Distributed LSTM autoencoder

Tao Deng[1], Shima Khoshraftar[2] and Aijun An[3]

ABSTRACT

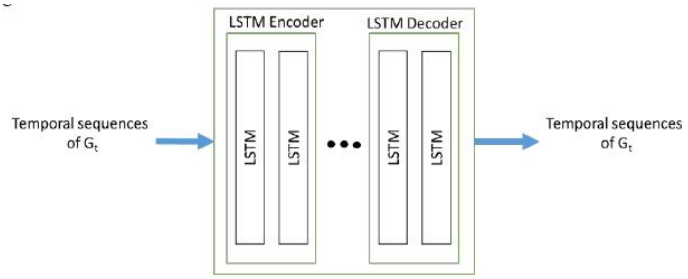
LSTM autoencoder is an important part of LSTM-Node2vec, which is a bigger model that LSTM and used for graph embedding based on basic neural network model. There including the tasks like "link prediction, anomaly detection, node classification and outperforms state-of-the-art models in most of the case". [1] In this project, we focus on LSTM autoencoder and how it works in a Synchronous Parallel distributed way by using Tensorflow. Our LSTM models has been created in different levels, start from simple to complicated and which need to run on both CPU and GPU server. The simple LSTM autoencoder is based on simple neural network model which designed for acquainting the key point of LSTM autoencoder. And the complicated LSTM autoencoder is translating from the existing Keras code by changing the relevant functions. This version is working with Node2vec graph embedding, which is more like a Seq2seq autoencoder and needs to compile with the folder "l10" which contains the value and nodes of the graphs. Because the network structure in Kears is different with the network structure in Tensorflow, so the process of translation is very complex.

KEYWORDS

LSTM autoencoder, TensorFlow, Keras, Node2vec, Seq2seq, Graph embedding, Synchronous Parallel distributed way.

1. Introduction

In recent years, data mining and machine learning are very hot topics, LSTM autoencoder is one of the famous tasks from those topics. LSTM means Long Short Term Memory, which is a type Recurrent Neural Network. LSTM's popular use is to produce the representation of words. Because of a collection of phrases, an LSTM autoencoder is trained to take a phrase as input and output the same phrase. So as the node representation, LSTM also can learn node representation in graphs for graph embedding. For training model and the structure, LSTM autoencoder comprises of one encoder LSTM layer and one decoder LSTM layer. As an example, here is the simple model diagram: [1]



Then is TensorFlow, which is a free and open-source software library and it is proposed by Google. People usually use TensorFlow for analyze and train neural network models, it is one of most popular way in recent years.

Another one is Keras, which is another Neural Network API. Compared to TensorFlow, Keras is in a much higher level, it can be running and executing when using TensorFlow, CNTK or Theano as the backend. Although TensorFlow has a Keras library named "tf.keras", that is still different with the original Keras API, which cannot easily translate to each other. Overall, TensorFlow is much more flexible than in Keras. Keras can quickly build and test the neural network, but because it is a higher level than TensorFlow, its flexibility and functionality are a little bit inferior to TensorFlow. The reasons are TensorFlow can offer some low-level library and more advanced operations which make the TensorFlow easier to use and achieve.[3]

EECS 4080 project report

[1] Student in Lassonde School of Engineering, York University, Toronto, Canada, dengtao731@gmail.com

[2] Department of Electrical Engineering and Computer Science, York University, Toronto, Canada, shima.khoshraftar@gmail.com

[3] Department of Electrical Engineering and Computer Science, York University, Toronto, Canada, ann@eecs.yorku.ca

Beside those two, there is a scalable feature learning for network which called “node2vec”, it is proposed by Aditya Grover. “Node2ve” is an algorithmic framework, which can learn continuous feature representations for the nodes. By optimizing a neighborhood maintaining goal, the “node2vec” structure knows low-dimensional depictions for nodes in a graph. The algorithm adapts by simulating partial random walks to different concepts of network neighborhoods.[2]

