مقدمه:

در این پروژه می‌خواهیم در یک note book و با استفاده از google colab شبکه عصبی ای برای داده‌های داده شده طراحی کنیم و یک مسیله classification را با استفاده از کتاب خانه pytorch حل می کنیم.

پروژه:

ابتدا داده‌ها را لود می‌کنیم و داریم:

```
[25]
class CustomToTensor(object):
    """Convert PIL Images in sample to pytorch Tensors."""

    ## the input image must be grayscaled first
    def __call__(self, image):
        image = np.array(image, dtype=np.float32)
        # numpy image: H x W
        return torch.from_numpy(image)
```

```
[26]
transform = transforms.Compose([transforms.Grayscale(),
                                CustomToTensor()
                               ]
                              ) ## composes multiple transforms into single one

dataset = ImageFolder("/content/drive/My Drive/categorized_products" ,transform=transform)
```

```
[27]
classes = dataset.classes
print(classes)
```

```
['Accessory Gift Set', 'Backpacks', 'Belts', 'Capris', 'Caps', 'Casual Shoes', 'Clutches', 'Cufflinks', 'Deodorant', 'Dresses', 'Dupatta', 'Earrings',
```

که در این جا در ابتدا داده‌ها فقط به صورت grayscale شده‌اند و هنوز نرمالایز نشده‌اند چرا که بعداً می‌توانیم با تقسیم هر پیکسل بر ۲۵۵ داده‌ها را نرمالایز کنیم

سپس از هر کلاس داده شده ۲۰ درصد را برای داده ی تست جدا می‌کنیم یعنی از هر کلاس ۸۰ درصد داده ی آموزش و ۲۰ درصد داده ی تست را خواهیم داشت:

```
[29]
batch_size = 64
validation_split = 0.2


indices = list(range(len(dataset))) # indices of the dataset
print(len(indices))

# TODO: split the dataset into train and test sets randomly with split of 0.2 and assign their indices in the original set to train_indices and test_indices
targets = dataset.targets
train_indices, test_indices = train_test_split(indices, test_size=0.2, train_size=0.8 , stratify=targets)
print(len(train_indices), len(test_indices))

# Creating PT data samplers and loaders:
train_sampler = SubsetRandomSampler(train_indices)
test_sampler = SubsetRandomSampler(test_indices)

train_loader = torch.utils.data.DataLoader(dataset, batch_size=batch_size, sampler=train_sampler, num_workers=128)
test_loader = torch.utils.data.DataLoader(dataset, batch_size=batch_size, sampler=test_sampler, num_workers=128)
```

```
37249
29799 7450
```

```
[28]
print(len(train_loader), len(test_loader))
```

```
466 117
```

همان‌طور که میبینیم در اینجا bach_Size=64 است و تعداد

num_worker = 128 انتخاب شده است تا سرعیت یادگیری و آموزش بیشتر شود

۱)

حال می‌خواهیم ۱۲ تصویر با اسم کلاس‌های آن را نمایش دهیم در اینجا برای این کار داریم:

```python
def imshow(img_array, labels, classes):
    plt.figure(figsize=(12, 2))
    _, axs = plt.subplots(2, 6)
    # axs.set_facecolor('b')
    for i in range(2):
      for j in range(6):
        axs[i][j].imshow(img_array[i * 6 + j], cmap='gray')
        title_obj = axs[i][j].set_title(classes[labels[i * 6 + j]])
        plt.setp(title_obj, color='r')
        axs[i][j].axis('off')
    plt.show()

data_iter = iter(train_loader)
data_iter.next()
images, labels = data_iter.next()

imshow(images[0:12], labels[0:12], classes)
```

که می‌بینیم ۱۲ عکس با اسم کلاس آن‌ها را داریم

(2

حال می‌خواهیم میزان پراکندگی کلاس هایمان را ببینیم برای این کار می‌توانیم داده‌ها را به تست و آموزش جدا کرده و سپس رسم کنیم. برای جدا کردن داده از هر کلاس ۲۰ درصد آن را جدا می‌کنیم و در داده‌های تست قرار می‌دهیم و ۸۰ درصد بقیه آن را برای آموزش استفاده می‌کنیم

برای این کار داریم:

```
batch_size = 64
validation_split = 0.2


indices = list(range(len(dataset))) # indices of the dataset
print(len(indices))

# TODO: split the dataset into train and test sets randomly with split of 0.2 and assign their indices in the original
targets = dataset.targets
train_indices, test_indices = train_test_split(indices, test_size=0.2, train_size=0.8 , stratify=targets)
print(len(train_indices), len(test_indices))

# Creating PT data samplers and loaders:
train_sampler = SubsetRandomSampler(train_indices)
test_sampler = SubsetRandomSampler(test_indices)

train_loader = torch.utils.data.DataLoader(dataset, batch_size=batch_size, sampler=train_sampler, num_workers=128)
test_loader = torch.utils.data.DataLoader(dataset, batch_size=batch_size, sampler=test_sampler, num_workers=128)
```

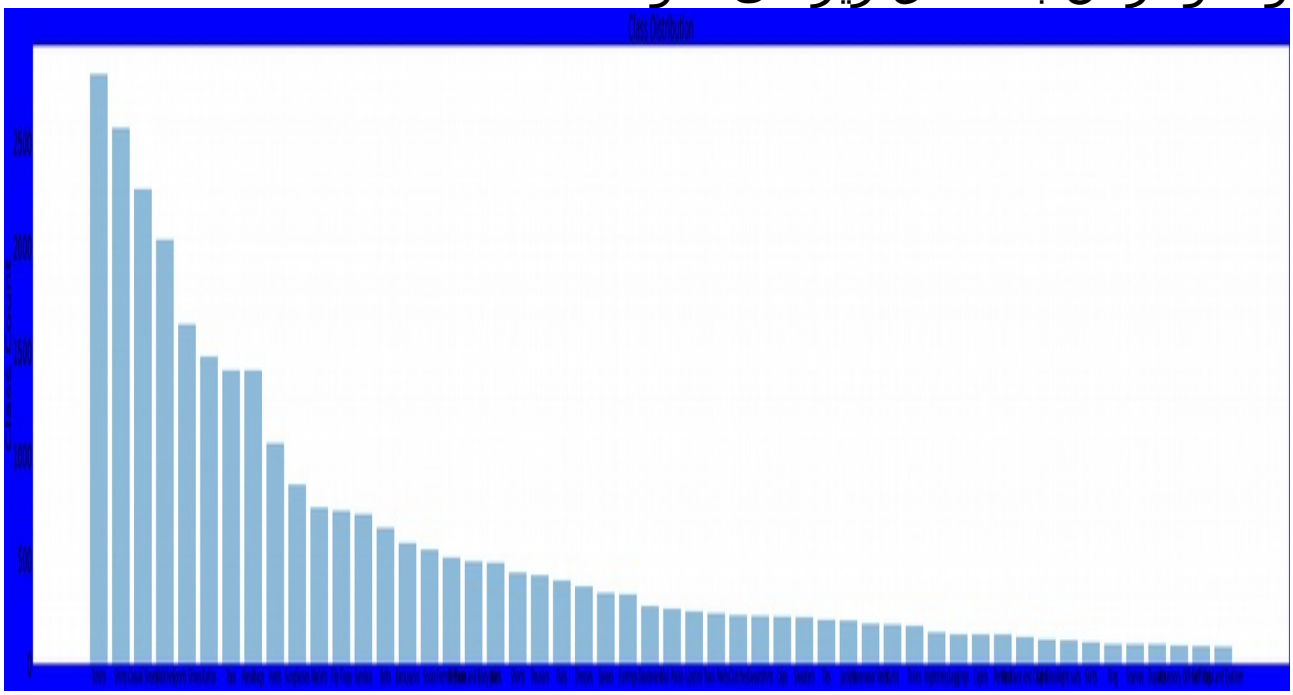
ابتدا مانند بالا کلاس‌ها را جدا می‌کنیم و می‌بینیم که داریم:

```
37249
29799 7450
```

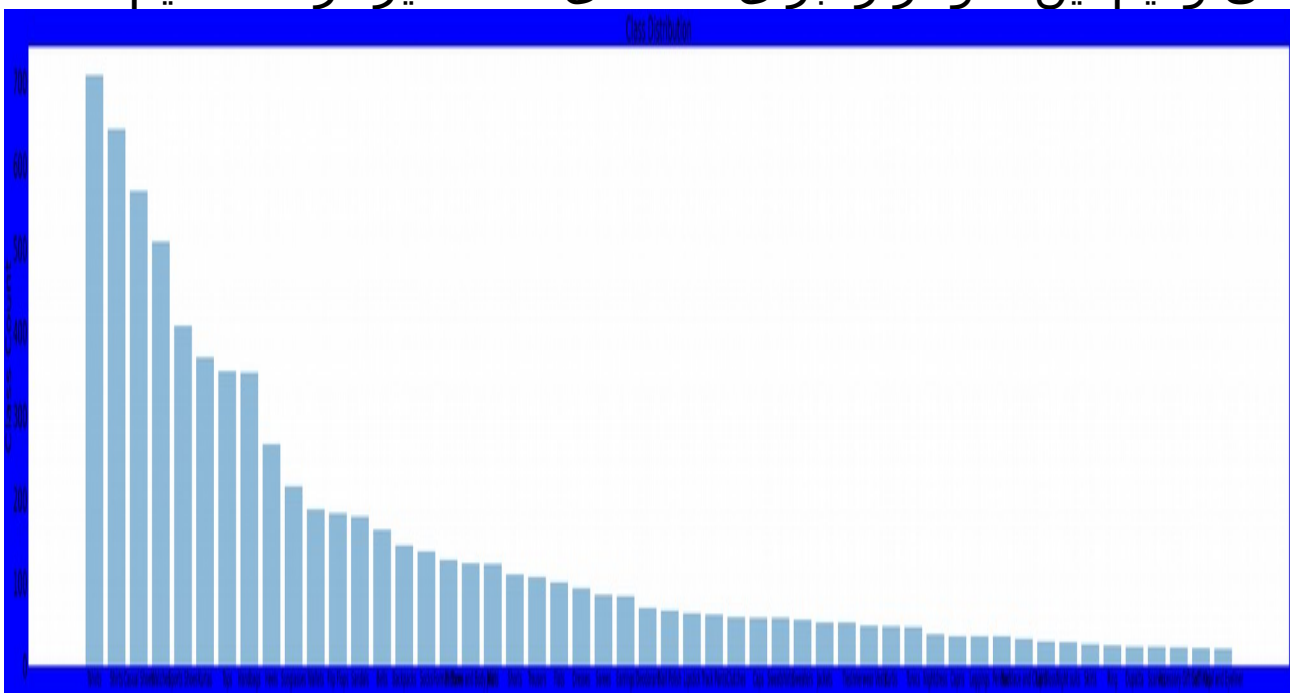
که تعداد کل داده‌ها ۳۷۲۴۹ است و از ۲۹۷۹۹ تا برای آموزش و از ۷۴۵۰ عدد برای تست استفاده می‌کنیم سپس میزان پراکندگی را می‌توانیم مانند زیر ببینیم:

```
For Train:
Accessory Gift Set : 89
Backpacks : 578
Belts : 650
Capris : 140
Caps : 226
Casual Shoes : 2277
Clutches : 232
Cufflinks : 85
Deodorant : 278
Dresses : 371
Dupatta : 93
Earrings : 334
Flats : 400
Flip Flops : 733
Formal Shoes : 510
Handbags : 1406
Heels : 1058
Innerwear Vests : 193
Jackets : 206
Jeans : 486
Kajal and Eyeliner : 82
Kurtas : 1475
Kurtis : 187
Leggings : 142
Lip Gloss : 115
Lipstick : 252
Nail Polish : 263
Necklace and Chains : 128
Night suits : 113
Nightdress : 151
Pendant : 140
Perfume and Body Mist : 490
Ring : 94
Sandals : 718
Sarees : 339
Scarves : 94
Shirts : 2570
Shorts : 438
Skirts : 102
Socks : 549
Sports Shoes : 1628
Sunglasses : 858
Sweaters : 222
Sweatshirts : 228
Ties : 209
Tops : 1409
Track Pants : 242
Trousers : 424
Tshirts : 2827
Tunics : 183
Wallets : 749
Watches : 2033
```

و نمودار آن به شکل زیر می شود:



می‌توانیم این نمودار را برای داده‌های تست نیز درست کنیم:



3)

