E:\1Polyu\year4sem1\LE3\polyu-logo.pngE:\1Polyu\year4sem1\LE3\polyu-comp-logo.png

**COMP4911/4912 Capstone Project Final Report (2015/16)**

**From Neural Network**

**to Deep Learning**

**on Spark**

**Supervisor** : Dr. LO Chi Lik Eric

**Co-examiner** : Dr SHIU Chi Keung Simon

**2nd Assessor** : Prof. BACIU George, Dr. LIU Yan

**Student Name** : GENG Xu

**Student ID** : 12132031d

**Program Title** : BSc (HONS) INFORMATION TECHNOLOGY

**Program Code** : 61431-FIT

**Date** : 2016/4/11

**Abstract**

Deep learning is an attractive approach in machine learning research community. It has achieved empirical success in applications like computer vision and natural language processing. To solve the high computational complexity problem of training a deep neural network (DNN), it’s quite common that researchers train a DNN architecture using the computational power of GPU. With the emergence of cloud computing, typically those use the computational power from a cluster of nodes, it shows some advantages over GPU computing and makes it another good solution for accelerating scientific computing.

Spark is a lightning fast cluster computing framework that grows fast in recent years. It has complete underlayer support on message concurrency, resilience and higher level scientific computation (linear algebra library). Spark is an ideal framework of implementing generic cluster computing applications.

This project implements basic layer structures and provide convenient interfaces to build up convolutional neural networks (CNN) on Spark. With deployment experiment on EC2, it achieved good performance in accuracy (as a classifier) and speed-up ratio (as a parallel algorithm). Related work please refer to: <https://github.com/PhilipGeng/SparkNNDL>

Keywords and Nomenclature

CNN: Convolutional neural network

FP/BP: forward propagation/ back propagation

Gradient-based learning

MNIST: handwritten digit database

**Acknowledgements**

I would like to express my deepest gratitude to my supervisor Dr. Eric LO who generously offered me the chance to carry out the project and provided insightful advice and help. More importantly, Eric guided me into big data and machine learning by his course COMP4434 big data analytics.

I would also show my appreciation to Prof. George BACIU, Dr. Yan LIU, Dr. Korris CHUNG, Prof. David ZHANG for their course in cloud computing, AI, data mining and biometrics which drove the project forward.

There are friends along with me to share their ideas or advice in mathematics and computer science which enables me to build the entire project from basic theories. They are QIU Duo(AMA, Tsinghua Univ.), TAN Yukun(AMA&CS, UIUC), MAO Chunxuan(AMA, SDNU), ZHANG Yifei(AMA, SDNU), YANG Jianzhi(AMA, HKPU), ZHANG Beibei(COMP, HKPU).

I really appreciate the chance to study in the Department of Computing of Hong Kong Polytechnic University, who always offer altruistic help and sufficient resource. I should show my appreciation to all staff whoever provided academic help and advice to me, especially Dr. Yuanqing ZHENG and Dr. Vincent T.Y.NG who are the most influential among them.

Last but not the least, I would like to thank my parents and all family members for their love, support and lenience.

**Table of contents**

Chapter 1. Introduction 9

1.1 Artificial Neural Network (ANN) 9

1.2 Network training 10

1.3 Convolutional neural network (CNN) 11

1.3.1 Forward 12

1.3.2 Back Propagation 13

1.3.3 Gradient 14

1.3.4 Utility Functions 15

1.4 Spark 16

Chapter 2. Literature review 17

2.1 Textbooks and MOOCs 17

2.2 Spark-scala 17

2.3 Caffe and theano 18

2.4 LeNet5 and MNIST 20

Chapter 3. Methodology 22

3.1 project scope 22

3.2 project architecture 23

3.2.1 UML class diagram 23

3.2.2 layer model 24

3.3 Network Training Algorithm 25

3.3.1 Local training 25

3.3.2 Cluster training 26

3.4 Network Classification Algorithm 29

3.5 layer components detail implementations 30

3.5.1 layer.scala 30

3.5.2 CL.scala 33

3.5.3 SL.scala 34

3.5.4 FL.scala 35

3.5.5 OL.scala 36

3.6 Correctness checking 37

3.6.1 Correctness of local training 37

3.6.2 Correctness of cluster training 37

Chapter 4. Deployment and experiment 38

4.1. MNIST pre-processing 38

4.2. Experiment on MLP 38

4.2.1 Architecture 38

4.2.2 Deployment 39

4.2.3 Accuracy 39

4.3. Experiment on LeNet5-like CNN architecture 39

4.3.1 Architecture 39

4.3.2 Training parameter 41

4.3.3 EC2 deployment 41

4.3.4 Accuracy 42

4.3.5 Time consumption and speed up 43

4.4. Analysis on the experiment 44

4.4.1 Conclusions from the experiment 44

4.4.2 Potential improvements 44

Chapter 5. Project schedule 46

Chapter 6. Conclusions 47

**List of tables**

t1 Caffe: basic features and functions 18

t2 LeNet5 training parameters by Yann 20

t3 LeNet experiment result by Yann 20

t4 CNN features provided by this project 22

t5 MLP experiment architecture 38

t6 MLP experiment deployment 38

t7 number of trainable parameters in LeNet5 40

t8 EC2 deployment 41

t9 CNN experiment accuracy 42

t10 CNN experiment time consumption 43

t11 Spark running configuration 44

t12 Project schedule 46

**List of figure**

pic1 Feedforward neural network 9

pic2 Locally connected and convolutional neural network 11

pic3 Typical CNN structure: Deep ID 12

pic4 LeNet for theano experiment 19

pic5 Result of Theano experiment on different batch size 19

pic6 Lenet5 20

pic7 LeNet experiment result by Yann 21

pic8 Project UML class diagram 23

pic9 Project layer model 24

pic10 Cluster train without cache() 27

pic11 Cluster train with cache() 28

pic12 Time consumption at each action operation 28

pic13 Lenet5 S2 to C3 input map 40

pic14 Spark UI on CNN training program 42

pic15 CNN experiment accuracy comparison 43

**List of mathematical proof**

f1 activate function 9

f2 squared error loss function 10

f3 output layer error 10

f4 hidden layer error 10

f5 gradient 11

f6 gradient descent update 11

f7 convolution forward 12

f8 subsampling forward 12

f9 Fully-connected forward

f10 output error 13

f11 FL errbp 13

f12 sl->cl errbp 14

f13 cl->sl errbp 14

f14 CL gradient 14

f15 SL gradient 14

f16 FL gradient 14

f17 matrix expansion 15

f18 matrix padding 15

f19 matrix rotation 15

f20 matrix convolution 15

f21 maximum likelihood for label classification 29

f22 stochastic gradient check 37

f23 batch gradient check 1 37

f24 batch gradient check 2 37

**List of code segments**

c1 Network local train 25

c2 Network local forward 25

c3 Network local calErr 25

c4 Network local update 25

c5 Network cluster train 26

c6 Network cache output 26

c7 Network cache Err 26

c8 Network clear cache 26

c9 Network local classify 29

c10 Network cluster classify 29

c11 package information 30

c12 layer.scala declaration 30

c13 layer.scala variables 30

c14 layer.scala function abstraction 31

c15 layer.scala utility functions 32

c16 CL.scala declaration 33

c17 CL.scala variables 33

c18 CL.scala input mapping functions 33

c19 CL.scala networking functionality 34

c20 SL.scala declaration 34

c21 SL.scala variables 34

c22 SL.scala networking functionality 35

c23 FL.scala declaration 35

c24 FL.scala variables 35

c25 FL.scala networking functionality 36

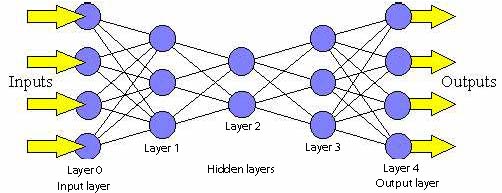
c26 OL.scala declaration 36

c27 OL.scala networking functionality 36

**Chapter 1. Introduction**

1.1 Artificial Neural Network (ANN)

ANN is a family of models inspired by biological neural networks. Feedforward neural network is a typical kind of neural network made up of activation units been connected by weighted, acyclic connections. Usually layers can be categorized into input layer, hidden layers and output layer.



