IE4483 Artificial Intelligence and Data Mining
Mini Project Final Report

Dogs vs. Cats (Option 2)

Team Members:
- Jonathan Kelvin Santoso / U2020632E
- Egan Valentino / U1920271J

Project Summary:

Programming Language:	Python 3.9.12
Framework:	Tensorflow and Keras

We have chosen this Dogs vs. Cats problem as we wanted to learn more about computer vision. Our project flow is as follows:

1. Data Preparation, which includes augmentations and preprocessing,
2. Model Building and Training,
3. Model Evaluation and Tuning,
4. Test Prediction Generation.

For data preparation, we have augmented the training dataset, thus our overall training set is 40,000 images. In terms of preprocessing, we have included a rescaling layer to rescale images' RGB values.
For model building and training, we initially built our own Convolutional Neural Network (CNN) model, and further fine-tuned it. Our best validation accuracy from this model is 93.88%, averaged from 5 training rounds. We also explored transfer learning, using a pretrained ResNet50 model, which yielded a validation accuracy of 88.78%.
Before choosing our final model for the Dogs vs. Cats problem, we evaluated each model's performance after performing training rounds on them. We experimented with adding layers, changing optimisers, loss functions, learning rates, and freezing of pretrained layers.
The final model we chose for the Dogs vs. Cats problem is our CNN, as it has the highest averaged validation set accuracy of 93.38%. We created a function to generate a CSV file for the final prediction of the test set.

For the CIFAR-10 dataset task, we did several changes to our model, such that it is able to perform the task with relatively high accuracy. Most notably, we removed two max pooling layers, changed loss functions, and changed the output layer. We obtained test accuracy results of 81.72%

All our codes and datasets are available in the zip file for reproducibility and checking.

Contribution:

1. Egan Valentino:
   Written explanation part for CNN in question b
2. Jonathan Kelvin Santoso:

Created notebooks for data augmentation, CNN model training, ResNet50 model training, and CIFAR10 training. Performed data augmentation, including creating the pipeline. Conducted training rounds, evaluated model performances and did hyperparameter tuning on all models used in this project. Written functions for test prediction CSV. Choose the final model and analyze the model's strengths and weaknesses. Written the project report and other explanations aside from CNN explanation.

Project Details:
(Answers to questions)

a. ***State the amount of image data that you used to form your training set and testing set. Describe the data pre-processing procedures and image augmentations (if any).***

The initial dataset given for this project consists of 20,000 training data, 5,000 validation data, and 500 testing data. The class distributions can be seen in the table below:

	Cats	Dogs	Total
Training	10,000	10,000	20,000
Validation	2,500	2,500	5,000
Testing	-	-	500

A. Data Augmentation
Data augmentation is the process of doing transformation on the original data with the goal of generating more data and subsequently increasing the training set size.

Training Dataset:
We had opted to perform data augmentation on the 20,000 training images. We believe that as more data is used for the model training, our model will subsequently be able to generalize better to unknown data. This will lead to a more robust model and a higher accuracy overall.

To do this, we used Albumentations, a Python library for image augmentations. The image augmentations that we used:

- Channel Shuffle:
  Randomly rearrange channels of the input RGB image. We set the probability of this augmentation to be 0.5.
- Hue Saturation Value (HSV):
  Randomly change hue, saturation and value of the input image. We set the probability of this augmentation to be 0.5.
- Random Rotate:
  Randomly rotate the input by 90 degrees zero or more times. We set the probability of this augmentation to be 0.5.

- Shift Scale and Rotate:
  Randomly apply affine transforms: translate, scale and rotate the input. We set the probability of this augmentation to be 0.75.
- Blur:
  Blur the input image using a random-sized kernel. We set the maximum kernel size for blurring the image to be 3, and the probability of this augmentation to be 0.75.
- Random Brightness Contrast:
  Randomly changes the brightness and contrast of the input image. We set the probability of this augmentation to be 0.5.
- Horizontal Flip:
  Randomly flips the image horizontally. We set the probability of this augmentation to be 0.5.
- Vertical Flip:
  Randomly flips the image vertically. We set the probability of this augmentation to be 0.5.

The way we augmented the training set, was we created an augmentation pipeline, such that every image that passes this pipeline, will have a possibility of being augmented by the above augmentations. For 1 input, the pipeline will have only 1 output. Thus, there is a probability that an image is augmented by all the augmentations, or not augmented at all. However, the probability of the latter is very low. We iterated through the training set to get an augmented output, thus, obtaining our final training dataset of 40,000 images, consisting of the original 20,000 images and 20,000 augmented images.

Technical Implementation:
We used Albumentation's Compose function to build the augmentation pipeline. The os, cv2, and shutil libraries were also used to access files, read and write images, and copy images respectively. Before iterating through the dataset, we fixed a random seed of 42 to ensure that our augmentations are reproducible. Both the notebook used for data augmentation and the final training dataset used are available in the zip file.

