THE UNIVERSITY *of* EDINBURGH
School of Physics
and Astronomy

Emulating Power Spectra of Dark Matter Halos using Neural Networks

Lukah Connolly Sams

2024

Abstract

Three Tensorflow neural networks were optimised in the number of layers and nodes per layer, training on data provided by the Quijote simulation. Attempting to emulate different regions of the power spectra from five cosmological parameters of the ΛCDM model, in order to determine which approach is most viable, if any. Three models were created: a low k model (ranging from $0.005 < k > 0.5$), a high k model (ranging from $0.2 < k > 0.5$) and a model also consisting of a low k as an extra input for a high k output (low k ranging from $0.005 < k > 0.2$, and high k ranging from $0.2 < k > 0.5$). The optimal number of layers were determined as 2, 2 and 3 with an optimal number of nodes of 600, 200, and 300 for the low k, the high k, and the low k input, high k output models respectively. These models could each emulate the power spectra accurately to a 5% error margin, with the Low k input, High k output model generating the most accurate predictions when compared with the average values from the Quijote simulation's $P(k)$ values. Further insights into whether the optimisation of the layers and nodes of each model were the result of statistical randomness or not could be ascertained via more focused re-runs of the optimisation functions with different seeds for training and testing samples, as well as a more precise number for the number of nodes, in order to achieve less error in final model predictions.

Declaration

I declare that this project and report is my own work.

Supervisor: Richard Neveux

Contents

1 Introduction

Understanding the distribution of matter in the universe is of great importance to modern physics, specifically as it provides insights into the conditions at the origin of the universe[1]. The power spectra of dark matter halos are useful tools for this endeavour as they provide details about the distribution and density of matter at a given point in time and space [2]. Being able to emulate this data accurately reduces the computational overheads and bottlenecks present with other methods[3].

Quijote, a Many-Body Simulation developed/licensed by Massachusetts Institute of Technology (MIT), provides data for over $82,000$ simulations for cosmological research[4]. The data computed by Quijote has been used in this project for training and validating the neural networks, as well as determining whether the machine learning approach presented is viable for emulating power spectrum data accurately.

In current cosmology research, machine learning is a commonly used tool for emulating large data sets[3]. Neural networks are critical to this as they accelerate calculations and can generate higher resolutions that are absent when computational compromises are made in simulations[5], [6].

1.1 Cosmology

1.1.1 Cosmological Model

Currently, the main cosmological model that seeks to encapsulate the rules of the universe is the Lambda Cold Dark Matter model (abbreviated to ΛCDM model or LCDM model)[7]. This model predicts, using the Friedmann equations, with a constant for a yet unobserved energy (dark energy) and postulated cold dark matter, that at large enough scales the universe should have a uniform distribution of energy[8]. The ΛCDM model is the current standard model of Big Bang cosmology and is a key focus of modern cosmological research[9].

Figure 1 shows an impression of the expansion of the universe, in the ΛCDM model. It also highlights key moments predicted to have happened during the expansion of the universe ranging from the time of the Big Bang up to today.

1.1.2 Power Spectra

Knowing the density, or the distribution of matter and energy in the universe is valuable information, and can be expressed through the use of the Two-Point Correlation Function, $\xi(r)$, and the corresponding power spectra, $P(k)$, (an example of which is shown in figure 2). The two point correlation function can be Fourier transformed and express the variance of the coefficients to produce the $P(k)$values [11].

Dark matter halos are regions of space which have decoupled from the cosmic expansion and are gravitationally bound [?]. Through the ΛCDM model, it is theorised that these dark matter halos always contain galaxies [?]. Although these structures have not been observed directly, they are inferred from —.

1.1.3 Cosmological Parameters

Outwith the ΛCDM model, there are up to 10 cosmological parameters that are used in research. The 5 parameters of interest, for the neural networks in this report are: Ω_m, the density of matter; Ω_b, the density of baryonic matter; h, the reduced Hubble constant ($H_0/100$); n_s, the scalar power law index;

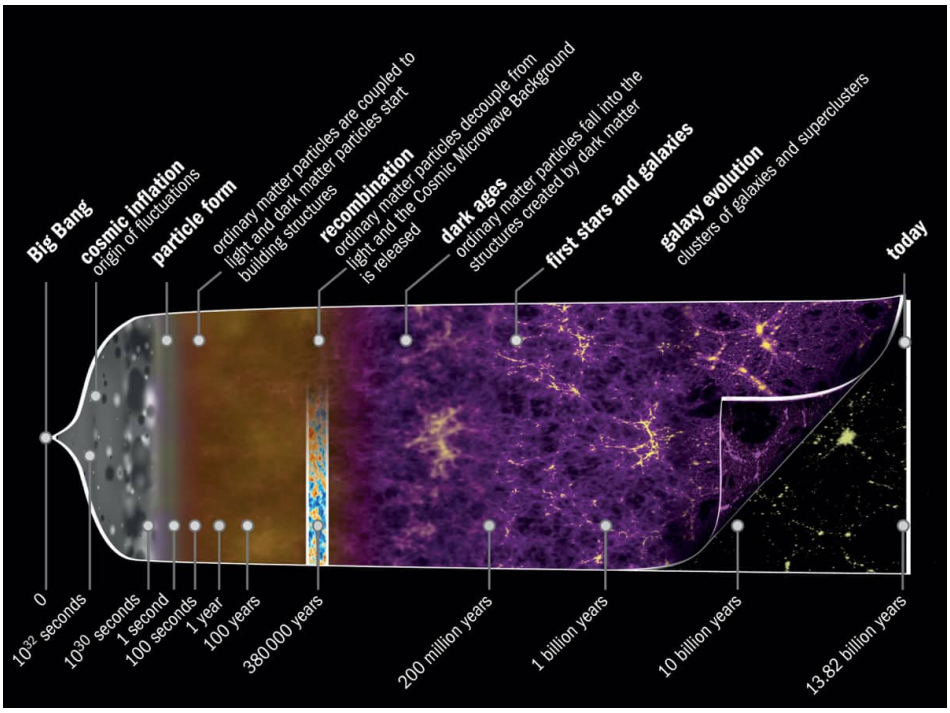


Figure 1: Interpretation of the expansion of the universe from the Big Bang to current time, showing the main stages of evolution. Found at [10].

and σ_8, the amplitude of the (linear) power spectrum on the scale of $8h^{-1}Mpc$. It has been shown that 6 parameters is the minimum needed to define an accurate state of the universe[12], [7].

