PetFinder.my - Pawpularity Contest

Predicting Pet Recommendation Scores To Improve Animal Welfare

Guoquan Lin, Otto Gaulke, Yen Chen Hsu, Wei-Chun Chang, Joyce Wu

MSBA 6421
Professor Yicheng Song

Table of Contents

Project Introduction **3**
Solution Overview **3**
Methodology **4**
Exploratory Data Analysis **4**
Training Data Structure 4
Pawpularity Score 5
Metadata Correlation with Pawpularity Dependent Variable 5
Training Image 6
Data Pre-Processing **7**
Model Building **8**
Solution 1: Custom CNN Regression Model 8
Solution 2: Pre-Trained CNN Regression Model 10
Solution 3: Classification Solution 12
Model Training **13**
Model Evaluation **15**
Model Performance Summary **16**
Business Application **17**
Future Step **18**

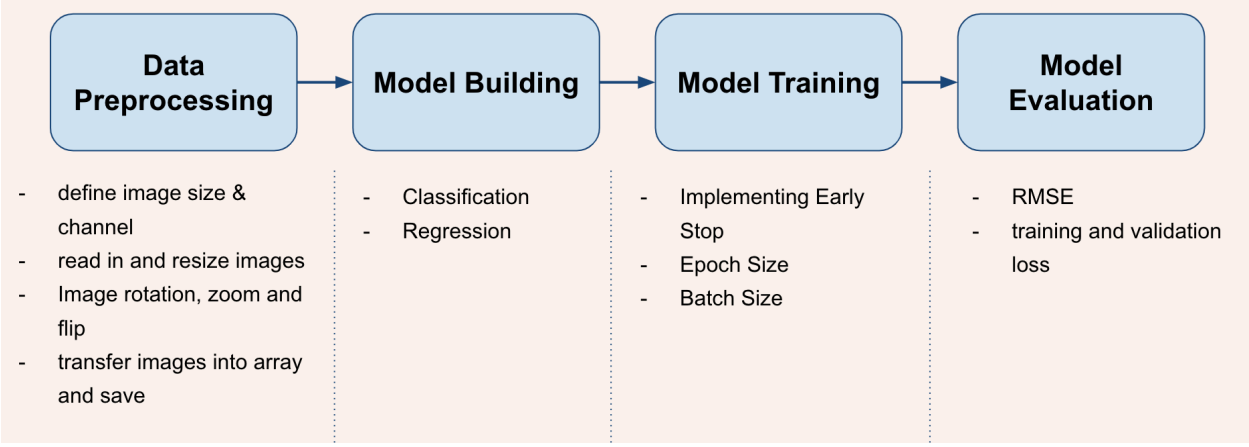
Project Introduction

PetFinder.my, a leading animal welfare platform in Malaysia that rehomes strays, empowers rescuers, and develops technology solutions for animal welfare. As of 2023 December, PetFinder is featuring 200k pets on their platform, and they have helped 70k pets find their new home. This project aims to develop an AI-based tool to enhance the appeal of pet photos on PetFinder's website, which will aid in increasing the chance of adoption for stray and shelter animals. By analyzing raw images and metadata of dogs and cats, our solution offers a deep learning algorithm that can predict the "Pawpularity" of pet photos, aside from PetFinder, the solution would also guide worldwide shelters and rescuers to save more innocent lives.

Solution Overview

In our mission to develop an effective solution for enhancing the appeal of pet photos on PetFinder's platform and increasing adoption rates, we embarked on a multi-step approach leveraging Convolutional Neural Networks (CNNs). Initially, our attempt to construct a custom CNN yielded suboptimal results. Recognizing the power of pretrained models, we transitioned to utilizing Keras' pretrained models, beginning with a regression version. Interestingly, our initial intuition suggested that metadata might not contribute significantly, as it could already be encapsulated in the pretrained model. To explore alternative avenues, we also reframed the problem as a classification task, treating the 100 integers of "Pawpularity" scores as 100 categories. Our incorporation of metadata through Support Vector Regression (SVR) proved unhelpful, affirming our earlier speculation. After extensive experimentation, we found that employing the regression-based approach led us to the most accurate predictions for our model with the lowest Root Mean Square Error (RMSE) score. This comprehensive exploration of model architectures and data representations underscores our commitment to delivering a robust AI-based tool for PetFinder, poised to make a meaningful impact on the lives of stray and shelter animals globally.

