به نام خدا

درس:

سیستم های هوشمند

پروژه نهایی

پارسا قدیمی

810199468

رضا مومنی

810199

بخش اول)

قبل از پیاده سازی و تحلیل نتایج به سراغ تعریف gan میرویم.

GAN مخفف عبارت Generative Adversarial Network است. این یک ساختار از مدل های یادگیری ماشین است که از دو شبکه عصبی تشکیل شده است: 1) مولد(generator) و 2) تمایز(discriminator). GAN ها برای تولید داده های جدید طراحی شده اند که شبیه به مجموعه داده های آموزشی است.

شبکه مولد نویز تصادفی را به عنوان ورودی دریافت می کند و سعی می کند داده های مصنوعی مانند تصاویر، صدا یا متن تولید کند. هدف آن تولید داده هایی است که از داده های واقعی قابل تشخیص نیستند. در ابتدا، ژنراتور خروجی های تصادفی تولید می کند، اما از طریق آموزش، می آموزد که با تنظیم پارامترهای داخلی خود، داده های واقعی تری تولید کند.

از سوی دیگر، شبکه تشخیص دهنده به عنوان یک طبقه بند باینری عمل می کند. هم داده های واقعی را از مجموعه آموزشی و هم داده های سنتز شده از مولد را به عنوان ورودی می گیرد و سعی می کند بین آنها تمایز قائل شود. تمایز دهنده آموزش داده شده است تا داده های واقعی را به درستی به عنوان واقعی و داده های تولید شده را جعلی طبقه بندی کند.

هدف آن دقت فزاینده در تشخیص داده های واقعی و جعلی است. این دو شبکه در یک فرآیند رقابتی با هم آموزش می بینند. هدف مولد این است که با تولید داده هایی که به اندازه کافی واقعی هستند تا به عنوان واقعی طبقه بندی شوند، متمایز کننده را فریب دهد، در حالی که هدف تمایزکننده این است که در تشخیص داده های واقعی و جعلی بهتر عمل کند.

در طول آموزش، مولد و تشخیص دهنده به طور متناوب به روز می شوند. پارامترهای مولد به گونه ای تنظیم می شوند که توانایی تمایزکننده را برای تمایز بین داده های واقعی و جعلی به حداقل برساند. به طور همزمان، پارامترهای تمایزکننده برای بهبود توانایی آن در طبقه بندی صحیح داده های واقعی و جعلی به روز می شوند. این فرآیند تا جایی ادامه می یابد که مولد داده هایی را تولید می کند که تشخیص آن از داده های واقعی برای تمایزکننده دشوار است.

کاربرد ها :

تولید تصویر مصنوعی : GAN ها می توانند تصاویر واقعی شبیه به عکس های واقعی تولید کنند. این ویژگی در زمینه های مختلفی مانند هنر، سرگرمی و تبلیغات کاربرد دارد. (GAN ها همچنین می توانند برای تولید تصاویر با کیفیت بالا از ورودی های با وضوح پایین یا تکمیل قسمت های از دست رفته تصاویر استفاده شوند.)

انتقال استایل : GAN ها می توانند استایل یک تصویر یا اثر هنری را بیاموزند و آن را بر روی تصویر دیگر اعمال کنند. این امکان ایجاد جلوه های هنری و تبدیل تصاویر به استایل های مختلف را فراهم می کند.

تولید ویدیو: GAN ها می توانند توالی های ویدیویی جدیدی را با گسترش یا تکمیل ویدیوهای موجود ایجاد کنند.

افزایش داده ها: GAN ها می توانند داده های مصنوعی تولید کنند که می تواند برای تقویت مجموعه داده های آموزشی استفاده شود. این به ویژه زمانی مفید است که داده های واقعی کمیاب هستند یا زمانی که نیاز به داده های متنوع تر وجود دارد. GAN ها می توانند نمونه های بیشتری تولید کنند که به بهبود عملکرد و تعمیم مدل های یادگیری ماشین کمک می کند.

و ....

همچنین extention هایی از gan نیز وجود دارد مانند :

GANهای مشروط (cGANs): در cGANها، هم مولد و هم متمایزکننده اطلاعات مشروط اضافی را به عنوان ورودی دریافت می کنند که امکان تولید هدفمند را فراهم می کند. به عنوان مثال، در سنتز تصویر، ژنراتور را می‌توان بر روی ویژگی‌های خاصی مانند رنگ مو، ژست یا پس‌زمینه مشروط کرد و در نتیجه خروجی کنترل‌شده‌تر و قابل تنظیم‌تری به دست آورد.

WGAN :Wasserstein GAN (WGAN) یک تابع از دست دادن متفاوت را بر اساس فاصله Wasserstein معرفی می کند. این فرمول به مسائل ناپایداری و فروپاشی حالت GAN های سنتی می پردازد و آموزش پایدارتر و ویژگی های همگرایی را ارتقا می دهد.

و .....

پیاده سازی:

در ابتدا پارامتر ها (رندوم سید ، تعداد ایپاک ، نرخ یادگیری ، سایز بتچ و تعداد تست) را تعریف کردیم :

```
# Set random seed for reproducibility
torch.manual_seed(42)

# Hyperparameters
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
batch_size = 100
learning_rate = 0.0002
num_epochs = 81
num_test_samples = 32
```

سپس دیتا ست cifar را لود کردیم و پیش پردازشش کردیم :

```
# Data loading and preprocessing
transform = transforms.Compose([
    transforms.Resize(64),
    transforms.ToTensor(),
    transforms.Normalize((.5, .5, .5), (.5, .5, .5))
])

cifar10_dataset = datasets.CIFAR10(root='./data', train=False, transform=transform, download=True)
data_loader = torch.utils.data.DataLoader(cifar10_dataset, batch_size=batch_size, shuffle=True)
```

بخش پیش پردازش شامل تبذیل تصاویر از سایز 32*32 به سایز 64*64 ، نرمالایز کردن و تبدیل به تنسور است.

