Exploring accuracy in fastai's Image Classification using Sunny-vs-Cloudy dataset:

https://www.kaggle.com/polavr/twoclass-weather-classification

Notes to add:

I didn't have enough time to run the experiments on the resnet50 model due to underestimating training times however I have left the notes and research I did about it in the model's section.

Intro:

This report will cover the building, testing and accuracy of Fastai's Convolutional neural network (CNN) I used to train on my dataset as it is known for image recognition training. The reason for using Fastai is that it provides us with the ability of transfer learning which is where the last layer of a pre trained model is changed to use the dataset instead thereby making the training of my dataset very quick. Even though it is quick we don't know how accurate it is as the pre trained model may have been trained on different types of data which we can't see, however by fine tuning parameters before training we can increase the accuracy and reliability of the model on the dataset.

My dataset consists of 2 folders, one labelled test and the other train indicating what they should both be used for. Both contain pictures of sunny and cloudy weather which is what the model will hope to label correctly. We can see the size of the dataset from the number of files of here:

```
import os
labels = os.listdir(path_img)
print("No. of labels: {}".format(len(labels)))
print("-----------------")

for label in labels:
  print("{}, {} files".format(label, len(os.listdir(path_img+"/"+label))))
```

```
No. of labels: 2
-----------------
sunny, 5000 files
cloudy, 5000 files
```

Metrics:

To cover and compare the accuracy when changing parameters and models I will be using the metrics of "accuracy" which gives the percentage of correct guesses and "error_rate" which provides a percentage of incorrect guesses. These provide clear a clear understanding of how accurate the model is. I will also be using a confusion matrix to identify and visualize each of the classes the model's incorrect guesses were in.

Current Accuracy:

I am using this model here to be compared to after results and to be experimented with and hopefully improved upon after changing a few of the variables to test against each other.

This model will have:

Batch size = 25

Image size = 224

Optimizer = default/none

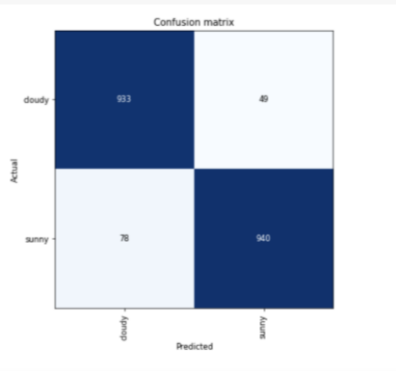
Architecture = resnet34

These are listed to ensure a fair test and a accurate comparison between this example and the others.

Results:

epoch	train_loss	valid_loss	accuracy	error_rate	time
0	0.483782	0.262389	0.895500	0.104500	01:29
1	0.267604	0.209919	0.918500	0.081500	01:24
2	0.197468	0.173644	0.933000	0.067000	01:22
3	0.145428	0.174711	0.936500	0.063500	01:22

epoch	train_loss	valid_loss	accuracy	error_rate	time
0	0.114736	0.172646	0.937500	0.062500	01:38
1	0.110051	0.174499	0.936500	0.063500	01:38



Above shows 3 matrixes that can be used to track how well the model done, after 4 epochs the accuracy came to 93.6% with a average time taken of 1:22 minutes. After unfreezing and running again the accuracy came to a highest of 93.7% before falling back down 1.2%. This could be due to overfitting the model with the learning rate chooses here. But can gather that the highest accuracy was around 93.7%.

Model Choice:

Resnet information reference - https://towardsdatascience.com/resnet-the-most-popular-network-in-computer-vision-era-973df3e92809#:~:text=ResNet%20was%20motivated%20to%20address,poorly%20for%20extremely%20deep%20networks.

DenseNet information reference - https://medium.com/the-advantages-of-densenet/the-advantages-of-densenet-98de3019cdac

EfficientNetV2 information reference - https://paperswithcode.com/method/efficientnetv2#:~:text=EfficientNetV2%20is%20a%20type%20convolutional,to%20jointly%20optimize%20training%20speed.

The models ive chosen to compare are ResNet50, DenseNet121 and EfficentNetB2 , ive chosen to compare these as they all vary in the architecture of the model and specialize in different ways. For each of these ill be testing them over 4 epochs to keep a fair test and comparison between them.

