SYDE 572 Final Project

Kritik Kaushal, k3kausha@uwaterloo.ca, 20854655

For the few-shot learning problem presented by the project, I chose to use **transfer learning** as the design methodology for the task. The training set provided was small, with only 5 training images per class. Given that, I decided to use a pretrained model, which I fine-tuned for this problem.

Given the constraints of the project, the biggest pretrained model I could pick was ResNet50. The model I picked had to be smaller than ResNet50. As such, I tried various models. The models I tried were:

- ResNet34
- ResNet18
- VGG11
- VGG16

After many iterations of trial and error, I found **ResNet18** had the best performance. This is likely because the other networks were overfitting, given the limited training images.

One big issue I was having was the mismatch of class labels when loading the training data. Pytorch was matching class "10" with ID 2. I suspect this is because Python is an interpreted language, and the source code is converted to bytecode which is in binary format. That explains why "10" (which is binary 2) was being mapped to ID 2. See Figure 1 below for the class to ID map of the training data.

```
Python Version:  3.11.5 (tags/v3.11.5:cce6ba9, Aug 24 2023, 14:38:34) [MSC v.1936 64 bit (AMD64)]
Pytorch Version:  2.1.0+cu121
CUDA is available!
CUDA Device Name: NVIDIA GeForce GTX 1650 with Max-Q Design
{'0': 0, '1': 1, '10': 2, '11': 3, '12': 4, '13': 5, '14': 6, '15': 7, '16': 8, '17': 9, '18': 10, '19': 11, '2': 12, '20': 13, '21': 14, '3': 15, '4': 16,
 '5': 17, '6': 18, '7': 19, '8': 20, '9': 21}
<class 'int'>
PS C:\Users\kriti\Desktop\4A\syde 572\project>
```

Figure 1: Class to ID map for training data loader

Clearly, this would affect the model performance, which is why my first couple of submissions on Kaggle have such low accuracy scores.

To combat this, I first tried to define a custom dictionary to convert the folder names to ints. However, this approach did not work. Instead, I used the same class_to_idx map for the test data loader as the train data loader. This ensures consistency across the two. Then, I also had to define a dictionary to convert the predicted IDs back to the given class labels. This was so that predicted submissions made sense.

After I fixed the class-ID mapping issue, the accuracy of my model instantly shot up to above 90%.

Below, I will explain the process of determining the best hyperparameters, data preprocessing steps I took, and the fully connected layer architecture.

For data preprocessing, I chose to apply the following transformations to the training and test set:

- **Resizing:** Both the train and test images were resized from 1020*530 to 360*360. While ResNet18 does not necessarily require square image inputs, I found from a data science forum post that smaller, square images are better to feed the network because it reduces the amount of noise and variance the network has to deal with, leading to better accuracy [1]. I found 360*360 images to give the best trade-off in terms of accuracy and image size.
- **Normalization:** Both the train and test images had the same normalization transform applied to them. This is important because it ensured that all pixel values were on the same scale. The specific values I chose for the mean and standard deviation were: mean = [0.5, 0.5, 0.5], std = [0.5, 0.5, 0.5]. My thought process behind choosing these specific values was that I wanted each dimension (R,G, and B) to be on a scale from 0 to 1. This is why I selected a mean and standard deviation of 0.5. This greatly helped with stability and accuracy of the model.

Hyperparameter tuning:

For fine-tuning the ResNet18 model I played around with the following hyperparameters:

- **Batch-Size:** This one is not as important as others, but I ended up using a batch size of 8. I tried using bigger batch sizes, but for some reason my GPU couldn't handle anything bigger than 32. As such, I selected a batch size of 8.
- **Loss Function:** Based on Assignment 4, I used the 'Cross Entropy Loss' loss function.
- **Optimizer:** Based on Assignment 4, I used the 'Adam' optimizer.
- **Epochs:** From trial and error, I found 35 epochs was giving me accuracies of between 96%-99%. I tried 20 epochs as well, and I was getting around 97% accuracy. I ended up choosing 35 epochs.
- **Learning Rate:** To help determine an appropriate learning rate, I printed the loss and training accuracy for each epoch. I started with a learning rate of 0.001 and gradually decreased it until I was satisfied with the loss. I wanted to get the loss as close to 0 as possible. I found that this happened when the learning rate was 0.0000875, combined with training the model for 35 epochs.

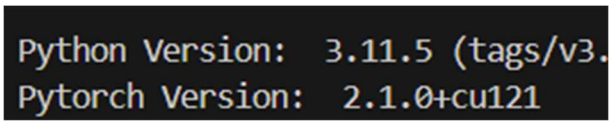
Fully Connected Layer Architecture:

For the fully connected layer, I tried many strategies to attempt to avoid overfitting such as including a Dropout layer, as mentioned in [2] and [3]. Interestingly however, I found that just a single 'Linear' layer corresponding to the number of class labels gave the best results. This is likely because ResNet18 is already very robust (since it was trained on ImageNet), so adding a Dropout layer was not necessarily helpful.

Overall, the ResNet18 model with fine-tuning gave great results. I was consistently getting high accuracies. Below are instructions on how to run the code.

Running the code:

The script is called "project.py". These are the dependencies for the scripts:



I used Python V3.11.5 and Pytorch V2.1.0 with support for CUDA, to allow me to train the model on my GPU.

The script will let you know if a CUDA device is available. In my case:

```
CUDA is available!
CUDA Device Name: NVIDIA GeForce GTX 1650
```

Ensure that the “5_shot” folder is in the same folder as “project.py” to ensure that the paths used for loading the train and test data are correct. The folder structure in my submission is accurate.

After running the script, a CSV file called “k3kausha_predictions.csv” will be generated. This is a sorted list of image IDs and predicted labels (categories).

References

[1] N. Slater, "Reason for square images in deep learning," [Online]. Available: https://datascience.stackexchange.com/questions/16601/reason-for-square-images-in-deep-learning#:~:text=Most%20of%20the%20advanced%20deep,size%20of%20224x224..

[2] J. Brownlee, "How to Avoid Overfitting in Deep Learning Neural Networks," [Online]. Available: https://machinelearningmastery.com/introduction-to-regularization-to-reduce-overfitting-and-improve-generalization-error/.

[3] L. Zhang, "Preventing Overfitting," [Online]. Available: https://www.cs.toronto.edu/~lczhang/360/lec/w05/overfit.html.