Enhancing Convolutional Neural Network Classification Performance through Dataset Enrichment: A Comparative Study

Jonas Buchart Svenningsen, Máté Guber, Magnus Just Sund Carlsen

***Abstract*—This research is an exploration to compare the effect of enriching a small dataset through different techniques. The techniques considered are duplicating images, adding augmented variations of the original images or generating synthetic images by means of an independent Deep Convolutional Generative Neural Network (DCGAN). Different classifiers were trained on different datasets, enriched through the different techniques. The classifiers trained on enriched datasets, including DCGAN generated datasets, shows promising results and were able to match or exceed the performance of the models trained on the baseline dataset. This suggests that artificially generating additional training data can enhance the classification process. However, further research is required to fully understand the limitations and potentials of these methods in different contexts and with various dataset characteristics.**

Date:	June 10, 2024
Project Type:	Master's Thesis
Project Period:	February 2024 - June 2024
Project Group:	cs-24-mi-10-07
Supervisor:	Anders Læsø Madsen
Total Page Number:	21

I. Summary

This paper investigates various techniques for enriching a small dataset, focusing on their impact on classification accuracy. The methods explored include simple image duplication, image augmentation, and generating synthetic images using a Deep Convolutional Generative Adversarial Network (DCGAN). The study aimes to determine how these techniques affect the performance of Convolutional Neural Network classifiers trained on datasets enhanced by these methods compared to a baseline, unchanged dataset.

Key methodologies included:

1) **Bootstrapping**: Random selecting images in the original dataset and copy these into the existing dataset.
2) **Data Augmentation**: Techniques such as rotation, scaling, zooming and flipping of original images to create variations in the dataset.
3) **Synthetic Image Generation using DCGANs**: New images are generated by a DCGAN which is trained to produce realistic images based on the original dataset.

The focus was on the DIME database [1], particularly on enriching the Iron Age coin dataset, which has the least samples.

Findings

The experiments demonstrated that enriching the dataset with synthetic images generated by DCGANs did enhance the performance of Convolutional Neural Network classifiers. These generated images helped address issues of data imbalance and small data sample sizes, leading to improved classification accuracy. *However*, the degree of improvement varied across different augmentation methods, indicating that the effectiveness of synthetic data augmentation can be context-dependent.

Limitations and Future Work

Despite promising results, several limitations were identified. The study focused on binary classification, leaving multi-class classification unexplored, which reflects the real-world applications. Additionally, the study was limited by the small subset of coin images used, suggesting that more experiments with larger and more diverse datasets are necessary. Areas of future research include:

- **Multi-Class Classification**: To understand the potential in complex scenarios of multi-class classification.
- **Advanced DCGAN Architectures**: Explore more advanced architectures to explore possible enhancements for training the DCGAN.
- **Higher Resolution Images**: The current experiments used 64x64 images, which may have lost some of the image features. Exploring using higher resolution images is something to investigate.
- **Combining Synthetic Data with Other Augmentation Strategies**: To further enhance model robustness and generalization.

Conclusion

This research highlights the potential of using DCGANs for dataset enrichment in the field of archaeological artifact classification. While the approach shows promise, it emphasizes the need for ongoing experimentation and tuning to fully benefit from generated images for enriching a dataset. Future research will focus on larger datasets, more complex classification tasks, and improved image resolution to further advance the classification and study of historically significant artifacts.

II. INTRO

In the archaeological study area, artifact classification of ancient coins is an area with a set of challenges and opportunities for the application of machine learning technologies. To address some of these challenges, we have access to the full DIME database [1], which includes images of different kinds of coins such as coins from the Viking Age, Reformation Time, Medieval Periods, and even from the Iron Age, which allows for the possibility of exploration and analysis based on generating new such images. However, a lack of samples and non-uniformity in the samples distribution across classes seriously deviates base performance for conventional classification models, e.g., Convolutional Neural Networks (CNNs) [2]. This imbalance prompts the need for innovative approaches to enhance the training datasets to improve model accuracy and reliability.

This research paper looks at the possibility of applying Deep Convolutional Generative Adversarial Networks (DCGANs) in order to augment/enrich the datasets of classes where examples are few by synthesizing new coin images for that specific class. The classifiers trained in this experiment are all binary classifiers, and the ”other” label will always match the coin label in size. By generating artificial coin images, this study aims to address the data imbalance issue and the issue of small datasets and examine its effects on the performance of a CNN classifier. Essentially, this is an experiment on different ways to enrich an existing small dataset in an attempt to make it larger, and more effective. The experiment will be done on multiple different options regarding dataset enrichment; namely bootstrapping, data augmentation, and most interestingly, the study seeks to determine whether augmentation of the dataset with synthetic images improves the accuracy and robustness of the classifier. After each classifier has been trained, they will all be validated and compared on a holdout set comprised of coins from the Viking Age. These coin images' contents are very similar to those of the images from the Iron Age, which makes the validation solid.

This paper, therefore, goes through the relevant literature regarding the use of GANs in image generation and data augmentation. It details the method that was applied in training the DCGAN and the CNN, giving comprehensive reasoning of the experimental results. The paper aims to contribute, in its turn, to the broader discussion of the optimal strategies to deploy artificial intelligence in the field of archaeological research, and more particularly, in the classification of historically relevant artifacts.

III. Background

A. Convolutional Neural Network

A Convolutional Neural Network (CNN) is a class of deep learning algorithms that is specialized for processing data structured in grids, like images; grids of pixels. They are commonly used in computer vision, because they have the ability to learn spatial hierarchies of the features in the input images. This section will provide a short overview of what a CNN is, and how it works.

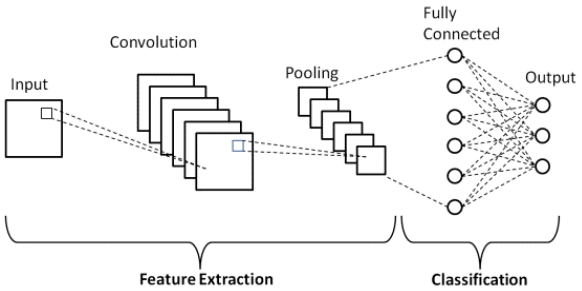


Fig. 1: A simple Convolutional Neural Network Architecture.

Looking at figure 1 it is clear that the architecture comprises three different types of layers, namely convolutional, pooling, and fully connected layers. Firstly, the purpose of the convolutional layer is to learn the representation of features in the input. Mathematically, convolution is an operation that can be done on two lists of numbers.[3] For two lists a and b, the convolution c is defined as:

$$c[n] = (a * b)[n] = \sum_{m=0}^{M-1} a[m] \cdot b[n-m]$$

where:

- $[n]$ is the nth element
- a and b are the input sequences of lengths M and N respectively.
- c is the output sequence of length $M + N - 1$.
- $a[m]$ is the m-th element of list a.
- $b[n-m]$ is the $(n-m)$-th element of list b, and if $(n-m)$ is out of bounds, $b[n-m]$ is considered to be 0.

Here's a step-by-step outline of convolving two lists:

1) Slide one list over the other.
2) Multiply overlapping elements.
3) Sum the products to get each element of the resulting list.

Example: Given $a = [1, 2, 3]$ and $b = [4, 5, 6]$:

$$c[0] = (1 \cdot 4) = 4$$
$$c[1] = (1 \cdot 5) + (2 \cdot 4) = 5 + 8 = 13$$
$$c[2] = (1 \cdot 6) + (2 \cdot 5) + (3 \cdot 4) = 6 + 10 + 12 = 28$$
$$c[3] = (2 \cdot 6) + (3 \cdot 5) = 12 + 15 = 27$$
$$c[4] = (3 \cdot 6) = 18$$

Resulting in $c = [4, 13, 28, 27, 18]$.

This works in a very similar way in the convolutional layers of a CNN. Instead of 1-dimensional lists doing the convolution, we have 2d grids where the convolutions are performed. We can think of the input image as a grid of pixel values, while the kernel sliding across is a grid of simple values, and as we can see in the example with the two lists, the output is different from both of them. In a pure convolution with two lists, the output will always be larger than both inputs, unless one of the lists is only of length 1. This happens due to the fact that in the beginning and the end of the convolutional operation, the two lists will not fully overlap, yielding a lengthier result. This is not a concern in the CNN, as we only consider the operation where the kernel fully overlaps with the input image.

