Introduction: (from the group report)

Group 3 has decided to improve on a body of existing code that classifies a series of images, known as TrashNet. Our task is to improve model efficacy, facilitating better garbage sorting recognition. The ultimate goal is to improve the pre-trained models in our baseline code via transfer learning, select more modern models, then build an application which showcases our ability to correctly classify trash.

Detailed in this whitepaper will be a description of the data, background on the networks, our setup for experimentation, results, and conclusions.

Individual work entry(what I did and results):

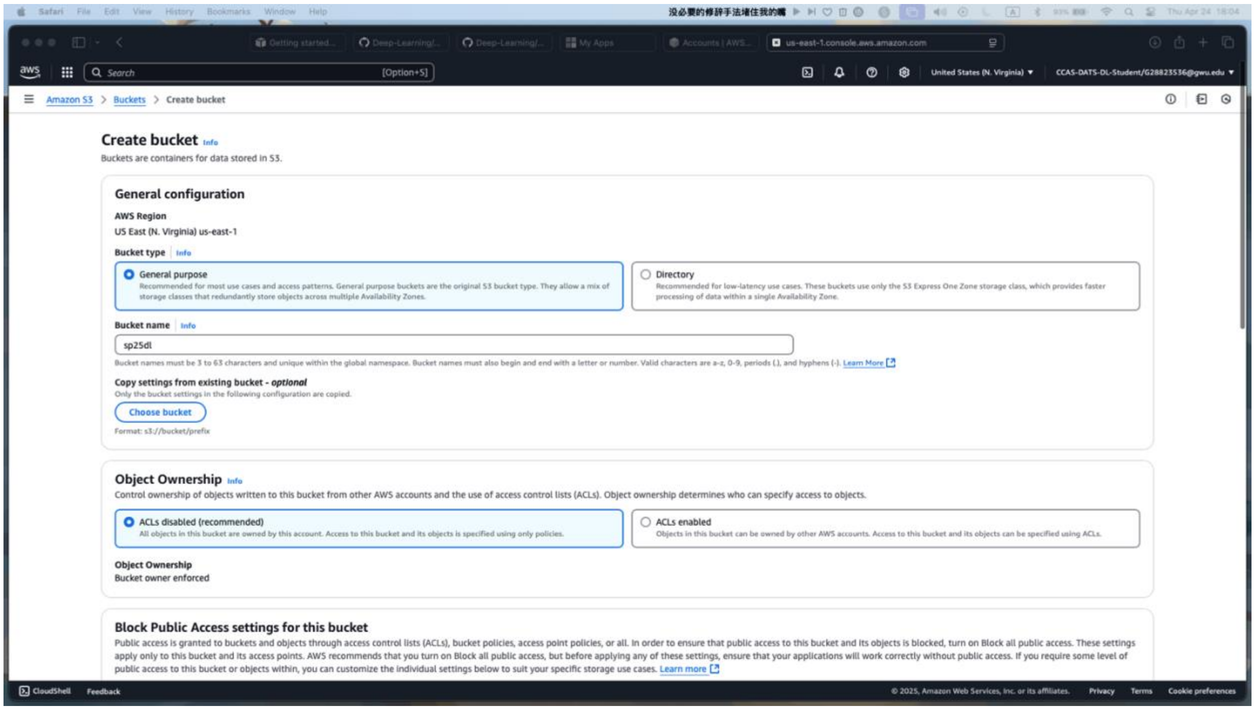
→(I am so sorry about the formatting but this was how I document my part when I was doing research/testing)

→ published all versions (mentioned below) into github JupiK9 branch (should be merged together now)

