CS6910 - Assignment 2

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

Arun Muthukkumaran

Roll No: CH19D751

https://wandb.ai/arunm917/CS6910_Assignment_2/reports/CS6910-Assignment-2--Vmlldzo0MDAwMDAz

https://github.com/arunm917/CS6910-Assignment-2

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)

ANSWER:

The code has been written such that the number of layers, number of filters, filter size, activation function, number of neurons in the dense layers, etc., are all hyperparameters that can be varied.

Total number of computations:

Lets assume the size of the input image is H*W*D; then the total number of computations in the first layer is:

H*W*D*k*k*m

In the second layer, the output of the first layer will be convolved by the subsequent layer, so the number of computations is:

H*W*m*k*k*m (Assuming dimensions in subsequent layers are same)

The above number of computations is repeated in the next 3 layers. The number of computations between the last convolution layer and the dense layer containing 'n' neurons is given by:

H*W*m*n

In this case, the total number of computations can be approximated to:

H*W*D*k*k*m + 3*(H*W*m*k*k*m)+ (H*W*m*n) + (10*n) = HWm* (D*(k^2) + 3m*(k^2) + n) + 10n

Total number of parameters:

Parameters in the first layer: k*k*D*m + m

Parameters in subsequent convolution layer = 3*(k*k*m + m)

Parameters in fully connected layer = k*k*m*n + n

Parameters in the final layer = n*10 + 10

Total no. of parameters = (k^2)Dm + 3(k^2)m + (k^2)mn + 4m + 11n + 10

Question 2 (15 Marks)

You will now train your model using the iNaturalist dataset. The zip file contains a train and a test folder. Set aside 20% of the training data 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)

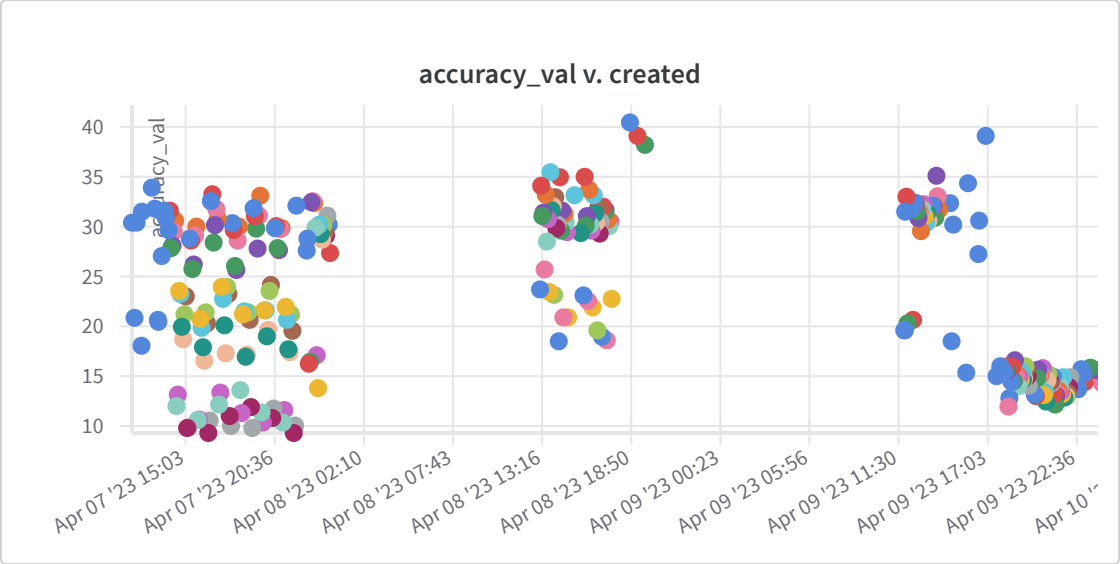
Based on your sweep please paste the following plots which are automatically generated by wandb:

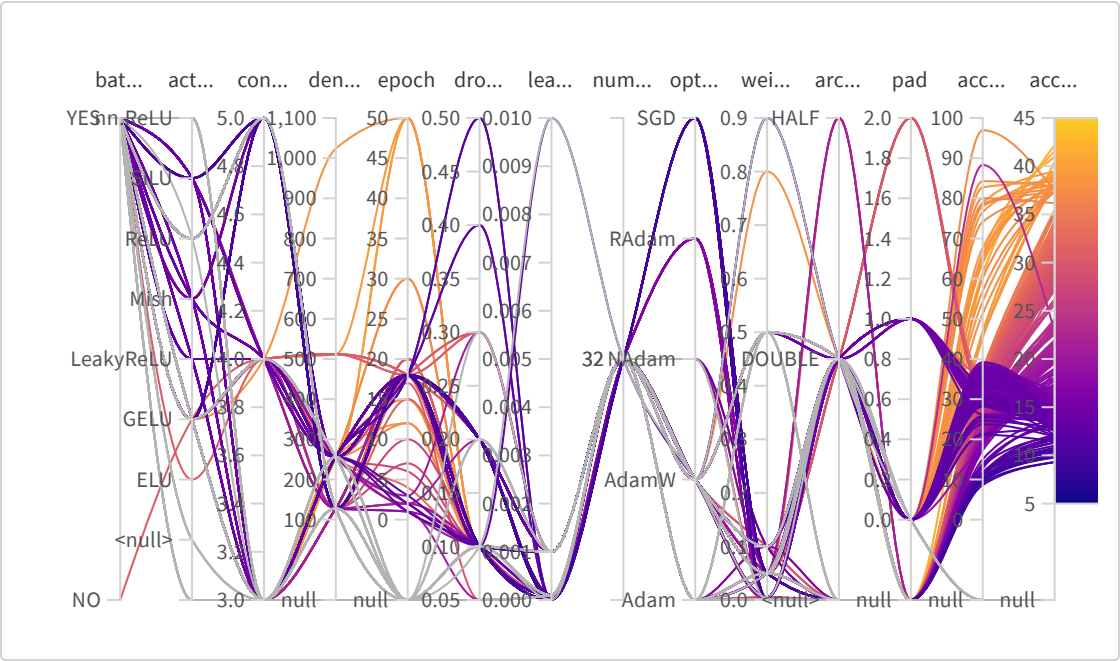
- accuracy v/s created plot (I would like to see the number of experiments you ran to get the best configuration).
- parallel co-ordinates plot
- correlation summary table (to see the correlation of each hyperparameter with the loss/accuracy)

Also, write down the hyperparameters and their values that you swept over. Smart strategies to reduce the number of runs while still

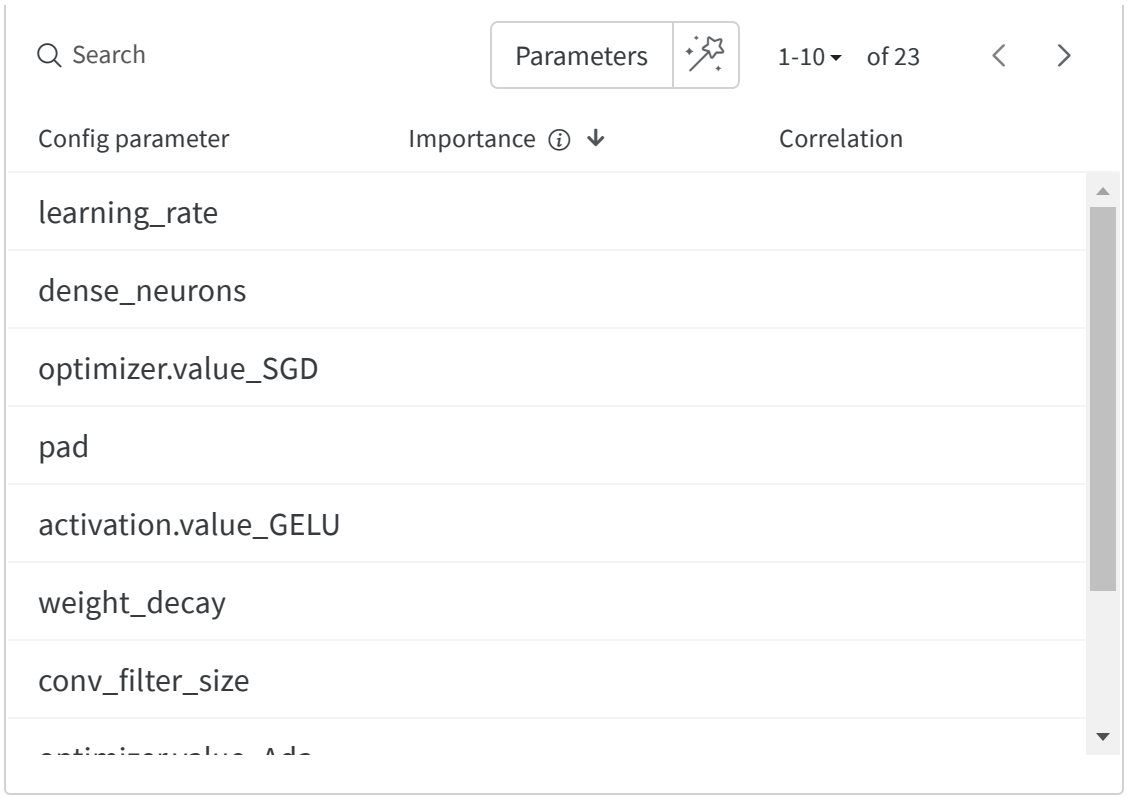
achieving a high accuracy would be appreciated. Write down any unique strategy that you tried.

