Xi'an Jiaotong-Liverpool University
西交利物浦大学

<INT305 MACHINE LEARNING REPORT >

By

<Zizhe-Wang>
<1929413>

Project Report

<03-Dec-2022>

1. DESCRIBE THE CONVOLUTIONAL KERNEL AND THE LOSS FUNCTIONS USED IN THE CNN FRAMEWORK

1.1 THE CONVOLUTIONAL KERNEL

First, the discrete two-dimensional convolution formula is given:

$$g(x, y) = w(x, y) * f(x, y) = \sum_{s=-a}^{s=a} \sum_{t=-b}^{s=b} w(s, t) f(x - s, y - t)$$

In the formula, f (x, y) represents a grayscale source image, and w (x, y) is a convolution kernel. The above formula has a feature that the sum of subscripts of the two corresponding variables for multiplication is (x, y), which aims to impose a constraint on this weighted sum. However, in the calculation of image processing, the convolution kernel is generally rotated by 180 degrees before calculation, which is to let us see the relationship between the two matrices more clearly. The advantage of this is that it is easy to popularize and understand its physical meaning. In practice, the matrix after flipping is used to calculate the inner product of the matrix directly.

The convolution kernel in CNN framework is used to complete the feature extraction of images. To convolve the image, the input is the pixel value of an area on the image, which is a matrix. Convolution kernel is a matrix used to describe a feature, whose size is exactly the same as the input matrix. Convolution is the multiplication and accumulation of each corresponding pixel value of two matrices.

Each convolution kernel has three dimensions: length, width and depth. The length and width of the convolution kernel are artificially assigned, and length * width is also called the size of the convolution kernel. The depth of the convolution kernel is the same as that of the current image, that is, the number of feature maps. In other words, in a certain convolution layer, the number of feature maps required for the next layer is the same as that of the current layer.

1.2 LOSS FUNCTION

In CNN, the loss function is used to calculate the deviation between the output result of CNN and the label result, and then used in the back propagation process to update the gradient. By continuously training and optimizing the parameters in CNN, the goal is to minimize the loss function and finally learn the best CNN model The CNN framework code of the project case uses cross entropy as the loss function.

Entropy is the theoretical minimum average coding length of events subject to a specific probability distribution. As long as we know the probability distribution of any event, we can calculate its entropy. However, if we do not know the probability distribution of events and want to calculate entropy, we need to estimate entropy. The process of entropy estimation leads to cross entropy. In CNN, by comparing the prediction results of the model with the real labels of the data, it can be concluded that as the prediction becomes more accurate, the value of the cross entropy becomes smaller and smaller. If the prediction is completely correct, the value of the cross entropy will be 0. Therefore, the cross entropy is used as the loss function when training the classification model.

1. Binary Classification expression:

$$\mathrm{L} = -[\mathrm{y} * \log(\mathrm{p}) + (1 - \mathrm{y}) * \log(1 - \mathrm{p})]$$

Note: y is the sample label, correct is 1, error is 0. p is the prediction accuracy.

2. Multi-Class Classification expression:

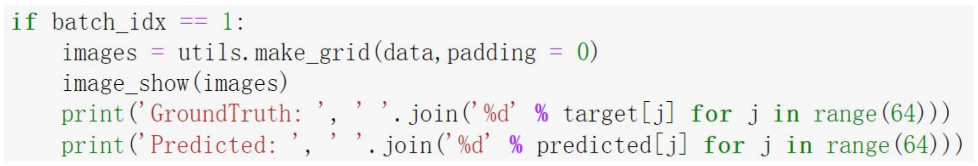
$$\mathrm{L} = -\sum_{\mathrm{c}=1}^{\mathrm{M}} \mathrm{y}_{\mathrm{c}} \log(\mathrm{p}_{\mathrm{c}})$$

Note: M is the number of categories, y_{c} is the sample label, and p_{c} is the prediction accuracy.

