

Cybersecurity: Robustness in Incremental Learning

Adversarial Attacks
Catastrophic Forgetting
Replay Mechanisms



DIRECTORY:

Dataset
Model
Result



DATASET



Dataset Overview

The CIC-DDoS2019 dataset is a highly reliable benchmark dataset that is widely used for detection and evaluation of distributed denial of service (DDoS) attacks. The dataset simulates a real network environment and provides a variety of network behavior samples by capturing normal traffic and attack traffic. The dataset is designed to provide strong data support for the training and testing of machine learning models.

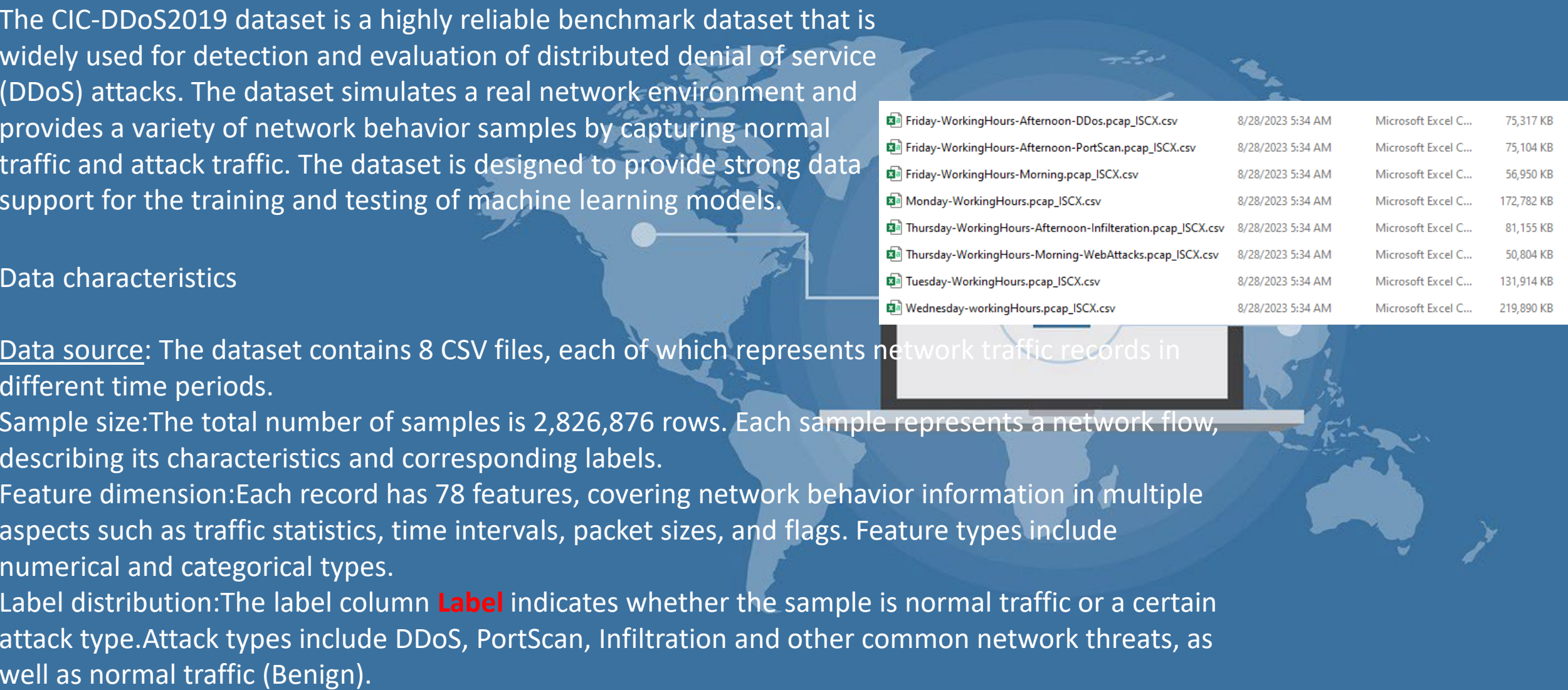
Data characteristics

Data source: The dataset contains 8 CSV files, each of which represents network traffic records in different time periods.

Sample size:The total number of samples is 2,826,876 rows. Each sample represents a network flow, describing its characteristics and corresponding labels.

Feature dimension:Each record has 78 features, covering network behavior information in multiple aspects such as traffic statistics, time intervals, packet sizes, and flags. Feature types include numerical and categorical types.

Label distribution:The label column **Label** indicates whether the sample is normal traffic or a certain attack type.Attack types include DDoS, PortScan, Infiltration and other common network threats, as well as normal traffic (Benign).



Friday-WorkingHours-Afternoon-DDos.pcap_ISCX.csv	8/28/2023 5:34 AM	Microsoft Excel C...	75,317 KB
Friday-WorkingHours-Afternoon-PortScan.pcap_ISCX.csv	8/28/2023 5:34 AM	Microsoft Excel C...	75,104 KB
Friday-WorkingHours-Morning.pcap_ISCX.csv	8/28/2023 5:34 AM	Microsoft Excel C...	56,950 KB
Monday-WorkingHours.pcap_ISCX.csv	8/28/2023 5:34 AM	Microsoft Excel C...	172,782 KB
Thursday-WorkingHours-Afternoon-Infiltration.pcap_ISCX.csv	8/28/2023 5:34 AM	Microsoft Excel C...	81,155 KB
Thursday-WorkingHours-Morning-WebAttacks.pcap_ISCX.csv	8/28/2023 5:34 AM	Microsoft Excel C...	50,804 KB
Tuesday-WorkingHours.pcap_ISCX.csv	8/28/2023 5:34 AM	Microsoft Excel C...	131,914 KB
Wednesday-workingHours.pcap_ISCX.csv	8/28/2023 5:34 AM	Microsoft Excel C...	219,890 KB

Data Processing Flow

Data integration and cleaning

- Merge all CSV files to generate a unified dataset.
- Delete columns containing only missing values and rows with any missing values to ensure data integrity.

```
dataframes = []
for path in file_paths:
    try:
        df = pd.read_csv(path)
        dataframes.append(df)
    except Exception as e:
        print(f"Error loading {path}: {e}")

# Combine all data into a single dataframe
combined_df = pd.concat(dataframes, ignore_index=True)
```

```
# Drop any columns with all NaN values and drop rows with any NaN values
cleaned_df = combined_df.dropna(axis=1, how='all') # Drop columns with all NaN
cleaned_df = cleaned_df.dropna() # Drop rows with any NaN values
```

```
# Remove leading and trailing whitespaces from column names
cleaned_df.columns = cleaned_df.columns.str.strip()
```

