Hybrid supervised learning: autoencoders, CNNs and GANs

Federico Banoy Restrepo, *Member, IEEE,*
Juan David Rengifo-Castro, *Member, IEEE,*
Salomón Cardeño Luján, *Member, IEEE,*

Abstract—When facing a machine learning problem in a real-world context, it is important to acknowledge that the process of learning becomes a part of solving the problem itself and not an axiomatic assumption to be blindly ignored. It is the task of the modeler to preserve this idea when designing and constructing actual intelligent systems. With this in mind, this study aimed to explore the different characteristics of working with autoencoders, convolutional neural networks (CNNs), and generative adversarial networks (GANs). Through a series of experiments, we aimed to demonstrate the effectiveness of these networks in various scenarios. Our findings suggest that each network type exhibits distinct strengths and weaknesses, and the selection of a suitable network is highly dependent on the specific application requirements. Furthermore, our study highlights the importance of careful consideration and thorough evaluation when utilizing machine learning techniques in real-world contexts.

Index Terms—Machine Learning, Intelligent Systems, Autoencoders, CNNs, GANs.

1 Introduction

Machine learning models such as autoencoders, CNNs, and GANs have shown great promise in solving complex tasks that were previously considered unattainable. However, we must be careful not to conflate the ability to learn from data with human-like intelligence. Machine learning models are only as good as the data and algorithms used to train them, and they can struggle in new and unanticipated situations. In this study, our focus is on exploring the unique characteristics of autoencoders, CNNs, and GANs in specific applications, to better understand how these models can be leveraged to solve real-world problems.

Autoencoders have a rich history in the field of unsupervised learning. First introduced in the 1980s as a type of neural network model for dimensionality reduction and feature extraction [1], autoencoders did not gain widespread attention until the early 2000s, when they were applied to various image and signal processing tasks [2]. Since then, with the advent of deep learning and the availability of large datasets and powerful computing resources [3], autoencoders have become increasingly popular. They are now widely used for unsupervised learning tasks such as dimensionality reduction, data compression, and anomaly detection. The basic idea behind autoencoders is to learn a compressed representation, or encoding, of a set of input data by mapping it to a lower-dimensional space [4]. The model then attempts to reconstruct the original data from this compressed representation using a decoder network. Autoencoders have been applied to a wide range of domains, including computer vision, natural language processing, and recommendation systems [2], and are effective in capturing the underlying structure of the input data, while also being robust to noise and other forms of variability [3]. Many advanced architectures, such as convolutional and recurrent autoencoders, have been developed to handle more complex data types and structures [2].

Convolutional Neural Networks (CNNs) have revolutionized the field of computer vision since their introduction in the 1980s [2]. However, it was not until the development of the LeNet-5 architecture in 1998 by Yann LeCun and his colleagues that CNNs gained widespread attention [5]. LeNet-5 was designed for handwritten digit recognition and used convolutional layers to extract features from the input image, followed by fully connected layers for classification. Since then, CNNs have become the state-of-the-art approach for various computer vision tasks, including image classification, object detection, and segmentation; they have also been successfully applied to natural language processing and speech recognition tasks [2]. One of the key advantages of CNNs is their ability to automatically learn and extract relevant features from the input data, which eliminates the need for manual feature engineering. With the availability of large datasets and the development of more advanced architectures, such as ResNet and VGG, CNNs continue to achieve impressive results in various domains [2].

Generative Adversarial Networks (GANs) are a class of deep learning models that have revolutionized the field of generative modeling. Introduced in 2014 by Ian Goodfellow [6], GANs have since become one of the most popular and widely used generative models due to their ability to generate high-quality synthetic data that is often indistinguishable from real data. GANs consist of two neural networks, a generator and a discriminator, which are trained together in a game-like fashion. The generator creates fake

- *F. Banoy Restrepo was with the universidad EAFIT, Medellín, Colombia. E-mail: fbanoyr@eafit.edu.co.*
- *J.D. Rengifo Castro was with the universidad EAFIT, Medellín, Colombia. E-mail: jdrengifoc@eafit.edu.co.*

Manuscript received February 19, 2023; revised February 31, 2023.

- *S. Cardeño Luján was with the universidad EAFIT, Medellín, Colombia. E-mail: scardenol@eafit.edu.co.*

data samples, while the discriminator learns to differentiate between fake and real data. As the generator gets better at creating realistic data, the discriminator gets better at identifying fake data, resulting in a continuous improvement of the overall GAN model. This adversarial training process has enabled GANs to achieve remarkable results in various applications, including image and video generation, text generation, and even music synthesis.

Moving forward, we will present the methodology used in our study, including the dataset, network architecture, training process, and evaluation metrics for the application of autoencoders, CNNs, and GANs. We will then present our results and analyses, comparing the performance of different models and discussing their implications. Finally, we will provide our conclusions in contrast to our findings.

2 METHODS

2.1 Autoencoders

Before introducing formally the concept of autoencoder, is worth noting what is a neural network (NN). A NN is the union of finite sequential layers of perceptrons (or neurons). Each layer is composed of a finite number of perceptrons that receive as input the data or the output of a previous layer, then combine linearly the received stimulus, and possibly transform the dimension with a weight matrix. There are three kinds of layers: the input layer (receives the input data), the output layer (provides the response/output signal), and the hidden layers (all the layers between the input and the output layer).