Validation Dataset:
We chose not to augment the validation dataset, as our main purpose of data augmentation is to increase the input training data and help the model achieve a greater accuracy.

Test Dataset:
We did make any changes to this dataset as it is used to benchmark our model.

It should also be noted that data augmentation comes with its limitations, namely still having the inherent bias of the original data.

B. Data Preprocessing

We preprocessed the input RGB values by rescaling it by 1./255. As the original input image is in RGB, it will have values in the range 0-255. This is normally too high for a model to process, thus we normalize the target values to be in the range of 0 and 1. This is done in the model itself, as we used a rescaling layer as the first layer of our model.

C. Conclusion

Final ratio of dataset:

- Training set:	40,000	(~88%)
- Validation set:	5,000	(~11%)
- Testing set:	500	(~1%)

We also ensured that the class distribution of Dogs and Cats are equal both in the training and validation dataset. This is such that the model does not have a bias towards the majority class, which could lead to bad accuracy for the minority class.

b. Select or build at least one machine learning model (e.g., CNN + linear layer, etc.) to construct your classifier. Clearly describe the model you use, including a figure of model architecture, the input and output dimensions, structure of the model, loss function(s), training strategy, etc.
Include your code and instructions on how to run the code. If a non-deterministic method is used, ensure reproducibility of your results by averaging the results over multiple runs, cross-validation, fixing random seed, etc.

To tackle the Dogs vs. Cats problem, we first built a Convolutional Neural Network (CNN) and adjusted the layers and hyperparameters. We also used a pretrained model, namely the Residual Networks (ResNet). Before we go into detail on our model architecture, we will first explain more about CNNs and how they work:

Explanation

Convolutional neural network is a class of artificial neural network and one of the most influential innovations in the field of computer vision and image processing. How the computer visualizes an image for CNN input would have to be in a representative array of its pixel size, and the value of each number inside the array would be a value ranging from 0 to 255, depending on the RGB pixel intensity at that point.

When we use CNN, the aim is for the computer to take in a certain array of numbers and output a probability that allows us to understand what a certain result would be (e.g. an image of a cat could have the computer understand it being 80% cat and 20% dog, hence, since the cat has a higher percentage, it decides to conclude the image as a cat). This way of visualizing reflects the way our brain works. For instance, when looking over an image of a cat, some identifiable features that could be found (e.g. having 4 legs, paws, a tail, pointy ears, etc.) allow us to distinguish it as a cat. Correspondingly, low-level features (e.g. edges, curves, etc.) could also be identified by a computer allowing it to perform image classification on the picture, subsequently accumulating to a more abstract and complex concept through a series of convolutional layers. Low-level features are used initially to gradually understand high-level features (e.g. paws, ears, eyes, etc.) that conclude what an image is categorized in detecting the pattern in the process. In general, CNN contains 3 layers that enables it to work:

1. Convolutional layer & ReLU Layer

In this stage, images are normally rescaled first to an optimal size and fed to the convolutional layer. There needs to be a specified number of filters (i.e. the one detecting the pattern such as edges and curves) each layer should have. This filter could be thought of as just a small matrix in which the users would decide the number of rows and columns it should consist of, with values initialized using random numbers.

Consider an example where the user chooses to use 1 filter consisting of a 3 x 3 matrix for the first convolutional layer. When an input of 28 x 28 matrix is fed, the filter would convolve a given input by sliding through each 3 x 3 array in the input calculating a dot product with it every step of the way starting from the top left corner all the way to the bottom right corner. The dot product would then generate an output of a 26 x 26 matrix feature map. This continues until a certain number of layers are reached.

When each dot product generates a certain value that does not equate to zero, it indicates that a specific part of the input causes the filters to activate, showing that it resonates with what the filter intended to do. As an example, if a filter is intended to detect a curve, the non-zero value shows that the input has a part that contains a curve feature causing the filter to activate. To generalize, the more filters, the greater the depth of the activation map, the more information the users have regarding the input volume.

The order of the convolution layer is such that the image array would be the input of the first convolution layer. Subsequently, the feature map generated by the first layer becomes the input for the second layer identifying different low-level patterns throughout the process. As more convolutional layers are used, plus having an extra set of filters to top it off, feature maps would represent increasingly complex features (e.g. a combination of shapes, eyes of animals, etc.). Adding the ReLU layer on top of the convolutional layer, a node would only activate if it is above a certain threshold or value.

