21COC102 - Advanced Artificial Intelligent Systems

Student ID: B820352

This report outlines the methodology, successes and lessons learned from the deliverable.

Part A

Tools

The main tools utilised were:

- Tensorflow (Keras)
- Matplotlib
- Docker
- Jupyter

I chose tensorflow and keras due to the high level methods that accelerate the iteration process, and the extensive documentation available. Whilst it may be less 'pythonic' when compared to PyTorch, the keras models are undeniably easier to read, and in my experience have less room for error. One example of this would be the fact that with keras, no input dimensions for the layers need to be explicitly set, as they follow from the previous layer.

One of the largest struggles was setting up the development environment, as the GPU utilisation with tensorflow requires extensive modification to the linux environment.
This is why I chose to employ docker, which has an official tensorflow/jupyter notebook image on docker hub. Through running a container on this image, forwarding a port and installing a few extra packages on my desktop linux environment, tensorflow and jupyter can run completely isolated from the rest of the system, whilst still utilising the GPU.
This offers many benefits, the largest being that the dependencies and conflicts are all containerised. For development, this meant simply starting a docker container and using a web browser to go to the forwarded port on loopback.
Matplotlib is the de facto graphing/image displaying library for python and was used for such purposes.

Methods

As a general approach, I attempted to keep the code as readable and efficient as possible. Keras helped with this, but did also push me to learn more abstract methods and ideas in order to utilise it effectively.
The main driving force behind changes made to the models was the loss and accuracy graphs. Most of the decisions made were through experimentation with different values for layers, and layers themselves.

The idea behind the deliverable models was to create two main models. One would be a 'standard' CNN which had the structure often seen with three pairs of convolution/pooling layers, and the second would be built on top of this structure, to attempt to make as effective a model as possible. For both of these models, I performed extensive experimentation to determine the best parameters given the constraints.

Once this was complete, I realised it would be possible to automate the parameter selection process to an extent, which is exactly what I did in order to make the second model as effective as possible.

Analysis of sources

There were four main sources for the deliverable, however the key fifth source was the tensorflow/keras API. I used it extensively in order to further my understanding of the library, especially for the second model. I will now analyse the contributions of the main four sources below.
The most critical source is that of the EuroSAT dataset[2] which I used for this project. There are two variations of the dataset - one as RGB and one with 13 bands. I decided to use the RGB variant as it is easier to understand for a human. The project would not have been possible without the contributions of this paper, namely labelling the 27000 images in the dataset.
In order to use the dataset, I had to learn how to load data into keras. This article showed how to, using the keras utils and preprocessing to cut out costly loading and tensor operations. It informed the decision to use the inbuilt dataset class in keras. I also used it to understand how to plot an image in matplotlib.
Accessing and plotting model loss and accuracy from training history was learned from the tensorflow classification tutorial[4].
Finally, the way to view feature maps from the first layer of the model was learned from the feature maps article[1] at machinelearningmastery.

Changes made to copied code

Only two sections of code were directly copied for the deliverable:

1. The dataloading[3] (Under 'Dataset' in the notebook). Any other way to achieve this result would be much longer and more error-prone.
2. The feature map subplots[1] (Under plot_feature_maps() in 'Plotting Functions' in the notebook). This was similarly the one obvious way to achieve this, any other way would be needlessly difficult.

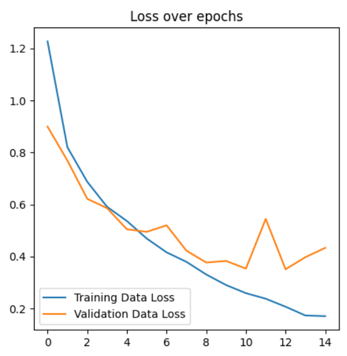
The only other sections which are directly influenced by the sources are the plotting functions[1][4], which similarly have to be written that way as there are only so many ways to use matplotlib.

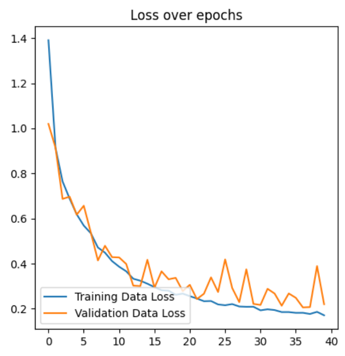
Part B

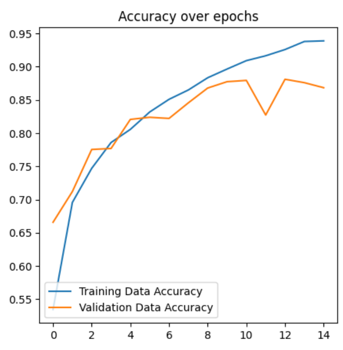
Report of results

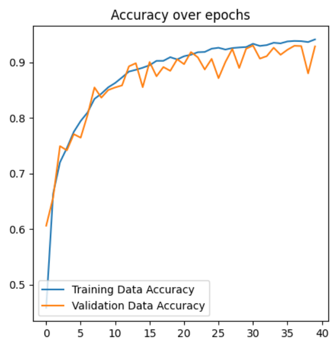
The most telling graphs are those of the loss and accuracy between the optimised versions of the two main models:

Conventional model **Advanced Model**



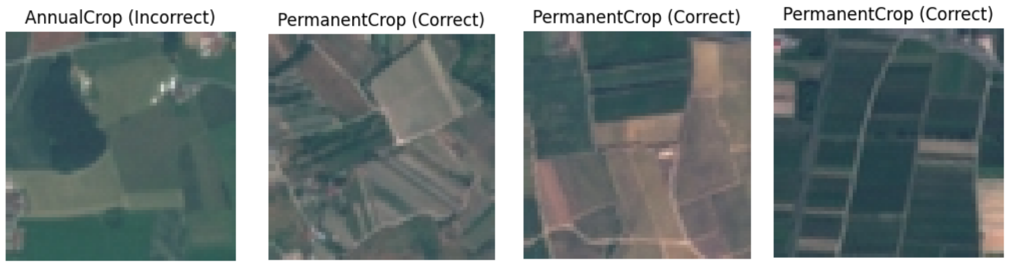






The results overall are very promising for both. An accuracy of 85% on the validation data on the simple model is in itself a good indicator of success, however it's the advanced model with around 93% accuracy that shows the advantage of the extra layers.
At first glance, it may seem that the more complex model requires much more training - however it does achieve the same results as the simpler model in roughly the same number of epochs. It is simply that the extra layers such as the random rotation or gaussian noise allow it to train further - to generalise more. This can be seen especially in the fact that the simpler model continues to improve on the training data, but becomes overfit.

Another important part to note is that oftentimes it is very difficult for a human to determine the difference between the images. The example below shows correct and incorrect evaluations by the advanced model. All show permanent crop:



I am sure that most people would be unable to tell the difference between these. As an example, after seeing examples of each of the classes I could only reliably achieve an accuracy of 85%. This, in my opinion, makes the advanced model accuracy of 93% much more impressive.

Successes

The most successful part of the advancement from simple to complex model was the introduction of the random rotation layer. This had a similar effect as to introducing a lot more training data, as any image was set to rotate up to 180 degrees. I chose this layer specifically because of the type of data explored here - had it been for classifying cars then you are unlikely to want to train a model to spot cars upside down. However, for satellite images this makes perfect sense.

The second biggest success was the introduction of the randomisation layers after flattening. The first attempt with this was to use a dropout layer, which sets random parameters to 0 with the selected frequency rate. This provided some improvement, however the real success was with the introduction of the random gaussian layer. This allowed the model to train much more generally, as with the random rotation layer. This had the added effect of keeping the loss between the training data and validation data much tighter.

The combination of these two findings dramatically improved the model, but they also increased the time to train for both models. This is unsurprising, as the model is in effect seeing slightly different data every epoch. It is also worth noting that although the number of epochs to train has more than doubled, it still only takes 3-4 minutes to train total.

The final major success was the automation of finding the best model parameters. Performing this operation took multiple hours, but provided invaluable insight into finely tuning the model. With the layers static, I automated the system to check through gaussian noise values, the number of neurons in the penultimate dense layer and the kernel sizes. The findings surprised me somewhat, with the current noise value being too high and the penultimate dense layer having too many neurons. This was invaluable, taking the accuracy from around 91% to 93%.

Another small, but time-saving, success was the increase of batch size. Increasing it from 8->16->32 not only made it train faster, but also resulted in less loss. Increasing beyond that point however resulted in worse performance on the validation data. Alongside this, using the inbuilt dataset with keras from the tutorial[3] increased speed considerably over loading it as numpy arrays.

Lessons learned

There were a number of changes to the model found during experimentation which either had no effect, or were detrimental to its performance. The first of which was during the implementation of the standard model. Following the success of the simple model correctly working, I decided to increase the number of trainable parameters. Interestingly, the increase from 0.3m to 19m made it perform much worse. It overfit from the third epoch onwards and loss went from ~0.5 to ~0.6. It also unsurprisingly took much longer to train.

Another failure encountered was the alternative types of activation function. I selected relu at the start because of how common it is, but it also turned out to be by far the best. The only activation function that came close was sigmoid, but that took almost double the number of epochs to train. On a similar vein, I tried multiple optimizers, but Adam (A combination of RMSProp and AdaGrad) almost always trained quickest and resulted in the least loss. This was with the default learning rate of 0.001, any higher and it behaved too erratically. Another lesson was with the kernel size and strides for filtering. I spent a lot of time trying alternative kernel sizes or strides, but 3x3 and a stride of 1 always resulted in the best performance, at least with this dataset.

Comparison to 13 band version

There is also a 13-spectral band version of this exact dataset[2]. Interestingly, the model trained in the paper cited managed 98% accuracy, which is indeed very high, but only 5% higher than my model with the 3 band version. It would be interesting to see a comparison of how much each extra band enhances the results of a model, but that is beyond the scope of this deliverable.

Conclusion

The dataset was well documented and provided an excellent set to perform training on, not proving too difficult to get started on but still providing room for improvement in the later stages. The eventual models performed very well on the dataset. Many of my personal misconceptions about machine learning have been confronted, especially by the automation of the parameter calculation stage. The most surprising part however, has been how effective the basic model, with just combinations of convolution and pooling layers was.

References

[1] Brownlee, Jason. 2019. "*How to Visualize Filters and Feature Maps in Convolutional Neural Networks.*" Machine Learning Mastery. Accessed March 14, 2022. https://machinelearningmastery.com/how-to-visualize-filters-and-feature-maps-in-convolutional-neural-networks/.

[2] Helber, Patrick, Benjamin Bischke, Andreas Dengel, and Damian Borth. n.d. "*phelber/EuroSAT: EuroSAT: Land Use and Land Cover Classification with Sentinel-2.*" GitHub. Accessed March 14, 2022. https://github.com/phelber/eurosat.

[3] Naseer, Nahil. 2020. "*Introduction to Keras, Part One: Data Loading | by Samhita Alla.*" Towards Data Science. Accessed March 14, 2022. https://towardsdatascience.com/introduction-to-keras-part-one-data-loading-43b9c015e27c.

[4] TensorFlow. 2022. "*Image classification.*" TensorFlow Learn. Accessed March 14, 2022. https://www.tensorflow.org/tutorials/images/classification.