Used a GAN with Wasserstein loss and gradient penalty to achieve Lipschitz continuity.

Trained on 63k pictures of anime faces (https://www.kaggle.com/datasets/splcher/animefacedataset) to generate new faces.

Example of training pictures:

Critic architecture:

```
def create_critic(self):
    critic_input = Input(shape = (48,48,3), name = "input_critic")

    critic_x = Conv2D(512,kernel_size = 3,activation = 'linear', name = "Conv2D_critic_4")(critic_input)
    critic_x = Activation("relu",name = "activation_critic_4")(critic_x)

    critic_x = Conv2D(256,kernel_size = 3,activation = 'linear', name = "Conv2D_critic_5")(critic_input)
    critic_x = Activation("relu",name = "activation_critic_5")(critic_x)

    critic_x = Conv2D(128,kernel_size = 3,activation = 'linear', name = "Conv2D_critic_6")(critic_x)
    critic_x = Activation("relu",name = "activation_critic_6")(critic_x)

    critic_x = Conv2D(64,kernel_size = 3,activation = 'linear', name = "Conv2D_critic_7")(critic_x)
    critic_x = Activation("relu",name = "activation_critic_7")(critic_x)

    critic_x = Conv2D(64,kernel_size = 3,activation = 'linear', name = "Conv2D_critic_8")(critic_x)
    critic_x = Activation("relu",name = "activation_critic_8")(critic_x)

    critic_x = Flatten(name = "flatten_critic")(critic_x)

    critic_x = Dense(150,activation = 'relu',name = "dense_critic_1")(critic_x)
    critic_x = Dense(50,activation = 'relu',name = "dense_critic_2")(critic_x)
    critic_output  = Dense(1,activation = 'linear', name = "dense_critic_output")(critic_x)

    critic = Model(inputs = [critic_input] , outputs = [critic_output])
    return critic
```

Wasserstein loss and Gradient penalty term

```
def wasserstein(y_true, y_pred):
    """Keras version of wasserstein loss"""
    return -K.mean(y_true * y_pred)
```

```
def gradient_penalty(self,interpolated):

    with tf.GradientTape() as gp_tape:

        gp_tape.watch(interpolated)      #tensors arent watch by default, only variables. picture is a tensor
        critic_prediction = self.critic(interpolated)

    grads = gp_tape.gradient(critic_prediction, [interpolated])[0]
    norm = tf.sqrt(tf.reduce_sum(tf.square(grads), axis=[1, 2, 3]))
    gp = tf.reduce_mean((norm - 1.0) ** 2)
    return gp
```

Critic training step:

```
def train_critic(self,batch_input_shape,real_images,fake_images,interpolated_images):

    with tf.GradientTape() as c_tape:

        critic_real_target = tf.ones((batch_input_shape,1), dtype=tf.float32)
        critic_fake_target = -tf.ones((batch_input_shape,1), dtype=tf.float32)

        critic_real_pred = self.critic(real_images)
        critic_fake_pred = self.critic(fake_images)

        critic_real_loss = wasserstein(critic_real_target, critic_real_pred)
        critic_fake_loss = wasserstein(critic_fake_target, critic_fake_pred)
        gp = self.gradient_penalty(interpolated_images)

        critic_loss = critic_real_loss + critic_fake_loss +  self.gp_weight * gp

    critic_gradients = c_tape.gradient(critic_loss, self.critic.trainable_variables)
    self.critic_optimizer.apply_gradients(zip(critic_gradients, self.critic.trainable_variables))

    return  critic_loss,critic_real_loss,critic_fake_loss,gp
```

Generator architecture:

```
def create_generator(self):
    random_latent_vectors_input = Input(shape=(self.z_dim,), name = "input_generator")
    generator_x = Dense(576, activation = "linear",name = "dense_generator")(random_latent_vectors_input)
    generator_x = Reshape(target_shape = (3,3,64)) (generator_x)

    generator_x = Conv2DTranspose(128,(3,3),(2,2),padding = "same", name = "Conv2DTranspose_generator_1")(generator_x)
    generator_x = BatchNormalization(name = "batch_norm_generator_1")(generator_x)
    generator_x = Activation("relu",name = "activation_generator_1")(generator_x)

    generator_x = Conv2DTranspose(128,(3,3),(2,2),padding = "same", name = "Conv2DTranspose_generator_2")(generator_x)
    generator_x = BatchNormalization(name = "batch_norm_generator_2")(generator_x)
    generator_x = Activation("relu",name = "activation_generator_2")(generator_x)

    generator_x = Conv2DTranspose(32,(3,3),(2,2),padding = "same", name = "Conv2DTranspose_generator_3")(generator_x)
    generator_x = BatchNormalization(name = "batch_norm_generator_3")(generator_x)
    generator_x = Activation("relu",name = "activation_generator_3")(generator_x)

    generator_x = Conv2DTranspose(4,(3,3),(2,2),padding = "same", name = "Conv2DTranspose_generator_4")(generator_x)
    generator_x = BatchNormalization(name = "batch_norm_generator_4")(generator_x)
    generator_x = Activation("relu",name = "activation_generator_4")(generator_x)

    generator_x = Conv2DTranspose(3,(3,3),(1,1),padding = "same", name = "Conv2DTranspose_generator_output")(generator_x)
    generator_x = BatchNormalization(name = "batch_norm_generator_output")(generator_x)
    generator_output = Activation("sigmoid",name = "activation_generator_output")(generator_x)

    generator = Model(inputs = [random_latent_vectors_input], outputs = [generator_x], name = "generator")
    return generator
```

