ERASMUS UNIVERSITY ROTTERDAM
Erasmus School of Economics
Master Thesis Econometrics and Management Science
Business Analytics and Quantitative Marketing

Hierarchical AC-GAN: Incorporating the Relations between Classes into the GAN's Image Generation Process using Node2Vec to Aid in the Detection of COVID-19 in Chest X-rays

Student: (Ruth) R.A. Smalbraak, 453014
Supervisor: Dr. (Kathrin) K. Gruber
Second Assessor: Dr. (Paul) P.C. Bouman
Date: April 28, 2022

Abstract

The COVID-19 pandemic has a lasting effect on the global economy and population's health. A main concern in detecting COVID-19 with deep learning methods such as convolutional neural networks (CNNs) remains the lack of datasets of sufficient size, which often leads to overfitting of the CNNs. In this research, we propose a customized Auxiliary Classifier GAN (AC-GAN) to synthesize chest X-rays (CXRs) that enlarge and balance the original dataset to aid in detecting COVID-19 in CXRs. AC-GAN models images belonging to different classes independently. However, classes can often be organized hierarchically, and their hierarchical relations contain valuable information which could provide useful clues for generating high-quality images. We therefore introduce Hierarchical AC-GAN (HAC-GAN), which aims to improve the quality and diversity of synthesized images by leveraging the relations between classes in the form of a hierarchy/graph in the GAN. We propose the Node2Vec graph embedding method to learn hierarchy-aware embeddings, which are fed to HAC-GAN to synthesize fake images. Experiments on three datasets show the effectiveness of HAC-GAN and demonstrate that it generates images that resemble the original dataset more closely than AC-GAN while producing a more diverse set of images. After augmenting the dataset with images from HAC-GAN, we employ deep transfer learning (DTL) to fine-tune several pre-trained CNNs for detecting COVID-19. The findings of this research reveal that the CNNs can accurately detect COVID-19 when trained on local data that is augmented by HAC-GAN. ResNet-18 performs best, and its accuracy increases to 98.5% after training on the augmented dataset. To leverage the potential of the proposed methods in practice, HAC-GAN and the CNNs need to be trained on local/"in-hospital" datasets.

Contents

1 Introduction

COVID-19 is a mild to severe acute respiratory disease caused by the RNA virus SARS-CoV-2. Since its outbreak in December 2019, almost half a billion cases have been reported, resulting in approximately 6.2 million deaths as of April 2022 (World Health Organization, 2021). Even though a vaccine has been developed, the pandemic continues to have a lasting effect on the health of the global population and the worldwide economy. Besides vaccinating the global population, one of the most efficient ways of protection against COVID-19 is to reduce the spread of the virus by detecting and isolating infected individuals.

To detect COVID-19, reverse transcriptase-polymerase chain reaction (RT-PCR) testing has been widely used (Corman et al., 2020). However, RT-PCR test results are found to be variable, potentially unstable, and time-consuming (Li et al., 2020). A complementary and reliable screening method for COVID-19 detection is radiography examination, where chest X-rays (CXR) and computed tomography (CT) scans are visually analyzed for indicators associated with a COVID-19 infection. Early studies suggested that characteristic abnormalities in chest radiography images can be detected in patients infected with COVID-19 (C. Huang et al., 2020; Guan et al., 2020). Diagnosing via CXRs is faster and has greater availability than employing CT scans, which makes screening via CXR images a good complement to viral testing (L. Wang, Lin, & Wong, 2020).

One of the biggest challenges of using CXRs for the detection of COVID-19 is the need for radiologists to interpret the scans themselves. This is a time-consuming process that is prone to errors. Therefore, methods that could aid these radiologists in analyzing CXR scans would be very helpful. Motivated by the need to develop solutions to support the fight against COVID-19 and to aid radiologists in this task, multiple deep learning methods that classify CXRs have been proposed recently (Minaee, Kafieh, Sonka, Yazdani, & Soufi, 2020; Narin, Kaya, & Pamuk, 2021; Khobahi, Agarwal, & Soltanalian, 2020). These deep learning networks train on labeled data and tune a huge amount of parameters, which causes them to easily overfit when trained on limited datasets. However, large and varying CXR datasets are difficult to obtain as the COVID-19 outbreak is recent. Minaee et al. (2020), Hall, Paul, Goldgof, and Goldgof (2020), and Rajaraman et al. (2020) state that one of the biggest challenges in detecting COVID-19 with deep learning methods is the presence of small datasets, and they highlight the need to obtain more data samples.

Data augmentation methods can be employed to alleviate the problem of small datasets. Generative Adversarial Networks (GANs), introduced by Goodfellow et al. (2020), are a class of deep

learning methods that generate synthetic images by learning the data distribution of the original dataset. A GAN employs two neural networks that compete with each other; the generator and the discriminator. The generator's objective is to synthesize fake samples that resemble real data, while the objective of the discriminator is to distinguish between the real and generated samples. However, GANs are found to have an unstable training process (Salimans et al., 2016). Mode collapse is a common problem where the generator fails to learn the full data generating distribution and only synthesizes samples with little variety. Another issue with GANs is the vanishing gradient problem. During training of the GAN, gradients are backpropagated from the final layer to the first layer and get increasingly smaller with each layer due to repetitive application of the chain rule. The gradient can become so small that the weights of the initial layers are not updated, and these layers stop learning entirely. Training a stable GAN that generates high-quality and diverse images is thus very challenging. As a result, many GAN variants have been proposed in the literature that aim to stabilize the training process. A conditional GAN (CGAN) stabilizes training and improves the quality of the images by conditioning on outside information, such as class labels (Mirza & Osindero, 2014). An auxiliary classifier GAN (AC-GAN) is similar to a CGAN but transforms the discriminator to predict the class instead of receiving it as an input, which further stabilizes training and improves the quality of generated images (Odena, Olah, & Shlens, 2017).

Notwithstanding their challenges, GANs can be applied to a wide range of fields and have been successful in generating medical images. Since the outbreak of COVID-19, several GANs have been introduced to generate COVID-19 CXR images (Loey, Smarandache, & M Khalifa, 2020; Waheed et al., 2020; Al-Shargabi, Alshobaili, Alabdulatif, & Alrobah, 2021). This research proposes a customized AC-GAN to generate synthetic CXR images for the COVID-19 pneumonia, non-COVID-19 pneumonia, and normal classes. We employ an AC-GAN because this model can produce high-quality images without bearing a high computational budget relative to other GANs.

An AC-GAN models images that belong to different classes independently. However, it is common to have some sort of relation between classes in real-world datasets, and the classes can often be organized hierarchically based on these relations. Just as incorporating class-conditional information in CGAN and AC-GAN improves the quality of synthesized images (Goodfellow, 2016), incorporating relations between classes could further improve a GAN's performance. The class hierarchy contains valuable information which could provide useful clues that help the GAN generate high-quality images. Therefore, we introduce Hierarchical AC-GAN (HAC-GAN), which aims to

further stabilize the training process and improve the quality and diversity of the generated images by incorporating class hierarchies (in the form of a graph) in the GAN. We use the popular graph embedding method Node2Vec to learn class embeddings that capture the hierarchical relations between those classes (Grover & Leskovec, 2016). The learned embeddings for each class of interest are fed to the generator, which then synthesizes images for each class.

After generating images with (H)AC-GAN to create an augmented dataset, we employ several pre-trained convolutional neural networks (CNNs) to aid in the detection of COVID-19. We employ the pre-trained CNNs SqueezeNet (Iandola et al., 2016), ResNet-18 (He, Zhang, Ren, & Sun, 2016), and AlexNet (Krizhevsky, Sutskever, & Hinton, 2012), which achieved state-of-the-art performance and are computationally less expensive compared to other popular CNNs. We employ deep transfer learning (DTL) to fine-tune the CNNs as this decreases computation time and the complexity of the training process (Zhuang et al., 2021). Also, DTL has achieved promising results in the detection of COVID-19 (Minaee et al., 2020; Hall et al., 2020; Chowdhury et al., 2020; Narin et al., 2021).

In order to aid in the detection of COVID-19 and to contribute to the field of synthesizing medical images (CXRs), this research aims to answer the following research questions; *i) Can a customized AC-GAN properly generate synthetic CXR images of the COVID-19 pneumonia, non-COVID-19 pneumonia, and normal classes? ii) Can class hierarchy be incorporated into the GAN's image generation process to improve the synthesis of CXRs, i.e., does HAC-GAN improve the quality and diversity of the generated images? iii) Can an augmented CXR dataset (by using AC-GAN or HAC-GAN) be used to aid in the detection of COVID-19 with DTL, and does it improve classification performance compared to training the pre-trained CNNs on the original dataset?*

To answer the research questions and to determine whether the proposed models generalize well to unseen data, we use the CXR dataset that comprises seven open-access datasets. Six datasets are combined in the open-access COVIDx V8A dataset, which contains COVID-19 pneumonia, non-COVID-19 pneumonia, and normal CXR images (L. Wang et al., 2020). We add a seventh dataset, part of the public Shenzhen CXR dataset containing only normal images (Jaeger et al., 2013; Candemir et al., 2013). The total CXR dataset consists of 8,336 normal, 5,575 non-COVID-19 pneumonia, and 2,358 COVID-19 CXR images. The images in the CXR dataset can be structured into a hierarchy of 2 levels. All images are first divided into two classes; normal or pneumonia. Next, we split the pneumonia class into COVID-19 pneumonia and non-COVID-19 pneumonia cases. We aim to incorporate this class hierarchy in HAC-GAN's image generation process to improve the

quality and diversity of the generated images such that the synthetic CXRs from the pneumonia classes are more similar to each other than to the CXRs belonging to the normal class.

Both AC-GAN and HAC-GAN generate realistic and diverse CXRs. We find that leveraging class hierarchy in HAC-GAN improves the quality and diversity of the generated images based on visual examination and two popular evaluation metrics; the Multiscale Structural Similarity (MS-SSIM) score (Pessoa, 2021) and the Frechet Inception Distance (FID) (Heusel, Ramsauer, Unterthiner, Nessler, & Hochreiter, 2017). HAC-GAN generates images that resemble the original dataset more closely (shown by lower FID scores), while it also produces a more diverse set of images (shown by lower MS-SSIM scores). Additional experiments on the Fruit (Mihai-Dimitrie, 2021) and Vegetable (Ahmed, Mamun, & Asif, 2021) benchmark datasets verify the effectiveness of HAC-GAN. Next, augmenting the original dataset with images from HAC-GAN improves the classification performance of the pre-trained CNNs, indicating that the synthetic CXRs contain meaningful features that enhance classification performance. However, the per-class classification results and the Gradient-weighted Class Activation Mapping (Grad-CAM) visualizations show that the CNNs suffer from learned differences caused by variations in image acquisition. Augmenting the dataset thus cannot guarantee that the CNNs generalize well to unseen data gathered by different hospitals. When trained and evaluated on a dataset augmented by HAC-GAN with a modified train/test split representing an analysis on local data (i.e., data from the same hospital), the CNNs can accurately detect the different classes. ResNet-18 performs best, and its accuracy increases from 95.9% to 98.5% after training on the augmented dataset. To leverage the potential of the proposed methods in practice, HAC-GAN and the CNNs need to be trained on local data.

The remainder of this research is organized as follows. Section 2 describes the used datasets. An overview of the relevant literature is provided in Section 3. Section 4 describes all used methods, after which Section 5 presents the experimental results. Finally, Section 6 summarizes our conclusions and suggests topics for future works.

2 Dataset

We elaborate on the dataset containing CXRs used for the detection of COVID-19 in Section 2.1. Next, Section 2.2 describes the benchmark datasets used to verify the effectiveness of HAC-GAN.

2.1 CXR Dataset

In this research we employ the "CXR dataset", which combines the COVIDx V8A and Shenzhen chest X-ray datasets. The open-access COVIDx V8A is introduced by L. Wang et al. (2020) and

is constructed to aid in the detection of COVID-19 pneumonia, non-COVID-19 pneumonia and normal classes in CXR images. This dataset consists of a total of 15,999 CXR images, and is one of the largest public CXR datasets that contains positive COVID-19 cases to the best of our knowledge. The dataset contains 8,066 normal, 5,575 non-COVID-19 pneumonia and 2,358 COVID-19 pneumonia CXR images. The resolution of the CXRs ranges from 156×157 to 4032×3024 pixels. L. Wang et al. (2020) combined the following six different public data repositories in COVIDx V8A;

1. The COVID-19 Image Data Collection (Cohen et al., 2020)
2. The Figure 1 COVID-19 Chest X-ray Dataset Initiative (Chung, 2020b)
3. The ActualMed COVID-19 Chest X-ray Dataset Initiative (Chung, 2020a)
4. The COVID-19 Radiography Database (v3) (Radiological Society of North America, 2019a)
5. The Radiological Society of North America (RSNA) Pneumonia Detection Challenge dataset (Radiological Society of North America, 2019b)
6. The RSNA International COVID-19 Open Radiology Database (RICORD) (Tsai et al., 2021).

Example images of each class from the COVIDx V8A dataset are shown in Figure 1. These examples illustrate the diversity and variety of patient cases in the dataset. Table 1 shows the (class) distribution of images over the different datasets. Dataset generation scripts for constructing the COVIDx dataset are available at `https://github.com/lindawangg/COVID-Net`.

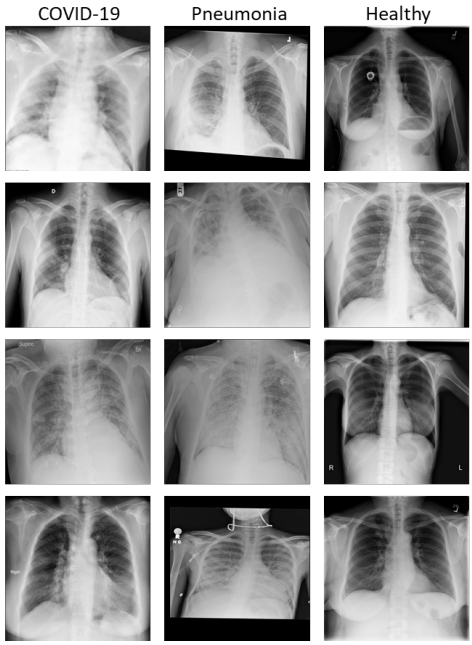


Figure 1: Example CXRs of COVID-19 pneumonia, non-COVID-19 pneumonia and normal patient cases in the COVIDx V8A dataset.

Table 1 *Class distribution of images over the six datasets*

Dataset (# and name)	Healthy (#)	Pneumonia (#)	COVID-19 (#)
1. COVID-19 Image Data collection	0	52	270
2. The Figure 1 COVID-19 Dataset	0	0	24
3. The ActualMed Dataset	0	0	25
4. COVID-19 Radiography Database	0	0	943
5. RSNA Pneumonia Detection Dataset	8,066	5,523	0
6. RICORD dataset	0	0	1,096
7. Shenzhen chest X-ray dataset	270	0	0
Total	8,336	5,575	2,358

L. Wang et al. (2020) proposed a certain train/test split in the COVIDx V8A dataset. Their train set contains 15,599 images from the following datasets listed above; 1, 2, 3, 4, 5 and 6 while the test set contains 400 images from the datasets 5 and 6. The test images are randomly selected from the scans gathered by the RSNA. The train and test sets thus contain images that originate from the same datasets. According to DeGrave, Janizek, and Lee (2021), such train/test splits can create optimal settings for artificial intelligence methods to achieve high performance by classifying based on "shortcuts". These shortcuts are learned patterns between the presence or absence of COVID-19 and features in the images that reflect variations in image acquisition (DeGrave et al., 2021). Their research states that models often do not learn the true pathological features that reflect the presence of COVID-19, but instead learn these shortcuts. For example, dataset 6 exclusively contains COVID-19-positive cases, and the systematic acquisition differences between datasets 6 and 5 correlate perfectly with the presence of COVID-19 as the source of the datasets corresponds with the COVID-19 disease status (DeGrave et al., 2021). The reported accuracy on the test set then does not reflect how well the model actually predicts the presence of COVID-19, but rather reflects how well the model recognizes from which dataset the images in the test set originate. This often causes the models to not generalize well to other newly introduced datasets and fail when tested in new hospitals (DeGrave et al., 2021).

We thus want to prevent CXR images that originate from the same datasets to be present in both the train and test sets and we therefore change the train/test split in the COVIDx V8A dataset. We select one of the data sources in it's entirety as test set, which represents a cross-dataset (or cross-hospital) evaluation. The COVID-19 Image Data Collection introduced by Cohen et al. (2020) contains both COVID-19 pneumonia and non-COVID-19 pneumonia cases, and it also contains roughly the same amount of images as the current test set, so we employ this dataset as test set. This dataset does not comprise normal CXRs, so we add a seventh dataset with unseen normal CXRs to the test set.

The public Shenzhen dataset contains 662 CXR images, of which 336 are cases of tuberculosis and 326 normal cases (Jaeger et al., 2013; Candemir et al., 2013). This dataset was collected by the National Library of Medicine, National Institutes of Health, Bethesda, MD, USA in collaboration with Shenzhen No.3 People's Hospital, Guangdong Medical College, Shenzhen, China and can be downloaded from; https://www.kaggle.com/datasets/kmader/pulmonary-chest-xray-abnormalities. We randomly select 270 of the normal images to be included in the test set. The train set now consists of the datasets 2, 3, 4, 5 and 6 from Table 1, while the test set consists of the datasets 1 and 7. There is no overlap between the datasets in the train and test set and we can now determine if the proposed models generalize well to unseen data collected in different hospitals. The train set consists of 2,088 COVID-19 images, 5,523 non-COVID-19 pneumonia images and 8,066 normal images. The test set contains 270 COVID-19 pneumonia images, 52 non-COVID-19 pneumonia images and 270 normal images.

The CXR dataset can be structured into the 2 level hierarchy illustrated in Figure 2. The images can first be divided into the normal and pneumonia classes. Next, we can split the pneumonia class into the non-COVID-19 pneumonia and COVID-19 pneumonia classes. The classes of interest are on the leaf nodes of the hierarchy/graph. We aim to incorporate this class hierarchy in the image generation process of the GAN to improve both the quality and diversity of the synthetic images.

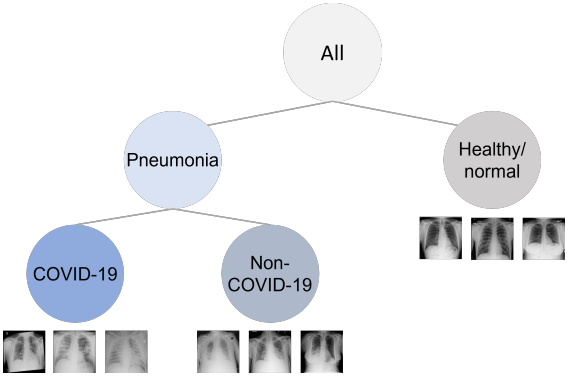
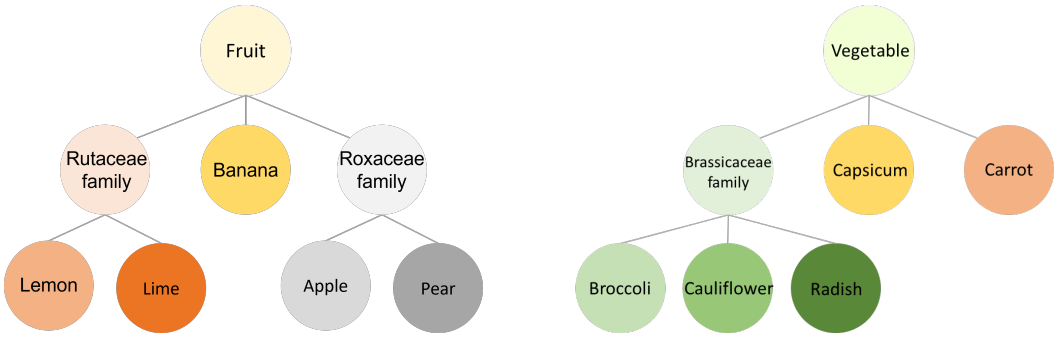


Figure 2: Hierarchy of the classes in the CXR dataset, with the classes of interest on the leaf nodes.

2.2 Benchmark Datasets

To verify if the implemented (H)AC-GANs are able to generate high-quality and diverse images, we use two additional benchmark datasets that have a clear hierarchical structure; the Fruit and Vegetable datasets that are adapted from Mihai-Dimitrie (2021) and Ahmed et al. (2021), respectively. These datasets contain more classes and different hierarchical structures than the CXR

dataset. The Fruit dataset consists of the following classes; apple, banana, lemon, lime and pear. Its hierarchy is given in Figure 3a. The Vegetable dataset consists of the following classes; broccoli, capsicum, carrot, cauliflower and radish and its hierarchy is given in Figure 3b. Example images of these datasets are given in Appendix A. The classes of interest in the Fruit and Vegetable datasets can be downloaded from https://www.kaggle.com/datasets/aelchimminut/fruits262 and https://www.kaggle.com/datasets/misrakahmed/vegetable-image-dataset, respectively. We collect 500 images from each class for both datasets.



(a) Fruit dataset **(b)** Vegetable dataset

Figure 3: Class hierarchies of the Fruit dataset (a) and Vegetable dataset (b).

3 Literature Review

This section conducts a survey on the relevant scientific work on applying deep machine learning on classifying and synthesizing (medical COVID-19) images. We discuss our research in relation to existing literature. This section consists of two main parts. First, Section 3.1 gives a short history of deep learning for (medical) image classification where Section 3.1.2 gives examples of deep (transfer) learning models that have been developed to detect COVID-19 in radiographic images. Section 3.2 reviews GANs and gives examples of models in the literature that generate COVID-19 medical images in Section 3.2.2. Lastly, Section 3.2.3 elaborates on including class hierarchy in the image generation process of a GAN via graph embedding methods.

3.1 Deep Learning for (Medical) Image Classification

Neural networks (NN), also known as artificial neural networks (ANN), are a subset of machine learning and many deep learning models utilize these networks as they can solve complex problems. The name and structure of NNs are inspired by the human brain, mimicking the way that the brain's neurons signal to each other. We elaborate on the rise of NNs and CNNs, and compare their architectures in Section 3.1.1. We review CNNs used in detecting COVID-19 in Section 3.1.2.

3.1.1 (Convolutional) Neural Networks

The development of neural networks dates back to the 1940s, when McCulloch and Pitts (1943) introduced the first simple NNs. Next, Werbos (1974) proposed to use backpropagation to optimize NNs and this method was adopted as a training technique with the research of Rumelhart, Hinton, and Williams (1985). With the adoption of backpropagation, researchers were able to train and use NNs in many research areas. However, as neural networks became deeper with more layers, backpropagation became slower and the advancement of neural networks halted. In the early 2010s, several studies recommended using Rectified Linear Unit (ReLU) activation functions in the neurons to transmit information and help train deeper and more complex neural networks efficiently (Nair & Hinton, 2010). Combined with more available data, the rise of better optimization algorithms (like Adaptive Moment Estimation (Adam)) and increasing computing power, the use and research of NNs revived and significant progress has been made in the field of neural networks.

Many different types and architectures of NNs exist but all comprise the same building blocks. Artificial neural networks consist of neurons/nodes that are organized in layers. Figure 4 shows an example of a three-layer NN with input $\boldsymbol{x} = \{x_1, ..., x_j\}$, two hidden layers with 4 and the n nodes respectively, and an output layer with two nodes. The neurons in the different layers are connected and each connection is associated with a certain weight. In Figure 4 the nodes are fully connected, meaning there is a connection between each node in consecutive layers. The weights between the neurons are the learnable parameters of the network. The output of a node is called an activation that is computed with linear combinations and non-linear transformations. To compute its output, each neuron/node takes previous connected activations as input, where for the first hidden layer the previous activations are the actual inputs of the network $\{x_1, ..., x_j\}$. Next, each activation is multiplied with its associated weight, and all combinations (the weights multiplied with activations) are summed up, a bias term is added and the result is passed through an activation function. The linear combinations and non-linear transformations via the activation functions allow for very flexible networks that can approach any function. Computing the output of a network by multiplying activations with weights, adding bias terms and transforming with activation functions is called the forward pass of the network. During training of the NN, the goal is to tune all weights and biases in the network

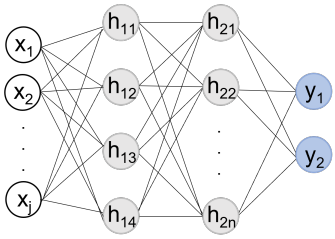


Figure 4: A fully connected neural network with two hidden layers and one output layer. Image by the author.

such that the loss function is minimized. The loss function is lower when the output of the network closely matches the target output. First, the weights and biases are initialized randomly and then the network starts slightly adjusting the learnable parameters by using backpropagation during the backward passes. The loss is calculated and the gradient of the loss function with respect to the model's parameters is computed by repeated application of the chain rule. The weights are updated by moving into the direction of steepest descent, i.e., the negative of the computed gradient, with a step size that is equal to the learning rate. Many different types of NNs exist, differing in the used activation functions, the number of layers and neurons, the learning rates or the optimizers.

LeCun et al. (1989) introduced a new type of NN called Convolutional Neural Networks (CNN) that performs particularly well on recognizing patterns in images. Just like traditional NNs, the CNN is comprised of layers with neurons who's associated weights are optimized through backpropagation. A CNN is characterized by convolutional layers, as opposed to fully connected layers in ANNs. A convolutional layer applies a convolution on its input, which allows for the identification of deep patterns with relatively few parameters. One of the key differences with traditional NNs is that the neurons within the CNN are organised into three dimensions; the height and width of the image, and the depth (the number of channels). Because of this structure, CNNs are able to retain the structural information contained in neighboring pixels of images and therefore have attracted great interest in the field of medical image analysis. Figure 5 gives a schematic illustration of an example CNN with input of size $6 \times 6 \times 2$ (with depth 2), two convolutional layers, and a fully connected layer that determines the final output layer with two nodes. CNNs often also contain pooling layers, but we refrain from showing these layers for simplicity. Compared to the NN from Figure 4, the neurons in the CNN in Figure 5 are structured along the three dimensions, which allows for retaining the spatial information in the image. Also, instead of being fully connected, the neurons in consecutive layers are now only locally connected via the kernel (which contains learnable weights) and "local receptive field", which is the overlapping area of the kernel and the input feature map at a specific kernel position. For example, the value of the orange node in the first convolutional layer is computed by only multiplying, summing and transforming the activations from the input image in the orange

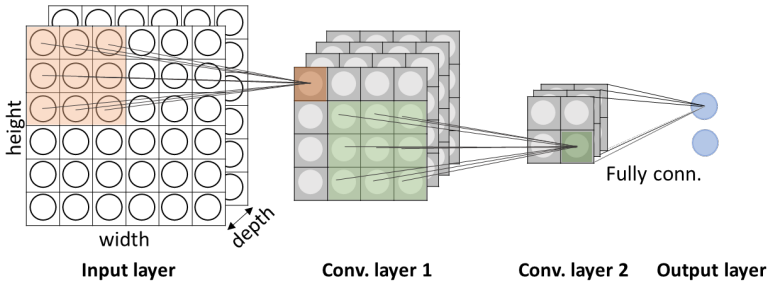


Figure 5: A CNN with two convolutional layers and one fully connected layer to create the output. Image by the author.

local receptive field with their corresponding kernel weights, which are also shared across the rest of the layer. Unlike a deep neural network, a CNN exploits three mechanisms that reduce the number of learnable parameters and computational costs; a local receptive field, weight sharing via the kernels and subsampling via the stride. We further elaborate on these concepts in Section 4.1.

Since their introduction, CNNs have been used in medical image applications such as breast tissue classification (Sahiner et al., 1996) and the detection of lung nodules (Lo et al., 1995) but deep CNNs could not be trained properly due to the lack of computational power. The use of deep learning methods in medical imaging has increased exponentially since the resurgence of deep learning in computer vision due to an increase in computing power and more available annotated data (Krizhevsky et al., 2012). For example, CNNs were employed to detect diabetic retinopathy (Serener & Serte, 2020), skin cancer (Esteva et al., 2017), and common thorax diseases (X. Wang et al., 2017). However, a large problem with medical image classification remains the lack of available labelled training data which can lead to overfitting of the CNNs. Deep learning methods need many data samples to accurately learn the data generating process as a large number of parameters need to be tuned. To increase the performance of image classification on small datasets several options exists, one of which is deep transfer learning (DTL). With DTL, a deep learning model pre-trained on another dataset is used to solve a different problem without needing to re-train the whole model from scratch. The parameters of the model start with good initial parameter values that only need to be tuned slightly towards a new task. Transfer learning has been very effective in medical imaging problems (Ravishankar et al., 2016). Another method is data augmentation, where (synthetic) samples are added to the dataset to increase its size. We elaborate on both methods in the next section, where we discuss the use of CNNs in detecting COVID-19 in radiographic images.

3.1.2 Deep Learning for Medical COVID-19 Images

Motivated by the need to develop solutions to aid in the fight against COVID-19, multiple deep learning methods that can detect the presence of COVID-19 have been proposed recently. For example, Khobahi et al. (2020) developed a new deep-learning based image processing and classification method for the detection of COVID-19 in CXR images. They considered a semi-supervised method based on autoencoders and trained an efficient classifier. They showed state-of-the-art performance based on accuracy (94%), sensitivity (>90%), and predicted positive value (>90%).

L. Wang et al. (2020) also proposed a novel deep learning architecture for the detection of

COVID-19. They introduced the open-source COVID-Net, a customized convolutional neural network specifically designed for the detection of COVID-19 in CXR images. They reported an accuracy of 93.3%. L. Wang et al. (2020) made their dataset COVIDx publicly available, and this dataset is a previous version of the dataset described in Section 2, where we also discuss how the train and test sets of COVIDx contain images that originate from the same datasets and that it thus remains unclear if COVID-Net generalizes well to unseen data, i.e., CXRs from other hospitals. To investigate this, Luz et al. (2021) tested COVID-Net with a cross-dataset evaluation to ascertain the power of generalization of the model regarding variations in image acquisition. Their results showed that the model only achieved a 51.3% accuracy when tested on a new (unseen) dataset, which shows that the model suffers from variations in image acquisition and does not generalize well to CXRs from other hospitals. This result verifies the claims made in the research of DeGrave et al. (2021) and supports our decision to change the train and test set split in our research.

