تمرین HW2 یادگیری عمیق – محمد اصولیان 99521073

Q1)

الف)

در شبکه‌های عصبی، overfittingو underfitting دو مشکل متفاوت هستند که ممکن است در فرآیند آموزش و استفاده از مدل‌های عصبی رخ دهند.

Overfitting:

Overfittingبه وقوع می‌پیوندد وقتی مدل عصبی به طور غیرمنطقی و بیش از حد به داده‌های آموزشی خود عادت کند و نتایج قابل قبولی روی داده‌های آموزشی حاصل کند، اما نتواند به خوبی بر روی داده‌های جدید یا داده‌های آزمون عمل کند. به طور خاص، علامت‌های شایع overfittingعبارتند از:

عملکرد بسیار خوب روی داده‌های آموزشی، اما عملکرد ضعیف روی داده‌های جدید.

اختلاف بسیار بزرگ بین خطای آموزش و خطای آزمون.

مدل کاملاً پاسخگو به تفاوت‌های جزئی در داده‌های آموزشی که معمولاً نویز یا جزئیات غیرضروری هستند.

Overfittingممکن است از بهینه‌سازی بیش از حد پیچیده مدل، تعداد کم داده‌های آموزش، عدم تعادل در توزیع داده‌ها، یا استفاده از ویژگی‌های غیرضروری در داده‌های آموزش ناشی شود.

برخی روش‌های مقابله با overfitting عبارتند از:

استفاده از تکنیک‌های منظم‌سازی مانند dropout و.regularization

استفاده از تکنیک‌های کاهش بعد مدل مانند.pruning

افزایش تعداد داده‌های آموزش.

استفاده از اعتبارسنجی متقاطع (cross-validation) برای ارزیابی مدل.

Underfitting:

Underfittingبه وقوع می‌پیوندد وقتی مدل عصبی عملکرد ضعیفی روی داده‌های آموزشی و جدید داشته باشد. به طور خاص، علامت‌های شایع underfitting عبارتند از:

عملکرد ضعیف روی داده‌های آموزشی و داده‌های جدید.

عدم قادر بودن مدل به یادگیری الگوهای مهم در داده‌ها.

خطای آموزش و خطای آزمون هر دو بالا و نزدیک به هم.

Underfittingممکن است ناشی از ساختار ساده مدل، تعداد کم لایه‌ها یا نرون‌ها، یا ناکافی بودن فرآیند آموزش باشد.

برخی روش‌هایمقابله با underfitting عبارتند از:

افزایش پیچیدگی مدل، مانند افزودن لایه‌های بیشتر یا افزایش تعداد نرون‌ها در هر لایه.

افزایش تعداد دوره‌های آموزش.

استفاده از ویژگی‌های بیشتر یا بهتر در داده‌های آموزش.

استفاده از روش‌های پیش‌پردازش داده برای بهبود کیفیت و استفاده از ویژگی‌های مهم.

در هر دو حالت، تلاش برای ایجاد توازن مناسب بین underfitting و overfitting مهم است. ایده‌آل استفاده از روش‌های ارزیابی مانند اعتبارسنجی متقاطع و تنظیم پارامترهای مدل به نحوی است که بهترین عملکرد روی داده‌های آموزش و داده‌های جدید را داشته باشد.

ب)

معیار overfitting مدل از مقایسه نتایج تست و train به دست می‌آید. در صورتی که به هیچ یک از آمار و نتایج و حتی داده‌های فرایند تست دسترسی نداشته باشیم نمی‌توان مشخص کرد که مدل overfit شده است یا نه. لازم است که حتما اطلاعاتی از فرایند ترین داشته باشیم تا بتوانیم در مورد overfitting قضاوت کنیم.

پ)

پی دی اف جواب در فایل زیپ موجود است.

Q2)

الف)

با کاهش مقدار k، مقدار بایاس هم کاهش پیدا میکند اما واریانس افزایش پیدا میکند. مثلا اگر فرض کنیم که k=1، نزدیک ترین همسایه هر نمونه در مجموعه train، خود آن نمونه خواهد بود. بنابراین تمام پیشبینی ها صحیح خواهند بود و بایاس 0 میشود. اما اگر با همین فرض نمونه های جدید را تست کنیم، به علت حساسیت بسیار بالا روی نزدیک ترین همسایه احتمالا نمونه های تست خطای زیادی خواهند داشت و این تفاوت در مجموعه تست و ترین به معنای واریانس بالا است.

اما اگر مقدار k را افزایش دهیم، مقدار واریانس کاهش و مقدار بایاس افزایش پیدا میکند. چون مدل به مجموعه بزرگ تری از نقاط نگاه میکند. این موضوغ باعث میشود که تفاوت داده های تست و ترین کم تر شود که منجر به کمتر شدن واریانس است اما در عین حال ممکن است برخی داده های ترین هم دچار اشتباه شوند.

ب)

استفاده از منظم سازی، ممکن است باعث تضعیف عملکرد مدل شود

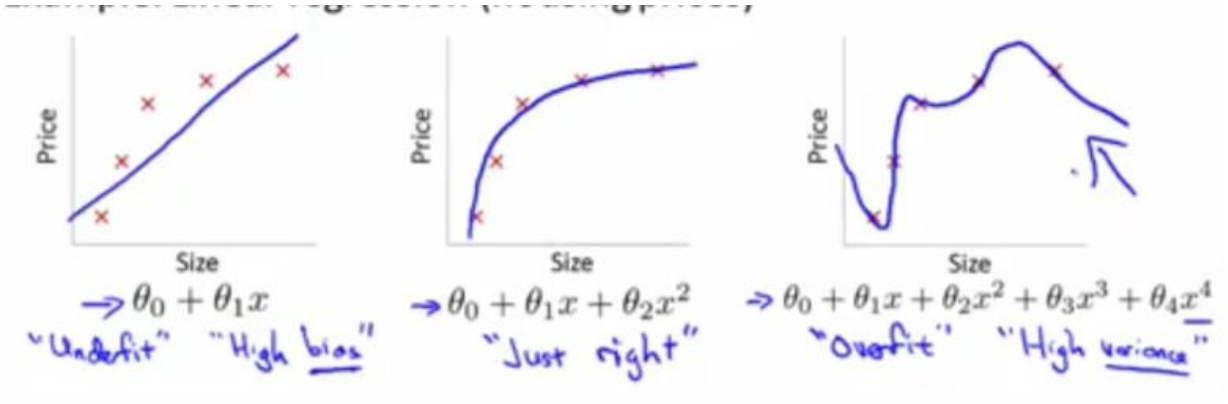
صحیح

در صورتی که بیش از اندازه از منظم سازی استفاده شود یا بدون وجود overfitting از منظم سازی استفاده شود، ممکن است که مدل پیچیدگی و انعطاف پذیری خود را از دست بدهد. در واقع قدرت مدل کاهش پیدا میکند ونمیتواند به خوبی عملیات یادگیری را انجام دهد. یعنی ممکن است مدل دچار underfitting شود.