1.2 Quijote Simulation

The Quijote Simulation is a collection of data and code that is licensed by MIT[4]. It aims to provide cosmological data for use in training machine learning algorithms. Many different types of cosmological data are available across multiple servers, some of which include void catalogues, correlation functions, and halo power spectra (which is the main focus of this project).

1.3 Machine Learning and Neural Networks

Machine learning, a subset of artificial intelligence (AI), deals with given inputs and outputs to generate a set of rules that govern how the data is processed[13]. In order to train models, the nodes in a neural network are assigned random weights and bias', which will change as the training proceeds, that govern the connections/likelihood of a connection given the previous layer's data[14]. Apart from a dataset for training the neural networks, and secondary set of data is necessary to valid/evaluate the results to avoid bias from the training data. A simplistic diagram of a generic feed forward neural network can be seen in figure 3.

There are many Python packages that can be employed for various uses in machine learning. In this project, the TensorFlow[15] and SciKit-Learn[16] packages were used for generating the Neural networks and separating data respectively.

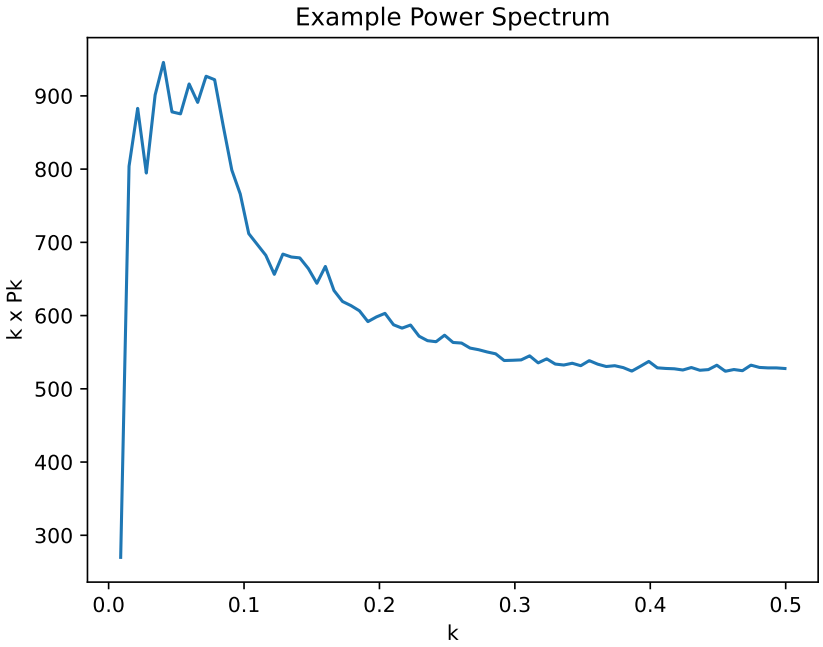


Figure 2: Example power spectra at $z = 0$ in the region $0.005 < k > 0.5$, using the data from the Quijote simulation. By convention, the y values are multiplied by their k values.

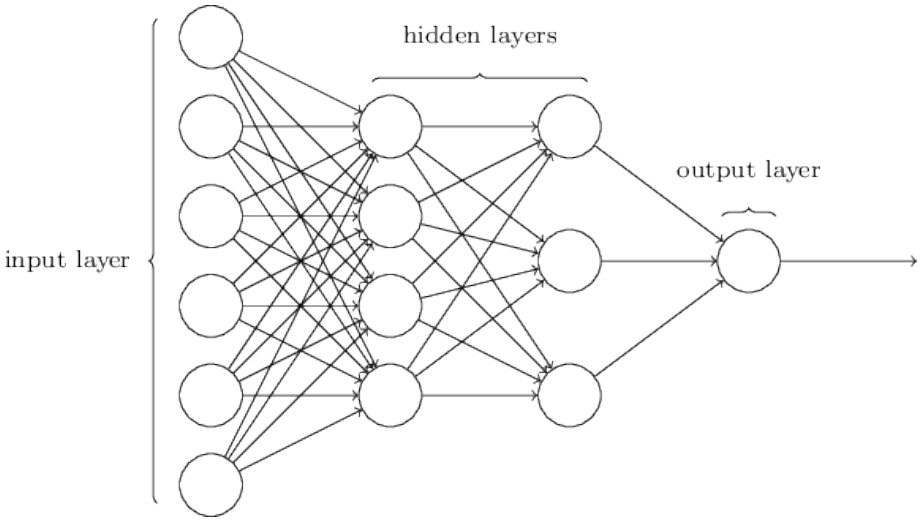


Figure 3: Example of a generic neural network architecture, showing input layers, hidden layers, and output layers each with varying numbers of nodes, found at[17].

2 Emulating the Power Spectrum

2.1 Preparing the Data

Before training the models, the cosmological parameters need preprocessing. This can be visualised by plotting them on a histogram, as shown in figure 4. It is clear to see that the values are of differ-

ent averages and spreads. To ensure that some cosmological parameters don't dominate the bias' or weightings in the neural network training, the values of each of the parameters were adjusted using the normalisation equation:

$$X_{i_{norm}} = \frac{X_i - X_{min}}{X_{max} - X_{min}} \text{ [18].}$$

The results were then divided by their averages to achieve a spread centred on 1. This final step, although unconventional[19], was done undertaken to create a more dynamic range of values by increasing the spread of the data by a factor of 2 (values ranging from 0 to 2 with an average of 1), thus making the models more sensitive. This step was introduced in an effort to emulate the 'bumps' and/or extreme values in the power spectra.

