Drill Bit Classificatio

Model Configuration:

- Most of the parameters that configure a particular training trail are compiled in a configuration yaml file. I tried different combinations of some of the hyperparameters as part of model tuning. You can find different versions of configurations that I tried in ../Neocis_Assessment/training/train_configs folder
- These model parameters are passed into different modules in training scripts as and when necessary

Creating Dataset Objects and Dataloaders:

Script Location: …/Neocis_Assessment/training/training_scripts/data.py

- I created a custom dataset object using pytorch Dataset module
- This Dataset object takes below arguments:
    - Paths: A dictionary that maps image id to image_path
    - Image_ids: A list of image ids to consider for data generation
    - Labels: A dictionary that maps image id to ground truth label
    - Class_list: Exhaustive list of all classes being considered in the classification
    - Crop: Cropping parameters to be used to crop the image
    - Transform: A boolean flag whether to apply transforms or not
- This Dataset object reads in an image using the image_id and image_path, transforms the image, creates the numerical label equivalent for this image and returns this image, label combination to the pytorch dataloader
- In particular, I used ColorJitter, GaussianBlur and RandomAdjustSharpness transforms from torchvision transforms. These transforms help in multiplying the input data as well as makes the prediction model robust to input images
- In some of the trials, I also tried random horizontal flip and random vertical flip transforms as these transforms will not alter the drill bit size with respect to the image frame of reference
- I also cropped the input images to include only the region of interest (a rectangle enclosing the drill bit) and to exclude unnecessary background from input images

Creating model template:

Script Location: …/Neocis_Assessment/training/training/training_scripts/model.py

- I created a standalone script that just consists of pytorch model template
- This script consists code for the pytorch model object than would be imported and used in train script
- This script is created such that the base model architecture can be flexible and can be mentioned in the configuration yaml file
- I modified the classifier layer of the base model according to the requirements of our classification problem
- I tried altering the classifier head architecture as part of model tuning. More about this in model training section

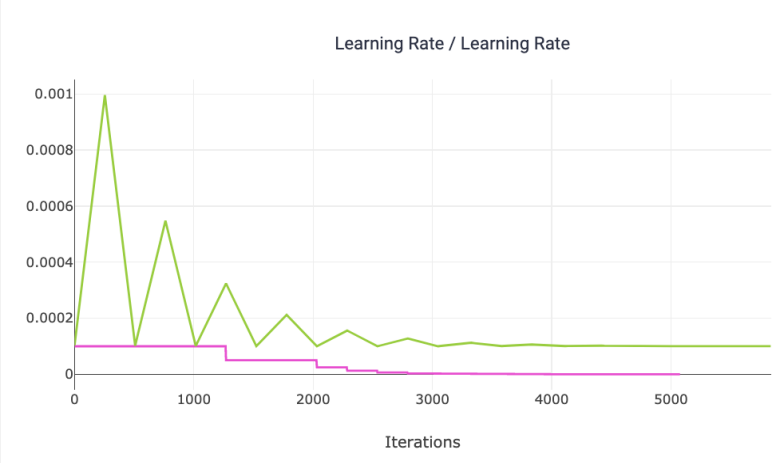
Model Training and Validation:

Script Location: …/Neocis_Assessment/training/training_scripts/train.py
Script Location: …/Neocis_Assessment/training/run_scripts/train_run.py

