UNIVERSITY COLLEGE CORK

BSC COMPUTER SCIENCE

FINAL YEAR PROJECT

A Deep Learning Regression Model to Predict Galaxy Types Using The Galaxy Zoo GZ2 Data Set

Author:
Hassan BAKER

Supervisor:
Dr. Gregory PROVAN

April 15, 2018

Contents

List of Figures

Abstract

A Deep Learning Regression Model to Predict Galaxy Types Using The Galaxy Zoo GZ2 Data Set

Hassan Baker

This project studies various aspects of deep learning, by using empirical comparisons. This involves a study of three learning optimization algorithms; Stochastic Gradient Descent (SGD), SGD with Nesterov momentum, and Adam. This is done across various learning rates to find an optimal combination. Furthermore, this project studies activation function selection, comparing Sigmoid to ReLu, and ReLu to Maxout finding that a Maxout-Sigmoid combination performs far greater for this particular data set. These studies are accomplished by building a deep convolutional neural network to form a regression model using TensorFlow. This study uses the Kaggle Galaxy Zoo Challenge *GZ2* data set, containing images of galaxies, and probability distribution solutions.

Chapter 1

Introduction

1.1 Galaxy Zoo

Galaxy Zoo is one of the world's largest citizen science projects in the world. The project collects images of galaxies and invites the public to classify the images into set classes. The classification process is done by giving the citizens a set of questions to answer regarding the morphology of the galaxies in the images. The public's assessment of the data is later aggregated and cleaned. The project managed to get more than 50 million classifications in its first year using images of galaxies taken from the Sloan Digital Sky Survey (SDSS).

Galaxy Zoo 2 (GZ2) refers to the second phase of the project. The images in the *GZ2* data set compromise of the brightest 25% from the SDSS [1]. This dataset contains more than 300,000 images. The public was asked to classify the galaxies in the images into 11 classes, which contain sub classes. This accumulates to 37 possible classes altogether. The galaxies were classified by the public's answering of questions like "Is the galaxy smooth and rounded, with no sign of a disk?" These questions form a decision tree of 37 possibilities, which are the labels of the dataset.

1.2 The Galaxy Challenge

The Galaxy Zoo Galaxy Challenge was launched on the 20th of December 2013 and hosted on Kaggle. The competition provided the set of *GZ2* data. A set of probability distributions relating to what class the public judges each image to belong to is also given as a CSV files. The objective of the competition is to reproduce a CSV file of probability distributions for a test set, that reflects the public's assessment of these images. The majority of the solutions produced for

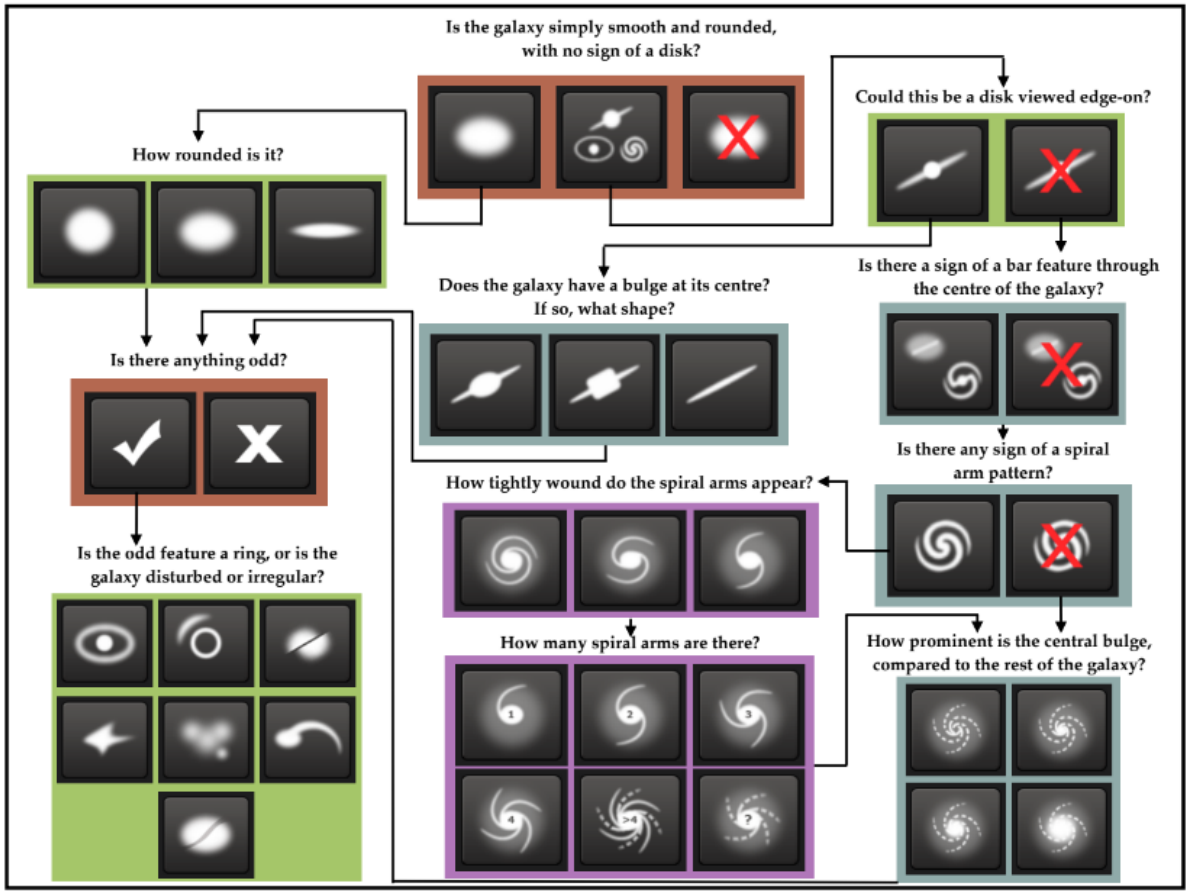


Figure 1. Flowchart of the classification tasks for GZ2, beginning at the top centre. Tasks are colour-coded by their relative depths in the decision tree. Tasks outlined in brown are asked of every galaxy. Tasks outlined in green, blue, and purple are (respectively) one, two or three steps below branching points in the decision tree. Table 2 describes the responses that correspond to the icons in this diagram.

