Deep Learning Attacks - Deepfake Generation

1. Introduction

Deep learning is a type of machine learning that utilises multilayered neural networks to retain information and learn from datasets *(Google Cloud, 2025)*. These neural networks are artificial, mimicking the way a human brain processes data to perform automated tasks *(Holdsworth, 2024)*.

A controversial application of deep learning is deepfake generation, which were originally created for the purpose of entertainment *(Western Australian Government, 2024),* deepfakes have evolved into a threat. This is due to authentic content being manipulated through the creation of artificial images, videos and audios for the purpose of stealing identities, spreading fake news and scamming unknowing victims *(CSIRO, 2024).* As of 2024, deepfake attacks have increased by 43%, the majority of the causes being corporations being scammed due to financial fraud and fake identity impersonations (*Western Australian Government, 2024)*. Deepfakes pose a risk to misinformation that may manipulate public opinion and may result in reputational damage for public figures (*Western Australian Government, 2024)*, as the edited and manipulated content exploits the trust people place on auditory and visual media. Thus, deepfake generation is a threat towards corporations, politics and individual safety.

This report aims to investigate and analyse various deepfake generation methods, centralising on facial manipulation. It will be through assessing each one's impact towards corporations, politics and individual safety, highlighting the importance of correctly differentiating a deepfake image from an authentic one. The chosen deepfake generation methods are the following:

- Autoencoder-Decoder Networks
- Generative Adversarial Networks (GANs)
- Diffusion Models

Each deepfake generation technique will be tested using the "FashionMNIST" dataset, which contains images of black and white fashion products. It is a replacement for the "CelebA" dataset that was planned for use, however it was unable to be implemented due to the dataset being too large for Google Colab to computationally handle.

The results collected will define how each deepfake generation technique performs, while highlighting the easy accessibility of deepfake generation through the utilisation of deep learning libraries that provide pre-built code. Thus, the public's easy access to deepfake generation using deep learning may result in the progressive increase of deepfake attacks.

2. Methods

2.1 Autoencoder-Decoder Networks

2.1.1 Method Description

Autoencoders in deep learning are easily accessible for individuals who lack prior experience with deepfake generation due to its straightforward application. Its architecture consists of an encoder, where complicated data is compressed into a simpler format *(v7labs, 2021),* and a decoder, where the simplified data is reconstructed based on the prompt the algorithm was given *(Prasad, 2024)*.

In the context of deepfake generation, two autoencoders with different decoders *(Patel, 2025)* are trained separately using specified target faces. As demonstrated in figure 1, the image of Target Face A is compressed by the first autoencoder's encoder, while the image of Target Face B is utilised to reconstruct the image with the second autoencoder's decoder.

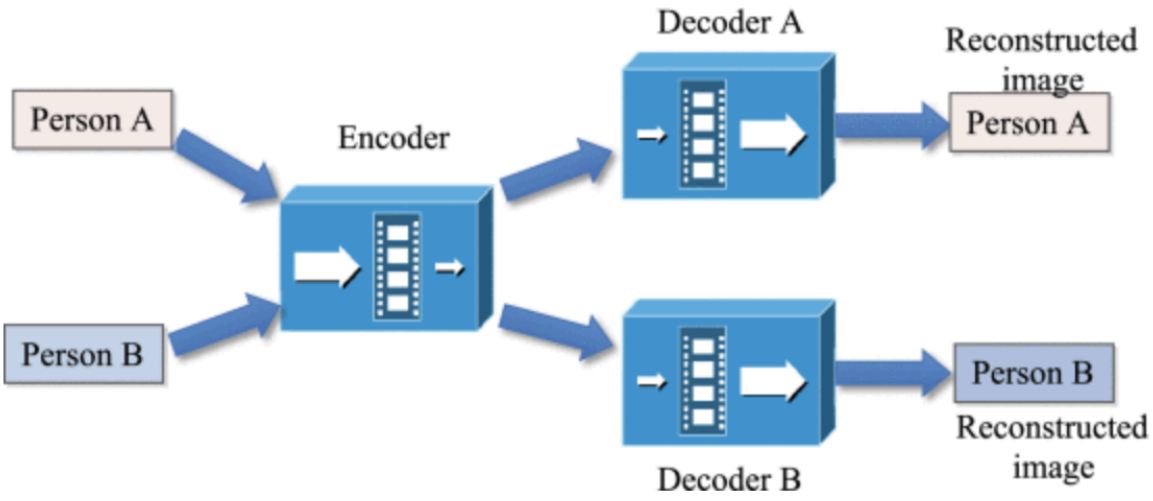

Figure 1 - Deepfake generation using autoencoder

In further detail, the encoder abstracts the input's features through discarding unessential information in order to capture relevant features such as facial expressions, skin texture, skin tone and eyes *(Patel, 2024)*. This enables the decoder to utilise these features for deepfake generation by reconstructing and manipulating them onto a target face *(Prasad, 2024)*, resulting in a seamless and realistic appearance which mimics the authentic source material *(Patel, 2025).*

As such, an autoencoder is useful for generating deepfakes through reconstructing the extracted features from the source material or by face-swapping two target faces using two decoders. The easy accessibility of autoencoders may lead attackers to misuse them for the purpose of creating inappropriate content and spreading misinformation through impersonation *(Alanazi, 2024)*.
Consequently, this poses a threat to digital safety and risks the reputations of corporations and public figures.

2.1.2 Key Components

A simple autoencoder was made using python and Google Colab to simulate the face swapping process. using the “FashionMNIST” dataset.

