Classification of Cosmological Data with Machine Learning

Vladimirs Zenko

Abstract

As the progress moves forward in the field of observational cosmology, the amount of data modern technologies allow to collect begins to exceed the volume of data we can process and analyse in a specific time frame. For that reason, introduction of new methods and techniques of data analysis is one of the hot topics in scientific communities around the world. This paper focuses on possible implementations of Deep Learning algorithms to classify cosmological data and adaptation of these algorithms for wider range of data features. Data sets created by state-of-art computer simulations were used to observe how efficient neural networks are at predicting features of the samples and how can the performance be improved using domain adaptation techniques.

Contents

Chapter 1

Introduction

Over the past years cosmology, astrophysics and observational astronomy have been in the spotlight among scientific communities. Such events as launch of James Webb space telescope, first ever image of black hole have drawn a lot of attention to the star-gazing areas of physics due to rapid technological progress involved there.

While new advanced technologies allow us to capture larger volume of data, manual approach to processing the date becomes less relevant and new, more efficient techniques need to be implemented.

Area of Machine Learning, while still on its early stages and yet to uncover its full potential, is one of the most modern and promising candidates for automation of numerous scientific processes. It is particularly useful for classification of data gathered by astronomers.

CAMELS Multifield Dataset[1] is a publicly available collection of magneto-hydro-dynamics and N-body simulations of galaxies brought to reality by the CAMELS[2] project. The data is classified into three different simulations. **IllustrisTNG** represents magneto-hydrodynamic simulations of evolution of gas, dark matter, stars, black-holes and magnetic fields.Contains 13 fields. **SIMBA** - hydrodynamic simulations of evolution of gas, dark matter, stars, and black-holes. Contains 12 fields. **N-body** - gravity only simulations of the evolution of dark matter. Only contains one field. Each element has a set of labels attached to it. For this project, only Ω_M (fraction of matter in the Universe) and σ_8 (smoothness of the distribution of matter in the Universe) parameters for M_{tot} and T fields were considered.

In this project we study how machine learning applies to classification of this data and what difficulties it presents. Namely, we study possibilities of universal techniques for classification of object defined by broad range of parameters.

1.1 Aims

This project is centered around implementation of machine learning algorithms for classification of computational simulations of cosmological data from the CMD.

In the reference paper[3] that we used as background for our project it was shown that deep learning model used to classify data performs well on the data which was used to train it, but poorly if evaluated on data from different simulation.

We compared performance of models trained on IllustrisTNG simulation of CMD dataset and evaluated on SIMBA simulation, and vice versa. While both simulations focus on same cosmological objects, behind modelling stands different physics. The aim of the project is to study of how model trained on one simulation performs when evaluated on the other and how the results can be manipulated.

1.2 Background knowledge

1.2.1 Neural Networks

The core element behind any machine learning problem. Reproduces human learning process as seen by a computer. The most basic yet efficient type of neural network consists of inter-connected, or dense, layers and is called a multi-layer perceptron.

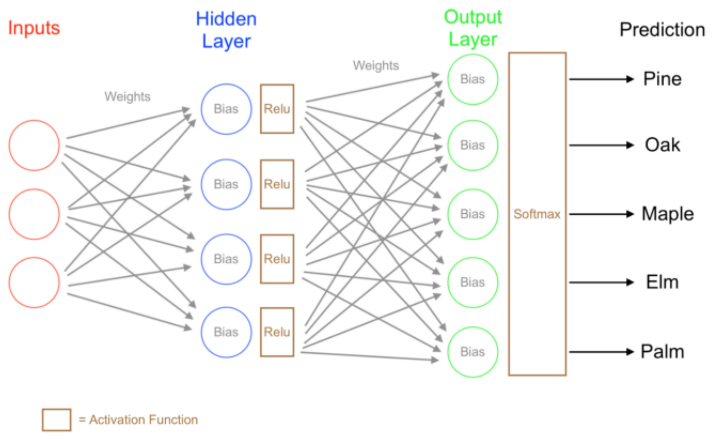


Figure 1.1: Basic dense neural network architecture example on classifying trees. Source: [4]

Neurons are the main building block of any neural network. They take vector of input values and apply bias to it. Bias is a constant that offsets the initial values in order to make it easier for activation function to normalize the output. Transformed value is then passed to the next level through weights. Weights are the main source of learning for a neural network. During training, the multi-layer perceptron passes inputs through random weights and compares the output layer prediction to the ground truth value for the data sample. It then propagates backwards through layers adjusting weights to bring prediction closer to the ground truth value. neuron outputs must be normalized to represent the data (i.e. probability of belonging to a class will be an output between 0 and 1). This is what activation function does.

Error in predictions is parameterized by the **loss function**. It is defined as average loss (error) over samples of a dataset. The goal of supervised learning is to minimize loss function. Common types of those are **Mean Absolute/Square Error** (MAE, MSE) for **regression** (predict value for certain feature of the sample) or **Cross Entropy** (divide samples into classes based on features) for **classification**.

We shall not delve deep into notation and definitions of dense layer since this does not bring us closer to the point of our project.

