

50.021 Artificial Intelligence

Project Report

Jan-Apr 2024

CanOrNot?

Classifying Waste and Detecting recyclables

<i>Member</i>	<i>Student ID</i>
Bryan Sitoh	1005040
Jarron Ng	1005548
Wang Zhao	1005460
Sharryl Seto	1005523

Project Overview.....	3
Problem and Our Pipeline.....	3
Pipeline.....	3
Why did we choose ImageNet?.....	4
Dataset.....	4
Pre-processing.....	4
Models Used.....	5
Simple Convolutional Neural Network (CNN).....	5
Model Architecture.....	5
Loss Curve.....	6
Analysis and Potential Actions (based on graph).....	7
ResNet50.....	8
Pre-processing.....	8
Model architecture.....	8
Loss Curve.....	10
Regularisation Test.....	11
VGG-16.....	12
Pre-processing.....	12
Model architecture.....	12
Loss Curve.....	13
Regularisation Test.....	13
Comparison.....	15
Accuracy.....	15
Multi-class ROC Curve.....	16
Show Prediction.....	17
Ensemble Learning.....	19
Bagging.....	19
Stacking.....	19
Evaluation of Ensemble Learning Predictions.....	19
Accuracy.....	19
F1 Score.....	20
Confusion Matrix.....	20
Model Chosen.....	22
Graphical User Interface (GUI).....	23
Application Development.....	23
Setting up the application.....	23
Starting frontend client.....	23
Starting backend server.....	23
Backend Routing.....	24
User Interface.....	25
Home Page.....	25
Results Page.....	26
Possible Improvements and conclusion.....	29
Bibliography.....	30

Project Overview

In an effort to contribute to environmental sustainability, our project aims to leverage the power of Computer Vision (CV) and Machine Learning (ML) to accurately sort materials into appropriate recycling categories. The initiative focuses on automating the process of identifying and categorising waste materials into six primary classes: metal, plastic, paper, glass, cardboard and trash. By automating waste sorting, we aspire to enhance recycling efficiency and support environmental conservation efforts by leveraging technology to enhance and raise awareness of waste management practices.

Our approach involves using state-of-the-art Computer Vision techniques to train a machine learning model on the provided dataset. The model will learn the characteristics of each category through the analysis of visual features present in the images. Once trained, the model will be capable of receiving an image input and predicting the correct recycling category, thereby streamlining the waste sorting process.

Problem and Our Pipeline

Globally, the average recycling rate is at a low of less than 20%[1]. This is due to

- Lack of public awareness of what can be recycled
- Landscape of recycling
 - It is easier to just throw things away as general waste, resulting in recyclables being contaminated.

Some existing solutions we found focused on streamlining the recycling process and automating material sorting in recycling plants, rather than emphasizing education and awareness about recyclable items[2]. Another study concentrated on recycling tetra packs, with only two categories available that identified the recycling code, CPAP81 and CPAP84[3]. Other projects also included using computer vision to detect recycling codes on waste packaging. However, not many of such packaging will have these printed labels, making these projects ineffective in countries that do not mandate printing recycling codes on waste packaging. Therefore, we want to create an educational recycling aid that is robust in detecting a wider range of materials through machine learning that encourages people to recycle and learn more about recycling practices.

Pipeline

In order to effectively perform and choose an appropriate model to train our dataset on, we made a few considerations and read up on numerous papers, all of which seemed to point that transfer learning would ensure the best results.

First, we had to decide whether it was necessary to perform transfer learning, or whether a simple CNN model would suffice for our dataset and still provide a good accuracy.

Secondly, if the accuracy isn't optimal, even after fine-tuning our parameters, we would then go ahead and try using transfer learning. Transfer learning allows us to fine-tune our model for specific tasks, significantly decreasing training time and increasing accuracy.

For our model, we plan on exploring between ResNet50, and VGG-16, trained on the **ImageNet** dataset. This allows us to use residual connections, enabling the training of deeper networks, and hence improving accuracy. To further improve our final model, we also explored the use of ensemble learning methods such as bagging and stacking.

Lastly, we would go ahead and load the best model onto our Graphical User Interface (GUI) so that whenever a user uploads a photo, we can determine the class the object in the photo belongs in, along with relevant information on how to recycle it. The best model would be determined using the metrics which we will define below in this report as well.

Why did we choose ImageNet?

ImageNet contains over 14 million annotated images.

The dataset is organised according to the WordNet hierarchy. Each meaningful concept in WordNet, potentially described by multiple words or word phrases, is called a "synset". There are more than 20,000 synsets in ImageNet, although not all are used in the challenges.

Dataset

The backbone of our project is a open-source dataset[4] comprising 2527 images, categorised into six distinct classes:

- Glass: 501 images
- Paper: 594 images
- Plastic: 482 images
- Metal: 410 images
- Cardboard: 403 Images
- Trash: 137 images

This diverse dataset serves as the training ground for our models, enabling it to learn and accurately identify various recyclable materials and waste.

Pre-processing

By performing **data augmentation** (for all models), we are able to increase the diversity of the data available without actually having to collect new data.

Training Data Generator

The train_datagen is configured with several data augmentation parameters:

- shear_range, zoom_range, width_shift_range, height_shift_range: These parameters introduce randomness in transformations to create variation in the training data, which helps the model generalise better to unseen data.
- horizontal_flip and vertical_flip: These settings randomly flip images horizontally and vertically, further augmenting the training data.

Models Used

This section explains the components of the pipeline used in the simple CNN model, and provides a simple loss curve to illustrate what the loss is. Afterwards, the same is done for the other models (ResNet50, VGG-16) that have undergone Transfer Learning on ImageNet, using the loss curve as well and highlighting the loss. The code can be found in `AI_Project.ipynb` notebook.

Simple Convolutional Neural Network (CNN)

Model Architecture

The model is built using the Sequential API from Keras, allowing us to stack layers in a linear fashion. The architecture comprises several key components designed to extract features from images and classify them into one of six categories:

- **Convolutional Layers:** The model starts with a series of three convolutional layers (Conv2D), each followed by max pooling (MaxPooling2D). Convolutional layers are the core building blocks of a CNN, responsible for capturing patterns in images such as edges, textures, or more complex patterns in deeper layers. Each convolutional layer in this model uses L2 regularization to combat overfitting by penalizing large weights, with a regularization factor of 0.001.
- **Flattening:** After the convolutional and pooling layers, the Flatten layer transforms the 3D output to a 1D vector, allowing convolutional layers to connect to dense layers.
- **Dense Layer:** A fully connected (Dense) layer follows, also with L2 regularization, serving to interpret the features extracted by the convolutional layers and pooling layers. It has 128 units and uses the ReLU activation function.
- **Dropout:** The Dropout layer is included with a rate of 0.5, which helps prevent overfitting by randomly setting input units to 0 at each update during training time, thus reducing the chance for neurons to co-adapt too much.
- **Output Layer:** The final layer is a dense layer with a softmax activation function, designed to output the probability distribution across the six classes. The number of units matches the number of classes (num_classes).

For the parameter tuning, we focused on adjusting the number of layers, L2 regularisation value and dropout value, using accuracy as a metric to determine the best set of values that would net us the highest accuracy. We also increased the number of epochs to see what values would be the best.

However, as this was done using just a simple CNN model, the accuracy was pretty low as seen in Figure 1, and switching the parameter values gave a value of around 0.42.

Model: "sequential"			
Layer (type)	Output Shape	Param #	
conv2d (Conv2D)	(None, 298, 298, 32)	896	Epoch 1/10
max_pooling2d (MaxPooling2D)	(None, 149, 149, 32)	0	142/142 [=====] - 55s 340ms/step - loss: 2.0978 - accuracy: 0.2053 - val_loss: 1.9108 - val_accuracy: 0.2458
conv2d_1 (Conv2D)	(None, 147, 147, 64)	18496	Epoch 2/10
max_pooling2d_1 (MaxPooling2D)	(None, 73, 73, 64)	0	142/142 [=====] - 47s 334ms/step - loss: 1.8088 - accuracy: 0.2588 - val_loss: 1.8071 - val_accuracy: 0.3250
conv2d_2 (Conv2D)	(None, 71, 71, 128)	73856	Epoch 3/10
max_pooling2d_2 (MaxPooling2D)	(None, 35, 35, 128)	0	142/142 [=====] - 46s 326ms/step - loss: 1.7070 - accuracy: 0.3133 - val_loss: 1.6467 - val_accuracy: 0.3417
flatten (Flatten)	(None, 156800)	0	Epoch 4/10
dense (Dense)	(None, 128)	20078528	142/142 [=====] - 48s 334ms/step - loss: 1.6118 - accuracy: 0.3571 - val_loss: 1.5837 - val_accuracy: 0.3417
dropout (Dropout)	(None, 128)	0	Epoch 5/10
dense_1 (Dense)	(None, 6)	774	142/142 [=====] - 47s 332ms/step - loss: 1.6072 - accuracy: 0.3562 - val_loss: 1.5913 - val_accuracy: 0.3458
Total params: 20164558 (76.92 MB)			Epoch 6/10
Trainable params: 20164558 (76.92 MB)			142/142 [=====] - 47s 334ms/step - loss: 1.5620 - accuracy: 0.3934 - val_loss: 1.5058 - val_accuracy: 0.4208
Non-trainable params: 0 (0.00 Byte)			Epoch 7/10
			142/142 [=====] - 48s 337ms/step - loss: 1.5034 - accuracy: 0.4071 - val_loss: 1.4843 - val_accuracy: 0.3958
			Epoch 8/10
			142/142 [=====] - 48s 340ms/step - loss: 1.4833 - accuracy: 0.4327 - val_loss: 1.5149 - val_accuracy: 0.3750
			Epoch 9/10
			142/142 [=====] - 48s 335ms/step - loss: 1.4792 - accuracy: 0.4252 - val_loss: 1.4518 - val_accuracy: 0.4167
			Epoch 10/10
			142/142 [=====] - 48s 336ms/step - loss: 1.4579 - accuracy: 0.4438 - val_loss: 1.4767 - val_accuracy: 0.4250

Figure 1. Simple CNN Model. Validation Loss & Accuracy

Loss Curve

The model seems to be fitting well since the training and validation losses are close and both decrease over time as seen in Figure 2.