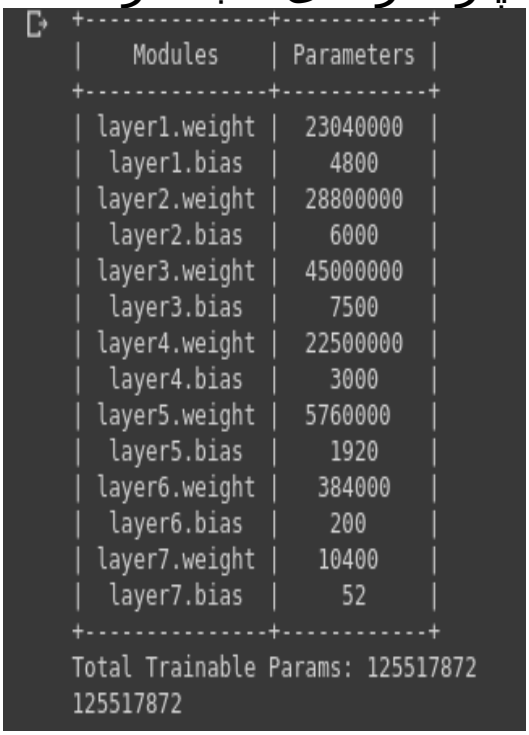
در این قسمت می‌خواهیم با شبکه ای که درست کرده‌ایم و داده‌های آموزش شبکه را آموزش دهیم تا کلاس داده‌ها را تشخیص دهد برای این کار ابتدا یک شبکه مانند زیر می سازیم:

```
self.layer1 = nn.Linear(4800, 4800)
self.act1 = act


self.layer2 = nn.Linear(4800, 6000)
self.act2 = act


self.layer3 = nn.Linear(6000, 7500)
self.act3 = act


self.layer4 = nn.Linear(7500, 3000)
self.act4 = act


self.layer5 = nn.Linear(3000, 1920)
self.act5 = act


self.layer6 = nn.Linear(1920, 200)
self.act6 = act


self.layer7 = nn.Linear(200, class_num)
# self.act3 = F.softmax(dim=1)
```

همانطور که می‌بینیم شبکه دارای 7 لایه با تعداد نورون های متفاوت است و در آخر آن‌ها 200 به ۵۲ بردیم که همان تعداد کلاس‌های ماست

و این شبکه را می‌خواهیم با داده‌های آموزش آموزش دهیم تا بتواند برای آن‌ها خروجی مناسب را پیدا کند

حال می‌توانیم تعداد پارامتر های شبکه را مانند زیر ببینیم:



Modules	Parameters
layer1.weight	23040000
layer1.bias	4800
layer2.weight	28800000
layer2.bias	6000
layer3.weight	45000000
layer3.bias	7500
layer4.weight	22500000
layer4.bias	3000
layer5.weight	5760000
layer5.bias	1920
layer6.weight	384000
layer6.bias	200
layer7.weight	10400
layer7.bias	52

Total Trainable Params: 125517872
125517872

همان‌طور که میبینیم در لایه اول تعداد بایاس ما همان تعداد خروجی است چرا که نود های ما همان نود های خروجی هستند و تعداد پارامتر های وزن ما همان ۴۸۰۰ * ۴۸۰۰ است که نورون های ورودی را به خروجی وصل میکند
برای تمام لایه ها این درست است که تعداد بایاس برابر با تعداد نود خروجی و تعداد پارامتر های ما همان تعداد ضرب تعداد نورون خروجی در نورون ورودی است چرا که هر ۱ وزن ۱ نورون ورودی را به ۱ نورون خروجی وصل می‌کند
و در آخر تعداد کل پارامتر هایی که باید آموزش ببیند درواقع جمع این تعداد پارامتر ها است

برای آموزش شبکه داریم:

```python
def fit(model, train_loader, device, criterion, optimizer, num_epochs=10):

  total_time = 0.
  average_loss_history = []
  for epoch in range(num_epochs):
      train_loss = 0.
      d1 = datetime.now()
      for images, labels in train_loader:

        images = images.to(device)
        labels = labels.to(device)

        # Clear gradients w.r.t. parameters
        optimizer.zero_grad()

        # Forward pass to get output/logits
        outputs = model(images)

        # Calculate Loss: softmax --> cross entropy loss
        loss = criterion(outputs, labels)

        # Getting gradients w.r.t. parameters
        loss.backward()

        # Updating parameters
        optimizer.step()
        train_loss += loss.item()

      average_loss = train_loss / len(train_loader)
      average_loss_history.append(average_loss)
      d2 = datetime.now()
      delta = d2 - d1
      seconds = float(delta.total_seconds())
      total_time += seconds
      print('epoch %d, train_loss: %.6f, time elapsed: %s seconds' % (epoch + 1, average_loss, seconds))
  print('total training time: %.3f minutes' % (total_time / 60))
  return average_loss_history
```

و بعد از اجرای آن داریم:

```
[14]  1 average_loss_history = fit(model, train_loader, device, criterion, optimizer)

epoch 1, train_loss: 2.741918, time elapsed: 94.066557 seconds
epoch 2, train_loss: 1.852217, time elapsed: 46.536453 seconds
epoch 3, train_loss: 1.559757, time elapsed: 49.717179 seconds
epoch 4, train_loss: 1.354699, time elapsed: 47.569328 seconds
epoch 5, train_loss: 1.217645, time elapsed: 47.723529 seconds
epoch 6, train_loss: 1.146557, time elapsed: 47.693737 seconds
epoch 7, train_loss: 1.074283, time elapsed: 47.38962 seconds
epoch 8, train_loss: 1.024752, time elapsed: 47.523909 seconds
epoch 9, train_loss: 0.965116, time elapsed: 47.156072 seconds
epoch 10, train_loss: 0.935261, time elapsed: 48.377509 seconds
total training time: 8.729 minutes
```

می‌بینیم که مقدار loss در هر ایپاک کمتر شده است و تقریباً در ایپاک ۱۰ به یک مقدار ثابت همگرا شده است آموزش شبکه ۸ دقیقه طول کشیده است

در این قسمت داده‌های ما نرمالایز نشده بوده‌اند به همین دلیل نیاز به نورون زیاد یا لایه ی زیاد بوده‌ایم تا بتواند این داده‌ها را با مقادیر زیاد هندل کند چرا که مقدار وزن های شبکه بسیار زیاد می‌شود و بعد از مدتی دیگر نمی‌تواند با gradient وزن ها را بیشتر از حدی کند به همین دلیل نیاز به لایه ی بیشتر و در کل نورون بیشتر بوده‌ایم

حال اگر همین شبکه را با داده‌های تست و ترین امتحان کنیم یعنی بگزاریم شبکه این داده‌ها را پیش‌بینی کند داریم:

```python
def test_model_accuracy(model, test_loader):
    # Calculate Accuracy
    correct = 0.
    total = 0.
    # Iterate through test dataset
    with torch.no_grad():
      for images, labels in test_loader:
        outputs = model(images.to(device))
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted.to('cpu') == labels).sum().item()

    accuracy = 100 * correct / total
    print('Accuracy: {}%'.format(accuracy))
```

```
[15]  1 print("Test Accuracy is: ")
      2 test_model_accuracy(model, test_loader)
      3 print("Train Accuracy is: ")
      4 test_model_accuracy(model, train_loader)

      Test Accuracy is:
      Accuracy: 67.18120805369128%
      Train Accuracy is:
      Accuracy: 67.9250981576563%
```

می‌بینیم که شبکه به دقت ۶۷ درصد برای داده‌های تست و تقریباً ۶۹ درصد برای داده‌های آموزش رسیده است

می‌توانیم نمودار loss بر ایپاک را نیز رسم کنیم:

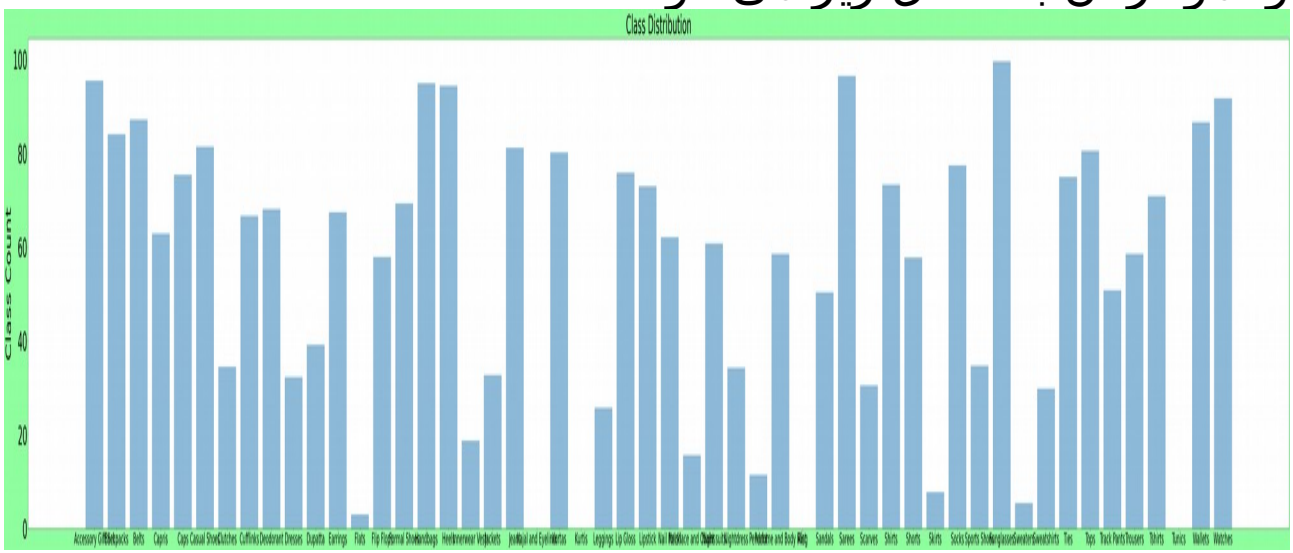
```
[23]  1 fig = plt.figure()
      2 fig.patch.set_facecolor('xkcd:mint green')
      3 plt.plot(average_loss_history)
      4 plt.title("Average Loss")
      5 plt.xlabel("Epoch")
      6 plt.ylabel("Loss")
      7 plt.show()
```

