Flower Recognition Experiments with Convolutional Neural Networks

Charles Booth, cbooth1@stevens.edu

1 Introduction

The main idea of this project is to showcase and experiment with convolutional neural networks (CNNs) based on the architecture from the 2015 paper, Very Deep Convolutional Networks For Large-Scale Image Recognition[1]. CNNs are very popular and widely used for many computer vision tasks, and public python libraries from the Zisserman and Simonyan paper will allow this paper to preform similar experiments on different data in a controlled fashion. The main idea of this project is to use these tools in python and similar experimental conditions covered in this paper, most notably the smaller 3x3 convolutional window size, and increased depth of layers as the experiments go on. More details on the architecture will be further described in the next section of this paper. The dataset that the experiments will be tested on is a flower dataset from kaggle, including 5 types of flowers: daisy, dandelion, rose, sunflower, and tulip. The project will use PyTorch to train the networks, CUDA for faster computational speeds, and the rest of the paper's architecture will be matched by using the same parameters.

2 Problem Formulation

There will be 5 tested networks in this paper, using the parameters from page 3, table 1 of the paper:

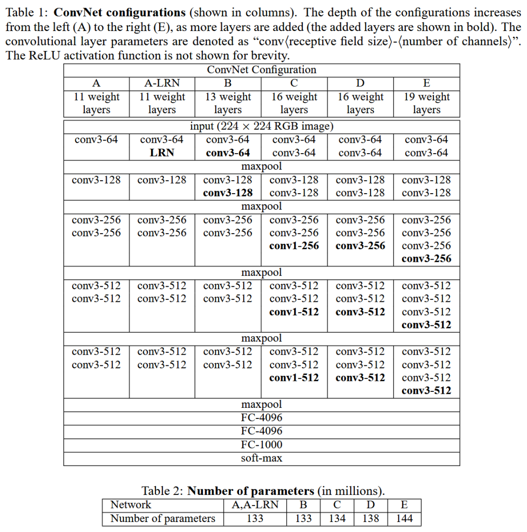


Table 1: **ConvNet configurations** (shown in columns). The depth of the configurations increases from the left (A) to the right (E), as more layers are added (the added layers are shown in bold). The convolutional layer parameters are denoted as "conv⟨receptive field size⟩-⟨number of channels⟩". The ReLU activation function is not shown for brevity.

ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 **conv3-128**	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 **conv1-256**	conv3-256 conv3-256 **conv3-256**	conv3-256 conv3-256 conv3-256 **conv3-256**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

Table 2: **Number of parameters** (in millions).

Network	A,A-LRN	B	C	D	E
Number of parameters	133	133	134	138	144

Figure 1: Parameters to be followed

The idea is to use PyTorch to aid with the architecture creation, and the CNN architecture will learn the 5 classes of the flowers mentioned before. Each of the configurations will follow the same process, beginning with training each model (A, B, C, D, E) on the classes, and then preforming the same tests covered in the paper to compare the performance. The choice to omit the A-LRN (Local Response

Normalization) convolutional layer was intentional, the research paper mentioned it providing no increased accuracy and worse computational preformance. For those reasons, and given the time constraints, it will not be included in this paper. The training process for each is highlighted above, and the tests to evaluate the model at the end will follow the Multi-Scale testing criteria, which will crop the image to a particular resolution in order to fully test the capabilities of the pattern recognition. Another important thing to note is the actual splits of data, 70% testing, 15% training and validation being used across all the models.

