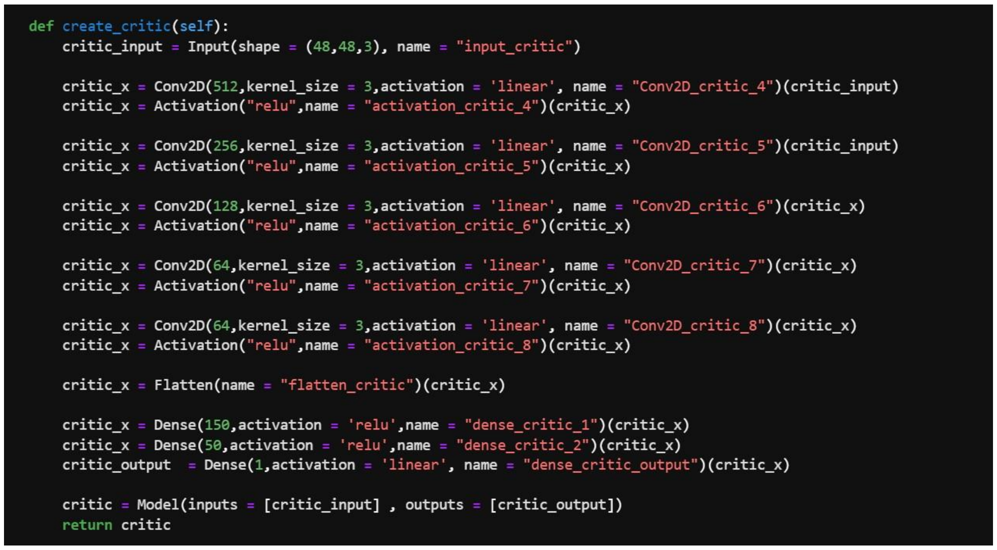
Used a GAN with Wasserstein loss and gradient penalty to achieve Lipschitz continuity.

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

Example of training pictures:

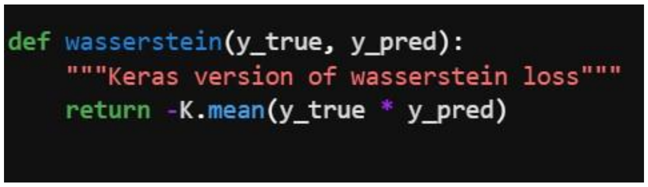
Class: 0.0
Class: 0.0
Class: 0.0
Class: 0.0
Class: 0.0

Critic architecture:

