Blood Cell Data Augmentation using Deep Learning Methods

Utökning av blodcellsdata med hjälp av deep learning

Oskar Klang
Martin Carlberg

Supervisors:
Martin Almers, CellaVision
Anders Heyden, Lund University
Examinator:
Niels Christian Overgaard, Lund University

Abstract

In this thesis we aim to improve classification performance on blood cell images by using deep learning techniques to augment data. The thesis was conducted at CellaVision, a company providing digital solutions for medical microscopy in the field of hematology. The goal of CellaVision's technology is to replace manual microscopes used for cell differentials in blood tests with digital microscopes that perform cell differentials automatically. Classyfying white blood cells is an important part of this technology and is achieved by using an artificial neural network. This classifier network requires a great amount of training data in order to perform well.

With the objective to improve the performance of the classifier, we augment training data consisting of blood cell images by generating synthetic data using a Generative Adversarial Network (GAN). Our goal is to generate images with close to equal quality of the real images and to use the generated images for classifier improvement. The results show that the GAN is able to generate images that, apart from some small artefacts, very much resemble the real images, so much that a medical technologist struggled to differentiate them from real images.

In order to generate class specific blood cell images, we implement a version of the Auxiliary Classifier GAN (AC-GAN), where we use a pre-trained generator and discriminator from a GAN able to produce high quality images. The generator and discriminator are freezed and connected to fully connected layers to be trained. By augmenting the training data with the generated images from this AC-GAN, classifier performance improved for the majority of classes resulting in an increased F1-score. This leads us to believe that augmenting blood cell image data by using synthetic images is a viable method for classifier performance improvement.

Acknowledgements

We would like to thank our LTH supervisor Anders Heyden for useful comments, suggestions and discussions throughout the course of this project and our supervisor at CellaVision Martin Almers for proposing the project and playing a key part in the creative process as well as in making us feel welcome at the company. Also, from Cellavision, we would like to thank Jesper Jönsson who has shared his knowledge of GANs and helped us overcome problems we had. We would also like to thank our master's thesis colleagues Oscar Odestål and Anna Palmqvist Sjövall. Discussing overlapping parts of our theses with them has been of much help. Together with Oscar and Anna, we would also like to thank Hugo Sellerberg and Nellie Carleke, who have been contributing to a stimulating social atmosphere at the office during our work on this thesis. Finally, we would like to thank all of CellaVision for the opportunity of doing our thesis work with them.

Contents

1 Introduction

1.1 Motivation

Blood tests are widely used in health care in order to detect diseases or other health state anomalies. Performing cell differentials by analyzing blood cell morphology, i.e. the structure, shape and size of a cell, is crucial in detecting anomalies in a test. This analysis has traditionally been performed by using manual microscopes, a time-consuming procedure requiring access to experienced personnel and taking them away from more important tasks.

CellaVision's technology aims to replace manual microscopes by providing digital hematology systems performing cell differentials automatically by using image analysis technology and artificial neural networks. One task of the digital imaging system provided by CellaVision is to pre-classify white blood cells into biologically defined classes. These will then be displayed to a medical technologist for review and verification, making this otherwise tedious and time consuming work much easier and faster. The classification is performed by an artificial neural network that necessitates a great amount of training data in order to yield good results. Specifically, there is a shortage of data for some blood cell classes, making the performance of the classifier weaker on these particular classes. Furthermore, lack of data increases overfitting of the model, i.e. the model fits too close to the training data and fails to generalize and predict new data. Shortage of training data can often be compensated for by using different methods of augmentation to increase the size of the dataset at hand. Common augmentation methods for image data are translation, rotation, flipping and scaling. However, the variability introduced by these augmentation methods is limited and an alternative to these classical augmentation methods would be to generate *synthetic data* and add this to the given dataset in an attempt to increase diversity and variability.

A method for generating synthetic data is to use a deep generative model such as the Generative Adversarial Network (GAN) introduced by Goodfellow et al. in 2014 [1]. The principal idea of this method is the following: Given a dataset, the GAN learns to produce data with similar statistical properties as the source data by pitting a generative network and a discriminative network against each other. The generator tries to fool the discriminator by producing data as similar as possible to the source while the task of the discriminator is to determine whether data is synthetic or real, i.e. produced by the generator or given by the original dataset. Both the generator and the discriminator improve in the training stage by giving each other cues. If the discriminator is able to discriminate between real and synthetic data, the generator will change its method in an attempt to decrease the discriminator's performance. This tug-of-war continues until the discriminator can no longer distinguish between real and synthetic data. The trained generator can then be used to generate synthetic data which can be used for augmentation.

1.2 Aim of the Thesis

The aim of this thesis is to evaluate how well GANs and variants of these can be used to generate blood cell images and if classifier performance can be improved by training on datasets augmented with generated images.

Questions we aim to answer are:

- Starting from noise, is it possible to generate blood cell images such that an expert in the field cannot differentiate them from real images?
- Is it possible to improve the performance of a classifier by training it on data augmented with generated images?

2 Related work

Previous work has been done where GANs have been used to generate data for data augmentation in order to improve classifier performance. Described below are two such examples.

2.1 Synthetic Data Augmentation using GAN for Improved Liver Lesion Classification

An example of related work is the 2018 paper by Maayan Frid-Adar et al. on data augmentation using GAN for improved liver lesion classification [2]. In their work, they augmented a limited dataset of computed tomography (CT) images of 182 liver lesions consisting of three classes: Cysts, metastases and hemangiomas. They managed to increase classification sensitivity from 78.6% using classical data augmentation to 85.7% by adding synthetic (generated) data while the corresponding specificity increased from 88.4% to 92.4%. Furthermore, they let two radiologists examine and classify some generated liver lesion images together with real liver lesion images to see whether they could discriminate between the two. Expert 1 had a classification accuracy of 78% and 77.5% on the real and the generated images respectively, while the corresponding results for Expert 2 were 69.2% and 69.2%. These results imply that they were successful in generating synthetic data having the appearance of real data.

2.2 Augmenting Training Data using GAN to Improve Segmentation of Brain Images

Another example of related work is the 2018 paper by Christopher Bowles et al. on data augmentation using GAN for segmentation of two datasets of brain images: CT Cerebrospinal Fluid (CSF) and Fluid-attenuated inversion recovery (FLAIR) Magnetic Resonance (MR) [3]. The CT brain images consist of three different classes: Cortical CSF, brain stem CSF and ventricular CSF. The FLAIR brain images belong to a single class. They performed and evaluated the segmentation both with a UNet and a UResNet. Their results show a significant improvement of segmentation when augmenting their datasets with a combination of GAN generated data and rotated data. This implies that a combination of classical augmentation methods and synthetic augmentation may be a reasonable approach when augmenting data for segmentation and classification improvement.

3 Background

3.1 Blood Cells

Blood consists of red blood cells (RBCs), white blood cells (WBCs), plasma and platelets [4]. One of the main goals of this thesis is to improve classification of WBCs. There are three types of WBCs: Lymphocytes, monocytes, and granulocytes, where granulocytes can be divided into neutrophils, eosinophils, and basophils. The WBCs consist of a nucleus and cytoplasm surrounding this nucleus while the red blood cells lack nuclei, see Figure 1. The data from CellaVision is further divided into 19 different classes of WBCs.

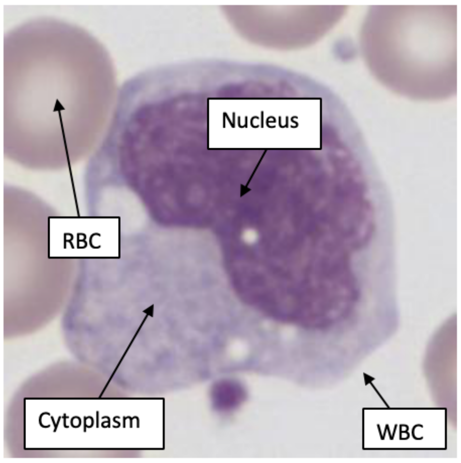


Figure 1: Image of a WBC with surrounding RBCs. The WBC consists of a nucleus located in cytoplasm while the RBCs lack nuclei.

3.2 Machine Learning

The discipline of machine learning aims to make computer programs learn to perform tasks without explicit instructions on how this learning should be achieved. The idea is that a machine learning algorithm should be able to improve its performance based on experience, instead of explicit instructions. Machine learning algorithms learn from data using statistical models and optimization, more specifically given a dataset, $\boldsymbol{x}$, it aims to learn the probability distribution, $p(\boldsymbol{x})$, which generated the dataset of interest. Some examples of tasks that machine learning is able to solve are:

- Classification: Assigning a category to some given input, i.e. predicting which category the data belongs to with respect to a number of categories.
- Regression: Predicting a numerical value given some input.
- Synthesis: Generating new examples with features similar to the training data.

Machine learning algorithms can be divided into two main categories, *unsupervised learning algorithms* and *supervised learning algorithms*. Unsupervised learning algorithms attempt to learn the structure of the data of interest, $\boldsymbol{x}$, observing its features in order to find the underlying probability distribution, $p(\boldsymbol{x})$. Examples of tasks where unsupervised learning is normally applied are clustering, density estimation and synthesis. In supervised learning, the input, $\boldsymbol{x}$, is associated with a label or a target, $\boldsymbol{y}$, and the objective is to predict $\boldsymbol{y}$ given $\boldsymbol{x}$, e.g. by estimating $p(\boldsymbol{y}|\boldsymbol{x})$. Classification and regression are typically considered to be supervised learning problems [5].

In general, a machine learning algorithm combines a model, a cost function and an optimization procedure and applies them on a dataset. A model could be $p(\boldsymbol{y}|\boldsymbol{x}, \boldsymbol{\theta})$, where $\boldsymbol{\theta}$ are parameters representing a range of distributions. The goal is then to find the $\boldsymbol{\theta}$ corresponding to the best representation of the true distribution of the data, p_{data}. Finding the best representation is an optimization problem which can be solved by defining and minimizing a cost function. Typically, cost functions are based on the principles of maximum likelihood or mean squared error. One commonly used cost function is the cross-entropy between the training data and the model distribution, defined as

$$J(\boldsymbol{\theta}) = -\mathbb{E}_{\boldsymbol{x},\boldsymbol{y}\sim\hat{p}_{data}} \log p_{model}(\boldsymbol{y}|\boldsymbol{x}), \tag{1}$$

where $\hat{p}_{data}$ is the empirical distribution of the dataset, i.e. the distribution observed in the dataset (and is in general not equivalent to the true data generating distribution, p_{data}) and p_{model} is the distribution approximated by the model. The minimizing can be performed by an optimization algorithm such as stochastic gradient descent.

3.3 Stochastic Gradient Descent

Optimization algorithms aim to minimize or maximize an objective function, which in the case of a minimizing problem corresponds to a cost function. If we denote the cost function as $J(\boldsymbol{\theta})$, then the goal of the minimizing procedure is to make changes in the parameter $\boldsymbol{\theta}$ such that the value of J reduces and to ultimately find the $\boldsymbol{\theta}$ for which J reaches its minimum value. Gradient descent methods make use of the fact that changes of $\boldsymbol{\theta}$ in the direction of the opposite sign of the gradient leads to a decrease in the cost function. The cost function can be decomposed as a sum of loss functions, each corresponding to the loss function for a given example in the training examples. Hence, Equation (1) can be formulated as

$$J(\boldsymbol{\theta}) = \mathbb{E}_{\boldsymbol{x},\boldsymbol{y}\sim\hat{p}_{data}} L(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{\theta}) = \frac{1}{m}\sum_{i=1}^{m} L(\boldsymbol{x}^{(i)}, \boldsymbol{y}^{(i)}, \boldsymbol{\theta}), \tag{2}$$

where m is the number of data points and L is the loss for each example, i.e. $L(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{\theta}) = -\log p(\boldsymbol{y}|\boldsymbol{x}; \boldsymbol{\theta})$. Since the cost function is a function mapping multiple inputs to a scalar, i.e. $J : \mathbb{R}^n \to \mathbb{R}$, deciding which direction to move in corresponds to computing the gradient, $\boldsymbol{g}$, of the cost function with respect to the parameters $\boldsymbol{\theta}$:

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \frac{1}{m}\sum_{i=1}^{m} \nabla_{\boldsymbol{\theta}} L(\boldsymbol{x}^{(i)}, \boldsymbol{y}^{(i)}, \boldsymbol{\theta}). \tag{3}$$

Equation (3) implies that the gradient is an expectation and as such it can be approximated by using a small set of samples of the training set. This is the main idea of stochastic gradient descent. By sampling a minibatch, uniformly drawing m' examples from the training set and computing the expectation for these examples, we get an estimate of the expectation for the entire training set consisting of m examples, i.e. the estimated gradient becomes

$$\hat{\boldsymbol{g}} = \frac{1}{m'} \nabla_{\boldsymbol{\theta}} \sum_{i=1}^{m'} L(\boldsymbol{x}^{(i)}, \boldsymbol{y}^{(i)}, \boldsymbol{\theta}). \tag{4}$$

Moving in the direction of the opposite sign of the gradient then corresponds to

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha \hat{\boldsymbol{g}}, \tag{5}$$

where α is the learning rate, representing the step size of the algorithm.

3.4 Deep Learning

Deep learning models are a subset of machine learning models and are based on artificial neural networks. The aim of an artificial neural network is to approximate some function, f, given input, $\boldsymbol{x}$. The goal could for example be to classify data, assigning it some label, y.

For some functions this approximation could be done using linear models such as linear and logistic regression. However, due to their linearity, these models have the limitation that they cannot model the interaction between input variables. To overcome this limitation the linear model can be applied to a non-linear transformation of the input, $\boldsymbol{x}$, instead of being applied directly to the input itself. The aim of the deep learning model will then be to learn the non-linear transformation, $\phi(\boldsymbol{x})$. The model can be represented as $y = f(\boldsymbol{x}; \boldsymbol{\theta}, \boldsymbol{w}) = \phi(\boldsymbol{x}; \boldsymbol{\theta})^T \boldsymbol{w}$ where $\boldsymbol{\theta}$ are parameters associated with a range of function classes and $\boldsymbol{w}$ are parameters mapping the transformed input, $\phi(\boldsymbol{x})$, to the output, y. The deep learning model learns ϕ through the parameters $\boldsymbol{\theta}$ in order to find the best approximation of f [5].

Multilayer Perceptron

The most fundamental artificial neural network is the *feed-forward network*, also known as the *multilayer perceptron*. The name multilayer perceptron refers to its similarities to the *perceptron algorithm* [6]. The simple perceptron is a model applying a non-linear activation function, H, to a linear combination of weight parameters, $\boldsymbol{w}$, mapping the input to the output, and a non-linear transformation of the input, $\phi(\boldsymbol{x})$, i.e.

$$y(\boldsymbol{x}) = H(\phi(\boldsymbol{x})^T \boldsymbol{w}), \tag{6}$$

where the function H is the Heaviside step function, defined as

$$H(a) = \begin{cases} +1, & a \geq 0, \\ -1, & a < 0. \end{cases} \tag{7}$$

A visualization of the perceptron can be seen in Figure 2 below.

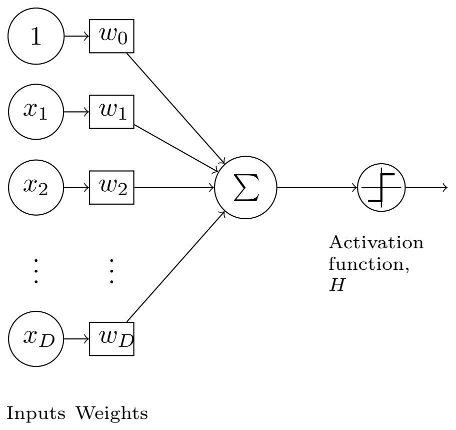


Figure 2: Visualization of a perceptron. Input, x_i, are linearly combined with weights, w_i, and then passed through an activation function, H.