Generator training step:

```
def train_generator(self,batch_input_shape,vector):

    generator_target =tf.ones((batch_input_shape,1), dtype=tf.float32)

    with tf.GradientTape() as g_tape:
        generated_images = self.generator(vector)
        critic_prediction = self.critic(generated_images)
        generator_loss = wasserstein(generator_target, critic_prediction)
    generator_gradients = g_tape.gradient(generator_loss, self.generator.trainable_variables)
    self.generator_optimizer.apply_gradients(zip(generator_gradients, self.generator.trainable_variables))

    return generator_loss
```

WGAN-GP constructor. The important thing here are the two different optimizers for critic and generator, so each one has its own configuration for training each model. Use 5 critic steps and 10 scalar multiplier for the gp term in the critic loss as the paper says. The size of z_dim vector for generating images is 15.

```
class WGAN_GP(tf.keras.Model):
    """WGAN-GP model. Creates both generator and critic, and trains them"""
    def __init__(self,z_dim,critic_steps,gp_weight):
        super().__init__()
        self.z_dim = z_dim
        self.critic_steps = critic_steps
        self.gp_weight = gp_weight
        self.critic = self.create_critic()
        self.generator = self.create_generator()

    def compile(self, critic_optimizer, generator_optimizer):
        super().compile()
        self.critic_optimizer = critic_optimizer
        self.generator_optimizer = generator_optimizer
```

WGAN-GP complete training process. Model input is a batch of pictures. We generate a fake image with generator and interpolate both fake and real images with an interpolation layer. Then we call the critic training step x times (define in the constructor) and then one step in the generator.

```
def train_step(self, data):

    batch_input_shape = tf.shape(data, name = "WGAN_GP_batch_size")[0]
    interpolation_layer = Interpolate("interpolation_layer")

    real_images = data
    random_latent_vectors = tf.random.normal(shape=(batch_input_shape,self.z_dim))
    fake_images = self.generator(random_latent_vectors)
    intepolated_images = interpolation_layer([real_images,fake_images])

    for i in range(self.critic_steps):
        critic_loss,critic_real_loss,critic_fake_loss,gp = self.train_critic(batch_input_shape,real_images,fake_images,intepolated_images)

    generator_loss = self.train_generator(batch_input_shape,random_latent_vectors)

    return {"generator_loss":generator_loss,"critic_loss":critic_loss,"critic_real_loss":critic_real_loss,"critic_fake_loss":critic_fake_loss,"gp":gp}
```

Interpolation Layer:

```
class Interpolate(tf.keras.layers.Layer):
    """keras layer thats interpolates two images"""
    def __init__(self,name = "interpolation"):
        super(Interpolate,self).__init__(name = name)
    def call(self, inputs, **kwargs):
        batch_input_shape = tf.shape(inputs[0], name = "input_shape")[0]
        alpha = tf.random.uniform((batch_input_shape, 1, 1, 1))
        return (alpha * inputs[0]) + ((1 - alpha) * inputs[1])
```

