Technical University of Cluj-Napoca

Faculty of Electronics, Telecommunications and Information Technology

Master's Program

Artificial Intelligence and Signal Processing in Electronics and Telecommunications

Network Intrusion Detection using Flow-to-Image Conversion and Deep Learning Classifiers

Student: Melinda-Henrietta Incze
Supervising Professor: Conf.dr.ing. Daniel Zinca

2026

Table of Contents

Abstract

This project investigates multi-class IoT network traffic classification using a PCAP-to-image approach. Raw packet captures are segmented into fixed time windows and encoded as grayscale images using a two-dimensional time–packet size histogram representation. The resulting image dataset is split into training, validation, and test sets and used to fine-tune a ResNet18 model via transfer learning. Experiments on a CICIoT2023 subset with five classes (benign traffic and four attack types) evaluate the impact of key training and encoding settings. The best performance reached 0.9867 test accuracy, while the highest overall class separability achieved an F1-score of 0.9933 with ROC–AUC of 1.0000, together with a minimum test loss of 0.0190.

1. Introduction

1.1 Motivation

The Internet is widely used in homes, industry, and healthcare, but many devices have limited security and are not updated regularly. Because of this, attackers can compromise devices and generate malicious network traffic such as flooding or spoofing. Detecting these attacks early is important for network monitoring and protection.

Many intrusion detection methods use handcrafted features extracted from network flows. These methods can work well, but they depend on feature design and may not generalize to different traffic patterns. In this project, I use a simpler idea: convert raw PCAP traffic into fixed-size images that represent packet behavior over time. This makes it possible to use convolutional neural networks (CNNs), which are strong at learning patterns from images.

1.2 Goals and Contributions

The goal of this project is to build and test an end-to-end pipeline for multi-class traffic classification in an IoT scenario using raw packet captures.

The main contributions are:

- A PCAP-to-image conversion method based on fixed time windows and a 2D time–packet length histogram.
- A multi-class classifier based on ResNet18 with transfer learning.
- An experimental study that varies key parameters (number of images, window size, image size, learning rate, epochs) and reports standard metrics.
- Evaluation using cross-entropy loss, accuracy, macro-F1, ROC–AUC (one-vs-rest), and confusion matrices.

1.3 Paper Organization

Section 2 presents the background: the attack types, the image encoding method, and the evaluation metrics. Section 3 explains the implementation, including the dataset setup (CICIoT2023), preprocessing, image generation, and model training/testing. Section 4 presents the experimental setup and results. Section 5 discusses limitations and future work, and Section 6 concludes the paper.

2. Background

2.1 Threat Model and Attack Types

This project assumes an IoT network where an attacker can generate malicious traffic toward victim devices or services.

The goal of the attacker is either to disrupt availability (denial of service) or to manipulate normal network behavior (spoofing). In this setting, traffic is captured as PCAP files and each capture is treated as one labeled class.

In the experiments, I focus on the following traffic types:

- **Benign traffic:** normal communication between IoT devices and services.
- **DoS / DDoS flooding attacks:** high-rate traffic that aims to exhaust network or server resources. In practice, floods create strong patterns such as bursts of many packets in a short time period. Typical variants include **SYN Flood**, **UDP Flood**, and **HTTP Flood**, which differ by protocol and packet behavior.
- **DNS spoofing:** an attacker forges DNS responses to redirect a victim to incorrect destinations. This can change the distribution and timing of DNS-related packets compared to normal traffic.

These attack families are relevant because they can produce measurable changes in packet timing and packet size distribution, which are exactly the signals used by the image encoding method.

2.2 Traffic-to-Image Encoding

Raw network traffic is a sequence of packets with timestamps and different packet sizes. To use a CNN, the input must have a fixed size, so the traffic is transformed into images using the following steps:

1. **Windowing:** packets are grouped into fixed time windows
2. **Feature extraction:** for each packet, two values are kept:
   - ✓ time offset inside the window (relative to the first packet)
   - ✓ packet length (clipped to a maximum)
3. **Binning into a 2D grid:** the window is converted into an n×n matrix:
   - ✓ the x-axis represents time bins inside the window
   - ✓ the y-axis represents packet length bins

Each packet increments one cell, so the matrix becomes a 2D histogram of activity.

4. **Normalization:** values are normalized to the range [0, 255] to form a grayscale image

The result is one image per time window, where brighter pixels indicate more packets with similar timing and size. This representation is intended to capture both burstiness (common in floods) and changes in packet size patterns.