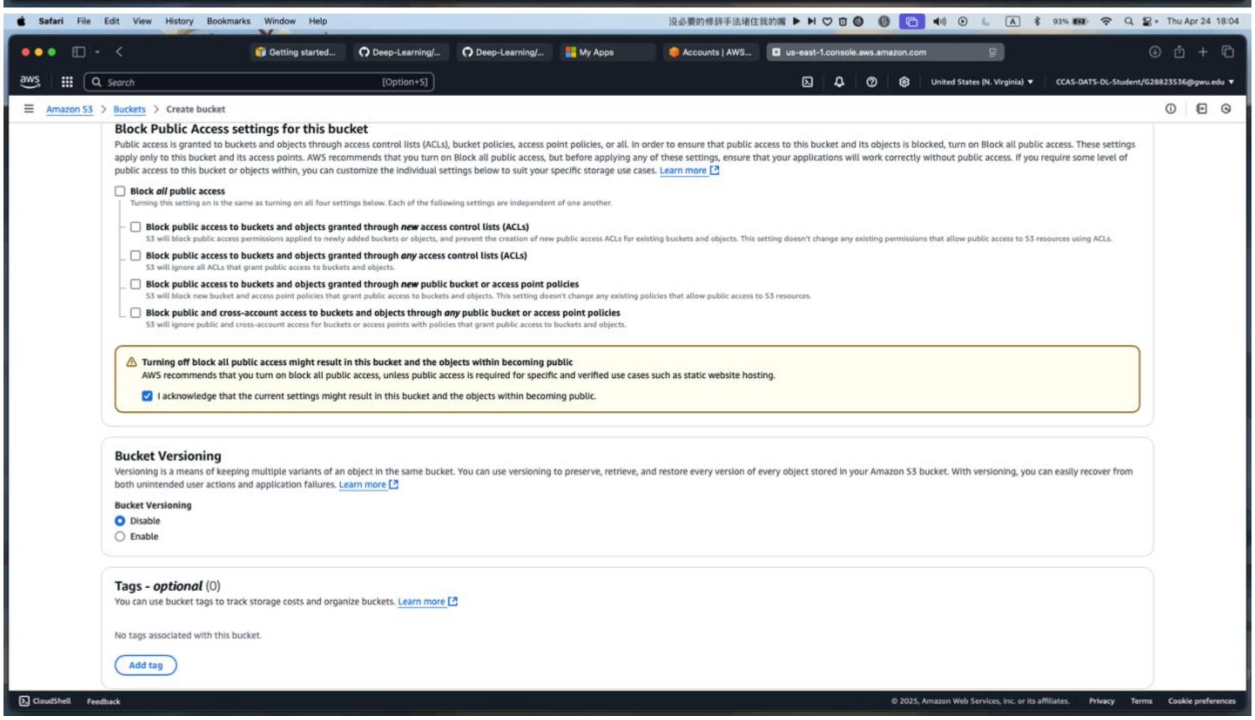
Data storage

Originally want to store it in S3 bucket using AWS. Went to create the bucket, only to find out that our deep learning class doesn't give permission to create S3 (also don't want to spend money) so ended up handing down to Stephen using GCP.

