MAIS 202 Deliverable 3

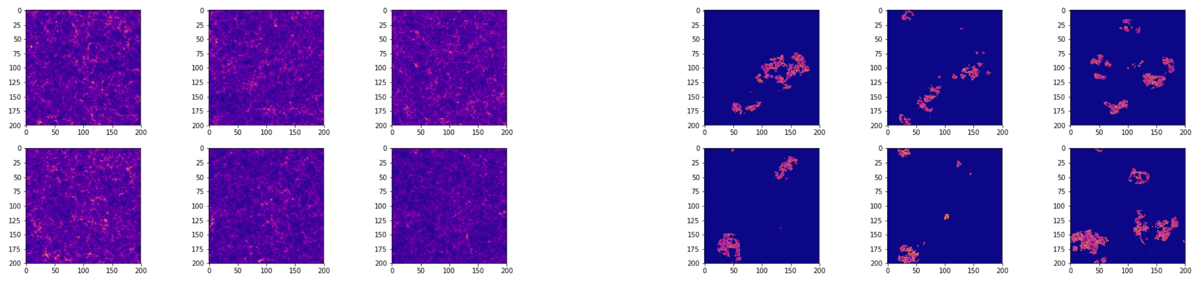
Stephen Fay

November 2019

1 Problem Statement

I have some simulated experimental data. There are 200 x 200 x 200 images that represent different objects in a 300 Mega parsec cubed region of space; I am interested in two types of these, the mass-density ones found in the “updated_smoothed_deltax” files, and the ionization maps found in the “delta_T” files. My goal is to match slices of the ionization maps with the parameters used to simulate them; it’s essentially an image classification problem where the images are classified by red-shift and by escape-fraction.

In cosmology red-shift is used to measure distance, so when the algorithm classifies the red-shift what it tells us is how far away the regions of space are. The escape fraction is the fraction of photons emitted radiation that make it out of their local area and start ionizing the intergalactic medium. Below is a picture of slices of the ionization maps (right) and slices of the mass density maps (left).



