Advances in machine vision

Course lead by professor Hedi Tabia

Use of GAN for data augmentation

Author:
Giuseppe Ricciardi

Student number:
20235324

Sunday 31st December, 2023

Contents

References

[1] I. Goodfellow, J. Pouget-Abadie, M. Mirza, *et al.*, "Generative adversarial networks," *Advances in neural information processing systems*, vol. 27, 2014.

[2] A. Radford, L. Metz, and S. Chintala, "Unsupervised representation learning with deep convolutional generative adversarial networks," *arXiv preprint arXiv:1511.06434*, 2015.

[3] M. Arjovsky, S. Chintala, and L. Bottou, "Wasserstein generative adversarial networks," pp. 214–223, 2017.

[4] A. B. L. Larsen, S. K. Sønderby, H. Larochelle, and O. Winther, "Autoencoding beyond pixels using a learned similarity metric," in *International conference on machine learning*, PMLR, 2016, pp. 1558–1566.

[5] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[6] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Communications of the ACM*, vol. 60, no. 6, pp. 84–90, 2017.

Introduction

Data augmentation is crucial for improving machine learning models by making training datasets more diverse, preventing overfitting, and enhancing the model's ability to perform well on new, unseen data.
Generative Adversarial Networks (GANs) play a key role in this by creating realistic synthetic data, enriching the training set, and helping the model better understand complex patterns in the data.
The collaboration between data augmentation and GANs boosts the overall strength and performance of machine learning models in various applications.
This report aims to explore the application of GANs in the context of data augmentation.
In Chapter 1, we present an overview of the dataset utilized for training various models, along with preliminary operations conducted on it. Chapter 2 delves into the detailed implementation of three GAN architectures: DCGAN, WGAN, and VAE-GAN (further elaborated in 2.1)—as well as the incorporation of a CNN classifier to evaluate the efficacy of the achieved results (2.2).
Chapter 3 systematically analyzes the outcomes, while Section 4 provides concluding remarks and explores potential directions for future development.

1 Dataset

The provided dataset comprises 164 color images, each sized 256x256 pixels, depicting various angles of 3D models. Among these, 82 images showcase a 3D model of a cow, while the remaining 82 represent a 3D model of a horse. The images include diverse angles to capture different perspectives of the models. The dataset has been segregated into 82 images for training and 82 images for testing, ensuring an equitable distribution with 42 images for each category in both sets. The project was realised entirely in python, using Pytorch.

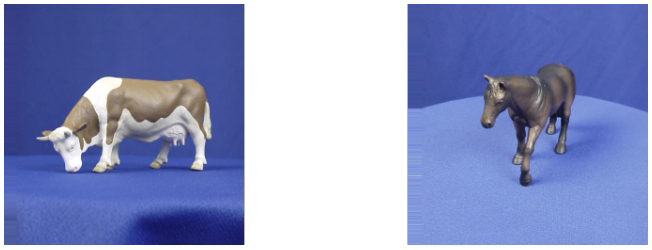


Figure 1: Examples of training images

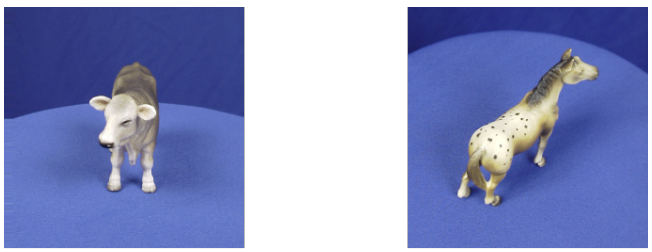


Figure 2: Examples of test images

It's important to note that the test images exhibit variations in the texture of the training models.
To utilize the images in the dataset, some preliminary operations were undertaken:

- *Resize to* 64×64: An attempt to use 128×128 images was made, but it yielded limited results, mainly due to prolonged computational times.
- Transformation of images into tensors.
- *Batch division* of the dataset, each batch has a size of 4.

Printing the standard deviation, mean, maximum and minimum values of the dataset shows that it is already *normalised*. Additionally, to further enhance the learning capabilities of the models, a **data augmentation** operation was performed.

```python
def perform_augmentation(self):
    augmentation_options = [
        transforms.RandomHorizontalFlip(),
        transforms.RandomRotation(degrees=10),
        transforms.RandomVerticalFlip(),
        transforms.RandomPerspective(distortion_scale=0.2)
    ]
    for file in self.files:
        img_path = os.path.join(self.root, file)
        original_image = Image.open(img_path).convert('RGB')
        # Randomly select one augmentation
        selected_augmentation = random.choice(augmentation_options)
        augmented_image = self.apply_transform(original_image, selected_augmentation)
        augmented_filename = f"augmented_{file}"
        augmented_root = os.path.join(self.root, "augmented")
        augmented_path = os.path.join(augmented_root, augmented_filename)
        print(f"Saving {augmented_path}")
        augmented_image.save(augmented_path)
```

Figure 3: Code of data augmentation operation

The `perform_augmentation` function enhances the dataset by iterating through each image file and applying a randomly chosen augmentation from a predefined set of options such as random horizontal and vertical flips, random rotation (up to 10 degrees), and random perspective distortion with a scale of 0.2.
For each image, the function first loads the original image, converts it to RGB format, selects an augmentation, and applies it using the `apply_transform` method. The resulting augmented image is then saved with a new filename in a designated directory.

2 Implementation

The project consists of 3 files: `GAN_models.py`, `CNN.py`, `main.py`.

