

مهدی فتیعی

401722136

پروژه سوم شناسایی الگو:

در قسمت اول از ما خواسته شده بود که فیلم توضیح شما را ببینیم و نحوه کارکرد با pytorch و ساخت یک شبکه عصبی در pytorch آموزش داده شده بود که انجام شد. و حاصل آن استفاده از کتابخانه زیر بود.

```
import numpy as np
import matplotlib.pyplot as plt
import torch
import torchvision
from torchvision import datasets, transforms
from torch import nn, optim
import cv2
from tqdm import tqdm
```

و به کمک قطعه کد زیر متوجه شدم که مدل بر روی کدام قسمت انجام می‌شود، Cpu یا کارت گرافیک.

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

Device cuda
```

از اونجا که می‌خواستم روی یک سری داده که عکس باشند کار کنم پس یک مدل transforms ساختم که با احتمال نیم درصد داده‌ها را Flip یعنی چرخش در آن ایجاد کند و به قسمت‌های از تصویر را به صورت تصادفی Crop می‌کنیم . همچنین به کمک normalization ، مقادیر میانگین و انحراف از معیار بر اساس بررسی transform

هایی که بر روی داده‌ای که در مرحله بعد درباره آن صحبت می‌کنم set شده است.

```
train_transform = transforms.Compose([
    transforms.RandomHorizontalFlip(p=0.5),
    transforms.RandomCrop(32, padding=4),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
])

test_transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
])
```

در قسمت دوم پروژه از لینک داده شده و با توجه به توصیه خوب شما دیتاست CIFAR10 برای کارکردن بر روی موارد خواسته شده پروژه انتخاب کردم . پس به کمک ماژول datasets از لایبرری techvision دانلودش می‌کنم و براساس transform مرحله قبل داده‌ها را به صورت ساختار یافته نگه می‌دارم .

```
dataset = datasets.CIFAR10(root='data/', download=True, transform=train_transform)
test_dataset = datasets.CIFAR10(root='data/', download=True, train=False, transform=test_transform)

Files already downloaded and verified
Files already downloaded and verified

dataset

Dataset CIFAR10
  Number of datapoints: 50000
  Root location: data/
  Split: Train
  StandardTransform
  Transform: Compose(
    RandomHorizontalFlip(p=0.5)
    RandomCrop(size=(32, 32), padding=4)
    ToTensor()
    Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
  )
```

سپس در قدم بعدی کلاس‌های دیتاست خود را نگاه می‌کنیم .

```
classes = dataset.classes
classes

['airplane',
 'automobile',
 'bird',
 'cat',
 'deer',
 'dog',
 'frog',
 'horse',
 'ship',
 'truck']
```

سپس نگاه می‌کنیم از هرکدام چقدر در داده‌های مربوط به train داریم .

```
class_count = {}
for _, index in dataset:
    label = classes[index]
    if label not in class_count:
        class_count[label] = 0
    class_count[label] += 1
class_count

{'frog': 5000,
 'truck': 5000,
 'deer': 5000,
 'automobile': 5000,
 'bird': 5000,
 'horse': 5000,
 'ship': 5000,
 'cat': 5000,
 'dog': 5000,
 'airplane': 5000}
```

بعد نگاه می‌کنیم به test , train shape .

```
print(dataset.data.shape)
# print(dataset.targets)
print(test_dataset.data.shape)
# print(test_dataset.targets.shape)

(50000, 32, 32, 3)
(10000, 32, 32, 3)
```

سپس از ۵۰۰۰۰ داده train حدود ۵۰۰۰ را به صورت تصادفی به عنوان داده validation در هنگام training قرار می‌دهیم .

```
torch.manual_seed(43)
val_size = 5000
train_size = len(dataset) - val_size

from torch.utils.data import random_split
train_ds, val_ds = random_split(dataset, [train_size, val_size])
len(train_ds), len(val_ds)

(45000, 5000)
```

در مرحله بعد با در نظر گرفتن batch size به میزان ۱۲۸ و تعداد انجام عملیات موازی به تعداد ۴ و shuffle شدن داده‌های train در هر epoch داده‌های train , validation , test را ساختار یافته‌تر کردم برای مرحله آموزش.

```
batch_size = 128
```

Add Code Cell Add Markdown Cell

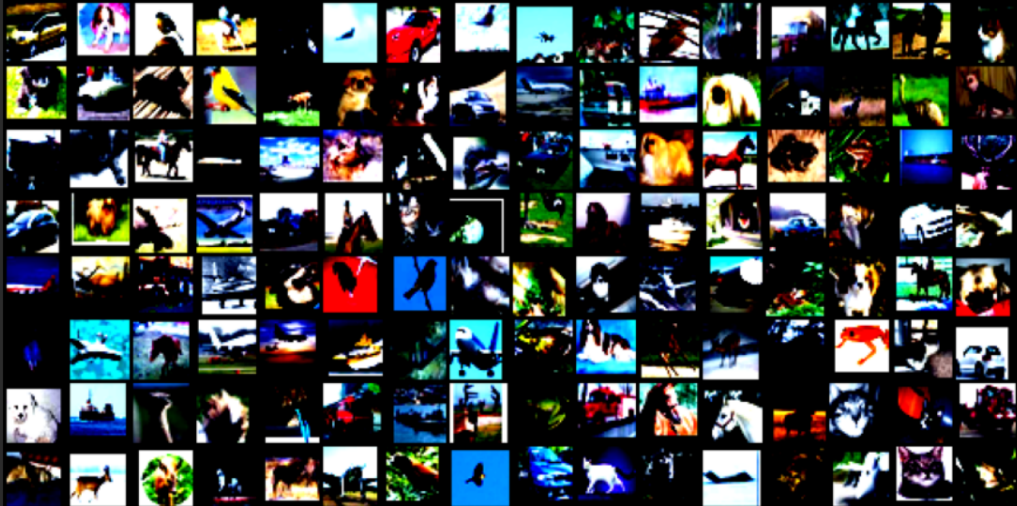
```
train_loader = torch.utils.data.DataLoader(train_ds, batch_size=batch_size, num_workers=4, shuffle=True)
val_loader = torch.utils.data.DataLoader(val_ds, batch_size = batch_size*2, num_workers=4, pin_memory=True)
test_loader = torch.utils.data.DataLoader(test_dataset, batch_size=batch_size*2, num_workers=4, shuffle=True, pin_memory=True)
```

سپس در مرحله بعد به ازای هر کلاس یک دید تصویری نسبت به داده‌هایی که می‌خواهیم یاد بگیریم به کمک کد زیر پیدا کردم.