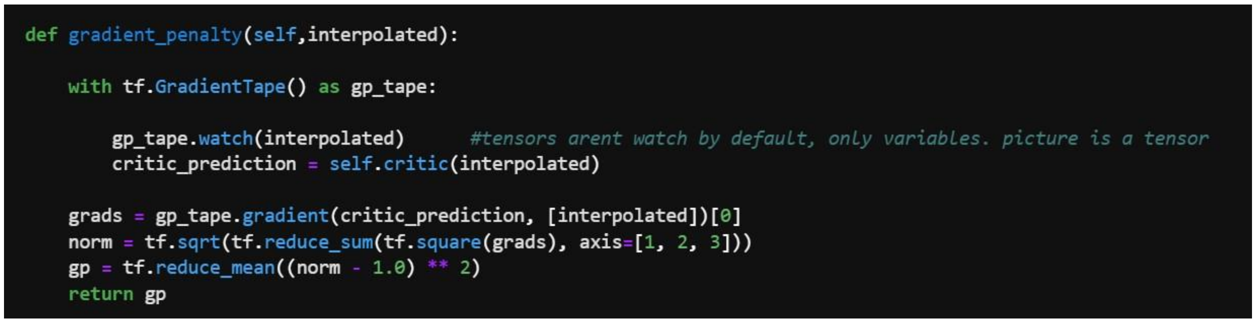


```
def create_critic(self):
    critic_input = Input(shape = (48,48,3), name = "input_critic")

    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

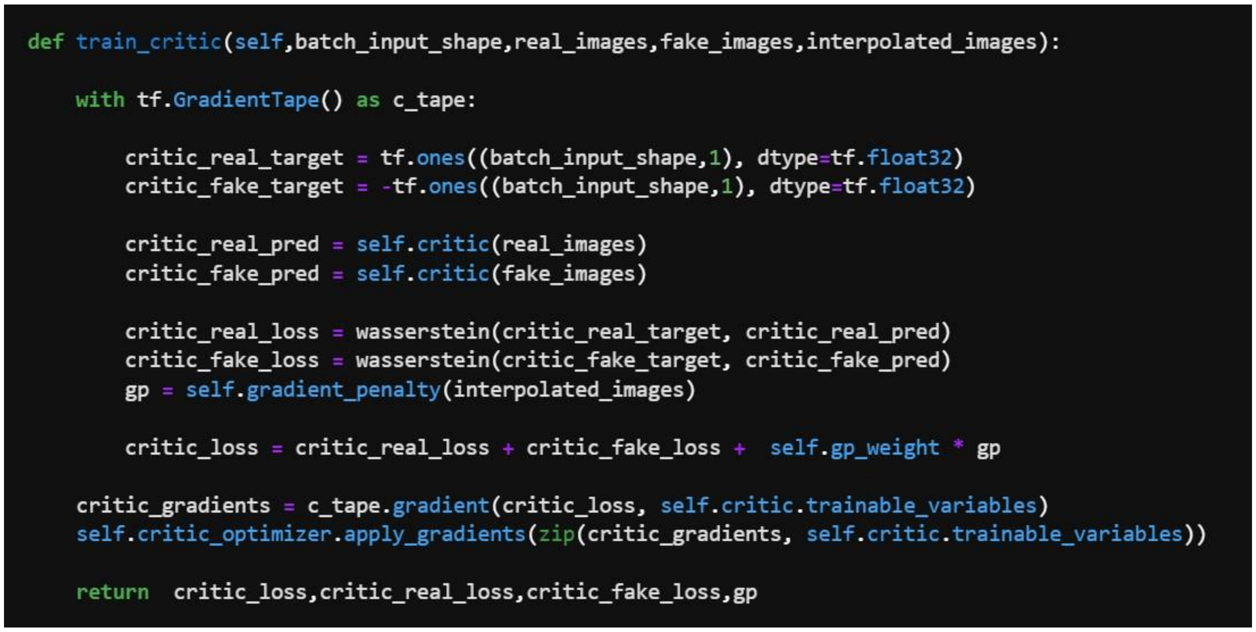


```python
def wasserstein(y_true, y_pred):
    """Keras version of wasserstein loss"""
    return -K.mean(y_true * y_pred)
```



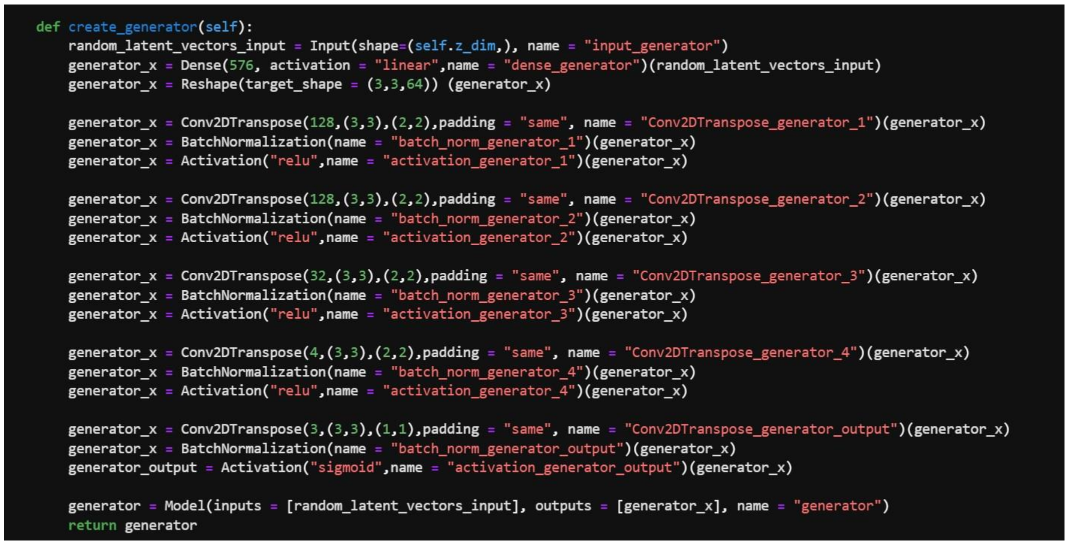
```python
def gradient_penalty(self,interpolated):

