DeepGossip: Peer-to-peer Collaborative Deep Learning

CPSC 416 – Distributed Systems

Introduction

Standard machine learning systems tend to be centralized and operate under a model that gives the central server direct access to testing data. Recently, distributed approaches such as *collaborative deep learning* have emerged that allow for devices to collaboratively learn while keeping training data local to the device. For our project 2 we will build DeepGossip, a collaborative learning system that emphasizes privacy and trust in an adversarial context. DeepGossip uses a peer-to-peer architecture in which each node maintains a deep learning model and collaborates with connected peers to improve the accuracy of the model. The definition of accuracy will generally be data-dependent, but in deep learning systems this is usually defined as an acceptably minimized loss function on some dataset. The goal of the system is to collaboratively learn an appropriate model of the dataset while providing strong privacy guarantees and defenses against poisoning attacks.

Background

Collaborative learning systems use a shared learning model that allows nodes to keep training data local, communicating only “masked features” [4] which represent gradients produced from the deep learning process (described in detail under System Architecture). Each node joins the network and is assumed to have access to some subset of the global dataset and performs local, independent training for a fixed period. After this, the collaborative learning process begins and the node can receive and publish updates from and to connected peers.

There are significant privacy issues that become relevant in a collaborative learning setting. It is often important to participants in such a setting to keep local data completely private from other participants. For example, this is crucial in a healthcare setting where data privacy can be a strict legal requirement.

Our system keeps data private to each node, with no node ever seeing the raw data used to train any other node’s local model. Potential benefits from this approach include smarter models, lower latency and less communication overhead. However, collaborative learning also brings distinct challenges.

The cost of keeping training data local to nodes is more complicated in such an environment. Nodes cannot inspect the raw data used to produce an update, and are thus restricted to analysis on received masked features. This is particularly relevant in an adversarial context with the threat of data poisoning from malicious nodes. Determinations of maliciousness must be made without access to the raw training data. DeepGossip will attempt to implement solutions to these problems.

Approach/System Architecture

Machine Learning

A combination of vast amounts of available data and advances in processing power has led to the explosion of **deep learning** in the last decade as one of the most promising machine learning techniques, with applicability to a wide range of tasks. Deep learning describes a deep neural network, which is a neural network that consists of a number of so-called *hidden* layers. Neural networks are generally organized into layers consisting of an input, a matrix of weights, and an activation function. The activation function is nonlinear, allowing neural networks to act as universal function approximators. The most common algorithm used to optimize neural networks is called stochastic gradient descent, where the stochasticity comes from using randomly shuffled data to train the network. Our system will use a variation of SGD called Distributed Selective SGD [5] to attend to the privacy concerns, and is described in the privacy section of this proposal.

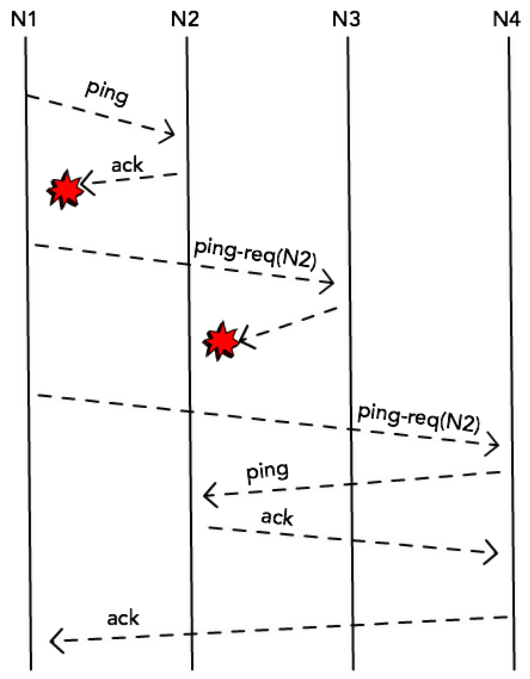
Nodes and Network Topology

Our system is comprised of a basic server for bootstrapping the network and learning nodes that provide an API for applications to leverage the results of the training.

