DA6401 Deep Learning Assignment 2

Student Details: Sudhanva Satish Roll No: DA24M023

WANDB Link: https://wandb.ai/da24m023-indian-institute-of-technology-madras/DL_A2/reports/DA6401-Assignment-2-Sudhanva-S-DA24M023--VmlldzoxMjE2NjY2MA?accessToken=xbo4d4vwou95vfx3plz4pi5bpsc5e1htmez0lc8zfpyoc1nahhe9fc2m1n1dz3iu **(public access)**

Github Link: suddu21/Deep-Learning-Assignment-2 **(public access)**

Submission Instructions : kindly make a PDF of your wandb.ai report, attach this page (with your details and links) as the front page to that, and then submit it.

Important Instructions:

- Students **must follow the updated submission format strictly**. Non-compliance will result in penalties.
- Any plagiarism will be reported and heavily penalized.
- You will lose marks if we can't access your Github repo and W&B links.
- Submitting non-modular, unstructured, or unreadable code will cause penalties.

DA6401 - Assignment 2 - Sudhanva S DA24M023

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

Sudhanva Satish da24m023

Created on April 7 | Last edited on April 20

Instructions

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

- The required CNN model has been implemented in the BasicCNN class

Total number of parameters

- First convolution layer has (k∗k∗3∗m)
- All 4 subsequent convolution layers have (k∗k∗m∗m)
- Dense layer has (8∗8∗m∗n+n)
    - Note: My model gives 8∗8 dimensional output after the whole convolutional blocks
- Output layer has (10∗n+10)
- The total number of parameters in the model is

```
k*k*3*m + (k*k*m*m)*4 + (8*8*m*n+n) + (10*n+10)
```

Total number of computations

To calculate the total number of multiply-accumulate operations for the BasicCNN model, we assume:

- 5 convolutional layers, each with m filters of size $\mathrm{k} * \mathrm{k}$.
- Dense layer with n neurons.
- Input image: $\mathrm{H} * \mathrm{W} * 3$ (RGB channels).
- Output: 10 classes.
- Padding preserves spatial dimensions before 2*2 max-pooling (halves dimensions).
- Initial $\mathrm{H} = 256, \mathrm{W} = 256$).

Calculation

- Convolutional Layers
- Each layer's multiply-accumulate operations are given by $\mathrm{m} * \mathrm{k} * \mathrm{k} * \mathrm{C_i} * \mathrm{H_o} * \mathrm{W_o}$, where $\mathrm{C_i}$ is the number of input channels, and $\mathrm{H_o} * \mathrm{W_o}$ is the output spatial size before pooling.

Layer 1:

- Input: $\mathrm{H} * \mathrm{W} * 3$, $\mathrm{C_i} = 3$.
- Output: $\mathrm{H} * \mathrm{W} * \mathrm{m}$.
- Operations: $\mathrm{m} * \mathrm{k} * \mathrm{k} * 3 * \mathrm{H} * \mathrm{W} = 3\mathrm{mk}^2\mathrm{HW}$.
- After pooling: $\mathrm{H}/2 * \mathrm{W}/2 * \mathrm{m}$.

Layer 2:

- Input: $H/2 * W/2 * m$, $C_i = m$.
- Output: $H/2 * W/2 * m$.
- Operations: $m * k * k * m * (H/2) * (W/2) = m^2k^2\frac{HW}{4}$.
- After pooling: $H/4 * W/4 * m$.

Layer 3:

- Input: $H/4 * W/4 * m$, $C_i = m$.
- Output: $H/4 * W/4 * m$.
- Operations: $m * k * k * m * (H/4) * (W/4) = m^2k^2\frac{HW}{16}$.
- After pooling: $H/8 * W/8 * m$.

Layer 4:

- Input: $H/8 * W/8 * m$, $C_i = m$.
- Output: $H/8 * W/8 * m$.
- Operations: $m * k * k * m * (H/8) * (W/8) = m^2k^2\frac{HW}{64}$.
- After pooling: $H/16 * W/16 * m$.

Layer 5:

- Input: $H/16 * W/16 * m$, $C_i = m$.
- Output: $H/16 * W/16 * m$.
- Operations: $m * k * k * m * (H/16) * (W/16) = m^2k^2\frac{HW}{256}$.