حال مدل جنریتور را تعریف کردیم :

```
# Generator model
class Generator(nn.Module):
    def __init__(self):
        super(Generator, self).__init__()
        self.linear = nn.Linear(100, 1024*4*4)
        self.deconv = nn.Sequential(
            nn.ConvTranspose2d(1024, 512, kernel_size=4, stride=2, padding=1, bias=False),
            nn.BatchNorm2d(512),
            nn.ReLU(inplace=True),
            nn.ConvTranspose2d(512, 256, kernel_size=4, stride=2, padding=1, bias=False),
            nn.BatchNorm2d(256),
            nn.ReLU(inplace=True),
            nn.ConvTranspose2d(256, 128, kernel_size=4, stride=2, padding=1, bias=False),
            nn.BatchNorm2d(128),
            nn.ReLU(inplace=True),
            nn.ConvTranspose2d(128, 3, kernel_size=4, stride=2, padding=1, bias=False),
            nn.Tanh()
        )

    def forward(self, x):
        x = self.linear(x)
        x = x.view(x.size(0), 1024, 4, 4)
        x = self.deconv(x)
        return x
```

ابتدا یک نویز تصادفی را دریافت میکند و ان را به یک تنسور تبدیل میکند سپس این تنسور را به یک شبکه شامل کانولشون ، نورمالیزیشن و توابع فعالساز میدهد.

اکنون به سراغ discriminator میرویم :

```
# Discriminator model
class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()
        self.conv = nn.Sequential(
            nn.Conv2d(3, 128, kernel_size=4, stride=2, padding=1, bias=False),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Conv2d(128, 256, kernel_size=4, stride=2, padding=1, bias=False),
            nn.BatchNorm2d(256),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Conv2d(256, 512, kernel_size=4, stride=2, padding=1, bias=False),
            nn.BatchNorm2d(512),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Conv2d(512, 1024, kernel_size=4, stride=2, padding=1, bias=False),
            nn.BatchNorm2d(1024),
            nn.LeakyReLU(0.2, inplace=True)
        )
        self.output = nn.Sequential(
            nn.Linear(1024*4*4, 1),
            nn.Sigmoid()
        )

    def forward(self, x):
        x = self.conv(x)
        x = x.view(x.size(0), 1024*4*4)
        x = self.output(x)
        return x
```

Discriminator به عنوان ورودی تصویر دریافت میکند و در خرجی یک احتمال میدهد که نشان دهنده واقغی یا مصنوعی بودن ان است.

داخل مدل discriminator نیز از یک سبکه سامل توابع فعال ساز ، نورمالیزیشن و کانولوشن استفاده شده است.

سپس وزن ها را مقداردهی اولیه کردیم :

```
# Initialize generator and discriminator
generator = Generator().to(device)
discriminator = Discriminator().to(device)

# Weight initialization
def weights_init(module):
    if isinstance(module, (nn.Conv2d, nn.ConvTranspose2d)):
        nn.init.normal_(module.weight.data, 0.0, 0.02)
    elif isinstance(module, nn.BatchNorm2d):
        nn.init.normal_(module.weight.data, 1.0, 0.02)
        nn.init.constant_(module.bias.data, 0)
```

حال باید برای هر جفت مدل لاس فانکشن و متود بهینه سازی تعریف کنیم :

```
generator.apply(weights_init)
discriminator.apply(weights_init)

# Loss function
criterion = nn.BCELoss()

# Optimizers
g_optimizer = optim.Adam(generator.parameters(), lr=learning_rate, betas=(0.5, 0.999))
d_optimizer = optim.Adam(discriminator.parameters(), lr=learning_rate, betas=(0.5, 0.999))
```

در نهایت نیز به سراغ ترین شبکه ، نشان دادن تصائیر مصنوعی ، لاس برای هر ایپاک و پلا لاس بر مبنای ایپاک میرویم :

