

Table of Contents

1. Introduction	2
2. Botnet Classification System	2
2.1 Introduction	2
2.2 Data Preprocessing	2
2.2.1 Dataset Analysis	2
2.2.3 Splitting of Dataset	3
2.3 Deep Neural Network Model	4
2.3.1 Model Building	4
2.3.2 Model Evaluation.....	5
2.4 Model Testing	6
2.5 Improving Accuracy	7
2.5.1 Scaling	7
2.5.2 Dropout Regularization	9
3. Image Classification System	11
3.1 Introduction	11
3.2 Data Preprocessing	11
3.3 Convolutional Neural Network Model	13
3.3.1 Model Building	13
3.3.2 Model Evaluation.....	13
3.4 Model Testing	14
3.5 Improving Accuracy	15
3.5.1 Model VGG.....	15
3.5.2 MobileNet Model.....	17
4. References.....	20
Appendix	20

1. Introduction

A regression and classification are two fundamental types of supervised machine learning. This report will be on the classification type of machine learning. A classification system or model is a type of machine learning model that divides data points into predefined classes (Mural & Kavlakoglu, 2024). Classification can then be applied with the input data or features to give a class. The features can be different type of data set including numerical, images, text or video (Sarvandani, 2023).

In this report, two different classification system will be build and to classify two different types of input data. Firstly, the botnet classification system which uses numerical data and a deep neural network (DNN) model to make predictions. The other is an image classification system which uses images and a convolutional neural network (CNN) model for classifications.

2. Botnet Classification System

2.1 Introduction

For this botnet classification system, it uses the numerical data of IoT devices behaviour and classify botnet attacks. The machine uses the anomaly of the IoT device behaviour to predict the classification. This classification can be used to help detect botnet attacks.

2.2 Data Preprocessing

2.2.1 Dataset Analysis

After importing the data, using `.info()`, the dataset information can be retrieved. The given dataset has 11 columns with 311913 data in each. All the data have an int64 data type which shows that it is numerical. Checking the sum of null values in the dataset is useful in the preparation of the dataset and can be retrieved by using `.isnull().sum()` function.

```
# dataset info
data.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 311913 entries, 0 to 311912
Data columns (total 11 columns):
#   Column                Non-Null Count  Dtype
---  -
0   Flags                  311913 non-null int64
1   Protocol                311913 non-null int64
2   Source Address          311913 non-null int64
3   Source port             311913 non-null int64
4   Direction               311913 non-null int64
5   Destination Address     311913 non-null int64
6   Destination port        311913 non-null int64
7   Total Packets           311913 non-null int64
8   Total Bytes             311913 non-null int64
9   State                   311913 non-null int64
10  Labels                  311913 non-null int64
dtypes: int64(11)
memory usage: 26.2 MB
```

Figure 1: Dataset Information

```
# Check for null values in dataset
data.isnull().sum()
```

	0
Flags	0
Protocol	0
Source Address	0
Source port	0
Direction	0
Destination Address	0
Destination port	0
Total Packets	0
Total Bytes	0
State	0
Labels	0

dtype: int64

Figure 2: checking for null values

2.2.2 Feature Selection

As the dataset has no null values, there is no need to further remove or replace any null values and move on to feature selection. For this model, the y data are the labels and the x data are the other data from the other columns which shows the features on a positive or negative botnet attack. The y data is then reshape from a column vector into an 1D array.

```
X = data.drop(['Labels'], axis=1)
y = data['Labels']
```

Figure 3: Feature selection

```
X = torch.tensor(X.values, dtype=torch.float32)
y = torch.tensor(y, dtype=torch.float32).reshape(-1, 1)
```

```
print("Shape of X: ", X.shape)
print("Shape of y", y.shape)
```

```
⇒ Shape of X: torch.Size([311913, 10])
Shape of y torch.Size([311913, 1])
```

Figure 4: Reshape of y data

2.2.3 Splitting of Dataset

The dataset is then split into train, test and validation data. The dataset was initially split into train and temp. A temporary dataset was created so that there's an equal split for the validation and test data. With train_test_split, the dataset was split into a ratio of train (0.7) : test (0.15) : validation (0.15).

```
# train-temp split the dataset
# temp dataset to be used for test and validation split
X_train, X_temp, y_train, y_temp = train_test_split(X, y, train_size=0.7, shuffle=True)

#test-val split from temp dataset
X_test, X_val, y_test, y_val = train_test_split(X_temp, y_temp, train_size=0.5, shuffle=True)

# The ratio of each split is train(0.7) : test (0.15) : validation (0.15)
```

Figure 5: Splitting the dataset

2.3 Deep Neural Network Model

2.3.1 Model Building

After feature selection and splitting the dataset, the DNN model can be build. The parameters of the DNN is built with the parameters seen in figure 6. As it is deep neural network, the number of inputs and outputs were kept at 10 features for the input and hidden layers. The output uses a sigmoid activation function as we are only looking for a binary classification which is 1 or 0.

```
model = IoTModel()
print(model.parameters)

<bound method Module.parameters of IoTModel(
  (layer1): Linear(in_features=10, out_features=10, bias=True)
  (act1): ReLU()
  (layer2): Linear(in_features=10, out_features=10, bias=True)
  (act2): ReLU()
  (layer3): Linear(in_features=10, out_features=10, bias=True)
  (act3): ReLU()
  (layer4): Linear(in_features=10, out_features=10, bias=True)
  (act4): ReLU()
  (output): Linear(in_features=10, out_features=1, bias=True)
  (sigmoid): Sigmoid()
)>
```

