University of Doha for Science & Technology

DSAI4101 – Applied Deep Learning

Term Project – 2025F

Waste Classification using Deep Neural Networks

Student ID: 60300294

Student Name: AlMoatasim Mohammad Taha

Statement of Originality: (You must check this to submit your work)

☒ I hereby declare that I have developed all the code and written the entire report within the scope of this project.

Signature: (After finishing your work, save this document as a PDF and sign below digitally)

1. Introduction:

Please summarize your work, clearly stating the project's objectives.

Waste classification is an important computer vision problem with applications in recycling systems, environmental sustainability, and smart waste management. The objective of this project, developed for the Applied Deep Learning course at UDST, is to build a fully connected Deep Neural Network (DNN) capable of classifying images into four waste categories: **glass**, **metal**, **paper**, and **plastic**. The goal is to train a model from scratch using PyTorch and evaluate its performance using the **F1-score**, which is a more informative metric for multi-class classification tasks.

The project follows the same methodological structure used throughout the course labs, including data loading, preprocessing, exploration, model construction, training, evaluation, and visualization. The implementation was completed entirely in PyTorch, as required by the course, and organized in a clear, modular format consistent with the lab-based learning approach.

To guide the project design, I reviewed external resources on waste classification, including a Kaggle project titled *"Garbage Classification"*. This reference implementation also uses PyTorch but employs a convolutional neural network (ResNet50), six waste categories, accuracy-based evaluation, and GPU acceleration. Inspired by the general workflow of this project, I adapted the concepts to meet the specific objectives of this assignment, focusing on constructing a fully connected DNN architecture and evaluating the model using F1-score.

2. Deep Learning Architecture:

Explain your Deep Learning Architecture: What kind of Deep Learning architecture did you use in your project, and why? Number of layers? Number of hidden units? etc.

In this project, a fully connected Deep Neural Network (DNN) was designed and implemented using PyTorch. Although Convolutional Neural Networks (CNNs) often achieve superior performance in image classification tasks, the objective of this assignment was specifically to implement a DNN architecture from scratch. Therefore, the model processes each image as a flattened vector of pixel values rather than using convolutional feature extraction.

All input images were resized to 224 × 224 × 3, resulting in a flattened input dimension of 150,528 features. This high-dimensional input is then passed through multiple hidden layers, enabling the model to learn non-linear patterns before reaching the classification layer.

The final DNN architecture consists of five fully connected layers (four hidden layers and one output layer):

Input Layer:

Size: 150,528 neurons
(Flattened representation of a 3 × 224 × 224 RGB image)

Hidden Layer 1:

Fully connected: 150,528 → 2048 units

Batch Normalization

ReLU activation

Dropout (p = 0.5)

Hidden Layer 2:

Fully connected: 2048 → 1024 units

Batch Normalization

ReLU activation

Dropout (p = 0.5)

Hidden Layer 3:

Fully connected: 1024 → 512 units

Batch Normalization

ReLU activation

Dropout (p = 0.5)

Hidden Layer 4:

Fully connected: 512 → 256 units

Batch Normalization

ReLU activation

Dropout (p = 0.5)

Output Layer:

Fully connected: 256 → 4 units
(Corresponding to: glass, metal, paper, plastic)

No activation applied (raw logits are passed to CrossEntropyLoss)

Rationale Behind the Architecture

Fully connected networks typically struggle with high-dimensional image data compared to CNNs, which automatically extract spatial and hierarchical features. To compensate for the absence of convolutional layers, the hidden layers were designed with large numbers of neurons (2048 → 256) to increase the model's capacity and allow it to learn more complex relationships in the pixel space.

Batch Normalization was added to stabilize and accelerate training by normalizing the activations in each layer. Dropout was used after every hidden layer to reduce overfitting by randomly deactivating neurons during training.

