Few-Shot Learning
For Intrusion Detection
Using Prototypical Networks

Presentation by *Mohammed Aawez* Mansuri
Prof. Wonjun Lee, Committee Chair
Prof. Ani Nahapetian, Committee Member
Prof. Robert McIlhenny, Committee Member

Background

Introduction
Network Intrusion
Rule-Based Systems & Limitations
Machine Learning Systems & Limitations
Few-Shot Learning & Its Approaches
Prototypical Networks

Introduction

The digital age brings with it an ever-evolving cyber threat landscape.

Intrusion Detection Systems (IDS) are crucial for identifying and mitigating these threats.

Traditional and machine learning-based IDS face challenges: high false-positive and false-negative rates, and scarcity of training data.

The need for a new approach: Few-Shot Learning.

Introduction (Cont'd)

Thesis focus: "Exploring Few-Shot Learning for Intrusion Detection using Prototypical Networks."

Prototypical Networks: Innovative models leveraging Few-Shot Learning to recognize patterns from limited examples.

Research aim: Assess the effectiveness of Prototypical Networks for network intrusion detection.

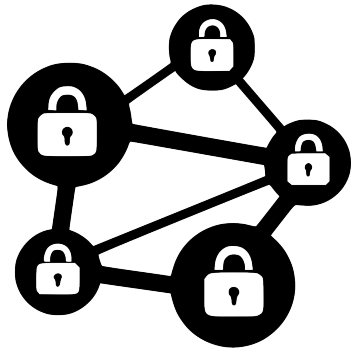
Potential impact: Lower false-positives, false-negatives & better model performance.

What is Network Intrusion?

Network Intrusion: Unauthorized activities aimed at compromising the integrity, confidentiality, or availability of a network.

Types of Intrusion: Malware infections, Distributed Denial of Service (DDoS) attacks, and unauthorized access to sensitive information.

Consequences of Intrusion: Data breaches, system downtime, financial loss, and damage to organizational reputation.



Rule-Based Systems

Traditional IDS rely heavily on predefined rules, which need to be manually created and updated.

These systems use hard-coded signatures to identify known threats.

The rules and signatures are static and do not adapt to changing threat landscapes.

Deprecated Approach.

Limitations: Rule-Based

Inability to Adapt: Traditional systems struggle to adapt to new and evolving cyber threats, often relying on predefined rules and signatures.

High False Positives: Often, these systems generate a high number of false positives, leading to unnecessary resource consumption and potential overlooking of real threats.

Resource-Heavy: Maintaining and updating rule sets and signatures in traditional IDS can be resource-intensive.

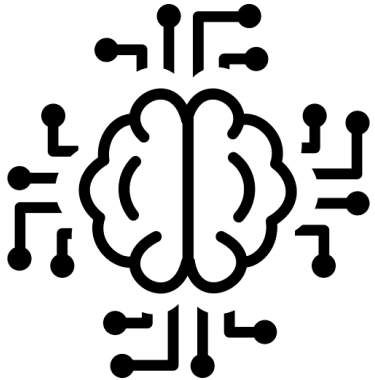
Machine Learning Systems

Machine Learning (ML) based systems can adapt to new threats by learning from patterns in the data.

ML models can detect patterns in network traffic that deviate from normal behavior, potentially identifying novel attacks.

These systems continually learn and improve over time without the need for manual rule updates.

Different ML methods, such as supervised, and unsupervised learning, offer various strategies for intrusion detection.



Limitations: Machine-Learning

Data Dependency: ML models require large amounts of labeled training data to perform well, and good quality data can be difficult to obtain.

Overfitting: ML models can overfit to the training data, limiting their ability to generalize to new, unseen threats.

