Engineering Department

Master's Degree Artificial Intelligence and Data Engineering

Università degli studi di Pisa
Accademic year - 2022-2023

Generative Adversarial Network with Automatic Differentiations (2023_C2)

STUDENTS:
Simone Bensi, Tommaso Nocchi

Symbolic and Evolutionary Artificial Intelligence

Contents

1 Introduction

1.1 Our Work

In this study, we aimed to replicate the success of Yoshua Bengio et al. in implementing a simplified version of a Generative Adversarial Network (GAN) using Multilayer Perceptron (MLP) networks for both the Generator and Discriminator components.

Unlike previous implementations that relied on third-party deep-learning functions, we implemented our version from scratch without using any pre-existing deep network focused libraries. We manually designed the structure of the models, including the layers and connections, and implemented both the forward and backward propagation steps.

One of the key challenges we encountered was to efficiently computing gradients in the various layers of the models during backpropagation. To overcome this, we leveraged the concept of dual numbers, which allowed us to optimize gradient calculations using automatic differentiation in forward mode. Dual numbers extended the real numbers by incorporating an infinitesimal component, enabling us to accurately compute gradients during the forward phase.

In contrast to the original implementation that employed the ADAM update algorithm for parameter optimization, we used a deterministic gradient descent approach. While ADAM is known for its effectiveness in convergence speed and robustness, we just wanted to explore the performance of a simpler optimization algorithm.

By implementing our GAN from scratch and manually handling every step of the process, we gained a deeper understanding of the inner workings of GANs, MLP networks, and the optimization process. Our study shows the feasibility of implementing GANs without relying on external deep-learning frameworks and highlights the potential benefits of utilizing automatic differentiation with dual numbers for efficient gradient calculations.

Yoshua Bengio's GAN link

1.2 Generative Adversarial Network

A GAN (Generative Adversarial Network) is a type of artificial neural network architecture that consists of two main components: a generator and a discriminator. GANs are designed to generate new data that resembles a training dataset by learning from the underlying patterns and distributions of the training data.
Here's a detailed explanation of how GANs work:

1. **Generator**: The generator is responsible for creating new data samples that mimic the training dataset. It takes random noise or a latent vector as input and transforms it into a synthetic data sample. Initially, the generator produces random and meaningless outputs, but through the training process, it gradually learns to generate realistic samples that resemble the training data.

2. **Discriminator**: The discriminator acts as a binary classifier that evaluates the generated samples and determines whether they are real (from the training data) or fake (generated by the generator). It takes a data sample as input and outputs a probability score indicating the likelihood of the input being real. The discriminator is trained using both real samples from the training dataset and fake samples from the generator.

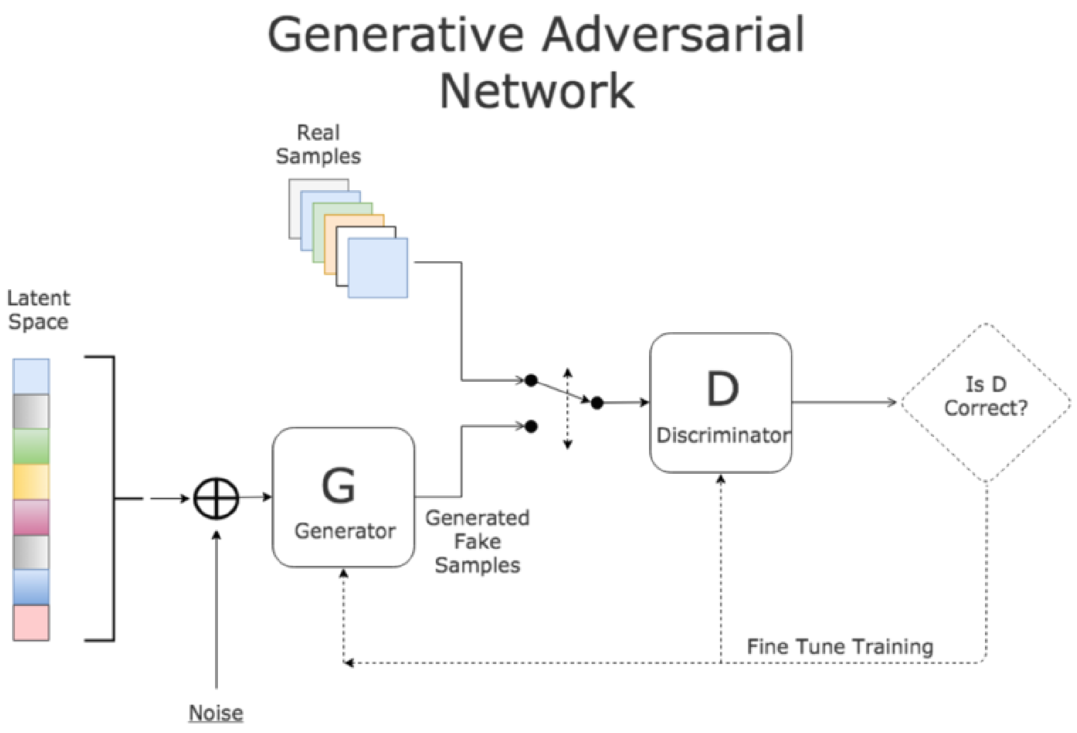


Figure 1: Architecture

The overall training process of a GAN involves a competitive interaction between the generator and the discriminator:

1. *Initialization*: the generator and the discriminator weights are initialized randomly.
2. *Training loop*: in each iteration of the training loop, the following steps are performed:
   - Generator update: the generator generates fake samples by taking random noise as input. These generated samples are passed to the discriminator.
   - Discriminator update: the discriminator receives both real samples from the training dataset and fake samples from the generator. It learns to classify them correctly by adjusting its weights to maximize the probability of assigning the correct label (real or fake) to each sample.

   The objective of the generator is to produce samples that fool the discriminator so it classifies them as real, while the objective of the discriminator is to accurately distinguish between real and fake samples. This competition between the generator and the discriminator leads to an adversarial training process.
3. Convergence: the GAN training continues until both the generator and discriminator reach a point of equilibrium, where the generator produces realistic samples that the discriminator cannot distinguish from the real data.

GANs have shown remarkable success in various domains, including image generation, video synthesis, text generation, and more. They have been used to generate realistic images, create deepfakes, enhance low-resolution images, and even generate entirely new and creative content.
It's worth noting that there are different variations and improvements to the basic GAN architecture, such as conditional GANs, Wasserstein GANs (WGANs), and progressive GANs, which introduce additional techniques to stabilize and enhance the training process.

2 GAN via MPL

A GAN (Generative Adversarial Network) can be implemented using Multilayer Perceptron (MLP) networks as the generator and the discriminator. MLPs are a type of artificial neural network architecture consisting of multiple layers of interconnected neurons.
To implement a GAN with MLP networks, we need to define the architecture and training process for both the generator and discriminator:

1. Generator: the generator network takes random noise as input and aims to transform it into synthetic data that resembles the real data. The architecture of the generator MLP typically consists of several fully connected layers followed by an output layer that produces the generated samples. The output layer's activation function depends on the data domain. For example, in image generation, the output layer may use the sigmoid or tanh activation function to produce pixel values in the range [0, 1] or [-1, 1], respectively.

2. Discriminator: the discriminator network takes as input either real data samples from the training set or generated samples from the generator. It aims to classify the input as either real or fake (generated). The discriminator MLP architecture is similar to the generator, usually comprising fully connected layers followed by a final output layer. The output layer typically uses a sigmoid activation function to produce a probability score indicating the likelihood of the input being real.

3. Training Process: the training process of the GAN involves an adversarial game between the generator and discriminator networks. The generator tries to generate synthetic samples that fool the discriminator, while the discriminator aims to correctly classify real and generated samples. In particular, this process is performed as follows:
   a) Initialization: initialize the weights of the generator and discriminator networks.
   b) For each training iteration, perform the following steps until a termination condition is met:
      i. Generate a batch of random noise as the generator's input.
      ii. Extract a real sample from the dataset.
      iii. Forward pass:
         A. Pass the noise through the generator to obtain generated samples.
         B. Feed both real and generated samples to the discriminator and obtain their predictions.
      iv. Backward pass:
         A. Calculate the loss for the discriminator based on its ability to correctly classify real and generated samples.
         B. Update the discriminator's weights using backpropagation and gradient descent technique to minimize the loss.
         C. Calculate the loss for the generator based on the discriminator's predictions of the generated samples.
         D. Update the generator's weights using backpropagation and gradient descent technique to maximize the loss (minimize the negative loss).

The training process aims to find an equilibrium where the generator produces realistic samples that the discriminator cannot distinguish from real data. This equilibrium indicates that the generator has successfully learned the underlying data distribution.
It's important to note that MLP-based GANs may face challenges in modeling complex data distributions, especially for high-dimensional and structured data like images. Convolutional Neural Networks (CNNs) are commonly used for the generator and discriminator networks in image-based GANs due to their ability to capture spatial dependencies. However, MLP-based GANs can still be effective for simpler data domains or as a starting point for GAN experimentation.