Methodology



Exploratory Data Analysis

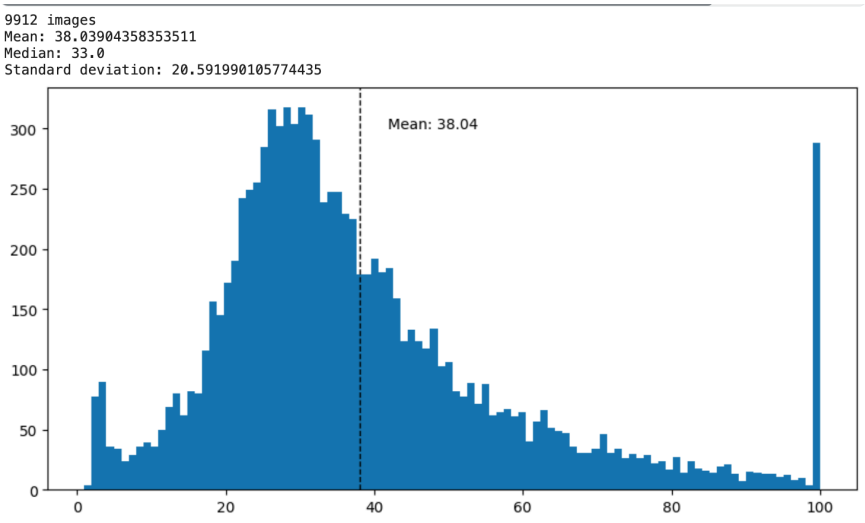
Training Data Structure

The data set contains training and test image files and CSV files of pets, the CSV contains the metadata of the images: 12 binary variables showing the image features (i.e. whether the pet has eyes/ face/ action.. etc) and 1 Pawpularity score column. Our objective is to predict the Pawpularity score for respective images.