2. Pooling layer

The pooling layer is a layer of operation that is normally added to CNN following the convolutional layer. When max pooling is added to a model, it reduces the dimensionality of images by reducing the numbers of pixels in the output from the previous convolutional layer. User needs to define the filter size and stride size (i.e. the number of block it would move along the matrix) to achieve its objective

Consider a continuation of the above sample of the 26 x 26 matrix feature map from the first convolutional layer as the input of the max pooling layer and the user decides to use a 2 x 2 matrix as the filter and a stride size of 2. The filter would slide across the input but instead of calculating the product, it would choose the max value in that 2 x 2 block and move both horizontally and vertically along the input in blocks of 2 generating a subsequent 13 x 13 matrix as the output.

Max pooling is essential since it reduces the resolution of the convolutional layer output (i.e. feature map), hence, reducing the amount of parameters in the network for the computer to be able to look at a larger areas of images at a time, thus, reducing the computational load, plus it reduces overfitting as well. This could work due to the nature that max pooling takes the maximum value/pixel in the filter size, thus, preserving an area that is the most activated in which it is highly likely to be the most relevant to the computer understanding of the features, discarding lower-valued pixels that are not as activated.

3. Fully connected layer

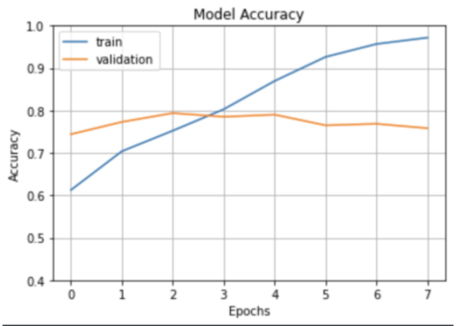
The fully connected layer is a layer where neurons of previous layers are connected to every neuron in subsequent layers. In this layer, all possible pathways from the input and output are considered, mirroring how our brain functions when deciding on what a particular image is based on our knowledge. This layer looks at what high-level features are present in a particular image and checks which of those have the most correlations to the image. This is crucial to be able to correctly compute the different probabilities of what image it would be based on the datasets given.

To understand how the filters are initialized with their own specificity, we must look at how the training works. This part consists of forward pass, loss function, backward pass, and weight update. In the forward pass, after the first iteration, no specific weights are given to any of the dataset probabilities (i.e. all of them would have the same probabilities). Going to the loss function, images are trained with labels on them, hence, making it clear with a 100% probability of what the image is showing. In contrast, output following the first iteration is very different, hence, loss is calculated with the goal of minimizing it, enabling the models to adjust the weight/probability of what an image would be from a given dataset. Through the backward pass, the most significant loss contributor would be detected and minimized. This could be calculated using dL/dW, where L is the loss and W is the weight of the corresponding filters (which indicates the feature it is trying to detect). Finally, the weight update keeps on adjusting the probability to find the best fit for what the image is supposed to be. In this part, the learning rate is determined (higher learning rate suggests more steps and more time towards optimization of weights).

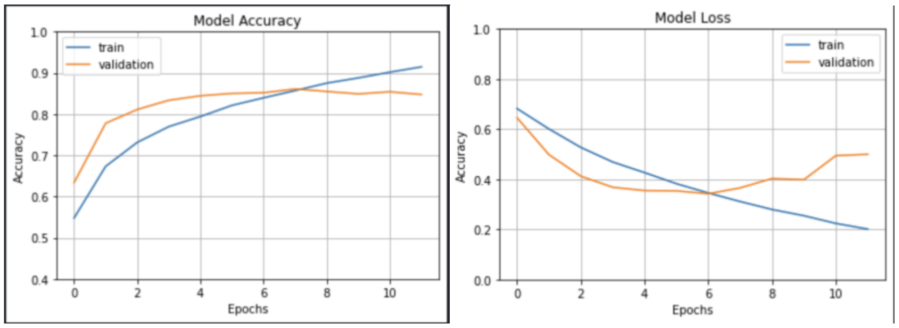
Model Description

I. Our CNN

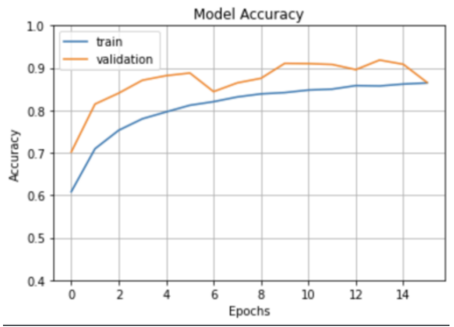
We first developed our model with 3 convolution layers and 3 max pooling layers. We then have a fully connected / dense layer and output layer that uses a sigmoid function to perform binary classification. We achieved an accuracy of 75.82%.



We noticed that the model had low validation accuracy and was overfitting, as the training accuracy was increasing while validation accuracy slightly decreased. Thus, we added 2 more pairs of convolution and max pooling layers. With this, we reached an accuracy of 84.76%.



