Programming Assignment 1: Classification

Joe Johnson CIS 472/572 Dr. Humphrey Shi

Foreword

I had many troubles with specific implementations, but found the tutorials provided to be quite helpful. That being said, I still struggled to figure out the specifics requested of this homework assignment; as I am unsure as to the formality required, the detail needed for each data point found, the number of data points required as well as some other specifics. I hope that the details and reports here are sufficient as I spent dozens of hours just waiting for tests to get done to make individual tweaks (before upgrading to Google Colab Pro, as there was installation issues with my personal rig).

1. Introduction

The following results are all trial runs of different methods, networks, and optimizations of the CIFAR10 dataset. All code is given in the included ipynb file, written in python, and making use of the PyTorch library. Testing done was using Google Colab Pro, so cuda implementation was tested on an Nvidia Tesla P100 (16GB version). A typical run for Part 1 can be seen in Figure 0. For the base case, a simple convolutional neural network (CNN) was used with a learning rate of 0.001 and a momentum of 0.9. Each training session was done 5 times and took advantage of 5 epochs for the comparative results; this average applied to the base case can be seen in Figure 1. More tests were done to see the cause/effect relationship between the manipulation of certain aspects of the networks, but these do not apply to any hard numbers provided going forward.

1.1. Learning Rate

The initial learning rate provided by the tutorial (0.001) seemed to be relatively consistent when compared to other learning rates tested [1]. The following numbers are the results I received when testing learning rates of 0.1, 0.01, 0.0001 and 0.00001 to compare to the base case. Results can be seen in Figure 2. On top of this decrease in loss, Figure 0 also shows a significant drop in the accuracy of the non-base rates. Run-time was virtually unaffected by these changes.

1.2. Momentum

The initial rate provided by the tutorial (0.9) tested quite well comparatively [1]. For the results to follow I tested with everything the same as the base case except the manipulation of the momentum at 0.3, 0.5, 0.95 and 0.99 for comparison. The loss on the base momentum was slightly better than most, and much better than .99 momentum. Results can be seen in Figure 3. On top of this decrease in loss, Figure 0 also shows a slightly better accuracy with the base case (once again, significantly better than .99).Run-time was virtually unaffected by these changes.

1.3. Architectures

For architectures there is many more factors that are hard to keep consistent throughout the multiple networks tested. I tried to control everything I could, but some variables might have been overlooked. The architectures tested were pretrained and were provided by PyTorch [2]. The architectures compared to the base CNN were Resnet18, Alexnet, VGG11_bn, Squeezenet, DenseNet and Inception v3. When referring to Figure 0, we can see that the run time for the simple CNN was significantly faster than the other architectures test (especially DenseNet and Inception v3). It also held up decently against its slower relatives from accuracy standpoint, despite being the second lowest in most test runs. The base case by far had the best loss values, but this did not seem to affect accuracy.

1.4. Linear Classifier

I tested a Linear Classifier both with and without weights. I found that the unweighted version had lower loss, at least for my classifier. This was surprising to me, and even more surprising was the higher accuracy I found in the unweighted version. The results for loss can be found in Figure 4 and the accuracy can be found in Figure 5. Run-time was unaffected.

1.5. Nearest Neighbor

I failed my attempt at constructing this. It even got to the point where I installed PyKeOps [2] to try and get a working version. I believe I was close, but after 24 hours straight of not figuring it out I gave up. I left my failed code in the included ipynb file.

