CS6910 - Assignment 2

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

Gowthamaan

Submitted by: Gowthamaan, ED23S037

Instructions

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.

Solution: The code is implemented such a way that the number of filters, size of filters, number of neurons in the dense layer and activation function of the convolution layers and dense layers can be customized while initiating the program.Refer the README.md file for more details

- 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)

Solution

To calculate the total number of computation in the network, we need to consider the computation in each layer. For simplicity, let's leave out the bias and the computations in Batchnorm and Dropout layer.

1. Convolutional Layer: In each convolutional layer, the number of computations is determined by the number of filters (m), the size of each filter ($k \times k$), and the size of the output feature map of the layer. If the output feature map has dimensions $h \times w \times c$ (height, width, channels), then the number of computations for one convolutional layer is $t_{ci} = (k \times k \times m) \times (h_i \times w_i \times c_i)$. In this case, $c_i = m$, since the number of filters in each layer is m. Here, the output size can be calculated from the input by, $h_i = \frac{h_{in} - k + 2p}{s} + 1$, where h_{in} is the input height, p is the padding and s is the stride of the convolution. The same applies to w_i.
2. Activation Layer: The activation layer usually involves element-wise operations. Here, the input to a activation layer will be the the output feature maps of the previous

convolution layer with dimensions $h \times w \times c$ (height, width, channels). So, $t_{ai} = (h_i \times w_i \times c_i)$.

3. Max Pooling Layer: The max pooling operation typically involves comparing a set of values and keeping only the maximum value. The number of computations is the size of the output feature map of the layer, $h \times w \times c$ (height, width, channels). The total computations in a layer is $t_{pi} = (h_i \times w_i \times c_i)$. Here, the output size can be calculated from the input by, $h_i = \frac{h_{in} - f}{s} + 1$, where h_{in} is the input height, f is the size of the filter in max pooling layer and s is the stride. The same applies to w_i.
4. Dense Layer: In the dense layer, the number of computations is determined by the number of neurons (n) and the size of the previous layer's output. If the previous layer has m feature maps of size $h \times w$, then the total number of computations for the dense layer is approximately $t_d = m \times h \times w \times n$.
5. Output Layer: The output layer computations depend on the specific task. For example, in classification tasks, if there are p classes, then there will be p neurons, each connected to every neuron in the previous layer. So, the number of parameters is $t_o = n \times p$, where n is the number of neurons in the previous layer.

Given these,

The total number of parameters in the network t_p is the sum of the following,

1. Convolution layer: $t_{ci} = (k \times k \times m) \times (h_i \times w_i \times c_i)$, where m - number of filters, k - size of feature map and $(h_i \times w_i \times c_i)$ is the output volume of the layer
2. Activation Layer: $t_{ai} = (h_i \times w_i \times c_i)$, where $(h_i \times w_i \times c_i)$ is the input to the layer

3. Max pooling layer: $t_{pi} = (h_i \times w_i \times c_i)$, where $(h_i \times w_i \times c_i)$ is the output of the layer
4. Dense layer: $t_d = m \times h \times w \times n$, where n - number of neurons in the layer and $h \times w \times n$ is the input to the layer
5. Output layer: $t_o = n \times p$, where p is the number of classes and n is the number of neurons in the previous 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)

Solution

To calculate the total number of parameters in the network, we need to consider the parameters in each layer.

1. Convolutional Layer: In each convolutional layer, the parameters consist of the filters and biases. If the size of each filter is $k \times k$ and there are m filters, and the input to the layer has c channels, then the number of parameters is $m \times ((k \times k \times c)+1)$. Assuming m filters in each convolution layer, the total parameters would be, $t_c = m \times ((k \times k \times c) + 1) + 4 * m \times ((k \times k \times m) + 1)$, where c is the number of channels in the input.
2. Activation Layer and Pooling Layer: These layers have got no learnable parameters.
3. BatchNorm Layer(Optional): Every batch norm layer has two learnable parameters. So, if every convolution block has a bacthnorm layer, number of parameters would be $t_b = 5 * 2 = 10$
4. Dropout Layer(Optional) and Flatten layer: These layers have got no learnable parameters.
5. Dense Layer: In the dense layer, each neuron is connected to every neuron in the previous layer. So, the number of parameters is determined by the number of neurons in the dense layer (n) and the size of the previous layer's output. If

the previous layer has m feature maps of size $h \times w$, then the number of parameters is $t_d = ((m \times h \times w \times) + 1) * n$.

6. Output Layer: The output layer parameters depend on the specific task. For example, in classification tasks, if there are p classes, then there will be p neurons, each connected to every neuron in the previous layer. So, the number of parameters is $t_o = n \times p$, where n is the number of neurons in the previous layer.

Given these,

The total number of parameters in the network t_p is,

$t_p = t_c + t_b + t_d + t_o$

$t_p = m \times ((k \times k \times c) + 1) + (b - 1) * m \times ((k \times k \times m) + 1) + (b * 2) + ((m \times h \times w \times) + 1) * n + (n \times p)$, where

m - Number of filters in each layer

k - Size of the filters

c - Number of channels in the input

b - Number of Convolution, Activation and Max pooling blocks (In this case, $b = 5$)

$h \times w$ - Size of the feature map which is the input to the dense layer

n - Number of neurons in the dense layer

p - Number of classes

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.

