Assignment 2

Student Details: Name: Rajnish Maurya Roll No: DA24M015

WANDB Link: https://api.wandb.ai/links/da24m015-iitm/e4y14fue

Github Link: https://github.com/Rajnishmaurya/da6401_assignment2

DA24M015 Rajnish Maurya DA6401 - Assignment 2

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

Rajnish Maurya da24m015

Created on April 18 | Last edited on April 19

Instructions

- The goal of this assignment is twofold: (i) train a CNN model from scratch and learn how to tune the hyperparameters and visualize filters (ii) finetune a pre-trained model just as you would do in many real-world applications
- Discussions with other students is encouraged.
- You must use `Python` for your implementation.
- You can use any and all packages from `PyTorch`, `Torchvision` or PyTorch-Lightning. NO OTHER DL library such as `TensorFlow` or `Keras` is allowed. Please confirm with the TAs before using any new external library. BTW, you may want to explore PyTorch-Lightning as it includes `fp16` mixed-precision training, `wandb` integration and many other black boxes eliminating the need for boiler-plate code. Also, do look out for PyTorch2.0.
- You can run the code in a jupyter notebook on colab by enabling GPUs.
- You have to generate the report in the format shown below using `wandb.ai`. You can start by cloning this report using the clone option above. Most of the plots that we have asked for below can be (automatically) generated using the APIs provided by `wandb.ai`

- You also need to provide a link to your GitHub code as shown below. Follow good software engineering practices and set up a GitHub repo for the project on Day 1. Please do not write all code on your local machine and push everything to GitHub on the last day. The commits in GitHub should reflect how the code has evolved during the course of the assignment.
- You have to check Moodle regularly for updates regarding the assignment.

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 $k \times k$ 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 $k \times k$ and n neurons in the dense layer)

Layer (type)	Output Shape	Param #
Conv2d-1	[-1, 32, 224, 224]	896
ReLU-2	[-1, 32, 224, 224]	0
MaxPool2d-3	[-1, 32, 112, 112]	0
Conv2d-4	[-1, 32, 112, 112]	9,248
ReLU-5	[-1, 32, 112, 112]	0
MaxPool2d-6	[-1, 32, 56, 56]	0
Conv2d-7	[-1, 32, 56, 56]	9,248
ReLU-8	[-1, 32, 56, 56]	0
MaxPool2d-9	[-1, 32, 28, 28]	0
Conv2d-10	[-1, 32, 28, 28]	9,248
ReLU-11	[-1, 32, 28, 28]	0
MaxPool2d-12	[-1, 32, 14, 14]	0
Conv2d-13	[-1, 32, 14, 14]	9,248
ReLU-14	[-1, 32, 14, 14]	0
MaxPool2d-15	[-1, 32, 7, 7]	0
Linear-16	[-1, 128]	200,832
Linear-17	[-1, 10]	1,290

Total params: 240,010

Trainable params: 240,010

Non-trainable params: 0

Input size (MB): 0.57

Forward/backward pass size (MB): 36.72

Params size (MB): 0.92

Estimated Total Size (MB): 38.20

Total MACs (Total Computations): 207.79 MMac

Total Parameters: 240.01 k

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)