In this project, we still focus on the two different strategies “TensorFlow” and “Keras” for LSTM autoencoder. They basically have done the same thing but use different way. Substantially, their differences are in the functions. In what we have done in this project, Keras and TensorFlow have different training and fitting function. Our major contributions in this project are:

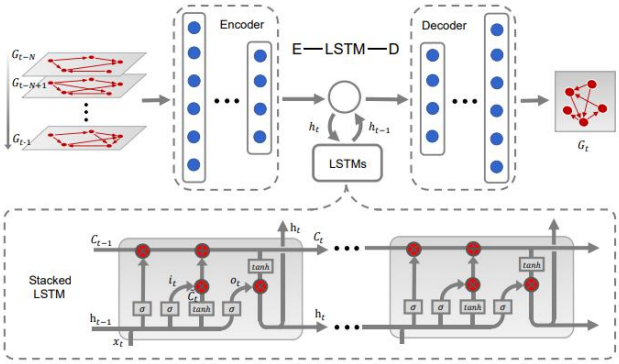
- Implement a TensorFlow based LSTM autoencoder model by translating the existing Keras based LSTM autoencoder model
- Implement a TensorFlow based LSTM autoencoder model by translating the existing Keras based LSTM autoencoder model in a distributed way
- Test the two TensorFlow LSTM autoencoder model from above in both CPU and GPU

The content section 2 which will provide the related work for this project. In section 3, we will describe our ideas for the designs, implementations and the reasons why it should be designing like this. In section 4, we will discuss the experimental results. In the last section, we will provide some possible open issues which still need to be solved and improved in the future.

2. Related Works

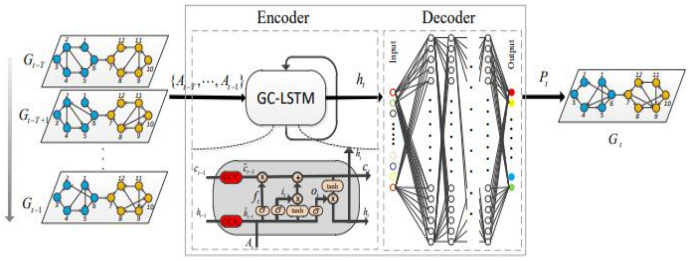
There are some widely uses for LSTM autoencoder, three of the most popular extension models are “Bidirectional Long Short Term Memory Recurrent Neural Network (BLSTM-RNN)”, “Graph Convolution Embedded Long Short Term Memory (GC-LSTM)” and “Encoder-Long Short Term Memory-Decoder (E-LSTM-D)”. [4][5][6][7][8] E-LSTM-D and GC-LSTM are used for dynamic link predictions. And BLSTM-RNN is used for learning distributed word representation.

E-LSTM-D is “a deep learning framework for dynamic network link prediction and a combination of encoder-decoder architecture[1] and stacked LSTM[2]”. [4] Its function is predicting links in the dynamic network, which has an encoder-decoder architecture to assist the model in learning network representation and reconstructing the graph on the basis of data obtained. Its overall framework shows like below, the encoder maps them to the latent room of the reduced dimensions. Each graph is converted into a matrix representing the characteristics of the structure. The stacked LSTM consist of multiple LSTM cells which learn from the extracted characteristics patterns of network evolution. Then the decoder builds the function charts obtained home into the initial room in order to get output graph. [4]



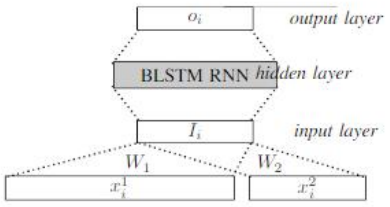
GC-LSTM is a new deep learning model which offering a novel end-to-end dynamic link prediction, which is also able to manage connections that will occur or vanish and managing high-dimensional, time-dependent and scarce organization pattern data effectively. GC-LSTM follows the deep learning model structure which consist of an encoder and a decoder. However, in this model, LSTM only would occur in the encoder. The overall framework is shown below. The encoder model is GCN (Graph Convolution Network) embedded LSTM model which uses GCN to learn the network structure of both cell and hidden layer state of each moment snapshot and uses LSTM to know the state of each link's temporary information. The decoder model is a network of fully connected layers to convert extracted features mapping back to the initial space. GC-LSTM will generate the expected network and execute a universal connection forecast. [5]

