Assignment2

1. Use Scikit-learn for classfication on CIFAR10
2. Use Pytorch to reimplement Lenet for classification on CIFAR10

Introduction to CIFAR-10

The CIFAR-10 dataset contains 60,000 color images of 32 x 32 pixels in 3 channels divided into 10 classes. Each class contains 6,000 images. The training set contains 50,000 images, while the test sets provides 10,000 images. This image taken from the CIFAR repository (https://www.cs.toronto.edu ~kriz/cifar.html). This is a classification problem with 10 classes(muti-label classification). We can take a view on this image for more comprehension of the dataset.

Use Scikit-learn for classfication on CIFAR10

Import library we need

```
import numpy as np
from keras.datasets import cifar10
from keras.utils import to_categorical
import warnings
from sklearn.exceptions import ConvergenceWarning
from sklearn.neural_network import MLPClassifier
import matplotlib.pyplot as plt
```

Load train and test data

```
(X_train, y_train), (X_test, y_test) = cifar10.load_data()
```

```
Downloading data from https://www.cs.toronto.edu/~kriz/cifar-10-
python.tar.gz
170498071/170498071 [==============================] - 3s 0us/step
```

Look througth the shape of the data

```
print("Shape of training data:")
print(X_train.shape)
print(y_train.shape)
print("Shape of test data:")
print(X_test.shape)
print(y_test.shape)
```

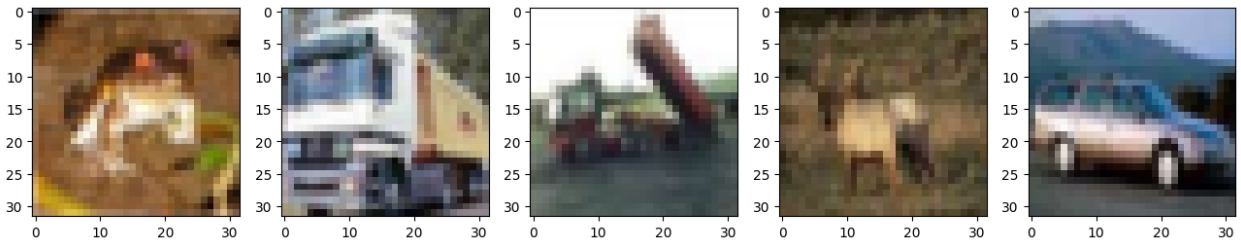
```
Shape of training data:
(50000, 32, 32, 3)
(50000, 1)
Shape of test data:
```

```
(10000, 32, 32, 3)
(10000, 1)
```

Visualization of the data `

```
cifar_classes = ['airplane', 'automobile', 'bird', 'cat', 'deer',
'dog', 'frog', 'horse', 'ship', 'truck']
print('Example training images and their labels: ' + str([x[0] for x
in y_train[0:5]]))
print('Corresponding classes for the labels: ' +
str([cifar_classes[x[0]] for x in y_train[0:5]]))

f, axarr = plt.subplots(1, 5)
f.set_size_inches(16, 6)
for i in range(5):
    img = X_train[i]
    axarr[i].imshow(img)
plt.show()
```

```
Example training images and their labels: [6, 9, 9, 4, 1]
Corresponding classes for the labels: ['frog', 'truck', 'truck',
'deer', 'automobile']
```



Normalization

```
# Transform label indices to one-hot encoded vectors
y_train = to_categorical(y_train, num_classes=10)
y_test = to_categorical(y_test, num_classes=10)

# Transform images from (32,32,3) to 3072-dimensional vectors
(32*32*3)
X_train = np.reshape(X_train,(50000,3072))
X_test = np.reshape(X_test,(10000,3072))
X_train = X_train.astype('float32')
X_test = X_test.astype('float32')

# Normalization of pixel values (to [0-1] range)
X_train /= 255
X_test /= 255
```

Define model

MLP Structure

For the MLP network, we build a fully connected network with 2 hidden layers and use relu as the activation function. In the first layer a 32323 image is expanded into a one-dimensional vecto. In the output layer output a 10*1 matrix and calculate its probability distribution using softmax.

```
from keras.models import Sequential
from keras.layers import Dense, Activation
from tensorflow.keras.optimizers.legacy import SGD

model = Sequential()
model.add(Dense(256, activation='relu', input_dim=3072))
model.add(Dense(256, activation='relu'))
model.add(Dense(10, activation='softmax'))
sgd = SGD(learning_rate=0.01, decay=1e-6, momentum=0.9, nesterov=True)
model.compile(optimizer=sgd,loss='categorical_crossentropy',metrics=['
accuracy'])
```

