تمرین دوم یادگیری عمیق - ملیکا محمدی فخار - ۹۹۵۲۲۰۸۶

۱.

الف.

مشکل کم‌برازش یا underfitting ناشی از این است که مدل بسیار ساده است و به این دلیل هنوز بر روی داده‌های آموزشی نیز fit نشده است و آن‌ها را به خوبی یاد نگرفته است. ممکن است علت این عدم توانایی یادگیری، تعداد کم پارامترهای قابل یادگیری، تعداد کم epochها باشد. بنابراین نه بر روی داده‌های آموزشی و نه بر روی داده‌های تست عملکرد مطلوبی ندارد. برای حل این مشکل باید مدل را پیچیده‌تر کرد تا توانایی یادگیری آن افزایش یابد، برای مثال تعداد لایه‌های میانی و یا تعداد نورون‌های آنان را افزایش دهیم، تعداد epochها را افزایش دهیم تا مدل برای iterationهای بیشتری به یادگیری بپردازد.

مشکل بیش‌برازش یا overfitting ناشی از این است که مدل با توان یادگیری بالا بر روی داده‌های آموزشی تا حد خیلی خوبی fit شده است تا آن‌جا که نویزها و جزئیات غیرضروری داده‌های آموزشی را نیز یاد گرفته است. بنابراین علی‌رغم عملکرد مناسب بر روی داده‌های آموزشی، بر روی داده‌های تست به خوبی عمل نمی‌کند(نسبت به عملکردش روی داده‌های آموزشی). تعداد بالای پارامترهای قابل یادگیری، کوچک بودن دیتاست، عدم استفاده از regularization می‌تواند باعث بروز این مشکل شود. برای حل آن بایست مدل را محدود کرد تا فقط ویژگی‌های اصلی و ضروری را یاد بگیرد. برای مثال از regularization, dropout, augmentation و batch normalization استفاده کنیم.

ب.

عملکرد مدلی که دچار بیش‌برازش است بر روی داده‌های تست و آموزشی تفاوت قابل توجهی دارد، به این صورت که روی داده‌های تست خطای بسیار بیشتری دارد. همچنین اگر به نمودار دقت توجه کنیم دقت در طول زمان بر روی داده‌های آموزشی بسیار بالا می‌رود درحالی‌که بر روی داده‌های تست، روند کاهشی می‌گیرد. علاوه بر این مدل دچار بیش‌برازش از آن‌جا که جزئیات را هم یادگرفته و نسبت به نویز حساس است، واریانس بالایی دارد.

پ.

از آن جا که در dropout mask تعداد برابری ۰ و ۱ وجود دارد، $p = 0.5$ می‌باشد.

train:

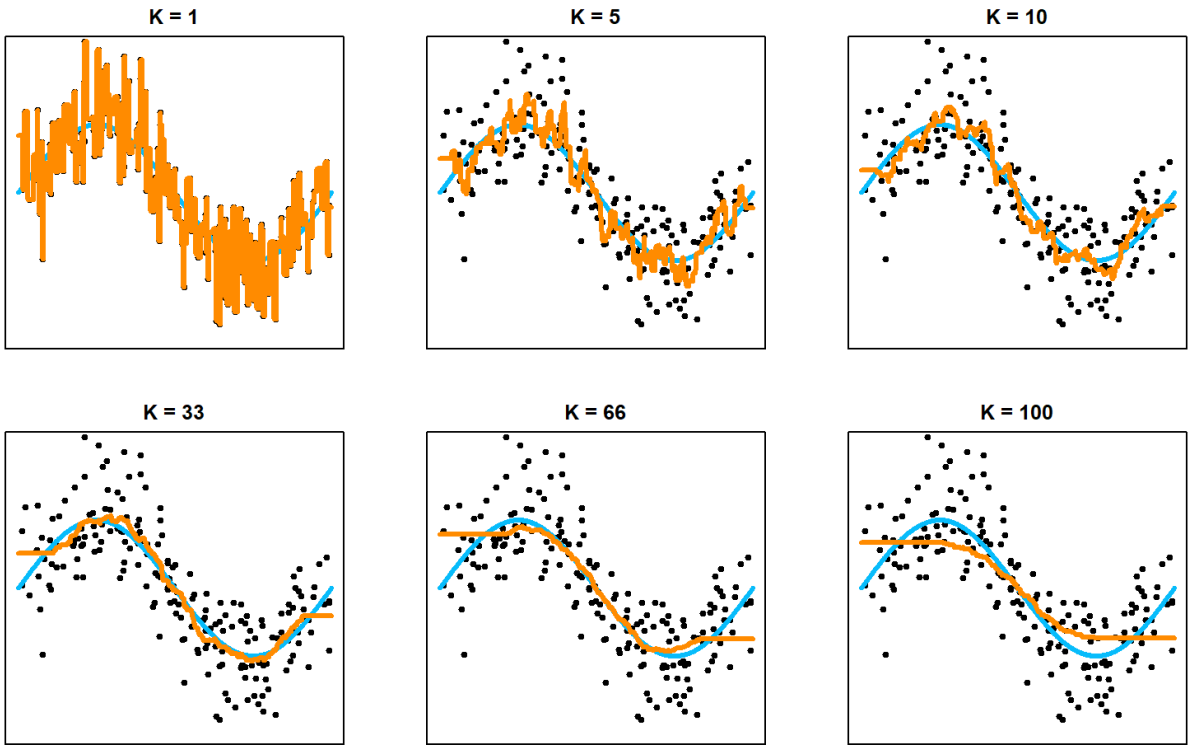
1.6	0	0	1.9
0	2.5	2.5	0
0	3.2	3.7	0
1.3	0	0	1.2

test:

0.8	-0.35	-0.1	0.95
-1.15	1.25	1.25	-0.45
-0.25	1.6	1.85	-0.2
0.65	-0.2	-1.3	0.6

۲. منبع: https://teazrq.github.io/stat542/rlab/knn.html

الف. می‌دانیم k تعداد نزدیک‌ترین همسایگانیست که با توجه به آن‌ها class داده‌ی جدید را مشخص می‌کنیم. اگر k افزایش یابد درواقع ما برای classify کردن داده جدید، به تعداد بیشتری از داده‌های نزدیک به داده جدید توجه می‌کنیم. در این صورت **واریانس کاهش می‌یابد** زیرا با توجه به تعداد بالای داده‌های نزدیک، حساسیت مدل به نویز کاهش می‌یابد (اگر کلاس تعداد کمی از داده‌های نزدیک را عوض کنیم، از آن‌جا که تعداد داده‌های نزدیک زیاد است، نتیجه برای داده جدید احتمالا تغییر نخواهد کرد.) نتیجه کاهش حساسیت به نویز این است که مدل کمتر قادر به یافتن الگوهای پیچیده در داده‌ها و بیشتر کلی‌نگر خواهد بود. به همین دلیل **بایاس مدل افزایش می‌یابد**.



ب.

- استفاده از منظم‌سازی ممکن است باعث تضعیف عملکرد مدل شود: این جمله صحیح است. زیرا استفاده از منظم‌سازی به منظور پرهیز از overfit شدن مدل صورت می‌پذیرد. اما منظم‌سازی بیش از حد می‌تواند مدل را بیش از حد ساده کند و باعث از دست رفتن الگوها و اطلاعات مهمی از داده‌ها شود و به این ترتیب عملکرد مدل را بدتر کند.

- اضافه کردن تعداد زیاد ویژگی‌های جدید، باعث جلوگیری از بیش‌برازش می‌شود: این جمله غلط است. زیرا بیش‌برازش از آن‌جا ناشی می‌شود که مدل جزئی‌ترین ویژگی‌های غیرضروری و نویزهای داده‌ها را هم یاد می‌گیرد و درواقع داده‌های آموزشی را حفظ می‌کند. حال اگر ما یکسری ویژگی با تعداد بالا وضعیت را بدتر می‌کنیم. البته بستگی نیز دارد که چه ویژگی‌هایی را اضافه می‌کنیم.

- با زیاد کردن ضریب منظم‌سازی، احتمال بیش‌برازش بیشتر می‌شود. این جمله غلط است زیرا ضریب منظم‌سازی میزان جریمه‌سازی وزن‌های بزرگ را مشخص می‌کند و درواقع مدل را محدود می‌کند که این از بیش‌برازش مدل جلوگیری می‌کند. حتی اگر این ضریب خیلی بزرگ باشد ممکن است باعث underfit شدن مدل شود.

پ.

- Wexp1

از آن جا که مقدار وزن‌ها کوچک و همگی نزدیک به هم هستند احتمالا از L2 استفاده شده است.

- Wexp2

از آن جا که همه وزن‌ها جز یکی ۰ هستند، به نظر می‌رسد از منظم‌سازی‌ای استفاده شده که تمایل به sparsity و پراکندگی دارد. این ویژگی مربوط به L1 می‌باشد.

- Wexp3

وزن‌های این مورد هیچ‌یک صفر نیستند و همچنین همگی بزرگ هستند و مقدار نزدیک به همی ندارند. بنابراین از هیچ منظم‌سازی‌ای استفاده نشده و مقادیر داده شده مربوط به وزن‌های یک مدل overfit هستند.

- Wexp4

مقادیر به یکدیگر نزدیک نیستند اما مقدار ۰ در بینشان وجود دارد بنابراین پراکندگی و sparsity دیده می‌شود که مربوط به L1 می‌باشد.

۳.

الف.

در یادگیری ماشین، تقطیر دانش به فرایندی گفته می‌شود که دانش از یک مدل بزرگ‌تر و پیچیده‌تر به یک مدل کوچک‌تر و ساده‌تر منتقل می‌شود. این تکنیک به ما این قابلیت را می‌دهد تا مدل خود را کوچک‌تر نگه داریم (زیرا از یک مدل pre-trained بزرگ‌تر کمک می‌گیرد) و از برای توسعه مدل‌های سریع با دقت مناسب استفاده می‌شود.

