# Semi-Supervised Learning with Generative Adversarial Networks

Ignacio Dominguez

## I. Introducción

El presente trabajo se centra en el Aprendizaje Semi-Supervisado con Redes Generativas Adversariales (SGAN), una extensión de las Redes Generativas Adversariales.

El articulo, "Semi-Supervised Learning with Generative Adversarial Networks," de Augustus Odena, Odena [2016] sienta las bases de esta monografía. Esta investigación aborda la cuestión de cómo mejorar la eficiencia de un clasificador en un entorno de aprendizaje semi-supervisado, donde las etiquetas de clase disponibles son escasas.

En particular, el trabajo propone modificaciones a un generador y un clasificador en un marco de Generative Adversarial Networks (GANs) semi-supervisado, conocido como SGAN. La novedad radica en que el discriminador de la GAN es modificado para realizar tareas de clasificación con multiples salidas, no solo de verdadero-falso. Esto no solo mejora el rendimiento de la clasificación en tareas semi-supervisadas, sino que también da lugar a la generación de datos de alta calidad.

En esta monografía, se explorarán las bases teóricas de las redes Convolucionales (ConvNets) y las GANs, así como las modificaciones realizadas en el modelo SGAN. Se describirá en detalle cómo estas ideas fundamentales se combinan para lograr una mayor eficiencia en tareas de clasificación semi-supervisada y una mejora en la calidad de los datos generados.

Además, se presentarán las pruebas extras realizadas como parte de este trabajo, donde se ha agregado ruido controlado al modelo y se ha observado su impacto en el rendimiento y la generación de datos.

## II. Antecedentes

En esta sección, proporcionaremos una visión general de los conceptos clave necesarios para comprender el trabajo presentado en esta monografía. Comenzaremos con las redes convolucionales, luego exploraremos las GANs y, finalmente, abordaremos cómo estas ideas fundamentales se aplican en el las SGAN.

### *A. Redes Convolucionales*

Las redes convolucionales, son redes neuronales para el procesamiento de datos con estructura de cuadrícula, como imágenes y datos de series temporales. A diferencia de las redes neuronales completamente conectadas, las ConvNets utilizan capas convolucionales para detectar características locales en los datos de entrada.

En cada capa convolucional, se aplican filtros o ventanas que se deslizan a lo largo de la entrada para realizar operaciones de convolución. Esto permite que la red aprenda características como bordes, texturas y patrones locales, lo que es extremadamente útil para el procesamiento de imágenes.

### *B. Redes Generative Adversarial Networks*

Una GAN consta de dos redes principales: un generador y un discriminador. El generador crea muestras, mientras que el discriminador evalúa si una muestra es real o generada.

Estas dos redes se entrenan simultáneamente en un proceso de competencia. El generador busca mejorar su capacidad para generar muestras realistas, mientras que el discriminador se entrena para mejorar su capacidad de distinguir entre muestras reales y generadas.

### *C. Aplicación en el Paper Referenciado*

El trabajo presentado en este artículo se basa en la idea de extender las GANs al contexto de aprendizaje semi-supervisado.

En lugar de simplemente generar datos, se modifica el discriminador de la GAN para que también realice la clasificación de clases en una tarea dada. Esto permite el aprendizaje conjunto de un generador y un clasificador, lo que resulta en un enfoque más eficiente para la clasificación semi-supervisada y la generación de datos de alta calidad.

En la siguiente sección, exploraremos más a fondo el modelo SGAN y sus resultados en tareas semi-supervisadas y de generación de datos.

## III. Modelo SGAN

La red generativa adversarial semi-supervisada (SGAN) es una extensión innovadora de las GANs. Introduce una modificación en la arquitectura tradicional de GANs al forzar al discriminador (D) a producir no solo una probabilidad de autenticidad para las muestras generadas, sino también etiquetas de clase. Esto significa que, en lugar de simplemente distinguir entre muestras auténticas y generadas, el discriminador ahora tiene la capacidad de clasificar las muestras generadas en diferentes clases, lo que lo convierte simultáneamente en un discriminador y un clasificador (D/C).