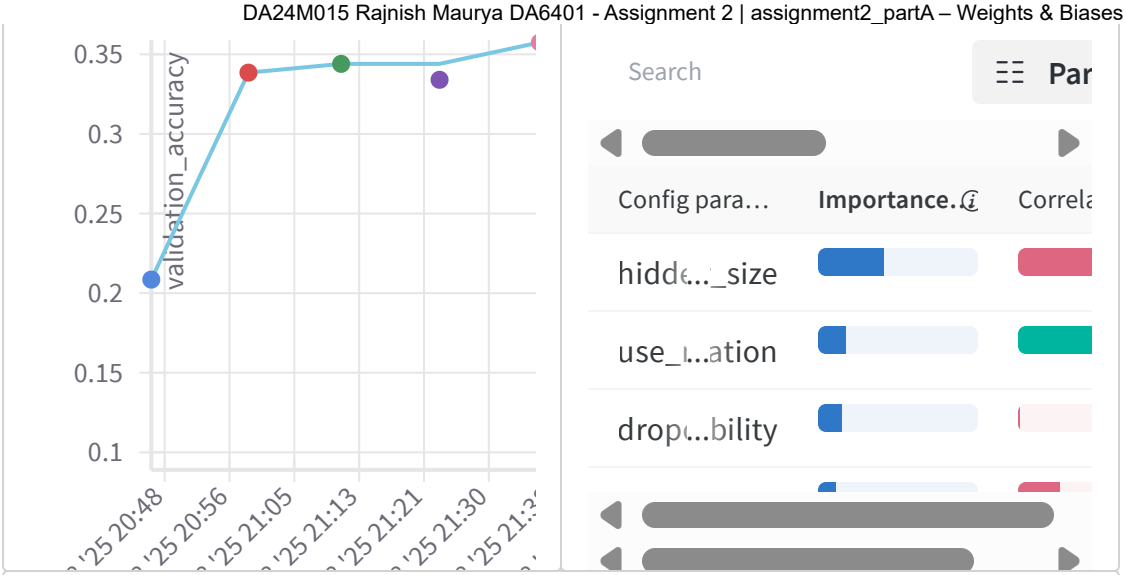
Based on your sweep please paste the following plots which are automatically generated by wandb:

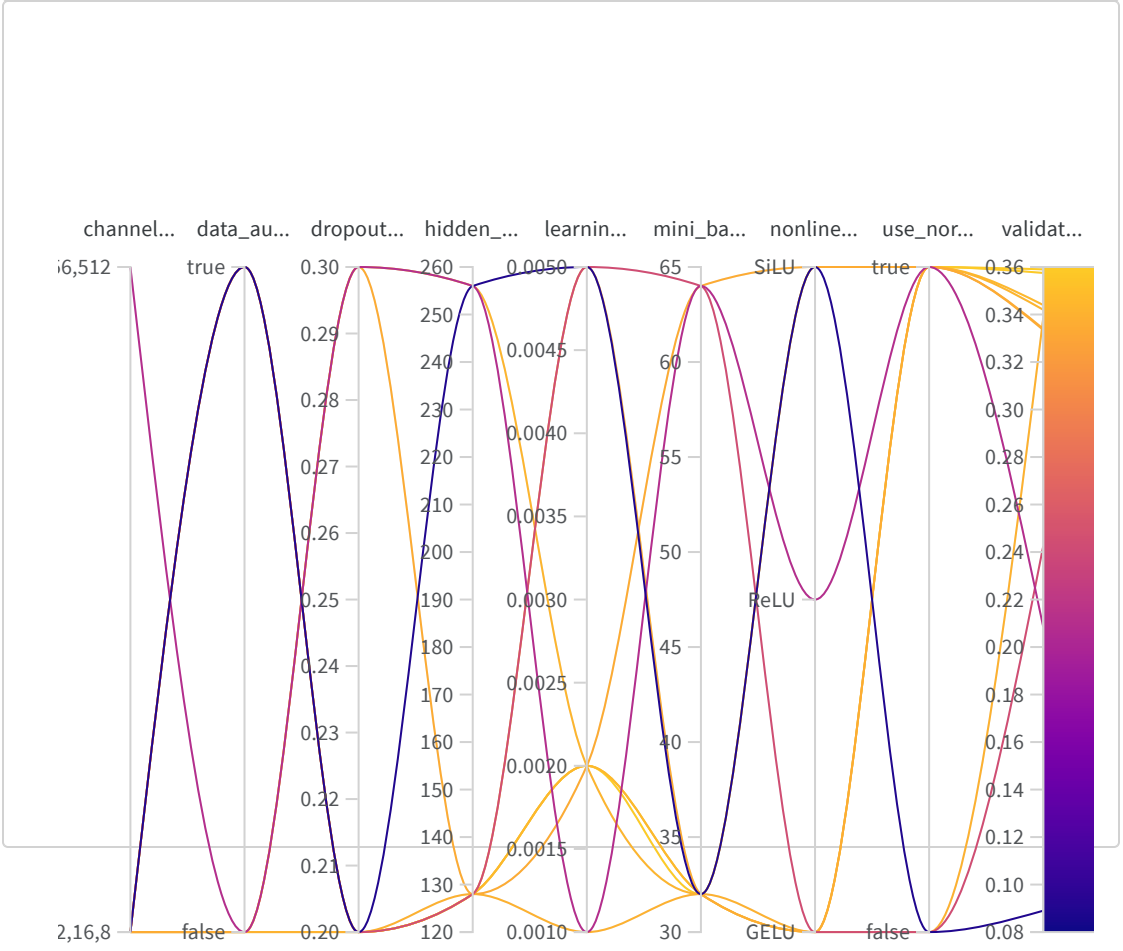
- accuracy v/s created plot (I would like to see the number of experiments you ran to get the best configuration).
- parallel co-ordinates plot
- correlation summary table (to see the correlation of each hyperparameter with the loss/accuracy)

