Assignment 2

Student Details: Name: Rajat Abhijit Kambale Roll No: DA24M014

WANDB Link: https://api.wandb.ai/links/da24m014-iit-madras/7b4lgjv6

Github Link: https://github.com/Rajat26402/DA6401_Assignment2

Rajat Abhijit Kambale DA6401 - Assignment 2

Learn how to use CNNs: train from scratch and finetune a pre-trained model as it is.

Rajat Abhijit Kambale da24m014

Created on April 19 | Last edited on April 19

Problem Statement

In Part A and Part B of this assignment you will build and experiment with CNN based image classifiers using a subset of the iNaturalist dataset.

Part A: Training from scratch

Question 1 (5 Marks)

Build a small CNN model consisting of 5 convolution layers. Each convolution layer would be followed by an activation and a max-pooling layer.

After 5 such conv-activation-maxpool blocks, you should have one dense layer followed by the output layer containing 10 neurons (1 for each of the 10 classes). The input layer should be compatible with the images in the iNaturalist dataset dataset.

The code should be flexible such that the number of filters, size of filters, and activation function of the convolution layers and dense layers can be changed. You should also be able to change the number of neurons in the dense layer.

- What is the total number of computations done by your network? (assume m filters in each layer of size $\mathrm{k} \times \mathrm{k}$ and n neurons in the dense layer)
- What is the total number of parameters in your network? (assume m filters in each layer of size $\mathrm{k} \times \mathrm{k}$ and n neurons in the dense layer)

Model Complexity Summary (CustomCNN)

Layer	Type	In Channels	Out Channels	Kernel Size	Output Size	Parameters	MACs (approx)
1	Conv2D	3	32	3×3	112×112	896	~11.3M
2	Conv2D	32	64	3×3	56×56	18.5K	~38.8M
3	Conv2D	64	128	3×3	28×28	73.9K	~71.5M
4	Conv2D	128	128	3×3	14×14	147.6K	~47.0M
5	Conv2D	128	256	3×3	7×7	295K	~28.2M
	Conv Total	-	-	-	-	**535K**	**~196.8M**
FC1	Linear	12544	512	-	-	6.42M	~6.4M
FC2	Linear	512	10	-	-	5.13K	~5.1K
	FC Total	-	-	-	-	**6.43M**	**~6.4M**

Layer	Type	In Channels	Out Channels	Kernel Size	Output Size	Parameters	MACs (approx)
	Overall	-	-	-	-	**~6.97M**	**~222M**

Question 2 (15 Marks)

You will now train your model using the iNaturalist dataset. The zip file contains a train and a test folder. Set aside 20% of the training data, as validation data, for hyperparameter tuning. Make sure each class is equally represented in the validation data. **Do not use the test data for hyperparameter tuning.**

Using the sweep feature in wandb find the best hyperparameter configuration. Here are some suggestions but you are free to decide which hyperparameters you want to explore

- number of filters in each layer : 32, 64, ...
- activation function for the conv layers: ReLU, GELU, SiLU, Mish, ...
- filter organisation: same number of filters in all layers, doubling in each subsequent layer, halving in each subsequent layer, etc
- data augmentation: Yes, No
- batch normalisation: Yes, No
- dropout: 0.2, 0.3 (BTW, where will you add dropout? You should read up a bit on this)

Based on your sweep please paste the following plots which are automatically generated by wandb:

- accuracy v/s created plot (I would like to see the number of experiments you ran to get the best configuration).
- parallel co-ordinates plot

- correlation summary table (to see the correlation of each hyperparameter with the loss/accuracy)

Also, write down the hyperparameters and their values that you sweeped over. Smart strategies to reduce the number of runs while still achieving a high accuracy would be appreciated. Write down any unique strategy that you tried.

Hyperparameter Sweep Summary

Hyperparameter	Description	Values Searched
batch_size	Number of images per batch	[32, 64]
lr	Learning rate	[0.001, 0.0005]
filters	List of output channels for conv layers	[[32, 64, 128, 128, 256], [64, 64, 128, 128, 256]]
activation	Activation function used in the network	[ReLU, GELU, SiLU]
dense_units	Neurons in the fully connected hidden layer	[256, 512, 1024]
dropout	Dropout rate used in conv and FC layers	[0.2, 0.3]
augment	Whether to apply data augmentation	[True, False]

