cs22m076 CS6910 - Assignment 2

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

Sai Sree Ram Putta cs22m076

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

- 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

In my network Input Size is $256 \times 256 \times 3$, and convolution layers have m filters with size to $k \times k$ and n neurons in the dense layer.

Across all convolution and max-pooling layer padding = 0, stride = 1 . Activation and max-pooling layers have no parameters.

The number of parameters in a single filter of a convolutional layer is equal to $k \times k$

We have m filters in each convolutional layer so the number of parameters is equal to $m \times k^2$

There is a bias m in each convolutional layer.

As the input Depth is 3 total number of parameters in convolutional layer1 is equal to $3 \times m \times k^2 + m$

For the remaining 4 convolutional layers depth is equal to the number of filters in the previous layer, the total number of parameters is equal to $4 \times (m^2 \times k^2 + m)$

The total number of parameters at the end of convolutional layers is equal to $3 \times m \times k^2 + 4 \times m^2 \times k^2 + 5 \times m$

Dense Layer parameters are equal to $n \times m \times (256 - 5(k + 1))^2 + n$

output layer parameters are equal to $10 \times n + 10$

Total number of **parameters** in the $3 \times m \times k^2 + 4 \times m^2 \times k^2 + 5 \times m + n \times m \times (256 - 5(k + 1))^2 + n + 10 \times n + 10$

To find the number of **computations** done by my network we have the number of parameters in that layer multiplied by the output height and width given by that layer.

Computations in 1st Convolution Layer are $3 \times m \times k^2 \times (256 - (k+1))^2$

Computations in the remaining Convolution Layer are $m^2 \times k^2 \times (256 - i(k+1))^2$ where i $\in 2, 3, 4, 5$

Computations in the Dense Layer are $n \times m \times (256 - 5(k+1))^2$

Computations in the Dense Layer are $10 \times n$

Each activation function layer takes an input of size h and outputs a size of h it will have a computation of h

Each max pool layer has a computation of inchannels*heightofinput/maxpoolkernalsize*widthofinput/maxpool kernalsize

All these are per each image.

If we have to find the number of computations for all the images = $(3 \times m \times k^2 \times (256 - (k+1))^2 + \Sigma_{i=2}^{5}(m^2 \times k^2 \times (256 - i(k+1))^2) + n \times m \times (256 - 5(k+1))^2 + 10 \times n) * (sizeoftrainingdata)$

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 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

hyperparameters used for sweep

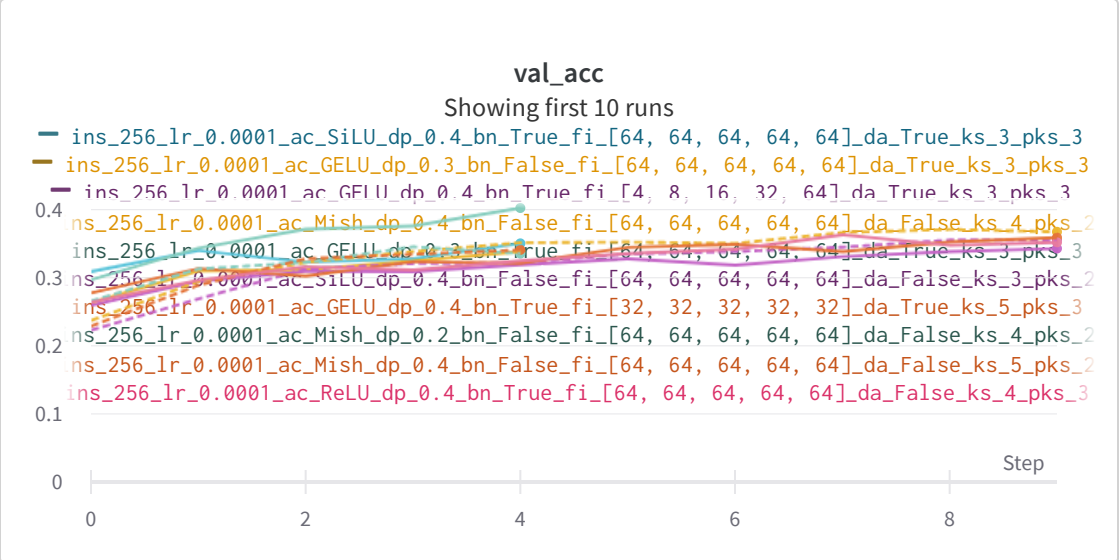
Bayesian Optimization:

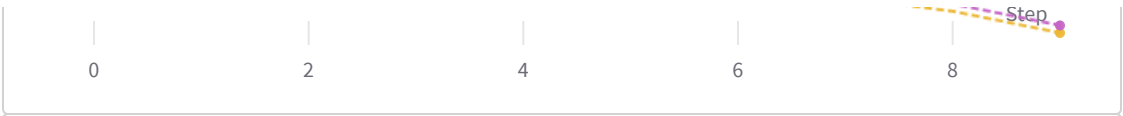
Bayesian optimization uses a probabilistic model to determine the next set of hyperparameters to evaluate based on the results of previous evaluations.