1.2.2 Convolutional Layers

Dense layers are purely feature interpreting layers, but for most of the real-world data it is also necessary to refine the samples in order to implicitly specify range of features the model should prioritize during training.

Feature extraction is done using Convolutional layers. This type of layer convolves the input image (2D tensor) with a set of filters (kernels), represented by a tensor of same shape but smaller dimensions to extract fundamental features. The kernel are composed of smaller constant parameters as compared to the input images, using these as transformations applied to image matrix values.

In a convolutional layer, a neuron is only connected to a local area of input neurons instead of full-connection so that the number of parameters to be learned is reduced significantly and a network can grow deeper with fewer parameters [5].

Dense layers can be substituted with convolutional ones, but in general it is worth to use a combination of both to maintain good balance between extraction and interpretation of features.

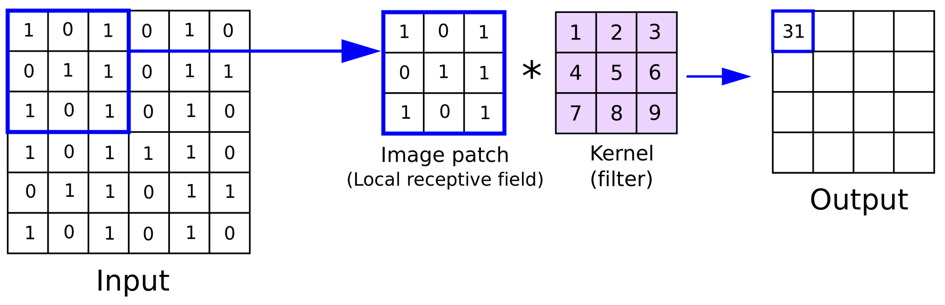


Figure 1.2: Basic convolutional layer operation. Source: [6]

Convolutional layers are frequently used in combination with Pooling layers, which also reduce features of the input tensor.

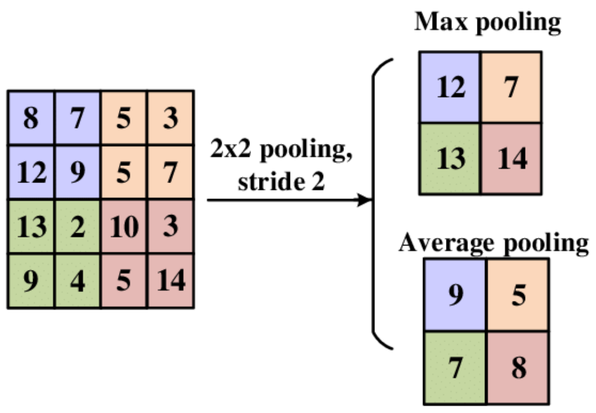


Figure 1.3: Max and Average pooling. Stride is the number of pixels shifts over the input tensor (2D matrix here). When the stride is two kernel moves one pixel at a time. Source: [7]

1.2.3 Data Augmentation

Data augmentation is simply a technique of increasing the dataset by introducing synthetically generated data with features alike those in the initial set. This is usually done by applying filters to the dataset elements and appending them to the dataset. Reason for this is to account for overfitting of the model. When model is overfit it interprets the features too elaborately, so while its performance on the training data gets better, loss for the test data will increase. Having more data reduces probability of overfitting.

Note that only training data needs to be augmented, since test data has to resemble the real world data to simulate what the model will encounter when being released and applied in practice.

1.2.4 Data Splits

When data is to be used for a machine learning problem, it is usually split into three separate components. **Training** subset, the largest of the three, is used by the model to learn. **Testing** subset is used to evaluate the model after the training is done. Since some models can be heavy and take much time and resources to run, it is inefficient to rely on post-training evaluation, and **Validation** subset is introduced. Models are always trained for a certain amount of epochs. One epoch is full iteration through the input data. If supplied, model is evaluated on the validation subset after each epoch, allowing to dynamically monitor performance of the model at each stage of training.

1.3 Technical tools

Overview of the packages used for the project.

1.3.1 TensorFlow

This project is mainly computational. All neural networks are defined using Python TensorFlow package [8]. Data used in the project was prepared as TensorFlow dataset by the project supervisor, Dr Adam Moss.

Keras API of ThensorFlow is very convenient for defining neural networks for supervised learning and has rather straightforward syntax which allows to focus on results rather than technical aspects of the projects.

1.3.2 Albumentations

Albumentations [9] is flexible Python package for image augmentation. Rotational transformations from the package were used to enhance CMD dataset with additional synthetic data.

1.3.3 Adapt

ADAPT [10] - Awesome Domain Adaptation Python Toolbox. The name speaks for itself, this Python package has wide collections of tools for parameter, feature or instance based tools to adapt our model for datasets based on different simulations.

Chapter 2

Method

This chapter is an overview of the method used to conduct research. Process of recreating a model used in the reference paper is described and details on how domain adaptation techniques were implemented to that model are provided. We also consider complications and difficulties encountered during the process and possible solutions to those.

