Name:Karthick Subramanian
ASU ID: 1223408524
Email: ksubra25@asu.edu

Homework 2
EEE515 Machine Vision and Pattern Recognition

2023-10-05

Problem Completion

0.1 Problem 1.

0.1.1 Subdivision 1. Completed successfully.

0.1.2 Subdivision 2. Completed successfully.

0.1.3 Subdivision 3. Completed successfully.

0.1.4 Subdivision 4. Completed successfully.

0.2 Problem 2.

0.2.1 Subdivision 1. Completed successfully.

0.2.2 Subdivision 2. Completed successfully.

0.2.3 Subdivision 3. Completed successfully.

0.2.4 Subdivision 4. Completed successfully.

0.2.5 Subdivision 5. Completed successfully.

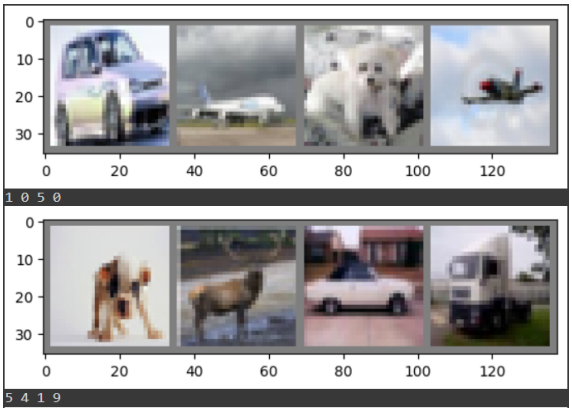
Problem 1

Subdivision 1.

For this problem, I have used the LeNet5 architecture for the CIFAR10 dataset:

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1            [-1, 6, 28, 28]             456
              ReLU-2            [-1, 6, 28, 28]               0
         MaxPool2d-3            [-1, 6, 14, 14]               0
            Conv2d-4           [-1, 16, 10, 10]           2,416
              ReLU-5           [-1, 16, 10, 10]               0
         MaxPool2d-6             [-1, 16, 5, 5]               0
            Linear-7                  [-1, 120]          48,120
              ReLU-8                  [-1, 120]               0
            Linear-9                   [-1, 84]          10,164
             ReLU-10                   [-1, 84]               0
           Linear-11                   [-1, 10]             850
================================================================
Total params: 62,006
Trainable params: 62,006
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.01
Forward/backward pass size (MB): 0.11
Params size (MB): 0.24
Estimated Total Size (MB): 0.36
----------------------------------------------------------------
```

(a) LeNet5 Architecture



(b) CIFAR10 Data set examples with corresponding classes

The dataset was imported using torchvision.datasets
A simple transform is done on the image to convert them to torch tensors and normaize the images along all 3 channels.
The hyper parameters used are as follows:
Learning rate = 0.001
batch size = 64
momentum = 0.9
Loss function: Cross entropy loss
Optimizer: Stochastic Gradient Descent optimizer

```python
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])

trainset = torchvision.datasets.CIFAR10(root='./data', train=True, download=True, transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=64, shuffle=True, num_workers=2)

testset = torchvision.datasets.CIFAR10(root='./data', train=False, download=True, transform=transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=64, shuffle=False, num_workers=2)


net = LeNet5()
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
net.to(device)


criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)
```

(a) Hyper Parameters and loading dataset

The training loop used is as follows:
No. of Epochs: 5
Device: Running on GPU using CUDA

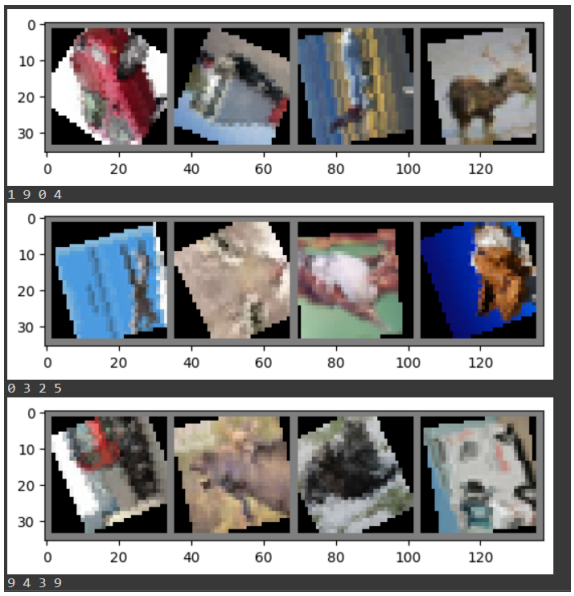
```python
num_epochs = 5
for epoch in range(num_epochs):
    net.train()
    running_loss = 0.0
    for i, data in enumerate(tqdm(trainloader, position=0, leave=True), 0):
        inputs, labels = data
        inputs, labels = inputs.to(device), labels.to(device)

        optimizer.zero_grad()
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        running_loss += loss.item()
        if i % 200 == 199:
            print(f"[{epoch + 1}, {i + 1}] Loss: {running_loss / 200:.3f}")
            running_loss = 0.0
```

(a) Training loop

The accuracy obtained for this model for 5 epochs is: 62.34%

Subdivision 2. Dataset Augmentation

Some examples of the augmented dataset and the corresponding augmentations(transforms) done:



(a) Augmented data

```
transform = transforms.Compose([
    transforms.RandomCrop(32,padding=4),
    transforms.RandomHorizontalFlip(),
    transforms.RandomRotation(degrees=(0, 180)),
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])
```

(b) Transforms performed

Augmentations:
1. Cropping a random part of the image.
2. Performing a horizontal flip on the image.
3. Performing a rotation by an angle between 0 and 30.
4. Adding noise separately.

The accuracy of the model after augmenting the images is: 64.18%

Subdivision 3. Implementing ResNet Architecture

The residual block and ResNet18 architecture used are as follows:

```
class BasicBlock(nn.Module):
    def __init__(self, in_channels, out_channels, stride=1):
        super(BasicBlock, self).__init__()
        self.conv1 = nn.Conv2d(in_channels, out_channels, kernel_size=3, stride=stride, padding=1, bias=False)
        self.bn1 = nn.BatchNorm2d(out_channels)
        self.relu = nn.ReLU(inplace=True)
        self.conv2 = nn.Conv2d(out_channels, out_channels, kernel_size=3, stride=1, padding=1, bias=False)
        self.bn2 = nn.BatchNorm2d(out_channels)
        self.downsample = nn.Sequential()
        if stride != 1 or in_channels != out_channels:
            self.downsample = nn.Sequential(
                nn.Conv2d(in_channels, out_channels, kernel_size=1, stride=stride, bias=False),
                nn.BatchNorm2d(out_channels)
            )

    def forward(self, x):
        residual = self.downsample(x)
        out = self.conv1(x)
        out = self.bn1(out)
        out = self.relu(out)
        out = self.conv2(out)
        out = self.bn2(out)
        out += residual
        out = self.relu(out)
        return out
```

(a) Basic residual block

```
class ResNet(nn.Module):
    def __init__(self, block, num_blocks, num_classes=10):
        super(ResNet, self).__init__()
        self.in_channels = 64
        self.conv1 = nn.Conv2d(3, 64, kernel_size=3, stride=1, padding=1, bias=False)
        self.bn1 = nn.BatchNorm2d(64)
        self.relu = nn.ReLU(inplace=True)
        self.layer1 = self._make_layer(block, 64, num_blocks[0], stride=1)
        self.layer2 = self._make_layer(block, 128, num_blocks[1], stride=2)
        self.layer3 = self._make_layer(block, 256, num_blocks[2], stride=2)
        self.layer4 = self._make_layer(block, 512, num_blocks[3], stride=2)
        self.avg_pool = nn.AdaptiveAvgPool2d((1, 1))
        self.fc = nn.Linear(512, num_classes)

    def _make_layer(self, block, out_channels, num_blocks, stride):
        layers = []
        layers.append(block(self.in_channels, out_channels, stride))
        self.in_channels = out_channels
        for _ in range(1, num_blocks):
            layers.append(block(out_channels, out_channels, stride=1))
        return nn.Sequential(*layers)
```

(b) ResNet18 architecture

The hyperparameters used are as follows:
Learning rate = 0.001
Weight_decay = 0.0001
Batch size = 64
Loss function: crossEntropyLoss Optimizer: SGD optimizer
The training loop is shown below: No. of epochs: 5 Device: Running on GPU using CUDA