Figure 1.1: Decision tree of galaxy labels in the GZ2 data set taken from Willett et al [1].

this competition involved machine learning, and a majority of that majority relied heavily on convolutional neural networks.

Given the large set of image data, convolutional neural networks are an obvious choice as their use is proven to be highly effective in image recognition and analysis. However, convolutional neural network solutions, like many machine learning solutions, contain a level of complexity that originate from just how varying the model can be. One can refine at each step of the process to yield better accuracy or lower error, but in many ways, one can also over refine.

1.3 Neural Networks

Neural Networks are constructed by layers that consist of parallel neurons (represented by figure 1.2) which contain weights and biases. The data is inputted into the first layer. At each neuron, the input is multiplied by the weight, and a bias is added to the solution. The output of each neuron is then put through a non-linear function. This is so as to introduce non-linearity in the network, as most models can be represented by non-linear functions. Each output is then passed

into the following layer, abiding by the connection scheme i.e. in a fully connected network, each neuron passes it's output to every other neuron in the following layer as shown in figure 1.3.

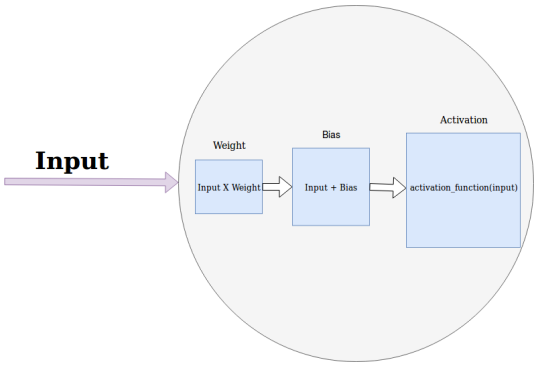


Figure 1.2: A neuron in a neural network which multiplies an input with a weight, adds a bias, and carrys out a non-linear activation function.

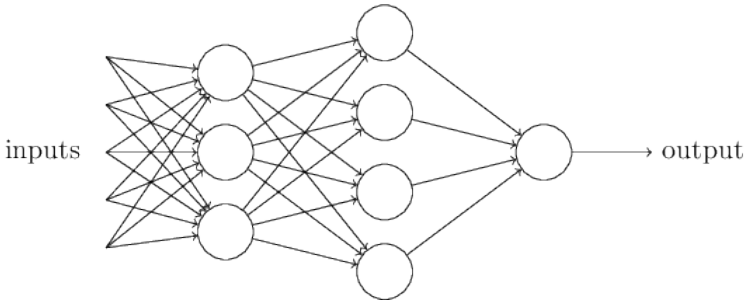


Figure 1.3: A fully connected network whereby all neurons in a $layer_i$ are connected to all neurons in $layer_{i+1}$.

The final layer in a neural network outputs the predictions, given an input. So as to give accurate predictions, the network is trained. Training involves the use of a labeled data set. By putting a set of training inputs through the network, and measuring the difference between what the network predicts and what the actual values it should predict are, one can calculate the loss. The loss is minimized throughout each training iteration using the back-propagation algorithm [3]. Back-propagation updates the weight values in relation to the partial deravitive of the gradient of the loss. This allows the model to predict better with repeated training.

1.4 Convolutional Neural Networks

Convolutional neural networks methods are used widely for image recognition, sound recognition, natural language processing [4], and a whole cohort of other applications.

Convolutional neural networks are currently one of the most popular machine learning techniques. They are powerful machine learning methods as they rely on breaking down the image into smaller convolutions and detects whether the learned patterns exist in these convolutions. The order and location of where these patterns occur is less important than in an

ordinary feed-forward neural network, which is actually beneficial for particular tasks, such as image recognition.

Convolutional neural networks are more efficient to train and need less data than regular feed-forward networks, even though their theoretical maximum accuracy is not as good. This is because the convolutional layers that make them quicker to train than standard neural networks. These layers convolute the data into separate learning phases. An earlier layer would tend to take on the job of edge detection, and pass that information to the following layer. As you move through the layers, the learned filter should get more complex, until it eventually recognizes whole objects. This is because each individual layer can be seen as a separate network which is optimized, through training, to produce the easiest solution for the following layer to work with.

This feature of convolutional neural networks allows for useful methods like transfer learning, whereby one can take the convolutional layers of one already trained network, attach a new fully connected layer to it, and train it to recognize something else entirely, far quicker than it would to train a whole new network.

Chapter 2

Analysis

2.1 Goals

The main goal of this project is to develop a robust convolutional neural network that can produce a solution file for the test set with a low level of error, using Root Mean Squared Error (RMSE) for error estimation.

This model will be built incrementally by creating an initial model, and using it to study some key components of deep learning methods and architecture. Taking the results of these studies, the model will be improved to produce solutions with lower error. As the data is from a Kaggle competition from four years ago, there are already solutions available on the internet. The solutions examined in this study are Team 6789 [5], Fang-Chieh Chou [6], and most notably Sander Dieleman [2]. Hence this project focuses on studying some of the methods used in those solutions, as well as alternatives put forward by the student.The key areas of study in this project will focus on data preparation and processing, deep learning optimization algorithms, activation function comparisons, and hyper-parameter searching.

This project will aim to validate what the most fitting learning optimization algorithm to use, comparing Stochastic Gradient Descent (SGD), SGD with Nesterov Momentum, and Adam Optimization. This project will also study the effects of using the ReLu function against the use of the Sigmoid function in the final layer of the network. This project also aims to study the effects of the Maxout unit, described in Goodfellow et al [7]. This project aims

As a tangential goal, the student has undertaken this project to further their knowledge in deep learning methods and principles in a practical sense.

2.2 Methodology

So as to empirically ensure improvements, a methodology will be adhered to throughout this project. Initially the data will be analyzed and a suitable convolutional neural network will be designed to fit the data.

To ensure the the network designed works as it is intended to, the network will be put through a validation stage, whereby it is trained for a small amount of time, on 90% of the training set, allowing the remaining 10% to be used for validation. Throughout the validation stage, the training and validation errors and losses are taken.

