به نام خدا

گزارش پروژه شبکه‌های عصبی

درس هوش محاسباتی

دانشگاه اصفهان

دانشکده مهندسی کامپیوتر

استاد درس : دکتر حسین کارشناس

تهیه کنندگان :

علیرضا دستمالچی ساعی، محمد حسین دهقانی، محمد توکلی

اردیبهشت ۱۴۰۲

مقدمه :

در این پروژه قصد داریم تعدادی تصویر را دسته‌بندی کنیم. یکی از رویکردهای مناسب جهت دسته‌بندی تصاویر، استفاده از شبکه‌های عصبی می‌باشد. در قسمت اول این پروژه ابتدا دسته بندی تصاویر توسط شبکه عصبی Resnet34 انجام می‌شود که معیار ما برای سنجش درستی شبکه عصبی آموزش دیده شده توسط خودمان است. گام بعدی پس از انتخاب معماری شبکه، انتخاب وزن های درست برای شبکه می‌باشد که در قسمت اول پروژه نیز، هدف پیدا کردن همین وزن‌ها برای دسته بندی هرچه بهتر تصاویر می‌باشد.

روش انجام :

در فاز ابتدایی پروژه ابتدا نیاز داریم که دیتاست هایی را برای تمرین و ارزیابی مدل حاصل از این قسمت آماده‌سازی کنیم. برای این کار از زبان پایتون و کتابخانه torchvision استفاده می‌کنیم. در این کتابخانه ماژول‌های models, datasets و trasforms و ماژول DataLoader از کتابخانه torch استفاده می‌کنیم.

پیاده سازی این قسمت به صورت زیر انجام می‌شود.

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

```
train_data = datasets.CIFAR10('data', train=True, download=True, transform=transform)
test_data = datasets.CIFAR10('data', train=False, download=True, transform=transform)

train_loader = DataLoader(train_data, batch_size=128, shuffle=True)
test_loader = DataLoader(train_data, batch_size=128, shuffle=False)
```

پس از انجام مراحل بالا train_loader و test_loader آماده استخراج ویژگی‌ها به وسیله شبکه resnet34 هستند و به ترتیب برای آموزش و ارزیابی مدل مورد استفاده قرار خواهند گرفت.

در گام بعدی باید با استفاده از resnet34 بردار ویژگی مدنظر را از داده‌ها استخراج کنیم. از آنجایی که این کار زمان نسبتا زیادی را برای اجرا شدن نیاز دارد از قدرت پردازشی gpu برای انجام محاسبات این بخش استفاده شده‌است. در این گام با حذف لایه آخر شبکه resnet34 بردار ویژگی‌های مدنظر را از داده‌ها استخراج می‌کنیم. مراحل تعریف resnet34، حذف آخرین لایه این شبکه، جلوگیری از آموزش دیدن این شبکه و نهایتا انتقال آن به gpu در تصویر زیر قابل مشاهده است.

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

resnet_model = models.resnet34(pretrained=True)

modules = list(resnet_model.children())[:-1]
resnet_model = torch.nn.Sequential(*modules)

resnet_model.eval()

for param in resnet_model.parameters():
  param.requires_grad = False

resnet_model.to(device)
```

در گام بعدی نوبت به استخراج ویژگی‌ها با استفاده از resnet34 می‌رسد. تصاویر زیر نحوه انجام این کار را نشان می‌دهند.

```
def extract_features(dataset):
  features = []
  labels = []
  with torch.no_grad():
    for images, label in dataset:
      images = images.to(device)
      output = resnet_model(images)
      features.append(output.cpu().numpy())
      labels.append(label.numpy())
  features = np.concatenate(features, axis=0)
  labels = np.concatenate(labels)
  return features, labels
```

```
x_train, y_train = extract_features(train_loader)
x_test, y_test = extract_features(test_loader)
```

```
x_train = x_train.reshape(50000, 512)
y_train = np.array(y_train)
```

پس از استخراج ویژگی‌های مدنظر با استفاده از resnet34 نوبت به پیاده سازی شبکه عصبی خودمان با ویژگی‌های خواسته شده می‌رسد. این شبکه دارای سه لایه می‌باشد که شامل یک لایه ورودی، یک لایه میانی (لایه پنهان) و یک لایه خروجی است. با توجه به ویژگی‌های شبکه عصبی و بردارهای ویژگی استخراج شده پارامترهای شبکه عصبی را مقداردهی اولیه می‌کنیم.

