

به نام خدا



دانشگاه تهران
پردیس دانشکده‌های فنی
دانشکده برق و کامپیوتر



درس سیستم‌های هوشمند

تمرین شماره 3

نام و نام خانوادگی : شایان واصف

شماره دانشجویی : 810197603

مهر 1400

سر فصل مطالب

سوال 1 : کاربرد شبکه های عصبی پیچشی در طبقه بندی (امتیازی) 3

الف : توضیحات مدل شبکه پیچشی 3

ب : تاثیر لایه مخفی 7

ج : تاثیر تابع فعال ساز 11

د : تاثیر بهینه ساز 14

ه : تاثیر حذف تصادفی 18

سوال 2 : شبکه عصبی (پرسپترون با چند لایه مخفی) 21

الف : تحلیلی 21

ب : تحقیق 23

ج : پیاده سازی شبکه پرسپترون در کاربرد رگرسیون 27

سوال 1 :

الف) توضیحات مدل شبکه پیچشی

برای افزایش سرعت آموزش مدل از GPU Colab استفاده می‌کنیم و با دستور زیر می‌توانیم بین CPU و GPU تعویض کنیم به طوریکه در هنگام آموزش مدل و محاسبه وزن های مطلوب بر روی GPU باشد و در هنگام محاسبه Accuracy و plot کردن ، دوباره دادگان را به CPU انتقال دهیم :

```
# use GPU if available
device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')
```

```
# send the model to the GPU
net.to(device)
```

```
# push data to GPU
x = x.to(device)
y = y.to(device)
```

شکل 1-1-1 : انتقال مدل و داده های آموزش و تست به GPU

در ادامه به کمک کتابخانه Torch vision ، دیتاست موردنظر را برای مجموعه آموزش و تست (10000 داده را به دادگان تست اختصاص می‌دهد) به طور جدا دانلود می‌کنیم و سپس 6000 داده را به صورت رندوم به دادگان ارزیابی و 4000 داده باقی مانده را به دادگان تست اختصاص می‌دهیم. و در نهایت به کمک کتابخانه Data Loader ، داده های آموزش (ورودی به مدل به همراه لیبل ها) را با تعداد دسته 32 (Batch size=32) تبدیل می‌کنیم و همچنین برای دادگان تست ، تعداد دسته را برابر طول دادگان تست در نظر می‌گیریم (برای دادگان تست Batch نداریم) . خوبی کتابخانه Torch vision در این است که به کمک آن می‌توان داده ها را Transform کنیم به این معنی که شکل کلی داده های اصلی را تغییر دهیم (Gray کردن ، در جهت چپ یا راست عکس را 90 درجه بچرخانیم (Horizontal Flip) ، نرمال کردن و ...) .

در اینجا ما دادگان را در هر بعد بر روی میانگین 0.5 و واریانس 0.5 نرمال می‌کنیم . کد مربوط به توضیحات بالا به صورت زیر می‌باشد :

```
# transformations
transform = T.Compose([ T.ToTensor(),
                        T.Normalize([.5,.5,.5],[.5,.5,.5])
                      ])

# import the data and simultaneously apply the transform
trainset = torchvision.datasets.CIFAR10(root='./data', train=True, download=True, transform=transform)
devtest = torchvision.datasets.CIFAR10(root='./data', train=False, download=True, transform=transform)

# split the devtest into two separate sets
randidx = np.random.permutation(10000) # a random permutation of indices
devset = Subset(devtest, randidx[:6000]) # Subset for devset
testset = Subset(devtest, randidx[6000:]) # Subset for test

# transform to dataloaders
batchsize = 32
train_loader = DataLoader(trainset, batch_size=batchsize, shuffle=True, drop_last=True)
dev_loader = DataLoader(devset, batch_size=batchsize) # note: devtest in batches!
test_loader = DataLoader(testset, batch_size=len(testset))
```

شکل 1-1-2 : شکل دادن دادگان آموزش ، تست و ارزیابی

بعد از اعمال مراحل ذکر شده ، سایز یک Batch از داده های آموزش و مینیم و ماکسیم مقدار موجود در آن را چک می کنیم :

```
#Check out
X,y = next(iter(train_loader))

# shape of one batch of train_set
print('Data shapes (train/test):')
print( X.data.shape )

#range of pixel intensity values
print('\nData value range:')
print( (torch.min(X.data),torch.max(X.data)) )

Data shapes (train/test):
torch.Size([32, 3, 32, 32])

Data value range:
(tensor(-1.), tensor(1.))
```

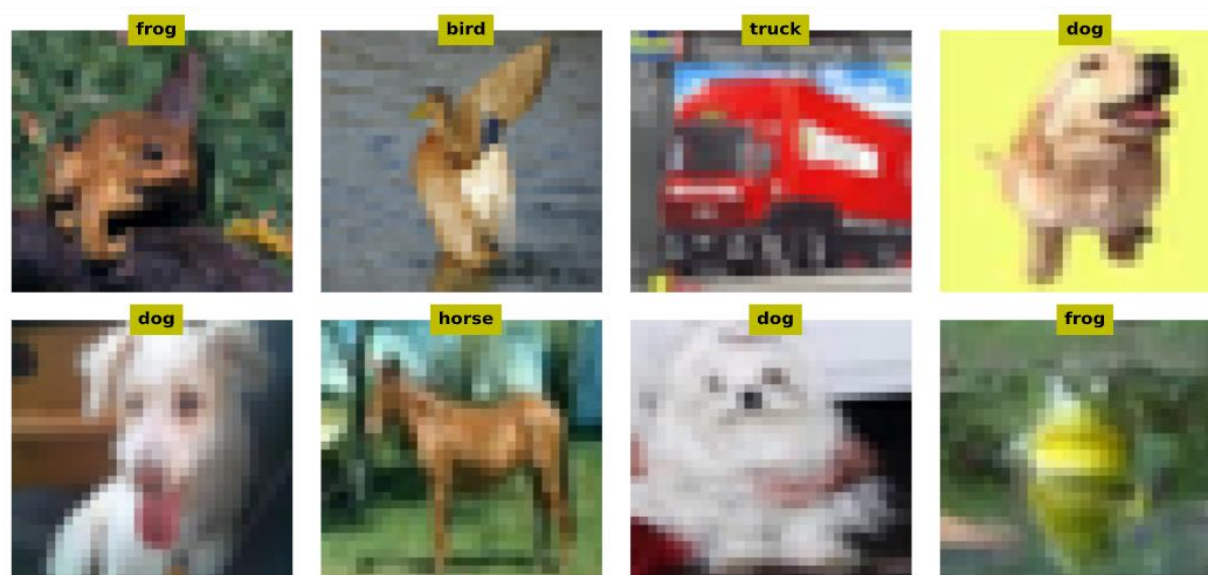
شکل 1-1-3 : چک کردن نتایج حاصل از پیش پردازش داده ها

که در بالا ، عدد 32 به تعداد Sample در یک Batch و عدد 3 ، به RGB بودن عکس اشاره دارد. همچنین بعد از اعمال Normalization ، تمامی مقادیر دادگان آموزش بین -1 و 1 قرار می گیرند. در دیتاست CIFAR10 در مجموع 10 Category زیر وجود دارد :

Data categories:
['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck']

شکل 1-1-4 : 10 کلاس موجود در دیتاست CIFAR10

به طور رندوم ، 16 عکس از دادگان موجود در مجموعه آموزش را در زیر به همراه لیبل آنها می بینیم :





شکل 1-5-1: نمایی رندوم از تعدادی از دادگان آموزش به همراه لیبل آنها

به طور کلی سه نوع لایه مختلف در CNN وجود دارد :

➤ Convolution: وظیفه این لایه ، یادگیری فیلترها (Kernels) می باشد تا Feature Map ها را ایجاد کند.

➤ Pooling: وظیفه این لایه ، کاهش بعد و افزایش سایز Receptive Fields می باشد (حداکثر

تعداد پیکسلی که شبکه می تواند در هر مرحله در اختیار داشته باشد)

➤ Fully Connected: در نهایت پیش بینی توسط این لایه انجام می شود.

در معماری اولیه مدل ، برای هر لایه کانولوشنی ، لایه Pooling و Batch Normalization اضافه می کنیم به طوریکه در لایه اول کانولوشنی به ترتیب 3 ورودی (RGB) ، 64 کانال (64 Feature maps) ، کرنل با سایز 3*3 و یک Padding و یک لایه Pooling با سایز 2 در نظر می گیریم. سپس برای سایر لایه ها (تعداد لایه های مطلوب در بخش بعد مشخص می شود) ، یک لایه Pooling با سایز 2 ، تعداد ورودی برابر تعداد کانال لایه قبل ، تعداد کانال جدید، دو برابر تعداد کانال شبکه قبل و همچنان کرنل با سایز 3*3 در نظر می گیریم . بعد از اتمام ساختار کانولوشنی ، به ساختار Fully Connected می رسیم که در اینجا به تعداد 3 لایه مخفی در نظر می گیریم و در نهایت در خروجی 10 نوروں به منظور Classification خواهیم داشت .

چالش چیدن لایه های Fully Connected در تعیین تعداد نوروں های ورودی می باشد . زیرا در هر مرحله قرار دادن لایه کانولوشنی ، سایز خروجی (Resolution تصویر) کاهش می یابد و ورودی اولین لایه Fully Connected باید به تعداد $Output\ size(last\ conv\ layer)^2 * Nchannels(last\ conv\ layer)$ باشد که

در آن $Output_size(last\ conv\ layer)$ ، تعداد پیکسل های موجود در هر سطر/ستون feature map می باشد که از آنجا که feature map ها مربعی هستند ، تعداد پیکسل موجود در هر کدام از آنها برابر $Output_size(last\ conv\ layer)^2$ می باشد .

تعداد پیکسل های موجود در هر سطر feature map در یک لایه دلخواه به صورت زیر می باشد :

$$N_h = \left\lfloor \frac{(M_h + 2p - k)}{s_h} \right\rfloor + 1$$

در صورتی که عکس ما مربعی باشد ، آنگاه N_h تعداد پیکسل بدست آمده در جهت عمودی/افقی در لایه حاضر ، M_h تعداد پیکسل موجود در جهت عمودی/افقی در لایه پیشین ، p مقدار Padding و k تعداد پیکسل موجود در کرنل در جهت عمودی/افقی (فرض کنیم که کرنل مربعی باشد) و s_h مربوط به Stride می باشد که به صورت Default برابر 1 در نظر گرفته می شود.

خلاصه ای از معماری مدل ذکر شده در بالا ، برای سه لایه کانولوشنی به صورت زیر می باشد :

```
Input: [32, 3, 32, 32]
First CPBR block: [32, 64, 16, 16]
Second CPR block: [32, 128, 7, 7]
Second CPR block: [32, 256, 2, 2]
Vectorized: [32, 1024]
Final output: [32, 10]
```

```
Output size:
torch.Size([32, 10])
```

Layer (type)	Output Shape	Param #
Conv2d-1	[32, 64, 32, 32]	1,792
BatchNorm2d-2	[32, 64, 16, 16]	128
Conv2d-3	[32, 128, 14, 14]	73,856
BatchNorm2d-4	[32, 128, 7, 7]	256
Conv2d-5	[32, 256, 5, 5]	295,168
BatchNorm2d-6	[32, 256, 2, 2]	512
Linear-7	[32, 256]	262,400
Linear-8	[32, 64]	16,448
Linear-9	[32, 10]	650

```
=====
Total params: 651,210
Trainable params: 651,210
Non-trainable params: 0
```

```
-----
Input size (MB): 0.38
Forward/backward pass size (MB): 29.55
Params size (MB): 2.48
Estimated Total Size (MB): 32.41
-----
```

شکل 6-1-1: خلاصه معماری و تعداد پارامتر مدل

طبق شکل بالا ، تعداد پیکسل در هر سطر/ستون feature map ها برای لایه اول، 16 ، لایه دوم، 7 و لایه سوم برابر 2 می باشد :

$$32(\text{input pixel in each row/column}) \rightarrow 16 \rightarrow 7 \rightarrow 2$$

بنابراین ما کسیمم تا 3 لایه کانولوشنی با معماری فرض شده (تعداد 3 پیکسل در هر راستای کرنل و padding=0 و stride=1) برای سوال می توان در نظر گرفت و برای لایه چهارم به صفر پیکسل در هر feature map می رسیم که شدنی نیست .

(ب) تاثیر لایه مخفی

طبق نکات ذکر شده در بخش قبل ، برای تعداد 1، 2 و 3 لایه کانولوشنی ، مدل را آموزش می دهیم. soft code مربوط به کلاس CNN در قسمت __init__ به صورت زیر می باشد:

```
def __init__(self, nLayers, printtoggle):
    super().__init__()
    #Creat a dictionary
    self.conv = nn.ModuleDict()
    self.bnorms = nn.ModuleDict()
    self.nLayers = nLayers
    # print toggle
    self.print = printtoggle

    ### ----- feature map layers ----- ###
    # first convolution layer
    self.conv1 = nn.Conv2d(3, 64, 3, padding=1)
    self.bnorm1 = nn.BatchNorm2d(64) # input the number of channels in this layer
    output_size = 16 # (32+2*1-3)/1 + 1 = 32/2 = 16 (/2 b/c maxpool)
    if nLayers >= 2:
        for i in range(2, nLayers+1):
            self.conv[f'conv{i}'] = nn.Conv2d(64*(i-1), 128*(i-1), 3)
            self.bnorms[f'batchnorm{i}'] = nn.BatchNorm2d(128*(i-1))
            output_size = np.round(((output_size-3)+1)/2).astype('int') # ((num_previous_pixels-3)/1 + 1)/2 (/2 b/c maxpool)
    ### ----- linear decision layers ----- ###
    self.fc1 = nn.Linear(output_size*output_size*128*(nLayers-1), 128*(nLayers-1))
    self.fc2 = nn.Linear(128*(nLayers-1), 64)
    self.fc3 = nn.Linear(64, 10)
    else :
        self.fc1 = nn.Linear(output_size*output_size*64, 64)
        self.fc2 = nn.Linear(64, 64)
        self.fc3 = nn.Linear(64, 10)
```

شکل 1-2-1 : نحوه تعریف ساختار کلی لایه ها به صورت پارامتری

همچنین soft code مربوط به کلاس CNN در قسمت forward به صورت زیر می باشد :

```
def forward(self,x):

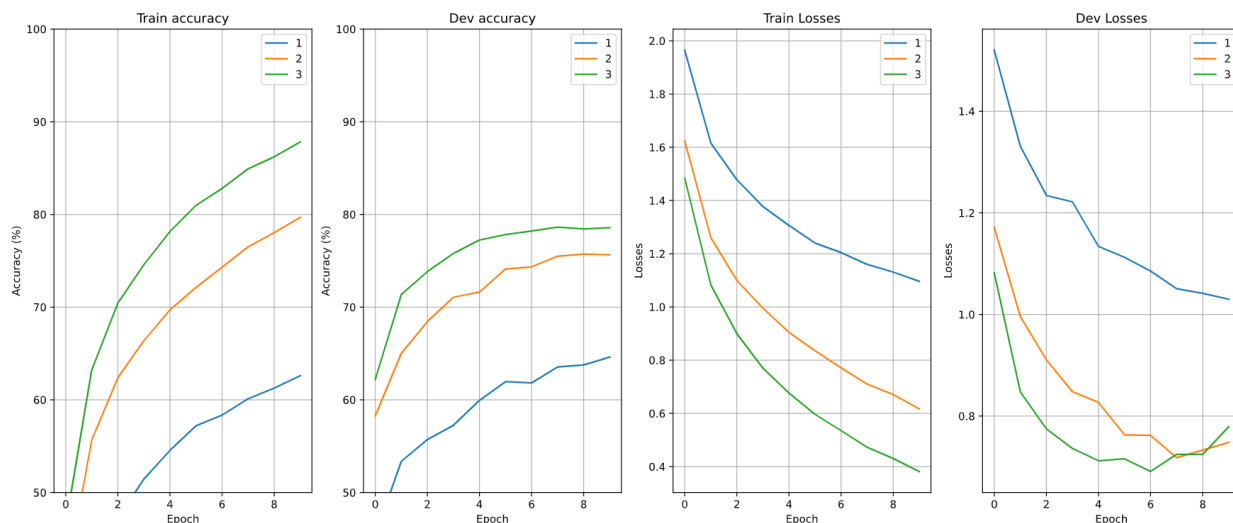
    if self.print: print(f'Input: {list(x.shape)}')

    # first block: convolution -> maxpool -> batchnorm -> relu
    x = F.max_pool2d(self.conv1(x),2)
    x = F.leaky_relu(self.bnorm1(x))
    if self.print: print(f'First CPBR block: {list(x.shape)}')
    #Setting blocks : convolution -> maxpool -> batchnorm -> relu
    for j in range(2,self.nLayers+1):
        x = F.max_pool2d(self.conv[f'conv{j}'](x),2)
        x = F.leaky_relu(self.bnorms[f'batchnorm{j}'](x))
        if self.print: print(f'Second CPR block: {list(x.shape)}')
    # reshape for linear layer
    nUnits = x.shape.numel()/x.shape[0]
    x = x.view(-1,int(nUnits))
    if self.print: print(f'Vectorized: {list(x.shape)}')
    # linear layers
    x = F.leaky_relu(self.fc1(x))
    x = F.dropout(x,p=.5,training=self.training)
    x = F.leaky_relu(self.fc2(x))
    x = F.dropout(x,p=.5,training=self.training) # training=self.training means to turn off during eval mode
    x = self.fc3(x)
    if self.print: print(f'Final output: {list(x.shape)}')

    return x
```

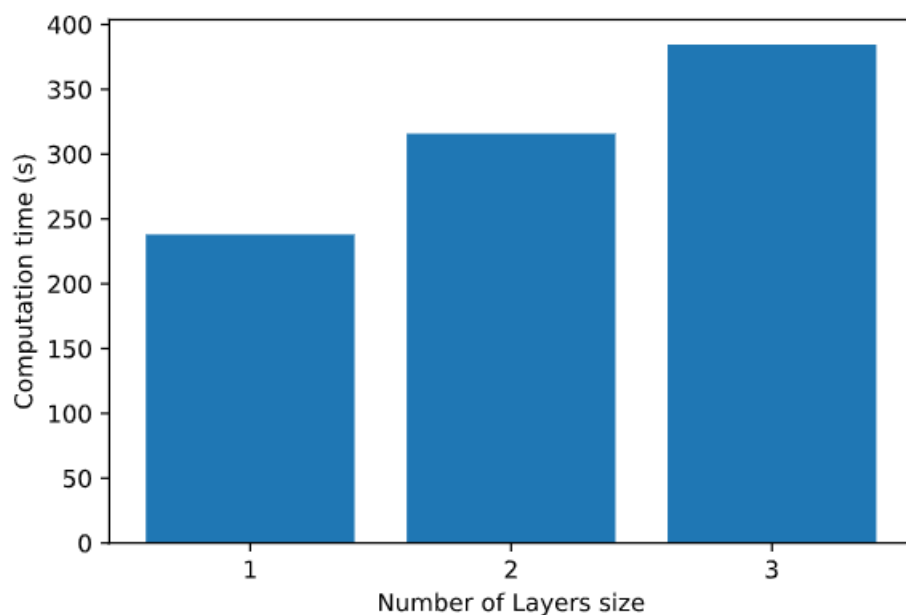
شکل 2-2-1 : نحوه soft code قسمت forwarding

در نهایت دقت و خطای بدست آمده را در مجموعه آموزش و ارزیابی به صورت جدا رسم می کنیم :



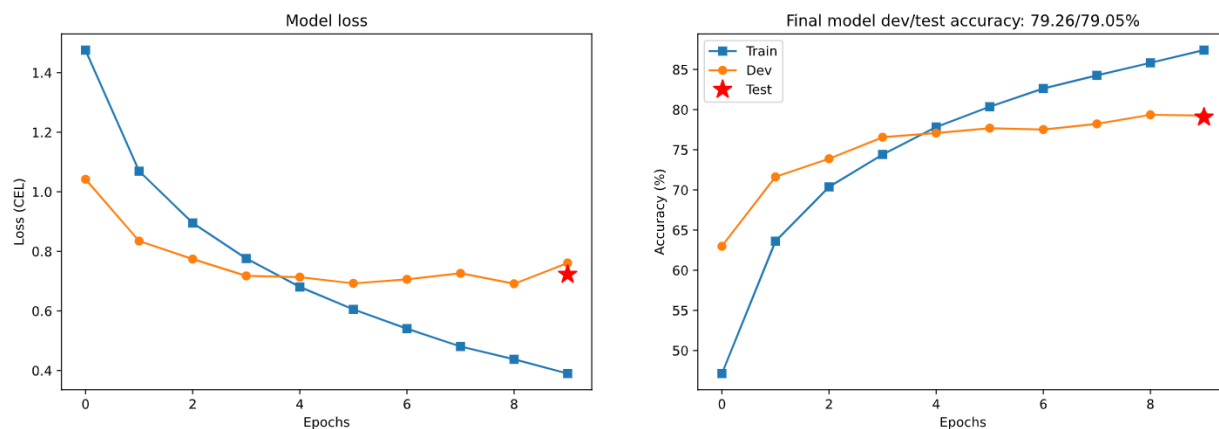
شکل 3-2-1 : دقت و خطای بدست آمده در مجموعه آموزش و ارزیابی برای هر کدام از لایه های ذکر شده

همچنین زمان محاسباتی لازم برای هر کدام از تعداد لایه ها در زیر بر حسب ثانیه آورده شده است :



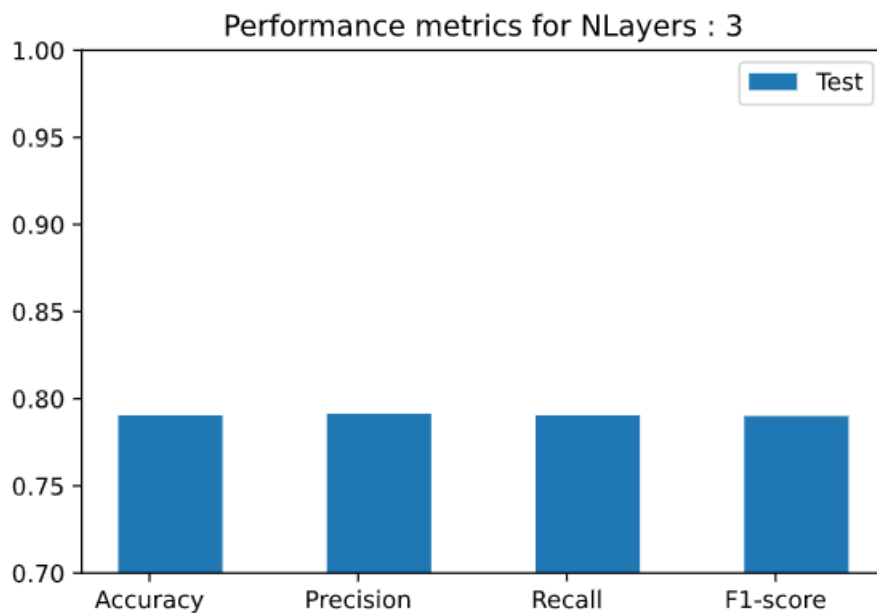
شکل 1-2-4: زمان محاسباتی برای تعداد لایه مختلف

طبق نتایج بالا، برای تعداد 3 لایه، بهترین دقت را در مجموعه ارزیابی (Dev set) داریم. بنابراین برای 3 لایه، عملکرد مدل را بر روی داده‌گان تست بررسی می‌کنیم:

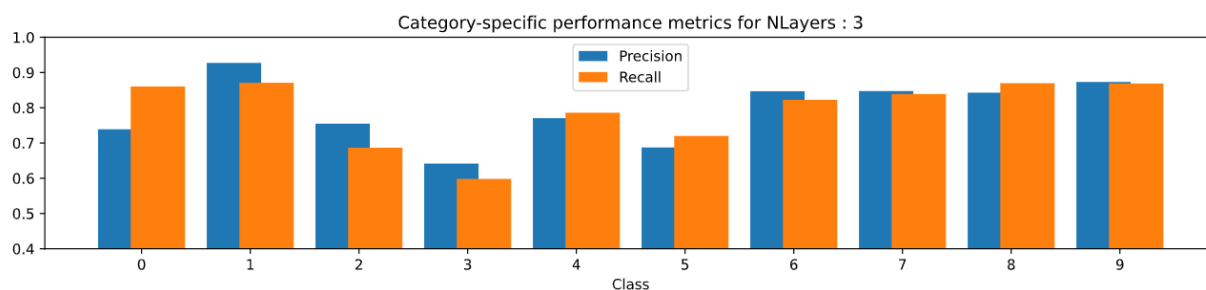


شکل 1-2-5: عملکرد مدل منتخب با 3 لایه بر روی داده‌گان تست

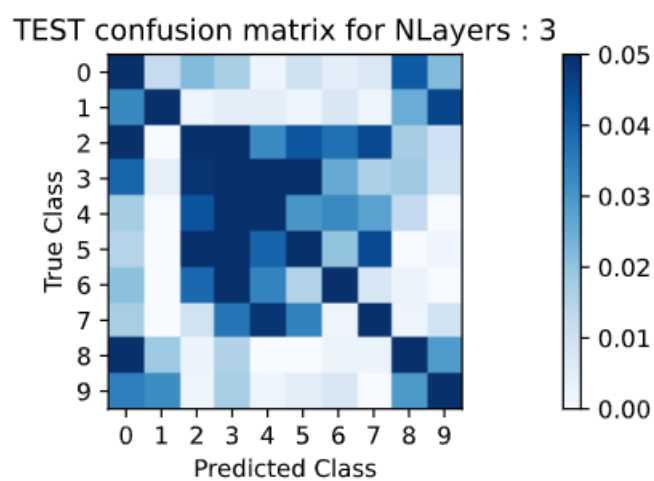
در ادامه از 4 متریک معروف Accuracy, Precision, Recall, F1-score برای تخمین بهتر مدل استفاده می‌کنیم:



شکل 6-2-1 : APRF برای 3 لایه کانولوشنی



شکل 7-2-1 : PR برای هر یک از 10 کلاس برای 3 لایه کانولوشنی



شکل 8-2-1 : ماتریس آشفتگی برای 3 لایه کانولوشنی

ج) تاثیر تابع فعال ساز

نحوه استفاده از توابع فعال ساز بسته به نوع لایه اهمیت دارد. نکته مهم این است که اگر در هر لایه از توابع خطی استفاده شود، آنگاه کل شبکه را میتوان به صورت ترکیب خطی از نورون های ورودی در نظر گرفت و در نتیجه تنها با یک لایه قابل نمایش خواهد بود و کارایی شبکه از بین می رود.

