Assignment 2

Student Details:

Name: Nikshay Jain Roll No: MM21B044

WANDB Link:
https://api.wandb.ai/links/mm21b044-indian-institute-of-technology-madras/ekxt7nnq

Github Link:
https://github.com/Nikshay-Jain/DA6401-Assign-2

Report: DA6401-Assignment-2

CNNs - Coding and applications

Nikshay Prashant Jain mm21b044

Created on April 18 | Last edited on April 19

Part A: Training from scratch

I loaded the iNaturalist dataset in a Kaggle notebook for repetitive use, and the same is reflected in my Python code files as well. You may replace the directory link with the relevant one if you want to try the code out (make sure you replace it explicitly wherever it is mentioned).

Question 1

The model is built in the `CustomCNN` class in the file named `A_classes.py`, which when executed via the `A_main.py`, builds the model with the necessary architecture. The code written is completely flexible such that the number of filters, size of filters, and activation function of the convolution layers and dense layers can be changed. You can also change the number of neurons in the dense layer.

The **total number of computations** done by the model = **19,549,180,416**, as computed by the function `calculate_total_computations` based on the formula to calculate it is:

$$C = m \cdot (3k^2) \cdot S^2 + \sum(m \cdot mk^2 \cdot (S/2i)^2) + (m(S/2^5)^2)n + (n * 10)\,.$$

The assumptions are that the number of filters (m) and filter size (k) are going to be uniform across all layers. Where, S=input_size and

n=number of neurons in the dense layers. Summation is over all hidden layers -> 4 in number.

The '3' represents the number of input channels, '5' in the power of '2 is the number of layers and 10 is for number of classes.

Similarly we have function `formula_parameter_count` to get the **total paramters = 1,985,770** with the formula:

$$P = m * (3k^2 + 1) + 4 * m * (mk^2 + 1) + (m * (S/32)^2)n + n + n * 10 + 10.$$

Here '4' stands for the number of hidden layers in the network.

Question 2

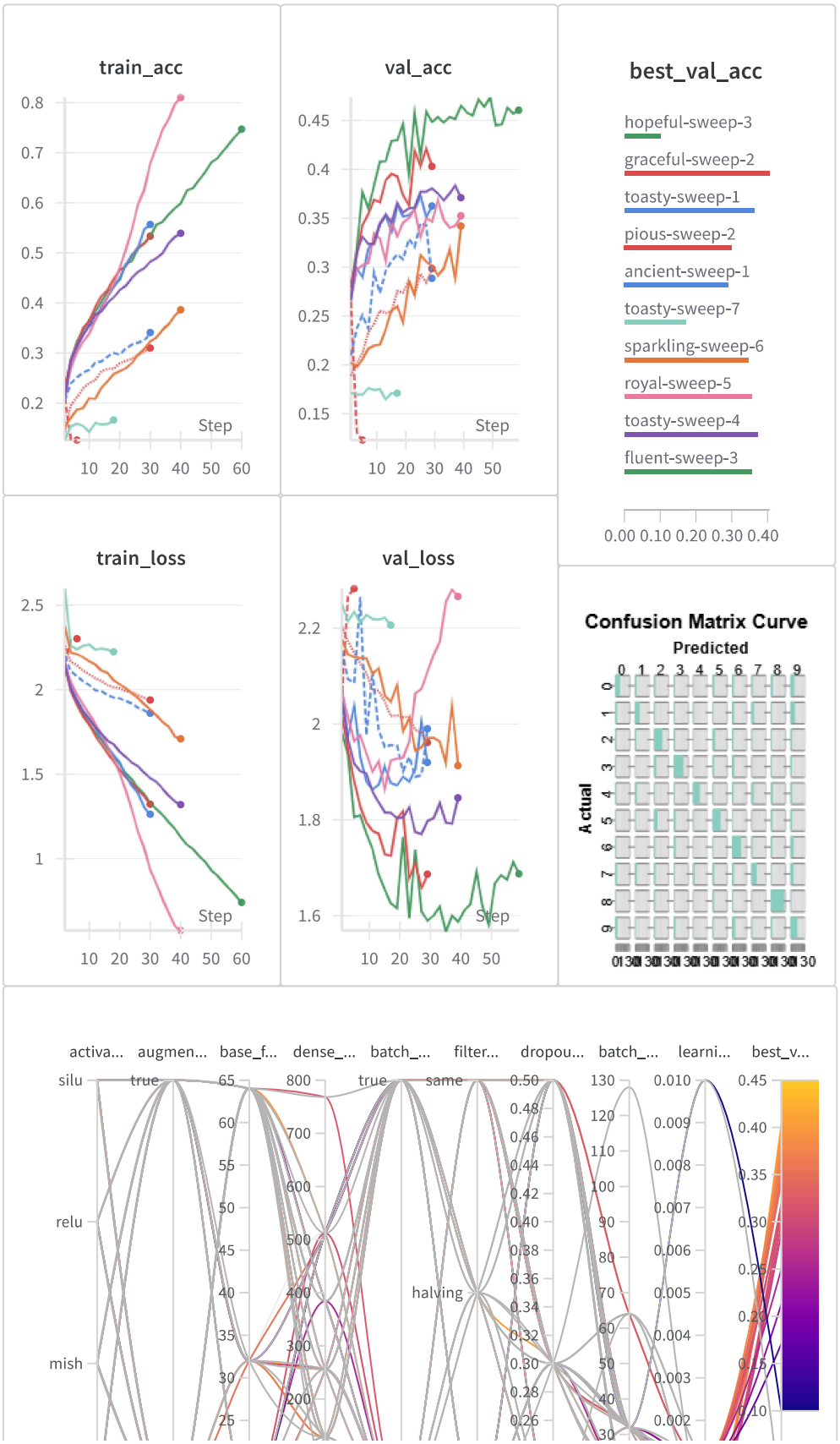
Here is the set of parameters in the sweep_config I ran a sweep over after all the filtering of parameters:

```
sweep_config = {
        'method': 'bayes',
        'metric': {'name': 'val_acc', 'goal': 'maximize'},
        'parameters': {
            'filter_counts_strategy': {'values': ['same', 'doubling'
            'base_filters':           {'values': [16, 32, 64]},
            'filter_size':            {'values': [3, 5]},
            'activation':             {'values': ['relu', 'gelu', 's
            'dense_neurons':          {'values': [128, 256, 384, 512
            'dropout_rate':           {'values': [0.2, 0.3, 0.5]},
            'learning_rate':          {'values': [0.0001, 0.001]},
            'batch_norm':             {'values': [True, False]},
            'batch_size':             {'values': [16, 32]},
            'augmentation':           {'values': [True, False]},
        }
```

