به نام خدا

عنوان: پروژه دوم درس هوش محاسباتی

استاد: حسین کارشناس

اعضای گروه:

علی پورقیصری

طاها داوری

محمد امین مولوی‌زاده

مقدمه:

این پروژه به سه بخش تقسیم شده است. در بخش اول به پیاده سازی یک شبکه عصبی پرداخته‌ایم که به دو بخش استخراج کننده ویژگی و دسته‌بند تقسیم شده است. استخراج کننده ویژگی ۳٤Resnet است و لایه آخر آن حذف شده و دسته‌بند که به صورت دستی نوشته شده است اضافه گردیده است.

این شبکه آموزش دیده و سپس ارزیابی می‌شود که در ادامه نحوه کار و همچنین نحوه ارزیابی نوشته شده است.

قسمت اول :

روش انجام آزمایش :

در ابتدا کتابخانه های مورد نیاز را ذکر می‌کنیم. در این پروژه از pytorch برای پیاده سازی در استخراج کننده ویژگی استفاده شده است.

در قالب داده شده برای این بخش یکسری کلاس داریم که به صورت زیر می‌باشد:

ReLU: این کلاس برای پیاده سازی فعال ساز ReLU است و در لایه‌ی مخفی استفاده شده که دو تابع forward و backward دارد. در تابع forward عملیات برای کلاس بندی و حرکت رو به جلو انجام می‌شود و عملکرد Relu طبق فرمول محاسبه آن پیاده سازی شده که به سادگی بیشینه مقدار ورودی و صفر است. در تابع backward مشتق تابع forward نسبت به ورودی داده شده گرفته می‌شود که برای تغییر وزن و عرض از مبدا های دسته‌بند استفاده می‌شود. سایر کلاس ها نیز این توابع را دارند که کارایی آنها نیز به همین صورت گفته شده می‌باشد.

Sigmoid: این کلاس برای پیاده سازی فعال ساز sigmoid است. این کلاس پیاده سازی شده است ولی از آن در کد استفاده نشده است.

Softmax: این کلاس برای پیاده سازی فعال ساز softmax است و فرمول آن پیاده سازی شده است و در خروجی دسته‌بند به کار برده شده.

Categorical_Cross_Entropy_loss: این کلاس به ما در مشخص کردن اختلاف هر خروجی به دست آمده از دسته‌بند و خروجی واقعی که از مجموعه داده به دست آمده

کمک می‌کند. کارایی این تابع به این صورت است که خروجی لایه softmax به تابع forward این کلاس داده می‌شود و با استفاده از تعداد ورودی‌ها و خروجی مورد انتظار، لگاریتم این مجموعه گرفته شده و به یک مجموعه تبدیل می‌شوند و سپس بر روی این مجموعه میانگین گرفته می‌شود و مقدار تفاوت به دست می آید. دقت شود که چون خروجی softmax ممکن است صفر باشد و لگاریتم صفر بی نهایت می‌شود پس بنابر این یک مقدار کم به این مقادیر اضافه شده است. در تابع backward هم به صورت قبل مشتق گرفته شده است اما اضافه بر مشتق، مقدار به اصطلاح one hot encoded مقادیر خروجی مجموعه داده گرفته شده است.

SGD: کارکرد این کلاس بسیار ساده است و الگوریتم کاهش شیب تصادفی را پیاده سازی می کند.

Dense: این کلاس همان لایه های ما است که در تابع سازنده مقادیری تصادفی برای وزنها و عرض از مبداها در نظر گرفته می‌شود. forward و backward این کلاس هم مانند کلاس‌های قبل عملیات رو به جلو و رو به عقب را با توجه به فرمول مشخصی پیاده سازی می‌کنند.

پس از پیاده سازی کلاس‌ها، نوبت به آماده سازی مجموعه داده برای استفاده می‌باشد. برای این کار مجموعه داده را بارگیری کرده و ذخیره می‌کنیم. پس از آن مشخص می‌کنیم که در هر بار گردش چه مقدار داده به شبکه داده شود تا بر اساس آنها آموزش ببیند. پس از آن داده های ذخیره شده را بر اساس اندازه در نظر گرفته شده برای ورودی های شبکه، بخش بندی می کنیم.

در مرحله بعد استخراج کننده ویژگی را که از قبل آموزش دیده است صدا می‌زنیم. لایه آخر آن را حذف کرده و تعداد تعداد خروجی های آن را ذخیره می کنیم.

