Finding a Well-Tuned Model for Kaggle's *Petals to the Metal* Competition

Wren McQueary
Department of Computer Science
George Mason University
Email: wmcquear@gmu.edu

***Abstract*—Kaggle's *Petals to the Metal* competition is a multi-class image classification challenge involving images of 104 types of flowers. Because we hadn't designed our own architectures in any assignments for this course, I found a well-tuned architecture and set of training hyperparameters for this Kaggle challenge. I found these attributes using a modified grid search followed by a more nuanced tuning process informed by the learning curves produced during training. I also compared my original model's performance to the performance of AlexNet, ResNet-50, and VGG16, all pretrained on ImageNet. My model's validation accuracy was 45.27%, compared with AlexNet's validation accuracy of 29.17%, ResNet-50's validation accuracy of 70.34%, and VGG16's estimated validation accuracy of 83.56%.**

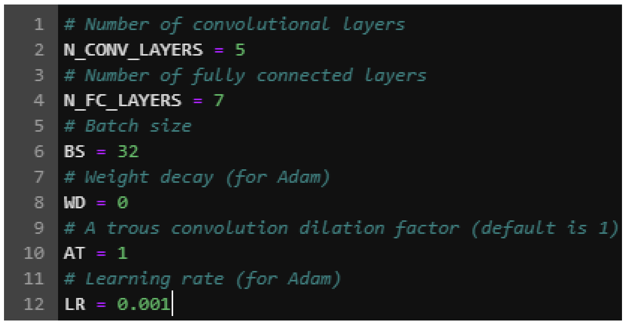
I. Introduction

Multiclass image classification is a problem domain in which an image is identified as belonging to one of three or more classes [1]. This domain has many applications, such as using an X-ray image to discern bone fracture types, using microscopy images to identify metallic grain structures, and discerning detected traffic signs from one another in automated driving. Convolutional neural networks (CNNs) are a typical approach to this problem domain, and state-of-the-art approaches, such as FixEfficientNet, continue to grow in accuracy and efficiency, and are often designed for easy combination with other approaches, such as through finetuning or transfer learning [2].

Petals to the Metal is an introductory Kaggle competition aimed at newcomers to Kaggle's website [3]. In this image classification problem, each image is of a different flower, and each label is the type of flower. This challenge contains a total of 104 flower labels, and the dataset consists of about 66,000 samples. My project goal was to build a CNN to enter into this competition.

Our previous homework assignments have included some limited tuning of hyperparameters for CNNs. However, there has been a lack of choosing the architecture itself. In this project, I closed that gap by designing an architecture myself.

To make it easy to test different architectures, I modified an existing simple `nn.Module` class so that it automatically produced a CNN with any number of convolutional and fully-connected layers, within a range. I then searched for a best-possible combination of architecture attributes and training hyperparameters through a modified grid search, followed by a more nuanced procedure of fine adjustment. To benchmark the accuracy of my final original model, I compared its accuracy



```
# Number of convolutional layers
N_CONV_LAYERS = 5
# Number of fully connected layers
N_FC_LAYERS = 7
# Batch size
BS = 32
# Weight decay (for Adam)
WD = 0
# A trous convolution dilation factor (default is 1)
AT = 1
# Learning rate (for Adam)
LR = 0.001
```

Fig. 1. I built an infrastructure allowing for large changes in the architecture and training hyperparameters of the model, simply by altering the values in this Jupyter notebook cell.

to the accuracies of three models that are pretrained on the ImageNet dataset: AlexNet, ResNet-50, and VGG. My best original model produced a validation accuracy of 45.27%, compared with AlexNet's validation accuracy of 29.17%, ResNet-50's validation accuracy of 70.34%, and VGG16's estimated validation accuracy of 83.56%.

II. Approach

I built my model using PyTorch and a Jupyter notebook. I started with an existing notebook provided for a different assignment in CS 747, but thoroughly gutted and repurposed it [4]. I used Adam as my optimizer. Convolutional layers are followed by a ReLU nonlinearity and maxpooling, and fully connected layers are followed by a ReLU nonlinearity. To find an ideal combination of architecture and training hyperparameters as efficiently as possible, I created an infrastructure for automatically building convolutional neural nets (CNNs) of various architectures and hyperparameter values. This infrastructure allowed me to alter the following attributes simply by changing one value in my Jupyter notebook per attribute: (1) number of convolutional layers, (2) number of fully connected layers, (3) batch size, (4) Adam weight decay, (5) a trous convolution dilation factor, and (6) Adam learning rate. The cell of my Jupyter notebook in which these values can be changed is shown in Figure 1

To allow for such ease of modification, I made the `__init__()` and `forward()` methods of my `nn.Module` classifier dynamic. The `__init__()` method now takes the desired number of convolutional layers, number of fully connected layers, and dilation factor as inputs, and generates CNN layers of the appropriate cardinality and dimensionality. The `forward()` method now adapts to the number of layers which `__init__()` created.

