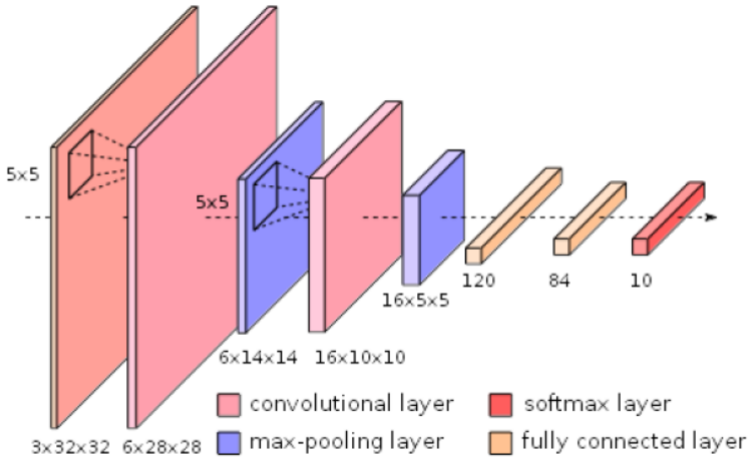
EE569 HW5 CNN Training on LeNet-5

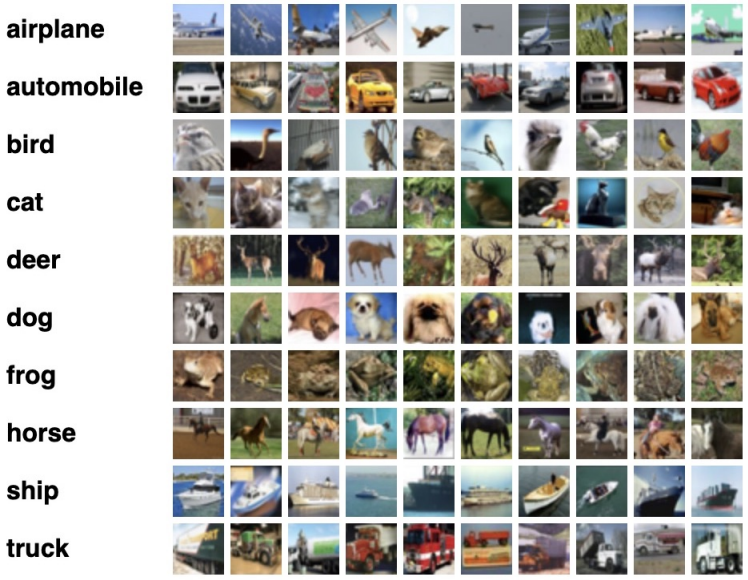
Introduction

Convolutional neural network is popular today for image and speech recognition, and LeNet-5 is the earliest ancestor. It is designed by Lecun and others in 1998. The architecture is shown below. As one of the earliest convolutional neural networks, it promotes the development of deep learning, and defined the basic components in CNN. LeNet-5's most classical application is to recognize hand-written numbers (MNIST). In this assignment, apart from MNIST, we also apply it to FashionMNIST and CIFAR-10 datasets, and see how good it will perform.



MNIST dataset

Fashion MNIST dataset



CIFAR 10 dataset

Procedure

Compare classification performance on different datasets

I use different configuration of LeNet-5, i.e. different learning rate, optimizer, regularization and so on, to find a relatively good outcome on each dataset. Generate learning curve.

Analysis on confusion classes and hard samples

With the best models on previous steps, I generate confusion matrix on testing set of every dataset, and see how and why our models make mistakes.

Classification with noisy data

Data in real world application could be noisy with wrong labels. In this part we assign wrong label to the MNIST training set with certain probability, and see how symmetric label noise will affect our model

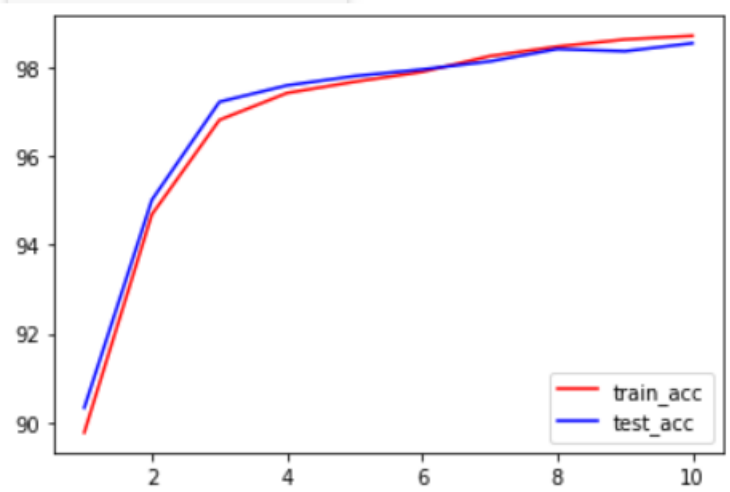
All the computation are done on Google Colab online. But cuda is not available there so no GPU acceleration is used.