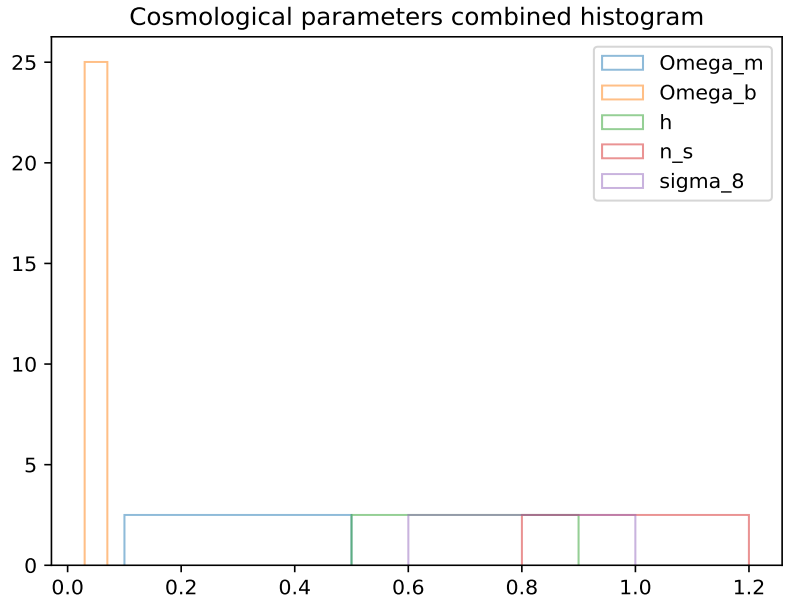


Figure 4: Histogram showing the distribution of unprocessed values for each of the cosmological parameters.

For the Low k input, High k output model, the low k power spectrum that was used as an input along side the normalised cosmological parameters. This data was normalised for each k, using the equation above, and was also divided by the averages for each k. Histograms showing the before and after of the normalisation of the cosmological parameters are shown in figures 4 and 5, respectively.

2.2 Optimising the Neural Network

To emulate the power spectra at low and high k's, the neural networks needed to be optimised for the number of hidden layers and nodes per layer, using the input data of the ΛCDM cosmological parameters, and generating the output $P(k)$ of the halo power spectrum. The TensorFlow[15] package was used to create, train and evaluate these neural networks.

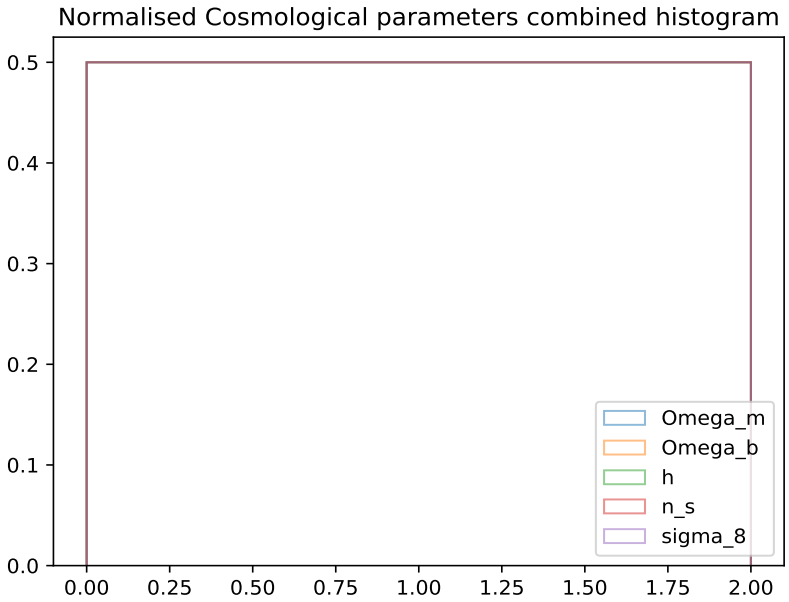


Figure 5: Histogram showing the distribution of normalised values for each of the cosmological parameters.

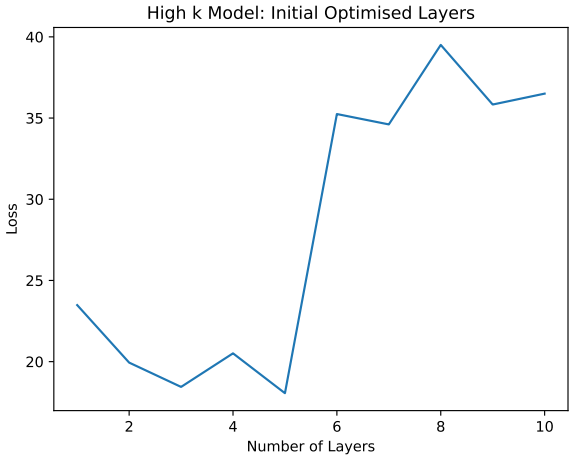
In order to model the low and high k behaviour separately, the data points for the power spectrum from all 2000 samples were constricted by a range of $0.005 < k > 0.5$, and $0.2 < k > 0.5$, for low and high k respectively. The models for low and high k were then processed by looping through a model constructing, training and evaluating function, which calculated the end loss values for each neural network structure, see appendix A for code.

The optimal number of layers and nodes were determined separately, with the optimisation in the number of layers being completed first (with an initial 100 nodes per layer). This was then followed by the number of nodes per layer, and then a final optimisation in the number of layers again. In this final optimisation, the optimum number of nodes was used to ensure that the correct optimal number of layers was correct. The losses were then plotted for each step which determined the minimised loss value for the different structures. Figures 6, 7, and 8 show the results from the three stages of optimisation for all three models.