1. encoder-decoder architecture: “Inspired by Autoencoder, which can efficiently learn representations of data in a supervised way.” [4]
2. stacked LSTM: “As a special kind of RNN, learn long-term dependencies and use of LSTM cells to store long-term memory and filter out the useless information.” [4]



BLSTM-RNN is based on the distributed representation of phrases, the former can also be enhanced by studying the latter easier. It is used for conducting a tagging assignment with only correct and incorrect two types of tags. The input is a series of phrases that is a standard phrase, replacing several phrases with uniformly random selected phrases from the vocabulary list (Input composed of word identities and connected with weight matrixes then updates them for training).

BLSTM-RNN usually learns representations of a feedforward network and learns parameters of normalization rather than representations. [7]

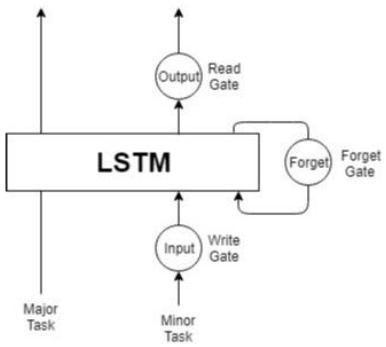


3. Designs and Implementations

In this project, I have designed programs in different difficulties. However, the main job is converting the Keras code to TensorFlow version and run it in both CPU and GPU. After this step, we change the design for the TensorFlow version to the distributed way and test it on GPU as well. Beside the major task, I have designed many small programs for getting familiar with LSTM autoencoder, [11][12][13][14][15][16][17][18][19][20][21][22] I will choose two typical programs as example to introduce at below.

In order to know LSTM autoencoder model, I have to know basic neural network model first. So, I choose the simple BP network model in CPU version. For the input data, I use the basic MNIST dataset. I set the input as 784 because MINST data has 28x28 pixels and output as 10 because there are 10 number in the MINST dataset (0-9). For the neural network model, the user has to set relevant activation function, calculate the loss and set appropriate optimizer for training the model and getting the accuracy. So as the simple model I have done, I basically use the most common way. For activation function, I use “relu” and “softmax” because in my model, the number of neurons is not large so use them to speed up then computation then use “reduce_mean” function in TensorFlow to get the loss, finally use Gradient Descent Optimizer, set appropriate learning rate to minimize the loss. The train the model to get the accuracy.

After that, I start to design and implement a LSTM neural network model in CPU version instead of the simple BP neural network model. For the structure of LSTM, it is like an advanced type of RNN. LSTM is designed for preventing gradient exploring or gradient vanishing. And basic LSTM contains three gates: Write Gate, Read Gate and Forget Gate. And in simple terms, there are two tasks in this model, one is major task which stands for the basic RNN task, and the other one is minor task which use the three gates to make connections with the major task. The Write Gate decides whether add the input into the main memory in major task or not. The Read Gate decides whether read the result from main memory in major task as output or not. The Forget Gate decides whether the model should forget the part of main memory in major task and use minor task instead of it in the main memory or not. [23]