```python
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])

trainset = torchvision.datasets.CIFAR10(root='./data', train=True, download=True, transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=64, shuffle=True, num_workers=2)

testset = torchvision.datasets.CIFAR10(root='./data', train=False, download=True, transform=transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=64, shuffle=False, num_workers=2)

net = ResNet(BasicBlock, [2, 2, 2, 2])
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
net.to(device)

criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(net.parameters(), lr=0.001, weight_decay=0.0001)
```

(a) ResNet Hyperparameters

```python
num_epochs = 5
for epoch in range(num_epochs):
    net.train()
    running_loss = 0.0
    for i, data in enumerate(tqdm(trainloader, position=0, leave=True), 0):
        inputs, labels = data
        inputs, labels = inputs.to(device), labels.to(device)

        optimizer.zero_grad()
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        running_loss += loss.item()
        if i % 200 == 199:
            print(f"[{epoch + 1}, {i + 1}] Loss: {running_loss / 200:.3f}")
            running_loss = 0.0
```

(b) Training loop for ResNet18

The basic block and the ResNet34 architecture are shown below:

```python
class BasicBlock(nn.Module):
    expansion = 1

    def __init__(self, in_planes, planes, stride=1):
        super(BasicBlock, self).__init__()
        self.conv1 = nn.Conv2d(in_planes, planes, kernel_size=3, stride=stride, padding=1, bias=False)
        self.bn1 = nn.BatchNorm2d(planes)
        self.conv2 = nn.Conv2d(planes, planes, kernel_size=3, stride=1, padding=1, bias=False)
        self.bn2 = nn.BatchNorm2d(planes)

        self.shortcut = nn.Sequential()
        if stride != 1 or in_planes != self.expansion * planes:
            self.shortcut = nn.Sequential(
                nn.Conv2d(in_planes, self.expansion * planes, kernel_size=1, stride=stride, bias=False),
                nn.BatchNorm2d(self.expansion * planes)
            )

    def forward(self, x):
        out = nn.ReLU()(self.bn1(self.conv1(x)))
        out = self.bn2(self.conv2(out))
        out += self.shortcut(x)
        out = nn.ReLU()(out)
        return out
```

(a) Basic residual block

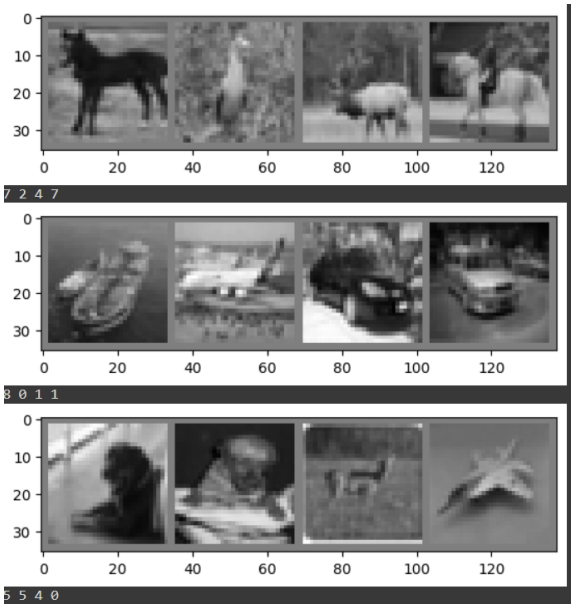
```python
class ResNet34(nn.Module):
    def __init__(self, num_classes=10):
        super(ResNet34, self).__init__()
        self.in_planes = 64
        self.conv1 = nn.Conv2d(3, 64, kernel_size=3, stride=1, padding=1, bias=False)
        self.bn1 = nn.BatchNorm2d(64)
        self.layer1 = self._make_layer(BasicBlock, 64, 3, stride=1)
        self.layer2 = self._make_layer(BasicBlock, 128, 4, stride=2)
        self.layer3 = self._make_layer(BasicBlock, 256, 6, stride=2)
        self.layer4 = self._make_layer(BasicBlock, 512, 3, stride=2)
        self.avgpool = nn.AdaptiveAvgPool2d(1)
        self.fc = nn.Linear(512, num_classes)