```
def init_params():
    W1 = np.random.rand(20, 512) - 0.5
    b1 = np.random.rand(20, 1)  - 0.5
    W2 = np.random.rand(10, 20) - 0.5
    b2 = np.random.rand(10, 1) - 0.5
    return W1, b1, W2, b2
```

سایر توابعی که برای پیاده سازی این شبکه عصبی نیاز داریم عبارتند از تابع فعال‌ساز ReLU، تابع softmax، تابع one_hot و ... می‌باشند که پیاده سازی آنها را در تصاویر زیر مشاهده می‌کنید.

```
def ReLU(Z):
    return np.maximum(Z, 0)

def softmax(Z):
    A = np.exp(Z) / sum(np.exp(Z))
    return A
```

```python
def ReLU_deriv(Z):
    return Z > 0

def one_hot(Y):
    one_hot_Y = np.zeros((Y.size, Y.max() + 1))
    one_hot_Y[np.arange(Y.size), Y] = 1
    one_hot_Y = one_hot_Y.T
    return one_hot_Y
```

علاوه بر توابع بالا به توابع forward_prop و backward_prop برای پیمایش شبکه و به دست آوردن حاصل و اصلاح وزن های شبکه و تابع update_params هم نیاز داریم که برای به‌روزرسانی پارامترهای شبکه کاربرد دارد. پیاده‌سازی این توابع را در تصاویر زیر مشاهده می‌کنید.

```python
def forward_prop(W1, b1, W2, b2, X):
    Z1 = W1.dot(X) + b1
    A1 = ReLU(Z1)
    Z2 = W2.dot(A1) + b2
    A2 = softmax(Z2)
    return Z1, A1, Z2, A2
```

```python
def backward_prop(Z1, A1, Z2, A2, W1, W2, X, Y):
    m = 50000
    one_hot_Y = one_hot(Y)
    dZ2 = A2 - one_hot_Y
    dW2 = 1 / m * dZ2.dot(A1.T)
    db2 = 1 / m * np.sum(dZ2)
    dZ1 = W2.T.dot(dZ2) * ReLU_deriv(Z1)
    dW1 = 1 / m * dZ1.dot(X.T)
    db1 = 1 / m * np.sum(dZ1)
    return dW1, db1, dW2, db2

def update_params(W1, b1, W2, b2, dW1, db1, dW2, db2, alpha):
    W1 = W1 - alpha * dW1
    b1 = b1 - alpha * db1
    W2 = W2 - alpha * dW2
    b2 = b2 - alpha * db2
    return W1, b1, W2, b2
```

پس از هربار پیمایش شبکه از لایه ورودی به سمت لایه خروجی و به دست آمدن خروجی و میزان خطای شبکه با استفاده از الگوریتم پس انتشار خطا اقدام به به‌روزرسانی وزن‌های شبکه می‌کنیم تا با گذشت هر مرحله خطای به دست آمده کمتر و کمتر شود و مدلی با دقت بالا داشته باشیم.

در نهایت با استفاده از تابع SGD به آموزش و ارزیابی شبکه می‌پردازیم. این تابع با ترکیب توابع تعریف شده قبلی ابتدا شبکه را آموزش داده و پس از هر مرتبه به‌روزرسانی وزن‌ها و بایاس‌های شبکه دقت شبکه را مورد ارزیابی قرار می‌دهد و خروجی ها را به فرمت مناسبی نمایش می‌دهد.

پیاده سازی تابع SGD و سایر توابع مربوطه را در تصویر زیر مشاهده می‌کنید.

