Module 3 Assignment

AI Solution Assignment

Building an End-to-End Machine Learning Pipeline

Mohammed Farrukh Ali Khan

Master of Professional Studies Analytics

Northeastern University

EAI6020 21176 AI Systems Technology

Prof. Mohammad Islam

03-16-2025

WEATHER CLASSIFICATION USING DEEP LEARNING

INTRODUCTION

This project focuses on developing a deep learning pipeline for classifying weather conditions from images. The dataset consists of 11 weather categories: ['dew', 'fogsmog', 'frost', 'glaze', 'hail', 'lightning', 'rain', 'rainbow', 'rime', 'sandstorm', 'snow']. The pipeline includes data preprocessing, model training, evaluation, and deployment using *FastAPI*. The objective is to create an accurate and efficient classification model while addressing common challenges in deep learning, such as overfitting and data imbalance.

Why did I choose this topic?

Automating weather classification using AI can assist meteorologists and researchers by quickly analyzing weather conditions from images. This reduces manual workload and enables real-time weather monitoring.

DATA PREPROCESSING

The dataset was chosen from Kaggle (Bhathena, n.d.) which was organized into 11 folders, each representing a weather condition. Each folder contains images corresponding to that condition. The dataset was loaded using *PyTorch's 'ImageFolder'* class.

ImageFolder allows direct loading of images while maintaining class labels based on folder names, simplifying the data pipeline.

Exploratory Data Analysis (EDA): Performing EDA ensures that the dataset is well-structured and balanced. Identifying issues early helps in making informed preprocessing decisions

- The dataset was inspected to ensure proper organization.
- Sample images from each class were displayed to verify the data quality (Refer figure 1).

Data Augmentation and Normalization

Data augmentation is applied only to the training set to improve generalization, while the validation and test sets remain unchanged to reflect real-world performance. To improve model generalization, the following transformations were applied to the training data:

- Random resized cropping
- Random horizontal flipping
- Random rotation
- Normalization using mean [0.5, 0.5, 0.5] and standard deviation [0.5, 0.5, 0.5].

For validation and testing, only resizing and normalization were applied.

Dataset Splitting

The dataset was split into:

- Training set: 70%
- Validation set: 15%
- Test set: 15%

A 70-15-15 split ensures that the model has enough data for learning while maintaining a sufficient validation set for tuning and a test set for unbiased performance evaluation.

MODEL BUILDING

Model Architecture: A pre-trained ResNet-18 model was used as the base architecture. The final fully connected layer was replaced to accommodate the 11 output classes.

Training: The model was trained using the following setup:

- Loss Function: Cross-Entropy Loss
- Optimizer: Adam with a learning rate of 0.001
- Epochs: 10
- Batch Size: 32

Training Process: The model was trained for 10 epochs, and the training/validation loss and accuracy were logged. Early stopping was implemented to prevent overfitting.

Rationale for choosing the models: ResNet-18 is computationally efficient while providing high accuracy. Its residual connections prevent vanishing gradients, allowing deep networks to train effectively. Adam combines momentum and adaptive learning rate techniques, making it efficient for deep learning tasks with complex data distributions

Handling Class Imbalance: Some weather categories had significantly fewer images than others (e.g., **rainbow** had fewer images compared to **dew**). To prevent the model from favoring dominant classes, ***WeightedRandomSampler*** was used during training to balance data sampling.

Why balance the dataset?
Without balancing, the model might predict dominant classes more often, leading to poor generalization. Weighted sampling ensures fair training across all classes.

Why early stopping?
Early stopping monitors validation loss and halts training when it stops improving, preventing unnecessary overfitting and reducing training time.

MODEL EVALUATION

Evaluation Metrics: The model was evaluated on the test set using accuracy as the primary metric.

Results:

- Test Accuracy: 85.83% (Refer figure 3)
- Training/Validation Metrics:
- Training accuracy improved from 60.38% to 85.55%.
- Validation accuracy improved from 58.31% to 83.87%.

Visualizations (Refer figure 2)

- Training/Validation Loss and Accuracy: Plots were generated to visualize the model's performance over epochs.
- Sample Predictions: A few test images were displayed with their predicted and actual labels to demonstrate the model's performance (Refer figure 4).

The reason I used *FastAPI* is because it is faster, supports asynchronous requests, and is more efficient for real-time AI model inference. Also, I have previous experience with *FastAPI* which made it easier to work with.

Confusion Matrix Analysis

The confusion matrix (Refer figure 5) provides a detailed breakdown of the model's classification performance for each weather category. Key observations include:

- Strong diagonal values indicate that most predictions are correct. For example, "rime" has 141 correctly classified instances, while "fog/smog" has 118 correct classifications.
- Misclassifications occur in some classes, such as "glaze" being confused with "frost" (11 instances) and "sandstorm" being misclassified as "fog/smog" (16 instances).
- Classes with similar visual features, such as "glaze" and "frost" or "rain" and "hail," show some degree of confusion.
- Underrepresented classes like "rainbow" (40 correct predictions) may have lower accuracy due to fewer training examples.

MODEL DEPLOYMENT

Saving the Model: The trained model was saved as '*weather_modelF.pth*' for future use.

API Development: A *FastAPI* application was created to serve the model for predictions. The API has a single endpoint:

- Endpoint: `/predict`
- Method: POST
- Input: An image file
- Output: Predicted weather class

Testing the API: The API was tested using

- *cURL*: Command-line tool for sending POST requests.
- Postman: GUI tool for API testing.
- HTML Form: A simple web interface for uploading images and viewing predictions.

CHALLENGES AND SOLUTIONS

Challenge	Solution
Overfitting	Applied data augmentation & early stopping
Class Imbalance	Used WeightedRandomSampler to ensure fair training across all classes.
API Errors	Ensured correct request formatting and tested with different tools.

Table 1: Challenges along with their solutions

INSTRUCTIONS FOR RUNNING THE CODE

Running the Code (Optional)

1. Install the required dependencies:

```
pip install fastapi uvicorn torch torchvision pillow
```

2. Download the dataset and organize it into folders.

3. Run the training script (Optional, as we already have the trained model file):

```
python train.py
```

4. Save the trained model as '*weather_modelF.pth'*.

There is no need for performing the above steps as it has already been done and the trained model file has been shared. You can continue with the below steps for testing the model.

RUNNING THE API

1. Start the FastAPI server:

```
python app.py
```

2. Access the API at 'http://localhost:8000/'.

3. Use the '/predict' endpoint to classify images:

```
cURL:
  curl -X POST -F "file=@path_to_image.jpg" http://localhost:8000/predict
```

```
Postman:
  Set the request type to POST.
  Enter the URL: 'http://localhost:8000/predict'.
  Go to the 'Body' tab, select 'form-data', & upload an image file under the key 'file'.
```

```
HTML Form (optional):
  Open `index.html` in your browser.
  Upload an image and click 'Predict'.
```

Figure 6 shows the output from using cURL from the command prompt while figure 7 shows the output using the Postman method. Both methods work and give a True predicted value.

Figure 8 shows output from index.html method, which you can access by going to the URL http://localhost:8000 after running the command app.py

CONCLUSION

The weather classification pipeline was successfully implemented, achieving a test accuracy of 85.83%. The model was deployed using *FastAPI*, and the API was tested using multiple tools. Future work could include hyperparameter tuning, experimenting with deeper architectures, and deploying the API to a cloud platform.

Future Improvements

- Hyperparameter tuning to further improve accuracy.

- Testing with deeper architectures like *ResNet-50* or *EfficientNet*.
- Deploying the API to a cloud platform for wider accessibility.

REFERENCES

He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 770-778).

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., ... & Chintala, S. (2019). PyTorch: An imperative style, high-performance deep learning library. Advances in neural information processing systems, 32.

Bhathena, J. (n.d.). *Weather Dataset*. Kaggle. Retrieved from https://www.kaggle.com/datasets/jehanbhathena/weather-dataset

FastAPI Documentation. (n.d.). Retrieved from https://fastapi.tiangolo.com/

PyTorch Documentation. (n.d.). Retrieved from https://pytorch.org/docs/stable/index.html

Khan, F. (n.d.). *EAI6020 Weather Classifier* https://github.com/farrukh-ak/EAI6020_WeatherClassifier

APPENDIX

Figure 1: Sample Data after loading

Figure 2: Line charts – Training & Validation Loss and Accuracy

```
# Step 8: Model Evaluation
test_loss, test_accuracy = evaluate_model(trained_model, test_loader, criterion)
print(f"Test Loss: {test_loss:.4f}, Test Accuracy: {test_accuracy:.2f}%")

Test Loss: 0.4360, Test Accuracy: 85.83%
```