ب.

teacher model یک مدل pre-trained است که نسبت به student model بزرگ‌تر است و پارامترهای بیشتری دارد. به ازای هر ورودی، خروجی teacher model یک soft target است که در واقع یک توزیع احتمال می‌باشد. در فرایند تقطیر دانش یک تابع ضرر داریم که با توجه به اختلاف خروجی مدل معلم (soft target) و خروجی مدل دانش‌آموز(T=t) و همچنین استفاده از hard labelها، مدل دانش‌آموز را جریمه می‌کند تا با مینیمم کردن مقدار این ضرر بتواندعملکرد بهتری داشته باشد. در واقع هدف این فرایند این است که یک مدل کوچک‌تر با تعداد پارامترهای کمتر بتواند تسک یک مدل بزرگ‌تر و پیچیده‌تر را انجام دهد.

در شکل یک متغیر T دیده می‌شود که در واقع Temprature parameter است که به عنوان ورودی به softmax داده شده است. این مقدار مشخص می‌کند خروجی مدل که توزیعی احتمالی است تا چه اندازه قطعیت داشته باشد. هرچه مقدار T بیشتر باشد خروجی softتر است به این معنی که قطعیت کمتری دارد. همانطور که در شکل مشخص است به ازای T=t خروجی soft label و به ازای T=1 خروجی hard label تولید شده است.

دو نوع ضرر نیز محاسبه شده است، که distillation loss از مقایسه teacher soft labels و student soft predictions به دست آمده است و student loss از مقایسه student hard predictions و hard labels y.

پ.

همان‌طور که در بخش ب توضیح داده شد، دو نوع loss محاسبه می‌شود و به هریک ضریبی اختصاص داده می‌شود تا با ترکیبی از آن‌ها وزن‌های مدل دانش‌آموز به روز رسانی می‌شوند. ضریب لاندا یک hyperparameter است.

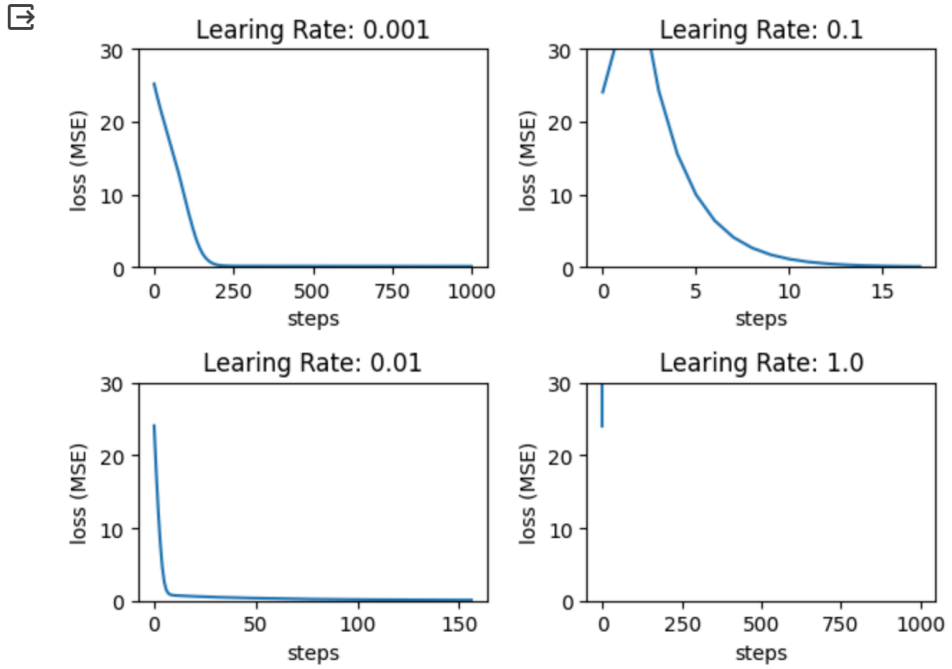
$$L = \lambda \ . \ distillation\_Loss + (1 - \lambda) \ . \ student\_Loss$$

۴. به طور کلی در اغلب lrها تابع momentum سریع‌تر مدل را به همگرایی می‌رساند که دلیل آن هم در نظر گیری سرعت طی حرکت است. اگر در یک مقطع میزان ضرر همواره کاهشی باشد به سرعت momentum افزوده می‌شود و گام‌های بلندتری برمی‌دارد. اما در SGD سرعتی تعریف نمی‌شود و ما در یک شیب ثابت همواره روند ثابت داریم. حال به تفکیک آنان را بررسی می‌کنیم:

مقایسه lrهای متفاوت (SGD): همان‌طور که از نمودارها مشخص است اگر lr بسیار کم (0.001) باشد روند همگرایی مدل کند خواهد بود و پس از طی حدود ۲۰۰ گام رخ داده است. به ازای مقدار 0.01 نیز می‌توان گفت روند همگرایی همچنان کند است زیرا پس از طی حدود ۱۲۰ گام اتفاق افتاده است.
به نظر می‌رسد برای SGD میزان lr = 0.1 نسبت به ۳ مقدار دیگر مناسب‌تر است زیرا در گام‌های کمی (حدود ۱۵) همگرا شده است. روند صعودی که تابع ضرر در گام‌های اولیه طی کرده است به دلیل این است که چون lr خیلی کوچک نیست ممکن است از روی نقطه بهینه عبور کند و با یک overshoot نه چندان شدید دوباره به سمت آن برگردد.
اگر مقدار lr خیلی زیاد باشد (۱)، ممکن است دچار overshoot بسیار زیاد شویم شویم تا جایی که تابع ضرر دچار واگرایی شود زیرا از نقطه بهینه رد می‌شود و پس از آن دچار gradient explosion می‌شود و همواره از نقطه بهینه دورتر می‌شود.

```
[ ]  def SGD(a, lr, _, __):
         a.data -= a.grad * lr
         a.grad = None
```

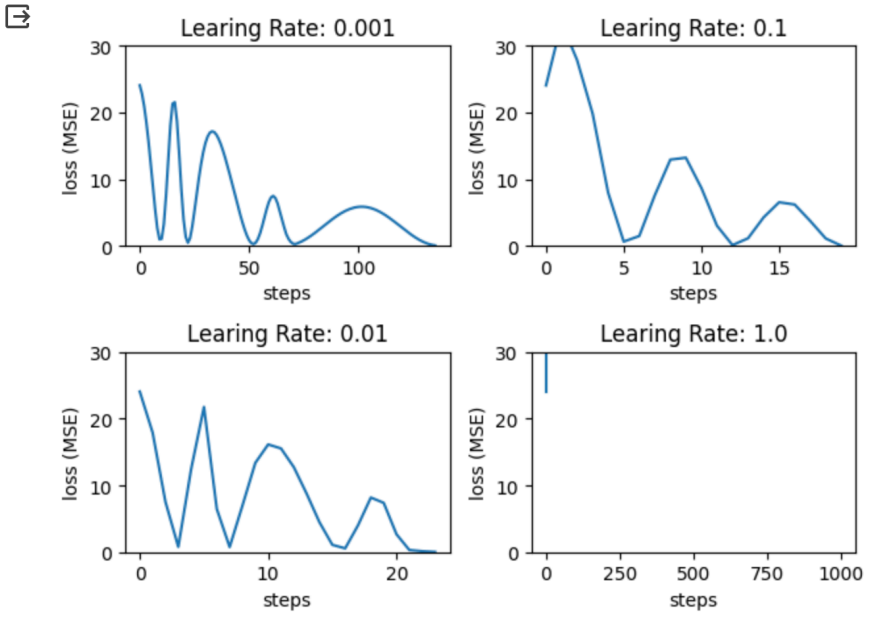
```
#train model with SGD
my_model.train(SGD)
```



مقایسه lrهای متفاوت (momentum): همان‌طور که از نمودارها مشخص است اگر lr بسیار کم (0.001) باشد روند همگرایی مدل کند خواهد بود و پس از طی حدود ۱۴۰ گام رخ داده است.
به ازای مقدار 0.01 روند همگرایی سریع‌تر شده است زیرا پس از طی حدود ۲۲ گام اتفاق افتاده است.
به نظر می‌رسد برای momentum میزان lr = 0.1 نسبت به ۳ مقدار دیگر مناسب‌تر است زیرا در گام‌های کمی (حدود 18) همگرا شده است.
روند صعودی که تابع ضرر momentum طی کرده است به دلیل این است که چون ما سرعت پیشینی را لحاظ می‌کنیم و در یک مسیر به سمت مینیمم همواره شتاب می‌گیریم ممکن است از روی نقطه بهینه عبور کنیم و با یک overshoot نه چندان شدید دوباره به سمت آن برگردیم. و یا اگر به یک local minimum برسیم با استفاده از این سرعت داخل آن متوقف نمی‌شویم و عبور می‌کنیم.
اگر مقدار lr خیلی زیاد باشد (۱)، ممکن است دچار overshoot بسیار زیاد شویم شویم تا جایی که تابع ضرر دچار واگرایی شود زیرا از نقطه بهینه رد می‌شود و پس از آن دچار gradient explosion می‌شود و همواره از نقطه بهینه دورتر می‌شود.

```
[13] def momentum(a, lr, moms, __):
         previous_momentum = moms[-1]

         mom = a.grad * lr + previous_momentum * (1 - lr)
         moms.append(mom)
         a.data -= mom
         a.grad = None
```

```
#train model with momentum
my_model.train(momentum)
```



