Carlos Gomes

Visão por Computador e Processamento de Imagem
Universidade de Minho
4704-553 Braga
pg47083@alunos.uminho.pt

1 Introdução

O presente trabalho prático, desenvolvido em conformidade com a unidade curricular de Visão por Computador e Processamento de Imagem, visa apresentar de forma elucidativa a aprendizagem derivada do contexto prático e teórico lecionado, principalmente a exploração de **GANs** e **Autoencoders**, assim como a exploração de **loss** e **manipulação** ou **combinações** de datasets.

Desta forma, foi-nos dado a liberdade de escolhermos um problema, tendo em conta que existem imensos datasets para um determinado problema. Após uma breve pesquisa, deparei-me com um problema bastante interessante: **criar caras de personagens de anime**.

Como para desenhar estas personagens em condições é necessário imenso esforço, é necessário que haja a possibilidade de gerar personagens, felizmente, encontramos-nos numa era em que a tecnologia está bastante avançada, somos capazes de completar várias tarefas complexas que antigamente seriam impossíveis.

As ferramentas utilizadas para este trabalho prático serão as mesmas ferramentas que foram utilizadas durante as aulas, isto é, como **Tensorflow**, entre outros.

2 Problema

2.1 Descrição

O problema para este trabalho prático será então: conseguir gerar caras de personagens de anime através de utilização de GANs e/out Autoencoders.

2.2 Dataset escolhido

Havendo imensos datasets disponíveis, foi escolhido o seguinte dataset: *https://github.com/bchao1/Anime-Face-Dataset*, é um dataset que consiste em mais de **63000 imagens**,retiradas de *www.getchu.com*, os tamanhos destas imagens poderão variar desde de **90x90** até **120x120**.

No entanto, há um pequeno problema relativamente a este dataset, poderá conter algumas imagens que estão invalidas, com 0 bytes, pelo que será necessário remover essas imagens através do seguinte script:

```
if not REMOVED:
   for root, dirs, files in
       os.walk("dataset"):
     for file in files:
       path = os.path.join(root, file)
       if os.stat(path).st_size == 0:
         print("Removing file:", path)
         os.remove(path)
```

Uma vez que estas imagens são removidas, é aconselhável ver algumas imagens do dataset com que iremos trabalhar.

Figure 1: Previsão de dataset

Portanto, ao longo deste trabalho, iremos ter como um dos objetivos conseguir criar imagens semelhantes às caras presentes no dataset.

3 Generative Adversarial Networks (GAN)

GANs é uma classe de machine learning que surgiu em 2014. Consiste em dois modelos que serão treinados ao mesmo tempo e irão constetar-se um ao outro, o generator aprende a criar imagens que parecem reais, enquanto o discriminator aprende a distinguir quais são as imagens reais e quais são as imagens falsas, em outras palavras,

o ganho de um modelo é o loss de outro modelo, um exemplo perfeito para esta situação é a conversão de zebra para cavalo e vice-versa. No inicio, quando o treino começa, o **Generator** irá criar imagens falsas que serão óbvias para o **Discriminator** rapidamente aprender que são falsas, mas à medida que o treino vai prosseguindo, o **Generator** irá apróximar-se aos poucos de uma imagem capaz de "enganar" o **Discriminator**.

3.1 Generator

Como foi referido, o **Generator** será responsável então por criar imagens de forma a conseguir "enganar" o **Discriminator**. Um modelo de **Generator** deverá começar com uma camada **Dense** que recebe como input um **random noise** e irá passar por várias camadas **Conv2DTranspose** até chegar finalmente ao tamanho **(64,64,3)**.

```
generator = tf.keras.Sequential()

generator.add(InputLayer(input_shape=
(NOISE_DIM,)))
generator.add(Dense(4*4*256))
generator.add(BatchNormalization())
generator.add(LeakyReLU())

generator.add(Reshape((4, 4, 256)))

generator.add(Conv2DTranspose(128,
    kernel_size=KERNEL_SIZE,
    strides=STRIDES,padding='same'))
generator.add(BatchNormalization())
generator.add(LeakyReLU())

generator.add(Conv2DTranspose(64,
    kernel_size=KERNEL_SIZE,
    strides=STRIDES, padding='same'))
generator.add(BatchNormalization())
generator.add(LeakyReLU())

generator.add(Conv2DTranspose(32,
    kernel_size=KERNEL_SIZE,
    strides=STRIDES, padding='same'))
generator.add(BatchNormalization())
generator.add(LeakyReLU())


generator.add(Conv2DTranspose(3,
    kernel_size=KERNEL_SIZE,
    strides=STRIDES, padding='same',
    activation='tanh'))
```

