Report: Assignment 2

Report on how to use CNNs: Train from scratch, Finetune a pretrained model, Use a pre-trained model as it is.

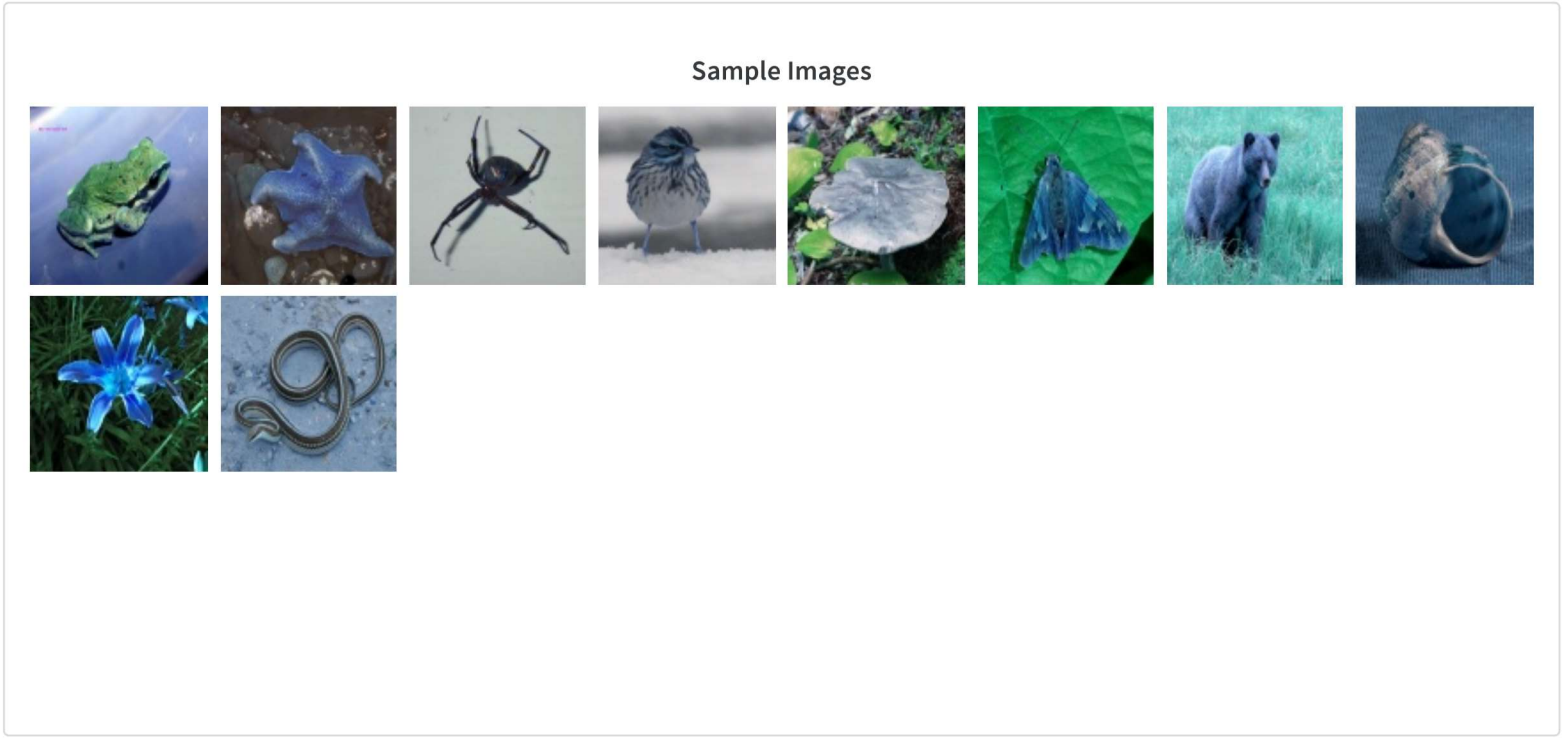
Rigved Sah cs20m053

In this assignment we implemented and trained a CNN using Tensorflow, finetuned a pre-trained model as well as used a pre-trained model as it is for a novel application. The dataset used to train and evaluate the model is **i-Naturalist** dataset.

Since every image in i-Naturalist dataset has different dimension, we resized each image to 100 x 100 pixels. This made input size common among all images as well as do not take too much RAM which crashes the system.

Specifically, given an input image (100 x 100 = 10000 pixels) from the i-Naturalist dataset, the network was trained to classify the image into 1 of 10 classes.

Below panel contains sample image for each class along with label present in the i-Naturalist dataset.



Part A: Training from Scratch

We implemented a CNN which takes images from the i-Naturalist dataset as input and outputs one of the 10 classes. The code was written from scratch using libraries such as Keras, Tensorflow and Numpy.

Our model consists of **5 Convolution Layers**. Each convolution layer is followed by a **ReLU** activation and a **Max Pooling** layer. After 5 such conv-relu-maxpool blocks of layers we have one dense layer followed by the output layer containing 10 neurons (1 for each of the 10 classes). We used **Softmax** as output function and **Cross-Entropy** as the loss function.

In addition to above layers we have also used **Batch Normalization** & **Dropout** layers in various different positions in some sweep runs to check the effect of these layers on convergence, accuracy, loss etc.

The code is written in a very flexible manner that allows changing any hyperparameters such as filters, size of filters and activation functions etc. at just one place.

The total number of computations done by our network is

1.) We calculate the number of computations in Convolutional Layer which are of two types: **Multiplication** and **Addition** and can be calculated as,

> No of Multiplications = Output Shape * k * k * Depth of Input
>
> No of Additions (within above multiplications) = Output Shape * (k * k * Depth of Input - 1)
>
> No of Bias Additions = Output Shape

- Convolutional layer Number 1 :

1. Shape of Input = 100 * 100 * 3 (We have resized the size of inpu
   images to 100 x 100 pixels)

2. Shape of Output = 98 * 98 * 64
3. Number of Multiplications = 98 * 98 * 64 * 3 * 3 * 3 = 16,595,712
4. Number of Additions = 98 * 98 * 64 * (3 * 3 * 3 -1) = 15981056
5. Bias Computation = 98 * 98 * 64 = 614656

Total number of Computations = 33,191,424

- Convolutional layer Number 2:

1. Shape of Input = 49 * 49 * 64
2. Shape of Output = 47 * 47 * 64
3. Number of Multiplications = 47 * 47 * 64 * 3 * 3 * 64 = 81,432,576
4. Number of Additions = 47 * 47 * 64 * (3 * 3 * 64 - 1) = 81,291,200
5. Bias Computation = 47 * 47 * 64 = 141376

Total number of Computations = 162,865,152

- Convolutional layer Number 3 :

1. Shape of Input = 23 * 23 * 64
2. Shape of Output = 21 * 21 * 64
3. Number of Multiplications = 21 * 21 * 64 * 3 * 3 * 64 = 16,257,024
4. Number of Additions = 21 * 21 * 64 * (3 * 3 * 64 -1) = 16,228,800
5. Bias Computation = 21 * 21 * 64 = 28224

Total number of Computations = 32,514,048

- Convolutional layer Number 4 :

1. Shape of Input = 10 * 10 * 64

2. Shape of Output = 8 * 8 * 64
3. Number of Multiplications = 8 * 8 * 64 * 3 * 3 *64 = 2,359,296
4. Number of Additions = 8 * 8 * 64 * (3 * 3 * 64 - 1) = 2,355,200
5. Bias Computation = 8 * 8 * 64 = 4096

Total number of Computations = 4,718,592

- Convolutional layer Number 5 :

1. Shape of Input = 4 * 4 * 64
2. Shape of Output = 2 * 2 * 64
3. Number of Multiplications = 2 * 2 * 64 * 3 * 3 * 64 = 147,456
4. Number of Additions = 2 * 2 * 64 * (3 * 3 * 64 - 1) = 1,47,200
5. Bias Computation = 2 * 2 * 64 = 256

Total number of Computations = 294,912

2.) We know compute the computations in Dense Layer. Here we have taken, n = 128 and input to Dense layer is of size 64.