We assume learning nodes and applications are on the same host. When a learning node comes online, an accompanying application connects through an application-learning node API. Without a connected application, a learning node is idle. To bootstrap the Peer-to-Peer (P2P) network, the server will accept registration events from learning nodes connected to applications as they come online and record their information. Nodes connected to the server can query the server for a set of random peers, provided the node is below the minimum threshold of connectivity (i.e. when a node comes online, or has detected peer failures).

Learning nodes form an unstructured P2P network topology and communicate via a gossiping protocol. Gossiping or epidemic protocols follow a model where initially a single node with new data is considered infective, and nodes without the update are susceptive. Infected individuals then infect susceptive nodes, spreading the update as fast as possible. Gossiping protocols have several desirable qualities for our system: they are relatively simple to implement, robust to churning, and have quick (logarithmic) convergence. Furthermore, we are afforded the option of gossiping since our application does not require strong consistency and the process of learning neural network models does not require synchronicity or strong concurrency guarantees. Specifically, we will use the SWIM protocol for failure detection and a rumour mongering approach to update dissemination.

SWIM protocol



- Every T seconds, node A select a random peer B from A's view of the system
- A PINGs B
- If A does not receive an ACK from B within a pre-defined timeout window, A sends a PING-REQ to k random peers, who try to PING B and relay the ACK to A
- If A does not get a response (direct or indirect) within the protocol period, it marks B as failed
- Piggybacking:
    - Every time a node sends a PING, PING-REQ, or ACK, it also includes a list of failed peers which are marked as failed by the receiving node

  Figure from https://www.brianstorti.com/swim/

Rumour Mongering

- All nodes are initially "susceptive" (ie they have not received the new update)
- When a node obtains new data, the update becomes a "hot rumour"
- A node that has a hot rumor periodically chooses a random node to spread to
- Eventually nodes lose interest in spreading the rumor and it dies since either it has been spread too many times (each node maintains a counter), or the node has already seen the rumor
- The order or time each node joins the system does not matter. I.e., if a node joins late, it will still go through the above processes

Failures

In the event of disconnection or node failure, the connected learning nodes proceed as normal. Eventually, the rest of the network will become aware of a node's disappearance through the SWIM protocol described above. In the case that a learning node is disconnected from the network but otherwise operational (i.e. transitory disconnection), it will continue training on its local data, and if it re-establishes connection to the network, it will then disseminate its new updates. We describe this as a *disconnected learning mode*.

Privacy

In the paper *Privacy-Preserving Deep Learning*, the authors present an alteration to the classic stochastic gradient descent algorithm called **Distributed Selective Stochastic Gradient Descent**. This algorithm is designed to provide strong privacy guarantees using two key observations:

1. Different datasets / data points contribute to different parameters (i.e. weights) of the network
2. Different input features (i.e. numerical values that could, for example, represent pixels in an image) do not contribute equally to the objective, or loss, function.

These two observations necessitate some sort of perturbation to the gradient updates so as not to reveal anything about the underlying data used to generate the gradients. The key idea behind DSSGD is a partial publishing of gradient data to the network. Instead of sending the full gradient from a given training cycle, a node will send a **selective parameter update**, where the selection is either random or done according to some heuristic. Knowing that received gradients have been selected randomly or perturbed somehow makes it difficult for any attacker to discern anything about the underlying data.

Trust

In our project, trust between nodes means some level of confidence, measured here as lack of suspicion about an update received from a connected peer. Suspicion in this case relates to the notion of a poisoning attack, which is defined in adversarial machine learning literature as an attempt to subvert the learning process by injecting misleading data into the network. The desired behavior in our network is for a node to reject data it is suspicious of being poisonous under certain conditions.