(pic Feedforward neural network)

For each unit (neuron), it sums the output value of all or part of units from the previous layer according to weight and output an activated value. The activation function implemented in this project involves sigmoid function (sigmoid), hyperbolic tangent (tanh) and rectified linear unit (ReLU), where:

(f1: activate function)

1.2 Network training

ANN is usually used to approximate function between input and output and commonly works as a classifier. Gradient descent back propagation is a way of training such a network classifier.

Loss function J(θ) is the function to measure the error of the classifier. The goal of NN training is to minimize J(θ). In this project, the loss function being implemented is squared error loss function:

(f2: squared error loss function)

Back propagation is an algorithm to propagate error(also called delta, or sensitivity) from the output layer to prior network layers.

For output layer, error is derived from loss function

(f3: output layer error)

For hidden layer, error is derived from posterior layer

(f4: hidden layer error)

Gradient descent is an algorithm to generate weight updates in order to minimize the loss function J(θ)

Gradient:

(f5: gradient)

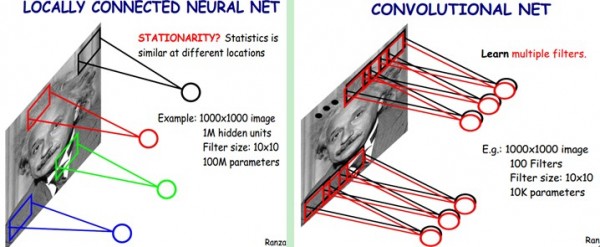
Update:

m:momentum

(f6: gradient descent update)

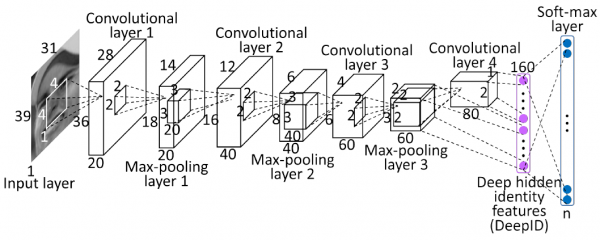
1.3. Convolutional Neural Network (CNN)

In image processing, different convolutional musk are used for extracting different features by applying overlapping receptive fields (shown in pic2). Subsampling is employed to increase the position invariance and spatial abstractness for higher feature abstractness.



(pic2. locally connected and convolutional neural network)

The figure in pic3 shows a typical CNN structure called Deep ID that developed by Dr. SUN Yi from CUHK, which achieve accuracy of 99% in face verification. CNN uses alternating convolutional layer and subsampling layer for feature abstraction before fully-connected (inner product) layer. All the weights, including convolutional kernels are randomly initialized and learnable.



(pic3. Typical CNN structure: Deep ID)

Like multi-layer perceptron, CNN follows similar training procedure:

1.3.1 Forward:

1.3.1.1 Convolutional layer(CL):

A CL may consist of multiple output feature maps (fm). Each feature map is derived from a series of input map by summing up the convolution result between each input map (im) and it’s own convolutional kernel. Each kernel is only shared through the convolutional operation for one input map.

(f7 convolution forward conv(): convolve, refer to f20)

After convolutional operation, the size of the image is slightly changed.

1.3.1.2 Subsampling layer(SL):

Each feature map in SL is one-to-one mapped and derived from a feature map from previous layer by carrying out sampling operation in a non-overlapping square neighborhood. Max sampling, min sampling and mean sampling are all frequently used subsampling operations.

(f8 subsampling forward)

After subsampling,, the size of feature maps are reduced by the dimension of the neighborhood on each dimension.

1.3.1.3 Fully-connected layer(FL):

The fully-connected layer is also called inner product layer. It derives output by multiplying input one-dimensional vector with two-dimensional weight matrix

(f9 Fully-connected forward)

1.3.2 Back Propagation:

Remark: All are vectorized implementation, :\* stands for element-wise multiplication

1.3.2.1 FL as Output layer(OL):

(f10 output error)

1.3.2.2 FLk🡪FLk-1:

(f11 fl->fl errbp)

1.3.2.3 FLk🡪CLk-1: each output fm in CL must be 1x1 in order to fit in the input size of next FL

(f11 fl->cl errbp)

1.3.2.4 FLk🡪SLk-1: each output fm in SL must be 1x1 in order to fit in the input size of next FL

(f11 fl->sl errbp)

1.3.2.5 SLk🡪CLk-1:

: SL kernel

(f12: sl->cl errbp expand(): refer to f17 )

1.3.26 CLk🡪SLk-1:

,

(f13 cl->sl errbp

conv(): convolve, refer to f20,

pad(): matrix padding, refer to f18,

rot180: matrix rotation, refer to f19 )

1.3.3 Gradient:

1.3.3.1 CL:

(f14: CL gradient)

1.3.3.2 SL:

(f15: SL gradient)

1.3.3.3 FL:

(f16: FL gradient)

1.3.4 Utility functions (implementations: section 3.5.1 layers-utility functions):

1.3.4.1 Matrix size expansion: expand(mat, dim)

If

(f17: matrix expansion)

1.3.4.2 Matrix padding: pad(mat,dim), surround mat with 0s of width dim.

If

(f18: matrix padding)

1.3.4.3 Matrix rotate 180 degree: rot180(mat)

If

(f19: matrix rotation)

1.3.4.4 Convolve conv(mat, kernel) = C

(f20: matrix convolution)

1.4 Spark

Spark is a parallel computing framework designed by UC Berkeley AMP lab. Spark grows up rapidly in recent years and has become more and more popular in the research and industrial community.

1.4.1 Spark computation model -- RDD:

Spark provides resilient distributed dataset (RDD) for data parallelism. Each RDD will be partitioned and distributed to different computational nodes for operation. There are two types of operation for RDD:

Transformation operation: transformations are operations on RDDs that return a new RDD.Some transformations are element-wise so that they don’t need to reconstruct data partitions (e.g. map()). Some transformations need to shuffle data partitions and repartition to reconstruct new RDD partition (e.g. groupBy()). Transformations are lazy. They are evaluated only when the program executes an action operation.

Action operation: action operation will collect all partitions of RDD to local program (e.g. foreach, saveAsTextFile, reduce).

1.4.2 Spark features:

RAM based: Spark is RAM based operation on RDDs, Cache() or Persist() function can be used to cache data in RAM or ROM. Lazy evaluation on transformations ensures that Spark won’t exhaust RAM too fast, which keeps Spark running fast.

Lineage and checkpoint: Spark will check the data integrity at checkpoints. If data partitions are missing, Spark will reconstruct that partition according to lineage.

**Chapter 2. Literature Review**

2.1 Textbooks and MOOCs

There is a list of textbooks and MOOCs(massive online open course) that help me to fully understand CNN and build mathematical derivations.

Han, Mitchell and Andrew Ng gave introductions for simple multi-layer perceptrons in their textbooks[8][12] , including topology, feedforward, back propagation, learning parameters and detailed calculation. Besides, Andrew Ng illustrated some basic machine learning skills, for example, gradient descent and gradient checking in his online course[5]. UFLDL and VISTALab in Stanford provide tutorials on Neural network and CNN, including convolution, pooling, stochastic gradient descent(SGD)[1][14]. Geoffrey bring the idea of batch training on CNN in his online course[6]. All fundamentals above help me to build mathematical descriptions on CNN as was shown in section 1.1-1.3.

2.2 Spark-scala

Spark-scala is the framework I used throughout this project to achieve the parallelism of CNNs.

Spark-scala is chosen mainly because of his RAM-based feature which enables fast speed. Besides, Spark-scala inherit the functional programming features of scala which are very suitable for scientific computational programming. Spark-scala has a strong linear algebra library (breeze.linalg) to support its massive linear algebra calculations. With these features, Spark provide a set of good programming interface to achieve parallel computations. Also, programming interfaces are provided to manipulate distributed data partitions, for example, cache(), broadcast() and repartition() are provided functions for programmers to have underlayer control on data partitions. This project used the cache() function to solve the problem of lazy evaluation (section 1.4) and accelerate training procedure.