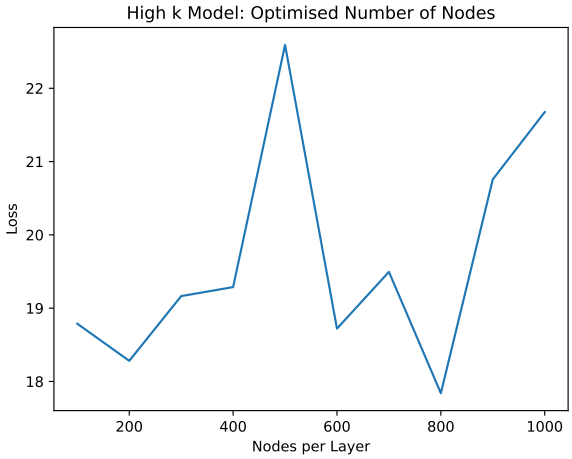
In all three models, it is interesting to note that at around $4-5$ layers the NN's loss values spike. This is likely an indication that the model is overfitting the data after these value. The same cannot be said for the node's losses as they appear random, with no significant difference between different values.

It is also clear that the difference in loss values are minimal and appear random, except for the Low k model. In order to ascertain whether this is the actual optimum value for the number of layers and nodes or a result of random statistics, multiple re-runs of the optimisation functions can be called with different training and testing seeds.

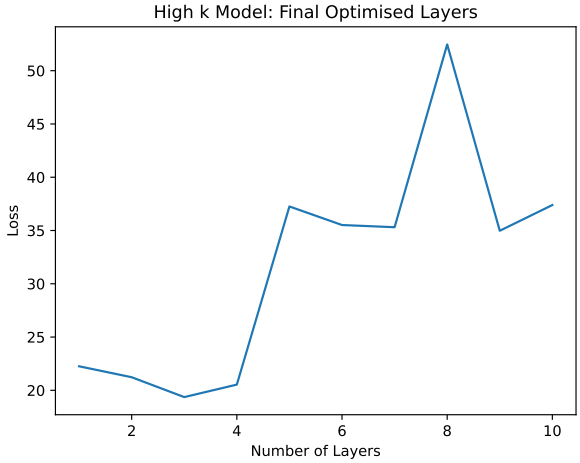
Using the data from figures 6, 7, and 8, the final values for the optimal number of hidden layers and nodes per hidden layer are shown in table 1.



(a) Graph displaying the initial run, optimising the number of hidden layers in the high k model. Showing the loss for each trained model with 100 nodes for each layer.



(b) Graph displaying the loss for each trained model with an optimised number of layers and set amount of nodes per layer in the high k model.

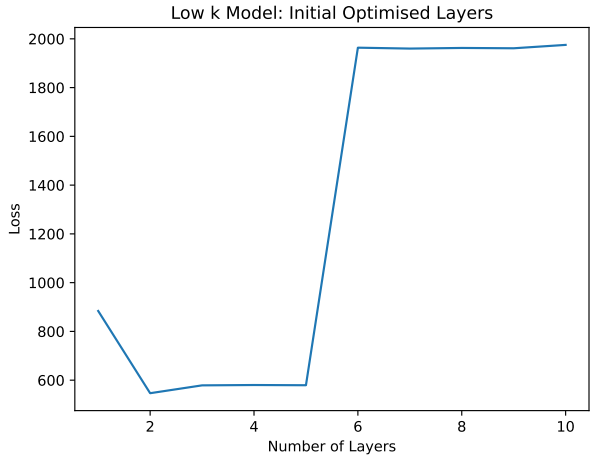


(c) Graph displaying the final run, optimising the number of hidden layers in the high k model. Showing the loss for each trained model with the optimised number of nodes for each layer.

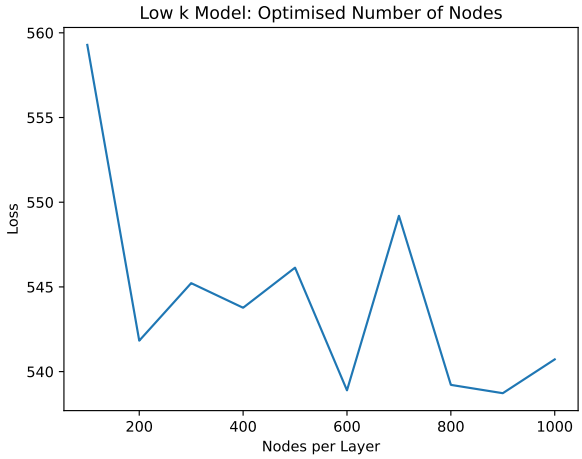
Figure 6: High k model optimisations graphs showing the loss values corresponding to the number of hidden layers and nodes per hidden layer.

Model	**Optimal Layers**	**Optimal Nodes**
Low k	2	600
High k	3	200
Low k Input, High k Output	3	300

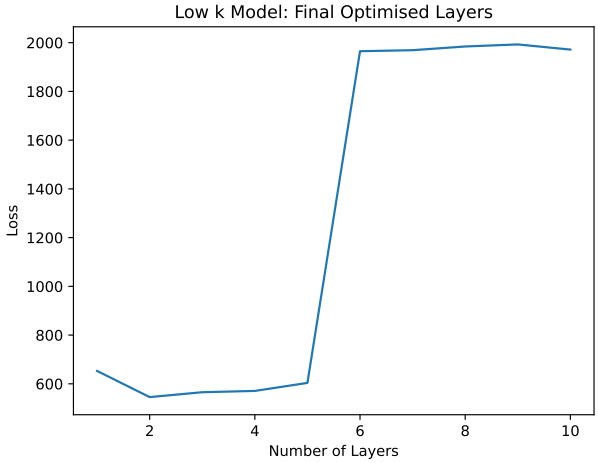
Table 1: Table of Optimal number of layers and nodes per layer for the neural networks of each model.



