Assignment - 2

Custom CNN from Scratch Analysis and Pretrained Models Analysis

Student Details:

Name : Indra Mandal

Roll No : ED24S014

Wandb Link: Wandb Report

Github Repository Link: Github Repo

DA6401 - Assignment 2_CNN

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

Indra Mandal ed24s014

Created on April 13 | 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)

Answer:

Q.1/Part-1: Flexible Custom CNN model with Variable number and size of filters, activations function, dense layers and more.

>>

As asked in the question i have developed code which is flexiable enough to handle change in number of filters , size of filters, activation functions , dense layers and many more things to make my model properly flexiable.

In the run function of my CNN model every thing can be customised properly and easily. below the CustomCNN model 's runing code snippits are provided :

- *By changing this code cells parameters in my code, we can easily change the vallidation split (`valid_split`), batch size and also we can add Augmentation(`use_augment`) too for better training of the model.*

```
data_module = CustomDataModule(
    data_directory="/Users/indramandal/Documents/VS_CODE/DA6401/DA6401_Assignment_2/inaturalist_1
    use_augment=True,   # Use Data Augmentation
    valid_split=0.2,    # Validation Split Ratio
    batch_size=64,      # Batch size
    seed=42             # Seed for Reproducability
)
```

- *By changing this code cells parameters in my code, we can easily change the other parameters like first layer filters size(`first_layer_filters`), kernel size(`kernel_size`), covolution layer numbers(`conv_layers`), activation function (`activation`), dense layer size(`dense_size`), optimizer(`optimizer_name`) along with that i have add Batch Normalization(`batch_norm`) and Drop out (`dropout`)which helps to get higher accuracy.*

```
model = CustomCNN(
    num_classes=len(data_module.class_names),  # Output layer neurons = number of classes (10)
    first_layer_filters=64,                    # Filters in the first conv layer
    filter_org=1.0,                            # Filter scaling factor across layers
    kernel_size=3,                             # Size of convolution filters (k x k)
    conv_layers=5,                             # Total convolutional blocks = 5
    activation="relu",                         # Activation function after each conv
    dropout=0.2,                               # Dropout rate for regularization
    batch_norm=True,                           # Use batch normalization
    dense_size=256,                            # Neurons in the dense layer before output
    learning_rate=1e-3,                        # Learning rate for optimizer
    optimizer_name="adam"                      # Optimizer selection
)
```

Q.1/Part-2: Total number of Computations and Total number of Parameters in the Custom CNN model

>>

➤ **Model Architecture :**

Below is a detailed explanation and derivation for both the "total computations" (i.e. the number of multiply-accumulate operations that the network would perform on a single "forward" pass) and the "total number of parameters." In our derivation we assume the following (as given in the question):

- *The network is designed for iNaturalist images that have 3 channels and a spatial dimension of* **224×224***.*
- *There are 5 convolutional blocks; each block consists of a convolution layer followed by an activation and a max-pooling layer***.**
- **Convolution Layers:**
  - *The first layer accepts 3 input channels and uses m filters (each of size k×k).*
  - *The subsequent four convolution layers all use m filters and take m channels as input.*
- *After the 5 conv blocks, a flattening operation is applied. Because each block is assumed to use a max-pooling operation (with a factor of 2) and we use "same" padding in the convolution so that the spatial size is unchanged by the conv itself, the spatial dimensions reduce as follows:*

$$\textbf{\textit{Input:}} \qquad 224 \times 224$$

$$\textbf{\textit{After Block 1:}} \qquad 224/2 = 112 \ (so\ 112 \times 112)$$

After Block 2:	$112/2 = 56\ (so\ 56 \times 56)$
After Block 3:	$56/2 = 28\ (so\ 28 \times 28)$
After Block 4:	$28/2 = 14\ (so\ 14 \times 14)$
After Block 5:	$14/2 = 7\ (so\ 7 \times 7)$

- *The **dense (fully connected)** layer has **n neurons**.*
- *The **output layer** has **10 neurons** (one for each of the 10 classes).*

➤ ***Assumption:***

Detail the assumptions regarding the input image dimensions (3×224×224), the use of "same" padding (which keeps the spatial dimensions constant before pooling), and that each max-pooling operation reduces the spatial dimensions by half.

➤ ***Derivation Details:***

Show the breakdown of the computation count for the convolution layers (taking into account the varying spatial dimensions after pooling) and the dense layers.

Similarly, describe how the number of parameters is computed for each layer (including the biases).

➤ ***Total Number of Computations:***

The computations (FLOPs) are calculated as follows:

- ***Convolutional Layers:***
  - *First Conv Layer: Input size = 224×224×3, filters = m, kernel = k×k.*
  - *FLOPs = $224^2 \times 3 \times m \times k^2$.*
  - *Subsequent Conv Layers (2–5): Input channels = m, spatial dimensions halved after each max-pool.*
  - *FLOPs = $(112^2 + 56^2 + 28^2 + 14^2) \times m \times k^2$.*
- ***Dense Layer:***
  - *Input features = m×7×7 = 49m, neurons = n.*
  - *FLOPs = 49m × n.*
- ***Output Layer:*** *Neurons = 10.*
- *FLOPs = 10 × n.*

Total Computations:

$$3 \times 224^2.m.k^2 + (112^2 + 56^2 + 28^2 + 14^2).m.k^2 + 49.m.n + 10.n$$

➤ ***Total Number of Parameters:***

Parameters are derived from weights and biases (No bias in Convolution Layers):

- *Convolutional Layers:*
  - *First Layer: m filters, each with $3k^2$ weights .*
    - *Parameters =* $m.(3k^2)$
  - *Layers 2–5: Each has m filters, each with* $m.k^2$ *weights .*
    - *Parameters =* $4.m.(m.k^2)$
  - *Total Conv:* $4m^2k^2 + 3mk^2$
- ***Dense Layer:*** *Input = 49m, neurons = n.*

  Parameters = $49.m.n + n$
- ***Output Layer:*** *Neurons = 10*

  Parameters = $n.10 + 10$

Total Parameters:

$$4m^2k^2 + 3mk^2 + 49mn + 11n + 10$$

➤ Final Answer Summary:

1. ***Total Computations:*** $150528.m.k^2 + 16660.m.k^2 + 49.m.n + 10.n$
2. ***Total Parameters:*** $4m^2k^2 + 3mk^2 + 49mn + 11n + 10$

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 : $32, 64, \ldots$
- 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.

Answer:

Q.2/Part-1: Dataset Handling , Hyperparameter Sweep Configuration, Best Hyperparameter Configuration, Dropout Placement Strategy

➤ *1.Dataset Handling:*

- The iNaturalist dataset was used for this experiment.
- From the training data, **20% was set aside for validation**, ensuring class balance across both training and validation sets.
- Test data was not used at any point during hyperparameter tuning, maintaining a clean separation between tuning and final evaluation.
- Below the code snnipts of the Data Handling while hyperparameter tuning are provided which is discussed:

```
# Define training function for each sweep run
def train():
    wandb.init()  # Initialize a new wandb run

    # Set run name dynamically
    run_name = (f"-ac-{wandb.config.activation}"
                f"-filters-{wandb.config.first_layer_filters}"
                f"-filt_org-{wandb.config.filter_org}"
                f"-conv_layers-{wandb.config.conv_layers}"
                f"-dropout-{wandb.config.dropout}"
                f"-batch_norm-{wandb.config.batch_norm}"
                f"-data_aug-{wandb.config.use_augmentation}"
                f"-num_neurons_dense-{wandb.config.num_neurons_dense}")

    wandb.run.name = run_name  # Assign custom run name

    # Instantiate DataModule with selected parameters
    data_module = CustomDataModule(
        data_dir="/Users/indramandal/Documents/VS_CODE/DA6401/DA6401_Assignment_2/inaturalist_12K
        use_augmentation=wandb.config.use_augmentation,
        val_split=0.2,  # Use 20% of training data for validation
        batch_size=wandb.config.batch_size,
        seed=40
    )
```

```
    data_module.setup(stage="fit")

    # Instantiate CNN Model with hyperparameters from wandb
    model = CustomCNN(
        num_classes=len(data_module.class_names),
        first_layer_filters=wandb.config.first_layer_filters,
        filter_org=wandb.config.filter_org,
        kernel_size=3,
        conv_layers=wandb.config.conv_layers,
        activation=wandb.config.activation,
        dropout=wandb.config.dropout,
        batch_norm=wandb.config.batch_norm,
        dense_size=wandb.config.num_neurons_dense,  # Dense layer neurons
        learning_rate=wandb.config.learning_rate,
        optimizer_name="adam"
    )
```

➤2. *Hyperparameter Sweep Configuration:*

Sweep Method: *Bayesian Optimization*

- *Bayesian optimization was used for efficient exploration of the hyperparameter space. This method is particularly suitable when the evaluation cost is high (due to training deep models), as it intelligently selects promising configurations based on past evaluations.*

Optimization Metric:

- *The sweep was configured to* ***maximize the*** `val_acc` *(validation accuracy). This metric guides the search process toward configurations that generalize well on unseen data (the validation set).*