Feature and label separation

- Separate feature columns and label columns to provide clear boundaries for the input and output of subsequent machine learning models.

```
# Separate features and labels
features = cleaned_df.drop(columns=['Label'])
labels = cleaned_df['Label']
```

```
# Replace infinite values with NaN and then drop rows containing NaN values
features.replace([np.inf, -np.inf], np.nan, inplace=True)
features.dropna(inplace=True)
```

```
# Update labels to match the cleaned features
labels = labels[features.index]
```

Numerically encode label columns

- encoding Benign as 0 and attack types as other integer values.

```
# Encode categorical labels into numerical values
label_encoder = LabelEncoder()
encoded_labels = label_encoder.fit_transform(labels)
```

Standardization

- Use the z-score standardization method to normalize all feature columns to a standard distribution with a mean of 0 and a standard deviation of 1 to avoid the scale differences between features affecting model training.

```
# Standardize the feature columns to have zero mean and unit variance
scaler = StandardScaler()
scaled_features = scaler.fit_transform(features)
```

Dataset partitioning

- Use stratified sampling to divide the data into training set (60%), validation set (20%) and test set (20%) to ensure that each category is evenly distributed in each subset.

```
# Split data into training, validation, and test sets (60% train, 20% validation, 20% test)
X_train, X_temp, y_train, y_temp = train_test_split(scaled_features, encoded_labels, test_size=0.4, random_state=42)
X_val, X_test, y_val, y_test = train_test_split(X_temp, y_temp, test_size=0.5, random_state=42)
```


Dataset features and advantages

The dataset covers a variety of network threats, which enhances the generalization ability of the model in different scenarios.

The ratio of attack to normal traffic is reasonably distributed, making the dataset representative when simulating real-world scenarios.

The 78 features provided cover all aspects of network traffic, providing sufficient information for feature engineering and pattern learning of the model.

The structure of the dataset facilitates the integration of more traffic records and is suitable for incremental learning scenarios.

The dataset has been widely used by the community and supports cross-study results comparison.



MODEL



The model design in this study is aimed at detecting distributed denial of service attacks (DDoS). It adopts an incremental learning framework based on a fully connected neural network, combined with adversarial attack mitigation and catastrophic forgetting solution mechanisms. The model architecture is simple and efficient, designed to handle high-dimensional features and cope with complex challenges in dynamic data streams.



Data poisoning strategy

1. Feature Perturbation: Introducing random noise or systematic perturbations at the feature level. The goal is to disrupt the normal feature distribution of the input data so that the model learns wrong or meaningless patterns.
2. Label Flip: Changing the true label of the sample, thereby confusing the model training. Usually labeling "positive" samples as "negative" or marking attack traffic as normal traffic.
3. Logic Disruption: Disrupt the logical relationship or dependency between features. Simulate malicious attackers to disrupt model learning by destroying normal feature correlations.
4. Malicious Pattern Injection: Inject artificially designed patterns into data to disguise as legitimate or malicious samples. Simulate complex network attack strategies and forge normal traffic to cover up attack behavior.
5. PGD Attack (Projected Gradient Descent): A classic adversarial attack method that generates adversarial samples through gradient calculation. The goal is to introduce minimal perturbations in the input features by maximizing the prediction loss of the model.

$$\mathbf{x}' = \text{clip}_{\mathbf{x}, \epsilon}(\mathbf{x} + \alpha \cdot \text{sign}(\nabla_{\mathbf{x}} \mathcal{L}))$$

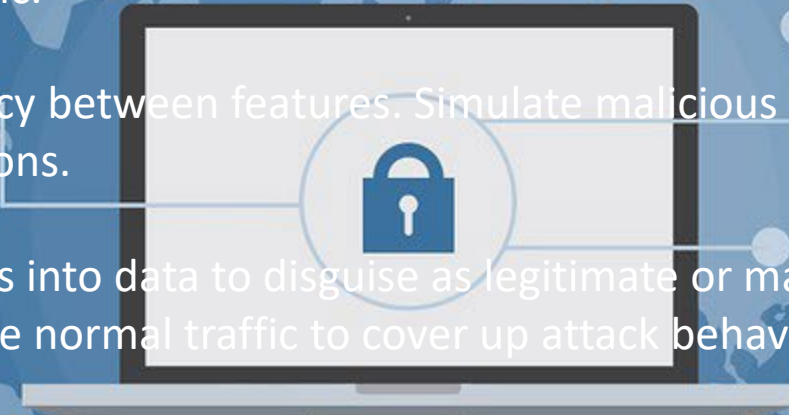
Where

\mathbf{x}' : Perturbed input.

ϵ : Maximum allowable perturbation.

α : Step size for each iteration.

\mathcal{L} : Loss function.



Model includes two parts:

1. Basic model

2. Adversarial Data Poisoning and Replay Mechanism

```
class IncrementalLearningModel(nn.Module):
    def __init__(self, input_size, num_classes):
        super(IncrementalLearningModel, self).__init__()
        # Define neural network structure with hidden layers
        self.fc1 = nn.Linear(input_size, 256)
        self.relu1 = nn.ReLU()
        self.fc2 = nn.Linear(256, 128)
        self.relu2 = nn.ReLU()
        self.fc3 = nn.Linear(128, num_classes)

    def forward(self, x):
        x = self.fc1(x)
        x = self.relu1(x)
        x = self.fc2(x)
        x = self.relu2(x)
        x = self.fc3(x)
        return x

    def train_model(self, X_train, y_train, X_val, y_val, num_epochs=50, lr=1e-4, poisoning_rate=0.8):
        # Convert numpy arrays to torch tensors
        train_data = X_train.clone().detach().float().to(device) if isinstance(X_train, torch.Tensor) else torch.tensor(X_train, dtype=torch.float32).to(device)
        train_labels = y_train.clone().detach().long().to(device) if isinstance(y_train, torch.Tensor) else torch.tensor(y_train, dtype=torch.long).to(device)
        val_data = X_val.clone().detach().float().to(device) if isinstance(X_val, torch.Tensor) else torch.tensor(X_val, dtype=torch.float32).to(device)
        val_labels = y_val.clone().detach().long().to(device) if isinstance(y_val, torch.Tensor) else torch.tensor(y_val, dtype=torch.long).to(device)

        # Define loss function and optimizer
        criterion = nn.CrossEntropyLoss()
        optimizer = optim.Adam(self.parameters(), lr=lr)

        # Lists to store losses for visualization
        train_losses = []
        val_losses = []

        # Training Loop
        for epoch in range(num_epochs):
            self.train()
            optimizer.zero_grad()
            outputs = self(train_data)
            loss = criterion(outputs, train_labels)
            loss.backward()
            optimizer.step()

            # Record training loss
            train_losses.append(loss.item())

            # Validation step
            self.eval()
            with torch.no_grad():
                val_outputs = self(val_data)
                val_loss = criterion(val_outputs, val_labels)
                val_losses.append(val_loss.item())

        print(f"Epoch [{epoch+1}/{num_epochs}], Train Loss: {loss.item():.4f}, Val Loss: {val_loss.item():.4f}")
```