2.3 Model and Evaluation Metrics

After converting traffic segments into images, the problem becomes a standard multi-class image classification task. A convolutional neural network (CNN) is suitable because it can learn visual patterns such as dense regions, repeated structures, and texture-like differences that correspond to traffic behavior.

In this project, the classifier is based on **ResNet18**, a well-known CNN architecture. ResNet introduces *residual connections* (skip connections), which help the network train more reliably by allowing gradients to flow through deeper layers. This reduces problems such as vanishing gradients and often leads to better performance than a plain CNN with the same depth.

A common approach is to use **transfer learning**, where the model starts from a CNN pretrained on a large image dataset and is then adapted to the target classes. This is useful because it reduces the amount of training data needed and often improves convergence compared to training from scratch.

During training, the model outputs a score (logit) for each class. These scores are converted into probabilities, and the model is trained to assign high probability to the correct class. The **loss function** is a single numeric value that measures how wrong the model's predictions are. A smaller loss means the predictions are closer to the true labels. For multi-class classification, the standard choice is **cross-entropy loss**, which penalizes confident wrong predictions more strongly than uncertain ones. Minimizing cross-entropy typically improves classification performance.

To reduce the loss, training uses an **optimizer**, which updates the model's parameters (weights) step by step. In each step, the optimizer uses gradients (computed by backpropagation) to decide how to change the weights to lower the loss. A widely used optimizer is **Adam**, because it is stable and adapts the update size for each parameter automatically.

A key hyperparameter is the **learning rate (LR)**. The learning rate controls the size of each update:

- If the learning rate is **too small**, training progresses very slowly and may get stuck with suboptimal performance.
- If the learning rate is **too large**, training can become unstable and the loss may oscillate or diverge.

In practice, learning rate and the number of training epochs are closely related: longer training can work with smaller learning rates, while larger learning rates often require careful tuning to avoid instability.

Performance evaluation in classification tasks requires metrics that reflect both correctness and class distribution.

- **Accuracy** measures the ratio of correct predictions to total samples but is often misleading in imbalanced datasets
- **Precision and Recall** capture the trade-off between false positives and false negatives
  Precision answers: *"When the model predicts positive, how often is it correct?"*

$$Precision = \frac{True\ Positives}{True\ Positives + False\ Positives} \quad (1)$$

  Recall answers: *"Out of all actual positives, how many did the model find?"*

$$Recall = \frac{True\ Positives}{True\ Positives + False\ Negatives} \quad (2)$$

- **F1-score** is the harmonic mean of precision and recall, used for binary classification
- **ROC-AUC** (Receiver Operating Characteristic - Area Under the Curve) evaluates ranking quality across thresholds

ROC-AUC measures how well a model separates positive from negative samples across all decision thresholds. It is based on the ROC curve, which relates the true positive rate to the false positive rate. A higher ROC-AUC value indicates a better ability to rank positive instances above negative ones. It is threshold-independent and commonly used in binary and multi-label classification via one-vs-all strategies.

For multi-label classification, metrics are computed per label and aggregated either macro (equal weight to all labels) or micro (weight by label frequency) to capture performance across the label set.

- **Confusion matrix:** shows which classes are most often confused and helps identify systematic error patterns.

3. Methodology

3.1 System Overview

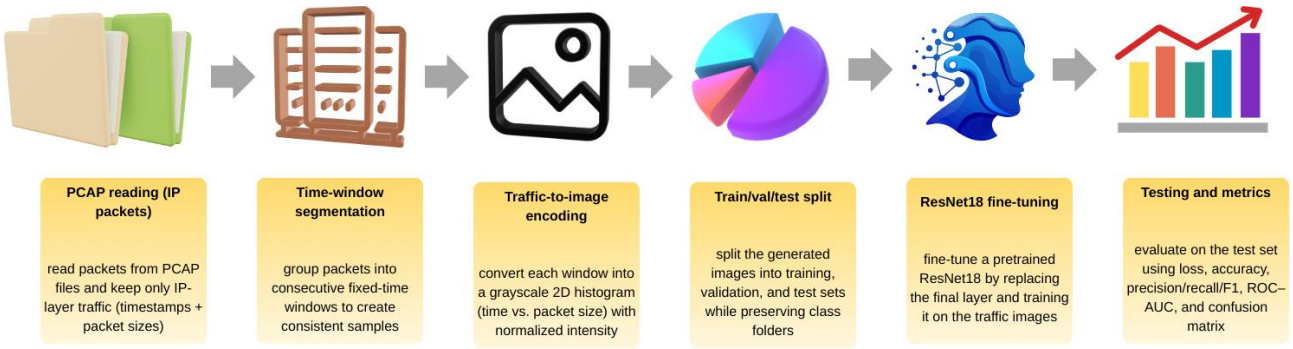

Figure 1. Code Flow

The proposed system follows a simple end-to-end pipeline as in *Figure 1*:

1. **Input acquisition:** network traffic is collected as PCAP files and each file is associated with a traffic class (benign or a specific attack type).
2. **Segmentation:** the packet stream is split into consecutive short time segments. Each segment is processed independently.
3. **Image encoding:** each time segment is converted into a fixed-size grayscale image that summarizes packet activity.
4. **Dataset preparation:** images are saved into class folders and split into training, validation, and test sets.
5. **Model training:** a pretrained CNN (ResNet18) is fine-tuned on the training images and selected using validation performance.
6. **Testing:** the best model is evaluated on the held-out test set using standard classification metrics.

3.2 Dataset

CICIoT2023 (IoT Dataset 2023) published by the Canadian Institute for Cybersecurity (UNB) was collected in a realistic IoT lab environment and includes benign traffic and a wide range of attack behaviors executed by malicious IoT devices against IoT victims. Overall, it contains 33 attacks grouped into 7 categories (DDoS, DoS, Recon, Web-based, Brute Force, Spoofing, Mirai).

Because CICIoT2023 is very large, downloading and processing the full dataset would be difficult on a normal student computer. To keep the study manageable and reproducible, I used a smaller subset: 5 PCAP files and 3 merged CSV files. I chose this subset for two reasons. First, it is enough to build and test the full pipeline (PCAP → image encoding → CNN training) while keeping the multi-class problem simple and fast to run. Second, the merged CSV files give a feature-based view of the same traffic, which helps with basic checks (like class balance) and can be used later to compare the image-based approach with a classical machine learning baseline.

3.3 Image Generation

A PCAP file contains packets ordered in time. Instead of processing the entire capture at once, the traffic is divided into many short, consecutive time windows. You can imagine the PCAP as a long timeline, and the method cuts this timeline into equal slices (for example, every 2 seconds). All packets whose timestamps fall inside the same slice belong to the same window. Each window

becomes one training sample. This is important because it creates many fixed-size samples from a single PCAP and makes the input format consistent.

For each packet in a window, only two simple values are used:

- **Timestamp:** when the packet occurred (used to place it inside the window)
- **Packet length:** the size of the packet in bytes

Packets that do not contain an IP layer are ignored. This keeps the focus on IP-level behavior, which is more relevant for most network attacks and reduces noise from unrelated link-layer frames. Packet lengths are also limited to a reasonable maximum so that extremely large values do not dominate the representation.

After collecting the packets for one window, the window is converted into a fixed-size grayscale image. The image is built as a 2D histogram, meaning it counts how many packets fall into different ranges of time and size:

- **x-axis** represents time progression inside the window, from the start to the end
- **y-axis** represents packet length, from small packets to large packets

Each packet is mapped to exactly one cell (pixel) in this grid based on its time position and packet size. When multiple packets fall into the same cell, that cell's count increases. In the final image, higher counts become brighter pixels.

Some windows may contain only a few packets (for example, when traffic is almost idle). Very sparse windows usually do not contain enough information for learning and can add noise to the dataset. For this reason, windows that contain too few packets are discarded and not saved as images.

A simple way to imagine the encoding is to think of drawing on squared graph paper like in *Figure 2*:

- The sheet has a fixed grid (for example, 32 squares wide and 32 squares high)
- The **width** of the sheet represents the full time window (for example, 2 seconds)
- The **height** of the sheet represents packet sizes from small to large (up to about 1500 bytes)

Now, each packet becomes a **dot** placed on the sheet:

- left to right depending on when it happened inside the window
- bottom to top depending on how large the packet is

If there are not enough dots to draw, the sheet is thrown away. If many dots land in the same square, that square becomes "darker" or "brighter" depending on the intensity scale. Flood attacks usually produce many dots packed into a short time range, creating dense bright areas. Benign traffic often produces fewer dots that are more spread out.