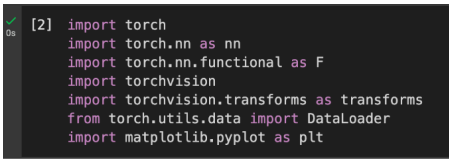

Figure 2: Imports

Figure 3: Loading “FashionMNIST” dataset

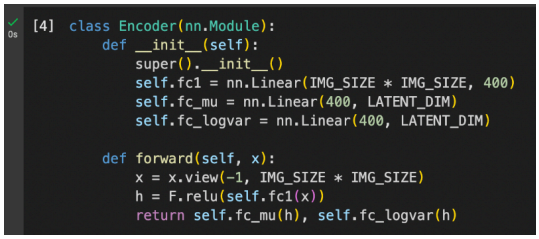

Figure 4: Encoder

Figure 5: Sampling latent vector - stabilizes model

```
[6]  class Decoder(nn.Module):
         def __init__(self):
             super().__init__()
             self.fc = nn.Sequential(
                 nn.Linear(LATENT_DIM, 400),
                 nn.ReLU(),
                 nn.Linear(400, IMG_SIZE * IMG_SIZE),
                 nn.Sigmoid()
             )
```

Figure 6: Decoder

```
[12] class DeepfakeVAE(nn.Module):
         def __init__(self):
             super().__init__()
             self.encoder = Encoder()
             self.decoderA = Decoder()
             self.decoderB = Decoder()

         def forward(self, x):
             mu, logvar = self.encoder(x)
             z = reparameterize(mu, logvar)
             reconA = self.decoderA(z)
             reconB = self.decoderB(z)
             return reconA, reconB, mu, logvar
```

Figure 7: Initialising one encoder (for input) and two decoders (output)

```
[8]  def vae_loss(recon_x, x, mu, logvar):
         BCE = F.binary_cross_entropy(recon_x, x, reduction='sum')
         KLD = -0.5 * torch.sum(1 + logvar - mu.pow(2) - logvar.exp())
         return BCE + KLD
```

Figure 8: VAE loss function - measuring difference between reconstructed image and original input

```
[9]  model = DeepfakeVAE().to(device)
     optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)
```

Figure 9: Training model

```
[10] for epoch in range(EPOCHS):
         model.train()
         total_loss = 0
         for x, _ in train_loader:
             x = x.to(device)
             reconA, reconB, mu, logvar = model(x)
             lossA = vae_loss(reconA, x, mu, logvar)
             lossB = vae_loss(reconB, x, mu, logvar)
             loss = lossA + lossB

             optimizer.zero_grad()
             loss.backward()
             optimizer.step()
             total_loss += loss.item()

         print(f"Epoch {epoch+1}/{EPOCHS}, Loss: {total_loss / len(train_loader.dataset):.4f}")
```

```
Epoch 1/10, Loss: 573.4757
Epoch 2/10, Loss: 512.0283
Epoch 3/10, Loss: 500.6793
Epoch 4/10, Loss: 495.2816
Epoch 5/10, Loss: 492.2400
Epoch 6/10, Loss: 490.0287
Epoch 7/10, Loss: 488.5064
Epoch 8/10, Loss: 487.1662
Epoch 9/10, Loss: 486.2007
Epoch 10/10, Loss: 485.3560
```

Figure 10: Training loop

```
model.eval()
with torch.no_grad():
    x, _ = next(iter(train_loader))
    x = x[:8].to(device)
    _, fake_images, _, _ = model(x)

    fig, axs = plt.subplots(2, 8, figsize=(12, 4))
    for i in range(8):
        axs[0, i].imshow(x[i].cpu().squeeze(), cmap="gray")
        axs[1, i].imshow(fake_images[i].cpu().squeeze(), cmap="gray")
        axs[0, i].axis('off')
        axs[1, i].axis('off')
    axs[0, 0].set_title("Original")
    axs[1, 0].set_title("Fake Output")
    plt.tight_layout()
    plt.show()
```

Figure 11: Comparing authentic image with generated image

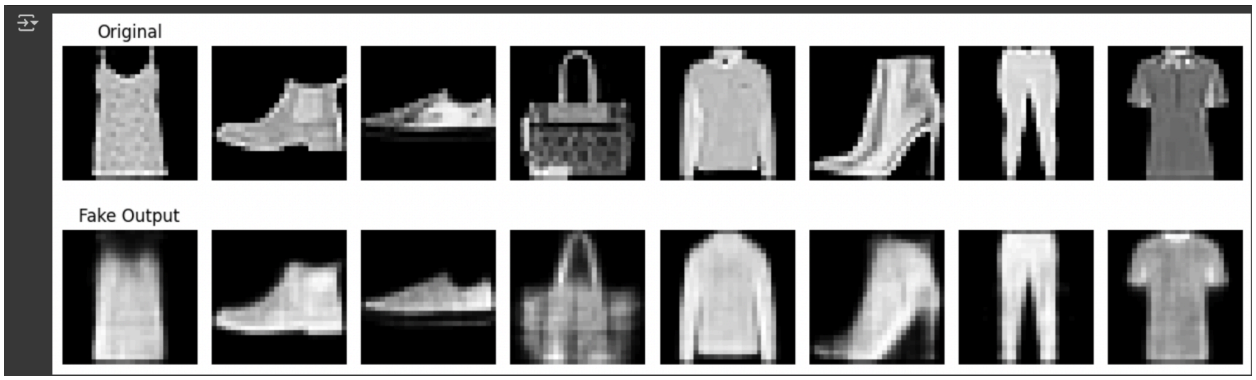

Figure 12: Results

2.2 Generative Adversarial Networks (GAN)

2.2.1 Method Description

Generative adversarial networks are often combined with autoencoders when generating deepfakes to enhance its output quality *(Prasad, 2024)*. A GAN's architecture consists of a generator, which is a neural network that generates images, and a discriminator, which is a competing network that evaluates each image's authenticity *(Prasad, 2024).* As these two neural networks compete against each other throughout training *(AWS, 2025)*, the quality of the image output enhances over time. This is due to the GAN aiming for the discriminator to be unable to differentiate between the authentic output and the ones created by the generator *(Prasad, 2024).*

In the context of deepfakes, the GAN's generator utilises a latent space to create images that closely resemble the inputs from the provided dataset *(Zivkovic, 2022)*. As shown in figure 2, the real images provided by a large dataset and the generated ones are processed by the

discriminator throughout the adversarial training process *(Alanazi, 2024)*, ensuring that seamless and realistic images are being produced over time after multiple cycles.