Usually, the NNs are employed to model a set of variables that are assumed to be dependent on the input data. Nevertheless, it is possible to model the input data as a function of itself. Initially, this could seem useless, although it allows making dimensionality reduction and feature extraction. The former can be achieved by employing fewer perceptrons than features in a unique hidden layer, while the latter requires employing more perceptrons than features in the hidden layer. As shown by the figure 1, the hidden layer receives the name of *code*, the input layer *encodes* the input data, and the output layer *decodes* the code.

2.1.1 Geographical Original of Music Data Set

To experiment with autoencoders we employed the geographical original of music data set (GOM) gathered by [7]. The GOM dataset was created using a personal selection of 1059 traditional, ethnic, or "world" tracks from 33 different countries. For each track, 68 features were extracted from the wavelet file by the Marsyas, a framework for audio analysis proposed by [8]. The data was normalized in the unit interval by mapping each column with the function presented in the equation 1.

$$x_i = \frac{x_i - \min x_i}{\max x_i - \min x_i} \tag{1}$$

2.1.2 Towards the Best Model

In the urge to have a comprehensive understanding of autoencoders we elaborated a mixed model. The first part of the model is a autoenconder (elaborated from scratch) and its code feeds a NN (elaborated with tensorflow) to model

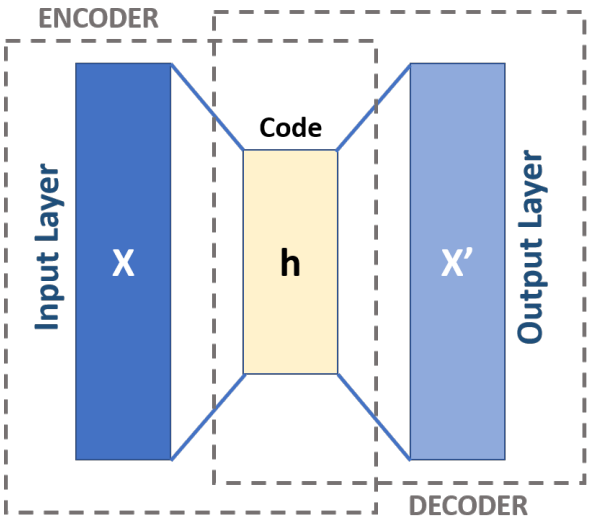

Fig. 1: Autoencoder architecture. Link to source.

the output data of the GOM dataset. The above was made for both, a dimensional reduction and a feature extraction autoencoder ($\mathcal{M}_1$ and $\mathcal{M}_2$ accordingly).

The GOM data was randomly divided according to a uniform distribution in three sets: training (60%), validation (20%) and test (20%). In the urge to find a proper NN to model the outputs (latitude and longitude coordinates of the track's origin country), we train $5 \times 5 \times 7 = 175$ neural networks, by combining $L = 1, ..., 5$ hidden layers with $l_1 = l_2 = l_3 \in \{1, ..., 5\}$ neurons per hidden layer and the learning rates $\eta \in \{0.0001, 0.001, 0.01, 0.1, 0.2, 0.5, 0.9\}$.

The training has a stop criteria a maximum of 100 epochs or that the average error is less than the tolerance of 0.01. Despite the stabilization of the gradient could be another stop criteria, this was unnecessary since the training was low time-demanding. The activation for all the layers was fixed as a sigmoid function and the weights were initialized from a continuous uniform distribution in the interval $[-1, 1]$. Once the training was done by gradient descendent, we selected as the best NN, the one with the lowest loss (average instant energy) in the *validation* data.

The above optimization process was also made for the selection of the optimal autoenconders, with the difference that we only consider one hidden layer and the number of neurons on it ranges from 1 to 67 for $\mathcal{M}_1$ and from 69 to 136 for $\mathcal{M}_2$. Finally, we contrast which mixed model was better, the one with a low or a high dimensional space.

2.2 Convolutional Neural Networks (CNNs)

The aim of modifying a LeNet-5 neural network by means of utilizing an individualized collection featuring handwritten numerals is to construct a prototype capable of correctly identifying and distinguishing written digits within our distinct dataset. This objective proves difficult as the initial design for the LeNet-5 model was created with regard to MNIST, which contains roughly 60k images utilized during training phases and approximately 10k examined in test stages. The utilization of a customized database together

with fewer examples can result in overfitting whilst simultaneously resulting in inadequate generalization performance levels.

The LeNet-5 structure comprises of seven distinctive layers, featuring two efficient convolutional strata tailored to extract valuable features from image inputs. There are also a pair of highly sophisticated subsampling layers that work in tandem with the former structures for enhancing feature extraction precision and efficiency. Additionally, this architecture integrates three fully connected topographies which allows seamless classification or regression tasks based on learned representations earlier in the network. The convolutional layers perform feature extraction by convolving the input image with a set of learnable filters. The subsampling layers perform downsampling to reduce the spatial resolution of the feature maps, while retaining important features. The fully connected layers perform classification by mapping the extracted features to the output classes. An illustration of the LeNet-5 architecture can be visualized in the figure 2 below.

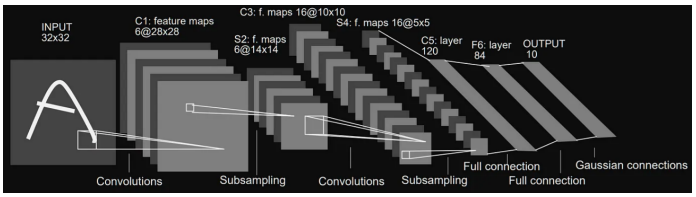

Fig. 2: LeNet-5 architecture. Link to source.