The program I designed is still setting MINST data as input dataset. I set 10 classes because there are 10 classes of digits in the MINST dataset (0-9). For each MINST data input, the image shape is 28x28 pixels, I set input as 28 and steps as 28 as well because I input every single column each time, and which need to take 28 steps in total to

finish. Next, set up the placeholder to hold the input and output. And then define the weights and bias for the input and output, in order to let the value go through the cell and hidden layers appropriately. The basic structure for my program is “Input→First Hidden Layer (28, 128) → LSTM Cell → Second Hidden Layer(128, 10) → Output”. The build for the first hidden layer has three steps: 1. Reshaping the input data. 2. Using “matmul” (matrix multiply) for fitting the weight and bias I define. 3. Reshaping the result from the last step then send it to the LSTM cells. For the LSTM cell, I use the “BacisLSTMCell” in neural network, after initializing the states in the cell, we use “dynamic_rnn” instead of “rnn” because dynamic RNN is much more efficient. Then build the second hidden layer for receiving the results from the LSTM cell, but right now the result should be a tensor, which need “unpack” function to change the tensor to a list. Then use the list to fit the output weight and bias for outputting the final results. In this program, I still choose “softmax” activation function to get the cross entropy, “reduce_mean” function to calculate the loss and “Adam” optimizer to reduce the cost. For the training step, I decide to output the accuracy after every 20 training steps. [23]

After those exercises, we start to move to the major task. The first thing I have done is translating the existing Keras code to TensorFlow version. The given Keras code is designed for implementation of node2vec and building LSTM autoencoder for graph embedding. It is basically a sequence to sequence autoencoder, and there are 29 graphs in files (from grapg10 to graph38) in the provided folder “l10”. The Keras code is sending these files as input, then reads a sequence from each file (one-by-one in loop), converts them to one-hot encoding, then input and output to LSTM autoencoder for training. [24]

As I mentioned before, their building model steps and training steps are quite different. The Keras code includes some defined functions, but not all of them are used in the main function. So, I only need change two of these defined functions, which are “create_lstm” and “create_initialize_lstm”. Then avoid something happened like type mismatch in parameter, I change the function “get_walks_n2v” as well. So as the main function.

First of all, for function “create_lstm”, instead of building model directly, I set one LSTM cell for getting the cell embedding size first, then set up the input data and output target, initialize the state and use dynamic RNN to build overall structure. Next define the weight and bias, noted that the weight can be initialize in random value here. Then build one fully hidden layer for input graph, and inside this layer I follow the Keras code which use “softmax” activation function to get the loss, “reduce_mean” to calculate the total loss and “Adam” optimizer to minimize the total loss. Now the model has been created, based on Keras code, which should train it for 500 times. After that is the fitting method, but unlike Keras, there is no “tf.fit” function in TensorFlow. As the result, I choose to use “tf.Saver” [22] to save the model as “.ckpt” first, and then use “pywrap_tensorflow.NewCheckpointReader” function to read the model in order to achieve the fitting function. This function is the most common and popular way for TensorFlow to fit the model with sequence. There is one short cut here, we can get the weight directly by checking the network name. For this project, the name is 'rnn/multi_rnn_cell/cell_0/basic_lstm_cell/kernel'. The other reason I get the weight here is the main function in Keras code. In Keras code, the author gets the weight by using “model.layer.get_weight”. Same as the fitting method, you cannot use TensorFlow to get the weight directly. So, once we get the weight and return it on the defined function, we can use the weight directly in the main function in TensorFlow version.

Another function is “create_initialize_lstm”, the structure and implementation are very similar to “create_lstm”. The only difference is in this model, the weight should be initialized for node2vec, not in random. I have used “tf.variable_scope('change_weigths')” for replacing the first hidden layer and used “array[lstm_one_hot_list.index(node), :emb_size] = n2v_embs[n2v_nodes.index(node)]” for stating n2v weights as initials in LSTM, then use the same way as “create_lstm” to state, train and fit the model .

Moreover, in Keras code “get_walks_n2v” function, “walks = [map(str, walk) for walk in walks]” this lane will cause parameter mismatching in TensorFlow. So, I change to use a for loop and map the “str” and “walk”, which stands for the same meaning as the meaning of the Keras code. And in main function, because of I have already got the weight in the defined functions, so I use the weight value directly in the main function instead of Keras function “model.layer.get_weight”.

