Waste Classification for Educational Recycling Web Application

Bryan Sitoh 1005040 | Li Xueqing 1005235 | Reina Peh 1005359 | Nguyen Thai Huy 1005374 | Sharryl Seto 1005523

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 increase recycling efficiency and support environmental conservation efforts by leveraging technology to enhance waste management practices.

Our approach involves using 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 predict the correct recycling category, thereby streamlining the waste sorting process.

I. 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.

Hence, we want to create an educational recycling aid to encourage people to recycle and learn more about how to recycle.

A. 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 [3]–[6], 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.

Thirdly, for our model, we plan on using ResNet50, VGG-16, and DenseNet121, trained on the ***ImageNet*** dataset. This allows us to use residual connections, enabling the training of deeper networks, and hence improving accuracy.

Lastly, we would go ahead and load the 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.

B. 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.

II. Dataset

The backbone of our project is a open-source dataset [1] comprising 2527 images, categorised into six distinct classes:

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

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

Fig. 1. Dataset Image Random Sampling

A. Oversampling Minority Classes

The classes are uneven with trash having the least images out of the rest. We decided to perform oversampling of the trash class to ensure a relatively equal class separation. The result can be seen in "Fig. 2".

B. Data Augmentation

By performing **data augmentation** (for all models), we are able to increase the size and diversity of our training set in order to reduce overfitting during model training.

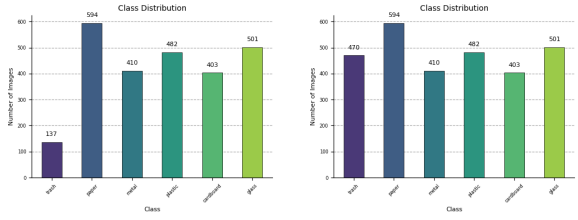

Fig. 2. Class Distribution Before and After Oversampling Trash class

1) 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.
- preprocessing_function: convert the input images from RGB to BGR, then will zero-center each color channel with respect to the ImageNet dataset, without scaling.

The data originally had 2,527 images. the function should have added 341 images to the trash class hence increasing the dataset size to 2,868 images. but the dataset ended up having only 2,860 images. so we used timestamp in naming the augmented images added to trash class but the no. increases to 2863 only.

Possible reasons

1) **File Name Collision**
   The ImageDataGenerator's flow method generates file names based on the save_prefix, the current time, and a hash, making collisions rare but not impossible. If two images are saved in quick succession, they might receive the same filename, leading to one overwriting the other.
2) **Augmentation Filtering by ImageDataGenerator**
   In some cases, if the augmentation parameters result in an image that is too similar to an original image or another augmented image, it's conceivable (though unlikely with default settings) that ImageDataGenerator might skip saving it to prevent redundancy. This behavior, however, is not documented as part of the Keras ImageDataGenerator and would be unusual.

A simple CNN model was created to check and evaluate the pre-processing steps' effectiveness.

```
Epoch 10/10
142/142 [==============================] - 54s 377ms/step - loss: 1.4713 - accuracy: 0.4239 - val_loss: 1.4385 - val_accuracy: 0.4292
```

Fig. 3. Final Loss & Accuracy of Simple CNN model before Dataset Preprocessing

After evaluating with a simple CNN model, we can see from "Fig. 3" and "Fig. 4" that its validation accuracy has improved significantly from 0.4292 to 0.6397, showing that oversampling and data augmentation is beneficial.

```
Epoch 10/10
160/160 [==============================] - 56s 347ms/step - loss: 1.2068 - accuracy: 0.6034 - val_loss: 1.1083 - val_accuracy: 0.6397
```

Fig. 4. Final Loss & Accuracy of Simple CNN model After Oversampling Trash class

III. 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, DenseNet) that have undergone Transfer Learning on ImageNet, using the loss curve as well and highlighting the loss.

The next section evaluates the models with metrics on a deeper level for comparison, and provides a higher level of analysis.

