CS6910 - Assignment 2

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

Aniket Keshri cs23m013

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)

Solution :

Number of parameters:

size for the input image : D x H x W

=> D - Depth(channel) , H - Height , W - weight

- `Final Image size after convolution` : ((W - F) + (2 * P)) / S + 1

  where , F - kernel size for max pool

  P - Padding (here 0 for all the layers)

  S - Stride (here 1 for all the layers)

- `Final Image after max pooling` : W + (2 * P) - (d * (F-1) - 1) / S + 1

  where , d - Dilation

  F - Kernel size in max pool

Now ,

Layer	Number of parameters
convolution L1	(D+ (k*k) +1)*m
convolution L2	(m+ (k*k) +1)*m
convolution L3	(m+ (k*k) +1)*m
convolution L4	(m+ (k*k) +1)*m
convolution L5	(m+ (k*k) +1)*m
Fully connected L1	(m*(H-5*F-5)*(W-5*F-5)+1)*n

therefore ,

- Total numbers of parameters : (D*k^2*m)+ (4*m^2*k^2) + 5*m + m * n * (H - 5*F - 5)(W - 5*F - 5) + n + c(1+n)

where , m= number of filters in each layer , k = size of filter (k*k)

now, for my configuration : `m = 8 , k = 3 , D = 3 , n = 16 , W = 256 , H = 256 , F = 1 , c = 10`

```
Convolution Layer 1 Weight Parameters: 216  , Convolution Layer Bias Parameters: 8

Convolution Layer 2 Weight Parameters: 576  ,  Convolution Layer Bias Parameters: 8

Convolution Layer 3 Weight Parameters: 576  ,  Convolution Layer Bias Parameters: 8

Convolution Layer 4 Weight Parameters: 576  ,  Convolution Layer Bias Parameters: 8

Convolution Layer 5 Weight Parameters: 576   ,  Convolution Layer Bias Parameters: 8

Convolution Layer 6 Weight Parameters: 7129088 , Convolution Layer Bias Parameters: 16

Convolution Layer 7 Weight Parameters: 160 , Convolution Layer Bias Parameters: 10

      : Resulting Parameters : 8074415
```

Number of computations done by network:

A. For Convolution Layers = `( (k * k )* D * ( W' * W' ) + 1 ) + m` , where : W' = Image size after Convolution operation and 1 added as bias.

B . For Dense Layers = `(W'' + 1) * n` , where W" = Number of neuron in Flatten layers and 1 added as bias.

therefore ,

Number of computations for : Layer 1: 13935464

Layer 2: 36000008

Layer 3: 34857224

Layer 4: 33732872

Layer 5: 32626952

Dense Layer: 7129104

Output Layer: 170

```
:Resulting Computations done by network : 657143459
```

Code for part A

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 :