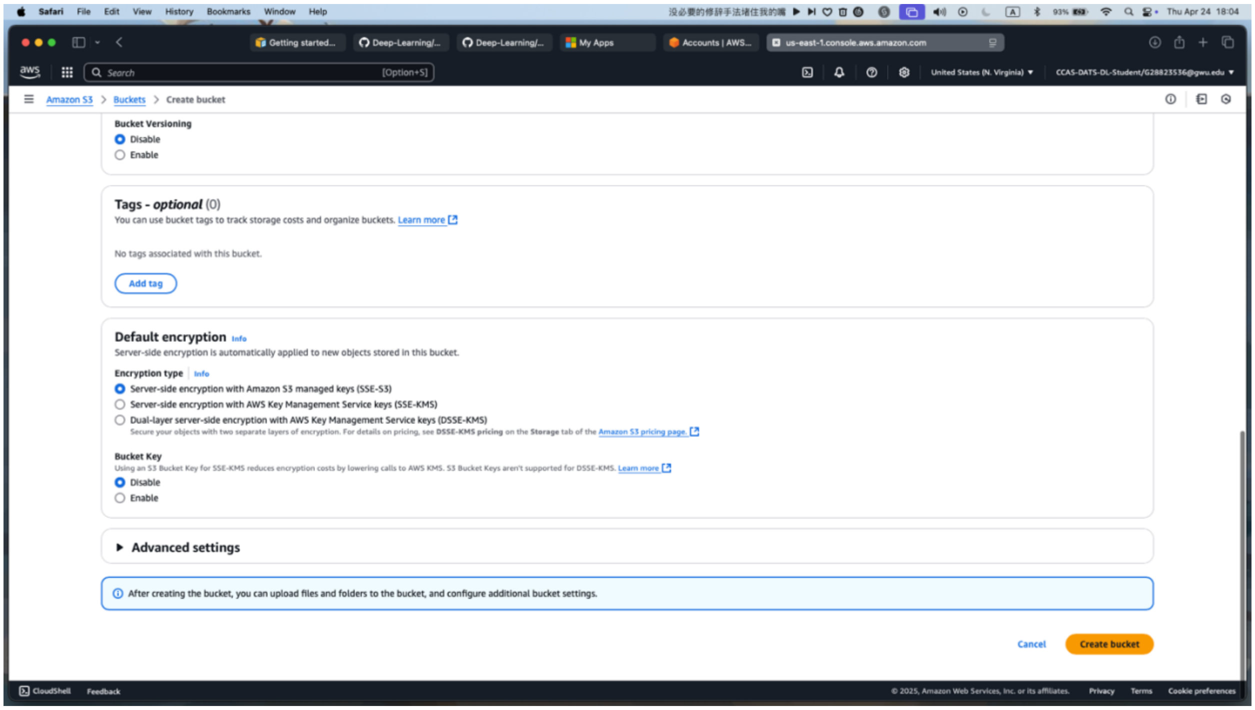
- Result: the S3 bucket not created successfully, because no permission guaranteed from the DL class- AWS account. Don't want to spend money
- Steps of setting up S3 bucket:

- 

Amazon S3 > Buckets > Create bucket
Create bucket Info
Buckets are containers for data stored in S3.
General configuration
AWS Region
US East (N. Virginia) us-east-1
Bucket type Info
General purpose
Recommended for most use cases and access patterns. General purpose buckets are the original S3 bucket type. They allow a mix of storage classes that redundantly store objects across multiple Availability Zones.
Directory
Recommended for low-latency use cases. These buckets use only the S3 Express One Zone storage class, which provides faster processing of data within a single Availability Zone.
Bucket name Info
sp25dl
Bucket names must be 3 to 63 characters and unique within the global namespace. Bucket names must also begin and end with a letter or number. Valid characters are a-z, 0-9, periods (.), and hyphens (-). Learn More
Copy settings from existing bucket - optional
Only the bucket settings in the following configuration are copied.
Choose bucket
Format: s3://bucket/prefix
Object Ownership Info
Control ownership of objects written to this bucket from other AWS accounts and the use of access control lists (ACLs). Object ownership determines who can specify access to objects.
ACLs disabled (recommended)
All objects in this bucket are owned by this account. Access to this bucket and its objects is specified using only policies.
ACLs enabled
Objects in this bucket can be owned by other AWS accounts. Access to this bucket and its objects can be specified using ACLs.
Object Ownership
Bucket owner enforced
Block Public Access settings for this bucket
Public access is granted to buckets and objects through access control lists (ACLs), bucket policies, access point policies, or all. In order to ensure that public access to this bucket and its objects is blocked, turn on Block all public access. These settings apply only to this bucket and its access points. AWS recommends that you turn on Block all public access, but before applying any of these settings, ensure that your applications will work correctly without public access. If you require some level of public access to this bucket or objects within, you can customize the individual settings below to suit your specific storage use cases. Learn more