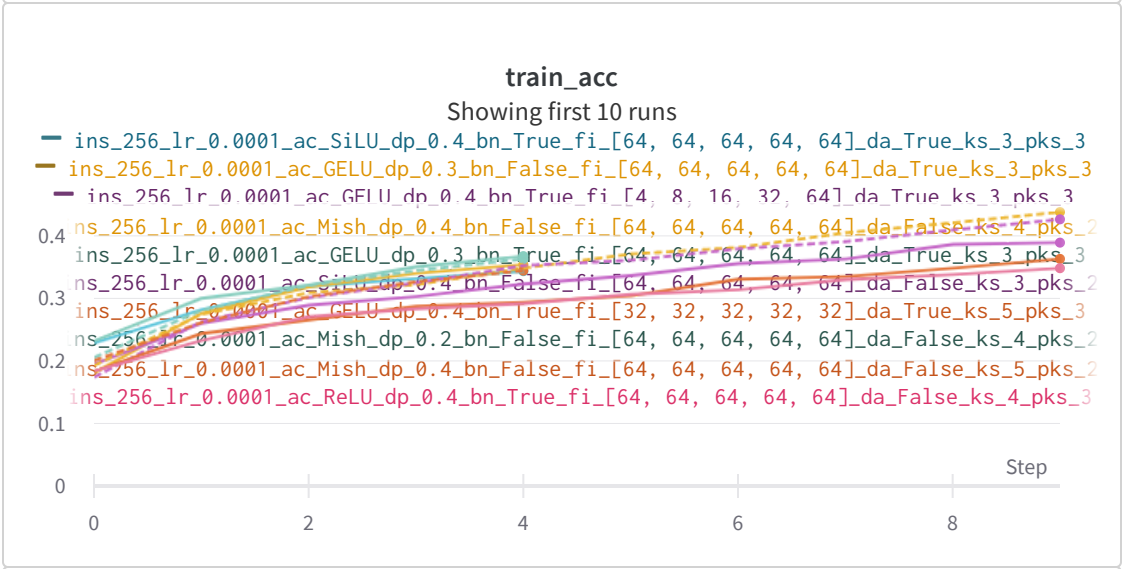
I utilized the Bayesian hyperparameter optimization search method in my study, as it is particularly effective when dealing with a large number of hyperparameters. The accuracy versus parameter graph was generated using this method. It is evident from the graph that the search method is attempting to find the best parameters by running only a few less accurate models. The Bayesian search method begins by selecting a random set of parameters and computing its accuracy or evidence. Based on this evidence, the

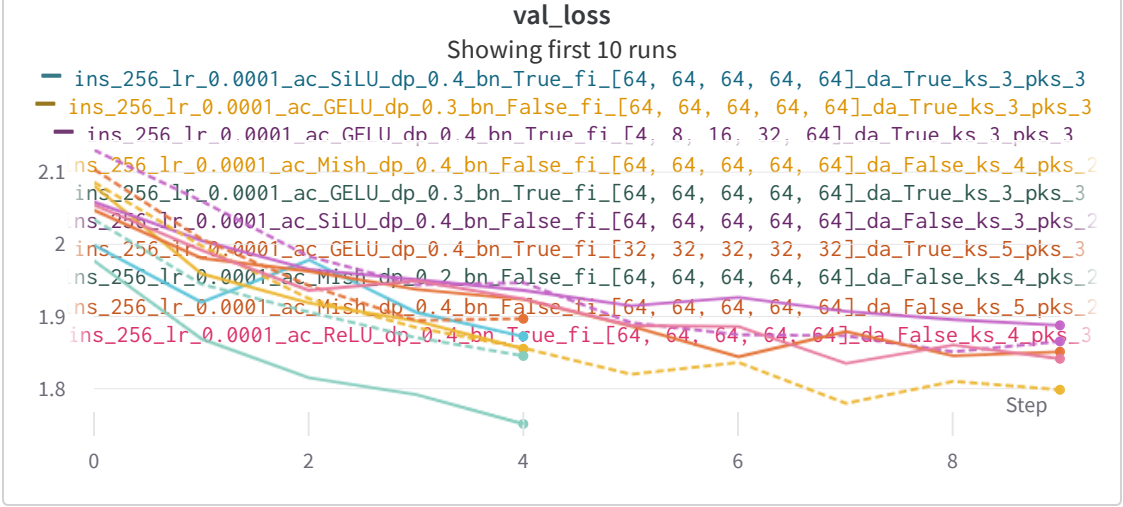
method adjusts its belief and selects the next parameter set to maximize the accuracy. This approach allows faster and more efficient identification of the best parameters while reducing the number of combinations that need testing.

