۱ الف

یادگیری تقویتی

در طراحی اولیه NAS ما یک ماژول کنترلر داریم که وظیفه ایجاد هایپرپارامتر ها برای تشکیل یک معماری جدید برای مدل را دارد. این ماژول به صورت یک شبکه عصبی بازگشتی با خروجی متغیر در نظر گرفته شده است. این شبکه بازگشتی (کنترلر) از طریق یادگیری تقویتی آموزش میبیند. برای تبدیل آموزش شبکه بازگشتی به یک وظیفه یادگیری تقویتی: لیست توکن هایی که یک شبکه (معماری شبکه) را توضیح میدهند به عنوان فضای اکشن در نظر میگیریم. دقت مدل ایجاد شده با ابرپارامتر های انتخابی بعد از همگرایی پارامتر ها به عنوان جایزه و در نهایت تابع ضرری که هدفش افزایش جایزه است.

یادگیری تکاملی

الگوریتم اول، NEAT، روشی را برای تکامل شبکه های عصبی از طریق الگوریتم ژنتیک پیشنهاد می دهد. این الگوریتم هم ساختار و توپولوژی شبکه را تکامل میدهد و هم وزن های شبکه را. الگوریتم دوم، AmoebaNet، از الگورتیم های تکاملی برای جست و جوی معماری هایی با عملکرد قوی استفاده می کند. در این روش با استفاده از انتخاب مبتنی بر تورنومنت در هر دوره به انتخاب بهترین کاندید (بهترین معماری کاندید) می پردازد و فرزندان جهش یافته آن را دوباره به جمعیت اصلی بر میگرداند. اما این الگوریتم انتخاب مبتنی بر تورنومنت را به گونه ای ویرایش کرده است که معماری های جدید تر شانس بیشتری داشته باشند و در هر چرخش قدیمی ترین معماری را دور میریزد. این روش که به آن تکامل مبتنی بر سن نیز گفته می شود به ما کمک میدهد که بخش بزرگتری از فضای جست و جو را کاوش کنیم و در همان ابتدا در فضای پارامتر های یک مدل نسبتا خوب گیر نکنیم.

روش های مبتنی بر گرادیان

در این روش ها ما پروسه انتخاب ابرپارامتر ها را با پروسه یادگیری وزن های شبکه ترکیب می کنیم. بدین ترتیب درکنار یادگیری وزن های شبکه می توانیم ابر پارامتر های شبکه را نیز یادبگیریم. منتها مشکل اصلی این روش انتخاب عملگرهایی است که مشتق پذیر باشند تا بتوان گرادیان آن ها را محاسبه کرد.

۱ ب

استفاده از روش مبتنی بر یادگیری تقویتی برای پارامتر سایز ورودی امکان پذیر است. در این حالت شبکه عصبی بازگشتی در ماژول کنترلر به تولید سایز های ورودی مختلف می پردازد و ما با آموزش و همگرایی شبکه های مختلف متناسب با این سایز های تولیدی به ارزیابی آن ها می پردازیم و با بدست آوردن میزان جایزه به مرحله بعد و تولید معماری جدید می رویم. به همین ترتیب بررسی لایه کانولوشن نیز امکان پذیر است. فقط کافی است که شبکه عصبی بازگشتی به تولید حالت های مختلف لایه های کانولوشنی بپردازد.

استفاده از روش مبتنی بر الگوریتم های تکاملی نیز برای هردو پارامتر خواسته شده امکان پذیر است. در این حالت ما باید سایز ورودی و لایه کانولوشن را در ژن مرتبط با هر معماری انکود کنیم و سپس الگوریتم های تکاملی به بهبود این ابرپارامتر ها از طریق عملگر های انتخاب و جهش می پردازند.

در روش مبتنی بر گرادیان ما باید به دنبال راه حلی باشیم که بتوان این ابر پارامتر ها را به نحوی مشتق پذیر (نسبت به تابع ضرر) در بیاوریم. یک راه حل استفاده از رویکرد One-Shot است. در این حالت یک مدل بزرگ متشکل از حالت های مختلف ابر پارامتر ها تشکیل می شود و سپس به آموزش این مدل بزرگ می پردازیم. سپس به بررسی مسیر های مختلف درون شبکه پرداخته و مسیر بهینه را انتخاب می کنیم. به کمک این رویکرد می توان ابرپارامتر لایه کانولوشن را مبتنی بر گرادیان بهینه کرد اما نمی توان سایز های مختلف ورودی را امتحان کرد زیرا باید برای هر سایز یک شبکه مستقل بزرگ جداگانه ایجاد شود که عملا اینکار همان جست و جوی شبکه ای برای پارامتر سایز ورودی است و از مزایای گرادیان بهره نمی برد.