- After pooling: $\mathrm{H}/32 * \mathrm{W}/32 * \mathrm{m} = 8 * 8 * \mathrm{m}$.
- Total convolutional operations:
  $3\mathrm{mk}^2\mathrm{HW} + \mathrm{m}^2\mathrm{k}^2\mathrm{HW}\left(\frac{1}{4} + \frac{1}{16} + \frac{1}{64} + \frac{1}{256}\right) = \mathrm{k}^2\mathrm{HW}\left(3\mathrm{m} + \mathrm{m}^2 * \frac{85}{256}\right)$
- Dense Layers

First dense layer:

- Input: $64\mathrm{m}$ (from $8 * 8 * \mathrm{m}$).
- Output: n.
- Operations: $64\mathrm{mn}$.

Second dense layer:

- Input: n.
- Output: 10.
- Operations: $10\mathrm{n}$.
- Total dense operations:$64\mathrm{mn} + 10\mathrm{n} = \mathrm{n}(64\mathrm{m} + 10)$
- The total number of computations is:
  $\mathrm{k}^2\mathrm{HW}\left(3\mathrm{m} + \frac{85\mathrm{m}^2}{256}\right) + \mathrm{n}(64\mathrm{m} + 10)$

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, ...
- 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

- The hyperparameter values I tested over were

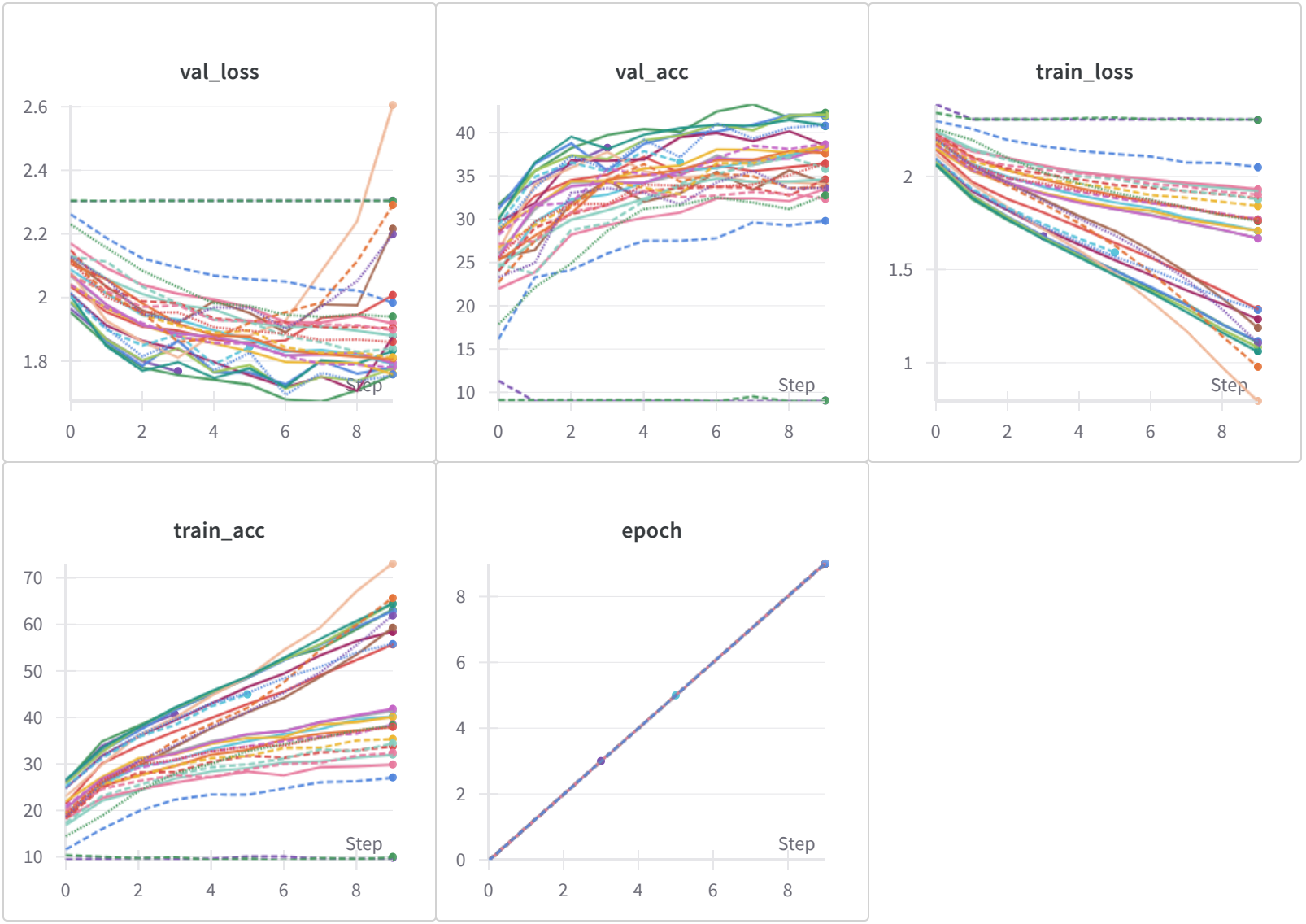
```
batch_size: [16, 32]
learning_rate: [0.0001, 0.001, 0.01]
epochs: 10
conv_filters: [[32, 32, 32, 32, 32], [32, 64, 128, 256, 512], [512, 256, 128, 64, 32]]
activation: ['relu', 'gelu', 'silu', 'mish']
dense_neurons: [32, 64, 128]
dropout_rate: [0.0, 0.2, 0.3]
use_batch_norm: [True, False]
data_augment: [True, False]
```