Besides, Minaee et al. (2020) employed several pre-trained deep learning models to detect the presence of COVID-19 from CXRs. Transfer learning was used on a dataset of 2,000 CXR images to train four widely used CNNs, including ResNet-18, ResNet-50, SqueezeNet, and DenseNet-121. They evaluated the trained models on a test set of 3,000 CXR images and achieved a sensitivity rate around 98% and a specificity rate of approximately 90% for all models.

Multiple researchers beside Minaee et al. (2020) used pre-trained DTL models for the detection of COVID-19 in CXR images. Narin et al. (2021) employed five pre-trained CNNs; ResNet-50, ResNet-101, ResNet-152, InceptionV3 and Inception-ResNetV2, with ResNet-50 achieving the highest classification accuracy. Hall et al. (2020) trained the pre-trained ResNet-50 and VGG-16 models on a balanced set of non-COVID-19 pneumonia and COVID-19 CXRs. They achieved a maximum accuracy of 94.4%. Chowdhury et al. (2020) trained eight different pre-trained models. They employed three relatively shallow models; MobileNetV2, SqueezeNet and ResNet-18. They also trained five deeper networks; InceptionV3, ResNet-101, CheXNet, VGG-19 and DenseNet201. All networks achieved an accuracy of >95% for the three-class classification problem. Rajaraman et al. (2020) proposed an iteratively pruned deep learning model for the detection of COVID-19 in CXRs and compared its performance to 8 different deep transfer models. All employed models achieved an accuracy of over 95%. These works illustrate that DTL models can accurately detect COVID-19 in radiographic images. We use transfer learning as it decreases computation time and the complexity of the training process, and previous research has shown to achieve promising results

with DTL. We employ the pre-trained models SqueezeNet (Iandola et al., 2016), ResNet-18 (He et al., 2016), and AlexNet (Krizhevsky et al., 2012) as they achieved state-of-the-art results in several tasks during recent years and are computationally less expensive compared to other popular CNNs like Xception (Chollet, 2017), Inception-ResNet (Szegedy, Ioffe, Vanhoucke, & Alemi, 2017) and DenseNet (G. Huang, Liu, Van Der Maaten, & Weinberger, 2017).

Even though Minaee et al. (2020) achieved promising results with DTL, they state that further experiments are needed due to the limited number of available COVID-19 CXR images. Also, Hall et al. (2020) and Rajaraman et al. (2020) assert that one of the limitations in their work is the small datasets. One of the main concerns in this research area of classifying medical images is thus the lack of available datasets of sufficient size. This poses a major challenge for detecting COVID-19 in CXRs using deep learning, as only limited datasets are available. Although the above mentioned models achieve high accuracies, the results cannot be trusted to generalize well to unseen data as overfitting often occurs when deep learning models are trained on limited datasets. This is also illustrated by the cross-dataset evaluation of COVID-Net by Luz et al. (2021) mentioned previously.

To alleviate this problem regarding small datasets, data augmentation methods can be used. A widely employed data augmentation technique uses image transformations and adjustments of image color to generate slightly different versions of existing images. These adjustments include but are not limited to scaling, rotating, cropping and adjusting brightness or contrast. This type of data augmentation is fast and reliable, but its changes are limited because an existing image is slightly modified and it does not create unseen data. A second way of augmenting a dataset is by adding synthetic data to the dataset. For example, deep learning architectures can be used to approximate the true data distribution and generate images that have not been seen before. One deep learning method that is able to generate synthetic data, is a generative adversarial network (GAN). Section 3.2 reviews GANs and their application to medical (COVID-19) images.

3.2 Generative Adversarial Networks

GANs, introduced by Goodfellow et al. (2014), have recently received a lot of attention in the machine learning field for their ability to learn complex, deep representations of data, without needing many labeled data samples. GANs are based on the idea of competition, in which two interacting neural networks, a generator G and a discriminator D, are trying to outsmart each other. The generator and discriminator are trained jointly where the objective of the generator is

to synthesize fake datasamples that resemble real data, while the objective of the discriminator is to distinguish between real and generated instances. If the discriminator is able to easily identify the instances coming from the generator then, relative to its discrimination ability, the generator is producing low quality datasamples. The discriminator, while also improving, provides feedback about the quality of the generated samples to the generator and forces the generator to increase its performance. When training is successful, the generator has learned to approximate the underlying data generating process of the real datasamples.

Next to its ability to learn complex data generating processes, this class of models has gained popularity because of its applicability to a wide range of fields. For example, GANs have been successful in music generation (Guimaraes, Sanchez-Lengeling, Outeiral, Farias, & Aspuru-Guzik, 2017), image blending (Wu, Zheng, Zhang, & Huang, 2019), speech synthesis (Kong, Kim, & Bae, 2020), text-to-image translation (Reed et al., 2016), generating high-resolution images from low-resolution images (X. Wang et al., 2018), and facial attribute manipulation (Zhao, Chang, Jie, & Sigal, 2018). Another popular field is image synthesis, which remains a main focus area of GANs as it has enabled the generation of very realistic synthetic images, a task that was impossible before.

However, GANs are not without problems as they are hard to train. Mode collapse is a common issue, where the generator fails to learn the full data generating distribution and only generates samples with little variety. Another problem with GANs is the vanishing gradient problem. During backpropagation, gradients flow from the final layer to the first layer and get increasingly smaller with each layer due to repetitive application of the chain rule. The gradient sometimes becomes so small that the weights of the first layers are not updated and the initial layers stop learning completely. Training a stable GAN that generates high-quality and diverse images is thus very challenging. As a result, many GAN variants have been introduced that aim to solve these problems. GAN modifications to improve performance either focus on the architecture or the loss perspective and many different options exist for building a GAN's architecture and choosing its objective function. We give a short overview of the relevant proposed GAN architectures in the literature in Section 3.2.1. We refer the reader to Z. Wang, She, and Ward (2021) for an overview of different loss functions for more stable GAN training. Next, we present GANs applied to COVID-19 medical images in Section 3.2.2 and review including class hierarchy into a GAN in Section 3.2.3.

3.2.1 GAN Architectures

A GAN thus comprises two neural networks, the discriminator D that distinguishes between real and generated images and the generator G that aims to fool the discriminator with synthetic images. The optimization and training of a GAN is done with respect to a joint loss function for D and G. The chosen architecture of the generator and the discriminator are important as those highly influence the training stability and performance of the GAN.

The first GAN introduced by Goodfellow et al. (2014) uses fully connected layers in both the generator and discriminator. The generator takes a noise vector $\boldsymbol{z}$ as input and outputs a fake sample $G(\boldsymbol{z})$. Next, the discriminator D takes a fake sample $G(\boldsymbol{z})$ or real sample $\boldsymbol{x}$ from the dataset as input and outputs the probability that the sample is real. Figure 6a illustrates the architecture of a traditional GAN. The generator G and discriminator D both have their own loss functions. The discriminator can be trained by minimizing the following loss function

$$L^{(D)} = -\mathbb{E}_{x\sim p_r} \log D(\boldsymbol{x}) - \mathbb{E}_{z\sim p_z} \log(1 - D(G(\boldsymbol{z}))), \tag{1}$$

where $\boldsymbol{x}$ is the real image sampled from real data distribution p_r, $G(\boldsymbol{z})$ the generated sample with $\boldsymbol{z}$ the latent vector sampled from latent distribution p_z, $\mathbb{E}$ represents the expectation, $D(\boldsymbol{x})$ represents the discriminator's prediction of the probability that $\boldsymbol{x}$ is real and $D(G(\boldsymbol{z}))$ represents the discriminator's prediction of the probability that $G(\boldsymbol{z})$ is real. As D wants to correctly identify real and synthetic data, it wants $D(\boldsymbol{x}) \to 1$ and $D(G(\boldsymbol{z})) \to 0$ such that $(1 - D(G(\boldsymbol{z}))) \to 1$. This is achieved by minimizing $L^{(D)}$ in equation (1). The goal of the generator G is to bring $D(G(\boldsymbol{z})) \to 1$, as it wants to fool the discriminator into believing that the generated data is real, i.e., the generator wants the synthetic data to be similar to the real data. The generator learns a distribution p_g as the distribution of the samples $G(\boldsymbol{z})$ such that p_g approximates p_r. The generator and discriminator thus have conflicting goals and the loss of the generator is the negative of the discriminator loss function from equation (1) and is given by

$$L^{(G)} = -L^{(D)} = \mathbb{E}_{x\sim p_r} \log D(\boldsymbol{x}) + \mathbb{E}_{z\sim p_z} \log(1 - D(G(\boldsymbol{z}))) = V(D, G) \tag{2}$$

where $L^{(G)}$ can be written as value function $V(D, G)$. From the perspective of the generator, $V(D, G)$ in equation (2) should be minimized, while the discriminator wants to maximize this value function. Therefore, the traditional GAN optimization problem can be written as a the

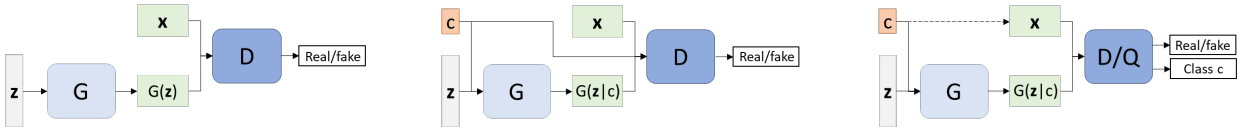
following zero-sum minmax game

$$\min_G \max_D V(D,G) = \min_G \max_D \mathbb{E}_{x\sim p_r} \log D(\boldsymbol{x}) + \mathbb{E}_{z\sim p_z} \log(1 - D(G(\boldsymbol{z}))). \tag{3}$$

Even though theoretical optimal solutions exist for these traditional GANs, the generator often fails to converge as the discriminator is confident in its prediction and will not update its parameters. With the current loss functions, the generator's gradient vanishes when the discriminator successfully rejects generator samples with high confidence (Goodfellow, 2016). The minmax game from equation (3) is useful for theoretical analysis, but thus does not perform well in practice (Goodfellow, 2016). The loss function of the generator can be rewritten to solve this and becomes

$$L^{(G)} = -\mathbb{E}_{z\sim p_z} \log(D(G(\boldsymbol{z}))), \tag{4}$$

where the generator now minimizes this loss function to maximize the chance that the discriminator believes that fake, generated data is real. Note that by rewriting the loss function, the objective function is no longer a zero-sum game and is now a heuristic as the optimization problem of the GAN can no longer be written as the single value function from equation (3) (Goodfellow, 2016).



(a) GAN **(b)** CGAN **(c)** AC-GAN

Figure 6: The architectures of (a) GAN, (b) CGAN, and (c) AC-GAN, where $\boldsymbol{z}$ is the latent vector, G is the generator, $G(\boldsymbol{z})$ is a generated image, $G(\boldsymbol{z}|c)$ is a generated image conditional on class label c, $\boldsymbol{x}$ is a real image, D is the discriminator and Q is the auxiliary classifier. Images are generated by the author.

Even with the modified loss function for the generator G, training if often still unstable. Several new GAN architectures were proposed to further stabilize the training process. The traditional GAN introduced by Goodfellow et al. (2014) employs fully connected layers in the discriminator and generator. Later, Radford, Metz, and Chintala (2015) introduced the Deep Convolutional Generative Adversarial Network (DCGAN). Compared to a traditional GAN, DCGAN removes the fully connected layers, replaces pooling layers with (transposed) convolutional layers used in CNNs, uses batchnorm in both the generator and discriminator, uses ReLU activations in the generator in all layers except for the last and uses LeakyReLU activations in the discriminator (Radford et al., 2015). DCGANs were the first GAN architecture to successfully learn to generate

high resolution images and after its introduction, this architecture has become best practice and a standard in GAN modelling (Goodfellow, 2016). We therefore base our GAN implementation on the DCGAN architecture and we elaborate on all used modules in the Methodology in Section 4.1.

In both the traditional GAN and DCGAN, the input of the generator G is the latent vector $\boldsymbol{z}$. This input is unrestricted and and there is no control over which kind of images (of a specific class for example) will be generated. This can lead to the collapse of the training process (Pan et al., 2019). To further improve the quality of the generated images, Goodfellow (2016) and Oord et al. (2016) state that incorporating outside information (such as class labels) in the GAN almost always results in an improvement of the generated images. It has not been theoretically proven why incorporating class labels improves GANs, but the incorporation of class information provides the training process with useful clues that help with optimization (Goodfellow, 2016). Mirza and Osindero (2014) introduced the Conditional Generative Adversarial Network (CGAN), which leverages additional information in the image generation process. In CGAN, both the generator and discriminator are conditioned on some auxiliary information c, e.g., class labels. The conditional variable c is fed to both the discriminator and generator as extra input (Mirza & Osindero, 2014). Figure 6b shows the architecture of CGAN, which illustrates that the generator now takes c and the latent vector $\boldsymbol{z}$ as input, while the discriminator is provided with the class alongside the generated sample $G(\boldsymbol{z}|c)$ (conditional on c) or real sample $\boldsymbol{x}$. The modified loss functions of the discriminator D and generator G can be written as

$$L^{(D)} = -\mathbb{E}_{x\sim p_r} \log D(\boldsymbol{x}|c) - \mathbb{E}_{z\sim p_z} \log(1 - D(G(\boldsymbol{z}|c))), \tag{5}$$

$$L^{(G)} = -\mathbb{E}_{z\sim p_z} \log(D(G(\boldsymbol{z}|c))). \tag{6}$$

which differ slightly from equations (1) and (4) as the real and generated samples are now conditional on the outside information c. $D(\boldsymbol{x}|c)$ represents the discriminator's prediction of the real data sample $\boldsymbol{x}$ being real for a given class c and $D(G(\boldsymbol{z}|c))$ represents the discriminator's prediction of the generated sample being real for a given class c.

Odena et al. (2017) next proposed the Auxiliary Classifier GAN (AC-GAN), which is a type of conditional GAN where the discriminator is slightly altered. Just like in CGAN, the generator G receives the noise vector $\boldsymbol{z}$ and class label c as input to synthesize class conditional samples $G(\boldsymbol{z}|c)$. However, instead of feeding outside information c as input to the discriminator, the discriminator

in AC-GAN is now tasked with reconstructing this additional information (Odena et al., 2017). The modified discriminator D still outputs the probability that the image is real, but it now also contains an auxiliary classifier Q that predicts the class of an image (see Figure 6c). This further stabilizes the training process and we therefore employ this architecture in our research. The loss functions of the discriminator and generator are slightly altered as the discriminator now has an additional classification task, i.e., predicting the class of the input image. The objective functions now consist of two parts; the original loss function for discriminating real and fake images, together with the auxiliary classifier loss function (Odena et al., 2017). The modified loss functions that D and G aim to minimize are given by

$$\begin{aligned} L^{(D)} = &-\mathbb{E}_{x\sim p_r} \log D(\boldsymbol{x}) - \mathbb{E}_{z\sim p_z} \log(1 - D(G(\boldsymbol{z}|c))) \\ &- \mathbb{E}_{x\sim p_r} \log Q(\boldsymbol{x}) - \mathbb{E}_{z\sim p_z} \log Q(G(\boldsymbol{z}|c)), \end{aligned} \tag{7}$$

$$L^{(G)} = -\mathbb{E}_{z\sim p_z} \log D(G(\boldsymbol{z}|c)) - \mathbb{E}_{z\sim p_z} \log Q(G(\boldsymbol{z}|c)), \tag{8}$$

where $Q(\boldsymbol{x})$ and $Q(G(\boldsymbol{z}|c))$ represent the predicted probability distributions over the class labels of the real sample $\boldsymbol{x}$ and the generated sample $G(\boldsymbol{z}|c)$, respectively. So next to the original task of identifying fake or real images, the discriminator now has the additional task of predicting the class of real and generated images, which is represented by the additional classifier loss function $-\mathbb{E}_{x\sim p_r} \log Q(\boldsymbol{x}) - \mathbb{E}_{z\sim p_z} \log Q(G(\boldsymbol{z}|c))$ in equation (7). The loss function of the generator is also slightly altered, as next to trying to mislead the discriminator, it now also asks the discriminator to correctly classify its generated images by adding $-\mathbb{E}_{z\sim p_z} \log Q(G(\boldsymbol{z}|c))$ in equation (8).

Besides CGAN and AC-GAN, many GAN architectures in the literature have been proposed. For example, Stacked Generative Adversarial Networks (StackGAN) stack multiple generator-discriminator pairs to synthesize high-quality images from text descriptions (H. Zhang et al., 2017). Another GAN that its able to produce high-quality images is a Progressive Growing GAN, which grows the generator and discriminator progressively by adding new layers that increasingly go from a low to a high resolution as training progresses (Karras, Aila, Laine, & Lehtinen, 2017). Even though those models generate higher resolution images, AC-GAN has a smaller computational budget and is less time-consuming and we therefore employ AC-GAN in this research. We refer the reader to Alqahtani, Kavakli-Thorne, and Kumar (2021), Hong, Hwang, Yoo, and Yoon (2019) and Creswell et al. (2018) for an overview of different GAN architectures proposed in the literature. The following section elaborates on GANs applied to detect COVID-19 in radiographic images.

3.2.2 GANs applied to Medical COVID-19 Images

One of the main challenges of detecting COVID-19 in medical images with deep learning methods is the lack of available COVID-19 datasets that are of sufficient size. As stated in the previous section, GANs can be employed to generate more samples to alleviate the issue of small datasets. Recently, several researchers utilized GANs to generate synthetic COVID-19 medical images, either CXR or CT images (Loey et al., 2020; Waheed et al., 2020; Al-Shargabi et al., 2021). These studies employed different kind of GAN architectures combined with several classification methods to improve the detection of COVID-19.

Loey et al. (2020) proposed to use a traditional GAN to generate synthetic examples of four different classes; normal, pneumonia bacterial, pneumonia virus and COVID-19. They selected three deep transfer learning models AlexNet, GoogleNet and ResNet-18 for classification. Without using an augmented dataset, they reported a classification accuracy of 52.0%, 52.8% and 50.0% for AlexNet, GoogleNet, and ResNet-18, respectively. After training on an augmented dataset with the generated CXR images, the accuracies increased to 66.7%, 80.6% and 66.7%, respectively.

Waheed et al. (2020) developed CovidGAN, an AC-GAN, to synthesize normal and COVID-19 CXR images. They use CovidGAN together with a VGG-16 network fine-tuned using DTL to detect COVID-19 (Waheed et al., 2020). They used VGG-16 for its simplicity and depth. Their method resulted in an increase in classification performance from 85% to 95% accuracy when training the pre-trained CNN on the original dataset versus the augmented dataset, respectively.

Recently, Al-Shargabi et al. (2021) customized a conditional GAN (CGAN) to synthesize CXR images for the normal, COVID-19, and pneumonia classes. They used five pre-trained CNNs for classification purposes; InceptionResNetV2, Xception, SqueezeNet, VGG-16, and AlexNet. All deep transfer learning models achieved high classification accuracy (>98,9%) when trained on the augmented dataset, with InceptionResNetV2 achieving the highest accuracy of 99.7%. Al-Shargabi et al. (2021) do not train the DTL models on the original dataset (without augmentation), so it remains unclear how much the accuracy has increased after augmenting the dataset. Besides, their test set does not only consist of real CXR images, but it also contains images that are generated by the CGAN. Their test set consists of 630 fake images and only 100 real images. As they used images generated by the same data generating process (p_g by the CGAN) to both train and test the deep transfer learning models, it remains unclear if the models generalize well to real (unseen) data

and the resulting accuracy scores do not tell us much about their actual classification performance.

Based on the properties of the GAN architectures explained in Section 3.2.1, we propose a customized AC-GAN as it comprises a relatively simple architecture, while it is still able to produce high resolution images. We choose AC-GAN over CGAN and a traditional GAN as using an auxiliary classifier stabilizes the training process and further improves performance (Odena et al., 2017). Also, AC-GAN has a smaller computational budget and is less time consuming compared to more complicated architectures that can produce higher resolution images. Even though Waheed et al. (2020) also employ an AC-GAN, there are several differences with our proposed customized model. First, we generate and classify CXR images for three different classes, where Waheed et al. (2020) work with two classes; negative and positive COVID-19. Second, the architecture of our customized AC-GAN is different from the architecture used in Waheed et al. (2020). For example, we base our implementation of AC-GAN on the DCGAN architecture, we add Gaussian layers to the discriminator, use dropout in both D and G, we use multiplication of the noise and class vectors instead of concatenation and we employ a different number of (transposed) convolutional layers with different kernel sizes and strides in the architecture. We discuss all these elements and the reason we add them to our customized implementation of AC-GAN in Section 4.1. Besides, the train and test datasets from Loey et al. (2020), Waheed et al. (2020) and Al-Shargabi et al. (2021) all contain images that originate from the same datasets/hospitals. We elaborated on this problem in Section 2, where we state that by employing such a train/test split it remains unclear whether the proposed models generalize well to unseen data and we cannot deduce if the models perform properly from the resulting accuracy scores. Therefore, we use a different train/test split such that the train and test sets do not contain images that originate from the same dataset/hospital.

Next, Section 3.2.3 reviews the literature regarding including class hierarchy into GANs, and briefly explains how we incorporate class hierarchy into our proposed model.

3.2.3 Class Hierarchies in GANs

The GANs mentioned in the previous sections model images that belong to different classes independently. However, the classes in datasets can often be organized hierarchically (like a graph), where the hierarchy is based on the relations between the classes. Just as incorporating conditional information in the form of a class label c in CGAN and AC-GAN improved the quality of generated images (Goodfellow, 2016), incorporating relations between classes could further improve a

GAN's performance. Graphs contain valuable information and could provide useful clues that help the GAN generate high-quality images. For example, the generated images of two classes with a common parent node should be more similar to each other than to the generated images belonging to classes with a different parent node. By including relations between classes in the form of a graph in the GAN, we provide the GAN with richer, more meaningful outside information that could improve the quality and diversity of the generated images and further stabilize the training process. We therefore introduce HAC-GAN, which incorporates relations between classes into the image generation process of our customized AC-GAN.

In CGAN and ACGAN, an embedding layer is used to include the class label c into the image generation process. The embedding layer is a simple trainable lookup table that stores embeddings of a fixed dimension for each class, which we elaborate on in Section 4.1. The embeddings stored in this lookup table are randomly initialized, independent of each other, and thus do not contain any information about the relations between classes. Directly using a raw graph as input to the embedding layer (instead of class label c) in the AC-GAN to utilize class hierarchies is difficult as it takes numerical representations as input. We therefore use a graph embedding method in HAC-GAN to generate hierarchy-aware embeddings of the classes in the input graph. HAC-GAN thus contains a custom embedding layer that learns embeddings which capture the hierarchical relations between classes. We use the Node2Vec algorithm introduced by Grover and Leskovec (2016) to learn hierarchy-aware node embeddings. For each class of interest, its learned embedding is fed to the generator of the GAN, which then synthesizes hierarchy-aware images for each class.

Node2Vec is one of the most widely used methods that maps nodes in a graph to an embedding space as it is an open-source network, is scalable, and parallelizes easily. Node2Vec preserves the original graph structure when generating node embeddings such that (dis)similar nodes will have (dis)similar embeddings (Grover & Leskovec, 2016). It is based on the Skip-gram method that optimizes a likelihood objective using SGD, backpropagation, and negative sampling (Mikolov, Sutskever, Chen, Corrado, & Dean, 2013). Node2Vec introduced a flexible notion of a node's network neighborhood and is based on a biased random walk procedure that efficiently explores diverse graph neighborhoods. Grover and Leskovec (2016) state that this flexibility is key to learning richer representations of nodes in graphs. Other examples of popular methods for learning representations of nodes in graphs are DeepWalk (Perozzi, Al-Rfou, & Skiena, 2014), LINE (J. Tang et al., 2015), and spectral clustering (L. Tang & Liu, 2011). These methods all learn node embeddings in a

low-dimensional space while aiming to preserve similarities between similar nodes. Node2Vec was shown to significantly outperform these methods for learning representations of nodes in graphs on various tasks (Grover & Leskovec, 2016). Therefore, we employ Node2Vec as a graph node embedding method in this research. Section 4.2.2 further elaborates on this method.

A work that shows the effectiveness of levering class hierarchies into a GAN's image generation process is the recently proposed TreeGAN (R. Zhang, Mou, & Xie, 2020). TreeGAN encodes the class hierarchy, feeds it as a prior into the GAN to generate images, then measures the consistency of the generated images with the hierarchy and uses the calculated consistency score to guide the training of the generator. R. Zhang et al. (2020) use an offline-trained hierarchical classifier to measure classification errors, where small errors indicate that the generated images are consistent with the class hierarchy. The objective function of the GAN is modified, and the generator is trained to also minimize this hierarchical classification error. Even though R. Zhang et al. (2020) include class hierarchy in their model in a different way than this research, it does demonstrate its effectiveness, and gives support to our contribution of levering class relations in HAC-GAN.

4 Methodology

This section explains all used methods in detail. A basic understanding of NNs, stochastic gradient descent and backpropagation is assumed. First, Section 4.1 describes our custom implementation of the AC-GAN. Second, Section 4.2 elaborates on how we employ the Node2Vec algorithm to generate hierarchy-aware class embeddings, which are then used in the image generation process of HAC-GAN. Note that the only difference between the customized AC-GAN and introduced HAC-GAN is the incorporation of class hierarchies via a custom embedding layer in HAC-GAN. Section 4.3 describes the training procedure of (H)AC-GAN. The different DTL methods used for classification are next explained in Section 4.4. Lastly, the performance measures of the GAN and of the classification methods are explained in Section 4.5 and 4.6, respectively. All figures used in this section are made by the author, unless stated otherwise.

All code is written in Python 3.8.2 and the software is implemented with the open-source machine learning library `PyTorch` (Paszke et al., 2019). All models are trained and evaluated on the Lisa cluster computer. We employ the GeForce 1080Ti NVIDIA GPU and use MobaXterm as interface to access the Lisa cluster computer. All created code (including short explanations) can be found at `https://github.com/rsmalbraak/Thesis`.

4.1 Auxiliary Classifier GAN (AC-GAN)

GANs consist of two NNs - a generator G and a discriminator D that compete with each other to create real-looking synthetic images. We implement an AC-GAN in this research as conditioning on outside information and transforming the discriminator to predict the class instead of receiving it as an input stabilizes training and improves the quality of generated images (Odena et al., 2017). We base our customized AC-GAN on the DCGAN implementation as it is considered best practice. The DCGAN has become a standardized approach to building GAN's as it is able to produce stable output. This GAN does not condition on outside class information, so we modify the original implementation of DCGAN by Radford et al. (2015) such that it operates as an AC-GAN and is able to incorporate class conditional information into the image generation process. Figure 7 shows the architecture of our implementation of AC-GAN. We first give a high level overview of the AC-GAN and later elaborate on the layers used in D and G. We define the size of a tensor as $(H \times W \times C)$, where H and W are the height and width of the tensor in pixels respectively, and C denotes the number of channels. An image has size $128 \times 128 \times 3$, where the three channels are the blue, green and red channels. The original implementation of DCGAN synthesizes $64 \times 64 \times 3$ images. However, Odena et al. (2017) found that synthesizing higher resolution images leads to increased discriminability, so we modify the architecture to synthesize images of size $128 \times 128 \times 3$.

In Figure 7, the generator G takes a noise vector $\boldsymbol{z}$ and a class label c as input. The label is first passed through an embedding layer of dimension 100 and is then multiplied with the noise vector $\boldsymbol{z}$. The output of this multiplication is passed through a series of "transposed convolution" blocks $B_1, ..., B_6$ that first upsample it to a $4 \times 4 \times 1024$ tensor, then to $8 \times 8 \times 512$ until a fake $128 \times 128 \times 3$ image $\boldsymbol{x}_{\text{fake}} = G(\boldsymbol{z}|c)$ is generated. The discriminator in Figure 7 receives either a real image $\boldsymbol{x}_{\text{real}}$ or fake image $\boldsymbol{x}_{\text{fake}}$ as input and outputs its predicted class label and predicted source, i.e., whether an image is real or fake. The input image is first downsampled from $128 \times 128 \times 3$ to $64 \times 64 \times 64$, and finally to $4 \times 4 \times 1024$ by the blocks $B_1, ..., B_5$ in D. This final tensor is passed through the output blocks B_6 and B_7 that return the predicted source and predicted class label, respectively. Block B_7 contains the auxiliary classifier Q that outputs a probability distribution over the classes. We elaborate on each layer in the blocks B in the discriminator D and generator G in Sections 4.1.1 and 4.1.2, respectively. A detailed overview of D and G is given in Figures 8 and 9, respectively. The training procedure to learn the GAN's parameters/weights is explained in Section 4.3. We implement our customized AC-GAN in the `PyTorch` library (Paszke et al., 2019).