۲

راهکار اول: Proxy Task Performance

در این روش ما از یک وظیفه ساده تر برای تخمین بازدهی شبکه اصلی به جای آموزش این شبکه از ابتدا می پردازیم. وظایف ساده تر از جمله:

آموزش بر روی یک مجموعه داده کوچک تر یا آموزش با تعداد ایپاک کمتر. همچنین می توان شبکه را به تعدادی سلول عین هم که پشت سر هم تکرار می شوند تقلیل داد و سپس به یادگیری فقط یک سلول بپردازیم و بعد از بدست آوردن یک سلول خوب به تکرار آن پشت سر هم بپردازیم. روش دیگر این است که از رگرسیون برای پیشبینی شکل نمودار عملکرد بر روی داده های ارزیابی استفاده کنیم. در این حالت با استفاده از عملکرد مدل در چند ایپاک ابتدایی به پیش بینی عملکرد مدل در ایپاک های بعدی می پردازیم.

راهکار دوم: اشتراک گذاری پارامتر ها

در این حالت به جای آموزش هر شبکه از ابتدا، ما از وابستگی های بین این شبکه ها (پارامتر های مشترک)استفاده می کنیم و به قولی از وزن ها مجددا استفاده می کنیم.

راهکار سوم: پیش بینی مدل

در این روش که از نظر بنده بسیار ایده جالبی دارد، ما به پیش بینی وزن های شبکه بر اساس ابرپارامتر های آن میپردازیم. به این صورت که ما یک بردار انکودینگ از ابرپارامتر های شبکه ایجاد می کنیم و سپس به کمک این بردار به پیش بینی وزن های شبکه ای با این ابر پارامتر ها می پردازیم. در این حالت نیازی به آموزش شبکه از ابتدا نداریم و مستقیما به کمک وزن های پیش بینی شده به ارزیابی شبکه می پردازیم.

راهکار چهارم: One-Shot

در این حالت به جای این که برای هر معماری یک شبکه مجزا از ابتدا اموزش ببیند ما تمام حالت های هر ابرپارامتر را در یک شبکه جای میدهیم و سپس به آموزش این شبکه بزرگ می پردازیم. بعد از همگرایی به بررسی هر مسیر (مجموعه ابر پارامتر) پرداخته و بهترین را انتخاب می کنیم. ایراد اصلی این روش حجم حافظه بسیار زیادی است که نیاز دارد.

۳ الف

برای حل مشکل عدم توازن کلاس ها می توان:

Downsampling:

تعداد داده های مثبت را کاهش داد تا توازن بهتری در دیتا مشاهده کنیم.

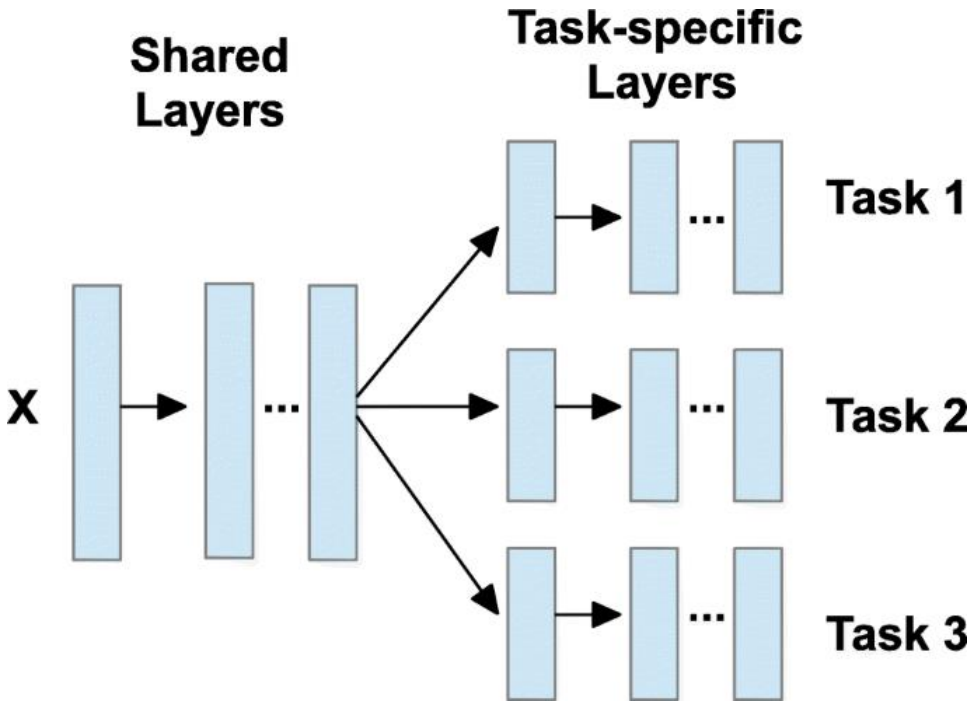
Upsampling:

تعداد داده های منفی و خنثی را با تکرار داده ها افزایش داد.