To train on the dataset, I first needed to translate it from its existing format (a set of .tfrec files intended for TensorFlow) to a format compatible with PyTorch. To achieve this, I followed an existing tutorial. The tutorial's translation approach involves loading the files into objects intended for TensorFlow, then copying relevant fields from those TensorFlow objects into objects intended for PyTorch [5].

I used cross-entropy loss as my loss function, because it is well known as an effective loss for multiclass image classification [6]. For discerning which final model had the best performance, I used percent accuracy (number of correct class guesses divided by number of samples), as this metric is identical to Average Precision for multiclass classification. For the rest of this report, I use the term "loss" to mean "cross-entropy loss" and "accuracy" to mean "percent accuracy, with a value of 1 meaning 100% accuracy".

A. Data Augmentation

Data augmentation of training data is understood to improve generalization [7]. Once I had obtained a version of the dataset compatible with PyTorch, I augmented the training data with the following perturbations from the torchvision.transforms library. All images were also normalized.

- Random perspective changes using RandomPerspective with a probability of 50% and distortion scale of 0.1.
- Random rotations using RandomRotation with a probability of 100% and a rotation range of $\pm 30°$.
- Random 128x128 crops using RandomCrop.

Originally, I avoided flipping images, on the assumption that chirality could give the model important information in distinguishing between certain flower types. I also avoided blurring the images, as the dataset already consisted of relatively low-resolution images: 128x128. However, as discussed later in this report, I later started using these augmentation techniques as well, and found that they improved the model's performance.

B. Architecture and Hyperparameter Tuning

1) Coarse Adjustment via Grid Search: Finding a well-tuned architecture and hyperparameters (which I collectively refer to as "attributes") started with some coarse tuning via a modified grid search. I made the following two changes to the usual grid search pattern, to accelerate the process while temporarily sacrificing some validity:

1) I altered one attribute of the model at a time, while holding the others constant. Once an optimal value for that attribute had been found, I didn't vary it again during coarse adjustment.
2) Initially, I used $\frac{1}{64}$th of the dataset to accelerate training. Once I had found a well-performing architecture and hyperparameters, I switched to using the whole dataset for more precise tuning.

The loss of validity incurred by these shortcuts was later made up for by fine adjustment, discussed later in this report.

I adjusted attributes in the following order: (1) number of convolutional layers, (2) number of fully connected layers, (3) training batch size, (4) Adam weight decay, (5) a trous convolution dilation factor, and (6) Adam learning rate. Each training session ran for 50 epochs. Whichever value of an attribute produced the highest accuracy, regardless of the epoch at which the accuracy was measured, was considered the best value for that attribute.

The following values were considered for each attribute:

- Number of convolutional layers: 2, 3, 4, 5
- Number of fully connected layers: 1, 3, 5, 7
- Training batch size: 16, 32, 64, 128
- Adam weight decay: 0, 0.001, 0.005, 0.01, 0.05
- A trous convolution dilation factor: 1, 2, 3
- Adam learning rate: 0.0005, 0.001, 0.005, 0.01, 0.05, 0.1

This process of coarse adjustment found the best values to be as follows: 4 convolutional layers, 5 fully connected layers, training batch size of 16, 0 weight decay, dilation factor of 3, and learning rate of 0.0005.

2) Fine Adjustment via Loss and Accuracy Curves: Fine adjustment began with the same model that coarse adjustment concluded with, except replacing certain "best" attributes with alternative values that had shown monotonic reduction in validation loss during coarse adjustment. This initial set of attributes consisted of 4 convolutional layers, 5 fully connected layers, training batch size of 16, 0.005 weight decay, dilation factor of 3, and learning rate of 0.005. However, this model performed poorly on the partial dataset, giving a maximum validation accuracy of 0.0517 (out of a total possible score of 1) after epoch 0 (the first epoch).