اضافه کردن تعداد زیاد ویژگی های جدید، باعث جلوگیری از بیش برازش می شود

غلط

در صورتی که مدل قدرت و انعطاف پذیری کافی داشته باشد، افزودن فیچرهای جدید میتواند منجر به این شود که مدل فیچرهای جدید را هم حفظ کند هر چند تاثیری در پاسخ مدل نداشته باشند و این موضوع باعث میشود که مدل نسبت به برخی فیچرهای بی اهمیت حساس شود و در صورتی که یک نمونه جدید در فیچرهای بی اهمیت اشتراکی با نمونه های آموزش نداشته باشد، پاسخی اشتباه برای آن پیشبینی شود. در کل در صورتی که فیچرهای اضافه شده به خوبی نمایانگر نمونه ها نباشند، این کار میتواند منجر به افزایش overfitting شود.



با زیاد کردن ضریب منظم سازی، احتمال بیش برازش بیشتر می شود.

غلط

با زیاد کردن ضریب منظم سازی، مقدار وزن ها بیشتر کاهش میابد. این یعنی که انعطاف پذیری و قدرت مدل کم میشود و این موضوع، overfitting را کاهش میدهد نه افزایش. در واقع هر چه ضریب منظم سازی بیشتر باشد، مدل ساده تر خواهد بود و مدل ساده تر احتمال کم تری در overfitting دارد.

پ)

نمونه اول:

- $W_{exp1} = [0.26, 0.25, 0.25, 0.25]$

در این نمونه احتمالا از L2 استفاده شده. خاصیت L2 این است که سعی میکند که همه وزن ها را به صورت یک دست کاهش دهد که در اینجا اتفاق افتاده.

نمونه دوم:

- $Wexp2 = [1, 0, 0, 0]$

این نمونه احتمالا L1 است. در نرمال سازی L1، این احتمال وحود دارد که وزن یک یا چند تا از فیجر ها صفر شود که در این قسمت اتفاق افتاده. به همین علت است که از L1 برای حذف فیچرها استفاده میکنند.

نمونه سوم:

- $W_{exp3} = [13.3, 23.5, 53.2, 5.1]$

در این نمونه وزن ها زیاد هستند و یک دست هم نیستند بنابراین به احتمال زیاد در این نمونه از نرمال سازی استفاده نشده.

نمونه چهارم:

- $W_{exp4} = [0.5, 1.2, 8.5, 0]$

برای این نمونه نمیتوان خیلی با قطعیت گفت. مقدار وزن ها نسبت به نمونه سوم کاهش داشته اما نسبت به نمونه یک و دو هنوز زیاد است. احتمال دقیقا صفر شدن در L2 خیلی کم است بنابراین به احتمال زیاد این نمونه برای L1 است.

(Q3

الف)

در زمینهٔ یادگیری عمیق، تقطیر دانش به معنای انتقال دانش از یک مدل پیچیده و حجیم (مدل اصلی) به یک مدل ساده‌تر و کم‌حجیم‌تر (مدل تقطیر) است. این روش برای انتقال دانش از یک مدل به مدل دیگر با هدف افزایش سرعت آموزش، کاهش حجم مدل‌ها، و حتی افزایش توانایی عملکرد مدل‌ها در مواردی مورد استفاده قرار می‌گیرد. مثلا در مواقعی که بخواهیم از مدل، بر روی دستگاه هایی با سخت افزار محدود استفاده کنیم. مثل موبایل، embeded system ها و ...

مراحل اصلی تقطیر دانش در یادگیری عمیق عبارتند از:

آموزش مدل اصلی :(Teacher Model) یک مدل پیچیده و عمیق به نام مدل استاذ یا مدل اصلی آموزش داده می‌شود. این مدل معمولاً دقت بالا و توانمندی خوبی در مسائل مورد نظر دارد.

استخراج دانش :(Knowledge Extraction) دانش موجود در مدل اصلی استخراج می‌شود. این می‌تواند شامل وزن‌های مدل، توزیع‌های احتمالاتی، ویژگی‌ها، الگوها، و دیگر اطلاعات مفید باشد.

آموزش مدل تقطیر :(Distillation Model) یک مدل ساده‌تر با استفاده از دانش استخراج شده آموزش داده می‌شود. این مدل، که به آن مدل تقطیر یا دانش‌تقطیر می‌گویند، معمولاً از تعداد کمتری پارامتر و حجم کمتری نسبت به مدل اصلی برخوردار است.

ارزیابی و تنظیم :(Evaluation and Fine-tuning) عملکرد مدل تقطیر بر روی داده‌های ارزیابی بررسی می‌شود، و در صورت نیاز، ممکن است تنظیمات جزئی انجام شود تا عملکرد بهینه بر روی وظایف مورد نظر حاصل شود.

استفاده از تقطیر دانش در یادگیری عمیق به ما این امکان را می‌دهد که از دانش مدل‌های پیچیده بهره‌مند شویم و در عین حال، مدل‌های ساده‌تر و کم‌حجم با توانمندی مشابه یا حتی بهتر از نسخه اصلی ایجاد کنیم. این موضوع به خصوص در مواقعی مفید است که مدل اصلی بسیار پرهزینه است یا نیاز به استفاده در سیستم‌های با منابع محدود داریم.

استفاده از تقطیر دانش در یادگیری عمیق اهداف متعددی را پشتیبانی می‌کند. در زیر، اهم منافع و کاربردهای تقطیر دانش در یادگیری عمیق را بررسی می‌کنیم:

کاهش حجم مدل:

مدل‌های عمیق اغلب دارای حجم زیادی از پارامترها هستند که نیاز به منابع محاسباتی بسیار بالا دارد. با استفاده از تقطیر دانش، می‌توان حجم مدل را به شدت کاهش داد و مدل‌های ساده‌تر و سبک‌تر با عملکرد مشابه یا حتی بهتر ایجاد کرد.

انتقال دانش:

تقطیر دانش به عنوان یک روش انتقال دانش از یک مدل به مدل دیگر عمل می‌کند. این انتقال دانش می‌تواند از مدل پیچیده به مدل ساده، از یک حوزه به حوزه دیگر، یا از یک مسئله به مسئله دیگر انجام شود.

سرعت آموزش:

مدل‌های ساده‌تر، معمولاً سریعتر آموزش می‌بینند. با استفاده از تقطیر دانش و استفاده از یک مدل تقطیر، می‌توان زمان آموزش را به شدت کاهش داد و از منابع محاسباتی بهینه‌تری استفاده کرد.

عملکرد در منابع محدود:

در برخی از موارد، از جمله دستگاه‌های قدیمی‌تر یا با منابع محدود، ممکن است اجرای مدل‌های پیچیده مشکل باشد. مدل‌های تقطیر شده می‌توانند عملکرد خوبی با منابع کمتر داشته باشند.

