Machine Problem 2 Report

Yihui He*

I. Implementation

I CHOOSE residual network as my archtecture, which also combined with other algorithms like moving average ensembles, PAC whitening, kmeans, LSUV initialization, etc. I implemented my residual network in tensorflow.

Key points of my implementation are shown below.

A. Layers construction

In tensorflow, I use *tf.placeholder* to represent CIFAR100 data. and use *tf.Variable* to store trainable parameters. Note that, in order to add regularization strength to hyperparameters. I add hyperparameters and loss function to a collection named *losses*.

```
weight_decay=tf.mul(tf.nn.l2_loss(var)
    ,wd,name='weight_loss')
tf.add_to_collection('losses',
    weight_decay)
```

When building optimizer, I add all collection items together in order to get cross entropy.

```
cross_entropy=tf.add_n(
    tf.get_collection('losses'))
```

B. Learning rate update

I want to employ Nesterov momentum as my update method. However, tensorflow doesn't provide it. So I use *tf.placeholder* to store learning rate. Before each *session.run* I compute learning rate myself. And use *feed_dict* to feed in learning rate.

C. Visualization

With tensorboard, it is easy to visualize details information. I add layer responses, loss history, and accuracy history to summary writer. Then visualize them using tensorboard. Some of my figures are downloaded from tensorboard.

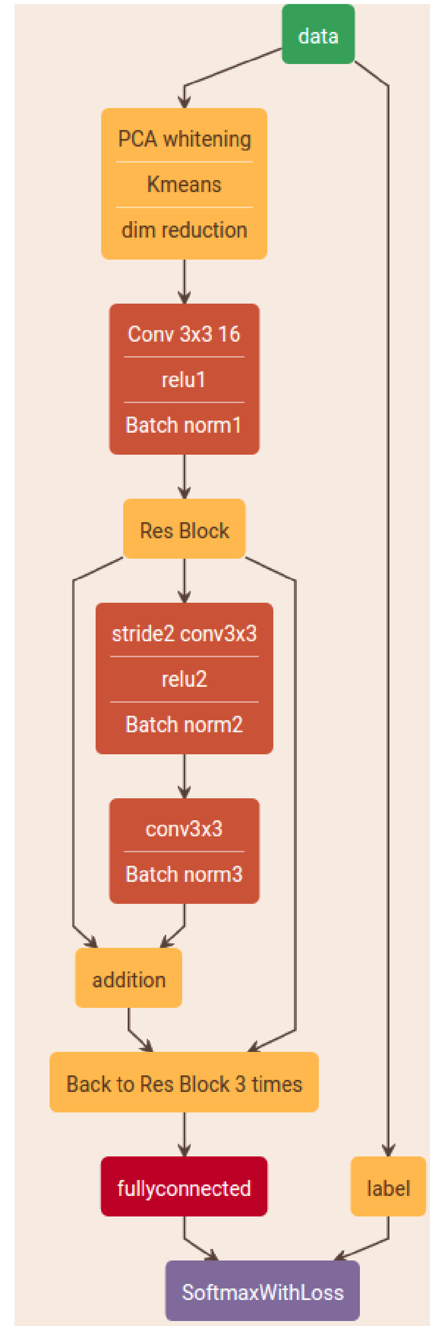

Fig. 1. residual neural network

II. Archtecture

The archtecture graph is shown in Fig.II.

*Exchange student, 2nd year CS undergrad from Xi'an Jiaotong University, yihuihe@foxmail.com

My archtecture can be described as followed.

1) I employ PCA whitening(Full resolution) and Kmeans to preprocessing data.
2) Since the demention after Kmeans is too high (27x27x1600), I use PCA to do dimention reduction. Transform input data to 27x27x256.
3) As in the original residual network paper, I use 7 layers which contents ReLU, batch norm, addition.
4) Finally, I use a fully connected layer connect neural network to 20 classes softmax.

One thing to also need to notice that is, I use dropout(0.5) in each trainable layer.