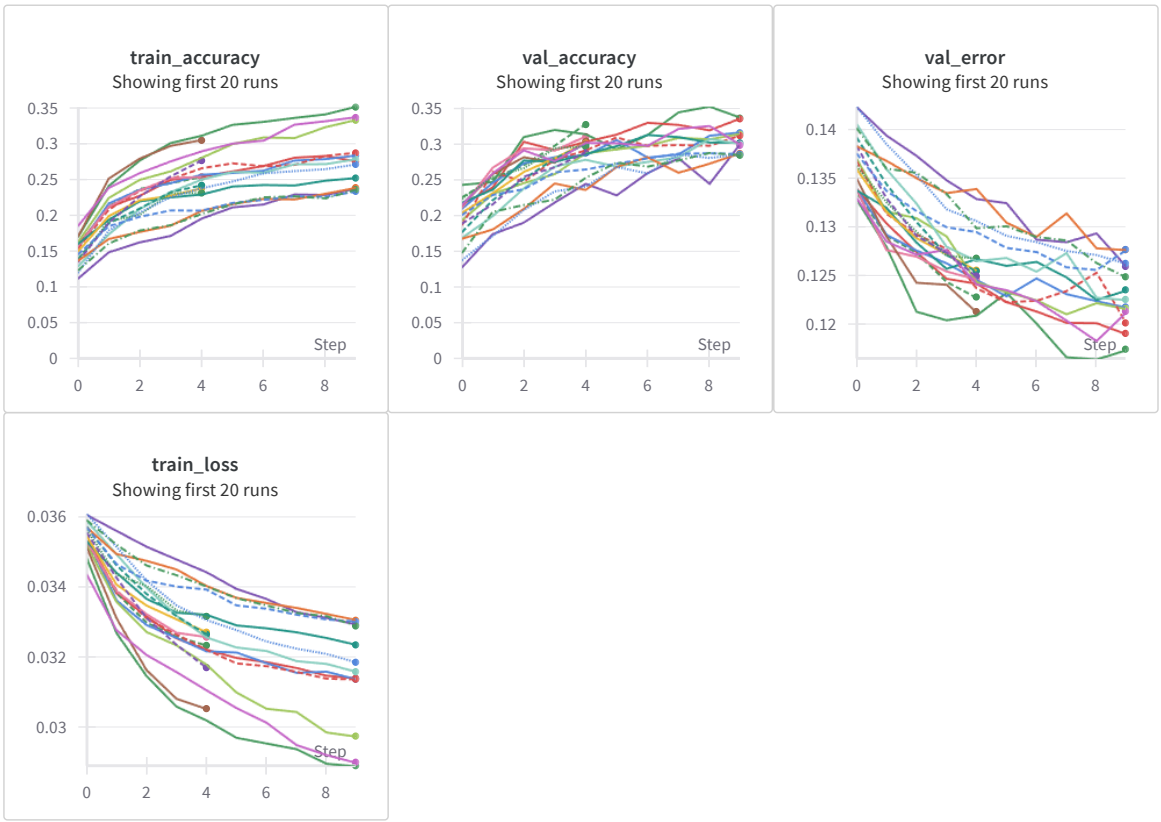
I used following configuration for my sweep :

```
config = {
  'name': 'cs23m013',
  'metric': {
    'goal': 'maximize',
    'name': 'Val accuracy'
    },
  "method": "random",
  "project": "Deep_Learning_A2",

  "parameters": {

     "image_size" : {
       "values": [224, 256]
    },
     "epochs": {
       "values": [5,10]
    },

    "activation_function":
    {
       "values" : ["LeakyReLU" ,"Mish", "SiLU", "GELU", "ReLU"]
    },
    "dropout_factor":
    {
       "values": [0 ,0.1 , 0.2 , 0.3, 0.4]
    },
    "num_of_filters": {
       "values": [32, 64]
```

```
        },
        "filter_multiplier": {
          "values": [0.3 , 0.4 , 0.5]
        },
        "learning_rate" : {
          "values" : [0.0001, 0.0003]
          },
        "apply_data_augmentation" : {
          "values" : [True]
          },
      }
    }
```

I run my sweep using above configuration in the random fashion and explored various hyperparameter and got to know which combinations of parameters results in required accuracy.

This sweep and best combination of parameter i got using this approach keeping in mind the limited resources i had , like the GPU which got overflowed in larger parameter space.