Figure 6: Model Parameters

The model is trained at 50 Epochs and at a batch size of 25. The 25 batch size was selected as it was tested to be the most optimal for both speed and accuracy in the model training. An early stopping function was also create to stop the model training once the validation loss plateau. The best accuracy is also printed to show the model best accuracy during training. It was also trained on three different optimizer to find the best that gives the best accuracy. The three selected are Adam, SGD and AdamW. The AdamW was used as it gives an additional regulation step with the weight decay. The learning rate for the three optimizer were set at their default.

```
# --- Early stopping logic ---
if val_loss < best_val_loss - 0.001: # small tolerance
    best_val_loss = val_loss
    best_weights = copy.deepcopy(model.state_dict())
    no_improve = 0
else:
    no_improve += 1
    if no_improve >= patience:
        print(f"■ Early stopping triggered at epoch {epoch}")
        break
```

Figure 7: Early stopping function

```
def model_train_adam(model, X_train, y_train, X_val, y_val, patience=8):
    # loss function and optimizer
    loss_fn = nn.BCELoss()
    optimizer = optim.Adam(model.parameters(), lr=0.0001)

    n_epochs = 50 # number of epochs to run
    batch_size = 25 # adjust as you like
    batch_start = torch.arange(0, len(X_train), batch_size)
```

Figure 8: Model Train with Adam optimizer

```
def model_train_adamw(model, X_train, y_train, X_val, y_val, patience=8):
    # loss function and optimizer
    loss_fn = nn.BCELoss()
    optimizer = optim.AdamW(model.parameters(), lr=0.001, weight_decay= 0.01)

    n_epochs = 50          # number of epochs to run
    batch_size = 25         # adjust as you like
    batch_start = torch.arange(0, len(X_train), batch_size)
```

Figure 9: Model Train with AdamW optimizer

```
def model_train_sgd(model, X_train, y_train, X_val, y_val, patience=8):
    # loss function and optimizer
    loss_fn = nn.BCELoss()
    optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9, nesterov=True)

    n_epochs = 50          # number of epochs to run
    batch_size = 25         # adjust as you like
    batch_start = torch.arange(0, len(X_train), batch_size)
```

Figure 10: Model Train with SGD optimizer

2.3.2 Model Evaluation

After putting the model to train with the different optimizer, X_train, X_val, y_train and y_val dataset, a comparison on the train loss, accuracy and validation loss can be made. The train loss, validation loss and validation accuracy shows which optimizer is the best. The AdamW optimizer outputs the best results with accuracy being able to learn close to 99% and the loss are close to 0. The other 2 optimizer requires more fine tuning as it gives a flat line for all plots. The training and validation loss remain at a constant of above 50 through all epochs. Whereas, there is also no learning curve for the model with SGD optimizer.

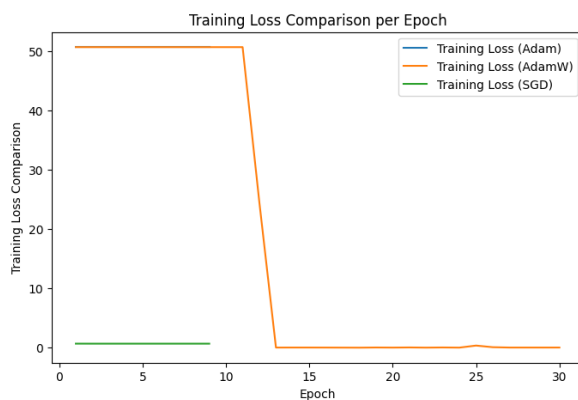


Figure 11: Training Loss comparison

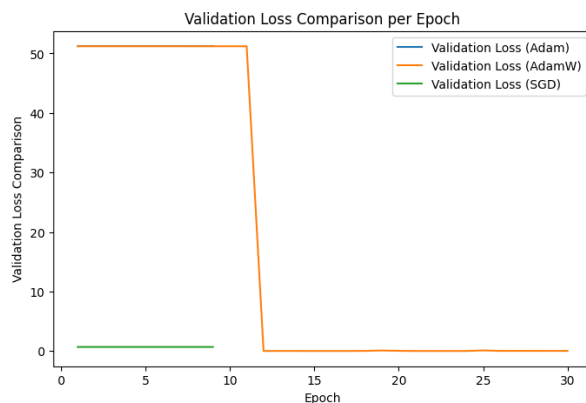
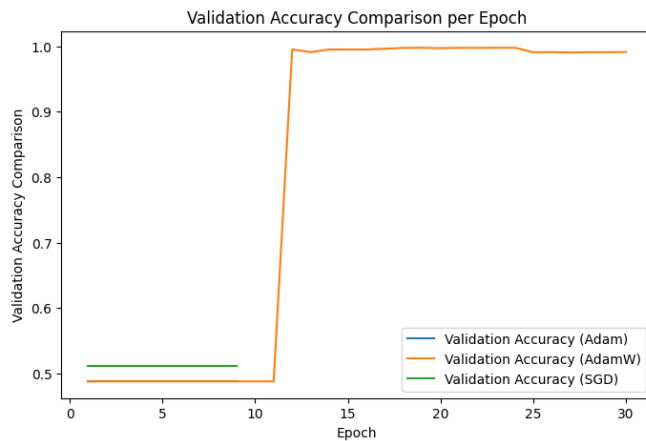