Now, in the case of the multilayer perceptron (or feed-forward network) the idea is to perform a series of transformations on the input, resulting in a network containing several "layers", with each layer corresponding to some transformation(s). The most elemental such network consists of an input layer, a hidden layer and an output layer. Each input variable is represented by a node in the input layer. The hidden layer is then composed by constructing an affine transformation of the input $x_1, ..., x_D$ as

$$a_j = \sum_{i}^{D} w_{ji}^{(1)} x_i + w_{j0}^{(1)}, \tag{8}$$

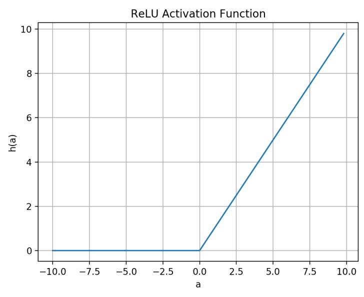
where $i = 1, ..., D$, $j = 1, ..., M$ is the number of *activations*, a_j, in the current layer of the network and w_{j0} are *biases.* These activations are then transformed by an *activation function*, h_1, as

$$z_j = h_1(a_j), \tag{9}$$

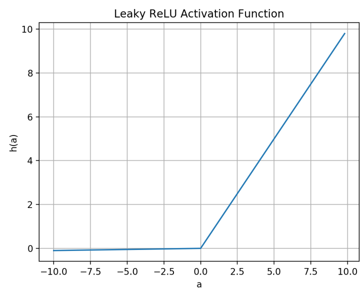
where z_j are called *hidden units* and the function, h_1, is non-linear. Some commonly used activation functions in deep learning are

- the rectified linear unit (ReLU), defined as $h(a) = \max(0, a)$,
- the leaky ReLU, defined as $h(a) = \begin{cases} a, \text{ if } a > 0 \\ 0.01a \text{ otherwise} \end{cases}$,
- the logistic sigmoid, defined as $h(a) = \frac{1}{1+\exp(-a)}$, often denoted as $\sigma(a) = h(a)$,
- the hyperbolic tangent (tanh) function, defined as $h(a) = \frac{\exp(2a)-1}{\exp(2a)+1}$.

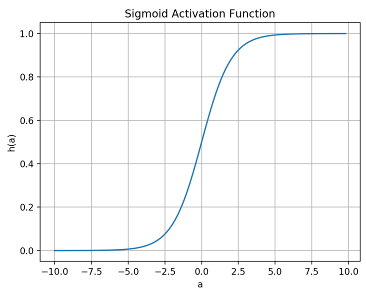
Visualizations of the aforementioned activation functions can be seen in Figure 3 below.



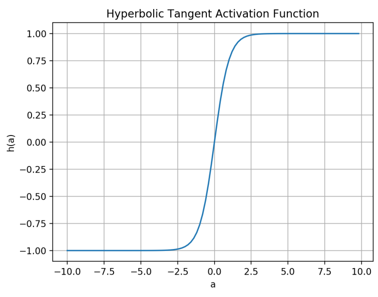
(a) ReLU activation function.



(b) Leaky ReLU activation function. Note the very slight slope for $a < 0$.



(c) Logistic sigmoid activation function.



(d) Hyperbolic tangent activation function.

Figure 3: Commonly used activation functions include the ReLU function, the leaky ReLU function, the logistic sigmoid function and the hyperbolic tangent function (tanh).

In order to compute the *output unit activations*, an affine transformation of the hidden units, z_j, is performed to yield the second layer of the network as

$$a_k = \sum_{j}^{M} w_{kj}^{(2)} z_j + w_{k0}^{(2)}, \tag{10}$$

where $k = 1, ..., K$ and K is the number of outputs. Transforming these activations with an output unit activation function, h_2, results in outputs, y_k, corresponding to the final layer of the network. The transformations performed in the network can then be summarized as

$$y_k(\boldsymbol{x}, \boldsymbol{w}) = h_2(\sum_{j}^{M} w_{kj}^{(2)} h_1(\sum_{i}^{D} w_{ji}^{(1)} x_i + w_{j0}^{(1)}) + w_{k0}^{(2)}). \tag{11}$$

The multilayer perceptron resembles the perceptron in that it consists of two transformations which are very similar to the transformation performed by the perceptron. The term *deep* in deep learning refers to the network having a great number of hidden layers as the ones in the multilayer perceptron. A visualization of the described network can be seen in Figure 4 below, where nodes correspond to units and edges correspond to weight parameters. For simplicity, the biases have been excluded in the illustration.

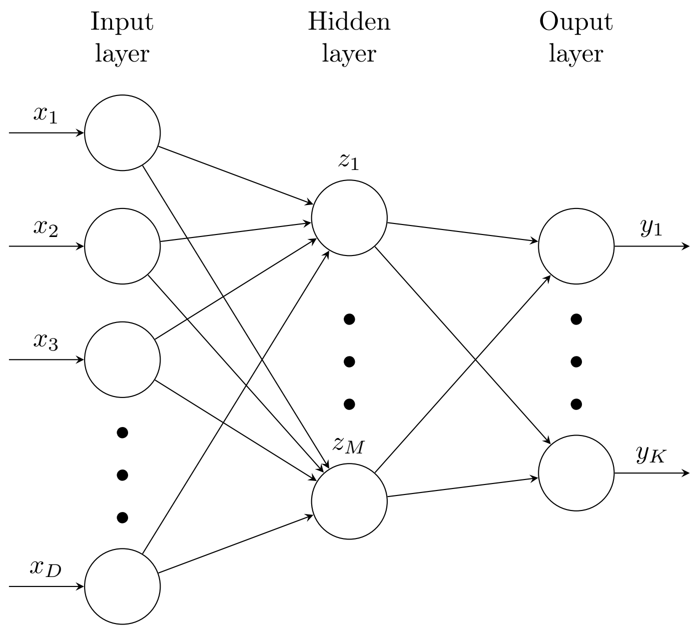


Figure 4: Visualization of a feed-forward network with one hidden layer. The nodes correspond to units and the edges correspond to weight parameters. The biases have been excluded for simplicity.

3.5 Backpropagation in Neural Networks

In order for the feed-forward network to learn how to approximate the function of interest, f, a cost function is to be minimized as described in Section 3.2. In Section 3.3 we saw that minimization of cost functions can be performed using an optimization method such as stochastic gradient descent. However, numerically evaluating the necessary gradients is computationally expensive. The method of *backpropagation* supplies an efficient way of computing the gradient of the cost function with respect to the weights [6]. The name backpropagation alludes to the fact that information is flowing backwards through the network in order to compute the gradient. In this section backpropagation is described in the context of a general feed-forward network.

The goal is to minimize a cost function, J. As seen in Section 3.3, the cost function can be decomposed as a sum of cost functions, one for each input-output pair, i.e.

$$J(\boldsymbol{w}) = \sum_{n=1}^{N} J_n(\boldsymbol{w}), \tag{12}$$

where N is the number of data points. Hence, the gradient of the cost function can be computed for each input-output pair and then we can sum over all the terms in order to get the total gradient of the cost function. The cost function can be in many different forms but to make the backpropagation procedure easier to understand, we will use the squared error function

$$J_n(\boldsymbol{w}) = \frac{1}{2}\sum_k (y_{nk} - t_{nk})^2, \tag{13}$$

where k is the index of the unit, y_{nk} are output units and t_{nk} are target variables.

Each activation in the network can be expressed as a linear combination of its input as

$$a_j = \sum_i w_{ji} z_i, \tag{14}$$

where the biases w_{j0} have been included in the sum by adding an additional unit, z_0, with value one. The activations are transformed by applying a non-linear activation function yielding the hidden units, z_j, as

$$z_j = h(a_j). \tag{15}$$

Now, the aim is to compute the derivative of the cost function, J_n, with respect to a weight, w_{ji}. The cost function, J_n, depends implicitly on the weight, w_{ji}, through the activation, a_j. Hence, applying the chain rule on J_n yields

$$\frac{\partial J_n}{\partial w_{ji}} = \frac{\partial J_n}{\partial a_j}\frac{\partial a_j}{\partial w_{ji}}. \tag{16}$$

Computing $\frac{\partial a_j}{\partial w_{ji}}$ using expression (14) results in

$$\frac{\partial a_j}{\partial w_{ji}} = z_i. \tag{17}$$

The first term in Equation (16), $\frac{\partial J_n}{\partial a_j}$, is called the *error* [6] and we denote this with δ_j. Equation (16) can then be formulated as

$$\frac{\partial J_n}{\partial w_{ji}} = \delta_j z_i, \tag{18}$$

and since the values of the units, z_i, are known, computing the derivative of the cost function with respect to the weights reduces to computing δ_j for every output and hidden unit. For the output units, the errors take the form

$$\delta_k = y_k - t_k, \tag{19}$$

and for the hidden units the errors take the form

$$\delta_j = \frac{\partial J_n}{\partial a_j} = \sum_k \frac{\partial J_n}{\partial a_k}\frac{\partial a_k}{\partial a_j}, \tag{20}$$

where the chain rule has been applied. The first factor in the sum is

$$\delta_k = \frac{\partial J_n}{\partial a_k}, \tag{21}$$

while the second factor is, making use of Equations (14) and (15),

$$\frac{\partial a_k}{\partial a_j} = h'(a_j) w_{kj}. \tag{22}$$

Substituting (21) and (22) into (20) gives us the backpropagation formula for hidden units as

$$\delta_j = h'(a_j) \sum_k w_{kj} \delta_k. \tag{23}$$

The formula shows how information flows backwards in the network, due to δ_k being higher up in the network than δ_j for every k. The errors for the output units are known through Equation (19) which makes it possible to compute the errors for the hidden units recursively by using (23). These results are then used in Equation (18) in order to update the weights in the network, using an optimization method such as stochastic gradient descent.

3.6 Capacity and Generalization

When developing a machine learning model, the model is often trained on a dataset on which it aims to minimize a *training error*. It is important that the model can *generalize* and perform well on data that is not part of the training data, therefore the model is evaluated on a test set by computing a *test error*, also known as a *generalization error* [5].

A model's *capacity* determines how well the model fits to a broad range of functions. If a model consists of a great amount of parameters, i.e. having high capacity, it is likely to fit well to the training data. However, if too many parameters are involved, the model might fit *too well* to the training data and fail to generalize to new data. This phenomenon is known as *overfitting* [5]. Thus, it is important to find a balance of the model's capacity. The model has to be complex enough to fit to the training data and yield a small training error while at the same time not being overly complex, such that it overfits. See Figure 5 for an example of this. Techniques that aim to reduce generalization error are in general called *regularization* techniques. One such technique is *dropout*, described in Section 3.7.

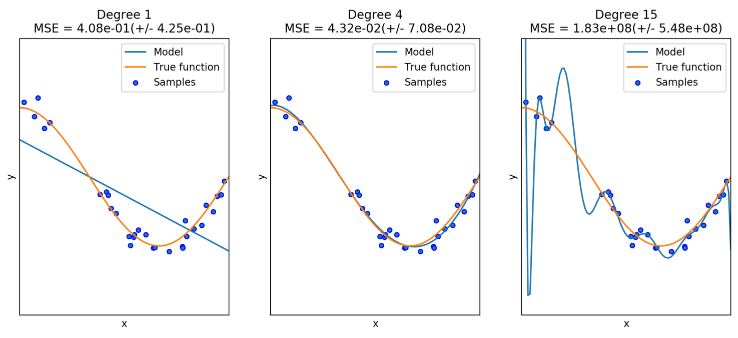


Figure 5: In the plot to the left, a polynomial with degree 1 is fitted to the data samples, yielding an example of underfitting. In the center plot a polynomial with degree 4 is fitted to the samples, yielding a good approximation of the function. The plot to the right shows an example of overfitting, where a polynomial with degree 15 has been used [7].

3.7 Dropout

One straightforward way to reduce generalization error is to train several different models and evaluate each of the models on the test data. Different models might give errors on different parts of the data so averaging over this *model ensemble* will yield a more robust result. However, this procedure is computationally expensive, therefore dropout is used as an inexpensive alternative to this method [5].

Dropout removes non-output units and their connections with a certain probability and trains the remaining network, which can now be seen as a subnetwork of the original network. This is done for all possible such subnetworks, i.e. dropout trains the ensemble of all subnetworks. In practice, for each batch of data, a subnetwork is constructed by multiplying each unit in the network by 1 with a probability p and by zero with a probability $1-p$. The resulting network is then trained on the batch as per the usual procedure: forward propagation, backpropagation and weight updating. Dropout decreases the generalization error and the reason why it is computationally feasible is due to the parameter sharing between the subnetworks. If the original network consists of n units, the maximum amount of subnetworks will be 2^n while the number of parameters will be $\mathcal{O}(n^2)$ or less due to the parameter sharing [8]. For an illustration of dropout, see Figure 6.

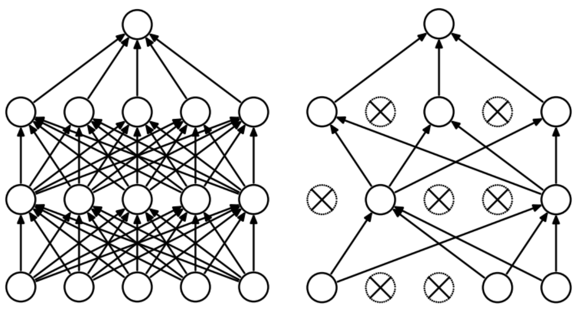

(a) Network without dropout. (b) Network with dropout.

Figure 6: Illustration of (a) a network without dropout, where every node in one layer is connected to every node in the next layer and (b) a network where dropout has been applied, resulting in a subnetwork of the original network [8].

3.8 Batch Normalization

Batch Normalization (BN) was introduced by Sergey Ioffe and Christian Szegedy in 2015 [9] as a way of speeding up and stabilizing the training of neural netrworks. The idea was to tackle the problem of the activations flowing from one layer to the next having very different distribution from iteration to iteration due to the parameters of the first layer having been updated by the optimizing scheme. The difference in distribution of the activations flowing in to a layer makes it difficult for the optimizer to find a good parameter-configuration and creates stability issues.

The BN-algorithm is as following: Given a batch of activations $\mathcal{B} = \{x_i\}_{i=1,\dots,m}$ of size m, the mean and variance is calculated:

$$* \, \mu_{\mathcal{B}} = \frac{1}{m}\sum_{i=1}^{m} x_i \tag{24}$$

$$\sigma^2_{\mathcal{B}} = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2. \tag{25}$$

The whole batch is then normalized by subtracting the mean and dividing by the square root of the variance. A small value ϵ is added to the variance for numerical stability. The normalized values are then formed as

$$\hat{x}_i = \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}}. \tag{26}$$

To avoid losing the representation of power in the network, the normalized values are then scaled and shifted using the learnable parameters γ and β to form the output of the BN-algorithm

$$y_i = \gamma \hat{x}_i + \beta. \tag{27}$$

Since the activations are usually multidimensional, the normalization is done component-wise and the model has two trainable parameters $\gamma^{(k)}$ and $\beta^{(k)}$ for each dimension k in the activations entering the BN-layer. The transformation entailed by the BN-algorithm is continous and the learnable parameters can therefore be learned by stochastic gradient descent.

3.9 Convolutional Neural Networks

Convolutional neural networks (CNN) are a form of neural networks that use the mathematical operation convolution in one or more of their layers. CNNs are characterized by being invariant to certain transformations such as translation. This characteristic makes them suitable for processing image data.

