TML ASSIGNMENT 2: MODEL STEALING

Team Number: 10

This report details the implementation and execution of a model stealing. The objective is to train a replica of a victim encoder by querying its API and to achieve the lowest L2 distance of output representations on private images between the stolen copy and the attacked encoder. The victim encoder is protected using B4B defense. The implemented solution, along with supporting files, is uploaded to the team's GitHub repository.

1. Goal of the Assignment

The primary goal of this assignment is to perform a Model Stealing attack against a trained encoder of unknown architecture, which is hidden behind an API and protected by B4B defense. Model stealing occurs when an attacker sends queries to a victim model via an API and uses the outputs (labels or embeddings) to train a replica of the victim model. As an attacker, the objective is to train a stolen copy of this victim encoder and minimize the L2 distance of output representations on a private image dataset between the stolen model and the victim encoder.

2. Data Accessible to the Adversary

For this attack, the following resources were provided to the adversary:

• A trained encoder of unknown architecture hidden behind an API and protected using B4B noising.

• A subset of the encoder's training data (ModelStealingPub.pt), which has a structure.

• An API to send queries to the victim encoder.

3. Approach Used

The implemented model stealing strategy involves several key steps:

i. Requesting the API The first step is to gain access to the encoder by requesting the API. This is done by sending a GET request to a specified URL with a token in the headers. The API returns important information: a **seed** and a **port**1013. The seed is crucial for submission, as providing an incorrect seed will result in an L2 distance on par with a random submission. The port is necessary for sending subsequent queries to the encoder. A short delay (e.g., 10 seconds) is often included to allow the server to boot up.

ii. Querying the API Once access to the API is established, queries can be sent to the hidden encoder. The model_stealing function is used to handle these queries.

- **Image Processing:** Images are converted to PNG format, base64 encoded, and compiled into a JSON payload.

• **Query Structure:** The API expects 1000 images per query. The provided code demonstrates a strategy of performing **multiple batches of 1000 images** (e.g., 10 batches for a total of 10,000 images).

• **Rate Limiting/Delays:** To avoid hitting the query limit too quickly or for stability, the example code includes a **60-second waiting period** between successive batches of queries.

• **Output:** The API returns a list of "representations" (embeddings) for the input images.

•**Data Storage:** All successfully queried images and their corresponding representations are saved into a pickle file (e.g., full_out.pickle) for subsequent use in training the stolen model.

iii. Model Definition (Stolen Encoder) A ResNetEncoder class is defined to serve as the architecture for the stolen model.

• It utilizes a **resnet18 backbone**.

• The final fully connected layer of resnet18 is replaced with a new nn.Linear layer to output representations of **size 1024**, matching the output dimensionality of the victim encoder.

• Before feeding images to the model, a transformation pipeline is applied, including resizing to 32x32, random horizontal flipping, color jitter, and conversion to a PyTorch tensor.

iv. Training the Stolen Model With the output representations obtained from the victim encoder, a new encoder (the stolen copy) is trained.

• **Normalization:** A l2_normalize function is applied to both the model's outputs and the target representations to normalize them. This is a common practice when learning embeddings to ensure they lie on a unit sphere.

• **Optimizer:** The Adam optimizer is used with a learning rate of 1e-3 and weight decay of 1e-4.

• **Loss Function:** nn.SmoothL1Loss is employed as the criterion, which is a robust loss function.

• **Training Loop:** The model is trained for 10 epochs. During training, the optimizer's gradients are zeroed, the model makes predictions, the loss is calculated, and backpropagation is performed to update model weights.

v. Export to ONNX For submission, the stolen model must be provided in **ONNX format**.

• The torch.onnx.export function is used to convert the trained PyTorch model to an ONNX file (e.g., stolen_model.onnx).

• **Input and Output Specifications:** It's crucial to correctly specify the input name ("x"), output name ("output"), and the expected input shape (1, 3, 32, 32) during export to ensure compatibility with the evaluation endpoint. The output dimensionality must be 1024.

vi. Validation and Submission Before final submission, local validation checks are recommended.