Results

Classification performance

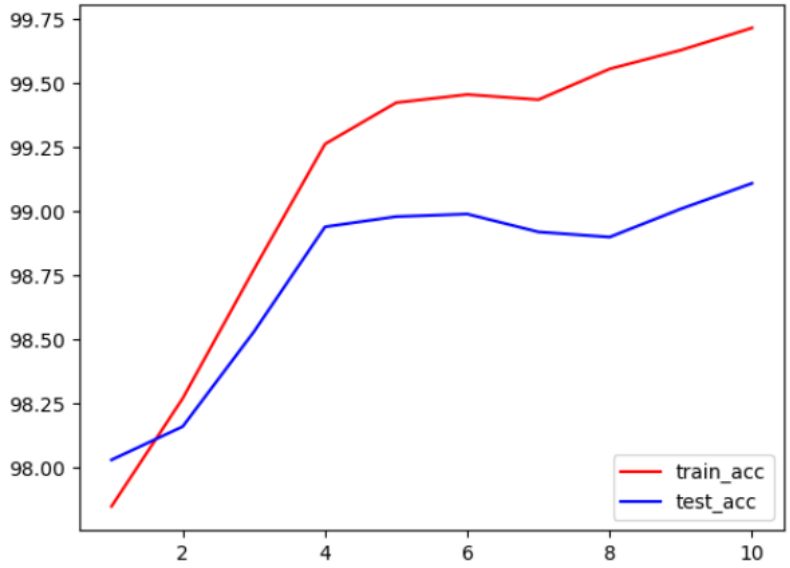
MNIST

- learning rate= 0.001, optimizer=SGD, momentum=0.9, epoch=10



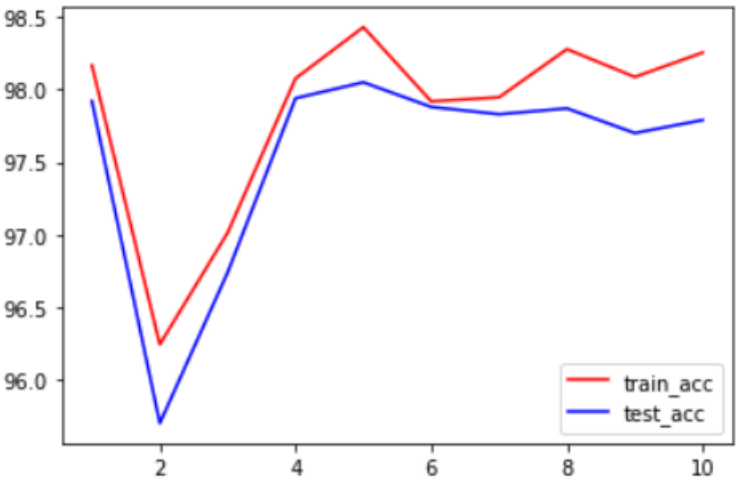
train_acc 98.707, test_acc 98.540

- learning rate=0.01, optimizer=SGD, momentum=0.9, epoch=10



train_acc 99.717, test_acc 99.110

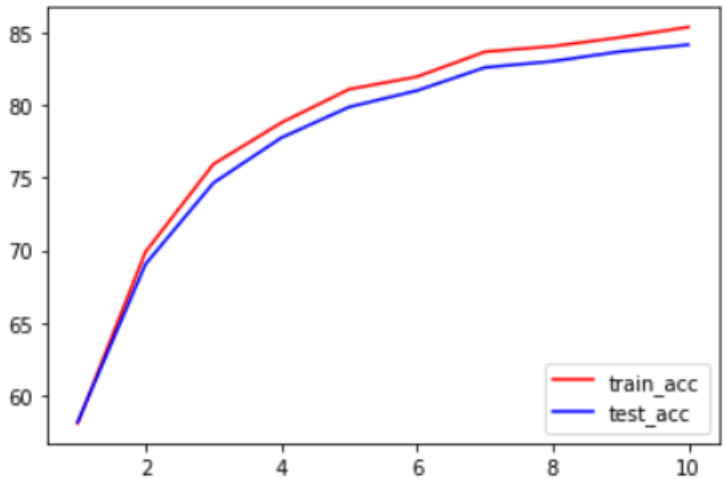
- learning rate=0.01, optimizer=Adam, momentum=0.9, epoch=10