A. Simple Convolutional Neural Network (CNN)

1) 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:

- **Convolution 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, **I2 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 "Fig. 5"

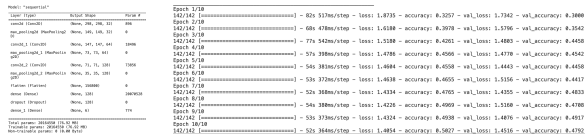

Fig. 5. Simple CNN Model, Training and Validation Loss & Accuracy

Despite the tuning of parameters, accuracy only reached a high of about 48%. 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)

The elaborations on its metrics can be viewed here.

B. ResNet50

Another method we have taken a look at is ResNet50 with Transfer Learning on ImageNet. This is because:

- **Deep Architecture**
  Capability to Learn Complex Features: With 50 layers, ResNet50 can learn a wide range of features at various levels of abstraction. This deep architecture allows it to perform well on complex visual tasks.
- **Residual Connections**
  Ease of Training: Residual connections, or "shortcuts" 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.
- **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.

1) 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.
- **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.
- **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.
- **Optimiser**
  Adam. Loss function: categorical_crossentropy. Suitable for multi-class classification. The primary metric for evaluation is accuracy.
- **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 "Fig. 3".

After training, we performed parameter tuning by adjusting the parameters to find the right 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)
```

Fig. 6. Parameter Tuning for Resnet50

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.

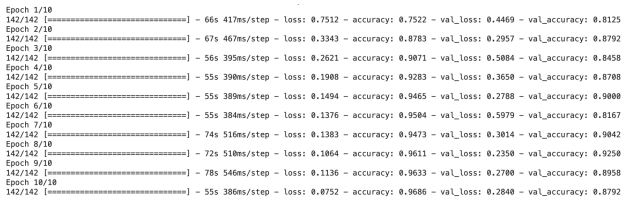

Fig. 7. ResNet50 Model Training and Validation Accuracy

The ResNet50 model performance is good, as seen in "Fig. 7". Its validation accuracy is 0.8792, and validation loss is 0.2840.

2) Regularisation Test: We performed regularisation with a dropout of a value of 0.5.

For ResNet50, comparing "Fig. 5" and "Fig. 8", the training accuracy has decreased from a high of 0.9686 to 0.723 after

```
# 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
```

Fig. 8. ResNet50 Model Training with regularisation and dropout of 0.5

regularisation. The validation accuracy has also decreased from a high of 0.8792 to 0.6.

This tells us how regularisation here, in fact lowers accuracy and is bad for the model. The Elaboration on the metrics can be viewed here.

C. VGG-16

The next model we will look at is VGG-16 with Transfer Learning on ImageNet. With 16 layers (13 convolutional, 3 fully-connected with ReLU activation), VGG-16 is simple and effective, performing 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 robust and accurate predictions.

That said, the model is not without limitations.

1) The model is slow to train due to the large number of parameters.
2) The training of 138 million parameters also leads to exploding gradients problem.
3) VGG-16 pretrained on ImageNet takes up a significant amount of disk space, posing an obstacle on training of other models.

1) Model architecture: Steps similar to ResNet50 have been completed.

