Detect and Defend - Addressing MPAF: Model Poisoning Attacks to Federated Learning based on Fake Clients

Summary

Federated learning is a distributed ML paradigm in which multiple clients jointly learn a model via a centralized server. Clients fit local models using local training data and send model updates to a server. The server aggregates client updates to a global update to learn a global model. Because a federated model relies on accuracy of the clients updates to learn, it is vulnerable to model poisoning attacks. Compromised clients can send malicious updates to decrease the accuracy of the global model. Previous attack research has explored the usage of compromised genuine clients to maliciously affect federated models. This research has concluded that while federated models are vulnerable to model poisoning, compromising enough genuine clients to cause a malicious effect is impractical and production FL models operate at too large a scale to be vulnerable. However, an attack, instead of compromising genuine clients, could instead focus on injecting "fake clients" on a sufficient scale to decrease the accuracy of the global model. The paper proposes using an attack in which an inaccurate model is used by fake clients to create updates that consitently move the global model towards inaccracy. Other approaches such as random updates and historical based negation updates have demonstrated inconsistency in their ability to affect the accuracy of a federated model. The goal is to minimize the difference between the performance of the final global model and the innacurate model. The paper concludes that even in the presence of common defensive update aggregation methods and update normalization, model poisoning with fake clients is an effective attack against FL models.

Insights/Imlementation

Classifier

> #### classifier.ipynb
>
> This is our notebook for training and testing the base model for our federated system. It uses the MNIST dataset to train and test a image classification model. The trained model is saved to classifier.pth. The current saved model has around a 90% accuracy

> #### classifier.py
>
> This file contains shared configuration and utility for out server and clients to interact with the model

>> ##### class Net()
>>
>> This is the classification model, it consists of (1, 1, 5) convolution layer, a (2, 2) pooling layer, another (1, 1, 5) convolution layer, and 3 other linear layers. Layers use the relu activatiion function

>> ##### def train_model(model, data_loader, epoch=5):
>>
>> Trains our model using SGD with a learning rate of 0.001 and cross_entropy loss function

>> ##### def test_model(model):
>>
>> Tests our model using mnist test set, returns model accuracy

>> ##### def random_dataloader(samples):
>>
>> Randomly samples {samples} samples from the mnist training dataset and returns as dataloader

Server

> #### server.py
>
>> ##### class FederatedLearning
>>
>>> ##### def init(self, model, threshold)
>>>
>>> initialize our federated learning model with global model {model}. {threshold} represents the number of client updates the server should collect before aggregating, stored in a list of client updates.
>>>
>>> ##### def aggregate(self)
>>>
>>> aggregates client updates, and updates global models. Takes mean of client updates and adjusts model
>>>
>>> ##### def update(self, client_update)
>>>
>>> called when a client update is sent to a server. Adds update to client update list. When threshold is exceeded calls aggregate and empties client update list. After every aggregation the model is tested for performance. Performance mterics are stored in a list: historical_accuracy

> #### Usage
>
> When running an optional threshold can be used (defaults to 100). ex. python server.py 10

> ### Endpoints
>
> > #### /model/
> >
> > > ##### GET
> > >
> > > return the federated models state as json
> > >
> > > ##### POST
> > >
> > > takes a client update as request body json and registers update with federated model
> >
> > #### /model/accuracy
> >
> > returns historical_accuracy list as json

Client

> ### class Client
>
> > #### def init(self, address, data_loader)
> >
> > loads global model from server
> >
> > #### def update_model(self)
> >
> > trains local model with local data and sends client update to server
>
> ### def load_model(address)
>
> utility function to load global model from address

> ### Usage
>
> takes an address and an optional argument representing the size for the local training data
>
> - python client.py http://localhost:5000
> - python client.py http://localhost:5000 100 (defaults to 4)

MPAF

> Doing bad things. Like client, but poisonus
>
> ### class MPAF
>
> > #### def init(self, address, scaling)
> >
> > initializes a target model with random weights.
> >
> > #### def send_bad_update(self):
> >
> > compares the state of the global model with the target model and sends a scaled update representing a move towad the target model

Clients Sim

> I don't have 1000 laptops so we need a client simulator to send requests to the server

> > ### class ClientSim:
> >
> > #### def init(self, address, percent_evil, num_clients, scaling, train_sample_size)
> >
> > initializes a (1 - percent_evil) * num_clients good clients each with train_sample_size training samples. Also initializes our fake client. Uses Client class from client.py and MPAF class from mpaf.py
> >
> > #### def send_updates(self):
> >
> > Main workhorse of ClientSim. Sends client updates from good_clients and sends malicious requests for every (1 - percent_evil) / percent_evil good clients
> >
> > #### def make_good_clients(self, address, num, training_size)

> Utility function to populate good clients
>
> ### def report(self)
>
> Calls server /model/accuracy endpoint to get a list of the historical model accuracy of the federated model

Usage

Simulates clients and sends updates to server prints historical accuracy of federated model before exiting python clients_sim.py http://localhost:5000 optional args (not robust must be in order, cannot use a later one without the prior optional arg lol):

> 1. percent_evil (defaults .2)
> 2. num_clients (defaults 100)
> 3. evil_scale (defaults 1)
> 4. train_sample_size = (defaults 4)