Intelligent Systems: Image Classification

Sophia Taylor

***Abstract*—The task for this coursework is to construct a neural network architecture, and evaluate its accuracy as an image classification model trained on the flowers-102 dataset. Using PyTorch, I have built this architecture, specifically a convolutional neural network, in order to create a model in which it is trained to match a label to the given image. This is trained, validated and tested using a series of subroutines, as well as the data in dataloaders. Using this methodology, I was able to create a trained model with an accuracy of 62%**

I. Introduction

FOR this project, we were instructed to create an image classifier using a neural network architecture. This model was then to be trained with the flowers-102 dataset. This dataset consists of 102 flower categories commonly occurring in the UK, with each class consisting of 40-258 images. Image classification is the fundamental task in visual recognition that aims to categorise an image as a whole under a specific label. This can be utilised in a medium of different fields, from detecting and classifying objects in autonomous driving systems to automating data categorisation on online stores. With this multitude of uses, image classification is becoming evermore prominent, highlighting its importance in developments in technology as time goes on.

Over the years, the approach to this problem has evolved. One of the earliest works was done as early as 1989 by Yann LeCun [1], this work was followed up and in 1998, presented the well known architecture known as 'LeNet-5' which became one of the default architectures that people used and began improving upon. Subsequently, there have been a multitude of developments, such as the AlexNet in 2012, where the authors managed to drop the error rate significantly using deep learning approaches, specifically CNNs. I chose to follow this approach due to its known success rate.

II. Method

This model was created in python, specifically in VS-code, using the open source machine learning framework PyTorch. I first created a model class, which consisted of 2 stacks, one consisting of a series of convolutional layers and the other being fully connected layers. These layers are then trained and validated, going through each of the layers using the forward function in the class on every epoch, in order to increase the models accuracy as a whole.

Before the model is trained, the data is split into training, testing and validation data using the default splits as specified by the assessment. This data, particularly train data, is augmented with a series of transformations using the v2 transforms package in PyTorch. These augmentations are explained in further detail in the results and evaluation section.

For the first stack, there are 5 convolutional layers, with additional pooling layers. A convolution layer consists of a mathematical operation which is proficient in extracting features. Each convolution is then normalised using BatchNorm2d. The purpose of batch normalisation is to stabilise the neural network and theoretically help to reduce overfitting. The both of these are then pooled using MaxPool2d, which down samples the layers, halving the size. This reduction in the input size can result in improved statistical efficiency and reduced memory requirements [2]. This is due to an introduction of invariance.

For the second stack, there are 2 fully connected layers, this is including the output layer. Fully connected layers first take the input features extracted from the output of the convolutional and pooling layers, and uses this for classifying the particular input image to a label.

In between the convolutional layers, as well as between the fully connected layers is the ReLU function. This is used to introduce non-linearity within a neural network.

In between the fully connected layers I have chosen to use a dropout function. Dropout is a technique that helps with the ovefitting problem, by randomly dropping units (along with their connections) from the neural network while training. This prevents units from co-adapting too much[3].

In terms of the number of convolutional and fully connected layers, I found that these worked best with my specific data augmentation and adding or subtracting any led to the model overfitting or not performing as well.

Most deep learning algorithms involve optimization of some sort. This refers to the task of either minimising or maximising some function f(x) by altering x. This is where the loss function comes in, where this first function is minimised [2]. This loss function is crucial and quantifies the difference between predicted and target values in this case being the labels for the classified images. For this particular model, I chose to use the CrossEntropyLoss function in PyTorch, this is because it is specifically used for multi-class classification use cases, and expects labels by default in contrast to other PyTorch loss functions.

III. Results and Evaluation

For this section I have decided to use testing accuracy in a different file with the saved model as means to test and properly evaluate the experiments that were carried out. If I were to use the accuracy in the training data it would be affected by overfitting and may not give an accurate representation of the true accuracy of the model.

To begin with I decided to look into image augmentation. In this case of not necessarily having sufficient training data, the regularisation technologies are commonly used to prevent

overfitting. Data augmentation, which refers to creating new similar samples to the training set, can be regarded as one of these technologies [4]. I used the v2 transforms due to them being more time efficient in comparison to the older library, along with this calculating values for the mean and standard deviation of the training dataset for the normalisation function. With these transformations I started off only with normalisation, resizing and flipping. As the project went on and I changed other hyperparameters, I increased both the amount and variety of transformations used. This decreased the amount of overfitting there was substantially, even though it took more epochs to get to a higher percentage.

For the following experiments I used the default hyperparameter choices I had made to begin with. This included batch size of 16, the optimiser function containing a learning rate of 1e-3, a momentum of 0.9, and a weight decay of 1e-3, and 50 epochs.