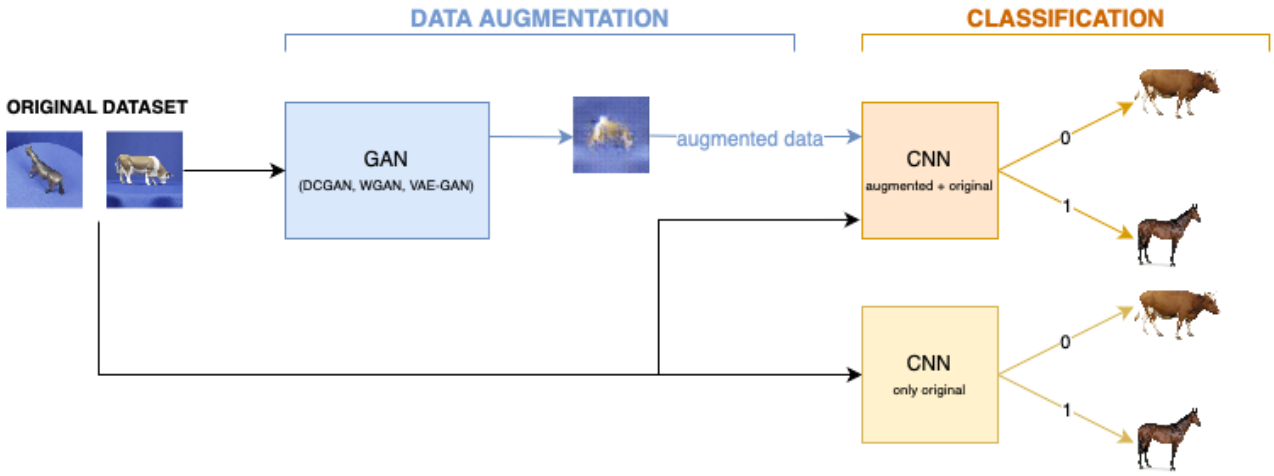


Figure 4: Representation of the application

The application's functionality is illustrated in Figure 4. When executing `main.py`, users have the option to perform a preliminary data augmentation operation (Fig. 3). After this, users choose which GAN model to train: DCGAN, WGAN, or VAE-GAN. Subsequently, the images generated by the trained model are processed by a CNN, specifically trained for classification to distinguish between cows and horses. The resulting outcomes are then compared with the same CNN model trained on the original dataset for evaluation.
In this chapter, we will delve into the various components of the application.

2.1 GAN

Generative Adversarial Networks (GANs) represent a class of artificial intelligence models, introduced by Ian Goodfellow in 2014 [1]. GANs employ a unique neural network architecture, engaging in an adversarial interplay between two key components. The **generator network** produces synthetic data instances, while the **discriminator network** evaluates their authenticity. This competitive process enables GANs to excel in generating high-quality, realistic synthetic data, proving invaluable in applications such as image synthesis, style transfer, and data augmentation.
Let x represent image data. $D(x)$ is the discriminator network, outputting the probability that x came from training data instead of the generator. For images, the input to $D(x)$ is an image of size $3 \times 64 \times 64$. Intuitively, $D(x)$ should output a HIGH value for real training data and a LOW value for generated data, acting like a traditional binary classifier.
For the generator, let z be a latent space vector sampled from a standard normal distribution. $G(z)$ is the generator function, mapping the latent vector z to data-space. The goal of G is to estimate the distribution of the training data so it can generate fake samples from that estimated distribution.
So, $D(G(z))$ is the probability that the output of the generator G is a real image. In the GAN minimax game, D aims to maximize the probability of correctly classifying real and fake images, while G tries to minimize the probability that D predicts its outputs as fake. The **GAN loss function** is defined as:

$$\min_G \max_D V(D,G) = E_{x \sim p_{data}(x)}[\log D(x)] + E_{z \sim p_z(z)}[\log(1 - D(G(z)))] \tag{1}$$

In theory, the solution is when $p_g = p_{data}$, and the discriminator guesses randomly between real and fake. However, the convergence theory of GANs is an ongoing research area, and real-world models may not always reach this ideal point. In the proposed realisation, three different GAN models were chosen: **DCGAN**, **WGAN**, **VAE-GAN**.

2.1.1 DCGAN

Introduced by Radford and his collaborators in 2015 [2], **DCGANs** are designed for image generation tasks, showcasing marked enhancements in both training stability and the quality of generated images.
DCGANs leverage a *deep convolutional architecture* in both the generator and discriminator networks, enabling them to more effectively capture hierarchical features and spatial dependencies in the input data. The generator network employs transposed convolutional layers to transform noise into progressively intricate representations, culminating in the production of high-resolution images. Simultaneously, the discriminator network utilizes convolutional layers to distinguish between real and generated images. Key architectural principles of DCGANs include the avoidance of fully connected layers, the incorporation of batch normalization for stable training, and the use of Leaky ReLU activations to mitigate *mode collapse*, an issue where the generator yields a limited range of outputs. The training of DCGAN involves the application of the **Binary Cross Entropy** (BCE) loss function, defined as:

$$l(x,y) = L = l_1, \ldots, {l_N}^T, l_n = -[y_n * log(x_n) + (1 - y_n) * log(1 - x_n)] \tag{2}$$

The function $l(x,y)$ denotes the Binary Cross Entropy (BCE) loss, measuring dissimilarity between predicted values x_n and true labels y_n across a set of samples. Commonly used in binary classification tasks, such as discerning real and generated samples in a GAN, this loss is pivotal in training machine learning models. The vector L, with the transpose T, aggregates individual loss values for each sample. l_n signifies the loss for the n-th sample, calculated through the BCE formula: $l_n = -[y_n \cdot \log(x_n) + (1 - y_n) \cdot \log(1 - x_n)]$, where y_n represents the true label (1 for positive, 0 for negative), x_n is the predicted probability for the n-th sample belonging to the positive class (e.g., real data in GANs).