1.6. Conclusion

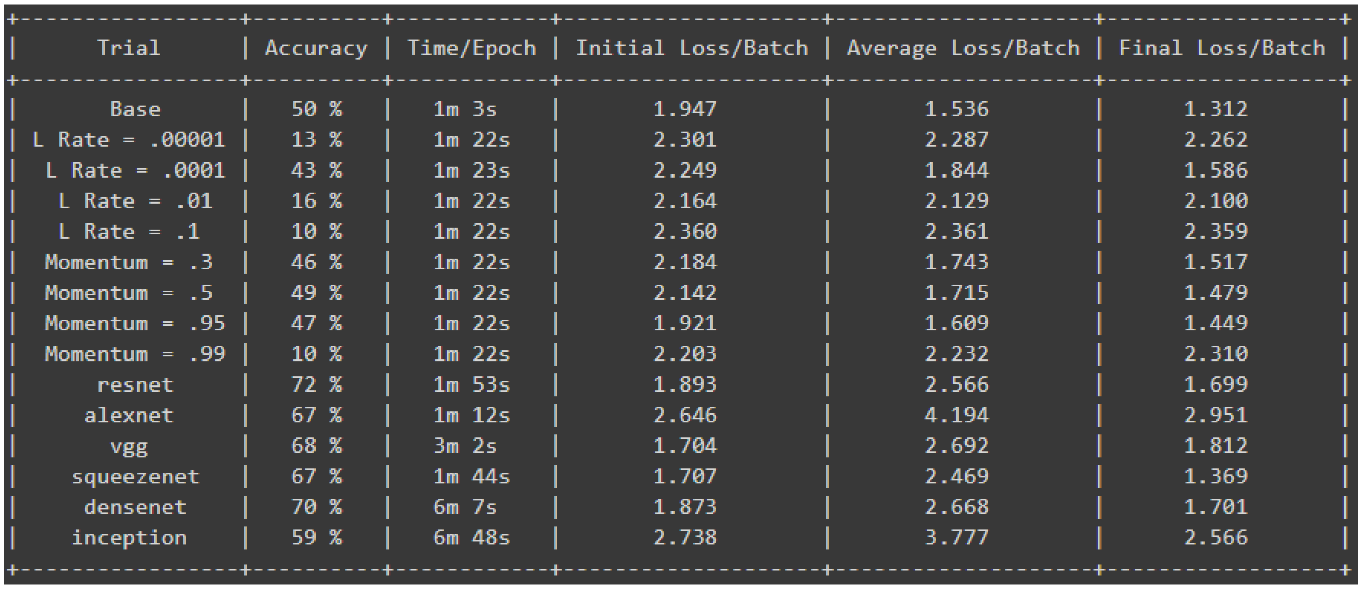
Although I did not finish all aspects of this assignment, I learned a lot about the implementation of machine learning in a practical sense (as opposed to the more theoretical or algorithmic approach usually presented in class). I found that the provided tutorial’s base cases tended to be the best overall. Also, I found that each of the differing architectures seem to have their own strengths. Some were faster, some were more accurate, and I think some just could not show their true potential through my limited testing.

References

[1] https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html#sphx-glr-beginner-blitz-cifar10-tutorial-py

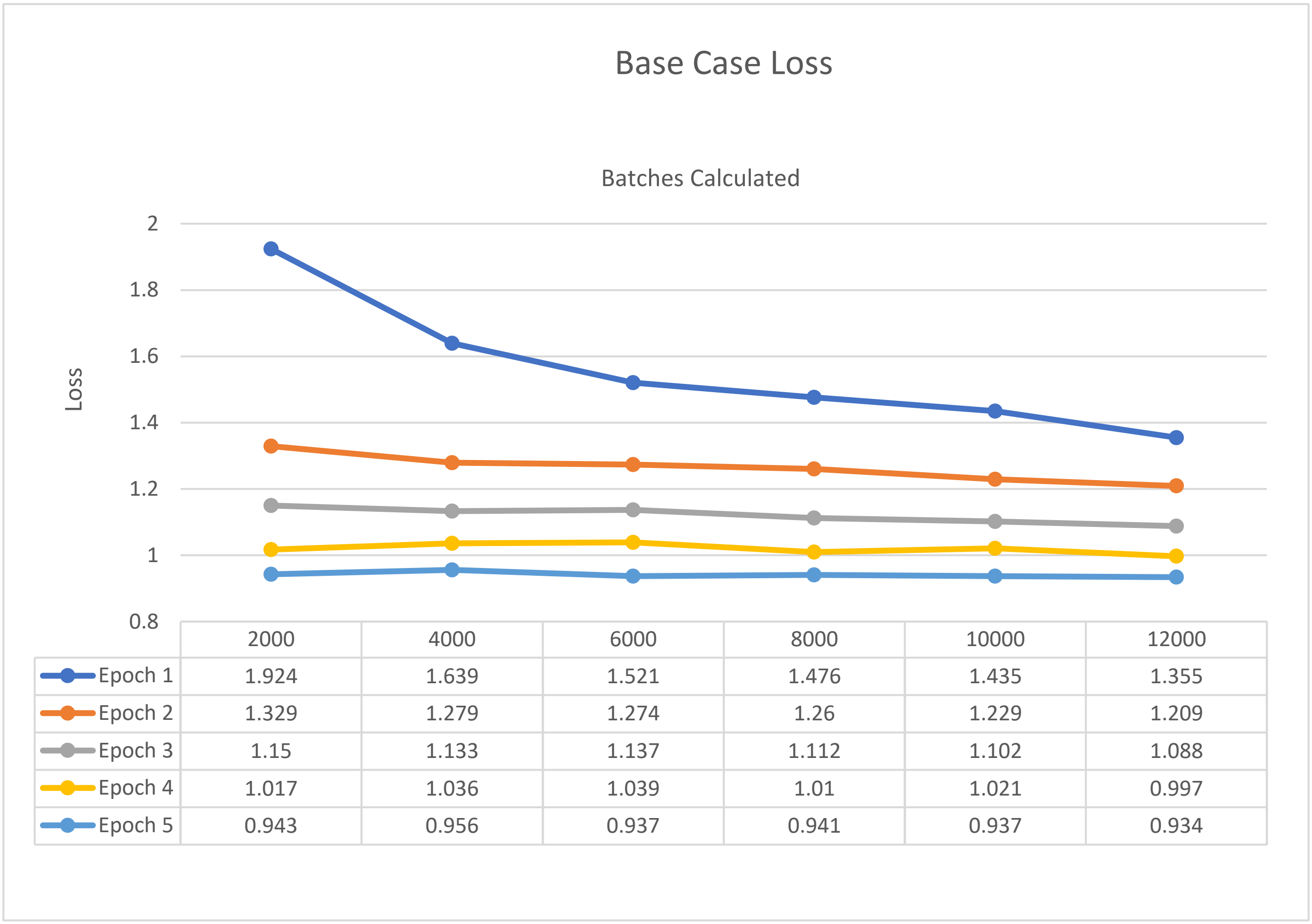
[2] https://pytorch.org/tutorials/beginner/finetuning_torchvision_models_tutorial.html