I will also be using Fp16 in my model as I want to use a lager architecture that will take longer to train, this will take down the time of that significantly and will also relive the stress of the GPU memory which is important to consider when using Colab especially.

ResNet50

ResNet50 has 50 layers and is popular choice of model in transfer learning since it has high accuracy and fixes the degradation problems when training error increases as layer increased which is a problem in other models, however, uses more memory due to storing the activations from the previous layer to fix the degradation issue.

DenseNet121

DenseNet121 is a CNN architecture that was created to “improve higher level architectures” as the problem was that “many of the layers in higher-layer networks were in a sense redundant”. The solution to this was to create a model that connects the following layer to all previous layers thereby solving the issue.

EfficentNetB2

EfficentNetV2 as the name states, it is designed to be efficient as possible in parameters, training speed and computational cost. It aims to create a balance between accuracy and efficiency which will be useful when compared to the other selected examples.

Predictions:

Before running these models with my "default" values for the other hyperparameters I would predict that the DenseNet model to achieve the highest accuracy due to each of the previous layers connecting to the following layer so that the data being passed through is always in use and since it has 121 layers being used in this way.

The architecture with the fastest training time I predict will be EfficientNetB2 as it is built to be efficient as possible including the training time taken to complete 4 epochs.

Results: Batch size 25, image size 224, optimizer default

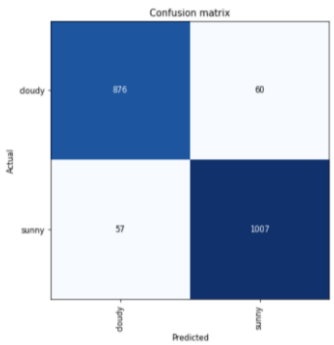
Learning rate reference minute 31:00: https://www.youtube.com/watch?v=uFugJXCjZ5M

When training the DenseNet121 it took on average about 3:47 to complete each epoch and after training giving an accuracy of 93.55%. After this was completed, I plotted the learning rate and found the minimum and steepest points using them to improve the learning rate in the next set of training. I then unfroze learner and carried out another 2 epochs which resulted in only a .02% increase in accuracy and higher training times. This could be since the learning rate change was not appropriate.

After changing the learning rate slice parameters to the ones below the accuracy has increased to 94.15% showing the changes made were beneficial.

```
learn_dense.unfreeze()
learn_dense.fit_one_cycle(2, lr_max=slice(10**-6,10**-5))
```

epoch	train_loss	valid_loss	accuracy	error_rate	time
0	0.082192	0.159675	0.939000	0.061000	04:24
1	0.086579	0.163073	0.941500	0.058500	04:25



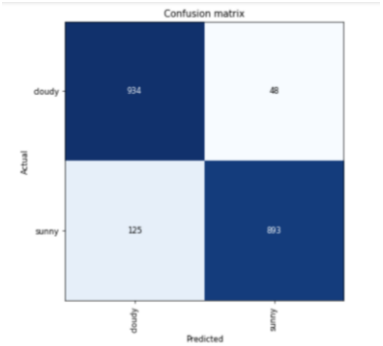
Above is a confusion matrix identifying where each prediction was made and guessed correctly or incorrectly.

I then used the same parameters on the EfficientNetB2 architecture to gather some base accuracy of the model. The accuracy after 4 epochs came to 91.4% which was lower than the DenseNet121 run however took 2min quicker than it.

epoch	train_loss	valid_loss	accuracy	error_rate	time
0	0.664338	0.498145	0.806000	0.194000	02:46
1	0.315621	0.259884	0.885500	0.114500	02:44
2	0.236487	0.211961	0.914000	0.086000	02:44
3	0.211611	0.201439	0.914000	0.086000	02:44

After unfreezing and choosing a learning rate the highest accuracy came to 91.5% this is still substantially lower than DenseNet as that came to 94.1%

epoch	train_loss	valid_loss	accuracy	error_rate	time
0	0.197643	0.198067	0.915000	0.085000	03:16
1	0.196271	0.204677	0.913500	0.086500	03:16



Above shows a confusion matrix for the EffeicientNetB2 model. Key points to take are is that it rarely guesses actual cloudy and predicted sunny. This could be due to overfitting the model with not a correct learning rate.