```
from torchvision.utils import make_grid
for images, _ in train_loader:
    print('images.shape:', images.shape)
    plt.figure(figsize=(16,8))
    plt.axis('off')
    plt.imshow(make_grid(images, nrow=16).permute((1, 2, 0)))
    break
```

WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers)

images.shape: torch.Size([128, 3, 32, 32])



خوب پس از انجام این مقدمات آماده برای مراحل سه تا شش شدم.

برای مرحله سوم از ما خواسته شده بود که یک شبکه mlp با پیش از یک لایه مخفی را آموزش دهیم و که برای انجام این کار یک شبکه با ساختار زیر ساختیم .

```
input_size = 3*32*32
output_size = 10
hidden_size = [256,128]

model = nn.Sequential(
    nn.Flatten(),
    nn.Linear(input_size, hidden_size[0]),
    nn.ReLU(),
    nn.Linear(hidden_size[0], hidden_size[1]),
    nn.ReLU(),
    nn.Linear(hidden_size[1], output_size)
)

print(model)

Sequential(
  (0): Flatten(start_dim=1, end_dim=-1)
  (1): Linear(in_features=3072, out_features=256, bias=True)
  (2): ReLU()
  (3): Linear(in_features=256, out_features=128, bias=True)
  (4): ReLU()
  (5): Linear(in_features=128, out_features=10, bias=True)
)
```

در لایه اول مدل دوبعدی را flat کردم و به تعداد بیت‌های حاصل ورودی در نظر گرفتم سپس یک لایه با ۲۵۶ نرون خروجی و یک لایه دیگر با ۲۵۶ نرون ورودی و ۱۲۸ نرون خروجی و در نهایت لایه آخر که لایه تصمیم هست یک لایه با ۱۲۸ و ورودی ۱۰ نرون خروجی به ازای ۱۰ کلاسمان در نظر گرفتم .

در این مرحله چند فانکشن که در ادامه تمام شش سری از آن استفاده می‌کنیم، توجه می‌کنیم .

```
def accuracy(outputs, labels):
    _, preds = torch.max(outputs, dim=1)
    return torch.tensor(torch.sum(preds == labels).item() / len(preds))
```

در فانکشن بالا به ازای label های ورودی و label های پیش‌بینی شده توسط مدل ما یک میزان دقت بدست می‌آوریم که برابر است با تعداد label های هایی که درست پیش‌بینی کرده‌ایم به تعداد کل label ها . در واقع کلاس ما به ازای هر کلاس یک مقداری را پیش‌بینی می‌کند که آنکه بیشتر از همه است را به عنوان معیار پیش‌بینی اصلی در نظر می‌گیریم .

```
import torch.nn.functional as F

def training_step(model, batch):
    images, labels = batch[0].to(device), batch[1].to(device) # load the batch to the available device (cpu/gpu)
    out = model(images) # Generate predictions
    loss = F.cross_entropy(out, labels) # Calculate loss
    return loss
```

در فانکشن training step به ازای هر batch از داده یک loss براساس cross entropy پیش بینی مدل از هر batch پیدا می کنیم و برمی گردانیم .
همین کار را برای داده های validation هم انجام می دهیم در تابع validation step با این تفاوت میزان دقت و accuracy را نیز برای این مدل پیدا می کنیم برای داده های validation.

```
def validation_step(model, batch):
    images, labels = batch[0].to(device), batch[1].to(device) # load the batch to the available device (cpu/gpu)
    out = model(images) # Generate predictions
    loss = F.cross_entropy(out, labels) # Calculate loss
    acc = accuracy(out, labels) # Calculate accuracy
    return {'val_loss': loss.detach(), 'val_acc': acc}
```

در انتهای هر epoch به کمک تابع زیر یک دیکشنری از برمیگردانیم که میزان میانگین , loss , accuracy را به ازای تمامی batch ها پیدا می کند .

```
def validation_epoch_end(outputs):
    batch_losses = [x['val_loss'] for x in outputs]
    epoch_loss = torch.stack(batch_losses).mean() # Combine losses
    batch_accs = [x['val_acc'] for x in outputs]
    epoch_acc = torch.stack(batch_accs).mean() # Combine accuracies
    return {'val_loss': epoch_loss.item(), 'val_acc': epoch_acc.item()}
```

و در انتها مهمترین بخش کد را می بینیم .