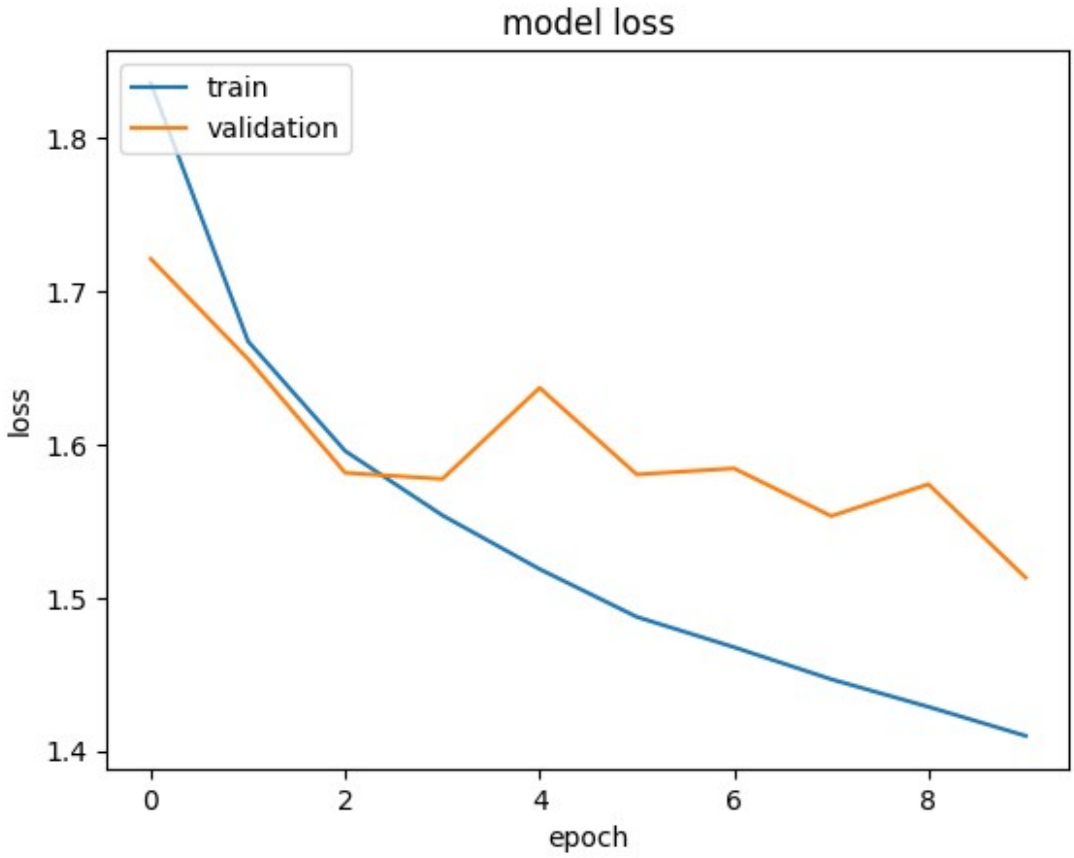
Train model

```
# Training the MLP
history = model.fit(X_train,y_train, epochs=10, batch_size=32,
verbose=1, validation_split=0.2)

Epoch 1/10
1250/1250 [==============================] - 21s 16ms/step - loss:
1.8356 - accuracy: 0.3361 - val_loss: 1.7209 - val_accuracy: 0.3860
Epoch 2/10
1250/1250 [==============================] - 14s 11ms/step - loss:
1.6672 - accuracy: 0.4029 - val_loss: 1.6557 - val_accuracy: 0.4124
Epoch 3/10
1250/1250 [==============================] - 11s 9ms/step - loss:
1.5956 - accuracy: 0.4297 - val_loss: 1.5814 - val_accuracy: 0.4350
Epoch 4/10
1250/1250 [==============================] - 12s 9ms/step - loss:
1.5537 - accuracy: 0.4419 - val_loss: 1.5774 - val_accuracy: 0.4393
Epoch 5/10
1250/1250 [==============================] - 12s 9ms/step - loss:
1.5186 - accuracy: 0.4572 - val_loss: 1.6368 - val_accuracy: 0.4139
Epoch 6/10
1250/1250 [==============================] - 12s 9ms/step - loss:
1.4874 - accuracy: 0.4665 - val_loss: 1.5803 - val_accuracy: 0.4428
Epoch 7/10
1250/1250 [==============================] - 12s 9ms/step - loss:
1.4677 - accuracy: 0.4724 - val_loss: 1.5843 - val_accuracy: 0.4305
Epoch 8/10
1250/1250 [==============================] - 12s 9ms/step - loss:
1.4468 - accuracy: 0.4815 - val_loss: 1.5532 - val_accuracy: 0.4490
Epoch 9/10
```

```
1250/1250 [==============================] - 12s 10ms/step - loss:
1.4288 - accuracy: 0.4896 - val_loss: 1.5740 - val_accuracy: 0.4439
Epoch 10/10
1250/1250 [==============================] - 14s 12ms/step - loss:
1.4099 - accuracy: 0.4978 - val_loss: 1.5131 - val_accuracy: 0.4734
```