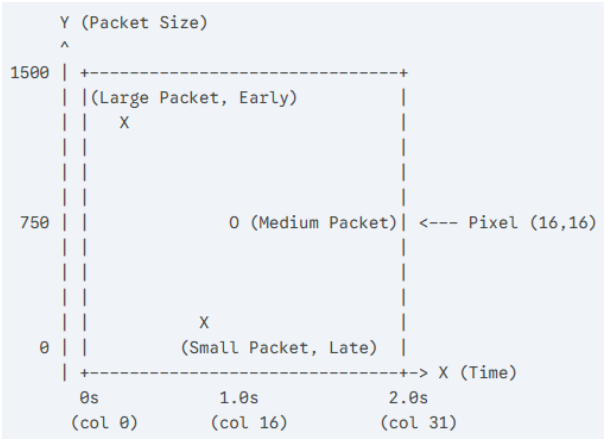

Figure 2. Conceptual Illustration of the Time–Size Grid Used for Image Generation

Traffic can be very uneven. In flood attacks, some cells may receive extremely high counts, while other cells remain almost empty. If the raw counts were used directly, a few cells could dominate the image and hide smaller but important patterns.

A logarithmic scaling compresses large values and makes both strong and weak activity visible. After that, the image is normalized so that all images use the same intensity range. This makes training more stable because the CNN always receives inputs with a comparable scale.

In the implementation, the PCAP-to-image conversion is handled by three main components: *iter_packets()*, *pcap_to_class_images()*, and *window_to_image()*.

First, *iter_packets()* streams packets from a PCAP file using *PcapReader*, keeps only IP packets (*IP in pkt*), and extracts for each packet its timestamp (*pkt.time*) and its length (*len(bytes(pkt))*), while clipping extreme lengths using *MAX_PKT_LEN = 1500*.

Next, *pcap_to_class_images()* groups packets into consecutive time windows controlled by *TIME_WINDOW_SEC = 2.0* using the variables *win_start*, *buf_t*, and *buf_l*; when a window ends, it is discarded if it contains fewer than *MIN_PKTS_PER_WINDOW = 10* packets, and otherwise it is converted into an image.

The actual image creation is done by *window_to_image()*, which maps packet times into x-bins and packet lengths into y-bins on a fixed *IMG_SIZE = 32* grid, incrementing a 2D histogram cell for each packet as in *Figure 3*.

Finally, the histogram is enhanced using *log1p* and normalization to obtain a stable grayscale intensity range (0-255), and the image is saved with PIL into a class-specific folder under *OUT_IMG_ROOT*, while *MAX_IMAGES_PER_PCAP* limits how many images are generated from each capture.