- 

Amazon S3 > Buckets > Create bucket
Block Public Access settings for this bucket
Public access is granted to buckets and objects through access control lists (ACLs), bucket policies, access point policies, or all. In order to ensure that public access to this bucket and its objects is blocked, turn on Block all public access. These settings apply only to this bucket and its access points. AWS recommends that you turn on Block all public access, but before applying any of these settings, ensure that your applications will work correctly without public access. If you require some level of public access to this bucket or objects within, you can customize the individual settings below to suit your specific storage use cases. Learn more
Block all public access
Block public access to buckets and objects granted through new access control lists (ACLs)
Block public access to buckets and objects granted through any access control lists (ACLs)
Block public access to buckets and objects granted through new public bucket or access point policies
Block public and cross-account access to buckets and objects through any public bucket or access point policies
Turning off block all public access might result in this bucket and the objects within becoming public
AWS recommends that you turn on block all public access, unless public access is required for specific and verified use cases such as static website hosting.
I acknowledge that the current settings might result in this bucket and the objects within becoming public.
Bucket Versioning
Versioning is a means of keeping multiple variants of an object in the same bucket. You can use versioning to preserve, retrieve, and restore every version of every object stored in your Amazon S3 bucket. With versioning, you can easily recover from both unintended user actions and application failures. Learn more
Bucket Versioning
Disable
Enable
Tags - optional (0)
You can use bucket tags to track storage costs and organize buckets. Learn more
No tags associated with this bucket.
Add tag

- 

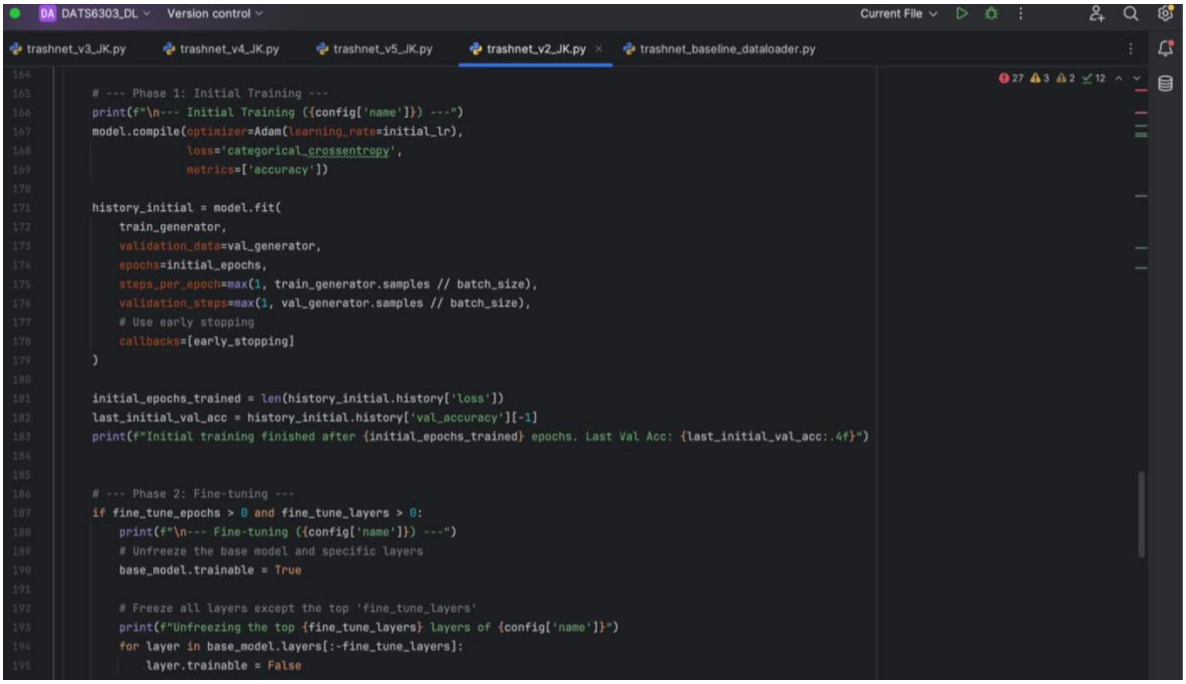
Improving baseline code -v2

Experimental setups: including training strategy, hyperparameter control ,regularization, data splitting, data augmentation and optimizer.