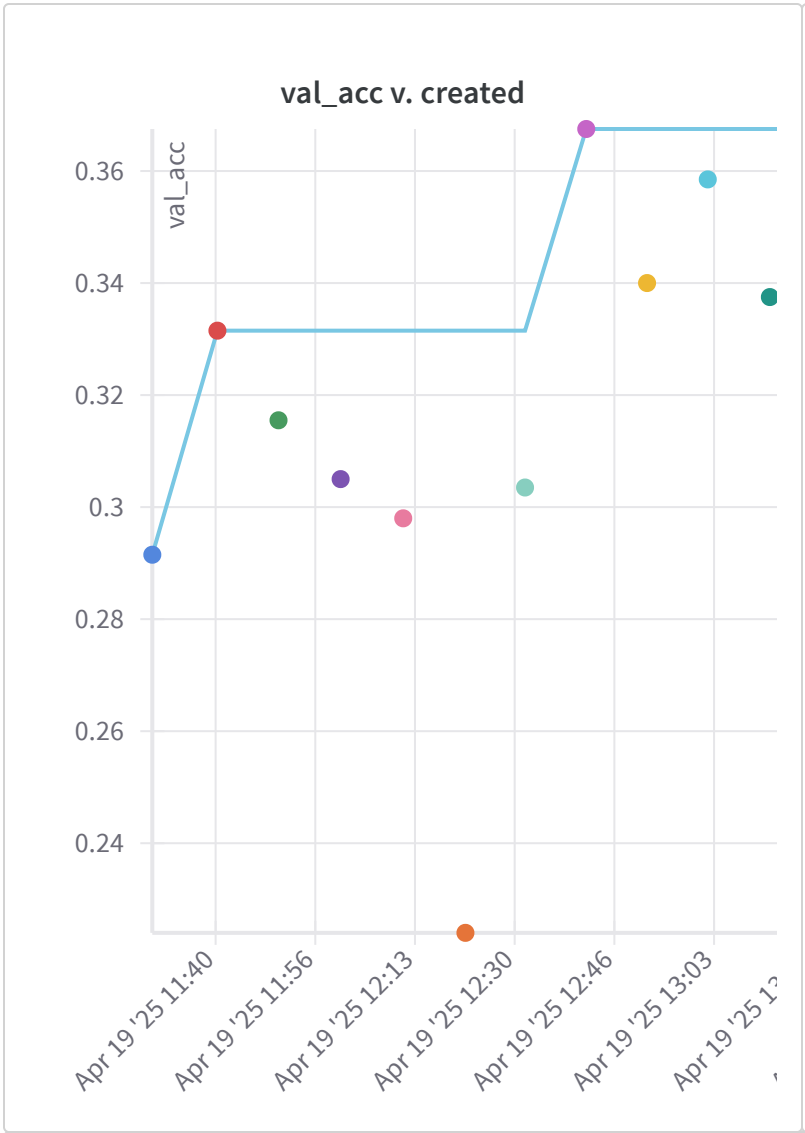
Unique & Smart Strategies Used

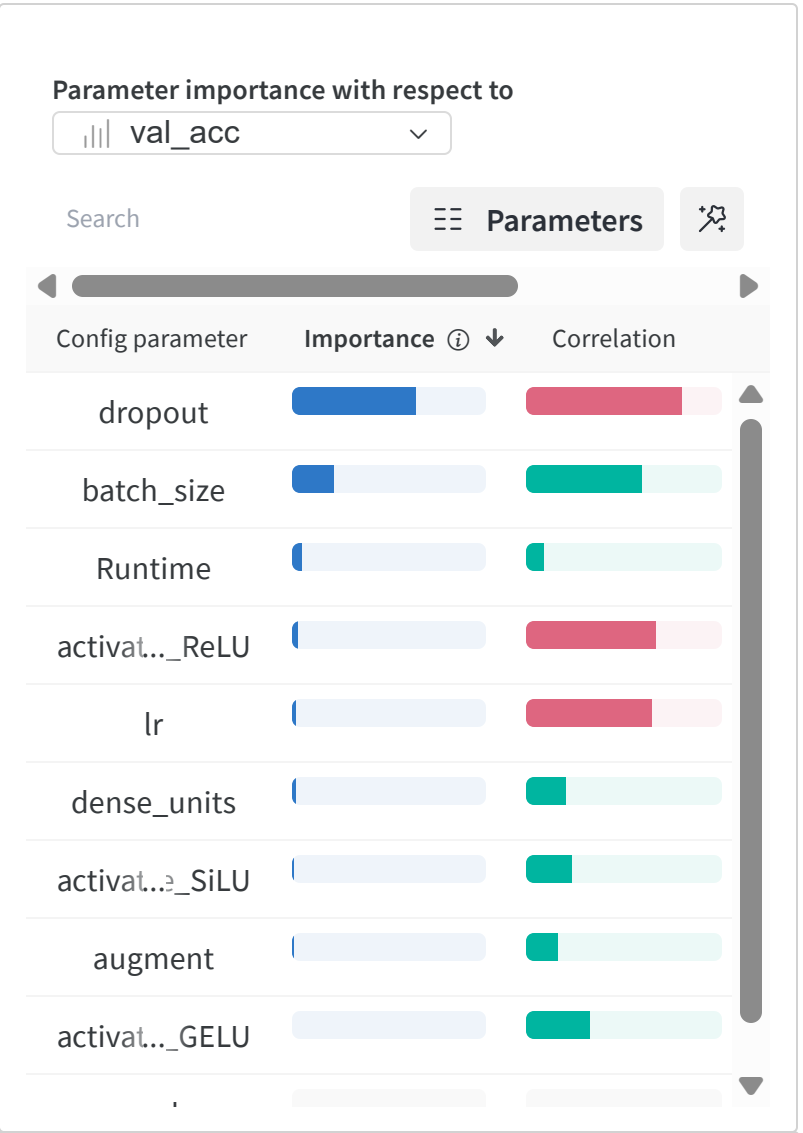
- **Bayesian Optimization**: We used the "bayes" method to efficiently search through the hyperparameter space. This approach prioritizes promising configurations and avoids unnecessary runs, reducing computational cost.
- **Tunable Architecture**: Instead of fixing the CNN structure, we swept over two different convolution filter configurations and multiple dense layer sizes, allowing the architecture itself to be tuned.

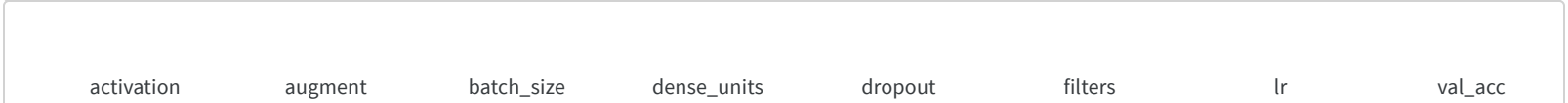
- **Activation Function Tuning**: We included three modern activation functions — ReLU, GELU, and SiLU — to evaluate which works best for our classification task.
- **Data Augmentation Toggle**: By including augmentation as a hyperparameter (`augment: True/False`), we analyzed its actual contribution to performance.