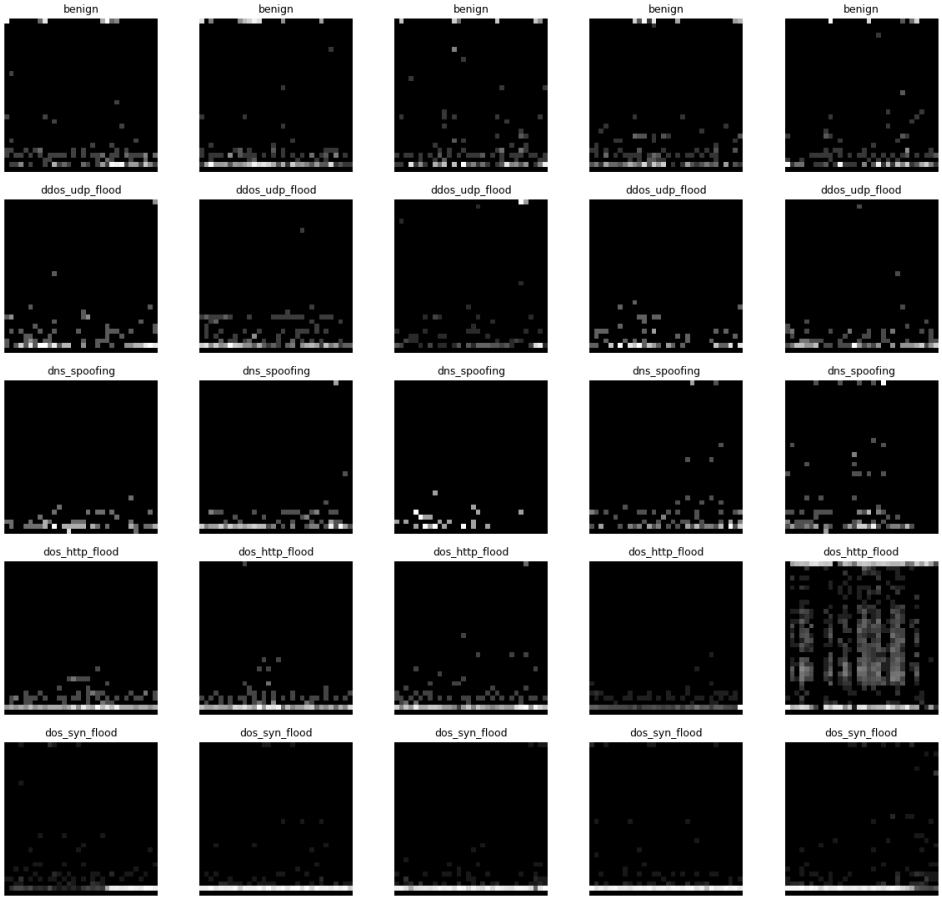

Figure 3. Visualization of Generated 32×32 Traffic Images Across Classes

3.4 Training and Testing

After generating the grayscale images, the dataset is prepared for supervised learning by creating a standard train/validation/test structure. In the code, the split is done per class folder using *TRAIN_RATIO = 0.70*, *VAL_RATIO = 0.15*, and *TEST_RATIO = 0.15*, with reproducibility controlled by *SEED*. The function *split_and_copy_class()* collects all image files, applies *random.shuffle()* to randomize them, then copies them into *SPLIT_ROOT/train/<class>*, *SPLIT_ROOT/val/<class>*, and *SPLIT_ROOT/test/<class>* using *shutil.copy2()*. This folder layout is required because the training code loads data with *datasets.ImageFolder*, which automatically assigns labels based on folder names and returns image–label pairs during training.

For the classifier, I chose *resnet18* with pretrained weights (*ResNet18_Weights.DEFAULT*) because it is a strong and widely used baseline for image classification and works well with transfer learning, especially when the dataset is not extremely large. In the implementation, the last layer is replaced using *model.fc = nn.Linear(model.fc.in_features, num_classes)* so the network outputs the correct number of classes for this project. Since ResNet expects RGB images and a larger input resolution, the images are converted and standardized using *transforms.Compose*: *transforms.Grayscale(num_output_channels=3)* converts the single-channel image to three channels, *transforms.Resize((224, 224))* matches the expected size, and *transforms.Normalize(mean=[...], std=[...])* applies ImageNet-style normalization. Data is fed to the network using *DataLoader* with *BATCH_SIZE* and *NUM_WORKERS*, and the model is trained on *device = torch.device("cuda" if torch.cuda.is_available() else "cpu")* for faster computation when a GPU is available.

Training is performed as a standard multi-class classification setup. The loss function is *nn.CrossEntropyLoss()* because the task is single-label multiclass prediction, and parameter updates are done with *torch.optim.Adam(model.parameters(), lr=LR)*. The training loop is implemented through *run_epoch()*, which runs one full pass over the dataset in either training mode (when an *optimizer* is provided) or evaluation mode (when *optimizer=None*).

During each epoch, the code computes the average loss and two main metrics: accuracy (via *accuracy_score*) and macro-F1 (via *f1_score(..., average="macro")*). The best model is selected using validation macro-F1: whenever validation macro-F1 improves, the weights are saved to *MODEL_PATH* using *torch.save()*, and metadata such as class names and hyperparameters are stored in *META_PATH* using *json.dumps(...)*. This is important because macro-F1 is more informative than accuracy when classes are not perfectly balanced.

Evaluation and testing follow the same logic but without gradient updates. After training, the best checkpoint is loaded using *torch.load(MODEL_PATH)* and *model.load_state_dict(state)*, then test predictions are collected in arrays (*all_preds*, *all_labels*) by applying *torch.argmax(logits, dim=1)* over the model outputs.

The performance is summarized using *classification_report(all_labels, all_preds, target_names=train_ds.classes)*, which provides per-class precision, recall, and F1, plus macro and weighted averages. For error analysis, a confusion matrix is computed and visualized using *ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=train_ds.classes)*.