- Calculating Computations in 1st Dense Layer :

1. Dense Layer (Multiplications) = n * Depth of Input = 128 * 64
2. Dense layer (additions):
    1. Addition within above Multiplicative terms = 128 * (64 - 1) = 128 * 63
    2. Bias Addition = 128
    3. Total Addition = 128 * 63 + 128 = 128 * 64

Total Computations of Dense layer = 16384

3.) We know compute the computations in Output Layer. Here we have taken, n = 10 and input to Dense layer is of size 128. The number of computations for dense layer can be calculated as,

1. Output Layer (Multiplications) = No of Classes * Depth of Input = 10 * 128
2. Dense layer (additions):
    1. Addition within above Multiplicative terms = 10 * (128 - 1) = 10*127
    2. Bias Addition = 10
    3. Total Addition = 10 * 127 + 10 = 10 * 128

Total Computations of Dense layer = 2560

Total Number of Computations = **233,603,072**

The total number of parameters in our network is

There will be non zero parameters in the convolutional layer, dense layer and the output layer while other layers such as activation and max-pooling layers will have 0 parameters. Now the number of parameters in a layer can be given as,

> Parameters in a Layer = k * k * Depth of Input * Number of Filters + Number of Filters (for Bias)

- Number of parameter in 1st Convolutional Layer $= m * k^2 * 3 + m = 2mk^2 + m$
- Number of parameter in 2nd, 3rd & 4th Convolutional layers are all equal and equal to $m^2 * k^2 + m = (mk)^2 + m$
- Therefore total parameters in 2nd , 3rd & 4th layer = $4((mk)^2 + m)$
- Number of parameter in 1st Dense layer = $m * n + n$
- Number of parameter in Output layer = $10n + 10$

Total Number of Parameter = $3mk^2 + m + 4((mk)^2 + m) + mn + 11n + 10$

Taking filter(m) = 64, filter size = 3 x 3, number of neurons in Dense layer (n) = 128 and number of classes = 10. Now putting all these in above equation we get total number of parameter = **159114**

Train-Val-Test Split

We used the standard train/test split of i-Naturalist and kept 10% of training data aside as validation dataset for hyperparameter tuning.

We shuffled the training data to randomly pick validation dataset and also made sure the distribution of each class in validation dataset is uniform so as to prevent bias towards any particular class.

The summary for different datasets is as:

> Training Data: 8999 images of size 100 x 100 pixels
>
> Validation Data: 1,000 images of size 100 x 100 pixels (100 images of each class)

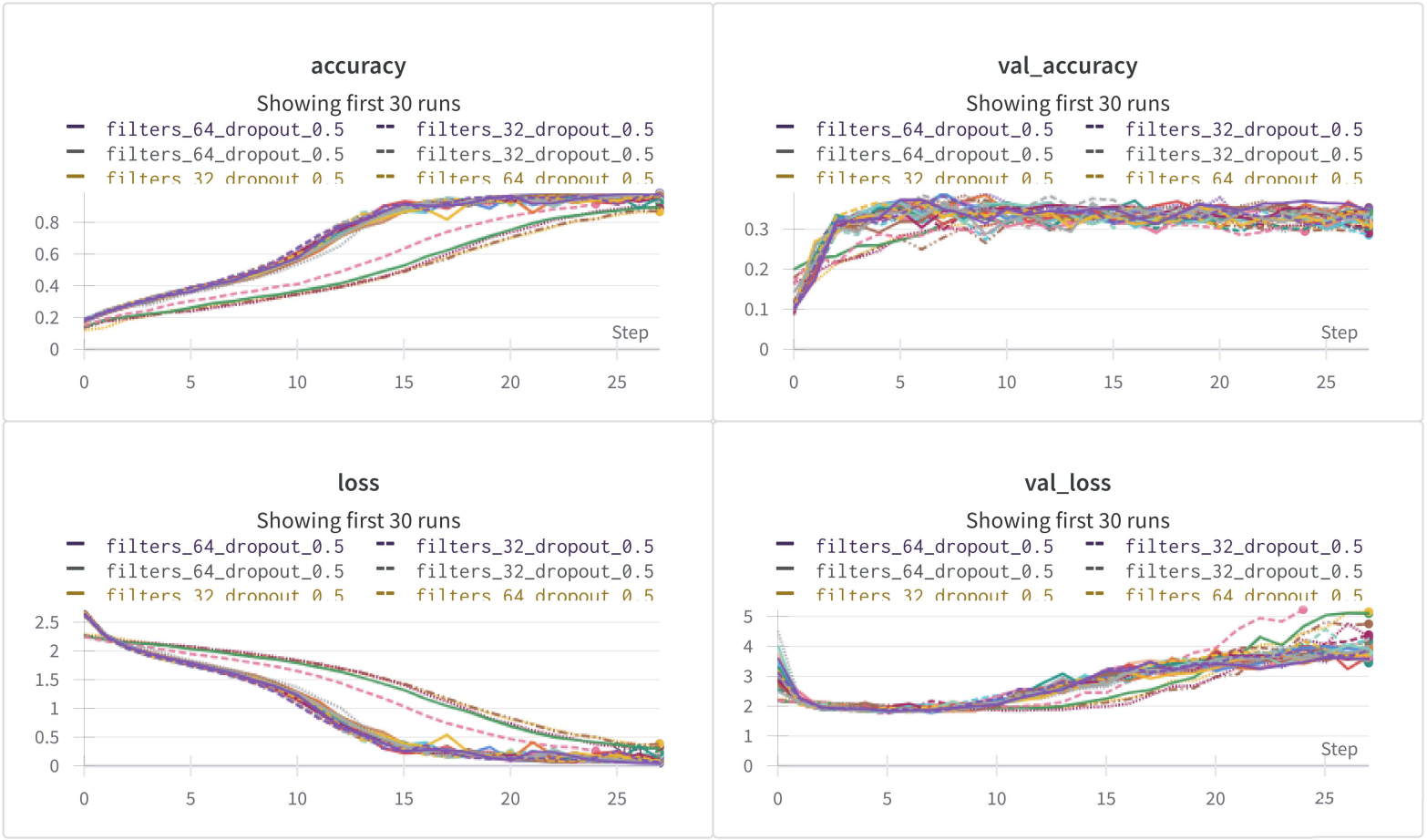
> Test Data: 2,000 images of size 100 x 100 pixels

