DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Pawpularity Contest: Predict the popularity of shelter pet photos

CS 354N: Computational Intelligence Lab

Name	Roll Number
Jha Rohan	210002041
Nishkarsh Luthra	210001045
Hrishesh Sharma	210003037

Course Instructor: Dr. Aruna Tiwari

Code of this project can be found at
https://github.com/rohanjha04/Pawpularity-CI.git

April 20, 2024

1 Introduction

The problem of finding the popularity of images on social media and other media based on the image and metadata, like accessories, eyes, subject, near, etc. is popular in Computer Vision and Machine Learning. This problem is extended to the images of Cats and Dogs by Petfinder.my, Malaysia's leading animal welfare platform. The challenge is to **predict the "pawpularity" score of images of cats and dogs** up for adoption based on their images and some metadata. Such a tool is very useful as it can help post the best images of the animals online, optimistically resulting in a higher adoption rate and fewer stray animals on the roads.

2 Analysis and Design

2.1 Data Collection and Preprocessing

The problem stated in Section 1 was hosted as a Kaggle challenge [1] around two years ago. We have used the dataset provided by the competition organisers. The dataset consists of **9912 images** and the corresponding metadata. The **metadata has 12 fields in it, namely: 'Subject Focus', 'Eyes', 'Face', 'Near', 'Action', 'Accessory', 'Group', 'Collage', 'Human', 'Occlusion', 'Info' and 'Blur'**.
After analyzing the data, we see that the dataset is highly **rightly skewed**. More than 7000 values have a score of less than 50, while only 1807 samples have a score of more than 50 in the training dataset. Even for less than 50, maximum samples lie in the range of 20-40.

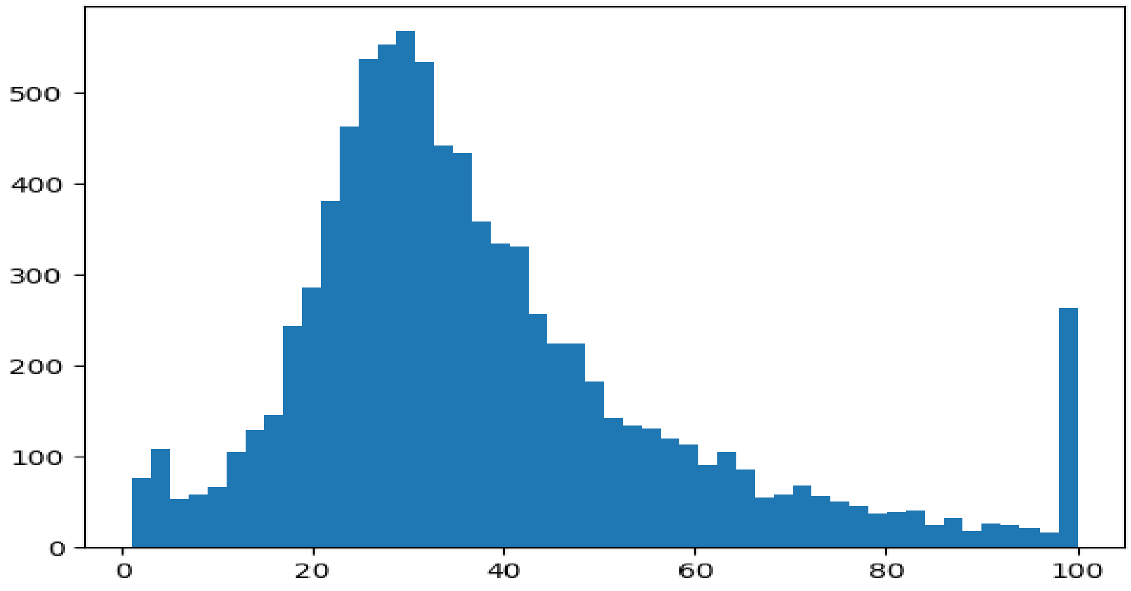

Figure 1: Input Data Skewness.

We have split the 9912 images into three sets:

1. **Train Dataset:** The first **8912** images and corresponding metadata are the training set.
2. **Validation Dataset:** Next **500** images and metadata are used for the validation dataset.
3. **Test Dataset:** The last **500** images and corresponding metadata are used for testing.

