香港中文大学（深圳）
The Chinese University of Hong Kong, Shenzhen

DDA3005

Numerical Methods

Instructor: Dr. Andre Milzarek

Course Project Singular Value Decomposition

Group Name: Macrohard

December 10, 2022

Group Members

ID	Name in Chinese	Name in English	Experiments Involved
120090549	温子雄	WEN Zixiong	1, 2, 3, 4, 5, 6
120090135	王子文	WANG Ziwen	1, 2, 3, 4, 5, 6
120090470	李鹏	LI Peng	1, 2, 3, 4, 5, 6
120090224	杨尚霖	YANG Shanglin	1, 2, 3, 4, 5, 6
120090771	邱纬纶	QIU Weilun	1, 2, 3, 4, 5, 6

Responsibilities and Contributions

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Contents

1 TeraSort based on HiBench

见于前

2 PageRank based on HiBench

见于前

3 Classification using NaiveBayes based on HiBench

见于前

4 Matrix Multiplication

4.1 Matrix Multiplication Algorithm

In the two Python code provided in the AIRS Cloud, one mapper and one reducer were implemented to compute the product of two matrices given by such formula:

$$C = (AB)_{ij} = \sum_{r=1}^{n} a_{ir} b_{rj}$$

For the purpose of distributed computing as a core feature of MapReduce, the computation process should be separated into independent procedures so that computation might be done on various nodes. By the formula above, we learn that c_{ij} are independent from each other, so that we can put them into the same key in the map phase. Then, in the reduce stage we can compute C by analyzing different elements.

4.2 Matrix Multiplication in MapReduce

MapReduce is a programming model developed by Google and made available as an Apache open source project . The model is used for processing large data sets across clusters of computers using a shared filesystem. Specifically, it is implemented in a fashion where user programs only need to ' Map' input data to values stored in a global distributed file system, and ' Reduce' those values to derive the final output . Because of its origins, its terms tend to reflect that origin with code that is written in Java and Python.

In our experiment, we first uploaded the input files to Hadoop's HDFS file system:

```
hadoop fs -put /home/team18/matrix/L1.txt /dataspace/team18/matrix/L1.txt
hadoop fs -put /home/team18/matrix/R1.txt /dataspace/team18/matrix/R1.txt
```

Then we gave writing and reading access to them:

```
hadoop fs -chmod 777 /dataspace/team18/matrix/
```

Once the input files are uploaded and accessible through the HDFS file system, we began using Hadoop's streaming utility:

```
mapred streaming -input /dataspace/team18/matrix \
    -file /home/team18/matrix/MatMulMapper.py \
    -mapper "python␣MatMulMapper.py" \
    -file /home/team18/matrix/MatMulReducer.py \
    -reducer "python␣MatMulReducer.py" \
    -output /dataspace/team18/matrix-output
```

