University of Groningen

Research Internship

Secure GAN trained via Federated Learning

Student:
Tom Maguire *(s4741986)*

Supervisor:
F. Turkmen
A.R. Ghavamipour

July 22, 2022

1 Introduction

The aim of this project was to create a generative adversarial network (GAN) which is trained using federated learning and is safe against attacks against the gradients being shared. To achieve this the idea was to implement privacy preserving techniques on top of a GAN trained via federated learning. For the rest of this report a GAN trained via federated learning is referred to as a federated GAN.

There is not an implementation of a federated GAN with privacy preserving techniques currently available. However, there is an implementation of an asynchronous federated GAN [1] that was used to motivate the choice of libraries and the GAN architecture.

2 Design

2.1 Choosing Paradigm for Federated Learning

The first stage of the design process was to choose the framework to use to implement federated learning. After reviewing the literature there were two distinct paradigms that could be used to achieve this.

2.1.1 Asynchronous Federated Learning

The first paradigm is asynchronous federated learning. In this approach there is only one discriminator and only one generator. The discriminator and generator pair is sent to each node in the network in turn. At each node in the network the GAN is trained for a set number of local iterations. Then the GAN is sent to the next node in the network. This training process repeats for a set number of iterations or until the model accuracy converges. Asynchronous federated learning is shown in Figure 1.

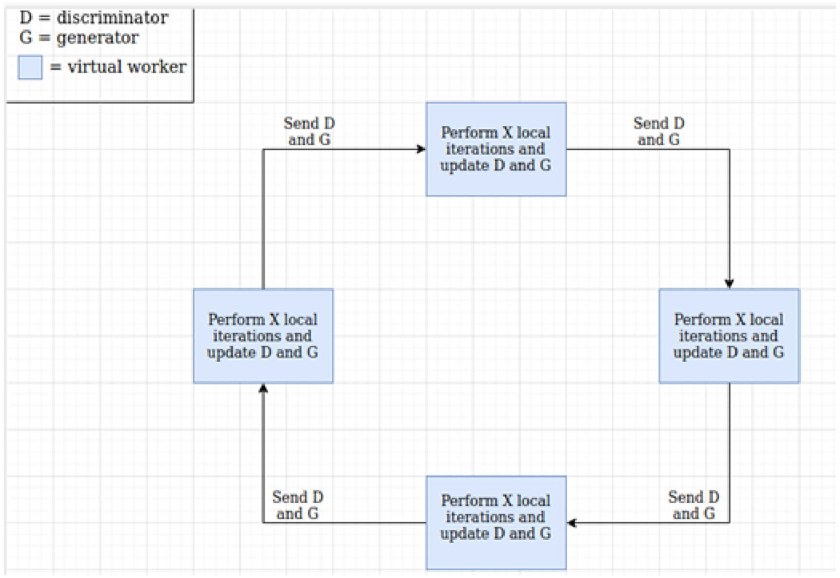

D = discriminator
G = generator
= virtual worker
Send D and G
Perform X local iterations and update D and G
Send D and G
Perform X local iterations and update D and G
Perform X local iterations and update D and G
Send D and G
Perform X local iterations and update D and G
Send D and G

Figure 1: Overview of asynchronous federated learning

2.1.2 Synchronous Federated Learning

The second paradigm is synchronous federated learning. The algorithm for implementing synchronous federated learning for GANs is simple [2]. In this approach there is a network consisting of one server node and an arbitrary number of client nodes. Each client node in the network trains its own GAN using its own unique local data. Therefore, each client node has its own generator and discriminator. After a predetermined number of local iterations X, all client nodes send their model parameters to the server node. Here the model parameters are averaged, and the server's discriminator and generator are assigned these averaged model parameters. Then these averaged model parameters are broadcast to the network and each client node in the network updates the parameters of there local discriminator and generator. Then another round of local training iterations begins. This training process repeats for a set number of training round. Previous research in this area has shown that merging both the discriminator and the generator at each step yields the best results [3]. Synchronous federated learning is shown in Figure 2.

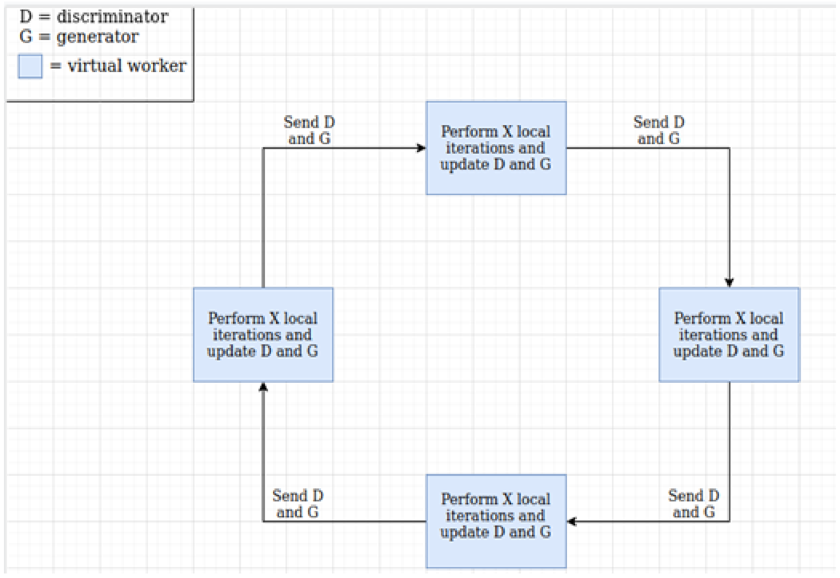

D = discriminator
G = generator
= virtual worker
Send D and G
Perform X local iterations and update D and G
Send D and G
Perform X local iterations and update D and G
Perform X local iterations and update D and G
Send D and G
Perform X local iterations and update D and G
Send D and G

Figure 2: Overview of Synchronous federated learning

2.1.3 Choice of Paradigm

This project used synchronous federated learning to implement the federated GAN. This paradigm was chosen for a few reasons. The first reason was that it was better suited for adding privacy preserving techniques on top of than asynchronous federated learning. Asynchronous federated learning just involves passing the model between different clients in the network. For this project the aim was to focus on protecting from attacks against model parameters being shared and thus synchronous federated learning was chosen.

Another advantage of choosing synchronous federated learning it that it allows client nodes in the network to perform their local iterations in parallel. If asynchronous federated learning was chosen each worker would have to wait to receive the GAN from the previous node in the network before it can begin its local iterations. Therefore, synchronous federated learning is clearly more efficient that asynchronous federated learning.

2.2 Choosing a Library to Implement Federated Learning

PySyft [4] is a Python library designed to allow for the implementation of federated learning. For this project PySyft version 0.2.9 was used.

PySyft was completely rewritten after this version and the newer versions of PySyft are much less suited for implementing synchronous federated learning. This is because the new versions don't allow for the remote execution of code on different clients in the network, they just allow them to act as data stores. Data scientists can connect to these data stores and train models on the data but the models can't be coupled with the data on the nodes as is required for synchronous federated learning.

PySyft 0.2.9 is suited for federated learning due to the virtual worker objects that allow for code to be remotely executed. This allows for a network to be made up of different virtual workers, each with their own data and GANs. These GANs can then be trained locally for each node in the network before the model parameters are then sent back to a central server. Therefore PySyft 0.2.9 allows for the simulation of a network of remote nodes in a single python file. For this project google colabatory was used to run the python code.

PySyft 0.2.9 also has functionality to implement some privacy preserving methods. PySyft allows for the implementation of secure multi-party computation (SMPC). PySyft 0.2.9 also has some support for homomorphic encryption and differential privacy, however, these are not fully implemented which was a big problem later on in the project. This will be expanded upon after the Implementation section.