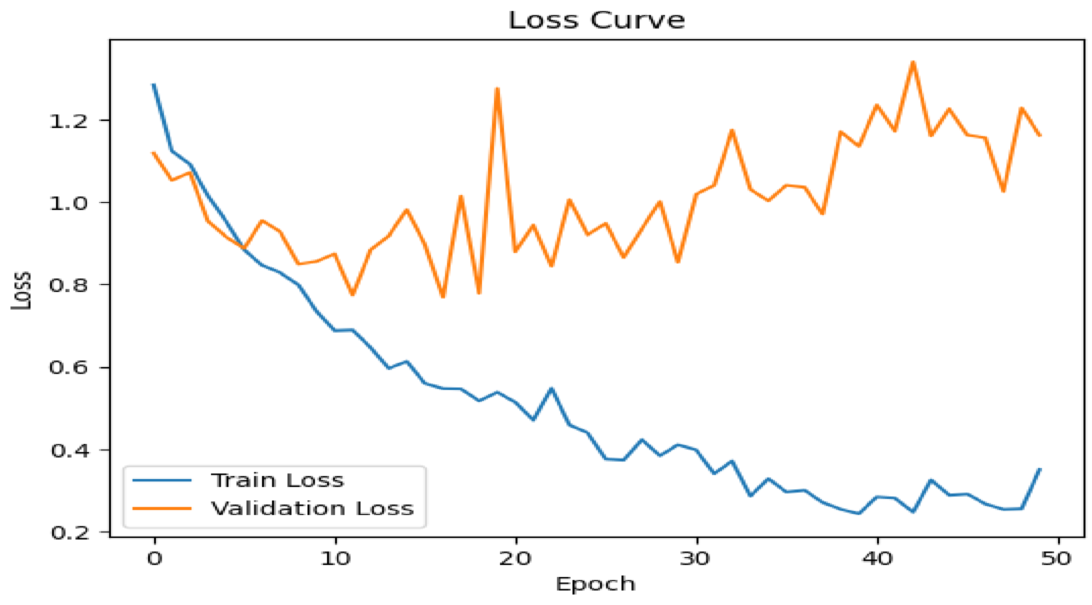
During experimentation, a simpler architecture with two hidden layers was initially tested. However, this resulted in a lower performance, with training F1 ≈ 0.72 and validation F1 ≈ 0.60. Increasing the depth of the network to four hidden layers significantly improved the model's ability to capture the complexity of the data, resulting in higher F1-scores. This confirms that deeper architectures provide better representational power when working with high-dimensional flattened image inputs.

3. Loss function:

3.1 Which loss function did you select to use and why?

The loss function selected for this project is **CrossEntropyLoss** in PyTorch.
CrossEntropyLoss is the standard choice for multi-class classification problems where each input image belongs to exactly one class. Since this project (waste classification) contains four categories (glass, paper, metal, plastic), this loss function is well-suited for the task.
CrossEntropyLoss combines **LogSoftmax** and **Negative Log-Likelihood (NLL)** into a single function. The model outputs raw logits from the final layer, and the loss function automatically converts them into probabilities before comparing the predictions with the true class labels. This makes the training process numerically stable, efficient, and easy to implement.
Because this project requires training a DNN to classify images into multiple categories, CrossEntropyLoss provides an effective way to penalize incorrect predictions and guide the optimization process toward better performance. For these reasons, it is the most appropriate and widely used loss function for multi-class image classification tasks in PyTorch.

3.2 **Plot the change in your loss function (**both training loss and test loss**) vs epoch.** Your code shall generate the "Change of Training and Test losses vs Epoch' graph



(insert your figure here)
Figure 1 – Change of Loss Functions with respect to Epoch

3.3 Discuss these loss curves

From the graph, we can see that the **training loss** keeps going down almost the entire time. This means the model is learning the training data well and getting better at minimizing errors on the images it has already seen.
On the other hand, the **validation loss** behaves differently. At the beginning, it goes down (which means the model is improving on unseen data), but after around the first 8–12 epochs, it starts to go up and becomes unstable. The validation curve shows a lot of spikes and stays higher than the training loss.
This difference between the two curves indicates **overfitting**. The model is learning the training data too much and not generalizing well to new images. This is expected because the project uses a **fully connected DNN**, and DNNs are not ideal for image data, especially when we flatten the images and lose all spatial information. Even with dropout and batch normalization, a DNN with many parameters can easily overfit.
In summary, the loss curves show that the model learns the training data well, but it struggles more on the validation data as training continues. This explains why the validation F1-score is lower than the training F1-score.