    def _make_layer(self, block, planes, num_blocks, stride):
        strides = [stride] + [1] * (num_blocks - 1)
        layers = []
        for stride in strides:
            layers.append(block(self.in_planes, planes, stride))
            self.in_planes = planes * block.expansion
        return nn.Sequential(*layers)

    def forward(self, x):
        out = nn.ReLU()(self.bn1(self.conv1(x)))
        out = self.layer1(out)
        out = self.layer2(out)
        out = self.layer3(out)
        out = self.layer4(out)
        out = self.avgpool(out)
        out = out.view(out.size(0), -1)
        out = self.fc(out)
        return out
```

(b) ResNet34 architecture

The hyperparameters used are as follows:
Learning rate = 0.001
Weight_decay = 0.0001
Batch size = 64
Loss function: crossEntropyLoss Momentum = 0.9 Optimizer: SGD optimizer
The training loop is shown below: No. of epochs: 5 Device: Running on GPU using CUDA
The accuracy obtained by the ResNet18 model is: 76.28%
The accuracy obtained by the ResNet34 model is: 74.51%
The accuracy obtained by the ResNet50 model is: 72.66%
The accuracy obtained by the ResNet101 model is: 70.53%

Subdivision 4. Grayscale conversion

Here, I represent some images of the grayscale training data:



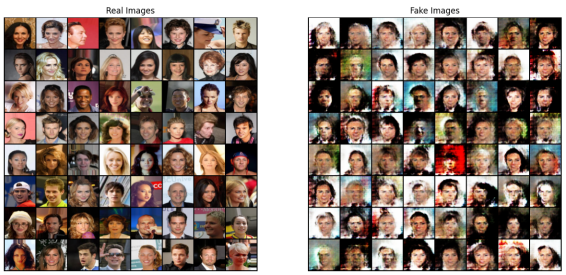
(a) Grayscale CIFAR10 images

The accuracy of the model when half the training images are in grayscale and tested in color test set is: 32.64%
The accuracy of the model when all the training images are in grayscale and tested in color test set is: 61.62%
The accuracy of the model when half the training images are in grayscale and tested in gray test set: 34.22%
The accuracy of the model when all the training images are in grayscale and tested in gray test set is: 76.99%

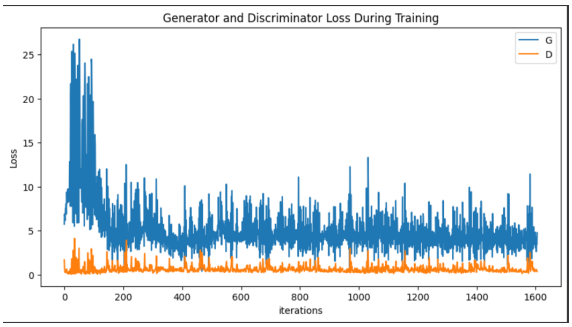
Problem 2.

Subdivision 1