After sorting the data augmentation, I chose to experiment with parameters - specifically the out channels aka the hidden layers within the convolutional layers. After trying out 16, 32, 64 and 128 as the first out layer after the initial 3, I settled with using 64 and doubling this value on each new convolution as this seemed most compatible with my framework and image augmentation.

Once the parameters for my model had been chosen it was time to move on to adjusting hyperparameters the first being the number of epochs. Generally the number of epochs can improve the overall accuracy of a model, however, too many can cause the issue of overfitting. So the first experiment was finding a good number of epochs before overfitting becomes too much of an issue. With the rest of the hyperparameters begin the same as above, the following table details the results of this experiment.

number of epochs	50	100	150	200	250	300	350
test accuracy	30	37	47	58	60	62	62

From this table it can be deduced that 300 epochs is the optimal amount of epochs before the model begins overfitting too much, and not making an overall difference for the end testing accuracy. So 300 will be used as the new hyperparameter for training the model, keeping all others the same.

I next chose to experiment with the hyperparameter of batch sizes. This hyperparameter represents a number of training samples that will be used during the training in order in order to make one update to the network parameters, in summary, controls the number of predictions made at one time [5]. With the hyperparameters being the same as the newly calculated hyperparameters mentioned earlier in this section, the table below represents the effect of batch size on the overall test accuracy of the model.

batch size	8	16	32
test accuracy	62	62	59

As can be seen in this table, the smaller the batch size, the more accurate the test data was. So after this experimentation I chose to use a batch size of 8 for this model.

I chose to use optim.SDG for my model since this was what was used in the majority of examples I had seen for image classification. This function consists of 3 hyperparameters being learning rate, momentum and weight decay. Throughout the duration of the project I did not vary from the original values I chose due to timing constraints.

However, before choosing I did research the most commonly used numbers for each of these hyperparameters in image classification settings. Using momentum in stochastic gradient descent is proven to be a simple effective way of overcoming the slow convergence problem of SDG[6]. In terms of the value it involves a dynamic equilibrium, so after reading articles and websites with other examples I chose to use 0.9 as my value. For the learning rate, I used 1e-3 as I had seen that previously used in a variety of image classification examples, with the learning rate being a parameter which governs the pace at which the algorithm updates. Then came onto weight decay, which improves the quality of learning of complex patterns, and as this links somewhat to the learning rate I chose to make both of these values equal to the same.

I ran this training program on my laptop which has a NVIDIA GeForce RTX 4060 Laptop GPU. The code itself was ran through anaconda on VScode and overall took a little over 2 hours to train.

IV. Diagram of Network Architecture

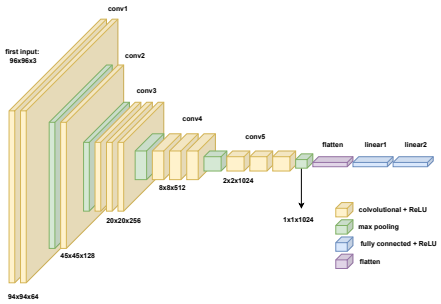

Fig. 1. Convolutional Neural Network Diagram

The diagram above is the convolutional neural network diagram I made in draw.io.

V. Conclusion

I was overall happy with the performance achieved considering it was prohibited to change the splitting of the data, and to use pretrained models. I was happy with the architecture I chose to use, I feel that the amount of convolutional layers and fully connected layers worked with the augmentation I used for the images. I feel that I could work on the data augmentation more, as on further thought, with the image size I have chosen to use, it could get rid of some of the detail of the flower making classifying them harder for the model. If I were to do this project again I would try more out in terms of image augmentation, as well as experiment with the values of the optimisation function making it work with my model the best it can.

REFERENCES

[1] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel, "Backpropagation Applied to Handwritten Zip Code Recognition," *Neural Computation*, vol. 1, no. 4, pp. 541–551, 12 1989. [Online]. Available: https://doi.org/10.1162/neco.1989.1.4.541

[2] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[3] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting," *The journal of machine learning research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[4] J. Shijie, W. Ping, J. Peiyi, and H. Siping, "Research on data augmentation for image classification based on convolution neural networks," in *2017 Chinese Automation Congress (CAC)*, 2017, pp. 4165–4170.

[5] P. M. Radiuk, "Impact of training set batch size on the performance of convolutional neural networks for diverse datasets," *Information Technology and Management Science*, vol. 20, no. 1, pp. 20–24, 2017.

[6] B.-C. Chen, A.-X. Chen, X. Chai, and R. Bian, "A statistical test of the effect of learning rate and momentum coefficient of sgd and its interaction on neural network performance," in *2019 3rd International Conference on Data Science and Business Analytics (ICDSBA)*, 2019, pp. 276–281.