Solution

I have run 20 runs in total using Bayesian search option in the wandb sweep. In this method, the next set of parameters for the network is selected based on the previously evaluated configurations. I have also implemented early stopping based

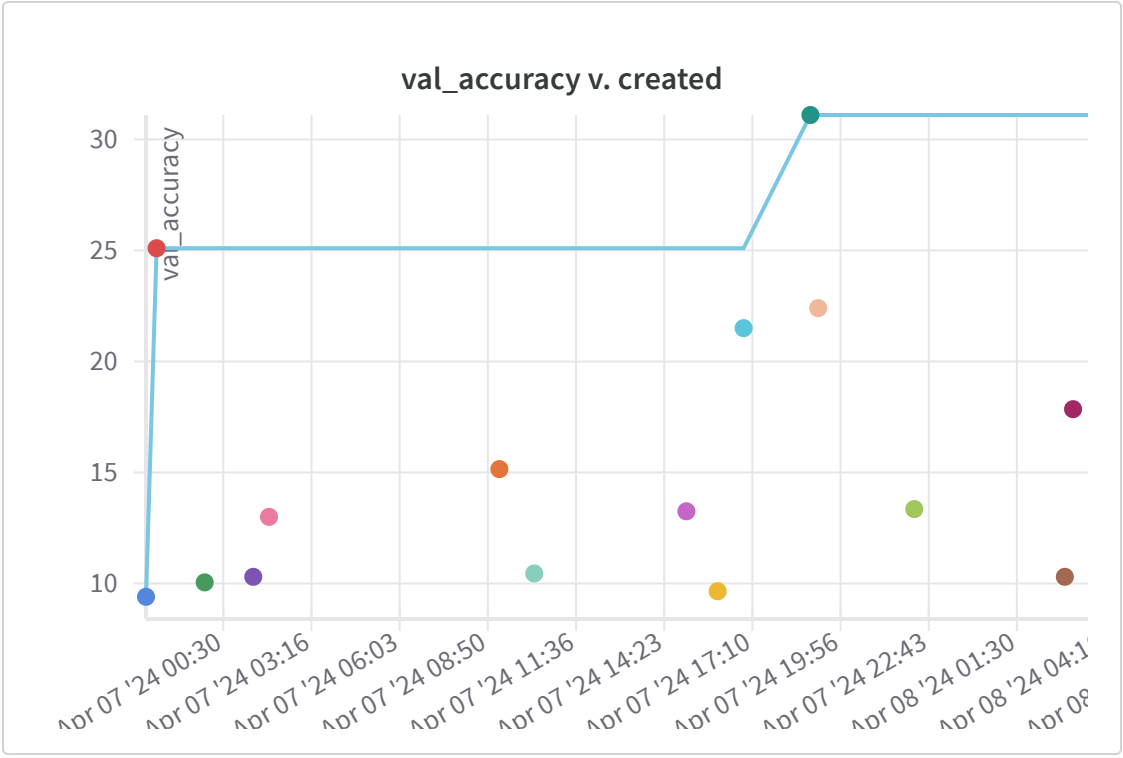
on validation accuracy to sweep over multiple configurations. The following is my sweep configuration.

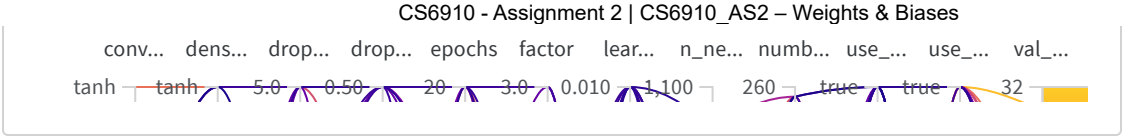
sweep_config = {

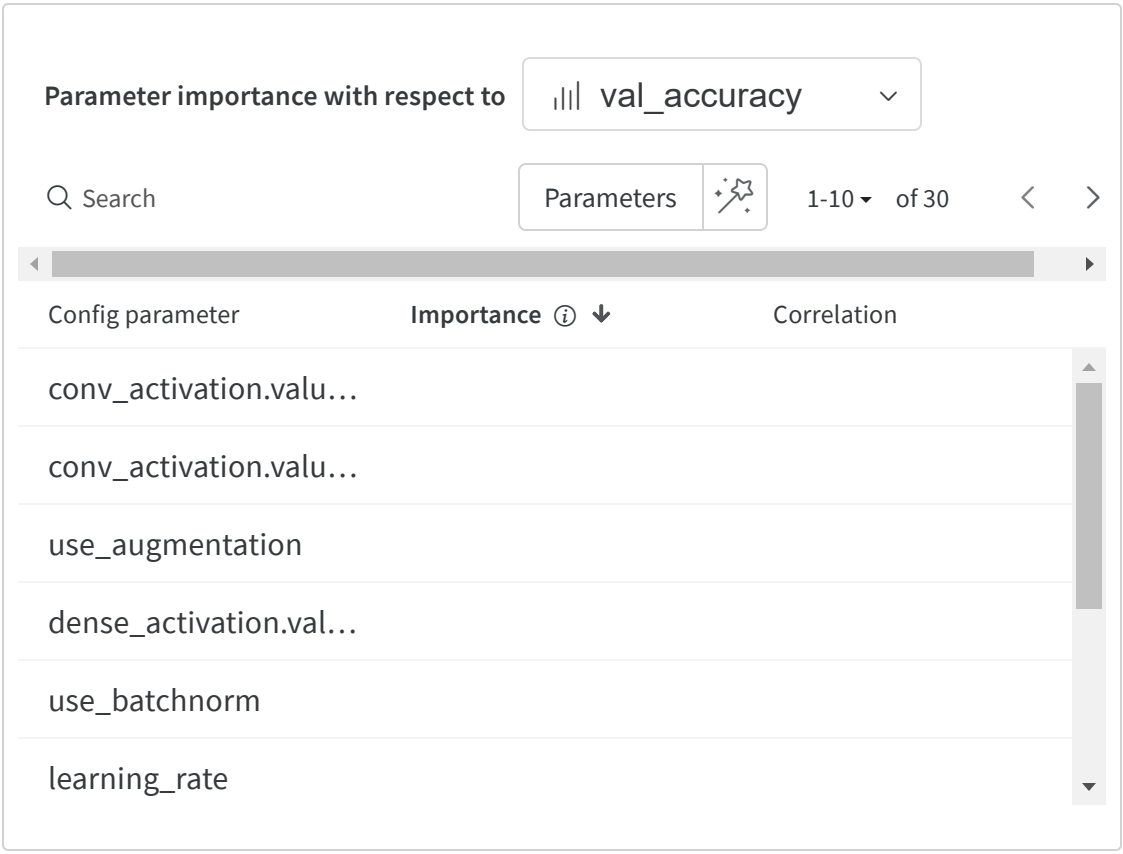
```
'method': 'bayes',

'name': 'PART_A_Q2_SWEEP_1',

'metric': {

    'name': "val_accuracy",

    'goal': 'maximize',

},

'parameters': {

    'number_of_filters': {'values': [16, 32, 64, 128, 256]},

    'n_neurons': {'values': [64, 128, 256, 512, 1024]},

    'conv_activation': {'values': ['relu', 'gelu', 'silu', 'mish

    'dense_activation': {'values': ['relu', 'gelu', 'silu', 'mis

    'dropout_rate': {'values': [0.2, 0.3, 0.4, 0.5]},

    'use_batchnorm': {'values': [True, False]},

    'factor': {'values': [1, 2, 3, 0.5]},

    'learning_rate': {'values': [1e-2, 1e-3, 1e-4, 1e-5]},

    'epochs': {'values': [5, 10, 15, 20]},

    'use_augmentation': {'values': [True, False]},

    'dropout_organisation': {'values': [1, 2, 3, 4, 5]},
```

```
},
```

```
}
```