On successful validation, we will move to a testing stage, whereby the network will then be trained for a longer period on the entire training set. Throughout the training solution files will be produced to get the test error.

After this training period, the network will be analysed as the training error, test error, and the solutions produced should offer insight as to how to improve on the current network.

Given the insights gained from training, a new network will be then be created to better fit the data. The new network will be put through validation and testing.

This process will be repeated throughout this project.

It's worth noting that since access to the training set's labels is not available, this report will mainly focus on the error values, rather than loss values. However, throughout development, loss had proven to be quite effective for ensuring that the models are behaving as expected.

2.3 Tools

2.3.1 Hardware

The primary machine used in this project is one of the computer science department's Alienware GPU workstations, containing an Intel i7-6800K hexacore CPU and an Nvidia GTX 1080 GPU. However, so as to run multiple experiments at once, one of the UCC Netsoc (Networking and Technology Society) servers was used, leela.netsoc.co, which contains an Intel Xeon E5-2630.

2.3.2 TensorFlow

The main framework used is TensorFlow. TensorFlow offers a very intuitive method of for developing neural networks, with many examples available. It contains plenty of up to date features and a large support base. A key reason for the selection of TensorFlow is that it has a Python API as teh student is very comfortable with Python. However, the most vital reason for the selection of TensorFlow is TensorBoard. It's highly difficult to debug neural networks. For the most part, the developer spends a majority of the time looking at graphs. The TensorBoard graphs are not only highly intuitive to code in TensorFlow, but also offer a lot more than alternatives like matplotlib. Moreover, TensorBoard simply offers a lot more, for a lot less work. For example,

on the scalar view, there is a smoothing function which applies a linear filter to graphs so as to reduce noise as with gradient descent methods there tends to be a lot of rapid oscillations, which when plotted exactly can look like a mess. It is worth noting that throughout this project the reported error values used are after a smoothing factor of 0.85. This makes the data more concise and better reflects the models' performences.

The student has however experienced some negative aspects of working with TensorFlow. Although there are many examples and StackOverflow answers available, the documentation is quite lacking. The documentation tends to assume that you know what a certain method or object does and how it does it. There are cases where on the documentation page, rather than giving meaningul information, only a citation of the paper which the implementation replicates is described. Another example of bad documentation is the session storing feature, which allows the developer to save and restore a TensorFlow graph and session. There is simply too many ways of implementing this, and each offer different purposes with little documentation on what the purposes are. Luckily there are StackOverflow examples for this. Moreover, an even greater issue experience with TensorFlow is the inability to write custom activation functions, as there is currently no Python implementation for updating gradients throughout a TensorFlow graph.

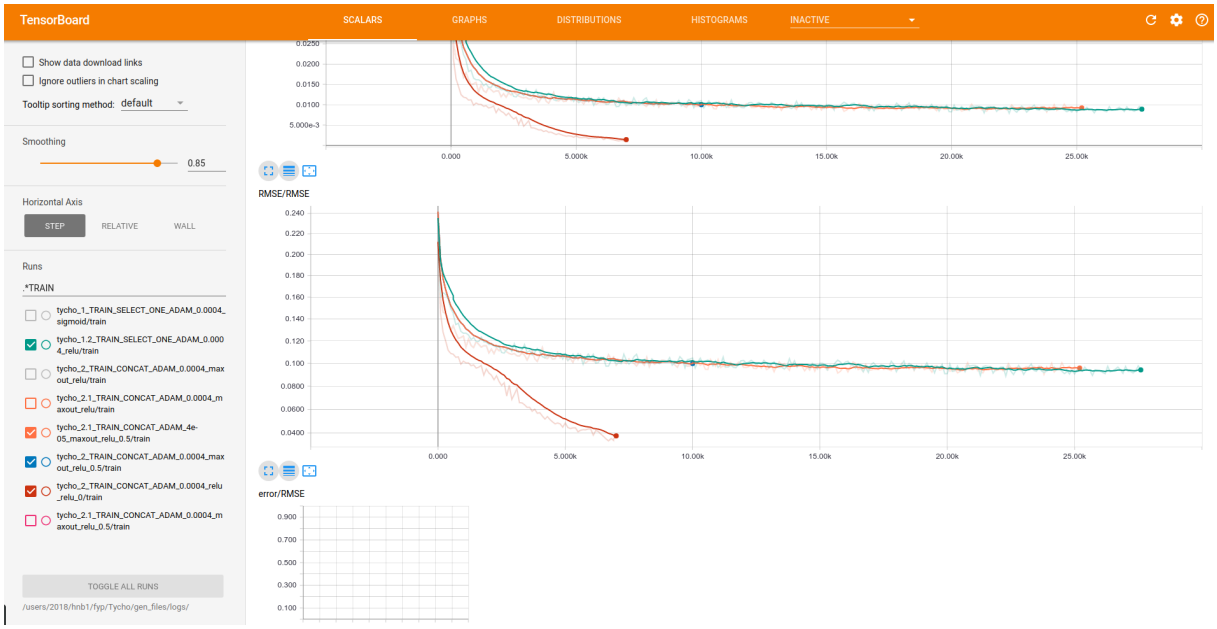


Figure 2.1: Screenshot of TensorBoard

2.3.3 IDE: Jupyter Notebook/PyCharm

Initially Jupyter Notebook was used to develop the codebase as it offers a great documentation feature, which is very handy for machine learning projects. However many issues started to arise as the codebase got more complex. For instance, when a cell in a Jupyter Notebook is run, it is automatically cached in main memory. This is not only inneffecient when working with a large dataset, but also problematic when working with TensorFlow, as when there is multiple Jupyter

Notebooks running, conflicts between different TensorFlow graphs arise which make debugging harder than need be. This resulted in a lot of kernel restarts. Moreover, it was found that Jupyter Notebook simply encouraged bad coding habits, as it's easier to get in the habit of writing small scripts in the cells of a Jupyter Notebook, rather modularising a code base for reuse. Hence the student switched to using the PyCharm IDE, which does not contain any of these issues.