2. TRAIN(FINE-TUNE) AND TEST

2.1 TRAIN AND TEST

In order to make the training results more intuitive, I added the following code in the train function to display the real value and predicted value of one of the batch datasetss of each epoch in the train datasetss, and also the data itself.



```
if batch_idx == 1:
    images = utils.make_grid(data,padding = 0)
    image_show(images)
    print('GroundTruth: ', ' '.join('%d' % target[j] for j in range(64)))
    print('Predicted: ', ' '.join('%d' % predicted[j] for j in range(64)))
```

Fig1. code of printing a batch of data

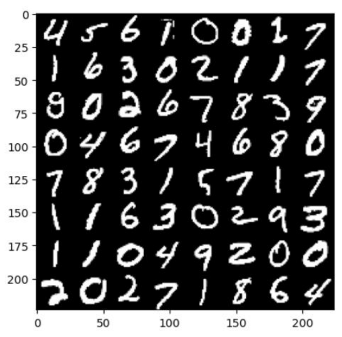
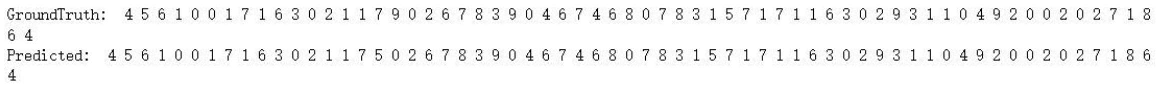
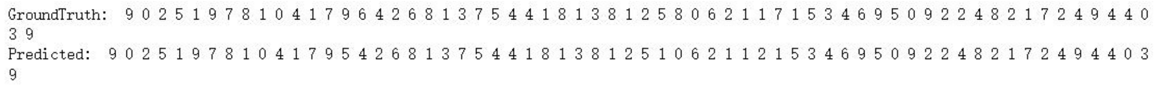


Fig2. a batch of train datasetss



```
GroundTruth:  4 5 6 1 0 0 1 7 1 6 3 0 2 1 1 7 9 0 2 6 7 8 3 9 0 4 6 7 4 6 8 0 7 8 3 1 5 7 1 7 1 1 6 3 0 2 9 3 1 1 0 4 9 2 0 0 2 0 2 7 1 8
6 4
Predicted:  4 5 6 1 0 0 1 7 1 6 3 0 2 1 1 7 5 0 2 6 7 8 3 9 0 4 6 7 4 6 8 0 7 8 3 1 5 7 1 7 1 1 6 3 0 2 9 3 1 1 0 4 9 2 0 0 2 0 2 7 1 8 6
4
```

Fig3. Ground Truth and Predict value in train datasetss(epoch = 2)



```
GroundTruth:  9 0 2 5 1 9 7 8 1 0 4 1 7 9 6 4 2 6 8 1 3 7 5 4 4 1 8 1 3 8 1 2 5 8 0 6 2 1 1 7 1 5 3 4 6 9 5 0 9 2 2 4 8 2 1 7 2 4 9 4 4 0
3 9
Predicted:  9 0 2 5 1 9 7 8 1 0 4 1 7 9 5 4 2 6 8 1 3 7 5 4 4 1 8 1 3 8 1 2 5 1 0 6 2 1 1 2 1 5 3 4 6 9 5 0 9 2 2 4 8 2 1 7 2 4 9 4 4 0 3
9
```

Fig4. Ground Truth and Predict value in test datasetss(epoch = 2)

We can see that in the second epoch, the accuracy of the train datasetss is already considerable. The default value of epoch is 14. There is the code to draw a line graph of accuracy to .

```
plt.plot(epochs,train_accs,color='r',label='train_acc')
plt.plot(epochs,test_accs,color='b',label='test_acc')
plt.xlabel('epochs')
plt.ylabel('accuracy')
plt.title("change of train(test) accuracy")
plt.legend()
plt.savefig('test.jpg')
plt.show()
```

Fig1. code of drawing line graph

At the end time of these epoch, the accuracy performance of the train datasetss is about 98.5%.