Spark mllib is a fast growing up library in recent months. Most of the data mining and machine learning algorithms that are considered to be implemented in the project proposal were done before the beginning of implementation, except neural networks. By the end of 2015, spark updated the implementation of multi-layer perceptron classifier(fully-connected, inner product layers). The MLP classifier was using batch update method to achieve the parallelism of network training, which is identical to my proposed idea of implementing CNN.

Spark-based cloud computing has some advantages over GPU computing. First of all, cloud computing is more scalable, portable and convenient to deploy, which is enabled by virtualization techniques. Compared with CPU, GPU virtualization is more complicated. Secondly, it’s much cheaper and easier to set up a cluster in cloud platforms than establishing a GPU cluster.

2.3 Caffe and theano

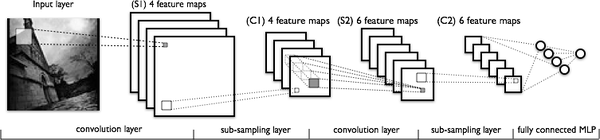
Caffe is a deep learning framework proposed by BVLC (Berkeley Vision and Learning Center) from UC Berkeley. Caffe provide deep learning implementations on GPU and well encapsulate layer components by configuration scripts. This project tried to implement as much as what Caffe done about CNN as shown in table.

|  |  |
| --- | --- |
| feature |  |
| CL | Rand weight, bias, input map |
| SL | Max, min, mean |
| FL | Rand weight, bias |
| Loss function | cross entropy loss/softmax loss/euclidean loss/multinnomial logistic loss, loss weights |
| activation | Sigmoid, tanh, relu, BNLL |
| Weight update | Base learning rate(lr), lr decay, momentum |
| configuration | Batch size, num of iteration |
| solver | Gradient descent, adadelta, adagrad … |

(t1 Caffe: basic features and functions)

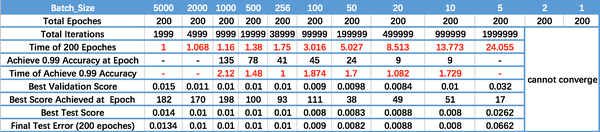
Theano is a python library for fast linear algebra calculation on GPU. Quite a lot of people nowadays are using it to implement some deep neural networks. It provides a set of programming interfaces to do that. There is a set of experiment reports convince me of the feasibility of batch update[4].

The network architecture is one of LeNet models:



(pic4. LeNet for theano experiment)

The experiment result on different batch size shows:



(pic5. Result of Theano experiment on different batch size)

The result shows:

1. For little batch size, the model does not converge.

2. When batch size increase, the processing speed (per epoch) get faster and faster (train CNN on Spark should also have the same feature, because large batch size saves communication tradeoff, which is a significant cost)

3. When batch size increase, efficiency descreases, number of epochs required to achieve same accuracy will increase.

Few conclusions are therefore been drawn:

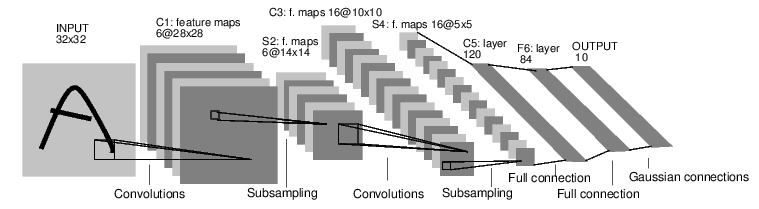
1. Batch update is a feasible measure to train neural network

2. Batch size needs to be well tuned to achieve the best performance in time and accuracy.

Additionally, for all different training experiment, the model will converge at different local optimas, which is also a very important point to address.

2.4 Lenet 5 and MNIST

LeNet5 is a renowned CNN architecture proposed by Yann Lecun in 1998 for document recognition. It’s composed of seven layers with topology shown below[10]:



(pic6. Lenet5)

This project implements a slightly modified LeNet5 as a test case for the layer components been implemented.

MNIST is a standard handwritten digit recognition dataset with 60000 training samples and 10000 testing samples.

Yann carry out the experiment using LeNet5 and MNIST database with parameters:

Activate function: tanh

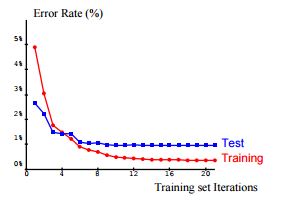
|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Num of iteration | 1,2 | 3,4,5 | 6,7,8 | 8,9,10,11 | 12-20 |
| Learning rate | 0.0005 | 0.0002 | 0.0001 | 0.00005 | 0.00001 |

(t2 LeNet5 training parameters by Yann)

Error (roughly, from pic7)

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| iteration | 1 | 2 | 3 | 4 | 5 | 6 | 7 | converge |
| train | 4.9% | 3.1% | 1.8% | 1.5% | 1.2% | 0.9% | 0.8% | 0.3% |
| test | 2.7% | 2.2% | 1.4% | 1.3% | 1.3% | 1.1% | 1.1% | 0.9% |

(t3 LeNet experiment result by Yann)



(pic7 LeNet experiment result by Yann)

**Chapter 3. Methodology**

3.1 project scope

This project aims at implementing CNN layer components on Spark. It started from fully-connected multi-layer perceptron model, and further enhanced to become CNN model.

There are several training methodologies implemented:

1. Stochastic gradient descent(SGD): train the network model use one sample each time.

2.mini-batch gradient descent: train the network model uses specific batch size of samples.

3. Update only when training forward result is wrong: This method will ignore those samples if they are already correctly classified during the forward procedure of training. This method is not implemented in the experiment due to the time limit and lack of existing examples.

This project provides convenient API for programmers to construct their own network models. The details about features been provided are listed in t4 below. Programmers can serialize CNN object for programmer to save trained model for future use. Also, demo programs, such as well structured LeNet5, local/cluster training and testing cases are provided.

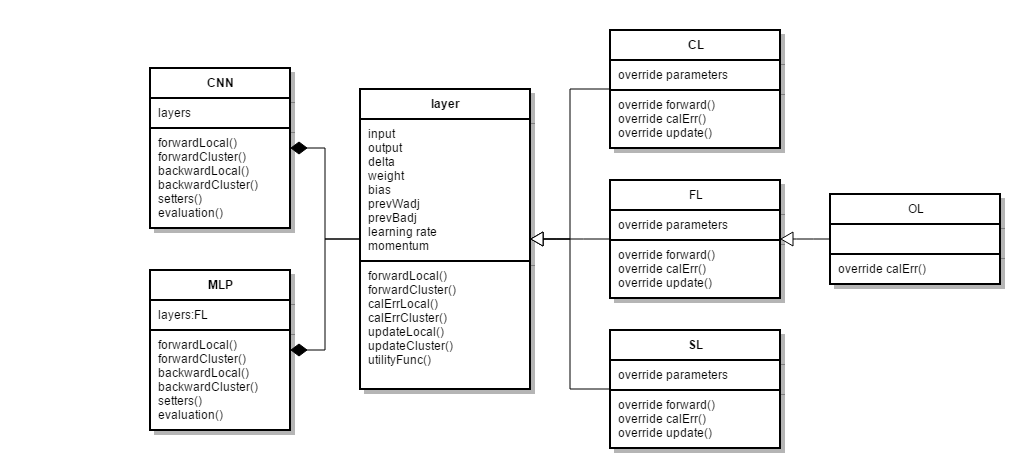
|  |  |
| --- | --- |
| feature | This project |
| CL | Rand weight, bias, input map |
| SL | Mean sampling |
| FL | Rand weight, bias |
| Loss function | Square error loss |
| activation | Sigmoid, tanh, relu |
| Weight update | Learning rate and momentum for each iteration |
| configuration | Batch size, num of iteration |
| solver | Gradient descent |

(t4 CNN features provided by this project)

Finally experiment on LeNet5 is carried out to test accuracy and speed-up ratio on time efficiency of cluster based training method.