The LeNet-5 architecture performed exceptionally well on the widely-used MNIST dataset for recognizing handwritten digits, leading to the further development of intricate convolutional neural networks that play a pivotal role in advanced deep learning. The utilization of these intricate constructions has greatly enabled exceptional progress in the realm of artificial intelligence, encompassing varied fields like linguistics, auditory perception, and visual analysis. This has opened new avenues toward cutting-edge research aimed at enhancing computational efficiency through these novel approaches.

The custom dataset used for retraining the LeNet-5 neural network in the artificial intelligence class consisted of handwritten digits ranging from 0 to 9. The digits were collected from various sources, including handwritten digits on paper and digital digits created using devices such as an iPad (see figure 3). The digits were then normalized to a consistent size and format to ensure consistency across the dataset (see figure 4).

Since the dataset included a diverse range of handwriting styles, it can be challenging to train the LeNet-5 model without using techniques to address the diverse handwriting styles. However, in the class, the focus may have been on demonstrating the process of retraining the model on a custom dataset rather than addressing the diverse handwriting styles.

Throughout the course of a trial, the LeNet 5 CNN was taught over a span of fifty cycles using data that had been randomly dispersed via an even allocation. It is worth mentioning that excessive training can result in the overfitting of instructional material. By utilizing uniform distribution techniques on our information pool, we guaranteed equal opportunity for all points to be selected and tested while avoiding any possible favoritism within final outcomes. In conclusion, these tests establish an excellent groundwork by which one might assess how effectively LeNet 5's CNN fares against datasets distributed at random intervals.

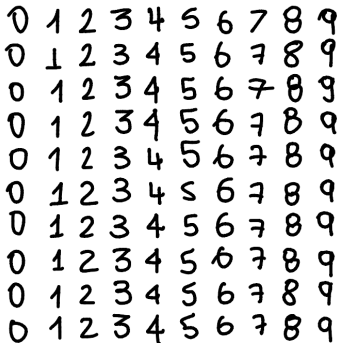

Fig. 3: Ipad written numbers

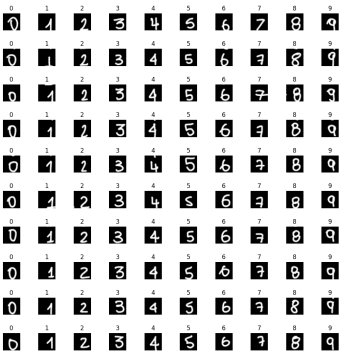

Fig. 4: Normalized numbers

2.3 Generative Adversarial Networks (GANs)

In order to investigate the use of generative adversarial networks (GANs) in image and video processing, we pursued two main goals in our study. First, we aimed to train a GAN consisting of a generator and discriminator using the MNIST dataset, a commonly used benchmark in the field of machine learning. This task involved generating synthetic images that resemble the hand-written digits in the dataset, while the discriminator distinguishes between real and generated images. Second, we sought to apply style transfer to a video by extracting each frame and using the trained GAN to transfer the style from a source image to each frame. We then reassembled the processed frames and original audio into a video. It is worth noticing that when using GANs in style transfer applications the actual output of the network is not used, instead, the resulting images are retrieved from the intermediate layers.

2.4 Training GANs with the MNIST dataset

The architecture of any GAN is composed of two neural networks: a generator network and a discriminator network. This implies that in order to train GANs we need to train two networks. While training the GAN, the generator network is fed with random noise (usually a vector, also known

as *latent vector*) and tries to generate new fake data. On each iteration, the generator tries to refine the generated data to more closely resemble real data. Meanwhile, the discriminator network is fed with the real data and the generated data from the generator and tries to differentiate the real data from the fake generated data. At some point, the discriminator won't be able to distinguish between the real and fake data which would stop the training process and indicate that the generator is ready to generate realistic data that resembles the real input data. One important thing to note is that each network is trained alternately, i.e., when training the generator the discriminator is not training and vice versa. An illustration of the architecture of GANs can be visualized in the figure 5 below.

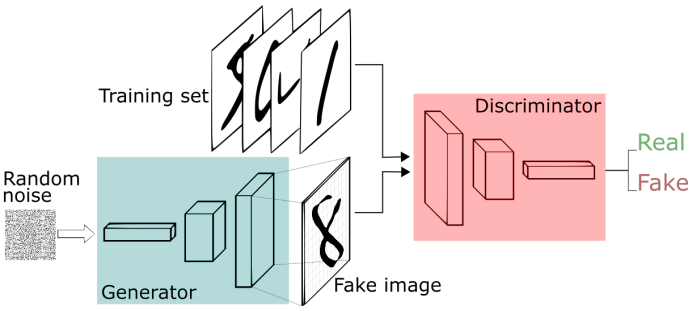

Fig. 5: Architecture of GANs. Link to source.

It is important to note that the generator and discriminator networks backpropagate differently during training. When training the discriminator, the discriminator classifies two classes of data: real and fake (generated). When this classification process occurs the discriminator loss is calculated. If the discriminator misclassifies either by classifying real data as fake or fake data as real, the discriminator weights are penalized via the loss function. Finally, the discriminator weights get updated through backpropagation exclusively throughout the discriminator network. When training the generator, the generator takes the random input vector and generates fake data. Then the discriminator network comes into play. By trying to classify the generated data as fake, the output of the discriminator is used as feedback to the generator by penalizing the generator for producing bad data. On the other hand, the generator loss is used to penalize the generator for not being able to "fool" the discriminator. The generator has then backpropagated by using its performance given by the discriminator output and by using its loss. Note that because each network trains one at a time, the backpropagation of the generator does not change or update the state of the discriminator in any manner, as it is simply used in a "frozen" state as a means to calculate the performance of the generator.