۵. ابتدا به توضیح کدهای اضافه شده در بخش الف می‌پردازیم:
در این بخش مدل تعریف شده است. مدل از نوع sequntial است. ابتدا یک لایه flatten قرار دارد تا ورودی را به به یک تنسور یک بعدی تبدیل کند. سپس ورودی‌ها به یک لایه linear که fully connected است داده می‌شود و خروجی هر لایه با عبور از تابع فعال‌سازی ReLU به لایه بعد می‌رود.

```
## Define the model
################ Your code ############
model = nn.Sequential(
    nn.Flatten(),
    nn.Linear(input_size, 256),
    nn.ReLU(),
    nn.Linear(256, 128),
    nn.ReLU(),
    nn.Linear(128, out_size)
)
#######################################
```

در کد زیر نوع تابع ضرر (CrossEntropyLoss)، تابع بهینه‌ساز(SGD) و نرخ یادگیری(lr) تعیین شده است:

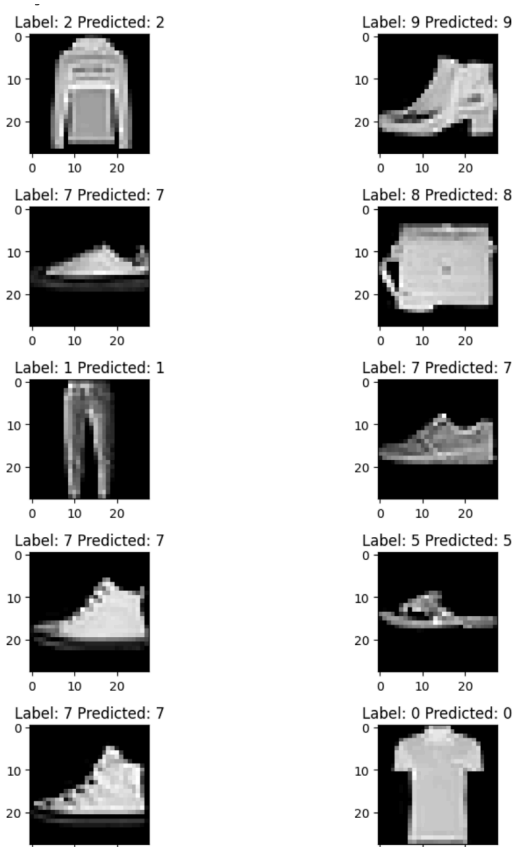
```
################ Your code ############
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=0.1)
#######################################
```

در این بخش تصاویر ورودی دیتاست به مدل داده می‌شود و خروجی (شامل پیش‌بینی‌های مدل) در output ریخته می‌شوند. در هر مرحله پیش‌بینی‌های مدل و لیبل‌ها به تابع ضرر داده می‌شوند تا مقدار ضرر محاسبه شود.

```
# forward pass
################ Your code ############
output = model(images)
loss = criterion(output,labels)
#######################################
```

از آن‌جا که ماژول d2l با ورژن سایر پکیج‌ها همخوان نمی‌شد با هماهنگی دستیار آموزشی به صورت دستی یک تابع پیاده سازی کردم تا ۱۰ تا از تصاویر دیتاست به همراه لیبل درست و لیبل پیش‌بینی‌شده را نمایش دهم.
در بدنه این تابع ابتدا batch اول تصاویر به همراه برچسب آنان گرفته شده سپس تصاویر به مدل داده شده و پیش‌بینی‌های آن ذخیره شده است. سپس ۱۰ تا از این داده‌ها به کمک کتابخانه matplotlib نمایش داده می‌شوند که خروجی نیز آورده شده است:

```python
def show_10_from_test_dataset(model, testloader):
  """Shows 10 images from the test dataset and their labels and predictions.

  Args:
    model: A PyTorch model.
    testloader: A PyTorch DataLoader for the test dataset.
  """

  # Get the first 10 images from the testloader
  images, labels = next(iter(testloader))

  # Make predictions on the images
  predictions = model(images)

  # Get the predicted labels
  _, predicted_labels = torch.max(predictions.data, 1)

  # Create a figure
  fig = plt.figure(figsize=(10, 10))

  # Add a subplot for each image
  for i in range(10):
    ax = fig.add_subplot(5, 2, i + 1)

    # Show the image
    image = images[i].cpu().numpy().squeeze()
    ax.imshow(image, cmap='gray')

    # Set the title of the subplot to the label and predicted label
    ax.set_title(f'Label: {labels[i]} Predicted: {predicted_labels[i]}')

  # Tighten the layout of the figure
  fig.tight_layout()

  # Show the figure
  plt.show()
```



سپس یک تابع test تعریف شده است که در هر مرحله میزان ضرر متوسط (با محاسبه مجموع ضرر و تقسیم آن بر تعداد کل) و همچنین دقت مدل (با محاسبه تعداد پیش‌بینی‌های درست و تقسیم آن بر تعداد کل) را به دست آورده و برمی‌گرداند:

```python
def test(model, testloader):
  correct = 0
  total = 0
  total_loss = 0
  with torch.no_grad():
    for images, labels in testloader:
      images = images.view(images.shape[0], -1)
      output = model(images)
      loss = criterion(output, labels)
      total_loss += loss.item()
      _, predicted = torch.max(output.data, 1)
      total += labels.size(0)
      correct += (predicted == labels).sum().item()

  accuracy = 100 * correct / total
  average_loss = total_loss / len(testloader)

  return average_loss, accuracy
```

```
Accuracy: 85.42%
Average loss: 0.41
```