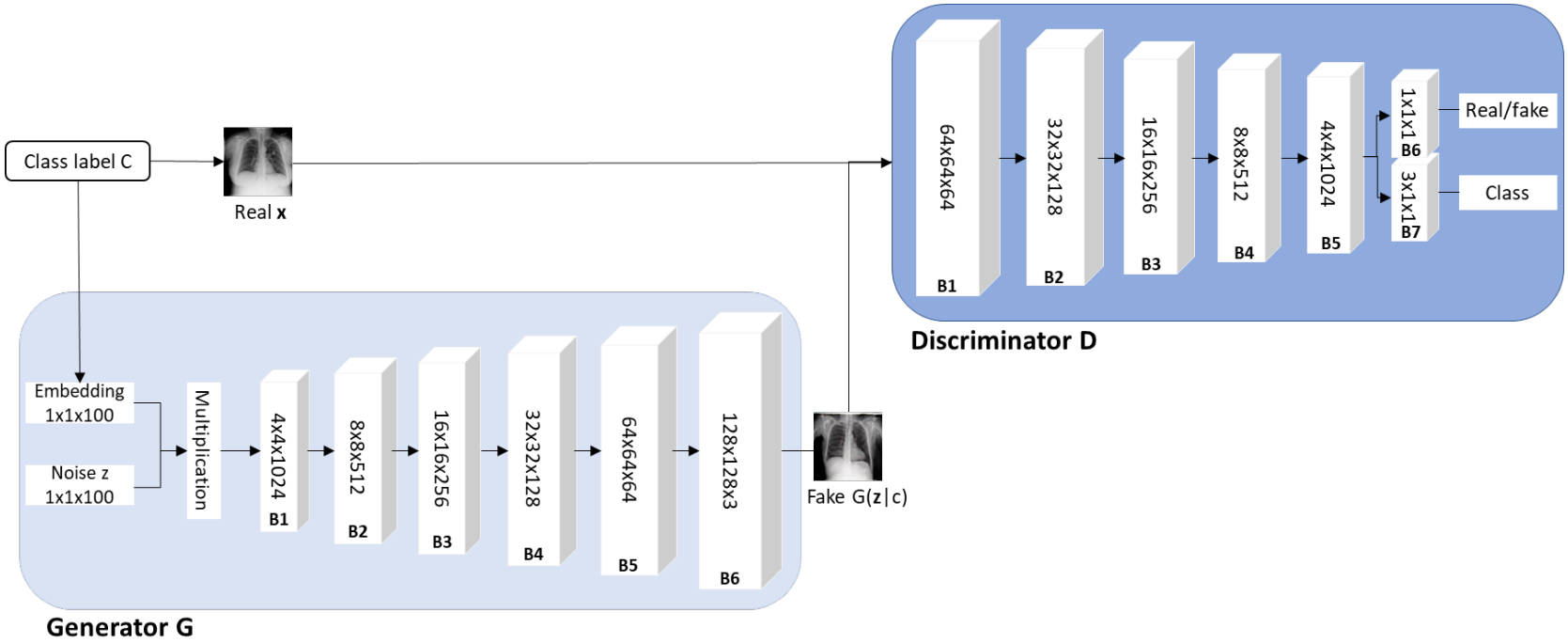


Figure 7: Overview of the customized AC-GAN architecture with the generator G and discriminator D. The dimensions ($H \times W \times C$) on the blocks denote the dimensions of that block's output tensor, where H and W are the height and width of the tensor in pixels respectively, and C denotes the number of channels. G takes a class label c and noise vector $\boldsymbol{z}$ as input and upsamples the input to a fake $128 \times 128 \times 3$ image $G(\boldsymbol{z}|c)$ using six blocks B_1,...,B_6. D takes either a real image $\boldsymbol{x}$ or a fake image $G(\boldsymbol{z}|c)$ as input and downsamples this through blocks B_1,...,B_7 to predict the source and class probabilities.

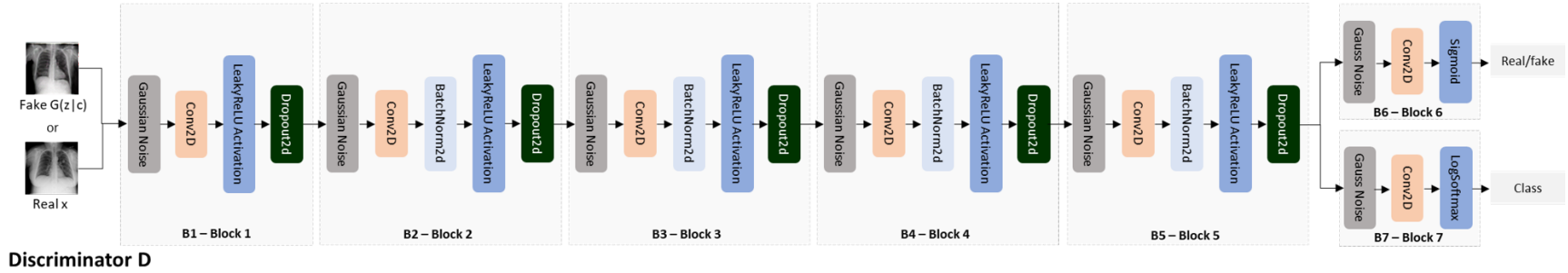


Figure 8: The proposed architecture of the AC-GAN discriminator D. D takes either a real image $\boldsymbol{x}$ or a fake image $G(\boldsymbol{z}|c)$ as input and downsamples this to predict the source and class probabilities using convolutional layers, batch normalization layers, activation functions, dropout layers and Gaussian layers.

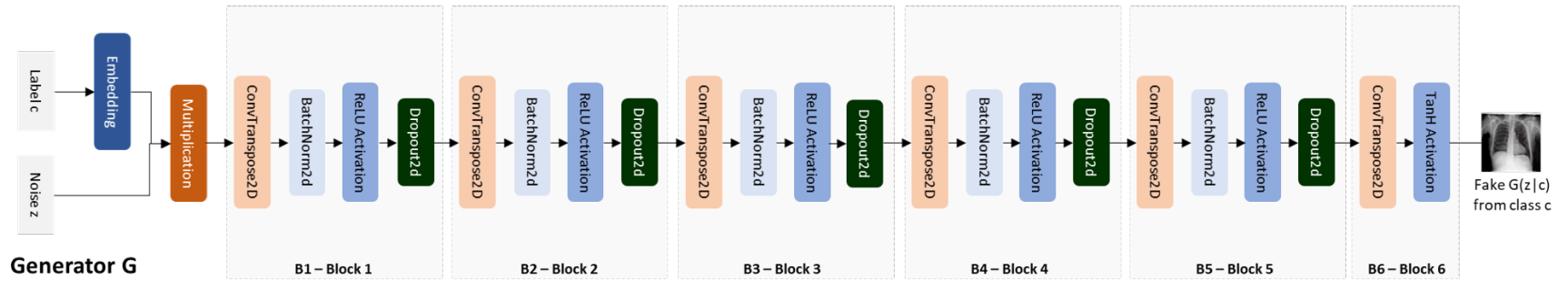


Figure 9: The proposed architecture of the AC-GAN generator G. G takes a class label c and noise vector $\boldsymbol{z}$ as input and upsamples the input to a fake $128 \times 128 \times 3$ image using six blocks that contain transposed convolutional layers, batch normalization layers, activation functions and dropout layers.

4.1.1 Discriminator D

The proposed architecture of the AC-GAN discriminator D is shown in Figure 8. The customized AC-GAN discriminator consists of 30 modules and contains approximately 11.2 million trainable parameters. The discriminator D takes an image as input and outputs a prediction if the image is real or fake (the source probability) together with a probability distribution over the class labels (the COVID-19 pneumonia, non-COVID-19 pneumonia and normal classes). Either a real image $\boldsymbol{x}$ or fake image $G(\boldsymbol{z}|c)$ of size $128 \times 128 \times 3$ is first forwarded through block B_1 to downsample the input image to $64 \times 64 \times 64$. This block consists of a Gaussian noise layer, a convolutional layer, a Leaky Rectified Linear Unit (LeakyReLU) activation function and a dropout layer. This block does not contain a batch normalization (batchnorm) layer, as Radford et al. (2015) found that applying batchnorm to all layers resulted in model instability. Next, the tensor is passed through blocks B_2 to B_5 to further downsample the image to size $4 \times 4 \times 1024$. Each of these blocks consists of a Gaussian noise layer, a convolutional layer, a batchnorm layer, the LeakyReLU activation and a dropout layer. Next, the source and class labels are predicted. The first output block B_6 consists of a Gaussian noise Layer, convolutional layer and it uses a Sigmoid function to predict the source of the image. The second output block B_7, the auxiliary classifier Q, uses a Softmax function to predict the class label. Note that the Gaussian noise layers and dropout layers are not used in the original implementations of DCGAN (Radford et al., 2015) or AC-GAN (Odena et al., 2017), but we add these layers to our customized AC-GAN to increase model stability. We elaborate on each layer/module used in the discriminator in the following sections. Appendix B gives an overview of the output size and the number of trainable parameters of each layer in D.

Gaussian Noise Layer

As stated before, a GAN's training process is often unstable and GANs are hard to train to convergence. Sønderby, Caballero, Theis, Shi, and Huszár (2016) found that adding instance noise to the inputs' pixels in the discriminator stabilizes training. Adding noise is a theoretically motivated way to remedy the poor convergence properties of GANs, and we refer to Sønderby et al. (2016) for its mathematical proof. The idea behind this method is to make the discriminator's job harder, as the discriminator often becomes too strong too quickly for the generator to learn anything and thus prevents the GAN from converging. We want to prevent the discriminator D from becoming too strong as this causes vanishing gradients, and we do this by contaminating its inputs.

Figure 10 illustrates this method, with real image $\boldsymbol{x}_{\text{real}}$, synthetic image $\boldsymbol{x}_{\text{fake}} = G(\boldsymbol{z}|c)$, $\boldsymbol{z}$ the latent vector that is forwarded through G and y the binary source label that represents whether the image is real or fake. Also, $\boldsymbol{x}$ is the image that is forwarded through the discriminator and it is either $\boldsymbol{x}_{\text{real}}$ or $\boldsymbol{x}_{\text{fake}}$, depending on the corresponding value of y. The noisy version of $\boldsymbol{x}$ is denoted by $\tilde{\boldsymbol{x}}$, and it is generated by adding Gaussian noise to $\boldsymbol{x}$. Adding the instance noise to create $\tilde{\boldsymbol{x}}$ weakens the discriminator as it now becomes harder to distinguish between $\boldsymbol{x}_{\text{real}}$ and $\boldsymbol{x}_{\text{fake}}$ because the noise added to both the real and fake images is sampled from the same Gaussian distribution with mean $\mu = 0$ and standard deviation $\sigma = 0.1$. The standard deviation σ is annealed linearly during training, as is done in Sønderby et al. (2016). In our implementation of the discriminator, we do not only add noise to the input image of the discriminator in block B_1, but we add Gaussian noise to each input tensor in the discriminator (in blocks B_1 to B_7) as this further stabilizes training (Salimans et al., 2016). We implement the module `GaussianNoise` in `PyTorch`, adapted from `https://github.com/ShivamShrirao/facegan_pytorch` (Shrirao, 2021). When forwarded an input feature map $\boldsymbol{x}$, `GaussianNoise` returns $\tilde{\boldsymbol{x}}$, where random noise is added to the tensor's pixels. This layer does not contain any parameters that are learned during training. After an input tensor is contaminated with noise in the Gaussian noise layer, it is forwarded through the convolutional layer to be downsampled.

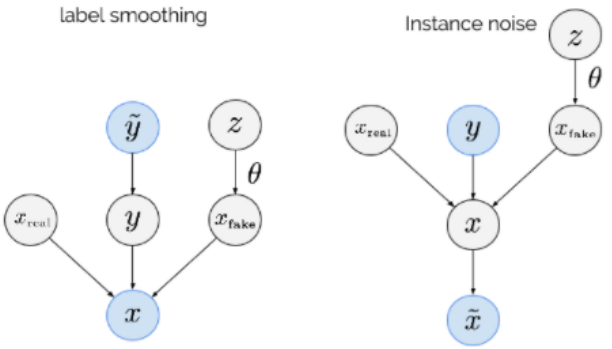


Figure 10: Graphical illustration of the label smoothing and instance noise methods. With label smoothing, the positive source label y is contaminated. With instance noise, an input tensor $\boldsymbol{x}$ in D is contaminated with Gaussian noise. Image from Huszár (2016)

Another way to make the discriminator's job harder is by adding noise to the (target) source label y, or equivalently, one-sided label smoothing (Salimans et al., 2016). In this technique, only the positive labels $y = 1$ (corresponding to real images) in the discriminator's training data are replaced with a smoothed value like 0.9. Deep neural networks are prone to producing highly confident predictions, resulting in low training losses that cause vanishing gradients (Goodfellow et al., 2014), and this technique can be used to encourage D to estimate soft probabilities rather than to extrapolate to extremely confident source predictions (Goodfellow, 2016). This is illustrated in Figure 10, where $\tilde{y}$ denotes the smoothed source label. We apply this method during the training of the GAN to increase training stability and we implement it by replacing the target source label for real images ($y = 1$) with $\tilde{y} = 0.9$ in the source loss function of D.

Convolutional Layer

A convolutional neural network (CNN) is a class of deep neural networks that is typically used to recognize patterns in images (LeCun et al., 1989). One of the layers in the CNN is a convolutional layer and it performs an operation called a "convolution". This operation uses learnable filters (kernels) to extract features from images. Each convolutional layer has its own set of kernels, and thus extracts different features from its input (Albawi, Mohammed, & Al-Zawi, 2017). The kernels' weights and biases are tuned during training of the network. The convolution decreases the image size, i.e., downsamples the image, while retaining all features in the image in a smaller set of pixels. The input of a convolutional layer is a tensor and it outputs a smaller tensor. We use the convolutional layers that are used in DCGAN in our implementation of the AC-GAN for their performance (Radford et al., 2015).

Before we elaborate on this operation, we discuss the structure of input images and intermediate/output tensors. All tensors are stored as matrices of pixel values and they feature three axes $[H \times W \times C]$, the height H, width W and the number of channels C for which ordering matters (Dumoulin & Visin, 2016). Tensors contain C different $H \times W$ feature maps that are stacked onto another, one for each channel. A convolution is a transformation that preserves this notion of ordering and takes the different axes into account (Dumoulin & Visin, 2016). The neurons in a convolutional layer are also arranged in these three dimensions. For example, a $128 \times 128 \times 3$ input image is represented by three 128×128 feature maps (three matrices filled with pixels) and the size of the input layer is $128 \times 128 \times 3$. The channel axis is thus used to access the red, green and blue channels of an image. The size of a hidden layer (in $B3$) in the discriminator is $16 \times 16 \times 256$, where the number of feature maps is 256 and the height and width of the feature maps equals 16. A convolutional layer's output is affected by the shape of its input as well as the choice of the kernel, stride and zero padding (Dumoulin & Visin, 2016). We discuss all aspects in this section.

The kernel is a matrix containing learnable weights that slides over the input to extract features from the input tensors. The kernel's weights can be learned by the network to extract different kinds of features. The kernel convolves with the receptive field, which is the overlapping area of the kernel and the input feature map at a specific kernel position. Neurons in the convolutional layer are locally connected with neurons in the previous layer via the kernel. The kernel is applied to multiple positions in the previous layer so the weights in the kernel are shared across the layer, reducing the number of parameters in the network. In Figure 11, the receptive field at a specific

kernel position is illustrated by the shaded area (dark grey) in the input feature map (light grey). To keep the illustration simple, a single input feature map and single kernel are represented, but usually multiple stacked feature maps convolve with several sets of kernels. At each kernel position, the dot product is computed between the kernel and the receptive field. Each element/weight of the kernel is multiplied with the input element it aligns with in the receptive field. The results are summed up to obtain the output in the current position. Figure 12 gives an example of how the output is computed at the first kernel position, where the input size is 6×6 and a 4×4 kernel is used. The combined outputs for all kernel positions constitute the output feature map, represented by the blue grid in Figures 11 and 12. A bias is often added to the output before passing it to the next layer in the neural network.

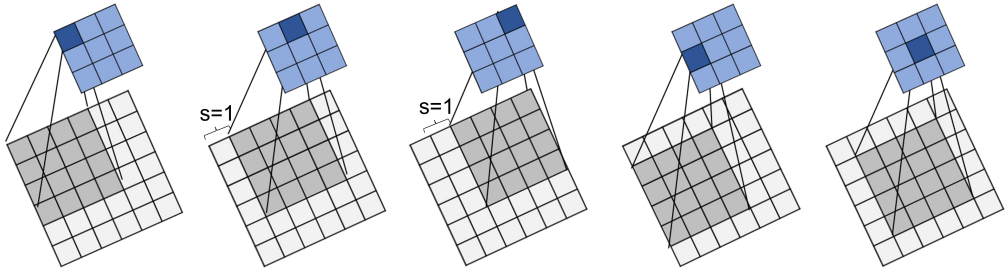


Figure 11: Illustration of part of a convolution of a 4×4 kernel over a 6×6 input feature map using unit stride and no zero padding ($k = 4$, $s = 1$, and $p = 0$). The receptive field is the shaded grey area in the input feature map (the light grey area). The output is represented by the 3×3 feature map, where the current output location is the shaded area in the blue grid.

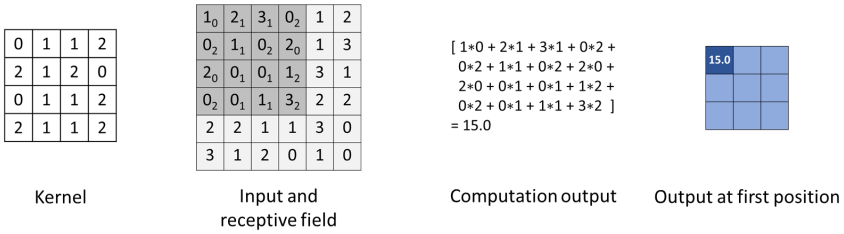


Figure 12: Computing the first output value of the convolution, which is the dot product between the kernel and the current receptive field (the dark grey shaded area in the input feature map).

As stated before, the input images/tensors actually consist of C input feature maps that are stacked onto another, one for each channel. The kernel of an input is therefore also C-dimensional, as each one of the input feature maps/channels convolves with a distinct kernel. For example, a $16 \times 16 \times 256$ tensor contains 256 channels and thus convolves with 256 different kernels. Also, the stacked input feature maps often convolve with different sets of kernels to create multiple output feature maps. A convolutional layer is thus always associated with a set of filters. Feature maps learned by different filters are stacked along the channel dimension. The number of channels of

the output (or the number of feature maps to be learned) is given by C_{out}. The size of the kernel is $C_{out} \times k_h \times k_w \times C$, where k_h and k_w are the height and width of the kernel, respectively. An example of a convolution mapping of two input feature maps to three output feature maps using a $3 \times 3 \times 3 \times 2$ set of kernels is given in Figure 13. Each kernel in set W_1 convolves with one of the input feature maps to create two intermediate feature maps. Kernel $W_{1.1}$ convolves with the first input feature map (light grey) while kernel $W_{1.2}$ convolves with the second input feature map (dark grey), and both operations create one intermediate feature map. The intermediate results associated with one set of kernels (i.e. set W_1) are summed elementwise to form the (first) output feature map. The same is done for the kernel sets W_2 and W_3 to form the second and third output feature maps, and all three are stacked together to form the output. We show only a single feature map and kernel for simplicity and illustration purposes in the remainder of this section.

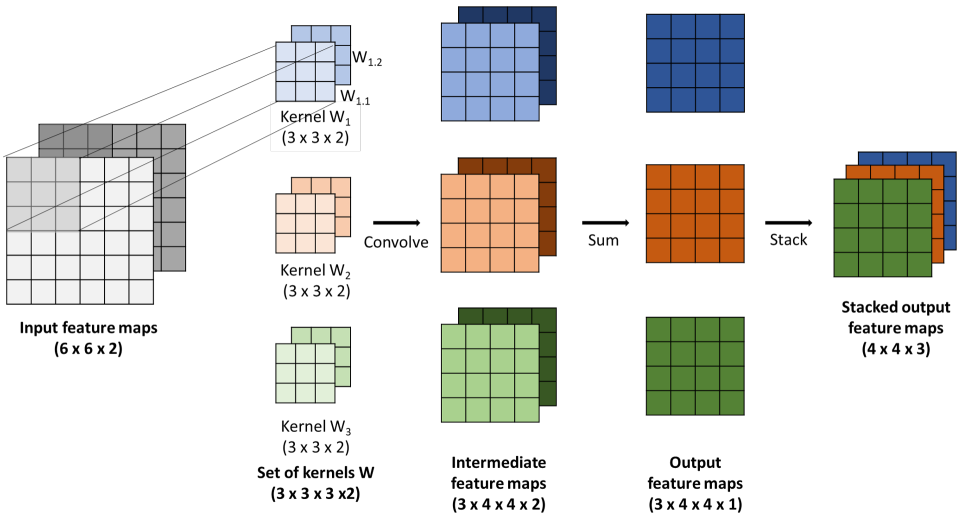


Figure 13: Illustration of a convolution mapping from two input feature maps (size $6 \times 6 \times 2$) to three output feature maps (size $4 \times 4 \times 3$) using a $3 \times 3 \times 3 \times 2$ set of kernels $\boldsymbol{W}$ with stride 1 and no zero padding. For example, each kernel in set W_1 is convolved with one of the input feature maps to create one intermediate feature map each. These two blue intermediate feature maps are summed up elementwise to create one blue output feature map. The three output feature maps created with each kernel set are stacked together to create the $4 \times 4 \times 3$ output.

To decrease the number of parameters in the network and reduce computational costs, the operator stride that acts as a subsampling operator was introduced (Albawi et al., 2017). The stride s is the distance between consecutive positions of the kernel in the input feature map and reduces the overlap of consecutive receptive fields and spatial dimensions of the output.

One of the shortcomings of the convolution is the loss of information that occurs on the borders of the image (Albawi et al., 2017). When the kernel moves over an image, the pixels on the edge of the image are used less in the convolution. The features on the edge of the image are thus

captured less than the features in the middle of the picture. To overcome this issue, padding was introduced. The degree of padding p is the number of zeros inserted at the beginning and end of an axis (Dumoulin & Visin, 2016). By adding those extra zeros, the kernel also captures the features at the edges of the image more often and it allows for a more accurate analysis of the images.

The operators stride and padding can be used to control the output size. With images, we are dealing with 2-dimensional convolutions as the kernel slides across the height and width of each input feature map. With h the height and w the width, we have square input size $i_w = i_h = i$, square kernel size $k_w = k_h = k$, strides $s_w = s_h = s$ and the same padding along each axis $p_w = p_h = p$ in our discriminator. The output size of each axis can be determined by

$$o = \left\lfloor \frac{i + 2p - k}{s} + 1 \right\rfloor, \tag{9}$$

where $o = o_h = o_w$ is output size for both the height and width axes.

We employ two different convolutional layers in our discriminator, as this is also done in the DCGAN architecture (Radford et al., 2015). The simplest convolution is used in the blocks B_6 and B_7 from Figure 8. The convolutional layers in these blocks use a kernel size $k = 4$, stride $s = 1$ and padding of $p = 0$. Figure 11 gives an example of such a convolution, where the input size is $i = 6$. Note that we only show part of the convolution. In this convolution, the kernel slides across every position of the input as the stride equals 1. Using equation (9), the output size simplifies to $o = (i - k) + 1$, such that the output size in our example equals $o = (6 - 4) + 1 = 3$. The output feature map is illustrated by the blue grid, where each blue shaded position represents the current output location. At each location, the product between each element in the kernel and the corresponding element from the receptive field is computed and all products over the receptive field are summed to obtain the output in the current location. An example of how the output at one specific kernel location is computed is given in Figure 12.

The kernel size, stride, and padding of the convolutions used in blocks B_1 to B_5 in the discriminator from Figure 8 are $k = 4$, $s = 2$ and $p = 1$, respectively. Figure 14 gives an example of such a convolution, where the input size equals $i = 6$. Again, only part of the convolution is illustrated. The light grey area represents the input feature map, where the shaded area illustrates the receptive field. The dashed blocks represent the zero padding, where 1 layer of zeros is added at the beginning and end of each axis (the height and width). The blue grid again illustrates the out-

put. Using equation (9), the output size equals $o = \left\lfloor \frac{i+2p-k}{s} + 1 \right\rfloor = \left\lfloor \frac{6+2*1-4}{2} + 1 \right\rfloor = \left\lfloor \frac{4}{2} + 1 \right\rfloor = 3$. The kernel now does not slide across every position of the input feature map (concatenated with the padding), but the distance between two consecutive positions of the kernel over the input is now $s = 2$, as can be seen in Figure 14. An example of how the output at one specific location is computed is given in Figure 15.

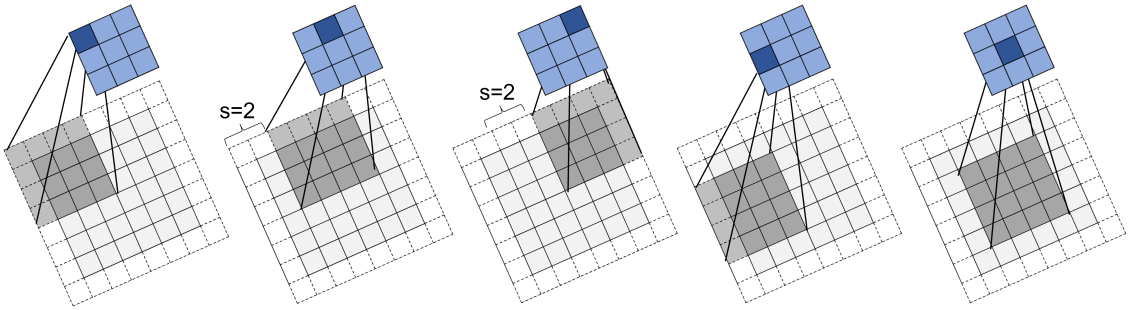


Figure 14: Illustration of part of a convolution of a 4×4 kernel over a 6×6 input feature map using non-unit stride and padding ($k = 4$, $s = 2$, and $p = 1$). The receptive field is the shaded area in the input feature map (the grey area), and the zero padding is depicted the dashed blocks around the input feature map. The output is represented by the blue 3×3 feature map, where the current output location is the shaded area.

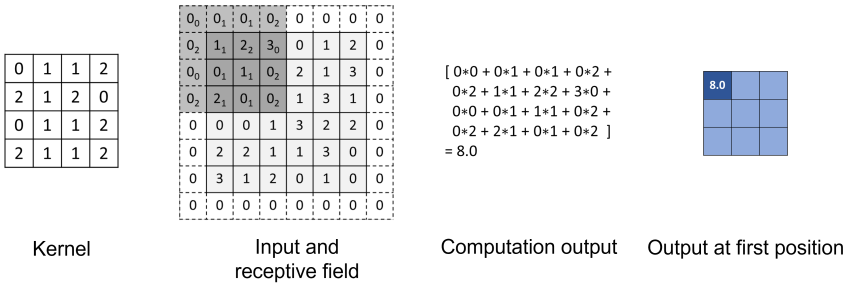


Figure 15: Computing the first output value of the convolution, which is the dot product between the kernel and the current receptive field (the dark grey shaded area in the input feature map concatenated with the zero padding).

As can be seen from the examples in Figures 11 and 14, the output size is smaller than the input size and the input feature map has been downsampled. The output size depends on the input, kernel, stride and padding. The weights in the kernels and the biases are learnt during training, which we will elaborate on in Section 4.3. We use the `Conv2d` module from the `torch` package in `PyTorch` to implement the convolutional layers in D (Collobert, Kavukcuoglu, & Farabet, 2011).

Batch Normalization Layer

After the input tensor is downsampled via the convolutional layer, the tensor is passed through the batch normalization layer. The authors of DCGAN recommend to use batch normalization (batch-norm) in both the generator and the discriminator (Radford et al., 2015). Batch normalization mitigates the effect of unstable gradients within deep neural networks and stabilizes learning (Ioffe

& Szegedy, 2015). It prevents mode collapse from happening, which occurs when the generator can only produce a small set of outputs. Also, networks that use batchnorm are more robust to bad initialization. Radford et al. (2015) found that applying batchnorm to all layers in the discriminator and generator resulted in model instability. We therefore do not apply batchnorm to the input layer in the discriminator and output layer of the generator.

Batch normalization standardizes and normalizes the input tensors in a batch to have zero mean and unit variance over each channel. Next, the normalized feature maps are rescaled and shifted with a scaling parameter γ and shifting parameter β. The batch normalization operation for one input feature map $\boldsymbol{x}$ computed over one channel in the mini-batch can be defined as

$$y = \gamma * \frac{x - \mu}{\sqrt{\sigma^2 + \epsilon}} + \beta, \tag{10}$$

where y is the normalized feature map and γ and β are the scaling and shifting parameter, respectively (Ioffe & Szegedy, 2015). Also, μ and σ^2 denote a batch's computed mean and the variance over a channel and ϵ is a small numerical value added to the denominator for numerical stability. Radford et al. (2015) found that removing the scaling and shifting parameter from equation (10) produced better results. We therefore set the parameter γ to 1 and the parameter β to 0. We set $\epsilon = 0.0005$, which is the default choice for this value. We use `BatchNorm2d` from the `torch` package in `PyTorch` to implement batch normalization in G and D (Collobert et al., 2011). `BatchNorm2d` is specifically designed for image-like tensors that are stacked along the channel axis. It normalizes input tensors in a batch to have 0 mean and unit variance over the channel dimensions.

Activation Functions - LeakyReLU, Sigmoid and LogSoftmax

After the stacked feature maps are passed through the convolutional layer and are normalized via batchnorm, the downsampled tensor is passed through an activation function to produce the output of a block. Activation functions are simple mathematical functions that introduce non-linearity into the network, such that networks can approximate any function. Activation functions thus give rise to flexible network architectures. As it is best practice in training GANs, the LeakyReLU activation function is used in blocks B_1 to B_5 from the discriminator, and the Sigmoid and LogSoftmax functions are used in B_6 and B_7, respectively (Radford et al., 2015).

The Leaky Rectified Linear Unit (Leaky ReLU) is used as Radford et al. (2015) found that this function works well in the discriminator. This function is based on the Rectified Linear Unit

(ReLU), but it has a small slope for negative values instead of a flat slope. With ReLU, the Dying ReLU problem is often present, where neurons become inactive and only output 0 for any input. To overcome this problem, the Leaky ReLU function was introduced, which is defined as

$$\text{LeakyReLU}(x) = \begin{cases} x & \text{if } x \geq 0 \\ \alpha * x, & \text{otherwise} \end{cases} \tag{11}$$

where α is the (non-learnable) slope parameter. We set $\alpha = 0.2$, as done in Radford et al. (2015).

The Sigmoid activation function is mostly employed for binary classification. As the output from block B_6 is the source probability prediction of an image (a binary classification problem), i.e., whether an image is real, the Sigmoid function is used in this block. The output of the Sigmoid function is a value between $[0, 1]$, which corresponds to the predicted probability that the input image is real. The Sigmoid non-linearity is defined as $\text{Sigmoid}(x) = \frac{1}{1+\exp(-x)}$.

LogSoftmax is mostly used for multi-class classification, and we use it to predict the class label of the input image in block B_7. The output of the LogSoftmax function is a vector (the probability distribution over the classes) where all values are between $[0, 1]$ and all elements of the output vector sum to 1. The probability that the input belongs to class i is given by $\text{LogSoftmax}(x_i) = \log\left(\exp(x_i)/\sum_{j=1}^{C}\exp(x_j)\right)$, where C is the number of classes in the data. We use the `LeakyReLU`, `Sigmoid` and `LogSoftmax` from `torch` in `PyTorch` to implement the activation functions (Collobert et al., 2011). Next, the transformed tensors are passed through a dropout layer.