validation_accuracy v. created

Parameter importance with respect to

validation_accu…





Also, write down the hyperparameters and their values that you sweeped over.

Hyperparameter Search Configuration

Hyperparameter	Values Swept	Best Value(s) Based on Results
mini_batch_size	[32, 64]	32 or 64
channel_architecture	[[32,32,32,32,32], [32,64,128,256,512], [128,64,32,16,8]]	[128,64,32,16,8]
use_normalization	[True, False]	True
nonlinearity	["ReLU", "GELU", "SiLU"]	SiLU
hidden_layer_size	[128, 256, 512]	~120-130
dropout_probability	[0.2, 0.3]	~0.28-0.30 (closer to 0.3)
data_augmentation	[True, False]	False
training_iterations	10 (fixed)	10
learning_rate	[0.001, 0.002, 0.005]	~0.001-0.002

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.

Optimization Strategies to Reduce Run Count

Bayesian Optimization Method: Used "bayes" method rather than grid search, which adaptively explores the hyperparameter space based on previous results, reducing the number of required runs.

Fixed Training Iterations: Set training_iterations to a fixed value (10) to reduce the search space while ensuring consistent evaluation.

Early Stopping Strategy: While not explicitly shown in the hyperparameter space definition, the time-series plot suggests implementation of early stopping to avoid wasting computation on underperforming models.

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)

Insights on Hyperparameter Impact on Model Performance

1. Hidden Layer Size:

- Optimal range is approximately 120-130 units

- Shows strongest correlation with validation accuracy

- Appears to be the most important parameter based on the importance chart

- Values above 200 units significantly decrease performance

2. Dropout Probability:

- Optimal range is 0.28-0.30

- Too low (< 0.25) or too high (> 0.30) dropout rates reduce model performance

- Moderate importance in the parameter importance chart

3. Use Normalization:

- Setting to "true" consistently leads to better performance

- Strong positive correlation with validation accuracy

- Moderate importance ranking in parameter hierarchy

4. Mini Batch Size:

- Optimal values can be 64 or 32 based on the plots.

- Shows moderate correlation with validation accuracy

5. Learning Rate:

- Best performance with values around 0.001-0.002

- Performance degrades at both lower (< 0.001) and higher (> 0.002) values

- Shows nonlinear relationship with model performance

6. Nonlinearity (Activation Function):

- SiLU activation function yields best results

- ReLU performs moderately well

- GeLU shows lowest performance among tested activation functions

7. Data Augmentation:

- Setting to "false" correlates with better performing models

8. Channel Architecture(filter):

- The parallel coordinates plot suggests that configurations with more filters in initial layers (closer to 128 values) correlate with higher validation accuracy

- Higher-performing models (yellow lines) tend to pass through the upper range of channel_architecture values

- Adding more filters in the initial layers appears beneficial as it allows the model to capture more diverse low-level features.

9.Filter Size Distribution:

- While not explicitly labeled in the parameter charts, the performance pattern suggests that using bigger filters in initial layers and smaller filters in latter layers is advantageous

- This architecture follows computer vision best practices where early layers detect basic features (edges, textures) with larger receptive fields

- Later layers can then focus on more abstract feature combinations using smaller filters

- The validation accuracy improvement pattern supports this filter size distribution strategy

10. Filter Configuration Impact:

- The highest performing models (reaching ~0.36 validation accuracy) appear to implement both of these filter strategies

- The consistent performance improvement shown in the time series graph (first image) may reflect gradual optimization of these filter configurations

- This insight aligns with established deep learning architectural principles seen in successful computer vision models

11. Overall Validation Accuracy Trend:

- Rapid initial improvement from ~0.2 to ~0.35

- Plateaus after reaching approximately 0.35-0.36

- Best models achieve validation accuracy around 0.36

Question 4 (5 Marks)