To optimize the model parameters during training, the DCGAN implementation utilizes the **RMSprop optimizer**. The Root Mean Square Propagation algorithm maintains a moving average of the squared gradients for each parameter. It then scales the learning rates inversely proportional to the square root of this moving average. This adaptive adjustment allows RMSprop to converge more efficiently and handle varying gradients across different parameters.

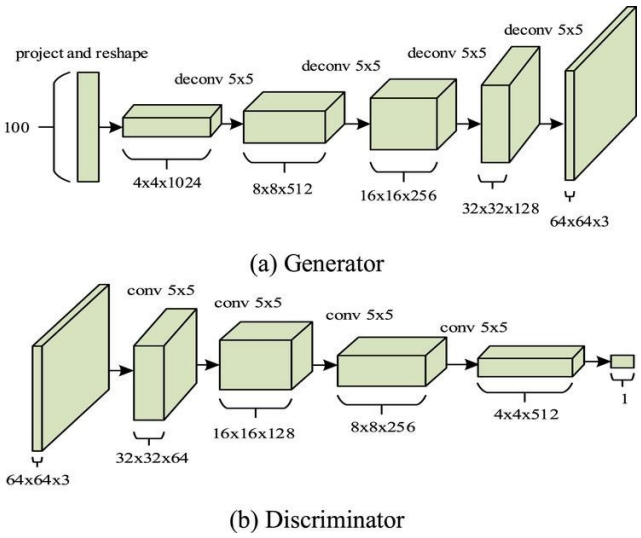


Figure 5: Typical DCGAN Architecture

Let's examinate the architecture of the proposed DCGAN: the generator initiates with a latent vector (z) of size nz and passes through a series of transposed convolutional layers, totaling five layers. These layers progressively upscale the input, starting with $ngf \times 8$ channels and ending with nc channels, generating images of 64×64 resolution. Batch normalization enhances stability during training, while Rectified Linear Unit (ReLU) activations introduce non-linearity. The discriminator, designed to distinguish between real and generated samples, comprises six convolutional layers. Beginning with nc channels of 64×64 resolution, the discriminator downscales the spatial dimensions, ending with a single channel and a Sigmoid activation function. Leaky ReLU activations are applied to introduce non-linearity and address the "dying ReLU" problem.

```
#Generator
self.generator = nn.Sequential(
    # input is Z, going into a convolution
    # state size. ngf*8 is the number of channels in the output image, 4 is the kernel size and 4 is the stride size (4x4).
    nn.ConvTranspose2d( self.nz, self.ngf * 8, 4, 1, 0, bias=False),
    #Batch normalization is a technique for improving the speed, performance, and stability of artificial neural networks. It is used to normalize the input layer by adjusting and scaling the activations.
    #This is done by calculating the mean and variance of the input layer, then applying the normalization equation to each input.
    nn.BatchNorm2d(self.ngf * 8),
    #Use the rectified linear unit function as the activation function. The rectified linear activation function is a piecewise linear function that will output the input directly if is positive, otherwise, it will output zero.
    nn.ReLU(True),
    #Repeat the same process as before, but with different number of channels in the output image. The number of channels in the output image is halved at each step.
    nn.ConvTranspose2d(self.ngf * 8, self.ngf * 4, 4, 2, 1, bias=False),
    nn.BatchNorm2d(self.ngf * 4),
    nn.ReLU(True),
    nn.ConvTranspose2d( self.ngf * 4, self.ngf * 2, 4, 2, 1, bias=False),
    nn.BatchNorm2d(self.ngf * 2),
    nn.ReLU(True),
    nn.ConvTranspose2d( self.ngf * 2, self.ngf, 4, 2, 1, bias=False),
    nn.BatchNorm2d(self.ngf),
    nn.ReLU(True),
    nn.ConvTranspose2d( self.ngf, self.nc, 4, 2, 1, bias=False),
    #Last activation function is a Tanh function, a type of activation function that is similar to the sigmoid. It is used to normalize the output of a network to a range between -1 and 1.
    nn.Tanh()
)
#Discriminator
self.discriminator = nn.Sequential(
    #Input is ``(nc) x 64 x 64``, where nc is the number of channels in the input image, 64 is the height and 64 is the width.
    #Convolutional layer with 3 input channels, 64 output channels, 4 kernel size and 2 stride size (4x4).
    nn.Conv2d(self.nc, self.ndf, 4, 2, 1, bias=False),
    #Use the Leaky ReLU function as the activation function. The Leaky ReLU function is an improved version of the ReLU function. The Leaky ReLU function addresses the "dying ReLU" problem
    #by having a small negative slope in the negative section, instead of altogether zero.
    nn.LeakyReLU(0.2),
    #Repeat the same process as before, but with different number of channels in the output image. The number of channels in the output image is doubled at each step.
    nn.Conv2d(self.ndf, self.ndf * 2, 4, 2, 1, bias=False),
    #Batch normalization applied to the output of the convolutional layer as for the generator.
    nn.BatchNorm2d(ndf * 2),
    nn.LeakyReLU(0.2),
    nn.Conv2d(self.ndf * 2, self.ndf * 4, 4, 2, 1, bias=False),
    nn.BatchNorm2d(self.ndf * 4),
    nn.LeakyReLU(0.2),
    nn.Conv2d(self.ndf * 4, self.ndf * 8, 4, 2, 1, bias=False),
    nn.BatchNorm2d(self.ndf * 8),
    nn.LeakyReLU(0.2),
    nn.Conv2d(self.ndf * 8, 1, 4, 1, 0, bias=False),
    #Last activation function is a Sigmoid function to maps input values to probabilities between 0 and 1.
    nn.Sigmoid()
)
```