In the two-dimensional case, such as for an image, the discrete convolution operation is defined as

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n), \tag{28}$$

where I is the input image and K is a weighting function called a *kernel*. In a CNN, the convolutional layer corresponds to a kernel sliding over the input image and calculating the sum of the products of an elementwise multiplication of the kernel and the region the kernel is covering (receptive field), see Figure 7. The *stride* specifies the step size of the kernel, for example a stride of one traverses every position in the image while a stride of two skips every other position. The output of these multiplications builds up a *feature map*. All of the units belonging to one such feature map share the same weights, which is one of the benefits of using CNNs, since it reduces the number of parameters that need to be storaged in memory [5]. The feature map is of a lower dimension than the input image and holds information about lower level features in the image, such as edges, curves and colors. The feature map has a depth that is determined by how many kernels that are used, where one kernel could for example detect edges while another one detects curves.

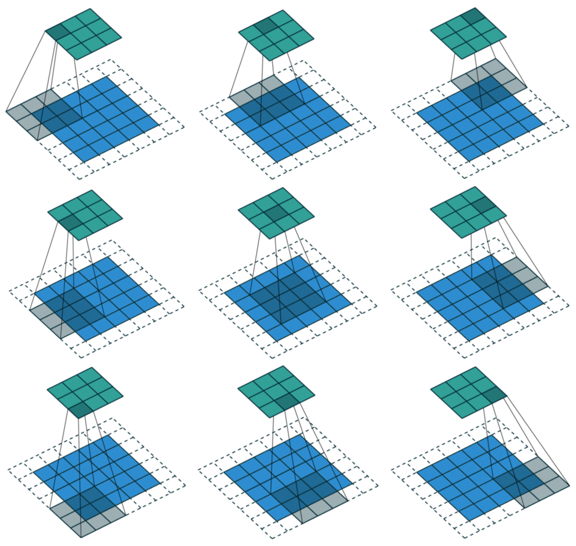


Figure 7: Visualization of how the kernel traverses the image to perform the elementwise multiplication [10]. Here, the convolution is performed with a stride of two, i.e. the kernel skips every other element of the image.

When the entire image has been traversed by the kernel and the feature map is complete, the output is run through a non-linear activation function which is often followed by a *pooling* layer. In the pooling layer the output of several units are combined, e.g. by computing the average (*average pooling*) or choosing the maximum value (*max pooling*) of the units, and is replaced by this value, thus reducing the dimension of the data while also decreasing the sensitivity to small translations of the input. When referring to a CNN, it is often in the context of several convolutional layers, pooling layers and non-linear activation functions applied on top of each other. The term CNN then refers to the entire structure and not a single convolutional layer. The input to one layer will be the feature map from the previous layer and the output will now be activations of higher level features, such as combinations of the lower level features. This way, and by applying backpropagation to update the weights, the network learns how to represent increasingly complex features.

In the multilayer perceptron described in Section 3.4 every input unit interacts with every output unit, i.e. it is a *fully connected* network. CNNs differ from this approach in that each unit only interacts with inputs corresponding to a restricted region of the image, meaning they are *sparse* networks [5]. A fully connected layer can be applied in the end of the convolutional neural network in order to perform e.g. classification. An illustration of a convolutional network including convolutional layers, ReLU activations, pooling and a fully connected layer with softmax activation (which is a multi-class variant of the logistic sigmoid function) for classification can be seen in Figure 8.

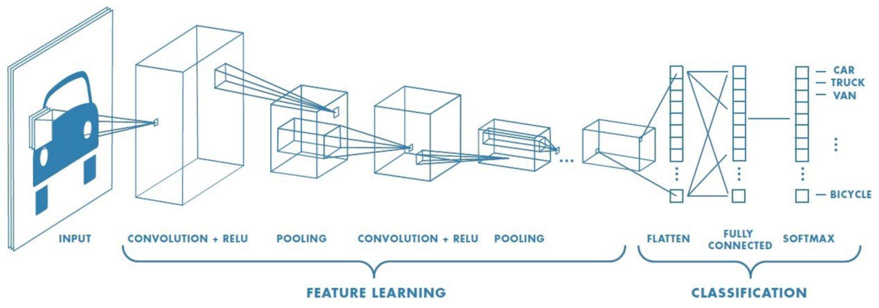


Figure 8: Illustration of a convolutional network taking an image as input, e.g. an image of a car, and assigning it a class label (car, truck, van, etc.) [10].

3.10 Transposed Convolutions

In the previous section it is indicated that convolutions can be used to downsample data. This raises the question if the convolution operation can be used to upsample data. The answer is yes and this particular use of the convolution operation is known as a *transposed convolution* [11]. The upsampling is done by sliding the kernel over not the original image, but an enlarged image consisting of the original pixles with zero-padding between them in each direction (up-down, left-right, diagonals) as well as outside the images. When the kernel slides over the padded image, the output is larger than the input, i.e an upsampling of the input. This upsampling is learned by the network by updating the kernel weights according to the information supplied by the loss function through backpropagation.

3.11 Generative Adversarial Networks (GAN)

Generative Adversarial Networks (GAN) were introduced by Goodfellow et al. [1] as a way to train generative deep learning models. The idea is to approximate the distribution of the data by using two neural networks and pit them against each other as *adversaries*. The first network, called the *generator*, is trained to generate new data samples. As originally proposed in the paper by Goodfellow et al., the generated data samples are created by sampling from a given probability distribution, called the *latent space* and feeding the sampled *latent vector* through the generating network. The second network, called the *discriminator*, is trained to assign a probability to whether a data sample is real or fake (i.e. generated).

Training a GAN can then be summarised as follows: Draw a number of real data samples and let the generator generate an equal amount. Both the real data and the generated data are then fed to the discriminator which outputs probabilities of each data sample being real. The networks are then updated according to the output of the discriminator. The generator is updated as to maximize the amount of generated images being assigned a high probability of being real. The discriminator is updated as to minimize the amount of generated samples being assigned high realness probabilities while maximizing the amount of real samples given high probabilities, thus *discriminating* between real samples and fake samples. The opposite goals of the two networks constitutes the *adversarial* nature of the method. In mathematical terms, if we let G be a function corresponding to the transformation performed by the generator and D be the corresponding for the discriminator, the networks can be said to compete in a minimax game with value function V. The game can then be expressed as

$$\min_G \max_D V(G, D) = \mathbb{E}_{\boldsymbol{x} \sim p_{\text{data}}(\boldsymbol{x})}[\log D(\boldsymbol{x})] + \mathbb{E}_{\boldsymbol{z} \sim p_{\boldsymbol{z}}}[\log(1 - D(G(\boldsymbol{z})))], \quad (29)$$

where $p_{\text{data}}(\boldsymbol{x})$ is the distribution of the real data and $p_{\boldsymbol{z}}(\boldsymbol{z})$ is the latent space. In practice, the generator is trained to minimize the loss function

$$L_G = \mathbb{E}_{z \sim p_{\boldsymbol{z}}(\boldsymbol{z})}[\log(1 - D(G(\boldsymbol{z})))], \quad (30)$$

while the discriminator is trained to minimize the loss function

$$L_D = \mathbb{E}_{\boldsymbol{x} \sim p_{\text{data}}(\boldsymbol{x})}[\log(1 - D(\boldsymbol{x})] + \mathbb{E}_{\boldsymbol{z} \sim p_{\boldsymbol{z}}(\boldsymbol{z})}[\log(D(G(\boldsymbol{z}))]. \quad (31)$$

The losses of GAN do not form a meaningful performance metric in the same way as for most other machine learning models, i.e. the losses do not indicate the quality of the generated data. Instead a GAN can be evaluated by inspection of the data it generates and, since the training of the GAN aims to reach a balance between the performance of the generator and the discriminator (an equilibrium in the minimax game), convergence of the generator and discriminator loss is sought [12].

3.12 Conditional GAN (CGAN)

The conditional GAN (CGAN), introduced by Mehdi Mirza and Simon Osindero [13], is an extension of the type of network described in Section 3.11. In order to generate data that is directed towards a certain mode, the generator and discriminator are conditioned on some extra information, $\boldsymbol{y}$, e.g. class labels for producing class conditional samples. The input to the generator will consist of a latent vector, $\boldsymbol{z}$, from a latent space $p_{\boldsymbol{z}}(\boldsymbol{z})$ and the information, $\boldsymbol{y}$. The input to the discriminator will consist of data, $\boldsymbol{x}$, coming from either the training data, p_{data}, or the generative model, G, together with the information, $\boldsymbol{y}$. The objective function for a CGAN is

$$\min_G \max_D V(G, D) = \mathbb{E}_{\boldsymbol{x} \sim p_{\text{data}}(\boldsymbol{x})}[\log D(\boldsymbol{x}|\boldsymbol{y})] + \mathbb{E}_{\boldsymbol{z} \sim p_{\boldsymbol{z}}}[\log(1 - D(G(\boldsymbol{z}|\boldsymbol{y})))]. \quad (32)$$

3.13 Auxiliary Classifier GAN (AC-GAN)

The auxiliary classifier GAN (AC-GAN) is a variant of the CGAN, introduced by Augustus Odena et al. [14]. Instead of supplying the discriminator with extra information, as in the CGAN, it is instructed to construct this extra information. For this purpose, the discriminator in an AC-GAN contains an auxiliary network. The auxiliary network works as a classifier, i.e. it outputs a corresponding class label for the input. Hence, the discriminator outputs both a label given by the auxiliary network and a probability of realness of the input (as in the regular GANs).

The generator takes as input a latent vector, $\boldsymbol{z}$, and a class label, c, and generates images, $X_{fake} = G(c, \boldsymbol{z})$. The discriminator outputs both a probability distribution over sources, $S = \{real, fake\}$, and over class labels, C, as $P(S|X), P(C|X) = D(X)$. The objective functions consist of

$$\begin{cases} L_S = \mathbb{E}[\log P(S = real|X_{real})] + \mathbb{E}[\log P(S = fake|X_{fake})], \\ L_C = \mathbb{E}[\log P(C = c|X_{real})] + \mathbb{E}[\log P(C = c|X_{fake})], \end{cases} \tag{33}$$

where the discriminator aims to maximize $L_S + L_C$ while the generator aims to maximize $L_C - L_S$. In this sense, the generator and discriminator are actually helping other as both want to maximize L_C to create good class conditioning.

3.14 Metrics

In order to measure the performance of a machine learning model, it is important to use a metric that suits the problem at hand. Some metrics that are used in machine learning for classification are *accuracy*, *precision*, *recall* and *F1 score* [15]. To define these metrics, we first introduce the notation true positives (TP), false positives (FP), true negatives (TN) and false negatives (FN). For simplicity, these terms are introduced with the help of a *confusion matrix.* A confusion matrix is a table that shows statistics about what class the data points *actually* belongs to and what class they were *predicted* by a classifier to belong to. In the case of a binary class problem, with classes 1 (true) and 0 (false), the confusion matrix can be seen in Figure 9. The true positives are data points that are true (1) and are predicted as true (1) while true negatives are data points that are false (0) and are predicted as false (0). In the same way, false positives and false negatives are data points that are false (0) but predicted as true (1) and vice versa.

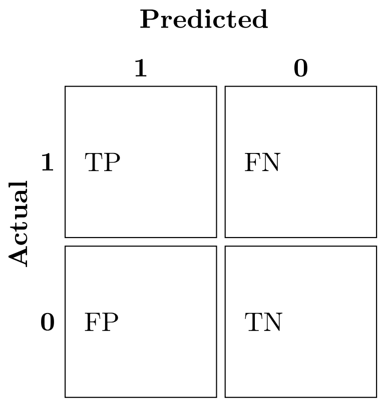


Figure 9: Confusion matrix for a binary classification problem.

Accuracy

Accuracy is the ratio between correct predictions and all predictions, i.e.

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}}. \tag{34}$$

Precision

Precision is the ratio between relevant data points that were retrieved and all the retrieved data points, i.e.

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}. \tag{35}$$

Recall

Recall (or *sensitivity*) is the ratio between the relevant data points that were retrieved and all the relevant data points, i.e.

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}. \tag{36}$$

A similar metric which measures the ratio between actual negative data points and the predicted negative data points is the *specificity*, defined as

$$\text{Specificity} = \frac{\text{TN}}{\text{TN} + \text{FP}}. \tag{37}$$

F1 score

The F1 score is the harmonic mean between precision and recall, i.e.

$$\text{F1} = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}. \tag{38}$$

The different metrics have different benefits and weaknesses. The accuracy is good when all the classes in the data are equally important, the precision is good to use when false positives can have a very negative impact and the corresponding relationship holds for recall and false negatives. The F1-score is a combination of precision and recall and as such a high F1-score means that the number of both false positives and false negatives is low. Also, the F1-score is more useful than the accuracy when there is an imbalanced class distribution in the dataset [15].

4 Methods

All models were built and trained using the TensorFlow-Keras framework in Python.

4.1 Data

The data used for training our GAN networks were provided by CellaVision and consisted of 249 999 labeled 256×256 RGB pixel images of blood cells belonging to 19 different classes. See Table 5 in Appendix for the class distribution. In Figure 10, nine such blood cell images are shown. To train our classifier, we used another dataset, consisting of 200 072 images with the same properties as in the aforementioned dataset. The class distribution for this dataset can be seen in Table 6 in Appendix.

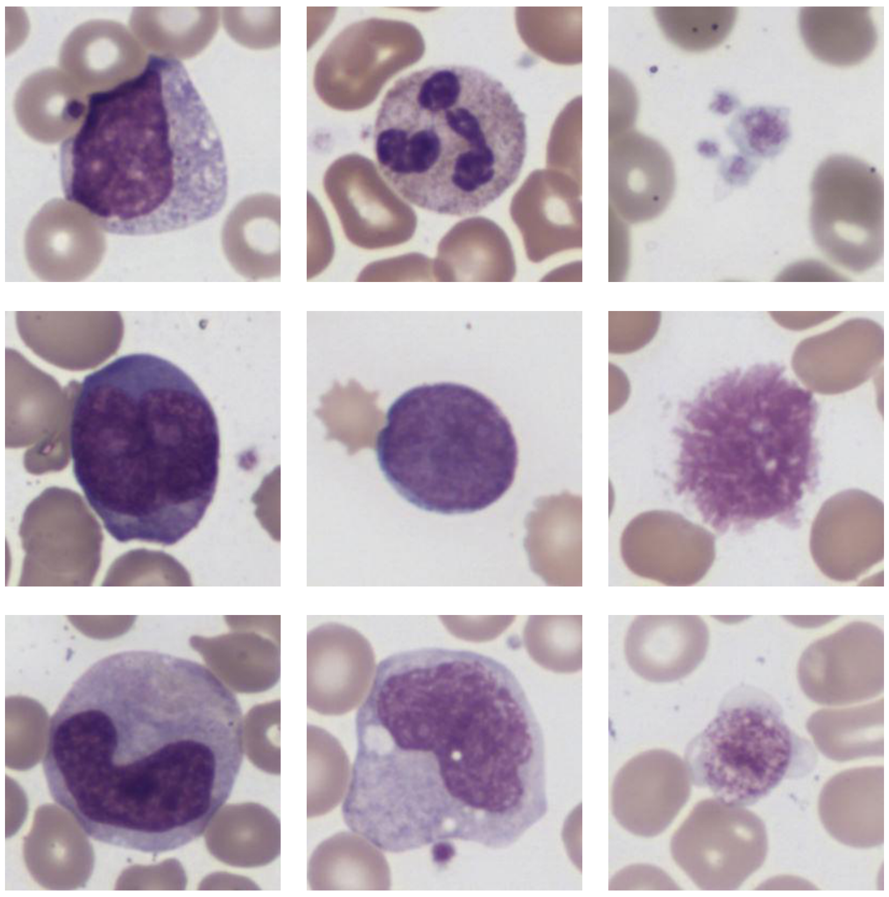


Figure 10: Examples of the data consisting of blood cell images.

