X-Ray Cluster Mass

Paul, Suchetana, Unik, Valentin

June 2023

1 Goal of the experiment

In this AI lab experiment, our primary goal is to measure the masses of galaxy clusters from the provided image dataset. The provided dataset consists of simulated X-ray images, specifically generated to mimic eROSITA observations, and the redshift information of each galaxy. To achieve this, we will be making the use of Convolutional Neural Networks and also incorporating a custom loss function based on likelihood.

Throughout this experiment, our goal will be to maximize the functionality of our neural network. We will be exploring different methods to do so, such as experimenting with the number of filters and kernel size, implementing different optimizers and learning rates, introducing dropout and regularizers to the network. We will also take a look at data preprocessing methods and the effect of batch sizes on the model performance. At the end, we will run an improved model taking into account all our individual experimentations, and document the results.

2 Galaxy Clusters : Introduction and Physics background (Suchetana)

A galaxy cluster is a structure that consists of anywhere from hundreds to thousands of galaxies that are bound together by gravity. It ranges in size from 100 to 1000 galaxies and has the length scale ranging from 1 - 10 MPc (Where MPc stands for MegaParsec). For reference, each individual galaxy is in the size range of 0.03 MPc. After galaxy filaments, these are the universe's largest known gravitationally bound formations.

The composition of galaxy clusters include galaxies, but they only account for around 1% of the total mass of clusters while being the only component that can be detected optically. 9% mass contribution comes from the intergalactic gas in intracluster medium (ICM). The ICM is primarily composed of hot, ionized gas, predominantly made up of hydrogen and helium. This plasma has been heated to temperatures in the range of 10 to 100 MegaKelvins, reaching X-ray emitting temperatures, which is what will be the focus of our experiment. The x-ray radiation is emitted by thermal bremsstrahlung process. The

high temperatures are a result of the gravitational energy released during the formation of the cluster and ongoing processes such as the accretion of matter onto the cluster and the interactions between galaxies. 90% of the mass of galaxy is contained in the form of dark matter, which, while being the most massive component, is determined from gravitational interactions rather than being observable. Hydrostatic equilibrium modeling enables the determination of the ICM's mass distribution profile from measurements of the temperature and density profiles in galaxy clusters. The mass distributions discovered using these techniques show masses that are far more than the luminous mass observed, which is a strong indicator of the presence of dark matter in galaxy clusters. In order to approach our task of estimating the masses of clusters using energy spectrum of the X Ray emission, we will be using the self consistent isothermal model. We will consider certain assumptions about the ICM :

- The intracluster gas distribution is assumed to be hydrostatic, giving us :

$$\nabla P = -\rho_g \nabla \phi(r)$$

  where $p = \rho_{\mathrm{g}} k T_{\mathrm{g}} / \mu \mathrm{m_p}$ is the gas pressure, and ρ_{g} is the gas density, and $\phi(r)$ is the gravitational potential of the cluster.

- We assume that the intracluster gas is locally homogeneous.

- The cluster is modelled assuming spherical symmetry, which gives us

$$\frac{1}{\rho_g}\frac{dP}{dr} = -\frac{d\phi}{dr} = -\frac{G\ M(r)}{r^2}$$

  where r is the radius from the cluster center and $M(r)$ is the total cluster mass within r. If the contribution of the intracluster gas to the gravitational potential is ignored, then the distribution of the intracluster gas is determined by the cluster potential $\phi(r)$.

- The ICM and galaxy distributions are both static and isothermal, and the galaxy and total mass distributions are identical.

- The cluster potential is assumed to be that of a self-gravitating isothermal sphere. The cluster total density and mass are given by using the analytic King approximation to the isothermal sphere, which is often used for simplicity. It is given as follows :

$$\rho(r) = \rho_o \left(1 + x^2\right)^{-3/2}$$
$$M(r) = 4\pi\rho_o r_c^3 \left\{\ln\left[x + \left(1 + x^2\right)^{1/2}\right] - x\left(1 + x^2\right)^{-1/2}\right\}$$

  where $x \equiv \frac{r}{r_c}$, , ρ_o is the central density and r_c is the core radius of the cluster. An isothermal cluster's line-of-sight velocity dispersion is correlated with its central density and core radius by $\sigma_r^2 = \frac{4\pi G \rho_o r_c^2}{9}$. With the

help of the previous assumptions, we can write the gas distribution as :

$$\rho_g(r) = \rho_{g0}\left[1 + \left(\frac{r}{r_c}\right)^2\right]^{-3\beta/2}$$

Here,

$$\beta \equiv \frac{\mu m_p \sigma_r^2}{kT_g} = 0.76\left(\frac{\sigma_r}{10^3\ \mathrm{km/s}}\right)^2\left(\frac{T_g}{10^8\ \mathrm{K}}\right)^{-1}$$

In context of our work, we need to take a look at the total gas mass and emission integral equations, which are as follows :

$$M_g = \pi^{3/2}\rho_{go}r_c^3\frac{\Gamma[3(\beta-1)/2]}{\Gamma(3\beta/2)}$$

$$= 3.15\times10^{12}M_\odot\left(\frac{n_o}{10^{-3}\ \mathrm{cm}^{-3}}\right)\left(\frac{r_c}{0.25\mathrm{Mpc}}\right)^3\frac{\Gamma[3(\beta-1)/2]}{\Gamma(3\beta/2)}$$