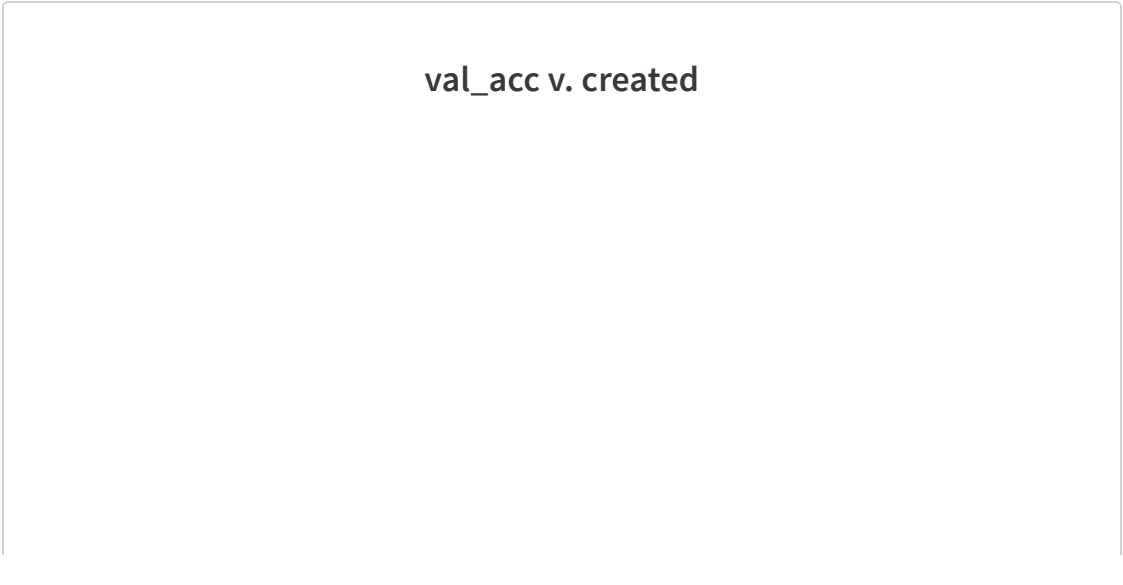
- number of epochs: 5, 10
- filter organization:[64,64,64,64,64],[32,32,32,32,32],[4,8,16,32,64], [64,32,16,8,4]
- kernel size : 3,4,5
- pool kernel size : 2,3
- learning rate: 1e-3, 1 e-4
- data augmentation: Yes, No
- batch normalisation: Yes, No
- dropout:0.2,0.3,0.4
- activation functions: ReLU, GELU, SiLU, Mish