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 9912 entries, 0 to 9911
Data columns (total 14 columns):
 #   Column         Non-Null Count  Dtype
---  ------         --------------  -----
 0   Id             9912 non-null   object
 1   Subject Focus  9912 non-null   int64
 2   Eyes           9912 non-null   int64
 3   Face           9912 non-null   int64
 4   Near           9912 non-null   int64
 5   Action         9912 non-null   int64
 6   Accessory      9912 non-null   int64
 7   Group          9912 non-null   int64
 8   Collage        9912 non-null   int64
 9   Human          9912 non-null   int64
 10  Occlusion      9912 non-null   int64
 11  Info           9912 non-null   int64
 12  Blur           9912 non-null   int64
 13  Pawpularity    9912 non-null   int64
dtypes: int64(13), object(1)
memory usage: 1.1+ MB
```

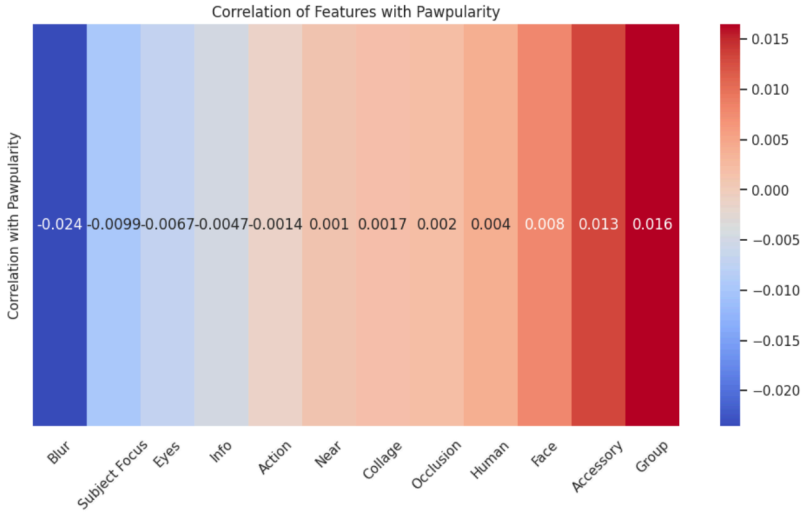
Pawpularity Score

In the training data set, we got 9912 images, with Pawpularity mean of 38.08, and standard deviation of 20.59. Note that Pawpularity score is integer, not float.



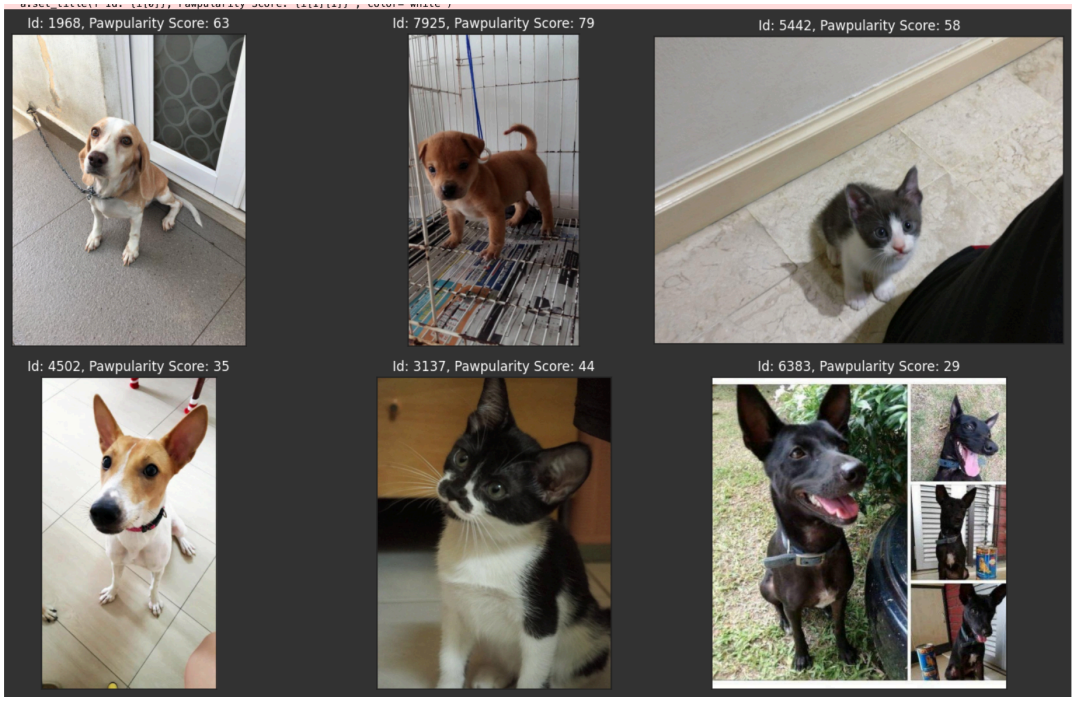
Metadata Correlation with Pawpularity Dependent Variable

Here, we explore the correlation between metadata features and popularity score, we applied Pearson correlation as our calculation metrics, and we observed a mild correlation between the categorical variables and Pawpularity. From the correlation heat map, we cannot find a distinctive relationship between meta and Pawpularity score. Furthermore, since the feature in the meta might already be captured in the image itself, we made a strong assumption that this metadata would not be really helpful in our modeling process.



Training Image

Here are some images and Pawpularity of our training data:



Data Pre-Processing

First, the dimensions of the images under consideration were specified.

Here we define a function to deal with the test images since we're going to read in the test set only when submitting the notebook, not read in them in advance.

```
# declare our image dimensions using color images

img_size = 250
channels = 3  # change to 1 if need to use grayscale image

# define function to read and process the images to an acceptable format for our model
train_score = pd.read_csv('pawpularitydataset/train.csv') # the pawpularity score is in this csv file

def read_and_process_image(list_of_images):
    X = [] # an array of resized images
    y = [] # an array of score

    for i, image in enumerate(list_of_images):
        X.append(cv2.resize(cv2.imread(image, cv2.IMREAD_COLOR), (img_size, img_size), interpolation=cv2.INTER_CUBIC)) # read the image
        y.append(train_score.Pawpularity[i]) # get the score

    return X, y
```

Next, preserve training arrays and labels for expedited loading and manipulation during subsequent model operations.

Mount my Google drive to save the processed training arrays and labels

So that I only need to upload the array.npy and label.npy to kaggle to train the model instead of redo data preprocessing everytime.