و اگر بخواهیم دقت را به ازای هر کلاس به صورت نمودار میله ای نمایش دهیم داریم:

```
class_correct = list(0. for i in range(len(classes)))
class_total = list(0. for i in range(len(classes)))

for data in test_loader:
    inputs, labels = data
    outputs = model(inputs.to(device))
    _, predicted = torch.max(outputs.data, 1)
    c = (predicted.to('cpu') == labels.data).squeeze()

    # print(labels.data, len(labels.data))
    for i in range(len(labels.data)):
        label = labels.data[i]
        class_correct[label] += c[i]
        class_total[label] += 1

accuracy = []
for i in range(52):
  accuracy.append(100 * class_correct[i].item() / class_total[i])
  print('Accuracy of {} : {} / {} = {:.4f} %'.format(classes[i], class_correct[i], class_total[i], 100 * class_correct[i].item() / class_total[i]))


objects = classes
y_pos = np.arange(len(objects))
performance = accuracy

fig = plt.figure(figsize=(len(classes) + 100, len(classes)/3))
fig.patch.set_facecolor('xkcd:mint green')
plt.bar(y_pos, performance, align='center', alpha=0.5)
plt.yticks(fontsize=50)
plt.xticks(y_pos, objects, fontsize=30)
plt.ylabel('Class Count', fontsize=50)
plt.title('Class Distribution' , fontsize=50)
```

```
Accuracy of Accessory Gift Set : 21.0 / 22.0 = 95.4545 %
Accuracy of Backpacks : 121.0 / 144.0 = 84.0278 %
Accuracy of Belts : 142.0 / 163.0 = 87.1166 %
Accuracy of Capris : 22.0 / 35.0 = 62.8571 %
Accuracy of Caps : 43.0 / 57.0 = 75.4386 %
Accuracy of Casual Shoes : 463.0 / 569.0 = 81.3708 %
Accuracy of Clutches : 20.0 / 58.0 = 34.4828 %
Accuracy of Cufflinks : 14.0 / 21.0 = 66.6667 %
Accuracy of Deodorant : 47.0 / 69.0 = 68.1159 %
Accuracy of Dresses : 30.0 / 93.0 = 32.2581 %
Accuracy of Dupatta : 9.0 / 23.0 = 39.1304 %
Accuracy of Earrings : 56.0 / 83.0 = 67.4699 %
Accuracy of Flats : 3.0 / 100.0 = 3.0000 %
Accuracy of Flip Flops : 106.0 / 183.0 = 57.9235 %
Accuracy of Formal Shoes : 88.0 / 127.0 = 69.2913 %
Accuracy of Handbags : 333.0 / 351.0 = 94.8718 %
Accuracy of Heels : 250.0 / 265.0 = 94.3396 %
Accuracy of Innerwear Vests : 9.0 / 48.0 = 18.7500 %
Accuracy of Jackets : 17.0 / 52.0 = 32.6923 %
Accuracy of Jeans : 99.0 / 122.0 = 81.1475 %
Accuracy of Kajal and Eyeliner : 0.0 / 20.0 = 0.0000 %
Accuracy of Kurtas : 296.0 / 369.0 = 80.2168 %
Accuracy of Kurtis : 0.0 / 47.0 = 0.0000 %
Accuracy of Leggings : 9.0 / 35.0 = 25.7143 %
Accuracy of Lip Gloss : 22.0 / 29.0 = 75.8621 %
Accuracy of Lipstick : 46.0 / 63.0 = 73.0159 %
Accuracy of Nail Polish : 41.0 / 66.0 = 62.1212 %
Accuracy of Necklace and Chains : 5.0 / 32.0 = 15.6250 %
Accuracy of Night suits : 17.0 / 28.0 = 60.7143 %
Accuracy of Nightdress : 13.0 / 38.0 = 34.2105 %
Accuracy of Pendant : 4.0 / 35.0 = 11.4286 %
Accuracy of Perfume and Body Mist : 72.0 / 123.0 = 58.5366 %
Accuracy of Ring : 0.0 / 24.0 = 0.0000 %
Accuracy of Sandals : 90.0 / 179.0 = 50.2793 %
Accuracy of Sarees : 82.0 / 85.0 = 96.4706 %
Accuracy of Scarves : 7.0 / 23.0 = 30.4348 %
Accuracy of Shirts : 471.0 / 643.0 = 73.2504 %
Accuracy of Shorts : 63.0 / 109.0 = 57.7982 %
Accuracy of Skirts : 2.0 / 26.0 = 7.6923 %
Accuracy of Socks : 106.0 / 137.0 = 77.3723 %
Accuracy of Sports Shoes : 141.0 / 407.0 = 34.6437 %
Accuracy of Sunglasses : 214.0 / 215.0 = 99.5349 %
Accuracy of Sweaters : 3.0 / 55.0 = 5.4545 %
Accuracy of Sweatshirts : 17.0 / 57.0 = 29.8246 %
Accuracy of Ties : 39.0 / 52.0 = 75.0000 %
Accuracy of Tops : 284.0 / 353.0 = 80.4533 %
Accuracy of Track Pants : 31.0 / 61.0 = 50.8197 %
Accuracy of Trousers : 62.0 / 106.0 = 58.4906 %
Accuracy of Tshirts : 501.0 / 707.0 = 70.8628 %
Accuracy of Tunics : 0.0 / 46.0 = 0.0000 %
Accuracy of Wallets : 162.0 / 187.0 = 86.6310 %
Accuracy of Watches : 466.0 / 508.0 = 91.7323 %
```

و نمودار آن به شکل زیر می‌شود



۴)

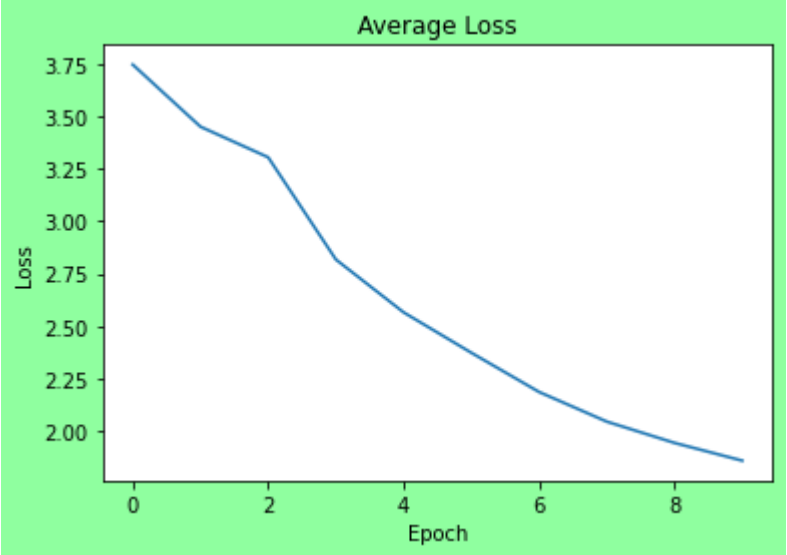
حال می‌خواهیم داده‌ها را نرمالایز کنیم. برای این کار چون می‌دانیم داده‌های ما عکس هستند پیکسل های آن‌ها را بر ۲۵۵ تقسیم می‌کنیم چرا که اندازه هر پیکسل بین ۰ تا ۲۵۵ است و به این صورت همه ی داده‌های ما بین ۰ تا ۱ می‌شود برای این کار داریم:

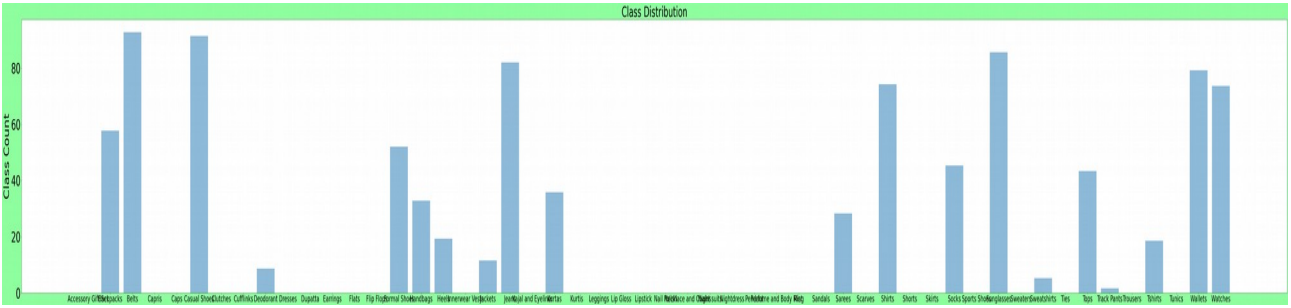
```
[3]  1
     2
     3 class CustomToTensor(object):
     4     """Convert PIL Images in sample to pytorch Tensors."
     5
     6     ## the input image must be grayscaled first
     7     def __call__(self, image):
     8         image = np.array(image, dtype=np.float32) / 255
     9         # numpy image: H x W
    10         return torch.from_numpy(image)
```

حال اگر شبکه را با این داده‌ها آموزش دهیم و همان کار های قبل را انجام دهیم داریم:

```
201]  1 average_loss_history = fit(model, train_loader, device, criterion, optimizer)

  epoch 1, train_loss: 3.744516, time elapsed: 42.912855 seconds
  epoch 2, train_loss: 3.449149, time elapsed: 42.777253 seconds
  epoch 3, train_loss: 3.303575, time elapsed: 43.475066 seconds
  epoch 4, train_loss: 2.817201, time elapsed: 42.577086 seconds
  epoch 5, train_loss: 2.565879, time elapsed: 43.360702 seconds
  epoch 6, train_loss: 2.373657, time elapsed: 42.295955 seconds
  epoch 7, train_loss: 2.187150, time elapsed: 43.361058 seconds
  epoch 8, train_loss: 2.046100, time elapsed: 43.074358 seconds
  epoch 9, train_loss: 1.944850, time elapsed: 43.02218 seconds
  epoch 10, train_loss: 1.860456, time elapsed: 43.302435 seconds
  total training time: 7.169 minutes
```

```
  total training time: 7.169 minutes

202]  1 print("Test Accuracy is: ")
      2 test_model_accuracy(model, test_loader)
      3 print("Train Accuracy is: ")
      4 test_model_accuracy(model, train_loader)

  Test Accuracy is:
  Accuracy: 37.40939597315436%
  Train Accuracy is:
  Accuracy: 38.393905835766304%
```





در اینجا ما با اسکیل کردن داده آن‌ها را بین ۰ تا ۱ آورده‌ایم به همین دلیل وزن های ما با مقادیر کمتری به سمت کاهش loss حرکت می‌کنند و دیرتر به مقدار اپتیموم خود می‌رسند در این صورت با اینکه دقت ما در ۱۰ ایپاک کمتر از حالت قبل است ولی می‌دانیم شبکه ما همچنان همگرا نشده است و برای اینکه به همگرایی خود برست یا باید تعداد ایپتک و یا نرخ یادگیری را افزایش دهیم

ولی با این حال در این قسمت دقت ما به دلیل اینکه وزن ها با مقادیر کمتری نسبت به قبل تغییر می‌کنند شبکه دیرتر همگرا می‌شود و برای رسیدن به دقت نهایی خود باید ایپاک یا نرخ یادگیری افزایش بیابد

۵)

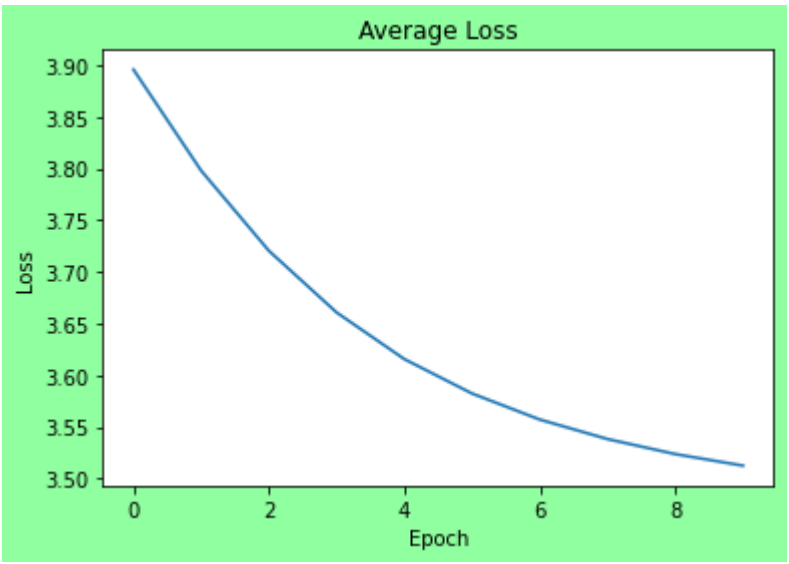
در این قسمت ابتدا تمام وزن های شبکه را ۰ می‌دهیم و با این مقادیر اولیه سعی می‌کنیم همان شبکه قبل را آموزش دهیم و داریم:

```
def func(x):
  if type(x) == torch.nn.modules.linear.Linear:
    x.weight.data.fill_(0)
    x.bias.data.fill_(0)
  model = model.apply(func)
```

با این کار تمام وزن ها و بایاس هارا ۰ می‌کنیم و برای یادگیری شبکه داریم:

```
[260]  1 average_loss_history = fit(model, train_loader, device, criterion, optimizer)

epoch 1, train_loss: 3.895573, time elapsed: 42.760149 seconds
epoch 2, train_loss: 3.797817, time elapsed: 42.352886 seconds
epoch 3, train_loss: 3.720346, time elapsed: 42.635493 seconds
epoch 4, train_loss: 3.660704, time elapsed: 42.107983 seconds
epoch 5, train_loss: 3.615625, time elapsed: 42.596955 seconds
epoch 6, train_loss: 3.582351, time elapsed: 42.733476 seconds
epoch 7, train_loss: 3.557170, time elapsed: 42.605618 seconds
epoch 8, train_loss: 3.538328, time elapsed: 42.572748 seconds
epoch 9, train_loss: 3.523792, time elapsed: 42.068154 seconds
epoch 10, train_loss: 3.512629, time elapsed: 42.492232 seconds
total training time: 7.082 minutes
```

```
[261]  1 print("Test Accuracy is: ")
       2 test_model_accuracy(model, test_loader)
       3 print("Train Accuracy is: ")
       4 test_model_accuracy(model, train_loader)

Test Accuracy is:
Accuracy: 9.48993288590604%
Train Accuracy is:
Accuracy: 9.48689553340716%
```



همان‌طور که میبینیم با دادن وزن اولیه ۰ به شبکه شبکه نتوانسته است آموزش ببیند چرا که وقتی وزن ها ۰ هستند در زمان اپدیت شدن نیز ۰ هستند و در هر تغییر خطایی با هر نرخ یادگیری ضرب شوند باز هم ۰ می‌شوند پس شبکه هیچ وقت نتوانسته تغییر در وزن ها ایجاد کند هر چند در مقدار بایاس لایه آخرتوانسته کمی تغییر ایجاد کند
به همین دلیل دقت شبکه به شدت کم شده است