```
Epoch 1/10
142/142 [==============================] - 85s 547ms/step - loss: 0.9225 - accuracy: 0.7372 - val_loss: 0.6477 - val_accuracy: 0.7875
Epoch 2/10
142/142 [==============================] - 77s 541ms/step - loss: 0.4434 - accuracy: 0.8465 - val_loss: 0.4899 - val_accuracy: 0.8292
Epoch 3/10
142/142 [==============================] - 81s 570ms/step - loss: 0.2769 - accuracy: 0.9066 - val_loss: 0.5970 - val_accuracy: 0.8250
Epoch 4/10
142/142 [==============================] - 84s 587ms/step - loss: 0.2534 - accuracy: 0.9142 - val_loss: 0.3885 - val_accuracy: 0.8625
Epoch 5/10
142/142 [==============================] - 57s 398ms/step - loss: 0.1890 - accuracy: 0.9345 - val_loss: 0.3688 - val_accuracy: 0.8833
Epoch 6/10
142/142 [==============================] - 60s 421ms/step - loss: 0.1497 - accuracy: 0.9509 - val_loss: 0.5109 - val_accuracy: 0.8250
Epoch 7/10
142/142 [==============================] - 58s 407ms/step - loss: 0.1307 - accuracy: 0.9562 - val_loss: 0.2886 - val_accuracy: 0.9042
Epoch 8/10
142/142 [==============================] - 57s 401ms/step - loss: 0.1094 - accuracy: 0.9642 - val_loss: 0.5035 - val_accuracy: 0.8417
Epoch 9/10
142/142 [==============================] - 60s 421ms/step - loss: 0.0957 - accuracy: 0.9664 - val_loss: 0.4682 - val_accuracy: 0.8750
Epoch 10/10
142/142 [==============================] - 57s 398ms/step - loss: 0.0828 - accuracy: 0.9721 - val_loss: 0.2863 - val_accuracy: 0.9250
```

Fig. 9. VGG-16 Model Validation Accuracy

As seen in "Fig. 9", the model shows strong performance: with a validation accuracy of 0.925, and validation loss of 0.2863, outperforming ResNet50.

Elaboration on the metrics can be viewed here.

D. DenseNet121

1) Model Architecture: Steps similar to ResNet50 have been completed.

Summary of Hyperparameter Tuning

To experiment with hyperparameter tuning and other techniques, we selected DenseNet121 to make our trials and errors more computationally efficient. Please view the appendix for the full details on the series of trials with DenseNet121 hyperparameter tuning.

In total, we completed 12 trials. Trial 11 seems to be the most ideal as the training and validation loss lines converge within 20 epochs, as opposed to other trials where the lines start to diverge within 20 epochs or even 10 epochs. After class balancing, instead of trash class oversampling (oversampling 5 classes to the biggest class size = 594) + Regularization with x = Dropout(0.5)(x) + Class weights + Adam(learning_rate=0.0001), we tried to correct the oversampling process to only train set, instead of all 3 sets, and changed number of units (neurons) in a dense layer from 1042 to 121.