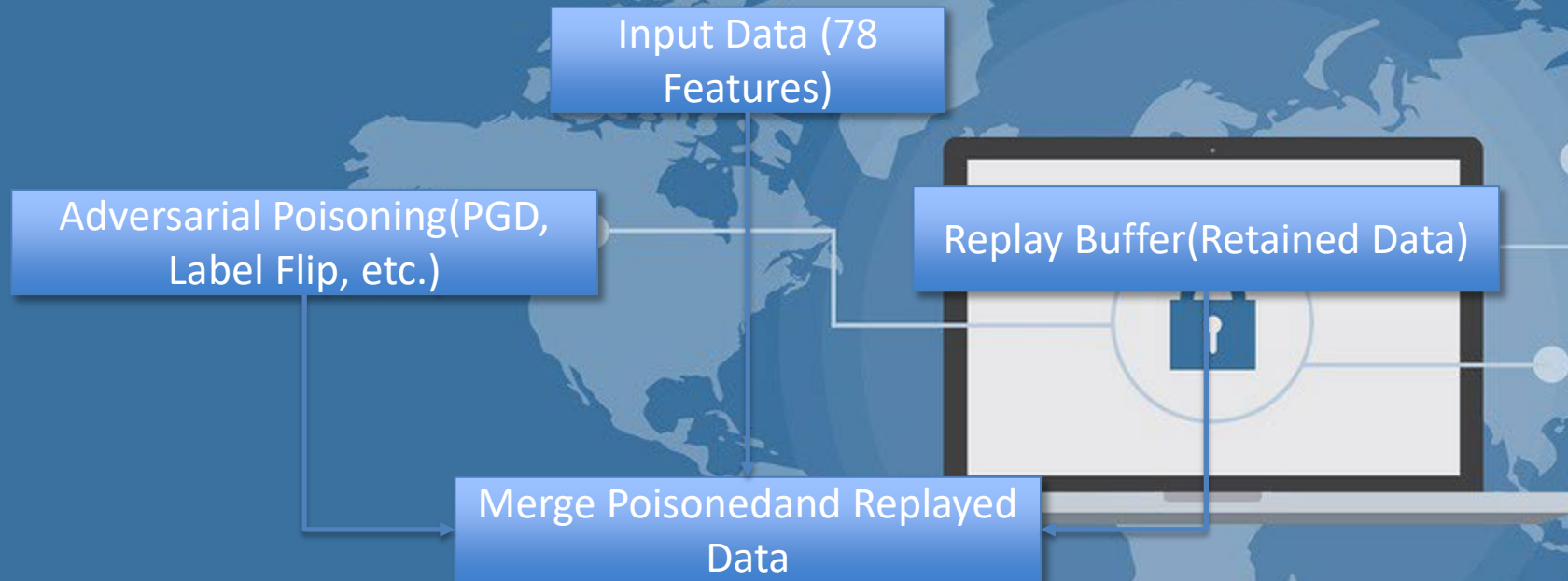
```
# Function to handle catastrophic forgetting using Replay mechanism
def replay_mechanism(model, previous_data, new_data, previous_labels, new_labels, num_epochs=5, lr=1e-4):
    # Convert numpy arrays to torch tensors
    previous_data = previous_data.clone().detach().float().to(device) if isinstance(previous_data, torch.Tensor) else torch.tensor(previous_data, dtype=torch.float32).to(device)
    previous_labels = previous_labels.clone().detach().long().to(device) if isinstance(previous_labels, torch.Tensor) else torch.tensor(previous_labels, dtype=torch.long).to(device)
    new_data = new_data.clone().detach().float().to(device) if isinstance(new_data, torch.Tensor) else torch.tensor(new_data, dtype=torch.float32).to(device)
    new_labels = new_labels.clone().detach().long().to(device) if isinstance(new_labels, torch.Tensor) else torch.tensor(new_labels, dtype=torch.long).to(device)

    # Combine previous and new data
    combined_data = torch.cat((previous_data, new_data), dim=0)
    combined_labels = torch.cat((previous_labels, new_labels), dim=0)

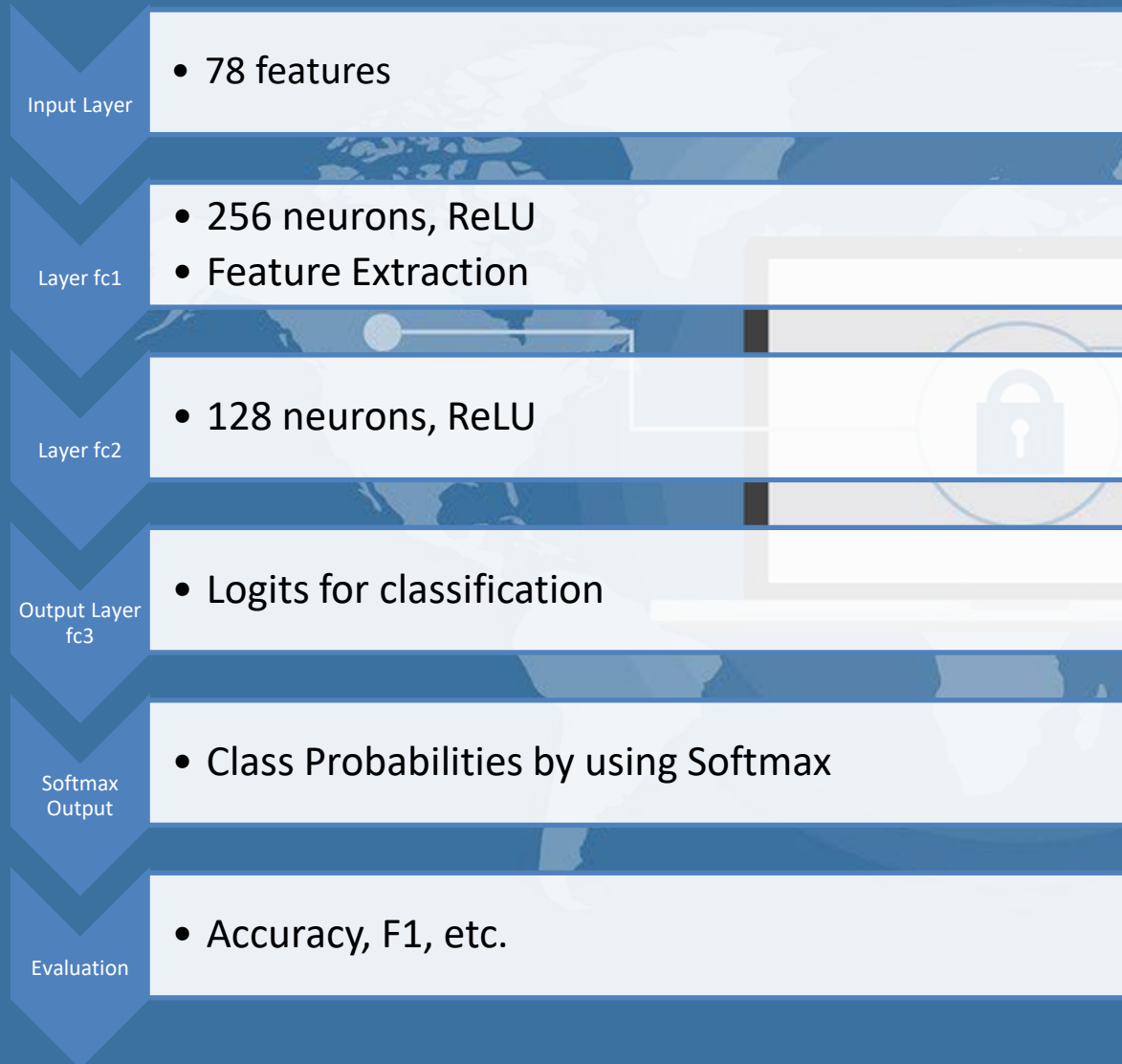
    # Retrain the model with combined dataset
    model.train_model(combined_data, combined_labels, combined_data, combined_labels, num_epochs, lr)
```



