TML Assignment # 2

Encoder Model Stealing Attack

Implementation and Workflow

In this assignment, we implemented a **model-stealing attack** on a target image encoder exposed via an API. The overall workflow is as follows:

1. retrieve session parameters (seed and port) from the server.
2. prepare the query dataset by loading the provided images and applying random augmentations along with some preprocessing (like conversion of grayscale images in the dataset to RGB and normalization).
3. query the target encoder in batches to obtain output embeddings.
4. train a surrogate ("stolen") encoder to match those embeddings.
5. export the surrogate model and submit it for evaluation. Each step is automated in code files, making the process **reproducible** under the given seed.

Throughout, we applied some measures to cope with the server's *B4B defense* (an active anti-stealing mechanism) and API rate limits. Below we document each stage in detail, with code snippets from the corresponding scripts.

1. Retrieving Model Details (Seed and Port)

The first step is to obtain the API session parameters. In `model_details.py` we send a GET request to the `/stealing_launch` endpoint using our assignment token. The server returns a JSON object containing a **seed** and a **port** number for our session. For example:

```
TOKEN = "54614611"
answer = {'seed': 18632002, 'port': '9054'}
```

We store these values for later use. Using the provided seed ensures **reproducibility**: for instance, when we shuffle image indices, we use a deterministic RNG seeded by this value.

2. Data Preparation, Preprocessing and Augmentation

Next, we prepare the image dataset. We load the provided `ModelStealingPub.pt` file, which contains our initial set of images. To **enlarge the dataset** and **mitigate anti-stealing defenses**, we apply one random augmentation per image. The augmentations include horizontal/vertical flips, small rotations, and color jitter. In code (in a separate script), we define the transforms and apply exactly one of them to each image:

```
single_transforms = [
    T.RandomHorizontalFlip(p=1.0),
    T.RandomVerticalFlip(p=1.0),
    T.RandomRotation(degrees=15),
    T.ColorJitter(brightness=0.3, contrast=0.3, saturation=0.3, hue=0.1),
]
```

Using augmentations provides two benefits.

1. It roughly **doubles the number of images** for training the surrogate model.
2. Second, it helps *evade the B4B defense*. Recent work (Han *et al.*, NeurIPS 2023) shows that the B4B mechanism tracks how much of the embedding space a user's queries cover, adding noise as coverage grows. By applying random augmentations, each query appears slightly different, which prevents triggering coverage-based penalties.

Crucially, it is known that the encoder produces **similar embeddings for an image and its augmented versions** so using augmented images for training does not hurt accuracy. This idea is also used in contemporary attacks: Chen *et al.* point out that one can send a single image to the victim model and train the stolen model on many augmented views of it, since the encoder's outputs are invariant to such. We leverage this fact to increase data without additional unique queries.

Preprocessing steps:

1. **RGB Conversion:** Out of 12,000 images, 1,356 were grayscale instead of RGB. Since the model requires inputs with dimensions (3, 32, 32), these grayscale images were converted to RGB by replicating the single channel across three channels to ensure consistent input shape.

2. **Normalization:** All images were normalized using `T.Normalize([0.2980, 0.2962, 0.2987], [0.2886, 0.2875, 0.2889])` to standardize pixel intensity values, which improves model convergence and training stability.

3. Querying the Target Encoder for Embeddings

With the dataset ready, we query the target encoder to collect embeddings. We combine the original and augmented images into one list. To ensure reproducibility, we create a shuffled index array using the given seed (`seed = int(answer['seed'])`). We then iterate over the images in **batches of 1000**. For each batch, we do the following:

- Extract the subset of images by index.
- **Encode each image as PNG and base64** (as required by the API).
- Send a GET request to api with our token and the JSON payload of images.
- Parse the returned JSON to get a list of embedding vectors.
- Store the embeddings and corresponding indices in lists.
- Save progress to `embeddings.pickle` after each batch to avoid data loss.
- Wait **60 seconds** before the next batch to respect the API rate limits and avoid triggering B4B.

An excerpt of the query loop in **embedding_extraction_from_encoder_api.py**:

```
for start in range(0, min(len(imgs), max_queries), chunk_size):
    batch_idx = shuffled_indices[start:start+chunk_size]
    batch_imgs = [imgs[i] for i in batch_idx]
    reps = query_api(batch_imgs, port)  # calls the API
    all_reps.extend(reps)
    all_idx.extend(batch_idx.tolist())
    with open("embeddings.pickle", "wb") as f:
        pickle.dump({"indices": all_idx, "reps": all_reps}, f)
    time.sleep(60)
```

This process collects a large set of input embeddings from the target encoder. The 60-second wait and random ordering help comply with the server's defense constraints. All retrieved embeddings (1024-dimensional vectors) are saved in `embeddings.pickle` for use in training the surrogate.