```python
def get_predictions(A2):
    return np.argmax(A2, 0)

def get_accuracy(predictions, Y):
    print(predictions, Y)
    return np.sum(predictions == Y) / Y.size

def stochastic_gradient_descent(X, Y, alpha, iterations):
    W1, b1, W2, b2 = init_params()
    for i in range(iterations):
        Z1, A1, Z2, A2 = forward_prop(W1, b1, W2, b2, X)
        dW1, db1, dW2, db2 = backward_prop(Z1, A1, Z2, A2, W1, W2, X, Y)
        W1, b1, W2, b2 = update_params(W1, b1, W2, b2, dW1, db1, dW2, db2, alpha)
        if i % 50 == 0:
            print("Iteration: ", i)
            predictions = get_predictions(A2)
            print(get_accuracy(predictions, Y))
    return W1, b1, W2, b2
```

فراخوانی تابع SGD بر روی داده‌های تمرینی با ضریب یادگیری و تعداد دور مشخص.

```python
W1, b1, W2, b2 = stochastic_gradient_descent(x_train.T, y_train, 0.001, 20)
```

در پایان فاز اول پروژه ماتریس درهم‌ریختگی و امتیاز F1 مدل آموزش داده شده توسط قطعه کد زیر محاسبه می‌شوند.

```python
predictions_train = get_predictions(forward_prop(W1, b1, W2, b2, x_train.T)[-1])
cm_train = confusion_matrix(y_train, predictions_train)
f1_train = f1_score(y_train, predictions_train, average='weighted')
print("Training confusion matrix:\n", cm_train)
print("Training F1 score:", f1_train)

predictions_test = get_predictions(forward_prop(W1, b1, W2, b2, x_test.T)[-1])
cm_test = confusion_matrix(y_test, predictions_test)
f1_test = f1_score(y_test, predictions_test, average='weighted')
print("\nTest confusion matrix:\n", cm_test)
print("Test F1 score:", f1_test)
```

نتایج اجرا برای ۲۰ دور و ضریب یادگیری داده شده به این شرح است:

```
Iteration:  0
[5 4 3 ... 5 6 1] [6 9 9 ... 3 7 7]
Accuracy: 7.201999999999999 %
Iteration:  1
[5 2 4 ... 5 6 1] [6 9 9 ... 3 7 7]
Accuracy: 8.584 %
```

```
Iteration:  2
[9 2 4 ... 5 6 1] [6 9 9 ... 3 7 7]
Accuracy: 9.628 %
Iteration:  3
[9 2 4 ... 5 6 1] [6 9 9 ... 3 7 7]
Accuracy: 10.372 %
Iteration:  4
[9 2 4 ... 5 6 1] [6 9 9 ... 3 7 7]
Accuracy: 11.042 %
Iteration:  5
[9 2 4 ... 5 6 1] [6 9 9 ... 3 7 7]
Accuracy: 11.612 %
Iteration:  6
[9 2 4 ... 5 6 1] [6 9 9 ... 3 7 7]
Accuracy: 12.24 %
Iteration:  7
[9 2 4 ... 5 6 1] [6 9 9 ... 3 7 7]
Accuracy: 12.844 %
Iteration:  8
[9 2 4 ... 0 6 1] [6 9 9 ... 3 7 7]
Accuracy: 13.446 %
Iteration:  9
[9 2 4 ... 0 6 1] [6 9 9 ... 3 7 7]
Accuracy: 14.084 %
Iteration:  10
[9 2 4 ... 0 6 1] [6 9 9 ... 3 7 7]
Accuracy: 14.734 %
Iteration:  11
[9 2 4 ... 0 6 1] [6 9 9 ... 3 7 7]
Accuracy: 15.354 %
Iteration:  12
[9 2 4 ... 0 6 1] [6 9 9 ... 3 7 7]
Accuracy: 15.994 %
Iteration:  13
[9 2 4 ... 0 6 1] [6 9 9 ... 3 7 7]
Accuracy: 16.636 %
Iteration:  14
[9 2 4 ... 0 6 1] [6 9 9 ... 3 7 7]
Accuracy: 17.276 %
Iteration:  15
[9 2 4 ... 0 6 1] [6 9 9 ... 3 7 7]
Accuracy: 17.942 %
Iteration:  16
[9 2 4 ... 0 6 1] [6 9 9 ... 3 7 7]
Accuracy: 18.58 %
Iteration:  17
[9 2 4 ... 0 6 1] [6 9 9 ... 3 7 7]
Accuracy: 19.28 %
Iteration:  18
[9 2 4 ... 0 6 1] [6 9 9 ... 3 7 7]
Accuracy: 19.95 %
Iteration:  19
[9 2 4 ... 0 6 1] [6 9 9 ... 3 7 7]
Accuracy: 20.62 %
```