Which yields the results as follows:

```
22/11/28 08:47:35 INFO mapreduce.Job: The url to track the job: http://master2.cuhk.com:8088/proxy/application_1669595859297_0007/
22/11/28 08:47:35 INFO mapreduce.Job: Running job: job_1669595859297_0007
22/11/28 08:47:42 INFO mapreduce.Job: Job job_1669595859297_0007 running in uber mode : false
22/11/28 08:47:42 INFO mapreduce.Job: map 0% reduce 0%
22/11/28 08:47:50 INFO mapreduce.Job: map 100% reduce 0%
22/11/28 08:48:02 INFO mapreduce.Job: map 100% reduce 100%
22/11/28 08:48:36 INFO mapreduce.Job: Job job_1669595859297_0007 completed successfully
```

```
22/11/28 08:48:36 INFO mapreduce.Job: Counters: 53
    File System Counters
        FILE: Number of bytes read=184842966
        FILE: Number of bytes written=370439648
        FILE: Number of read operations=0
        FILE: Number of large read operations=0
        FILE: Number of write operations=0
        HDFS: Number of bytes read=23083042
        HDFS: Number of bytes written=393911
        HDFS: Number of read operations=11
        HDFS: Number of large read operations=0
        HDFS: Number of write operations=2
    Job Counters
        Launched map tasks=2
        Launched reduce tasks=1
        Data-local map tasks=2
        Total time spent by all maps in occupied slots (ms)=34152
        Total time spent by all reduces in occupied slots (ms)=264864
        Total time spent by all map tasks (ms)=11384
        Total time spent by all reduce tasks (ms)=44144
        Total vcore-milliseconds taken by all map tasks=11384
        Total vcore-milliseconds taken by all reduce tasks=44144
        Total megabyte-milliseconds taken by all map tasks=34971648
        Total megabyte-milliseconds taken by all reduce tasks=271220736
    Map-Reduce Framework
        Map input records=2048
        Map output records=16384
        Map output bytes=184777424
        Map output materialized bytes=184842972
        Input split bytes=202
        Combine input records=0
        Combine output records=0
        Reduce input groups=72
        Reduce shuffle bytes=184842972
        Reduce input records=16384
        Reduce output records=16448
```

```
                Spilled Records=32768
                Shuffled Maps =2
                Failed Shuffles=0
                Merged Map outputs=2
                GC time elapsed (ms)=759
                CPU time spent (ms)=51570
                Physical memory (bytes) snapshot=2747916288
                Virtual memory (bytes) snapshot=16727359488
                Total committed heap usage (bytes)=2665480192
                Peak Map Physical memory (bytes)=2122207232
                Peak Map Virtual memory (bytes)=4675809280
                Peak Reduce Physical memory (bytes)=1531682816
                Peak Reduce Virtual memory (bytes)=8512802816
            Shuffle Errors
                BAD_ID=0
                CONNECTION=0
                IO_ERROR=0
                WRONG_LENGTH=0
                WRONG_MAP=0
                WRONG_REDUCE=0
            File Input Format Counters
                Bytes Read=23082840
            File Output Format Counters
                Bytes Written=393911
        22/11/28 08:48:36 INFO streaming.StreamJob: Output directory:
            /dataspace/team18/matrix-output-8
```

After the MapReduce is successfully done on the remote machine, we then run the clean up command to delete the generated output directory after copying the result to local file system.

To experiment further, we used the pipe operator to run the mapper and reducer separately locally to make sure the functions work correctly:

```
cat L1.txt, R1.txt | python3 MatMulMapper.py | python3 MatMulReducer.py
```

Now in the application of the MapReduce algorithm to the matrix multiplication problem, we studied the following aspects.

First is the data or file structure. Matrix data is stored in binary and separated by lines. The former led to the use of the ‘binascii‘ module and the latter is to make separation of concerns easier done.

Second is the computation process. To convert the algorithm into MapReduce, we have to implement three phases: map, shuffle and reduce.

In the map phase, we marke a_{ij} to ‘<key, value>‘ of number I, where ‘key‘ $= (i, k), k = 1, 2, ...I$ and ‘value‘ $= (a, j, a_{ij})$. The same goes for the B matrix. The key bridges the computation results, and the value separates numbers from different matrices.

```
    if A_B == "L":
    ib = (int)(lineno)/BLOCKSIZE # note here the input data starts from 1, the result may differ from that in ppt
    for jb in range(NB):
        # the key is the BLOCK Number
        intermediate_key = '%05d'%(ib * NB + jb)
        # the value is the {L/R}:{LineNo}:{values of current line}
        intermediate_value = 'L:%s:%s'%(lineno, row_value)
        # key and value are seperated by a tab
        print("%s\t%s" % (intermediate_key, intermediate_value))

if A_B == "R":
    jb = (int)(lineno)/BLOCKSIZE
    for ib in range(NB):
        intermediate_key = '%05d'%(ib * NB + jb)
        intermediate_value = 'R:%s:%s'%(lineno, row_value)
        print("%s\t%s"%(intermediate_key, intermediate_value))
```

In the shuffle phase, values with the same key will be packed into a list and passed to reduce. This is automatically done by Hadoop.

In the reduce phase, we have constructed the key as a form in the Map stage. And we also marked in the Map phase. The next thing to do is to parse the list(value), the elements from are placed in an array alone, and the elements from are placed in another array. Then, we calculate two arrays (each as a vector), the value that can be calculated.