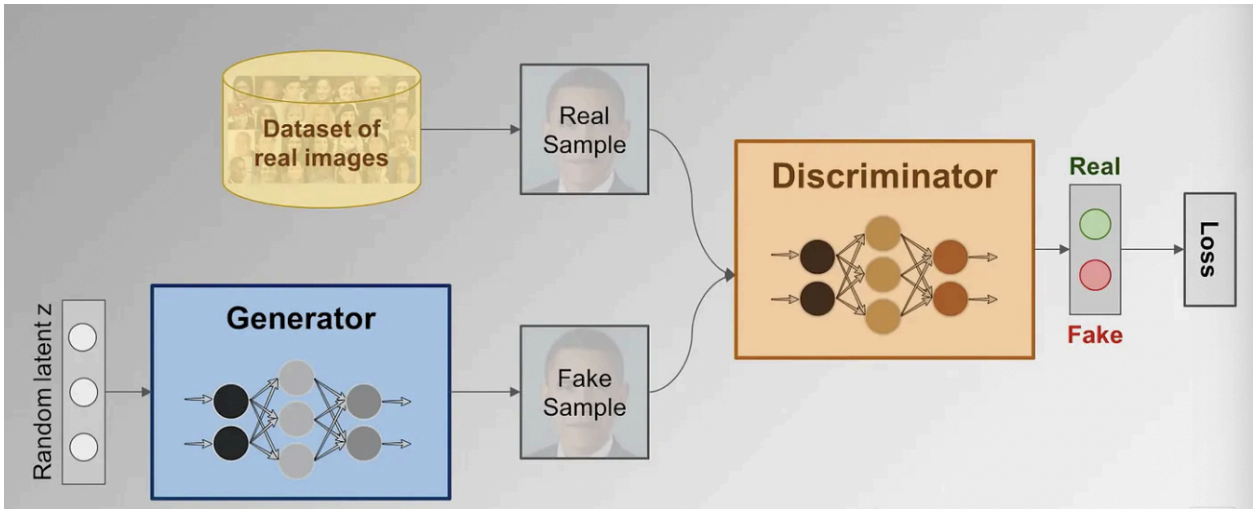

Figure 13: Deepfake generation using Generative Adversarial Networks

As such, GANs may be a threat to digital safety due to their ability to generate realistic images through adversarial training. This can result in attackers exploiting GANs for the purpose of impersonating victims by generating IDs or passport photos, risking the security of corporations and the reputations of public figures.

2.2.2 Key Components

```
[1]  from __future__ import print_function
     import os
     import random
     import torch
     import torch.nn as nn
     import torch.nn.parallel
     import torch.backends.cudnn as cudnn
     import torch.optim as optim
     import torch.utils.data
     import torchvision.datasets as dset
     import torchvision.transforms as transforms
     import torchvision.utils as vutils
     import matplotlib.pyplot as plt

     import numpy
```

Figure 14: Imports

```
[12] dataset = dset.FashionMNIST(root='./data', download=True,
                                 transform=transforms.Compose([
                                 transforms.ToTensor(),
                                 transforms.Normalize((0.5,), (0.5,)),
                                ]))

     dataloader = torch.utils.data.DataLoader(dataset, batch_size=64,
                                              shuffle=True, num_workers=2)
```

Figure 15: Loading "FashionMNIST" dataset

```
[6]  figure = plt.figure(figsize=(8, 8))
     cols, rows = 3, 3
     for i in range(1, cols * rows + 1):
         sample_idx = torch.randint(len(dataset), size=(1,)).item()
         img, label = dataset[sample_idx]
         figure.add_subplot(rows, cols, i)
         plt.axis("off")
         plt.imshow(img.squeeze(), cmap="gray")
     plt.show()
```

Figure 16: Taking 9 images from "FashionMNIST" dataset

Figure 17: 9 images shown

```
[17] def weights_init(m):
         classname = m.__class__.__name__
         if classname.find('Conv') != -1:
             m.weight.data.normal_(0.0, 0.02)
         elif classname.find('BatchNorm') != -1:
             m.weight.data.normal_(1.0, 0.02)
             m.bias.data.fill_(0)
```

Figure 18: Initialising weights - helps model to train more efficiently

```
[18] class Generator(nn.Module):
         def __init__(self, ngpu, nc=1, nz=100, ngf=64):
             super(Generator, self).__init__()
             self.ngpu = ngpu
             self.main = nn.Sequential(
                 nn.ConvTranspose2d(     nz, ngf * 8, 4, 1, 0, bias=False),
                 nn.BatchNorm2d(ngf * 8),
                 nn.ReLU(True),
                 nn.ConvTranspose2d(ngf * 8, ngf * 4, 4, 2, 1, bias=False),
                 nn.BatchNorm2d(ngf * 4),
                 nn.ReLU(True),
                 nn.ConvTranspose2d(ngf * 4, ngf * 2, 4, 2, 1, bias=False),
                 nn.BatchNorm2d(ngf * 2),
                 nn.ReLU(True),
                 nn.ConvTranspose2d(ngf * 2,     ngf, 4, 2, 1, bias=False),
                 nn.BatchNorm2d(ngf),
                 nn.ReLU(True),
                 nn.ConvTranspose2d(    ngf,      nc, kernel_size=1, stride=1, padding=2, bias=False),
                 nn.Tanh()
             )
```

Figure 19: Generator

```
[20] class Discriminator(nn.Module):
         def __init__(self, ngpu, nc=1, ndf=64):
             super(Discriminator, self).__init__()
             self.ngpu = ngpu
             self.main = nn.Sequential(
                 nn.Conv2d(nc, ndf, 4, 2, 1, bias=False),
                 nn.LeakyReLU(0.2, inplace=True),
                 nn.Conv2d(ndf, ndf * 2, 4, 2, 1, bias=False),
                 nn.BatchNorm2d(ndf * 2),
                 nn.LeakyReLU(0.2, inplace=True),
                 nn.Conv2d(ndf * 2, ndf * 4, 4, 2, 1, bias=False),
                 nn.BatchNorm2d(ndf * 4),
                 nn.LeakyReLU(0.2, inplace=True),
                 nn.Conv2d(ndf * 4, 1, 4, 2, 1, bias=False),
                 nn.Sigmoid()
             )
```