Weighted Class Learning:

استفاده از کلاسیفایر هایی که به ما اجازه می دهند برای هر کلاس وزن در نظر بگیریم و متناسب با وزن هر کلاس یادگیری انجام شود. این کلاسیفایر ها یک پارامتری از وزن کلاس ها را در تابع ضرر خود استفاده می کنند.

همچنین می توان از دانش درون تسک دیگر که Topic Classification هست نیز استفاده کرد. به این صورت که ممکن است برخی تاپیک ها به احساسات منفی یا خنثی بیشتر تمایل داشته باشند. در این حالت می توان در هنگام دسته بندی احساسات، علاوه بر نظرات مشتریان تاپیک پیش بینی شده را نیز به عنوان ورودی در نظر گرفت تا مدل بتواند از اطلاعات بیشتری استفاده کند.



برای یادگیری بازنمایی های مشترک می توانیم از یک شبکه دو بخشی استفاده کنیم. بخش اول شبکه که بین دو تسک به اشتراک گذاشته می شود وظیفه یادگیری بازنمایی های مشترک را دارد و بخش دوم که دو ماژول متفاوت (متناظر با هر تسک) است وظیفه یادگیری بازنمایی های خاص آن تسک را دارد. برای اینکه بخش مشترک بتواند بازنمایی های مشترک خوبی را بیاموزد میتوانیم از یک تابع ضرر ترکیبی استفاده کنیم که ضرر حاصل از دو تسک را ترکیب کرده و به بخش مشترک می دهد. باید دقت کرد که در هنگام آموزش، در هر بچ داده های متناظر با هر دو تسک موجود باشد. در غیر این صورت شبکه نسبت به یک تسک تمایل بیشتری پیدا می کند.

۳ ج

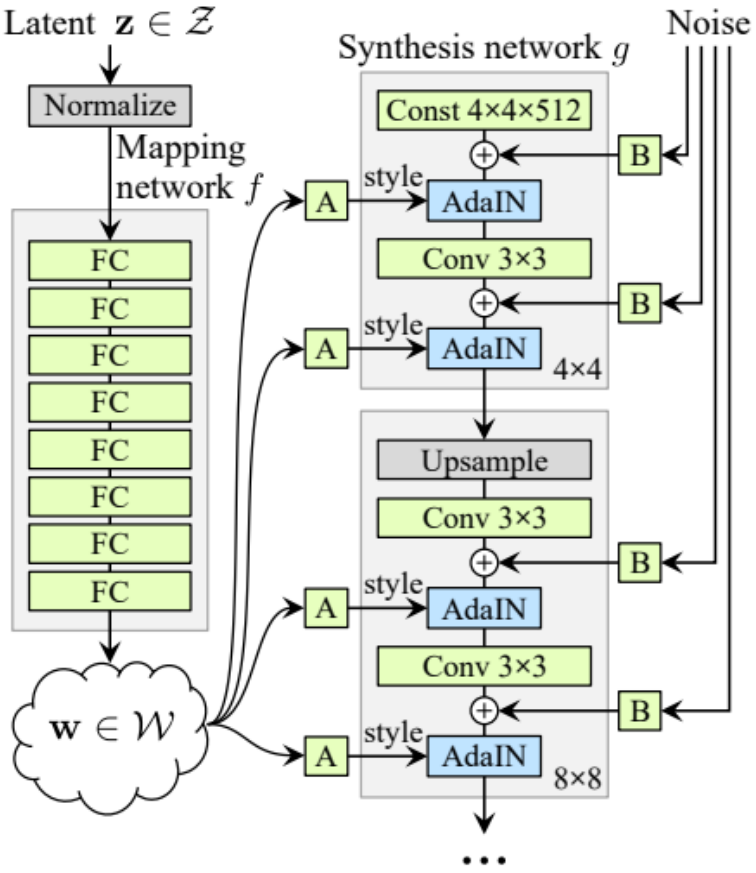
مدل های زبانی از پیش آموخته شده یک دانش کلی از ساختار زبان را در درون خود دارند. به همین جهت استفاده از آن ها در بخش اشتراکی (بخش اول) می تواند سرعت و تعمیم بهتری را برای ما داشته باشد. با تنظیم دقیق این مدل در دوره آموزش می توانیم علاوه بر دانش کلی زبانی، دانش خاص این دو تسک خود را در این مدل ها تزریق کنیم که به بهبود عملکرد نهایی سیستم ما منجر می شود. از طرفی استفاده از این مدل ها به جای آموزش مدل اشتراکی از ابتدا، مزیت سرعت بیشتر و نیاز به داده کمتر را با خود دارد.