(a) Graph displaying the initial run, optimising the number of hidden layers in the low k model. Showing the loss for each trained model with 100 nodes for each layer.



(b) Graph displaying the loss for each trained model with an optimised number of layers and set amount of nodes per layer in the low k model.



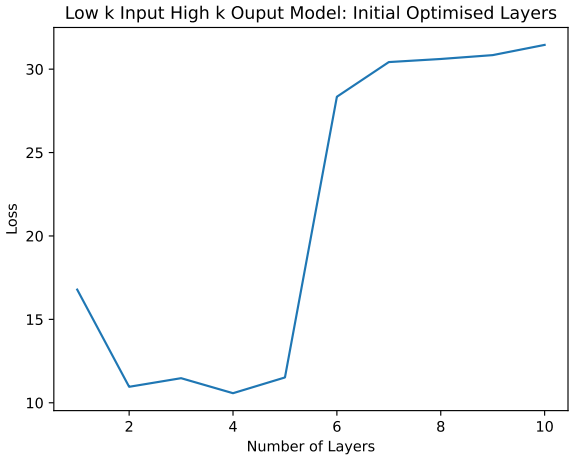
(c) Graph displaying the final run, optimising the number of hidden layers in the low k model. Showing the loss for each trained model with an optimised number of nodes for each layer.

Figure 7: Low k model optimisations graphs showing the loss values corresponding to the number of hidden layers and nodes per hidden layer.

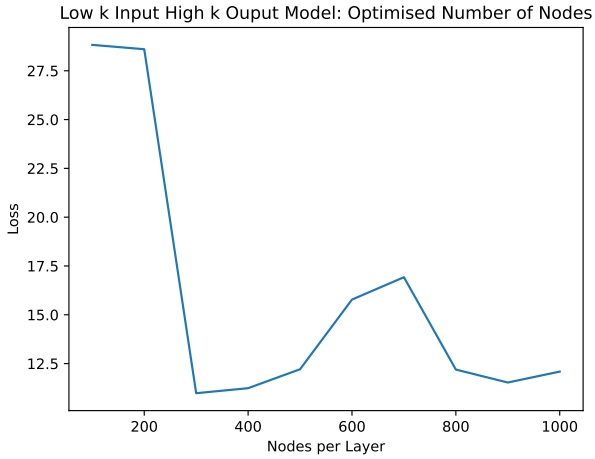
2.3 Comparing Data with Quijote Simulations

After the optimal number of layers and nodes are determined for each model, and thus defining the final architecture of the Neural Network, the models can then be trained so that they can produce predicted outputs from any input. A new split of training and testing data was created using a different random seed for the training of the final models, shown in figure 9.

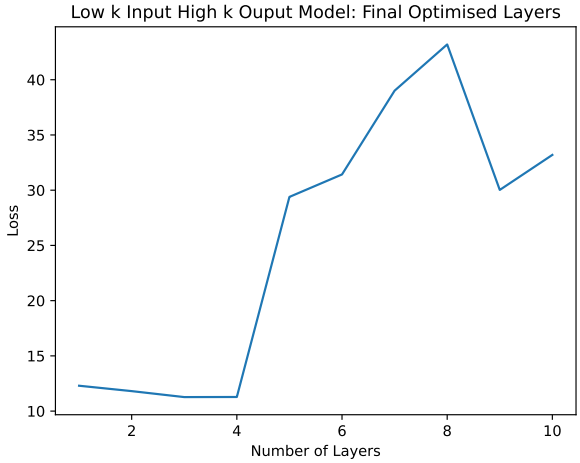
It is clear to see that whilst using SciKit-Learn's `train_test_split` function[16], no clear bias has been made, for any of the cosmological parameters, when splitting the data for testing and training.



(a) Graph displaying the initial run, optimising the number of hidden layers in the low k input, high k output model. Showing the loss for each trained model with 100 nodes for each layer.



(b) Graph displaying the loss for each trained model with an optimised number of layers and set amount of nodes per layer in the low k input, high k output model.



(c) Graph displaying the final run, optimising the number of hidden layers in the low k input, high k output model. Showing the loss for each trained model with an optimised number of nodes for each layer.

Figure 8: Low k input, high k output k model optimisations graphs showing the loss values corresponding to the number of hidden layers and nodes per hidden layer.

As shown in figures 10, and 11 all models are able to accurately emulate the power spectra data of the testing samples within a 5% error. It is also clear to see that the Low k input, High k output model provides a more accurate prediction than the other two models, likely due to the increased constraints of the $0.005 < k > 0.2$ region of the spectra as an input.

Figure 9: Corner plot showing the distribution of values for the cosmological parameters: Ω_m, Ω_b, h, n_s, and σ_8. The split for the cosmological parameters between the training and testing data sets, used in the final neural network models, are shown with their corresponding colour (blue and orange, respectively).

