REPORT

Simulation Question 4.

This code serves as a baseline model for the CIFAR-10 classification task, using 80% of the training data for training and the remaining 20% for validation.

1. **Set the device**: The code first checks if a GPU is available and sets the device accordingly (either 'cuda' or 'cpu').
2. **Define the CIFAR10Classifier model**: The code defines the CIFAR10Classifier model, which is a PyTorch module that consists of two convolutional layers, two dropout layers, and two fully connected layers.
3. **Load the CIFAR-10 dataset**: The code loads the CIFAR-10 dataset using the datasets.CIFAR10 function from the torchvision library. The dataset is downloaded and extracted to the ./data directory if it doesn't already exist.
4. **Split the training data into training and validation sets**: The code splits the CIFAR-10 training dataset into a training set (80% of the data) and a validation set (20% of the data). This is done using the Subset function from the torch.utils.data library.
5. **Create data loaders**: The code creates two DataLoader objects: one for the training set and one for the validation set. These DataLoaders are responsible for feeding the data to the model during training and validation.
6. **Initialize the model and move it to the device**: The code creates an instance of the CIFAR10Classifier model and moves it to the device (either CPU or GPU) specified earlier.
7. **Define the loss function and optimizer**: The code defines the loss function (Cross-Entropy Loss) and the optimizer (Adam) that will be used to train the model.
8. **Train the model**: The code trains the model for 20 epochs. In each epoch, the following steps are performed:
    - Training:
        - The model is set to training mode using `model.train()`.
        - The training loss is calculated by passing the training inputs and labels through the model, computing the loss, and backpropagating the gradients.
        - The optimizer is used to update the model's parameters.
    - Validation:
        - The model is set to evaluation mode using `model.eval()`.
        - The validation loss and accuracy are calculated by passing the validation inputs and labels through the model and computing the loss and accuracy.
    - The training loss, validation loss, and validation accuracy are printed at the end of each epoch.

Simulation Question 5.

1. **Set the device**:
    - The code checks if a GPU is available and sets the device accordingly, either to 'cuda' (GPU) or 'cpu'.
2. **Load the CIFAR-10 dataset**:
    - The code loads the CIFAR-10 dataset, which is a commonly used image classification dataset.
    - The data is transformed using the `transforms.Compose` function, which applies the following transformations:
        - transforms.ToTensor(): Converts the image to a PyTorch tensor.
        - transforms.Normalize(): Normalizes the pixel values to the range [-1, 1] using the specified mean and standard deviation.
3. **Split the training data into training and validation sets**:
    - The code splits the training data into a training set (80%) and a validation set (20%).
    - It shuffles the indices of the training data and then selects the first 80% for the training set and the remaining 20% for the validation set.
    - The Subset class is used to create the training and validation subsets from the original dataset.
4. **Create data loaders**:
    - The code creates PyTorch DataLoader objects for the training and validation sets.
    - The DataLoader handles the data loading and batching process.
    - The batch size is set to 64, and the number of worker threads is set to 2.
    - The training data loader is set to shuffle the data, while the validation data loader keeps the order.
5. **Initialize the model and move it to the device**:

    - The code initializes the `CIFAR10Classifier` model and moves it to the specified device (GPU or CPU).
6. **Define the loss function and optimizer**:
    - The code sets up the loss function as `nn.CrossEntropyLoss()`, which is a commonly used loss function for classification tasks.
    - The optimizer is set to `optim.Adam()` with a learning rate of 0.001.
7. **Initialize the Privacy Engine**:
    - The code initializes the Privacy Engine, which is a tool for applying differential privacy to the model and training process.
8. **Make the model, optimizer, and data loader private**:
    - The code uses the Privacy Engine to make the model, optimizer, and training data loader private.
    - The `make_private()` method applies the necessary modifications to ensure differential privacy, including adding noise to the gradients.
    - The noise multiplier is set to 0.1, and the maximum gradient norm is set to 1.0.
9. **Train the model**:
    - The code trains the model for 15 epochs.
    - During each epoch, the model is set to training mode, and the training loss is calculated and accumulated.
    - After the training, the model is set to evaluation mode, and the validation loss and accuracy are calculated.
    - The current privacy budget (epsilon) is also calculated and printed at the end of each epoch.
10. **Save the trained model**:
    - The final trained model is saved to a file named 'ModifiedModel.pth'.