A sizable collection of handwritten numbers can be found in the MNIST database (Modified National Institute of Standards and Technology database). The main source for everything related to MNIST can be found in [9]. There are 60,000 examples in the training set and 10,000 examples in the test set. It is a subset of two larger NIST Special Databases, Special Database 1 (which contains handwritten digits written by high school students) and Special Database 3 (which contains handwritten digits produced by US Census Bureau personnel). The digits have been centered in a fixed-size image and size-normalized. To fit in a 20x20 pixel box while maintaining their aspect ratio, the original black and white (bilevel) photographs from NIST were size normalized. As a result of the normalization algorithm's anti-aliasing approach, the final photos have grey levels. The images were centered in a 28x28 image by computing the center of mass of the pixels, and translating the image so as to position this point at the center of the 28x28 field.

For the experiment, we considered a batch size of 512, 200 epochs to train the GAN, a fixed sample size of 64, and a random vector of length 128. The generator neural network is composed of 4 layers:

1) 128 inputs (the random vector length), 256 outputs, and a LeakyRelu activation function with a negative slope of 0.2.
2) 256 inputs, 512 outputs, and a LeakyRelu activation function (see figure 19) with a negative slope of 0.2.
3) 512 inputs, 1024 outputs, and a LeakyRelu activation function with a negative slope of 0.2.
4) 1024 inputs, 784 outputs (the size of a flattened MNIST 28×28 image), and a tanh activation function.

On the other hand, the discriminator network is composed of 4 layers:

1) 784 inputs (the size of a flattened MNIST 28×28 image), 1024 outputs, LeakyRelu activation function with a negative slope of 0.2, and a dropout (Bernoulli probability of randomly zeroing elements of the input tensor) with a rate of 0.3.
2) 1024 inputs, 512 outputs, LeakyRelu activation function with a negative slope of 0.2, and dropout with a rate of 0.3.
3) 512 inputs, 256 outputs, LeakyRelu activation function with a negative slope of 0.2, and dropout with a rate of 0.3.
4) 256 inputs, 1 output, and a sigmoid activation function.

To optimize each network we use 2 adam optimizers (which are the extended versions of stochastic gradient descent) with a learning rate of 0.0002, respectively. Also, we considered the Binary Cross-Entropy Loss Function for both networks. The mathematical expression of said function is as follows

$$\mathcal{L}(x, y) = \{l_1, ..., l_N\}^T,$$
$$l_n = -w_n[y_n \log x_n + (1 - y_n) \log(1 - x_n)], \quad \forall n \in [1, N]$$

2.5 style-based GANs

Style-based GANs are a type of generative adversarial network that have been used for the task of style transfer. In a typical GAN architecture, the generator network takes as input a random noise vector and produces an image, while the discriminator network tries to distinguish between real images and those generated by the generator. In style-based GANs, the generator network is composed of two parts: a mapping network and a synthesis network. The mapping network takes a style vector (reference style) as input and

learns to map it to a high-dimensional latent code, which is then used as input to the synthesis network. The synthesis network then generates the final image by applying style modulation to the intermediate feature maps in each layer of the network. Style-based GANs allow for more control over the generated images, as different style vectors can be used to produce different styles and attributes in the generated images. An illustration of the basic architecture of these networks can be visualized in the figure 6 below.

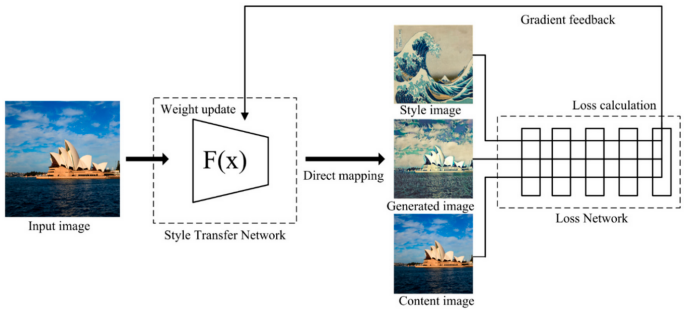

Fig. 6: Architecture of style-based GANs. Link to source.

Our goal was to apply transfer to an input video given an arbitrary style reference image. Because of this, we first extracted each frame of the video resulting in a finite set of images based on the frames per second (FPS) and duration. Then, we performed the fast-style transfer method proposed by [10] to each frame, resulting in a stylized image. After stylizing each frame, we then proceeded to reconstruct the video by merging all the frames back together based on the FPS and duration of the original video. The audio was extracted from the original video as well and added to the resulting video.

The optimization function objective for the style transfer of an image can be described as

$$\min_{x} \mathcal{L}_c(x, c) + \lambda_s \mathcal{L}_s(x, s)$$

where $\mathcal{L}_c(x, c)$ and $\mathcal{L}_s(x, s)$ are the content and style losses, respectively and λ_s is a Lagrange multiplier that weights the relative strength of the style loss. Lower-level and higher-level features are associated as the activations within a given set of lower and higher layers $\mathcal{S}$ and $\mathcal{C}$ in an image classification network, respectively. The content and style losses are defined as follows

$$\mathcal{L}_s(x, s) = \sum_{i \in S} \frac{1}{n_i} \left|\left| \mathcal{G}[f_i(x)] - \mathcal{G}[f_i(x)] \right|\right|_F^2$$

$$\mathcal{L}_c(x, c) = \sum_{j \in S} \frac{1}{n_j} \left|\left| \mathcal{G}[f_j(x)] - \mathcal{G}[f_j(x)] \right|\right|_2^2$$

where $f_l(x)$ are the activation functions of layer l, n_l is the total number of units at layer l and $\mathcal{G}[f_l(x)]$ is the Gram matrix associated with the layer l activation functions. In style transfer, the Gram matrix is a mathematical representation of the correlations between the features extracted from a given image. Specifically, the Gram matrix is calculated by taking the outer product of the flattened feature map with itself and computing the dot product between them. This results in a matrix that summarizes the information about which features tend to activate together, effectively capturing the "style" of the image. By comparing the Gram matrices of the style and content images, it is possible to extract the style information from the style image and transfer it to the content image. Formally, the Gram matrix is a square, symmetric matrix that measures the spatially averaged correlation structure across the filters within a layer's activation functions.