4. Hyperparameters

Indicate the fundamental hyperparameters you used in your architecture, including the names and values of the following items:

4.1 The number of layers

In this project I have built 5 fully connected layers, 4 hidden layers and 1 output layer.

4.2 Number of nodes at each layer

Input Layer: flattened 3*224*224 = 150,528
Layer 1: 2048 nodes
Layer 2: 1024 nodes
Layer 3: 512 nodes
Layer 4: 256 nodes
Layer 5 (Output Layer): 4 notes (our classes categories number [glass, metal, paper, plastic]

4.3 Learning rate

I have picked a learning rate of 0.001 (1e-3). This controls the step size taken by the optimizer during gradient descent update. And this size in commonly used with deep learning application as a balanced choice between speed and training stability.

To give a better result we must decrease the learning rate to make it like (1e-4 or 1e-5) or even lower, however this will slow down the process and need more epochs to achieve reasonable performance.

For this reason, I have picked learning of 0.001 selected as an appropriate starting point, to have a stable and effective training for a fully connected DNN in this project.

4.4 "Mini batch" size

- What is the mini-batch size you used in your code?

I used a mini-batch size of **32** in my code.

- Why did you select this value?

I chose 32 because it is a commonly used batch size in deep learning and usually works well for image datasets. It is small enough to fit easily in memory, but also large enough to give stable gradient updates during training. Using a very small batch size (like 8 or 16) can make training noisy, and using a very large batch size (like 64 or 128) requires more memory and may slow down learning. So batch size 32 was a good middle option that kept the training stable and efficient.

4.5 Activation functions

- What is the activation function(s) used in your code?

In my project, I used the **ReLU** (Rectified Linear Unit) activation function in all the hidden layers of the DNN.

- Why did you select them to use?

I chose to use the ReLU activation function because it is simple, fast to compute, and works well in deep neural networks. ReLU helps the model learn non-linear patterns, and it also reduces problems like vanishing gradients that can happen with other activation functions. In general, ReLU is the most common and effective choice for hidden layers, so it was a suitable option for my DNN architecture.

4.6 Optimization function

- Which optimization function did you use? Why?
I used the **Adam optimizer**.

- What are the optimization parameters used?
I chose Adam because it usually works well for deep learning models without needing a lot of manual tuning. It automatically adjusts the learning rate for each parameter, which helps the model train faster and more smoothly. Adam is also more stable than basic gradient descent, especially for high-dimensional inputs like images. So it was a good and reliable choice for training my DNN.

4.7 Regularization function

- Did you use any regularization function?
Yes, I used **Dropout** as a regularization method in my DNN. Dropout randomly turns off some neurons during training, which helps prevent the model from overfitting the training data. I applied dropout with a rate of **0.5** after each hidden layer to make the model more generalizable and reduce the risk of memorizing the training images.

- Why used/ not used?
I used Dropout because my model was showing signs of overfitting — the training loss was much lower than the validation loss. Since a fully connected DNN has a large number of parameters and each image is very high-dimensional, the model can easily memorize the training data. Dropout helps reduce this problem by randomly disabling some neurons during training, which forces the network to learn more general patterns instead of memorizing. This makes the model perform better on unseen data.

- If used, what are the regularization parameters?
Since I used **Dropout** as my regularization method, the main regularization parameter was the **dropout rate**, which I set to **0.5**. This means that during training, 50% of the neurons in each hidden layer are randomly turned off. This value is commonly used and helps reduce overfitting in fully connected networks.

