Summary of The BPCryptonets

Dian Li

2017/20/11

1 Introduction

Microsoft Research's CryptoNets makes it possible to apply machine learning on encrypted data. And their Neural Networks is a type of convolution neural networks which is a deep networks. In this work, I present a light-weight BPCryptoNets which has only one hidden layer compared to the CryptoNets' four hidden layers. The neural networks is actually a kind of back propagation neural networks which has only three layers in total. Due to the high overhead of the calculation in fully connected layer and the shallow networks the BPCryptoNets achieve 95% accuracy and can make 4 predictions per hour on a single PC compared to the 99% accuracy and 51000 prediction per hour of Microsoft's CryptoNets.

Unlike Microsoft's application scene, I just assume that there are two parties in the BPCryptoNets prediction. The client encrypted the privacy data and send it to the prediction provider, referred to as the cloud. The cloud is able to compute the prediction over the encrypted data and send back the results that the client can decrypt and read.

The main ingredients of the BPCryptoNets are Back propagation Neural Networks and homomorphic encryption. I make the assumption that the cloud already has the BPNeraulNetwork model, it means that the networks was previously trained using a set of unencrypted data. However, training models on encrypted data which referred as privacy data-mining is still a big challenge. I implement a privacy preserving Linear Means Classification which can only apply simple data-mining several days ago. So it still has to train the networks on unencrypted data in order to get more precise parameters and iterations. In this work, I use the SEAL library for homomorphic encryption which is a leveled homomorphic encryption scheme of Junfeng and Frederic Vercauteren.

2 Neural Networks

I use the shallow Back propagation Neural Networks which has three layers. The bottom-most layer is the input layer, each node of the input layer is fitted with proper weights for training. The second layer is a hidden layer which contains an activation function for the error propagation. And the activation function is the key point of a neural networks. The top-most layer are the output layer of the neural networks. In the last page there are graphical representation of the neural networks.

In my setting, I don't use the ordinary Sigmoid activation function due to its high degree that the homomorphic encryption can't evaluate it. According to the Microsoft Research's advice, I use the lowest-degree non-linear polynomial function, which is the square function: $sqr(z) = z^2$. However, the change of the activation brings the change of the way that the error back propagation through the networks in the training period. In order to get proper weights for prediction, I must change the origin training algorithm, it's a big challenge for me. And I solve it and get the correct approach in the end.

2.1 Forward Propagation Stage

Suppose that each input x_k of the input layer has weight $w_{k,i}$, where $k \leq l$, $l$the number of nodes in input layer $i \leq m$, m is the number of nodes in hidden layer. From the input layer to the hidden layer there are two calculations:

1. *Weighted sum* :$y'_i = \sum_k x_k * w_{k,i}$
2. *Square* :$y_i = (y'_i)^2$

Thus the inputs of the hidden layer is y_i.From the hidden layer to the output layer, assuming that the output layer has n nodes each of weight $w_{i,j}$, there are two calculations too:

1.*Weighted sum* :$z'_j = \sum_i y_i * w_{i,j}$

2.*Sigmoid* :$z^j = \frac{1}{1+exp(-z'_j}$

Through the above four calculations we get the output of the predictions, however, during the training stage, we use the testing data to adjust the initial weights such that we can get proper weights which are suitable for testing data. In this experiment, I don't add bias term in the weighted sum, some networks also add a bias term to the result of the weighted sum. Now, we define an error function:

$$E(w) = \frac{1}{2}\sum_{j=0}^{n-1}(z_j - d_j)^2$$

where d_j is the standard output of the testing data. Thus, according to the error function we calculate the error growth with different weights therefore to get the relationship between the gradient and the error growth. So, we can modify the weights repeatedly to acquire more precise weights with smaller error result.