Training strategy:

- Two phase training:
  - Phase 1 – initial training: trains only the new classification head for initial_epochs. Quickly trains the new layers to adapt to the dataset based on general features
  - Phase 2 – fine-tuning: selectively freezes all layers except the top fine_tune_layers. Recompiled with low learning rate and trained for additional fine_tune_epochs. → allows the model adjust pre-trained weights

in the later layers, becoming more complex and leads to significant accuracy improvements.
    - Using low learning rate is crucial here to avoided catastrophically disrupting the valuable pre-trained weights

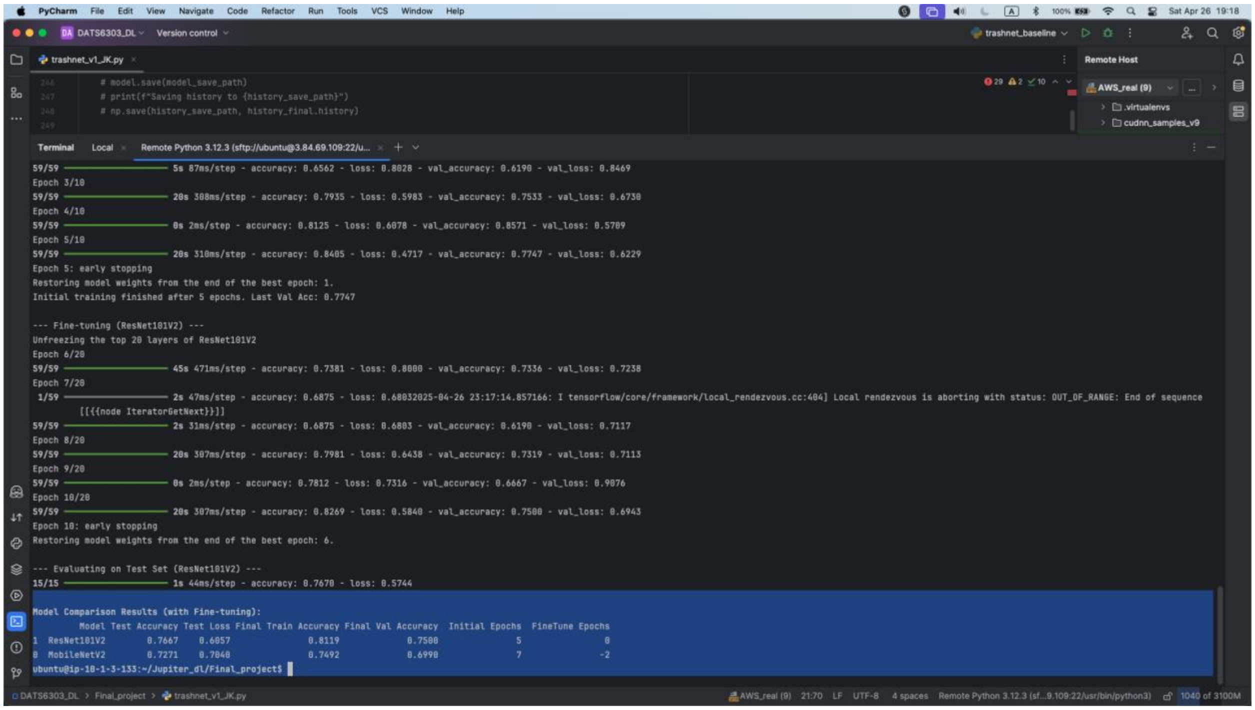


```
# --- Phase 1: Initial Training ---
print(f"\n--- Initial Training ({config['name']}) ---")
model.compile(optimizer=Adam(learning_rate=initial_lr),
              loss='categorical_crossentropy',
              metrics=['accuracy'])

history_initial = model.fit(
    train_generator,
    validation_data=val_generator,
    epochs=initial_epochs,
    steps_per_epoch=max(1, train_generator.samples // batch_size),
    validation_steps=max(1, val_generator.samples // batch_size),
    # Use early stopping
    callbacks=[early_stopping]
)

initial_epochs_trained = len(history_initial.history['loss'])
last_initial_val_acc = history_initial.history['val_accuracy'][-1]
print(f"Initial training finished after {initial_epochs_trained} epochs. Last Val Acc: {last_initial_val_acc:.4f}")


# --- Phase 2: Fine-tuning ---
if fine_tune_epochs > 0 and fine_tune_layers > 0:
    print(f"\n--- Fine-tuning ({config['name']}) ---")
    # Unfreeze the base model and specific layers
    base_model.trainable = True

    # Freeze all layers except the top 'fine_tune_layers'
    print(f"Unfreezing the top {fine_tune_layers} layers of {config['name']}")
    for layer in base_model.layers[:-fine_tune_layers]:
        layer.trainable = False
```

