# Keras 1D CNN

import numpy as np

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Embedding, Conv1D, GlobalMaxPooling1D, Dense

# Parameters

word\_embedding\_dim = 3  # Dimensionality of word embeddings

num\_filters = 64  # Number of filters in the convolutional layers

filter\_sizes = [3, 4, 5]  # Sizes of filters in the convolutional layers

hidden\_units = 64  # Number of units in the hidden layer

dropout\_rate = 0.4  # Dropout rate for regularization

# Define the model

'''

sequential model, each added layer will be executed sequentially

for each layer, only need to specify output-related parameters, for input related parameters, calculated automatically

'''

model = Sequential()

model.add(Embedding(input\_dim=vocabulary\_size, output\_dim=word\_embedding\_dim, input\_length=max\_sequence\_length))

'''

input dim: (max\_sequence\_length, word\_embedding\_dim)

output dim: (max\_sequence\_length - filter\_sizes[0] + 1, num\_filters)

num\_filters means how many conv kernels,

kernel\_size means the window lenth for the kernel. eg if size=3, 3 words are convolved together

kerel's shape: (kernel\_size, word\_embedding\_dim, num\_filters)

'''

model.add(Conv1D(filters=num\_filters, kernel\_size=filter\_sizes[0], activation='relu'))

model.add(Conv1D(filters=num\_filters, kernel\_size=filter\_sizes[1], activation='relu'))

'''

output dim: (max\_sequence\_length - padding... , num\_filters)

'''

model.add(Conv1D(filters=num\_filters, kernel\_size=filter\_sizes[2], activation='relu'))

'''

max along the sequence length dimension (extract the most important feature), reduces the input dimension to (num\_filters,)

output dim: (num\_filters,)

'''

model.add(GlobalMaxPooling1D())

'''

first param specifies the num of neurons (perceptrons in the layer)

in keras, all associated with a bias term

input: vector with len num\_filters

ouput: vector with len hidden\_units

'''

model.add(Dense(hidden\_units, activation='relu'))

'''

input: len=hidden\_units

output: len=1, since activation is sigmoid, the output is probability

'''

model.add(Dense(1, activation='sigmoid'))

# Compile the model

model.compile(loss='binary\_crossentropy', optimizer='adam', metrics=['precision', 'recall'])

# Print the model summary

model.summary()

train:

model.fit(X, y, epochs=10, batch\_size=32, validation\_split=0.2)

test:

predictions = model.predict(new\_padded\_sequences)

predicted\_labels = [1 if pred > 0.5 else 0 for pred in predictions]

# 1D CNN with pytorch

Need to reshape from (B, L, emb) to (B, emb, L)

Pooling:  
F.max\_pool1d(x\_fm, kernel\_size=2) (or F.avg\_pool1d(x\_fm, kernel\_size=2))

[B, C, L] -> [B, C, L//2]

# 2D CNN with PyTorch

Import:

import torch

import torch.nn as nn

import torch.optim as optim

import torchvision

import torchvision.transforms as transforms

import torch.nn.functional as F

Load data:

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=2)

testset = torchvision.datasets.CIFAR10(root='./data', train=False,

                                       download=True, transform=transform)

testloader = torch.utils.data.DataLoader(testset, batch\_size=4,

                                         shuffle=False, num\_workers=2)

classes = ('plane', 'car', 'bird', 'cat',

           'deer', 'dog', 'frog', 'horse', 'ship', 'truck')

Construct model:

'''A new class Net is defined inheriting from nn.Module'''

class Net(nn.Module):

    '''\_\_init\_\_ function is the constructor where you define the layers of the neural network'''

    def \_\_init\_\_(self):

        super(Net, self).\_\_init\_\_()     # super: call the method of the superclass

        '''

        nn.Conv2d(in\_channels, out\_channels, kernel\_size)

        kernel\_size: by default square, or can: (3, 5)

        stride: if set to none: default to ?

        or: nn.Conv2d(in\_channels=3, out\_channels=16, kernel\_size=3, stride=1, padding=1)

        '''

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

        '''

        nn.MaxPool2d(kernel\_size, stride)

        if stride is None, default is kernel\_size

        or: nn.MaxPool2d(kernel\_size=2, stride=2)

        eg:

            Input tensor:

            tensor([[[[0.5804, 0.0462, 0.6624, 0.8841],

                    [0.4399, 0.4161, 0.0346, 0.6780],

                    [0.6716, 0.0941, 0.3934, 0.9827],

                    [0.9619, 0.0462, 0.2167, 0.4764]]]])

            Output tensor after MaxPool2d:

            tensor([[[[0.5804, 0.8841],

                    [0.9619, 0.9827]]]])

        '''

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

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

        '''

        nn.Linear(in\_features, out\_features)

        fully connected layer, weight init randomly, dft have bias

        does not include activation

        '''

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

        self.fc2 = nn.Linear(120, 84)

        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):

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

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

        x = x.view(-1, 16 \* 5 \* 5)

        x = F.relu(self.fc1(x))

        x = F.relu(self.fc2(x))

        x = self.fc3(x)

        return x

train:

net = Net()

# Define loss function and optimizer

criterion = nn.CrossEntropyLoss()

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

# Train the network

for epoch in range(10):  # loop over the dataset multiple times

    '''keep track of the cumulative loss (or average loss) over a period of time, jut for reference, what is the "average" loss in a time'''

    running\_loss = 0.0

    '''

    0: start param

    data is a tuple (?), first element is input, second is label

    '''

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

        '''

        input shape eg: (4, 3, 32, 32) for 4 images with 3 channels of size 32x32

        labels shape eg: (4) for 4 labels

        '''

        inputs, labels = data

        '''

        clears the gradient of all optimized tensors

        '''

        optimizer.zero\_grad()

        outputs = net(inputs)

        loss = criterion(outputs, labels)

        loss.backward()

        '''

        updates the model's parameters based on the computed gradients and optimizer settings

        '''

        optimizer.step()

        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

test:

correct = 0

total = 0

'''

no\_grad() is a context manager that is used to deactivate gradient calculation.

This is particularly useful when you are evaluating a model rather than training it.

By using torch.no\_grad(), you are telling PyTorch not to track operations in terms of gradients within the block of code where it's applied.

'''

with torch.no\_grad():

    for data in testloader:

        images, labels = data

        outputs = net(images)

        \_, 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))