2.3 Privacy Preserving Techniques

This project aimed to implement three privacy preserving techniques: secure multi-party computation, homomorphic encryption, and differential privacy.

2.3.1 Secure multi-party Computation

Secure multi-party computation (SMPC) is a protocol where data can be distributed across multiple parties without any of the parties seeing the other party's data. This allows for computations to be done on distributed data in a private manner. In the context of a federated GAN this is useful as it allows for the sum of the model parameters to be calculated in such a way that each node in the network only knows its model parameters.

2.3.2 Homomorphic Encryption

Homomorphic encryption is a type of encryption that allows for arithmetic operation to be performed on the encrypted data. This is useful in the context of a federated GAN as it means each client in the node can encrypt its model parameters before sending them to the server. The server can then average the encrypted model parameters without ever having to decrypt them. The encrypted averaged model parameters can then be broadcast across the network. The clients in the network can then finally decrypt the averaged model parameters locally. This means there is never a risk of the model parameters being intercepted in an insecure form.

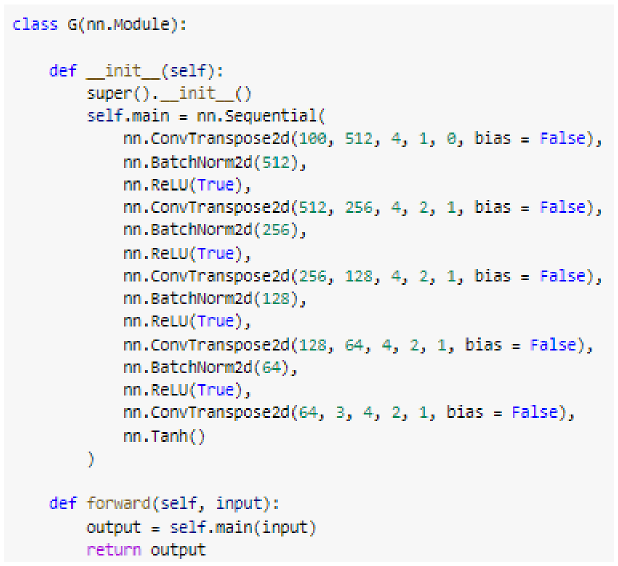
2.3.3 Differential Privacy

Differential privacy is a method to provide data anonymity by adding noise. In the context of a federated GAN differential privacy adds Gaussian noise to the gradient updates to make any attack on the gradients less effective. Differential privacy for model training also requires gradients to be computed for each individual example instead of per batch which means differential privacy adds a lot of time to the time taken to train the model.

3 Implementation

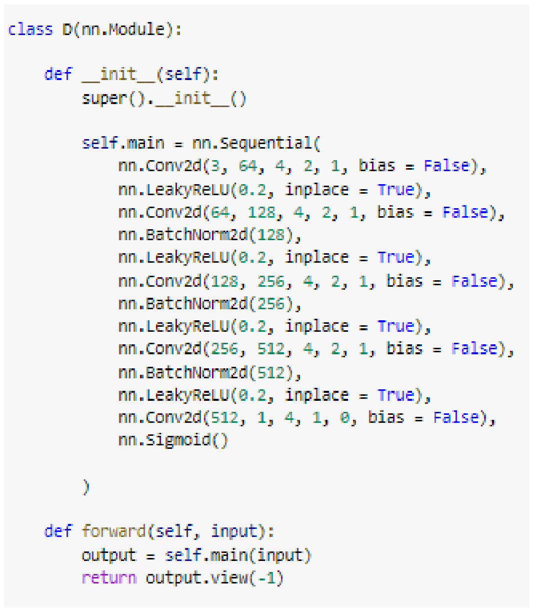
3.1 GAN Architecture

The architecture of the discriminator and the generator is taken from the Implementation of a Federated GAN notebook. This choice was made for two reasons. Firstly the implementation of the GAN itself is not an important focus for this project so there was no benefit to reinventing the wheel or spending a lot of time on the GAN architecture itself. Secondly having the same GAN architecture is beneficial as it gives a comparison point. The images generated by the generator of the federated GAN can be compared to the images generated by the generator in the implementation of a federated GAN notebook. This means it is possible to see if the federated GAN is working as expected. PyTorch is used to implement the models and the code for the generator and discriminator is shown in figure 3 and figure 4 respectively:



```python
class G(nn.Module):

    def __init__(self):
        super().__init__()
        self.main = nn.Sequential(
            nn.ConvTranspose2d(100, 512, 4, 1, 0, bias = False),
            nn.BatchNorm2d(512),
            nn.ReLU(True),
            nn.ConvTranspose2d(512, 256, 4, 2, 1, bias = False),
            nn.BatchNorm2d(256),
            nn.ReLU(True),
            nn.ConvTranspose2d(256, 128, 4, 2, 1, bias = False),
            nn.BatchNorm2d(128),
            nn.ReLU(True),
            nn.ConvTranspose2d(128, 64, 4, 2, 1, bias = False),
            nn.BatchNorm2d(64),
            nn.ReLU(True),
            nn.ConvTranspose2d(64, 3, 4, 2, 1, bias = False),
            nn.Tanh()
        )

    def forward(self, input):
        output = self.main(input)
        return output
```

Figure 3: Architecture of Generator

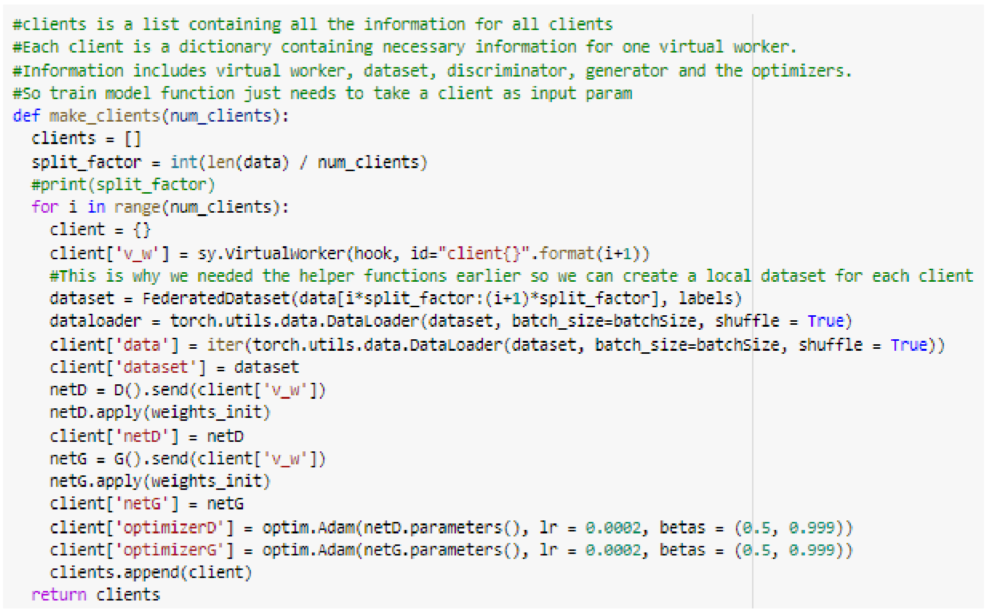


```
class D(nn.Module):

    def __init__(self):
        super().__init__()

        self.main = nn.Sequential(
            nn.Conv2d(3, 64, 4, 2, 1, bias = False),
            nn.LeakyReLU(0.2, inplace = True),
            nn.Conv2d(64, 128, 4, 2, 1, bias = False),
            nn.BatchNorm2d(128),
            nn.LeakyReLU(0.2, inplace = True),
            nn.Conv2d(128, 256, 4, 2, 1, bias = False),
            nn.BatchNorm2d(256),
            nn.LeakyReLU(0.2, inplace = True),
            nn.Conv2d(256, 512, 4, 2, 1, bias = False),
            nn.BatchNorm2d(512),
            nn.LeakyReLU(0.2, inplace = True),
            nn.Conv2d(512, 1, 4, 1, 0, bias = False),
            nn.Sigmoid()

        )

    def forward(self, input):
        output = self.main(input)
        return output.view(-1)
```

