2023

COMP3010 Assignment

20175535

KEVIN MESQUITA

1 CONTENTS

2 INFORMATION REGARDING ASSIGNMENT

There are a few things about my assignment that I would want to take note of.

The first thing is that I decided to do all the computation for training the model on my computer, I did this since Google Collab was not allowing me to use the T4 GPU for long enough to test and optimize different parameters of my model.

The main specs of my computer if necessary are:

- i7 9700k
- RTX 2060 6gb
- 16gb ram

The gpu is the main resource being used, to speed up the training of the model, I put all the computation on the CUDA cores of the GPU. This means that all the timings and testing is done with respect to my GPU, I'm not sure how much faster/slower the Google Collab times would be, and also against the v100 GPU used in myrtle.ai which would be at least twice as fast.

The GPU only had 6gb of VRAM, which meant that I couldn't scale the first image above 96x96 (the initial size from the template) for section 2. And for section 3 since CIFAR had a image size of 32x32 with 3 channels, it was even more restricted, a lot of the resent model relies on a larger image size so by the time it gets to the last layer, the convolution and pooling layers don't really do anything, which would hinder the overall accuracy by a great deal. Either way, I tried to optimize with a lower image size (which did have a benefit of giving faster training times).

Secondly, the D2L.ai resnet model didn't meet my requirement so I decided to implement a template model from a different website. [1] A tutorial by Nouman for implementing Resnet in pytorch. I did however change the parameters as much as I could to copy that off the resent model on d2l.

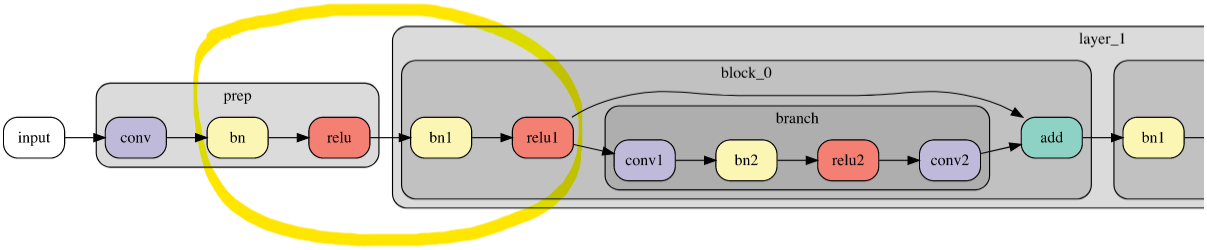
Finally, for the data collected from looking at different parameters, the graphs are in the document, but the raw numbers would be in the excel documents for section 2 and 3 each with the excel pages in the order from start to finish reflecting the steps I discuss I took in this document to optimise the model.

3 SECTION 1: NOTES

Notes from https://myrtle.ai/learn/how-to-train-your-resnet/ [9].

3.1 BASELINE

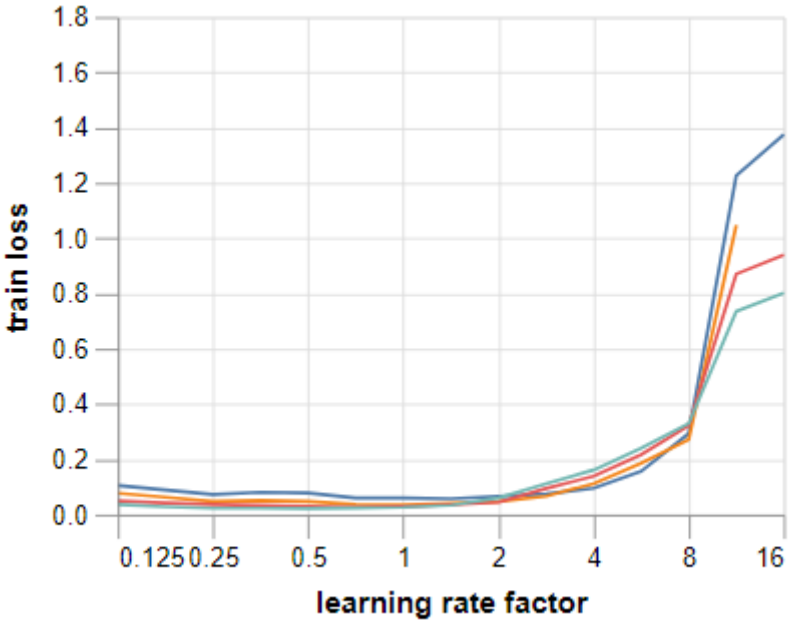
- Repeat of certain functions such as batch normalization and relu don't need to be repeated, this just adds time.



- For pytorch, for each training epoch, a new iteration of the data loader is initialized. This would add to extra processing before every training method adding time per epoch.
- Operations and processes that don't need to be done during and before each training cycle should be done at the very beginning of the training in one big batch.

3.2 BATCHES

- Faster training speed can be made with higher learning rates.
- To decrease the amount of time to train the model, you can increase the batch size, this means that there are less iterations needed to go through speeding up the process.
- For stochastic gradient descent training with the weights changes being held until the end of the batch training, the learning rate would need to be increased to make up for these larger steps to the weight rather than smaller more frequent adjustment with low learning rate.
- Effects to the higher order functions in the linear separator don't happen if the learning rate is too low.
- Catastrophic Loss: When training with batches and applying SGD to weights at the end of each batch, it could act like multiple different training iterations. With each new batch, everything that was learnt from the previous batches is slowly forgotten. This can occur with high enough learning rates.



- The optimum learning rate is just before the loss starts to drastically increase. (and have catastrophic loss)

3.3 Regularization:

- Integer precision can provide a large improvement to training speed. For example single precision numbers are faster to compute than half precision.
- Generally, the code would need to optimized for the GPU on how it processes data most efficiently. So this can include data types, data ordering.
- Various regularization techniques would need to be explored for specific models, for CIFAR10 specifically, cutout regularization works well (going from 0/5 runs with 94% accuracy to 5/5)

3.4 Architecture:

- To help train a residual network, by removing the long branch of the network and looking at the shortest path and improving it, can help the overall network.
- Kernel of 1x1 with stride of 2 by itself would discard information by skipping over pixels reducing accuracy and learning of model
- Pooling is better than stride convolutions, gives a bit better accuracy. However it takes longer to train
- At the very end of the network, max pooling is better than average pooling significantly.
- Brute Force Architecture search, trying different combinations of layer structure, max pooling, convolution and activation functions.

3.5 Hyper-Parameter Tuning:

- Tuning hyper parameters has a small effect on the overall accuracy of a model
- A method used to optimize the hyper parameters learning rate, momentum, weight decay is to first find a optimal learning rate, from there momentum then weight decay then finally go

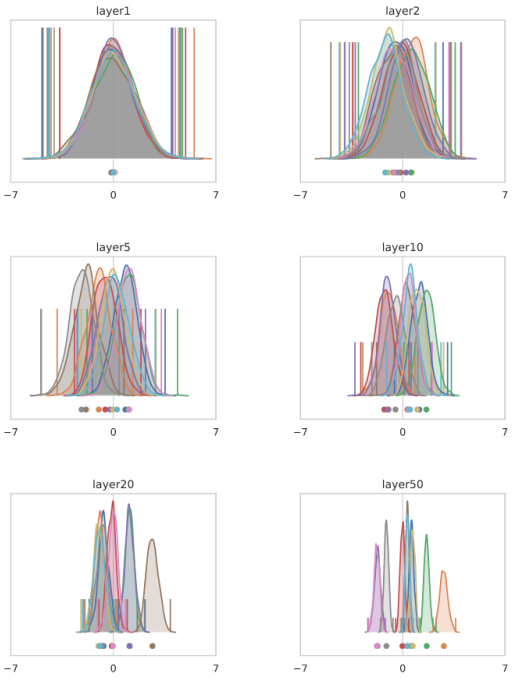
back to learning rate and momentum. When optimizing learning rate and momentum, keep the equation ($\frac{learning\ rate * weight\ decay}{1-momentum}$) value the same by adjusting the learning rate.