Coding part of the project can be accessed at GitHub repository:

https://github.com/VladZenko/CosmologyClassificationDL

2.1 Data Processing

In any machine learning problem, the most important part is the data itself. Before the model can be trained on the data, certain procedures need to take place for dataset elements to take desired shape. This governs efficiency of training in terms of computational resources and precision of the final model. However, one has to be careful when preprocessing the data. While manipulations to enrich the data are important, it is crucial to preserve key features defining the data.

2.1.1 Loading the data

As mentioned earlier, the CMB dataset has been prepared into TensorFlow dataset. Hence, we can use same syntaxis as used for examples in TensorFlow documentation.

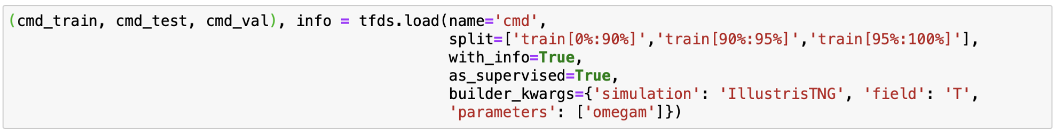
Firstly, install the package, **astro_datasets** [11]. This can be done using **pip** via command prompt:

pip install astro-datasets tensorflow

Then, import tensorflow_datasets module, that is a ready-to-use collection of TensorFlow datasets.

import tensorflor_datasets as tfds

Scene is now set up to load the data.

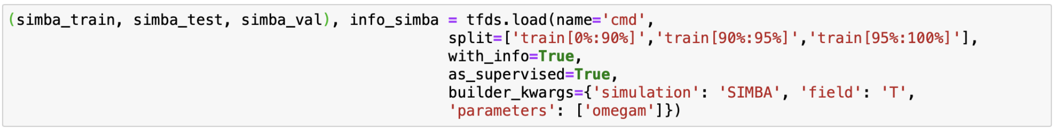


```
(cmd_train, cmd_test, cmd_val), info = tfds.load(name='cmd',
                                                  split=['train[0%:90%]','train[90%:95%]','train[95%:100%]'],
                                                  with_info=True,
                                                  as_supervised=True,
                                                  builder_kwargs={'simulation': 'IllustrisTNG', 'field': 'T',
                                                  'parameters': ['omegam']})
```

Figure 2.1: Upload of the CMD dataset IllustrisTNG simulation, gas T field and Ω_M parameter.

The called function returns three separate dataset splits and one info variable, which contains all the information about the dataset loaded. Arguments for the tfds.load() function specify what kind of data will be uploaded.

Same procedure was then repeated for SIMBA field.



```
(simba_train, simba_test, simba_val), info_simba = tfds.load(name='cmd',
                                                  split=['train[0%:90%]','train[90%:95%]','train[95%:100%]'],
                                                  with_info=True,
                                                  as_supervised=True,
                                                  builder_kwargs={'simulation': 'SIMBA', 'field': 'T',
                                                  'parameters': ['omegam']})
```

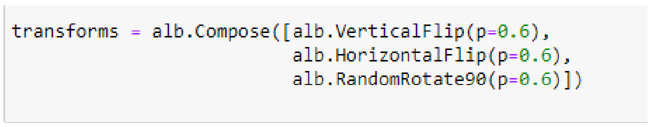
Figure 2.2: Upload of the CMD dataset SIMBA simulation, gas T field and Ω_M parameter.

After loading the data the next step is to make it suitable for input of the deep learning model.

2.1.2 Augmenting the Data

In order to prevent overfitting and increase generality of models performance, data augmentation pipeline was introduced.

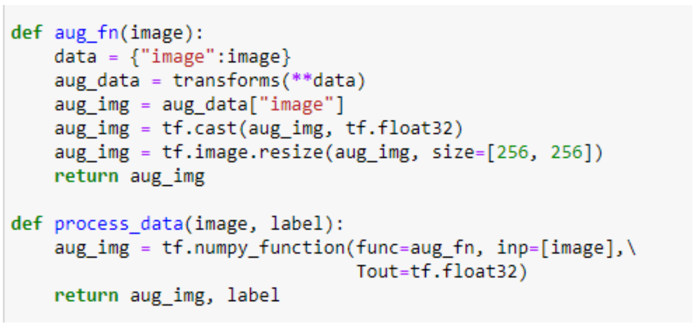
One of the problems encountered was that it does not seem to be possible to append specific elements to a TensorFlow dataset, so applying transformations to an image and appending it to the list with corresponding label was not an option. A simple yet elegant solution was to use **.repeat(count = *number of repeats*)** method. It is applied to a dataset and repeats the dataset specified amount of times. This method saves time since no loops are needed and images are generated together with corresponding labels. Last thing to do with the enlarged dataset is to pass it through augmentation pipeline, applying transformations to elements. With probability specified in the pipeline, the image will be transformed.



```python
transforms = alb.Compose([alb.VerticalFlip(p=0.6),
                          alb.HorizontalFlip(p=0.6),
                          alb.RandomRotate90(p=0.6)])
```

Figure 2.3: Augmentation pipeline defined using 'Albumentations' library. generates variable containing the transformations. These include vertical flip, horizontal flip and 90° rotation in random direction. All of thoese have 60% probability of being applied.