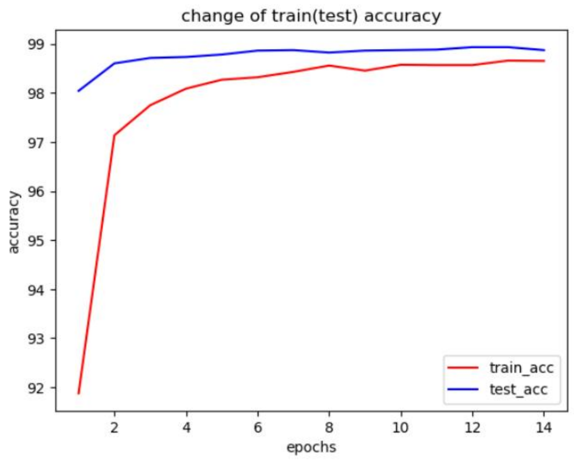
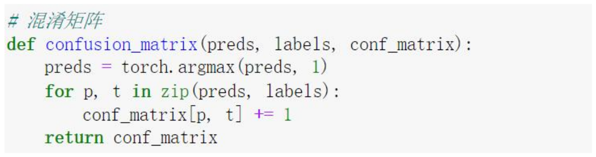


Fig5. Line graph of train accuracy and test accuracy

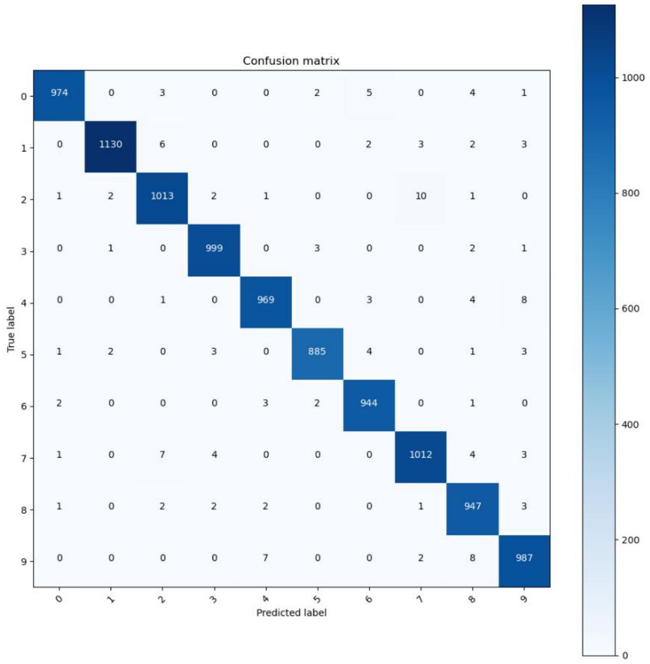
Then, draw a confusion matrix to see the distribution of correct and incorrect classifications.

```
def plot_confusion_matrix(cm, classes, normalize=False, title='Confusion matrix', cmap=plt.cm.Blues):
    if normalize:
        cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
        print("Normalized confusion matrix")
    else:
        print('Confusion matrix, without normalization')


    print(cm)
    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title(title)
    plt.colorbar()
    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes, rotation=45)
    plt.yticks(tick_marks, classes)


    fmt = '.2f' if normalize else '.0f'
    thresh = cm.max() / 2.
    for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
        plt.text(j, i, format(cm[i, j], fmt)+"%", horizontalalignment="center", color="white" if cm[i, j] > thresh else "black")


    plt.tight_layout()
    plt.ylabel('True label')
    plt.xlabel('Predicted label')
```



```python
# 混淆矩阵
def confusion_matrix(preds, labels, conf_matrix):
    preds = torch.argmax(preds, 1)
    for p, t in zip(preds, labels):
        conf_matrix[p, t] += 1
    return conf_matrix
```