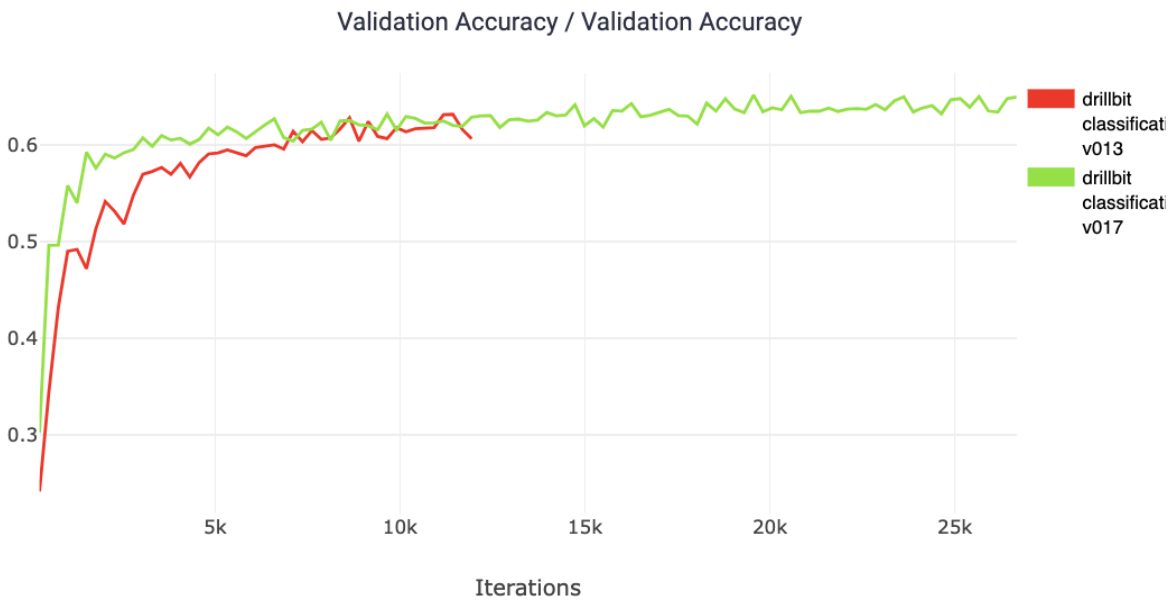
- The trian.py script consists of modelTrainer object that combines earlier modules and compile the process of training
- Firstly, configuration of the particular experiment is read using yaml loader and params variable is created which stores all crucial parameters used
- The data generator function in this object creates two pytorch data generator objects one each for training and validation. This function uses another helper function input_to_generators that takes path to the image meta data csv file, splits data randomly into train and validation and returns the splits along with dictionaries with image paths and image labels
- The train_model function runs a for loop for epochs times where in each loop, forward pass and backward pass happen using the batch of the data generated by training data generator. For every 25 iterations, it will also log the train loss and train error to the clearml logger
- After completion of every epoch, this will also computer validation loss, validation error, log them to clearml logger and also saves the model weights checkpoint
- I also included a small script for early stopping though this was not very useful in my case. I implemented this in utils.py and imported the object into the trian script

Hyperparameter tuning:

- I believe that learning rate is one of the most important hyperparamter that needs tuning in deep learning. So I will explain about the learning rate first
- I used cyclic lr scheduler and exponential lr scheduler from pytorch to adjust learning rate while training.
- Exponential scheduler reduces LR by a certain factor after certain number of epochs (which can be configured) where as cyclic lr bumps the learning rate up and down cyclically from base_lr till max_lr and also reduces the max_lr as training progresses
- Below is a plot depicting exponential lr (pink) and cyclic lr (green) in action



- Another hyperparameter was the base model to use. I tried mobilenet_v2 and efficientnet_v2 from torchvision models. Though efficient_v2 gave a little better results than mobilenet_v2, I chose the later for my final model. This is because mobilenet_v2 has around 3.5M parameters where as efficientnet_v2 as almost 21.5M parameters. Clearly mobilenet is a good choice though it needs some compromise in terms of model performance a little
- I also tried to change the classifier head of the mobilenet_v2 model by increasing the linear layers but could not see any improvement in performance. This could be because of the inductive bias of the model and the quality of the data
- Overall, I tried 17 different model configurations and found two configurations to be promising. Below are the validation accuracy plots of these two configurations



- Version 13 has horizontal and vertical flip augmentations where as 17 does not do any flips to input images. Below are the links to clearml experiment pages. One may need to signup on clearml to view these experiment results
- Link to clearml Version 013 experiment: https://app.clear.ml/projects/d01555fa38ba4d0f9fb86cc569f67324/experiments/c4ad762e1b4443b8abd57630eb8015f9/output/execution
- Link to clearml Version 017 experiment: https://app.clear.ml/projects/d01555fa38ba4d0f9fb86cc569f67324/experiments/851566d2691948249473fcbafd1cdfb9/output/execution

Prediction and Inference:

Script Location: …/Neocis_Assessment/inference/inference.py

I selected 4 different checkpoints from version 13 and four other from version 17 and combined them using functorch package to create a ensemble of eight models. While I tested this ensemble model on randomly selected images, it was performing with around 80% accuracy.

Further Improvements:

- Trying out more pretrained models, possibly exploring the usage of ViTs for classification
- Creating attention maps using thresholding based image edge detection and appending them into the model training along with raw image
- Writing more modular and dynamic code by having each and every possible parameter in configuration file