Dropout Layer

Next, we add dropout layers after the activation functions in the discriminator of our customized AC-GAN. We add dropout to all blocks in D except for blocks B_6 and B_7. For image-like tensors, dropout is a regularization method where randomly selected channels are dropped from the neural network. The addition of dropout helps reduce the capacity of the network during training and helps avoid overfitting (Srivastava, Hinton, Krizhevsky, Sutskever, & Salakhutdinov, 2014). The probability of a channel being zeroed out is 30%. Note that the original implementations of DCGAN (Radford et al., 2015) and AC-GAN (Odena et al., 2017) do not contain dropout layers. We implement the dropout layer with `Dropout2d` from the `torch` package in the `PyTorch` library (Collobert et al., 2011), which is specifically designed to effectively apply dropout to image-like tensors. We also add dropout layers to all blocks of the generator, except for the last block B_6.

4.1.2 Generator G

Our implementation of the AC-GAN generator is shown in Figure 9. The custom AC-GAN generator consists of 23 modules and contains approximately 12.8 million trainable parameters. The class label is first forwarded through an embedding layer of 100 dimensions to include class conditional information in the image generation process. This embedding vector is then multiplied with the latent vector $\boldsymbol{z}$ and is of size $1 \times 1 \times 100$. The tensor created from this multiplication is passed though six different blocks to generate a fake image of size $128 \times 128 \times 3$. The $1 \times 1 \times 100$ embedding/noise tensor is first upsampled to $4 \times 4 \times 1024$ in the first block B_1. The tensor is further upsampled to size $64 \times 64 \times 64$ from block B_2 to B_5. Each of these blocks consists of a transposed convolutional layer, followed by a batch normalization layer, a ReLU activation and a dropout layer. The last block upsamples the tensor to $128 \times 128 \times 3$ and does not contain a batch normalization layer or dropout layer and is followed by a TanH activation layer, following the implementation of the generator in Radford et al. (2015). Appendix C gives an overview of the output size and the number of trainable parameters of each layer in G. We elaborate on each different type of layer used in the generator in the following sections.

Embedding Layer

First, the class label c is fed to the generator together with the latent vector $\boldsymbol{z}$. To incorporate the class label into the image generation process, the embedding layer is used. The embedding layer is a simple trainable lookup table that stores embeddings (containing the learnable weights) of a fixed dimension for each of the classes. The trainable weights of the module have shape 3×100 (the number of classes $\times$ the embedding dimension) and are randomly initialized from $\mathcal{N}(0, 1)$. The input to the module is a class label and it returns the corresponding class embedding from the lookup table. In our model, the embedding size equals the size of the latent vector and is equal to $1 \times 1 \times 100$. We use `Embedding` from `torch` to implement the embedding layer in G (Collobert et al., 2011). The embeddings stored in this lookup table are randomly initialized and are independent of each other, so the embeddings do not contain any information about the relations between classes. In Section 4.2.3 we elaborate on how hierarchy-aware class embeddings are computed in the embedding layer of HAC-GAN to take the relations between classes into account.

Next, the class embedding is multiplied with the noise vector $\boldsymbol{z}$ to produce the input to G. The generator is now conditioned on a certain class when generating a new image. Note that with multiplication, the embedding and noise vector are merged and the output shape remains the same.

In many GANs, the latent vector and embedding are combined by concatenation (Waheed et al., 2020; Al-Shargabi et al., 2021). When concatenating, the dimensions of the noise and embedding vector are added, resulting in a different dimension. In practice, the images generated in a GAN that employs concatenation do not showcase much variety. With multiplication, many different images per class are often generated. We therefore use multiplication to combine the embedding and noise in G. Next, the multiplicated input tensor is forwarded through the transposed convolutional layer to be upsampled.

Transposed Convolutional Layer

We use another type of CNN layer to increase (upsample) the spatial dimensions of input feature maps in the generator. This operation again uses learnable kernels that are shared across the layer to extract features from images. The kernels' weights and biases are tuned during training of the network. In contrast to the convolution operation that downsamples the input, the transposed convolution produces an output that is larger than the input (Zeiler, Krishnan, Taylor, & Fergus, 2010). The transposed convolutional layer basically does exactly what the regular convolutional layer does, but on a modified input map and with modified stride and padding. Applying a transposed convolution to the original input with operators s, p and k is equivalent to applying a convolution to a modified input feature map with operators zero insertions z, padding p', stride s' and kernel of size k, where $z = s - 1$, $p' = k - p - 1$ and $s' = 1$. The input map is modified by using the zero insertions operator z. We only show a single feature map and kernel for illustration purposes in the remainder of this section, but the input tensors again consist of multiple stacked feature maps that convolve with several sets of kernels, just as in the convolutional layer.

The number of zero insertions z defines how many zeros are inserted between each row and column of the input, where z is defined as $z = s - 1$. An example of how the modified input is generated is given in Figure 16, where the input is of size 4×4 and a stride of $s = 2$ is used in the transposed convolution. In our example, z equals $z = s - 1 = 1$, so one zero is inserted between each row and column of the original input feature map to create the modified input feature map.

Now that a modified input feature map is created, a regular convolution is applied on the modified input with a modified value for the stride s' and padding p'. The new stride always equals $s' = 1$. The new padding operator p' is defined as $p' = k - p - 1$ and denotes the number of zeros added to the beginning and end of an axis of the modified input feature map before applying the convolution. The output size is calculated as $o = s(i - 1) - 2p + k$ (Dumoulin & Visin, 2016).

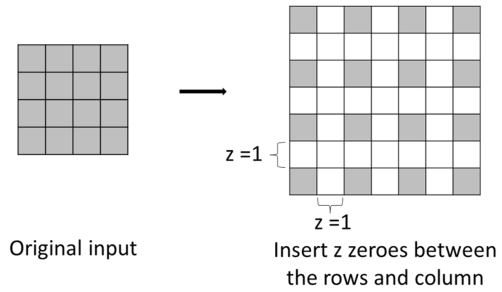


Figure 16: Example of how a modified input feature map is created. The input is of size 4×4 and a stride of 2 is used in the transposed convolution. As $z = s - 1 = 1$, one zero is inserted between each row and column of the original input. The white pixel blocks represent the inserted zeros. The modified input image is of size 7×7.

Consider our example with the 4×4 original input feature map, modified input feature map from Figure 16, 4×4 kernel, modified stride $s' = 1$, and zero padding $p' = 4 - 1 - 1 = 2$. Part of the corresponding transposed convolution is illustrated in Figure 17. The output for each kernel position is now computed by taking the dot product (elementwise multiplication) between the kernel and current receptive field in the modified input feature map. This is done in the same way as in the examples given in Figure 12 and 15. The output of our example in Figure 17 with input size $i = 4$, kernel size $k = 4$, a stride of $s = 2$ and the original degree of padding $p = 1$ has size $o = s(i-1) - 2p + k = 2 * 3 - 2 + 4 = 8$. The spatial dimension of the output (8×8) is larger than the dimension of the original input (4×4) and the input feature map has been upsampled.

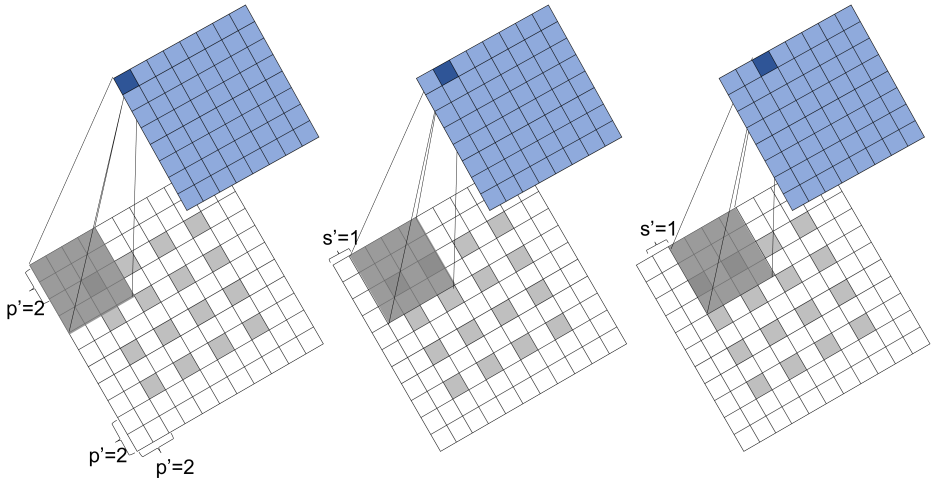


Figure 17: Illustration of part of a transposed convolution of a 4×4 kernel over a 4×4 original input with original stride $s = 2$ and original degree of padding $p = 1$. This is equivalent to convolving a 4×4 kernel over a modified 7×7 input feature map (with $z = s - 1 = 1$ zero inserted between each row and column) with a padding of $p' = k - p - 1 = 2$ and stride $s' = 1$. The receptive field is the shaded area in the modified input feature map concatenated with the zero padding (the grey area). The output is represented by the blue 8×8 feature map, where the current output location is the shaded area.

In our generator, two types of transposed convolutional layers are used as this is best practice in training GANs (Radford et al., 2015). The simplest transposed convolution is used in block B_1 from Figure 7 and 9. The transposed convolutional layer in this block uses a kernel of size $k = 4$, stride $s = 1$ and padding of $p = 0$. The 100-dimensional input vector (the multiplication of the noise vector $\boldsymbol{z}$ and the embedding for class c) is upsampled to a $4 \times 4 \times 1024$ tensor. Next, the $4 \times 4 \times 1024$ tensor is further upsampled to a $128 \times 128 \times 3$ image in blocks B_2 to B_6. The transposed convolutional layers in blocks B_2 to B_6 use a kernel of size $k = 4$, stride $s = 2$ and padding of $p = 1$ (Radford et al., 2015). We use the `ConvTranspose2d` module from `torch` in `PyTorch` to implement the 2-dimensional transposed convolutional layers in G (Collobert et al., 2011). The outputs of the transposed convolutional layers in blocks B_1 to B_5 are passed through batch normalization layers to normalize the outputs. Next, the outputs are forwarded through the activation functions.

Activation Functions - ReLU and TanH

As stated before, activation functions give rise to flexible network architectures. The Rectified Linear Unit (ReLU) activation is used in all blocks in the generator except for the last block B_6, which employs the Hyperbolic Tangent (TanH) activation function. Since its introduction in NNs by Glorot, Bordes, and Bengio (2011), the ReLU function is the most popular activation function in neural networks. The ReLU function is defined as $\text{ReLU}(x) = \max(0, x)$ (Nair & Hinton, 2010).

Radford et al. (2015) found that using a bounded activation function in the output layer allows the network to learn more quickly to saturate, which greatly shortens the learning cycle. The TanH function is therefore employed to return all pixel values in the generated image to the input data range of $[-1, 1]$. The TanH activation function is defined as $\text{Tanh}(x) = \frac{\exp(x) - \exp(-x)}{\exp(x) + \exp(-x)}$. We use the `ReLU` and `Tanh` modules from `torch` to implement the activations in the generator (Collobert et al., 2011). The transformed tensors are next passed through a dropout layer in blocks B_1 to B_5.

4.2 Hierarchical AC-GAN (HAC-GAN)

Many existing GANs, including the customized AC-GAN from the previous section, model images that belong to distinct classes independently. The embedding layer is used to include class label c into the image generation process of AC-GAN. However, it is common to have some sort of relation between different classes in real-world datasets. The classes can often be organized hierarchically based on these relations and this hierarchy contains valuable information. Incorporating class hierarchy into the image generation process could thus improve the quality and diversity of

the generated images and we therefore introduce the Hierarchical AC-GAN (HAC-GAN). In this section, we explain how we incorporate class hierarchy in the form of a hierarchy-aware embedding layer in the GAN. The implementation of HAC-GAN is based on our customized implementation of AC-GAN from Section 4.1, their only difference being the used embedding layer.

The hierarchical structure of the classes can be represented by a graph, which is an ordered pair $G(V, E)$ with sets of nodes V and edges $E \subseteq V \times V$ that model the pairwise relations between the nodes. An edge $(u, v) \in E$ represents that v is a subset of u. In this research, the classes can be represented by the graph structure pictured in Figure 2, with the classes of interest, i.e., COVID-19 pneumonia, non-COVID-19 pneumonia, and normal on the leaf nodes. Directly using a graph in its raw form as input in the embedding layer of AC-GAN is difficult as its typical input is numerical. To incorporate class hierarchy in the GAN we implement a graph embedding method that learns embeddings for nodes in a graph while capturing the hierarchical relations between the classes.

We use the graph embedding method Node2Vec to learn numerical vector representations (embeddings) of nodes in a graph. Node2Vec preserves the original structure of the graph such that (dis)similar nodes will have (dis)similar embeddings (Grover & Leskovec, 2016). Node2Vec optimizes a graph-based objective function that is motivated by Word2Vec, prior work on natural language processing (Mikolov, Chen, Corrado, & Dean, 2013). To understand how Node2Vec learns hierarchy-aware embeddings, an understanding of Word2Vec is necessary. We first elaborate on this method in Section 4.2.1. Node2vec employs a 2^{nd} order random walk approach to generate the input from the graph for the Word2Vec algorithm (Grover & Leskovec, 2016). Section 4.2.2 elaborates on the 2^{nd} order random walks and how Node2Vec learns embeddings for each class. Section 4.2.3 elaborates on how these learned embeddings are used in HAC-GAN's embedding layer.

4.2.1 Word2Vec

Node2Vec is heavily inspired by Word2Vec's Skip-gram model, so we elaborate on Word2Vec in this section. Word2Vec is a commonly used word embedding method that captures complex relations and is relatively fast. It is a two-layer neural network that enables the learning of vector representations, i.e., word embeddings for each word in a text. These embeddings represent the words in a multi-dimensional vector space where similar words are mapped close to each other (Mikolov, Chen, et al., 2013). The underlying assumptions of Word2Vec are that the meaning of a word depends on its context, and the embeddings are able to capture the meaning of a word

such that (dis)similar words have (dis)similar encodings. Two words that share in context thus share in meaning, which results in a similar vector representation from the model. There are two main architectures that enable the success of this embedding method; Continuous Bag of Words (CBOW) and the continuous Skip-gram model (Mikolov, Chen, et al., 2013). Node2Vec is inspired by the Skip-gram model, so we only elaborate on this model.

Skip-gram is a two-layer neural network with one hidden layer who's goal it is to learn a word embedding for each word in the vocabulary. It takes a word from the text as input and it outputs the probability predictions of other words in the vocabulary occurring in the context of the input word (Mikolov, Chen, et al., 2013). The predicted probabilities are used to compute the loss function that the network aims to minimize, and the weights of the network are updated by applying backpropagation and stochastic gradient descent (SGD) (Rong, 2014). The weights of the network that are learned during training represent the word embeddings, which capture various characteristics of the words in a limited dimension. We first discuss the architecture of the network, and later how the weights over the network are iteratively updated.

Skip-gram - Notation and Architecture

Figure 18 presents the architecture of the Skip-gram neural network. The layers are fully connected. The number of neurons in the hidden layer is d, which equals the embedding dimension. The vocabulary V is the set of unique words w in the text, with $v = |V|$ the size of the vocabulary. The set of words surrounding and including a word define the context of that word. The context includes words to the left and/or right of the input word. We denote the context with C, where $c = |C|$ is the context size. The size of the context depends on the window size, which is defined as the maximum distance between words in the context window with the input word in the center. The context size does not always equal the window size, depending on the position of the input word in the text. In Figure 19 for example, in the text "Today is a great day" we first take "Today" as the input word with a window size of three. The context of size two is given by "Today is" as the input word is the first word in the text. Next, if we take "is" as center word, the context is given by "Today is a" and its size equals the window size.

The input layer of the Skip-gram network in Figure 18 is a one-hot encoded vector where given an input word w_i, only the node x_i in $\{x_1, ..., x_v\}$ that represents the input word is equal to one. All other nodes are zeros. The weights between the input layer and hidden layer are represented

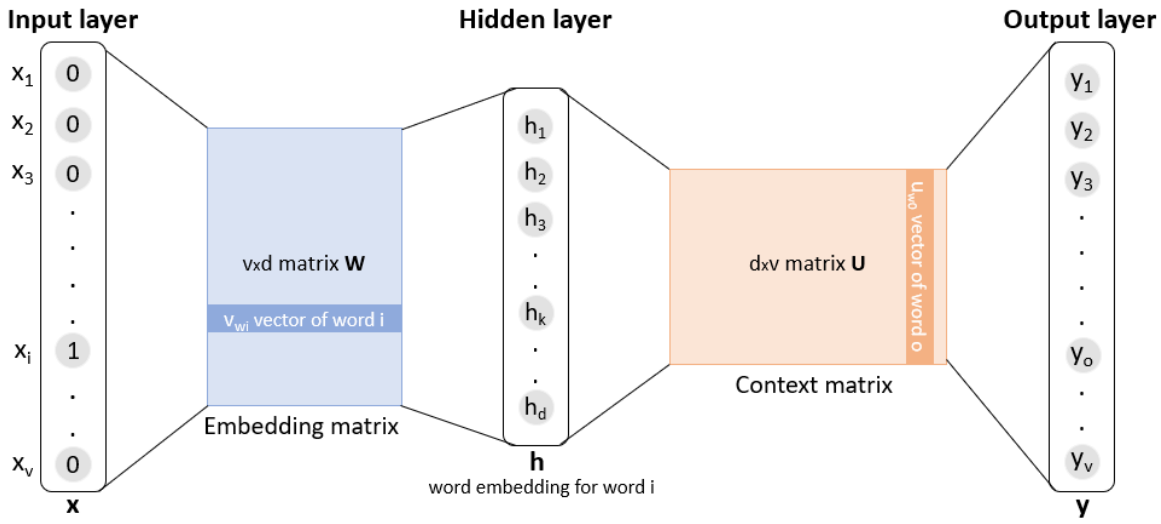


Figure 18: The Skip-gram architecture. The input $\boldsymbol{x}$ is a one-hot encoded vector and the hidden layer $\boldsymbol{h}$ is the d-dimensional word embedding $\boldsymbol{v}_{w_i}$ for input word w_i. Each value in output $\boldsymbol{y}$ represents the probability that a word from the vocabulary occurs in the context of the input word w_i. Adapted from Weng (2017).

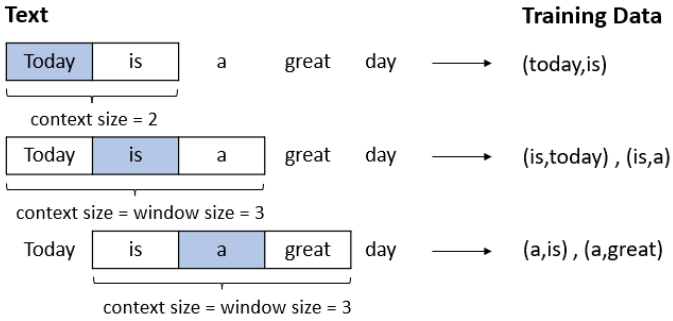


Figure 19: Example of the context size versus window size in and the generation of training data in Skip-gram.

by a $v \times d$ matrix $\boldsymbol{W}$ while the weights between the hidden layer and output layer are represented by a $d \times v$ matrix $\boldsymbol{U}$ (Rong, 2014). The matrix $\boldsymbol{W}$ holds all the word embeddings (Weng, 2017). Consider an input word w_i with index i in the vocabulary with one-hot encoded vector $\boldsymbol{x}$ where $x_i = 1$ and $x_k = 0 \; \forall \; k \neq i$. Its d-dimensional word embedding $\boldsymbol{v}_{w_i}$ is retrieved as

$$\boldsymbol{v}_{w_i} = \boldsymbol{x}^T \boldsymbol{W} = \boldsymbol{h}, \tag{12}$$

where the multiplication of the input vector $\boldsymbol{x}$ and word embedding matrix $\boldsymbol{W}$ essentially returns the i^{th} row of $\boldsymbol{W}$ (Rong, 2014). This d-dimensional embedding forms the hidden layer $\boldsymbol{h}$ in Figure 18. The output vector $\boldsymbol{y}$ is produced by multiplication of the hidden layer and the matrix $\boldsymbol{U}$, followed by a soft-max transformation to convert the raw output into values between 0 and 1. Each value in $\boldsymbol{y}$ represents the predicted probability that a specific word from vocabulary V occurs in

the context of the input word w_i. Note that all words w_j in the vocabulary are linked to two vector representations; $\boldsymbol{v}_{w_j}$ and $\boldsymbol{u}_{w_j}$, which are the j^{th} row of $v \times d$-dimensional $\boldsymbol{W}$ and the j^{th} column from $d \times v$-dimensional $\boldsymbol{U}$, respectively (Rong, 2014). The d-dimensional vector $\boldsymbol{v}_{w_j}$ represents the word embedding for word w_j in the vocabulary. Based on the two different vector representations, the conditional probability that word w_o (with index o in the vocabulary) is the context word given the input word w_i is given by

$$p(w_o|w_i) = y_o = \frac{\exp\left(\boldsymbol{u}_{w_o}^T \cdot \boldsymbol{v}_{w_i}\right)}{\sum_{j=1}^{v} \exp\left(\boldsymbol{u}_{w_j}^T \cdot \boldsymbol{v}_{w_i}\right)}, \tag{13}$$

where y_o is the output of the o^{th} node of the output layer from Figure 18, which represents the conditional probability that word w_o from the vocabulary is in the context of word w_i (Rong, 2014). For simplicity, we label the score $z_{i,j} = \boldsymbol{u}_{w_j}^T \cdot \boldsymbol{v}_{w_i}$ where w_i is the input word and w_j a word from the vocabulary. The score $z_{i,j}$, the dot product between two vector representations, is a similarity measure between words w_i and w_j (Mikolov, Chen, et al., 2013).

Skip-gram - Training and Objective Function

Before learning the word embeddings and updating the weight matrices $\boldsymbol{W}$ and $\boldsymbol{U}$ with SGD and backpropagation, Word2Vec's network starts with a random initial state. To update $\boldsymbol{W}$ and $\boldsymbol{U}$ and to learn the word embeddings $\boldsymbol{v}_{w_j}$, the model goes through the training data for a certain number of epochs. In one epoch, Word2Vec iterates over the training samples of all input words in the text and updates the weights in each iteration. The training data for an input word consists of all input-context pairwise combinations of that word and all other words in its context window. The target word w_c in the input-context pair is the actual context word of the input word w_i and is the word we want the model to predict (Rong, 2014). An example of how the pairwise training data is generated is given in Figure 19, where the window size is equal to 3. The words highlighted in blue are the input words. The first input word w_i is "Today" and its training data consists of the pairwise combination $(today, is)$, where "is" is the target context word w_c. The training data for the second input word "is" consists of the input-context pairs $(is, today)$ and (is, a).

Consider a training data sample with input word w_i and target context word w_c. The conditional probability $p(w_c|w_i)$ represents the probability that the network predicts the actual target word w_c as the context word of w_i. The model's training objective for one training sample (w_i, w_c) is to maximize the conditional probability $p(w_c|w_i)$ and is given by

$$
\begin{aligned}
\max p(w_c|w_i) &= \max y_c \\
&= \max \log y_c \\
&= \boldsymbol{u}_{w_c}^T \cdot \boldsymbol{v}_{w_i} - \log \sum_{j=1}^{v} \exp\left(\boldsymbol{u}_{w_j}^T \cdot \boldsymbol{v}_{w_i}\right) \\
&= z_{i,c} - \log \sum_{j=1}^{v} \exp z_{i,j} \\
&:= -E,
\end{aligned} \tag{14}
$$

where v is the vocabulary size and $E = -\log p(w_c|w_i)$ is the loss function that the model aims to minimize for the input-context training sample (w_i, w_c). The loss function E can be seen as a special case of the cross-entropy loss function (Rong, 2014). This objective implies that the network tries to maximize the similarity measure $z_{i,c}$ between input word w_i and target word w_c while minimizing the similarity score between the input word and all other words in the vocabulary that are not in the context, represented by the expression $\log \sum_{j=1}^{v} \exp z_{i,j}$ in equation (14).

The full training objective of Skip-gram is given

$$
-\sum_{w_i \in S} \sum_{w_c \in C(w_i)} \log p(w_c|w_i), \tag{15}
$$

where S is the text containing all words $w_i = 1, ..., |S|$ and the set $C(w_i)$ represents the context of current input word w_i. Besides, $p(w_c|w_i)$ is defined as in equation (13). Note that the combined summations in (15) basically are a summation over all input-context pairs generated from the text. Skip-gram's goal is to minimize this objective function. Skip-gram iterates through all input words w_i with corresponding training samples (w_i, w_c) from the text to update the weights in $\boldsymbol{W}$ and $\boldsymbol{U}$ by applying stochastic gradient descent (SGD) and backpropagation to the model in each iteration (Mikolov, Chen, et al., 2013; Rong, 2014). The network computes the conditional probability predictions and the corresponding loss. Next, the gradient of the loss function with respect to the model's parameters is computed by repeated application of the chain rule. All weights are slightly tuned in each iteration by moving into the direction of steepest descent, i.e., the negative of the computed gradient, with a step size that is equal to the learning rate. See the work of Rong (2014) for the derivation of the update equations for the weights in the network.

Negative Sampling

For a text with large vocabulary V, the number of weights in $\boldsymbol{W}$ and $\boldsymbol{U}$ is huge. Tuning all these parameters will take a long time as the denominator $\sum_{j=1}^{v} \exp\left(\boldsymbol{u}_{w_j}^T \cdot \boldsymbol{v}_{w_i}\right)$ in equation (13) needs to be computed in each iteration. Computing this denominator for each position in the text is computationally expensive as the number of words v in the vocabulary can be large (Rong, 2014). To make the training process more efficient, two approaches have been suggested; hierarchical soft-max and negative sampling (Mikolov, Sutskever, et al., 2013). The negative sampling approach is used in Node2Vec, so we elaborate on this method. Negative sampling modifies the optimization objective such that only a small percentage of the model's weights are updated in each iteration instead of them all, thus gaining computational efficiency. See the research of Rong (2014) for an in-depth explanation of the hierarchical soft-max approach.

In training sample (w_i, w_c), we refer to the actual context word w_c as the "positive" word, while all other words from the vocabulary that are not the target context word are called "negative" words. In the original Word2Vec implementation, the weights for all nodes/words (the positive and all negative words) in the vocabulary V are updated with iteration. With the negative sampling method, a smaller number of negative words is sampled and the weights for the positive word and only the sampled negative words are updated. Some of the output nodes are thus sampled and only the weights for these nodes are updated, which limits the number of comparisons that the networks needs to make. The subset of negative samples is chosen using the unigram distribution, where the probability of selecting a word is related to its frequency of occurrence in the text. Words that appear more frequently in the text have a bigger chance of being chosen. The unigram distribution raised to the power of 0.75 is chosen for best quality results (Mikolov, Sutskever, et al., 2013). The chance that word w_k is sampled with the modified unigram distribution is given by

$$P(w_k) = \left(\frac{f(w_k)}{\sum_{j=1}^{v} f(w_j)}\right)^{0.75}, \tag{16}$$

where $f(w_j)$ is the frequency of occurrence of word w_j in the text and v the vocabulary size, i.e., the total number of distinct words in the text. This makes $\sum_{j=1}^{v} f(w_j)$ the total number of words in the text and it can be seen as a normalizing constant.

By applying negative sampling, Word2vec basically employs a simplified objective function per training sample (Mikolov, Sutskever, et al., 2013). After sampling, the simplified loss function for

training sample (w_i, w_c) changes to

$$E = -\log \sigma(\boldsymbol{u}_{w_c}^T \cdot \boldsymbol{v}_{w_i}) - \sum_{k=1}^{K} \log \sigma\left(-\boldsymbol{u}_{w_k}^T \cdot \boldsymbol{v}_{w_i}\right), \tag{17}$$

where the set $\{w_k | k = 1, ..., K\}$ contains the K negative words sampled with the unigram distribution in equation (16) and $\boldsymbol{v}_{w_j}$ and $\boldsymbol{u}_{w_j}$ are the two vector representations of word w_j (Mikolov, Sutskever, et al., 2013). The loss function in (17) is used to replace every $-\log p(w_c|w_i)$ for each training sample (w_i, w_c) in equations (14) and (15) (Mikolov, Sutskever, et al., 2013). Besides, $\sigma(x) = \frac{1}{1+\exp(-x)}$ is the sigmoid function and Goldberg and Levy (2014) provide a theoretical analysis on why this modified objective function is used. The modified loss for each position in the text is computed and the weights of the sampled nodes/words in the network are now updated via backpropagation and SGD in each iteration. In the following section we elaborate on how Node2Vec employs the Skip-gram architecture to generate node embeddings.

4.2.2 Node2Vec

Node2Vec is a method that learns vector representations for nodes in a graph, while preserving the graph neighborhoods (i.e., hierarchical structure) of the nodes (Grover & Leskovec, 2016). This enables the generation of hierarchy-aware embeddings. Node2Vec is based on the Skip-gram method explained in Section 4.2.1, which learns vector representations for words in a text by optimizing a loss function using SGD and negative sampling (Mikolov, Chen, et al., 2013; Mikolov, Sutskever, et al., 2013). To utilize Skip-gram's architecture, we need to present the graph as a text. Node2Vec utilizes random walks to generate directed sequences of the nodes in a graph. These sequences represent sentences from a text, and the nodes in the sequence represent words in a sentence. The sampled sequences are used as input to train Word2Vec's Skip-gram neural network. Instead of predicting the context target word given an input word, Node2Vec predicts a neighboring node from the graph given a source node. Node2Vec thus starts by extracting a set of random walks from the input graph, which are then are used as input for the Skip-gram network and the network produces an embedding for each node in the graph. We first elaborate on how these random walks are sampled.

2^{nd} Order (Biased) Random walks

Node2Vec employs 2^{nd} order biased random walks to generate node sequences from the graph as

this is a flexible sampling strategy. We elaborate on this flexibility later in this section. Given a source node, Node2Vec samples a random walk of fixed length l. The node c_i represents the i^{th} node on the walk, where c_0 is the source node (Grover & Leskovec, 2016). The nodes c_i following node c_0 in the sequence are sampled from the following distribution

$$P(c_i = x | c_{i-1} = v) = \begin{cases} \frac{\pi_{vx}}{Z} & \text{if } (v, x) \in E \\ 0 & \text{otherwise} \end{cases}, \tag{18}$$

where $P(c_i = x | c_{i-1} = v)$ represents the probability of moving from node v to node x, π_{vx} represents the unnormalized transition probability between nodes v and x and Z is a normalizing constant (Grover & Leskovec, 2016). The transition probability equals 0 when there is no edge $(v, x) \in E$ between nodes v and x.