val_accuracy v. created
val_accuracy
30
25
20
15
10
Apr 07 '24 00:30
Apr 07 '24 03:16
Apr 07 '24 06:03
Apr 07 '24 08:50
Apr 07 '24 11:36
Apr 07 '24 14:23
Apr 07 '24 17:10
Apr 07 '24 19:56
Apr 07 '24 22:43
Apr 08 '24 01:30





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)

Solution

We can observe the following from the above plots:

- Using more filters in the initial layer or more number of filters in all the layer didn't lead to better performance. The model tends to overfit when the number of parameters increases i.e. as the model becomes more complex
- Halving the number of filters in every layer gave poor results compared to doubling or remaining same number of filters in every layer.
- Dropout do help to prevent the model from overfitting the data. But the number of dropout layers and which layers have dropout also determines the performance of the network. No dropout or having dropout in all the layers results in poor performance. Adding dropout before the fully connected layer improves performance. Also, higher dropout rate produces poor results.
- Data augmentation did not improve the performance of the model. In some cases, accuracy reduces due to data augmentation, we can infer this from the correlation plot.
- ReLU activation performs far better than any other activation functions.Tanh activation is the second best from the choice of activations used in the sweep.
- Less number of neurons in the dense layer lead to poor performance. All the sweep runs with more than 512 neurons in the dense layer have better validation accuracy.
- Batch-normalization seems to improve accuracy of the model in general.
- Higher learning rate or very low learning rate leads to poor performance. From the correlation plot, we can observe that the learning rate is negatively correlated with the validation accuracy. A moderate learning rate (1e-4 oe 1e-3) yield better results

I believe that the hyperparameter subspace that I have used to run sweeps on, may not be the best. So, the network couldn't learn a proper function that represents the dataset. This might

be the reason that I am not getting more than 35% validation accuracy.

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.

The following is the best accuracy configuration I got from the sweep.

