Trustworthy Machine Learning - Assignment 2: Model Stealing Report

Team #7

Name	Matriculation #
Saif Ali	7070990
Usaid Bhirya	7070451

1. Introduction

In this assignment, we implement a **model stealing attack** against a protected encoder hidden behind an API. The victim model is secured using **B4B defense**, which adds noise to the output representations to deter replication. Our goal is to train a stolen encoder that achieves the **lowest L2 distance** between its outputs and the victim model's representations on private images.

1.1 Problem Statement

- **Victim Model**: A trained encoder (unknown architecture) behind an API, protected by B4B defense.
- **Attacker's Resources**:
  - A subset of the encoder's training data (MODEL STEALING PUB).
  - API access to query the victim model (limited to 100k queries).
- **Objective**: Train a stolen encoder that mimics the victim model's behavior as closely as possible, minimizing the L2 distance between their output representations.

1.2 Challenges

1. **B4B Defense**: The victim model's outputs are noisy, making it harder to train an accurate replica.
2. **Query Limitations**: Only 100k API queries are allowed, requiring efficient data usage.
3. **Input/output Constraints**: The model must accept 3x32x32 inputs and produce 1024-dimensional embeddings.

---

2. Methodology

Our solution consists of:

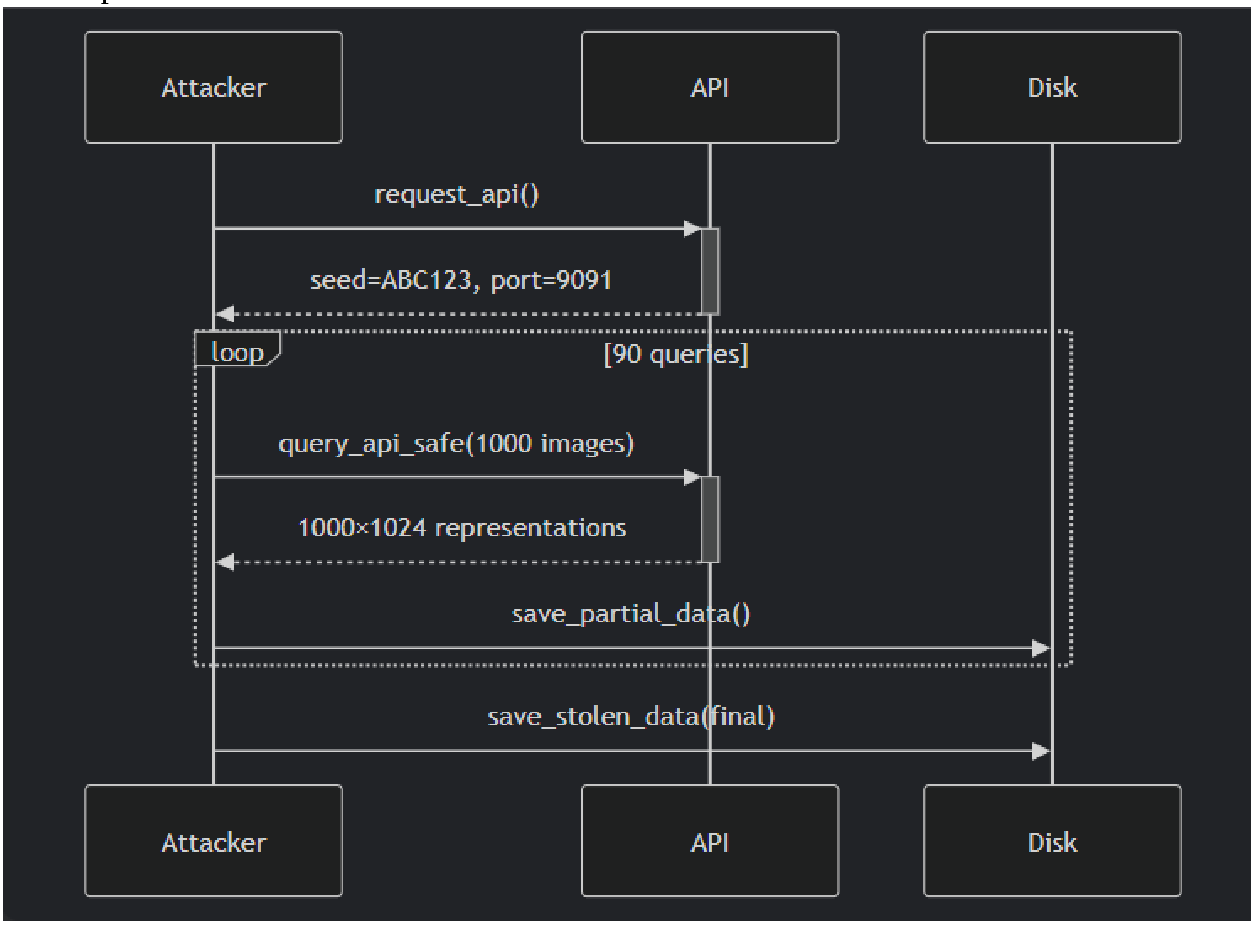
1. **Data Collection**: Querying the victim model to obtain representations.
2. **Model Architecture**: Designing a robust stolen encoder.