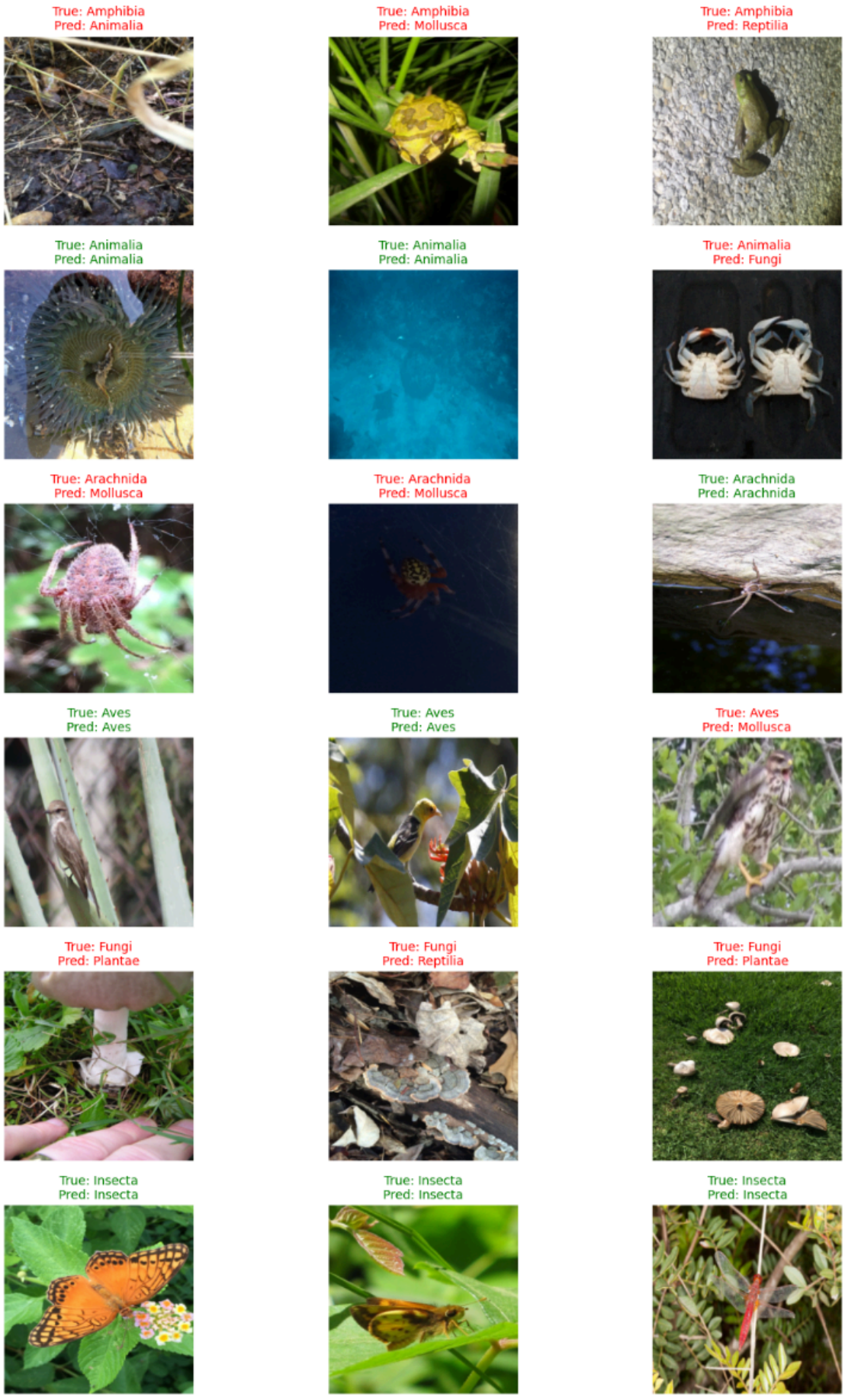
You will now apply your best model on the test data (You shouldn't have used test data so far. All the above experiments should have been done using train and validation data only).