```
Epoch 1/10
142/142 [==============================] - 64s 399ms/step - loss: 3.0290 - accuracy: 0.4779 - val_loss: 1.5059 - val_accuracy: 0.5958
Epoch 2/10
142/142 [==============================] - 55s 388ms/step - loss: 1.5265 - accuracy: 0.6173 - val_loss: 1.2862 - val_accuracy: 0.6833
Epoch 3/10
142/142 [==============================] - 54s 381ms/step - loss: 1.3760 - accuracy: 0.6434 - val_loss: 1.3132 - val_accuracy: 0.6417
Epoch 4/10
142/142 [==============================] - 54s 383ms/step - loss: 1.2741 - accuracy: 0.6619 - val_loss: 1.2432 - val_accuracy: 0.6542
Epoch 5/10
142/142 [==============================] - 54s 382ms/step - loss: 1.2201 - accuracy: 0.6712 - val_loss: 1.2796 - val_accuracy: 0.6542
Epoch 6/10
142/142 [==============================] - 55s 389ms/step - loss: 1.1399 - accuracy: 0.6912 - val_loss: 1.1631 - val_accuracy: 0.6458
Epoch 7/10
142/142 [==============================] - 56s 393ms/step - loss: 1.1217 - accuracy: 0.6965 - val_loss: 1.1618 - val_accuracy: 0.7125
Epoch 8/10
142/142 [==============================] - 54s 377ms/step - loss: 1.1100 - accuracy: 0.6876 - val_loss: 1.0724 - val_accuracy: 0.6708
Epoch 9/10
142/142 [==============================] - 55s 387ms/step - loss: 1.0548 - accuracy: 0.7084 - val_loss: 1.0836 - val_accuracy: 0.6750
Epoch 10/10
142/142 [==============================] - 54s 382ms/step - loss: 1.0367 - accuracy: 0.7159 - val_loss: 1.0379 - val_accuracy: 0.7167
```

Fig. 10. DenseNet121 Model Validation Accuracy (K=5, last fold)

2) With K-fold: DenseNet121's performance is great, as seen in Figure 9. Its validation accuracy is 0.7167, and validation loss is 1.0379, but not as good as ResNet50 or VGG-16.

Elaborations on its metrics can be viewed here.

IV. Evaluation

We use evaluation metrics namely confusion matrix, ROC, accuracy and F1 score of the models, and make a comparison between them, sufficiently stating the reasons as to why we believe that the results are as shown. Finally, we check to see how accurate our models are by comparing the predicted class on a random test set image's actual class.

A. Training and Validation Loss Curves

"Fig. 11" show the validation and loss curves of each model's performance.

With the difference in the training and validation loss curves for ResNet50 and VGG-16, we can tell that there is a slight overfitting of the training data.

Overall, the ResNet50 model seems to be the best performer due to the low training and validation loss of about 0.3, with VGG-16 as a close second. However, the slope of its validation loss is much higher than that for training loss with some volatility, which suggests there might be room for improvement in terms of the models' configurations, data

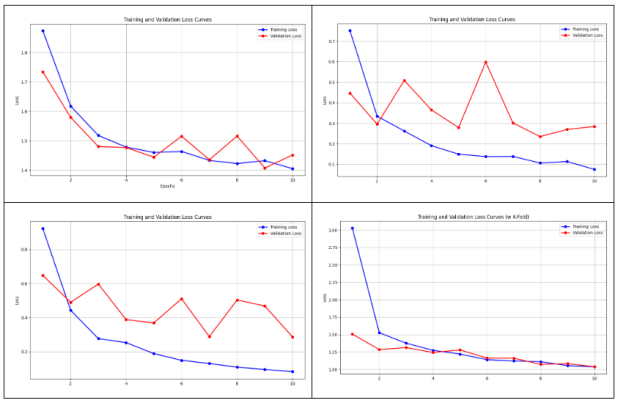

Fig. 11. Comparison of Loss Curves across 4 models

preprocessing, or augmentation to achieve better stability and generalisation.

B. F1-Score & Accuracy

$$F1 = 2 * \frac{precision * recall}{precision + recall} \tag{1}$$

F1 score is calculated based on this formula, taking both false positives and false negatives into account, measuring test accuracy and is used to evaluate the performance of binary classification models.

Our models all have a high F1 score. This indicates how great the balance between precision and recall is. If the model has a high f1 score, it 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 our dataset, correctly labeling the positive instances as positive and avoiding many false negatives and false positives.

"Fig. 12" below show the overall classification report of each model's performance on the test set.

	precision	recall	f1-score	support
cardboard	0.93	0.97	0.95	40
glass	1.00	0.70	0.82	50
metal	0.88	0.88	0.88	41
paper	0.93	0.95	0.94	59
plastic	0.71	0.98	0.82	48
trash	1.00	0.54	0.70	13
accuracy			0.88	251
macro avg	0.91	0.84	0.85	251
weighted avg	0.90	0.88	0.87	251

16/16 [==============================] - 1s 62ms/step
Loss: 570.0427
Accuracy: 0.2191

	precision	recall	f1-score	support
cardboard	0.93	0.97	0.95	40
glass	1.00	0.70	0.82	50
metal	0.88	0.88	0.88	41
paper	0.93	0.95	0.94	59
plastic	0.71	0.98	0.82	48
trash	1.00	0.54	0.70	13
accuracy			0.88	251
macro avg	0.91	0.84	0.85	251
weighted avg	0.90	0.88	0.87	251

16/16 [==============================] - 2s 104ms/step
Loss: 0.2904
Accuracy: 0.8765

	precision	recall	f1-score	support
cardboard	0.97	0.97	0.97	40