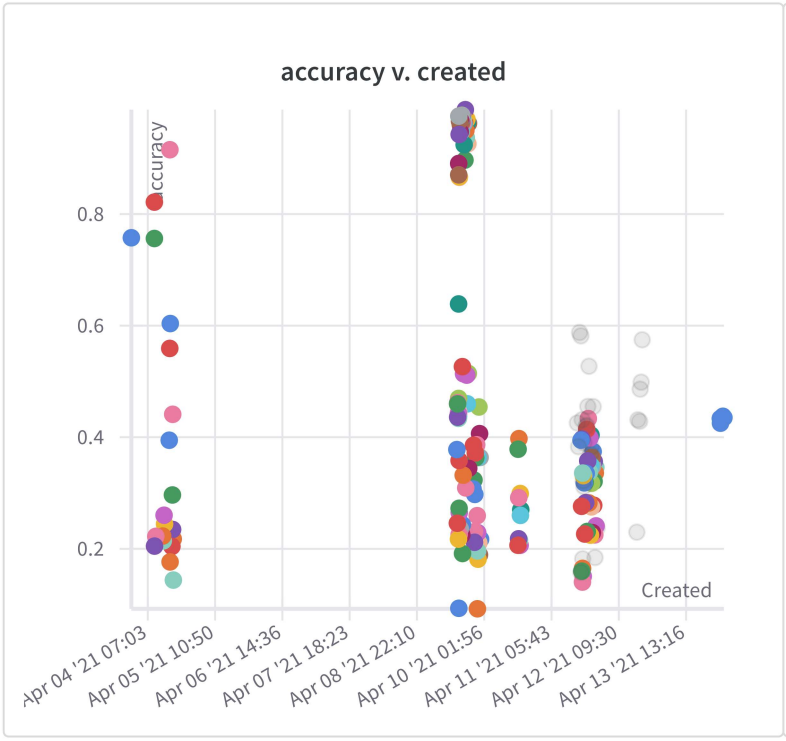
Hyperparameter Tuning

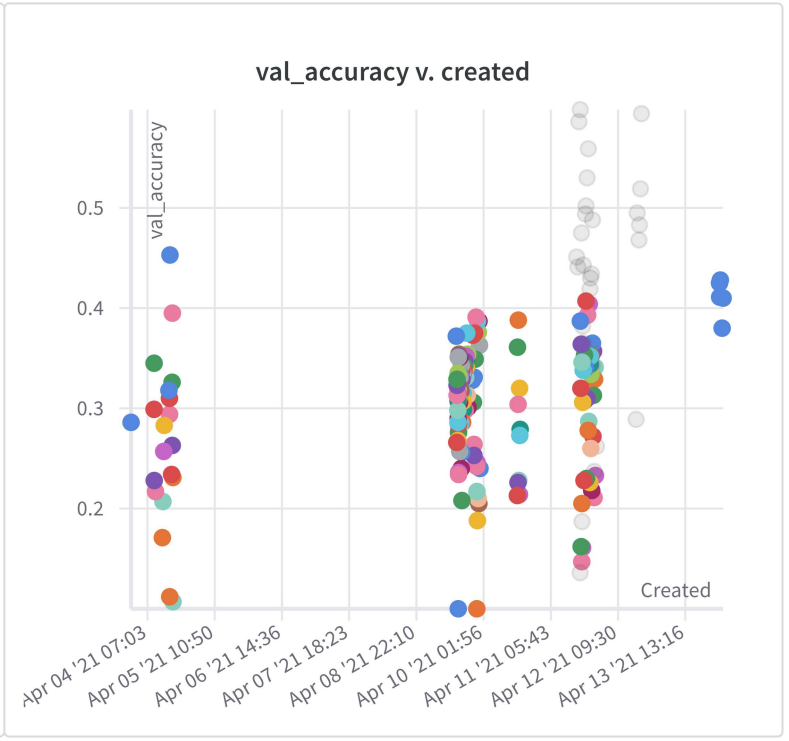
The hyperparameters considered for sweep as as:

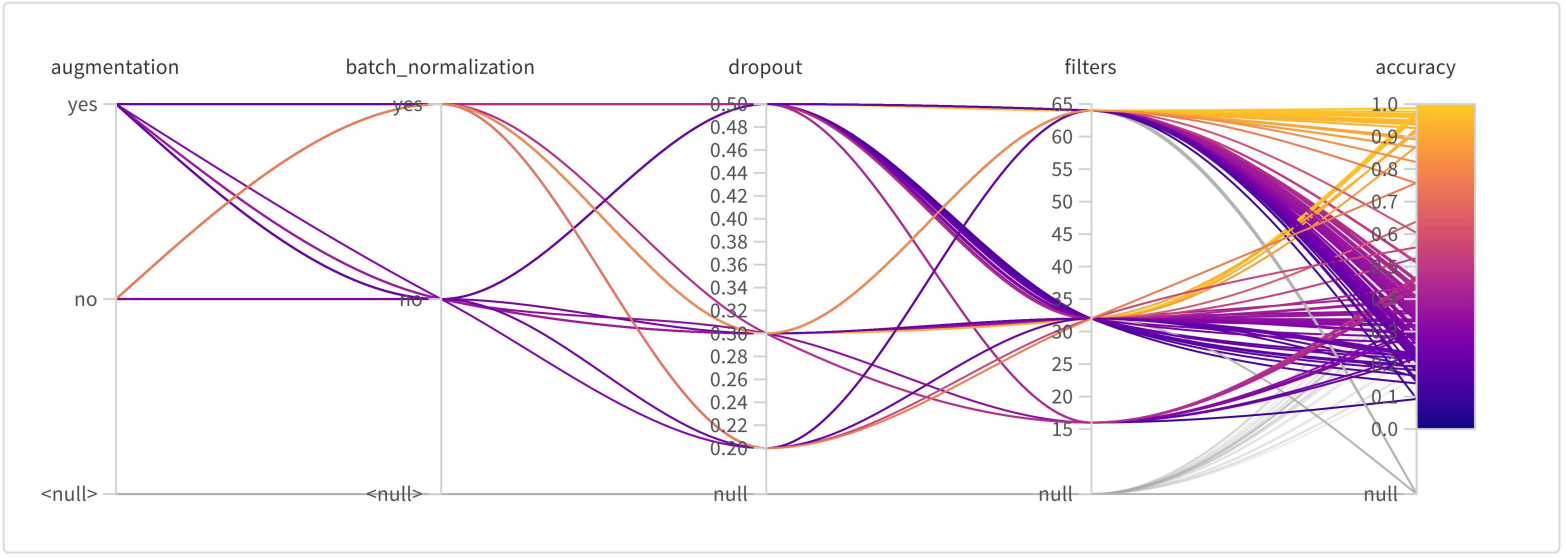
- Number of Filters in each Layer: **16, 32, 64**
- Filter Organisation: **Same, Double, Half (after each conv-layer)**
- Data Augmentation: **Yes, No**
- Dropout: **0.2, 0.3, 0.5**
- Batch Normalization **Yes, No**