Finally, the code includes a simple example-level test with *predict_image(img_path, topk=3)*: it loads a single image, applies the same evaluation preprocessing (*eval_tf*), computes probabilities using *torch.softmax(logits, dim=1)*, and prints the top predicted classes, allowing qualitative checking that the model behaves reasonably on individual samples.

4. Experiments and Results

4.1 Experimental Setup

The experiments follow a controlled design where a baseline configuration is defined and then key parameters are varied to study their impact on performance. The main goals are (i) to measure the overall classification quality on a held-out test set and (ii) to understand how preprocessing and training choices influence the results.

To ensure comparability across runs, the same model architecture (ResNet18 with transfer learning), data split strategy (train/validation/test), and evaluation procedure are used throughout. The ablation study includes ten runs, summarized in *Table 1*, with variations in: dataset size (maximum number of generated images per capture), time-window length, image resolution, minimum packet threshold per segment, learning rate, and training duration.

Table 1. Image and Training Parameters

Run	MAX_IMAGES _PER_PCAP	TIME_WINDOW _SEC	IMG_SIZE	MIN_PKTS _PER_WINDOW	EPOCHS	LR	BATCH
1	100	2.0	32	10	12	1e-4	32
2	100	2.0	32	10	20	1e-4	32
3	100	2.0	32	10	40	1e-4	32
4	100	2.0	32	10	40	3e-4	32
5	100	2.0	32	10	40	3e-5	32
6	100	2.0	32	10	40	1e-4	16
7	100	2.0	32	10	40	1e-4	64
8	100	4.0	32	10	40	3e-4	32
9	100	2.0	32	15	40	3e-4	32
10	100	2.0	64	10	40	3e-4	32
11	200	2.0	32	10	40	3e-4	32

4.2 Performance Results

Table 2. Comparative Results of Experimental Runs

Run	accuracy	loss	precision	recall	f1-score	ROC-AUC
1	0.9333	0.1378	0.9339	0.9333	0.9333	0.9942
2	0.9333	0.1378	0.9339	0.9333	0.9333	0.9942
3	0.9333	0.1761	0.9361	0.9333	0.9336	0.9931
4	0.9733	0.2090	0.9765	0.9733	0.9732	0.9898
5	0.9200	0.2156	0.9375	0.9200	0.9181	0.9922
6	0.9467	0.1405	0.9467	0.9467	0.9467	0.9940
7	0.9333	0.1633	0.9339	0.9333	0.9333	0.9947
8	0.9333	0.3356	0.9371	0.9333	0.9341	0.9904
9	0.9467	0.2075	0.9467	0.9467	0.9467	0.9918
10	0.9867	0.0403	0.9875	0.9867	0.9867	0.9991
11	0.9333	0.0190	0.9935	0.9933	0.9933	1.0000

The quantitative performance of all experimental runs is summarized in *Table 2*, which reports the main evaluation metrics on the test set. Based on these results, the best-performing configuration

is **Run 11**, as it achieves the highest overall performance across the reported metrics (notably the best F1-score and ROC–AUC).

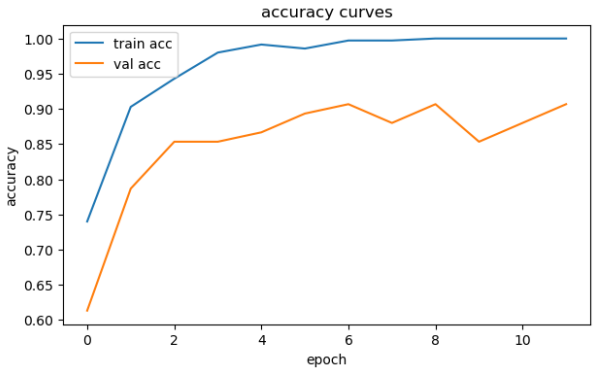

Figure 4. Accuracy curves (Run 1)

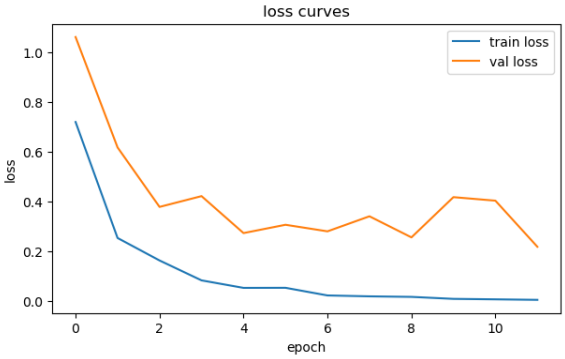

Figure 5. Loss curves (Run 1)