به طور کلی Activation Function در لایه مخفی باید ویژگی های زیر را داشته باشد :

➡ غیر خطی باشد . (Nonlinearity)

➡ از نظر محاسباتی ساده باشد. (چون قرار است بار ها صدا زده شود و اگر پیچیده باشد ، زمان محاسباتی به شدت بالا خواهد رفت)

➡ به بازه خاصی محدود نباشد

➡ تا حد امکان موجب Vanishing/Exploding Gradient نشود .

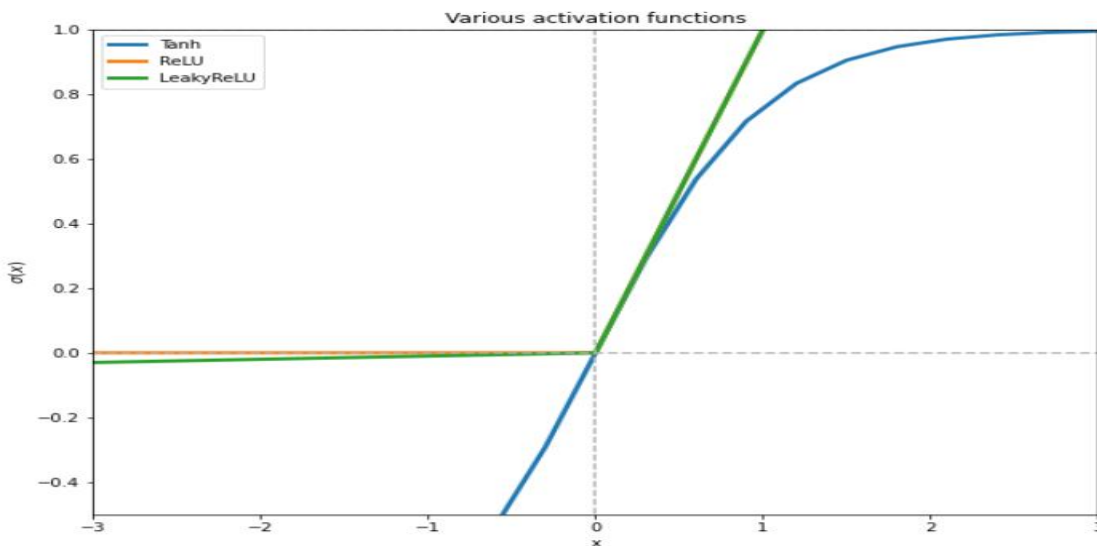
همچنین Activation Function در لایه خروجی باید ویژگی های زیر را داشته باشد :

➡ بسته به نوع مساله باید انتخاب شود . برای Regression از تابع خطی و برای Classification از تابع غیر خطی استفاده شود .

➡ در طبقه بندی می خواهیم خروجی به صورت احتمالی در بیاید. پس باید خروجی تابع انتخاب شده بین صفر و یک باشد .

حال به ترتیب سه Activation ، Tanh و ReLU و Leaky-ReLU را که در زیر آنها را plot کردیم ، مقایسه

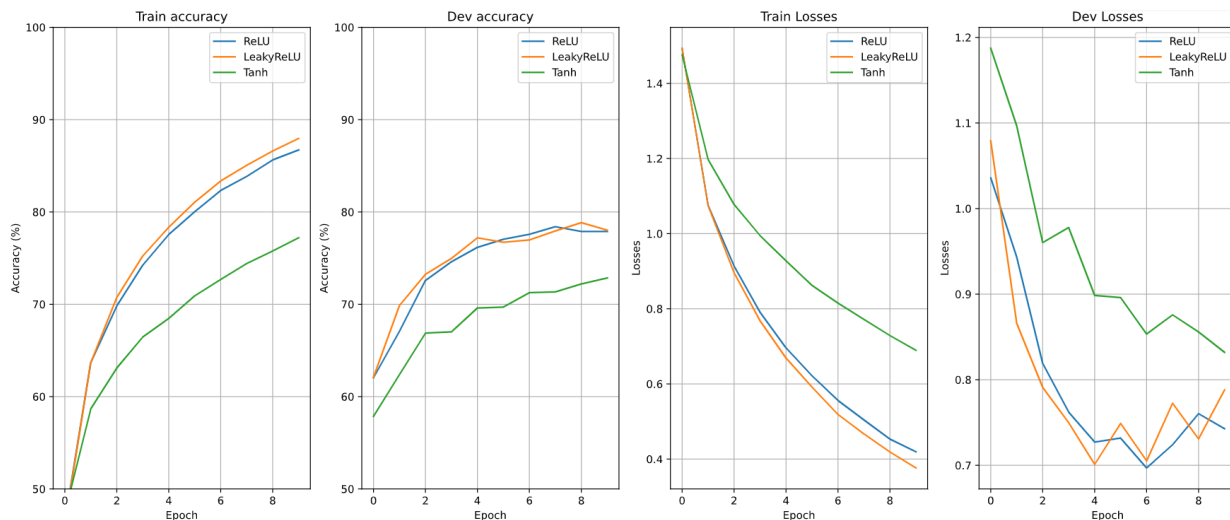
می کنیم :



شکل 1-3-1 : مقایسه بین Activation های مورد استفاده در سوال

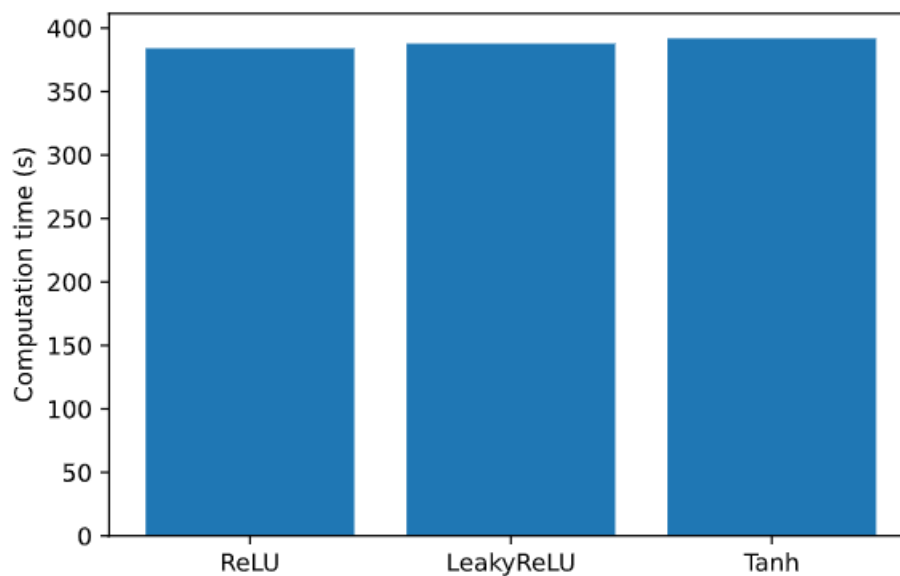
همانطور که مشخص است ، دو تابع ReLU و Leaky-ReLU بسیار مشابه هم هستند با این تفاوت که Leaky-ReLU ، یک شیب بسیار کم برای قسمت منفی در نظر می گیرد .

نتایج حاصل از آموزش مدل توسط این سه تابع فعال ساز را مقایسه می کنیم :



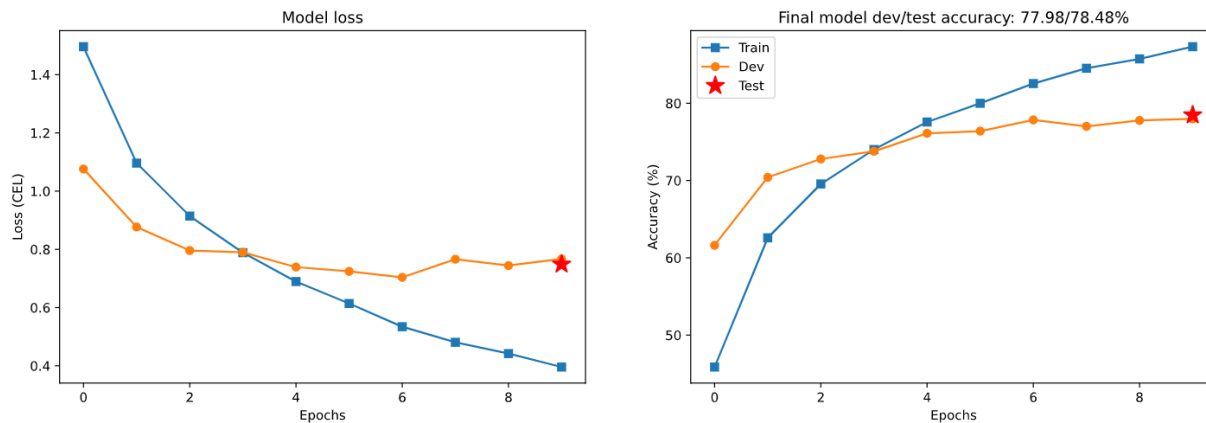
شکل 2-3-1: دقت و خطای بدست آمده در مجموعه آموزش و ارزیابی برای هر کدام از لایه های ذکر شده طبق نتایج بالا ، تابع Leaky-ReLU کمی بهتر تابع ReLU عمل کرده و هر دو نسبت به Tanh برتری دارند.

همچنین زمان محاسباتی مورد نیاز برای آموزش با هر کدام از توابع فعال ساز به صورت زیر می باشد :



شکل 3-3-1: زمان محاسباتی برای توابع فعال ساز مختلف

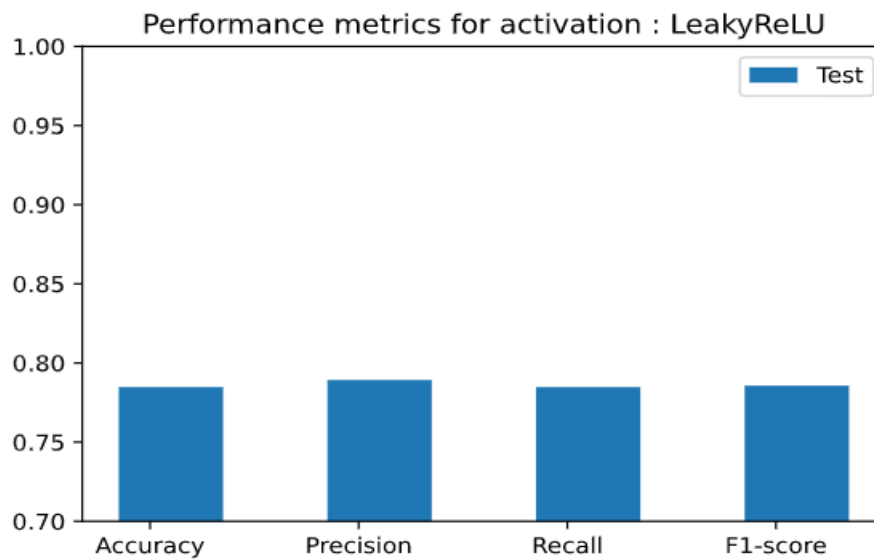
با انتخاب Leaky-ReLU به عنوان تابع فعال ساز ، دادگان تست را با مدل آموزش دیده ، تخمین می‌زنیم :



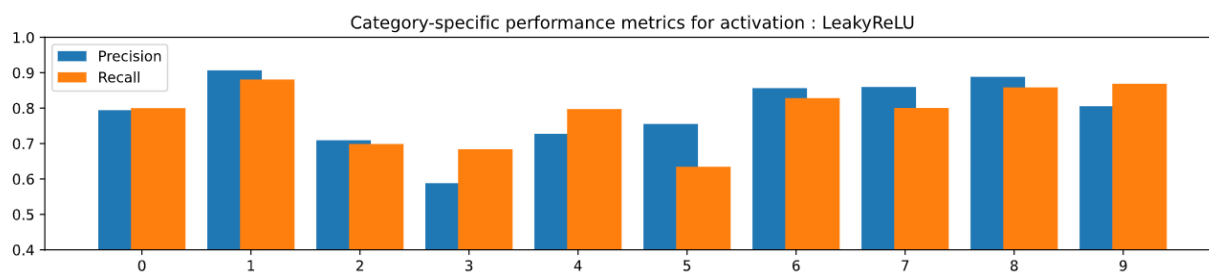
شکل 4-3-1 : عملکرد مدل منتخب با تابع فعال ساز Leaky-ReLU بر روی دادگان تست

در ادامه از 4 متریک معروف Accuracy , Precision , Recall , F1-score برای تخمین بهتر مدل استفاده

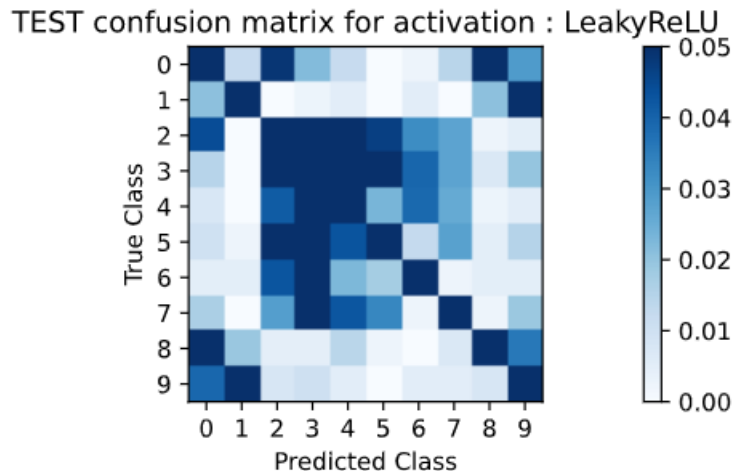
می‌کنیم :



شکل 5-3-1 : APRF برای تابع فعال ساز Leaky-ReLU



شکل 6-3-1 : PR برای هر یک از 10 کلاس برای تابع فعال ساز Leaky-ReLU



شکل 7-3-1: ماتریس آشفتگی برای تابع فعال ساز Leaky-ReLU

د) تاثیر بهینه ساز

ما در این قسمت ، دو Optimizer را مقایسه می کنیم :

SGD: در این روش ما هر بار ماتریس وزن ها را در جهت خلاف گرادیان تابع هزینه و اسکالر α

آپدیت می کنیم :

$$W_{new} = W_{old} - \eta d_L$$

نکته مهم در اینجا این است که ما برای هر Sample ، ماتریس W را آپدیت می کنیم ، پس در جایی که دادگان شباهت زیادی به یکدیگر دارند ، این روش می تواند بسیار خوب عمل کند . بنابراین یادگیری آن بسیار حساس است و در صورت وجود Outlier در داده ها می تواند ضعیف عمل کند. به همین دلیل ایده Mini batch SGD هم مطرح شد تا بتواند تا حدی این مشکل را حل کند. همچنین برای اینکه روند تغییرات نمودار SGD را کمی Smooth تر کنیم از Momentum استفاده می کنیم . در واقع در Momentum ، هر نقطه ترکیب وزن دار خودش با نقطه قبلی می باشد که به SGD with Momentum معروف است :

$$\begin{cases} v = (1 - \beta)d_L + \beta v_{t-1} \\ W - \eta v \rightarrow W \end{cases}$$

برای $\beta = 0$ ، به همان روش SGD می رسیم .


RMSprop (RMS → root – mean – square , prop → propagation) 

ایده RMSprop مشابه Momentum می‌باشد ، با این تفاوت که به جای bias کردن گرادیان ، Learning Rate را متناسب با اندازه گرادیان بایاس می‌کند :

$$\begin{cases} s = (1 - \beta)d_L^2 + \beta s_{t-1} \\ W - \frac{\eta d_L}{\sqrt{s + \epsilon}} \rightarrow W \end{cases}$$

که در آن ضریب ϵ یک مقدار بسیار کوچک می‌باشد که به مشکل خطای تقسیم بر صفر مواجه نشویم. در واقع RMSprop بیان می‌کند که اگر اندازه گرادیان زیاد باشد ، s زیاد می‌شود و در نتیجه Learning rate کاهش می‌یابد و بر عکس .

نکته مهم دیگر آن است که d_L یک ماتریس می‌باشد و در واقع RMSprop ، Learning rate را برای هر جهت (هر وزن) تنظیم می‌کند.

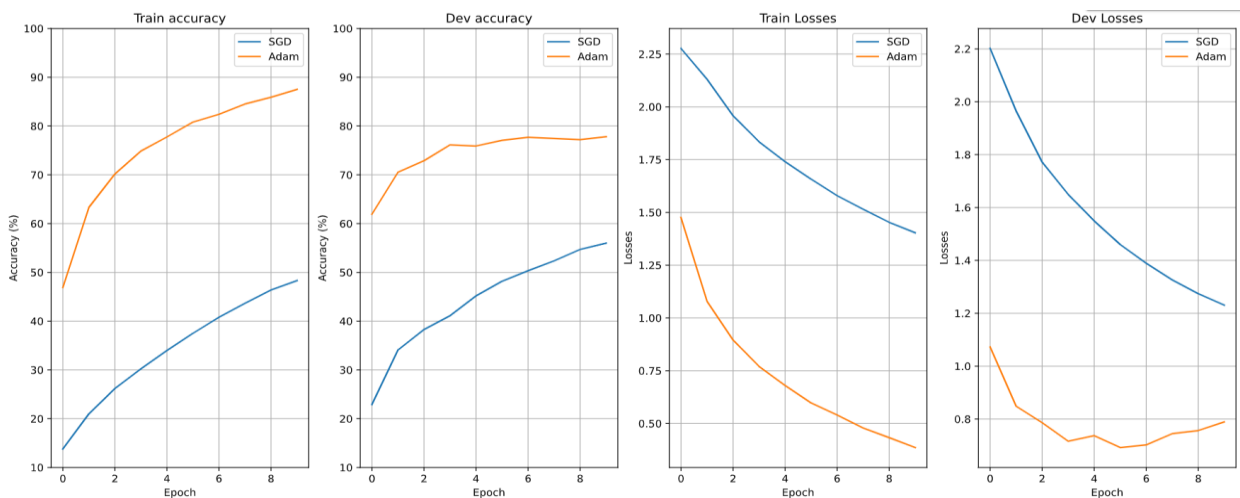
Adam (Adaptive Momentum)  در Adam ، ما از مزیت هر دوی Momentum و RMSprop بهره

می‌بریم و به صورت زیر نوشته می‌شود:

$$\begin{cases} v = (1 - \beta_1)d_L + \beta_1 v_{t-1} \text{ (Average)} \\ s = (1 - \beta_2)d_L^2 + \beta_2 s_{t-1} \text{ (Varinace)} \\ W - \frac{\eta \tilde{v}}{\sqrt{\tilde{s} + \epsilon}} \rightarrow W \end{cases} \text{ , Bias Correction : } \begin{cases} \tilde{v} = \frac{v}{1 - \beta_1^t} \\ \tilde{s} = \frac{s}{1 - \beta_2^t} \end{cases}$$

در واقع Adam از هر دوی میانگین و واریانس گرادیان در آپدیت وزن ها استفاده می‌کند. عبارت Bias Correction هم به این معنی است که ما در ابتدای یادگیری ، با مقدار بزرگ تری از Learning rate شروع می‌کنیم و هر چه جلوتر می‌رویم ، آنرا کاهش می‌دهیم .

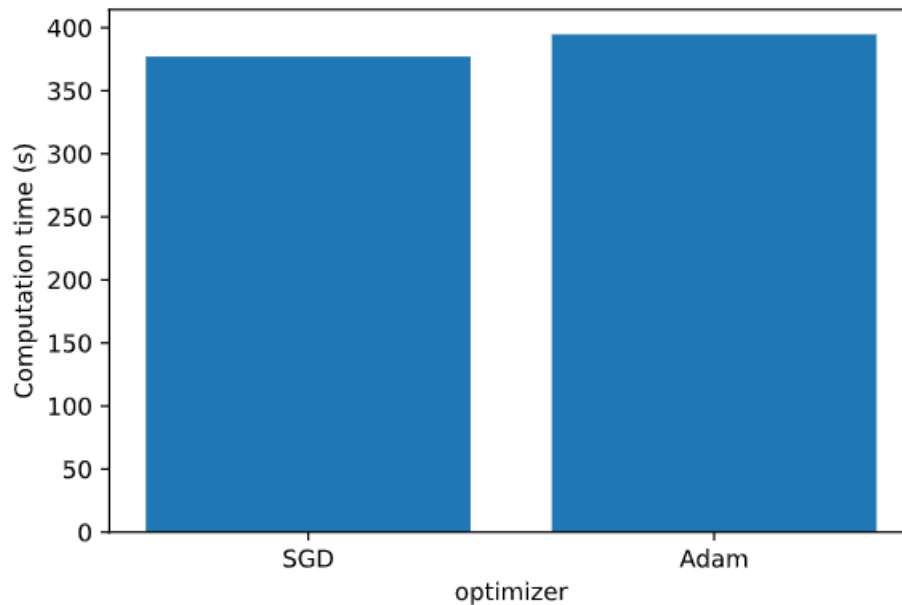
حال نتایج حاصل از آموزش مدل برای دو Optimizer بالا را مقایسه می‌کنیم :



شکل 1-4-1: دقت و خطا برای دادگان آموزش و ارزیابی

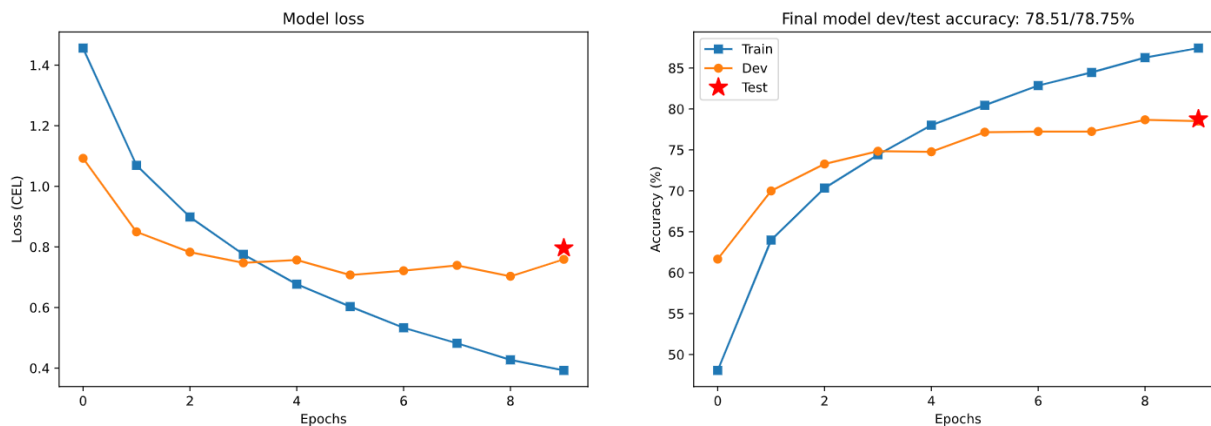
طبق نتایج بالا ، بهینه ساز Adam به مراتب بهتر از SGD عمل کرده است . البته با افزایش تعداد epoch و یا نرخ یادگیری ، در نهایت مدل به کمک SGD نیز می تواند یادگیری داشته باشد ولی با Adam در همان epoch های اولیه ، مدل عملکرد بسیار خوبی دارد .

همچنین زمان محاسباتی مورد نیاز برای آموزش با هر کدام از توابع بهینه ساز به صورت زیر می باشد :



شکل 1-4-2 : زمان محاسباتی برای توابع بهینه ساز مختلف

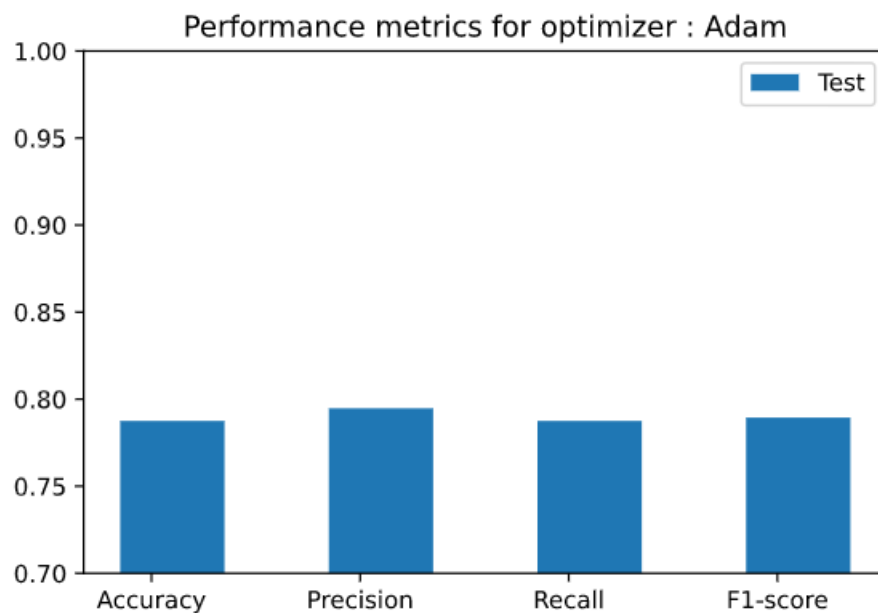
با انتخاب Adam به عنوان تابع بهینه ساز ، دادگان تست را با مدل آموزش دیده ، تخمین می زنیم :



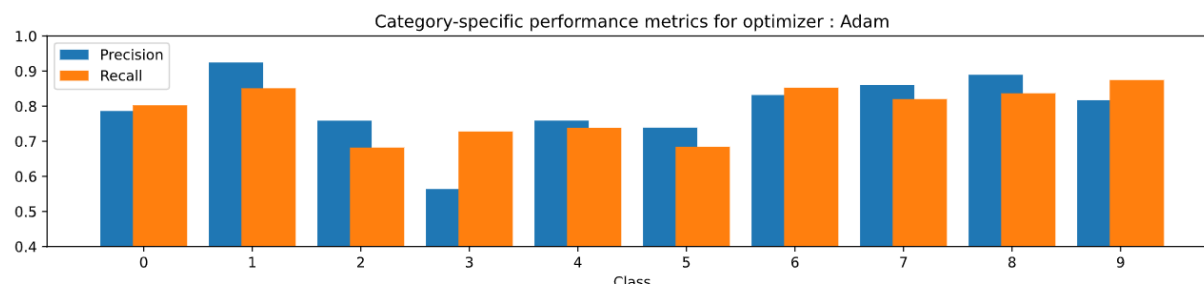
شکل 1-4-3 : عملکرد مدل منتخب با تابع بهینه ساز Adam بر روی دادگان تست

در ادامه از 4 متریک معروف Accuracy , Precision , Recall , F1-score برای تخمین بهتر مدل استفاده

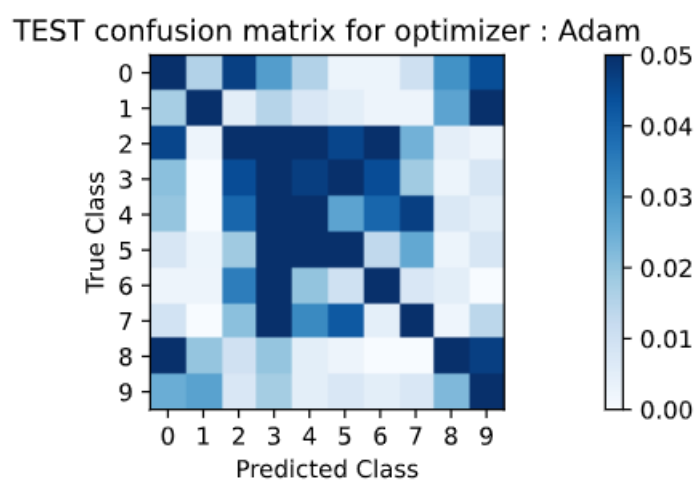
می کنیم :



شکل 4-4-1: APRF برای تابع بهینه ساز Adam



شکل 5-4-1: PR برای هر یک از 10 کلاس برای تابع بهینه ساز Adam



شکل 6-4-1: ماتریس آشفتگی برای تابع بهینه ساز Adam

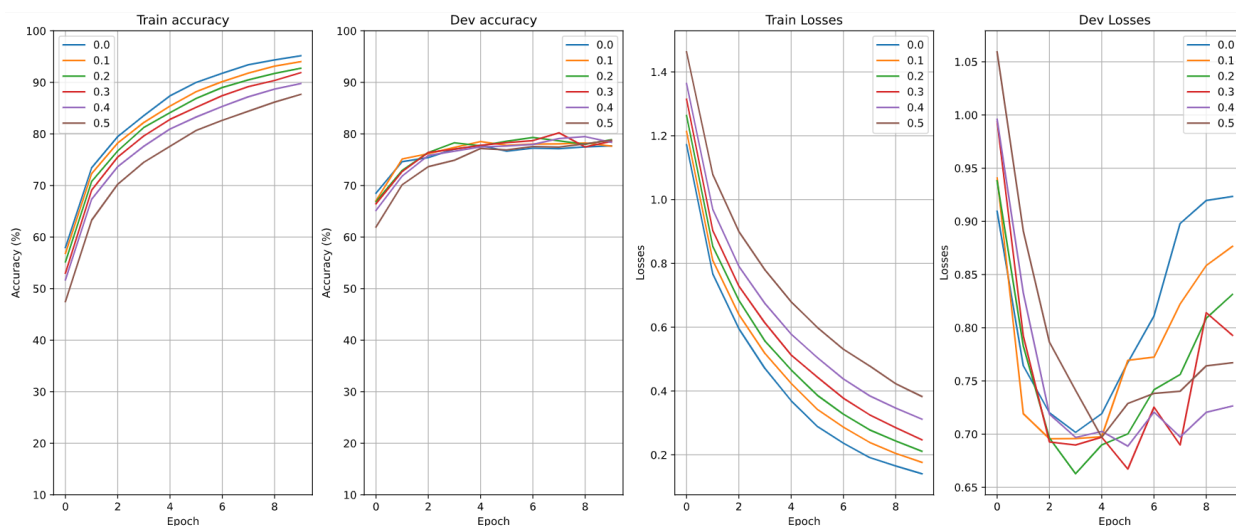
ه) تاثیر حذف تصادفی

در این قسمت برای 5 مقدار مختلف [0,0.1,0.2,0.3,0.4,0.5] به عنوان Dropout Rates، شبکه را آموزش می‌دهیم که این مقادیر احتمال صفر شدن وزن‌های مربوط به یک نورون را در طی آموزش نشان می‌دهد. همچنین Drop Out را تنها به لایه Fully Connected اضافه می‌کنیم.

هدف اصلی Drop Out این است که هر نورون در مدل، یادگیری را مستقل از نورون دیگر یاد بگیرد ولی نکته مهم اینجاست که در استفاده از کرنل‌ها، ما پیکسل‌های مختلف را به هم وابسته می‌کنیم و بین پیکسل‌های لایه خروجی هر لایه Correlation وجود دارد. بنابراین اضافه کردن درصد زیادی Drop Out مثل 0.5، احتمال اینکه مدل در یادگیری وابستگی بین پیکسل‌ها به مشکل بخورد را بالا می‌برد. ولی اضافه کردن مقدار کوچکی مثل 0.2 یا پایین‌تر، مانند اضافه کردن نویز به عکس‌های ما اثر می‌کند و می‌تواند باعث کاهش Over fit شود.

در مجموع نظرات متفاوتی نسبت به تاثیر مثبت/منفی Drop Out در اضافه کردن به شبکه CNN وجود دارد ولی به عنوان یک procedure کلی می‌توان مقدار کمی Drop Out به قسمت کانولوشنی CNN اضافه کرد که کمی داده‌ها را نویزی کند و همچنین می‌توان مقدار بیشتری به لایه Fully Connected در CNN اضافه کرد زیرا دیگر در این لایه مشکل وابستگی بین پیکسل‌ها را نداریم و قصد طبقه‌بندی داریم.

در زیر نتایج را برای مجموعه آموزش و ارزیابی (Devset) مقایسه می‌کنیم:



شکل 1-5-1: نمودار دقت و خطا برای نرخ متفاوت Drop out در دادگان آموزش و ارزیابی

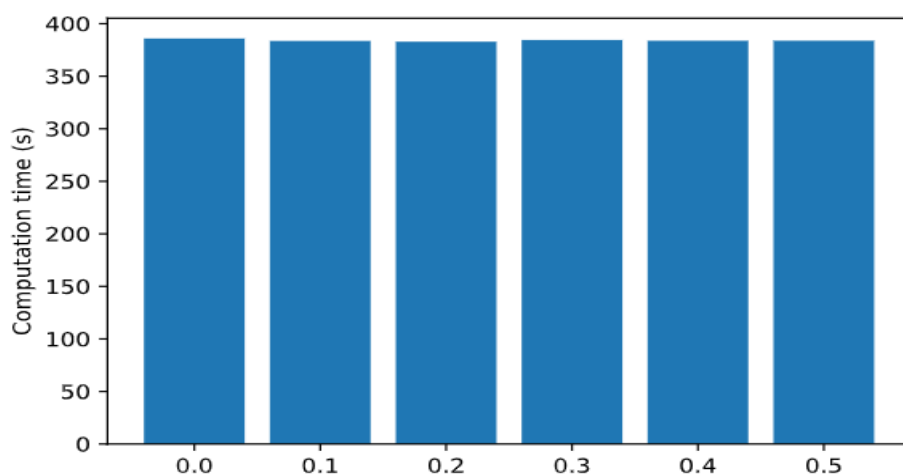
همانطور که مشاهده می‌شود ، در دادگان آموزش چون Over fit می‌شویم ، با افزایش Drop out ، دقت کاهش پیدا می‌کند. ولی در دادگان ارزیابی ، در نمودار دوم ، دقت ها در نهایت به هم نزدیک بوده و از روی آن نمی‌توان بهترین Drop out را پیدا کرد . با کمک قطعه کد زیر بهترین مقدار Drop out را پیدا می‌کنیم :

```
dropoutRates[np.argmax(devAcc[9, :])]
```

0.2

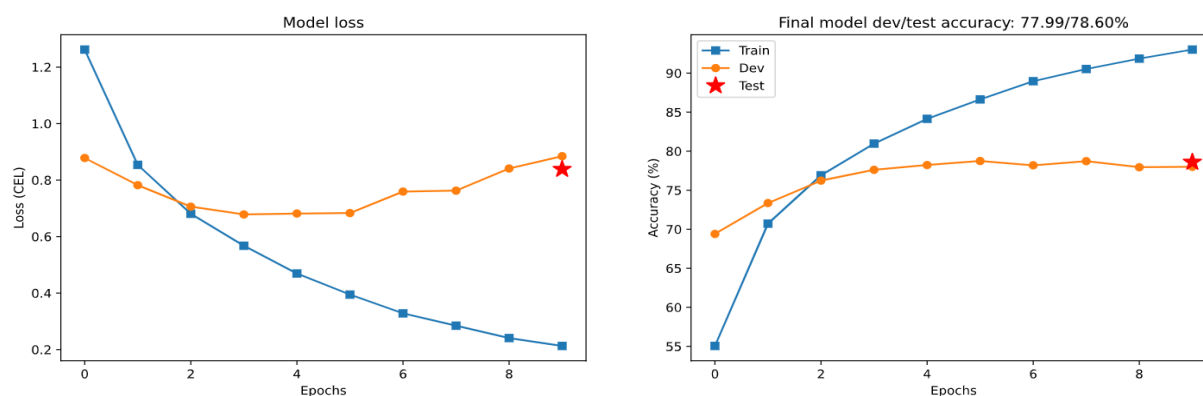
شکل 1-5-2 : بهترین مقدار Drop out بدست آمده از روی Performance دادگان ارزیابی

همچنین زمان محاسباتی بدست آمده برای مقادیر مختلف Drop out به صورت زیر می‌باشد :



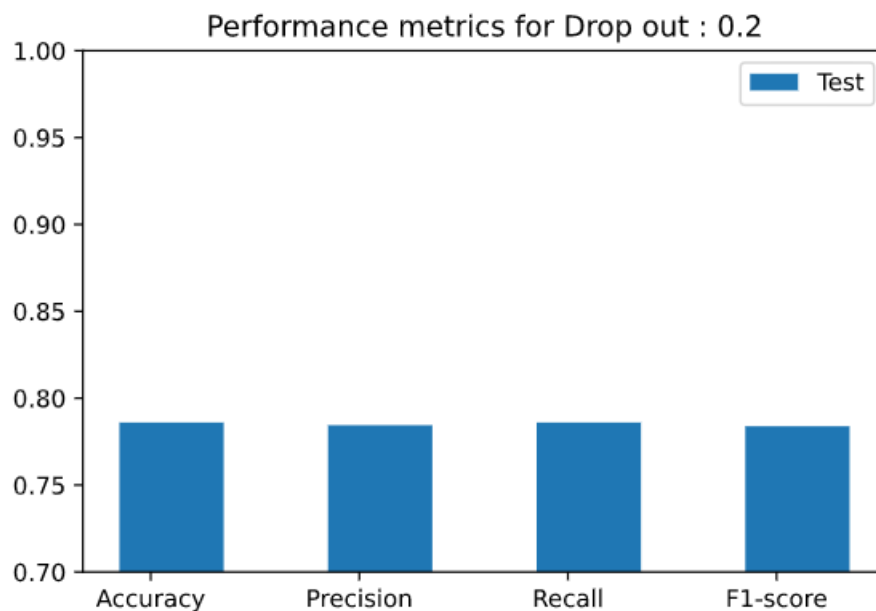
شکل 1-5-3 : زمان محاسباتی ناشی از مقادیر مختلف Drop out

با انتخاب Drop out=0.2 از روی دادگان ارزیابی (Dev set) ، برای دادگان تست مدل را تخمین می‌زنیم :

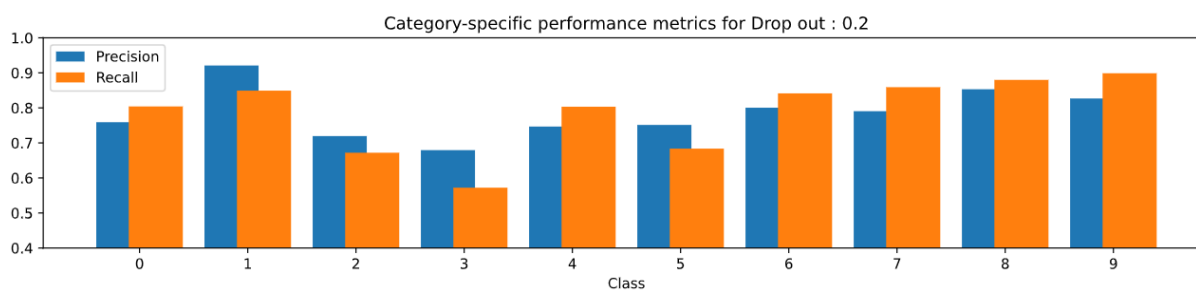


شکل 1-5-4 : دقت و خطای بدست آمده برای دادگان تست

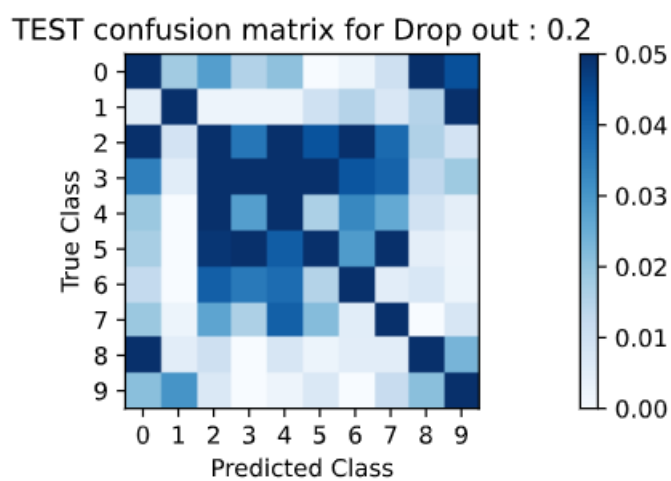
در ادامه از 4 متریک معروف Accuracy , Precision , Recall , F1-score برای تخمین بهتر مدل استفاده می‌کنیم :



شکل 5-5-1 : APRF برای DR=0.2



شکل 6-5-1 : PR برای هر یک از 10 کلاس موجود و DR=0.2



شکل 7-5-1 : ماتریس آشفتگی برای DR=0.2

سوال 2 :

الف : تحلیلی

نام و نام خانوادگی: _____ HW#3 - 11.09.1403 - 1403

$$out = b_3 + w_3^T x + \text{relu}(\tanh(b_1 + x^T w_1) w_2^T + b_2) \quad , \quad loss = \frac{1}{2} (out - 6)^2$$

مشتق از مربع

$$\rightarrow \frac{\partial loss}{\partial w_3} = \frac{1}{2} \times 2 (out - 6) \times \frac{\partial out}{\partial w_3} = (out - 6) [x + \cdot] = (out - 6) \cdot x$$

$$\rightarrow \frac{\partial loss}{\partial b_3} = (out - 6) \frac{\partial out}{\partial b_3} = (out - 6) \cdot 1$$

$$\rightarrow \frac{\partial loss}{\partial w_2} = (out - 6) \frac{\partial out}{\partial w_2} = (out - 6) \frac{\partial [\text{relu}(\tanh(b_1 + x^T w_1) w_2^T + b_2)]}{\partial w_2}$$

* $F(x) = \text{relu}(x) \rightarrow F' = \begin{cases} 0 & x < 0 \\ 1 & x > 0 \end{cases}$ $\xrightarrow{\text{در مشتق نداد}}$ $F'(x) = u(x)$, $g(u) = \tanh(u) \rightarrow g'(u) = (1 - \tanh^2(u)) = 1 - g^2$
 $\xrightarrow{\text{step function}}$

i) $u_1 = b_1 + x^T w_1$, $u_2 = w_2^T$, $\frac{\partial F(g(u_1))}{\partial x} = F'(g(u_1)) g'(u_1)$

$$\rightarrow \frac{\partial loss}{\partial w_2} = (out - 6) \frac{\partial F(g(u_1) u_2^T + b_2)}{\partial u_2} = (out - 6) \left[\frac{\partial (g(u_1) u_2^T + b_2)}{\partial u_2} \times F'(g(u_1) u_2^T + b_2) \right]$$

$$= (out - 6) [g(u_1) \times u_2^T] = (out - 6) [\tanh(b_1 + x^T w_1) \times u_2^T]$$

$$\rightarrow \frac{\partial loss}{\partial b_2} = (out - 6) \frac{\partial out}{\partial b_2} = (out - 6) \frac{\partial F(g(u_1) u_2^T + b_2)}{\partial b_2} = (out - 6) [1 \times F'(g(u_1) u_2^T + b_2)]$$

$$= (out - 6) [u_2^T \tanh(b_1 + x^T w_1)]$$

$$\rightarrow \frac{\partial loss}{\partial w_1} = (out - 6) \frac{\partial F(g(u_1) u_2^T + b_2)}{\partial w_1} = (out - 6) \frac{\partial (g(u_1) u_2^T + b_2)}{\partial w_1} \times F'(g(u_1) u_2^T + b_2)$$

$$\rightarrow g(u_1) u_2^T + b_2 = g(b_1 + x^T w_1) u_2^T + b_2 = k(w_1) \rightarrow \frac{\partial k(w_1)}{\partial w_1} = \frac{\partial g(b_1 + x^T w_1)}{\partial w_1} \times g'(b_1 + x^T w_1) u_2^T = x \times (1 - \tanh^2(b_1 + x^T w_1)) u_2^T$$

$$\rightarrow \frac{\partial loss}{\partial w_1} = (out - 6) \times \frac{\partial k(w_1)}{\partial w_1} = (out - 6) \times x \times (1 - \tanh^2(b_1 + x^T w_1)) u_2^T$$

$$\rightarrow \frac{\partial loss}{\partial b_1} = (out - 6) \frac{\partial (g(u_1) u_2^T + b_2)}{\partial b_1} \times F'(g(u_1) u_2^T + b_2) , \quad \frac{\partial k(b_1)}{\partial b_1} = \frac{\partial g(b_1 + x^T w_1)}{\partial b_1} \times g'(b_1 + x^T w_1) u_2^T$$

$$\rightarrow \frac{\partial k(b_1)}{\partial b_1} = 1 \times (1 - \tanh^2(b_1 + x^T w_1)) u_2^T \rightarrow \frac{\partial loss}{\partial b_1} = (out - 6) \times (1 - \tanh^2(b_1 + x^T w_1)) u_2^T$$

شکل 1-1-2 : قسمت اول محاسبات (بدست آوردن گرادیان های مربوطه)

$a = 3, b = 0$: ابتدا معادلات وزن ها را به صورت زیر می‌نویسیم:

$$W_1^1 = \begin{pmatrix} 1.3 & -1.73 & 0 \\ -1.2 & -1.3 & 1.3 \end{pmatrix}, b_1^1 = \begin{pmatrix} 1.1 \\ 1.3 \end{pmatrix}, W_2^1 = \begin{pmatrix} 1.35 & -1.5 & 1.3 \end{pmatrix}, b_2^1 = 1.1$$

$$W_3^1 = \begin{pmatrix} -1.3 \\ 1 \end{pmatrix}, b_3^1 = 1.1, x^1 = \begin{pmatrix} 2 \\ 3 \end{pmatrix}$$

$\rightarrow out^1 = 1.1 + (-1.3 \times 1) \left(\frac{2}{3} \right) + \text{relu} \left[\text{tanh} \left((1.1 \times 1.1 \times 1.3) + (2.3) \left(\begin{pmatrix} 1.35 & -1.73 & 0 \\ -1.2 & -1.3 & 1.3 \end{pmatrix} \right) \begin{pmatrix} 1.35 \\ -1.5 \\ 1.3 \end{pmatrix} \right) + 1.1 \right] = 4.147$

\rightarrow

$$W_3^2 = W_3^1 - \alpha \frac{\partial loss}{\partial W_3^1} = \begin{pmatrix} -1.3 \\ 1 \end{pmatrix} - 1.1 \times \left[(4.147 - 6) \right] \begin{pmatrix} 2 \\ 3 \end{pmatrix} = \begin{pmatrix} 1.06 \\ 1.459 \end{pmatrix}$$

$$b_3^2 = b_3^1 - \alpha \frac{\partial loss}{\partial b_3^1} = 1.1 - 1.1 \times \left[(4.147 - 6) \right] = 1.253$$

$$W_2^2 = W_2^1 - \alpha \frac{\partial loss}{\partial W_2^1} = \begin{pmatrix} 1.35 & -1.5 & 1.3 \end{pmatrix} - 1.1 \times \left[(4.147 - 6) \right] \left(\text{tanh} \left((1.1 \times 1.1 \times 1.3) + (2.3) \left(\begin{pmatrix} 1.35 & -1.73 & 0 \\ -1.2 & -1.3 & 1.3 \end{pmatrix} \right) \begin{pmatrix} 1.35 \\ -1.5 \\ 1.3 \end{pmatrix} \right) + 1.1 \right) \begin{pmatrix} 1.35 \\ -1.5 \\ 1.3 \end{pmatrix} \right)$$

$$x \left(\begin{pmatrix} -1.997 & -1.999 & 1.883 \\ -1.5 \\ 1.3 \end{pmatrix} + 1.1 \right) = \begin{pmatrix} 1.53 & -1.653 & 1.65 \end{pmatrix}$$

1.997

$$b_2^2 = b_2^1 - \alpha \frac{\partial loss}{\partial b_2^1} = 1.1 - 1.1 \times \left[(4.147 - 6) \right] \text{relu} \left(\frac{1.997}{1.1} \right) = 1.253$$

$$W_1^2 = W_1^1 - \alpha \frac{\partial loss}{\partial W_1^1} = \begin{pmatrix} 1.3 & -1.73 & 0 \\ -1.2 & -1.3 & 1.3 \end{pmatrix} - 1.1 \times \left[(4.147 - 6) \right] \left(\frac{1 - \left\| \begin{pmatrix} -1.997 & -1.999 & 1.883 \end{pmatrix} \right\|}{2.1772} \right) \begin{pmatrix} 1.35 & -1.5 & 1.3 \end{pmatrix}$$

$$= \begin{pmatrix} 1.568 & 1.7471 & -1.163 \\ -1.898 & -1.893 & 1.056 \end{pmatrix}$$

$$b_1^2 = b_1^1 - \alpha \frac{\partial loss}{\partial b_1^1} = \begin{pmatrix} 1.1 \\ 1.3 \end{pmatrix} - 1.1 \times \left[(4.147 - 6) \right] \times \left(1 - \frac{2.1772}{2.1772} \right) \begin{pmatrix} 1.35 \\ -1.5 \\ 1.3 \end{pmatrix} = \begin{pmatrix} -1.366 \\ 1.7456 \\ 1.2919 \end{pmatrix}$$

$\rightarrow out^2 = 1.253 + (1.06 \times 1.459) \left(\frac{2}{3} \right) + \text{relu} \left[\text{tanh} \left((-1.366 \times 1.7456 \times 1.2919) + (2.3) \left(\begin{pmatrix} 1.568 & 1.7471 & -1.163 \\ -1.898 & -1.893 & 1.056 \end{pmatrix} \right) \begin{pmatrix} 1.53 \\ -1.653 \\ 1.65 \end{pmatrix} \right) + 1.253 \right]$

$x \left(\begin{pmatrix} 1.53 \\ -1.653 \\ 1.65 \end{pmatrix} + 1.253 \right) = 4.1642$

\rightarrow

$$W_3^3 = W_3^2 - \alpha \frac{\partial loss}{\partial W_3^2} = \begin{pmatrix} 1.06 \\ 1.459 \end{pmatrix} - 1.1 \times \left[(4.1642 - 6) \right] \begin{pmatrix} 2 \\ 3 \end{pmatrix} = \begin{pmatrix} 1.278 \\ 1.866 \end{pmatrix}$$

$$b_3^3 = b_3^2 - \alpha \frac{\partial loss}{\partial b_3^2} = 1.253 - 1.1 \times \left[(4.1642 - 6) \right] = 1.389$$

$$W_2^3 = W_2^2 - \alpha \frac{\partial loss}{\partial W_2^2} = \begin{pmatrix} 1.53 & -1.653 & 1.65 \end{pmatrix} - 1.1 \times \left[(4.1642 - 6) \right] \left(\text{tanh} \left((-1.366 \times 1.7456 \times 1.2919) + (2.3) \times \begin{pmatrix} 1.568 & 1.7471 & -1.163 \\ -1.898 & -1.893 & 1.056 \end{pmatrix} \begin{pmatrix} 1.53 \\ -1.653 \\ 1.65 \end{pmatrix} \right) + 1.253 \right) \begin{pmatrix} 1.53 \\ -1.653 \\ 1.65 \end{pmatrix}$$

شکل 2-1-2: قسمت دوم محاسبات (بدست آوردن وزن های بروز شده و خروجی در epoch اول)

$$b_2^3 = b_2^2 - \alpha \frac{\partial \text{loss}}{\partial b_2^2} = 1.253 - 0.1 \times 0 = 1.253$$

همچون $u(\tanh(b^T + 2W_1^T)W_2^T + b_2^2) = 0$ ، جمع $b_1^3, W_1^3, b_2^3, W_2^3$ اینترمستند پس مقایسه این در نهایت

epoch = 2 تیسرین کند.

$$W_1^3 = \begin{pmatrix} 1.568 & 1.7411 & -1.763 \\ -1.898 & -1.893 & 1.056 \end{pmatrix}, h_1^3 = \begin{pmatrix} -1.366 \\ 1.7456 \\ 1.2919 \end{pmatrix}$$

$$\rightarrow \text{out}^3 = 1.389 + (1.278 \ 1.866 \ 1.2) \begin{pmatrix} 2 \\ 3 \end{pmatrix} + \text{relu}(\tanh(-1.366 \ 1.7456 \ 1.2919) + (2.3) \begin{pmatrix} 1.568 & 1.7411 & -1.763 \\ -1.898 & -1.893 & 1.056 \end{pmatrix})$$

$$\times \begin{pmatrix} 1.503 \\ -1.653 \\ 1.165 \end{pmatrix} + 1.253 \Big] = 6.54$$

شکل 3-1-2: قسمت آخر محاسبات (مقدار نهایی خروجی بعد از دو epoch)

طبق قسمت دوم محاسبات ، در اولین epoch ، مقدار خروجی از 4.47 به 4.642 می‌رسد و در دومین epoch به مقدار 6.54 می‌رسد .

از آنجایی که برای هر مسئله ، Learning Rate بهینه وجود دارد ، در اینجا با Learning rate مناسب تر می‌توانستیم در epoch دوم ، به مقدار خروجی اصلی (6) نزدیکتر شویم .

نکته مهم در معماری شبکه در این است که خروجی های حاصل از گذر از سه تابع فعال ساز Tanh تنها زمانی در تغییر خروجی اثر می‌گذارند که جمع آنها مقداری مثبت باشد(به دلیل حضور ReLU) . در epoch دوم شاهد این اتفاق هستیم که تنها وزن های W^3, b^3 در ساخت خروجی موثر هستند و وزن های W^2, b^2, W^1, b^1 تغییری نسبت به مرحله قبل نمی‌کنند.

ب : تحقیق

1. تابع هزینه رگرسیون :

در ادامه در مورد سه تابع loss ذکر شده بحث می‌کنیم :

L1Loss:

این تابع مشابه تعریف MAE (Mean Absolute Error) می‌باشد ، با انی تفاوت که دیگر مانند MAE

میانگین گیری نداریم :

$$L1Loss = \sum_{i=1}^n |y_{true} - y_{predicted}|$$

L2Loss:

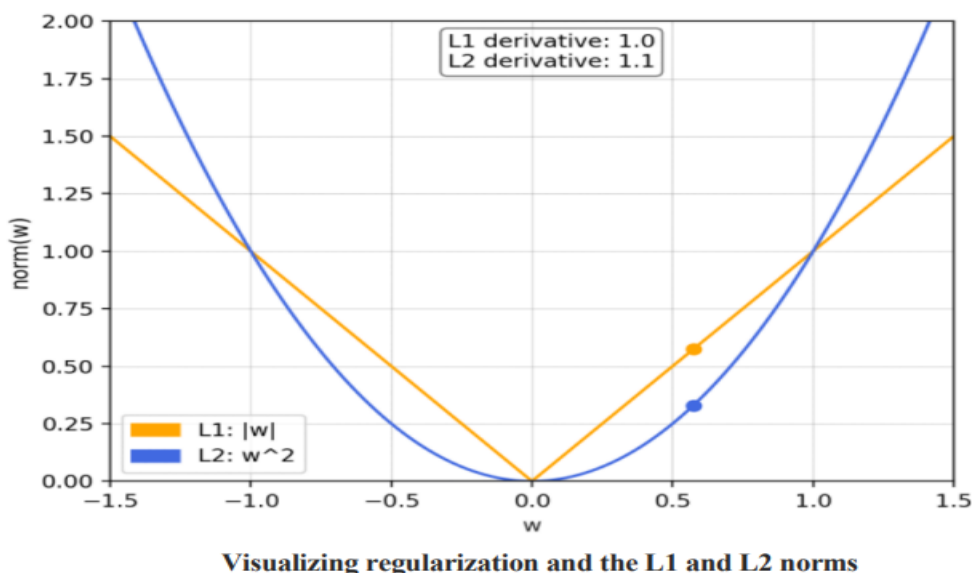
این تابع مشابه تعریف MSE (Mean Squared Error) می‌باشد ، با این تفاوت که دیگر مانند MSE میانگین گیری نداریم :

$$L2Loss = \sum_{i=1}^n (y_{true} - y_{predicted})^2$$

انتخاب بین L1Loss و L2Loss :

در اکثر موارد استفاده از L2Loss کاربرد داشته و ترجیح داده می‌شود . ولی در هنگامی که در دیتاست Outlier وجود داشته باشد (تعدادی از Sample ها نسبت به به توزیع بقیه Sample ها در فضای n بعدی ، به طور قابل توجهی تفاوت دارند.) معمولا وجود Outlier را با Scatter plot و یا box plot می‌توان تا حدی تشخیص داد) استفاده از L1Loss ارجحیت دارد . زیرا از آنجا که وجود Outlier ها می‌تواند در یادگیری ، ماشین را به اشتباه بندازد ، باید تاثیر آنها را کمتر کرد و یا مشابه تاثیر سایر داده ها در نظر گرفت . با در نظر گرفتن L2Loss برای دادگان دارای Outlier ، اگر دو داده از نظر مقداری تا حدودی در یک محدوده باشند ، تفاوت توان 2 آنها مشابه سایر دو داده دیگر می‌باشد ولی اگر یکی از دو داده ، Outlier باشد ، آنگاه با توان 2 رساندن ، ما گویی تاثیر آنها را بیشتر در نظر گرفتیم و به همین دلیل بهتر از L1Loss استفاده شود که به این مشکل بر نخوریم .

*در نتیجه تابع L2Loss بسیار محدب بوده و سرعت همگرایی را بالا می‌برد ولی در مواجهه با Outlier دچار نقص می‌شود . از طرفی L1Loss نسبت به Outlier حساسیت کمتری دارد ولی سرعت همگرایی آن پایین تر است . در زیر شکل دو تابع Loss مربوطه را آورده‌ایم :



شکل 1-1-2-2 : مقایسه دو تابع L1Loss و L2Loss

Huber Loss:

ایده مطرح Huber Loss استفاده از robustness (مقامت) L1Loss در برابر Outlier و شکل Convex (محدب) ، L2Loss می باشد که سرعت همگرایی را بالا ببرد . برای این منظور باید یک تابع Piecewise (شرطی) تعریف کنیم که در مرز بین دو شرط مشتق پذیر باشد و به صورت زیر تعریف می شود :

$$L_D(x) = \begin{cases} \frac{1}{2\delta}x^2 + \frac{1}{2}\delta & |x| \leq \delta \\ |x| & |x| \geq \delta \end{cases}$$

تابع $L_D(x)$ در بالا ، مشتق دوم ندارد ، به همین دلیل Smooth نیست . برای Smooth کردن ایت تابع راههای زیادی وجود دارد که معروفترین آن ، استفاده از Pseudo-Huber loss می باشد :

$$L_{HP}(x) = \delta \left(\sqrt{1 + \frac{x^2}{\delta^2}} \right)$$

که در نزدیکی صفر برابر $\frac{1}{2\delta}x^2 + \frac{1}{2}\delta$ و در خط مجانبی برابر $|x|$ می باشد .

2. استفاده از داده ارزیابی :

دو دلیل برای اینکه خطای دادگان آموزش و ارزیابی به هم نزدیک باشد ، وجود دارد :

❶ امکان دارد در هنگام تقسیم دادگان به بخش آموزش و ارزیابی ، بخش ساده تر دادگان به بخش ارزیابی اختصاص یابد و باعث شود که مدل در پیشبینی دادگان ارزیابی به خوبی دادگان آموزش عمل کند (در صورتی که دادگان اولیه را رندوم Shuffle کنیم ، این مشکل احتمالا تصادفی می باشد)

❷ هنگامی که از دادگان ارزیابی در دادگان آموزش استفاده شده باشد . این اتفاق بیشتر در بحث های object detection در CNN امکان وقوع دارد ، از آنجایی که ما از بحث های مثل Data augmentation و یا Transformer ها (rotate عکس ، نویزی کردن ، ...) استفاده می کنیم ، امکان دارد تعدادی Sample با Correlation بالا در هر دو بخش آموزش و ارزیابی وجود داشته باشد .

در این صورت مدل به جای اینکه داده های از قبل ندیده را (Unseen data) پیش بینی کند ، از اطلاعات دادگان آموزش در پیش بینی دادگان ارزیابی استفاده می کند که به آن data leakage هم می گویند.

3. گرادیان نزولی به همراه تکانه :

:SGD

در این روش ما هر بار ماتریس وزن ها را در جهت خلاف گرادیان تابع هزینه و اسکیل α آپدیت می کنیم:

$$W_{new} = W_{old} - \eta d_L$$

نکته مهم در اینجا این است که ما برای هر Sample ، ماتریس W را آپدیت می کنیم ، پس در جایی که دادگان شباهت زیادی به یکدیگر دارند ، این روش می تواند بسیار خوب عمل کند . بنابراین یادگیری آن بسیار حساس است و در صورت وجود Outlier در داده ها می تواند ضعیف عمل کند. به همین دلیل ایده Mini batch SGD هم مطرح شد تا بتواند تا حدی این مشکل را حل کند.

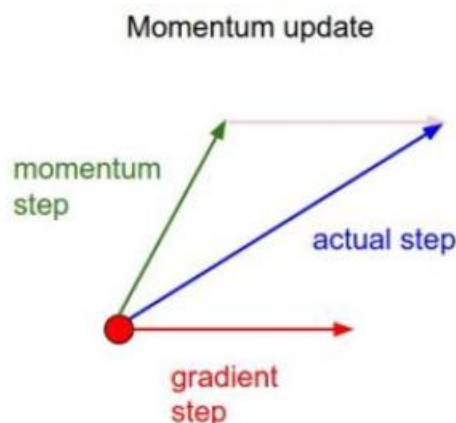
:SGD with Momentum

برای اینکه روند تغییرات نمودار SGD را کمی Smooth تر کنیم از Momentum استفاده می کنیم . در واقع در Momentum ، هر نقطه ترکیب وزن دار خودش با نقطه قبلی می باشد که به SGD with Momentum معروف است :

$$\begin{cases} v = (1 - \beta)d_L + \beta v_{t-1} \\ W - \eta v \rightarrow W \end{cases}$$

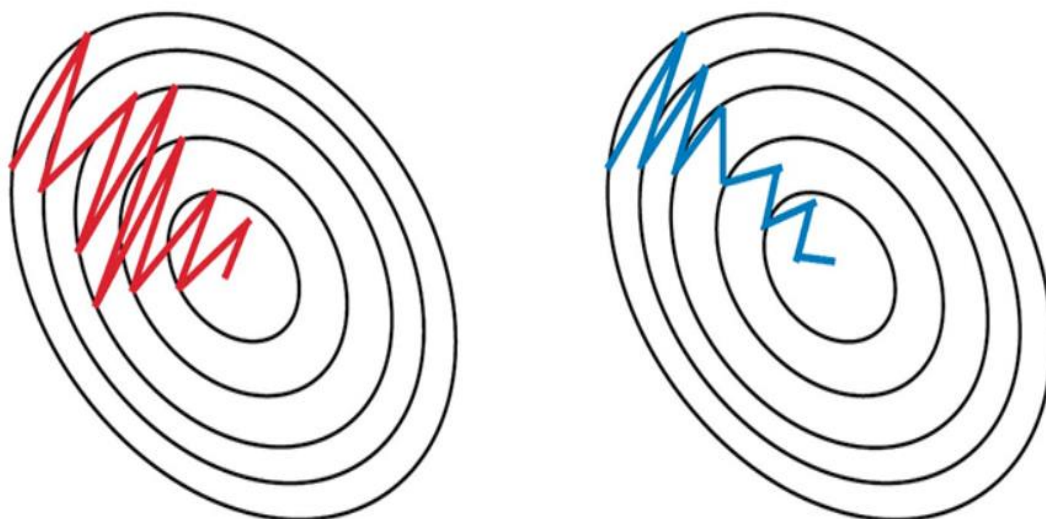
به دو دلیل می توان گفت که SGD with Momentum بهتر از SGD کار می کند :