Figure 12: Validation Loss comparison



Best validation accuracy for model with Adam optimizer: 48.79%

Best validation accuracy for model with AdamW optimizer: 99.75%

Best validation accuracy for model with SGD optimizer: 51.21%

Figure 13: Validation Accuracy comparison

2.4 Model Testing

Using the best weights of the AdamW optimizer, the model is then tested with the test dataset. The accuracy return with a 99.72 % accuracy which is considered highly accurate. The model is then tested with 5 random test data and it was able to correctly predict the label with the probability quite reliable. A confusion matrix was also plotted and it shows that only a total of 128 data are labelled incorrectly, which shows that the model is quite accurate in predicting botnet attacks.

```
metrics = evaluate_model(model, X_test, y_test, adamw_best_weights)
print(metrics)

{'test_loss': 0.015622919425368309, 'test_accuracy': 0.9972642064094543}
```

Figure 14: Model metrics on test dataset

```
Sample 1: True = 1.0, Pred = 1, Prob = 0.9865
Sample 2: True = 1.0, Pred = 1, Prob = 1.0000
Sample 3: True = 0.0, Pred = 0, Prob = 0.0000
Sample 4: True = 0.0, Pred = 0, Prob = 0.0000
Sample 5: True = 0.0, Pred = 0, Prob = 0.0000
```

Figure 15: Model Predictions on 5 random sample data

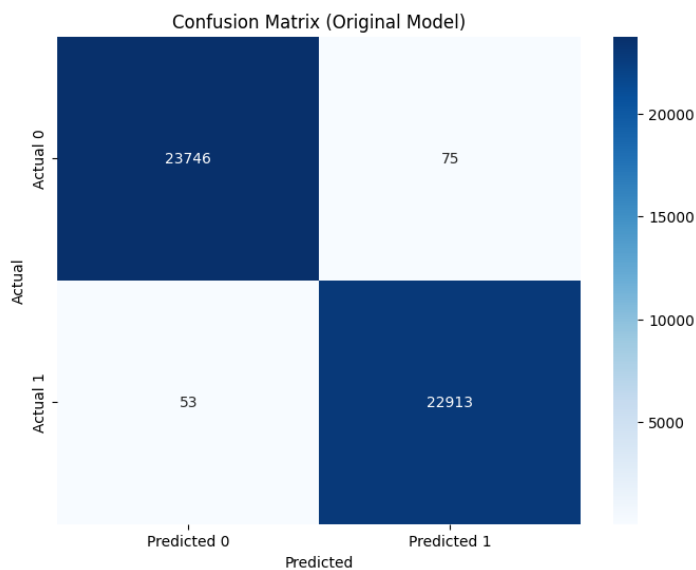


Figure 16: Confusion matrix

2.5 Improving Accuracy

2.5.1 Scaling

Looking at the standard deviation of the original dataset for the total packets and bytes column, the values are quite high which might affect the predictability of the model. Therefore scaling the dataset might help improve the accuracy of the model.

Total Packets	Total Bytes
311913.000000	3.119130e+05
12.275183	6.281414e+03
107.528281	1.131069e+05

Figure 17: Standard Deviation (last row) for total packets and total bytes

The X dataset was scaled with standard scaler and then followed by training with the AdamW optimizer. With the scaled dataset, model gives a best validation accuracy of 99.98% accuracy. Plotting the loss per epoch and accuracy per epoch, it shows that there is a gradual learning curve which makes the model quite reliable. The model is also evaluated with the scaled test dataset and the accuracy also return 99.98%.

```
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_val_scaled = scaler.transform(X_val)

X_train_scaled = torch.tensor(X_train_scaled, dtype=torch.float32)
X_val_scaled = torch.tensor(X_val_scaled, dtype=torch.float32)
```