3.6 Weight Decay

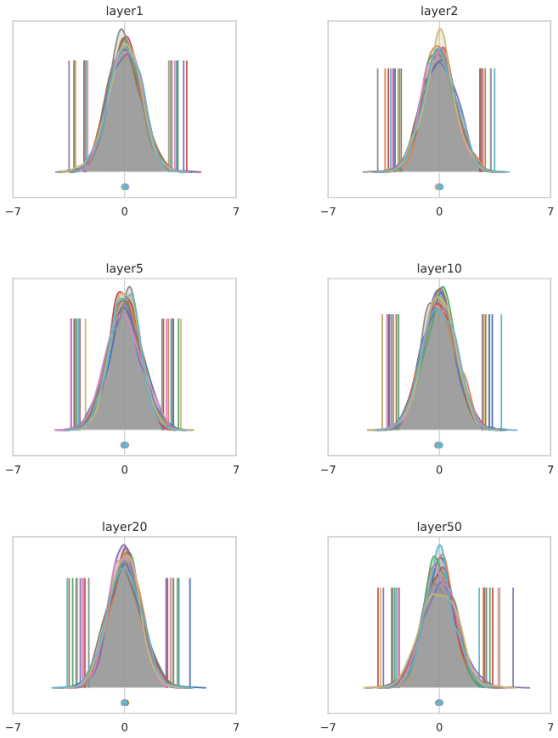
- For smaller weights, learning/growth of the weights has the biggest impact and vice versa for large weights and decay.
- The further layers you go into a network the less the decay/growth on the weights of those layers occur.
- With stochastic gradient descent, after large batches it is variant under scaling. Scaling the weights by a factor r, then gradients g scale by r^-1 rather than the r required for invariance.
- Can use a different optimizer called LARS instead of SGD

3.7 Batch Norm

- Batch normalization acts by removing the mean and normalizing the standard deviation of a channel of activations. i.e scaling the data from a scale of x to y to 0 to 1. There can be variables added to change other scales (using beta and gamma variables for a linear translation and scaling)
- Good because:
    - Stabilizes optimization allowing for higher learning rates and faster training.
    - Injects noise (generalization)
    - Reduces sensitivity to weight initialization
    - Interacts with weight decay to control the learning rate dynamics
- Bad:
    - Slow
    - Different at training and test time (therefore fragile)
    - Ineffective for small batches and various layer types
    - Has multiple interacting effects which are hard to separate



- Without batch normalization, the mean and deviation of each channel changes, by the end, they take individual values rather than a spread with a mean at 0



- Without Relu activation function and normalization, there isn't as much of change
- Relu seems to be the problem, because its output is always non-zero and returns only positive values. Putting that through the weights of each layer, the change slowly gets more and more obvious even though the weights can output negative values it starts to compound.
- Basically without normalization, the network computes a constant function.
- There are certain eigvalues and eigenvectors for SGD depending on the model which cause the output values to fall into certain outlying values which increase loss. (This is called Hessian values) cause instability in SGD

3.8 Bag of Tricks

- Time wasted on preprocessing the data on the cpu, can speeden this up by transferring the data onto the gpu processing it there and then transferring it back.
- Using CELU activation instead of RELU, CELU is just a smoothed out RELU
- Ghost batch norm, normalizing the 512 size batch in smaller batches, since batch normalization is done faster in smaller batches.
- Frozen batch normalization, fixing the batch normalization scale to a constant.
- Using a 'whitening version' of batch normalization to get rid of covariance between channels and pixels
- To help a network classify augmented data (flipped images), present both the input and flipped image and come to a consensus by averaging network outputs for the two versions, to get invariance. Can do this by splitting the network into two branches, with each branch seeing one of the images, merging the branches at the end

4 Section 2:

4.1 Preface:

The model being used is not the d2l template however it is a resnet implementation, by Nourman [1] with parts taken from [4]. I tried to make the model as close to the d2l implementation as possible before starting the optimisations, it is also a resnet18.

The main values in the base case are:

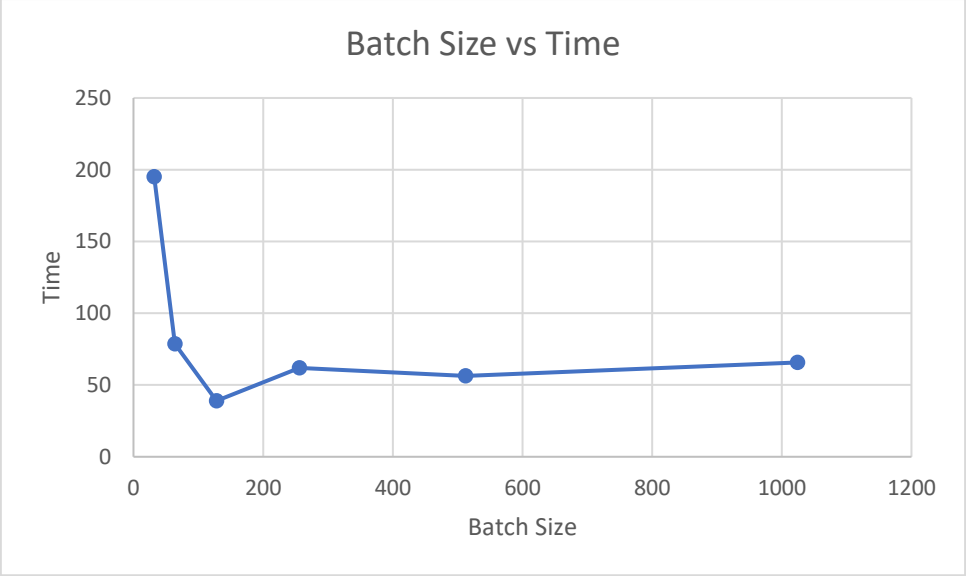
1. Learning rate: 0.1
2. Batch Size: 128
3. Number of worker: 0
4. Epochs: 10

To make things a bit simpler for comparing training times, the accuracy at the end of the 10 epochs will be compared along with the average time to train per epoch.

4.2 Batch Size:

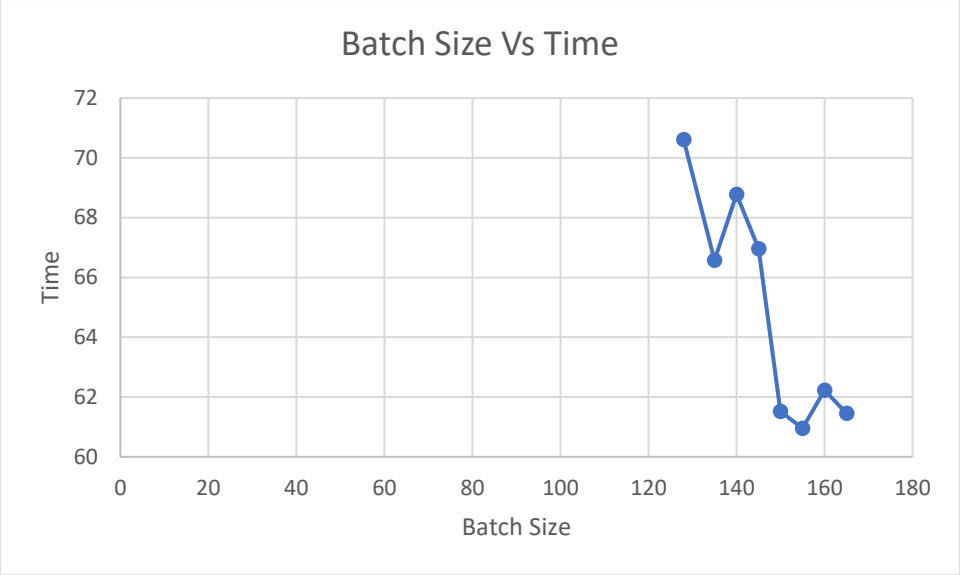
For the batch size, just like in the myrtle ai [9], increasing the batch size should decrease the training time for the model. The first run consisted of testing 128 (the current batch size), 256, 512 and 1024.

The average time to train per epoch was graphed against the batch size.



As can be seen in the figure above, the current batch size of 128 is already close to minima. This would require finer search around it to find a better value.

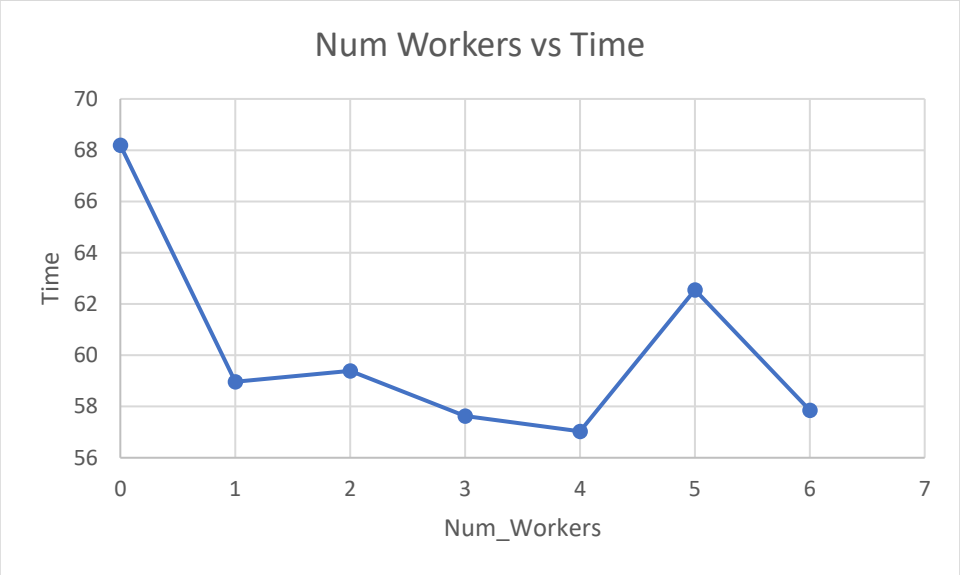
Batch sizes 128, 135, 140, 145, 150, 155, 160, 165 were tested. Note these results are not added to the previous figure as due to training the model on my computer, and being on a different day, the training times were different. *Hindsight I should have used larger steps and also multiples of 32 for batch size (I do this later)*



The batch size of 155 looks to be the best although there isn't much difference between the other larger batch sizes. I could try test the higher batch sizes but it would take to long (hours) to go through them. Therefore, the new batch size will be 155.

4.3 NUMBER OF WORKERS:

The data loader is an object which feeds data to the model for training. The size of the batches of data can be set through this object. What will be looked at is the time change from the base case for increasing the number from the base case 0.

