3330 Assignment 2 report

Sol Scobie c3302821 and Matan Melamed cxxxxxxx

**Abstract.** A report on the computer vision models created for assignment 2

1. Introduction
   1. Model setup

Training data was divided into subfolders by identifier numbers in each image's name, with asparagus being 00 and cheese 39. For each category of food, if the training data contained more than one image, a random image was removed and placed into a separate validation set folder. Images were then randomly cropped, resized to 224 by 224, randomly flipped and randomly rotated by up to 5 degrees. The fact that the training dataset did not include multiples of every type of item has obviously reduced the accuracy of the testing and validation.

During training models are changed to evaluation mode and tested against the validation set with the best performance on the validation set being saved as a checkpoint for the model. Once training was completed the best performing version of the model was saved and of these saved models the best performing again were kept in the saved models included in the Appendix.

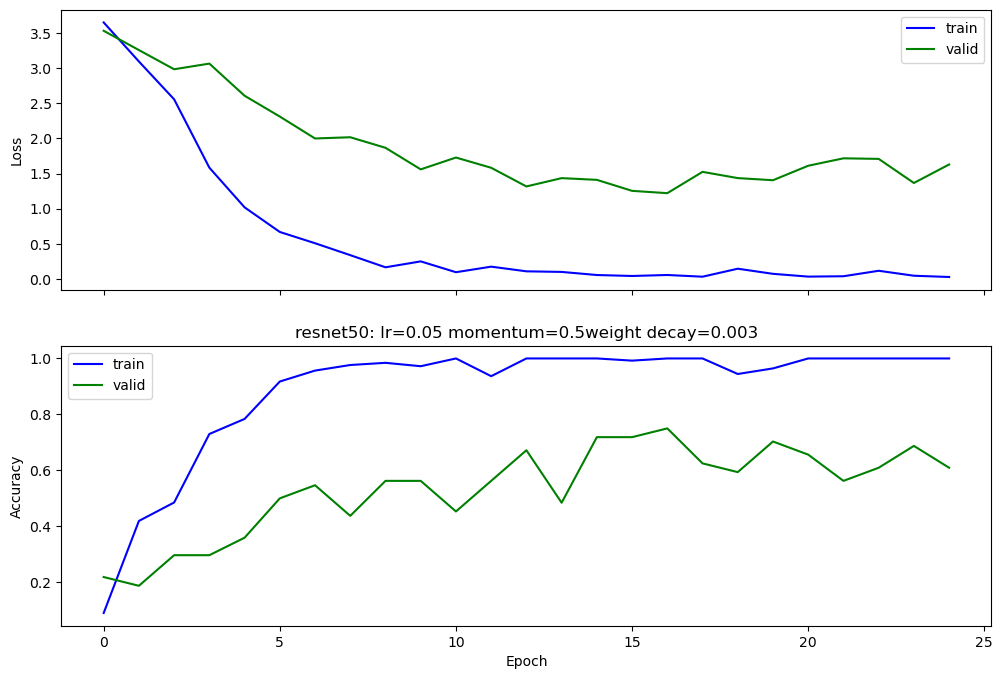
Resnet was the worse performing model of the two types implemented, with Squeezenet managing 100% accuracy on the validation set while Resnet never came close to that accuracy.

1. Resnet
   1. Resnet50

Resnet was chosen arbitrarily as it was one of the listed models on the assignment specification. The model was initialised with a learning rate of 0.1, momentum of 0.5 and weight decay of 0.003. These numbers were based on lab material and the classification suggestions on the Torchvision GitHub [1]. Graphs of training data for Resnet50 and 18 are included in the Appendix.

Most tests for these models were conducted over 25 epochs, taking approximately 1 minute 20 seconds, this was consistently the time the models took to train. Each of the learning values were tweaked ranging between 0.005 and 0.2 for learning rate, 0.5 and 0.9 for momentum and 0.0001 and 0.005 for weight decay.

The consistently best performing model was found with a learning rate of 0.05, momentum of 0.5 and weight decay of 0.003, achieving a peak validation accuracy of 73.44%, see Fig 1 for training results. After saving this model, the same setup was tested again for 90 epochs with no noticeable improvement and yielded the same peak accuracy.