ب)
یک راه حل خوب برای داشتن وزن اولیه مناسب این است که وزن ها را طبق یک تابع توزیع به صورت رندوم بدست بیاوریم این تابع توزیع به اکتیویشن های شبکه ما نیز بستگی دارد مثلاً یکی از تابع های توزیع خوب برای تابع Relu و یا Leaky Relu می‌تواند رادیکال ۲ تقسیم بر تعداد نورون ها در لایه ی قبل باشد این مقادیر به صورت اثبات شده‌ای هستند و می‌توانند گزینه خوبی باشند. در هر صورت یک مقادیر خاص برای همه شبکه‌ها نمی‌تواند درست باشد و هر چه مقادیر داده شده بهتر باشد می‌تواند شبکه را سریع‌تر و حتی با دقت بیشتری همگرا کند ولی چیزی که در این پروژه دیدیم نمی‌توانیم همه ی وزن ها را مقادیر کوچک و همه را شبیه به هم بدهیم چرا که شبکه نمی‌تواند به خوبی آموزش ببیند

6)
در این قسمت می‌خواهیم با تغییر نرخ یادگیری سعی کنیم به شبکه بهتری دست یابیم
برای این کار می‌خواهیم نرخ یادگیری‌های مختلفی را امتحان کنیم
ابتدا نرخ یادگیری را ۰.۰۰۵ می گذاریم:

```
learning_rate = 0.005
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)
```

```
Test Accuracy is:
Accuracy: 27.328859060402685%
Train Accuracy is:
Accuracy: 27.947246551897713%
```

همان‌طور که می‌بینیم با کم شدن نرخ یادگیری دقت ما کاهش یافته است
که این درست است چرا که با کم کردن این نرخ درواقع شبکه گام های اهسته تری را در جهت کم شدن لاس بر می‌دارد و شبکه ما در حالت قبل نیز به همگرایی نرسیده بود و با اهسته تر کردن آن نیز نمی‌تواند برسد پس دقت کم شده است

حال اگر نرخ یادگیری را زیاد کنیم در اینجا ۰.۰۵ گذاشته‌ایم خواهیم داشت:

```
learning_rate = 0.05
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)
```

```
average_loss_history = fit(model, train_loader, device, criterion, optimizer)
```

```
epoch 1, train_loss: 3.364327, time elapsed: 42.999617 seconds
epoch 2, train_loss: 2.685828, time elapsed: 43.526297 seconds
epoch 3, train_loss: 2.292553, time elapsed: 42.614961 seconds
epoch 4, train_loss: 1.979082, time elapsed: 43.753672 seconds
epoch 5, train_loss: 1.763481, time elapsed: 43.336255 seconds
epoch 6, train_loss: 1.626607, time elapsed: 43.004348 seconds
epoch 7, train_loss: 1.503564, time elapsed: 42.83412 seconds
epoch 8, train_loss: 1.450618, time elapsed: 42.979144 seconds
epoch 9, train_loss: 1.440681, time elapsed: 42.692541 seconds
epoch 10, train_loss: 1.326012, time elapsed: 42.954814 seconds
total training time: 7.178 minutes
```

```
[292]
print("Test Accuracy is: ")
test_model_accuracy(model, test_loader)
print("Train Accuracy is: ")
test_model_accuracy(model, train_loader)
```

```
Test Accuracy is:
Accuracy: 53.369127516778526%
Train Accuracy is:
Accuracy: 54.29041242994732%
```

همان‌طور که میبینیم شبکه به همگرایی نزدیک‌تر شده است ولی همچنان همگرا نشده است ولی با این حال می‌دانیم با زیاد کردن نرخ یادگیری شبکه می‌تواند سریع‌تر همگرا شود

حال اگر نرخ یادگیری را خیلی کم یعنی ۰.۰۰۱ بگذاریم داریم:

```
learning_rate = 0.001
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)
```

```
[303] average_loss_history = fit(model, train_loader, device, criterion, optimizer)
```

```
epoch 1, train_loss: 3.945507, time elapsed: 42.524621 seconds
epoch 2, train_loss: 3.928861, time elapsed: 42.291793 seconds
epoch 3, train_loss: 3.912642, time elapsed: 43.378361 seconds
epoch 4, train_loss: 3.895559, time elapsed: 42.576236 seconds
epoch 5, train_loss: 3.875481, time elapsed: 42.572247 seconds
epoch 6, train_loss: 3.847156, time elapsed: 42.89743 seconds
epoch 7, train_loss: 3.788294, time elapsed: 42.949562 seconds
epoch 8, train_loss: 3.620657, time elapsed: 42.114499 seconds
epoch 9, train_loss: 3.496782, time elapsed: 42.422221 seconds
epoch 10, train_loss: 3.472352, time elapsed: 43.39849 seconds
total training time: 7.119 minutes
```

```
[304] print("Test Accuracy is: ")
      test_model_accuracy(model, test_loader)
      print("Train Accuracy is: ")
      test_model_accuracy(model, train_loader)
```

```
Test Accuracy is:
Accuracy: 9.48993288590604%
Train Accuracy is:
Accuracy: 9.48689553340716%
```

می‌بینیم که شبکه نتوانسته آموزش ببیند و دقت به شدت پایین است زیرا با کم کردن نرخ یادگیری نیز اروم تر می‌شود و برای شبکه ی ما که در حالت با نرخ بیشتر نیز همگرا نشده بود دیر تر همگرا شده و در ۱۰ ایپاک به شدت عقب است

حال نرخ را ۰.۱ می‌گذاریم و داریم:

```
[311] learning_rate = 0.1
      criterion = nn.CrossEntropyLoss()
      optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)
```

```
[315]  1 average_loss_history = fit(model, train_loader, device, criterion, optimizer)
```

```
epoch 1, train_loss: 3.263031, time elapsed: 42.293316 seconds
epoch 2, train_loss: 2.724873, time elapsed: 42.493456 seconds
epoch 3, train_loss: 2.451261, time elapsed: 42.173401 seconds
epoch 4, train_loss: 2.105071, time elapsed: 42.911496 seconds
epoch 5, train_loss: 1.901026, time elapsed: 42.720884 seconds
epoch 6, train_loss: 1.733606, time elapsed: 42.702745 seconds
epoch 7, train_loss: 1.621092, time elapsed: 42.767877 seconds
epoch 8, train_loss: 1.493706, time elapsed: 42.406389 seconds
epoch 9, train_loss: 1.393518, time elapsed: 43.041303 seconds
epoch 10, train_loss: 1.344555, time elapsed: 42.484735 seconds
total training time: 7.100 minutes
```

```
[316]  1 print("Test Accuracy is: ")
       2 test_model_accuracy(model, test_loader)
       3 print("Train Accuracy is: ")
       4 test_model_accuracy(model, train_loader)
```

```
Test Accuracy is:
Accuracy: 57.40939597315436%
Train Accuracy is:
Accuracy: 58.22007449914427%
```

اگر نرخ را بالاتر بگذاریم مثلاً ۰.۳ خواهیم داشت:

```
[323]  1 learning_rate = 0.3
       2 criterion = nn.CrossEntropyLoss()
       3 optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)
```

```
[327]  1 average_loss_history = fit(model, train_loader, device, criterion, optimizer)
```

```
epoch 1, train_loss: 3.478570, time elapsed: 42.319633 seconds
epoch 2, train_loss: 3.371868, time elapsed: 42.520634 seconds
epoch 3, train_loss: 3.089815, time elapsed: 42.594879 seconds
epoch 4, train_loss: 2.960725, time elapsed: 43.004877 seconds
epoch 5, train_loss: 2.911348, time elapsed: 42.435862 seconds
epoch 6, train_loss: 2.915105, time elapsed: 42.559973 seconds
epoch 7, train_loss: 2.904950, time elapsed: 42.433087 seconds
epoch 8, train_loss: 3.211859, time elapsed: 43.473628 seconds
epoch 9, train_loss: 3.459007, time elapsed: 43.297738 seconds
epoch 10, train_loss: 3.447751, time elapsed: 42.967251 seconds
total training time: 7.127 minutes
```

```
[328]  1 print("Test Accuracy is: ")
       2 test_model_accuracy(model, test_loader)
       3 print("Train Accuracy is: ")
       4 test_model_accuracy(model, train_loader)
```

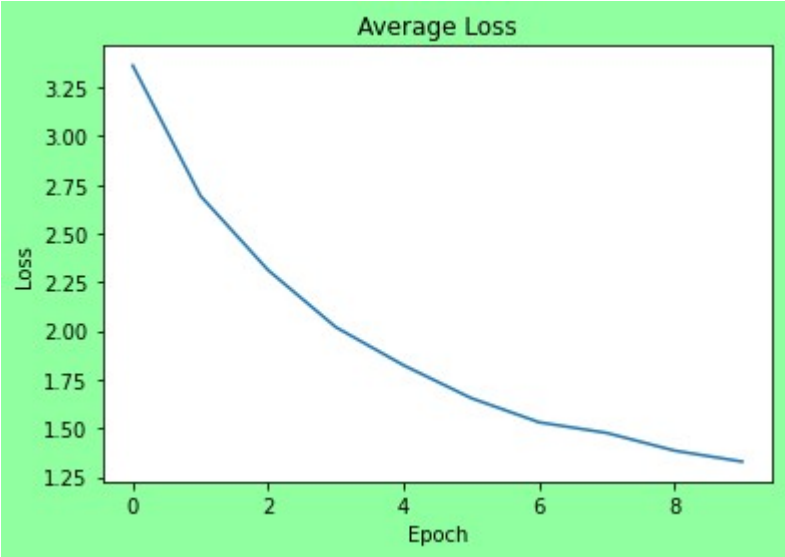
```
Test Accuracy is:
Accuracy: 12.295302013422818%
Train Accuracy is:
Accuracy: 12.33934024631699%
```

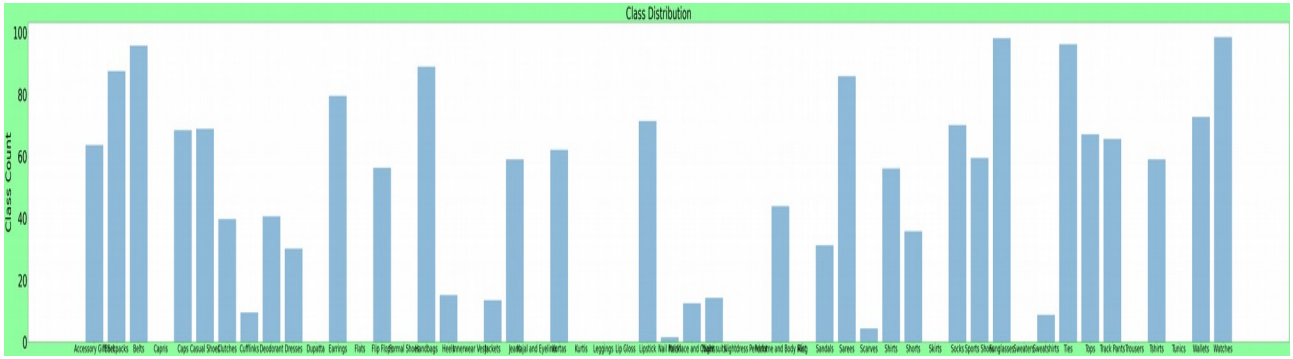
می‌بینیم شبکه نتوانسته همگرا شود و در یک local minimum گیر کرده چرا که با نرخ بالا حرکات به سرعت است و ممکن است از جالت بهینه رد شود

پس نرخ بهینه همان‌طور که دیدم ۰.۱ بوده است که داریم:

```
[339]  1 average_loss_history = fit(model, train_loader, device, criterion, optimizer)

epoch 1, train_loss: 3.362055, time elapsed: 44.037296 seconds
epoch 2, train_loss: 2.692929, time elapsed: 45.761846 seconds
epoch 3, train_loss: 2.311353, time elapsed: 43.764721 seconds
epoch 4, train_loss: 2.018428, time elapsed: 43.835134 seconds
epoch 5, train_loss: 1.823007, time elapsed: 45.293128 seconds
epoch 6, train_loss: 1.654063, time elapsed: 45.635384 seconds
epoch 7, train_loss: 1.530323, time elapsed: 46.600166 seconds
epoch 8, train_loss: 1.475429, time elapsed: 44.236256 seconds
epoch 9, train_loss: 1.384596, time elapsed: 45.022177 seconds
epoch 10, train_loss: 1.328091, time elapsed: 44.314234 seconds
total training time: 7.475 minutes
```

```
[340]  1 print("Test Accuracy is: ")
       2 test_model_accuracy(model, test_loader)
       3 print("Train Accuracy is: ")
       4 test_model_accuracy(model, train_loader)

Test Accuracy is:
Accuracy: 56.48322147651007%
Train Accuracy is:
Accuracy: 57.02204771972214%
```