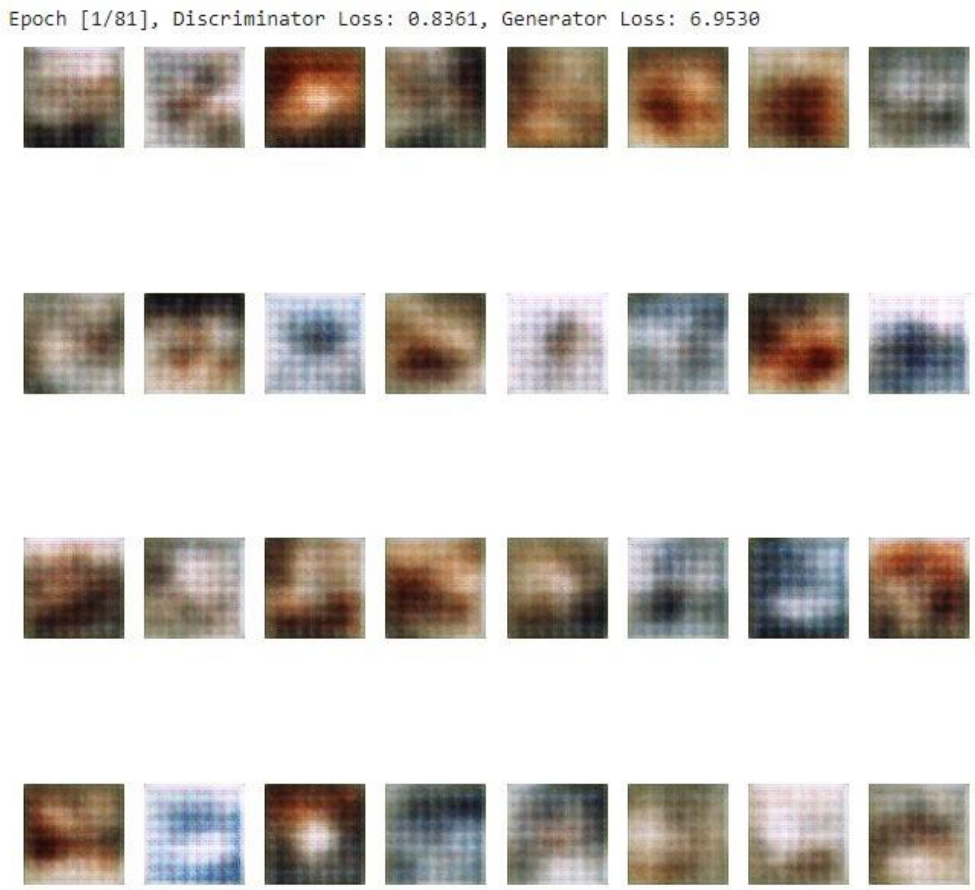
```
# Fixed noise for visualization
fixed_noise = torch.randn(num_test_samples, 100, device=device)

# Training loop
discriminator_losses = []
generator_losses = []

for epoch in range(num_epochs):
    # Initialize loss variables for this epoch
    epoch_discriminator_loss = 0
    epoch_generator_loss = 0

    for i, (real_images, _) in enumerate(data_loader):
        real_images = real_images.to(device)

        # Discriminator forward pass with real images
        real_labels = torch.ones(batch_size, 1, device=device)
        real_outputs = discriminator(real_images)
        d_loss_real = criterion(real_outputs, real_labels)

        # Generator forward pass and discriminator backward pass with fake images
        noise = torch.randn(batch_size, 100, device=device)
        fake_images = generator(noise)
        fake_labels = torch.zeros(batch_size, 1, device=device)
        fake_outputs = discriminator(fake_images.detach())
        d_loss_fake = criterion(fake_outputs, fake_labels)

        d_loss = d_loss_real + d_loss_fake

        discriminator.zero_grad()
        d_loss.backward()
        d_optimizer.step()
```

```
        # Generator forward pass and discriminator backward pass with fake images
        fake_outputs = discriminator(fake_images)
        g_loss = criterion(fake_outputs, real_labels)

        generator.zero_grad()
        g_loss.backward()
        g_optimizer.step()

        # Accumulate losses for this epoch
        epoch_discriminator_loss += d_loss.item()
        epoch_generator_loss += g_loss.item()

    # Average losses for this epoch
    epoch_discriminator_loss /= len(data_loader)
    epoch_generator_loss /= len(data_loader)

    discriminator_losses.append(epoch_discriminator_loss)
    generator_losses.append(epoch_generator_loss)

    # Print progress
    print(f"Epoch [{epoch+1}/{num_epochs}], Discriminator Loss: {epoch_discriminator_loss:.4f}, Generator Loss: {epoch_generator_loss:.4f}")
```

```
    # Generate images
    with torch.no_grad():
        generated_images = generator(fixed_noise).detach().cpu()

    # Display generated images
    if (epoch+1) % 2 == 0 or epoch == 0:
        plt.figure(figsize=(8, 8))
        for j in range(num_test_samples):
            # Denormalize the image
            image = generated_images[j].permute(1, 2, 0)
            image = (image + 1) / 2  # Undo normalization

            plt.subplot(4, 8, j+1)
            plt.imshow(image)
            plt.axis('off')
        plt.show()

# Plotting the loss curve
plt.figure()
plt.plot(range(1, num_epochs+1), discriminator_losses, label="Discriminator Loss")
plt.plot(range(1, num_epochs+1), generator_losses, label="Generator Loss")
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.legend()
plt.savefig("loss_curve.png")
plt.show()
```

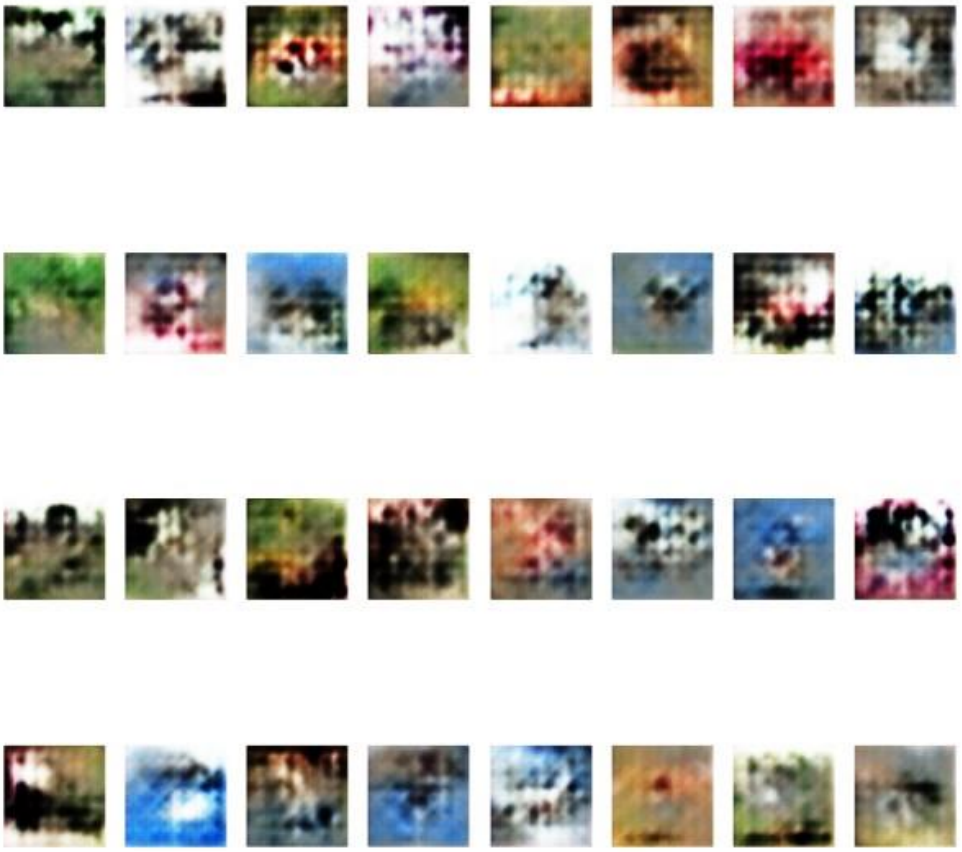
نتایج :

(به علت حجم زیاد نتایج بخشی از ان در ریپورت قرار داده شده است)



