Introduction:

All the code can be found on: https://github.com/davpod/deep_learn_HW1/tree/main

I decided to use the Cassava Leaf Disease Classification dataset that was provided in the updates notice. The dataset was compiled by the **Makerere University AI Lab** as to help the farmers in Africa to be able to better and faster classify different diseases of the Cassava plant who is a key food security crop grown by smallholder farmers because it can withstand harsh conditions.

The dataset consists of 5 different leaf conditions ranging from what appears to be healthy leaves to not very healthy leaves mainly differentiated by spot color, size, count and shapes on the affected leaves and our task is to build an ai that is able to differentiate and classify those conditions properly given an image of affected leaves.

I will be using my personal computer as I have an NVidia rtx-gpu and I want to assess how good it is for different ai tasks, and since I'm too lazy and lacking time to figure out how to use university resources, I will be cutting down on different aspects to save runtime.

1 exploratory data analysis: https://www.kaggle.com/competitions/cassava-leaf-disease-classification/overview

you can get this data by running task1_eda.py

a) The dataset consists of 21397 images.

b) Each sample is of size 800x600 across 3 channels, who are RGB.
To save time and later be able to use the pretrained ImageNet models I will have to resize all images to 224x224 which will be a big hit on the data stored on the images.
We can use augmentation in the form of rotations, and mirroring, but since much of the data is quite colorful and noisy often including things that look more like a forest rather than a leaf, adding noise is not the best idea especially considering we resize the images massively. Also since the spots are also differentiated by color, changing the colors is also not that great of an idea as we may start feeding trash data.

c) The dataset is heavily imbalanced, with 1 class consisting of more than 60% of the whole dataset. 3 classes being relatively similar in size and class 0 being very small.
Class 0: 1087
Class 3: 13158
Class 1: 2189
Class 2: 2386
Class 4: 2577

d) There are benchmark results on the competition site, they are averaging between the 90%-91%, with the top being 91.3%, with seemingly all of the using pretrained imagenet models and finetuning them, and sometimes averaging out the results. All of them training their models on bigger images around the 448+ size, and upon manual inspection of data I have to admit its quite impressive, considering sometimes even I couldn't even differentiate between the classes, let alone correctly classify them.

e) For class 0, we have very different images who look nothing alike(mind its 3 random images not even hand picked, I will maybe get to hand picking later to show how some of the classes look the same.

class 1 is again very diverse, with one leaf looking completely healthy while the rest are a mess more normal

class 2 is looking more coherent, but its sometimes quite similar to class 1:

class 3 is again a total mess, the the left most image looking very similar to the right most image of class 1, the right most image looks for me also very similar to the right most image of class 2.

class 4 is again very diverse

all of those images were random sample yet we already see how diverse some of the images within the same class are, and how similar are the classes(after all it's the same plant), also we can easily notice how inconsistent the images are with what they show, with some properly showing the leaf, while some showing you a bush from far away, and adding noises by different lightning conditions, shades and background that further makes them more difficult to properly classify.

Custom neural network graph:

Before the actual assignment:

I have a big problem with me not properly reading the instructions before doing the work, initially that we are supposed to make the best model to begin with, and as advised in the class I have decided to do a small sanity check by trying to cause my model to overfit a small set of the data, yet I've reached a plateau of around 0.67 acc in both train and validation, suggesting that my model was underfitted, and it indeed was just guessing class 3 for almost everything without

realy learning. So I though that my model was lacking parameters, so with the advice of chat gpt I have expanded my model to a vgg style model with 4 convolution layers reaching a depth of 512, and 3 fully connected layers, making my model to be 56M parameters. After trying again the sanity overfit test, I have reached the same result of not reaching above 0.67 acc.
After a while it has occurred to me that my learning rate was a bit too high with it being 1e-3, which caused my model to bounce around a minima, and to learn too much trash in the beginning making it guess to guess class 3 for almost everything. After playing with the learning rate and gradually dropping it first to 1e-4, and eventually to 1e-5 I have finally was able to reach a high of around 0.9 for training and 0.6 for validation. While it was a great improvement, it still wasn't 100%, meaning no full overfit yet. So upon a closer inspection and consultation with not so smart gpt I have finally removed my augmentation that consisted of twists, mirroring and color changes and so I was finally able to overfit my model.
Later I have decided to once again make my model smaller as 56M params is a little too big for my tiny 7K balanced dataset(later on that in section B). you can probably still find my bigger model in backup.py
Trying to reduce model size found in backup.py: Another interesting observation I've made, if I try and not use fc3, my model which overfits quote quickly on a low set, gets stuck around the 0.15acc.
Turns out it was a bug in my code, I've removed fc3 but I still had "x = F.relu(self.fc2(x))" meaning I've passed the last layer via relu, meaning that instead of returning raw logits, I've returned a non negative number so I've blocked cross max entropy and so I couldn't progress further.
Upon removing conv4, we still are able to overfit, yet the param count is still around the 51M which is a bit too big for my likings.
Upon adding 1 more pooling after conv3 im finally down to 13M which is good.
I do overfit a lot more slowly, also when trying to overfit with an even smaller learning rate, the loss is very bouncy and unstable.
NOTE: fixed by reducing batch size from 96 to 32 for the 250 images set, now im able to reach acc=1 in 30 epochs, but still kinda bouncy.
After reducing batch size to 16 I've massively improved overfit time and finally got quite the stability, ill settle down with this model.
Conclusion: don't trust chatgpt to make you a model or try to solve your problems, the most scientific way of change things until it works and then confirm with gpt is still the most reliable method.
Also turns out its very important to not dropout after the last fc layer, as it confuses training and gives way smaller acc results compared to even validation, even if the model is biased toward 1 class from the very start.

a) **THE CODE IS FOUND IN**: **task2_cnn.py**. As I'm kinda restricted with time and I don't want my electricity bill to skyrocket, instead of the training on the full

dataset, I will instead stay with a more '"proof of concept" approach especially with the experimentation I have already done to get to where I'm(reducing a huge model to a smaller one and trying to make it to work, learning the difference between trying to make a bigger model to overfit vs making a smaller model to overfit, learning to adjust the learning rate, and understanding that lower and the more unbalanced the set, the lower learning rate I need or else it will just converge quickly to guessing the most popular choice, and be unstable), I'll be using only 1k images for the training as I know that the result will be 0.6 acc anyway as before balancing the dataset to be more equal, the model will just guess 3 for everything and be fine. I will be running the original train.csv as the train set.

Hyper parameters:

NUM_CLASSES = 5
INPUT_SIZE = 224
BATCH_SIZE = 32
NUM_EPOCHS = 10
LEARNING_RATE = 5e-5
K_FOLDS = 5
DROPOUT_PROB = 0.3
TRAIN_SIZE = 1000

With the probably final model:

```
class SmallCNN(nn.Module):  2 usages
    def __init__(self, num_classes=NUM_CLASSES, dropout_prob=DROPOUT_PROB):
        super(SmallCNN, self).__init__()
        self.conv1 = nn.Conv2d( in_channels: 3,  out_channels: 64,  kernel_size: 3, padding=1)
        self.bn1 = nn.BatchNorm2d(64)
        self.conv2 = nn.Conv2d( in_channels: 64,  out_channels: 128,  kernel_size: 3, padding=1)
        self.bn2 = nn.BatchNorm2d(128)
        self.conv3 = nn.Conv2d( in_channels: 128,  out_channels: 256,  kernel_size: 3, padding=1)
        self.bn3 = nn.BatchNorm2d(256)
        self.pool = nn.MaxPool2d( kernel_size: 2,  stride: 2)
        self.fc1 = nn.Linear(256*(INPUT_SIZE//16)*(INPUT_SIZE//16),  out_features: 256)
        self.dropout1 = nn.Dropout(dropout_prob)
        self.fc2 = nn.Linear( in_features: 256, num_classes)

    def forward(self, x):
        x = self.pool(F.relu(self.bn1(self.conv1(x))))
        x = self.pool(F.relu(self.bn2(self.conv2(x))))
        x = self.pool(F.relu(self.bn3(self.conv3(x))))
        x = self.pool(x)
        x = x.view(x.size(0), -1)
        x = F.relu(self.fc1(x))
        x = self.dropout1(x)
        x = self.fc2(x)
        return x
```