However, because Albumentations defined augmentation pipelines are usually applied to **numpy** arrays and data for this project is confined within a TensorFlow dataset (generally a **dict** object), applying transformations to the data uses specific syntax.



```python
def aug_fn(image):
    data = {"image":image}
    aug_data = transforms(**data)
    aug_img = aug_data["image"]
    aug_img = tf.cast(aug_img, tf.float32)
    aug_img = tf.image.resize(aug_img, size=[256, 256])
    return aug_img

def process_data(image, label):
    aug_img = tf.numpy_function(func=aug_fn, inp=[image],\
                                Tout=tf.float32)
    return aug_img, label
```

Figure 2.4: Functions used to augment the data

First function, **aug_fn**, is responsible for image transformations. It takes input of an image, embeds it into a dict object, applies transformations to it and returns transformed image of data type *float32* with dimensions 256×256 pixels.

The second function applies the previously defined augmentation function to the dataset. Does so by applying **aug_fn** to dataset image as numpy function. Returns transformed image and corresponding unchanged label.

Because **tf.numpy_function()** resets shape of an object, images are brought back into shape [256, 256, 1] by TensorFlow's **.set_shape()** method.

2.1.3 Normalizing the Data

Depending on what feature we want the data to represent, it should always be refined to take account of outliers and elements that may cause disruption to the training process. In case of images, these can be dead pixels or excessively bright pixels that contrast with the total picture.

this was done by taking inverse hyperbolic sine of an image pixel values and subtracting a constant from it to take account of problematic values. Same effect can be achieved by taking natural logarithm of the data with constant 1 added to each pixel value.

image = tf.math.asinh(image) - 26

As a final touch, data subsets were batched with 128 elements per batch and prefetched. Prefetching allows for smoother and faster runtime of the code as it always keeps an element (batch in this case) ready to be passed to the model for training, so that after training of one batch no time is wasted to pass the second one to the model.

The data is now augmented, normalized, batched, prefetched and ready to be passed to the model for training.

2.2 Defining the Deep Learning Model

The deep learning model is defined using TensorFlow's Keras API. Keras Sequential model is used to recreate architecture used in the reference paper.

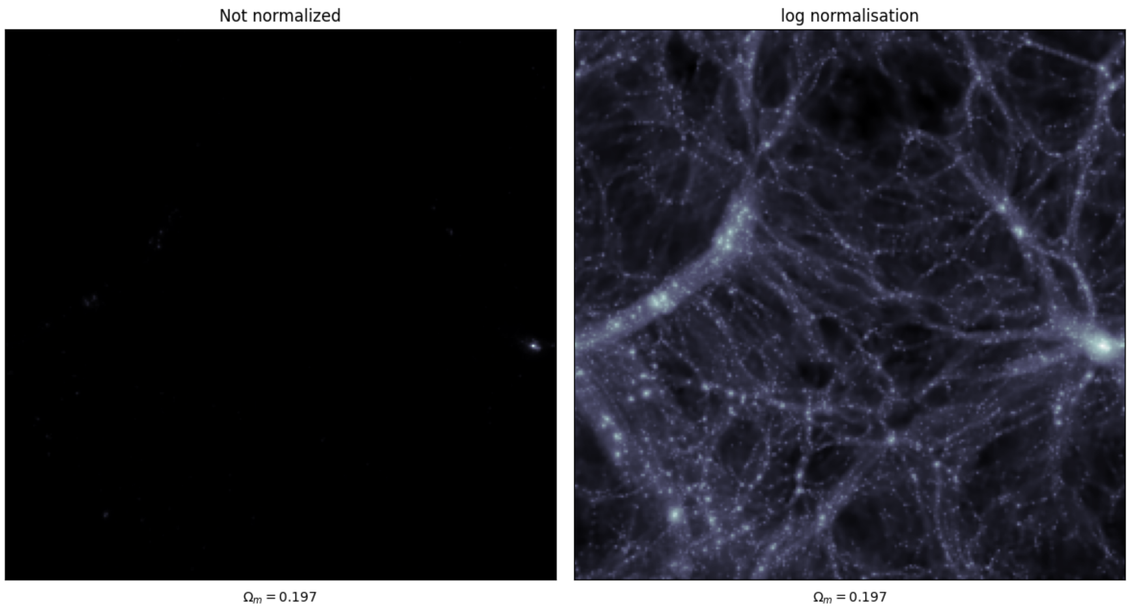


Figure 2.5: Sample element from CMD dataset. IllustrisTNG N-body simulation, Ω_M parameter of M_{tot} field. Left image is not normalized, just raw input data. Right image has **tf.math.log()** normalization applied to it.

The architecture consists in a set of 6 blocks, where each block follows the structure CBACBACBA, where C, B, and A are convolutional, batchnorm, and LeakyReLU layers, respectively. In each block, the first two convolutional layers have kernel size of 3, stride 1 and padding 1, while the last layer has kernel size of 2, padding 0 and stride 2. The first convolutional layer of the first block is not followed by a batchnorm layer. After the six blocks, we use a smaller block with CBA where the convolutional layer has kernel size 4, stride 1 and padding 0. The output of the last block is flattened and passed into two fully connected layers. [3]

Dropout layers with rate 0.2 are added after the Flatten layer and the last LeakyReLU activation layer. Also worth noting that normalization layers are added before activation layers which is a common practice since it normalizes the initial values rather than ones passed through activation function.