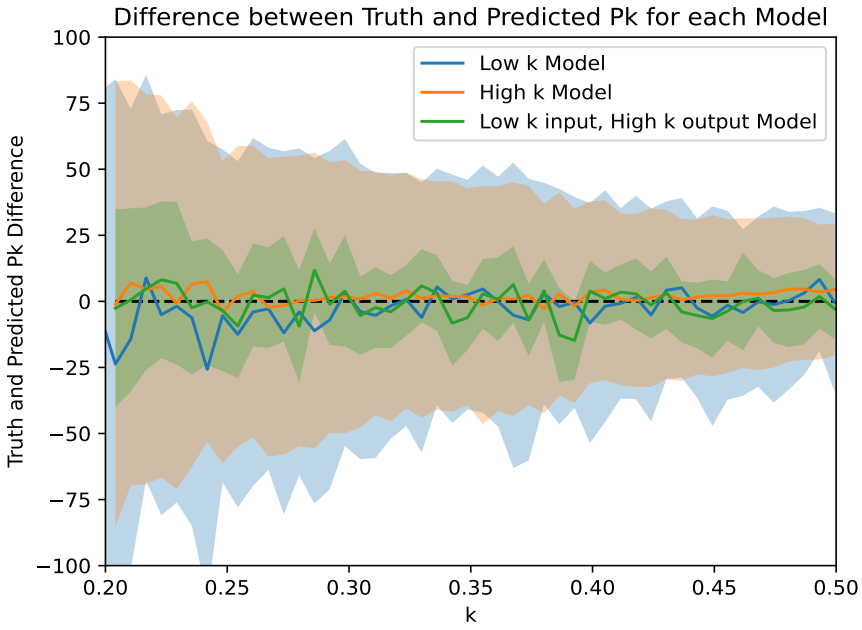


Figure 10: Plot of the difference between the average truth value and average predicted values for each k. The different models (High k, Low k, and Low k input, High k output) are shown in orange, blue, and green colours, respectively, with a shaded region defined by the standard deviation of each set of data.

3 Discussion

The results of the optimisation clearly show that the optimal number of layers is $< 4 - 5$ and that any more hidden layers in the neural networks would likely overfit. However, it is key to note that the differences between loss values for layer numbers below these values are negligible and would require multiple runs with different seeds to determine accurately, and without bias from one set of training and testing samples.

With the predicted values resulting in having an error of $< 5\%$, these models can be deemed moderately accurate when comparing with similar result from a study[6], where they accepted an error of $< 5\%$ for their least accurate model.
An improvement to optimising the number of nodes could be made by further increasing the precision through lower node intervals/more data points to the graphs. This option would take up a significant amount of extra computing time, but could be sped up by using progressively smaller node intervals and ranges.

The optimisation functions were simplistic in that each hidden layer contained the same number of nodes as each other. This resulted in a very restricted rectangular neural network architecture. Changing the overall shape of the neural network, so that individual hidden layers can be optimised with different numbers of nodes, could produce better connections between layers. It should be noted that this process could induce a significant computing time cost.

The data and methods used in this project could be extended to learn about how redshift affects how the models are optimised, and the accuracy of their predictions. This would give more insight into the

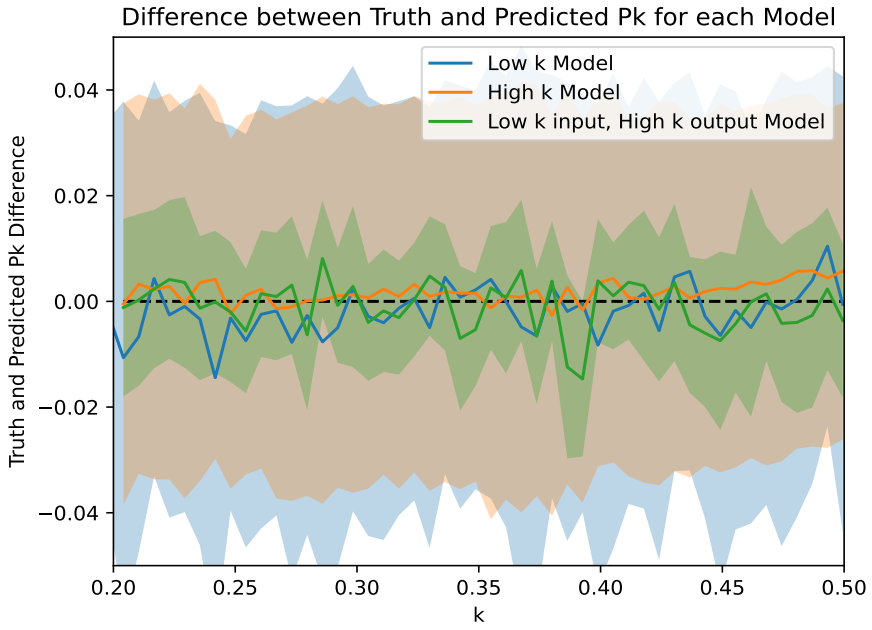


Figure 11: Plot of the difference between the average truth value and average predicted values for each k, divided through by the average truth value for each k. The different models (High k, Low k, and Low k input, High k output) are shown in orange, blue, and green colours, respectively, with a shaded region defined by the standard deviation of each set of data.

evolution of the universe, rather than just one snapshot in time.

4 Conclusion

In conclusion, in an attempt to motivate an accurate procedure for emulating power spectra at $z = 0$ three models were created and compared. After optimising their respective neural network's number of layers and nodes, the most accurate model (Low k Input, High k Output) produced predicted data that was within 2% of the true values. This model required the input of the 5 cosmological parameters: Ω_m, Ω_b, h, n_s, and σ_8, as well as the lower end of the power spectrum ($0.005 < k > 0.2$) in order to predict the high k region of the power spectra ($0.2 < k > 0.5$). Although the results here were restricted to data from z=0, similar methods can be employed for varying redshifts. Improvements such as varying node sizes per layer, or more precise node sizes, could be made in order to improve the accuracy of the results. Given more time, this project would have included procedures to reduced statistical randomness from the optimisation steps, which would have led to more accurate predictions.

5 Acknowledgements

I would like to thank my supervisor, Richard Neveux for their guidance throughout this project, in particular with the many technical difficulties I encountered. Addintionally, I had a number of health complications during the course of this project, and would like to thank my family, personal tutor Beth Biller, and Richard Neveux again, for their support.

6 References

[1] Daniel J. Eisenstein et al. "Can Baryonic Features Produce the Observed 100 $h1Mpc$ Clustering?" In: *The Astrophysical Journal* 494.1 (Jan. 1998), p. L1. DOI: 10.1086/311171. URL: https://dx.doi.org/10.1086/311171.