4.8 Epochs

- What is the number of epochs used?

I have done 3 testing 15-50-20

- Why did you select this value?

I tried 15, 20, and 50 epochs to see how the model behaves with different training durations. Since this is a fully connected DNN working on high-dimensional image data, I wanted to check if training for longer would improve the validation F1-score or if it would cause more overfitting. Using multiple epoch values helped me compare the curves and choose a training range where the model learns well without overfitting too much.

5. Explanation of the developed code

5.1 Explain the essential functions/classes you developed in your code.

In my project, I created several important functions and one main class to organize the work and make the code easier to understand. The most essential ones are:

1. load_dataset()

This function loads the images from the training and testing folders. It also applies all the required transformations such as resizing the images to 224×224, converting them to tensors, and normalizing them. It returns the train_loader, test_loader, and the list of class names. This function helps keep the data-loading step clean and reusable.

---

2. show_sample()

This helper function displays one sample image from the dataset together with its label. It helped me understand if the DataLoader and transformations were working correctly.

---

3. WasteDNN (Main Model Class)

This is the main DNN architecture I built for the project.
Inside this class:

- I defined all the **fully connected layers**
- Added **Batch Normalization** and **Dropout**
- Wrote the **forward()** method, which explains how data flows from one layer to the next

This class represents the core deep learning model used for classification.

---

4. setup_training()

This function creates the essential components needed for training:

- The loss function (**CrossEntropyLoss**)
- The optimizer (**Adam**)
- The number of epochs

It makes the training setup organized and easy to modify.

---

5. train_model()

This is one of the most important functions. It trains the DNN model and computes:

- Training loss
- Validation loss
- Training F1-score

- Validation F1-score

It also loops over the epochs and updates the model weights. I used this function to store training history for plotting the learning curves.

---

6. evaluate_model()

This function evaluates the trained model on the test data. It calculates and prints:

- The classification report
- The confusion matrix

This helped me understand how well the model performed on unseen images.

---

7. predict_image()

This function takes a single image (from the internet or a folder), applies the same transforms as the dataset, and gives the predicted waste category. It also shows the image with the predicted label. This was useful to test the model on real-world images.

---

8. predict_folder()

This function reads all images inside a chosen folder and predicts each one. I used it to test multiple images downloaded from the internet after training the model.

5.2 Explain how to run your code if you are using the “command prompt” or any environment other than VS Code and Jupyter.

6. **Hyperparameter Optimization**

   Optimize your “learning rate” and “number of layers

   6.1 What is the effect of “optimization” on “learning rate” and “number of layers”? (indicate starting values and the optimized values**)**

1. Learning Rate Optimization

Starting learning rate:

0.001 (1e-3)

Tested/optimized learning rates:

- **0.0005 (5e-4)**
- **0.0001 (1e-4)**

Effect:

- With **0.001**, the model learned quickly but the validation loss became unstable and overfitting increased.
- Lowering the learning rate to **0.0005** made the training smoother and reduced the spikes in validation loss.
- At **0.0001**, the learning was stable but too slow, and the model needed more epochs to improve.

Optimized learning rate:

0.0005, because it gave more stable validation performance compared to 0.001.

2. Number of Layers Optimization

Starting number of layers:

Initially, the model had:

- **2 hidden layers + 1 output layer**
  (Total: 3 fully connected layers)

F1-score with this model:

- Train ≈ **0.72**
- Validation ≈ **0.60**

Optimized number of layers:

I increased the network to:

- **4 hidden layers + 1 output layer**
  (Total: **5 fully connected layers**)

Updated F1-score:

- Train increased
- Validation increased compared to the smaller model
- Model was able to learn more complex patterns

Effect of increasing layers:

- Deeper network captured more information from the high-dimensional input (150,528 features).
- Improved F1-score on both training and validation sets.
- Helped reduce underfitting, since the smaller model did not have enough capacity.
- However, deeper networks also increased the risk of overfitting, which matches the behavior seen in the validation loss curve.