Adversarial Data Poisoning and Replay Mechanism



Basic Model (FC MEANS FULL CONNECTION)



Analysis Results

This study systematically evaluates the robustness of a class-incremental learning system under various adversarial data poisoning strategies. Through detailed experiments and metrics analysis, the following key insights were obtained.

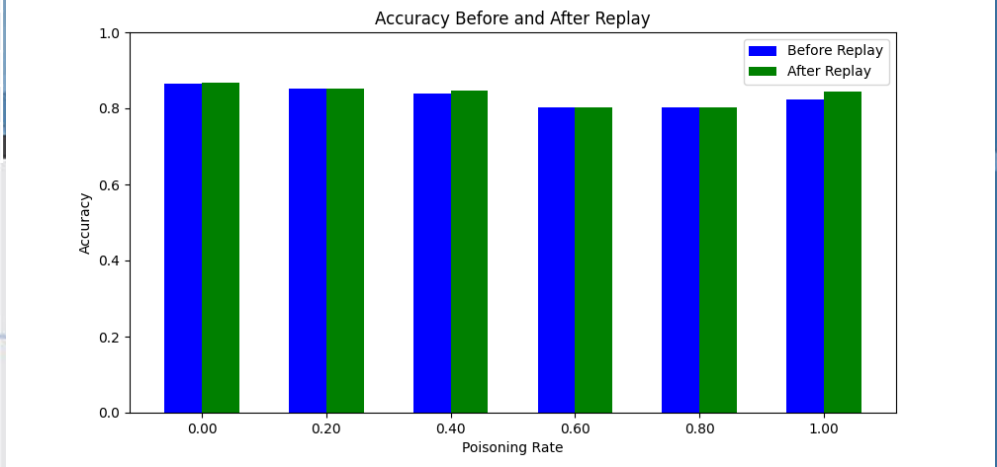
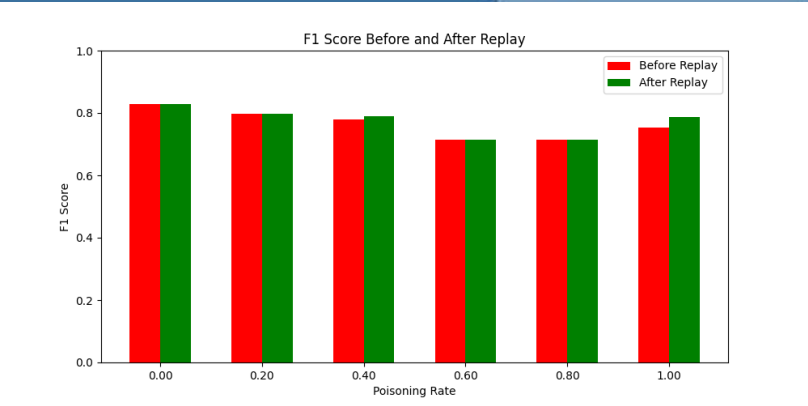


Feature Perturbation

Injecting random noise into feature data causes the model to learn erroneous or non-relevant patterns.