As per callbacks i use three. A learning rate scheduler, a callback that saves both critic and generator weights, as well as their respective optimizer configurations, and one that generates images given a defined frequency. Note that each model and their respective optimizer is treated differently. We could use two different learning scheduler formulas for each optimizer.

```
def lr_schedule(epoch, lr):
    """Staricase learning rate schduler. Reduce learning rate every 10 epochs"""
    if epoch == 0:
        return lr
    elif epoch % 50 == 0:
        return lr * 0.1
    else:
        return lr


class LearningRateScheduler(tf.keras.callbacks.Callback):
    """Learning rate scheduler callback. Applies different schedulers to critic and generator optimizers"""
    def __init__(self, generator_schedule, critic_schedule):
        super(LearningRateScheduler, self).__init__()

        self.generator_schedule = generator_schedule
        self.critic_schedule = critic_schedule

    def on_epoch_begin(self, epoch, logs=None):
        if not hasattr(self.model.optimizer, "lr"):
            raise ValueError('Optimizer must have a "lr" attribute.')

        generator_lr = float(tf.keras.backend.get_value(self.model.generator_optimizer.learning_rate))
        critic_lr = float(tf.keras.backend.get_value(self.model.critic_optimizer.learning_rate))

        generator_new_lr = self.generator_schedule(epoch, generator_lr)
        critic_new_lr = self.critic_schedule(epoch, critic_lr)

        tf.keras.backend.set_value(self.model.generator_optimizer.lr, generator_new_lr)
        tf.keras.backend.set_value(self.model.critic_optimizer.lr, critic_new_lr)

        print(f"Generator lr in epoch {epoch} is {generator_new_lr}")
        print(f"Critic lr in epoch {epoch} is {critic_new_lr}")
```

```python
class Save_model(tf.keras.callbacks.Callback):
    """Saves weights and optimizers of critic and generator with the given frecuency"""
    def __init__(self,save_frecuency,path):
        self.save_frecuency = save_frecuency
        self.path = path + "/model"

        create_folder(self.path)

    def on_epoch_end(self, epoch, logs=None):
        if epoch % self.save_frecuency == 0:

            print("Saving weights")

            create_folder(f"{self.path}/epoch_{epoch}")

            self.model.generator.save_weights(f"{self.path}/epoch_{epoch}/generator_weights.hdf5")
            self.model.critic.save_weights(f"{self.path}/epoch_{epoch}/critic_weights.hdf5")

            print("Saving optimizer weights")

            output_file_generator = f"{self.path}/epoch_{epoch}/generator_optimizer.npy"
            output_file_critic = f"{self.path}/epoch_{epoch}/critic_optimizer.npy"

            np.save(output_file_generator, self.model.generator_optimizer.get_weights())
            np.save(output_file_critic, self.model.critic_optimizer.get_weights())


class Save_images(tf.keras.callbacks.Callback):
    """Saves numpy arrays generated by generator with the given frecuency"""
    def __init__(self,save_frecuency,path,amount_of_images):
        self.save_frecuency = save_frecuency
        self.amount_of_images = amount_of_images
        self.path = path + "/images"

        create_folder(self.path)

    def on_epoch_end(self, epoch, logs=None):
        if epoch % self.save_frecuency == 0:

            create_folder(f"{self.path}/epoch_{epoch}")

            print("Generating vector")
            vector = np.random.normal(size=(self.amount_of_images, self.model.z_dim))
            print("Generating image")
            images = self.model.generator.predict(vector)
            print("Saving image")
            np.save(f"{self.path}/epoch_{epoch}/images", images)
```

Disclaimer:

The process for training a GAN, especially a WGAN-GP, is long and quite tedious. I only have one computer with one gpu, and use it for work, university, and entertainment. I could only train the estimator at night while sleeping. For resuming training after the last epoch, I need to save both weights

from the models as well and optimizer configurations. There is a trick for loading the optimizers configurations. Each time we load the config after creating the optimizers, we need to use them with the trainable parameters of the models to define the shape of the optimizer's weights. We also need to do this before resuming training, but without moving the wights from the models. We apply gradients to the zero array with respect to the trainable parameters, this allows the optimizers to have the weights shape defined for loading the saved ones, as well as not moving the trainable parameters of the model. The process is shown in the picture below:

```
#Logic for loading optimizer and model weights to continue training
#Need to specify the dimensions of the optimizer weights, done by applaying gradients to model trainable parameters with respect to a 0 vector
if resume_training:
    model.generator.load_weights(epoch_load + "/generator_weights.hdf5")
    model.critic.load_weights(epoch_load + "/critic_weights.hdf5")

    generator_opt_weights = np.load(epoch_load + "/generator_optimizer.npy", allow_pickle=True)
    critic_opt_weights = np.load(epoch_load + "/critic_optimizer.npy", allow_pickle=True)

    generator_grad_vars = model.generator.trainable_weights
    critic_grad_vars = model.critic.trainable_weights

    generator_zero_grads = [tf.zeros_like(w) for w in generator_grad_vars]
    critic_zero_grads = [tf.zeros_like(w) for w in critic_grad_vars]

    opt2.apply_gradients(zip(generator_zero_grads, generator_grad_vars))
    opt1.apply_gradients(zip(critic_zero_grads, critic_grad_vars))

    opt2.set_weights(generator_opt_weights)
    opt1.set_weights(critic_opt_weights)
```

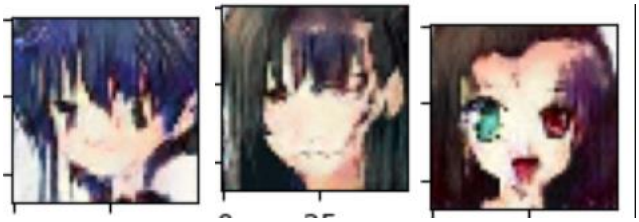
I trained the generator for approximately 150 epochs. The losses for the first 50 epochs look as follows:

No sign of collapse or critic converging to zero, that's a win. After a few epochs the losses find some "equilibrium".

If we trained the model with a lot more epochs, the generated images would be more constant and more defined.

Some images that look quite nice:

Some images that look suspicious in terms of quality:



A thing to note is the eyes of the generated images. The model constantly generates eyes of different colours and sizes for each face. Maybe is bias in the pictures and the generator found out that by having different eyes for each face it can fool the critic.