Weekly report Feburary 19,2019

Rong Ding
dingrong@sjtu.edu.cn

Abstract

I finished the last two steps of my miniproject about VGG16.

1 Complementary explanations for the previous report

1.1 Describe the image classification task in a technical way

- Input . It is cifar-10 datasets. There are 5 batches for training and 1 batch for testing. Each picture is represent by an array of 3072 $(3 \times 32 \times 32)$. It is divided into 3 blocks . Each block represents RGB values in turn.
- Algorithm. I used AlexNet and VGG16 in my experiments. I might try ResNet later.
- Output. There are many items to record after training and testing, such as train accuracy , val accuracy ,test accuracy , loss and so on.
- Other information to keep track of. Learning rate, optimizer methods(e.g. Adam) and optimizer schedule.

1.2 Explanations for benefits of preprocessing data

I used two ways to preprocess data. First is random flip for data augmentation, which prevents overfitting. Second is normalization . This kind of technique make figures obey a overall distribution and become similar in means and variance , which helps loss to converge.

2 My experiments on VGG16

I have tried many ways to tune my VGG16 these days. I'd like to show my overall process including my mistakes.

2.1 General Discription

I didn't find pretrained VGG16 for cifar-10 , so I wrote it in pytorch by myself. I used Adam optimizer in the project.

2.2 Experiment 1 : Combination of different learning rates and batches

I use 100 epoch and no optimizer schedule in this part. I have tried learning rates of $1e-2$, $3e-3$ and $1e-3$, and I tried batch size of 100, 200 and 500. The best result is 91.5% in val set , with batchsize 500 and learning rate 1e-2.

2.3 Experiment 2: Trying different optimizer schedules

In this part I tried two new optimizer schedules, using initial learning rate of $1e-2$ and 100 epoch. The first schedule is $optim.lr\_scheduler.MultiStepLR()$, which

changes learning rate by multipling parameter γ according milestones. I used $optim.lr\_scheduler.MultiStepLR(self.optimizer, milestones = [40, 60, 80], gamma = 0.5)$ in one experiment and achieved a satisfactory val accuracy(92.0%). Here is the curve of its loss.

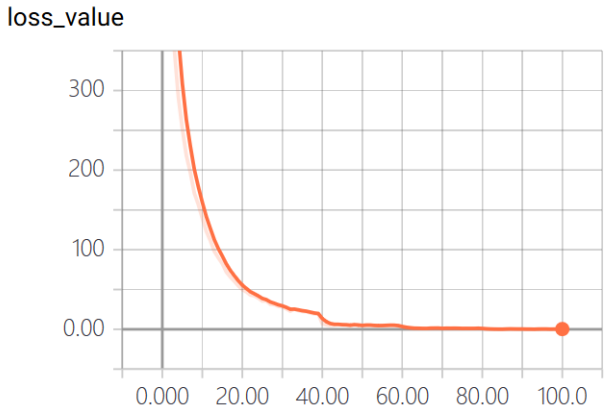


Figure 1: loss curve

As we can see, the curve drops distinctly at 40 and 60 epochs. It is because when epoch number is 40,60 or 80 , the learning rate will decrease by a half , which prevents loss to be stuck at a high value.
The second schedule is $optim.lr\_scheduler.ReduceLROnPlateau(self.optimizer, mode = 'min', factor = 0.5, patience = 10)$. When the loss doesn't decrease in 10 epoches, then the learning rate will be reduced by half. The val accuracy of obeying this schedule is 91.10%.

2.4 Experiment3 : Pretrained model

I didn't find pretrained VGG for cifar-10 on Internet. So I made my own pretrained model. In the original paper , pretrained model is used to initialize and accelerate training. Here are the different methods I have tried. I use 100 epoch and learning rate of 1e-2 in this part.

- I record parameters of a fine-tuned model and pretrained its classifier layer(a Lineaer layer of 512×10). I found that the loss decreased so quickly that its val accuracy reached above 90% after 10 epoches. Besides, it's obvious that the training time is much shorter $(3618s)$, which is nearly one fours of that before $(15443s)$. I think the reason is that parameters of previous layers have successfully extracted necessary features, so just tune the classifier layer can obtain a good result in a short time. However, after 100 epoch, compared to the previous fine-tune model,the network's final accuracy didn't improved. I supposed it is because the feature layers(all layers before classifier layer) , which are the basis of classifier layer, are identical, which prevents the classifier layer to perform better.
- I also tried retraining the feature layers and classifier layer in different learning rates. Like the former example, the loss also decreased fast. Theoretically this method will achieve a better performance. However , my best result is 91.5 % , which is a bit lower than that of the pretrained fine-tuned model (92.0%). I think it is because my training policy is not so proper.

In conclusion , pretrained model can reduce training time. It is said that it also helps to improve accuracy , whereas I didn't varify it in my experiments.

2.5 Experiment 4 : Scale jittering

In the original paper, I found an interesting method to prevent overfitting, called scale jittering. In general the method aims to individually rescale each training image by randomly sampling its size S from a certain range $[S_{min}, S_{max}]$, and then crop it into the initial size. I did some experiments about this method.