2.4 The Data

The *GZ2* dataset consists of 61,578 training images, 79,975 test images, a CSV file consisting of the 37 probability distributions for each example in the training set, as well as CSV files containing all one ones, all zeros and central pixel benchmarks for comparison when training. The test set solutions are not provided, instead one is required to generate a CSV file with their own solutions and submit them. Seeing as this is a large dataset, *holdout* validation was used instead of *k-fold*, as *k-fold* would otherwise be far too time consuming. A validation set was created from 10% of the training set. This 10% portion was moved into its own directory and was not used for analysis or training. This allowed for *holdout* validation throughout analysis and training.

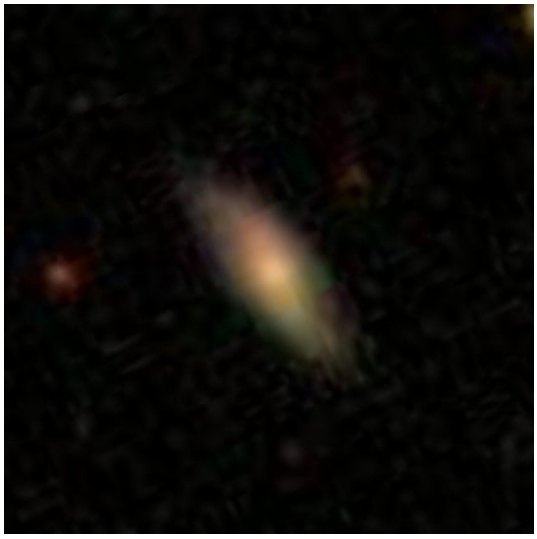


Figure 2.2: An image of a galaxy from the GZ2 data set.

2.4.1 Cropping

The images in *GZ2* are already centered and are 424×424 in size. It does not seem feasible to fit every pixel of an example as a feature into the network as that would result in an input layer of shape $[424, 424, 3]$. This is far too big and would slow down learning greatly. The galaxies in the images are surrounded with black space. Sometimes these images contain some other stellar objects that have no apparent significance for the model. The black space and the stellar objects

make for plenty of noise. Therefore it is only reasonable to uniformly crop these images so as to cut off a majority of the noise but still keep every galaxy intact.

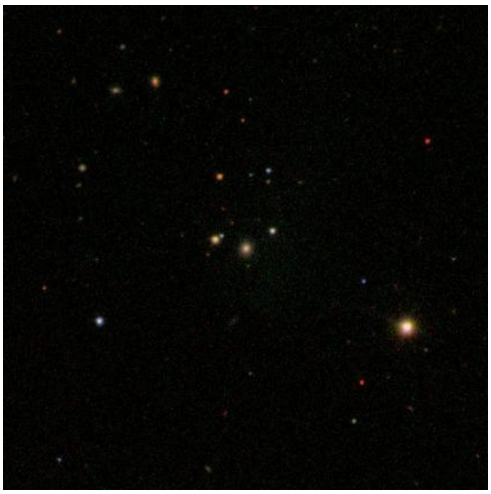
The winning solution, Sander Dieleman [2] cropped the images to a standard size of 207×207. However, there is no indication as to how the value 207×207 was selected in this solution.

To investigate this, using a Python script, all the images in the training set were analysed to validate this crop amount. The most important and active parts of the image are at the centre, accounting for most of the brightness in each image. Whereas the more dim areas are more likely to contain noise. Using brightness as a measure of activity, we can measure how much we have to crop each image to keep a selected threshold k of activity, k being the percentage of brightness in an image. Hence, each image's colour channels are normalized to fit a the range $0 < i < 1$, and are averaged into 1 channel (gray-scale).

Each individual image is then iteratively cropped towards the centre. In each iteration a 1 pixel wide perimeter was cropped out of the image. k', the percentage of brightness of an image was taken at each iteration. Once k' falls below k, the amount of iterations carried out for that particular image are appended to a list Z.

Once the whole training set is analysed, the mean of the values in Z is taken. Using trivial geometry, one can then determine the optimal crop amount for a mean of k brightness in the images. The mean value of z, denoted by z amounts to how many pixels down the diagonal of the image to crop from. Given the value H, the length of the diagonal from one corner of the original image to the opposite corner. This means that the length of the diagonal from one corner to the opposite corner of a cropped image, h is found using $h = H - 2z$. Furthermore, seeing as the crop results in square dimensions, taking the dimension values as (x, x), x can be found using Pythagoras.

In this analysis, k was selected to be 90%. This yielded a mean crop amount of 218×218, which is only 11×11 more than the 207×207 amount selected in Dieleman's solution [2], and 18×18 more than the 200×200 amount used by Team 6789 [5]. Which leads to a conclusion that the crop amount in these solutions was either selected with good intuition, which is very possible, or else selected in a similar fashion with an extra few pixels kept for extra measure so as to make sure that outliers do not impact the results. To verify this, along with the mean crop amount, the maximum and minimum crop amounts for $k = 90\%$ were also taken. The maximum crop amount was 398×398, and the minimum was a crop amount of 0×0. It is easy to see how the minimum and maximum could skew this investigation by just looking at the image itself.



(a) The image requiring maximum crop, because the galaxy is very small and is easily confused with other stellar objects.

(b) The image that requires minimum crop due to the very bright stellar object is in the top right corner.

Figure 2.3: Outlier images found in GZ2

So as to keep to a middle-ground, and to stick to comparable measure, the crop amount that was selected in this study is 207×207, the same used in Dieleman's solution [2].

2.4.2 Down-scaling

Cropping alone, still results in a very input large size as it would result in an input layer with 128,547 parameters, which will still stagnate the learning time, as all these parameters will have to be updated. Hence, the images were down-scaled by a factor of 1/3, as seen in Dieleman's solution [2]. This results in the images taking the shape of $[69, 69, 3]$.

2.4.3 Data Augmentation

The images in the *GZ2* data set contain exploitable variances. For instance, as they are images of galaxies, there is no up or down, or left or right. They are of uniform shapes, meaning that it is relatively easy to extract new features out of the given features.

Inspired by the augmentation section in Dieleman's solution [2], a class for augmenting images was written. This class contains methods for cropping, down-scaling, rotation, and flipping. Uusing this class, one can can augment an image to extract 16 examples out of each individual image.