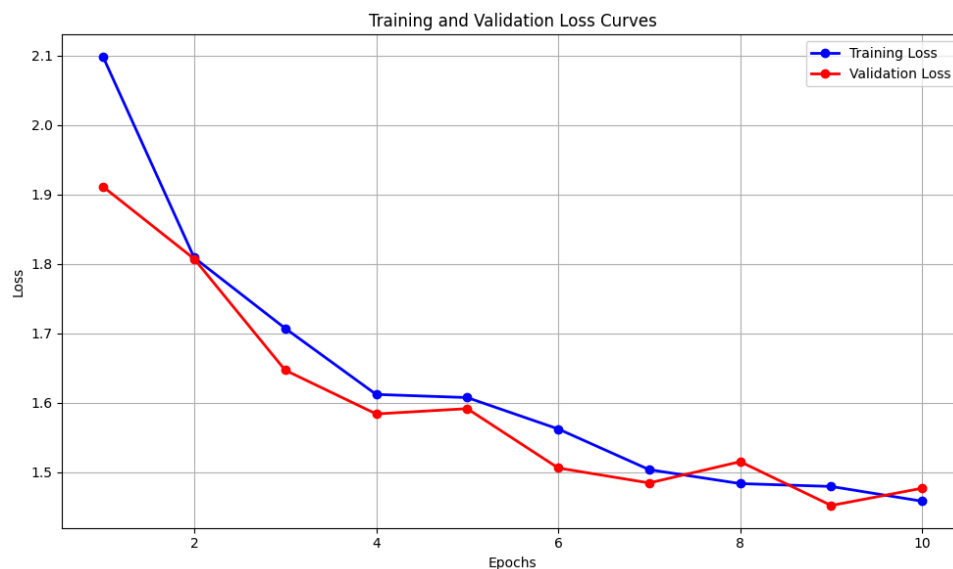


Figure 2. Simple CNN Model Training and Validation Loss Curves

The loss on the training dataset (blue curve) starts significantly higher, suggesting that the model began with a relatively poor understanding of the data. However, as epochs progress, the training loss decreases sharply, indicating that the model is learning and improving its predictions on the training data. After the initial sharp decline, the curve begins to flatten out, showing smaller improvements as the model converges to a solution.

The loss on the validation dataset (red curve) is crucial as it indicates how well the model generalises to new, unseen data. Initially, the validation loss decreases. However, it then shows some volatility, increasing and decreasing throughout the remaining epochs. Ideally, this curve should decrease and then stabilise, just like training loss.

We can infer a lot about the training dynamics and make informed decisions to improve performance.

Analysis and Potential Actions (based on graph)

- **Early Convergence:** Both curves converge early on, which suggests that either the model has quickly learned the primary structure in the dataset or it might be too simple and isn't capable of capturing more complex patterns.
- **Overfitting:** There is no clear sign of overfitting (validation loss does not increase consistently as the epochs progress). Overfitting is typically indicated by a continuous increase in the validation loss, diverging from the training loss.

Despite the tuning of parameters, accuracy only reached a high of 42%. It is possible that the model has not learnt enough to make more accurate predictions yet. There may be:

1. Insufficient training (10 vs 5 vs 100 epochs net the same accuracy)
2. Data augmentation needed (accuracy before and after is similar)
3. Complexity of the data (need to consider new model)

ResNet50

Having observed underwhelming performance with CNN, we initiated exploration into alternative variants. Among the models under consideration, ResNet50 emerges as a promising candidate based on our research. This selection is motivated by the:

1. Deep Architecture

Composed of 50 layers, it can learn a wide range of features at various levels of abstraction. This deep architecture allows it to perform well on complex visual tasks.

2. Residual Connections

Ease of training due to residual connections, or "shortcuts" that allow gradients to flow through the network more effectively, mitigating the vanishing gradient problem that is common in deep networks. So, ResNet50 is easier to train compared to other networks of similar depth

3. Improved accuracy.

By enabling the training of deeper networks, residual connections help improve the model's accuracy on various tasks without the degradation problem, where accuracy saturates or degrades rapidly with the addition of more layers.

With these 3 points, we begin to explore transfer learning with ResNet50 on ImageNet and monitor its performance.

ResNet50 with Transfer Learning on ImageNet

Pre-processing

- `shear_range`, `zoom_range`, `width_shift_range`, `height_shift_range`: These parameters introduce randomness in transformations to create variation in the training data, which helps the model generalise better to unseen data.
- `horizontal_flip` and `vertical_flip`: These settings randomly flip images horizontally and vertically, further augmenting the training data.

This is similar to what was done for Simple CNN, in order to accurately compare the accuracy between both models.

Model architecture

Loading the Base Model

- We load ResNet50 with `weights='imagenet'` to utilise the knowledge it has gained from ImageNet. `include_top=False` excludes the top (or last fully connected) layers of the model, making it adaptable for our custom classification task. The `input_shape` is set to (300, 300, 3), tailored to our dataset's image dimensions.

Freezing the Base Model Layers

- **Layer Freezing:** To retain the learned features, all layers of the base model are set to trainable = False. This prevents the weights from being updated during training, allowing us to utilise the extracted features as they are.

Adding Custom Layers

- **GlobalAveragePooling2D:** Reduces the spatial dimensions of the output from the base model to a vector. This step condenses the feature maps to a single value per map, reducing the total number of parameters and computation in the network.
- **Dense Layers:** A fully connected layer with 1024 neurons follows, introducing the capacity to learn high-level features specific to our dataset. The final dense layer outputs the predictions across num_classes categories using a softmax activation, turning logits into probabilities.

Model Compilation

- **Optimiser:** Adam. Loss function: categorical_crossentropy. Suitable for multi-class classification. The primary metric for evaluation is accuracy.

Training the Model

- **Model Fitting:** The model is trained using train_generator for the input data, with a defined number of steps_per_epoch calculated by dividing the total number of samples by the batch size. The process is repeated for a specified number of epochs (10 in this case), with performance evaluated against a separate validation set provided by validation_generator as seen in Figure 3.

```
# Load ResNet50 base model
base_model = ResNet50(weights='imagenet', include_top=False, input_shape=(300, 300, 3))

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

x = base_model.output
x = GlobalAveragePooling2D()(x)
x = Dense(1024, activation='relu')(x)
predictions = Dense(num_classes, activation='softmax')(x)

model_tl = Model(inputs=base_model.input, outputs=predictions)

model_tl.compile(optimizer='adam',
                 loss='categorical_crossentropy',
                 metrics=['accuracy'])
history_tl = model_tl.fit(
    train_generator,
    steps_per_epoch=train_generator.samples // train_generator.batch_size,
    epochs=10,
    validation_data=validation_generator,
    validation_steps=validation_generator.samples // validation_generator.batch_size
)
```

```
Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/resnet/resnet50_weights_tf_dim_ordering_tf_kernels_
94765736/94765736 [=====] - 0s 0us/step
Epoch 1/10
142/142 [=====] - 58s 363ms/step - loss: 0.7549 - accuracy: 0.7615 - val_loss: 0.4793 - val_accuracy: 0.8250
Epoch 2/10
142/142 [=====] - 51s 357ms/step - loss: 0.3274 - accuracy: 0.8832 - val_loss: 0.2399 - val_accuracy: 0.9042
Epoch 3/10
142/142 [=====] - 51s 357ms/step - loss: 0.2437 - accuracy: 0.9173 - val_loss: 0.4046 - val_accuracy: 0.8417
Epoch 4/10
142/142 [=====] - 51s 361ms/step - loss: 0.2312 - accuracy: 0.9164 - val_loss: 0.2800 - val_accuracy: 0.8667
Epoch 5/10
142/142 [=====] - 50s 353ms/step - loss: 0.1526 - accuracy: 0.9460 - val_loss: 0.2316 - val_accuracy: 0.9000
Epoch 6/10
142/142 [=====] - 51s 360ms/step - loss: 0.1170 - accuracy: 0.9597 - val_loss: 0.4818 - val_accuracy: 0.8458
Epoch 7/10
142/142 [=====] - 51s 361ms/step - loss: 0.1102 - accuracy: 0.9575 - val_loss: 0.1866 - val_accuracy: 0.9375
Epoch 8/10
142/142 [=====] - 51s 358ms/step - loss: 0.1058 - accuracy: 0.9642 - val_loss: 0.4069 - val_accuracy: 0.8792
Epoch 9/10
142/142 [=====] - 50s 354ms/step - loss: 0.1020 - accuracy: 0.9646 - val_loss: 0.1510 - val_accuracy: 0.9250
Epoch 10/10
142/142 [=====] - 51s 356ms/step - loss: 0.0827 - accuracy: 0.9699 - val_loss: 0.3211 - val_accuracy: 0.8958
```