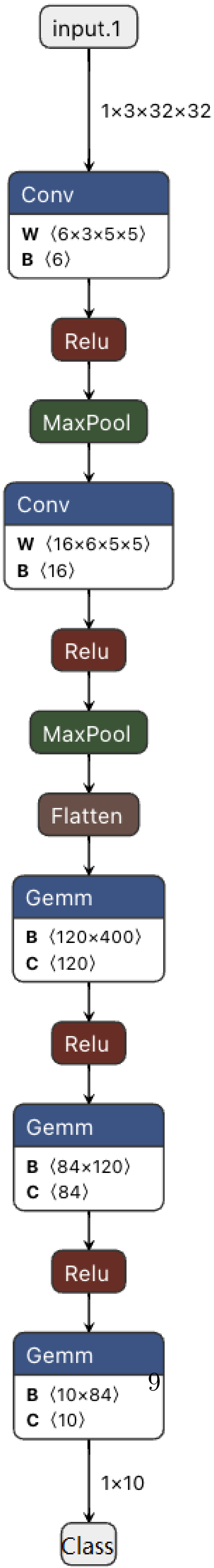
```
    blockno = int(input_key)
    A_B, index, row_value = input_value.split(":")

    if A_B == "L":
        LeftMatrixBlock.append(row_value.split(" "))
    if A_B == "R":
        RightTransposeMatrixBlock.append(row_value.split(" "))

res = [[0 for col in range(BLOCKSZIE)] for row in range(BLOCKSZIE)]
for i in range(BLOCKSZIE):
    for j in range(BLOCKSZIE):
        for k in range(TOTALSIZE):
            left_val = int(struct.unpack("I", binascii.a2b_hex(LeftMatrixBlock[
                i][k][2:]))[0])
            right_val = int(struct.unpack("I", binascii.a2b_hex(
                RightTransposeMatrixBlock[j][k][2:]))[0])
            res[i][j] += left_val * right_val
        print(res[i][j])
```