❶ در SGD ، ما گرادیان دقیق تابع Loss را حساب نمی کنیم و در عوض آن را روی هر Batch یا Mini Batch تخمین می زنیم . این به این معنی است که در هر بار ما در جهت optimum حرکت نمی کنیم زیرا گرادیان ما دارای نویز می باشد . با کمک Momentum و میانگین گیری ، می توانیم تا حدی اثر این نویز را کاهش دهیم و در جهت مناسب تر حرکت کنیم . در زیر جهت حرکت هر دو روش را می بینیم :



شکل 2-2-3-1 : Momentum step vs Gradient step

❌ دلیل دوم وجود دره ها (Ravines) می باشد که در واقع Contour هایی هستند که نقاط مینیمم محلی در شبکه عصبی را نشان می دهند و SGD در جهت یابی آنها به مشکل می خورد و به جای اینکه در جهت آنها حرکت کند در بین آنها نوسان می کند . به کمک Momentum می توانیم به گرادیان در جهت مناسب شتاب دهیم . در زیر نحوه برخورد این دو روش را با Ravines ها می بینیم :



شکل 2-3-2-2 : SGD Oscillation(Left) vs SGD With Momentum Oscillation(Right)

پ : پیاده سازی شبکه پرسپترون در کاربرد رگرسیون

1. تولید دادگان :

```
#Creating Data
X=np.linspace(-2*np.pi,2*np.pi,10**4).reshape((10**4,1)) # X coordinate
Y=np.linspace(-2*np.pi,2*np.pi,10**4).reshape((10**4,1)) #y coordinate
np.random.shuffle(X) #Shuffling X
np.random.shuffle(Y) #Shuffling y
F=np.sin(X+Y)
Data=np.hstack((X,Y,F))#(X,y) coordinates with actual values
X_train,y_train=Data[:,2],Data[:,2].reshape((10000,1))#Train splite(X , y in [-2*pi,2*pi])
X_test=np.hstack((np.linspace(0,2*np.pi,10**4).reshape((10**4,1)),np.zeros((10**4,1))))#Test Splite (y=0 for all samples , X in [0,2*pi])
y_test=np.sin(X_test[:,0]+X_test[:,1])#Creating Sin(x)
```

شکل 1-1-3-2 : تولید دادگان آموزش و ارزیابی

طبق گفته سوال ، برای دادگان آموزش مختصات های (x,y) را در بازه $[-2 * \pi, 2 * \pi]$ انتخاب می کنیم . همچنین برای دادگان ارزیابی ، تمامی y ها را صفر فرض می کنیم و مقادیر x را در بازه $[0, 2 * \pi]$ انتخاب می کنیم.

2. پیش پردازش :

در این سوال باتوجه به اینکه مختصات (x,y) در بازه محدودی انتخاب شده‌اند ، نرمال کردن دادگان تاثیر چندان مثبتی نسبت به نکردن آن نخواهد داشت .

3. پیاده سازی مدل :

در زیر تمامی توابع مورد نیاز (اعم از Loss Function یا Activation) و مشتق آنها را تعریف کردیم :

```
# loss function and its derivative
def mse(y_true, y_pred):
    return np.mean(np.power(y_true-y_pred, 2))

def mse_prime(y_true, y_pred):
    return 2*(y_pred-y_true)/y_true.size
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def sigmoid_prime(x):
    return np.exp(-x) / (1 + np.exp(-x))**2

def tanh(x):
    return np.tanh(x)

def tanh_prime(x):
    return 1 - np.tanh(x)**2

def relu(x):
    return np.maximum(x, 0)

def relu_prime(x):
    return np.array(x >= 0).astype('int')
```

شکل 2-3-3-1: تعریف تابع و مشتق تابع Activation/Loss

همچنین یک کلاس (Class) کلی به نام Network تعریف می‌کنیم ، که در آن عملیات افزایش یک لایه مخفی ، آموزش مدل و پیشبینی مدل را انجام می‌دهیم . از طرفی در کلاسی دیگر بنام FCLayer ، دو عملیات Forward و Backward Propagation را انجام می‌دهیم .

در نهایت مدل Sequential حاصل از ادغام لایه های مخفی و توابع فعال ساز و خطا ، به صورت زیر می‌باشد :

```
nepochs=30
net = Network()
net.add(FCLayer(2, 25))
net.add(ActivationLayer(tanh, tanh_prime))
net.add(FCLayer(25, 25))
net.add(ActivationLayer(tanh, tanh_prime))
net.add(FCLayer(25, 1))
net.add(ActivationLayer(tanh, tanh_prime))
net.use(mse, mse_prime)
Error,Error_test=net.fit(X_train.reshape((10000,1,2)), y_train.reshape((10000,1,1)),X_test,y_test, epochs=nepochs, learning_rate=0.03)
```

شکل 2-3-3-2: معماری مدل آموزش داده شده

طبق شکل بالا ، از سه لایه مخفی استفاده کردیم که ورودی تا لایه اول یک ماتریس 25×2 ، لایه اول تا دوم ، ماتریس 25×25 ، لایه دوم تا سوم، ماتریس 25×25 و لایه سوم تا خروجی ، ماتریسی با سایز 1×25 وجود دارد .

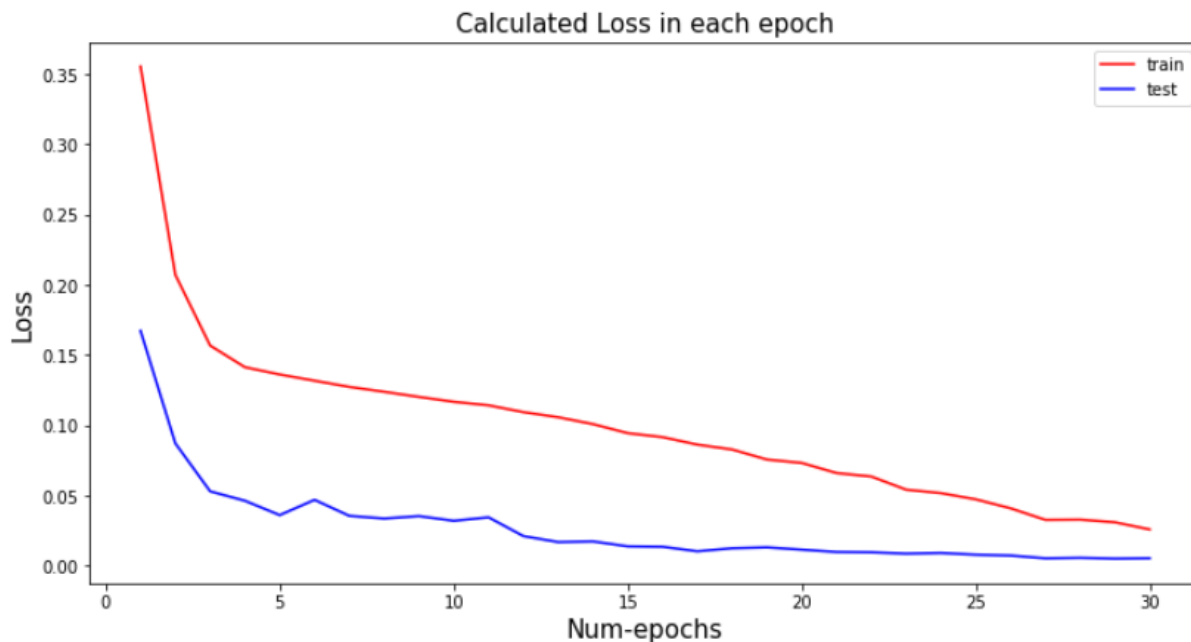
در هر لایه از توابع فعال ساز Tanh استفاده کردیم و تابع خطا برای مرحله Backward را MSE در نظر گرفتیم.

در نهایت با آموزش مدل بر روی دادگان آموزش ، دو مقدار خطای آموزش و ارزیابی را در یک آرایه ذخیره می‌کنیم . مقادیر خطای دادگان آموزش برای تعداد تکرار epoch=30 ، بصورت زیر می‌باشد :

epoch 1/30	error=0.355473
epoch 2/30	error=0.207209
epoch 3/30	error=0.156831
epoch 4/30	error=0.141409
epoch 5/30	error=0.136227
epoch 6/30	error=0.131790
epoch 7/30	error=0.127393
epoch 8/30	error=0.123908
epoch 9/30	error=0.120261
epoch 10/30	error=0.116774
epoch 11/30	error=0.114278
epoch 12/30	error=0.109436
epoch 13/30	error=0.105799
epoch 14/30	error=0.100906
epoch 15/30	error=0.094450
epoch 16/30	error=0.091631
epoch 17/30	error=0.086287
epoch 18/30	error=0.082825
epoch 19/30	error=0.075663
epoch 20/30	error=0.073266
epoch 21/30	error=0.066067
epoch 22/30	error=0.063590
epoch 23/30	error=0.054227
epoch 24/30	error=0.051763
epoch 25/30	error=0.047350
epoch 26/30	error=0.040979
epoch 27/30	error=0.032749
epoch 28/30	error=0.032917
epoch 29/30	error=0.031008
epoch 30/30	error=0.025920

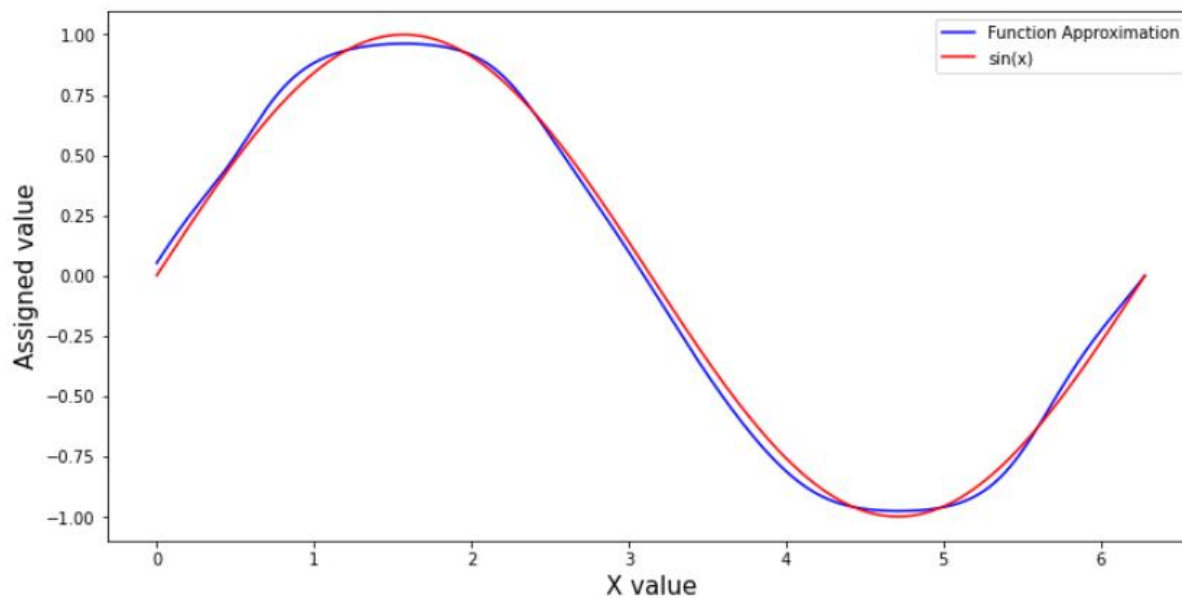
شکل 3-3-3 : خطای بدست آمده در هر epoch آموزش مدل برای دادگان آموزش

در ادامه ، نمودار خطای دادگان آموزش و ارزیابی را در نمودار بر حسب تعداد epoch مدل رسم می‌کنیم :



شکل 2-3-4: نمودار خطای بدست آمده دادگان آموزش و ارزیابی بر حسب تعداد epoch

در نهایت با مدل آموزش دیده ، برای دادگان ارزیابی ، خروجی تابع داده شده را پیشبینی کرده و بر حسب مقدار اصلی تابع (True Value) ، رسم می کنیم :



شکل 2-3-5: True Value (قرمز) vs Predicted Value (آبی) تابع $\sin(x)$