به نام خدا

تمرین چهارم یادگیری عمیق

غزل زمانی‌نژاد

۹۷۵۲۲۱۶۶

1. به الگوریتم‌های AdaGrad، Adadelta، RMSProp و Adam الگوریتم‌های adaptive می‌گوییم زیرا می‌توانند نرخ یادگیری را بر پارامترها تطبیق دهند. RMSProp تقریبا مشابه AdaGrad است اما مشکل کوچک شدن نرخ یادگیری به صورت رادیکالی را برطرف کرده‌است. Adam عملیات bias correction را بر روی RMSProp انجام می‌دهد و از momentum نیز استفاده می‌کند.

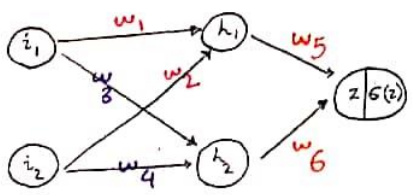
اگر دیتاستی با داده‌های پراکنده داریم، بهتر است از بهینه‌سازهای تطبیقی استفاده کنیم. چون این بهینه‌سازها در راستایی که تغییرات گرادیان زیاد است، نرخ یادگیری را کاهش می‌دهند تا در گام‌های کوچکتری جستجو کنند. و در راستایی که تغییرات گرادیان کم است، نرخ یادگیری را افزایش می‌دهند. یعنی در این گونه از بهینه‌سازها نیازی به fine-tune کردن نرخ یادگیری به صورت دستی نیست. در میان این بهینه‌سازها، معمولا Adam بهتر است.

از طرفی در بسیاری از مقالات از SGD همراه با کاهش نرخ یادگیری (annealing schedule) استفاده شده‌است. مطابق نمودارها دیده می‌شود که SGD می‌تواند نقاط مینیمم را پیدا کند اما نسبت به بهینه‌سازهای تطبیقی بسیار آهسته‌تر عمل می‌کند. همچنین SGD به مقداردهی اولیه robust و چگونگی کم کردن نرخ یادگیری در طول بهینه‌سازی بسیار وابسته است و بیشتر ممکن است در نقاط زینی نسبت به نقاط مینیمم محلی گیر کند.

در دیتاست dogs-vs-cats مشاهده می‌شود که Adam کمترین خطای آموزش را دارد اما کمترین خطای اعتبارسنجی را ندارد. SGD به همراه Nesterov هنوز underfit هستند و سرعت همگرایی آن آهسته تر است. در این مدل، SGD به همراه Nesterov از سایر بهینه‌سازها عملکرد بهتری داشته چون این مدل کوچک است و تنها ۳ لایه مخفی دارد. اما ممکن است در مدل‌های پیچیده SGD عملکرد ضعیف‌تری نسبت به بهینه‌سازهای تطبیقی داشته باشد.

نتیجه‌گیری: اگر داده‌های ورودی پراکنده هستند و می‌خواهیم سریعتر به همگرایی برسیم، یا اگر یک شبکه عصبی عمیق و پیچیده داریم، توصیه می‌شود از بهینه‌سازهای تطبیقی استفاده شود. در صورت استفاده از SGD، generalization شبکه بهتر خواهد بود اما عملکرد آهسته تری دارد و باید در epoch های بیشتری آموزش ببیند. در این مقاله توصیه شده که ابتدا آموزش را با بهینه‌ساز Adam آغاز کنیم و وقتی به یک triggering condition رسیدیم از SGD استفاده کنیم. این کار باعث می‌شود از مزیت هر دو بهینه‌ساز یعنی سرعت Adam و تعمیم SGD با هم استفاده کنیم.