2.6 Code implementation and GitHub repository

All the code was structured and managed in a public GitHub repository created for the course (link to the repository). We implemented every algorithm in Google Colab which currently runs in `Python 3.9` with the use of either GPU or TPU acceleration. The computational resources of Google Colab are listed in table 1 below.

Resource	CPU	GPU	TPU
Processing power	12 GB RAM, 2 vCPUs	12 GB RAM, 1 GPU	16 GB HBM, 8 cores
Model	Intel Xeon or AMD EPYC processor	NVIDIA Tesla K80	Google TPU v2
Cores	Varies by model and configuration	2496 CUDA cores	8 cores
Frequency	Varies by model and configuration	Varies by model and configuration	700 MHz
Memory	Varies by model and configuration	12 GB GDDR5 VRAM	16 GB HBM
Memory Bandwidth	Varies by model and configuration	Varies by model and configuration	900 GB/s
Version	N/A	N/A	v2

TABLE 1: Computational resources of Google Colab. Link to source.

The available resources for each Colab session depend on whether the user's plan is free or paid. The paid version can be customizable, which means that every user can have particular sessions. A direct link for the notebook of each network implementation is provided in the table 2 below.

Method	Direct link
Autoencoders with MLP networks	link to source
CNNs with LeNet5 and MNIST	link to source
GANs for handwritten digit generation with MNIST and custom database	link to source
GANs for video style transfer	link to source

TABLE 2: Direct links to each Google Colab notebook for each network application found in the GitHub repository.

3 Results and Discussion

3.1 Autoencoders

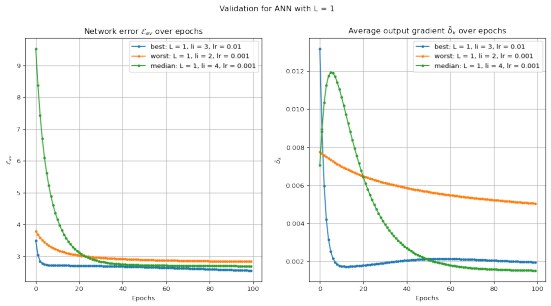

Fig. 7: Average instant energy and local output gradient by epoch for the best, median and worst NN in the validation data.

The search of the best NN following the process described in the previous section, quickly showed that the lowest and the three highest learning rates were unsatisfactory to optimize by gradient descendent the NNs. Conversely, all the NNs with learning rates of 0.01 and 0.1 presented a clear convergence of the loss.

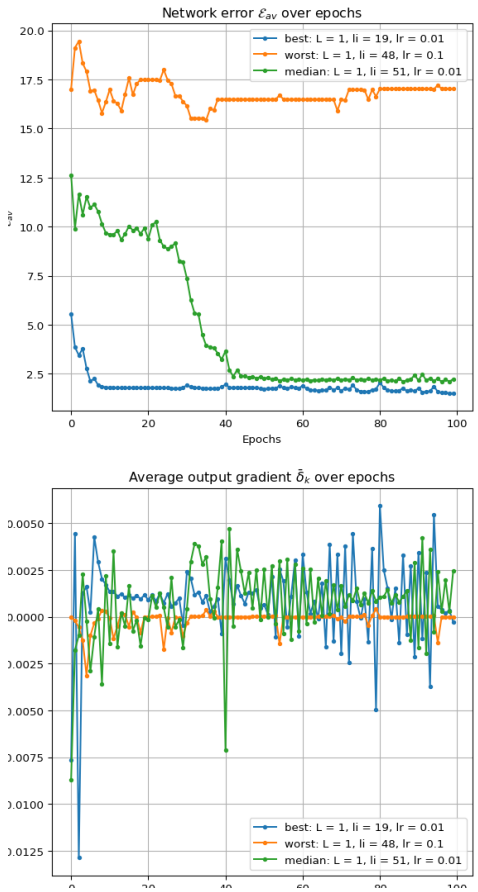

Fig. 8: Average instant energy and local output gradient by epoch for the best, median and worst $\mathcal{M}_1$ in the validation data.

The above is clear from the figure 7, which exhibits the average instant energy and local output gradient by epoch for the best, median and worst NN in the validation data with learning rates of 0.01 and 0.1. From it is also clear that the best NN have only one hidden layer with three neurons and a learning rate of 0.01. In consequence, we implemented a NN in tensorflow with this paremeterization.