ارتقاء عملکرد مدل‌های کم‌مصرف:

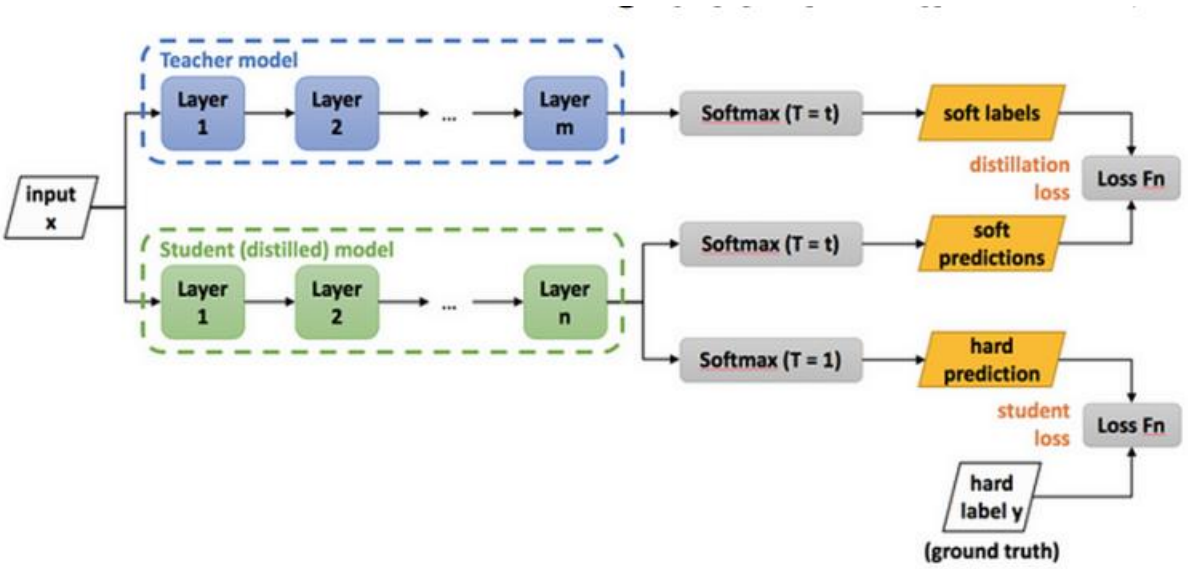
در برخی از موارد، مثلاً در اینترنت اشیاء (IoT) یا دیگر سیستم‌های با منابع محدود، استفاده از مدل‌های کم‌مصرف و سبک اهمیت دارد. تقطیر دانش می‌تواند به ارتقاء عملکرد مدل‌های کم‌مصرف کمک کند.

تعمیم بهتر:

مدل‌های تقطیر شده معمولاً دارای توانمندی تعمیم بهتری هستند. زیرا در فرآیند تقطیر، مدل سعی در بازنمایی اطلاعات مهم و کلیدی می‌کند که می‌تواند به تعمیم بهتر موجودیت‌ها و الگوهای داده کمک کند.

در کل، تقطیر دانش یک ابزار مهم در دامنه یادگیری عمیق است که برای بهبود عملکرد، کاهش پیچیدگی مدل، و افزایش کارایی در شرایط خاص مورد استفاده قرار می‌گیرد.

ب)



در این فرایند ابتدا مدل Teacher به صورت کامل آموزش میبیند تا همگرا شود. تابع لاس برای آموزش Teacher میتواند هر نوع تابعی باشد و بستگی به مسئله دارد. تابع لاس هنگام آموزش مدل Teacher از مقایسه بین پیشبینی های Teacher و لیبل‌های واقعی به دست میآید.

پس از این که آموزش مدل Teacher به اتمام رسید، با استفاده از تقطیر دانش، دانش استخراج شده به مدل student منتقل میشود. نحوه انتقال به این صورت است که ابتدا یک مدل سبک تر و مناسب سخت افزار برای مدل student انتخاب میکنیم، سپس مدل student را به کمک مدل Teacher آموزش میدهیم.

در عملیات forward، نمونه X را هم به مدل Student و هم مدل Teacher میدهیم. مدل Teacher پس از اعمال softmax با ضریب هموارسازی t، نتایجی را تولید میکند که به آن softlabel میگوییم. از softlabel برای محاسبه لاس برای مدل student استفاده میکنیم.

مدل Student هم در عملیات forward، با اعمال دو تابع softmax با ضرایب هموارسازی t و 1، نتایجی را تولید میکند. نتایج تولید شده توسط تابع با ضریب هموارسازی t، با softlabel ها مقایسه شده و از آنها برای محاسبه lossای به نام distillation loss استفاده میشود. نتایج تولید شده توسط تابع با ضریب هموارسازی 1 هم با لیبل های اصلی مقایسه شده و از آنها برای محاسبه student loss استفاده میشود.

در عملیات backpropagation، مدل Teacher دچار تغییر نمیشود. در واقع از مدل Teacher در این مرحله فقط برای forward استفاده میشود.

تابع loss نهایی مدل student از ترکیب distillation loss و student loss محاسبه میشود. سپس از لاس نسبت به وزن های student مشتق گرفته میشود و عملیات backpropagation روی وزن های student اعمال میشود و با تکرار این مراحل، مدل student آموزش دیده و همگرا میشود.

پ)

همانطور که در بخش قبل گفتم، لاس نهایی student از ترکیب distillation loss و student loss محاسبه میشود.

Teacher Loss L_T: (between actual lables and predictions by teacher network)

$$L_{TS} = \alpha * \text{Student Loss} + \text{Distallation Loss}$$

از یک ضریب آلفا برای تنظیم میزان اهمیت مدل به لیبل‌های اصلی و softlabel ها استفاده میشود. در این فرایند، t و alpha هایپرپارامترهای مدل هستند که میتوان آنها را تنظیم کرد. هر چه ضریب هموارسازی بیشتر باشد توزیع هموار تر خواهد بود.

Q4)

ابتدا کد های نوشته شد را توضیح میدهم و بعد نمودارها رو تحلیل میکنم.

در بلوک اول، torch برای ساخت مدل و matplotlib برای نمایش نمودار ها import شده:

```
[1]  import torch
     import matplotlib.pyplot as plt
```

در بلوک بعدی کلاس مدل ما تعریف شده که توابع زیر را دارد:

```
def _random_tensor(self, size): return (torch.randn(size)).requires_grad_()
```

این تابع برای ایجاد یک ماتریس رندوم همراه با ماتریس گرادیان آن است. از ماتریس گرادیان برای فرایند آموزش استفاده میشود در صورتی که از requires_grad استفاده نوشد، تنها یک ماتریس از اعداد بدون ماتریس گرادیان آنها خواهیم داشت.

```
def _initalize_moms(self):
    self.moms_w1, self.moms_b1 = [0], [0]
    self.moms_w2, self.moms_b2 = [0], [0]

def _initalize_RMSs(self):
    self.RMSs_w1, self.RMSs_b1 = [0], [0]
    self.RMSs_w2, self.RMSs_b2 = [0], [0]
```

این دو تابع، متغیرهایی که برای الگوریتم های momentum و rms نیاز داریم را initialize میکند.

```
def _initalize_parameters(self):
    self.weights_1 = self._random_tensor((x.shape[1],3))
    self.bias_1 = self._random_tensor(1)
    self.weights_2 = self._random_tensor((3,1))
    self.bias_2 = self._random_tensor(1)
```

این تابع وزن ها را با ماتریس های رندوم گرادیان دار initialize میکند.

```
def __init__(self, x, y):
    self.x = x
    self.y = y

    self._initalize_parameters()
    self._initalize_moms()
    self._initalize_RMSs()
```

این تابع، کل مدل را initailzie میکند. شامل وزن ها و متغیرهای الگوریتم های optimization

```
def _loss_func(self, preds, yb):
    return ((preds-yb)**2).mean()
```