۳ د

در تسک تحلیل احساسات، چالش تعداد بیشتر داده های مثبت باعث می شود که استفاده از معیار Accuracy عملا مفید نباشد. زیرا در صورتی که شبکه ما به سمت داده های مثبت متمایل شود یا در حالت افراطی تمام نظرات را مثبت اعلام کند، به دلیل وجود تعداد داده های زیاد مثبت دقت بالایی خواهد داشت که عملا صحیح نیست. در این حالت استفاده از معیار F-1 بهتر است. این معیار با ترکیب Precision Recall از در نظر گرفتن تمایل به سمت کلاس مثبت پرهیز می کند و معیار دقیق تری است. می توانیم این معیار را برای هر کلاس اندازه گرفته و سپس به کمک مکانیزم های Macro AVG و Micro AVG آن ها را ترکیب کنیم.

از طرف دیگر به دلیل عدم توازن کلاس ها، استفاده از AUC-ROC برای یافتن نقطه مناسب برای جدا سازی کلاس ها می تواند مناسب باشد.

در تسک طبقه بندی موضوعات به دلیل توازن کلاس ها دستمان برای انتخاب معیار ارزیابی باز تر است. در این حالت می توانیم از Accuracyیا F1 برای هر کلاس استفاده کنیم. سپس به کمک میانگین MIcro یاMacro، کل سیستم را ارزیابی کنیم.



(b) Style-based generator

معماری ماژول mapping network متشکل از ۸ لایه Fully Connected پشت سر هم است. که اندازه ورودی و خروجی هر لایه ۵۱۲ است. این لایه ها بردار latent نرمالایز شده را دریافت می کنند. هدف استفاده از این شبکه این است که بتوان یک راهکار برای برداشت نمونه هایی از هر استایل داشت و تصاویری متناظر با آن استایل تولید کرد. در این حالت با تغییر بخشی از latent میتوان استایل تصویر را تغییر داد.

۴ ب

در Gan های سنتی، latent از طریق یک لایه ورودی در شبکه اعمال می شد. اما در stylegan این لایه ورودی کاملا کنارگذاشته می شود و latent در ابتدا به یک نگاشت غیر خطی (mapping net) وارد می شود و سپس خروجی این نگاشت به شبکه سنتر وارد می شود.

۴ ج

این ماژول برای نرمالایز کردن هر نقشه ویژگی به صورت مجزا و جدایی از سایر نقشه ها استفاده می شود. این ماژول از اسکیل و بایاسی متناظر با بردار ویژگی استفاده می کند. این نرمالیزشن بعد از هر عملگر کانولوشن استفاده شده است و به کمک بردار استایل کنترل می شود.

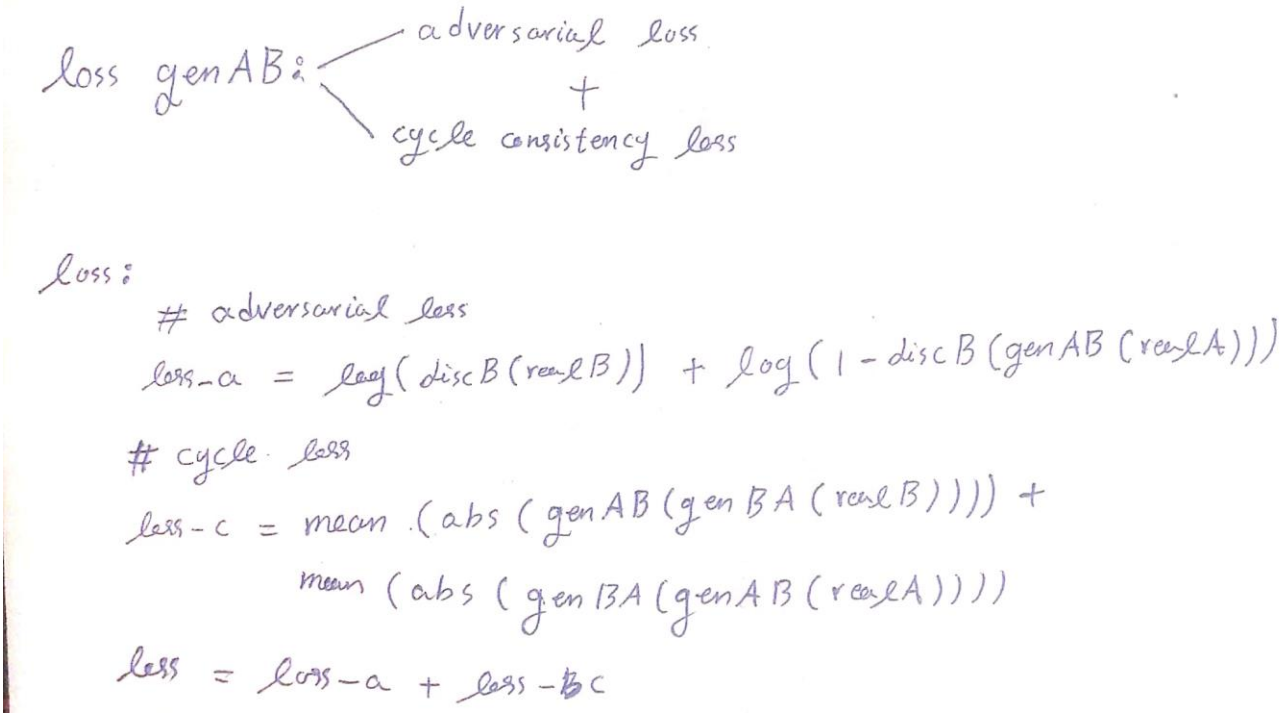
۴ د

ماژول affine transfomr وظیفه ایجاد استایل ها بر اساس خروجی maaping net را دارد. به این صورت که بعد از وارد کردن latent به mapping net خروجی آن به کمک affine transform به استایل تبدیل می شود. این استایل ها وظیفه کنتر AdaIN بعد از هر لایه کانولوشنی را دارند.