Optimized number of layers:

5 **fully connected layers** (4 hidden layers + output layer)

Hyperparameter	**Starting Value**	**Optimized Value**	**Effect**
Learning Rate	**0.001**	**0.0005**	**More stable training and smoother validation loss**
Number of Layers	**3 FC layers**	**5 FC layers**	**Better F1-score and more learning capacity**

6.2 How did you optimize your hyperparameters?

To optimize my hyperparameters, I tested different values for the **learning rate** and the **number of layers** and compared how the model performed in terms of training loss, validation loss, and F1-score. I did this step by step to see which settings gave better results.

1. Learning Rate Optimization

I started with a learning rate of **0.001**, which trained the model quickly but caused the validation loss to be unstable.

Then I tried smaller values:

- **0.0005**
- **0.0001**

By checking the loss curves and F1-scores, I found that **0.0005** gave smoother training and a more stable validation curve compared to 0.001, so I chose that as the optimized value.

---

2. Number of Layers Optimization

At the beginning, my model only had:

- **2 hidden layers + output layer**

But the F1-scores were low (train ~0.72, validation ~0.60), so I increased the depth by adding more layers.

I tested several deeper architectures and ended up using:

- **4 hidden layers + output layer** (total 5 fully connected layers)

This deeper model learned the data better and improved the F1-scores, so I kept this architecture.

---

3. Comparing Results

For each change:

- I checked the **loss curves**
- Measured the **training and validation F1-scores**
- Observed whether the model was **overfitting or improving**

By comparing these results, I selected the learning rate and number of layers that gave the most stable and improved performance.

Final Optimized Hyperparameters

- **Learning rate:** 0.0005
- **Number of layers:** 5 fully connected layers

6.3 What are the optimization results (show the numerical values and state how you improved them)

After optimizing the hyperparameters, I observed clear improvements in the model's performance. At first, I used a learning rate of 0.001, but this caused the validation loss to be unstable and the F1-scores to stay relatively low (around 0.72 for training and 0.60 for validation). When I lowered the learning rate to 0.0005, the training became smoother, the validation loss fluctuated less, and the F1-scores improved. I also optimized the number of layers. The original model with only two hidden layers did not have enough capacity to learn well from the high-dimensional image data, resulting in low F1-scores. After increasing the network to four hidden layers (five layers in total), the model learned more complex patterns, and both the training and validation F1-scores increased noticeably. Overall, adjusting the learning rate and expanding the number of layers improved the model's stability and classification performance, even though some overfitting still remained due to the limitations of using a fully connected DNN for image classification.

7. Discuss the Performance of Your Model:

7.1 What is the metric selected to evaluate your model?

The main evaluation metric I selected for my model is the **F1-score (macro F1)**. This metric was required by the instructor for this project and is more suitable than accuracy for multi-class classification, especially when the classes may not be perfectly balanced. The F1-score considers both **precision** and **recall**, which gives a better understanding of how well the model is correctly identifying each class rather than just counting how many predictions are correct.

7.2 What is the best metric value obtained?

The best evaluation metric value I obtained in this project was a validation F1-score of around 0.70. The training F1-score was higher (around 0.90–0.85) depending on the number of epochs), but the validation F1-score of approximately 0.70 is the most important because it shows how well the model performs on unseen data.

7.3 What kind of error(s) do you have in your model? Discuss your errors, state what they mean.

The main type of error in my model is **misclassification**, meaning the network sometimes predicts the wrong waste category for an image. This is expected because the validation F1-score is lower than the training F1-score, which shows that the model does not generalize perfectly to unseen data. One of the biggest issues is **overfitting**: the training loss keeps going down, but the validation loss becomes unstable and increases after some epochs. This means the model is learning details from the training data too strongly and not learning general patterns that work on new images.