(where $\beta > 1$) and,

$$EI = \pi^{3/2}\left(\frac{n_c}{n_p}\right)n_o^2r_c^3\frac{\Gamma(3\beta-3/2)}{\Gamma(3\beta)}$$

$$= 3.09\times10^{66}\ \mathrm{cm}^{-3}\left(\frac{n_0}{10^{-3}\ \mathrm{cm}^{-3}}\right)^2\left(\frac{r_c}{0.25\mathrm{Mpc}}\right)^3\frac{\Gamma(3\beta-3/2)}{\Gamma(3\beta)}$$

where ($\beta > 1/2$)

When we are measuring the X-rays from the cluster, they are always shifted by a redshift parameter Z. So, in our experiment, we will take that into account as well otherwise it may lead to inaccurate outputs. Thus we have two kinds of input, one is the dataset of the X-ray images, that are of 10 images consisting of 50x50 pixels and the other is the redshift parameter Z corresponding to each galaxy. Our trained CNN model gives the output as the mass of the observed galaxy.

3 Introduction to CNNs and Hyperparameters (Valentin Kiendl)

3.0.1 Introduction to CNN´s

A Convolutional Neural Network is a type of neural network that specializes in analyzing data that has a grid-like topology, for example, visual data (images,videos). They are designed to learn spatial features on their own. The CNN typically consist of three layer types: a convolutional layer, a pooling layer and a fully connected layer. The convolutional layer extracts features by performing convolutions of two matrices, with one being the input and one the

filter itself. The filter matrix consists of weights which are being trained. The more filters one uses the more features can be detected by the network. After this layer a pooling layer is most common, since they help to introduce translational invariance. It involves aggregating the information inside it by using either a max or average pooling function. Lastly the output of the pooling layer gets passed into a fully connected neural network. This is your standard neural network type. It will map the representation between input and output. A quick look at the architecture of a CCN can be found in Figure 1.

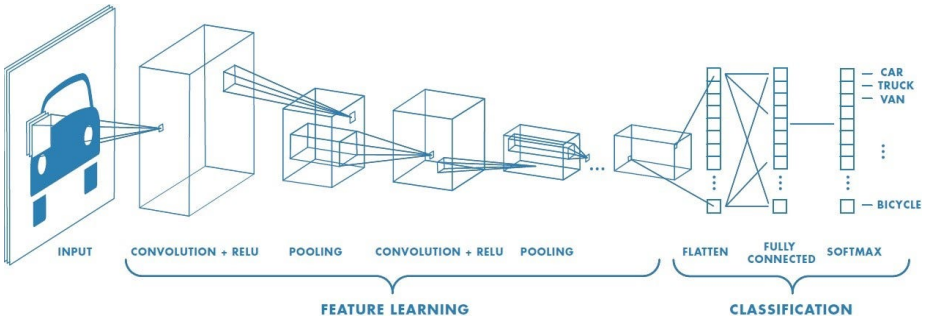


Figure 1: Typical CNN architecture [1]

3.0.2 Hyper parameters

In this section we will briefly go over all used hyper parameters in the experiment.

Fully Connected Layer:

It makes up the fully connected neural network part. Kera's dense layers preset is used for the fcnn and the hyper parameters for this type of layer are the amount of neurons per dense layer N_{neuron} as well as the number of dense layers N_{dense}. We expect that higher values for both the number of neurons and the number of layers would make a better fit the more complex a model gets.

Optimizer:

It is used to update the weights and biases of the neural network. The hyper parameter from this part is the specific optimizer algorithm used. To choose the right Optimizer is a highly important task and influences a models performance a lot.

Number of Epochs, Learning Rate, Batch-size:

The number of epochs determines how often the program will run over the

generated training set. More epochs means the neural network can learn longer to more accurately fit to a models complexity. The learning rate determines how big of an update the individual layers weights receive. The learning rate determines how fast a minimum is being reached, the lower the learning rate the slower the optimization. And lastly the batch-size defines the number of samples which contribute to a single change in the models parameters.

Loss Function:

The loss function measures the deviation of the actual outcome against the predicted outcome. The choice of the specific loss function is our last hyper parameter covered for this experiment.

Convolutional Layer:

It is the main block of the CNN. The convolutional layer slides filters or also called kernels, of chosen size over the image input to extract features. One can chose the width as well as the height of a single filter. Another factor to decide is how many filters one wants to convolve over the image and therefore train. The resulting hyper parameters from these types of layers are the size of the kernel S_{kernel} along with the amount of filters N_{filter}

Activation Function:

Another hyper parameter for the convolutional layer and the dense layer is their activation function. We chose to set it to $LeakyRELU$ and $Linear$ respectively.

Pooling layer:

Its purpose is to reduce the spatial dimension of previous convolutional layers. For this experiment we use the method of max pooling, which takes the maximum value in a given pooling window. A pooling layer comes with several hyper parameters, but we only used the size of the pooling window. The hyper parameter related with a pooling layer is the window size $S_{pooling}$.