3.2 project architecture

3.2.1 UML class diagram



(pic8 Project UML class diagram)

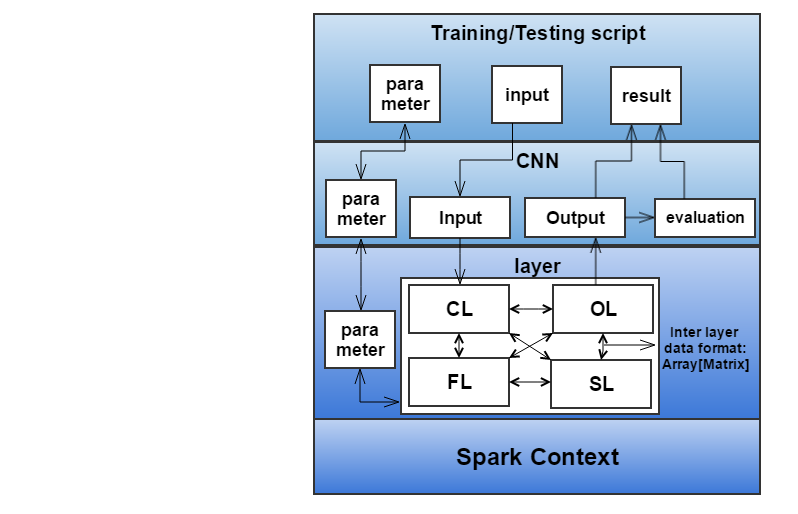
The project is mainly composed of network models and layer models.

There are four kinds of layers in CNN including convolutional layer(CL) ,subsampling layer(pooling layer, SL), fully connected layer(FL) and output layer(OL). OL is also a kind of FL but with ability to calculate loss of the model, so OL will have different implementation of error calculation, which makes it a subclass of FL.

The class layer is the common ancestor of CL, FL and SL which regulates parameter list (including data for calculation, network parameters and training parameters) and function list (forward, error calculation and update in local and cluster mode). Also the layer class implements some utility functions as specified in f16-f19 for descendants to use.

The detailed implementations of parameters and functions are indicated in each class with different algorithms. For calculate Error function(calErr) and update function(update()), each layer implement algorithms with considering possible posterior type of layer components.

3.2.2 layer model



(pic9 Project layer model)

The layer model shown above in pic9 is also the runtime object model of this project.

Layer components are basic computational modules for the training/classification of the network. They can be inter-connected via well-implemented interface. The data communication format among them is Array[DenseMatrix[Double]] (array of densematrices of double). The layer components do not need programmers’ modification

The CNN object will instantiate the layer component topology, provide parameters/input data and finally fetch result from OL and evaluate result. The CNN object can be modified by programmers.

The training/testing script are capable to instantiate CNN object, set parameters, fill in input data and fetch back output.

3.3 Network Training Algorithm

3.3.1 Local training

1. init/load a network architecture and parameters

2. for all data in mnist\_train

3. res = network.forwardLocal(data)

4. Network.calErrLocal(data.label)

5. Network.updateLocal()

(c1 Network local train)

Network.forwardLocal(data):

1. fp = data

2. for all layers in network

3. fp = layer.forwardLocal(fp)

4. return fp

(c2 Network local forward)

Network.calErrLocal(label)

1. err = label

2. for all layers in network

3. err = layer.calErrLocal(nextLayer)

(c3 Network local calErr)

Network.updateLocal()

1. for all layers in network

2. layer.updateLocal()

(c4 Network local update)

3.3.2 Cluster training

1. instantiate/load a network architecture and parameters

2. partition mnist\_train by batch size

3. for all partition in mnist\_train

4. data = sparkContext.parallelize(partition) //parallel as RDD

5. res = network.forwardCluster(data)

6. Network.cacheResult()

7. Network.calErrCluster(data.label)

8. Network.cacheErr()

9. Network.updateCluster() //reduce at each layer

10. Network.clearCache()

(c5 Network cluster train)

Network.cacheResult():

1. for all layers in network

2. layer.outputCluster.cache() //cache RDD

(c6 Network cache output)

Network.cacheErr():

1. for all layers in network

2. layer.deltaCluster.cache() //cache RDD

(c7 Network cache Err)

Network.clearCache()

1. for all layers in network

2. layer.outputCluster.unpersist() //uncache RDD

3. layer.deltaCluster.unpersist() //uncache RDD

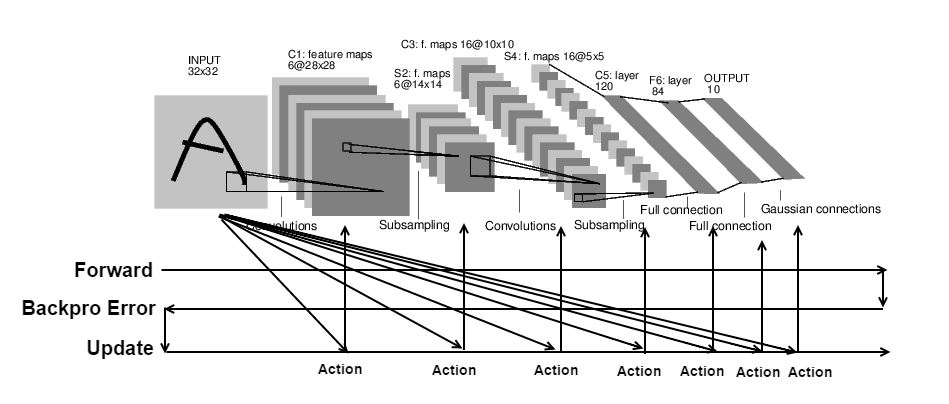
(c8 Network clear cache)

Network.forwardCluster()/calErrCluster()/updateCluster() has similar implementations in CNN.class. The difference on implementations exists in the corresponding layer component implementations.

Cache() function:

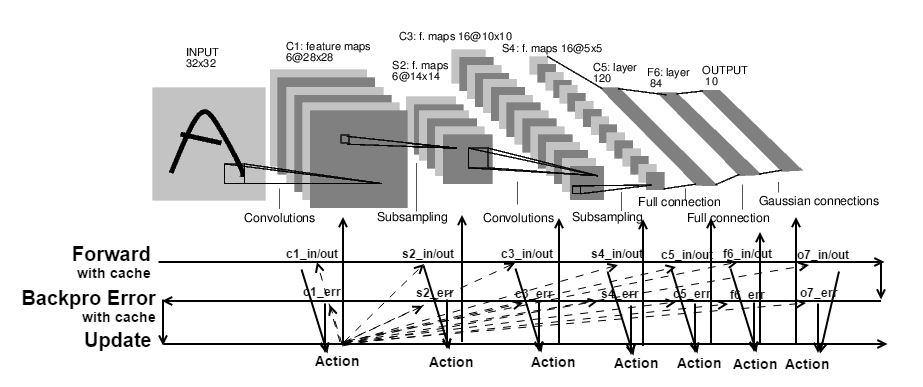
The reason to cache() RDD of middle computational steps and then uncache() them is to save computational time and reduce unnecessary replications. In order to calculate each middle step only once, they need to be cache()ed and reused. After batch update(), we need to release the cache by unpersist() to save memory.

Pic10 and pic11 illustrate the training procedure of LeNet5 without (pic10) or with (pic11) cache(). LeNet5 totally have 7 layers. After forwarding and back propagation, there are 7 action operations to update weights in 7 layers. Since Spark’s evaluation on RDDs is lazy (see section1.4 Spark computational model), we need to calculate all input/output/error value for all 7 layers sequentially from the input layer for each action operation. Even if they are all calculated at the first action operaion, the same calculation will replicate at the second action operation for second layer update.