از کلاس Dense دو نمونه می سازیم. همچنین از دو فعال ساز که یکی ReLU و دیگری Softmax است نمونه سازی می کنیم. خروجی هر یک به عنوان ورودی دیگری داده می‌شود. از Categorical_Cross_Entropy_loss و SGD نیز نمونه سازی می‌کنیم.

مقادیر همه پارامتر ها در استخراج کننده ویژگی را ثابت می کنیم و فقط لایه آخر را تغییر می‌دهیم. حال نوبت به حلقه آموزش شبکه می‌رسد. در این حلقه که ۲۰ دور اجرا می‌شود، یک حلقه دیگر وجود دارد که بر روی مجموعه چند تایی از مجموعه داده

۱۰CIFAR است. ورودی و خروجی مورد انتظار را گرفته و به استخراج کننده ویژگی می‌دهیم سپس خروجی آن را به لایه اول دسته بند، خروجی لایه اول را به ورودی فعال ساز اول، خروجی فعال ساز را به وردی لایه دوم، خروجی لایه دوم را به ورودی فعال ساز دوم و خروجی فعال ساز را به نمونه Categorical_Cross_Entropy_loss می‌دهیم. همه این کارها با توابع forward موجود در هر نمونه انجام می‌شود. حال بر روی خروجی‌های به دست آمده و خروجی‌های مورد انتظار تحلیل انجام می‌دهیم به این صورت که آن‌ها را مقایسه کرده و تعداد خروجی هایی که با خروجی اصلی یکسان است را می‌شماریم. سپس این مقدار گزارش می‌کنیم. بعد از گزارش دادن این مقادیر نوبت به مرحله اصلی برای آموزش شبکه می‌رسد که باید خروجی‌ها را رو به عقب به شبکه داده که طبق آنها پارامتر ها را تغییر دهیم. پس همان کاری که در مرحله قبل انجام دادیم را به صورت برعکس انجام می‌دهیم و در آخر با استفاده از کاهش شیب تصادفی این مقادیر را به روز رسانی می‌کنیم.

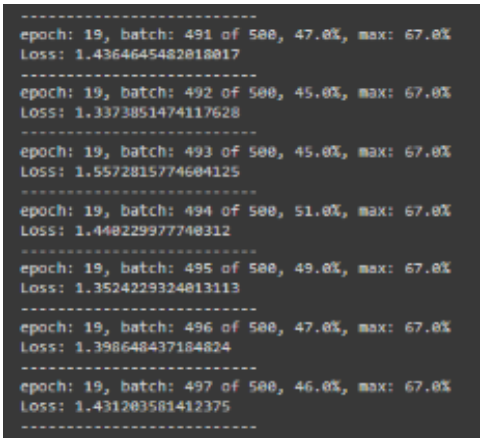
بعد از مرحله آموزش نوبت به ارزیابی این شبکه می‌رسد. ارزیابی همان آموزش است ولی قسمت به‌روزرسانی را ندارد.

پس از به پایان رسیدن مرحله ارزیابی، اطلاعاتی که داریم را با نمودار به تصویر می‌کشیم که در مرحله بعد آمده است.

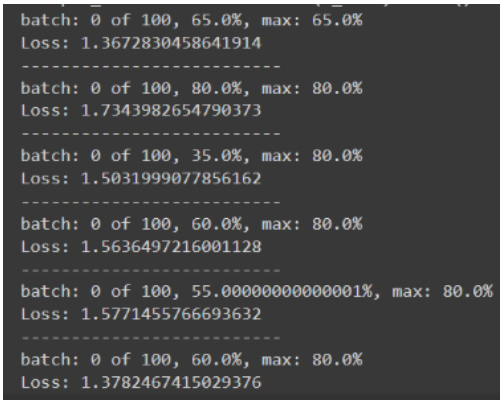
نتایج به دست آمده:

نتایج به دست آمده از انجام این پروژه به صورت عکس آورده شده است.