Figure 4: Architecture of Discriminator

3.2 Setting up the clients

As discussed in the design section PySyft allows for the creating of a network of nodes where each client node is represented by a virtual network. All other necessary information for each client is stored in a dictionary. Each client had its own local data that is not replicated on any of the other clients. Also, each client node has its own discriminator and generator and the respective optimizers. The method used to create the network takes a parameter which specifies the number of clients to be included in the network. The code to setup the clients for the simulated networks is shown in Figure 5:



```
#clients is a list containing all the information for all clients
#Each client is a dictionary containing necessary information for one virtual worker.
#Information includes virtual worker, dataset, discriminator, generator and the optimizers.
#So train model function just needs to take a client as input param
def make_clients(num_clients):
  clients = []
  split_factor = int(len(data) / num_clients)
  #print(split_factor)
  for i in range(num_clients):
    client = {}
    client['v_w'] = sy.VirtualWorker(hook, id="client{}".format(i+1))
    #This is why we needed the helper functions earlier so we can create a local dataset for each client
    dataset = FederatedDataset(data[i*split_factor:(i+1)*split_factor], labels)
    dataloader = torch.utils.data.DataLoader(dataset, batch_size=batchSize, shuffle = True)
    client['data'] = iter(torch.utils.data.DataLoader(dataset, batch_size=batchSize, shuffle = True))
    client['dataset'] = dataset
    netD = D().send(client['v_w'])
    netD.apply(weights_init)
    client['netD'] = netD
    netG = G().send(client['v_w'])
    netG.apply(weights_init)
    client['netG'] = netG
    client['optimizerD'] = optim.Adam(netD.parameters(), lr = 0.0002, betas = (0.5, 0.999))
    client['optimizerG'] = optim.Adam(netG.parameters(), lr = 0.0002, betas = (0.5, 0.999))
    clients.append(client)
  return clients
```

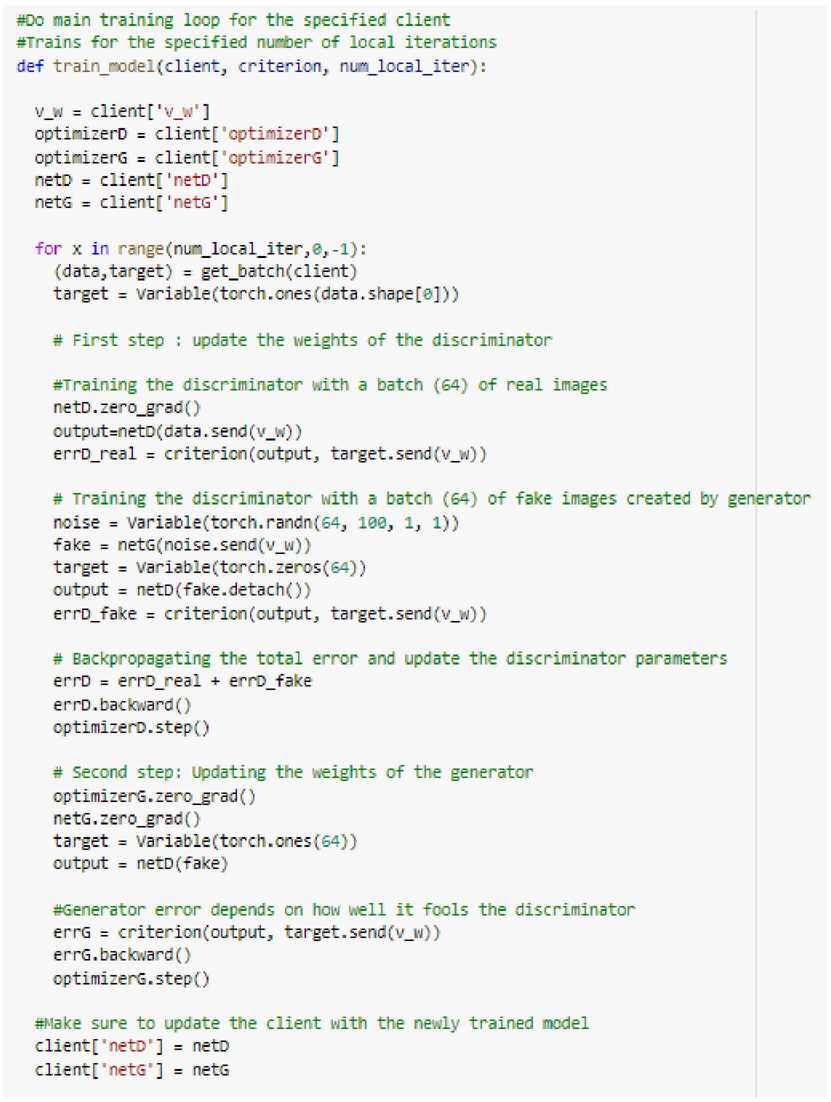
Figure 5: Method used to make the client nodes in network

3.3 Local GAN training

The implementation of the local GAN training is done via two methods. The first method implements the training loop for a GAN. This method takes three parameters as input. The first parameter is the client to perform the local iterations on. The second parameter is used for calculating binary cross entropy. The final parameter is used to specify the number of local iterations to perform.

Each training step for a standard GAN model consists of two steps. The first step is to update the weights of the discriminator. This is done by training the discriminator on a batch of real image and a batch of fake images generated by the generator. Binary cross entropy is used to calculate the loss between the discriminators output and the target output for both cases. The total error is the sum of the error on the real images and the error on the fake images. The total error is then backpropagated to update the weights of the discriminator.

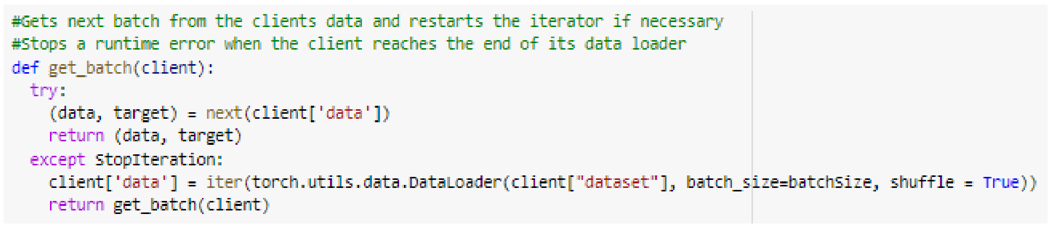
The second step is to update the weights of the generator. This is done by generating a batch of fake images with the generator and then calculating the error of how well the generator fools the discriminator. The error is then backpropagated to update the weights of the generator. The method for training the GANs is shown in Figure 6:



```
#Do main training loop for the specified client
#Trains for the specified number of local iterations
def train_model(client, criterion, num_local_iter):

  v_w = client['v_w']
  optimizerD = client['optimizerD']
  optimizerG = client['optimizerG']
  netD = client['netD']
  netG = client['netG']

  for x in range(num_local_iter,0,-1):
    (data,target) = get_batch(client)
    target = Variable(torch.ones(data.shape[0]))

    # First step : update the weights of the discriminator

    #Training the discriminator with a batch (64) of real images
    netD.zero_grad()
    output=netD(data.send(v_w))
    errD_real = criterion(output, target.send(v_w))

    # Training the discriminator with a batch (64) of fake images created by generator
    noise = Variable(torch.randn(64, 100, 1, 1))
    fake = netG(noise.send(v_w))
    target = Variable(torch.zeros(64))
    output = netD(fake.detach())
    errD_fake = criterion(output, target.send(v_w))

    # Backpropagating the total error and update the discriminator parameters
    errD = errD_real + errD_fake
    errD.backward()
    optimizerD.step()

    # Second step: Updating the weights of the generator
    optimizerG.zero_grad()
    netG.zero_grad()
    target = Variable(torch.ones(64))
    output = netD(fake)

    #Generator error depends on how well it fools the discriminator
    errG = criterion(output, target.send(v_w))
    errG.backward()
    optimizerG.step()

  #Make sure to update the client with the newly trained model
  client['netD'] = netD
  client['netG'] = netG
```