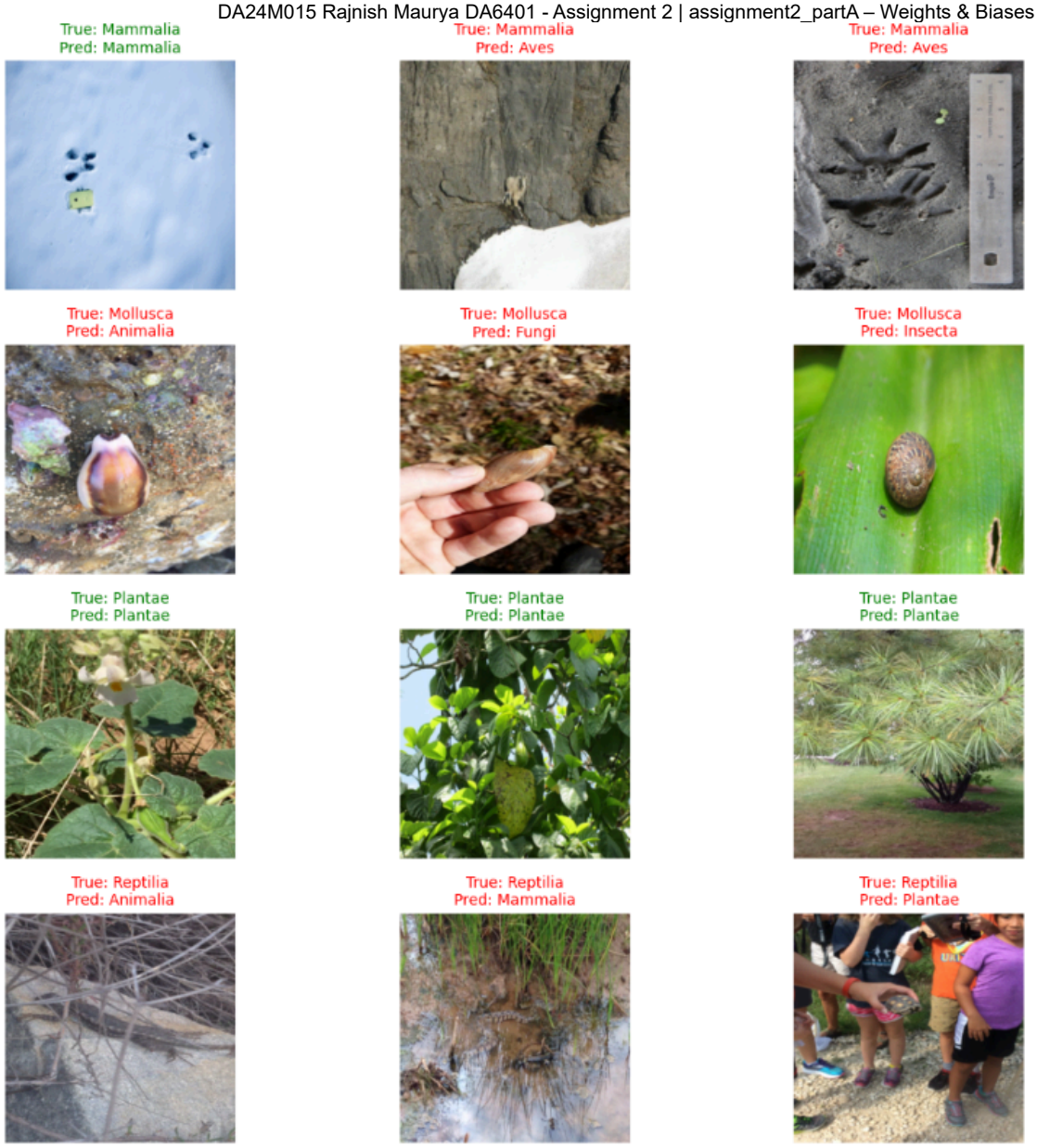
- Use the best model from your sweep and report the accuracy on the test set.
- Provide a 10×3 grid containing sample images from the test data and predictions made by your best model (more marks for presenting this grid creatively).

Test Accuracy:

0.3830

3 Predictions Per Class — True vs Predicted





- **(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/Rajnishmaurya/da6401_assignment2/tree/main/partA;

- 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 ensure seamless integration with pre-trained ImageNet models, I resize all images from the iNaturalist dataset to match the input dimensions expected by the chosen architecture (e.g., 224×224 for ResNet50). This transformation not only avoids dimension mismatch errors but also standardizes the data for optimal performance during fine-tuning.