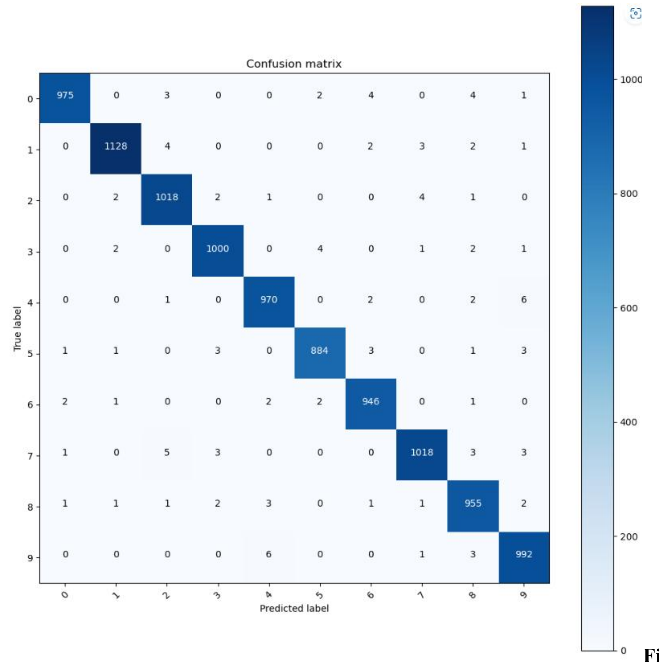
Fig6. Code of confusion matrix



Confusion matrix

True label \ Predicted label	0	1	2	3	4	5	6	7	8	9
0	974	0	3	0	0	2	5	0	4	1
1	0	1130	6	0	0	0	2	3	2	3
2	1	2	1013	2	1	0	0	10	1	0
3	0	1	0	999	0	3	0	0	2	1
4	0	0	1	0	969	0	3	0	4	8
5	1	2	0	3	0	885	4	0	1	3
6	2	0	0	0	3	2	944	0	1	0
7	1	0	7	4	0	0	0	1012	4	3
8	1	0	2	2	2	0	0	1	947	3
9	0	0	0	0	7	0	0	2	8	987

Fig7. Confusion matrix in epoch3

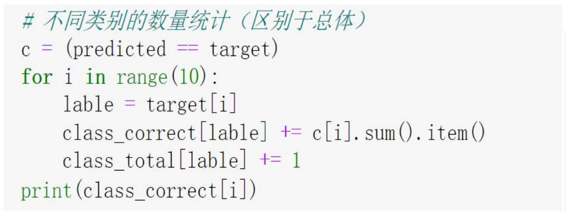


Confusion matrix

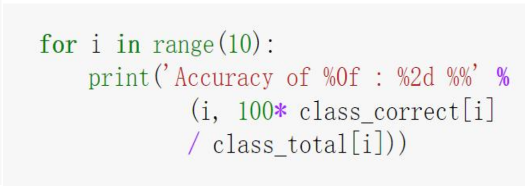
True label \ Predicted label	0	1	2	3	4	5	6	7	8	9
0	975	0	3	0	0	2	4	0	4	1
1	0	1128	4	0	0	0	2	3	2	1
2	0	2	1018	2	1	0	0	4	1	0
3	0	2	0	1000	0	4	0	1	2	1
4	0	0	1	0	970	0	2	0	2	6
5	1	1	0	3	0	884	3	0	1	3
6	2	1	0	0	2	2	946	0	1	0
7	1	0	5	3	0	0	0	1018	3	3
8	1	1	1	2	3	0	1	1	955	2
9	0	0	0	0	6	0	0	1	3	992

Fig8. Confusion matrix in epoch 7

Then, classification confidence value:

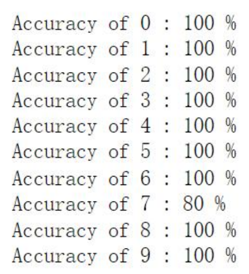


```
# 不同类别的数量统计（区别于总体）
c = (predicted == target)
for i in range(10):
    lable = target[i]
    class_correct[lable] += c[i].sum().item()
    class_total[lable] += 1
print(class_correct[i])
```

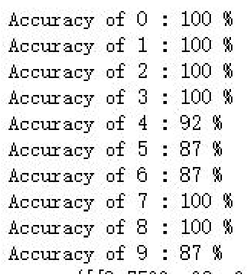


```
for i in range(10):
    print('Accuracy of %0f : %2d %%' %
          (i, 100* class_correct[i]
          / class_total[i]))
```

Fig9-1. Code for classification confidence value



```
Accuracy of 0 : 100 %
Accuracy of 1 : 100 %
Accuracy of 2 : 100 %
Accuracy of 3 : 100 %
Accuracy of 4 : 100 %
Accuracy of 5 : 100 %
Accuracy of 6 : 100 %
Accuracy of 7 : 80 %
Accuracy of 8 : 100 %
Accuracy of 9 : 100 %
```



```
Accuracy of 0 : 100 %
Accuracy of 1 : 100 %
Accuracy of 2 : 100 %
Accuracy of 3 : 100 %
Accuracy of 4 : 92 %
Accuracy of 5 : 87 %
Accuracy of 6 : 87 %
Accuracy of 7 : 100 %
Accuracy of 8 : 100 %
Accuracy of 9 : 87 %
```