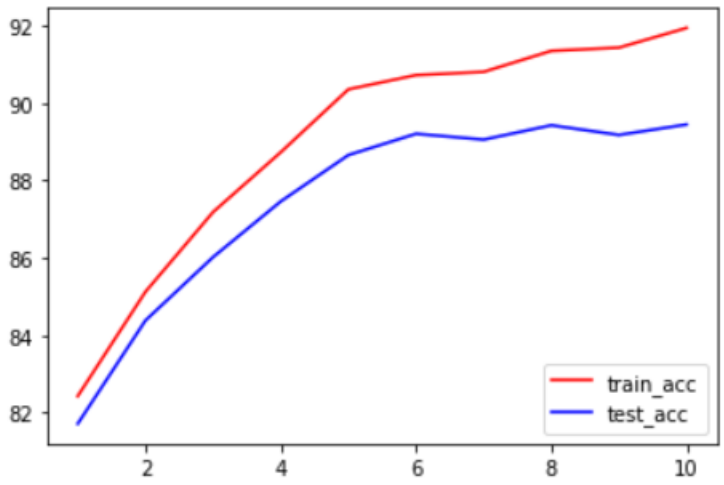
train_acc 98.255, test_acc 97.790

Fashion MNIST

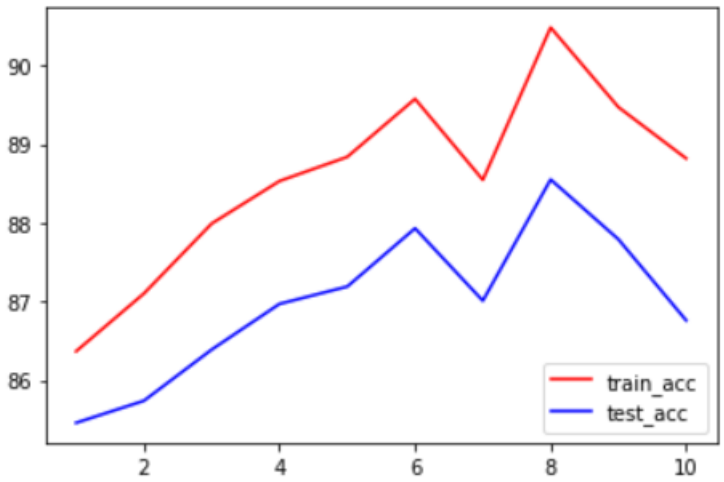
- learning rate= 0.001, optimizer=SGD, momentum=0.9, epoch=10



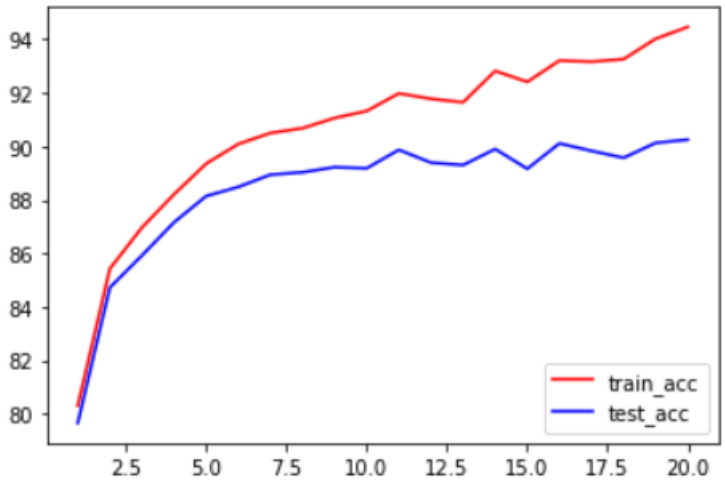
- learning rate=0.01, optimizer=SGD, momentum=0.9, epoch=10



- learning rate=0.01, optimizer=Adam, momentum=0.9, epoch=10

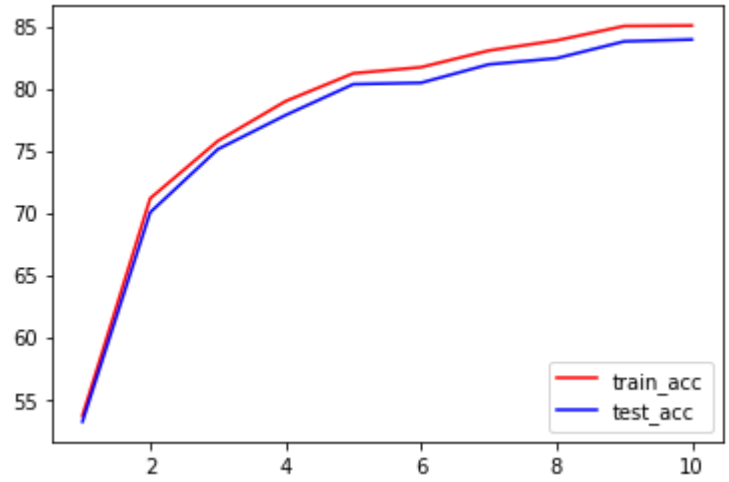


learning rate=0.01, optimizer=SGD, momentum=0.9, epoch=20



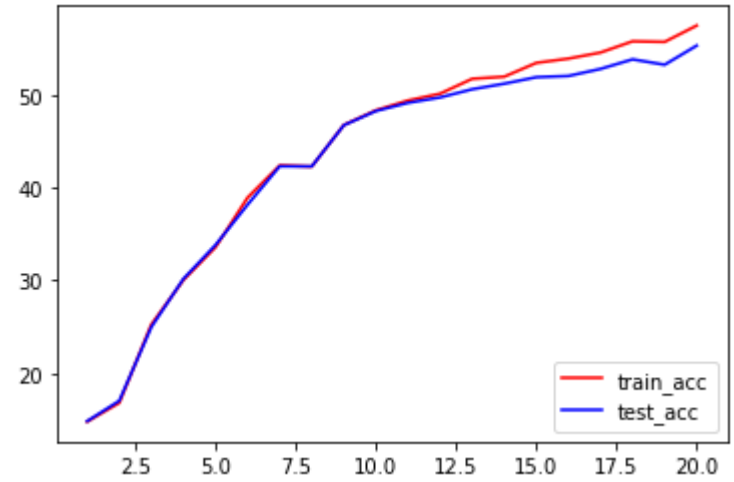
train_acc 94.465, test_acc 90.250

- learning rate= 0.001, optimizer=SGD, momentum=0.9, epoch=10, dropout=0.2, L2 regularization