En el paper de Ogata, se concluyen dos cosas, que la SGAN ofrece ventajas al mejorar la calidad de las muestras generadas comparadas a una GAN normal y que se obtiene un mejor rendimiento de clasificación en comparación de un clasificador tradicional, esto es especialmente cierto en escenarios donde el conjunto de datos de entrenamiento es limitado.

## IV. Experimentos y Resultados

En esta sección, se describen las pruebas realizadas para evaluar el impacto de ruido en la entrada de tanto los datos reales como en la salida del generador.

El experimento se centró en determinar cómo el nivel de ruido variable afecta el rendimiento del modelo. No se recopilaron datos adicionales, y aparte de la agregación de ruido, la arquitectura de la red neuronal se mantuvo igual.

### A. Configuración Experimental

- **Modelo de Red Neuronal**: Se utilizó una arquitectura de red neuronal preexistente sin cambios en su estructura. (figura 1)
- **Introducción de Ruido**: La característica principal del experimento fue la variación de la cantidad de ruido en los datos de entrada y en la salida del generador a lo largo del tiempo. El ruido tiene una estructura constante, pero experimenta desplazamientos aleatorios en su ubicación a medida que evoluciona.
  - **Generador 1:** Este generador crea señales y luego se le agrega una línea de ruido horizontal y otra vertical.
  - **Generador 2:** El segundo generador aumenta a tres horizontales y tres verticales.
  - **Generador 3:** El tercer generador introduce líneas de ruido dinámico aleatorio que varían entre 1 y 5 líneas. A continuación, se muestra un ejemplo de una señal generada por este generador.

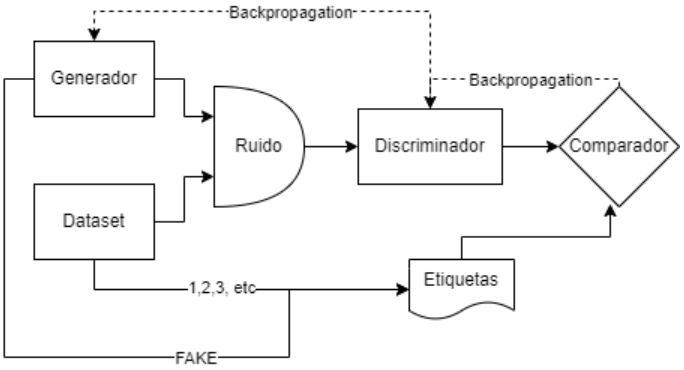


Fig. 1. Resultados del entrenamiento de las 3 SGAN

### B. Procedimiento de Entrenamiento

El procedimiento de entrenamiento se diseñó de la siguiente manera:

1) **Introducción de Ruido Variable**: Antes de cada época de entrenamiento, se aplicó ruido a los datos de entrada y a la salida del generador. Para las dos primeras pruebas la cantidad de ruido no varia con el tiempo. Para la ultima prueba en cambio la cantidad de ruido varia entre épocas.
2) **Función de Pérdida**: Se utilizó un promedio entre la función de pérdida de entropía cruzada binaria y la función de entropía cursada. La primera para distinguir si la imagen venia del generador o del dataset de entrenamiento. Y la segunda para ver si el label de categorización que se le asigno es correcto. Estas pérdidas se calculó en función de las salidas del modelo con la introducción del ruido.
3) **Métricas de Rendimiento**: Durante el entrenamiento, se registraron como métricas de rendimiento como la pérdida del discriminador, la del generador y la precisión de el discriminador en la tarea.

### C. Visualización de los Resultados

En la figura 2 mostramos la media de las métricas en cada época del entrenamiento de todos los generadores.