4 Convolutional Architecture: Number of Filters and Kernel Size (Valentin Kiendl)

In this section we will go over the optimization of our model by changing the architecture of the convolutional layer part. To be specific we do a hyper parameter scan of two different types: we change the number of filter while fixing the size of the kernel window or we change the kernel window size while fix-

ing the number of filters implemented. To determine the performance of our model we plot the loss on the training data as well as the loss on the validation data. A second evaluation of the performance will give us a scatter plot of the predicted mass vs. the actual mass that comes with the labeled data. The last evaluation is by comparing the spread of the second evaluation against the hyper parameter scan. The results can be seen in the next two sections. The hyper parameters used for this task of the experiment are fixed, where the learning rate is set to 10^{-4} with the optimizer being $ADAM$. The size of a batch is set to 64 with the number of epochs being 200. As a loss we chose the MSE for the first evaluation of the data. For the fully connected layer we used an architecture of five dense layers with number of neurons as following: $512, 256, 128, 64, 1$.

For quick lookup here are the main hyper parameters listed again:

- $\eta = 10^{-4}$
- $optimizer = "ADAM"$
- $batchsize = 64$
- $epochs = 200$
- $loss = MSE$
- $denselayout = 512, 256, 128, 64, 1$

4.1 Number of Filters (Valentin Kiendl)

To find the optimal value of the number of filters that are trainable by the neural network we fixed all other hyper parameters and repeated the same model evaluation. We chose to fix the kernel size to a value of two and iterated over a list that covered one filter up to 90 in steps of 10. The number of filters stays the same for all our convolutional layers. In general we would assume that the more complex the image gets the more filters one needs to capture all of its complexity. Since our input shape are images of size 50x50x10 we settled for a high amount of filters. Figure 2 below shows the training loss and the validation loss for the model with 80 filters. Right next to it Figure 3 gives a scatter plot of the actual against the predicted mass, where the red line shows the optimal linear distribution. Considering the range of x and y axis values the spread is rather small. If we now plot the spread Figure 3 against the number of filters used we can spot the general trend in Figure 4, in which we can see that the more filters you use the more features can be learned and the better the fit will be, up to a certain point where the model would start overfitting and hence the performance would be going down again.

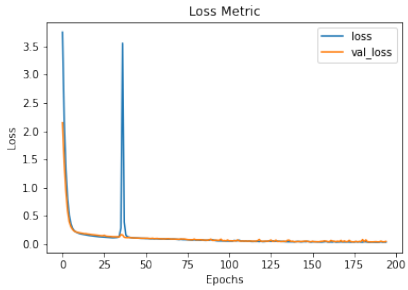


Figure 2: Train and Validation Loss

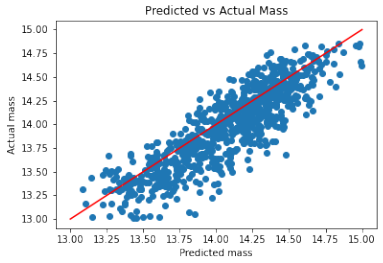


Figure 3: Actual vs Predicted Mass

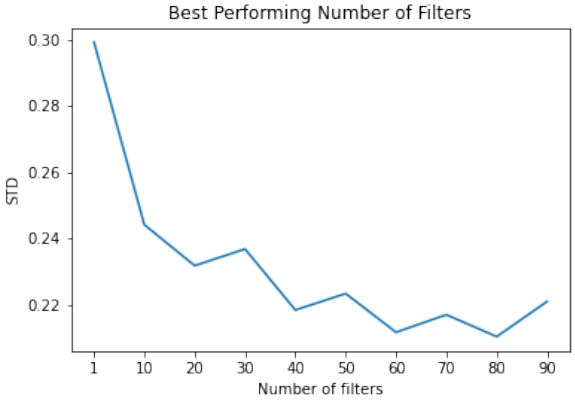


Figure 4: Standard deviation of all tested Filter Numbers

4.2 Kernel Size (Valentin Kiendl)

To find the optimal kernel size we repeated the same evaluation as for the number of filters, this time fixing them on a value of ten. The kernel sizes that are being tested range from one to 18 in steps of two. One has to keep in mind that the input image that is being convolved only has a spatial dimension of 50, meaning that higher kernel sizes are coarse-graining the image very quickly. We assumed that for this image a kernel size of around 10 % of its input size would give the best results. This performance trend came out to be true, where a kernel size value of four shows a minimum for the standard deviation. This trend could be explained by the fact that a larger kernel size gives us more trainable parameters which would then improve a fit to a more complex underlying model until, as mentioned above and how we expected it to be, the image gets coarse grained too much and we lose too much information. Figure 5 below shows the train and validation loss for the best performing model. The scatter plot in Figure 6 shows a rather nice linear dependence, with little spread, of the actual against the predicted mass. This spread is again plotted against the hyper parameter scan of the kernel size in Figure 7, where we can see that, as written above, we expect an improvement in performance up to a kernel size of around four and afterwards the model starts overfitting.