poisoning_rate	accuracy_before	precision_before	recall_before	f1_score_before	accuracy_after	precision_after	recall_after	f1_score_after
0	0.865686	0.8706	0.865686	0.828017	0.86744	0.871617	0.86744	0.828546
0.2	0.852068	0.871875	0.852068	0.796423	0.852107	0.871776	0.852107	0.796464
0.4	0.839141	0.86446	0.839141	0.779978	0.846779	0.870003	0.846779	0.790008
0.6	0.803011	0.841816	0.803011	0.715278	0.803011	0.841816	0.803011	0.715278
0.8	0.803011	0.841816	0.803011	0.715278	0.803011	0.841816	0.803011	0.715278
1	0.822415	0.854413	0.822415	0.754416	0.843738	0.868866	0.843738	0.786134

```
# Vectorized feature perturbation
poisoned_data = data.clone()
perturbation = torch.randn_like(poisoned_data[poisoned_indices]) * 1.0 # Larger perturbation scale
poisoned_data[poisoned_indices] += perturbation
return poisoned_data, labels
```



Low poisoning rate (Poisoning Rate \leq 0.4): It can be seen that with the increase of poisoning rate, the accuracy of the data has decreased significantly, but the poisoning rate is low, so the decrease in the data is not too obvious from 0.87 to 0.84, but the F1 value has a more obvious decrease from 0.83 to 0.78. However, after using the replay mechanism, the data has improved to varying degrees.

High poisoning rate (Poisoning Rate > 0.4): The accuracy dropped to around 0.80 and the F1 score dropped to 0.71. The recovery effect of Replay is weakened, and the performance loss is still obvious under high poisoning rates.

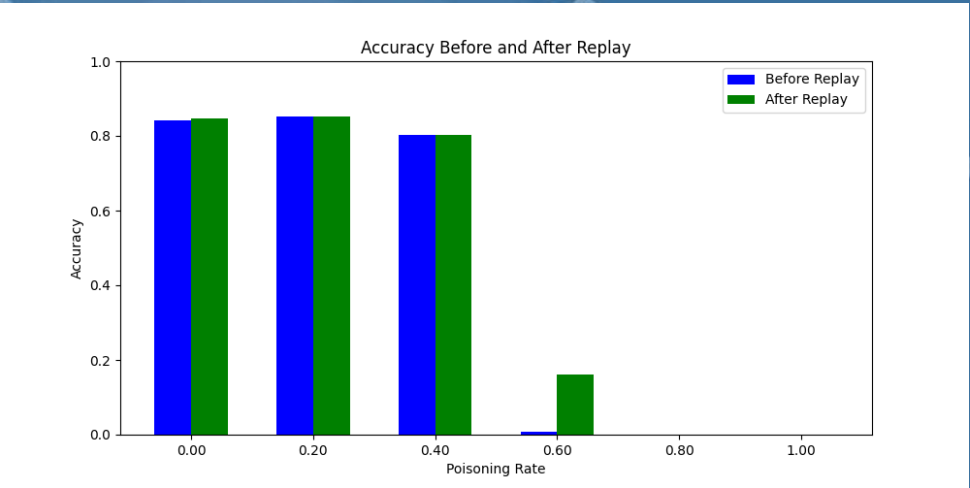
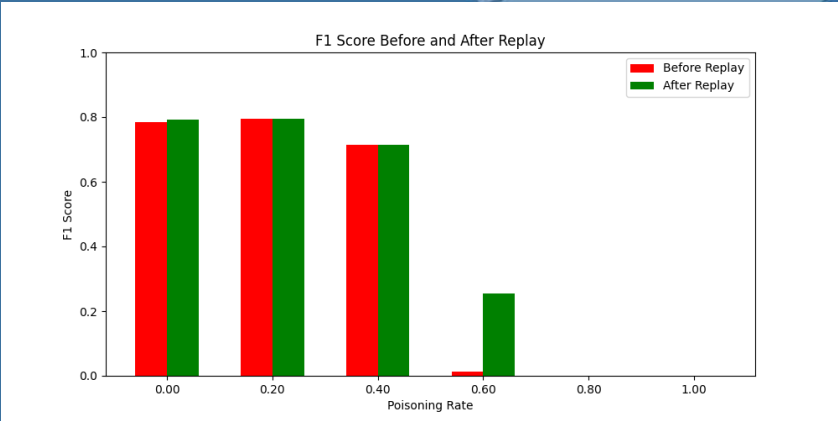
So that, the impact of feature perturbations gradually emerges, but the overall destructiveness is limited. However, we still can observe that replay has good recovery ability at low poisoning rates.

Label Flip

Randomly flip the labels to increase the confusion of the categories.

poisoning_rate	accuracy_before	precision_before	recall_before	f1_score_before	accuracy_after	precision_after	recall_after	f1_score_after
0	0.842051	0.866497	0.842051	0.783897	0.847566	0.87054	0.847566	0.790993
0.2	0.851504	0.870155	0.851504	0.795888	0.851511	0.870217	0.851511	0.795896
0.4	0.803011	0.841816	0.803011	0.715278	0.803011	0.841816	0.803011	0.715278
0.6	0.008163	0.918696	0.008163	0.013855	0.159986	0.860659	0.159986	0.255541
0.8	0.00069	0.997357	0.00069	1.01E-06	0.00069	0.997357	0.00069	1.02E-06
1	0.00069	0.997357	0.00069	9.50E-07	0.00069	0.997357	0.00069	9.55E-07

```
# Vectorized label flipping
poisoned_labels = labels.clone()
poisoned_labels[poisoned_indices] = (labels[poisoned_indices] + 1) % len(torch.unique(labels))
return data, poisoned_labels
```



Low poisoning rate (Poisoning Rate≤0.2):Accuracy is between 0.84 - 0.85, and F1 score is close to 0.79.Replay can effectively restore F1 score and accuracy.

High poisoning rate (Poisoning Rate>0.4): Accuracy drops below 0.01, and F1 score is close to 0. Replay has limited recovery effect in extreme cases, and performance loss is difficult to make up.