ANSWER:





Parameter importance with respect to accuracy_val



The set of hyperparameters tuned are mentioned below:

- Epocs: [15, 18, 50] (For hyperparameter tuning the epochs was fixed at 18)
- Architecture: [DOUBLE, HALF, EQUAL]
- Batch norm: [YES, NO]
- Number of filters: [16, 32, 64] (in the case of EQUAL architecture, the mentioned values were tried for each layer. For DOUBLE,, the number of filters was kept at 8 for the first layer and the number of filters was doubled. For HALF, the number of filters was fixed as 128 for the first layer and the number of filters was halved in the subsequent layers)
- Convolution layer filter size: [3, 4, 5]
- Dropout: [0.1, 0.2, 0.3]
- Activation functions: [ReLU, Leaky ReLU, GELU, SiLU, Mish, ELU]
- Dense neurons: [128, 256]
- Padding: [0, 1, 2]
- learning rate: [1e-3, 1e-4, 1e-5]
- weight_decay: [0.005, 0.05, 0.1, 0.5, 0.8]

- Optimizer: [Adam, NAdam, RAdam, AdamW, SGD]
- Learning Rate Scheduler: CosineAnnealingLR and StepLR
- DataAugmentation: [YES, NO] (Mirroring , rotating and random crop were tried during data transformation. Implemented using transforms.RandomRotation(), transforms.HorizontalFlip(), transforms.RandomResizedcrop())

Sweep strategy:

1. Grid search was used for sweeping across multiple hyperparameters. Since applying grid search directly on the search space would lead to an infeasible number of runs, grid search was performed at multiple levels. The hyperparameters were optimized in multiple rounds.
2. Preliminary grid search was aimed at identifying the most effective optimizer, learning rate, activation function, network architecture, Convolution filter size, padding and number of dense neurons.
3. After optimizing the above parameters, regularization parameters such as dropout and weight decay were optimized.
4. Following this, a learning rate scheduler was chosen and a choice was made on whether to augment data or not.

Note: The number of runs in sweep is high because many test runs were not successful or terminated but still got logged into wandb.

Question 3 (15 Marks)

Based on the above plots write down some insightful observations.
For example,

- adding more filters in the initial layers is better
- Using bigger filters in initial layers and smaller filters in latter layers is better
- ... ...

(Note: I don't know if any of the above statements is true. I just wrote some random comments that came to my mind)

ANSWER:

- Doubling the filters in each layer gave better results when compared to halving the number of filters and keeping a constant number of filters across the convolution layers
- Batch normalization improved the training and prediction accuracy.
- Smaller filter size gave better results compared to applying filters of larger size
- Adding dropout layers and using weight decay improved the prediction accuracy and helped reduce overfitting
- From the correlation plot it can be observed that increasing dropout decreased the validation accuracy (negatively correlated)
- Using Cosine annealing learning rate scheduler improved prediction accuracy and helped avoid overfitting. This scheduler might also help the optimizer get out of local minima if the optimizer happens to get stuck in one.
- Data augmentation techniques such as mirroring, rotating and random cropping were tried. Though data augmentation reduced overfitting, it also had an adverse effect on validation accuracy. Hence while choosing the best model, data augmentation was not used.

Question 4 (5 Marks)

You will now apply your best model on the test data (You shouldn't have used test data so far. All the above experiments should have been done using train and validation data only).

- Use the best model from your sweep and report the accuracy on the test set.
- Provide a 10×3 grid containing sample images from the test data and predictions made by your best model (more marks for presenting this grid creatively).
- **(UNGRADED, OPTIONAL)** Visualise all the filters in the first layer of your best model for a random image from the test set. If there are 64 filters in the first layer plot them in an 8×8 grid.
- **(UNGRADED, OPTIONAL)** Apply guided back-propagation on any 10 neurons in the CONV5 layer and plot the images which excite this

neuron. The idea again is to discover interesting patterns which excite some neurons. You will draw a 10×1 grid below with one image for each of the 10 neurons.

ANSWER:

Best Model hyperparameters:

- Epocs: [15]
- Architecture: [DOUBLE] (starting with 16 filters in the first layer)
- Batch norm: [YES]
- Convolution layer filter size: [3]
- Dropout: [0.1]
- Activation functions: [GELU]
- Dense neurons: [256]
- Padding: [0]
- learning rate: [1e-3]
- weight_decay: [0.5]
- Optimizer: [AdamW]
- Scheduler: CosineAnnealingLR
- DataAugmentation: NO

Training accuracy: 61.47

Validation accuracy: 42.20

Test accuracy: 42.80

Question 5 (10 Marks)

Paste a link to your github code for Part A

https://github.com/arunm917/CS6910-Assignment-2/blob/main/cs6910_assignment_2_PartA.py

- 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.