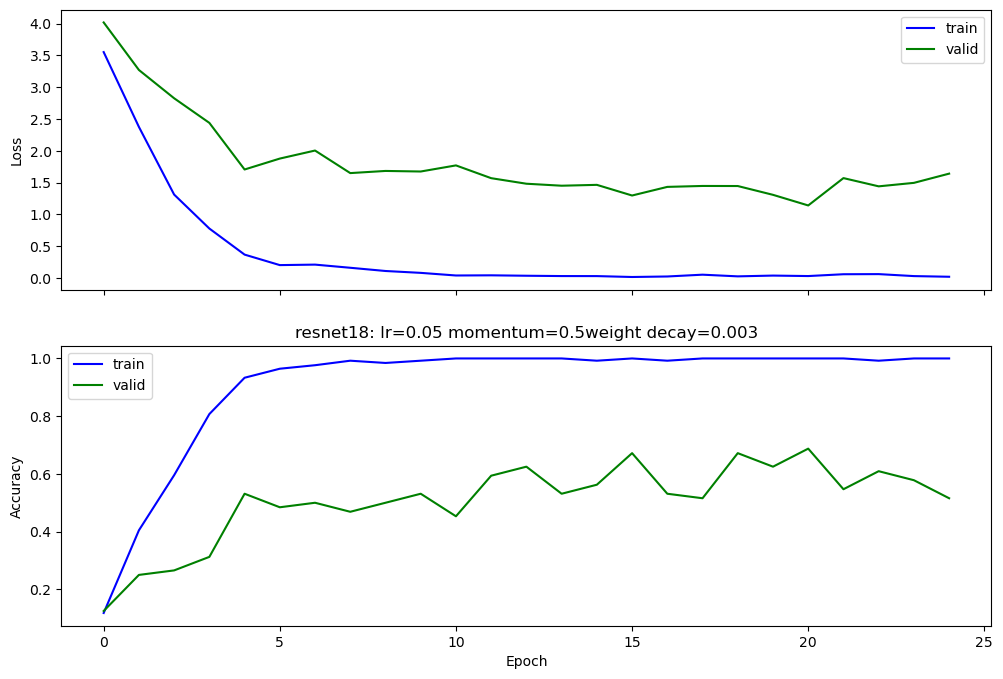


**Fig. 1.** Best performing Resnet50 model, trained in 1 minute and 17 seconds

* 1. Resnet18

Resnet18 performed worse nearly across the board. However, one model when training managed 75% accuracy on the validation set, outperforming resnet50’s peak. This model has been saved and is the example model included in the appendix. Subsequent trainings with the same parameters did not yield this peak of 75%, however, results in the 60’s were common. Resnet18 was not meaningfully faster or slower than resnet50 on average.

Values between 0.01 and 0.05 for learn rate, 0.3 and 0.7 for momentum and 0.003 and 0.005 for weight decay yielded no noticeable improvement over the best performing parameters from resenet50. The best results of training with that same values were included in Figure 2.

**Fig. 1.** Best performing Resnet18 model, trained in 1 minute and 9 seconds

* 1. Results

This implementation of Resnet will win no awards for its accuracy. This model with ~70% accuracy for the suggested use case will simply annoy or bemuse a user with how frequently it is wrong. That said, with no requirement for online connection and fast classification on a pretrained model it would be possible to make use of the system.

1. Squeezenet
   1. Subtitle

SqueezeNet stood out as an option to our team due to the relatively few resources it required to produce high validation accuracy. SqueezeNet has been specifically designed to work on small electronics with limited computation, which had to be considered as the team was designing models for a smart fridge. Changes to hyperparameters and their effect on performance will be discussed, and afterwards the overall predicted efficiency of the network on smaller devices.

The tests mentioned in the report refer to the test graphs produced by the models. These are stored in the “SqueezeNetGraphs” folder and each hyperparameter test set is listed under its own appropriately named folder. The tests consist of 5 instances of the same base model being trained on data with all the same hyperparameters, and then all graphed onto the same plot. With 5 tests it’s easier to rule out outliers and see a more general trend in the data. Each hyperparameter has multiple tests with different values, while the remaining hyperparameters stay unchanged throughout a specific hyperparameters test set for consistency.

While modifying multiple hyperparameters may allow them to ‘synergize’ (e.g. lr = 0.0001 works well with wd = 0.002 but lr = 0.001 works better with wd = 0.0001), the results still show a general trend in the effect that individual hyperparameters have on the performance of the model.

* 1. Learning Rate

The optimum learning rate was found to be around 0.0005, as the rate was apparently quite sensitive to modification. When the learning rate was pushed to 0.01, the model struggled to attain even 10% accuracy over 15 epochs – this is compared to lr = 0.0005 where 100% accuracy was typically attained within 11-12 epochs (Test1 and Test3).