Many training issues can be diagnosed with learning curves, and then fixed by altering the model's architecture and/or hyperparameters [8]. The remainder of Fine Adjustment consisted of iterative tweaks to these attributes, guided by the shapes of loss and accuracy curves obtained during training. Because the model at the time hadn't improved its learning after the first epoch, I surmised that its learning rate might be too high, causing it to overshoot loss minima on the loss-vs-weight landscape. I started by reducing the learning rate of the model to 0.0005. The improved the validation accuracy to 0.1034, but this still left significant room for improvement.

To improve the validity of results from this point onward, I started using the entire dataset: 51,012 training samples and 14,848 validation samples. I also noticed that my learning rate of 0.0005 was unusually low for Adam's default of 0.001 in PyTorch [9]. Therefore I tried rerunning training with the learning rate set to 0.001, and all other attributes the same as before – except for a total of 10 epochs instead of 50, to make training time more manageable. This yielded a maximum validation accuracy of 0.2894 after the final epoch. As can be seen in Figure 2, the training and validation losses remained close together even in the later epochs of training. This suggests that overfitting was minimal.

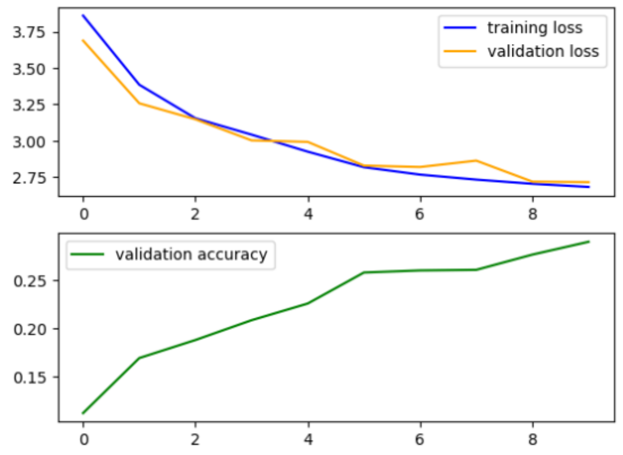


Fig. 2. Losses and accuracies from 10 epochs of training with the following attributes: 4 convolutional layers, 5 fully connected layers, training batch size of 16, 0.005 weight decay, dilation factor of 3, and learning rate of 0.001. Horizontal axis is epoch count, starting at 0. Vertical axis is cross-entropy loss in the upper plot, accuracy in the lower plot.

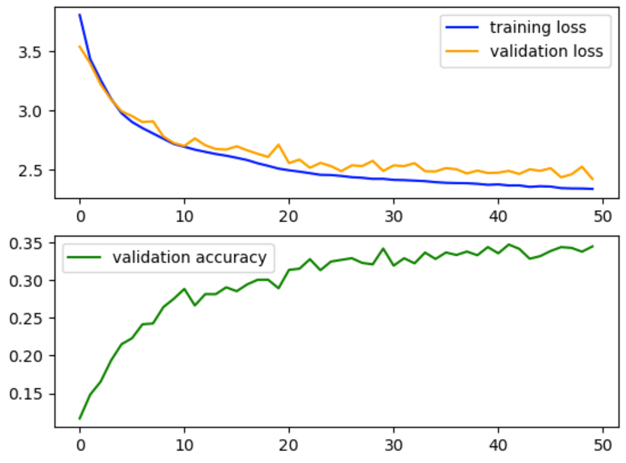


Fig. 3. Losses and accuracies from 50 epochs of training with the following attributes: 4 convolutional layers, 5 fully connected layers, training batch size of 16, 0.005 weight decay, dilation factor of 3, and learning rate of 0.001. Horizontal axis is epoch count, starting at 0. Vertical axis is cross-entropy loss in the upper plot, accuracy in the lower plot.

Because of the low level of overfitting that the previous training session elicited, I chose to rerun the training for more epochs to see if epochwise double-descent would occur. Epochwise double-descent is an occasional phenomenon where continuing to train a model for many epochs after the onset of overfitting will eventually lead to validation loss descending to a new all-time minimum [10]. I chose to retrain for 50 epochs, as a balance between duration and feasibility with limited computation resources. Epochwise double-descent did not occur – in fact, there was hardly any overfitting either – but accuracy continued to improve, plateauing at about 0.34 and peaking at 0.3442 after epoch 41. The loss and accuracy curves can be seen in Figure 3.

