PHAS0056 Machine Learning Mini Project

Exploring the Fundamental Parameters of the Universe through Machine Learning

Ben Elliot

Examining 2D simulations from the CAMELS Multifield Dataset (CMD), employing deep learning to understand underlying physics and evaluate the predictive value of various map types on cosmological parameters, emphasizing key parameters for universe analysis.

Physics Dept.
University College London
United Kingdom
March 21, 2024

1. Introduction

This report presents an analysis of 2D simulations of the universe sourced from the **C**osmology and **A**strophysics with **M**achin**E** **L**earning **S**imulations (**CAMELS**) dataset, employing deep learning methodologies to further our comprehension of the fundamental characteristics of our universe. Our investigation includes training models on individual map types as well as combinations of multiple maps, with a dual focus on attaining optimal accuracy in predicting cosmological parameters through specific machine learning techniques and evaluating the significance of individual map types towards overall predictive accuracy. Through doing this, we hope to leverage machine learning methodologies to develop a deep network capable of discerning the intrinsic correlations between maps and the underlying parameters dictating their production, shedding light on the fundamental physical laws governing cosmological parameters and the resultant universe configurations.

2. CAMELS Simulations and Current Literature

The CAMELS dataset comprises a collection of both hydrodynamic and N-body simulations depicting the universe. "*As of January 2024, CAMELS contains 12,903 cosmological simulations: 5,164 N-body and 7,739 hydrodynamic simulations*" and over 70TB of data total [1].

The CAMELS simulations serve diverse purposes, among which parameter estimation is a notable application. This involves employing machine learning algorithms to approximate general functions effectively. Specifically, in the context of parameter estimation, such algorithms aid in constraining parameters, such as cosmological parameters, based on a given set of observations. For conventional observables like galaxy clustering, the likelihood function underlying the data is well-defined. Through sampling this likelihood function, constraints on the parameters can be established [2].

An example of this in current literature is "*The CAMELS Multifield Dataset: Learning the Universe's Fundamental Parameters with Artificial Intelligence*" [3] which looked at data to carry out parameter inference within the CAMELS datasets.

2.1. This report

This study, in contrast to the aforementioned comprehensive investigation, focuses solely on a specific subset of simulations within the CAMELS Multifield Dataset (CMD). Specifically, it centres on the analysis of IllustrisTNG simulations, characterized as "*magneto-hydrodynamic simulations that track the evolution of gas, dark matter, stars, and black holes, while also incorporating magnetic fields*" [4]. Furthermore, the study narrows its scope to exclusively examine the 1P simulations within this dataset, wherein only one parameter is systematically varied at a time, resulting in the production of 2D maps.

Despite this refinement, the dataset remains substantial, with each map yielding 990 images or a total of 12,870 images in total. Each image is presented in a 256x256 grayscale format, a considerable volume of data for analysis.

3. The Data

The IllustrisTNG_1P simulations which we are looking at are carried out for a series of 13 maps: Gas density - **Mgas**, Gas velocity - **Vgas**, Gas temperature - **T**, Gas pressure - **P**, Gas metallicity - **Z**, Neutral hydrogen density - **HI**, Electron number density - **ne**, Magnetic field - **B**, Magnesium over iron - **MgFe**, Dark matter density - **Mcdm**, Dark matter velocity - **Vcdm**, Stellar mass density - **Mstar**, Total mass density - **Mtot**. The simulations producing these maps start from a series of initial conditions:

Ω_m	Representing the fraction of matter in the universe
σ_8	Controls the smoothness of the distribution of matter in the universe
A_{SN1} & A_{SN2}	Control two properties of supernova feedback.
A_{AGN1} & A_{AGN2}	Control two properties of black-hole feedback

Table 1: Cosmological parameters used for simulations and as labels for model training. Information from: [4]

These parameters will serve as the target labels for training our network, both to try to develop a model that can learn the underlying physical principles governing the universe simulations as well as enabling the analysis of the utility of various map types in relation to the cosmological parameters governing the universe. It is worth noting that among these parameters, two hold greater significance for analyzing our universe ($\Omega_M and \sigma_8$), while the remaining four are primarily instrumental in the generation of simulations but hold comparatively lesser relevance in practical applications.

4. Methodology

4.1. Dataset manipulation

The image data from the CAMELS simulations comes in the form of multiple .npy image files and a .txt labels file. These needed to be combined into a useable dataset for model training and this was done by creating a custom dataset: Camels_Dataset. This currently does not inherit a parent class dataset (eg. Torch Dataset) which could improve efficiencies through processes like automatic batchloading, GPU acceleration for data manipulation, multithread and subprocess data loading amongst other features that a collection of numpy arrays does not.