- For the few initial sweeps, I got really bad results for a few values that I could notice on analysing the plots, e.g. learning_rate = 0.01, dense_neurons = 64; I decided to remove them from the sweep_config to **avoid useless iterations**.
- I also noticed that for large batch sizes like 64, 128 and filter_size = 7, 9, the **runtime was crashing** due to the high GPU compute required. I also had to remove them, but this time it was more of a **necessity**.
- To increase the overall sweep efficiency, I figured out that after 15 epochs the accuracy saturates, so there is no point in running

sweeps for more no of epochs. Also, I applied an early stopping condition which terminates the training if no improvemnt is being shown.

The plots for the sweeps that I ran successfully are listed below:

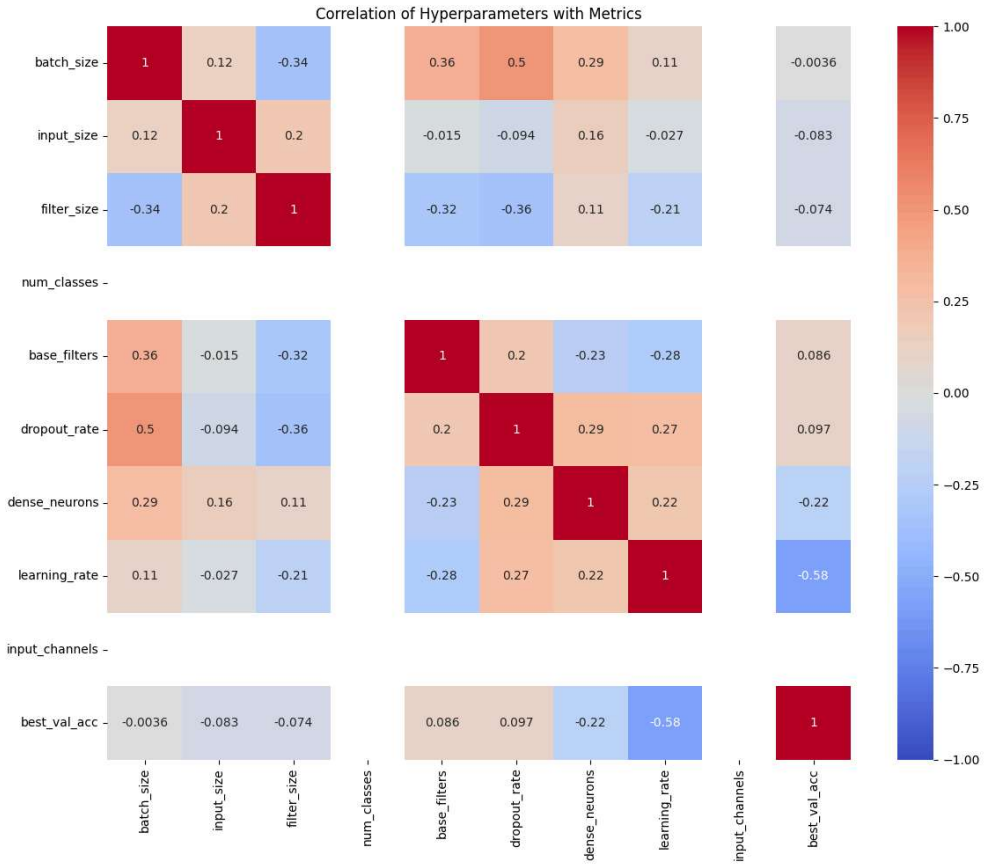


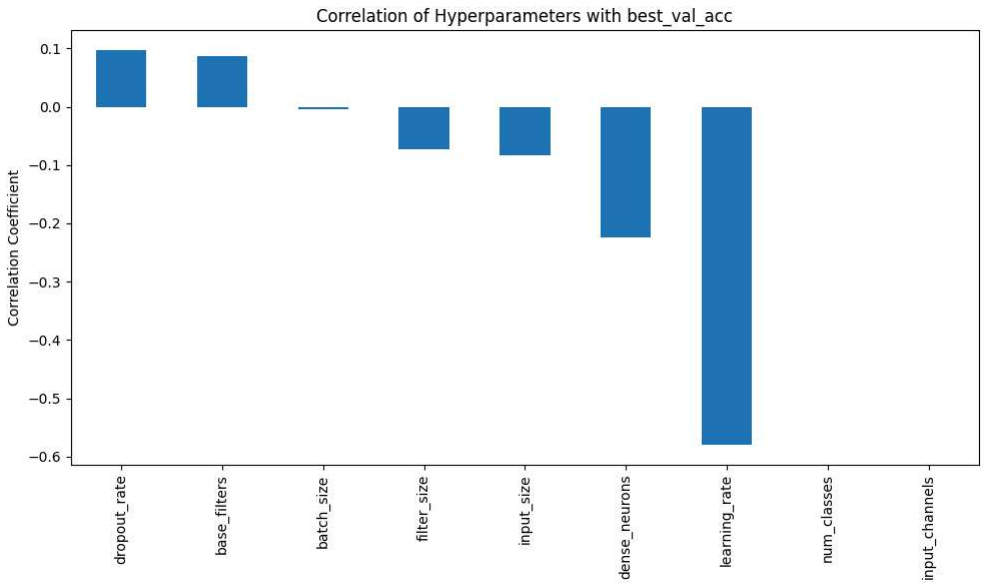
runs.summary["confusion_matrix_table"]

Filter

	Actual	Predicted
1	0	0
2	0	1
3	0	2
4	0	3
5	0	4
6	0	5
7	0	6
8	0	7
9	0	8

1 - 10 of 100 Export as CSV Col





Question 3

- Generally, gelu as an activation function did not perform well enough in comparison to other options like silu and relu.
- Augmentation, when turned off gave better results than when it was kept on, as it was leading to overfitting for the model.
- Low base filters, i.e. those <=16 in size give pretty bad accuracies as it is a multi-class classifier and images hold pretty complicated data, looking at the size of input being 224, which I feel needs a larger filter to note the characteristic features.
- Good performance needs neurons above a particular theshold. It is evident that this is certainly above **128**, looking at the input data size of **224x224**. However, it is interesting to see that increasing the size arbitrarily to **768** or something higher would take a toll on validation accuracy. This is because of overfitting, as the network becomes too dense for the model to learn and would rather memorise patterns in the images, giving high training accuracy but low validation and test accuracy.
- **Learning_rate** has the strongest correlation (negative) with the **val_acc**, coming to be = -0.58. When it is large, like 0.01 the accuracy comes down because it misses the local minima in the loss function plot due to bigger 'jumps' that the algo takes.
- Very less values of dense_neurons = 64; I decided to remove them from the sweep_config to **avoid useless iterations**.
- I also noticed that for large batch sizes like 64, 128 and filter_size = 7, 9, the **runtime was crashing** due to the high GPU compute

required. I also had to remove them, but this time it was more of a **necessity.**