Figure 20: Discriminator

```
[11] for epoch in range(niter):
         for i, data in enumerate(dataloader, 0):
             netD.zero_grad()
             real_cpu = data[0].to(device)
             batch_size = real_cpu.size(0)
             label = torch.full((batch_size,), real_label, device=device)

             output = netD(real_cpu)
             errD_real = criterion(output.float(), label.float())
             errD_real.backward()
             D_x = output.mean().item()

             noise = torch.randn(batch_size, nz, 1, 1, device=device)
             fake = netG(noise)
             label.fill_(fake_label)
             output = netD(fake.detach())
             errD_fake = criterion(output.float(), label.float())
             errD_fake.backward()
             D_G_z1 = output.mean().item()
             errD = errD_real + errD_fake
             optimizerD.step()

             netG.zero_grad()
             label.fill_(real_label)
             output = netD(fake)
             errG = criterion(output.float(), label.float())
             errG.backward()
             D_G_z2 = output.mean().item()
             optimizerG.step()
             print(f'[{epoch:d}/{niter:d}][{i:d}/{len(dataloader):d}] Loss_D: {errD.item():.4f} Loss_G: {errG.item():.4f} D(x): {D_x:.4f} D(G(z)): {D_G_z1:.4f} /
```

Figure 21: Training loop

```
[15] G = Generator(ngpu=1).eval()

     G.load_state_dict(torch.load('weights/netG_epoch_24.pth'))
     if torch.cuda.is_available():
         G = G.cuda()

     batch_size = 1000
     latent_size = 100

     fixed_noise = torch.randn(batch_size, latent_size, 1, 1)
     if torch.cuda.is_available():
         fixed_noise = fixed_noise.cuda()
     fake_images = G(fixed_noise)

     fake_images_np = fake_images.cpu().detach().numpy()
     fake_images_np = fake_images_np.reshape(fake_images_np.shape[0], 28, 28)
```

Figure 22: Generator generating images

```
[10] figure = plt.figure(figsize=(8, 8))
     R, C = 5, 5
     for i in range(1, cols * rows + 1):
         figure.add_subplot(R, C, i)
         plt.axis("off")
         plt.imshow(fake_images_np[i], cmap="gray")
     figure.suptitle('Deepfake image samples', fontsize=16)
     plt.show()
```

Figure 23: Deepfake images

Figure 24: Deepfake images shown

```
[16] figure = plt.figure(figsize=(8, 8))
     cols, rows = 5, 5
     for i in range(1, cols * rows + 1):
         sample_idx = torch.randint(len(dataset), size=(1,)).item()
         img, label = dataset[sample_idx]
         figure.add_subplot(rows, cols, i)
         plt.axis("off")
         plt.imshow(img.squeeze(), cmap="gray")
     figure.suptitle('Real image samples', fontsize=16)
     plt.show()
```

Figure 25: Real images

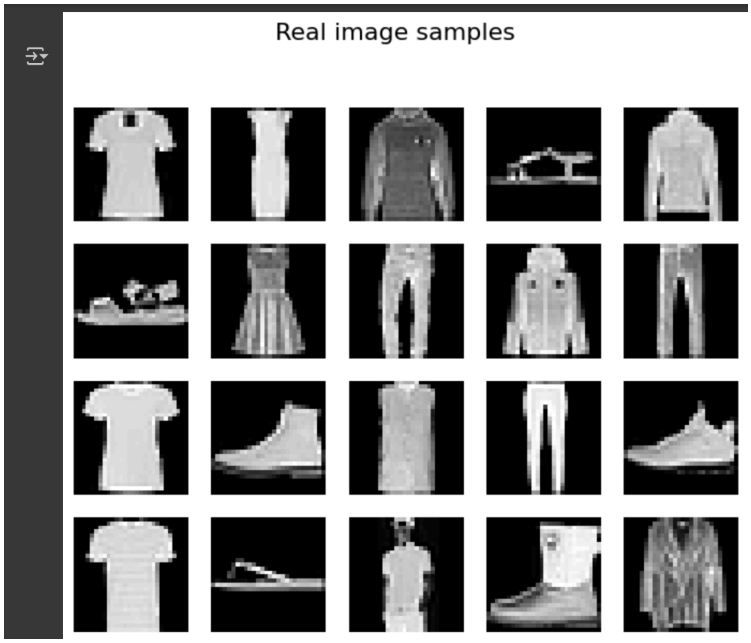

Figure 26: Real images shown

2.3 Diffusion Models

2.3.1 Method Description

Diffusion models generate high-quality images through progressively corrupting the input with noise and reconstructing it by reversing the process *(SuperAnnotate, 2025)*. As demonstrated in figure 3, the model systematically goes through its steps in a linear manner, with a forward and backward approach.

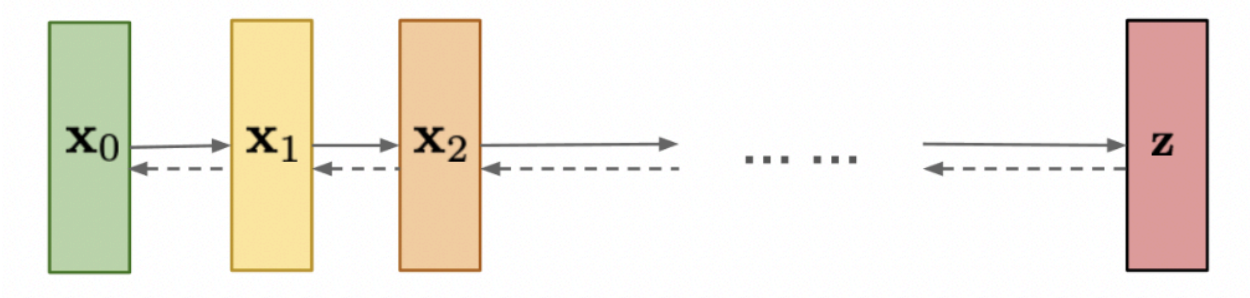

Figure 27: Process of diffusion model steps

This reflects the model's method of image generation, which consists of two processes: The forward diffusion process, where the model samples a Gaussian distribution and adds noise to the input at each step until it reaches a pure noise distribution *(Sapien, 2024)*. In a mathematical sense, this process is recognised as the Markov chain *(Zhong, 2024),* where the noisy state of the output during each step is dependent on its previous state *(De, 2020)*. This is due to the added noise being gradually increased throughout the forward diffusion process (Sapien, 2024), enabling the model to learn as the utilisation of larger noise results in a more balanced and seamless output *(Bergmann, 2024).*

Furthermore, once the model has been fully trained it advances onto the reverse diffusion process, where the model reconstructs and generates new outputs based on the prompt it was given *(SuperAnnotate, 2025).* This phase involves the model estimating the noise distributed on each step to remove the noise and reverse the forward diffusion process *(Sapien, 2024).* This enables the diffusion model to generate new content from the random noise as it has learnt the foundation and special features of the input image *(Sapien, 2024).*

In regards to deepfake generation, diffusion models are utilised for recreating and reconstructing realistic facial features as they generate high-quality images by eliminating any inconsistencies such as smudges and asymmetric features *(Bhattacharyya, 2025)*. They are also able to demonstrate diversity within their outputs through training with large datasets such as "LAION-5B" and "CelebA" *(Bhattacharyya, 2025)*, adding an authentic quality to the generated deepfakes.