```
def evaluate(model, val_loader):
    outputs = [validation_step(model, batch) for batch in val_loader]
    return validation_epoch_end(outputs)

def fit(epochs, lr, model, train_loader, val_loader, opt_func=torch.optim.SGD):
    history = []
    optimizer = opt_func(model.parameters(), lr, weight_decay=0.0005)
    for epoch in range(epochs):
        # Training Phase
        for batch in train_loader:
            loss = training_step(model, batch)
            loss.backward()
            optimizer.step()
            optimizer.zero_grad()
        # Validation phase
        result = evaluate(model, val_loader)
        epoch_end(epoch, result)
        history.append(result)
    return history
```

در این بخش ما یک تابع fit که کار train داده را انجام می دهد .
تابع fit ما یک تعداد epochs، میزان learning rate، یک دیتاست train و validation و یک function که مشخص کننده نوع optimizer ما هست را به عنوان ورودی می گیرد .
یک لیست به عنوان تاریخچه کار train در نظر میگیریم و optimizer را براساس تابع optimizer گرفته شده و براساس $weight\ decay = 0.0005$, $learning\ rate$ می سازم .
پس به ازای هر epoch یک batch از داده های train خودم بیرون می کشم و میزان loss را بدست می آورم براساس میزان loss به کمک backpropagation آپدیت می کنم سپس میزان accuracy , loss را براساس وزن های جدید بر روی داده validation set بدست می آورم و در history خود نگه می دارم.

در انتها به کمک فانکشن زیر یک نمودار براساس loss , یک نمودار براساس accuracy در مراحل مختلف history می کشم .

```
def plot_losses(history):  
    losses = [x['val_loss'] for x in history]  
    plt.plot(losses, '-x')  
    plt.xlabel('epoch')  
    plt.ylabel('loss')  
    plt.title('Loss vs. No. of epochs');
```

```
def plot_accuracies(history):  
    accuracies = [x['val_acc'] for x in history]  
    plt.plot(accuracies, '-x')  
    plt.xlabel('epoch')  
    plt.ylabel('accuracy')  
    plt.title('Accuracy vs. No. of epochs');
```

حالا به کمک تعریف این function سراغ آموزش مدل شبکه عصبی ساده ای که در ابتدا کار درباره آن صحبت کردیم .

```
lrs = [1e-1,1e-2,1e-3,1e-4]
```

در چهار مرحله با learning rate بالا شبکه را آموزش می دهیم .

```
from datetime import datetime  
for lr in lrs:  
  
    print(f"-----({datetime.now()})-----")  
    history += fit(10, lr, model, train_loader, val_loader)  
    print(f"-----({datetime.now()})-----")
```

در نتیجه خواهیم داشت .

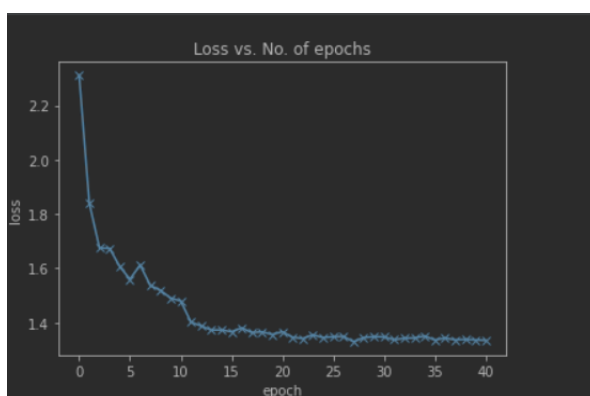
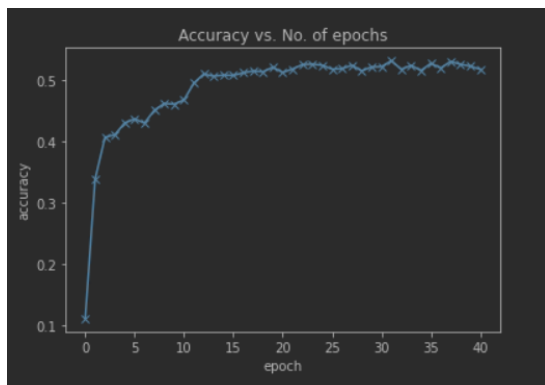
```
----- (2022-12-21 16:21:17.213201) -----  
Epoch [0], val_loss: 1.8400, val_acc: 0.3387  
Epoch [1], val_loss: 1.6752, val_acc: 0.4069  
Epoch [2], val_loss: 1.6734, val_acc: 0.4112  
Epoch [3], val_loss: 1.6063, val_acc: 0.4304  
Epoch [4], val_loss: 1.5600, val_acc: 0.4369  
Epoch [5], val_loss: 1.6124, val_acc: 0.4301  
Epoch [6], val_loss: 1.5374, val_acc: 0.4510  
Epoch [7], val_loss: 1.5190, val_acc: 0.4621  
Epoch [8], val_loss: 1.4909, val_acc: 0.4609  
Epoch [9], val_loss: 1.4796, val_acc: 0.4680  
----- (2022-12-21 16:24:01.070258) -----
```

```
----- (2022-12-21 16:24:01.070311) -----  
Epoch [0], val_loss: 1.4020, val_acc: 0.4952  
Epoch [1], val_loss: 1.3881, val_acc: 0.5101  
Epoch [2], val_loss: 1.3731, val_acc: 0.5065  
Epoch [3], val_loss: 1.3734, val_acc: 0.5084  
Epoch [4], val_loss: 1.3668, val_acc: 0.5078  
Epoch [5], val_loss: 1.3809, val_acc: 0.5121  
Epoch [6], val_loss: 1.3633, val_acc: 0.5150  
Epoch [7], val_loss: 1.3663, val_acc: 0.5137  
Epoch [8], val_loss: 1.3567, val_acc: 0.5212  
Epoch [9], val_loss: 1.3664, val_acc: 0.5126  
----- (2022-12-21 16:26:44.866870) -----
```