3.2 Discriminator

Enquanto o **Generator** tem o papel de criar imagens, o **Discriminator** terá como o papel aprender a analisar as imagens e dizer se são reais ou não.

```
# Discriminator Model
discriminator = tf.keras.Sequential()
# Get as input 28x28x1
discriminator.add(Conv2D(32,
    kernel_size=KERNEL_SIZE,
    strides=STRIDES, padding='same',
    input_shape=[IMAGE_SIZE, IMAGE_SIZE,
    3]))
discriminator.add(LeakyReLU())
discriminator.add(Dropout(0.3))

discriminator.add(Conv2D(64, kernel_size=
    KERNEL_SIZE, strides=STRIDES,
    padding='same'))
discriminator.add(BatchNormalization())
discriminator.add(LeakyReLU())
discriminator.add(Dropout(0.3))

discriminator.add(Conv2D(128,
    kernel_size= KERNEL_SIZE,
    strides=STRIDES, padding='same'))
discriminator.add(BatchNormalization())
discriminator.add(LeakyReLU())
discriminator.add(Dropout(0.3))

discriminator.add(Conv2D(256,
    kernel_size= KERNEL_SIZE,
    strides=STRIDES, padding='same'))
discriminator.add(BatchNormalization())
discriminator.add(LeakyReLU())
discriminator.add(Dropout(0.3))

# Returns 1x1 as output, if positive then
    it's a real image, otherwise it's a
    generated image
discriminator.add(Flatten())
discriminator.add(Dense(1))
```

Agora, como ambos os modelos estão criados, se pedirmos ao **Generator** que crie uma cara de personagem, iremos só ter como resultado noise pois os modelos ainda não estão treinados.

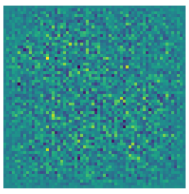

Figure 2: Primeiro resultado

3.3 Discriminator and Generator Loss

Um **GAN** irá ter duas funções para **loss**: **Discriminator Loss** e **Generator Loss**.

O **Discriminator Loss** representa o quão bem é capaz de distinguir as imagens reais de imagens falsas, para isso irá comparar as predictions em imagens reais com um array de 1s e predictions em imagens falsas com um array de 0s.

```
def discriminator_loss(real_face,
    fake_face):
   real_loss =
       cross_entropy(tf.ones_like(real_face),
       real_face)
   fake_loss =
       cross_entropy(tf.zeros_like(fake_face),
       fake_face)

   total_loss = real_loss + fake_loss

   return total_loss
```

```
Epoch = 0, Generator Loss = 0, Discriminator Loss = 0
Epoch = 1, Generator Loss = 0.6882780194282532, Discriminator Loss = 1.6968992948532104
Epoch = 2, Generator Loss = 0.7314543724060059, Discriminator Loss = 1.6520030498504639
Epoch = 3, Generator Loss = 0.7209100723266602, Discriminator Loss = 1.4887299537658691
Epoch = 4, Generator Loss = 0.7846875190734863, Discriminator Loss = 1.5149145126342773
Epoch = 5, Generator Loss = 0.834614098072052, Discriminator Loss = 1.335925579071045
Epoch = 6, Generator Loss = 0.754235565662384, Discriminator Loss = 1.437612533569336
Epoch = 7, Generator Loss = 0.7344276309013367, Discriminator Loss = 1.545384407043457
Epoch = 8, Generator Loss = 0.7184399366378784, Discriminator Loss = 1.4950412511825562
Epoch = 9, Generator Loss = 0.7430804371833801, Discriminator Loss = 1.4044320583343506
```

Figure 3: Loss obtido com 10 epochs

Relativamente ao **Generator Loss** representa o quão bem consegue enganar o **Discriminator**, comparando através da decisão do **Discriminator** com um array de 1s.

```
def generator_loss(fake_face):
   return
       cross_entropy(tf.ones_like(fake_face),
       fake_face)
```