glass	0.96	0.90	0.93	50
metal	0.86	0.90	0.88	41
paper	0.95	0.93	0.94	59
plastic	0.85	0.94	0.89	48
trash	1.00	0.77	0.87	13
accuracy			0.92	251
macro avg	0.93	0.90	0.91	251
weighted avg	0.92	0.92	0.92	251

16/16 [==============================] - 2s 132ms/step
Loss: 0.3034
Accuracy: 0.9203

	precision	recall	f1-score	support
cardboard	0.88	0.53	0.66	40
glass	0.73	0.72	0.73	50
metal	0.70	0.76	0.73	41
paper	0.71	0.83	0.77	59
plastic	0.60	0.69	0.64	48
trash	0.80	0.62	0.70	13
accuracy			0.71	251
macro avg	0.74	0.69	0.70	251
weighted avg	0.72	0.71	0.71	251

16/16 [==============================] - 2s 114ms/step
Loss: 1.0560
Accuracy: 0.7092

Fig. 12. Comparison of F1 Scores & Accuracies across 4 models

The VGG-16 model aces this evaluation with the highest F1 Score of 0.92, and the highest accuracy of 0.92, followed by ResNet50 and DenseNet121.

C. 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 we used.

- **Prediction and True Labels**
  - 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.
- **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.
- **Heatmap Visualisation**
  - A heatmap is drawn using seaborn's heatmap function, emphasising class predictions easily.

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. "Fig. 13" show the confusion matrix of each model on the test set.

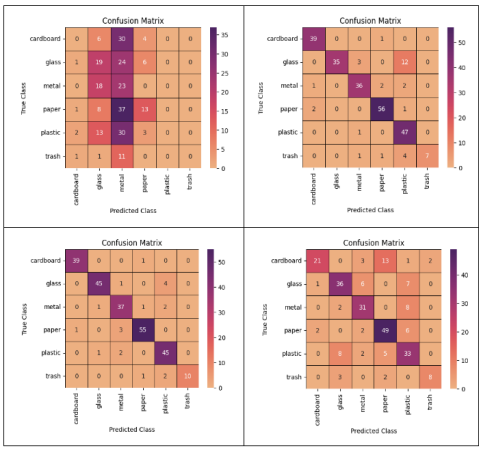

Fig. 13. Comparison of Heatmaps across 4 models

The simple CNN model appears to have significant issues and misclassified most objects into the glass, metal and paper classes.

For ResNet50, the model performed very well but tends to misclassify glass as plastic, as shown by the 12 predictions in the heatmap column.

The VGG-16 model was outstanding, and faced a similar problem as ResNet50 regarding glass, but did significantly better, with only 4 wrong predictions.

DensNet121's model did well, but often misclassified cardboard as paper. It also performed best with paper and plastic, although there's room for improvement in these categories as well.

D. Multi-Class ROC Curve

Main key functions used in this ROC (receiver operating characteristic) 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.

Each class's ROC curve is plotted in a different color, making it easy to distinguish between them. The curves are plotted on a graph with the FPR on the x-axis and the TPR on the y-axis. A dashed diagonal line represents a no-skill classifier that randomly guesses the class; a good classifier's ROC curve will appear above this line.

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.

We can see that the results here show that ResNet50 is the best model out of all the 4 here, as it achieved 3 classes with AUC values = 1.00. VGG-16 has also fared comparably well, with DenseNet121 following close after.

E. Show Prediction

We determine if the models are actually accurate by showcasing the models' predictions compared to the actual labels for a selection of images without any preprocessing steps applied by data generators.

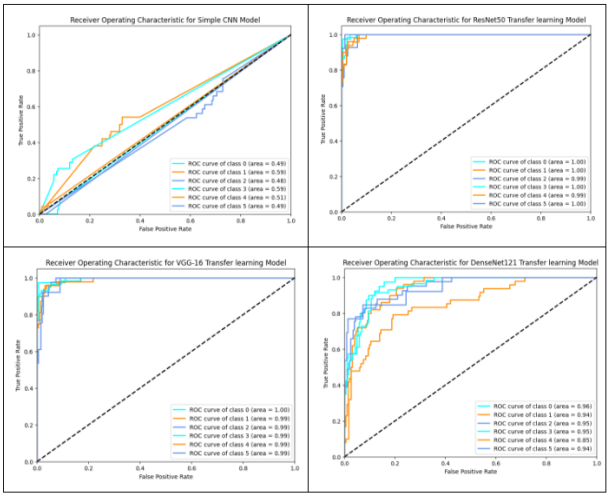

Fig. 14. Comparison of ROC Curves across 4 models

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 preprocessing. 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.

As shown in "Fig. 15", ResNet50 and VGG-16 are able to accurately predict the material of the object and it matches with the ground truth, whereas simple CNN misclassified everything, and DenseNet121 misclassified paper as metal and trash.

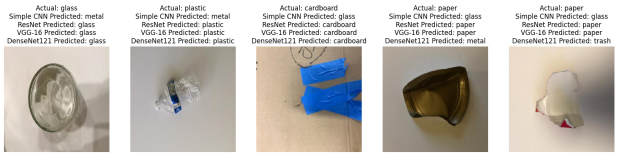

Fig. 15. Model Prediction vs Actual Label

V. Model Chosen: Majority Voting

As it can be seen from the above metrics, ResNet50, VGG-16 and DenseNet121 are able to relatively accurately predict the images given. To improve our accuracy, we took a step further and tried out other methods that can combine all the knowledge that we have learnt: hard voting. This technique often leads to:

- **Increased Accuracy**: The combination of different models reduces the risk of an incorrect prediction, as individual errors are likely to be outvoted by the correct ones from other models.
- **Reduced Overfitting**: Different models may overfit to different aspects of the training data, but when combined, the diverse perspectives can cancel out the overfitting.

We are able to combine the predictions of all our classifiers, whereby every individual classifier will vote, and the majority wins.

