Classification of Flower Images Using a Convolutional Neural Network

Lewis Morgan - Y3921704
lm2283@york.ac.uk

***Abstract*—In this paper, I'm applying a powerful method of deep learning to classify a diverse collection of flower pictures into 102 species of flora. After some initial attempts, I redesigned my application to be easily configurable from command line arguments. This allowed me to set up many experiments to find the best hyper-parameters. The network has achieved a test accuracy of 65% [Fig 5] after training and validation for 9 hours on an Apple M1 Neural Engine.**

I. Introduction

THE aim of the project is to find a way to accurately classify the test images from the Oxford 102 flower dataset[1]. The dataset contains 1024 images for training and validation and 6149 images for testing. The Oxford website[1] also includes a tar file of the segmentation masks for the full ∼8000 images.

Image classification, a core objective in computer vision, seeks to enable computers to recognise and categorise objects or scenes depicted in images. Traditionally developers relied on writing image analysis algorithms by hand that could spot features like edges and shapes within an image. This approach required significant expertise and did not handle image variations very well. Due to the manual nature of their development, these algorithms were expensive to develop.

Machine learning for image classification aims to make computers learn, extract important information and categorise in the same way a human brain does. Recently, we've seen big improvements in using deep neural networks to recognize patterns in images, so now they're ubiquitous in computer vision algorithms [8][9][10]. They benefit from the traditional approaches by; learning automatically; handling variations well; and the same algorithm can be applied to different types of images. Deep learning is achieved through convolutional 'layers' that apply filters to extract the important information from the images. The many layers are connected and the resulting 'predictions' are output at the end of the pipeline.

II. Method

MY approach to this problem was to make use of the PyTorch[3] framework to create a convolutional neural network (CNN).

Initially, I created a simple PyTorch CNN application with just two layers[12] adding some visual indicators to show a progress bar and report the validation and testing accuracy figures. Then I experimented with adjusting the hyper-parameters, image transformations and some elements of the CNN such as adding pooling and dropouts. For each variation, the results were recorded for analysis to find the best combinations that increased the accuracy scores.

The approach was lengthy and awkward, so I explored various combinations for better accuracy. I reworked the application to accept a wide range of command line arguments to adjust various aspects of the CNN[13]. For example; the number of images in a batch; image dimensions; the number of epochs to run; the number of layers (the app's layer count is configurable); random rotation amount and many more. I wrote a simple script to execute these combinations overnight while my laptop was not in use and I could analyse the results in the daytime.

I later spotted that the Oxford website[1] referenced in the project description, also included the image segmentation masks. So I enhanced the program to download and apply these masks to the images; the idea was to remove the unimportant foliage from the image[Fig. 1-3]. This improved the accuracy at the cost of slower processing[14].

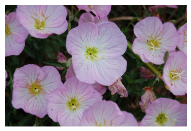


Fig. 1: before image

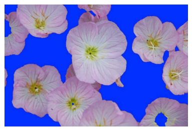


Fig. 2: mask

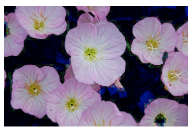


Fig. 3: after image

I read the documentation about the 'normalisation' transformation [4] and noted that there were suggestions that the mean and standard deviation used would be better if based on the actual image dataset, not some arbitrary commonly used values. So I wrote another application to load the dataset and calculate these values for the train images, which I used in my main CNN application.

My flower CNN application with fine-tuned hyper-parameters was getting an accuracy ∼40%. Next, I started to look more widely at how other well-known algorithms worked and if there were any insights or approaches that might help me [2][5][6]. I thought I'd attempt to understand and implement my own version of VGG-16 [6] in my program. In fact, VGG was almost the same as what I was doing but with additional layers with the same number of input as output channels, pooling every second layer and batch normalisation[15]. However, when I started testing I found my computer was just not powerful enough to handle the amount of memory and processing required. So I went back to the original but I applied some of the ideas VGG had in the hope this improved the accuracy (more fc layers, more pooling, batch normalisation)[16].