The augmentation process is thus. The 207×207 crop is applied to each image, a 45 degree rotation of that same image, as well as a horizontal flip of the unaltered image, and the rotated one. This produces four images. This is illustrated in figure 2.4.

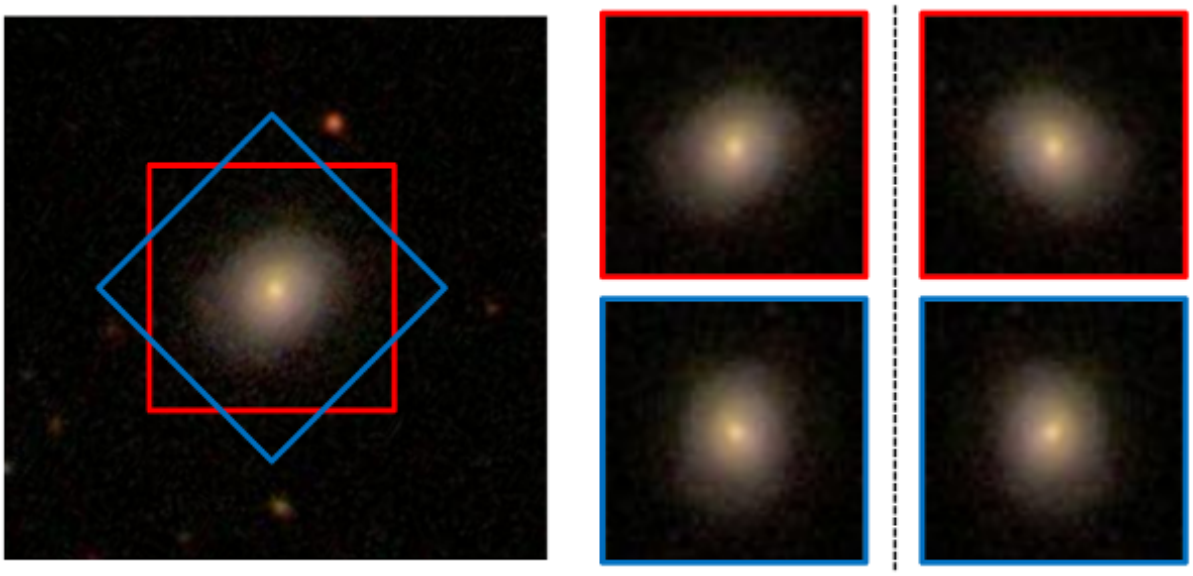


Figure 2.4: Visualization of how 4 features are extracted from one image. Figure taken from Sander Dieleman's blog [2]

Furthermore, four crops of 45×45 overlapping parts are taken from each corner of the images extracted, which produces 16 images in total. The 16 images are then rotated to ensure that all extracted images are oriented so that the galaxy is at the bottom right.

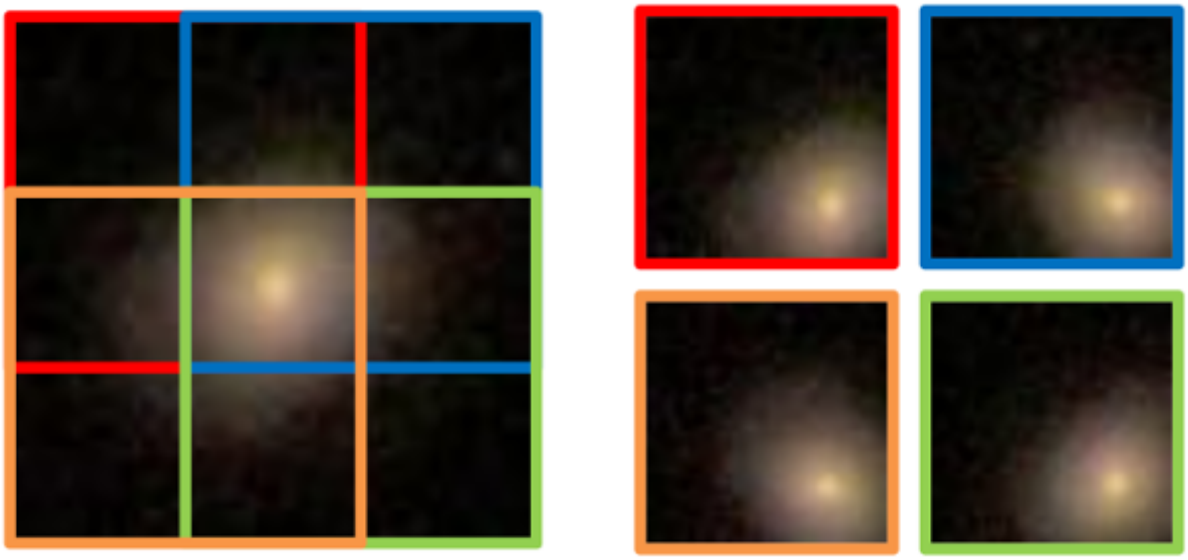


Figure 2.5: Visualization of how the four overlapping parts are extracted from each images created in 2.4. Figure taken from Sander Dieleman's blog [2]

These extracted images overlap by design. This is so as to increase parameter sharing in the model. The bottom right of all extracted images contains the centre of the galaxy, however, the top right contains black space, or noise. Due to this augmentation, the values of the noisy sections are generally different in each extracted image, hence this process makes it more likely for a model to be more active with the bottom right sections.

Chapter 3

The Network: Tycho1

The initial convolutional neural network, named Tycho1 is a modified version of the model described in Dieleman's blog [2]. It contains four convolutional layers, and three dense layers. Tycho one differs from Dieleman [2] in the dense layers, and does not perform a concatenation on augmented image segments after they have passed the convolutional layers. This is because Tycho1 is designed to train faster, so as to validate the studies regarding learning optimization algorithms, learning rate searching, and activation function comparisons.