Fig9-2. Classification confidence value（in CPU） Classification confidence value（in GPU）

Note: The precision of floating point number calculation on the GPU side is different from that on the CPU side. GPU defaults to float precision and CPU defaults to double precision. This means that if the result of floating point multiplication and division is accumulated in GPU, there must be errors

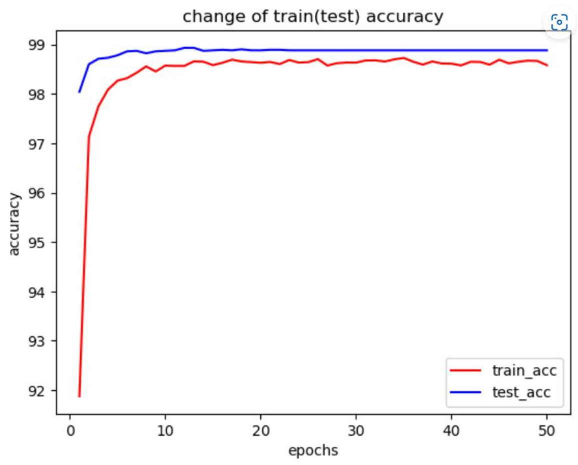
2.2 FINE-TUNE

The process of training with others' parameters, modified network and own data to make the parameters adapt to their own data is usually called fine-tuning. By training small data sets we have (i.e. back propagation), we can fine tune the existing networks. These networks are trained on large data sets like ImageNet to achieve the effect of fast tF that the context of our datasets is not very different from that of the original datasets (such as ImageNet), the pre-trained model will have learned the features related to our own classification problem. The usual way to fine tune is as follows: truncate the last layer (softmax layer) of the pre-trained network and replace it with a new softmax layer related to our own problem. For example, the pre-trained network on MNIST has 1000 categories of softmax layers, and our task is to classify ten categories, then the new softmax layer of the network will be composed of 10 categories instead of 1000 categories. Then, we run the pre-trained weights on the network. Then use a smaller learning rate to train the network, we do not want to quickly distort them too much. The general approach is to make the initial learning rate 10 times smaller than that of Training from scratch and consider changing the learning rate scheduler. Finally, try to adjust the batch size, adjust the epochs of the training set, and output some visual results.

In this project, since the datasets can get good results by using the bare model to run and adjust hyperparameter, the basic fine-tuning is used, that is, simply adjusting the hyperparameter. The remaining methods are not fully implemented due to time and visual results are output. They will be attached to the job in code form (fine_tune.ipynb) for reference only:

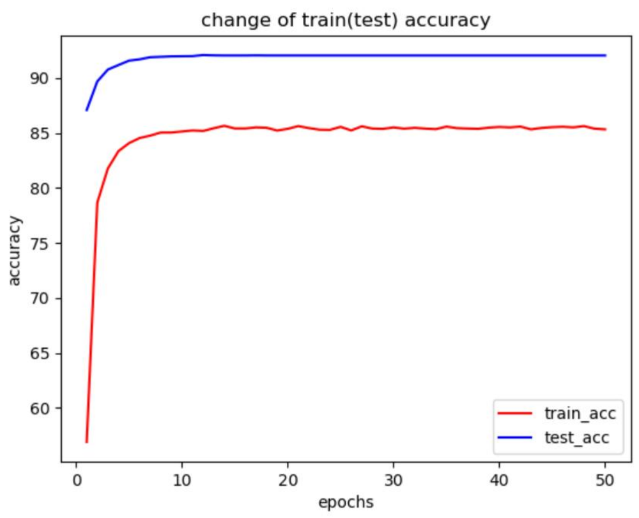
1. Epoch times (50)

Since epoch=14, the accuracy rate has entered a stable state, but there are still some ups and downs. When epoch=22, the accuracy rate of the training set starts to remain unchanged, and the final accuracy rate has not significantly improved, so consider whether the learning rate has been increased.