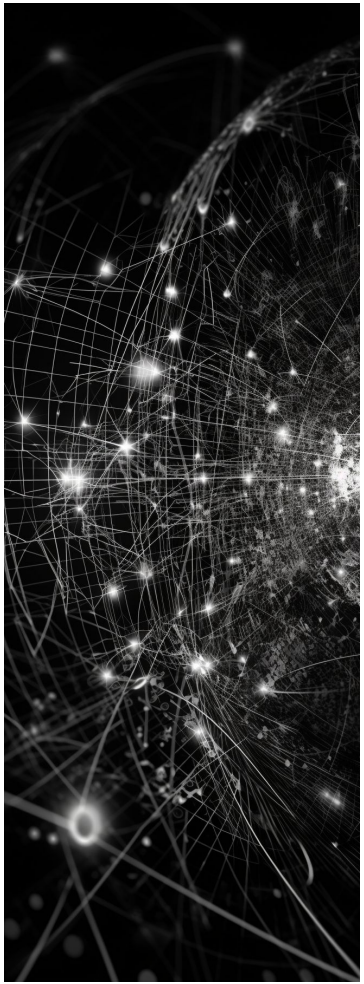
High False Positives: Despite their learning capabilities, traditional ML algorithms can still produce high false positives, particularly in the face of novel or sophisticated attacks.

Few-Shot Learning

Few-shot learning refers to the idea of understanding new concepts with minimal examples or data.

Originally applied in image classification, it has since found use in various domains such as computer vision (face recognition, object detection), natural language processing (text classification), and now, intrusion detection.

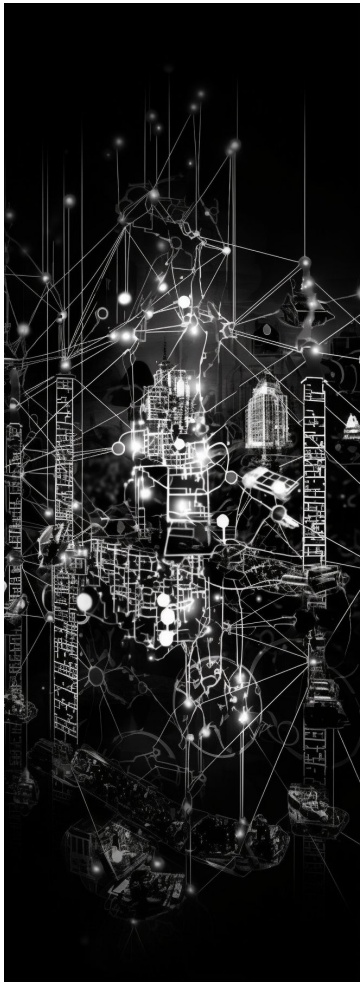
Few-Shot Learning offers the potential to recognize unseen threats from minimal examples, addressing the challenge of limited reliable training data.



Few-Shot Approaches

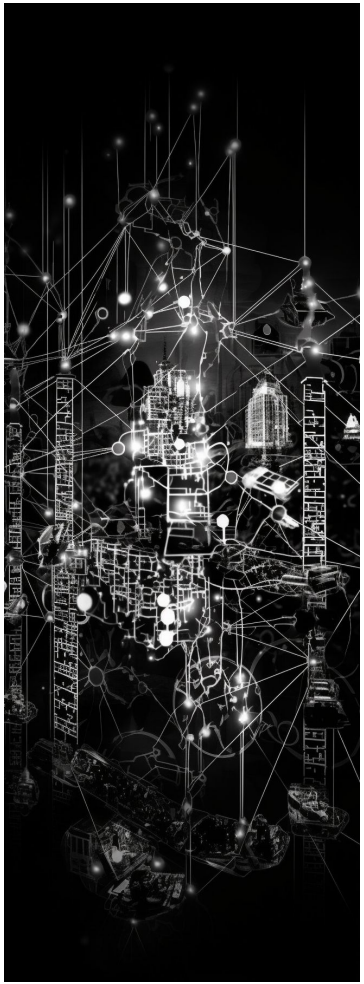
Metric-based Meta-Learning: These methods learn a distance function over the input space, which can then be used to classify new examples based on their similarity to existing ones. An example of this approach is *Prototypical Networks*.

Optimization-based Meta-Learning: These approaches aim to learn an optimal initialization of a model's parameters, such that the model can quickly adapt to new tasks with a few gradient steps. Example: *Model-Agnostic Meta-Learning* (MAML) algorithm.