4.2 Preprocessing and Data Flow

We used the Python computer vision library openCV to read the images into Python. OpenCV read the images into $256 \times 256 \times 3$-dimensional numpy arrays with each pixel being represented by an integer between 0 and 255. Before the training, we converted the pixel values p_i to floats and rescaled to $[-1, 1]$ using the transformation

$$\hat{p}_i = \frac{p_i - 127.5}{127.5}. \tag{39}$$

As the data used in our training sessions consisted of several hundred thousand images, it was not feasible to keep all of the data in the memory for the whole training sessions. This problem is solved by using a *data generator* which loads a portion of the data into the memory, and keeping it there only when it is needed. To avoid confusion with the generator network in the GAN model, we will refer to the data generator as *data reader.* The data reader provided by TensorFlow was, in our opinion, too closely reliant on other TensorFlow modules which we chose not to use when training our models, so we wrote our own data reader. This data reader reads, shuffles, and batches the data according to specification by the user as well as performing preprocessing.

4.3 Network Architecture

With very small variations, all of the networks used in our experiments have the same network architecture. The network architecture we chose was inspired by the Deep Convolutional GAN (DCGAN) introduced by Radford et al. [16]. In this architecture, transposed convolutions are used instead of pooling functions, such that the generator learns its own spatial upsampling and the discriminator its own spatial downsampling [16]. Another feature of the architecture is that fully connected layers are only used to connect the input and output with the highest convolutional features for the generator and discriminator respectively. In order to stabilize training, batch normalization is applied on all layers in the generator except for the output layer. In the generator, the leaky ReLU function is used as non-linear activation function for all layers except the output layer, where the tanh function is used. In the discriminator, the leaky ReLU function is used as activation function for the inner layers while the output layer has a sigmoid activation. Dropout is performed after each layer in the discriminator except the output layer. The structure of the generator and discriminator can be seen in Figure 11 and 12 respectively. Figure 11 shows how the generator upsamples an input latent vector (which has been excluded in the figure) until it has the size of an image while Figure 12 shows how the discriminator downsamples an image to a scalar corresponding to a probability. Absent in these illustrations are the activation functions, batch normalization and dropout layers. These can instead be seen in the more detailed network visualizations in Figures 22-26 in the Appendix.

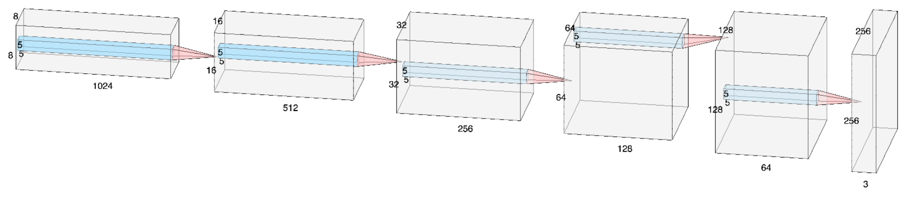


Figure 11: Illustration of the generator network architecture used in our experiments. The numbers next to the boxes signify width, height, kernel size, and depth (number of filters). The illustration was drawn using software on the website `http://alexlenail.me/NN-SVG/AlexNet.html` created by Alex Lenail.

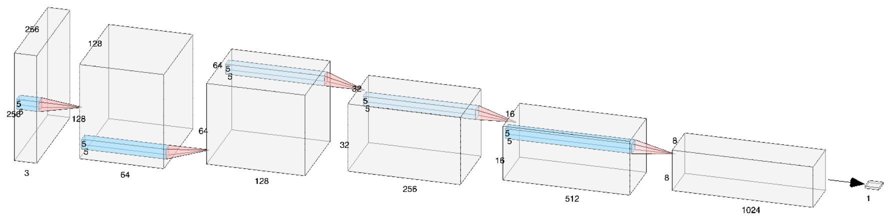


Figure 12: Illustration of the discriminator network architecture used in our experiments. The numbers next to the boxes signify width, height, kernel size, and depth (number of filters). The illustration was drawn using software on the website `http://alexlenail.me/NN-SVG/AlexNet.html` created by Alex Lenail).

Hyperparameters

Hyperparameters that need to be set in our GAN models are: buffer size (amount of data used), batch size, number of epochs (i.e. the number of times the entire training data flows through the model in the training process), latent space dimension, learning rates of the optimizers used by the generator and discriminator respectively, as well as kernel size and stride for the convolutional layers. Furthermore, the dimensions of the first convolutional layer in the generator and discriminator must be set, which (together with the kernel size and stride) also determines the dimensions of the following layers.

In addition to the above stated hyperparameters, the AC-GAN requires that the number of classes be set. In the AC-GAN additional fully connected layers are implemented and the amount of these and their dimensions must also be set.

4.4 Generating Blood Cell Images using GAN

The first problem we aimed to solve was that of generating data following the same distribution as the training data without conditioning on class labels, i.e.

we wanted the output to be images resembling real blood cells belonging to any class. For this purpose, we used a GAN. In the text that follows, we will refer to this GAN as Vanilla-GAN since it is the most basic GAN used in our experiments.

The GAN we used consists of a generator and discriminator, as described in Section 3.11. The network architectures for the generator and discriminator follow that described in Section 4.3 and can be see in Figures 22 and 23 in the Appendix. The blood cell images' pixel values were rescaled and the data was loaded into the GAN model by using our data reader as described in Section 4.2. The generator takes as input a 100-dimensional noise vector and transforms this to produce an image of size $256 \times 256 \times 3$. In other words, the generator can be described as a mapping $G : \mathbb{R}^{100} \to \mathbb{R}^{256\times256\times3}$. The discriminator takes as input an image of size $256 \times 256 \times 3$ and maps this down to a probability of realness, i.e. $D : \mathbb{R}^{256\times256\times3} \to \mathbb{R}$.

The cost function used for the Vanilla-GAN is the cross-entropy defined in Equation (1) and the optimizer used is the *Adam* optimizer, introduced by Kingma and Lei Ba in 2014 [17]. Adam is an adaptive implementation of the SGD algorithm described in Section 3.3, it is adaptive in the sense that it computes individual adaptive learning rates for different parameters. The choice of optimizer followed the approach used by Radford et al. in the DCGAN paper that our architecture is based on as described in Section 4.3.

Testing our Images with an Expert

In order to test the quality of our Vanilla-GAN generated blood cell images, we let an expert, in the form of a medical technologist, examine a dataset consisting of 50 real blood cell images and 50 Vanilla-GAN generated images and assess which were real and which were synthetic. The dataset was shuffled so that real and fake images appeared in random order. The expert did not beforehand know the relative frequency of real and fakes.

4.5 Generating Class-specific Blood Cell Images using AC-GAN

Generating general blood cells using Vanilla-GAN is a good way of ensuring that the network architecture is sufficient for generating good quality images. However, its practical use is limited as there is no way of controlling what kind of blood cell is being created. This, for obvious reasons, renders the images unusable for augmenting training data for a classifier. The CGAN, as described in Section 3.12, aims to solve the problem of generating images of a given class. We did not succeed in implementing a CGAN which yielded satisfactory results. The images generated often lacked the desired image quality which we had seen in the ones generated by our Vanilla-GAN. In order to achieve a similiar image quality as the Vanilla-GAN while at the same time getting a strong conditioning on class labels, we tried a variant of the CGAN which not only feeds extra information into the discriminator, but instead encourages this to construct the extra information. This is done by adding an auxiliary classifier to the discriminator which is being trained in parallel to the regular discriminator. This implementation of a GAN is known as AC-GAN, as described in Section 3.13.

The blood cell images were loaded into the model using our data reader in the same way as in the previous section. The generator takes as input a 100-dimensional latent vector together with a one dimensional label and maps these as $G : \mathbb{R}^{100} \times [0, 18] \subset \mathbb{Z} \to \mathbb{R}^{256\times256\times3}$. The discriminator takes as input an image and maps this to both a probability of realness and a class label as $D : \mathbb{R}^{256\times256\times3} \to \mathbb{R} \times [0, 18] \subset \mathbb{Z}$. Note that the AC-GAN does not get feeded information about what class the input image belongs to (as in the CGAN), instead this information is constructed by the auxiliary classifier and is part of the output of the discriminator. The auxiliary classifier is implemented in the discriminator by adding a fully connected layer with softmax activation, outputting probabilities of class association where the index with maximum value gives the class label. This layer is in addition to the fully connected layer with sigmoid activation which serves as the discriminator, outputting a probability of realness, just as in the Vanilla-GAN.

To further improve the quality of the generated images, which were of really low quality for our first implementations of the AC-GAN (while showing promising conditioning abilities), we used a pre-trained generator and discriminator from a Vanilla-GAN and freezed all of the layers of the generator and all the layers of the discriminator except for the fully connected layers corresponding to the discriminative and classyfying layers, which are both held trainable. Three fully connected layers were added to the generator, connecting the input to the first layer of the generator, and three fully connected layers were addded to the dicriminator, connecting the last convolutional layer to the auxiliary classifier. The principal idea of this was that the freeezed pre-trained generator and discriminator would contribute with information about how to produce high quality images of general blood cells while the trainable fully connected layers would contribute with the conditioning on class labels. In a sense, one can see the training of these fully connected layers as a way of locating which areas of the latent space belongs to which class.

The network architectures for the generator and discriminator of our AC-GAN can be seen in Figure 24 and Figure 25 in the Appendix. The Sequential boxes in the figures correspond to the pre-trained generator and discriminator model respectively, which are the same as in the Vanilla-GAN.

4.6 Latent Space Experiments

The work done in this section was prompted by our wish to better understand the mapping between the latent space and the image space induced by training a Vanilla-GAN. While not all of it is directly related to the aim of this thesis, we feel that the results are valuable in evaluating the result of the training.

Interpolation of Sampled Images

With a trained Vanilla-GAN generator it is possible to create random images by sampling vectors from the latent space. If we sample two vectors from the latent space and generate images, the two images will most likely be quite different, which is to be expected given the relatively heterogeneous nature of the dataset. Given the two generated images, we can ask ourselves the question of what images lie between the two in the image space. A partial answer can be attained by linearly interpolating between the two latent vectors and see which images in the image space they map to. Denoting the two latent vectors z_1 and z_2, an n-step interpolation is given by

$$\hat{z}_k = z_1 + \frac{z_2 - z_1}{n} k, \tag{40}$$

where $\hat{z}_k$ is the resulting latent vector at the k:th step of the interpolation. For example, the midpoint between z_1 and z_2 is given by setting $k = n/2$. By generating the corresponding sequence $\{G(\hat{z}_k)\}_{k=0}^{n}$ we can examine how the images generated by points on the line between the two latent vectors look.

Finding the Best Reconstruction of a Real Image

Given a real image, we wanted to examine if there was a vector in the latent space which could produce an image which is similar. For an individual image I and a pretrained GAN generator G, this is an optimization problem where the sought after latent vector $\tilde{z}$ is attained as

$$\tilde{z} = \underset{z}{\operatorname{argmin}} ||I - G(z)||, \tag{41}$$

where the norm is, for example, L^1, and the subtraction is pixelwise. Rather than solving the optimization for each example, we wanted to see if we could train a convolutional neural network to map an image to $\tilde{z}$. We refer to this network as an *image-to-latent network*. Since the problem is quite similar to the one solved by the discriminator in the Vanilla-GAN model, we adopted a similar architecture, changing the output from a scalar to a 100-dimensional vector. See Figure 26 in the appendix for details. The training process can be summarised as follows:

1. Draw a batch of real images.
2. Generate latent vectors by passing them to the image-to-latent network.
3. Generate images by passing the latent vectors to the GAN generator.
4. Calculate the loss as the norm between the real and generated images.
5. Let the optimizer update the parameters of the image-to-latent network.

We used the Adam optimizer and L^1-norm as loss function.

Generating Data By Perturbing Reconstructed Images

The reconstruction of real images introduced in the previous section has the nice property that it gives real images a representation in the latent space. This, for example, enables us to do interpolation between two reconstructed images, using the method outlined in Section 4.6. We also evaluated a possible method for generating data using reconstructed images. The idea is as follows: When generating data, we are looking for samples that are close but not equal to the data we already have. Given a reconstructed image, we can create samples that are slightly different by adding noise to the reconstructing latent vector, thereby *perturbing* the reconstructed image. Denoting the reconstructing latent vector $\tilde{z}$ and the GAN generator G, we can generate any number of new images as

$$I_{perturbed}(\tilde{z}, \epsilon) = G(\tilde{z} + \epsilon), \tag{42}$$

with $\epsilon \in N(0, \sigma^2 \boldsymbol{I})$ where $\boldsymbol{I}$ is the identity matrix for the latent space dimension. The choice of σ^2 will determine how perturbed the images will be.

4.7 Training A Classifier On Synthetically Augmented Datasets

One of the main goals of this thesis is to improve classification performance by augmenting a dataset with synthetic data. The classifier we used is the Xception classifier, introduced by Francois Chollet in 2017 [18]. As this thesis is about *improving* the performance of the classifier by means of synthetic augmentation and not about choosing the optimal classifier, we will not go into details about the classifier or why we chose the Xception classifier in particular other than that it has performed well in experiments at CellaVision. The important aspect is how the augmentation affects the performance of the given classifier.

To get a baseline, in order to evaluate how the augmentation affects the classification performance, we first wanted to train the chosen classifier on a dataset without any augmentation. We trained the classifier on a dataset that did not have any overlap with the dataset used to train the Vanilla-GAN and AC-GAN. In addition to this, we also trained the classifier on the dataset augmented with the classical data augmentation methods rotate and zoom. After we had our baseline results, we augmented both the original dataset and the classically augmented dataset with synthetic data generated by our AC-GAN and trained the classifier on these new datasets respectively.

5 Results

5.1 Generating Blood Cell Images using GAN

The hyperparameters used for training the Vanilla-GAN model can be seen in Table 1. The learning rates of the generator and discriminator are referred to as α_G and α_D respectively while the number of filters in the first convolutional layer of the generator and discriminator are referred to as Depth_G and Depth_D respectively.

Table 1: Hyperparameter settings for training of the Vanilla-GAN model.

Hyperparameters	
Buffer size	240 000
Batch size	20
Epochs	75
Latent dimension	100
α_G	$1 \cdot 10^{-5}$
α_D	$1 \cdot 10^{-5}$
Kernel size	5×5
Stride	2
Depth_G	1024
Depth_D	64

In Figure 13, nine blood cell images generated by the Vanilla-GAN are shown. The images were generated by inputting nine different 100-dimensional latent vectors consisting of Gaussian noise into the trained generator. The generator produces images with a quality not far from the true data, see Figure 10. Some of the red blood cells are a bit more smudgy than in the original dataset and the contours of both the red and white blood cells are not quite as sharp as in the real images. Since the Vanilla-GAN does not condition on classes, we cannot know what classes these generated blood cells belong to but based on the images in Figure 13, the generator seems to be able to produce images with a large variation, i.e. it is not prone to produce duplicates.