As such, open sourced diffusion models such as "Stable Diffusion" are a threat to digital safety due to their ability to generate high quality deepfakes. This makes them useful for impersonation

attacks where it can enable attackers to bypass security systems and scam victims, negatively impacting individual digital safety.

2.3.2 Key Components

```
[1]  import torch
     import torchvision
     import matplotlib.pyplot as plt
     import math
     import torch.nn.functional as F

     from torchvision import transforms
     from torch.utils.data import DataLoader
     import numpy as np
     from torch import nn
     import math
     from torch.optim import Adam
```

Figure 28: Imports

```
[2]  def show_images(dataset, num_samples = 20, cols = 4):
         plt.figure(figsize=(15, 15))
         rows = math.ceil(num_samples / cols)

         for i, (img, _) in enumerate(dataset):
             if i == num_samples:
                 break
             plt.subplot(rows, cols, i + 1)
             plt.imshow(img)
             plt.axis('off')
         plt.show()

     data = torchvision.datasets.FashionMNIST(root = ".", download = True)
     show_images(data)
```

Figure 29: Loading “FashionMNIST” dataset

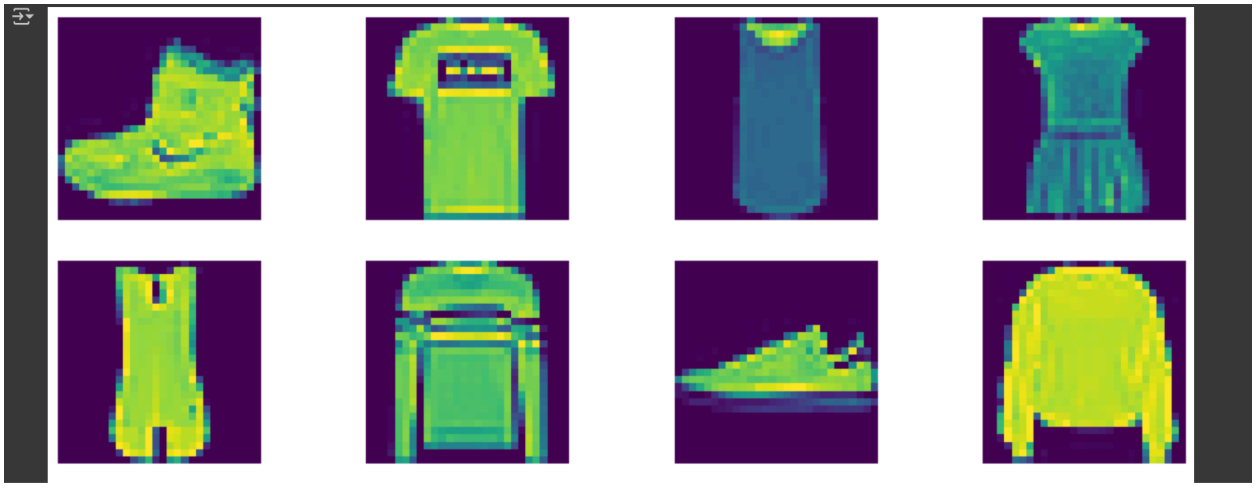

Figure 30: Images from “FashionMNIST” dataset

```python
def linear_beta_schedule(timesteps, start = 0.0001, end = 0.02):
    return torch.linspace(start, end, timesteps)

def get_index_from_list(vals, t, x_shape):
    batch_size = t.shape[0]
    out = vals.gather(-1, t.cpu())
    return out.reshape(batch_size, *((1,) * (len(x_shape) - 1))).to(t.device)
```

Figure 31: Beta schedule for diffusion process - defines how much noise will be increased after each step

```python
def forward_diffusion_sample(x_0, t, device = "cpu"):
    noise = torch.randn_like(x_0)
    sqrt_alphas_cumprod_t = get_index_from_list(sqrt_alphas_cumprod, t, x_0.shape)
    sqrt_one_minus_alphas_cumprod_t = get_index_from_list(
        sqrt_one_minus_alphas_cumprod, t, x_0.shape
    )
    return sqrt_alphas_cumprod_t.to(device) * x_0.to(device) \
    + sqrt_one_minus_alphas_cumprod_t.to(device) * noise.to(device), noise.to(device)

T = 300
betas = linear_beta_schedule(timesteps=T)

alphas = 1. - betas
alphas_cumprod = torch.cumprod(alphas, axis=0)
alphas_cumprod_prev = F.pad(alphas_cumprod[:-1], (1, 0), value=1.0)
sqrt_recip_alphas = torch.sqrt(1.0 / alphas)
sqrt_alphas_cumprod = torch.sqrt(alphas_cumprod)
sqrt_one_minus_alphas_cumprod = torch.sqrt(1. - alphas_cumprod)
posterior_variance = betas * (1. - alphas_cumprod_prev) / (1. - alphas_cumprod)
```

Figure 32: Forward diffusion process

```python
[4] IMG_SIZE = 64
    BATCH_SIZE = 128

    def load_transformed_dataset():
        data_transforms = [
        transforms.Grayscale(num_output_channels=3),
        transforms.Resize((IMG_SIZE, IMG_SIZE)),
        transforms.RandomHorizontalFlip(),
        transforms.ToTensor(),
        transforms.Lambda(lambda t: (t * 2) - 1)
    ]
        data_transform = transforms.Compose(data_transforms)

        train = torchvision.datasets.FashionMNIST(root=".", download=True,
                                            transform=data_transform)

        test = torchvision.datasets.FashionMNIST(root=".", train=False, download=True, transform=data_transform)

        return torch.utils.data.ConcatDataset([train, test])
    def show_tensor_image(image):
        reverse_transforms = transforms.Compose([
            transforms.Lambda(lambda t: (t + 1) / 2),
            transforms.Lambda(lambda t: t.permute(1, 2, 0)),
            transforms.Lambda(lambda t: t * 255.),
            transforms.Lambda(lambda t: t.numpy().astype(np.uint8)),
            transforms.ToPILImage(),
        ])


        if len(image.shape) == 4:
            image = image[0, :, :, :]
        plt.imshow(reverse_transforms(image))

    data = load_transformed_dataset()
    dataloader = DataLoader(data, batch_size=BATCH_SIZE, shuffle=True, drop_last=True)
```