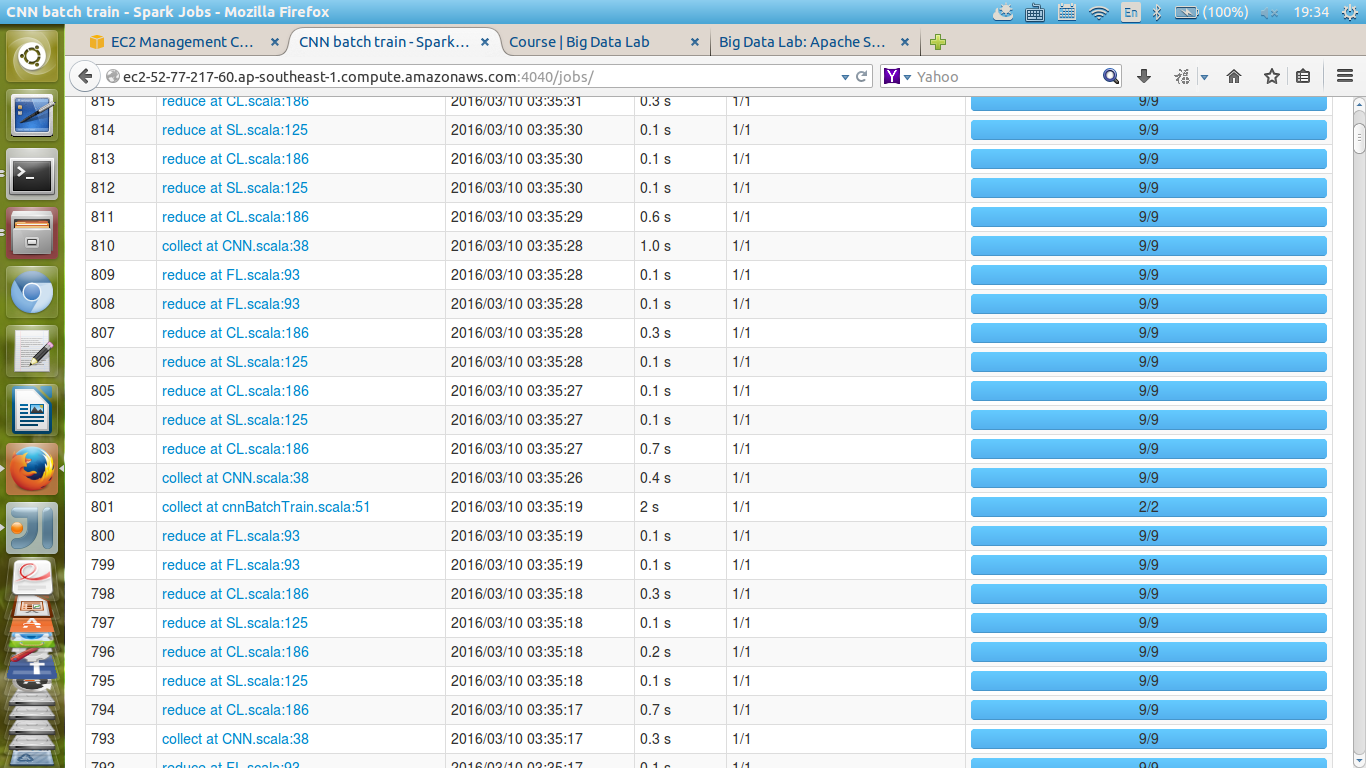
(pic10 cluster train without cache())

As showed by pic11, all input/output/error values for 7 layers are calculated and cache()ed at the first action and reused in following 6 actions.



(pic11 cluster train with cache())

Pic12 shows the first action operation take the longest time among all action operations, since it will calculate and cache() all previous variables.



(pic12 time consumption at each action operation)

3.4 Network Classification Algorithm

The network classification algorithm is simply the forward() function. After getting the forwarding result, we get label from the forwarded vector by maximum likelihood:

(f21 maximum likelihood for label classification)

Local mode:

1. load a network architecture and parameters from file

2. label = Array()

3. for all data in mnist\_test

4. res = network.forwardLocal(data)

5. label = label ++Array(argmax(res))

(c9 Network local classify)

Cluster mode

1. load a network architecture and parameters from file

2. data = sparkContext.parallelize(mnist\_test) //parallel to RDD

3. res = network.forwardCluster(data)

4. label = res.map(argmax(\_)).collect() //collect to local

(c10 Network cluster classify)

3.5 layer components detail implementations

In order to parallel the training procedure of a neural network, the best way is to construct the algorithm to be a pipeline so that all data go straight forward without branching, merging or traceback. Functional programming is a programming paradigm that complies with this requirement. There are several key points in implementing each operation on the pipeline:

1. Use collection operations instead of iterators. The data partitions need to be treated as a whole for RDD parallelism. Iterators (for loops) does not comply with functional programming methodology so that Spark do not treat it as a transformation operator.

2. Use immutable variables (val) rather than mutables (var), unless that the variables are updated across iterations (i.e. weights, bias).

Package information:

import breeze.linalg.{DenseVector=>DV, DenseMatrix=>DM, sum}

import org.apache.spark.rdd.RDD

import breeze.numerics.{tanh,sigmoid}

(c11 package information)

3.5.1 layer.scala

Declaration:

abstract class layer extends Serializable

(c12 layer.scala declaration)

Variables (setters omitted)

var eta: Double = 0.01

var momentum: Double = 0.9

var numPartition: Int = 2

(c13 layer.scala variables)

function abstraction

def forward(input:RDD[Array[DM[Double]]]):RDD[Array[DM[Double]]]

def calErr(nextLayer:SL):Unit

def calErr(nextLayer:CL):Unit

def calErr(nextLayer:FL):Unit

def calErr(nextLayer:OL):Unit

def calErr(target:RDD[DV[Double]]):Unit

def adjWeight():Unit

def clearCache():Unit

def forwardLocal(input:Array[DM[Double]]):Array[DM[Double]]

def calErrLocal(nextLayer:SL):Unit

def calErrLocal(nextLayer:CL):Unit

def calErrLocal(nextLayer:FL):Unit

def calErrLocal(nextLayer:OL):Unit

def calErrLocal(target:DV[Double]):Unit

def adjWeightLocal():Unit

(c14 layer.scala function abstraction)

Utility functions: (mathematics: f17-20)

def flattenOutput(output:Array[DM[Double]]):DV[Double]=

{new DV(output.map(\_(0,0)))}

def formatOutput(output:DV[Double]): Array[DM[Double]]=

{output.toArray.map(DM.fill(1,1){\_})}

def expandDM(mat:DM[Double],scalar:Int):DM[Double]={

DM.tabulate(Mat.rows\*scalar,Mat.cols\*scalar){case(i,j)=>

Mat(i/scalar,j/scalar)

}

}

def convolveDM(Mat:DM[Double],kernel:DM[Double]):DM[Double]={

DM.tabulate(Mat.rows-kernel.rows+1,Mat.cols-kernel.cols+1){case(i,j)=>

sum(Mat(i to i+kernel.rows-1,j to j+kernel.cols-1):\*kernel

}

}

def rot180(in:DM[Double]):DM[Double]=

{DM.tabulate(in.rows,in.cols){case(i,j)=>in(in.rows-i-1,in.cols-j-1)}}

def padDM(mat:DM[Double],padding:Int):DM[Double]={

val vert=DM.fill(padding,mat.cols){0d}

val horz=DM.fill(mat.rows+2\*padding,padding){0d}

DM.horzcat(horz,DM.vertcat(vert,mat,vert),horz)

}

def makeZip(s: Array[RDD[DM[Double]]]):RDD[Array[DM[Double]]]={

if(s.length==1) s.head.map(e=>Array(e)

else s.head.zip(makeZip(s.tail)).map(e=>Array(e.\_1)++e.\_2)

}

def unzip(s:RDD[Array[DM[Double]]],numOfCol:Int):Array[RDD[DM[Double]]]=

{(0 until numOfCol).map(i=>s.map(arr=>arr(i)))}

(c15 layer.scala utility functions)

3.5.2 CL.scala

Declaration:

class CL(val numfm:Int, val dim\_conv:Int) extends layer

(c16 CL.scala declaration)

Variables (setters omitted)

var fm\_input\_map: Array[Array[Int]] =

(0 to numfm-1).map(x=>Array(x)).toArray

var outputIndex: Array[Array[(Int,Int)]] = calcInvIndex(fm\_input\_map)

var kernel: Array[Array[DM[Double]]] = fm\_input\_map.map(arr=>

arr.map(fm=>DM.rand[Double](dim\_conv,dim\_conv):\*=2d:-=1d))

var bias: Array[Double] = (DV.rand(numfm):\*=2d:-=1d).toArray

var input: RDD[Array[DM[Double]]] = \_

var delta: RDD[Array[DM[Double]]] = \_

var output: RDD[Array[DM[Double]]]= \_

var inputLocal: Array[DM[Double]] = \_

var deltaLocal: Array[DM[Double]] = \_

var outputLocal: Array[DM[Double]]= \_

var kadj: Array[Array[DM[Double]]] =

fm\_input\_map.map(arr=>arr.map(fm=>DM.fill[Double](dim\_conv,dim\_conv){0d}))

var badj: Array[Double] = (DV.fill(numfm){0d}).toArray

(c17 CL.scala variables)