3 Methods

The methods used to learn each of the classes follow a classical CNN architecture, where the model is fed a normalized 224×224 RGB image from the training dataset, passing through five convolutional blocks, each followed by max-pooling layers. Following the VGGNet configurations (A through E), the depth increases progressively as more convolutional layers are added to each block. Following the research paper, the model utilizes mini-batch gradient descent with momentum (0.9) to optimize the multinomial logistic regression objective, with each batch containing 256 samples. Training is conducted for up to 50 epochs with an initial learning rate of 0.01, employing early stopping with a patience of 10 epochs if validation loss shows no improvement. Each convolutional layer is followed by ReLU activation functions for non-linearity. After the final max-pooling layer, the feature maps are flattened and passed to the classification head, which consists of three fully-connected layers. Dropout regularization (ratio = 0.5) is applied to the first two fully-connected layers to prevent overfitting. The final fully-connected layer outputs class predictions, which are normalized using a softmax activation function to produce a probability distribution across the five flower classes.

4 Dataset and Experiments

These experiments as previously described will be tested on a kaggle flowers dataset, containing 2505 images of various sizes for training and testing with the respective classes: daisy, dandelion, rose, sunflower, and tulip. Each of these classes have 501 images, so the dataset is evenly split. Considering this, the default accuracy of randomly guessing one type of flower every single time is 20%, which is just another helpful tool to quickly describe the performance of the model over random guessing.

4.1 Model A

The architecture of model A includes 11 weight layers, 8 convolutional and 3 fully connected. The feature extraction component consists of five convolutional blocks with progressively increasing channel dimensions: 64, 128, 256, 512, and 512 channels respectively. Each convolutional layer utilizes 3×3 kernels with padding to preserve spatial dimensions, followed by ReLU activation functions. Max-pooling layers with 2×2 kernels and stride 2 are applied after each block to reduce spatial resolution. Notably, dropout regularization (p=0.5) is applied after the first two convolutional layers to mitigate overfitting. The classification head of the network has 3 layers, increasing the dimensions from 512, 4096, and then 5, the number of classes. ReLU activations are applied after the first two fully-connected layers, with dropout (p=0.5) following each to further regularize the network during training. Weight initialization follows the Kaiming normal scheme for convolutional and linear layers when using 'fan_out' mode, with biases initialized to zero. This initialization strategy is particularly suited for ReLU activations and helps stabilize training dynamics. As a result of this architecture, there were 128,786,821 trainable parameters, which slightly deviates from the paper.

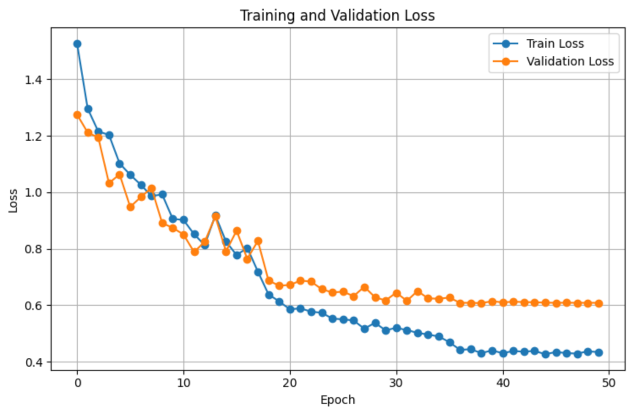


Figure 2: Validation started leveling off around epoch 30

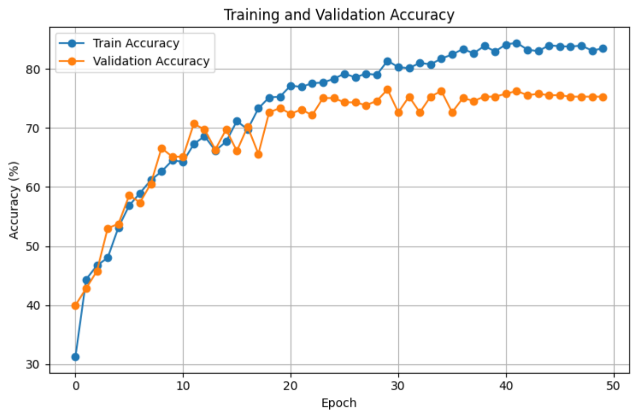


Figure 3: Best validation accuracy ended up being 76.16%

The model ended up speeding up around the 20th epoch, the validation and testing accuracy remaining relatively stagnant afterward. These are good results considering the default accuracy mentioned before, a 50% increase in predictions from default. Below is a table discussing the accuracy per each class from Model A.