Figure 3: Output metrics showing Loss & Accuracy for test data

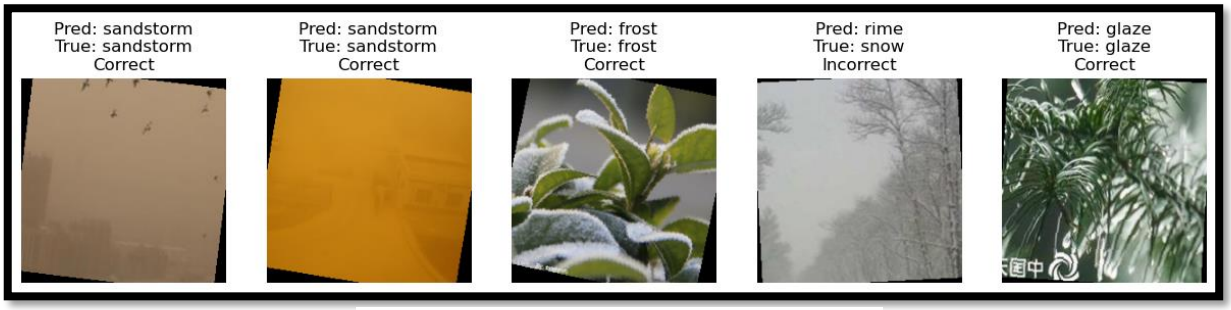

Figure 4: Predictions showing predicted & true values

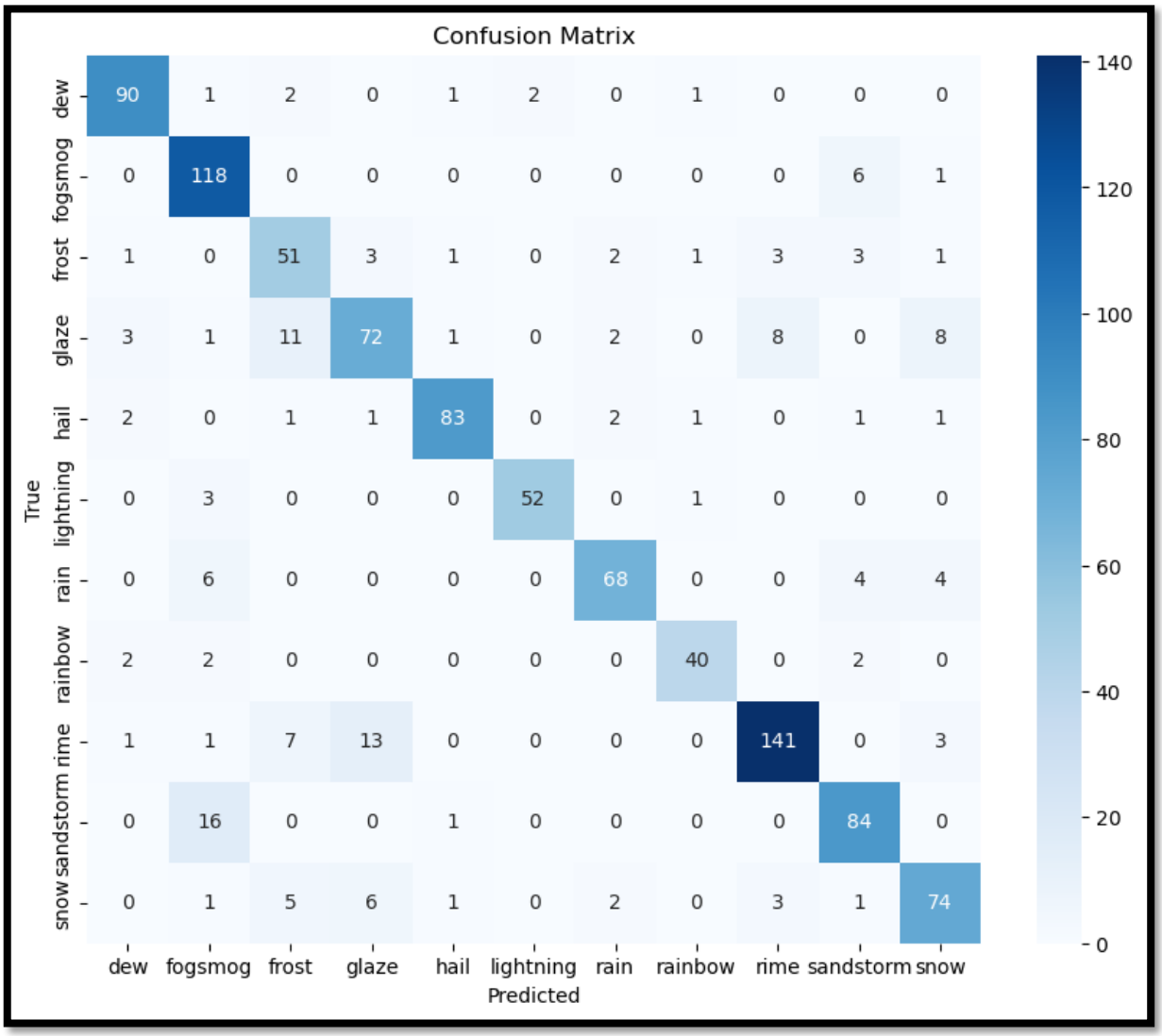

Figure 5: Confusion Matrix for the model

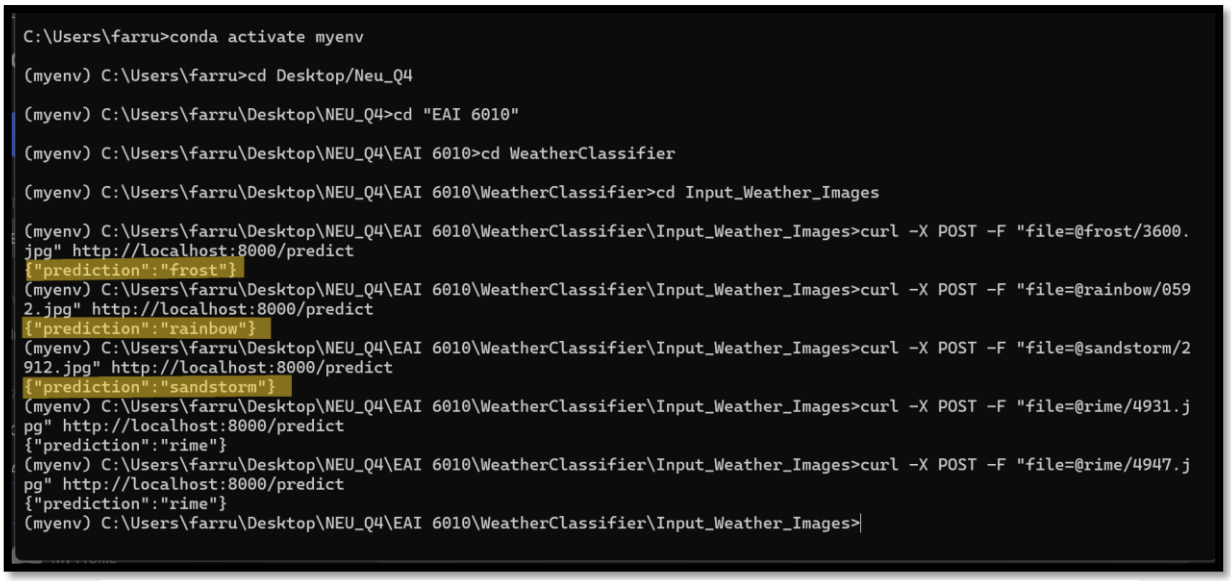

Figure 6: Output using postman. The predicted class is shown as a json block. Different classes were tested