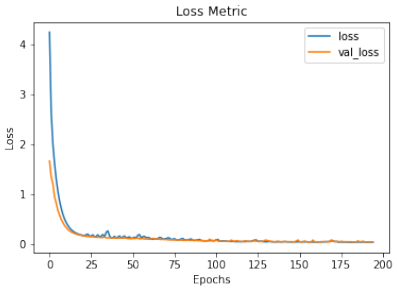


Figure 5: Train and Validation Loss

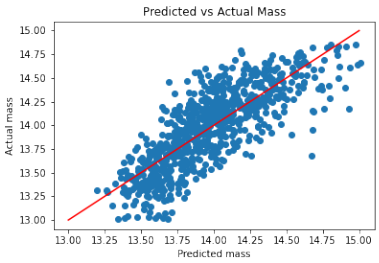


Figure 6: Actual vs Predicted Mass

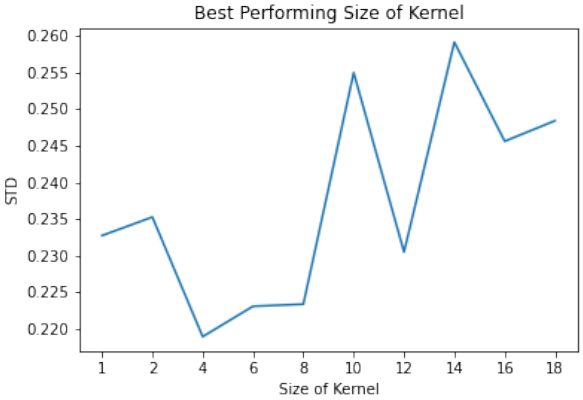


Figure 7: Standard deviation of all tested kernel sizes

5 LogLoss

The log loss is a commonly used loss function for estimating the error associated with the predictions of a neural network. It is particularly suitable when assuming a Gaussian distribution for the conditional probability of the target variable given the input and network parameters. It also has a minima at some finite value for y(if we assume it to be a continous single variable function) which helps us to avoid underfitting and overfitting of the curve. The log loss is defined as:

$$\log \text{loss} = -\log\left(\frac{1}{\sqrt{2\pi\sigma(x,\theta)^2}}\exp\left(-\frac{(y-f(x,\theta))^2}{2\sigma(x,\theta)^2}\right)\right)$$

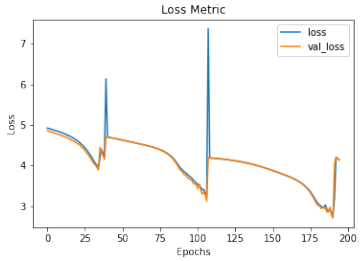


Figure 8: Log loss and accuracy

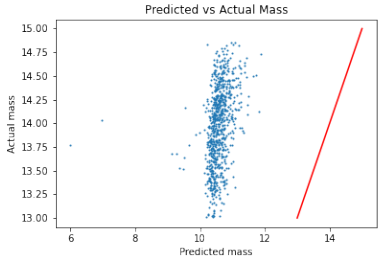


Figure 9: Log loss scatter plot

Training a model with log loss can be challenging. Therefore, we used state initializers for both the standard deviation (sigma) and mass values to facilitate early convergence. The attached plots illustrate the progress and convergence of the training process.

6 Optimizer and learning rates (Paul Kopold)

6.1 Architecture setup

For this section the architecture used for all models is the following:

- 3 convolutional layers with kernel size 2 and number of filters 20
- a maximum pooling layer with size 4
- 2 convolutional layers with kernel size 2 and number of filters 20
- a maximum pooling layer with size 4
- five dense layers with sizes 512,256,128,64,1

Every layer was followed by a LeakyReLU activation function except for the output layer. The batch size during training was 512. With this architecture fixed the effect of the choice of the optimizer was examined. The loss function used was always the mean squared error loss function.

6.2 Effect of the learning rate and the optimizer

We expect the loss to continually decrease up to a point where it stays in a minimum, while the loss for the test data might go down slower and maybe have a higher minimum or even increase after a high number of epochs. The learning rate determines how fast a local minimum is being reached, the lower the learning rate the slower the optimization. However, if the learning rate is high, the steps might jump over a local minimum, in that case we expect more noise in the loss function. With a lower loss function we expect the loss function to get stuck in a higher local minimum.
Two different optimizers were tested, Adam and Stochastic gradient descent. Since stochastic gradient descent just updates weights based on the values of the current batch without taking into account first and second moments as does Adam, we expect SGD to get more easily stuck in local minima and have less fluctuations in the loss, which we expect to lead to the loss converging to a higher value. The models were trained for 1000 epochs, the training history for a model with Adam optimizer trained with learning rate 10^{-3} is shown in Figure 10.