We saw that the model's validation accuracy increased, as expected, however, there is still some overfitting, indicated by the low validation accuracy and high training accuracy. It was also shown by the loss graph, where validation loss started to increase towards the end of the training. This is an overfitting problem, thus, we added dropout and batch normalization layers to reduce overfitting. The accuracy we achieved with this was in the range of 86-91%, with our final accuracy to be 86.58%.



We then tried to add another block of convolution, drop out, and batch normalization layers, with the hopes of increasing both training and validation accuracies. However, the results did not differ from our previous model without this, thus we decided to remove it. We suspect that it is due to the input shape going into the convolution layer being too small, which led to an ineffective convolution.

After not being able to add any more layers to improve the model's performance, we proceeded to test out different optimizers, as we were using RMSprop initially. We switched to Adam, as it allowed the validation loss to decrease even further, which led to a higher validation accuracy. Top accuracy we achieved with this was 93.88%, while average accuracy was 93.38% out of 5 training rounds. Details of our final CNN model:

a. Model Architecture:

No	Layer	Input Shape	Output Shape	Description
1	Rescale	128, 128, 3	128, 128, 3	Normalizes RGB values to be in the range of 0-1
2	Convolution	128, 128, 3	126, 126, 32	Filters=32, kernel=3x3, activation function=ReLU
3	MaxPooling	126, 126, 32	63, 63, 32	Pool size=2x2
4	BatchNorm	63, 63, 32	63, 63, 32	Normalizes batch data
5	Dropout	63, 63, 32	63, 63, 32	Probability=0.25
6	Convolution	63, 63, 32	61, 61, 64	Filters=64, kernel=3x3, activation function=ReLU
7	MaxPooling	61, 61, 64	30, 30, 64	Pool size=2x2
8	BatchNorm	30, 30, 64	30, 30, 64	Normalizes batch data
9	Dropout	30, 30, 64	30, 30, 64	Probability=0.25
10	Convolution	30, 30, 64	28, 28, 64	Filters=64, kernel=3x3, activation function=ReLU
11	MaxPooling	28, 28, 64	14, 14, 64	Pool size=2x2
12	BatchNorm	14, 14, 64	14, 14, 64	Normalizes batch data
13	Dropout	14, 14, 64	14, 14, 64	Probability=0.25
14	Convolution	14, 14, 64	12, 12, 64	Filters=64, kernel=3x3, activation function=ReLU
15	MaxPooling	12, 12, 64	6, 6, 64	Pool size=2x2
16	BatchNorm	6, 6, 64	6, 6, 64	Normalizes batch data
17	Dropout	6, 6, 64	6, 6, 64	Probability=0.25
18	Convolution	6, 6, 64	4, 4, 128	Filters=128, kernel=3x3, activation function=ReLU
19	MaxPooling	4, 4, 128	2, 2, 128	Pool size=2x2
20	BatchNorm	2, 2, 128	2, 2, 128	Normalizes batch data

21	Dropout	2, 2, 128	2, 2, 128	Probability=0.25
22	Flatten	2, 2, 128	512	Flattens input array into 1 dimension
23	Dropout	512	512	Probability=0.5
24	Fully Connected	512	512	Activation function=ReLU
25	Fully Connected	512	1	Activation function=Sigmoid

Total Parameters:	431,681
Trainable Parameters:	430,977
Non-trainable Parameters:	704

b. Loss Function:
We used the binary cross entropy loss function for this task, as we were performing binary image classification. Cross entropy quantifies the difference between two probability distributions. Our model predicts a model distribution of {p, 1-p} as we have a binary distribution. We use binary cross-entropy to compare this with the true distribution {y, 1-y}.

Binary cross entropy = -(ylog(y_pred) + (1-y)log(1-y_pred))