The lack of overfitting and low plateau suggested that I should make the model more expressive. I increased the number of convolutional layers to 5 and the number of fully connected layers to 7. However, to increase the number of convolutional layers to 5, I needed to reduce the dilation factor to 1; otherwise the final feature maps would be too small for the convolutional kernel. I also set weight decay to 0 to further increase expressiveness, given the absence of overfitting seen so far. I reran training with these attributes for 10 epochs, and obtained a maximum validation accuracy of 0.3712 after epoch 9. The curves from this run can be seen in Figure 4.

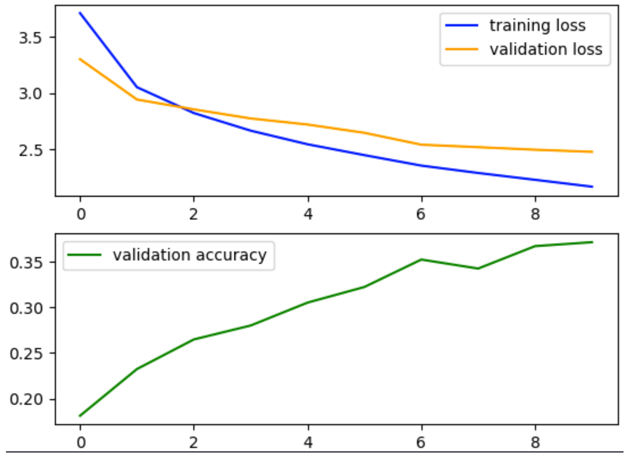


Fig. 4. Losses and accuracies from 10 epochs of training with the following attributes: 5 convolutional layers, 7 fully connected layers, training batch size of 16, 0 weight decay, dilation factor of 1, and learning rate of 0.001. Horizontal axis is epoch count, starting at 0. Vertical axis is cross-entropy loss in the upper plot, accuracy in the lower plot.

At this point, one might predict that further expressiveness might further improve model performance, given the trend set by the previous increase in expressiveness. However, the growing gap between training and validation loss in Figure 4 suggests that further increases in expressiveness might result in an undesirable level of overfitting. Therefore I turned my attention to reducing overfitting instead. I tried increasing weight decay to 0.001, but this gave terrible performance: a maximum validation accuracy of 0.0614 after epoch 0, out of 10 epochs.

I took a break from the overfitting issue to find a new best batch size, given the other attributes from Figure 4. I settled on 32, which elicited a maximum validation accuracy of 0.4298 at epoch 9, out of 10 epochs. The curves from this training session can be seen in Figure 5. I then returned to troubleshooting the overfitting problem.

Because increasing weight decay was not solving the overfitting problem, I tried making the training data augmentation more aggressive. I increased the range of random rotation from $\pm 30^\circ$ to $\pm 180^\circ$, added a 50% chance of a horizontal flip, and a 50% chance to gently blur the image using torchvision.transforms.RandomAdjustSharpness. I reran training with the same attributes from before, producing a model with a maximum accuracy of 0.4083 at epoch 9, out of 10 total epochs. This training session is shown in Figure 6. Although the validation accuracy dropped by about 0.02, overfitting was

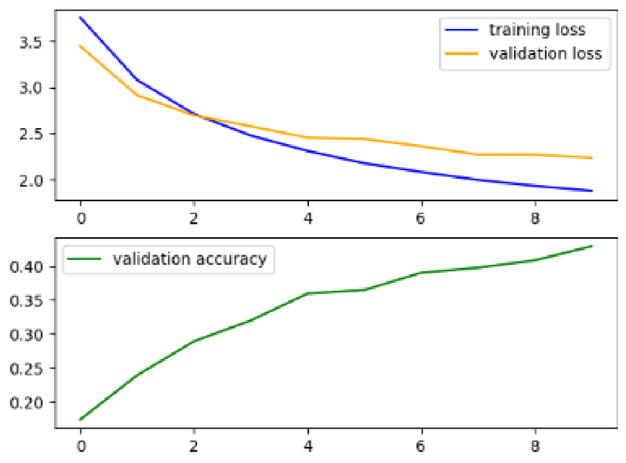


Fig. 5. Losses and accuracies from 10 epochs of training with the following attributes: 5 convolutional layers, 7 fully connected layers, training batch size of 32, 0 weight decay, dilation factor of 1, and learning rate of 0.001. Horizontal axis is epoch count, starting at 0. Vertical axis is cross-entropy loss in the upper plot, accuracy in the lower plot.

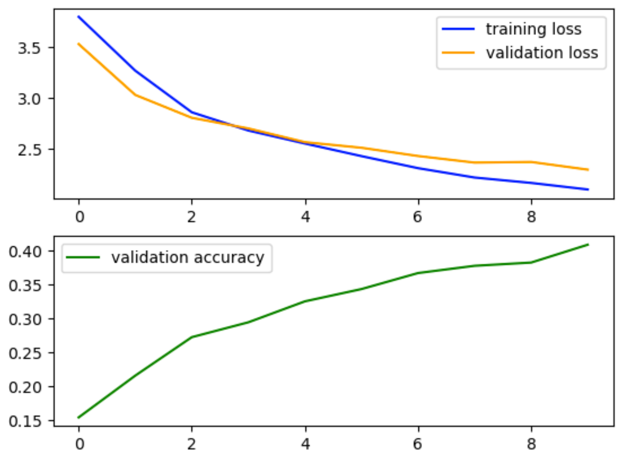


Fig. 6. Losses and accuracies from 10 epochs of training after making data augmentation more aggressive. Horizontal axis is epoch count, starting at 0. Vertical axis is cross-entropy loss in the upper plot, accuracy in the lower plot.

reduced, suggesting that we might get improved performance by training for more epochs.

To see how much more performance could be obtained by training this combination of attributes for more epochs, I set weight decay to 0.00001 (to prevent overfitting from becoming too severe) and reran training for 50 epochs. This resulted in a maximum validation accuracy of 0.4527 after epoch 46. Because I had exhausted the options within my problem scope to further reduce overfitting, I stopped tuning here and accepted this as my best original model. The complete architecture of this model is shown in Figure 7.

III. Results

My final model uses the following attributes: 5 convolutional layers, 7 fully connected layers, training batch size of 32, 0.00001 weight decay, dilation factor of 1, and learning rate of 0.001. Its validation accuracy peaks at 0.4527 after epoch 46. The training curves are shown in Figure 8.

IV. Related Work

Most submissions to the *Petals to the Metal* competition use a VGG16 model pre-trained on ImageNet; the competition suggests in a sample notebook that participants use this model [3]. The median test accuracy of all user submissions on Petals to the Metal is 0.8356.

To see how other pretrained models would perform, I chose to train an AlexNet and a ResNet-50, both pretrained on ImageNet, on the *Petals to the Metal* competition as well [11], [12]. When training each of these models, I trained for 50 epochs using a batch size of 32, a weight decay of 0, and a learning rate of 0.001.

Under these training conditions, the AlexNet model produced a maximum accuracy of 0.2917 at epoch 24. Its full learning curves are shown in Figure 9. The learning curves reveal that the model's losses and accuracy plateau around epoch 15. Interestingly, the validation loss does not worsen with further training beyond this point. Perhaps training for even more epochs would result in overfitting, or perhaps it would result in further descent.

Using the same training conditions, the ResNet-50 model produced a maximum accuracy of 0.7034 at epoch 35. Its full learning curves are shown in Figure 10. The learning curves reveal that the model's losses and accuracy plateau at around epoch 20. Training loss descends to nearly 0 by epoch 20, suggesting that learning almost entirely ceases by that point. The relatively steady gap between the training and validation loss curves suggests that while overfitting is significant, it does not increase much beyond epoch 20. Given that most submissions to the Kaggle competition probably use the VGG16 model provided in the sample notebook, I tentatively estimate that VGG16 produces an average validation accuracy of roughly 0.8356, the median score of all submissions.

See Figure 11 for a visual comparison of the four models' accuracies.

V. Summary/Discussions/Conclusion

I designed a CNN architecture and combination of hyperparameters from scratch to recognize images of 104 types of flowers. I built an infrastructure for easy mixing and matching of these attributes, to facilitate rapid grid searching and fine adjustment. My model's final validation accuracy was 0.4527. I compared its accuracy to AlexNet's final accuracy of 0.2917, ResNet-50's final accuracy of 0.7034, and VGG's estimated final accuracy of roughly 0.8356.

Prof. Kosekca suggested that because *Petals to the Metal* is a fine-grained classification task, my original model might be performing at the upper limit of models that aren't pretrained on some other dataset. However, my model could probably still be improved by decreasing the learning rate further and running for many more epochs, or by using batch normalization. It's possible that such improvements could increase the validation accuracy further.