```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

```
# save to google drive so that don't need to load the image each time
np.save("/content/drive/MyDrive/Colab Notebooks/group250/training_X.npy", X)
np.save("/content/drive/MyDrive/Colab Notebooks/group250/training_y.npy", y)
```

Subsequently, the training data and corresponding labels were imported.

```
X_TRAIN = np.load("/kaggle/input/training250/training_X.npy", allow_pickle = True)
y_TRAIN = pd.read_csv("/kaggle/input/categorical-score-no-metadata/with_category_score.csv")

print("shape of train images:", X_TRAIN.shape)
print("shape of labels:", y_TRAIN.shape)
```

```
shape of train images: (9912, 250, 250, 3)
shape of labels: (9912, 100)
```

Following that, a partition was performed, segregating the data into training and validation sets.

```
# split the data into train and validation set
from sklearn.model_selection import train_test_split
X_train, X_val, y_train, y_val = train_test_split(X_TRAIN, y_TRAIN, test_size=0.20, random_state=42)

print("shape of train images:", X_train.shape)
print("shape of validation images:", X_val.shape)
print("shape of labels:", y_train.shape)
print("shape of labels:", y_val.shape)
```

```
shape of train images: (7929, 250, 250, 3)
shape of validation images: (1983, 250, 250, 3)
shape of labels: (7929, 100)
shape of labels: (1983, 100)
```

Subsequent to the data partitioning, an augmentation process was applied exclusively to the training image data, preceding its utilization in the model.

```
from tensorflow.keras.preprocessing.image import ImageDataGenerator

# this would helps prevent overfitting, since we are using a small dataset
train_datagen = ImageDataGenerator(rescale = 1./255,   # scale the image between 0 and 1
                                   rotation_range = 60,
                                   width_shift_range = 1.0,
                                   height_shift_range = 1.0,
                                   shear_range = 0.4,
                                   zoom_range = [0.1, 2],
                                   horizontal_flip = True,
                                   vertical_flip = True,
                                   fill_mode='nearest')

val_datagen = ImageDataGenerator(rescale = 1./255)  # do not augment validation data. we only perform rescale

# create the image generators
train_generator = train_datagen.flow(X_train, y_train, batch_size=batch_size)
val_generator = val_datagen.flow(X_val, y_val, batch_size=batch_size)
```

Model Building

Solution 1: Custom CNN Regression Model

We started the model building processes by constructing our own CNN model for regression from scratch. This provided us with a baseline to compare with future models, so rather arbitrary settings were chosen as a simple starting point. We began by constructing the convolutional layers. Different numbers layers were chosen but eventually it was decided that five sufficiently extracted important image features. The first hidden layer was set to accept an image array shape of 250 by 250 by 3. It was decided that the number of filters would increase by a factor of two for each convolutional layer. Layer one was assigned 32 filters, layer two was assigned 64 filters, and so on until the fifth layer which was assigned 512 filters. A stride length of one by one was used with padding set to ‘same’ so as to retain the edges of the images. ReLU activation was used in each convolutional layer. Maximum pooling and batch normalization was used between

each convolutional layer. A dropout setting of 0.25 was applied to the first three convolutional layers, and 0.5 to the rest to reduce model complexity and avoid overfitting.

```
img_size = 250
```

```
# convolutional layer 1
model.add(Conv2D(filters=32, kernel_size=3, data_format='channels_last', input_shape=(img_size, img_size, 3), padding='same', strides=1))
model.add(BatchNormalization())
model.add(Activation("relu"))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
```

Next, after the flatten layer, we built the dense hidden layers. Five dense layers were added to begin with reasonable model complexity. Neurons were added in descending complexity by a factor of one half; the first layer was assigned 512, the second layer was assigned 256, and so on until the final layer was assigned 32 neurons. The rest of the settings from the convolutional layers were applied to the dense layers. Finally, the output layer was assigned one neuron with a ReLU activation function to predict Pawpularity.