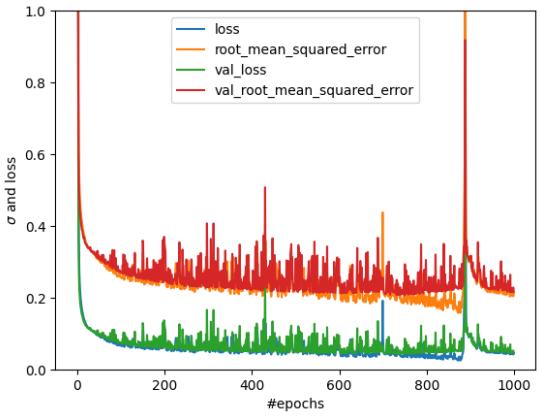
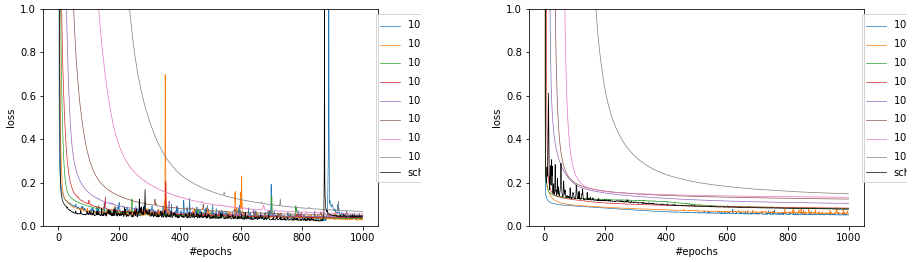


Figure 10: Training history with Adam Optimizer, learning rate 10^{-3}

During training the standard deviation of the models prediction minus the labels was computed for the training dataset and the evaluation dataset, it is denoted by root_mean_squared_error in the legend of the figure. The training loss follows the evaluation loss for the epochs we used, we will just look at the training loss in the following discussion.

6.3 Hyperparameter search

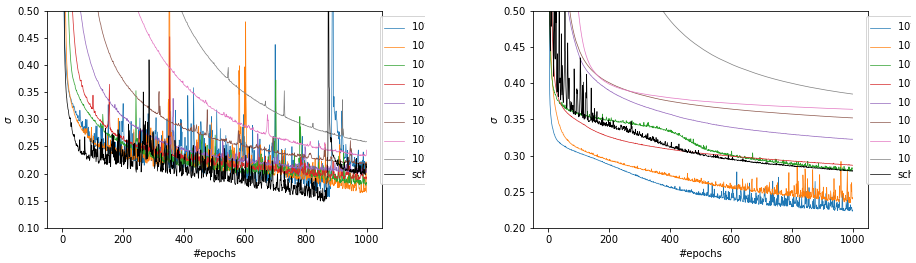
The model was trained with Stochastic gradient descent and Adam for 10 different learning rates with values between 10^{-5} and $10^{-2.66}$. Additionally there is a model which was trained with a scheduled learning rate with an exponentially decaying learning rate with values between $10^{-2.66}$ and 10^{-3}. The loss during training can be seen in Figure 11a for the Adam optimizer and in Figure 11b for Stochastic gradient descent. In both figures the loss is only shown in the range of values between zero and one. For the Adam optimizer we find that there is a lot of noise while training for high learning rates, while for low learning rates, for lower learning rates the loss takes longer to converge to a low value. The lowest loss during the hyperparameter search for fixed learning rates was reached for the learning rate $10^{-2.66}$ in epoch 877. This can be more clearly seen in Figure 12, where we compare the standard deviations of the predictions minus the labels for all trained models during training. After this low value in the loss and the standard deviation however the loss has a spike and then continues to decrease again. After 1000 epochs of training the best model is the one with learning rate 10^{-3}, even though the loss and standard deviation are higher than with learning rate $10^{-2.66}$ in epoch 877. In the later stages of training, the stability of the training process becomes more and more important, since getting a spike in the last epoch of the training would be undesirable. One way to deal with this issue is to stop the training once the loss reaches a certain value another is one we also explored with a scheduled learning rate. The model improves quickly while the learning rate is still high and then when the learning rate becomes smaller the loss improves more smoothly.
For the SGD optimizer the training is less noisy for all the learning rates tested, but the model doesn't find values low in the loss as quickly as with Adam. The model with the decaying learning rate is here clearly outperformed by the ones with fixed learning rates $10^{-2.66}$ and 10^{-3}. As with the Adam optimizer, for lower learning rates convergence is slower and the model did not converge after 1000 epochs yet.



(a) Adam optimizer

(b) SGD optimizer

Figure 11: training loss histories



(a) Adam optimizer

(b) SGD optimizer

Figure 12: training standard deviation histories

7 Introducing Dropout (Suchetana)

Dropout is a training technique in which a predetermined percentage of neurons are randomly selected to be eliminated. As a result, the model is only able to depend on a limited number of neurons, which aids in regularizing it. In our experiment, we have decided to experiment with placement of dropout layers, as well as altering the dropout rates to see how it impacts our model performnace. We have tried training the model by placing one singular dropout layer between dense layers, but the best performance was shown when we placed one dropout layer after every dense layer. The standard deviation of our experiment was decreased by a good amount after implementing dropout. We have also observed the fact that this allowed us to train the model for longer times for higher numbers of epochs, without encountering overfitting.