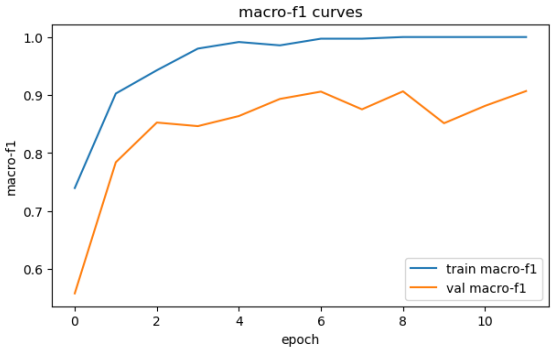

Figure 6. F1-score curves (Run 1)

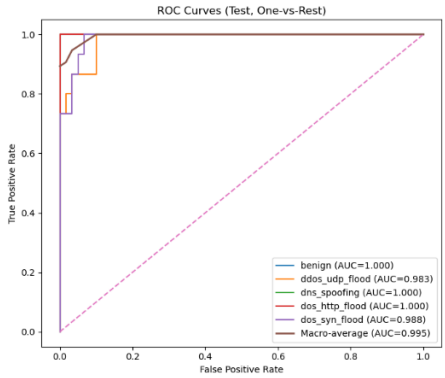

Figure 7. ROC curves (Run 1)

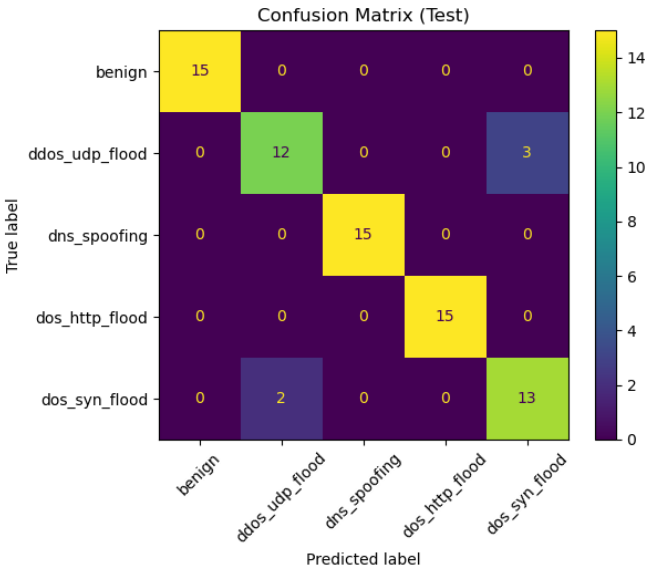

Figure 8. Confusion Matrix (Run 1)

```
sample image: E:\Project\flow2img\data\images\pcap_multiclass_split\test\benign\benign_000003.png
true class folder: benign
top predictions:
 - benign: 0.9994
 - ddos_udp_flood: 0.0004
 - dos_http_flood: 0.0002
 - dos_syn_flood: 0.0001
 - dns_spoofing: 0.0000
```

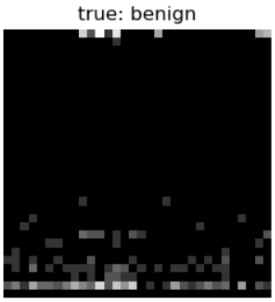

Figure 9. Prediction on a sample image from test folder (Run 1)

From *Figure 4* to *Figure 9*, the results show that the model learns the training data very quickly: training accuracy and F1 increase fast and reach very high values, while training loss keeps decreasing.

Validation performance improves mainly in the first epochs and then fluctuates below the training curves, which suggests some overfitting and limited generalization gains after the early stage. The confusion matrix indicates that most samples are classified correctly, with the main mistakes happening between similar flood classes (especially *ddos_udp_flood* and *dos_syn_flood*).

The single-image prediction example confirms that the model can produce confident and correct decisions on individual samples (high probability for the true class).

Finally, the ROC curves are close to the top-left corner and the AUC values are very high, meaning the model separates the classes well when using probability scores, not only hard labels.

4.3 Analysis

- Standard run for comparison (baseline): Run 1 is the reference (accuracy ≈ 0.93, F1 ≈ 0.93, ROC–AUC ≈ 0.99). Runs 1–3 show that simply training longer with the same learning rate did not improve results.