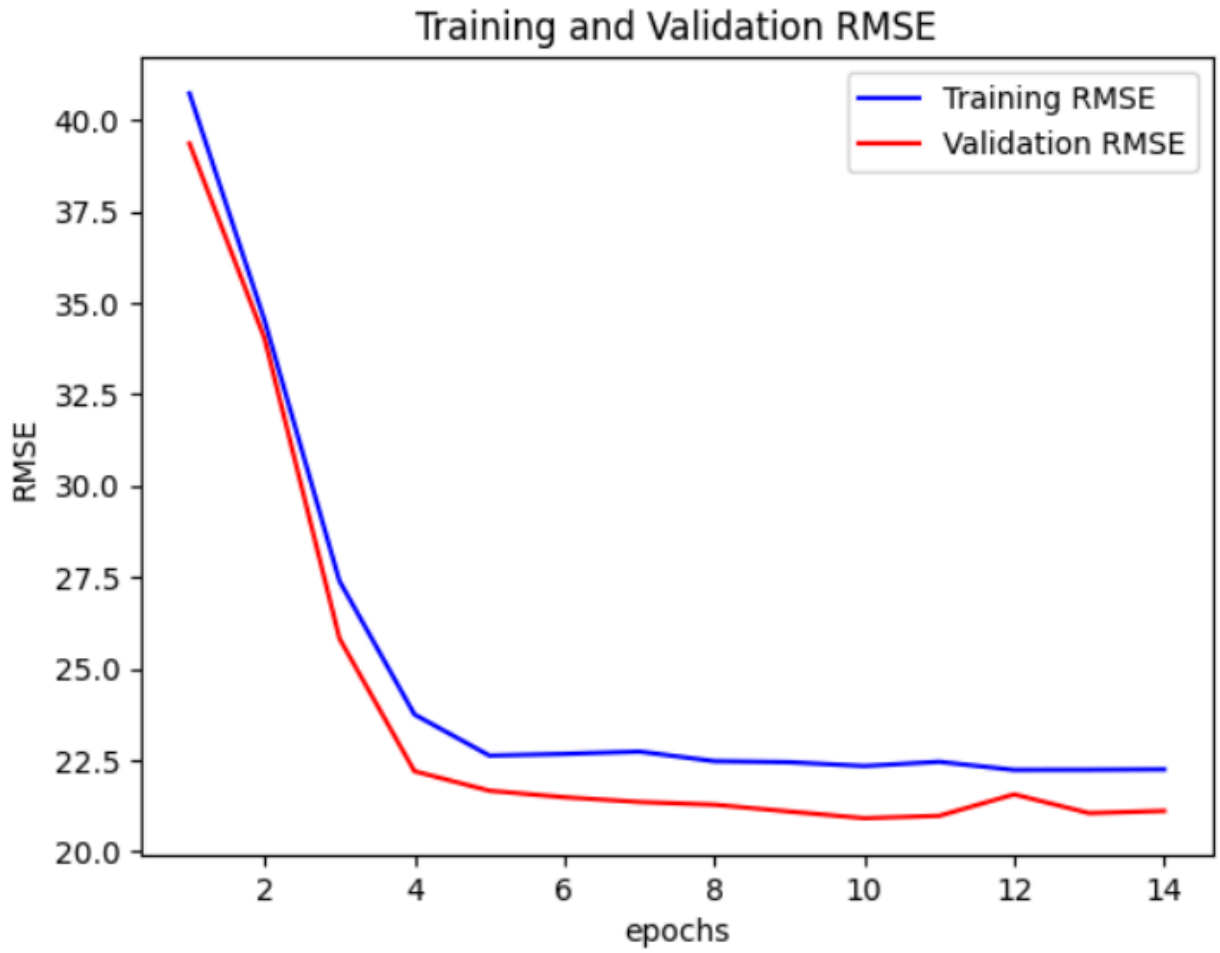
```
# dense layer 5
model.add(Dense(units=32))
model.add(BatchNormalization())
model.add(Activation("relu"))
model.add(Dropout(0.5))

# dense layer 6, i.e. output layer (size=1 for regression)
model.add(Dense(units=1, activation='relu'))
```

Finally, the custom CNN model was compiled. Adam was used as an optimizer. MSE was used as a loss function and RMSE was used as a performance metric as is commonly used in regression tasks.

```
# compile
model.compile(optimizer='adam', loss='mse', metrics=[tf.keras.metrics.RootMeanSquaredError()])
```

Solution 2: Pre-Trained CNN Regression Model



. We can see that validation loss is around 21, which might not be good enough, so we want to try the pretrainined model under Keras structure.

From the plot we can see that loss in train set and validation set decreased gradually in similar patterns as training epochs increased, and the training process was interrupted by early stopping after 21 epochs, so we got 8-21 epochs.