2 Pre-processing

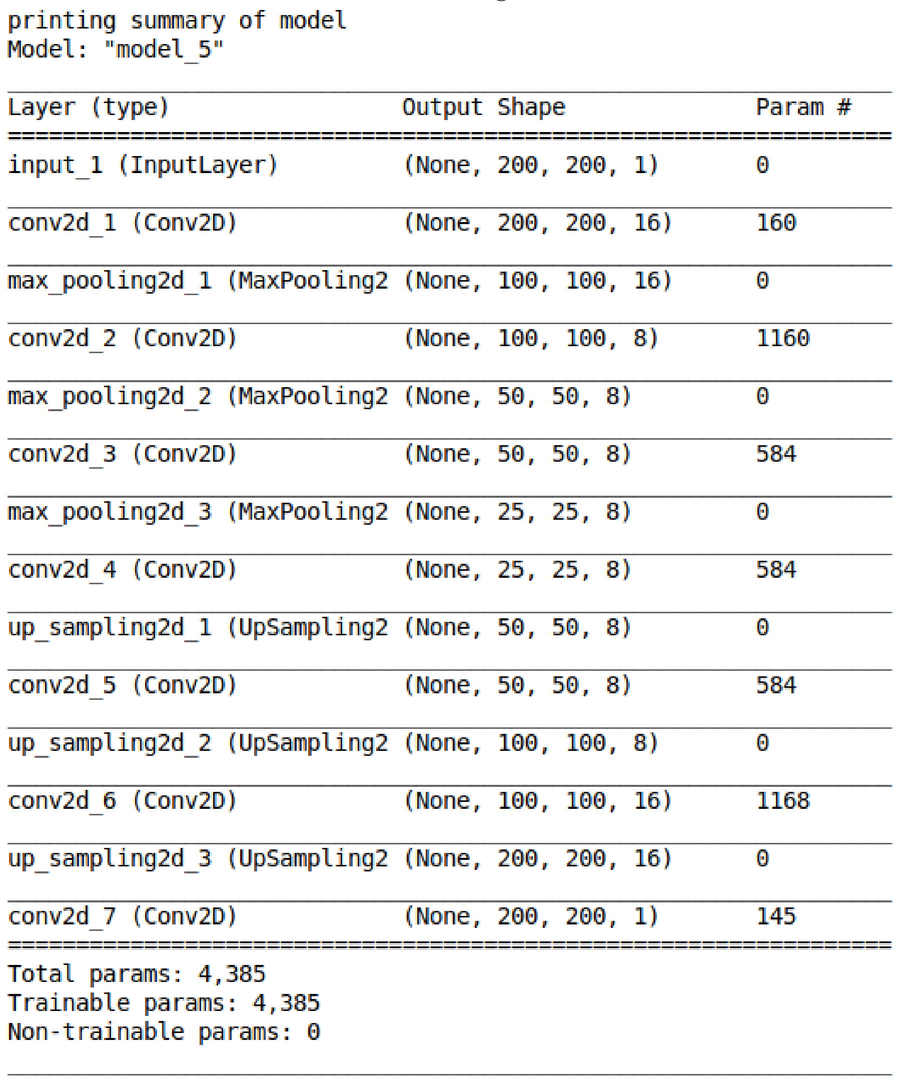
2.1 PCA 50 x 50 pixels

Initially I cut the blocks of data into 64 smaller blocks and then sliced up each of the blocks into 50 x 50 pixel images. I then sampled a few thousand of these images and applied principal component analysis to try to classify them this way. It did not work too well (see deliverable 2).

2.2 Auto-encoder

My next cunning plan was to use an auto-encoder for dimensionality reduction. Here instead of cutting the cubes up into smaller sub-cubes and then into 50 x 50 images, I went straight into slicing up the cube into 200 x 200 squares and train an auto-encoder to reduce these down from 40 000 to 5000 dimensions. Reading up I found that auto-encoders ‘fell out of fashion as we started realizing that better random weight initialization schemes were sufficient for training deep networks from scratch’ [see references at bottom] (in 2012 - 2014) for this kind of image classification problem. Another reason I wanted the Auto-encoder was because of the fantastic failure of classifying the 200 by 200 slices by their escape fraction.

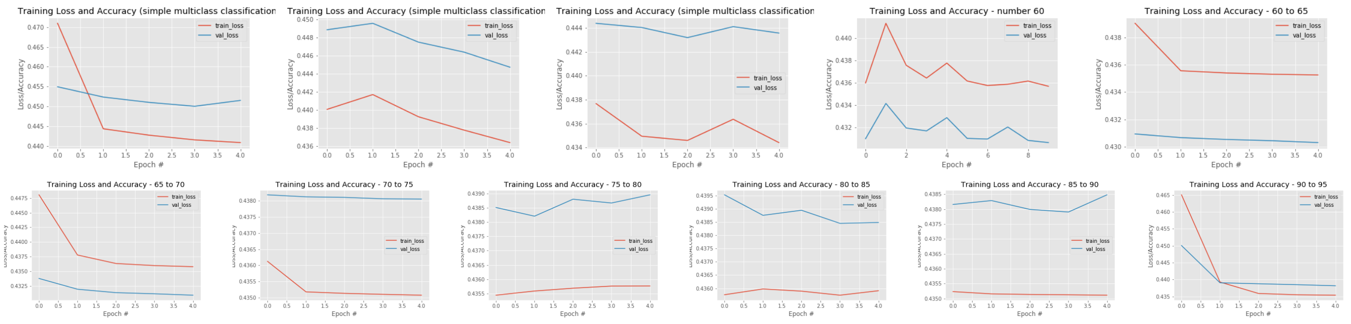
Training the Auto-encoder was tough for my laptop, so I did the training in bite-sizes of 5 epochs at a time, these took about 25 mins to train each; I did not have the forward thinking to save all the parameters when I started training so the plot for this is awful (see below). First I fetched the data and sliced it up with a google colab notebook, saved 6528 200 x 200 slices into a numpy array. I fed this through my keras auto-encoder with the following architecture



```
printing summary of model
Model: "model_5"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         (None, 200, 200, 1)       0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 200, 200, 16)      160
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 100, 100, 16)      0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 100, 100, 8)       1160
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 50, 50, 8)         0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 50, 50, 8)         584
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 25, 25, 8)         0
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 25, 25, 8)         584
_________________________________________________________________
up_sampling2d_1 (UpSampling2 (None, 50, 50, 8)         0
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 50, 50, 8)         584
_________________________________________________________________
up_sampling2d_2 (UpSampling2 (None, 100, 100, 8)       0
_________________________________________________________________
conv2d_6 (Conv2D)            (None, 100, 100, 16)      1168
_________________________________________________________________
up_sampling2d_3 (UpSampling2 (None, 200, 200, 16)      0
_________________________________________________________________
conv2d_7 (Conv2D)            (None, 200, 200, 1)       145
=================================================================
Total params: 4,385
Trainable params: 4,385
Non-trainable params: 0
_________________________________________________________________
```