Figure 6: Code snippet of the implemented DGAN

2.1.2 WGAN

Wasserstein Generative Adversarial Networks (WGAN) represent a significant advancement in the GAN framework, introducing a novel approach to the training dynamics and addressing issues related to mode collapse and training instability. Proposed by Arjovsky et al. in 2017 [3], WGAN diverges from the traditional GAN setup by introducing the *Wasserstein distance* as the training objective. The distinctive feature of WGAN lies in its emphasis on the Wasserstein distance, also known as Earth Mover's distance, as a metric for measuring the dissimilarity between the true and generated distributions. Unlike traditional GANs, WGAN encourages the discriminator (critic) to produce output values that are not just probabilities but represent the difference in the Wasserstein space. This modification has profound implications, leading to more stable training, improved convergence, and mitigation of mode collapse, where the generator produces limited varieties of outputs. The WGAN critic is designed to have a Lipschitz continuity, which ensures the existence of gradients almost everywhere. To enforce this Lipschitz constraint, *weight clipping* has been conventionally applied to the critic's parameters.

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

Require: : α, the learning rate. c, the clipping parameter. m, the batch size. n_{critic}, the number of iterations of the critic per generator iteration.
Require: : w_0, initial critic parameters. θ_0, initial generator's parameters.

while θ has not converged **do**
  for $t = 0, ..., n_{\text{critic}}$ **do**
    Sample $\{x^{(i)}\}_{i=1}^{m} \sim \mathbb{P}_r$ a batch from the real data.
    Sample $\{z^{(i)}\}_{i=1}^{m} \sim p(z)$ a batch of prior samples.
    $g_w \leftarrow \nabla_w \left[\frac{1}{m}\sum_{i=1}^{m} f_w(x^{(i)}) - \frac{1}{m}\sum_{i=1}^{m} f_w(g_\theta(z^{(i)}))\right]$
    $w \leftarrow w + \alpha \cdot \text{RMSProp}(w, g_w)$
    $w \leftarrow \text{clip}(w, -c, c)$
  end for
  Sample $\{z^{(i)}\}_{i=1}^{m} \sim p(z)$ a batch of prior samples.
  $g_\theta \leftarrow -\nabla_\theta \frac{1}{m}\sum_{i=1}^{m} f_w(g_\theta(z^{(i)}))$
  $\theta \leftarrow \theta - \alpha \cdot \text{RMSProp}(\theta, g_\theta)$
end while

Figure 7: Algorithm for the WGAN, taken from [3]

The WGAN implementation follows the architecture of a DCGAN. The generator and discriminator networks share a similar structure with transposed convolutions and leaky rectified linear unit (ReLU) activations. The crucial departure lies in the choice of the **Wasserstein loss function**.
The generator consists of several transposed convolutional layers, starting from a latent space vector (z) and gradually upsampled to produce realistic images. Batch normalization and leaky ReLU activations are applied to stabilize and enhance the learning process.
The discriminator, designed to distinguish between real and generated images, employs convolutional layers with leaky ReLU activations as well.
The implemented weight initialization function assigns appropriate initial values to the weights of convolutional and batch normalization layers. Convolutional layer weights are initialized with a normal distribution with a mean of 0 and a standard deviation of 0.02, while batch normalization layer weights are initialized with a mean of 1 and a standard deviation of 0.02, with biases set to 0.
The Wasserstein loss function is a pivotal element in the training process, specifically tailored for WGANs. The loss function computes the Wasserstein distance between the distributions of real and generated samples. Notably, the implementation utilizes the RMSprop optimizer for both the generator and discriminator, emphasizing the magnitude of recent gradients to normalize training dynamics.
During training iterations, the discriminator's weights are clipped to adhere to the Lipschitz constraint, facilitating more stable learning. The Wasserstein loss is then computed by evaluating the Wasserstein distance between the distributions of real and generated samples. This strategic combination of weight clipping and the Wasserstein loss function contributes to the overall stability and effectiveness of the WGAN implementation.

2.1.3 VAE-GAN

The VAE-GAN architecture [4] seamlessly combines Variational Autoencoders (VAEs) and Generative Adversarial Networks (GANs), creating a robust generative modeling framework. The encoder utilizes convolutional layers with batch normalization and leaky ReLU activations to extract meaningful representations. Two parallel fully connected layers parameterize the latent space, generating mean (μ) and log variance ($\log \sigma^2$) vectors. The reparameterization trick samples latent vectors from a normal distribution. The decoder, mirroring the encoder, employs transposed convolutions to upsample latent representations, aiming for faithful input reconstruction. The VAE loss includes a reconstruction term measuring fidelity and a regularization term enforcing a Gaussian latent space. Adversarial training introduces a GAN-like discriminator to distinguish real and generated samples, enhancing image quality and diversity. This hybrid approach optimizes both reconstruction accuracy and adversarial training for improved generative performance.