3 Min-Max GAN Loss

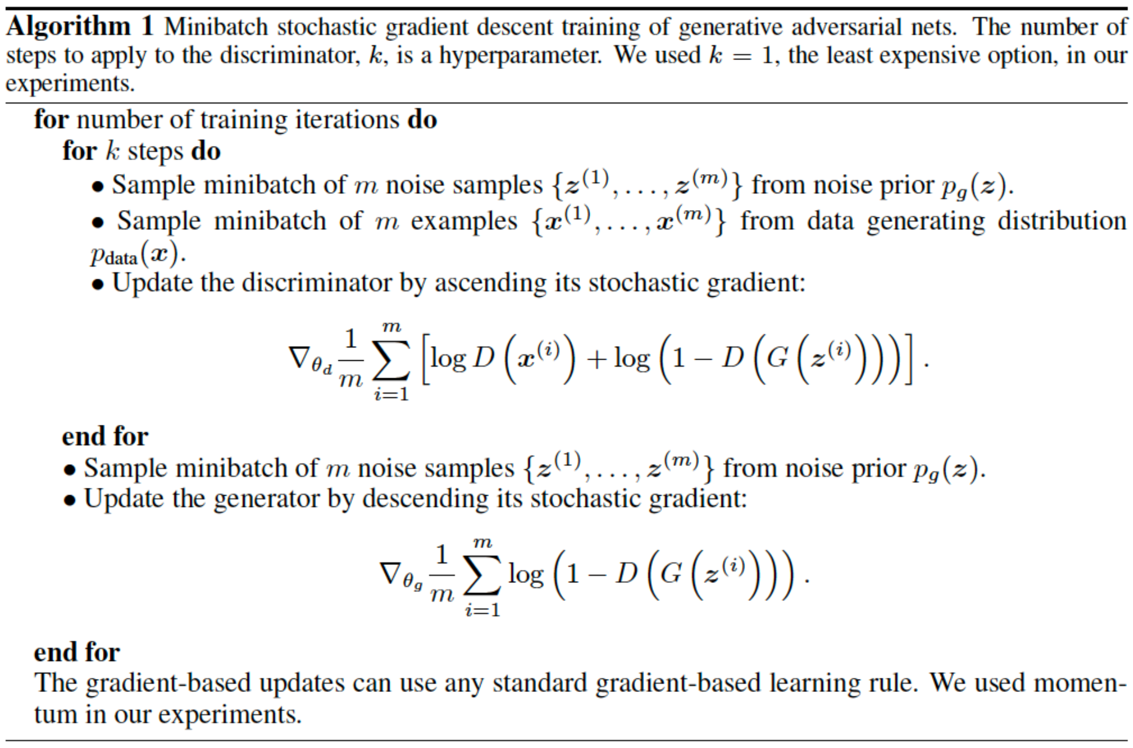
The minimax GAN loss, also known as the adversarial loss, is a fundamental component of Generative Adversarial Networks (GANs). It is used to train the generator and the discriminator networks in a GAN model. The objective of the minimax GAN loss is to guide the generator towards producing realistic samples while simultaneously training the discriminator to accurately distinguish between real and generated data.
The minimax GAN loss is based on a two-player game scenario, where the generator and discriminator compete against each other.
The loss is defined by the following equation:

$$\mathcal{L}_{\text{GAN}}(G, D) = \mathbb{E}_{\mathbf{x}\sim p_{\text{data}}(\mathbf{x})}[\log(D(\mathbf{x}))] + \mathbb{E}_{\mathbf{z}\sim p_{\mathbf{z}}(\mathbf{z})}[\log(1 - D(G(\mathbf{z})))] \quad \textbf{(1)}$$

Here, G refers to the generator network, D refers to the discriminator network, x represents a real data sample from the training set, z denotes a random noise input to the generator, and E[...] represents the expectation over the data distribution.
The first term, $E[\log(D(x))]$, represents the expectation of the logarithm of the discriminator's output when given real data samples x. The objective of this term is to **maximize** the probability that the discriminator correctly classifies real data as real. In other words, it encourages the discriminator to assign high probabilities to real data.
The second term, $E[\log(1 - D(G(z)))]$, represents the expectation of the logarithm of the discriminator's output when given generated samples G(z). Here, G(z) denotes the output of the generator when provided with random noise z. The objective of this term is to **maximize** the probability that the discriminator correctly classifies generated data as fake. The generator aims to **minimize** this second term by generating samples that the discriminator perceives as real.
The overall objective of the minimax GAN loss is to find a balance between the two competing objectives. The generator tries to minimize the discriminator's ability to distinguish between real and generated data, while the discriminator tries to maximize its ability to correctly classify the two types of data. This adversarial process drives both networks to improve iteratively.
During training, the generator and discriminator are updated alternately. The generator updates its weights to minimize the second term of the loss, while the discriminator updates its weights to maximize the loss. This back-and-forth process continues until an equilibrium is reached, ideally when the generator produces samples that are indistinguishable from real data. In practice, often an alternative termination condition is evaluated, such as a maximum number of epochs.

3.1 Our conjecture and GAN LOSS

As discussed above, our work draws inspiration from a project conducted by Yoshua Bengio et al., which successfully realized a functioning GAN using MLPs.
In their research, Yoshua Bengio et al. identified a potential limitation in the gradient provided by Equation 1, suggesting that it may not be sufficient for effective learning of the generator (G). To address this issue, they proposed an alternative training approach, rather than minimizing the second term of the equation the aim is to **maximize** $\log(D(G(z)))$. This modification was motivated by the goal of achieving stronger gradients during the early stages of the learning process.
Our project focuses on this concept, adapting it to our implementation of the GAN using MLPs. By incorporating Bengio et al.'s insights, we aim to improve the training dynamics and performance of our GAN model, enabling more effective learning and generation of realistic samples.



Algorithm 1 Minibatch stochastic gradient descent training of generative adversarial nets. The number of steps to apply to the discriminator, k, is a hyperparameter. We used $k = 1$, the least expensive option, in our experiments.

```
for number of training iterations do
    for $k$ steps do
        • Sample minibatch of $m$ noise samples $\{z^{(1)}, \dots, z^{(m)}\}$ from noise prior $p_g(z)$.
        • Sample minibatch of $m$ examples $\{x^{(1)}, \dots, x^{(m)}\}$ from data generating distribution $p_{\text{data}}(x)$.
        • Update the discriminator by ascending its stochastic gradient:
              $\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[ \log D\left(x^{(i)}\right) + \log\left(1 - D\left(G\left(z^{(i)}\right)\right)\right) \right].$
    end for
    • Sample minibatch of $m$ noise samples $\{z^{(1)}, \dots, z^{(m)}\}$ from noise prior $p_g(z)$.
    • Update the generator by descending its stochastic gradient:
          $\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log\left(1 - D\left(G\left(z^{(i)}\right)\right)\right).$
end for
The gradient-based updates can use any standard gradient-based learning rule. We used momentum in our experiments.
```

During our research study, we encountered certain challenges that required us to make informed decisions in order to carry out error backpropagation. In Algorithm 1, it is specified that we need to use the average of the loss function over the entire mini-batch, returning a scalar value.

3.1.1 Issue in the generator

The problem arises when the generator is implemented as an MLP with N output neurons, as error backpropagation requires associating an error value with each output neuron. Unlike the approach used by Yoshua Bengio et al., where they employed the dlgradient function that only requires a scalar error and the model structure to compute gradients, we faced the task of handling errors for individual output neurons. In our implementation, we adopted the loss function of the generator from Algorithm 1 without applying the averaging step. We replicated this error for each output neuron to facilitate classical error backpropagation. Subsequently, we obtained a tensor and performed averaging over the third dimension, which corresponds to the minibatch size.

3.1.2 Issue in the discriminator

In Algorithm 1, a single scalar error value is obtained for the discriminator, which does not pose a problem when using the dlgradient function for calculating gradients, as demonstrated by Yoshua Bengio et al. However, in our manual implementation of backpropagation, this created challenges due to the separate forward steps in the discriminator for real and generated images from the generator. Consequently, we had two values associated with each weight and activation of every neuron in each layer of the discriminator. To address this obstacle, we divided the loss function formula of the discriminator into two separate loss functions. This enabled us to perform backpropagation for each of the two forward steps executed within the discriminator.

These adaptations and modifications were necessary to overcome the specific challenges encountered during our study and allowed us to successfully perform error from theoretic point of view the backpropagation in our manual implementation of the GAN.

4 Automatic Differentiation

Automatic differentiation (AD), also known as algorithmic differentiation, is a technique used in computational mathematics and machine learning to efficiently compute the derivatives of mathematical functions with respect to their inputs. It provides a way to compute derivatives accurately and efficiently by exploiting the chain rule of differentiation.
The main differences between automatic differentiation, symbolic differentiation, and numerical differentiation are as follows:

1. Symbolic Differentiation: Symbolic differentiation involves manipulating mathematical expressions symbolically to obtain exact formulas for derivatives. It works by applying predefined rules of differentiation to expressions. Symbolic differentiation can handle any function as long as it can be represented symbolically. However, it can become computationally expensive and complex for functions with complicated expressions or for high-dimensional problems. It is generally suitable for functions with analytical expressions. $f(x) = \frac{\cos(x)}{x^2-1}$, the symbolic differentiation of $f(x)$ with respect to x can be represented as:

$$f'(x) = \frac{d}{dx}\left(\frac{\cos(x)}{x^2-1}\right)$$

2. Numerical Differentiation: Numerical differentiation approximates derivatives by calculating the difference quotients using finite differences. It estimates the derivative by evaluating the function at multiple points and computing the slope of the secant line between those points. Numerical differentiation is straightforward to implement and can be applied to functions without explicit formulas. However, it is prone to errors due to finite precision and numerical stability issues. It can also be computationally expensive when evaluating the function multiple times.

$$f'(x) \approx \frac{f(x+h) - f(x)}{h}$$

   where $f'(x)$ represents the derivative of f at x.

3. Automatic Differentiation: Automatic differentiation overcomes the limitations of both symbolic and numerical differentiation. It leverages the chain rule of differentiation to compute derivatives efficiently and accurately. It breaks down complex functions into elementary operations and their derivatives. By computing the derivatives of each operation, it obtains the derivative of the entire function. Automatic differentiation provides exact derivatives without approximation or truncation errors, making it more precise than numerical differentiation. It can handle complex functions with high-dimensional inputs and is especially useful in machine learning for gradient-based optimization algorithms.

Automatic differentiation can be further classified into two modes: forward mode and reverse mode.

- Forward Mode: In forward mode automatic differentiation, derivatives are computed from input variables towards the output. It computes the derivative of each elementary operation while propagating the derivatives forward through the computational graph. Forward mode AD is efficient for functions with a large number of inputs and a single output.

- Reverse Mode: In reverse mode automatic differentiation, derivatives are computed from the output variables towards the inputs. It calculates the derivative of each elementary operation while propagating the derivatives backward through the computational graph. Reverse mode AD is efficient for functions with many outputs and a smaller number of inputs.

It's important to note that for more complex functions and larger-dimensional problems, symbolic differentiation may become computationally expensive or even infeasible, while numerical differentiation may introduce approximation errors. Automatic differentiation remains accurate and efficient for a wide range of functions and dimensions.
Overall, automatic differentiation combines the accuracy of symbolic differentiation and the efficiency of numerical differentiation.

5 Dual Numbers

A dual number is a mathematical construct that extends the real numbers by introducing an additional element, typically denoted as ε (epsilon), with the property $\varepsilon^2 = 0$. It is a useful tool in mathematical analysis and automatic differentiation, as it allows for the computation of derivatives with respect to a variable.
In the dual number system, any dual number can be represented as a sum of a real part (a) and a dual part (bε), where a and b are real numbers. The real part represents the value of the number, while the dual part represents the derivative of the number with respect to a variable.
To perform arithmetic operations with dual numbers, the real parts and dual parts are handled separately. The real parts are added or multiplied following the usual rules of arithmetic, while the dual parts combine using the property $\varepsilon^2 = 0$.
Operations with dual numbers follow specific rules:

- Addition: $(a + b\varepsilon) + (c + d\varepsilon) = (a + c) + (b + d)\varepsilon$
- Subtraction: $(a + b\varepsilon) - (c + d\varepsilon) = (a - c) + (b - d)\varepsilon$
- Multiplication: $(a + b\varepsilon) \cdot (c + d\varepsilon) = (a \cdot c) + (a \cdot d + b \cdot c)\varepsilon$
- Division: $\frac{(a+b\varepsilon)}{(c+d\varepsilon)} = \left(\frac{a}{c}\right) + \left(\frac{b \cdot c - a \cdot d}{c^2}\right)\varepsilon$

Beyond those operations, it was needed to define also the absolute value of a dual number because it revealed useful during the development of out GAN. In particular, the absolute value of a number $x = a + b\varepsilon$ is defined as follows:

$$|x| = |a| + sign(a)\varepsilon$$

Note that the infinitesimal component ε squares to zero, so higher powers of ε can be ignored.

6 Experiments

6.1 Experiment: Impact of Adaptive Learning Rate on GAN Training

In this experiment, our objective was to investigate the impact of using an adaptive learning rate on training a Generative Adversarial Network (GAN) compared to a non-adaptive learning rate. We specifically focused on replacing the ADAM update function used in Yoshua Bengio et al.'s code with the Stochastic Gradient Descent function in Matlab. The aim was to assess how the use of an adaptive learning rate affected the training process and the resulting GAN performance.

Results analysis revealed that the training process was slower when using the sgdmupdate function instead of ADAM. Specifically, we observed a significant increase in the number of epochs required to generate the first discernible digits, going from 3 epochs in the original ADAM-based version to more than 10 epochs in the SGD-based version. Furthermore, throughout the entire training duration of 50 epochs, a notable decrease in accuracy was observed in comparison to the original ADAM-based implementation.

These findings indicate that the adaptive learning rate provided by ADAM, as employed by Bengio et al., plays a crucial role in accelerating the training process and enhancing the overall accuracy of the GAN. The use of a non-adaptive learning rate, such as SGD, resulted in slower convergence and reduced accuracy. These observations underscore the significance of employing adaptive learning rate algorithms like ADAM for efficient and effective training of GANs.

The experiment provides valuable insights into the impact of adaptive learning rates on GAN training, highlighting the advantages and limitations associated with different optimization methods. Future work may involve further investigation into alternative adaptive learning rate algorithms and their effects on GAN performance.

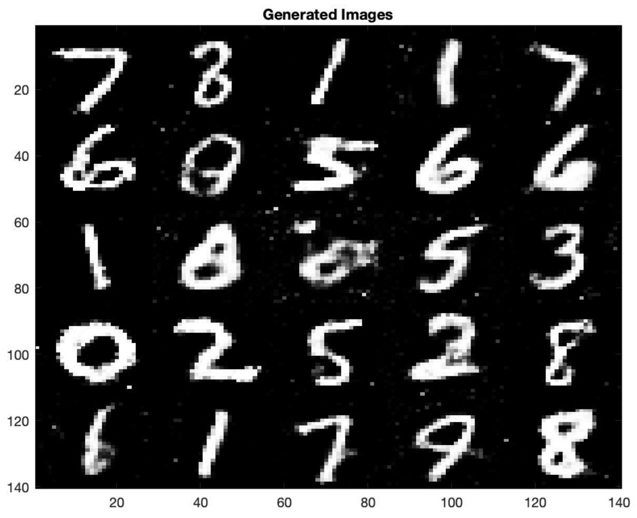


Figure 2: Bengio implementation standard implementation

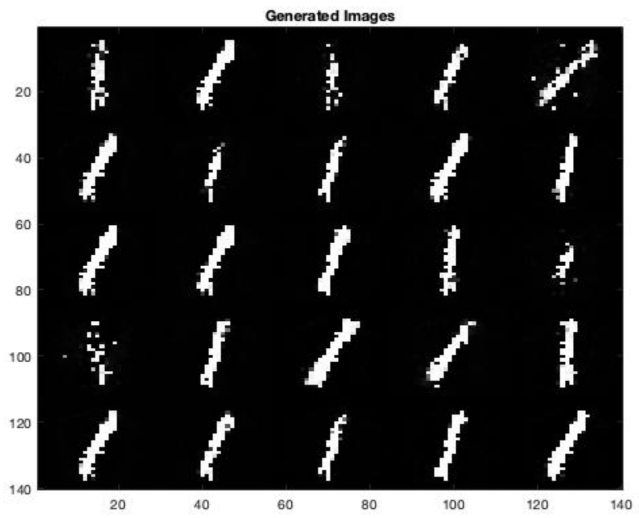


Figure 3: Bengio implementation without ADAM algorithm

6.2 Experiment: Changing the Mini Batch value to align the SGD with DGD

First of all we give a brief introduction on the differences between SGD (Stochastic Gradient Descent) and DGD (Deterministic Gradient Descent, also known as Batch Gradient Descent).

1. Sample Selection:
   - SGD randomly selects a subset of training samples (mini-batch) to compute the gradient at each iteration. The size of the mini-batch is typically small, ranging from 1 to a few hundred samples.
   - DGD uses the entire training dataset to compute the gradient at each iteration. It calculates the gradient using all available training samples.
2. Convergence:
   - SGD: due to the random nature of selecting mini-batches, SGD has a tendency to exhibit high variance in the objective function and may take longer to converge. However, the variance can be beneficial in escaping local minima and exploring different areas of the optimization landscape because choosing always a random subset od the batch we slightly change always the error surface having higher probability to escape from the local minima.
   - DGD, on the other hand, has a deterministic nature, as it uses the entire dataset to compute the gradient. It tends to have a smoother convergence and may reach the minimum of the objective function more directly. On the other hand it mantains the error surface fixed having higher probability to be caught by a local minima.
3. Computational Efficiency:
   - SGD is computationally efficient since it processes only a small subset of data at each iteration. It is well-suited for large-scale datasets as it reduces the computational burden.

- DGD requires computing the gradient using the entire dataset at each iteration, which can be computationally expensive for large datasets. It may be more suitable for smaller datasets or problems where computational resources are not a limiting factor.

In order to compare even better our work with the one of Yousha Bengio, after changing the optimization algorithm from ADAM update tu SGD update, since our implementation employs the deterministic GD, we decided to move the mini-batch value to 1.
When the mini-batch size is equal to one, Stochastic Gradient Descent (SGD) is essentially equivalent to a variant of Gradient Descent (GD) called "Batch Gradient Descent" (BGD).
In both GD and BGD, the gradients are computed by evaluating the cost function over the entire dataset. The difference lies in the parameter update step, where GD uses the average gradient over the entire dataset, while BGD updates the parameters using the gradient computed from a single batch of examples. BGD provides a compromise between accuracy and computational efficiency, making it a commonly used optimization algorithm in machine learning.

7 GAN structure

As said, a GAN is essentially composed of 2 nets, the generator and the discriminator both implemented via MLP. We can see below the structure of a general MLP network.

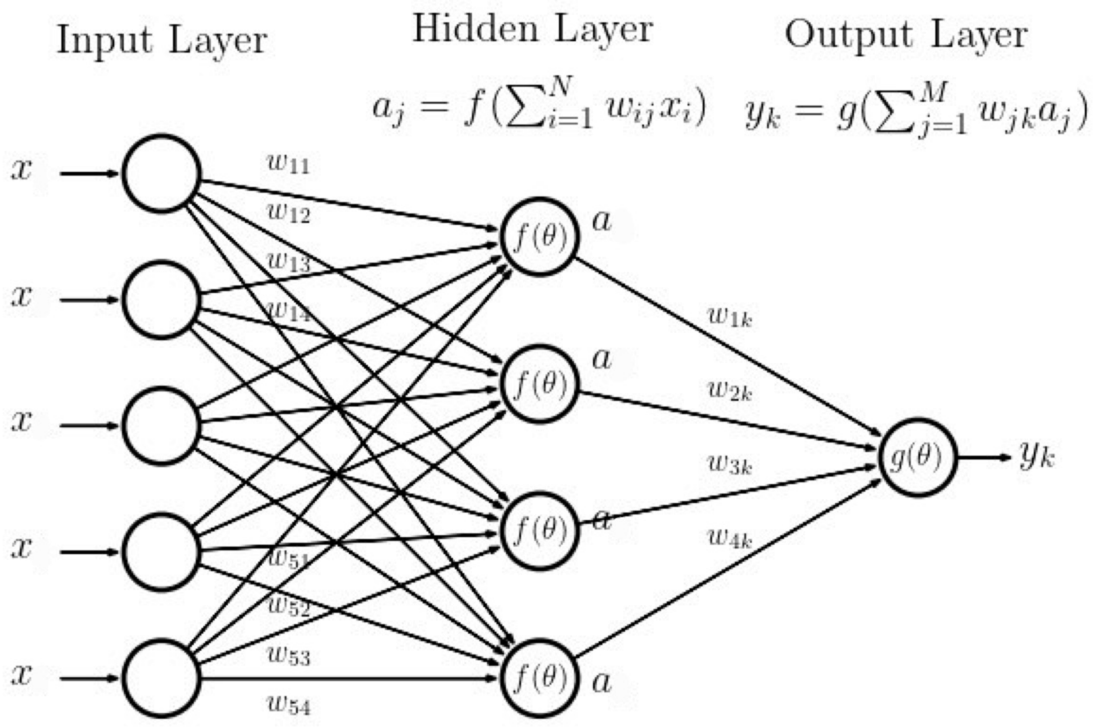


Figure 4: MLP structure

In our code we referred to sum of the product of weights and inputs with the letter ***z***, the output of the neuron with the letter ***a***. Each letter in our code is followed by a number which indicates the layer. Furthermore, we implemented those two MLP networks through the use of Dual Numbers. To do this, we exploited an existent class which was tailored on vectors, so we had to modify it to handle matrices of dual numbers in the case the batch_size is higher than 1.

7.1 Generator structure

The generator is implemented as an MLP network composed by one input layer, 3 hidden layers and one output layer, which contain 100, 256, 512, 1024 and 784 neurons respectively. As in a standard MLP network, those layers are linked by RxC weight matrices, where R is the number of neurons of the next layer and C is the number of neurons of the previous layer plus a bias unit. Those weights are randomly initialized drawing from a normal distribution, while the biases are initialized to zeros vectors. In particular, this network takes as input a batch of noisy data to generate a fake image. This input has size 100 x batch_size.
The following steps are then performed until the maximum number of epochs (50) has been reached.

7.1.1 Forward pass

Considering as input data the k x batch_size output matrix of the previous layer (where k is the number of neurons of the previous layer) or the noisy data in the case of the first layer, at the i-th layer the following operations are performed:

1. A bias unit is added as last row of the input matrix **real_a$_i$**, obtaining the k+1 x batch_size matrix **real_a$_i$**.
2. The *i*-th matrix of weights **w$_i$** are multiplied by **real_a$_i$**, obtaining a p x batch_size matrix z_i, where p is the number of neurons of the *(i+1)*-th layer.
3. The **z$_i$** matrix is passed to the activation function of the *i*-th layer, obtaining the matrix **a$_i$** having the same size but containing dual numbers.
4. From the **a$_i$** matrix is extracted the matrix **grad_a$_i$** containing the dual part of the dual numbers.
5. From the **a$_i$** matrix is extracted the matrix **real_a$_i$** containing the real part of the dual numbers.

The real output matrix **real_a$_5$** of the last layer is the actual generated fake image. All the other real and dual matrices **real_a$_i$** and **grad_a$_i$** resulting as output of each layer will be used during the backpropagation procedure.

```
% hidden layer 1
real_a1 = add_bias_unit(noise, 'row'); % real_a1: 101 x batch_size
z2 = generator_model.w1*real_a1; % z2: 256 x batch_size
a2 = leakyReLU(z2, 0.2);
grad_a2 = getDual(a2);
real_a2 = getReal(a2); % real_a2: 256 x batch_size

% hidden layer 2
real_a2 = add_bias_unit(real_a2, 'row'); % real_a2: 257 x batch_size
z3 = generator_model.w2*real_a2; % z3: 512 x batch_size
a3 = leakyReLU(z3, 0.2);
grad_a3 = getDual(a3);
real_a3 = getReal(a3); % real_a3: 512 x batch_size

% hidden layer 3
real_a3 = add_bias_unit(real_a3, 'row'); % real_a3: 513 x batch_size
z4 = generator_model.w3*real_a3; % z4: 1024 x batch_size
a4 = leakyReLU(z4, 0.2);
grad_a4 = getDual(a4);
real_a4 = getReal(a4); % real_a4: 1024 x batch_size

% output layer
real_a4 = add_bias_unit(real_a4, 'row'); % real_a4: 1025 x batch_size
z5 = generator_model.w4*real_a4; % z5: 784 x batch_size
z5 = DualArray(z5,ones(size(z5)));
a5 = tanh(z5);
grad_a5 = getDual(a5);
real_a5 = getReal(a5); % real_a5: 784 x batch_size
```

Figure 5: Generator forward pass Matlab implementation

7.1.2 Loss function

As loss function for the generator we used the following formula:

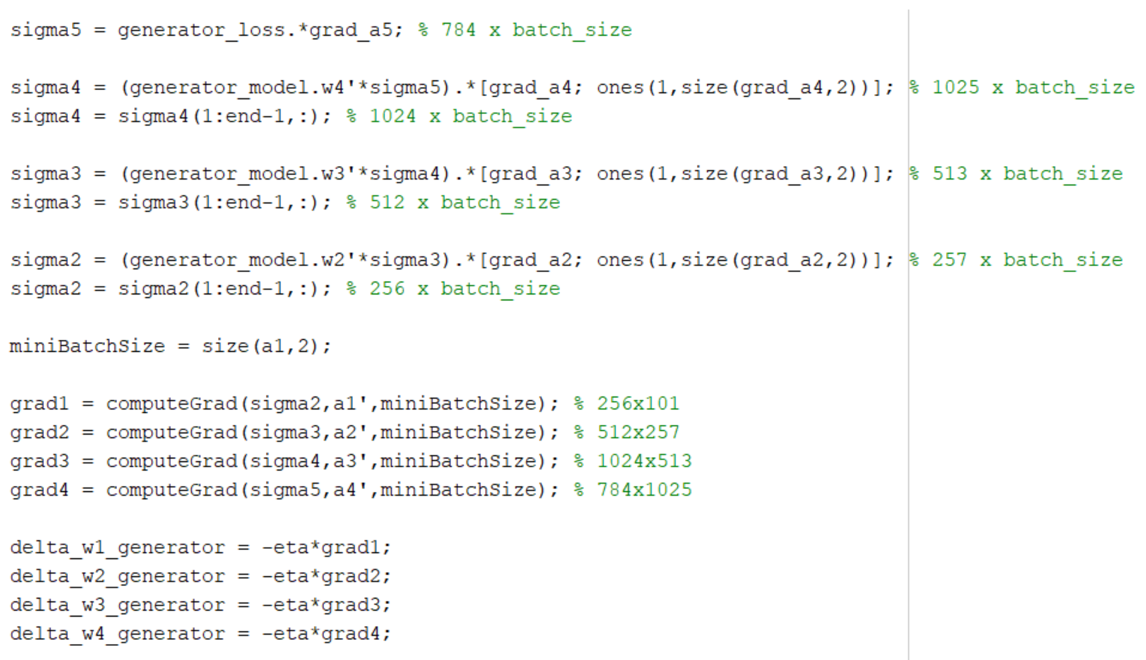
$$L_{\text{generator}} = \log(output\_fake)$$

where output_fake is the output given by the discriminator when receiving as input a fake image. Indeed, the aim of the generator is to fool the discriminator, so it wants to get as output from the discriminator a value as close as possible to 1, so that the logarithm tends to zero. Therefore, the generator goal is to maximise the loss.

7.1.3 Backward pass

This step is computed to get a delta matrix for each weights matrix, in order to update them. Those delta matrices are computed through the following procedure:

1. Multiply in a element-wise fashion $\mathbf{L_{generator}}$ and **grad_a$_5$** obtaining **sigma$_5$**.
2. For each layer *i* starting from the third hidden layer towards the first hidden layer:
   a) multiply the weights matrix $\mathbf{w_i}$ by **grad_a$_{i+1}$**.
   b) multiply in a element-wise fashion the previous step output matrix by the matrix composed by the first row of values equal to 1 to take into account the bias and **grad_a$_i$** saved during the forward pass.
   c) from **sigma$_{i+1}$** and the transposition of $\mathbf{a_i}$, build the **grad$_i$** matrix in the following way:
      i. for each k=1:batch_size, compute a matrix obtained multiplying the *k*-th column of **sigma$_{i+1}$** and *k*-th row of $\mathbf{a_i}$ transposed.
      ii. compute the matrix having for each cell *(x,y)* the mean of the batch_size cells having position *(x,y)* in the batch_size matrices computed at the previous step
   d) multiply **grad$_i$** by the learning rate η, with a minus sign because we have to go in the opposite direction with regard to the error.

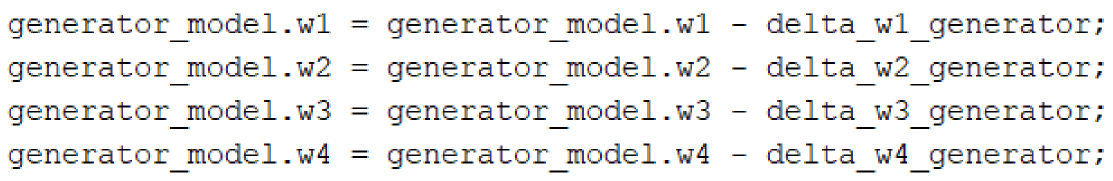


```
sigma5 = generator_loss.*grad_a5; % 784 x batch_size

sigma4 = (generator_model.w4'*sigma5).*[grad_a4; ones(1,size(grad_a4,2))]; % 1025 x batch_size
sigma4 = sigma4(1:end-1,:); % 1024 x batch_size

sigma3 = (generator_model.w3'*sigma4).*[grad_a3; ones(1,size(grad_a3,2))]; % 513 x batch_size
sigma3 = sigma3(1:end-1,:); % 512 x batch_size

sigma2 = (generator_model.w2'*sigma3).*[grad_a2; ones(1,size(grad_a2,2))]; % 257 x batch_size
sigma2 = sigma2(1:end-1,:); % 256 x batch_size

miniBatchSize = size(a1,2);

grad1 = computeGrad(sigma2,a1',miniBatchSize); % 256x101
grad2 = computeGrad(sigma3,a2',miniBatchSize); % 512x257
grad3 = computeGrad(sigma4,a3',miniBatchSize); % 1024x513
grad4 = computeGrad(sigma5,a4',miniBatchSize); % 784x1025

delta_w1_generator = -eta*grad1;
delta_w2_generator = -eta*grad2;
delta_w3_generator = -eta*grad3;
delta_w4_generator = -eta*grad4;
```

Figure 6: Generator backward pass Matlab implementation

7.1.4 Generator weights update

The generator weights are then updated by simply subtracting the previously computed delta from the current weights.



```
generator_model.w1 = generator_model.w1 - delta_w1_generator;
generator_model.w2 = generator_model.w2 - delta_w2_generator;
generator_model.w3 = generator_model.w3 - delta_w3_generator;
generator_model.w4 = generator_model.w4 - delta_w4_generator;
```

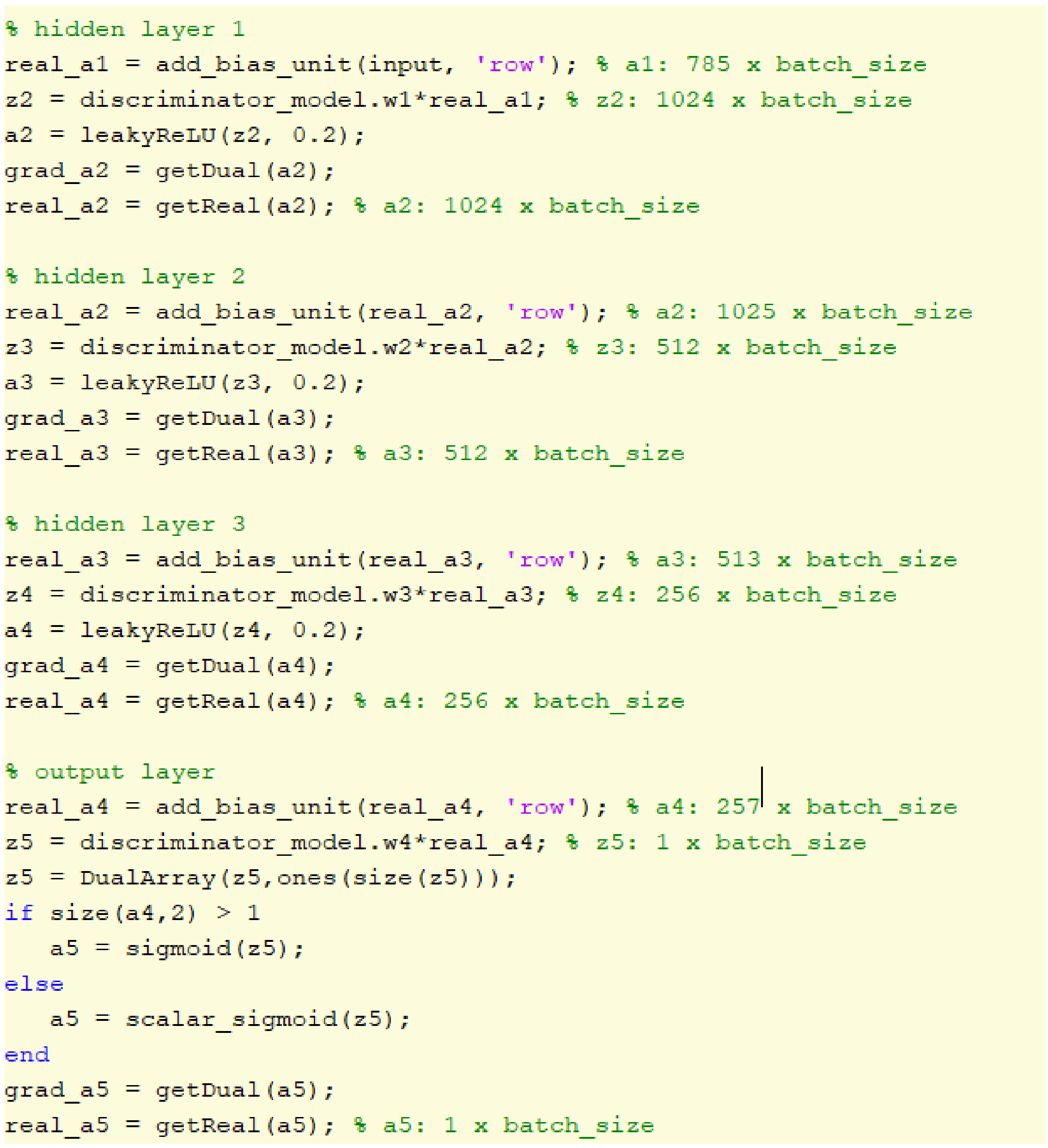
Figure 7: Generator weights update

7.2 Discriminator structure

The discriminator has a symmetric structure with regard to the generator. It's implemented as an MLP network composed by one input layer, 3 hidden layers and one output layer, which contain 784, 1024, 512, 256 and 1 neurons respectively. Also in this case, those layers are linked by RxC weight matrices, where R is the number of neurons of the next layer and C is the number of neurons of the previous layer plus a bias unit. Those weights are randomly initialized drawing from a normal distribution, while the biases are initialized to zeros vectors. This network takes as input a batch of images (which can be either real or fake) and returns for each image a real value proportional to the likelihood the image is real. Each of the image of the batch given as input to this network has 2 dimensions having each size 28, so the total size of the input matrix is 784 x batch_size. The following steps are then performed until the maximum number of epochs (50) has been reached.

7.2.1 Forward pass

1. A bias unit is added as last row of the input matrix, obtaining a k+1 x batch_size.
2. The i-th matrix of weights are multiplied by the previous step resulting matrix, obtaining a p x batch_size matrix, where p is the number of neurons of the next layer.
3. The previous step resulting matrix is passed to the activation function of the i-th layer, obtaining a matrix with the same size but containing dual numbers.
4. From the previous output matrix is extracted the matrix containing the dual part of the dual numbers.
5. From the 3-rd step output matrix is extracted the matrix containing the real part of the dual numbers.



```
% hidden layer 1
real_a1 = add_bias_unit(input, 'row'); % a1: 785 x batch_size
z2 = discriminator_model.w1*real_a1; % z2: 1024 x batch_size
a2 = leakyReLU(z2, 0.2);
grad_a2 = getDual(a2);
real_a2 = getReal(a2); % a2: 1024 x batch_size

% hidden layer 2
real_a2 = add_bias_unit(real_a2, 'row'); % a2: 1025 x batch_size
z3 = discriminator_model.w2*real_a2; % z3: 512 x batch_size
a3 = leakyReLU(z3, 0.2);
grad_a3 = getDual(a3);
real_a3 = getReal(a3); % a3: 512 x batch_size

% hidden layer 3
real_a3 = add_bias_unit(real_a3, 'row'); % a3: 513 x batch_size
z4 = discriminator_model.w3*real_a3; % z4: 256 x batch_size
a4 = leakyReLU(z4, 0.2);
grad_a4 = getDual(a4);
real_a4 = getReal(a4); % a4: 256 x batch_size

% output layer
real_a4 = add_bias_unit(real_a4, 'row'); % a4: 257 x batch_size
z5 = discriminator_model.w4*real_a4; % z5: 1 x batch_size
z5 = DualArray(z5,ones(size(z5)));
if size(a4,2) > 1
   a5 = sigmoid(z5);
else
   a5 = scalar_sigmoid(z5);
end
grad_a5 = getDual(a5);
real_a5 = getReal(a5); % a5: 1 x batch_size
```

Figure 8: Discriminator forward pass Matlab implementation

7.2.2 Loss function

As already said, in each GAN iteration the discriminator runs two times, one to evaluate a real image and one to evaluate the fake image produced by the generator. This means that there will be two forward passes and two backward passes. In particular, each backward pass starts from a specific loss function, which are specified below:

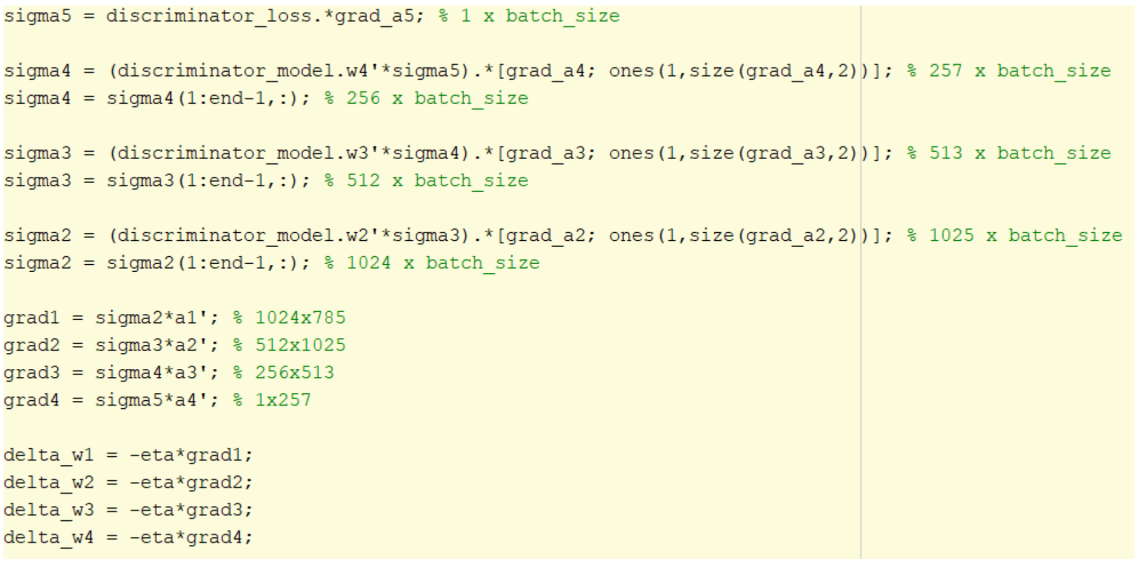
$$L_{\text{discriminator\_fake}} = mean(log(1 - output\_fake))$$

$$L_{\text{discriminator\_real}} = mean(log(output\_real))$$

where output_fake is the output given by the discriminator when receiving as input a fake image, while output_real is the output given by the discriminator when receiving as input a fake image. Indeed, the aim of the discriminator is to minimize the output_fake, and so to maximise the loss. In the case of the real image given as input the goal of the discriminator is to maximize the output_real, and therefore to maximise the loss.

7.2.3 Backward pass

This step is computed to get a delta matrix for each weights matrix, in order to update them. Those delta matrices are computed through the same procedure used for the generator.

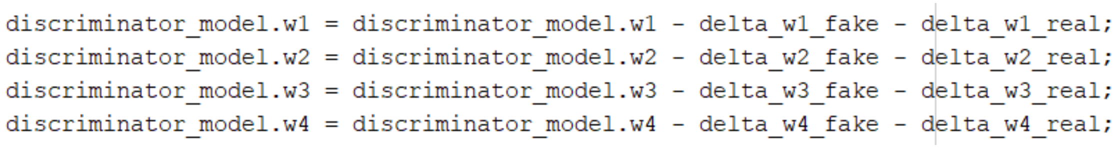


```
sigma5 = discriminator_loss.*grad_a5; % 1 x batch_size

sigma4 = (discriminator_model.w4'*sigma5).*[grad_a4; ones(1,size(grad_a4,2))]; % 257 x batch_size
sigma4 = sigma4(1:end-1,:); % 256 x batch_size

sigma3 = (discriminator_model.w3'*sigma4).*[grad_a3; ones(1,size(grad_a3,2))]; % 513 x batch_size
sigma3 = sigma3(1:end-1,:); % 512 x batch_size

sigma2 = (discriminator_model.w2'*sigma3).*[grad_a2; ones(1,size(grad_a2,2))]; % 1025 x batch_size
sigma2 = sigma2(1:end-1,:); % 1024 x batch_size

grad1 = sigma2*a1'; % 1024x785
grad2 = sigma3*a2'; % 512x1025
grad3 = sigma4*a3'; % 256x513
grad4 = sigma5*a4'; % 1x257

delta_w1 = -eta*grad1;
delta_w2 = -eta*grad2;
delta_w3 = -eta*grad3;
delta_w4 = -eta*grad4;
```

Figure 9: Discriminator backward pass Matlab implementation

7.2.4 Weights update

The discriminator weights are updated in the same way with regard to the generator weights.



```
discriminator_model.w1 = discriminator_model.w1 - delta_w1_fake - delta_w1_real;
discriminator_model.w2 = discriminator_model.w2 - delta_w2_fake - delta_w2_real;
discriminator_model.w3 = discriminator_model.w3 - delta_w3_fake - delta_w3_real;
discriminator_model.w4 = discriminator_model.w4 - delta_w4_fake - delta_w4_real;
```

Figure 10: Discriminator weights update

7.3 Modifications to use Dual Numbers

During the forward pass of the discriminator, in the output layer the sigmoid has been used as activation function, whose mathematical formulation is shown below.

$$S(x) = \frac{1}{1+e^{-x}}$$

However, using the standard sigmoid function we have to compute the exponential of a Dual Number which is not straightforward to implement. Therefore, we introduced an approximation that employs only the arithmetic operations (+,-,*,/). Its mathematical formulation is shown below.

$$S_{\text{approximated}}(x) = \frac{1}{2}(\frac{x}{1+|x|} + 1)$$

We compared the outputs of those two versions to understand whether we could use the approximated version (orange curve) in place of the standard one (blue curve).

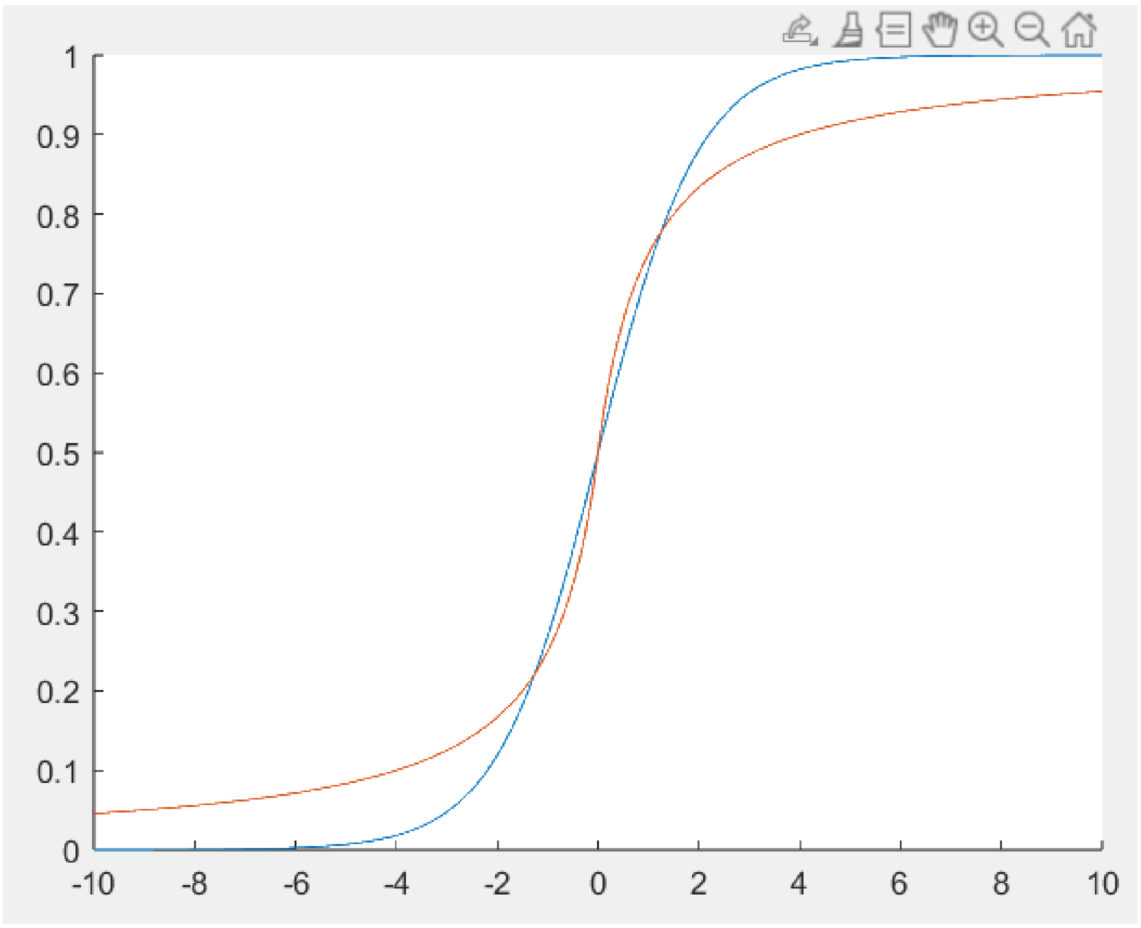


Figure 11: Comparison of sigmoids

We can see that the approximated function is very similar to the standard one, so we can use it, obtaining a very important simplification.
The same reasoning has been performed for the tanh function, which has been used in the output layer of the generator. The mathematical formulation of this function is the following.

$$tanh(x) = \frac{e^{2x} - 1}{e^{2x} + 1}$$

However, as well as for the sigmoid function, using the standard tanh function we have to compute the exponential of a Dual Number which is not straightforward to implement. Therefore, also in this case we introduced an approximation that employs only the arithmetic operations (+,-,*,/). Its mathematical formulation is shown below.

$$tanh_{\text{approximated}}(x) = \frac{1}{1+|2x|}$$

We compared the outputs of those two versions to understand whether we could use the approximated version (orange curve) in place of the standard one (blue curve).

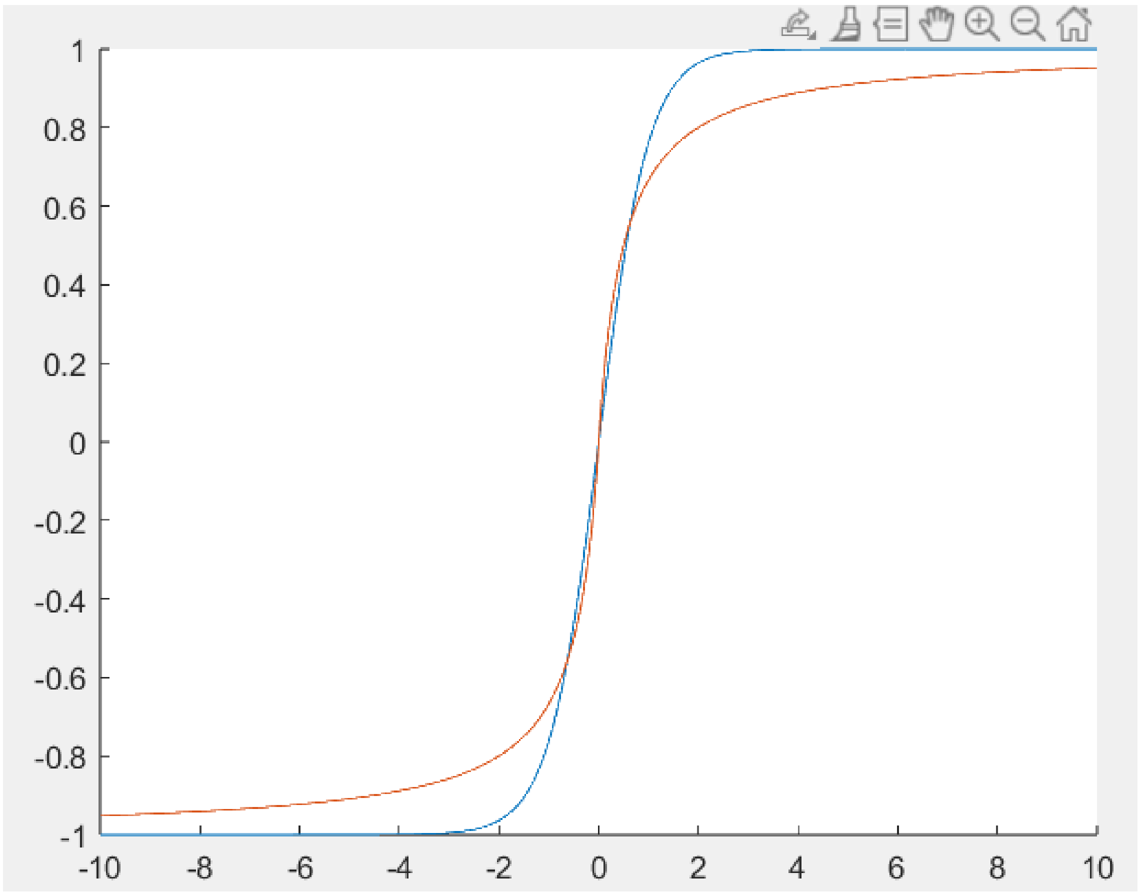


Figure 12: Comparison of tanh

Also in this case we can see that the approximated function is very similar to the standard one.

8 Conclusion

Our implementation of the Generative Adversarial Network (GAN) via Multi-Layer Perceptron (MLP) was inspired by the pioneering work of Bengio and his team. However, despite following the same architectural structure and utilizing similar initialization techniques, we encountered challenges that prevented us from achieving concrete results in generating realistic images resembling those from the training set. In particular, we observed a dramatically slower process, that took about 12 hours to finish a single epoch, while for the original implementation it took about 31 minutes. This involves that we didn't get any tangible results, so we didn't know if our architecture performed good enough in terms of accuracy, we only can conclude that after one single epoch it produced a much different output compared to the reference one. Below we can see in the first figure the output we obtained after the execution of one epoch, compared to the output that the Bengio's model got (in the second figure).

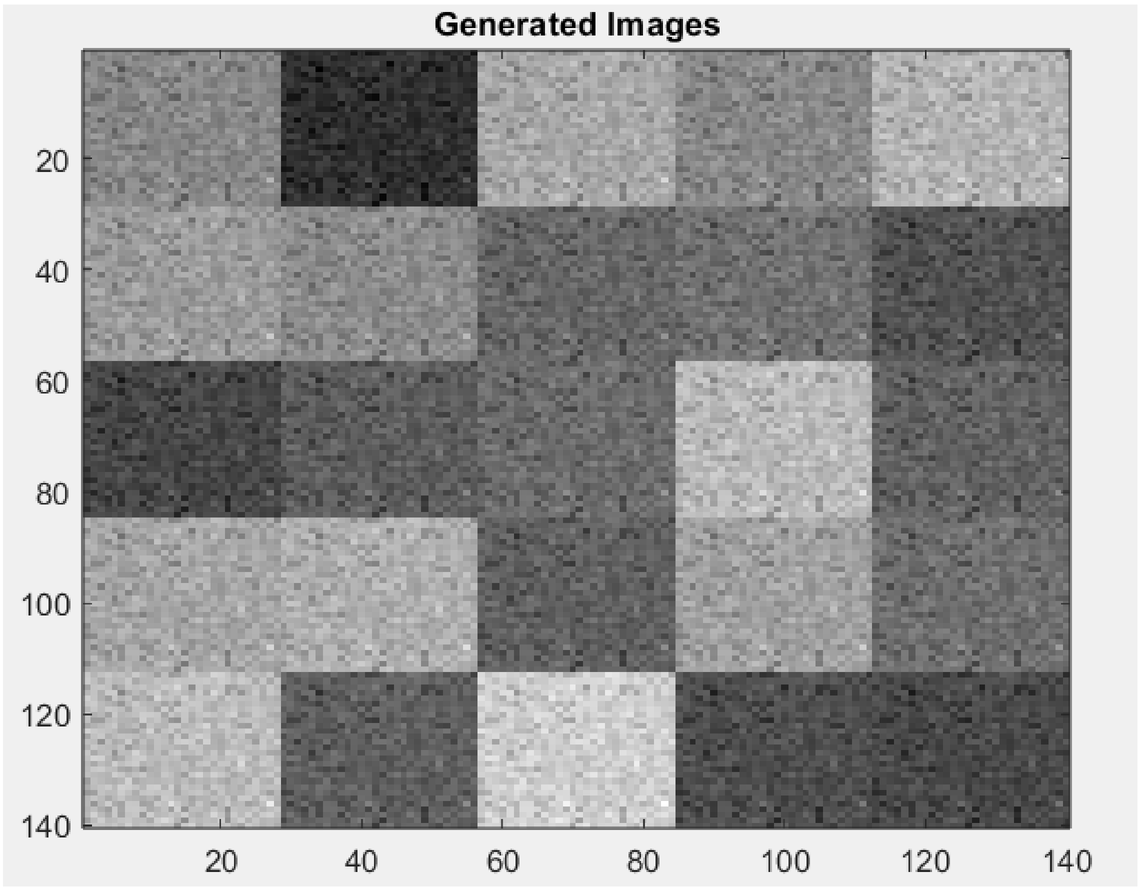


Figure 13: Our model output after one epoch

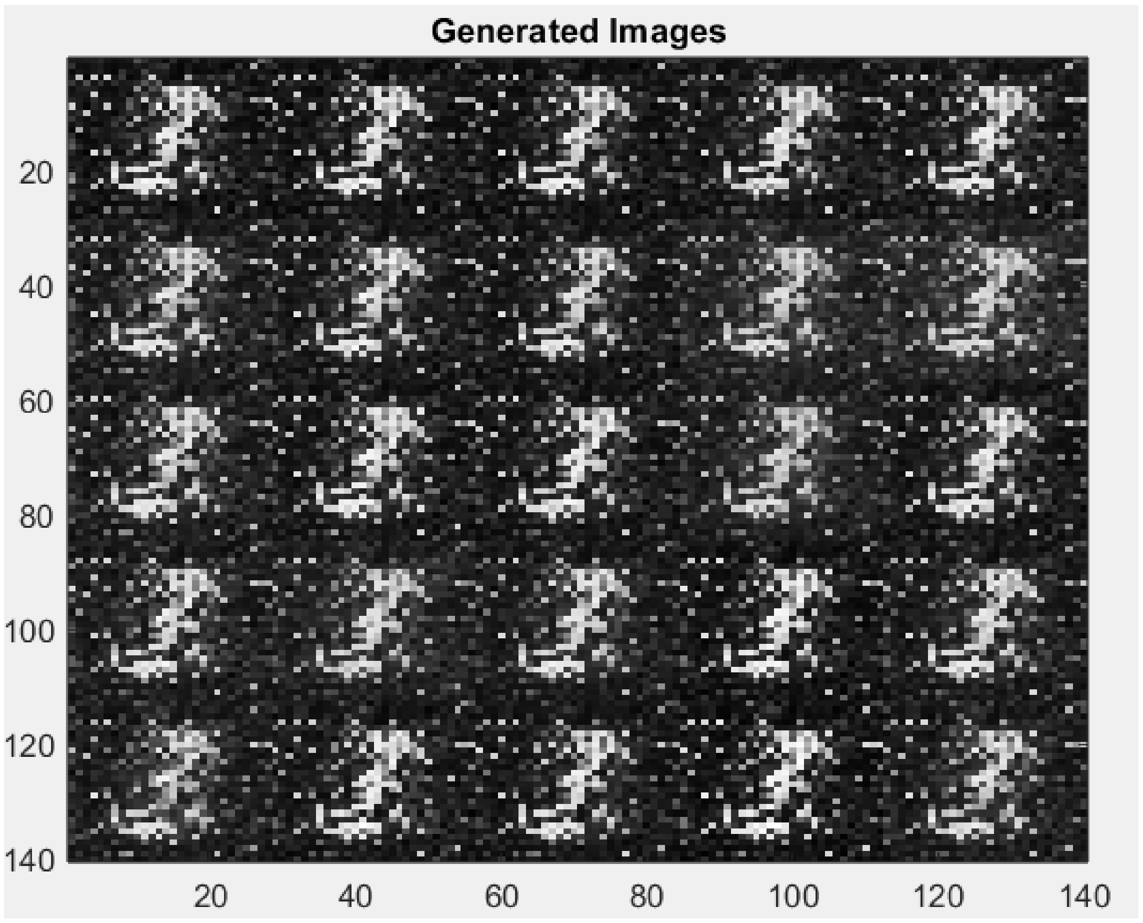


Figure 14: Bengio's model output after one epoch

Our aim is to elucidate the reasons behind this outcome and highlight the factors that contributed to the divergence between our model and Bengio's.

One significant aspect that distinguished our implementation was our decision to construct and train the GAN manually, without relying on external libraries or pre-built functions specifically designed for GAN training. While this approach allowed us to have full control over each step of the process, it also introduced a higher degree of complexity and room for error. Consequently, we encountered difficulties in accurately handling the loss function and performing backpropagation, which are crucial stages in training the GAN.

One particular challenge we faced was associated with the generator G. In Bengio's implementation, the loss function and error backpropagation were handled seamlessly, as they used a deep learning library with built-in functions capable of handling the required computations. However, in our case, we needed to replicate the error for each output neuron of the generator G manually. This approximation might have introduced discrepancies and hindered the generator's ability to learn effectively.

Additionally, we encountered challenges related to the discriminator D. In Bengio's model, the loss function for the discriminator was well-defined and suitable for backpropagation using the library's functions. However, in our manual implementation, the loss function of the discriminator had to be broken down into two separate functions to accommodate the two forward steps performed on real and generated images. This deviation from the original formulation could have had unintended consequences and affected the overall training process.

Despite these challenges, we found that the initial forward phase of our model and Bengio's model produced identical results for each neuron of both the generator G and the discriminator D. This observation, along with the fact that we used a default seed for random initialization of the weights in both works, suggests that the divergence between our model and Bengio's model can be attributed to the handling of the loss function and the subsequent backpropagation steps.

In conclusion, while we attempted to implement the GAN via MLP following Bengio's work, the decision to adopt a manual procedure without utilizing external library functions hindered our ability to generate realistic images comparable to those in the training set in a reasonable time. The discrepancies in the weights of our model compared to Bengio's, which initially showed promising similarity, can be attributed to our approximations and guesswork during the handling of the loss function and backpropagation phases. These challenges highlight the importance of utilizing established libraries and functions specifically designed for GAN training to ensure more reliable and accurate results.