Few-Shot Approaches (Cont'd)

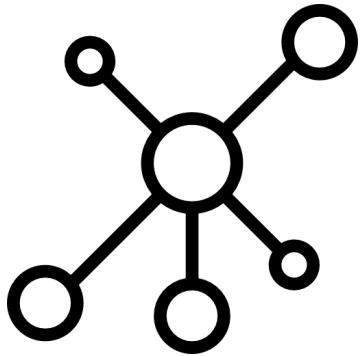
Model-based Meta-Learning: These methods leverage internal or external memory components to store information about tasks, allowing the model to generalize from past tasks to new ones. Examples include *Memory-Augmented Neural Networks* (MANN).



Prototypical Networks

Prototypical Networks are a type of metric-based meta-learning model. They classify new instances based on their similarity to prototype representations, which are the mean embeddings of examples in the same class.

Prototypical Networks are designed for Few-Shot Learning scenarios, where they learn to form class representations (prototypes) from a few training examples.

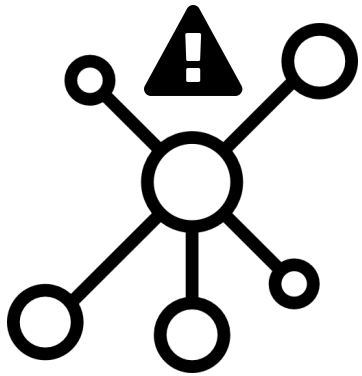


Why Prototypical Networks?

Ability to learn from few examples: This is crucial in IDS scenarios where high-quality labeled data is scarce.

Adaptability to new threats: By forming new prototypes, Prototypical Networks can adapt to novel intrusion patterns.

Reduced false positives: By learning more representative prototypes, these networks have the potential to reduce the false positive rate in IDS.



Methodology

Tools Used
Dataset
Data Pre-Processing
Model Training
Model Testing
Performance Metrics
Model Comparison

Tools Used ⚒

Programming Language: Python

Platform: Jupyter Notebook

Data Handling: Numpy, Pandas

Data Processing: Sklearn

Data Visualization: Matplotlib

Machine Learning: PyTorch

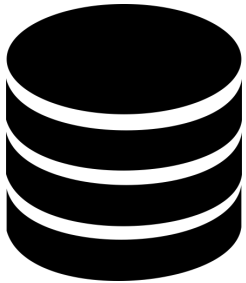
NSL-KDD Dataset

An improved version of the KDD'99 dataset widely used for building and evaluating intrusion detection systems.

The NSL-KDD dataset provides a wide range of different intrusions.

The dataset has been preprocessed to remove redundant records, making it more efficient for model training.

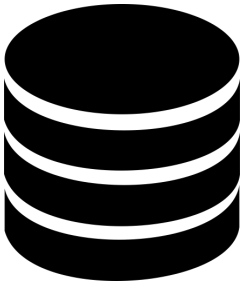
It's a well-accepted benchmark in the intrusion detection domain.



NSL-KDD Dataset (Cont'd)

The NSL-KDD dataset includes separate training and testing data.