Mientras tanto en las figuras 3, 4 y 5 vemos las mismas métricas para cada red individual pero agregado la varianza que hubo en los valores de cada epoca sombreado

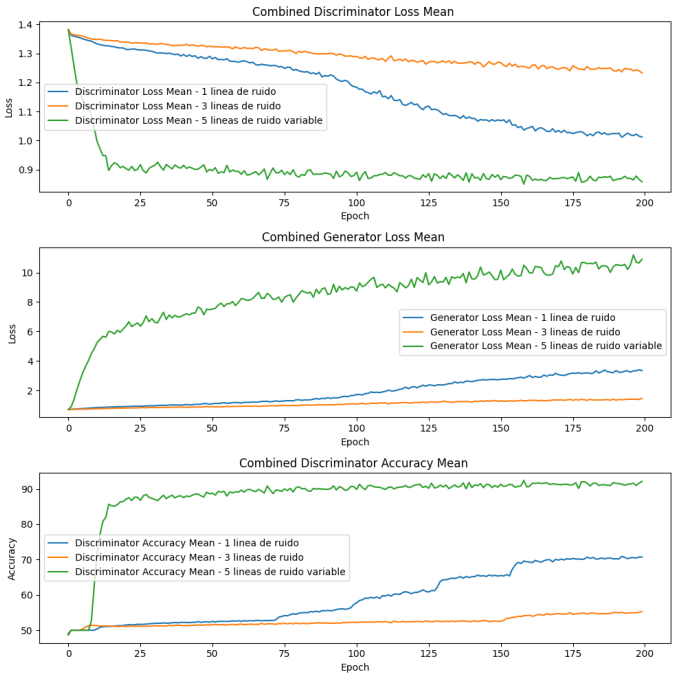


Fig. 2. Resultados del entrenamiento de las 3 SGAN

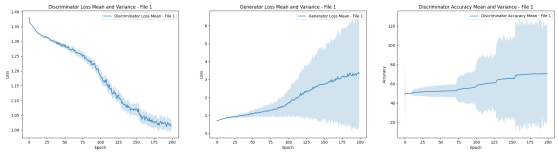

Fig. 3. Resultados del entrenamiento con 1 Línea de Ruido

### D. Analisis del entrenamiento

*1) Perdida del discriminador:* La perdida del discriminador en todas las pruebas disminuyo a lo largo que más paso el tiempo. El de bajo ruido y el de medio ruido constante tienen

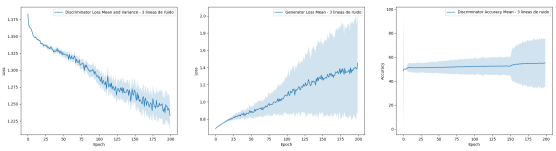

Fig. 4. Resultados del entrenamiento con 3 Líneas de Ruido

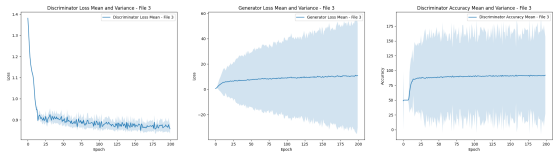

Fig. 5. Resultados del entrenamiento con 5 Líneas de Ruido Variables en el Tiempo

la misma forma mientras que el de ruido variante baja mucho más rápido y luego tiene una pendiente mucho menor según avanza el entrenamiento.

La diferencia entre el de ruido bajo y medio es que el de ruido medio termina con mayor perdida además de tener varianza mas elevada.

Otra cosa notable es la baja varianza del de ruido variante siendo parecido al de bajo ruido.

*2) Perdida del generador:* Comienzan todas iguales pero para la que tiene ruido variable esta sube vertiginozamente, mientras que en la de ruido moderado y constante esta se mantiene constante. Por ultimo en la de ruido bajo esta se mantienen constante durante todo el entrenamiento.

Curiosamente en la varianza (dentro de una misma época) la de bajo ruido tiene mayor varianza que la de ruido moderado y constante. Sin embargo claramente la que más varianza tiene es la de ruido variable.