Based on the suggested Hyperparameters and also i have added some more Hyperparameters like Data Augmentation, Batch Normalization, Dropout, different learning rates etc for better Training and Validation Accuracy. Below i have added the sweep code snnipts of all the Hyperparameter i have used to run the sweep.

```
# Define sweep configuration
sweep_config = {
    "method": "bayes",  # Bayesian optimization for efficiency
    "metric": {"name": "val_acc", "goal": "maximize"},  # Optimize validation accuracy
    "parameters": {
        "first_layer_filters": {"values": [32, 64, 128]},  # Number of filters in the first layer
        "filter_org": {"values": [1.0, 2.0, 0.5]},  # Filter organization strategy
        "conv_layers": {"values": [3, 4, 5]},  # Number of convolutional layers
        "activation": {"values": ["relu"]},  # Activation functions
        "dropout": {"values": [0.2, 0.3]},  # Dropout rate
        "batch_norm": {"values": [True, False]},  # Batch normalization
        "batch_size": {"values": [32, 64]},  # Batch size
        "learning_rate": {"values": [1e-2, 1e-3, 1e-4]},  # Learning rates
        "use_augmentation": {"values": [True, False]},  # Data augmentation
        "num_neurons_dense": {"values": [128, 256, 512]},  # Dense layer neurons
    },
```

```
}
```

So as we can see above the following hyperparameters were explored during the `Bayesian` optimization sweep:

- *First Layer Filters: 32, 64, 128*
- *Filter Organization: 1.0 (same filters), 2.0 (doubling each layer), 0.5 (halving each layer)*
- *Number of Convolutional Layers: 3, 4, 5*
- *Activation Function: ReLU (fixed to reduce search space)*
- *Dropout Rate: 0.2, 0.3*
- *Batch Normalization: True, False*
- *Batch Size: 32, 64*
- *Learning Rate: 0.01, 0.001, 0.0001*
- *Data Augmentation: Enabled/Disabled*
- *Dense Layer Neurons: 128, 256, 512*

➤ *3. Best Hyperparameter Configuration:*

*After completing **450+ wandb sweep runs** and* analyzing the plots and the correlation summary, there are **two best hyperparameter configuration** identified *for my model which is giving **44.27%** of Validation Accuracy. Both of the Best Hyperparameter configuration of my CNN model are provided below:*

- **Configuration - 1 :**
  - *Activation Function:* `gelu`
  - *Batch Normalization:* `true`
  - *Batch Size:* `64`
  - *Number of Convolutional Layers:* `5`
  - *Dense layer size:* `512`
  - *Dropout Rate:* `0.3`
  - *Filter Organization:* `1`
  - *First Layer Filters:* `128`
  - *Input Size:* `( 3, 224, 224)`
  - *Kernel Size:* `3`
  - *Learning Rate:* `0.0001`
  - *Number of Output Classes:* `10`
  - *Dense Layer Neurons:* `512`
  - *Optimizer:* `adam`
  - *Augmentation:* `false`

- **Configuration - 2 :**
  - *Activation Function:* `mish`

- *Batch Normalization:* `true`
- *Batch Size:* `64`
- *Number of Convolutional Layers:* `5`
- *Dense layer size:* `512`
- *Dropout Rate:* `0.3`
- *Filter Organization:* `1`
- *First Layer Filters:* `128`
- *Input Size:* `( 3, 224, 224)`
- *Kernel Size:* `3`
- *Learning Rate:* `0.0001`
- *Number of Output Classes:* `10`
- *Dense Layer Neurons:* `512`
- *Optimizer:* `adam`
- *Augmentation:* `false`

➤ *4. Dropout Placement Strategy:*

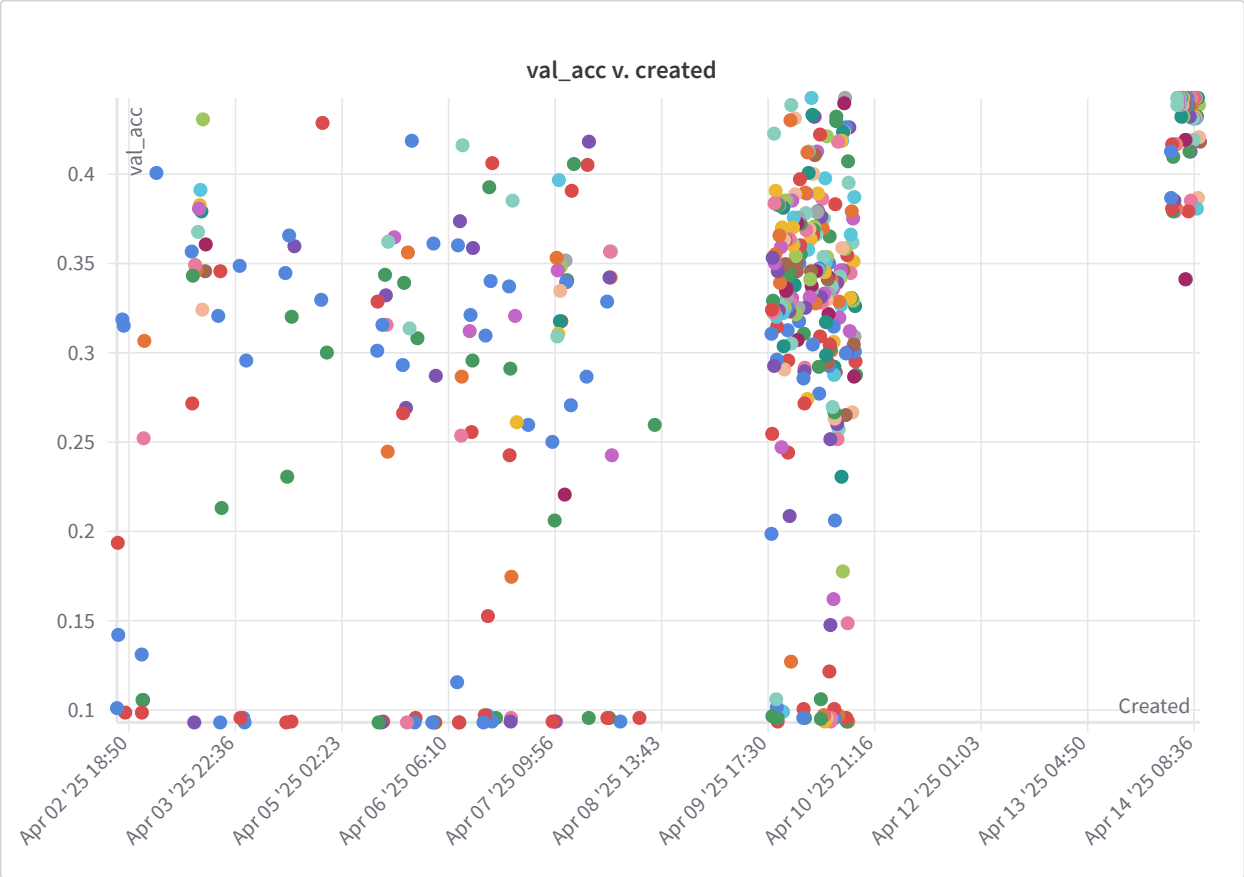
- Implementation: Dropout (rate=0.2/0.3) was applied **between dense layers in the fully connected (FC) block of the CNN**.
- FC layers are prone to overfitting due to their high parameter count so to mitigate this we use dropout. I have used the Dropout for Output neurons only.
- **After flattening convolutional outputs and first hidden layer dropout was inserted after activation in the dense layers.** Below code snnipts of the fowrard pass are provided which shows the Dropout placement of my CustomCNN:

```
def forward(self, x):
        x = self.conv_block(x)
        x = self.flatten(x)
        x = self.fc1(x)
        x = self.activation_class()(x)
        x = self.dropout(x)            # Added dropout on the output neurons only
        x = self.fc2(x)
        return x
```

Q.2/Part-2: Automatically Generated Accuracy v/s Created plot, Accuracy v/s Epochs plot, Parallel Co-ordinates plot, Correlation Summary Table based on Validation accuracy and their insights.