The output size (O) of an image after applying a convolutional layer in a CNN can be calculated using the following equation:

$$O = \left\lfloor \frac{I + 2P - K}{S} \right\rfloor + 1$$

Where:

- I is the input size.
- P is the amount of zero padding applied to the input image.
- K is the size of the kernel.
- S is the stride of the convolution.
- $+1$ accounts for the starting position of the kernel

This means that after the convolution has been performed in a CNN, the output is actually smaller than the input, however the pixel values will have changed based on what values are inside the kernels sliding across. The values inside the kernels are learned values and the resulting feature map is seldomly interpretable by humans. Here is an example of a 3x3 kernel that *is* interpretable, and will have a very simple effect on an input image:

$$\begin{bmatrix} 0.25 & 0.5 & 0.25 \\ 0 & 0 & -0 \\ -0.25 & -0.5 & -0.25 \end{bmatrix}$$

The values inside this kernel sum to 1, which means that when sliding across an image, this kernel will not change anything, as long as it is sliding over the same

color. However, if it detects horizontal edges, it will highlight them as seen in figure 2 (the kernel is sliding left to right, top to bottom. red values are negative, blue are positive):

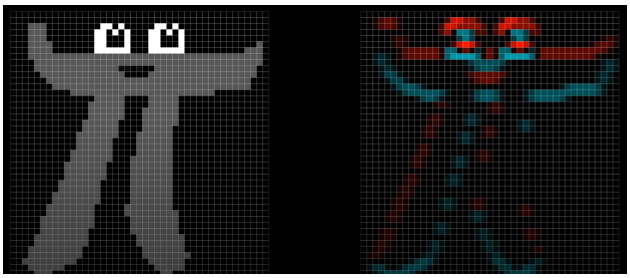


Fig. 2: Feature map from simple kernel [4]

By doing this convolution over the input with the learned kernel, and applying an activation function, the new feature map is obtained. The feature value at location (i, j) in the kth feature map of the lth layer, $z^l_{i,j,k}$ can be calculated:

$$z^l_{i,j,k} = {\mathbf{w}^l_k}^T \mathbf{x}^l_{i,j} + b^l_k \tag{1}$$

Here $\mathbf{w}^l_k$ and b^l_k represent the weight vector and bias term of the kth filter in the lth layer, respectively, and $\mathbf{x}^l_{i,j}$ is the input patch centered at location (i, j) in the lth layer. [5] According to Gu et al. the kernel $\mathbf{w}^l_k$ creating the shared feature map $\mathbf{z}^l_{:,:,k}$ employs a weight sharing mechanism. This mechanism simplifies the model and makes training easier.

The activation function applied on the output of the convolution operation introduces non-linearities, which allows multi-layer networks to detect non-linear functions. Sigmoid, tanh and ReLU are some of the most common activation functions used with CNNs. [5]

Next type of layer we see in figure 1 is the pooling layer. This layer is used to reduce the spatial dimensions of the input feature maps. There are two common pooling operations [5]:

- Max pooling: The maximum value within the pooling window is selected
- Average pooling: The average of all values within the pooling window is selected

Let pool$(\cdot)$ denote the pooling function, then for each feature map $\mathbf{a}^l_{:,:,k}$, we compute the corresponding pooled representation, $y^l_{i,j,k}$ as:

$$y^l_{i,j,k} = \text{pool}(a^l_{m,n,k}), \forall (m, n) \in R_{ij} \tag{2}$$

where R_{ij} is a local neighborhood around location (i, j). Given that the receptive field of the convolving kernels increase with the depth of the network, by stacking alternating convolutional and pooling layers, it is possible to extract increasingly higher level feature-representations [5].

The third and last type of layer in the CNN is the fully-connected layer. This layer takes as input the final representations of the input image, as feature maps. This layer then performs high-level reasoning based on the global information it has about the feature map. This is done, in practice, by having a connection between every single neuron between layers. For traditional CNNs, the fully-connected layer is very common; but it can be swapped out for alternatives such as 1x1 convolution, that reduce the number of channels in the image, while preserving the spatial dimensions, or global average pooling, that reduce each channel of the feature maps to a single value. These both alleviate the need to flatten the feature map and use a fully-connected layer.

The last part of the CNN is the output layer which, in the case of a classifier, outputs the classification. Gu et al. suggest that obtaining the optimal parameters for any given task involves minimizing a suitable loss function defined for that task. This allows for track keeping of the improvements of the model. The loss of a CNN, denoted as L, can be calculated:

$$L = \frac{1}{N} \sum_{n=1}^{N} l(\theta; \mathbf{y}^{(n)}, \mathbf{o}^{(n)}) \tag{3}$$

where θ is every parameter of a CNN, the number of desired input-output relations is N $\{(\mathbf{x}^{(n)}, \mathbf{y}^{(n)}); n \in [1, \ldots, N]\}$, with $\mathbf{x}^{(n)}$ being the n-th input data and $\mathbf{y}^{(n)}$ being its corresponding label and $\mathbf{o}^{(n)}$ being the output of the CNN. Along these lines, Gu et al. [5] argue that training a CNN is a global optimization problem. By minimizing the loss function, the optimal parameters can be identified. They highlight stochastic gradient descent as a common method for optimizing CNNs. [5]

B. Deep Convolutional Generative Adversarial Network (DCGAN)

Inspiration of the Deep Convolutional Generative Adversarial Network (DCGAN) model is taken from the original papers on the topic (including the orignal Generative Adversarial Network (GAN)) [6][7] and PyTorch's implementation [8].

The Generative Adversarial Network consists of two connected networks namely a Generator for producing a fake image dataset drawn from a latent space and a Discriminator which purpose is to classify an input as real or fake, where the fake image is generated by the Generator.

The DCGAN model is a type of Generative Adversarial Network (GAN) specifically designed for generating high-quality images. It comprises two deep learning

models: the Generator for generating images and the Discriminator for guessing between real and fake images. The generator will be the part of the model which is used afterwards (i.e. after training) for image generation.

- **Generator**: The Generator takes random noise in terms of a latent vector as input and generates synthetic images, based on how it learns the latent representation during training. It learns to transform the input noise into meaningful data representations resembling the real images from the dataset. The Generator's goal is to create images that are indistinguishable from real ones.
- **Discriminator**: The Discriminator acts as a classifier that evaluates the authenticity of the generated images. It receives both real images from the dataset and fake images produced by the Generator. The Discriminator's objective is to correctly distinguish between real and fake images. The output is a probability of being real or fake i.e. [0;1].

Training the DCGAN involves a competitive process between the Generator and the Discriminator:

- **Generator Training**: Initially, the Generator produces random images that are far from realistic. As training progresses, it learns to generate images that increasingly resemble the real ones. The goal is for the Generator to fool the Discriminator into classifying its generated images as real.
- **Discriminator Training**: Simultaneously, the Discriminator is trained to improve its ability to differentiate between real and fake images. It learns to correctly classify real images as real and fake images as fake. The Discriminator's objective is to become proficient at distinguishing between genuine and synthetic data.

1) Objective: During training, both networks continually improve their performance through iterative adjustments. The process continues until either the Generator produces images that are convincingly real or the Discriminator becomes highly skilled at distinguishing between real and fake images or the model breaks.

The DCGAN model [7] components can be viewed and described as seen in figure 3:

- z: Random noise vector sampled from a latent space.
- $G(z;\theta_g)$: Generator function parameters θ_g, which maps the noise vector z to a synthetic image.
- x: Real image sampled from the dataset.
- $G(z)$ is the output of the generator i.e. fake images.
- $D(x, G(z);\theta_d)$: Discriminator function parameters by θ_d, which evaluates the probability of authenticity of the input image.
- θ: Model parameters, which include both θ_g (generator parameters) and θ_d (discriminator parameters).

The objective and training of the DCGAN can be formulated as a minmax game between the Generator and the Discriminator, where the goal for the Generator is to maximize the Discriminators loss, i.e., generate images which the Discriminator cannot distinguish between real and fake images. It can be described as:

$$\min_{\theta_g}\max_{\theta_d} V(D,G) = \frac{1}{m}\sum_{i=1}^{m}\Big[\log D(x^{(i)}) + \log(1 - D(G(z^{(i)})))\Big] \quad (1)$$

where:

- m is the batch size, representing the number of samples in each batch used during training.
- $D(x)$ is the discriminator's output for a real sample x, representing the probability that x is a real data point.
- $G(z)$ is the generator's output for a latent vector z, representing a generated data sample.
- $D(G(z))$ is the discriminator's output for a generated sample $G(z)$, representing the probability that the generated sample is a real data point.
- $\log D(x^{(i)})$ is the log probability that the discriminator correctly identifies the real data sample $x^{(i)}$ as real.
- $\log(1 - D(G(z^{(i)})))$ is the log probability that the discriminator correctly identifies the generated sample $G(z^{(i)})$ as fake.
- θ_g represents the parameters of the generator that are optimized to minimize the loss.
- θ_d represents the parameters of the discriminator that are optimized to maximize the loss.
- $x^{(i)}$ represents the i-th image in the batch.
- $z^{(i)}$ represents the latent space vector corresponding to the i-th image.

C. The Generator

The Generator aims to minimize this objective by generating images that are indistinguishable from real

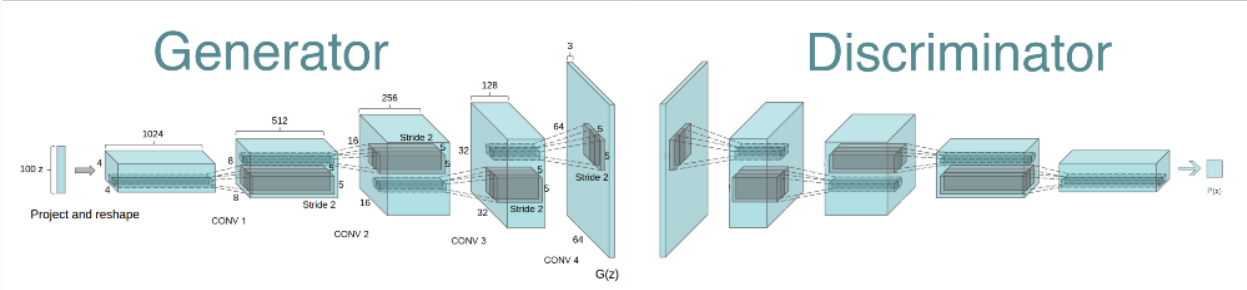


Fig. 3: Illustration and intuition of the DCGAN architecture [7].

ones, while the Discriminator aims to maximize it by correctly distinguishing between real and fake images.

