AI Kidney Cortex DataSet Training

Introduction

This AI prediction model was created for the second year module Com2028(Artificial Intelligence) lectured at university of surrey.
The objective of this project is to develop the most accurate prediction model possible with a dataset that consists of kidney cortex cells of 8 categories.
For this project we use the BBBC05130, available from the Broad Bioimage Benchmark Collection24 as provided by the university.
The dataset contains 236,386 human kidney cortex cells, segmented from 3 reference tissue specimens and organized into 8 cell categories.
The source dataset was then split with a ratio of 7 : 1 : 2 into training, validation and test sets.
Each gray-scale image is 32×32×7 pixels, where 7 denotes 7 slices.
We take maximum values across the slices and resize them into 28×28 gray-scale images.
This dataset was created by incorporating 3D nuclear signatures from a DNA stain, DAPI as this according to the study can be incorporated in most experimental imaging and can be used in cell classification in intact human kidney tissue.
This is a very interesting dataset because it's very poorly balanced which provides a bigger challenge and is closer to what a real dataset would most likely look like if we were to try and solve a real problem.
This dataset is imbalanced because of the reoccurring existence of the first kind of cell, which overwhelms the other ones making it hard to not overfit.

References

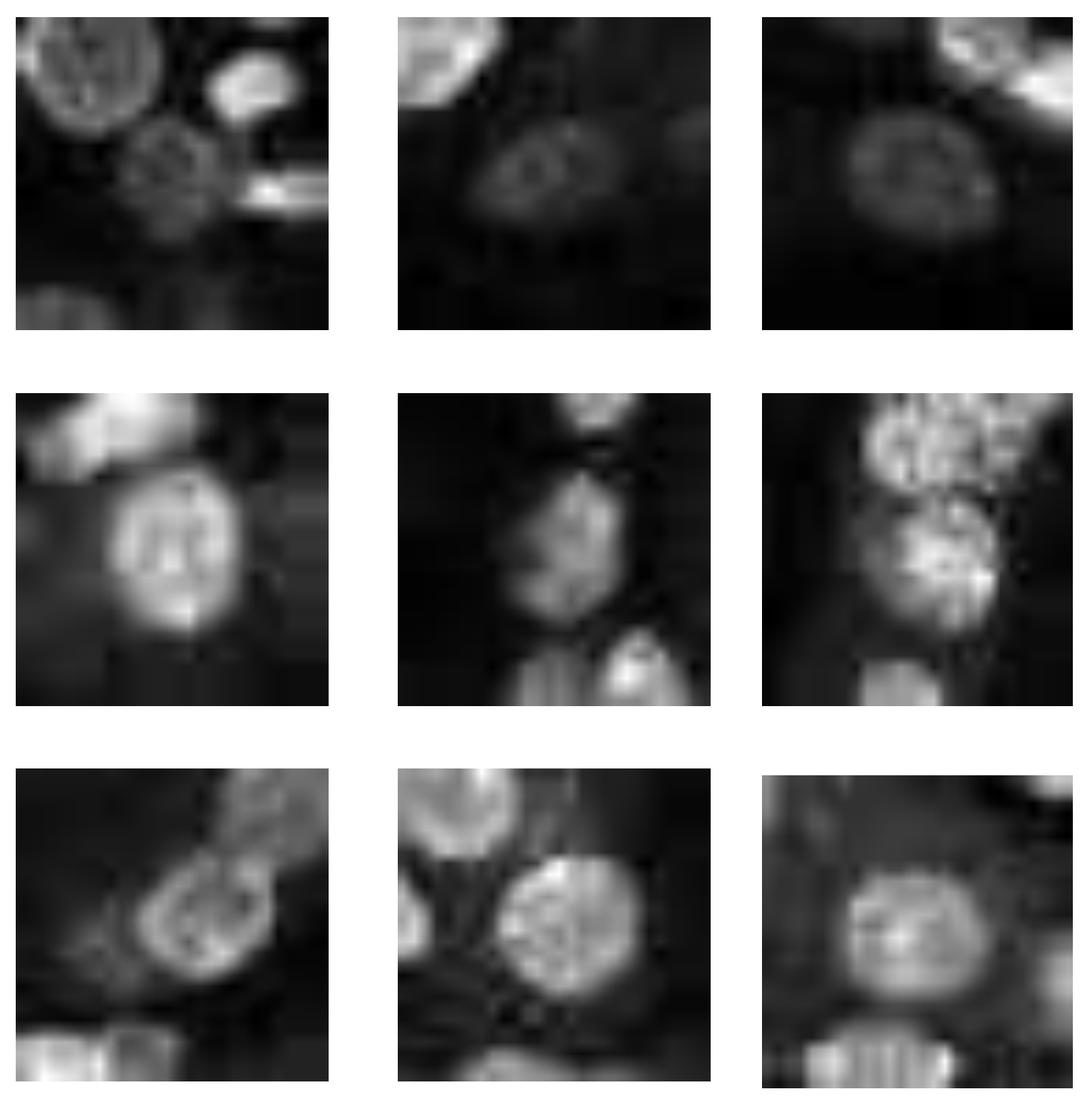
More information about the original dataset used in this project can be found in the flollowing published pappers:

-"In situ classification of cell types in human kidney tissue using 3D nuclear staining." Cytometry Part A;

-"Medmnist v2: A large-scale lightweight benchmark for 2d and 3d biomedical image classification."

Data Example and analysis of the problem

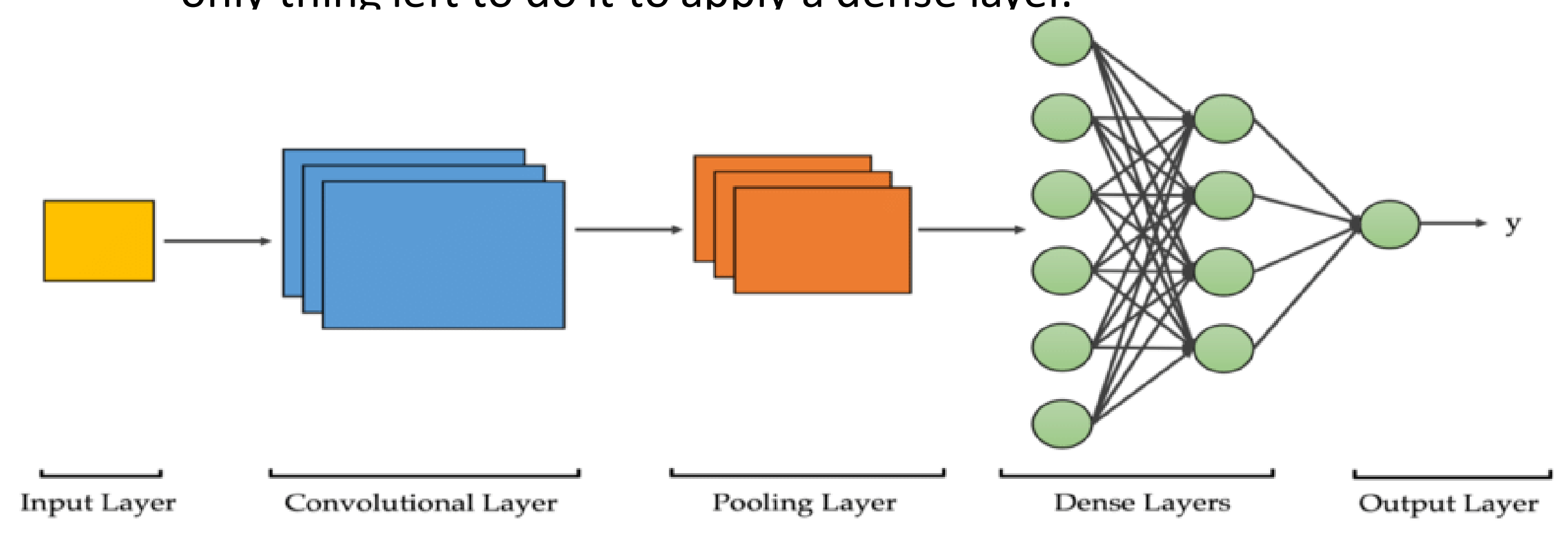
In my first implementation of the model I started by thinking about the key factors of this dataset and how they differentiated from the experience I had from previous datasets, some of the first observations that I had were that the dataset was in black and white, making it obvious right away that there was no point in using anything else to treat the data besides grayscale on the colour mode since it would just create an inaccurate model when compared with the tests that would later be run on kaggle.