۴ ه

استایل میکسینگ یکی از قابلیت های StyleGAN است که به ما اجازه میدهد که استایل تصاویر مختلف را با یکدیگر ترکیب کنیم. برای این کار از مکانیزم Mixing regularization استفاده می شود. این مکانیزم دو latent مختلف را در طول آموزش با یکدیگر ترکیب می کند به این شکل که در طول آموزش شبکه سنتز کننده به صورت رندوم بین این دو latent سوییچ می کند. این کار به ما اجازه تولید تصاویری را میدهد که استایل دو تصویر را با هم ترکیب کرده اند.

۵



۶ الف

آپتونا یک کتابخانه پایتون است که به ما اجازه می دهد به جست و جوی پارامتر ها بپردازیم. این کتابخانه وابستگی به کتابخانه ماشین لرنینگ دیگر ندارد و شما برای پیدا کردن ابرپارامتر های هر مدلی در پایتورچ،تنسورفلو یا سایر کتابخانه ها می توانید از آن استفاده کنید. به کمک این کتابخانه با تعریف بازه های مشخص برای هر ابر پارامتر،می توانید ترکیبات مختلف که به صورت تصادفی ایجاد شده اند را تست کنید و بهترین ترکیب را پیدا کنید. این کتابخانه به شما اجازه میدهد که از هسته های مختلف به صورت موازی برای جست و جوی خود استفاده کنید.

۶ ب: بخش های مهم کد توضیح داده شده است. همه کد در نوت بوک قابل مشاهده است نه اینجا

برای انجام این تمرین از این مخزن کمک گرفته شده است.
https://github.com/elena-ecn/optuna-optimization-for-PyTorch-CNN/blob/main/optuna_optimization.py

برای حل این تمرین چند ماژول و تابع به شکل زیر تعریف کردیم:

class CNN(nn.Module): کلاسی برای ایجاد مدل کانولوشنی

def __init__(self, trial, num_conv_layers, num_dense_layers, num_filters, num_neurons):

super(CNN, self).__init__()

input_size = 32 تصاویر دیتاست ما ۳۲ در ۳۲ هستند

kernel_size = 3 به صورت پیش فرض مقدار اندازه کرنل لایه های کانولوشن را ۳ در نظر گرفته ایم

define the convolutional layers

self.conv_layers = nn.ModuleList([nn.Conv2d(3, num_filters[0], kernel_size=(3, 3))]) لیستی برای نگه داری لایه های کانولوشن

out_size = input_size - kernel_size + 1 در این متغیر اندازه نقشه فعال سازی را بعد از اعمال هر لایه کانولوشن ذخیره می کنم

for i in range(1, num_conv_layers): این حلقه یکی یکی لایه های کانولوشن را با توجه به ابرپارامتر های انخاب شده در ازمایش فعلی ایجاد می کمند

conv = nn.Conv2d(in_channels=num_filters[i-1], out_channels=num_filters[i], kernel_size=(3, 3))

self.conv_layers.append(conv)

out_size = out_size - kernel_size + 1

size of flattened features from convs

self.out_feature = num_filters[num_conv_layers-1] * out_size * out_size در اینجا اندازه بردار ویژگی را حساب می کنیم

```
# define the dense layers لایه های شبکه عصبی عادی را اینجا تعریف کرده ایم
self.dense_layers = nn.ModuleList([nn.Linear(self.out_feature, num_neurons[0])])
for i in range(1, num_dense_layers):
dense = nn.Linear(num_neurons[i-1], num_neurons[i])
self.dense_layers.append(dense)

def forward(self, x): تابع حرکت رو به جلو شبکه عصبی
# applying conv layers این تابع ابتدا به ترتیب لایه های کانولوشن را روی تصویر اهمال می کند
for i, conv_i in enumerate(self.conv_layers):
x = F.relu(conv_i(x))

# flatten the cnn features نقشه ویژگی جاصل از اخرین کانولوشن را فلت می کنیم
x = x.view(-1, self.out_feature)

# applying dense layers لایه های شبکه عصبی عادی را روی بردار ویژگی اعمال می کنیم
for i, dense_i in enumerate(self.dense_layers):
if i == len(self.dense_layers) - 1: # last layer: log_softmax
x = F.log_softmax(dense_i(x), dim=1)
else: # non-last layers: ReLU
x = F.relu(dense_i(x))

return x
```