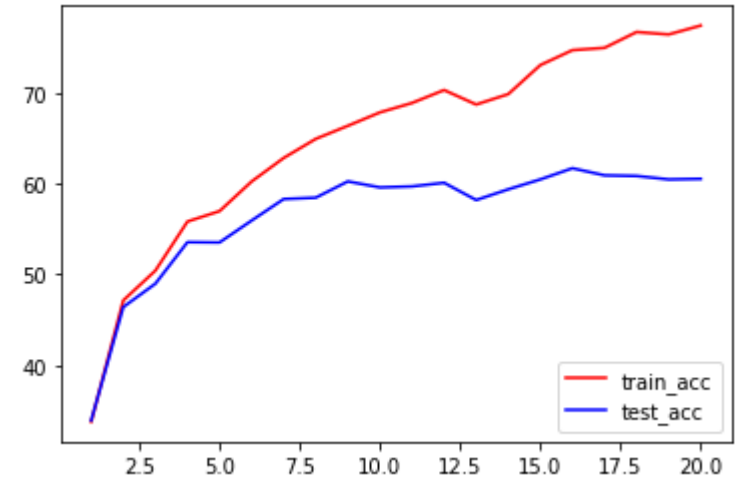


CIFAR

- learning rate= 0.001, optimizer=SGD, momentum=0.9, epoch=20

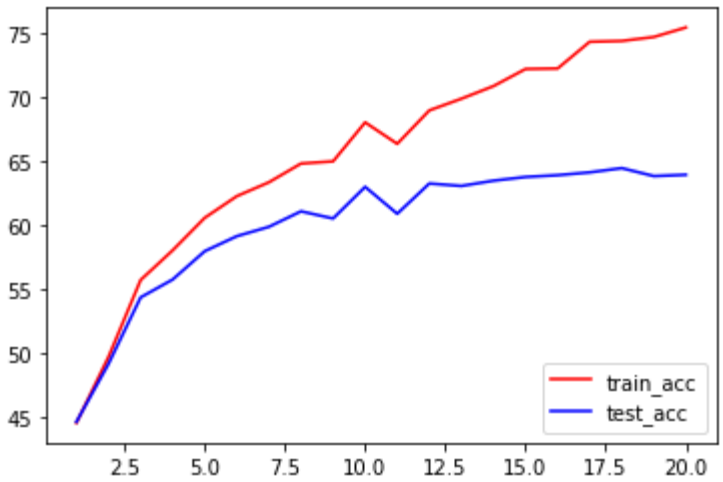


- learning rate= 0.01, optimizer=SGD, momentum=0.9, epoch=20



epoch 20: train_acc 77.404, test_acc 60.530

- learning rate= 0.001, optimizer=Adam, epoch=20, L2 regularization

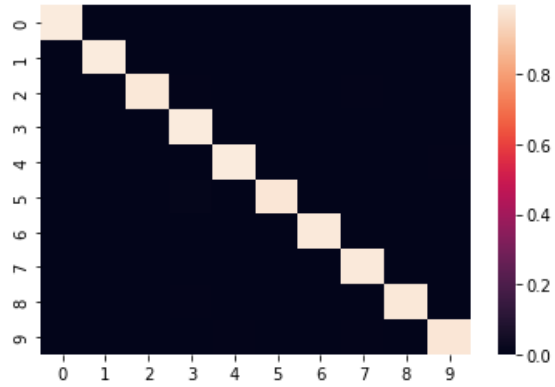


epoch 20: train_acc 75.378, test_acc 63.880

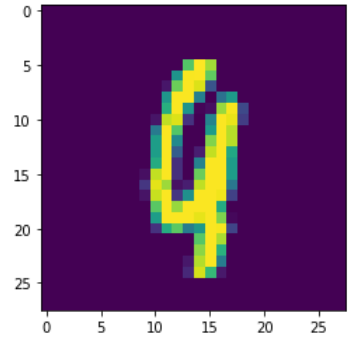
Confusion Matrix

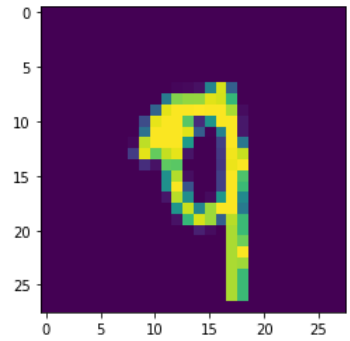
MNIST

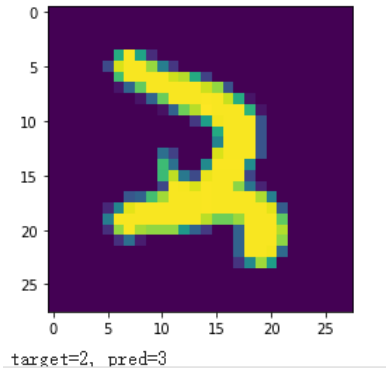
```
[[0.997 0.001 0.    0.    0.    0.    0.001 0.    0.    0.001]
 [0.    0.997 0.001 0.001 0.    0.    0.    0.    0.    0.001]
 [0.003 0.    0.987 0.005 0.    0.    0.    0.004 0.001 0.   ]
 [0.    0.    0.001 0.995 0.    0.003 0.    0.    0.    0.001]
 [0.    0.001 0.    0.    0.995 0.    0.    0.    0.    0.004]
 [0.001 0.    0.    0.009 0.    0.983 0.002 0.    0.003 0.001]
 [0.003 0.003 0.    0.    0.002 0.001 0.99  0.    0.001 0.   ]
 [0.    0.002 0.003 0.001 0.    0.    0.    0.991 0.002 0.001]