Since we trained all our models on Kaggle, we had restrictions on how much RAM and GPU power we could use. So, we resized all our images to `(256, 256, 3)` and employed a Custom DataLoader to implement additional functions to perform the data augmentation and loading required. We fixed a batch size of 32 and trained and tested all our models batch-wise to avoid OOM (out of memory) errors on Kaggle. Tensorflow allows us to overload the inbuild DataLoader. The function `data generation` performs the image loading and preprocessing tasks. We have cast the image to `tf.float16` to reduce the precision of the images to fit it in the available memory on Kaggle. We also perform random flips and normalise the image. We have normalised the labels to avoid bias in 0-1 instead of 0-100.

2.2 Algorithm

To tackle the problem, we have employed the following architecture.

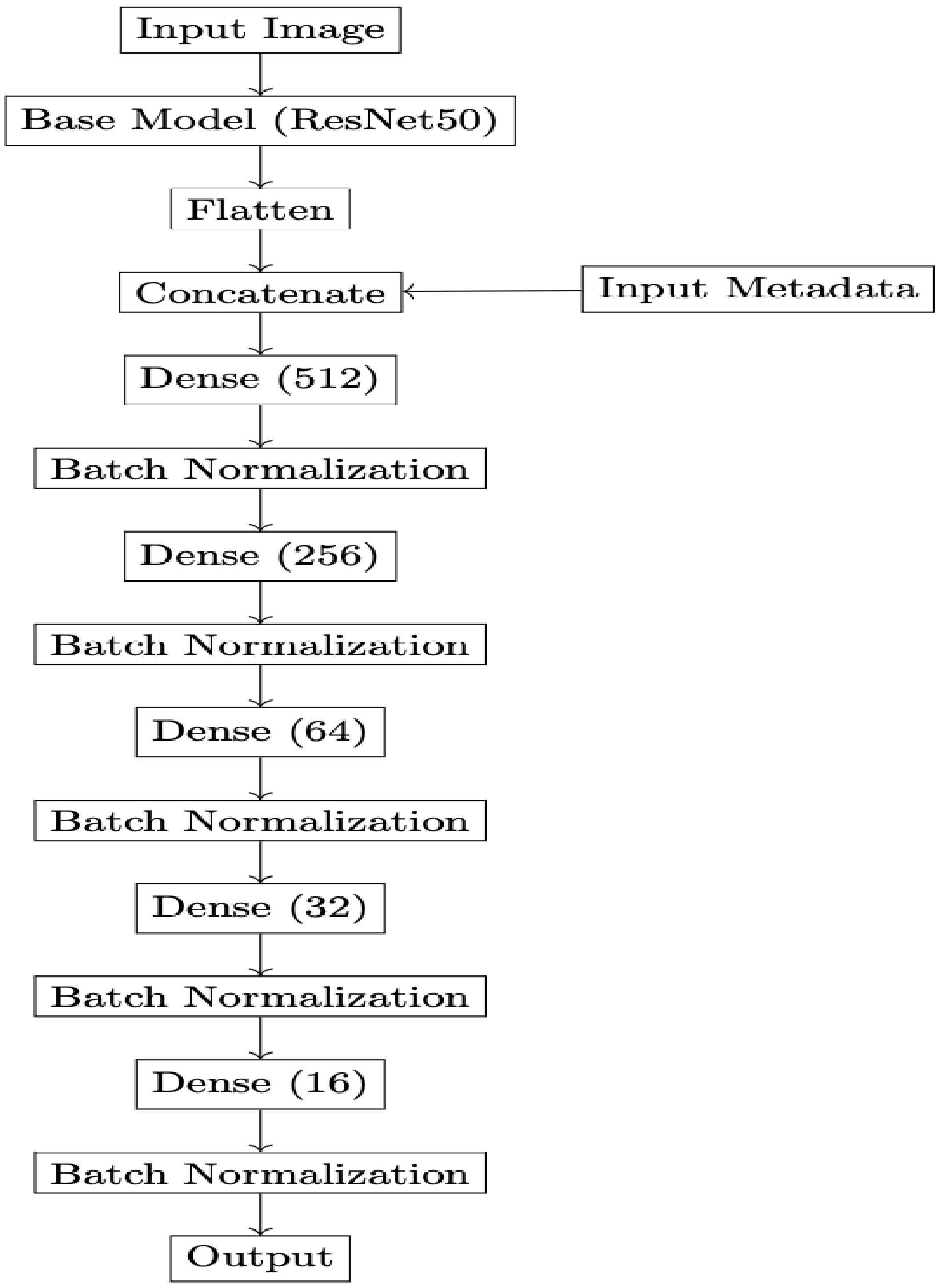

Figure 2: Block Diagram.