5 Image Classification

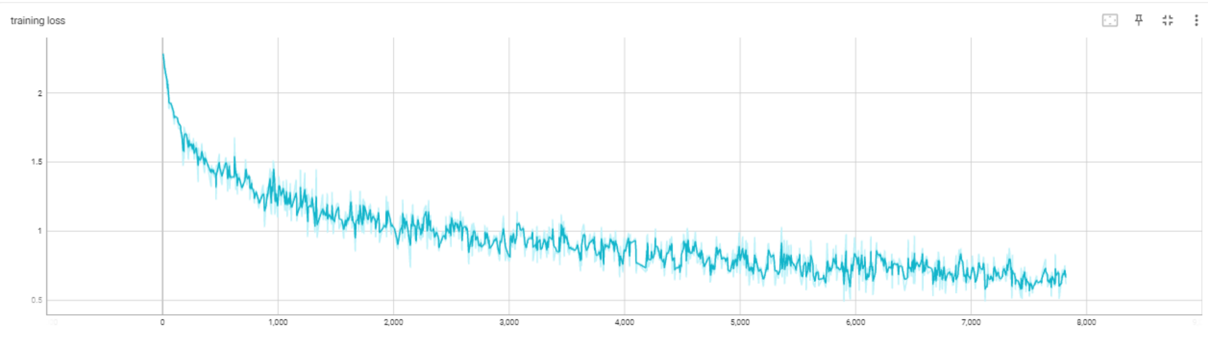
5.1 The structure of network structure



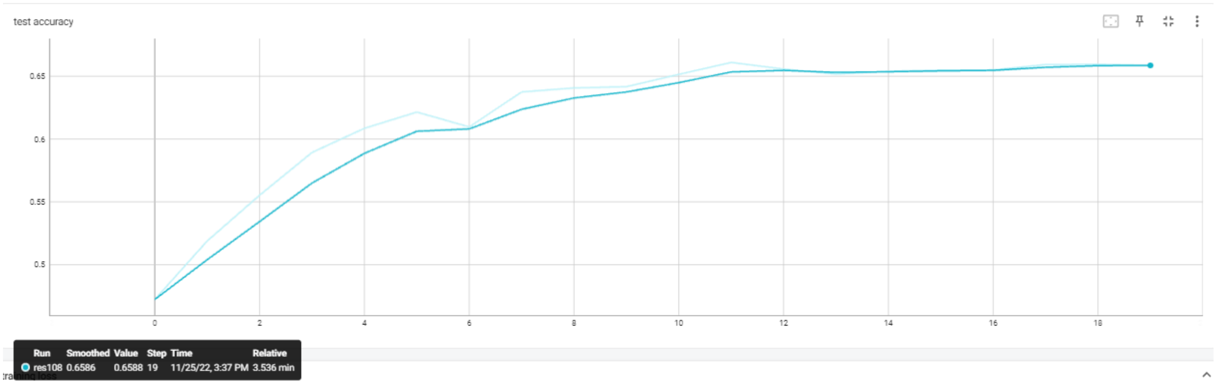
In this report, we use CNN to classify the images, where Conv get the feature of the input and Relu to activate and then use pool to up sampling. This CNN is Feedforward neural network.

5.2 Result with default settings

The training loss with default settings are below.



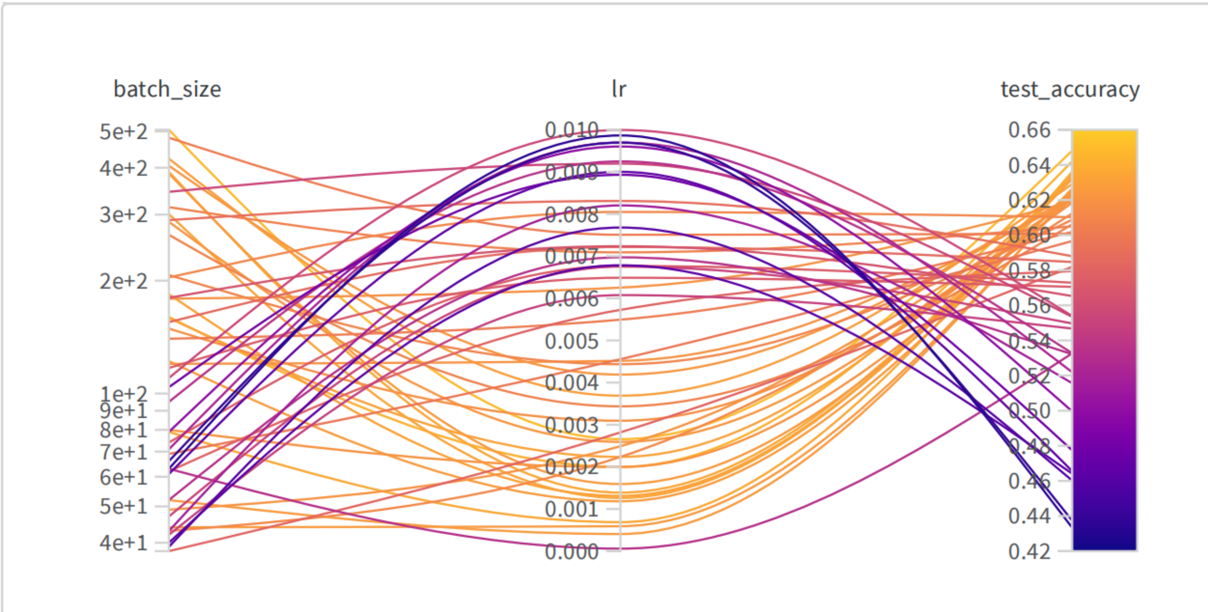
The accuracy with default settings are below.(where X-axis is time)



With the training, the accuracy trend to be 66%. training time are 3.536mins.

5.3 Result with different settings

In this section, we use different settings to obtain the task, and the result are below.



Through the picture, we can see the best accuracy is obtained with batch size = 4e+2 and the learning rate =0.002.