 [0.002 0.001 0.001 0.004 0.001 0.001 0.001 0.    0.988 0.001]
 [0.002 0.001 0.    0.001 0.006 0.001 0.    0.005 0.001 0.983]]
```



some mistakes



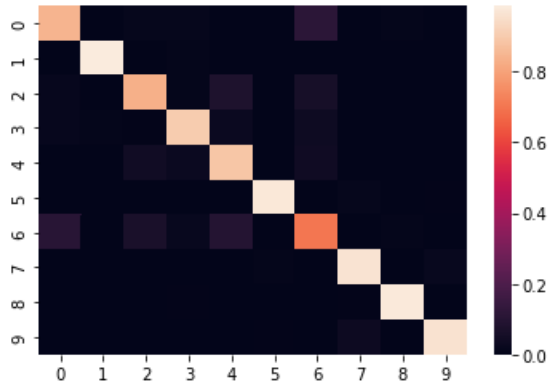




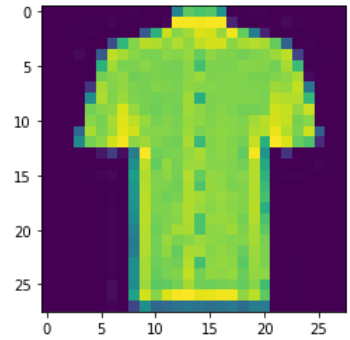
Fashion MNIST

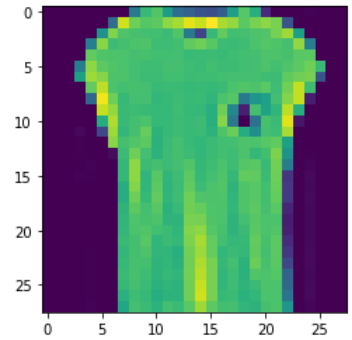
- 0: T-shirt/top
- 1: Trouser
- 2: Pullover
- 3: Dress
- 4: Coat
- 5: Sandal
- 6: Shirt
- 7: Sneaker
- 8: Bag
- 9: Ankle boot

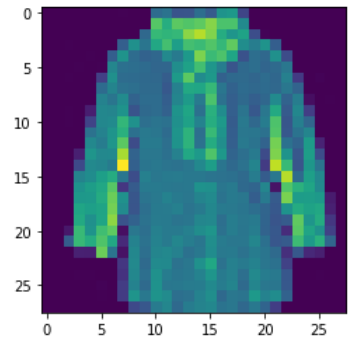
```
[[0.845 0.    0.014 0.013 0.003 0.003 0.112 0.    0.01  0.   ]
 [0.    0.986 0.    0.008 0.002 0.    0.003 0.    0.001 0.   ]
 [0.016 0.001 0.835 0.01  0.078 0.    0.059 0.    0.001 0.   ]
 [0.018 0.009 0.006 0.902 0.028 0.    0.035 0.    0.002 0.   ]
 [0.001 0.    0.045 0.027 0.887 0.    0.039 0.    0.001 0.   ]
 [0.    0.    0.    0.    0.    0.978 0.    0.015 0.    0.007]
 [0.105 0.    0.066 0.023 0.095 0.001 0.701 0.    0.009 0.   ]
 [0.    0.    0.    0.    0.    0.01  0.    0.965 0.002 0.023]
 [0.003 0.002 0.003 0.005 0.002 0.001 0.    0.004 0.98  0.   ]
 [0.    0.    0.    0.    0.    0.007 0.    0.032 0.    0.961]]
```