Difference between ReLU (rectified linear units) and leakyReLU activations is that ReLU is zero for all values below threshold, while LeakyReLU adds little slope to values below zero.

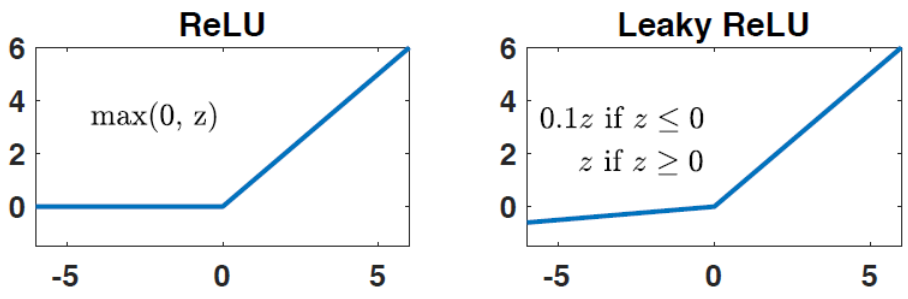
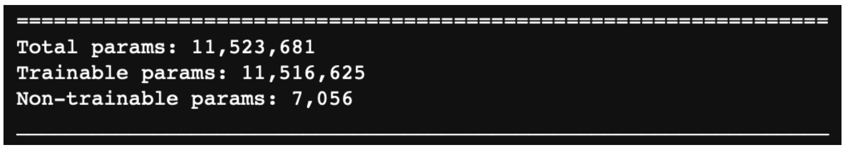


Figure 2.6: ReLU vs leakyReLU. Source: [12]

’Saturation’ corresponds to the problem of exploding and vanishing gradients. When gradients explode, the activation functions return extremely big values and the update in model weights is too large and the model loss increases and it cannot learn to solve the problem. When gradients vanish, the update in the model weights is too small so that weights cannot be updated properly, increasing the loss.

ReLU is used to overcome saturation since all values over 1 are left unchanged and values below 0 are set to 0. LeakyReLU in turn allows to account for negative values as well.



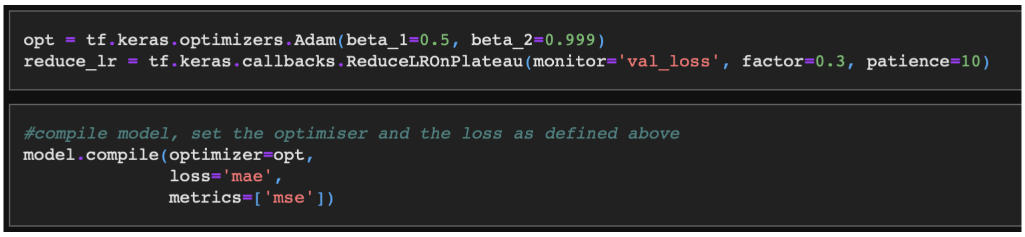
```
================================================================
Total params: 11,523,681
Trainable params: 11,516,625
Non-trainable params: 7,056
________________________________________________________________
```

Figure 2.7: Parameters of the model. Parameters encode neurons, weights and biases of the model.

Before compiling the model it is necessary to define the optimizer. Optimizers are used to set the learning rate of the model that encodes how the weights are updated.

If the learning rate is too high, the model overshoots the minima point of the loss function. If the learning rate is too small, it takes too long to reach the minima.

Adam is an optimizer which takes gradients of parameters defining the loss function and uses them to scale the learning rate at each step of learning. Precisely, Adam estimates 1^{st} and 2^{nd} momenta of the gradient. The first moment is an estimate of the historical momentum, and the second moment is used to rescale the learning rate [12].



```
opt = tf.keras.optimizers.Adam(beta_1=0.5, beta_2=0.999)
reduce_lr = tf.keras.callbacks.ReduceLROnPlateau(monitor='val_loss', factor=0.3, patience=10)
```

```
#compile model, set the optimiser and the loss as defined above
model.compile(optimizer=opt,
              loss='mae',
              metrics=['mse'])
```

Figure 2.8: Optimizer, learning rate reducing callback and compiled model.

β_1 and β_2 are decay rates of the 1^{st} and 2^{nd} momenta respectively.

We set β_1 to below-default value (0.5) and define a callback that will reduce the learning rate by constant factor (0.3) after 10 epochs since increase in validation loss (this is what patience argument is for). This is to assist the optimizer with adjusting the weights.

Loss function of Mean Absolute Error was used with Mean Squared Error as an additional metric. Model is then compiled.

Additionally, checkpoint callback was defined to only save the model when validation loss decreases to ensure best possible result.

2.3 Extracting results

Data is fed to the model usint **.fit()** method with the training subset, number of epochs, list of callbacks and the validation subset as arguments.