Biases in the transpose layers are omitted to stabilize training, as the layer normalization process already includes adjustable learnable parameters.

The deconvolutional operation in the generator involves a sequence of transpose convolutional layers, each followed by layer normalization, and the model itself ends with a tanh [9] activation function. The input for the generator is a projected latent vector $\mathbf{z}$ s.t. $\mathbf{z} \in \mathbb{R}^{n,1,1}$ and $n = 100$.

The generator model can be described as follows:

$$\mathbf{h}_0 = \mathbf{z} \tag{1}$$

$$\mathbf{h}_i = \begin{cases} \text{ReLU}_i\left(\text{LayerNorm}_i\left(\text{ConvTranspose}_i(\mathbf{h}_{i-1})\right)\right) & \text{if } i < k \\ \text{ConvTranspose}_k(\mathbf{h}_{k-1}) & \text{if } i = k \end{cases} \tag{2}$$

$$\text{for } i = 1, 2, \ldots, k \tag{3}$$

$$G(\mathbf{z}) = \tanh(\mathbf{h}_k) \tag{4}$$

where:

- $\mathbf{z} \in \mathbb{R}^n$ is the latent vector.
- $G(\mathbf{z})$ is the output of the generator given the latent vector $\mathbf{z}$.
- k is the number of transpose convolutional layers.
- $\mathbf{h}_0 = \mathbf{z}$ is the initial input to the generator.
- $\mathbf{h}_i$ represents the output of the i-th layer.
- $\text{ConvTranspose}_i(\cdot)$ represents the transpose convolution operation applied in the i-th layer.
- $\text{LayerNorm}_i(\cdot)$ represents the layer normalization operation applied after the transpose convolution in the i-th layer.
- $\text{ReLU}_i(\cdot)$ represents the ReLU [10] activation function applied after the layer normalization in the i-th layer.
- $\tanh(\cdot)$ represents the tanh activation function applied at the end of the model.

The latent vector $\mathbf{z}$ is processed through a series of transpose convolutional layers, each followed by layer normalization. The final output is obtained by applying a tanh activation function to the output of the last layer.

1) Layer Normalization: Following Pytorch's implementation of the DCGAN layer normalization is used to stablize the output from each transpose operation. Layer normalization is described as:

$$\text{LayerNorm}(\mathbf{x}) = \gamma \odot \frac{\mathbf{x} - \mu}{\sigma} + \beta \tag{1}$$

where:

- $\mu = \frac{1}{d}\sum_{i=1}^{d} x_i$ is the mean of the feature map.
- $\sigma = \sqrt{\frac{1}{d}\sum_{i=1}^{d}(x_i - \mu)^2}$ is the standard deviation of the feature map.
- $\gamma = [\gamma_1, \gamma_2, \ldots, \gamma_d]$ is a vector of learnable parameters that scale the normalized values.
- $\beta = [\beta_1, \beta_2, \ldots, \beta_d]$ is a vector of learnable parameters that shift the normalized values.
- $\odot$ denotes element-wise multiplication along each column in the feature map with the vector γ.
- $\mathbf{x} - \mu$ centers the feature map by subtracting the mean.
- $\frac{\mathbf{x}-\mu}{\sigma}$ normalizes the feature map to have unit variance.

Layer normalization is applied independently to each feature map of each data sample, ensuring that each feature map has zero mean and unit variance, with additional learnable parameters γ and β for flexibility.

2) Transpose Convolution: The transpose convolution [11] is used for upscaling the latent vector representation

to a real looking image. To perform the transpose convolution operation, an input matrix is needed, as well as a kernel/filter. Each of these have different parameters. The transpose convolution operation can be described as:

- Input tensor Y with dimensions (C_{in}, H_{in}, W_{in})
- Kernel tensor K with dimensions $(C_{out}, C_{in}, H_f, W_f)$
- Stride s
- Padding p
- Dilation d
- Output tensor X with dimensions $(C_{out}, H_{out}, W_{out})$

where:

- H_{out} is the height of the output feature map,
- W_{out} is the width of the output feature map,
- H_{in} and W_{in} are the height and width of the input feature map,
- H_f and W_f are the height and width of the filter/kernel,
- s is the stride,
- p is the padding, and
- d is the dilation factor,
- C_{in} and C_{out} is the number of input and output channels.

The output dimension after applying the transpose convolution on input feature map can be calculated as:

$$H_{out} = s \cdot (H_{in} - 1) + d \cdot (H_f - 1) + 1 - 2p \quad (1)$$

$$W_{out} = s \cdot (W_{in} - 1) + d \cdot (W_f - 1) + 1 - 2p \quad (2)$$

[11]

The transpose convolution can be described as:

$$X_{c_{out}}(i,j) = \sum_{c_{in}=0}^{C_{in}-1} \sum_{m=0}^{H_f-1} \sum_{n=0}^{W_f-1} Y_{c_{in}} \left(\left\lfloor \frac{i-m+p}{s} \right\rfloor \cdot d, \left\lfloor \frac{j-n+p}{s} \right\rfloor \cdot d \right) \cdot K(c_{in}, c_{out}, m, n) \quad (1)$$

[11]

where:

- $X_{c_{out}}(i,j)$ is the value at position (i,j) in the c_{out}-th channel of the output tensor,
- $Y_{c_{in}}$ is the value at the adjusted position in the c_{in}-th channel of the input tensor,
- $K(c_{in}, c_{out}, m, n)$ is the value of the kernel at position (m,n) connecting the c_{in}-th input channel to the c_{out}-th output channel.
- C_{in} is the number of input channels.
- i and j are the indices representing the spatial position in the output feature map $X_{c_{out}}$.
- m and n are the indices representing the spatial position in the filter/kernel K.
- p is the padding applied to the input feature map.
- s is the stride of the transposed convolution operation.
- d is the dilation factor.

Floor operator values are used due to map indices are integers.

D. The Discriminator

The Discriminator is the second part of the DCGAN model. It consits of input layer (the image) with a Leaky ReLU activation function [12] with a slope s = 0.01. The following n-layers consists of convolution steps (as seen in figure 1) with added batch normalization layers and leaky ReLU activation functions.

It can be described as:

$$\mathbf{h}_0 = \mathbf{x} \quad (1)$$

$$\mathbf{h}_i = \begin{cases} \text{LeakyReLU}\left(\text{Conv}_i(\mathbf{h}_{i-1})\right) & \text{if } i = 1 \\ \text{BatchNorm}_i\left(\text{LeakyReLU}\left(\text{Conv}_i(\mathbf{h}_{i-1})\right)\right) & \text{if } i > 1 \end{cases}$$

$$\text{for } i = 1, 2, \ldots, k-1 \quad (2)$$

$$D(\mathbf{x}) = \sigma\left(\text{Conv}_k(\mathbf{h}_{k-1})\right) \quad (3)$$

where:

- $x \in \mathbb{R}^{n_{\text{channels}} \times 64 \times 64}$ is the input image.
- $D(x)$ is the output of the discriminator given the input image x.
- k is the number of convolutional layers.
- $h_0 = x$ is the initial input to the discriminator.
- h_i represents the output of the i-th layer.
- $\text{Conv}_i(\cdot)$ represents the convolution operation applied in the i-th layer.
- LeakyReLU$(\cdot)$ represents the LeakyReLU activation function with a negative slope of 0.2.

- $\text{BatchNorm}_i(\cdot)$ represents the batch normalization operation applied after the convolution in the i-th layer (omitted for the first and last layers).
- $\sigma(\cdot)$ represents the sigmoid activation function applied at the end of the model.

1) Batch Normalization: Batch normalization [13] involves normalizing the activation's of each layer by subtracting the mean and dividing by the standard deviation of the mini-batch. It's primary usage is to introduce regularization/generalization into the Discriminator model.

Let $\mathbf{x}$ be the input to a layer, and μ and σ be the mean and standard deviation of the mini-batch, respectively. Batch normalization can be represented mathematically as follows:

$$\text{BatchNorm}(\mathbf{x}) = \frac{\mathbf{x} - \mu}{\sqrt{\sigma^2 + \epsilon}} \cdot \gamma + \beta \tag{1}$$

where:

- $\mathbf{x}$ represents the input image or feature maps.
- μ is the mean of the c-th channel across the batch.
- σ^2 is the variance of the c-th channel across the batch.
- ϵ is a small constant added to avoid division by zero.
- γ is a learnable scale parameter.
- β is a learnable shift parameter.

E. Final note on the DCGAN

Weight initialization is implemented as it helps to set the initial values of weights. Proper weight initialization can prevent issues like vanishing or exploding gradients, which can hinder the training process, or as former mention break the model. This is done with respect to Pytorch's implementation of the DCGAN [8].