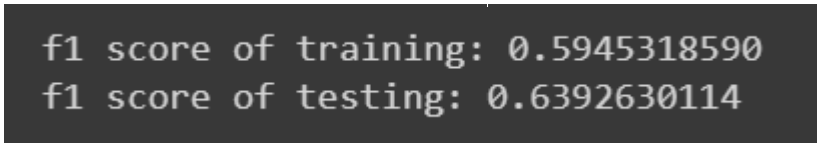
نمودار های اجرا با سایز ورودی ۱۰۰ تایی:



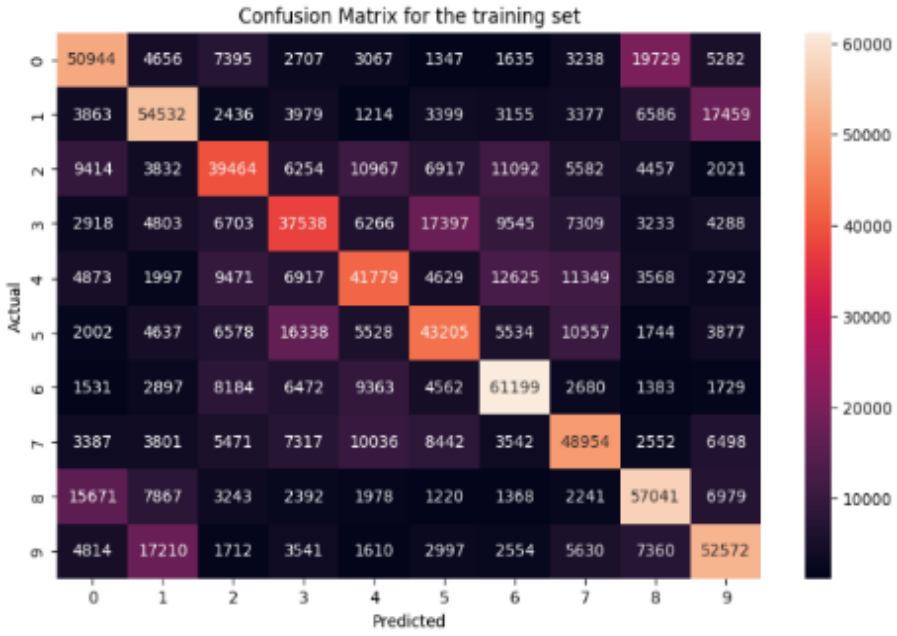
شکل۱: اطلاعات چاپ شده در فرایند آموزش



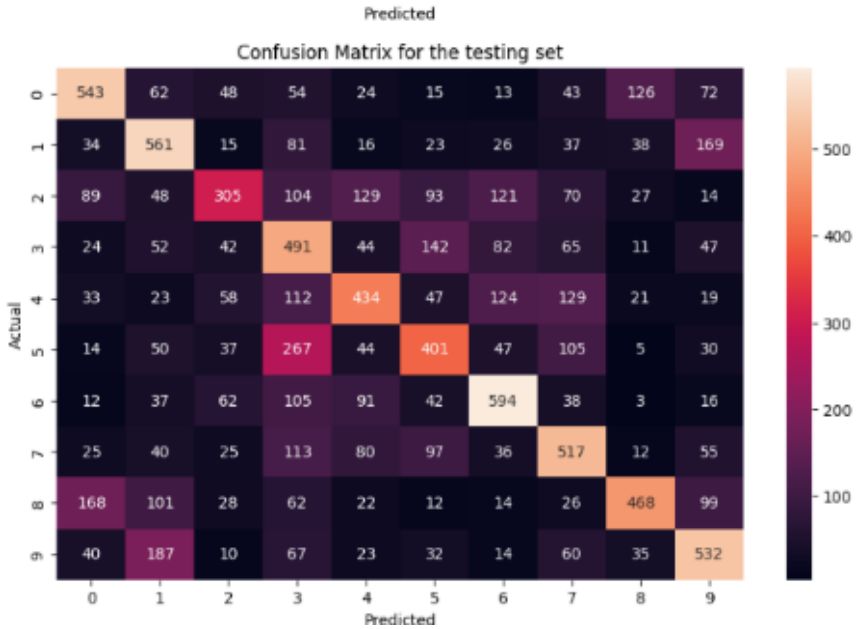
شکل۲: اطلاعات چاپ شده در فرایند ارزیابی



شکل۳: اطلاعات چاپ شده پس از اجرای کامل



شکل۵: نمودار رسم شده از ماتریس درهم‌ریختگی آموزش



شکل٦: نمودار رسم شده از ماتریس درهم‌ریختگی آموزش

تحلیل نتایج:

با توجه به نمودار ها مشاهده می‌شود که نمودار آموزش و آزمایش به درستی تا حد قابل قبولی توانایی شناسایی کلاس‌ها را دارند. قطر مورب اصلی این نمودار ها میزان شناسایی درست هر کلاس می‌باشد.