For the model with only dense layers with no dropout integration, we achieved a standard deviation of 0.2278, and after implementing dropout of 0.05, we immediately were able to lower it to 0.205. After this, we experimented with what dropout rate performs best for our particular dataset. For a dropout rate of 0.2, the standard deviation was observed to be 0.2035, which was one of the best results we obtained. For 0.1, it was 0.204 while for a dropout of 0.3 the deviation from actual mass values was 0.206. But when increasing the dropout to 0.4, the standard deviation went up to a value of 0.2139, which was a surprising result. Even after rerunning it, the performance did not improve much. Upon increasing it to 0.45, we immediately got a better performance of a deviation of 0.204. For dropout of 0.5, we got a value of 0.205, which is around the same threshold. We have noticed a decrease in validation loss after implementing dropout throughout all the models. In order to be consistent, we have attempted to run all the models for a similar number of epochs (for around 400-450) but adjusted accordingly whenever it seemed that the model was performing its best around a certain number of epochs.

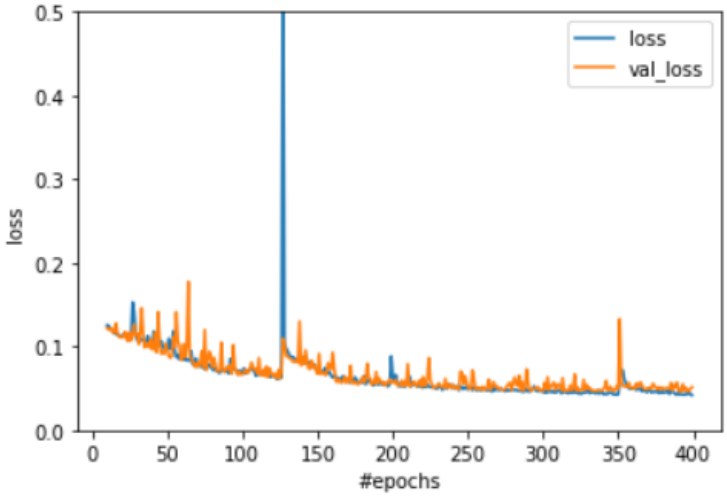


Figure 13: Model performances with training of model with no dropout layer

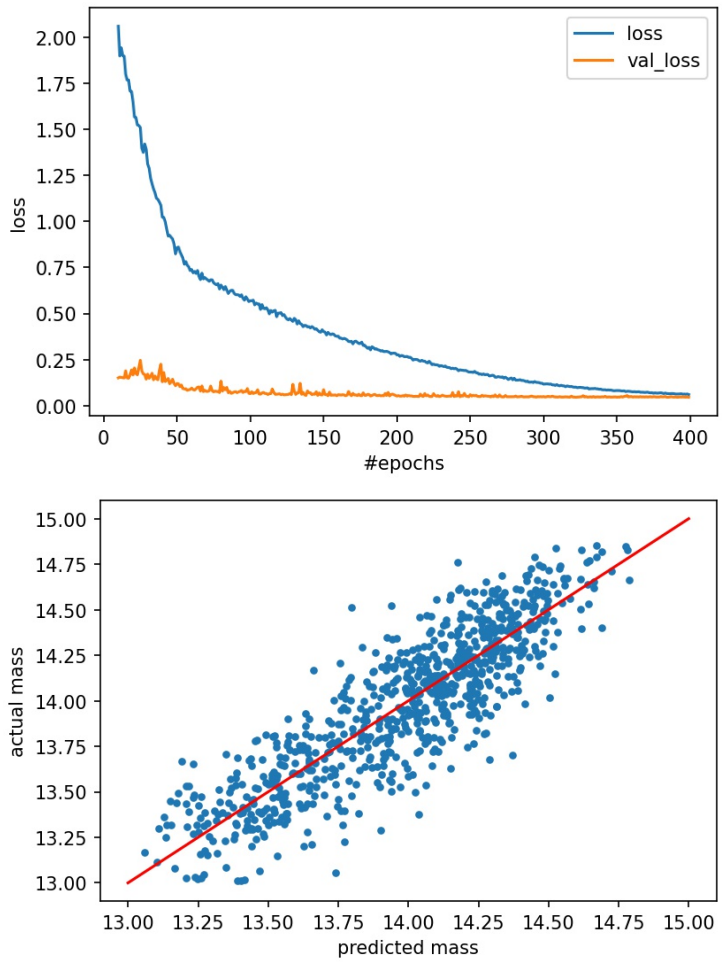


Figure 14: Training curve for model with 0.2 dropout rate which performed the best for us. The scatter plot between the actual and predicted mass is of dropout rate 0.2.

As a conclusion we can comment that while dropout greatly improved model performance upon implementation, the increasing of dropout rate did not correlate very strongly to an increase in model performance. The performances for different dropouts were all in quite the same threshold with a few surprising dips, which was very interesting to observe.

8 Effect of Regularizers (Suchetana)

When training a model in Keras, regularizers are used to impose penalties to the model's parameters. By encouraging the weights to remain tiny or sparse,

they help in preventing overfitting and enhance the model's capacity for generalization. The types of regularizers available on Keras are the L1 (Lasso) and L2 (Ridge) regularizers, and the L1L2 (Elastic Net) regularizer. Regularization penalties are applied on a per-layer basis, and we have to option to apply the penalty on either the kernel, bias or output of the layer.