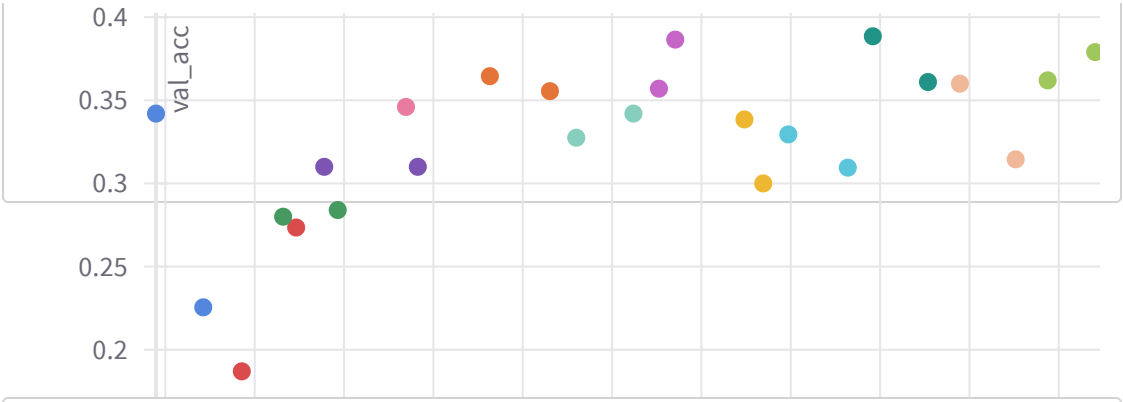


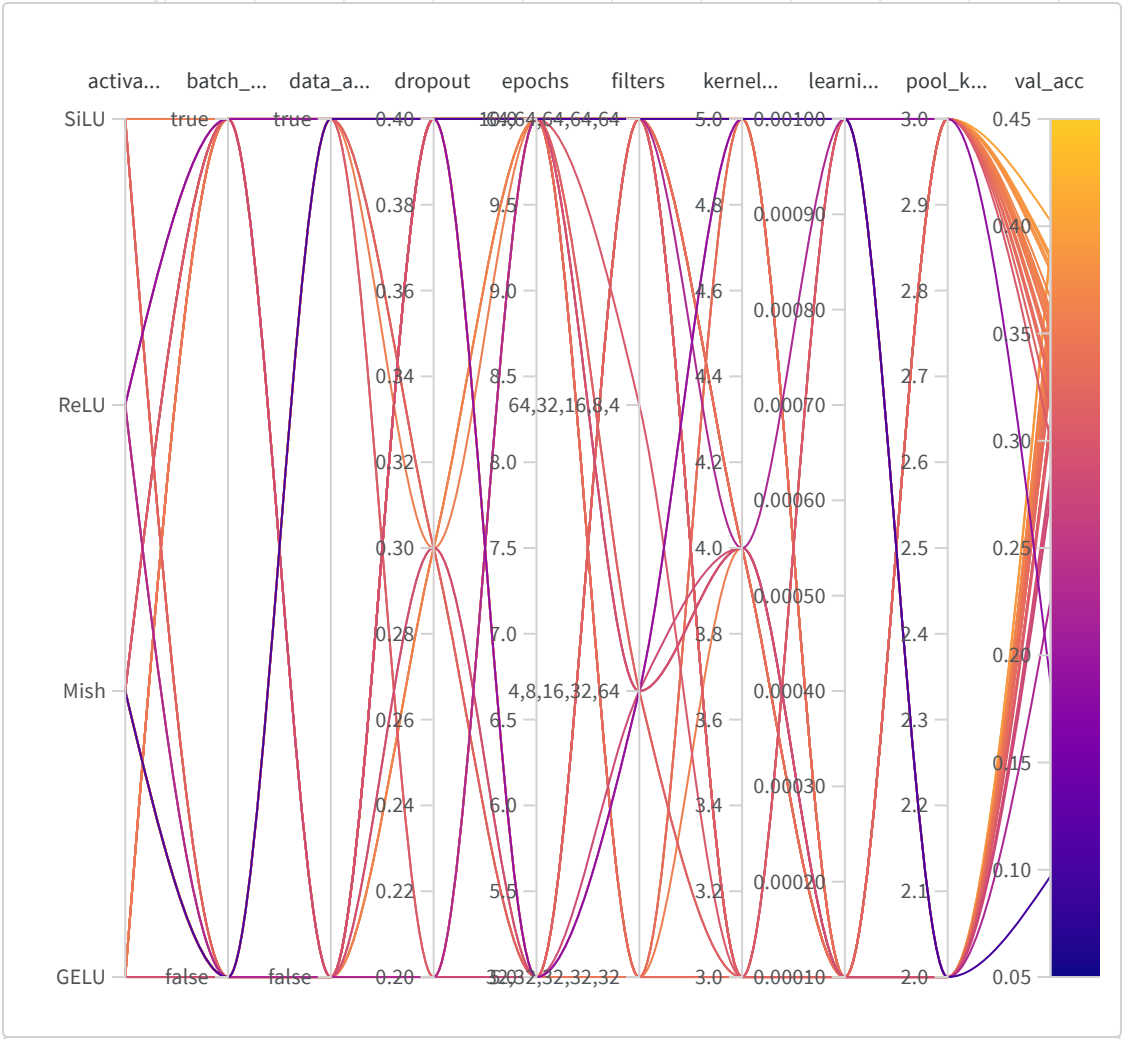

Step
0
2
4
6
8


train_acc
Showing first 10 runs
ins_256_lr_0.0001_ac_SiLU_dp_0.4_bn_True_fi_[64, 64, 64, 64, 64]_da_True_ks_3_pks_3
ins_256_lr_0.0001_ac_GELU_dp_0.3_bn_False_fi_[64, 64, 64, 64, 64]_da_True_ks_3_pks_3
ins_256_lr_0.0001_ac_GELU_dp_0.4_bn_True_fi_[4, 8, 16, 32, 64]_da_True_ks_3_pks_3
ins_256_lr_0.0001_ac_Mish_dp_0.4_bn_False_fi_[64, 64, 64, 64, 64]_da_False_ks_4_pks_2
ins_256_lr_0.0001_ac_GELU_dp_0.3_bn_True_fi_[64, 64, 64, 64, 64]_da_True_ks_3_pks_3
ins_256_lr_0.0001_ac_SiLU_dp_0.4_bn_False_fi_[64, 64, 64, 64, 64]_da_False_ks_3_pks_2
ins_256_lr_0.0001_ac_GELU_dp_0.4_bn_True_fi_[32, 32, 32, 32, 32]_da_True_ks_5_pks_3
ins_256_lr_0.0001_ac_Mish_dp_0.2_bn_False_fi_[64, 64, 64, 64, 64]_da_False_ks_4_pks_2
ins_256_lr_0.0001_ac_Mish_dp_0.4_bn_False_fi_[64, 64, 64, 64, 64]_da_False_ks_5_pks_2
ins_256_lr_0.0001_ac_ReLU_dp_0.4_bn_True_fi_[64, 64, 64, 64, 64]_da_False_ks_4_pks_3
0.4
0.3
0.2
0.1
0
Step
0
2
4
6
8


val_loss
Showing first 10 runs
ins_256_lr_0.0001_ac_SiLU_dp_0.4_bn_True_fi_[64, 64, 64, 64, 64]_da_True_ks_3_pks_3
ins_256_lr_0.0001_ac_GELU_dp_0.3_bn_False_fi_[64, 64, 64, 64, 64]_da_True_ks_3_pks_3
ins_256_lr_0.0001_ac_GELU_dp_0.4_bn_True_fi_[4, 8, 16, 32, 64]_da_True_ks_3_pks_3
ins_256_lr_0.0001_ac_Mish_dp_0.4_bn_False_fi_[64, 64, 64, 64, 64]_da_False_ks_4_pks_2
ins_256_lr_0.0001_ac_GELU_dp_0.3_bn_True_fi_[64, 64, 64, 64, 64]_da_True_ks_3_pks_3
ins_256_lr_0.0001_ac_SiLU_dp_0.4_bn_False_fi_[64, 64, 64, 64, 64]_da_False_ks_3_pks_2
ins_256_lr_0.0001_ac_GELU_dp_0.4_bn_True_fi_[32, 32, 32, 32, 32]_da_True_ks_5_pks_3
ins_256_lr_0.0001_ac_Mish_dp_0.2_bn_False_fi_[64, 64, 64, 64, 64]_da_False_ks_4_pks_2