[2] Risa H. Wechsler and Jeremy L. Tinker. "The Connection Between Galaxies and Their Dark Matter Halos". In: *Annual Review of Astronomy and Astrophysics* 56 (2018), pp. 435–487. ISSN: 1545-4282. DOI: https://doi.org/10.1146/annurev-astro-081817-051756. URL: https://www.annualreviews.org/content/journals/10.1146/annurev-astro-081817-051756.

[3] Alessio Spurio Mancini et al. "CosmoPower: emulating cosmological power spectra for accelerated Bayesian inference from next-generation surveys". In: *Monthly Notices of the Royal Astronomical Society* 511.2 (Jan. 2022), pp. 1771–1788. ISSN: 0035-8711. DOI: 10.1093/mnras/stac064. eprint: https://academic.oup.com/mnras/article-pdf/511/2/1771/42440282/stac064.pdf. URL: https://doi.org/10.1093/mnras/stac064.

[4] Francisco Villaescusa-Navarro et al. "The Quijote Simulations". In: *apjs* 250.1, 2 (Sept. 2020), p. 2. DOI: 10.3847/1538-4365/ab9d82. arXiv: 1909.05273 [astro-ph.CO].

[5] Yin Li et al. "AI-assisted superresolution cosmological simulations". In: *Proceedings of the National Academy of Sciences* 118.19 (2021), e2022038118. DOI: 10.1073/pnas.2022038118. eprint: https://www.pnas.org/doi/pdf/10.1073/pnas.2022038118. URL: https://www.pnas.org/doi/abs/10.1073/pnas.2022038118.

[6] Jamie Donald-McCann, Kazuya Koyama, and Florian Beutler. "matryoshka II: accelerating effective field theory analyses of the galaxy power spectrum". In: *Monthly Notices of the Royal Astronomical Society* 518.2 (Nov. 2022), pp. 3106–3115. ISSN: 0035-8711. DOI: 10.1093/mnras/stac3326. eprint: https://academic.oup.com/mnras/article-pdf/518/2/3106/47465457/stac3326.pdf. URL: https://doi.org/10.1093/mnras/stac3326.

[7] Gary Hinshaw et al. "Nine-year Wilkinson Microwave Anisotropy Probe (WMAP) observations: cosmological parameter results". In: *The Astrophysical Journal Supplement Series* 208.2 (2013), p. 19.

[8] P. J. E. Peebles and Bharat Ratra. "The cosmological constant and dark energy". In: *Reviews of Modern Physics* 75.2 (Apr. 2003), pp. 559–606. ISSN: 1539-0756. DOI: 10.1103/revmodphys.75.559. URL: http://dx.doi.org/10.1103/RevModPhys.75.559.

[9] https://www.facebook.com/gourav.khullar. *The Bullet Cluster – A Smoking Gun for Dark Matter!* Nov. 2016. URL: https://astrobites.org/2016/11/04/the-bullet-cluster-a-smoking-gun-for-dark-matter/.

[10] Syed Faisal ur Rahman. *The enduring enigma of the cosmic cold spot*. Feb. 2020. URL: https://physicsworld.com/a/the-enduring-enigma-of-the-cosmic-cold-spot/.

[11] Hannu Kurki-Suonio. "Galaxy Survey Cosmology". In: *University of Helsinki, Finland* (2023).

[12] NASA / LAMBDA Archive Team. *LAMBDA - Parameters*. URL: https://lambda.gsfc.nasa.gov/education/graphic_history/parameters.html.

[13] IBM. *AI vs. Machine Learning vs. Deep Learning vs. Neural Networks: What's the difference?* June 2023. URL: https://www.ibm.com/blog/ai-vs-machine-learning-vs-deep-learning-vs-neural-networks/.

[14] M.Y Rafiq, G Bugmann, and D.J Easterbrook. "Neural network design for engineering applications". In: *Computers & Structures* 79.17 (2001), pp. 1541–1552. ISSN: 0045-7949. DOI: https://doi.org/10.1016/S0045-7949(01)00039-6. URL: https://www.sciencedirect.com/science/article/pii/S0045794901000396.

[15] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: https://www.tensorflow.org/.

[16] F. Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[17] Michael A Nielsen. *Neural Networks and Deep Learning*. 2018. URL: http://neuralnetworksanddeeplearning.com/chap1.html.

[18] Farrukh Aslam Khan et al. "A novel two-stage deep learning model for efficient network intrusion detection". In: *IEEE Access* 7 (2019), pp. 30373–30385.

[19] Sebastian Raschka and Vahid Mirjalili. *Python machine learning: Machine learning and deep learning with Python, scikit-learn, and TensorFlow 2*. Packt publishing ltd, 2019, p. 125.