Using DCGAN given, the results are as seen below:



(a) Real images and Fake images



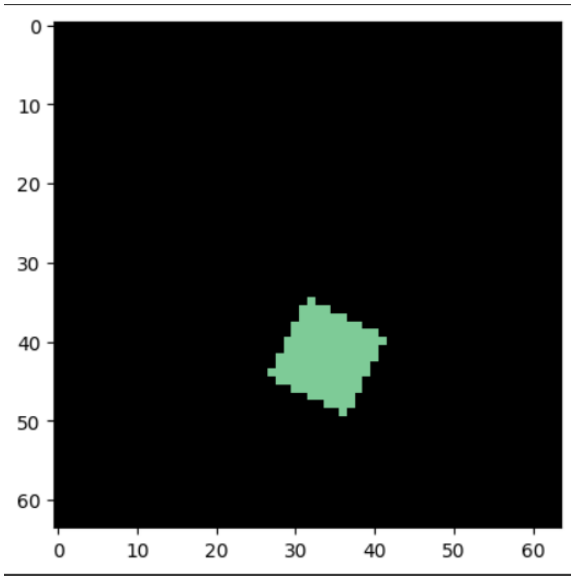
(b) Discriminator and Generator loss

(a) Fake faces generated

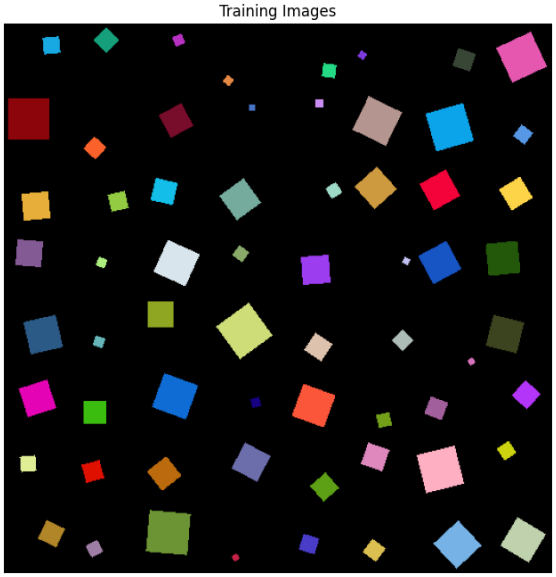
The number of epochs = 3. The results can be made better by running for more number of epochs. But this would take up more time. The results are comparatively better than using a custom data set because it is very divers and not a lot of data augmentation had to be done.

Subdivision 2 Generate Colored Squares Dataset

The squares are generated as seen below:



(a) Example of generated square



(b) Training examples

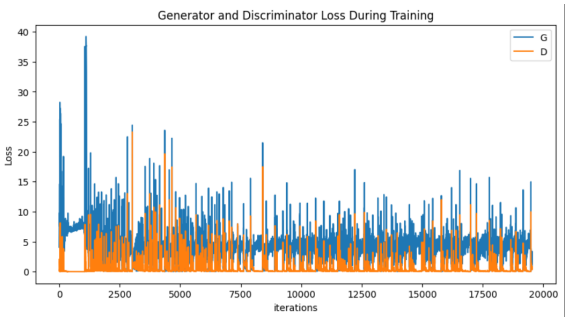
Initially, I create the black canvas, then randomly assign properties for the squares to be included (color, size, rotation angle - between 0 and 180 degrees).
Then I randomly determine the position of the square within the image, ensuring that the square does not exceed the image boundaries.

Subdivision 3 Training on squares

I have used a custom dataloader to load the generated squares data using torchvision datasets. I get the images as torch tensors, which are used for training:

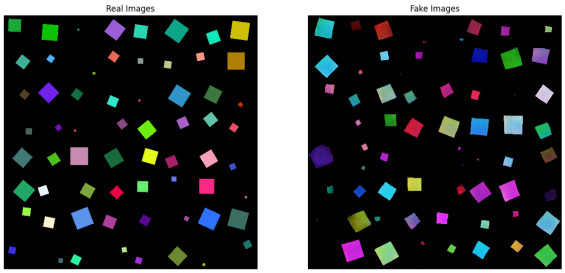
```python
transform = transforms.Compose([
    transforms.Resize((image_size, image_size)),
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
])


class CustomDataset(Dataset):
    def __init__(self, root_dir, transform=None):
        self.root_dir = root_dir
        self.transform = transform
        self.file_list = os.listdir(root_dir)

    def __len__(self):
        return len(self.file_list)

    def __getitem__(self, idx):
        img_name = os.path.join(self.root_dir, self.file_list[idx])
        image = Image.open(img_name)
        if self.transform:
            image = self.transform(image)
        return image