- For convolutional layers, the weights are initialized from a normal distribution with mean 0 and standard deviation 0.02.
- For batch normalization layers, the weights are initialized to 1 and the biases are initialized to 0, both with a standard deviation of 0.02.

IV. Experiment Setup

A. Sanitizing the data

The first step was to prepare and curate the real-world dataset. In the context of generating images, making sure that we have as pure a dataset as possible is important, as there is an inherent variability in both image quality and content

The dataset is made up of approximately 64k images of archaeological coin artifacts. All of these images are divided into 4 smaller subsets, representing the age of origin for the different artifacts: Iron Age (2k), Medieval (45k), Reformation (13k), and Viking (4k). However, none of these subsets were without their challenges, and these needed to be addressed before proceeding with any further analysis or model training

A manual curation process seemed to be the best course of action, since the process of evaluating the output images can not be generalized without using some form of score that can be obtained by a feature extractor. However, due to the fact that the point of the study is to generate images to put into our own classifier, introducing a classifier simply for the purpose of feature extracting the generated images, seems to go against the study. This process was undertaken to increase the general quality and relevance of the images in the dataset. This meant reviewing each individual image in order to identify and eliminate instances that did not live up to some predetermined criteria. Specifically, the images were assessed loosely based on these factors:

Clarity: Images with low resolution, blurriness, or excessive noise were deemed unsuitable for further analysis, as they hindered the accurate representation and interpretation of the coin artifacts.

Visual Complexity: Images featuring cluttered backgrounds or extraneous objects that detracted from the focus on the coin artifact were excluded to maintain clarity and focus.

Pixelation: Images suffering from significant pixelation, which compromised the integrity and fidelity of the coin artifact representation, were removed from the dataset.

Size of Coin: Images where the coin artifact occupied a negligible portion of the frame, resulting in insufficient detail for meaningful analysis, were excluded to ensure adequate representation and feature extraction.

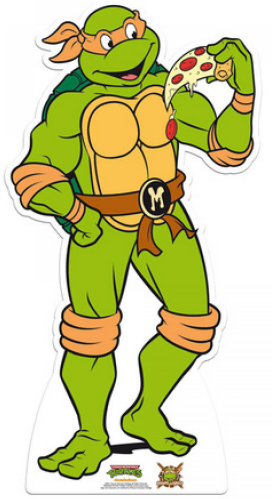
Erosion: Images where the coin artifact is eroded or severely defective.

Relevance: Images that did not depict coin artifacts or were misclassified as such were eliminated to maintain the integrity and authenticity of the dataset.

Following the manual curation process, approximately 5,000 images were identified and removed from the initial dataset. This filtration ensured that only high-quality, relevant images useful to subsequent analyses and model training remained.

The sanitization of the initial dataset lays a robust foundation for the subsequent phases of the study, particularly image generation and classification tasks. By attempting to eliminate noise and irrelevant data, the curated dataset enables more accurate and reliable model training, hopefully enhancing the performance and interpretability of the resulting classifiers.

Below in figure 4, some examples of deleted images are shown. It becomes evident (especially with the Teenage Mutant Ninja Turtle) that some of the images from the original dataset were not supposed to have been included, and will likely have a negative influence on the experiments.



(a) Non-coin example

(b) Busy background example

(c) Low clarity example

(d) Small coin example

Fig. 4: Deleted image examples

B. Data augmentation for GAN training

The Iron Age subset of the Coin subset of the entire DIME metal detector findings dataset, is the smallest natural subset and is a good candidate for this experiment. In an effort to enhance the Iron Age coin dataset (we will call this the Vanilla dataset) and mitigate the small size, before training the GAN, we employed data augmentation and bootstrapping to the existing images. The result of enhancing the dataset was 3 new sets

enhanced with bootstrapped data, augmented data or both.

1) Bootstrapping: The bootstrapping technique involves selecting random images from the original dataset and making copies of these. We went for a factor of 4 in increase in dataset size, meaning that on average every image now appears 4 times in the dataset. In reality the balance is not perfect, and every image will not have been copied the same amount of times. Some will have been copied more than 3 times, and some fewer.

2) Augmenting: In the process of augmenting the data, the images will undergo direct transformation; they will not simply be copied. The transformations applied are horizontal flips, rotations, and random zooms. Each of these augmentations are applied every time to every image but with slight differences. When an image is rotated, is is rotated a random amount of degrees between 0 and 90, and when zooming, between 80% (zooming out) and 150% (zooming in). In figure 5 an example of an augmented image is shown.

Fig. 5: Augmented image

C. Challenges in GAN tuning

There are a variety of different knobs and handles to turn when tuning a General Adversarial Network, in order to enhance the quality and realism of the generated images. The kernels, their sizes, strides, and paddings are important parameters for defining the structure and the operations of the generator and discriminator networks. However, in the context of tweaking these values in order to find the best possible model, they present a challenge due to their direct influence on the dimensions of the produced images; and for this reason we have chosen to not tweak these values while tuning the models in the network. This reduced the architecture's complexity, made sure we always get the correct dimensions for the output images, and lets us focus on other essential hyperparameters. Dynamically resizing each image to the desired dimensions could have been an option. However, the nature of downscaling images inherently means that some image details will be lost. This was avoidable by only generating images with the desired dimensions.

D. Tuning the GAN

Given that the study aims to compare the different ways enriching a small dataset can affect the performance of a CNN classifier, we selected the smallest subset of data that we have, namely the Iron Age coins. To generate realistic Iron Age coin images and address the dataset's small size, we trained multiple Generative Adversarial Networks (GANs), each with distinct training datasets (based on the Iron Age set) and hyperparameter configurations. This section details the tuning process and the selection of the best-performing model based on manual validation of generated images.

We created four variants of the GAN, each trained on a different version of the Iron Age coin dataset:

Dataset	Dataset size	Enrichment method
Vanilla	1700	None
Vanilla + boot	6800	Duplication
Vanilla + aug	3400	Flip, rotation, zoom
Vanilla + aug + boot	8500	All

TABLE I: Datasets used for GAN training

For each GAN variant, we tuned several hyperparameters to optimize the quality of the generated images. The parameters included batch size, the number of feature maps in the discriminator (ndf), the combined learning rate for the network (lr), the latent vector size (nz), and the number of feature maps in the generator (ngf). These parameters were adjusted to find the most stable and efficient training configuration. Batch size influenced how many samples were processed before updating the model's parameters. The ndf controlled the depth of the discriminator, affecting its ability to differentiate real images from generated ones. The learning rate determined the step size at each iteration during training. The latent vector size, sampled from a standard normal distribution, served as the input for the generator. Finally, the ngf determined the depth of the generator network, influencing its capacity to produce high-quality images.

Hyperparameter	Search range / values	Selected value
Batch size	32, 64, 128, 256	128
ndf	32 -128	70
lr	$x \in \mathbb{R} \mid 0.0001 \leqslant x \leqslant 0.001$	0.0005
nz	50, 100, 150, 200	100
ngf	$x \in \mathbb{N} \mid 32 \leqslant x \leqslant 128$	32

TABLE II: GAN tuning search space

We validated the models manually by examining the generated images to assess their realism and fidelity to actual Iron Age coin images. This process was critical in determining the effectiveness of each model configuration.

The best-performing model emerged from the dataset that combined augmented and bootstrapped images. This

configuration yielded the most realistic and detailed coin images, demonstrating the advantages of using a larger training dataset. The augmented and bootstrapped dataset provided the GAN with a rich array of features to learn from, resulting in superior performance compared to models trained on smaller or less varied datasets.

The manual validation process ensured that the selected model not only performed well numerically but also produced visually plausible and high-quality images. This model will be used in further experiments to evaluate its impact on the performance of the CNN classifier for archaeological coin artifacts.

1) Variance Filtering of Generated images: After the GAN model was in place, and generated images were readily available, some measure to filter out the poorest generated images of the generated bunch had to be established, in an attempt to give the classifier the best possible images. The simplest solution that would also yield some tangible results was to calculate the variance of the generated images, and filter out any images with a variance below a certain threshold. Calculating the variance in an image is a measure of the dispersion of pixel values. This gives us a way to quantify how much each pixel value differs from the mean pixel value; a higher variance is an indication of more variability and contrast in the image, while a lower variance indicates a more uniform image. This was the simplest way of filtering out generated images that were more or less black squares, which was the case for a small portion of the generated images. The variance was calculated using the Numpy Python variance function [14], which is calculated as follows:

$$Variance = \frac{1}{MN} \sum_{i=1}^{M} \sum_{j=1}^{N} (I_{ij} - \mu)^2$$

Where:

- M and N are the dimensions of the image (height and width, respectively).
- I_{ij} is the pixel value at position (i, j).
- μ is the mean pixel value of the image.

E. CNN setup

Arguably, the classifiers form the backbone of this project, therefore much attention was put on the tuning and training the CNN models. In order to determine the best hyperparameter configurations unique to each dataset, a systematic approach was adopted with the help of the Ray Tune library [15]. Ray provides a flexible framework for tuning, which allows for the combination of grid and random search strategies over our parameters, as illustrated by Table III.