Also, we noticed that the validation loss is around 21, which might not be good enough, so we want to try the pretrained model under Keras structure. Several pre-trained models were used: InterceptionResNetV2, EfficientNetB0 - EfficientNetB7, and XceptionNet. To begin we used an input layer set to match our image data dimensions: 250 by 250 by 3. Then we added the KerasLayer() function to deploy the desired pre-trained model.

```
model_name = "efficientnetv2-b4"

model_handle_map = {'efficientnetv2-b4': '/kaggle/input/efficientnet/tensorflow2/b4-feature-vector/1'}

model_image_size_map = {
  "efficientnetv2-b4": img_size,
}

model_handle = model_handle_map.get(model_name)
pixels = model_image_size_map.get(model_name, img_size)

print(f"selected model path: {model_name} : {model_handle}")

IMAGE_SIZE = (pixels, pixels)
print(f"input size {IMAGE_SIZE}")
```

```
model = tf.keras.Sequential([
    # explicitly define the input shape: (250, 250, 3)
    tf.keras.layers.InputLayer(input_shape=IMAGE_SIZE + (3,)),

    # set trainable = True for fine tune
    hub.KerasLayer(model_handle, trainable = True),
```

Next, we added the hidden dense layers. Fourteen dense layers were added for sufficient layer complexity. Neurons between 32 and 512 were set to the layers using a similar method as our custom built model. Batch normalization was used, and a droprate of 0.5 was applied to each layer (some were 0.12, 0.6, and 0.7). Through experimentation it was found that a mix of LeakyReLU and ReLU activation functions resulted in the best training and validation performances. An output layer with a ReLU activation function was used for regression prediction of Pawpularity scores.

```
# dense layer 14
tf.keras.layers.Dense(units=32),
tf.keras.layers.BatchNormalization(),
tf.keras.layers.Activation("relu"),
tf.keras.layers.Dropout(0.25),

# dense layer 15, output layer
tf.keras.layers.Dense(units=1, activation='relu') # dimension for output is 1 for regression problem
```

Finally, the models were compiled. Nadam was used as an optimizer, and this was decided upon after experimentation. MSE was used as a loss, and RMSE was used as a performance metric. And in the training and validation phase, RMSE could be reduced to around 20. Nevertheless, wanting to try other alternative avenues, we also tried to view the problem as a classification tasks in the following.

```
# compile the model, using 'mse' as loss function
# since this version is dealing with regression problem
model.compile(optimizer='nadam',
              loss='mse',
              metrics=[tf.keras.metrics.RootMeanSquaredError()])
```

Solution 3: Classification Solution

Treat Pawpularity Score Prediction as a Classification Task:

Aside from the regression solution, we also tried to build the model with a classification model. To be more specific, we found out that the data type Pawpularity score is integer, hence, we came up with the idea of treating 1~100 score as categorical. And we built a classification CNN model to predict the Pawpularity score of each image.

For the classification model, we still used efficientnetv2-b4 as our transfer learning base and added 10 dense layers after the pre-trained model.

```
model_name = "efficientnetv2-b4"
model_handle_map = {'efficientnetv2-b4': '/kaggle/input/efficientnet/tensorflow2/b4-classification/1'}
model_image_size_map = {"efficientnetv2-b4": img_size}

model_handle = model_handle_map.get(model_name)
pixels = model_image_size_map.get(model_name, img_size)

print(f"selected model path: {model_name} : {model_handle}")

IMAGE_SIZE = (pixels, pixels)
print('input size:', IMAGE_SIZE)
```

```
selected model path: efficientnetv2-b4 : /kaggle/input/efficientnet/tensorflow2/b4-classification/1
input size (250, 250)
```

```
    # dense layer 6, output layer
    tf.keras.layers.Dense(units=100, activation='softmax') # we view the 100 Pawpularity score as 100 categories
])
model.build((None,)+IMAGE_SIZE+(3,))

# check the model structure
model.summary()
```

The output layer is different from the regression model, we utilize “softmax” as our activation function since we are viewing the 100 Pawpularity scores as 100 categories. In total, we are using 20 million trainable parameters.

Considering Meta Data in Prediction:

We also try to see if the performance can improve with the help of meta data. We trained a SVR model where we regressed Pawpularity Score on features to generate a prediction Pawpularity Score.

Lastly, we will take the average of the score made by the classification CNN model and the SVR score as our final Pawpularity Score.