Parameters: batch size, picture size, optimizer

Further improvements to accuracy could be made through the changes in hyperparameters such as batch size, picture size and optimization choices, which all play a key role in the models training process.

Batch size: 10, 40

I have chosen these batch sizes as they can give a clear representation of how scaling the batch size can affect performance and any tradeoffs that come with it. I have also made sure to keep them all 15 apart from each other to ensure a fair test. I would scale higher than 40 however I don't have the computing power to do so with certain models.

While training the model with different batch sizes I have observed that larger sizes will take a longer amount of time to train, and smaller sizes take less time. This can easily be explained as with larger sizes it will go through the whole data set a lot more quickly. However, I have also noticed accuracy going down with the larger sizes. This could be because the model takes an estimate from a larger range gap.

10:

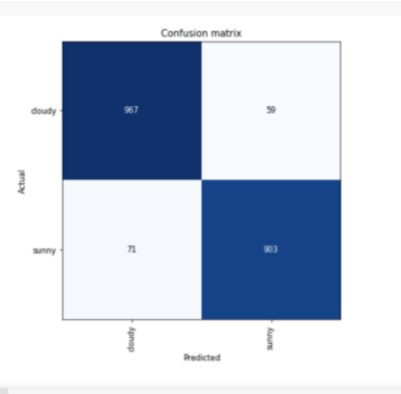
Training Densenet121 on a batch size of 10 supports the finding that lower batch size equals a larger amount of training time as the training time for each epoch averages out to be 4:01 minutes taken.

```
learn_dense.unfreeze()
learn_dense.fit_one_cycle(2, lr_max=slice(10**-6.3,10**-3))
```

epoch	train_loss	valid_loss	accuracy	error_rate	time
0	0.196176	0.169365	0.933500	0.066500	04:38
1	0.143370	0.170117	0.935000	0.065000	04:39

The figure shows that that after another 2 epochs accuracy rose to 93.5%. Which doesn't support my statement previously that smaller batch sizes would equal a more accurate

model. This again could be due to other factors as well such as the architecture of Densenet121 or the learning rate.



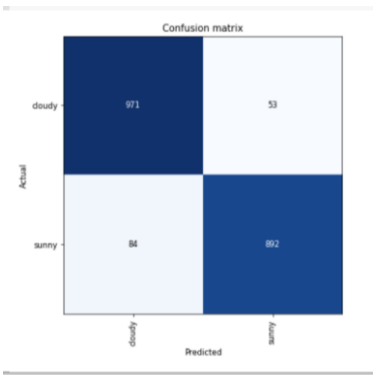
The figure shows the classes each guess was divided into.

Training the EffecientNetB2 model with a batch size of 10 resulted in the following output. Which shows the last epoch reaching 92.9% accuracy in 2:55

epoch	train_loss	valid_loss	accuracy	error_rate	time
0	0.559350	0.411497	0.819000	0.181000	02:54
1	0.355198	0.234726	0.908000	0.092000	02:53
2	0.281668	0.195934	0.924000	0.076000	02:54
3	0.294532	0.183949	0.929000	0.071000	02:55

After training another 2 epochs after the accuracy then increased to 93.1% after the 2nd epoch showing that the accuracy increases with a smaller batch size.

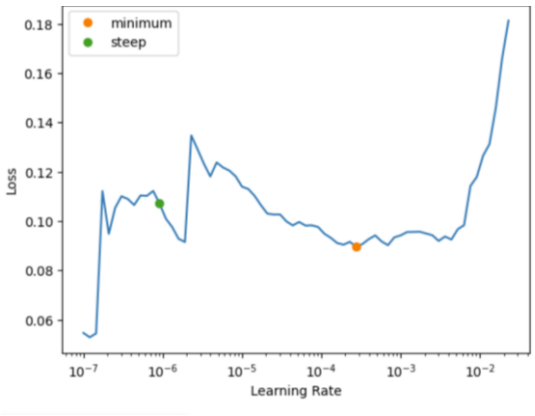
epoch	train_loss	valid_loss	accuracy	error_rate	time
0	0.220650	0.183107	0.932500	0.067500	03:41
1	0.284979	0.182876	0.931500	0.068500	03:41



The figure shows the confusion matrix for this model.