some mistakes

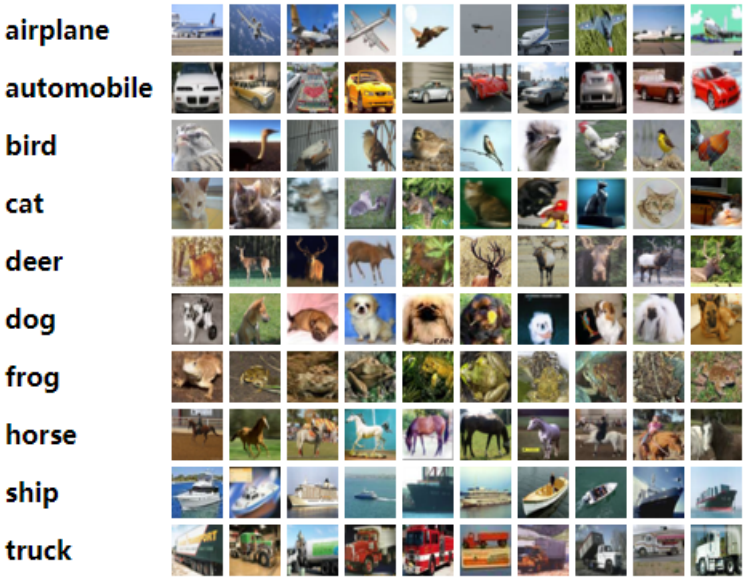




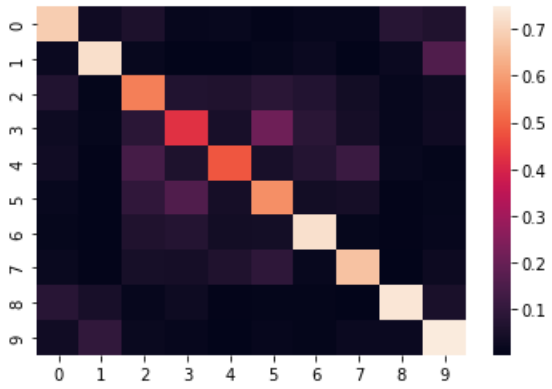


*here target and predict index start from 0. i.e. T-shirt is 0

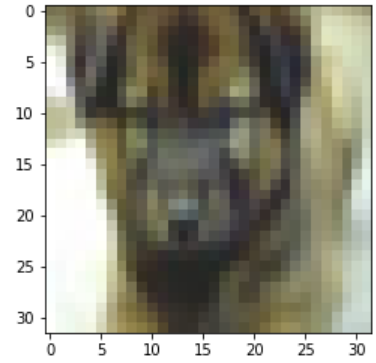
CIFAR10



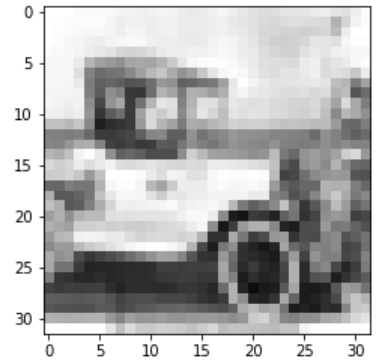
```
[[0.687 0.034 0.059 0.017 0.018 0.008 0.015 0.016 0.081 0.065]
 [0.026 0.721 0.02  0.005 0.003 0.014 0.024 0.005 0.024 0.158]
 [0.067 0.009 0.548 0.068 0.065 0.087 0.07  0.039 0.017 0.03 ]
 [0.031 0.013 0.085 0.426 0.05  0.215 0.085 0.046 0.015 0.034]
 [0.036 0.008 0.137 0.064 0.484 0.051 0.075 0.118 0.018 0.009]
 [0.02  0.003 0.098 0.158 0.045 0.573 0.038 0.045 0.008 0.012]
 [0.014 0.005 0.065 0.075 0.039 0.043 0.724 0.012 0.007 0.016]
 [0.022 0.007 0.048 0.045 0.065 0.094 0.019 0.668 0.003 0.029]
 [0.083 0.05  0.017 0.03  0.008 0.009 0.009 0.005 0.736 0.053]
 [0.035 0.1   0.022 0.015 0.004 0.016 0.011 0.026 0.024 0.747]]
```



