Electronics Science Technology and Application (2024) Volume 11 Issue 4 **ISSN: 2424-8460(O) 2251-2608(P)**

Original Research Article

Application and research of dcgan model in image generation tasks

Lubin Liu

Shenzhen University School of Electronics and Information Engineering, Shenzhen, Guangdong Province, 518000, China

Abstract: This article aims to delve into the principles of generative learning and its applications in image generation tasks by constructing cartoon image generation models and handwritten digit generation models. The experiment is based on the innovation of model architecture using Deep Convolutional Generative Adversarial Network (DCGAN), including adding convolutional layers to enlarge image size and applying Gaussian filtering and average pooling in cartoon image generation tasks, as well as designing conditional generative adversarial networks to generate handwritten images of specific numbers based on labels in handwritten digit generation tasks. The experiment elaborated on the model architecture, data processing, training process, and testing methods, and explored the impact of different network architectures and parameter settings on model performance. The results indicate that the proposed model architecture can effectively improve the quality of image generation, providing new ideas for the research and application of generative learning.

Keywords: Generative learning; DCGAN; Cartoon image generation; Handwritten digit generation; Conditional Generative Adversarial Network

1. Introduction

The aim of this experiment is to gain a deeper understanding of the principles of generative learning and its applications in image generation by constructing two generative models - the cartoon image generation model and the handwritten digit generation model. The main purpose of the experiment is to master the basic theory and implementation techniques of generative models, including Generative Adversarial Networks (GANs) and Variational Autoencoders (VAEs). At the same time, the experiment will explore the impact of different network architectures and parameter settings on model performance, such as layers, convolution kernel size, learning rate, and batch size, to optimize image generation quality. In terms of data processing, we will explore how to collect, clean, and preprocess image data, and evaluate the quality of generated images.

2. Model architecture and innovation

2.1. Cartoon image generation

In the cartoon image generation task, an enhanced generator was designed by adding a convolutional layer to the original DCGAN, enlarging images to 128x128 pixels, and applying Gaussian filtering and average pooling for smoothness and noise reduction. The discriminator retains the DCGAN structure but introduces layer outputs for feature matching loss calculation. The generator creates high-resolution images through multi-layer deconvolution, followed by Gaussian filtering and pooling. The discriminator extracts features through multi-layer convolution, outputting discrimination results and layer feature maps for loss calculation and generator training guidance.

2.2. Handwritten digit generation

For handwritten digit generation, a cGAN generator named Generation-mnist was designed, expanding input from 100 to 110 dimensions with 10 for labels. It has five deconvolutional layers, upsampling input to a 28x28 image, with batch normalization and ReLU/LeakyReLU. Tanh maps outputs to [-1,1]. The conditional discriminator Discretionar_monster, like DCGAN but adjusted, outputs an 11-dimensional Sigmoid vector for multi-classification (0-9). This enhances conditional generation by determining image authenticity and classifying numbers.

3. Experimental core code

3.1. Cartoon image generation

(1) Data section

When preparing the dataset, preprocessing was performed on the images, including randomly flipping 10% of the images horizontally to enhance the model's generalization ability, and standardizing the images. Use transforms Compose combines these transformations, loads image data through ImageFolder, and builds a training data loader DataLoader.

```
print("prepare dataset")
data_dir='../data/resized_anime_face_part'
transform=transforms.Compose([
transforms.RandomHorizontalFlip(p=0.1),
transforms.ToTensor(),
transforms.Normalize(mean=(0.5,0.5,0.5),std=(0.5,0.5,0.5))
])
dset=datasets.ImageFolder(data_dir,transform)
train_loader=torch.utils.data.DataLoader(dset,batch_size=batch_size,shuffle=True)
```

(2) Model section

Generator and discriminator: explanation as above

(3) Training section

Initialize the generator and discriminator models and move them to the specified device.

```
generator=Generator(ngf,output_channels).to(device)
discriminator=Discriminator(ndf,input_channels).to(device)
```