Figure 3. ResNet50 Model. Validation Loss & Accuracy

After training, we performed parameter tuning by adjusting the parameters to find the right optimiser and activation function. Below is the code used to find the best optimiser and activation function for the best accuracy.

```
activation_functions = {
    'relu': nn.ReLU(),
    'sigmoid': nn.Sigmoid(),
    'tanh': nn.Tanh()
}
optimisers = {
    'sgd': optim.SGD,
    'adam': optim.Adam
}
# loop over activation functions and optimisers
for name, activation_func in activation_functions.items():
    for opt_name, opt_class in optimisers.items():
        print(f'Training with activation function: {name} and optimiser: {opt_name}')
        model = FeedForwardNN(input_size=784, num_classes=10, hidden_dims=[512, 256, 128],
                                dropout=0.5, activation_func=activation_func)
        if torch.cuda.is_available():
            model = model.cuda()
        optimiser = opt_class(model.parameters(), lr=0.001)
        train_model(model, train_loader, criterion, optimiser, epochs=5)
```

As a result, we found that using ReLu as an activation function, and Adam as an optimiser, was able to net us the greatest accuracy.

Loss Curve

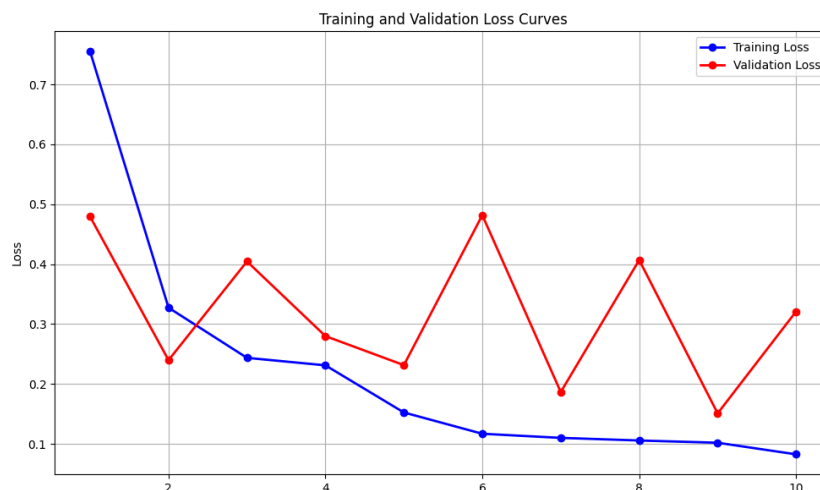


Figure 4. ResNet50 Model Training and Validation Loss Curves

The ResNet50 model is performing well, as seen in Figure 3 and 4. The model's accuracy nets a max of 0.8958. The training loss curve ends with a small loss of about 0.08, while the validation loss curve is about 0.32. The validation loss curve has a large difference with the training loss curve and shows high volatility, increasing and decreasing throughout the remaining epochs. Ideally, this curve should decrease

and then stabilise, just like training loss. The model may also be experiencing more variance in the learning process and potentially overfitting as seen from the difference between the blue and red curves. In that case, we consider using regularisation to counter this overfitting.

Regularisation Test

We performed regularisation with a dropout of a value of 0.5.

```
# Load ResNet50 base model
base_model = ResNet50(weights='imagenet', include_top=False, input_shape=(300, 300, 3))

# Add custom layers on top of the base model with L2 regularization and Dropout
x = base_model.output
x = GlobalAveragePooling2D()(x)
x = Dense(1024, activation='relu', kernel_regularizer=l2(0.01))(x)
x = Dropout(0.5)(x)
predictions = Dense(num_classes, activation='softmax')(x)

model_tl_dropout = Model(inputs=base_model.input, outputs=predictions)

model_tl_dropout.compile(optimizer='adam',
                        loss='categorical_crossentropy',
                        metrics=['accuracy'])

history_tl_dropout = model_tl_dropout.fit(
    train_generator,
    steps_per_epoch=train_generator.samples // train_generator.batch_size,
    epochs=10,
    validation_data=validation_generator,
    validation_steps=validation_generator.samples // validation_generator.batch_size
)
```

Epoch 1/10
142/142 [=====] - 100s 484ms/step - loss: 6.7053 - accuracy: 0.4093 - val_loss: 13.5206 - val_accuracy: 0.3125
Epoch 2/10
142/142 [=====] - 67s 468ms/step - loss: 1.9265 - accuracy: 0.5248 - val_loss: 1.9689 - val_accuracy: 0.3875
Epoch 3/10
142/142 [=====] - 66s 466ms/step - loss: 1.3140 - accuracy: 0.5960 - val_loss: 1.8560 - val_accuracy: 0.4042
Epoch 4/10
142/142 [=====] - 68s 475ms/step - loss: 1.1779 - accuracy: 0.6097 - val_loss: 2.6334 - val_accuracy: 0.3667
Epoch 5/10
142/142 [=====] - 68s 476ms/step - loss: 1.1162 - accuracy: 0.6358 - val_loss: 1.6411 - val_accuracy: 0.4417
Epoch 6/10
142/142 [=====] - 67s 472ms/step - loss: 1.0538 - accuracy: 0.6602 - val_loss: 1.4157 - val_accuracy: 0.5708
Epoch 7/10
142/142 [=====] - 68s 481ms/step - loss: 0.9874 - accuracy: 0.6708 - val_loss: 1.3338 - val_accuracy: 0.5583
Epoch 8/10
142/142 [=====] - 68s 479ms/step - loss: 0.9176 - accuracy: 0.6947 - val_loss: 1.6495 - val_accuracy: 0.4708
Epoch 9/10
142/142 [=====] - 68s 477ms/step - loss: 0.8774 - accuracy: 0.7195 - val_loss: 1.9109 - val_accuracy: 0.4750
Epoch 10/10
142/142 [=====] - 68s 477ms/step - loss: 0.8752 - accuracy: 0.7230 - val_loss: 1.3706 - val_accuracy: 0.6000

Figure 5. ResNet50 Model Training with regularisation and dropout of 0.5

For ResNet50, comparing Figure 3 and 5, the training accuracy has decreased from a high of 0.9699 to 0.723 after regularisation. The validation accuracy has also decreased from a high of 0.8958 to 0.6.

This tells us how regularisation here, in fact lowers accuracy and is bad for the model, especially since there is no overfitting involved.

VGG-16

Despite the excellent performance of ResNet50, we wish to explore some other alternatives to complement our model. The other option that we will be exploring is VGG-16. With 16 layers (13 convolutional, 3 fully-connected with ReLU activation), it is simple, effective and performs strongly on complex visual tasks. The stack of convolutional layers followed by max-pooling layers, with progressively increasing depth, enables the model to learn intricate hierarchical representations of visual features, leading to accurate predictions. As such, we have chosen this class of CNN to be a candidate for experimentation.

However, there are some known limitations based on our research:

1. It is slow to train.
2. 138 million parameters leads to exploding gradients problem.
3. VGG-16 pretrained on ImageNet takes up a significant amount of disk space.

Despite these limitations, we will still perform transfer learning on it and monitor its performance.

VGG-16 with Transfer Learning on ImageNet

Pre-processing

Steps similar to Simple CNN and [ResNet50 pre-processing](#) were performed.

Model architecture

Steps similar to [ResNet50](#) were performed.

```
# Load VGG-16 base model
base_model = VGG16(weights='imagenet', include_top=False, input_shape=(300, 300, 3))

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

x = base_model.output
x = GlobalAveragePooling2D()(x)
x = Dense(1024, activation='relu')(x)
predictions = Dense(num_classes, activation='softmax')(x)

model_tl_vgg16 = Model(inputs=base_model.input, outputs=predictions)

model_tl_vgg16.compile(
    optimizer='adam',
    loss='categorical_crossentropy',
    metrics=['accuracy'],
)

# es = EarlyStopping(monitor='val_accuracy', mode='max', patience=5, restore_best_weights=True)

history_tl_vgg16 = model_tl_vgg16.fit(
    train_generator,
    steps_per_epoch=train_generator.samples // train_generator.batch_size,
    epochs=10,
    validation_data=validation_generator,
    validation_steps=validation_generator.samples // validation_generator.batch_size
)

Epoch 1/10
142/142 [=====] - 66s 398ms/step - loss: 1.3319 - accuracy: 0.4646 - val_loss: 1.0686 - val_accuracy: 0.5458
Epoch 2/10
142/142 [=====] - 52s 367ms/step - loss: 1.0156 - accuracy: 0.6044 - val_loss: 0.8615 - val_accuracy: 0.6667
Epoch 3/10
142/142 [=====] - 53s 372ms/step - loss: 0.8901 - accuracy: 0.6633 - val_loss: 0.7872 - val_accuracy: 0.7083
Epoch 4/10
142/142 [=====] - 58s 410ms/step - loss: 0.8158 - accuracy: 0.6885 - val_loss: 0.7835 - val_accuracy: 0.7083
Epoch 5/10
142/142 [=====] - 53s 371ms/step - loss: 0.7615 - accuracy: 0.7168 - val_loss: 0.7697 - val_accuracy: 0.7208
Epoch 6/10
142/142 [=====] - 53s 374ms/step - loss: 0.7233 - accuracy: 0.7283 - val_loss: 0.7254 - val_accuracy: 0.7292
Epoch 7/10
142/142 [=====] - 53s 370ms/step - loss: 0.7212 - accuracy: 0.7310 - val_loss: 0.6545 - val_accuracy: 0.7708
Epoch 8/10
142/142 [=====] - 52s 368ms/step - loss: 0.6529 - accuracy: 0.7597 - val_loss: 0.6665 - val_accuracy: 0.7792
Epoch 9/10
142/142 [=====] - 53s 372ms/step - loss: 0.6303 - accuracy: 0.7730 - val_loss: 0.7082 - val_accuracy: 0.7333
Epoch 10/10
142/142 [=====] - 52s 365ms/step - loss: 0.6153 - accuracy: 0.7743 - val_loss: 0.6979 - val_accuracy: 0.7625
```