Input mapping:

// set input mapping, format and randomize kernels

def set\_fm\_input\_map(map:Array[Array[Int]]): Unit

// calculate invert index for input mapping

def calcInvIndex(input\_map:Array[Array[Int]]): Array[Array[(Int,Int)]]

(c18 CL.scala input mapping functions)

Networking functionality:

override def forward(input:RDD[Array[DM[Double]]]):RDD[Array[DM[Double]]]

override def forwardLocal(input: Array[DM[Double]]): Array[DM[Double]]

override def calErr(nextLayer:FL): Unit

override def calErrLocal(nextLayer:FL): Unit

override def calErr(nextLayer:SL): Unit

override def calErrLocal(nextLayer:SL): Unit

override def adjWeight(): Unit

override def adjWeightLocal(): Unit

override def clearCache(): Unit

(c19 CL.scala networking functionality)

3.5.3 SL.scala

Declaration:

class SL (val numfm: Int, val dim\_neighbor:Int) extends layer

(c20 SL.scala declaration)

Variables (setters omitted)

var weight:Array[Double] = Array.fill(numfm){scala.util.Random.nextDouble()\*2-1d}

var bias:Array[Double] = Array.fill(numfm){scala.util.Random.nextDouble()\*2-1d}

var output:RDD[Array[DM[Double]]]=\_

var delta:RDD[Array[DM[Double]]]=\_

var input:RDD[Array[DM[Double]]]=\_

var outputLocal:Array[DM[Double]]=\_

var deltaLocal:Array[DM[Double]]=\_

var inputLocal:Array[DM[Double]]=\_

var wadj:Array[Double] = Array.fill(numfm){0d}

var badj:Array[Double] = Array.fill(numfm){0d}

(c21 SL.scala variables)

Networking functionality:

override def forward(input:RDD[Array[DM[Double]]]):RDD[Array[DM[Double]]]

override def forwardLocal(input: Array[DM[Double]]): Array[DM[Double]]

override def calErr(nextLayer:CL): Unit

override def calErrLocal(nextLayer:CL): Unit

override def adjWeight(): Unit

override def adjWeightLocal(): Unit

override def clearCache(): Unit

(c22 SL.scala networking functionality)

3.5.4 FL.scala

Declaration:

class FL (val num\_in:Int, val num\_out:Int) extends layer

(c23 FL.scala declaration)

Variables (setters omitted)

var weight:DM[Double] =

DM.fill[Double](num\_in, num\_out){scala.util.Random.nextDouble()\*2-1d}

var bias: DV[Double] =

DV.fill[Double](num\_out){scala.util.Random.nextDouble()\*2-1d}

var delta: RDD[DV[Double]] = \_

var output: RDD[DV[Double]] = \_

var input: RDD[DV[Double]] = \_

var inputLocal: DV[Double] = \_

var outputLocal: DV[Double] = \_

var deltaLocal: DV[Double] = \_

var wadj:DM[Double] = DM.fill[Double](num\_in,num\_out){0d}

var badj:DV[Double] = DV.fill[Double](num\_out){0d}

(c24 FL.scala variables)

Networking functionality:

override def forward(input:RDD[Array[DM[Double]]]):RDD[Array[DM[Double]]]

override def forwardLocal(input: Array[DM[Double]]): Array[DM[Double]]

override def calErr(nextLayer:FL): Unit

override def calErrLocal(nextLayer:FL): Unit

override def calErr(nextLayer:OL): Unit

override def calErrLocal(nextLayer:OL): Unit

override def adjWeight(): Unit

override def adjWeightLocal(): Unit

override def clearCache(): Unit

(c25 FL.scala networking functionality)

3.5.5 OL.scala

Declaration:

class OL (override val num\_in:Int, override val num\_out:Int)

extends FL(num\_in, num\_out)

(c26 OL.scala declaration)

Variables (variables are all inherited)

Networking functionality:

override def calErr(target: RDD[DV[Double]]): Unit

override def calErrLocal(target: DV[Double]): Unit

def loss(output:DV[Double],target:DV[Double]): Double

(c27 OL.scala networking functionality)

3.6 Correctness checking

3.6.1 Correctness of local training (stochastic gradient descent)

To ensure the correctness of gradient descent implementation, gradient check is adapted for each layer, basically by verifying the gradient(partial derivative) by formula[5]:

(f22 stochastic gradient check)

3.6.2 Correctness of cluster training (mini-batch gradient descent)

To ensure the correctness of batch gradient descent, several approaches are used:

1. A batch of same input samples **vs** SGD with same input sample(by comparing weight updates)

(f23 batch gradient check 1)

2. batch size=1 **vs** SGD with same input sample

(f24 batch gradient check 2)

Besides, the experiment result shows that the LeNet5-like network architecture achieves quite similar accuracy with those shown in Yann’s work (see section2.4 LeNet5 and MNIST), which also prove the correctness of implementation.

**Chapter 4. Deployment and experiment**

4.1. MNIST pre-processing

The original MNIST database contains 60000 samples for training and 10000 samples for test in IDX file format, each sample from which is a 28\*28 from 0-255 (inclusive). There are few preprocessing work been done before flushing them into the network model:

1. Format:

The whole database is read out from IDX format to csv format by a python script, which actually increase the file size but easy for Spark program to handle.

2.Partition:

The training data file is partitioned into 6 separate files namingly from mnist\_train1 to mnist\_train6 for sake of fitting into 512M executor memory.

3. Padding and scaling:

Each matrix in the sample is 28\*28 which does not fit in LeNet5 input dimension (32\*32). They are padded before training. Also, all data are normalized by a scalar (1/255) in order to fit in range 0 to 1.

4.2. Experiment on MLP

The experiment on MLP is done in the midterm checkpoint of this project, as my first taste on Spark as well as neural network and machine learning. Later as I finished measuring speed-up ratio of CNN on Spark, I didn’t go back to simple MLP. The experiment been proposed here is exactly similar with my midterm report of this project.

4.2.1 Architecture

|  |  |  |
| --- | --- | --- |
| Network architecture | Input layer | 28\*28=784, no padding preprocess for input data |
| hidden layer | 2 layer, 30 units each |
| Output layer | 10 units |
| classification | | maximum likelihood (argmax) |
| Loss function | | Squared error loss function(see f2) |

(t5 MLP experiment architecture)

4.2.2 Deployment

The Spark program for MLP experiment is deployed on my own laptop, for the reason that the evaluation figure is only accuracy.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Item | OS | Jdk | Scala | Spark | maven | Hadoop |
| version | Ubuntu kylin 14.10 | 1.7.0 | 2.10.4 | 1.5.1 | 2.2.1 | 2.6 |

(t6 MLP experiment deployment)

4.2.3 Accuracy

After 3 epochs (totally 60000\*3=180000 samples) of local training with learning rate = 0.5 and momentum = 0.9:

the MLP model achieved classification accuracy of 93.9% of MNIST test dataset.

4.3. Experiment on LeNet5-like CNN architecture

The goal of carrying out experiment on CNN architecture is to come up with classification accuracies and speed-up ratios as evaluation figures.

4.3.1. Architecture

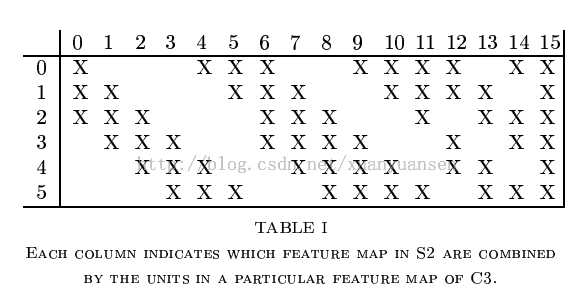
The architecture used in this experiment is very similar with Yann’s LeNet5 model. The activation function in this experiment is sigmoid function, instead of tanh function in input/hidden layers and RBF in output layers as Yann proposed[10].