Hyperparameter	Search range / values	Search strategy
Learning rate	$1x10^{-4}$ to $1x10^{-1}$	log-uniform dist.
Batch size	8, 16, 32, 64	grid search
Epoch #	50, 100, 150	grid search
Conv. kernel size	3, 5, 7	random search
Conv. kernel stride	1, 2, 3	random search
Pooling kernel size	2, 3	random search
Pooling kernel stride	1, 2	random search
FC1 size	128, 256, 512	random search
FC2 size	64, 128, 256	random search

TABLE III: CNN hyperparameter search space

When selecting possible values for the search space we tried to include lower and upper extremes as well as a few options in-between to cover as much ground as possible within the limits of our data. Inspiration for these values was taken from examples and recommendations in the Ray documentation [15]. Generally, this selection process was a balancing act between increasing coverage while keeping the size of the search space as small as possible to retain a reasonable performance.

For the learning rate, we used a log-uniform distribution within the range of $1x10^{-4}$ to $1x10^{-1}$. This distribution is particularly suitable for tuning the learning rate, as it spans several orders of magnitude. This reflects that there can be a large variance between optimal learning rates depending on the characteristics of the datasets. By using a log-uniform distribution, we ensure that each order of magnitude is equally represented in the search space, thus increasing the likelihood of identifying an optimal learning rate.

As for the batch size, we employed a grid search over the values [8, 16, 32, 64], which let's us systematically evaluate the performance of the model at these sizes. This approach ensures that we can find a batch size that balances memory usage and convergence speed effectively.

The number of epochs indicates how many times the entire training dataset passes through the neural network. We used a grid search over the values [50, 100, 150] to determine the optimal number of epochs required for training. This helps in assessing the trade-off between sufficient training and the risk of overfitting.

The size of the convolutional kernel (or filter) impacts the receptive field of the convolutional layers. We tested kernel sizes of 3, 5, and 7 to identify the most effective filter size for capturing features in the data.

The convolutional stride determines the step size by which the filter moves across the input volume. We explored strides of 1, 2, and 3 to find an optimal balance between spatial resolution and computational efficiency.

The pooling kernel size is used in the pooling layers to downsample the feature maps. We tested kernel sizes of 2 and 3 to evaluate their effect on reducing spatial dimensions while preserving important features.

The pooling stride defines the step size for the pooling

operation. We considered strides of 1 and 2 to determine their impact on the downsampling process and the resulting feature map size.

Additionally, the kernel sizes and strides had to be carefully considered from the perspective of the input image dimensions. Namely, the size of 64x64 pixels limited our choices of these parameters, since a larger kernel size combined with a larger stride would quickly reduce the input image to an unusable size.

The size of the first fully connected (FC) layer influences the number of neurons and hence the model's capacity to learn complex representations. We explored sizes of 128, 256, and 512 neurons to identify the optimal network capacity.

The size of the second fully connected layer further refines the model's representational capacity. We tested sizes of 64, 128, and 256 neurons to optimize the architecture of the fully connected layers for better performance.

In order to ensure a thorough exploration of the hyperparameter space we elected to explore the search space three times. This approach adds robustness to our process of hyperparameter tuning by allowing the models to explore different regions of the space multiple times, thereby reducing the risk of missing potentially optimal combinations due to the randomness inherent in the tuning process.

By applying the above outlined tuning strategy to our datasets, we get the best performing CNN hyperparameter configurations over our search space, tailored to the characteristics of each individual dataset. To preserve time, the holdout method was used for splitting the datasets into training and testing sets. While this approach has inherent weaknesses when it comes to achieving maximum performance, that is not an important factor for tuning process itself. So while other methods, such as k-fold cross-validation, might produce overall higher performance numbers, the execution time for the holdout method was deemed a more crucial aspect, and made it our choice for hyperparameter tuning.

The five datasets for which tuning was performed are: vanilla dataset, augmented dataset, bootstrapped dataset, enriched with filtered-generated dataset, enriched with unfiltered-generated dataset. These datasets and the models trained on them form the pillars of our experiments. In order to remain consistent in the naming throughout the experimentation, a short legend is provided in Table IV. "Other artefact" images are always used in a volume that creates a balanced dataset and the Vanilla Iron Age coins are present in every other dataset.

Model name	Coins	Composition
Vanilla	1779	Original iron age coins
Augmented	3558	Augmented coins
Bootstrapped	7116	Bootstrapped coins
First filtered	3558	GAN generated coins, var. fil.
Second filtered	3558	GAN generated coins, var. fil.
Big filtered	5337	First fil. + Second fil.
First unfiltered	3558	GAN generated coins, not fil.
Second unfiltered	3558	GAN generated coins, not fil.
Big unfiltered	5337	First unfil. + Second unfil.

TABLE IV: Model names, dataset compositions

We've included Tables V and VI to illustrate the difference between the number of parameters of two classifiers. The Vanilla model has the lowest amount of parameters, while the Augmented model has the largest. The other models fall in between these two. The difference in number of parameters can be largely attributed to the size of the last two fully connected layers, as shown by Tables VII and VIII.

Name	Value
Total params	1,129,194
Trainable params	1,129,194
Non-trainable params	0
Total mult-adds (M)	6.61
Input size (MB)	0.05
Forward/backward pass size (MB)	0.24
Params size (MB)	4.52
Estimated Total Size (MB)	4.80

TABLE V: Vanilla CNN Parameters

Name	Value
Total params	1,519,674
Trainable params	1,519,674
Non-trainable params	0
Total mult-adds (M)	4.79
Input size (MB)	0.05
Forward/backward pass size (MB)	0.27
Params size (MB)	6.08
Estimated Total Size (MB)	6.39

TABLE VI: Augmented CNN Parameters

TABLE VII: Vanilla CNN Architecture

Layer (type (var_name))	Input Shape	Output Shape	Param #	Param %	Kernel Shape	Mult-Adds	Trainable
LeNet5_Dynamic (LeNet5_Dynamic)	[1, 3, 64, 64]	[1, 2]	–	–	–	–	True
Sequential (conv_layers)	[1, 3, 64, 64]	[1, 16, 11, 11]	–	–	–	–	True
Conv2d (0)	[1, 3, 64, 64]	[1, 6, 58, 58]	888	0.08%	[7, 7]	2,987,232	True
ReLU (1)	[1, 6, 58, 58]	[1, 6, 58, 58]	–	–	–	–	–
MaxPool2d (2)	[1, 6, 58, 58]	[1, 6, 29, 29]	–	–	2	–	–
Conv2d (3)	[1, 6, 29, 29]	[1, 16, 23, 23]	4,720	0.42%	[7, 7]	2,496,880	True
ReLU (4)	[1, 16, 23, 23]	[1, 16, 23, 23]	–	–	–	–	–
MaxPool2d (5)	[1, 16, 23, 23]	[1, 16, 11, 11]	–	–	2	–	–
Sequential (fc_layers)	[1, 1936]	[1, 2]	–	–	–	–	True
Linear (0)	[1, 1936]	[1, 512]	991,744	87.83%	–	991,744	True
ReLU (1)	[1, 512]	[1, 512]	–	–	–	–	–
Linear (2)	[1, 512]	[1, 256]	131,328	11.63%	–	131,328	True
ReLU (3)	[1, 256]	[1, 256]	–	–	–	–	–
Linear (4)	[1, 256]	[1, 2]	514	0.05%	–	514	True
Softmax (5)	[1, 2]	[1, 2]	–	–	–	–	–

TABLE VIII: Augmented CNN Architecture

Layer (type (var_name))	Input Shape	Output Shape	Param #	Param %	Kernel Shape	Mult-Adds	Trainable
LeNet5_Dynamic (LeNet5_Dynamic)	[1, 3, 64, 64]	[1, 2]	–	–	–	–	True
Sequential (conv_layers)	[1, 3, 64, 64]	[1, 16, 13, 13]	–	–	–	–	True
Conv2d (0)	[1, 3, 64, 64]	[1, 6, 60, 60]	456	0.03%	[5, 5]	1,641,600	True
ReLU (1)	[1, 6, 60, 60]	[1, 6, 60, 60]	–	–	–	–	–
MaxPool2d (2)	[1, 6, 60, 60]	[1, 6, 30, 30]	–	–	2	–	–
Conv2d (3)	[1, 6, 30, 30]	[1, 16, 26, 26]	2,416	0.16%	[5, 5]	1,633,216	True
ReLU (4)	[1, 16, 26, 26]	[1, 16, 26, 26]	–	–	–	–	–
MaxPool2d (5)	[1, 16, 26, 26]	[1, 16, 13, 13]	–	–	2	–	–
Sequential (fc_layers)	[1, 2704]	[1, 2]	–	–	–	–	True
Linear (0)	[1, 2704]	[1, 512]	1,384,960	91.14%	–	1,384,960	True
ReLU (1)	[1, 512]	[1, 512]	–	–	–	–	–
Linear (2)	[1, 512]	[1, 256]	131,328	8.64%	–	131,328	True
ReLU (3)	[1, 256]	[1, 256]	–	–	–	–	–
Linear (4)	[1, 256]	[1, 2]	514	0.03%	–	514	True
Softmax (5)	[1, 2]	[1, 2]	–	–	–	–	–