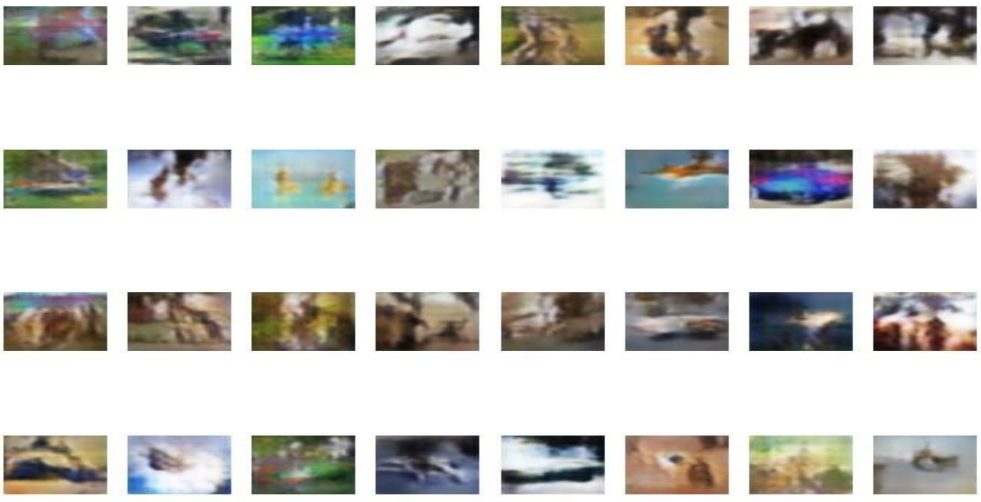
همان طور که از نتایج پیداس در ایپاک های اولیه تصاویر جنریت شده (با استفاده از نویز تصادفی) تصاویر مطلوبی نیستند و همچنین در ایپاک های اولیه discriminator تصاویر جنریت شده را تقریبا به طور کامل درست تشخیص میدهد.(که فیک هستند)

Epoch [4/81], Discriminator Loss: 0.9325, Generator Loss: 2.8463

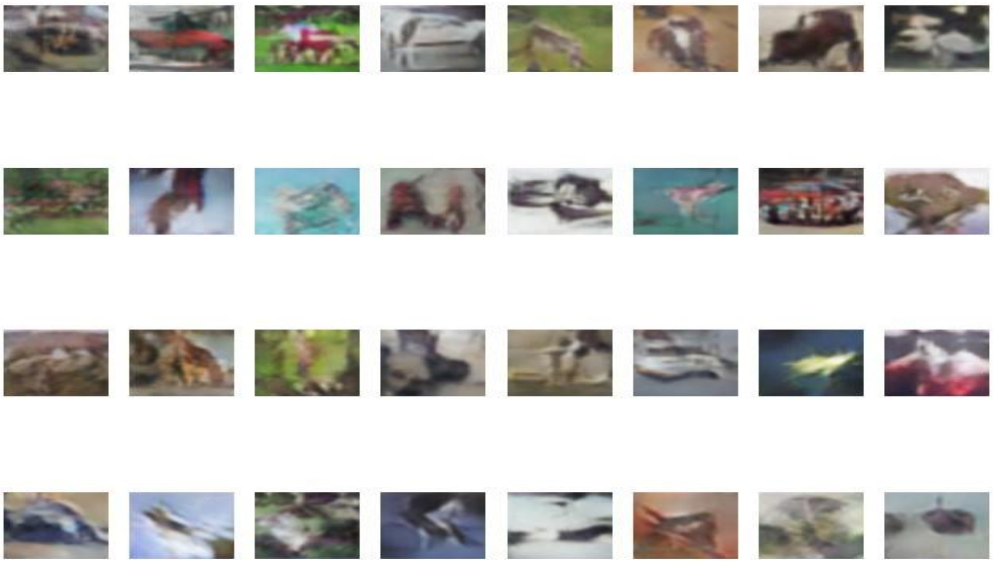


با گذشت هر ایپاک جنریتور یاد میگیرد تا تصاویر بهتری تولید کند در جهت فریب discriminator .

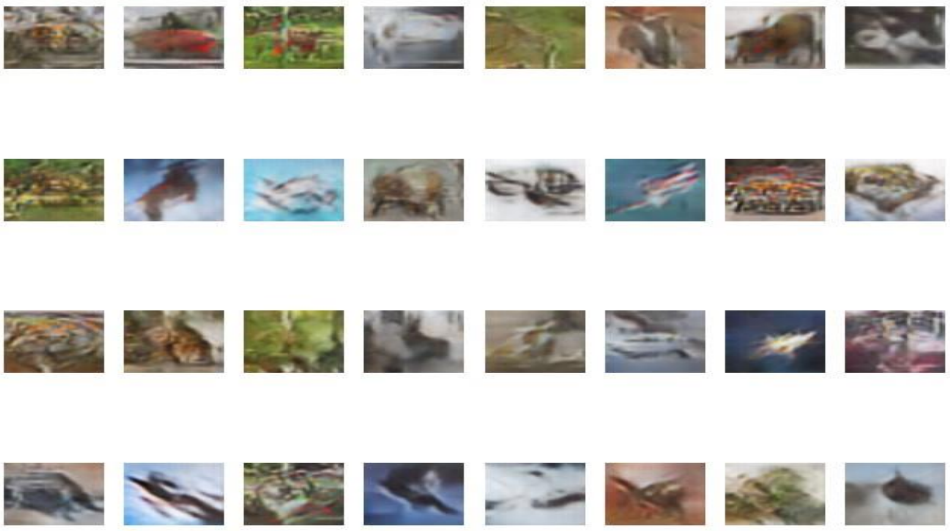
Epoch [30/81], Discriminator Loss: 1.1400, Generator Loss: 2.3692



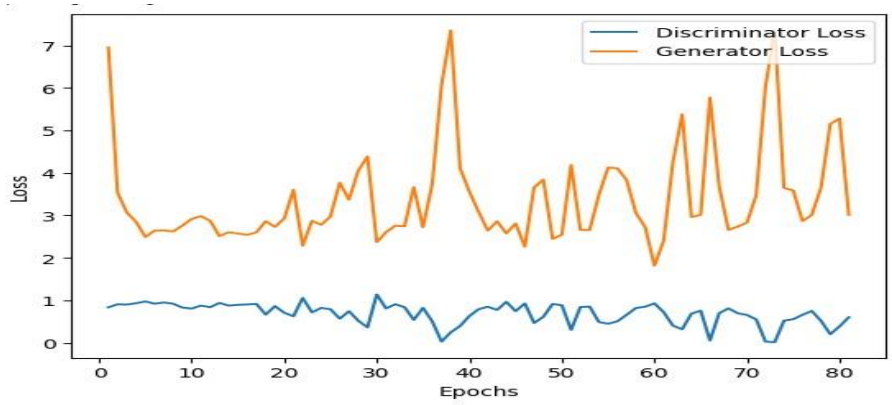
Epoch [68/81], Discriminator Loss: 0.8108, Generator Loss: 2.6586



Epoch [72/81], Discriminator Loss: 0.0290, Generator Loss: 6.0300



حال همان طور که مشخص است تصاویر جنریت شده توسط جنریتور به تصاویر واقعی نزدیک شده اند.