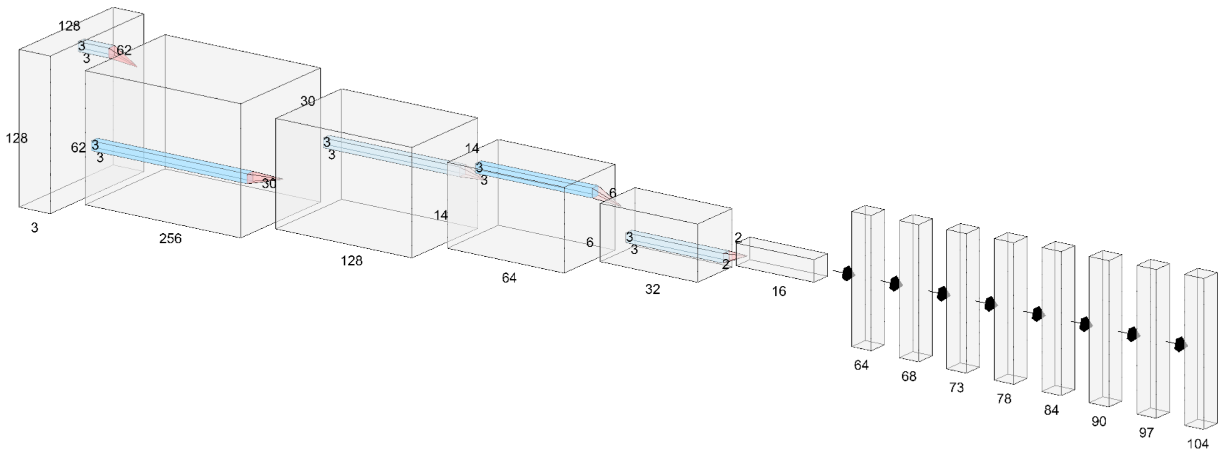


Fig. 7. Final architecture of my original model.

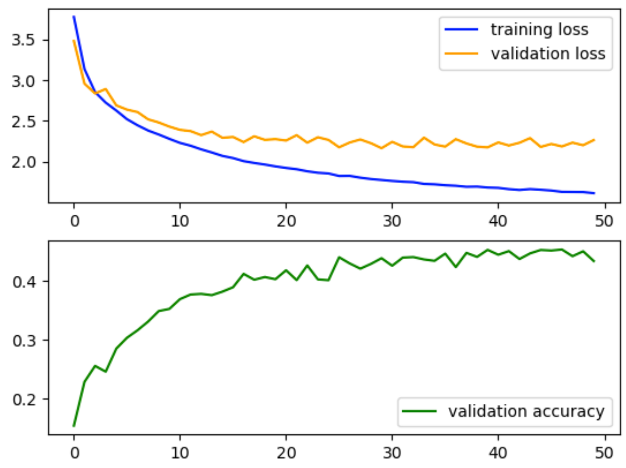


Fig. 8. Losses and accuracies for the best original model from 50 epochs of training.

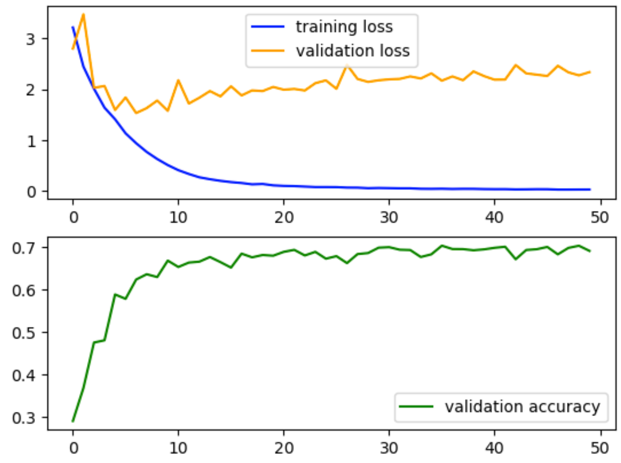


Fig. 10. Losses and accuracies for training an ImageNet-pretrained ResNet-50 model on the *Petals to the Metal* dataset.