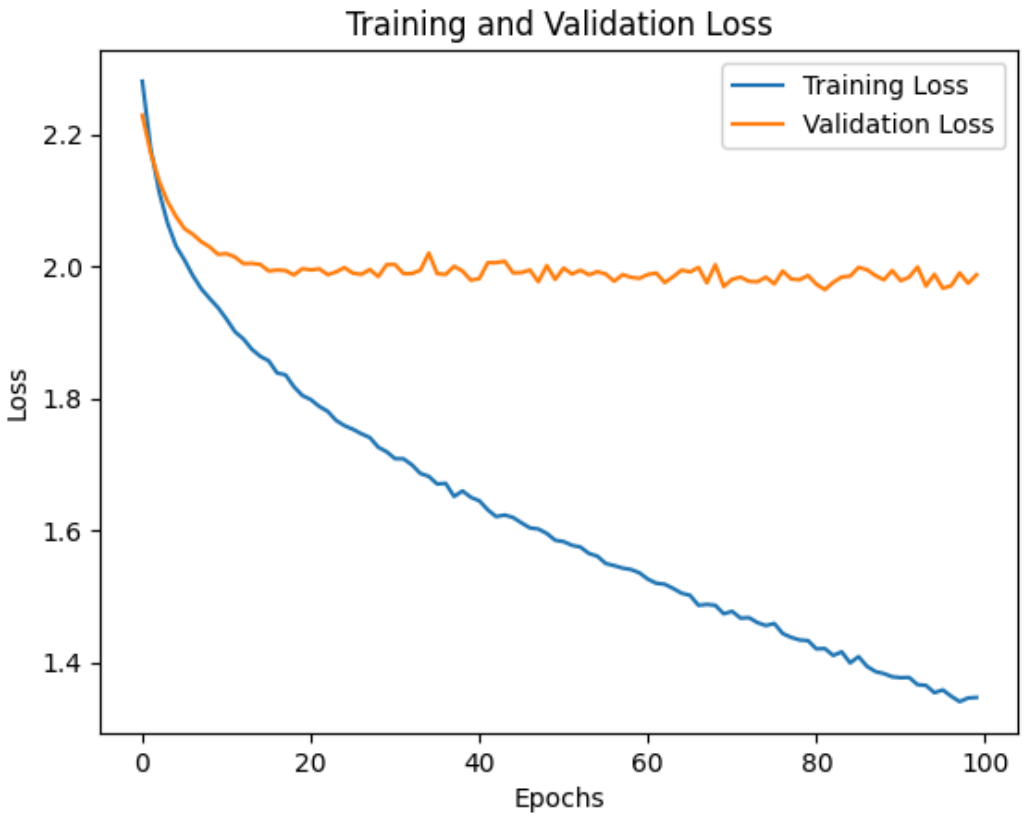
config = {

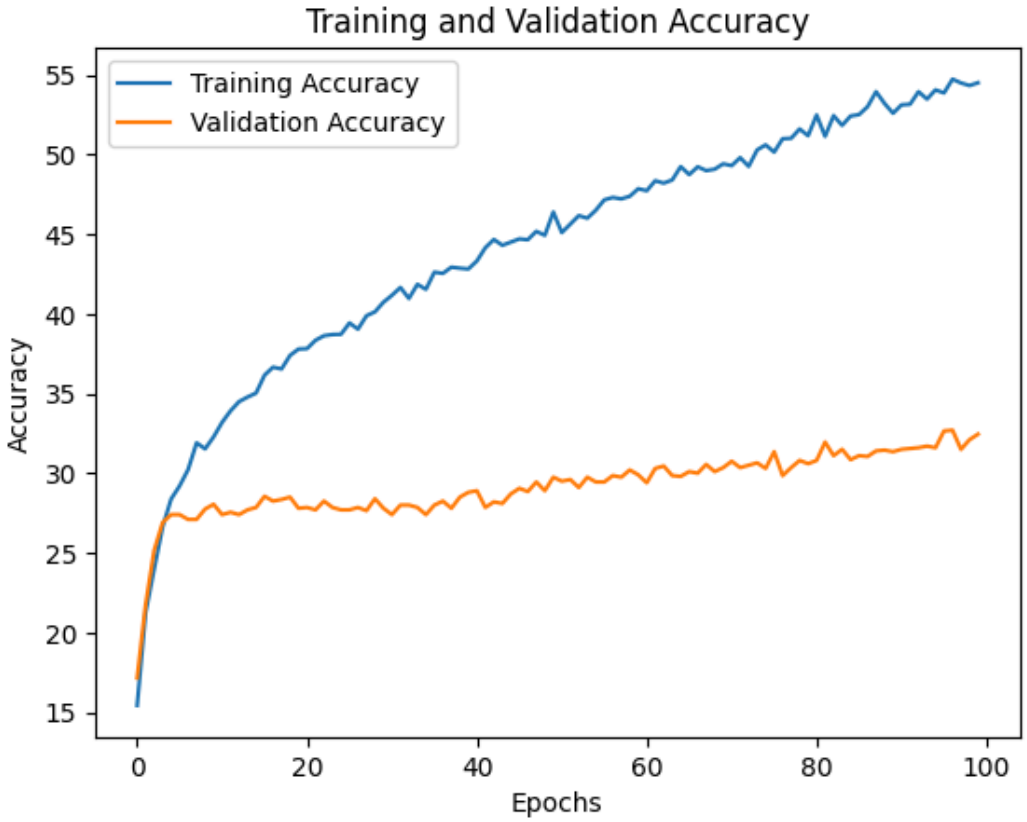
```
'number_of_filters': 32,

'filter_size': 3,

'stride': 1,

'padding': 1,

'max_pooling_size': 2,

'n_neurons': 512,

'n_classes': 10,

'conv_activation': 'relu',

'dense_activation': 'relu6',

'dropout_rate': 0.2,

'use_batchnorm': True,

'factor':1,
```

```
'learning_rate': 1e-4,

'batch_size':64,

'epochs':100,

'use_augmentation': False,

'dropout_organisation': 3,
```

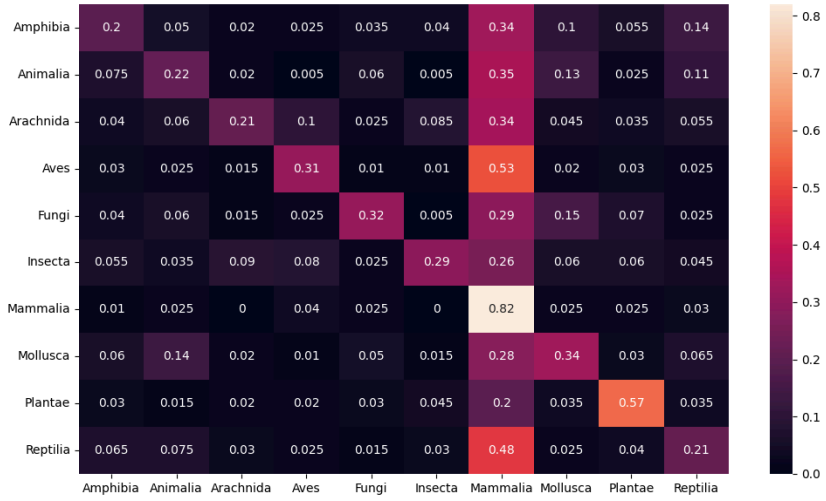
}

I have increased the number of filters, batch size and epochs (which I have reduced for sweep due to limited GPU) and trained the model again to get these following results.