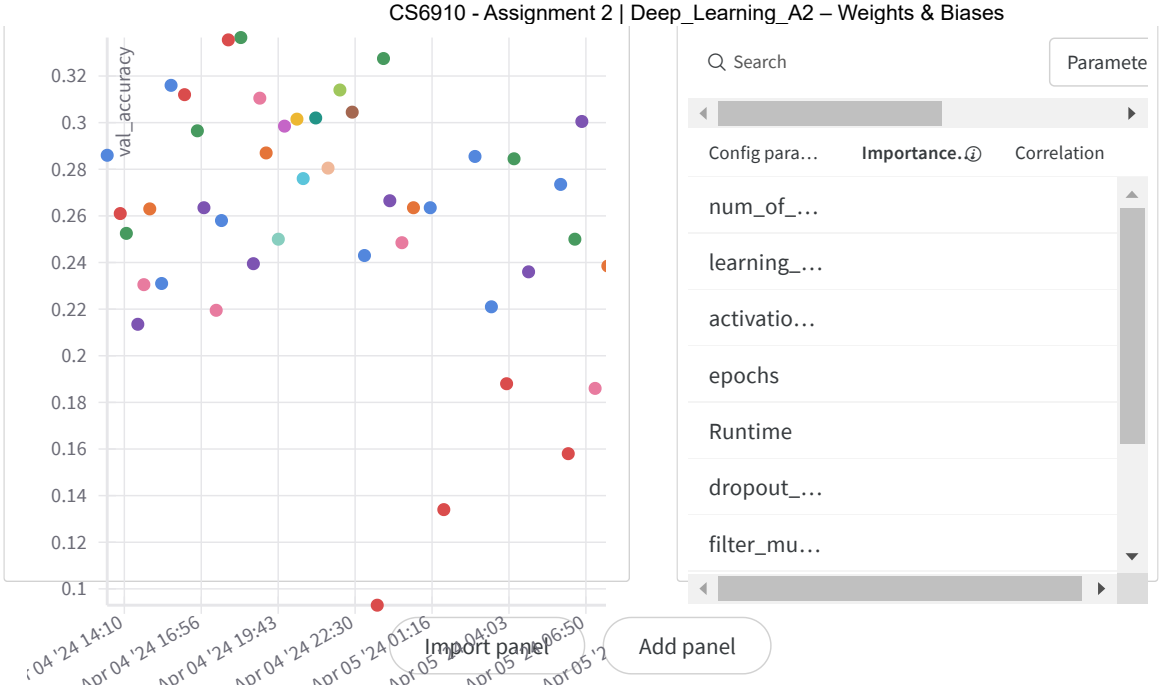


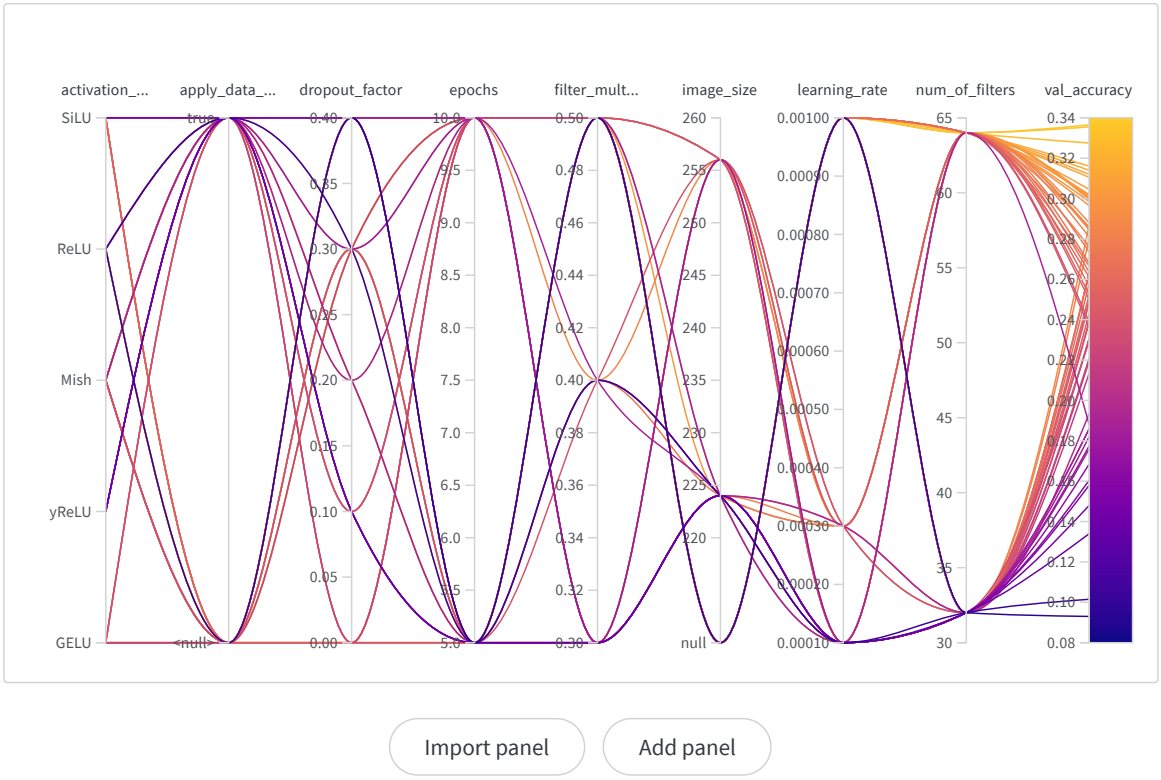
Import panel Add panel

val_accuracy v. created

Parameter importance with respect to

val_accuracy





Above plots visualize all my runs and its accuracy.