2.2 Back propagation Stage

1.Output layer to hidden layer For the weight $w_{i,j}$. We have:

$$\begin{aligned}
\Delta w_{i,j} &= \frac{\partial E(w)}{\partial w(i,j)} \\
\frac{\partial E(w)}{\partial w_{i,j}} &= \frac{1}{\partial w_{i,j}} \cdot \frac{1}{2} \sum_{j=0}^{n-1} (z_j - d_j)^2 \\
&= (z_j - d_j) \cdot \frac{\partial d_j}{\partial w_{i,j}} \\
&= (z_j - d_j) \cdot f(z_j')(1 - f(z_j')) \cdot \frac{\partial z_j'}{\partial w_{i,j}} \\
&= (z_j - d_j) \cdot f(z_j')(1 - f(z_j')) \cdot y_i \\
&= \delta_{i,j} \cdot y_i
\end{aligned}$$

In the equation f is the sigmoid function, and $\delta_{i,j} = (z_j - d_j) \cdot f(z_j')(1 - f(z_j')$. Let η the learning rate we can get the updating formula of the weights $w_{i,j} = w_{i,j} - \eta \cdot \delta_{i,j} \cdot x_i$

2.Hidden layer to Input layer:

$$\begin{aligned}
\frac{\partial E(w)}{\partial w_{k,i}} &= \frac{1}{\partial w_{k,i}} \cdot \frac{1}{2} \sum_{j=0}^{n-1} (z_j - d_j)^2 \\
&= \sum_{j=0}^{n-1} \cdot f'(z_j') \cdot \frac{\partial z_j'}{\partial w_{k,i}} \\
&= \sum_{j=0}^{n-1} (z_j - d_j) f'(z_j') \cdot \frac{\partial z_j'}{\partial y_i} \cdot \frac{\partial y_i}{\partial y_i'} \cdot \frac{\partial y_i'}{\partial w_{k,i}} \\
&= \sum_{j=0}^{n-1} (z_j - d_j) f(z_j')(1 - f(z_k')) \cdot w_{i,j} \cdot 2y_i' \cdot x_k \\
&= \delta_{ki} \cdot x_k
\end{aligned}$$

Thus we can get the updating formula of weights $w_{k,i}$:

$$w_{k,i} = w_{k,i} - \eta \cdot \delta_{k,i} \cdot x_k$$

where $\delta_{k,i} = \sum_{j=0}^{n-1} \delta_{i,j} \cdot w_{i,j} \cdot 2y_i' \cdot x_k$

From the hidden layer to input layer, unlike ordinary BP Neural Networks the difference is that we use the square activation function of degree only 2. On the one hand, due to the low degree of the activation function it is easy for us to evaluate it homomorphically in the prediction. On the other hand, the convergence rates of the square function is faster than using the ordinary sigmoid function, and it seems that the square function is better than sigmoid function in applying neural networks for image data. When I start to learn the related knowledge of CryptoNets the part of Gradient Descent confused me a long time, I don't understand how to change the sigmoid function to square function, however I finally understand it and the results of using square activation function looks not so bad. Using the gradient descent method I input the 60000 samples of the MINIST set, and I can get the proper weights of 95% accuracy of prediction results.

2.3 Description of the BP Neural Networks

1.Input layer :The input image is of 28×28 pixel. So there are 748 nodes in the input layer in total. In my setting, the hidden layer has 128 nodes.

2.Square activationlayer :This layer squares the weighted sum value from input layer.

3.Output layer :This layer has 10 nodes corresponding to the digits 0-9 and the standard output is a vector of hamming weight 1, for example, if the input is 5, then the standard output should be $(0,0,0,0,0,1,0,0,0,)$.

4.Sigmoid layer :In the training stage we need this layer to restrict the value in $(0,1)$. And in the prediction we don't need to evaluate this layer.