Model is loaded from the checkpoint to ensure best performance, using

tf.keras.models.load_model(*model name*)

We then manually calculated the Mean Absolute error on test subsets for two different simulations to compare them.

$$\left\langle \frac{\partial P}{P} \right\rangle = \frac{\sum \frac{|y-\hat{y}|}{y}}{N} \tag{2.1}$$

Where P is parameter of the field (i.e. Ω_M or σ_8), y is label (ground truth value) of a sample, $\hat{y}$ is prediction for the sample and N is the total number of samples.

Labels were extracted from the datasets into separate lists using numpy methods:

y = **np.concatenate([y for x, y in *test_subset*], axis=0)**

Predictions were evaluated using TensorFlow's **.predict()** method:

$\hat{y}$ = **model.predict(*test_subset*)**

Finally, the labels and predictions are used to evaluate the percentage error on the different field datasets. Note that because the lists contain all the values, the resulting error list will also contain values for each sample, hence no need to sum over them. Instead, mean value of the list was taken.

$\langle \partial P/P \rangle$ = **np.mean(np.abs((y - $\hat{y}$) / y))**

This is the final result of an absolute error of model prediction for a certain dataset. At this point this value is used to quantify performance of the model. It is the goal of the project to bring this value to minimum and to make sure it is consistent for different simulations.

2.4 Domain Adaptation

Three adaptation models were implemented to study the effects of domain adaptation techniques on the models performance between different simulation sets. Namely, we used **MDD** (Margin Disparity Discrepancy), **DeepCORAL** (Deep CORrelation ALignment) and **RegularTransferNN**.

DeepCORAL and MDD being both feature based use the same syntax and are implemented as an additional part of a model before training, while RegularTransferNN is used as a tuner for a pre-trained model.

2.4.1 DeepCORAL & MDD

DeepCORAL learns a nonlinear transformation which aligns correlations of layer activations in deep neural networks. The method consists in training both an encoder and a task network. The encoder network maps input features into new encoded ones on which the task network is trained [13].

MDD is a feature-based domain adaptation method originally introduced for unsupervised classification Domain Adaptation. The goal of MDD is to find a new representation of the input features which minimizes the disparity discrepancy between the source and target domains. The discrepancy is estimated through adversarial training of three networks: An **encoder** a **task** network and a **discriminator**. [14]

Encoder encodes features of the data. The encoder network contains all the convolutional layer blocks from previously defined deep learning model, up to and including the flatten layer.

Task interprets the features. Contains the dense layers from the previously defined model.

The encoder network maps input features into new encoded ones on which the task network is trained.

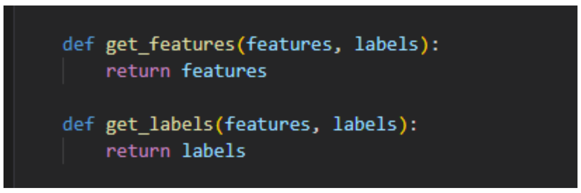
The split is done to allow for **discriminator** input. The discriminator is another classifier network. It tries to distinguish source data (initial dataset) from the target data (domain that model is adapted for) hence allowing the model to perform better on the target data.

Different optimizers for task, encoder and discriminator are defined as Adam optimizer with β_1=0.5 and β_2=0.999 for task and discriminator networks and β_1=0.5 and β_2=0.499

for the encoder network.

Learning rate reduction callback left unchanged.

Another point is that it is desired for the model to learn distinguish data, hence target data has to be separated from labels. This is done using predefined functions mapped on to the target dataset.



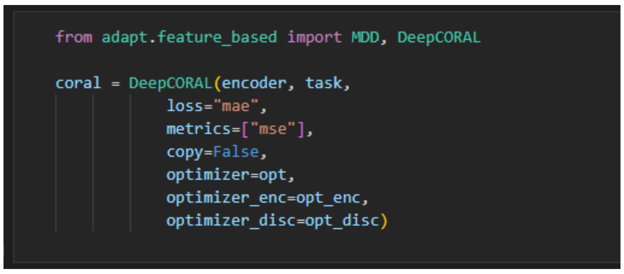
```
def get_features(features, labels):
    return features

def get_labels(features, labels):
    return labels
```

Figure 2.9: Label and image extraction functions.

Images and labels are separated into two variables and prefetched.

images = dataset.map(get_features).prefetch(tf.data.AUTOTUNE)

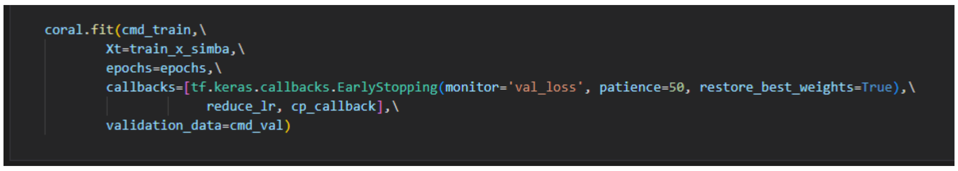
labels = dataset.map(get_labels).prefetch(tf.data.AUTOTUNE)



```
from adapt.feature_based import MDD, DeepCORAL

coral = DeepCORAL(encoder, task,
                  loss="mae",
                  metrics=["mse"],
                  copy=False,
                  optimizer=opt,
                  optimizer_enc=opt_enc,
                  optimizer_disc=opt_disc)
```

Figure 2.10: DeepCORAL model defined following the syntax from documentation [10]. As before, loss function is set to be Mean Absolute Error with Mean Squared Error as metric.