III. Model Building

1) archtecture of the convNet
   I compared 3 archtectures: 7 layers, 13 layers and 19 layers. They can go through 36, 18, 12 epoches within 3 hours, respectively. However, 3 hours training on CPU seems not enough for showing their difference. Their accuracy do not differ too much. For simplicity, I choose 7 layers as my archtecture.
2) number of filters
   I use 16 filters in the first convolutional layer, double the number of filters after each pair of convolutional layers. At the last layer, I have 64 filters.
3) the filter size
   As in traditional convolutional network, all of my filters have size of 3x3.
4) regularization strength
   The original residual network have no regularization strength. However, from my experiment, it can easily overfit. So I use dropout as regularization, which have dropout probability of 50%.
5) learning rate
   I tried learning rate of 0.01 and 0.1. 0.01 makes the training slow, from my experiment.
6) batch size
   Traditional residual network uses 256 batch size. I tried 64, 128, and 256. Interestingly, 256 will make it converge faster in the first few minutes. Within 3 hours, they do not have much difference.
7) My last layer is fully connected from convolutional layer to softmax. So there's no hidden neurons in my network.
8) learning strategy
   I tried Adagrad and Nesterov momentum. Adagrad learns faster within 3 hours, which is consistent with others' experiment(Adagrad learns faster, but can not beat Nesterov momentum in the long run)
9) Platform
   I tried two platforms: tensorfow and mxnet. I use 20 layers residual neural network to compare their effiency. According to soem technical reports online, in the backpropagation period, mxnet is 4x faster than tensorflow. I don't know how to compare time cost in each period. So I implement the same archtecture in both framework respectively, and compare training time cost. Interestingly, in my experiment, mxnet is only a little bit faster than tensorflow when I'm doing training.

IV. Extra Credits

I put results in the next section, in order to compare our different neural networks I built and did experiments.

A. *number of hidden neurons*

I have one hidden layer at the end of residual neural network. I compared three different numbers of hidden neurons: 64,128,256. 64 is used by the original paper. At the beginning, when I'm implementing ResNet, I felt that 64 is too small for a hidden layer, since other archtectures like AlexNet using 384 hidden nodes at the end. However, when I'm comparing these three different numbers of hidden neurons. I found that 128 and 256 didn't show any superior, only time cost increase a little bit.(5% which can be ignored)

I also tried adding an additional hidden layer(200 nodes) into ResNet. Strangely, it learns slower, and the accuracy stuck at 50% in one hour. Never increase for the remaining 2 hours. I also add an additional hidden layer to my machine problem one architecture, the same thing happen again. Maybe it is over complex or it needs more time to train.

B. *filter sizes of the pooling layer*

In the middle layers, I adopt mixture of pooling and convolution, which is 2x2 convolution with stride of 2. In the end of convolutional stacks,

there's a global pooling layer. I tried to modify 2 to 3, which make accuracy drop 5%.

I also tried to replace the final global pooling layer with a stride 2 pooling layer, which does not change final accuracy.

C. filter sizes of the convolutional layer

By default, the reception field size of filters is 3x3 usually. I tried 5x5 in the first layer, differences can't be seen. However, filter depth is game changer. Most image recognition winner use very deep filter depth(500 or more). We can regard filter depth as number of meta features in each layer. In my experiment, due to our 3 hours training time limitation. The significance of filter depth hasn't been shown. There's no difference in result. One reason should be when I increase filter depth, training speed also drop. Although loss decrease more in each epoch. The final accuracy seems no difference. Kind of balance.

D. initialization methods

I tried three initialization methods: LUSV initialization, Kaiming He's initialization, and orthognal initialization.

1) LSUV
   LSUV which does a layer-by-layer normalization with a mini-batch, works very bad in my experiment. It makes initial loss very high. The orginal paper said that we should normalize each layer's response variance by 1. Turn down the layer response variance can make initial loss back to normal. However, it shows no difference in the long run.
2) Kaiming He's initialization.
   What this initialization method doing is basically normalizing filters' weight by $sqrt(2/Fan_{in}/receptionFieldSize)$. It is wonderful from my experiment. It makes loss drop faster at the beginning. The orginal paper also claims that this method gurantees deep network (more than 30 layers) converge, and orthognal initialization cannot.
3) orthognal
   This initialzation use SVD decomposition to make filters' weights orthognal. Differences haven't been seen from my experiments. Code is as follow.

```
flat_shape = (shape[0], np.prod(shape[1:]))
a = np.random.standard_normal(flat_shape)
u, _, v = np.linalg.svd(a, full_matrices=False)
q = u if u.shape == flat_shape else v
q = q.reshape(shape)
```

E. dropout

In machine problem one, I find that, dropout could significantly improve accuracy(5%). However, this time, I found that, dropout make my training procedure slower. The author, Kaiming He claims that residual network uses its special architecture to prevent overfitting. He don't use any kind of regularization. This is consistent with my experiment.