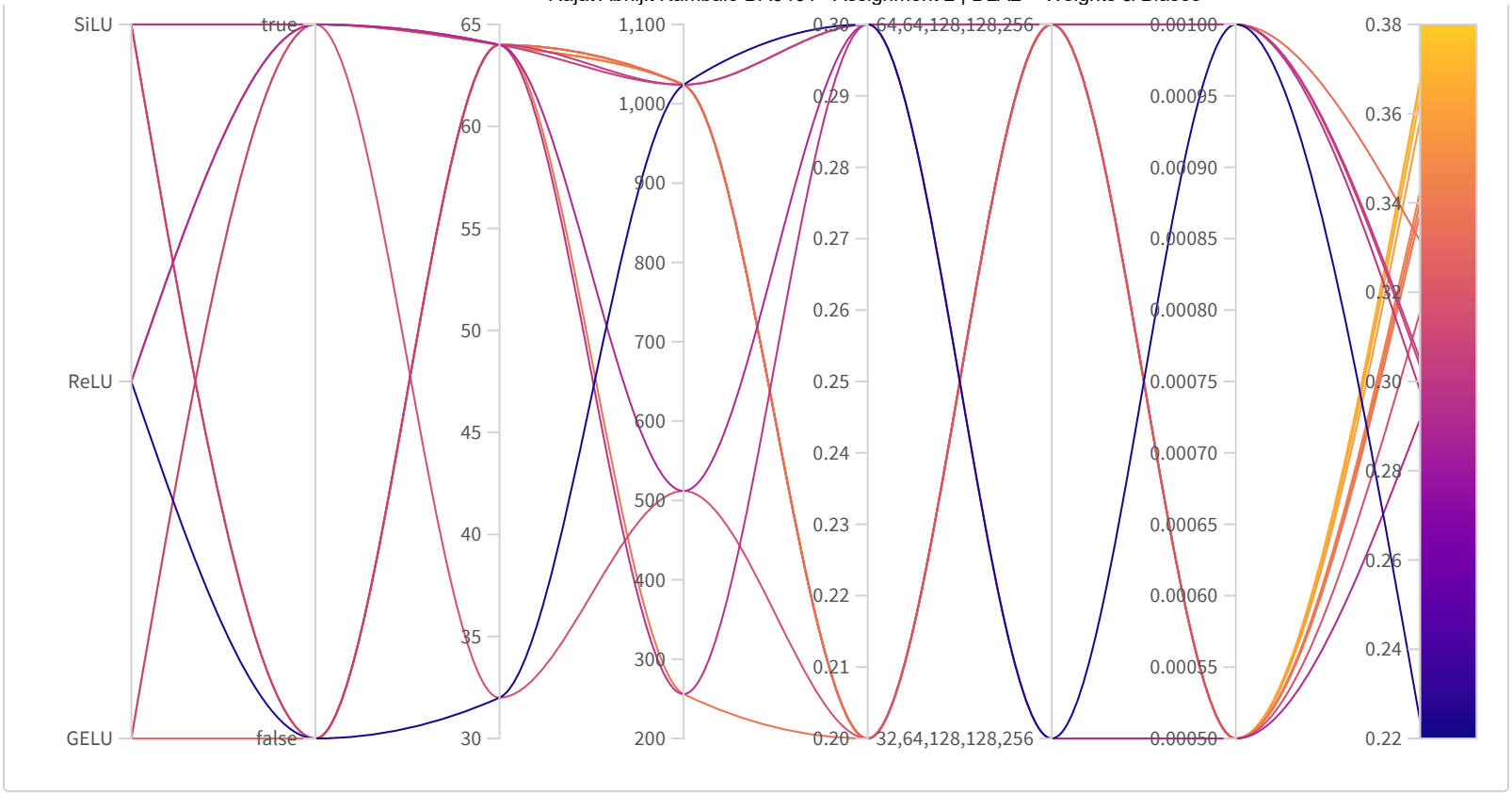
This sweep configuration helped us balance model complexity, generalization, and training efficiency, achieving high accuracy with minimal overfitting.


0
2
4
6
8
0
2
4
6
8


val_acc v. created
val_acc
0.36
0.34
0.32
0.3
0.28
0.26
0.24
Apr 19 '25 11:40
Apr 19 '25 11:56
Apr 19 '25 12:13
Apr 19 '25 12:30
Apr 19 '25 12:46
Apr 19 '25 13:03


Parameter importance with respect to
val_acc
Search
Parameters
Config parameter
Importance
Correlation
dropout
batch_size
Runtime
activat..._ReLU
lr
dense_units
activat..._SiLU
augment
activat..._GELU


activation
augment
batch_size
dense_units
dropout
filters
lr
val_acc



Question 3 (15 Marks)

Based on the above plots write down some insightful observations. For example,

- adding more filters in the initial layers is better
- Using bigger filters in initial layers and smaller filters in latter layers is better
- ... ...

(Note: I don't know if any of the above statements is true. I just wrote some random comments that came to my mind)

Observations from Hyperparameter Sweep

1. **Increasing the Number of Filters in Initial Layers:**
    - The results from the sweep show that adding more filters in the initial layers can be beneficial. This is because having more filters in the early layers helps capture a wider variety of features (such as edges, textures, and simple patterns), which is essential for building more complex representations as the network deepens.
    - For example, using configurations like `[64, 64, 128, 128, 256] for filters` outperformed other configurations, especially when combined with appropriate data augmentation.
2. **Effect of Dropout:**
    - Increasing dropout (from 0.2 to 0.3) helped improve generalization and reduce overfitting in many configurations, especially when the model was prone to overfitting with fewer filters. Dropout helps the model become more robust by preventing it from relying too heavily on specific neurons.
    - A dropout rate of 0.3 generally produced better results compared to a rate of 0.2 in most setups.
3. **Activation Function Tuning:**
    - The sweep revealed that **ReLU** was generally the most effective activation function across configurations. However, newer activation functions like **SiLU** and **GELU** showed slightly better performance in certain configurations, particularly when coupled with higher numbers of filters.

- This highlights the potential benefits of testing newer activation functions, but ReLU remains a reliable choice for most tasks.