*3) Precisión del discriminador:* Falta hacer testeos pero preliminarmente podemos decir que todas rápidamente subieron a 50% de precisión luego la de ruido variable creció muy rápido alrededor de la época 10 hasta conseguir una precisión de 90%

La de ruido bajo tardo más en comenzar a diferir muy ligeramente del 50% a partir de la época aprox 13 luego se ven saltos o escalones bastante marcados en al que el discriminador sube su precisión hasta llegar a un alrededor del 70%

Por ultimo la de ruido moderado tiene una precisión baja durante todo el entrenamiento el cual comenzó a incrementarse apreciablemente a partir ya de aproximadamente la época 150 para terminar con aproximadamente una precisión del 55%

### *E. Ejemplos Generados por Diferentes Generadores + Ruido*

En esta sección, en las figuras 6, 7 y 8 hay ejemplos de señales generadas por los diferentes generadores, las cuales después fueron pasadas por una función la cual le agrego el ruido determinado antes de pasárselo al discriminador

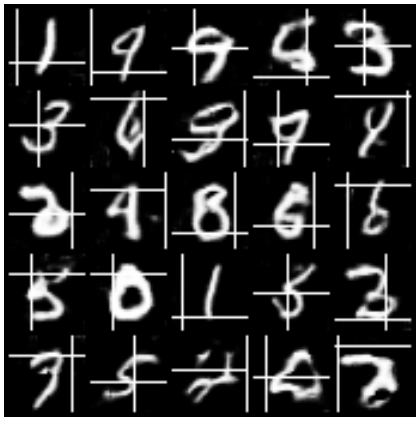

Fig. 6. Generador con 1 linea de ruido

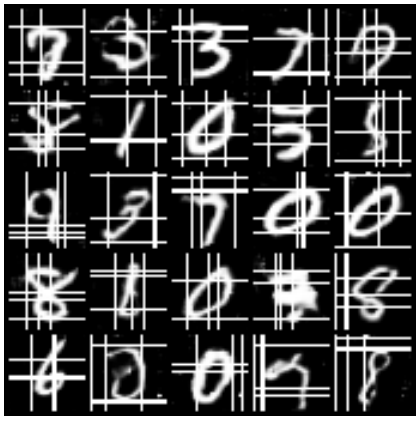

Fig. 7. Generador con 3 lineas de ruido

### *F. Ejemplos del generador previo al ruido*

En Las Figuras 9, 10, y 11 esta sección se presentan ejemplos generados por el modelo antes de la introducción del ruido para comparar los resultados a los que llega el generador.

Es interesante como para los primeros dos generadores cuyo ruido es contante el generador aprendió a generar imágenes sin el ruido a pesar de que al discriminador nunca le llegara sin el.

Otra cosa notable es el echo de que cuando el ruido es variable, el generador puede llegar a generar imágenes con ruido lo cual es indeseado.

### *G. Comparación de la Precisión del Discriminador*

En esta sección, comparamos la precisión de diferentes modelos de discriminadores cuando se prueban con datos conocidos. Específicamente, evaluamos cuán bien los discriminadores pueden clasificar correctamente señales de entrada sin ruido. (tabla I)

Es interesante notar que el generador con ruido variable si bien el generador termino generado imágenes con ruido resulto con un discriminador más robusto a pesar de tener en general ruido menor que el de 1 linea.

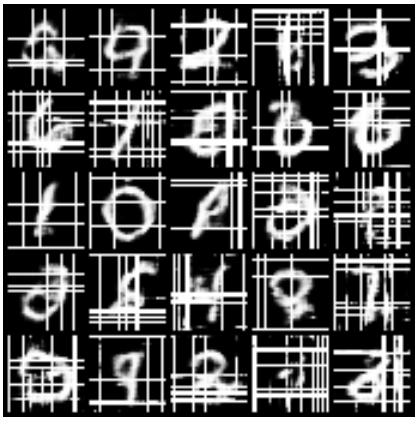

Fig. 8. Generador con 5 lineas de ruido variable

Fig. 9. Generador - Prueba con 1 Línea de Ruido

| Modelo del Discriminador | Precisión en Datos Conocidos |
|---|---|
| Generador 1 | 50.46% |
| Generador 2 | 21.00% |
| Generador 3 | 92.29% |

TABLE I
COMPARACIÓN DE LA PRECISIÓN DE LOS DISCRIMINADORES EN DATOS CONOCIDOS.