Figure 18: Scaling the dataset

■ Early stopping triggered at epoch 26

Best validation accuracy: 99.98%

Figure 19: Model best validation accuracy score

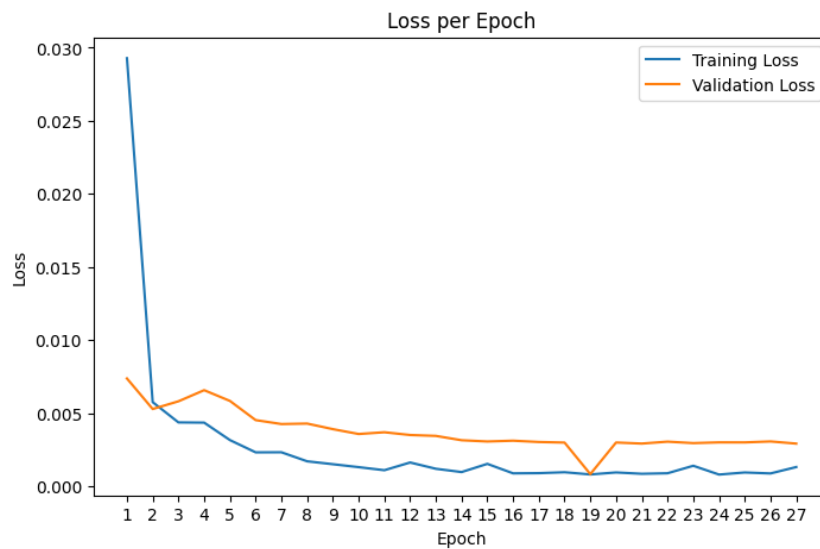


Figure 20: Training and Validation loss curve

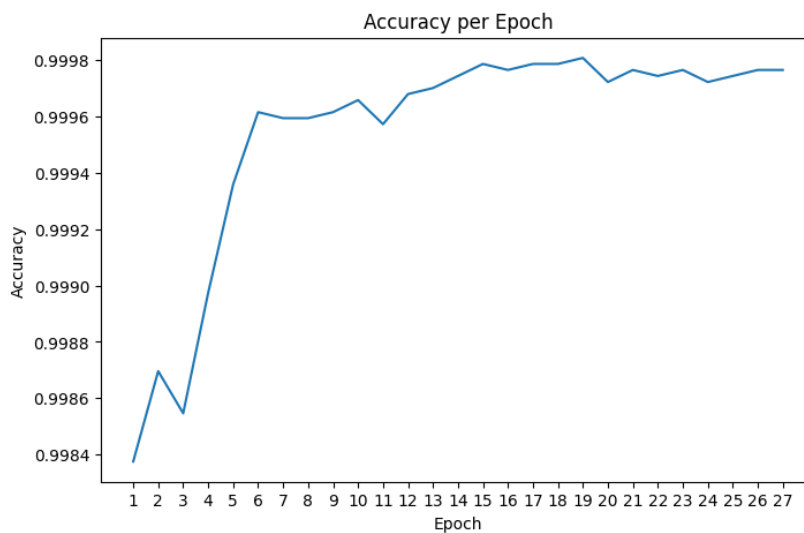


Figure 21: Accuracy curve

```
# scaling on x test

X_test_scaled = scaler.fit_transform(X_test)
X_test_scaled = torch.tensor(X_test_scaled, dtype=torch.float32)

# evaluating with scaled data
model = IoTModel() # Create an instance of the model
metrics = evaluate_model(model, X_test_scaled, y_test, scaled_best_weights)
print(metrics)

{'test_loss': 0.0009359324467368424, 'test_accuracy': 0.9998076558113098}
```

Figure 22: Test accuracy and loss metrics

2.5.2 Dropout Regularization

Another method to improve accuracy was adding dropout regularization to the model. At each layer, a dropout of 0.2 was added. This helps prevent the model to overfit by randomly deactivating a subset of neurons during each training (Marimuthu, 2024). A dropout model was created with similar parameters but with an additional dropout function as seen in figure 23. The model is then train with the scaled dataset and it returns a best validation accuracy of 99.32%.

```
<bound method Module.parameters of IoTModel_Dropout(
  (dropout0): Dropout(p=0.2, inplace=False)
  (layer1): Linear(in_features=10, out_features=10, bias=True)
  (act1): ReLU()
  (dropout1): Dropout(p=0.2, inplace=False)
  (layer2): Linear(in_features=10, out_features=10, bias=True)
  (act2): ReLU()
  (dropout2): Dropout(p=0.2, inplace=False)
  (layer3): Linear(in_features=10, out_features=10, bias=True)
  (act3): ReLU()
  (dropout3): Dropout(p=0.2, inplace=False)
  (layer4): Linear(in_features=10, out_features=10, bias=True)
  (act4): ReLU()
  (dropout4): Dropout(p=0.2, inplace=False)
  (output): Linear(in_features=10, out_features=1, bias=True)
  (sigmoid): Sigmoid()
)>
```

Figure 23: Dropout model parameters

■ Early stopping triggered at epoch 22

Best validation accuracy: 99.32%

Figure 24: Model best validation accuracy score

The learning curve was also plotted and from the loss for train and validation, it shows that the model is learning effectively. It also shows that there were no signs of overfitting which confirms the dropout regularization function. However for the accuracy curve, it is not smooth and it fluctuates a lot between epochs. Although the model still return a high accuracy, there are some fine tuning of the parameters needed such as increasing the batch size.