    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:



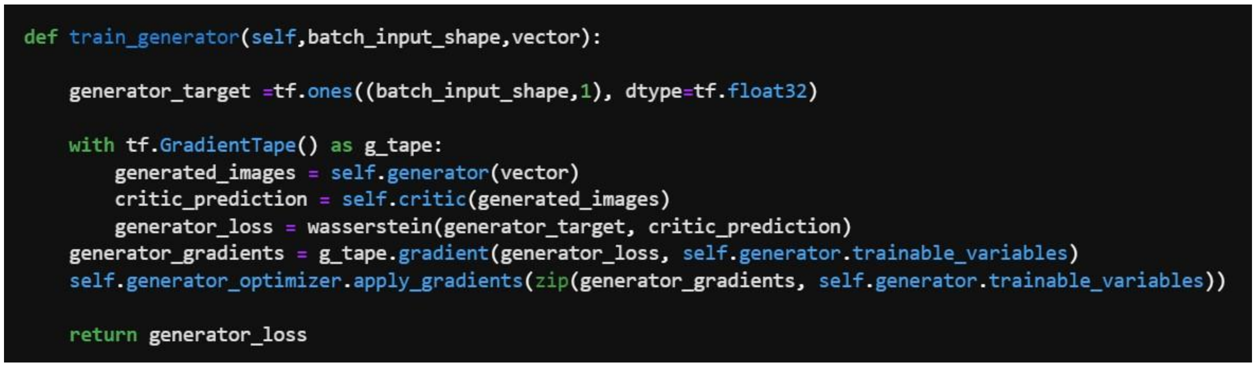
```python
def train_critic(self,batch_input_shape,real_images,fake_images,interpolated_images):

    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:



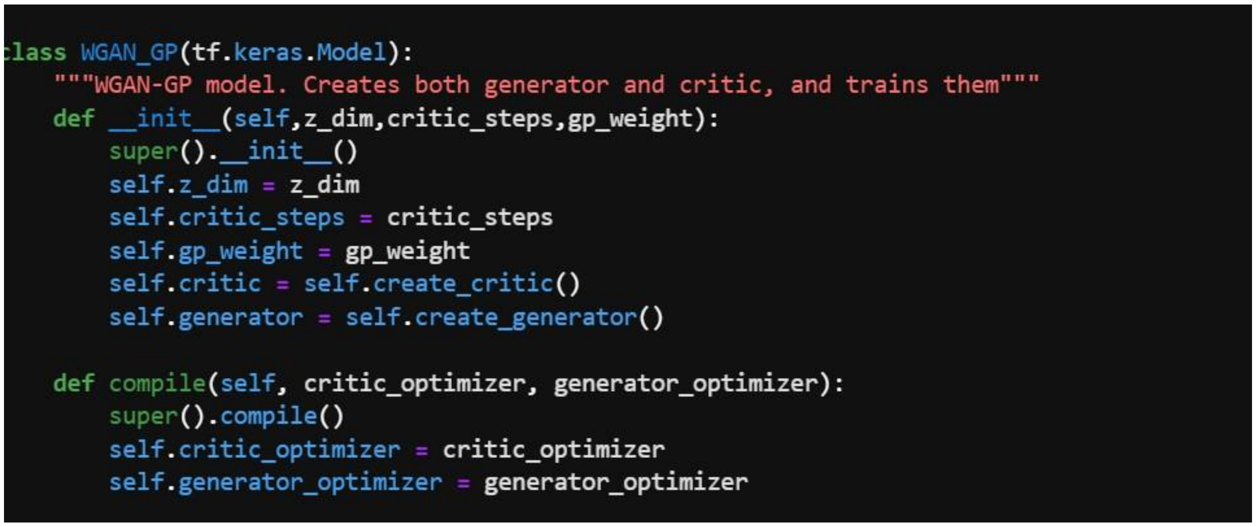
```python
def create_generator(self):
    random_latent_vectors_input = Input(shape=(self.z_dim,), name = "input_generator")
    generator_x = Dense(576, activation = "linear",name = "dense_generator")(random_latent_vectors_input)
    generator_x = Reshape(target_shape = (3,3,64)) (generator_x)