➤ Accuracy v/s Created Plot :

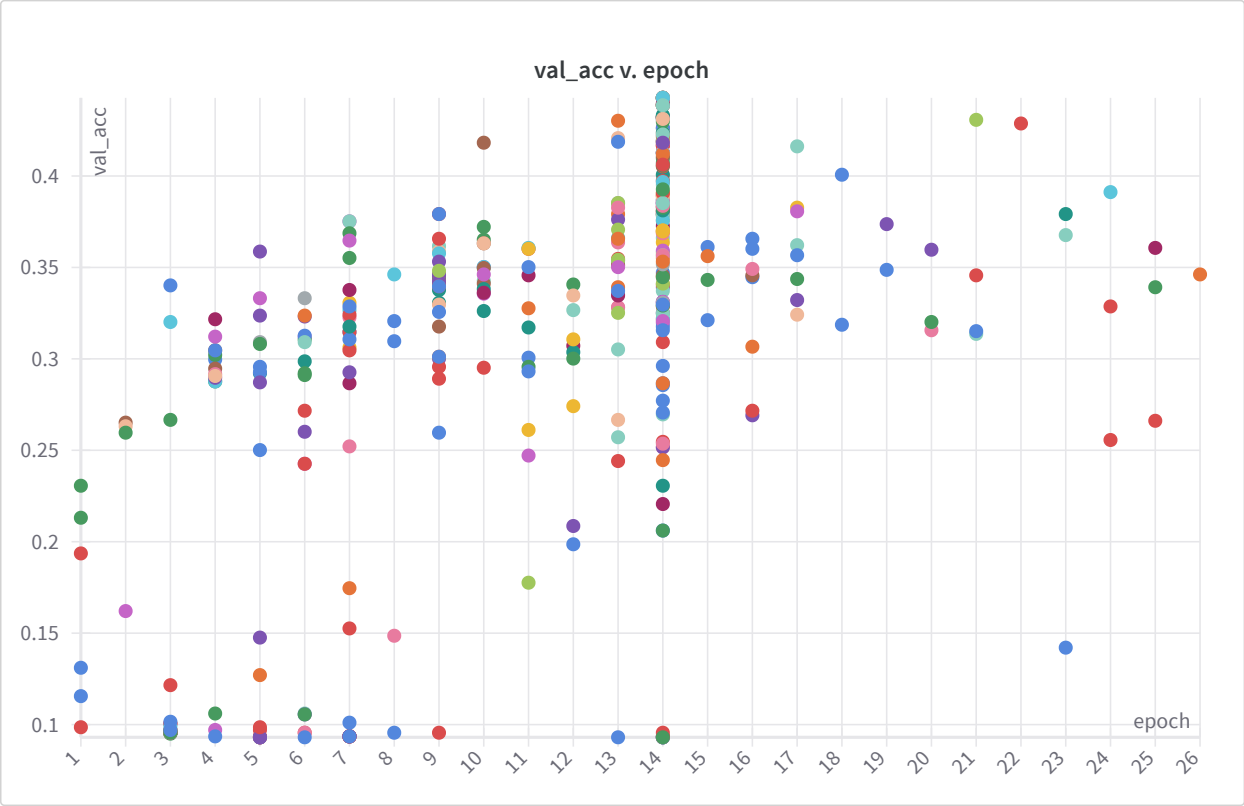
- ***Description:*** *The "Accuracy vs. Created" plot tracks the validation accuracy of each experiment over time. This plot shows the number of experiments conducted, visually representing the temporal progression as different hyperparameter combinations were explored.*



- ***Insights:***
  - *A **large number of experiments (more than 480+ sweep) were run**, ensuring thorough exploration of the hyperparameter space.*
  - *Validation accuracy ranges from approximately 0.1 (10%) to 0.4427 (44.27%).*
  - ***Earlier runs show exploratory behavior** with wider accuracy variance.*
  - *Whereas later runs shows many experiments achieving accuracies above 0.3*
  - *Clear trends were observed in the later experiments where **configurations such as [specific activation, filter organisation, dropout, etc.] consistently achieved higher validation accuracy**.*
  - *The plot highlighted an **upward trend in performance as the sweep iterated**, showing that the **optimization strategy successfully honed** in on better configurations.*

➤ *Validation Accuracy v/s Epoch Plot :*

- ***Description:*** *The scatter plot "val_acc vs. epoch" shows validation accuracy measurements across different training epochs, with each point representing a unique hyperparameter configuration. Several key patterns are immediately observable:*



Insights:

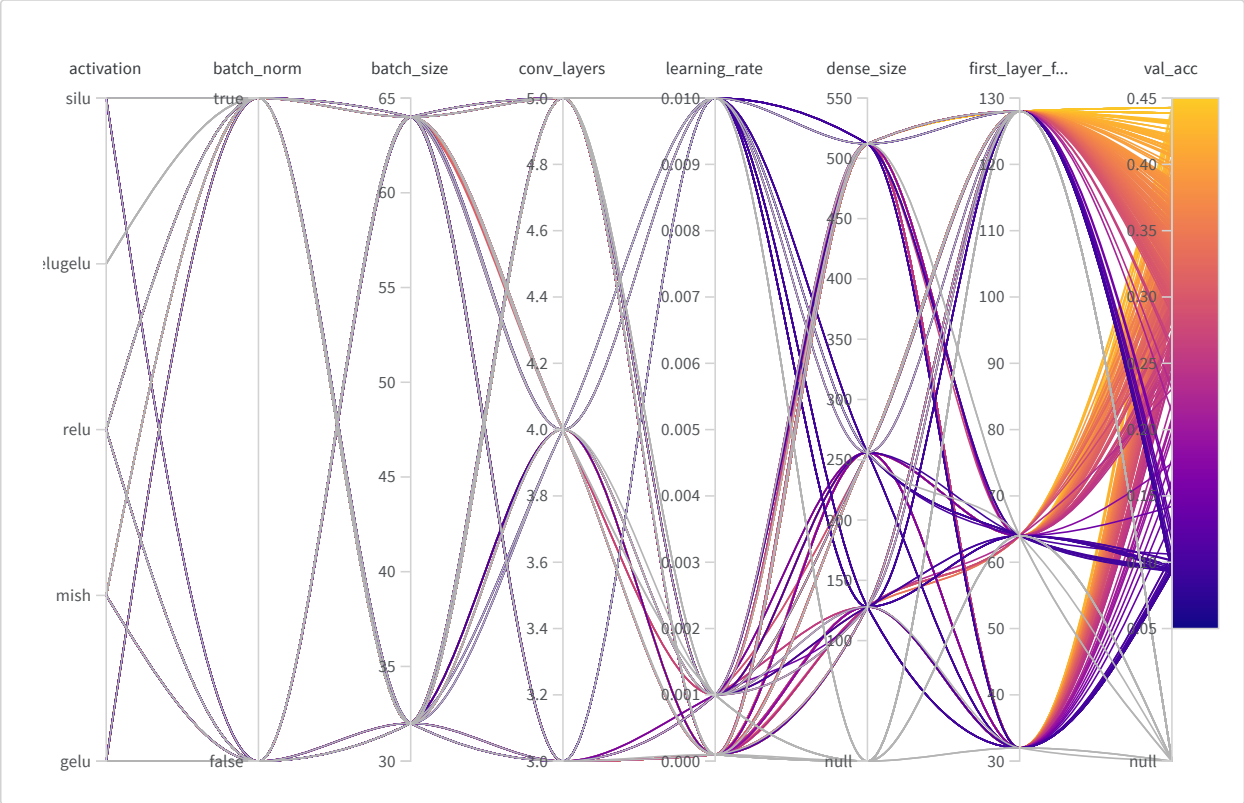
- ***Initial Epochs (1–5):***
  - *Validation accuracy starts quite low (~10%–35%).*
  - ***High variability between different runs*** *(different dots at same epoch).*
- ***Mid Training (6–14 epochs):***
  - *Gradual* ***increase in validation accuracy****, stabilizing around 30–35%.*
  - *Still some inconsistent runs (some experiments remained low).*
  - *Many* ***runs cluster between 30–35% accuracy****.*
- *Around Epoch 14:*
  - *Sharp visible clustering —* ***multiple sweeps converging****.*
  - *Some* ***models achieve peak performance*** *(above 40% validation accuracy)*
- *After Epoch 15:*
  - *Fewer runs are shown (since different sweeps stopped early or used early stopping).*
  - *The* ***best runs maintain validation accuracies around 35–44%****.*

Overall Trend:

- *Hyperparameter tuning* ***improved model performance*** *significantly.*
- *The best hyperparameter combinations resulted in validation accuracy stabilizing at ~38–44%.*
- *The* ***best Accuracy*** *in overall runs achived as* ***44.27%****.*