Img.1 Sample example taken from the original dataset

The second realization was that this data was divided into 8 classes of cortex cells meaning that the class mode that I would have to use would be categorical.
After having set this up for the dataset on my code, I went ahead and decided how I wanted to split the data and ended up deciding on 80% for the test set and 20% for the validation set, now the only thing in data treatment left would be to match the name of the y and x col with the ones provided on kaggle and choosing some of the parameters like the image size and the batch size I would like to give to my model.
Now after having all the data treated and ready to be given to my model

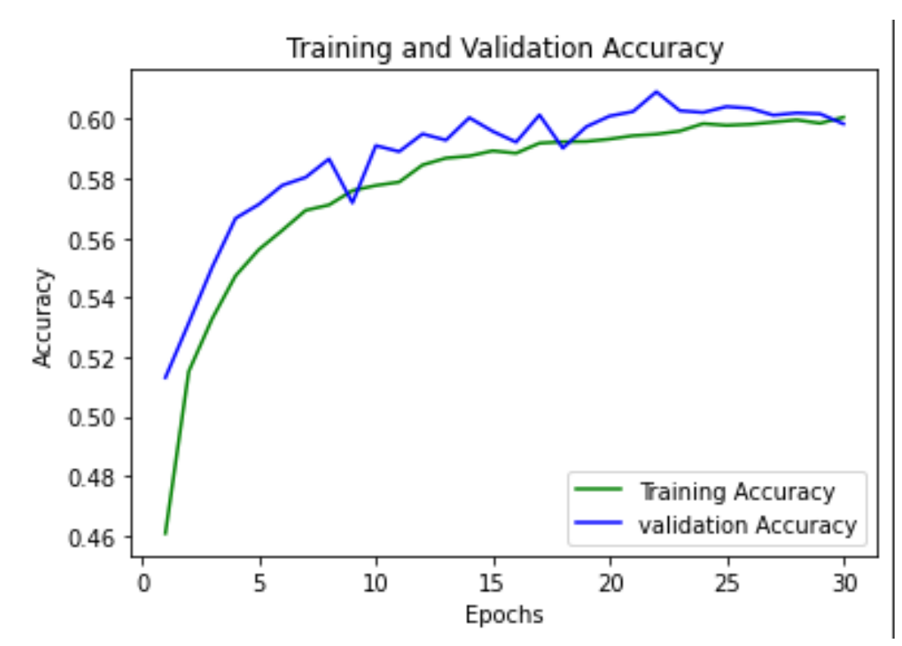
First Implementation

In this section I will be explaining my thought process for my first implementation of the model and how I later found out that this wasn't the best.
I will also be adding some graphs of the model accuracy and loss so we can compare them to the final model and see that there is improvement.
From the beginning I decided to take a CNN(convolutional neural network) approach to the problem, convolutional neural network usually works by applying to the initial input layer one or more convolutional layers that will serve as an activation layer and detect features from the specific object, then usually a pooling layer is used so that we can extract either the max, the average or min feature from the ones identified by the activation layer, by the end the only thing left to do it to apply a dense layer.