• **Validation Checks:** The onnxruntime library is used to load the exported ONNX model and verify its input/output shapes, ensuring it matches the required 3x32x32 input and 1024 output.

• **Submission:** The ONNX model file is then submitted to the evaluation server via an HTTP POST request. The submission requires the ONNX file itself, the team's token, and critically, the **seed** obtained during the initial API launch. An incorrect seed will lead to a very high L2 error.

4. Justification for Implementation Choices

The selected approach and specific choices in the implementation are justified as follows:

• **ResNet18 as Stolen Model Architecture:** Given that the victim encoder's architecture is unknown, resnet18 is a common and effective choice for image-based tasks. Its versatility and ability to learn complex features make it a suitable candidate for replicating the victim's embedding behavior. The final layer is modified to ensure the output dimensionality matches the victim encoder's 1024-size representations.

• **L2 Normalization of Representations:** Normalizing the output representations of both the stolen model and the target representations (l2_normalize) before calculating the loss is important in representation learning. This forces the embeddings onto a unit hypersphere, which can help in learning a better, more comparable embedding space and improve the stability of training.

• **nn.SmoothL1Loss:** This loss function is chosen over MSELoss (L2 loss) or L1Loss. SmoothL1Loss is less sensitive to outliers than MSELoss and provides a smoother gradient near zero, which can lead to more stable training, especially when dealing with potentially noisy outputs from a B4B-defended model.

• **Image Augmentations:** The use of transforms.RandomHorizontalFlip() and transforms.ColorJitter() during the data loading process introduces variability into the training data for the stolen model. These standard data augmentation techniques help the stolen model generalize better to unseen images and become more robust, which is essential for achieving a low L2 distance on private images.

• **Multi-Batch Querying with Delays:** The strategy of querying the API in batches of 1000 images with a waiting period (e.g., 60 seconds) between batches is a pragmatic choice. It manages the query limit (100k queries total), allows for the collection of a substantial dataset for training the stolen model, and potentially avoids overwhelming the API server or hitting rate limits too quickly.

• **ONNX Export and Validation:** The requirement to export the model to ONNX and the inclusion of validation checks using onnxruntime ensures that the submitted model adheres to the specified interface and can be correctly loaded and evaluated by the grading system. This is crucial for successful submission.

5. Results

The primary metric for success in this assignment is the L2 distance between the submitted model's representations and the victim model's representations on a private dataset.

6. Other Ideas on Implementation and Leveraged Concepts

Our approach leverages several core concepts pertinent to trustworthy machine learning:

• **Model Stealing as an Attack Vector:** The assignment directly demonstrates the process of model stealing, illustrating how an attacker can replicate a proprietary model by observing its black-box behavior via an API.

• **Defense Mechanism (B4B):** The presence of B4B defense on the victim encoder implicitly challenges the attacker to develop strategies that can overcome this protection. This might involve robust training techniques or larger datasets to average out the noising effects.

• **Representation Learning:** The objective of matching output representations (embeddings) rather than specific classifications highlights the importance of learning effective feature extractors or encoders.

• **Practical API Interaction:** The assignment simulates real-world interaction with a deployed model via an API, including considerations for query limits, data formats, and response handling.

• **Model Deployment Formats (ONNX):** Requiring submission in ONNX format provides practical experience with model serialization and deployment standards, essential for sharing and utilizing machine learning models across different platforms.

7. Files and Their Descriptions

The key files relevant to this submission include:

• highscore.py: The main Python script implementing the model stealing pipeline, from API querying to model training and ONNX export.

• ModelStealingPub.pt: The public dataset provided for the assignment, containing images and their associated data.

• full_out.pickle: A cached file generated by highscore.py during execution, containing the images queried from the API and their corresponding output representations from the victim encoder.

• stolen_model.onnx: The final ONNX version of the trained stolen encoder, ready for submission to the evaluation server.

• README.md: A required documentation file that points to the most important code files and pieces of the solution.

• ModelStealing_Report.pdf: This detailed report describing the solution and implementation, submitted in PDF format.