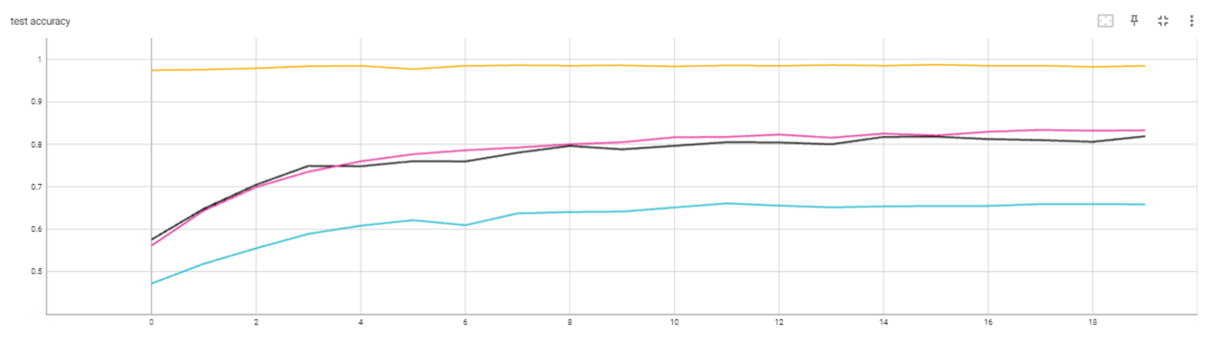
However, with other model, we get a higher acc.

The baseline is 63% (blue line),using 3 mins.

When we use resnet108, we can get an acc about 81%(black line), using 30 mins.

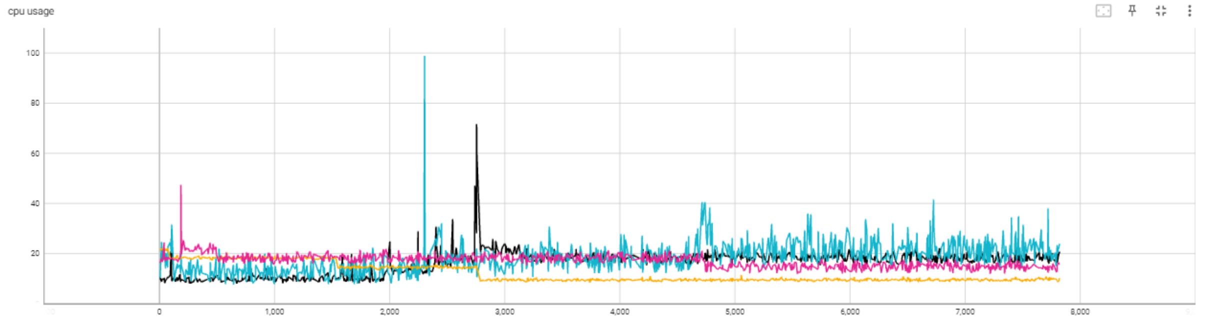
With Vit_small, we get an acc about 82%(purple line), using 36mins.

With swinv2, we get an acc about 98%(yellow line), using1.21 hrs.

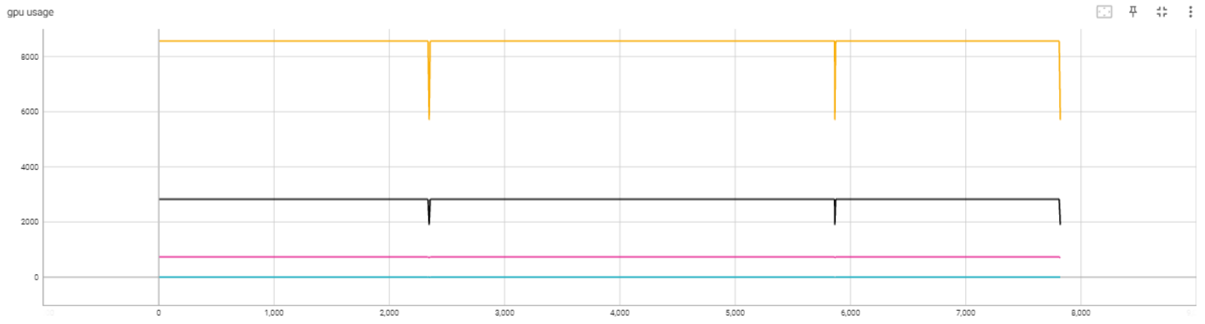


5.4 Recording of system run time information

The CPU usage is below.