```
Training confusion matrix:
 [[4010   61  235  105  106  102   68  101  653  459]
 [ 204 4148   66   87   57   86   71   85  265  931]
 [ 462   56 2746  439  685  445  358  392   64   53]
 [ 134   68  558 2065  374 1602  330  436   41   32]
 [ 181   40  569  256 3152  282  283  411   65   21]
 [  83   35  511 1033  271 3342  174  438   43   10]
 [  23   39  384  297  201  180 4131   92   26    7]
 [  97   33  284  156  318  321   47 3674   28   42]
 [ 613  136   80   50   80   74   36   55 3899  987]
 [ 277  454   64   60   44   62   29   77  240 3803]]

Training F1 score: 0.6153867036361499

Test confusion matrix:
 [[810  18  90  27  26  26  17  37 231 138]
 [ 49 889  14  20   8  19  13  25  62 901]
 [131  16 600  63 160 102  71 110  16  17]
 [ 33  22 174 439  90 340  70 103  11   8]
 [ 38  10 167  62 635  76  52 100  16   4]
 [ 17  18 143 348  77 620  36 112  14   5]
 [  6  15 121  90  48  47 858  23   9   3]
 [ 28  12  87  41 107  95  15 741  11  35]
 [177  54  30  18  31  29  10  18 582  51]
 [ 80 152  17  14  10  22   7  26  48 634]]

Test F1 score: 0.4910593188355117
```

تحلیل نتایج :

نتایج به دست آمده در مرحله قبل را می‌توان با افزایش تعداد دور به صورت قابل توجهی بهبود داد.

به نتایج زیر که از اجرای همان الگوریتم و صرفا با افزایش تعداد دور به دست آمده‌اند توجه کنید:

```
Iteration: 0
[6 5 4 ... 4 5 8] [6 9 9 ... 1 1 5]
Accuracy: 9.152 %
Iteration: 50
[6 5 4 ... 4 5 8] [6 9 9 ... 1 1 5]
Accuracy: 40.57000000000001 %
Iteration: 100
[6 5 4 ... 4 5 8] [6 9 9 ... 1 1 5]
Accuracy: 45.424 %
Iteration: 150
[6 5 4 ... 4 5 8] [6 9 9 ... 1 1 5]
Accuracy: 48.624 %
Iteration: 200
[6 5 4 ... 4 5 8] [6 9 9 ... 1 1 5]
Accuracy: 51.438 %
Iteration: 250
[6 5 4 ... 4 5 8] [6 9 9 ... 1 1 5]
Accuracy: 53.982 %
Iteration: 300
[6 5 4 ... 4 5 8] [6 9 9 ... 1 1 5]
Accuracy: 56.052 %
Iteration: 350
[6 5 4 ... 4 5 8] [6 9 9 ... 1 1 5]
Accuracy: 58.150000000000006 %
Iteration: 400
[6 5 4 ... 4 5 8] [6 9 9 ... 1 1 5]
Accuracy: 59.82000000000001 %
Iteration: 450
[6 5 4 ... 4 5 8] [6 9 9 ... 1 1 5]
Accuracy: 61.354 %
Iteration: 500
[6 5 4 ... 4 5 8] [6 9 9 ... 1 1 5]
Accuracy: 62.762 %
Iteration: 550
[6 5 4 ... 4 5 8] [6 9 9 ... 1 1 5]
Accuracy: 63.984 %
Iteration: 600
[6 5 4 ... 4 5 8] [6 9 9 ... 1 1 5]
Accuracy: 64.84000000000002 %
Iteration: 650
[6 5 4 ... 4 5 8] [6 9 9 ... 1 1 5]
Accuracy: 65.66000000000001 %
Iteration: 700
[6 5 4 ... 4 5 8] [6 9 9 ... 1 1 5]
Accuracy: 66.48 %
Iteration: 750
[6 5 4 ... 4 5 8] [6 9 9 ... 1 1 5]
Accuracy: 67.27000000000001 %
Iteration: 800
[6 5 4 ... 4 5 8] [6 9 9 ... 1 1 5]
Accuracy: 68.096 %
Iteration: 850
[6 5 4 ... 4 5 8] [6 9 9 ... 1 1 5]
Accuracy: 68.874 %
Iteration: 900
[6 5 4 ... 4 5 8] [6 9 9 ... 1 1 5]
Accuracy: 69.582 %
```