Firstly, we **load the image using the DataLoader** and pass it on to **ResNet50 [3] for feature extraction**. The name "ResNet50" comes from the fact that it consists of 50 layers, including convolutional layers, pooling layers, fully connected layers, and shortcut connections. We have initialised the weights from the imagenet dataset, which contains classes for Dogs and Cats. This information in the network was helpful, providing us with acceptable results. We then f**lattened the output features and concatenated them with the metadata** to form the final train features. The output of the concatenate layer gives us 1 lakh features to train the model. We then pass it through a few fully connected dense layers. Finally, we have used a **one-node output with `sigmoid` activation function** to predict values between 0-1 as required since we know that sigmoid has a range of 0-1. We have also used a **dynamic learning rate with exponential decay** [4] to prevent overfitting. Initially, the learning rate is set to a higher value of 0.008. This is to provide initial momentum and prevent trapping in local minima. Further, to prevent overfitting, we have used a decay rate of 0.94 after every 10000 step to get the exponential decay. We also tried some other approaches with the architecture, which is discussed in 3

2.3 Performance Measures Used

To train the model, we used **Mean Squared Error** since we were training the model in a Regression fashion. Also, since the data we have used is highly skewed, we have given a **weight of 2.0 for all the errors in labels greater than 50 and 2.5 for all errors in labels less than 20.** This helped us to generalise the model since fewer values are present in these ranges. So, whenever an error is encountered, we give more weight to the error for those labels.
To evaluate the model, we have used the **Root Mean Squared Error**, the same metric used by the competition organisers to evaluate during the competition.

3 Experiment and Results

To tackle the above problem, we used the following experiments:

1. **Try to train Resnet50 itself:** We first developed an architecture as given in Figure 2 and set **all the layers as trainable** in the model. So, we tried to train a deep network consisting of 50 layers of ResNet and 10 deep layers after the concatenation to get the results. However, due to GPU and RAM limitations on Kaggle, we could train the model for only 10 epochs. Though we got a train RMSE score of 0.22 and a testing RMSE score of 0.21563, the generalised accuracy of the model was poor since it was predicting values **only in the range of 0.43-0.45**. This resulted in an overall lesser RMSE score but a useless model, which just guessed a value from the majority class. The main culprit of this problem was the skewed input data, and this experiment forced us to explore some techniques to handle right-skewed data.

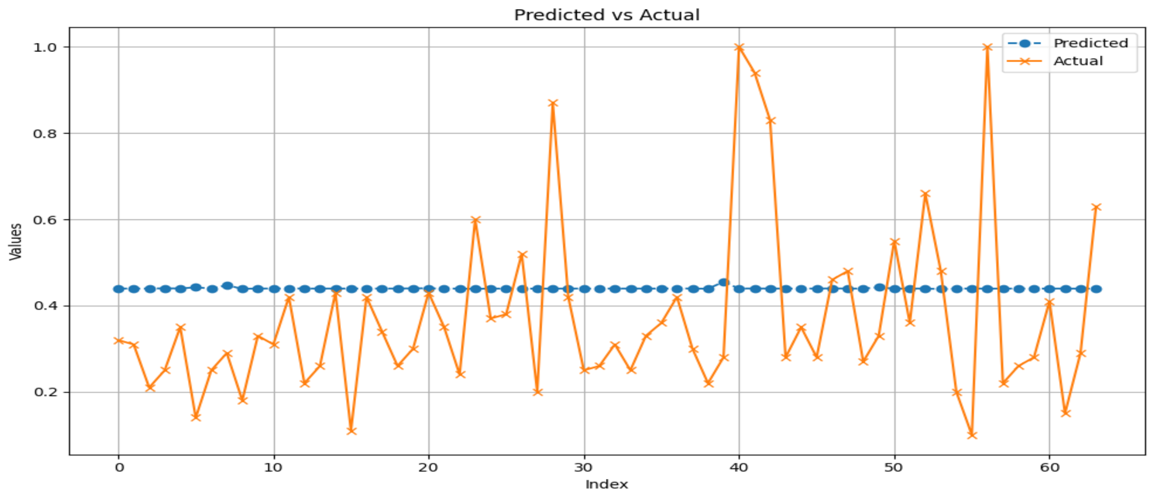

Figure 3: Results for expt 1.