```
def predict_with_model(model, generator):
    steps = np.ceil(generator.samples / generator.batch_size)
    predictions = model.predict(generator, steps=steps)
    return np.argmax(predictions, axis=1)

def majority_voting(predictions_list):
    predictions_per_image = np.array(predictions_list).T
    majority_votes = [np.bincount(image_preds).argmax() for image_preds in predictions_per_image]
    return majority_votes

#only testing with two models now my cnn and my resnet50, to replace with the actual models - resnet50 vgg16 and densenet
# predictions_model_tl = predict_with_model(model_tl, test_generator)
# predictions_model = predict_with_model(model, test_generator)
predictions_cnn = predict_with_model(model, test_generator)
predictions_resnet50 = predict_with_model(model_tl, test_generator)
predictions_vgg16 = predict_with_model(model_tl_vgg16, test_generator)
predictions_densenet121 = predict_with_model(model_tl_dense, test_generator)

# predictions_placeholder = np.random.choice([0, 1], size=(len(test_generator.filenames)))

final_predictions = majority_voting([predictions_cnn, predictions_resnet50, predictions_vgg16])
```

Fig. 16. Majority voting code

A. Methodology

The approach consists of the following steps:

- **Prediction Generation**
  Individual models make predictions on the same set of input images. These models include custom-trained CNNs and pre-trained models like ResNet50 and VGG16. Pre-trained models are typically trained on large datasets like ImageNet and are fine-tuned for specific tasks, which provides a good starting point due to their already learned features.
- **Majority Voting**
  After predictions are made, a majority voting system is applied.For each of the images, the predicted class labels from all models are considered, and the final prediction is the one that the majority of models agree on.
- **Testing with accuracy as metric**
  Afterwards, we will use accuracy as a metric, which was able to accurately give us a correct result.

```
ground_truth = test_generator.classes

correct_predictions = sum(final_predictions == ground_truth)

total_predictions = len(ground_truth)
accuracy = correct_predictions / total_predictions

print(f'Accuracy: {accuracy * 100:.2f}%')

#this prints out the accuracy - to do comparison between the 3 methods after this
```

