32process.IDLE_PRIORITY_CLASS,
           win32process.BELOW_NORMAL_PRIORITY_CLASS,
           win32process.NORMAL_PRIORITY_CLASS,
           win32process.ABOVE_NORMAL_PRIORITY_CLASS,
           win32process.HIGH_PRIORITY_CLASS,
           win32process.REALTIME_PRIORITY_CLASS]
   if pid == None:
     pid = win32api.GetCurrentProcessId()
   handle = win32api.OpenProcess(
   win32con.PROCESS_ALL_ACCESS, True, pid)
   win32process.SetPriorityClass(handle, priorityclasses[priority])

setpriority(None, priority=5)
model.fit(x_train, y_train, epochs=5, verbose=1, shuffle=True)
setpriority(None,priority=2)

setpriority(None, priority=5)
model.evaluate(x_test, y_test, verbose=1)
setpriority(None,priority=2)
```

A.18 Modified SoftMax function change [10] Experiment 19

```
from keras.layers import Activation
from keras import backend as K
from keras.utils.generic_utils import get_custom_objects

def custom_activation(x):
   exps = K.exp(x - K.max(x))
   return exps / K.sum(exps)

get_custom_objects().update({'custom_activation':
Activation(custom_activation)})
```