پیشنهاد:

می‌توان با استفاده از کم و زیاد کردن اندازه ورودی و همچنین تعداد دور اجرای ورودی‌ها، دقت شبکه را افزایش داد

قسمت دوم :

مقدمه:

این کد یک مثال از یادگیری ژنتیکی را بر روی داده های CIFAR-۱۰ اجرا می کند. یادگیری ژنتیکی یک الگوریتم بهینه سازی است که با شبیه سازی فرآیند تکامل زندگی، جهت یافتن بهینه ترین حالت پارامترها برای شبکه های عصبی اعمال می شود. در این کد ابتدا یک شبکه عصبی با معماری سفارشی، بر روی داده های CIFAR-۱۰ آموزش داده می شود، سپس یک الگوریتم یادگیری ژنتیکی برای بهبود عملکرد شبکه به کار گرفته می شود. الگوریتم یادگیری ژنتیکی با استفاده از مفاهیم ژنتیکی، مانند انتخاب، ترکیب و جایگزینی فرزند، جهت پیدا کردن بهترین معماری از نظر دقت کار می کند. این الگوریتم در هر نسل معماری های مختلف را تولید و بررسی می کند و بهترین معماری را به عنوان معماری اصلی انتخاب می کند، و این فرآیند تا جایی ادامه می یابد که دقت شبکه به حد مطلوب برسد.

روش انجام آزمایش :

در ابتدا کتابخانه ها و وابستگی های مورد نیاز را به فایل پایتون اضافه می کنیم :

```
import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np
import pandas as pd

from torchvision import datasets, transforms
```

```
# Load CIFAR-10 dataset
train_dataset=datasets.CIFAR10(root='./data',train=True,  download=True,
transform=transforms.ToTensor())
test_dataset=datasets.CIFAR10(root='./data',train=False,  download=True,
transform=transforms.ToTensor())

# Prepare dataloaders
train_loader =torch.utils.data.DataLoader(train_dataset,  batch_size=64,
shuffle=True)
test_loader  =  torch.utils.data.DataLoader(test_dataset,  batch_size=64,
shuffle=False)
```

این قسمت ابتدای کد، داده های CIFAR-۱۰ را با استفاده از کتابخانه PyTorch بارگیری می کند. سپس با استفاده از تابع transforms.ToTensor()، تصاویر از فرمت اولیه یعنی NumPy به فرمت تانسور PyTorch تبدیل می شوند. سپس داده ها به دو بخش آموزش و تست تقسیم شده، که هر کدام در قسمتی به نام dataloader ذخیره می شوند. پیاده سازی dataloader متناظر با داده های آموزش و تست است، که فرآیند چرخه ای بارگذاری داده ها را مدیریت می کند. هر بار به صورت دسته ای (batch) داده ها را load می‌کند و این کار را برای دست‌کاری شبکه انجام می دهد.

```
# Define the model architecture
class MyNetwork(nn.Module):
    def __init__(self, input_size, hidden_layers=1,
neurons_per_layer=10, activation='ReLU'):
        super(MyNetwork, self).__init__()

        self.input_size = input_size
        self.hidden_layers = hidden_layers
        self.neurons_per_layer = neurons_per_layer
        self.activation = activation

        self.layers = nn.ModuleList()
        self.layers.extend([
            nn.Linear(input_size, neurons_per_layer),
            nn.ReLU()
        ])

        for i in range(hidden_layers - 1):
         ….
```

در این قسمت، معماری شبکه عصبی را به صورت سفارشی تعریف می‌کنیم. با توجه به ورودی input_size، تعداد لایه های مخفی hidden_layers، تعداد نورون‌های هر لایه neurons_per_layer، و تابع فعالیت activation، معماری شبکه سفارشی ساخته می‌شود.

در ابتدا، یک شبکه عصبی با لایه ورودی و لایه خروجی ساخته می‌شود، از آن‌جایی که تعداد لایه های مخفی hidden_layers یک یا بیش‌تر است، با استفاده از پیمایش یک حلقه for، لایه های مخفی شبکه به تعداد مشخص شده تولید و به شبکه اضافه می شود. سپس لایه خروجی شبکه را به بعد ۱۰ تنظیم می کنیم.