Figure 6. VGG-16 Model, Validation Loss & Accuracy

Loss Curve

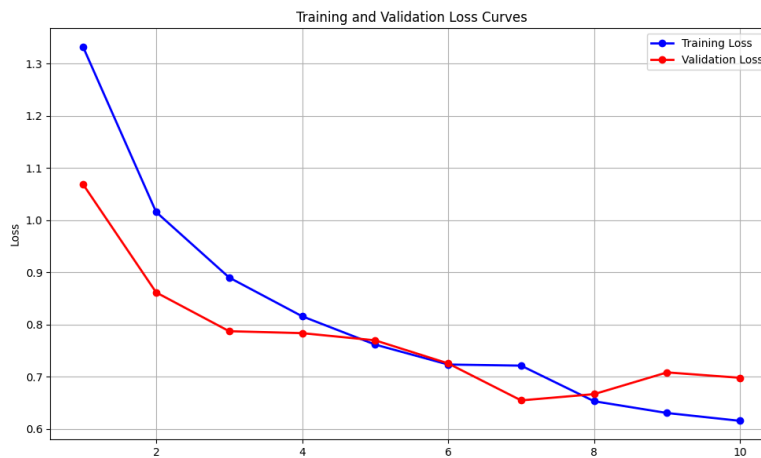


Figure 7. VGG-16 Model Training and Validation Loss Curves

The VGG-16 model performance is fairly good, as seen in Figure 6 and 7. Its validation accuracy is 0.7625, and validation loss is 0.6979. The validation loss follows training loss, and decreases and then stabilises. It shows little sign of overfitting.

Regularisation Test

For insights and analysis, we performed regularisation with a dropout of a value of 0.5.

```
# Load VGG-16 base model
base_model = VGG16(weights='imagenet', include_top=False, input_shape=(300, 300, 3))

# Add custom layers on top of the base model with L2 regularization and Dropout
x = base_model.output
x = GlobalAveragePooling2D()(x)
x = Dense(1024, activation='relu', kernel_regularizer=l2(0.01))(x)
x = Dropout(0.2)(x)
predictions = Dense(num_classes, activation='softmax')(x)

model_tl_dropout = Model(inputs=base_model.input, outputs=predictions)

model_tl_dropout.compile(optimizer='adam',
                        loss='categorical_crossentropy',
                        metrics=['accuracy'])

history_tl_dropout = model_tl_dropout.fit(
    train_generator,
    steps_per_epoch=train_generator.samples // train_generator.batch_size,
    epochs=10,
    validation_data=validation_generator,
    validation_steps=validation_generator.samples // validation_generator.batch_size
)
```

Epoch 1/10
142/142 [=====] - 74s 488ms/step - loss: 4.4403 - accuracy: 0.2332 - val_loss: 2.3677 - val_accuracy: 0.2417
Epoch 2/10
142/142 [=====] - 70s 491ms/step - loss: 1.9880 - accuracy: 0.3199 - val_loss: 1.8671 - val_accuracy: 0.2833
Epoch 3/10
142/142 [=====] - 70s 494ms/step - loss: 1.7013 - accuracy: 0.3677 - val_loss: 1.7579 - val_accuracy: 0.3125
Epoch 4/10
142/142 [=====] - 70s 494ms/step - loss: 1.5849 - accuracy: 0.3934 - val_loss: 1.6725 - val_accuracy: 0.3042
Epoch 5/10
142/142 [=====] - 70s 495ms/step - loss: 1.5649 - accuracy: 0.3912 - val_loss: 1.7837 - val_accuracy: 0.3167
Epoch 6/10
142/142 [=====] - 70s 488ms/step - loss: 1.5271 - accuracy: 0.4221 - val_loss: 1.6694 - val_accuracy: 0.3083
Epoch 7/10
142/142 [=====] - 70s 494ms/step - loss: 1.4866 - accuracy: 0.4257 - val_loss: 1.5319 - val_accuracy: 0.4042
Epoch 8/10
142/142 [=====] - 70s 491ms/step - loss: 1.4361 - accuracy: 0.4482 - val_loss: 1.5169 - val_accuracy: 0.4042
Epoch 9/10
142/142 [=====] - 69s 486ms/step - loss: 1.5356 - accuracy: 0.4190 - val_loss: 1.5675 - val_accuracy: 0.4083
Epoch 10/10
142/142 [=====] - 69s 485ms/step - loss: 1.4124 - accuracy: 0.4739 - val_loss: 1.5218 - val_accuracy: 0.4458

Figure 8. VGG-16 Model Training with regularisation and dropout of 0.5

Comparing Figure 6 and 8, training accuracy has decreased from 0.98 to 0.4739 after regularisation. The validation accuracy also decreased from 0.8667 to 0.4458. Note that there is actually less overfitting as (similar training and validation accuracy). But, this still lowers accuracy and is bad for the model.

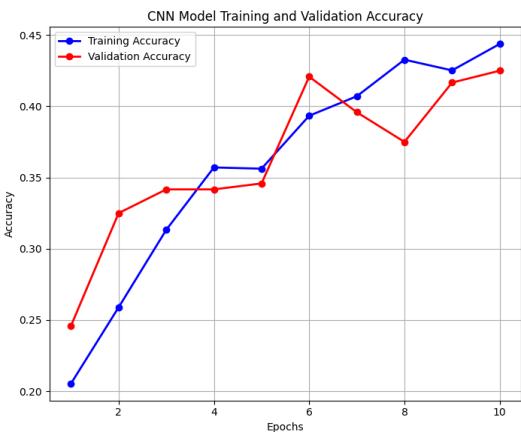

Comparison

This section will provide evaluation metrics such as accuracy and ROC curve of the models to make a simple comparison between them, sufficiently stating the reasons as to why we believe that the results are as shown (highlighting the intricacies of the models which make the results as such).

We then check to see how accurate our models are by comparing the images, and giving its prediction vs ground-truth label.

Accuracy

The difference in accuracy is significant. Simple CNN: 45%, ResNet50: 90%, VGG-16: 85%.

Model & Accuracy Graph	Analysis
<div><p><i>Figure 9. Simple CNN Model Accuracy</i></p></div>	<div><div>1. Improvement Over Time: The training and validation accuracy both increase over epochs, indicating that the model is learning from the data.</div><div>2. Convergence: There is a point, around epoch 9, where both training and validation accuracy stabilise, suggesting that the model has largely learned the features from the dataset and is not improving significantly after this point.</div><div>3. Small Gap: The gap between training and validation accuracy is relatively small, which indicates good generalization. However, the overall accuracy values are not very high (peaking around 45%), which might suggest that the problem is challenging, the model is not complex enough, or the model may need more data or further tuning to improve its performance.</div></div>
<div><p><i>Figure 10. ResNet50 Model Accuracy</i></p></div>	<div><div>1. High Accuracy: The training accuracy is very high, close to 97.5%, which is expected as ResNet50 is a powerful model pre-trained on a large dataset (ImageNet).</div><div>2. Validation Accuracy: The validation accuracy increases and closely follows the training accuracy until around epoch 2 where it fluctuates rapidly while the training accuracy remains stable. This divergence can be a sign of overfitting, where the model performs well on the training data but increasingly poorly on unseen data.</div></div>

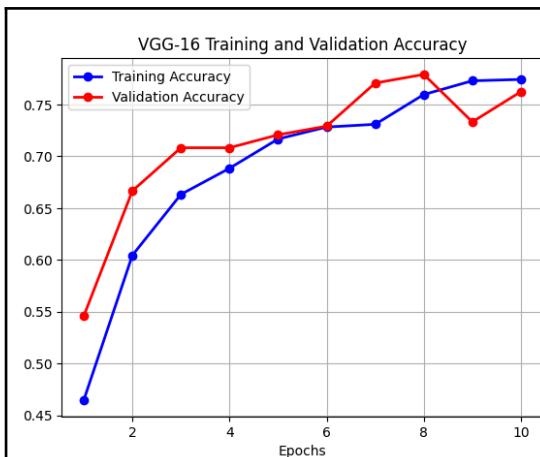


Figure 10. VGG-16 Model Accuracy

1. **Improvement Over Time:** The training and validation accuracy both increase over epochs, indicating that the model is learning from the data.
2. **Convergence:** There is a point, around epoch 5, where both training and validation accuracy stabilise, suggesting that the model has largely learned the features from the dataset and is not improving significantly after this point.
3. **Small Gap:** The gap between training and validation accuracy is relatively small, which indicates good generalization. However, the overall accuracy values are not very high (peaking around 75%), which might suggest that the problem is challenging, the model is not complex enough, or the model may need more data or further tuning to improve its performance.