7)

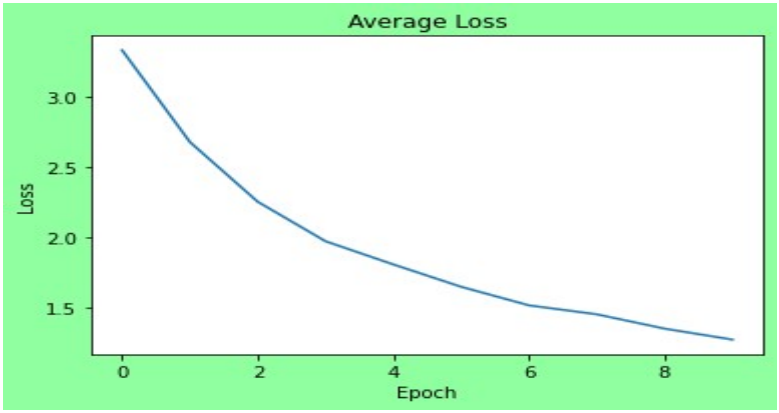
در این قسمت می‌خواهیم تغییر batch size را مقایسه کنیم

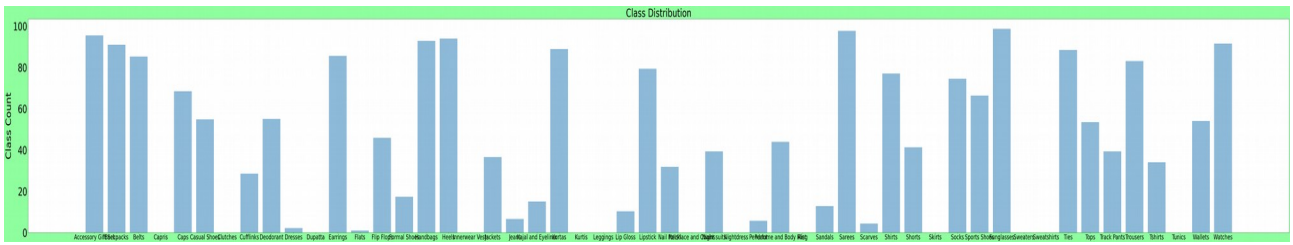
ابتدا batch size را عدد ۳۲ می گذاریم:

```
batch_size = 32
validation_split = 0.2


indices = list(range(len(dataset))) # indices of the dataset
print(len(indices))

# TODO: split the dataset into train and test sets randomly with split of 0.2 and assign their indices in the original set t
targets = dataset.targets
train_indices, test_indices = train_test_split(indices, test_size=0.2, train_size=0.8 , stratify=targets, random_state=42)
print(len(train_indices), len(test_indices))

# Creating PT data samplers and loaders:
train_sampler = SubsetRandomSampler(train_indices)
test_sampler = SubsetRandomSampler(test_indices)

train_loader = torch.utils.data.DataLoader(dataset, batch_size=batch_size, sampler=train_sampler, num_workers=16)
test_loader = torch.utils.data.DataLoader(dataset, batch_size=batch_size, sampler=test_sampler, num_workers=16)
```

و داریم:

```
[356]  1 average_loss_history = fit(model, train_loader, device, criterion, optimizer)

epoch 1, train_loss: 3.330490, time elapsed: 45.75499 seconds
epoch 2, train_loss: 2.678015, time elapsed: 44.409191 seconds
epoch 3, train_loss: 2.253968, time elapsed: 42.462521 seconds
epoch 4, train_loss: 1.973366, time elapsed: 42.427768 seconds
epoch 5, train_loss: 1.808613, time elapsed: 43.163455 seconds
epoch 6, train_loss: 1.649778, time elapsed: 43.234024 seconds
epoch 7, train_loss: 1.518011, time elapsed: 42.797315 seconds
epoch 8, train_loss: 1.454421, time elapsed: 43.181895 seconds
epoch 9, train_loss: 1.352641, time elapsed: 42.925584 seconds
epoch 10, train_loss: 1.274855, time elapsed: 42.970295 seconds
total training time: 7.222 minutes

[357]  1 print("Test Accuracy is: ")
       2 test_model_accuracy(model, test_loader)
       3 print("Train Accuracy is: ")
       4 test_model_accuracy(model, train_loader)

Test Accuracy is:
Accuracy: 58.053691275167786%
Train Accuracy is:
Accuracy: 58.95499848988221%
```



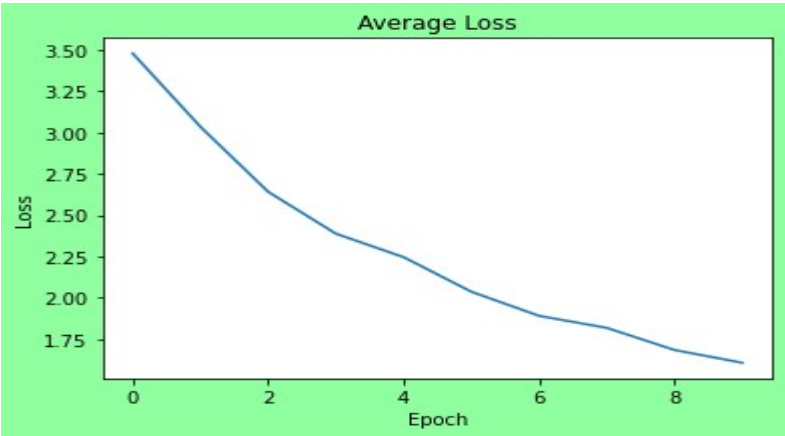


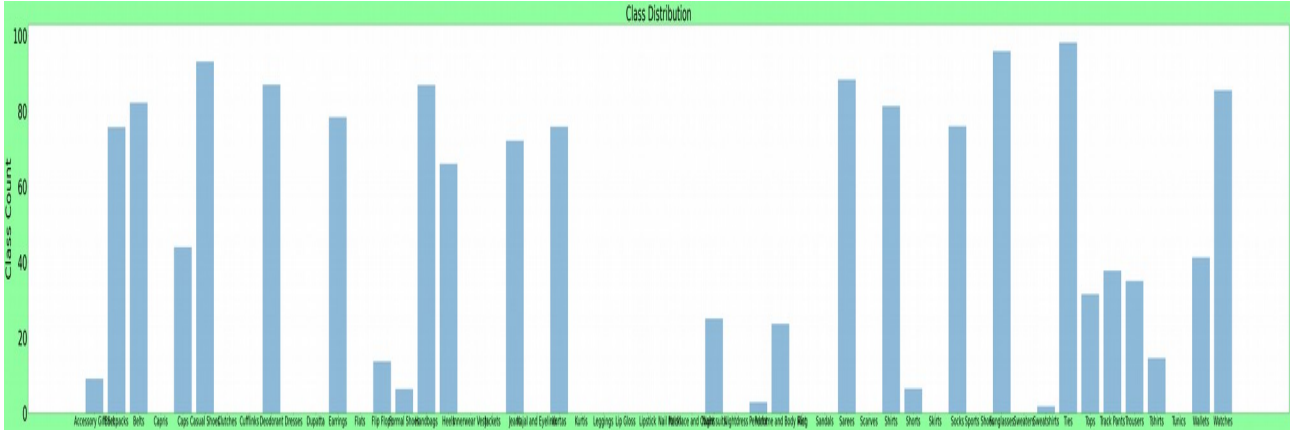
حال اگر batch size را ۱۲۸ بگذاریم داریم:

```
batch_size = 128
validation_split = 0.2


indices = list(range(len(dataset))) # indices of the dataset
print(len(indices))

# TODO: split the dataset into train and test sets randomly with split of 0.2 and assign their indices in the original set to t
targets = dataset.targets
train_indices, test_indices = train_test_split(indices, test_size=0.2, train_size=0.8 , stratify=targets, random_state=42)
print(len(train_indices), len(test_indices))

# Creating PT data samplers and loaders:
train_sampler = SubsetRandomSampler(train_indices)
test_sampler = SubsetRandomSampler(test_indices)

train_loader = torch.utils.data.DataLoader(dataset, batch_size=batch_size, sampler=train_sampler, num_workers=16)
test_loader = torch.utils.data.DataLoader(dataset, batch_size=batch_size, sampler=test_sampler, num_workers=16)
```

```
[ ]  average_loss_history = fit(model, train_loader, device, criterion, optimizer)

epoch 1, train_loss: 3.478358, time elapsed: 41.334984 seconds
epoch 2, train_loss: 3.034521, time elapsed: 42.049667 seconds
epoch 3, train_loss: 2.641148, time elapsed: 41.070876 seconds
epoch 4, train_loss: 2.387853, time elapsed: 41.04325 seconds
epoch 5, train_loss: 2.245686, time elapsed: 41.053591 seconds
epoch 6, train_loss: 2.036804, time elapsed: 40.549509 seconds
epoch 7, train_loss: 1.890778, time elapsed: 42.126107 seconds
epoch 8, train_loss: 1.817938, time elapsed: 41.217677 seconds
epoch 9, train_loss: 1.684098, time elapsed: 41.350781 seconds
epoch 10, train_loss: 1.606984, time elapsed: 41.552035 seconds
total training time: 6.889 minutes
```

```
[ ]  print("Test Accuracy is: ")
     test_model_accuracy(model, test_loader)
     print("Train Accuracy is: ")
     test_model_accuracy(model, train_loader)

Test Accuracy is:
Accuracy: 48.25503355704698%
Train Accuracy is:
Accuracy: 48.41437632135307%
```





همان‌طور که میبینیم وقتی داده‌ها را در batch های ۱۲۸ می‌گذاریم شبکه کمی دقت اش پایین می‌آید چرا که دیگر داده‌ها را به صورت جدا جدا بررسی نمی‌کند و به جای هر ۶۴ ۶۴ تا ۱۲۸ ۱۲۸ تا این کار را انجام می‌دهد ولی همین‌طور که مشخص است میزان زمان یادگیری شبکه کاهش پیدا کرده است و شبکه تنوانسته در زمان کمتری آموزش ببیند که این از مزیت های batch size بالاتر است در این شبکه چون فقط ۱۰ ایپاک آموزش انجام داده‌ایم این مقدار زیاد به چشم نیامده و در حد ۱ دقیقه بوده است ولی هر چقدر زمان اجرا بیشتر باشد این اختلاف واضح تر نیز می‌شود

حال وقتی batch ها را ۳۲ تا در نظر می‌گیریم می‌بینیم دقت شبکه کمی بیشتر شده است و زمان بیشتری برای آموزش شبکه طول می‌کشد ولی همان‌طور که میبینیم این افزایش دقت به‌قدری نبوده که بخواهیم شبکه زمان بیشتری برای آموزش بگیرد و همان‌طور که گفته شد در این شبکه چون زمان آموزش کم بودن است این تأخیر زیاد به چشم نیاکده ولی در شبکه‌های بزرگ که زمان بین چند روز تا چند هفته است این اختلاف زیاد می‌شود و شاید همیشه خوب نباشد که ۱ ۲ درصد افزایش دقت برای اضافه شدن چند روز به آموزش مؤثر باشد.