2)



$(i_1, i_2) = (3, 5)$ $\quad y = 1$

random initialize:

w_1	w_2	w_3	w_4	w_5	w_6
0.1	0.2	0.3	0.4	0.5	0.6

learning rate = 0.1

epoch 1 forward: $h_1 = w_1 i_1 + w_2 i_2 = 0.3 + 0.5 = 0.8$

$h_2 = w_3 i_1 + w_4 i_2 = 0.9 + 2 = 2.9$

$z = w_5 h_1 + w_6 h_2 = \underbrace{0.5 \times 0.8}_{0.4} + \underbrace{0.6 \times 2.9}_{1.74} = 2.14$

$o = \sigma(z) = \frac{1}{1+e^{-2.14}} \simeq 0.89$

loss: $MSE = \sum_{i=1}^{1} (y_i - o_i)^2 = \underbrace{(1 - 0.89)^2}_{0.11} = 0.0121$

backward: $\partial l / \partial w_5 = \partial l / \partial a \times \partial a / \partial z \times \partial z / \partial w_5 = 2(y-a)(-1)\, a(1-a)\, h_1$

chain rule استفاده از

$= -2\underbrace{(1-0.89)}_{0.11} \times \underbrace{0.89(1-0.89)}_{0.11} \times 0.8 \simeq \boxed{-0.017}$

$\partial l / \partial w_6 = -2(y-a)\, a(1-a)\, h_2 = -2(1-0.89)\, 0.89(1-0.89)\, 2.9 \simeq \boxed{-0.062}$

$\partial l / \partial w_1 = \partial l / \partial a \; \partial a / \partial z \; \partial z / \partial h_1 \; \partial h_1 / \partial w_1 = -2(y-a)\, a(1-a)\, w_5\, i_1 =$

$= -2(1-0.89)\, 0.89(1-0.89)\, 0.5 \times 3 \simeq \boxed{-0.032}$

$\partial l / \partial w_2 = \partial l / \partial a \; \partial a / \partial z \; \partial z / \partial h_1 \; \partial h_2 / \partial w_2 = -2(y-a)\, a(1-a)\, w_5\, i_2 =$

$= -2(1-0.89)\, 0.89(1-0.89)\, 0.5 \times 5 \simeq \boxed{-0.053}$

$$\partial l/\partial w_3 = \partial l/\partial a \; \partial a/\partial z_2 \; \partial z_2/\partial h_2 \; \partial h_2/\partial w_3 = -2(y-a)\,a(1-a)\,w_6\,i_1 =$$

$$= -2(1-0.89)\,0.89(1-0.89)\,0.6 \times 3 \simeq \boxed{-0.038}$$

$$\partial l/\partial w_4 = \partial l/\partial a \; \partial a/\partial z_2 \; \partial z_2/\partial h_2 \; \partial h_2/\partial w_4 = -2(y-a)\,a(1-a)\,w_6\,i_2 =$$

$$= -2(1-0.89)\,0.89(1-0.89)\,0.6 \times 5 \simeq \boxed{-0.064}$$

update : $w_1 = w_1 - \alpha \dfrac{\partial l}{\partial w_1} = 0.1 - 0.1 \times (-0.032) = \boxed{0.1032}$

$w_2 = w_2 - \alpha \, \partial l/\partial w_2 = 0.2 - 0.1 \times (-0.053) = \boxed{0.2053}$

$w_3 = w_3 - \alpha \dfrac{\partial l}{\partial w_3} = 0.3 - 0.1(-0.038) = \boxed{0.3038}$

$w_4 = w_4 - \alpha \dfrac{\partial l}{\partial w_4} = 0.4 - 0.1(-0.064) = \boxed{0.4064}$

$w_5 = w_5 - \alpha \dfrac{\partial l}{\partial w_5} = 0.5 - 0.1(-0.017) = \boxed{0.5017}$

$w_6 = w_6 - \alpha \dfrac{\partial l}{\partial w_6} = 0.6 - 0.1(-0.062) = \boxed{0.6062}$

epoch 2 forward: $h_1 = 0.1032 \times 3 + 0.2053 \times 5 \approx 1.336$

$h_2 = 0.3038 \times 3 + 0.4064 \times 5 \approx 2.943$

$z = 0.5017 \times 1.336 + 0.6062 \times 2.943 \approx 2.454$

$O = \sigma(z) \approx 0.92$

$loss = (1-0.92)^2 = 0.0064$

backward: $\partial l / \partial w_5 = -2\underbrace{(1-0.92)}_{0.08} 0.92(1-0.92) 1.336 \approx \boxed{-0.015}$