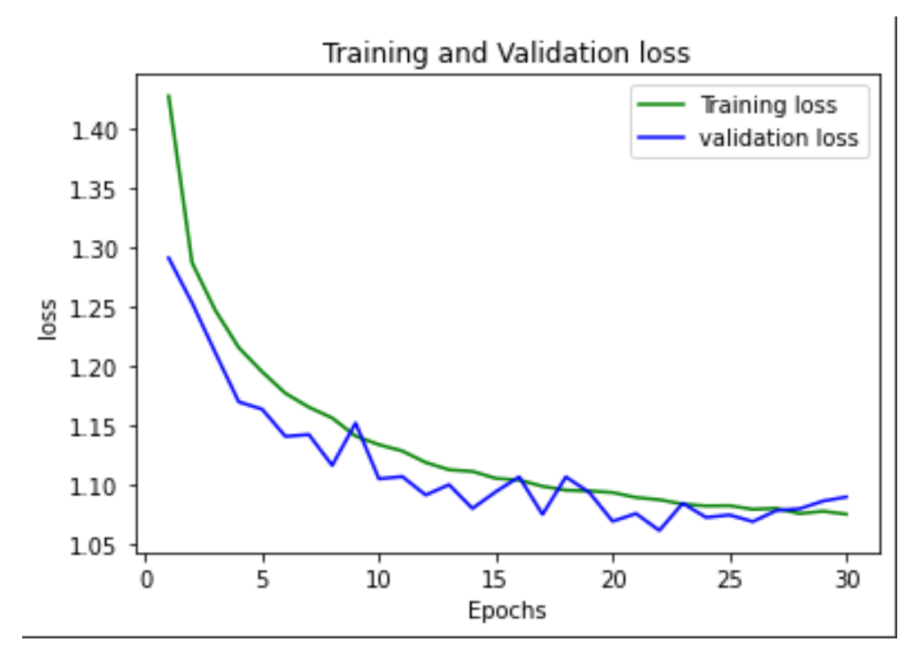


Img.2 Figurative eexample of a CNN

In my case I decided to use a model that uses one max pooling for each two convolution layers and one dropout for each 4 convolutional layers, this is both to help extract the strongest features in the Conv2D and also with the dropout to help getting rid of possible nodes that weren't contributing for the learning of the model.
As a last step I used 'adam' as a optimizer.
At this point I felt pretty confident with my model and decided to test it by running model.fit with 30 epochs and Model Checkpoints



