Generative Adversarial Networks

GENERATE IMAGES BASED ON THE SMALLNORB DATASET

Dirk Stulgies, Prosper Kwabena Adjei

INTRODUCTION

- This project was created as part of the opencampus.sh course Generative Adversarial Networks.
- The goal was to train a neural network with the smallNorb dataset and afterwards generate images similar to those in the original dataset.

DATASET

- **Images of 50 toys.**
  - **Imaged by two cameras**
  - **Five categories** (four-legged animals, human figures, airplanes, trucks, and cars)
  - **Ten instances** (lion, hippo, van etc.)
  - **Six lighting conditions**
  - **Nine elevations** (30 to 70 degrees every 5 degrees)
  - **18 azimuths** (0 to 340 every 20 degrees)

Ill. 1: Images from the smallNorb dataset

NETWORK

- Wasserstein GAN
- Gradient penalty
- Adam optimize
- BCE with logins loss function
- Initialise weights for Conv2D and Conv2DTranspose based on normal distribution.

```
mean_iteration_critic_loss = 0
for _ in range(RUN_CRITIC_REPEAT):
    # Update the critic.
    crit_opt.zero_grad()
    fake_noise = get_noise(len(real), RUN_Z_DIM, device=RUN_DEVICE)
    fake = gen(fake_noise)
    crit_fake_pred = crit(fake.detach())
    crit_real_pred = crit(real)

    # Calculate the gradient penalty.
    epsilon = torch.rand(len(real), 1, 1, 1, device=RUN_DEVICE, requires_grad=True)
    gradient = get_gradient(crit, real, fake.detach(), epsilon)
    gp = gradient_penalty(gradient)
    crit_loss = get_crit_loss(crit_fake_pred, crit_real_pred, gp, RUN_CRITIC_LAMBDA)

    # Keep track of the average critic loss in this batch.
    mean_iteration_critic_loss += crit_loss.item() / RUN_CRITIC_REPEAT

    # Update the gradients.
    crit_loss.backward(retain_graph=True)

    # Update the optimizer.
    crit_opt.step()

# Add the critics loss value to the list, after each step.
crit_loss_step += [mean_iteration_critic_loss]
```

Ill. 2: Coded snippet, critics repeat, adding gradient penalty.

GENERATOR

- Five blocks containing:
  - ConvTranspose2d layer
  - InstanceNorm2d layer
  - LeakyReLU activation layer
  - Dropout layer
- The final block containing:
  - ConvTranspose2d layer
  - InstanceNorm2d layer
  - Tanh activation layer

```
self.layers = torch.nn.Sequential(
    self.convTranspose2d_block(z_dim, hidden_dim, 3, 1, dropout_value, leak_value),
    self.convTranspose2d_block(hidden_dim, hidden_dim, 3, 2, dropout_value, leak_value),
    self.convTranspose2d_block(hidden_dim, hidden_dim, 5, 1, dropout_value, leak_value),
    self.convTranspose2d_block(hidden_dim, hidden_dim, 5, 2, dropout_value, leak_value),
    self.convTranspose2d_block(hidden_dim, hidden_dim, 7, 1, dropout_value, leak_value),
    self.convTranspose2d_block(hidden_dim, 1, 7, 2, dropout_value, leak_value, final_layer=True)
)
```

```
if not final_layer:
    # Define the block for not final layer.
    return torch.nn.Sequential(
        torch.nn.ConvTranspose2d(input_channels, output_channels, kernel_size, stride),
        torch.nn.InstanceNorm2d(output_channels),
        torch.nn.LeakyReLU(leak_value, inplace=True),
        torch.nn.Dropout(dropout_value)
    )
else:
    # Define the block for the final layer.
    return torch.nn.Sequential(
        torch.nn.ConvTranspose2d(input_channels, output_channels, kernel_size, stride),
        torch.nn.InstanceNorm2d(output_channels),
        torch.nn.Tanh()
    )
```

Ill. 3: Code snippet, generator structure.