این تابع، مقدار loss را با توجه به pred و لیبل اصلی حساب میکند. میبینیم که این لاس، همان MSE است.

```
def _nn(self, xb):
    l1 = xb @ self.weights_1 + self.bias_1
    l2 = l1.max(torch.tensor(0.0))
    l3 = l2 @ self.weights_2 + self.bias_2
    return l3
```

این تابع با ضرب ماتریس ها، عملیات forward pass را انجام میدهد و نتیجه نهایی را تولید میکند. میبینیم که برای خروجی های لایه اول، تابع فعالسازی relu استفاده شده.

```
for i, lr in enumerate(lrs):
    losses = []
    while(len(losses) == 0 or losses[-1] > 0.1 and len(losses) < 1000):
        preds = self._nn(self.x)
        loss = self._loss_func(preds, self.y)
        loss.backward()
        optimizer(self.weights_1, lr, self.moms_w1, self.RMSs_w1)
        optimizer(self.bias_1, lr, self.moms_b1, self.RMSs_b1)
        optimizer(self.weights_2, lr, self.moms_w2, self.RMSs_w2)
        optimizer(self.bias_2, lr, self.moms_b2, self.RMSs_b2)
        losses.append(loss.item())
    all_losses.append(losses)
```

در نهایت این حلقه اصلی تابع train است که مدل را آموزش میدهد. در شرط حلقه میبینیم که فرایند آموزش تا وقتی ادامه دارد که loss کم تر از 0.1 شود یا 1000 مرحله آموش دیده باشد.

در ادامه این تابع هم کدهایی برای نمایش نمودارها وجود دارند که پیچیدگی ای ندارند و نیاز به تحلیل خاصی ندارند.

```
[12] def generate_fake_labels(x3, x2, x1):
         return (x3**3 * 0.8) + (x2**2 * 0.1) + (x1 * 0.5) + 4.
```

```
[20] x = torch.tensor([[0.7,0.3,0.7],
                       [0.4, 1., 0.4],
                       [0.2, 1.1, 0.1],
                       [0.4, 0.7, 0.2],
                       [0.1, 0.5, 0.3]])
     y = torch.tensor([generate_fake_labels(i[0],i[1],i[2]) for i in x])
     print(x.shape, y.shape, y)

     torch.Size([5, 3]) torch.Size([5]) tensor([4.6334, 4.3512, 4.1774, 4.2002, 4.1758])
```

```
     y = torch.tensor([4.6334, 4.3512, 4.1774, 4.2002, 4.1758])
```

در این قسمت، نمونه های ورودی و لیبل های آنها را تولید میکنیم که فیک هستند و معنایی ندارند.

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

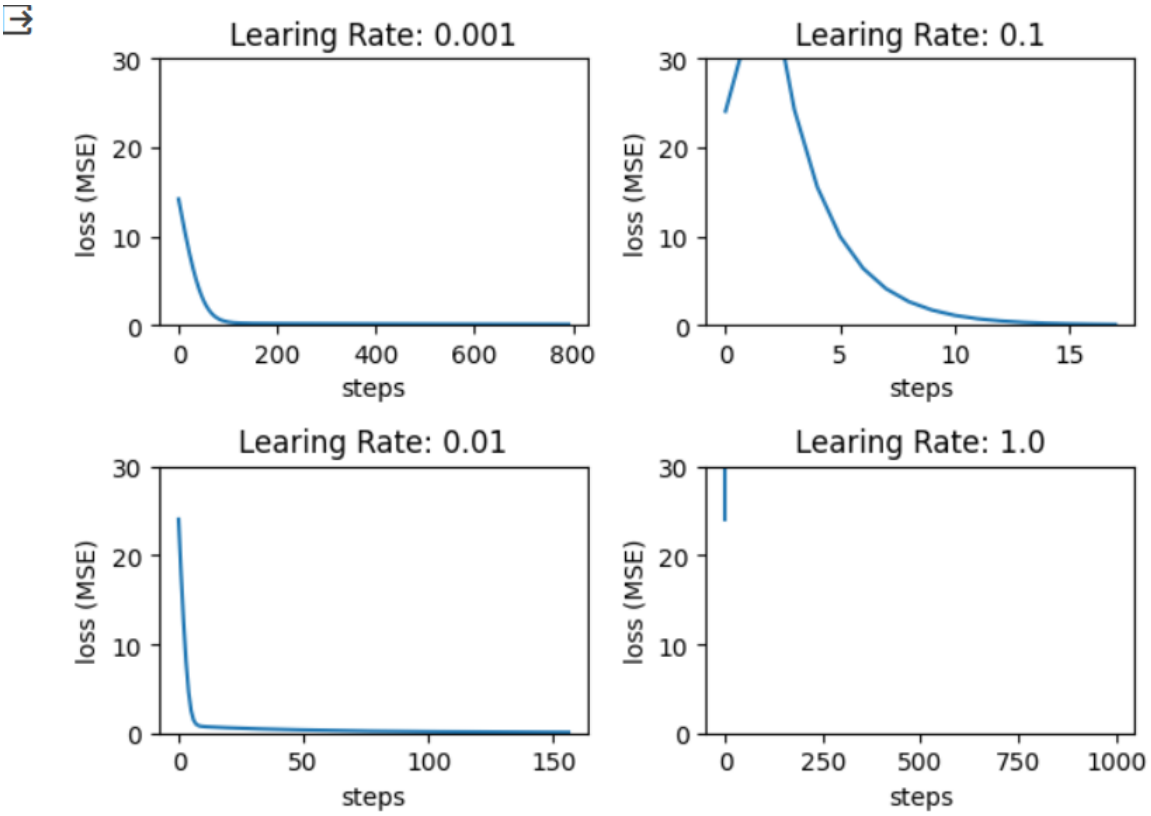
در اپتیمایزر SGD، تنها یک آپدیت ساده برای وزن ها انجام میدهیم. سپس گرادیان را صفر میکنیم تا برای دور بعدی آموزش آماده باشد.

```
[27] 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
```

در اپتیمایزر momentum، آپدیت وزن ها از طریق روش momentum انجام میگیرد که به این صورت است که زمانی که گرادیان جدید با گرادیان های قبلی هم جهت باشد، سرعت یادگیری بیشتر و در غیر این صورت، سرعت یادگیری کم تر است. این کار کمک میکند که از نوسان دوری کنیم و مدل با ثبات تر به سمت نقطه مینیمم حرکت کند.