Question 4

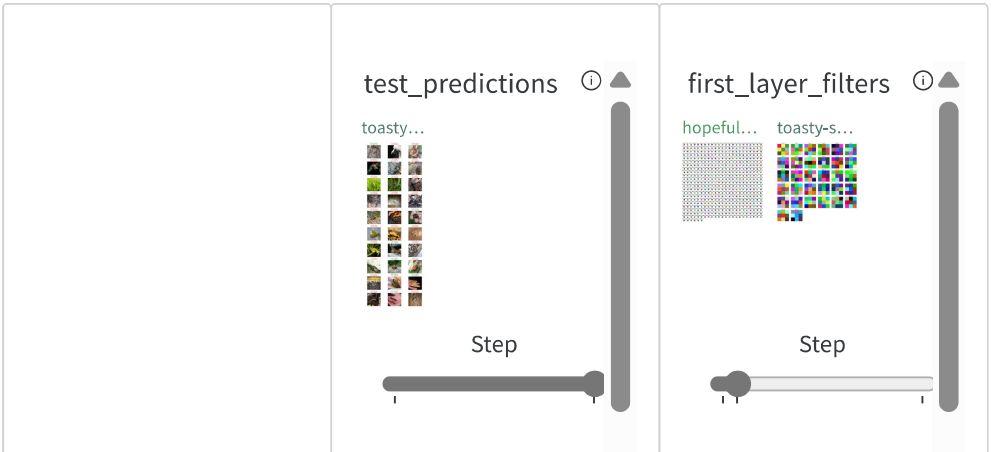
I have used the `val` folder in the given datasetas test, while splitting `train` in the ratio of 80:20 for train:val for model training. So there was no data leakage of any sort.

The best config that I found after the sweep was:

```
best_config = {
                    'activation': 'relu',
                    'batch_norm': True,
                    'batch_size': 16,
                    'input_size': 224,
                    'filter_size': 3,
                    'num_classes': 10,
                    'augmentation': False,
                    'base_filters': 32,
                    'dropout_rate': 0.,
                    'filter_sizes': [3, 3, 3, 3, 3],
                    'dense_neurons': 256,
                    'filter_counts': [512, 256, 128, 64, 32],
                    'learning_rate': 0.0001,
                    'input_channels': 3,
                    'filter_counts_strategy': 'halving'}
```

It gave me a `test_acc` of **0.4404999911785126** with a `test loss` of **1.708396553993225**.

The needed plots for `test_predictions` and `filters` in first layer are mentioned below:



test_predictions_grid (24 25

hopeful...

Question 5

Tl k to github is: https://github.com/N shay-Jain/DA6401-Assign-2/tree/main/Part-A

The Readme.md file is attached in the par t directory.

Part B

Question 1

- I have used `transforms` from the **torchvision library** to get the images transformed to the lower size that is needed, ie 224x224 for the models (Resnet50).
- I had to replace the final fully connected layer with a new one for **10 classes** using the enlisted one in the **load_pretrained_model** function:

```
# Replace the final fully connected layer
in_features = model.fc.in_features
model.fc = nn.Linear(in_features, num_classes)
```