40:

For DenseNet121 the accuracy started on the first epoch started off quite low at 88.9% however by the last epoch jumping to 93.65% which compared to the batch size of 25 is an improvement which doesn't fit my first observation when assuming a bigger batch size would equal lower accuracy. However, this could be due to the nature of DenseNet121 being more complex and benefiting from the larger batch sizes.

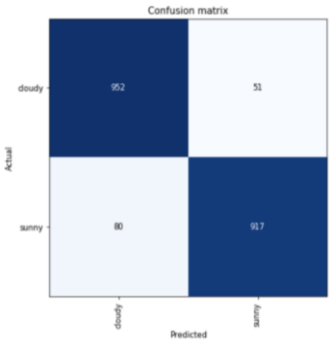


From the learning rate to loss graph above it was quite hard to pick and accurate slice to represent where the learning rate was it its steepest and also lowest. After attempts to find the optimal learning rate slice this was the one that gave the best accuracy.

```
learn_dense.unfreeze()
learn_dense.fit_one_cycle(2, lr_max=slice(10**-6,10**-6.5))
```

epoch	train_loss	valid_loss	accuracy	error_rate	time
0	0.092884	0.171069	0.938500	0.061500	04:21
1	0.075174	0.170876	0.938000	0.062000	04:21

In the figure above it shows accuracy after 2 epochs being at 93.8% which isn't as high as the batch size of 25. This could either be due to the learning rate input or the batch size increase.



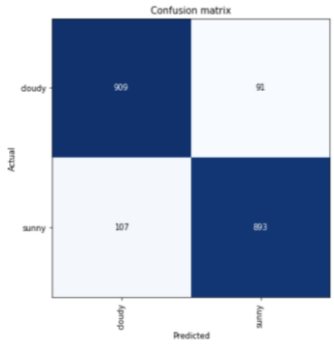
The confusion matrix shows a more even distribution compared to the batch size of 25 when guessing correctly and that Actual: “cloudy” Predicted: “Sunny” was the least guess class.

I then ran EffecientNetB2 with a batch size of 40, this then outputted the figure shown below. This shows the lowest accuracy so far after 4 epochs coming to 89.9%. This is leading me to believe that my previous prediction would be correct as with batch size accuracy also dropped.

epoch	train_loss	valid_loss	accuracy	error_rate	time
0	0.749080	0.479203	0.811000	0.189000	02:39
1	0.360577	0.278260	0.881000	0.119000	02:38
2	0.246948	0.251307	0.899000	0.101000	02:39
3	0.215900	0.238903	0.899500	0.100500	02:38

After adjusting the learning rate another 2 epochs were ran giving a highest percentage of 90.2% then lowering to 90.1% which will be due to the learning rate choose.

epoch	train_loss	valid_loss	accuracy	error_rate	time
0	0.196539	0.237839	0.902000	0.098000	03:08
1	0.198407	0.234216	0.901000	0.099000	03:08



The figure above shows the confusion matrix for the model.

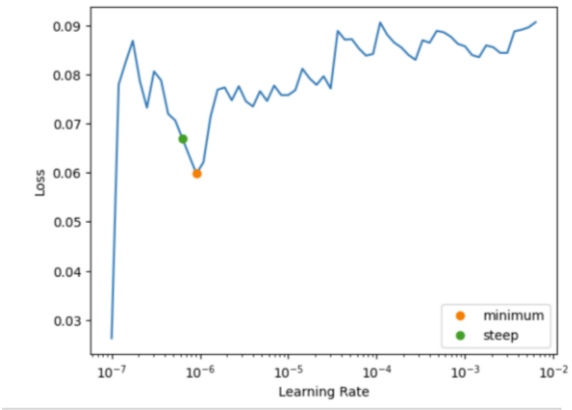
Image size: 100, 400

Prediction:

When decreasing image size, I expect the model to train faster as there is less pixels to use the CNN filter on. Therefore, increasing image size would do the opposite and increase training time and increase accuracy of the model. Ill be using each architecture best found sizes when testing this.