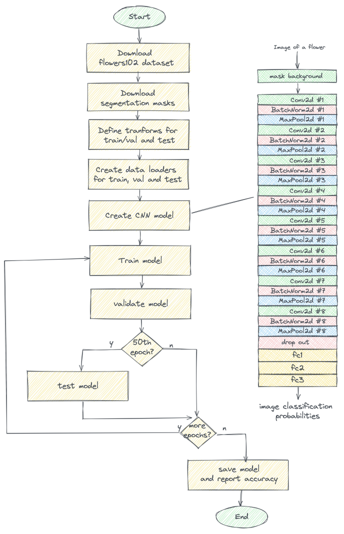
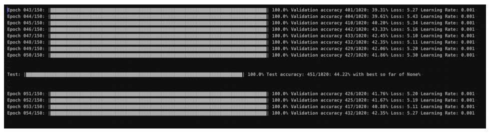


Fig. 4: overview of the application

The application is very flexible, depending on the 'layers' argument passed in, it will run N number of layers of the model [Fig. 4]. The application will train, validate and report those results, but every 50th epoch it will put itself in 'test' mode and attempt the test dataset and report those figures.



```
Epoch 043/150: |████████████████████| 100.0% Validation accuracy 401/1020: 39.31% Loss: 5.27 Learning Rate: 0.001
Epoch 044/150: |████████████████████| 100.0% Validation accuracy 404/1020: 39.61% Loss: 5.43 Learning Rate: 0.001
Epoch 045/150: |████████████████████| 100.0% Validation accuracy 410/1020: 40.20% Loss: 5.34 Learning Rate: 0.001
Epoch 046/150: |████████████████████| 100.0% Validation accuracy 442/1020: 43.33% Loss: 5.16 Learning Rate: 0.001
Epoch 047/150: |████████████████████| 100.0% Validation accuracy 433/1020: 42.45% Loss: 5.10 Learning Rate: 0.001
Epoch 048/150: |████████████████████| 100.0% Validation accuracy 432/1020: 42.35% Loss: 5.11 Learning Rate: 0.001
Epoch 049/150: |████████████████████| 100.0% Validation accuracy 429/1020: 42.06% Loss: 5.20 Learning Rate: 0.001
Epoch 050/150: |████████████████████| 100.0% Validation accuracy 427/1020: 41.86% Loss: 5.30 Learning Rate: 0.001


Test: |████████████████████| 100.0% Test accuracy: 451/1020: 44.22% with best so far of None%


Epoch 051/150: |████████████████████| 100.0% Validation accuracy 426/1020: 41.76% Loss: 5.20 Learning Rate: 0.001
Epoch 052/150: |████████████████████| 100.0% Validation accuracy 425/1020: 41.67% Loss: 5.19 Learning Rate: 0.001
Epoch 053/150: |████████████████████| 100.0% Validation accuracy 417/1020: 40.88% Loss: 5.11 Learning Rate: 0.001
Epoch 054/150: |████████████████████| 100.0% Validation accuracy 432/1020: 42.35% Loss: 5.27 Learning Rate: 0.001
```

The loss function is Pytorch's Cross-Entropy Loss [7]:

$$loss = -\frac{1}{N}\sum_i \left(\log\left(\frac{\sum_j e^{x_{i,j}}}{e^{x_{i,y_i}}}\right)\right)$$

N is the number of samples in the dataset; $x_{i,j}$ is the element of the input corresponding to the class y for sample i; and x_{i,y_i} is the 'correct' element of the input corresponding to the class y for sample i.

This formula calculates the negative log probability of the correct class. The loss is averaged over all samples in the batch. This is interesting as I found experimenting with smaller batch sizes sometimes, but not reliably, produced good accuracy. Bigger batches generally proved to be better. PyTorch automatically applies a softmax function to the network's output before calculating the loss, so I didn't need to include a softmax layer in the network.

III. Results & Evaluation

Running the application through the night with different parameters provided some insights.

Segmentation masks improved accuracy but slowed processing. Some transforms are good, particularly Rotation, Affine, Sharp and Equalize. Others give a moderate advantage, Horizontal and Vertical Flip. Some are to be avoided, Contrast, Blur, Solarize and Jitter. The larger batch sizes 32, 64, 128 and 256 are good, lower sizes are worse. Images of 500x500px where best as that is close to the originals, however, this uses lots more memory. I found with the flips a random rotation of 70% was best.

My app has a configurable number of layers, 6-8 was best in testing for my machine. When I tried with my VGG implementation it had 16 layers and struggled to run on the hardware. I tried SGD for the optimiser but I found Adam to be faster and more accurate. Some things that I tried but didn't help where; BiLinear pooling and reducing learning rate after N tests.

To execute the application with the best parameters and loading existing training data for 8 layers, try:

```
python attempt5.py --num_epochs=500
--batch_size=256
--image_width=500 --image_height=500
--equal=True --sharp=True --rotation=70
--patience=20 --layers=8 --load_training=True
```

I initially executed on the CPU but swapping to an Apple M1 neural engine increased performance substantially. On my attempt1.py it went from 2000 epochs in 105mins on the CPU, to 23mins on this NPU.