۶.
ب. در این بخش بایست تغییراتی ایجاد کنیم تا مدل overfit شود. به منظور این امر چندین تغییر ایجاد کرده‌ام. ابتدا تعداد لایه‌های مدل و تعداد نورون‌های هر لایه را افزایش داده‌ام، سپس lr را به ۰.۰۱ کاهش داده و تعداد epochها را از ۱۰ به ۵۰ رسانده‌ام تا مدل قادر به یادگیری الگوهای جزئی داده‌ها نیز باشد:

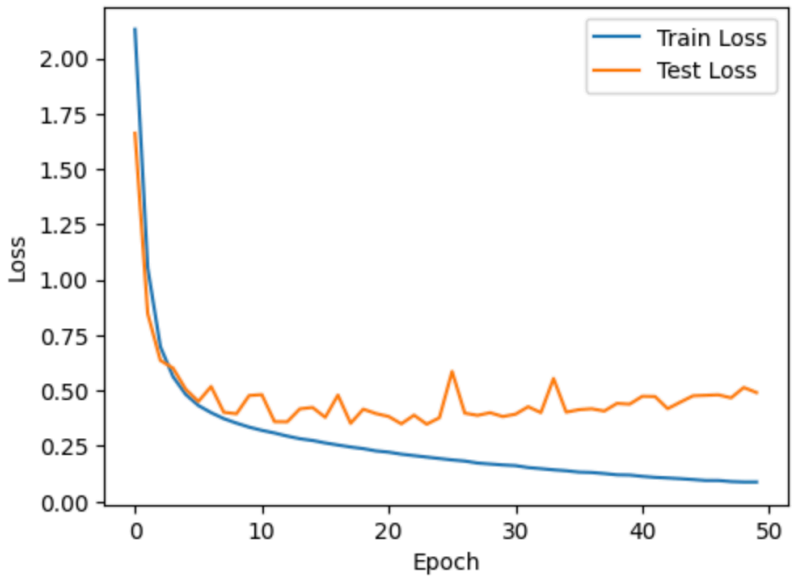
```
# Increase model capacity by incresing number of layers and units
overfit_model = nn.Sequential(
    nn.Flatten(),
    nn.Linear(input_size, 512),
    nn.ReLU(),
    nn.Linear(512, 256),
    nn.ReLU(),
    nn.Linear(256, 128),
    nn.ReLU(),
    nn.Linear(128, 64),
    nn.ReLU(),
    nn.Linear(64, 32),
    nn.ReLU(),
    nn.Linear(32, out_size)
)

overfit_optimizer = torch.optim.SGD(overfit_model.parameters(), lr=0.01)  # Decrease learning rate

# Train for more epochs
overfit_epochs = 50
```

حال مدل را آموزش می‌دهیم:
حلقه بیرونی بر روی epochها iteration انجام می‌دهد. سپس overfit_model.train مدل را به فاز آموزشی می‌برد. مقدار اولیه running loss را صفر قرار می‌دهیم و هر گاه ضرر را محاسبه کردیم با آن جمع می‌کنیم. images.view(images.shape[0], -1) تصاویر ورودی را به صورت یک tensor فلت درمی‌آورد، overfit_optimizer.zero_grad گرادیان‌های بهینه شده را پاک می‌کند، سپس تصاویر را به مدل ورودی می‌دهیم و پیش‌بینی‌های آن را در output ذخیره می‌کنیم. حالا میزان ضرر را طبق اختلاف برچسب و پیش‌بینی محاسبه می‌کنیم. با استفاده از loss.backward گرادیان‌ها را برمی‌گردانیم و با overfit_optimizer.step پارامترهای مدل را طبق گرادیان بهینه می‌کنیم. میزان loss را با running_loss جمع می‌کنیم تا در میانگین‌گیری از آن استفاده کنیم.
در فاز آزمون نیز مدل را به تابع test که بالاتر آن را توضیح دادیم ورودی می‌دهیم تا میزان ضرر و دقت بر روی داده‌های تست محاسبه شود و میزان ضرر در هر ایپوک را چاپ می‌کنیم:

```python
for e in range(overfit_epochs):
    overfit_model.train()
    running_loss = 0
    for images, labels in trainloader:
        images = images.view(images.shape[0], -1)
        overfit_optimizer.zero_grad()
        output = overfit_model(images)
        loss = criterion(output, labels)
        loss.backward()
        overfit_optimizer.step()
        running_loss += loss.item()
    train_loss = running_loss / len(trainloader)
    train_losses.append(train_loss)

    # Test the model
    test_loss, test_accuracy = test(overfit_model, testloader)
    test_losses.append(test_loss)
    test_accuracies.append(test_accuracy)

    print(f"Epoch {e+1}/{overfit_epochs}, Train Loss: {train_loss:.4f}, Test Loss: {test_loss:.4f}")
```

