Johannes Kepler University in Linz

Institute for Machine Learning

and

University of South Bohemia in České Budějovice

Faculty of Science

Generative Adversarial Networks and Applications in Bioinformatics

Bachelor Thesis

Author: Nikita Kolesnichenko
Supervisor: Univ.-Prof. Dr. Sepp Hochreiter
Guarantor: Ing. Ph.D. Rudolf Vohnout

České Budějovice ,2021

BIBLIOGRAPHICAL DETAIL

Kolesnichenko, N., 2021: Generative Adversarial Networks and Applications in Bioinformatics. Bachelor Thesis, in English. – 45 p., Institute for Machine Learning Johannes Kepler University, Linz, Austria

ANNOTATION

Generative Adversarial Networks (GAN) are currently considered a state-of-the-art method for image generation. Recently, Deep Convolutional Generative Adversarial Networks (DCGAN) yielded promising results in protein contact maps generation. The algorithm generated realistic protein structures, which were less erroneous than previously used generative methods. However, DCGAN is notorious for being hard to train due to the limitations of its loss function and complications in optimization. Wasserstein Generative Adversarial Networks (WGAN) was proposed, employing the Wasserstein loss function that stabilizes training and alleviates some of the DCGAN's training problems. In this thesis, a hyperparameter grid search for DCGAN and WGAN was conducted on the CIFAR-10 dataset. Runs with different hyperparameters were compared using Fréchet Inception Distance to determine whether WGAN is more stable than DCGAN.

DECLARATION

Place, Date Signature

Linz, 15.11.2021
……………………… [illegible]………………………

Nikita KOLESNICHENKO

ABSTRACT

Generative Adversarial Networks (GAN) have become a widely used tool for generating new instances, such as images, sound, or text. Judging by the output quality and diversity, they are substantially better than previously introduced generative methods, such as variational autoencoder (VAE). At the same time, they are not so simple to train and difficult to evaluate. Recently, GANs have shown promising results in the generation of *de novo* protein maps that can be translated into 3D protein structures. In the future, this could allow bioinformaticians and biochemists to create novel biological and chemical structures faster and cheaper than by utilizing current state-of-the-art models that require a lot of computational power and time. The theoretical part of this thesis gradually introduces the reader to the basic concepts behind GANs and briefly explains concepts from biology needed to understand protein contact map generation. The practical part of this thesis conducts a hyperparameter grid search with two well-known types of GANs – Deep Convolutional GAN (DCGAN) and Wasserstein GAN (WGAN) by training them on the CIFAR-10 dataset. This experiment's intention is to find out which GAN type is more stable and how Wasserstein distance and Jensen-Shannon divergence differ. The difference in performance between DCGAN and WGAN is compared using Fréchet Inception Distance. Finally, future work on the topic and the usage of GANs in protein generation tasks will be discussed.

Table of contents

1. Introduction

In the Information age, the amount and complexity of the data increased significantly, the variability of information available to us has become enormous, in all the different forms and shapes, be it text, music, images, or videos. Machine Learning specializes in techniques and models that can learn from data and then utilize the acquired knowledge to produce meaningful insights based on the learned information. Many different algorithms have been developed that are now widely used in business [1] and in academic research [2]. Nowadays, an extremely powerful subset of Machine Learning - Deep Learning (DL), which is based on the utilization of artificial neural networks, is actively leveraged by famous companies and institutes worldwide. Thanks to neural networks, it is possible to learn from high-dimensional data and use the knowledge for different purposes.

Deep Learning provides the ability to tackle various tasks, like classification, computer vision, natural language processing and many more. Unsurprisingly, neural networks can be used for generative tasks too. Among generative DL models, Generative Adversarial Networks (GANs) [3] have a special place. Initially, GANs were used for image generation and yielded good results. Jaydeep et al. [4] compared the Deep Convolutional GAN (DCGAN) [10] architecture to Variational Autoencoder (VAE) [5], which is a type of generative network, that consists of two parts – encoder and decoder. The encoder part of the network compresses input into a so-called „code " and decoder part „reconstructs" input from the reduced representation. Compared to VAE, images generated by GANs were better in quality. Specifically, they were less blurry and more diverse than images generated by autoencoder. Eventually, GANs became very popular among the science community due to the possibility of generating different types of data, such as high-resolution images [6], text [7] or sound [8].

GAN is especially fascinating because of the way it generates data. It consists of two networks – discriminator and generator, that train in parallel by optimizing a minimax learning task, where one network tries to overcome the other. The generator's task is to produce data that looks as real as possible, while the discriminator's task is to discern between the real data and the data produced by the generator. The goal of training GANs is to train a generator that can generate samples that would match the distribution of real dataset, and as a consequence, the discriminator would not be able to discern between real and fake data.

GANs can be applied to various tasks, from data augmentation to generating novel biological or chemical substances. Recently, in the year 2018, Anand et al. [9] used GANs to generate proteins represented as contact maps. Currently, GAN model architecture can be considered a state-of-the-art for generative tasks in Machine Learning. However, GANs have several shortcomings, including their tendency to be unstable during training, they might be very time and resource-consuming to train, and it is generally challenging to evaluate their performance numerically. As with many other deep learning models, GANs are prone to gradient vanishing, a problem first discussed in [10]. Gradient vanishes when after backpropagation, it is so small that it barely changes parameters‘ values, and the model hardly learns anything.

The evaluation problem of the GANs persists and is a research subject. On the one hand, results‘ quality and diversity can be assessed by visually inspecting generated images, but this might take a long time if the amount of generated instances is high. On the other hand, there must be a quantitative method that can automate performance evaluation and give meaningful feedback, which correlates well with human expert’s opinions [11]. Only a few evaluation methods can give meaningful insights into how well GANs are trained [12], or rather how real generated images look.

In the Foundations and Background part of this thesis, the main concepts underlying conducted experiments will be outlined - particularly several topics from Deep Learning, including neural networks, several variations of GANs such as Vanilla (GAN), Deep Convolutional (DCGAN) [13], and Wasserstein (WGAN) [14]. Some of the existing evaluation metrics, Inception Score (IS) [15], Fréchet Inception Distance (FID) [16], as well as biological background regarding protein generation, will be provided.

In the practical part, two well-known GAN architectures WGAN and DCGAN will be trained on CIFAR-10 dataset several times with different hyperparameters. Performance of both models will be compared to find out whether WGAN performs better than DCGAN or vice versa.

2. Foundations and Background

2.1. Neural Networks

Deep Learning (DL) is a subset of Machine Learning (ML) that utilizes Neural Network (NN) algorithm. NN can also be called a Multi-Layered Perceptron (MLP), an acyclic graph with several layers between the input and output that pass information in a feedforward fashion. However, not all the NNs are acyclic. For example, there is a subset of NNs called Recurrent Neural Networks (RNNs), which have cyclic connections in the hidden layers. The amount of hidden layers in between input and output is arbitrary, so, in general, when one refers to NNs with two or more hidden layers, they are often called Deep Neural Networks (DNNs). Any state-of-the-art DL model nowadays is based mainly on the utilization of DNNs.

2.1.1. Fully-Connected Neural Networks

Fully-Connected Neural Networks (FCNN) are the feedforward networks that pass information forward through all the layers to the output. Each FCNN consists of several layers of neurons: the computational units, each one of them is connected to all the units in the preceding and following layers, such that they form a chain, where each layer’s output is the next layer’s input. The amount of layers define the neural network's depth, and the amount of neurons in the layer defines the network’s width. The deeper and wider the network is – the more computations will be produced, the more memory will be required to store the network’s parameters (or weights), and the more complex the model will become.

Forward Pass: every neuron computes the sum of products of so-called weights and inputs from the previous layer and adds the bias constant to compute the preactivation value. This value then serves as an input to the activation function, which introduces nonlinearity, such that the network learns a more complex representation of data (shown in Figure 2.1). This chain of computations proceeds until the output layer is reached. Finally, the resulting value from the output layer is compared to the label by the loss function. This procedure is often called forward propagation or forward pass of the neural network.