    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:



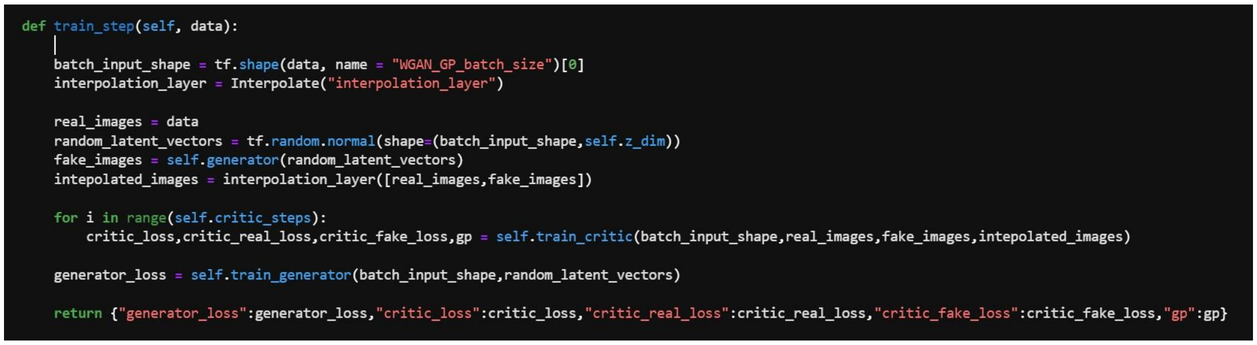
```python
def train_generator(self,batch_input_shape,vector):

    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
```

The WGAN-GP (Wasserstein Generative Adversarial Network with Gradient Penalty) constructor is characterized by distinct optimization strategies for both the critic and generator components. This separation ensures tailored training configurations for each model. Specifically, the critic undergoes training over 5 steps, and the gradient penalty term in the critic's loss is scaled using a scalar multiplier of 10, as recommended in the original research paper. Additionally, the dimensionality of the **z_dim** vector utilized for image generation is set to 15.



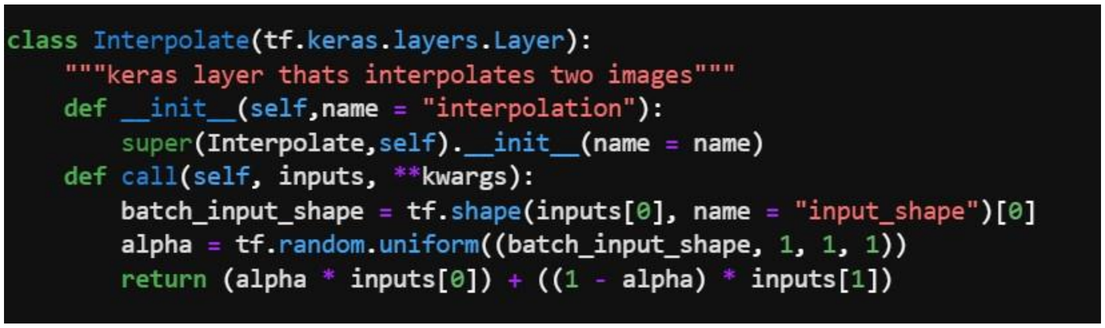
```python
lass WGAN_GP(tf.keras.Model):
    """WGAN-GP model. Creates both generator and critic, and trains them"""
    def __init__(self,z_dim,critic_steps,gp_weight):
        super().__init__()
        self.z_dim = z_dim
        self.critic_steps = critic_steps
        self.gp_weight = gp_weight
 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

In the WGAN-GP training process, a batch of images serves as the model input. Initially, a synthetic image is produced using the generator. Subsequently, both the generated and real images undergo interpolation via a specialized interpolation layer. Following this, the training sequence is executed: the critic's training step is invoked a specified number of times, as defined within the constructor. After completing the critic's training iterations, a single training step is performed for the generator.