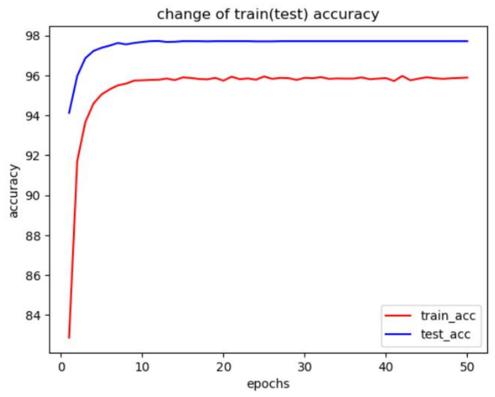
2. Learning rate(0.1)

When the learning rate becomes 0.1, it is obvious that the network converges very slowly, increasing the time to find the optimal value. In addition, it converges when it finally enters the local extreme point, and there is no really found optimal solution.

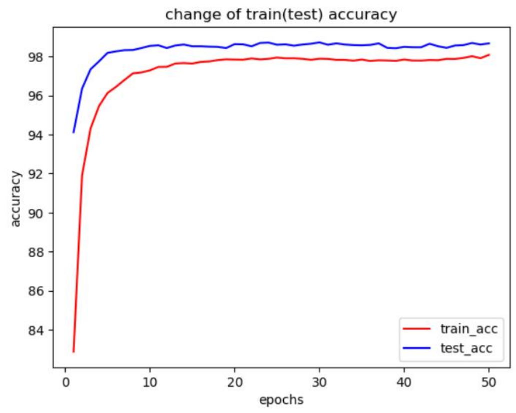


3. Learning rate decline strategy(scheduler)

Although the learning rate of 0.1 is very bad, we can try to change the learning rate decline strategy to see whether selecting the appropriate scheduler in this case can bring better improvement. Choose two to see the effect:



ExponentialLR



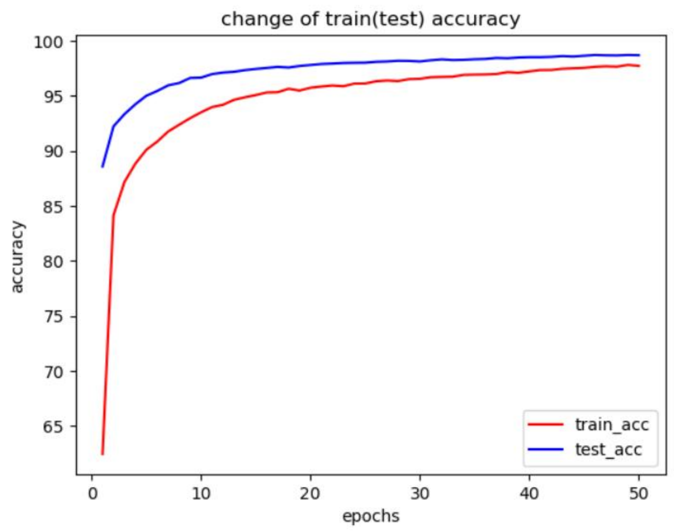
CosineAnnealingLR

It is obvious that CosineAnnealingLR is better than ExponentialLR.

4. Batch size

We use small batch because it tends to converge faster, because it does not need to completely traverse training data to update weights,. However, the random gradient descent is continuous, and it is not easy to parallelize if small batches are used. Using a larger batch size allows us to perform parallel computing to a greater extent, because we can split training examples between different work nodes, which in turn can significantly speed up model training [1]. Although large batch size can achieve training error similar to small batch size, it often has worse generalization effect on test data.

It can be clearly seen that when the batch size is 64, there is no change from epoch to 22, but when the batch size is 640, the accuracy rate of epoch is still slightly increased when the batch size is 50.



3. OTHER METHOD

The fine-tuning in the second part is actually to improve the classification performance, but it is more focus on the adjustment of super parameters, such as epoch, learning rate (the fine-tuning focuses on the replacement of the scheduler by the same optimizer and the initial learning rate value adjustment), batch size, and so on. This part will focus on improving classification performance or reducing model size by changing the structure or module of the model, such as changing the depth and width of the network layer, the selection of optimizer, Convolutional design, etc.