$\partial l / \partial w_6 = -2(1-0.92) 0.92(1-0.92) 2.943 \approx \boxed{-0.034}$

$\partial l / \partial w_1 = -2(1-0.92) 0.92(1-0.92) 0.5017 \times 3 \approx \boxed{-0.017}$

$\partial l / \partial w_2 = -2(1-0.92) 0.92(1-0.92) 0.5017 \times 5 \approx \boxed{-0.029}$

$\partial l / \partial w_3 = -2(1-0.92) 0.92(1-0.92) 0.6062 \times 3 \approx \boxed{-0.021}$

$\partial l / \partial w_4 = -2(1-0.92) 0.92(1-0.92) 0.6062 \times 5 \approx \boxed{-0.035}$

update: $w_1 = 0.1032 - 0.1 \times (-0.017) = \boxed{0.1049}$

$w_2 = 0.2053 - 0.1 \times (-0.029) = \boxed{0.2082}$

$w_3 = 0.3038 - 0.1(-0.021) = \boxed{0.3059}$

$w_4 = 0.4064 - 0.1(-0.035) = \boxed{0.4099}$

$w_5 = 0.5017 - 0.1(-0.015) = \boxed{0.5032}$

$w_6 = 0.6062 - 0.1(-0.034) = \boxed{0.6096}$

با توجه به اینکه شبکه تنها یک داده ورودی دارد، بعد از آموزش دیدن شبکه، دقت آن روی همین داده برابر با ۱۰۰ است. در این شبکه لایه‌های میانی تابع فعالسازی ندارند. به دلیل استفاده از MSE به عنوان تابع ضرر، مقدار loss همواره کوچک است. مقدار گرادیان‌های محاسبه شده نیز کوچک هستند و باعث می‌شود پارامترهای شبکه خیلی کم آپدیت شوند. در اینجا شبکه هنوز همگرا نشده است چون اگر آموزش را ادامه دهیم همچنان شبکه به یادگیری ادامه می‌دهد و مقدار پارامترها تغییر می‌کند.

3. در سلول اول کتابخانه‌های مورد نیاز را import می‌کنیم.

در سلول دوم، باید دیتاست CIFAR10 را دانلود کنیم. این دیتاست شامل ۶۰هزار تصویر با ابعاد ۳۲ * ۳۲ است که در ۱۰ کلاس طبقه‌بندی می‌شود. کلاس‌ها عبارتند از:

airplanes, cars, birds, cats, deer, dogs, frogs, horses, ships, and trucks

از کتابخانه torchvision دیتاست را دانلود می‌کنیم. برای داده آموزشی پارامتر train را True و برای داده آزمون آن را False قرار می‌دهیم (تا شبکه بر روی داده‌های آزمون، آموزش نبیند). اندازه batch را برابر ۴ قرار می‌دهیم.

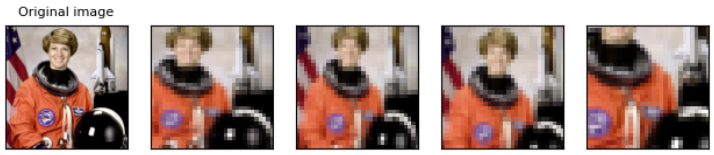
```
batch_size = 4
train_set = torchvision.datasets.CIFAR10(root='./data', train=True,
                                         download=True, transform=transform)
train_loader = torch.utils.data.DataLoader(train_set, batch_size=batch_size,
                                           shuffle=True, num_workers=2)

test_set = torchvision.datasets.CIFAR10(root='./data', train=False,
                                        download=True, transform=transform_test)
test_loader = torch.utils.data.DataLoader(test_set, batch_size=batch_size,
                                          shuffle=False, num_workers=2)
```