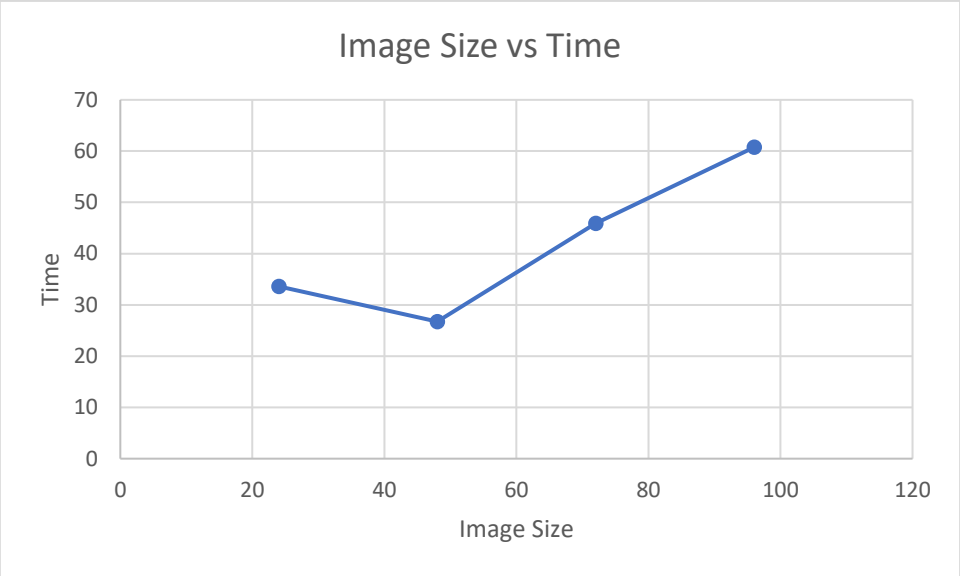


There is a noticeable decrease in time per epoch from 0 to 1 but after that the training times increase slows down. The increase at 5 is a bit of an outlier, with 6 being around an expected value as well. I feel like the difference in training times between 1 and 6 is just variance, it looks a bit more obvious when looking at the times, some of the lower timing have outlier epochs with a few epochs having greater than 60 second training times. Furthermore, when comparing the outputs to other previous tests, it can be seen that the test cases that overlap between tests output different results.

Regardless, the new set num_workers parameter is 2.

4.4 Changing Dataset Image Size:

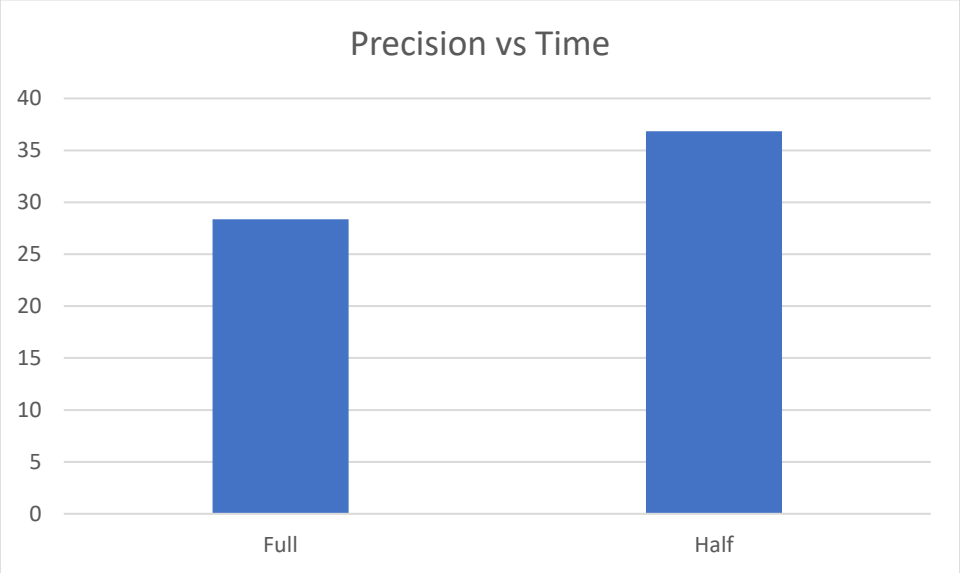
The standard image size for FashionMNIST data set is 24x24. The model on the other hand transforms the data before training into a 96x96 set. It is obvious that with a smaller input image size, there would be less processing needed as there are less pixels to process. Note that all scaled images are ant aliased beforehand (this does not effect time). Changing the image size can be done by a Pytorch transform when loading the image.



Testing image sizes of 24x24, 48x48, 72x72 and 96x96 (since they are multiple of each other. The 48x48 images size performs the best, I was expecting the time to increase linearly with image size, not sure why 48x48 is fastest. Even if 24x24 was faster, the accuracy results were very unstable, going from high eighties to mid fifties in epochs one after the other.

4.5 Changing Tensor Precision:

By default, the precision of a pytorch model is at full precision [10]. Half precision floating point can help speed this up by both reducing the amount of memory and the time per calculation. The method I used to implement this was using the pytorch auto grad scaler and this code for help [7].

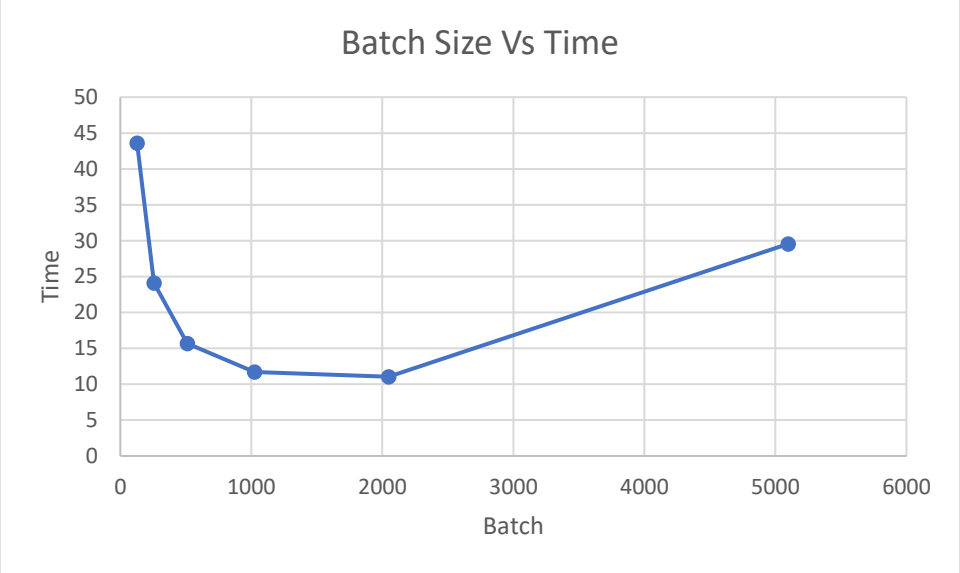


It was found that with the half precision, there was an increase in training times. There were two conflicting predictions I had, lower precision would mean the less space required to store which would mean the speed shouldn't change for a batch size of 155, on the other hand, lower floats means easier to compute, so faster training time. I was not expecting it to be slower, it might be from converting the full precision to half precision.

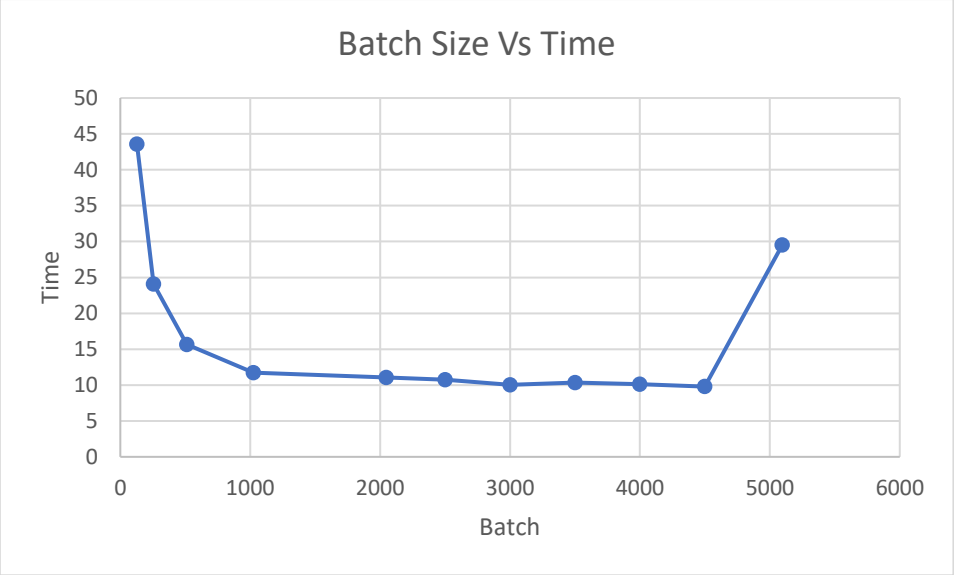
As a test, I tried to train on a larger batch size, 1024. The results were way better, with an 11 second average. This is where I realised that I made a mistake with the optimisation of batch size. I should have optimised batch size along with converting to half precision.