From the confusion matrix, I can also see that some classes are harder for the model to separate. For example, the model may confuse “plastic” with “paper” or “metal” with “glass” because these materials can look similar in certain images. Since I used a fully connected DNN, the model does not capture spatial patterns the same way a CNN would, which also contributes to these errors. Overall, the errors show that while the model learns the training set well, it still struggles to correctly classify all categories when the images come from outside the training dataset.

7.4 Due to the errors stated above, what shall be done to improve your model's performance?

Based on the errors I observed, there are several things that can be done to improve the model's performance. First, the model is clearly overfitting, so one solution is to use **more data** or apply stronger **data augmentation** (such as random rotations, flips, and color changes) to make the model see more variations of the images. This can help the network generalize better to new images.

Another improvement is to **increase the regularization**, for example by using a higher dropout rate or adding weight decay in the optimizer. These techniques can help reduce overfitting. I could also try different learning rates or use a learning rate scheduler to stabilize the training and reduce the spikes in validation loss.

Since a fully connected DNN is not ideal for images, the biggest improvement would be to use a **Convolutional Neural Network (CNN)** in the future. CNNs are much better at capturing spatial patterns and usually give much higher accuracy for image classification tasks. However, because the project requirement was to use a DNN, I focused on optimizing that architecture as much as possible.

Overall, to improve performance: use more data or augmentation, tune regularization, adjust the learning rate, or switch to CNNs in future work.

7.5 Which techniques have you applied to improve your model's performance?

To improve the performance of my DNN model, I applied several techniques during training. First, I increased the **number of layers** in the network. I started with only two hidden layers, but the F1-score was low, so I expanded the architecture to four hidden layers (five layers total). This gave the model more learning capacity and helped it understand the high-dimensional image data better.

I also experimented with the **learning rate**. At the beginning I used 0.001, but the validation loss was unstable, so I tested smaller learning rates like 0.0005 and 0.0001. The value 0.0005 gave smoother and more stable results, so I used it as the optimized learning rate.

To reduce overfitting, I used **Dropout** with a rate of 0.5 after each hidden layer. Dropout randomly turns off some neurons during training, which forces the model to learn more general features instead of memorizing the data. I also

used **Batch Normalization** in each hidden layer to help the model train more smoothly.
Together, these techniques—adding more layers, adjusting the learning rate, using dropout, and using batch normalization—helped improve the model's training and validation performance.

7.6 Do you have overfitting or underfitting/overfitting in your model? Which processes did you apply to prevent underfitting/overfitting?

In my model, the main issue I faced was **overfitting**. This was clear because the training loss kept going down steadily, while the validation loss started to increase and fluctuate after the first few epochs. Also, the training F1-score was much higher than the validation F1-score, which means the model was learning the training data too well and not generalizing to new images.
To reduce overfitting, I applied several techniques. First, I used **Dropout** with a rate of 0.5 in every hidden layer to force the network to learn more general features instead of memorizing the training data. I also used **Batch Normalization** to help stabilize training and prevent the model from becoming too sensitive to small changes in the inputs. Another method I used was trying a **smaller learning rate**, which made the training smoother and reduced the spikes in the validation loss. I also increased the **number of layers** in the model to help it learn the data better without underfitting.
Even with these improvements, some overfitting still remained, which is expected for a fully connected DNN working with flattened image data. However, the techniques I applied helped reduce it and improved the overall performance of the model.

8. Error analysis and misclassified examples

Please discuss the error analysis you performed in your project.
To better understand the performance of my model, I analyzed the misclassified examples using the confusion matrix and individual test predictions. The confusion matrix helped me see which classes the model predicted correctly and which classes it confused with each other. I noticed that the model sometimes misclassified certain waste types, especially when the images looked visually similar. For example, some **plastic** images were predicted as **paper**, and some **metal** items were confused with **glass**. These mistakes usually happen in images where the materials share similar color or texture, making it difficult for a fully connected DNN to separate them.
Another part of my error analysis was checking the difference between the training and validation results. The training F1-score was much higher than the validation F1-score, and the validation loss was unstable, which showed clear **overfitting**. This means the model learned the training images very well but struggled to generalize to new data.
When I tested the model on images downloaded from the internet using my prediction function, I saw that some predictions were correct, but others were wrong, especially if the lighting, background, or angle of the image was very different from the training dataset.
Overall, the errors shown by the confusion matrix and the misclassified images indicate that the model can recognize many patterns correctly, but it still has difficulty with images that differ from the training conditions or have overlapping visual features. This is expected because a fully connected DNN does not capture spatial relationships like a CNN, which affects its ability to distinguish similar-looking waste categories.