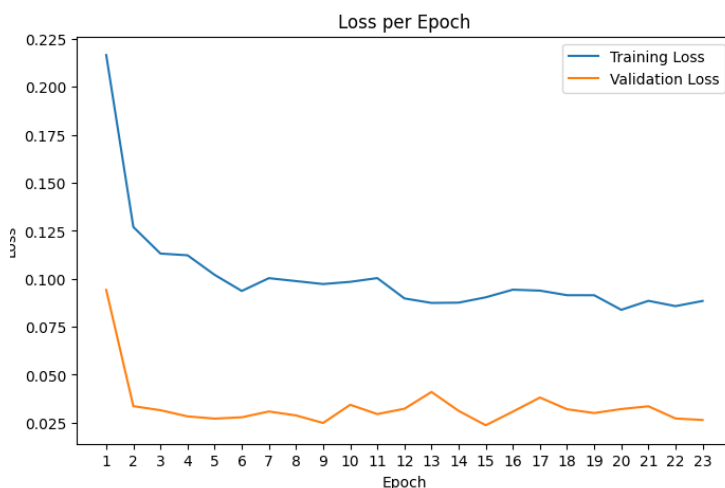


Figure 25: Loss curve for model with dropout

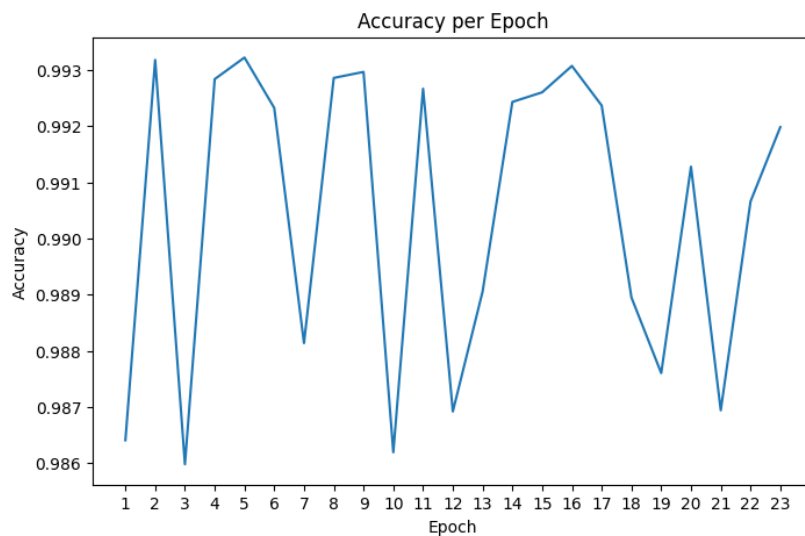


Figure 26: Accuracy curve for model with dropout

The model is then tested on the test dataset and the accuracy returns 99.29% which is quite accurate. Although the model with dropout regularization did not return an improve accuracy, but it return a better learning curve compared to the original model. With some fine tuning on the dropout model, the accuracy would be better than the original.

```
metrics = evaluate_model(model, X_test_scaled, y_test, dropout_best_weights)
print(metrics)
```

```
{'test_loss': 0.022789139300584793, 'test_accuracy': 0.9929254055023193}
```

Figure 27: Dropout Model metrics

Best validation accuracy for original model: 99.75%

Best validation accuracy for model with scaled data: 99.98%

Best validation accuracy for model with dropout: 99.32%

Figure 28: Best validation accuracy comparison among all the model

3. Image Classification System

3.1 Introduction

For this image classification system, it is to classify the images into vehicles and animals. Different images of animals and vehicles were pass through the machine for learning and it was able to return the correct classification of the image.

3.2 Data Preprocessing

Data preprocessing was done after the image data were inputted. Train data is then make by having a function to label the different classes to the image. The two classes that were labelled for the images were animal and vehicles.

```
def assign_class(img, class_type):
    return class_type

def make_train_data(class_type, dir):
    for img in os.listdir(dir):
        print('print DIR:', dir)
        label = assign_class(img, class_type)
        path = os.path.join(dir, img) # Corrected path construction
        print(path)
        img = cv2.imread(path, cv2.IMREAD_COLOR)
        img = cv2.resize(img, (IMG_SIZE, IMG_SIZE))
        X.append(np.array(img))
        Z.append(str(label))

# make train data for animal class
make_train_data('animal', animal_dir)

print DIR: /content/drive/MyDrive/OIDS/Classification/animal
/content/drive/MyDrive/OIDS/Classification/animal/test_image_png_371.png
print DIR: /content/drive/MyDrive/OIDS/Classification/animal
/content/drive/MyDrive/OIDS/Classification/animal/test_image_png_354.png
print DIR: /content/drive/MyDrive/OIDS/Classification/animal
/content/drive/MyDrive/OIDS/Classification/animal/test_image_png_366.png
print DIR: /content/drive/MyDrive/OIDS/Classification/animal
/content/drive/MyDrive/OIDS/Classification/animal/test_image_png_350.png
print DIR: /content/drive/MyDrive/OIDS/Classification/animal
```

Figure 29: Making the train data

```
# confirming animal data
plt.imshow(X[3])

<matplotlib.image.AxesImage at 0x798c332f2a50>
0
20
40
60
80
100
120
140
0 20 40 60 80 100 120 140
```




Figure 30: Confirmation of animal data


```
data_augmentation = keras.Sequential(
    [
        layers.RandomFlip("vertical",
                           input_shape=(150,
                                           150,
                                           3)),
        layers.RandomFlip("horizontal",
                           input_shape=(150,
                                           150,
                                           3)),
        layers.RandomRotation(0.1),
        layers.RandomZoom(0.1),
        layers.RandomContrast(0.1),
    ]
)
```

Figure 34: Data Augmentation

3.3 Convolutional Neural Network Model

3.3.1 Model Building

The CNN model was built with the following parameters and compiled with the Adam optimizer. A global average pooling is used for this model as the dataset is small and would not work well with the flatten function. It computes the average of each feature map to make the necessary classification.

```
# compile the model
model.compile(optimizer='Adam', loss='binary_crossentropy', metrics=['accuracy'])
```

```
# viewing the summary of the model
model.summary()
```

Model: "sequential_1"

Layer (type)	Output Shape	Param #
sequential (Sequential)	(None, 150, 150, 3)	0
conv2d (Conv2D)	(None, 150, 150, 32)	896
conv2d_1 (Conv2D)	(None, 150, 150, 32)	9,248
max_pooling2d (MaxPooling2D)	(None, 75, 75, 32)	0
conv2d_2 (Conv2D)	(None, 75, 75, 64)	18,496
conv2d_3 (Conv2D)	(None, 75, 75, 64)	36,928
max_pooling2d_1 (MaxPooling2D)	(None, 37, 37, 64)	0
conv2d_4 (Conv2D)	(None, 37, 37, 128)	73,856
max_pooling2d_2 (MaxPooling2D)	(None, 18, 18, 128)	0
global_average_pooling2d (GlobalAveragePooling2D)	(None, 128)	0
dense (Dense)	(None, 64)	8,256
dense_1 (Dense)	(None, 1)	65

Total params: 147,745 (577.13 KB)
 Trainable params: 147,745 (577.13 KB)
 Non-trainable params: 0 (0.00 B)

Figure 35: Model Compiler and parameters

3.3.2 Model Evaluation

After the model was trained, a plot can be plotted to show the validation loss, accuracy and training loss across the different epoch. It shows that the machine was able to learn with the increasing accuracy readings and decreasing loss readings. The validation and train loss was less than 0.5 after 30 epochs and the accuracy managed to be above 80% for both.