Table 1: VGG-A Classification Performance on Test Set

Class	**Precision**	**Recall**	**F1-Score**	**Support**
daisy	0.82	0.83	0.82	76
dandelion	0.75	0.79	0.77	97
rose	0.64	0.65	0.64	75
sunflower	0.82	0.84	0.83	75
tulip	0.69	0.61	0.65	92
Accuracy		0.74		415
Macro avg	0.74	0.74	0.74	415
Weighted avg	0.74	0.74	0.74	415

Per-class Accuracy: daisy = 82.89%, dandelion = 79.38%, rose = 65.33%, sunflower = 84.00%, tulip = 60.87%

4.2 Model B

Model B provides 13 weight layers, keeping the same parameters as Model A but now with an additional 64 and 128 channel convolution layer in the first and second blocks of the network respectively.

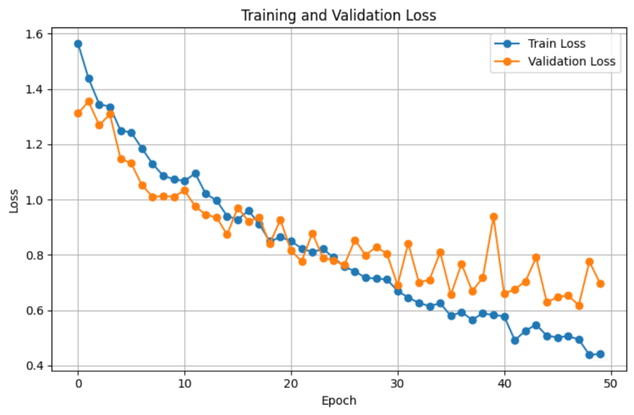


Figure 4: Mini-batch gradient descent had trouble optimizing the batches around epoch 40

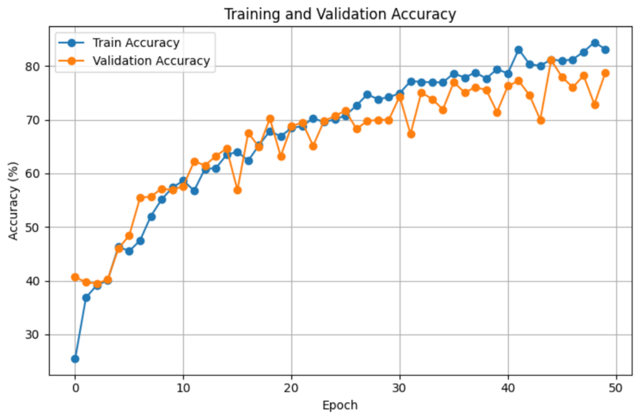


Figure 5: Best validation accuracy ended up being 81.11%

Just like before, the model ended up converging to a slightly improved accuracy around Model A's. The multi-scale testing criteria preformed as follows, obtaining mostly the same results as before.

Table 2: VGG-B Classification Performance on Test Set

Class	**Precision**	**Recall**	**F1-Score**	**Support**
daisy	0.86	0.75	0.80	76
dandelion	0.74	0.81	0.77	97
rose	0.63	0.61	0.62	75
sunflower	0.78	0.85	0.82	75
tulip	0.68	0.64	0.66	92
Accuracy		0.73		415
Macro avg	0.74	0.73	0.73	415
Weighted avg	0.74	0.73	0.73	415

Per-class Accuracy: daisy = 75.00%, dandelion = 81.44%, rose = 61.33%, sunflower = 85.33%, tulip = 64.13%

4.3 Model C

Model C is slightly different than the rest of the models, not just in the fact that adds 3 more weight layers, but that it also reduces the receptive size to just 1x1, a big change from the rest of the models that utilize a 3x3 window. Once again, the preformance is comparable to that of the other models, still decent, and achieving a high of 77.24% accuracy on validation.