در آخر با استفاده از کتابخانه matplotlib نمودار ضرر آموزش و آزمون را نمایش می‌دهیم. همانطور که مشاهده می‌شود مدل دارای بیش برازش است زیرا برخلاف روند کاهشی ضرر بر روی داده‌های آموزشی، میزان ضرر بر روی داده‌های تست بسیار بیشتر است و در ایپوک‌های پایانی روند افزایشی داشته است:

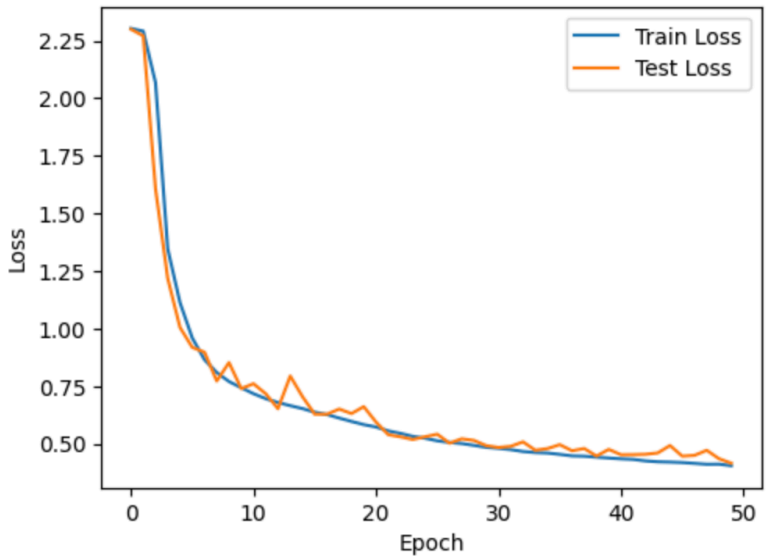
```
Epoch 40/50, Train Loss: 0.1190, Test Loss: 0.4390
Epoch 41/50, Train Loss: 0.1132, Test Loss: 0.4745
Epoch 42/50, Train Loss: 0.1088, Test Loss: 0.4734
Epoch 43/50, Train Loss: 0.1058, Test Loss: 0.4186
Epoch 44/50, Train Loss: 0.1026, Test Loss: 0.4481
Epoch 45/50, Train Loss: 0.0988, Test Loss: 0.4766
Epoch 46/50, Train Loss: 0.0946, Test Loss: 0.4788
Epoch 47/50, Train Loss: 0.0945, Test Loss: 0.4809
Epoch 48/50, Train Loss: 0.0898, Test Loss: 0.4673
Epoch 49/50, Train Loss: 0.0878, Test Loss: 0.5141
Epoch 50/50, Train Loss: 0.0878, Test Loss: 0.4914
```



پ. حال بایست با داده‌افزایی از بیش‌برازش شدن مدل جلوگیری کنیم. دقت کنید کد مربوط به فرایند آموزش تغییری نکرده است و نیاز به توضیح مجدد نیست. فقط در بخش transform به کمک توابعی که torch در اختیارمان گذاشته داده‌افزایی کرده‌ایم که به توضیح این بخش می‌پردازیم.
من از ۳ تا از این توابع جهت داده‌افزایی استفاده کرده‌ام. تابع اول که RandomHorizontalFlip نام دارد به احتمال ۵۰ درصد، تصویر را در جهت افقی برعکس می‌کند. تابع RandomHorizantalFlip نیز همین کار را در جهت عمودی تکرار می‌کند و تابع RandomRotation نیز تصویر ورودی را به صورت رندم بین ۰ تا ۲۰ درجه می‌چرخاند. در آخر تصویر را به یک تنسور تبدیل می‌کنیم.
اگر به خروجی دقت کنیم می‌بینیم اختلاف ضرر مدل بر روی داده‌های تست و آموزشی بسیار کم شده است که این یعنی ما توانسته‌ایم از overfit شدن مدل جلوگیری کنیم.

```
transform_train = transforms.Compose([
    transforms.RandomHorizontalFlip(),
    transforms.RandomVerticalFlip(),
    transforms.RandomRotation(20),
    transforms.ToTensor(),
])
```

```
Epoch 40/50, Train Loss: 0.4381, Test Loss: 0.4756
Epoch 41/50, Train Loss: 0.4346, Test Loss: 0.4524
Epoch 42/50, Train Loss: 0.4322, Test Loss: 0.4529
Epoch 43/50, Train Loss: 0.4257, Test Loss: 0.4544
Epoch 44/50, Train Loss: 0.4218, Test Loss: 0.4602
Epoch 45/50, Train Loss: 0.4205, Test Loss: 0.4924
Epoch 46/50, Train Loss: 0.4184, Test Loss: 0.4464
Epoch 47/50, Train Loss: 0.4150, Test Loss: 0.4497
Epoch 48/50, Train Loss: 0.4112, Test Loss: 0.4716
Epoch 49/50, Train Loss: 0.4115, Test Loss: 0.4352
Epoch 50/50, Train Loss: 0.4048, Test Loss: 0.4157
```