این تابع تابع هدف کتابخانه است که به کمک این تابع ازمایش ها را ایجاد و انجام میدهیم :def objective(trial)

ابر پارامتر های مربوط به مدل را در اینجا ایجاد می کنیم # model hyperparams value ranges

```
num_conv_layers = trial.suggest_int('num_conv_layers', 1, 4)

num_dense_layers = trial.suggest_int('num_dense_layers', 1, 3)

num_filters = [int(trial.suggest_float(f'num_filter_conv{i}', 8, 64, step=8)) for i in
range(num_conv_layers)]

num_neurons = [int(trial.suggest_float(f'num_neuron_dense{i}', 8, 64, step=8)) for i in
range(num_dense_layers-1)]

num_neurons += [100]
```

ابرپارامتر های مربوط به بهینه ساز را اینجا درست می کنیم # optimizer hyperparams value ranges

```
optimizer_name = trial.suggest_categorical('optimizer_name', ['Adam', 'RMSprop', 'SGD'])

lr = trial.suggest_float('lr', 1e-3, 1e-2, log=True)
```

با ابرپارامتر های ایجاد شده مدل و بهینه ساز را ایجاد می کنیم # init optimizer and model based on suggested params

```
model = CNN(trial, num_conv_layers, num_dense_layers, num_filters, num_neurons)

model = model.to(device)
```

از ساختار مچکیس برای انتخاب بهینه ساز مناسب استفاده کرده ایم :match optimizer_name

```
case 'Adam':

optimizer = torch.optim.Adam(model.parameters(), lr)

case 'RMSprop':

optimizer = torch.optim.RMSprop(model.parameters(), lr)

case 'SGD':

optimizer = torch.optim.SGD(model.parameters(), lr)
```

```
# training loop اینجا جلقه اموزش مدل را نوشته ایم
for epoch in range(n_epochs):
model.train() # set model to training mode مدل را در حالت اموزش قرار میدهیم
از اینجا به بعد هرخط علاوه بر کد توضیح دارد.
for X, y in train_loader: # load a training batch
# clear prev batch grads
optimizer.zero_grad()
# send X and y to suitable device for model
X = X.to(device)
y = y.to(device)
# forward
p = model(X)
# compute and propagate loss through network
loss = F.nll_loss(p, y)
loss.backward()
optimizer.step()

# set model to evaluation mode
model.eval()
num_correct = 0
# turn off gradient calculation
with torch.no_grad():
for X, y in test_loader: # load a testing batch
```

```
# send X and y to suitable device for model
X = X.to(device)
y = y.to(device)
# forward
p = model(X)
# finding max value in each row, return indexes of max values
pred = p.data.max(1, keepdim=True)[1]

# count correct predictions
num_correct += pred.eq(y.data.view_as(pred)).sum()

# calculating accuracy
acc = num_correct / len(test_loader.dataset)
```
این قسمت برای پاسخ به قسمت ج این سوال است. اگر قسمت ب مد نظر است این قسمت کامنت شود.
```
# for pruning (stops trial early if not promising)
trial.report(acc, epoch)
# handle pruning based on the intermediate value.
if trial.should_prune():
raise optuna.exceptions.TrialPruned()

return acc
```

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu") در اینجا دستگاه مورد نظر را انتخاب کرده ایم
print(device)
n_epochs = 10 تعداد ایپاک
time_out = 1800 تایم اوت را ۳۰ دقیقه گذاشته ایم
bs_train = 64 بچ سایز اموزش
bs_test = 1024 بچ ساز تست