This dataset performs normalisation on the data using a max-scaling normalisation which can either be for the maximum value of the entire loaded dataset, or for each individual map type loaded. It also by default performs a random shuffle (although there is an option to turn this off) on the data before splitting into training, validation, and test data. In carrying out dataset manipulation (and training) there were also some memory issues, more about this can be found in the appendix A.

4.2. Data Augmentation

Data augmentation was looked at as a possibility for improving final accuracy, with several different data augmentation techniques used including image flipping (horizontal and vertical), random rotations and random resized crops. The data augmentation was carried out using both offline and online techniques, explained in A. Neither technique improved test accuracy drastically and for the current model actually decreased test accuracy. This process also increased processing time, therefore data augmentation was not carried out further. A discussion on why data augmentation might not have worked can be found in section 5.4.

4.3. Model Selection

As this is an image-to-value regression task, a feedforward deep neural network was used. The exact shape of this network was determined by hyperparameter tuning. This was done using scikit learn's ParameterGrid method [5], which gives all combinations of a given dictionary of lists of hyperparameters. The determination of the best model was then determined based on a 'cost' measure chosen to balance the size of the model (number of parameters) with the validation loss it achieved. This was done both to counteract overfitting possibilities of very large models, but also because we are trying to explain a physical phenomenon, so if something can explain (or predict) a certain physical phenomenon in a 'simpler' way than something else, as physicists we tend to prefer the simpler explanation. This is an example of Occam's razor. The cost function used was as follows: $\text{Cost} = \text{loss}^7 \times \text{Num Params}$

The total grid search size was well over 4000 combinations of models varying from 25,000 to several million parameters, a very large and somewhat unnecessary number given the very small improvement in the final costs that the model actually achieved due to this hyperparameter search. It is worth noting that for this task PyTorch was utilized as opposed to keras. The reason for this is twofold: The PyTorch library is generally faster than the Keras API (though not faster than the base theano/TensorFlow libraries) and PyTorch has native Windows GPU support whereas TensorFlow dropped Windows native GPU support after version 2.10. The hyperparameter tuning was carried out for just a single map, the Z map type, this was because carrying out hyperparameter tuning for all maps would've taken a very long time and all maps were of a similar structure and look, so a good set of hyperparameters for one map was likely to be a good set of hyperparameters on all other maps. A flow diagram of the simplified algorithm used to carry out the hyperparameter tuning can be seen below:

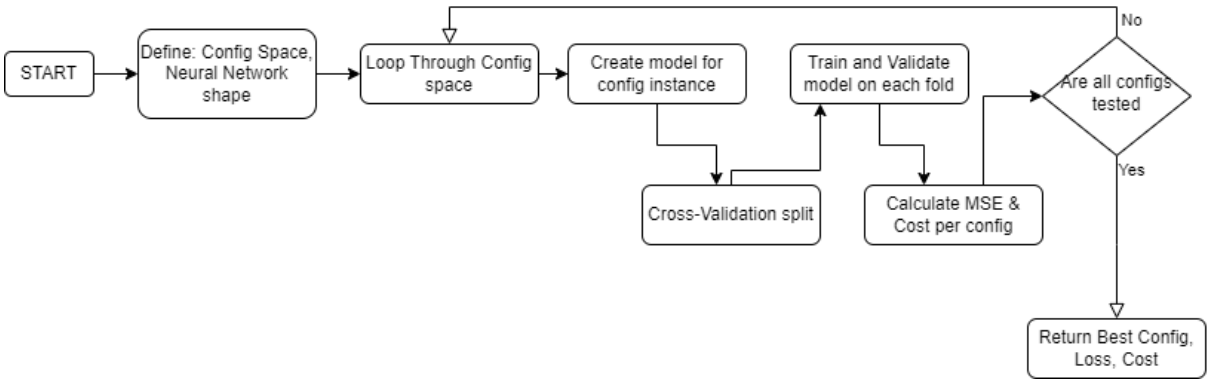
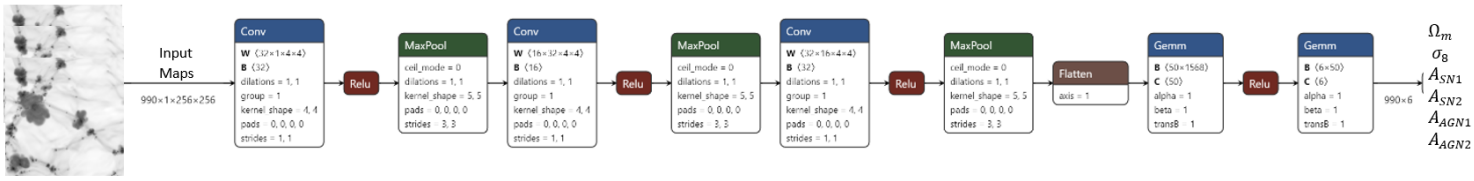


Figure 1: Flow chart demonstrating the hyperparameter tuning

Also included in this hyperparameter training were early stopping measures, which ensured models with low potential were discarded before taking too much processing time. This early stopping was done based on both the validation loss of the model as well as its 'cost' with respect to the best-achieved loss and cost up to that point. K-folds cross-validation was used to preserve the size of the training data and heuristically is shown to work well [6].

