Computers and Artificial Intelligence; Vol.1, No.1; 2024
E-ISSN: 3006 -3728 P-ISSN: 3006-371X
DOI: https://doi.org/10.70267/eta2tt72
Published by: Zeus Press

Application research of image classification based on Pytorch and Convolutional neural network

Zhaoyu Li

Guangdong University Of Science And Technology, China

Abstract

The image classification problem of convolutional neural network (CNN) on CIFAR-10 dataset is studied, and the model is implemented and experimented with PyTorch. Through the training and testing of the model, we analyze the performance of the model and explore the possibility of improvement. The experimental results show that the model achieves a certain accuracy in the image classification task, but there is space for improvement in some categories.

Keywords

Image classification, neural network, PyTorch, Model, Training

1. Introduction

Convolutional neural network (CNN) is a kind of feedforward neural network with convolutional computation and deep structure, which is one of the representative algorithms of deep learning. LeNet, the earliest convolutional neural network, was successful in handwritten character recognition tasks and laid the foundation for subsequent deep learning research (LeCun et al., 1998). After the 21st century, with the proposal of deep learning theory and the improvement of numerical computing equipment, CNN have developed rapidly. Convolutional neural network (CNN), as an important model of deep learning, is widely used in image classification, object detection and other tasks(Goodfellow et al., 2016). Image classification is an important task in the field of computer vision. The purpose of this study is to explore the application of PyTorch-based CNN model on CIFAR-10 dataset and analyze its classification performance.

2. Related work

In recent years, significant progress has been made in the field of deep learning, with many researchers applying deep learning techniques, especially convolutional neural networks, to image classification tasks. For example, AlexNet, proposed by Krizhevsky et al. (2012) achieved notable results in the ImageNet competition. Or the ResNet model proposed by He et al. (2016), which achieved significant performance improvements in the ImageNet image classification challenge, winning the 2015 ImageNet Competition. At the same time, it also greatly promotes the application of deep learning in other computer vision tasks. Based on these, this study uses PyTorch framework to implement the CNN model and conducts classification experiments on the CIFAR-10 dataset.

3. Methods

The model structure used in this paper consists of two convolutional layers, using ReLU activation function and maximum pooling layer respectively, which are subsequently classified by the fully connected layer. The specific implementation is shown below.

First, import the CIFAR-10 dataset using PyTorch and pre-process it. The dataset contains 60,000 32x32 color images divided into 10 categories. In data pre-processing, we normalized the images to the [-1, 1] range and set the mini-batch size to 4.

The first convolution layer of the model contains 6 5x5 convolution kernels with a step length of 1; The second convolution layer contains 16 5x5 convolution kernels. After two convolution and pooling operations, the feature map is flattened and fed into the fully connected layer, which finally outputs the probabilities of 10 classes.

4. Experimental setup

The experiment was conducted over a 20-round training course using a cross entropy loss function and stochastic gradient Descent (SGD) optimizer with momentum. The learning rate was set to 0.001 and the momentum parameter to 0.9. The experimental environment is PyTorch framework and the hardware is a normal GPU server.

5. Experimental results

During the training process, we recorded the loss value after every 2000 mini-batch. The results showed that the loss value gradually decreased with the increase of the rounds, indicating that the model performed well during the learning process. In the end, the accuracy of the model on the test set was 61 percent.

6. Discuss

While the model performed better in some categories, the overall accuracy was only 61%, indicating that the model still has space for improvement. In particular, the model performed poorly on image classification tasks against complex backgrounds. The preliminary conclusion is that future research can try to adjust the learning rate of existing model, add data enhancement techniques, or adopt deeper network structures to improve performance. In order to further improve the existing model in the future, by analyzing the ResNet model proposed by He et al. (2016), it is found that in the existing model, a new feature representation is directly output after the input goes through multiple convolutional layers, nonlinear activation layers and pooling layers. However, when the depth of the network increases, the problem of gradient disappearance or gradient explosion can cause training difficulties and affect the learning rate. The core innovation of ResNet was the introduction of a residual-block structure, whereby inputs were directly skipped over certain layers via a shortcut connection and then added to the outputs of those layers. The advantage is very obvious, in deep networks, by introducing residual connections, skipping connections can directly transfer the gradient from the back layer to the front layer, alleviating the problem of gradient disappearance as the number of layers increases. As a result, ResNet successfully trained a network with a depth of 152 layers, without the problem of training degradation (i.e., training error increases as the number of layers increases) that occurs in traditional networks at around 30-40 layers, which is one of the problems found in the existing model during the experiment.