حال با تغییر نرخ یادگیری اثر ان را بررسی میکنیم :

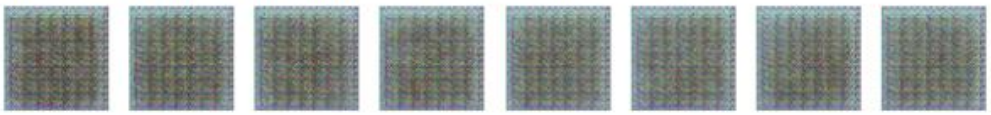
حالت 1)

```
# Set random seed for reproducibility
torch.manual_seed(42)

# Hyperparameters
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
batch_size = 100
learning_rate = 0.000075
num_epochs = 200
num_test_samples = 16
```

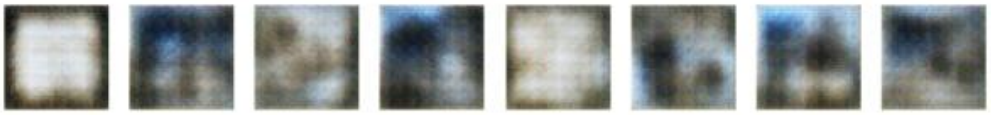
نرخ یادگیری را کوچک کردیم و همچنین تعداد ایپاک را افزایش دادیم (متاسفانه به دلیل ری استارت شدن سشن در کولب تنها تا ایپاک 114 رفتیم)

نتایج :

```
Epoch [1/200], Discriminator Loss: 0.1666, Generator Loss: 5.5667
```



```
Epoch [2/200], Discriminator Loss: 0.2014, Generator Loss: 6.9684
Epoch [3/200], Discriminator Loss: 0.0948, Generator Loss: 7.0529
```

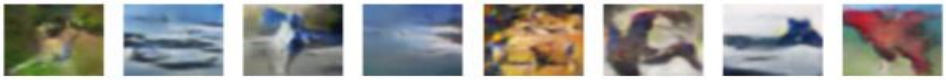


```
Epoch [22/200], Discriminator Loss: 0.5217, Generator Loss: 3.1643
Epoch [23/200], Discriminator Loss: 0.3969, Generator Loss: 3.0581
Epoch [24/200], Discriminator Loss: 0.4713, Generator Loss: 3.1988
```

```
Epoch [25/200], Discriminator Loss: 0.4906, Generator Loss: 3.0607
Epoch [26/200], Discriminator Loss: 0.5543, Generator Loss: 2.9967
Epoch [27/200], Discriminator Loss: 0.4779, Generator Loss: 3.1299
```

```
Epoch [109/200], Discriminator Loss: 0.0214, Generator Loss: 5.8580
Epoch [110/200], Discriminator Loss: 0.0178, Generator Loss: 5.9469
Epoch [111/200], Discriminator Loss: 0.1827, Generator Loss: 5.9211
```

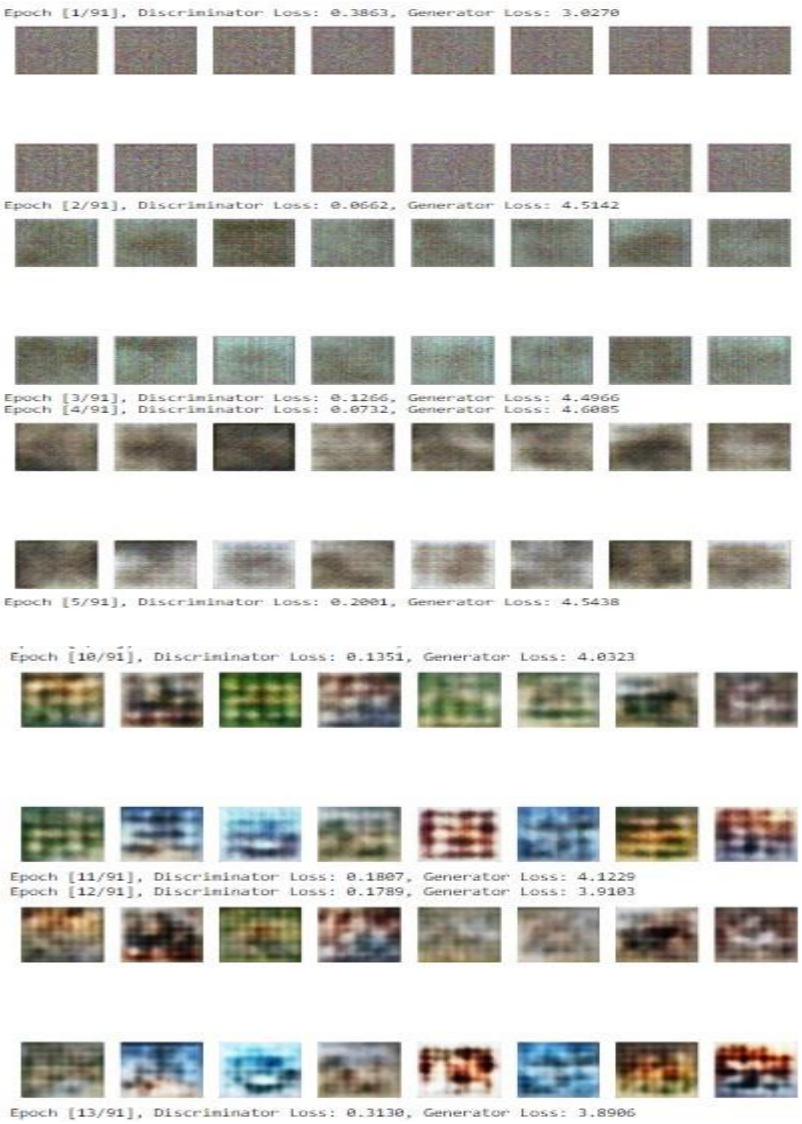
```
Epoch [112/200], Discriminator Loss: 0.2348, Generator Loss: 4.3573
Epoch [113/200], Discriminator Loss: 0.3180, Generator Loss: 3.9795
Epoch [114/200], Discriminator Loss: 0.0578, Generator Loss: 4.8159
```