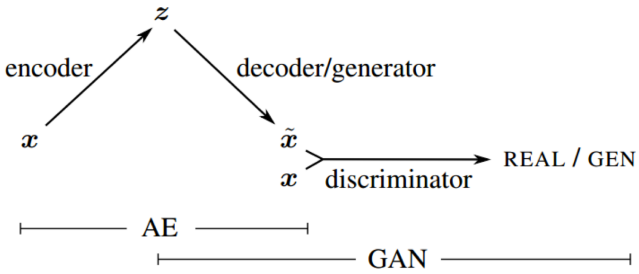


Figure 8: Vae-Gan architecture representation

The VAE-GAN implementation involves two main components: the VAE and the Discriminator (Dis). The VAE captures the latent space structure and provides a mechanism for generating realistic samples, while the Discriminator evaluates the authenticity of generated and real images. The training procedure iteratively updates both components to achieve a balance between reconstruction accuracy and adversarial training.
The VAE component consists of an encoder and decoder. The encoder processes input images through convolutional layers, producing mean and log variance vectors. The reparameterization trick generates latent vectors, which are then decoded by transposed convolutional layers to reconstruct the input.
The implemented VAE further includes a custom weight initialization function to set the initial weights of convolutional and batch normalization layers.
The Discriminator is composed of convolutional layers followed by a fully connected layer. It evaluates the authenticity of both real and generated images. The VAE's reconstructed images and real images are fed into the Discriminator, and their scores are used to compute adversarial loss. The weight clipping technique is applied during training to enforce Lipschitz continuity on the Discriminator's parameters.
The training process involves optimizing the VAE and Discriminator separately.
The VAE minimizes a loss function comprising a Mean Squared Error (MSE) term and a Kullback-Leibler Divergence (KLD) term, ensuring faithful reconstruction and regularization of the latent space.
The Discriminator is trained using a binary cross-entropy loss, distinguishing between real and generated samples.

Listing 1: VAE-GAN Architecture and Loss script

```
class VAE(nn.Module):
    def __init__(self, nz, nc, device):
        super(VAE, self).__init__()
        self.device = device
        self.nz = nz
        self.nc = nc
        self.encoder = nn.Sequential(
            nn.Conv2d(self.nc, 16, kernel_size=3, stride=2, padding=1),
            nn.BatchNorm2d(16),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Conv2d(16, 32, kernel_size=3, stride=2, padding=1),
            nn.BatchNorm2d(32),
            nn.LeakyReLU(0.2,inplace=True),
            nn.Conv2d(32, 32, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(32),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Conv2d(32, 64, kernel_size=3, stride=1, padding=1),
```

```
            nn.MaxPool2d((2, 2)),
        )
        self.encoder_fc1 = nn.Linear(64 * 8 * 8, self.nz)
        self.encoder_fc2 = nn.Linear(64 * 8 * 8, self.nz)
        self.decoder_fc = nn.Linear(self.nz, 64 * 8 * 8)  # Adjusted output size
        self.decoder = nn.Sequential(
            nn.ConvTranspose2d(64, 32, kernel_size=3, stride=3, padding=1,
                output_padding=0),
            nn.ReLU(),
            nn.ConvTranspose2d(32, self.nc, kernel_size=3, stride=3, padding=1,
                output_padding=0),
            nn.Sigmoid(),
        )

class VAE_GAN (nn.Module):
    def __init__(self, nz, ngf, ndf, nc, lr,device,dataloader):
        super(VAE_GAN, self).__init__()
        #nz is the size of the latent z vector
        self.nz = nz
        #ngf is the size of feature maps in the generator
        self.ngf = ngf
        #ndf is the size of feature maps in the discriminator
        self.ndf = ndf
        #nc is the number of channels in the training images. For color images this
            is 3
        self.nc = nc
        self.device = device
        #VAE part
        self.vae = VAE(self.nz,self.nc,self.device)
        #GAN part (Discriminator)
        #Discriminator is composed of convolutional layers.
        #The discriminator takes as input the reconstructed images from the VAE and
            the real images and classifies them as real or fake.
        self.dis = nn.Sequential(
            nn.Conv2d(self.nc, 32, 3, stride=1, padding=1),
            nn.LeakyReLU(0.2, True),
            nn.MaxPool2d((2, 2)),
            nn.Conv2d(32, 64, 3, stride=1, padding=1),
            nn.LeakyReLU(0.2, True),
            nn.MaxPool2d((2, 2)),
        )
        self.fc = nn.Sequential(
            nn.Linear(8 * 8 * 64, 1024),
            nn.LeakyReLU(0.2, True),
            nn.Linear(1024, 1),
            nn.Sigmoid()
        )
        self.optimizerV = optim.Adam(self.vae.parameters(), lr=lr-0.00006)
        self.optimizerD = optim.Adam(self.dis.parameters(), lr=lr)
        self.dataloader = dataloader
        self.Dlosses = []
        self.Vaelosses = []
    def loss_function(self, recon_x, x, mean, logstd):
        # Compute Mean Squared Error (MSE) loss
        MSECriterion = nn.MSELoss().to(self.device)
        recon_x_resized = F.interpolate(recon_x, size=x.size()[2:], mode='bilinear',
            align_corners=False)
        MSE = MSECriterion(recon_x_resized, x)
        # Because var is a log-variance, we need to exponentiate it to get the
            variance
        var = torch.pow(torch.exp(logstd), 2)
        # Compute Kullback-Leibler Divergence (KLD) loss
        KLD = -0.5 * torch.sum(1 + torch.log(var) - torch.pow(mean, 2) - var)
        return MSE + KLD
```

2.2 CNN

Convolutional Neural Networks (CNNs) represent a sophisticated class of neural networks, designed to extract intricate hierarchical features from input data, particularly well-suited for image-based tasks [5], [6] such as binary classification.
The first layer in a CNN is the input layer, which takes in the raw data, often an image represented as a grid of pixels. Following the input layer, a series of convolutional layers come into play.
These layers employ convolutional filters, also known as kernels, to scan the input data. Each filter specializes in recognizing specific patterns or features. Convolutional operations systematically slide these filters across the input image, enabling the network to learn local patterns, edges, and complex hierarchical features. Strategically placed pooling layers follow the convolutional layers. These layers, commonly max-pooling or average-pooling, serve to downsample the spatial dimensions of the data while retaining the most salient features. Pooling reduces computational complexity and makes the network more resilient to variations in input scale and orientation.
Subsequently, fully connected layers are introduced to integrate the learned features and make final predictions. These layers connect every neuron to every neuron in the preceding and succeeding layers, capturing global dependencies within the data. The final layer typically employs a softmax activation function, enabling the network to output probability scores for each class. The class with the highest probability is then chosen as the predicted class for the input instance.