Plot model loss

```
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'validation'], loc='upper left')
plt.show()
```



Evaluate Model

```
# Evaluating the MLP
score = model.evaluate(X_test, y_test, batch_size=128, verbose=0)
```

```
print(model.metrics_names)
print(score)
```

```
['loss', 'accuracy']
[1.4909449815750122, 0.47679999470710754]
```

Use Pytorch to reimplement Lenet for classification on CIFAR10

Import library

```
import torch
import torchvision
import torchvision.transforms as transforms
import matplotlib.pyplot as plt
import numpy as np
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
```

Download data

```
# transforms.Compose()splices the two functions together.

transform =
transforms.Compose([transforms.ToTensor(),transforms.Normalize((0.5,
0.5, 0.5), (0.5, 0.5, 0.5))])
trainset = torchvision.datasets.CIFAR10(root='../data',
train=True,download=True, transform=transform)



trainloader = torch.utils.data.DataLoader(trainset,
batch_size=4,shuffle=True, num_workers=0)

# Download test data
testset = torchvision.datasets.CIFAR10(root='../data',
train=False,download=False, transform=transform)
testloader = torch.utils.data.DataLoader(testset,
batch_size=4,shuffle=False, num_workers=0)
classes = ('plane', 'car', 'bird', 'cat','deer', 'dog', 'frog',
'horse', 'ship', 'truck')
```

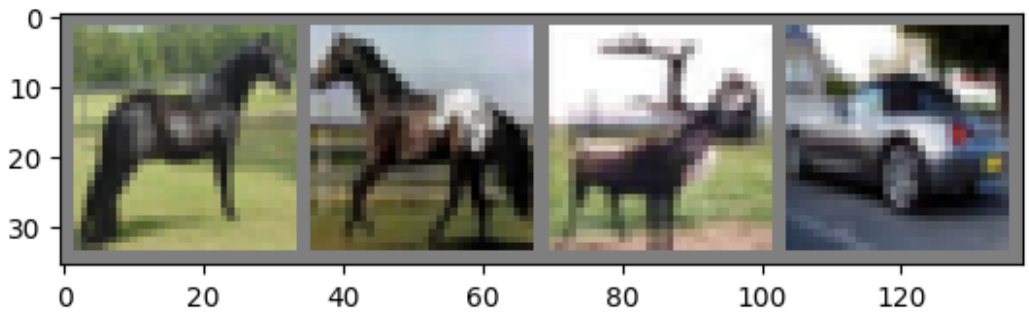
```
Downloading https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz to
../data/cifar-10-python.tar.gz
```

```
100%|██████████| 170498071/170498071 [00:01<00:00, 92672470.46it/s]
```

```
Extracting ../data/cifar-10-python.tar.gz to ../data
```

Visualization of the data

```
def imshow(img):
    # normalize：output = (input-0.5)/0.5
    # unnormalize：input = output*0.5 + 0.5
    img = img / 2 + 0.5     # unnormalize
    npimg = img.numpy()

    plt.imshow(np.transpose(npimg, (1, 2, 0)))
    plt.show()

# Obtained train data randomly
# next() Returns the next item of the iterator.
# next()
dataiter = iter(trainloader)
images, labels = next(dataiter)


# torchvision.utils.make_grid() Splice several pictures into one
picture
imshow(torchvision.utils.make_grid(images))

# Print Label

print(' '.join('%5s' % classes[labels[j]] for j in range(4)))
```