```
Iteration: 950
[6 5 4 ... 4 5 8] [6 9 9 ... 1 1 5]
Accuracy: 70.328 %
```

Training confusion matrix:

```
[[1969    0    4   14    6   15   26    4  103   17]
 [   3 2350    1    6    1    7    7    2   38   44]
 [ 159    0 1422   77  120   72  105   73   23    5]
 [  45    3   19 1626   63  181   46   34   25   18]
 [  28    1   25   24 1907   28   58   99   13    3]
 [  34    2   12  134   48 1749   19   56   20   29]
 [  16    2   22   59   27   19 1843    8    4    0]
 [  11    0   10   34   64   28    2 1865    5   22]
 [  29    0    1    6    2    5    4    2 1959   18]
 [  11    5    0    9    1    7    4    7   36 1930]]
```

Training F1 score: 0.701220594044287

Test confusion matrix:

```
[[1263   49  182   75  133  155  149  130  504  360]
 [ 160  838  145  120  143  152  158   73  469  902]
 [ 357   21  358  266  378  395  281  215   92   37]
 [ 180   29  159  361  168  786  184  119   63   51]
 [ 129   16  179  163  674  383  150  248   62   16]
 [ 102   22  130  384  134  871  128  135   46    8]
 [  60   55  146  205  246  174 1090   32   24    8]
 [ 139   41  152  206  374  437   88  467   58   18]
 [ 288   83   67   77   57   47   44   34 1085  276]
 [ 193  147   51  106   45  112   60   73  191 1022]]
```

Test F1 score: 0.3086479529518663

جست و جوی معماری شبکه عصبی (NAS)

در این بخش از الگوریتم‌های تکاملی برای پیدا کردن بهترین معماری استفاده شده است. بدین صورت که با اندازه جمعیت ۱۰ معماری به اندازه ۱۰ نسل جلو رفته و درنهایت بهترین معماری با بهترین **fitness** را انتخاب می‌کنیم.

کلاس **NAS** هنگام ساخته شدن مقادیر فضای جست و جو را در خود ذخیره می‌کند:

```
class NAS:
    def __init__(self) -> None:
        self.search_space = {
            'num_layers': [1, 2, 3],
            'layer_sizes': [64, 128, 256, 512],
            'activations': ['ReLU', 'Sigmoid'],
            'feature_extractor': [
                nn.Sequential(*list(models.resnet18().children())[:-1]),
                nn.Sequential(*list(models.resnet34().children())[:-1]),
                nn.Sequential(*list(models.vgg11().features.children())[:-1])
            ]
        }
```

و هنگام اجرای الگوریتم تکاملی، عملگرهای تغییر برای تکامل از این مقادیر استفاده می‌کنند.
سپس برای مقداردهی اولیه از فضای جست و جو به صورت رندوم انتخاب می‌کنیم توسط تابع random_network که با صدا زدن آن یک معماری خروجی می‌دهد.

ویژگی‌های الگوریتم تکاملی استفاده شده

برای الگوریتم تکاملی در فاز ۲ پروژه، قسمت‌های مختلف را به صورت زیر پیاده‌سازی کرده-ایم:

۱. **انتخاب:** روش انتخاب مورد استفاده در الگوریتم تکاملی به صورت tournament selection می‌باشد که هربار به صورت رندوم ۳ شبکه را انتخاب و یکی از بهترین آن‌ها را انتخاب می‌کند.