Input: 32x32 image

C1: convolutional layer with 6 5x5 kernels, output 6 28x28 feature maps, totally 156 trainable parameters (with bias).

S2: mean subsampling layer with 6 2x2 neighborhoods, output 6 14x14 feature maps, totally 12 trainable parameters (with bias).

C3: convolutional layer with 60 5x5 kernels, output 16 10x10 feature maps, totally 1516 trainable parameters (with bias). The connection topology is shown in pic8 below



(pic13: Lenet5 S2 to C3 input map)

S4: mean subsampling layer with 16 2x2 neighborhoods, output 16 5x5 feature maps. Totally 32 trainable parameters (with bias).

C5: convolutional layer with 1920 5x5 kernels, output 120 1x1 feature maps (a vector of length 120), each output feature map is fully connected with previous layer 16 maps. Totally 48120 trainable parameters (with bias).

F6: fully connected hidden layer with 84 units. Totally 10164 trainable parameters (with bias).

O7: fully connected output layer with 10 units. Totally 850 trainable parameters (with bias).

Detailed calculation on trainables are shown:

Number of trainable parameters:

|  |  |  |  |
| --- | --- | --- | --- |
| Number | C1 | S2 | C3 |
| Weight | 6\*5\*5=150 | 6\*1 | (3\*6+4\*9+6\*1)\*5\*5 = 1500 |
| Bias | 6\*1 = 6 | 6\*1 | 6+9+1 = 16 |
| Trainables | 156 | 12 | 1516 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Number | S4 | C5 | F6 | O7 |
| Weight | 16\*1 | 120\*16\*5\*5=48000 | 120\*84=10080 | 84\*10=840 |
| Bias | 16\*1 | 120 | 84 | 10 |
| Trainables | 32 | 48120 | 10164 | 850 |

(t7 number of trainable parameters in LeNet5)

4.3.2. Training parameter

Local train:

Model: newly instantiated

Number of epoch: 5

Learning rate: 0.01,0.0095,0.009,0.0085,0.008

Momentum:0.9

Cluster train:

Model: 1 epoch local trained model

Batch size: 50 samples (1200 iteration/epoch)

Number of epoch: 4 epoch cluster train

Learning rate: 0.0095,0.009,0.0085,0.008

Momentum: 0.9

4.3.3. EC2 deployment

There are two clusters been deployed on Amazon Elastic Cluster(EC2) for this experiment as shown below:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Num of nodes | Instance type | Num of core/node | RAM/ node | Spark executor RAM/node |
| Group1 | 1 machine | t2.micro | 1 | 1G | 512M |
| Group2 | 1 master +  9slave |
|  | Jdk | scala | maven | Hadoop | Spark |
| Group1 | 1.7 | 2.10.4 | 2.2.1 | 2.6 | 1.5 |
| Group2 | Not used |

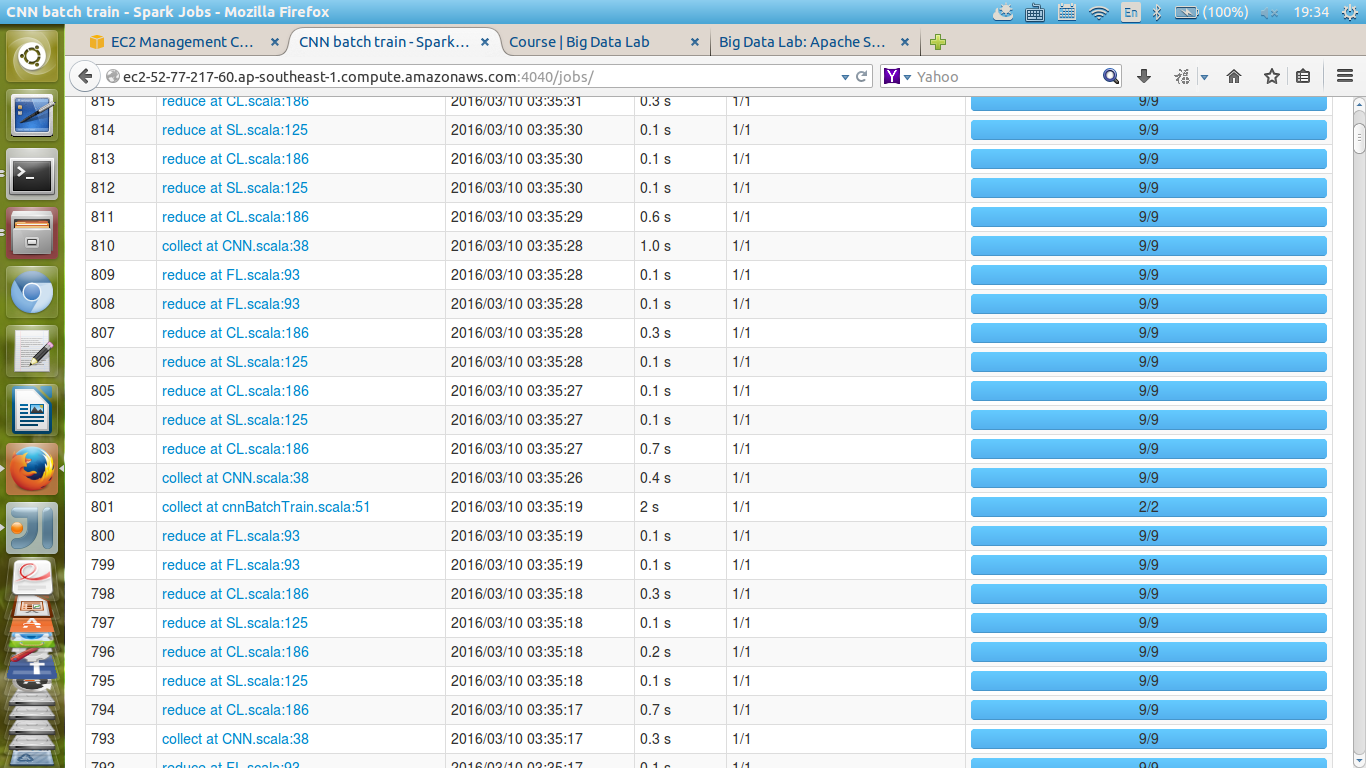
(t8 EC2 deployment)

All local train (SGD) are deployed on group 1.

All cluster train (mini-batch) are deployed on group2

Recall purpose of experiment on cluster is to ensure Spark can save the training time of CNN and come up with a speed-up ratio. As the batch size couldn’t be too large, and each iteration no not require large memory to cache and compute everything, this enables the feasibility to use free instance t2.micro on EC2 to carry out the experiment.

The following screenshot shows the Spark UI on the CNN training



(pic14 Spark UI on CNN training program)

4.3.4. Accuracy

The table below shows the accuracy on test set classification for each epoch

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| epoch | 1 | 2 | 3 | 4 | 5 |
| Local (Group1) | 0.9248 | 0.9625 | 0.9724 | 0.977 | 0.9811 |
| Cluster (Group2) | 0.9482 | 0.9695 | 0.9731 | 0.9746 |

(t9 CNN experiment accuracy)

To compare performance, line chart shows(local, cluster and Yann’s experiment result on LeNet5 without distortion on training data):

(pic15 CNN experiment accuracy comparison)

4.3.5. Time consumption and speed up

The table below shows the experiment results on running time.

SUR1 is speed-up ratio compared with local train.