To determine π_{vx}, Grover and Leskovec (2016) define a 2^{nd} order random walk with two parameters p and q, which can "bias" the random walk towards different graph sampling strategies. The "return" parameter p controls the likelihood of immediately sampling/returning to the previous node in the walk, while q is the "in-out" parameter that determines whether the walk is restricted to a local neighborhood or explores previously unseen parts of the graph. Consider a random walk that transitioned from node t to node v and now decides on the next step by evaluating the transition probabilities π_{vx} for all nodes x that are adjacent to node v. The unnormalized transition probability is given by $\pi_{vx} = \alpha_{pq}(t, x) \cdot w_{vx}$, where w_{vx} represents the weight of edge (v, x). In case of unweighted graphs all edge weights are equal to 1, such that $w_{vx} = 1 \; \forall \; (v, x) \in E$. Grover and Leskovec (2016) set $\alpha_{pq}(t, x)$ to

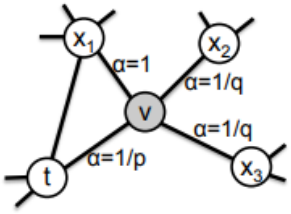


Figure 20: The random walk just traversed from node t to v and evaluates its next step. The edge labels denote the search biases α. Image from Grover and Leskovec (2016).

$$\alpha_{pq}(t, x) = \begin{cases} \frac{1}{p} & \text{if } d_{tx} = 0 \\ 1 & \text{if } d_{tx} = 1 \\ \frac{1}{q} & \text{if } d_{tx} = 2 \end{cases}, \tag{19}$$

where d_{tx} denotes shortest path distance between nodes t and x. Note that node t precedes node v and x denotes the next node in the sequence. Figure 20 illustrates the random walk procedure in Node2Vec. In Figure 20, the shortest path distance d_{tx_1} between nodes t and x_1 equals 1, while

the path distance between d_{tx_2} between nodes t and x_2 equals 2. The biases on the edges (the unnormalized transition probabilities) in Figure 20 are set according to equation (19). The 2^{nd} order biased random walks can be generated using these transition probabilities. We are working with unweighted graphs in this research, so we use $\pi_{vx} = \alpha_{pq}(t, x)$. The transition probability from equation (18) of moving from node v to x is then given by

$$P(c_i = x | c_{i-1} = v) = \begin{cases} \frac{\alpha_{pq}(t,x)}{Z} & \text{if } (v, x) \in E \\ 0 & \text{otherwise} \end{cases}, \tag{20}$$

where Z is a normalizing constant and $\alpha_{pq}(t, x)$ is given by equation (19). Next, we elaborate on different random walk sampling strategies, the role of the parameters p and q in the 2^{nd} order random walk and how the node embeddings are generated by Node2Vec.

Sampling Strategies

The main contribution of Grover and Leskovec (2016) is the introduction of the parameters p and q for flexibility in the sampling strategy. Grover and Leskovec (2016) define a flexible notion of a node's graph neighborhood and design a 2^{nd} order biased random walk procedure such that different types of neighborhoods can be explored. They argue that this added flexible notion of a neighborhood is essential to learning rich node representations and demonstrate the efficacy of Node2Vec over existing node embedding techniques.

Nodes in a graph can be organized based on *homophily*, the communities they belong to. The organization can also be based on *structural equivalence*, the structural roles of the nodes in the graph. For example, the nodes u and s_3 in Figure 21 belong to the same community, while the nodes u and s_6 do not belong to the same community but have a similar structural role in the graph (Grover & Leskovec, 2016). As the similarity of nodes depends on the notion of neighborhood (homophily and/or structural equivalence), it is essential to allow for a flexible sampling strategy. Grover and Leskovec (2016) introduce this flexibility with the parameters p and q, where p is the return parameter and q is the in-out parameter. Node2Vec is able to learn representations such that nodes that are in the same community yield similar embeddings, or that nodes with similar structural roles yield similar embeddings. By

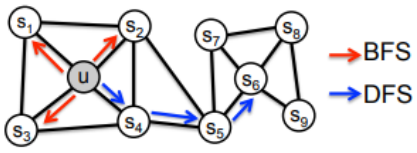


Figure 21: Illustration of homophily versus structural equivalence, and the BFS and DFS sampling strategies starting from node u. Image from (Grover & Leskovec, 2016).

setting the parameters p and q in the 2^{nd} order biased random walk, the walk can thus be biased towards different graph exploration/sampling strategies (Grover & Leskovec, 2016).

Parameter p controls the probability of immediately returning to the previous node in the walk. Setting p to a high value ($p > \max(q, 1)$) results in a low value for $\frac{1}{p}$, thus ensuring that the walk is less inclined to move towards the already visited node t in Figure 20, for example. A low value of p keeps the walk close to the previous node and thus local. Parameter q allows the search to differentiate between "inward" and "outward" nodes. If $q > 1$, the walk is biased towards nodes close to the previous node t and a local sampling strategy is ensured. Figure 20 also illustrates this; if $q > 1$, the transition probability from node v to x_2 will be $\pi_{vx_2} = \frac{1}{q} < 1$, while the probability from node v to x_1 equals $\pi_{vx_1} = 1$. As $\pi_{vx_1} > \pi_{vx_2}$, the walk is more likely to stay closer to previous node t, and a local sampling strategy is ensured. If $q < 1$, the walk is more likely to sample nodes that are further away from node t and a more outward exploration is ensured.

The parameters p and q thus ensure a certain sampling strategy. Grover and Leskovec (2016) define two extreme strategies for generating the neighborhood set $\mathcal{N}_u$ for node u. The Breadth-first Sampling (BFS) strategy restricts the neighborhood $\mathcal{N}_u$ to nodes that are direct neighbors of the source node u (Grover & Leskovec, 2016). With Depth-first Sampling (DFS) the neighborhood $\mathcal{N}_u$ of node u contains nodes that are sequentially sampled at increasing distances from the source node (Grover & Leskovec, 2016). For example in Figure 21, the neighborhood of node u contains the nodes s_1, s_2 and s_3 with BFS while the neighborhood contains s_4, s_5 and s_6 with DFS.

To ensure structural equivalence in the sampling strategy, the local neighborhoods of each node need to be characterized accurately. The neigborhoods sampled by BFS thus lead to embeddings that correspond to structural equivalence such that nodes with similar structural roles yield similar node embeddings (Grover & Leskovec, 2016). With DFS, larger parts of the graph are explored as the walks move further away from the source node u. The neighborhoods sampled by DFS lead to embeddings that correspond to homophily such that nodes that are in the same community will be embedded close in the vector space (Grover & Leskovec, 2016). To illustrate how BFS and DFS ensure structural equivalence and homiphily, respectively, consider a genealogy. To determine if two "nodes" or people in the genealogy have the same structural role, i.e., being parents of multiple children, we need to determine if this node has multiple child nodes. In Figure 21 we need to sample from the local neighborhood to determine structural equivalence (the number of child nodes in this case) and this is ensured by BFS. Now, if we want to ascertain if two people are

from the same family/community in the genealogy, we need to ensure that they follow the same bloodline, i.e., if they have the same parents, grandparents etc. This is illustrated in Figure 21 by DFS, where we basically sample a lineage and have a sampling strategy based on homophily. The Node2Vec algorithm can interpolate between BFS and DFS by setting the parameters p and q. Using either BFS, DFS or a combination of the two can result in different node embeddings and p and q thus need to be tuned carefully. Section 4.2.3 describes which values for p and q we use in this research. We next elaborate on how Node2Vec generates hierarchy-aware node embeddings with the extracted random walks, which are sampled based on a certain sampling strategy.

Learning Node Embeddings with Node2Vec

Node2Vec generates hierarchy-aware node embeddings as follows. A set of 2^{nd} order random walks is first extracted from the input graph. For each node in the graph, r random walks of fixed length l are sampled based on the chosen sampling strategy. At every step in the walk, sampling is done based on the transition probabilities π_{vx} (Grover & Leskovec, 2016). The walks can be represented as directed sequences of words or sentences, where each node represents a word. These random walks are then treated as sentences in a text. Figure 22 shows how multiple random walks are generated from nodes in a graph. For generating the training data, the source node u is equivalent to the input word w_i, and the sampled node neighborhood $\mathcal{N}_u$ is equivalent to the context $C(w_i)$. However, the generation of the training data slightly differs between Node2Vec and Word2vec. The difference is that the word w_i is always in the center of the context $C(w_i)$, while the source node u is always at the beginning of neighborhood $\mathcal{N}_u$ (shown in Figure 22). Next, source-neighborhood pairs are generated, consisting of all combinations of the source node and other words in its neighborhood. The full set of source-neighborhood pairs denotes the training data. Using the training data, Skip-gram then learns embeddings for each node with backpropagation and negative sampling, as explained in Section 4.2.1. The next section elaborates on the implementation of Node2Vec in HAC-GAN and how the hierarchy-aware embeddings are used in the embedding layer.

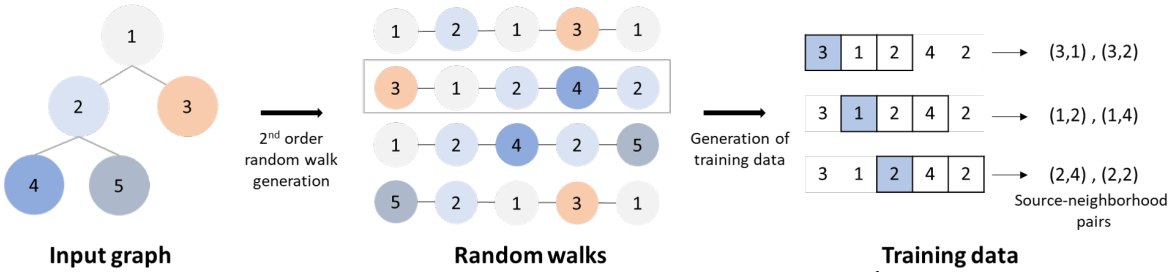


Figure 22: Illustration of how training data for Skip-gram is generated using 2^{nd} order biased random walks.

4.2.3 Embedding Layer in HAC-GAN

In this research, we aim to generate images such that the synthetic images of two classes that share a common parent node in the graph are more similar to each other than to the generated images belonging to a class with a different parent node. We assume that classes that share a common parent node also share pathological features. In our hierarchy of classes from Figure 2, we thus want the synthetic CXR images from the COVID-19 pneumonia and non-COVID-19 pneumonia classes to be more similar to each other than to the images belonging to the healthy class. The classes belonging to the same community should yield similar embeddings such that the classes in the "Pneumonia" community are embedded close to each other, while the "Healthy" community/class should yield a dissimilar embedding. The organization of the classes in our graph is thus based on homophily. The neighborhoods sampled by DFS lead to embeddings that correspond to this sampling strategy, and we chose our parameters accordingly.

So with Node2Vec, we learn an embedding for each class while capturing the hierarchical relationship between those classes based on homophily. These learned embeddings are used in the embedding layer in HAC-GAN. For each class on the leaf nodes ((non-)COVID-19 pneumonia and normal), its embedding is fed to the GAN's generator, which then synthesizes hierarchy-aware images. To generate the hierarchy-aware embeddings within the embedding layer, we employ the `node2vec` implementation in the `torch-geometric` library in `PyTorch` (Fey & Lenssen, 2019). The used parameter values are in line with the default settings in the research of Grover and Leskovec (2016). We set the number of random walks per node $r = 10$, the length of the walk $l = 80$, context size $c = 10$, and the number of negative samples $k = 10$. Grover and Leskovec (2016) use embedding dimension $d = 128$, but we set $d = 100$ as this is also the dimension of the latent vector that we multiply with the embedding vector. For p and q, we use the parameter settings that are used in Grover and Leskovec (2016) to ensure a DFS sampling strategy, and we set the parameter values to $p = 1$ and $q = 0.5$. We first train Node2Vec separately for one epoch for an optimization of the weights as done in Grover and Leskovec (2016), with the SparseAdam optimizer with $lr = 0.01$ (used in the `node2vec` example implementation by Fey and Lenssen (2019)). Next, this pre-trained Node2Vec network is used as embedding layer in HAC-GAN, and the weights of the Node2Vec embedding layer are trained jointly with the other layers in HAC-GAN to synthesize fake images.

4.3 Training Procedure (H)AC-GAN

The customized AC-GAN and HAC-GAN are trained to synthesize CXR images for the COVID-19 pneumonia, non-COVID-19 pneumonia and healthy classes. To train the models, the images in the original dataset are first pre-processed. The image pre-processing steps include resizing the image to $128 \times 128 \times 3$ and normalizing its pixels from $[0, 255]$ to $[-1, 1]$. Pixel normalization is done to ensure that the pixels are scaled to the range of the TanH activation function.

After pre-processing, the discriminator D and generator G are trained, i.e. the weights and biases in the NNs are tuned. All parameters are first randomly initialized from a normal distribution with $\mu = 0$ and $\sigma = 0.02$ following Radford et al. (2015). Next, the optimization of the parameters is done through the minimization of loss functions using backpropagation (BP). The output of the network is first computed for a certain number of training samples (with batchsize bs) in the forward pass and the corresponding losses are computed. The loss functions of the discriminator D and generator G in (H)AC-GAN are given by equations (7) and (8), respectively. D is trained by minimizing $L^{(D)}$ and G is trained by minimizing $L^{(G)}$. The losses $L^{(D)}$ and $L^{(G)}$ consist of two parts; the log-likelihood of the correct source and the log-likelihood of the correct class (Odena et al., 2017). The source loss of D is given by $-(\mathbb{E}_{x\sim p_r} \log D(\boldsymbol{x}) + \mathbb{E}_{z\sim p_z} \log[1 - D(G(\boldsymbol{z}|c))])$ and the class loss by $-(\mathbb{E}_{x\sim p_r} \log Q(\boldsymbol{x}) + \mathbb{E}_{z\sim p_z} \log Q(G(\boldsymbol{z}|c)))$ in equation (7) . In G, $-\mathbb{E}_{z\sim p_z} \log D(G(\boldsymbol{z}|c))$ is the source loss while $-\mathbb{E}_{z\sim p_z} \log Q(G(\boldsymbol{z}|c))$ represents the class loss. We use the Binary Cross Entropy loss (BCELoss) function and Negative Log Likelihood loss (NLLLoss) function to compute the source loss and class loss, respectively (Odena et al., 2017).

Next, the gradients of the loss functions with respect to all the weights in the discriminator D or generator G are calculated by repetitively applying the chain rule during the backward pass. Each gradient (of both $L^{(D)}$ and $L^{(G)}$) is fed to an optimizer which uses it to update the weights in all previous layers in the direction of steepest descent. As done in the DCGAN architecture, we employ two separate Adam optimizers with hyperparameters $\beta_1 = 0.5$, $\beta_2 = 0.999$ and a learning rate of $lr = 0.0002$ for updating the weights in D and G (Radford et al., 2015). The Adam optimizer is an optimization method that computes individual adaptive learning rates for different parameters and includes momentum by taking the running average gradient with respect to previous gradients. By including adaptive learning rates the optimization process performs updates depending on the importance of each parameter, and by including momentum the optimization process is sped up (Kingma & Ba, 2015). See Kingma and Ba (2015) for further details about this optimizer.

The full training procedure for (H)AC-GAN is given in Algorithm 1. We set the number of epochs to $e = 1000$ and the batchsize to $bs = 64$, such that the number of iterations per epoch equals $k = \lceil \frac{n}{bs} \rceil$, where n is the number of images in the training dataset. Note that the weights of the networks are slightly updated with each input batch (in each iteration). We start by updating the weights of the discriminator D. Goodfellow (2016) and Salimans et al. (2016) recommend to calculate $L^{(D)}$ and its gradient in two steps; once for a batch of real samples and once for a batch of fake samples. First, 64 real images are sampled from the training dataset and forwarded through the discriminator to compute $D(\boldsymbol{x})$ and $Q(\boldsymbol{x})$. The loss $L^{(D)}_{\text{real}}$ for the current batch is computed using the BCELoss and NLLLoss and then its gradient is calculated during a backward pass. Secondly, 64 fake images are generated by G and forwarded through D after which the loss $L^{(D)}_{\text{fake}}$ of the current batch and its gradient are computed. The gradients of the real and fake batches are accumulated and the parameters of D are updated by the Adam optimizer. After updating the parameters of D, the parameters in G are updated. The generator synthesizes 64 samples $G(\boldsymbol{z}_i|c)$ for $i = 1, ..., 64$ and the loss $L^{(G)}$ is computed. The parameters of G are updated by descending its stochastic gradient with Adam. This is repeated for all k iterations in all e epochs.

To train (H)AC-GAN, we use the `Adam`, `BCELoss` and `NLLLoss` modules from `torch` in `PyTorch` (Collobert et al., 2011). AC-GAN and HAC-GAN each take around 25 hours to train for 1000 epochs. After training, we create an enlarged and balanced dataset by generating fake images from the trained generator G for the COVID-19 pneumonia, non-COVID-19 pneumonia and normal classes. We augment the dataset such that each class contains 10.000 images.

Algorithm 1 Training Procedure (H)AC-GAN

Set the number of epochs e, the number of iterations per epoch k, the batchsize m, and $\beta_1 = 0.5$, $\beta_2 = 0.999$ and $lr = 0.0002$ in the Adam optimizer.

Initialize $\theta_0^{(D)}$ (the parameters of discriminator D) and $\theta_0^{(G)}$ (the parameters of generator G)

for i = *1,...,e* **do**

  for j = *1,...,k* **do**

    Update the discriminator D

    1: Sample minibatch of m real images $\{\boldsymbol{x}_1, ..., \boldsymbol{x}_m\}$ from the original dataset

    2: Compute the loss of D for the real samples and compute its gradient $\nabla_{\theta_{i-1}^{(D)}} L_{\text{real}}^{(D)}$ where

$$L_{\text{real}}^{(D)} = -\frac{1}{m} \sum_{i=1}^{m} [\log D(\boldsymbol{x}_i) + \log Q(\boldsymbol{x}_i)]$$

    3: Sample minibatch of m noise vectors $\{\boldsymbol{z}_1, ..., \boldsymbol{z}_m\}$ from the noise distribution p_z

    4: Generate m fake images $G(\boldsymbol{z}_i|c)$ with the generator for $i = 1, ..., m$

    5: Compute the loss of D for the fake samples and compute its gradient $\nabla_{\theta_{i-1}^{(D)}} L_{\text{fake}}^{(D)}$ where

$$L_{\text{fake}}^{(D)} = -\frac{1}{m} \sum_{i=1}^{m} [\log[1 - D(G(\boldsymbol{z}_i|c))] + \log Q(G(\boldsymbol{z}_i|c))]$$

    6: Update the parameters of D by descending its accumulated gradient with the Adam optimizer:

$$\theta_i^{(D)} = \text{Adam}\left(\nabla_{\theta_{i-1}^{(D)}} L_{\text{real}}^{(D)} + \nabla_{\theta_{i-1}^{(D)}} L_{\text{fake}}^{(D)}, \theta_{i-1}^{(D)}, \beta_1, \beta_2, lr\right)$$

    Update the generator G

    7: Sample minibatch of m noise vectors $\{\boldsymbol{z}_1, ..., \boldsymbol{z}_m\}$ from the noise distribution p_z

    8: Generate m fake images $G(\boldsymbol{z}_i|c)$ with the generator for $i = 1, ..., m$

    9: Compute the loss for generator G with

$$L^{(G)} = -\frac{1}{m} \sum_{i=1}^{m} [\log D(G(\boldsymbol{z}_i|c)) + \log Q(G(\boldsymbol{z}_i|c))]$$

    10: Update the parameters of G by descending its stochastic gradient with the Adam optimizer:

$$\theta_i^{(G)} = \text{Adam}\left(\nabla_{\theta_{i-1}^{(G)}} L^{(G)}, \theta_{i-1}^{(G)}, \beta_1, \beta_2, lr\right)$$

  end for

end for

4.4 Deep Transfer Learning (DTL)

For classifying the COVID-19 pneumonia, non-COVID-19 pneumonia and normal test CXR images, we train several convolutional neural networks (CNNs) for image classification using deep transfer learning (DTL). We employ transfer learning as it decreases computation time and the complexity of the training process (Zhuang et al., 2021). With DTL, a pre-trained CNN (trained on a very large dataset like ImageNet) is used as a fixed feature extractor or as initialization where the whole CNN is fine-tuned. With feature extraction, the parameters of the pre-trained model are used to extract features from the input images and a simple classifier is trained on top of it. First, either the last fully-connected layer or the last convolutional layer of the pre-trained CNN, which acts as a classifier for the 1000 classes in ImageNet, is removed. The remaining part of the model, for which all weights are freezed, now acts as a feature extractor and a classifier for the new dataset is trained on top of it to fit the task at hand. With fine-tuning the last fully connected/convolutional layer is also replaced by a new classifier. However, the weights of the pre-trained network are not freezed but rather used as initialization and are later fine-tuned by continuing the backpropagation. The range of classes in ImageNet is much broader than the three classes present in the CXR dataset. We therefore use the weights of the pre-trained CNN, which are tuned to ImageNet, as initial values and employ fine-tuning to tune all the weights in the model for detecting COVID-19. We employ three popular deep transfer learning models: SqueezeNet, ResNet-18, and AlexNet. We use these specific models as they achieved state-of-the-art results in several tasks during recent years and are computationally less expensive compared to other popular CNNs like Xception, InceptionResNet and DenseNet (Loey et al., 2020). We provide a brief description of each model in the following sections and refer to the reader to Iandola et al. (2016), He et al. (2016) and (Krizhevsky et al., 2012) for an in depth explanation of the architectures and introduced modules. A summary of the modified CNNs is given in Appendix D, based on the implementation of the models in the `torchvision` library `PyTorch` (Paszke et al., 2019).

SqueezeNet is a small CNN that achieves AlexNet-like accuracy but with 50 times fewer parameters (Iandola et al., 2016). SqueezeNet was designed as a small neural network with few parameters that can more easily fit into computer memory. The first layer in SqueezeNet is a convolutional layer, which is followed by eight fire modules. SqueezeNet ends with a classifier that contains the final convolutional layer that outputs the class probability. It also performs max-pooling between some of the layers. The network has an input image size of $227 \times 227 \times 3$. To fine-tune and modify

the network for the task at hand, we replace the last convolutional layer with a convolutional layer that is modelled to classify COVID-19, non-COVID-19 pneumonia and normal CXRs.

ResNet-18 is one of the most popular CNN architectures. It was the winner of the 2015 ImageNet competition and it provides easier gradient flow for more efficient training (He et al., 2016). In order to solve the vanishing gradient problem, He et al. (2016) introduced Residual Networks which use a technique called skip connections. The skip connection skips training for a few layers and connects its output directly to the output of the connected layer. The advantage of adding this type of connection is that it provides a direct path to earlier layers in the network, making the gradient updates for those layers easier (Minaee et al., 2020). There are multiple versions of the ResNet model, their main difference being the number of layers present in the network. We employ ResNet-18 as it contains less layers and is less computationally expensive compared to the other versions of the model. The complete network consists of 18 layers, containing 17 convolutional layers and one fully-connected layer. There is an average pooling operation between the last convolutional layer and the fully-connected layer. Throughout the network, skip connections are inserted between the layers. The network has an input image size of $224 \times 224 \times 3$. To fine-tune ResNet-18 and modify the CNN, we replace the last fully-connected layer with a linear classifier that is able to detect the three classes of interest in this research.

AlexNet is a CNN of eleven layers deep, which achieved state-of-the art performance on classifying the ImageNet dataset (Krizhevsky et al., 2012). AlexNet won the 2012 ImageNet competition. It contains five convolution layers, some of which are followed by max-pooling layers, and three fully-connected layers using ReLU functions. The input size is $256 \times 256 \times 3$. We remove the last fully-connected layer and replace it with a linear classifier that is modelled for the task at hand.

Algorithm 2 introduces the training procedure of the DTL models. The set of DTL models is given by $M = \{\text{SqueezeNet, ResNet-18, AlexNet}\}$, and each of these models is fine-tuned by training on the original and augmented dataset. The last fully connected/convolutional layer of these models is removed and replaced with a modified classifier. Before training, we first generate images with (H)AC-GAN to augment the dataset. Next, the images are resized according to the input requirements of the deep transfer learning models. The images are resized to $227 \times 227 \times 3$ pixels, $224 \times 224 \times 3$ pixels, and $256 \times 256 \times 3$ pixels for the SqueezeNet, ResNet-18, and AlexNet networks, respectively. Next, all images are normalized to be in the range $[-1, 1]$. The input dataset is divided into a training set and validation set, where 80% is used for training and 20% is used as

validation set. This split has been proven efficient in many researches.

After pre-processing, training starts. We train all models for $e = 25$ epochs. The number of iterations per epoch equals $k_{train} = \lceil \frac{n}{bs} \rceil$, where n is the number of images in the training set and the batchsize equals $bs = 64$. In each iteration, a sampled batch of images from the training dataset is forwarded through model $l \in M$, which outputs class predictions. Next, we use the cross entropy loss function to compute the loss between the model's predictions and target labels. We employ the cross entropy loss at it is often used in classification problems with multiple classes. The gradient of this loss is backpropagated through the network, and the networks' weights are updated in each iteration using the Adam optimizer with default hyperparameter settings $\beta_1 = 0.9$, $\beta_2 = 0.999$, and learning rate $lr = 0.001$. During the validation phase, a batch from the validation dataset is sampled and forwarded through the network. We again compute the loss between the network's output and target labels. However, no backpropagation or updating of the parameters takes place in this phase. Per epoch, the average validation loss over the different mini-batches is computed. If the validation loss for the current epoch is lower than the lowest validation loss up to this epoch, we save the current model as the best model. It takes approximately 2 hours and 3.5 hours to train all three CNNs on the original and augmented datasets for 25 epochs, respectively.

After the training and validation phase, the testing phase starts. For each model $l \in M$, the test dataset is forwarded through the saved best model and the images are classified. Next, the performance measures are calculated based on the network's output. We elaborate on the performance measures in Section 4.6. To implement the DTL models, we load the pre-trained models `squeezenet1_1`, `resnet18` and `alexnet` from `torchvision.models` in the `PyTorch` library (Paszke et al., 2019).

Algorithm 2 Classification using Deep Transfer Learning

For the training phase, set the number of epochs e, the number of training and validation iterations per epoch k_{train} and k_{val}, the batchsize m, and $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $lr = 0.001$ in the Adam optimizer. The used DTL models are $M = \{\text{SqueezeNet, ResNet-18, AlexNet}\}$, where the last fully connected or convolutional layer is replaced by a new classifier. Set $L_{best}^{(val)} = \infty$ in the validation phase. For the testing phase, set the number of mini-batches in the test dataset to k_{test}.

Initialize $\theta_0^{(l)}$, the pre-trained weights/parameters for all DTL models $l \in M$

Pre-processing

- Generate fake COVID-19, non-COVID-19 pneumonia and normal CXR images with (H)AC-GAN
- Augment the original dataset with the synthetic images generated by (H)AC-GAN
- Resize the images to dimensions $227 \times 227 \times 3$, $224 \times 224 \times 3$ and $256 \times 256 \times 3$
- Normalize each image from the range $[0, 255]$ to $[-1, 1]$

TRAINING
for *l = 1,...,|M|* **do**
  for *i = 1,...,e* **do**
    Training phase
    for *j = 1,...,k_{train}* **do**
      1: Sample minibatch of m images from the (augmented) training dataset
      2: Forward the input images in the minibatch through the network to compute predictions and compute the cross entropy loss $L^{(train)}$ between the model's predictions and target labels
      3: Update the weights θ^l of the DTL model l with the Adam optimizer

$$\theta_i^{(l)} = \text{Adam}\left(\nabla_{\theta_{i-1}^{(l)}} L^{(train)}, \theta_{i-1}^{(l)}, \beta_1, \beta_2, lr\right)$$

    end for

    Validation phase
    for *j = 1,...,k_{val}* **do**
      4: Sample minibatch of m images from the (augmented) validation dataset
      5: Forward the images in the minibatch through model$^l(\theta_i^{(l)})$ to compute predictions and compute the cross entropy loss $L_{i,k_{val}}^{(val)}$ between the model's predictions and target labels
    end for

    6: Compute the average validation loss $L_i^{(val)} = \frac{1}{k_{val}} \sum_{j=1}^{k_{val}} L_{i,k_{val}}^{(val)}$ for current epoch i

    if $L_i^{(val)} < L_{best}^{(val)}$ **then**
      7: Set $L_{best}^{(val)} = L_i^{(val)}$ and model$_{best}^{(l)}$= model$^{(l)}(\theta_i)$
    end if
  end for
end for

TESTING
for *l = 1,...,|M|* **do**
  for *j = 1,...,k_{test}* **do**
    9 : Classify the images in the current batch from the test set using fine-tuned network model$_{best}^{(l)}$
  end for
end for

4.5 Performance Measures GAN

The objective of (H)AC-GAN is to produce a diverse set of images that matches the real dataset. To measure the quality and diversity of the images synthesized by GANs several evaluation measures exist, some of which are reviewed in Borji (2019). We use two of the most widely used measures to evaluate the quality of the synthesized CXRs; the Multiscale Structural Similarity (MS-SSIM) score and the Fréchet Inception Distance (FID). Both measures prefer models that generate realistic and diverse samples and are shown to agree with human perceptual evaluation. We elaborate on the MS-SSIM score in Section 4.5.1 and the FID is explained in Section 4.5.2. We compute the MS-SSIM and FID measures by adapting the implementations `pytorch-msssim` by Pessoa (2021) and `clean-fid` by Parmar, Zhang, and Zhu (2022) in the `PyTorch` library (Paszke et al., 2019).