Now additionally i did this to improve my accuracy , on analyzing above visuals i constrained few parameters like epochs , image size and learning rate and then run in Bayesian optimization to select parameters with the aim to improve further the model's performance .

```
config = {
  'name': 'cs23m013',
  'metric': {
    'goal': 'maximize',
    'name': 'Val accuracy'
    },
  "method": "bayes",
  "project": "Deep_Learning_A2",

  "parameters": {

     "image_size" : {
       "values": [256]
     },
     "epochs": {
       "values": [10]
     },

     "activation_function":
     {
       "values" : ["LeakyReLU" ,"Mish", "SiLU", "GELU", "ReLU"]
     },
     "dropout_factor":
     {
       "values": [0 ,0.1 , 0.2 , 0.3, 0.4]
     },
     "num_of_filters": {
       "values": [32, 64]
     },
     "filter_multiplier": {
       "values": [0.3 , 0.4 , 0.5]
     },
     "learning_rate" : {
       "values" : [0.0001]
       },
     "apply_data_augmentation" : {
       "values" : [True]
       },
  }
}
```

Code for part A

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 :

1. **Number of filters expressed as a positive correlation**, whereas dropout factor and batch size expressed as a positive correlation with accuracy above.

2. **GELU** and **ReLU** activation functions generally overpowered others.

3. Batch normalization results a better results when applied after conv2d and on applying it after activation functions it didn't impacted much the accuracy.

4. **Dropout rates of 0.3 and 0.5** generally resulted better accuracy, preventing overfitting

5. ReLU and GELU activation functions generally outperformed others.

6. **Learning rate of 0.0001** with **GELU** results **better accuracy** compared to the 0.0003 learning rate.

7. I ran various experiments but achieving **accuracy** above **33% - 40%** was challenging due to the limited dataset size that is 8k training samples and 2k validation samples.

8. Increasing training dataset size to 9k-10k might improve validation accuracy above 40%.

9. Increased resources like GPU , processor might increase the accuracy as GPU memory crashes due to a high number of parameters.

10. Adding more filters improves performance for certain configurations.

11. **Adding few more convolution layers** and further max pool layers might **increase the 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.

Solution :

```
#Best Configuration
config = {
  'name': 'cs23m013',
  'metric': {
     'goal': 'maximize',
     'name': 'Val accuracy'
     },
  "method": "random",
  "project": "Deep_Learning_A2",

  "parameters": {
```

```
        "image_size" : {
          "values": [256]
        },
        "epochs": {
          "values": [10]
        },
        "activation_function":
        {
          "values" : ["ReLU"]
        },
        "dropout_factor":
        {
          "values": [0.2 ]
        },
        "num_of_filters": {
          "values": [ 64]
        },
        "filter_multiplier": {
          "values": [ 0.5]
        },
        "learning_rate" : {
          "values" : [0.0001]
          },
        "apply_data_augmentation" : {
          "values" : [True]
          },
    }
}
```