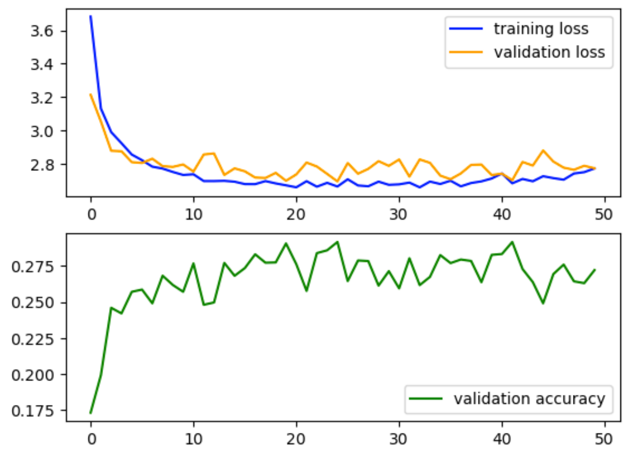


Fig. 9. Losses and accuracies for training an ImageNet-pretrained AlexNet model on the *Petals to the Metal* dataset.

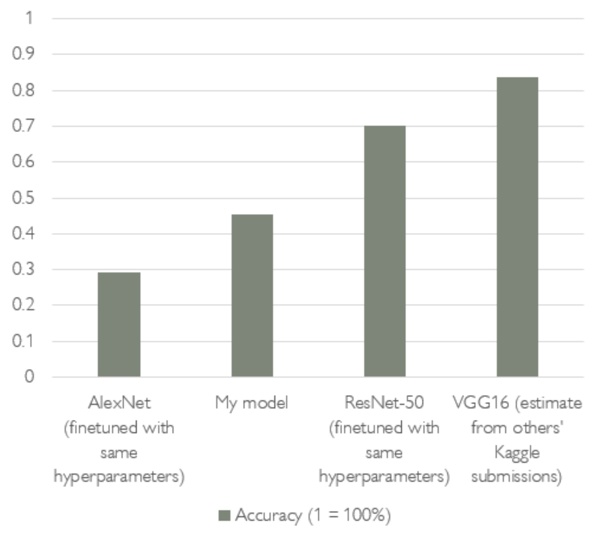


Fig. 11. Comparison of my original model's accuracy with AlexNet, ResNet-50, and VGG16.

References

[1] N. Desai, "Multiclass image classification," https://medium.com/geekculture/multiclass-image-classification-dcf9585f2ff9, accessed: 2022-11-30.

[2] H. Touvron, A. Vedaldi, M. Douze, and H. Jégou, "Fixing the train-test resolution discrepancy," *Advances in neural information processing systems*, vol. 32, 2019.

[3] Kaggle, "Petals to the metal - flower classification on tpu," https://web.archive.org/web/20221103061850/https://www.kaggle.com/competitions/tpu-getting-started/overview, accessed: 2022-11-28.

[4] J. Kosecka, "Assignment 2 part 1: Multi-label image classification," https://web.archive.org/web/20221129161130/https://cs.gmu.edu/~kosecka/cs747/CS747_Assignment_2.html, accessed: 2022-11-29.

[5] S. Chatterjee, "How to read tfrecords files in pytorch!" https://web.archive.org/web/20211023181816/https://medium.com/analytics-vidhya/how-to-read-tfrecords-files-in-pytorch-72763786743f, accessed: 2022-11-28.

[6] J. Brownlee, "How to choose loss functions when training deep learning neural networks," https://web.archive.org/web/20221103075117/https://machinelearningmastery.com/how-to-choose-loss-functions-when-training-deep-learning-neural-networks/, accessed: 2022-11-28.

[7] H. Saleh, *Applied Deep Learning with PyTorch: Demystify neural networks with PyTorch*. Packt Publishing Ltd, 2019.

[8] J. Brownlee, "How to use learning curves to diagnose machine learning model performance," https://web.archive.org/web/20221103075114/https://machinelearningmastery.com/learning-curves-for-diagnosing-machine-learning-model-performance/, accessed: 2022-11-28.

[9] T. P. Foundation, "torch," https://web.archive.org/web/20221117021831/https://pytorch.org/docs/stable/torch.html, accessed: 2022-11-28.

[10] C. Stephenson and T. Lee, "When and how epochwise double descent happens," *arXiv preprint arXiv:2108.12006*, 2021.

[11] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Communications of the ACM*, vol. 60, no. 6, pp. 84–90, 2017.

[12] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.