Overall, the ResNet50 model seems to be the best performer in terms of accuracy. However, the volatility in the validation accuracy for ResNet50 and relatively low accuracy of VGG-16 models suggests there might be room for improvement in terms of the models' configurations, data preprocessing, or augmentation to achieve better stability and generalisation.

Multi-class ROC Curve

Some key functions used in plotting the ROC curve:

1. **Binarization of Labels:** Since ROC analysis is typically used for binary classification, the multi-class labels from the validation generator are binarised using the `label_binarise` function from scikit-learn. This creates a binary matrix representation of the input, suitable for multi-class/multi-label tasks.
2. **Model Predictions:** The `model_tl` and `model` are used to predict class probabilities on the validation data. The `predict` method outputs the probability of each class for each sample, which is necessary to compute ROC curves.
3. **ROC Computation:** For each class, we use the `roc_curve` function from scikit-learn to compute the FPR and TPR, which are stored in dictionaries indexed by class. The `auc` function is then used to calculate the area under the ROC curve (AUC) for each class, providing a single score that summarises the ROC curve's shape.

The curves are plotted on a graph with the FPR on the x-axis and the TPR on the y-axis. The AUC values range from 0 to 1, where 1 indicates perfect classification and 0.5 indicates no discriminative power. In the context of multi-class classification, the micro-average, macro-average, or weighted-average AUCs can also be computed to summarise overall performance.

By plotting ROC curves and calculating AUC scores for each class, we can visually and numerically assess the model's performance across all classes. Each colored line represents the ROC curve for a different class, plotted by false positive rate (FPR) on the x-axis and true positive rate (TPR) on the

y-axis. The dashed black line represents a random classifier's performance; a good classifier's ROC curve will appear above this line.

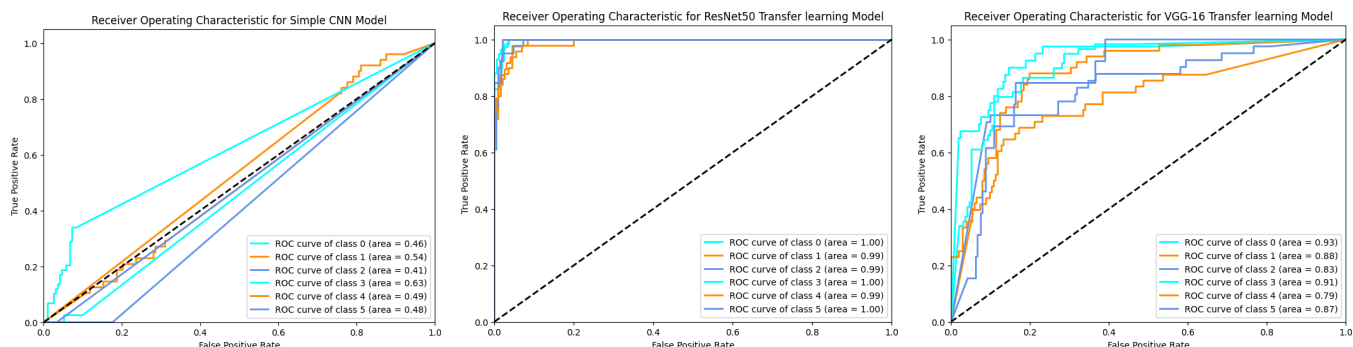


Figure 11. Simple CNN Model vs ResNet50 vs VGG-16 ROC curve

Key observations and interpretations:

1. **Perfect AUC:** For ResNet50, the area under the ROC curve (AUC) for some of the classes is 1.00, which indicates perfect classification with 100% TPR and 0% FPR for those classes. This usually implies that the model can perfectly distinguish between the positive class and negatives for these particular classes.
2. **High AUC for All Classes:** For ResNet50, all the AUC scores are very close to 1.00, suggesting that the model has an excellent performance across all classes.
3. **Class Separation:** It also could be the case that the classes are very well separated in the feature space, and the ResNet50 model can easily distinguish between them, which is more likely in cases with very distinct and clear-cut class differences.

The simple CNN model would likely result in more classification errors as it can be seen from the lower AUC values and curves closer to the diagonal line of no discrimination, compared to the others.

On the other hand, ResNet50 has a nearly perfect AUC value, which suggests that it could be very effective in practical applications, which is likely due to having pre-trained on the ImageNet dataset.

VGG-16 performed well, but not as good as ResNet50, with AUC scores around 0.8.

Overall, it can be said that ResNet50 appears to be vastly superior to Simple CNN and VGG-16 in terms of the ROC curve, and hence more reliable for making predictions.

Show Prediction

Finally, we will determine if the models are actually accurate, by showcasing how the model's predictions compare to the actual labels for a selection of images, prior to any preprocessing steps applied by data generators during training or validation phases.

To do that, the following functions are applied:

1. **Index-Class Mapping:** The function first reverses the `class_indices` dictionary to map numeric indices back to class names. This is necessary for interpreting the model's predictions, which are output as numeric class indices.

2. **Image Selection:** It then compiles a list of all images in the dataset, traversing each class directory and collecting image paths. From this list, it randomly selects num_images images to use for prediction and display.
3. **Image Loading and Preprocessing:** For each selected image, the function loads the image, resizes it to the target size expected by the model (300x300 pixels), and applies the necessary pre-processing. This step is crucial as it aligns the images with the format and scale the model was trained on, ensuring accurate predictions.
4. **Model Prediction:** With the images prepared, the function uses the model to predict the class of each image, then maps the predicted class indices to class names using the index-class mapping prepared earlier.
5. **Visualization:** Each selected image is displayed alongside its predicted and actual class names. This side-by-side comparison provides a clear visual reference to evaluate the model's predictive accuracy on individual images.



Figure 12. Prediction by Models and Actual Labels

As shown in Figure 12, Simple CNN misclassified all of them, while VGG-16 misclassified plastic as glass, trash, and metal, and misclassified cardboard as paper.

Meanwhile, ResNet50 shows its superiority by accurately predicting the material of the object and it matches with the ground truth.

Ensemble Learning

In order to increase the models' accuracy, we decided to use ensemble learning on our 2 models, ResNet50 and VGG-16. We explored 2 types of ensemble learning methods: Bagging and Stacking. The code can be found in `ensemble_learning.ipynb` notebook.

Bagging

Bagging is a technique used to enhance the performance of our transfer learning model based on ResNet50 and VGG16 for image classification tasks. It involves training multiple instances of the base learning algorithm (in this case the ResNet50 and VGG16) on different subsets of the training data and then combining their predictions to obtain a final prediction.

We used the previously trained ResNet50 and VGG16 models to perform bagging via voting in the function `perform_bagging`. It takes input data X (in this case, the validation data generator) and a list of models. It returns the final predictions obtained through voting. The `perform_bagging` function will be called with the validation generator and the list of models (ResNet50 & VGG16) to obtain the final predictions.

Stacking

In stacking, multiple base models are trained independently on the entire training dataset.

Instead of directly combining the predictions of the base models, the predictions are used as features for a meta-model (also known as a blender or a meta-learner). The meta-model is trained on the predictions made by the base models, learning how to best combine their outputs to make the final prediction.

Stacking aims to leverage the strengths of different base models by learning a higher-level function that effectively combines their predictions, potentially leading to better performance than any individual base model.

We concatenated the predictions of both models horizontally along the column axis (axis=1), resulting in a new feature matrix where each row represents a sample, and the columns contain the predictions made by both ResNet50 and VGG16 models for that sample. Then we initialized a random forest classifier with 100 decision trees and set the random state to ensure reproducibility. Finally we trained the random forest classifier on the new feature matrix and the true class labels. The classifier learns to predict the target classes based on the combined predictions of the ResNet50 and VGG16 models.

Evaluation of Ensemble Learning Predictions

We conducted evaluations comparing the accuracy, F1 Score and confusion matrices of the two original models (ResNet50 and VGG16) with those obtained after applying two ensemble learning methods.

Accuracy

As seen in Figure 13, **Stacking** achieved the highest accuracy of **0.932**, outperforming Bagging and the original two models used for its ensemble learning.

```
Accuracy of Bagging (ResNet50 + VGG16): 0.9123505976095617
Accuracy of Stacking (ResNet50 + VGG16): 0.9322709163346613
Accuracy of ResNet50: 0.9243027888446215
Accuracy of VGG16: 0.7689243027888446
```

Figure 13. Accuracy of Bagging, Stacking, ResNet50 and VGG16

F1 Score

F1 score is calculated based on this formula: $F1 = 2 \times \frac{p \times r}{p + r}$. It is the harmonic mean of precision and recall, taking both false positives and false negatives into account, measuring test accuracy and usually used to evaluate the performance of binary classification models.

```
F1 Score of Bagging (ResNet50 + VGG16): 0.9121490668062721
F1 Score of Stacking (ResNet50 + VGG16): 0.9334300378957021
F1 Score of ResNet50: 0.9240417711039396
F1 Score of VGG16: 0.7716666776382977
```

Figure 14. F1 Score of Bagging, Stacking, ResNet50 and VGG16

For **Stacking**, the F1 score is the highest at about **0.933**.