The GPU usage is below.



The baseline is blue line, the resnet108 is black line, Vit_small is purple line and the swinv2 is yellow line.

Through the usage of system, we can see the usage of CNN provided is low at first. Then it increases and waves frequently. It may be the simple structure of the network and the thread of CPU. And the usage of GPU is constant, which is mainly because the parallel computing of GPU is high.

6 Image to Text

6.1 Experiment Specification

The experiment background is a image captioning task on the dataset COCO2014 (Microsoft Common Objects in Context) invloving Computer Vision and Natural Language Processing. Rather than performing large-scale object detection, segmentation, key-point detection, and captioning, which COCO2014 is commonly adopts, our model only performs the captioning, i.e, for an input image , the model outputs the caption of objects in the image.
Our experiment is to train the model with different hyperparameters and compare the model performance under different metrics. We are tuning the following hyperparameters: $batch\_size$
Encoder (Convolutional Neural Network): $embed\_size$
Decoder (Recurrent Neural Network): $embed\_size$, num_layers, $hidden\_size$
Optimizer (Adam): $learning\_rate$

6.2 Interpretation of Perplexity

Perplexity is a common performance metric in NLP. Concisely, Perplexity is the exponential of Cross-Entropy. For a given sentence $W = (w_1, w_2, \cdots, w_n)$, where w_i denotes the ith word, its Perplexity is defined as:

$$Perplexity(W) = 2^{H(W)} \tag{1}$$

where $H(W)$ denotes the Cross-Entropy of W, noted that the base is not necessarily be 2, in our experiment, we use e. Entropy denotes the expectation of information, after simplifying we get

$$H(W) = -\log P(w_1, w_2, \cdots, w_n) \tag{2}$$

In the view of bit-length, Perplexity can be interpreted as the number of outputs that are of the same probability (that's why we commonly use 2 as the base in Eq.1). For a given historical information, the less equal probability outputs, the less 'confused' the model is and hence the better modeling.

6.3 Analysis of Different Hyperparameters

Notice that while we try different hyperparameters, we do it in a control experiment manner and keep other hyperparameters of their default setting:

Table 1: Default Setting

embed_size	*num_layers*	*batch_size*	*hidden_size*	*learning_rate*
299	1	128	512	0.001

6.3.1 embed_size

embed_size is a hyperparameter shared by both encoder and decoder. Generally, large *embed_size* will not hurt the model performance but results in higher training cost. We tried *embed_size* $= \{50, 100, 200, 300, 400\}$, the performance and time cost for each *embed_size*:

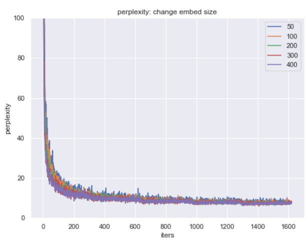


Figure 1: Perplexity

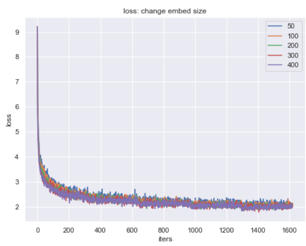


Figure 2: Loss

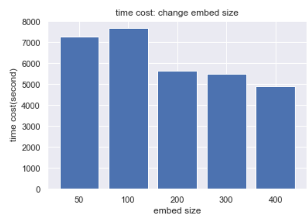


Figure 3: Time cost

Hence we can see that *embed_size* $= 400$ achieves the better accuracy and convengence speed.

6.3.2 learning rate

learning_rate is a hyperparameter in the Optimizer Adam. We tried *learning_rate* $= \{0.1, 0.01, 0.001\}$, the performance and time cost for each *learning_rate* :

Figure 4: Perplexity

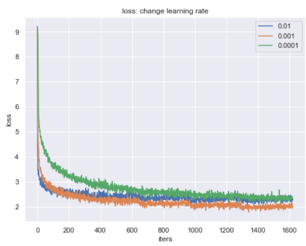


Figure 5: Loss

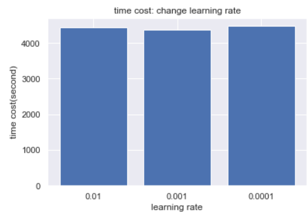


Figure 6: Time cost

With no significant difference in time costing, a bolder choice of $learning\_rate = 0.01$ received the better Perplexity.