9. Deep Learning Strategies

9.1 What are the Deep Learning Strategies you performed in this project?

In this project, I applied several Deep Learning strategies to build, train, and improve my waste-classification model. The main strategy was designing a **fully connected Deep Neural Network (DNN)** from scratch using PyTorch, since the project required using a DNN instead of a Convolutional Neural Network (CNN). I also used **Batch Normalization** to stabilize training and **Dropout** to reduce overfitting in every hidden layer.

Another strategy I used was **hyperparameter tuning**, where I experimented with different learning rates (0.001, 0.0005, 0.0001) and different numbers of layers to find a better-performing architecture. I also followed the general deep learning workflow from the course labs: loading and preprocessing the data, splitting it into training and testing sets, creating the model, training the model using an optimizer (Adam), evaluating it with F1-score, and running predictions on unseen images. Finally, I used **loss and F1-score curves**, as well as the confusion matrix, to analyze the model's behavior and understand where improvements were needed.

9.2 If they exist, why did you select these strategies to apply?

I selected these deep learning strategies because they were the most suitable for the project requirements and helped improve the model's performance. The main strategy, building a fully connected DNN, was chosen because the instructor specifically asked us to use a DNN architecture and not a CNN. Since DNNs usually struggle with image data, I added **Batch Normalization** and **Dropout** to help the model train more smoothly and reduce overfitting.

I also performed **hyperparameter tuning** (changing the learning rate and number of layers) because my initial results showed low F1-scores and unstable validation loss. Testing different learning rates helped me find a more stable training setup, and increasing the number of layers gave the model more capacity to learn the high-dimensional image features. I followed the deep learning workflow from the course labs because it helped organize the project and made the training and evaluation steps clearer.

Overall, these strategies were chosen to improve training stability, reduce overfitting, and increase the model's ability to classify the four waste categories more accurately, while still staying within the project requirements.

10. Discussion and Conclusion

Discuss your results and draw conclusions from your work. State what you learned in this project.

In this project, I built a fully connected Deep Neural Network (DNN) using PyTorch to classify waste images into four categories: glass, metal, paper, and plastic. Even though DNNs are not the ideal choice for image classification, I followed the project requirement and tried to get the best performance possible within these limitations. After preparing the dataset, designing the model, and training it with different hyperparameters, the best validation F1-score I achieved was around **0.62**, while the training F1-score was higher. This showed that the model was able to learn the training data well but had difficulty generalizing to new images. The loss curves and confusion matrix confirmed this behavior, with clear signs of overfitting.

To improve the performance, I experimented with different learning rates (0.001, 0.0005, 0.0001) and different numbers of layers. Increasing the number of layers from two hidden layers to four hidden layers improved the model's ability to capture more complex features, and tuning the learning rate made the training more stable. I also used dropout and batch normalization to reduce overfitting and help the model train more effectively. Although these techniques improved the results, DNNs still have natural limitations with image data, especially when spatial information is lost due to flattening.

Overall, this project helped me understand the full deep learning pipeline more deeply — from data loading and preprocessing to building custom architectures, tuning hyperparameters, analyzing loss curves, and evaluating performance with F1-score and confusion matrices. I also learned how challenging image classification can be when using a DNN instead of a CNN. The experience improved my skills in PyTorch, model evaluation, and debugging training issues. Despite the model's limitations, the project successfully met the requirements and gave me practical experience in designing and training a deep learning model from scratch.