```
----- (2022-12-21 16:26:44.867181) -----  
Epoch [0], val_loss: 1.3465, val_acc: 0.5184  
Epoch [1], val_loss: 1.3417, val_acc: 0.5254  
Epoch [2], val_loss: 1.3553, val_acc: 0.5262  
Epoch [3], val_loss: 1.3451, val_acc: 0.5243  
Epoch [4], val_loss: 1.3483, val_acc: 0.5180  
Epoch [5], val_loss: 1.3494, val_acc: 0.5187  
Epoch [6], val_loss: 1.3301, val_acc: 0.5244  
Epoch [7], val_loss: 1.3447, val_acc: 0.5152  
Epoch [8], val_loss: 1.3487, val_acc: 0.5216  
Epoch [9], val_loss: 1.3476, val_acc: 0.5220  
----- (2022-12-21 16:29:26.013023) -----
```

```
----- (2022-12-21 16:29:26.013098) -----  
Epoch [0], val_loss: 1.3372, val_acc: 0.5317  
Epoch [1], val_loss: 1.3430, val_acc: 0.5178  
Epoch [2], val_loss: 1.3437, val_acc: 0.5241  
Epoch [3], val_loss: 1.3508, val_acc: 0.5158  
Epoch [4], val_loss: 1.3363, val_acc: 0.5274  
Epoch [5], val_loss: 1.3438, val_acc: 0.5199  
Epoch [6], val_loss: 1.3367, val_acc: 0.5305  
Epoch [7], val_loss: 1.3385, val_acc: 0.5259  
Epoch [8], val_loss: 1.3366, val_acc: 0.5229  
Epoch [9], val_loss: 1.3350, val_acc: 0.5178  
----- (2022-12-21 16:32:09.410399) -----
```

همانطور که می بینید در مرحله چهارم هرچند که میزان learning rate را هم کاهش داده بودیم ولی خوب در حال overfit شدن بودیم و دقت بهبود چندانی پیدا نکرد .
با هم نمودارها را در مراحل مختلف یادگیری بررسی میکنیم بر روی داده validation set.



و سپس میزان دقت و loss را بر روی داده تست را بررسی می کنیم .

```
evaluate(model, test_loader)
```

```
{'val_loss': 1.371119737625122, 'val_acc': 0.512988269329071}
```

در قسمت چهار و پنج از ما خواسته شده بود که یک دو مدل CNN ساده را پیاده سازی کنیم با این فرض که در مدل دومی با تمامی شباهت‌هایی که مدل اول دارد صرفاً Batch-Normalization و Dropout را اضافه می‌کنیم.

مدل شبکه اول :

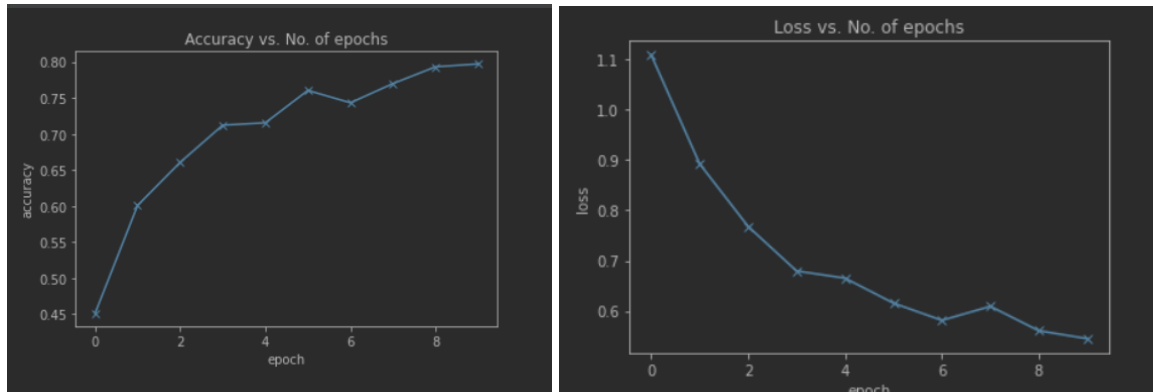
```
model_cnn = nn.Sequential(  
    nn.Conv2d(3, 32, kernel_size=3, padding=1),  
    nn.ReLU(),  
    nn.Conv2d(32, 64, kernel_size=3, stride=1, padding=1),  
    nn.ReLU(),  
    nn.MaxPool2d(2, 2), # output: 64 x 16 x 16  
    # nn.BatchNorm2d(64),  
  
    nn.Conv2d(64, 128, kernel_size=3, stride=1, padding=1),  
    nn.ReLU(),  
    nn.Conv2d(128, 128, kernel_size=3, stride=1, padding=1),  
    nn.ReLU(),  
    nn.MaxPool2d(2, 2), # output: 128 x 8 x 8  
    # nn.BatchNorm2d(128),  
  
    nn.Conv2d(128, 256, kernel_size=3, stride=1, padding=1),  
    nn.ReLU(),  
    nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1),  
    nn.ReLU(),  
    nn.MaxPool2d(2, 2), # output: 256 x 4 x 4  
    # nn.BatchNorm2d(256),  
  
    nn.Flatten(),  
    nn.Linear(256*4*4, 10),  
)
```

مدل شبکه دوم :

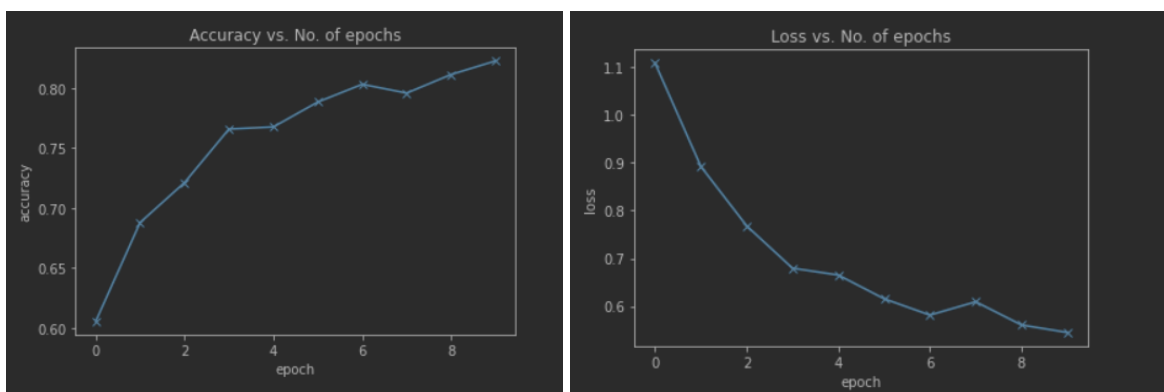
```
model_cnn2 = nn.Sequential(  
    nn.Conv2d(3, 32, kernel_size=3, padding=1),  
    nn.ReLU(),  
    nn.Conv2d(32, 64, kernel_size=3, stride=1, padding=1),  
    nn.ReLU(),  
    nn.MaxPool2d(2, 2), # output: 64 x 16 x 16  
    nn.Dropout(0.2), # 20% Probability  
    nn.BatchNorm2d(64),  
  
    nn.Conv2d(64, 128, kernel_size=3, stride=1, padding=1),  
    nn.ReLU(),  
    nn.Conv2d(128, 128, kernel_size=3, stride=1, padding=1),  
    nn.ReLU(),  
    nn.MaxPool2d(2, 2), # output: 128 x 8 x 8  
    nn.Dropout(0.2), # 20% Probability  
    nn.BatchNorm2d(128),  
  
    nn.Conv2d(128, 256, kernel_size=3, stride=1, padding=1),  
    nn.ReLU(),  
    nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1),  
    nn.ReLU(),  
    nn.MaxPool2d(2, 2), # output: 256 x 4 x 4  
    nn.Dropout(0.2), # 20% Probability  
    nn.BatchNorm2d(256),  
  
    nn.Flatten(),  
    nn.Linear(256*4*4, 10),  
)
```

دقت شود که برای هر دو از ۱۰ epoch و 0.001 learning rate استفاده کرده‌ام .
و در این دو مسئله از optimizare Adam استفاده کرده‌ام .

مدل اول CNN نمودارها براساس validation set در هنگام training :



مدل دوم CNN نمودارها براساس validation set در هنگام training :



پس از انجام مرحله train به بررسی دو مدل بر روی داده test پرداختم .

خروجی دقت و loss برای دو مدل بر روی داده تست به ترتیب برابر خواهد بود با :

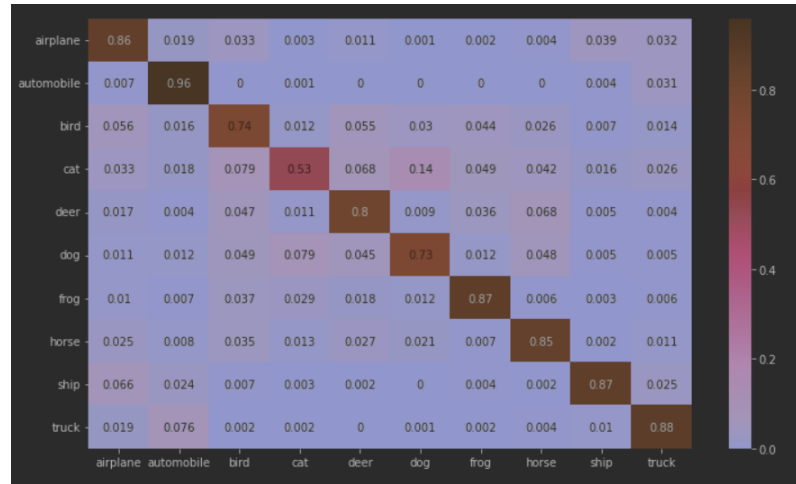
```
{'val_loss': 0.5280101299285889, 'val_acc': 0.818066418170929}
```

```
{'val_loss': 0.5673812627792358, 'val_acc': 0.8121093511581421}
```

همانطور که می بینید مدل دوم از دقت بالاتر و میزان loss کمتری برخوردار هست و همه شرایط را هم یکسان در نظر گرفته ایم .

ابتدا confusion ماتریس را بررسی کنیم .

مدل CNN اول بدون
Dropout , batch normalize
همانطور که می بینید دقت این
مدل روی گربه بسیار کم است و
سگ و پرند را گهگاهی با آن
اشتباه در نظر میگیرد .



در مدل CNN دومی دقت
پیش بینی گربه به نحو خوبی
افزایش پیدا می کند ولی خوب



بیایم برای بررسی معیار دوم Precision هر دو مدل را کنار هم قرار دهیم به ازای هر کلاس .

```
airplane : 0.7835888187556357
automobile : 0.9148296593186372
bird : 0.7447236180904523
cat : 0.7096774193548387
deer : 0.8449258836944128
dog : 0.8065610859728507
frog : 0.782871972318339
horse : 0.8528265107212476
ship : 0.8746298124383021
truck : 0.8774703557312253
```

```
airplane : 0.7781818181818182
automobile : 0.838737949167397
bird : 0.7191448007774538
cat : 0.7763157894736842
deer : 0.7795121951219512
dog : 0.7758985200845666
frog : 0.8482490272373541
horse : 0.8097050428163654
ship : 0.9050104384133612
truck : 0.8516377649325626
```

اگر به صورت میانگین این حاصل را نگاه کنیم برابر خواهد بود با :

```
Precision: 0.819100
```

```
Precision: 0.809100
```

همانطور که می بینید به صورت میانگین یک درصد افزایش پیدا کرده است .
بیایم برای بررسی معیار سوم Recall هر دو مدل را کنار هم قرار دهیم به ازای هر کلاس .