1. The depth and width of the network layer

Because GPUs are processed in parallel, increasing the width is more friendly to GPUs than increasing the depth. Many studies also show that widening the network is easier to train than deepening the network. The width enables each layer to learn more features. The shallow features are very important, so the width of the shallow network is a very sensitive coefficient. The computation amount brought by the width increases by the order of squares. At the same time, the benefits brought by the width are subject to the marginal effect (the greater the width of each layer is, the less the model performance will be improved by increasing the layer width). However, according to the data, the layer width of the excellent models proposed by the academia is gradually decreasing.

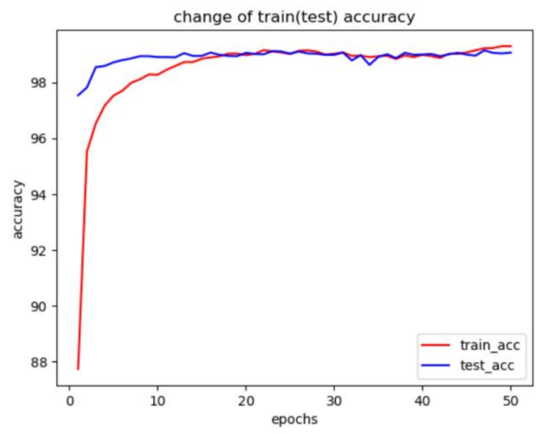
2012 $\longrightarrow$ 2017						
name	AlexNet	GoogleNet	VGG16	ResNet18	SqueezeNet1.0	MobileNet1.0
channel (first layer)	96	64	64	64	64	32

One of the great benefits of the deeper network is the abstraction layer by layer and the continuous refinement and extraction of knowledge. For example, the first layer learns the edge, the second layer learns the simple shape, the third layer begins to learn the shape of the target, and the deeper network layer can learn more complex expressions. Choosing the appropriate network depth is also one of the important ideas of design. Generally, adding more layers will improve the accuracy, while sacrificing some speed and memory. However, it is also subject to the marginal effect (the more layers are added, the less accuracy will be improved by adding each layer). (the code for this kind of CNN should refer to the fine-tune.ipynb)

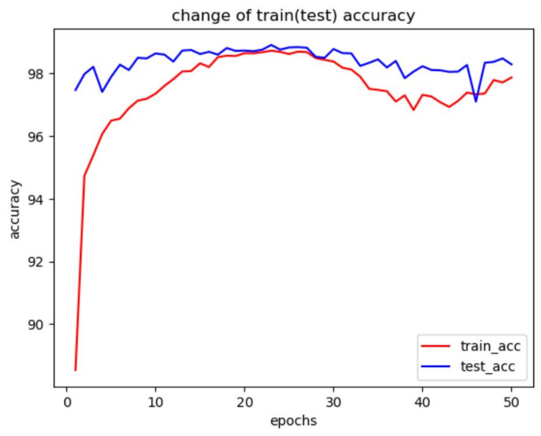
2. The selection of optimizer (CosineAnnealingLR, initial lr = 1.0, batch size = 64, epoch = 50)

Optimizer selection idea: keep trying and making mistakes from easy to difficult. We start with the simplest SGD optimizer:

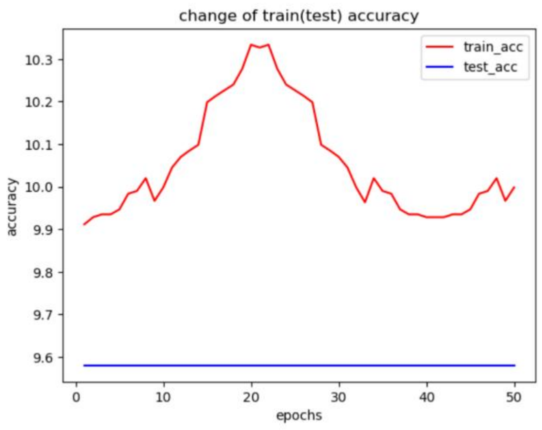
The optimizer can be used in combination. In the first half of the training, use a fast optimizer to set a lower learning efficiency. In the second half of the training, select a slower optimizer to combine the optimizer.