100:

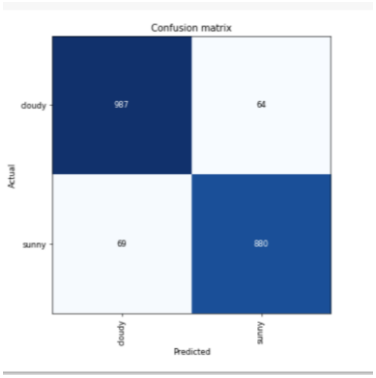
On initial run of DenseNet121 with a batch size of 25. I found that after the 4 epochs the highest accuracy was 93.1% and that it took an average of 3:50 to train. The accuracy Is the lowest yet in the DenseNet tests which is no surprise as it was expected in the prediction statement showing they corelate.



The results below show the final 2 epochs after unfreezing for the 100-image size run. Key things to point out are is that the accuracy jumped up to 93.5% this could be the result of good learning rate appliance to the fit one cycle as the graph was easy to read.

```
learn_dense.unfreeze()
learn_dense.fit_one_cycle(2, lr_max=slice(10**-6.3,10**-6))
```

epoch	train_loss	valid_loss	accuracy	error_rate	time
0	0.091168	0.154553	0.937500	0.062500	05:05
1	0.087770	0.152338	0.935000	0.065000	05:03



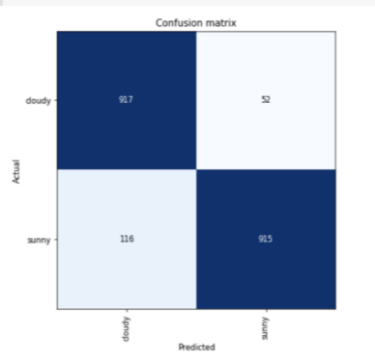
Above is the corresponding confusion matrix.

The accuracy of a image size of 100 has come to 91.8% which is nearly 2% lower than DenseNets however taking an average of 2:47 per epoch to complete which is a lot quicker than DenseNet.

epoch	train_loss	valid_loss	accuracy	error_rate	time
0	0.687754	0.417249	0.833500	0.166500	02:44
1	0.305075	0.240903	0.903000	0.097000	02:44
2	0.253685	0.211469	0.915000	0.085000	02:50
3	0.190104	0.205629	0.918000	0.082000	02:55

The accuracy then decreased after each epoch this could be due to overfitting and not choosing a suitable learning rate for the model however it is a good indication of the total accuracy for this size of image.

epoch	train_loss	valid_loss	accuracy	error_rate	time
0	0.196839	0.204715	0.917500	0.082500	03:18
1	0.185957	0.208284	0.916000	0.084000	03:16



The figure shows confusion matrix for the model above.

400:

After running the image size of 400 on the DenseNet121 I found that the average time was higher than the 100-image size coming to 4:23 minutes to train and accuracy showing to be the highest so far in a DenseNet run at 93.31% after the first 4 epochs.

In the figure below it shows the accuracy after unfreezing and applying learning rate changes didn't change at all after the first epoch

```
learn_dense.unfreeze()
learn_dense.fit_one_cycle(2, lr_max=slice(10**-6.3,10**-5.3))
```

epoch	train_loss	valid_loss	accuracy	error_rate	time
0	0.094826	0.168657	0.933500	0.066500	04:55
1	0.092934	0.177034	0.933500	0.066500	04:55

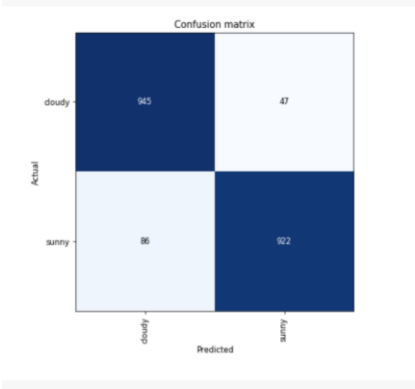


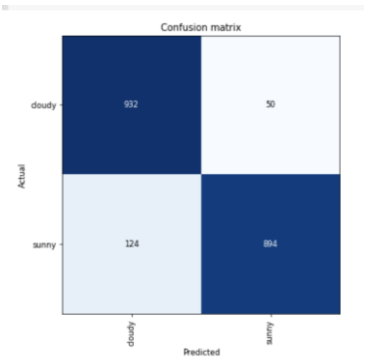
Figure shows confusion matrix for DenseNet121 with 400 image size.