FIgure 33: Resizing "FashionMNIST" images

```
[6]  image = next(iter(dataloader))[0]

     plt.figure(figsize=(15,15))
     plt.axis('off')
     num_images = 10
     stepsize = int(T/num_images)

     for idx in range(0, T, stepsize):
         t = torch.Tensor([idx]).type(torch.int64)
         plt.subplot(1, num_images+1, int(idx/stepsize) + 1)
         img, noise = forward_diffusion_sample(image, t)
         show_tensor_image(img)
```

Figure 34: Diffusion process

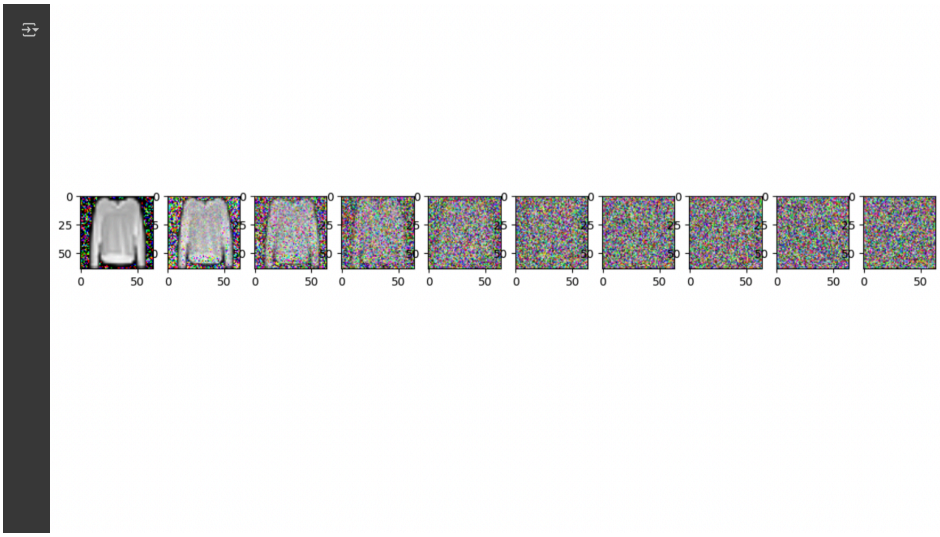

Figure 35: Diffusion process result

```
[7]  class Block(nn.Module):
         def __init__(self, in_ch, out_ch, time_emb_dim, up=False):
             super().__init__()
             self.time_mlp =  nn.Linear(time_emb_dim, out_ch)
             if up:
                 self.conv1 = nn.Conv2d(2*in_ch, out_ch, 3, padding=1)
                 self.transform = nn.ConvTranspose2d(out_ch, out_ch, 4, 2, 1)
             else:
                 self.conv1 = nn.Conv2d(in_ch, out_ch, 3, padding=1)
                 self.transform = nn.Conv2d(out_ch, out_ch, 4, 2, 1)
             self.conv2 = nn.Conv2d(out_ch, out_ch, 3, padding=1)
             self.bnorm1 = nn.BatchNorm2d(out_ch)
             self.bnorm2 = nn.BatchNorm2d(out_ch)
             self.relu  = nn.ReLU()

         def forward(self, x, t, ):
             h = self.bnorm1(self.relu(self.conv1(x)))
             time_emb = self.relu(self.time_mlp(t))
             time_emb = time_emb[(..., ) + (None, ) * 2]
             h = h + time_emb
             h = self.bnorm2(self.relu(self.conv2(h)))
             return self.transform(h)
```

Figure 36: Building denoising network

```
[10] def get_loss(model, x_0, t):
         x_noisy, noise = forward_diffusion_sample(x_0, t, device)
         noise_pred = model(x_noisy, t)
         return F.l1_loss(noise, noise_pred)
```

Figure 37: Loss function - difference between actual and predicted noise added on input during each step (part of training model)

```
[19] @torch.no_grad()
     def sample_timestep(x, t):
         betas_t = get_index_from_list(betas, t, x.shape)
         sqrt_one_minus_alphas_cumprod_t = get_index_from_list(
             sqrt_one_minus_alphas_cumprod, t, x.shape
         )
         sqrt_recip_alphas_t = get_index_from_list(sqrt_recip_alphas, t, x.shape)

         model_mean = sqrt_recip_alphas_t * (
             x - betas_t * model(x, t) / sqrt_one_minus_alphas_cumprod_t
         )
         posterior_variance_t = get_index_from_list(posterior_variance, t, x.shape)

         if t == 0:
             return model_mean
         else:
             noise = torch.randn_like(x)
             return model_mean + torch.sqrt(posterior_variance_t) * noise
```

Figure 38: Reverse diffusion process

```
[ ] device = "cuda" if torch.cuda.is_available() else "cpu"
    model.to(device)
    optimizer = Adam(model.parameters(), lr=0.001)
    epochs = 100 # Try more!

    for epoch in range(epochs):
        for step, batch in enumerate(dataloader):
          optimizer.zero_grad()

          t = torch.randint(0, T, (BATCH_SIZE,), device=device).long()
          loss = get_loss(model, batch[0], t)
          loss.backward()
          optimizer.step()

          if epoch % 10 == 0 and step == 0:
            print(f"Epoch {epoch} | step {step:03d} Loss: {loss.item()} ")
            sample_plot_image()
```