4.6 BATCH SIZE V2:

With the half precision, I redid the testing for batch size, from 128, 256, 512, 1024 and 2056 batch sizes, I can now do 2056 sizes since the memory usage on my GPU is halved therefore even with 6gb on my rtx 2060 it can still fit it all in. All the previous parameters are the same.



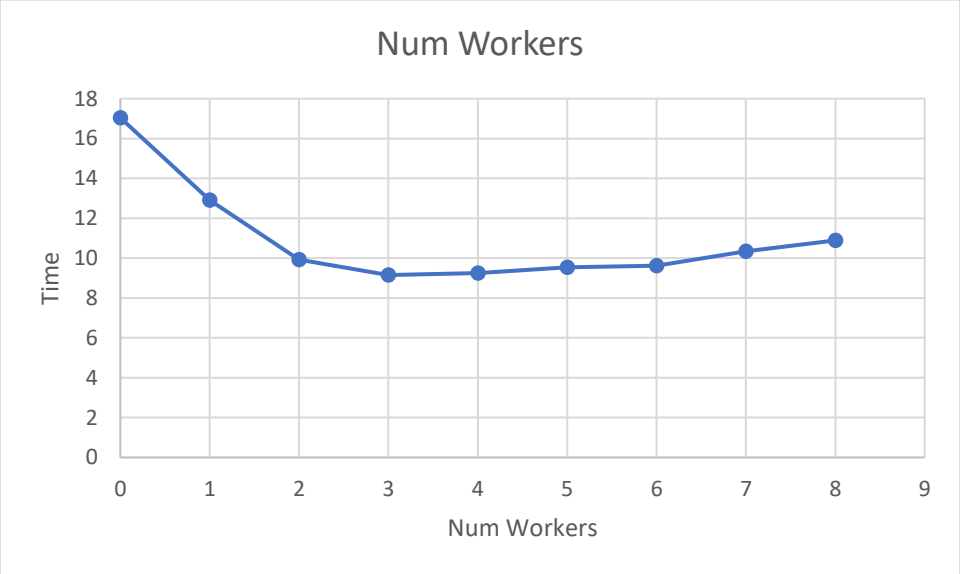
From this graph you can see that there is gap between 2048 and 5096, which needs to be looked at, I feel like the training speed could be better by fractions of a second, the graph looks to take a 1/x shape with an asymptote around the sub 10 second mark. This trend looks to stop somewhere before the 5096-batch size.



This is the highest resolution I am willing to go, the 4500 batch size looks to have the best by a few fractions of a second at 9.79 seconds per epoch on average.

4.7 Num Workers v2:

With a change in the batch size, redoing the number of workers in the data loader would need to be looked at again. Checking a range of 0 to 8:



The best value is 3 but there isn't much of a gain between its neighbouring values.

4.8 Architecture:

From section 3 architecture, I learnt that I could decrease the time by reducing the output channels on the residual layers.

I don't have time to test all cases, but since I have accuracy to spare, I will reduce the channels by half. This means the output channels for the layers are 32, 64, 128, 256 respectively.

Regular Resnet18 output channels:

Num Workers: 3 LR: 0.1
Epoch, Loss, Time, val_acc
1 , 0.6910666227340698 , 11.803538700012723 , 74.38
2 , 0.536470890045166 , 10.086293299973477 , 79.45
3 , 0.4947056472301483 , 9.595598700019764 , 81.01
4 , 0.3900723159313202 , 9.639502499980154 , 84.39999999999999
5 , 0.31376174092292786 , 10.077725199982524 , 85.76
6 , 0.3186308741569519 , 10.113291300018318 , 85.2
7 , 0.25556641817092896 , 9.985268300020834 , 87.33
8 , 0.23804450035095215 , 10.048375200014561 , 88.32
9 , 0.20972073078155518 , 9.87097079999512 , 89.58

Half Output Channels:

Num Workers: 3 LR: 0.1
Epoch, Loss, Time, val_acc
1 , 0.7405751943588257 , 8.531301799986977 , 73.42999999999999
2 , 0.4162430465221405 , 7.911559400003171 , 82.87
3 , 0.37104809284210205 , 7.915440799988573 , 84.78
4 , 0.3938436210155487 , 8.018956500018248 , 85.35000000000001
5 , 0.3397643566131592 , 7.939368500025012 , 84.83000000000001
6 , 0.2798999845981598 , 7.979829399992013 , 87.16000000000001
7 , 0.26615262031555176 , 8.02372060000198 , 87.83999999999999
8 , 0.24686041474342346 , 7.987532500002999 , 87.06
9 , 0.2520042359828949 , 7.888644099992234 , 88.25
10 , 0.22613894939422607 , 7.996164200012572 , 88.64999999999999

With this, you can see that the half output channels model gets to 80% accuracy in 2 epochs (16.44286 seconds) compared to the base (31.48543 base)

5 Section 3:

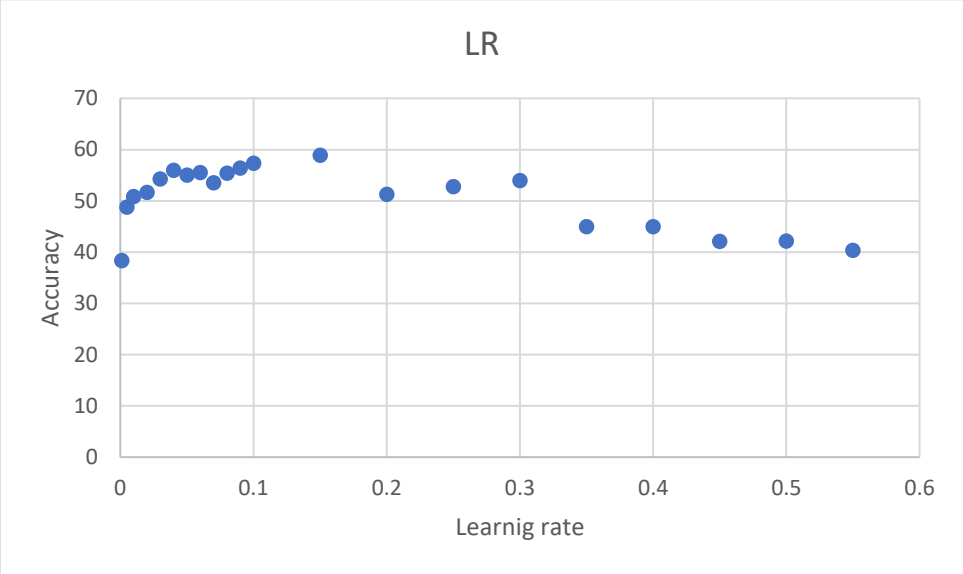
5.1 Preface

Before I start this section, I came across an error converting the model over to take in CIFAR10, for some reason when I changed the number of input channels, the data loader decided it didn't like having more than 2 workers, therefore I had to reduce it down from 3 to 2.

Another thing is that all optimisations are being carried over from section 1.

5.2 Learning Rate Optimisation V1

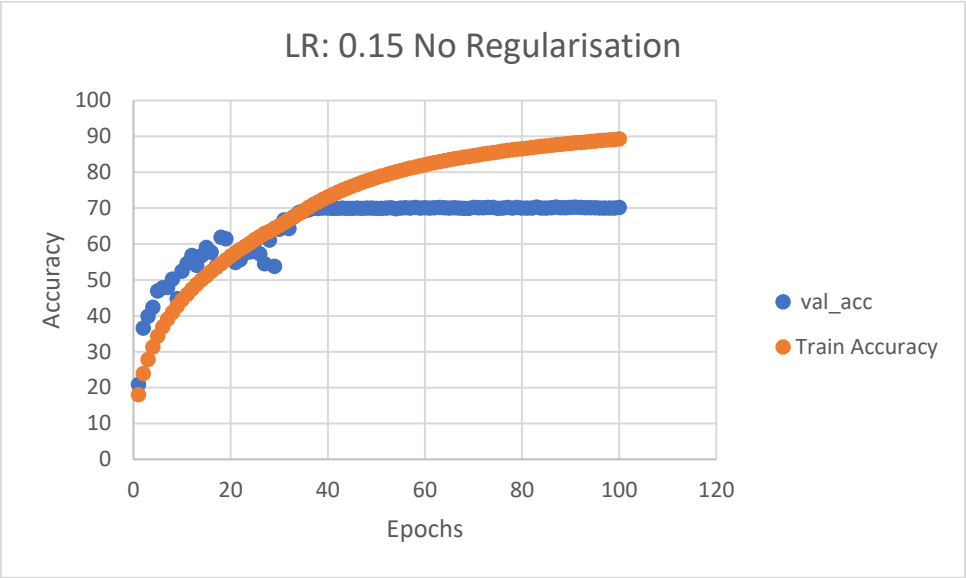
For this section, I will start of with the same parameters as the previous model. To start off, I will try varying the learning rate, with an epoch of 15.