حالت 2)

```
# Set random seed for reproducibility
torch.manual_seed(42)

# Hyperparameters
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
batch_size = 100
learning_rate = 0.00002
num_epochs = 91
num_test_samples = 16
```

نتایج :



Epoch [52/91], Discriminator Loss: 0.4316, Generator Loss: 2.7998

Epoch [53/91], Discriminator Loss: 0.4624, Generator Loss: 2.7559
Epoch [54/91], Discriminator Loss: 0.2908, Generator Loss: 2.7292

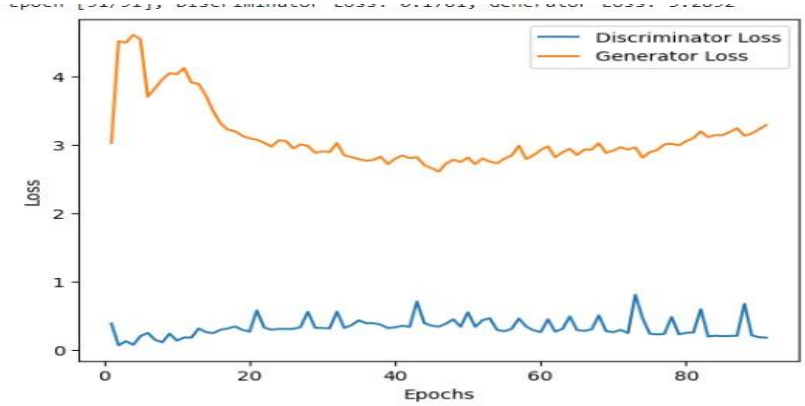
Epoch [55/91], Discriminator Loss: 0.2711, Generator Loss: 2.7968

Epoch [76/91], Discriminator Loss: 0.2241, Generator Loss: 2.9237

Epoch [77/91], Discriminator Loss: 0.2328, Generator Loss: 3.0063
Epoch [78/91], Discriminator Loss: 0.4800, Generator Loss: 3.0147

Epoch [79/91], Discriminator Loss: 0.2241, Generator Loss: 2.9913

Epoch [88/91], Discriminator Loss: 0.6730, Generator Loss: 3.1329



تحلیل :

1) با کوچک کردن نرخ یادگیری عملکرد مدل بهتر و منطقی تر میشود .

2) با افزایش تعداد ایپاک به نتایج بهتری دست میابیم.

3) با عمیق تر کردن مدل ها نتایج بهتر میشوند زیرا پترن ها و الگو های پیچیده تری را یاد میگیرند.

پس برای رسیدن به نتایج بهتر میتوانیم نرخ یادگیری را کاهش دهیم تعداد ایپاک را افزایش دهیم و مدل ها را عمیق تر کنیم .(همچنین میتوان از ساختارهای دیگری به جای gan ساده ای که در اینجا پیاده سازی شده است استفاده کرد مانند cgan ها که تنها یه شرط به مسئله اضافه میکنند ولی همین تغییر کوچک تاثیر به سزایی دارد)

مراجع :

https://www.techtarget.com/searchenterpriseai/definition/generative-adversarial-network-GAN

https://machinelearningmastery.com/what-are-generative-adversarial-networks-gans/

https://machinelearningmastery.com/how-to-develop-a-generative-adversarial-network-for-a-cifar-10-small-object-photographs-from-scratch/

https://medium.com/@emad.bin.abid/generative-adversarial-network-on-cifar-10-8d8deeec1fd7

https://github.com/richardkxu/GANs-on-CIFAR10

https://www.geeksforgeeks.org/generative-adversarial-network-gan/

پردازش زبان طبیعی(NLP)

پردازش زبان طبیعی که از زبان‌شناسی محاسباتی (computational linguistics) تکامل یافته است. این عملیات روش‌های حوزه‌های مختلفی، مانند علوم رایانه(computer science) ، هوش مصنوعی(artificial intelligence) ، زبان‌شناسی (linguistics) و علم داده(data science) ، استفاده می‌کند تا رایانه‌ها بتوانند زبان انسان را به‌دو صورت نوشتاری و شفاهی درک کنند. با تجزیه‌وتحلیل ساختار جمله و دستور زبان، وظایف مختلف NLP به رایانه‌ها این امکان را می‌دهد که بتوانند زبان انسان را بفهمند و بخوانند. وظایف متداول NLP عبارت‌اند از:

- خلاصه‌سازی :(Summarization) این تکنیک خلاصه‌ای از متن‌های طولانی را برای ایجاد خلاصه‌ای مختصر و منسجم از نکات اصلی متن ارائه می‌کند.
- برچسب‌گذاری اجزای گفتار :(Part of Speech Tagging/ PoS) این تکنیک برچسبی را به هر توکن اختصاص می‌دهد که مشخص می‌کند آن توکن چه نقشی را در جمله دارد، برای مثال اسم، فعل، صفت و غیره. این مرحله تجزیه‌وتحلیل معنایی را روی متن بدون ساختار امکان‌پذیر می‌کند.
- دسته‌بندی متن :(Text Categorization) این وظیفه که به‌عنوان طبقه‌بندی متن (Text Classification) نیز شناخته می‌شود وظیفه‌ی تجزیه‌وتحلیل اسناد متنی و طبقه‌بندی آن‌ها را براساس موضوع بر عهده دارد.
- تجزیه‌وتحلیل احساسات :(Sentiment analysis) این وظیفه احساسات مثبت یا منفی را از منابع داده داخلی یا خارجی تشخیص می‌دهد و به ما امکان می‌دهد تغییرات نگرش مشتریان را در طول زمان پیگیری کنیم. معمولاً برای دریافت اطلاعات درمورد نظر مشتریان درباره‌ی محصولات و خدمات استفاده می‌شود. اطلاعاتی که به دست می‌آیند می‌توانند به بهبود ارتباط با مشتریان و بهبود فرایندها و تجربیات کاربری کمک کنند.