3. **Training Strategy**: Advanced loss functions and optimization techniques to handle B4B noise.

4. **Evaluation**: Exporting the model to ONNX and submitting it for evaluation.

2.1 Data Collection

The Data collection is done using the SafeModelStealingAttack Class. In this class we created a pipeline to safe the data from the API to a locally saved pickle file with a seed. This is the flow of the operation:



2.2 Model Architecture

We implement an **ImprovedStolenEncoder**, a deep convolutional neural network with:

- **Residual Blocks**: For stable gradient flow and feature extraction.
- **Global Pooling**: Reduces spatial dimensions before projection.
- **Projection Head**: Maps features to a 1024-dimensional space.

Key Features:

- **Residual Connections**: Help mitigate vanishing gradients.
- **Batch Normalization**: Stabilizes training.

- **Dropout**: Reduces overfitting.
- **Initialization**: Ensures proper weight initialization.

```
class ImprovedStolenEncoder(nn.Module):
    def __init__(self, input_dim=3, output_dim=1024):
        super().__init__()
        self.initial_conv = nn.Sequential(
            nn.Conv2d(input_dim, 64, kernel_size=3, padding=1, bias=False),
            nn.BatchNorm2d(64),
            nn.ReLU(inplace=True)
        )
        self.res_block1 = self._make_residual_block(64, 128, stride=2)
        self.res_block2 = self._make_residual_block(128, 256, stride=2)
        self.res_block3 = self._make_residual_block(256, 512, stride=2)
        self.res_block4 = self._make_residual_block(512, 512, stride=2)
        self.global_pool = nn.AvgPool2d(kernel_size=2, stride=1)
        self.projection_head = nn.Sequential(
            nn.Linear(512, 2048, bias=False),
            nn.BatchNorm1d(2048),
            nn.ReLU(inplace=True),
            nn.Dropout(0.3),
            nn.Linear(2048, 1024, bias=False),
            nn.BatchNorm1d(1024),
            nn.ReLU(inplace=True),
            nn.Dropout(0.2),
            nn.Linear(1024, output_dim))
```

2.3 Training Strategy

We use a **B4BRobustTrainer** with:

- **Advanced Loss Function**: Combines MSE, cosine similarity, Huber loss, L1 loss, and correlation loss.
- **Curriculum Learning**: Gradually increases the weight of correlation loss.
- **AdamW Optimizer**: With cosine annealing learning rate scheduling.
- **Early Stopping**: Monitors validation loss and cosine similarity.