The result of this process was a model of structure:



Layer (type)	Output Shape	Param #
Conv2d-1	[-1, 32, 253, 253]	544
MaxPool2d-2	[-1, 32, 83, 83]	0
Conv2d-3	[-1, 16, 80, 80]	8,208
MaxPool2d-4	[-1, 16, 26, 26]	0
Conv2d-5	[-1, 32, 23, 23]	8,224
MaxPool2d-6	[-1, 32, 7, 7]	0
Dropout-7	[-1, 1568]	0
Linear-8	[-1, 50]	78,450
Dropout-9	[-1, 50]	0
Linear-10	[-1, 6]	306

Total params: 95,732
Trainable params: 95,732
Non-trainable params: 0

Input size (MB): 0.25
Forward/backward pass size (MB): 18.33
Params size (MB): 0.37
Estimated Total Size (MB): 18.94

Figure 2: Network architecture produced by hyperparameter tuning. The first graph is adapted from a Netron graph [7], where Gemm layers are an alternate name for dense, fully connected layers.

The model produced by the hyperparameter tuning process was actually very small, something which was wanted and enforced by the added cost function in the tuning. In doing the hyperparameter search, the model with the best pure loss not taking into account model size was also tracked, and this larger model (∼190k params) only outperformed the smaller 95k

parameter model by ∼2.5% which is a marginal difference for the vast difference in model size.

4.4. Training on a single map

After model selection, full training of the model was carried out. The length of time the model was trained for varied by the early stopping mechanism, more of which can be found in A, with a graph of the typical training cycle given in Fig 3.

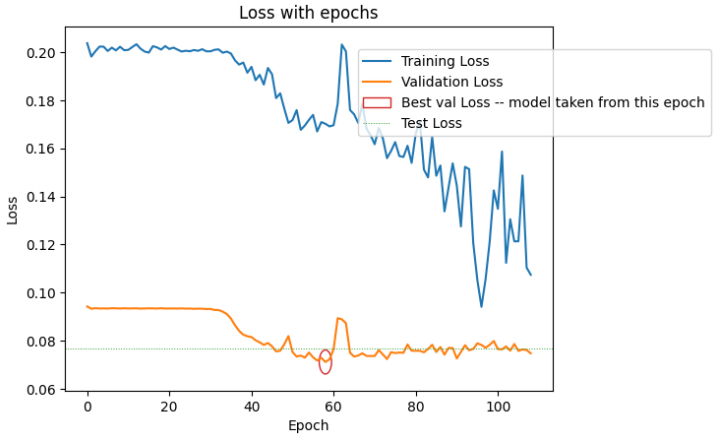


Figure 3: Graph showing losses on data for model training on map Z, also highlighting the early stopping algorithm

A separate model was trained on each of the 13 maps. The models were then tested on both the test data of the map they were trained on as well as the test data of all other maps. This was done to analyse if any map types were of increased importance in the simulation of the universe. Using this run, each model's percentage errors on the individual parameters were also calculated. This was done to see how close to the true values the models were in a more meaningful way than just mean squared error and also to see if certain parameters were easier or harder for the model to predict.

4.5. Training on multiple maps

Following the initial training of all models on a single map, subsequent training sessions were conducted using multiple maps concurrently. This process entailed the utilization of two distinct techniques. The first method involved concatenating arrays of images and their corresponding labels sequentially, followed by feeding them into the model in a randomized sequence. Conversely, the second approach entailed incorporating additional maps and their respective data as supplementary channels to the input layer. Notably, all channels maintained uniform label values throughout the process.

Both methodologies were implemented across 20 distinct combinations of maps, ranging from 2 to 5 maps in total, resulting in a total of 80 models. As illustrated in Fig. 4, employing the technique of incorporating extra channels exhibited markedly superior performance in terms of test loss. Furthermore, this method yielded a notable reduction in running time, amounting to nearly one-third of the original duration