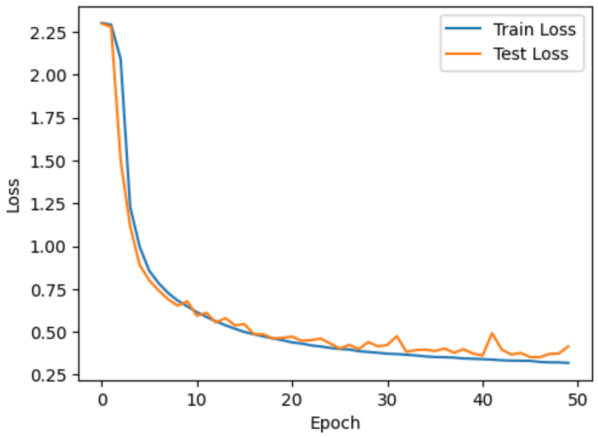


ت.
در این بخش نیز بایست از بیش‌برازش شدن مدل جلوگیری کنیم اما این بار از منظم‌سازی استفاده می‌کنیم. منظم‌سازی با جریمه کردن وزن‌های مدل از بزرگ شدن آن‌ها جلوگیری کرده و از احتمال بیش‌برازش می‌کاهد. تنها تغییر ایجاد شده در کد این قسمت تعریف یک ضریب منظم‌سازی است که بایست آن را طبق کد زیر به تابع بهینه‌ساز ورودی دهیم تا وزن‌ها را جریمه کند:

```
# Set the regularization strength (l2_lambda)
l2_lambda = 0.025

# Define the loss function with L2 regularization
criterion_l2 = nn.CrossEntropyLoss()

# Define the optimizer with L2 regularization
optimizer_l2 = optim.SGD(regularized_model.parameters(), lr=0.001, weight_decay=l2_lambda)  # Use weight_decay for L2 regularization
```

منظم‌سازی عملکرد خوبی بر روی مدل داشته و با ضریب ۰.۰۰۱ توانسته از بیش‌برازش مدل جلوگیری کند، از اختلاف ضرر داده‌های آموزشی و تست بکاهد (میانگین اختلاف در ۱۰ ایپوک آخر به حدود ۰.۰۵ رسیده است!) و بر روی داده‌های تست نیز عملکرد خوبی به نمایش بگذارد.

```
Epoch 40/50, Train Loss: 0.3430, Test Loss: 0.3733
Epoch 41/50, Train Loss: 0.3406, Test Loss: 0.3613
Epoch 42/50, Train Loss: 0.3377, Test Loss: 0.4916
Epoch 43/50, Train Loss: 0.3341, Test Loss: 0.3975
Epoch 44/50, Train Loss: 0.3320, Test Loss: 0.3675
Epoch 45/50, Train Loss: 0.3313, Test Loss: 0.3772
Epoch 46/50, Train Loss: 0.3305, Test Loss: 0.3526
Epoch 47/50, Train Loss: 0.3243, Test Loss: 0.3517
Epoch 48/50, Train Loss: 0.3220, Test Loss: 0.3709
Epoch 49/50, Train Loss: 0.3218, Test Loss: 0.3738
Epoch 50/50, Train Loss: 0.3186, Test Loss: 0.4149
```



ث. در این بخش بایست از هر ۳ روش منظم‌سازی، داده‌افزایی و dropout استفاده کنیم. برای دو روش اول طبق آن‌چه در دو بخش قبلی توضیح داده شد عمل می‌کنیم و برای dropout هر خروجی را به احتمال 20 درصد با استفاده از کد زیر غیرفعال می‌کنیم:

```
nn.Dropout(0.2),  # Adding dropout after the second ReLU
```

از آن‌جا که داریم از چند نوع روش برای جلوگیری از بیش‌برازش شدن مدل استفاده می‌کنیم. بهتر است به هیچ یک وزن خیلی زیادی ندهیم تا توان مدل زیاد گرفته نشود و مدل به سمت آندرفیت شدن حرکت نکند. از این رو تنها از ۲ نوع تابع داده‌افزایی استفاده شده، ضریب منظم‌سازی ۰.۰۰۱ و ضریب drop out نیز ۰.۲ در نظر گرفته شده. البته این مقادیر که در واقع هایپرپارامتر هستند بر اثر تجربه به دست آمده‌اند و من مدل را با مقادیر متفاوت آموزش داده‌ام تا بهترین مقادیر هایپرپارامترها را پیدا کنم.
همانطور که مشخص است این مدل نسبت به مدل‌های قبلی عملکرد بهتری داشته است و توانایی تعمیم آن بر روی داده‌هایی که تا به حال مشاهده نکرده است قابل توجه است، تنها در چند ایپوک آخر دچار نوسان شده که نشان می‌دهد بهتر بود به ۴۰ ایپوک اکتفا می‌کردیم: