Journal of Network and Computer Applications 116 (2018) 1–8

Contents lists available at ScienceDirect

Journal of Network and Computer Applications

journal homepage: www.elsevier.com/locate/jnca

Distributed learning of deep neural network over multiple agents

Otkrist Gupta*, Ramesh Raskar

Massachusetts Institute of Technology, 77 Massachusetts Ave, Cambridge, MA 02139, USA

ARTICLE INFO

Keywords:
Multi party computation
Deep learning
Distributed systems

ABSTRACT

In domains such as health care and finance, shortage of labeled data and computational resources is a critical issue while developing machine learning algorithms. To address the issue of labeled data scarcity in training and deployment of neural network-based systems, we propose a new technique to train deep neural networks over several data sources. Our method allows for deep neural networks to be trained using data from multiple entities in a distributed fashion. We evaluate our algorithm on existing datasets and show that it obtains performance which is similar to a regular neural network trained on a single machine. We further extend it to incorporate semi-supervised learning when training with few labeled samples, and analyze any security concerns that may arise. Our algorithm paves the way for distributed training of deep neural networks in data sensitive applications when raw data may not be shared directly.

1. Introduction

Deep neural networks have become the new state of the art in classification and prediction of high dimensional data such as images, videos and bio-sensors. Emerging technologies in domains such as biomedicine and health stand to benefit from building deep neural networks for prediction and inference by automating the human involvement and reducing the cost of operation. However, training of deep neural nets can be extremely data intensive requiring preparation of large scale datasets collected from multiple entities (Chervenak et al., 2000; Chuang and Sirbu, 2000). A deep neural network typically contains millions of parameters and requires tremendous computing power for training, making it difficult for individual data repositories to train them.

Sufficiently deep neural architectures needing large supercomputing resources and engineering oversight may be required for optimal accuracy in real world applications. Furthermore, application of deep learning to such domains can sometimes be challenging because of privacy and ethical issues associated with sharing of de-anonymized data. While a lot of such data entities have vested interest in developing new deep learning algorithms, they might also be obligated to keep their user data private, making it even more challenging to use this data while building machine learning pipelines. In this paper, we attempt to solve these problems by proposing methods that enable training of neural networks using multiple data sources and a single supercomputing resource.

2. Related work

Deep neural networks have proven to be an effective tool to classify and segment high dimensional data such as images (Krizhevsky et al., 2012), audio and videos (Karpathy and Fei-Fei, 2015). Deep models can be several hundreds of layers deep (He et al., 2016), and can have millions of parameters requiring large amounts of computational resources, creating the need for research in distributed training methodologies (Dean et al., 2012). Interesting techniques include distributed gradient optimization (Mcdonald et al., 2009; Zinkevich et al., 2010), online learning with delayed updates (Langford et al., 2009) and hashing and simplification of kernels (Shi et al., 2009). Such techniques can be utilized to train very large scale deep neural networks spanning several machines (Agarwal and Duchi, 2011) or to efficiently utilize several GPUs on a single machine (Agarwal et al., 2014). In this paper we propose a technique for distributed computing combining data from several different sources.

Secure computation continues to be a challenging problem in computer science (Sood, 2012). One category of solutions to this problem involve adopting oblivious transfer protocols to perform secure dot product over multiple entities in polynomial time (Avidan and Butman, 2006). While this method is secure, it is somewhat impractical when considering large scale datasets because of resource requirements. A more practical approach proposed in Avidan and Butman (2006) involves sharing only SIFT and HOG features instead of the actual raw data. However, as shown in (Dosovitskiy and Brox), such feature vectors can be inverted very accurately using prior knowledge

* Corresponding author.
E-mail address: otkrist@mit.edu (O. Gupta).

https://doi.org/10.1016/j.jnca.2018.05.003
Received 11 October 2017; Received in revised form 30 April 2018; Accepted 6 May 2018

of the methods used to create them. Neural networks have been shown to be extremely robust to addition of noise and their denoising and reconstruction properties make it difficult to compute them securely (Vincent et al., 2010). Neural networks have also been shown to be able to recover an entire image from only a partial input (Pathak et al.), rendering simple obfuscation methods inert.

Widespread application of neural networks in sensitive areas such as finance and health, has created a need to develop methods for both distributed and secure training (Secretan et al., 2007; Chonka et al., 2011; Wu et al., 2007) and classification in neural networks. Under distributed and secure processing paradigms, the owner of the neural network doesn't have access to the actual raw data used to train the neural network (Barni et al., 2006). This also includes secure paradigms in cloud computing (Karam et al., 2012; Subashini and Kavitha, 2011), virtualization (Mackay et al., 2012) and service oriented architectures (Baker et al., 2015). The secure paradigms may also extend to the neural activations and (hyper)parameters. Such algorithms form a subset inside the broader realm of multi-party protocol problems involving secure computation over several parties (Goldreich et al., 1987; Yao, 1986). Some interesting solutions include using Ada-boost to jointly train classifier ensembles (Zhang and Zhong, 2013), using random rotation perturbations for homomorphic pseudo-encryption (Chen and Liu) and applying homomorphic cryptosystem to perform secure computation (Orlandi et al., 2007).