I also tried dropConnect, a improved version of dropout, which drop the weights instead of activation. It is the same fate with dropout in ResNet.

F. different optimization method

Teacher optimization
This optimization method comes from paper: do deep neural network really need that deep?.
This method is two-folds:

1) Train a deep neural network on the training dataset.
2) Store the deep neural network's prediction of log probabilities for each class on training dataset.
3) Train a shallow neural network. Instead of directly use the labels to do softmax optimization. It removes the softmax layer, and does a linear regression optimization on the log probabilities of shallow neural network and previously stored log probabilities of deep neural network.

Interestingly, this paper claims that it can not only speed up shallow neural network training, but also reach accuracy that traditional optimization method can not reach.

We change the cost function, The code of training part is as follow:

```
cost = tf.square(teacher_label -
  softmax_linear)
```

However, I don't have a good teacher model, my best teacher model(20 layers ResNet) only have 70% accuracy, which I trained for 1 day. Using this model as teacher, this optimization method works.

But result is worse than traditional method. I guess it is because my teacher model is bad that prevent me from reproducing this paper's result.

V. Results

In this section, I compare three archtectures I implemented: AlexNet, ResNet, and Kmeans Single Layer net . The following table shows my three methods accuracy:
(due to the random initialization, the following accuracy is not guranteed, but should be close)

TABLE I
Differences between update methods

	AlexNet	Kmeans	ResNet
parameters	1M	.4M	.13M
Layers	7	3	14
learning rate	.1	5×10^{-4}	.1
regularization	L2	Dropout,.3	None
epoch	10	140	18
Batch size	128	128	256
Time(min)	180	80	180
CIFAR10 Acc	82%	75%	84%
Train accuracy	90%	80%	86%
Test	56%	56%	63%

Interesting insights can be noticed in this table:

1) ResNet can reach the same accuracy with less parameters(10 times less than AlexNet).
2) ResNet does not suffer from increasingly deep layers.
3) Kmeans based feature method can still get good result with even only one hidden layer.
4) Without regularization, ResNet do not suffer very much from overfitting, compared with AlexNet.
5) ResNet and AlexNet can not show their power, due to training epoches are not enough.
6) Scalar of three methods on CIFAR10 and CIFAR100 is not consistent.

VI. Challenges

I encountered many challenges through this machine problem

Installing tensorflow
I have windows machine at first time. But I failed to install to onto my virtual machine. So I switch to Ubuntu. In Linux, I still cannot install tensorflow. And the problem I suffered is different from the one described in machine problem 2 description. Thanks to stackoverflow, I managed to install it.

making program work
Programming with tensorflow is really different from normal programming. Each layer you defined in archtecture is an operation. and variables can only be updated when you run sess one time. It is kind of like a black box. When I want to use Nesterov momentum, I can't find it in tensorflow library. Implementing it need to change learning rate from Variable to placeholder. And feed it as a feed dict on each run.

Tuning hyperparameters
This part differs from machine problem one a lot. convolutional neural network needs too much time to run. It is not possible for me to run a brute force search on the space of hyperparameters like mp1. So that, most of my hyperparameters are set according to papers.

VII. Possible Improvements

With longer training time, it is possible to reach higher accuracy.

If I add strong regularization on ResNet, maybe the accuracy will be improved.

A combination of feature based methods and ResNet maybe result in better accuracy.

I also read three papers: BinaryConnect, BinayNet, XNORnet. These papers claim that they can hugely cut memory cost and time cost. Although I guess implementing these algorithms must be annoying, maybe they can improve accuracy a lot.