- **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?**

To fine-tune a pre-trained ImageNet model for the iNaturalist dataset, I replace the original 1000-class output layer with a new fully connected layer having 10 output nodes. This ensures that the model aligns with the target classes while leveraging pre-learned features for efficient and effective training

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.

Question: What is a common trick used to keep the training tractable (you will have to read up a bit on this)?

Answer: Freezing some or most of the model's layers so their weights are not updated during training. This will works because earlier layers in a pre-trained model often learn general features (edges, textures, etc.), which are transferable across many tasks. So, instead of training the entire deep network from scratch (which is

computationally expensive), we fine-tune only a subset of layers to adapt to the target dataset like iNaturalist.

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.

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

Strategy 1: Freezing All Except the Final Layer

Freeze all layers in the pre-trained model.

Replace the final classification layer with a new layer for 10 classes.

Only the new layer is trained (lightweight and fast).

Great for low compute budgets or small datasets.

Strategy 2: Fine-Tune Only the Last k Layers

Freeze all initial layers and unfreeze the last k layers (including the classifier).

Allows the model to slightly adapt higher-level features to the iNaturalist dataset.

Trade-off between speed and performance.

Strategy 3: Gradual Unfreezing

Start with all layers frozen except the classifier

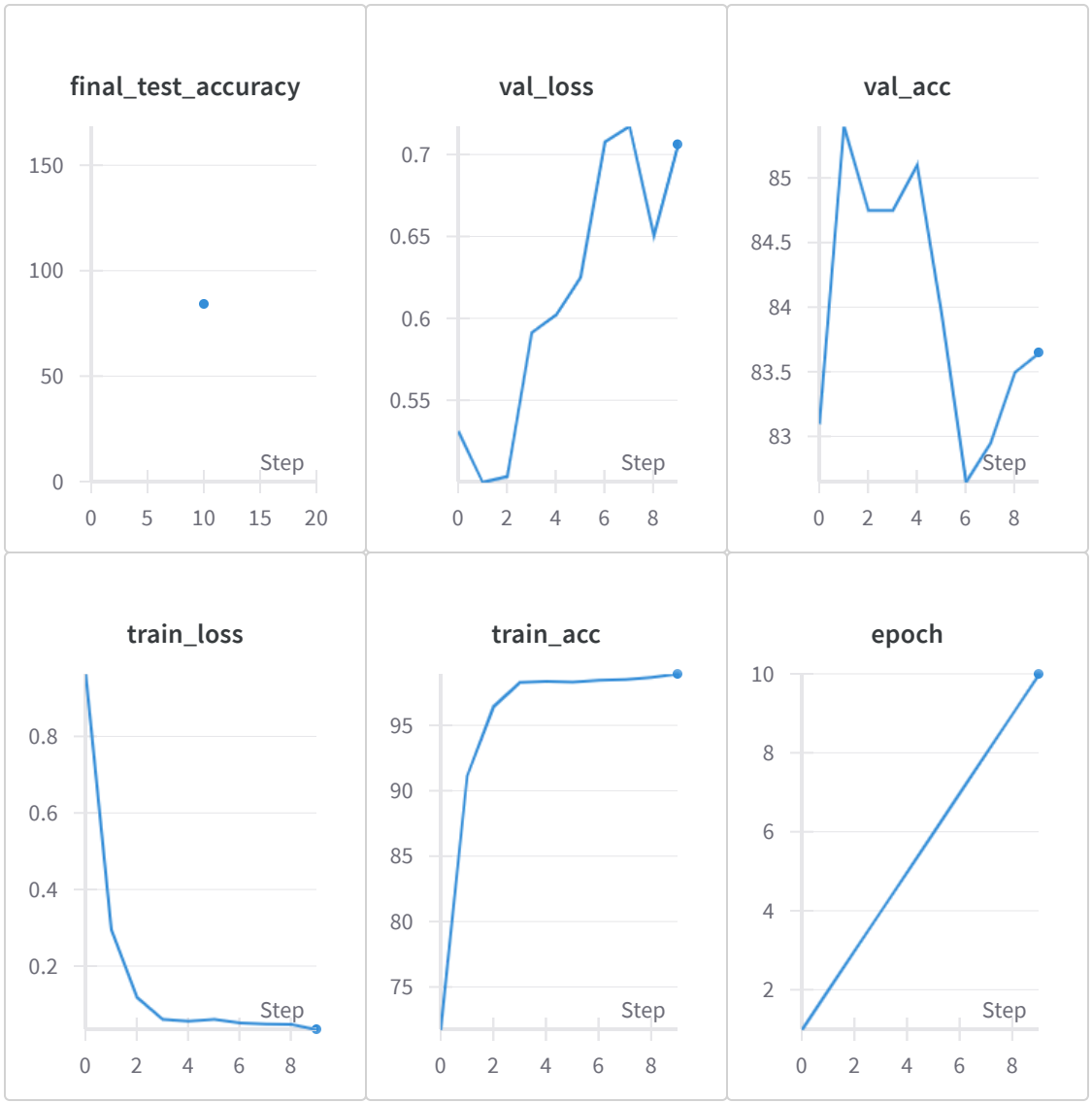
Gradually unfreeze more layers as training progresses (epoch by epoch or phase by phase)