Una posibilidad es que el discriminador puede separar el ruido de manera más eficiente, facilitando así la clasificación de la imagen original.

Además, dado que en este caso el generador introduce un ruido similar en su salida, a lo cual se le suma aun más ruido esto podría hacer más fácil identificar las imágenes generadas, por ejemplo con una imagen con cantidad excesiva de lineas. Sin embargo, este último factor por sí solo no parece ser suficiente para explicar la rápida mejora del discriminador, ya que solo contribuiría a alcanzar rápidamente un 50% de precisión (al encontrar los ejemplos falsos antes que los etiquetados).

## V. Conclusion y Trabajos Futuros

Este trabajo se ha centrado en explorar el potencial del Aprendizaje Semi-Supervisado con Redes Generativas Adversariales (SGAN). La investigación, basada en el artículo "Semi-Supervised Learning with Generative Adversarial Networks" de Augustus Odena, ha abordado la eficiencia de los clasificadores en entornos de aprendizaje semi-supervisado, donde al convertir el discriminador en un clasificador (D/C),

Fig. 10. Generador - Prueba con 3 Líneas de Ruido

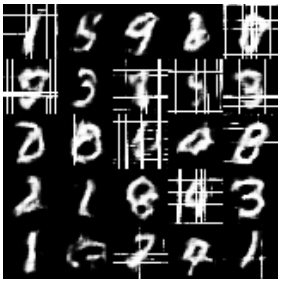

Fig. 11. Generador - Prueba con 5 Líneas de Ruido Variables en el Tiempo

la SGAN permite un aprendizaje conjunto más eficiente entre el generador y el clasificador.

Además, los experimentos realizados sirvieron evaluar el impacto del ruido en los datos de entrada y salida del generador han revelado tendencias interesantes en cuanto a la respuesta del modelo a diferentes niveles y tipos de ruido. Los resultados muestran una variación significativa en la pérdida del discriminador y el generador, así como en la precisión del discriminador, en relación con la introducción de distintos niveles y variaciones de ruido en los datos. Este análisis detallado ha proporcionado información valiosa sobre cómo la variación del ruido puede influir en el rendimiento del modelo y en la calidad de los datos generados.

Finalmente, se planteó una posible teoría que explica por qué el discriminador mostró un rendimiento superior en el caso de ruido variable en comparación con otros escenarios.

Para trabajos futuros, se puede explorar en mayor profundidad el comportamiento y la respuesta del modelo ante distintos tipos de ruido y su impacto en la generación de datos y la clasificación.

También se puede considerar la aplicación de SGAN en conjuntos de datos más complejos y en escenarios del mundo real para evaluar su rendimiento en situaciones más desafiantes.

Otra opción seria utilizar el discriminador con su clasificador para poder obtener salidas particulares del generador.

## VI. Codigo

En el siguiente repositorio estan todos los codigos utilizados para entrenar y para visualizar esta monografia: https://github.com/igna0797/PyTorch-GAN-sgan-modification.git