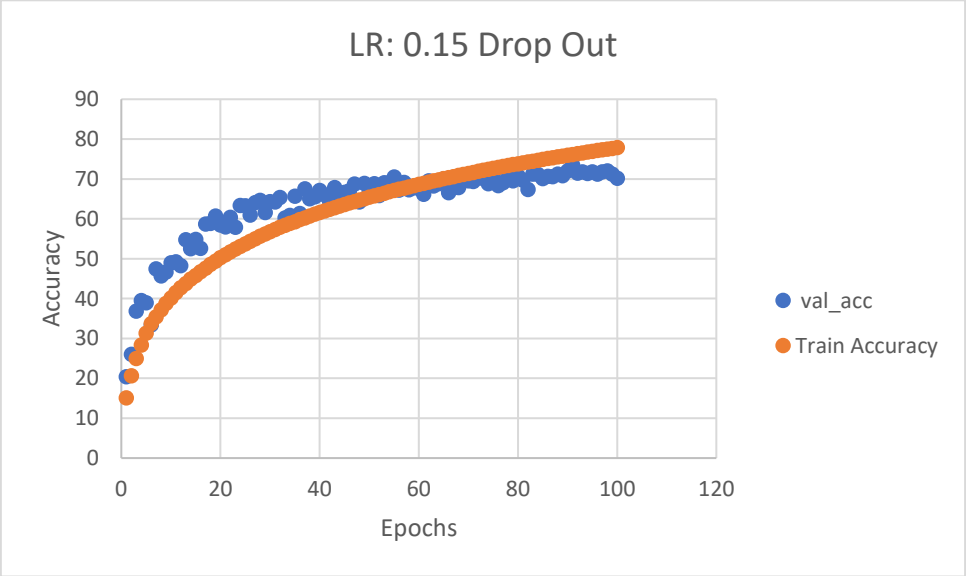
The validation accuracy looks to be the best for the 0.15 learning rate.

5.3 Drop Out

15 epochs didn't look like enough to get a good story on how the model was learning the data, so I decided to add the training accuracy calculation to the outputs of my trainer and ran the model over 100 epochs.



With this I saw that the data was over fitted to the training set, so I added drop out and trained for another 100 epochs. (The drop out value I chose was decided on by checking different values, can see it in the excel page, "Dropout" of section 3 spreadsheet.

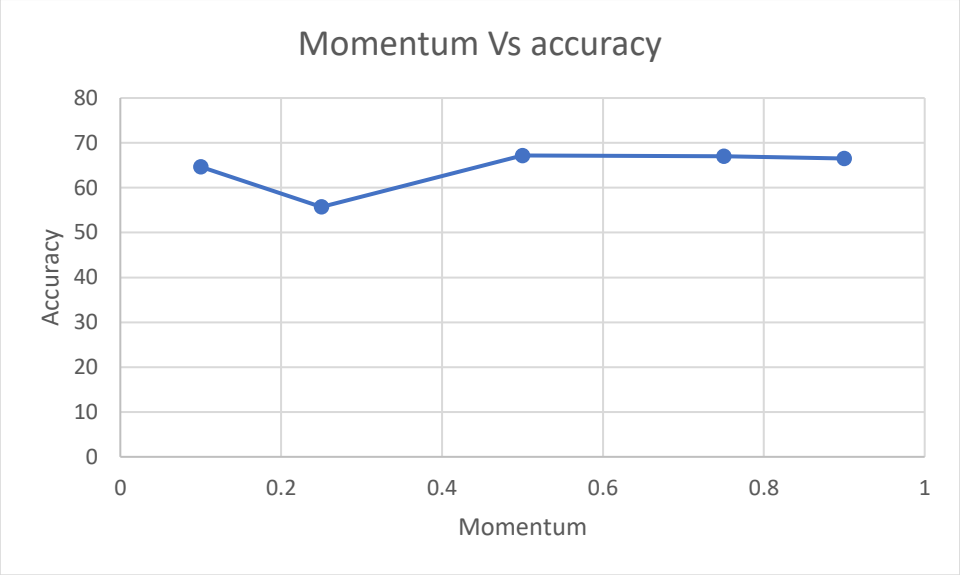


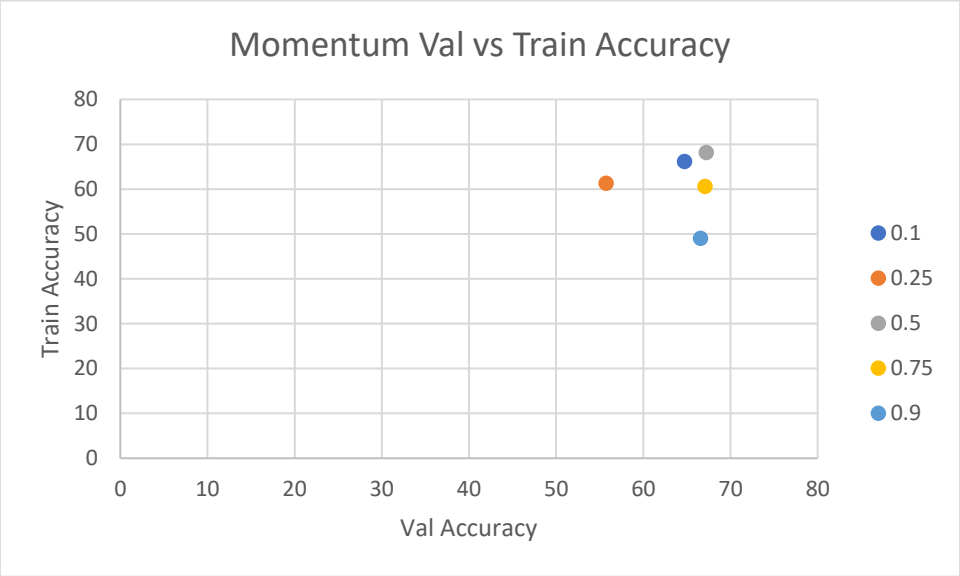
The training accuracy did go down, but the validation accuracy didn't improve at all.

It was here that I realised, that I should have looked at optimising momentum and weight decay the same way done in myrtle.ai. (Doing this because weight decay is regularisation technique.)

5.4 Momentum LR and Weight Decay

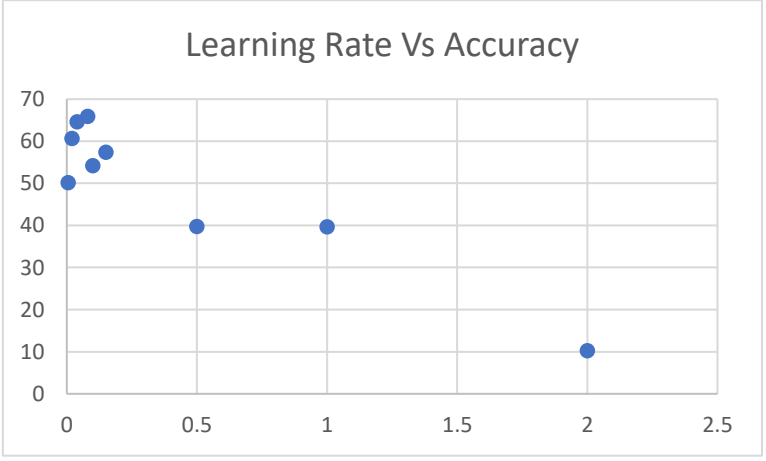
Removing drop out and optimizing momentum:

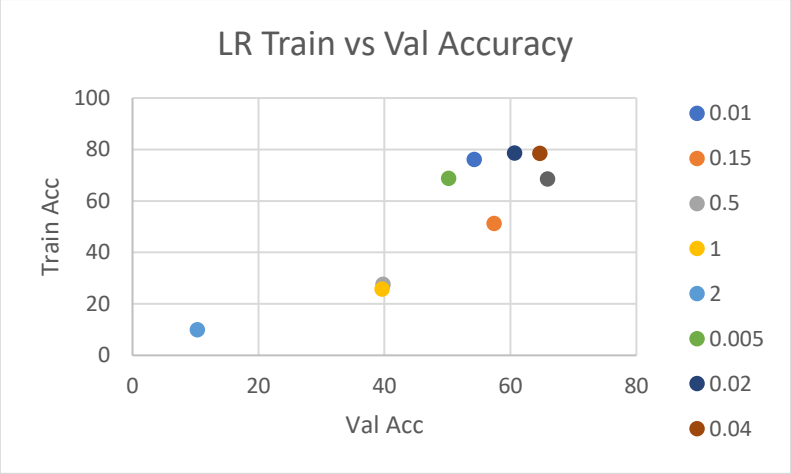




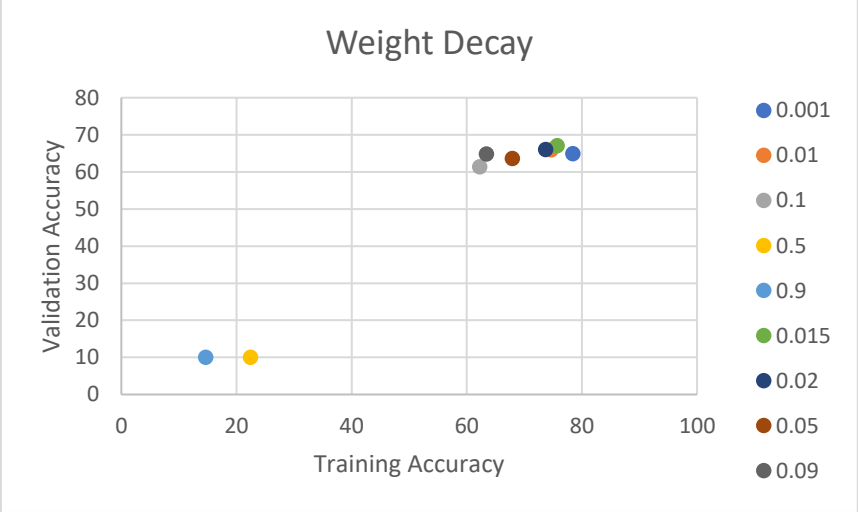
They momentum value of 0.5 had a very similar validation accuracy as 0.75 although the training accuracy was a lot higher, I feel like that could mean that the model is learning the data but overfitting it, which can be fixed with regularisation techniques to get a higher accuracy.