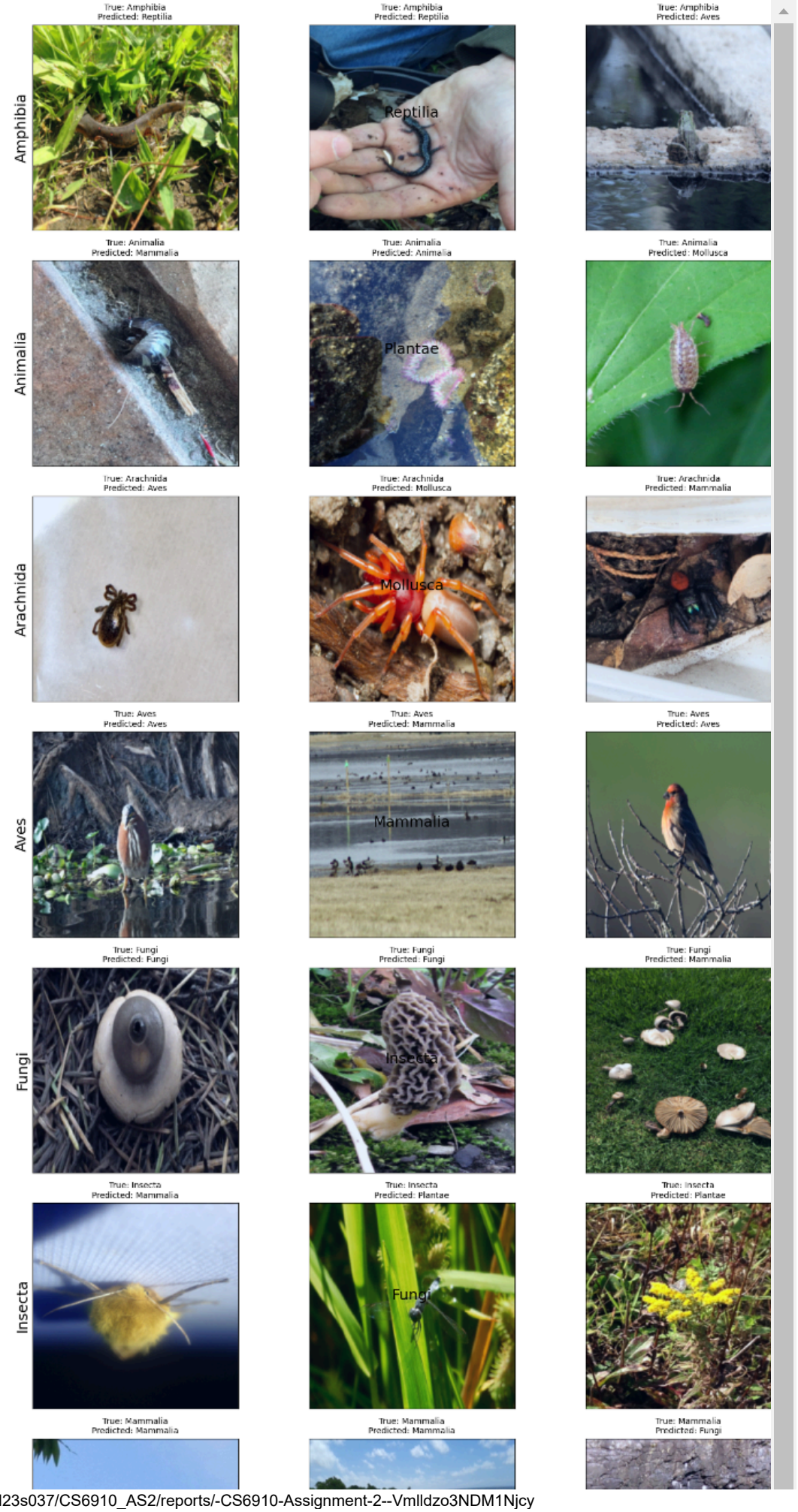


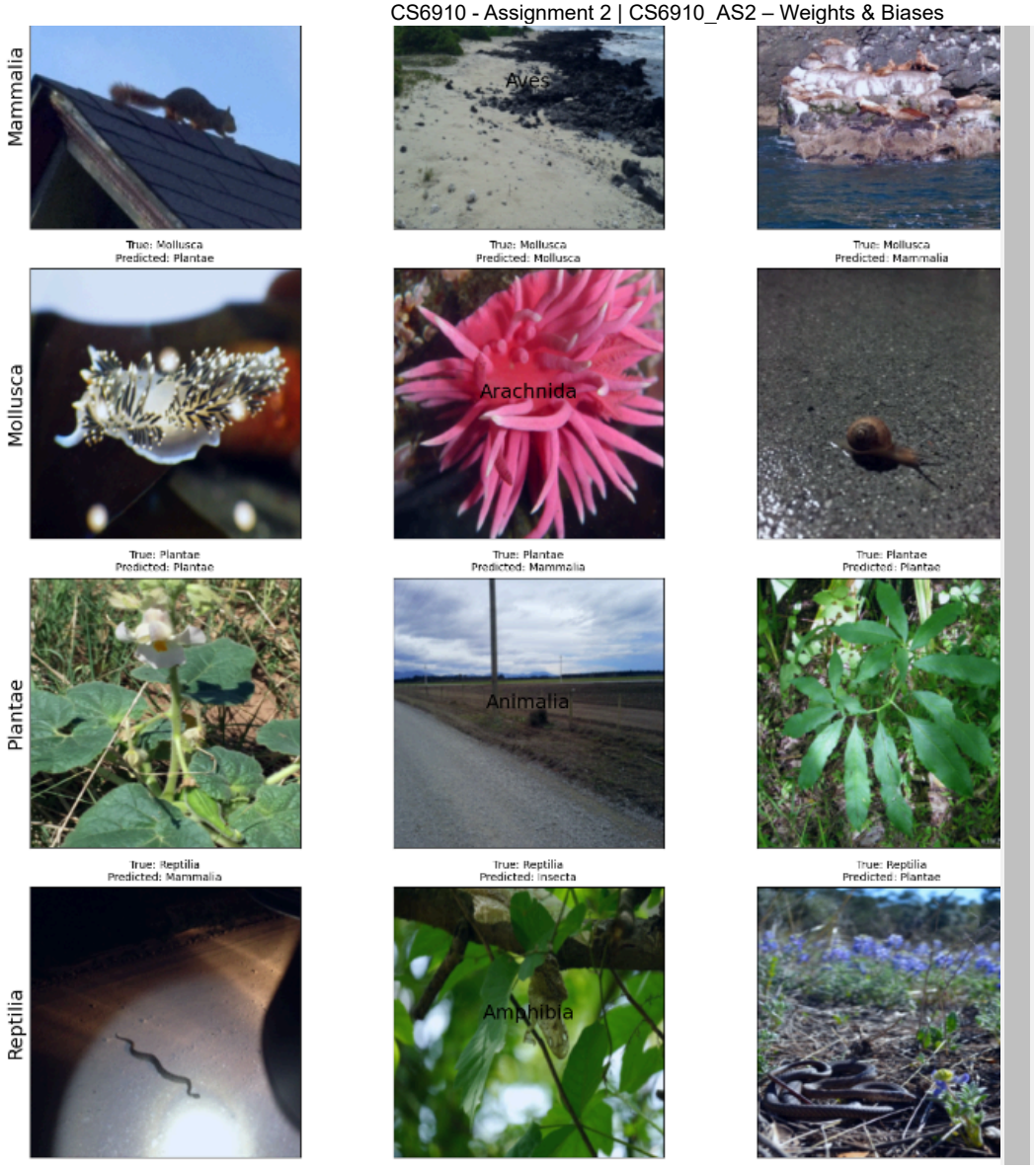


I have saved the model from the above training. And ran it test set. I have got **34.95%** accuracy in the test set



- 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)


True: Amphibia
Predicted: Reptilia
True: Amphibia
Predicted: Reptilia
True: Amphibia
Predicted: Aves
Amphibia
Reptilia
True: Animalia
Predicted: Mammalia
True: Animalia
Predicted: Animalia
True: Animalia
Predicted: Mollusca
Animalia
Plantae
True: Arachnida
Predicted: Aves
True: Arachnida
Predicted: Mollusca
True: Arachnida
Predicted: Mammalia
Arachnida
Mollusca
True: Aves
Predicted: Aves
True: Aves
Predicted: Mammalia
True: Aves
Predicted: Aves
Aves
Mammalia
True: Fungi
Predicted: Fungi
True: Fungi
Predicted: Fungi
True: Fungi
Predicted: Mammalia
Fungi
Insecta
True: Insecta
Predicted: Mammalia
True: Insecta
Predicted: Plantae
True: Insecta
Predicted: Plantae
Insecta
Fungi
True: Mammalia
Predicted: Mammalia
True: Mammalia
Predicted: Mammalia
True: Mammalia
Predicted: Fungi



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

Question 5 (10 Marks)

Paste a link to your github code for Part A

Example: https://github.com/<user-id>/cs6910_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.