Este es el codigo utilizado para entrenar las redes:

```
import os
import numpy as np
import math
import pandas as pd  # Import Pandas for CSV handling
import pickle

import torchvision.transforms as transforms
from torchvision.utils import save_image

from torch.utils.data import DataLoader
from torchvision import datasets
from torch.autograd import Variable

import torch.nn as nn
import torch.nn.functional as F
import torch

from utils import parseArguments , get_directory , get_opt_path , add_lines

os.makedirs("images", exist_ok=True)
cuda = True if torch.cuda.is_available() else False
print(f"Grapphics card accelertation: {cuda}")

if __name__ == "__main__":
    opt = parseArguments()
    # Get the directory where the script is located
    directory = get_directory(__file__, opt.max_lines , opt.random_amount_lines)
    optionsPath = os.path.join(directory,"opt.pkl")
    #Save options
    os.makedirs(os.path.dirname(optionsPath), exist_ok=True)  # Create the directory if it doesn't exist
    with open(optionsPath,"wb") as f:
       pickle.dump(opt,f)
else:
    #Load options
    #directory = get_directory(__file__,3, False)
    CallerOptions = parseArguments()
    optionsPath = get_opt_path(__file__, weights_path= CallerOptions.weights_path)
    try:
        with open(optionsPath, "rb") as f:
            opt = pickle.load(f)
    except (FileNotFoundError, IOError, pickle.UnpicklingError) as e:
        # Handle the exception by printing an error message or providing default values
        print(f"An error occurred: {e}")
        print(f"optionsPath: {optionsPath}")
        raise
def weights_init_normal(m):
    classname = m.__class__.__name__
    if classname.find("Conv") != -1:
        torch.nn.init.normal_(m.weight.data, 0.0, 0.02)
    elif classname.find("BatchNorm") != -1:
        torch.nn.init.normal_(m.weight.data, 1.0, 0.02)
        torch.nn.init.constant_(m.bias.data, 0.0)


class Generator(nn.Module):
    def __init__(self):
        super(Generator, self).__init__()

        self.label_emb = nn.Embedding(opt.num_classes, opt.latent_dim)

        self.init_size = opt.img_size // 4  # Initial size before upsampling
        self.l1 = nn.Sequential(nn.Linear(opt.latent_dim, 128 * self.init_size ** 2))

        self.conv_blocks = nn.Sequential(
            nn.BatchNorm2d(128),
            nn.Upsample(scale_factor=2),
            nn.Conv2d(128, 128, 3, stride=1, padding=1),
            nn.BatchNorm2d(128, 0.8),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Upsample(scale_factor=2),
            nn.Conv2d(128, 64, 3, stride=1, padding=1),
            nn.BatchNorm2d(64, 0.8),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Conv2d(64, opt.channels, 3, stride=1, padding=1),
            nn.Tanh(),
        )

    def forward(self, noise):
        out = self.l1(noise)
        out = out.view(out.shape[0], 128, self.init_size, self.init_size)
        img = self.conv_blocks(out)
        return img


class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()

        def discriminator_block(in_filters, out_filters, bn=True):
            """Returns layers of each discriminator block"""
            block = [nn.Conv2d(in_filters, out_filters, 3, 2, 1), nn.LeakyReLU(0.2, inplace=True), nn.Dropout2d(0.25)]
            if bn:
                block.append(nn.BatchNorm2d(out_filters, 0.8))
            return block

        self.conv_blocks = nn.Sequential(
            *discriminator_block(opt.channels, 16, bn=False),
            *discriminator_block(16, 32),
            *discriminator_block(32, 64),
            *discriminator_block(64, 128),
        )
```

```
        # The height and width of downsampled image
        ds_size = opt.img_size // 2 ** 4

        # Output layers
        self.adv_layer = nn.Sequential(nn.Linear(128 * ds_size ** 2, 1), nn.Sigmoid())
        self.aux_layer = nn.Sequential(nn.Linear(128 * ds_size ** 2, opt.num_classes + 1), nn.Softmax())

    def forward(self, img):
        out = self.conv_blocks(img)
        out = out.view(out.shape[0], -1)
        validity = self.adv_layer(out)
        label = self.aux_layer(out)

        return validity, label

if __name__ == "__main__":
  print("Los datos estan guardados en:" + directory)
  os.makedirs(directory, exist_ok=True)  # Create the directory if it doesn't exist
  # Loss functions
  adversarial_loss = torch.nn.BCELoss()
  auxiliary_loss = torch.nn.CrossEntropyLoss()

  # Initialize generator and discriminator
  generator = Generator()