```
airplane : 0.869  
automobile : 0.913  
bird : 0.741  
cat : 0.66  
deer : 0.741  
dog : 0.713  
frog : 0.905  
horse : 0.875  
ship : 0.886  
truck : 0.888
```

```
airplane : 0.856  
automobile : 0.957  
bird : 0.74  
cat : 0.531  
deer : 0.799  
dog : 0.734  
frog : 0.872  
horse : 0.851  
ship : 0.867  
truck : 0.884
```

اگر به صورت میانگین این حاصل را نگاه کنیم برابر خواهد بود با :

```
Recall: 0.819100
```

```
Recall: 0.809100
```

همانطور که می بینید به صورت میانگین یک درصد افزایش پیدا کرده است .

حالا بریم سراغ معیار F1:

```
airplane : 0.8240872451398766  
automobile : 0.9139139139139139  
bird : 0.7428571428571429  
cat : 0.683937823834197  
deer : 0.7895578050079914  
dog : 0.7569002123142251  
frog : 0.839517625231911  
horse : 0.8637709772951628  
ship : 0.8802781917536016  
truck : 0.882703777335984
```

```
airplane : 0.8152380952380952  
automobile : 0.8939747781410554  
bird : 0.7294233612617051  
cat : 0.6306413301662708  
deer : 0.7891358024691358  
dog : 0.7543679342240494  
frog : 0.8599605522682445  
horse : 0.8298391028766455  
ship : 0.8855975485188968  
truck : 0.8675171736997055
```

اگر به صورت میانگین این حاصل را نگاه کنیم برابر خواهد بود با :

```
F1 score: 0.819100
```

```
F1 score: 0.809100
```


قسمت شش از ما خواسته شده بود که با transfer learning آشنا شویم .

برای انجام این مهم چندین سوال پرسیده شده بود .

الف) مزایای استفاده از transfer learning را توضیح دهید.

فکر کنیم یک ماشین یادگیری داشته باشیم که به خوبی سگ‌ها را پیش بینی می کند .

حال می‌خواهیم یک ماشین یادگیری دیگه داشته باشیم که این ماشین گربه‌ها را به خوبی تشخیص دهد .

به دلیل شباهت ظاهری هر دو مسئله ، می‌توانیم از ماشین یادگیری قبلی برای یادگیری concept جدید استفاده

کنیم .

خوب حقیقت این است که احتمالاً برای این ماشین جدید هم نیازی نیست که یک معماری جدید ماشین لرنینگ

جدید به کار ببریم . پس از همان معماری قبلی استفاده می‌کنیم از طرفی پارامترهایی که برای معماری جدید ما

بدست آمده منحصر به فرد به یادگیری سگ بوده است آیا می‌شود پارامترهای یادگیری گربه را نیز همانند همین مورد

قرار دهیم ؟ جواب این هست که همه پارامترها نه ولی بسیاری بله .

پس استفاده از یک معماری که به خوبی جواب گرفته و عدم تغییر بسیاری از پارامترهای یادگیری موجب: ۱

۱. کاهش تایم ساخت یک محصول یادگیری

۲. کاهش زمان یک یادگیری ماشینی به دلیل داشتن مقدار قابل توجهی از پارامترهای یادگیری است که توسط

یک ماشین یادگیری دیگر یاد گرفته شده است .

۳ . کاهش تایم رابطه مستقیمی با کاهش هزینه دارد که در نتیجه ساخت یک ماشین یادگیری جدید ارزان تر

می‌شود .

۴ . امکان می‌دهد که یک ماشین یادگیری را در طول زمان‌های مختلف نسبت به داده‌های جدید که در طول

زمان بوجود آمدن با هزینه زمانی و مالی کمتری آپدیت نگه داریم یعنی فرض کنید ما یک ماشین داریم که سگ‌ها

را با دقت بالای پیش بینی می‌کند اما نسبت به یک نوع سگ که در داده‌های آموزش ما نیست دقت پیش بینی

مناسپی ندارد، برای اینکه دقت ماشین را نسبت به این قضیه بالا ببریم می‌توانیم بخش‌های پایه‌ای ماشین یادگیری را

freeze کنیم و فقط بخش‌های انتهایی که در تصمیم‌گیری مورد نیاز هست را به کمک مجموعه دادگان گذشته و

داده اضافه شده آپدیت کنیم تا نسبت به مجموعه دادگان جدید نیز دقت مناسبی داشته باشد .

۵. دقت بالاتر بعد از آموزش: با نقطه شروع بهتر و نرخ یادگیری بالاتر، یادگیری انتقالی یک مدل یادگیری ماشینی

را برای همگرایی در سطح عملکرد بالاتر فراهم می‌کند و خروجی دقیق‌تری را ممکن می‌سازد.

ب) در صورتی که مجموعه داده کوچک باشد یا بزرگ باشد و یا مجموعه داده فعلی به مجموعه داده مدل اولیه شباهت زیادی داشته باشد و یا شباهتی کمی داشته باشد چه کاری باید انجام داد. (هر کدام از 4 حالت ذکر شده را تحلیل کنید.)

وقتی مجموعه داده کوچک باشد :

اگر داده‌ها کم باشد ولی به دیتاهای مدل transfer ما که می‌خواهیم از آن استفاده کنیم شبیه باشد، فقط لایه‌های آخر که مربوط به تصمیم‌گیری است، پارامترها را freeze نمی‌کنیم .
اگر داده‌ها کم باشد ولی به مدل اولیه شباهتی نداشته باشد باید لایه‌های بیشتری را آزاد برای یادگیری بگذاریم ولی این کار موجب افزایش احتمال overfit شدن رو داده کم می‌شود که می‌تواند خطرناک باشد پس بهتر است که داده‌های ما تا حد امکان به داده‌های مدل transform نزدیک باشد. هرچند که با data augmentation می‌شود تعداد داده‌های train را زیاد کرد اما اگر با data augmentation نیز به تعداد مورد نیاز داده برای آموزش نرسیم به دلیل عدم شباهت با مدل transform و احتمال زیاد کردن لایه‌های که نباید freeze شوند احتمال overfit شدن افزایش می‌یابد .

وقتی مجموعه داده زیاد باشد :

اگر داده‌ها به مدل transfer نزدیک باشند، باز نیز فقط لایه‌های آخر را freeze نمی‌کنیم البته به میزانی که تعداد داده‌های ما بیشتر می‌شود باید تا یک حد متوسطی از عمق شبکه را اجازه بدهیم تا دیتاست جدید ما وزن‌ها را تغییر دهد تا کمک کنیم میزان یادگیری مدل جدید بشینه شود، در این حالت مهمترین مسئله برای ما زمان خواهد بود و دقت که بین آن‌ها یک رابطه عکس وجود دارد هرچقدر دقت را بالاتر ببریم الان نیاز به زمان بیشتر دارد و برعکس پس براساس این دو معیار عمق تغییرات پارامترها را حساب می‌کنیم .

اگر از میزان شباهت دورتری برخوردار تر باشند به طور ویژه‌تری مجبور می‌شویم عمق تغییرات پارامتر را بیشتر و میزان لایه‌های کمتری را freeze کنیم این عمق می‌تواند حتی به اولین لایه‌ها متناسب با میزان دوری شباهت برسد که در این حالت صرفاً ما می‌خواهیم از معماری مدل مربوط برای پیدا کردن بهترین دقت استفاده کنیم .

حال برای پاسخگویی به قسمت ج من دوباره از CIFAR 10 برای پاسخگویی استفاده کردم با همون شیوه تغییرات برای پاسخ دادن قسمت های قبلی و هیچ تغییری به وجود نیاوردم .
برای انجام این کار از transform که مربوط به پردازش تصویر هست یعنی VGG16 استفاده کردم .