3.1 The Architecture

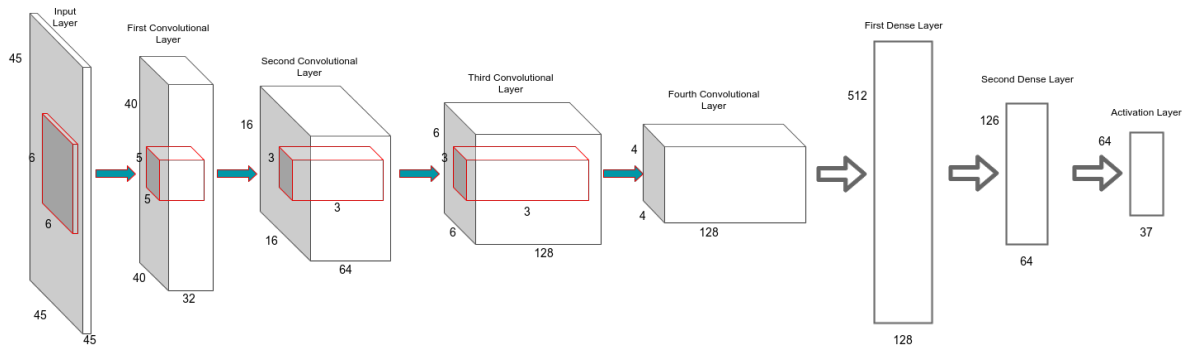


Figure 3.1: Diagram of the Tycho1 network

The input layer takes inputs of shape $[BATCH\_SIZE, 45, 45, 3]$. The first element is the amount of images in a given batch, the second and third represent the size of each image, and the last element is the number of channels. This is then fed into the convolutional segment.

The network consists of four convolutional layers. The first convolutional layer contains a filter of size 6, and an output shape of 32. The second layer contains a filter of size 5 and an output shape of 64. The third and fourth both have a filter size of 3, and an output size of 128. The first, second, and fourth layers are followed by 2×2 max-pooling layers. Once the data passes through the last convolutional layer, it is flattened into a shape of [some shape I can't remember rn].

The flattened output the enters the dense layers. The weights in the first dense layer and the second dense layer are of shapes $[512, 128]$ and $[128, 64]$, respectively.

All layers, bar the last contain a Rectified Linear Unit (ReLu) as the activation function.

$$f(x) = max(x, 0) \tag{3.1}$$

The ReLu activation function was chosen as it is extremely useful for object recognition [8]. A distinct feature of ReLu activation is that it has sparse activity allowing for mostly meaningful activations to be passed onto the next layer. This decreases training time, and makes activations far more efficient.

The weights in all layers, dense and convolutional, except the last, are initialized to be in a normal distribution of standard deviation 0.01, and a mean of 0. This restricts the weights to a range of $-0.01 < W < 0.01$.

The biases in all layers, including the last, are set to be a constant of 0.1. This was originally set to 0.0, but after some testing, 0.0 proved to be ineffective and resulted in dead units. Hence 0.1 is used instead.

The last layer is also a dense layer. This is of shape [BATCH_SIZE, NUM_LABELS].This shape corresponds to the amount of images to fed into the network, and the number of values/labels to predict. With this data, there are 37 labels. The weights are initialized to be in a normal distribution of standard deviation 0.001, and a mean of 0.

As described by Dieleman [2] and Fang-Chieh [6], a modified version of ReLu or Softmax, with a normalization segment was initially considered. However this is where TensorFlow has it's drawbacks, as there is no way to write custom activation functions using the TensorFlow Python API.

As an alternative, Sigmoid activation 3.2 is chosen for the last layer. This seems like the best fit as it squashes all values between a range of $0 < y' < 1$.

$$h(x) = \frac{1}{1 + e^{-}x} \tag{3.2}$$

The error evaluation function used for the Kaggle competition is Root Mean Squared Error (RMSE), hence, for measuring loss, Mean Squared Error (MSE) was used, so as to avoid using the same function for measuring loss and error.

$$MSE = \frac{1}{n}\Sigma_{i=1}^{n}(y_i - y_i')^2 \tag{3.3}$$

$$RMSE = \sqrt{\frac{1}{n}\Sigma_{i=1}^{n}(y_i - y_i')^2} \tag{3.4}$$

Where n is the the batch size, or more generally, the number of predicted outputs. y is the the correct answers given a prediction, and y' is the predicted output.

3.2 Missing Component

There's a vital missing component in Tycho1 that has been purposely left out. This is the learning optimization algorithm. This will be studied further and decided on in chapter 4

3.3 Data Pre-processing

What is interesting about the *GZ2* dataset is that the test set contains more examples than the training set. To accommodate for this, the image augmentation described in 2.4.3 is used. However, unlike Dieleman's model [2], the resulting 16 images from the augmentation process we're not concatenated. Instead, one random image from those 16 generated was selected. This serves to create more inputs to help the model generalize more towards the active parts of each image (bottom right, where the galaxies are), and reduce chances of over-fitting.

Chapter 4

Deep Learning Optimizer Algorithms & Hyper-Parameter Search

A major key to deep learning is the learning optimization algorithm. The model described by Dieleman [2] details the usage of Stochastic Gradient Descent (SGD) of an initial learning rate of 4×10^{-2}, and a Nesterov momentum of 0.9.

However, there are alternatives to this algorithm. Most notably, SGD without Nesterov momentum and Adam optimization. A comparison of these algorithms was made on separate learning rates so as to empirically decide on the most suitable.

4.1 Study Description

[Todo - Add tables and graphs]

The Tycho1 network was set up to take an input of a mini-batch of 16 images, that are augmented 16 times each, with one of the augmented images chosen at random. This model would train for 2000 epochs.

To compare the algorithms, the training was run in a loop to try out the different algorithms along with four different learning rates 4×10^{-2}, $4 \times 10^{-}3$, $1 \times 10^{-}4$, and 1×10^{-4}. Throughout these iterations, the loss and error values on both the training set and validation set is taken every 10 epochs. TensorBoard proved to be a vital tool for carrying out this comparison. It made it so that the loss and error values were stored and easily compared by overlaying them through the UI.

4.2 Results

Each algorithm took roughly thirty five minutes to complete one iteration on one learning rate, so the whole comparison took about 7 hours to complete.

4.2.1 Stochastic Gradient Descent

Stochastic Gradient Descent performed well on the first learning rate, 4×10^{-2}, producing an RMSE of 0.1766 on the training set, and 0.1969 on the validation set, on the final epoch. However, when the learning rate was decreased, it performed very poorly, resulting in training and validation RMSE of approximately 0.45 for all other learning rates.