Repo Link: https://github.com/Gowthamaan-P/cs6910_assignment2/partA

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?

Solution: I will resize the images using torch.transforms.Resize() to match the input size expected by the pre-trained model.

- 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?

Solution: I will replace the last fully connected layer(the output layer) of the pre-trained model with a new fully connected layer having 10 nodes to match the naturalist dataset. Then, fine tune the model with the inaturalist dataset.

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).

The common trick used to keep the training tractable is Transfer learning. The different strategies, I want to try are,

- Freezing all the layers, except the fully connected layers

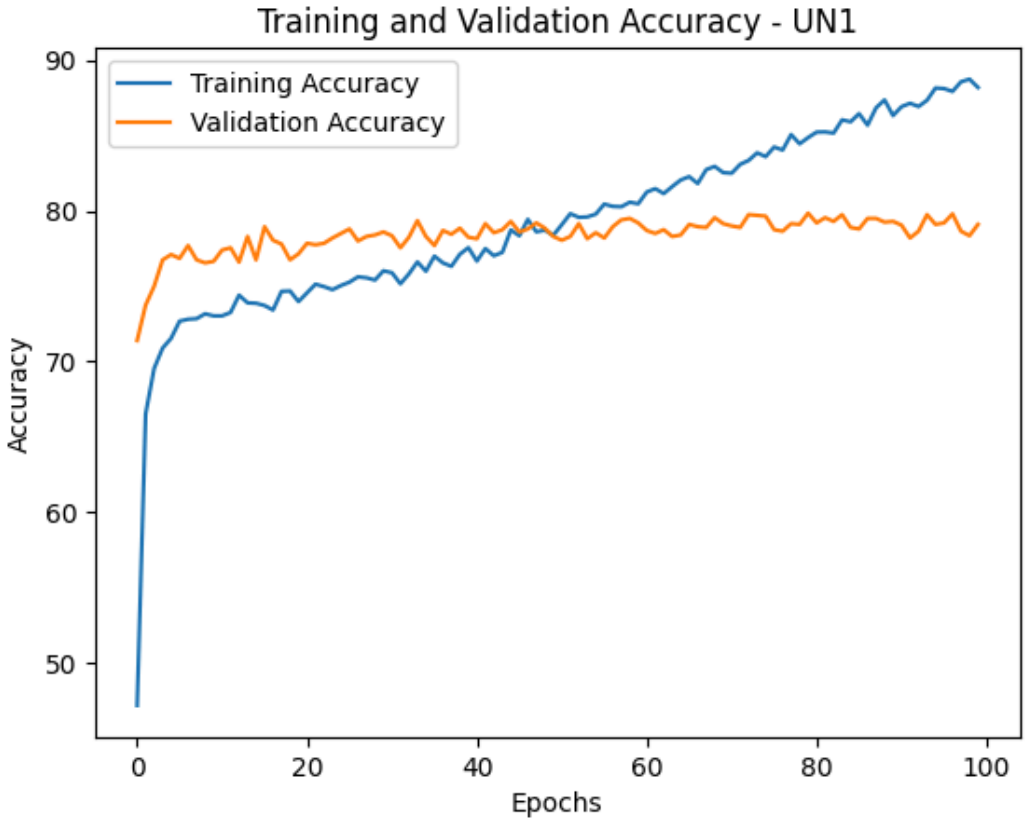
- Freezing only the initial layers (up to k layers) and fine tuning the remaining layers
- Unfreezing all the layers and fine tuning with a low learning rate
- Progressively unfreezing the layers in every step of the training
- Unfreezing layers based on a probability during training
- Unfreezing only the layers at the end of the network (k layers from the last)