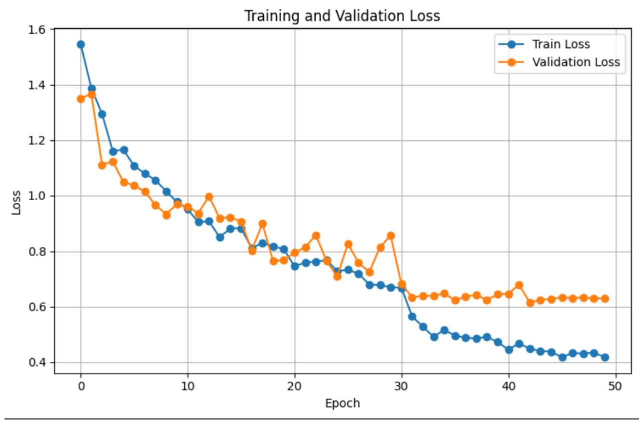


Figure 6: Levels off toward epoch 40

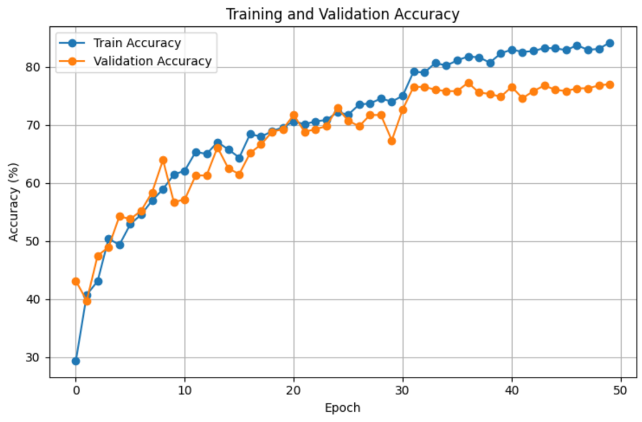


Figure 7: Best validation accuracy achieved at epoch 34, 77.24%

The multi-scale testing criteria favored some classes more than others, likely as a result of an issue with balancing the validation, training, and testing sets.

Table 3: VGG-C Classification Performance on Test Set

Class	**Precision**	**Recall**	**F1-Score**	**Support**
daisy	0.78	0.89	0.83	76
dandelion	0.86	0.78	0.82	97
rose	0.61	0.63	0.62	75
sunflower	0.76	0.89	0.82	75
tulip	0.77	0.63	0.69	92
Accuracy		0.76		415
Macro avg	0.76	0.77	0.736	415
Weighted avg	0.76	0.76	0.76	415

Per-class Accuracy: daisy = 89.47%, dandelion = 78.35%, rose = 62.67%, sunflower = 89.33%, tulip = 63.04%

4.4 Model D

Model D follows the exact same weight parameters as before, but instead of reducing the kernel to 1x1, we return to the 3x3 receptive field size.

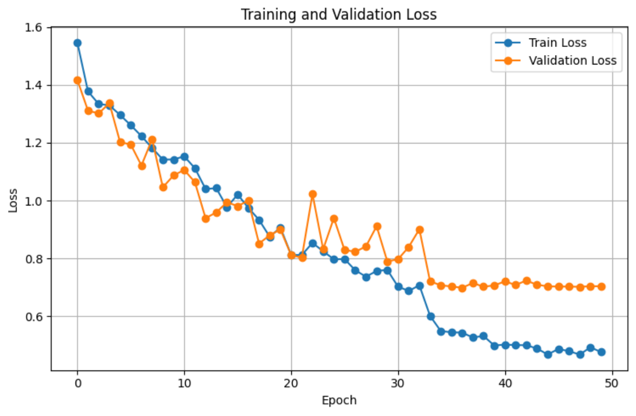


Figure 8: Model D validation vs. Training loss

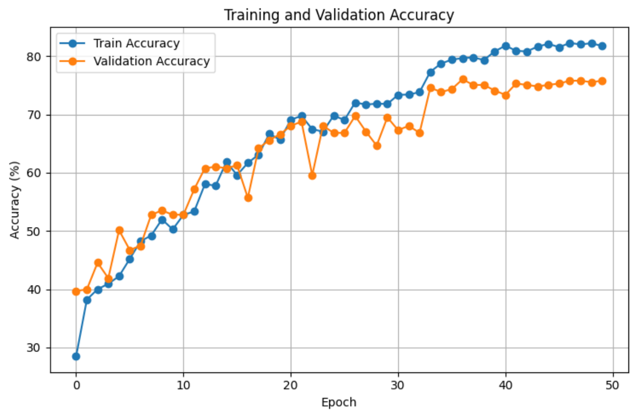


Figure 9: Model D validation accuracy, best score: 76.03%

Model D ended up struggling a lot more with the metric-scaled tests, the accuracies for some classes dropping while others increase despite an increase in parameters.

Table 4: VGG-D Classification Performance on Test Set

Class	**Precision**	**Recall**	**F1-Score**	**Support**
daisy	0.81	0.78	0.79	76
dandelion	0.79	0.81	0.80	97
rose	0.61	0.63	0.62	75
sunflower	0.71	0.87	0.78	75
tulip	0.73	0.58	0.64	92
Accuracy		0.73		415
Macro avg	0.73	0.73	0.73	415
Weighted avg	0.73	0.73	0.73	415

Per-class Accuracy: daisy = 77.63%, dandelion = 81.44%, rose = 62.67%, sunflower = 86.67%, tulip = 57.61%

4.5 Model E

Model E has by far the most weight layers and parameters, totaling 19, and adding an extra 256, and two 512 layers into the last two convolutional blocks. Model E contains 139 million parameters, 10 million more than the previous models A-C.

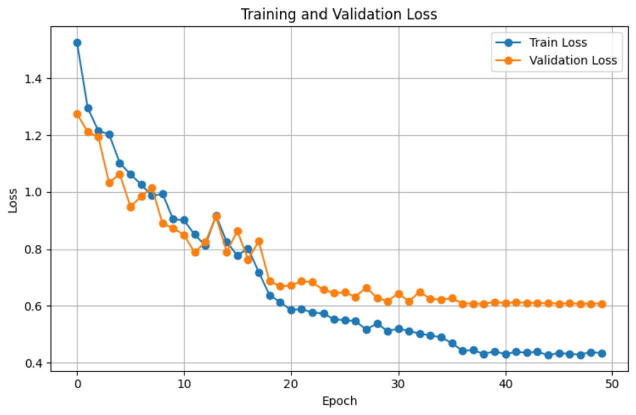


Figure 10: Model E validation vs. Training loss

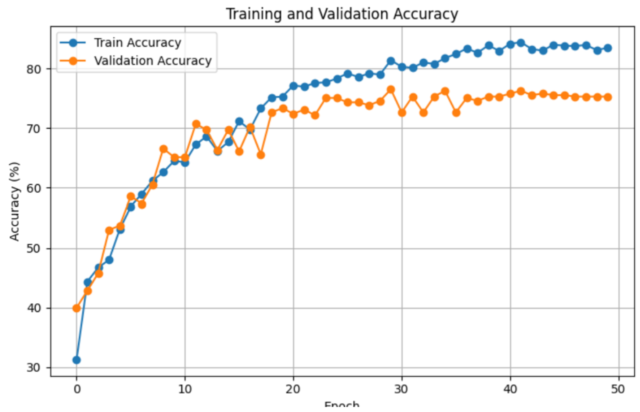


Figure 11: Model E validation accuracy, best score: 76.51%

Model E had a tougher time classifying specific flowers, and converged to a particular accuracy fairly early. It began to overfit as time went on, and maybe a larger kernel size would allow the multi-scale testing to be more accurate.