dataset = CustomDataset(root_dir="extracted_images", transform=transform)
dataloader = DataLoader(dataset, batch_size=batch_size, shuffle=True)
```

(a) Dataset loader

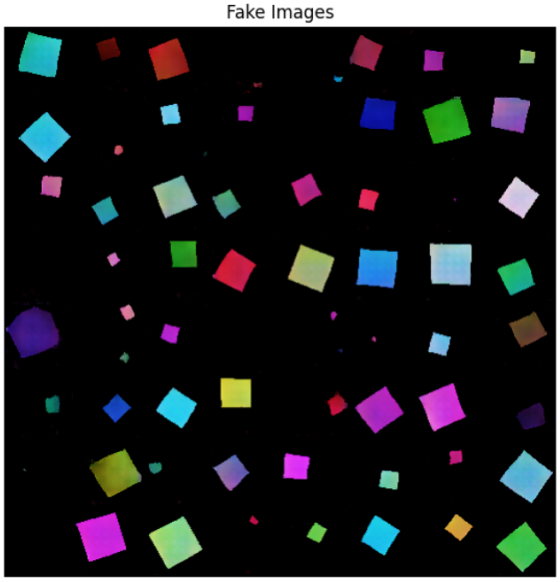


(b) Loss of Generator and Discriminator

On the left, the real images used for training can be seen. On right, the images generated by the generator can be seen:



(a) Comparison between real and generated(fake) images

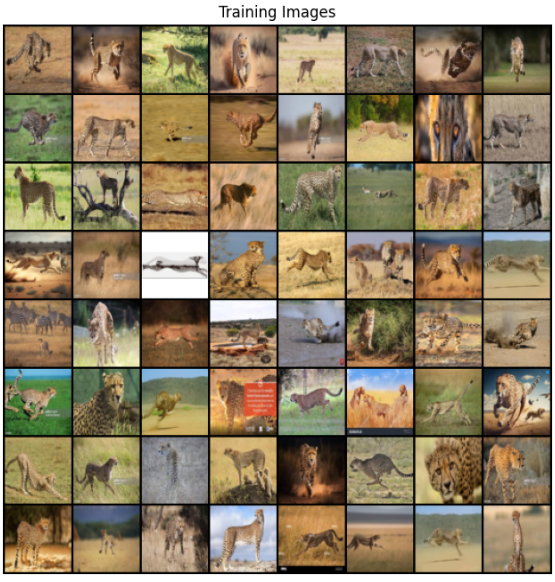


(a) Fake images generated running 5 epochs for 500,00 images

Reason for good images being generated is that I am using a very huge data set. The size of the data set being used is 500,000 images. The accuracy can be further increased by increasing the number of epochs.

Subdivision 4.

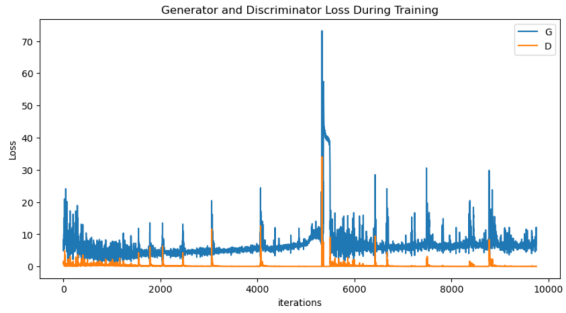
Training data used with pictures of Cheetah:



(a) Sample Cheetah images used for training

Subdivision 5.

The above images are examples of the set used to train the generator network.



(a) Loss of Generator and Discriminator

(b) Real and fake images

The images are better from when I had submitted previously. I was able to run the entire program for 150 epochs.
Some of the pictures generated seem to have a figure similar to that of a cheetah running such as the following:

(a) Generated image 1

(b) Generated image 2

Even though there is a lot of difficulty generating the animal, it is clear that the generator is easily able to create the background of the savannah.
The generator could perform better with more epochs. But that becomes more time consuming. Alternatively, the data set could be increased. Either of these techniques could result in better images being generated.