Configure loss function and optimizer, and set learning rate adjustment strategy. Select binary cross entropy loss function and custom loss function, use Adam optimizer to optimize the parameters of generator and discriminator respectively, and set different initial learning rates. At the same time, configure learning rate adjusters with exponential decay for both to gradually reduce the learning rate. Start training, i epochs, each epoch has j batches

```
for i in range(epoch):
for j,data in enumerate(train_loader):
```

During the training process, the gradients of the discriminator are first cleared and real images are obtained from the dataset. If the batch size is smaller than the preset value, skip the current iteration. After assigning labels to the image, input the real image into the discriminator, calculate the output and loss, and perform backpropagation to update the weights. Next, identify false images and calculate the loss to update the discriminator. Subsequently, the generator is trained to clear its gradients and generate fake images, which are then fed into the discriminator to calculate the loss. The generator weights are updated through backpropagation. At the same time, calculate the feature matching loss and update the weights to ensure that the loss is negligible and the loss value is reasonable before updating.

```
gen_optim.zero_grad()
gen_dis_fake,_,dis_fake_rof=discriminator.forward(gen_fake)#Judgment on false images
gen_loss=torch.sum(loss_func(gen_dis_fake,y_real_))#Encourage fake images to be more authentic
gen_loss.backward()
gen_optim.step()
gen_loss_rof=torch.sum(loss_func_rof(dis_real_rof,dis_fake_rof))
gen_loss_rof=gen_loss_rof.detach_().requires_grad_(True)
gen_loss_rof.backward()
gen_optim.step()
```

At the end of each training cycle, save the currently trained generator and discriminator models to the specified paths. The file name contains the task name and model type. At the same time, update the learning rates of the generator and discriminator, and gradually adjust them using pre-defined learning rate adjustment strategies (such as exponential decay) to meet the needs of the training process.

```
save(generator,model_path+”/”+task+”_generator.pth”)
save(discriminator,model_path+”/”+task+”_discriminator.pth”)
gen_step_schedule.step()
dis_step_schedule.step()
```

(4) Testing section

To ensure that the same image is generated each time for comparison, a fixed set of random numbers fixed_z is set. This set of random numbers has a specified shape and follows a standard normal distribution. Subsequently, convert this set of random numbers into tensors and move them to the designated device. When generating images, turn off gradient calculation to avoid unnecessary computational overhead.

```
fixed_z=torch.Tensor(batch_size,100,1,1).normal_(0,1).to(device)
with torch.no_grad():
fixed_z=Variable(fixed_z)
```

At each epoch of the training process, set the generator to evaluation mode to generate an image. Using the previously fixed random number fixed_z as input, the generated fake image is propagated forward through the generator. Afterwards, switch the generator back to training mode. Move the generated images to the CPU and use the image_check function to save the first 25 images to the specified path. The file name should include the task name and the current epoch number for subsequent comparison and analysis.

```
generator.eval()
gen_fake,_=generator.forward(fixed_z)
generator.train()
gen_fake=gen_fake.cpu()
image_check(gen_fake.data[:25],i,save_path+’/’+task+’_’+str(i+1)+’.png’)
```

After training, conduct model testing. Firstly, load the generator model and set it to evaluation mode. To generate test images, a random number seed is set to ensure the reproducibility of the results, and a set of random numbers is generated as input. Subsequently, false images are propagated forward through the generator and moved to the CPU. Finally, use the image_check function to save the generated image to the specified path for viewing the effect of the model generated image.

3.2. Handwritten digit generation

(1) Data section

Prepare the MNIST dataset and specify the data storage path as/ data, Select the training set and apply a series of image transformations: resize the images to image_2, crop the center to image_2, convert to tensors, and standardize them. If the dataset does not exist, it will be automatically downloaded. Subsequently, create a data loader using DataLoader, set the batch size to batch_2, and enable random shuffling of data order to enhance the model's generalization ability.

```
dataset=torchvision.datasets.MNIST(
root='../data',
train=True,
transform=transforms.Compose([transforms.Resize(image_size),transforms.CenterCrop(image_size),
transforms.ToTensor(),
transforms.Normalize((0.5,),(0.5,)),]),
download=True)
##Create data loader
train_loader=torch.utils.data.DataLoader(dataset=dataset,batch_size=batch_size,
shuffle=True)
```