CRITIC

- Five blocks containing:
  - Conv2d layer
  - InstanceNorm2d layer
  - LeakyReLU activation layer
  - Dropout layer
- The final block containing:
  - Conv2d layer

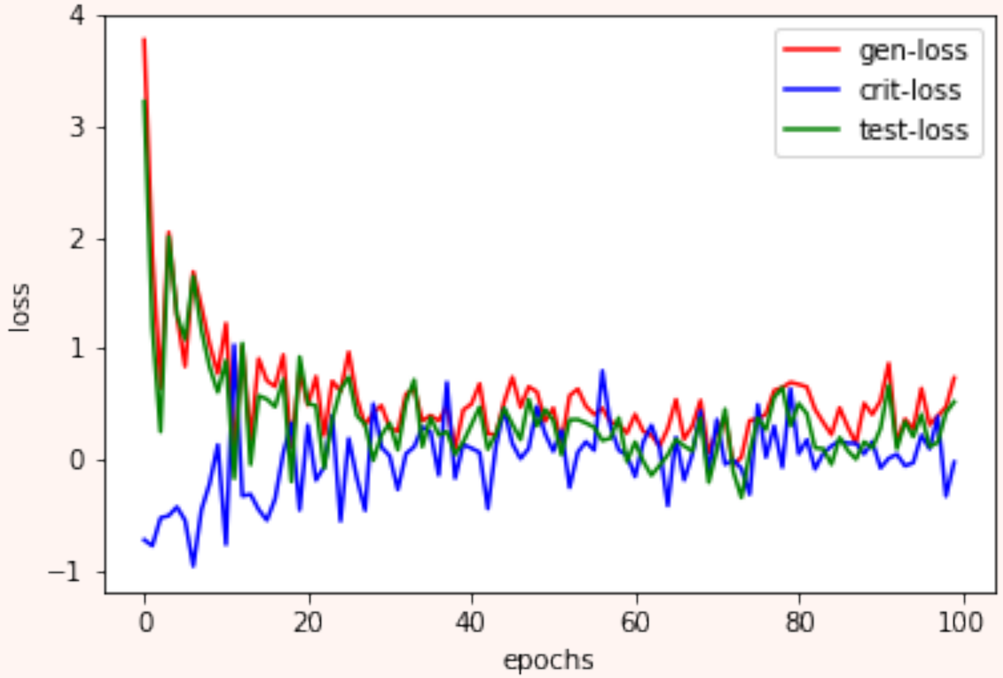
```
self.layers = torch.nn.Sequential(
    self.conv2d_block(img_channel, hidden_dim, 7, 2, dropout_value, leak_value),
    self.conv2d_block(hidden_dim, hidden_dim, 7, 1, dropout_value, leak_value),
    self.conv2d_block(hidden_dim, hidden_dim, 5, 2, dropout_value, leak_value),
    self.conv2d_block(hidden_dim, hidden_dim, 5, 1, dropout_value, leak_value),
    self.conv2d_block(hidden_dim, hidden_dim, 3, 2, dropout_value, leak_value),
    self.conv2d_block(hidden_dim, 1, 3, 1, dropout_value, leak_value, final_layer=True)
)
```

```
if not final_layer:
    # Define the block for not final layer.
    return torch.nn.Sequential(
        torch.nn.Conv2d(input_channels, output_channels, kernel_size, stride),
        torch.nn.InstanceNorm2d(output_channels),
        torch.nn.LeakyReLU(leak_value, inplace=True),
        torch.nn.Dropout(dropout_value)
    )
else:
    # Define the block for the final layer.
    return torch.nn.Sequential(
        torch.nn.Conv2d(input_channels, output_channels, kernel_size, stride)
    )
```

Ill. 4: Code snippet, critics structure.

RESULTS UNCONDITIONAL GAN

Ill. 5: Results after each epoch.



Ill. 6: Losses

REFERENCES

- Dataset
  - smallNorb dataset: https://cs.nyu.edu/~ylclab/data/norb-v1.0-small/
- Cousera courses
  - Build basic GAN’s:
  - Buils better GAN’s
  - Apply GAN's
- Code examples from other authors
  - Reading the smallNorb files: https://www.kaggle.com/code/leshabirukov/small-norb-load
  - Generating images based on the smallNorb dataset: https://medium.com/analytics-vidhya/applying-generative-adversarial-network-to-generate-novel-3d-images-ba70e1176dac