8)

در این قسمت می‌خواهیم تأثیر momentum بر روی آموزش شبکه را ببینیم:

الف) momentum یک پارامتری است که ما آن را در در اپدیت کردن وزن ها اضافه می‌کنیم این پارامتر به شبکه کمک می‌کند تا سریع‌تر به همگرایی برسد و به شدت در سرعت یادگیری و همگرایی شبکه تأثیر دارد

این پارامتر برای الگوریتم SGD استفاده می‌شود چرا که این الگوریتم سرعت کمی دارد و اگر دیتا ست ما بزرگ باشد ممکن است زمان زیادی طول بکشد تا الگوریتم همگرا شود برای همین از momentum استفاده می‌کنیم فواید استفاده از آن می‌تواند gradient های بسیار کوچک که ممکن است باعث عدم اپدیت وزن ها شود را از بین ببرد و از طرفی می‌تواند gradient های نویز دار که در شبکه‌هایی با تعداد لایه بالا به وجود میارد را کمک کند

از معایب آن هم می‌تواند به اضافه کردن پیچیدگی به شبکه نام برد چرا که می‌دانیم هر چه شبکه ما پیچیده‌تر باشد احتمال overfit شدن آن بر روی داده‌های آموزش بیشتر می‌شود و ممکن است دقت شبکه را پایین‌تر نیز بیاورد

حال می‌خواهیم شبکه را با momentum های مختلف امتحان کنیم ابتدا momentum را ۰.۵ می گذاریم:

```
learning_rate = 0.1
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate, momentum=0.5)
```

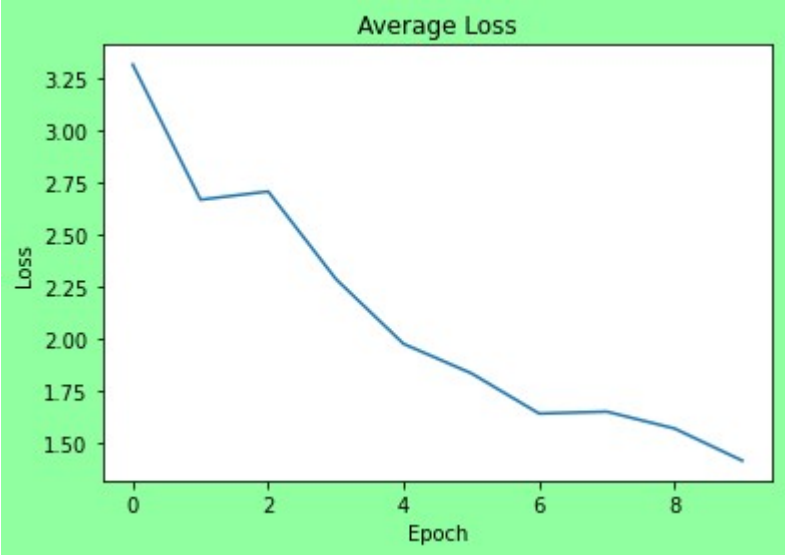
و بعد از اجرا داریم:

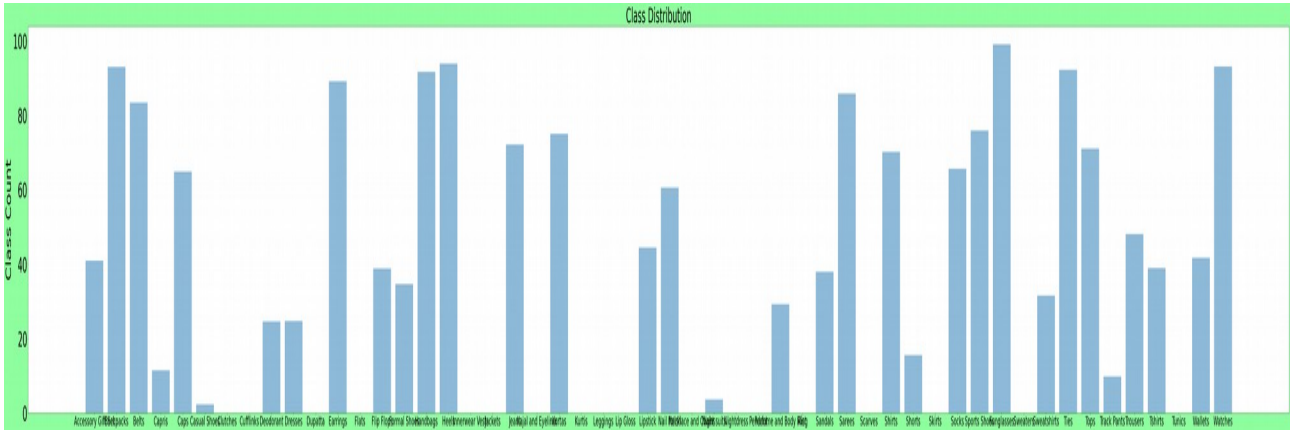
```
[16]  1 average_loss_history = fit(model, train_loader, device, criterion, optimizer)
```

```
epoch 1, train_loss: 3.317158, time elapsed: 1145.902087 seconds
epoch 2, train_loss: 2.667853, time elapsed: 30.396069 seconds
epoch 3, train_loss: 2.707525, time elapsed: 30.220551 seconds
epoch 4, train_loss: 2.286506, time elapsed: 29.809835 seconds
epoch 5, train_loss: 1.974663, time elapsed: 30.149432 seconds
epoch 6, train_loss: 1.833424, time elapsed: 30.177882 seconds
epoch 7, train_loss: 1.639948, time elapsed: 29.924229 seconds
epoch 8, train_loss: 1.649006, time elapsed: 30.07698 seconds
epoch 9, train_loss: 1.568008, time elapsed: 30.181947 seconds
epoch 10, train_loss: 1.413523, time elapsed: 30.430562 seconds
total training time: 23.621 minutes
```

```
[17]  1 print("Test Accuracy is: ")
      2 test_model_accuracy(model, test_loader)
      3 print("Train Accuracy is: ")
      4 test_model_accuracy(model, train_loader)
```

```
Test Accuracy is:
Accuracy: 54.013422818791945%
Train Accuracy is:
Accuracy: 54.33403805496829%
```





حال با momentum برابر ۰.۹ داریم:

```
learning_rate = 0.1
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate, momentum=0.9)
```

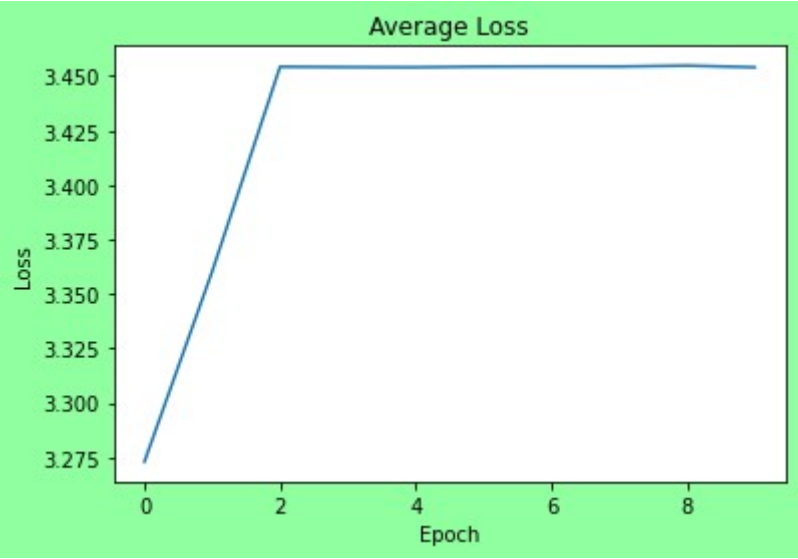
و بعد از اجرا داریم:

```
average_loss_history = fit(model, train_loader, device, criterion, optimizer)
```

```
epoch 1, train_loss: 3.273101, time elapsed: 30.912133 seconds
epoch 2, train_loss: 3.360442, time elapsed: 30.483941 seconds
epoch 3, train_loss: 3.454280, time elapsed: 30.988877 seconds
epoch 4, train_loss: 3.454123, time elapsed: 30.429968 seconds
epoch 5, train_loss: 3.454055, time elapsed: 30.548479 seconds
epoch 6, train_loss: 3.454331, time elapsed: 31.369157 seconds
epoch 7, train_loss: 3.454403, time elapsed: 30.787729 seconds
epoch 8, train_loss: 3.454367, time elapsed: 31.433699 seconds
epoch 9, train_loss: 3.454808, time elapsed: 30.61259 seconds
epoch 10, train_loss: 3.454035, time elapsed: 31.112784 seconds
total training time: 5.145 minutes
```

```
print("Test Accuracy is: ")
test_model_accuracy(model, test_loader)
print("Train Accuracy is: ")
test_model_accuracy(model, train_loader)
```

```
Test Accuracy is:
Accuracy: 9.48993288590604%
Train Accuracy is:
Accuracy: 9.48689553340716%
```



و اگر momentum را عدد ۰.۹۸ بگذاریم داریم:

```
learning_rate = 0.1
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate, momentum=0.98)
```

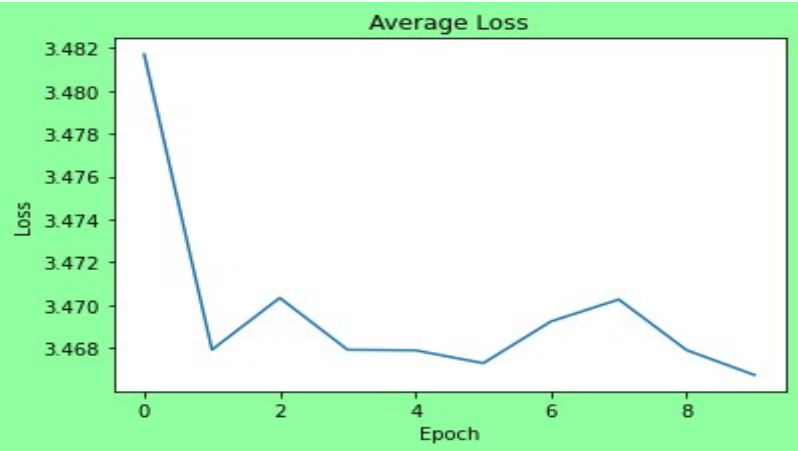
و بعد از اجرا داریم:

```
average_loss_history = fit(model, train_loader, device, criterion, optimizer)
```

```
epoch 1, train_loss: 3.481701, time elapsed: 30.903565 seconds
epoch 2, train_loss: 3.467926, time elapsed: 30.609867 seconds
epoch 3, train_loss: 3.470346, time elapsed: 30.458799 seconds
epoch 4, train_loss: 3.467924, time elapsed: 31.14013 seconds
epoch 5, train_loss: 3.467897, time elapsed: 30.41048 seconds
epoch 6, train_loss: 3.467296, time elapsed: 30.455529 seconds
epoch 7, train_loss: 3.469247, time elapsed: 31.759028 seconds
epoch 8, train_loss: 3.470271, time elapsed: 31.410324 seconds
epoch 9, train_loss: 3.467907, time elapsed: 30.514925 seconds
epoch 10, train_loss: 3.466749, time elapsed: 31.156351 seconds
total training time: 5.147 minutes
```

```
print("Test Accuracy is: ")
test_model_accuracy(model, test_loader)
print("Train Accuracy is: ")
test_model_accuracy(model, train_loader)
```

```
Test Accuracy is:
Accuracy: 9.48993288590604%
Train Accuracy is:
Accuracy: 9.48689553340716%
```



مومنتوم درواقع راهی برای بهینه کردن search و افزایش سرعت شبکه است momentum باعث می‌شود شبکه برای اپدیت کردن وزن ها علاوه بر نرخ یادگیری یک عبارت دیگر نیز به آن اضافه کند که این عبارت متأثر از ضریب momentum و شیب پیشرفت در مرحله قبل است درواقع همانند این است که به جای استفاده از نرخ یادگیری ثابت از نرخ داینامیک استفاده کنیم
momentum در کل باعث می‌شود سرعت تغییرات وزن ها بیشتر شده و در نتیجه شبکه با سرعت بیشتری همگرا شود ولی این در صورتی است که یک ضریب متناسب با نرخ یادگیری برای شبکه پیدا کنیم اگر ضریب از حدی بیشتر شود این سرعت پیشرفت زیاد می‌شود و در نتیجه وزن ها نمی‌توانند به مقادیر اپتیموم خود برسند و شبکه از همگرایی خارج شده مقدار لاس زیاد می‌شود و دقت کم می‌شود
در این پروژه همان‌طور که در مثال‌ها دیدیم وقتی ضریب را ۰.۵ گذاشته‌ایم شبکه توانست با سرعت بیشتری همگرا شود و درواقع دقت بالاتری گرفتیم که نشان می‌دهد این ضریب برای ما مناسب بوده است ولی در حالت ۰.۹ و ۰.۹۸ دیدیم که ضریب عدد بالایی برای مدل بوده است و مدل از همگرایی خارج شده چرا که وزن ها دچار پرش زیاد شده و از حالت اپتیموم خود خارج شده‌اند و شبکه نتوانسته به دقت کافی برای آن برسد
پس ضریب بهینه ما ۰.۵ بوده است.