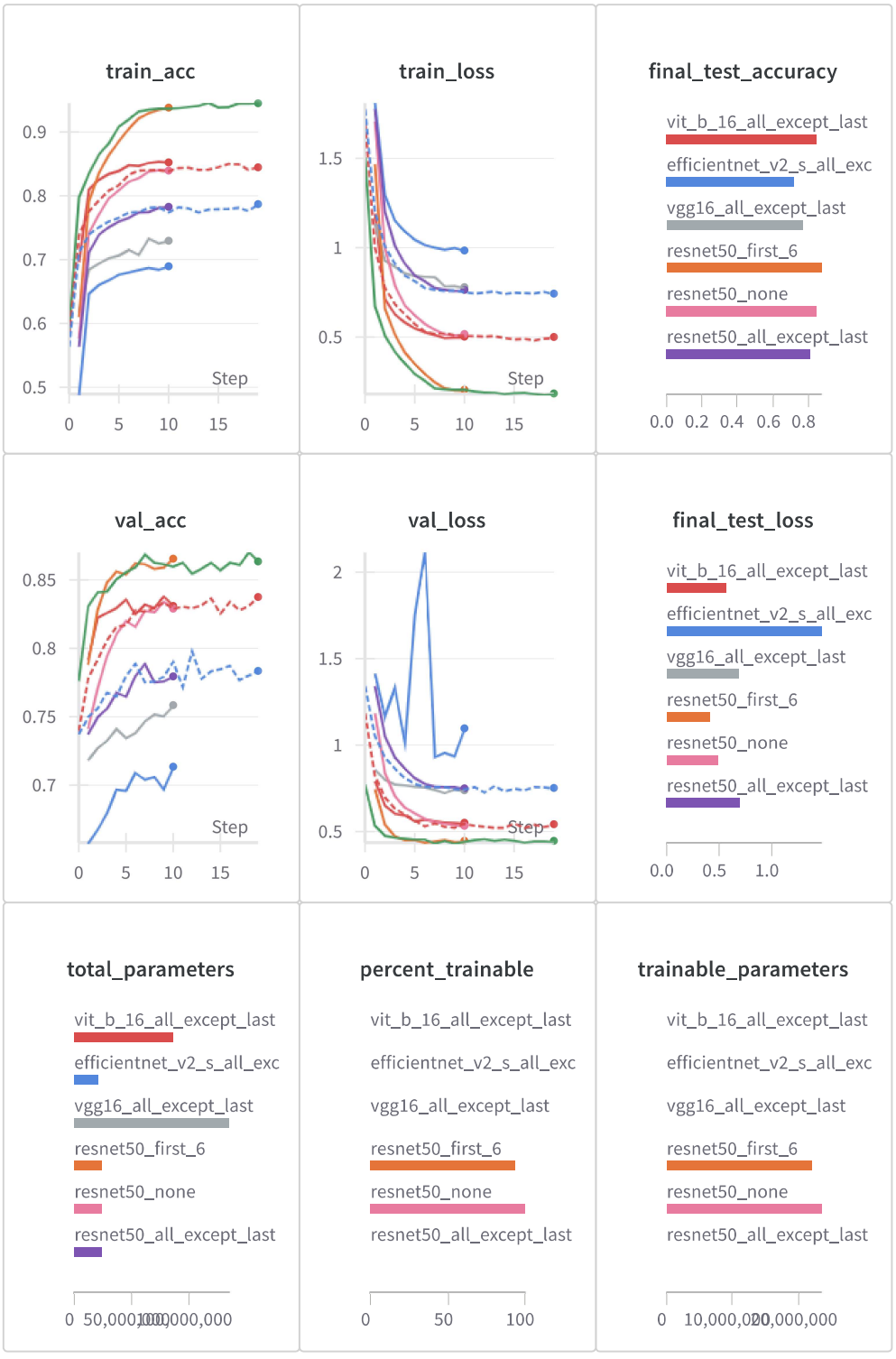
Question 2

I used the following strategy for freezing layers:

- "all_except_last": Freeze all layers except the last layer
- "none": Don't freeze any layers (full fine-tuning)
- "first_k": Freeze only the first k layers
- "all_except_k": Freeze all layers except the last k layers

I started with `resnet50` as the model and tried out these strategies to get to an optimum solution, but then I also iterated over different models, keeping the strategy constant, just to get a rough idea of how well they are performing with the equivalent at `resnet50`.

Iterating among all these led me to the best results. Here are the necessary plots of the metrics that I had logged while fine-tuning.