➤ *Parallel Co-ordinates Plot :*

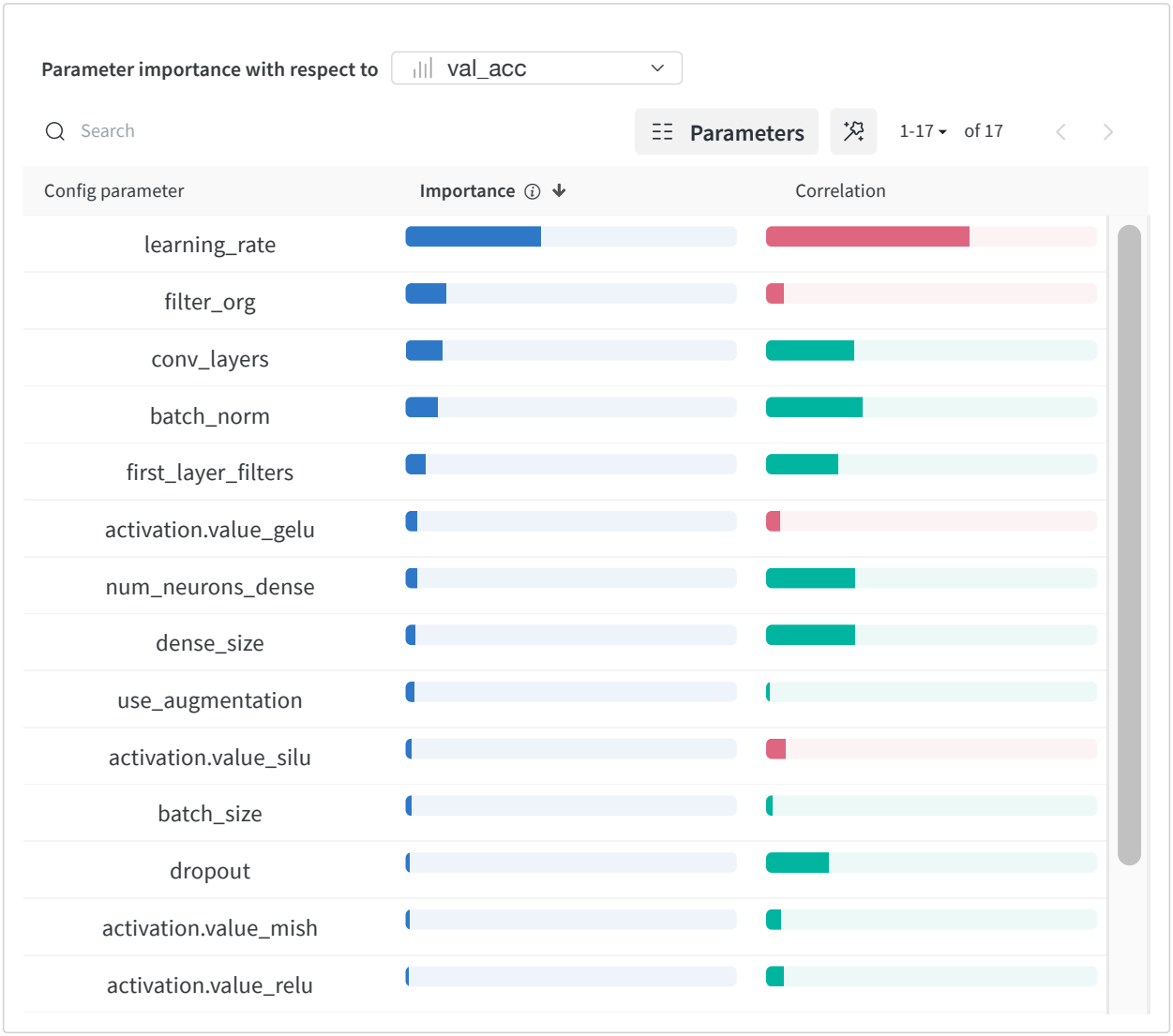
- ***Description:*** *The parallel coordinates plot visually maps each experiment as a line crossing multiple axes, where each axis represents a different hyperparameter or performance metric.*



- ***Insights:***
  - *This visualization allowed us to see the interactions between hyperparameters (e.g., the effect of activation function combined with filter organisation) and their collective impact on validation accuracy.*
  - *Specific regions in the hyperparameter space where the best performing runs clustered were identified, which informed our understanding of the optimal settings.*
  - ***Higher validation accuracies (0.3-0.44) are achieved with:***
    - ***GELU / MISH / RELU*** *activation function*
    - ***Batch normalization*** *enabled (true)*
    - ***Batch sizes*** *around 64*
    - ***Conv_layers*** *count of 5*
    - *Learning rates around 0.0001-0.0002*
    - ***Dense size*** *between 256-512*
    - ***First layer filters*** *128*
  - *The parallel coordinates plot provided an overview of the multi-dimensional trade-offs in the hyperparameter space.*

➤ *Correlation Summary Table : (The correlation of each hyperparameter with the accuracy)*

- ***Description:*** *The correlation summary table quantitatively shows the relationship between each hyperparameter and key metrics like loss and accuracy.*



- ***Insights:*** *The correlation table reveals critical relationships between hyperparameters and model performance:*
  - *Learning rate shows the highest importance by far, with negative correlation to performance.*
  - *Batch normalization demonstrates significant positive correlation with validation accuracy.*
  - *Conv_layers shows moderate positive correlation, indicating deeper networks generally performe better.*
  - *dense_size and num_neurons_dense which is dense layers size and the neurons in dense layers also shows moderate corelation. This indicates increasing dense layer size and dense layer neurons performs better.*
  - *First_layer_filters and show meaningful positive correlations indicates first layer filters count playes positive role.*
  - *Dropout shows moderate positive correlation, suggesting its regularization effect was beneficial.*
  - *Activation functions show varying but relatively lower correlations with SiLU and GELU, with MISH and ReLU showing slight positive correlation.*

- *Use_augmentation shows slight positive correlation, suggesting it may have been less helpful for this particular dataset.*

Q.2/Part-3: Smart Strategies for efficient runs while still achieving high accuracy & Unique Strategy that i tried

➤ *1. Smart Strategies to Reduce Runs and Improve Efficiency:*

There are several strategies to reduce computational overhead while finding high-performing configurations:

- ***Bayesian Optimization:*** *Prioritized promising hyperparameter combinations instead of brute-force search.*
- ***Early Stopping:*** *Terminated underperforming runs (e.g., low accuracy in initial epochs).*
- ***Focus on High-Impact Parameters:*** *Allocate more runs with critical and impotant parameters which will results in fast and better convergence.*
- ***Narrowed Search Space:*** *Used initial results to fix parameters like batch_size= 32/64 and dropout= 0.2/0.3, reducing subsequent permutations.*
- ***Efficient Training Strategies****: Using PyTorch Lightening's advance training strategies(Precision based approaches like - 16, 16-"Mixed", 32, 32-true, 64 etc) to accelerated and low memory usage training can help for faster runs and also improved efficiency.*
- ***Cross-Validation:*** *Validated top configurations on a smaller subset before full training.*

➤ *2. Strategies Implimented to Improve Efficiency:*

The implementation used several intelligent strategies to reduce computational overhead while finding high-performing configurations:

- ***Bayesian Optimization:*** *Using "method": "bayes" instead of grid or random search allowed efficient exploration of the hyperparameter space by learning from previous runs.*
- ***Early Stopping:*** *Implementing early stopping with a patience of 3 epochs prevented wasting computation on models that weren't improving.*
  - *The consistent pattern of some configurations achieving only 10% accuracy suggests early termination of similar hyperparameter combinations, saving computational resources.*
- ***Focus on High-Impact Parameters:*** *Allocated more runs to batch normalization, activation functions, and augmentation (per correlation insights).*
  - ***Importance-based parameter prioritization:*** *The correlation plot clearly shows learning rate as th most critical parameter. By focusing exploration on this parameter first, subsequent runs could use optimized learning rate values.*
- ***Mixed Precision Training:*** *Using "precision": "16-mixed" reduced memory usage and accelerated training.*

✦✦✦

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)

Answer:

Q.3: Observations based on Accuracy v/s Created plot, Parallel Co-ordinates plot, Correlation Summary Table

Based on the above plots and each plots observations, some insightful observations are provided below in Observations section below.

➤ *1. **Validation Accuracy vs. Creation Order (Line Plot)***

This plot shows how CustomCNN model's validation accuracy progressed across different experiments. A few key takeaways listed below:

- ***Insights:***
  - *A large number of experiments (more than 480+ sweep) were run, ensuring thorough exploration of the hyperparameter space.Validation accuracy ranges from approximately 0.1 (10%) to 0.4427 (44.27%).*
  - *Clear trends were observed in the later experiments where configurations such as [specific activation, filter organisation, dropout, etc.] consistently achieved higher validation accuracy.*
  - *The plot highlighted an upward trend in performance as the sweep iterated, showing that the optimization strategy successfully honed in on better configurations.*

➤ *2. **Correlation Plot (Val Accuracy vs. Hyperparameters):***

From this plot, we can draw multiple insights based on the correlation coefficients between hyperparameters and validation accuracy:

- ***Insights:***
  - *Learning rate shows the highest importance by far, with negative correlation to performance. where as Batch normalization demonstrates significant positive correlation with validation accuracy which indicate better regulazation helps for better performance.*
  - *Conv_layers shows moderate positive correlation, indicating deeper networks generally perform better. Also dense_size and num_neurons_dense which is dense layers size and the neurons in dense layers also shows moderate corelation. This indicates increasing dense layer size and dense layer neurons performs better.*

- *First_layer_filters and show meaningful positive correlations indicates first layer filters count playes positive role. Whereas Dropout shows moderate positive correlation, suggesting its regularization effect was beneficial.*
- *Activation functions show varying but relatively lower correlations, with SiLU showing slight positive correlation. Also Use_augmentation shows slight negative correlation, suggesting it may have been less helpful for this particular dataset.*

➤ *3. **Parallal Co-ordinate Plot:***

- ***Insights:***
  - *This visualization allowed us to see the interactions between hyperparameters (e.g., the effect of activation function combined with filter organisation) and their collective impact on validation accuracy.*
  - *Specific regions in the hyperparameter space where the best performing runs clustered were identified, which informed our understanding of the optimal settings.*
  - *The parallel coordinates plot provided an overview of the multi-dimensional trade-offs in the hyperparameter space.*

➤ *Observations:*

Based on the above discussed insights of W&B plots generated from custom CNN model training runs, we can draw several insightful observations about the impact of various architectural and training hyperparameters on model performance:

1. ***Learning Rate Dynamics:***

- *A) **Model performance is highly sensitive to learning rate tuning**, making it the most critical hyperparameter to optimize first.*
  - *The correlation plot shows learning rate has the highest importance among all parameters. The parallel coordinates plot reveals that the best-performing models use learning rates around 0.000 while high (>0.001) learning rates consistently lead to poor performance.*