SUR2 is speed-up ratio comparing cached/uncached training

|  |  |  |  |
| --- | --- | --- | --- |
| Description | Time/epoch | SUR1 | SUR2 |
| Local train (Group1) | 350 mins | -- | -- |
| Cluster train (Group 2) batchsize=50, cached | 83min | 4.22 | 4.01 |
| Cluster train (Group 2) batchsize=50,uncache | 333min | 1.05 | -- |
| Cluster train (Group 2) batchsize=100,cached | 69min | 5.07 | 3.9 |
| Cluster train (Group 2) batchsize=100,uncache | 269min | 1.30 | -- |
| Cluster train (Group 2) batchsize=10000,cached | 45min | 7.78 | 4.0 |
| Cluster train (Group 2) batchsize=10000,uncache | 180min | 1.94 | -- |

(t10 CNN experiment time consumption)

4.4. Analysis on the experiment

4.4.1 Conclusions from the experiment

1. The accuracy on MNIST test set of this project shows my implementation approximates the classic LeNet5 model proposed by Yann quite well, which, in another way, certifies the correctness.

2. Applying cache() function in this project is a successful optimization on algorithm acceleration. Without cache(), the cluster training algorithm will be quite cumbersome.

3. Parallel the training procedure on Spark is successful. In this case, the computational power brought by parallel nodes overcomes the inter-networking communication tradeoff among them.

4.4.2 Potential improvements

There are two possible ways to further improve the performance.

The first way is to adjust the batch size. As we can see from the example of theano experiment (section2.3 and pic5), a well tuned batch size can achieve best performance of accuracy and time. In this experiment, actually the batch size is not well tuned. There are totally 9 executors to partition 50/100/10000 batch size data which will cause load-imbalancing problem.

Another way is to use better instance on EC2. For business or research applications, better instance means better performance. Better instance may provide more cores, more RAM and better inter-networking.

More computational cores will certainly help to improve time efficiency a lot. Duplicating number of nodes may suffer from more serious networking problem, while duplicating cores on each node may not.

More RAM on each node may improve time efficiency only when the batch size is large. In this project, the network training problem is not so RAM-consuming. The following provide analysis:

The entire MNIST\_train dataset in csv format is 104.4MB

* Read them in as Matrices of Double and label, totally the size is:

60000samples \* 769 Double/sample\*8B/Double = 369MB

* Although spark need to store lineage and cache()ed results in RAM and need spare RAM space to keep run fast, but 369MB is already the largest RDD size (regardless of matrix type information) on the case that we are doing full-batch training, on a cluster of executors with 512MB RAM each.
* Conclusion can be draw that increase RAM space will certainly improve time efficiency, but may not be as significant as increasing number of cores. CNN training does not require as much RAM as other algorithms on Spark need.

Better inter-networking will surely improve the time efficiency. T2.micro instance on EC2 is accused that it has serious networking problem. In my experiment, disassociation of executors is very frequent. Finally, I tuned Spark configurations as below to enlarge the allowed timeout for network failure and inter-networking failure, detect heartbeat more frequently and allow longer heartbeat pauses to finish the experiment.

|  |  |  |
| --- | --- | --- |
| parameter | value | default |
| spark.network.timeout | 600s | 120s |
| spark.akka.timeout | 600s | 100s |
| spark.akka.heartbeat.interval | 100s | 1000s |
| spark.akka.heartbeat.pauses | 6000s | 5000s |

(t11. Spark running configuration)

**Chapter 5. Project schedule**

|  |  |  |
| --- | --- | --- |
| Finish date | Event | Deliverable |
| 2015/Nov/10 | Implement BPNN on scala | Scala program |
| 2016/Nov/15 | Learn MNIST data format and evaluate BPNN | Test program and error rate, midterm report |
| 2016/Jan/15 | Learn CNN, math proof explore machine learning techniques | Math proof, implementation strategy |
| 2016/Feb/10 | Layer components and LeNet5 | Scala program  (local program) |
| 2016/Feb/15 | Evaluate local LeNet5 on MNIST | Testcases, gradient checking, error rate |
| 2016/Mar/1 | Parallel CNN, write testcases | Spark program |
| 2016/Mar/23 | Deploy on EC2, run testcases | Time comsumption evaluation |
| 2016/Mar/30 | This report | report |

(t12 project schedule)

**Chapter 6. Conclusions**

Although GPU based deep learning is still the main stream in the research community, this project explores the feasibility of implementing CNN, a typical kind of deep neural network with the help of Spark on general usage cloud platforms. It may provide an alternative solution for this topic, since CPU based cluster computing also have some advantages over GPU clusters(see section 2.2).

As was tested out, convolutional neural networks can maintain its correctness on Spark clusters when it’s paralleled, as showed by the classification accuracy in the experiment. Also, parallel CNN training in the way of batch gradient descent is a feasible way to achieve some sort of improvement in time efficiency according to the computational resources it attaches to.

Also, this project provides convenient and complete API, as well as CNN examples, testcases for Spark developers to construct their own CNN. As atomic components in this project are layer models, it also can be adapted to various batch image processing tasks as an extension of this project, since convolutions and subsamplings are basic operations in image processing.

There are also lots of Spark developers working on the implementation of deep learning algorithms, although they seldom haven’t got merged with recent Spark versions. However, deep learning on Spark may have a quite promising future with the development of Spark and cloud computing, as well as integrating Spark with GPU computing.

**References**

**1. An Introduction to Convolutional Neural Networks. (n.d.). from** [**http://white.stanford.edu/teach/index.php/An\_Introduction\_to\_Convolutional\_Neural\_Networks**](http://white.stanford.edu/teach/index.php/An_Introduction_to_Convolutional_Neural_Networks)

**2. Apache Spark™ - Lightning-Fast Cluster Computing. (n.d.). from** [**http://spark.apache.org/**](http://spark.apache.org/)

**3. Bishop, C. M. (2006). *Pattern recognition and machine learning*. New York: Springer.**

**4. Convolutional Neural Networks (LeNet) using theano. (n.d.). from** [**http://deeplearning.net/tutorial/lenet.html**](http://deeplearning.net/tutorial/lenet.html)

**5. Andrew. N. (n.d.). Coursera - Machine Learning. from** [**https://www.coursera.org/learn/machine-learning**](https://www.coursera.org/learn/machine-learning)

**6. Geoffrey. H. (n.d.). Coursera - Neural Networks for Machine learning. from** [**https://class.coursera.org/neuralnets-2012-001**](https://class.coursera.org/neuralnets-2012-001)

**7. Hamstra, M., & Zaharia, M. (2013). *Learning Spark Lightning-fast big data analytics*. Sebastopol, CA: O'Reilly & Associates.**

**8. Han, J., & Kamber, M. (2001). *Data mining: Concepts and techniques*. San Francisco: Morgan Kaufmann.**

**9. Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R., . . . Darrell, T. (2014). Caffe: Convolutional Architecture for Fast Feature Embedding. *ArXiv Preprint ArXiv:1408.5093*.**

**10. Lecun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE Proc. IEEE,* *86*(11), 2278-2324.**

**11. MNIST Demos on Yann LeCun's website. (n.d.). from** [**http://yann.lecun.com/exdb/lenet/index.html**](http://yann.lecun.com/exdb/lenet/index.html)

**12. Mitchell, T. M. (1997). *Machine Learning*. New York: McGraw-Hill.**

**13. Russell, S. J., & Norvig, P. (1995). *Artificial intelligence: A modern approach*. Englewood Cliffs, NJ: Prentice Hall.**

**14. Unsupervised Feature Learning and Deep Learning Tutorial. (n.d.). from** [**http://deeplearning.stanford.edu/tutorial/**](http://deeplearning.stanford.edu/tutorial/)

**15. Apache/spark. (n.d.). from** [**https://github.com/apache/spark/tree/master/mllib/src/main/scala/org/apache/spark/ml**](https://github.com/apache/spark/tree/master/mllib/src/main/scala/org/apache/spark/ml)

**16. Jie, G. Y. (2014). *Data Processing with Spark, Technology, Application and Performance Optimization*. Beijing: China Machine Press.**

**17. Jake Bouvrie, ” Notes on Convolutional Neural Networks”, 2006**

**18. J.Li,et al., ”Brief introduction of back propagation (BP) neural network algorithm and its improvement, Advances in Intelligent and Soft Computing, pp553-558 , 2012**

**19. M.Zaharia, et al.,”Resilient Distributed Datasets: A fault-tolerant abstraction for in-memory cluster computing”, NSDI 2012, 2012, Best paper award**

**20. M.Zaharia, et al.,”Spark: Cluster Computing with Working Sets”, HotCloud 2010, 2010**