4.5.1 Multiscale Structural Similarity (MS-SSIM) Score

One of the most successful methods for quantitatively evaluating the diversity of generated images is the multiscale structural similarity (MS-SSIM) score (Z. Wang, Simoncelli, & Bovik, 2003; Odena et al., 2017). MS-SSIM scores are in the range of $[0, 1]$, where a lower value corresponds to perceptually more diverse images. MS-SSIM is the multi-scale variant of the Structural Similarity (SSIM) measure that aims to measure similarity between (sets of) images and was shown to correlate well with human perception of diversity (Odena et al., 2017). Let $\boldsymbol{x} = \{x_i | i = 1, 2, ..., N\}$ and $\boldsymbol{y} = \{y_i | i = 1, 2, ..., N\}$ be two sets of images, and let μ_x, σ_x^2 and σ_{xy} be the mean of $\boldsymbol{x}$, the variance of $\boldsymbol{x}$ and the covariance of $\boldsymbol{x}$ and $\boldsymbol{y}$, respectively. SSIM is based on the following three image features; the luminance $l(\boldsymbol{x}, \boldsymbol{y})$, contrast $c(\boldsymbol{x}, \boldsymbol{y})$ and structure $s(\boldsymbol{x}, \boldsymbol{y})$ comparison measures that are given by the following formulas

$$l(\boldsymbol{x}, \boldsymbol{y}) = \frac{2\mu_x\mu_y + C_1}{\mu_x^2 + \mu_y^2 + C_1}, \tag{21}$$

$$c(\boldsymbol{x}, \boldsymbol{y}) = \frac{2\sigma_x\sigma_y + C_2}{\sigma_x^2 + \sigma_y^2 + C_2}, \tag{22}$$

$$s(\boldsymbol{x}, \boldsymbol{y}) = \frac{\sigma_{xy} + C_3}{\sigma_x\sigma_y + C_3}, \tag{23}$$

where C_1, C_2 and C_3 are small constants (Z. Wang et al., 2003). The general form of the SSIM metric between sets $\boldsymbol{x}$ and $\boldsymbol{y}$ is given by

$$\text{SSIM}(\boldsymbol{x}, \boldsymbol{y}) = [l(\boldsymbol{x}, \boldsymbol{y})]^\alpha \cdot [c(\boldsymbol{x}, \boldsymbol{y})]^\beta \cdot [s(\boldsymbol{x}, \boldsymbol{y})]^\gamma, \tag{24}$$

where α, β and γ are parameters used to define the relative importance of the three components (Z. Wang et al., 2003). These parameters are usually set to $\alpha = \beta = \gamma = 1$.

The perception of an image depends on many aspects and is subjective to the viewing condition of the observer. To be able to capture more of those aspects and provide a better metric, the multi-scale variant of SSIM was proposed. MS-SSIM is more robust with regard to variations in viewing conditions by evaluating an image at M different image scales. This is accomplished by repeatedly performing the image analysis in $M-1$ iterations, where all images are downsampled by a factor of two in each iteration. The original image is indexed as scale 1, and the image at the $(M-1)^{\text{th}}$ iteration as scale M (Z. Wang et al., 2003). At the j^{th} scale, the contrast and structure comparisons are denoted as $c_j(\boldsymbol{x}, \boldsymbol{y})$ and $s_j(\boldsymbol{x}, \boldsymbol{y})$, respectively. The luminance $l_M(\boldsymbol{x}, \boldsymbol{y})$ score is only calculated for the M^{th} scale. The overall MS-SSIM metric can be computed as

$$\text{MSSSIM}(\boldsymbol{x}, \boldsymbol{y}) = [l_M(\boldsymbol{x}, \boldsymbol{y})]^{\alpha_M} \cdot \prod_{j=1}^{M} [c_j(\boldsymbol{x}, \boldsymbol{y})]^{\beta_j} \cdot [s_j(\boldsymbol{x}, \boldsymbol{y})]^{\gamma_j}, \tag{25}$$

where M is the number of different scales and α_M, β_j, and γ_j are again the parameters used to define the relative importance of the three components in each iteration (Z. Wang et al., 2003). We set $\alpha_M = 1$, $\beta_j = \gamma_j = 1$ for $j = 1, ..., M$ as these are the default values. We measure the MS-SSIM scores between 100 randomly sampled pairs from the total dataset and within each class, as is done in Odena et al. (2017).

4.5.2 Fréchet Inception Distance (FID)

The Fréchet Inception Distance (FID) introduced by Heusel et al. (2017) is a similarity measure between two sets of images that has also been shown to correlate well with human visual judgement. The score summarizes how similar two image datasets are by computing a distance measure between them. The FID utilizes the InceptionV3 network that is pre-trained on the ImageNet dataset. The real and generated images are first passed through this network and the activations of the last intermediate pooling layer in the network are then used as feature representations which summarize the images (Heusel et al., 2017). The assumption is made that these activations can be approximated by a multivariate normal distribution. Two Gaussian distributions are then fitted to the activations of the two datasets containing real or generated images. Next, the FID is calculated by computing the Fréchet Distance between these Gaussian distributions (for the real dataset r

and generated dataset g) with the following formula

$$FID(r,g) = ||\boldsymbol{\mu}_r - \boldsymbol{\mu}_g||_2^2 + Tr\left(\boldsymbol{\Sigma}_r + \boldsymbol{\Sigma}_g - 2(\boldsymbol{\Sigma}_r\Sigma_g)^{\frac{1}{2}}\right), \tag{26}$$

where $(\boldsymbol{\mu}_r, \boldsymbol{\Sigma}_r)$ and $(\boldsymbol{\mu}_g, \boldsymbol{\Sigma}_g)$ represent the mean vector and variance matrix of the fitted distributions for the datasets containing real and generated images, respectively (Heusel et al., 2017). In equation (26), $Tr(\cdot)$ represents the trace operation on a matrix. A lower FID means smaller distances between the distributions and thus greater similarity between the generated and real data distributions.

4.6 Performance Measures Classification Methods

Many measures to evaluate the classification performance of deep learning models exist (Tharwat, 2020). Some of the most widely used measures are accuracy, precision and recall/sensitivity (Sokolova, Japkowicz, & Szpakowicz, 2006). We employ these metrics to evaluate the classification performance of the deep transfer learning models used in this research. The formulas of the measures mentioned above are given by

$$\text{Accuracy} = (TP + TN) / (TP + FN + TN + FP), \tag{27}$$

$$\text{Precision} = TP / (TP + FP), \tag{28}$$

$$\text{Sensitivity} = TP / (TP + FN), \tag{29}$$

where TP is number of true positives, TN the number of true negatives, FN the number of false negatives, and FP the number of false positives. As we are interested in detecting COVID-19 in our research, TP refers to the correctly classified COVID-19 cases, FP refers to the normal or non-COVID-19 pneumonia cases that are classified as COVID-19, TN refers to the correctly classified normal or non-COVID-19 pneumonia cases while FN refers to COVID-19 cases that are classified as normal or non-COVID-19 pneumonia cases. To further evaluate the performance of the deep transfer learning models, we also present confusion matrices which show multi-class classification performance (Tharwat, 2020).

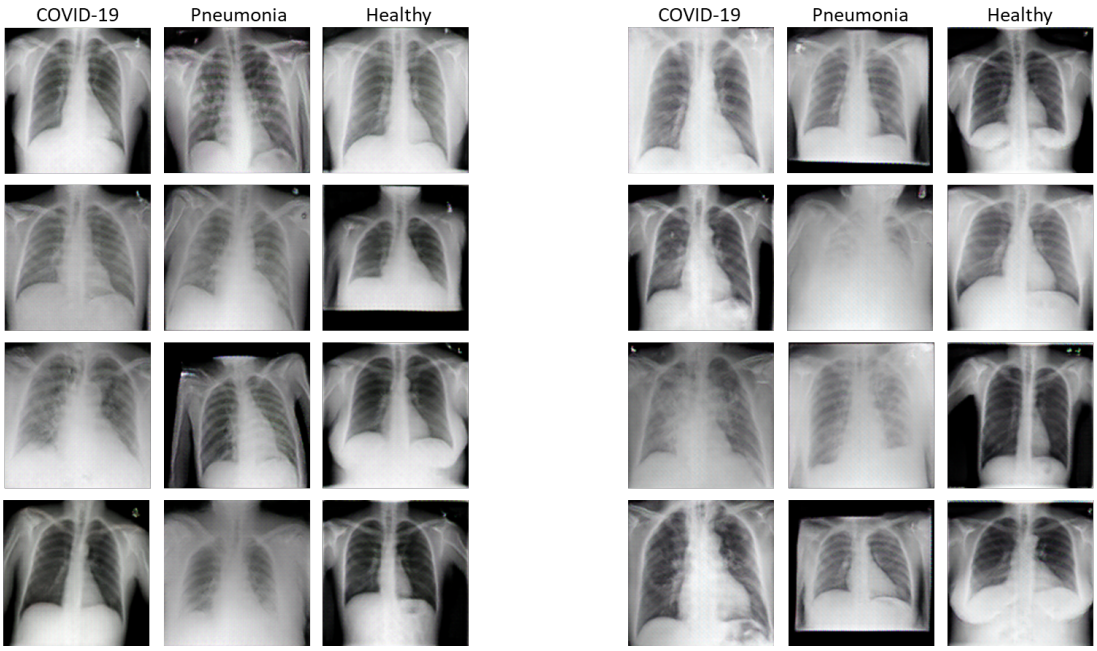
5 Results

In this section, we present the results of our methods. We analyze the effect of including class hierarchy in the image generation process of the GANs. First, the quality and diversity of the generated images by (H)AC-GAN for the CXR, Fruit, and Vegetable datasets are compared in Section 5.1. Next, we analyze the effect of augmenting the original CXR dataset with synthetic data on the detection of COVID-19 and the classification performance of the CNNs in Section 5.2.

5.1 The Quality and Diversity of Images Generated by (H)AC-GAN

After training AC-GAN and HAC-GAN, we generate images for the COVID-19 pneumonia, non-COVID-19 pneumonia and normal classes. We elaborate on the training process of AC-GAN and HAC-GAN in Appendix E, where we evaluate the losses of the discriminator D and generator G over the training iterations. We found that the losses $L^{(D)}$ of D and $L^{(G)}$ of G converged over the iterations, indicating successful and stable training. Besides, Appendix F visualizes the training progression of images generated by G in both AC-GAN and HAC-GAN over several epochs. Examples of the generated images by AC-GAN and HAC-GAN after full training are given in Figures 23a and 23b. The generated images by both methods clearly resemble the CXRs from the original dataset, are realistic, and also seem to be diverse (a clear distinction can be made between man and female chests, for example). This indicates that no mode collapse has occurred, that is when the GAN only outputs images with little variety, and training was successful.

We compute the FID and MS-SSIM scores to evaluate the generated images quantitatively. We measure the quality of the generated images with the FID score and determine if they resemble the original dataset. A lower FID score means a closer resemblance to the original dataset and is thus better. We also examine the diversity within the synthetic dataset with the MS-SSIM score, where a lower score means more diversity. The FID and MS-SSIM scores achieved by AC-GAN and HAC-GAN are presented in Table 2. On average and for all individual classes, both the FID scores and MS-SSIM scores for HAC-GAN are lower than for AC-GAN. HAC-GAN thus generates images that resemble the original dataset more closely than AC-GAN (shown by the lower FID scores), while it also produces a more diverse set of images (shown by the lower MS-SSIM scores). These results indicate that incorporating class hierarchy into the image generation process improves a GAN's performance, which is in line with the findings of R. Zhang et al. (2020).



(a) AC-GAN **(b)** HAC-GAN

Figure 23: Examples of CXRs generated by AC-GAN (a) and HAC-GAN (b).

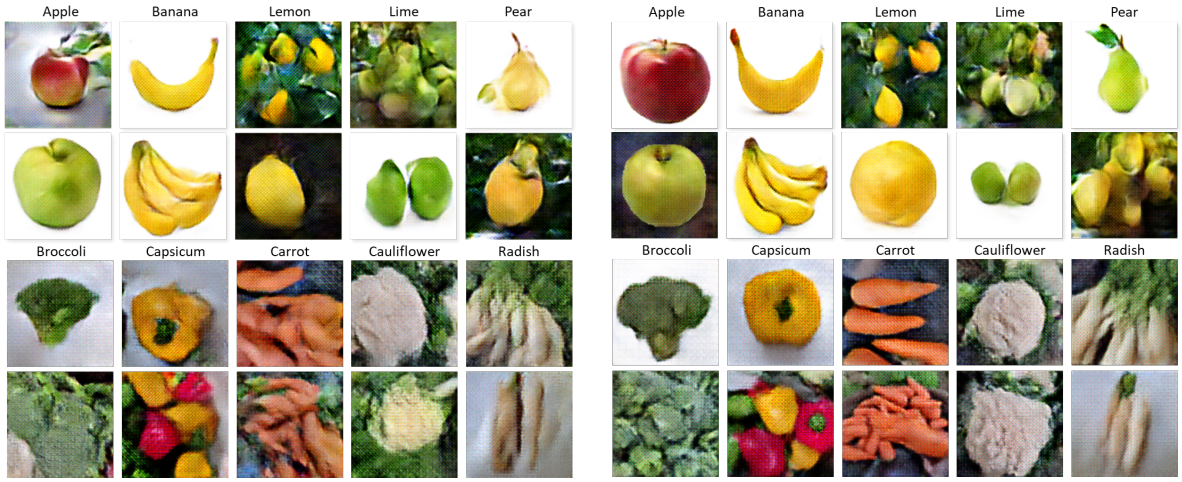
Table 2 *FID and MS-SSIM scores achieved on the fake CXR images. Best results are highlighted in bold.*

CXRs	FID		MS-SSIM	
	AC-GAN	HAC-GAN	AC-GAN	HAC-GAN
COVID-19	134.88	**123.25**	0.494	**0.486**
Pneumonia	149.73	**130.13**	0.492	**0.451**
Healthy	158.11	**137.61**	0.527	**0.518**
All	135.076	**114.50**	0.475	**0.464**

To validate if including class hierarchy leads to higher quality and more diverse synthetic samples, we also train AC-GAN and HAC-GAN on our benchmark Fruit and Vegetable datasets. Figures 24a and 24b show samples of generated images of all classes in the Fruit and Vegetable datasets, respectively. The different fruits and vegetables are clearly recognizable. These samples show that the images generated by HAC-GAN are more realistic and more distinctive than the images generated by AC-GAN. This is especially the case for the vegetable dataset, where ACGAN's synthesized carrots are not recognizable as carrots, and the capsicum samples are not as defined as those generated by HAC-GAN, for example. This further demonstrates that incorporating the hierarchy of classes in a GAN improves the quality of synthesized images.

We also compute the FID and MS-SSIM scores for the generated fruit and vegetable images, shown in Tables 3 and 4. Again, HAC-GAN achieves lower FID scores and lower MS-SSIM scores on average and for almost all individual classes. So overall, HAC-GAN achieves lower FID and

lower MS-SSIM scores and is able to generate a diverse set of high-quality, distinctive images. The results demonstrate that leveraging class hierarchy in a GAN improves the quality and diversity of the generated images. Therefore, we select HAC-GAN to create an augmented dataset to aid in detecting COVID-19 with deep transfer learning. We evaluate the effect of augmenting the original CXR dataset on the performance of the classification methods in the next section.



(a) AC-GAN **(b)** HAC-GAN

Figure 24: Examples of fruit and vegetable images generated by AC-GAN (a) and HAC-GAN (b).

Table 3 *FID and MS-SSIM scores achieved on the fake fruit images. Best results are highlighted in bold.*

Fruit	FID		MS-SSIM	
	AC-GAN	HAC-GAN	AC-GAN	HAC-GAN
Apple	196.78	**158.80**	0.224	**0.208**
Banana	**358.97**	361.01	**0.167**	0.173
Lemon	272.92	**249.03**	0.170	**0.147**
Lime	241.94	**227.69**	0.167	**0.151**
Pear	217.61	**181.46**	0.245	**0.148**
All	204.06	**192.47**	0.154	**0.138**

Table 4 *FID and MS-SSIM scores achieved on the fake vegetable images. Best results highlighted in bold.*

Vegetable	FID		MS-SSIM	
	AC-GAN	HAC-GAN	AC-GAN	HAC-GAN
Broccoli	342.36	**337.15**	0.143	**0.113**
Capsicum	379.31	**351.35**	0.125	**0.117**
Carrot	379.97	**357.41**	0.155	**0.122**
Cauliflower	345.50	**400.53**	0.158	**0.133**
Radish	331.63	**324.00**	0.155	**0.117**
All	294.11	**286.29**	0.124	**0.090**

5.2 Classification Performance

We use the following three metrics to evaluate the performance of the DTL models SqueezeNet, ResNet-18, and AlexNet on detecting COVID-19; accuracy (Acc), precision (Pr), and sensitivity (Sn), as defined in Section 4.6. Table 5 shows the performance measures for the three classification methods, trained on both the original and augmented datasets and tested on the modified test set (see Section 2). The best results per model are highlighted in bold. The DTL models SqueezeNet, ResNet-18, and AlexNet achieve an accuracy of 77.9%, 82.4% and 65.4% on the test set when trained on the original dataset, respectively. When trained on the augmented dataset that contains images generated by HAC-GAN, the models achieve an accuracy of 86.1%, 84.3%, and 79.6%, respectively. After augmentation, SqueezeNet performs best based on the accuracy. Table 5 also shows an increase in precision and sensitivity for all DTL models when trained on the augmented dataset. The increase in sensitivity is essential since we aim to limit the number of missed COVID-19 samples as much as possible. The increase in precision for all methods, although only slightly for ResNet-18, means that fewer false positive COVID-19 predictions were made, which could result in a decrease of the burden on the health care system, as no additional screening with PCR testing and quarantine measures are necessary. Augmenting the dataset thus improves the COVID-19 detection performance of all three pre-trained CNNs and indicates that the generated images contain features that enhance classification performance.

Table 5 *Performance measures for detecting COVID-19 with the classification methods trained on the original and augmented datasets (with HAC-GAN). Best results per model are highlighted in bold.*

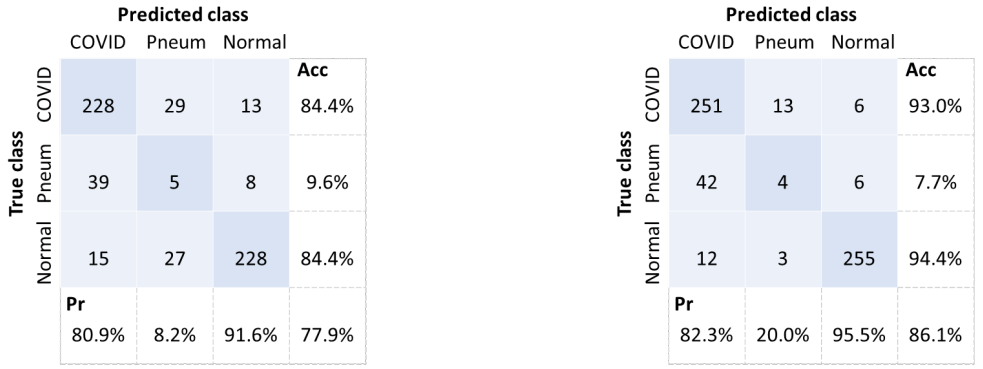
Dataset	**SqueezeNet**			**ResNet-18**			**AlexNet**		
	Acc (%)	Pr (%)	Sn (%)	Acc (%)	Pr (%)	Sn (%)	Acc (%)	Pr (%)	Sn (%)
Original	77.9	80.9	84.4	82.4	82.5	87.4	65.4	63.5	70.4
Augmented	**86.1**	**82.3**	**93.0**	**84.3**	**82.9**	**89.6**	**79.6**	**77.1**	**93.3**

Next, we analyze the accuracy/sensitivity and precision per class. Figure 25 shows the confusion matrices of SqueezeNet when trained on the original and augmented datasets, and exhibits the accuracy/sensitivity (Acc) and precision (Pr) for the COVID-19 pneumonia, non-COVID-19 pneumonia, and normal classes. We evaluate the confusion matrices and per-class results for SqueezeNet, as this CNN achieves the highest overall accuracy (see Table 5). The results for ResNet-18 and AlexNet can be found in Appendix G. Figure 25 demonstrates that SqueezeNet's classification performance for the COVID-19 and normal classes improves when trained on the augmented dataset, resulting in an increase in accuracy and precision. However, low accuracy and low precision for the

non-COVID-19 pneumonia class are observed in both Figures 25a and 25b. This pattern is also observed in the confusion matrices of ResNet-18 and AlexNet (see Appendix G). We investigate whether this is due to deficiencies in HAC-GAN or the DTL classification models.

There is a slight decrease in classification accuracy for the non-COVID-19 pneumonia class when trained on the augmented dataset. Combined with the large number of pneumonia cases classified as COVID-19 cases in Figure 25b, this could indicate that HAC-GAN produces pneumonia images that are very similar to the COVID-19 class and is thus unable to generate distinctive non-COVID-19 pneumonia CXRs. However, we discussed the quality and diversity of the generated images in Section 5.1 and HAC-GAN was shown to produce high-quality and diverse images for all classes. It is thus unlikely that the low classification performance for the non-COVID-19 pneumonia class is due to the inability of HAC-GAN to generate images that resemble the original dataset. Also, the accuracy and precision are already at only 9.6% and 8.2% when trained on the original dataset, compared to 84.4% and 80.9% for the COVID-19 class and 84.4% and 91.6% for the normal class. This indicates that the DTL models have not learned the true pathological features that reflect the presence of non-COVID-19 pneumonia in the CXRs but could have learned patterns that reflect variations in image acquisition (DeGrave et al., 2021). The train and test sets contain images that originate from different datasets/hospitals for which the image acquisition process varies. Suppose a CNN has learned patterns that reflect variations in image acquisition. In that case, its performance will be low if the images in the train and test set originate from different datasets as the learned acquisition patterns from the train set are not present in the images from the test set. Augmenting the dataset with HAC-GAN then does not improve classification performance as the images generated by HAC-GAN resemble the images in the training set, which vary from the images in the test set.

So, employing HAC-GAN for data augmentation improves the detection of COVID-19, but it does not improve classification performance for all classes. This indicates that even though HAC-GAN generates high-quality and diverse synthetic samples, augmenting the dataset cannot guarantee that the classification models generalize well to unseen data for all the classes (i.e., data gathered by different hospitals). The results show that the proposed DTL models may suffer from variations caused by image acquisitions.



(a) Original dataset

(b) Augmented dataset by HAC-GAN

Figure 25: Confusion matrices for COVID-19 detection using SqueezeNet trained on the original dataset (a) and the augmented dataset by HAC-GAN (b). The accuracy/sensitivity (Acc) and precision (Pr) of each class are also given.

To gain better insights into the "black box" of how CNNs make predictions and to determine whether the DTL models make detection decisions based on pathological features or learned patterns based on variations in image acquisitions, we employ activation maps visualized with Grad-CAM (Selvaraju et al., 2017). A CNN's activation map indicates/highlights the regions of an image that most influence the model's class prediction. By analyzing the regions of greatest influence, we determine if the CNNs base their decision on relevant learned pathological features or other improper information. Appendix H elaborates on how Grad-CAM produces activation maps for the CNNs. The Grad-CAM activation maps of SqueezeNet on example images of each class from the train and test set are shown in Figure 26. We show both train and test images evaluated by Grad-CAM to visualize which patterns the CNN has learned from the training data, and determine if a difference exists in the decision process of train and test images (and thus if the model generalizes well to unseen data by employing proper learned patterns). The shown COVID-19 and normal test images are correctly classified by SqueezeNet, while the illustrated non-COVID-19 pneumonia image is wrongly classified. For the COVID-19 class, Figure 26 shows that SqueezeNet leverages lung areas in detecting COVID-19 pneumonia in the example CXRs. The highlights in the activation maps emphasize abnormalities in the lungs, which correspond to the presence of COVID-19. Abnormalities are highlighted in both the train and test images for the COVID-19 class, indicating that the CNN generalizes well to the COVID-19 test data. For the normal class, SqueezeNet has not found any abnormalities in the lungs that indicate the presence of (COVID-19) pneumonia, and the activation maps highlight other chest areas in both the train and test CXRs. The examples of the COVID-19 pneumonia and normal classes indicate that SqueezeNet has learned the true pathological features for detecting these classes. However, we observe that

the activation map for the non-COVID-19 pneumonia training image emphasizes areas outside the lung/chest fields. All image acquisition markers in the training image are highlighted, i.e., the laterality tokens, text, and arrows. The pneumonia test image does not contain these markers as its acquisition process differs from the training image. Here, the bottom of the image that does not contain any information about the presence/absence of (COVID-19) pneumonia is highlighted, resulting in a wrong classification. This validates that the CNN has learned patterns that reflect variations in image acquisition to detect non-COVID-19 pneumonia. As the learned acquisition patterns are not present in the test set due to varying acquisition processes (the train and test sets originate from different datasets/hospitals), the CNNs cannot correctly classify the non-COVID-19 pneumonia test images based on proper information. The activation maps verify that the proposed DTL models suffer from variations caused by image acquisitions for the non-COVID-19 pneumonia class and do not generalize well to unseen data gathered by different hospitals.

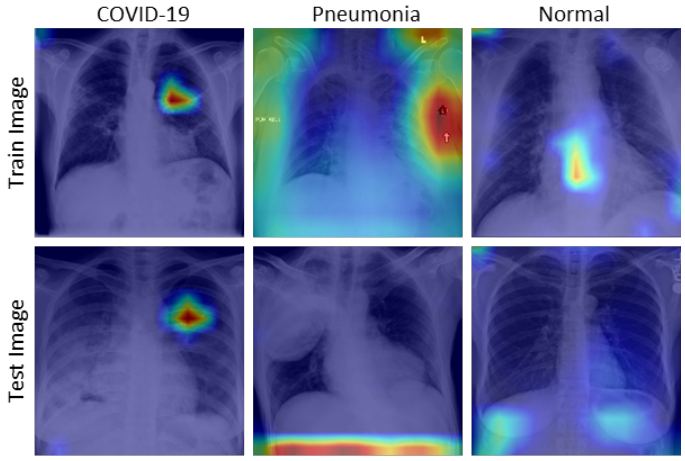


Figure 26: Grad-CAM activation maps for SqueezeNet indicating the regions of greatest influence on the CNN's prediction in each CXR. Both the activation maps for a train and test image are shown for each class, indicating which patterns the model has learned and if the model generalizes well to unseen data from the test set.

To determine if the proposed methods perform properly with "in-hospital" classification (i.e., when trained and tested on local data that originates from the same datasets/hospitals), we apply the DTL models on a new test set, which contains 270 COVID-19 images, 52 pneumonia images and 270 normal images that are randomly sampled from the combined datasets 1-7 from Section 2. This new train/test split represents an analysis on local data. With the new split, we again train HAC-GAN for 1000 epochs, augment the dataset, and train SqueezeNet, ResNet-18, and AlexNet for 25 epochs on both the original and augmented datasets. Table 6 shows the performance measures for the classification methods tested on an in-hospital test set. The best results per model are highlighted in bold. SqueezeNet, ResNet-18, and AlexNet achieve an accuracy of 93.6%, 95.9%,

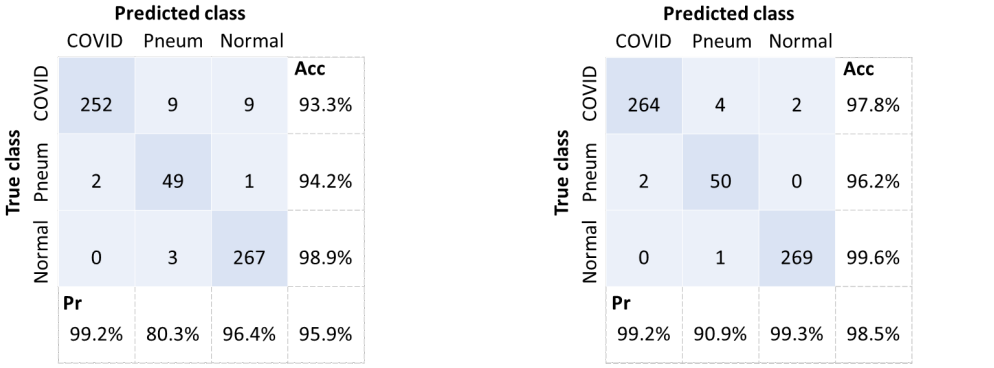
and 94.8% on the test set when trained on the original dataset, respectively. When trained on the augmented dataset, the CNNs' accuracies increase to 97.5%, 98.5%, and 96.8%, respectively. ResNet-18 performs best based on the accuracy. Table 6 also shows an increase in precision and sensitivity for all DTL models when trained on the augmented dataset. Augmenting the dataset again improves the detection performance of COVID-19 of all used pre-trained detection models. Next, we analyze the accuracy and precision per class. Figure 38 shows the confusion matrices of ResNet-18, which achieves the highest overall accuracy when trained on the augmented dataset. The matrices of SqueezeNet and AlexNet can be found in Appendix G. These results show that augmenting the dataset increases the accuracy and precision for all classes. The performance for the non-COVID-19 pneumonia class has increased compared to the original train/test split (see Figure 25). This demonstrates that the classification models can accurately detect the different classes when trained on local data, and that the CNNs performance suffered from variations caused by image acquisitions when trained and evaluated on the original train/test split.

Several Grad-CAM activation maps of correctly classified non-COVID-19 pneumonia CXRs by ResNet-18 are shown in Figure 28. We only show activation maps for this class as the CNNs failed to properly learn and detect the pathological features of the non-COVID-19 pneumonia class when trained on the original train/test split (see Figure 26). After training on the new split that represents an analysis on local data, ResNet-18's activation maps of both the train and test images now highlight abnormalities in the lung area, which correspond to the presence of non-COVID-19 pneumonia. Compared to the activation maps in Figure 26 for the pneumonia class, the CNNs now base their decisions for this class on learned pathological features rather than learned variations caused by differences in image acquisitions. Lung abnormalities are highlighted in both the train and test images for the non-COVID-19 pneumonia class, indicating that the DTL models can now also accurately detect the non-COVID-19 pneumonia class based on relevant information without relying on acquisition markers.

So, augmenting the dataset with images generated by HAC-GAN improves the classification performance of the fine-tuned CNNs SqueezeNet, ResNet-18 and AlexNet. This suggests that the synthesized images contain meaningful features that enhance the DTL classification performance. Also, when trained on local data, the DTL models can accurately detect the COVID-19 pneumonia, non-COVID-19 pneumonia, and normal classes in the CXRs based on learned pathological features.