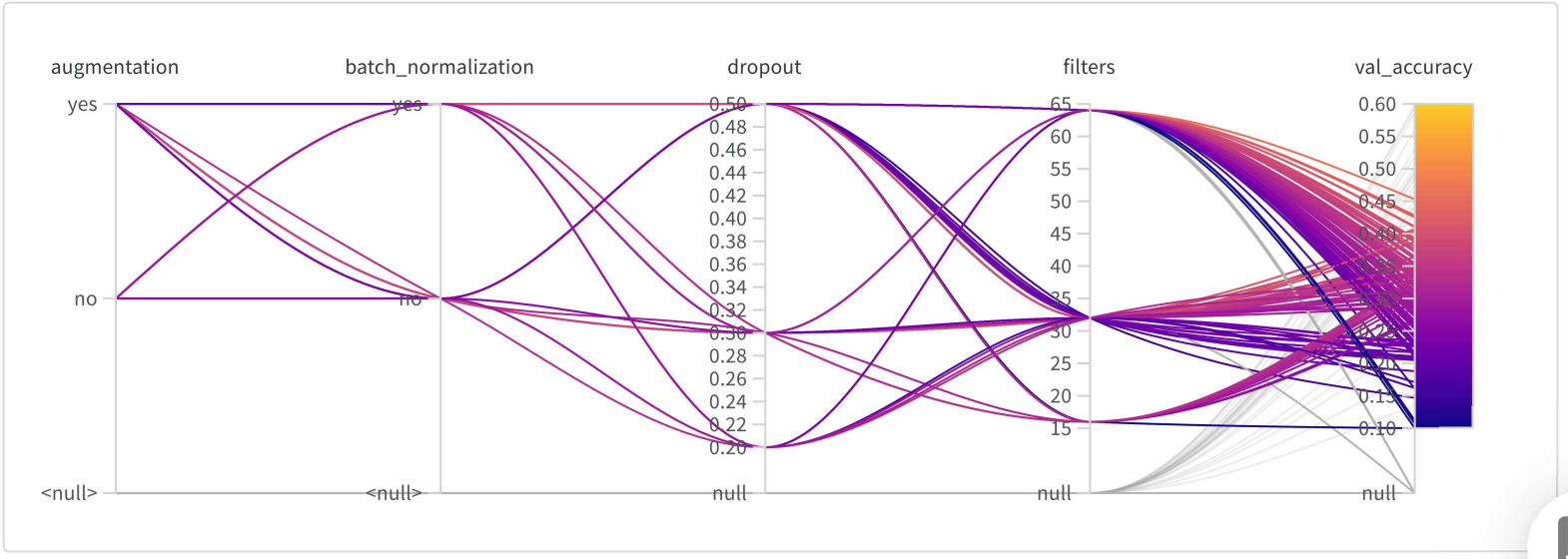
Plots from Hyperparameter Tuning

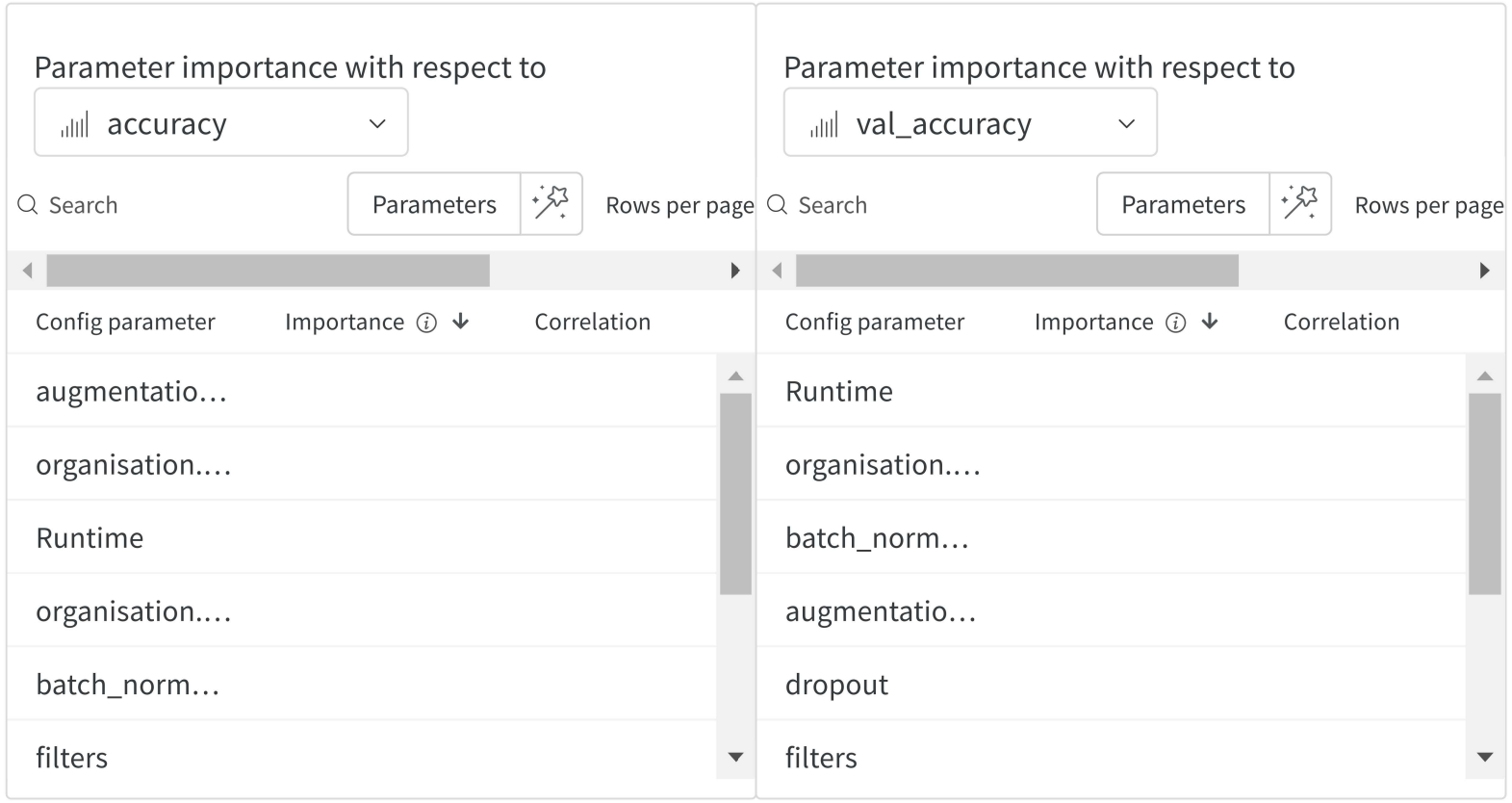



accuracy v. created
accuracy
0.8
0.6
0.4
0.2
Created
Apr 04 '21 07:03
Apr 05 '21 10:50
Apr 06 '21 14:36
Apr 07 '21 18:23
Apr 08 '21 22:10
Apr 10 '21 01:56
Apr 11 '21 05:43
Apr 12 '21 09:30
Apr 13 '21 13:16


val_accuracy v. created
val_accuracy
0.5
0.4
0.3
0.2
Created
Apr 04 '21 07:03
Apr 05 '21 10:50
Apr 06 '21 14:36
Apr 07 '21 18:23
Apr 08 '21 22:10
Apr 10 '21 01:56
Apr 11 '21 05:43
Apr 12 '21 09:30
Apr 13 '21 13:16


augmentation
batch_normalization
dropout
filters
accuracy
yes
no
<null>
yes
no
<null>
0.50
0.48
0.46
0.44
0.42
0.40
0.38
0.36
0.34
0.32
0.30
0.28
0.26
0.24
0.22
0.20
null
65
60
55
50
45
40
35
30
25
20
15
null
1.0
0.9
0.8
0.7
0.6
0.1
0.0
null


augmentation
batch_normalization
dropout
filters
val_accuracy
yes
no
<null>
yes
no
<null>
0.50
0.48
0.46
0.44
0.42
0.40
0.38
0.36
0.34
0.32
0.30
0.28
0.26
0.24
0.22
0.20
null
65
60
55
50
45
40
35
30
25
20
15
null
0.60
0.55
0.50
0.45
0.10
null



Strategies to Reduce Runs

As we can see above, we will have exponential number of hyperparameters combinations therefore, to perform hyperparameter tuning efficiently we used,

1. Wandb's **Bayes Optimization** search strategy which uses gaussian function to model the function and then chooses parameter to optimize probability of improvement. Here we are optimizing (maximizing) the **validation accuracy.**
2. Wandb's **Hyperband** stopping criteria. It speeds up hyperparameter search by stopping poorly performing runs.

Observations & Inferences

- From above sweep we can conclude that **Double** organisation gives better accuracy as compared to other organizations. For double organisation we got validation accuracy of **0.453**. It tells that adding more filters in the deeper layers is better . It may be due to abstraction of more complex patterns from the previous layer.
- **Halving** the number of filters in each subsequent layer produces worse results. This is obvious as the deeper layers should have more number of filters to capture more patterns in data. In fact WandB automatically stopped almost all **Half** organisation runs since we used early stopping and it didn't gave good accuracy.
- **Augmentation** seems to work really well in preventing overfitting on the training data. Without data augmentation, the training accuracy reaches maximum of **0.9871,** while validation accuracy was just **0.347** . This is a clear cut case of overfitting, in fact without augmentation validation accuracy does not even cross **0.4** but by augmentation validation accuracy reached **0.40 - 0.45.**
- **Dropout** is more preferable to add in the dense layer because these layers have high density of parameters. Adding dropout to low density convolutional layers might underfit the model by not learning enough patterns from the data. Based on above sweep we found dropout of **20%** to be more preferable than **30%** & **50%** (Our best model is trained with this value).
- On average the model requires **25-30 epochs** to train after which the model almost converges. epochs causes underfitting resulting in low accuracy while high epochs sometimes decreases t

validation accuracy and increases validation loss.

- In many runs, the validation accuracy first increases, then decreases after few epochs. This is due to mainly overfitting of the data. As more and more epochs, more the overfitting and hence less generalization to the validation data.