To begin with, we considered a model with no regularizer integration in its dense layer architecture. We get a standard deviation of 0.2102 after a training of 400 epochs. Upon implementing kernel regularizer L2 (for the same number of epochs and no other difference in model architecture), we immediately notice an improvement in performance with the standard deviation being 0.2073. Upon switching to L1 regularizer, the deviation is observed to be 0.2141, and after running it through L1L2 kernel regularizer, the deviation comes to be 0.2246 (we took L1 and L2 to be 0.0001, and in all the other cases we also took the value as 0.0001, as we found out this was where the model performance was best). Now, we decided to try switching the penalty from being applied to the kernel and now to the bias. We move ahead with L2 class, and receive a standard deviation of 0.2202. After applying the "activity_regularizer" that applies the penalty to the output, we also receive standard deviation around the same threshold as before. In conclusion, our model performance increased the most when we applied the L2(ridge) kernel regularizer to our dataset.

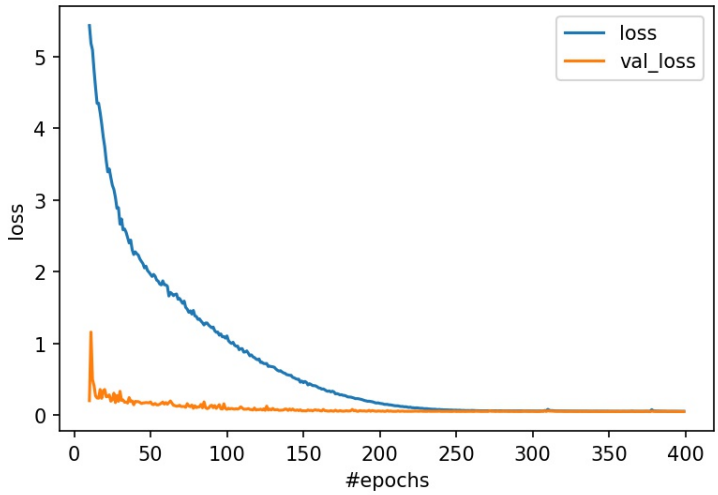


Figure 15: Training process of model with no regularizer integration

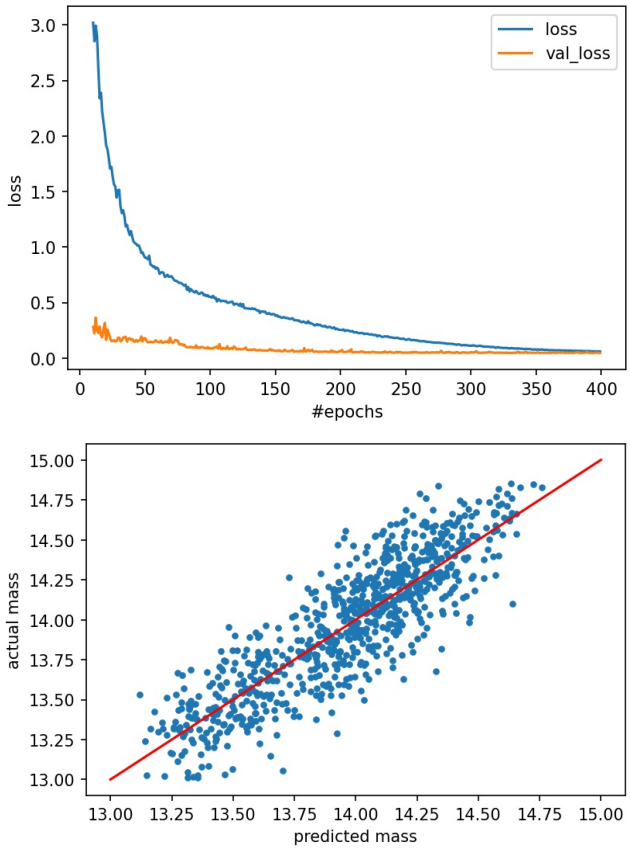


Figure 16: Performance of model with L2 kernel regularization, followed by the the scatter plot for our best performing regularizer.

9 Experimenting with Batch Sizes (Unik)

Batch size refers to the number of training examples utilized in one iteration. The initial model we started with had a batch size of 64. To experiment with different batch sizes, we compiled models with various batch sizes multiple times and compared their performance. All these different experimentations are saved in the folder `saved_model_U/Batch_Size`. Analyzing the effects of different batch sizes is crucial for fine-tuning our model and ensuring its efficiency during training.

To begin, we needed to ensure that the batch size isn't too low as it may result in a noisy training process. We iterated over batch sizes ranging from 20 to 200 and trained the model while saving the results.

We arrived at the following conclusions:

1. The batch size, in general, has a negligible effect on reaching convergence,

provided we run the model for a sufficient number of epochs (as shown in Figure 13).

2. However, having a low batch size can lead to problems such as the introduction of random noise during training and increased code compilation time per epoch (as there are more batches for the same dataset size).

3. Having a very high batch size requires a large amount of RAM, which can pose resource constraints.

Therefore, for the final model, we decided to use a batch size of around 200. Considering our limited RAM resources and dataset size, this choice proved to be the most suitable.