Table 6 *Performance measures for detecting COVID-19 with the classification methods trained on the original and augmented datasets (by HAC-GAN) on the new train/test split (that represents an analysis on local data). Best results per model are highlighted in bold.*

Dataset	SqueezeNet			ResNet-18			AlexNet		
	Acc (%)	Pr (%)	Sn (%)	Acc (%)	Pr (%)	Sn (%)	Acc (%)	Pr (%)	Sn (%)
Original	93.6	97.0	95.9	95.9	99.2	93.3	94.8	98.5	95.9
Augmented	**97.5**	**99.6**	**97.8**	**98.5**	**99.2**	**97.8**	**96.8**	**98.9**	**97.0**



(a) Original dataset **(b)** Augmented dataset by HAC-GAN

Figure 27: Confusion matrices for COVID-19 detection using ResNet-18 trained on the original dataset (a) and the augmented dataset by HAC-GAN (b) utilizing the new train/test split (which represents an analysis on local data). The accuracy/sensitivity (Acc) and precision (Pr) of each class are also given.

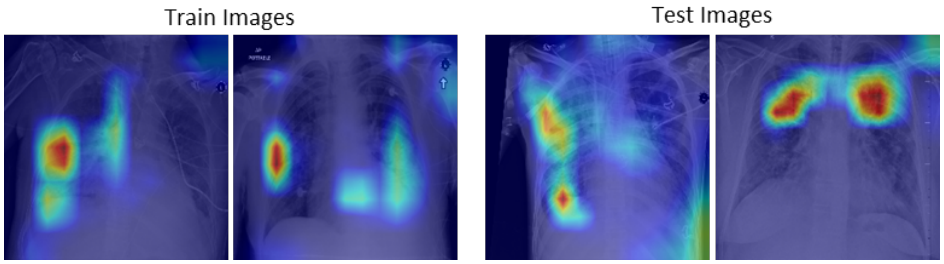


Figure 28: Grad-CAM activation maps for ResNet-18 indicating the regions of greatest influence on the CNN's prediction in four correctly classified non-COVID-19 pneumonia CXRs. The activation maps for several train and test images are shown, illustrating which patterns the CNN has learned and employs for detection.

6 Conclusion

COVID-19 is a respiratory disease that has a lasting effect on the health of the global population and worldwide economy, which stresses the importance of developing solutions that aid in the detection of COVID-19. Multiple deep learning methods have been proposed for detecting COVID-19 in radiographic images. However, a main concern with medical image classification remains the lack of available training data of sufficient size, which can lead to overfitting of the CNNs. This poses a major challenge for detecting COVID-19 using deep learning, as only limited datasets are available due to the relatively recent outbreak of COVID-19. In this research, we propose a customized

AC-GAN to generate fake CXR images that enlarge and balance the original dataset to improve the classification performance of CNNs. We further introduce HAC-GAN, which aims to stabilize training and improve the quality and diversity of the generated images by incorporating class hierarchies (in the form of a graph) in the AC-GAN. We propose to use the Node2Vec graph embedding method to learn hierarchy-aware node embeddings for each class of interest, which are then fed to the generator of HAC-GAN to synthesize images for each class. We propose a particular train/test split in the CXR dataset such that the train and test sets contain images that originate from different datasets, i.e., hospitals for which the image acquisition process varies. We employ this split to determine if the proposed models generalize well to unseen data.

The AC-GAN and introduced HAC-GAN are trained on the CXR dataset to synthesize images for the COVID-19 pneumonia, non-COVID-19 pneumonia, and normal classes. To verify if including hierarchy into a GAN improves the quality and diversity of the synthetic images, we also train AC-GAN and HAC-GAN on our benchmark Fruit and Vegetable datasets that contain more classes and different hierarchical structures than the CXR dataset. The generated images by both GANs are realistic, diverse, and resemble the CXRs from the original dataset. Both AC-GAN and HAC-GAN can properly generate synthetic CXRs for all classes. On average and for all individual classes, both the FID scores and MS-SSIM scores for HAC-GAN are lower than for AC-GAN. HAC-GAN thus generates images that resemble the original dataset more closely than AC-GAN, while it is also able to produce a more diverse set of CXRs. The same results are found for the Fruit and Vegetable datasets, and the experiments on the three datasets show the effectiveness of the introduced HAC-GAN, which is in line with the results of R. Zhang et al. (2020). These findings demonstrate that leveraging class hierarchy in GANs improves the quality and diversity of the generated images, and we therefore select HAC-GAN to augment the CXR dataset to aid in the detection of COVID-19 with deep transfer learning.

For classifying the CXRs in the test set, we train the CNNs SqueezeNet, ResNet-18, and AlexNet on both the original and augmented datasets using DTL. The results demonstrate that training on an augmented dataset improves the overall classification performance of all three CNNs based on accuracy, precision, and sensitivity. SqueezeNet performs best (based on the accuracy after augmentation) and its accuracy increases from 77.9% to 86.1%. The precision and sensitivity for detecting COVID-19 increase from 80.9% to 82.3% and 84.4% to 93.0%, respectively. The per-class results and the Grad-CAM visualizations of the COVID-19 pneumonia and normal classes indicate

that the CNNs have learned the true pathological features for detecting these classes. However, the per-class results and the Grad-CAM illustrations of the non-COVID-19 pneumonia class on the original train/test split show that the DTL models suffer from learned differences caused by variations in image acquisition and do not generalize well to unseen non-COVID-19 pneumonia data gathered by different hospitals. To determine if the proposed methods perform properly with "in-hospital" classification, we also train and evaluate HAC-GAN and the DTL models using a new train/test split that represents an analysis on local data, where the test set is randomly sampled from all available data. Augmenting the dataset now improves both the overall and per-class classification performance for all classes of the CNNs. Besides, the Grad-CAM results demonstrate that the CNNs now base their classification decisions on learned pathological features for each class without relying on acquisition markers. ResNet-18 performs best, and its accuracy increased from 95.9% to 98.5% after training on the augmented dataset. The precision and sensitivity for detecting COVID-19 with ResNet-18 increase to 99.2% and 97.8%, respectively. The increase in sensitivity is essential since we aim to limit the number of missed COVID-19 samples as much as possible. The increase in precision means that fewer false-positive COVID-19 predictions are made, which could result in a decrease of the burden on the health care system. These results demonstrate that the CXRs generated by HAC-GAN contain meaningful features that enhance the classification performance of all three CNNs. In conclusion, the findings of this research reveal that the DTL models can accurately detect the different classes when trained on local data that is augmented by HAC-GAN. To leverage the potential of the proposed methods in practice, HAC-GAN and the CNNs need to be trained on local or "in-hospital" datasets. Note that our method is not a substitute for PCR testing and should always be used in combination with medical assistance and clinical testing.

For future works, we recommend extending this research in four directions. First, it would be interesting to explore different architectures, improved evaluation during training, and different hyperparameter settings of the GANs. We employ a customized AC-GAN as it is able to produce relatively high-quality images while being computationally efficient. However, many new GAN architectures have been proposed in the literature recently. We recommend implementing GANs other than AC-GAN that can produce higher resolution images with increased discriminability that might better capture the pathological features of COVID-19 (Odena et al., 2017). Also, the training of GANs is often unstable. To determine successful training, the MS-SSIM and FID metrics could be tracked and evaluated during training. A mode collapse of the generator will be associated with

increasing MS-SSIM and FID scores as training progresses. Besides, we base our implementation of (H)AC-GAN and the used hyperparameters during training on the work of Odena et al. (2017) and Radford et al. (2015). However, the use of different settings to optimize training could be investigated.

Second, we use the popular node embedding method Node2Vec to incorporate class hierarchies in HAC-GAN. We employ the default parameter settings proposed in Grover and Leskovec (2016) to ensure a DFS sampling strategy. To further refine the embeddings generated by Node2Vec, a grid search where Node2Vec's model parameters are optimized could be considered. Also, many graph embedding methods have been proposed in recent years, and it would be interesting to employ and evaluate those in future works.

Third, the per-class classification results and the Grad-CAM visualizations for the non-COVID-19 pneumonia class on the original train/test split demonstrate that the proposed DTL models suffer from variations caused by image acquisitions and do not generalize well to unseen data acquired from different hospitals for this class. Therefore, it would be interesting for future works to comment on the performance of GANs and CNNs when all present acquisition markers, including text, laterality tokens, and arrows, are first removed from the CXRs before training the models. Future works could investigate whether models trained on this pre-processed dataset containing CXRs without any acquisition markers generalize well and are not biased towards a specific image acquisition process.

Lastly, HAC-GAN is not limited to synthesizing CXRs and it would be interesting to examine the performance of HAC-GAN in other domains. For example, HAC-GAN can be employed in other medical imaging fields or any research where the classes of interest are structured hierarchically.

References

Ahmed, M. I., Mamun, S. M., & Asif, A. U. Z. (2021). Dcnn-based vegetable image classification using transfer learning: A comparative study. In *2021 5th international conference on computer, communication and signal processing (icccsp)* (pp. 235–243).

Albawi, S., Mohammed, T. A., & Al-Zawi, S. (2017). Understanding of a convolutional neural network. In *2017 international conference on engineering and technology (icet)* (pp. 1–6).

Alqahtani, H., Kavakli-Thorne, M., & Kumar, G. (2021). Applications of generative adversarial networks (gans): An updated review. *Archives of Computational Methods in Engineering*, *28*(2), 525–552.

Al-Shargabi, A. A., Alshobaili, J. F., Alabdulatif, A., & Alrobah, N. (2021). Covid-cgan: Efficient deep learning approach for covid-19 detection based on cxr images using conditional gans. *Applied Sciences*, *11*(16), 7174.

Borji, A. (2019). Pros and cons of gan evaluation measures. *Computer Vision and Image Understanding*, *179*, 41–65.

Candemir, S., Jaeger, S., Palaniappan, K., Musco, J. P., Singh, R. K., Xue, Z., ... McDonald, C. J. (2013). Lung segmentation in chest radiographs using anatomical atlases with nonrigid registration. *IEEE transactions on medical imaging*, *33*(2), 577–590.

Chollet, F. (2017). Xception: Deep learning with depthwise separable convolutions. In *Proceedings of the ieee conference on computer vision and pattern recognition* (pp. 1251–1258).

Chowdhury, M. E., Rahman, T., Khandakar, A., Mazhar, R., Kadir, M. A., Mahbub, Z. B., ... others (2020). Can ai help in screening viral and covid-19 pneumonia? *IEEE Access*, *8*, 132665–132676.

Chung, A. (2020a). Actualmed covid-19 chest x-ray data initiative. Retrieved from https://github.com/agchung/Actualmed-COVID-chestxray-dataset

Chung, A. (2020b). Figure 1 covid-19 chest x-ray data initiative. Retrieved from https://github.com/agchung/Figure1-COVID-chestxray-dataset

Cohen, J. P., Morrison, P., Dao, L., Roth, K., Duong, T. Q., & Ghassemi, M. (2020). Covid-19 image data collection: Prospective predictions are the future. *arXiv preprint arXiv:2006.11988*. Retrieved from https://github.com/ieee8023/covid-chestxray-dataset

Collobert, R., Kavukcuoglu, K., & Farabet, C. (2011). Torch7: A matlab-like environment for machine learning. In *Biglearn, nips workshop*.

Corman, V. M., Landt, O., Kaiser, M., Molenkamp, R., Meijer, A., Chu, D. K., ... others (2020). Detection of 2019 novel coronavirus (2019-ncov) by real-time rt-pcr. *Eurosurveillance*, *25*(3), 2000045.

Creswell, A., White, T., Dumoulin, V., Arulkumaran, K., Sengupta, B., & Bharath, A. A. (2018). Generative adversarial networks: An overview. *IEEE Signal Processing Magazine*, *35*(1), 53–65.

DeGrave, A. J., Janizek, J. D., & Lee, S.-I. (2021). Ai for radiographic covid-19 detection selects shortcuts over signal. *Nature Machine Intelligence*, 1–10.

Dumoulin, V., & Visin, F. (2016). A guide to convolution arithmetic for deep learning. *arXiv preprint arXiv:1603.07285*.

Esteva, A., Kuprel, B., Novoa, R. A., Ko, J., Swetter, S. M., Blau, H. M., & Thrun, S. (2017). Dermatologist-level classification of skin cancer with deep neural networks. *nature*, *542*(7639), 115–118.

Fey, M., & Lenssen, J. E. (2019). Fast graph representation learning with PyTorch Geometric. In *Iclr workshop on representation learning on graphs and manifolds.*

Glorot, X., Bordes, A., & Bengio, Y. (2011). Deep sparse rectifier neural networks. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics* (pp. 315–323).

Goldberg, Y., & Levy, O. (2014). word2vec explained: deriving mikolov et al.'s negative-sampling word-embedding method. *arXiv preprint arXiv:1402.3722*.

Goodfellow, I. (2016). Nips 2016 tutorial: Generative adversarial networks. *arXiv preprint arXiv:1701.00160*.

Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., ... Bengio, Y. (2014). Generative adversarial nets. *Advances in neural information processing systems*, *27*.

Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., ... Bengio, Y. (2020). Generative adversarial networks. *Communications of the ACM*, *63*(11), 139–144.

Grover, A., & Leskovec, J. (2016). node2vec: Scalable feature learning for networks. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining* (pp. 855–864).

Guan, W.-j., Ni, Z.-y., Hu, Y., Liang, W.-h., Ou, C.-q., He, J.-x., ... others (2020). Clinical characteristics of coronavirus disease 2019 in china. *New England journal of medicine*, *382*(18), 1708–1720.

Guimaraes, G. L., Sanchez-Lengeling, B., Outeiral, C., Farias, P. L. C., & Aspuru-Guzik, A. (2017).

Objective-reinforced generative adversarial networks (organ) for sequence generation models. *arXiv preprint arXiv:1705.10843*.

Hall, L. O., Paul, R., Goldgof, D. B., & Goldgof, G. M. (2020). Finding covid-19 from chest x-rays using deep learning on a small dataset. *arXiv preprint arXiv:2004.02060*.

He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. In *Proceedings of the ieee conference on computer vision and pattern recognition* (pp. 770–778).

Heusel, M., Ramsauer, H., Unterthiner, T., Nessler, B., & Hochreiter, S. (2017). Gans trained by a two time-scale update rule converge to a local nash equilibrium. *Advances in neural information processing systems*, *30*.

Hong, Y., Hwang, U., Yoo, J., & Yoon, S. (2019). How generative adversarial networks and their variants work: An overview. *ACM Computing Surveys (CSUR)*, *52*(1), 1–43.

Huang, C., Wang, Y., Li, X., Ren, L., Zhao, J., Hu, Y., ... others (2020). Clinical features of patients infected with 2019 novel coronavirus in wuhan, china. *The lancet*, *395*(10223), 497–506.

Huang, G., Liu, Z., Van Der Maaten, L., & Weinberger, K. Q. (2017). Densely connected convolutional networks. In *Proceedings of the ieee conference on computer vision and pattern recognition* (pp. 4700–4708).

Huszár, F. (2016). *Instance noise: A trick for stabilising gan training.* Retrieved from https://www.inference.vc/instance-noise-a-trick-for-stabilising-gan-training/

Iandola, F. N., Han, S., Moskewicz, M. W., Ashraf, K., Dally, W. J., & Keutzer, K. (2016). Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <0.5mb model size. *arXiv:1602.07360*.

Ioffe, S., & Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning* (pp. 448–456).

Jaeger, S., Karargyris, A., Candemir, S., Folio, L., Siegelman, J., Callaghan, F., ... others (2013). Automatic tuberculosis screening using chest radiographs. *IEEE transactions on medical imaging*, *33*(2), 233–245.

Karras, T., Aila, T., Laine, S., & Lehtinen, J. (2017). Progressive growing of gans for improved quality, stability, and variation. *arXiv preprint arXiv:1710.10196*.

Khobahi, S., Agarwal, C., & Soltanalian, M. (2020). Coronet: A deep network architecture for semi-supervised task-based identification of covid-19 from chest x-ray images. *MedRxiv*.

Kingma, D. P., & Ba, J. (2015). Adam: A method for stochastic optimization. *CoRR*, *abs/1412.6980*.

Kong, J., Kim, J., & Bae, J. (2020). Hifi-gan: Generative adversarial networks for efficient and high fidelity speech synthesis. *arXiv preprint arXiv:2010.05646*.

Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, *25*.

LeCun, Y., Boser, B., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W., & Jackel, L. D. (1989). Backpropagation applied to handwritten zip code recognition. *Neural computation*, *1*(4), 541–551.

Li, Y., Yao, L., Li, J., Chen, L., Song, Y., Cai, Z., & Yang, C. (2020). Stability issues of rt-pcr testing of sars-cov-2 for hospitalized patients clinically diagnosed with covid-19. *Journal of medical virology*, *92*(7), 903–908.

Lo, S.-C., Lou, S.-L., Lin, J.-S., Freedman, M. T., Chien, M. V., & Mun, S. K. (1995). Artificial convolution neural network techniques and applications for lung nodule detection. *IEEE transactions on medical imaging*, *14*(4), 711–718.

Loey, M., Smarandache, F., & M Khalifa, N. E. (2020). Within the lack of chest covid-19 x-ray dataset: a novel detection model based on gan and deep transfer learning. *Symmetry*, *12*(4), 651.

Luz, E., Silva, P., Silva, R., Silva, L., Guimarães, J., Miozzo, G., ... Menotti, D. (2021). Towards an effective and efficient deep learning model for covid-19 patterns detection in x-ray images. *Research on Biomedical Engineering*, 1–14.

McCulloch, W. S., & Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, *5*(4), 115–133.

Mihai-Dimitrie, M. (2021). *Fruits dataset. fruits classification.* Retrieved from https://www.kaggle.com/datasets/aelchimminut/fruits262

Mikolov, T., Chen, K., Corrado, G., & Dean, J. (2013). Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*.

Mikolov, T., Sutskever, I., Chen, K., Corrado, G. S., & Dean, J. (2013). Distributed representations of words and phrases and their compositionality. *Advances in neural information processing systems*, *26*.

Minaee, S., Kafieh, R., Sonka, M., Yazdani, S., & Soufi, G. J. (2020). Deep-covid: Predicting covid-19 from chest x-ray images using deep transfer learning. *Medical image analysis*, *65*,

101794.

Mirza, M., & Osindero, S. (2014). Conditional generative adversarial nets. *arXiv preprint arXiv:1411.1784*.

Nair, V., & Hinton, G. E. (2010). Rectified linear units improve restricted boltzmann machines. In *Icml*.

Nakashima, K. (2020). *Pytorch implementation of grad-cam, vanilla/guided backpropagation, deconvnet, and occlusion sensitivity maps.* GitHub. Retrieved from https://github.com/kazuto1011/grad-cam-pytorch

Narin, A., Kaya, C., & Pamuk, Z. (2021). Automatic detection of coronavirus disease (covid-19) using x-ray images and deep convolutional neural networks. *Pattern Analysis and Applications*, 1–14.

Odena, A., Olah, C., & Shlens, J. (2017). Conditional image synthesis with auxiliary classifier gans. In *International conference on machine learning* (pp. 2642–2651).

Oord, A. v. d., Kalchbrenner, N., Vinyals, O., Espeholt, L., Graves, A., & Kavukcuoglu, K. (2016). Conditional image generation with pixelcnn decoders. *arXiv preprint arXiv:1606.05328*.

Pan, Z., Yu, W., Yi, X., Khan, A., Yuan, F., & Zheng, Y. (2019). Recent progress on generative adversarial networks (gans): A survey. *IEEE Access*, *7*, 36322–36333.

Parmar, G., Zhang, R., & Zhu, J.-Y. (2022). On aliased resizing and surprising subtleties in gan evaluation. In *Cvpr*.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., ... Chintala, S. (2019). Pytorch: An imperative style, high-performance deep learning library. In *Advances in neural information processing systems 32* (pp. 8024–8035). Curran Associates, Inc. Retrieved from http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

Perozzi, B., Al-Rfou, R., & Skiena, S. (2014). Deepwalk: Online learning of social representations. In *Proceedings of the 20th acm sigkdd international conference on knowledge discovery and data mining* (pp. 701–710).

Pessoa, J. (2021). *Pytorch differentiable multi-scale structural similarity (ms-ssim) loss.* GitHub. Retrieved from https://github.com/jorge-pessoa/pytorch-msssim

Radford, A., Metz, L., & Chintala, S. (2015). Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434*.

Radiological Society of North America. (2019a). *Covid-19 radiography database.* Retrieved from

https://www.kaggle.com/tawsifurrahman/covid19-radiography-database

Radiological Society of North America. (2019b). *Rsna pneumonia detection challenge.* Retrieved from https://www.kaggle.com/c/rsna-pneumonia-detection-challenge/data

Rajaraman, S., Siegelman, J., Alderson, P. O., Folio, L. S., Folio, L. R., & Antani, S. K. (2020). Iteratively pruned deep learning ensembles for covid-19 detection in chest x-rays. *IEEE Access*, *8*, 115041–115050.

Ravishankar, H., Sudhakar, P., Venkataramani, R., Thiruvenkadam, S., Annangi, P., Babu, N., & Vaidya, V. (2016). Understanding the mechanisms of deep transfer learning for medical images. In *Deep learning and data labeling for medical applications* (pp. 188–196). Springer.

Reed, S., Akata, Z., Yan, X., Logeswaran, L., Schiele, B., & Lee, H. (2016). Generative adversarial text to image synthesis. In *International conference on machine learning* (pp. 1060–1069).

Rong, X. (2014). word2vec parameter learning explained. *arXiv preprint arXiv:1411.2738*.

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1985). *Learning internal representations by error propagation* (Tech. Rep.). California Univ San Diego La Jolla Inst for Cognitive Science.

Sahiner, B., Chan, H.-P., Petrick, N., Wei, D., Helvie, M. A., Adler, D. D., & Goodsitt, M. M. (1996). Classification of mass and normal breast tissue: a convolution neural network classifier with spatial domain and texture images. *IEEE transactions on Medical Imaging*, *15*(5), 598–610.

Salimans, T., Goodfellow, I., Zaremba, W., Cheung, V., Radford, A., & Chen, X. (2016). Improved techniques for training gans. *Advances in neural information processing systems*, *29*.

Selvaraju, R. R., Cogswell, M., Das, A., Vedantam, R., Parikh, D., & Batra, D. (2017). Grad-cam: Visual explanations from deep networks via gradient-based localization. In *Proceedings of the ieee international conference on computer vision* (pp. 618–626).

Serener, A., & Serte, S. (2020). Geographic variation and ethnicity in diabetic retinopathy detection via deep learning. *Turkish Journal of Electrical Engineering & Computer Sciences*, *28*(2), 664–678.

Shrirao, S. (2021). *Generate 256x256, 512x512 resolution images with simple convolutional gan by adding gaussian noise to discriminator layers.* Retrieved from https://github.com/ShivamShrirao/facegan_pytorch

Sokolova, M., Japkowicz, N., & Szpakowicz, S. (2006). Beyond accuracy, f-score and roc: a family of discriminant measures for performance evaluation. In *Australasian joint conference on*

artificial intelligence (pp. 1015–1021).

Sønderby, C. K., Caballero, J., Theis, L., Shi, W., & Huszár, F. (2016). Amortised map inference for image super-resolution. *arXiv preprint arXiv:1610.04490*.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, *15*(1), 1929–1958.

Szegedy, C., Ioffe, S., Vanhoucke, V., & Alemi, A. A. (2017). Inception-v4, inception-resnet and the impact of residual connections on learning. In *Thirty-first aaai conference on artificial intelligence.*

Tang, J., Qu, M., Wang, M., Zhang, M., Yan, J., & Mei, Q. (2015). Line: Large-scale information network embedding. In *Proceedings of the 24th international conference on world wide web* (pp. 1067–1077).

Tang, L., & Liu, H. (2011). Leveraging social media networks for classification. *Data Mining and Knowledge Discovery*, *23*(3), 447–478.

Tharwat, A. (2020). Classification assessment methods. *Applied Computing and Informatics*.

Tsai, E., Simpson, S., Lungren, M., Hershman, M., Roshkovan, L., Colak, E., ... Wu, C. (2021). Data from medical imaging data resource center (midrc) - rsna international covid radiology database (ricord) release 1c - chest x-ray, covid+ (midrc-ricord-1c). *The Cancer Imaging Archive*. doi: https://doi.org/10.7937/91ah-v663.

Waheed, A., Goyal, M., Gupta, D., Khanna, A., Al-Turjman, F., & Pinheiro, P. R. (2020). Covidgan: data augmentation using auxiliary classifier gan for improved covid-19 detection. *Ieee Access*, *8*, 91916–91923.

Wang, L., Lin, Z. Q., & Wong, A. (2020). Covid-net: A tailored deep convolutional neural network design for detection of covid-19 cases from chest x-ray images. *Scientific Reports*, *10*(1), 1–12.

Wang, X., Peng, Y., Lu, L., Lu, Z., Bagheri, M., & Summers, R. M. (2017). Chestx-ray8: Hospital-scale chest x-ray database and benchmarks on weakly-supervised classification and localization of common thorax diseases. In *Proceedings of the ieee conference on computer vision and pattern recognition* (pp. 2097–2106).

Wang, X., Yu, K., Wu, S., Gu, J., Liu, Y., Dong, C., ... Change Loy, C. (2018). Esrgan: Enhanced super-resolution generative adversarial networks. In *Proceedings of the european conference on computer vision (eccv) workshops* (pp. 0–0).

Wang, Z., She, Q., & Ward, T. E. (2021). Generative adversarial networks in computer vision: A survey and taxonomy. *ACM Computing Surveys (CSUR)*, *54*(2), 1–38.

Wang, Z., Simoncelli, E. P., & Bovik, A. C. (2003). Multiscale structural similarity for image quality assessment. In *The thrity-seventh asilomar conference on signals, systems & computers, 2003* (Vol. 2, pp. 1398–1402).

Weng, L. (2017). Learning word embedding. *URL: https://lilianweng. github. io/lil-log/2017/10/15/learning-wordembedding. html*.

Werbos, P. (1974). Beyond regression:" new tools for prediction and analysis in the behavioral sciences. *Ph. D. dissertation, Harvard University*.

World Health Organization. (2021). *Who coronavirus (covid-19) dashboard.* Retrieved from https://covid19.who.int/

Wu, H., Zheng, S., Zhang, J., & Huang, K. (2019). Gp-gan: Towards realistic high-resolution image blending. In *Proceedings of the 27th acm international conference on multimedia* (pp. 2487–2495).

Zeiler, M. D., Krishnan, D., Taylor, G. W., & Fergus, R. (2010). Deconvolutional networks. In *2010 ieee computer society conference on computer vision and pattern recognition* (pp. 2528–2535).

Zhang, H., Xu, T., Li, H., Zhang, S., Wang, X., Huang, X., & Metaxas, D. N. (2017). Stackgan: Text to photo-realistic image synthesis with stacked generative adversarial networks. In *Proceedings of the ieee international conference on computer vision* (pp. 5907–5915).

Zhang, R., Mou, L., & Xie, P. (2020). Treegan: Incorporating class hierarchy into image generation. *arXiv preprint arXiv:2009.07734*.

Zhao, B., Chang, B., Jie, Z., & Sigal, L. (2018). Modular generative adversarial networks. In *Proceedings of the european conference on computer vision (eccv)* (pp. 150–165).

Zhuang, F., Qi, Z., Duan, K., Xi, D., Zhu, Y., Zhu, H., ... He, Q. (2021). A comprehensive survey on transfer learning. *Proceedings of the IEEE*, *109*(1), 43-76. doi: 10.1109/JPROC .2020.3004555

A Example Images of the Fruit and Vegetable Datasets

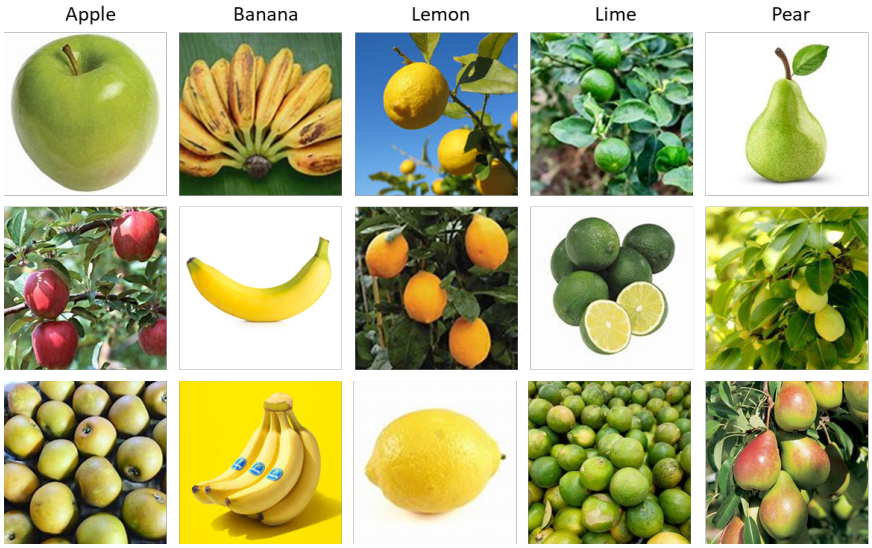


Figure 29: Example images of the Fruit dataset (Mihai-Dimitrie, 2021), which contains the following classes; apple, banana, lemon, lime and pear.

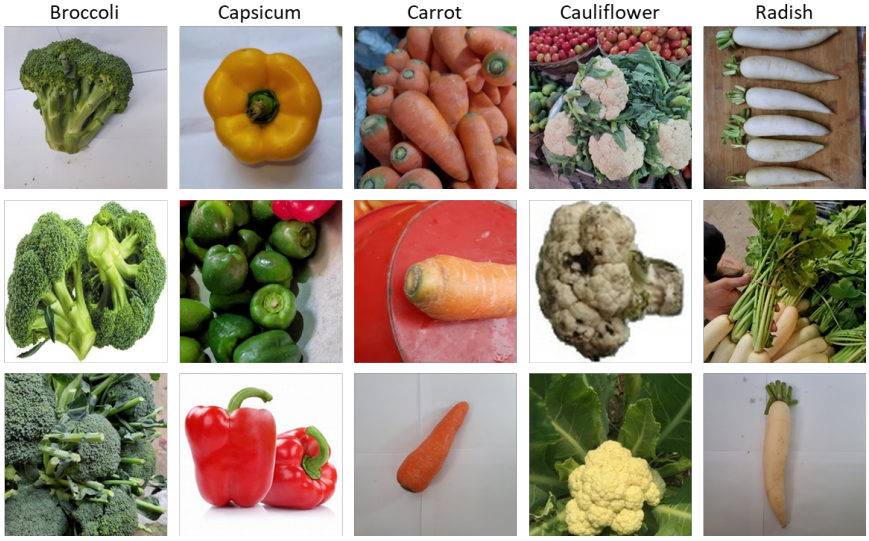


Figure 30: Example images of the Vegetable dataset (Ahmed et al., 2021), which contains the following classes; broccoli, capsicum, carrot, cauliflower and radish.