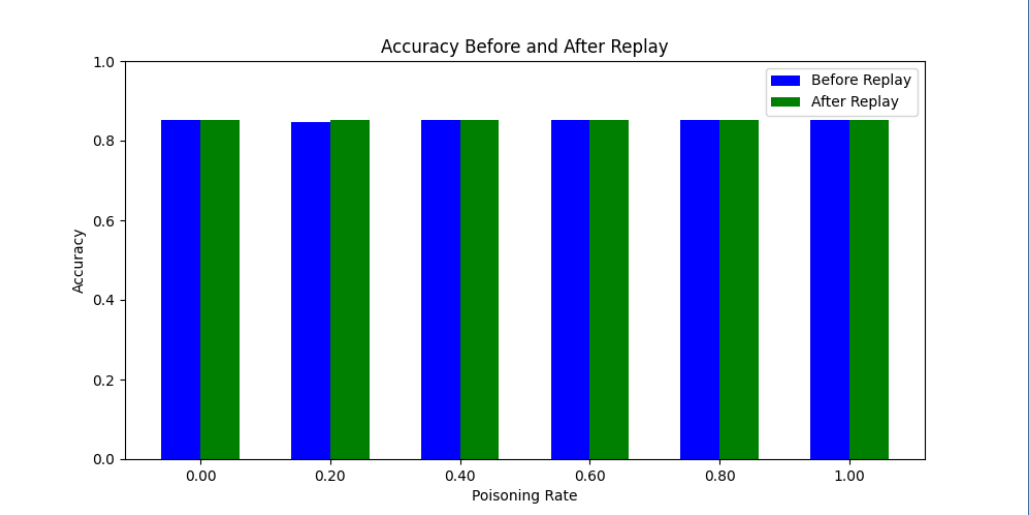
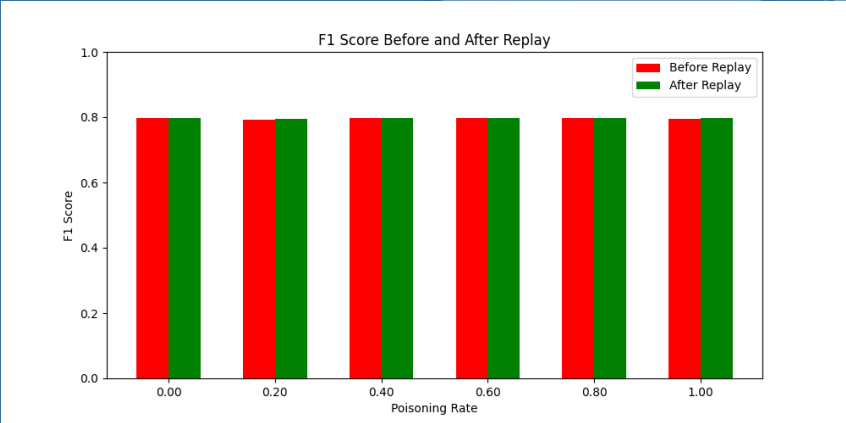
Label Flip is one of the most destructive strategies, especially at high poison rates. Replay does not recover much in extreme situations.

Logic Disruption

Destroy the logical relationship between features, such as disrupting the order.

poisoning_rate	accuracy_before	precision_before	recall_before	f1_score_before	accuracy_after	precision_after	recall_after	f1_score_after
0	0.851963	0.872902	0.851963	0.796367	0.851995	0.872915	0.851995	0.796402
0.2	0.8476	0.870686	0.8476	0.791039	0.851019	0.873017	0.851019	0.795234
0.4	0.851986	0.873008	0.851986	0.796369	0.85199	0.872957	0.85199	0.796371
0.6	0.852395	0.870023	0.852395	0.796907	0.852409	0.87003	0.852409	0.796922
0.8	0.851947	0.873388	0.851947	0.796336	0.852002	0.873398	0.852002	0.7964
1	0.851672	0.872696	0.851672	0.796089	0.851691	0.872685	0.851691	0.796106

```
# Logic Disruption (breaking feature relationships)
poisoned_data = data.clone()
for feature_idx in [0, 1, 2]:
    poisoned_data[:, feature_idx] = poisoned_data[:, feature_idx][torch.randperm(poisoned_data.size(0))]
return poisoned_data, labels
```



There was little significant impact on model performance, with accuracy consistently maintained in the 0.85 - 0.87 range. Replay's performance improvement is almost negligible.

The model has low reliance on feature logic and high robustness to Logic Disruption. The Replay mechanism has limited effect under this strategy.

Malicious Pattern Injection

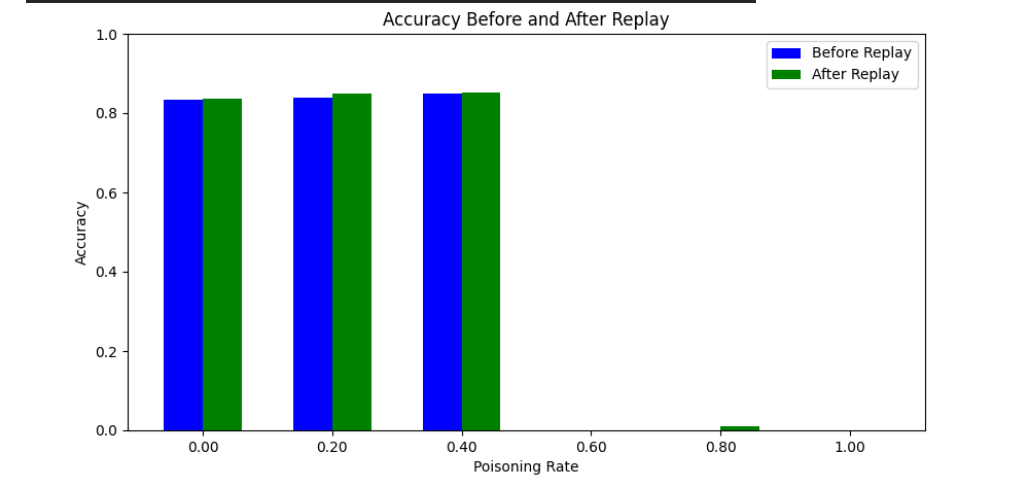
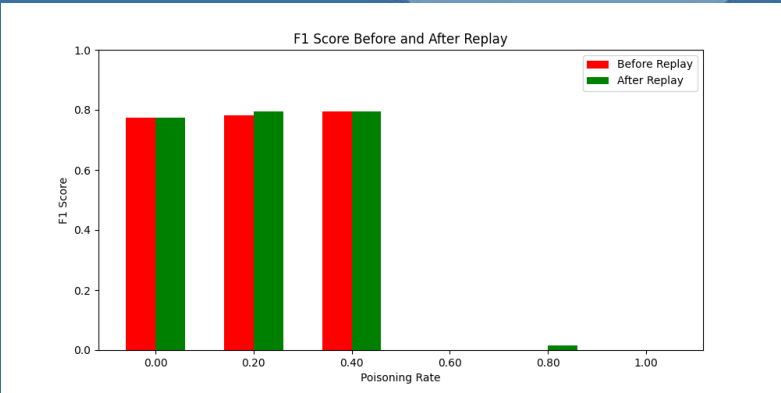
Inject artificially designed malicious patterns into the data to interfere with model classification.