در این مدل، فقط دو تابع فعالیت، ReLU و Sigmoid، پشتیبانی می شوند. تابع forward نیز با استفاده از تمام لایه های ایجاد شده، فرایند یادگیری را انجام می‌دهد و output شبکه را ارائه می‌دهد.

```python
# Define testing function
def test(model, test_loader, criterion):
    model.eval()

    test_loss = 0
    test_correct = 0

    with torch.no_grad():
        for data, targets in test_loader:
            outputs = model(data)
            test_loss += criterion(outputs, targets).item()
            _, predictions = torch.max(outputs.data, 1)
            test_correct += (predictions == targets).sum().item()

    # Calculate testing accuracy and loss
    test_accuracy = 100. * test_correct / len(test_loader.dataset)
    test_loss /= len(test_loader.dataset)

    return test_accuracy, test_loss
```

این قسمت کد، تابع آزمایش مدل است که با دادن داده‌های آزمایشی به مدل، مقدار داده‌های خروجی را محاسبه کرده و با استفاده از تابع خطای مشخص شده، خطای مدل را محاسبه می کند. سپس دقت و خطا روی داده های آزمایشی را محاسبه و به عنوان خروجی تابع ارائه می‌دهد. این تابع در حالت آزمایشی (eval mode)، یعنی بدون به روزرسانی وزن‌ها کار می‌کند و بدون محاسبه گرادیان.

```
# Define genetic algorithm functions
def generate_individual():
    # Hidden layer count
    num_layers = np.random.randint(3)

    # Neuron count per layer
    neurons_per_layer = np.random.choice([10, 20, 30])

    # Activation function
    activation_fn = np.random.choice(['ReLU', 'Sigmoid'])

    return [num_layers, neurons_per_layer, activation_fn]
```

این تابع تولید کننده جدید به صورت تصادفی برای استفاده در الگوریتم ژنتیک است. این مورد جدید شامل سه متغیر تعداد لایه‌های پنهان، تعداد نورون‌های هر لایه و تابع فعال‌سازی است. بعد از تولید فرد جدید، آن را بازمی‌گردانده و برای استفاده در الگوریتم ژنتیک درنظر می‌گیریم.

```
def generate_population(population_size):
    population = []
    for i in range(population_size):
        population.append(generate_individual())
    return population
```

این تابع یک جمعیت را با اندازه ی population_size تولید می‌کند. برای تولید هر یکی از اعضای این جمعیت از تابع generate_individual استفاده می‌شود. هر بار که generate_individual صدا زده می‌شود، یک فرد جدید با تعداد لایه‌ها، تعداد نورون‌های هر لایه و تابع فعال‌سازی تصادفی تولید می‌شود. پس این تابع نیز یک جمعیت تصادفی از افراد برای استفاده در الگوریتم ژنتیک تولید می‌کند.

```
def fitness(individual, model, train_loader, test_loader, criterion,
epochs=5):
    # Generate a new model based on individual
    num_layers, neurons_per_layer, activation_fn = individual
    model = MyNetwork(input_size=3072,
                      hidden_layers=num_layers,
                      neurons_per_layer=neurons_per_layer,
                      activation=activation_fn)
    optimizer = optim.Adam(model.parameters(), lr=0.001)

….
```

این تابع، سازگاری یک فرد در الگوریتم ژنتیک است. ورودی‌های این تابع عبارتند از:

individual: فردی که می‌خواهیم سازگاری آن را بررسی کنیم.

model: مدلی که قبلاً در این پروژه تعریف شده است و برای تولید مدل جدید از همین مدل استفاده می‌کنیم.

train_loader: داده‌های آموزشی.

test_loader: داده‌های تست.

criterion: تابع هزینه.

ابتدا، برای تولید مدل جدید، تعداد لایه‌ها، تعداد نورون‌های هر لایه و تابع فعال‌سازی را از فرد استخراج می‌کنیم، سپس با استفاده از این اطلاعات، یک مدل جدید با دقت روی داده‌های تست، ساخته و به عنوان مدل جدید در نظر می‌گیریم. در ادامه، مدل جدید ساخته شده به مدت تعداد اپوک‌های مشخص شده، با داده‌های آموزشی آموزش داده می‌شود. سپس دقت و هزینه مدل روی داده‌های آموزشی و تست محاسبه می‌شود. در نهایت، دقت مدل روی داده‌های تست به صورت میانگین دقت‌های ۵ بار اجرا، محاسبه شده و به عنوان سازگاری فرد در نظر گرفته می‌شود. بنابراین، هدف این تابع بررسی دقت مدل با پارامترهای مختلف است و استفاده از آن در الگوریتم ژنتیک، برای یافتن بهترین مدل از میان جمعیت ایجاد شده است.

```
def select_parents(population, k=2):

    parents = []

    fitnesses = {}



    # Calculate fitness for all individuals

    for individual in population:

        fitnesses[str(individual)]    =    fitness(individual,    model,
train_loader, test_loader, criterion)



  …
```

این تابع، از جمعیت تولید شده در الگوریتم ژنتیک، k فرد با بالاترین سازگاری (بر اساس تابع fitness) را جدا می‌کند و به‌عنوان والدین انتخاب می‌کند. ورودی‌های این تابع عبارتند از:

population: جمعیتی از افراد در الگوریتم ژنتیک.

k: تعداد فردی که به عنوان والدین، از جمعیت انتخاب می‌شوند.