The last but the most important one is running the TensorFlow version that I translated in a distributed way through multiple GPUs. The only change from the previous one is add adding a function

called “average_gradients”. I learn it from Github for training model in distributed way through multiple GPU in parallel training by using TensorFlow. For this function, the goal is “calculating the average gradient for each shared variable across all models and provides a synchronization point across all models.” [25] The input is a list for the gradient and variable in each model, and the output is the list of gradient and variable in average for all the models.[25] I use this function in both defined functions “create_lstm” and “create_initialize_lstm” because those two functions are used for creating models. I use “tf.AUTO_REUSE” first because all GPU in the server need to share the network variable. Then use “range” to find the number of GPU in the server, and calculate the gradient in each GPU, then apply for “average_gradients” to calculate the average gradient. In this step, usually use "tf.device('/cpu:0'):" to set up the available device for executing, if there is no device can be used in the server or no device existing and some variables used in different GPU may cause extremely long execution or unusual complication errors. So there is one method that I find which called “tf.ConfigProto(allow_soft_placement=True)”, it can automatically choose one existing and useable device to make execution. That is perfectly solve this problem. At last, change the single device to multiple devices to run the code in main function.

4. Experimental Results

For the basic BP neural network model in CPU version, I decide use 500 in batch size, 300 hidden units and 0.5 as the learning rate. The output is starting at a very high accuracy which is around 0.7252, and the accuracy will continue increasing after each training iteration. The accuracy has obvious change for the first 5 training iterations, but the change is getting smaller after that. And the highest accuracy is the last training iteration I set which is around 0.9266. From the observation, basic BP neural network model takes about 147.6 seconds for outputting 20 training accuracy results. It is a little bit slow though the accuracy is not bad. So, for training a large dataset, BP network probably is not the first choice.

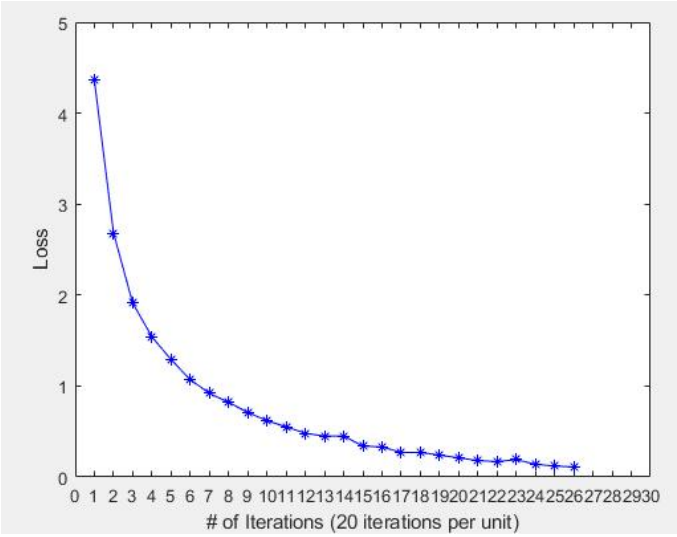
Then for the LSTM Autoencoder model which run in CPU, I set 128 as batch size, 128 hidden units and 0.001 as the learning rate. The output is starting at a low accuracy which is around 0.2656, but it increases dramatically after 3 training iteration which is becoming around 0.8828. It has an ascending flow in overall, but at some iteration, the accuracy will decrease in a small range. The highest accuracy is around 0.9922, but the at the last iteration I set, the accuracy is becoming 0.9687. For this model, there are 40 training iterations in total, it takes about 50.7 seconds. Compared with the basic BP neural network model, LSTM Autoencoder is much better in both speed and overall accuracy.