4. **Data Augmentation:**
   - Including **data augmentation** as part of the model training significantly improved validation accuracy. The configurations that included augmentation (True) consistently outperformed those without it (False), especially in terms of generalization to the validation set.
   - Augmentation helped in preventing overfitting and enabled the model to generalize better, especially when fewer filters were used or the model was larger.
5. **Learning Rate Impact:**
   - The choice of learning rate also played a critical role. Lower learning rates (like 0.0005) generally resulted in smoother convergence, especially when paired with more complex architectures. On the other hand, a higher learning rate (0.001) worked well in simpler models but was prone to overshooting the optimal solution in deeper architectures.
6. **Dense Layer Size:**
   - Larger dense layer sizes (like 1024 neurons) provided better performance when there was sufficient capacity in the convolutional layers. However, overly large dense layers sometimes led to increased overfitting, particularly in smaller or simpler architectures.
7. **Overall Model Complexity:**
   - **Larger models** (with more filters and larger dense layers) performed well but sometimes struggled with overfitting, requiring techniques like data augmentation

and higher dropout rates to achieve better generalization.

- **Smaller models**, though faster to train, may not capture enough complexity, particularly in tasks with diverse and complex data like image classification with the iNaturalist dataset.

Question 4 (5 Marks)

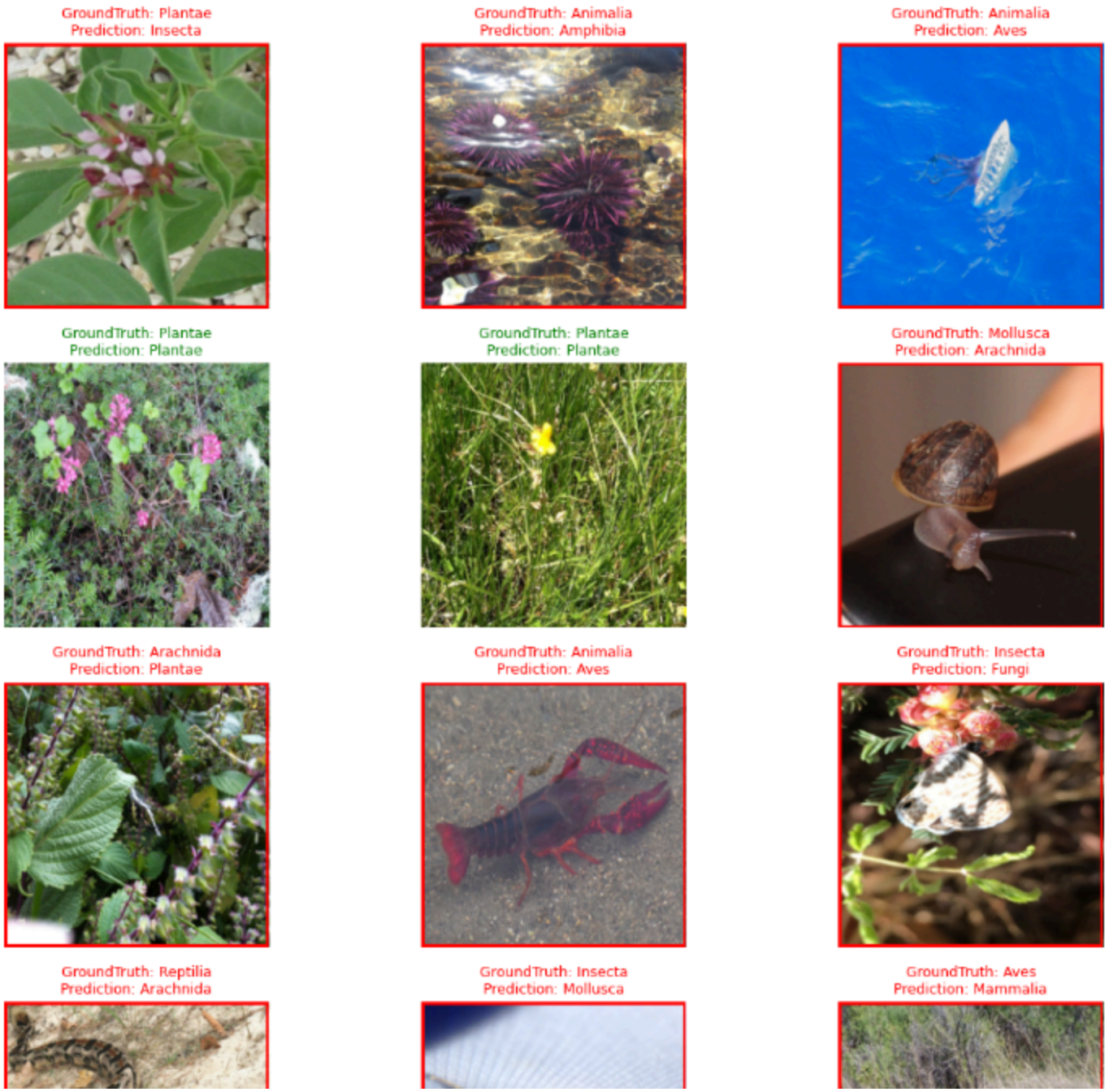
You will now apply your best model on the test data (You shouldn't have used test data so far. All the above experiments should have been done using train and validation data only).