۹)

در این قسمت می‌خواهیم تأثیر تعداد epoch را بر شبکه ببینیم
برای همین ابتدا شبکه را با ۲۰ epoch انجام می دهیم:

```
average_loss_history = fit(model, train_loader, device, criterion, optimizer, 20)
```

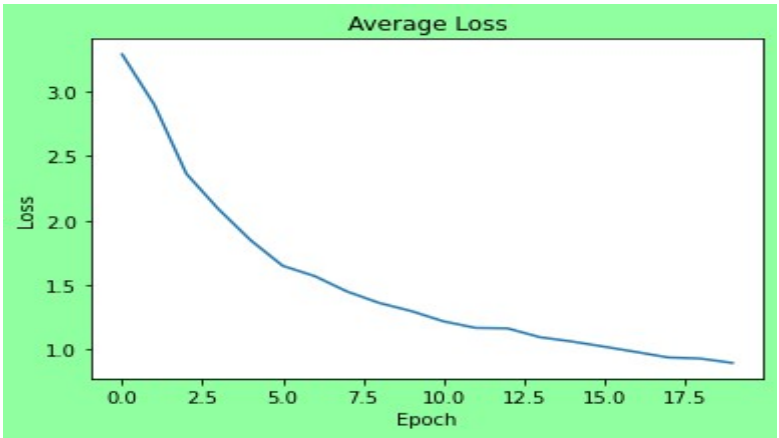
و بعد از اجرا داریم:

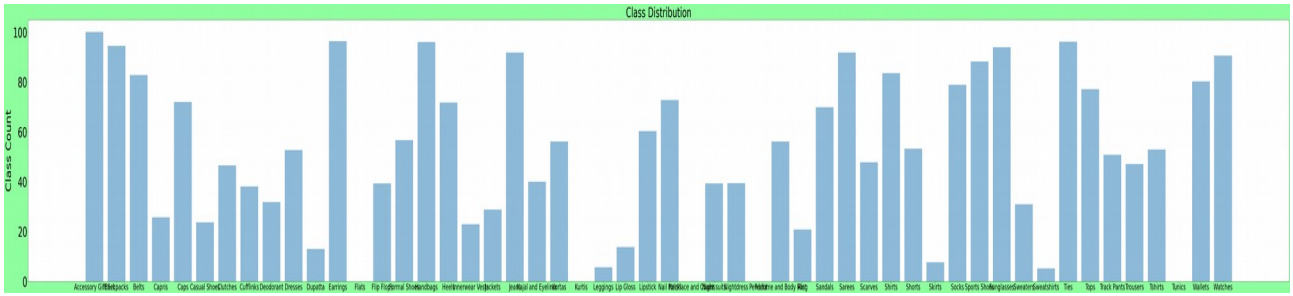
```
average_loss_history = fit(model, train_loader, device, criterion, optimizer, 20)
```

```
epoch 1, train_loss: 3.289797, time elapsed: 30.900932 seconds
epoch 2, train_loss: 2.899108, time elapsed: 30.73644 seconds
epoch 3, train_loss: 2.363157, time elapsed: 30.971068 seconds
epoch 4, train_loss: 2.088678, time elapsed: 31.302545 seconds
epoch 5, train_loss: 1.847477, time elapsed: 31.123652 seconds
epoch 6, train_loss: 1.648844, time elapsed: 31.500247 seconds
epoch 7, train_loss: 1.568405, time elapsed: 30.919157 seconds
epoch 8, train_loss: 1.449461, time elapsed: 31.291395 seconds
epoch 9, train_loss: 1.360990, time elapsed: 30.853701 seconds
epoch 10, train_loss: 1.297691, time elapsed: 31.119908 seconds
epoch 11, train_loss: 1.218250, time elapsed: 31.890037 seconds
epoch 12, train_loss: 1.167255, time elapsed: 31.825144 seconds
epoch 13, train_loss: 1.163278, time elapsed: 30.573179 seconds
epoch 14, train_loss: 1.095194, time elapsed: 30.765265 seconds
epoch 15, train_loss: 1.061556, time elapsed: 30.792183 seconds
epoch 16, train_loss: 1.021624, time elapsed: 31.261687 seconds
epoch 17, train_loss: 0.980273, time elapsed: 31.119836 seconds
epoch 18, train_loss: 0.937602, time elapsed: 31.137211 seconds
epoch 19, train_loss: 0.929221, time elapsed: 31.003903 seconds
epoch 20, train_loss: 0.895226, time elapsed: 30.991002 seconds
total training time: 10.368 minutes
```

```
print("Test Accuracy is: ")
test_model_accuracy(model, test_loader)
print("Train Accuracy is: ")
test_model_accuracy(model, train_loader)
```

```
Test Accuracy is:
Accuracy: 64.02684563758389%
Train Accuracy is:
Accuracy: 65.62971911809122%
```





همان‌طور که میدانیم شبکه با استفاده از الگوریتم gradient decent سعی می‌کند با بدست آوردن یک ویژگی‌های خاص برای هر کلاس آن‌ها را تشخیص دهد ولی شبکه و الکوریتم برای بدست آوردن این ویژگی‌ها نیاز به تعداد بالایی داده دارد و بدست آوردن این مقدار دیتا هم نیاز به label کردن آن‌ها هم ذخیره آن‌ها را به وجود می‌آورد که کار تقریباً غیر ممکن است ولی با داده کم‌تر هم می‌شود این ویژگی‌ها را استخراج کرد. الگوریتم gradient decent برای استخراج این ویژگی هاست برای اینکه بتواند آن‌ها را بدست آورد باید از مقدار زیادی داده استفاده کند که این کقدار ممکن نیست پس به جای آن همان یکسری داده را بار ها با این الگوریتم ران می‌کنیم تا بتواند مقادیر وزن ها را اپتیموم کند و در نهایت همگرا شود و می‌دانیم همگرایی این الگوریتم کاری زمان بر است.
می‌دانیم همیشه افزایش تعداد ایپاک باعث افزایش دقت نمی‌شود هر ایپاک درواقع زمان آموزش شبکه را زیاد تر می‌کند و می‌دانیم شبکه بعد از تعداد دور محدودی درواقع هم گرا می‌شود و با زیاد کردن ایپاک از آن به بعد ممکن است دچار overfit شود پس باید تعداد دور را تا جایی زیاد کنیم که شبکه همگرا شده یعنی مقدار اررور آن تقریباً ثابت شده باشد.

همان‌طور که در پروژه دیدیم شبکه با ۱۰ ایپاک نتوانسته بود همگرا شود چرا که همچنان لاس آن در حال کاهش بوده است ولی وقتی تعداد دور بخ ۲۰ رسید شبکه به حالت همگرایی خود نزدیک‌تر شد و در نتیجه دقت آن بالاتر رفت هر جند برای همگرایی کامل این شبکه باز هم به تعداد بیشتری ایپاک نیاز داشته ایم.

10)

در این قسمت می‌خواهیم توابع اکتیویشن مختلف را امتحان کرده و تأثیر آن‌ها را ببینیم

برای این کار ابتدا تابع tanh را به جای relu قرار می دهیم:

```
model = Model(len(classes), act=torch.tanh)

model = model.to(device)
```

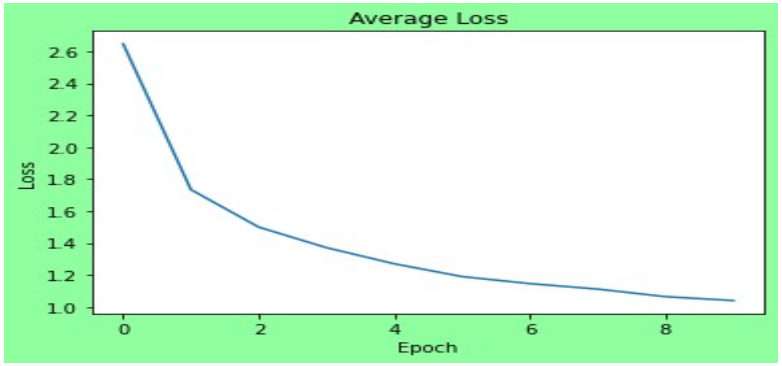
و بعد از اجرا داریم:

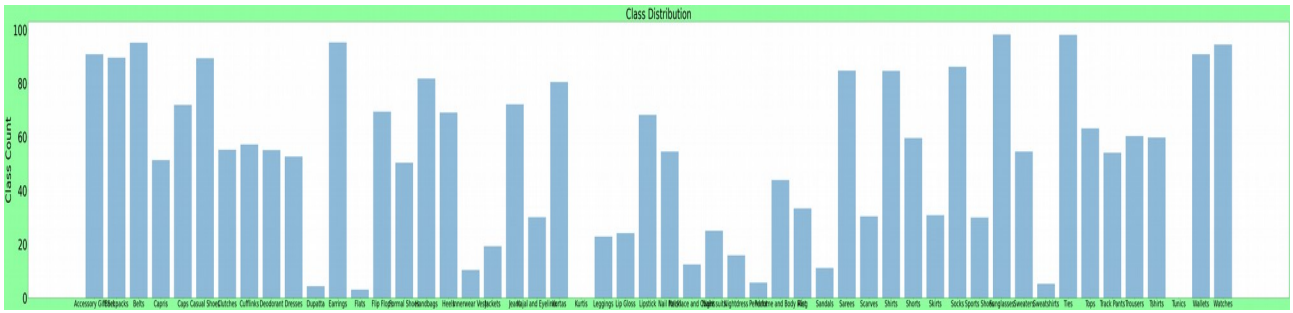
```
average_loss_history = fit(model, train_loader, device, criterion, optimizer)
```

```
epoch 1, train_loss: 2.646268, time elapsed: 2119.446899 seconds
epoch 2, train_loss: 1.734297, time elapsed: 46.686499 seconds
epoch 3, train_loss: 1.499635, time elapsed: 46.96271 seconds
epoch 4, train_loss: 1.370874, time elapsed: 46.404475 seconds
epoch 5, train_loss: 1.270107, time elapsed: 46.026838 seconds
epoch 6, train_loss: 1.189911, time elapsed: 46.584665 seconds
epoch 7, train_loss: 1.146149, time elapsed: 46.822605 seconds
epoch 8, train_loss: 1.111849, time elapsed: 46.055643 seconds
epoch 9, train_loss: 1.064455, time elapsed: 46.241655 seconds
epoch 10, train_loss: 1.040137, time elapsed: 47.051017 seconds
total training time: 42.305 minutes
```

```
print("Test Accuracy is: ")
test_model_accuracy(model, test_loader)
print("Train Accuracy is: ")
test_model_accuracy(model, train_loader)
```

```
Test Accuracy is:
Accuracy: 66.7248322147651%
Train Accuracy is:
Accuracy: 68.61639652337327%
```





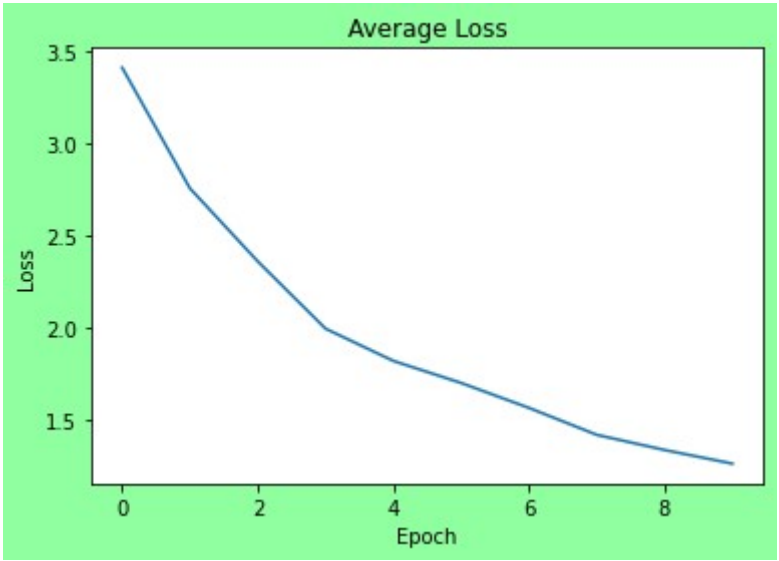
حال اگر تابع را Leaky Relu بگذاریم:

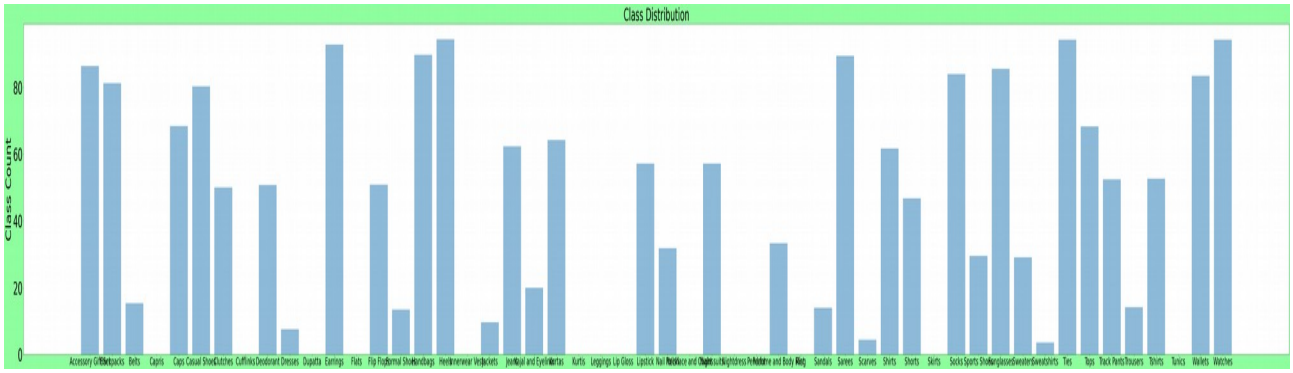
```
1 average_loss_history = fit(model, train_loader, device, criterion, optimizer)
```

```
epoch 1, train_loss: 3.409290, time elapsed: 49.096277 seconds
epoch 2, train_loss: 2.756123, time elapsed: 51.039385 seconds
epoch 3, train_loss: 2.361075, time elapsed: 53.560408 seconds
epoch 4, train_loss: 1.997043, time elapsed: 56.052658 seconds
epoch 5, train_loss: 1.822586, time elapsed: 47.105589 seconds
epoch 6, train_loss: 1.703257, time elapsed: 46.953776 seconds
epoch 7, train_loss: 1.568385, time elapsed: 46.430649 seconds
epoch 8, train_loss: 1.422668, time elapsed: 46.146531 seconds
epoch 9, train_loss: 1.340712, time elapsed: 46.796951 seconds
epoch 10, train_loss: 1.266896, time elapsed: 46.605801 seconds
total training time: 8.163 minutes
```

```
1 print("Test Accuracy is: ")
2 test_model_accuracy(model, test_loader)
3 print("Train Accuracy is: ")
4 test_model_accuracy(model, train_loader)
```

```
Test Accuracy is:
Accuracy: 57.006711409395976%
Train Accuracy is:
Accuracy: 57.713346085439106%
```





همان‌طور که می‌بینیم در اینجا با تابع tanh جواب بهتر از relu و با leaky_relu جوابی کمی بد تر از relu گرفته‌ایم
در کل می‌دانیم در اکثر مواقع مخصوصاً وقتی تعداد لایه ها زیاد است تابع reluمی تواند به خوبی عمل کند چون در الگوریتم gradient decent می‌دانیم در هر مرحله به مرحله قبلی که می ریم از مشتق خروجی مرحله بعد داریم استفاده می‌کنیم و قتی تابع ما relu است مشتق آن نیز ساده است که در منفی ۰ و در مثبت که یک خط با شیب یک است یک مقدار ثابت می‌دهد این باعث می‌شود مدل ساده باشد و این مقدار به مراحل قبل برسد ولی وقتی tanh استفاده می‌کنیم چون مشتق آن باعث بوجود آمدن یک tanh دیگر می‌شود به پیچیدگی مدل می افزاید و وقتی تعداد لایه ها زیاد است ممکن است این مقادیر به لایه‌های اولیه نرسد و درواقع آن‌ها دیگر اپدیت نمی‌شوند و مشکل vanishin gradient به وجود اید که در حالت relu وجود ندارد ولی خوبی آن این است که می‌تواند مقادیر منفی را نیز به خوبی مدل کند در حالی که relu نمی‌تواند
تابع leaky relu مقادیر منفی را تا حدی عبور می‌دهد ولی وقتی تعداد لایه افزایش می‌یابد آن هم با مشکل vanishing gradient رو به رو می‌شود در اینجا چون به relu فرق زیادی نمی‌کند درواقع ما کمی پیچیدگی فقط اضافه کرده‌ایم و سرعت مدل را کم کرده‌ایم که باعث کم شدن دقت نیز شده است.
در کل ما باید با توجه به مدلمان اینکه عدد منفی می‌خواهیم یا نه و تعداد لایه ها و هایپر پارامتر های دیگر تابع را انتخاب کنیم ولی در حالت کلی تابع relu می‌تواند خوب عمل کند و سپس می‌توانیم توابع دیگر را نیز امتحان کنیم.

11)

الف) یک مدل آموزش به خودی خود یک واریانسی دارد که ممکن است همان‌گونه که برای داده‌های آموزش عمل می‌کند نتواند برای هر داده تستی عمل کند Regularization می‌تواند واریانس یک مدل را کم کند و در عین حال بایاس مدل را تغییر زیاد ای ندهد و پارامتر Regularization می‌تواند این تأثیر را کنترل کند به این صورت که هر چه بیشتر باشد اندازه وزن ها را می‌تواند کم کند پس واریانس مدل کم می‌شود ولی از یک جایی به بعد با زیاد شدن این پارامتر بایاس مدل را بسیار افزایش داده و باعث underfitt می‌شود این کاهش واریانس می‌تواند از overfit شدن شبکه جلوگیری کند

در شبکه‌های عصبی این regularization در cost function اضافه می‌شود که دو مدل معروف آن L1 و L2 است که درواقع یک عبارتی است که با Loss Function ای که داشتیم مثلاً Cross Entropy جمع می‌شود و باعث کاهش مقدار وزن ها به نزدیک صفر و در نتیجه کاهش واریانس به خاطر ساده‌تر شدن مدل و در نتیجه کم شدن overfit می شود.

ب) با اضافه کردن weight decay به تابع optimizer درواقع می‌خواهیم با تغییر لاس از روش regularization که توضیح داده شد استفاده کنیم این پارامتر می‌تواند یک پارامتر به تابع لاس اضافه کند مانند زیر:

```
Loss = MSE(y_hat, y) + wd * sum(w^2)
```

When we update weights using gradient descent we do the following:

```
w(t) = w(t-1) - lr * dLoss / dw
```

Now since our loss function has 2 terms in it, the derivative of the 2nd term w.r.t `w` would be:

```
d(wd * w^2) / dw = 2 * wd * w (similar to d(x^2)/dx = 2x)
```

در این صورت ما هر دفعه برای اپدیت وزن ها دیگر نرخ یادگیری را در خود وزن ضرب نمی‌کنیم و در ۲ برابر wd در وزن ضرب می‌کنیم که یعنی در اپدیت سعی می‌شود مقدار وزن ها هر دفع کمتر شود و این کاهش وزن و واریانس از overfit جلوگیری می کند

ج) حال شبکه را با Weight Decay برابر ۰.۱ آموزش می دهیم:

```
learning_rate = 0.1
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate, momentum=0.5, weight_decay=0.1)
```

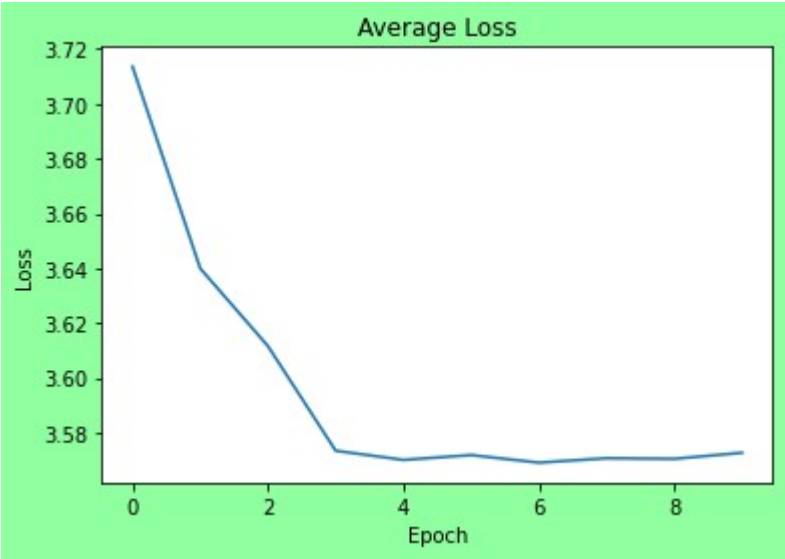
و داریم:

```
average_loss_history = fit(model, train_loader, device, criterion, optimizer)
```

```
epoch 1, train_loss: 3.713599, time elapsed: 1000.470115 seconds
epoch 2, train_loss: 3.639896, time elapsed: 48.245489 seconds
epoch 3, train_loss: 3.611539, time elapsed: 46.573375 seconds
epoch 4, train_loss: 3.573315, time elapsed: 47.070218 seconds
epoch 5, train_loss: 3.569977, time elapsed: 46.820359 seconds
epoch 6, train_loss: 3.571742, time elapsed: 49.135913 seconds
epoch 7, train_loss: 3.568972, time elapsed: 48.043333 seconds
epoch 8, train_loss: 3.570583, time elapsed: 47.51299 seconds
epoch 9, train_loss: 3.570363, time elapsed: 47.454761 seconds
epoch 10, train_loss: 3.572583, time elapsed: 49.104224 seconds
total training time: 23.841 minutes
```

```
print("Test Accuracy is: ")
test_model_accuracy(model, test_loader)
print("Train Accuracy is: ")
test_model_accuracy(model, train_loader)
```

```
Test Accuracy is:
Accuracy: 9.48993288590604%
Train Accuracy is:
Accuracy: 9.48689553340716%
```



حال با Weight Decay برابر ۰.۰۱ امتحان می کنیم:

```
learning_rate = 0.1
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate, momentum=0.5, weight_decay=0.01)
```

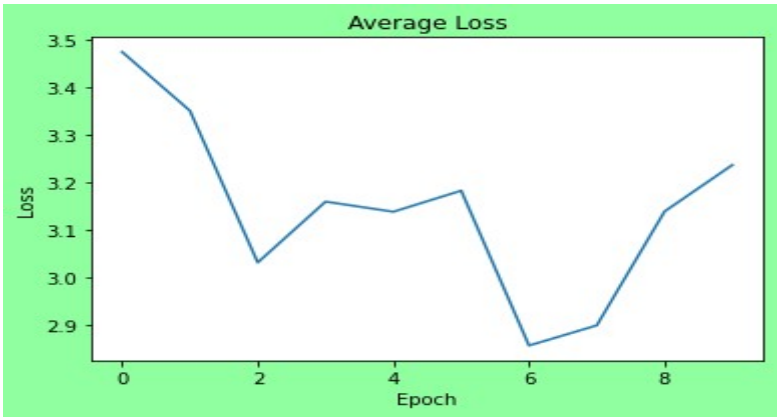
و داریم:

```
average_loss_history = fit(model, train_loader, device, criterion, optimizer)
```

```
epoch 1, train_loss: 3.475443, time elapsed: 44.219985 seconds
epoch 2, train_loss: 3.351666, time elapsed: 44.415506 seconds
epoch 3, train_loss: 3.032493, time elapsed: 44.208501 seconds
epoch 4, train_loss: 3.160555, time elapsed: 44.18392 seconds
epoch 5, train_loss: 3.139263, time elapsed: 44.019649 seconds
epoch 6, train_loss: 3.183616, time elapsed: 43.940685 seconds
epoch 7, train_loss: 2.857849, time elapsed: 43.747722 seconds
epoch 8, train_loss: 2.900295, time elapsed: 44.278771 seconds
epoch 9, train_loss: 3.139984, time elapsed: 43.352659 seconds
epoch 10, train_loss: 3.237772, time elapsed: 43.676177 seconds
total training time: 7.334 minutes
```

```
print("Test Accuracy is: ")
test_model_accuracy(model, test_loader)
print("Train Accuracy is: ")
test_model_accuracy(model, train_loader)
```

```
Test Accuracy is:
Accuracy: 9.48993288590604%
Train Accuracy is:
Accuracy: 9.48689553340716%
```



این‌طور که می‌بینیم شبکه در هیچ یک از دو حالت با استفاده از weight decay نتوانسته به درستی همگرا شود
چرا که این کاهش وزن باعث شده که وزن زیادی کم شود و شبکه از همگرایی خود خارج شود
دقت آن از حالت بهینه momentm خالی کم‌تر است
ولی در حالت کلی weight decay باعث می‌شود که شبکه ما در حالت تست بهتر عمل کند چرا که با کم کردن وزن از پیچیدگی آن کم می‌کنیم و از overfit جلوگیری می‌کند
مقدار ۰.۰۱ برای اکثر شبکه‌ها مفید است ولی در این پروژه چوت ۷ لایه زیاد بوده است مقدار این باعث کاهش دقت شبکه و همگرا نشدن شبکه شده است.