Classifying HiRISE Images

SAASHIV VALJEE

saashivvaljee@hotmail.co.uk

Compiled June 12, 2023

A Convolutional Neural Network, or CNN for short, is a deep learning algorithm that is able to learn directly from data. CNN's are particularly good at finding patterns in images, and are capable of doing things like recognizing and classifying sets of objects. These kind of neural networks have become staple in modern computer vision.

1. INTRODUCTION

In this review paper, I will review how to use CNN's to classify pictures from the Hi-RISE data-set which features 70 thousand pictures of mars. We will discuss the decisions made when preparing the data, the architecture of the CNN and review the results. For future, we will gloss over what could have been done to improve the performance of the model(s). Note that this paper is meant to be red alongside the notebook file itself, containing the code and more outputs.

2. FIGURES AND TABLES

2.1. Sample Figure

Figure 1 shows the network architecture.
Figure 2 shows the multi-class training and validation accuracy, and class based prediction accuracy
Figure 3 shows 20 images along with the models predictions
Figure 4 shows the binary models train accuracy, test accuracy and ROC curve

3. PRE-PROCESSING THE DATA

After importing all the necessary packages and functions, we can begin by loading in the text file containing the image titles and classes into a pandas data-frame. Data-frames are a useful data type because they allow us to easily manipulate data where we have more than just one feature. If we have a list of houses, but also want to store their worth, color, maximum occupants, etc. This would be difficult without using data-frames or something similar. By using data-frames we are able to filter all of our rows of information for specific conditions as well as alter the data-frame based on specific conditions. In that sense, they are very versatile and serve their purpose very well here.

3.1. Class Balancing

To begin, we can prevent the model from over-fitting by shuffling the data-frame and make the order of rows random. Upon observation we see that the image classes are very heavily biased towards the 'other' class, with roughly 61 thousand of the total 73 thousand images belonging to said class. A quick way to deal with this was to loop over each separate class in the data-frame and find how many rows include each class number. From here we can find which class has the least amount of images. With this information, we can randomly sample the rows of each class set and only take the same amount of rows as the maximum possible rows for the smallest class set. This will turn our severely biased data frame into a perfectly balanced data-frame, as all things should be.

3.2. Train Test Split

With our even data-frame, we can use train-test-split to convert our data frame into 3 smaller portions, each of sizes we have control over. We do this to create data-sets that we don't give our model straight away. That way the model can test itself on images it hasn't seen yet. We can finally create our image sets (using IDGs) from our 3 smaller data-frames. Note that because our model requires normalized numbers, we must specify in our image data generator that we are converting to gray-scale by re-scaling

with a factor of 1/255. We are also able to slightly alter the images randomly. This helps prevent over fitting which could be a big problem given our relatively small amount of data

3.3. Down-Scaling

Due to hardware limitations, we will have to down-scale the resolution of the images we are passing to our model. Thus, it is good practice to view our images at the down-scaled resolution. That way we can tell whether our down-scaling factor changes the resolution to something low enough for our hardware to deal with, but also high enough for the picture to be classifiable by ourselves and the model.

4. THE MODEL

The model we will use will be a sequential CNN. This type of model API is a deep learning model that allows us to create a CNN by sequentially adding on layers. After defining the model's variables (like batch-size, epochs, input shape). The general structure involves sets of: Conv2D, pooling and dropout layers, followed by a flatten layer and finishing with more dropout layers mixed with dense layers at the end of the model. The choices made while initializing these layers aswell as the architecture of the model is discussed below.

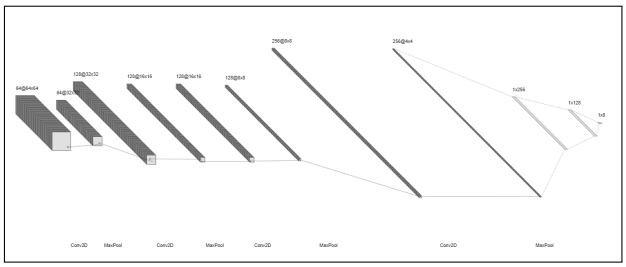

Fig. 1. Visual interpretation of multi-classification CNN. I have now realised that these diagrams are very incorrect :)

4.1. Keras Down-Scaling

As mentioned earlier, the hardware limitations warrant down-scaling the images we process. This is our first layer and helps the model train quicker and decrease the load on the computer. We can do this by writing in our resize function and specifying our new resolution for each image.