[3] https://pypi.org/project/pykeops



```
+-----------------+----------+------------+--------------------+--------------------+-------------------+
|      Trial      | Accuracy | Time/Epoch | Initial Loss/Batch | Average Loss/Batch | Final Loss/Batch  |
+-----------------+----------+------------+--------------------+--------------------+-------------------+
|       Base      |   50 %   |   1m 3s    |       1.947        |       1.536        |       1.312       |
| L Rate = .00001 |   13 %   |   1m 22s   |       2.301        |       2.287        |       2.262       |
|  L Rate = .0001 |   43 %   |   1m 23s   |       2.249        |       1.844        |       1.586       |
|   L Rate = .01  |   16 %   |   1m 22s   |       2.164        |       2.129        |       2.100       |
|   L Rate = .1   |   10 %   |   1m 22s   |       2.360        |       2.361        |       2.359       |
|  Momentum = .3  |   46 %   |   1m 22s   |       2.184        |       1.743        |       1.517       |
|  Momentum = .5  |   49 %   |   1m 22s   |       2.142        |       1.715        |       1.479       |
|  Momentum = .95 |   47 %   |   1m 22s   |       1.921        |       1.609        |       1.449       |
|  Momentum = .99 |   10 %   |   1m 22s   |       2.203        |       2.232        |       2.310       |
|      resnet     |   72 %   |   1m 53s   |       1.893        |       2.566        |       1.699       |
|     alexnet     |   67 %   |   1m 12s   |       2.646        |       4.194        |       2.951       |
|       vgg       |   68 %   |   3m 2s    |       1.704        |       2.692        |       1.812       |
|    squeezenet   |   67 %   |   1m 44s   |       1.707        |       2.469        |       1.369       |
|     densenet    |   70 %   |   6m 7s    |       1.873        |       2.668        |       1.701       |
|    inception    |   59 %   |   6m 48s   |       2.738        |       3.777        |       2.566       |
+-----------------+----------+------------+--------------------+--------------------+-------------------+
```

Figure 0: A typical run of 5 epochs (slight error in average loss calculation of other architectures, rest is confirmed. No time to fix calculation and rerun)



	2000	4000	6000	8000	10000	12000
Epoch 1	1.924	1.639	1.521	1.476	1.435	1.355
Epoch 2	1.329	1.279	1.274	1.26	1.229	1.209
Epoch 3	1.15	1.133	1.137	1.112	1.102	1.088
Epoch 4	1.017	1.036	1.039	1.01	1.021	0.997
Epoch 5	0.943	0.956	0.937	0.941	0.937	0.934

Figure 1: Results averaging 5 runs of 5 epochs.