  discriminator = Discriminator()

  if cuda:
      generator.cuda()
      discriminator.cuda()
      adversarial_loss.cuda()
      auxiliary_loss.cuda()

  # Configure data loader
  os.makedirs("../../data/mnist", exist_ok=True)
  dataloader = torch.utils.data.DataLoader(
      datasets.MNIST(
          "../../data/mnist",
          train=True,
          download=True,
          transform=transforms.Compose(
              [transforms.Resize(opt.img_size), transforms.ToTensor(), transforms.Lambda(lambda x: add_lines(x, opt.max_lines, opt.random_amount_lines)) , transforms.Normalize([0.5], [0.5] )]
          ),
      ),
      batch_size=opt.batch_size,
      shuffle=True,
  )

  # Optimizers
  optimizer_G = torch.optim.Adam(generator.parameters(), lr=opt.lr, betas=(opt.b1, opt.b2))
  optimizer_D = torch.optim.Adam(discriminator.parameters(), lr=opt.lr, betas=(opt.b1, opt.b2))

  FloatTensor = torch.cuda.FloatTensor if cuda else torch.FloatTensor
  LongTensor = torch.cuda.LongTensor if cuda else torch.LongTensor

  """
  # Initialize weights
  if os.path.exists("/content/PyTorch-GAN-sgan-modification/implementations/sgan/generator_weights.pth") and os.path.exists("/content/PyTorch-GAN-sgan-modification/implementations/sgan/discriminator_weights.pth"):
    generator.load_state_dict(torch.load("/content/PyTorch-GAN-sgan-modification/implementations/sgan/generator_weights.pth"))
    discriminator.load_state_dict(torch.load("/content/PyTorch-GAN-sgan-modification/implementations/sgan/discriminator_weights.pth"))
    print("Loaded pre-trained weights.")
  else:
    generator.apply(weights_init_normal)
    discriminator.apply(weights_init_normal)

  # Initialize weights
  if os.path.exists( directory + "/generator_weights.pth") and os.path.exists(directory +"/discriminator_weights.pth"):
      generator.load_state_dict(torch.load(directory + "/generator_weights.pth"))
      discriminator.load_state_dict(torch.load(directory +"/discriminator_weights.pth"))
      print("Loaded pre-trained weights.")
  else:
      generator.apply(weights_init_normal)
      discriminator.apply(weights_init_normal)
      print("Creating new weights.")
  """
# Create variables to track batch and epoch
  current_epoch = 0
  current_batch = 0

  # Check if a checkpoint file exists
  checkpoint_directory = os.path.join(directory , "checkpoint.pth")
  if os.path.exists(checkpoint_directory):
      checkpoint = torch.load(checkpoint_directory)
      current_epoch = checkpoint["epoch"]
      current_batch = checkpoint["batch"]
      generator.load_state_dict(checkpoint["generator_state_dict"])
      discriminator.load_state_dict(checkpoint["discriminator_state_dict"])
      optimizer_G.load_state_dict(checkpoint["optimizer_G_state_dict"])
      optimizer_D.load_state_dict(checkpoint["optimizer_D_state_dict"])
      print(f"Loaded checkpoint from epoch {current_epoch}, batch {current_batch
```

```
        }")


    # Visualize a couple of real images from the dataset
    sample_data = next(iter(dataloader))
    sample_images, _ = sample_data

    # Save the visualization images in the same folder as generated images
    save_image(sample_images[:20], directory + "/dataset_visualization.png", nrow=5, normalize=True)

    # ----------
    #  Training
    # ----------
#gen_loss_log = open(directory + "/loss_log.txt", "w")

    # Initialize an empty list to collect loss data
    loss_data = []

    # Define the directory where you want to save images
    image_dir = directory + "/training/images"
    # Create the directory if it doesn't exist
    os.makedirs(image_dir, exist_ok=True)
    for epoch in range(current_epoch, opt.n_epochs):
        for i, (imgs, labels) in enumerate(dataloader):
            current_batch = i
            current_epoch = epoch
            batch_size = imgs.shape[0]

            # Adversarial ground truths
            valid = Variable(FloatTensor(batch_size, 1).fill_(1.0), requires_grad=False)
            fake = Variable(FloatTensor(batch_size, 1).fill_(0.0), requires_grad=False)
            fake_aux_gt = Variable(LongTensor(batch_size).fill_(opt.num_classes), requires_grad=False)

            # Configure input
            real_imgs = Variable(imgs.type(FloatTensor))
            labels = Variable(labels.type(LongTensor))

            # -----------------
            #  Train Generator
            # -----------------

            optimizer_G.zero_grad()

            # Sample noise and labels as generator input