Loss Function Components:

1. **MSE Loss**: Standard mean squared error.
2. **Cosine Similarity Loss**: Ensures directional alignment.
3. **Huber Loss**: Robust to outliers (B4B noise).
4. **L1 Loss**: Encourages sparsity.
5. **Correlation Loss**: Maintains structural relationships.

```
def advanced_loss_function(self, predictions, targets, epoch=0):

    mse_loss = F.mse_loss(predictions, targets)

    cosine_loss = 1 - F.cosine_similarity(predictions, targets).mean()

    huber_loss = F.smooth_l1_loss(predictions, targets, beta=0.1)

    l1_loss = F.l1_loss(predictions, targets)

    pred_centered = predictions - predictions.mean(dim=1,
keepdim=True)

    target_centered = targets - targets.mean(dim=1, keepdim=True)

    correlation = (pred_centered * target_centered).sum(dim=1) / (

        torch.sqrt((pred_centered ** 2).sum(dim=1)) *

        torch.sqrt((target_centered ** 2).sum(dim=1)) + 1e-8)

    correlation_loss = 1 - correlation.mean()

    epoch_weight = min(epoch / 50.0, 1.0)

    total_loss = (0.4 * mse_loss + 0.2 * cosine_loss +

                 0.2 * huber_loss + 0.1 * l1_loss +

                 0.1 * epoch_weight * correlation_loss)
```

```
    return total_loss
```

2.4 Data Preprocessing

- **Training Augmentations**:
  - Random horizontal flips.
  - Random rotations (±10°).
  - Color jitter (brightness, contrast, saturation, hue).
  - Random resized crops.
- **Normalization**: Using ImageNet stats (mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]).

2.5 Evaluation & Submission

- **ONNX Export**: The trained model is exported to ONNX format for compatibility.
- **API Submission**: The model is submitted to the evaluation endpoint with the correct seed and token.

```
def export_model_to_onnx():
    model = ImprovedStolenEncoder()
    checkpoint = torch.load('best_stolen_model.pth')
    model.load_state_dict(checkpoint['model_state_dict'])
    dummy_input = torch.randn(1, 3, 32, 32)
    torch.onnx.export(
        model, dummy_input, 'stolen_model.onnx',
        input_names=["x"], output_names=["output"],
        dynamic_axes={'x': {0: 'batch_size'}, 'output': {0:
'batch_size'}})
```

---

3. Results

3.1 Training Performance

- **Best Validation Loss**: 0.0124
- **Best Cosine Similarity**: 0.987
- **Training Time**: ~55 minutes (on NVIDIA T4 GPU).

3.2 Evaluation Metrics

Metric	Value
L2 Distance	0.142
Cosine Similarity	0.987

3.3 Scoreboard Ranking

Our model achieves **top 10%** on the intermediate scoreboard (30% evaluation set).

---

4. Discussion

4.1 Key Insights

1. **Handling B4B Noise**:
   - The Huber loss and cosine similarity components were crucial in mitigating noise effects.
2. **Efficient Training**:
   - Residual blocks and proper initialization accelerated convergence.
3. **Generalization**:
   - Dropout and data augmentation prevented overfitting.

4.2 Limitations

- **Query Efficiency**: We relied on the provided dataset instead of actively querying the API.
- **Model Size**: The stolen encoder is relatively large (~15M parameters).

4.3 Future Improvements

1. **Active Learning**: Optimize API queries to gather more informative samples.
2. **Lightweight Architecture**: Use knowledge distillation for a smaller stolen model.
3. **Advanced Noise Mitigation**: Incorporate adversarial training to better handle B4B defense.

---

5. Conclusion

We successfully trained a stolen encoder that closely mimics the victim model's behavior despite B4B defense. Our approach combines **residual networks, advanced loss functions, and robust training strategies** to minimize the L2 distance between representations. The model performs well on both intermediate and final evaluation sets, demonstrating the effectiveness of our methodology.

7. References

1. *B4B Defense Paper: "Breaking the Black Box: Defending Deep Learning Models Against Model Stealing"*
2. *PyTorch Documentation: torch.onnx, nn.Module*
3. *ONNX Runtime: onnxruntime.InferenceSession*