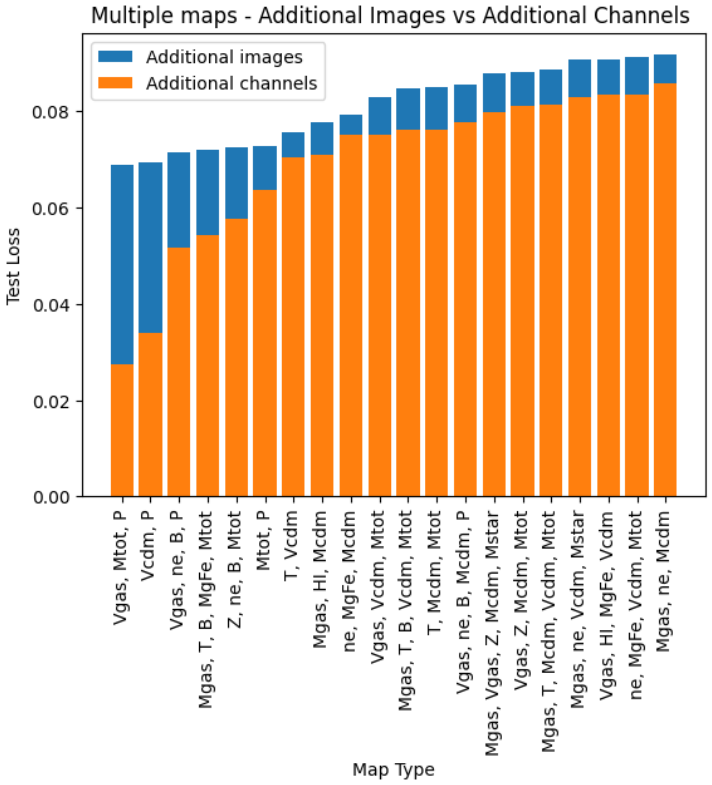


Figure 4: Graph showcasing the difference in performance on test loss for different approaches to adding additional maps to training data

Following the identification of the superior approach for incorporating additional maps into the training process, a broader exploration of map combinations became feasible. This expanded investigation encompassed a total of 685 distinct combinations, spanning a range of lengths from 2 maps up to 12 maps. There was skepticism about whether 685 out of 8177 combinations. ($\sim$8.4%) was enough to draw meaningful conclusions. For this reason, a statistical analysis was carried out to determine the confidence level [1].

$$n = \frac{n'}{1 + \frac{Z^2 \times p(1-p)}{E^2 N}}; \qquad n' = \frac{Z^2 \times p(1-p)}{E^2} \tag{1}$$

Where: n = sample size, Z = Z-score, p = population proportion, E = margin of error, N = total population size Here we assume unknown data so p = 0.5 and we want a 95% confidence level (standard used in statistical analysis) so E = 0.05.

$$n' = \frac{1.96^2 \times 0.5(1-0.5)}{0.05^2}; \qquad n' = 384.16 \tag{2}$$

$$n = \frac{384.16}{1 + \frac{0.96^2 \times 0.5(1-0.5)}{0.05^2 8177}}; \qquad n = 379.88 \approx 380 < 685 \tag{3}$$

[1]Note, the formula used here does have some assumptions: Normalised data, independent data, random sampling, and a known population standard deviation. However, as an approximation for testing whether we have enough data to be significant, this formula is sufficient.

Therefore conclusions drawn from our sample size of 685 are well within the 95% confidence level to be true of the full dataset, so we can meaningfully say things about the full dataset from our sample. Another added reason why we can draw meaningful conclusions is that the sampling is random, but also representative of the entire set, with lengths of all combinations included.

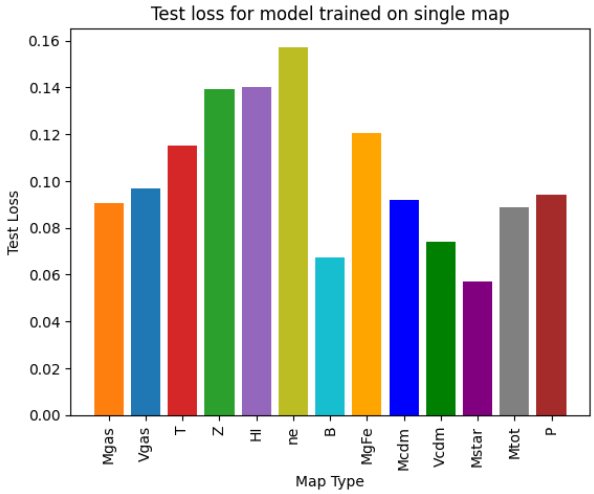
To get around this potential issue as well, the multiple map analysis was also carried out for just 2 map combinations. This was done because there are a lot fewer combinations of these (78) so all of them could be tested in a reasonable time frame and data analysis could be done for this entire population.