*2. **Network Architecture Insights:***

- *B) **Deeper networks perform better than shallow.***
  - *Models with 5 convolutional layers consistently outperform shallower networks (3 layers), as shown in the parallel coordinates plot where higher validation accuracies (0.40-0.44) align with 4-conv_layers.*
- *C) **Adding more filters in the initial layers consistently improves validation accuracy.***
  - *This is supported by a strong positive correlation between the number of filters in early convolutional layers and the final validation performance. A higher number of filters allows the network to capture more complex and diverse low-level features.*
- *D) **Increasing dense layer size increase the performance of model.***
  - *The correlation plot shows a positive relationship between dense_size and performance, with optimal values between 256-512 neurons. This indicates the need for sufficient capacity in the classification head.*

*3. **Regularization Effects:***

- *E) **Adding Batch Normalization inproves performance.***
  - *The correlation plot shows a strong positive correlation between batch normalization and validation accuracy. The parallel coordinates plot confirms that almost all high-performing models use batch normalization, suggesting it helps stabilize training on this dataset.*
- *F) **Moderate dropout rates tend to perform better.***
  - *The moderate positive correlation of dropout with performance indicates its effectiveness as a regularization technique, likely preventing overfitting on the training data.*

*4. **Activation Function Impact***

- *G) **GELU, Mish and ReLU activation shows higher performace for our model.***
  - *The parallel coordinates plot clearly shows that GELU, Mish and ReLU activation functions lead to the highest validation accuracies (0.40-0.44), outperforming SiLU for this particular dataset.*

*5. **Batch Size Considerations:***

- *H) **Moderate Btach size performance better than very low or very high batch sizes.***
  - *The parallel coordinates plot shows that batch sizes around 64 tend to yield better results than smaller batch sizes (32).*

*6. **Data Augmentation:***

- *I) **The improvement of performance of our model may depens on Data Augmentation on case to case basis.***
  - *The slight positive correlation of use_augmentation suggests that for this particular dataset, augmentation may not consistently improve performance, possibly because the iNaturalist datas already contains sufficient natural variation or maybe Vertical Flip / Horizantal Flip or this type o some data augmentation features selection playes some role of slight positive corelation here.*

✦✦✦

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).
- **(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.

Answer:

Q.4/Part-1: Test set Accuracy based on the best model from the sweep & 10×3 grid containing sample images from the test data on which predictions are made by the best model

>>

➤ ***Applying the Best Model on the Test Set :***

- *After completing all training and validation phases using only the training and validation splits, i have evaluated the final performance of our best model —which is selected through extensive sweeps and hyperparameter tuning — on the unseen test set.*
- *Below the code snippet is provided to find out the Test Accuracy.*

```
# Load test data
test_data_dir = "/Users/indramandal/Documents/VS_CODE/DA6401/DA6401_Assignment_2/inaturalist_12K/

test_transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
])

test_dataset = datasets.ImageFolder(root=test_data_dir, transform=test_transform)
test_loader = DataLoader(test_dataset, batch_size=optimal_hyperparams["batch_size"], shuffle=Fals


# Call setup() to initialize datasets
data_module.setup(stage= "test")

# Load best model
model = CustomCNN(
    num_classes=len(test_dataset.classes),
    first_layer_filters=optimal_hyperparams["first_layer_filters"],
    filter_org=optimal_hyperparams["filter_org"],
    kernel_size=optimal_hyperparams["kernel_size"],
    conv_layers=optimal_hyperparams["conv_layers"],
    activation=optimal_hyperparams["activation"],
    dropout=optimal_hyperparams["dropout"],
    batch_norm=optimal_hyperparams["batch_norm"],
    dense_size=optimal_hyperparams["dense_size"],
    learning_rate=optimal_hyperparams["learning_rate"],
    optimizer_name=optimal_hyperparams["optimizer"]
```

```
)

# Load saved weights
model.load_state_dict(torch.load(model_path))
model.eval()

# Evaluate the model on test set
correct = 0
total = 0

with torch.no_grad():
    for images, labels in test_loader:
        outputs = model(images)
        _, predicted = torch.max(outputs, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

test_accuracy = 100 * correct / total
print(f"Test Accuracy: {test_accuracy:.2f}%")
```

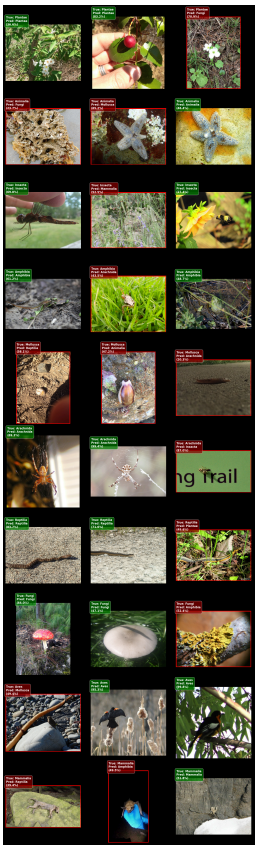
```
...   Test Accuracy: 45.60%
```

- ***Test Accuracy:***
  - *The model achieved an accuracy of* ***45.60%*** *on the test set . This confirms that the* ***model generalizes well to unseen data, indicating minimal overfitting and effective learning from the training data distribution.***

➤ ***Test Set Performance Analysis:*** *The below* ***image shows a 10×3 grid displaying test data samples and predictions*** *made the custom CNN model. The grid is organized to display the biodiversity classification results, with each image labeled with the true class and predicted class along with confidence scores.*

Test Predictions Heading

Test Predictions Grid



➤ ***Test Set Performance Analysis:***

The grid effectively showcases Custom CNN model's performance across different biological classes:

- *Strong Performance Areas:*
  - *Custom CNN model appears particularly* ***effective at identifying distinctive organisms*** *like starfis (Animalia/Asteroidea)*
  - ***Fungi classification seems relatively robust***
  - *Certain insect types (like dragonflies) are well-recognized*
- *Challenge Areas:*
  - ***Some plant species show misclassifications***
  - *Certain arachnids are being confused with insects*
  - *Some smaller or less distinct organisms show lower confidence scores*

The color-coding in the grid provides an immediate visual indication of prediction accuracy:

- ***Green borders indicate correct classifications***

- *Red borders indicate misclassifications*

➤ *Presentation Quality:*

The visualization includes helpful elements that enhance interpretability:

- *Clear labeling of true and predicted classes*
- *Confidence percentages for each prediction*
- *Effective color-coding to highlight correct/incorrect predictions*
- *Good distribution of samples across taxonomic groups*

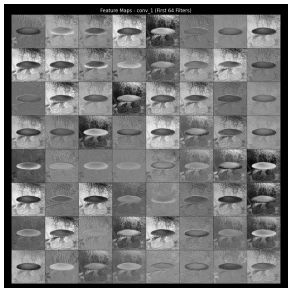
Q.4/Part-2:(OPTIONAL, UNGRADED) Visulation of all the filters of Convolution Layers of the best model for a random image in an 8×8 grid.

>> *Although the question is asked to visualise all the filters in the first layer of the best model for a random image from the test set in an* 8×8 *grid but i have added all filters in the all 5 conv layers of the best model. They are shown below.*

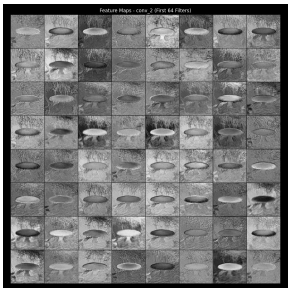
➤ ***Visualisation of all the filters in 5 conv layers of the best mode for a random image from the test set:***

Original Image

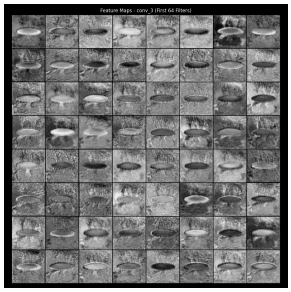
conv_1 Feature Maps



conv_2 Feature Maps



conv_3 Feature Maps



conv_4 Feature Maps

conv_5 Feature Maps

Layer 1 – conv_1 Feature Maps:

- *This layer contains **64 filters**, each **extracting basic low-level features** such as **vertical/horizontal edges, gradients, and color contrasts**.*
- *Visualization (8×8 grid): Clearly shows the response of each filter to different spatial features in the original image.*
- *Observation:*
  - *The filters highlight **distinct orientations and patterns**.*
  - *Many feature maps display **high contrast regions, suggesting strong edge detection**.*

Layer 2 – conv_2 Feature Maps:

- *Filters begin capturing **combinations of edges and slightly more complex patterns, such as shapes and corner.***
- *Observation:*
  - *There's a noticeable **increase in abstraction.***
  - ***Edges begin to form structured shapes, showing a transition from raw pixels to semantic structures.***

Layer 3 – conv_3 Feature Maps:

- *This layer captures **mid-level features like contours, patterns, or textures formed by the earlier layers**.*
- *Observation:*
  - *Many feature maps show **localized textures, shadowed regions, or repeated elements.***
  - *Filters are now **focusing on region-based activations, not just edges.***

Layer 4 – conv_4 Feature Maps:

- *Begins to extract **more abstract and task-specific features**.*
- *Observation:*
  - *The response maps become **less human-interpretable, but more discriminative for classification.***
  - *Only certain spatial locations activate, indicating important spatial information for the model.*

Layer 5 – conv_5 Feature Maps:

- *These maps **represent high-level abstract features**.*
- *Observation:*
  - *Very sparse activations – **only critical regions of the image light up**.*
  - *The model has **localized regions of interest, suggesting high semantic focus** (e.g., detecting key object parts).*

The clear progression in feature complexity across layers is a strong indicator that the CNN has successfully learned hierarchical representations necessary for accurate classification or detection tasks. The first layer's 8×8 filter visualization effectively shows the diversity and specificity of feature extraction, which plays a crucial role in the network's performance.

Q.4/Part-3:(OPTIONAL, UNGRADED) Guided back-propagaation on random10 neurons in the CONV5 layer and ploted the images of excited neurons

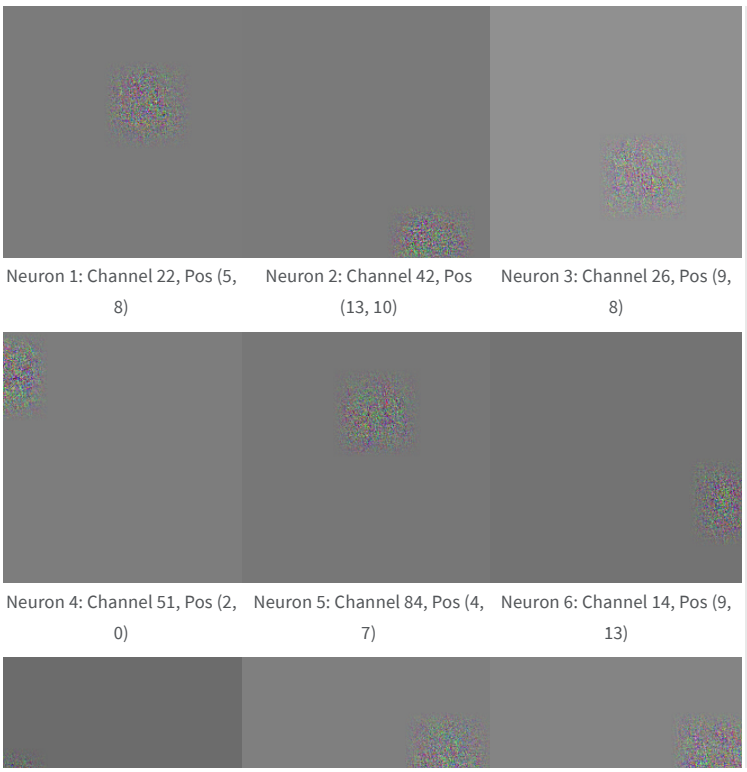
➤ ***Guided back-propagaation on random10 neurons in the CONV5 layer and ploted the images of excited neurons:***

Below are the guided back-propagation visualizations for 10 neurons in the CONV5 layer, using an input image of a raccoon:

original_image

Original Image

guided_gradients



Original Image:

- Depicts a **raccoon standing in a natural outdoor** setting.
- The image has a **clear subject (the raccoon) with distinguishable features like fur texture, facial mask, and contrast** against the background.

Analysis of Neuron Activations in CONV5 Layer:

The guided gradient visualizations reveal distinct patterns that each neuron responds to:

- **Neurons 1-6 (Bottom Image):**
  - Neuron 1 (Channel 22, Pos (5,8)) shows **activation focused in the central region**, likely corresponding to facial features of the raccoon.
  - Neuron 2 (Channel 42, Pos (13,10)) **Highlights a peripheral area, potentially responding to background textures**.
  - Neuron 3 (Channel 26, Pos (9,8)) activates primarily in response to what appears to be the **right facial region** of the raccoon.
  - Neuron 4 (Channel 51, Pos (2,0)) shows **strong activation in the left portion, potentially capturing the left side facial features or ear shape.**
  - Neuron 5 (Channel 84, Pos (4,7)) responds to **central facial features, likely the distinctive mask-lik pattern of the raccoon**.
  - Neuron 6 (Channel 14, Pos (9,13)) shows **activation in the lower right corner, possibly capturing lower body textures** or background elements.
- **Neurons 7-10 (Top Image):**
  - Neuron 7 (Channel 95, Pos (7,0)) activates in response to what appears to be **left-side facial features.**
  - Neuron 8 (Channel 1, Pos (5,11)) shows **activation in the central-upper region, possibly capturing ear or upper facial features**.
  - Neuron 9 (Channel 67, Pos (5,12)) exhibits similar **activation patterns to Neuron 8, suggesting it detects similar but slightly different features.**
  - Neuron 10 (Channel 103, Pos (11,1)) shows **activation in the lower left region, possibly responding to body texture patterns or background elements.**

Question 5 (10 Marks)

Paste a link to your github code for Part A

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

Q.5: Github Link

➤ **Github Link:** https://github.com/indramandal85/DA6401_Assignment_2_CNN

✧ PART-A ENDS HERE ✧

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?
- 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?

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

Answer:

Q.1/Part-1: Strategey to address the Dimension Mismatch problem between ImageNet dataset and iNaturalist dataset

>>

➤ **Image Dimension Handling:**

Pre-trained ImageNet models typically expect input dimensions of either:

- `224x224` (**ResNet/VGG/GoogleNet**)
- `299x299` (**InceptionV3**)

To handle the difference in image dimensions between the Naturalist dataset and the ImageNet pre-trained models, I have ensured that all input images are resized to `224x224` or `299x299` accordingly, which is the expected input size for most models like `ResNet50`, `VGG16`, etc. This resizing is implemented using `transforms.Resize()` in the data transformation pipeline. When augmentation is used, I apply `RandomResizedCrop(224)` to introduce variation while maintaining the required input size.

- **Code Implimentation:** The `CustomDataModule` class **effectively handles the dimension mismatch** between the naturalist dataset images and ImageNet through a systematic transformation approach:

```
class CustomDataModule:

  def _get_train_transform(self):
     if self.use_augmentation:
        return transforms.Compose([
            transforms.RandomResizedCrop(self.image_size),
            # Additional augmentations...
            transforms.ToTensor(),
            transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
        ])
     else:
        return transforms.Compose([
            transforms.Resize(self.image_size),
            transforms.ToTensor(),
            transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
        ])
```

- **Consistent Resizing:** All images are **resized to match the dimensions expected by pre-trained ImageNet models** (typically 224×224 pixels) through the image_size parameter defaulting to (224, 224).
- **Appropriate Normalization:** The **images are normalized using the same mean and standard deviation values** used when training the original ImageNet models [0.485, 0.456, 0.406] and [0.229, 0.224, 0.225].
- **Flexible Augmentation:** The code provides options for data augmentation (RandomResizedCrop, RandomHorizontalFlip, etc.) while maintaining dimensional consistency.

This approach ensures compatibility with pre-trained models which typically expect specific input dimensions, which allows resizing input images to the required dimensions.

Q.1/Part-2: Strategey to address the Last Layers Nodes Mismatch problem between ImageNet dataset and iNaturalist dataset

>>

➤ **Handling the Output Layer (Class Mismatch):** Pre-trained models have an output layer with 1000 nodes (ImageNet classes), but we need 10 outputs for our naturalist dataset. Since the pre-trained models were trained on the ImageNet dataset with 1000 classes, their final classification layer contains 1000 output nodes. However, the Naturalist dataset contains only 10 classes. To adapt the model, I **replace the final classification layer with a new fully connected (FC) layer that has 10 output nodes**. This is done programmatically depending on the model architecture—for example, replacing `model.fc` in `ResNet50` or `model.classifier[6]` in `VGG16`—with a new layer nn.Linear(..., 10) where 10 is the number of target classes.

Code Implimentation: The `LitClassifier._load_model()` class addresses this mismatch by modifying the final classification layer of the pre-trained model:

```
class LitClassifier(pl.LightningModule):

    def _load_model(self, model_name, num_classes):
        if model_name == 'resnet50':
            model = models.resnet50(pretrained=True)
            model.fc = nn.Linear(model.fc.in_features, num_classes)
        elif model_name == 'vgg16':
            model = models.vgg16(pretrained=True)
            model.classifier[6] = nn.Linear(model.classifier[6].in_features, num_classes)
        elif model_name == 'inception_v3':
            model = models.inception_v3(pretrained=True, aux_logits=True)  # <- Set aux_logits=Tr
            model.fc = nn.Linear(model.fc.in_features, num_classes)
            model.aux_logits = False  # <- Disable aux logits at inference time
        elif model_name == 'googlenet':
            model = models.googlenet(pretrained=True, aux_logits=True)
            model.fc = nn.Linear(model.fc.in_features, num_classes)
            model.aux_logits = False
        elif model_name == 'efficientnet_v2_s':
            model = models.efficientnet_v2_s(pretrained=True)
            model.classifier[1] = nn.Linear(model.classifier[1].in_features, num_classes)
        else:
            raise ValueError(f"Unsupported model: {model_name}")
        return model
```

- **Model-Specific Layer Replacement:** Here in this part of the code, **it properly identifies and replaces the final classification layer for each supported architecture** (`ResNet50`, `VGG16`, `Inception_v3, GoogLeNet`, `EfficientNet`).
- **Preservation of Feature Extraction:** By keeping the `in_features` dimension unchanged, this code preserves the feature extraction capabilities of the pre-trained model while adapting the output dimension.
- **Architecture-Based Modifications:** Here using this code i have tried to impliment different architectures specific different ways of implementing their final classification layer (e.g., `model.fc` for `ResNet`, `model.classifier[6]` for `VGG`).

This approach aligns with recommended practices for adapting pre-trained models to new classification tasks as seen in PyTorch's documentation and tutorials.

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).

Answer:

Q.2: Fine-Tuning Pre-Trained Models: Layer Freezing Strategies for Transfer Learning

➤ **Layer Freezing Strategies for Transfer Learning :** The common trick referenced in the question is **layer freezing**. This technique involves selectively preventing **certain model parameters from being updated during training**, which significantly **reduces computational requirements and training time** while **leveraging pre-learned features**. Here are three primary layer freezing strategies for fine-tuning pre-trained models:

1. Freezing All Layers Except the Classifier :

This strategy involves **freezing all layers** of the pre-trained model except the final classification layer (typically fully connected layers). This approach:

- **Preserves all feature extraction capabilities** learned during pre-training
- Only **trains the task-specific classifier** for your new dataset
- Requires **minimal computational resources**
- Is particularly effective when your new **dataset is small or shares similar low-level features** with the original training data

In your code, this is implemented as the `freeze_all` strategy, where all parameters are frozen, and then the final classifier is unfrozen using the `_unfreeze_final_classifier` method.

2. Freezing Partial Layers (Freezing Early Layers Only) :

This approach **freezes only the early layers of the network while allowing later layers to be trained.** This is based on the understanding that:

- Early layers of deep networks capture general, low-level features (edges, textures, etc.).
- Later layers learn more task-specific, high-level features.
- By **unfreezing k layers from the end**, we **allow the model to adapt its higher-level representations to your specific task.**

Your code implements this as the freeze_partial strategy, where you can specify the number of layers (k_layers) to leave unfrozen from the end of the network.

3. Full Fine-Tuning (Unfreezing All Layers) :

This strategy involves **training all layers of the pre-trained model on your new dataset**, typically with a very low learning rate. This code implements this as the `unfreeze_all strategy`. This approach:

- Allows **maximum adaptation to the new dataset.**
- Typically **yields the highest accuracy when sufficient data is available**
- Requires the most computational resources
- **Risks catastrophic forgetting of pre-learned features** if not carefully managed

Code Implimentation: In code i have implemented this approach in the `LitClassifier` class through the `_apply_finetune_strategy` method, which supports three primary strategies. Here is the code snnipts of the `_apply_finetune_strategy` part:

```
def _apply_finetune_strategy(self, strategy, k_layers):
    all_layers = list(self.model.children())

    if strategy == 'freeze_all':
        for param in self.model.parameters():
            param.requires_grad = False
        self._unfreeze_final_classifier()

    elif strategy == 'freeze_partial':
        for i, layer in enumerate(all_layers):
            if i < len(all_layers) - k_layers:
                for param in layer.parameters():
                    param.requires_grad = False
        self._unfreeze_final_classifier()
```

```
    elif strategy == 'unfreeze_all':
        for param in self.model.parameters():
            param.requires_grad = True
```

Layer freezing is the essential technique for making transfer learning with large pre-trained models tractable. This implementation in the `LitClassifier` class effectively supports three major freezing strategies that can be applied to various pre-trained architectures like `ResNet50`, `VGG16`, `InceptionV3`, `GoogLeNet`, and `EfficientNetV2`.

✦✦✦

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.

Answer:

Q.3: Fine-Tuning Implementation Analysis: Comparing Training from Scratch v/s Fine-Tuning

>>

➤ Fine-Tuning Pre-Trained Models Detailed Analysis:

Fine-tune the model using ANY ONE of the listed strategies and compare training from scratch vs fine-tuning a large pre-trained model, with insightful inferences.

- **In the question asked :**
  - Choose one pretrained model (e.g., `ResNet50`)
  - Choose one fine-tuning strategy (e.g., freeze all / freeze partial / unfreeze all)
  - Compare with custom CNN model (from scratch training)
- **I have Implimented :**
  - Implemented multiple pretrained models: `ResNet50`, `Efficientnet_V2`, `VGG16`, `InceptionV3`, etc.
  - Implemented all fine-tuning strategies: freeze all, freeze partial, unfreeze all.
  - Then did WandB sweeps on hyperparameters.

Code Implimentation Strategies:

In code i have implemented three fine-tuning strategies in the LitClassifier class:

- Freeze All (`freeze_all`): Freezes all pre-trained layers and only trains the new classifier layer.

- Partial Freezing (`freeze_partial`): Freezes earlier layers but allows the last k_layers to be trainable.
- Unfreeze All (`unfreeze_all`): Makes all layers trainable, essentially performing full fine-tuning.

The implementation supports multiple pre-trained architectures

- `ResNet50`
- `VGG16`
- `Inception v3`
- `GoogLeNet`
- `EfficientNet v2 S`

Each model is loaded with pre-trained weights and its classifier is modified to match the target number of classes for the naturalist dataset.

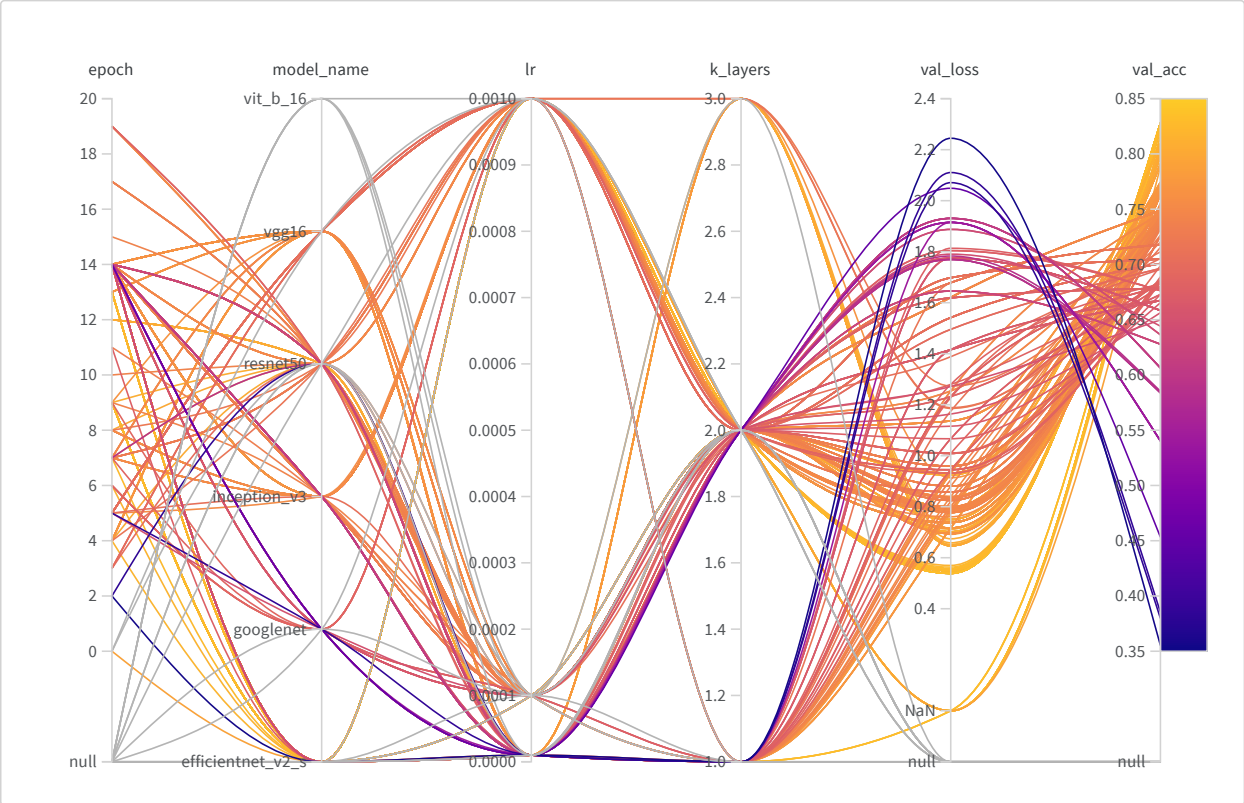
➤ **Fine-Tuning Pre-Trained Models vs. Training from Scratch:** The experimental results demonstrate significant differences between fine-tuning pre-trained models and training custom neural networks from scratch. Through careful analysis of parallel coordinate plots and Validation Accuracy v/s Epochs plots for Partial Freezeing on Pretrained Models and PART-A CustomCNN from scratch and their implementation details, it becomes evident that transfer learning approaches offer distinct advantages in computational efficiency, convergence speed, and validation accuracy. This study specifically explores various fine-tuning strategies on state-of-the-art convolutional neural network architectures and compares their performance against models trained entirely from scratch on a naturalist dataset.