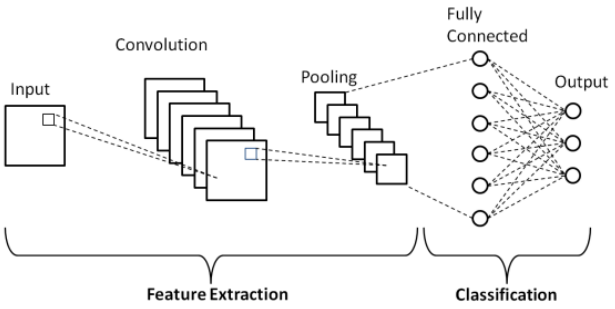


Figure 9: Generalization of CNN architecture

The implemented CN architecture comprises multiple convolutional layers interleaved with Rectified Linear Unit (ReLU) activation functions and max-pooling layers for spatial downsampling. Beginning with a 3-channel input layer, the network progresses through convolutional blocks with increasing filter sizes (32, 64, 128, and 256). Each convolutional block is followed by max-pooling to capture essential features and reduce spatial dimensions. The final flattened layer connects to fully connected layers with decreasing neurons (1024, 512) before culminating in a 2-neuron output layer representing the binary classification of cow or horse. The *Kaiming weight initialization* method is employed to initialize the weights of convolutional and linear layers. Training utilizes the Adam optimizer with a learning rate of 0.001 over 10 epochs. Performance evaluation on a test set includes metrics such as accuracy, a classification report, and a confusion matrix.
The project is organized into three key stages:

1. The first step involves the generation of new images using the GAN models.
2. Subsequently, the original dataset images are subjected to classification using the CNN model.
3. The final step extends the classification process to include images from the augmented dataset, employing the same CNN architecture.

3 Results

This chapter presents a detailed analysis of outcomes achieved in the three distinct stages of the project. The performance and results of each GAN model are examined in 3.1.
Following 3.2 the exploration of GAN models, attention is shifted to the application of Convolutional Neural Network (CNN) on the original dataset.
Subsequently, focus is directed towards the evaluation of CNN performance on the augmented dataset in 3.3.
Through these analyses, a comprehensive understanding of the project's results is provided, highlighting key insights and implications.

3.1 GAN results

The GAN training process involved inputting both 128x128 and 64x64 images. Despite the fact that using larger dimensions led to sharper results, especially in the case of VAE-GAN, the computational and time costs favored the decision to utilize 64x64 images. All the results presented in this section are obtained through training with a learning rate of 0.0001 and an Adam optimizer beta value of 0.49. These specific values for the learning rate and Adam beta are selected based on state-of-the-art practices in the field. Additionally, it's noteworthy that the training utilized manually augmented data (3), where 41 new images were added for each class.
The results of the Deep Generative Adversarial Network DGAN training are presented below. The output is organized into batches, where each batch consists of a certain number of iterations. For instance, in the 50th batch, the discriminator loss is 0.0000, the generator loss is 23.3620, the real data discriminator output is 1.0000, and the generated data discriminator output is 0.0000 / 0.0000.

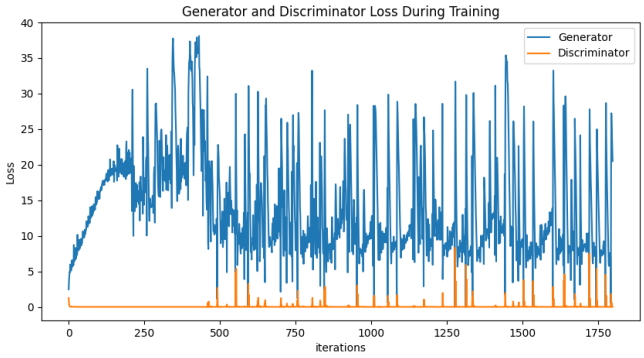


Figure 10: Loss graphic of DGAN

Convergence of the discriminator loss to 0 is not necessarily advantageous. The ideal scenario in GAN training involves a balanced competition between the generator and the discriminator. The discriminator's loss reaching 0 too quickly could indicate that the generator is failing to produce samples challenging enough to deceive the discriminator. The sudden drop in the generator's loss to 0, could indicate a collapse in the diversity of generated samples. This means that the generator might be producing a limited set of samples that the discriminator easily recognizes, resulting in a loss of the adversarial dynamics crucial for effective GAN training.

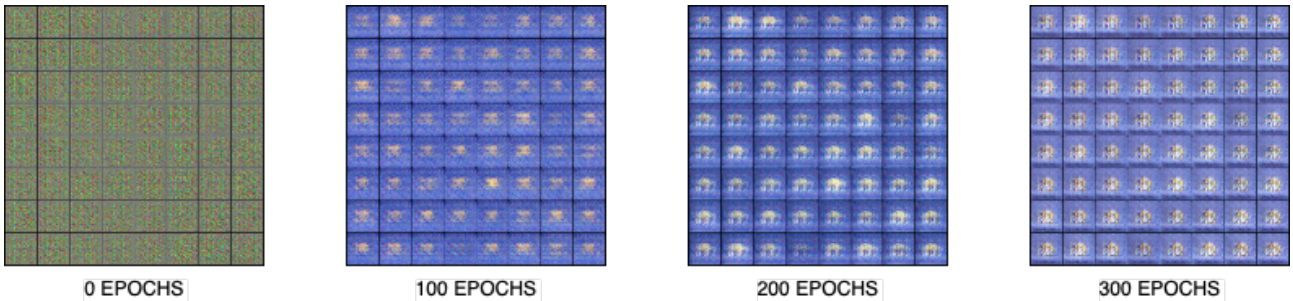


Figure 11: DCGAN Generated images during training

The WGAN exhibits distinctive characteristics in its training dynamics that contribute to superior results. Unlike traditional GANs, the adversarial interplay between the generator and discriminator in WGAN is notably improved due to the use of Wasserstein distance. The discriminator's loss, instead of converging to 0 rapidly, tends to stabilize at a non-zero value.