poisoning_rate	accuracy_before	precision_before	recall_before	f1_score_before	accuracy_after	precision_after	recall_after	f1_score_after
0	0.834974	0.861955	0.834974	0.774123	0.835592	0.862208	0.835592	0.775012
0.2	0.840066	0.864875	0.840066	0.781241	0.849644	0.871934	0.849644	0.793561
0.4	0.84938	0.872461	0.84938	0.794111	0.851127	0.872931	0.851127	0.795393
0.6	0.00069	0.997357	0.00069	9.73E-07	0.00069	0.997357	0.00069	9.83E-07
0.8	0.00069	0.997357	0.00069	1.01E-06	0.009123	0.997357	0.009123	0.016693
1	0.00069	0.997357	0.00069	9.50E-07	0.00069	0.997357	0.00069	9.50E-07

```
# Malicious Pattern Injection
poisoned_data = data.clone()
pattern_strength = 0.5
pattern = torch.randn(len(poisoned_indices), data.size(1)).to(device) * pattern_strength
poisoned_data[poisoned_indices] += pattern

# Randomly flip some Labels to further confuse the model
poisoned_labels = labels.clone()
poisoned_labels[poisoned_indices] = (labels[poisoned_indices] + 1) % len(torch.unique(labels))

return poisoned_data, poisoned_labels
```



Low poisoning rate (Poisoning Rate \leq 0.4): Accuracy is 0.83 - 0.85, F1 score is 0.77 - 0.79.Replay can effectively restore some performance, especially when the poisoning rate is low.

High poisoning rate (Poisoning Rate $>$ 0.4): Accuracy and F1 scores both dropped significantly, and Replay was of limited use.

Injecting malicious patterns can significantly interfere with the model's classification capabilities. Replay is effective at low poison rates, but has weak recovery at high poison rates.

PGD Attack

Generate adversarial examples using gradient optimization to maximize the model’s classification error.

poisoning_rate	accuracy_before	precision_before	recall_before	f1_score_before	accuracy_after	precision_after	recall_after	f1_score_after
0	0.83529	0.861762	0.83529	0.774581	0.835557	0.861841	0.835557	0.774964
0.2	0.851505	0.873641	0.851505	0.795829	0.851695	0.873652	0.851695	0.796051
0.4	0.842886	0.867832	0.842886	0.785009	0.850323	0.872969	0.850323	0.794406
0.6	0.852084	0.872496	0.852084	0.796465	0.852103	0.872169	0.852103	0.796478
0.8	0.832097	0.853689	0.832097	0.770048	0.835242	0.856016	0.835242	0.774616
1	0.803011	0.841816	0.803011	0.715278	0.803011	0.841816	0.803011	0.715278

```
# PGD attack
if model is None or criterion is None:
    raise ValueError("Model and criterion are required for 'pgd_attack' strategy.")

poisoned_data = data.clone().detach().to(device)
poisoned_data.requires_grad_ = False

batch_size = 32
for i in range(10):
    for start_idx in range(0, num_poisoned, batch_size):
        end_idx = min(start_idx + batch_size, num_poisoned)
        batch_indices = poisoned_indices[start_idx:end_idx]
        batch_data = poisoned_data[batch_indices].clone().detach().requires_grad_(True) # Ensure batch_data is a leaf tensor
        batch_labels = labels[batch_indices]

        # Forward pass through the model
        outputs = model(batch_data)

        # Compute loss
        loss = criterion(outputs, batch_labels)

        # Zero out any previous gradients
        model.zero_grad()
        if batch_data.grad is not None:
            batch_data.grad.zero_()

        # Backward pass
        loss.backward()

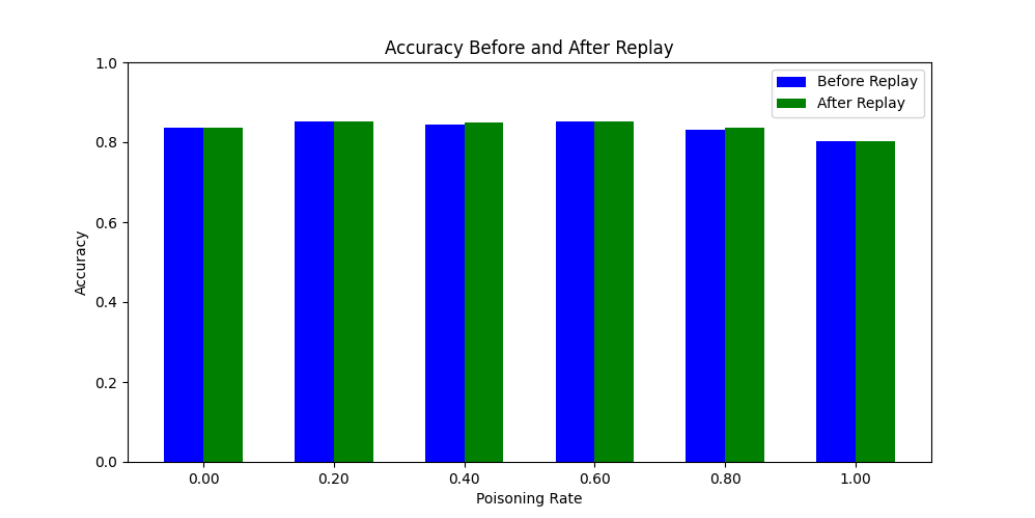
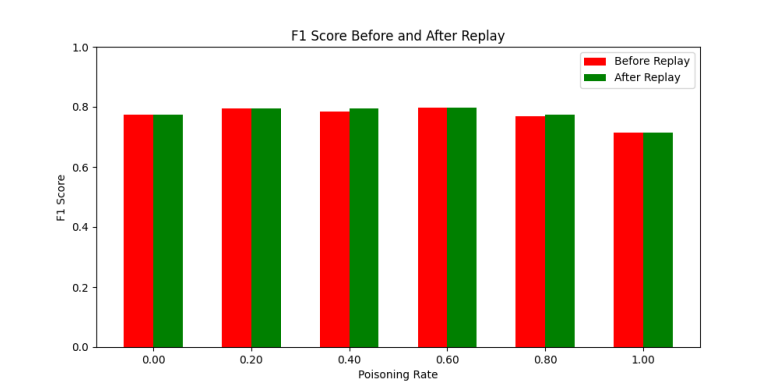
        # Retain gradients on batch_data so we can use them
        batch_data.retain_grad()

        # Check if gradients are available
        if batch_data.grad is not None:
            # Generate adversarial perturbation
            gradient = batch_data.grad.sign()
            batch_data = batch_data + alpha * gradient

            # Clamp the perturbed data
            batch_data = torch.clamp(batch_data, min_data[batch_indices] - epsilon, max_data[batch_indices] + epsilon)
            batch_data = torch.clamp(batch_data, 0, 1) # Assuming the data is in range [0, 1]

            # Update the poisoned data (avoid in-place modification of leaf variable)
            poisoned_data = poisoned_data.clone().detach()
            poisoned_data[batch_indices] = batch_data.clone().detach() # Use clone().detach() to avoid in-place update of tensor requiring grad
        else:
            print("Warning: Gradient is None during PGD attack.")

torch.cuda.empty_cache()
```