5. Results

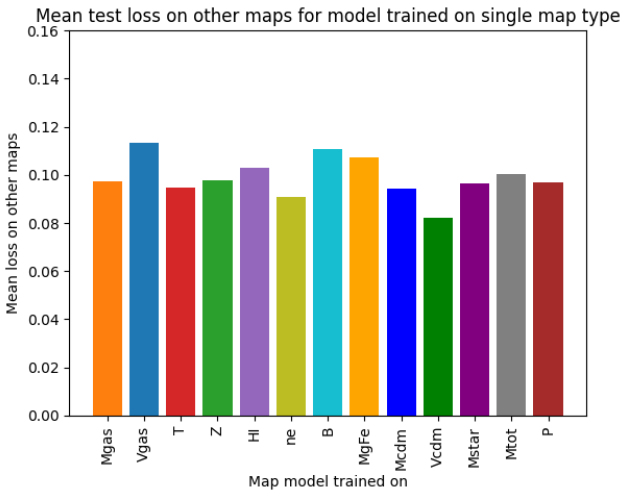
5.1. Single Map

Test loss, calculated as mean squared error from true values can be seen in Fig. 5a for each model trained on a single map type. From this figure we can see the model trained on map MStar achieves the lowest test loss at around 0.06 or around 6% error on the physical variables (or a 94% accuracy). This is a reasonably small error and shows that our model is learning something about the parameters simulating this map. Some of the other models trained on other maps do significantly worse, with anything up to a 16% total error. This could be taken as a sign that this map (or similarly badly performing ones) are less useful for predicting the physical variables of the simulations. However, having run this multiple times throughout building the code, which map performs well or badly for this test seems to be completely random, changing very often. This can be especially seen for map Z on which the hyperparameter training was carried out on. For the model in Fig. 5a, with the same architecture with all the same hyperparameters and model architecture performs achieves a test loss of 0.14, whereas in Fig. 3, the model trained on map Z achieves a loss of under 0.08, a quite substantial difference in performance.

Following this investigation, the test loss for a model trained on one type of map and then tested on all other maps was generated as can be seen in Fig. 5b. This again shows some randomness and also highlights that just because one map does well on its own test data does not mean it will perform well on other maps. It also shows the complete opposite as well, just because the performance was poor on its own test data, does not mean it will perform badly on other map types (eg. map: ne). From Fig. 5b we also cannot draw any conclusions about which maps generalise better to being able to predict the physical variables for other maps, with no map drastically outperforming any others.



(a) Test losses on maps trained on for models trained on single maps



(b) Average test loss for models trained on one map and tested on all other maps

Figure 5

Using all these trained models, the percentage error on the individual physical variables was also calculated using the formula: Error $= \frac{|y_{predicted} - y_{true}|}{y_{true}} * 100\%$. The results of this can be seen in Fig. 6, showing that all models were good at predicting σ_8 the parameter controlling the smoothness of matter and were particularly bad at predicting A_{SN1}&A_{SN2} the parameters controlling supernova feedback. Following making this graph, it was also theorised whether the variance in the test data physical variables also had anything to do with the error in certain values and while we can see some correlation, it is definitely not the main reason for the models doing well on certain variables and very poorly on others.

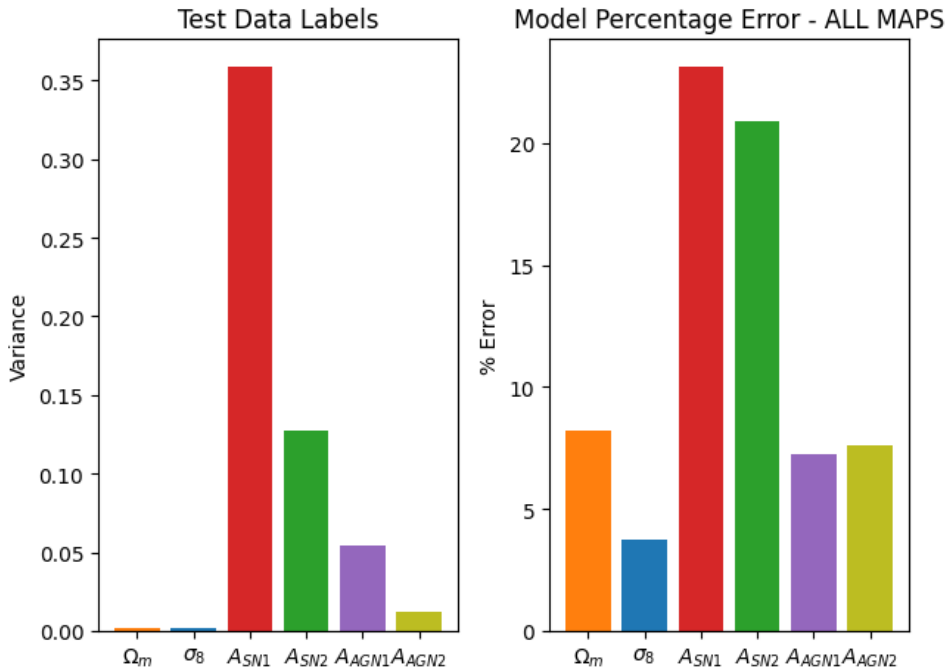


Figure 6: Left: Variance in test data labels. Right: % Error of model predictions on the physical variables for all models on their own test data

5.2. Multiple Maps

Following the investigation into models trained on single maps, 685 different maps were then trained on different multiple-map combinations. Fig. 7 shows the 20 best performing maps on test loss for all 685 maps tested. We can see that the best maps here perform marginally better than the single map with an error at best of 4% as opposed to 6% in Fig. 5a. Considering this was performed for a full 685 combinations and the best test error of any combination was only 2% better than a single map, even when being fed up to 12 maps at a time, explains why data augmentation was not helping for this dataset. One possible reason for this is explored in section 5.4.