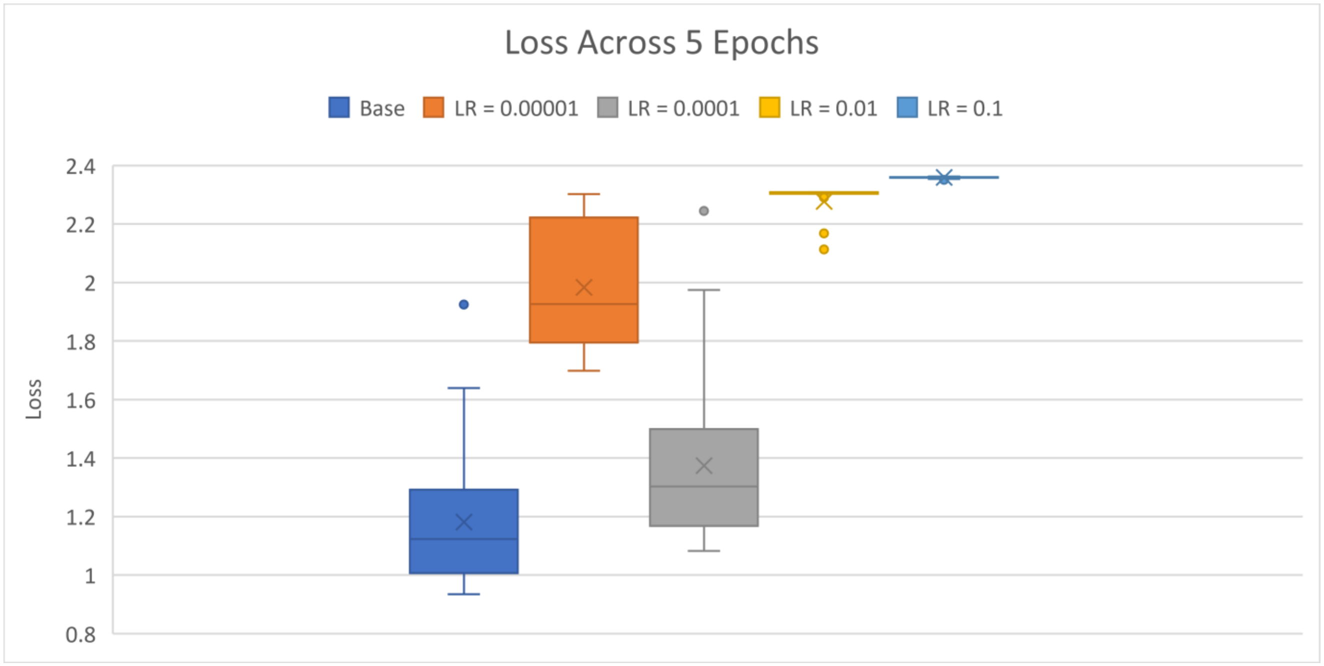


Figure 2: Results averaging 5 runs of 5 epochs.