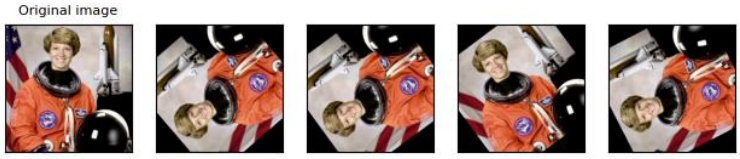
از data augmentation نیز استفاده می‌کنیم. با این کار از هر نمونه داده چند نمونه جدید تولید می‌کنیم. افزایش تعداد داده‌های آموزشی باعث بهبود عملکرد و افزایش robustness شبکه می‌شود. در بخش torchvision.transforms انواع روش‌های augmentation وجود دارد که در این سوال از موارد زیر استفاده شده:

- `RandomHorizontalFlip()`: تصویر را با احتمال p به صورت افقی فلیپ می‌کنیم.

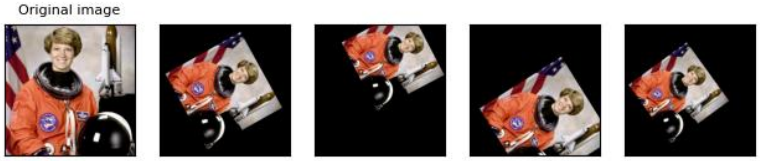
- RandomResizedCrop(): قسمتی از تصویر را به صورت تصادفی کراپ می‌کند و به اندازه داده شده resize می‌کند.



- RandomRotation(): تصویر را با زاویه تصادفی (در بازه داده شده) می‌چرخاند.



- RandomAffine(): بر روی تصویر random affine را پیاده‌سازی می‌کند.



- RandomInvert(): به صورت تصادفی رنگ‌های تصویر را وارونه می‌کند.

سپس تصاویر را به تنسور تبدیل کرده و تنسورها را نرمالایز می‌کنیم (تا پراکندگی داده‌ها زیاد نباشد).

```
transform = transforms.Compose([transforms.RandomHorizontalFlip(),                 # Horizontally flip the given image randomly with a given probability
                                transforms.RandomResizedCrop((32, 32)),            # Crop a random portion of image and resize it to a given size
                                transforms.RandomRotation(15),                     # Rotate the image by angle
                                transforms.RandomAffine(0, shear=10, scale=(0.8,1.2)), # Random affine transformation of the image keeping center invariant
                                transforms.RandomInvert(p=0.3),                    # Inverts the colors of the given image randomly with a given probability
                                transforms.ToTensor(),
                                transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)) # Normalize
                               ])
```

Transform ساخته شده را به هنگام دانلود داده‌ها به تابع مربوطه پاس می‌دهیم.

برای داده‌های آزمون نباید از augmentation استفاده کنیم. پس تنها آنها را به تنسور تبدیل کرده و تنسورها را نرمالایز می‌کنیم.

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

در سلول سوم، دسترسی به GPU را بررسی می‌کنیم. در صورت وجود GPU از آن برای سرعت بخشیدن به انجام محاسبات استفاده می‌کنیم.

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

```
using cuda
```

در سلول چهارم،شبکه عصبی را با ارث‌بری از nn.Module می‌سازیم. در ورودی constructor این کلاس، مقادیر مورد نیاز شبکه از جمله تعداد نورون‌های هیدن، اندازه کرنل، اندازه padding و ... را داریم. این شبکه از دو قسمت کلی تشکیل شده:

اول) لایه کانولوشنی:

```
self.conv_layer = nn.Sequential(
    nn.Conv2d(n_in, n_hidden1, kernel_size_cnn, padding=padding), #(in_channels, out_channels, kernel_size)
    nn.BatchNorm2d(n_hidden1), #(number of features)
    nn.ReLU(),
    nn.Conv2d(n_hidden1, n_hidden2, kernel_size_cnn, padding=padding),
    nn.ReLU(),
    nn.MaxPool2d(kernel_size_pool),

    nn.Conv2d(n_hidden2, n_hidden3, kernel_size_cnn, padding=padding),
    nn.BatchNorm2d(n_hidden3),
    nn.ReLU(),
    nn.Conv2d(n_hidden3, n_hidden4, kernel_size_cnn, padding=padding),
    nn.ReLU(),
    nn.MaxPool2d(kernel_size_pool),
    nn.Dropout(p),

    nn.Conv2d(n_hidden4, n_hidden5, kernel_size_cnn, padding=padding),
    nn.BatchNorm2d(n_hidden5),
    nn.ReLU(),
    nn.Conv2d(n_hidden5, n_hidden6, kernel_size_cnn, padding=padding),
    nn.ReLU(),
    nn.MaxPool2d(kernel_size_pool), # output shape = (128, 4, 4)
)
```

دوم) لایه خطی:

```
self.fc_layer = nn.Sequential(
    nn.Dropout(p),
    nn.Flatten(),
    nn.Linear(n_hidden6*4*4, n_l1),
    nn.ReLU(),
    nn.Linear(n_l1, n_l2),
    nn.ReLU(),
    nn.Dropout(p),
    nn.Linear(n_l2, n_output)
)
```

با توجه به عمیق بودن شبکه و کوچک بودن اندازه تصویر، در لایه‌های کانولوشنی به ورودی padding اضافه می‌کنیم (در صورت اضافه نکردن اندازه خروجی صفر می‌شود).

تعداد کانال‌های ورودی یک لایه کانولوشنی، همان تعداد کانال‌های خروجی لایه کانولوشنی قبلی است.

از لایه Dropout به عنوان یک روش Regularization استفاده می‌کنیم تا شبکه overfit نشود و در هر مرحله از آموزش تعدادی از نورون‌ها را با احتمال p حذف کند.

خروجی لایه آخر Linear باید ۱۰ نورون داشته باشد (به اندازه تعداد کلاس‌ها).

در تابع فوروارد این شبکه، ورودی را از ابتدا از لایه کانولوشن و سپس از لایه خطی رد می‌کنیم و خروجی را برمی‌گردانیم.

سپس نمونه‌ای از کلاس Network می‌سازیم. آن را به device موجود منتقل می‌کنیم. خلاصه شبکه را چاپ می‌کنیم.

```
Network(
  (conv_layer): Sequential(
    (0): Conv2d(3, 4, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): BatchNorm2d(4, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU()
    (3): Conv2d(4, 8, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (4): ReLU()
    (5): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (6): Conv2d(8, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (7): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (8): ReLU()
    (9): Conv2d(16, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (10): ReLU()
    (11): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (12): Dropout(p=0.2, inplace=False)
    (13): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (14): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (15): ReLU()
    (16): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (17): ReLU()
    (18): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (fc_layer): Sequential(
    (0): Dropout(p=0.2, inplace=False)
    (1): Flatten(start_dim=1, end_dim=-1)
    (2): Linear(in_features=2048, out_features=1000, bias=True)
    (3): ReLU()
    (4): Linear(in_features=1000, out_features=100, bias=True)
    (5): ReLU()
    (6): Dropout(p=0.2, inplace=False)
    (7): Linear(in_features=100, out_features=10, bias=True)
  )
)
```

در سلول پنجم، از تابع ضرر Cross entropy استفاده کرده و آن را به device منتقل می‌کنیم.