In the paper *AUROR: Defending Against Poisoning Attacks in Collaborative Deep Learning Systems*, the authors describe a system that can reliably develop suspicions about incoming data from connected learning nodes. The authors use the same privacy model as mentioned in the preceding section on privacy, where nodes communicate **masked features**. They are called masked because they are not the raw features fed into each learning node, but the resulting gradients from training on that data. The key insight made by the authors is that so-called **indicative features** are a subset of all masked features that display a *different distribution* in an attack setting. That is, if a node receives some masked feature from a peer that is an indicative feature, it can examine the distribution on the incoming data. If it is sufficiently different from the expected distribution for that indicative feature, the receiver marks the node it received the anomalous data from as “suspicious”.

With enough “suspicious” markings associated with a given node, and according to certain thresholds, a node can be blacklisted by the network. Given the peer-to-peer nature of our system, one nodes blacklisting of another can be spread through the network according to the gossip protocol described above, suggesting that peers also blacklist the suspect node.

One possible alteration we may make to the approach in the AUROR paper is to allow a peer to poll some subset of its neighbors to see if they too view a given update as being anomalous. If a majority view it as non-anomalous, it should be okay for the node to accept the update. The fact that something appears anomalous to a given node may simply mean that the node in question has not seen that kind of data before, and so should poll other neighbors to get a broader view of the data before discarding it.

The authors use the **K-Means** algorithm to group indicative features out of the total set of masked features, and this will be something that needs to be implemented for our project.

Application API

In our system each learning node in the P2P network has an attached application that is responsible for feeding it data. Each node is connected to exactly one application, no more and no less.

The API will tentatively provide at minimum, the following behaviour to an application:

```
Train(data) -> ok, err
```

a. Here the application sends some part of its local data to its connected node, which then uses the data to perform local training and distribute the results to the network.

```
Get(data) -> result, err
```

b. Here the application seeks to actually make use of the neural network, which depending on the data context may mean different things. In a classification setting, for example, the application may pass in a description of some financial asset, with the network returning a classification of **Risky** or **NotRisky**.

Assumptions

We assume the following properties of our system:

- The server never fails
- No more than 30% of the network is malicious. This is the assumption made in the AUROR paper and we maintain that assumption here.
- Learning nodes and applications exit on the same host, and fail together
- All applications that interact with the system at one time have some subset of a global dataset

Testing

General system testing: As a general test of our system, we want to be able to display that the learning is indeed collaborative across the network. One way of achieving this would be to start our system with some number of nodes whose connected applications are given some partition of a global dataset. These partitions may be distinct or partially overlapping. After a set period of training on these mostly distinct datasets, we would want to introduce new data to a particular application (and therefore its learning node) and verify that this application's learning node can properly classify/perform regression (whatever is dictated by the particular neural network) the new data. This new data will have belonged to a different application's original partition, so the goal is to see that an application benefits from the training progress made by its peers.

Poisoning testing: Another crucial aspect of our system to test is its resistance to poisoning attacks. In order to test this, we will set up our network with some number of non-malicious nodes, along with another group of malicious nodes that attempt to subvert the network. We will design the malicious nodes to, for example, attempt to make the network mis-classify some data. We will then verify that despite poisoning attempts, the network is not generally poisoned.

Finally, we will iteratively perform our system testing on Azure to ensure that our system works in a production environment.

References

1. Federated Learning: Collaborative Machine Learning without Centralized Training Data. https://research.googleblog.com/2017/04/federated-learning-collaborative.html
2. SWIM: Scalable Weakly-consistent Infection-style Process Group Membership Protocol. https://pdfs.semanticscholar.org/8712/3307869ac84fc16122043a4a313604bd948f.pdf
3. Distributed Algorithms: Epidemic Dissemination. http://disi.unitn.it/~montreso/ds/handouts/05-epidemic.pdf
4. AUROR: Defending Against Poisoning Attacks in Collaborative Deep Learning Systems. http://www.comp.nus.edu.sg/~shruti90/papers/auror.pdf
5. Privacy-Preserving Deep Learning. https://www.cs.cornell.edu/~shmat/shmat_ccs15.pdf