3. Theory

In this paper we propose new techniques that can be used to train deep neural networks over multiple data sources while mitigating the need to share raw labeled data directly. Specifically we address the problem of training a deep neural network over several data entities (Alice(s)) and one supercomputing resource (Bob). We aim at solving this problem while satisfying the following requirements:

1. A single data entity (Alice) doesn't need to share the data with Bob or other data resources.
2. The supercomputing resource (Bob) wants control over the architecture of the Neural Network(s)
3. Bob also keeps a part of network parameters required for inference.

In upcoming sections we will show how to train neural networks between multiple data entities (Alice(s)) and a supercomputing resource (Bob). Techniques will include methods which encode data into a different space and transmit it to train a deep neural network. We will further explore how a third-party can use this neural network to classify and perform inference. Our algorithm can be run using one or multiple data entities, and can be run in peer-to-peer or centralized mode. Please see Fig. 1 for the schematic depiction of algorithm modalities.

3.1. Distributed training over single entity

We will start by describing the algorithm in its simplest form which considers training a neural network using data from a single entity and supercomputing resource. Let us define a deep neural network as a function F, topologically describable using a sequence of layers $\{L_0, L_1, \ldots L_N\}$. For a given input ($data$), the output of this function is given by $F(data)$ which is computed by sequential application of layers $F(data) \leftarrow L_N(L_{N-1}\ldots(L_0(data)))$.

Let $G_{loss}(output, label)$ denote the customized loss function used for computing gradients for the final layer. Gradients can be backpropagated over each layer to generate gradients of previous layers and to update the current layer. We will use $L_i^T(gradient)$ to denote the process of backpropagation over one layer and $F^T(gradient)$ to denote backpropagation over the entire Neural Network. Similar to forward propagation, backpropagation on the entire neural network is comprised of sequential backward passes $F^T(gradient) \leftarrow L_1^T(L_2^T \ldots (L_N^T(gradient)))$. Please note that the backward passes will require activations after the forward pass on individual perceptrons.

Finally, $Send(X, Y)$ represents the process of sending data X over the network to entity Y. In the beginning, Alice and Bob initialize their parameters randomly. Alice then iterates over its dataset and transmits encoded representations to Bob. Bob then computes losses and gradients and sends the gradients back to Alice. Algorithm 1 describes how to train a deep neural classifier using a single data source.

3.1.1. Correctness

Here we analyze if training using our distributed algorithm produces the same results as a normal training procedure. Under a normal training procedure we would first compute forward pass $output \leftarrow F(data)$ followed by computation of loss gradients $gradients \leftarrow G(output, label)$. These gradients will be backpropagated to refresh weights $F' \leftarrow F^T(gradients)$.

Algorithm 1
Distributed Neural Network training over 2 agents.

Initialize:
  $\phi \leftarrow$ Random Initializer (Xavier/Gaussian)
  $F_a \leftarrow \{L_0, L_1, \ldots L_n\}$
  $F_b \leftarrow \{L_{n+1}, L_{n+2}, \ldots L_N\}$
Alice randomly initializes the weights F_a using ϕ
Bob randomly initializes the weights of F_b using ϕ
while Alice has new data to train on **do**
  Alice uses standard forward propagation on data
  $\triangleright\ X \leftarrow F_a(data)$
  Alice sends n^{th} layer output X and label to Bob
  $\triangleright\ Send((X, label), Bob)$.
  Bob propagates incoming features on its network
  $\triangleright\ output \leftarrow F_b(X)$
  Bob generates gradients for its final layer
  $\triangleright\ gradient \leftarrow G'(output, label)$
  Bob backpropagates the error in F_b until L_{n+1}
  $\triangleright\ F_b', gradient' \leftarrow F_b^T(gradient)$
  Bob sends gradient of L_n to Alice
  $\triangleright\ Send(gradient', Alice)$
  Alice backpropagates gradients received
  $\triangleright\ F_a', \_ \leftarrow F_a^T(gradient')$
end while

Since forward propagation involves sequential application of individual layers we concur that $F(data)$ is same as $F_b(F_a(data))$. Therefore the process of sequential computation and transmission followed by computation of remaining layers is functionally identical to application of all layers at once. Similarly because of the chain rule in differentiation, backpropagating $F^T(gradients)$ is functionally identical to sequential application of $F_a^T(F_b^T(gradients))$. Therefore, we can conclude that our algorithm will produce identical results to a normal training procedure.

Algorithm 2
Distributed Neural Network over N + 1 agents.

Initialize:
  $\phi \leftarrow$ Random Initializer (Xavier/Gaussian)
  $F_{a,1} \leftarrow L_0, L_1, \ldots L_n$
  $F_b \leftarrow L_{n+1}, L_{n+2}, \ldots L_N$
$Alice_1$ randomly initializes the weights of $F_{a,1}$ using ϕ
Bob randomly initializes the weights of F_b using ϕ
Bob sets $Alice_1$ as last trained

(continued on next page)