Low poisoning rate (Poisoning Rate≤0.2): Accuracy and F1 score are close to 0.85 - 0.86.Replay shows good recovery ability at low poisoning rate.

High poisoning rate (Poisoning Rate>0.4): Accuracy drops below 0.80 and F1 score drops to 0.77.Replay has limited recovery ability at high poisoning rate.

PGD attack has moderate impact on the model, and Replay can effectively mitigate performance loss at low poisoning rate.

Combined Strategy

Combine multiple attack strategies (such as Label Flip and Feature Perturbation) to maximize the performance loss.

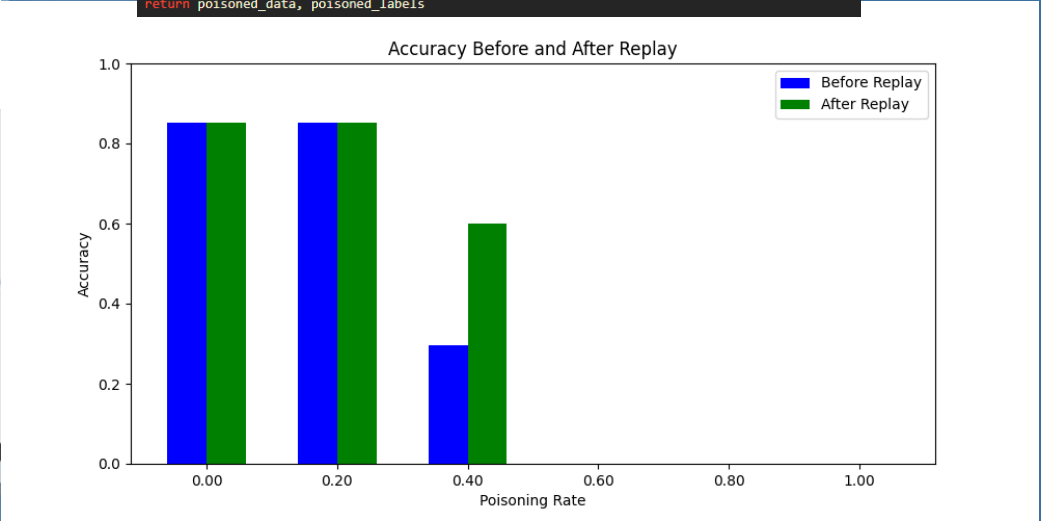
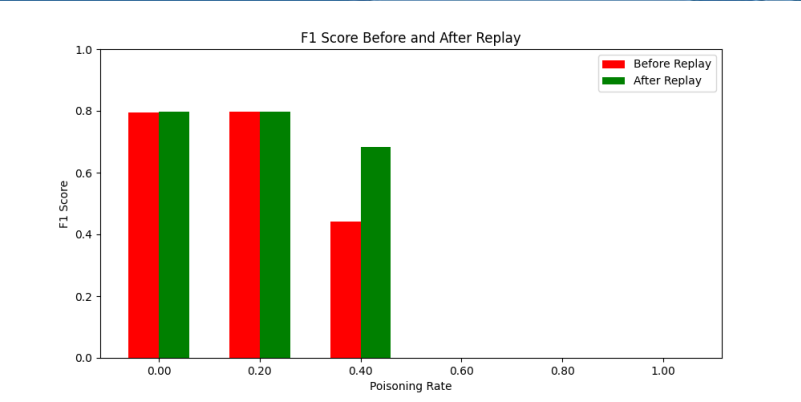
poisoning_rate	accuracy_before	precision_before	recall_before	f1_score_before	accuracy_after	precision_after	recall_after	f1_score_after
0	0.851585	0.874223	0.851585	0.795949	0.851825	0.874279	0.851825	0.796233
0.2	0.852085	0.872143	0.852085	0.79646	0.852103	0.872114	0.852103	0.796477
0.4	0.296821	0.991691	0.296821	0.44216	0.600043	0.911969	0.600043	0.683943
0.6	0.00069	0.999311	0.00069	9.50E-07	0.00069	0.999311	0.00069	9.50E-07
0.8	0.00069	0.999311	0.00069	9.50E-07	0.00069	0.997357	0.00069	9.50E-07
1	0.00069	0.999311	0.00069	9.50E-07	0.00069	0.999311	0.00069	9.50E-07

```
# Combined Label flipping and feature perturbation
poisoned_data = data.clone()
poisoned_labels = labels.clone()

# Apply feature perturbation
perturbation = torch.randn_like(poisoned_data[poisoned_indices]) * 0.5
poisoned_data[poisoned_indices] += perturbation

# Flip Labels
poisoned_labels[poisoned_indices] = (labels[poisoned_indices] + 1) % len(torch.unique(labels))

return poisoned_data, poisoned_labels
```



Low poisoning rate (Poisoning Rate≤0.4): Both accuracy and F1 score drop significantly, but Replay can partially recover the performance.

High poisoning rate (Poisoning Rate>0.4): The accuracy is close to 0 and the F1 score is almost not restored. Replay cannot effectively alleviate the performance loss in extreme cases.

Combined Strategy is a destructive strategy.
Replay mechanism is effective at low poisoning rate, but has no significant effect at high poisoning attack.

Thank you for watching my report