start=>
Epochs : 1 Train Accuracy : 30.9625 Train Loss 2.098196043968201
Epochs : 1 Validation Accuracy : 31.1256405248230 Validation Loss 28.96448224112056
Epochs : 2 Train Accuracy : 33.7375 Train Loss 1.963401982307434
Epochs : 2 Validation Accuracy : 32.21610805402701 Validation Loss 1.9299305828790816
Epochs : 3 Train Accuracy : 34.325 Train Loss 1.9051520924568177
Epochs : 3 Validation Accuracy : 32.31615807903952 Validation Loss 1.924699003734286
Epochs : 4 Train Accuracy : 36.8 Train Loss 1.856482506752014
Epochs : 4 Validation Accuracy : 33.066533266633314 Validation Loss 1.8835758046498374
Epochs : 5 Train Accuracy : 39.1375 Train Loss 1.8265055546760558
Epochs : 5 Validation Accuracy : 35.51775887943972 Validation Loss 1.850120463068523
Epochs : 6 Train Accuracy : 39.1375 Train Loss 1.777289710998535
Epochs : 6 Validation Accuracy : 34.71735867933967 Validation Loss 1.8382663745728751
Epochs : 7 Train Accuracy : 39.3625 Train Loss 1.751934485912323
Epochs : 7 Validation Accuracy : 36.46823411705853 Validation Loss 1.806052495562841
Epochs : 8 Train Accuracy : 40.625 Train Loss 1.7294755606651306
Epochs : 8 Validation Accuracy : 37.46873436718359 Validation Loss 1.8174887838817777
Epochs : 9 Train Accuracy : 40.8625 Train Loss 1.7075780301094055
Epochs : 9 Validation Accuracy : 37.618809404702354 Validation Loss 1.8041663890414767
Epochs : 10 Train Accuracy : 41.7125 Train Loss 1.6950566010475159
Epochs : 10 Validation Accuracy : 38.3191595797899 Validation Loss 1.7900837527381048
Done!!

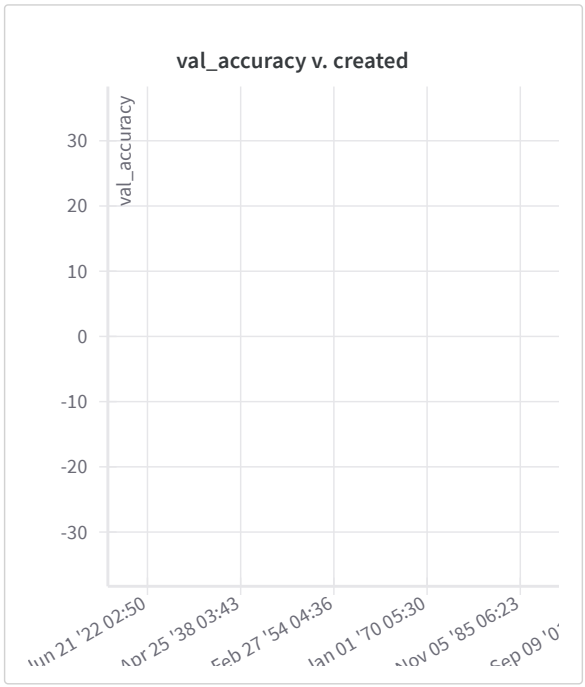
Run summary:

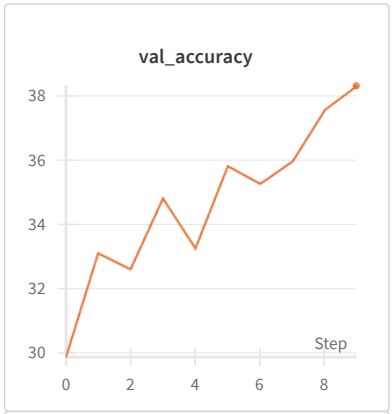
train_accuracy 41.7125

train_loss 1.6950566010475159

val_accuracy 38.3191595797899

val_error 1.7900837527381048



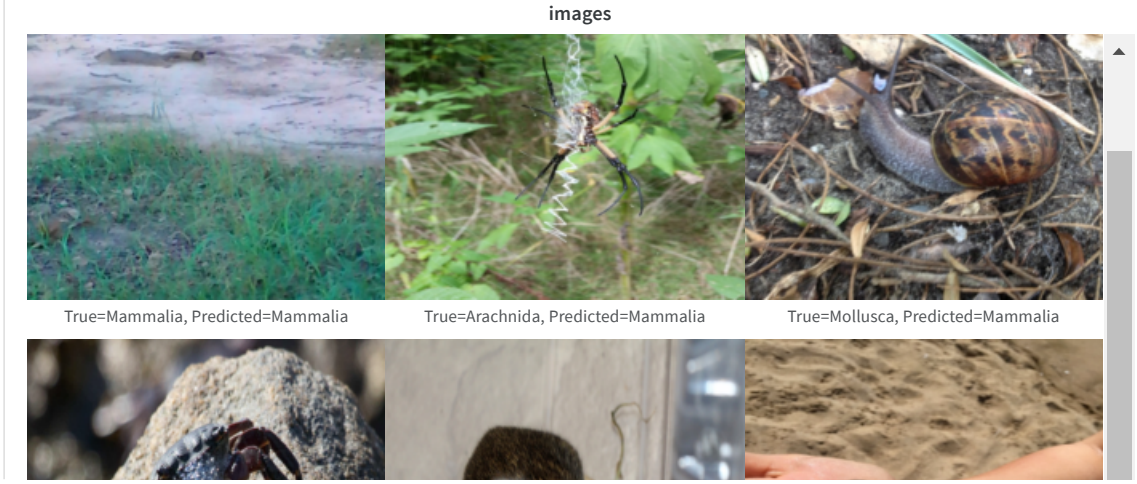


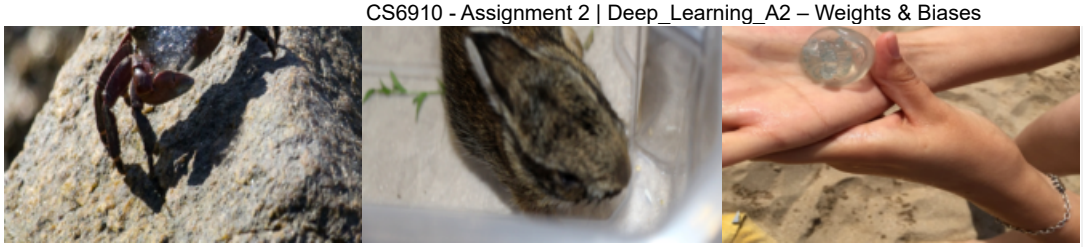
Import panel Add panel

- 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).

Images:

images with Actual and Predicted labels





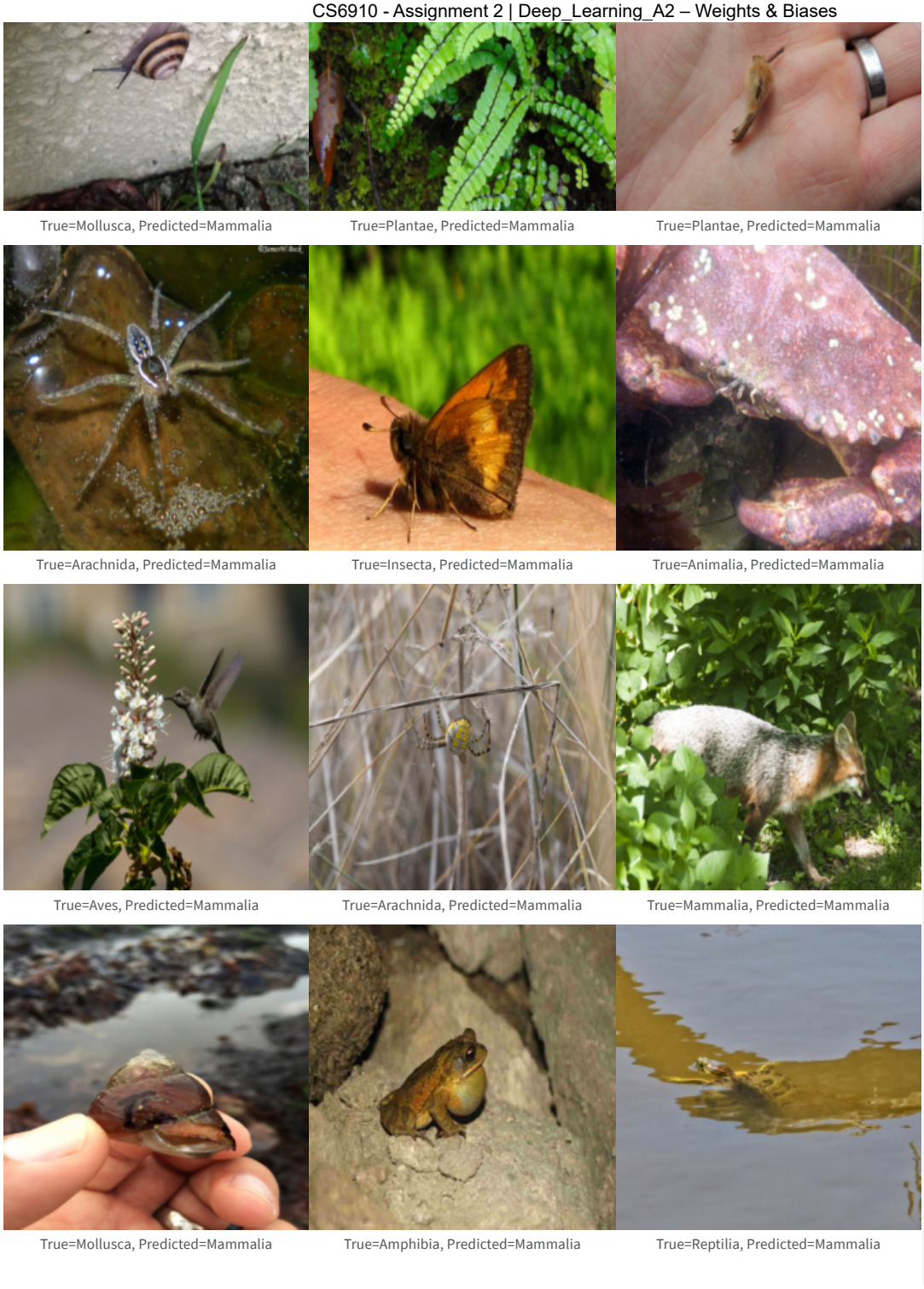
True=Animalia, Predicted=Mammalia | True=Mammalia, Predicted=Mammalia | True=Animalia, Predicted=Mammalia