Figure 39: Training loop

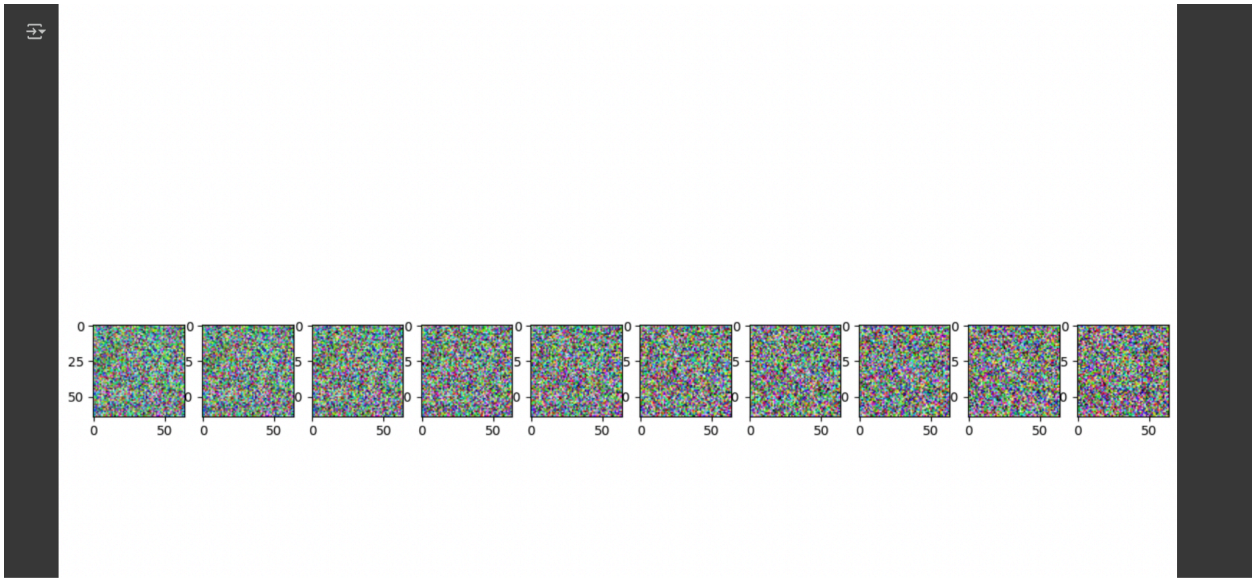

Figure 40: Generated image

3. Results

3.1 Autoencoder-Decoder Network

A performance metric was used to evaluate the model with an autoencoder-decoder network that was tasked for deepfake generation. Its reconstruction accuracy and the output's image quality will be assessed to determine if the implemented model is successful in generating deepfakes.

Metric	Value
Average MSE	0.0146
Average PSNR	18.87 dB
Average SSIM	0.6617

Figure 41: Table of performance metric for autoencoder-decoder network

As demonstrated in figure 41, the MSE of 0.0146 details how reconstruction is almost accurate with moderate structural similarity. It also reveals how the generated image has moderate quality due to the PSNR being 18.87. Thus, these results showcase the model was partially successful in generating deepfakes.

3.2 Generative Adversarial Network (GAN)

The image spectrum of the real image and the generated image was compared to assess the GAN's performance in successfully recreating the original image through its output. As showcased in figure 42, the fake image was able to capture the same frequency as the real image, resulting in the generator being able to produce clear generated images. There are no abnormalities except for the out corners and edges lacking, which can mean that the generated images need more spatial details and texture. Thus, these results demonstrate that the GAN model was mostly successful in generating a precise deepfake

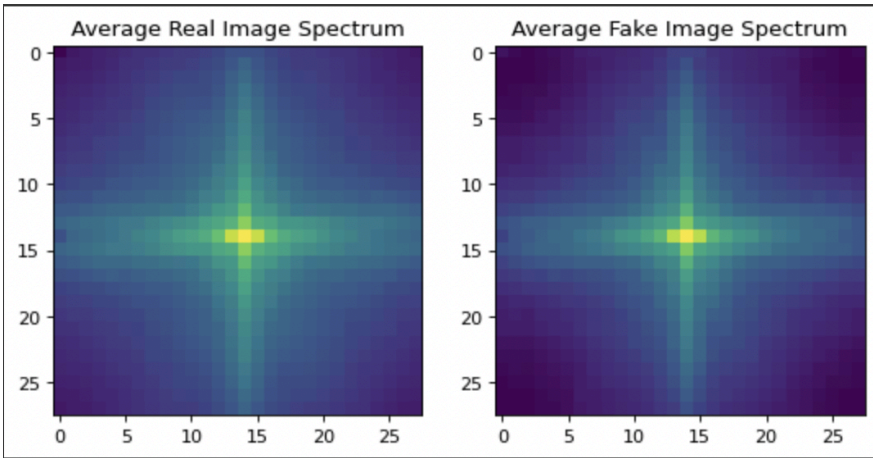

Figure 42: Average image spectrum of the real image and the fake image

2.3 Diffusion Model

The implemented diffusion model did not successfully generate an image. This can be due to undertraining as the training loop was stopped too early, resulting in the model to be unable to properly reverse the added noise as shown in figure 43.