2. **Scaling Data Using *log1p*:** Since it was clear that input data was skewed, we tried to fix it by **taking log1p of the input data labels**. Log1p is a function in Numpy which calculates the natural log of (1+x) for all x belonging to the input array. So, a label of 0 was converted to $log(1+0) = 0$ and a label of 1 was converted to $log(1+1) = log(2) = 0.6931$. Thus, after the transformation, the input data looked as below. We trained this transformed data with architecture in figure 2.

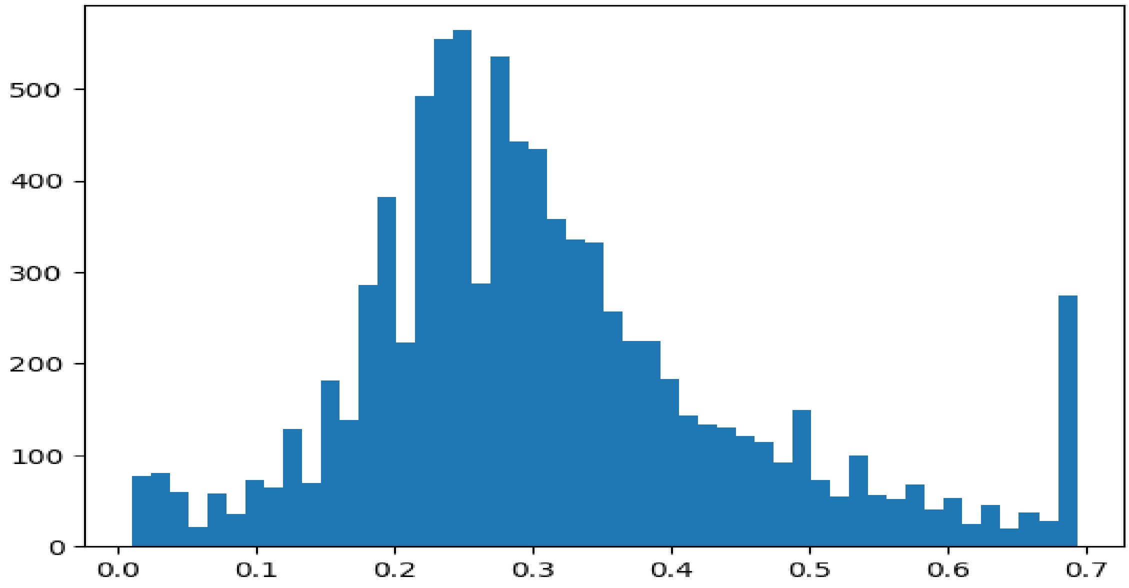

Figure 4: Input labels after log1p norm.

However, there was only a minor improvement in the overall RMSE, and the results were similar to Figure 3. This is mainly because though taking *log1p* norm helped to make the data more like a normal distribution, the number of samples in the range of $0.2 - 0.4$ remained significantly higher than other classes. So, we had to find other ways to deal with the unbalance in the data.

3. **Try to use Machine Learning Methods:** Another approach we parallelly tried was machine learning approaches like **XG Regressor, SVRegressor**, etc. However, the main problem was that these models **did not support partial fit**, and our data was massive. So, all data could not be loaded into memory to perform "fit" at once. We tried to implement some code to enable us to **continue learning** about the data; however, we could not get good results. The XGRegressor's output was always either 0 or 1 despite much fine-tuning.