Model Performance on Test Data

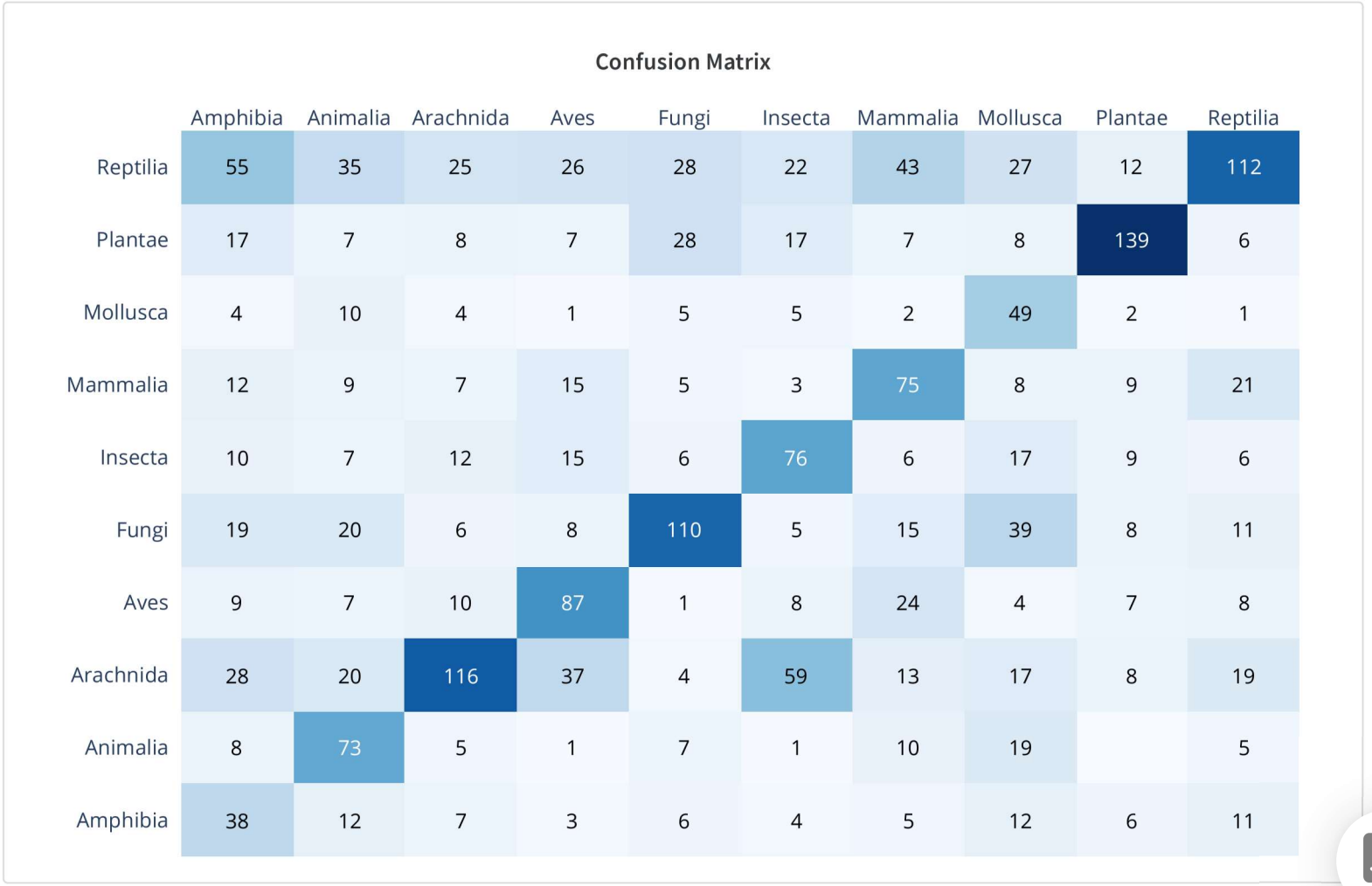
We trained our best model with the following configuration,

Conv Layers:	**5**
Activation Function	**ReLU**
Number of Filters:	**64**
Neurons in Dense Layer	**128**
Filter Organisation:	**Double (After every Conv Layer)**
Filter Size:	**3 x 3**
Max Pool Filter Size:	**2 x 2**
Augmentation:	**Yes**
Dropout	**20% (After 1st Dense Layer)**
Batch Normalization:	**Yes**
Batch Size	**128**

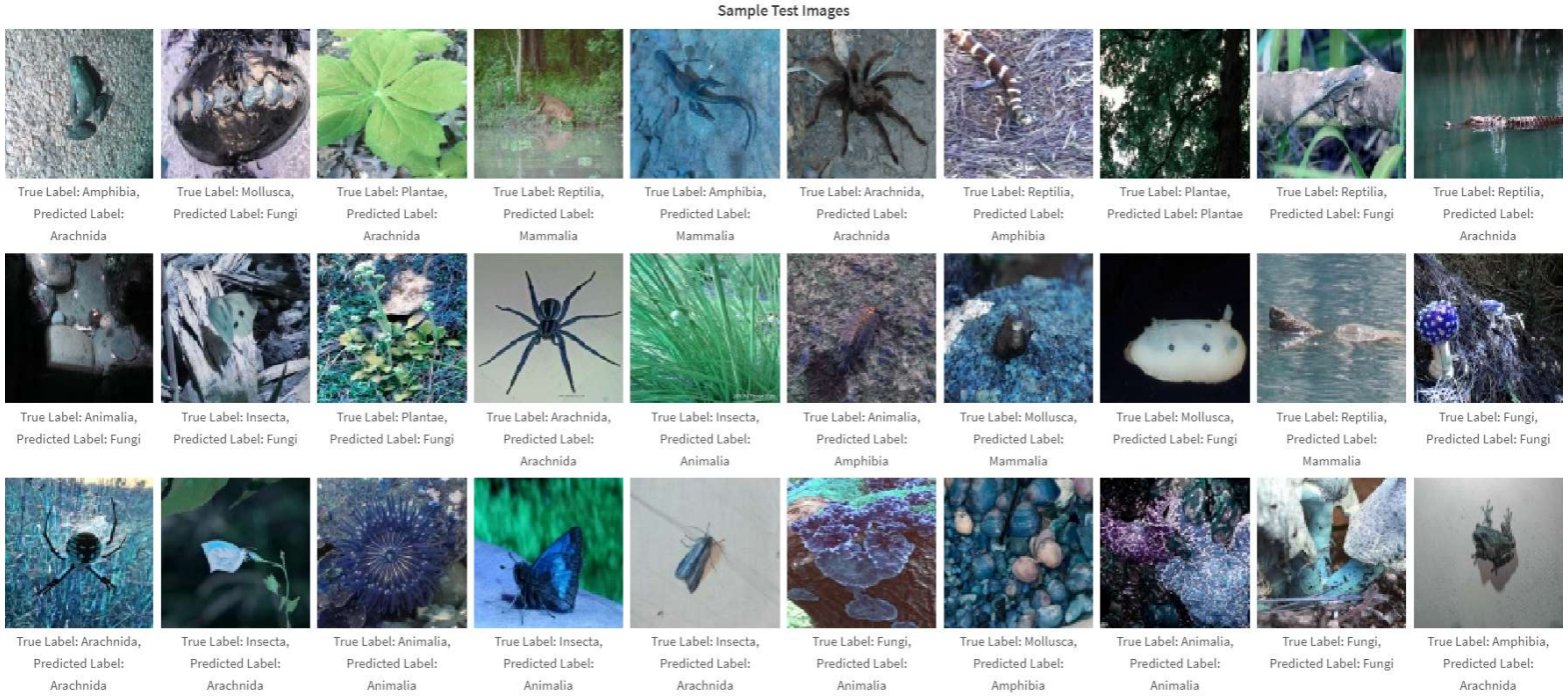
Statistics of our Model

Confusion Matrix

Here **x-axis** corresponds to **True Labels** and **y-axis** corresponds to **Predicted Labels.**