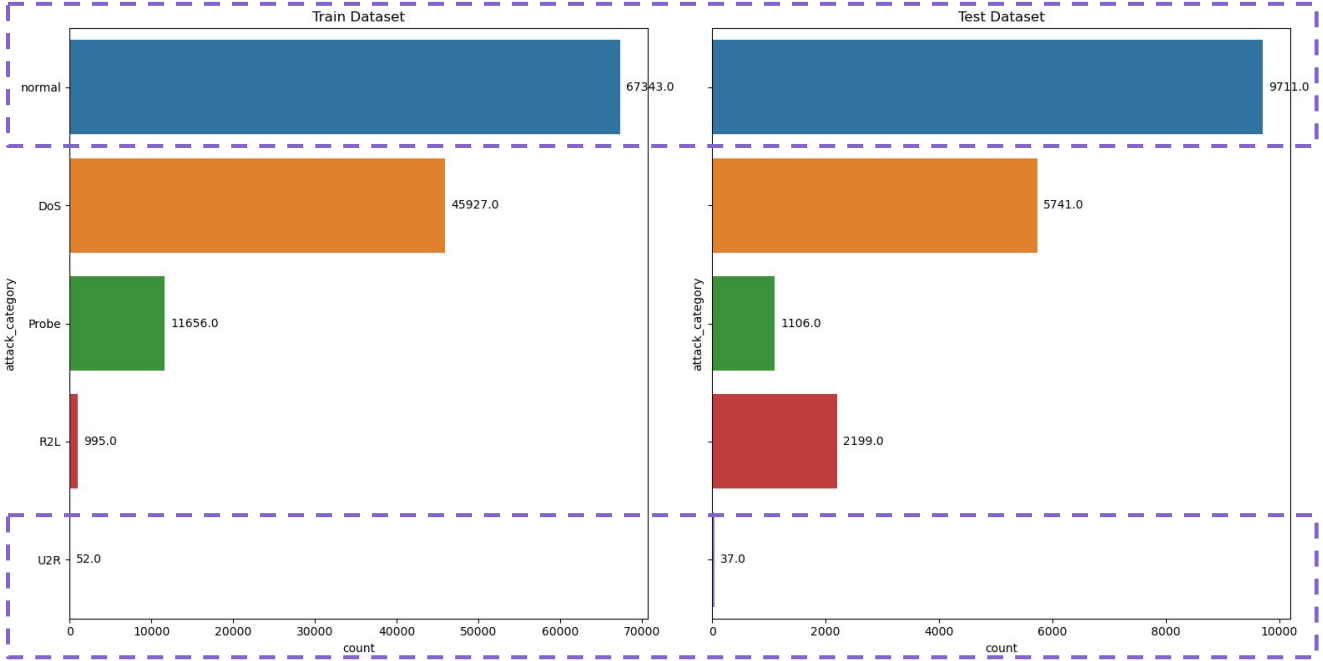
The training dataset comprises 125,973 samples and while the testing dataset includes 22,544 samples.



This dataset contains five classes: Normal, DoS (Denial of Service), Probe, R2L(Remote to Local), and U2R (User to Root).

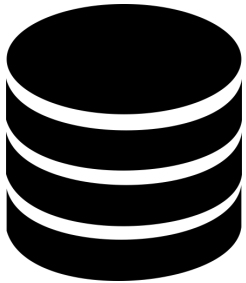
To evaluate the effectiveness of Prototypical Network with limited data, we trained the model solely on Normal and U2R instances.

NSL-KDD Dataset (Cont'd)



NSL-KDD Dataset (Cont'd)

The NSL-KDD dataset contains a total of 41 features, which can be grouped into the following categories:



Basic features: protocol type, service, and flag.

Content-related features: failed login attempts, file operations.

Traffic features: packets sent/received.

Host-based features: root accesses, file creations.

Time-based features: time between connections, duration.

Pre-Processing Pipeline (Cont'd)

Loading the Data in Pandas DataFrame.

Mapping & Labeling U2R records.

Performing Undersampling to balance the dataset.

One-Hot Encoding the categorical features.

Normalizing Data.

Pre-Processing Pipeline (Cont'd)

Mapping U2R Class

```
u2r_attacks = [
    "buffer_overflow",
    "loadmodule",
    "perl",
    "rootkit"
]
```

```
# Replace class labels
train_data['is_u2r'] = train_data['attack_type'].apply(lambda x: 1 if x in u2r_attacks else 0)
test_data['is_u2r'] = test_data['attack_type'].apply(lambda x: 1 if x in u2r_attacks else 0)
```

Undersampling

```
# Perform under-sampling
rus = RandomUnderSampler(random_state=42)
X_train_resampled, y_train_resampled = rus.fit_resample(X_train, y_train)
```

```
X_train_resampled.shape
```

```
(104, 40)
```

Pre-Processing Pipeline (Cont'd)

One-Hot Encoding

```
categorical_columns = ["protocol_type", "service", "flag"]

# Initialize the OneHotEncoder and fit_transform the combined_data
encoder = OneHotEncoder()
combined_data_encoded = encoder.fit_transform(combined_data[categorical_columns])
```