First, we load our metadata and conduct the train test split.

```
df = pd.read_csv('/kaggle/input/petfinder-pawpularity-score/train.csv')
df = df.drop(columns = 'Id')

X_train_SVR = df.iloc[:, 0:12]
y_train_SVR = df.iloc[:, 12]

from sklearn.model_selection import train_test_split
X_train_svr, X_val_svr, y_train_svr, y_val_svr = train_test_split(X_train_SVR, y_train_SVR, test_size=0.2, random_state=42)
```

We then conduct the grid search to discover the optimal hyperparameter for the SVR model, here we obtain the hyperparameter with {'C': 0.1, 'epsilon': 10, 'gamma': 1, 'kernel': 'rbf'}, with a root mean square error (RMSE) of 20.644052638806244. (To save time for final auto-grading in the system, we did the GridSearch in advance in another notebook, we will attach that in the appendix)

```
# define the hyperparameter grid
svr_grid = {'kernel': ['rbf'], 'C': [10, 1, 0.1], 'epsilon': [10, 1, 0.1], 'gamma': [10, 1, 0.1, 0.01]}
```

```
# define the svm regressor: SVR
svr = svm.SVR() # for svr, y is expected to have floating point values instead of integer values
```

```
svr_clf = GridSearchCV(estimator = svr, param_grid = svr_grid, scoring = 'neg_root_mean_squared_error', cv = 10, refit = True)
svr_clf.fit(X_train_svr, y_train_svr)
print("Best hyperparameters settings: ", svr_clf.best_params_)
print('RMSE: ', -svr_clf.best_score_)
```

```
Best hyperparameters settings:  {'C': 0.1, 'epsilon': 10, 'gamma': 1, 'kernel': 'rbf'}
RMSE:  20.644052638806244
```

Model Training

Batch Size:

```
# set up small batch to avoid run out of memory when allocating
batch_size = 32
```

Batch size is a hyperparameter that specifies the number of training examples utilized in one iteration by the neural network in each training step. A larger batch size may provide a more stable learning process, as the gradient estimates are based on a larger and potentially more representative set of samples. However, it also requires more memory, as all the samples in the batch must be stored. Typical batch sizes are powers of 2, such as 32, 64, 128, etc. Weighing in on computational resources, our final setting for batch size is 32.

Early Stop:

```
# define the early stopping callback
early_stopping = EarlyStopping(monitor='val_loss',
                               patience=4,
                               restore_best_weights=True)
```

To prevent overfitting and to stop training when further iterations are unlikely to improve the model's generalization performance. We implemented an EarlyStopping callback that monitors the validation loss during training. If there is no improvement in validation loss for 4 consecutive epochs, the training will stop, and the model will be restored to the weights that provided the best performance on the validation set.

```
# train the model
history = model.fit(train_generator,
                    steps_per_epoch = len(X_train) // batch_size,
                    epochs=30,
                    initial_epoch = 7,
                    validation_data=val_generator,
                    validation_steps = len(X_val) // batch_size,
                    callbacks=[early_stopping])
```

Epochs:

During an epoch, the neural network processes every example in the training set exactly once, both forward (through the network to make predictions) and backward (to calculate gradients and update weights). The model will be trained for 30 epochs.

Steps Per Epochs:

This is the number of batches of training samples to process in one epoch. It is set to the total number of training samples divided by the batch size (len(X_train) // batch_size).

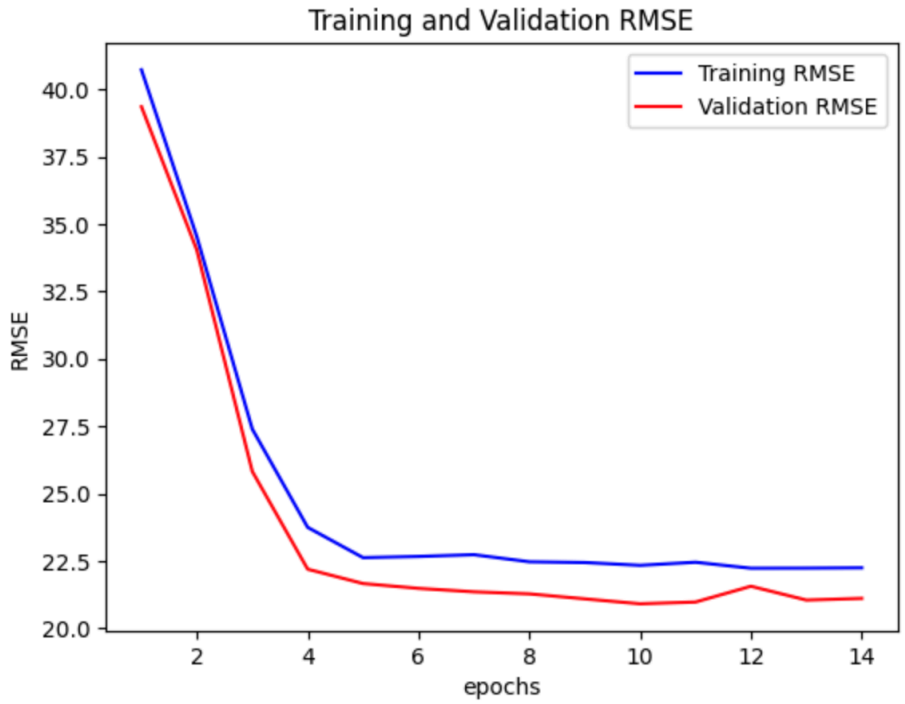
Initial Epochs:

We used this parameter to specify from which epoch the training process should start. Instead of starting the training process from the first epoch (epoch 1), it will resume training from the specified epoch, in our case, epoch 7. This helps us effectively extend the training process without starting from scratch. This parameter is particularly useful when we have limited computational resources and want to perform training in multiple stages or when implementing a training schedule with different learning rates or other hyperparameters at different stages of training.

Model Evaluation

For this project, we are using Root Mean Square Error (RMSE) score as our evaluation metrics. Our goal is to predict Pawpularity score, which is a continuous numerical value. This measures the average magnitude of the errors between model predicted score and actual score.

Out of all the models we've experimented, our best model yields a RMSE of 20.51. From the below graph you can see that as we continue into further iterations of CNN, both our training and validation RMSE gradually decreases. RMSE stabilized before reaching the setted 30 epochs, and the training process was interrupted by early stop.



Model Performance Summary

Here's a summary table of model performances from our experimentation, which includes only the major version of our models after hyperparameters are fine-tuned.

Model Name	**CV score**
self -built CNN	20.767
self-built ResNet-34 CNN	42.500
efficientnet_b0	20.546
efficientnet_b1	20.700
efficientnet_b2	71.269
efficientnet_b3	20.670
efficientnet_b4	20.510
efficientnet_b5	32.594
efficientnet_b6	50.600
inception_resnet_v2	20.744
Only svr for metadata	21.170
resnot_50	20.920

Business Application

The application of our model involves the following business scenarios:

1. Enhanced Pet Photo Appeal:
    - Implement our deep learning algorithm to predict the "Pawpularity" of pet photos, thereby providing a quantitative measure of their appeal.
    - Utilize this information to prominently feature pets with higher "Pawpularity" scores on the platform, increasing their visibility and chances of adoption.
2. Personalized Adoption Recommendations:
    - Leverage the algorithm's predictions to offer personalized recommendations to potential adopters, aligning their preferences with pets that have a higher likelihood of being adopted.
3. Global Impact:
    - Extend the solution beyond PetFinder.my to provide guidance and insights to shelters and rescuers worldwide.
    - Share the developed algorithm and best practices with global entities, contributing to the broader effort of saving innocent lives and increasing adoption rates for animals in need.
4. User Engagement and Experience:
    - Enhance the overall user experience on PetFinder's platform by presenting users with pets that align with their preferences, improving engagement and interaction.
5. Data-Driven Decision Making:
    - Enable PetFinder and other global shelters to make data-driven decisions on which pets to feature prominently, optimizing the adoption process.

In summary, our model's predictive capabilities offer a transformative solution for PetFinder.my, revolutionizing how pets are presented on the platform and increasing the chances of successful adoptions. Furthermore, the global applicability of this tool positions it as a valuable asset for animal welfare organizations worldwide.

Future Step

After consulting with Professor, we've come up with several directions to improve the model performance:

1. Although some of the better models have similar performance, we can still try stacking to see if the performance could be improved
2. Since we use random hyperparameters in the hidden layers after the convolutional layers and flatten layer, a better way to utilize the pre-trained model is:
    a. First set trainable = False to fix the good parameters in the pre-trained model and let hidden layers learn their parameters
    b. Then set trainable = True to train the convolutional layers as well
    c. Abandon the first several layers for more stable performance

Through this way we won't dilute the good parameters in the pre-trained model yet could also let it learn from our own training set.