This indicates that the stacking model has a good balance between precision and recall, meaning the model is good in not only identifying the relevant instances (high precision) but also ensuring that it identifies most of the relevant instances (high recall).

This suggests that the model is robust in classifying the positive class in the dataset, correctly labeling the positive instances as positive and avoiding many false negatives and false positives.

Confusion Matrix

The confusion matrix is a crucial diagnostic tool in classification tasks as it provides insight into the types of errors made by the classifier. These are the key components and code.

Prediction and True Labels

- The code starts by using the predict method of the trained model model_tl to obtain predictions for the validation set supplied by validation_generator.
- The predictions are probabilities for each class; np.argmax is applied to convert these probabilities into actual class predictions.
- The true_labels are extracted directly from the validation_generator. They hold actual class labels.

Confusion Matrix Computation

- Using scikit-learn's confusion_matrix function, we compute the confusion matrix cm which compares the predicted classes with the true labels to show the frequency of each type of error.
- This matrix is then converted into a DataFrame called confusion_df for easy visualization and annotation, with class labels from validation_generator.class_indices used as row (Actual) and column (Predicted) headers.

Masking for Diagonal Highlighting

- A mask array mask is created with the same dimensions as the confusion matrix and filled with False. The diagonal is set to True so that we can overlay a different colormap on the diagonal cells to highlight correct predictions.

Heatmap Visualisation

- Two heatmaps are drawn on top of each other using seaborn's heatmap function.
 - The first heatmap (mask=~mask) visualises the non-diagonal elements of the confusion matrix, showing where the model made errors. The second heatmap (mask=mask) overlays the diagonal elements with a green colormap to emphasize correct predictions.
 - The cbar=False argument hides the color bar for a cleaner look.

Plot Customisation

- The plot is titled 'Confusion Matrix' and axes are labeled 'Actual' and 'Predicted' for clarity.
- plt.show() is called to display the plot.

This visualization strategy makes it easier to spot the strengths and weaknesses of the classification model at a glance, providing actionable insights for model improvement.

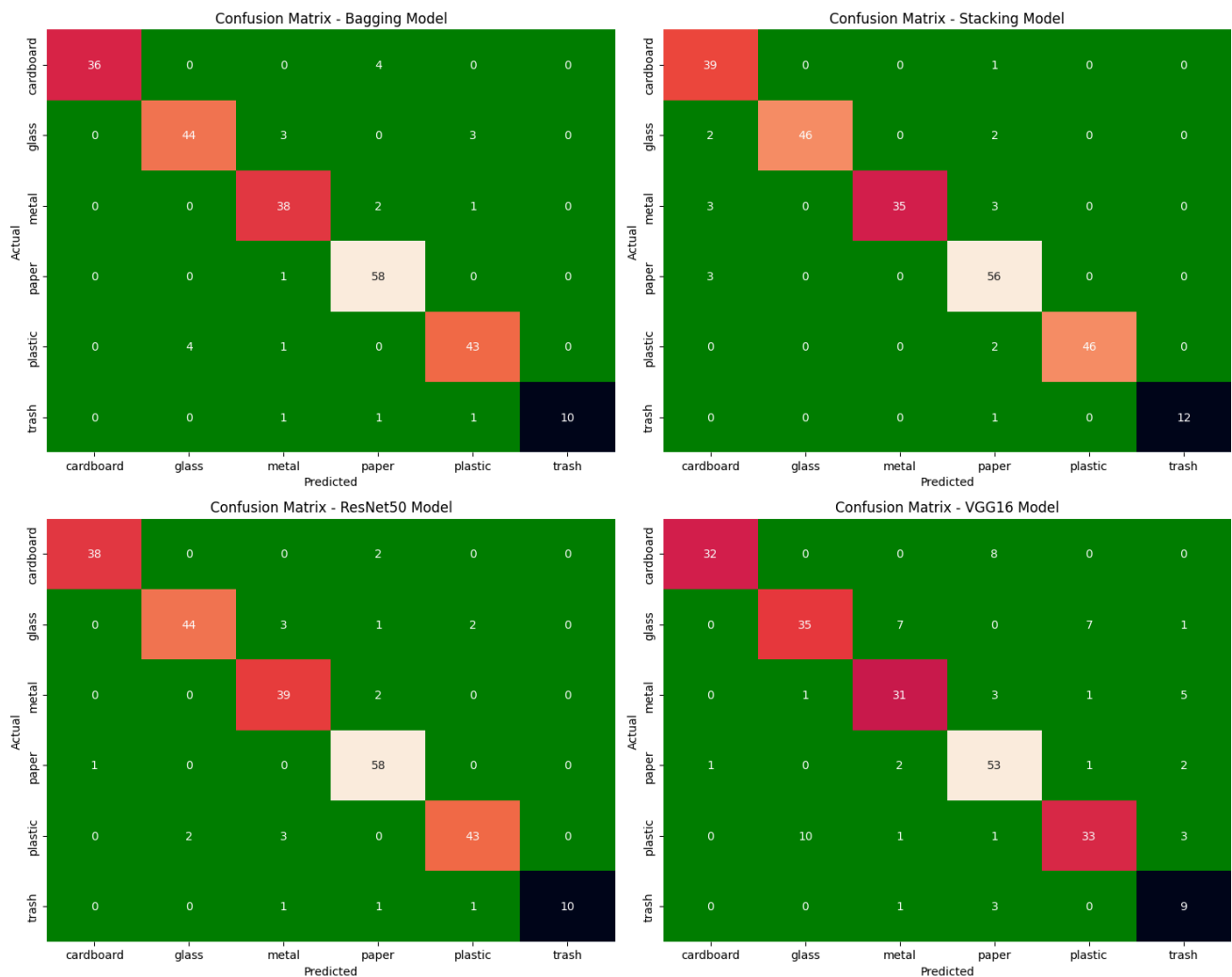


Figure 15. Confusion Matrix of Bagging, Stacking, ResNet50 and VGG16

As shown in figure 15, ResNet50 is very good at classifying papers, with only 1 misclassification. It struggles the most with classifying glass and plastic. It misclassified glass as metal 3 times, paper once, and twice as plastic. It misclassified plastic as glass twice and metal 3 times. Overall, the ResNet50 model seems to perform well.

The VGG-16 model is not bad, but has many misclassifications. It struggles the most with classifying plastic, and misclassifies it as glass 10 times. The model is reasonably good at classifying paper, with very few misclassifications.

The Stacking model demonstrated superior performance in classifying cardboard, glass, plastic, and trash compared to the other three models. However, ResNet50 showed better classification accuracy for metals, while also performing equally well with bagging in the classification of paper.

Model Chosen

Based on the results obtained after ensemble learning, particularly with Stacking, it is evident that ensemble methods can notably enhance classification performance compared to individual models. Stacking, in particular, emerged as the top-performing model, showcasing superior accuracy, F1 score, and class-specific classification accuracy.

Hence, we have decided to leverage the **Stacking** model for our application, considering its remarkable performance in handling classification tasks. This choice is based on its proven capability to provide robust and reliable predictions across various classes, making it well-suited for our intended use case.

Graphical User Interface (GUI)

Application Development

With our model ready, we created a user interface to allow users to upload an image of the waste to be classified. The web application is developed using ReactJs for the front end and FastAPI for the back end server.

- **FastAPI backend:** We chose FastAPI for its high performance and support for asynchronous programming and seamless integration with ML frameworks to accelerate model inference. Moreover, FastAPI streamlines RESTful API development with built-in features like request validation, accelerating the development process.
- **ReactJs frontend:** React's component-based architecture allows for modular and reusable UI components, enabling us to build user interfaces efficiently and quickly. Additionally, React built in capabilities allows for fast rendering and updating of UI components, resulting in responsive web applications.

Setting up the application

The project structure divides the backend and frontend components, evident from the root directory. Backend-related processes are contained within the "backend" folder, whereas the frontend components reside directly in the root directory.

Starting frontend client

First, it is necessary to install all essential packages utilized by the frontend. Additionally, our application requires node version 18. Ensure you are in the project's root directory and execute `npm i` in the terminal. This command will automatically install all dependencies. Following this step, launch the application by running `npm start`. By default, the application runs on port 3000.

Starting backend server

Due to some dependency issues of tensorflow, it is exclusively compatible with Python versions ranging from 3.9.2 to 3.11. Therefore, ensure to verify that your Python version falls within this specified range. Subsequently, navigate to the `/backend` directory and execute `pip install -r requirements.txt` to install the necessary libraries. Create a new folder called `models` in the same directory and download the models' weights inside it. Our models' weights can be found on the huggingface repository, accessible through this link: <https://huggingface.co/Jarron/canOrNot/tree/main>. Once installed, initiate the FastAPI server by running `uvicorn main:app --reload`. The application operates on port 8000 by default.