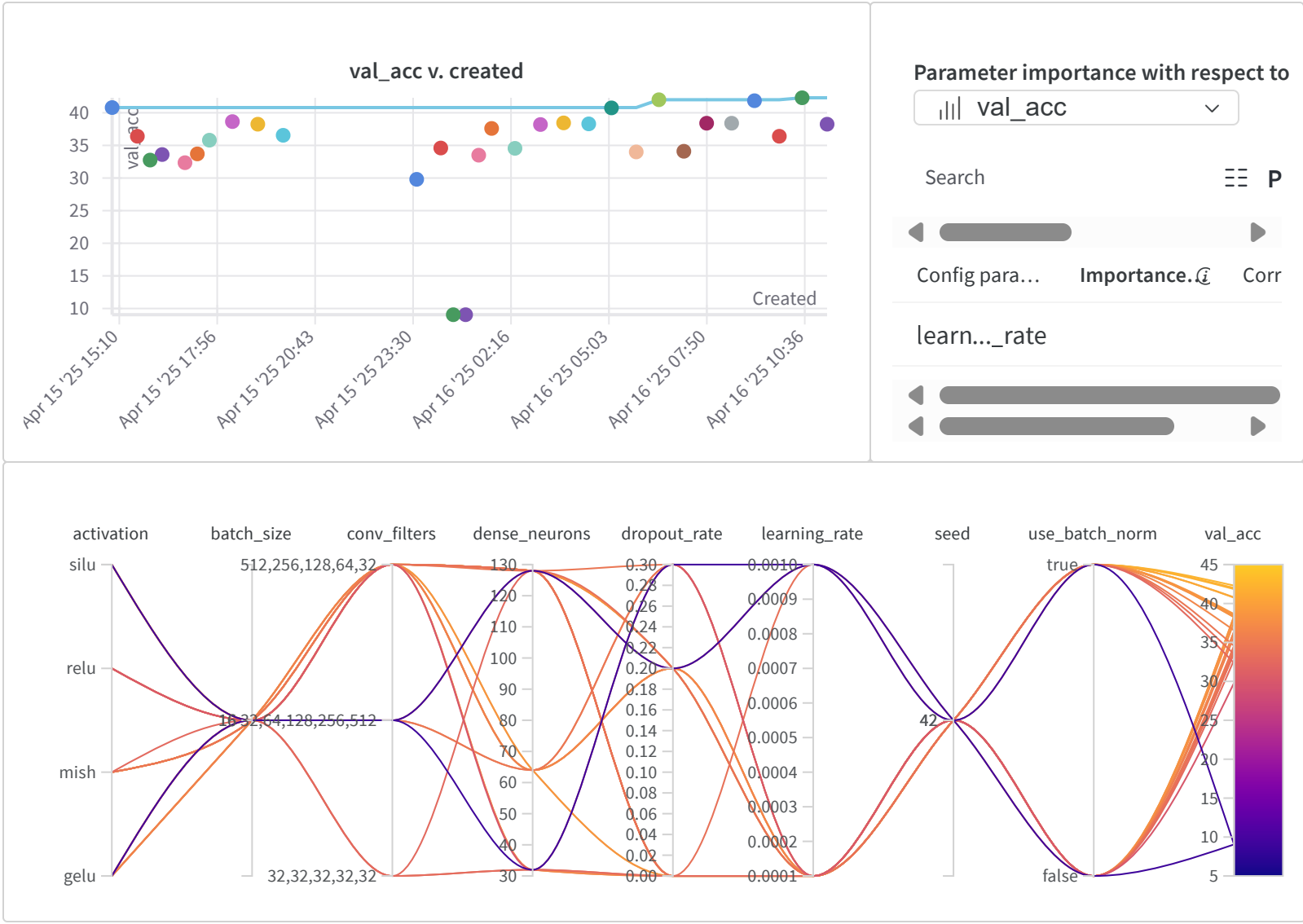
- The best validation accuracy I was able to get on the dataset was 42.3% for a model with config as follows

```
batch_size=16,
learning_rate=0.001
conv_filters=[512,256,128,64,32],
activation='silu',
dense_neurons=64,
dropout_rate=0,
use_batch_norm=True
data_augment=False
```