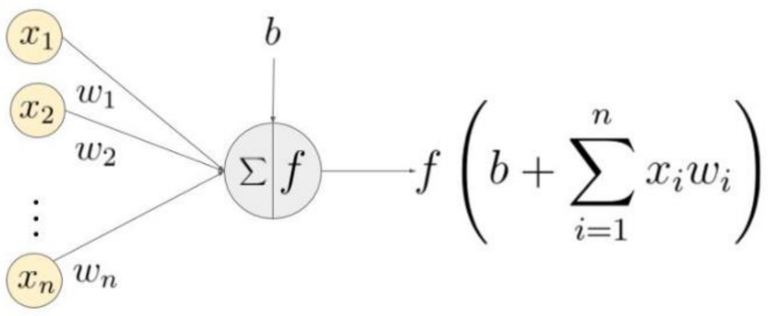

Figure 2.1: Overview of feedforward computation for a single neuron. Given input values x_n and weights w_n for each respective neuron and bias, activation of the sum of products of inputs and weights is computed for each neuron in the following layer [17].

As described by Nielsen et al. [18], to perform the forward pass, it is required to compute the vector of preactivation values $\boldsymbol{h}^l$, where superscript l stands for the number of layer. Then, pass $\boldsymbol{h}^l$ to the activation function f^l to acquire activations $\boldsymbol{a}^l$ for the respective layer. For each layer $l = 1,2,3,\ldots,L$ where $l = 1$ is an input layer, we compute

$$\boldsymbol{h}^l = \boldsymbol{W}^l \boldsymbol{a}^{l-1} + \boldsymbol{b}^l \tag{2.1}$$

$$\boldsymbol{a}^l = f^l(\boldsymbol{h}^l). \tag{2.2}$$

Backward pass (or backpropagation): is a crucial part of the neural network's learning process that starts when a forward pass is complete. First, the gradient of the cost function with respect to the weights of the output layer is computed. Then, the backpropagation algorithm iteratively proceeds to calculate the gradient with respect to every layer's weights until the input. The output is first compared to the true label to acquire output error $\boldsymbol{\delta}^L$, where superscript L is the last layer's number

$$\boldsymbol{\delta}^L = \nabla_{\boldsymbol{a}} C \odot f'(\boldsymbol{h}^L) \tag{2.3}$$

where $\nabla_{\boldsymbol{a}}$ is the vector of partial derivatives $\frac{\partial L}{\partial a_j^L}$, the operator $\odot$ is a Hadamard product (or element-wise multiplication), and C is the cost function. Then, the algorithm goes backward and computes the error $\boldsymbol{\delta}^l$ for each layer until it reaches the input. For each layer $l = L-1,\ L-2,\ldots,1$, we compute

$$\boldsymbol{\delta}^l = ((\boldsymbol{W}^{l+1})^T \boldsymbol{\delta}^{l+1}) \odot f'(\boldsymbol{h}^l) \tag{2.4}$$

where $(\boldsymbol{W}^{l+1})^T$ is a transposed weight matrix and $\boldsymbol{\delta}^{l+1}$ is a computed error of the previous layer (as we go backward from output to input). Finally, the gradient of the cost function with respect to a particular weight in the network can be computed

$$\frac{\partial C}{\partial \boldsymbol{W}_{jk}^{l}} = \boldsymbol{a}_k^{l-1}\boldsymbol{\delta}_j^{l} \tag{2.5}$$

where $\boldsymbol{a}_k^{l-1}$ is the activation input for a particular weight and $\boldsymbol{\delta}_j^{l}$ is the error of the output from particular weight. Both forward pass and backward pass computations are produced in a single epoch – pass of the whole dataset through the network. The amount of epochs is a hyperparameter that can be modified.