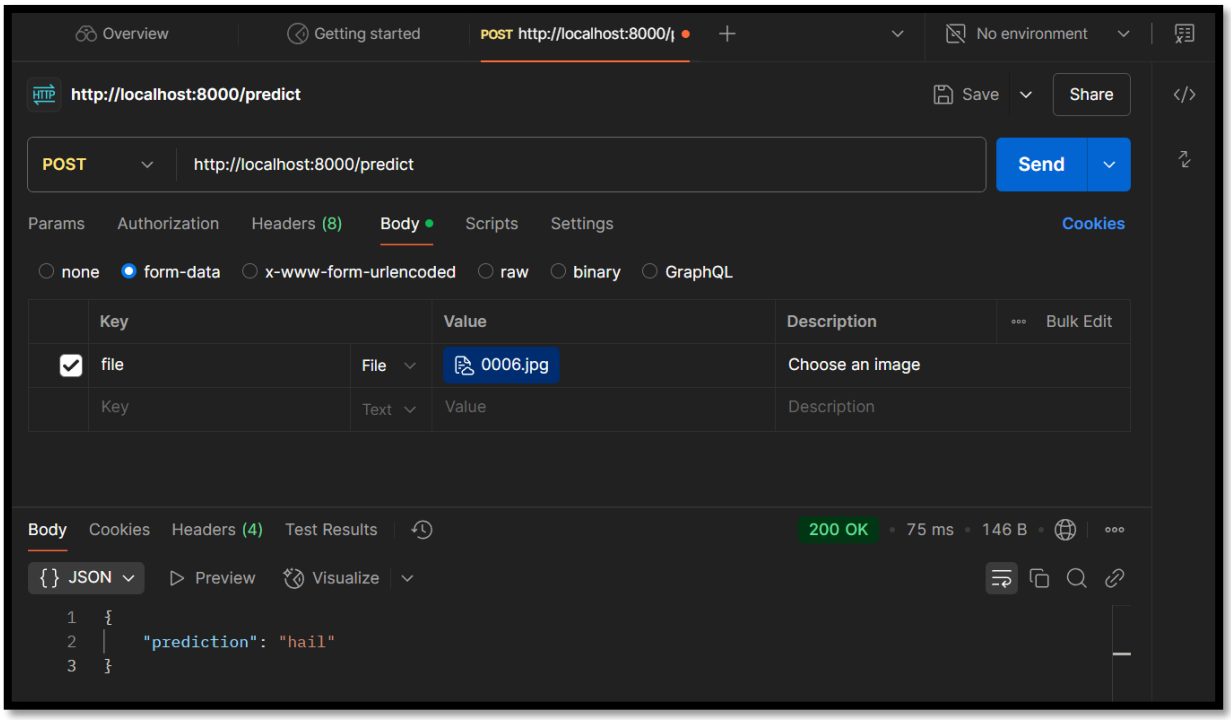

Figure 7: Output from Postman app

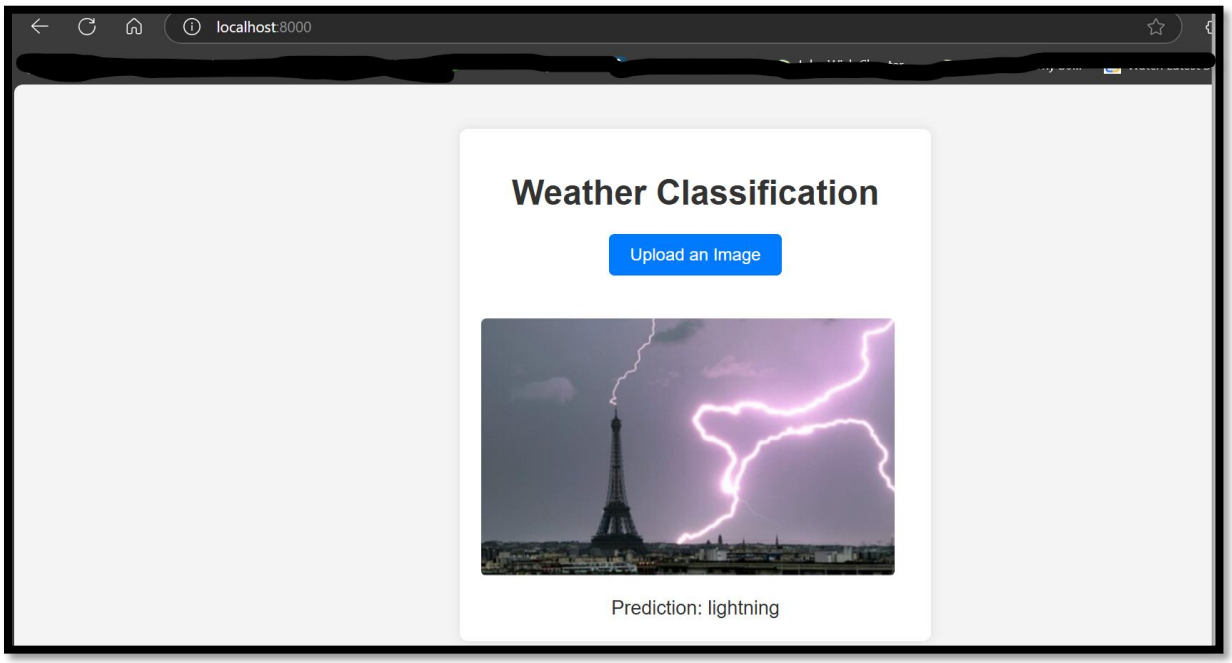

Figure 8: Output from index.html