4.2. Conv2D Layers

The Conv2D layer accepts a filter count, the dimensionality of the kernel and the activation function. Optionally, you can include the input shape (it's good practice to do this at least once at the start). A Conv2D layer is a layer type for neural networks that are often used when processing images. It will apply a defined set of learn-able filters to the input image by sliding over the image in regions of whatever size the kernel is defined as. The process extracts spatial features that are expected to be seen in other images of the same class, or able to be analyzed and developed further in the following layers of the network.

We are also using the Re-LU activation function. This is because it allows our network to learn more complex relationships with the data. It is also one of our simplest and most effective options (and for those reasons, one of the most popular). The reason Re-LU is so capable is because the function introduces non linearity to the output of each neuron by applying:

$$f(x) = max(0, x) \quad \textbf{(1)}$$

For example, if the Re-LU function input is negative, the output will be 0, and if the input is positive, the input would match the output (a Piece-wise function!). What this does to the network is allow it to understand more complex relationships between input and outputs. Without it, the network would be limited to a linear decision boundary. (These boundaries are just threshold functions that determine what the neuron will output). This kind of limitation would bottleneck complicated classification tasks which is why we have chosen Re-LU as our activation function.

4.3. Max Pooling layers

Max Pooling layers are usually found directly after a Conv2D layer. They are there to reduce the spatial dimensionality of the feature map while simultaneously keeping the important information. Thus, they are included to increase the efficiency and speed at which the model learns. Max Pooling layers can be thought of as an operation that chooses the maximum value from a region of the input feature map (where the size of the region can be chosen). It will create a new feature map that retains most of the important information from its predecessor while reducing its dimensionality. The size of the feature map is determined by the kernel size of the layer. To imagine how this works, imagine an initial size 32x32 feature map that gets passed into

a max pooling layer with a 2x2 kernel size. The output feature map will have a size of 16x16 because the height and width get reduced by a factor of 2. In some senses, the way it works and why it's useful can be understood by relating it to resolution down-scaling.

4.4. Dropout Layers

The final layer within the layer set; the Dropout layer, which follows the Max pooling layer in our network. The dropout layer's purpose is to randomly discard a chosen fraction of the output features of the layer it follows during training. This forces the network to increase the variation of the type of features it's using to learn and prevents the network being too reliant on any singular feature (thus, preventing over-fitting). Dropout layers are usually set to have dropout values at relatively low numbers between 0 and 1. This is because having a relatively high number risks dropping larger and potentially more important features. Low numbers reduce the risk while retaining the efficiency and reliability of the network.

4.5. Flatten Layer

After a couple sets of our triad layer configuration, we write in a layer that flattens our outputs. Typically, the output of our layers is something along the lines of a 3D tensor with height, width and channels as its dimensions. The flatten layer is able to reshape this tensor (which holds the dimensions of our feature map and the number of filters used) into a 1D vector. This is incredibly useful to us as it lets us pass our outputs into the final layers of the network; the dense layers.

4.6. Dense Layer

The final layers of our network are dense layers, which are fully connected layers that can learn complex relationships between the input data and the output. These layers are used to produce the final output of the network, like a probability distribution(!). In our model, we have 3 dense layers, two with many neurons and one with a number of neurons equal to the number of classes in our classification problem. The larger dense layers are able to learn complex, non-linear relationships between the input and output, while the smaller dense layer is used to produce the final output of the network based on those learned relationships. The output itself is a vector, the size of which is determined by the layers amount of filters, each value represents the chance of the image belonging to each respective class. The activation function in the final dense layer of the network is set to soft-max. This is because it is able to provide probability distributions over a chosen number of classes, and makes our life easier when it comes to interpreting the outputs from our network.

4.7. Loss and Optimization Functions

To run our network, we should define loss and optimization functions. These will help us track our network's performance. When classification problems are present, the standard loss function is the Categorical Cross-Entropy loss function. This will allow us to monitor the difference between the predicted probability distribution and the true probability distribution as well as apply sensible correction multipliers to the weights when the model correctly predicts the class with a poor probability distribution vs incorrectly predicting the class with a better probability distribution.