Figure 6: Method used to train the local GANs

There is also a helper method that restarts the iterator over the client's dataset if needed to stop an error being thrown. This method is shown in Figure 7:



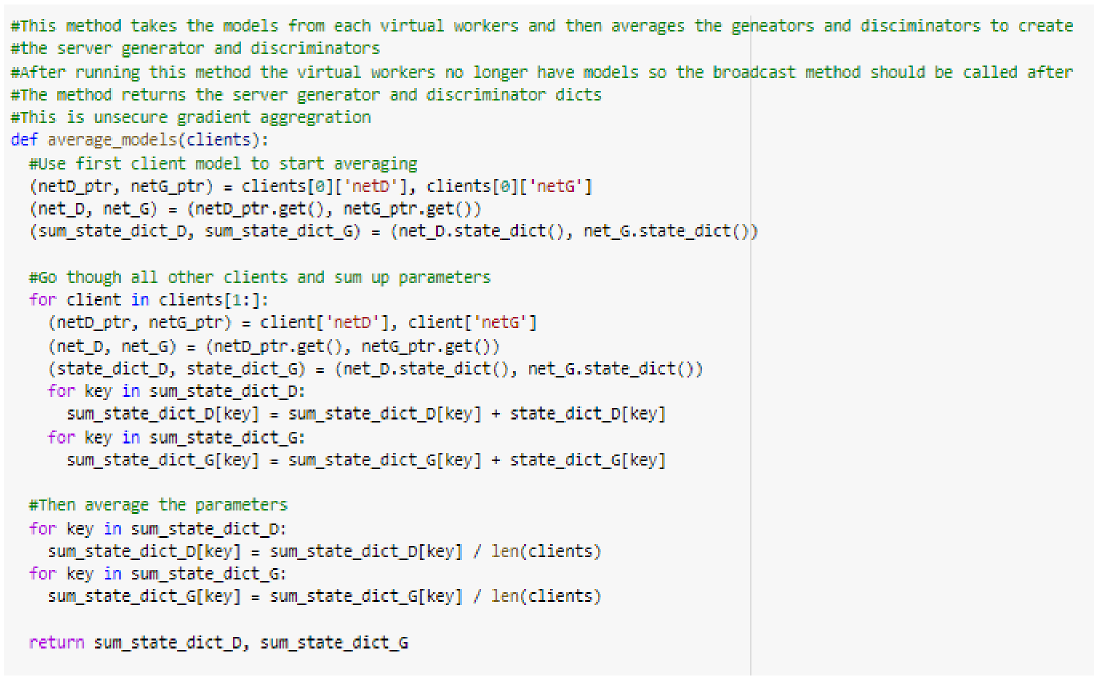
```
#Gets next batch from the clients data and restarts the iterator if necessary
#Stops a runtime error when the client reaches the end of its data loader
def get_batch(client):
  try:
    (data, target) = next(client['data'])
    return (data, target)
  except StopIteration:
    client['data'] = iter(torch.utils.data.DataLoader(client["dataset"], batch_size=batchSize, shuffle = True))
    return get_batch(client)
```

Figure 7: Method used to get next batch

3.4 Averaging and Broadcasting Model Parameters

The most important step that allows for the GAN to be trained via federated learning is to implement the code to average the model parameters and then broadcast these new model parameters to all clients in the network.

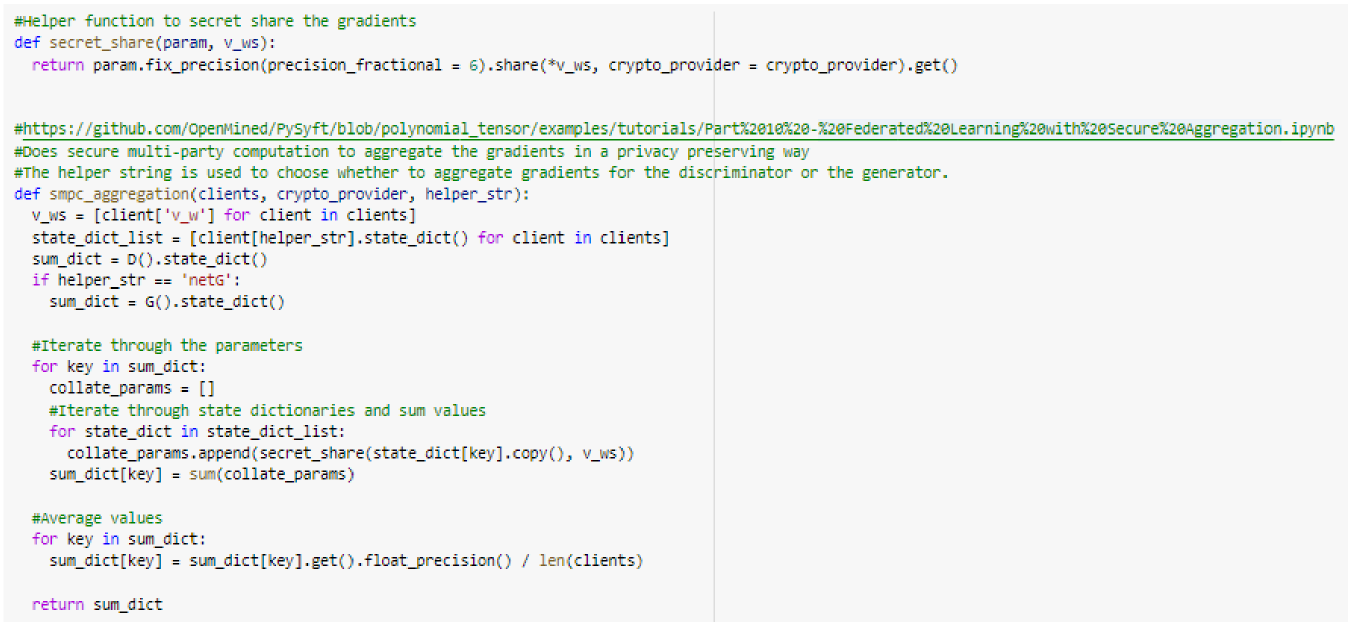
The first method in this section implements the averaging of the models. This is done by summing the state dictionaries of the discriminators and generators from all of the clients. Each layer in the resulting state dictionaries is then divided by the number of clients in the network to create the average state dictionary. This method is shown in Figure 8:



```
#This method takes the models from each virtual workers and then averages the geneators and disciminators to create
#the server generator and discriminators
#After running this method the virtual workers no longer have models so the broadcast method should be called after
#The method returns the server generator and discriminator dicts
#This is unsecure gradient aggregration
def average_models(clients):
  #Use first client model to start averaging
  (netD_ptr, netG_ptr) = clients[0]['netD'], clients[0]['netG']
  (net_D, net_G) = (netD_ptr.get(), netG_ptr.get())
  (sum_state_dict_D, sum_state_dict_G) = (net_D.state_dict(), net_G.state_dict())

  #Go though all other clients and sum up parameters
  for client in clients[1:]:
    (netD_ptr, netG_ptr) = client['netD'], client['netG']
    (net_D, net_G) = (netD_ptr.get(), netG_ptr.get())
    (state_dict_D, state_dict_G) = (net_D.state_dict(), net_G.state_dict())
    for key in sum_state_dict_D:
      sum_state_dict_D[key] = sum_state_dict_D[key] + state_dict_D[key]
    for key in sum_state_dict_G:
      sum_state_dict_G[key] = sum_state_dict_G[key] + state_dict_G[key]

  #Then average the parameters
  for key in sum_state_dict_D:
    sum_state_dict_D[key] = sum_state_dict_D[key] / len(clients)
  for key in sum_state_dict_G:
    sum_state_dict_G[key] = sum_state_dict_G[key] / len(clients)

  return sum_state_dict_D, sum_state_dict_G
```