4. Training the Surrogate (Stolen) Encoder

After gathering (image, embedding) pairs, we train our stolen model to mimic the target encoder's embedding space. We first load `embeddings.pickle` to retrieve the (index, embedding) pairs and representation vectors (as PyTorch tensors). Jjust as in the data collection phase. During training, we apply the same normalization transform to ensure consistency. Our dataset is fed into a `DataLoader` with a **batch size of 512**. This larger batch size was selected to **accelerate training** without sacrificing convergence stability (validated during hyperparameter tuning).

Model architecture

We design the surrogate model using a **ResNet-18 backbone (no pre-training)**, carefully modified to suit **32×32 images**:

- The original ResNet-18 uses a `7×7` convolution (stride=2) and an aggressive max-pool — unsuitable for small images.
- We replace the first convolution with a `3×3` kernel (`stride=1`, `padding=1`) and **remove the initial max-pool** (replaced by `nn.Identity()`).
- The model processes the image through all layers up to the global average pooling.
- The resulting 512-dimensional feature vector is passed through a custom fully connected layer projecting to 1024-dimensional embeddings to match the target encoder's output size.

```
base = models.resnet18(weights=None)
base.conv1 = nn.Conv2d(3, 64, kernel_size=3, stride=1, padding=1, bias=False)
# remove initial large conv
base.maxpool = nn.Identity()  # remove unnecessary pooling
self.encoder = nn.Sequential(*list(base.children())[:-1])  # remove classifier
self.fc = nn.Linear(512, 1024)  # project to target dimension
```

This architecture is **purpose-built** for small image inputs while preserving ResNet's depth and capacity.

Loss & optimizer

We train the model to **directly match the target encoder's embeddings** using **Mean Squared Error (MSE) loss**. This choice aligns with standard practice in encoder-stealing, where minimizing the L2 distance between surrogate and target embeddings ensures the surrogate approximates the target representation space.

We optimize using:

- **Adam optimizer** (`lr=1e-3`) for its adaptive learning rate and robustness.
- **Weight decay (1e-4)** to provide regularization and prevent overfitting.

The model is trained for **40 epochs**, which was determined via hyperparameter tuning to achieve stable convergence while avoiding unnecessary overfitting.

This setup — small-input-resilient ResNet, MSE loss on embeddings, Adam with weight decay, batch size 512, and 40 epochs — reflects a **deliberate design** to efficiently and effectively learn a surrogate encoder that replicates the target's embedding space.

Results

Average L2 distance over the validation set was around 4.10 and on the test data it was around 5.785 when I submitted my model.

5. Exporting the Stolen Model and Submission

Once training is complete, we export the trained model to ONNX format for submission. We use `torch.onnx.export` with a dummy input to define the graph:

Before submission, our evaluation script also reloads the ONNX bytes to ensure validity. Finally, we send the ONNX file to the `/stealing` endpoint using our token and seed:

A successful response indicates our model has the correct output shape and is evaluated against the target. The entire process – from querying to submission – is scripted, making the experiment **reproducible** given the same seed and data.

Challenges and Considerations

- **API Constraints and B4B Defense:** The target employs an *active defense* ("B4B") that adds noise based on how much of the embedding space a user covers. To avoid triggering this, we limited each batch to 1000 images, waited 60 seconds between batches, and used random augmentations so that each query appears unique. These measures slow down the attack but are necessary to stay below the defense threshold.

- **Data Augmentation Effectiveness:** We relied on the encoder's invariant to transformations. If this invariance were weaker, our surrogate model might receive mismatched labels. Fortunately, literature (and our experiments) confirm that random flips/rotations produce very similar embeddings, justifying this strategy.

- **Model Capacity and Architecture:** We assumed a ResNet-like encoder. By customizing ResNet-18 (small first conv, no pooling), we provided enough capacity to learn the mapping. However, if the target used a significantly different architecture or larger model, our surrogate might underfit. In practice, we monitor training loss and, if needed, could try deeper networks.

- **Normalization:** The provided mean/std values were applied to inputs so that our model sees images similarly normalized to the target's training data. Mismatched normalization can degrade performance.

- **Reproducibility:** We used the given seed to control randomness (data shuffling, augmentation sampling). We also saved intermediate results (like `embeddings.pickle`) so we can resume without restarting from scratch.

Overall, the code (in `model_details.py`, `embedding_extraction from_encoder_api.py` , and the training script) is organized to reflect each step. By following the sequence of loading data, augmenting, querying, training, and exporting, one can reproduce the stolen encoder. The approach is backed by prior work: minimizing embedding loss is a standard attack, and data augmentation is a known trick in encoder stealing. Our documentation and code snippets above detail the implementation choices and justify our methodology.

References

Key ideas in our approach are supported by recent literature. For example, Chen *et al.* note that an encoder yields similar embeddings for an image and its augmented views, and Sha *et al.* demonstrate that minimizing MSE between target and surrogate embeddings effectively steals an encoder. The B4B defense mechanics are described by Han *et al*, which motivated our query strategy and augmentations.

Github Repo Link :- https://github.com/MuhammadSaqib001/TML25_A2_19