تبدیل کلمات به بردار با استفاده از glove

```
[6]  word_to_vec_map = {}
     with open('/kaggle/input/int-sys-main/glove.6B.50d.txt', encoding='utf-8') as f:
         for line in f:
             values = line.split()
             word = values[0]
             coefs = np.asarray(values[1:], dtype='float32')
             word_to_vec_map[word] = coefs

     # Map words in the dataset to their corresponding word vectors
     def words_to_vectors(words):
         return [word_to_vec_map[word] for word in words if word in word_to_vec_map]

     imdb_data = pd.DataFrame(columns=['sentence'])
     imdb_data['sentence'] = df_word['sentence'].apply(words_to_vectors)
```

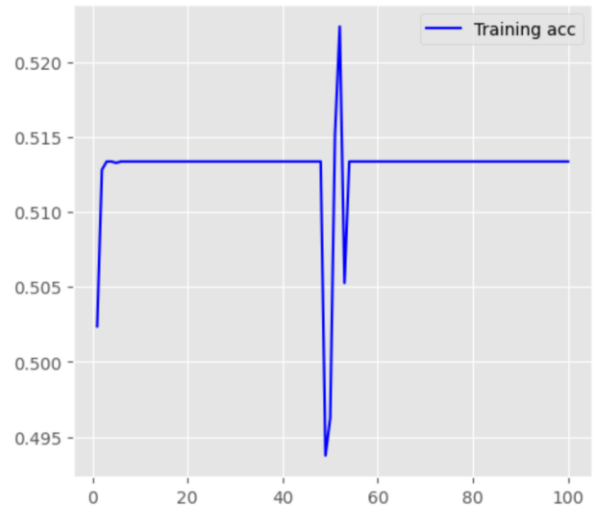
حالت کلی مدل ها

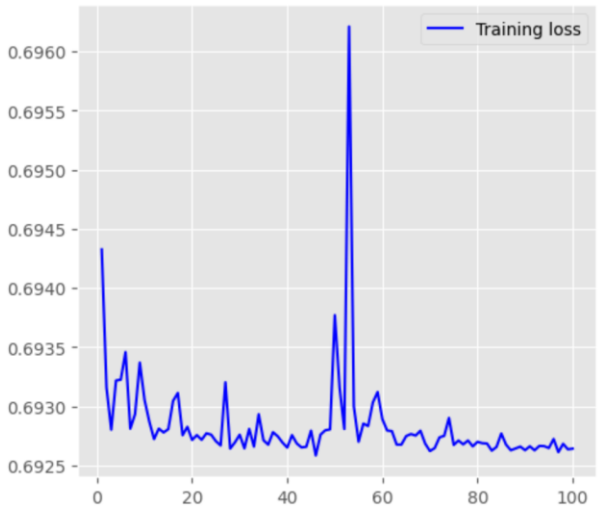
```
optimizer3 = SGD(lr=0.00001, momentum=0.9)
model3 = models.Sequential([
    layers.Dense(max_len, input_shape=(max_len, 50)),
    layers.Dense(128, activation='linear'),
    layers.Dense(64, activation='sigmoid'),
    layers.Dense(32, activation='sigmoid'),
    layers.Dense(1, activation='relu')
    ])
model3.compile(optimizer=optimizer3, loss='binary_crossentropy', metrics=['accuracy'])

model3.summary()
```

```
WARNING:absl:`lr` is deprecated in Keras optimizer, please use `learning_rate` or use th
Model: "sequential_6"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_20 (Dense)            (None, 998, 998)          50898

 dense_21 (Dense)            (None, 998, 128)          127872

 dense_22 (Dense)            (None, 998, 64)           8256

 dense_23 (Dense)            (None, 998, 32)           2080

 dense_24 (Dense)            (None, 998, 1)            33

=================================================================
Total params: 189139 (738.82 KB)
Trainable params: 189139 (738.82 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

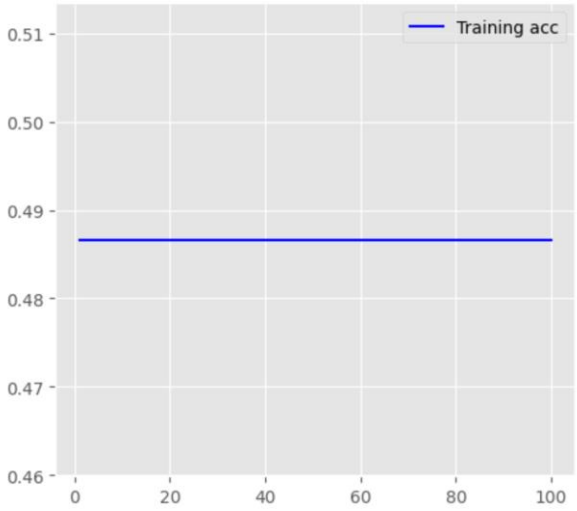
خروجی ها

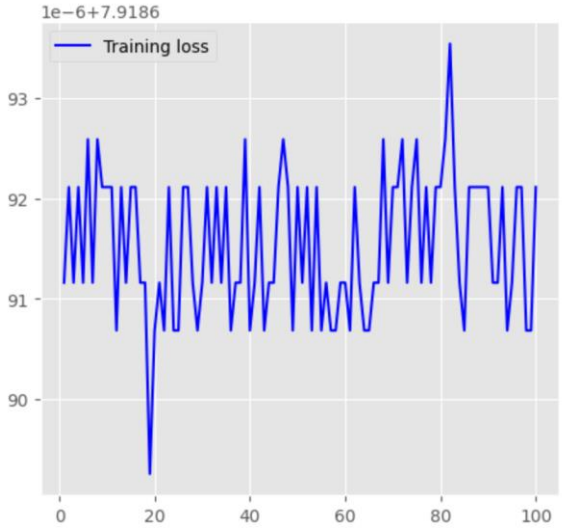
```
plot_history(history2)
```



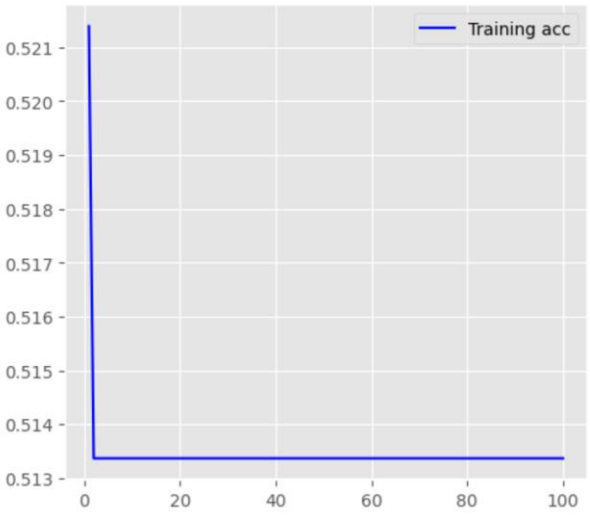


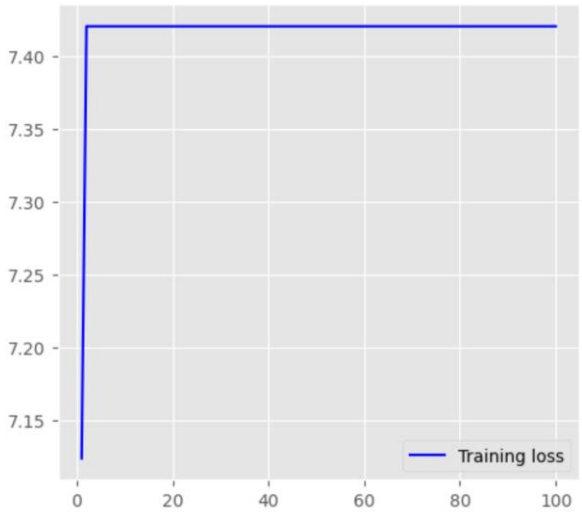