Simulation Question 8.Model1

The code in this section defines the architecture and training process for a shadow model and an attacker model.

1. **Define the shadow model architecture**:
    - The shadow model is defined as a PyTorch `nn.Sequential` module.
    - It consists of two linear layers with a ReLU activation function in between.
    - The input size and output size of the model are specified by the `input_size` and `output_size` variables, respectively.
2. **Define the attacker model architecture**:
    - The attacker model is also defined as a PyTorch `nn.Sequential` module.
    - It has a similar structure to the shadow model, with two linear layers and a ReLU activation function.
    - However, the output layer of the attacker model has 10 units, representing the output of the 2 classes: member or non-member.
3. **Define the optimizers and loss functions**:
    - For the attacker model, the optimizer is set to `optim.Adam()` with a learning rate of 0.001, and the loss function is set to `nn.BCELoss()`.
    - For the shadow model, the optimizer is also set to `optim.Adam()` with a learning rate of 0.001, and the loss function is set to `nn.CrossEntropyLoss()`.
4. **Move the models to the device**:
    - Both the attacker model and the shadow model are moved to the specified device (GPU or CPU).
5. **Train the models**:
    - The training loop runs for `num_epochs` iterations.
    - During each iteration:
        - Both the attacker model and the shadow model are set to training mode.
        - For each batch of data from the `attacker_train_loader`:
            - The features and labels are moved to the device.
            - The attacker model's optimizer is zeroed, and the attacker model's output is used to calculate the attacker loss, which is then backpropagated and the optimizer is updated.
            - The shadow model's optimizer is zeroed, and the shadow model's output is used to calculate the shadow loss, which is then backpropagated and the optimizer is updated.

The purpose of this code is to train a shadow model and an attacker model simultaneously. The shadow model is trained on the same task as the target model, while the attacker model is trained to distinguish between the target model's outputs and the shadow model's outputs, with the goal of identifying whether a given input was used to train the target model (member inference attack).

The specific details of the attack and the training process would depend on the context and the overall implementation of the attack. This code snippet provides a basic structure for the shadow model and attacker model, but the complete implementation of the attack would likely involve additional steps and considerations.

Simulation Question 8.Model2

This code snippet is related to a membership inference attack, which is a type of privacy attack that aims to determine whether a given data sample was part of the training dataset used to train a target machine learning model.

1. **Train shadow models**:
    - The code trains multiple (5 in this case) shadow models using the `LogisticRegression` classifier from scikit-learn.
    - The shadow models are trained on the same task as the target model, but on different datasets.
    - The `train_features` and `train_labels` tensors are used to train the shadow models.
    - The shadow models are stored in the `shadow_models` list.
2. **Membership inference attack**:
    - The code initializes a single attacker model using the `LogisticRegression` classifier from scikit-learn.
    - It concatenates the `train_features`, `other_features`, and `test_features` tensors into a single tensor `all_features`, and similarly concatenates the corresponding labels into `all_labels`.
    - The `train_test_split()` function from scikit-learn is used to split the `all_features` and `all_labels` tensors into training and validation sets.
3. **Train attacker model**:
    - The attacker model is trained on the training data (`train_data` and `train_labels`) using the `fit()` method.

The purpose of this code is to train a membership inference attacker model that can distinguish between samples that were used to train the target model (members) and samples that were not used to train the target model (non-members).

The key steps in a membership inference attack are:

1. **Training shadow models**: The shadow models are trained on the same task as the target model, but on different datasets. This helps the attacker model learn the differences between the target model's outputs on members and non-members.
2. **Extracting features and labels**: The code concatenates the features and labels from the training, other, and test sets to create a comprehensive dataset for training the attacker model.
3. **Training the attacker model**: The attacker model is trained on the training and validation sets using the `LogisticRegression` classifier. The attacker model learns to distinguish between member and non-member samples based on the differences in the target model's outputs.

After training the attacker model, it can be used to predict whether a given sample was part of the target model's training dataset or not. This information can be used to infer sensitive information about the training data and the individuals it represents, which is the purpose of a membership inference attack.