Backend Routing

This section explains the workings of our application. Specifically, how images are passed from the frontend to the FastAPI server, how the image is processed and subsequently running inference on it.

- ***/predict* route:** This route handles receiving images from the frontend, preprocessing the image, loading the trained model weights and performing inference. The weights of the models are saved locally and are loaded upon calling this route.

As the user uploads an image on the website and proceeds, a `FormData()` object is created that includes the filename, file type and the file itself (the image) as this is what FastAPI expects to receive as inputs for files. This object is forwarded to our backend servers through the route “*/route*” where the image is asynchronously read into bytes while we load the models’ weights.

Once the image has been loaded, we convert the image bytes into an array that will be resized to 300x300, since this is what the model expects as input dimensions. We then further preprocess the image to fit into ResNet50’s expected inputs. Subsequently, we call the `predict()` method from VGG16 and ResNet50 and perform stacking on the results. Finally, we perform one last inference using the meta-model and return the final prediction to the frontend.

User Interface

We made a simple application consisting of only 2 pages: The home page, where the user is able to upload their waste and the results page where the prediction is shown. We have limited the number of uploads to one image for simplicity and only accept files of type .png, .jpeg and .jpg.

Home Page

When the user accesses the application's homepage, they'll encounter a prompt to upload an image to evaluate its recyclability. They can opt to select an image from their gallery or capture a photo.

After choosing their preferred photo, it will be displayed below. Should they decide to replace the image, they can simply click the "cross" icon to remove it. Upon clicking "Can Or Not," the image will be transmitted to our backend servers for preprocessing. The model will then commence inference, and the user will be directed to the results page.

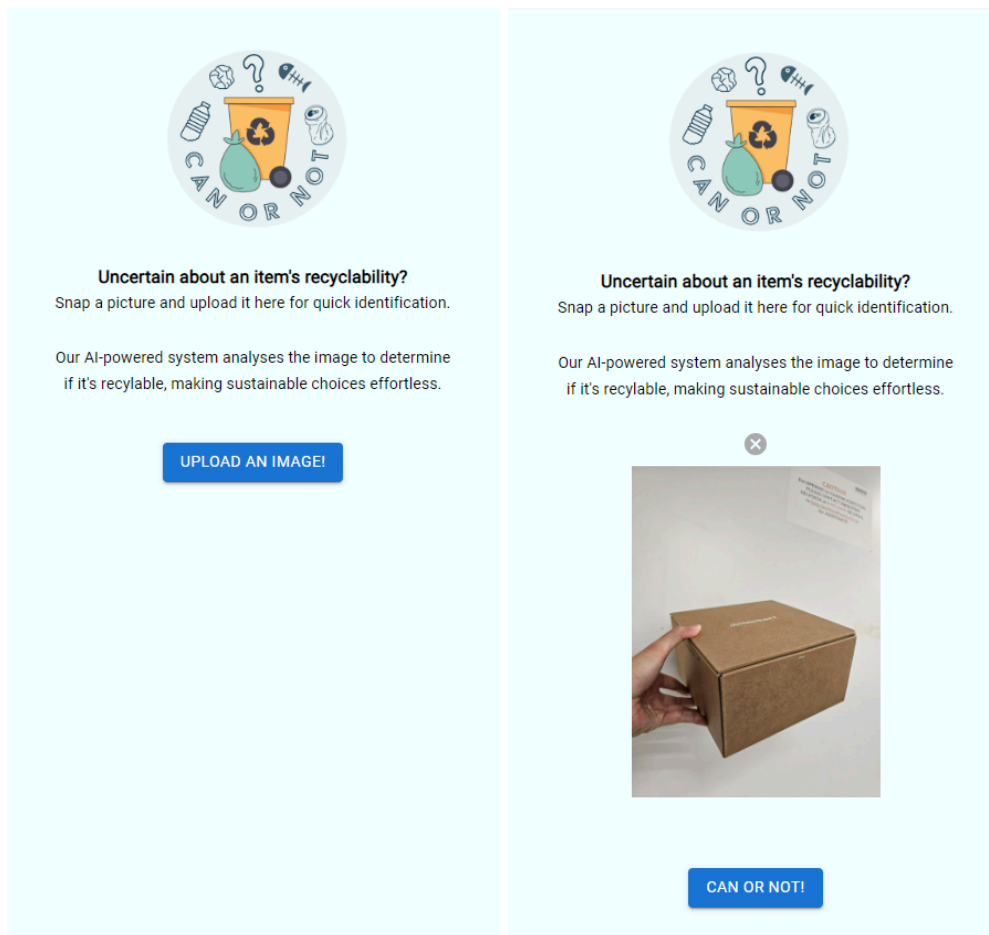


Figure 16. Homepage UI

Results Page

On the results page, there will be a brief wait while the model processes the prediction. Inference typically takes around 6-7 seconds to complete due to CPU utilization instead of GPU.

After inference concludes, the results are returned to the application, displaying the image's classification and its recyclability status. Furthermore, users will find additional information about the material of the item, providing insights and necessary steps to take before placing the item in the recycling bin if it is recyclable. The user may also run inference on another image by clicking on “Check another item”.

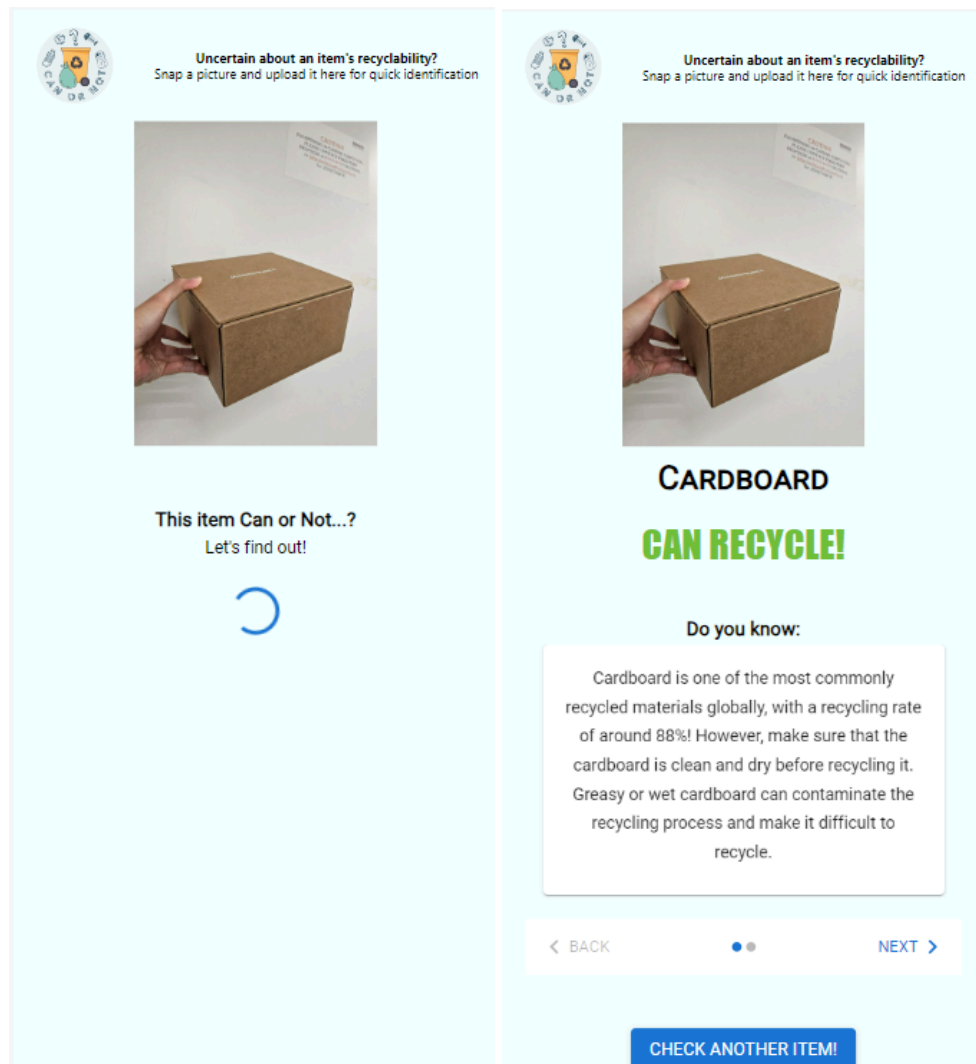


Figure 17. Results Page UI for Cardboard Class

The model is also able to classify metal, glass, paper, trash and plastic.