- hyperparameter control and flexibility
    - introduces more specific hyperparameters: initial_epochs, fine_tune_epochs, initial_lr, fine_tune_lr, fine_tune_layers.
        - To precisely control the two phase training process
        - Separate learning rates are crucial for fine-tuning
        - Specifying the number of layers to unfreeze the fine_tune_layers provides control over how much of the base model is adapted.
- Regularization
    - Added a dropout layer before the final output layer
        - Helps prevent overfitting problem
        - How: randomly set a fraction of neuron outputs to 0 during training, forcing the network to learn more robust features
    - Implements earlystopping during both training phases.
        - Monitors the validation loss or accuracy

- Stops training if it doesn't improve for a set number of epochs: early_stopping_patience
- Prevents overfitting by stopping before the model starts performing worse on unseen data,
- Save training time
- Restores the weights from the best performing epoch

Result (only with two model configs: ResNet101V2 and MobileNetV2)



1. V3

- slightly changed the test split from the baseline code to 20%.
- dataset is not big enough and we have to use all we have, the original baseline did a second ImageDataGenerator, in which it has its own train-val 75/25.
- leaving it as a potential data leakage problem, due to the independent application of val_split by each generator on the full dataset,
- test samples included in either train or validation set of the second generator, as well as leading overlap and miscounts.

→ the day we have proper data storage and data loader

only add back all the models

Result of having all model config trained, tested and validated.

Model Comparison Results (with Fine-tuning):

	Model	Test Accuracy	Test Loss	Final Train Accuracy	Final Val Accuracy	Initial Epochs	FineTune Epochs
2	ResNet152V2	0.7896	0.6110	0.7974	0.7516	5	0
1	ResNet101V2	0.7688	0.6442	0.8151	0.7566	5	2
0	MobileNetV2	0.7292	0.7354	0.7401	0.6908	7	-2
3	MobileNet	0.6729	0.8270	0.7176	0.6595	5	0
5	MobileNetV3Large	0.3583	1.4839	0.3248	0.3257	5	0
4	MobileNetV3Small	0.2542	1.6552	0.2219	0.2549	5	0

V4:

- optimizer: to AdamW, added weight_decay hyperparameter
    - decouples weight decay from the adaptive learning rate adjustments
    - more effective way to apply regularization, can lead to better model generalization and final performance

For v4.py, some of the improvements are:

- Increase epoch slightly, for initial, fine_tune;
- Slightly decrease fine_tune LR

- Unfreeze more layers
- Increase patience for callback
- Weighted decay parameter for AdamW
-
- Add another dense layer and dropout to the head: (line 168)

    ```
    # Add new dense layer:
    x = GlobalAveragePooling2D(name="global_avg_pooling")(base_model.output)
    x = Dense(256, activation='relu', name="dense_256")(x)
    x = Dropout(dropout_rate, name="dropout_1")(x)
    x = Dense(128, activation='relu', name="dense_128")(x)
    x = Dropout(dropout_rate, name="dropout_2")(x)
    outputs = Dense(6, activation='softmax', name="output_softmax")(x)
    ```
-
- In training loop – phase 2:
    - Freeze all layers except for the top fine_tune_layer
    - Use AdamW optimizer with fine-tune LR and weight decay