```
vgg16 = torchvision.models.vgg16(pretrained=True)
vgg16
```

با همدیگه نگاهی به معماری این transform بندازیم .

```
VGG(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU(inplace=True)
    (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): ReLU(inplace=True)
    (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (6): ReLU(inplace=True)
    (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (8): ReLU(inplace=True)

    (9): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU(inplace=True)
    (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (13): ReLU(inplace=True)
    (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (15): ReLU(inplace=True)
    (16): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (17): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (18): ReLU(inplace=True)
    (19): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))

    (20): ReLU(inplace=True)
    (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (22): ReLU(inplace=True)
    (23): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (24): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (25): ReLU(inplace=True)
    (26): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (27): ReLU(inplace=True)
    (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (29): ReLU(inplace=True)
    (30): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)

  ),
  (avgpool): AdaptiveAvgPool2d(output_size=(7, 7))
  (classifier): Sequential(
    (0): Linear(in_features=25088, out_features=4096, bias=True)
    (1): ReLU(inplace=True)
    (2): Dropout(p=0.5, inplace=False)
    (3): Linear(in_features=4096, out_features=4096, bias=True)
    (4): ReLU(inplace=True)
    (5): Dropout(p=0.5, inplace=False)
    (6): Linear(in_features=4096, out_features=1000, bias=True)
  )
)
```

چونکه لایه آخر این مدل برای کلاس بندی ۱۰۰۰ کلاس هست و ما ۱۰ کلاس داریم پس باید لایه classifier را تغییر دهیم .

در نتیجه به شکل زیر کلاس classifier را تغییر می دهیم .

```
class Network(nn.Module):
    def __init__(self, pretrained_model):
        super().__init__()
        self.features = nn.Sequential(
            *list(pretrained_model.features.children())
        )
        self.classifier = nn.Sequential(
            nn.Linear(in_features=512, out_features=256),
            nn.ReLU(inplace=True),
            nn.Linear(in_features=256, out_features=128),
            nn.ReLU(inplace=True),
            nn.Linear(in_features=128, out_features=10)
        )

    def forward(self, t):
        t = self.features(t)
        t = t.reshape(t.size(0), -1)
        t = self.classifier(t)

    return t

vgg16 = Network(vgg16)
vgg16
```

همانطور که مشاهده می کنید لایه features را هیچ تغییری ندادم و صرفا لایه classification را متناسب با ۱۰ کلاس کوچک کردم .

در قسمت بعدی لایه هایی از مدل که باید freeze می کردم را مشخص کردم .

```
# transfer learning (first 8 layers of vgg16)
# freeze the transferred weights which won't be trained
for layer_num, child in enumerate(vgg16.features.children()):
    if layer_num < 19:
        for param in child.parameters():
            param.requires_grad_(False)
```

متناسب با لایه هایی که باید پارامترهایشان learn می شد ، optimizer را تعریف کردم به شکل زیر .

البته در این سری میزان learning rate را برای لایه features و classification متفاوت در نظر گرفتم .

```
optimizer = optim.SGD(
    [
        # parameters which need optimization
        {'params': vgg16.features[19:].parameters(), 'lr': 0.001},
        {'params': vgg16.classifier.parameters()}
    ], lr=0.01, momentum=0.9, weight_decay=1e-3)

history = train(vgg16, train_loader, val_loader, 30, optimizer)
```

هرچند که بدنه تابع fit برای یادگیری این مدل هم خوب کار می کرد اما خوب ترجیح دادم یک تابع دیگر به اسم train بسازم همانند تابع fit که در هر epoch مدل یادگرفته شده نیز ذخیره می شد به دلیل اینکه ممکن بود colab قطع شود ، پس تابع را به صورت زیر باز تعریف کردم .