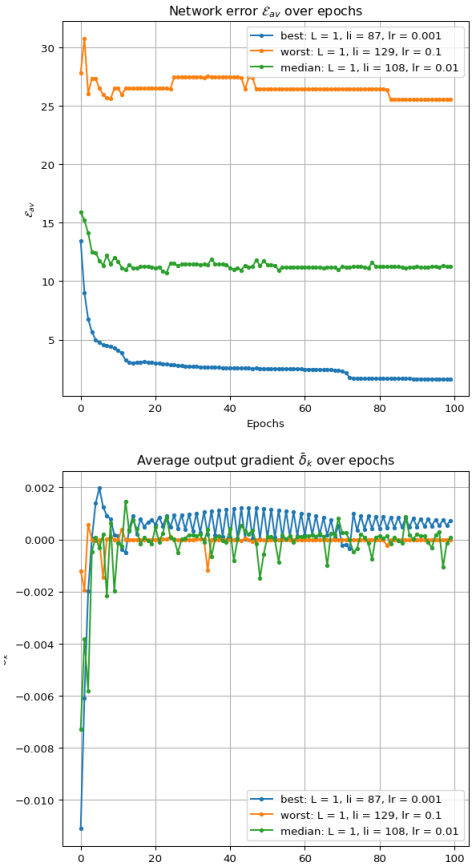

Fig. 9: Average instant energy and local output gradient by epoch for the best, median and worst $\mathcal{M}_2$ in the validation data.

Henceforth the results will only consider the learning rates 0.01 and 0.1. Since similarly to the NNs, these give the best results in the autoencoders. The figures 8 and 9 present the average instant energy and local output gradient by epoch for the best, median and worst $\mathcal{M}_1$ and $\mathcal{M}_2$ in the validation data. In general, higher neurons in the hidden layer throws worst results though it has exceptions, for instance the median $\mathcal{M}_1$, which is very close to the best, has more perceptrons than the worst $\mathcal{M}_1$.

In constrast to the training of the NNs, the average output gradient does not converge for the best and median autoenconder in neither high and low dimensional spaces. Conversely, it oscillates around low values near to zero. Another interesting aspect to stressed is that the the $\mathcal{M}_1$ autoencoders are very sensitive to the number of hidden neurons.

From the optimization process we obtain that the optimal low dimensional autoencoder $\mathcal{M}_1^*$ is given by 19 neurons in the hidden layer, while the optimal high dimension autoenconder $\mathcal{M}_2^*$ has 87 hidden neurons. Despite both parameterizations differ considerably, the loss is very similar.

Finally we tested the performance of our models. For this we contrasted the true outputs against the predicted outputs of the best model. The figures 10 and 11 present the plots for the latitude, where is clear that neither the low nor the high dimensional model learn to determine the origin location of the tracks. Similar results were obtained for the longitude and can be found in the appendix.

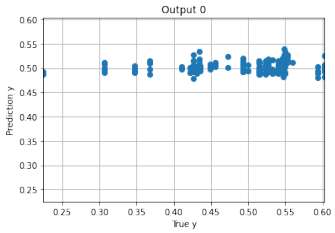

Fig. 10: True latitude against predicted latitude of the best model found in a low dimensional space

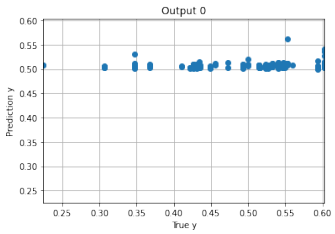

Fig. 11: True latitude against predicted latitude of the best model found in a high dimensional space

3.2 Convolutional Neural Networks (CNNs)

Figure 12 shows that around iteration 5, the training set and the validation set started to diverge in both loss and accuracy. It can be inferred from this evidence that the model is potentially exhibiting signs of overfitting to the initial training data with an accuracy of Train equal to 0.973 and Validation equal to 0.615, as it appears excessively tailored to and reliant on said data which may consequently hinder its capacity for extrapolation onto unfamiliar datasets.

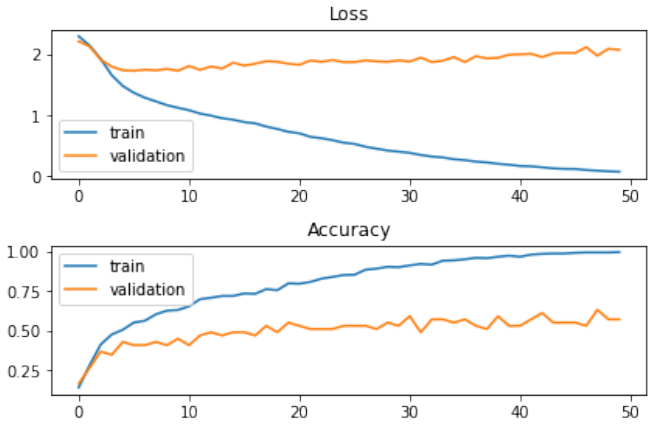

Fig. 12: Accuracy and validation on LeNet 5 with 50 epochs.

In reaction to the observed issue of overfitting during the last trial, we made a decision to lower the number of epochs from 50 down to only 10 for control purposes. However, this alteration failed in producing our desired outcome with an accuracy of Train equal to 0.645, and Validation equal to 0.492 (see Figure 13); instead, it seemed that reducing epoch count had no positive impact on the model performance. In fact, compared to results seen in previous tests concerning the accuracy and loss metrics within training data sets performed less favorably leading us toward theories regarding underfitting as a potential explanation. Surprisingly though, while verifying test outcomes did not improve noticeably with regards to overfit sensitivity perhaps hinting at an indication that suggests lowering the number of epochs may prove unfruitful when trying to decrease effects associated with over-fitting concerns as the training set stops improving instead of getting worse in the Figure 12.