7. Conclusion

This paper studies the application of convolutional neural network based on PyTorch in image classification task. Experiments show that the performance of the model on the CIFAR-10 dataset has certain reference value, but there is still space for improvement. Future work will focus on model optimization and improving accuracy.

References

LeCun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). Gradient-based learning applied to document recognition. Proceedings of the IEEE, 86(11), 2278-2324.

Goodfellow, I., Bengio, Y., & Courville, A. (2016). Deep Learning. MIT Press.

Krizhevsky, A., Sutskever, I. and Hinton, G. E. (2012). ImageNet classification with deep convolutional neural networks. *Advances in Neural Information Processing Systems,* vol. 25, pp. 1097-1105.

He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR), 770-778.

Funding

This research received no external funding.

Conflicts of Interest

The authors declare no conflict of interest.

Acknowledgment

The author would like to thank Ms. Gao for their invaluable guidance and insightful feedback throughout this research.

Copyrights

Addendum

According to the data set and problem description, I will use convolutional neural network (CNN) to analyze the programming code and results of classification prediction for images in the CIFAR-10 dataset.

The CIFAR-10 dataset contains 60,000 32*32 color images divided into 10 categories, with 6,000 images per category. There were 50,000 training images and 10,000 test images.

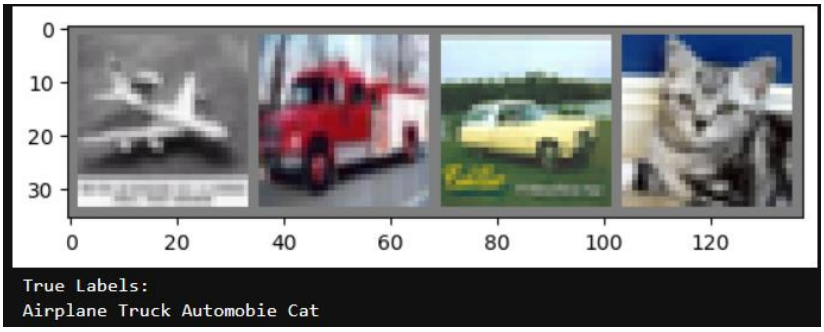
In order to implement convolutional neural network (CNN), we will use PyTorch. The experimental procedure code and results are as follows.

```
# [1]

# Import necessary packages with database

import torch

import torchvision

import torchvision.transforms as transforms
```

```
import matplotlib.pyplot as plt

import torch.nn as nn

import torch.nn.functional as F

import numpy as np

from torch import device

import torch.optim as optim

from torch.testing. internal.data.network1 import Net


# [2]

# Since the CIFA10 dataset in the torchvision database is PIL images output in the range [0,1], we are going to convert them to Tensors in the range [-1,1]

transform = transforms.Compose(

    [transforms.ToTensor(),

     transforms the Normalize ((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])


# Here we set the mini-batch size to 4, meaning that there will be four pieces of data (4 images) in a mini-batch in the training dataset.

batch_size = 4


# Define training datasets ‘trainset’ and test datasets ‘testset’, store them in the root folder of './data 'and convert them to Tensors and finally put them in the appropriate data loader

trainset = torchvision.datasets.CIFAR10(root='./data', train=True, download=True, transform=transform)

trainloader = torch.utils.data.DataLoader(trainset, batch_size=batch_size, shuffle=True, num_workers=2)

testset = torchvision.datasets.CIFAR10(root='./data', train=False, download=True, transform=transform)

testloader = torch.utils.data.DataLoader(testset, batch_size=batch_size, shuffle=False, num_workers=2)

classes = ('Airplane', 'Automobie', 'Bird', 'Cat',

           'Deer', 'Dog', 'Frog', 'Horse', 'Ship', 'Truck')


# [3]

# Here we can randomly print a random 4 image of the training data loader
```

```
# functions to show an image

def imshow(img):
    img = img / 2 + 0.5 # unnormalize
    npimg = img.numpy()
    plt.imshow(np.transpose(npimg, (1, 2, 0)))
    plt.show()

# get some random training images

# Create an iterator object and feed it into the training data loader,

dataiter = iter(trainloader)

# Simultaneously call the next() function to get the next data until you have iterated through the batch-size sequence

images, labels = next(dataiter)

# Call imshow function to output image

imshow(torchvision.utils.make_grid(images))

# Print the correct label corresponding to the data

print("True Labels:")

print(' '.join(f'{classes[labels[j]]:5s}' for j in range(batch_size)))
```