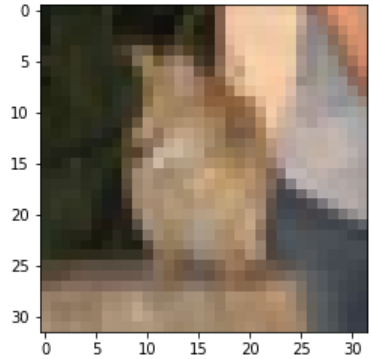
some mistakes



target=5, pred=3



target=1, pred=9



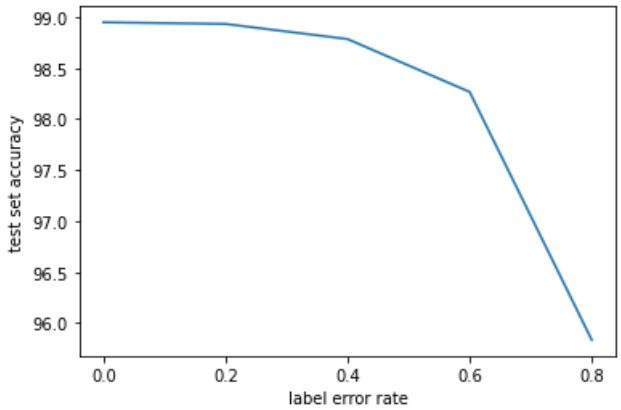
target=3, pred=5

Noisy Label

confusion matrix for $\epsilon = 0.4$

```
[[0.6   0.044 0.044 0.044 0.044 0.044 0.044 0.044 0.044 0.044]
 [0.044 0.6   0.044 0.044 0.044 0.044 0.044 0.044 0.044 0.044]
 [0.044 0.044 0.6   0.044 0.044 0.044 0.044 0.044 0.044 0.044]
 [0.044 0.044 0.044 0.6   0.044 0.044 0.044 0.044 0.044 0.044]
 [0.044 0.044 0.044 0.044 0.6   0.044 0.044 0.044 0.044 0.044]
 [0.044 0.044 0.044 0.044 0.044 0.6   0.044 0.044 0.044 0.044]
 [0.044 0.044 0.044 0.044 0.044 0.044 0.6   0.044 0.044 0.044]
 [0.044 0.044 0.044 0.044 0.044 0.044 0.044 0.6   0.044 0.044]
 [0.044 0.044 0.044 0.044 0.044 0.044 0.044 0.044 0.6   0.044]
 [0.044 0.044 0.044 0.044 0.044 0.044 0.044 0.044 0.044 0.6  ]]
```

Under specific error rate, the model was trained 5 times independently, and get 5 test set accuracy. 5 different error rate are used. The mean of accuracy vs. label error rate are plotted in the line chart.



The standard deviation of the 5 test set accuracy under each error rate are:

0.097%, 0.056%, 0.064%, 0.273% 0.743%

Discussion

CNN components

1. Convolutional layer: convolutional layer is composed of some tunable filter. When the filter detect some features, for example, texture, or objects, they'll be activated. The output of convolutional layer is the features extracted. It greatly reduce the workload comparing with inputting every pixel into the FC layer.
2. Pooling layer: always added between two consecutive convolutional layer. The main purpose is to compress information. For example, in max pooling, we may only retain the local maximum in a 4*4 block. By doing so, we can keep the most significant features while remove most redundant information.
3. FC layer: fully connected layer. Always the last component of the network. It maps the implicit feature space obtained by convolution layer and pooling layer into the label space, just like a classifier. One of the main computational work is to tune the millions of parameter in fc layer during backpropogation.
4. Activation function: it can appears anywhere in the neural networl. the major purpose is to add non-linearity. If there's no activation function, then every layer is linear and the whole model will be linear model. By adding activation functions, we can then approximate any curve.
5. Softmax function: always in the output layer. it convert a vector of numbers into a vector of probability. Each of them equals to exponetial of the original number at that position divided by sum of exponential of every element. The vector of probability represents how likely the input belongs to each class, and of course, it sums to 1. Another benefit of using softmax is, when collaborate with cross-entropy, the partial derivative of the loss function to each element is just $\hat{y} - y$, where y is one-hot encoded.

Overfitting issue

Overfitting means the model approximate the training data too much. It has small bias while variance will be large. It performs well on training set, but cannot generalize well.

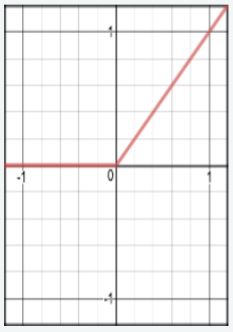
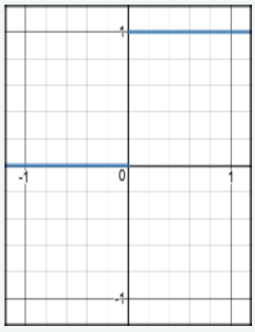
Some techniques to avoid overfitting

- regularization: Overfitting often involves large weights. So to stop parameters becoming unnecessarily large, we add a term, which is always first norm or second norm of vector w, to loss fcuntion, in order to pose penalty on large weight.
- dropout layer: during training process, we randomly drops a certain proportion of weights and only using the rest. This can be understand as distribute the responsibility of some parameters to the rest, in case it takes up too much responsibility and becomes too large.
- learning rate schedule: we often decrease the learning rate as training goes on, to make the model converge. But when the training epoch is large enough, small learning rate could lead to overfitting. Thus, we have to properly schedule our learning rate, to let model converge as quickly as it can, at the same time not overfit.

Activation function ReLU, Leaky ReLU, and ELU

ReLU

The most commonly used activation function. It abondons all values less than 0, and keeps positive values.

Function	Derivative
$R(z) = \begin{cases} z & z > 0 \\ 0 & z <= 0 \end{cases}$	$R'(z) = \begin{cases} 1 & z > 0 \\ 0 & z < 0 \end{cases}$
	

Pros

- easy to compute
- gradient is either 0 or 1, which will not cause vanishing gradient problem

Cons

- gradient not defined at 0
- 0 gradient for x<0 can make some neuron never activated during forward propogation, and also can make some nerons never updated during back propogation. -Dying ReLU problem
- Will not compress the range of output. The value will get larger and larger from layer to layer.