4.2.2 Stochastic Gradient Descent With Nesterov Momentum

SGD with Nesterov momentum proved even more effective on the 4×10^{-2} learning rate, producing a training and validation RMSE of 0.1766 and 0.1725 on the validation set on the final epoch. However, as the learning rate decreased, the performance decreased too. Interestingly though, the decrease was not as drastic as SGD without Nesterov momentum, as on the $4 \times 10^{-}3$, it achieved a training and validation RMSE of 0.1950, and 0.1564 respectively. This algorithm did however result in error values of about 0.45 on the last two learning rates, just as the previous algorithm did.

4.2.3 Adam

On the first learning rate, the Adam algorithm achieved considerably greater results than other algorithms, producing an RMSE of 0.1811 and 0.1632 on the training and validation sets respectively on the final epoch, with the learning rate of $4 \times 10^{-}2$. This performance did however decrease on the second learning rate of $4 \times 10^{-}3$, to being 0.2304 and 0.2482 for training and validation, respectively. Moreover, on the learning rate of $4 \times 10^{-}4$. It did however decrease to a train RMSE of 0.1517 and a validation RMSE of 0.1479. For the learning rate of $1 \times 10^{-}4$ it achieved an RMSE of 0.1801, and 0.1852 on the training and validation sets respectively. These results showed that the Adam algorithm was far more suitable and stable for this model.

4.2.4 Further Learning Rate Tests On The Adam Algorithm

From the results of section 4.2.3, it looks apparent that lower learning rates perform better, with the exception of the first learning rate of $4 \times 10^{-}2$. So in that vain, to smaller learning rates of $4 \times 10^{-}5$ and $4 \times 10^{-}6$ were tested with the same configuration as the tests above but produced lower RMSE values than it did with a learning rate of $4 \times 10^{-}4$.

Algorithm	Learning Rate	Training RMSE	Validation RMSE
SGD	4×10^{-2}	0.1766	0.1969
	4×10^{-3}	0.4410	0.4526
	4×10^{-4}	0.4528	0.4519
	1×10^{-4}	0.4472	0.4528
Nesterov	4×10^{-2}	0.1819	0.1725
	4×10^{-3}	0.1950	0.1564
	4×10^{-4}	0.4361	0.4329
	1×10^{-4}	0.4446	0.4567
Adam	4×10^{-2}	0.1811	0.1632
	4×10^{-3}	0.2304	0.2482
	4×10^{-4}	0.1566	0.1531
	1×10^{-4}	0.1801	0.1852
	4×10^{-5}	0.1632	0.1824
	4×10^{-6}	0.1593	0.1876

Figure 4.1: The resultant RMSE's for each learning rate on each algorithm

4.3 Conclusion

As the Adam algorithm performed the best on all learning rates, that will be the chosen algorithm for the models. Furthermore, by looking at the graphs on TensorBoard of all the tests carried out on the Adam algorithm, with a smoothing of 0.85, the learning rate of $4 \times 10^{-}4$ seemed to be a marginally better choice than $4 \times 10^{-}2$, even though it began to converge later, it ended with a lower RMSE. Furthermore also sloped further down than the learning rate of $4 \times 10^{-}2$, indicating that it will perform even better if it was run further. Hence the chosen learning rate is $4 \times 10^{-}4$.

Chapter 5

Tycho1 - Evaluation

5.1 Training Details

A training script was written for the Tycho1 model using the Adam Optimizer, on a learning rate of ... The script is set to train on a mini-batch of 256, using the full training set (including the data used in validation for chapter 4), for 10,000 epochs. During the tests in chapter 4, it was noticed that recording the logging summaries took a substantial amount of time, when compared to training, hence the recording interval is set to 100. This means that the TensorBoard log summary was taken every 100 epochs. A nice side effect of this is smoother graphs.

Taking away more data from the training set for validation does not seem viable here, as the training set itself is already smaller than the test set. Hence, a test interval of 1000 is set, This allows for a CSV of predictions on the test set to be generated ever 1000 epochs so as to accurately assess the rate of change in the model's error, in comparison to the train error, as the validation set proved too small in chapter 4.

Unfortunately there was an error in the code, that made it so test conditions were never satisfied. However the status of the model was saved every 100 intervals, meaning allowing the training to be paused and for the script to be altered so as to meet these conditions and produce the test results. Hence we will find that the first few results do not match the test interval.

As the training surpassed $10,000$, it becomes apparent that there is still room for improvement, and that training the model further could in fact keep improving the results. Hence, it was allowed to train for another day and a half. In total training took four days, including the time taken to generate the predictions on the test file, which on average took 45 minutes to generate.

Furthermore, as the training started to produce results, it produced some very low values, but no hard *zero's*. This was a known consequence of using Sigmoid activation in the final

layer. However, the impact seemed greater as the training went on, as the more training is done the values that should be zero, tend closer to zero, but never get to zero. Hence we see a very drastic retardation in loss and error. So as to test this, the code regarding the custom activation layer from Deleemm [2] was adapted so as to normalize the prediction on the test set. This normalization was practiced when the training reached the 12,000th epoch, and onward from there.

5.2 Results

Epochs	Training Error	Test Error
27102	0.1252	0.12111
4001	0.1181	0.11452
6901	0.1060	0.10968
8203	0.1020	0.10636
9004	0.1077	0.10635
11005	0.1016	0.10446
12000	0.1072	0.10283
12000 - Normalized	0.1072	0.10278
14700	0.1043	0.10239
14700 - Normalized	0.1043	0.10220

The final RMSE of 0.10220 would rank the Tycho1 model 55th out of the 326 participants in both the private and public leaderboards of the Kaggle Galaxy Zoo competition. This puts it in the bronze ranking, and 5 spots away from a silver ranking.

5.3 Conclusion

The Tycho1 network does seem to under-fit the data, however not very much, as it has successfully generated a model with what is empirically a good ranking by the competitions standards. However, some issues have come to light from this training.