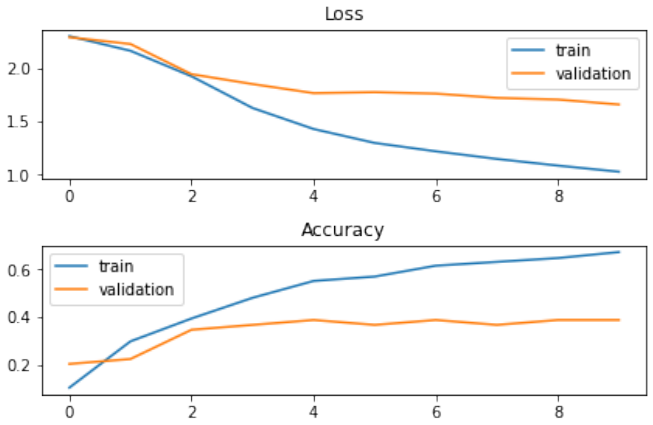

Fig. 13: Accuracy and validation on LeNet 5 with 10 epochs.

3.3 Generative Adversarial Networks (GANs)

3.3.1 *Handwritten digit generation*

The results of our study's first goal for GANs are promising, as the GAN was able to generate synthetic images resembling the hand-written digits in the MNIST dataset. As we observed in the generated images at epochs 1, 100, and 200 from figure 14, the GAN improved its ability to produce increasingly clear and less randomized digit images with each epoch of training. In particular, the generated images at epoch 200 closely resemble the real hand-written digits in the MNIST dataset, indicating the GAN's high degree of proficiency in generating digit images. However, the persistent blurriness and randomness in the generated images, even at epoch 200, suggest that the GAN still has limitations in producing high-quality synthetic images. Further improvements in the GAN's architecture and training techniques could potentially address these limitations and enhance its ability to generate more realistic synthetic images.

In addition to the gradual improvement in image quality observed in the generated images over the epochs, it is worth noting that the GAN's ability to produce specific digit images varied during training. For example, some digit images were generated with high accuracy from the early epochs, while others only began to emerge with clarity later in the training process. This suggests that the GAN may

Fig. 14: Generated images over 3 different epochs: epoch 1 at the start of the GAN training, epoch 100 on half of the training iterations, and epoch 200 on the last iteration.

be better suited to certain digit images than others, which could be due to variations in the digit images' complexity, frequency of occurrence, or other factors. Moreover, while the GAN's ability to generate high-quality images improved over the epochs, there were still instances where the generated images were incomplete, distorted, or completely different from the original digit image. It is worth noticing that the use of a LeakyReLU activation function for intermediate layers of both the generator and discriminator networks allowed for non-linearity and more robustness against vanishing gradients. The use of the Tanh activation function in the output layer of the generator and the sigmoid activation function in the discriminator's output layer allowed for the range of the generated images to be between -1 and 1, which is consistent with the MNIST dataset. The inclusion of dropout with a probability of 0.3 for each layer in the discriminator network helped to prevent overfitting, which is a common problem in GANs.

3.3.2 *Video style transfer from reference image*

The results of our second goal with GANs, which involved applying style transfer to a video using a pre-trained GAN, are promising. Figure 15 shows a frame of the input video in comparison to the reference style image of "The Starry Night" by Vincent Van Gogh. As we can see, the frame of the input video does not share any visual characteristics with the reference image. However, after applying the style transfer to the same frame, we can see a significant improvement in the visual appearance of the frame. Figure 16 shows this behavior by comparing a different frame of the input video before and after the style transfer. We can see that the visual characteristics of the reference image are now clearly visible in the resulting frame. These results demonstrate the effectiveness of the style transfer technique in transferring the style of a reference image to a video frame and suggest its potential for use in various applications such as video editing and digital art. However, it is important to note that the quality of the output may depend on the choice of the reference image and other parameters used in the style transfer process.

The use of a pre-trained GAN for arbitrary style transfer, as we did in our study, has several advantages over traditional style transfer methods. One of the main advantages is the ability to perform fast style transfer with high-quality results. This is due to the network's specific architecture and pre-trained weights, which have learned to extract and transfer the style information efficiently. Another advantage is the ability to perform style transfer with only one reference image, making the process simpler and more convenient for the user.

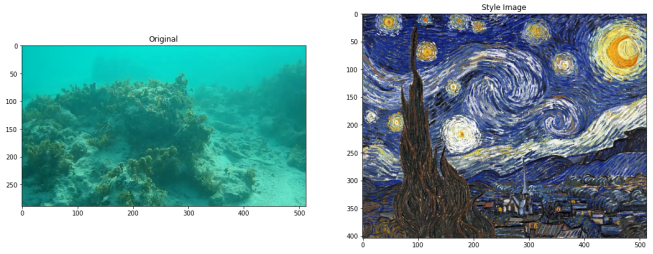

Fig. 15: A frame of the input video compared with the reference style image.

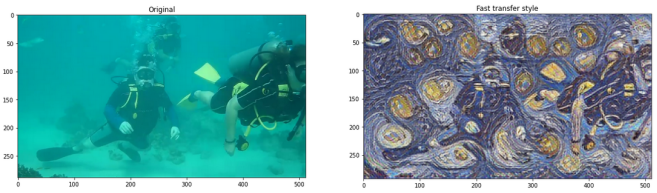

Fig. 16: A frame of the input video compared with the same frame after style transfer. The output video can be found on the following link.

4 Conclusion

During this work we explore the performance of NNs and autoencoders under different parameterizations. Were we learn important insights. First is important to made an extensive search of the optimal parameterization of the models since can improve considerably the results, hence employ an efficient implementation is critical to ease the burden of this task. Second, despite AI has gain a remarkable popularity in modern world, not every neural network will capture the behaviour of the data, indeed, even the best founded model failed to learn the GOM dataset. Therefore is critical that the modeler is deeply involved in the training process which should not be done in an automatic and generic way. Finally, we learn that in practice is difficult to develop a good model that could capture a subset of the reality, indeed in our experiment we failed to do it. Hence is suggested as future work to make a wider search of the optimal parameterization of the model.