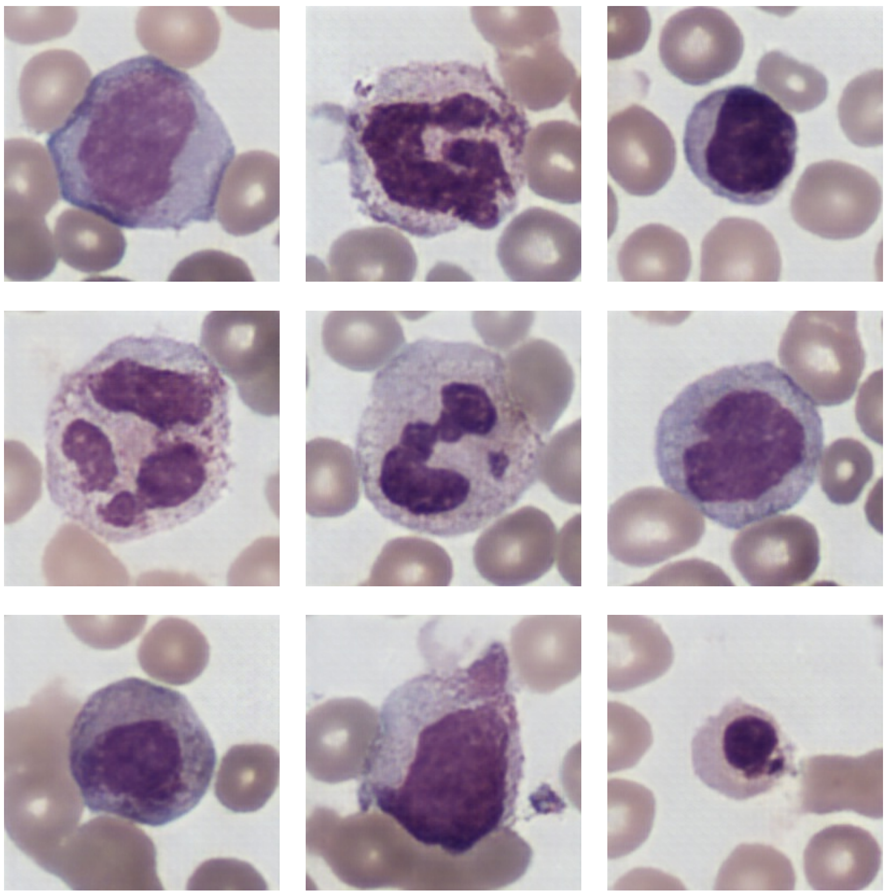


Figure 13: Blood cell images generated after 75 epochs of training by a Vanilla-GAN.

In Figure 14, the losses of the generator and discriminator are shown as functions of the number of epochs. From the plot we can see that the generator and discriminator reach equilibrium around epoch 30 and keep this equilibrium till around epoch 75 where the losses start to diverge. This, together with the good image quality, is why we chose to generate images with the generator that had been trained for 75 epochs.

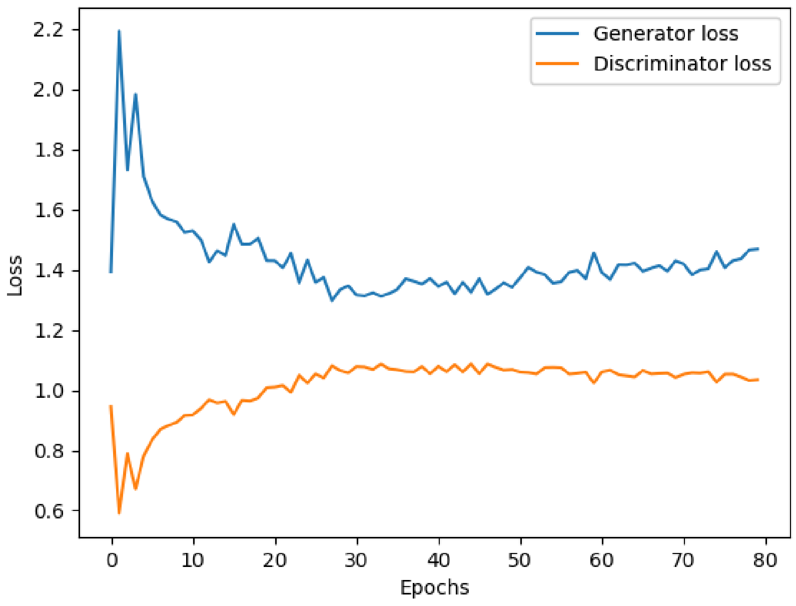


Figure 14: The losses of the generator and discriminator as functions of the number of epochs.

Testing our Images with an Expert

In Table 2, the confusion matrix for the medical technologist's classification of real images and Vanilla-GAN generated images can be seen. The results show a recall (sensitivity) of 0.88 and a specificity of 0.54.

Table 2: Confusion matrix for the medical technologist's classification of real images and Vanilla-GAN generated images. The total amount of images were 100: 50 real and 50 fake.

		Predicted	
		Real	Fake
Actual	Real	44	6
	Fake	23	27

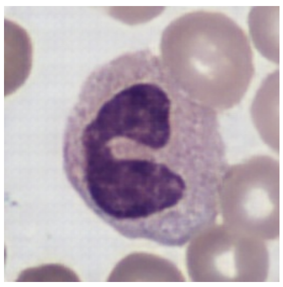
5.2 Generating Class-specific Blood Cell Images using AC-GAN

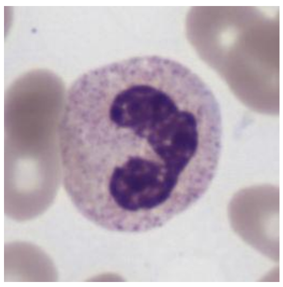
In Table 3, the hyperparameters used for training the AC-GAN are shown. Additional hyperparameters are the number of fully connected layers that are added to connect the input and the output with the highest convolutional features for the generator and discriminator respectively. Three fully connected layers, each consisting of 100 neurons, with ReLU activation functions were added to connect the input with the pre-trained layers of the generator and three fully connected layers on the same form were added to the discriminator to connect the pre-trained layers with the auxiliary classifier.

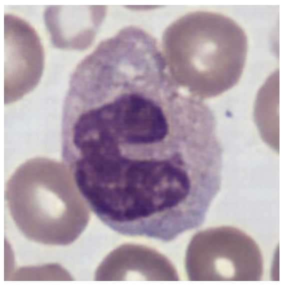
Table 3: Hyperparameter settings for training of the AC-GAN model.

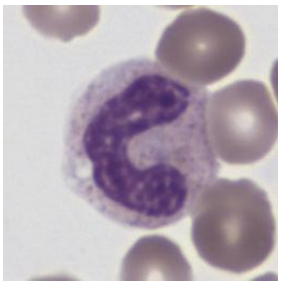
Hyper parameters	
Buffer size	238 080
Batch size	32
Epochs	1
Latent dimension	100
Number of classes	19
α_G	$2 \cdot 10^{-4}$
α_D	$2 \cdot 10^{-4}$

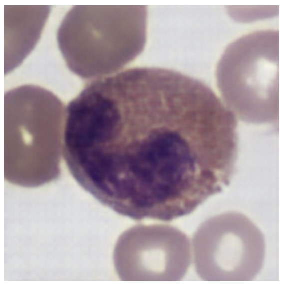
In Figure 15, comparisons between AC-GAN generated (fake) blood cell images and real blood cell images are shown. The images to the left are fake while the images to the right are real. It should be said that for this comparison we have chosen cell images from the real data that resembles the generated data as much as possible and so the comparisons shown here might not be representative over the entire set of real and fake data. The selection was made to show the class conditioning abilities of the AC-GAN. The quality of the images are not quite as high as the Vanilla-GAN generated images in Figure 13. Achieving good in-class variability using AC-GAN proved difficult as can be seen in Figure 16 which shows nine AC-GAN generated images belonging to class 0. The generator seems keen on generating images belonging to certain modes, i.e. certain output types. This is an example of *mode collapse*, a common problem with GANs. In the figure, three such modes are shown (one mode for each row).

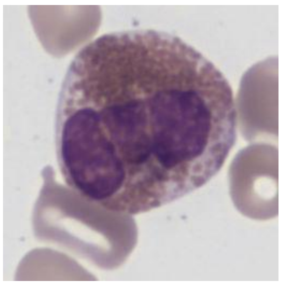

Fake Class 0

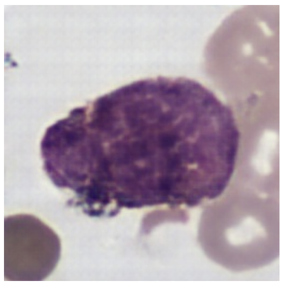

Real Class 0

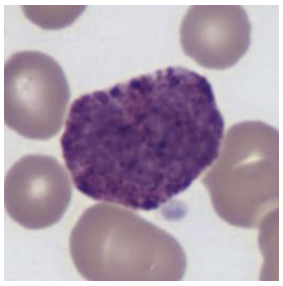

Fake Class 1

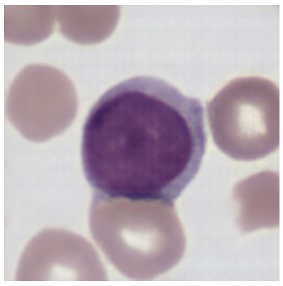

Real Class 1

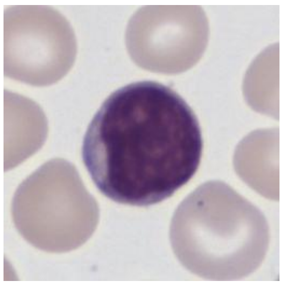

Fake Class 2

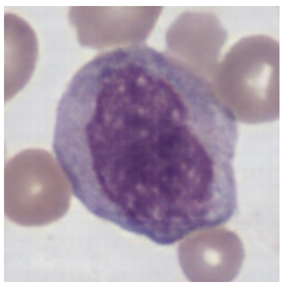

Real Class 2

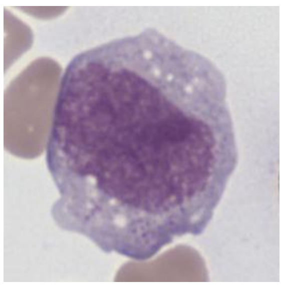

Fake Class 3

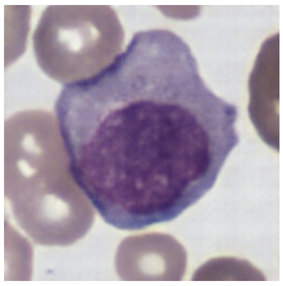

Real Class 3

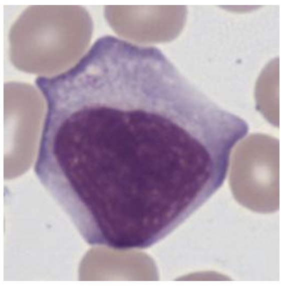

Fake Class 4

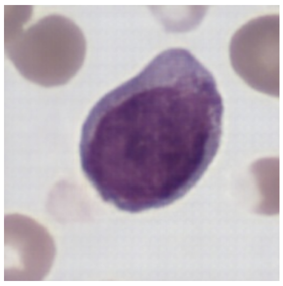

Real Class 4

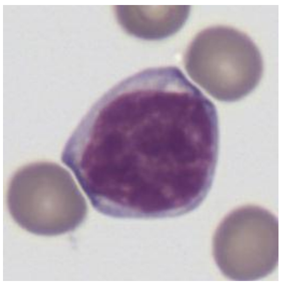

Fake Class 5

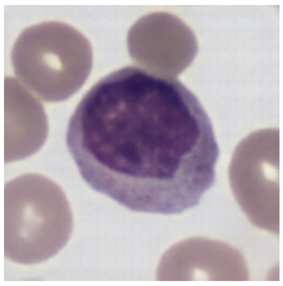

Real Class 5

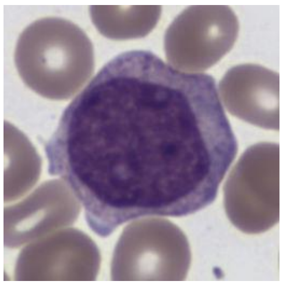

Fake Class 6

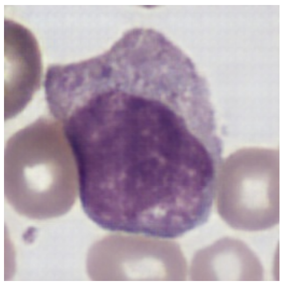

Real Class 6

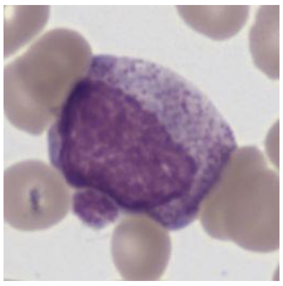

Fake Class 7

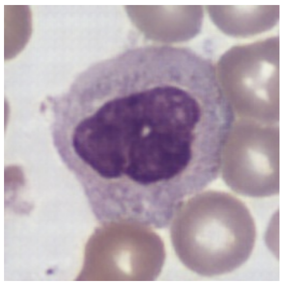

Real Class 7

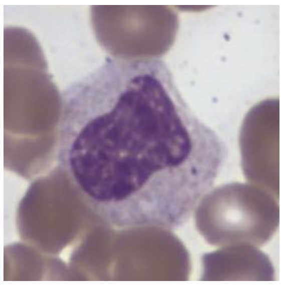

Fake Class 8

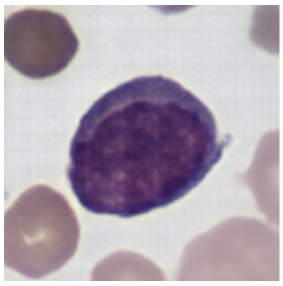

Real Class 8

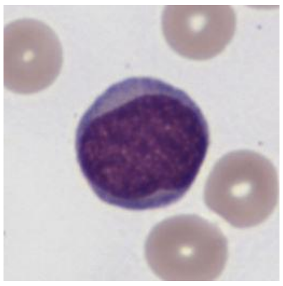

Fake Class 9

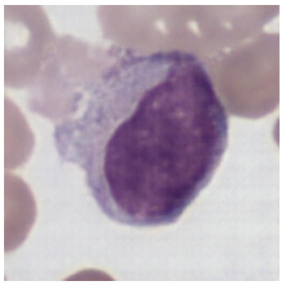

Real Class 9

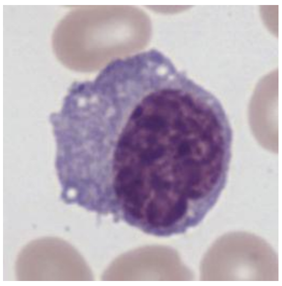

Fake Class 10

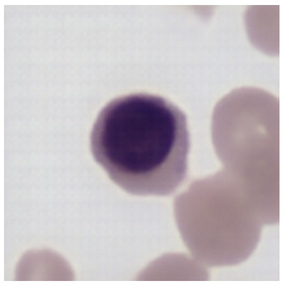

Real Class 10

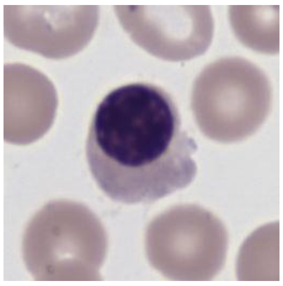

Fake Class 11

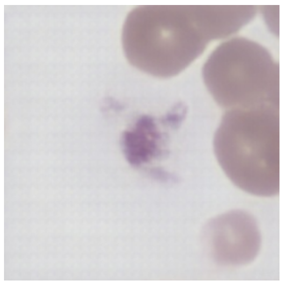

Real Class 11

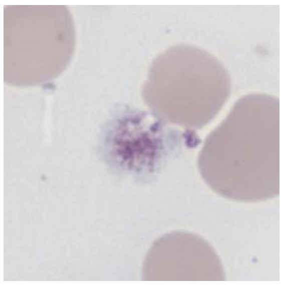

Fake Class 12

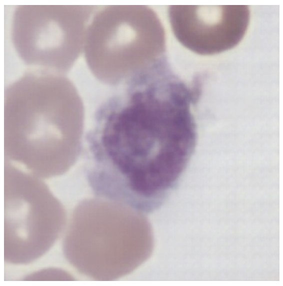

Real Class 12

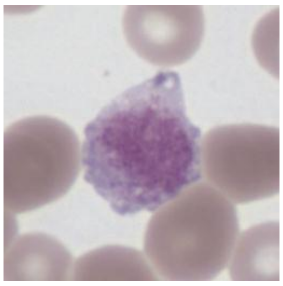

Fake Class 13

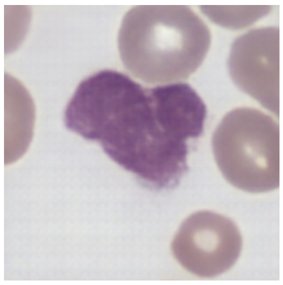

Real Class 13

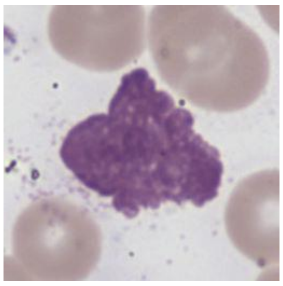

Fake Class 14

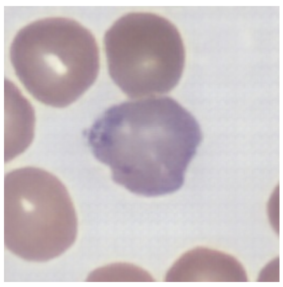

Real Class 14

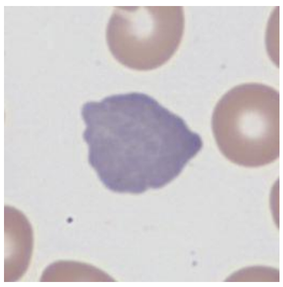

Fake Class 15

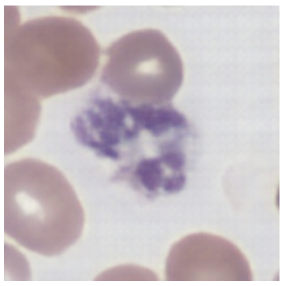

Real Class 15


Fake Class 16


Real Class 16


Fake Class 17


Real Class 17


Fake Class 18

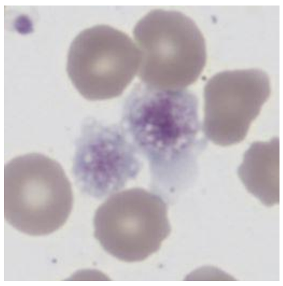

Real Class 18

Figure 15: Comparison between images generated by an AC-GAN and real images, one such pair for each of the 19 classes (0-18).