ins_256_lr_0.0001_ac_Mish_dp_0.4_bn_False_fi_[64, 64, 64, 64, 64]_da_False_ks_5_pks_2
ins_256_lr_0.0001_ac_ReLU_dp_0.4_bn_True_fi_[64, 64, 64, 64, 64]_da_False_ks_4_pks_3
2.1
2
1.9
1.8
Step
0
2
4
6
8


val_acc v. created





Parameter importance with respect to val_acc

Search | Parameters | 1-10 of 16

Config parameter	Importance	Correlation
learning_rate		
kernel_size		
batch_normalisation		

Runtime
activation.value_ReLU

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

- Validation accuracy is significantly affected by the learning rate, and a decrease in the learning rate typically leads to an increase in validation accuracy.
- As kernel size increases validation accuracy decreases, and a decrease in the kernel size typically leads to an increase in validation accuracy.
- If we increase the size of the image accuracy also increases but the time taken for training will be more times it will go out of memory because of this.
- Tried different optimizers but Adam is better than others.
- The model is overfitting for most of the sweep configs with higher training accuracy and very low validation accuracy.
- The activation function GELU is better than other activation functions.

- Dropout prevents overfitting by randomly dropping out neurons during training, and the absence of dropout can lead to overfitting and poor generalization performance.
- Data Augmentation improves the validation accuracy by stopping overfitting on training data.
- An increase in Maxpool kernel size has positive importance as we are reducing the parameters and likely overfitting.

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.