Img.3 First model accuracy map ploted

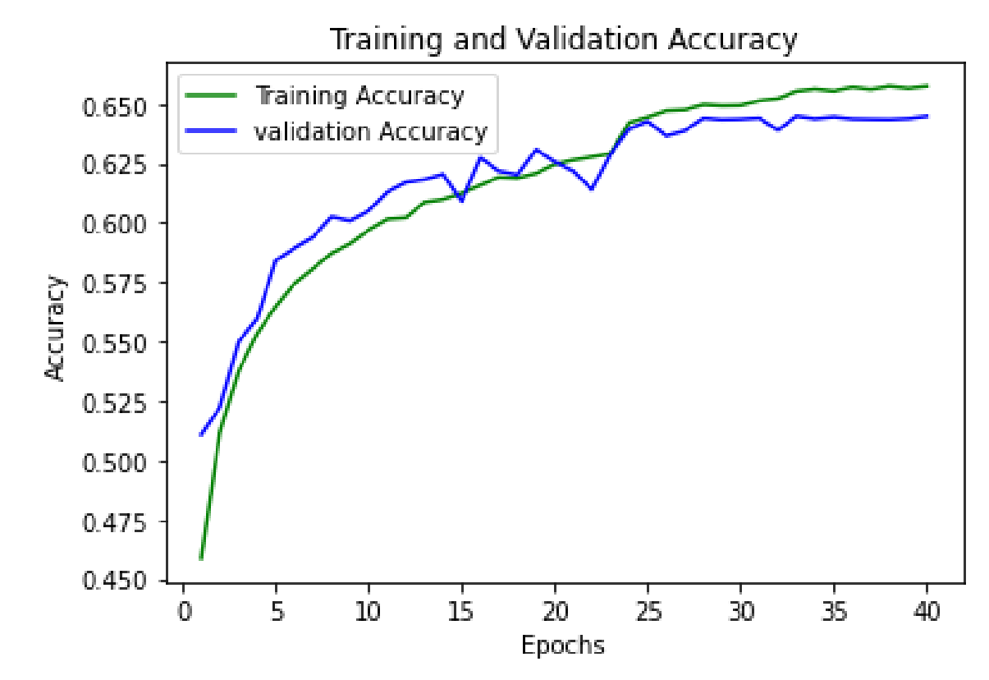


Img.4 First model loss map ploted

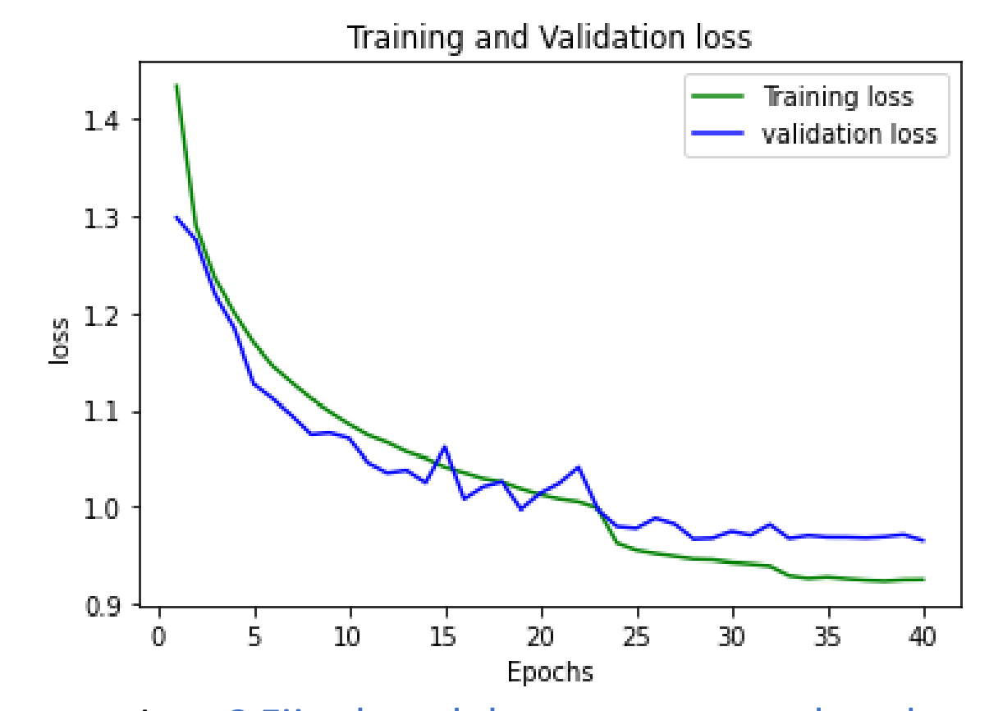
As we can see in this graphs the learning rate is smooth and it has a relatively good accuracy rate, but once it gets near 0.6 both the learning and the validation accuracy stagnate and aren't able to improve any further.

Final Implementation

In this last section I will be explaining my thought process for my first implementation of the model and how I was able to improve from the first model after seeing results I got in the graphs.
As a comparison I will also be adding the plotted graph for this model in the end of this section so we can compare them to the initial model and see that there is improvement.
Right from the beginning I knew that there were a couple problems that were causing the previous model to stagnate and also that it wasn't learning as much as I would like.
As a mean to solve the biggest problem that I had the first thing I researched was a way of stopping the model from stagnation, this was when I found ReduceLROnPlateau from the keras call-backs, I then named this lr_reduce and set the parameter to so that this would monitor the val_loss, have a patience of 4 which means that it will wait for the model not to learn anything for 4 epochs until it then changes the learning rate by 0.2 this method was the main thing that helped my model prevent stagnation and allowed it to get more accuracy.
I also decided to change the optimizer I was using from adam to adamax as in the documentation it says that adamax can sometimes be superior to adam, especially in models that use embedding.
I also decided to change the epochs from 30 to 40, for the rest I left the model was left the same as I felt confident it was doing a good job.



Img.5 Final model accuracy map ploted



Img.6 Fiinal model accuracy map ploted

As we can see in this graphs the learning rate is different and is getting a noticeably better accuracy, this is manely due to the lr_reduce.We can also see that this time the model is not stagnating anymore, it's actually just overfitting , which is something that is usually solved with data augmentation and dropouts, but since I was already using both I couldn't find a good way of improving it. This is still a very good model with room for improvement, with the needed time.
If I were to start again I think that the two main components that I would add right from the beginning would be data augmentation and lr reduce, data augmentation because it helps making a more diverse dataset and lr_reduce because it helps preventing stagnation.

Author: David Reto
URN:6647000