```
def selection(self, population, k=3):
    selected = []
    while len(selected) < len(population):
        individuals = random.sample(population, k)
        print(individuals)
        fittest = max(individuals, key=lambda network: self.evaluate(network))
        selected.append(fittest)

    return selected
```

۲. **بازترکیب:** روش بازترکیب به کار گرفته شده بدین صورت است که از crossover برا تغییر استخراج کننده ویژگی‌ها و لایه‌ها و توابع فعال‌سازی آنها استفاده شده است. تمامی احتمال های انتخاب شده برای بازترکیب ۰/۵ می‌باشد:

```python
def crossover(self, parents):
    offspring = []
    for i in range(len(parents)):
        parent1 = parents[i]
        parent2 = parents[(i+1)%len(parents)]

        feature_extractor1, network1 = parent1
        feature_extractor2, network2 = parent2

        if random.random() < 0.5:
            feature_extractor = feature_extractor1
        else:
            feature_extractor = feature_extractor2

        layer_sizes = []
        activations = []

        for j in range(min(len(network1.layer_sizes), len(network2.layer_sizes))):
            if random.random() < 0.5:
                layer_sizes.append(network1.layer_sizes[j])
            else:
                layer_sizes.append(network2.layer_sizes[j])

        for j in range(min(len(network1.activations), len(network2.activations))):
            if random.random() < 0.5:
                activations.append(network1.activations[j])
            else:
                activations.append(network2.activations[j])

        num_layers = len(layer_sizes)

        input_size = 512
        output_size = 10
        offspring_network = MLP(input_size, output_size, layer_sizes, activations, loss_function="cross_entropy", learning_rate=0.001)
        offspring.append((feature_extractor, offspring_network))

        return offspring
```

۳. **جهش:** برای عملگر جهش نیز با احتمال ۰/۱ متغیرهای فضای جست و جو تغییر پیدا می‌کند:

```python
def mutation(self, offspring, mutation_rate=0.1):
    mutated_offspring = []
    for i in range(len(offspring)):
        feature_extractor, network = offspring[i]

        for j in range(len(network.layer_sizes)):
            if random.random() < mutation_rate:
                network.layer_sizes[j] = random.choice(self.search_space['layer_sizes'])

        for j in range(len(network.activations)):
            if random.random() < mutation_rate:
                network.activations[j] = random.choice(self.search_space['activations'])

        input_size = 512
        output_size = 10
        mutated_network = MLP(input_size, output_size, network.layer_sizes, network.activations, loss_function="cross_entropy", learning_rate=0.
        mutated_offspring.append((feature_extractor, mutated_network))

    return mutated_offspring
```

۴. **جای‌گذاری:** در عملیات جای‌گذاری، از ساده‌ترین روش استفاده شده و بدین گونه است که ابتدا شبکه‌ها براساس fitness خود مرتب شده، سپس از میان آنها به تعداد اندازه جمعیت بهترین آنهارا انتخاب می‌کنیم:

```
def replacement(self, population, offspring):
    combined_population = population + offspring
    sorted_networks = sorted(combined_population, key=lambda x: self.evaluate(x), reverse=True)
    population = sorted_networks[:len(population)]

    return population
```

۵. **اجرا:** این تابع الگوریتم تکاملی را با توجه به آرگومان‌های پاس داده شده به تابع اجرا می‌کند. که عبارتند از تعداد نسل‌ها و تعداد جمعیت. این تابع پس از اتمام اجرا، نتیجه‌ای از عملکرد خود را چاپ می‌کند:

```
def run(self, generations=10, population_size=10):

    population = [self.random_network() for _ in range(population_size)]

    for generation in range(generations):
        print(f'--- Generation {generation + 1} ---')

        parents = self.selection(population)

        offspring = self.crossover(parents)

        offspring = self.mutation(offspring)        Alireza Saei, 8 hours ago

        population = self.replacement(population, offspring)

    report = self.evaluate(population[0])
    print(f'Report: {report}')
```

۶. **برازندگی:** درنهایت به قسمت Evaluation الگوریتم تکاملی خود می‌رسیم که این تابع با گرفتن مقادیر ورودی و خروجی و اجرا به اندازه ۵ بار با لیست‌های مختلف برای بدست آوردن برازندگی، نتیجه نهایی را به صورت میانگین برازندگی هر ۵ بار ارسال می‌کند:

```python
def evaluate(self, network: MLP, X=x_train_features, Y=y_train):
    fitness_list = []
    n_samples = X.shape[0]
    indices = list(range(n_samples))
    random.shuffle(indices)
    subsample_size = int(n_samples / 5)

    for i in range(5):
        subset_indices = indices[i*subsample_size:(i+1)*subsample_size]
        X_subset = X[subset_indices]
        Y_subset = Y[subset_indices]

        y_hat = network.forward(X_subset)
        loss = network.loss.forward(y_hat, Y_subset)
        fitness = 1.0 / (1.0 + loss)
        fitness_list.append(fitness)

    avg_fitness = sum(fitness_list) / len(fitness_list)
    return avg_fitness
```

تحلیل نتایج

با اجرای الگوریتم تکاملی و انتخاب درست هایپرپارامترها و ... معماری‌های انتخاب شده به سوی دقت بالاتر و loss کمتر حرکت می‌کند و در نهایت معماری‌ای با بهترین استخراج کننده ویژگی، بهترین تعداد لایه‌ها و بهترین تعداد نورون هر لایه همراه با توتاع فعال‌سازی آنها.
بدلیل مدت زمان اجرای بلند نمودارها پس اجرا شدن روی سرورهای گوگل قرار گرفته می‌شوند.

خوشه‌بندی عکس‌ها با SOM

در این بخش، بدون درنظر گرفتن لیبل‌های دیتاست و صرفا براساس ویژگی‌های بدست آمده، اقدادم به خوشه بندی می‌کنیم. برای این کار از SOM استفاده شده است. بدین صورت که کلاس SOM به صورت زیر مقداردهی می‌شود (موقع initialize کردن):

```
Alireza Saei, 13 hours ago | 1 author (Alireza Saei)
class SOM:
    def __init__(self, x, y, input_dim, sigma=1.0, lr=0.1):
        self.x = x
        self.y = y
        self.input_dim = input_dim
        self.sigma = sigma      Alireza Saei, 13 hours ago • Added
        self.lr = lr
        self.weights = np.random.rand(x, y, input_dim)
```

سپس از توابع دیگر که برای خوشه بندی استفاده شده اند:

neighborhood اول تابع همسایگی گاوسی را با مرکزیت مختصات (c,r) محاسبه می کند.
این تابع در تابع دوم، Train(self, data, epochs) استفاده می شود که الگوریتم (SOM) را برای یادگیری نمایش داده های با ابعاد بالا با ابعاد پایین پیاده سازی می کند.