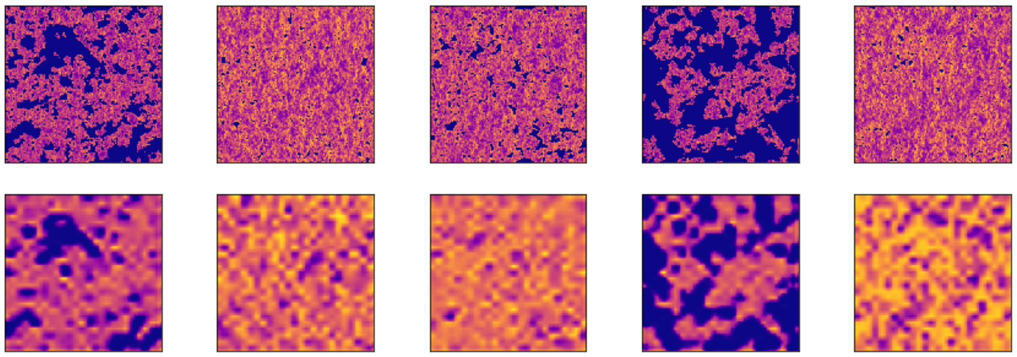
The dimensions are reduced from 200 x 200 = 40 000 to 25 x 25 x 8 = 5000 dimensions. Also this is just a conjecture but the fact that our image goes from 2-d to 3-d means that some of the information might be encoded in the block structure, which is good - it means we get more bang for our buck. So now before training each network I feed the images through the auto-encoder.

Here are some plots of the loss function’s progress as it went though training, sadly I don’t have anything before the 40’th epoch.



If you are observant you notice that the validation loss jumps under the training loss, here is where I have to admit my sin : I did not use the same training and validation set for the whole way though. What happened is at the beginning I shuffle and then split everything, so every time I stopped training and opened up the jupyter notebook once again I was essentially mixing the training and validation sets; this is bad because it means that if my model starts to over-fit I won't be able to detect it with as much accuracy (or perhaps at all). Note: The way I trained this was very experimental, I often fiddled with the training rates and optimizers.

In the end the auto-encoder gave results that something like this (with error of ≈ 0.1), euclidean distance.