```python
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}
```

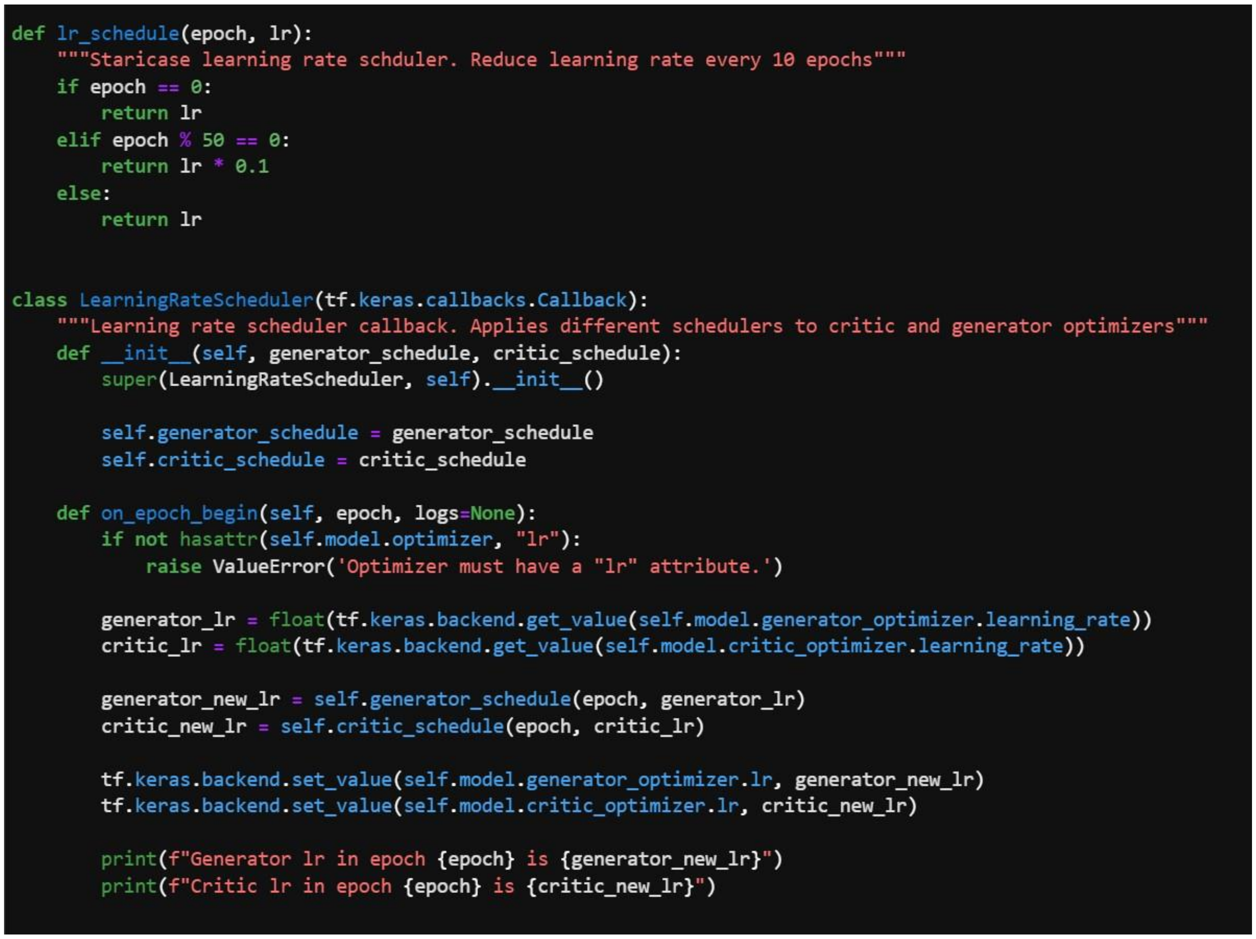
Interpolation Layer:



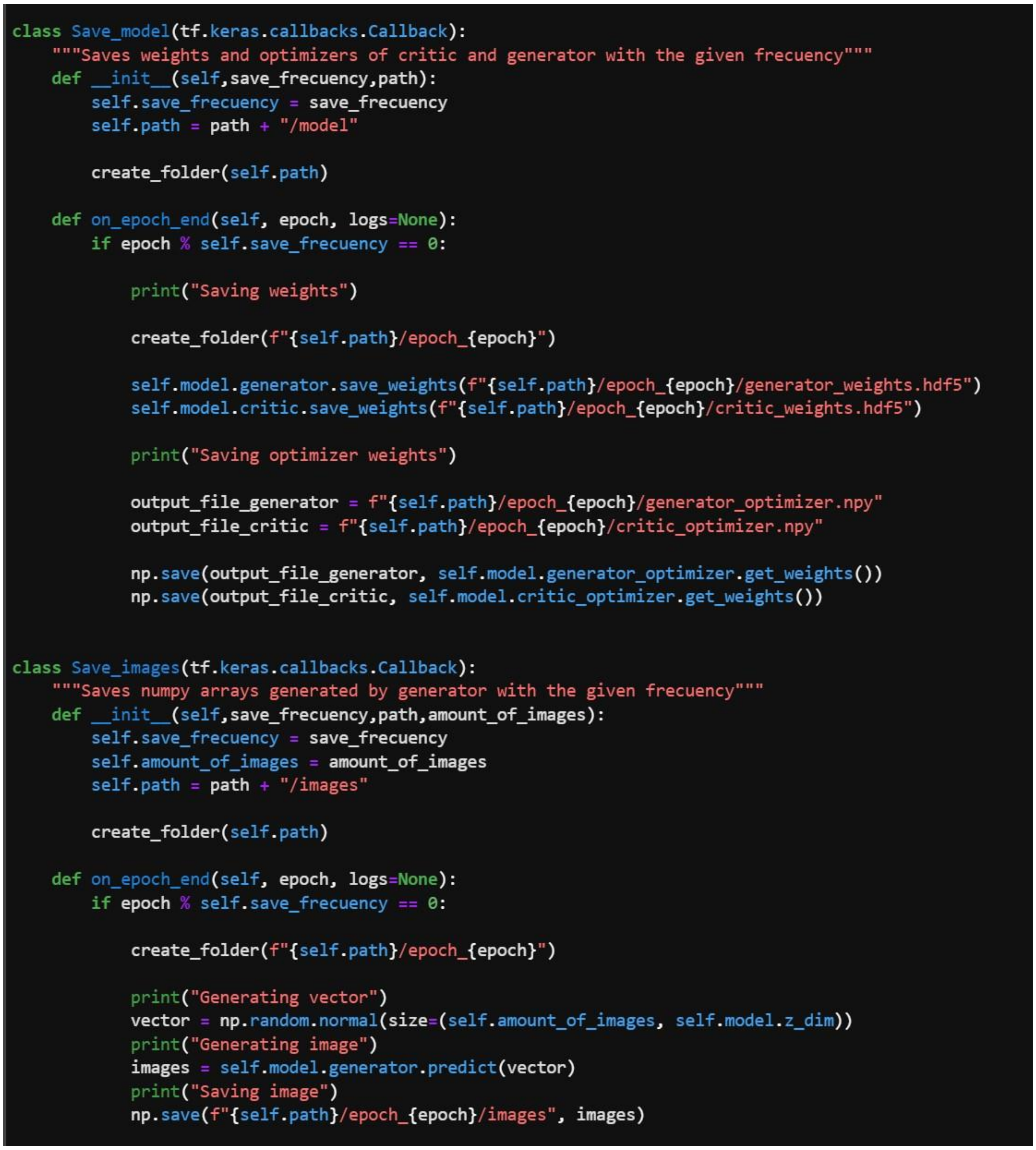
```python
class Interpolate(tf.keras.layers.Layer):
    """keras layer thats interpolates two images"""
    def __init__(self,name = "interpolation"):
        super(Interpolate,self).__init__(name = name)
    def call(self, inputs, **kwargs):
        batch_input_shape = tf.shape(inputs[0], name = "input_shape")[0]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