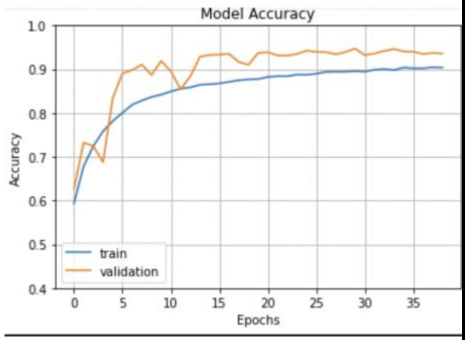
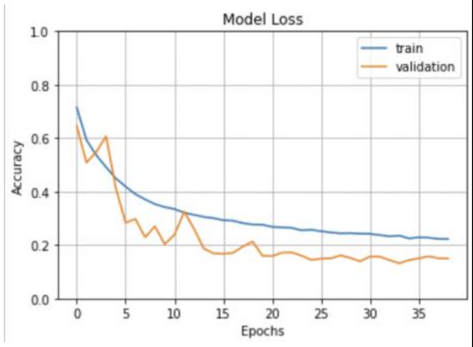
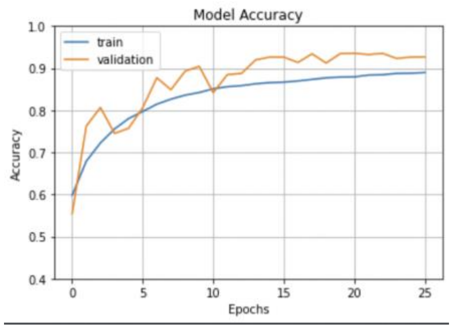
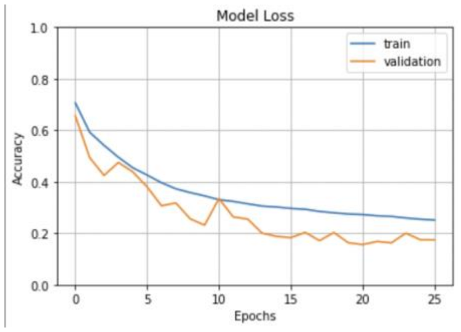
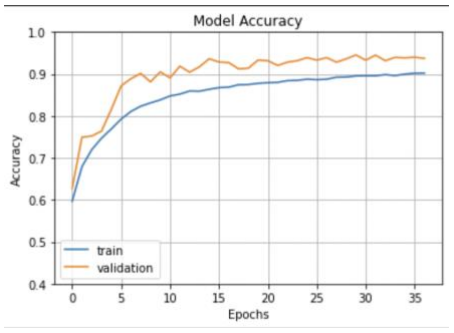
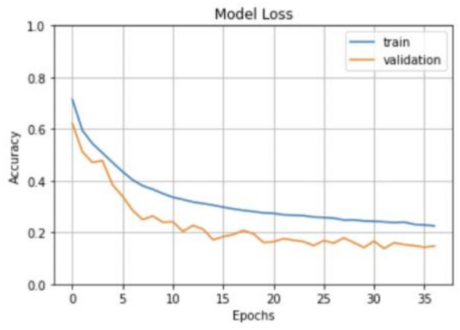
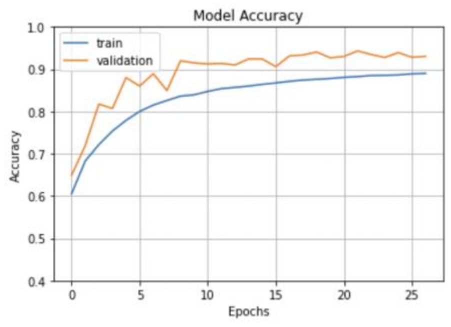
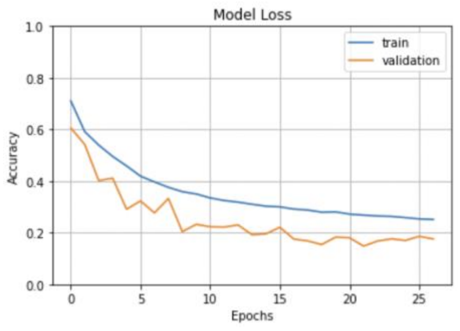
This loss function pairs well with our activation function in the output layer, which is sigmoid. The sigmoid function returns a value between 0 and 1 which we can infer to be how confident the model is of the example being in the class.

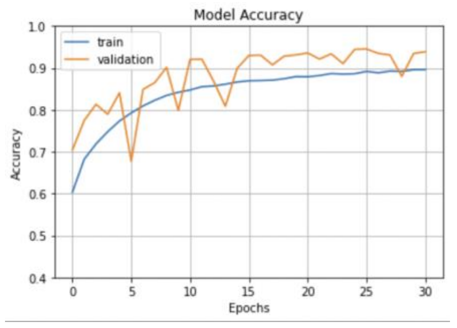
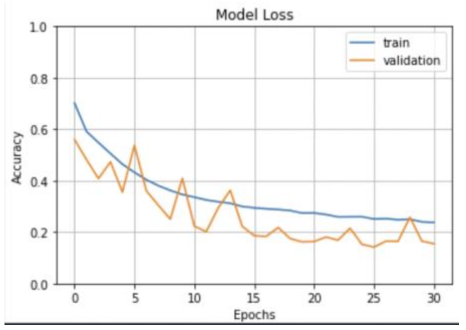
c. Optimizer and Learning Rate:
The optimizer used for our final model is Adam. It is an adaptive optimizer that updates learning rates based on the average first moment (the mean) and the average of the second moments of the gradients (the uncentered variance).
For learning rate, we used 0.001 as our initial learning rate, as we found that it is small enough such that it converges, and large enough, such that it does not take relatively long computation times.

d. Epochs and Early Stopping:
For our training strategy, we used early stopping, such that our model stops training when validation loss no longer decreases. We set the patience (number of epochs it will wait) to be 5. Thus, if the validation loss does not decrease within 5 epochs, the training round will stop.
We set the initial number of epochs to be 50, such that the model can train for longer, if it does not hit the early stopping. Our final model has training rounds ranging from 17 to 34 epochs from our own training.

e. Accuracies and Losses:

Below is a table summarizing our 5 training rounds, ensuring reproducibility, as this is a non-deterministic method.