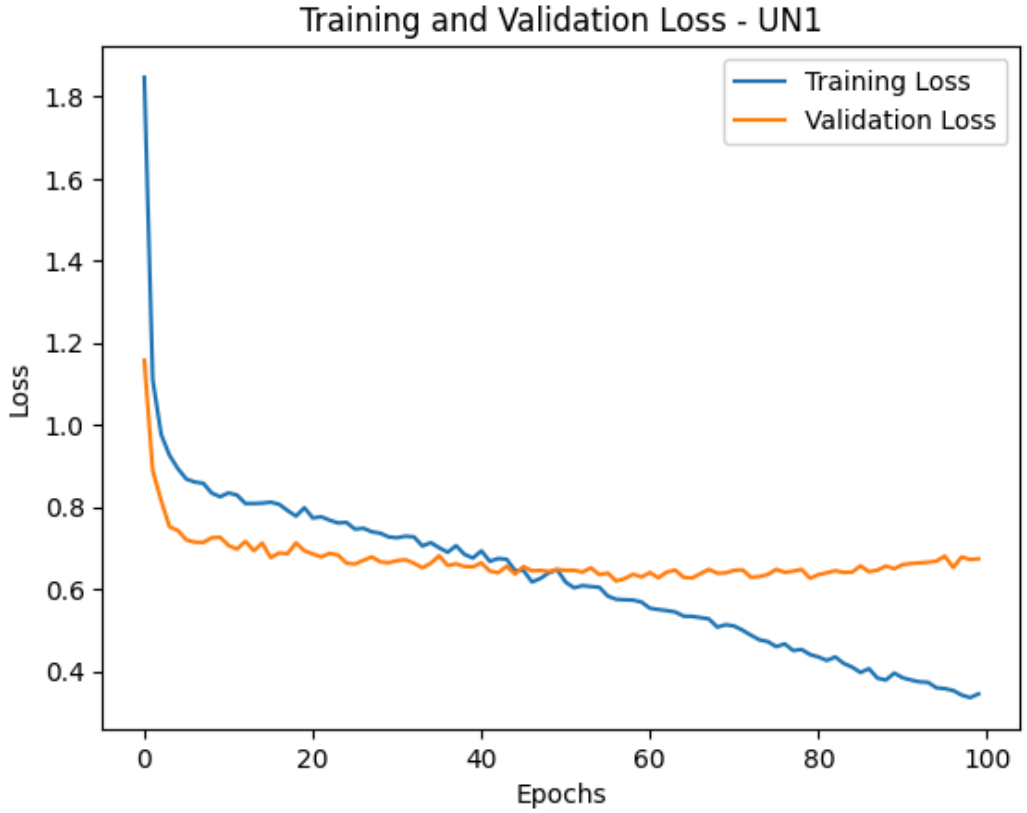
Question 3 (10 Marks)

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.

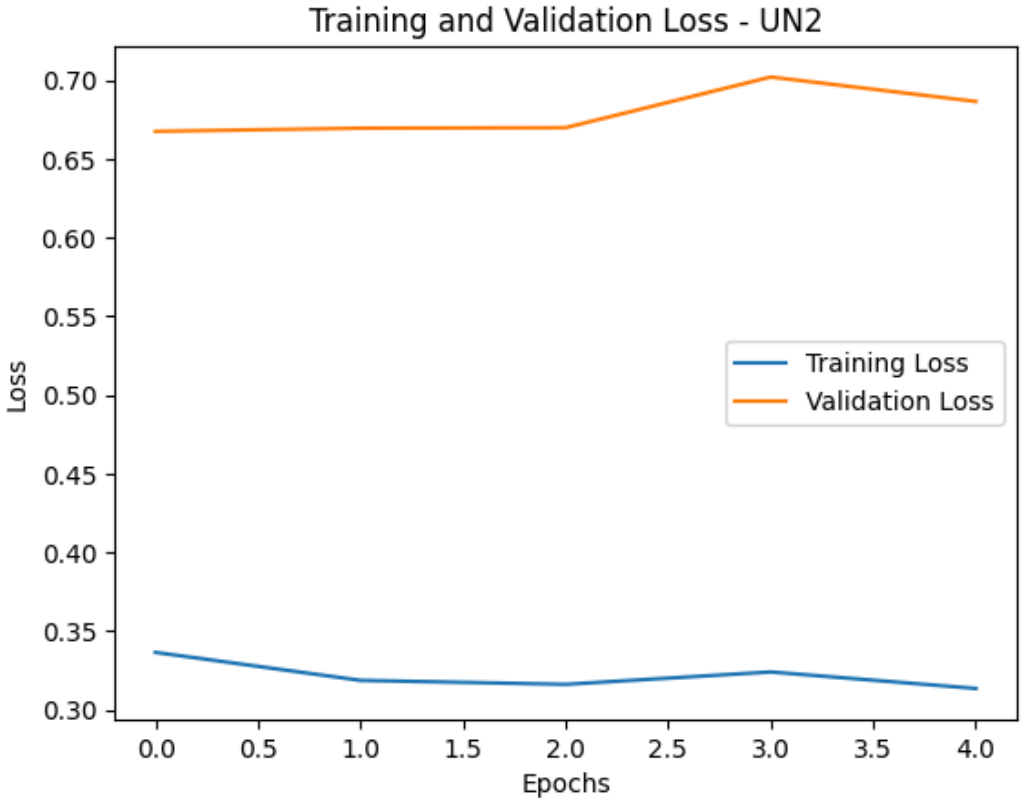
I am going to combine two of the above mentioned strategies and choose to progressively unfreeze k layers at the end of the model. Freezing the initial layer will lead to better feature extraction since the ImageNet challenge dataset is also similar to the inaturalist dataset. Progressive unfreezing will lead to stable training and prevent the model from losing the pre-trained representation.