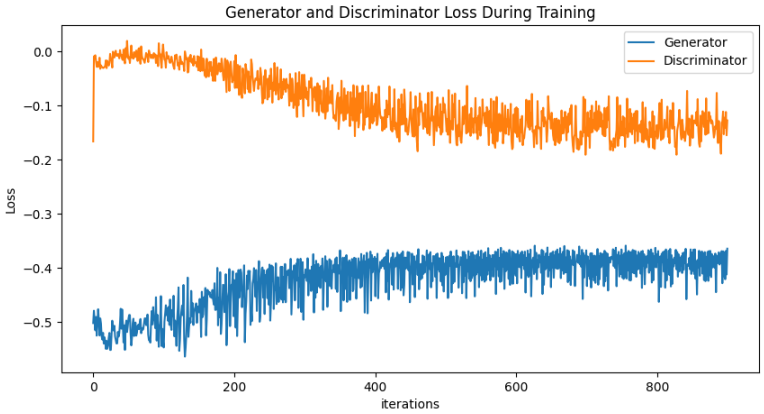


Figure 12: Wgann loss graphic

The result is a GAN that is less prone to issues like mode collapse and is capable of generating high-quality images with improved fidelity and variety.

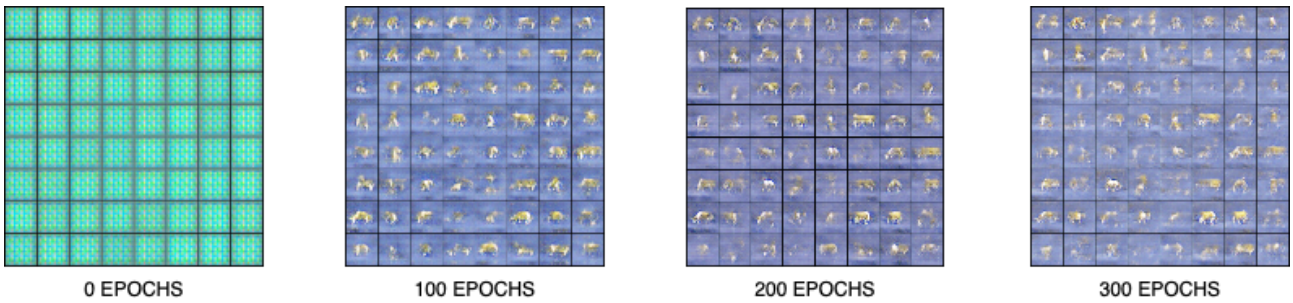


Figure 13: WGAN Generated images during training

The results from the VAE-GAN didn't perform as well compared to the other methods we tried. It struggled to generate good images, except when using a resolution of 128x128 pixels. The issue was that achieving good results at this resolution took a really long time on the computer. So, while it showed some promise in making decent images, the VAE-GAN faced challenges in finding the right balance between image quality and the time it takes to create them. It indicates that there's room for improvement and tweaks to make VAE-GAN more effective in generating high-quality images without taking forever to do it.

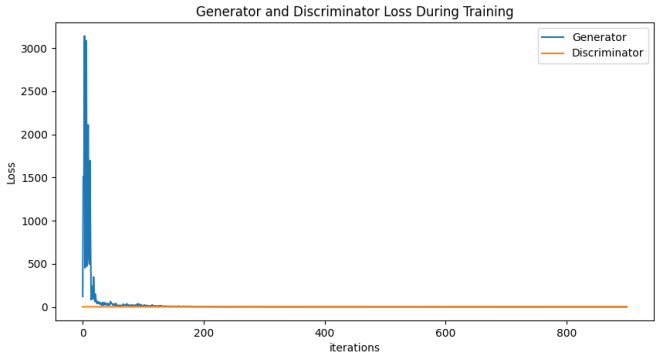


Figure 14: VAE-GAN losses values

In the plot, it's noticeable that the generator's loss values are quite high, indicating some challenges or struggles in its training process. On the other hand, the discriminator's loss tends toward zero. This suggests that the generator is finding it difficult to fool the discriminator consistently.