- Epochs beyond 10 was not really improving validation performance but taking a lot of time so I ran only for 10 epochs
- Dropout can be done after each of the convolution and maxpooling blocks for good effect
- One smart strategy to reduce the number of runs is to use the Bayesian sweeping method which chooses the next run's hyperparameter config based on which configs had given good results in past runs. This gives us better results with a smaller number of runs

- This strategy can be used to greatly reduce overall sweep time





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

- Using Batch Normalization helped the models perform better as it normalizes the outputs after each convolution operation
- Halving the number of filters as we go down the layers gave the best performance
- Using large number of filters to capture more features from the initial input image and reducing the number of filters further is a good approach
- Silu activation outperformed all the other activations. Gelu performed well too
- Dropout seemed to have a random effect of regularization on the model performance, with small amounts of dropout improving performance, however the best model used no dropout
- Learning rate affected the model performance hugely as expected with a lower learning rate of 0.0001 giving the best performance
- Increasing the number of dense neurons seemed to improve performance, but only upto a limit. Increasing beyond this simply increased training time without any accuracy improvement
- Lower the number of filters in convolution layers lesser is the runtime of every epoch

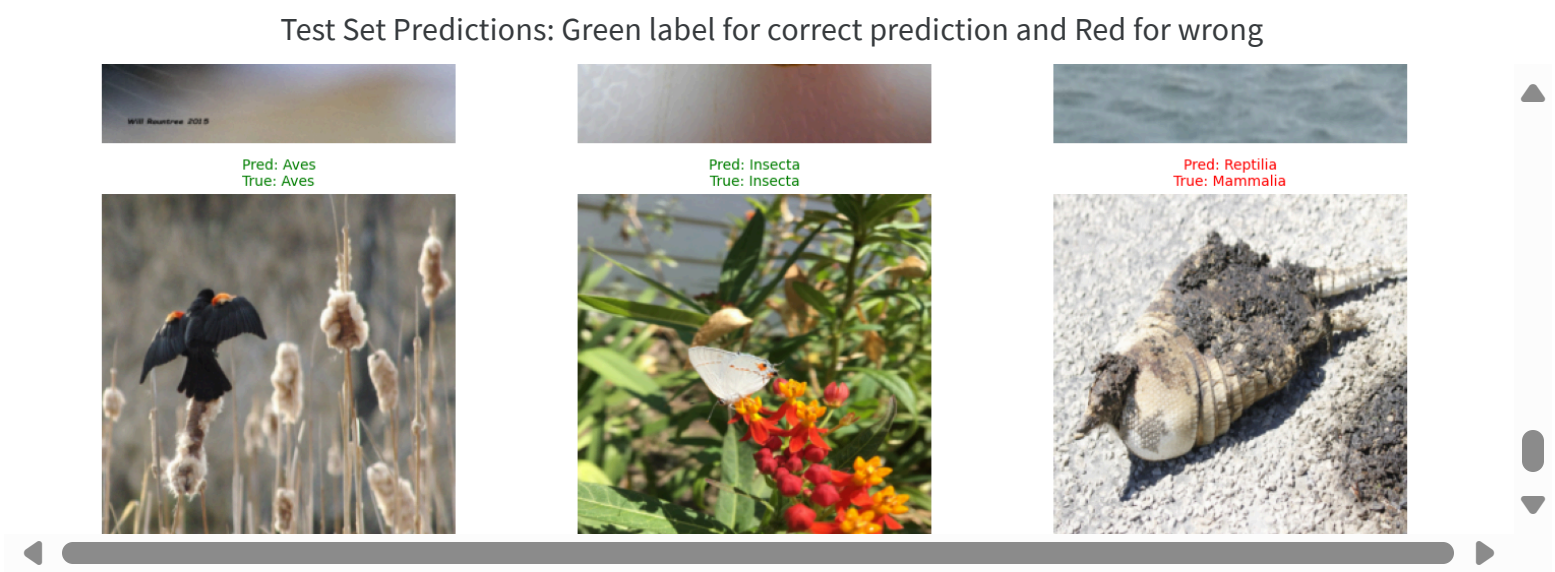
Question 4 (5 Marks)