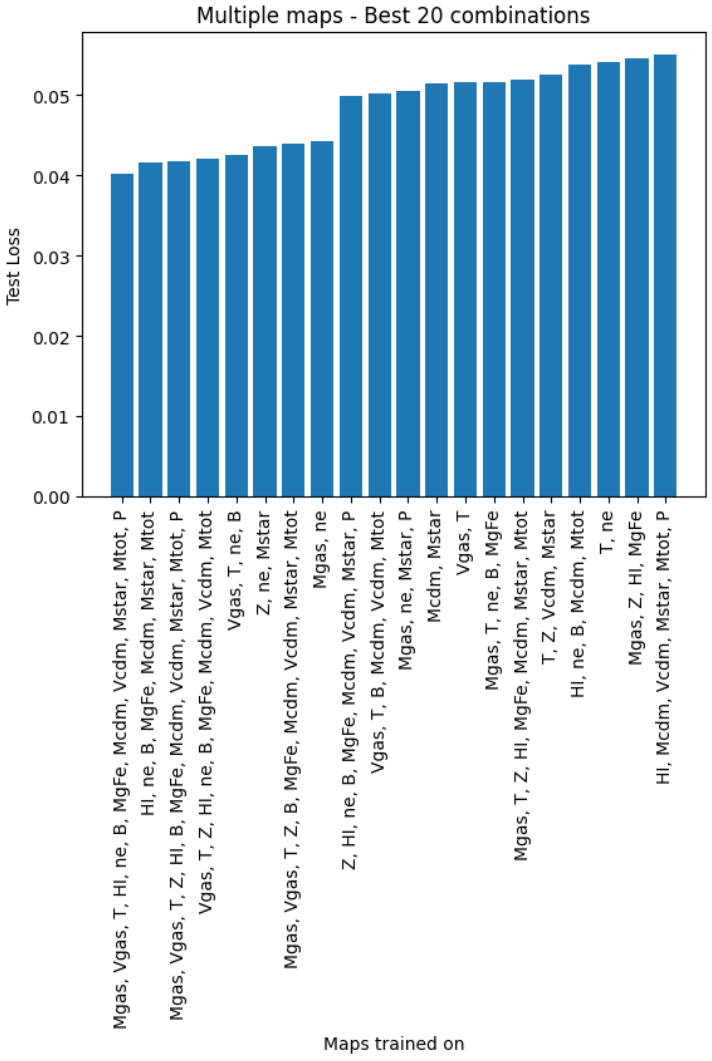


Figure 7: 20 Best combinations by test loss from the 685 total tested combinations

The utility of each map type was also looked at for multiple map combinations through 2 different methods, the number of occurrences of each map type in the best 50 map types as well as the average loss of all tested combinations that include a specific map type as can be seen in Fig. 8. From the occurrences graph some differences between the best maps can be seen, however when plotting the average loss for all 685 tested map combinations so we get a reasonable confidence level we can see very little variation between the map types, suggesting one map is not significantly better than any other.

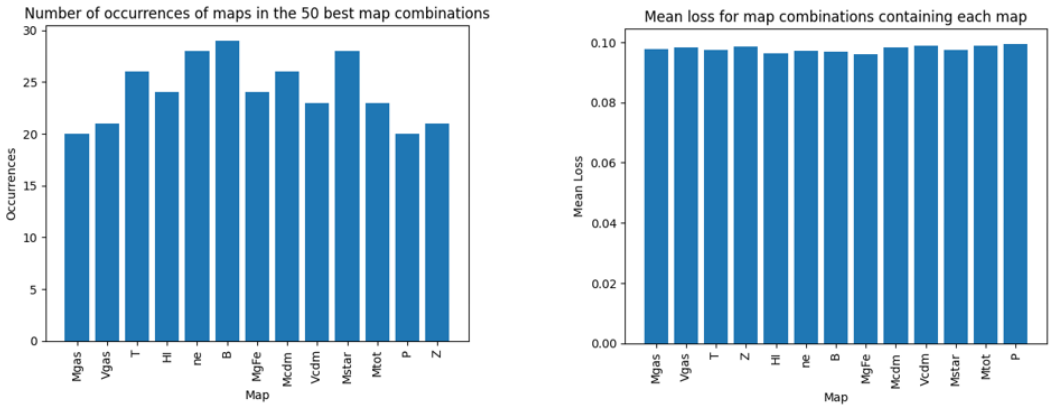


Figure 8: Comparing usefulness of map type, Left: The number of occurrences of each map contained within multiple map combinations for the 50 best performing combinations. Right: Average loss on all models trained on all map combinations containing each map type

An investigation into whether an increasing number of maps improved test loss (and therefore accuracy) was also carried out and can be seen in Fig. 9. For number of maps 1 to 11 there is no clear trend, however we do see a sudden drop off in loss for 12 maps. This could suggest that 12 maps does actually improve performance, but the difference in error is only around 2% from the other number of maps which is not substantial enough to call statistically significant.