4. **Use weighted back propagation:** Confused how to tackle the problem of unbalanced data, we came across [2]. This paper discusses using a **loss function asymmetric about the origin** for imbalanced classes, defined as $\tilde{L}(x) = e^{ax} - ax - 1$. So, we use a **linear loss function for the majority class, and for the minority class, we use an exponential loss function**. Choose the value of a so that higher loss values are given to the minority class. Implementing this for our architecture 2, we decided to use **exponentially weighted loss** during backpropagation. During the backpropagation, we set sample weights as $e^{(5*y)}$, where y is the value of the label, scaled between 0-1 instead of 0-100. Applying the idea of exponential loss provided us with some promising results as:

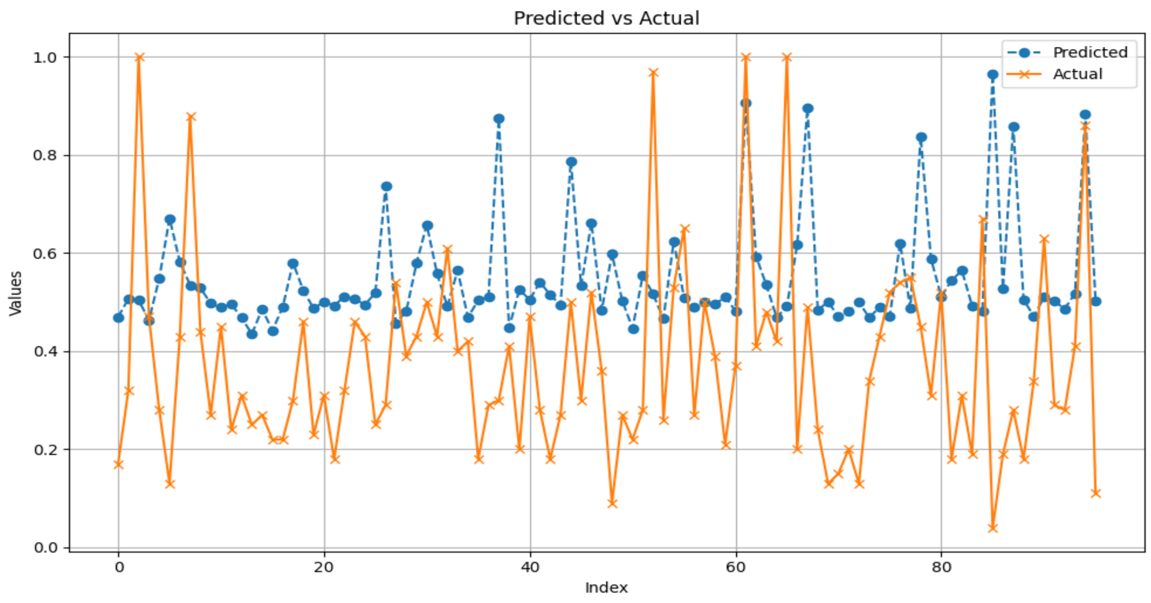

Figure 5: Output for exponential weighted loss

The values diverged from the range of $0.2 - 0.4$. However, $e^{(5*y)}$ scaled the values a little too much, and the model became **biased for higher values**. Thus, in Figure 5, we can see that the model is overall shifted upwards, though it is following the pattern. The RMSE score that we obtained was 0.357989 and 0.352862 for the testing and training dataset due to this particular reason.

So, we **decided to use** $e^{(3*y)}$**,** reducing the weights a little. In doing so, we got the following results:

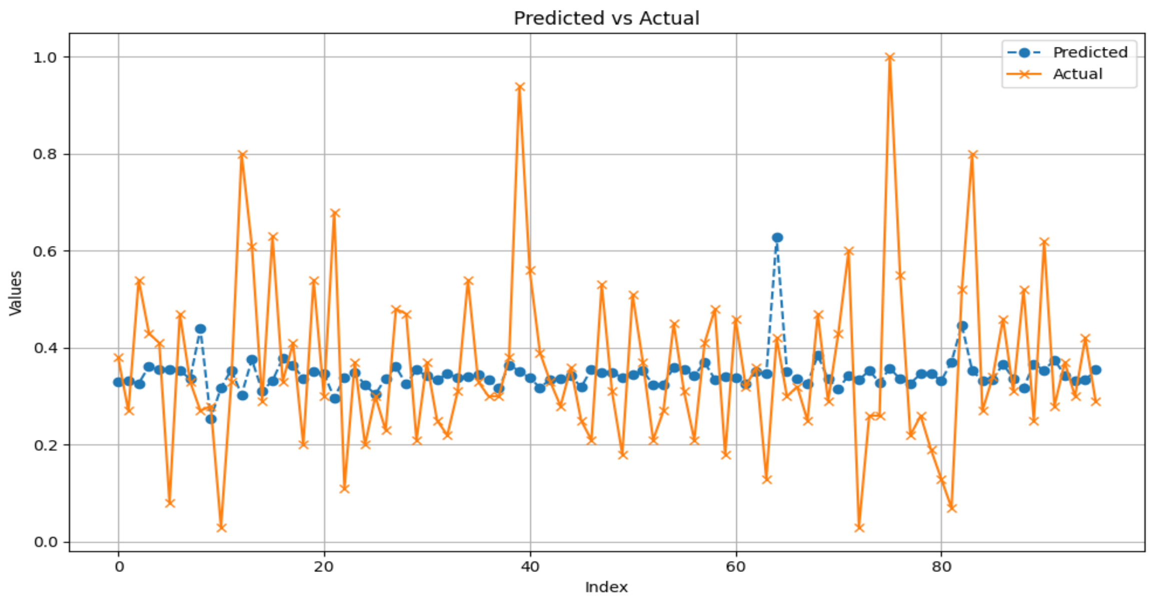

Figure 6: Output for reduced exponential weighted loss

However, though the RMSE score was reduced to 0.22257 and 0.22258 for the testing and training dataset, the model **struggled to predict higher values correctly**.

To combat these problems with exponential loss, we decided to use a discontinuous weighing mechanism where we gave **sample weights of 2.5 for all the labels greater than 0.5;** otherwise, we gave a weight of 1. In doing so, we found that the model was again becoming too biased for higher values. We tried to reduce the weight to 2 instead of 2.5, but the problem remained. The testing RMSE scores in the two cases were 0.213438 and 0.2031116 The outputs for the two are as follows:

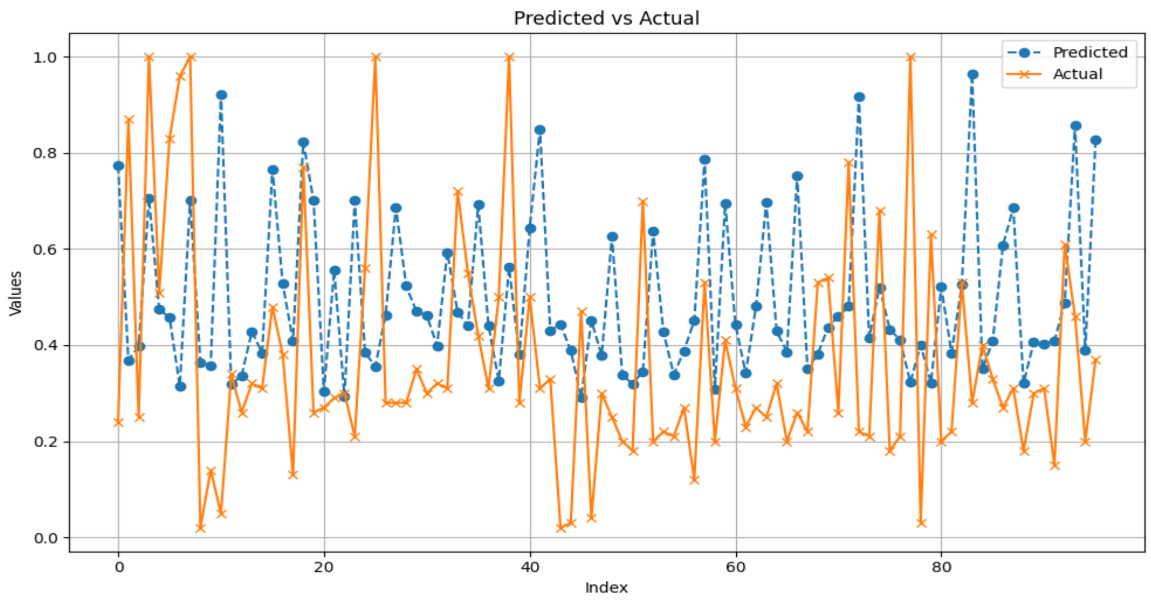

Figure 7: Output for 2.5 for label>0.5 weighted loss

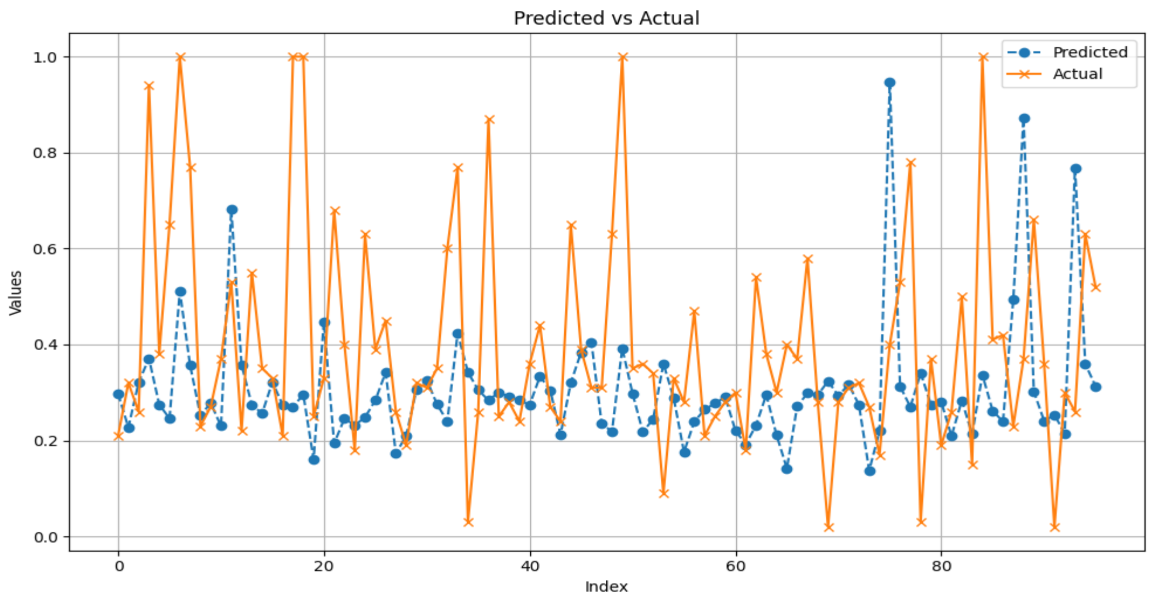

Figure 8: Output for 2.0 for label>0.5 weighted loss

Though the sample weights helped the model learn for values of labels greater than 0.5, the model also struggled with values less than 0.2 to predict accurately. So, we defined our **sample weights as 2.5 for labels less than 0.2, 2.0 for values greater than 0.5 and 1 otherwise.** This gives us the **best RMSE score** of 0.19746 and 0.19759 on the training and testing dataset. The plot is as below. We can see that the general fit is better than the previous models.