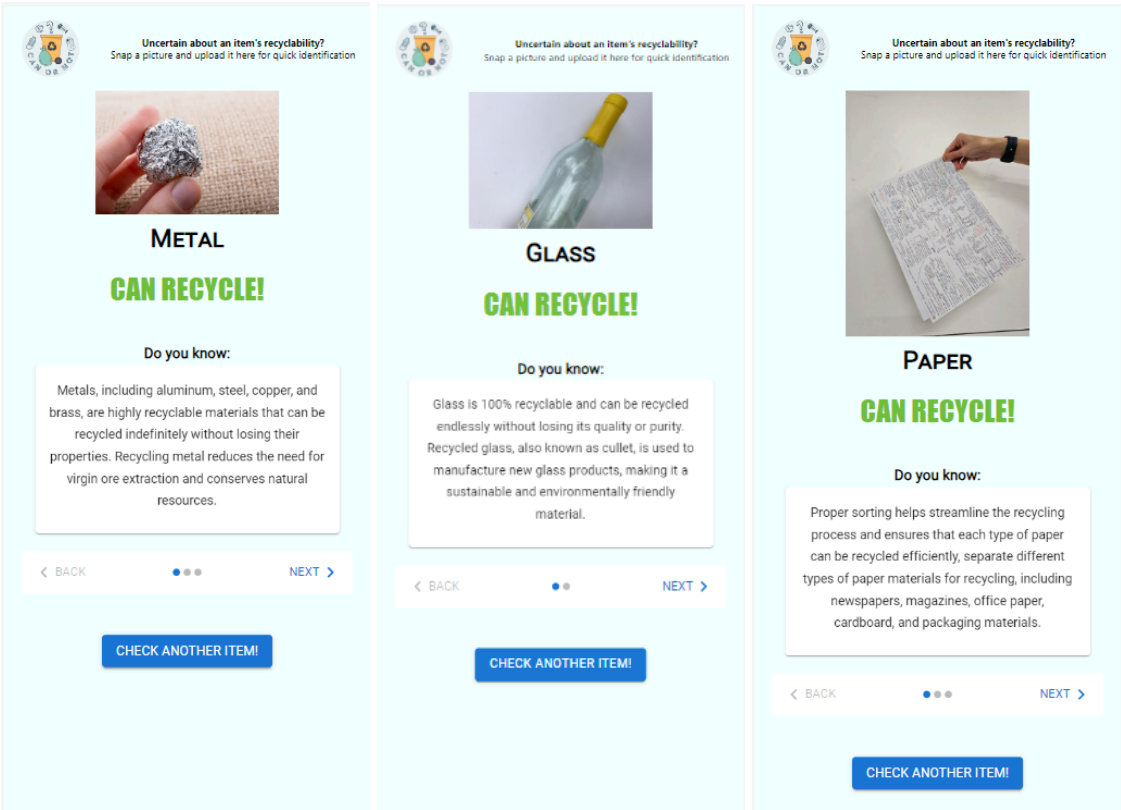


Figure 18. Results Page UI for Metal, Glass, and Paper Class

However, if the model identifies the object as plastic, the user is asked to choose the resin code corresponding to the item, which usually can be found on the item itself. Typically, resin codes 1, 2, 4 and 5 are accepted for recycling globally. So if the user selects either one of these resin codes, it will show “Can Recycle!” on the results tab. If the resin codes are 3, 6, 7 on the other hand, the app will show “Cannot Recycle!” instead.

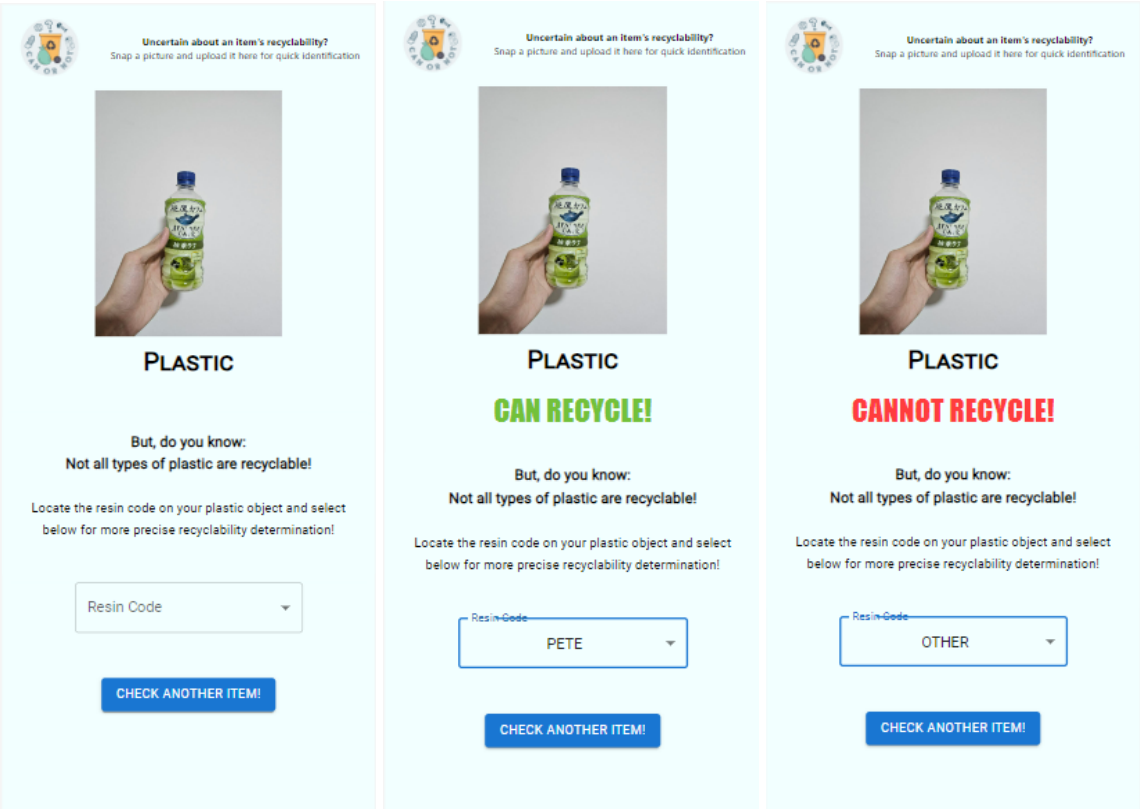


Figure 19. Results Page UI for Plastic Class

Possible Improvements and conclusion

Despite hyperparameter tuning, the accuracy of our model is at a high of only 0.932. There are a few possible reasons:

1. Our dataset is relatively small at about 2,500+ images in total.
 - a. This may cause a poor representation of our labels, as the small data size would not be representative of their respective classes, leading to the model not able to effectively identify unseen labels.
 - b. Training on such a small dataset might lead to overfitting of our data as well, causing our models to not perform accurately.
2. The dataset is imbalanced, with trash being a significant minority class.
 - a. This may cause the accuracy to be misleadingly high due to predicting the majority class correctly.
3. Resizing the image for pre-processing may distort it and cause the resolution to decrease. The background of each image in the dataset is also always a beige background.
 - a. This would result in the model being unable to generalise on images that are not clear. So when our model is tested with an image with a non-neutral background, it may misclassify the object.

Future improvements to overcome this includes:

1. Training our models on a larger dataset with more images for each class.
2. Balance class weights or oversample the trash class to overcome the problems that an imbalanced dataset may cause.
3. Bounding boxes may help the model classify the objects more accurately, as the user may not take perfect images with the object in the centre.

However, due to the interest of time, most of the above were not able to be performed. For instance, we were unable to train with additional data since most of the training of the models had already been done with the initial dataset, and we had no extra time to perform additional training.

Moving forward, we can concentrate on improving our model's performance, broadening the dataset to include more waste materials, and exploring additional techniques to enhance classification accuracy. Our project aims to promote sustainable waste management practices and support the UN SDG goal 12. By educating individuals about recycling and proper waste disposal using our application, we can safely say that we are on track to achieving this goal. Through our continual efforts, we aspire to contribute even more to the preservation of our planet for future generations.

Bibliography

- [1] CCHANG, "Garbage Classification," [www.kaggle.com](https://www.kaggle.com/datasets/asdasdasdas/garbage-classification).
<https://www.kaggle.com/datasets/asdasdasdas/garbage-classification>
- [2] M.I.B. Ahmed, et al, "Deep Learning Approach to Recyclable Products Classification: Towards Sustainable Waste Management,". *Sustainability*, 2023, 15, 11138.
<https://doi.org/10.3390/su151411138>
- [3] S.C. Kamireddi, "Comparison Of Object Detection Models- to detect recycle logos on tetra packs,"
<https://www.diva-portal.org/smash/get/diva2:1679144/FULLTEXT02?ref=blog.roboflow.com>
- [4] D. Filipenco, "World waste: statistics by country and short facts," *DevelopmentAid*, Mar. 07, 2023. <https://www.developmentaid.org/news-stream/post/158158/world-waste-statistics-by-country>
- [5] X. Du, Y. Sun, Y. Song, H. Sun, and L. Yang, "A Comparative Study of Different CNN Models and Transfer Learning Effect for Underwater Object Classification in Side-Scan Sonar Images," *Remote Sensing*, vol. 15, no. 3, p. 593, Jan. 2023, doi: <https://doi.org/10.3390/rs15030593>.
- [6] L. Pal, "Image classification: A comparison of DNN, CNN and Transfer Learning approach," *Medium*, Sep. 13, 2019.
<https://medium.com/analytics-vidhya/image-classification-a-comparison-of-dnn-cnn-and-transfer-learning-approach-704535beca25>
- [7] K. Bhanot, "How Transfer Learning can be a blessing in deep learning models?," *Medium*, Apr. 15, 2021.
<https://towardsdatascience.com/how-transfer-learning-can-be-a-blessing-in-deep-learning-models-fbc576dc42> (accessed Apr. 14, 2024).