6.3.3 number of layers

num_layer is a hyperparameter in Decoder(LSTM). We tried $num\_layers = \{1, 2, 4\}$, the performance and time cost for each num_layer:
:

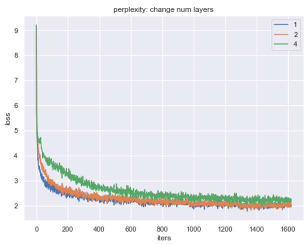


Figure 7: Perplexity

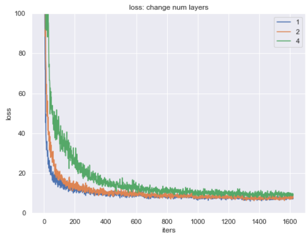


Figure 8: Loss

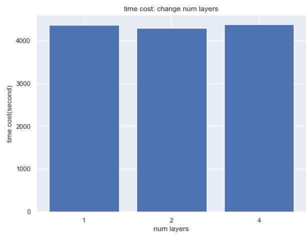


Figure 9: Time cost

With no significant difference in time costing, decreasing LSTM complexity $num\_layers = 1$ received the better Perplexity and convengence speed.

6.3.4 hidden size

$hidden\_size$ is a hyperparameter in Decoder(LSTM). We tried $hidden\_size = \{128, 256, 512\}$, the performance and time cost for each $hidden\_size$:

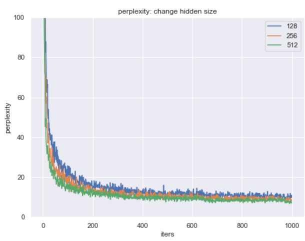


Figure 10: Perplexity

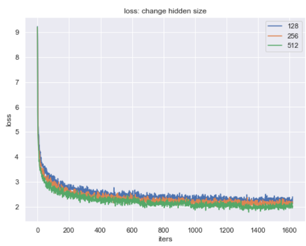


Figure 11: Loss

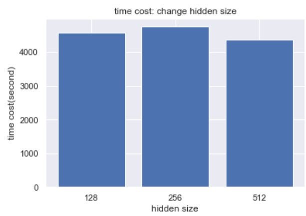


Figure 12: Time cost

With slightly lower time costing, increasing LSTM complexity $hidden\_sizes = 512$ received the better Perplexity and convengence speed.

6.3.5 batch size

batch_size is a hyperparameter in training. Dividing data into batches can decrease training memory usage and by updating parameters after every batch, the algorithm converges faster. The performance and time cost for each *hidden_size*:

Figure 13: Perplexity

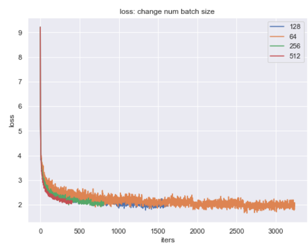


Figure 14: Loss

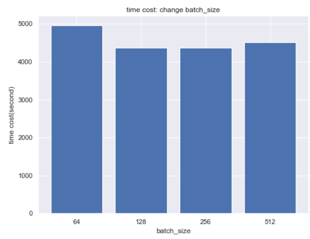


Figure 15: Time cost

Dividing the training set into batches of 128 or 256 or 512 increases the training speed while $batch\_size = 512$ converges faster than others.

6.3.6 GPU and CPU usage analysis

With the increase of *batch_size*, the use rate of CPU and GPU will both increase. That's because we need to load more data into CPU and GPU in one batch.
With the increase of *learning_rate*, the use rate of CPU and GPU will stay almost the same. That's because the learning rate have no influnce on the memory.
With the increase of *hidden_size*, the use rate of CPU stays the same. but GPU will increase because the we need to load more parameters to GPU.
With the increase of *embed_size* the use rate of CPU and GPU will both increase. That's because the smaller the embed sizethe smallerthe size ofinput data, which also means smaller usage of CPU and GPU.