Leaky ReLU

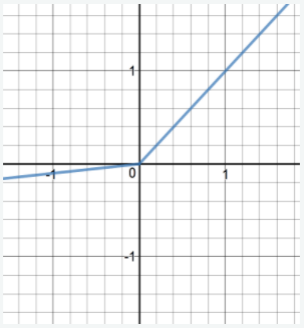
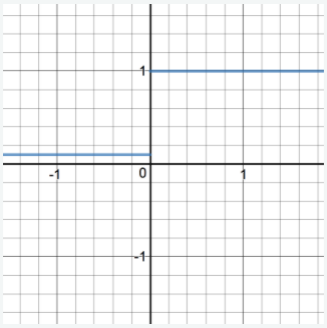
An improved version of ReLU

Pros

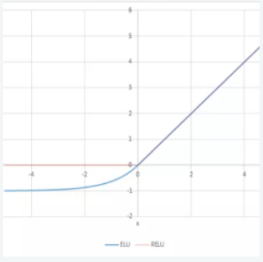
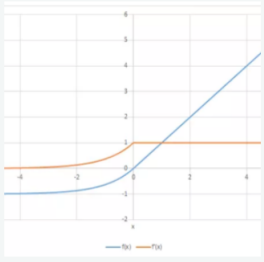
- fixed dying ReLU problem

Cons

- Not necessarily performs better than ReLU, as well as sigmoid and Tanh
- Still not differntiable at 0
- Can blow up the output

Function	Derivative
$R(z) = \left\{ \begin{array}{ll} z & z > 0 \\ \alpha z & z <= 0 \end{array} \right\}$	$R'(z) = \left\{ \begin{array}{ll} 1 & z > 0 \\ \alpha & z < 0 \end{array} \right\}$
	

ELU

Function	Derivative
$R(z) = \left\{ \begin{array}{ll} z & z > 0 \\ \alpha.(e^z - 1) & z <= 0 \end{array} \right\}$	$R'(z) = \left\{ \begin{array}{ll} 1 & z > 0 \\ \alpha.e^z & z < 0 \end{array} \right\}$
	

Pros

- Can output negative values
- slowly and smoothly converge to $-\alpha$ as x goes to $-Inf$.
- Differentiable at 0

Cons

- more expensive to compute
- Can blow up the output

Loss functions L1Loss, MSELoss and BCELoss

L1 Loss

$$\sum |\hat{y}_i - y_i|$$

In most cases, we would prefer L2 loss to L1 loss. But when there're some outliers, L2 Loss will be significantly interfered, because outliers will generate huge loss with square error. In this case, we can use L1 loss instead of L2 loss.

MSE Loss

$$\frac{1}{n}\sum(\hat{y}_i - y_i)^2$$

If we suppose our prediction are normally distributed with mean value equals true label, then we can use MSE loss. Also, the square term is consistent with many physical meanings, like energy.

BCE Loss

$$\frac{1}{n}\sum y_i log(\hat{y}_i) + (1 - y_i) log(1 - \hat{y}_i)$$

Very commonly used. Shows the difference of two distribution. Compare with MSE, BCE will give larger loss when prediction is close to target. This improves the training speed. Besides, when using softmax in the output layer, computing derivative of BCE loss will have a very tidy outcome, which simplify calculation.

b Difference performance of three datasets

I tried to vary optimizer, learning rate, regularization and dropout. It shows typically learning rate=0.01 works better than 0.001 for SGD optimizer, but is too large for Adam optimizer. Adam works better in complicated datasets like CIFAR10, but not that good in simple ones. Regularization and dropout will slightly decrease the train accuracy, but can reduce overfitting, and let the model more likely do better in generalization.

We can achieve very high accuracy easily on MNIST dataset. It's also not hard on Fashion MNIST. For CIFAR, it's challenging. They require different configuration to get the best performance.

The major reason is the dataset itself. MNIST are all numbers, which is the most determinate in shape. The FashionMNIST are clothes, which is not determinate how they should look like. So is CIFAR, it contains animals, and they're colored picture, which makes it most difficult to classify.x

c Confusion matrix and confusion pairs

I think in many cases it's not the fault of the program. Can human tell every shirt from T-shirt? No, I can not. And some will write 1 more similar to 7 than 1 itself. The confusion are caused by ambiguous human definition of what it is.

d Noisy label data

The noisy label with specific true label=i under certain ϵ follows the discrete distribution

$$P(noisylabel = i) = 1 - \epsilon, P(noisylabel = j, j \neq i) = \frac{\epsilon}{9}$$

I use *np.random.choice* to generate random variable following this distribution.

The result is quite surprising. Although accuracy drop accelerate, and deviation of test result becomes larger as label error rate begin to rise, the overall performance is excellent. When 80% of the labels are wrong, the model still reach 96% accuracy on testing set after 10 epoches of training, even though the accuracy on noisy training set is quite low. That shows CNN has strong robustness on symmetric label noise. I guess as long as the correct label has the highest probability, the resultant force will still drag the gradient to descent in the right direction. If it is trained for more iterations, maybe it will converge more sufficiently and achieve better result.

others

When constructing the network, I come up with the question. We only define the size of the convolution kernel, but did not define how the kernels are like. After reading the document, I found there're actually two conv2d function. The one used in this assignment is

```
torch.nn.Conv2d(in_channels, out_channels, kernel_size, stride=1, padding=0, dilation=1,
groups=1, bias=True, padding_mode='zeros', device=None, dtype=None)
```

The kernels are pre-defined by pytorch, and I did not find any access to them.

To customize our own kernel, like gaussian filters, we should use

```
torch.nn.functional.conv2d(input, weight, bias=None, stride=1, padding=0, dilation=1,
groups=1)
```

here 'input' is the tensor representing our customized kernel.