```
plot_history(history3)
```



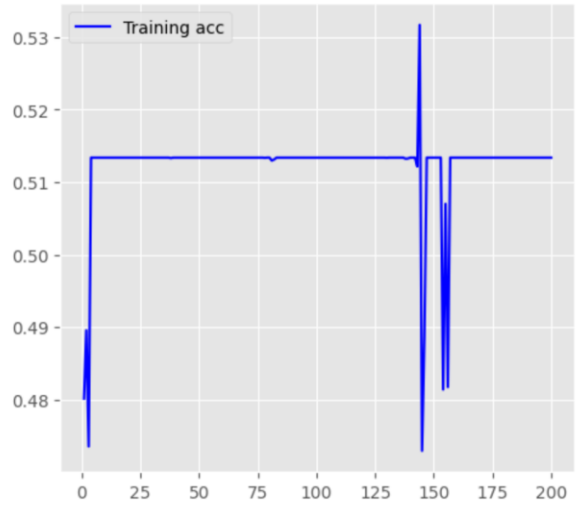


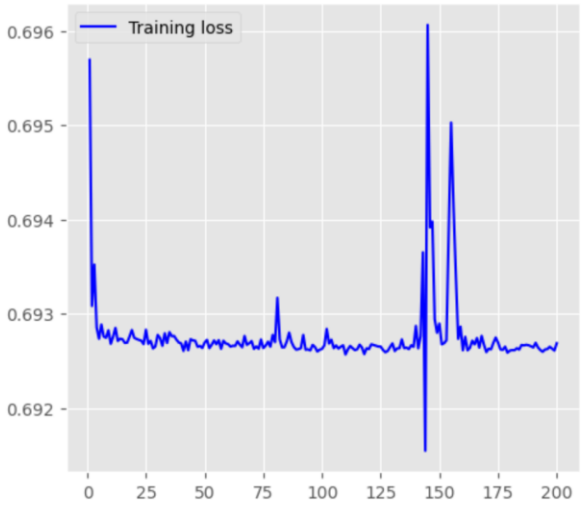
```
plot_history(history4)
```



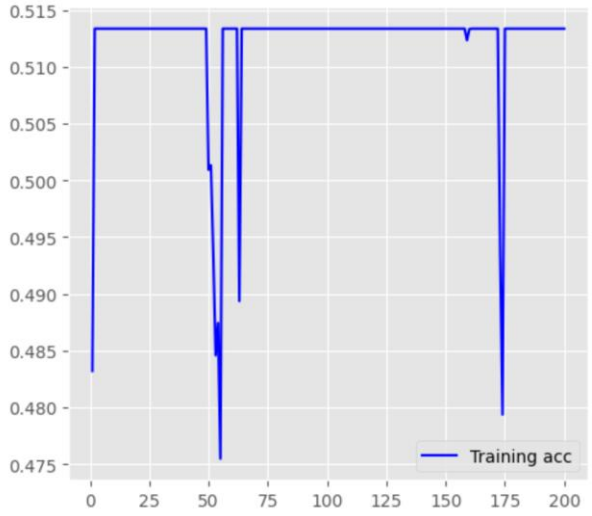


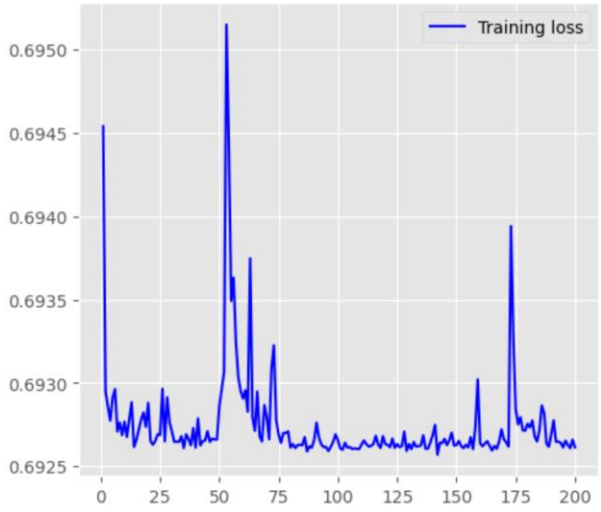

```
plot_history(history)
```





```
plot_history(history1)
```





منبع:

فرادرس

https://realpython.com/python-keras-text-classification/

https://turbolab.in/text-classification-with-keras-and-glove-word-embeddings/