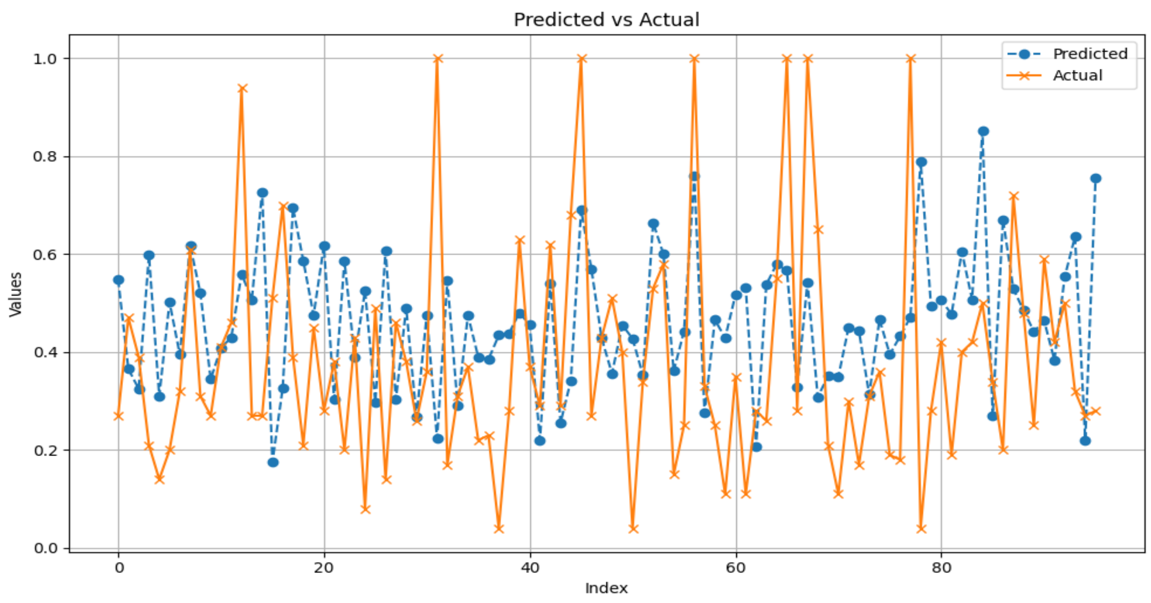

Figure 9: Output for new sample weights

Since this gave us the best generalisation result, we chose this model as our final. All the results are tabularized below for easy comparison:

Table 1: RMSE Scores

Method	Train RMSE	Test RMSE
Train ResNet directly	0.22	0.21563
Scaling using log1p	0.223	0.22078
Machine Learning Methods	>0.5	>0.5
Exponential weighted loss $e^{(5*y)}$	0.352862	0.357989
Exponential weighted loss $e^{(3*y)}$	0.2225	0.22257
Sample weighted loss 2.5, $y > 0.5$	0.2132747	0.213438
Sample weighted loss 2.0, $y > 0.5$	0.20298	0.2031116
Sample weighted loss 2.5, $y < 0.2$; 2.0, $y > 0.5$	**0.19746**	**0.19759**

Thus, we performed extensive experiments and read to develop the best-generalised model that would suit our needs.

Further, to provide an **end-to-end solution** for our problem, we have developed a **simple interface using Streamlit Python**. Any test picture can be uploaded to the interface. Click Process Image to get the score. We can also select the metadata features using a checkbox. The app loads the model weights for the best model, i.e. Sample weighted loss 2.5, $y < 0.2$; 2.0, $y > 0.5$ and predicts the score.

4 Conclusion

1. **Diverse Experimentation:** We performed various experiments to address the challenge of predicting the "pawpularity" score of images of cats and dogs. From training ResNet50 directly to employing machine learning methods and weighted backpropagation, the team explored multiple avenues to optimize the model and generalised performance.

2. **Data Analysis and Preprocessing:** An in-depth dataset analysis revealed its highly skewed nature, with most samples scoring below 50. This insight led to exploring techniques such as data scaling using log1p to normalize the distribution of labels and enhance model generalization.

3. **Architecture Design:** We employed a sophisticated architecture involving feature extraction using ResNet50, concatenation with metadata, and dense layers for prediction. The design was tailored to accommodate the unique characteristics of the dataset and maximize model performance within resource constraints.

4. **Dynamic Learning Strategies:** To prevent overfitting and enhance model adaptability, dynamic learning rate scheduling with exponential decay was employed. This strategy allowed for fine-tuning the learning process and optimization of model convergence.

5. **Optimization Challenges and Solutions:** The team encountered model bias and scalability challenges, particularly given the imbalanced data. We successfully mitigated bias issues and improved model accuracy across different label ranges through experimentation with weighted backpropagation and discontinuous weighting mechanisms. These findings underscore the importance of thoughtful optimization strategies in machine learning model development.

References

[1] Petfinder pawpularity score. `https://www.kaggle.com/competitions/petfinder-pawpularity-score/overview`.

[2] Saiji Fu, Duo Su, Shilin Li, Shiding Sun, and Yingjie Tian. Linear-exponential loss incorporated deep learning for imbalanced classification. *ISA transactions*, 140:279–292, 2023.

[3] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016.

[4] Kaichao You, Mingsheng Long, Jianmin Wang, and Michael I Jordan. How does learning rate decay help modern neural networks? *arXiv preprint arXiv:1908.01878*, 2019.