Solution

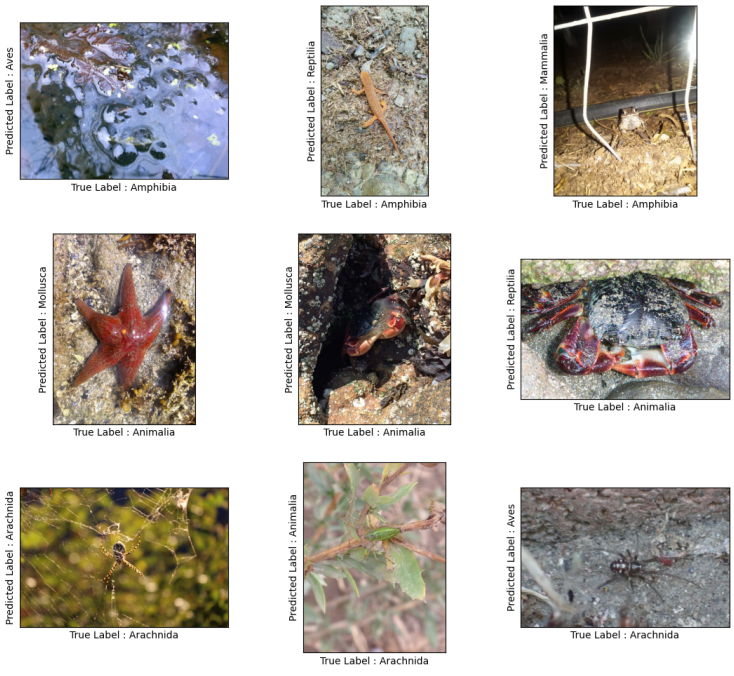
```
Learning rate  : 0.0001
Activation function hidden Layer : "GELU"
```

```
Optimizer : "ADAM"

Number of epochs : 5
Filter Organization: [64,64,64,64,64 ]
Batch size : 16
Batch Normalisation : True
Data Augmentation : True
dropout : 0.3
Image Size : 256
kernel Size(Filter)  : 3
Pool Kernel Size :3

Training Accuracy : 0.36% validation Accuracy : 40%
Test Acc : 38.9%
```

Question 4



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.

Solution

https://github.com/saisreeram-p/CS6910-Assignment02/tree/main/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?
- 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.)

Solution

The pre-trained models such as GoogLeNet, ResNet50, or VGG use 224 x 224 as image size.

So to use these pre-trained models we have to reshape the training and test images to the image size used by the pre-trained model.

Some pre-trained models need the image size to be atleast 224x224

We can replace the last layer of the pre-trained model with a new layer that has 10 nodes corresponding to the 10 classes in the Naturalist dataset, or add a new output layer with 10 classes on top of the pre-trained model with activation as softmax.

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).

Solution

The last layer is modified to match the number of classes in the iNaturalist dataset.

- **Fine-tuning with Freezing all layers except the Last Layer:** By keeping these layers fixed, the model can save computational resources and training time while still being able to adapt to a new task. This technique is useful when the new dataset is small and similar to the original dataset used to pre-train the model. The lower layers contain general features that are transferable to the new task.
- **Fine-tuning with K frozen layers:** involves freezing a specified number of K layers in the pre-trained model and updating only the weights of the remaining layers during training is beneficial when dealing with a small new dataset. This technique is particularly useful because the early layers of the pre-trained model often hold general features relevant to the new task. By freezing them, the model can utilize this existing knowledge.
- **Full fine-tuning:** involves training the entire pre-trained model on a new dataset, which includes updating the weights of all layers during training. This method is computationally expensive and requires a large amount of training data, as it involves updating the early layers that learn general features as well as the later layers that learn task-specific features.
- **Adding new layers on top of pre-trained model network**

Question 3 (10 Marks)