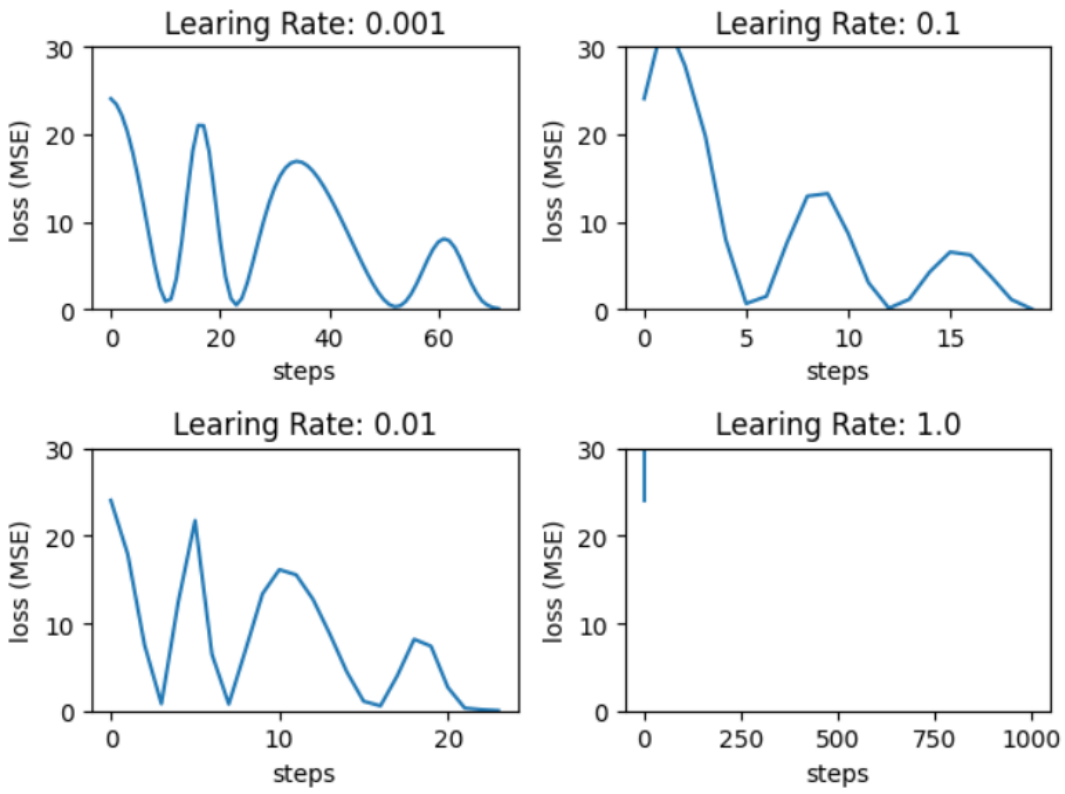
در نهایت بلوک ها را ران و نتایج را بررسی میکنیم:



در یادگیری توسط الگوریتم SGD با لرنینگ ریت های مختلف میبینیم که

- در صورتی که lr =1، به خاطر زیاد بودن لرنینگ ریت، الگوریتم دچار نوسان های خیلی زیاد شده و به جای حرکت به سمت نقطه minimum، از آن فاصله گرفته و loss به سرعت زیاد ترمیشود. در نهایت پس از 1000 مرحله هم مقدار loss بهبودی نمیکند چون لرنینگ ریت خیلی زیاد است.
- در صورتی که lr = 0.01، میبینیم که یادگیری به درستی انجام میشود. مدل نوسانی ندارد و به مرور یادگیری انجام میشود و پس از 150 مرحله، loss به کم تر از 0.1 میرسد.
- در صورتی که lr = 0.001، میبینیم که باز هم مدل بدون نوسان یادگیری را انجام میدهد. اما این بار مقدار lr خیلی کم است و یادگیری را خیلی کند انجام میدهد. نمودار آن شبیه نمودار lr = 0.01 شده اما این بار، پس از 800 مرحله مقدار loss به کم تر از 0.1 رسیده و میبینیم که با شیب کم تری کاهش پیدا کدره
- در صورتی که lr = 0.1، میبینیم که طی فقط 15 مرحله مقدار loss به کم تر از 0.1 رسیده که در مقایسه با 150 مرحله در lr 0.01 = خیلی بهتر است. البته در ابتدای آموزش یک نوسان وجود دارد که میتواند به خاطر lr بالا یا به خاطر stochastic بودن الگوریتم باشد. برای اطمینان میتوان لرنینگ ریت را کمی کم تر کرد. مثلا 0.05 و نتایج را بررسی کرد...

[↓]



در یادگیری توسط الگوریتم momentum با آلفا های مختلف میبینیم که:

- در صورتی که lr = 1، مانند الگوریتم SGD به خاطر خیلی زیاد بودن لرنینگ ریت، مدل دچار جهش های بسیار بزرگ شده و از نقطه مینیمم دور میشود و loss آن هیچگاه به کم تر از 0.1 نمی رسد
- در صورتی که lr = 0.001، میبنییم که یادگیری به خوبی انجام شده و پس از 60 مرحله، loss به کمتر از 0.1 رسیده است.

- در صورتی که lr = 0.01، باز هم میبینیم که loss در 20 مرحله به کمتر از 0.1 رسیده است که بهتر از lr = 0.001 است. زیرا مقدار lr بیشتر شده و یادگیری سریع تر انجام میشود. نکته دیگر این است که نمودار lr = 0.01 کمی شارپ تر از lr = 0.001 میباشد که این هم به خاطر نرخ یادگیری بالاتر است. مدل حرکات سریع تری میکند و کند نیست.
- در صورتی که lr = 0.1، باز هم میبینیم که یادگیری سریع تر شده و در 15 مرحله انجام میشود. همچنین نمودار باز هم شارپ تر شده که به خاطر lr بیشتر است.

در مقایه بین SGD و Momentum میبینیم که:

Momentum بسیار سریع تر از SGD همگرا شده است. این تفاوت به خاطر ماهیت momentum است که در مواقع لازم سرعت میگیرد و در مواقع دیگر کند میشود.

SGD با رسیدن به نقطه مینیمم از آن دور نمیشود اما momentum، دور 0 نوسان میکند. علت این نوسان هم همان وجود مفهومی شبیه سرعت در الگوریتم momentum است. در واقع مدل مانند یک گوی درون چاله نوسان میکند تا آرام آرام به نقطه صفر برسد.

در momentum، تغییر دادن آلفا از 0.001 تا 0.1 میتواند سرعت همگرایی را از60 به 15 برساند اما در SGD از 800 به 15 میرسد که دامنه خیلی بزرگتری است. علت این است که در SGD با چند برابر کردن آلفا، سرعت یادگیری هم چند برابر میشود و به همین علت این تغییرات شدید را میبینیم. اما در momentum، در صورتی که مدل در یک جهت ثابت حرکت کند، با حداکثر سرعت پیش میرود. فرقی نمیکند که آلفا 0.1 باشد یا 0.01 یا 0.001. تفاوتی که آلفا های مختلف در momentum ایجاد میکنند تنها سرعت تغییر جهت مدل است. هر چه آلفا بیشتر باشد، مدل سریع تر تغییر جهت میدهد همانطورکه در نمودار میبینیم که نوسان ها به طور شدید تر انجام شده. بنابراین با تغییر آلفا از 0.001 به 0.1، تنها سرعت مدل را در نوسانات بیشتر میکنیم و سرعت مدل در حرکات تغییر چندانی نمیکند.

Q5)

الف)

فرایند خواسته شده در نوتبوک موجود در فایل زیپ اعمال شده.

توضیح کد ها:

در ابتدا از یک مدل ساده استفاده کردم که تنها یک لایه پنهان با 100 نورون دارد:

```
## Define the model
################ Your code ############
model = nn.Sequential(
  nn.Flatten(),
  nn.Linear(input_size, 100),
  nn.ReLU(),
  nn.Linear(100, out_size),
)
#######################################
```

در اولین لایه از flatten استفاده کردم زیرا نمیخواهیم از لایه های کانولوشنی استفاده کنیم. برای تالع فعالسازی لایه میانی هم از relu استفاده کدرم.