Figure 8: Method used to average the model parameters unsafely

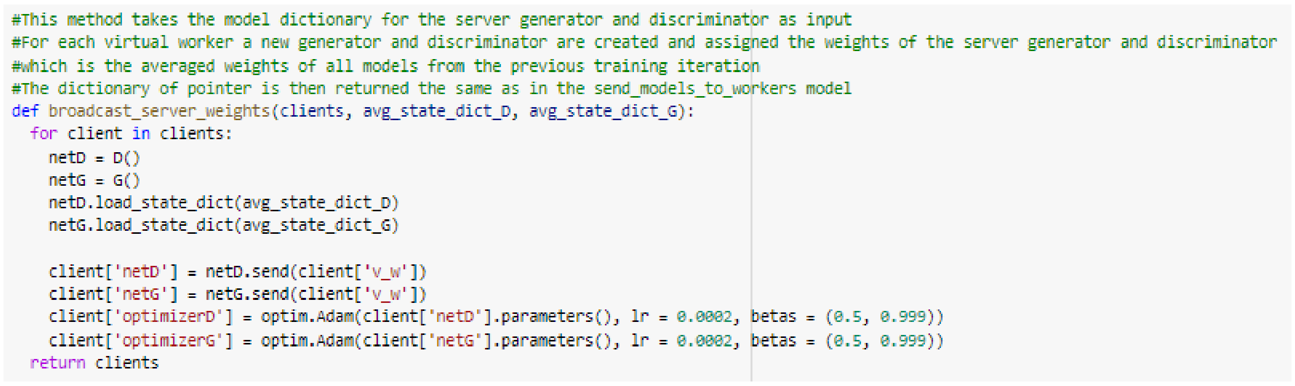
The above method does not secure the model parameters in any way and is thus unsafe. Another method was implemented which uses SMPC to sum the model parameters such that each client is the only node to ever see its model parameters. The method takes a string as input to specify whether to average the state dictionary for the discriminator or generator. This is done to avoid having to write very similar code twice. This method uses in-built functionality from the PySyft 0.2.9 library. The share() function from PySyft implements additive secret sharing. The tensor is split into shares which are encrypted and then shared between all participants. The code for this method is shown in Figure 9:



```
#Helper function to secret share the gradients
def secret_share(param, v_ws):
  return param.fix_precision(precision_fractional = 6).share(*v_ws, crypto_provider = crypto_provider).get()


#https://github.com/OpenMined/PySyft/blob/polynomial_tensor/examples/tutorials/Part%2010%20-%20Federated%20Learning%20with%20Secure%20Aggregation.ipynb
#Does secure multi-party computation to aggregate the gradients in a privacy preserving way
#The helper string is used to choose whether to aggregate gradients for the discriminator or the generator.
def smpc_aggregation(clients, crypto_provider, helper_str):
  v_ws = [client['v_w'] for client in clients]
  state_dict_list = [client[helper_str].state_dict() for client in clients]
  sum_dict = D().state_dict()
  if helper_str == 'netG':
    sum_dict = G().state_dict()

  #Iterate through the parameters
  for key in sum_dict:
    collate_params = []
    #Iterate through state dictionaries and sum values
    for state_dict in state_dict_list:
      collate_params.append(secret_share(state_dict[key].copy(), v_ws))
    sum_dict[key] = sum(collate_params)

  #Average values
  for key in sum_dict:
    sum_dict[key] = sum_dict[key].get().float_precision() / len(clients)

  return sum_dict
```

Figure 9: Method used to average the model parameters via SMPC

Next the averaged model parameters need to be broadcast to all of the clients in the network. This is done by the method shown in Figure 10:

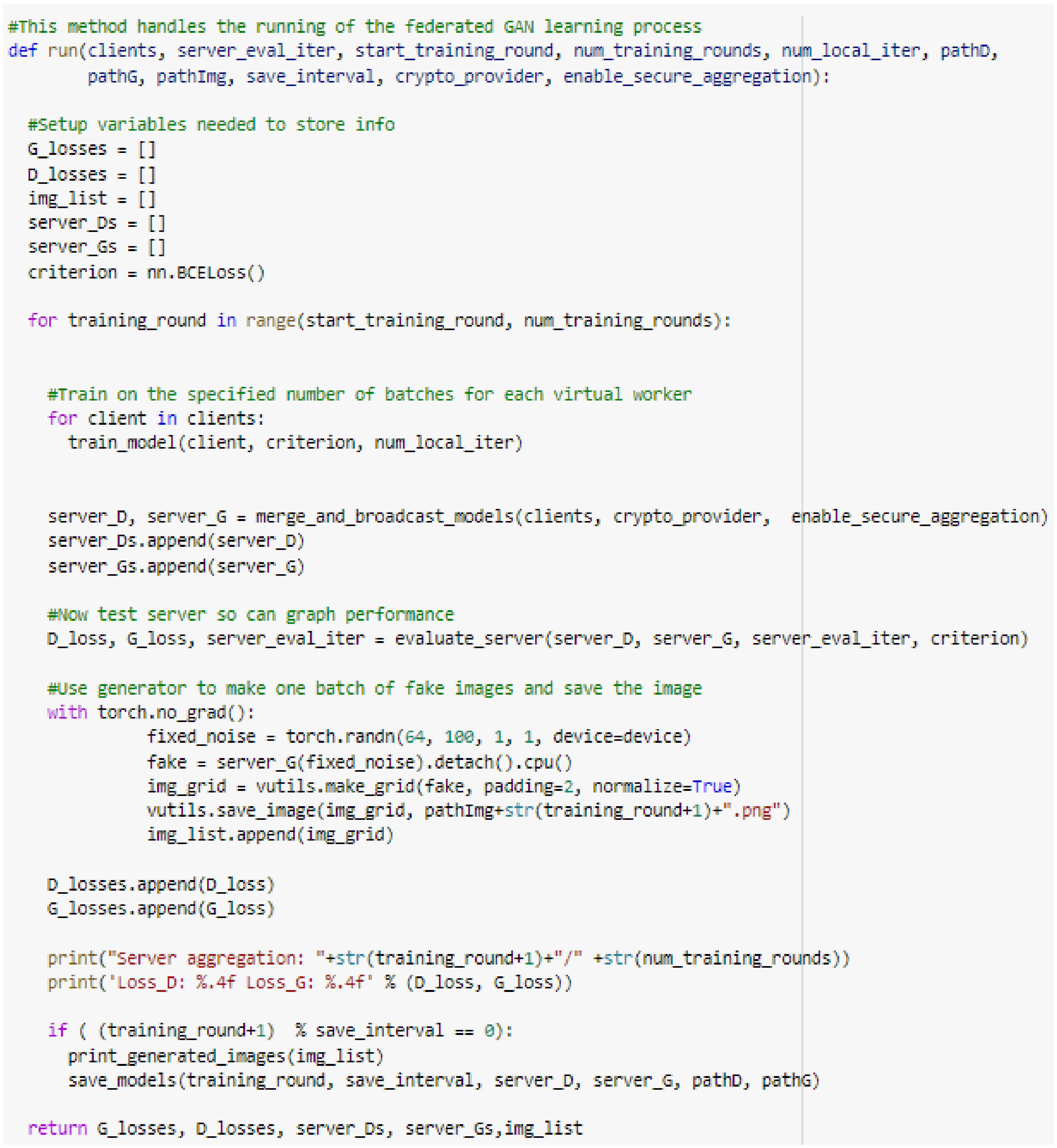


```
#This method takes the model dictionary for the server generator and discriminator as input
#For each virtual worker a new generator and discriminator are created and assigned the weights of the server generator and discriminator
#which is the averaged weights of all models from the previous training iteration
#The dictionary of pointer is then returned the same as in the send_models_to_workers model
def broadcast_server_weights(clients, avg_state_dict_D, avg_state_dict_G):
  for client in clients:
    netD = D()
    netG = G()
    netD.load_state_dict(avg_state_dict_D)
    netG.load_state_dict(avg_state_dict_G)

    client['netD'] = netD.send(client['v_w'])
    client['netG'] = netG.send(client['v_w'])
    client['optimizerD'] = optim.Adam(client['netD'].parameters(), lr = 0.0002, betas = (0.5, 0.999))
    client['optimizerG'] = optim.Adam(client['netG'].parameters(), lr = 0.0002, betas = (0.5, 0.999))
  return clients
```