TABLE IX: DCGAN Discriminator Architecture and Number of Parameters

Layer (type:depth-idx)	Input Shape	Output Shape	Param #	Kernel Shape	Mult-Adds
D	[128, 3, 64, 64]	[128, 1, 1, 1]	–	–	–
Sequential: 1-1	[128, 3, 64, 64]	[128, 1, 1, 1]	–	–	–
Conv2d: 2-1	[128, 3, 64, 64]	[128, 64, 32, 32]	3,072	[4, 4]	402,653,184
LeakyReLU: 2-2	[128, 64, 32, 32]	[128, 64, 32, 32]	–	–	–
Conv2d: 2-3	[128, 64, 32, 32]	[128, 128, 16, 16]	131,072	[4, 4]	4,294,967,296
BatchNorm2d: 2-4	[128, 128, 16, 16]	[128, 128, 16, 16]	256	–	32,768
LeakyReLU: 2-5	[128, 128, 16, 16]	[128, 128, 16, 16]	–	–	–
Conv2d: 2-6	[128, 128, 16, 16]	[128, 256, 8, 8]	524,288	[4, 4]	4,294,967,296
BatchNorm2d: 2-7	[128, 256, 8, 8]	[128, 256, 8, 8]	512	–	65,536
LeakyReLU: 2-8	[128, 256, 8, 8]	[128, 256, 8, 8]	–	–	–
Conv2d: 2-9	[128, 256, 8, 8]	[128, 512, 4, 4]	2,097,152	[4, 4]	4,294,967,296
BatchNorm2d: 2-10	[128, 512, 4, 4]	[128, 512, 4, 4]	1,024	–	131,072
LeakyReLU: 2-11	[128, 512, 4, 4]	[128, 512, 4, 4]	–	–	–
Conv2d: 2-12	[128, 512, 4, 4]	[128, 1, 1, 1]	8,192	[4, 4]	1,048,576
Sigmoid: 2-13	[128, 1, 1, 1]	[128, 1, 1, 1]	–	–	–

Discriminator summary statistics (Table IX):
Total params: 2,765,568
Trainable params: 2,765,568
Non-trainable params: 0
Total mult-adds (Units.GIGABYTES): 13.29

Input size (MB): 6.29
Forward/backward pass size (MB): 184.55
Params size (MB): 11.06
Estimated Total Size (MB): 201.90

Generator summary statistics (Table X):
Total params: 3,576,704
Trainable params: 3,576,704

TABLE X: DCGAN Generator Architecture and Number of Parameters

Layer (type:depth-idx)	Input Shape	Output Shape	Param #	Kernel Shape	Mult-Adds
G	[128, 100, 1, 1]	[128, 3, 64, 64]	–	–	–
Sequential: 1-1	[128, 100, 1, 1]	[128, 3, 64, 64]	–	–	–
ConvTranspose2d: 2-1	[128, 100, 1, 1]	[128, 512, 4, 4]	819,200	[4, 4]	1,677,721,600
BatchNorm2d: 2-2	[128, 512, 4, 4]	[128, 512, 4, 4]	1,024	–	131,072
ReLU: 2-3	[128, 512, 4, 4]	[128, 512, 4, 4]	–	–	–
ConvTranspose2d: 2-4	[128, 512, 4, 4]	[128, 256, 8, 8]	2,097,152	[4, 4]	17,179,869,184
BatchNorm2d: 2-5	[128, 256, 8, 8]	[128, 256, 8, 8]	512	–	65,536
ReLU: 2-6	[128, 256, 8, 8]	[128, 256, 8, 8]	–	–	–
ConvTranspose2d: 2-7	[128, 256, 8, 8]	[128, 128, 16, 16]	524,288	[4, 4]	17,179,869,184
BatchNorm2d: 2-8	[128, 128, 16, 16]	[128, 128, 16, 16]	256	–	32,768
ReLU: 2-9	[128, 128, 16, 16]	[128, 128, 16, 16]	–	–	–
ConvTranspose2d: 2-10	[128, 128, 16, 16]	[128, 64, 32, 32]	131,072	[4, 4]	17,179,869,184
BatchNorm2d: 2-11	[128, 64, 32, 32]	[128, 64, 32, 32]	128	–	16,384
ReLU: 2-12	[128, 64, 32, 32]	[128, 64, 32, 32]	–	–	–
ConvTranspose2d: 2-13	[128, 64, 32, 32]	[128, 3, 64, 64]	3,072	[4, 4]	1,610,612,736
Tanh: 2-14	[128, 3, 64, 64]	[128, 3, 64, 64]	–	–	–

Non-trainable params: 0
Total mult-adds (Units.GIGABYTES): 54.83

Input size (MB): 0.05
Forward/backward pass size (MB): 264.24
Params size (MB): 14.31
Estimated Total Size (MB): 278.60

The DCGAN model architecture and number of parameters for both the discriminator and generator is shown in table IX and X respectively. During training the Discriminator network processes inputs of shape $[128, 3, 64, 64]$ (batch size, channels, height, width) and outputs a prediction of shape $[128, 1, 1, 1]$. During training the Generator network takes inputs of shape $[128, 100, 1, 1]$ and generates images of shape $[128, 3, 64, 64]$ (batch size, channels, height, width).

Summary statistics explanation:

- **Total params**: Total number of trainable parameters in the Discriminator network.
- **Trainable params**: Parameters that are updated during training to minimize the network's loss function.
- **Non-trainable params**: Parameters that remain fixed during training, typically associated with normalization layers.
- **Total mult-adds (Units.GIGABYTES)**: Total number of multiply-accumulate operations required during inference, measured in gigabytes.
- **Input size (MB)**: Size of the input data in megabytes.
- **Forward/backward pass size (MB)**: Memory required for storing activations during forward and backward passes.
- **Params size (MB)**: Memory occupied by the model parameters.
- **Estimated Total Size (MB)**: Approximate total memory footprint of the model during inference.

V. Experiment

The training and evaluation of the CNN models were done on a computer equipped with 16Gb RAM, an AMD Ryzen 5 5600X 6-core CPU, and an NVIDIA GeForce RTX 3070 GPU. The training time was approximately 180 minutes per classifier, while the evaluation took an insignificant amount of time, between 15-20 seconds.

The training of the DCGAN model was done on AAU Claaudia, an AI Cloud [16] server equipped with 980Gb RAM of which 256Gb were used, 8 Nvidia A800 Graphic cards from which 4 were used and an AMD EPYC CPU with 128 cores, of which 64 cores were utilized.

A. Evaluation Metrics

During the experiments, the following evaluation metrics were utilized for assessing the performance of the Convolutional Neural Network (CNN) classifier:

- **Accuracy**: Measures the overall correctness of the classifier's predictions.
- **Precision**: Calculates the amount of true positive predictions among all positive predictions made by the classifier.
- **Recall**: Measures the proportion of true positive predictions among all actual positive instances in the dataset.
- **F1 Score**: The harmonic mean of precision and recall, providing a balanced assessment of the classifier's performance.
- **ROC AUC**: Summarizes the classifier's ability to discriminate between the positive and negative classes across different thresholds.

B. Evaluating DCGAN

To assess the performance of the DCGAN during training, the Binary Cross Entropy loss function was utilized, with a mean reduction [17]. This loss function is well-suited for binary classification tasks, such as distinguishing between real and generated images in the context of the adversarial training framework. The Binary Cross Entropy (BCE) Loss for a batch of N samples is:

$$\text{BCE}(\hat{y}, y) = \frac{1}{N} \sum_{i=1}^{N} \left[-y_i \log(\hat{y}_i) - (1 - y_i) \log(1 - \hat{y}_i) \right] \tag{1}$$

where:

- y_i is the true label for the i-th sample (0 or 1),
- $\hat{y}_i$ is the predicted probability for the i-th sample (a value between 0 and 1),
- N is the number of samples in a batch.

Batches of generated images were used for different runs in the experiments of the CNN classifier. These batches of generated images are then run through the variance filter IV-D1. The following images in figure 6 is a small subset not removed from the variance filter. The darkness in color is due to the normalized pixel values in the tanh activation function in the generator.

Cross Validation Experiment Results with CNN classifier: Performing the Cross Validation with baseline models using the original datasets were evaluated with a 3-, 5- and 10-fold as shown in table XI. This was done to ensure the models' performances were not dependent on a specific subset of data.

TABLE XI: Performance Metrics for Vanilla Models with Different K-Folds

Metric	3-fold	5-fold	10-fold
Accuracy	0.7038	0.7198	0.7166
Precision	0.7014	0.7097	0.7185
Recall	0.7116	0.7442	0.7161
F1 Score	0.7061	0.7264	0.7168
ROC AUC	0.7038	0.7198	0.7175

The results shows that the performance for the models is consistent across different cross-validations, with only as small difference.

Since the model does not show significant changes in performance with different k-folds, the rest of the experiments are done with k = 3.