After This I went back and decided to optimise the learning rates again:





You can see a downward trend with an increase in learning rate. The best learning rate was found to be 0.04, which still has the higher training accuracy by 14 percent, which hopefully will be fixed in the weight decay optimisation.



Now plotting weight decay values, with a training of 20 epochs. The best validation accuracy came from weight decay of 0.015. Both the training and validation accuracy is still low. But it is better than the start.

5.5 Learning Rate Scheduler:

The accuracy from optimising the network with a static learning rate doesn't seem to be giving a good enough accuracy, barely getting to 70% on the validation set. Looking on Pytorch to have similar dynamic learning rates like that on the myrtle.ai model, I found learning rate schedulers [6].

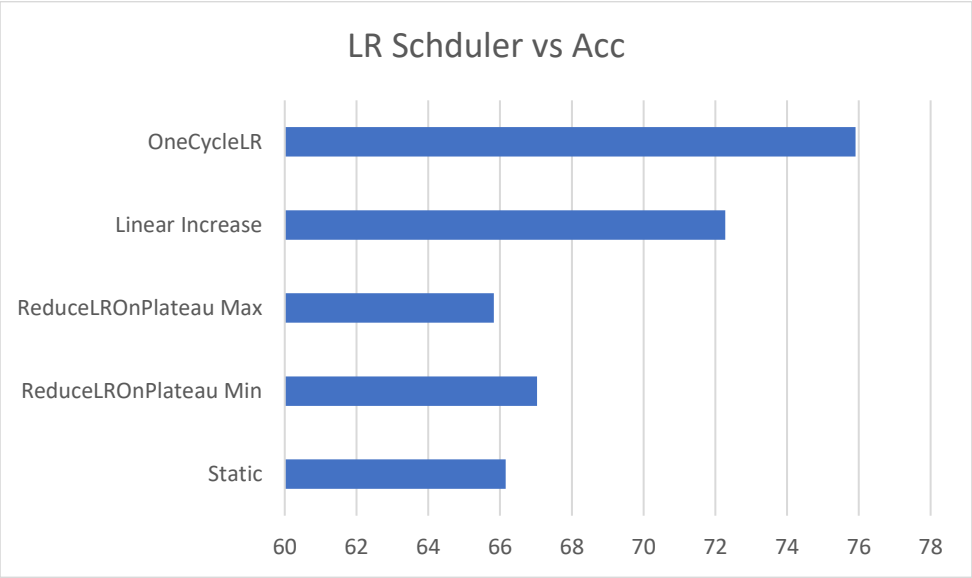
The functions I used:

- torch.optim.lr_scheduler.ReduceLROnPlateau()
- torch.optim.lr_scheduler.CyclicalLR

For ReduceLROnPlateau(), pytorch says that the learning rate reduces when a metric stagnates, I am guessing this is the loss function, since im not feeding it the accuracies.

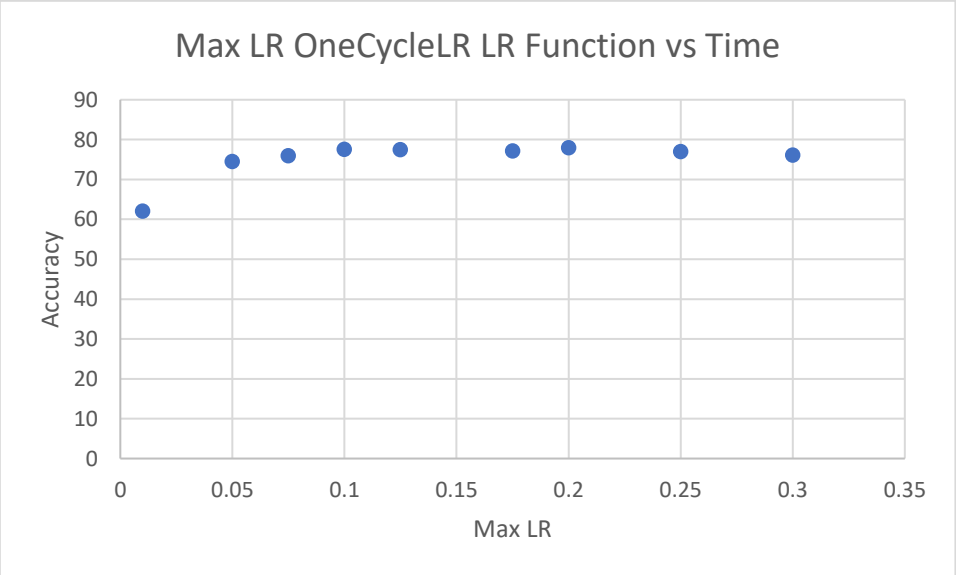
The linear increasing learning rate, with OneCycleLR, I could modify the 'mid point' of the function it produces to be near the end of the epoch training cycle, which basically gave a linear increase to learning rate.

For the OneCycleLR, I used a max learning rate of 0.15 for this first trial.

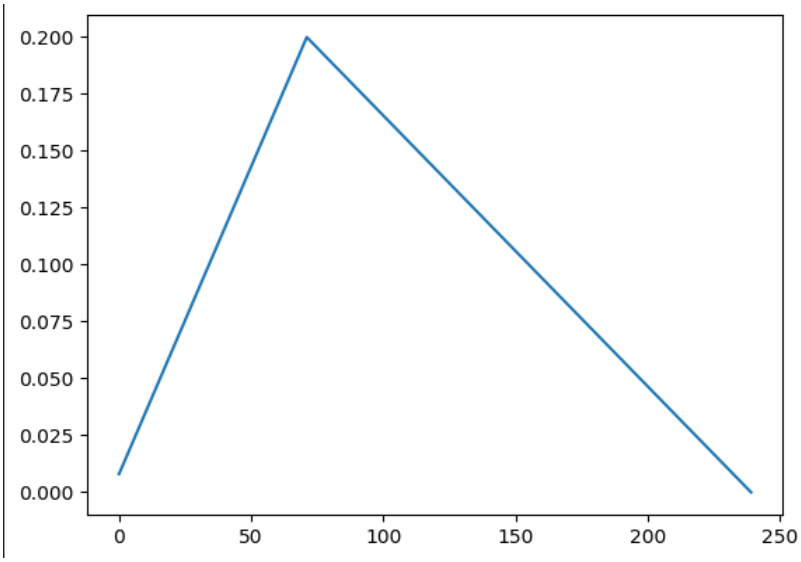


The OneCyleLR looks to be the fastest, it is similar to that used in myrtle.ai

Looking more into the OneCycleLR, changing the max LR:



Most get to a similar accuracy of 77%. The LR of 0.1 is chosen because it has both a good validation accuracy and testing accuracy. The others have lower testing accuracies.

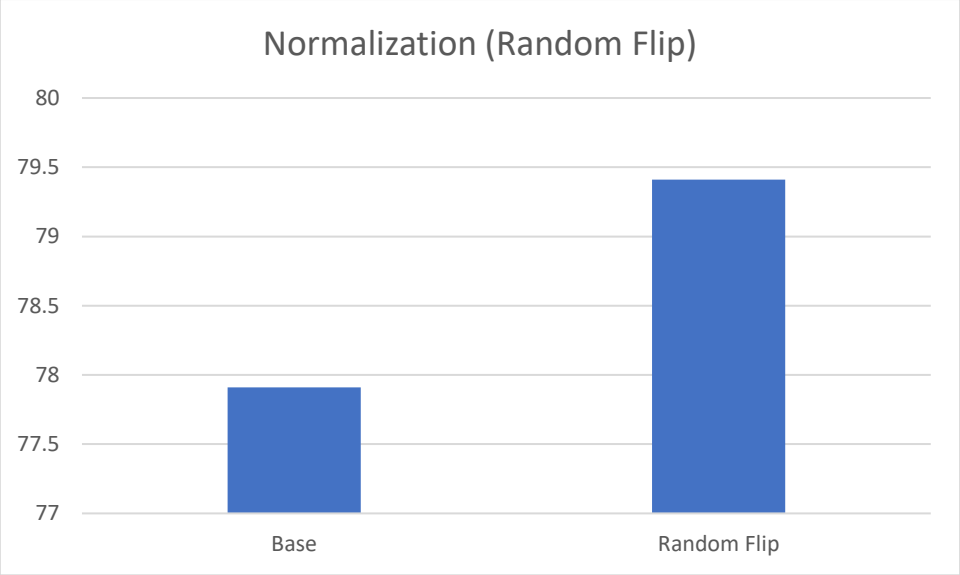


[5] This is how the OneCycleLR function looked like (note the is the one for max LR of 0.2)