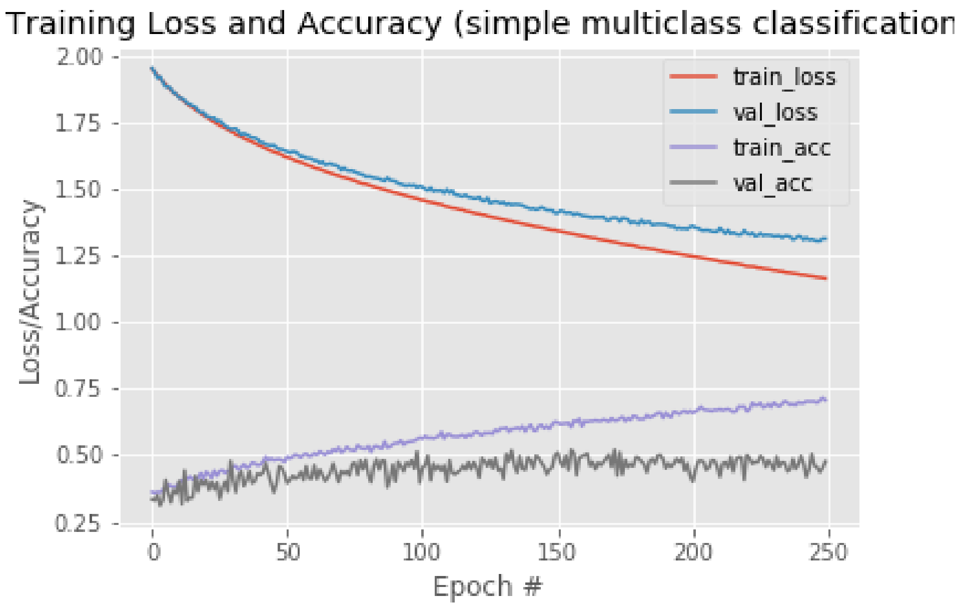


3 Classification Method

3.1 Neural nets on the 50 x 50 images - classifying one single parameter

3.1.1 Classifying Red-shift

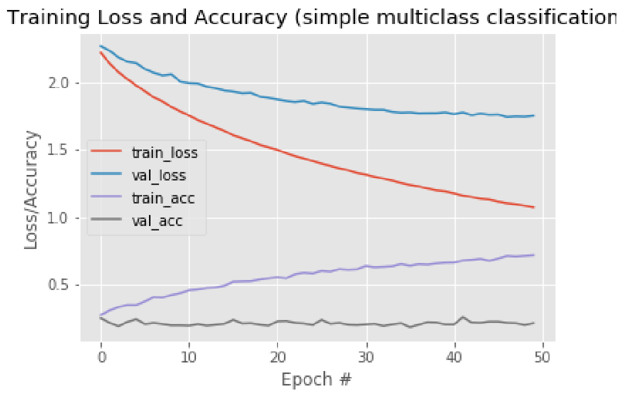
The first neural-net classification attempt I made directly on the 50 x 50 images, the training set I used here was the data where the escape fraction parameter was fixed at 0.070 - the reason I did this simpler classification was kind-of to test the waters to gage what kind of training would be required for the more general classification. I tried three slightly differing architectures for this one varying the number of hidden layers and their size; the most successful was a single hidden-layered fully connected neural network (2500 flattened image input - 350 - 17 classes), this one obtained a 48% accuracy validation accuracy after 250 epochs of training. This is not bad considering that the spacial information was largely lost by the flattening and that some more information may have been lost by the reduction in size of the data. Also I ran this locally, my computer has limited resources and I wanted something that would be easy to train on it.

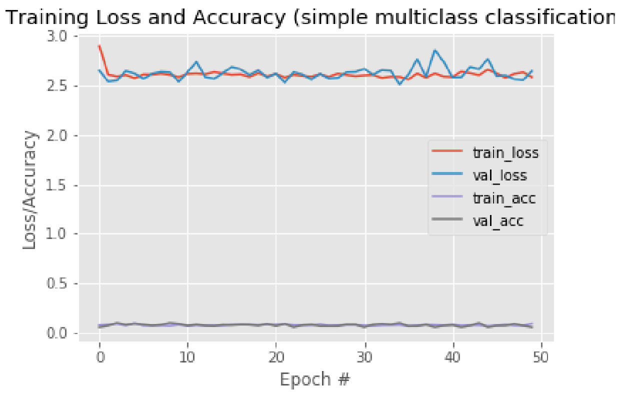


So the network above classifies the images into 17 classes, with different red-shift values.

3.1.2 Classifying Esc-Frac

My next step was an attempt to classify images by their the escape fraction parameter, to do this I used only images with a red-shift value of 0.750, I did this because the two parameters are most likely *not* independent variables. Here I tried stepping it up one notch and classifying directly the 200 x 200 images. This was a miserable failure.



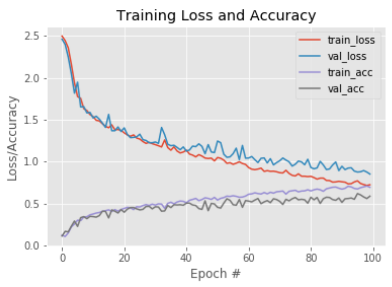


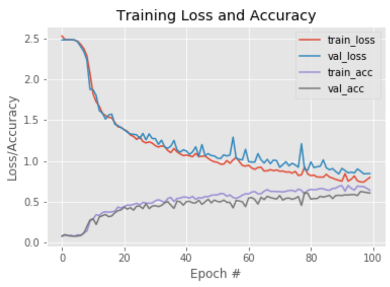
The plot on the left hand side is using the SGD optimizer, classic case of over-fitting. The right hand side is using the ‘adam’ optimizer. They both performed within a couple percentiles of random guessing. Even so it’s interesting to note that the sgd over-fitted whereas the adams just went nowhere.

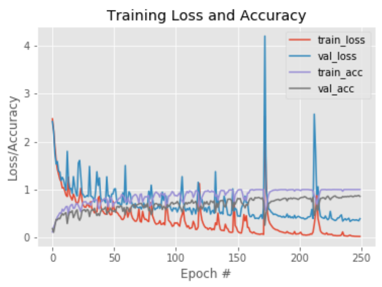
3.2 CNN applied on the encoded images.

3.2.1 Classifying ESC-FRAC for fixed Red-shift - with encoder

Here is a comparison of different results I tried out. It was a bit hit and miss.



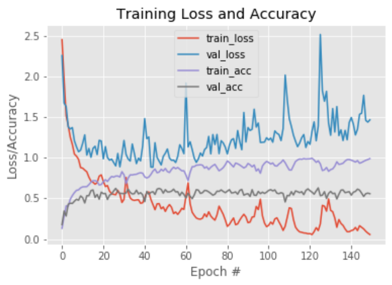




The above plots are accuracy / loss plots of the networks' progress during training - different architectures trained for different lengths. The right-most is probably over-fitting even though it is the best performing network.

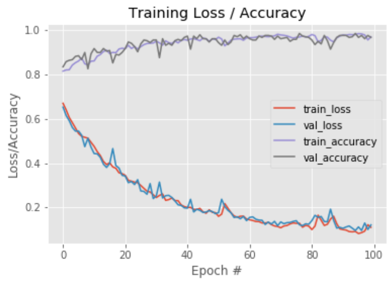
3.2.2 Classifying ESC-FRAC for fixed Red-shift - without auto-encoder

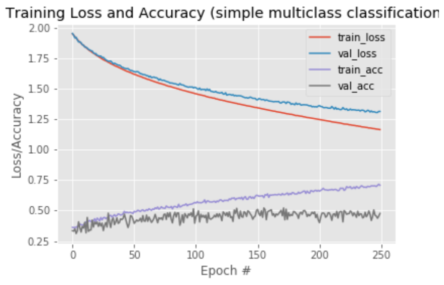
Without auto-encoder the results were not so good.



3.2.3 Classifying Red-shift for fixed ESC FRAC - with encoder

It turns out that the convolutional neural network + encoder (left) had a much easier time classifying the redshift that the fully connected neural net without the encoder which was the baseline (right), it also converged much quicker.





Without auto-encoder

3.2.4 Classifying Red-shift for general ESC FRAC - Architecture and performance

By now it was clear that the auto-encoders were a good idea, specially given the limitations of my machine. The networks performed well on the general redshift prediction.

3.2.5 Classifying encoded ESC-FRAC for fixed Red-shift - Architecture and performance

with auto encoder

Without auto-encoder

4 Notes

There is something in the slides about convolution layers where you learn which kernels are the best.

You can use a pre-trained models on a few pictures, you don't want to train the first couple layers because they are low level feature detectors and you can use them.

Pooling layers (max pooling, average pooling) - if you want to classify stuff, pooling layers are good and useful!

For notes on depth, stride and padding.

- Depth - how bay kernels you apply and feed into network
- Stride - How many strides you step over
- Padding - just the zeros you add on the edges so that you convolve over everything (two types of padding in keras : same = one pixel boarder ; valid = no padding)

5 Future Work

I started this project with the idea that I wanted to do something useful; although I learned a-lot, I did not achieve what I had wanted to. Future work on this would be to train a network to generate it's own intensity map. Another ambition I had in this project was to match the mass density maps with the ionization maps.

6 References

1. https://academic.oup.com/mnras/article-abstract/483/2/2524/5228756?redirectedFrom=fulltext - a free pdf of this article can be found here https://arxiv.org/pdf/1807.03317.pdf
2. Had a look at Samuel Gagnon's research project as a resource (uses same / similar? data that I use) - https://github.com/samgagnon/remove-wedge
3. https://medium.com/@pallawi.ds/ai-starter-train-and-test-your-first-neural-network-classifier-in-keras-from-scratch-b6a5f3b3ebc4
4. The keras documentation was my friend the whole way though. https://keras.io/optimizers/
5. more keras documentation https://keras.io/losses/

6. Also slack was my friend https://stackoverflow.com/questions/52271644/extract-encoder-and-decoder-from-trained-autoencoder

7. Guid to Sequential Model : https://keras.io/getting-started/sequential-model-guide/

8. Guid for implementing keras CNN : https://towardsdatascience.com/image-recognition-with-keras-convolutional-neural-networks-e2af10a10114

9. How padding works https://stackoverflow.com/questions/45013060/how-to-do-zero-padding-in-keras-conv-layer45013181