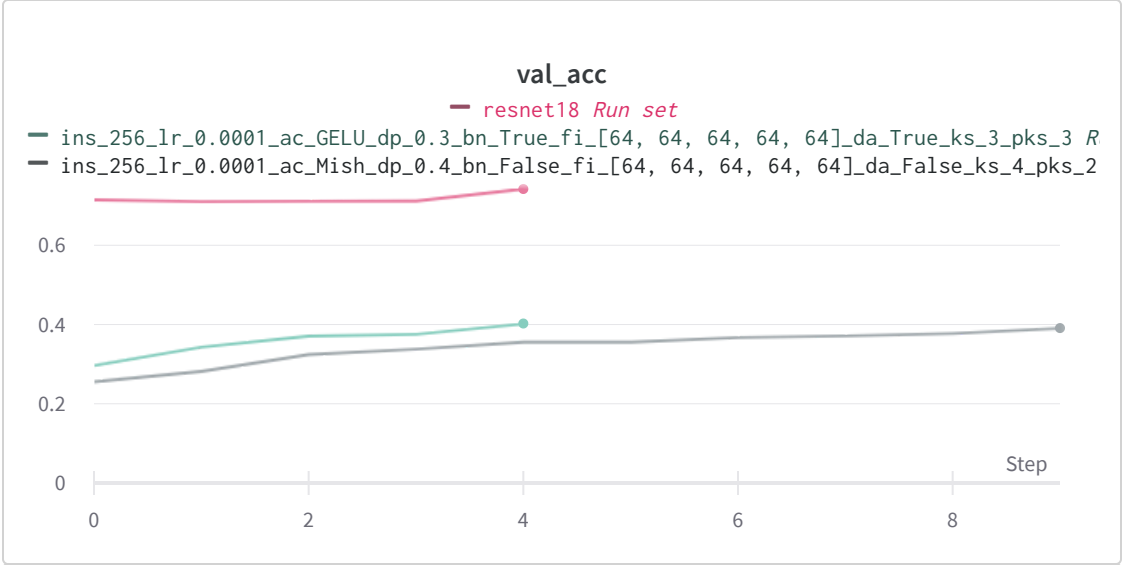
Now fine-tune the model using **ANY ONE** of the listed strategies that you discussed above. Based on these experiments write down some

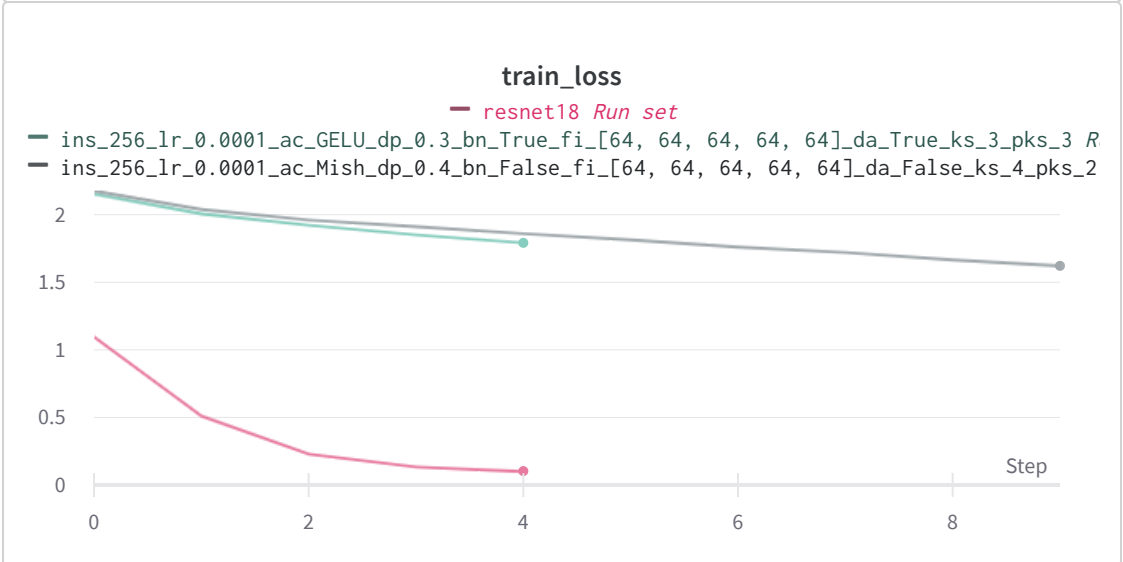
insightful inferences comparing training from scratch and fine-tuning a large pre-trained model.