Figure 36: Model Training and Validation loss

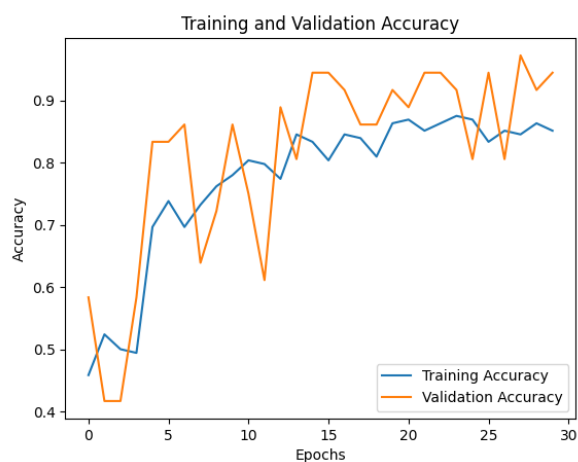


Figure 37: Model Training and Validation accuracy

3.4 Model Testing

After training, the model is then tested with the test dataset. The model gives a 88% accuracy with the test dataset which shows that model to be quite accurate. Giving 5 random images, the model was able to predict the classes accurately. A confusion matrix and F1 score can also be run to show the model's accuracy and performance. The model gives a F1 score of 0.9048 and have a low number in predicting the two classes incorrectly, total 4.

```
2/2 ————— 2s 195ms/step - accuracy: 0.8947 - loss: 0.3473
Test Accuracy: 88.89%
```

Figure 38: Model Test Accuracy

```
# Calculate F1 score for the initial model
f1_initial = f1_score(y_test, y_pred_initial)
print(f"F1 Score for Initial Model: {f1_initial:.4f}")

F1 Score for Initial Model: 0.9048
```

Figure 39: Model F1 score

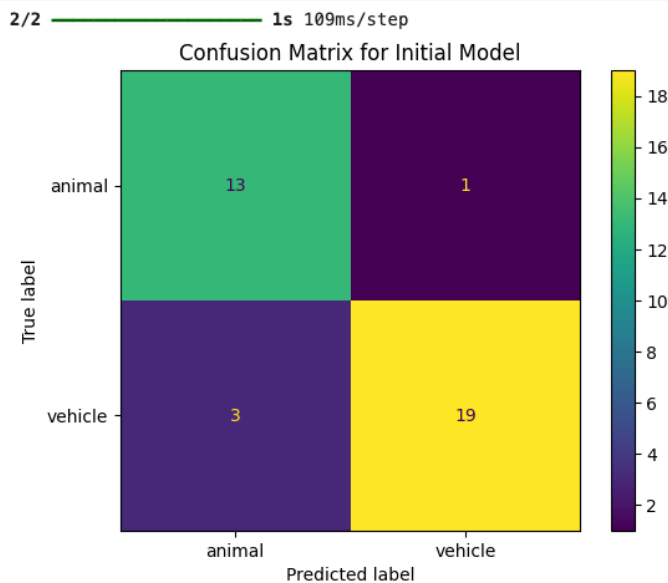


Figure 40: Model confusion matrix

3.5 Improving Accuracy

For this image classification system, I have tested two ways to improve the accuracy of the model. It is to use ready-configured model with specific layers set for training. The two models are Model VGG and MobileNet.

3.5.1 Model VGG

Model VGG is a deep convolutional neural network model that either have 16 or 19 layers. For this study, the 16 layers were used instead. This model is known for its effectiveness in image recognition tasks. The model was built with the preset number of layers by importing the application from Keras. The model is then trained and the loss and accuracy can be plotted to for analysis.

```
from keras.applications import VGG16
conv_base = VGG16(weights='imagenet', include_top=False, input_shape=(150, 150, 3))

Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/vgg16/vgg
58889256/58889256 — 0s 0us/step

for layer in conv_base.layers:
    layer.trainable = False

model_vgg = Sequential([
    layers.Input((150,150,3)),
    conv_base,
    layers.GlobalAveragePooling2D(),
    layers.Dropout(0.4),
    layers.Dense(1, activation='sigmoid')
])

model_vgg.compile(optimizer='Adam', loss='binary_crossentropy', metrics=['accuracy'])
```

Figure 41: Model VGG configuration

After training the model, it can be seen that the loss and accuracy curve for both training and validation have a gradual decrease and increase. The original model have an unstable learning curve which might affect the model's reliability. With these gradual learning curve, the model can considered quite reliable.