B Summary of the Discriminator in (H)AC-GAN

Summary of the discriminator D in (H)AC-GAN, where the output shape and number of trainable parameters of each layer/module are given.

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
     GaussianNoise-1          [-1, 3, 128, 128]               0
            Conv2d-2           [-1, 64, 64, 64]           3,072
        LeakyReLU-3           [-1, 64, 64, 64]               0
        Dropout2d-4           [-1, 64, 64, 64]               0
     GaussianNoise-5           [-1, 64, 64, 64]               0
            Conv2d-6          [-1, 128, 32, 32]         131,072
       BatchNorm2d-7          [-1, 128, 32, 32]             256
        LeakyReLU-8           [-1, 128, 32, 32]               0
        Dropout2d-9           [-1, 128, 32, 32]               0
    GaussianNoise-10          [-1, 128, 32, 32]               0
           Conv2d-11          [-1, 256, 16, 16]         524,288
      BatchNorm2d-12          [-1, 256, 16, 16]             512
       LeakyReLU-13           [-1, 256, 16, 16]               0
       Dropout2d-14           [-1, 256, 16, 16]               0
    GaussianNoise-15          [-1, 256, 16, 16]               0
           Conv2d-16            [-1, 512, 8, 8]       2,097,152
      BatchNorm2d-17            [-1, 512, 8, 8]           1,024
       LeakyReLU-18            [-1, 512, 8, 8]               0
       Dropout2d-19            [-1, 512, 8, 8]               0
    GaussianNoise-20            [-1, 512, 8, 8]               0
           Conv2d-21           [-1, 1024, 4, 4]       8,388,608
      BatchNorm2d-22           [-1, 1024, 4, 4]           2,048
       LeakyReLU-23           [-1, 1024, 4, 4]               0
       Dropout2d-24           [-1, 1024, 4, 4]               0
    GaussianNoise-25           [-1, 1024, 4, 4]               0
           Conv2d-26              [-1, 1, 1, 1]          16,384
          Sigmoid-27              [-1, 1, 1, 1]               0
    GaussianNoise-28           [-1, 1024, 4, 4]               0
           Conv2d-29              [-1, 3, 1, 1]          49,152
       LogSoftmax-30              [-1, 3, 1, 1]               0
================================================================
Total params: 11,213,568
Trainable params: 11,213,568
Non-trainable params: 0
----------------------------------------------------------------
```

C Summary of the Generator in (H)AC-GAN

Summary of the generator G in (H)AC-GAN, where the output shape and number of trainable parameters of each layer/module are given.

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
         Embedding-1                 [-1, 100]             300
   ConvTranspose2d-2           [-1, 1024, 4, 4]       1,638,400
       BatchNorm2d-3           [-1, 1024, 4, 4]           2,048
              ReLU-4           [-1, 1024, 4, 4]               0
         Dropout2d-5           [-1, 1024, 4, 4]               0
   ConvTranspose2d-6            [-1, 512, 8, 8]       8,388,608
       BatchNorm2d-7            [-1, 512, 8, 8]           1,024
              ReLU-8            [-1, 512, 8, 8]               0
         Dropout2d-9            [-1, 512, 8, 8]               0
  ConvTranspose2d-10          [-1, 256, 16, 16]       2,097,152
      BatchNorm2d-11          [-1, 256, 16, 16]             512
             ReLU-12          [-1, 256, 16, 16]               0
        Dropout2d-13          [-1, 256, 16, 16]               0
  ConvTranspose2d-14          [-1, 128, 32, 32]         524,288
      BatchNorm2d-15          [-1, 128, 32, 32]             256
             ReLU-16          [-1, 128, 32, 32]               0
        Dropout2d-17          [-1, 128, 32, 32]               0
  ConvTranspose2d-18           [-1, 64, 64, 64]         131,072
      BatchNorm2d-19           [-1, 64, 64, 64]             128
             ReLU-20           [-1, 64, 64, 64]               0
        Dropout2d-21           [-1, 64, 64, 64]               0
  ConvTranspose2d-22          [-1, 3, 128, 128]           3,072
             Tanh-23          [-1, 3, 128, 128]               0
================================================================
Total params: 12,786,860
Trainable params: 12,786,860
Non-trainable params: 0
----------------------------------------------------------------
```

D Architectures of SqueezeNet, ResNet-18 and AlexNet

Summaries of the pre-trained CNNs SqueezeNet, ResNet-18 and AlexNet adjusted for detecting COVID-19 in the CXR dataset. The last fully connected/convolutional layer of these models is removed and replaced with a modified classifier. The modified classifiers are highlighted in yellow. The used DTL models are the pre-trained CNNs `squeezenet1_1`, `resnet18` and `alexnet` from `torchvision.models` in the `PyTorch` library (Paszke et al., 2019). The following summary is of the pre-trained CNN SqueezeNet.

```
SqueezeNet(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(2, 2))
    (1): ReLU(inplace=True)
    (2): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=True)
    (3): Fire(
      (squeeze): Conv2d(64, 16, kernel_size=(1, 1), stride=(1, 1))
      (squeeze_activation): ReLU(inplace=True)
      (expand1x1): Conv2d(16, 64, kernel_size=(1, 1), stride=(1, 1))
      (expand1x1_activation): ReLU(inplace=True)
      (expand3x3): Conv2d(16, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (expand3x3_activation): ReLU(inplace=True)
    )
    (4): Fire(
      (squeeze): Conv2d(128, 16, kernel_size=(1, 1), stride=(1, 1))
      (squeeze_activation): ReLU(inplace=True)
      (expand1x1): Conv2d(16, 64, kernel_size=(1, 1), stride=(1, 1))
      (expand1x1_activation): ReLU(inplace=True)
      (expand3x3): Conv2d(16, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (expand3x3_activation): ReLU(inplace=True)
    )
    (5): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=True)
    (6): Fire(
      (squeeze): Conv2d(128, 32, kernel_size=(1, 1), stride=(1, 1))
      (squeeze_activation): ReLU(inplace=True)
      (expand1x1): Conv2d(32, 128, kernel_size=(1, 1), stride=(1, 1))
      (expand1x1_activation): ReLU(inplace=True)
      (expand3x3): Conv2d(32, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (expand3x3_activation): ReLU(inplace=True)
    )
    (7): Fire(
      (squeeze): Conv2d(256, 32, kernel_size=(1, 1), stride=(1, 1))
      (squeeze_activation): ReLU(inplace=True)
      (expand1x1): Conv2d(32, 128, kernel_size=(1, 1), stride=(1, 1))
      (expand1x1_activation): ReLU(inplace=True)
      (expand3x3): Conv2d(32, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (expand3x3_activation): ReLU(inplace=True)
    )
    (8): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=True)
    (9): Fire(
      (squeeze): Conv2d(256, 48, kernel_size=(1, 1), stride=(1, 1))
      (squeeze_activation): ReLU(inplace=True)
      (expand1x1): Conv2d(48, 192, kernel_size=(1, 1), stride=(1, 1))
      (expand1x1_activation): ReLU(inplace=True)
      (expand3x3): Conv2d(48, 192, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (expand3x3_activation): ReLU(inplace=True)
    )
    (10): Fire(
      (squeeze): Conv2d(384, 48, kernel_size=(1, 1), stride=(1, 1))
      (squeeze_activation): ReLU(inplace=True)
      (expand1x1): Conv2d(48, 192, kernel_size=(1, 1), stride=(1, 1))
```

```
      (expand1x1_activation): ReLU(inplace=True)
      (expand3x3): Conv2d(48, 192, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (expand3x3_activation): ReLU(inplace=True)
    )
    (11): Fire(
      (squeeze): Conv2d(384, 64, kernel_size=(1, 1), stride=(1, 1))
      (squeeze_activation): ReLU(inplace=True)
      (expand1x1): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1))
      (expand1x1_activation): ReLU(inplace=True)
      (expand3x3): Conv2d(64, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (expand3x3_activation): ReLU(inplace=True)
    )
    (12): Fire(
      (squeeze): Conv2d(512, 64, kernel_size=(1, 1), stride=(1, 1))
      (squeeze_activation): ReLU(inplace=True)
      (expand1x1): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1))
      (expand1x1_activation): ReLU(inplace=True)
      (expand3x3): Conv2d(64, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (expand3x3_activation): ReLU(inplace=True)
    )
  )
  (classifier): Sequential(
    (0): Dropout(p=0.5, inplace=False)
    (1): Conv2d(512, 3, kernel_size=(1, 1), stride=(1, 1))
    (2): ReLU(inplace=True)
    (3): AdaptiveAvgPool2d(output_size=(1, 1))
  )
)
```

Summary of the pre-trained CNN ResNet-18 adjusted for detecting COVID-19 in the CXR dataset.

```
ResNet(
  (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace=True)
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
  (layer1): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
    (1): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (layer2): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(64, 128, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
```

```
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (layer3): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(128, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(128, 256, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (layer4): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(256, 512, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(1, 1))
  (fc): Linear(in_features=512, out_features=3, bias=True)
)
```

Summary of the pre-trained CNN AlexNet adjusted for detecting COVID-19 in the CXR dataset.

```
AlexNet(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(11, 11), stride=(4, 4), padding=(2, 2))
    (1): ReLU(inplace=True)
    (2): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
    (3): Conv2d(64, 192, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
```

```
    (4): ReLU(inplace=True)
    (5): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
    (6): Conv2d(192, 384, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (7): ReLU(inplace=True)
    (8): Conv2d(384, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (9): ReLU(inplace=True)
    (10): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU(inplace=True)
    (12): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(6, 6))
  (classifier): Sequential(
    (0): Dropout(p=0.5, inplace=False)
    (1): Linear(in_features=9216, out_features=4096, bias=True)
    (2): ReLU(inplace=True)
    (3): Dropout(p=0.5, inplace=False)
    (4): Linear(in_features=4096, out_features=4096, bias=True)
    (5): ReLU(inplace=True)
    (6): Linear(in_features=4096, out_features=3, bias=True)
  )
)
```

E (H)AC-GAN Training Process

This section elaborates on the training process of the implemented GANs after training on the CXR dataset. As stated in Section 3.2.1, the generator G aims to minimize $L^{(G)}$ while the discriminator D aims to minimize $L^{(D)}$. D and G have conflicting goals, which means that improvements to one network come at the expense of the other. If the generator improves with training, the discriminator performance gets worse as it cannot easily distinguish between real and generated images. If the discriminator gets better, the losses of the generator will increase. This does not necessarily mean that the quality of the generated images also deteriorates, only that the discriminator gets better at predicting the source of the images. The goal of training is to find an equilibrium between the generator and discriminator, while the challenge of simultaneously training these two competing models is that they can fail to converge due to unstable training processes.

Figure 31 and Figure 32 show the training process in terms of the losses of the generator G ($L^{(G)}$ from equation (8)) and discriminator D ($L^{(D)}$ from equation (7)) over the training iterations of AC-GAN and HAC-GAN, respectively. The figures show that the training processes of AC-GAN and HAC-GAN follow the same patterns. The losses of both generators G first decrease and start increasing around iteration 15.000. $L^{(G)}$ then converges at around 5 after iteration 150.000. The discriminator loss $L^{(D)}$ decreases as training progresses and stabilizes at around 0.4. Note that the values of the GAN losses are not informative themselves, but the loss convergence signifies that the models have learned properly and an equilibrium has been found, indicating successful training.

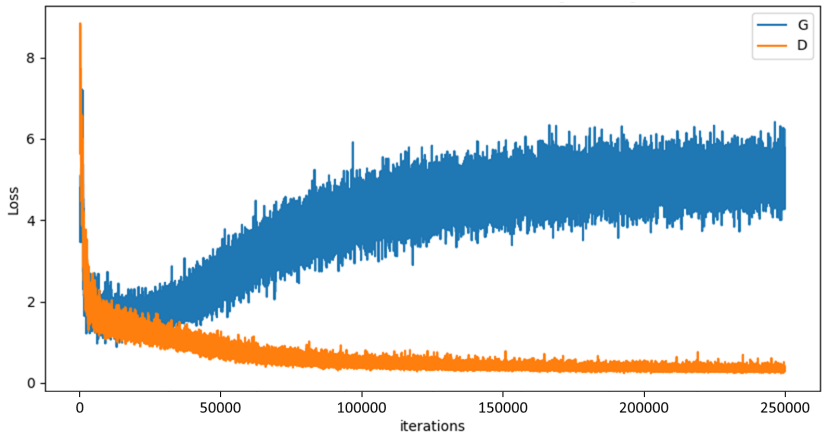


Figure 31: Losses $L^{(G)}$ of generator G and $L^{(D)}$ of discriminator D of AC-GAN over the training iterations.

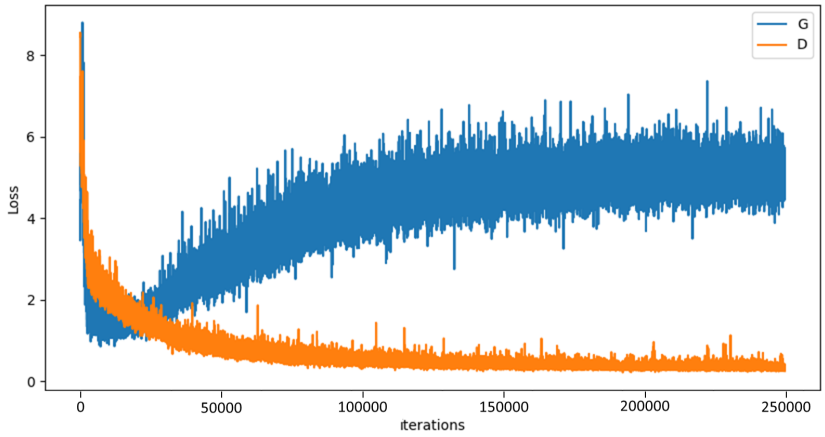


Figure 32: Losses $L^{(G)}$ of generator G and $L^{(D)}$ of discriminator D of HAC-GAN over the training iterations.

F Visualization Output Generator (H)AC-GAN during Training

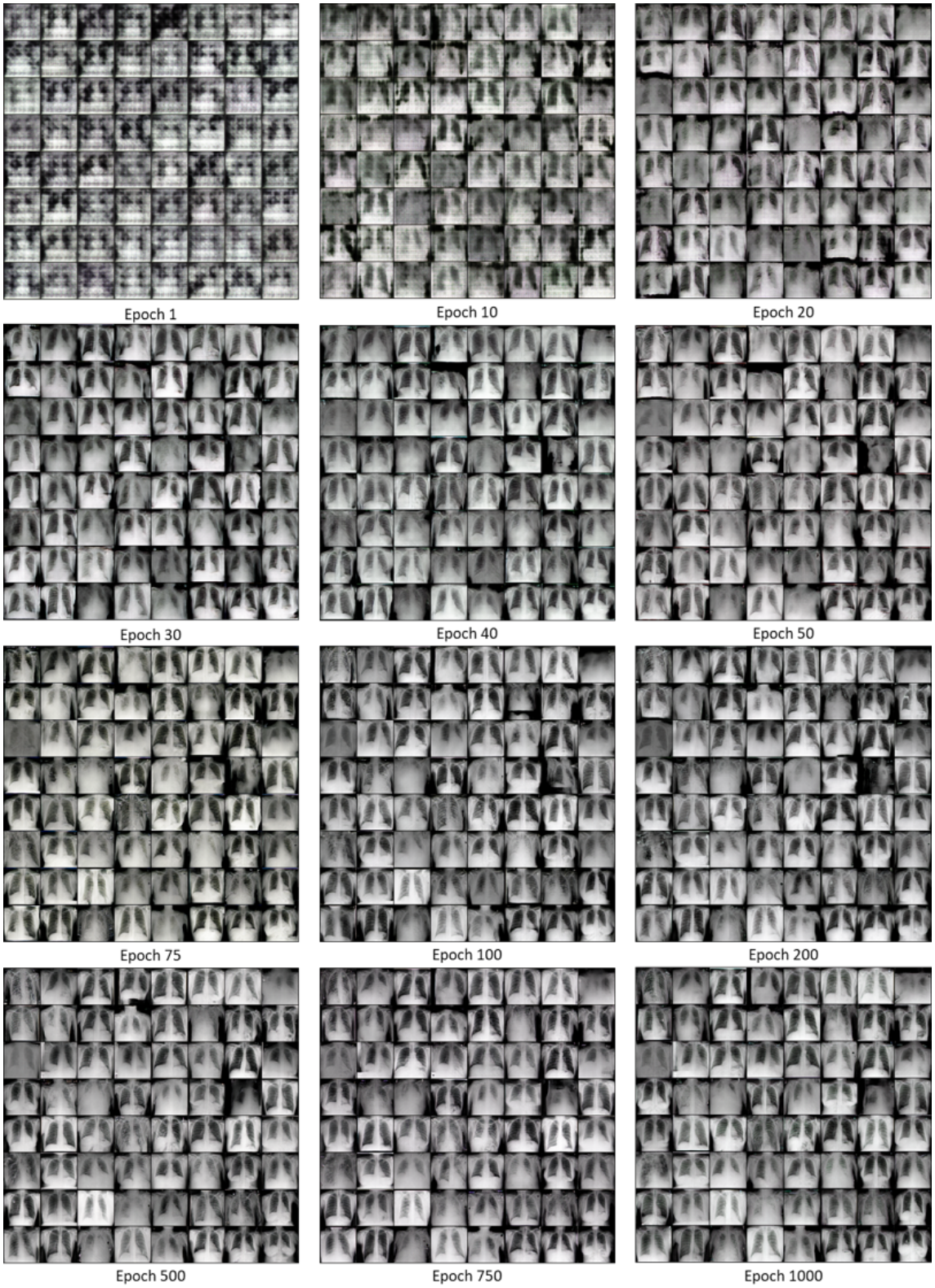


Figure 33: Visualization of the training process of images generated by the generator G in AC-GAN. Over the epochs, the generated images clearly become more detailed and distinctive, indicating successful training.

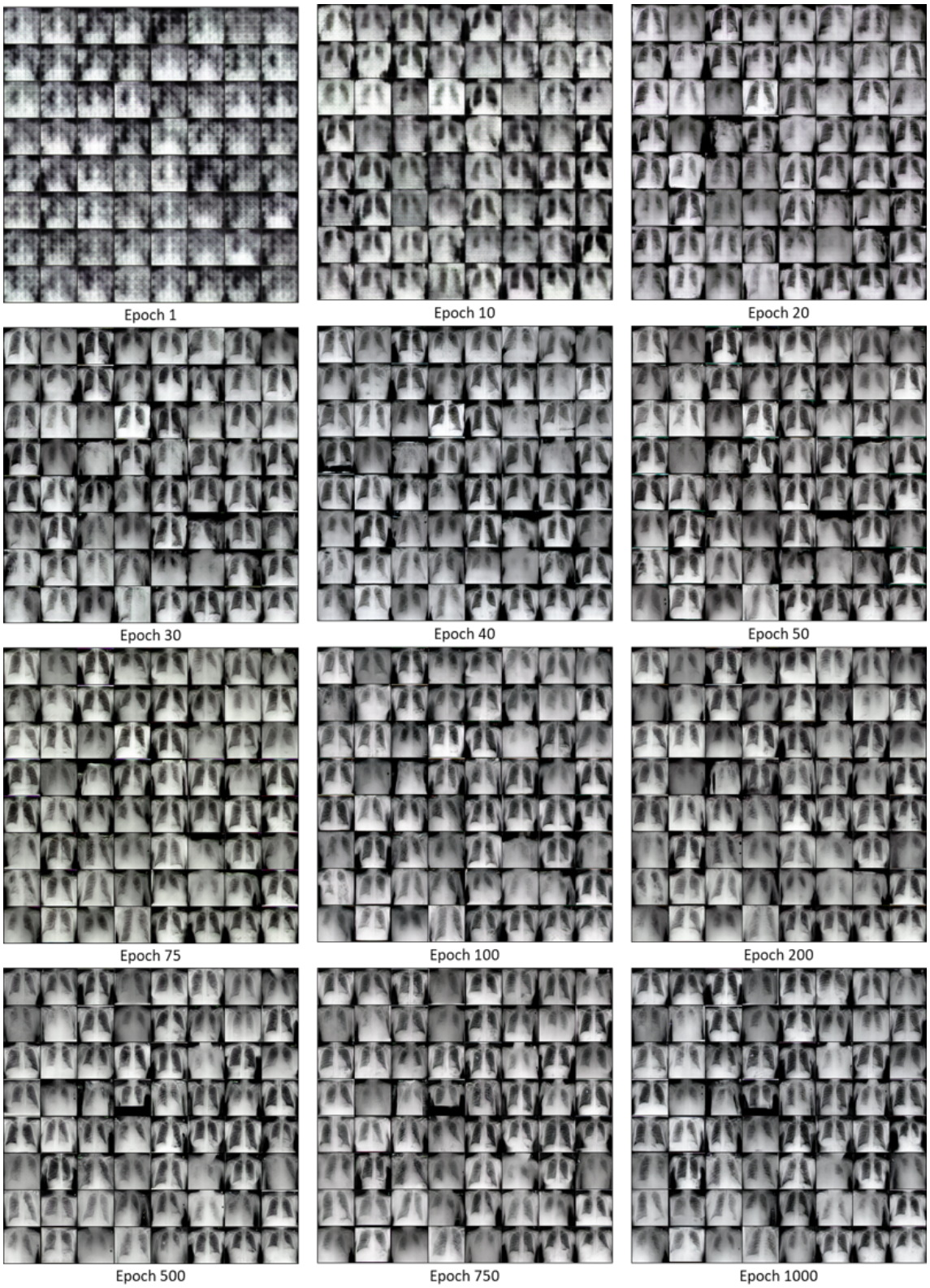
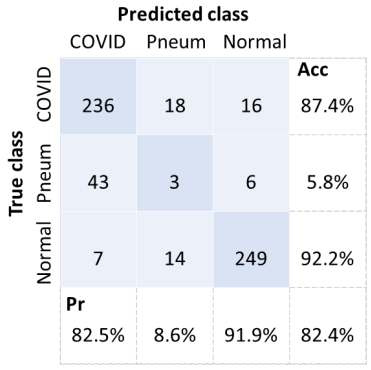
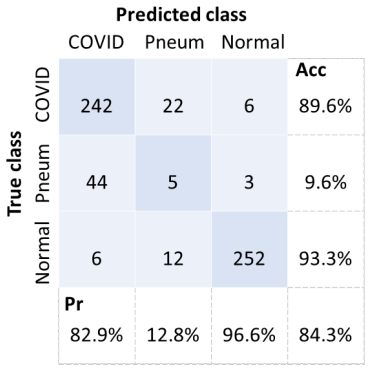


Figure 34: Visualization of the training process of images generated by the generator G in HAC-GAN. Over the epochs, the generated images clearly become more detailed and distinctive, indicating successful training.

G Confusion Matrices DTL Classification Models

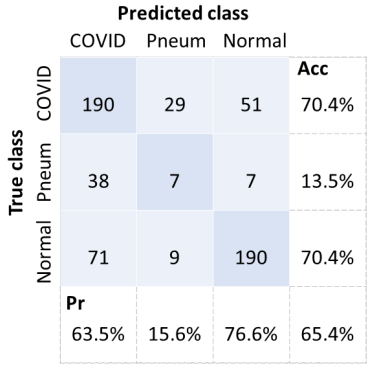


(a) Original dataset

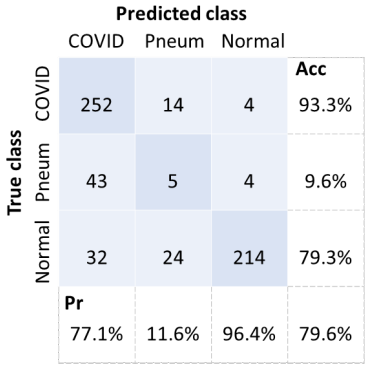


(b) Augmented dataset by HAC-GAN

Figure 35: Confusion matrices for COVID-19 detection using ResNet-18 trained on the original dataset (a) and the augmented dataset by HAC-GAN (b). The accuracy/sensitivity (Acc) and precision (Pr) of each class are also given.

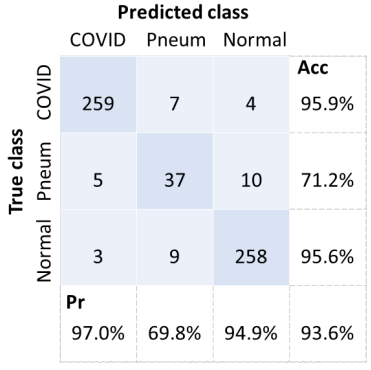


(a) Original dataset

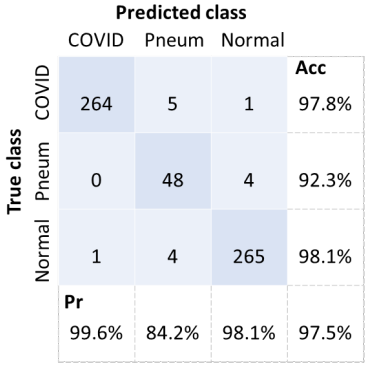


(b) Augmented dataset by HAC-GAN

Figure 36: Confusion matrices for COVID-19 detection using AlexNet trained on the original dataset (a) and the augmented dataset by HAC-GAN (b). The accuracy/sensitivity (Acc) and precision (Pr) of each class are also given.

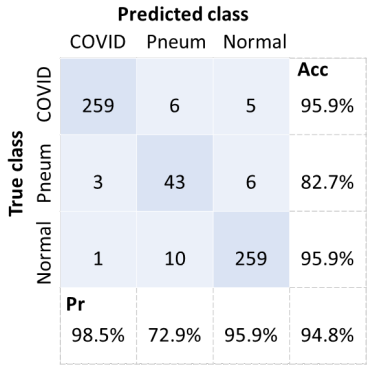


(a) Original dataset

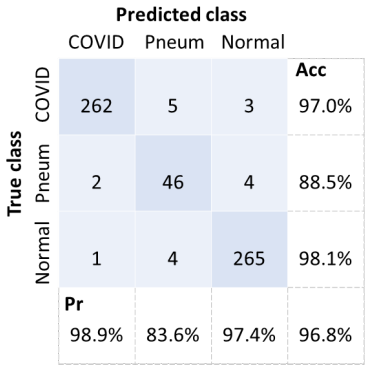


(b) Augmented dataset by HAC-GAN

Figure 37: Confusion matrices for COVID-19 detection using SqueezeNet trained on the original dataset (a) and the augmented dataset by HAC-GAN (b) with the modified train/test split (which represents an analysis on local data). The accuracy/sensitivity (Acc) and precision (Pr) of each class are also given.



(a) Original dataset



(b) Augmented dataset by HAC-GAN

Figure 38: Confusion matrices for COVID-19 detection using AlexNet trained on the original dataset (a) and the augmented dataset by HAC-GAN (b) with the modified train/test split (which represents an analysis on local data). The accuracy/sensitivity (Acc) and precision (Pr) of each class are also given.

H Gradient-weighted Class Activation Mapping (Grad-CAM)

This section elaborates on Grad-CAM, which is a widely used method that generates visualizations to help understand how CNNs make class detection decisions. Grad-CAM exploits spatial information preserved through the convolutional layers to visualize the image's regions of influence on the CNN's classification decision (Selvaraju et al., 2017). Grad-CAM is class-specific as it can produce separate visualizations for any class in the dataset. First, Grad-CAM takes an image as input and forwards it through the CNN, which is cut off at the final convolutional layer. The feature maps produced by the last convolutional layer are then used to create an activation map as Selvaraju et al. (2017) state that "we can expect the last convolutional layers to have the best compromise between high-level semantics and detailed spatial information". To produce an activation map, the input image is then forwarded through the full CNN to obtain output predictions and the target class score y_c is calculated, which is the output of the CNN for class c. Next, the gradient of the score for a target class c with respect to each feature map A^k ($\frac{\partial y^c}{\partial A^k}$) of the final convolutional layer is calculated using backpropagation. Note that there are C feature maps $A_k \; \forall \; k = 1, ..., C$ each of height h and width w, where C is the number of channels in the final convolutional layer. The calculated gradients are global-average-pooled over the weight and height dimensions to obtain the following importance weights for feature map k for target class c

$$\alpha_k^c = \frac{1}{Z} \sum_{i=1}^{h} \sum_{j=1}^{w} \frac{\partial y_c}{\partial A_{ij}^k}, \tag{30}$$

where $Z = h * w$ and A_{ij}^k represents the value at height index i and weight index j in the feature map A^k (Selvaraju et al., 2017). The weight a_k^c captures the "importance" of feature map A^k for detecting target class c in the input image (Selvaraju et al., 2017). Lastly, a weighted combination of the feature maps is performed followed by a ReLU activation to obtain the activation map

$$L_{\text{Grad-CAM}}^c = \text{ReLU}\left(\sum_{k=1}^{C} \alpha_k^c A^k \right), \tag{31}$$

where C is the number of channels or number of feature maps in the final convolutional layer and α_k^c are the weights calculated by equation (30) (Selvaraju et al., 2017). The ReLU activation is used as we are only interested in the features that have a positive influence on the predictions of class c. Note that $L_{\text{Grad-CAM}}^c$ is of size $h \times w$, which is the size of feature maps generated by the

final convolutional layer. Lastly, the activation map $L^c_{\text{Grad-CAM}}$ is therefore rescaled/up-sampled to be of the same size as the original image.

We base our implementation of Grad-CAM on the implementation `grad-cam-pytorch` adapted from `https://github.com/kazuto1011/grad-cam-pytorch` in the `PyTorch` library (Nakashima, 2020). Within the implementation of Nakashima (2020), we use the fixed weights of our trained DTL models to compute the feature maps A_k and class scores y_c for the input images which are then used to compute the activation maps $L^c_{\text{Grad-CAM}}$.