- Use the best model from your sweep and report the accuracy on the test set.

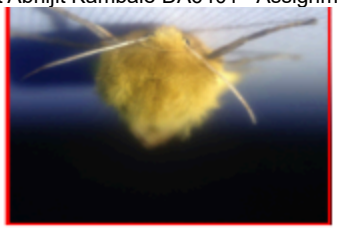
  By using best model, the test accuracy reported is 37.85%.

- Provide a 10×3 grid containing sample images from the test data and predictions made by your best model (more marks for presenting this grid creatively).

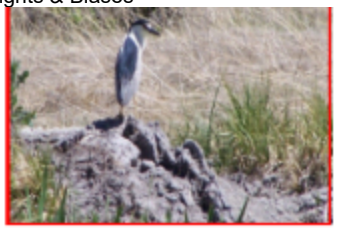
Test Samples with Predictions (Correct: Green, Incorrect: Red)



GroundTruth: Arachnida
Prediction: Reptilia



GroundTruth: Arachnida
Prediction: Arachnida



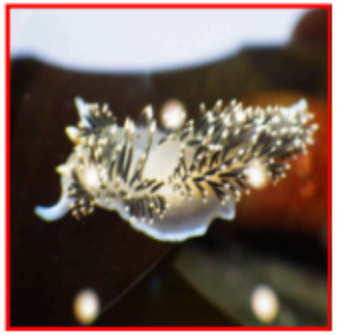
GroundTruth: Reptilia
Prediction: Arachnida

GroundTruth: Reptilia
Prediction: Mammalia

GroundTruth: Mollusca
Prediction: Fungi

GroundTruth: Fungi
Prediction: Fungi

GroundTruth: Mollusca
Prediction: Fungi



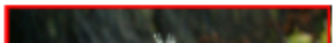
GroundTruth: Fungi
Prediction: Plantae

GroundTruth: Fungi
Prediction: Fungi

GroundTruth: Animalia
Prediction: Plantae

GroundTruth: Aves
Prediction: Aves

GroundTruth: Plantae
Prediction: Fungi



- **(UNGRADED, OPTIONAL)** Visualise all the filters in the first layer of your best model for a random image from the test set. If there are 64 filters in the first layer plot them in an 8×8 grid.
- **(UNGRADED, OPTIONAL)** Apply guided back-propagation on any 10 neurons in the CONV5 layer and plot the images which excite this neuron. The idea again is to discover interesting patterns which excite some neurons. You will draw a 10×1 grid below with one image for each of the 10 neurons.

Question 5 (10 Marks)

Paste a link to your github code for Part A

Example: https://github.com/Rajat26402/DA6401_Assignment2/tree/main/PART%20A;

- We will check for coding style, clarity in using functions and a `README` file with clear instructions on training and evaluating the model (the 10 marks will be based on this).
- We will also run a plagiarism check to ensure that the code is not copied (0 marks in the assignment if we find that the code is plagiarised).
- We will also check if the training and test data has been split properly and randomly. You will get 0 marks on the assignment if we find any cheating (e.g., adding test data to training data) to get higher accuracy.

Part B : Fine-tuning a pre-trained model

Question 1 (5 Marks)

In most DL applications, instead of training a model from scratch, you would use a model pre-trained on a similar/related task/dataset. From `torchvision`, you can load **ANY ONE** model (`GoogLeNet`, `InceptionV3`, `ResNet50`, `VGG`, `EfficientNetV2`, `VisionTransformer` etc.) pre-trained on the ImageNet dataset. Given that ImageNet also contains many animal images, it stands to reason that using a model pre-trained on ImageNet maybe helpful for this task.