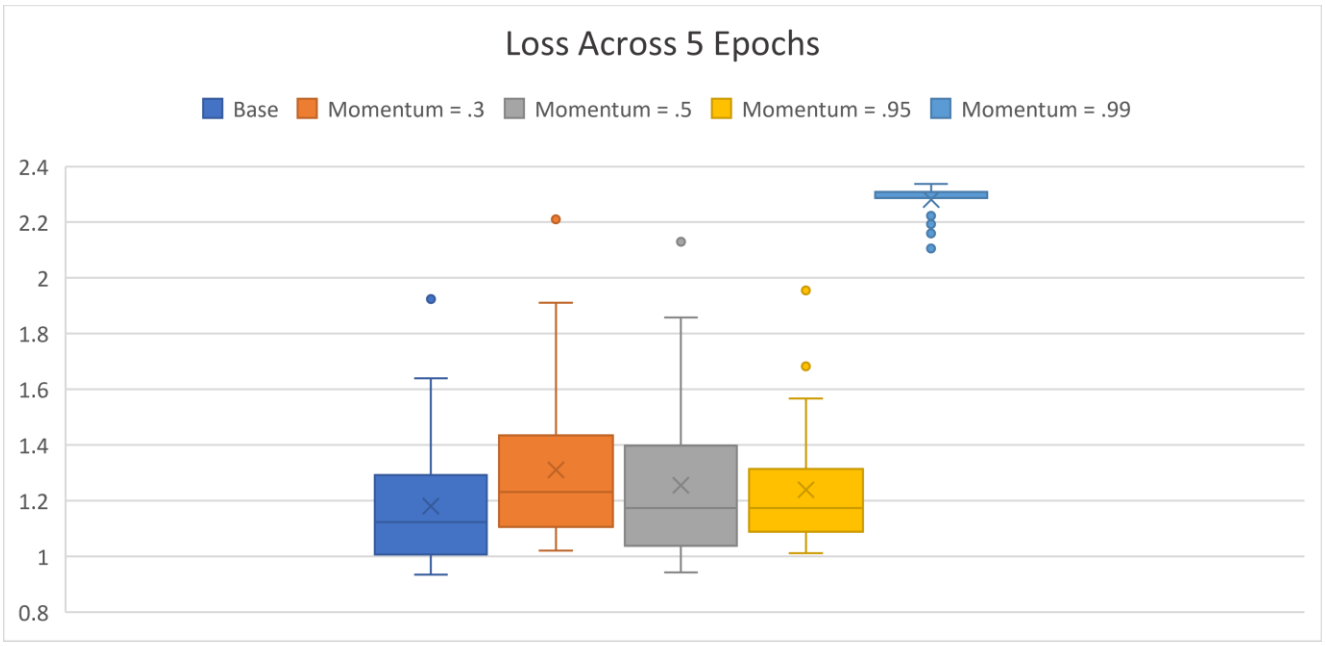
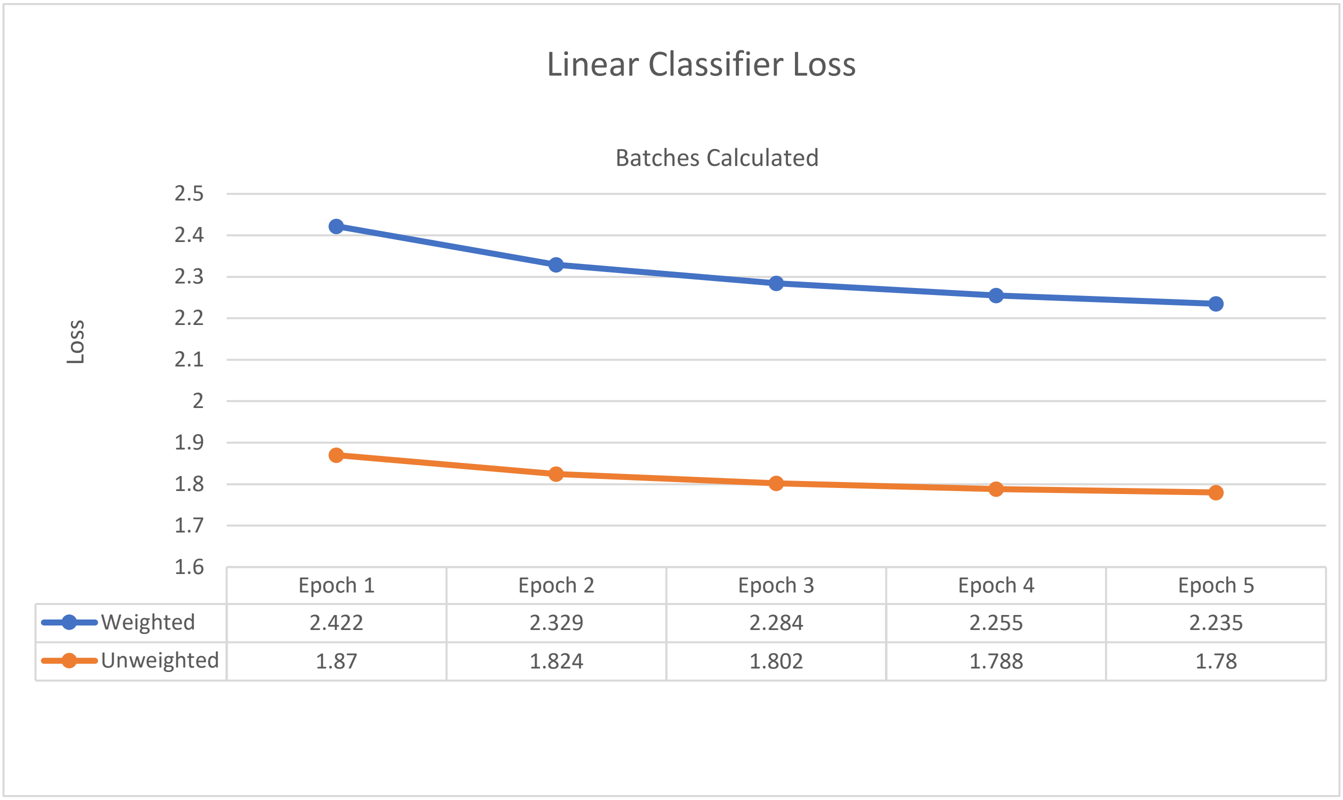
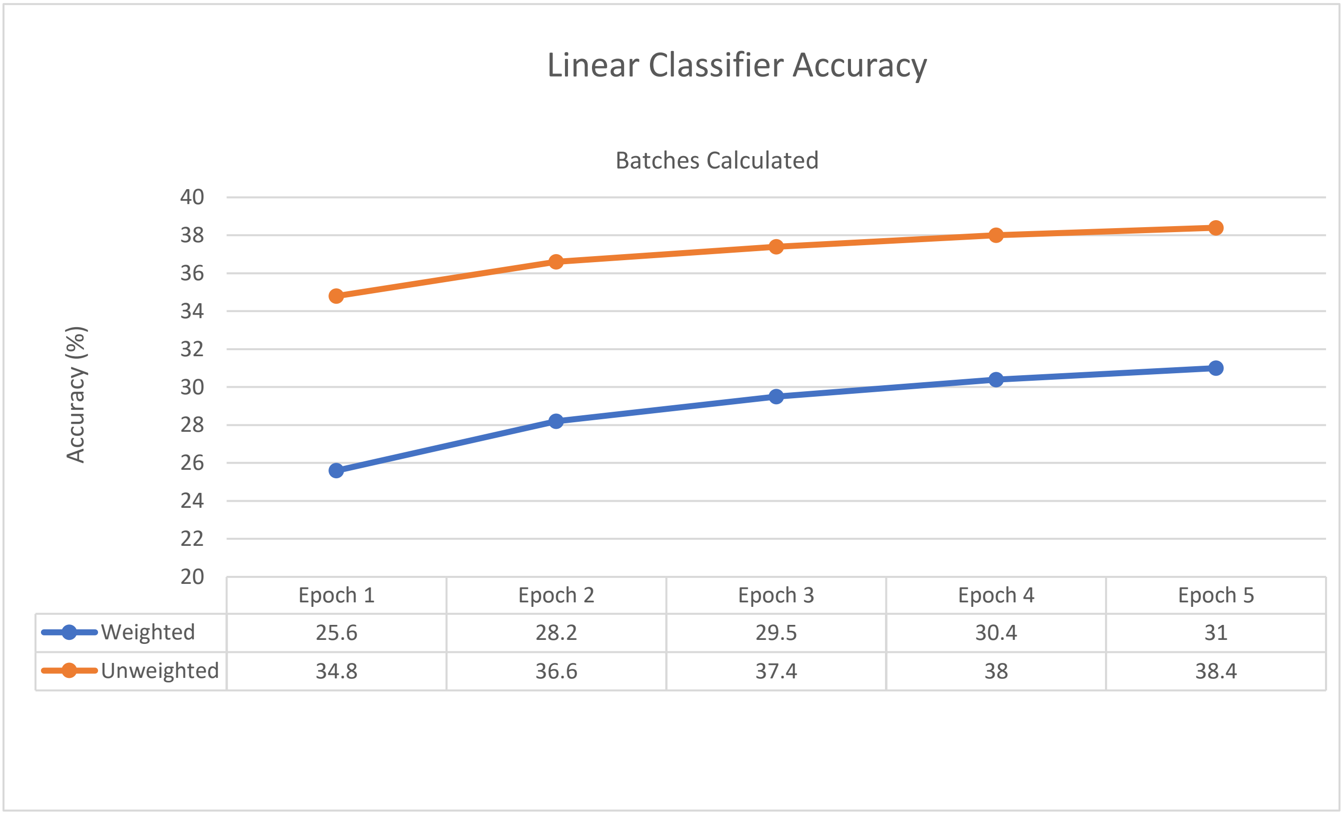


Figure 3: Results averaging 5 runs of 5 epochs.



	Epoch 1	Epoch 2	Epoch 3	Epoch 4	Epoch 5
Weighted	2.422	2.329	2.284	2.255	2.235
Unweighted	1.87	1.824	1.802	1.788	1.78

Figure 4: Results averaging 5 runs of 5 epochs.



	Epoch 1	Epoch 2	Epoch 3	Epoch 4	Epoch 5
Weighted	25.6	28.2	29.5	30.4	31
Unweighted	34.8	36.6	37.4	38	38.4

Figure 5: Results averaging 5 runs of 5 epochs.