```
Accuracy: 96.60%
```

Fig. 17. Majority voting accuracy

The accuracy is high at 96.6%, significantly higher than the models, with VGG-16 coming in only at 92%. We went ahead to deploy this concept of majority voting in our GUI, as it provides us with the greatest accuracy.

VI. Graphical User Interface (GUI)

With our model ready, we created a user interface (UI) to allow users to upload a photo to be classified. Our waste classification web application aims to empower users to make informed decisions about waste management and recycling. Through a seamless UI, users can upload images of waste items, which are then classified into one of six categories: Cardboard, Glass, Paper, Metal, Plastic, or Trash. Additionally, users can explore detailed information about each waste category, including recycling instructions and environmental impact.

A. Implementation

1) Frontend - React Vite: The frontend of the application is built using React, a JavaScript library known for its component-based architecture and efficient state management. The user interface consists of several components, each responsible for different aspects of the application's functionality.

- **HomePage.jsx**
  This component serves as the main entry point of the application. It manages the layout and transitions between different views based on user interactions.
- **Landing.jsx**
  Acting as the landing page, this component provides introductory text about waste recycling and encourages users to take action. To the right, it also lists various waste categories as clickable buttons for exploration.
- **Classify.jsx**
  Here, users can upload images for classification. The component utilizes Axios to send image data to the Flask backend, where our model classifies the waste item. The predicted category is then displayed to the user along with the link to explore further about the classified category.
- **Explore.jsx**
  When users click on the link in the Classification page, they will navigate to this component where they can learn

more about the specific waste category. It presents description as well as recycling instructions of the selected category.

Throughout the development process, emphasis was placed on creating a responsive and visually appealing user interface. Smooth animations and consistent styling were implemented using the motion library to enhance the overall user experience.

2) Backend - Flask + Keras: The backend of our application is powered by Flask, a lightweight and flexible web framework for Python, renowned for its simplicity and extensibility. Below are the key components and functionalities of the backend implementation:

- **Flask Setup** The Flask application is initialized with CORS (Cross-Origin Resource Sharing) enabled to allow communication between the frontend and backend hosted on different domains via HTTP requests.
- **Model Loading** Upon initialization, the Flask server loads the .h5 file of our model that is responsible for classifying waste images.
- **Classification Endpoint** The 'classify' route serves as the endpoint for image classification requests. When a user uploads an image via the frontend, the image data is sent to this endpoint.
  - **Image Preprocessing**
    Upon receiving an image file from the frontend, the backend preprocesses the image before classification: The image data is read and converted into a byte stream using the BytesIO module. Using the image module from Keras, the byte stream is loaded as an image with our model's target input size of 300x300 pixels. It is then converted to a numerical array representation that matches the input shape expected by the model.
  - **Image Prediction**
    Upon preprocessing, the preprocessed image array is passed to the trained machine learning model for prediction. The predicted label is encapsulated in a JSON object and returned to the frontend for display to the user.

By leveraging Flask for backend development, we ensure efficient handling of image classification requests while maintaining scalability and ease of maintenance. The seamless integration of Flask with the React frontend results in a cohesive and user-friendly waste classification web application that empowers users to make informed decisions about waste management and recycling practices.

B. Final Product