Figure 10: Method used to broadcast the new model parameters to all clients in the network

3.5 Running Federated Learning

This method handles the coordination of the training rounds for the federated GAN. Unfortunately, the current implementation does not allow for the clients to perform their local GAN training in parallel. Therefore each client performs there training iterations sequentially. After all clients have completed their local iterations, the models are averaged, and the averaged model parameters are then broadcast back to the clients. The method takes parameters to specify the number of training rounds, the number of local iterations to be performed during each training round, the file paths to save the models, and a parameter to specify how often to save the models. The code for this method is shown in Figure 11:

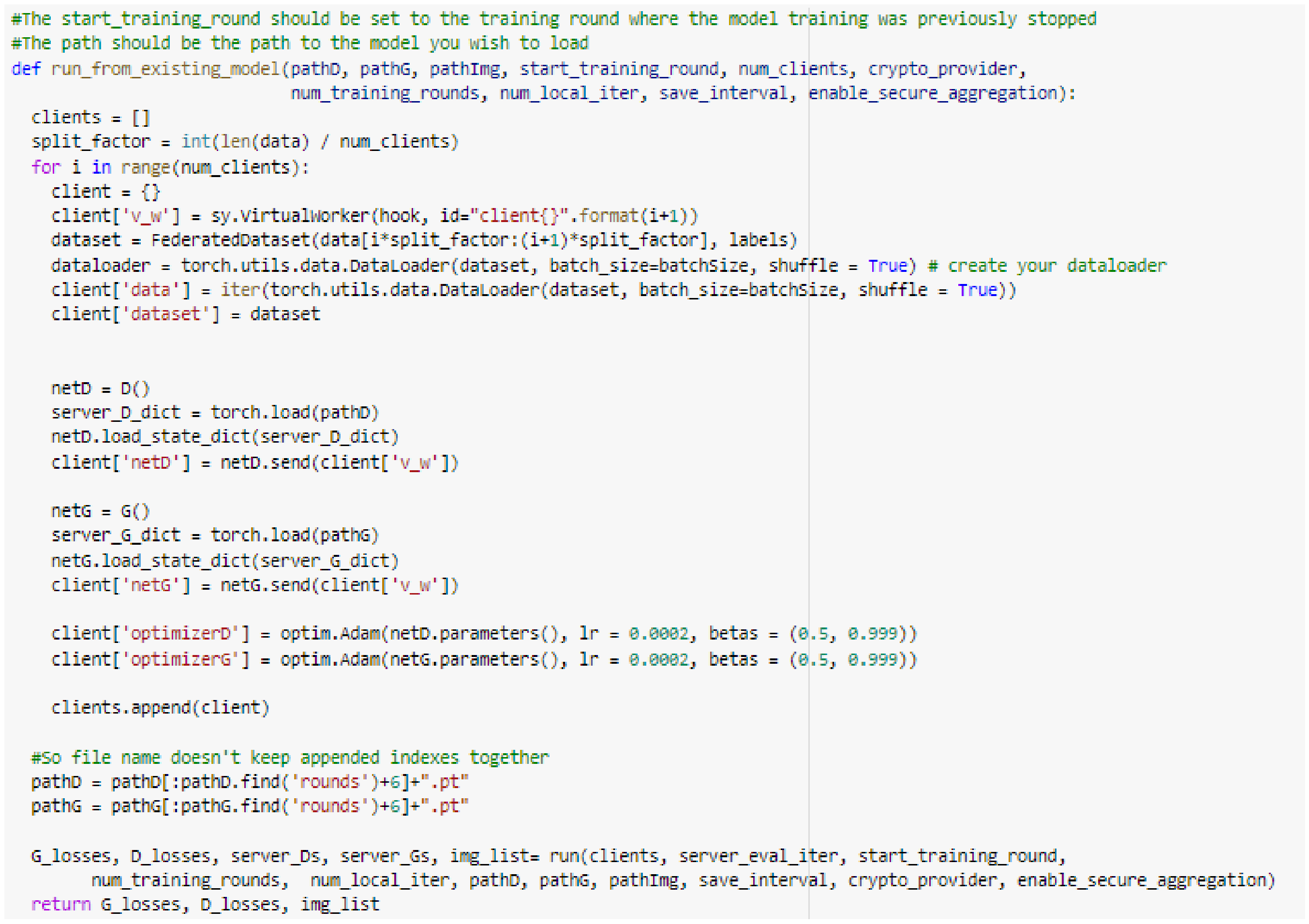


```
#This method handles the running of the federated GAN learning process
def run(clients, server_eval_iter, start_training_round, num_training_rounds, num_local_iter, pathD,
        pathG, pathImg, save_interval, crypto_provider, enable_secure_aggregation):

  #Setup variables needed to store info
  G_losses = []
  D_losses = []
  img_list = []
  server_Ds = []
  server_Gs = []
  criterion = nn.BCELoss()

  for training_round in range(start_training_round, num_training_rounds):


    #Train on the specified number of batches for each virtual worker
    for client in clients:
      train_model(client, criterion, num_local_iter)


    server_D, server_G = merge_and_broadcast_models(clients, crypto_provider,  enable_secure_aggregation)
    server_Ds.append(server_D)
    server_Gs.append(server_G)

    #Now test server so can graph performance
    D_loss, G_loss, server_eval_iter = evaluate_server(server_D, server_G, server_eval_iter, criterion)

    #Use generator to make one batch of fake images and save the image
    with torch.no_grad():
              fixed_noise = torch.randn(64, 100, 1, 1, device=device)
              fake = server_G(fixed_noise).detach().cpu()
              img_grid = vutils.make_grid(fake, padding=2, normalize=True)
              vutils.save_image(img_grid, pathImg+str(training_round+1)+".png")
              img_list.append(img_grid)

    D_losses.append(D_loss)
    G_losses.append(G_loss)

    print("Server aggregation: "+str(training_round+1)+"/" +str(num_training_rounds))
    print('Loss_D: %.4f Loss_G: %.4f' % (D_loss, G_loss))

    if ( (training_round+1)  % save_interval == 0):
      print_generated_images(img_list)
      save_models(training_round, save_interval, server_D, server_G, pathD, pathG)

  return G_losses, D_losses, server_Ds, server_Gs,img_list
```