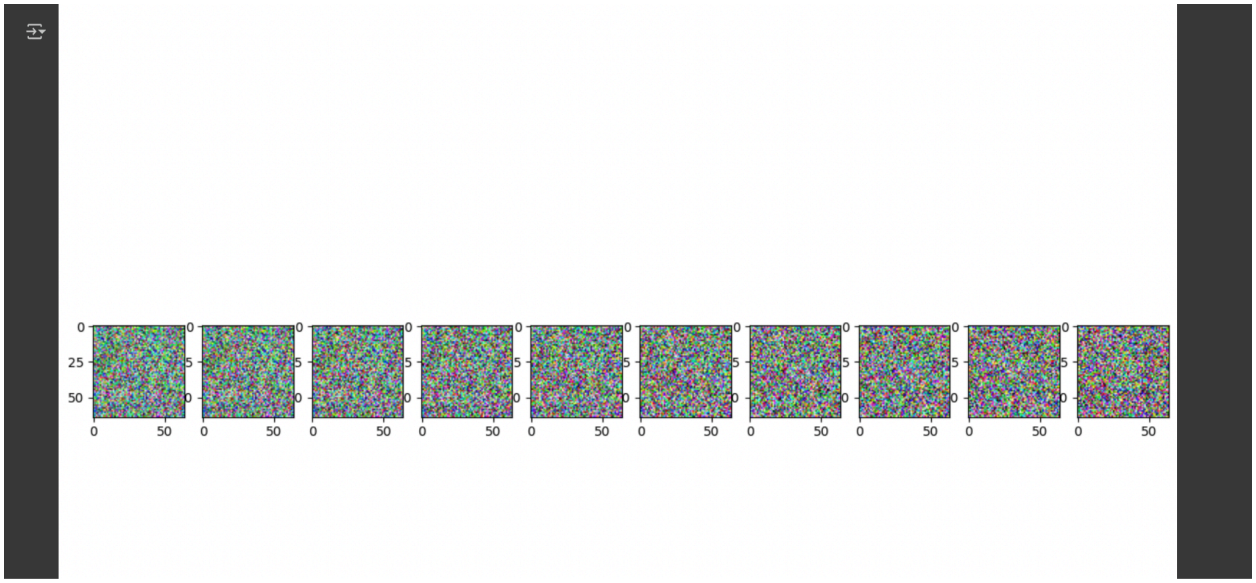

Figure 43: Generated image by diffusion model

4. Reflection

This project investigated the technical procedure of generating deepfakes through the method of utilising deep learning libraries such as “PyTorch” and “TensorFlow”. By building autoencoder-decoder networks, generative adversarial networks and diffusion models, a strong understanding of the architecture and how each model functions was developed.

However, there were challenges encountered throughout the project, such as the inability to utilise the “CelebA” dataset due to its large file size. Google Colab was unable to process it as it experienced computational constraints. The “FashionMNIST” dataset was used as a replacement, however it lacked the facial complexity necessary to simulate facial manipulation in deepfake generation as it only consisted of images of fashion objects. In future projects, an extended GPU could potentially be used to help with the computational costs needed to process large datasets needed for model training.

Furthermore, it was noted how easily accessible it was to generate deepfakes, as demonstrated through the two out of three successful deepfake models that were built. This showcases how deepfake attacks may progressively increase and risk digital safety, as deepfake tools may be exploited for the purpose of creating deepfakes for malicious motivations. As such, it is important to develop the necessary technology to detect deepfakes in order to prevent impersonation attacks or financial scams.

5. Conclusion

In this report, the investigation of three deepfake generation methods were proposed to assess how deepfakes negatively impact corporations, politics and individual safety. Amongst the three, Generative Adversarial Networks have demonstrated to be the most effective deepfake generator, as it was capable of teaching itself through adversarial training in order to generate hyper realistic images. The Autoencoder-Decoder Network was able to produce a deepfake image at a moderate quality, which is due to its simple structure and straightforward functions. Lastly, the Diffusion model was unable to successfully generate any images due to lack of training.

Furthermore, the investigation of these deepfake generation methods have revealed the easy accessibility to deepfake generation resources, enabling the public to be able build their own models through utilising the deep learning libraries available online. This may potentially lead to deepfake misuse and can result in deepfake attacks that are detrimental to digital safety. As such, this report reinforces the importance of developing deepfake detection systems in order to prevent these threats.

6. References

1. Alanazi, S., Asif, S. (2024). *Exploring deepfake technology: creation, consequences and countermeasures.* Volume 6, pages 49–60, (2024) https://link.springer.com/article/10.1007/s42454-024-00054-8

2. AWS. (2025). *What is a GAN.* https://aws.amazon.com/what-is/gan/#:~:text=A%20generative%20adversarial%20network%20(GAN,from%20a%20database%20of%20songs.

3. Babei, R., Cheng, S., Duan, R., Zhao, S., (2025, February 6). *Generative Artificial Intelligence and the Evolving Challenge of Deepfake Detection: A Systematic Analysis.* Volume 14, *14*(1), 17. https://doi.org/10.3390/jsan14010017

4. Bergmann, D., Stryker, C. (2024, August 21). *What are diffusion models?* https://www.ibm.com/think/topics/diffusion-models#:~:text=Diffusion%20models%20are%20generative%20models,to%20generate%20high%2Dquality%20images.

5. Bhattacharyya, C., Wang, H., Zhang, F., Kim, S., Zhu, X. (2025). *Diffusion Deepfake.* https://arxiv.org/pdf/2404.01579

6. CSIRO. (2024, February 9). *Keeping it real: How to spot a deepfake.* https://www.csiro.au/en/news/all/articles/2024/february/detect-deepfakes

7. De, M (2020, March 26). *Markovian.* https://community.ibm.com/community/user/blogs/moloy-de1/2020/03/26/points-to-ponder

8. Google Cloud. (2025). *What is Deep Learning?* https://cloud.google.com/discover/what-is-deep-learning

9. Holdsworth, J. (2024, June 17). *What is deep learning?* https://www.ibm.com/think/topics/deep-learning

10. Patel, Y., Tanwar, S., Gupta, R., Bhattacharya, P., Davidson, I., Nyamejo, R. (2025). *Deepfake Generation and Detection: Case Study and Challenges.* Volume 11 https://ieeexplore.ieee.org/document/10354308

11. Prasad, P. (2024, April 2). *Artificial Faces: The Encoder-Decoder and GAN Guide to Deepfakes.* https://medium.com/@priyanshuprasad1718/artificial-faces-the-encoder-decoder-and-gan-guide-to-deepfakes-75a1eed0e265

12. Sapien. (2024). How Diffusion Models Work: A Detailed Step-by-Step Guide. https://www.sapien.io/blog/how-diffusion-models-work-a-detailed-step-by-step-guide#:~:text=In%20the%20forward%20diffusion%20process,the%20previous%20step's%20noisy%20state.

13. Superannotate. (2025). *Introduction to diffusion models for machine learning.* https://www.superannotate.com/blog/diffusion-models#what-are-diffusion-models

14. v7labs. (2021). *Autoencoders in Deep Learning: Tutorial & Use Cases [2024].* https://www.v7labs.com/blog/autoencoders-guide

15. Western Australian Government. (2024). *The Growing Threat of Deepfakes: What You Need to Know.* https://www.wa.gov.au/system/files/2024-10/the.growing.threat.of_.deepfakes.article.docx

16. Zvkovic, S. (2022, September 5). *#012 Understanding Latent Space in Generators.* https://datahacker.rs/015-understanding-latent-space-in-generators/

17. Zhong, W., Lin, J., Chen., Lin, L., Li,G., (2024, May). *High-fidelity and Lip-synced Talking Face Synthesis via Landmark-based Diffusion Model.* VOL. 14, NO. 8 https://arxiv.org/pdf/2408.05416