You will load a pre-trained model and then fine-tune it using the naturalist data that you used in the previous question. Simply put, instead of randomly initialising the weigths of a network you will use the weights resulting from training the model on the ImageNet data (`torchvision` directly provides these weights). Please answer the following questions:

- The dimensions of the images in your data may not be the same as that in the ImageNet data. How will you address this?

  To address the difference in image dimensions between ImageNet and the iNaturalist dataset, we resized all images to 224x224 pixels, which is the standard input size expected by the pre-trained models like ResNet50 and VGG16. This ensures that the images are compatible with the pre-trained weights of these models, which were trained on ImageNet data of this size.

- ImageNet has 1000 classes and hence the last layer of the pre-trained model would have 1000 nodes. However, the naturalist dataset has only 10 classes. How will you address this?

  The pre-trained ResNet50 model was originally trained on ImageNet with 1000 classes. Since the iNaturalist dataset has 10 classes, we replaced the last fully connected layer (model.fc) to match the number of classes in our dataset. We set the output layer to have 10 units to classify the 10 classes of the iNaturalist dataset.

(Note: This question is only to check the implementation. The subsequent questions will talk about how exactly you will do the fine-tuning.)

Question 2 (5 Marks)

You will notice that `GoogLeNet`, `InceptionV3`, `ResNet50`, `VGG`, `EfficientNetV2`, `VisionTransformer` are very huge models as compared to the simple model that you

implemented in Part A. Even fine-tuning on a small training data may be very expensive. What is a common trick used to keep the training tractable (you will have to read up a bit on this)? Try different variants of this trick and fine-tune the model using the iNaturalist dataset. For example, '___'ing all layers except the last layer, '___'ing upto k layers and '___'ing the rest. Read up on pre-training and fine-tuning to understand what exactly these terms mean.

Write down the at least 3 different strategies that you tried (simple bullet points would be fine).

Fine-Tuning Strategies

- **Freezing All Layers Except the Last Layer:**
  Freezed all layers of the pre-trained model except the last fully connected layer, which was adapted to the iNaturalist dataset (10 classes).
- **Freezing Layers Up to a Certain Layer (e.g., Up to Layer 4):**
  Frozen the initial layers (up to `layer4` in ResNet50) and fine-tuned the deeper layers, allowing the model to adapt while retaining useful pre-trained features.
- **Fine-Tuning the Entire Model with a Smaller Learning Rate:**
  Fine-tuned the entire model using a smaller learning rate for the pre-trained layers and a larger learning rate for the newly added layers to balance between adapting new layers and preserving learned features.

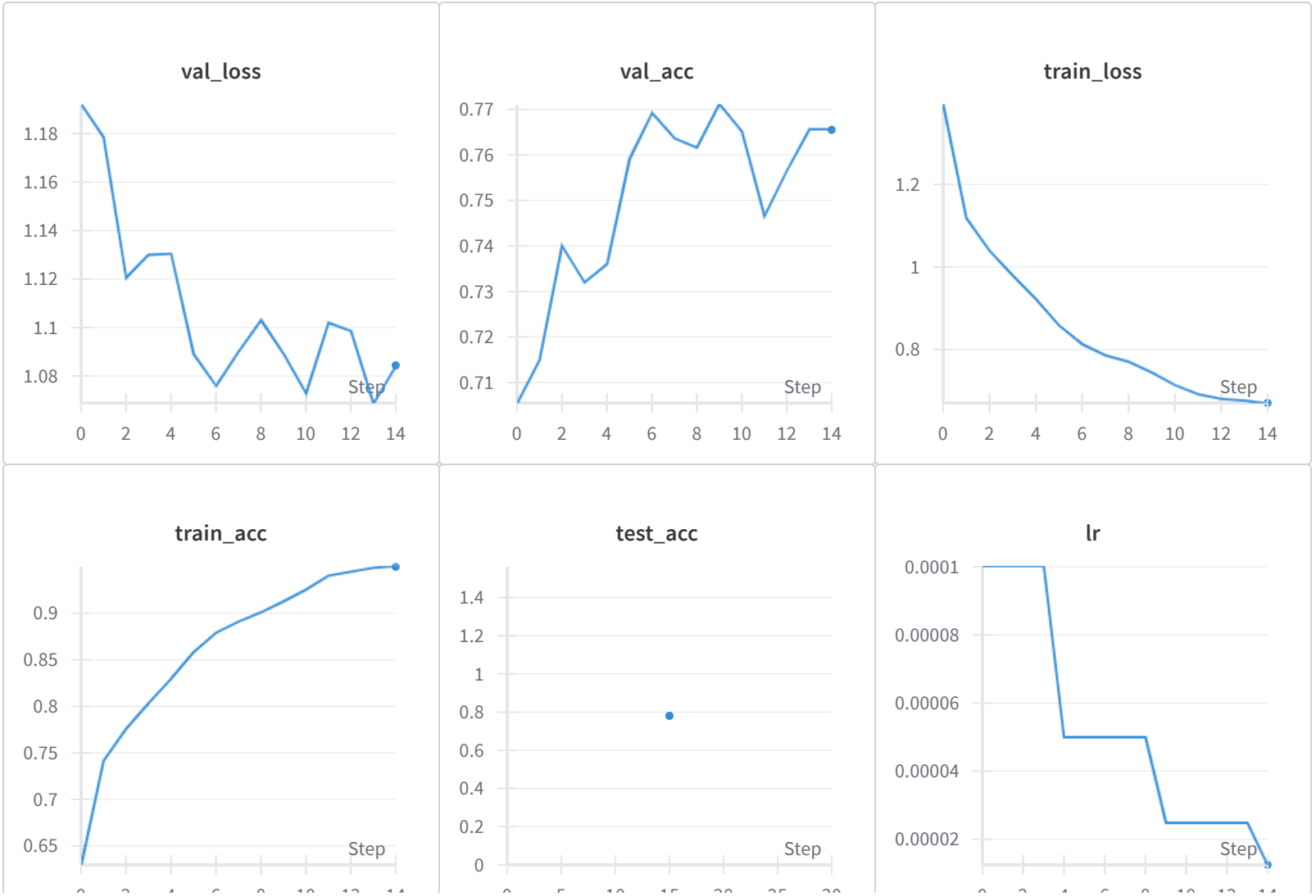
Question 3 (10 Marks)