The accuracy for EffecientNetB2s image size of 400 in prediction would provide a lower accuracy than the size of 100. Despite this being correct it is not by a substantial amount which could be due to other factors including the architecture

epoch	train_loss	valid_loss	accuracy	error_rate	time
0	0.687228	0.398144	0.845000	0.155000	02:40
1	0.354603	0.250388	0.899500	0.100500	02:42
2	0.254419	0.213154	0.917500	0.082500	02:43
3	0.198412	0.203616	0.915000	0.085000	02:43

Further training after adjusting the learning rate shows that the accuracy increased by 0.3% then dropping 0.5% this could be due to pssibly overfitting the model.

epoch	train_loss	valid_loss	accuracy	error_rate	time
0	0.191867	0.202524	0.918500	0.081500	03:11
1	0.197211	0.209446	0.913000	0.087000	03:16



The figure shows confusion matrix for the model.

Optimizer: RMSprop, SGD

RMSprop information reference: https://machinelearningmastery.com/gradient-descent-with-rmsprop-from-scratch/#:~:text=RMSProp%20is%20designed%20to%20accelerate,in%20a%20better%20final%20result.

SGD information reference: https://towardsdatascience.com/stochastic-gradient-descent-clearly-explained-53d239905d31

RMSprop is an optimizer that is built to automatically change the learning rate for each parameter. It does this by dividing the learning rate by the increasingly dropping average of the squared gradients.

SGD is an optimizer that makes use of gradient descent. SGD works by adjusting the parameters in the opposite direction of the gradient therefore refining them.

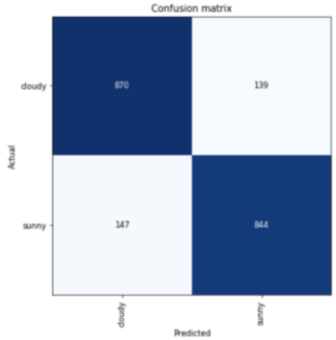
RMSprop:

After training DenseNet121 on RMSprop for 4 epochs it has resulted in the worst accuracy so far, this could indicate that the RMSprop is not suitable for this architecture and would be better trying others.

epoch	train_loss	valid_loss	accuracy	error_rate	time
0	0.808825	0.438607	0.800000	0.200000	04:11
1	0.521395	0.344326	0.851000	0.149000	04:09
2	0.447481	0.328318	0.858000	0.142000	04:09
3	0.412687	0.320843	0.860500	0.139500	04:08

After running a new learning rate for the next 2 epochs the accuracy went down resulting in a final accuracy of 85.7% this most likely due to the archeticture for being so low however. Choosing an inappropiate learning rate may have taken it down further.

epoch	train_loss	valid_loss	accuracy	error_rate	time
0	0.438374	0.322884	0.856500	0.143500	04:43
1	0.460239	0.321784	0.857000	0.143000	04:45

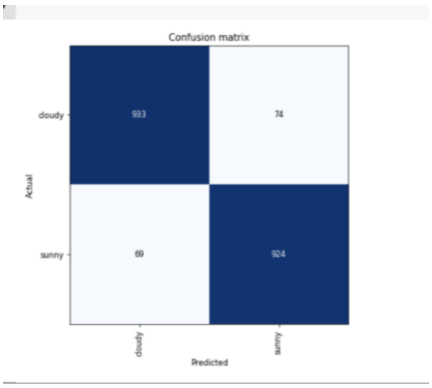


This figure shows the confsuion matrix for this model

I then ran EffecientNetB2 with RMSprop and It gave a good result coming to 92.4% after 4 epochs. Further growing 0.45% after the next 2 epochs aswell suggesting that the learning rate was appropiate and improved the accuracy of the model

epoch	train_loss	valid_loss	accuracy	error_rate	time
0	0.681546	0.387360	0.836000	0.164000	02:55
1	0.352553	0.231600	0.903500	0.096500	02:46
2	0.242232	0.185013	0.923500	0.076500	02:46
3	0.225094	0.180775	0.924500	0.075500	02:46

epoch	train_loss	valid_loss	accuracy	error_rate	time
0	0.198759	0.175723	0.926500	0.073500	03:22
1	0.183466	0.173678	0.928500	0.071500	03:21



This figure shows the confusion matrix for this model.