TABLE XII: Performance Metrics for Augmented Data

Metric	Augmented
Accuracy	0.7496
Precision	0.7367
Recall	0.7782
F1 Score	0.7567
ROC AUC	0.7496

The impact of using augmented data the model's performance increased across all metrics. The augmentation results which are shown in table XII, led to significant performance increase and particular in recall and F1 score. Since the dataset is balanced, an accuracy of $0.7496 \approx 75\%$ for a true positive result. With a precision around $\approx 74\%$, indicates a low false positive rate. The recall is $\approx 78\%$ which means the model has a low false negative rate. The F1 score of $\approx 76\%$ suggests a good balance between Precision and Recall. The ROC AUC with $\approx 75\%$ suggest a 3/4 discrimination success between true and false samples.

Using the bootstrap suggest a better performance in the metrics as shown in table XIII. *However*, each sample in the bootstrap dataset is possibly duplicated multiple times, and therefore each sample might appear in multiple folds (i.e. a data leak), which can make the

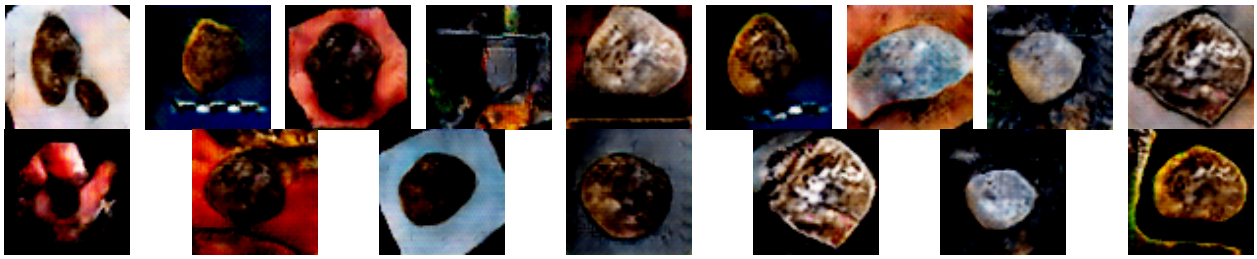


Fig. 6: Small selection of some generated images from the best performing DCGAN. Normalized values (darker) due to tanh activation function in the output layer of the generator.

TABLE XIII: Performance Metrics for Bootstrapped Data

Metric	Bootstrapped
Accuracy	0.8939
Precision	0.8902
Recall	0.8988
F1 Score	0.8944
ROC AUC	0.8939

model overfit to specific data and data features. Therefore these results does not provide the full picture of the model performance.

The performance metrics for models trained and validated with generated data indicate notable improvements as seen in table XIV:

TABLE XIV: Performance Metrics for Generated Data

Metric	Generated 2 Runs	Generated Big Run
Accuracy	0.7867	0.8202
Precision	0.7536	0.8158
Recall	0.8555	0.8272
F1 Score	0.8004	0.8212
ROC AUC	0.7867	0.8202

Increasing the number of generated images in the training dataset leads to improved performance metrics, particularly in the Generated Big Run scenario.

The result hereby shows improvement in both scenarios, but much more consistent with the Generated Big Run.

C. Testing models on unseen data

After the models have been trained and validated with the 3-fold cross validation, the results tell us how the models are performing. However, due to the nature of cross validation and the enriched datasets, the results are not compatible for a direct comparison. In the case of the model trained and validated on the bootstrapped dataset, the validation has image leaks. The entire bootstrapped dataset was split into 3 folds which, in any case, will include duplicates in the fold that is being used for validation. This means that the validation set is not comprised of entirely unseen images. Most likely the opposite, meaning that the model is already fitted to recognize the images in the validation set. In the case of the model trained on the augmented dataset, the evaluation of the model, in reality, does not tell us how the model performs on the original Iron Age coin images. It tells us how the model performs on a combination of the original images, and the augmented variants, since a vanilla image and its augmented version could be in two different folds. To make an experiment that allows for direct comparison on performance, on unseen images, through all of the trained models, a static test set comprising coin images from another subset, the Viking Age coins, will be used. This can be done as the image quality and content across these two datasets are more or less identical. In figure 7 and 8 are a few examples from each of the two subsets, as to illustrate their similarities.

Fig. 7: Iron Age coins

Fig. 8: Viking Age coins

This test was performed on every single model mentioned in Table IV. The Viking Age subset contains a little over 3k images, and each of these models were tested with 3 different test sets, each containing 1k images from the Viking Age set, and 1k from the ”other” set. This means that each model was tested on 3 unique sets with a 50/50 split of coins and ”other”. The results of these tests are presented below:

Metric	Test Set 1	Test Set 2	Test Set 3
Accuracy	0.5975	0.6035	0.5685
Precision	0.5815	0.5833	0.5545
Recall	0.6960	0.7250	0.6970
F1 Score	0.6336	0.6465	0.6176
ROC AUC	0.5975	0.6035	0.5685

TABLE XV: Vanilla Model Performance

Metric	Test Set 1	Test Set 2	Test Set 3
Accuracy	0.5850	0.5845	0.5750
Precision	0.5602	0.5609	0.5530
Recall	0.7910	0.7780	0.7820
F1 Score	0.6559	0.6519	0.6479
ROC AUC	0.5850	0.5845	0.5750

TABLE XVI: Augmented Model Performance

Metric	Test Set 1	Test Set 2	Test Set 3
Accuracy	0.5775	0.5810	0.5805
Precision	0.5479	0.5482	0.5485
Recall	0.8870	0.9210	0.9110
F1 Score	0.6774	0.6873	0.6847
ROC AUC	0.5775	0.5810	0.5805

TABLE XVII: Bootstrapped Model Performance

Metric	Test Set 1	Test Set 2	Test Set 3
Accuracy	0.5615	0.5435	0.5615
Precision	0.5398	0.5252	0.5419
Recall	0.8350	0.9060	0.7960
F1 Score	0.6557	0.6650	0.6448
ROC AUC	0.5615	0.5435	0.5615

TABLE XVIII: First Filtered Model Performance

Metric	Test Set 1	Test Set 2	Test Set 3
Accuracy	0.6105	0.5760	0.5740
Precision	0.5876	0.5553	0.5558
Recall	0.7410	0.7630	0.7370
F1 Score	0.6555	0.6428	0.6337
ROC AUC	0.6105	0.5760	0.5740

TABLE XIX: Second Filtered Model Performance

Metric	Test Set 1	Test Set 2	Test Set 3
Accuracy	0.5645	0.5775	0.5500
Precision	0.5394	0.5533	0.5329
Recall	0.8820	0.8050	0.8110
F1 Score	0.6694	0.6558	0.6431
ROC AUC	0.5645	0.5775	0.5500

TABLE XX: Big Filtered Model Performance

Metric	Test Set 1	Test Set 2	Test Set 3
Accuracy	0.5000	0.5000	0.5030
Precision	0.5000	0.5000	0.5015
Recall	1.0000	1.0000	0.9890
F1 Score	0.6667	0.6667	0.6655
ROC AUC	0.5000	0.5000	0.5030

TABLE XXI: First Unfiltered Model Performance

Metric	Test Set 1	Test Set 2	Test Set 3
Accuracy	0.5000	0.5000	0.5000
Precision	1.0000	1.0000	1.0000
Recall	0.0000	0.0000	0.0000
F1 Score	0.0000	0.0000	0.0000
ROC AUC	0.5000	0.5000	0.5000

TABLE XXII: Second Unfiltered Model Performance

Metric	Test Set 1	Test Set 2	Test Set 3
Accuracy	0.4985	0.5000	0.5115
Precision	0.0000	1.0000	0.5059
Recall	0.0000	0.0000	0.9840
F1 Score	0.0000	0.0000	0.6683
ROC AUC	0.4985	0.5000	0.5115

TABLE XXIII: Big Unfiltered Model Performance

Metric	Test Set 1	Test Set 2	Test Set 3
Accuracy	0.5895	0.6045	0.5925
Precision	0.5749	0.5888	0.5778
Recall	0.6870	0.6930	0.6870
F1 Score	0.6260	0.6367	0.6277
ROC AUC	0.5895	0.6045	0.5925

TABLE XXIV: Noisy Vanilla Model Performance

At a quick glance over all of these models' results, it seems that they are performing similarly. Notably, they very best accuracy achieved was done with the model in table XIX, one of the models that was trained on a dataset enriched with filtered synthetic images.

1) Consistency Across Tests: Most of the models exhibited a degree of consistency in their respective performance metrics. The vanilla model in Table XV demonstrates a relatively stable accuracy and AUC ROC score through the 3 test sets, suggesting that the model trained on the baseline dataset is generalizing fairly well. The consistency is an indication that the model is not too dependant on the specific characteristics of the data it was trained on, nor on any specific characteristics from any of the test sets. On the other hand some models, such as the model seen in table XXII exhibits extreme values with a perfect precision but zero recall, highlighting some issues in training, likely introduced by the non-filtered dataset. This model does not learn anything.

2) Precision and Recall Dynamics: We can observe that several models, those in Tables XVI, XVII, XVIII, XIX, XX, and XXI have a notably higher recall compared to precision. This is a trend that seems to suggest that these models are more likely to identify as many positive instances ("other" label) as possible, even when it will result in false positives. This can be seen as a slight problem in the case of a precision task like classification, and it seems that the enriched datasets has the effect of making the models lean towards the positive instance.