همچنین از بهینه‌ساز Adam با نرخ یادگیری 0.001 استفاده می‌کنیم.

```
loss_fn = nn.CrossEntropyLoss().to(device)
optimizer = optim.Adam(model.parameters(), lr=0.001, weight_decay=0.005)
```

در سلول ششم ابتدا تابع train را تعریف می‌کنیم. در یک حلقه هر بار به اندازه batch که در اینجا ۴ است روی داده‌ها iterate می‌کنیم. ابتدا ورودی و لیبل آن را به device منتقل می‌کنیم. بعد تمامی گرادیان‌ها را برابر با صفر قرار می‌دهیم. سپس پیش‌بینی مدل از ورودی داده شده را می‌یابیم. بعد مقدار ضرر را محاسبه می‌کنیم. در پس‌انتشار مقدار گرادیان‌ها را حساب کرده و پارامترهای شبکه را آپدیت می‌کنیم. در صورتی که پیش‌بینی شبکه با لیبل داده‌شده برابر باشد، به متغیر correct یک عدد اضافه می‌کنیم.

```
def train(dataloader, model, loss_fn, optimizer):
  size = len(dataloader.dataset)
  correct = 0
  model.train()
  for batch, (x, y) in enumerate(dataloader):
    x, y = x.to(device), y.to(device)
    optimizer.zero_grad()

    pred = model(x)
    loss = loss_fn(pred, y)

    # Backpropagation
    loss.backward()
    optimizer.step()

    correct += (pred.argmax(1) == y).type(torch.float).sum().item()
    if batch % 500 == 0:
        loss, current = loss.item(), batch * batch_size
        print(f"loss: {loss:>7f}  [{current:>5d}/{size:>5d}]")

  print(f"total train accuracy: {correct*100/size:>7f}")
```

بعد از هر ۵۰۰ batch مقدار تابع ضرر را چاپ می‌کنیم. همچنین بعد از هر epoch مقدار دقت را بدست می‌آوریم.

```
Epoch 1
-------------------------------
loss: 2.275115  [    0/50000]
loss: 1.925573  [ 2000/50000]
loss: 1.852545  [ 4000/50000]
loss: 2.354488  [ 6000/50000]
loss: 1.852010  [ 8000/50000]
loss: 1.970184  [10000/50000]
loss: 1.995650  [12000/50000]
loss: 2.248348  [14000/50000]
loss: 2.021744  [16000/50000]
loss: 2.040428  [18000/50000]
loss: 2.993144  [20000/50000]
loss: 2.113527  [22000/50000]
loss: 2.301047  [24000/50000]
loss: 1.972690  [26000/50000]
loss: 2.044314  [28000/50000]
loss: 2.266176  [30000/50000]
loss: 2.218407  [32000/50000]
loss: 2.179538  [34000/50000]
loss: 1.844463  [36000/50000]
loss: 1.995457  [38000/50000]
loss: 1.576590  [40000/50000]
loss: 2.082208  [42000/50000]
loss: 1.965498  [44000/50000]
loss: 2.110747  [46000/50000]
loss: 2.166228  [48000/50000]
total train accuracy: 22.540000
Epoch 2
-------------------------------
loss: 2.030478  [    0/50000]
loss: 2.009756  [ 2000/50000]
loss: 1.941110  [ 4000/50000]
loss: 1.688848  [ 6000/50000]
loss: 2.260008  [ 8000/50000]
loss: 2.044127  [10000/50000]
loss: 2.182343  [12000/50000]
loss: 2.850788  [14000/50000]
loss: 2.150543  [16000/50000]
loss: 1.973233  [18000/50000]
loss: 2.249617  [20000/50000]
loss: 2.428895  [22000/50000]
loss: 2.457697  [24000/50000]
loss: 1.954258  [26000/50000]
loss: 1.852434  [28000/50000]
loss: 1.731977  [30000/50000]
loss: 2.197763  [32000/50000]
loss: 2.090644  [34000/50000]
loss: 2.131619  [36000/50000]
loss: 1.742185  [38000/50000]
loss: 2.040437  [40000/50000]
loss: 2.114441  [42000/50000]
loss: 2.561153  [44000/50000]
loss: 1.843608  [46000/50000]
loss: 2.004793  [48000/50000]
total train accuracy: 22.818000
Finished Training
```