(2) Model section

Defined a generator called generator-mnist for handwritten digit generation. This generator inherits from nn Module, Contains one input layer and four deconvolution layers. The input layer receives a 110 dimensional vector (containing 100 dimensional noise and 10 dimensional digital labels), outputs a feature map through the first deconvolution layer (layer1), and gradually upsamples through subsequent deconvolution layers (layer2 to layer4). Finally, the fifth deconvolution layer (layer5) outputs a generated image with a specified number of channels (default is 3, but in this task it should be set to 1, representing grayscale images) and size (64x64 pixels). After each deconvolution layer, the batch normalization layer and activation function (ReLU or LeakyReLU) are followed. The last layer uses the Tanh activation function to map the output values to the [-1,1] interval to fit the range of image pixel values.

(3) Training section

In the model section, a generator called Generation-mnist was defined for handwritten digit generation. This generator inherits from nn Module, Contains one input layer and four deconvolution layers. The input layer receives a 110 dimensional vector (containing 100 dimensional noise and 10 dimensional digital labels), outputs a feature map through the first deconvolution layer (layer1), and gradually upsamples through subsequent deconvolution layers (layer2 to layer4). Finally, the fifth deconvolution layer (layer5) outputs a generated image with a specified number of channels (default is 3, but in this task it should be set to 1, representing grayscale images) and size (64x64 pixels). After each deconvolution layer, the batch normalization layer and activation function (ReLU or LeakyReLU) are followed. The last layer uses the Tanh activation function to map the output values to the [-1,1] interval to fit the range of image pixel values.

generator=Generator_mnist(ngf,output_channels).to(device)

discriminator=Discriminator_mnist(ndf,input_channels).to(device)

Prepare loss function and optimizer: use binary cross entropy loss (BCELoss) as the loss function of the discriminator, and cross entropy loss (CrossEntropy Loss) for conditional label prediction of the generator (if applicable); Configure Adam optimizers separately for the generator and discriminator, with different learning rates set, where the learning rate of the generator is 5 times that of the discriminator. Next, the training process begins by iterating through each epoch and every batch within it through nested loops: the outer loop traverses epochs a certain number of times, and the inner loop traverses every batch data in the data loader train-loader.

```
loss_func=nn.BCELoss()
loss2=nn.CrossEntropyLoss()
gen_optim=torch.optim.Adam(generator.parameters(),lr=5*learning_rate,betas=(0.5,0.999))
dis_optim=torch.optim.Adam(discriminator.parameters(),lr=learning_rate,betas=(0.5,0.999))
Start training, i epochs, each epoch has j batches
for i in range(epoch):
for j,data in enumerate(train_loader):
```

To train the discriminator, first clear its gradient and obtain real images and labels. If the batch size is too small, skip the iteration. Input real images and labels into the discriminator, calculate their loss (including image and label parts), and perform backpropagation. Generate fake images and labels, calculate the discriminator's loss on them, and update weights through backpropagation. Calculate the average output for real and fake images to evaluate performance. To identify fake images, generate a noise vector, concatenate it with labels, and pass through the generator to get fake images. Input fake images into the discriminator, reshape its output, and calculate image and label losses using binary cross-entropy and cross-entropy loss functions, respectively. Use retain_graph=True for future gradient calculations. Calculate the average output for fake images, accumulate losses to get the total loss, and update discriminator parameters using the optimizer.