Data Normalization

```
# Normalize the data
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

Training Parameters

Epochs: 10

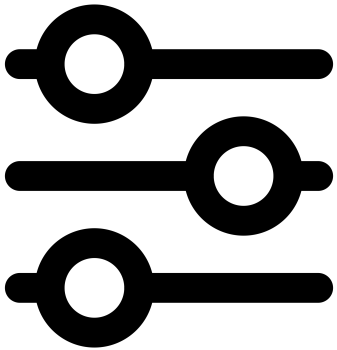
Optimizer: Adam

Learning Rate: 0.001

Batch Size: 40

Support Set: 10 (Each Class)

Query Set: 20

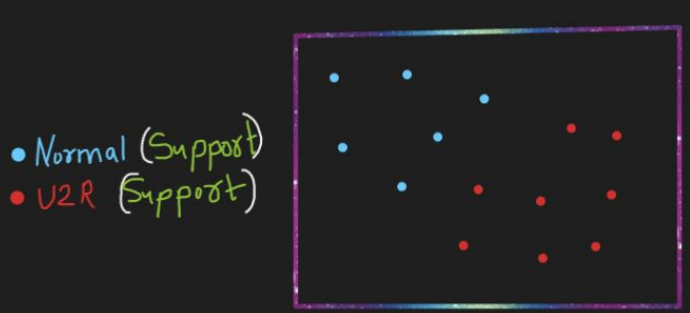


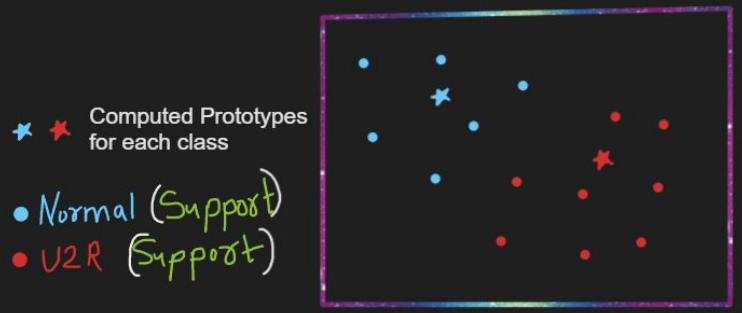
Model Training

Support Set

Compute Embeddings

Calculate Class Prototypes

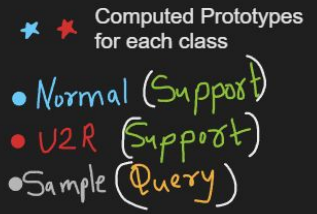


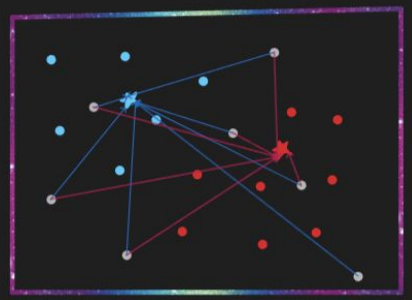


Model Training

Query Set

Compute Embeddings → Calculate Loss





Calculate euclidean distance between the prototypes & query set.

Apply softMax function to negative distances to convert them into probability distributions such that the sample closer to the prototype will have higher probability.

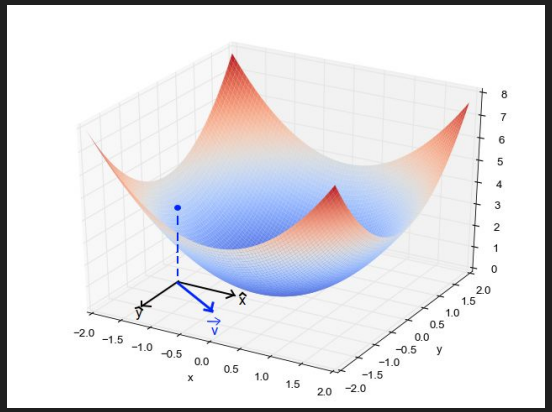
Compute cross-entropy loss between the probabilities and target labels.

Model Training

Repeat for 10 (N) Epochs

Compute Gradients

Optimize The Model



Update the model parameters using the Adam optimizer and the computed gradients.

End of Training

After 10 (N) Epochs, we would have an accurate embedding representation reflecting class similarity. The samples that belongs to the same classes will be much closer to each other.

Model Training