        ```
        # Use AdamW optimizer
        optimizer_finetune = AdamW(learning_rate=fine_tune_lr, weight_decay=weight_decay)
        model.compile(optimizer=optimizer_finetune,
                      loss='categorical_crossentropy',
                      metrics=['accuracy'])
        ```
-
- Sort the result as best val accuracy before looking at the test accuracy
- Result::

```
Model Comparison Results (Improved Tuning):
             Model  Test Accuracy  Test Loss  Best Val Accuracy  Final Train Accuracy  Initial Epochs Run  FineTune Epochs Run
0      MobileNetV2         0.7354     0.6738             0.9048                0.7460                  15                   -8
2      ResNet152V2         0.7625     0.7249             0.8095                0.6897                   7                    0
1      ResNet101V2         0.7417     0.7820             0.7812                0.6190                   7                    0
3        MobileNet         0.6500     0.8861             0.7023                0.5740                   7                    0
4  MobileNetV3Small        0.3417     1.6610             0.4286                0.2047                   7                    0
5  MobileNetV3Large        0.4021     1.5741             0.4013                0.2524                   7                    0
ubuntu@ip-10-1-3-133:~/Jupiter_dl/Final_project/DEEP_LEARNING_6303_GROUP3/Code/Trashnet/code$
```
-

V5

Stephen made a class weight parameter, fine tuning in alpha parameter. Because we never actually classified them, so they've been underrepresented. After adding the class weights by changing the alpha level, (that's when I started to play around with the data)

By increasing alpha value from 0.2 to 0.5: the result increases:

Model Comparison Results (with Fine-tuning):

	Model	Test Accuracy	Test Loss	Final Train Accuracy	Final Val Accuracy	Initial Epochs	FineTune Epochs
0	EfficientNetV2S	0.8458	0.4713	0.8934	0.8240	10	-5
2	ResNet101V2	0.7750	0.6347	0.7958	0.7385	5	0
3	ResNet152V2	0.7688	0.6384	0.7958	0.7253	5	0
1	MobileNetV2	0.7229	0.7663	0.7224	0.6957	5	0
4	MobileNet	0.7125	0.7649	0.6961	0.6826	5	0
6	MobileNetV3Large	0.4250	1.5067	0.3065	0.3684	5	0
5	MobileNetV3Small	0.2479	1.7233	0.1870	0.2401	5	0

```
ubuntu@ip-10-1-3-133:~/Jupiter_dl/Final_project/DEEP_LEARNING_6303_GROUP3/Code/Trashnet/code$ python3 trashnet_v5_JK.py
```

My version: need to see if increasing alpha helps with model performance or just an accident, so here are some new changes:

- Define alpha values to test for class weight damping: 0,0.2,0.5,0.8,1
- Shuffle train data, don't shuffle val data
- Outer loop for testing diff alpha values: so that the result comes inside the loop.

```
# ---  Outer loop for testing different alpha values --- NEW JK
for alpha in alpha_values_to_test:
    print(f"\n\n{'='*20} TESTING ALPHA = {alpha:.2f} {'='*20}\n")

    class_weight_damped = 1.0 + alpha * (base_weights - 1.0)
    class_weight_dict = dict(enumerate(class_weight_damped))
    print(f"Using Class Weights for alpha={alpha:.2f}: {class_weight_dict}")
```

-
- Usually, using alpha level can help with enhancing model performance, because alpha adjusts the balance between classes, similar to weighted cross-entropy but with added flexibility.
    - Highly skewed datasets: in datasets where the minority class is extremely rare, setting alpha around 0.25 for the majority class and 0.75 for the minority class can be effective.
- not really worth it because it takes way too long to run the script

- alpha level is not automatically uploaded, selected randomly with not very precise calculation or based off results from the model performance. So disgard.
- ///
-
- data augmentation:
    - adding Sheer_range , brightness_range to ImageDataGenerator used for training and validation
        - to further increase the diversity of the training data by applying more types of random transformations
        - helps model become more robust to variations, reduce overfitting

citation:

Yassin, A. (2024, November 13). Adam vs. AdamW: Understanding Weight Decay and Its Impact on Model Performance. *Medium*. https://yassin01.medium.com/adam-vs-adamw-understanding-weight-decay-and-its-impact-on-model-performance-b7414f0af8a1

Code percentage:

Roughly: original script 136 lines

Me editing/modifying/ 180 lines

This is very hard to say because the baseline code is a easier model so since v2 I didn't trace every single line, but I did generated the codes and modified the codes after understanding them. There's no reason for me to lie and say I hard code every single line but I do understand them and with the help of how to make the code writes smoother and better. The architecture is indeed very interesting and it is hard to combine what we learnt in class to a real life case. I enjoy the class.