5.6 Random Flip Regularisation:

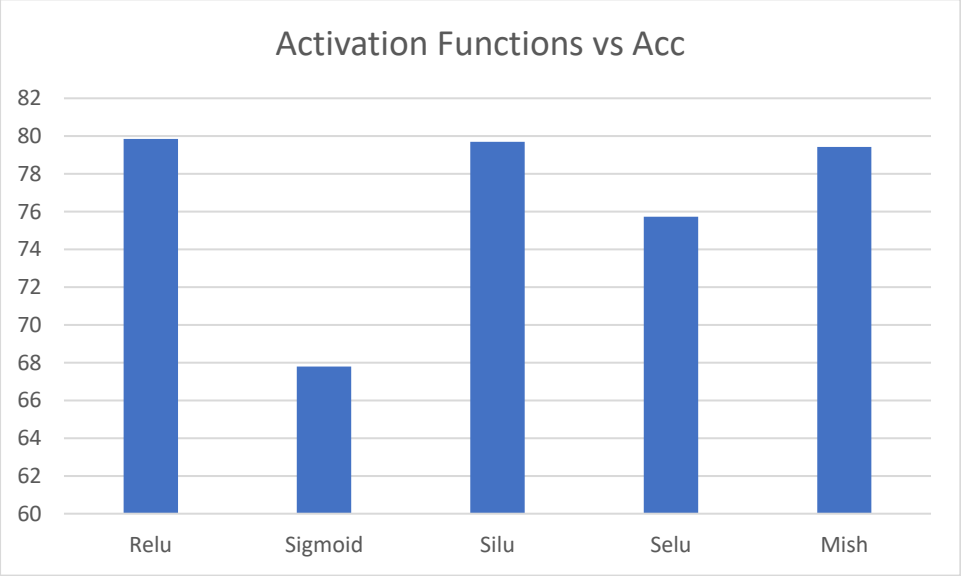
Epoch	Loss	Time	Val acc	Train acc
1	1.707556	13.81161	40.21	26.192
2	1.410372	12.40005	49.63	35.593
3	1.238942	12.24413	54.85	41.29
4	1.149174	12.51659	57.44	45.849
5	0.914529	12.0901	59.63	49.118
6	0.836017	12.17738	63.37	52.04367
7	0.811182	11.85057	63.7	54.71486
8	0.753971	11.63435	66.57	57.04425
9	0.719056	11.36715	65.95	59.16022
10	0.537512	11.46039	68.06	61.0168
11	0.514157	11.34847	69.76	62.86145
12	0.477307	11.42486	69.62	64.58117
13	0.391324	11.38775	70.39	66.14246
14	0.372732	11.47054	70.41	67.66443
15	0.277522	11.55009	72.48	69.07267
16	0.243399	11.39245	72.71	70.50338
17	0.168562	11.60479	73.02	71.89259
18	0.106512	11.40627	73.96	73.24567
19	0.060598	11.57859	74.35	74.55389
20	0.036858	11.5222	74.52	75.7968

Even though the model is already doing better on validation set than test set, I wanted to see what random flipping the input training and testing data would do to the validation accuracy. Turns out, it helped it by a few percent.



5.7 Activation Function:

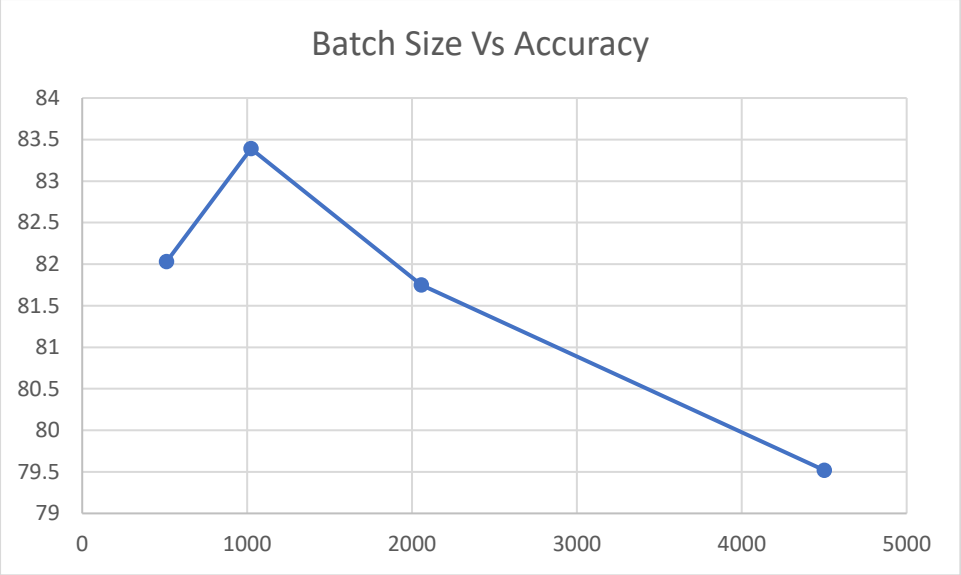
Now, I wanted to quickly check if Relu was the best activation function for CIFAR10. Testing:



Relu still has the best accuracy, and the training times are basically the with all functions.

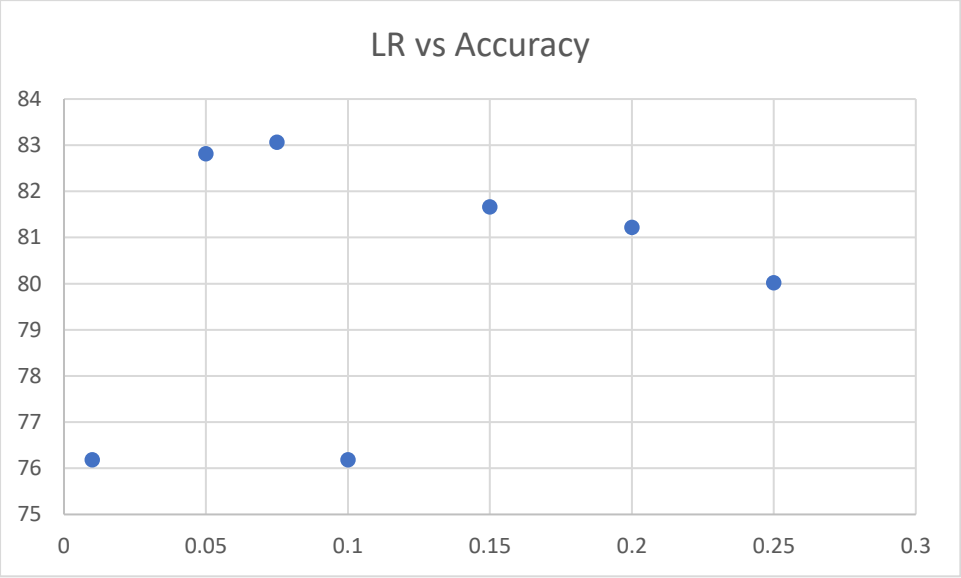
5.8 Batch Size:

At this point I felt like with the basic hyper parameter optimisation there should at least be a better accuracy. So I decided to go back and test batch size for the accuracy of the model, since the previous section, I traded accuracy training speed, in this, I will look at smaller batch sizes and their effectiveness on validation accuracy:



Along with an increase in accuracy, I got an increase in training speed from batch size 4500: 11.05 seconds down to 10.76 seconds. My guess on the better accuracy is that with the really large batch size of 4500, the data loading bottle necked the speed as so much had to be loaded at once (and I can't increase the number of workers) and that with large batch sizes, the changes are so large between epochs that the model keeps forgetting what it learnt in previous epochs.

With this new batch size, I need to redo the learning rate:

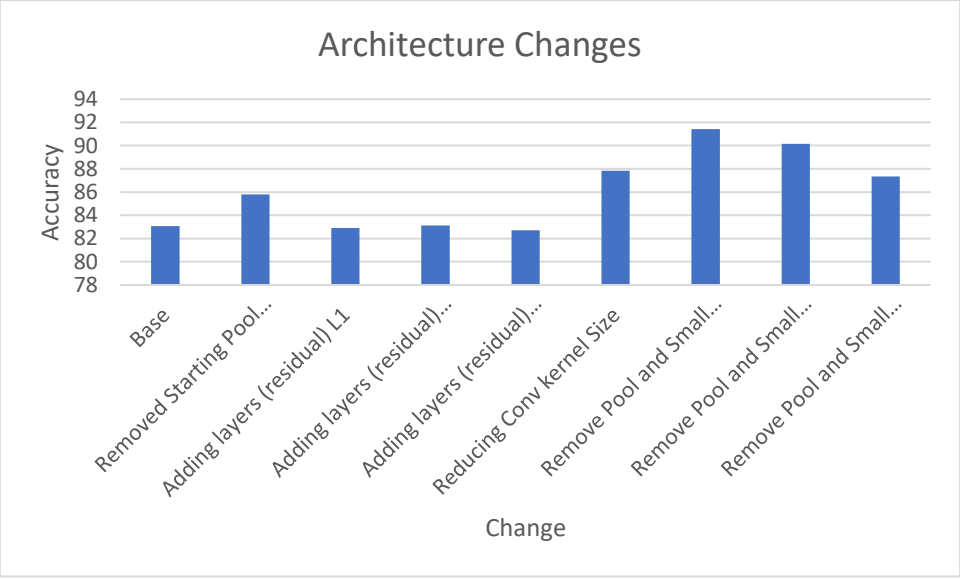


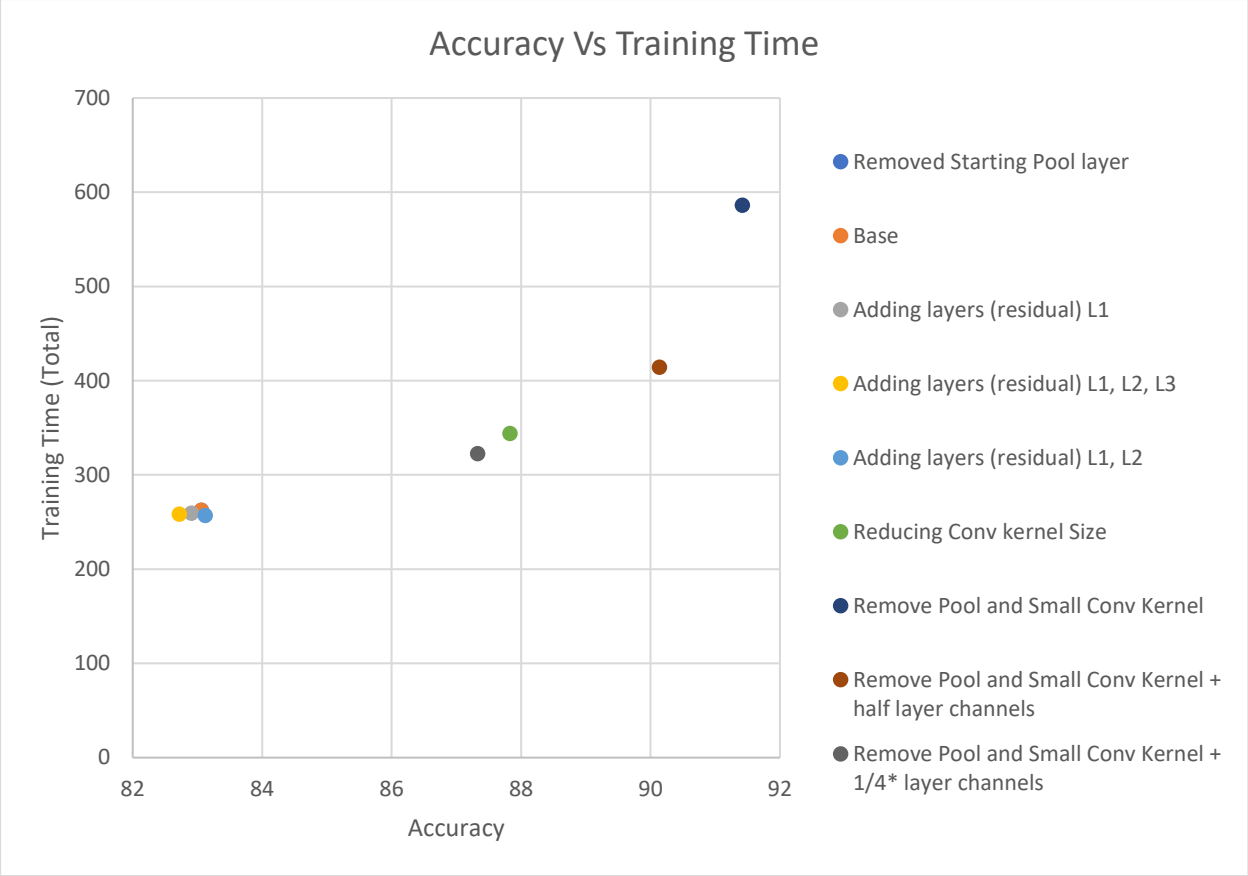
A max learning rate for OneCycleLR of 0.075 looks to be the best. Giving an accuracy of 83.06%.

5.9 Modifying Architecture:

There were a few different ideas I got from looking a the myrtle.ai resnet optimisation but I didn't know how do most of them. The base case was 83.06% accuracy with 13.118 second per epoch. These are the few that I tried:

1. Removing the starting average pool layer. This layer caused the data dimension to be severely reduced, this meant that the image was really small so when it went into the residual layers, it didn't have to process as much but also didn't get a better accuracy. With this the accuracy I got was 85.8% with 15.9 second per epoch
2. I tried to add a conv-norm-relu function that added these to the residual layers of the network. These didn't seem to change anything, even the training time and accuracy were off by a little deviation (not enough to warrant changing), I checked in the summary function of the network to see if my function changed anything which it did, there were extra layers but they didn't do anything.
3. The next one was changing the first convolution layer to a 3x3 kernel with no padding or strides. Just like the average pooling layer, I wanted to see what the accuracy would be like if the convolution layer of kernel size 7 with strides and padding of 2 and 3 respectively did. The accuracy was a lot better, almost reaching the 90% mark but came at the cost of 4-5 seconds extra training time per epoch. 87.83% accuracy with 17.19 seconds per epoch.
4. Combining both the reduced kernel size and removal of the pooling layer, this would maximize the data going into the residual layers, meaning maximisation of data retention as none of it is lost in the starting non residual layer. This gave the highest training time but also gave an accuracy of 91.42%.
5. The later last two data tests was using the removed kernel and making a smaller convolution kernel layer at the start. But also testing it with reduced size output channels for the residual network, this was in the hopes to that there wasn't as many features that needed to be learnt, and that smaller less channels can extract them just as well. The output accuracies were not as good as I expected, although one of them gave 90% accuracy with average 20 seconds per epoch.





From this graph you can see there is a exponential relationship between the time to desired accuracy of the model and the it takes to train it.

If the target accuracy was still 90%, I would have chosen the method which removes the pool, has a small conv kernel, and halves the residual layer output sizes. Even though it was 2 minutes and 30 seconds longer to train. But since 80% accuracy is the target, the base case looks to be the best with a training time of 4 minutes and 22 seconds and 83.06% accuracy.

5.10 Final Models

These two different models will have two jupyter notebook submissions:

1. The quicker model with 83.06% accuracy will be under section_3_2.ipynp
2. The slower 90% accuracy model will be under section_3_2.ipynp

6 WHAT I COULD DO NEXT:

For the first model, I didn't have time to go back and create a dynamic learning rate for it, which could have increased the accuracy even more. This would give me more room to change the architecture (from what I learnt in the next section, which I didn't have time to properly update in section 2).

For Section 2 the learning rate scheduler could be optimised even more, there are different functions, or even hand made ones. I feel like having the basic starting rise to the learning rate to half the epochs works well but reducing the learning rate to 0 after than isn't the optimum way, maybe having a slightly higher rate constant after 75% of the epochs have been trained.

7 REFERENCES:

[1] Nouman, *Writing ResNet from Scratch in PyTorch*, Paperspace Blog, 2019. [Online]. Available: https://blog.paperspace.com/writing-resnet-from-scratch-in-pytorch/. [Accessed: May 23, 2023].

[2] Radečić Dario, *PyTorch: How to Train and Optimize A Neural Network in 10 Minutes*, Appsilon, 2022. [Online]. Available: https://appsilon.com/pytorch-neural-network-tutorial/. [Accessed: May 23,2023].

[3] Bronwlee Jason, *How to Reduce Overfitting With Dropout Regularization in Keras*, Machine Learning Mastery, 2018. [Online]. Available: https://machinelearningmastery.com/how-to-reduce-overfitting-with-dropout-regularization-in-keras/#:~:text=CNN%20Dropout%20Regularization,is%20just%20a%20rough%20heuristic.&text=In%20this%20case%2C%20dropout%20is,cell%20within%20the%20feature%20maps . [Accessed May, 23, 2023]

[4] Desi_Avenger, *Pytorch-CNN_Resnet18-CIFAR10*, Kaggle, 2020. [Online]. Available: https://www.kaggle.com/code/greatcodes/pytorch-cnn-resnet18-cifar10/notebook . [Accessed May: 23,2023]

[5] Monigatti Leonie, *A Visual Guide to Learning Rate Schedulers in Pytorch*, Medium, 2022.[Online]. Available: https://towardsdatascience.com/a-visual-guide-to-learning-rate-schedulers-in-pytorch-24bbb262c863 . [Accessed May, 23, 2023].

[6] Tam Adriuan, *Using Learning Rate Schedule in Pytorch Training*, Machine Learning Master, 2023. [Online]. Available: https://machinelearningmastery.com/using-learning-rate-schedule-in-pytorch-training/ . [Accessed May 23, 2023]

[7] Maheshkar Saurav, *How To Use GradScaler in Pytorch,* Weights and Biases, 2023. [Online]. Available: https://wandb.ai/wandb_fc/tips/reports/How-To-Use-GradScaler-in-PyTorch--VmlldzoyMTY5MDA5 . [Accessed: May 23, 2023].

[8] davidcpage, *cifar10-fast*, Github, 2019. [Online]. Available: https://github.com/davidcpage/cifar10-fast . [Accessed: May 23, 2023].

[9] *How to Train Your ResNet*, Myrtle.ai, n.a. [Online]. Available: https://myrtle.ai/learn/how-to-train-your-resnet/ . [Accessed: May 23, 2023].

[10] ptrblck, How to know if model is half or full precision?, Pytorch Foums, 2020. [Online]. Available: https://discuss.pytorch.org/t/how-to-know-if-model-is-half-or-full-precision/88817. [Accessed: May 23, 2023]