This is likely due to the gradient decent algorithm ‘skipping’ over any minima when the learning rate was too high. At lr = 0.001, performance was similar to lr = 0.0005: the model managed to achieve higher accuracies slightly faster but was ultimately more unstable and still struggled to find the optimum solution when approaching the minima (Test2). With more image classes, it’s likely the model would have struggled to achieve high accuracy with this rate. At lr = 0.0001, the model made steady progress and obtained an optimum solution with more epochs but struggled to reach 100% within the 15 epoch limit (Test4) and was deemed unnecessarily slow by comparison.

* 1. Momentum

Changes in momentum seemed to cause significant changes in a model’s performance. With m = 0.5, all test models seemed to follow the same improvement in accuracy over the epoch training period, with slight divergence (Test1). When momentum was increased to m = 0.9, the models not only seemed to tighten the previous divergence, but also reached peak accuracy quicker, hitting near 100% accuracy at around epoch 11 (Test2). When momentum was further increased to m = 0.99, the model performed extremely poorly, even more so than the m = 0.5 models. All 5 tests on m = 0.99 achieved drastically different results and the accuracy of even the best test seemed to fall short of the overall accuracy of the m = 0.9 models (Test3). The optimum momentum was set at m = 0.9.

* 1. Weight Decay

The model didn’t seem to be sensitive to weight decay. The differences between wd = 0.00001 (Test2), wd = 0.01 (Test6) and wd = 0 (Test1) were minimal. At higher weight decay the model started to show signs of inconsistency between models in the same test, but ultimately, I believe this can be chalked up to randomization more than the effect of weight decay. The tightest grouping was still found to be around wd = 0.001(Test 4) so it was chosen for optimum (even though that isn’t a performance metric), but given the similarity in performance it the choice for optimum feels arbitrary. The minimal effect of weight decay may be a product of the fewer training epochs. Accuracy is usually topped out on epoch 10 with the current models so further epochs become unnecessary, and weight decay doesn’t get to make an impact.

* 1. Optimiser

SGD (Test1) achieved 100% accuracy within 15 epochs consistently with the given hyperparameters. Adam (Test2) managed to achieve 100% in one of its 5 test runs, but every one of its tests still performed far more poorly than SGD. RMSprop (Test3) was also attempted but struggled to get past 10% accuracy within the 15-epoch limit. I’m assuming this had more to do with improper implementation of this optimizer function, but with SGD performing satisfactorily I found it unnecessary to fix.

* 1. Overall Efficiency

As the model must be deployed and used in the context of a smart fridge, its resource efficiency is a significant factor. SqueezeNet1.0 (the model used in the tests) has 125 million parameters, though through the use of pruing the model can be reduced to 420,000 parameters. With data stored in 32-bit floats, the whole pretrained model only takes up 4.8MB of storage space. An additional 2MB allocated for overhead allows the entire program to run within 6.8MB of RAM.

This can easily be run on a hobby microprocessor set-up like a Rasberry Pi. Arduinos are too weak to run the program, but a mirco-processor between the capacity of these two platforms should be able to achieve a maximum efficiency to cost ratio. This sort of processor should be standard for a smart fridge.

Internet access will not be required, but the weights are pretrained, so the model did need to be downloaded initially. At a factory producing custom chips for the purpose of food identification, the software would be embedded in the chips during manufacture – the customer would not need internet to run this feature of their fridge.

1. Conclusion

As mentioned in the introduction, SqueezeNet achieved the higher accuracy of the two models. SqueezeNet was also specifically designed to be implemented by devices with limited resources for computation, which made it the ideal candidate for the smart fridge image identification network.

References

1. Torchvision Classification Page, <https://github.com/pytorch/vision/tree/main/references/classification>
2. SQUEEZENET: ALEXNET-LEVEL ACCURACY WITH 50X FEWER PARAMETERS AND <0.5MB MODEL SIZE (2017) - DeepScale & UC Berkeley, Stanford University

Appendix

Graphs of resnet training results are included in the folder resnetGraphs

Code is included as part of the submission zip file.

ClassifyData.py classifies training data, seperation for validation must be done manually

TestModels.py prepares a folder of images for classification then allows saved model selection and saves results to a csv

Squeezenet.py and resnetTrainer.ipynb train the specified model