When training the generator, clear its gradient and set the label to real. Generate a fake image using the generator and input it into the discriminator to obtain a discrimination result. Reshape the result into a 2D tensor and extract separate discrimination values for the image and label parts. Calculate two losses: one for the image part (encouraging the discriminator to treat the fake image as real) and one for the label part (ensuring the image matches the specified label). Use binary cross-entropy for the image loss and cross-entropy for the label loss. Perform backpropagation to update generator weights. Note that retain_graph=True is usually unnecessary due to PyTorch's dynamic graph, unless multiple .backward() calls are required. Finally, calculate the average output of the discriminator for the generated image and update generator parameters using the optimizer.

```
generator.zero_grad()
label.fill_(real_label)
gen_dis_fake=discriminator(gen_fake).view(-1)
gen_dis_fake=gen_dis_fake.view([b_size,11])
real_label_label=gen_dis_fake[:,0]
```

```
real_label_pic=gen_dis_fake[:,1:]
gen_loss=loss_func(real_label_label,label.float())
gen_loss.backward(retain_graph=True)
dis_fake_pic=loss2(real_label_pic,data[1])
dis_fake_pic.backward()
D_G_z2=gen_dis_fake.mean().item()
gen_optim.step()
Save model (for each epoch)
save(generator,model_path+"/"+task+"_generator.pth")
save(discriminator,model_path+"/"+task+"_discriminator.pth")
```

(4) Testing section

During training, at each epoch's end, set the generator to eval mode for stable image generation. Create a random noise tensor and a one-hot encoded label tensor (e.g., all labels set to 9). Concatenate them to form generator input, convert to a PyTorch tensor, and move to the correct device. Forward propagate through the generator to get the fake image. Switch the generator back to train mode for the next epoch. Move the generated image to the CPU and use a custom function to save/display it.

```
generator.eval()
noise=torch.randn(batch_size,100,device=device)
labels_onehot=np.zeros((batch_size,10))
labels_onehot[np.arange(batch_size),9]=1
noise=np.concatenate((noise.cpu().numpy(),labels_onehot),axis=1)
noise=noise.reshape([-1,110,1,1])
noise=Variable(torch.from_numpy(noise).float()).to(device)
gen_fake=generator.forward(noise)
generator.train()
gen_fake=gen_fake.cpu()
image_check_for_mnist(gen_fake.data[:25],epc,save_path+'/'+task+'_'+str(epc+1)+'.png')
```

After the training is completed, enter the model testing phase. Firstly, load and prepare the generator model, set it to evaluation mode to ensure stable performance when generating images. Then, generate random noise and a single hot encoded label of the specified number, concatenate the two and input them into the generator to generate a fake image. Finally, use the imshow function of matplotlib to visualize one of the generated images, and save all generated images using the image_check_for_mnist function (or similar function).

4. Experimental summary

This study implemented an image generation system using DCGAN. DCGAN captures image features with convolutional layers, enhancing realism and detail. It preserves spatial information by avoiding pooling layers, using stride-2 convolutions for downsampling. To address training imbalance, a feature matching loss function was introduced. Visualizing intermediate layer outputs deepened understanding of CNNs, aiding network optimization. This experiment enhanced understanding of conditional GANs and accumulated practical experience, supporting future image generation research.

About the author

Liu Lubin (2003.7 ~), male, Shenzhen, Guangdong Province, bachelor of Electronic Information Engineering, Class 2, Grade 2021, School of Electronic and Information Engineering, Shenzhen University.

References

[1] Xie Tianqi, Wu Yuanyuan, Jing Chao, Sun Weiheng Overview of GAN Model Generated Image Detection Methods [J]. Computer Engineering and Applications, 2024, 60 (22): 74-86.

[2] Zhao Hong, Li Wengai Research on Text Generation Image Model Based on Diffusion Generative Adversarial Network [J]. Chinese Journal of Electronics and Information Technology, 2023, 45 (12): 4371-4381.

[3] Liu Zerun, Yin Yufei, Xue Wenhao, Guo Rui, Cheng Lechao A review of conditional guided image generation based on diffusion models [J]. Journal of Zhejiang University (Science Edition), 2023, 50 (06): 651-667.

[4] Wang Shibin, Gao Zidiao, Liu Dong An improved DCGAN image generation method based on limited data [J]. Journal of Henan Normal University (Natural Science Edition), 2023, 51 (06): 39-46.

[5] Xu Yongshi, Ben Kerong, Wang Tianyu, Liu Sijie Improvement of DCGAN Model and Research on SAR Image Generation [J]. Computer Science, 2020, 47 (12): 93-99.