یادگیری عمیق

تمرین دوم

استاد درس: دکتر محمدی

مهسا موفق بهروزی

زمستان ۱۴۰۱

سوال یک)

الف: به سراغ پرسپترون چندلایه رفتیم تا بتوانیم با اضافه کردن لایه‌های میانی، استخراج ویژگی انجام دهیم. درصورتی‌که از توابع فعال‌سازی استفاده نکنیم، به عنوان مثال در یک شبکه ۲ لایه، قرار دادن دو لایه‌ی خطی متوالی هیچ تفاوتی با یک لایه‌ی خطی ندارد. زیرا که:

$$O = (XW^1 + b^1)\, W^2 + b^2$$
$$= XW^1W^2 + b^1W^2 + b^2$$
$$= XW + b$$

بنابراین در شبکه‌های عصبی حتما از تابع غیرخطی یا همان تابع فعال‌سازی استفاده می‌کنیم تا فرمول خطی را به نحوی غیرخطی کنیم.

ب: در زمان آموزش، نورون یا صفر می‌شود یا تقسیم بر dropout rate – 1 می‌شود. با توجه به mask داده شده، نورون دوم و چهارم صفر شده و مابقی تقسیم بر ۰.۵ (ضرب در ۲) می‌شود.

خروجی به صورت: [3 , 0 , 6 , 0 , -9] خواهد بود.

البته باتوجه به این‌که در کلاس حل تمرین گفته شد که dropout rate = 0.4 خواهد بود، به جز نورون دوم و چهارم که صفر می‌شوند، مابقی بر ۰.۶ تقسیم می‌شوند و داریم: [2.5 , 0 , 5 , 0 , -7.5]

در زمان تست، هیچ فرقی ندارد که dropout داشته یا نداشته است. در حقیقت با حالتی که dropout نداریم یکسان است.

و داریم: [1.5 , 6 , 3 , -0.75 , -4.5]

ج: یک شبکه دو لایه ساده با تعداد نورون کافی می‌تواند هر تابعی را پیاده‌سازی کند. برای مساله‌ی دسته‌بندی دو کلاسه به نظر می‌رسد اضافه کردن ۲ نورون به نورون‌های قبلی و داشتن ۴ نورون برای حل مساله کافی نیست. شاید بتوان با اضافه کردن ۲۵۶ نورون شبکه را به خوبی آموزش داد ولی اضافه کردن یک میلیون نورون، باعث افزایش پارامترها شده و درنتیجه پیچیدگی مدل افزایش پیدا کرده و ممکن است دچار overfitting شود.

د: یکی از دلایل gradient vanishing می‌تواند استفاده از توابع فعال‌سازی‌ای مانند sigmoid یا tanh باشد که مشتق آن‌ها در بسیاری از مواقع نزدیک صفر است که باعث می‌شود گرادیان بسیار کوچک بوده و شبکه به خوبی یاد نگیرد یا اصلا یاد نگیرد.

دلایل دیگری نیز از جمله مقدار دهی اولیه وزن‌ها، عمق شبکه و استفاده از الگوریتم‌های بهینه‌سازی قدیمی نیز ممکن است سبب ایجاد gradient vanishing شوند.

برای جلوگیری از این اتفاق می‌توان از توابع فعال‌سازی‌ای مانند ReLU یا مشتقات آن و یا از الگوریتم‌های بهینه‌سازی‌ای مانند Adam استفاده کرد.

سوال دو)

الف: نادرست. تکنیک‌های منظم‌سازی مانند L1 و L2 و ... می‌توانند عملکرد مدل را بهبود دهند و از overfitting جلوگیری کنند.

ب: نادرست. این کار حتی می‌تواند احتمال overfitting را افزایش دهد به این صورت که اگر مدل دارای تعداد ویژگی‌های بسیار زیادی نسبت به تعداد داده‌های آموزش باشد، می‌تواند باعث افزایش پیچیدگی مدل و overfitting شود.

ج: نادرست. با افزایش ضریب منظم‌سازی، پنالتی مدل برای اندازه پارامترها نیز افزایش می‌یابد و باعث جلوگیری از overfitting میشود. هرچند که ممکن است باعث underfitting نیز شود. بنابراین بسیار مهم است که مقدار مناسبی انتخاب شود.

سوال سه)

الف: محاسبات به کمک پایتون انجام شده است.

```
def sigmoid(x):
  return 1/(1 + np.exp(-x))
```

$$H_3 = x_1 w_{13} + x_2 w_{23} = 0.35 \times 0.1 + 0.9 \times 0.8 = 0.755$$

$$H_4 = x_1 w_{14} + x_2 w_{24} = 0.35 \times 0.4 + 0.9 \times 0.6 = 0.68$$

$$y_3 = \sigma(H_3) = \sigma(0.755) = 0.6802671966986485$$

$$y_4 = \sigma(H_4) = \sigma(0.68) = 0.6637386974043528$$

$$O_5 = y_3 w_{35} + y_4 w_{45} = 0.68 \times 0.3 + 0.66 \times 0.9 = 0.801444986673512$$

$$y_5 = \sigma(O_5) = 0.6902834929076443$$

ب: با در نظر گرفتن تابع ضرر MSE، خطا برابر است با:

$$Loss = (y - y_5)^2 = (y - \sigma(O_5))^2 = (y - \sigma(y_3 w_{35} + y_4 w_{45}))^2$$

طبق chain rule داریم:

$$\frac{\partial Loss}{\partial w_{35}} = \frac{\partial Loss}{\partial y_5}\frac{\partial y_5}{\partial O_5}\frac{\partial O_5}{\partial w_{35}} = -2(y - y_5) \times \sigma(O_5)\big(1 - \sigma(O_5)\big) \times y_3$$
$$= 0.055348069877435735$$

$$\frac{\partial Loss}{\partial w_{45}} = \frac{\partial Loss}{\partial y_5}\frac{\partial y_5}{\partial O_5}\frac{\partial O_5}{\partial w_{45}} = -2(y - y_5) \times \sigma(O_5)\big(1 - \sigma(O_5)\big) \times y_4$$
$$= 0.05400327398201483$$

$$\frac{\partial Loss}{\partial w_{13}} = \frac{\partial Loss}{\partial y_5}\frac{\partial y_5}{\partial O_5}\frac{\partial O_5}{\partial y_3}\frac{\partial y_3}{\partial H_3}\frac{\partial H_3}{\partial w_{13}}$$
$$= -2(y - y_5) \times \sigma(O_5)\big(1 - \sigma(O_5)\big) \times w_{35} \times \sigma(H_3)\big(1 - \sigma(H_3)\big) \times x_1$$
$$= 0.0018581423216193194$$

$$\frac{\partial Loss}{\partial w_{23}} = \frac{\partial Loss}{\partial y_5}\frac{\partial y_5}{\partial O_5}\frac{\partial O_5}{\partial y_3}\frac{\partial y_3}{\partial H_3}\frac{\partial H_3}{\partial w_{23}}$$
$$= -2(y - y_5) \times \sigma(O_5)\big(1 - \sigma(O_5)\big) \times w_{35} \times \sigma(H_3)\big(1 - \sigma(H_3)\big) \times x_2$$
$$= 0.004778080255592536$$

$$\frac{\partial Loss}{\partial w_{14}} = \frac{\partial Loss}{\partial y_5}\frac{\partial y_5}{\partial O_5}\frac{\partial O_5}{\partial y_4}\frac{\partial y_4}{\partial H_4}\frac{\partial H_4}{\partial w_{14}}$$
$$= -2(y - y_5) \times \sigma(O_5)\big(1 - \sigma(O_5)\big) \times w_{45} \times \sigma(H_4)\big(1 - \sigma(H_4)\big) \times x_1$$
$$= 0.005720151544890908$$

$$\frac{\partial Loss}{\partial w_{24}} = \frac{\partial Loss}{\partial y_5}\frac{\partial y_5}{\partial O_5}\frac{\partial O_5}{\partial y_4}\frac{\partial y_4}{\partial H_4}\frac{\partial H_4}{\partial w_{24}}$$

$$= -2(y - y_5) \times \sigma(O_5)\big(1 - \sigma(O_5)\big) \times w_{45} \times \sigma(H_4)\big(1 - \sigma(H_4)\big) \times x_2$$

$$= 0.014708961115433766$$

حال می‌توان وزن‌ها را آپدیت کرد، طبق فرض مساله نرخ یادگیری برابر ۱ است.

$$w_{35} = w_{35} - \alpha\left(\frac{\partial Loss}{\partial w_{35}}\right) = 0.24$$

$$w_{45} = w_{45} - \alpha\left(\frac{\partial Loss}{\partial w_{45}}\right) = 0.84$$

$$w_{13} = w_{13} - \alpha\left(\frac{\partial Loss}{\partial w_{13}}\right) = 0.09$$

$$w_{23} = w_{23} - \alpha\left(\frac{\partial Loss}{\partial w_{23}}\right) = 0.79$$

$$w_{14} = w_{14} - \alpha\left(\frac{\partial Loss}{\partial w_{14}}\right) = 0.39$$

$$w_{24} = w_{24} - \alpha\left(\frac{\partial Loss}{\partial w_{24}}\right) = 0.58$$

سوال چهار)

مقدار batch_size برابر با ۱۲۸ قرار داده شده است.

```
## TODO: set number of samples per batch to load
batch_size = 128
```

در بخش define network architecture، برای طراحی ماژولار شبکه، از nn.Module ارث‌بری کرده‌ایم. همچنین پارامتر activation_function به منظور امتحان کردن توابع فعال‌سازی متفاوت به جای تعریف ثابت یک تابع فعال‌سازی، به مدل پاس داده شده است.
لایه ورودی با تعداد نورون ۲۸ * ۲۸ (برابر با سایز تصاویر) به یک لایه نهان با ۱۰۰ نورون متصل می‌شود. لایه نهان اول نیز به لایه نهان بعدی با ۵۰ نورون متصل شده است. برای جلوگیری از overfit شدن مدل، از dropout استفاده کرده‌ایم و سپس لایه نهان دوم به لایه خروجی با ۱۰ نورون (کلاس‌های ۰ تا ۹) متصل شده است.
برای استفاده از تابع فعال‌سازی در هر لایه از پارامتر activation_function استفاده کرده‌ایم که در ادامه کاربرد آن را خواهیم دید.

```
import torch.nn as nn
import torch.nn.functional as F

## TODO: Define the NN architecture
#class Net(nn.Module):
    #def __init__(self):


    #def forward(self, x):


class Net(nn.Module):
    def __init__(self,activation_function):
        super(Net,self).__init__()


        self.linear1 = nn.Linear(28*28, 100)
        self.linear2 = nn.Linear(100, 50)
        self.dropout = nn.Dropout(p=0.2)
        self.final = nn.Linear(50, 10)
        #self.relu = nn.ReLU()
        self.activation_function = activation_function()

    def forward(self, image):
        a = image.view(-1, 28*28)
        #a = self.relu(self.linear1(a))
        #a = self.relu(self.linear2(a))
        a = self.activation_function(self.linear1(a))
        a = self.activation_function(self.linear2(a))
        a = self.final(a)
        return a
```

در بخش زیر، آرایه‌ی مدل، که شاید بهتر بود اسم آن را activation functions قرار میدادیم، تعریف شده است. این آرایه شامل ۳ تابع فعال‌سازی ReLU، LeakyReLU و sigmoid می‌باشد. به هرکدام یک tag اختصاص داده شده است تا هنگام ترکیب توابع فعال‌سازی با توابع ضرر و بهینه‌سازهای دیگر، مشخص شود از کدام یک استفاده کرده‌ایم.

```
# initialize the NN
model = [
    {
        "tag": "relu ",
        "item": nn.ReLU #Net()
    },
    {
        "tag": "LeakyReLU ",
        "item": nn.LeakyReLU  #Net(nn.LeakyReLU)
    },
    {
        "tag": "Sigmoid ",
        "item": nn.Sigmoid # Net(nn.Sigmoid)
    }
]
print(model)
```

در ادامه، تابع ضرر و بهینه‌ساز همانند تابع فعال‌سازی تعریف شده‌اند. از آنجایی که مساله، مساله‌ی classification می‌باشد، از توابع ضرر مخصوص classification یعنی CE و NLL استفاده شده، همچنین SGD و Adam به عنوان بهینه‌ساز انتخاب شده‌اند.

```
## TODO: specify loss function
criterion = [
    {
        "tag": "loss function: cronss entropy ",
        "item": nn.CrossEntropyLoss()
    },
    {
        "tag": "loss function: nllloss ",
        "item": nn.NLLLoss()
    },
    # {
    #     "tag": "loss function: L1Loss()",
    #     "item": nn.L1Loss()
    # }
]

## TODO: specify optimizer
optimizer = [
    {
        "tag": "optimizer: SGD ",
        "item": torch.optim.SGD
    },
    {
        "tag": "optimizer: Adam ",
        "item": torch.optim.Adam
    }
]
```

برای آموزش مدل، یکی از راه‌هایی که می‌توان overfitting را تشخیص داد، استفاده از بخشی از داده‌ها به عنوان داده‌های validation می‌باشد. از آن‌جایی که در نوت‌بوک داده شده، داده‌ها به دو بخش تست و آموزش تقسیم شده اند و از ابتدا داده‌ی ولیدیشن در نظر گرفته نشده است، با در نظر گرفتن validation_size = 0.2، می‌خواهم ۲۰ درصد انتهایی داده‌های آموزشی را به عنوان داده ولیدیشن انتخاب کنم تا overfitting را تشخیص دهیم.

برای در نظر گرفتن ترکیب تمام توابع فعال‌سازی، توابع ضرر و بهینه‌سازهایی که قبلا تعریف کرده‌ایم ۳ حلقه for ایجاد شده است. هر بار به ازای هر تابع فعال‌سازی و هر بهینه‌ساز، یک مدل جدید ایجاد شده و در صورتی که داده‌هایی که میخواهیم مدل را با آن‌ها آموزش دهیم، متعلق به ۸۰ درصد اول داده آموزشی باشند، (مابقی را به عنوان داده ولیدیشن استفاده خواهیم کرد) فرآیند آموزش مدل به ازای هر تابع ضرر انجام خواهد شد.

متغیر old_valid_loss که در ابتدا یک مقدار بزرگ (۹۹۹۹۹۹۹۹۹۹) دارد، برای تشخیص overfitting استفاده خواهد شد. به این نحو که زمانی که مدل را به فاز evaluation می‌بریم، اگر داده‌ها متعلق به داده‌های ولیدیشن باشند، خروجی مدل و مقدار loss حساب خواهد شد و هر بار این مقدار loss به متغیر valid_loss که در ابتدای هر epoch برابر با صفر قرار داده می‌شود، اضافه می‌گردد. و اگر مقدار خطا بیشتر از متغیر old_valid_loss باشد، نشان دهنده‌ی وقوع overfitting است. در ابتدا مقدار این متغیر عدد بزرگی‌ست تا شبکه در همان ابتدا درست کار کند. همچنین مقدار old_valid_loss هربار با مقدار جدید (valid_loss) جایگزین می‌شود.

```
validation_size = 0.2
epoch = 30
models = []

def train(epoch):
  for modelItem in model:
    for criterionItem in criterion:
      for optimizerItem in optimizer:
        old_valid_loss = 9999999999
        myModel = Net(modelItem['item'])
        mOptimizer = optimizerItem['item'](myModel.parameters(),lr=0.1)
        for epoch in range(epoch):
          valid_loss = 0
          myModel.train()
          for i, (data, target) in enumerate(train_loader):
            if i < int(0.8*len(train_loader)):
              mOptimizer.zero_grad()
              output = myModel(data)
              loss = criterionItem['item'](output, target)
              loss.backward()
              mOptimizer.step()

          myModel.eval()
          for i, (data, target) in enumerate(train_loader):
              if i > int(0.8*len(train_loader)):
                output = myModel(data)
                loss = criterionItem['item'](output, target)
                valid_loss += loss.item()*data.size(0)
          if valid_loss > old_valid_loss:
            print(f'overfit occured in epoch {str(epoch)}')
          old_valid_loss = valid_loss
        models.append(
            {
                "tag": modelItem['tag'] + criterionItem['tag'] + optimizerItem['tag'],
                "model": myModel
            }
        )

train(epoch)
```