# loading (downloading) cifar100 datasets دیتاست را لود کرده و ترنسفورم اولیه ای روی ان اعمال می کنیم.
transform = torchvision.transforms.Compose([
torchvision.transforms.ToTensor(),
torchvision.transforms.Normalize((0.1307,), (0.3081,))
])
train_loader = DataLoader(
torchvision.datasets.CIFAR100('.', train=True, download=True, transform=transform),
batch_size=bs_train, shuffle=True
)
test_loader = DataLoader(
```

torchvision.datasets.CIFAR100('.', train=False, download=True, transform=transform),

batch_size=bs_test, shuffle=False

)

create an optuna study to maximize test accuracy

study = optuna.create_study(direction="maximize") یک مطالعه با هدف ماکسیمم سازی ایجاد کرده ایم

study.optimize(objective, n_trials=None, timeout=time_out, show_progress_bar=True, gc_after_trial=True) تابع هدف به همراه تایم اوت را برای بهینه کردن داده ایم.

درخواست ترسیم نوار پیشرفت و پاک کردن آبجکت های بدون استفاده بعد از هر آزمایش را نیز داده ایم. به هدف کاهش مصرف حافظه

۶ ج

این روش برای کاهش تعداد عملگر های مدل استفاده می شود. در مدل های عمیق تعداد زیادی پارامتر که کاربرد چشمگیری در ایجاد نتیجه نهایی ندارند وچود دارد و صرفا پروسه پیش بینی را کند و مدل را حجیم می کنند. هدف از این روش شناسایی و از کار انداختن این پارامتر ها است. این روش به دو صروت ساختمند و غیر ساختمند انجام می شود. در حالت غیر ساختمند ما تعدادی از پارامتر ها را انتخاب می کنیم و آن ها را برابر با صفر قرار می دهیم. در اکثر سیستم ها این پارامتر ها به صورت دستور no-ops در می آیند که به پردازنده می گوید نیازی به انجام هیچ عملیاتی نیست. به این شکل سرعت اجرا افزایش میابد. در حالت ساختمند با برخی از نورون ها یا لایه ها یا فیلتر ها را به طور تمام و کمال حذف می کنیم. این باعث تغییر ساختار و اندازه ها در شبکه می شود اما از روش قبل سریع تر است زیرا دیگر هیچ دستوری به پردازنده ارسال نمی شود!

در قسمت ب ما بخشی از کد را که به این روش اختصاص دارد مشخص کرده ایم.

با پرونینگ: ۲۹.۳

بدون پرونینگ: ۲۵.۱

نتایج ما نشان می دهد که استفاده از پرونینگ به بهبود عملکرد کلی مدل منجر شده است. به طور کلی گفته می شود که اگر میزان پرونینگ به اندازه مناسب باشد میتواند به شکل یک

رگولایزر عمل کند و تعمیم پذیری مدل را بهبود بخشد. اما استفاده بیش از حد از این روش میتواند به کاهش دقت مدل بیانجامد.

۶ د

انتخاب بهینه ساز مناسب به نظر بنده مهم ترین بخش انتخاب ابرپارامتر است. زیرا بهینه ساز است که مسولیت هدایت مدل به سمت نقطه بهینه را بر عهده دارد و سرعت رسیدن به این نقطه و کیفیت این نقطه کاملا وابسته به بهینه ساز است. در مرحله بعد انتخاب نرخ آموزش خیلی اهمیت دارد. تعداد لایه های کانولوشنی نیز اهمیت دارند زیرا این لایه ها وظیفه استخراج ویژگی ها را دارند و اگر ویژگی ها به خوبی استخراج شوند، دسته بندی بهتر می شود. در نهایت تعداد لایه های معمولی (دنس) و سایز این لایه ها و لایه کانولوشن قرار دارد.

۷: بخش های مهم کد توضیح داده شده است. همه کد در نوت بوک قابل مشاهده است نه اینجا

```
# Define the training function

def train_word2vec(corpus, window_size, embedding_dim,
num_epochs, learning_rate):

# Preprocess the corpus and build the vocabulary

در این بخش به حذف نقطه و علامت تعجب پرداخته ایم

کورپس را به جملات می شکنیم

sentences = [sent.strip().replace('!', '') for sent in corpus.split('.')]

sentences = [f'<bos> {sent} <eos>' for sent in sentences]

vocab = []
```

for sent in sentences: به کمک این حلقه و تابع زیر جملات را به توکن ها تبدیل می کنیم

vocab += word_tokenize(sent)

vocab = list(set(vocab)) یک لیست از توکن های یکتای بکار رفته در کورپس

stoi = {v:k for k, v in enumerate(vocab)} دیکشنری برای تبدیل توکن به ایندکس

itos = {k:v for k, v in enumerate(vocab)} برعکس قبلی برای تبدیل ایندکس به توکن

Create the target-context word pairs

training_pairs = []

for sent in sentences: به کمک این حلقه زوج ها را در می اوریم

sent_toks = word_tokenize(sent) ابتدا جمله مان را به لیستی از توکن تبدیل می کنیم

for i in range(len(sent_toks)-window_size): به ازای هر موقعیت پنجره را قرار داده و توکن میانی را به عنوان تارگت و سایر توکن ها را کانتکست در نظر میگیریم

window_toks = sent_toks[i:i+window_size]

target = window_toks[window_size//2]

for context in window_toks:

if context == target:

continue

training_pairs.append((torch.tensor(stoi[target]), torch.tensor(stoi[context])))

Initialize the Word2Vec model

device = torch.device('cuda') if torch.cuda.is_available() else torch.device('cpu')

```
model = Word2Vec(len(vocab), embedding_dim)
model = model.to(device)
# Define the loss function and optimizer
loss_func = nn.CrossEntropyLoss() از کراس انتروی استفاده کرده ایم
optimizer = optim.AdamW(model.parameters(), learning_rate)
for epoch in range(num_epochs): حلقه اصلی اموزش
total_loss = 0.0
for target_word, context_word in training_pairs:
# Zero the gradients
optimizer.zero_grad()
# Forward pass
target_word = target_word.to(device)
context_word = context_word.to(device)
target_emb, context_emb = model(target_word, context_word)
# Compute the loss
loss = loss_func(target_emb, context_emb)
# Backward pass
loss.backward()
# Update the model parameters
optimizer.step()
# Accumulate the loss
total_loss += loss.item()
```