3.4 Training

O training começa com o generator a receber um **random seed** como input que será utilizado para criar uma cara, de seguida, o discriminator é utilizado para classificar imagens reais vindo do dataset e imagens falsas produzidas pelo generator, o loss será então calculado para cada um desses modelos e os gradients serão atualizados respetivamente.

```
def train_step(images):
   noise = tf.random.normal([BATCH_SIZE,
       NOISE_DIM])

   with tf.GradientTape() as gen_tape,
       tf.GradientTape() as disc_tape:

       generated_images = generator(noise,
           training=True)

       real_output = discriminator(images,
           training=True)
       fake_output =
           discriminator(generated_images,
           training=True)

       gen_loss =
           generator_loss(fake_output)
       disc_loss =
           discriminator_loss(real_output,
           fake_output)

   gradients_of_generator =
       gen_tape.gradient(gen_loss,
       generator.trainable_variables)
   gradients_of_discriminator =
       disc_tape.gradient(disc_loss,
       discriminator.trainable_variables)

   generator_optimizer.apply_gradients(
   zip(gradients_of_generator,
       generator.trainable_variables))
   discriminator_optimizer.apply_gradients(
   zip(gradients_of_discriminator,
       discriminator.trainable_variables))

   return gen_loss, disc_loss
```

3.5 Resultados

Nesta versão de GAN foi treinado para: 10 epochs, 30 epochs, 100 epochs e 200 epochs, é possível observer uma evolução das imagens à medida que o modelo é treinado, assim como a variação que existe entre **Generation Loss** e **Discriminator Loss** ao longo dos epochs.

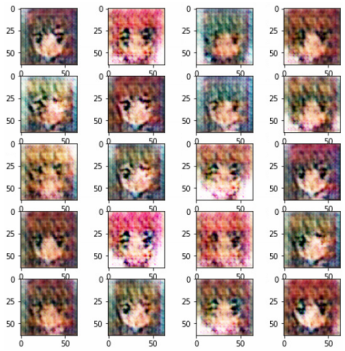

Figure 4: Resultado com 10 epochs

Para melhorar os resultados obtidos, poderá ser necessário treinar mais epochs, utilizar ou combinar datasets diferentes ou fazer modificações aos modelos utilizados.

Figure 5: Resultado com 200 epochs

4 Variational Autoencoders(VAE)

Autoencoder é uma espécie de neural network que é treinado para copiar o seu input para output, por exemplo, a imagem de um digito escrito à mão, o autoencoder primeiro irá utilizar o encode na imagem e depois faz o decode da imagem, tenta reconstruir a imagem. Um autoencoder irá aprender a compressionar a informação enquanto minimiza o erro da construção. Normalmente pode ser utilizado por exemplo para deteção de anomalias ou denoising de imagens. No entanto, VAE é como um **autoencoder**, só que em vez de converter o input para um map, VAE irá transformar o input em parametros da probabilidade como **Mean** ou **variance of gaussian**.

4.1 Encoder

Portanto, um VAE irá ter um **Encoder** que irá tentar compressionar as imagens para um latent space, que beneficia principalmente para geração de imagens, este modelo irá ter então uma camada **Dense** que recebe a imagem como input.

```
encoder_inputs = keras.Input(shape=(64,
    64, 3))

x = layers.Conv2D(32,
    kernel_size=KERNEL_SIZE,
    activation="relu", strides=STRIDES,
    padding="same", name =
    "conv_1")(encoder_inputs)
x = BatchNormalization()(x)

x = layers.Conv2D(64,
    kernel_size=KERNEL_SIZE,
    activation="relu", strides=STRIDES,
    padding="same", name = "conv_2")(x)
x = BatchNormalization()(x)

x = layers.Conv2D(128,
    kernel_size=KERNEL_SIZE,
    activation="relu", strides=STRIDES,
    padding="same", name = "conv_3")(x)
x = BatchNormalization()(x)

x = layers.Flatten()(x)
x = layers.Dense(256, activation="relu",
    name ="dense_1")(x)
x = BatchNormalization()(x)

z_mean = layers.Dense(512,
    name="z_mean")(x)
z_log_var = layers.Dense(512,
    name="z_log_var")(x)
z = Sampling()([z_mean, z_log_var])

encoder = keras.Model(encoder_inputs,
    [z_mean, z_log_var, z],
    name="encoder")
```

4.2 Decoder

VAE também irá ter um **Decoder** que irá tentar reconstruir a imagem original.

```
latent_inputs = keras.Input(shape=(512,))
x = layers.Dense(8 * 8 * 64,
    activation="relu")(latent_inputs)
x = BatchNormalization()(x)
x = layers.Reshape((8, 8, 64), name =
    "reshape_1")(x)


x = layers.Conv2DTranspose(128,
    kernel_size=KERNEL_SIZE,
    activation="relu", strides=STRIDES,
    padding="same", name ="transp_1")(x)

x = layers.Conv2DTranspose(64,
    kernel_size=KERNEL_SIZE,
    activation="relu", strides=STRIDES,
    padding="same", name = "transp_2")(x)

x = layers.Conv2DTranspose(32,
    kernel_size=KERNEL_SIZE,
    activation="relu", strides=STRIDES,
    padding="same", name = "transp_3")(x)

decoder_outputs =
    layers.Conv2DTranspose(3, 3,
    activation="sigmoid",
    padding="same")(x)
decoder = keras.Model(latent_inputs,
    decoder_outputs, name="decoder")
```