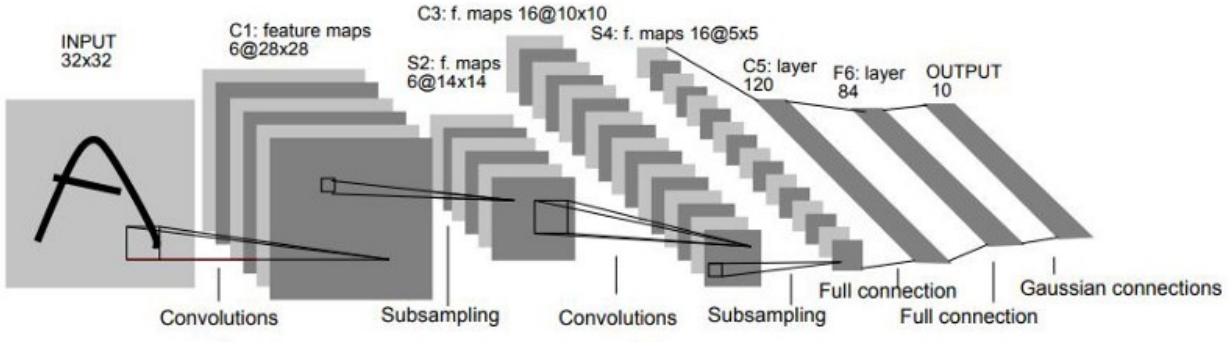

```
horse horse  deer   car
```

Define Model

LeNet Structure

The LeNet-5 architecture consists of two sets of convolutional and average pooling layers, followed by a flattening convolutional layer, then two fully-connected layers and finally a

softmax classifier.



- First Layer: The input for LeNet-5 is a 32×32×3 image which passes through the first convolutional layer with 6 feature maps or filters having size 5×5 and a stride of one. The image dimensions changes from 32x32x3 to 28x28x6.
- Second Layer: Then the LeNet-5 applies average pooling layer or sub-sampling layer with a filter size 2×2 and a stride of two. The resulting image dimensions will be reduced to 14x14x6.
- Third Layer: Next, there is a second convolutional layer with 16 feature maps having size 5×5 and a stride of 1. In this layer, only 10 out of 16 feature maps are connected to 6 feature maps of the previous layer as shown below.
- Fourth Layer: The fourth layer is again an average pooling layer with filter size 2×2 and a stride of 2. This layer is the same as the second layer (S2) except it has 16 feature maps so the output will be reduced to 5x5x16.
- Fifth Layer: The fifth layer is a fully connected convolutional layer with 120 feature maps each of size 1×1. Each of the 120 units in C5 is connected to all the 400 nodes (5x5x16) in the fourth layer.
- Sixth Layer: The sixth layer is a fully connected layer (F6) with 84 units.
- Output Layer: Finally, there is a fully connected softmax output layer ŷ with 10 possible values corresponding to the digits from 0 to 9.

```
import torch
import torch.nn as nn
import torch.nn.functional as F


# Run on GPU if you have one, CPU if you don't.
device = torch.device("cuda:0" if torch.cuda.is_available() else
"cpu")

class LeNet(nn.Module):
    def __init__(self):
        super(LeNet,self).__init__()

        # convolution layer 1: input image depth = 3, output image
depth = 16, convolution kernel size = 5 * 5, convolution step size =
1;
        # 16 represents the output dimension and the number of
```

```
convolution cores
        self.conv1 =
nn.Conv2d(in_channels=3,out_channels=16,kernel_size=5,stride=1)


        # pool layer 1: Maximum Pool, area set size = 2 * 2. Pool step
size = 2
        self.pool1 = nn.MaxPool2d(kernel_size=2,stride=2)

        # convolution layer 2
        self.conv2 =
nn.Conv2d(in_channels=16,out_channels=32,kernel_size=5,stride=1)

        # pool layer 2
        self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2)

        # full connected layer 1: input size = 32 * 5 * 5, output size
= 120
        self.fc1 = nn.Linear(32*5*5,120)

        # full connected layer 2:
        self.fc2 = nn.Linear(120,84)

        # full connected layer 3:
        self.fc3 = nn.Linear(84,10)

    def forward(self,x):
        x = F.relu(self.conv1(x))  # input(3, 32, 32) output(16, 28,
28)
        x = self.pool1(x)  # output(16, 14, 14)
        x = F.relu(self.conv2(x))  # output(32, 10, 10)
        x = self.pool2(x)  # output(32, 5, 5)
        x = x.view(-1, 32 * 5 * 5)  # output(32*5*5)
        x = F.relu(self.fc1(x))  # output(120)
        x = F.relu(self.fc2(x))  # output(84)
        x = self.fc3(x)  # output(10)
        return x

net=LeNet()
net=net.to(device)
# View the network structure
print("Network Structure")
print(net)
```

```
Network Structure
LeNet(
  (conv1): Conv2d(3, 16, kernel_size=(5, 5), stride=(1, 1))
  (pool1): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
  (conv2): Conv2d(16, 32, kernel_size=(5, 5), stride=(1, 1))
```

```
  (pool2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
  (fc1): Linear(in_features=800, out_features=120, bias=True)
  (fc2): Linear(in_features=120, out_features=84, bias=True)
  (fc3): Linear(in_features=84, out_features=10, bias=True)
)
```