```
def train(model, train_set, valid_loader, num_epochs, optimizer):

    history = []
    comment = f'-transferlr_vgg16' # will be used for naming the run
    model.to(device)

    for epoch in range(num_epochs):
        #####
        # train the model #
        #####
        train_loop = tqdm(train_loader)
        model.train() # set the model to train mode
        for batch in train_loop:

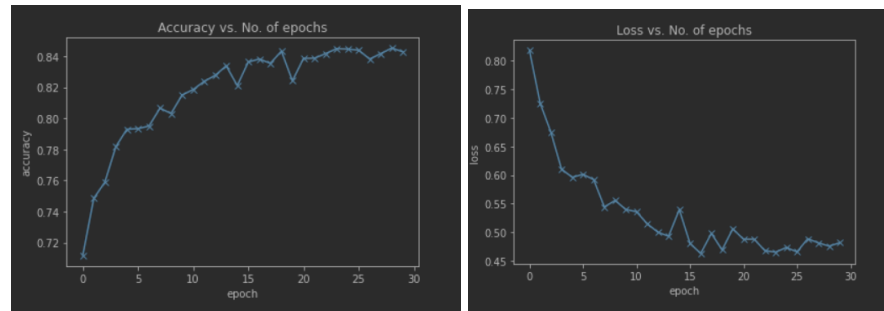
            loss = training_step(model, batch) # calculate loss
            optimizer.zero_grad() # clear accumulated gradients from the previous pass
            loss.backward() # backward pass
            optimizer.step() # perform a single optimization step

        model.eval() # set the model to evaluation mode
        with torch.no_grad(): # turn off grad tracking, as we don't need gradients for validation
            #####
            # validate the model #
            #####
            result = evaluate(model, val_loader)
            epoch_end(epoch, result)
            history.append(result)

        torch.save(model.state_dict(), f'transferlr_vgg16.pth')

    return history
```

در نهایت به میزان یادگیری ماشین در ۱۰ epoch نگاهی بیندازیم .

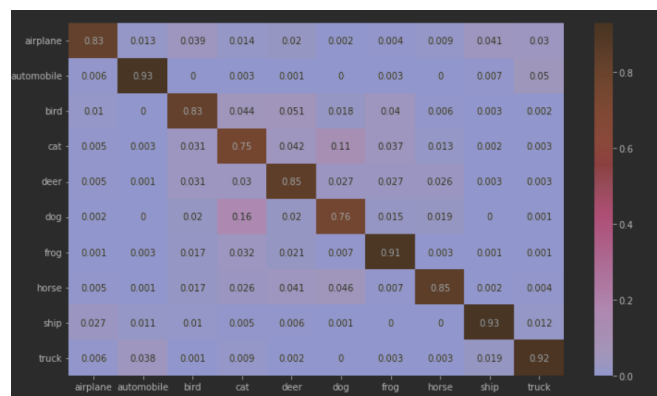


و میزان دقت بر روی داده تست :

```
evaluate(vgg16, test_loader)

{'val_loss': 0.44929057359695435, 'val_acc': 0.8534179925918579}
```

برای این مدل confusion ماتریس برابر است با :



همچنین precision برابر است با :

```
airplane : 0.9251396648044693
automobile : 0.93
bird : 0.8326612903225806
cat : 0.7001862197392924
deer : 0.8058991436726927
dog : 0.781985670419652
frog : 0.8704761904761905
horse : 0.9150537634408602
ship : 0.9224652087475149
truck : 0.8965853658536586
```

همچنین Recall به ازای هر کلاس برابر است با :

```
airplane : 0.828
automobile : 0.93
bird : 0.826
cat : 0.752
deer : 0.847
dog : 0.764
frog : 0.914
horse : 0.851
ship : 0.928
truck : 0.919
```

همچنین برای F1 به ازای هر کلاس برابر است با :

```
airplane : 0.8738786279683377
automobile : 0.93
bird : 0.8293172690763052
cat : 0.7251687560270011
deer : 0.825938566552901
dog : 0.7728882144663631
frog : 0.8917073170731707
horse : 0.8818652849740932
ship : 0.9252243270189431
truck : 0.9076543209876544
```

اگر میانگین هر سه را نگاهی بندازیم خواهیم داشت :

```
Precision: 0.855900
-----
Recall: 0.855900
-----
F1 score: 0.855900
```

خوب تمامی معیارها نشان دهنده این بود که :

یک مدل با transform از همه بهتر دقت دارد روی داده‌های CIFAR10
در جایگاه بعد یک مدل CNN ساده با batch normalize و Droupout قرار می‌گیرد
بعد یک مدل CNN ساده .
و در انتها یک mlp با دو لایه .

نکات تکمیلی :

در این مقاله از دو تابع بهینه سازی SGD و ADAM استفاده شد .

برای قسمت آخر و قسمت مربوط به mlp ساده از SGD و قسمت چهار و پنج که مقایسه دو نوع CNN بود را از optimizer Adam استفاده کردم .

تمامی تابع خطای من در این پروسه، cross entropy بود .

چرا cross entropy را برای پیدا کردن Loss انتخاب کردم ؟

1. Using the cross-entropy error function instead of the sum-of-squares for a classification problem leads to **faster training** as well as **improved generalization**.
2. Cross-entropy loss, or log loss, measure the performance of a classification model whose output is a probability value between 0 and 1
3. Cross-entropy is preferred for **classification**
4. However, when using the softmax function as the output layer activation, along with cross-entropy loss, you can compute gradients that facilitate **backpropagation**. The gradient evaluates to a simple expression, easy to interpret and intuitive
5. I don't know any better loss function for do this project :