3 Homomorphic encryption

In my implementation, I use the SEAL library which has all API for the FV encryption scheme including the encryption, decryption and encoding. Here I demonstrate the encryption scheme:

Setup$(1^\lambda, 1^L)$: L is the multiplicative depth, $n = (\lambda, L)$, error distribution $\chi =$

$\chi(\lambda, L)$ appropriately for LWE that achieves at least 2^{λ}security against known attacks. $m = m(\lambda, L) = O(n \log_q)$

SecretKeyGen(λ): Sample $\vec{s} \leftarrow \mathbb{R}_q$. Output $sk = \vec{s}$.

PublicKeyGen(sk): Sample $\vec{a} \leftarrow \mathbb{R}_q$, and $\vec{e} \leftarrow \mathbb{R}_2$. Output the pk=$(\left[-(\vec{a}\vec{s} + \vec{e})\right]_q, \vec{a}$).

Encrypt(pk,m): For $m \in \mathbb{R}_t$, let pk=$(\vec{p_0}, \vec{p_1})$. Sample $\vec{u} \leftarrow \mathbb{R}_2$, and $\vec{e_1}, \vec{e_2} \leftarrow \mathbb{R}_2$. Compute:

$$ct = (\left[\left\lfloor\frac{q}{t}\right\rfloor m + \vec{p_0}\vec{u} + \vec{e_1}\right]_q, \left[\vec{p_1}\vec{u} + \vec{e_2}\right]_q)$$

Decrypt(ct,sk): Set $\vec{s} = sk$,$\vec{c_0}$= $\mathsf{ct}[0]$, and $\vec{c_1}$= $\mathsf{ct}[1]$. Output

$$\left[\left\lfloor\frac{t}{q}\left[\vec{c_0} + \vec{c_1}\vec{s}\right]_q\right\rceil\right]_t$$

In the encryption scheme, $\mathbb{R}_t$=$\mathbb{Z}_t[x]/(x^n + 1)$ and $\mathbb{R}_q$=$\mathbb{Z}_q[x]/(x^n + 1)$. The security level of the system depends on the parameter n, q, t and the mount of noise added. In applying neural networks, a common operation is to add or multiply some value, which is derived from the data with some known constant. The main parameters defining the cryptosystem are the plaintext modulus t, the coefficient modulus q and the degree n of the polynomial modulus $(x^n + 1)$. The value used are $t = 2^15$, the coefficient modulus $2^{116} - 2^{18} + 1$ and the polynomial modulus $x^4096 + 1$ allow for computing the desired network correctly.

4 Results

I tested BPCryptoNets on the MINST dataset. This dataset consists of 60000 images of hand written digits. Each image is a 28×28 pixel array. And I represent the image in its gray level for 0 is white pixel and 1 is black pixel. The accuracy of the training network is 95%. Because every pixel is encrypted as a single ciphertext the size of an encrypted image is large. Each image requires $28 \times 28 \times 128$ bytes or 98 MB. The response of the BPCryptoNets contains only 10 values and therefore the message size is 10×128 which is 1.25MB.

The other details of the result I represent them in the following several tables. From table 1 we can see the input layer for weighted sum dominate the time to apply BPCryptoNets because the input layer to hidden layer is a full connected layer unlike the Convolution Neural Networks. Therefore, the input layer needs $784 \times 128 = 100352$ times homomorphic mutiplication in total which is a high overhead for the neural networks. And it will improve the performance with batching technique which is follow-up work for me.

Table 1:Breakdown the time it takes to apply BPCryptoNets to one sample.

Layer	**Description**	**Time to compute**
Input layer	Weighted sums layer with 784 inputs and send them to square layer. Finally there are 128 outputs to hidden layer.	805.18seconds
Square layer	Square each of the 784 weighted sums.	1.98 seconds
Output layer	Weighted sum that generates 10 outputs(corresponding to the 10 digits) from 128 outputs of the square layer	17.33 seconds

Because of the high overhead from input layer to the hidden layer, the BPCryptoNets can only predict 4 digits per hour, which is not so good as Microsoft's CryptoNets' performance.