The optimization function is set to "Adam", which is a gradient descent optimizer that helps the model converge quicker than the alternatives and also improves the results of the model in most classification cases.

5. MULTI-CLASS RESULTS

Here I will discuss the ways I have reviewed the performance of my model, as well as highlight give-aways that indicate specific points about the models training process. Note that due to the natural variance of this kind of code, each time the code is run the analysis graphs will vary but the general trends will remain.

5.1. Train Performance

Now that our model is ready to go, we fit our model to our image set created from the train data-frame, as well as check our models performance with the validation image set. We can then visualize our models performance by observing the change in accuracy and loss over each epoch. This is very important because the main way we are able to spot over-fitting is by having a train accuracy much higher than a validation accuracy. This would signal to us that the model became too adjusted to the train data and was

not able to correctly classify our validation data. To gauge the overall performance of the network, I have created a plot that shows the percentage chance of the model correctly predicting each class. This was done by adding the predictions to our data-frame and tallying which rows for each class have identical class and predicted class labels.

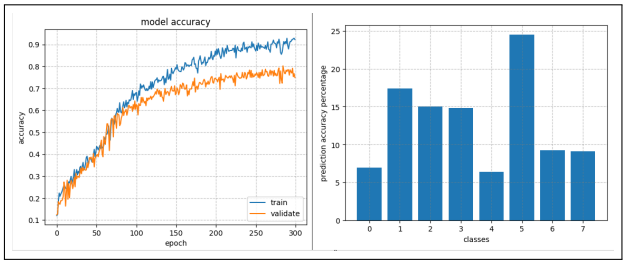

Fig. 2. Train, Validation accuracy of the model, class based prediction performance .

5.2. Prediction and Probability Testing

To further visualize the performance of the model, I have created a subplot containing 20 images, being sure to include at least one image per class. Each individual image within the subplot comes with: its respective probability distribution, its actual class, and the class the model predicted. Written both as numbers and their labels.

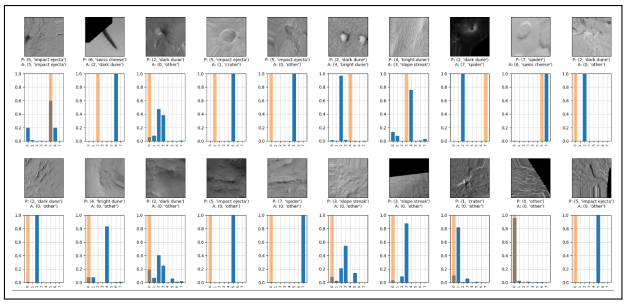

Fig. 3. predictions of 20 images, including 1 of each class. P: predicted class, A: actual class

5.3. Model Performance Review

There is alot to be said about the performance of our model, as well as the data. There are also alot of things that we could change that could improve the performance of the model. In an absolute sense, the model has not performed very well, the highest chance of us correctly identifying an image is rarely over 25%. Having more data per class would have been nice, so an alternative aspect that could improve the model is our class balancing method. We could only remove the overabundance of class 0 as opposed to every class, this could make things very different. Having less severe hardware limitations would also let us use higher resolution copies of our images, as well as a bigger network with more layers.

6. BINARY CLASSIFICATION

We have decided to create a new model that no longer classifies each image into one of the 8 respective classes. Instead, this model will classify each image based on it's a dune (whether it be a bright dune or a dark dune). To do this, we must make some changes to the way we pre-process the data aswell as make changes to our CNN model itself.

7. PRE-PROCESSING

In order for our new model to be able to class images based on whether or not it thinks they're dunes, we have to change the classes of all the images. As opposed to their classes being represented by a numerical value ranging from 0 to 7 (inclusive), they must instead be binary based (0 or 1). Note that the train-test-split and the downscaling parts of code are identical to their versions mentioned above and will not be mentioned or explained again below.