For callbacks, I employ three distinct mechanisms. Firstly, a learning rate scheduler dynamically adjusts the learning rate during training. Secondly, there's a callback responsible for saving the weights of both the critic and generator. This callback also preserves the configurations of their individual optimizers. Lastly, there's a callback designed to generate images at specified intervals throughout the training process.

It's essential to highlight the differentiated treatment of each model and its corresponding optimizer within these callbacks. Additionally, it's worth noting that separate learning rate scheduling formulas can be employed for each optimizer, offering flexibility and customization in the training dynamics.



```python
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)
```

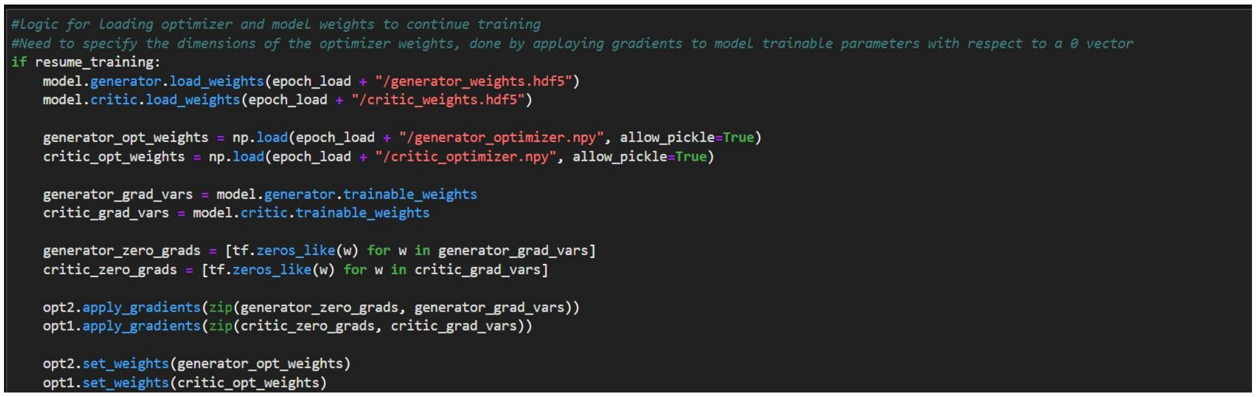
Disclaimer:

Training a GAN, especially the WGAN-GP variant, demands considerable time and meticulous attention. With only a single computer equipped with a GPU, and its shared roles in work, university tasks, and personal entertainment, my training opportunities were limited to overnight sessions.

Resuming training after halting requires careful consideration. It's essential to save not only the weights of the models but also the configurations of their respective optimizers. A crucial aspect to note is the unique procedure for reloading the optimizer configurations. When reloading the optimizer configuration after its creation, it's imperative to employ them in conjunction with the models' trainable parameters. This action delineates the shape of the optimizer's weights.