5 Resultados

Na versão GAN é fácil para o discriminator aprender a detetar arestas com blur, pelo que o generator irá ter a necessidade de produzir imagens mais nitidas para melhorar a loss, mas com VAE, a loss normalmente penaliza mais se o generator puser uma feature no lugar errado do que utilizar blur, pelo que os resultados acabam por ter bastante blur. Apesar de estes resultados apresentarem bastante blur, foram geradas caras com boa qualidade.

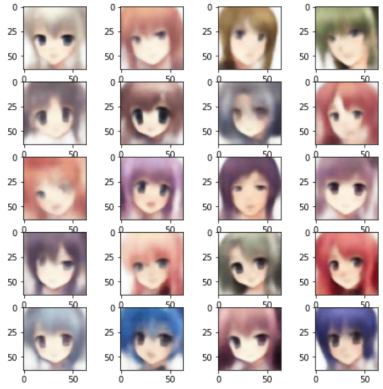

Figure 6: Resultado com modelo VAE

6 GAN com Wasserstein loss (WGAN)

Esta versão de GAN irá utilizar então o **Wasserstein loss** com o objetivo de ter uma melhor experiência de como loss é capaz de afectar o modelo GAN.

6.1 Critic e Generator

Para utilizarmos WGAN, é necessário deixar de usar **Discriminator** uma vez que o **Discriminator** é uma neural network que aprende a classificar as imagens se são reais ou não, usando um sigmoid activation na camada output e é alimentado com a utilização de cross entropy loss function, finalmente faz a predição de probabilidade de uma imagem ser real ou não. Em vez disso, irá usar então **Critic**, que simplesmente dá uma pontuação de realidade à uma imagem, portanto foi utilizado um novo modelo para **Critic**. Relativamente ao modelo **Generator** será o mesmo modelo utilizado na versão **GAN**.

```
critic = tf.keras.Sequential()
critic.add(tf.keras.layers.Conv2D(32,
    kernel_size=KERNEL_SIZE,
    strides=STRIDES, padding="same",
    input_shape=(IMAGE_SIZE, IMAGE_SIZE,
    3)))
critic.add(LeakyReLU())

critic.add(tf.keras.layers.Conv2D(64,
    kernel_size=KERNEL_SIZE,
    strides=STRIDES, padding="same"))
critic.add(LeakyReLU())

critic.add(tf.keras.layers.Conv2D(64,
    kernel_size=KERNEL_SIZE,
    strides=STRIDES, padding="same"))
critic.add(LeakyReLU())

critic.add(Flatten())
critic.add(Dense(1,activation="linear"))
```

6.2 Loss

Algorithm 1 WGAN, our proposed algorithm. All experiments in the paper u
the default values $\alpha = 0.00005$, $c = 0.01$, $m = 64$, $n_{\text{critic}} = 5$.

Require: : α, the learning rate. c, the clipping parameter. m, the batch s
n_{critic}, the number of iterations of the critic per generator iteration.
Require: : w_0, initial critic parameters. θ_0, initial generator's parameters.
while θ has not converged **do**
  for $t = 0, ..., n_{\text{critic}}$ **do**
    Sample $\{x^{(i)}\}_{i=1}^{m} \sim \mathbb{P}_r$ a batch from the real data.
    Sample $\{z^{(i)}\}_{i=1}^{m} \sim p(z)$ a batch of prior samples.
    $g_w \leftarrow \nabla_w \left[\frac{1}{m}\sum_{i=1}^{m} f_w(x^{(i)}) - \frac{1}{m}\sum_{i=1}^{m} f_w(g_\theta(z^{(i)}))\right]$
    $w \leftarrow w + \alpha \cdot \text{RMSProp}(w, g_w)$
    $w \leftarrow \text{clip}(w, -c, c)$
  end for
  Sample $\{z^{(i)}\}_{i=1}^{m} \sim p(z)$ a batch of prior samples.
  $g_\theta \leftarrow -\nabla_\theta \frac{1}{m}\sum_{i=1}^{m} f_w(g_\theta(z^{(i)}))$
  $\theta \leftarrow \theta - \alpha \cdot \text{RMSProp}(\theta, g_\theta)$
end while