```
for epoch in range(epochs):
    model.train()
    epoch_loss = 0.0

    for batch in train_loader:
        data, labels = batch
        data, labels = data.to(device), labels.to(device)

        # Assuming you have balanced classes in each batch, you can compute the number of classes:
        n_classes = len(np.unique(labels.cpu().numpy()))

        # Select the support set and query set
        support_set = data[:n_support * n_classes]
        query_set = data[n_support * n_classes:]
        targets = labels[n_support * n_classes:]

        # Calculate prototypes
        model.zero_grad()
        support_set_embeddings = model(support_set)
        prototypes = torch.mean(support_set_embeddings.view(n_classes, n_support, output_dim), dim=1)

        # Calculate loss
        query_set_embeddings = model(query_set)
        loss = prototypical_loss(prototypes, query_set_embeddings, targets, n_support)

        # Update weights
        loss.backward()
        optimizer.step()
```

```
class ProtoNet(nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim):
        super(ProtoNet, self).__init__()
        self.encoder = nn.Sequential(
            nn.Linear(input_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, output_dim)
        )
```

```
def prototypical_loss(prototypes, query_set, targets, n_support):
    distances = euclidean_distance(query_set.unsqueeze(1), prototypes.unsqueeze(0))

    # Apply the softmax function to the negative distances
    probabilities = F.softmax(-distances, dim=1)

    # Compute the cross-entropy loss
    loss = F.cross_entropy(probabilities, targets)

    return loss
```

Model Testing

```
model.eval()
with torch.no_grad():
    # Embed test and train data
    embeddings_test = model(torch.tensor(X_test, dtype=torch.float32).to(device))
    embeddings_train = model(torch.tensor(X_train, dtype=torch.float32).to(device))

    # Calculate prototypes for each class
    unique_classes = torch.unique(torch.tensor(y_train)).tolist()
    prototypes = []
    for cls in unique_classes:
        class_indices = (y_train == cls)
        class_embeddings = embeddings_train[class_indices]
        prototype = torch.mean(class_embeddings, dim=0)
        prototypes.append(prototype)
    prototypes = torch.stack(prototypes)
```

```
# Calculate distances and make predictions
distances = torch.cdist(embeddings_test, prototypes, p=2)
_, predicted_indices = torch.min(distances, dim=1)

# Convert predicted indices to match original labels
predicted_labels = []
for pred in predicted_indices:
    predicted_labels.append(unique_classes[pred.item()])