Figure 11: Method used to Coordinate Federated Learning

3.6 Running Federated Learning for a Saved Model

Due to the amount of time training the federated GAN takes it was necessary to create an additional method to allow training to resume from a saved method. Restarting the training process is fairly simple the only thing that needs to be done is to initiate all the client nodes with the necessary information. This method is shown in Figure 12.



```
#The start_training_round should be set to the training round where the model training was previously stopped
#The path should be the path to the model you wish to load
def run_from_existing_model(pathD, pathG, pathImg, start_training_round, num_clients, crypto_provider,
                            num_training_rounds, num_local_iter, save_interval, enable_secure_aggregation):
  clients = []
  split_factor = int(len(data) / num_clients)
  for i in range(num_clients):
    client = {}
    client['v_w'] = sy.VirtualWorker(hook, id="client{}".format(i+1))
    dataset = FederatedDataset(data[i*split_factor:(i+1)*split_factor], labels)
    dataloader = torch.utils.data.DataLoader(dataset, batch_size=batchSize, shuffle = True) # create your dataloader
    client['data'] = iter(torch.utils.data.DataLoader(dataset, batch_size=batchSize, shuffle = True))
    client['dataset'] = dataset


    netD = D()
    server_D_dict = torch.load(pathD)
    netD.load_state_dict(server_D_dict)
    client['netD'] = netD.send(client['v_w'])

    netG = G()
    server_G_dict = torch.load(pathG)
    netG.load_state_dict(server_G_dict)
    client['netG'] = netG.send(client['v_w'])

    client['optimizerD'] = optim.Adam(netD.parameters(), lr = 0.0002, betas = (0.5, 0.999))
    client['optimizerG'] = optim.Adam(netG.parameters(), lr = 0.0002, betas = (0.5, 0.999))

    clients.append(client)

  #So file name doesn't keep appended indexes together
  pathD = pathD[:pathD.find('rounds')+6]+".pt"
  pathG = pathG[:pathG.find('rounds')+6]+".pt"

  G_losses, D_losses, server_Ds, server_Gs, img_list= run(clients, server_eval_iter, start_training_round,
        num_training_rounds,  num_local_iter, pathD, pathG, pathImg, save_interval, crypto_provider, enable_secure_aggregation)
  return G_losses, D_losses, img_list
```

Figure 12: Method used to resume federated learning from a saved model

4 Issues with Implementation

4.1 Differential Privacy

This project did not manage to successfully implement differential privacy. PySyft 0.2.9 has some support for differential privacy. The privacy engine from the Opacus [5] library provides support for training GANs with differential privacy, however, there were a few issues that meant this could not be successfully implemented. The main issue was that PySyft 0.2.9 did not provide the functionality to allow the privacy engine to be used on models being trained remotely on virtual workers. This meant that differential privacy and synchronous federated learning can't be used at the same time in PySyft 0.2.9.

One potential solution tried was to work around this problem by sending the model back to the server and doing the training on the server and then sending the model back to the client node. However, there will still issues with this. The first problem was that using the privacy engine greatly increased the amount of RAM used by the colabatory network. This would cause the notebook to crash after the limit of available RAM was used.

Another problem with this attempted solution was that the privacy engine only supported some PyTorch layers, and the GAN architecture used in the project contained batch norm layers which the privacy engine did not support. To successfully implement differential privacy the architecture of the GANs would have to be changed.

4.2 Homomorphic Encryption

PySyft 0.2.9 has some support for homomorphic encryption. Using the Pailier encryption scheme PySyft 0.2.9 allows for tensors to be encrypted. Unfortunately, it is not possible to transfer encrypted tensors between the virtual workers. This means the gradients can be encrypted but they can't be sent back to the main server or broadcast to the client nodes. This means PySyft 0.2.9 does not allow for the implementation of synchronous federated learning with homomorphic encryption.

5 Results

The final implementation of the project did not allow for much experimentation due to the failure to implement some of the privacy preserving techniques. The main experiment that could be performed was to evaluate how much longer the federated GAN takes to train when SMPC is used for the gradient aggregation instead of unsecure gradient aggregation. The experiment was done with the exact same parameters for both federated GANs. The network consisted of five client nodes and one server node. Each node did ten iterations of local training for each training round. Overall, 130 training rounds were completed for each federated GAN. On average the training round that used SMPC for the gradient aggregation took sixteen seconds longer. The plot below shows the time difference.

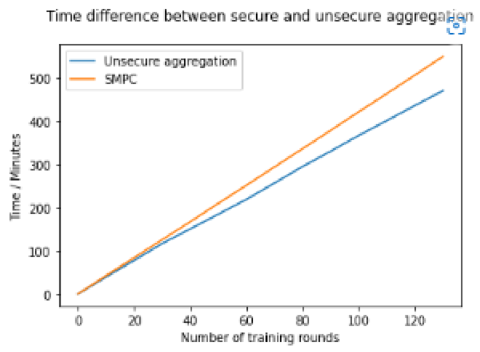
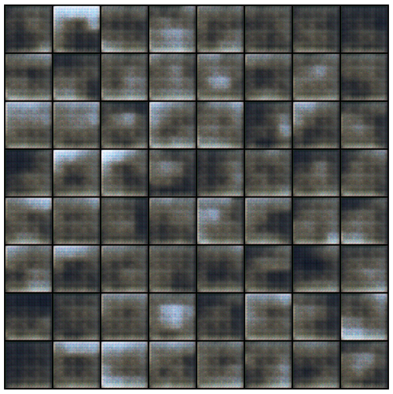
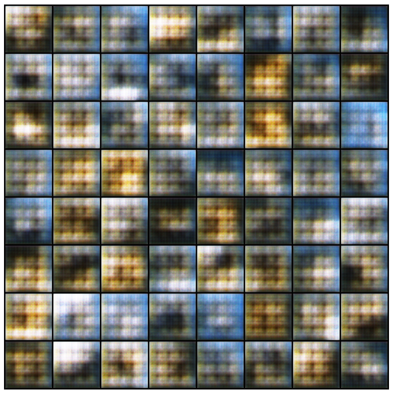
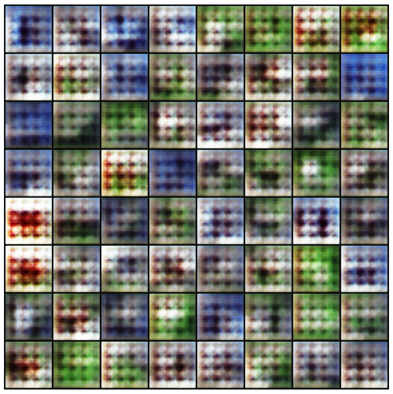