```
def neighborhood(self, c, r):
    d = np.sqrt((np.arange(self.x) - c)**2 + (np.arange(self.y) - r)**2)
    return np.exp(-(d**2) / (2 * self.sigma**2))
```

تابع "train" با استفاده از داده ورودی، شبکه‌ی SOM را برای تعداد مشخصی از دوره‌ها آموزش می‌دهد. در ادامه جزئیات عملکرد این تابع آمده است:

۱. داده ورودی با استفاده از StandardScaler نرمال سازی می‌شود.

۲. وزن‌های شبکه با استفاده از الگوریتم K-Means مقداردهی اولیه می‌شوند.

۳. برای هر دوره از آموزش، مراحل زیر اجرا می‌شود:

- داده‌های ورودی را به صورت تصادفی مخلوط می‌کنیم.

- نرخ یادگیری و فاصله همسایگی براساس شماره فعلی دوره محاسبه می‌شوند.

- برای هر داده ورودی، واحدی که بیشترین شباهت را با آن دارد (BMU) پیدا می‌شود.

- وزن واحدهای شبکه بر اساس فاصله آن‌ها از BMU و همچنین فاصله همسایگی با BMU بروزرسانی می‌شوند.

- نرخ یادگیری و فاصله همسایگی با توجه به شماره فعلی دوره، کاهش می‌یابد.

- خطای کوانتیزاسیون (QE) محاسبه می‌شود. این مقدار میانگین فاصله بین هر داده ورودی و BMU آن است.

- در صورت ارتباط با معیار متوقف کننده، آموزش در صورت لزوم متوقف می‌شود.

۴. SOM آموزش داده شده، با استفاده از تابع "predict" قابل استفاده است تا بتوانیم BMUs برای داده‌های جدیدی پیش‌بینی کنیم.

۵. تابع "visualize" می‌تواند برای رسم شبکه SOM با برچسب‌های متنی و رنگ‌آمیزی استفاده شود.

در کل، تابع "train" وزن‌های SOM را به‌گونه‌ای بروزرسانی می‌دهد که داده‌های مشابه در همسایگی یکدیگر در شبکه SOM قرار گیرند. با این کار، نگاشتی از فضای بعد بالای داده ورودی به یک شبکه دو بعدی (یا چند بعدی) کوچکتر به‌وجود می‌آید که به تصویر سازی و تحلیل الگوهای داده امکان می‌دهد.

تابع "predict" در کلاس SOM یک مجموعه از داده‌های ورودی را گرفته و به ازای هر داده بهترین واحد تطبیق (BMU) را برمی‌گرداند.

در ابتدا، داده‌های ورودی با استفاده از `StandardScaler` نرمال‌سازی می‌شوند. سپس برای هر داده ورودی، BMU را با محاسبه فاصله اقلیدسی بین آن و تمام واحدهای شبکه SOM پیدا می‌کند. BMU به عنوان واحدی تعریف می‌شود که دارای کمترین فاصله اقلیدسی با داده ورودی است.

تابع "predict" مختصات BMU را برای هر داده ورودی در قالب یک تاپل با دو آرایه (یکی برای مختصات x و دیگری برای مختصات y) برمی‌گرداند. اندازه این آرایه‌ها متناظر با تعداد داده‌های ورودی است.

```
def predict(self, data):
    scaler = StandardScaler()
    data = scaler.fit_transform(data)
    bmu_idxs = np.argmin(np.linalg.norm(self.weights.reshape(-1, self.input_dim) - data[:, np.newaxis, :], axis=-1), axis=(1, 2))
    return np.unravel_index(bmu_idxs, (self.x, self.y))


def visualize(self):
    import matplotlib.pyplot as plt
    from mpl_toolkits.axes_grid1 import make_axes_locatable

    fig, ax = plt.subplots(figsize=(self.x, self.y))
    im = ax.imshow(np.zeros((self.x, self.y)), cmap='viridis')

    for i in range(self.x):
        for j in range(self.y):
            c = self.weights[i, j]
            ax.text(j, i, str(np.round(c, 2)),
                    va='center', ha='center')

    divider = make_axes_locatable(ax)
    cax = divider.append_axes('right', size='5%', pad=0.05)
    plt.colorbar(im, cax=cax)
    plt.show()
```

تقسیم وظایف:

- **علیرضا دستمالچی ساعی**
فایل‌های main و SOM و Neural Network و NAS و داک مربوط به بخش های ۲و۳ داک

- **محمدحسین دهقانی**
فایل‌های phase1 و داک مربوط به بخش ۱ داک

- **محمدتوکلی**
قالب‌بندی کلاس‌ها و تمپلیت داک و رفع باگ‌های runtime در کدها