# Calculate accuracy
accuracy = accuracy_score(y_test, predicted_labels)
print(f'Test Accuracy: {accuracy}')
```

Model Evaluation Metrics

Recall is a measure of how well the model is able to identify all positive or all negative cases.

Recall = TP / TP + FP

Precision is a measure of how accurate the positive predictions are.

Precision = TP / TP + FN

F1 Score is the combination of Recall & Precision.

F1 Score = 2 * Precision * Recall / Precision + Recall

Model Result

Test Accuracy: ***95.69%***

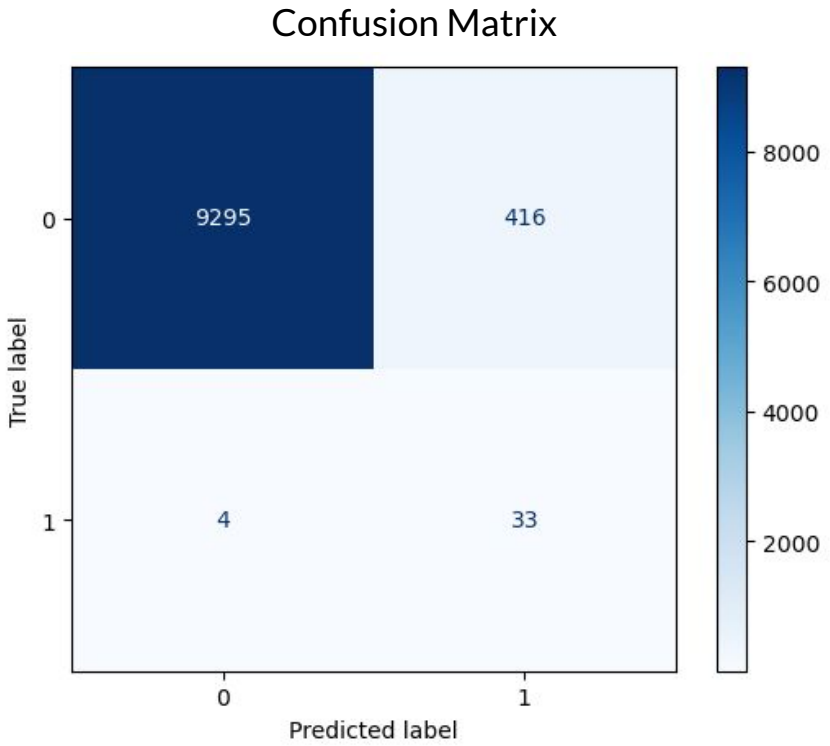
Higher Recall Rate for both Normal & U2R

Correctly Classified ***96%*** of **Normal Samples**

Correctly Classified ***89%*** of **U2R Samples**

```
Test Accuracy: 0.9569142388182191
              precision    recall  f1-score   support

           0       1.00      0.96      0.98      9711
           1       0.07      0.89      0.14        37
```



Model Result (Cont'd)

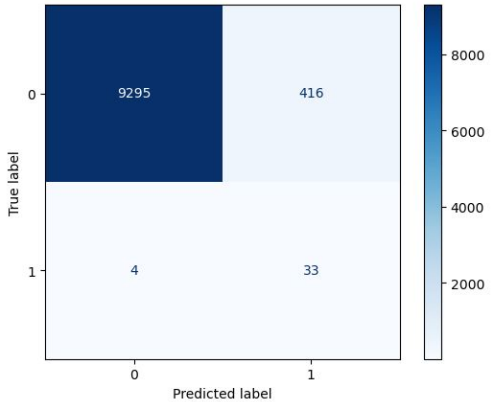
Used only **52** Normal Samples out of **67343** for Training

Used only **0.077 %** of Normal Data

With only **52** U2R Training Samples, model was able to achieve higher recall of **89%**

```
Test Accuracy: 0.9569142388182191
              precision    recall  f1-score   support

           0       1.00      0.96      0.98      9711
           1       0.07      0.89      0.14        37
```



Model Performance Comparison

Neural Network vs Prototypical Networks

NN: 83.7% *Accuracy*, 84% *Normal Recall*, 81% *U2R Recall*

Prototypical Network: 95.6% *Accuracy*, 96% *Normal Recall*, 89% *U2R Recall*

```
Test Accuracy: 0.8370
              precision    recall  f1-score   support

           0       1.00      0.84      0.91      9711
           1       0.02      0.81      0.04        37

    accuracy                           0.84      9748
   macro avg       0.51      0.82      0.47      9748
weighted avg       1.00      0.84      0.91      9748

Confusion Matrix:
[[8129 1582]
 [   7   30]]
```

```
Test Accuracy: 0.9569142388182191
              precision    recall  f1-score   support

           0       1.00      0.96      0.98      9711
           1       0.07      0.89      0.14        37

    accuracy                           0.96      9748
   macro avg       0.54      0.92      0.56      9748
weighted avg       1.00      0.96      0.97      9748

[[9295  416]
 [   4   33]]
```

Model Performance Comparison

Recurrent Neural Network vs Prototypical Networks

RNN: *66.84% Accuracy*, *67% Normal Recall*, *100% U2R Recall*

Prototypical Network: *95.6% Accuracy*, *96% Normal Recall*, *89% U2R Recall*