2.5.1 A failing try

At a very begin , I tried in this way: I updated train data after every 50 epoches and trained it in 200 epoches. However, beyond my expectation, the best val accuracy is achieved in the 50th epoch(88.10%). Here is its loss curve.

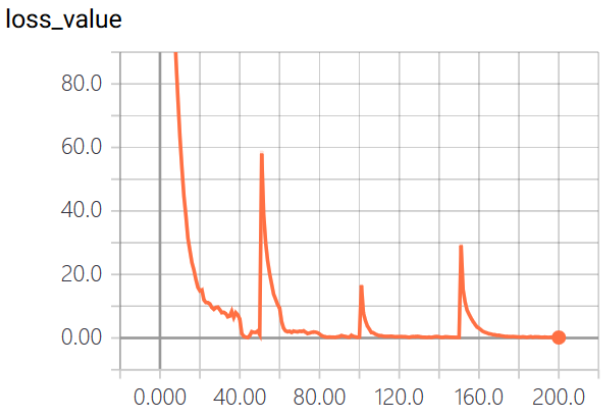


Figure 2: scale jittering loss curve

By analysising the figure, I have following conclusions:

- I noticed that after 74 epoch, the training accuracy has reached 0.999 , which means training data can be predict perfectly and overfitting still exist in this scenario.
- In 1th , 51th , 101th and 151th epoch, the loss is large. But in 50th , 100th , 150th and 200th epoch, the loss is quite small and overfitting exists. So a conclusion is drawn that in each 50 epoches , the network becomes overfitting towards training data in this period.
- Increasing the frequency of changing training data may helps.

2.5.2 Successful experiments

After consideration , I used following policy . I changed training data once an epoch, randomly sampling each training image in the range $[S_{min}, S_{max}]$. S_{min} is set to be 32, and S_{max} is 36 or 48. Besides , I ran an experiment without scale jittering for comparation, which is identical to $S_{max} = 32$. Here are their loss and val accuracy curves.

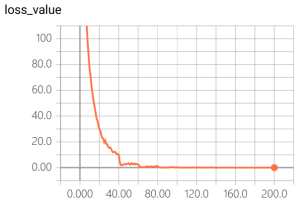


Figure 3: loss of $S_{max} = 32$

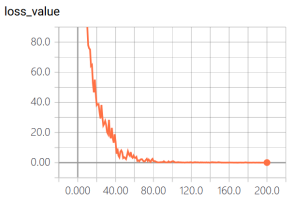


Figure 4: loss of $S_{max} = 36$

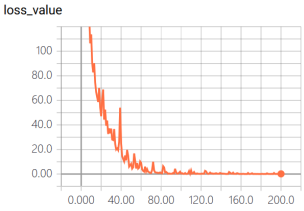
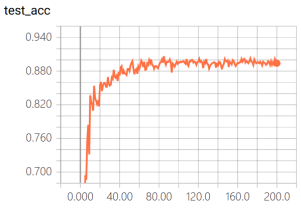
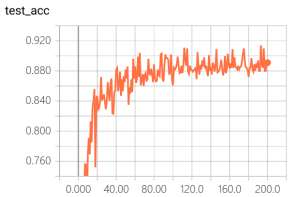


Figure 5: loss of $S_{max} = 48$





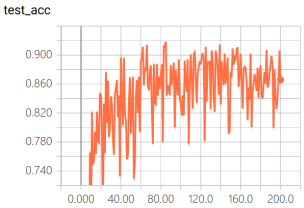


Figure 6: val acc,$S_{max} = 32$ Figure 7: val acc, $S_{max} = 36$ Figure 8: val acc, $S_{max} = 48$

As we can see , with bigger S_{max} , the loss curve and val accuracy curve will stir more acutely. Table 1 lists the result of these three methods and in which epoch number(≤ 200) the best val accuracy appears.

Table 1: Best val acc and when it appears

	best val acc	epoch num
$S_{max} = 32$	90.7%	84
$S_{max} = 36$	91.4 %	193
$S_{max} = 48$	91.9 %	85

I combined Figure 3 to Figure 8 and Table 1 , then got following conclusions:

- Scale jittering does help to prevent overfitting.
- Scale jittering is one method of data augmentation. It provides new data and adds noise to the network to make it adapt to more sophisticated data.
- Although scale jittering provides noise and makes the val accuracy curve less stable, it usually achieves a better result owing to the stir it brings.
- Different S_{max} I choose will lead network to achieve best val accuracy in different epoch numbers. So waiting with patience is necessary.

2.6 Experiment 5: Drop out

As we can see before, scale jittering does help prevent overfitting in some ways. However, due to the imperfect training policies , the best result of scale jittering(91.9%) is still lower than that we have obtained without scale jittering(92.0%). So I tried another way to prevent overfitting , which is drop out. I added a ReLU layer and a drop out layer in the classifier layers. By doing so , I got the best val accuracy during my whole project(92.4%).

3 Questions and Plans

- Training network is so time consuming that usually I will spend 3 to 4 hours to finish this process. Is there some ways to improve efficiency?
- As for plans , I still don't know is there any project waiting for me later. If any , I will finish it according to my time schedule.