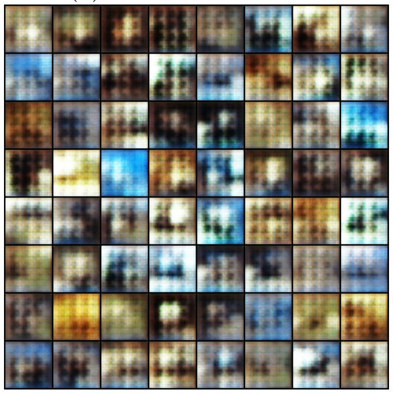
Figure 13: Results of experiment

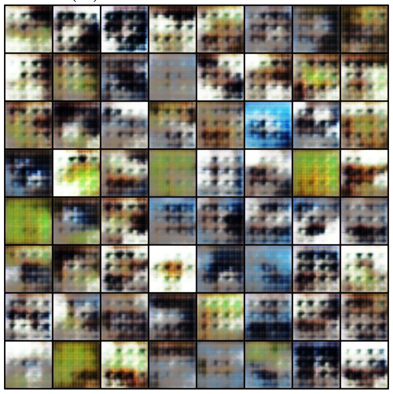
The results of the generator after a set number of training rounds in intervals of ten is shown in Figure 14. The federated GAN takes around 120 training rounds to converge. The images generated by the generator are the same as those of the central GAN from the implementation of a federated GAN notebook. As both this project and the implementation of a federated GAN notebook use the same model architecture the fact the generators converge to similar images shows the federated learning training process has been implemented successful. Therefore, we can conclude that the federated GAN works successfully with SMPC.

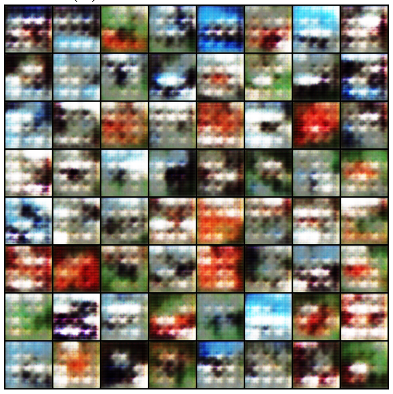

(a) After 10 iterations

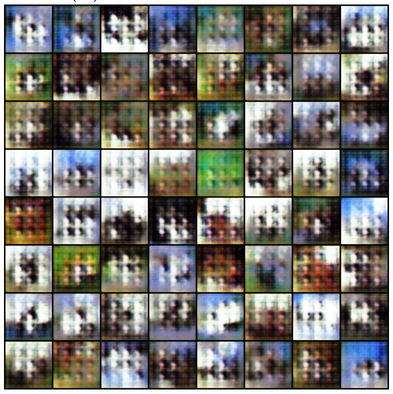

(b) After 20 iterations

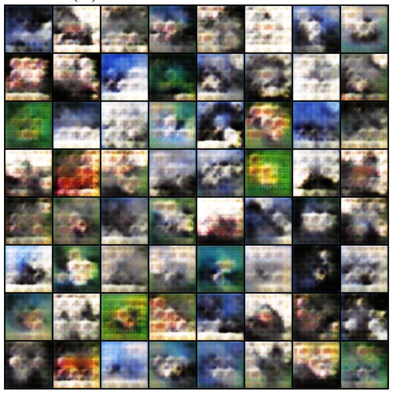

(c) After 30 iterations

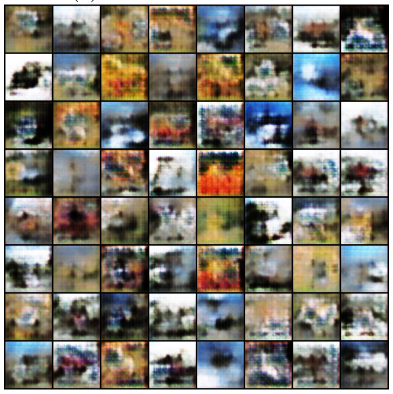

(d) After 40 iterations

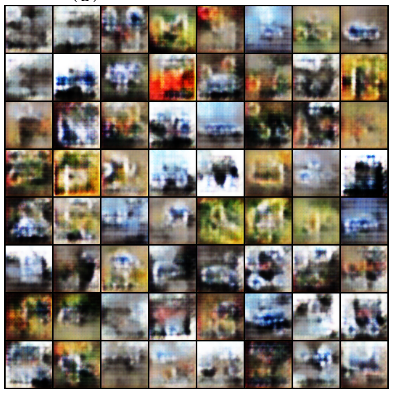

(e) After 50 iterations

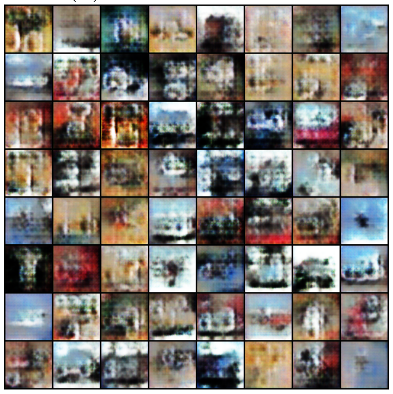

(f) After 60 iterations


(g) After 70 iterations


(h) After 80 iterations


(i) After 90 iterations


(j) After 100 iterations


(k) After 110 iterations

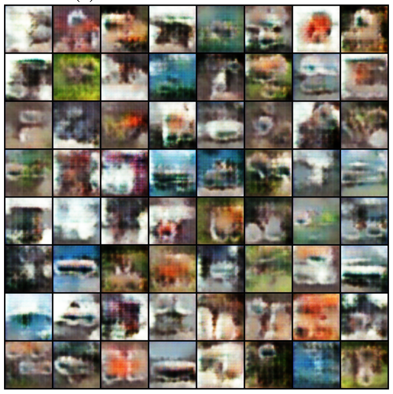

(l) After 120 iterations

Figure 14: Training of a Federated GAN with SMPC

6 Evaluation

The main issues with this project were caused by mistakes made in the initial design phase of the project. The choice of PySyft 0.2.9 to implement the federated GAN was short-sighted as although it worked for implementing the federated GAN, it meant that the privacy preserving techniques could not be implemented on top of the federated GAN. This was one of the aims of the project and thus from this perspective the project was not as successful as it could have been.

To improve on this project is not easy as in the project's current state it is not possible to add more privacy preserving techniques. Another problem is that due to limitations of PySyft 0.2.9 virtual workers it is not possible to remotely execute code in parallel. This means that the client nodes in the simulated network all have to execute their code sequentially. This greatly increases the time required to train the model.

These problems are difficult to fix as they would require a complete rewrite of the project's framework. All the limitations are down to PySyft and thus a new library would have to be chosen to allow for the writing of a federated GAN where the privacy preserving techniques and parallelisation could be added. As mentioned earlier new version of PySyft are not appropriate for the rewrite of this project. This is because the new versions don't allow for the remote execution of code on different clients and thus synchronous federated learning can't be implemented.

References

[1] ghonimaraghda. *Implementation-of-federated-GANs*. URL: `https://github.com/ghonimaraghda/Implementation-of-federated-GANs`.

[2] Mohammad Rasouli, Tao Sun, and Ram Rajagopal. "FedGAN: Federated Generative Adversarial Networks for Distributed Data". In: *CoRR* abs/2006.07228 (2020). arXiv: `2006.07228`. URL: `https://arxiv.org/abs/2006.07228`.

[3] Chenyou Fan and Ping Liu. "Federated Generative Adversarial Learning". In: *CoRR* abs/2005.03793 (2020). arXiv: `2005.03793`. URL: `https://arxiv.org/abs/2005.03793`.

[4] openminded. *PySyft*. URL: `https://github.com/OpenMined/PySyft`.

[5] *Opacus*. URL: `https://opacus.ai/`.