A Appendix Neural Network Optimisation Code

```
import tensorflow as tf
import numpy as np

def optimal_layers(max_layers, x_train, y_train, x_valid, y_valid, node_size=100,
                                                activation='relu', batch_size=16, epochs=
                                                2000, patience=15):
    """
    Creates multiple neural networks with different numbers of hidden layers. Then
                                                    trains and evaluates them, returning
                                                    the loss value for each hidden layer
                                                    number.

    Parameters
    ----------
    max_layers : int
        Max number of hidden layers to train
    x_train : array-like
        Input training data
    y_train : array-like
        Output training data
    x_valid : array-like
        Input validation data
    y_valid : array-like
        Output validation data
    node_size : int, optional
        Number of nodes per hidden layer, by default 100
    activation : str, optional
        Activation function for hidden layers, by default 'relu'
    batch_size : int, optional
        Batch size for evaluation, by default 16
    epochs : int, optional
        Number of epochs for training, by default 2000
    patience : int, optional
        Patience for EarlyStopping callback function, by default 15

    Returns
    -------
    array-like/list
        list of layers values and corresponding loss values.
    """

    #arrays/list for final values
    layer_number = np.arange(1, max_layers + 1)
    layers_losses = []

    #loop for the number of layers
    for i in range(len(layer_number)):

        #initialise model
        model = tf.keras.Sequential()
        model.add(tf.keras.layers.Input(shape=len(x_train[0])))

        #loop to add set amount of layers
        x = 0
        while x < layer_number[i]:

            model.add(tf.keras.layers.Dense(node_size, activation=activation))
            x += 1

        #add output layer
        model.add(tf.keras.layers.Dense(len(y_train[0]), activation='linear'))

        #compile model with adam optimizer and MeanAbsoluteError loss function
```

```
        model.compile(optimizer=tf.keras.optimizers.Adam(), loss=tf.keras.losses.
                                                    MeanAbsoluteError(), metrics=['
                                                    accuracy'])

        #train model with Early Stopping callback
        model.fit(x_train, y_train, epochs=epochs, validation_data=(x_valid, y_valid),
                                                     callbacks=[tf.keras.callbacks.
                                                    EarlyStopping(monitor='loss',
                                                    patience=patience)])

        #evaluate model and return history
        history = model.evaluate(x_valid, y_valid, batch_size=batch_size, return_dict=
                                                    True)

        #record final loss value for model
        layers_losses.append(history['loss'])

        #clear for new model
        tf.keras.backend.clear_session()


    return layer_number, layers_losses

def optimal_nodes(max_nodes, x_train, y_train, x_valid, y_valid, start_nodes=100,
                                            node_interval=100, layer_number=2,
                                            activation='relu', batch_size=16, epochs=
                                            2000, patience=15):
    """
    Creates multiple neural networks with different numbers of nodes per hidden layer.
                                                 Then trains and evaluates them,
                                                returning the loss value for each node
                                                per hidden layer value.

    Parameters
    ----------
    max_nodes : int
        Maximum number of nodes per hidden layer
    x_train : array-like
        Input training data
    y_train : array-like
        Output training data
    x_valid : array-like
        Input validation data
    y_valid : array-like
        Output validation data
    start_nodes : int, optional
        Starting point for different model's nodes per hidden layer, by default 100
    node_interval : int, optional
        Nodes per hidden layer interval for different models, by default 100
    layer_number : int, optional
        Number of hidden layers, by default 2
    activation : str, optional
        Activation function for hidden layers, by default 'relu'
    batch_size : int, optional
        Batch size for evaluation, by default 16
    epochs : int, optional
        Number of epochs for training, by default 2000
    patience : int, optional
        Patience for EarlyStopping callback function, by default 15

    Returns
    -------
    array-like/list
        list of nodes per hidden layer values and corresponding loss values.
    """
```

```
    #arrays/list for final values
    node_number = np.arange(start_nodes, max_nodes + node_interval, node_interval)
    nodes_losses = []

    #loop for the number of nodes per hidden layer
    for i in range(len(node_number)):

        #initialise model
        model = tf.keras.Sequential()
        model.add(tf.keras.layers.Input(shape=(len(x_train[0]),)))

        #loop to add set amount of layers
        x = 0
        while x < layer_number:

            model.add(tf.keras.layers.Dense(node_number[i], activation=activation))
            x += 1

        #add output layer
        model.add(tf.keras.layers.Dense(len(y_train[0]), activation='linear'))

        #compile model with adam optimizer and MeanAbsoluteError loss function
        model.compile(optimizer=tf.keras.optimizers.Adam(), loss=tf.keras.losses.
                                                MeanAbsoluteError(), metrics=['
                                                accuracy'])

        #train model with Early Stopping callback
        model.fit(x_train, y_train, epochs=epochs, validation_data=(x_valid, y_valid),
                                                 callbacks=[tf.keras.callbacks.
                                                EarlyStopping(monitor='loss',
                                                patience=patience)])

        #evaluate model and return history
        history = model.evaluate(x_valid, y_valid, batch_size=batch_size, return_dict=
                                                True)

        #record final loss value for model
        nodes_losses.append(history['loss'])

        #clear for new model
        tf.keras.backend.clear_session()


    return node_number, nodes_losses

def optimal_nn(nodes, layers, x_train, y_train, x_valid, y_valid, activation='relu',
                                            epochs=2000, patience=15):
    """
    Creates a neural network of specified size, then proceeding to train and test
                                                model from samples given, using Adam
                                                optimizer and MeanAbsolutError loss
                                                function.

    Parameters
    ----------
    nodes : int
        Number of nodes per hidden layer
    layers : int
        Number of hidden layers
    x_train : array-like
        Input training data
    y_train : array-like
        Output training data
    x_valid : array-like
```

```
        Input validation data
    y_valid : array-like
        Output validation data
    activation : str, optional
        Type of activation function used for hidden layers, by default 'relu'
    epochs : int, optional
        Number of epochs for training, by default 2000
    patience : int, optional
        Patience for EarlyStopping callback function, by default 15

    Returns
    -------
    tf.keras.Model
        Trained tensorflow model
    """

    #initialise model and input layer
    model = tf.keras.Sequential()
    model.add(tf.keras.layers.Input(shape=(len(x_train[0],))))

    #loop to add set amount of layers
    x = 0
    while x < layers:

        model.add(tf.keras.layers.Dense(nodes, activation=activation))
        x += 1

    #add output layer
    model.add(tf.keras.layers.Dense(len(y_train[0]), activation='linear'))

    #compile with Adam optimiser, Mean absolute error loss, and accuracy metrics
    model.compile(optimizer=tf.keras.optimizers.Adam(), loss=tf.keras.losses.
                                            MeanAbsoluteError(), metrics=['accuracy
                                            ']) # can comput roc after

    #train model
    model.fit(x_train, y_train, epochs=epochs, validation_data=(x_valid, y_valid),
                                            callbacks=[tf.keras.callbacks.
                                            EarlyStopping(monitor='loss', patience=
                                            patience)])

    #return model
    return model
```