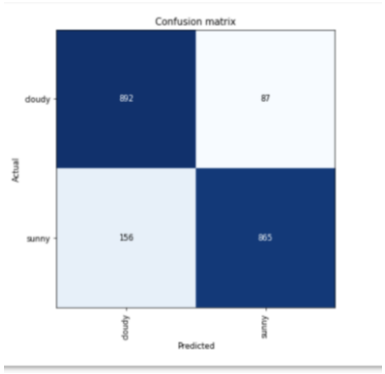
SGD:

Here in the SGD first run you can see that the accuracy is the lowest by far after 4 epochs not even reaching 90% for DenseNet121 and reaching 4minutes of training time each epoch. This is clearly not a suitable optimizer for this architecture, or the other parameters may not suit it.

epoch	train_loss	valid_loss	accuracy	error_rate	time
0	0.721624	0.488296	0.795000	0.205000	04:15
1	0.512628	0.302394	0.871000	0.129000	04:10
2	0.443349	0.303326	0.871000	0.129000	04:09
3	0.454402	0.282016	0.878000	0.122000	04:10

The next set of epochs reinforces this statement as even with improvements it improves a very insignificant amount.

epoch	train_loss	valid_loss	accuracy	error_rate	time
0	0.429219	0.304544	0.870500	0.129500	04:48
1	0.434313	0.282954	0.878500	0.121500	04:47



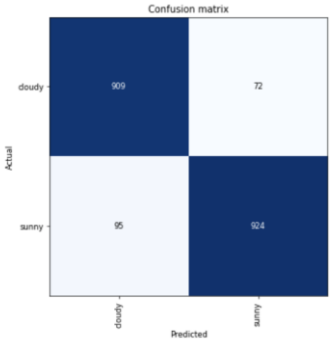
This figure shows the confusion matrix for this model.

I then ran the SGD on the EffecientNetB2 model and gave a better accuracy compared to the run on DenseNet121 however is not as accurate as EffecientNetB2 on RMSProp

epoch	train_loss	valid_loss	accuracy	error_rate	time
0	0.706595	0.499512	0.814000	0.186000	02:49
1	0.340213	0.245006	0.890000	0.110000	02:43
2	0.275207	0.206471	0.908000	0.092000	02:42
3	0.185873	0.201353	0.912000	0.088000	02:42

The accuracy continued to increase following the changes to the learning rate and another 2 epochs

epoch	train_loss	valid_loss	accuracy	error_rate	time
0	0.192234	0.203052	0.911500	0.088500	03:17
1	0.193633	0.198253	0.916500	0.083500	03:15



This figure shows the confusion matrix for this model.

Results/Conclusion:

When comparing back to my “default” model which was my base to improve upon I found that DenseNet121 provided the highest accuracy 94.1% with an image size of 224 and batch size of 25 this shows my prediction was correct. However, I found the fastest model to provide good accuracy was the default model with resnet34, batch size 25 and image size 224. In turn if I was building a model for fastest training and good accuracy, I would choose the default model. However, if I had more time to spend on training DenseNet121 would be ideal and the optimizers chosen to test with this model were both not ideal.

For increasing parameter sizes such as image size and batch size you’ll be using a lot more computational power but also may be increasing the accuracy of the model a lot. So is important to consider multiple factors when developing as a time constraint may mean dropping accuracy to get it completed. Another feature to test in feature experiments would be the epochs used. By demonstrating this I would imagine that the accuracy would decrease a lot but would make training times a lot smaller.

By manually tweaking and adding hyperparameters it is shown to improve the accuracy and performance of the model. This is useful for beginners who are learning about what factors can change the accuracy of the model. However, by adding augmentation features such as “progressive resizing”, “label smoothing” it can automatically change these parameters without the time-consuming need of human input on every test change made.