Table 2:The performance of BPcryptoZNets for MINST

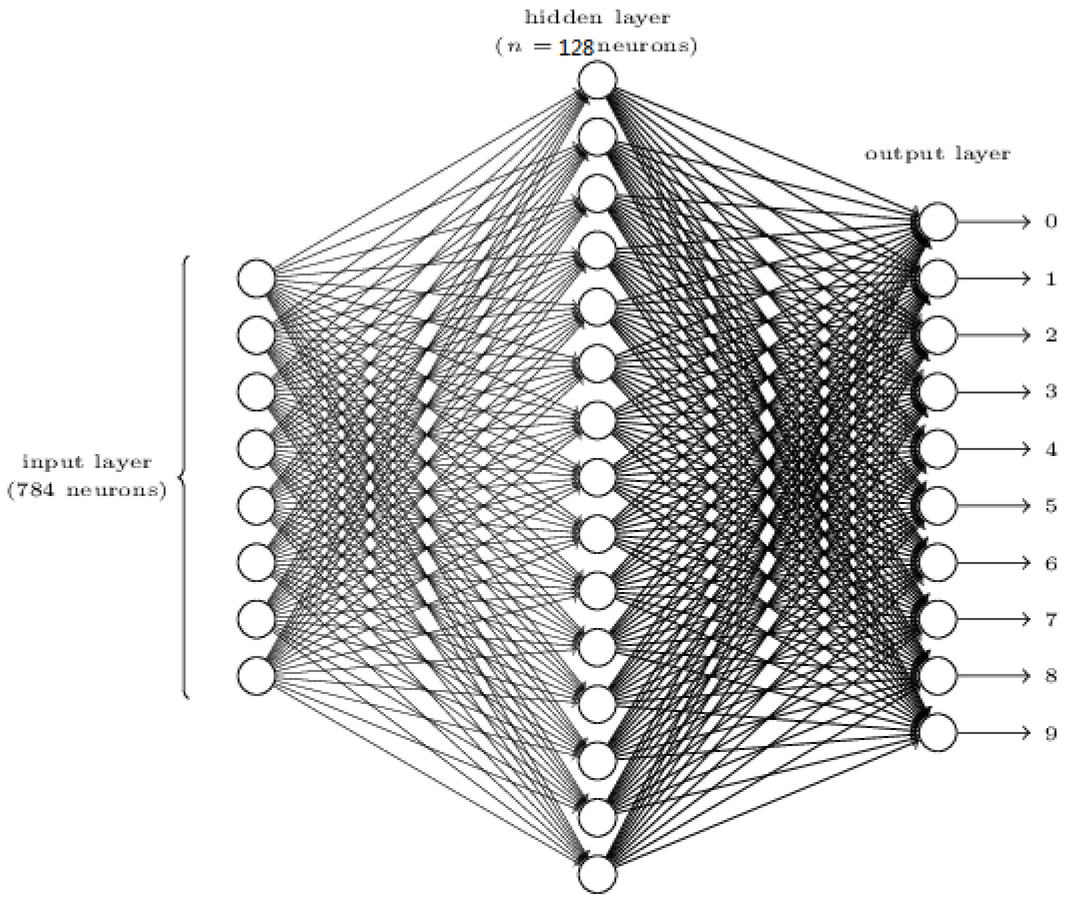
Stage	**Latency**	**Throughput**
Encoding+Encryption	24.35 seconds	148 per hour
Network application	824.49 seconds	4 per hour
Decryption+Decoding	0.071 seconds	50704 per hour

The large amounts of nodes in the input layer also brings the problem of large message size from owner to cloud like in table 3. Batching is also a feasible method to solve the defect.

Table 3:MessageSize

	Message Size
$Owner \rightarrow Cloud$	98.19 MB
$Cloud \rightarrow Owner$	1.875MB

Network.png


hidden layer
($n =$ 128 neurons)
output layer
input layer
(784 neurons)
0
1
2
3
4
5
6
7
8
9