Train model

```
net = LeNet()
net = net.to(device)
loss_function = nn.CrossEntropyLoss()  # Cross-entropy loss function


# optimizer = Adam , learning rate = 0.001
optimizer = optim.Adam(net.parameters(), lr=0.001)


for epoch in range(10):  # iterate 10 times
    running_loss = 0.0  # Initialize the loss function value loss=0

    for i, data in enumerate(trainloader, start=0):

            # Get the training data
            inputs, labels = data
            inputs, labels = inputs.to(device), labels.to(device)  #
Load data and labels to the GPU/CPU

            # Clear the weight parameter gradient
            optimizer.zero_grad()

            # Forward and reverse propagation
            outputs = net(inputs)#  Call the above neural network,
forward propagation
            loss = loss_function(outputs, labels)# the loss function
calculates the difference between the convolutional neural network
value and the original value
            loss.backward() # Call pytorch's automatic back
propagation function to automatically generate gradients
            optimizer.step()# Execute the optimizer to propagate the
gradient back to each network

            # Print loss
            running_loss += loss.item()
            if i % 2000 == 1999:  # print every 2000 mini-batches
                print('[%d, %5d] loss: %.3f' % (epoch + 1, i + 1,
running_loss / 2000))
                running_loss = 0.0

    print('Finished Training')
```

```
[1,  2000] loss: 1.845
[1,  4000] loss: 1.563
[1,  6000] loss: 1.478
[1,  8000] loss: 1.432
[1, 10000] loss: 1.398
[1, 12000] loss: 1.347
Finished Training
[2,  2000] loss: 1.276
[2,  4000] loss: 1.250
[2,  6000] loss: 1.250
[2,  8000] loss: 1.229
[2, 10000] loss: 1.211
[2, 12000] loss: 1.223
Finished Training
[3,  2000] loss: 1.129
[3,  4000] loss: 1.143
[3,  6000] loss: 1.131
[3,  8000] loss: 1.128
[3, 10000] loss: 1.132
[3, 12000] loss: 1.117
Finished Training
[4,  2000] loss: 1.047
[4,  4000] loss: 1.047
[4,  6000] loss: 1.049
[4,  8000] loss: 1.064
[4, 10000] loss: 1.044
[4, 12000] loss: 1.054
Finished Training
[5,  2000] loss: 0.982
[5,  4000] loss: 0.996
[5,  6000] loss: 0.984
[5,  8000] loss: 1.010
[5, 10000] loss: 1.007
[5, 12000] loss: 0.995
Finished Training
[6,  2000] loss: 0.906
[6,  4000] loss: 0.930
[6,  6000] loss: 0.940
[6,  8000] loss: 0.985
[6, 10000] loss: 0.958
[6, 12000] loss: 0.965
Finished Training
[7,  2000] loss: 0.885
[7,  4000] loss: 0.894
[7,  6000] loss: 0.914
[7,  8000] loss: 0.915
[7, 10000] loss: 0.930
[7, 12000] loss: 0.920
Finished Training
[8,  2000] loss: 0.849
```

```
[8,  4000] loss: 0.863
[8,  6000] loss: 0.903
[8,  8000] loss: 0.881
[8, 10000] loss: 0.890
[8, 12000] loss: 0.897
Finished Training
[9,  2000] loss: 0.807
[9,  4000] loss: 0.846
[9,  6000] loss: 0.873
[9,  8000] loss: 0.850
[9, 10000] loss: 0.869
[9, 12000] loss: 0.856
Finished Training
[10,  2000] loss: 0.797
[10,  4000] loss: 0.810
[10,  6000] loss: 0.818
[10,  8000] loss: 0.844
[10, 10000] loss: 0.843
[10, 12000] loss: 0.850
Finished Training
```