True=Reptilia, Predicted=Mammalia | True=Reptilia, Predicted=Mammalia | True=Plantae, Predicted=Mammalia

True=Mammalia, Predicted=Mammalia | True=Arachnida, Predicted=Mammalia | True=Fungi, Predicted=Mammalia

True=Animalia, Predicted=Mammalia | True=Plantae, Predicted=Mammalia | True=Animalia, Predicted=Mammalia

True=Insecta, Predicted=Mammalia | True=Mammalia, Predicted=Mammalia | True=Insecta, Predicted=Mammalia



Import panel Add panel

- **(UNGRADED, OPTIONAL)** Visualize 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

- GitHub link for Part A

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 :

- The pytorch gives utility to transform the input data in its loading phase , below code will reshape the image and will convert it to tensor objects.

```
Transformation = transforms.Compose([
                                     transforms.Resize((256,256)),
                                     transforms.ToTensor(),
                                     ])
```

- 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 :

- We can solve this using following methods:

```
1. Adding new layer can solve this as  nn.linear(1000,10) after last FC layer.

2 . Do modification in last layer do extract the in_features of last FC layer and connect
it to 10 neurons .

                                Cnn_lnet = models.googlenet(pretrained=True)
```

```
                                num_of_features = Cnn_lnet.fc.in_features # Connect last
FC layer to 10 outputs
                                model.fc = nn.Linear(num_of_features, 10)
```

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 :

I have used **VGG (Visual Geometry Group)** , a standard deep Convolutional Neural Network (CNN) architecture with multiple layers. Here "deep" refers to the number of layers with **VGG-16 or VGG-19** consisting of **16 and 19 convolutional layers**

strategies i applied:

`1. Freeze Early Layers:`
I Freezes the initial convolutional layers and only fine-tuned the FC layers at the end.

`2. Freeze all Layers except last:`
I Freeze the entire VGG model except for the last layer, adjusting all convolutional layers .

`3 . Use Learning Rate Warmup:`
I gradually increased the learning rate during the initial epochs to help stabilize training.

`4. Use Data Augmentation:`
I applied data augmentation techniques (45 degree random rotations, flips) which artificially expands training
dataset and help prevent overfitting.