در تابع test، ابتدا شبکه را روی حالت eval قرار می‌دهیم تا از لایه‌هایی مثل dropout استفاده نکند. دو دیکشنری با کلید نام لیبل‌ها ایجاد می‌کنیم. در یکی تعداد پیش‌بینی‌های درست شبکه از هر کلاس و در دیگری تعداد کل نمونه‌های متعلق به آن کلاس را نگه‌داری می‌کنیم. بعد از torch.no_grad() استفاده می‌کنیم تا پارامترهای شبکه را آپدیت نکند. روی داده‌های آزمون iterate می‌کنیم. X را به شبکه می‌دهیم. پیش‌بینی شبکه از آن داده، ایندکس خانه‌ای است که بیشترین احتمال را دارد پس از تابع armgax استفاده می‌کنیم. در صورت درست بودن پیش‌بینی شبکه یک عدد به دیکشنری با کلید آن لیبل اضافه می‌کنیم.

```
23 def test(dataloader, model, loss_fn, optimizer):
24   correct_pred = {c: 0 for c in classes}
25   total = {c: 0 for c in classes}
26
27   model.eval()
28   with torch.no_grad():
29     for data in dataloader:
30       x, y = data
31       x, y = x.to(device), y.to(device)
32       out = model(x)
33       pred = out.argmax(1)
34       for y, pred in zip(y, pred):
35         if y == pred:
36           correct_pred[classes[y]] += 1
37         total[classes[y]] += 1
38
39   # print results
40   for c in classes:
41     acc = float(correct_pred[c]) / total[c]
42     print(f"Accuracy of {c}: {acc*100}%")
43
```

در پایان دقت شبکه روی هر کلاس از داده‌های آزمون را چاپ می‌کنیم.

```
Accuracy of plane: 12.0%
Accuracy of car: 70.5%
Accuracy of bird: 3.5000000000000004%
Accuracy of cat: 6.6000000000000005%
Accuracy of deer: 42.8%
Accuracy of dog: 54.7%
Accuracy of frog: 16.5%
Accuracy of horse: 49.0%
Accuracy of ship: 58.5%
Accuracy of truck: 9.6%
```

از تابع train که قبل تعریف کرده‌ایم استفاده می‌کنیم و در ۲ epoch شبکه را آموزش می‌دهیم. سپس تابع test را صدا می‌زنیم تا دقت پیش‌بینی شبکه روی داده‌های آزمون را بیابیم (نتایج آن در تصاویر بالا دیده می‌شود).

```python
for epoch in range(2):
  print(f"Epoch {epoch+1}\n-------------------------------")
  train(train_loader, model, loss_fn, optimizer)
print('Finished Training')

test(test_loader, model, loss_fn, optimizer)
```

نتیجه‌گیری: دقت شبکه روی داده‌های آموزشی بعد از ۲ epoch ۲۲ درصد است که بسیار کم است. یعنی شبکه در ۲ لوپ، کمی بهتر از پیش‌بینی تصادفی عمل کرده است. پس برای آموزش شبکه باید تعداد epochها را زیاد کنیم. همچنین به دلیل کم بودن batch size آموزش بسیار طول می‌کشد (batch size ۴ تقریبا شبیه sgd است). دیده می‌شود که دقت مدل روی بعضی از کلاس‌ها با سایر کلاس‌ها تفاوت چشمگیری دارد (مثلا کلاس ماشین در مقایسه با پرنده). شبکه نتوانسته به خوبی generalize کند و بعضی از کلاس‌ها را خیلی بیشتر آموزش دیده‌است.

با زیاد کردن اندازه batch و همچنین تعداد epochها شبکه بهتر آموزش می‌بیند و سرعت یادگیری نیز بیشتر خواهد شد.

منابع استفاده‌شده:

https://medium.com/geekculture/a-2021-guide-to-improving-cnns-optimizers-adam-vs-sgd-495848ac6008

https://pytorch.org/vision/stable/auto_examples/plot_transforms.html#sphx-glr-auto-examples-plot-transforms-py