7.1. Binary Classification within Data-frame

An optional pre first step is converting the numbers in the class column into integer's from strings, originally they are strings because the image generator requires them to be so, however to make the change to binary classification, changing them to integer's makes any future modification more intuitive. Should this be done, you can later convert them back to strings in order for the image generator to work. Changing the class values to binary based values can be done very simply by finding all the rows in the main data-frame, where the class value represents something that isn't a dune, and setting it to 0. The same can be done for the numbers that represent dunes (except they'd be set to 1 instead).

8. THE MODEL

The previous model can be mostly reused for our binary classification. However, there are still some small changes we need to make in order for our model to predict whether an image is either a dune or not a dune, as opposed to classing the image into a more specific set of classes. The layers within the

model will not be described again. Instead, here I will focus on the changes made.

8.1. Filter Changes

To change the previous model to perform binary classification, the first thing we should look to change is the final layer of the model; the final Dense layer. We need to change the filter count from its initial 8, to either 1 or 2. The choice between one or 2 is largely up to us as both will perform binary classification (either way, the model will class images as belonging to one class or the other). The reason we have to make this change is the filter count in the final layer, as mentioned previously, determines how many classes we are trying to classify images into.

The reason having either a filter size of 1 or 2 works is because having a filter size of 1 will tell the network to return a singular probability per image between 0 and 1. To make these probabilities useful, we'd have to find a threshold which could be the mean probability across all images, or another statistically relevant number that says: "if the probability of the image is above the threshold, the probability is replaced with 1 and the image belongs to class 1, if the probability is lower then the threshold, it becomes 0 and it is classed into class 0".

The reason having a filter size of 2 also works is because the model will return 2 probabilities per image. The structure of the predictions becomes (per image): [probability of belonging to class 0, probability of belonging to class 1]. I have chosen to use a filter size of 2.

8.2. Activation Changes

Another change that needs to be made to the final layer is the it's activation function. We want to change it from our previous Re-LU to sigmoid. This is because sigmoid is far better suited for binary classification problems as it squishes the range of the output values to range from 0 to 1. Because of this, it provides a much more clear interpretation of what the models predicted probabilities are. Sigmoid also has a nicely defined derivative (Back-propagation is the method updates the weights of the model during training, this method involves computing the derivative of the activation function) having a well defined gradient makes this method easier and more stable. In comparison, other activation functions like Re-LU include non-differentiable points in it's function, namely at x=0. This is because, as mentioned earlier, the functions output jumps to 0 when the input is lower then 0. Graphing this, it would be clear that the gradient at this point could potentially be infinite and makes Back-propagation incredibly difficult.

9. BINARY RESULTS

9.1. Binary Train, Validation Accuracy and ROC-curve

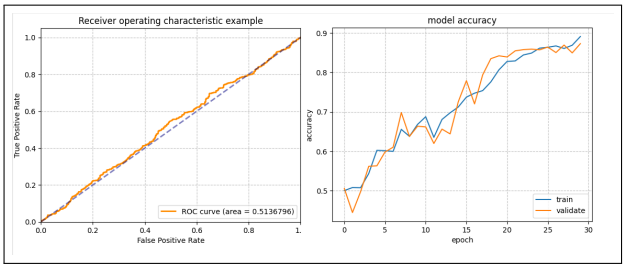

Fig. 4. Visual interpretation of multi-classification CNN. more epochs would have been nice :)

As we can see, the train and validation accuracy are very high, however from the ROC curve we can clearly see that this is a deceptive portrayal of the models effectiveness. The ROC curve essentially determines that the model has a very high false positive rate, and is about as effective as guessing ourselves. If I had more time, I would have liked to experiment more with the variables of the model (kernel and filter size, different amount of layers, different types of layers etc). It would also have been nice to have more images to work with, although this would make far less of a difference then it would have earlier. I would have liked to experiment with different network structures and different types of layers aswell. With more time for testing and fine tuning, I believe the performance of this network could be significantly improved.

If we had used 1 filter, it is likely that the overall performance of the model could have been better. At the very least, the models accuracy graph may have more accurately represented the models performance. Attempting to use 1 filter as opposed to 2 caused the model to plateau at a 0.5 accuracy 99% of the time, I was not able to recreate the 1% within the time constraints. I believe that with more time the network could be altered to avoid this behaviour.