ابتدا برای هر فرد در جمعیت، سازگاری آن) بر اساس تابع fitness) محاسبه می‌شود و در یک دیکشنری ذخیره می‌شود. سپس k فرد با بالاترین سازگاری به عنوان والدین انتخاب شده، در لیست parents ذخیره می‌شوند و این لیست به عنوان ورودی برای تولید فرد بعدی در الگوریتم ژنتیک استفاده می‌شود. در نتیجه، هدف این تابع انتخاب والدین برای تولید نسل بعد، بر اساس سازگاری آن‌ها است.

```
def crossover(parents):

    offspring = []

…

def mutation(individual, mutation_rate=0.1):

    num_layers, neurons_per_layer, activation_fn = individual

…

def  evolve(population,  model,  train_loader,  test_loader,  criterion,
popSize=10, k=2, mutation_rate=0.1):

    new_population = []

                                                                    …
```

تابع crossover دو والدین را می‌گیرد و با احتمال ۰.۵ به صورت یکنواخت از یک والدین یا دیگری، هر ژن را به عنوان فرزند در نظر می‌گیرد.

تابع mutation با احتمال mutation_rate یک ژن را به صورت تصادفی جایگزین می‌کند. برای هر کدام از سه ژن (تعداد لایه‌های مخفی، تعداد نورون در لایه‌های مخفی و تابع فعال‌سازی)، احتمال تعیین شده‌ای برای جایگزینی داریم.

تابع evolve یک جمعیت را به کمک تابع select_parents برای انتخاب والدین، تابع crossover برای تولید فرزندان و تابع mutation برای اعمال جهش به هر یک از فرزندان که بازگشتی است؛ تولید کرده و به عنوان جمعیت جدید خروجی می‌دهد.

نتایج به دست آمده:

نتایج به دست آمده به شرح زیر است:

Generation 0

Best individual: [1, 30, 'Sigmoid'], Fitness score: 37.43

Generation 1

Best individual: [1, 30, 'Sigmoid'], Fitness score: 36.78

Generation 2

Best individual: [1, 30, 'Sigmoid'], Fitness score: 36.8

Generation 3

Best individual: [1, 30, 'Sigmoid'], Fitness score: 36.65

Generation 4

Best individual: [2, 30, 'ReLU'], Fitness score: 39.21

Generation 5

Best individual: [1, 30, 'Sigmoid'], Fitness score: 36.21

Generation 6

Best individual: [1, 30, 'Sigmoid'], Fitness score: 35.73

Generation 7

Best individual: [1, 30, 'Sigmoid'], Fitness score: 35.99

Generation 8

Best individual: [1, 30, 'Sigmoid'], Fitness score: 36.98

Generation 9

Best individual: [1, 30, 'ReLU'], Fitness score: 37.34

Best individual: [1, 30, 'ReLU'], Fitness score: 37.34

تحلیل نتایج:

این برنامه، یک الگوریتم ژنتیک برای بهبود معماری شبکه عصبی برای دسته بندی تصاویر CIFAR-۱۰ استفاده شده است. در هر نسل، ابتدا برای تمام فردها دوشاخه سازی انجام شده و بهترین دو فرد برای تولید فرزندان انتخاب شده اند. پس از تولید این فرزندان، با احتمال معینی میزانی از جهش‌ها رخ می‌دهد که باعث تغییر ژنوم آن‌ها می‌شود.

نتایج به دست آمده نشان می‌دهد که در نسل پنج، یک فرد با معماری دو لایه‌ای با ۳۰ نرون در هر لایه با تابع فعال‌سازی ReLU، بهترین معماری برای دسته بندی تصاویر CIFAR-۱۰ است.

هم‌چنین مشاهده می‌شود که در این آزمایش، با افزایش تعداد نسل‌ها، دقت مدل بهبود یافته است.

پیشنهادها:

افزایش تعداد نسل‌ها؛ با افزایش تعداد نسل‌ها، الگوریتم بهبود یافته و می‌تواند به معماری بهینه‌تری برسد.