اپتیمایزر و تابع لاس هم به شکل خواسته شده تعیین کردم:

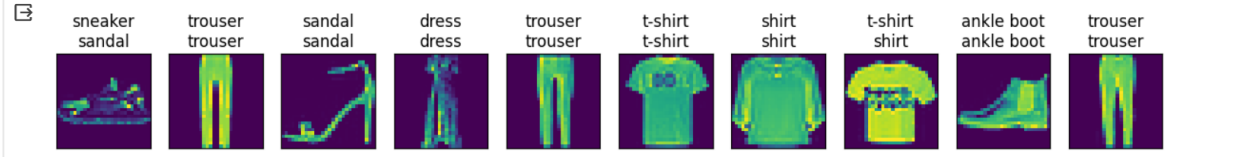
```
[25] ################ Your code ############
     criterion = nn.CrossEntropyLoss()
     optimizer = optim.SGD(model.parameters(), lr=0.01)
     #######################################
```

خلاصه مدل:

```
Sequential(
  (0): Flatten(start_dim=1, end_dim=-1)
  (1): Linear(in_features=784, out_features=100, bias=True)
  (2): ReLU()
  (3): Linear(in_features=100, out_features=10, bias=True)
)
```

تست مدل را هم به کمک D2L انجام دادم. نتایج به این صورت بود:

```
## Test your model
# from d2l import torch as d2l
d2l.predict_ch3(model,testloader,n = 10)
```



ب)

در این قسمت لازم است که بلوک کد train را تغییر دهیم. زیرا برای تشخیص overfitting نیاز به این داریم که

- تست مدل را هم حین فرایند ترین انجام دهیم نه پس از آن
- از نتایج مدل در حین ترین و تست، هیستوری تهیه کنیم

بنابراین یک بلوک جدید کد برای ترین کردم نوشتم و از بلوک های قبلی به جز train loader و test loader دیگر استفاده ای نکردم.

توضیح کدهای نوشته شده:

مدل جدیدی که ایجاد کردم را به این صورت انتخاب کردم

```
model = nn.Sequential(
  nn.Flatten(),
  nn.Linear(input_size, 128),
  nn.ReLU(),
  nn.Linear(128, 64),
  nn.ReLU(),
  nn.Linear(64, 32),
  nn.ReLU(),
  nn.Linear(32, 16),
  nn.ReLU(),
  nn.Linear(16, out_size),
)
```

این کار برای این بود که سعی کردم مدل لایه های زیادی داشته باشد تا انعطاف پذیری زیادی داشته باشد و همچنین تعداد نورونهای هر لایه زیاد نباشد تا مدل زود overfit شود.

از تابع ترین موجود در فایل نوتبوک حل تمرین پایتورچ استفاده کردم و هیستوری ترین را در این دیکشنری ذخیره کردم:

```
history = dict()
history['train_loss'] = list()
history['train_acc'] = list()
history['val_loss'] = list()
history['val_acc'] = list()
```

و فرایند test را هم در هر ایپاک انجام دادم تا با train قابل مقایسه باشد:

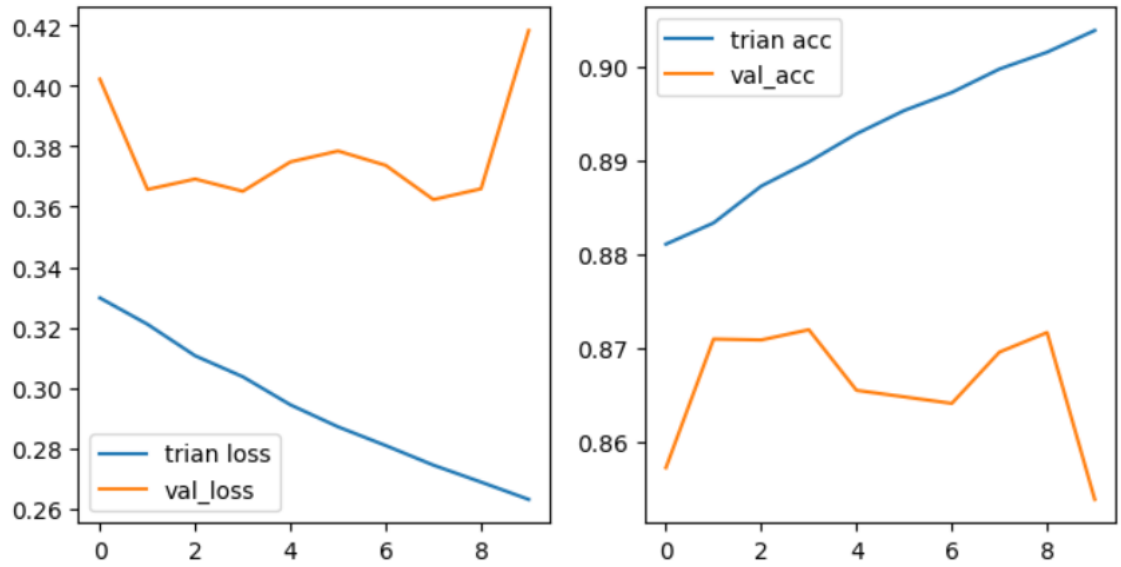
```
if epoch % val_per_epoch == val_per_epoch - 1:
  running_loss = 0.0
  running_acc = 0
  model.train()
  with torch.no_grad():
    for idx, (x, y) in enumerate(tqdm(testloader)):
      yhat = model(x)
      loss = criterion(yhat, y)

      running_loss += loss.item()
      running_acc += accuracy(yhat, y)

    running_loss /= len(testloader)
    running_acc /= len(testloader)
    history['val_loss'].append(running_loss)
    history['val_acc'].append(running_acc)
    print(f"epoch = {epoch}\tvalidation loss = {running_loss}\tvalidation accuracy = {running_acc}")
```

نتایج به این صورت شدند:

```
<matplotlib.legend.Legend at 0x7f36e0dfe2f0>
```



میبنییم که دقت مدل در train همواره رو به افزایش است اما در test، دقت در اواخر کاهش پیدا میکند که نشان دهنده overfit شدن مدل است.

علت اورفیت شدن این است که مدل دیگر نمیتواند بیشتر از این ویژگی های معنا دار از تصویر پیدا کند که روی داده test هم قابل استناد باشد. بنابراین به جای ویژگی های معنا دار، فیچرهایی را حفظ کرده که فقط برای train قابل استناد هستند.