Normalizing the output, does indeed improve the results, albeit slightly. This makes it apparent that a more suitable activation function is need in the final layer, as although one can normalize the output, the model cannot calculate the gradient for the normalized output on the training set. Hence, it cannot update variable values accordingly to these values.

Although the Sigmoid activation function does produce decent results, it does hit a limit very quickly, which stagnates learning greatly.

Allowing the network to run for another day or two would very likely get it in the silver ranking, however, this is not a very effective strategy, as the Tycho1 model does show some heavy signs of retardation when it it comes to loss, and error.

An important conclusion to note, is that the training errors and test errors are reasonably close to each other, indicating that the assumption of the validation set being to small is correct. This means that we can expect validation loss and error to be lower than training loss and error throughout.

Chapter 6

Tycho1.2

So as to decrease the error, ReLu activation is used in the final layer of the Tycho1.2 network. The reasoning here is that although ReLu does not restrict the output to a range of $0 < x < 1$, this shouldn't be an issue as the input is already normalized to that range, and all the variables in the network (weights and biases) are bound to that range as well. Moreover, a required feature of ReLu is it's ability to detect hard zeros, which the model contains plenty of.

6.1 Validation

So as to validate this hypothesis, an experiment on the Tycho1.2 network is run with the Adam optimizer at a learning rate of ... on a mini-batch of 16 for 2000 epochs. This is the same configuration used in the training of of Tycho1, and was validated in section ... Hence we compare the results of the Tycho1.2 validation to the results of Tycho1 of the same configuration in section ...

6.1.1 Experiment Results

After 2000 epochs, the Tycho1.2 model produced a training error of 0.1559, and a validation error of 0.1287, where as the Tycho1 model had produced a training error of 0.1566, and a validation error of .... However, it is more interesting to see the results at the start of the experiments. The Tycho1 network starts with a training error of and a validation error of , where as the Tycho1.2 begins with a training error of , and a validation error of . Moreover, the Tycho1.2 network converges quicker, as after after 200 epochs the Tycho1 network produces a training error of ..., and a validation error of ..., where as the Tycho1.2 network produces a a training error of ..., and a validation error of ....

6.1.2 Experiment Conclusion

The results shown in this experiment prove that using ReLu activation in the final layer of this network produces better predictions given this dataset, hence this should reflect in the training.

6.2 Training Details Results

The Tycho1.2 network is run with the Adam optimizer at a learning rate of ... on a mini-batch of 256 for 2000 epochs, just as the Tycho1 network was in section ... Throughout the training of this model, the test error was evaluated so as to compare with the Tycho1 solutions.

The training of this model proved to be far more effective than the Tycho1 model, as the effects of the ReLu rectifier in the final layer were apparent early on. A solution CSV file was generated after 500 training epochs with the Tycho1.2 network so as to analyse it at an early stage. This CSV file contains plenty of hard zeros in it, which is as hypothesized . The Tycho1.2 network shows much quicker convergence from the start and keeps doing so up to around 5000 training epochs. Although the network does continue to improve after 5000 epochs, this improvement does slow down. So as to analyse the network further, it was left to train for approximately 7 days. At the .... epoch, it produced a solution with a test error of ....

6.3 Conclusion

What we can conclude from this training is that using ReLu activation in the final layer empirically improves the model's prediction, and greatly speeds up training. The lowest test error which this model produces is of ..., which would rank 38^{th} in the Kaggle competition, which although significant, did take 7 days to achieve. Throughout those 7 days of training, the model's improvement did show great signs of slowing down. Hence, leaving it train any further would not be too wise, as this model is still underfitting.

Chapter 7

The Network: Tycho2

Seeing as the Tycho1, and Tycho1.2 networks both exhibit signs of underfitting, we will need a network with more parameters to better fit the data. Hence, we develop a new network, titled Tycho2. This network will have larger fully connected layers so as to increase parameters. Moreover, this network will use a similar data augmentation step as described by Sanders dd [2].

Each input image is processed to generate 16 different augments as described in section ...data... All of these 16 augments are fed through the convolutional layers. After the augmented images go through the convolutional layers, the 16 augments of each image are concatenated into one bigger feature. This feature is then fed through two fully connected dense layers. This sees that the first dense layer is 16 times greater than that in Tycho1. This process is described in figure ...

7.1 Validation

The Tycho2 network is trained for 2000 epochs with a mini-batch of 16, just as the Tycho1.2 network was. This validation doesn't give way to much insight, but it does show that increasing the parameters does indeed improve the model, as both training and validation errors are lower after 2000 epochs.

7.2 Training Details Results

This

7.3 Conclusion

Chapter 8

Tycho4

8.1 Validation

8.2 Training Details Results

8.3 Conclusion

Chapter 9

Model Averaging

Chapter 10

Project Conclusion

Chapter 11

Closing Remarks

Figure .1: test

References

[1] K. W. Willett, C. J. Lintott, S. P. Bamford, K. L. Masters, B. D. Simmons, K. R.V. Casteels, E. M. Edmondson, L. F. Fortson, S. Kaviraj, W. C. Keel, T. Melvin, R. C. Nichol, M. J. Raddick, K. Schawinski, R. J. Simpson, R. A. Skibba, A. M. Smith, and D. Thomas. Galaxy zoo 2: detailed morphological classifications for 304,122 galaxies from the sloan digital sky survey. *Mon. Not. R. Astron. Soc.*, 000:1–29, 2013.

[2] S. Dieleman. My solution for the galaxy zoo challenge, 2014.

[3] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back propagating errors. 323:533–536, 10 1986.

[4] Ashwin Bhandare, Maithili Bhide, Pranav Gokhale, and Rohan Chandavarkar. Applications of convolutional neural networks. *(IJCSIT) International Journal of Computer Science and Information Technologies*, 7(5):2206–2215, 2016.

[5] Tu Dinh Nguyen and Truyen Tran. Galaxy zoo challenge with convolutional neural networks. 2014.

[6] C Fang-Chieh. Galaxy zoo challenge: Classify galaxy morphologies from images. 2014.

[7] Ian Goodfellow, David Warde-Farley, Mehdi Mirza, Aaron Courville, and Yoshua Bengio. Maxout networks. 2013.

[8] V. Nair and G. E. Hinton. Rectified linear units improve restricted boltzmann machines. 2010.