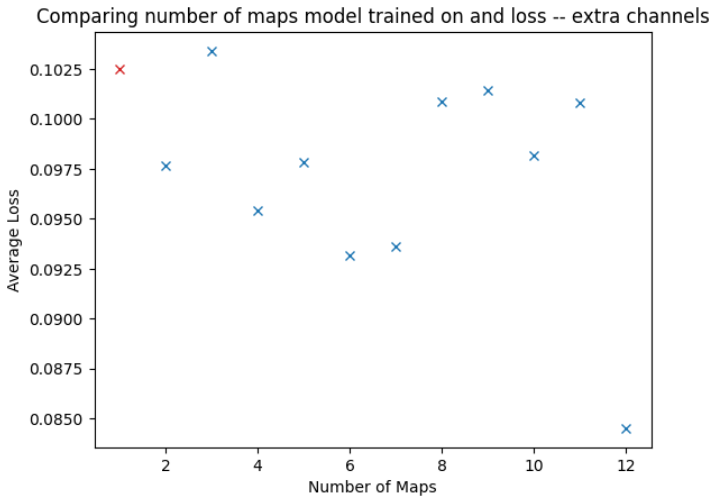


Figure 9: Average test loss with increasing number of maps included in training

5.3. 2 Map Only

As described in 4.5, the same analysis as Fig. 8 was also carried out for only 2 map combinations. This can be seen in Fig. 10 where much more of a difference between the map types can be seen with maps: ne, Mcdm and B all occurring most frequently in the best map combinations and all having some of the lowest errors as well. The difference in this error is only ∼1-2% which is

still small, so to claim any of these maps are significantly better than any others would still be unwise here without further justification and testing.

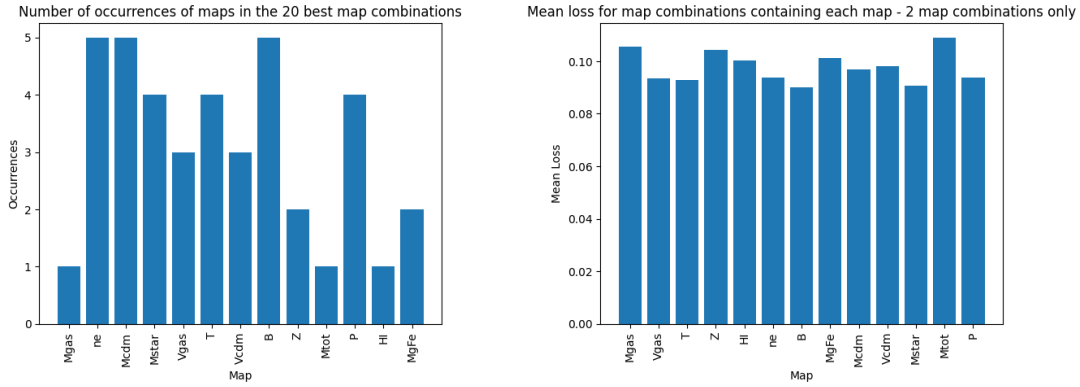


Figure 10: Comparing usefulness of map type for 2 map combinations only, Left: The number of occurrences of each map contained within 2 map combinations for the 50 best performing combinations. Right: Average loss on all models trained on 2 map combinations containing each map type

5.4. The issue with accuracy

The main issue in training the models to achieve a good error (under 5% for all parameters) could be the lack of labels rather than the lack of maps. There are 12870 maps to train the model on which is large enough to train a good model, however, all these images are only formed from 66 different sets of labels, some with a very small variance in their values across all 66 different labels. This is a very small number and so training a model on this little amount of label data leads it to have a decreased accuracy. This is also the reason data augmentation does not work, there are already enough input images and we cannot carry out data augmentation on the labels as this would produce different maps.

6. Summary

In this paper, we have trained multiple neural networks on the CAMELS illustris 1P dataset achieving a best accuracy on the physical simulation parameters of $\sim 4\%$ from training models both on single map types as well as multiple maps at the same time. We also looked at whether certain map types had greater importance or utility in predicting the physical parameters and found that no map had a significantly raised importance.