پ)

برای حل این قسمت از این لینک کمک گرفتم:

[MNIST Digits - PyTorch CNN, ~99% | Kaggle]

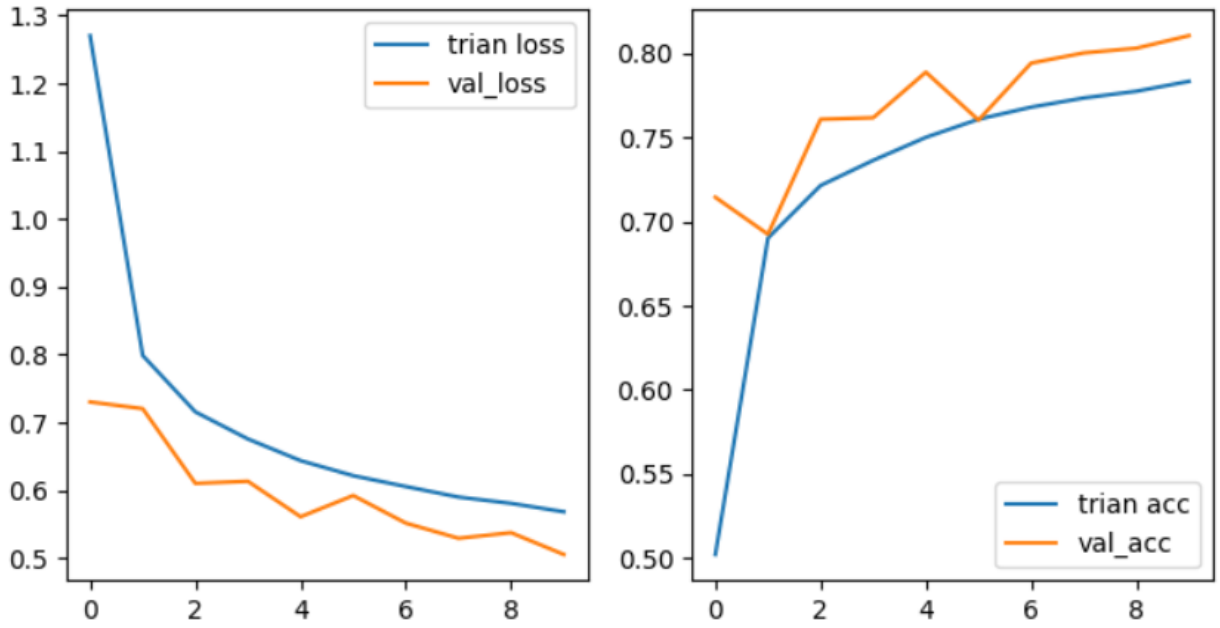
در اینجا از rotation، affine و افزودن نویز به عنوان DataAugmentation استفاده کردم. برای استفاده از این ها کافی است که loader را طوری تغییر دهیم که قبل از داده داده ها، آنها را agment کند:

```
train_transform = transforms.Compose(
                    [
                    # transforms.ToPILImage(),
                    transforms.RandomRotation(30),
                    transforms.RandomAffine(degrees=20, translate=(0.1,0.1), scale=(0.9, 1.1)),
                    transforms.ColorJitter(brightness=0.2, contrast=0.2),
                    transforms.ToTensor(),
                    transforms.Normalize((0.1307,), (0.3081,)),
                    ])
```

اما برای داده های تست، از augmentation استفاده نکردم تا نتایج واقعی مدل را ببینیم.

```
test_transform = transforms.Compose(
                    [
                    # transforms.ToPILImage(),
                    transforms.ToTensor(),
                    transforms.Normalize((0.1307,), (0.3081,)),
                    ])
```

مدل را اجرا کرده و نتایج را رسم میکنیم:



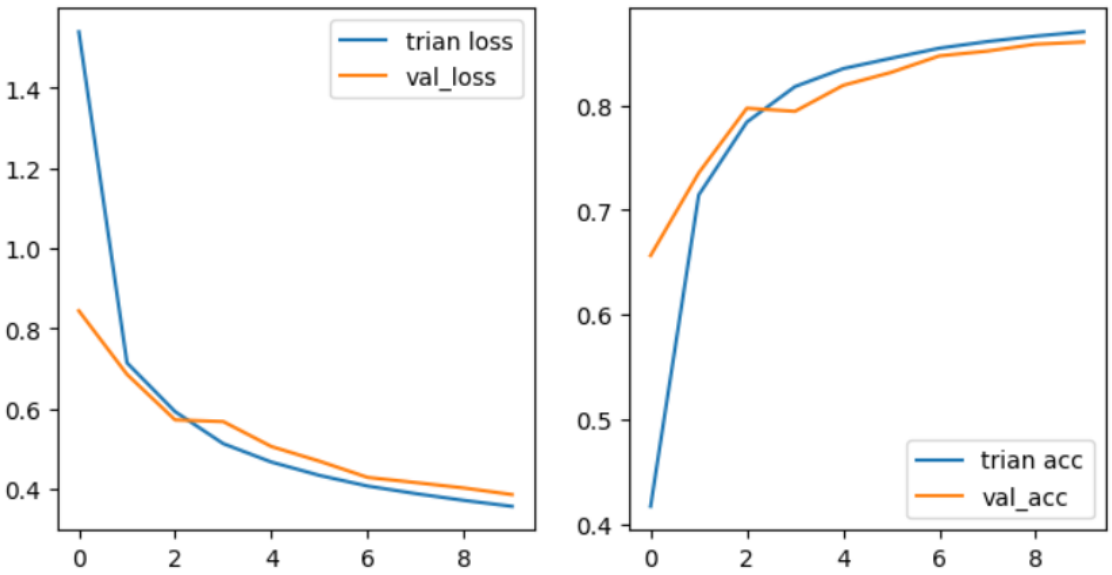
میبینیم که این بار، دقت test از train بشتر شده. علت این است که داده های test ساده و بدون تغییر هتسند اما داده های train پیچیده تر هستند که پیشبینی صحیح را سخت تر میکند. متاسفانه مدل نتوانسته دقتی بهتر از دفعه قبل را کسب کند چون مدل انعطاف پذیری زیادی ندارد و نتوانسته الگوهای تصاویر augment شده را پیدا کند. برای بالاتر بردن دقت حالا باید دوباره روی قدرتمند تر کردن مدل کار کنیم و سپس دوباره data augmentation انجام دهیم. البته اگر این مدل را چند epoch دیگر هم اجرا کنیم ممکن است دقتی بهتر از بدون data augmentation به دست بیاورد اما چیزی که واضح است این است که مدل اصلا overfit نشده است.

ت)

در این قسمت برای ساده سازی از ساده سازی L2 استفاده کردم. برای اینکار کافیست تا یک پارامتر wight decay به optimizer اضافه کنیم:

```
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=0.01, weight_decay=1e-3)
n_epochs = 10
val_per_epoch = 1
```

مجددا مدل را مثل قبل ایجاد و ترین کردم. نتایج به این صورت بود:

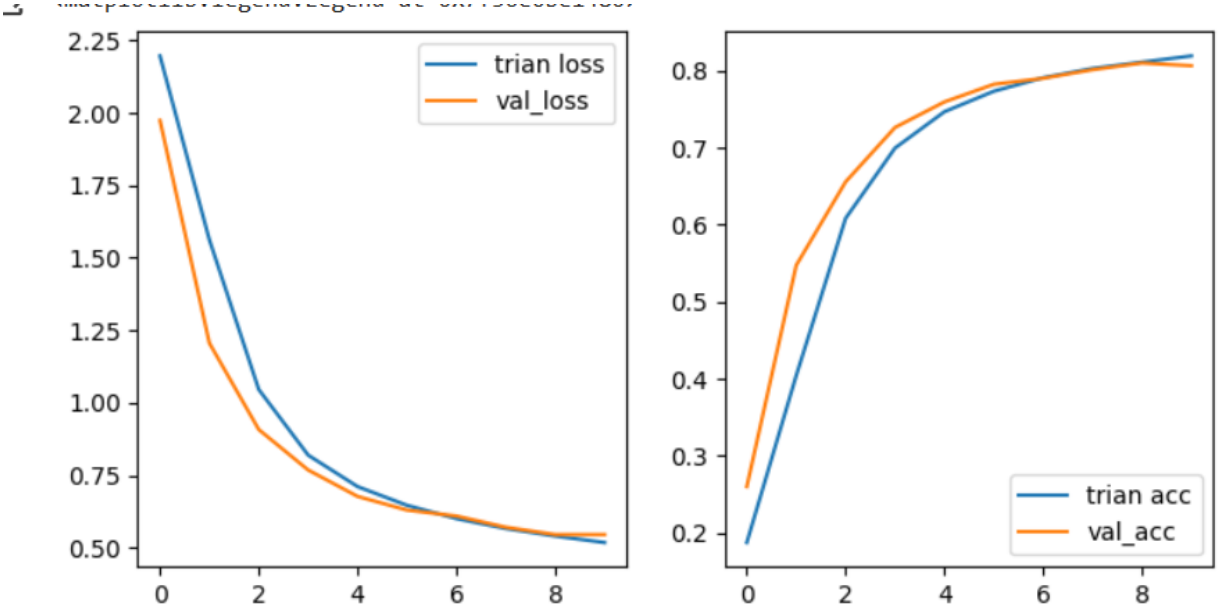


میبینیم که دقت مدل در تست سه درصد بهتر شده و مشکل overfitting هم بهتر شده. (دقت تست به 86 و Train به 87 درصد رسیده)

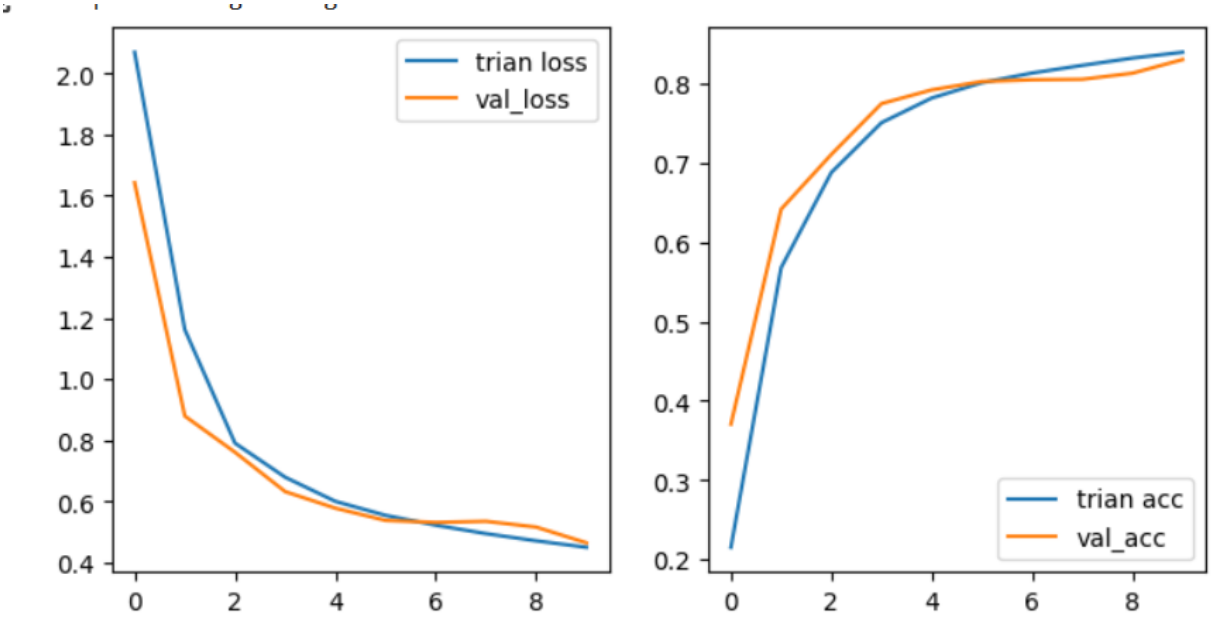
ث)

در این قسمت برای تست ترکیب این موارد، هر کدام از آنها را دو به دو با هم امتحان کردم. نتایج به این صورت بودند:

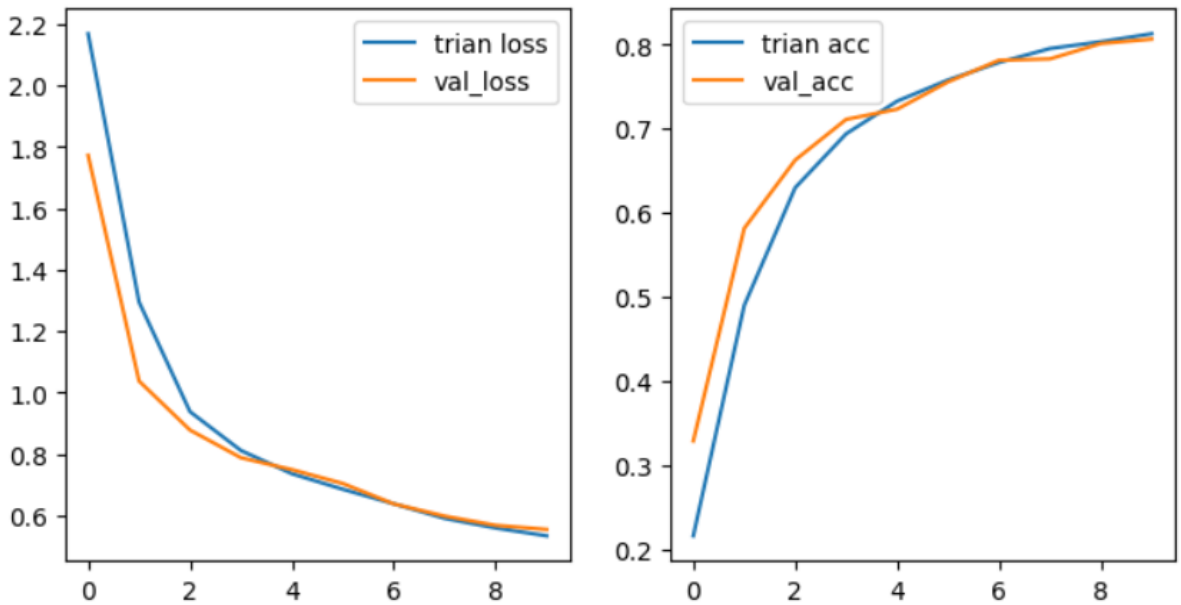
Data augmentation and dropout:



Data augmentation and Regularization:



:Regularization and Dropout



میبینیم که ترکیب Data augmentation و Regularization نتیجه بهتری نسبت به دو مورد دیگر داشت. دقت این ترکیب 83 درصد برای تست و ترین بود اما دو ترکیب دیگر دقت 80 و 81 درصد برای تست و ترین داشتند.

البته ترکیب این موارد بستگی به مسئله دارد و نمیتوان گفت همواره این ترکیب بهترین نتیجه را میدهد.

در نهایت بهتر بود که کد تمیز تری مینوشتم. میتوانستم بلوک کد Train را یک تابع کنم تا تمیزتر باشد اما متاسفانه وقت نکردم.