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

- The best config model mentioned above gave about 38.5% accuracy on the test dataset which it had never been exposed to before
- This seems expected given how complex and diverse the images are, even within classes
- The sample predictions over the test images is displayed below
- The ones predicted correctly are highlighted in green whereas the wrong ones are in red



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.

Answer

- Github - https://github.com/suddu21/Deep-Learning-Assignment-2

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

- I have chosen VGG19 for the pre-trained model as I have worked with it before and it is a relatively simpler model that performs well. Also is less computationally demanding
- The first question regarding image dimensionality, I have solved by resizing the input images to the size accepted by the pretrained model. For VGG19, the accepted input dimension is 224x224
- For the second question, a simple fix is to replace the last layer of the pre-trained model which has 1000 neurons for 1000 classes with a layer containing 10 neurons for 10 classes and train this newly added last layer

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

There are some fine-tuning strategies that help us save time and train models efficiently such as

- Last Layer Fine-Tuning:
    - Freeze all layers of the VGG19 model except the last fully connected layer. Only the parameters of the final linear layer (modified to output 10 classes) are updated during training.
- Partial Fine-Tuning:
    - Freeze the early layers of the VGG19 model. Unfreeze the last n (upto your choice) convolutional layers and the entire classifier module, allowing their parameters to be updated during training.
- Full Fine-Tuning:
    - Unfreeze all layers of the VGG19 model, allowing all parameters (both convolutional and classifier layers) to be updated during training.

Question 3 (10 Marks)