The model is then fit with the source dataset input data and target image data. Model was trained for 50 epochs, with the learning rate reduction callback, early stopping callback (unnecessary) with patience of 50 epochs, and the checkpoint callback which saves the best model after each epochs if loss improves.



```
coral.fit(cmd_train,\
        Xt=train_x_simba,\
        epochs=epochs,\
        callbacks=[tf.keras.callbacks.EarlyStopping(monitor='val_loss', patience=50, restore_best_weights=True),\
                     reduce_lr, cp_callback],\
        validation_data=cmd_val)
```

Figure 2.11: Fit the data to DeepCORAL network. **cmd_train** is the IllustrisTNG source data, **train_x_simba** is the target images from SIMBA simulation, and **cmd_val** is the IllustrisTNG validation data. T field with Ω_M parameter was used for this example.

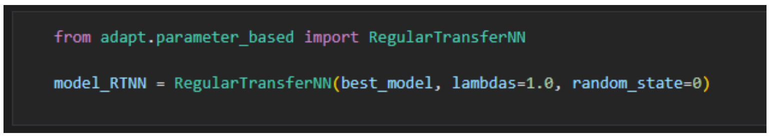
The Mean Absolute Error on each simulation dataset is then evaluated using equation (2.1) and corresponding numerical algorithm present in previous section.

Because MDD and DeepCORAL use the same syntax, to switch between the two, 'DeepCORAL' module was changed into 'MDD' (Figure 2.9) and vice versa.

2.4.2 RegularTransferNN

RegularTransferNN serves as an addition to the regular deep learning model. After setting up and training neural network as described in Section 2.2, RTNN model is defined using the pretrained neural network as an input.

The method is based on the assumption that a good target estimator can be obtained by adapting the parameters of a pretrained source estimator using a few labeled target data. The approach consist in fitting a neural network on target data according to an objective function regularized by the distance between source and target parameters. [15]



```
from adapt.parameter_based import RegularTransferNN

model_RTNN = RegularTransferNN(best_model, lambdas=1.0, random_state=0)
```

Figure 2.12: RegularTransferNN defined.

The **lambdas** argument is 1.0 by default and sets up the trade-off parameter of layers. The parameter arises from definition of regular transfer and describes relevance/importance of certain neural network parameter, i.e. how this NN parameter affects the prediction result on source and target data.

the **random_state** argument is nothing more than a seed parameter.

The target data (now with labels) is then fit to the new RTNN model. Amount of epochs is set to 50 and separate checkpoint callback is defined to preserve the previous model (the one with no Domain Adaptation involved).

The Mean Absolute Error on each simulation dataset is then evaluated using equation (2.1) and corresponding numerical algorithm present in previous sections.

Chapter 3

Results

Using the method described in previous chapter, Mean Absolute errors have been calculated for Ω_M and σ_8 parameters of M_{tot} field and Ω_M parameter of T field. T field corresponds to gas temperature, M_{tot} - total matter density.

3.1 No adaptation

3.1.1 M_{tot} field

For models trained on IllustrisTNG simulation data:

$$\left\langle \frac{\partial\Omega_M}{\Omega_M} \right\rangle_{IllistrisTNG} = 3.27\% \tag{3.1}$$

$$\left\langle \frac{\partial\Omega_M}{\Omega_M} \right\rangle_{SIMBA} = 3.72\% \tag{3.2}$$

$$\left\langle \frac{\partial\sigma_8}{\sigma_8} \right\rangle_{IllistrisTNG} = 2.23\% \tag{3.3}$$

$$\left\langle \frac{\partial \sigma_8}{\sigma_8} \right\rangle_{SIMBA} = 3.63\% \tag{3.4}$$

For models trained on SIMBA simulation data:

$$\left\langle \frac{\partial \Omega_M}{\Omega_M} \right\rangle_{IllistrisTNG} = 3.58\% \tag{3.5}$$

$$\left\langle \frac{\partial \Omega_M}{\Omega_M} \right\rangle_{SIMBA} = 3.21\% \tag{3.6}$$

$$\left\langle \frac{\partial \sigma_8}{\sigma_8} \right\rangle_{IllistrisTNG} = 2.88\% \tag{3.7}$$

$$\left\langle \frac{\partial \sigma_8}{\sigma_8} \right\rangle_{SIMBA} = 2.19\% \tag{3.8}$$

3.1.2 T field

For model trained on IllustrisTNG simulation data:

$$\left\langle \frac{\partial \Omega_M}{\Omega_M} \right\rangle_{IllistrisTNG} = 7.2\% \tag{3.9}$$

$$\left\langle \frac{\partial \Omega_M}{\Omega_M} \right\rangle_{SIMBA} = 56.1\% \tag{3.10}$$

It is worth noting that 56.1% is not the largest error produced on SIMBA test subset for T field. The fluctuations were very high and the highest amplitude reached was above 200.0%. Because of these fluctuations, two additional runs were completed for this case. 56.1% is averaged value over all runs.