The output looks like this:



[4]

```
# Build a convolutional neural network (CNN) model

class Net(nn.Module):

    def __init__(self):

        super().__init__()

            # First layer: There are 6 5*5 convolution filters where the stride is 1

        self.conv1 = nn.Conv2d(3, 6, 5, 1)

            # Define a 2*2 maximum pooled layer after each filter, where the stride is 2

        self.pool = nn.MaxPool2d(2, 2)

            # Second layer: There are 16 5*5 convolution filters where the stride is 1

        self.conv2 = nn.Conv2d(6, 16, 5, 1)

        # After two layers of convolution and pooling, a 32*32*3 image is finally input into the fully
connected layer as a 5*5*16 feature map

            # Define a fully connected layer with dimension 120

        self.fc1 = nn.Linear(16 * 5 * 5, 120)

# Define a fully connected layer with dimension 84

        self.fc2 = nn.Linear(120, 84)

# Define a fully connected layer of dimension 10 to output unnormalized fractions for 10 categories

        self.fc3 = nn.Linear(84, 10)


def forward(self, x):

# Activate functions to use ReLU uniformly

        x = self.pool(F.relu(self.conv1(x)))

        x = self.pool(F.relu(self.conv2(x)))

        x = torch.flatten(x, 1)

        x = F.relu(self.fc1(x))

        x = F.relu(self.fc2(x))

        x = self.fc3(x)
```

```
        return x

net = Net()

# Print the Net model specific structure

print(net)
```

The output is as follows:

```
Net(
  (conv1): Conv2d(3, 6, kernel_size=(5, 5), stride=(1, 1))
  (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (conv2): Conv2d(6, 16, kernel_size=(5, 5), stride=(1, 1))
  (fc1): Linear(in_features=400, 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)
)
```

```
# [5]

# In the CNN model, the classification cross entropy loss function acts as our loss function

criterion = nn.CrossEntropyLoss()

# Here we use stochastic gradient Descent (SGD) with momentum as the optimizer, where learning-rate takes 0.001 and momentum takes the common 0.9

optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)


# [6]

# Start training our CNN model

epochs = 20

for epoch in range(epochs):    # loop over the dataset multiple times

    running_loss = 0.0

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

        # get the inputs; data is a list of [inputs, labels]

        inputs, labels = data

        # zero the parameter gradients

        optimizer.zero_grad()
```

```
            # forward + backward + optimize
            outputs = net(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()
            # print statistics
            running_loss += loss.item()
            if i % 2000 == 1999: # The model parameters are recorded with 2000 Mini-batches per iteration
                print(f'[{epoch + 1}, {i + 1:5d}] loss: {running_loss / 2000:.3f}')
                running_loss = 0.0

print('Finished Training')
```

The output is as follows:

The loss of Epoch 1-20 is as follows:

```
[1,  2000] loss: 2.174
[1,  4000] loss: 1.860
[1,  6000] loss: 1.667
[1,  8000] loss: 1.590
[1, 10000] loss: 1.517
[1, 12000] loss: 1.454
[2,  2000] loss: 1.403
[2,  4000] loss: 1.380
[2,  6000] loss: 1.327
[2,  8000] loss: 1.325
[2, 10000] loss: 1.316
[2, 12000] loss: 1.292
[3,  2000] loss: 1.222
[3,  4000] loss: 1.215
[3,  6000] loss: 1.227
[3,  8000] loss: 1.191
[3, 10000] loss: 1.200
[3, 12000] loss: 1.163
[4,  2000] loss: 1.123
[4,  4000] loss: 1.108
[4,  6000] loss: 1.123
[4,  8000] loss: 1.108
[4, 10000] loss: 1.121
[4, 12000] loss: 1.121
[5,  2000] loss: 1.039
[5,  4000] loss: 1.066
[5,  6000] loss: 1.036
[5,  8000] loss: 1.034
[5, 10000] loss: 1.056
[5, 12000] loss: 1.078
[6,  2000] loss: 0.969
[6,  4000] loss: 0.965
[6,  6000] loss: 1.006
[6,  8000] loss: 1.004
[6, 10000] loss: 1.000
[6, 12000] loss: 1.012
[7,  2000] loss: 0.915
[7,  4000] loss: 0.939
[7,  6000] loss: 0.963
[7,  8000] loss: 0.935
[7, 10000] loss: 0.946
[7, 12000] loss: 0.962
[8,  2000] loss: 0.882
[8,  4000] loss: 0.895
[8,  6000] loss: 0.913
[8,  8000] loss: 0.903
[8, 10000] loss: 0.915
[8, 12000] loss: 0.941
[9,  2000] loss: 0.819
[9,  4000] loss: 0.868
[9,  6000] loss: 0.862
[9,  8000] loss: 0.867
[9, 10000] loss: 0.894
[9, 12000] loss: 0.901
[10,  2000] loss: 0.800
[10,  4000] loss: 0.822
[10,  6000] loss: 0.818
[10,  8000] loss: 0.868
[10, 10000] loss: 0.872
[10, 12000] loss: 0.844
[11,  2000] loss: 0.771
[11,  4000] loss: 0.774
[11,  6000] loss: 0.806
[11,  8000] loss: 0.825
[11, 10000] loss: 0.827
[11, 12000] loss: 0.844
[12,  2000] loss: 0.740
[12,  4000] loss: 0.768
[12,  6000] loss: 0.804
[12,  8000] loss: 0.802
[12, 10000] loss: 0.810
[12, 12000] loss: 0.802
[13,  2000] loss: 0.695
[13,  4000] loss: 0.743
[13,  6000] loss: 0.779
[13,  8000] loss: 0.779
[13, 10000] loss: 0.791
[13, 12000] loss: 0.798
[14,  2000] loss: 0.684
[14,  4000] loss: 0.716
[14,  6000] loss: 0.742
[14,  8000] loss: 0.764
[14, 10000] loss: 0.747
[14, 12000] loss: 0.784
[15,  2000] loss: 0.667
[15,  4000] loss: 0.675
[15,  6000] loss: 0.716
[15,  8000] loss: 0.764
[15, 10000] loss: 0.778
[15, 12000] loss: 0.765
[16,  2000] loss: 0.650
[16,  4000] loss: 0.664
[16,  6000] loss: 0.694
[16,  8000] loss: 0.745
[16, 10000] loss: 0.753
[16, 12000] loss: 0.758
[17,  2000] loss: 0.627
[17,  4000] loss: 0.667
[17,  6000] loss: 0.686
[17,  8000] loss: 0.709
[17, 10000] loss: 0.736
[17, 12000] loss: 0.745
[18,  2000] loss: 0.622
[18,  4000] loss: 0.669
[18,  6000] loss: 0.691
[18,  8000] loss: 0.702
[18, 10000] loss: 0.700
[18, 12000] loss: 0.741
[19,  2000] loss: 0.604
[19,  4000] loss: 0.649
[19,  6000] loss: 0.658
[19,  8000] loss: 0.683
[19, 10000] loss: 0.701
[19, 12000] loss: 0.745
[20,  2000] loss: 0.602
[20,  4000] loss: 0.643
[20,  6000] loss: 0.660
[20,  8000] loss: 0.694
[20, 10000] loss: 0.691
[20, 12000] loss: 0.707
Finished Training
```

Analysis of results:

In the process of gradient descent, the change of the value of loss is non-linear, indicating that the value of learning-rate is relatively large, which is one of the points worthy of optimization of this CNN model