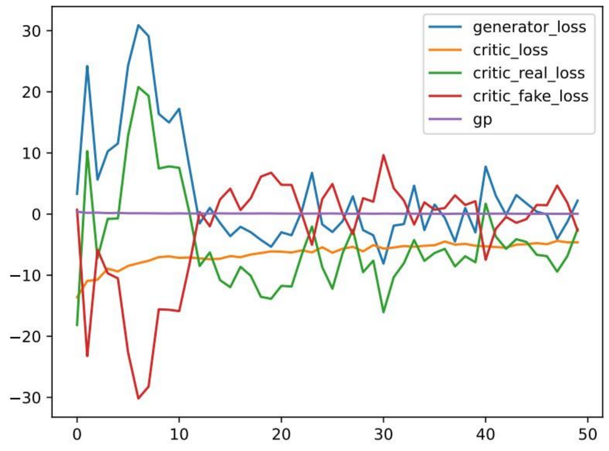
Before resuming the training process, a preparatory step is crucial. However, this step must be executed without altering the weights of the models. A workaround involves applying gradients to a zero-filled array concerning the trainable parameters. This action ensures that the optimizer's weights are correctly shaped for the subsequent loading of saved configurations, while simultaneously maintaining the integrity of the models' trainable parameters.

For a visual representation and clarity on this process, please refer to the accompanying image 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)
```

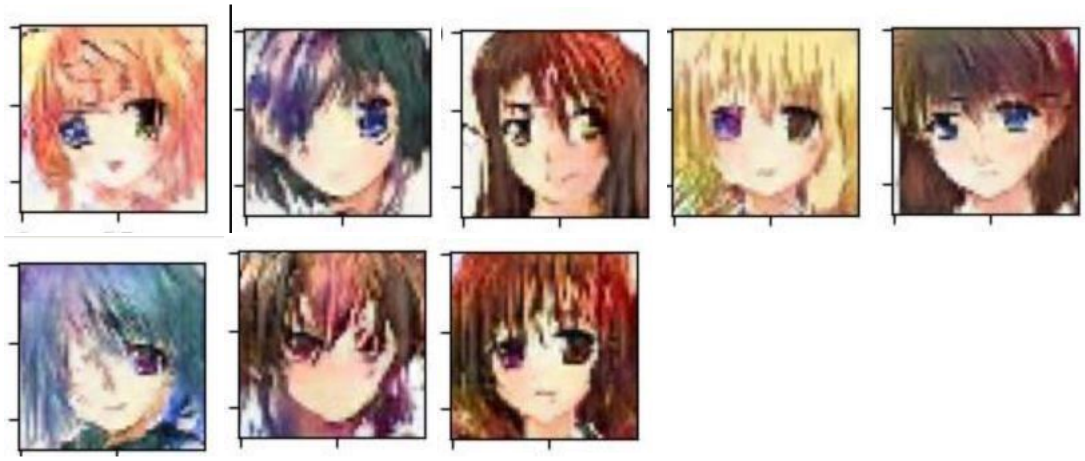
I trained the generator for roughly 150 epochs. Here are the loss values for the initial 50 epochs:



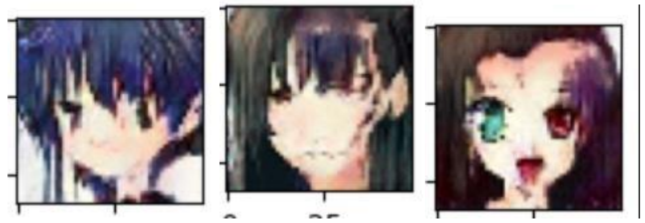
There's no evidence of the generator collapsing or the critic converging to zero, which is a positive outcome. After several epochs, the loss values stabilize, indicating a certain level of equilibrium.

With an extended training duration, the generated images would likely exhibit greater consistency and clarity.

Some images that look quite nice:



Some images that look suspicious in terms of quality:



It's worth noting the variability in the eyes of the generated images. The model consistently produces eyes with diverse colors and sizes for each face. This behavior might be attributed to biases in the training data. The generator could have learned that by introducing such variations in the eyes, it becomes more challenging for the critic to discern the generated images from real ones.