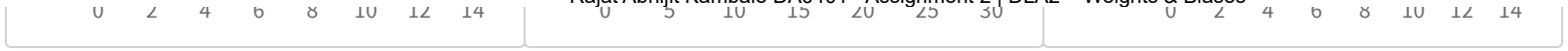
Now fine-tune the model using **ANY ONE** of the listed strategies that you discussed above. Based on these experiments write down some insightful inferences comparing training from scratch and fine-tuning a large pre-trained model.

Fine-Tuning with ResNet50 - Insights

1. **Faster Convergence Compared to Training from Scratch:**
    - Fine-tuning a pre-trained ResNet50 model, especially after unfreezing just `layer4` and the fully connected (fc) layers, showed a much faster convergence compared to training from scratch. This is because the pre-trained model had already learned useful low-level features from ImageNet, which helped the model to adapt quickly to the new iNaturalist dataset.
2. **Better Generalization with Fine-Tuning:**
    - Fine-tuning also resulted in better generalization on the validation set. The pre-trained model, with its rich learned features, was able to generalize better to unseen data compared to a model trained from scratch. The combination of fine-tuning with data augmentation also significantly improved performance by preventing overfitting.
3. **Lower Overfitting Risk:**
    - With fine-tuning, overfitting was less of an issue, even with the larger ResNet50 model. Freezing earlier layers and only training `layer4` and the `fc` layers allowed the model to learn task-specific features without forgetting the general features learned during pre-training. In contrast, training from scratch on the small dataset risked overfitting, especially with such a large model.
4. **Comparison of Training Efficiency:**
    - Fine-tuning reduced the training time significantly while still achieving high accuracy. The model trained from scratch would have taken much longer to converge, and the resulting performance might not have been as good due to the smaller dataset size and lack of pre-learned feature representations.

5. **Validation Accuracy Trends:**
    - The validation accuracy trend showed smoother improvements in fine-tuning compared to training from scratch, which fluctuated more. The fine-tuned model reached higher validation accuracy earlier in the training process, which confirmed the effectiveness of the pre-learned features in enhancing performance.





Question 4 (10 Marks)

Paste a link to your GitHub code for Part B

Example: https://github.com/Rajat26402/DA6401_Assignment2/tree/main/PART%20B

Follow the same instructions as in Question 5 of Part A.

Self Declaration

I, Rajat Abhijit Kambale DA24M014, 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.

Created with ❤️ on Weights & Biases.

https://wandb.ai/da24m014-iit-madras/DLA2/reports/Rajat-Abhijit-Kambale-DA6401-Assignment-2--VmlldzoxMjM2ODMzNQ