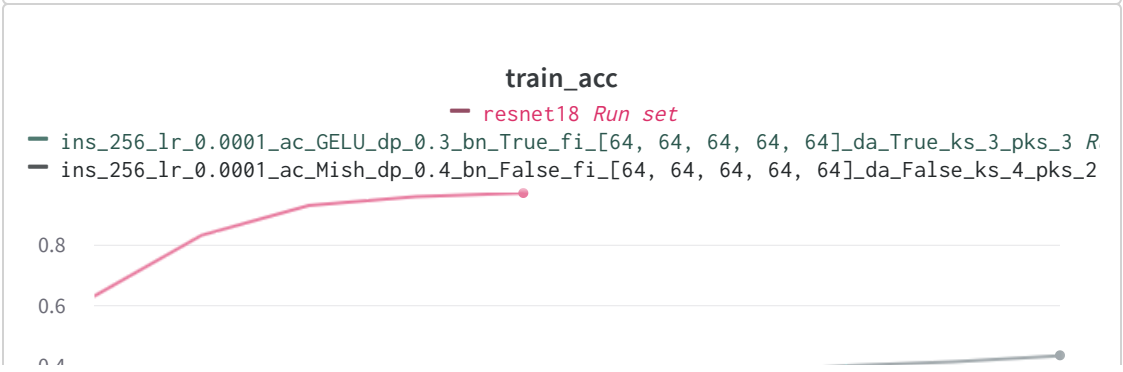
Solution

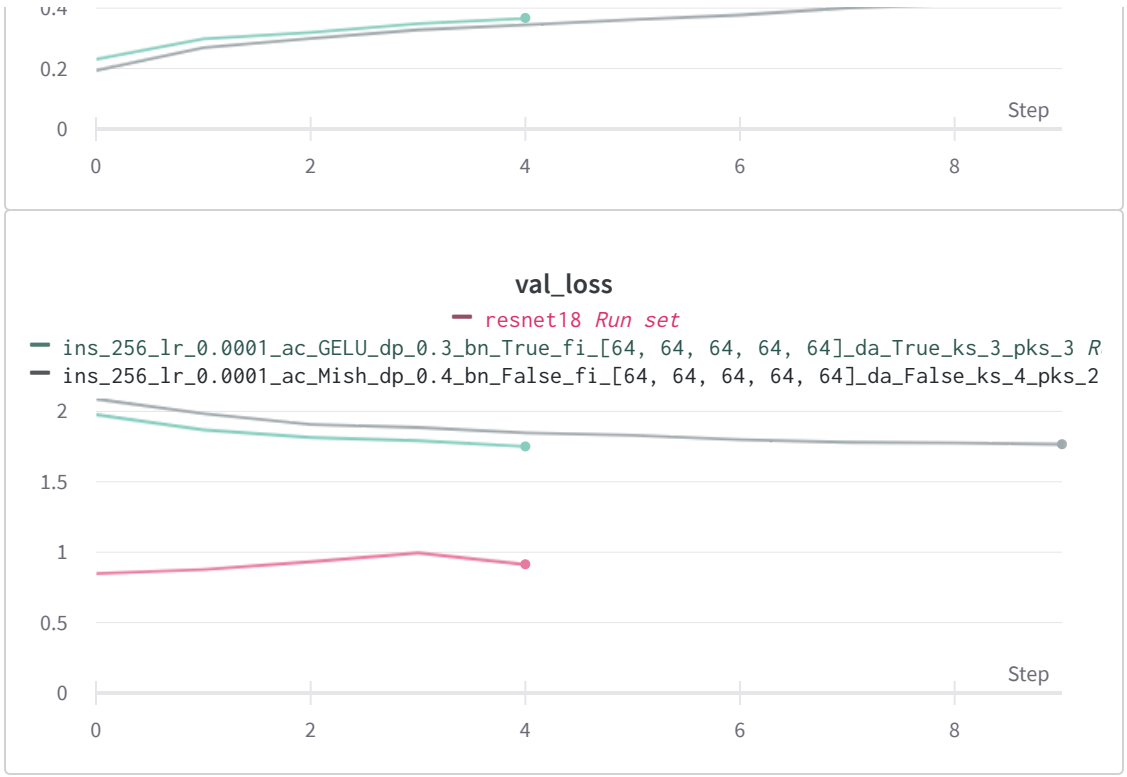
I have used **Fine-tuning with Freezing all layers except the Last Layer** method.

Below are the Wandb plots of the Resnet18 Fine tune model and the 2 best models from training from scratch.









By using a pre-trained model, we can take advantage of the knowledge that the model has already acquired from the Imagenet dataset, which can save time and resources during training and improve the performance of the model on new tasks.

Validation, Test data accuracy is better in the pre-trained model compared to training from scratch.

Time taken for training is less compared to training from scratch as weights of all the layers except the last layer are frozen.

The convergence rate is faster compared to training from scratch.

Slowing down the learning rate can lead to a slower convergence rate but can also improve the validation, and test accuracy.

This technique is useful when the new dataset is small and similar to the original dataset used to pre-train the model. The lower layers contain general features that are transferable to the new task.

Encountered overfitting while doing sweeps for training from scratch models with high training accuracy and very low validation accuracy which is not the case with the fine-tuning model.

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.

Solution

https://github.com/saisreeram-p/CS6910-Assignment02/tree/main/partB

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

Self Declaration

I, Sai Sree Ram (Roll no: CS22M076), 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.