Step 1: User arrives at the Landing Page. Here, users can click on "Take Action" to upload images for classification. or learn more about various waste categories listed as clickable buttons on the right ("Fig. 18").

Step 2: Once users click on "Take Action", it will take them to the Classification Page. Here, users can upload an image for classification ("Fig. 19").

Fig. 18. GUI Landing Page

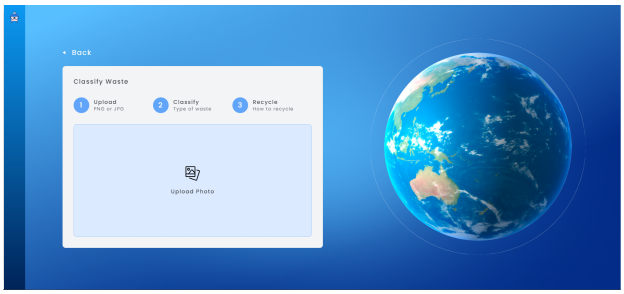

Fig. 19. GUI Classification Page - Upload

Step 3: Once the user has uploaded an image, the application processes the image and classifies it into one of the waste categories ("Fig. 20").

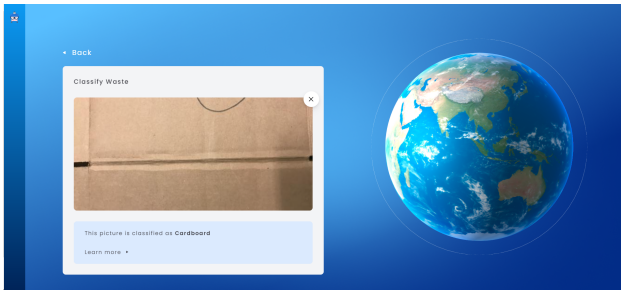

Fig. 20. GUI Classification Page - Classified

Step 4: After receiving the classification result, users can click on "Learn more" to learn more about the predicted waste category, including recycling instructions and environmental impact ("Fig. 21").

Fig. 21. GUI Exploration Page

More examples on each class and their results can be found in the Appendix

VII. Future Works

Our model can be further improved as we have identified a few problems:

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 being unable 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) 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.

Future improvements to overcome this includes:

1) Training our models on a larger dataset with more images for each class.
2) Using ensemble learning methods to increase the overall accuracy of our proposed models.

However, due to the interest of time, most of the above were not able to be performed well. For instance, we initially tried combining other image datasets to increase our dataset size, but this made the classes very unbalanced, with each class images being drastically different (background, colour schemes and edge pixels). This caused a large decrease in our F1 Score and accuracy metrics, hence we decided against it given the time constraints.

Beyond its technical achievements, our project holds significant implications for waste management practices and environmental sustainability. By automating the classification process, our system has the potential to streamline waste sorting operations in low-income countries, reduce human error, and enhance overall efficiency in recycling facilities and waste management plants.

Furthermore, the scalability and adaptability of our solution make it well-suited for integration into existing waste management infrastructure, offering a practical and cost-effective means of improving waste sorting processes on a broader scale.

Looking ahead, further research and development efforts could focus on refining our model's performance, expanding the dataset to encompass a broader range of waste materials, and exploring additional features or techniques to enhance classification accuracy.

Our project represents a significant step towards achieving more sustainable and environmentally conscious waste management practices, ultimately contributing to the preservation of our planet for future generations.

References

[1] D. Filipenco, "DevelopmentAid," DevelopmentAid, Mar. 07, 2023. https://www.developmentaid.org/news-stream/post/158158/world-waste-statistics-by-country

[2] Kaggle. (2019). Garbage Classification Dataset. Retrieved from https://www.kaggle.com/datasets/asdasdasasdas/garbage-classification

[3] 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.

[4] 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

[5] 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

[6] J. McDermott, "Hands-on Transfer Learning with Keras and the VGG16 Model," www.learndatasci.com. https://www.learndatasci.com/tutorials/hands-on-transfer-learning-keras/

[7] T. Welch and M. Comolli, "Multi-Label Waste Classification with Data Augmentation." Available: https://cs230.stanford.edu/projects_spring_2021/reports/10.pdf

[8] "TrashBox: Trash Detection and Classification using Quantum Transfer Learning — IEEE Conference Publication — IEEE Xplore," ieeexplore.ieee.org. https://ieeexplore.ieee.org/document/9770922

Appendix

A. Code Repository

Link to Code Repository

B. GUI Demo Video

Link to Demo Video

C. DenseNet121 Hyperparameter Tuning

Link to Hyperparameter Tuning Trials

D. GUI Image Classification

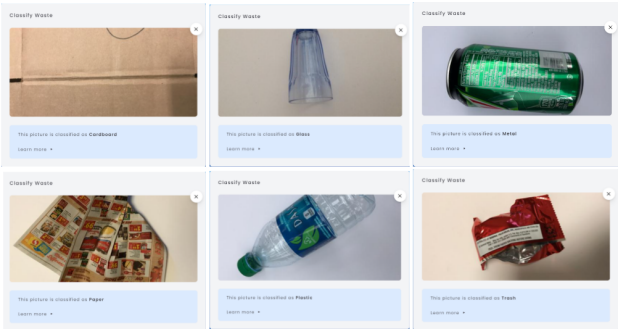

Fig. 22. Examples of Image Classification for All Labels

E. GUI Waste Exploration

Fig. 23. Waste Recycling Instructions and Environmental Impact on Exploration Page