Intro To AI - Assignment 2

Deep Learning Model Training

Version 0.1.0

By: Brandon Toews
June 2023

Table of Contents

Section 1 – Project Overview

1.1 Purpose of Document

The purpose of this document is to detail the building of deep learning models using a convolutional neural network architecture. The different techniques, models and methods used to improve performance will be discussed.

Section 2 – Dataset

2.1 Bee vs Wasp

For this project I chose the Bee vs Wasp dataset found on Kaggle at https://www.kaggle.com/datasets/jerzydziewierz/bee-vs-wasp. I imported the dataset and created a new folder called images that I then put subfolders bee1, bee2, wasp1, wasp2, other_insect and other_noinsect into. The data loader in my custom train_models function then creates classes based on the folder structure and feeds that to the model. The data set itself isn't the cleanest as it seems that some images have not been placed in the correct folder which will sometimes give the model wrong information. No doubt this will affect the accuracy that can be attained with this dataset.

Section 3 – Experimenting

3.1 Trial and Error

To begin with I created a custom function named train_models that I could use to conduct my tests a little faster. With a trial and error approach, I began manually trying different learning rates, model types and image sizes, along with training models with unfrozen weights. Eventually I thought I should start trying to automate some of these tuning methods and, by doing so, hopefully optimize the outcomes.

3.2 Automating Hyperparameter Tuning

In research I found a Python library called Optuna that could be used to automate hyperparameter tuning. Optuna does this by created a “study” that runs a user specified amount of trials and uses an objective function to suggest user specified parameters to optimize for a certain metric. So in this case, I created a custom objective function named tune_hyperparameters that takes in learning rate, batch size, and weight decay parameters and returns the error rate of the model trained with those parameters. The Optuna optimize function then suggests hyperparameters that should start lowering the error rate of successive trials. I then wrote another custom function called optimization_study that ran the Optuna study using the tune_hyperparameters function. The optimization_study functions also selects the trial that did the best and proceeds to unfreeze all of the weights and train the model again with the best found hyperparameters. Some of my initial tests with this automated hyperparameter tuning proved promising as I was able to get the error rate lower than I had previously gotten it.

3.3 Automate Testing Different Models

As I started to achieve some good results with my automations I decided to go even further. I wrote another custom function called try_models that loops through a list of different models, runs an Optuna study on it and saves the model state from the best trial from that particular study on that particular model. Once the try_model function has finished looping through the list of models it selects the model that achieved the lowest error rate, creates a learner from that model and loads the model state of the best trial from that model. It then proceeds to unfreeze all of the weights and train the model again with hyperparameters from that particular model's best trail. After training is complete the function freezes the weights again, displays the results, and returns the model. I found some success using this new function as long as I kept the trial size relatively low as when I increased the trial size it exponentially increases compute time and quickly reaches the limits of free tier kernels.

3.4 Data Augmentation

I also briefly experimented with some data augmentation, namely randomly cropping to a 224x224 image size and introducing a random horizontal flip to the images. Tests with this didn't seem to yield any improved results, in fact it seems it may have adversely affected model performance in training. I theorize that this didn't have much effect because the dataset already possesses a great deal of randomness so injecting more isn't advantageous.

Section 4 – Kernels

4.1 Usage Limits

Very early on it was clear that usage limits of free tier kernels would significantly limit the ability to experiment, test and iterate. For this reason, the approach was taken to use more than one kernel so that when one reached its limit the other could be used to continue with the project. Google Colab and Kaggle were both used to complete this project and in the following two items(4.2 & 4.3) in this section I detail what each kernel was primarily used for. A notebook from each kernel is provided in this project submission, with part 1 and part 3 being included in the Google Colab notebook and part 2 being included in the Kaggle notebook.

4.2 Google Colab

I started my initial experimentation in Google Colab that is why it starts with the heading Part 1. Part way through my refinement of my custom automation functions I reached my limit with Google Colab so Part 2 of my code is found in the Kaggle notebook. The final part of my testing and code can be found under part 3 of the Google Colab notebook. In Part 3 I decided to purchase some Pay-As-You_Go compute so that I could continue the rest of my project without further delays.

4.3 Kaggle

The Kaggle notebook starts with the heading of Part 2 as it is the point where I switched from Google Colab. The Kaggle notebook only include one part and it is where most of my refinements on my custom function can be found. I was able to make some fairly large tests at the end of the Kaggle notebook but then reached my limit. At this point I switched back to finish things off in my Google Colab notebook under Part 3.

Section 5 – Performance

To improve performance SqueezeNet, EfficientNet, Resnet and VGG models were tested along with various batch sizes, learning rates and weight decays. Two (2) different image sizes were tested: (1) 224x224 and (2) 896x896. The parameters that yielded the worse and best results are detailed below in items 5.1 and 5.2.

5.1 Worst Performance

I wasn't able to test VGG16 to long before I ran into limit restrictions on the kernel but it wasn't performing all that well from what was seen. Further investigation would be required to confirm that VGG16 is not a good model for this dataset. SqueezeNet models did not perform as well as the other models which is not surprising giving the size and architecture of SqueezeNet models. The 896x896 image size did not seem to yield better results and neither did batch sizes 16 and 64. Learning rate range 1e-5 - 1e-1 did not yield good results as well as weight decay range 1e-5 – 1e-3.

5.2 Best Performance

After study some of the tests I started to isolate that a batch size of 32 did consistently well. Along with training only with a 32 batch size I narrowed the learning rate range to 1e-3 – 1e-2 and the weight decay range to 1e-5 -1e-4 as these ranges seems to provide the best results. In the end of all my testing the best performance I achieved was from a Resnet32 model trained with a 224 image size, 32 batch size, a learning rate of 3.102551277095900e-3 and a weight decay of 7.49113519525403e-05. This yielded a model with a training loss of 0.022758, valid loss of 0.065226, and error rate of 0.015762. These results show that the model is slightly overfitted but performing quite well.