By its part, our MNIST dataset of only 488 images may not be enough for satisfactory outcomes when partitioning into training, validation, and testing subsets, especially with complex models like LeNet 5, which may suffer from overfitting due to the small training set size. Dividing the dataset can also introduce complications with image style and content in each subset. To address these issues, careful selection of dataset size and distribution techniques, along with generative properties like transfer learning and augmentation tools, can improve performance despite limited data. These considerations are crucial for small-scale projects, including technology development processes, and model evaluation workflows.

We explored the effectiveness of generative adversarial networks (GANs) in generating synthetic images that re-

semble hand-written digits in the MNIST dataset. Through a series of experiments, we demonstrated that GANs can generate increasingly clear and less randomized digit images with each epoch of training, with the generated images at the last training epoch closely resembling the real hand-written digits in the dataset. However, some degree of blurriness and randomness persisted even when considering 200 epochs, indicating that there is still room for improvement in the GAN's ability to generate high-quality synthetic images. Our findings suggest that GANs can be a powerful tool for image generation tasks, but careful consideration and thorough evaluation are necessary when selecting the appropriate network for specific applications.

By using a pre-trained network to perform style transfer to an input video based on an arbitrary reference image, we were able to leverage the existing knowledge in the network to achieve impressive style-transfer results on our video frames. This approach saved us the time and computational resources required to train a new GAN specifically for style transfer on our dataset. However, it is important to note that the pre-trained network was not designed for the specific style transfer task we performed, which may have limited the degree of customization and control we had over the style transfer process. Despite this limitation, our results still demonstrate the potential of pre-trained networks for style transfer applications, and future research could explore the use of customized pre-trained networks or fine-tuning techniques to further improve results.

REFERENCES

[1] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *nature*, vol. 323, no. 6088, pp. 533–536, 1986.

[2] I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning*. MIT press, 2016.

[3] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[4] G. E. Hinton and R. R. Salakhutdinov, "Reducing the dimensionality of data with neural networks," *science*, vol. 313, no. 5786, pp. 504–507, 2006.

[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] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial networks," https://arxiv.org/pdf/1406.2661.pdf, 2014.

[7] F. Zhou, Q. Claire, and R. D. King, "Predicting the geographical origin of music," in *2014 IEEE International Conference on Data Mining*. IEEE, 2014, pp. 1115–1120.

[8] G. Tzanetakis and P. Cook, "Marsyas: A framework for audio analysis," *Organised sound*, vol. 4, no. 3, pp. 169–175, 2000.

[9] Y. LeCun, C. Cortes, and C. Burges, "The mnist database of handwritten digits," http://yann.lecun.com/exdb/mnist/, accessed: 2023-03-20.

[10] G. Ghiasi, H. Lee, M. Kudlur, V. Dumoulin, and J. Shlens, "Exploring the structure of a real-time, arbitrary neural artistic stylization network," https://arxiv.org/pdf/1705.06830.pdf, 2017.

APPENDIX A
ADDITIONAL FIGURES

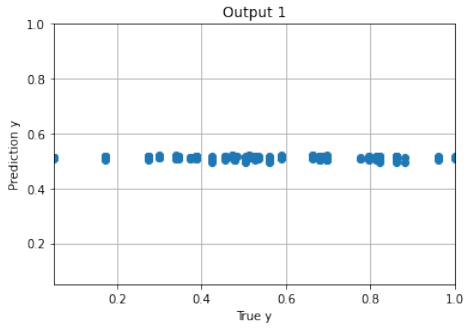

Fig. 17: True longitude against predicted longitude of the best model found in a low dimensional space

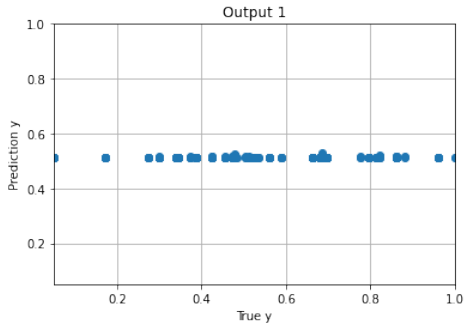

Fig. 18: True longitude against predicted longitude of the best model found in a high dimensional space

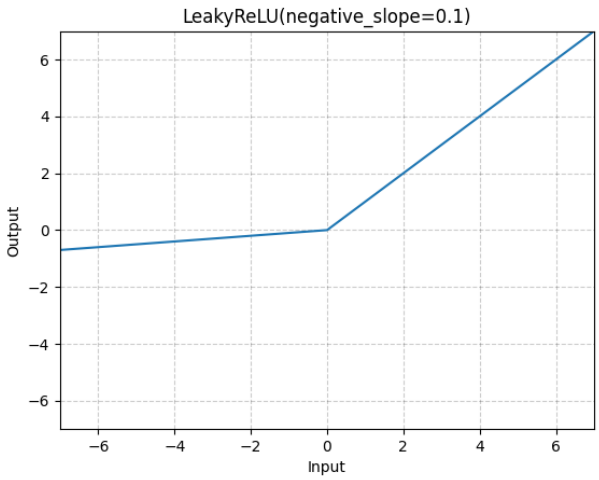

Fig. 19: LeakyReLU activation function. Link to source.