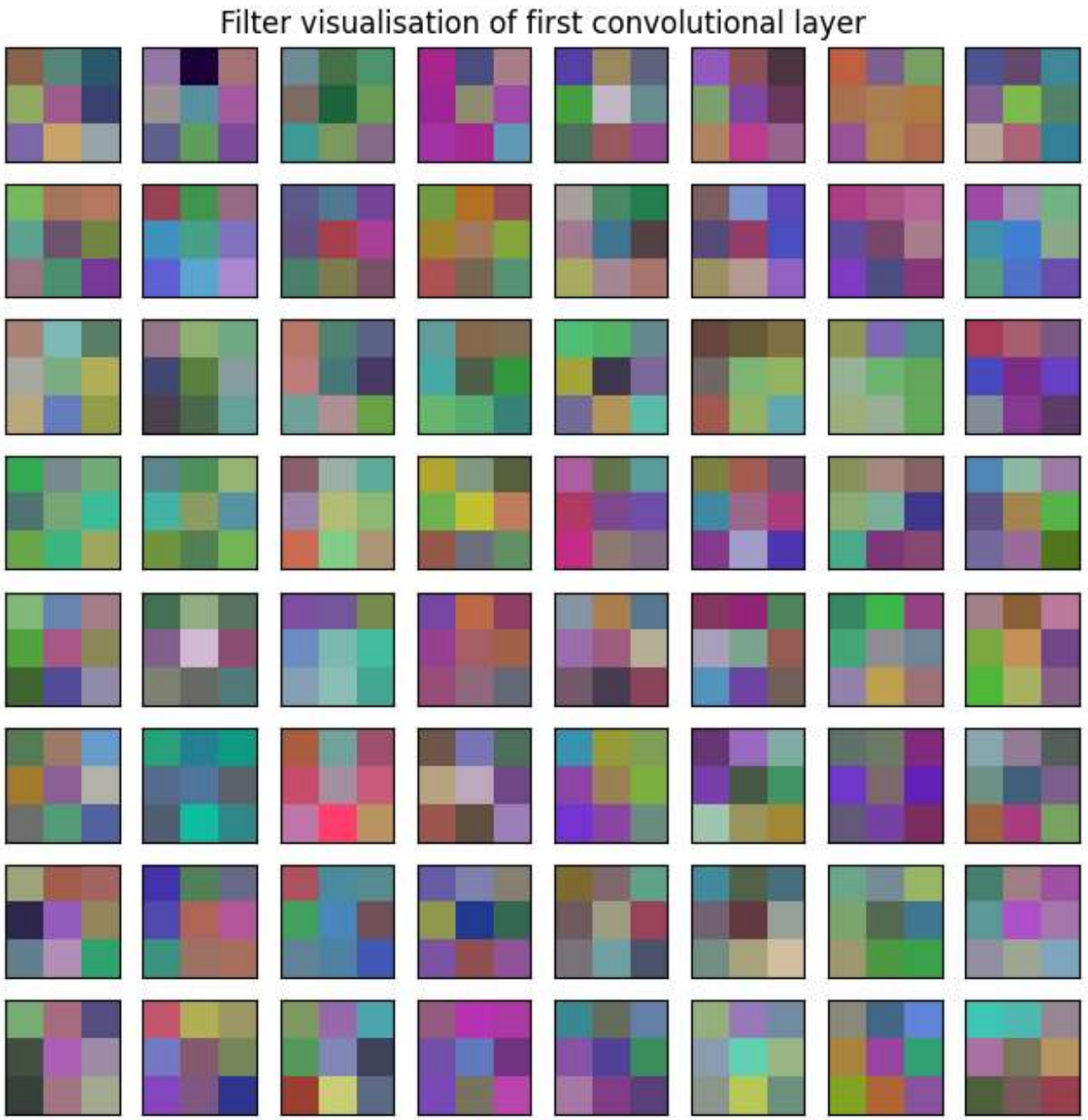


Randomly Sampled Test Images & Predictions made by our Model



Filter Visualization

We have visualized all the **filters** in the **1st layer** of our best model. There are **64 filters** of **size 3 x 3** in the first layer of our model which we have plotted in an 8 x 8 grid.

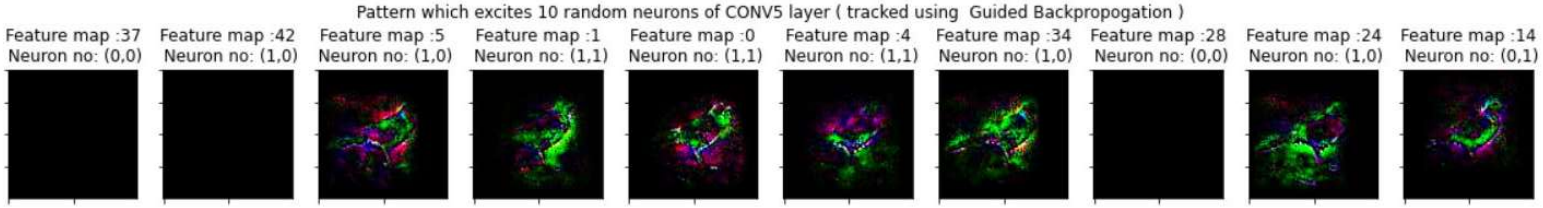

Filter visualisation of first convolutional layer

Guided Backpropagation

We applied guided back propagation on random 10 neurons in the CONV5 layer for an image and plotted the images which excite this neuron.

Original Image

Images which excites Neurons



Github Link

The code for this part can be found at
https://github.com/rigvedsah000/CS6910/tree/master/Assignment%202/Part%20A

We will include a README file at root of Assignment 2 folder on how to run the code.

Part B: Fine Tuning a pre-trained Model

We loaded some pre-trained models such as InceptionV3, InceptionResNetV2, ResNet50, Xception and fine-tuned them using the i-Naturalist dataset. It means we used the already trained weights of the model that it learnt during training on ImageNet instead of initializing random weights. Since ImageNet dataset has different dimensions for input and output we made some changes to the model to make it work on our dataset as,

a.) The dimensions of the images in our data are not the same as that in the ImageNet data. To address this we used the provided the shape (dimension) of our input data to the respective model function while initializing the model i.e.

```
model = InceptionV3(input_shape = (h, w, d), weights = "imagenet", include_top = False)
```

b.) ImageNet has 1000 classes and hence the last layer of the pre-trained model would have 1000 nodes. However, the i-Naturalist dataset has only 10 classes. To address this we require two steps as,

1. Remove the last layer (output layer) from the pre-trained model by passing `include_top = False` to model initializer as,

```
model = InceptionV3(input_shape = (h, w, d), weights = "imagenet", include_top = False)
```

2.) Flatten the output of the `model` and create a new new **Dense layer having 10 neurons** representing the **10 classes** of our datasets, use **softmax** as activation function and create a new model `new_model` using `model` with output as the layer we created now as,

```
new_model = Model(inputs = model.input, outputs = Dense(n_labels, activation = "softmax")(Flatten
```

The code is written in a very flexible manner that allows to swap in any model from InceptionV3, InceptionResNetV2, ResNet50, Xception.

```
model = get_pre_trained_model(code, h, w, d)
```

where `code` can be anything from `inception_v3, inception_resnet_v2, resnet_50, xception` whichever model we want to implement.

Making Training Tractable

Since InceptionV3, InceptionResNetV2, ResNet50, Xception are very huge models so even fine-tuning on a small training data may be very expensive.