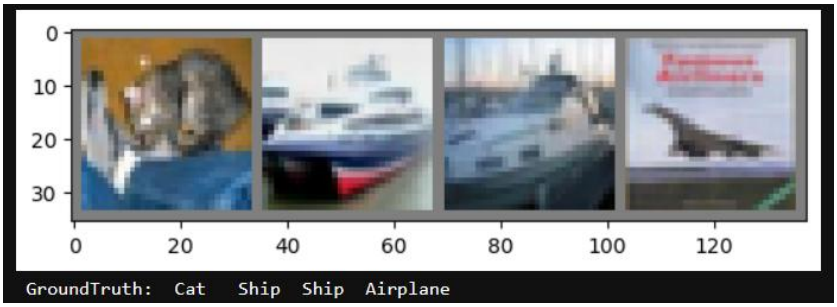
```
# [7]

# Save the trained CNN model

PATH = './cifar_net.pth'

torch.save(net.state_dict(), PATH)


# [8]

# Create an iterator object and feed it into the test data loader,

dataiter = iter(testloader)

# Simultaneously call the next() function to get the next data until the batch-size sequence has been traversed

images, labels = next(dataiter)
```

#

imshow(torchvision.utils.make_grid(images))

print('GroundTruth: ', ' '.join(f'{classes[labels[j]]:5s}' for j in range(batch_size)))

The output looks like this:



[9]

Call the CNN model to test

net = Net()

net.load_state_dict(torch.load(PATH))

outputs = net(images)

Call torch.max() to take the largest tensor value in the outputs as predicted

_, predicted = torch.max(outputs, 1)

print('Predicted: ', ' '.join(f'{classes[predicted[j]]:5s}' for j in range(4)))

The output is as follows:

```
Predicted:  Frog  Dog   Automobie Horse
```

Result analysis:

There is a gap between the predicted value of the model ‘predicted’ and the actual value ‘GroudTruth’.

[10]

```
# Variables ‘correct’ and ‘total’ were used to count the number of correct predictions and total number to calculate the model prediction accuracy

correct = 0

total = 0

# We don't need to do any more gradient descent on the test data set

with torch.no_grad():

    for data in testloader:

        images, labels = data

        total += labels.size(0)

        outputs = net(images)

        _, predictions = torch.max(outputs, 1)

        for label, prediction in zip(labels, predictions):

            if label == prediction:

                correct += 1

# Print prediction accuracy in a test dataset with 10,000 images

print(f'Accuracy of the network on the 10000 test images: {100 * correct // total} %')
```

The output is as follows:

```
Accuracy of the network on the 10000 test images: 61 %
```

The prediction was 61 percent accurate

```
# [11]

# Calculate the accuracy of predicting images for each category separately

correct_pred = {classname: 0 for classname in classes}

total_pred = {classname: 0 for classname in classes}

# Do not do gradient descent

with torch.no_grad():
```

```
    for data in testloader:
        images, labels = data
        outputs = net(images)
        _, predictions = torch.max(outputs, 1)
        # collect the correct predictions for each class
        for label, prediction in zip(labels, predictions):
            if label == prediction:
                correct_pred[classes[label]] += 1
            total_pred[classes[label]] += 1

# Print separately to predict the accuracy of each category of images
for classname, correct_count in correct_pred.items():
    accuracy = 100 * float(correct_count) / total_pred[classname]
print(f'Accuracy for class: {classname:5s} is {accuracy:.1f} %')
```

The output is as follows:

```
Accuracy for class: Airplane is 63.8 %
Accuracy for class: Automobie is 80.9 %
Accuracy for class: Bird  is 47.8 %
Accuracy for class: Cat   is 40.4 %
Accuracy for class: Deer  is 57.5 %
Accuracy for class: Dog   is 44.9 %
Accuracy for class: Frog  is 64.4 %
Accuracy for class: Horse is 68.3 %
Accuracy for class: Ship  is 74.2 %
Accuracy for class: Truck is 72.9 %
```

Analysis of results:

By analyzing the prediction results of the model on the test data set, it can be seen that the CNN model has learned some features of each of the ten categories of images after the training of the CIFAR-10 training data set, but we can still see which categories it performs well and which ones poorly in the prediction. For the class with high accuracy, we can determine that the model is better to learn, while for the class with low accuracy, we need to improve the model (for example, make appropriate adjustments to the model learning rate) or use more training data to improve the performance.