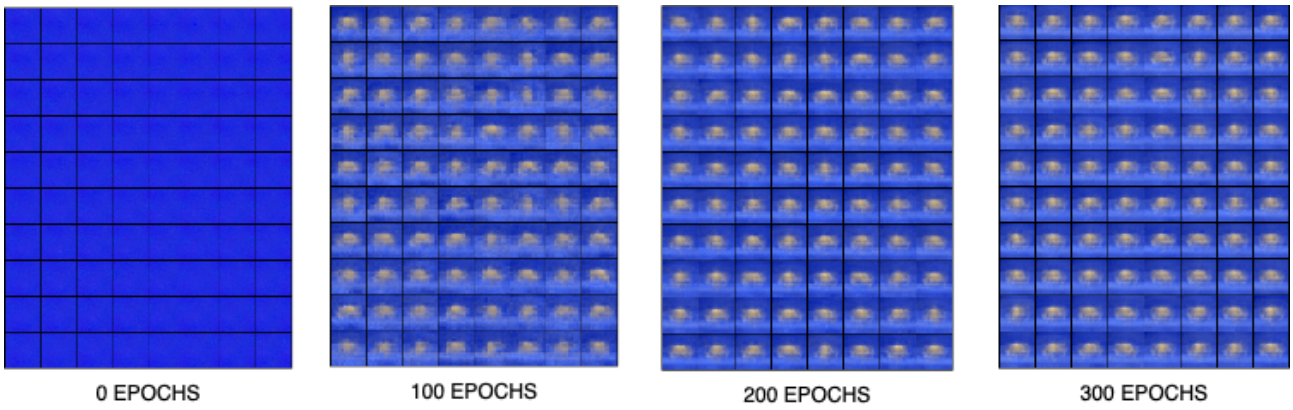


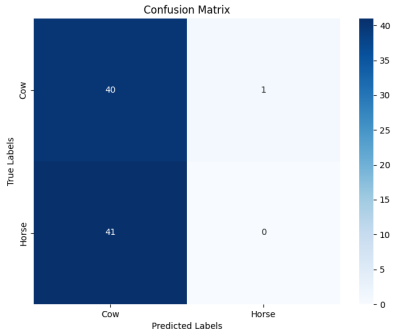
Figure 15: VAE-GAN generated images during training

The generated images exhibit a lack of distinguishable features related to different classes, making them unsuitable for effective classification. In the context of image classification tasks, clear and discernible features are essential for accurate model performance.

3.2 CNN classification without augmented data

A notable challenge in the classification with original dataset arises from the small dimension of the dataset, leading to an imbalanced distribution of classes and impacting the model's ability to generalize effectively. The dataset comprises 82 images, categorizing them into two classes: 'Cow' and 'Horse.' However, the limited dataset size results in an inherent class imbalance, particularly impacting the model's ability to generalize effectively. The class distribution reveals that the training set consists of 28 'Horse' images and 24 'Cow' images, while the validation set comprises 13 'Horse' images and 17 'Cow' images. The model's performance is reflected in the training and validation losses over ten epochs, showcasing a decrease in both losses, yet the validation accuracy plateaus at 57.14% after the first epoch.

On the test set, the overall accuracy is approximately 48.78%, indicating challenges in predicting the minority class ('Horse'). The detailed classification report further highlights these difficulties, with low precision, recall, and F1-score for class 'Horse.'



Test Set Results:
Accuracy: 0.4390
Classification Report:

	precision	recall	f1-score	support
0	0.46	0.73	0.57	41
1	0.35	0.15	0.21	41
accuracy			0.44	82
macro avg	0.41	0.44	0.39	82
weighted avg	0.41	0.44	0.39	82

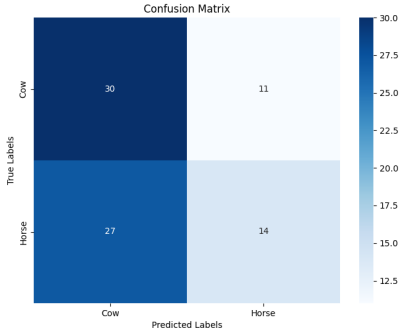
Figure 16: Confusion Matrix (a) and Report (b) of testing with original dataset

3.3 CNN Classification with augmented data

After adding 10 images generated from the GAN for each labe to augment the dataset using the WGAN model, we proceed with retraining the CNN. Below, are presented some of the images that have been added to the dataset:

Figure 17: New images added to the dataset

The new training is done with the same parameters as before to having a better comparison about the influence of the new data to the predictions. These are the obtained results



Test Set Results:
Accuracy: 0.5366
Classification Report:

	precision	recall	f1-score	support
0	0.53	0.73	0.61	41
1	0.56	0.34	0.42	41
accuracy			0.54	82
macro avg	0.54	0.54	0.52	82
weighted avg	0.54	0.54	0.52	82

Figure 18: Confusion Matrix (a) and Report (b) of the test with augmented dataset

After retraining the CNN with the augmented dataset, the test set results demonstrate an improved accuracy of 53.66%. The classification report indicates a balance in precision, recall, and f1-score for both classes. Specifically, class 'Cow' (0) shows a precision of 53%, recall of 73%, and an f1-score of 61%, while class 'Horse' (1) exhibits a precision of 56%, recall of 34%, and an f1-score of 42%. This contrasts with the results obtained from the CNN trained on the original dataset, where the overall accuracy was 48.78%, and the classification report highlighted challenges in predicting the minority class ('Horse'), with low precision, recall, and f1-score. The augmentation with WGAN-generated images has evidently contributed to a more balanced and improved performance in classifying both 'Cow' and 'Horse' images.

4 Conclusion

In conclusion, this project explored the synergies between Generative Adversarial Networks (GANs) and data augmentation to enhance the performance of a Convolutional Neural Network (CNN) in classifying images of cows and horses. The GAN models, including DCGAN, WGAN, and VAE-GAN, were trained to generate synthetic images, with WGAN demonstrating superior stability and image quality. The CNN, initially trained on the original dataset with limited images, faced challenges due to class imbalance. However, after augmenting the dataset with images generated by WGAN, significant improvements were observed. The retrained CNN achieved an accuracy of 53.66%, showcasing balanced precision, recall, and f1-score for both 'Cow' and 'Horse' classes. This project highlights the potential of GANs in addressing data scarcity issues and improving model generalization. In future iterations of this project, exploring hyperparameter tuning could significantly enhance model performance. Experimenting with different optimization techniques, such as adjusting learning rates or employing advanced optimization algorithms, may also yield improvements. Additionally, considering alternative model architectures or incorporating transfer learning approaches could be explored to enhance the overall capabilities of the image generation models. Evaluating the impact of larger dataset sizes and leveraging more advanced GPU resources could further contribute to refining the quality of generated images.