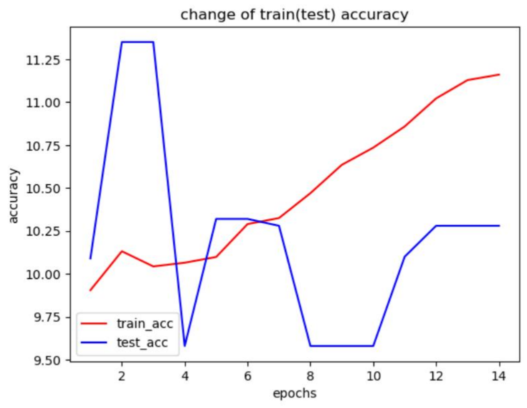
SGD



SGDM(Momentum)

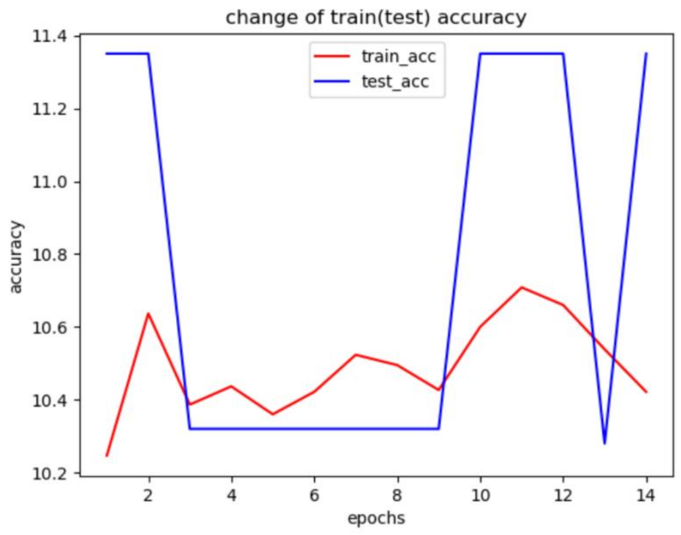


RMSprop



Adam

SGD is the most common optimizer, which can also be said to have no acceleration effect. Momentum is an improved version of SGD, which adds the Momentum principle, and RMSprop is an upgraded version of Momentum Adam is an upgraded version of RMSprop. However, the accuracy of visualization results is very low and the vibration is very severe, which may caused by batch size or learning rate.



Adam(epoch=14, batch_size = 640)

I have adjusted the initial value of the batch size and learning rate for many times, but it still fails to converge. Later, I went to query the papers and found that more than one paper pointed out that Adam had obvious problems in convergence, and no matter how the parameters were adjusted, sometimes it was impossible to achieve the goal. However, some scholars have proposed the AdamW structure, which can solve this problem to some extent. At present, the fastest way to train the neural network is to use the AdamW optimization algorithm+super convergence.

3. Convolutional design

When viewing the paper, I found that the performance can be improved through well-designed convolutions: MobileNets uses deeply separated convolutions to greatly reduce the consumption of computing and memory, while only sacrificing a very small accuracy rate (the degree of sacrifice can be adjusted as needed) [2]. ShuffleNet uses point group convolutions and channel randomization to greatly reduce computing costs, while maintaining a higher accuracy rate than MobileNets [3]. XNOR Net uses binary convolution [4], that is, only two possible values are involved: 0 or 1. This design enables the network to have a high degree of sparsity, and is easy to be compressed without consuming too much memory.

Reference:

[1] N. S. Keskar, D. Mudigere, J. Nocedal, M. Smelyanskiy, and P. T. P. Tang, "On large-batch training for Deep Learning: Generalization gap and sharp minima," *arXiv.org*, 09-Feb-2017. [Online]. Available: https://arxiv.org/abs/1609.04836. [Accessed: 03-Dec-2022].

[2] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "MobileNetV2: Inverted residuals and linear bottlenecks," *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2018.

[3] X. Zhang, X. Zhou, M. Lin, and J. Sun, "ShuffleNet: An extremely efficient convolutional neural network for mobile devices," *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2018.

[4] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "XNOR-net: ImageNet classification using binary convolutional Neural Networks," *Computer Vision – ECCV 2016*, pp. 525–542, 2016.