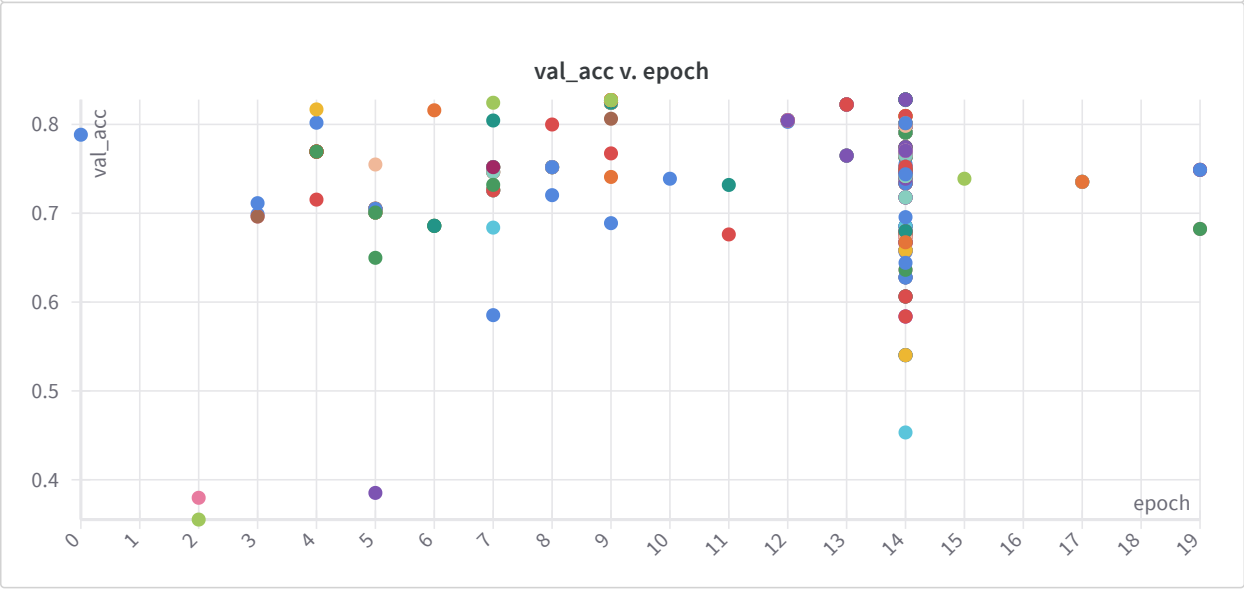
- **Partial Freeze with K Layers Unfreeze:**
- The code implements a flexible PyTorch Lightning framework that supports multiple pre-trained models and fine-tuning strategies. For this analysis, I'll focus on the partial freezing strategy as implemented in the LitClassifier class:

```
def _apply_finetune_strategy(self, strategy, k_layers):
    all_layers = list(self.model.children())

    if strategy == 'freeze_partial':
        for i, layer in enumerate(all_layers):
            if i < len(all_layers) - k_layers:
                for param in layer.parameters():
                    param.requires_grad = False
        self._unfreeze_final_classifier()
```

- This approach freezes earlier layers of the network while allowing the final k_layers and the classifier to be trainable. The strategy leverages the general feature extraction capabilities learned from large datasets like ImageNet while permitting adaptation of higher-level features to the specific naturalist dataset.
    - I have run 450+ sweep using Partial Freeze using k ayers(`k_layers` = 1,2,3) unfreeze with all 5 different type of pretrained model `ResNet50`, `VGG16`, `InceptionV3`, `GoogLeNet`, and `EfficientNetV` Below Parallal Co-ordinate Axis Plot and Validation Accuracy v/s Epochs Plot are provided for all different pretrained models.





➤ **Chosen Strategy and Model (with Justification):** Below detailed discussion and Observations with Justification is provided about Chosen Strategy and Model out of all 450+ Sweep Runs.

- Chosen Pre-trained Model: **EfficientNetV2-S**
  - EfficientNetV2-S is a lightweight and powerful CNN architecture designed for fast and accurate image classification.
  - It is optimized for both training speed and accuracy, making it highly suitable for transfer learnin tasks where computational efficiency matters.
  - It uses compound scaling, meaning it systematically scales width, depth, and resolution together, which often results in better generalization compared to traditional architectures.

    - Pre-training on ImageNet provides EfficientNetV2-S with rich feature representations, particularl for general-purpose object recognition, which closely aligns with the types of images found in iNaturalist (e.g., plants, animals).
    - Best Pretrained Model's Configuration:
        - *Model Name:* `EfficientNetV2-S`
        - *Input Size:* `( 3, 224, 224)`
        - *K Layers:* `2`
        - *Learning Rate:* `0.0001`
        - *Number of Output Classes:* `10`
        - Validation Accuracy: 82.79%
- Chosen Fine-Tuning Strategy: **Partial Freezing** (`freeze_partial`)
    - In partial freezing, early convolutional layers (which capture general low-level features like edges and textures) are frozen. Only deeper layers (which capture high-level, task-specific features like object parts) are unfrozen and fine-tuned.
    - This approach strikes the perfect balance between:
        - Preserving learned general features from ImageNet
        - Adapting to new domain-specific features from iNaturalist
    - Why partial freezing?
        - Freezing too much (freeze all) may not adapt enough to the new task (iNaturalist differs a bit f ImageNet).
        - Unfreezing everything (fine-tuning full model) risks overfitting and longer training time.
    - How many layers to unfreeze?
        - Experimentally tuned. Initially, unfroze around the last 20% of layers.
        - Used hyperparameter sweep (via WandB) to find the optimal number of trainable layers.

➤ **Performance Comparison:**

- **Validation Accuracy :**
    - The most striking difference is in the achievable validation accuracy.
        - Custom CNN (from scratch): The validation accuracy ranges primarily between 0.05-0.4427 (5%-44.27%), with most configurations falling between 0.15-0.35 (15%-35%).
        - Fine-tuned models (partial freezing): The validation accuracy ranges between 0.45-0.82 (45%-82.79%), with many configurations achieving above 0.50 (50%).
    - This substantial difference in performance (with fine-tuned models achieving up to 40% higher accuracy) demonstrates the power of transfer learning when working with limited domain-specific data.
- **Model Architecture Impact :**
    - The parallel coordinate plot for fine-tuned models reveals interesting patterns about how differe architectures respond to partial freezing.
        - EfficientNet-v2-S: Tends to achieve the highest validation accuracies (up to 0.8279) when fine-tuned, especially with careful learning rate selection.
        - VGG16 and Inception V3: Show good performance in the mid-range (0.65-0.75 accuracy).

        - ResNet50: Demonstrates consistent performance across various hyperparameter configuratio with close to. 0.80 val accuracy.
    - This contrasts with the custom CNN model, where architectural choices produce more variable and generally lower performance.
- **Learning Dynamics and Training Efficiency :**
    - The fine-tuning approach demonstrates several efficiency advantages:
        - **Faster Convergence:** The fine-tuned models achieve good validation accuracy with fewer epoc (mostly between 10-15), indicating faster learning.
        - **Learning Rate Sensitivity:** The partial freezing approach shows particular sensitivity to learni rate, with optimal values typically in the 0.0003-0.0008 range. This is different from training fr scratch, where more hyperparameter tuning across multiple dimensions is necessary to achie even moderate performance.
        - **Parameter Efficiency:** By freezing early layers and only training a subset of the network, the fi tuning approach requires significantly less computation despite working with deeper models.

➤ **Key Insights :**

- **Knowledge Transfer Advantage:** Fine-tuned models consistently outperform models trained from scratch, demonstrating effective transfer of general visual features from pre-training on larger datasets.
- **Reduced Training Time:** Fine-tuned models require fewer training epochs to achieve superior performance compared to training from scratch.
- **Parameter Efficiency:** By freezing most parameters and only training a small subset (especially in the freeze_all strategy), fine-tuning uses significantly fewer trainable parameters while achieving better results.
- **Architecture Impact:** Different pre-trained architectures show varying performance levels, suggesting that model selection is an important consideration alongside the fine-tuning strategy.
- **Hyperparameter Sensitivity:** The performance of fine-tuned models shows significant variance based on learning rate and the number of unfrozen layers, indicating the importance of hyperparameter tuning in transfer learning scenarios.

➤ **Practical Implications and Recommendations:** Based on the implementation and analysis, several practical recommendations emerge:

- **Use Transfer Learning for Limited Datasets:** The substantial performance improvement with fine-tuning makes it the preferred approach for naturalist datasets, which typically have limited labeled samples compared to datasets like ImageNet.
- **Model Selection Matters:** Different pre-trained architectures respond differently to fine-tuning. For naturalist datasets, EfficientNet variants appear particularly effective when using the partial freezing strategy.
- **Hyperparameter Focus:** When fine-tuning, concentrate on optimizing:
    - Learning rate (most critical parameter)
    - Number of unfrozen layers (k_layers)
    - Number of training epochs (typically fewer needed than when training from scratch)
- **Partial Freezing Balances Performance and Efficiency:** The partial freezing strategy strikes an effective balance between the computational efficiency of complete freezing and the performance

potential of full fine-tuning.Practical Implications and Recommendations.

✦✦✦

Question 4 (10 Marks)

Paste a link to your GitHub code for Part B

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

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

Q.5: Github Link

➤ **Github Link:** https://github.com/indramandal85/DA6401_Assignment_2_CNN

✧ PART-B ENDS HERE ✧

(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, Name **INDRA MANDAL** (Roll No: **ED24S014**), 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.