No	Accuracy Graph	Acc (%)	Loss Graph	Loss
1		93.58		0.1493
2		92.68		0.1740
3		93.72		0.1472
4		93.04		0.1759

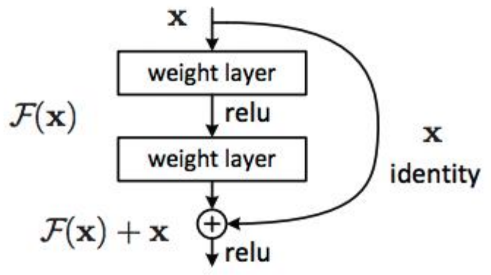
5		93.88		0.1548

Average validation accuracy:	93.38%
Average validation loss:	0.16024

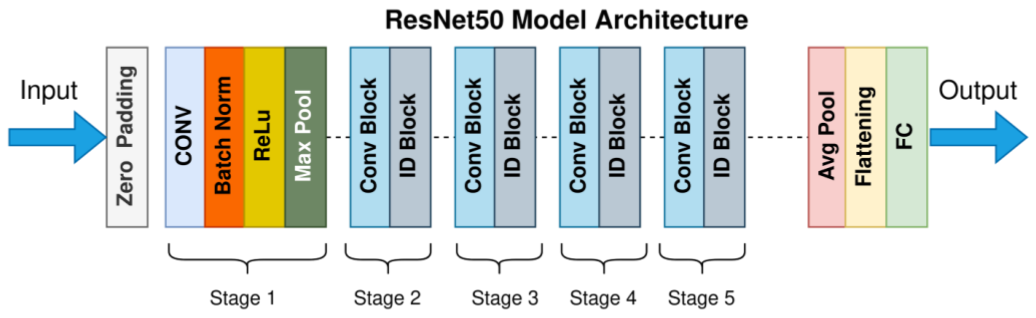
II. ResNet50

We also tried a different model architecture by using transfer learning. Transfer learning is a method where a model developed for a task is reused as the starting point for a model on a second task, with the motivation of increasing accuracy and reducing compute times. The model used was ResNet50 that was pretrained on ImageNet.

The ResNet model itself was created such that deep learning models could go deeper in terms of number of layers, while still having good performance. Residual networks aims to achieve this by developing a residual block. This block allowed inputs to “skip” connections and be added in subsequent layers. Below is an image showing the residual block.


x
weight layer
relu
weight layer
$\mathcal{F}(\mathbf{x})$
x
identity
$\mathcal{F}(\mathbf{x}) + \mathbf{x}$
relu

We used the 50 layer residual network, hence the name ResNet50, keeping all the pretrained weights except for the top layer, which we replaced with an output layer with the sigmoid activation function. Below is the ResNet50 model architecture.

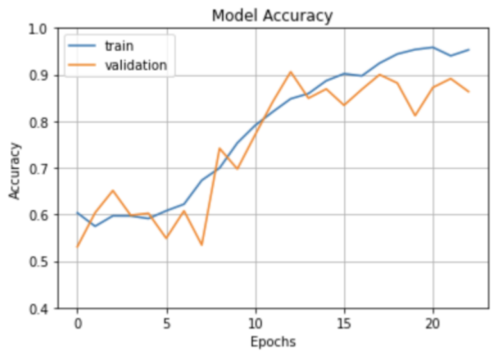


Thus, our model structure was:

1. Rescaling layer: Normalizes input RGB values to be in the range of 0-1. Output shape of (128, 128, 3).
2. ResNet50: ResNet50 pretrained layers, except for the last layer. Output shape of (2048).
3. Fully connected layer: Output layer with sigmoid activation function. Output shape of 1.

For other hyperparameters, we used the same as our CNN.

Total parameters:	23,589,761
Trainable parameters:	23,536,641
Non-trainable parameters:	53,120



As shown, we achieved an accuracy of 88.78% using this model architecture.

Performance and Conclusion:
Our CNN's validation accuracy, 93.38% is higher than the ResNet50 validation accuracy 88.78%. Thus, we will be using our CNN model to generate the test set predictions.

c. ***Discuss how you consider and determine the parameters (e.g., learning rate, etc.) / settings of your model as well as your reasons for doing so.***

<u>No. of epochs:</u>
Our set number of epochs is 50. However, we used early stopping, thus our number of epochs per training round is different, depending on the particular training round itself. We set the patience level for early stopping to be 5 epochs, as initially, when setting it to 10, we noticed that the model would often show overfitting after 5 epochs. Thus, we wanted to reduce training time while having a reasonable early stopping patience.