Then for the major tasks, I make some detailed comparisons and divide for three groups by the experimental results:

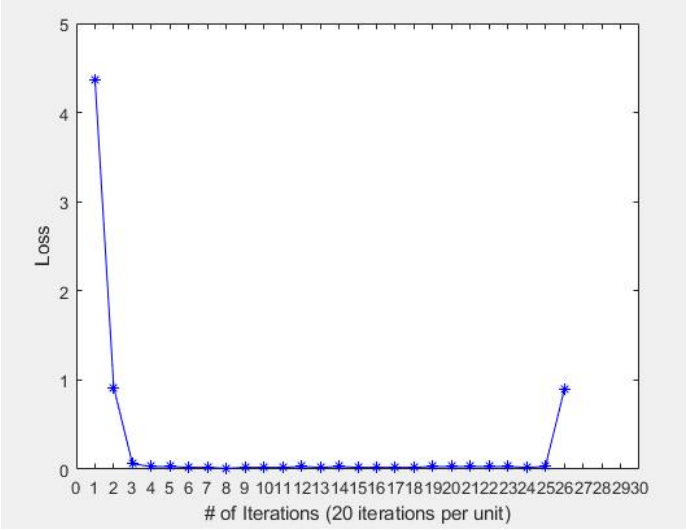
1. Simple LSTM Autoencoder model in CPU vs. Simple LSTM Autoencoder model in GPU
2. LSTM Autoencoder model in Keras (in GPU) vs. LSTM Autoencoder model in TensorFlow (in GPU)
3. LSTM Autoencoder model in GPU vs. LSTM Autoencoder model in distributed way in GPU

For the first group, the same LSTM Autoencoder model runs in GPU provide exactly the same output as it runs in CPU. However, the training speed is shrinking to around 38.9 seconds, which is much quicker than it runs in CPU.

For the second group, the Keras version takes about 90 seconds to finish 500 trainings for one graph input, and because every time for the training will provide different training results, so I just provide one possible situation. For the first 20 training iterations, the training loss decreases from 4.3695 to 2.6311, which stands for the increase of the accuracy; for the last 20 training iterations, the training loss decreases from 0.1425 to 0.1316, which also stands for the increase of the accuracy. So, for Keras version, the overall accuracy is increasing but existing very small decreasing, and the overall training loss is in a descending order but exist some very small increasing flows. Here is the possible plot to show the relation between loss and number of iterations for Keras code:



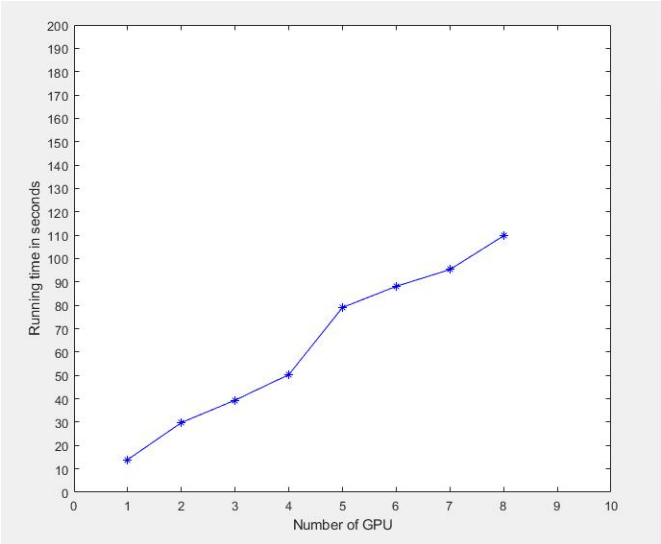
Compared with Keras version, TensorFlow version only takes about 15 seconds to complete the 500 trainings for one graph input. For the first 20 training iterations, the training loss decreases from 4.3765 to 0.9029, which also stands for the increase of the accuracy; and for the last 20 training iterations, the training loss increases from 0.0214 to 0.8979, that stands for a decrease in accuracy. Here is the possible plot to show the relation between loss and number of iterations for TensorFlow code:



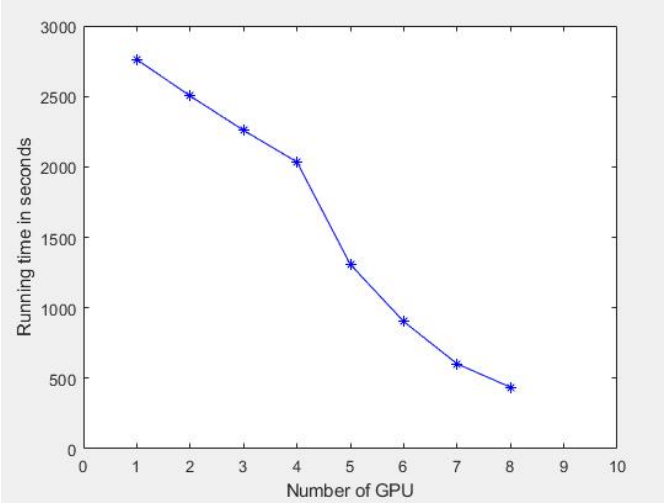
But in overall, TensorFlow version provides a decreasing in training loss and increasing in accuracy as well. From the observations, it is easily to say that Keras is much more stable than TensorFlow, the observation probably is one of the reasons to show why Keras is in a higher level than TensorFlow.

For the third group, by running the same LSTM Autoencoder model in TensorFlow, the distributed way takes about 80 seconds to finish one graph input, which is much slower than without distributed way. For the first 20 training iterations, the training loss decreases from 4.3668 to 1.9307, which stands for an increase of the accuracy; and for the last 20 training iterations, the training loss increases from 0.0185 to 0.0195, which stands for a small decrease of the accuracy. In overall, this version also provides a decreasing in training loss and increasing in accuracy, but compare to the normal way, the change in loss for distributed way is becoming smaller, and the model becomes more stable as well.

Then I set use different GPU to run the distributed version, the goal is checking how fast for using the different number of GPU in distributed way. The result is surprised, for only run one graph as input, the speed for the distributed way in using different number of GPU is shown as below:



As shown, the speed is going up when the number of GPU is increased. The reason should be the input sequence is too small, which cannot show the advantage of distributed way. So I use all graphs as input, then test it again. The goal is increasing the input and see if it can provide differnet result with the last observation. The reslut is showing like this:



As the picture shows, distributed way does can speed up the running time if there is a huge input.

5. Possible Open Issues

As we known, the training process should be a big problem. Both Keras and TensorFlow have their advantages and shortages, Keras has a great accuracy but take a very long processing time; TensorFlow has a short processing time but with an unstable model and lower accuracy. As my observations, even for TensorFlow, the distributed way also needs to train a long time for only one graph embedding input. Then if change to Keras, the processing time at least should be doubled or even tripled, or even much longer. So once we get one large scale dataset, which strategy should we choose will become an essential problem.

REFERENCES

[1] A. An, "EECS 4080 Computer Science Project Contract," Department of Electrical Engineering and Computer Science, York University, 2019, pp 1-2.

[2] A. Grover and J. Leskovec, "node2vec: Scalable feature learning for networks," in Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining. ACM, 2016

[3] A. Nain, "TensorFlow or Keras? Which one should I learn?" in Imploding Gradients, 2017, via https://medium.com/implodinggradients/tensorflow-or-keras-which-one-should-i-learn-5dd7fa3f9ca0

[4] J. Chen, et al. "E-LSTM-D: A Deep Learning Framework for Dynamic Network Link Prediction," arXiv preprint arXiv:1902.08329, 2019.

[5] J. Chen, et al. "GC-LSTM: Graph Convolution Embedded LSTM for Dynamic Link Prediction," arXiv preprint arXiv:1812.04206, 2018.

[6] P. Goyal, E. Ferrara, "Graph embedding techniques, applications and performance: a survey," in Knowledge based systems journal, 2018.

[7] P. Wang, et al. "Learning distributed word representations for bidirectional lstm recurrent neural network," Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies. 2016.