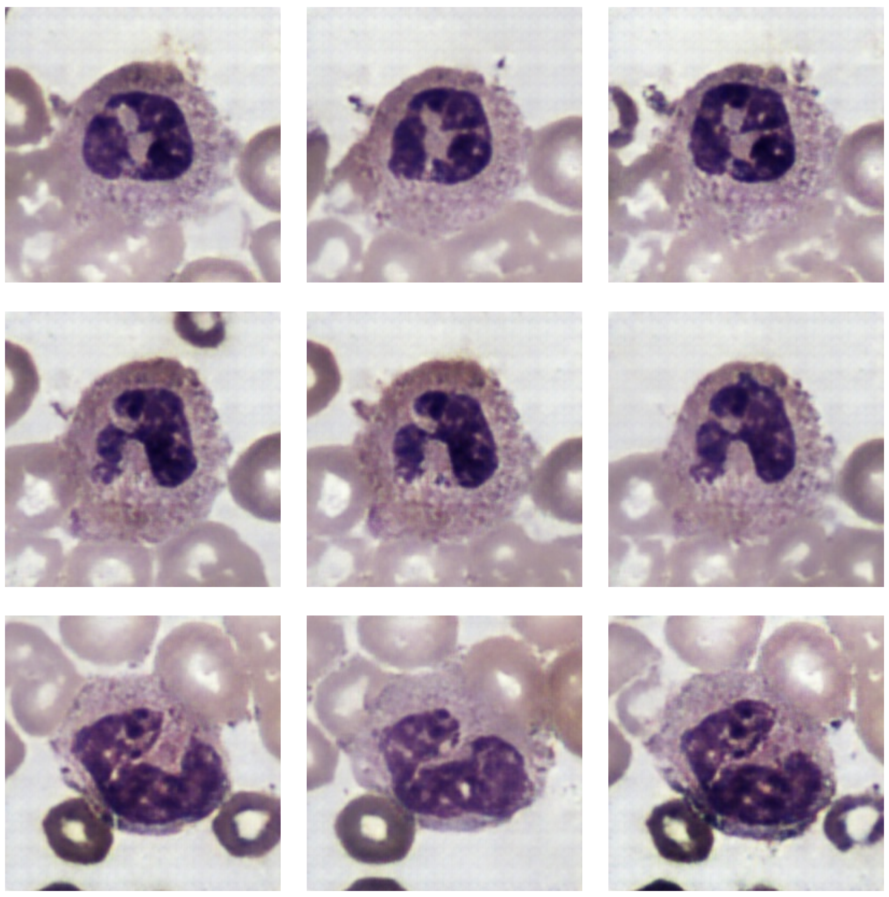
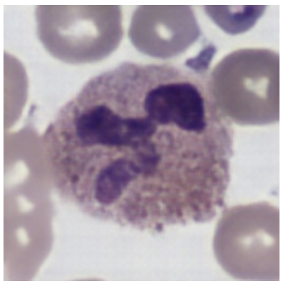


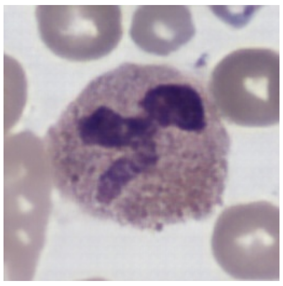
Figure 16: Nine AC-GAN generated blood cell images belonging to class 0 exhibiting the mode collapse tendencies of our AC-GAN.

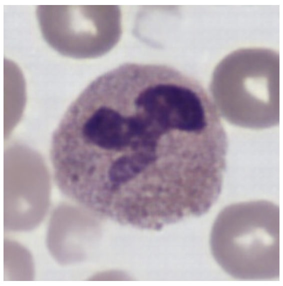
5.3 Latent Space Experiments

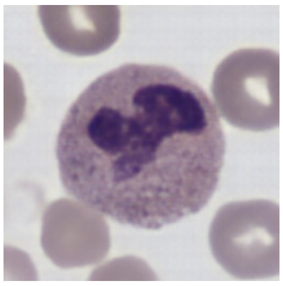
Interpolating Between Generated images

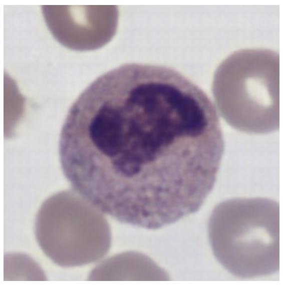
In Figure 17 we see an interpolation between two generated images using the method outlined in Section 4.6. The blood cells morph into each other quite smoothly, with the nucleus shape, red blood cell configurations, and cell texture all approaching the second cell.

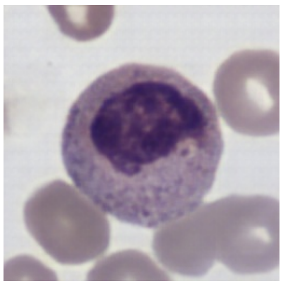

(a) Generated image 1

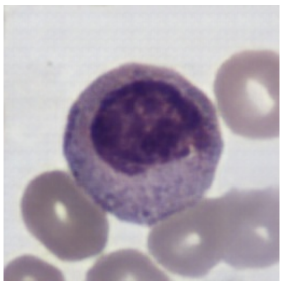

(b) interpolation 1

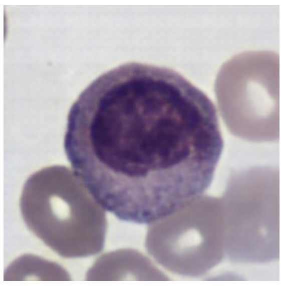

(c) interpolation 2


(d) interpolation 3


(e) interpolation 4


(f) interpolation 5


(g) interpolation 6


(h) interpolation 7

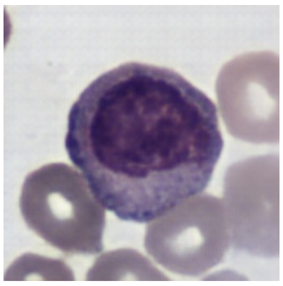
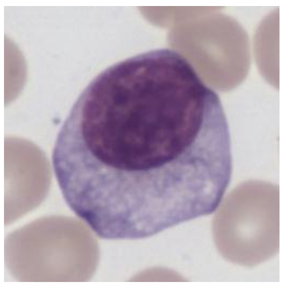
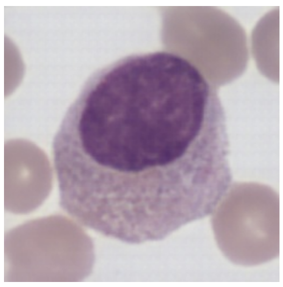

(i) Generated image 2

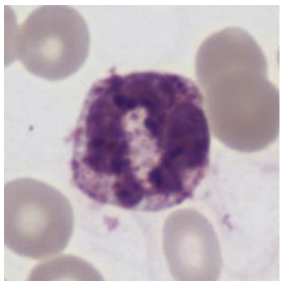
Figure 17: An 8-step interpolation between two generated images.

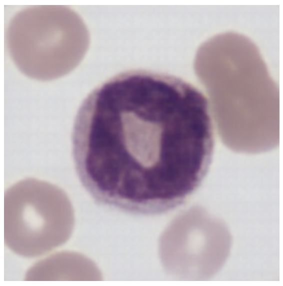
Finding the Best Reconstruction of a Real Image

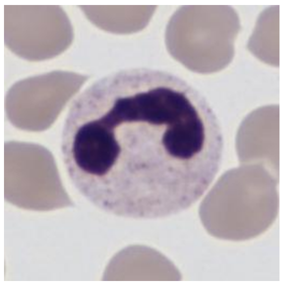
In Figure 18 we see five images in the dataset and their reconstructions created by finding their best reconstructing latent vector using the image-to-latent network. We used a learning rate of $1 \cdot 10^{-5}$ while training the image-to-latent network.

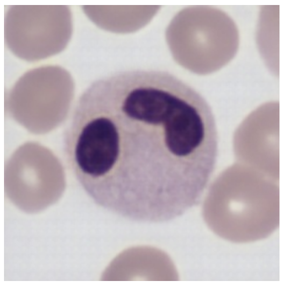

Real image

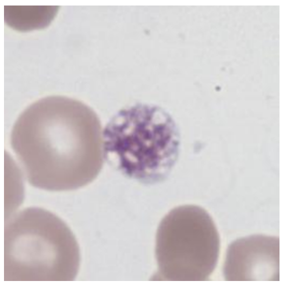

GAN reconstruction

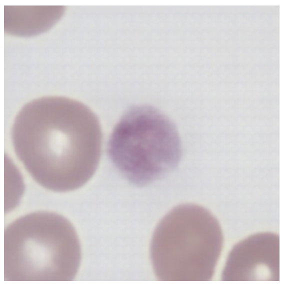

Real image


GAN reconstruction


Real image


GAN reconstruction


Real image


GAN reconstruction

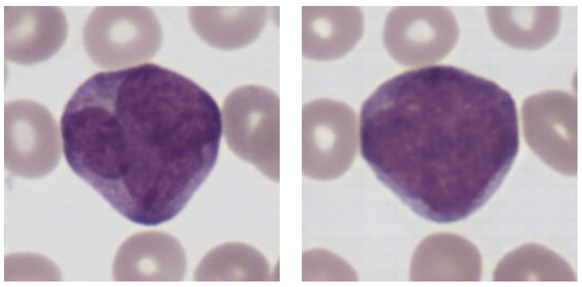


Figure 18: Comparison between real images and their reconstructions.

Generating Data By Perturbing Reconstructed Images

In Figure 18 we see images generated by adding zero-mean Gaussian noise with variance $\sigma^2 = 0.04$ to the best reconstructing latent vectors of the images seen in Figure 18 and generating images from the thereby perturbed latent vectors. We see five perturbed images for each original image.

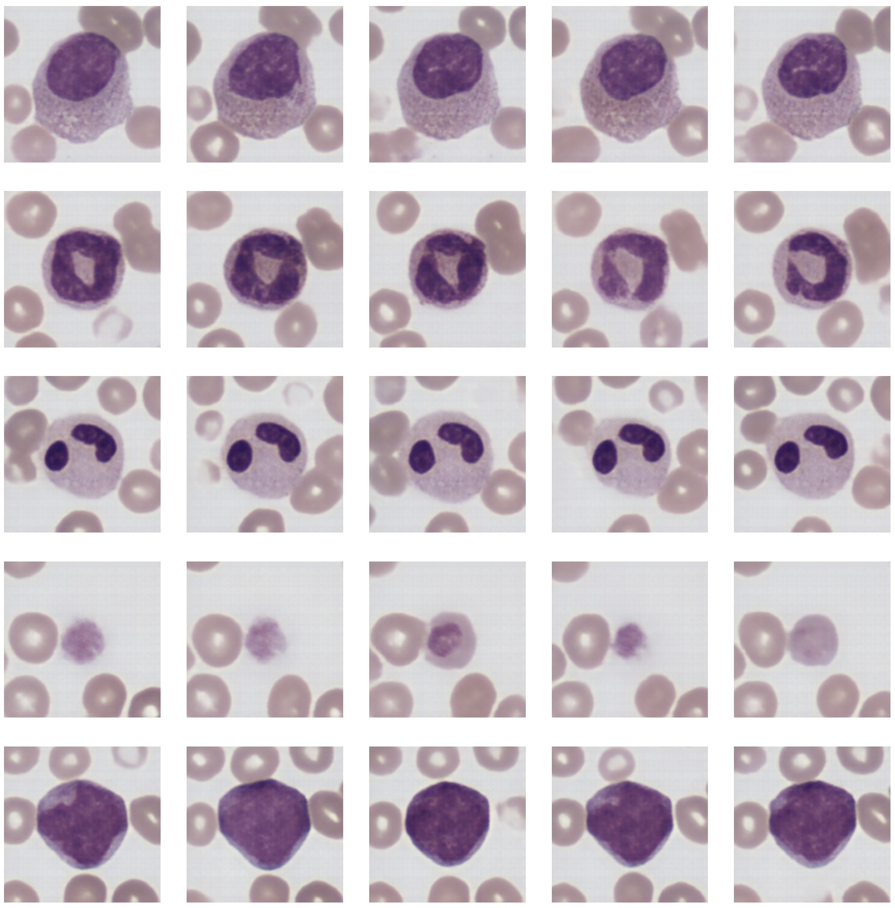


Figure 19: Images generated by perturbing the best reconstructing latent vector of the images in Figure 18.

5.4 Training A Classifier On Synthetically Augmented Datasets

In Figure 20 we see the confusion matrix from classifying a test set using an Xception-classifier trained on non-augmented data. The dataset was divided into 138 496 training images, 30 787 validation images and 30 789 test images. The classification made by this classifier will serve as a baseline. The classification is overall good with most of the classes having above 90 percent correctly classified cells. However, in classes 1, 6, 9, 10, 12, 16 and 18 there seem to be room for improvement.