برای تست، به ازای هر کدام از مدل‌هایی که ایجاد کردیم، به حالت evaluate برده و داده‌ی تست را به مدل داده و خروجی را به دست می‌آوریم. خروجی به این شکل است که به ازای هر کلاس، یک احتمال برمی‌گرداند. ایندکسِ بزرگترین احتمال را در pred قرار داده و با مقدار احتمال آن کاری نداریم. خروجی مدل که به صورت یک تنسور می‌باشد را تبدیل به یک numpy array کرده و با inner_y_pred.extend به آرایه‌ی inner_y_pred اضافه می‌کنیم. همین کار را برای target نیز تکرار می‌کنیم.

از آن‌جایی که چند مدل داریم که می‌خواهیم داده‌های تست را برای هر کدام اعمال کنیم، inner_y_pred و inner_y_test را در mymodel['y_pred'] و mymodel['y_test'] قرار می‌دهیم تا نتایج واقعی و نتایجی که مدل پیش‌بینی می‌کند را در y_pred و y_test همان مدل ذخیره کنیم.

```
## TODO:
################################################
#               test the model                 #
################################################

def test():
  for mymodel in models:
    inner_y_pred = []
    inner_y_test = []
    mymodel['model'].eval()
    with torch.no_grad():
      for data, target in test_loader:
        output = mymodel['model'](data)
        _, pred = torch.max(output.data, -1)
        inner_y_pred.extend(pred.cpu().detach().numpy())
        inner_y_test.extend(target.cpu().detach().numpy())
    mymodel['y_pred'] = inner_y_pred
    mymodel['y_test'] = inner_y_test
test()
```

بخش بعدی، از بخش‌های آماده نوت بوک بود که با توجه به اینکه چندین مدل داشتیم، نیاز به تغییراتی داشت تا با شیوه‌ای که نوشته‌ام کار کند.

این بخش پیش‌بینی مدل و target را در کنار یک‌دیگر قرار می‌دهد و یک دید از عملکرد مدل ارائه می‌دهد. با توجه به نتایج، می‌توان گفت ترکیبات زیر بهترین نتایج را داشته اند:

بهینه‌ساز	تابع ضرر	تابع فعال‌سازی
SGD	cronss entropy	relu
SGD	cronss entropy	LeakyReLU
SGD	cronss entropy	Sigmoid
Adam	cronss entropy	Sigmoid

برای محاسبه‌ی avg test loss، به ازای هر مدل تعداد پیش‌بینی‌های اشتباه را تقسیم بر کل داده‌های تست می‌کنیم. که مجددا مشاهده می‌شود، ترکیباتی که در جدول بالا وجود دارند، منجر به میانگین خطای کمتری شده‌اند و تنها این ترکیبات، مقدار خطای زیر ۰.۱ داشته‌اند.

```
## TODO: calculate and print avg test loss
# test_loss =
# print('Test Loss: {:.6f}\n'.format(test_loss))

for mymodel in models:
  y_pred = mymodel['y_pred']
  y_test = mymodel['y_test']
  fault = 0
  for x in range (0,len(y_pred)):
    if y_pred[x] != y_test[x]:
      fault = fault + 1
  print(mymodel['tag'])
  print('Test Loss: {:.6f}\n'.format(fault/len(y_pred)))
```

باقی بخش‌های نوت‌بوک، To Do نداشتند و صرفا به نحوی آن‌ها را تغییر دادم که بتواند با شیوه‌ای که نوشته‌ام کار کنند.