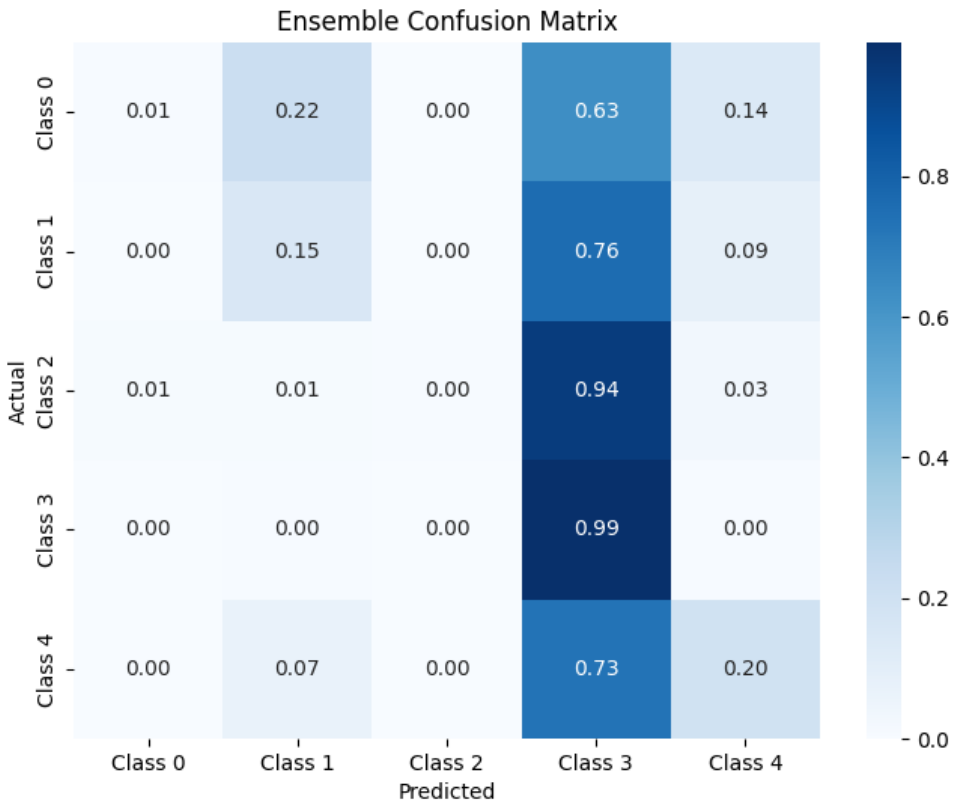
We get the results:

```
--- Fold 1 ---
Epoch 1/10 Train Loss: 1.8097 Acc: 0.5188 Val Loss: 1.4850 Acc: 0.5500
Epoch 2/10 Train Loss: 1.1106 Acc: 0.6238 Val Loss: 1.1998 Acc: 0.5850
Epoch 3/10 Train Loss: 1.0508 Acc: 0.6300 Val Loss: 1.2496 Acc: 0.5850
Epoch 4/10 Train Loss: 0.9996 Acc: 0.6362 Val Loss: 1.1236 Acc: 0.5950
Epoch 5/10 Train Loss: 0.9318 Acc: 0.6488 Val Loss: 1.1133 Acc: 0.5850
Epoch 6/10 Train Loss: 0.9496 Acc: 0.6512 Val Loss: 1.0956 Acc: 0.6000
Epoch 7/10 Train Loss: 0.9191 Acc: 0.6575 Val Loss: 1.1154 Acc: 0.6100
Epoch 8/10 Train Loss: 0.8416 Acc: 0.6687 Val Loss: 1.2740 Acc: 0.6000
Epoch 9/10 Train Loss: 0.8285 Acc: 0.6887 Val Loss: 1.0917 Acc: 0.6050
Epoch 10/10 Train Loss: 0.7818 Acc: 0.6875 Val Loss: 1.0907 Acc: 0.5900
Saved model for fold 1 at saved_models/model_fold1.pth

--- Fold 2 ---
Epoch 1/10 Train Loss: 1.6354 Acc: 0.5125 Val Loss: 1.4326 Acc: 0.5000
Epoch 2/10 Train Loss: 1.1416 Acc: 0.6038 Val Loss: 1.1489 Acc: 0.6500
Epoch 3/10 Train Loss: 1.0899 Acc: 0.6162 Val Loss: 1.0396 Acc: 0.6500
Epoch 4/10 Train Loss: 1.0176 Acc: 0.6262 Val Loss: 1.0716 Acc: 0.6450
Epoch 5/10 Train Loss: 0.9781 Acc: 0.6438 Val Loss: 0.9989 Acc: 0.6600
Epoch 6/10 Train Loss: 0.9254 Acc: 0.6538 Val Loss: 0.9665 Acc: 0.6600
Epoch 7/10 Train Loss: 0.8892 Acc: 0.6937 Val Loss: 1.0468 Acc: 0.6150
Epoch 8/10 Train Loss: 0.8388 Acc: 0.6900 Val Loss: 1.0752 Acc: 0.6200
Epoch 9/10 Train Loss: 0.8255 Acc: 0.6887 Val Loss: 0.9842 Acc: 0.6500
Epoch 10/10 Train Loss: 0.7615 Acc: 0.7212 Val Loss: 0.9666 Acc: 0.6600
Saved model for fold 2 at saved_models/model_fold2.pth

--- Fold 3 ---
Epoch 1/10 Train Loss: 1.7366 Acc: 0.5188 Val Loss: 1.5125 Acc: 0.4400
Epoch 2/10 Train Loss: 1.1501 Acc: 0.5900 Val Loss: 1.0530 Acc: 0.6750
Epoch 3/10 Train Loss: 1.1062 Acc: 0.6100 Val Loss: 1.0752 Acc: 0.6950
Epoch 4/10 Train Loss: 1.0899 Acc: 0.6100 Val Loss: 0.9495 Acc: 0.6850
Epoch 5/10 Train Loss: 1.0581 Acc: 0.6238 Val Loss: 0.9862 Acc: 0.6750
Epoch 6/10 Train Loss: 1.0128 Acc: 0.6362 Val Loss: 0.9558 Acc: 0.6850
Epoch 7/10 Train Loss: 0.9748 Acc: 0.6450 Val Loss: 0.9207 Acc: 0.6950
Epoch 8/10 Train Loss: 0.9074 Acc: 0.6663 Val Loss: 0.9174 Acc: 0.7000
Epoch 9/10 Train Loss: 0.8535 Acc: 0.6787 Val Loss: 0.9059 Acc: 0.6850
Epoch 10/10 Train Loss: 0.8386 Acc: 0.6925 Val Loss: 0.9174 Acc: 0.6900
```

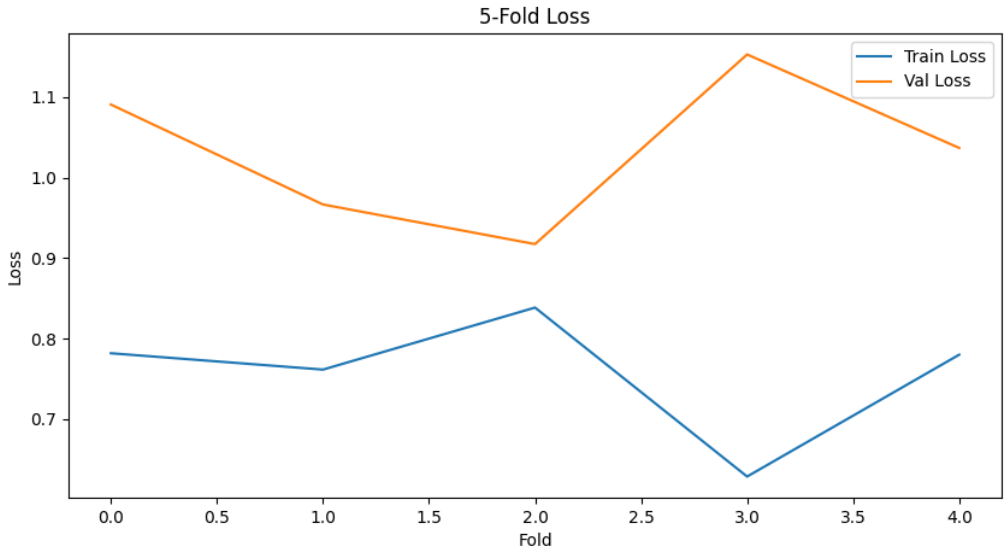
```
--- Fold 4 ---
Epoch 1/10 Train Loss: 1.5143 Acc: 0.5400 Val Loss: 1.5885 Acc: 0.1450
Epoch 2/10 Train Loss: 1.1070 Acc: 0.6075 Val Loss: 1.1885 Acc: 0.5950
Epoch 3/10 Train Loss: 0.9935 Acc: 0.6488 Val Loss: 1.1884 Acc: 0.6100
Epoch 4/10 Train Loss: 0.9541 Acc: 0.6538 Val Loss: 1.1391 Acc: 0.6050
Epoch 5/10 Train Loss: 0.8878 Acc: 0.6663 Val Loss: 1.1581 Acc: 0.6150
Epoch 6/10 Train Loss: 0.8920 Acc: 0.6737 Val Loss: 1.1858 Acc: 0.5950
Epoch 7/10 Train Loss: 0.8647 Acc: 0.6625 Val Loss: 1.2244 Acc: 0.5800
Epoch 8/10 Train Loss: 0.7879 Acc: 0.7063 Val Loss: 1.1387 Acc: 0.6150
Epoch 9/10 Train Loss: 0.6800 Acc: 0.7375 Val Loss: 1.1529 Acc: 0.5750
Epoch 10/10 Train Loss: 0.6287 Acc: 0.7688 Val Loss: 1.1530 Acc: 0.6000
Saved model for fold 4 at saved_models/model_fold4.pth

--- Fold 5 ---
Epoch 1/10 Train Loss: 1.7163 Acc: 0.5350 Val Loss: 1.3898 Acc: 0.5900
Epoch 2/10 Train Loss: 1.1115 Acc: 0.6262 Val Loss: 1.1402 Acc: 0.5900
Epoch 3/10 Train Loss: 1.0303 Acc: 0.6275 Val Loss: 1.1936 Acc: 0.5900
Epoch 4/10 Train Loss: 1.0306 Acc: 0.6312 Val Loss: 1.2020 Acc: 0.5900
Epoch 5/10 Train Loss: 0.9527 Acc: 0.6538 Val Loss: 1.0948 Acc: 0.5950
Epoch 6/10 Train Loss: 0.9032 Acc: 0.6650 Val Loss: 1.0624 Acc: 0.6250
Epoch 7/10 Train Loss: 0.8450 Acc: 0.6600 Val Loss: 1.0985 Acc: 0.5950
Epoch 8/10 Train Loss: 0.8088 Acc: 0.6863 Val Loss: 1.0451 Acc: 0.6050
Epoch 9/10 Train Loss: 0.8076 Acc: 0.6950 Val Loss: 1.1162 Acc: 0.5750
Epoch 10/10 Train Loss: 0.7800 Acc: 0.6887 Val Loss: 1.0369 Acc: 0.6200
Saved model for fold 5 at saved_models/model_fold5.pth
entering testing

Ensemble Test Accuracy: 0.2830
```

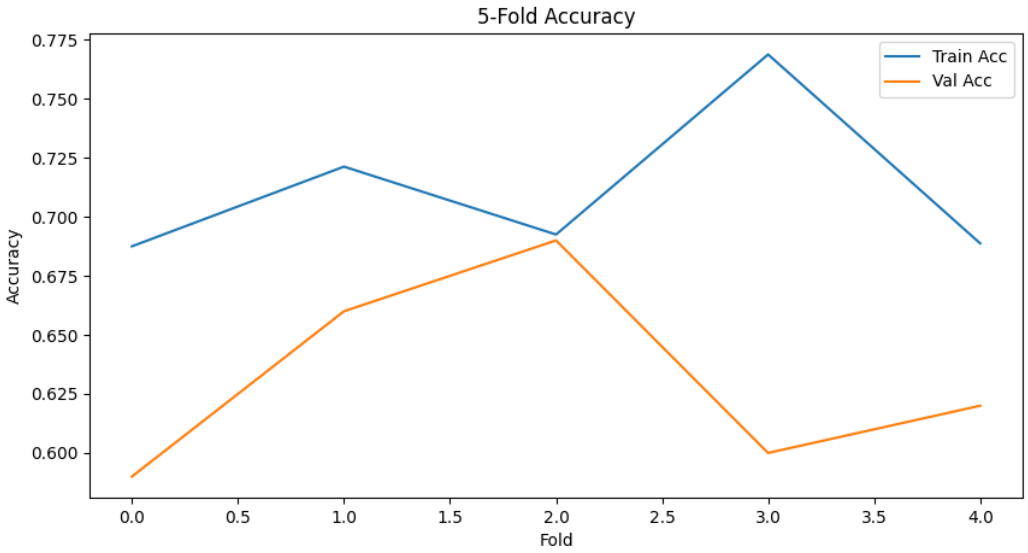
note that my test set is a balanced one so its not a fair metric, and validation would be a more accurate way to aprox performance as its more "real" in the terms that its as unbalanced as the real set.

As for the results we have:

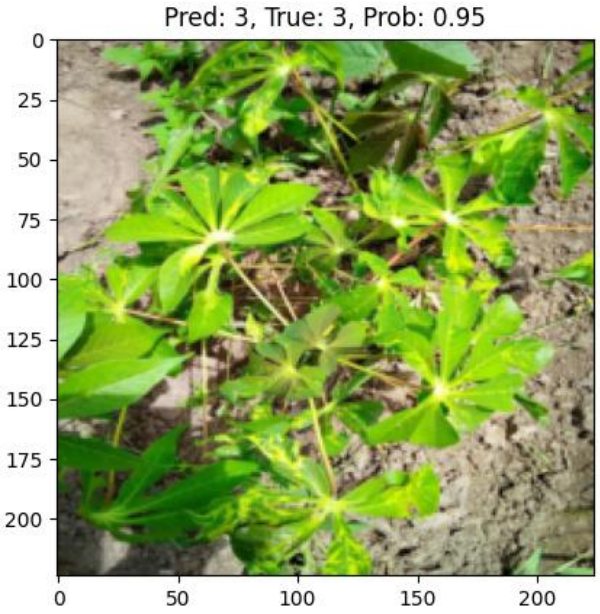


as we can see, the model is very biased towards class 3, and can somewhat guess other classes but not as reliably.
as for the graphs:

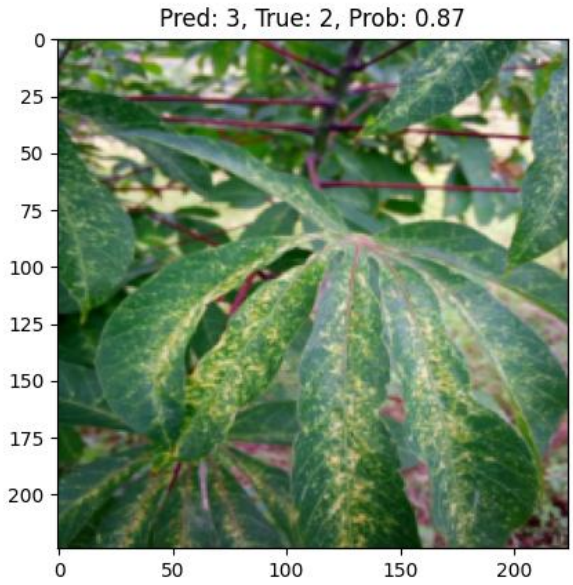




we do see paterns of memorization of the training set, but suprisingly the balanced test score is higher than the scores for the less overfitting results which happened in my expirements(more on that later).



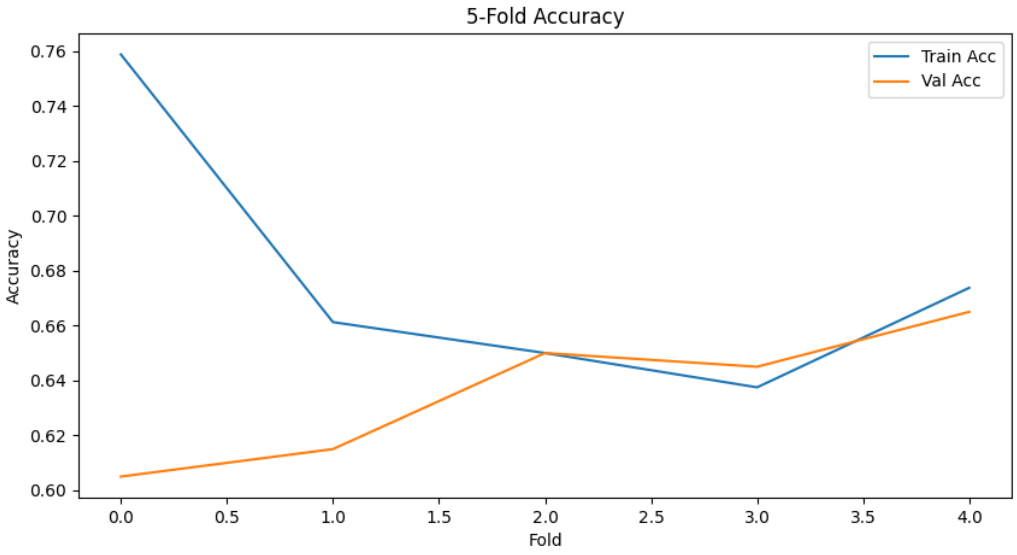
we can see that our model is very confident about class 3.

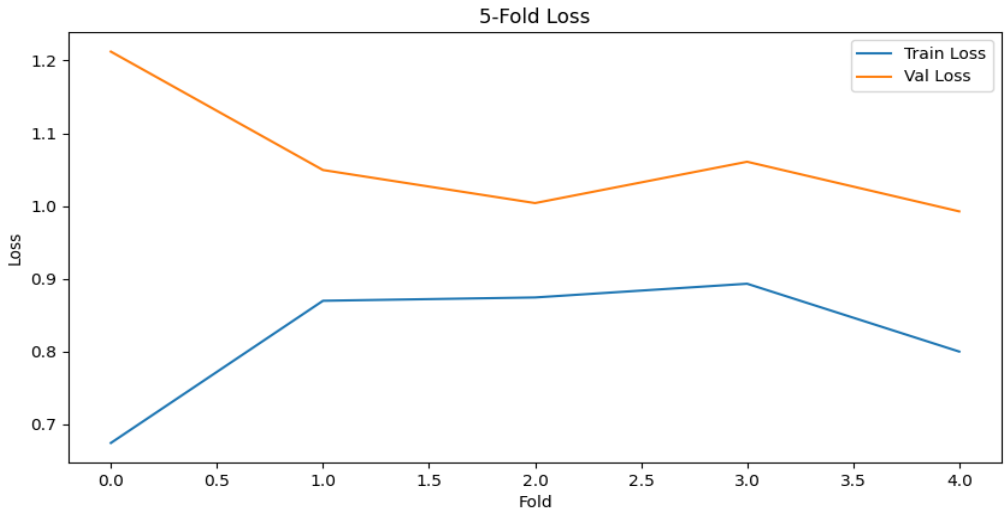


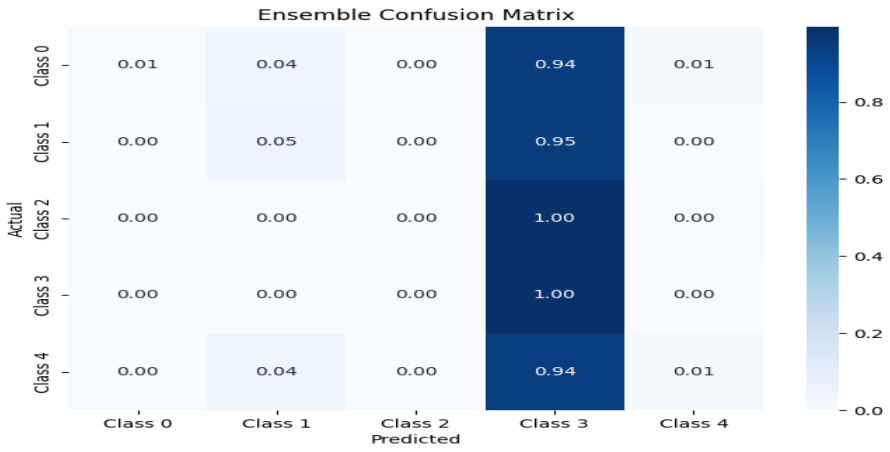
and also very confident that everything else is class 3 😊 , and its mainly due to the ungodly biased dataset with 63%+ being just class 3.

Now to the experiments ive performed to understand hyper parameters better, don’t worry it wont affect my answer to question B since there are 2 main problems that overweight every other possible problem, which are unholy unbalance, and tiny data set(I've used only 1K to save time which is a huge problem for generalization).

Experiments: batch size=64: Ensemble Test Accuracy: 0.2240

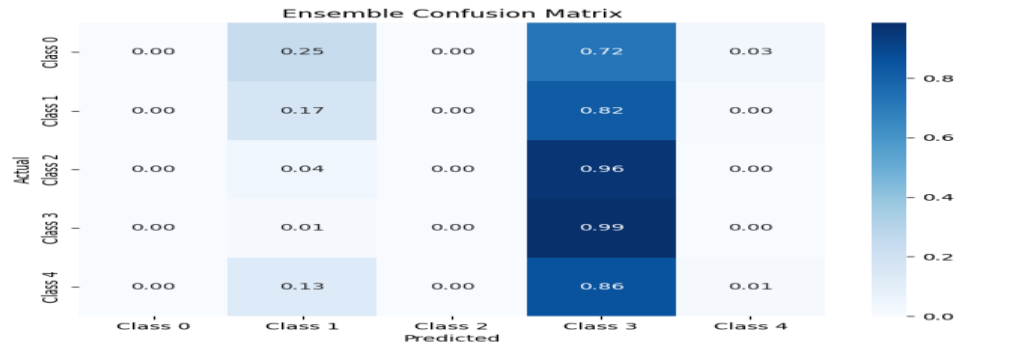


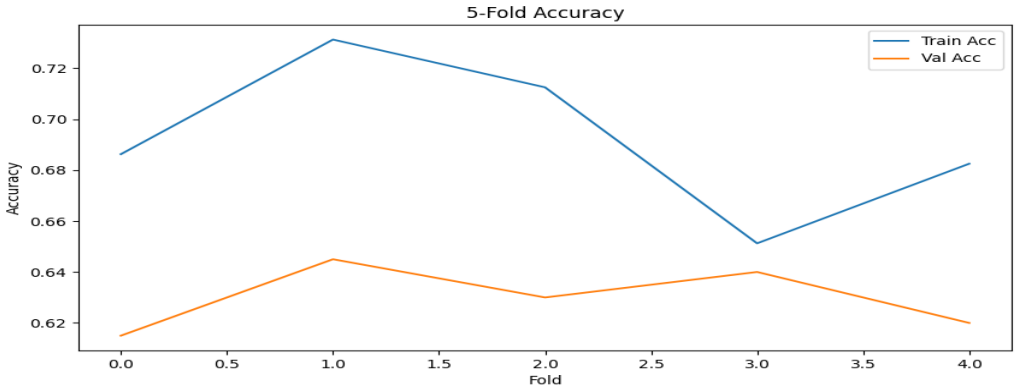


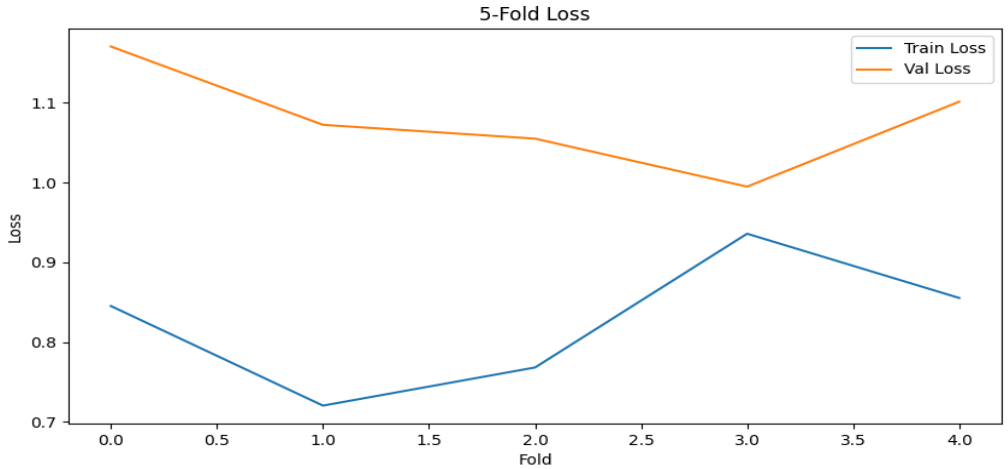


similar results, and it seems overall more balanced compared to the validation, but final test results show a huge bias problem, despite looking like less overfitting. Which is strange because I'd expect bigger batch-> easier memory.

Batch size=96: Ensemble Test Accuracy: 0.2460



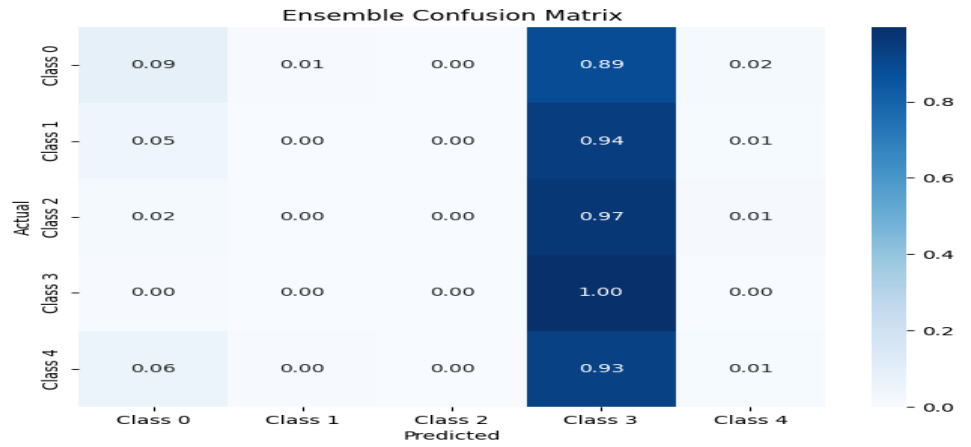


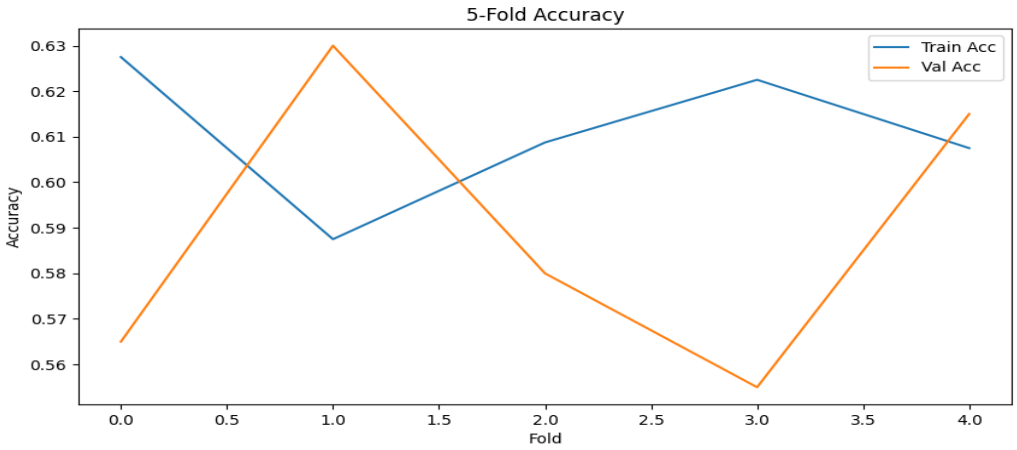


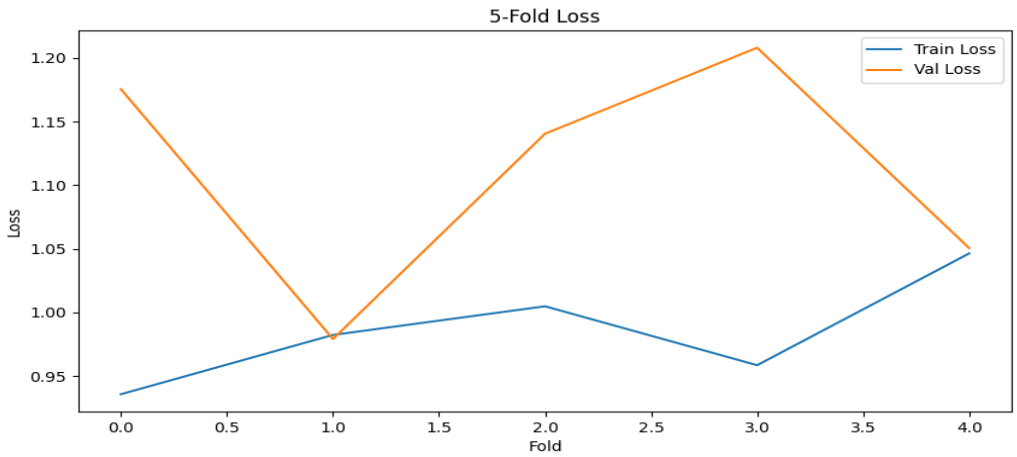
we do see a bit more defined overfitting patern, but not any real difference, there is a chance I got lucky with my original train set to have more difficult and diverse samples for some classes, or perhaps did share some of the set with the test set as my test set is not really foreign to the full training set as the split happened later. My test set is more like general balanced test and is not a good metric for the biased training set, but is a metric for overall model performance as im not realy sure about the leaf diseas distribution in the real world so general=better for me.

Tried training again with the 32 batch size, again got more overfitting pattern but a test score of 0.275 which is better than the last 2 expirements.

Learning rate=5E-4: Ensemble Test Accuracy: 0.2250







over all the trainng was slower and way less eficient, and while it was still quite stable, with only tiny spikes that were quickly corrected, it still did perform very poorly, I'd guess its because its learned more trash in the beginning and became a class 3 fanboy way too quickly.

Conclusion: smaller unbalanced and biased datasets prefer slower learning.

b) As I've said before, my data is so unbalanced that just guessing class 3 all the time will yield great results in training and validation, and as we did see in the confusion matrix(especially in the experiments) it was the main way the model decided to operate. Regarding complexity, I was able to overfit the model on a smaller dataset, and while the data is very noisy and inconsistent, there is not that much data and the number of classes is fairly small, meaning that the model is somewhat fit for the data.

Fixes:

1) Balance the data, delete 10k+ of class 3 images and maybe we will be alright
2) Train on more images to avoid overfitting, as 1k images is quite a small dataset that I chose because more images on biased data would most likely still yield the same results.
3) Add augmentation to avoid overfitting, and while adding noise is bad, no one prevents us from rotating our images and mirroring them
4) Increase resolution, quite self-explanatory as we do lose most of our data when we squeeze 800x600 into 224x244, and we deprive our poor model from data that way.
5) Add more epochs, useless now as our model refuses to properly train, and we risk just overfitting, but for when our model will be able to properly train on normal data, it will probably not learn fast enough to finish in 10 epochs.

c) I obviously would choose to balance the data and increase the training set, the biased data prevents our model from actually learning, and the 1k set I train on is very small not allowing to fully generalize and mature.

I will run the code in balancer.py to get a balanced set in balanced.scv, I will be taking only 1400 images from each class to not deprive class 0 too much, as if I take 2.5k, class 0 with its 1k images will not get enough representation.

After that run balanced_test_train_split.py to get a test and train sets over the balanced data, it's the same test set I was using in task a.

The second improvement I will do is to increase the training set size to 5k as our model simply didn't see enough data to properly and fully train.

I also will be increasing my batch size to 96 to train it faster, and because the data is not quite balanced, I don't need to be as afraid from bias and overfitting so a smaller batch size isn't as important anymore.

hyper parameters:

NUM_CLASSES = 5

INPUT_SIZE = 224

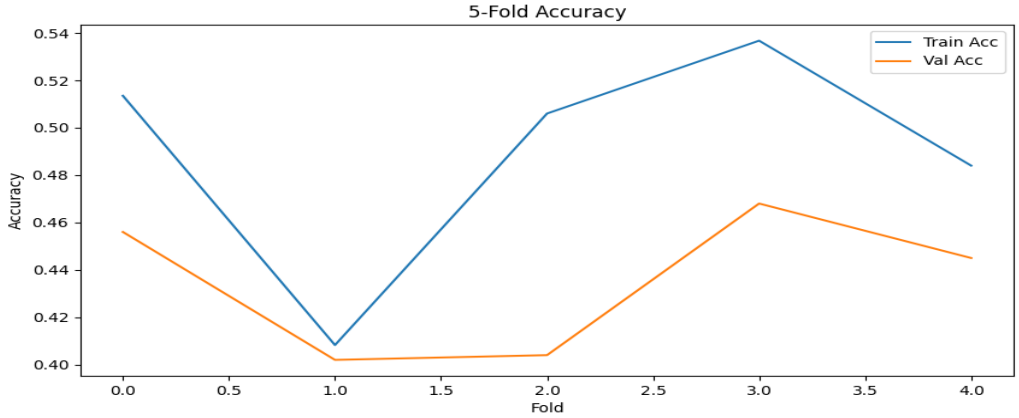
BATCH_SIZE = 96

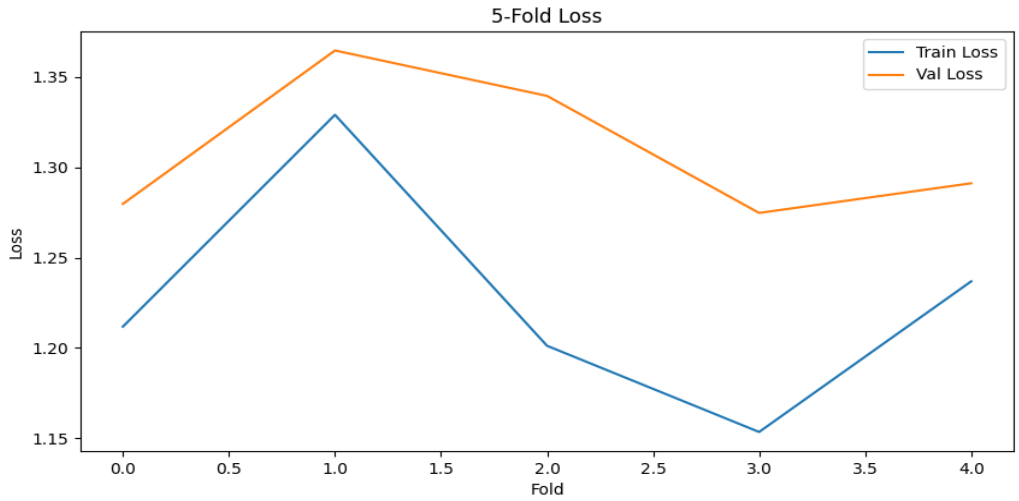
NUM_EPOCHS = 10

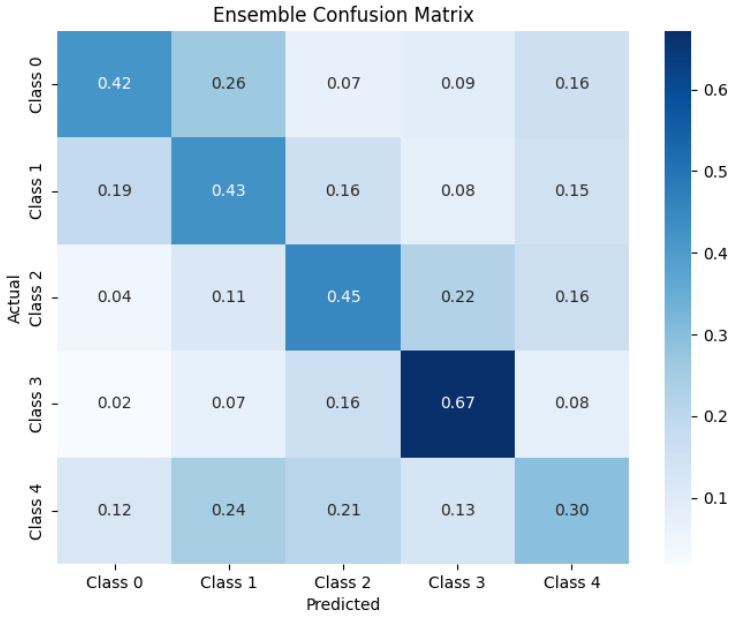
LEARNING_RATE = 5e-5

K_FOLDS = 5

DROPOUT_PROB = 0.3
TRAIN_SIZE = 5000
We get:







```
--- Fold 1 ---
Epoch 1/10 Train Loss: 1.8251 Acc: 0.2480 Val Loss: 1.5613 Acc: 0.2830
Epoch 2/10 Train Loss: 1.5225 Acc: 0.3157 Val Loss: 1.4941 Acc: 0.3120
Epoch 3/10 Train Loss: 1.4736 Acc: 0.3460 Val Loss: 1.4496 Acc: 0.3710
Epoch 4/10 Train Loss: 1.4264 Acc: 0.3728 Val Loss: 1.4021 Acc: 0.4070
Epoch 5/10 Train Loss: 1.3763 Acc: 0.4153 Val Loss: 1.3905 Acc: 0.3930
Epoch 6/10 Train Loss: 1.3454 Acc: 0.4355 Val Loss: 1.3469 Acc: 0.4220
Epoch 7/10 Train Loss: 1.3157 Acc: 0.4485 Val Loss: 1.3760 Acc: 0.3950
Epoch 8/10 Train Loss: 1.2710 Acc: 0.4785 Val Loss: 1.3260 Acc: 0.4250
Epoch 9/10 Train Loss: 1.2684 Acc: 0.4788 Val Loss: 1.3306 Acc: 0.4280
Epoch 10/10 Train Loss: 1.2118 Acc: 0.5135 Val Loss: 1.2797 Acc: 0.4560
Saved model for fold 1 at saved_models/model_fold1.pth

--- Fold 2 ---
Epoch 1/10 Train Loss: 1.8361 Acc: 0.2260 Val Loss: 1.6048 Acc: 0.2380
Epoch 2/10 Train Loss: 1.5511 Acc: 0.2855 Val Loss: 1.5406 Acc: 0.2750
Epoch 3/10 Train Loss: 1.5224 Acc: 0.3045 Val Loss: 1.4998 Acc: 0.3370
Epoch 4/10 Train Loss: 1.4821 Acc: 0.3277 Val Loss: 1.4643 Acc: 0.3650
Epoch 5/10 Train Loss: 1.4401 Acc: 0.3575 Val Loss: 1.4312 Acc: 0.3790
Epoch 6/10 Train Loss: 1.4150 Acc: 0.3670 Val Loss: 1.4268 Acc: 0.3620
Epoch 7/10 Train Loss: 1.3886 Acc: 0.3772 Val Loss: 1.3911 Acc: 0.4160
Epoch 8/10 Train Loss: 1.3674 Acc: 0.3882 Val Loss: 1.3747 Acc: 0.3900
Epoch 9/10 Train Loss: 1.3465 Acc: 0.3955 Val Loss: 1.3751 Acc: 0.3960
Epoch 10/10 Train Loss: 1.3290 Acc: 0.4083 Val Loss: 1.3646 Acc: 0.4020
Saved model for fold 2 at saved_models/model_fold2.pth

--- Fold 3 ---
Epoch 1/10 Train Loss: 1.8734 Acc: 0.2587 Val Loss: 1.5764 Acc: 0.2220
Epoch 2/10 Train Loss: 1.4904 Acc: 0.3438 Val Loss: 1.4862 Acc: 0.3190
Epoch 3/10 Train Loss: 1.4347 Acc: 0.3625 Val Loss: 1.4277 Acc: 0.3970
Epoch 4/10 Train Loss: 1.3807 Acc: 0.3967 Val Loss: 1.4237 Acc: 0.3760
Epoch 5/10 Train Loss: 1.3420 Acc: 0.4283 Val Loss: 1.3710 Acc: 0.4110
Epoch 6/10 Train Loss: 1.3169 Acc: 0.4410 Val Loss: 1.3524 Acc: 0.4150
Epoch 7/10 Train Loss: 1.2775 Acc: 0.4677 Val Loss: 1.3499 Acc: 0.4130
Epoch 8/10 Train Loss: 1.2517 Acc: 0.4738 Val Loss: 1.3223 Acc: 0.4310
Epoch 9/10 Train Loss: 1.2187 Acc: 0.5010 Val Loss: 1.3325 Acc: 0.4240
Epoch 10/10 Train Loss: 1.2012 Acc: 0.5060 Val Loss: 1.3394 Acc: 0.4040
--- Fold 4 ---
Epoch 1/10 Train Loss: 1.7387 Acc: 0.2695 Val Loss: 1.5905 Acc: 0.2610
Epoch 2/10 Train Loss: 1.4851 Acc: 0.3332 Val Loss: 1.4900 Acc: 0.3430
Epoch 3/10 Train Loss: 1.4321 Acc: 0.3785 Val Loss: 1.4199 Acc: 0.3700
Epoch 4/10 Train Loss: 1.3823 Acc: 0.4085 Val Loss: 1.4199 Acc: 0.3980
Epoch 5/10 Train Loss: 1.3198 Acc: 0.4460 Val Loss: 1.3804 Acc: 0.4020
Epoch 6/10 Train Loss: 1.2855 Acc: 0.4612 Val Loss: 1.3381 Acc: 0.4510
Epoch 7/10 Train Loss: 1.2425 Acc: 0.4818 Val Loss: 1.3334 Acc: 0.4360
Epoch 8/10 Train Loss: 1.2196 Acc: 0.4910 Val Loss: 1.3717 Acc: 0.4290
Epoch 9/10 Train Loss: 1.1855 Acc: 0.5138 Val Loss: 1.3024 Acc: 0.4600
Epoch 10/10 Train Loss: 1.1536 Acc: 0.5367 Val Loss: 1.2747 Acc: 0.4680
Saved model for fold 4 at saved_models/model_fold4.pth

--- Fold 5 ---
Epoch 1/10 Train Loss: 1.7417 Acc: 0.2492 Val Loss: 1.6112 Acc: 0.2030
Epoch 2/10 Train Loss: 1.5256 Acc: 0.3105 Val Loss: 1.4825 Acc: 0.3390
Epoch 3/10 Train Loss: 1.4707 Acc: 0.3530 Val Loss: 1.4475 Acc: 0.3540
Epoch 4/10 Train Loss: 1.4284 Acc: 0.3752 Val Loss: 1.4049 Acc: 0.4090
Epoch 5/10 Train Loss: 1.3767 Acc: 0.4052 Val Loss: 1.3962 Acc: 0.3920
Epoch 6/10 Train Loss: 1.3446 Acc: 0.4248 Val Loss: 1.3546 Acc: 0.4440
Epoch 7/10 Train Loss: 1.3256 Acc: 0.4445 Val Loss: 1.3476 Acc: 0.4320
Epoch 8/10 Train Loss: 1.2870 Acc: 0.4612 Val Loss: 1.3384 Acc: 0.4240
Epoch 9/10 Train Loss: 1.2542 Acc: 0.4803 Val Loss: 1.3302 Acc: 0.4390
Epoch 10/10 Train Loss: 1.2369 Acc: 0.4840 Val Loss: 1.2911 Acc: 0.4450
Saved model for fold 5 at saved_models/model_fold5.pth
entering testing

Ensemble Test Accuracy: 0.4550
```

Overall we see a very noticeable improvement, with our model finally being able to learn, it is a bit strange that it still aces class 3 but who am I to judge it, our diagonal is way better and we see that our model doesn't realy predicts 3 to anything other than class 3 and a bit of class 4 and class 2. We do see instabilities in the validations of some folds toward the later epochs, and turns out there is a StratifiedKFold that balanced train and val classes, while I was using a normal kfold that may explain part of the instability.

Possible reasons for misclassification: we do see noticeable improvements all the way down, suggesting that there is room for improvements and our models just don't have enough time to fully train, and we may be able to use a higher learning rate but to find that out I will have to run some expirements.

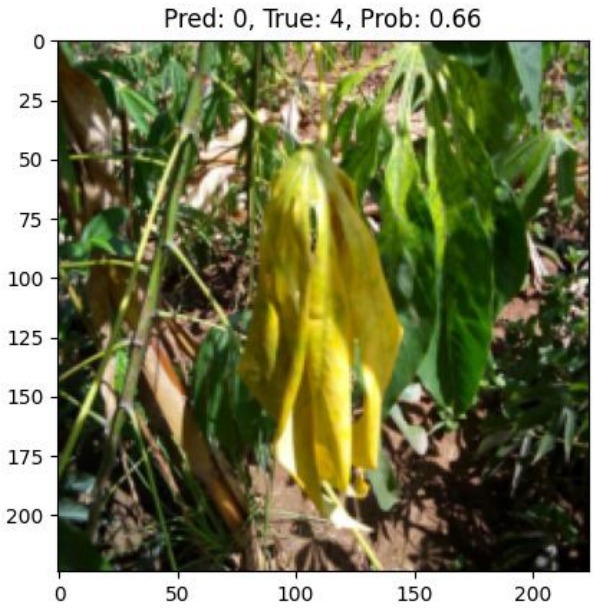
While our data is finally balanced, the quality of the data didn't really change, with the images being inconsistent and incoherent in what they show, being naturally very noisy and so on.

Our images are still very compressed, losing most of their data, and the training set, while bigger at 5K images, is still somewhat small.

Improvements:

1) Increase number of epochs, our model wasn't able to converge yet and was steadily learning, meaning it need more time.
2) Increase image resolution, 224x224 loses almost all the data(90%), which is crucial for such noisy and complex data.
3) Fine tune learning rate, we do see steady and noticeable improvements, but there may be a more optimal learning rate which we are yet to find. Or perhaps we need to add a scheduler to allow more flexible learning.
4) We don't yet see a real overfitting, with training and validation scores growing together, and being quite close to the test score, but a larger train set probably won't hurt.
5) I don't know how much it counts, but finding better data is probably very important as sometimes I'm unable to classify or even differentiate those classes apart, especially on the less obvious samples. Just take a look at the

less confident prediction:

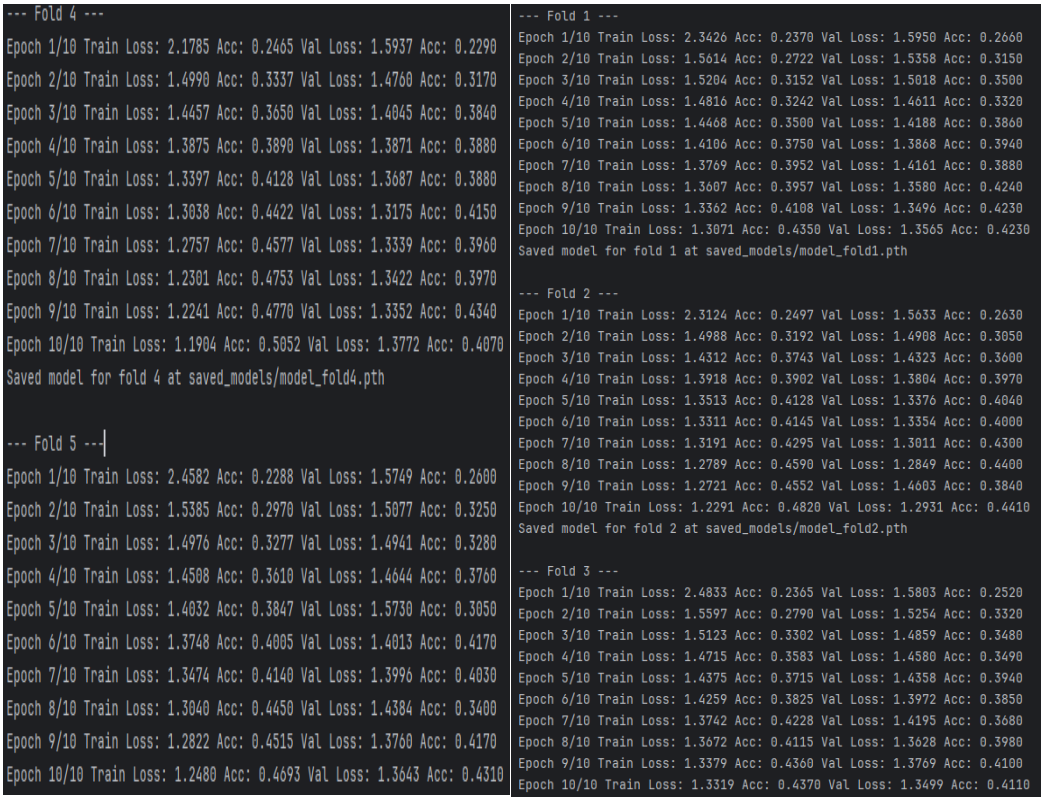


If you remember my random samples in task 1, the right most image of class 0 looks exactly like this class 4 image, so I doubt a human could predict it properly, let alone a small ai.

You can find the saved models in the 2C_models folder

Experiments: since kfold vs StratifiedKFold only really affects validation, I won't be changing it as it will hinder the isolation part of experiments.
Observation: looking at my hardware utilization, and my poor gpu is bottlenecked by the cpu trying to feed it new batches, my poor cpu just can't keep up(although its only at 50% utilization so it's not that bad). My batch size of 96 saturates around 12GB of my vram, so I have 4 GB free so I will probably increase batch size to 128. But my gpu is just too fast for the small batch size. Also my gpu is jumping between 15% and 40% utilization, and the main drop appears right before the epoch results are printed, its most likely caused by things like:
outputs_all.append(F.softmax(outputs, dim=1).cpu())
labels_all.append(lbls.cpu())
which mean the cpu requests data back from the gpu and running more computations, giving the gpu more idle time.

Increasing learning rate to 1E-4: Ensemble Test Accuracy: 0.4430
while the graphs look quite similar, a strange thing I've noticed is that with x2 the learning rate, we started to learn more slowly, validation scores were quite unstable, and learning was slower overall, but with all that our test accuracy is still very similar. What's even more interesting is how inconsistent the training was, with sometimes it being perfectly stable, slowly getting better and in the last few epochs take a huge leap forward. Sometimes it was unstable, fluctuating a lot, and sometimes it was perfectly and stably learning in a good rate. I suppose we sometimes were just overshooting the minimal points.



```
--- Fold 4 ---
Epoch 1/10 Train Loss: 2.1785 Acc: 0.2465 Val Loss: 1.5937 Acc: 0.2290
Epoch 2/10 Train Loss: 1.4990 Acc: 0.3337 Val Loss: 1.4760 Acc: 0.3170
Epoch 3/10 Train Loss: 1.4457 Acc: 0.3650 Val Loss: 1.4045 Acc: 0.3840
Epoch 4/10 Train Loss: 1.3875 Acc: 0.3890 Val Loss: 1.3871 Acc: 0.3880
Epoch 5/10 Train Loss: 1.3397 Acc: 0.4128 Val Loss: 1.3687 Acc: 0.3880
Epoch 6/10 Train Loss: 1.3038 Acc: 0.4422 Val Loss: 1.3175 Acc: 0.4150
Epoch 7/10 Train Loss: 1.2757 Acc: 0.4577 Val Loss: 1.3339 Acc: 0.3960
Epoch 8/10 Train Loss: 1.2301 Acc: 0.4753 Val Loss: 1.3422 Acc: 0.3970
Epoch 9/10 Train Loss: 1.2241 Acc: 0.4770 Val Loss: 1.3352 Acc: 0.4340
Epoch 10/10 Train Loss: 1.1904 Acc: 0.5052 Val Loss: 1.3772 Acc: 0.4070
Saved model for fold 4 at saved_models/model_fold4.pth

--- Fold 5 ---
Epoch 1/10 Train Loss: 2.4582 Acc: 0.2288 Val Loss: 1.5749 Acc: 0.2600
Epoch 2/10 Train Loss: 1.5385 Acc: 0.2970 Val Loss: 1.5077 Acc: 0.3250
Epoch 3/10 Train Loss: 1.4976 Acc: 0.3277 Val Loss: 1.4941 Acc: 0.3280
Epoch 4/10 Train Loss: 1.4508 Acc: 0.3610 Val Loss: 1.4644 Acc: 0.3760
Epoch 5/10 Train Loss: 1.4032 Acc: 0.3847 Val Loss: 1.5730 Acc: 0.3050
Epoch 6/10 Train Loss: 1.3748 Acc: 0.4005 Val Loss: 1.4013 Acc: 0.4170
Epoch 7/10 Train Loss: 1.3474 Acc: 0.4140 Val Loss: 1.3996 Acc: 0.4030
Epoch 8/10 Train Loss: 1.3040 Acc: 0.4450 Val Loss: 1.4384 Acc: 0.3400
Epoch 9/10 Train Loss: 1.2822 Acc: 0.4515 Val Loss: 1.3760 Acc: 0.4170
Epoch 10/10 Train Loss: 1.2480 Acc: 0.4693 Val Loss: 1.3643 Acc: 0.4310
```

```
--- Fold 1 ---
Epoch 1/10 Train Loss: 2.3426 Acc: 0.2370 Val Loss: 1.5950 Acc: 0.2660
Epoch 2/10 Train Loss: 1.5614 Acc: 0.2722 Val Loss: 1.5358 Acc: 0.3150
Epoch 3/10 Train Loss: 1.5204 Acc: 0.3152 Val Loss: 1.5018 Acc: 0.3500
Epoch 4/10 Train Loss: 1.4816 Acc: 0.3242 Val Loss: 1.4611 Acc: 0.3320
Epoch 5/10 Train Loss: 1.4468 Acc: 0.3500 Val Loss: 1.4188 Acc: 0.3860
Epoch 6/10 Train Loss: 1.4106 Acc: 0.3750 Val Loss: 1.3868 Acc: 0.3940
Epoch 7/10 Train Loss: 1.3769 Acc: 0.3952 Val Loss: 1.4161 Acc: 0.3880
Epoch 8/10 Train Loss: 1.3607 Acc: 0.3957 Val Loss: 1.3580 Acc: 0.4240
Epoch 9/10 Train Loss: 1.3362 Acc: 0.4108 Val Loss: 1.3496 Acc: 0.4230
Epoch 10/10 Train Loss: 1.3071 Acc: 0.4350 Val Loss: 1.3565 Acc: 0.4230
Saved model for fold 1 at saved_models/model_fold1.pth

--- Fold 2 ---
Epoch 1/10 Train Loss: 2.3124 Acc: 0.2497 Val Loss: 1.5633 Acc: 0.2630
Epoch 2/10 Train Loss: 1.4988 Acc: 0.3192 Val Loss: 1.4908 Acc: 0.3050
Epoch 3/10 Train Loss: 1.4312 Acc: 0.3743 Val Loss: 1.4323 Acc: 0.3600
Epoch 4/10 Train Loss: 1.3918 Acc: 0.3902 Val Loss: 1.3804 Acc: 0.3970
Epoch 5/10 Train Loss: 1.3513 Acc: 0.4128 Val Loss: 1.3376 Acc: 0.4040
Epoch 6/10 Train Loss: 1.3311 Acc: 0.4145 Val Loss: 1.3354 Acc: 0.4000
Epoch 7/10 Train Loss: 1.3191 Acc: 0.4295 Val Loss: 1.3011 Acc: 0.4300
Epoch 8/10 Train Loss: 1.2789 Acc: 0.4590 Val Loss: 1.2849 Acc: 0.4400
Epoch 9/10 Train Loss: 1.2721 Acc: 0.4552 Val Loss: 1.4603 Acc: 0.3840
Epoch 10/10 Train Loss: 1.2291 Acc: 0.4820 Val Loss: 1.2931 Acc: 0.4410
Saved model for fold 2 at saved_models/model_fold2.pth

--- Fold 3 ---
Epoch 1/10 Train Loss: 2.4833 Acc: 0.2365 Val Loss: 1.5803 Acc: 0.2520
Epoch 2/10 Train Loss: 1.5597 Acc: 0.2790 Val Loss: 1.5254 Acc: 0.3320
Epoch 3/10 Train Loss: 1.5123 Acc: 0.3302 Val Loss: 1.4859 Acc: 0.3480
Epoch 4/10 Train Loss: 1.4715 Acc: 0.3583 Val Loss: 1.4580 Acc: 0.3490
Epoch 5/10 Train Loss: 1.4375 Acc: 0.3715 Val Loss: 1.4358 Acc: 0.3940
Epoch 6/10 Train Loss: 1.4259 Acc: 0.3825 Val Loss: 1.3972 Acc: 0.3850
Epoch 7/10 Train Loss: 1.3742 Acc: 0.4228 Val Loss: 1.4195 Acc: 0.3680
Epoch 8/10 Train Loss: 1.3672 Acc: 0.4115 Val Loss: 1.3628 Acc: 0.3980
Epoch 9/10 Train Loss: 1.3379 Acc: 0.4360 Val Loss: 1.3769 Acc: 0.4100
Epoch 10/10 Train Loss: 1.3319 Acc: 0.4370 Val Loss: 1.3499 Acc: 0.4110
```

<u>Adding a scheduler:</u> I will also be lowering training set to 2k as it takes too much time.

```
scheduler = OneCycleLR(
   optimizer,
   max_lr=1e-4,  # peak LR
   steps_per_epoch=steps_per_epoch,
   epochs=NUM_EPOCHS,
   pct_start=0.3,  # fraction of steps to reach peak LR
   anneal_strategy='cos',  # cosine annealing down
)
```

well from a balanced model we just went to a fast overfiter, no point in showing any metrics, train acc and loss steadly and quite quickly go down, but val metrics are unstable and remain low the entire time, which is classic overfitting.

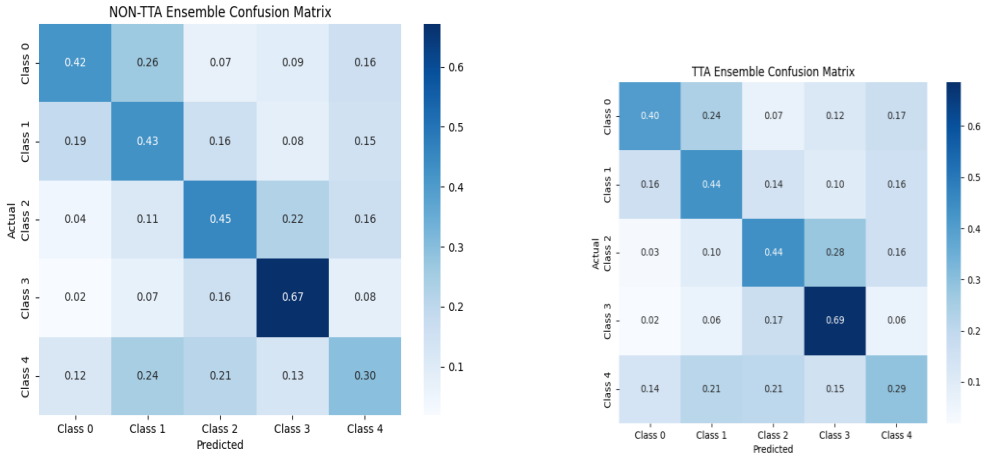
Epoch 10/10 Train Loss: 1.1975 Acc: 0.5325 Val Loss: 1.3727 Acc: 0.3700
Epoch 10/10 Train Loss: 1.1826 Acc: 0.5394 Val Loss: 1.3843 Acc: 0.3950
Epoch 10/10 Train Loss: 1.1585 Acc: 0.5494 Val Loss: 1.4335 Acc: 0.3750
Epoch 10/10 Train Loss: 1.2056 Acc: 0.5275 Val Loss: 1.3428 Acc: 0.4350
Ensemble Test Accuracy: 0.4160, not that bad afterall

I will try to increase dropout rate and lower batch size to mitigate this problem, but since I'm time constricted, I will stop there.

Batch size 96->32, dropout 0.3->0.5

We already see much better results, with training going hand to hand with validation. But now training speed is once again very small.
Epoch 10/10 Train Loss: 1.3887 Acc: 0.3631 Val Loss: 1.4383 Acc: 0.3575
I will let it run 1 more fold, if it's the same, I'll reduce dropout.
Epoch 10/10 Train Loss: 1.3949 Acc: 0.3613 Val Loss: 1.4211 Acc: 0.4000
still too slow, ill decrease dropout to 0.4 and increase max learning rate to 2e-4:
While still perfectly fitting, we do get more instability and remain very low, sadly I don't have enough time to run more experiments and fine tune everything, so ill have to stop here and continue with the work, maybe ill run more experiments on my own.
Epoch 10/10 Train Loss: 1.3156 Acc: 0.3881 Val Loss: 1.3805 Acc: 0.3775
Epoch 10/10 Train Loss: 1.3679 Acc: 0.3663 Val Loss: 1.3822 Acc: 0.3525
I honestly unsure why does adding a scheduler hinder our progress so much, my best guess is that the data is so bad, that the scheduler needs a lot of fine tuning to be able to steadly learn things, and I simply don't have enough time to find those tunes, nor do I have the resources to clean the data.

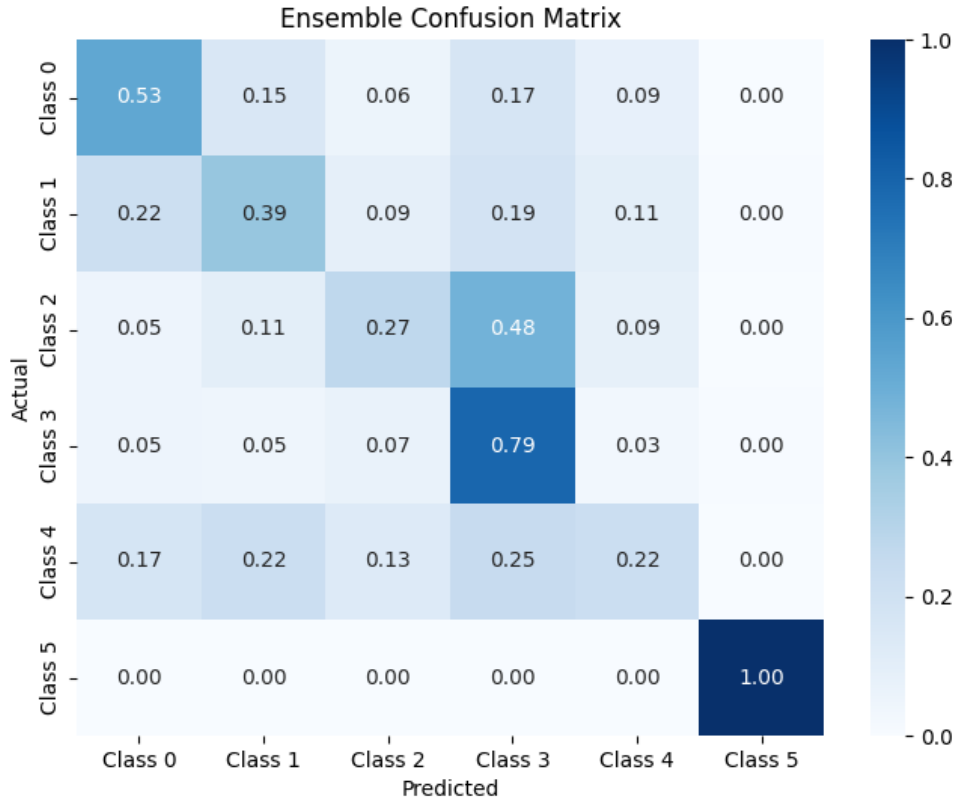
d) The code is in: task2_model_tester_d.py.
We will use the classical augmentation of flipping.
Ensemble Accuracy without TTA: 0.4550
TTA Ensemble Accuracy: 0.4540
Accuracy improvement with TTA: -0.0010
we got a tiny bit worse, but not really.



Looking at them side by side, it appears that thanks to the data being so chaotic, rotating the images made our model confuse class 3 and 2 more, but overall the changes are minimal.

e) I will be using the data set in https://www.kaggle.com/competitions/classify-leaves/data, as I couldn't find the same domain as the leaf diseas, but its still a plant.

Because the data in the csv had different labels, i.e image instead of image_id, and all images were of form: image/image_name, I had to manually change the labels and the data to allow easier integration. I've also dumped all of the new images to my normal train_images directory for again easier integration.
To extract the data, I ran the code in TASK2_E_extract_new_class.py, after analyzing the data, I chose at random a class with 100+ samples, I will reclassify it as label 5. Then split into train and test data for the new label.
Then we will run the code in TASK2_E_train_with_new_class.py, where we will load our training data as normal, and add this new data along with it.
I will also be using batch size 128 to make the training faster. We do run more of an overfit risk, but I doubt this part really cares about performance, and I'm more than sure this part comes to teach us how to add and deal with new data.
interesting observation: with my old 56M model, increasing batch size from 96 to 128 added an extra of around 4 GB if not more, now the same increase added around 1.4GB of vram. Conclusion: most of the vram is used by **activations** during forward/backward passes. Which scale linearly with model size as bigger model=more weights and more activations per each image.



Ensemble Test Accuracy: 0.4553.
Explanation: the new data is so different from our normal leaves, that it can easily be taught, or at least taught that its not the other classes.

Even in my overfitting sanity check, class 5 was a perfect 1.00 thanks to it being so unique.

Task 3 pretrained models:

a) Code is in task3_A.py

I will be training them all with

NUM_CLASSES = 5
INPUT_SIZE = 224
BATCH_SIZE = 256
NUM_EPOCHS = 10
LEARNING_RATE = 1e-3
TRAIN_SIZE = 6000

I wont be finetuning the hyper params for each model as its long, and will not be fair if I find a good tune for 1 and a bad tune for the other

Model Name	# parameters	Validation Loss	Validation Accuracy	Test Loss	Test Accuracy	# unique correct samples	# unique errors	Run time	Nores
Resnet18	11.17M	1.0976	0.5611	1.1359	0.5630	563	437	238.36 seconds	Great and stable
alexnet	62.37M	1.2056	0.5119	1.2017	0.5250	525	475	228.79 seconds	Quite unstable compared to resnet
mobilenet_v3_small	2.9M	1.2118	0.4996	1.2318	0.5050	505	495	228.62 seconds	Seems to be overfitting very fast and val seems to freeze at some point
EfficientnetB0	5.3M	1.0840	0.5743	1.1041	0.5730	573	427	256.02 seconds	The best stability I've seen until a small plunge in the last epoch

d) the code for the feature extracting is found in TASK3_D_feature_Extractor.py

since the runtimes don't differ that much anyway, I will be using efficientnet as my feature extractor because it shows the best result.

As an ML I will be using XGBClassifier because it can actually run on the gpu, I though about gradientboostingclassifier but it runs sequentily on a single core of the cpu, meaning a huge time waste.

attempt	n_estimators	Training Accuracy	Validation Accuracy	Test Accuracy	max_depth	learning_rate	runtime	confuse
1.	100	0.96	0.559	0.564	5	0.05	44.69 seconds	
2.	20	0.8	0.5	0.513	5	0.05	76.62 seconds	Confusion Matrix: [[85 28 10 15 25] [30 109 16 27 27] [11 22 90 49 37] [9 19 34 132 16] [27 41 25 19 97]]
3.	50	0.61	0.504	0.509	3	0.03	32.32 seconds	Confusion Matrix: [[90 22 12 11 28] [35 101 17 28 28] [9 29 87 54 30] [8 19 34 136 13] [28 43 22 21 95]]
4.	100	0.67	0.53	0.534	3	0.03	32.86 seconds	
5.	100	0.789	0.544	0.555	4	0.03	35.83 seconds	Confusion Matrix: [[94 25 11 10 23] [31 112 15 23 28] [10 24 102 40 33] [9 12 32 144 13] [25 41 25 15 103]]
6	125	0.765	0.538	0.534	4	0.02	34.59 seconds	Confusion Matrix: [[92 26 12 9 24] [30 108 17 23 31] [9 25 96 44 35] [10 12 36 139 13] [29 41 23 17 99]]
7	150	0.786	0.544	0.55	4	0.02	36.57 seconds	
8	175	0.804	0.550	0.561	4	0.02	36.14 seconds	[[94 23 12 10 24] [29 112 14 23 31] [10 22 108 36 33] [9 11 35 144 11] [28 38 25 15 103]]

As we can see, we don’t seem to be able to pass the 0.565 part, as we are already overfitting heavily, my best guess is that it happens because our data is so incredibly awful, inconsistent and noisy that a simple ML simply can't make much sense of it sometimes even if all the features are given.

Experiment: increasing the images sizes to 448x448 did yield us better results using the same parameters as 8. We got:

Training Accuracy: 0.827

Validation Accuracy: 0.608

Test Accuracy: 0.603

Runtime: 103.53 seconds

```
[[ 90  28   3  12  30]
 [ 25 112  21  17  34]
 [ 11  24 127  25  22]
 [  4  13  27 157   9]
 [ 27  38  19   8 117]]
```

And all of that improvement for just triple the time, ten times more utilization and I haven't checked electricity usage but after my undervolting, 100% uses around 140w, and normally 10% uses around 30w so id say 5 times more electricity.

So it seems like the best way forward is indeed increasing image resolution which I wont be doing due to hardware, time and wallet constraints.

e) I have no idea what experiments you are talking about, my main experiments happened in part d and the table is already present.

The only experiments I've done before that were increasing my train set to 6k and batch size to 256 and learning rate to 1e-3, after seeing that thanks to me only training the last layer, vram utilization was extremely low even for my 256 batch.

Although there was an observation, while all models used no more than 2 gb for the 256 batch, efficientNet used a whole 5 gb with all the same parameters, but non of those are really experiments.

CONCLUSIONS:

1) Smaller models sometimes can overfit faster than bigger models, requiring lower batch size and lower training rate to keep things safe and stable, this observation was made when comparing the result from my 13M model with the results of my 56M model that is found in backup.py.
2) Vram usage as a function of batch size scales linearly with model size, this happens because most of the vram is filled with activation data for all the images in the batch, meaning bigger model-> more activations-> more vram usage. This observation is even easier to spot for pretrained model where you reteach only 1 layer, making your vram consumption plum down, allowing you to use bigger batches and consequentially allows you to use more train data as bigger batches means less steps meaning faster training.
3) Pretrained models are less prone to overfitting than new models, as the backbone stays frozen preventing memorization, and the last layer requires more time to memorize. Allowing you to once again use bigger batches.
4) Bigger batch= more overfitting, this is caused because bigger batch=more stable gradient and precise steps allowing it to quickly descend to a sharp minima. Leading to a quick memorization and getting stuck.
5) Sometimes there is nothing you can do when your data is awful, and its even worse when you are constraint by hardware and time meaning you need smaller input size meaning even more of this awful data is lost.
6) Training ai locally allows you to directly monitor your hardware, allowing you to learn how each parameter and change affects hardware performance. For example, the relation between the cpu and gpu and bottle necking: for small models, the gpu is often undersaturated, meaning that it quickly performs its steps and have to wait for the cpu to send the next batch, this problem is often worsened by the fact that consumer grade gpus aren't that high on vram, thankfully I have 16 GB which proved to be not that much after all, and its even worse for smaller datasets that don't allow you to use bigger batches to avoid overfitting, meaning that your gpu most of the time idles waiting for the cpu. Vram is even more of the problem, since its usage scales linearly with model complexity, meaning that x4 parameter mean that x4 data per image have to be stored, meaning that we have even less batch sizes even when the gpu is far from being saturated further slowing down our training time.

7) Overfitting models sometimes perform ok, meaning that while they do memorize its training set, they do learn in the process some patterns, allowing them to sometimes perform better than not so much overfitting models.
8) Chat-gpt is not great in Deep learning related problem, and while can be used for code, often requires you to guide him toward the possible answer(like always), and may lie or confuse you and contradict itself. So the best way to learn remain fail and trial, which is kinda problematic as data scale often changes the rules, but you can't experiment on large models and data because it takes a lot of time.
9) Schedulers require fine tuning and sometimes are not the best idea for small data scales, as they often accelerate overfitting.
10) Not so much of a new conclusion, but garbage in garbage out, when you have data that you as a human find hard to differentiate let alone classify, squeezing the data and losing 90% of it in the process, will seriously limit how much you can train an ai on it.
11) Data imbalance= ai bias toward a dominating class, making the model refuse learning and instead opting for just guessing the dominant class always.
12) Obviously different data can survive overfitting and heavy bias, if a class is very different from the bias or all the other classes, even a biased or overfitted model can still easily classify it correctly.
13) K fold often works better than a single train model, but it requires k times training a model meaning it’s a huge time consumption, and not always worth it.

There are probably more conclusions that I don’t remember, or thinks that I think are obvious like that a confusion matrix is a great tool for assessing model strength and biases

I have no idea how a blogpost normally looks, and the few that I did see were fairly big so I assume this report is sufficiently compact, especially considering that half of the size is just images.