Table 5: VGG-E Classification Performance on Test Set

Class	**Precision**	**Recall**	**F1-Score**	**Support**
daisy	0.82	0.83	0.82	76
dandelion	0.75	0.79	0.77	97
rose	0.64	0.65	0.64	75
sunflower	0.82	0.84	0.83	75
tulip	0.69	0.61	0.65	92
Accuracy		0.74		415
Macro avg	0.74	0.74	0.74	415
Weighted avg	0.74	0.74	0.74	415

Per-class Accuracy: daisy = 82.89%, dandelion = 79.38%, rose = 65.33%, sunflower = 84.00%, tulip = 60.87%

5 Conclusion

Using the tools and architecture in the research ended up providing a very useful guide on learning novel data with low a dimension kernel and deep layers. The only issue with using a low dimensional kernel is it seems that there is a threshold where using a higher amount of weight layers creates more noise, making it harder for the models to capture the patterns in the underlying images. The highest preforming model was Model B, which achieved a validation accuracy of 81.11%, a 60% improvement from default. I found that implementing this research paper on new data was a very useful exercise for professional practice in research. I am rather new when it comes to the nuance of preforming this kind of research on machine learning applications, but I did end up learning a lot more about CNNs and generally how to write research articles.

Table 6: Comparison of VGG Model Performance Across All Configurations

Model	**Daisy**	**Dandelion**	**Rose**	**Sunflower**	**Tulip**	**Best Val. Acc.**
VGG-A	82.89%	79.38%	65.33%	84.00%	60.87%	76.16%
VGG-B	75.00%	81.44%	61.33%	85.33%	64.13%	81.11%
VGG-C	89.47%	78.35%	62.67%	89.33%	63.04%	77.24%
VGG-D	77.63%	81.44%	62.67%	86.67%	57.61%	76.03%
VGG-E	82.89%	79.38%	65.33%	84.00%	60.87%	76.51%

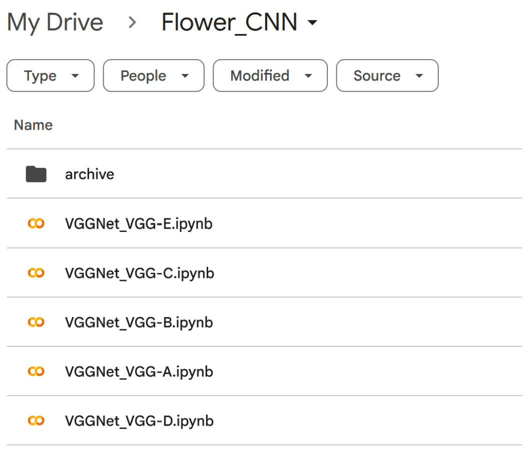
Note: Per-class accuracies represent test set performance, while Best Validation Accuracy indicates the highest accuracy achieved during training on the validation set.

References

[1] Andrew Zisserman Karen Simonyan. Very deep convolutional networks for large-scale image recognition. 2015.

A How to use these models

The environment that I used to run all my models was Google Colab, as it allowed for me to use their T4 GPU to work with CUDA. The only thing that needs to be done by the user to utilize these models is to import the files into Google drive, download the flowers-dataset from kaggle, and keep these files in the same directory. I used Google's mount-point in order to access and train all of the images, and I recommend that the user does the same. In order to properly utilize Google Colab's T4 GPU, make sure to change the runtime type and T4, after that, the code should run properly. Here is a screenshot to show the folder architecture if needed.



archive
VGGNet_VGG-E.ipynb
VGGNet_VGG-C.ipynb
VGGNet_VGG-B.ipynb
VGGNet_VGG-A.ipynb
VGGNet_VGG-D.ipynb

Figure 12: The user may need to change the file paths for each notebook