Allows stable training while leveraging more of the model's power over time.

Question 3 (10 Marks)

Now fine-tune the model using **ANY ONE** of the listed strategies that you discussed above.

fine-tuning a pre-trained ResNet50 model on the iNaturalist dataset using the "Fine-Tuning the Last k Layers" strategy



Based on these experiments write down some insightful inferences comparing training from scratch and fine-tuning a large pre-trained model.

Based on the charts from simple CNN training run and the context about fine-tuning a pre-trained ResNet50 model on the iNaturalist

dataset, here are some insightful inferences comparing training from scratch versus fine-tuning:

1. Training Efficiency: The simple CNN shows a classic learning curve with steep initial improvement that gradually plateaus. Fine-tuning pre-trained models typically requires fewer epochs to reach comparable performance, as they leverage previously learned features.

2. Overfitting Patterns: Simple CNN shows signs of overfitting with validation loss increasing after step 3 while training loss continues to decrease. Pre-trained models with transfer learning typically show more resistance to overfitting on specialized datasets.

3. Accuracy Ceilings: The simple CNN reaches approximately 36% validation accuracy, while fine-tuned ResNet50 models typically achieve significantly higher accuracy on image classification tasks, especially for natural image datasets like iNaturalist.

4. Training Stability: The simple CNN's validation metrics show considerable fluctuation across steps, particularly after step 5. Fine-tuned models generally demonstrate more stable learning curves with less variance between epochs.

5. Feature Extraction Value: Simple CNN had to learn all features from scratch, while the ResNet50 fine-tuning approach leverages ImageNet pre-training, allowing it to utilize general image features and focus computational resources on domain-specific adaptations.

6. Parameter Efficiency: The CNN model would have required extensive hyperparameter tuning as shown in previous charts, while fine-tuning typically requires optimizing fewer parameters (primarily learning rate, number of layers to fine-tune, and regularization strength)

Question 4 (10 Marks)

Paste a link to your GitHub code for Part B

Example: https://github.com/Rajnishmaurya/da6401_assignment2/tree/main/partB

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

(UNGRADED, OPTIONAL) Part C : Using a pre-trained model as it is

Question 1 (0 Marks)

Object detection is the task of identifying objects (such as cars, trees, people, animals) in images. Over the past 6 years, there has been tremendous progress in object detection with very fast and accurate models available today. In this question you will use a pre-trained YoloV3 model and use it in an application of your choice. Here is a cool demo of YoloV2 (click on the image to see the demo on youtube).

Go crazy and think of a cool application in which you can use object detection (alerting lab mates of monkeys loitering outside the lab,

detecting cycles in the CRC corridor, ...).

Make a similar demo video of your application, upload it on youtube and paste a link below (similar to the demo I have pasted above).

Also note that I do not expect you to train any model here but just use an existing model as it is. However, if you want to fine-tune the model on some application-specific data then you are free to do that (it is entirely up to you).

Notice that for this question I am not asking you to provide a GitHub link to your code. I am giving you a free hand to take existing code and tweak it for your application. Feel free to paste the link of your code here nonetheless (if you want).

Example: https://github.com/<user-id>/da6401_assignment2/partC

Self Declaration

I, Rajnish Maurya (Roll no: DA24M015), 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.