[8] Y. Zuo, etal. "Embedding temporal network via neighborhood formation," KDD, 2018

[9] TesnsorFlow video tutorial via https://www.youtube.com/watch?v=-h0cWBiQ8s8

[10] Tensorflow 2 Install (neural network tutorials) via https://www.youtube.com/watch?v=UOuQ8q01aPE&list=PLXO45tsB95cJHXaDKpbwr5fC_CCYylw1f&index=2

[11] Tensorflow 3 example 1 (neural network tutorials) via https://www.youtube.com/watch?v=8Lq_fmAsANI&list=PLXO45tsB95cJHXaDKpbwr5fC_CCYylw1f&index=3

[12] Tensorflow 4 tf coding structure (neural network tutorials) via https://www.youtube.com/watch?v=lqbJg2_cwv8&list=PLXO45tsB95cJHXaDKpbwr5fC_CCYylw1f&index=4

[13] Tensorflow 5 example2 (neural network tutorials) via https://www.youtube.com/watch?v=PFijwks2K6o&list=PLXO45tsB95cJHXaDKpbwr5fC_CCYylw1f&index=5

[14] Tensorflow 6 Session (neural network tutorials) via https://www.youtube.com/watch?v=hWmbAglaHxk&list=PLXO45tsB95cJHXaDKpbwr5fC_CCYylw1f&index=6

[15] Tensorflow 7 Variable (neural network tutorials) via https://www.youtube.com/watch?v=UYyqNH3r4lk&list=PLXO45tsB95cJHXaDKpbwr5fC_CCYylw1f&index=7

[16] Tensorflow 8 placeholder (neural network tutorials) via https://www.youtube.com/watch?v=Yl5lDaYvNqI&list=PLXO45tsB95cJHXaDKpbwr5fC_CCYylw1f&index=8

[17] Tensorflow 9 activation function (neural network tutorials) via https://www.youtube.com/watch?v=Kd7gDHY_OUU&list=PLXO45tsB95cJHXaDKpbwr5fC_CCYylw1f&index=9

[18] Tensorflow 10 example3 def add_layer() function (neural network tutorials) via https://www.youtube.com/watch?v=Vu_IIJ_Yexk&list=PLXO45tsB95cJHXaDKpbwr5fC_CCYylw1f&index=10

[19] Tensorflow 11 example3 build a network (neural network tutorials) via

https://www.youtube.com/watch?v=8XKSQTCzXEI&list=PLXO45tsB95cJHXa
DKpbwr5fC_CCYylw1f&index=11
[20] Tensorflow 13 Optimizers (neural network tutorials) via
https://www.youtube.com/watch?v=hXtEQavhrLk&list=PLXO45tsB95cJHXaDK
pbwr5fC_CCYylw1f&index=13
[21] Tensorflow 16 Classification (neural network tutorials) via
https://www.youtube.com/watch?v=AhC6r4cwtq0&list=PLXO45tsB95cJHXaDK
pbwr5fC_CCYylw1f&index=16
[22] Tensorflow 18 Saver (neural network tutorials) via
https://www.youtube.com/watch?v=e05zY-TJb5k&list=PLXO45tsB95cJHXaDK
pbwr5fC_CCYylw1f&index=18
[23] Tensorflow 20.2 RNN LSTM Classifer via
https://www.youtube.com/watch?v=SeffmcG42SY&list=PLXO45tsB95cJHXaD
Kpbwr5fC_CCYylw1f&index=20
[24] Existing node2vec code via https://github.com/aditya-grover/node2vec
[25] Github for parallel training in GPU via
https://github.com/petewarden/tensorflow_makefile/blob/master/tensorflow/mode
ls/image/cifar10/cifar10_multi_gpu_train.py
[26] Using GPU via https://www.tensorflow.org/guide/using_gpu
[27] ConfigProto via https://blog.csdn.net/dcrmg/article/details/79091941