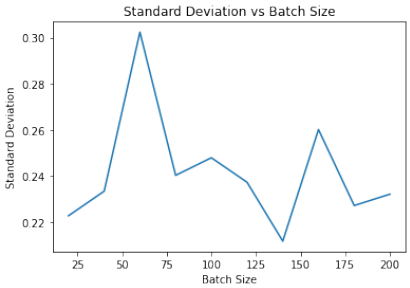


Figure 17: Standard deviation of test cases vs batch size

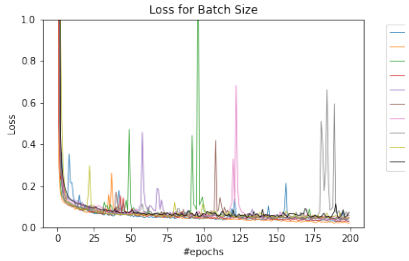


Figure 18: Loss curves vs number of epochs

Figure 13 illustrates the relationship between the standard deviation of test cases and batch size. We observe no correlation between the two. For every compilation, this graph is randomized, with any batch size corresponding to the lowest standard deviation. This makes sense as we start with randomized initial conditions.

Figure 14 displays the loss curves versus the number of epochs. We can observe that with a sufficient number of epochs, the Mean Squared Error (MSE) loss saturates to nearly the same point.

10 Data Preprocessing(Unik

After splitting the data into a training and test set with a 90-10 ratio, we obtained a dataset of size (7151, 50, 50, 10). This can be interpreted as 7151 images, each with dimensions of 50x50 pixels and 10 X-ray channels. Our task now is to utilize TensorFlow's preprocessing capabilities to generate additional samples from the images. We primarily focus on two processes:

1. `tf.keras.layers.RandomFlip`: This function randomly flips the images vertically, horizontally, or both.

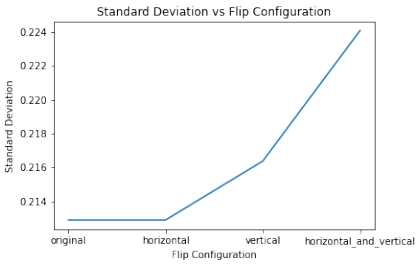


Figure 19: Flip state vs standard deviation

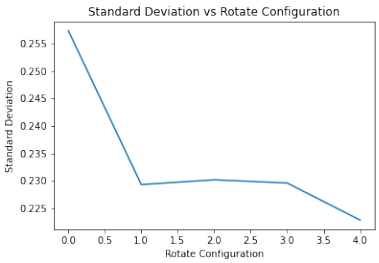


Figure 20: Random rotation state vs standard deviation

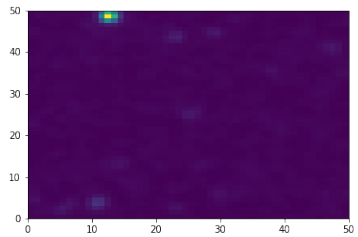


Figure 21: Original image

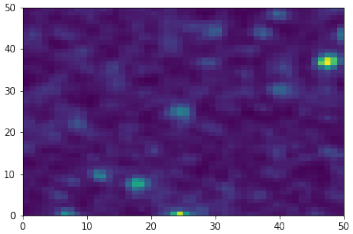


Figure 22: Rotated image showing noise

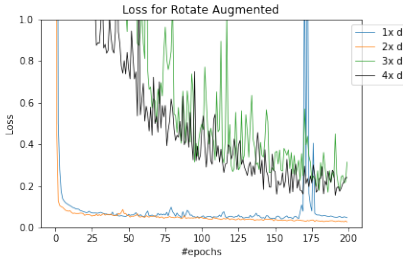


Figure 23: Loss curve for augmented model

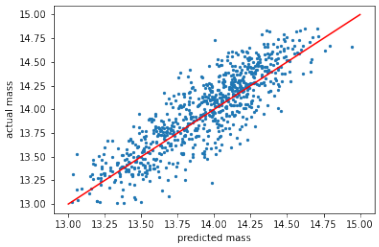


Figure 24: Fit for the best model

2. `tf.keras.layers.RandomRotation`: This function randomly rotates the dataset from 0 to 2π.

We generate four types of models to compare their performance:

1. Model consisting of datasets with random rotations.
2. Model consisting of datasets with random flips.
3. Augmentation of models with random rotations.
4. Augmentation of models with the best parameters.

The random rotations are independent of each other and are plotted for comparison against the original dataset (plotted at x=0). The same applies to random flips.

In the case of the first augmented model, we observe that as the size of the dataset increases, the performance of the model decreases. There could be two reasons for this:

1. The generation of random rotations using TensorFlow's preprocessing involves interpolation techniques, which may introduce noise in the data due to the relatively small image size of 50x50 pixels (as shown in Figure 15).
2. Limited resources and a low number of trained epochs may prevent the model from reaching true convergence (as shown in Figure 16).

Lastly, we train the best model with only flips or 90/180/270-degree rotations. This model is considered the best since it avoids any random noise that may have been generated during the rotation of the matrix to non-square angles.

11 Sources

$https://d33wubrfki0l68.cloudfront.net/a7664cf19de33b2c71a482629f27a0d70f715b77/6949d/images/blog/comprehensive-guide-to-convolutional-neural-networks-the-eli5-way.jpg$ [1]

$https://ned.ipac.caltech.edu/level5/March02/Sarazin/frames.tml$[2]