<u>Image Size:</u>
We had chosen 128x128 to be our image size, as it was a relatively large image size that did not require long computation times to train. We also noted that an image size of 128x128 still has defining features of dogs and cats.

<u>Batch Size:</u>
When choosing batch size, we wanted to use a larger batch size, as it will speed up our training times. However, we do not want to use a batch size that is too large, as too large of a batch size will lead to a poor generalization. Also, we had limited memory hardware to use, thus we opted

to use a batch size of 32. In our CNN model structure, we included batch normalization layers to add regularization and counter overfitting.

<u>Layers:</u>
We used 5 blocks of convolution layers, as by experimentation, it was what yielded the best results. We added dropout and batch normalization layers to reduce overfitting. A fully connected layer is added as the second last layer, such that the model can "select" features. The fully connected layer at the top serves as the output layer, outputting values in the range of 0-1. Predictions are made by rounding this value to the closest integer. 0 corresponds to cat and 1 corresponds to dog.

<u>Optimizer:</u>
We used the Adam optimizer for our models. We chose this as it provides adaptive learning rates, which means that learning rates change as the model is training. Although RMSprop also allows adaptive learning rates, through experimentation, we concluded that Adam was better for our use case.

<u>Learning Rate:</u>
We set the initial learning rate of our Adam optimizer to be 0.001, as it is small enough, such that the model can converge, but still big enough, such that it does not take long computation times. At first we had the learning rate to be 0.0001, however, it was taking too long to converge. Thus, we increased the learning rate by a factor of 10.

<u>Loss Function:</u>
We used the binary cross entropy loss function. This is the loss function we chose as our problem only has 2 classes, thus turning our problem into a binary classification task.

d. ***Report the classification accuracy on validation set. Apply the classifier(s) built to the test set. Submit the "submission.csv" with the results you obtained.***

	Training Accuracy	Validation Accuracy
CNN	90.16%	93.38%
ResNet50	95.31%	88.78%

The classifier we will be using is our CNN. Note that our ResNet50 model overfits the training data. The CSV is named submission_sorted.csv.

e. ***Analyze some correctly and incorrectly classified (if any) samples in the test set. Select 1-2 cases to discuss the strength and weakness of the model.***

To perform this task, we created a function to output a CSV file, consisting of wrong predictions by the model. The columns of the CSV file are actual class, file name/id, and predicted class.

From our observations, we noticed that our CNN model misclassified more dogs than cats. For a training round that had 325 wrong predictions, 125 were wrongly predicted cats, while 200 were wrongly predicted dogs. This is 38.46% and 61.54% respectively.

We also saw that our model was able to differentiate dogs and cats based on their features, e.g. ears, tail, etc. It classified dogs and cats more towards their posture, as most dogs were standing and cats were lying down.
Correct predictions: dog.1791.jpg, cat.3482.jpg
In those two images that were correctly classified, the dog and cat has clearly different postures.

In addition, we noticed that it had the weakness of not being able to correctly classify images where the main features of the animal were covered or were of similar color to the background.
Wrong predictions: dog.216.jpg, cat.13.jpg
In dog.216.jpg, the ears of the dog are not clearly recognizable, and the posture of the dog is upright, thus making the classification harder. In cat.13.jpg, the cat’s color and background color is similar, thus making classification harder.

f. *Discuss how different choice of models and data processing may affect the project in terms of accuracy on validation set*

Data processing:

- If no augmentations were done, it would lead to a smaller training dataset. This could result in a lower accuracy, as the model will be exposed to a limited number of images. It is important to note, however, that data augmentation does not remove the inherent bias of the training data. Thus, to obtain optimal results, one must not keep too many augmented copies of the training data. By augmenting the data, we can get higher accuracy.
- If no rescaling / normalization of RGB input values was done, it could lead to longer convergence.

Model selection:

- Choosing a model will depend on the training dataset size. We always want to minimize the bias and variance of models. If we have a large dataset, and we use a small model, our bias will be high, as our model is not able to learn all the features present in the data. However, if we use a large model on a small dataset, our model will tend to overfit to the training data, thus resulting in a high variance. When choosing or building a model, we must select one that suits our dataset. The approach we used was to first train on a smaller model, then gradually build a bigger one. When we saw overfitting, we added regularization by introducing dropout and batch normalization layers. This combination of layers resulted in a high accuracy.