```
# Print the average loss for the epoch
print(f"Epoch {epoch+1} Loss: {round(total_loss/len(training_pairs), 3)}")
# Return the trained Word2Vec model
return model, vocab, stoi
```

```
def most_similar(word, embeds, stoi):  این تابع شبیه ترین کلمه به کلمه گفته شده را پیدا می کند
word_embed = embeds[stoi[word]]
best_word = ''
best_score = -1
for w in stoi.keys():  کلمه داده شده به کمک این حلقه با تمام کلمات یکتای ما مقایسه می شود
i = stoi[w]
if w == word:
continue
```

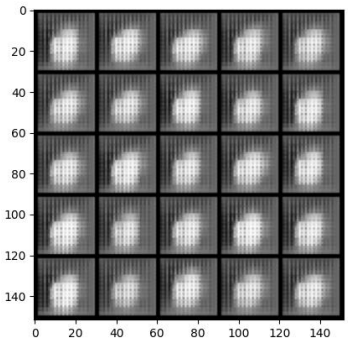
```
# cosine sim معیار شباهت کسینوسی
s = np.dot(word_embed, embeds[i]) / (np.linalg.norm(word_embed, 2)
* np.linalg.norm(embeds[i], 2))
if s > best_score: شبیه ترین کلمه انتخاب می شود
best_score = s
best_word = w

return best_word
```

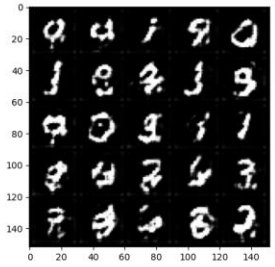
۸: این تمرین به صورت کامل کردنی بوده است. توضیحات درون کد نوشته شده است و ما صرفا به تکمیل کد پرداخته ایم.

نتایج:

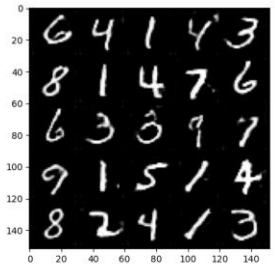
بعد از ۵۰۰ دور

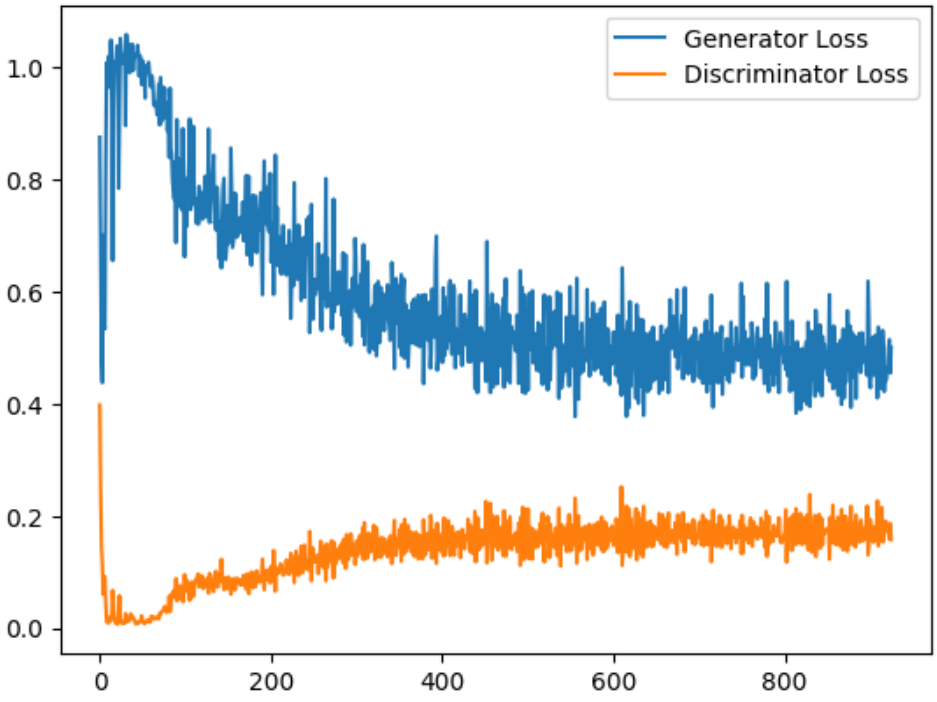


بعد از ۳۰۰۰ دور



بعد از ۱۸۰۰۰ دور





همانگونه که از نمودار مشخص است، با گذشت زمان مولد ما به قدرت بهتری در تولید تصویر رسیده است و تصاویری را ایجاد می کند که مدل مکشف در تشخیص آن دچار اشتباه می شود و لاس آن افزایش میابد.