Image Classification using a Convolutional Neural Network on the Flowers-102 Dataset

Y3911637

***Abstract*—In this report, I research how to construct a suitable neural network classifier, using the flowers-102 dataset for training and flower classification. Due to this research, I created a convolutional neural network (CNN) that utilizes 6 convolutional layers and 4 fully connected layers. Furthermore, I implemented methods such as a cross-entropy loss function, the optimizer AdamW, and a learning rate scheduler that reduces the learning rate on plateau. I reached these findings by reviewing a multitude of methods on the PyTorch documentation, and by surveying different articles about CNN's. Additionally, I experimented with different hyper-parameters, to fine-tune my CNN classifier. In doing so I achieved a classification accuracy of 60.30% while training my network for 150 epochs in 38.87 minutes.**

I. Introduction

THIS report focuses on my research into constructing a suitable neural network classifier for the flowers-102 dataset. By utilizing an image as an input my neural network architecture must then predict the specific flower class as its output. Knowing my neural network operates on image classification I subsequently decided to use a convolutional neural network(CNN). Since it is ideal for image classification due to its ability to learn features and patterns from an input image. Traditionally they are used for all sorts of image classification for instance facial recognition, medical imaging, autonomous driving, and document analysis. Other researchers have been tackling the issue of image classification for years, the first CNN was invented in the 1980s, called Neocognitron it was used to recognize Japanese characters, which compared to the technology of today is a lot easier to compute on more recent CNN architectures. Such as ResNet created in 2015. This CNN is extremely powerful and has inspired many other CNN architectures due to it using skip connections allowing the network to go deeper than usual CNN's with up to 152 layers without comprising the generalisation power of the model. The advancements made in image classification are very important because they can provide us with a better quality of life such as better services like autonomous vehicles or saving lives by identifying anomalies in the body that a doctor wouldn't be able to pick up. Therefore due to its importance, I have researched how I shall tackle this problem of image classification. Through reading articles and going through the PyTorch documentation, I have constructed a CNN that utilizes 6 convolutional layers and 4 fully connected layers and implemented methods such as a cross-entropy loss function, the optimizer AdamW, and a learning rate scheduler that reduces the learning rate on a plateau which should be suitable to classify the flowers-102 dataset.

II. Method

A. Layers

The CNN I have constructed employs 6 convolutional layers, starting with an in-channel of 3 to accommodate for RGB, and thereafter increasing filter sizes 64, 128, 256, 512, 512, and 1024. This was chosen so that I could extract deeper and more complex features and patterns for instance finer edges, textures, and shapes from the input image, while also keeping within 15GB of RAM, attributable to Google Colab's T4 GPU which I used for the majority of my experiments on constructing this model. The kernel size within my convolutional layers is 3, with a stride and padding of 1, I found this to be the most optimal way to retrieve features without missing pixels. I also introduced max pooling of size 2 by 2, after every convolutional layer. Pooling is applied to reduce the computational load on the GPU while also managing overfitting by reducing the spatial dimension of the feature map after each convolutional layer. Additionally along with convolutional layers I used 4 fully connected layers that are used to classify the image by combining all the extracted features and then bringing it to a prediction. The fully connected layers I implemented takes the in-feature 9216 and converts it to 2048, 1024, 512, and then 102 to match the amount of classes in the dataset to predict. Moreover, to prevent this complex model from overfitting I applied a dropout of probability 0.5 to the first two fully connected layers. Dropout is a method that randomly sets a fraction of input units to zero, doing this prevents overfitting by making it more challenging for the CNN to find a pattern, thus making it look for new patterns instead of stagnating. On top of that, within every layer except the last fully connected layer(output layer) I used the activation function ReLU(Rectified Linear Unit) to accomplish non-linearity, enabling the model to learn increasingly complex patterns. Another vital aspect of the CNN is batch normalization. On all convolutional layers, I use 2D batch normalization, and for all fully connected layers except the output layer I used 1D batch normalization. Batch normalization is made use of to stabilize the layers inputs, and also speeds up the processing of the model. Finally, the model returns in the output layer the probabilities for each class, using the logits which were converted by the log softmax activation.

B. Loss Function

The loss function I decided to train my CNN with was the Cross-Entropy Loss function provided by the PyTorch library. This loss function combines log softmax(returns the log probabilities for each class) and negative log-likelihood