            z = Variable(FloatTensor(np.random.normal(0, 1, (batch_size, opt.latent_dim))))

            # Generate a batch of images
            gen_imgs = generator(z)
            gen_imgs = add_lines(gen_imgs, opt.max_lines , opt.random_amount_lines )
            # Loss measures generator's ability to fool the discriminator
            validity, _ = discriminator(gen_imgs)
            g_loss = adversarial_loss(validity, valid)

            g_loss.backward()
            optimizer_G.step()

            # ---------------------
            #  Train Discriminator
            # ---------------------

            optimizer_D.zero_grad()

            # Loss for real images
            real_pred, real_aux = discriminator(real_imgs)
            d_real_loss = (adversarial_loss(real_pred, valid) + auxiliary_loss(real_aux, labels)) / 2

            # Loss for fake images
            fake_pred, fake_aux = discriminator(gen_imgs.detach())
            d_fake_loss = (adversarial_loss(fake_pred, fake) + auxiliary_loss(fake_aux, fake_aux_gt)) / 2

            # Total discriminator loss
            d_loss = (d_real_loss + d_fake_loss) / 2

            # Calculate discriminator accuracy
            pred = np.concatenate([real_aux.data.cpu().numpy(), fake_aux.data.cpu().numpy()], axis=0)
            gt = np.concatenate([labels.data.cpu().numpy(), fake_aux_gt.data.cpu().numpy()], axis=0)
            d_acc = np.mean(np.argmax(pred, axis=1) == gt)

            d_loss.backward()
            optimizer_D.step()

            print(
                "[Epoch %d/%d] [Batch %d/%d] [D loss: %f, acc: %d%%] [G loss: %f]"
                % (epoch, opt.n_epochs, i, len(dataloader), d_loss.item(), 100 * d_acc, g_loss.item())
            )

            batches_done = epoch * len(dataloader) + i
            if batches_done % opt.sample_interval == 0 and
```

```
                batches_done !=0:
279             save_image(gen_imgs.data[:25],
                    image_dir + "/%d.png" %
                    batches_done, nrow=5,
                    normalize=True)
280
281             torch.save({
282               "epoch": current_epoch,
283               "batch": current_batch,
284               "generator_state_dict":
                      generator.state_dict(),
285               "discriminator_state_dict":
                      discriminator.state_dict()
                      ,
286               "optimizer_G_state_dict":
                      optimizer_G.state_dict(),
287               "optimizer_D_state_dict":
                      optimizer_D.state_dict(),
288           }, checkpoint_directory)
289             # Create a DataFrame from the
                    collected data
290             if os.path.exists(directory + "/
                    training/loss_data.csv"):
291               original_loss_df = pd.read_csv
                      (directory + "/training/
                      loss_data.csv")
292               new_loss_df = pd.DataFrame(
                      loss_data)
293               # Create a list of DataFrames
                      to concatenate
294               dataframes_to_concat = [
                      original_loss_df,
                      new_loss_df]
295               # Use pd.concat to concatenate
                       the DataFrames
296               loss_df = pd.concat(
                      dataframes_to_concat)
297             else:
298               print("no enconto")
299               loss_df = pd.DataFrame(
                      loss_data)
300             # Save the loss data to a CSV
                    file
301             loss_df.to_csv(directory + "/
                    training/loss_data.csv",
                    index=False)
302             loss_data=[]
303         loss_data.append({
304             'Epoch': epoch,
305             'Batch': i,
306             'Discriminator Loss': d_loss.
                    item(),
307             'Discriminator Accuarcy' : 100 *
                     d_acc,
308             'Generator Loss': g_loss.item(),
309         })
310
311 #loss_log.close()
312
313   # Save generator weights
314   torch.save(generator.state_dict(), directory
          + "/generator_weights.pth")
315   # Save discriminator weights
316   torch.save(discriminator.state_dict(),
          directory +"/discriminator_weights.pth")
```

## References


Augustus Odena. Semi-supervised learning with generative adversarial networks, 2016. URL https://arxiv.org/abs/1606.01583.