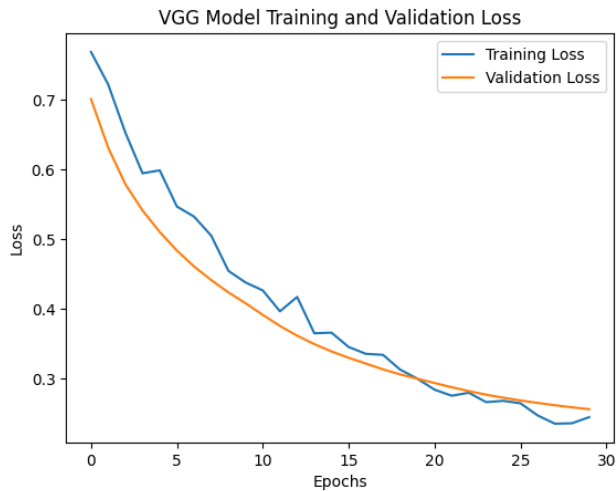


Figure 42: VGG Model Loss curve

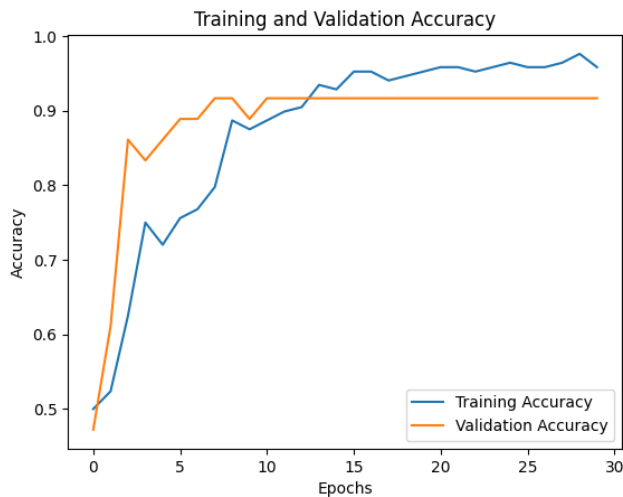


Figure 43: VGG Model Accuracy curve

The model is then tested with the test dataset and was able to accurately classify 4 out of 5 images correctly. Although the model gives a test accuracy of 86%, it can be still considered as an improved accuracy with the improved reliability of the model. With more fine tuning of the model, the accuracy can be improved even more. A confusion was matrix was also plotted showing that only 5 images are mis-predicted as animal when their true classification is vehicle.

```
model_vgg_loss_metric=model_vgg.evaluate(X_test,y_test)
print(f"Test Accuracy: {model_vgg_loss_metric[1]*100:.2f}%")
```

2/2 ————— 8s 780ms/step - accuracy: 0.8657 - loss: 0.3636
Test Accuracy: 86.11%

Figure 44: Model VGG test accuracy



Figure 45: Incorrect classification

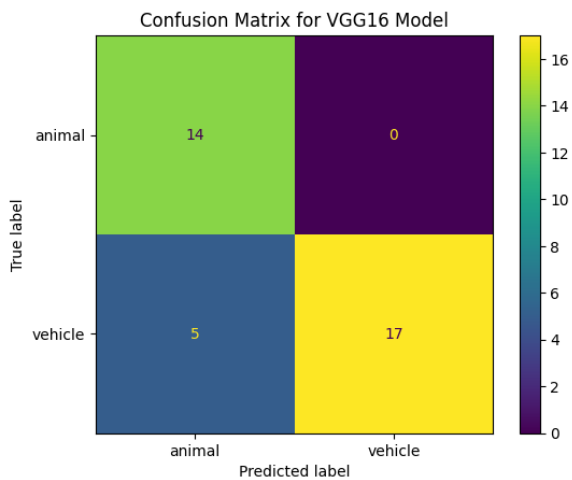


Figure 46: Confusion matrix for VGG16 Model

3.5.2 MobileNet Model

The second model used is the MobileNet model developed by Google. It is considered to be lightweight as it designed to be resource-constrained environment like mobile phones. Although it is lightweight, it is still quite efficient, providing high accuracy and low latency (Klingler, 2024). The model parameters were also downloaded from the Keras application library.

```

from keras.applications.mobilenet import MobileNet

base_layer = MobileNet(weights='imagenet', include_top=False, input_shape=(150, 150, 3))

/tmp/ipython-input-228242933.py:3: UserWarning: `input_shape` is undefined or non-square, or `r
base_layer = MobileNet(weights='imagenet', include_top=False, input_shape=(150, 150, 3))
Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/mobilenet/mob
17225924/17225924 0s 0us/step

for layer in base_layer.layers:
    layer.trainable = False

model_mobilenet = Sequential([
    layers.Input((150,150,3)),
    base_layer,
    layers.GlobalAveragePooling2D(),
    layers.Dropout(0.4),
    layers.Dense(1, activation='sigmoid')
])

model_mobilenet.compile(optimizer='Adam', loss='binary_crossentropy', metrics=['accuracy'])

```

Figure 47: MobileNet configuration

After training the model, the loss and accuracy curve were observed to be slightly gradual as well. There were no epochs that had a drastic change in readings which can be considered quite reliable. The accuracy of this model on the test model had an increase to 94%. A confusion matrix was also plotted and the model only gave 2 incorrect predictions, which makes this model configuration quite accurate. Lastly comparing all the F1 scores among the 3 models, the mobilenet returns the best score which makes mobilenet the best model configuration to use.