3) F1 score and Balanced Performance: The F1 score that balances precision and recall gives a more holistic view of the model. Given that no model really breaks through the 0.6 AUC ROC score but still most models have similar values, shows a fairly balanced approach to managing both positive and negative instances, while showing that there is room for improvement.

4) Outliers and Anomalies: Several models exhibited extreme behavior. Looking at the models in Tables XXI, XXII, and XXIII we see either precision at 1 or recall at 1, or both at 0. This clearly indicates that somehow the unfiltered GAN generated datasets completely throws off the models and makes them predict solely one label over the other. This seems to indicate the importance of filtering out the poor images from the generated datasets.

5) General Trends: Looking at all of the models, the accuracy typically ranged from 0.5-0.6 which reflects a poor to moderate performance, and the AUC ROC in relation to the accuracy indicates that the models have a moderate ability to distinguish the two labels, while leaning towards "other". The trend of the moderate accuracy and ROC AUC suggests that there is room for improvement, but also that enriching the datasets with synthetic data could in theory be a valid option for increasing the performance of the classifiers.

VI. Discussion

As previously mentioned, the kernel size specifies the size of the filter that is used for convolution. Changing the dimensions of the kernel means varying the receptive field, which in turn will affect the amount of the image an individual convolution operation will capture. Increasing the dimensions of the kernel means capturing more spatial context in any one operation, but it will also lower the resolution of the output image. On the other hand, smaller kernels will have smaller receptive fields, and may be able to focus on finer details in the image but it will increase the resolution of the output image. In addition to the size, the stride of the kernel defines the skip size of the filter when in moves across the image. This means that bigger strides will result in less convolutional operations, which means a smaller resolution in the output image, while lower strides maintains higher image resolutions. Padding, adding blank space to the image during convolution will also have an effect on the output image's resolution. Adding a thicker padding will mean a smaller output resolution, and vice versa. Incorrect kernel sizes, strides or padding will introduce unwanted image sizes that, without further dynamic resizing, will make the output images useless for our application. And given that the output images will act as part of a dataset that will be used on a classifier that is made to interpret 64 by 64 images, we will need our generator to generate this exact size, and our discriminator to accept this same size, in an effort to not lose any details with subsequent image size changes.

A critical issue that was encountered during the process of the 3-fold cross validation on the model trained on the bootstrapped dataset, was data leakage. Data leakage happens when images from outside of the training dataset inadvertently influences the model. This can lead to overly optimistic results. In our case the bootstrapping process in combination with the 3-fold cross validation introduced a significant risk of data leakage. As mentioned, bootstrapping is a technique used to "puff up" a small dataset by duplicating the existing images. The fact that identical images appear multiple times in the dataset makes th 3-fold technique risky. The folds that are made should be mutually exclusive as to provide an unbiased evaluation of the model. However, due to the bootstrapping process it is highly likely that identical images will appear in both the training and the testing folds. This unfortunate overlap results in the model encountering the same images during training and validation. In an attempt to mitigate data leakage, future work should consider alternative strategies for validation, that will avoid the pitfalls of bootstrapping, such as a holdout set, like the Viking set we validated all models on.

There is no leak in the cross validation tests including the filtered GAN generated images, however we see a noticeable drop in performance from the 3-fold cross validation test to the test on unseen viking coins. This drop in performance could be an indication that the coins from the Viking Age set and the coins from the Iron Age set may not be as similar as first assumed.

VII. Conclusion

This research explored the promising application of Deep Convolutional Generative Adversarial Networks (DCGANs) to address the challenges in artifact classification of ancient coins, particularly focusing on data imbalance and small dataset issues. By leveraging the full DIME database, which includes a diverse collection of coin images from various historical periods, we aimed to synthesize new coin images to enrich datasets with fewer examples, namely the Iron Age coins dataset. Our primary goal was to investigate the impact of these synthetic images on the performance of a Convolutional Neural Network (CNN) classifier in comparison to other dataset augmentation techniques.

Throughout the study, several key methodologies were employed, including bootstrapping, data augmentation techniques, and the innovative use of DCGANs for generating synthetic images. Each approach was assessed based on its ability to improve the accuracy and robustness of the CNN classifiers trained on the respective augmented datasets. The classifiers were validated and compared using a holdout set of Viking Age coins, which were deemed to be sufficiently similar to the Iron Age coins, ensuring a reliable evaluation of their performance.

The results of our experiments indicate that augmenting the dataset with synthetic images generated by DCGANs can indeed enhance the performance of CNN classifiers. The synthesized images contributed to mitigating the issues related to data imbalance and small sample sizes, leading to improved classification accuracy. *However*, the degree of improvement varied across different augmentation methods, highlighting the need for further investigation.

Despite the promising findings, our study also revealed several limitations and areas for future research. The variability in the performance gains suggests that the effectiveness of synthetic data augmentation may be heavily context-dependent, influenced by factors such as the specific characteristics of the coin images and the inherent complexity of the classification task. Additionally, the current study focused on binary classification, leaving room for exploration in multi-class scenarios, which are more reflective of real-world applications.

To build on this work, we recommend conducting more extensive experiments with larger and more diverse datasets, incorporating multi-class classification

tasks, and exploring advanced GAN architectures and training techniques. Since this study strictly only used the smallest subset of coin images found in DIME, the performance of our GAN was severely limited. If time allowed, every coin subset should be used for training the GAN and CNN, to get a higher level view of the impact of the dataset size. Another approach worth pursuing is generating a significantly higher number of synthetic images combined with a more targeted and fine-tuned variance filtering approach. Focusing on a more rigorous and robust filtering technique combined with a larger pool of synthetic images to choose from would allow for a more meaningful enrichment process of the original datasets. Additionally, exploring the effects of using higher resolution images could yield interesting results, as the 64x64 images used in this study might have lost important features, affecting the performance of the GAN and classifier both. While this choice was made out of necessity in our case, if more time and compute resources are available, increasing the resolution of the images becomes a viable avenue to pursue. Future research should also investigate the integration of synthetic data with other data augmentation strategies to further enhance model robustness and generalization.

In conclusion, this research underscores the potential of using DCGANs for dataset enrichment in the field of archaeological artifact classification. While the approach shows promise, our findings emphasize the necessity for ongoing experimentation and refinement to fully harness the benefits of synthetic data augmentation. Through continued efforts, we aim to contribute to the broader discussion on deploying artificial intelligence in archaeological research, ultimately advancing the classification and study of historically significant artifacts.

REFERENCES

1 (2022) Dime metal detektor fund. [Online]. Available: https://www.metaldetektorfund.dk/ny/

2 "Solving class imbalance problem in cnn," https://medium.com/x8-the-ai-community/solving-class-imbalance-problem-in-cnn-9c7a5231c478.

3 "Intuitive conlolution," https://betterexplained.com/articles/intuitive-convolution/.

4 3Blue1Brown, "But what is a convolution?" 2017, [Online; accessed 29-May-2024]. [Online]. Available: https://www.youtube.com/watch?v=KuXjwB4LzSA&t=898s

5 Gu, J., Wang, Z., Kuen, J., Ma, L., Shahroudy, A., Shuai, B., Liu, T., Wang, X., Wang, G., Cai, J., and Chen, T., "Recent advances in convolutional neural networks," *Pattern Recognition*, vol. 77, pp. 354–377, 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0031320317304120

6 Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., and Bengio, Y., "Generative adversarial nets," 2014, accessed: 2024-06-06. [Online]. Available: https://arxiv.org/pdf/1406.2661

7 Radford, A., Metz, L., and Chintala, S., "Unsupervised representation learning with deep convolutional generative adversarial networks," 2015. [Online]. Available: https://arxiv.org/pdf/1511.06434

8 PyTorch, "Deep convolutional generative adversarial network tutorial," 2023. [Online]. Available: https://pytorch.org/tutorials/beginner/dcgan_faces_tutorial.html

9 PyTorch. (2024) torch.nn.tanh. [Online]. Available: https://pytorch.org/docs/stable/generated/torch.nn.Tanh.html

10 ——. (2024) torch.nn.relu. Accessed: 2024-06-06. [Online]. Available: https://pytorch.org/docs/stable/generated/torch.nn.ReLU.html

11 PyTorch, "torch.nn.convtranspose2d," 2023. [Online]. Available: https://pytorch.org/docs/stable/generated/torch.nn.ConvTranspose2d.html

12 PyTorch. (2024) torch.nn.leakyrelu. [Online]. Available: https://pytorch.org/docs/stable/generated/torch.nn.LeakyReLU.html

13 PyTorch, "torch.nn.batchnorm2d," 2023, accessed: 2024-06-06. [Online]. Available: https://pytorch.org/docs/stable/generated/torch.nn.BatchNorm2d.html

14 "Numpy variance," ://numpy.org/doc/stable/reference/generated/numpy.var.html/.

15 Ray. (2024) Ray tune: Hyperparameter tuning. Accessed: 2024-06-06. [Online]. Available: https://docs.ray.io/en/latest/tune/index.html

16 University, A. (2024) Ai cloud documentation. [Online]. Available: https://aicloud-docs.claaudia.aau.dk/introduction/

17 PyTorch. (2024) torch.nn.bceloss. [Online]. Available: https://pytorch.org/docs/stable/generated/torch.nn.BCELoss.html