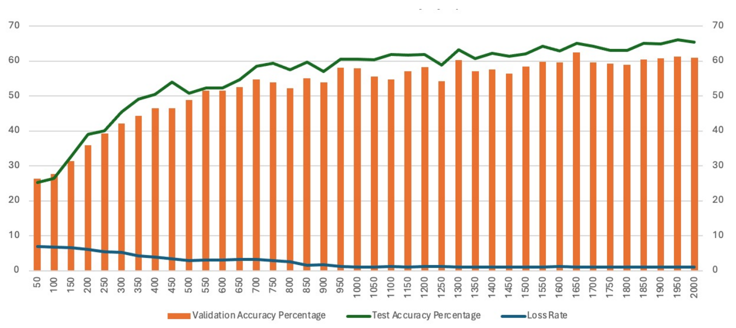


Fig. 5: Validation and test accuracy by epoch

IV. Conclusion & Further work

The CCN developed accurately classifies flowers from this dataset. Given better hardware, more threads, and time it could achieve even higher accuracy.

The approach of making the application very configurable in terms of layers, batch size, epochs, image size, rotation, transforms, etc means it was possible to hone in on the best hyper-parameters for the task. Running multiple scenario tests as night made use of the limited resources efficiently. It was enhanced by the concepts from VGG, and perhaps ideas from other algorithms could take it even further.

For further work, it might be good to write several CNNs using different algorithm approaches and then using 'Ensemble learning' to combine their predictions. Crudely, the app can easily do this now by running (train/validate/test) with various arguments and then saving at the end. Run again with new parameters, load the model (train/validate/test), and save again. Repeat this multiple times.

References

[1] https://www.robots.ox.ac.uk/∼vgg/data/flowers/102/
The Oxford Flower 102 dataset and segmentation masks.

[2] .Simonyan and A. Zisserman, “Very deep concolutional networks for large-scale image recognition”, NETWORKS FOR LARGE-SCALE IMAGE RECOGNITION”, ICLR 2015.

[3] https://pytorch.org
PyTorch is an open-source machine learning (ML) framework based on the Python programming language and the Torch library.

[4] https://pytorch.org/vision/stable/transforms.html
PyTorch Vision transforms documentation.

[5] Isha Patel, Sanskruti Patel, ”An Optimized Deep Learning Model For Flower Classification Using NAS-FPN And Faster RCNN”, 2020, ISSN 2277-8616.

[6] Tanaya Pakhale, ”Transfer Learning using VGG16 in Pytorch” (2021) https://www.analyticsvidhya.com/blog/2021/06/transfer-learning-using-vgg16-in-pytorch/

[7] https://pytorch.org/docs/stable/generated/torch.nn.CrossEntropyLoss.html
PyTorch cross entroy loss documentation.

[8] Wang, X., Sun, L., Lu, C., & Li, B. (2024). ”A Novel Transformer Network with a CNN-Enhanced Cross-Attention Mechanism for Hyperspectral Image Classification. Remote Sensing”, 16(7), 1180. https://doi.org/10.3390/rs16071180

[9] Oyelade, O.N., Irunokhai, E.A. & Wang, H. ”A twin convolutional neural network with hybrid binary optimizer for multimodal breast cancer digital image classification”. Sci Rep 14, 692 (2024). https://doi.org/10.1038/s41598-024-51329-8

[10] Jasmin Praful Bharadiya, ”Convolutional Neural Networks for Image Classification”, 2023, https://www.researchgate.net/profile/Jasmin-Bharadiya-4/publication/370944952_Convolutional_Neural_Networks_for_Image_Classification/links/646b960b7b575d49292a0be3/Convolutional-Neural-Networks-for-Image-Classification.pdf

[11] Lewis Morgan, IMLO Assessment source code
https://github.com/LewisMorgan1/IMLOAssessment

[12] Lewis Morgan - attempt 1
https://github.com/LewisMorgan1/IMLOAssessment/attempt1.py

[13] Lewis Morgan - attempt 2
https://github.com/LewisMorgan1/IMLOAssessment/attempt2.py

[14] Lewis Morgan - attempt 3
https://github.com/LewisMorgan1/IMLOAssessment/attempt3.py

[15] Lewis Morgan - attempt 4
https://github.com/LewisMorgan1/IMLOAssessment/attempt4.py

[16] Lewis Morgan - attempt 5
https://github.com/LewisMorgan1/IMLOAssessment/attempt5.py