Test model

```
correct = 0
total = 0


# with torch.no_grad() Indicates that it includes content that does
not need to compute gradients, nor does it need to be propagated
backward, saving memory
with torch.no_grad():
    for data in testloader:
        images, labels = data
        images, labels = images.to(device), labels.to(device)
        outputs = net(images)
        # torch.max(outputs.data, 1)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

print('Accuracy of the network on the 10000 test images: %d %%' % (100
* correct / total))

# Print the accuracy of 10 classes
class_correct = list(0. for i in range(10)) #class_correct=[0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
class_total = list(0. for i in range(10)) #class_total=[0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
with torch.no_grad():
```

```
    for data in testloader:
        images, labels = data
        images, labels = images.to(device), labels.to(device)
        outputs = net(images) # outputs vector : 4*10
        # torch.max(outputs.data,
        # predicted vector : 4*1
        _, predicted = torch.max(outputs, 1)

        c = (predicted == labels).squeeze()
        for i in range(4):
            label = labels[i]
            class_correct[label] += c[i].item()
            class_total[label] += 1
for i in range(10):
    print('Accuracy of %5s : %2d %%' % (classes[i], 100 *
class_correct[i] / class_total[i]))
```

```
Accuracy of the network on the 10000 test images: 63 %
```

Save the weights and parameters

```
save_path = 'Lenets.pth'
torch.save(net.state_dict(), save_path)
```

Make predict by the model we train

```
import torch
import torchvision.transforms as transforms
from PIL import Image
from torch import nn
import torch.nn.functional as F


transform = transforms.Compose([transforms.Resize((32,
32)),transforms.ToTensor(),transforms.Normalize((0.5, 0.5, 0.5), (0.5,
0.5, 0.5))])

classes = ('plane', 'car', 'bird', 'cat','deer', 'dog', 'frog',
'horse', 'ship', 'truck')

class LeNet(nn.Module):
    def __init__(self):
        super(LeNet,self).__init__()
        # Convolution layer 1
        self.conv1 =
nn.Conv2d(in_channels=3,out_channels=16,kernel_size=5,stride=1)
        # Pool Layer 1
        self.pool1 = nn.MaxPool2d(kernel_size=2,stride=2)
        # Convolution layer 2
```

```
        self.conv2 =
nn.Conv2d(in_channels=16,out_channels=32,kernel_size=5,stride=1)
        # Pool layer 2
        self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2)
        # Full connected layer 1
        self.fc1 = nn.Linear(32*5*5,120)
        # Full connected layer 2
        self.fc2 = nn.Linear(120,84)
        # Full connected layer 3
        self.fc3 = nn.Linear(84,10)

    def forward(self,x): # Link the input layer, the network layer and
the output layer for forward data transmission
        x = F.relu(self.conv1(x))  # input(3, 32, 32) output(16, 28,
28)
        x = self.pool1(x)  # output(16, 14, 14)
        x = F.relu(self.conv2(x))  # output(32, 10, 10)
        x = self.pool2(x)  # output(32, 5, 5)
        x = x.view(-1, 32 * 5 * 5)  # output(32*5*5)
        x = F.relu(self.fc1(x))  # output(120)
        x = F.relu(self.fc2(x))  # output(84)
        x = self.fc3(x)  # output(10)
        return x

net = LeNet()
net.load_state_dict(torch.load('Lenets.pth'))

im = Image.open('cat.jpg')
im = transform(im)  # [C, H, W]

# [batch , channel , height , width] ,
im = torch.unsqueeze(im, dim=0)  # [N, C, H, W]

with torch.no_grad():
    outputs = net(im)
    predict = torch.max(outputs, dim=1)[1].data.numpy()
print(classes[int(predict)])

# Predit result
with torch.no_grad():
    outputs = net(im)
    predict = torch.softmax(outputs,dim=1) #
[batch , channel , height , width],
print(predict)

cat
tensor([[0.0062, 0.0065, 0.0446, 0.3172, 0.0830, 0.2256, 0.0430,
0.1104, 0.1509,
         0.0127]])
```