Assignment 04: Generative Models
CSI5340 – Introduction to deep learning and reinforcement learning
Cristopher McIntyre Garcia
300025114

Designs
In this assignment three generative models were designed with several configurations. The first model is a Variational AutoEncoder (VAE), the second is a Generative Adversarial Network (GAN), and the third is a Wasserstein Generative Adversarial Network (WGAN). Each model was implemented in such a way as to allow the user to use different network depths and latent space dimensions. In this particular implementation, the modules can be as deeps as five layers and as shallow as one layer. Because these networks are all two-module based, the depth of each network is double their module depth. Each network module is implemented as a deep Convolutional Neural Network (CNN). To keep training code consistent, the loss of each network is calculated by itself and returned.

The VAE consists of an encoder and a decoder network. The encoder network is a simple architecture of convolution layers followed by linear layers. Every convolution layer is followed by a batch normalization layer and an activation function. The final two linear layers are branched from a former linear layer to produce the mean and standard deviation of the Gaussian distribution particular to the class of the input image, in the latent space. The decoder network can be thought of as the reverse of the encoder network. Firstly, a latent representation is fed into a linear layer which maps it to a dimension compatible with the first hidden layer. The result is passed through a set of transpose convolution layers to upscale the representation. The transpose convolution layers are all followed by a batch normalization layer and an activation function. The final layer’s activation function is tanh. With the encoder and decoder, the VAE is able to compute the reconstruction and KL divergence loss of an image and a provided reconstruction. The steps can be seen in the following modules.

```
def Encoder(x, depth, h_dim, z_dim):
        for _ in range(depth):
                x = conv(x)
                x = batch_norm(x)
                x = relu(x)
        z = linear(x, h_dim)
        μ = linear(z, z_dim)
        σ = linear(z, z_dim)
        return  μ, σ
```

Module 1 (a) Encoder

```
def Decoder(z, depth, h_dim):
        x’ = linear(z, h_dim)
        for _ in range(depth-1):
                x’ = transpose(x’)
                x’ = batch_norm(x’)
                x’ = relu(x’)
        x’ = conv(x’)
        x’ = batch_norm(x’)
        x’ = tanh(x’)
        return  x’
```

Module 1 (b) Decoder

```
def VAE(x, encoder, decoder):
        μ, σ = encoder(x)
        z = N(σ) * σ +  μ
        x’ = decoder(z)
        rec = mse(x, x’)
        KL = -0.5 * mean(sum(1 + log(σ²) - μ² - e^log(σ²)))
        return  rec + KL * KL_factor
```

Module 1 (c) VAE

It is up to the encoder to produce proper distributions in the latent space, and it is up to the decoder to generate an image, given a latent representation.

The GAN consists of a generator and a discriminator network. The generator network is an architecture of transpose convolution layers, which are all followed by a batch normalization layer and an activation function. The finally transpose convolution layer omits the batch normalization layer, and uses tanh as the activation function. The purpose of the generator is to realize an image from a latent representation. The discriminator is an architecture of convolution layers, which are mostly all followed by a batch normalization layer and an activation function. The first and last convolution layers are not followed by a batch normalization layer. After the convolution layers, a linear function maps the representation to 1 dimension, and is passed through a sigmoid activation function. The purpose of the discriminator is to identify whether a given image comes from the dataset distribution or not. With the generator and discriminator, the GAN is able to compute the generative and discriminative loss of a real and fake image. The steps can be seen in the sudo code found in the following modules.

```
def Generator(z, depth):
    for _ in range(depth):
        z = conv_transpose(z)
        z = batch_norm(z)
        z = relu(z)
    x’ = conv_transpose(z)
    x’ = tanh(x’)
    return x’
```

Module 2 (a) Generator

```
def Discriminator(x, depth):
    x = conv(x)
    x = leakyRelu(x)
    for _ in range(depth-1):
        x = conv(x)
        x = batch_norm(x)
        x = leakyRelu(x)
    x = conv(x)
    x = leakyRelu(x)
    i = linear(x, 1)
    i = sigmoid(i)
    return  i
```

Module 2 (b) Discriminator

```
def GAN(x, generator, discriminator):
    z = N(0, 1)
    x’ = generator(z)
    real = discriminator(x)
    fake = discriminator(x’)
    gen_loss = mse(fake, 1)
    disc_loss = (mse(fake, 0) + mse(real, 1)) / 2
    return gen_loss, disc_loss
```

Module 2 (c) GAN

It is up to the generator to images from a latent representation that can fool the discriminator into classifying it as a real image. It is up to the discriminator to correctly classify whether an image was generated or came from the dataset distribution. The loss computed will be used to minimize the JSD between the distribution of generated images and that of the dataset.

The WGAN is very similar to the GAN, and mostly differs in its means of calculating the loss. The generator is identical to that found in the GAN. The critic is identical to the discriminator except for the final activation function. The discriminator uses the sigmoid activation function, while the critic omits the last activation function. The loss

```
def Generator(z, depth):
        for _ in range(depth):
                z = conv_transpose(z)
                z = batch_norm(z)
                z = relu(z)
        x’ = conv_transpose(z)
        x’ = tanh(x’)
        return x’
```

Module 3 (a) Generator

```
def Critic(x, depth):
        x = conv(x)
        x = leakyRelu(x)
        for _ in range(depth-1):
                x = conv(x)
                x = batch_norm(x)
                x = leakyRelu(x)
        x = conv(x)
        x = leakyRelu(x)
        i = linear(x, 1)
        return  i
```

Module 3 (b) Critic

```
def GAN(x, generator, critic):
        z = N(0, 1)
        x’ = generator(z)
        real = critic(x)
        fake = critic(x’)
        gen_loss = - mean(fake)
        critic_loss = mean(fake) - mean(real)
        return gen_loss, disc_loss
```

Module 3 (c) WGAN

The generator plays the same role as it did in the GAN network. The critic plays a similar role as the discriminator from the GAN network. The difference here is that the Critic is not determining whether an image is fake or real, instead it evaluates how likely it is to be real. The loss is used to minimize the Wasserstein distance between the distribution of generated images and that of the dataset.

Training

Because the loss is computed in the network modules, training is simple and only evolves running the model and performing backward propagation. The VAE is the simplest because the entire model is trained with a single optimizer.

```
def train_VAE(model, x):
    for _ in range(epochs):
        for _ in range(iterations)
            loss = model(x)
            loss.backward()
            optimizer.step()
    return model
```

Algorithm 1 Train VAE

The GAN and WGAN are trained similarly to each other. Because the modules of these networks are essentially competing with each other, they need to be trained separately.

```
def train_GAN(model, x):
    for _ in range(epochs):
        for _ in range(iterations)
            loss = model(x)
            loss.backward()
            optimizer.step()
    return model
```

Algorithm 2 Train GAN

Generation

Results

First we look at the results obtained when using the VAE model. Depending on the KL factor chosen, the generated images will look more or less similar to the original image. The hyper parameters used to generate the following images are as follows.

z_dim = 100

lr = 3e-4

depth = 5

First we look at the results on the MNIST dataset.

Discussion

End of report.