3.2 With adaptation

To save time and computational resources, only T field was considered since errors on M_{tot} field parameters are too low for Domain Adaptation effects to be noticeable.

Source data (trained on): IllustrisTNG simulation

Target data (adapted for): SIMBA simulation

3.2.1 MDD

Errors on predictions for IllustrisTNG and SIMBA simulations' T field Ω_M parameter datasets using the MDD adapter are the following.

$$\left\langle \frac{\partial \Omega_M}{\Omega_M} \right\rangle_{IllistrisTNG} = 14.7\% \tag{3.11}$$

$$\left\langle \frac{\partial \Omega_M}{\Omega_M} \right\rangle_{SIMBA} = 29.8\% \tag{3.12}$$

3.2.2 DeepCORAL

Errors on predictions for IllustrisTNG and SIMBA simulations' T field Ω_M parameter datasets using the DeepCORAL adapter are the following.

$$\left\langle \frac{\partial \Omega_M}{\Omega_M} \right\rangle_{IllistrisTNG} = 34.4\% \tag{3.13}$$

$$\left\langle \frac{\partial \Omega_M}{\Omega_M} \right\rangle_{SIMBA} = 54.9\% \tag{3.14}$$

3.2.3 RegularTransferNN

Errors on predictions for IllustrisTNG and SIMBA simulations' T field Ω_M parameter datasets using the RTNN adapter are the following.

$$\left\langle \frac{\partial\Omega_M}{\Omega_M} \right\rangle_{IllistrisTNG} = 25.7\% \tag{3.15}$$

$$\left\langle \frac{\partial\Omega_M}{\Omega_M} \right\rangle_{SIMBA} = 41.4\% \tag{3.16}$$

Chapter 4

Discussion

From observing errors on predictions made by model with no domain adaptation methods implemented it becomes evident that model performs better on the test subset of the same field it was trained on. This was observed for both Ω_M and σ_8 parameters. For M_{tot} field it does not seem to be very important since the values are relatively low.

However, difference between efficiency for two different simulations becomes extreme when considering T field. At that point effects of domain adaptation methods become important. While the error on predictions for target data is still very large, it becomes slightly smaller and does not fluctuate as much as for models without domain adaptation techniques implemented. This behaviour is consistent over different adapters, so it is clear that domain adaptation does help with the problem of increased error on target data predictions. However, this comes with a price of increased error on the source data.

Overall, the domain adaptation does bring the errors on both target and source data closer together and makes the error on target data more stable. However, the errors are still too big for the model to be considered as a good way to approximate cosmological parameters for observational data.

This implies that further investigation is needed. While we have been able to determine that domain adaptation can indeed be used to improve performance of convolutional neural networks over different domains, it is still uncertain what is the best adaptation technique and how it can affect different data types. After all, CMD is just a simulated data, not real observational data. Besides, there is a possibility that it is not only a matter of good implementation of domain adaptation. Deep Learning model architecture plays major role at making the learning process as efficient as possible, and it might not be possible to create an architecture that will satisfy wide range of different simulations equally well.

Chapter 5

Conclusion

While the the results ended up not being as satisfactory as it could be desired, the research provided insight on complications encountered when using machine learning for processing scientific data. We have been able to derive that while data describing same parameters but of different origins can confuse neural networks, it is possible to adapt the neural network for more consistent performance over data from two different domains. This comes at a price of increased error on the initial, source, dataset, but allows to make consistent predictions for different domains. To apply domain adaptation to a deep learning model and further enhance its performance, further investigations into the model architecture and properties of different adapters are needed.

Chapter 6

References

[1] - CAMELS Multifield Dataset, https://camels-multifield-dataset.readthedocs.io/en/latest/

[2] - CAMELS project, https://www.camel-simulations.org

[3] - Villaescusa-Navarro, F. et al. (2021) Robust marginalization of baryonic effects for cosmological inference at the field level, arXiv.org. Available at: https://arxiv.org/abs/2109.10360 (Accessed: December 20, 2022).

[4] - How Neural Networks work. https://medium.com/@madA_reverse/neural-networks-and-machine-learning-for-the-non-technical-3b2d8b6e942a

[5] - Agarwal, B. (2020) Deep Learning Techniques for Biomedical and Health Informatics. London, United Kingdom: Academic Press.

[6] - Convolutional Neural Networks (CNNs), https://anhreynolds.com/blogs/cnn.html

[7] - Yingge, Huo & Ali, Imran & Lee, Kang-Yoon. (2020). Deep Neural Networks on Chip - A Survey. 589-592. 10.1109/BigComp48618.2020.00016 (Accessed: December 21, 2022).

[8] - TensorFlow documentation

[9] - https://albumentations.ai/

[10] - https://adapt-python.github.io/adapt/

[11] - https://github.com/adammoss/astro_datasets

[12] - Machine Learning in Science (MLiS), Part II. The University of Nottingham module notes.

[13] - https://adapt-python.github.io/adapt/generated/adapt.feature_based.DeepCORAL.html

[14] - https://adapt-python.github.io/adapt/generated/adapt.feature_based.MDD.html

[15] - https://adapt-python.github.io/adapt/generated/adapt.parameter_based.RegularTransferNN.html