انتخاب بهترین تابع فعال‌سازی برای هر لایه؛ برای مثال، ممکن است تابع فعال‌سازی‌ای مثل ELU یا LeakyReLU برای لایه‌های پنهان نتایج بهتری داشته باشد.

افزایش تعداد لایه‌های پنهان؛ می‌توانید تا حد ممکن تعداد لایه‌های پنهان را بیشتر کنیم تا مدل بیشتری قدرت یادگیری داشته باشد و دقت بیشتری در پیش‌بینی داشته باشد.

تغییر در حجم داده‌های آموزش و تست؛ ممکن است مدل را تمرین کرده باشید تا به حداکثر دقتش برسد، اما اگر داده‌های آموزش و تست نمونه اعتباردار کافی نباشند، نتایج بدست آمده به درستی معین نخواهند بود. بنابراین، باید درصدد جمع‌آوری داده‌های بیش‌تر و بیش‌تر از اعتبار سنجی مدل با داده‌های تست استفاده کنید.

قسمت سوم :

روش انجام آزمایش :

هدف این بخش خوشه بندی بردارهای استخراج شده از شبکه ResNet۳٤ است. مجموعه آموزشی CIFAR۱۰ را بدون برچسبهای آن برای این بخش در نظر میگیریم. با استفاده از شبکه از پیش آموزش دیده ResNet۳٤ ،بردار ویژگی را برای هر یک از تصاویر این مجموعه استخراج کرده و سپس آنها را به کمک شبکه SOM که در لایه خروجی ۱۰ نورون دارد، با آموزشی به تعداد ۲۰ دور خوشه بندی می کنیم. در ادامه به توضیح کد زده شده می پردازیم :

در ابتدا کتابخانه ها و وابستگی های مورد نیاز را به فایل پایتون اضافه می کنیم :

```python
import torch
import torchvision.models as models
import numpy as np
import torchvision
from sklearn.cluster import MiniBatchKMeans
import matplotlib.pyplot as plt
```

سپس مدل ResNet۳٤ بدون لایه پایانی را با استفاده از تابع
(models.resnet34(pretrained=True بارگذاری می کنیم و لایه پایانی آن را با استفاده از
torch.nn.Identity() حذف می کنیم:

```python
# Load the pre-trained ResNet34 model without its final layer
model = models.resnet34(pretrained=True)
model.fc = torch.nn.Identity()
```

داده های ۱۰CIFAR را با استفاده از تابع ۱۰torchvision.datasets.CIFAR بارگیری و بدون برچسب در متغیر cifar_dataset ذخیره می کنیم. سپس با استفاده از متد torch.utils.data.DataLoader، داده ها را به صورت دسته ای با اندازه ٦٤ به مدل داده می دهیم:

```
# Load the CIFAR10 dataset without labels
cifar_dataset = torchvision.datasets.CIFAR10(root='./data', train=True, download=True, transform=torchvision.transforms.ToTensor())
cifar_data_loader = torch.utils.data.DataLoader(cifar_dataset, batch_size=64, shuffle=False)
```

بردارهای ویژگی برای هر تصویر در متغیر feature_vectors با استفاده از مدل ۳٤ResNet و بازنمایی برداری ۱۲۸ بعدی به دست می آیند و برای خوشه بندی، از الگوریتم K-Means با تعداد ۱۰ خوشه استفاده می کنیم:

```
# Extract feature vectors for each image in the dataset using the pre-trained ResNet34
feature_vectors = []
for images, _ in cifar_data_loader:
    with torch.no_grad():
        features = model(images)
    feature_vectors.append(features.numpy())
feature_vectors = np.concatenate(feature_vectors)

som_net = MiniBatchKMeans(n_clusters=10, random_state=0, batch_size=100, max_iter=20)
```

و خوشه بندی را با استفاده از بردارهای ویژگی تصاویر در متغیر feature_vectors و با استفاده از مدل K-Means با دستور زیر عملی می کنیم:

```
# Train the SOM network
som_net.fit(feature_vectors)
```

در نهایت، برای نمایش دادن وزن های مختلف برای خوشه‌ها و همچنین برای گزارش شمارش

برچسب های مختلف برای هر خوشه دستورات plt را می‌نویسیم.

```
# Plot the weight vectors for the final feature mapping
plt.imshow(som_net.cluster_centers_.reshape(10, -1))
plt.show()
```