References

[1] Readthedocs.io. (2020). CAMELS — CAMELS 0.1 documentation. [online] Available at: https://camels.readthedocs.io/en/latest/ [Accessed 8 Mar. 2024].

[2] Parameter Estimation. (2016). CAMELS - Parameter estimation. [online] Available at: https://www.camel-simulations.org/parameter-estimation [Accessed 8 Mar. 2024].

[3] Villaescusa-Navarro, F., Genel, S., Angles-Alcazar, D., Thiele, L., Dave, R., Narayanan, D., Nicola, A., Li, Y., Villanueva-Domingo, P., Wandelt, B.D., Spergel, D.N., Somerville, R.S., Zorrilla, M., Mohammad, F.G., Hassan, S., Shao, H., Digvijay Wadekar, Eickenberg, M., Kaze and Contardo, G. (2022). The CAMELS Multifield Data Set: Learning the Universe's Fundamental Parameters with Artificial Intelligence. The Astrophysical Journal Supplement Series, [online] 259(2), pp.61–61. doi:https://doi.org/10.3847/1538-4365/ac5ab0.

[4] Readthedocs.io. (2021). Description — CAMELS Multifield Dataset 1.0 documentation. [online] Available at: https://camels-multifield-dataset.readthedocs.io/en/latest/data.html#d-maps [Accessed 8 Mar. 2024].

[5] scikit-learn. (2024). sklearn.model_selection.ParameterGrid. [online] Available at: https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.ParameterGrid.html [Accessed 8 Mar. 2024].

[6] Dino, L. (2022). K-fold CV — Hyper-parameter tuning in Python - Little Dino - Medium. [online] Medium. Available at: https://medium.com/@24littledino/k-fold-cv-hyper-parameter-tuning-in-python-4ad95880e477 [Accessed 16 Mar. 2024].

[7] lutzroeder (2024). GitHub - lutzroeder/netron: Visualizer for neural network, deep learning and machine learning models. [online] GitHub. Available at: https://github.com/lutzroeder/netron [Accessed 18 Mar. 2024].

A. Appendix: Algorithms and Issues

Early Stopping Algorithm:

The early stopping algorithm consisted of two distinct components. Firstly, early stopping is based on the model's failure to demonstrate improvement in validation loss. Secondly, during hyperparameter tuning, the algorithm halted the training of models with low potential to prevent them from consuming excessive computational resources.

In the first method, a patience variable was assigned, representing the number of epochs the model would undergo training regardless of a lack of improvement in validation loss. This value would reset each time the model achieved a validation loss either lower than or within a small tolerance of the previous best validation loss.

The second method also utilized a patience parameter, but this time the algorithm waited for the the model to get a cost metric (as detailed in the main text) lower or within a slightly larger tolerance than that of the previous best model. This approach ensured that models exhibiting either significantly higher loss compared to the best previous model or those with exceedingly large parameter counts were not trained excessively, thereby speeding up the hyperparameter tuning process.

Online vs Offline data augmentation:

Online data augmentation was implemented to apply augmentations to maps within each training batch, with a specified probability. Consequently, every map received different augmentations each time a batch was formed. This approach offers the advantage of avoiding dataset size inflation to alleviate memory and disk space constraints. However, it necessitates training models for more epochs to achieve comparable effects compared to offline data augmentation.

Offline data augmentation, conversely, entails applying the same augmentation techniques to all images simultaneously and then appending them as additional images to the dataset. This method substantially augments the number of maps available for model training.

Memory Issues:

While still only using a small proportion of the entire CMD dataset, we still have a large amount of data and this did lead to some memory issues. The size of the data files we are using here are:
$256px \times 256px \times 4$ byte greyscale float$/1e6 \approx 0.26$mb per image
990 images per map $\times\ 0.26mb = 257.4mb$ per map
13 maps total $\times\ 257.4mb \approx 3.3Gb$ total size

For this project, I am running 16Gb of RAM which means when performing data manipulations using the above dataset there were occasional RAM overflows or when utilizing GPU accelerated training on a CUDA enabled NVIDA 4060 laptop GPU with 8Gb of vRAM there were occasional CUDA vRAM overflow errors. This was accounted for and mitigated

by manually triggering python garbage collection after deleting variables that were no longer needed to free up RAM space as well as manually clearing the vRAM cache using pytorch's torch.cuda.empty_cache() method. While this helped reduce the frequency of the errors, it did not completely solve the problem, especially if I was trying to do anything else on my laptop while the code was running which could use up additional memory space. Therefore I also implemented an autosave feature, where after every milestone (whether that be hyperparameter config tested or model trained on specific map combination) the important information for that run was saved to a file using a pickle dump. This ended up being helpful not just for memory crash issues, but also meant that runs could be stopped and reloaded without redoing what had already been done in a previous run.