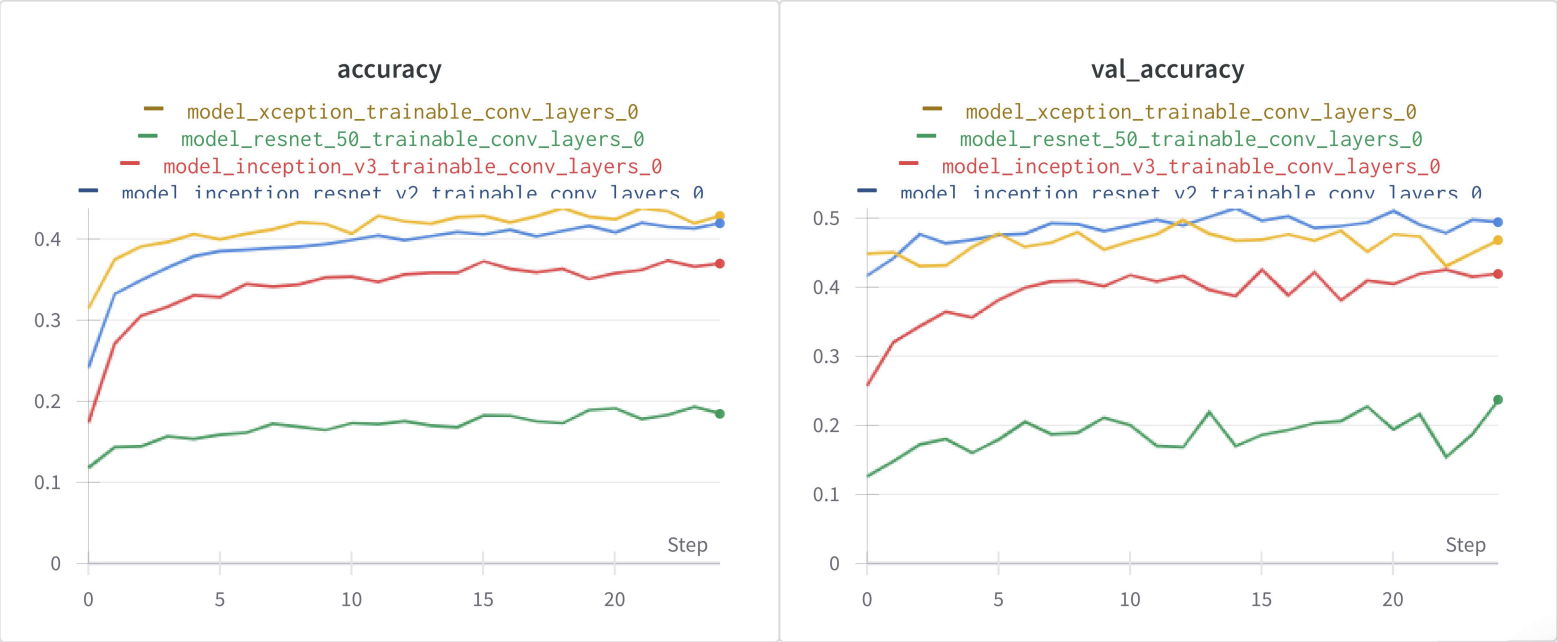
To make training tractable we used some strategies such as,

- **Freezing** all layers except the last layer and only training the last layer (output layer). This strategy is fast since only the last output layer has to be trained, thus reducing trainable parameters size.
- **Freezing** up to last k layers and **training** the rest layers. Since we used different *k(s)* for different models I am using k instead of writing a particular number. In essence we trained from last 1 & last 2 convolution layers and all layers afterwards for each model. For e.g. We trained InceptionV3 from k = 12 & k = 17 layers as it's 1st last and 2nd last conv-layers starts from there.
- **Training** all the layers. Here we trained the whole model.

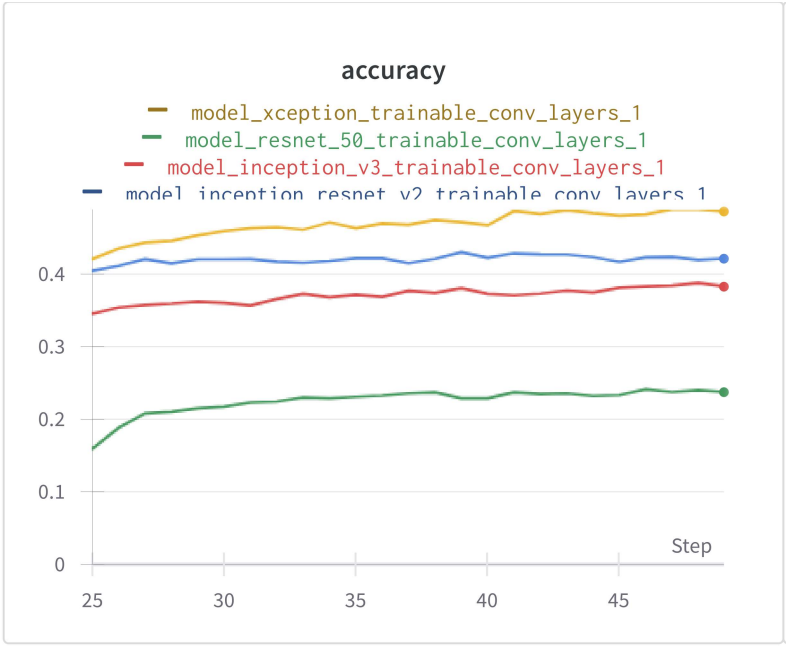
One thing to note is that before finetuning the model we have to train the output layer first and then start training the model. It is because if we do not train output layer which is initialized with random weights it and directly train the model, it will destroy the pre-trained weights that the model learnt from ImageNet dataset.