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 :

Code for part B

A. Training from scratch

```
start=> Training from scratch
Epochs : 1 Train Accuracy : 0.518625 Train Loss 0.015845
Epochs : 1 Validation Accuracy : 38.9765 Validation Loss 0.014589
Epochs : 2 Train Accuracy : 0.533500 Train Loss 0.008054
Epochs : 2 Validation Accuracy : 0.415678 Validation Loss 0.013698
Epochs : 3 Train Accuracy : 0.654750 Train Loss 0.006025
Epochs : 3 Validation Accuracy : 0.428972 Validation Loss 0.013458
Epochs : 4 Train Accuracy : 0.678375 Train Loss 0.005025
Epochs : 4 Validation Accuracy : 0.439015 Validation Loss  0.013320
Epochs : 5 Train Accuracy : 0.713250 Train Loss 0.004852
Epochs : 5 Validation Accuracy : 0.543059  Validation Loss 0.013004
Done!!
Total time for training : 269.859674415247 sec
Run summary:

train_accuracy 71.3250
train_loss 0.004852
val_accuracy  54.3059
val_error  0.013004
```

B. Freeze All Layers except last

```
start=> Freezing all the layers except last.
Epochs : 1 Train Accuracy : 0.5545108966342179 Train Loss 0.011488564360891655
Epochs : 2 Validation Accuracy : 0.5145108966342179 Validation Loss : 0.011488564360891655
Epochs : 2 Train Accuracy : 0.610021982994399 Train Loss 0.008874567184034307
Epochs : 2 Validation Accuracy : 0.5845 Validation Loss 0.004091669113188984
Epochs : 3 Train Accuracy : 0.68023598539472 Train Loss 0.008308491562948142
Epochs : 3 Validation Accuracy : 0.646 Validation Loss 0.004181718900054695
Epochs : 4 Train Accuracy : 0.705256988545138 Train Loss 0.0077442533940789175
Epochs : 4 Validation Accuracy : 0.6835 Validation Loss 0.00437046833336356
Epochs : 5 Train Accuracy : 0.763026598989532 Train Loss 0.0076004578472554825
Epochs : 5 Validation Accuracy : 0.735 Validation Loss 0.00932397773861885
Done!!
Total time for training : 261.415784961536 sec
Run summary:

train_accuracy 76.3026598989532
train_loss 0.76004578472554825
val_accuracy  73.5
val_error  0.932397773861885
```

C. Fine-tuning a large pre-trained model

```
start=> fine-tuning a large pre-trained model
Epochs : 1 Train Accuracy : 0.299625 Train Loss 0.2981
Epochs : 1 Validation Accuracy :0.300008 Validation Loss 0.289644
Epochs : 2 Train Accuracy : 0.307375 Train Loss 0.19634
Epochs : 2 Validation Accuracy : 0.322161 Validation Loss 0.19299
Epochs : 3 Train Accuracy : 0.31325 Train Loss 0.19051
```

```
Epochs : 3 Validation Accuracy : 0.323161 Validation Loss 0.19246
Epochs : 4 Train Accuracy : 0.348 Train Loss 0.18564
Epochs : 4 Validation Accuracy : 0.330665 Validation Loss  0.18835
Epochs : 5 Train Accuracy : 0.371375 Train Loss 0.18268
Epochs : 5 Validation Accuracy : 0.345177  Validation Loss 0.18501
Done!!
Total time for training : 263.415748596235 sec
Run summary:

train_accuracy  37.1375
train_loss      0.18268
val_accuracy    37.1375
val_error       0.18501
```

`Inferences from above runs`

1. Though the validation accuracy is good but it takes more time to `train from scratch` for the larger dataset.

2. It results good accuracy when training last layer and takes less time also , as only we have to update the last layers gradient .

3. Training only the last layer takes less time because we update only the gradients of the last layer.

4. `Training only last layer` and `freezing` rest results more efficient results as initial layer extract important features and train weights only for last layer also takes less time.

5. `Fine tuning` can results poor results as we have to learn the parameters of the initial layers of the networks .

Question 4 (10 Marks)

Paste a link to your GitHub code for Part B

- GitHub link for Part B

(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>/cs6910_assignment2/partC

Self Declaration

I, ANIKET KESHRI (Roll no: CS23M013), 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.