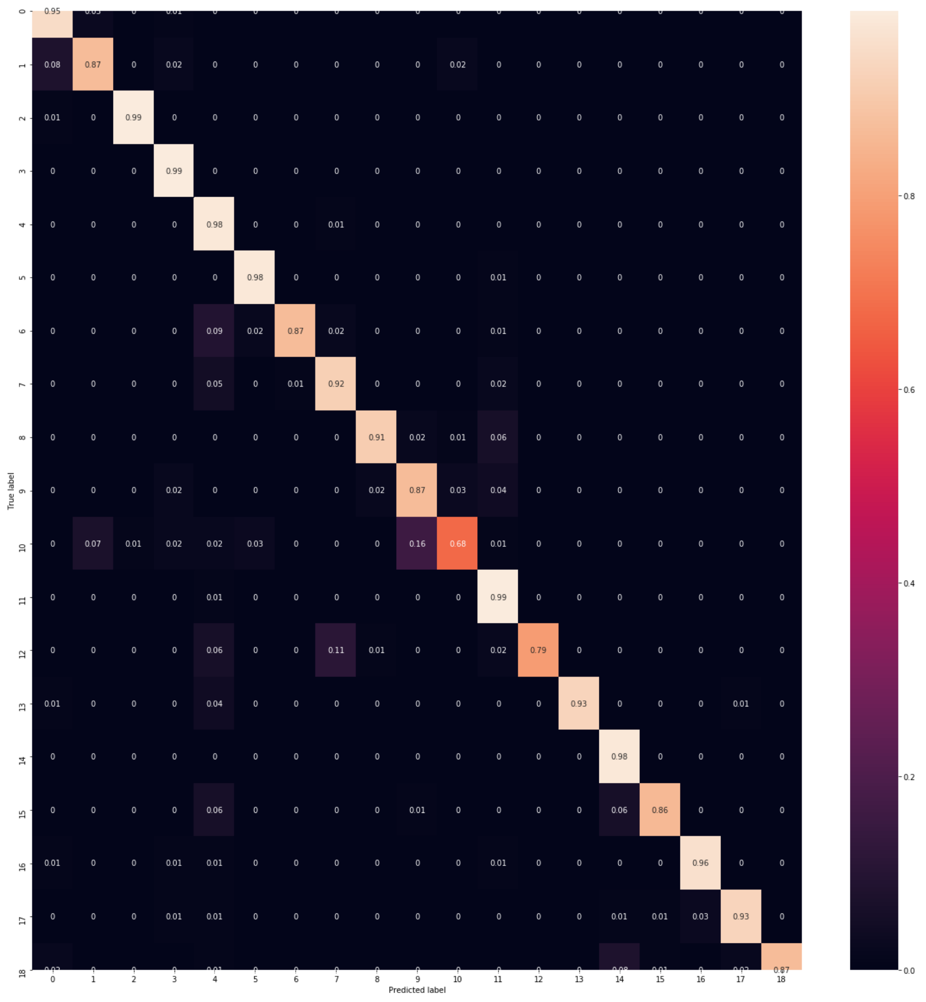


Figure 20: Confusion matrix for the baseline classifier, i.e. the classifier trained on the original dataset without augmentation.

In Figure 21, the confusion matrix from classifying a test set using an Xception-classifier trained on synthetically augmented data can be seen. Each class in the training set was increased by 50 % by augmenting with AC-GAN generated images. The accuracy remains at similar values as for the baseline classifier with some minor reductions for classes 2 (-2 p.p.), 3 (-1 p.p.), 11 (-2 p.p.), 14 (-1 p.p.) and larger drops for classes 1 (-12 p.p.), 4 (-7 p.p.) and 10 (-7 p.p.). Increases in accuracy can be seen for classes 0 (+3 p.p.), 6 (+10 p.p.), 7 (+4 p.p.), 8 (+4 p.p.), 9 (+6 p.p.), 12 (+13 p.p.), 13 (+4 p.p.), 15 (+7 p.p.), 16 (+2 p.p.), 17 (+2 p.p.) and 18 (+5 p.p.). For class 5 the accuracy is the same for the baseline and the synthetically augmentation trained classifier.

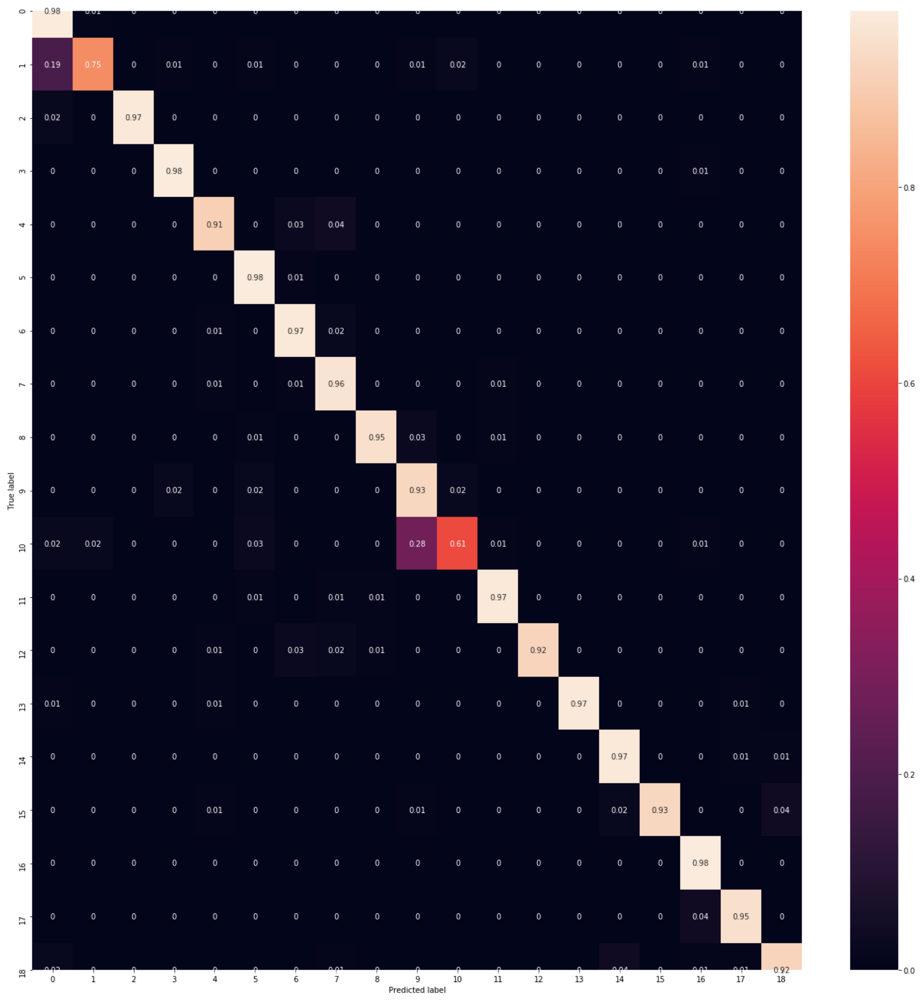


Figure 21: Confusion matrix for the classifier trained on the original dataset augmented with AC-GAN generated images.

To measure the overall performance of the classifier and not only its performance on specific classes the weighted average F1-scores were computed for each trained classifier. In Table 4, the weighted average F1-scores for the baseline and the synthetic augmentation trained classifiers can be seen. The scores are weighted in the sense that metrics are calculated for each class and their average weighted by support is computed to account for class imbalance.

Table 4: Weighted average F1-scores for baseline and synthetic augmentation trained classifier.

Classifier	**Weighted F1**
Baseline	0.946
Synthetic	0.953

Results from training a classifier on data with classical augmentation schemes as well as a combination of classical augmentation and synthetic augmentation were considerably worse than both the baseline seen in Figure 20 and the synthetic augmentation results seen in Figure 21. Therefore, we decided to omit these results from the report and confine ourselves to the comparison between baseline and synthetic augmentation.

6 Discussion

Vanilla-GAN

The generator of our Vanilla-GAN seems to have learned the data distribution of the dataset well in the sense that it is able to generate a large variety of realistic blood cell images that to our eyes are very similar to the images in the dataset. However, a closer look at the images reveal some differences. There is a fine, gridlike texture visible in the background of the images and in the cytoplasma of some cells, which is not visible in the real images. This makes differentiating between real and generated images a quite easy task if you know what to look for. The texture is even more pronounced and of a coarser nature in images from earlier training stages, indicating that the texture is a feature of the model which is partially "trained away". Even so, in the later stages of training, when the model showed clear signs of convergence with regards to cell quality and loss functions, the texture was not completely gone, implying a shortcoming of the selected model. We suspect that the texture is a product of kernel size and/or the number of convolutional filters and that further experimentation might remedy the problem.

These cosmetic concerns aside, the Vanilla-GAN model largely achieved its purpose in our thesis, which was to show that a GAN with the DC-GAN architecture can learn the distribution of our data set and produce credible images. This was further confirmed by testing our Vanilla-GAN generated images on an expert in the form of a medical technologist. The recall of 0.88 shows that the expert performed well on classifying real images as real while the specificity of 0.54 shows that the expert struggled with classifying the fake images: Almost 50% of the fake images were classified as being real. This attests to the quality of our Vanilla-GAN generated images.

AC-GAN

Evaluating the performance of the AC-GAN reduces to two main factors: Does the AC-GAN succeed in generating blood cell images that resemble the real images of a given class and can the generated images be used to improve classifier performance?

We will start by discussing the first question while discussing the latter at the end of this section. As seen in Figure 15, the AC-GAN seems able to condition on every class in the dataset and is able to generate rather high quality images from these classes. Especially the generated images beloning to classes for which the training set consists of many images, for example class 0, exhibit a detailed appearance and a high resemblance to the real images. However, the AC-GAN struggles with some of the classes for which there is a shortage of data, such as classes 12 and 18, and for classes with a distinct appearance that set them apart from the other classes, such as classes 14 and 18. The reason why the AC-GAN struggles with the latter classes could be that they do not share many features with the other classes, instead having a very distinct appearance. The generator has not been exposed to these features as much as for the classes which share more features, hence it is less equipped with generating from them due to lack of exposition in the training phase. For these classes the generator does not benefit from being well trained on the shared characteristics between many of the classes.

A problem with the AC-GAN is that for some classes, such as class 0, it is keen on generating images concentrated on certain modes, a sign of mode collapse. This is illustrated in Figure 16 where three such modes are shown. There are minor differences between the images in a certain mode, but they can almost be considered as duplicates. This can be a problem when using the AC-GAN generated images to augment the training set for classifier improvement. If a large fraction of the augmented images consists of duplicates, the classifier will be exposed to a less diverse dataset during training. This will decrease the classifier's ability to generalize in comparison to if it would have been trained on a dataset with high variability.

Latent Space Experiments

The interpolation between generated images visible in Figure 17 is quite smooth, although some parts of the images exhibits "jumps" from one image to the next, such as a red blood cell disappearing between interpolation 5 and 6. We believe that the fact that such an interpolation is possible is a sign that the Vanilla-GAN generator has generalized well from the images it was trained on, and not just learned to generate images that are very similar to images that are already in the training set.

The images generated by finding the best reconstructing latent vector generally capture the shape of the WBCs as well as the configuration of RBCs. They lack in the finer details and in some cases the colour and texture. The combined structure of the image-to-latent network and GAN generator can be viewed as an auto-encoder, with the aforementioned networks playing the part of encoder and decoder respectively. From this point of view, it is quite remarkable that an image comprised of $256 \times 256 \times 3 = 196608$ pixels can be reconstructed so well from 100 floating point numbers. It might be interesting to explore what this entails in the context of, for example, compression of images.

Having access to the best reconstructing latent vector enables making *changes* to a cell in the dataset, which we believe entails many possible experiments that we have only begun to explore in this thesis. The images in Figure 19 that are perturbed by adding noise to the best reconstructing latent vector do for the most part show the desired small changes in cell appearance. However, some of the perturbations result in changes that very much alter the overall characteristics of the cell. Smaller values for σ^2 yielded variation which was barely visible, implying that the desired small but visible variation comes with the cost of undesired large changes. This is problematic if the objective is in-class data augmentation as the perturbed images might be "pushed" out of the class it belongs to. For this reason, we decided not to attempt to improve a classifier by augmenting datasets with perturbed images. A possible solution to the problem would be to identify a set of restrictions for each class, determining directions in the latent space that the best reconstructing latent vectors can be perturbed in, with the aim of introducing variance without loosing class-defining features. Another way might be to interpolate between different reconstructed images of the same class.

Classifier Performance Improvement

The confusion matrices in Figure 20 and Figure 21 show that synthetic augmentation is able to significantly increase the accuracy of the classifier for some classes, such as classes 6, 9, 12, 15, while we see a rather large decrease in accuracy for classes 1, 4 and 10. Observing the subdiagonal of the confusion matrix in Figure 21, we see that the large drop in accuracy for class 1 seems to occur due to the fact that many of the blood cell images belonging to this class are being classified as class 0. This is not too surprising since the class 1 images generated by our AC-GAN have a high resemblance with the class 0 images, see Figure 15. Furthermore, the class distribution of the dataset used for training the classifier, see Figure 6 in Appendix, shows that there are only 2565 images belonging to class 1 (in comparison, there are 13126 belonging to class 0), so the classifier does not have much data to train on for this class. This in combination with the AC-GAN's inability to generate class 0 and class 1 images with high discrepancy could explain the large drop in accuracy for class 1. The same arguments can be used to explain the drop in accuracy for class 10. The AC-GAN generated images resemble class 9 and in the confusion matrix in Figure 21 we see that many class 10 images are being classified as class 9. An interesting note about the misclassification of class 1 as class 0 and class 10 as class 9 is the biological relation between the classes. Class 0 consists of segmented neutrophils which are a more mature version of the class 1 band neutrophils, succeding them in the process of cell evolution [19]. The same is true for class 9, consisting of myelocytes, which precede the class 10 metamyelocytes in the cell evolution. Our generative model seems to be able to produce images that are on the borderline between classes, e.g. between class 0 and class 1, making it difficult for the classifier to discriminate between the two.

Some of the notable increases in accuracy are seen for classes 6, 9, 12 and 15. In particular, the high increases in accuracy for classes 12 and 15 are of interest since these classes are two of the three classes (class 18 is the third, for which we also see an increase in accuracy) with the least amount of data in the original dataset. This implies that synthetic augmentation seems to be a way to improve classifier performance for classes with a shortage of data. The weighted F1-scores in Table 4 further implies that training a classifier on synthetically augmented data improves the classification performance.

6.1 Conclusions

Conclusions to be drawn from our results are:

- The Vanilla-GAN can generate images with quite high resemblance to real blood cell images, as implied by an expert in the field classifying 46% of the fake images as real.
- The AC-GAN is able to condition on classes and in parts improve classifier performance, especially for classes for which there is a shortage of data. However, the generated images does not quite have the quality of the

Vanilla-GAN generated images and tendencies of mode collapse can be observed.

6.2 Further Work

Further work involves solving the problem of mode collapse for the AC-GAN. On Google's page on GAN, they suggest that a remedy to mode collapse is to implement a GAN using the Wasserstein loss [20]. This type of GAN is called Wasserstein GAN (WGAN), as proposed by Arjovsky et al. in their paper *Wasserstein GAN* [21]. Such an implementation could be worth a try in order to avoid mode collapse. However, how it would affect our version of the AC-GAN is difficult to know.

In order to further improve the quality and diversity of the generated images the GAN models should be trained on a larger amount of data. More data was available for us to use at CellaVision, but we decided to limit the dataset we used in order to achieve faster training of our models.

To improve the classification, different augmentations can be tried. We augmented each class with 50% of the size of the already existing data, it would be interesting to try augmenting it with for example 100% of the size of the existing data. Also, a different choice of classifier might help with improving the classification results. Given a synthetically augmented dataset, there might be classifiers which perform better than the Xception model we used.

References

[1] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial networks. 2014.

[2] Maayan Frid-Adar, Eyal Klang, Michal Amitai, Jacob Goldberger, and Hayit Greenspan. Synthetic data augmentation using gan for improved liver lesion classification. *2018 IEEE 15th International Symposium on Biomedical Imaging (ISBI 2018), Biomedical Imaging (ISBI 2018), 2018 IEEE 15th International Symposium on*, pages 289 – 293, 2018.