Figure 7: Algoritmo WGAN

Como podemos observar na imagem 10, as funções loss da critic e generator são respetivamente **[average critic score on real faces] – [average critic score on fake faces]** e **Generator Loss = -[average critic score on fake images]**

```
# Wasserstein loss for the critic
def d_wasserstein_loss(pred_real,
    pred_fake):
   real_loss = tf.reduce_mean(pred_real)
   fake_loss = tf.reduce_mean(pred_fake)
   return fake_loss - real_loss

# Wasserstein loss for the generator
def g_wasserstein_loss(pred_fake):
   return -tf.reduce_mean(pred_fake)
```

```
- d_loss: -189.5314 - g_loss: -158.7919
- d_loss: -1756.4604 - g_loss: -4945.5054
- d_loss: -2443.5486 - g_loss: -26379.2461
- d_loss: 62.0237 - g_loss: -32147.8027
- d_loss: 699.0718 - g_loss: -10340.7246
- d_loss: 197.9997 - g_loss: -1739.7673
- d_loss: 17.3131 - g_loss: -204.0571
- d_loss: -1.6431 - g_loss: -11.9808
- d_loss: -3.6939 - g_loss: -4.5089
- d_loss: -6.4914 - g_loss: 2.9774
- d_loss: -5.4658 - g_loss: 33.5040
```

Figure 8: Loss obtido com 50 epochs

6.3 Training

Mais uma vez, como é possível observar na imagem 10, para cada batch devemos:

1. Treinar o Generator e o Critic
2. Calcular o loss para Generator e Critic
3. Calcular o gradient penalty
4. Multiplicar este penalty por uma constante weight factor
5. Adicionar o penalty ao loss de Critic
6. Retornar o loss de Generator e Critic como loss dictionary

6.4 Resultados

Para esta ultima versão, utilizou-se 10 epochs, 20 epochs e 50 epochs.

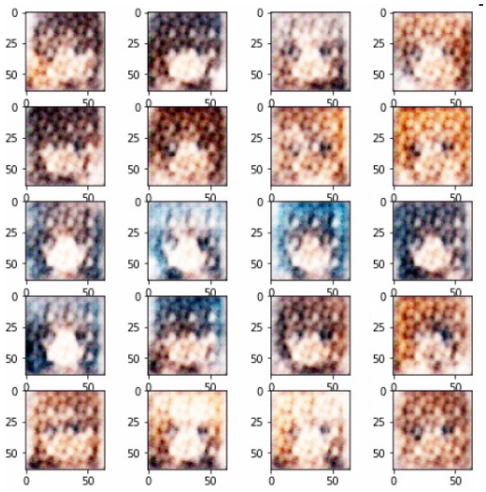

Figure 9: WGAN com 10 epochs

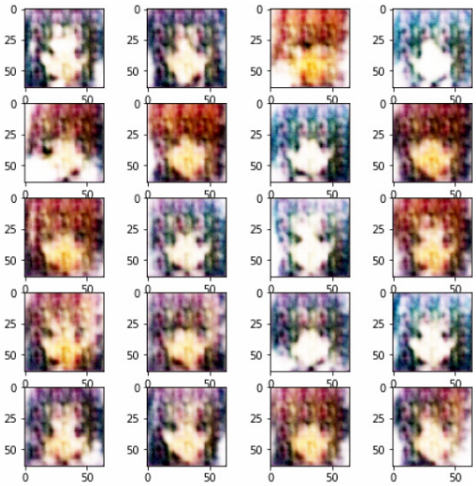

Figure 10: WGAN com 50 epochs

Para obter melhores resultados, talvez seja necessário melhorar o modelo output aumentando o tamanho da network, por exemplo, aumentar o filters para **Conv2D** e **Conv2DTranspose** ou simplesmente treinar mais epochs.

7 Conclusão e Trabalho Futuro

Ao longo deste trabalho prático, explorou-se três versões: **GAN**, **VAE** e **WGAN**, cada um com as suas vantagens e desvantagens. O objetivo deste trabalho prático era conseguir gerar caras de personagens de anime, algo que todos os modelos conseguiram, e também estudar e entender como uma loss pode afectar nestes modelos. Como seria de esperar, os resultados não são perfeitos, é necessário mais tempo de treino, tentar novos modelos.

Para o trabalho futuro, seria interessante experimentar alternativas como **Conditional Generative Adversarial Network**, **StyleGan**, entre outros.