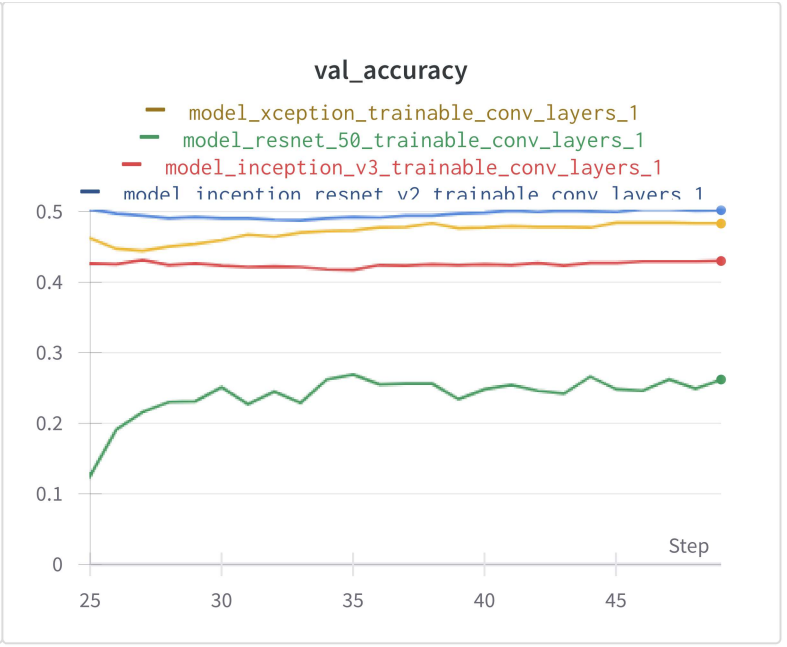
Fine-Tuning the Models

Freezed all layer & Trained only last layer

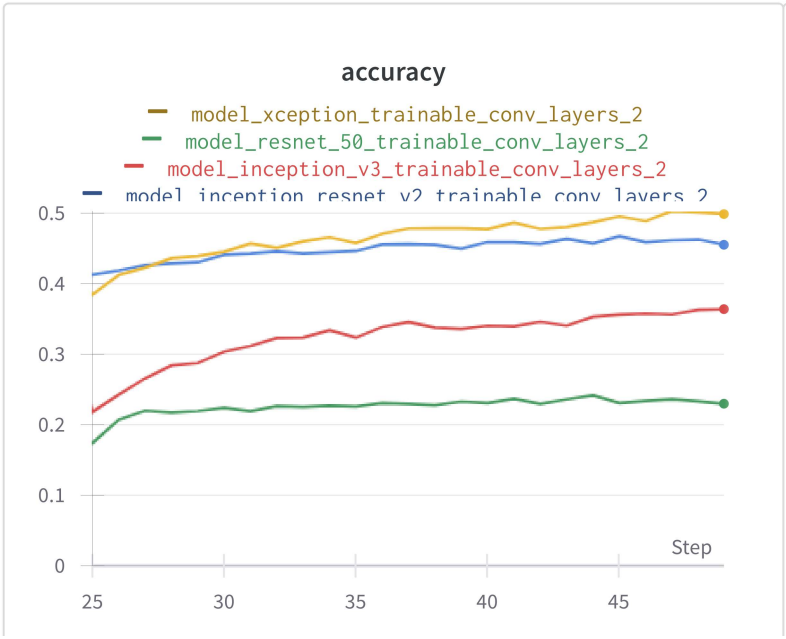


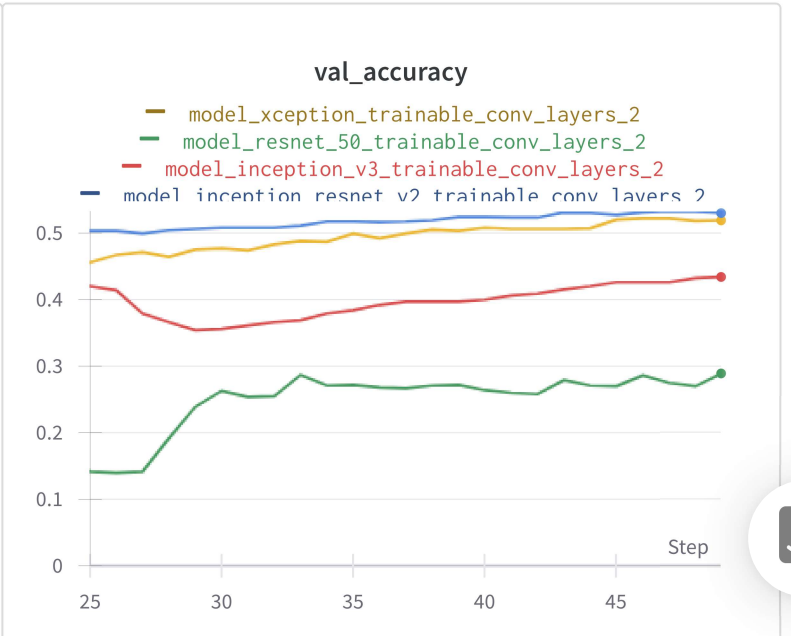
Trained from 1st last Convolutional layer up to Output Layer



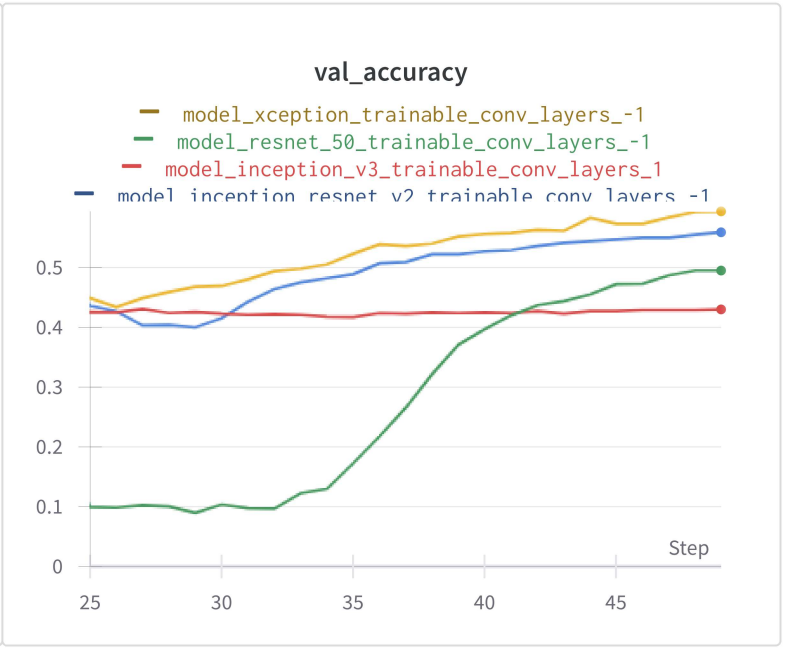


Trained from 2nd last Convolutional layer up to Output Layer





Trained on all layers



Observations & Inferences

- Using all pre-trained models except ResNet50 are giving better validation accuracy than training our own network.
- Xception is performing best and Resnet50 is performing worst amongst all of them.
- Even training on simply last layer pre-trained models are giving better accuracy and validation accuracy as compared to our model keeping other conditions same such as training data, epoc
using augmentation in training both the models.

- The highest validation accuracy reported for our model is **0.453** and highest accuracy reported for pre-trained model is **0.594** of Xception when trained completely.
- Our simple model leads to faster convergence as opposed to pre trained model such where we train some or all the layer in them.
- ResNet50 takes more than 35 epoch to properly train (validation accuracy to cross 0.3 mark)
- As expected on increasing the number of trainable layers (trainable parameters) the accuracy of our models increases as it adjusts the pre-trained weights to rely more on our dataset rather than ImageNet dataset.

Github Link

The code for this part can be found at
https://github.com/rigvedsah000/CS6910/tree/master/Assignment%202/Part%20B

We will include a README file at root of Assignment 2 folder on how to run the code.

Part C: Using a pre-trained Model as it is

We used a pre-trained YoloV3 model for object detection and used it in an application of our choice.

Application of YoloV3 Model

One problem faced by nearly every city in India is of herds of animals roaming free on busy road, streets and even on highways. Even big cities like Delhi, Mumbai, Ahmedabad face this issue. According to this article by India.com along 300 people died along in Punjab in past 2.5 years due to accidents caused by stray cows on roads. Not only humans but this article accounts that nearly

1K stray animals also lost their lives in past 4 months in Nagpur alone. And if this is not enough huge traffic jams have be known to be caused by animals on streets.

Now one solution can be that Municipal Corporation send their personnel to every busy road, every busy street in whole city for 24/7 looking for strays and inform them but as we all know this is infeasible and nearly impossible in big cities.

To solve this problem more efficiently we have a proposed a solution using a pre-trained YoloV3 model.

To implement this solution we can install Object Detection softwares such as YoloV3 on Municipal Corporation's servers that receives feeds from various CCTV cameras installed on roads, streets and highways. Now running the softwares on the footage from theses cameras we can easily detect the presence of animals on roads. More precisely we can even detect the type of animal and even detect the density of traffic. Based on detections we can generate an automated message to respective authorities such as traffic control or municipal corp to take swift action.

For e.g. if we detect the density of traffic thought a road is very high and a herd of cows are coming that way we can inform the nearby team to navigated cows away from streets.

Let's take some another example, if some wild animal like tiger is seen on cameras at night then right away we can inform Forest Department to take action.

This simple solution if implemented will not only save human and animal lives but also reduce the latency in manually informing authorities. It will also help to curb traffic jams due to roaming animals.

YouTube Link

We have made a demo video of our idea implemented. Please click this link to vide our demo video.

Other Applications

This simple can be extended to other applications as well. One such instance is as,

In many villages it is a common problem for framers and breeders that animals such as elephants destroyed their crops or tigers, wolves killed their goats and hens. This heavily impacts the farmer financially and mentally and is often a cause for their debt or even suicide.

Due to uncertainty of this problem before the Forest Dept. official even reach the scene to scare of or capture the animal maximum damage is done. Now we install cameras on boundaries and outskirts of forests we can inform the forest officials before the animals reach the farms and damage the crops.

Github Link

The code for this part can be found at
https://github.com/rigvedsah000/CS6910/tree/master/Assignment%202/Part%20C

We will include a README file at root of Assignment 2 folder on how to run the code.

Self-Declaration

Contributions of the two team members are as,

CS20M053 (50% Contribution)

- Implemented Pre-Trained Models
- Drew Observations & Inferences
- Set up sweep in wandb
- Plotted the Confusion Matrix

CS20M067 (50% Contribution)

- Implemented YoloV3 Model & Object Detection
- Drew Observations & Inferences
- Set up sweep in wandb
- Plotted Filter Visualization

We, **RIGVED SAH, CS20M053** and **SUMIT NEGI, CS20M067**, swear on our honor that the above declaration is correct.

https://wandb.ai/rigvedsah/Assignment%202/reports/Report-Assignment-2---Vmlldzo2MDQ1NDg