```
Accuracy: 66.84%
Confusion Matrix:
 [[6479 3232]
 [   0   37]]
Classification Report:
              precision    recall  f1-score   support

           0       1.00      0.67      0.80      9711
           1       0.01      1.00      0.02        37

    accuracy                           0.67      9748
   macro avg       0.51      0.83      0.41      9748
weighted avg       1.00      0.67      0.80      9748
```

```
Test Accuracy: 0.9569142388182191
              precision    recall  f1-score   support

           0       1.00      0.96      0.98      9711
           1       0.07      0.89      0.14        37

    accuracy                           0.96      9748
   macro avg       0.54      0.92      0.56      9748
weighted avg       1.00      0.96      0.97      9748

[[9295  416]
 [   4   33]]
```

Model Performance Comparison

K-Nearest Neighbours vs Prototypical Networks

KNN: *80.54% Accuracy*, *81% Normal Recall*, *76% U2R Recall*

Prototypical Network: *95.6% Accuracy*, *96% Normal Recall*, *89% U2R Recall*

```
Test Accuracy: 0.8054

Classification Report:
              precision    recall  f1-score   support

           0       1.00      0.81      0.89      9711
           1       0.01      0.76      0.03        37

    accuracy                           0.81      9748
   macro avg       0.51      0.78      0.46      9748
weighted avg       1.00      0.81      0.89      9748

Confusion Matrix:
[[7823 1888]
 [   9   28]]
```

```
Test Accuracy: 0.9569142388182191
              precision    recall  f1-score   support

           0       1.00      0.96      0.98      9711
           1       0.07      0.89      0.14        37

    accuracy                           0.96      9748
   macro avg       0.54      0.92      0.56      9748
weighted avg       1.00      0.96      0.97      9748

[[9295  416]
 [   4   33]]
```

Work Comparison

An Intrusion Detection Method Using Few-Shot Learning
YINGWEI YU & NAIZHENG BIAN

Authors have performed multi-class classification using all 5 classes in the dataset.

They achieved overall accuracy of 92.34% using less than 1% of train data.

Our model achieved 95.69% accuracy using less than 0.080% of train data.

Methodology	Training dataset	KDDTest+ Accuracy (%)
J48 [20]	20% KDDTrain+	81.05
Naive Bayes [20]	20% KDDTrain+	76.56
NB Tree [20]	20% KDDTrain+	82.02
Random Forest [20]	20% KDDTrain+	80.67
Random Tree [20]	20% KDDTrain+	81.59
Multi-layer perceptron [20]	20% KDDTrain+	77.41
SVM [20]	20% KDDTrain+	69.52
RNN [26]	Full KDDTrain+	83.28
Fuzzy based semi-supervised learning [2]	Full KDDTrain+(10% labled data)	84.12
CBR-CNN [4]	80% KDDTrain+	89.41
Our model	**Less than 1% KDDTrain+**	**92.34**

Its Limitation

An Intrusion Detection Method Using Few-Shot Learning
YINGWEI YU & NAIZHENG BIAN

Trained their model with all 5 classes which makes it difficult to understand how the model would react with classes that has limited training data such as U2R.

Used “Testing” data for support set. It means that they have used some data from their testing data to train the model. Later, they tested the model on the same test data. Thus, it cannot be determined how the model would perform on unseen data.

Used over-sampling technique on the classes that has limited samples such as U2R. Thus, again hard to tell how the model would perform on limited samples.

Conclusion

Prototypical Network is able to achieve higher accuracy overall.

It works better with limited data compared to other machine learning models.

It has low **false positive rate**, meaning there is less chance of normal class being falsely classified as U2R as opposed to other models which has higher false rate.

As it uses less data, it is effective on tasks for which it is hard to obtain training data such as *Intrusion Detection Systems*.

Challenges

The novel nature of the algorithm and insufficient experimentation have hindered the implementation of most prototypical networks in real-world scenarios.

Although few-shot learning is widely used in Computer Vision, its effectiveness in other domains such as NLP, Cognitive Computing, and other classification tasks may be limited.

Thank You

Mohammed Aawez **Mansuri**