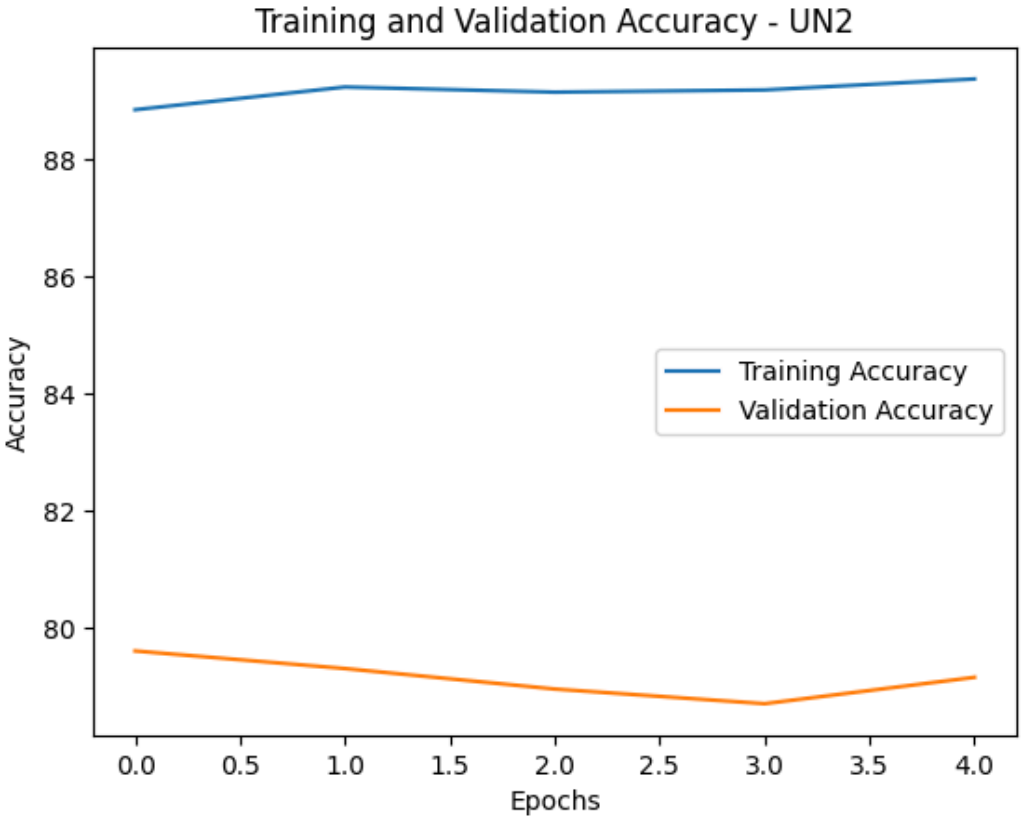
These are my results. Initially I unfreezed the fully connected layer and the final convolution layer alone. This is my loss and accuracy output.



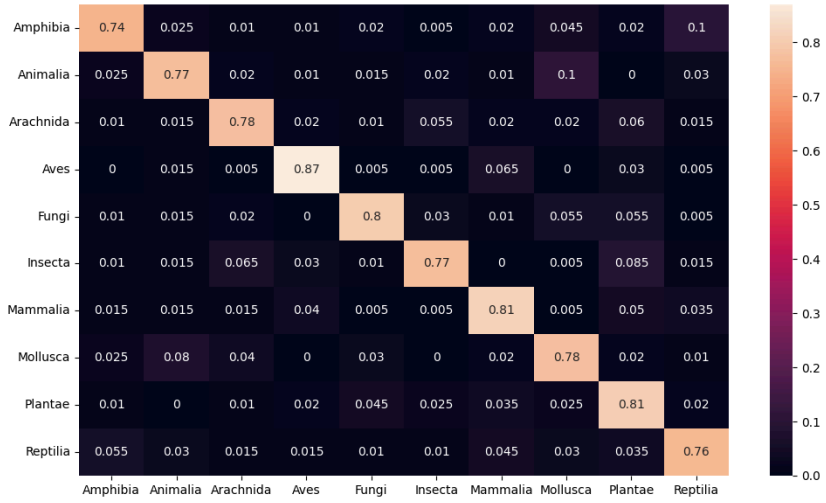


After this, I unfreezed the previous convolution layer also.





In the test set, I got **78.75%**



Training from scratch and fine-tuning a large pre-trained model are two common strategies in deep learning, each with its own advantages and disadvantages. Here are some insightful inferences comparing the two approaches:

- Pre-trained model has already learned meaningful representations from a large dataset, fine-tuning adapts these representations to the specific task requiring less training to achieve good performance whereas, training a model from scratch is very expensive in terms of training.
- Pre-trained model may lead to better generalization than training from scratch
- Number of Hyperparameters in finetuning is very less than training a model from scratch
- The chance of over fitting is very less when fine tuning a pre-trained model instead of training one from scratch

Question 4 (10 Marks)

Paste a link to your GitHub code for Part B

Example: https://github.com/<user-id>/cs6910_assignment2/partB

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

Repo Link: https://github.com/Gowthamaan-P/cs6910_assignment2/partB

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

Self Declaration

Created with ❤️ on Weights & Biases.

https://wandb.ai/ed23s037/CS6910_AS2/reports/-CS6910-Assignment-2--Vmlldzo3NDM1Njcy