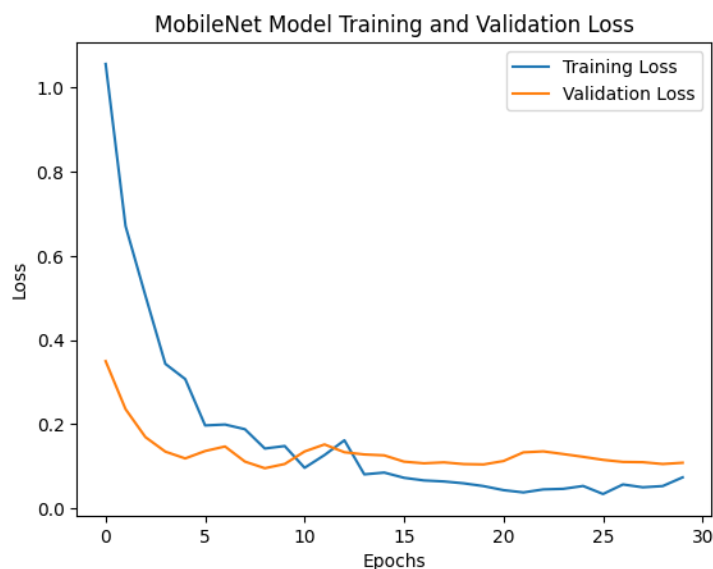


Figure 48: MobileNet model loss curve

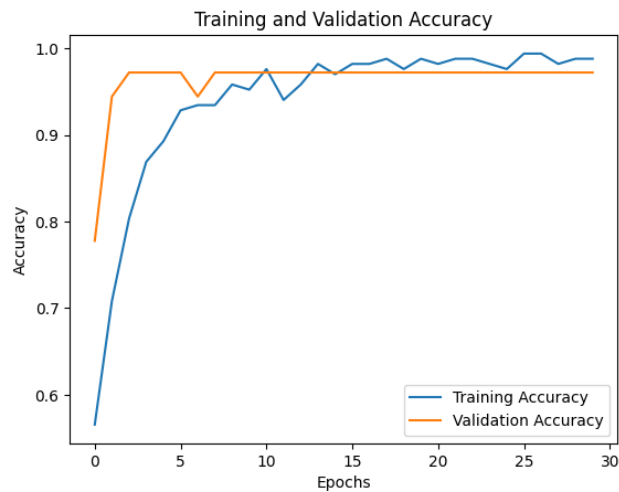


Figure 49: MobileNet Accuracy curve

2/2 ————— 1s 124ms/step - accuracy: 0.9525 - loss: 0.4201
 Test Accuracy: 94.44%

Figure 50: MobileNet test accuracy

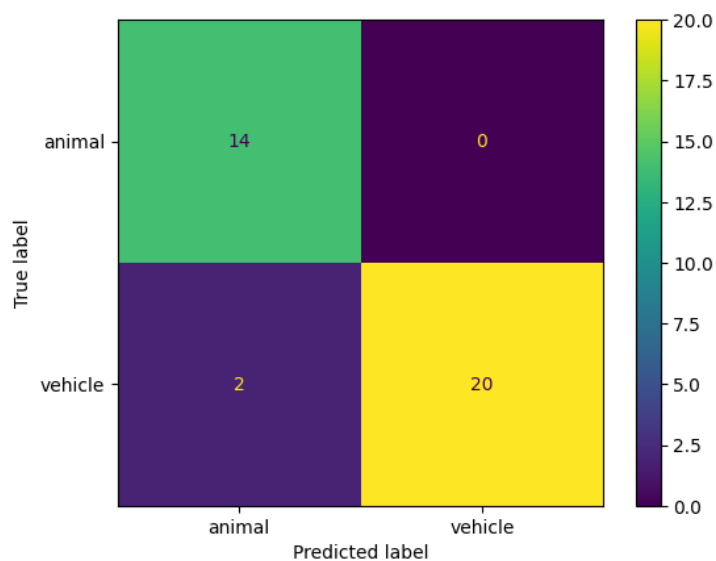


Figure 51: MobileNet confusion matrix

```
print(f"F1 Score for Initial Model: {f1_initial:.4f}")
print(f"F1 Score for VGG16 Model: {f1_vgg:.4f}")
print(f"F1 Score for MobileNet Model: {f1_mobilenet:.4f}")
```

F1 Score for Initial Model: 0.9048
 F1 Score for VGG16 Model: 0.8718
 F1 Score for MobileNet Model: 0.9524

Figure 52: Comparison of F1 score for all 3 models

4. References

Murel, J., & Kavlakoglu, E. (2024, July 31). What are classification models? IBM. <https://www.ibm.com/think/topics/classification-models>

Sarvandani, M. H. (2023, June 9). Top 9 applications of classification in machine learning based on data type. Medium. <https://medium.com/@mohamadhasan.sarvandani/top-applications-of-classification-in-machine-learning-e7b4351f64eb>

Marimuthu, P. (2024, December 12). *Dropout regularization in deep learning*. Analytics

Vidhya. <https://www.analyticsvidhya.com/blog/2022/08/dropout-regularization-in-deep-learning/>

Klinger, N. (2024, May 6). *MobileNet – efficient deep learning for mobile vision*. Viso.ai. <https://viso.ai/deep-learning/mobilenet-efficient-deep-learning-for-mobile-vision/>

Appendix

Colab notebooks link:

https://drive.google.com/drive/folders/1_Yn6pmavMRzsFTWX7MBWOji1SaZ0Gddb?usp=drive_link