بعد از این مراحل برچسب‌های دادگان CIFAR-۱۰ از دیتاست مورد استفاده را به عنوان برچسب‌های واقعی در نظر می‌گیریم و در متغیر cifar_labels ذخیره می‌کنیم. سپس با استفاده از شبکه‌ی SOM، برچسب خوشه‌بندی برای هر داده‌ی ورودی (تصویر) در متغیر cluster_labels ذخیره می کنیم. و برای هر یک از ۱۰ خوشه‌ی حاصل از خوشه‌بندی SOM، تعداد داده‌هایی که به آن خوشه تعلق می گیرند و همچنین تعداد هر یک از برچسب‌ها در آن خوشه شمرده می‌شود. برای این کار، ابتدا با استفاده از mask حاصل از برچسب‌های خوشه‌بندی، داده‌هایی که به آن خوشه تعلق دارند را از بین تمام داده‌ها انتخاب می‌کنیم. سپس با تهیه‌ی لیست mul که حاصل ضرب عنصر به عنصر mask و برچسب‌های واقعی است، تعداد هر یک از برچسب‌ها در آن خوشه شمرده و در متغیر label_counts ذخیره می‌شود. در نهایت، پراکندگی برچسب‌ها در هر خوشه، با چاپ کردن متغیر label_counts برای هر خوشه‌ی SOM را نمایش می دهیم.

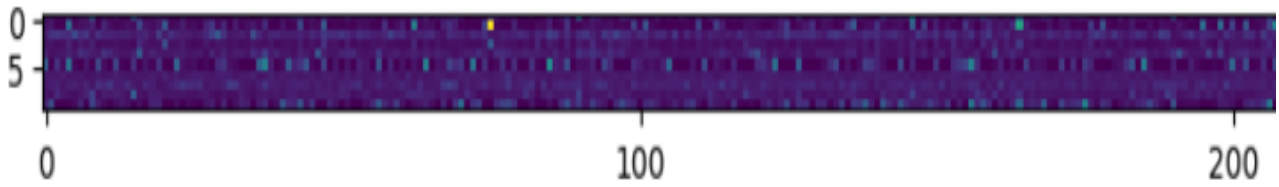
```
# Report the scatter of the different labels of each cluster
cifar_labels = cifar_dataset.targets
cluster_labels = som_net.predict(feature_vectors)

for i in range(10):
    mask = cluster_labels == i
    mul = [a*b for a,b in zip(mask, cifar_labels)]
    label_counts = np.bincount(mul)
    print(f"Cluster {i}: Label Scatter = {label_counts}")
```

نتایج به دست آمده:

نتایج بدست آمده حین اجرای کد و بعد از اتمام و پلات نتایج به شرح زیر است:

```
Cluster 0: Label Scatter = [48053    34   410    95   594   159   341   246    25    43]
Cluster 1: Label Scatter = [49510    23   126     9    57     3    43     8   200    21]
Cluster 2: Label Scatter = [46057   972   269   492   113   430   122   180   577   788]
Cluster 3: Label Scatter = [46837   225   496   151   443   108   310   187   986   257]
Cluster 4: Label Scatter = [47619   407   124   206   167   151   566   164   168   428]
Cluster 5: Label Scatter = [49985     6     0     2     0     1     0     0     0     6]
Cluster 6: Label Scatter = [27431  2182  2644  2836  2575  2701  2522  2481  2656  1972]
Cluster 7: Label Scatter = [45007   491   294   782   312   854   324   877   261   798]
Cluster 8: Label Scatter = [44531   660   630   427   721   593   771   853   127   687]
Cluster 9: Label Scatter = [49970     0     7     0    18     0     1     4]
```



تحلیل نتایج:

طبق نتایج داده ها به ۱۰ دسته تقسیم شده و در هر دسته تعدادی از داده‌ها قرار گرفته است.

پیشنهاد:

تغییر تعداد نورون ها و خوشه ها باعث ایجاد نتایج متفاوتی خواهد شد همچنین میتوانیم از الگوریتم های دیگری جهت استخراج ویژگی ها و همچنین برای خوشه بندی استفاده کنیم.

منابع:

PyTorch

https://chat.openai.com/chat

https://colab.research.google.com

https://stackoverflow.com/questions/64174522/https://stackoverflow.com/questions/58817026

https://pytorch.org/

https://www.tensorflow.org/

https://towardsdatascience.com

https://howsam.org/

https://www.researchgate.net/figure/CIFAR-10-Accuracy-Comparisons-with-Evolutionary-Approaches_tbl2_344176307