- Does performance drop drastically with few data? With 100 images/PCAP the model already reaches decent performance (~0.93), so it does not collapse, but it is sensitive to training settings.
- How much do we gain with more data (×2)? In this table we only tested 2× data (Run 11: 200 images/PCAP). With 2×, ROC–AUC and F1 become very high, suggesting better class separability, but accuracy does not increase here.
- Short/bursty attacks vs wider context: Increasing the time context to 4 seconds (Run 8) hurts performance (higher loss, no accuracy gain). This suggests the shorter window (2s) captures the burst patterns better in this setup.
- Convergence / stability (learning rate):
  - ✓ Too small LR (Run 5, 3e-5) reduces performance → training is likely too slow/under-trained.
  - ✓ Higher LR (Run 4, 3e-4) improves accuracy a lot, but loss is higher, meaning predictions may be less well-calibrated.
- Better minimum even if it takes longer: Longer training alone (Run 3 vs Run 1) did not help; improvement came mainly from a better LR and representation changes (e.g.Run 10).
- Overfitting: From the learning curves (train much higher than validation and validation loss oscillations), overfitting is present after the first epochs, so choosing the best checkpoint by validation score is important.
- Smaller, noisier steps generalize better (batch size): Run 6 (batch 16) performs better than the larger batch runs (especially batch 64), which supports the idea that smaller batches can improve generalization.
- Winning parameter combination: The strongest overall setup is *LR = 3e-4*, 40 epochs, plus improved encoding resolution (*IMG_SIZE = 64*, Run 10) or more data (200 images/PCAP, Run 11).

5. Future Work

5.1 Limitations

This study uses a reduced subset of CICIoT2023 and labels traffic at the PCAP level, which may limit generalization to more diverse or mixed real-world traffic.

The train/validation/test split is performed on generated windows, so samples from the same capture can be temporally similar, which may lead to optimistic results.

In addition, the image encoding uses only timing and packet-size information attacks that differ mainly in higher-level protocol behavior may be harder to separate with this representation.

5.2 Directions for Extension

Future work can extend the evaluation to more captures and attack families from CICIoT2023 and use stricter split strategies (e.g. split by capture files or by non-overlapping time blocks) to reduce temporal leakage.

The encoding could be enriched with additional signals such as packet direction, inter-arrival times, or protocol-specific information.

Finally, performance could be compared against classical ML baselines using the provided CSV features and against alternative deep models (e.g. EfficientNet or Vision Transformers) to validate whether the chosen architecture is optimal for this task.

7. Conclusion

Overall, the experiments confirm that the proposed PCAP-to-image approach combined with a CNN can achieve strong multi-class traffic classification on the selected CICIoT2023 subset.

The model learns discriminative patterns quickly, and the high ROC–AUC values indicate good class separability.

However, the gap between training and validation performance shows that overfitting can appear, so careful model selection and controlled tuning are important for reliable results.

References

[1] E. C. P. Neto, S. Dadkhah, R. Ferreira, A. Zohourian, R. Lu, and A. A. Ghorbani, “CICIoT2023: A Real-Time Dataset and Benchmark for Large-Scale Attacks in IoT Environment,” *Sensors*, vol. 23, no. 13, Art. 5941, 2023, doi: 10.3390/s23135941.

[2] K. He, X. Zhang, S. Ren, and J. Sun, “Deep Residual Learning for Image Recognition,” in *Proc. IEEE CVPR*, 2016, pp. 770–778, doi: 10.1109/CVPR.2016.90.

[3] P. García-Teodoro, J. Díaz-Verdejo, G. Maciá-Fernández, and E. Vázquez, “Anomaly-based Network Intrusion Detection: Techniques, Systems and Challenges,” *Computers & Security*, vol. 28, no. 1–2, pp. 18–28, 2009, doi: 10.1016/j.cose.2008.08.003.

[4] A. L. Buczak and E. Guven, “A Survey of Data Mining and Machine Learning Methods for Cyber Security Intrusion Detection,” *IEEE Communications Surveys & Tutorials*, vol. 18, no. 2, pp. 1153–1176, 2016, doi: 10.1109/COMST.2015.2494502.

[5] T. Fawcett, “An Introduction to ROC Analysis,” *Pattern Recognition Letters*, vol. 27, no. 8, pp. 861–874, 2006, doi: 10.1016/j.patrec.2005.10.010.