loss(measures the difference between the predicted probability and the actual) to evaluate how well the model performs on the dataset. This evaluation is then decreased every Epoch using an optimizer. The optimizer chosen through my research was AdamW(Adaptive Moment Estimation with Weight Decay). An optimizer takes the gradients computed and then minimizes the loss function. The AdamW optimizer uses adaptive learning rates which I initially set to 0.0001, thus leading to quicker convergence and a much better performance than for example Stochastic Gradient Descent with Momentum. Weight decay is also implemented to further prevent overfitting and increase the generalization of the CNN, it works by penalizing large weights and decreasing them so that the network doesn't become too complex and after experimenting with different decays I settled with a weight decay of $1e-5$. Additionally to help further the performance of the learning rate especially I used a learning rate scheduler. This scheduler reduces the learning rate on a plateau. The plateau is derived from stagnation in the returned validation loss for a certain length of stagnation called patience. My patience was set to 5 as I found other values to cause overfitting. Along with patience the learning rate scheduler also has the parameters mode and factor which I set to "min" and 0.5 respectively. Mode "min" evaluates the validation loss and then reduces the learning rate when it stagnates, and the factor is how much we decrease the learning rate by, also additionally you must include a scheduler.step() with the validation loss passed as a parameter in the epoch loop so that the learning rate updates. Furthermore, to continue with the loss function, we must look into the training loop. Within the training loop I compute the gradients by first zeroing the gradients and then performing two function calls to retrieve the output from the model and loss from the loss function. Once retrieved it then uses Backpropagation to compute the gradients by executing loss.backwards(), after this update the weights using the optimizer with optimizer.step(). Finally, it then accumulates the loss over all batches and returns the average loss of the epoch.

$$\ell(x, y) = L = \{l_1, \dots, l_N\}^\top, \quad l_n = -w_{y_n} \log \frac{\exp(x_{n,y_n})}{\sum_{c=1}^{C} \exp(x_{n,c})} \cdot 1\{y_n \neq \text{ignore\_index}\}$$

Fig. 1. Formula for Cross-Entropy Loss function courtesy of PyTorch[1]

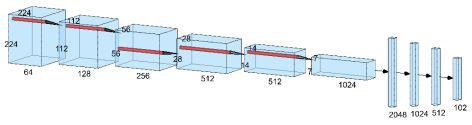

Fig. 2. Visualisation of layers in CNN with dimensions

III. Results and Evaluation

The above-mentioned methods are the final findings of my research, but to get to those methods I had to experiment thoroughly with different values for every hyperparameter, different functions, and transformations. To evaluate all these changes I used the final validation accuracy in the training loop as a metric of how well the change performed. To not waste time and also stop unnecessary overfitting I also implemented an early stop algorithm that stops the loop if the validation loss stagnates for 10 consecutive epochs. Below is a table of my findings showing the result of generalized experiments:

Experiment:	Result:
3 convolutional layers	Lower overall accuracy faster to overfit
4 convolutional layers	Medium overall accuracy still fast to overfit
5 convolutional layers	Medium overall accuracy takes longer to overfit, learns slowly
6 convolutional layers	High overall accuracy, and overfits in suitable iterations
Many transformations in the training set	Fast accuracy increase but even faster overfitting
Low transformations on the training set	Gradual increase in accuracy and overfits in suitable iterations
Learning rate: 0.01	Fast accuracy increase, leads to a plateau very quickly
Learning rate: 0.0001	Perfect amount of increase, and takes long to plateau
Learning rate: 0.00001	Takes way too long to increase accuracy, uses too much time.

Fig. 3. Table of comparison between experiment and its result

By running these experiments I concluded that 6 convolutional layers, a low amount of transformations on the training set, and a learning rate of 0.0001 in the optimizer gave the best outcome out of the experiments I ran. When it came to how many neurons I used in my model I found that using the maximum amount of features allowed within 15GB's of GPU RAM was the most optimal. Thus I had a max out channel on my 6Th convolutional layer as 1024, and I gradually decreased my out features in the fully connected layers since it proved to help the accuracy have a gradual increase and overfit slower. Finally, to show the results of my research, I ran the trained model on the test dataset and got a classification accuracy of 60.30%. The trained model was trained for 150 epochs in 38.87 minutes using Google Colab's L4 GPU. All training was done in Google Colab as my environment, and the hardware used was the L4 GPU provided by Google Colab(for testing) or the T4 GPU provided by Google Colab(for experimenting).

IV. Conclusion

In conclusion, my architecture achieved a good performance by attaining 60.30% as its test classification accuracy. I am very impressed by how fast my architecture can run an epoch, and by how far the accuracy was able to get given the limited amount of GPU RAM available while researching. What didn't go very well was the amount of time it took to experiment and work out which methods to use since I had not built a neural network before. Also working out how to use Cuda to run code on the GPU took a lot of troubleshooting. Finally, if I were to do this research again I would invest in better GPUs with more RAM and also experiment with deeper neural networks that would require the extra RAM and processing power.

REFERENCES

[1] PyTorch Developers, *PyTorch Documentation*, version 2.3, 2024. Available: https://pytorch.org/docs/stable/index.html

[2] B. Kumar, *Convolutional Neural Networks: A Brief History of their Evolution*, Medium, 2021. Available: https://medium.com/appyhigh-technology-blog/convolutional-neural-networks-a-brief-history-of-their-evolution-ee3405568597

[3] Nouman, *Writing CNNs from Scratch in PyTorch*, Netherlands: Paperspace, 2021. Available: https://blog.paperspace.com/writing-cnns-from-scratch-in-pytorch/