Question 3

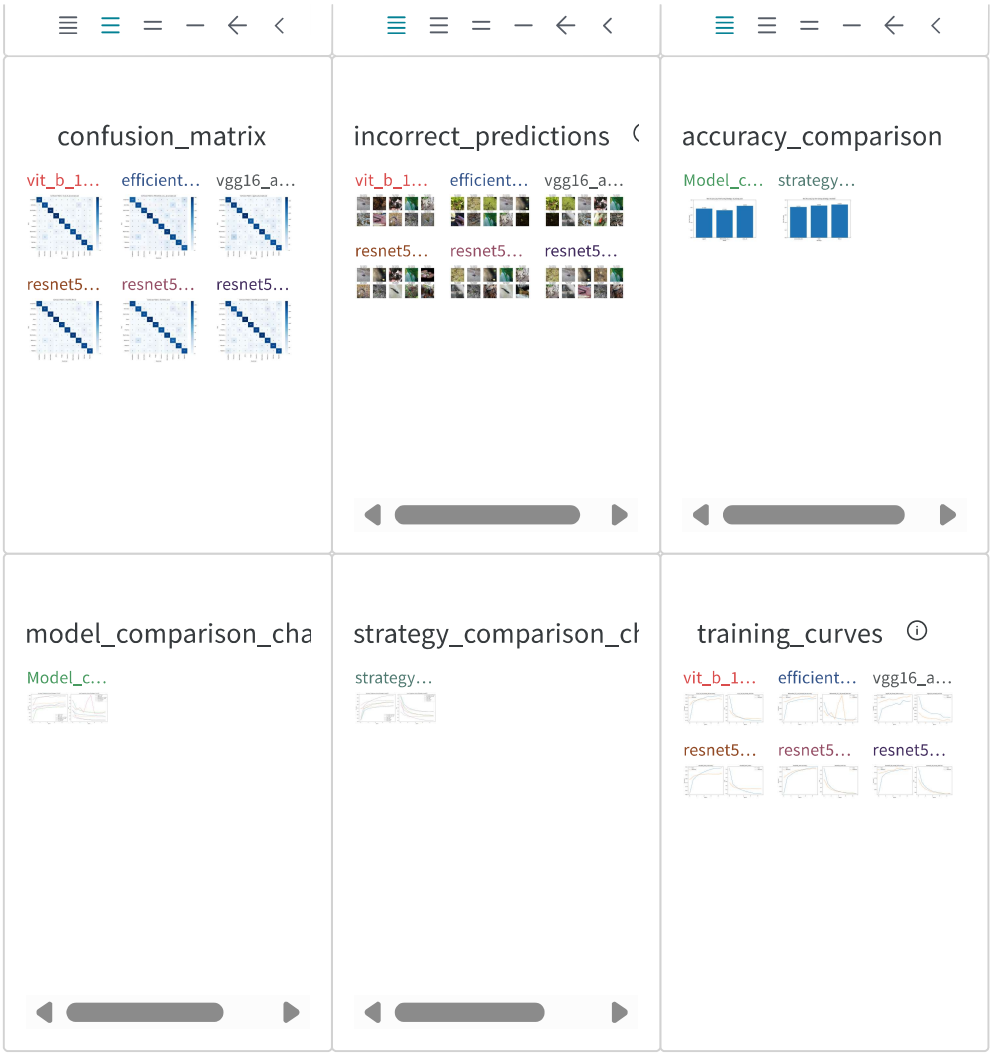
Now, I have attached the media outputs that include:

- The confusion matrices
- strategy comparison tables
- Model comparison tables

- incorrect predictions images

The inferences that I could make are:

- Finetuning large pretrained models takes much less time than training smaller neural networks from scratch. The performance is better, too, in the fine-tuning case. So it's advisable to use fine-tuning in case of complex problems. For simpler probles, it'll end up being an over-kill.
- The initial layers in the model consisted of a maximum of neurons, and later ones were negligibly small, which is evident from the plot of trainable parameters above.
- Freezing the first few layers gave the best results, as that helped us capture the inference the model had from the raw images after only a few transformations. Going in the reverse fashion, i.e. freezing everything except the last few, gave very poor results as it left not enough room for specialisation on this dataset.
- Resnet gave one of the best performances overall per unit parameters. That implies, compared to larger models, it had much better accuracy with fewer neurons, which directly implies high computational efficiency.
- vit_b_16 alone had better performance than ResNet50, but the architecture was too huge to be carried forward in all the cases, especially where compute was limited.



Question 4

The link to the Github repo is: https://github.com/Nikshay-Jain/DA6401-Assign-2/tree/main/Part-B

The Readme.md file is attached in the parent directory.

Self Declaration

I, Nikshay Jain (Roll no: MM21B044), swear on my honour that I have written the code and the report by myself and have not copied it from the internet or other students.

https://wandb.ai/mm21b044-indian-institute-of-technology-madras/inaturalist_cnn_sweep/reports/Report-DA6401-Assignment-2--VmlldzoxMjM1MzQ0Mg