[3] Christopher Bowles, Liang Chen, Ricardo Guerrero, Paul Bentley, Roger Gunn, Alexander Hammers, David Alexander Dickie, Maria Valdés Hernández, Joanna Wardlaw, and Daniel Rueckert. Gan augmentation: Augmenting training data using generative adversarial networks. *arXiv*, 2018.

[4] Laura Dean. *Blood Groups and Red Cell Antigens*. National Center for Biotechnology Information (US), Bethesda, MD, 2005.

[5] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, Cambridge, MA, 2017.

[6] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer Science+Business Media, LLC, New York, NY, 2006.

[7] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[8] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(6):1929 – 1958, 2014.

[9] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. 2015.

[10] Sumit Saha. A comprehensive guide to convolutional neural networks — the eli5 way. `https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53`, 2018, Accessed: 2020-03-05.

[11] Naoki Shibuya. Up-sampling with transposed convolution. `https://medium.com/activating-robotic-minds/up-sampling-with-transposed-convolution-9ae4f2df52d0`, 2017, Accessed: 2020-03-05.

[12] Jason Brownlee. A gentle introduction to generative adversarial network loss functions. `https://machinelearningmastery.com/generative-adversarial-network-loss-functions/`, 2020, Accessed: 2020-03-09.

[13] Mehdi Mirza and Simon Osindero. Conditional generative adversarial nets. 2014.

[14] Augustus Odena, Christopher Olah, and Jonathon Shlens. Conditional image synthesis with auxiliary classifier gans. 2016.

[15] Mohammed Sunasra. Performance metrics for classification problems in machine learning. `https://medium.com/thalus-ai/performance-metrics-for-classification-problems-in-machine-learning-part-i-b085d432082b`, 2017, Accessed: 2020-03-05.

[16] Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv*, 2015.

[17] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv*, 2014.

[18] François Chollet. Xception: Deep learning with depthwise separable convolutions. *arXiv*, 2016.

[19] Myeloid cells - hematopoesis. `http://tbl.med.yale.edu/myeloid_cells/reading.php`, 2020, Accessed: 2020-03-09.

[20] Gan - common problems. `https://developers.google.com/machine-learning/gan/problems`, 2020, Accessed: 2020-03-05.

[21] Martin Arjovsky, Soumith Chintala, and Léon Bottou. Wasserstein gan. *arXiv*, 2017.

A Appendix

Table 5: Class distribution of the dataset used for GAN training provided by CellaVision.

Class	#
0	39931
1	10035
2	14962
3	6858
4	30943
5	29404
6	12160
7	10333
8	1698
9	10055
10	5882
11	21153
12	1240
13	11567
14	16994
15	2904
16	13394
17	7833
18	2653

Table 6: Class distribution of the dataset used for classifier training provided by CellaVision.

Class	#
0	13126
1	2565
2	2791
3	1867
4	26552
5	20021
6	15269
7	22749
8	1370
9	2235
10	1171
11	18232
12	1113
13	1933
14	4175
15	546
16	14109
17	3590
18	557

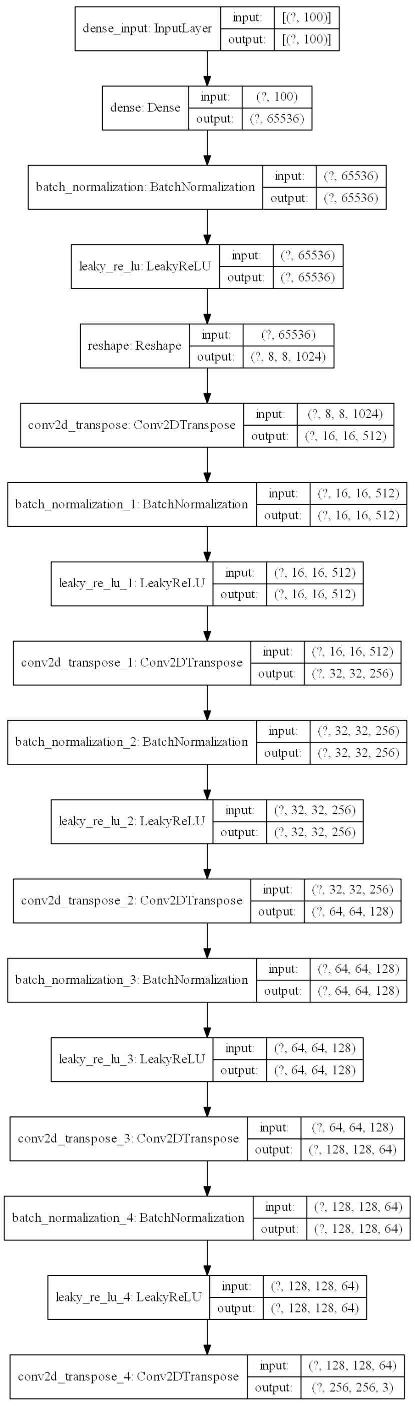

dense_input: InputLayer | input: [(?, 100)] | output: [(?, 100)]
dense: Dense | input: (?, 100) | output: (?, 65536)
batch_normalization: BatchNormalization | input: (?, 65536) | output: (?, 65536)
leaky_re_lu: LeakyReLU | input: (?, 65536) | output: (?, 65536)
reshape: Reshape | input: (?, 65536) | output: (?, 8, 8, 1024)
conv2d_transpose: Conv2DTranspose | input: (?, 8, 8, 1024) | output: (?, 16, 16, 512)
batch_normalization_1: BatchNormalization | input: (?, 16, 16, 512) | output: (?, 16, 16, 512)
leaky_re_lu_1: LeakyReLU | input: (?, 16, 16, 512) | output: (?, 16, 16, 512)
conv2d_transpose_1: Conv2DTranspose | input: (?, 16, 16, 512) | output: (?, 32, 32, 256)
batch_normalization_2: BatchNormalization | input: (?, 32, 32, 256) | output: (?, 32, 32, 256)
leaky_re_lu_2: LeakyReLU | input: (?, 32, 32, 256) | output: (?, 32, 32, 256)
conv2d_transpose_2: Conv2DTranspose | input: (?, 32, 32, 256) | output: (?, 64, 64, 128)
batch_normalization_3: BatchNormalization | input: (?, 64, 64, 128) | output: (?, 64, 64, 128)
leaky_re_lu_3: LeakyReLU | input: (?, 64, 64, 128) | output: (?, 64, 64, 128)
conv2d_transpose_3: Conv2DTranspose | input: (?, 64, 64, 128) | output: (?, 128, 128, 64)
batch_normalization_4: BatchNormalization | input: (?, 128, 128, 64) | output: (?, 128, 128, 64)
leaky_re_lu_4: LeakyReLU | input: (?, 128, 128, 64) | output: (?, 128, 128, 64)
conv2d_transpose_4: Conv2DTranspose | input: (?, 128, 128, 64) | output: (?, 256, 256, 3)

Figure 22: Network architecture of the generator in the Vanilla-GAN.

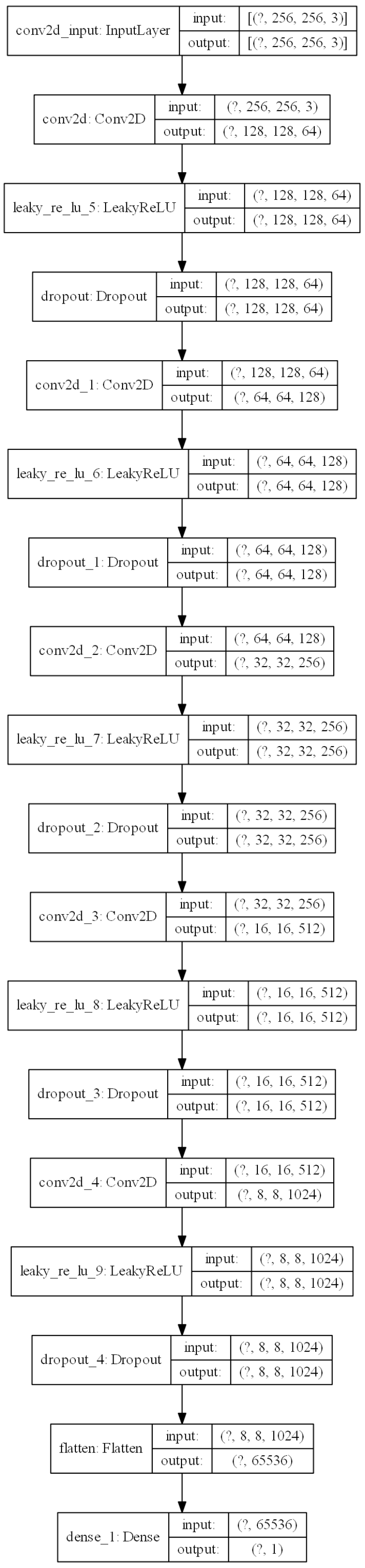

conv2d_input: InputLayer | input: [(?, 256, 256, 3)] | output: [(?, 256, 256, 3)]
conv2d: Conv2D | input: (?, 256, 256, 3) | output: (?, 128, 128, 64)
leaky_re_lu_5: LeakyReLU | input: (?, 128, 128, 64) | output: (?, 128, 128, 64)
dropout: Dropout | input: (?, 128, 128, 64) | output: (?, 128, 128, 64)
conv2d_1: Conv2D | input: (?, 128, 128, 64) | output: (?, 64, 64, 128)
leaky_re_lu_6: LeakyReLU | input: (?, 64, 64, 128) | output: (?, 64, 64, 128)
dropout_1: Dropout | input: (?, 64, 64, 128) | output: (?, 64, 64, 128)
conv2d_2: Conv2D | input: (?, 64, 64, 128) | output: (?, 32, 32, 256)
leaky_re_lu_7: LeakyReLU | input: (?, 32, 32, 256) | output: (?, 32, 32, 256)
dropout_2: Dropout | input: (?, 32, 32, 256) | output: (?, 32, 32, 256)
conv2d_3: Conv2D | input: (?, 32, 32, 256) | output: (?, 16, 16, 512)
leaky_re_lu_8: LeakyReLU | input: (?, 16, 16, 512) | output: (?, 16, 16, 512)
dropout_3: Dropout | input: (?, 16, 16, 512) | output: (?, 16, 16, 512)
conv2d_4: Conv2D | input: (?, 16, 16, 512) | output: (?, 8, 8, 1024)
leaky_re_lu_9: LeakyReLU | input: (?, 8, 8, 1024) | output: (?, 8, 8, 1024)
dropout_4: Dropout | input: (?, 8, 8, 1024) | output: (?, 8, 8, 1024)
flatten: Flatten | input: (?, 8, 8, 1024) | output: (?, 65536)
dense_1: Dense | input: (?, 65536) | output: (?, 1)

Figure 23: Network architecture of the discriminator in the Vanilla-GAN.

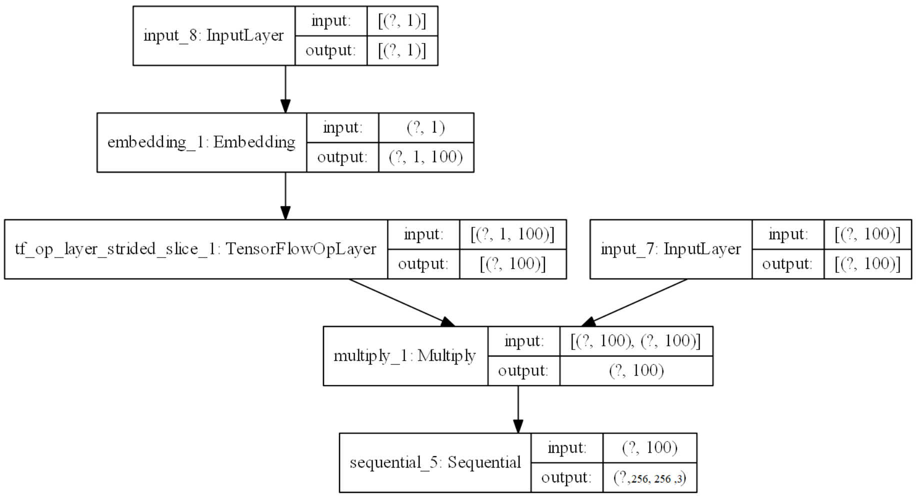


Figure 24: Network architecture of the generator in the AC-GAN.

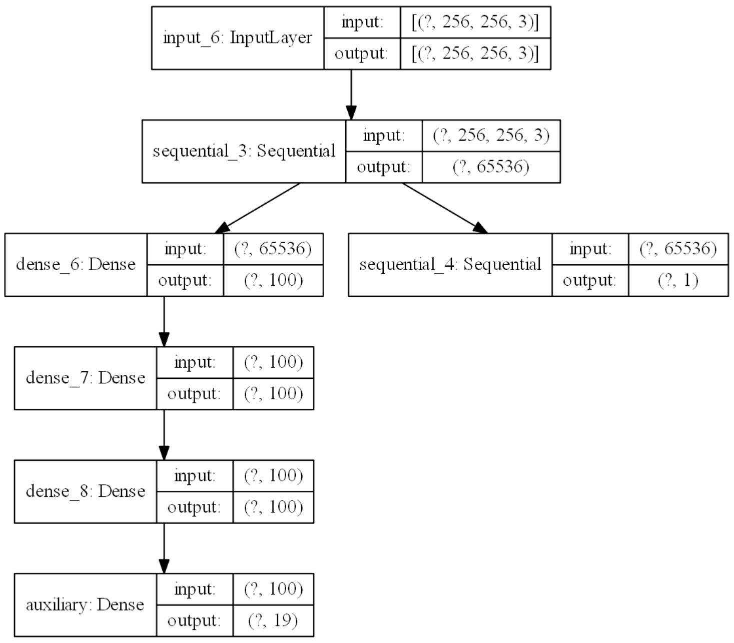


Figure 25: Network architecture of the discriminator in the AC-GAN.

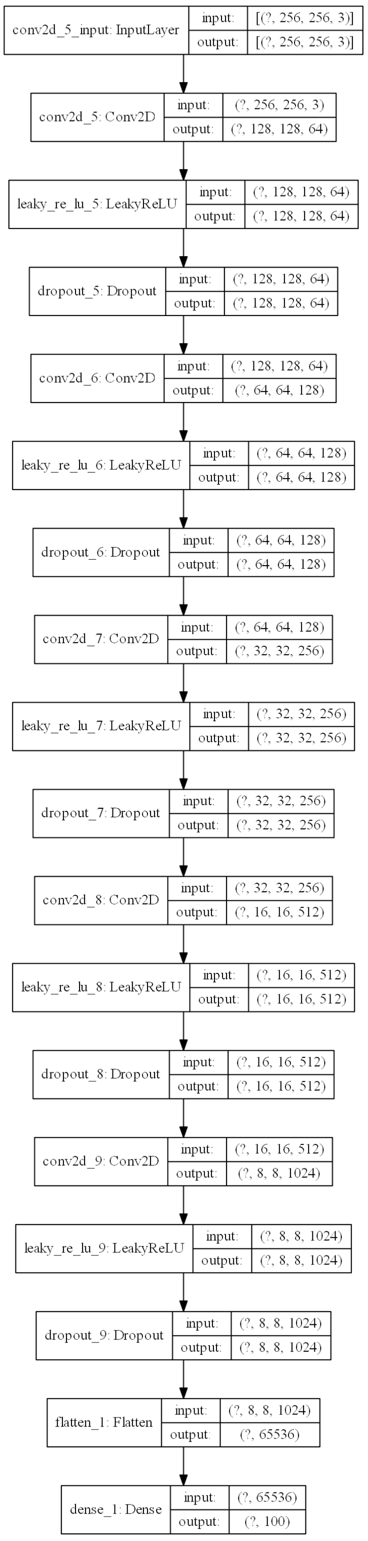


Figure 26: Network architecture of the image-to-latent network