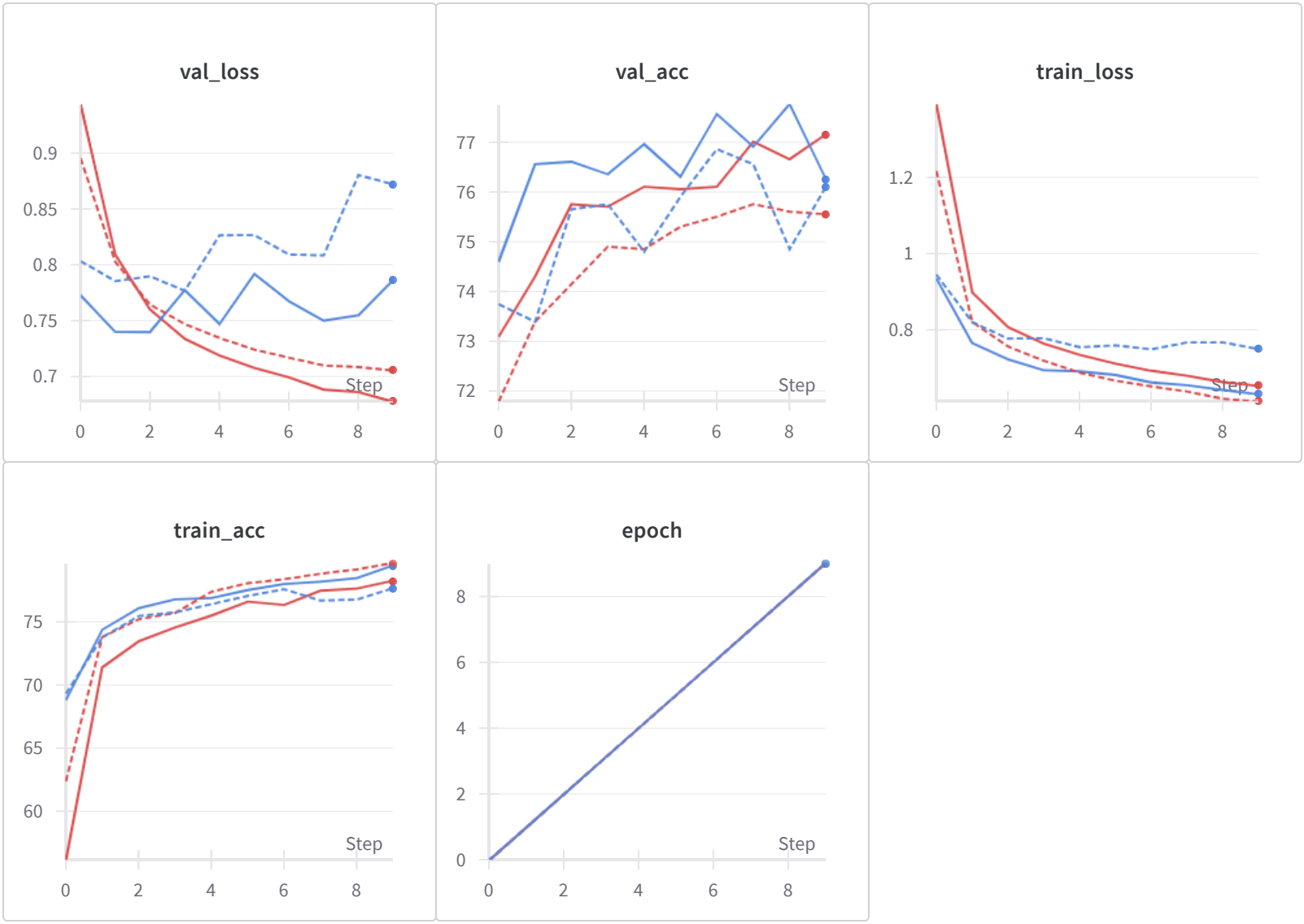
Now fine-tune the model using **ANY ONE** of the listed strategies that you discussed above. Based on these experiments write down some insightful inferences comparing training from

scratch and fine-tuning a large pre-trained model.

Answer

- I have opted to go with Last Layer fine-tuning which freezes all the layers of the pretrained model except for the newly added 10 neuron last layer.
- The intuition is that the pre-trained model was trained on a dataset that contained a lot of animal images, so it has already learned weights that work well for this purpose, hence we stick with them.
- The newly added layer is untrained, so we continue to train it
- A good validation accuracy of around 78% was obtained by following this procedure over some parameter value sweeps
- The pre-trained VGG19 model was originally trained on the large ImageNet dataset and contains approximately 143 million parameters, while the simple model we built in the first part consisted of only five convolutional layers and two fully-connected layers, thus it had much lesser number of parameters.
- Training from scratch on the iNaturalist12K subset needed a lot of epochs to converge and achieved a test accuracy of around 42% which is not very good.
- But finetuning the pretrained VGG19 model by freezing all layers except the newly added 10-neuron last layer gave a much better test accuracy of approximately 78% in significantly lesser time.
- This last-layer fine-tuning approach efficiently leveraged the pre-trained features, minimizing computational cost and avoiding the instability of updating all layers simultaneously
- For those who do not have a lot of compute resources to train large/small models from scratch (like me), finetuning an exisiting state of the art model, popularly known as transfer learning is a good way to go

- Some runs from one the sample sweeps I tried for finetuning are displayed below



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.

Answer

Github - https://github.com/suddu21/Deep-Learning-Assignment-2

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

Self Declaration

I, Sudhanva Satish (DA24M023), 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.