Part B: Fine-tuning a pre-trained model

(Note: This question is only to check the implementation. The subsequent questions will talk about how exactly you will do the fine-tuning.)

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?

ANSWER:

1. The size of the images can be resized to the same size as those used to train the pretrained model.
2. The output layer of the pretrained model can be replaced by a custom layer that contains only 10 nodes.

Question 2 (5 Marks)

You will notice that `GoogLeNet`, `InceptionV3`, `ResNet50`, `VGG`, `EfficientNetV2`, `VisionTransformer` are very huge models as compared to the simple model that you implemented in Part A. Even fine-tuning on a small training data may be very expensive. What is a common trick used to keep the training tractable (you will have to read up a bit on this)? Try different variants of this trick and fine-tune the model using the iNaturalist dataset. For example, '___'ing all layers except the last layer, '___'ing upto k layers and '___'ing the rest. Read up on pre-training and fine-tuning to understand what exactly these terms mean.

Write down the at least 3 different strategies that you tried (simple bullet points would be fine).

ANSWER:

One common trick to keep the training tractable when dealing with large models is *transfer learning*. Transfer learning involves using a pre-trained model as a starting point for training on a new dataset or task instead of training the model from scratch. This allows us to leverage the pre-trained model's learned features and weights, significantly reducing the amount of training data and time required for the new task.

To fine-tune a pre-trained model on the iNaturalist dataset, we can use transfer learning and apply different variants of transfer

learning as mentioned below:

1. **Freezing all layers except the last layer:** In this approach, we freeze all the layers in the pre-trained model except for the last layer, which is replaced with a new layer for the iNaturalist dataset. This allows us to fine-tune the model on the new dataset while keeping the previously learned features intact. This is implemented by setting the parameter update property of all layers except the last layer to False.
2. **Freezing up to k layers and training the rest:** In this approach, we freeze the first k layers of the pre-trained model and fine-tune the rest of the layers on the iNaturalist dataset. This allows us to fine-tune the model on the new dataset while retaining some of the earlier learned features. This is implemented by setting the parameter update of the first k layers to False and setting the parameter update property of the remaining layers to True.
3. **Fine-tuning all layers:** In this approach, we fine-tune all the layers of the pre-trained model on the iNaturalist dataset. This approach may be more computationally expensive than the previous approaches, but it allows us to adapt the pre-trained weights to the new dataset more extensively. This is implemented by retraining the pretrained model using a small learning rate and updating all the model parameters.

Question 3 (10 Marks)

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

ANSWER:

Resnet50 was chosen, and the model could predict the images in iNaturalist dataset with high accuracy. Training was carried out by replacing the last layer of Resnet50 using a layer that outputs 10 classes instead of 1000.

After 15 epochs the model gave a test accuracy of 79.75%. Data augmentation was performed to make sure that the model did not overfit. Following this a fully connected layer was introduced with 512 neurons and GELU activation function. Learning rate was halved after every 5 epochs using StepLR scheduler. A weight decay of 0.2 was used. Adding an additional layer increased the test accuracy to around 81%.

- The pre-trainied model gave better accuracy compared to the model built in part A. The pretrained model has a large number of parameters compared to the model built in part A. This helps the model learn more features from images compared to simpler models such as the one in part A.
- The parameters of the pre-trained model have been optimized by training on the much larger ImageNet dataset, this has made the model learn more complex patterns and relationships.
- Learning from the wide range of classes present in the Imagenet dataset has helped the pre-trained model generalize better. Hence the model is able to give good results on the iNaturalist dataset as well.
- The complex architecture of the Resnet50 model (large number of layers with skip-connections) helps improve the model's performance.

Hence it can be concluded that for the given dataset, transfer learning is the better approach of the two. The combination of pretraining on large amounts of diverse data, transfer learning, and advanced architectures and make large pretrained models more effective and efficient than simpler models built from scratch. Extra layers can be added to the pre-trained model, and the model can be tuned for accomplishing specific tasks.

Question 4 (10 Marks)

Paste a link to your GitHub code for Part B

https://github.com/arunm917/CS6910-Assignment-2/blob/main/cs6910_assignment_2_PartB.py

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

(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).

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

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).

Self Declaration

I, Arun M. (Roll no: CH19D751), swear on my honor that I have written the code and the report by myself and have not copied it from the internet or other students.