- Transfer learning using a pretrained model could also yield optimal results. However, in our case, we saw overfitting on the ResNet50 model.
- For transfer learning, one should use a model that was pretrained on a similar dataset. If the selected model is too large, it might overfit very quickly, which will lower the validation accuracy.

g. ***Apply and improve your classification algorithm to a multi-category image classification problem for Cifar-10 dataset (https://www.cs.toronto.edu/~kriz/cifar.html). Describe details about the dataset and classification problem that you are solving. Explain how your algorithm can tackle this problem, and what changes you make compared to solving Dogs vs. Cats problem. Report your results for the testing set of Cifar-10.***

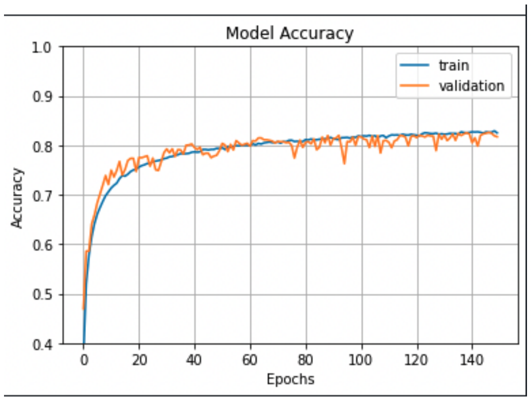
This problem is different from the Dogs vs. Cats problem, as it is not a binary classification problem anymore. The dataset consists of images belonging to 10 distinct classes that do not overlap, namely, airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck. In total, the dataset consists of 60,000 images of 32x32 colored images. Each class is evenly distributed, having 6,000 images. The training set is 50,000 while testing is 10,000. However, the training set is divided into 5 batches, where the class distribution is not even.

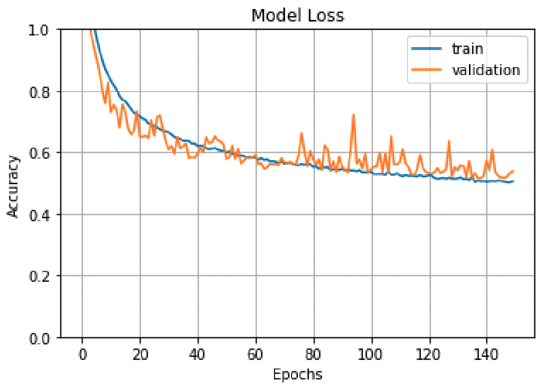
To make our CNN be able to classify these images, we must first change the image size to 32x32. Then, we must change both the output layer and the loss function used for our model. Previously, our model had an output layer of 1 node that returns a value in the range of 0-1. Now, we need an output layer of 10 nodes, which will return a probability for each class. For this, we must change our activation function as well, to be softmax instead of sigmoid. We also changed our loss function to be categorical cross entropy, such that it can perform multiclass classification.

The way the data is imported will be different as well, as with TensorFlow and Keras, we can call tf.keras.datasets.cifar10, which directly imports both the training and testing datasets for cifar10.

Another key difference would be that we need to remove some pooling layers. Since the image size is now 32x32, as opposed to 128x128, we are not able to have as many max pooling layers since they decrease the image shape by the factor of the kernel size, which is 2 in our case. Thus, we removed the first and third max pooling layers from our original model.

We also increased the number of epochs limit to 150, and increased the patience to 20 epochs. This was done as we saw that both the model’s bias and variance was still decreasing. Thus, we wanted to train the model longer. After 150 epochs, we got the final accuracy of the testing set to be 81.72% and testing loss to be 0.5383.





h. ***Train the classifier for (g), while some of the classes in the Cifar-10 training dataset contain much fewer labeled data. How can you improve your algorithm to tackle the data unbalancing issue? Describe and justify at least 2 approaches you use.***

- Data augmentation: we augment minority classes, such that they match the number of images of the majority class.
- Usage of all training data: we can combine the 5 training batches into 1, thus eliminating the issue of class imbalance.

How to Reproduce:

File Structure (inside submission):

```
datasets/
├─ test/
├─ val/
│   ├─ cat/
│   ├─ dog/
├─ train/
│   ├─ cat/
│   ├─ dog/
├─ train_aug/
│   ├─ cat/
│   ├─ dog/
submission_sorted.csv
submission_resnet.ipynb
submission_gradual_cnn.ipynb
submission_cifar10.ipynb
data_augmentation.ipynb
project_report.docx
```

Steps:

1. Data:
- Copy the test, val, and train datasets into the datasets folder.
- Create a new folder train_aug in datasets, and two subfolders, cat and dog.
- Install albumentations, opencv-python, tqdm (if not installed already)
- Run the data_augmentation.ipynb notebook. On the pipeline to augment a folder cell, change the base_dir and dest_dir for cat, after generating for dog.

2. CNN Model Training:
- Run through the submission_gradual_cnn.ipynb until the ‘Saving Output’ cell.
- Install all required dependencies if not already installed.
- Ensure that the number of images matches the number in this report.
- Once the ‘Saving Output’ markdown has been reached, no need to further run the notebook, as the cells below are meant for CSV generation.

3. ResNet50 Training:
- Run through the submission_resnet.ipynb notebook.

4. CIFAR-10 dataset:
- Run through the submission_cifar10.ipynb notebook