Gradient Descent (GD): Gradient-based learning is the typical optimization strategy for training neural networks. The goal of GD is to find the best possible parameters that minimize the loss function. Graphically, during training, GD makes steps downhill towards the minima in each iteration. The size of the step is defined by a learning rate, an important hyperparameter that defines the rate at which gradient descent makes steps towards minima. As described by Goodfellow et al. [19]. Given that our dataset is of size m, GD is computed as follows

$$g = \frac{1}{m}\sum_{i=1}^{m} \nabla_w C\left(x^{(i)}, y^{(i)}; w_t\right) \tag{2.6}$$

where C is a cost (or loss) function, x and y are single sample-label pairs from the dataset, and w are parameters (or weights). GD uses all the samples in the dataset to compute the gradient, which can be quite time-consuming to calculate if the dataset is large. Instead of computing gradient on the whole dataset, it is usually better to divide the dataset into randomly composed mini-batches of training samples and the gradient based on iterative approximation of these mini-batches. This type of GD is called Stochastic Gradient Descent (SGD). SGD might converge faster than GD, as it only uses a subset of all samples per iteration [20]. By using small batches of randomly picked samples from the training set of size m', the approximation of the gradient is,

$$g = \frac{1}{m'}\sum_{i=1}^{m'} \nabla_w C\left(x^{(i)}, y^{(i)}; w_t\right) \tag{2.7}$$

where C is a cost (or loss) function, x and y are single example-label pairs from the dataset, and w are parameters (or weights). Furthermore, the update for weights is defined as,

$$\theta := \theta - \eta g \tag{2.8}$$

where η is the learning rate.

2.1.2. Convolutional Neural Networks

Convolutional Neural Networks (CNNs), which were first proposed by Fukushima et al. [21], are best known for their parameter sharing feature, sparsity of connections and increased receptive field throughout the network.

The convolutional layer performs discrete convolution operation on the input (see Figure 2.2), such that the original size of the input is decreased, depending on the kernel and stride parameters. The kernel can be viewed as a *k*-dimensional matrix that "strides" across the input. Convolution operation for a 2D (e.g., image of size 64x64 pixels) input is formulated as

$$S[i,j] = (I * K)[i,j] = \sum_m \sum_n I[m,n]K[i-m,j-n] \tag{2.9}$$

where *S* stands for the output of convolution, *I* for an input matrix of size [*m*,*n*], *K* for a kernel matrix of size [*i*,*j*], and the asterisk symbol stands for a convolution operator.

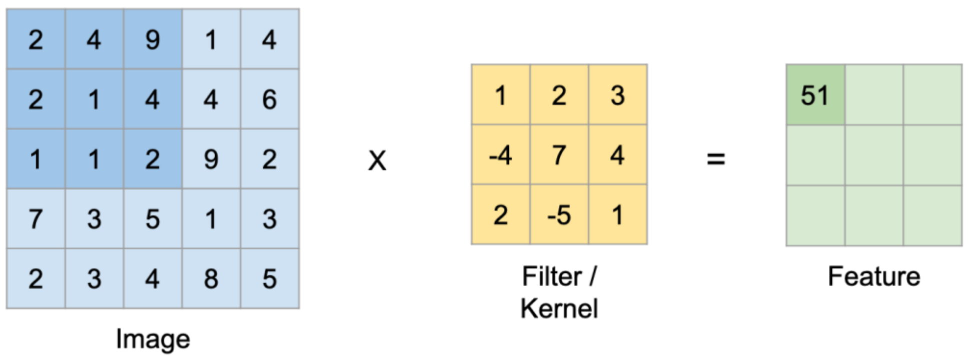

Figure 2.2: Convolution operation example. Image is represented as a matrix to which a convolutional filter is applied. Feature matrix values are calculated, as kernel slides across the whole image with predefined stride [22].

For example, suppose image input is passed to the 2D convolution function. By applying a kernel iteratively to every region of the image and performing convolution, we acquire a so-called feature map (shown in Figure 2.2). Because the values of kernel are not changed, these feature maps are essentially computed using the same parameters – this is a parameter sharing feature of CNN [23].

CNN captures small structures of an image, such as edges and curves on the earlier layers, and much more complex structures of an image on later layers, such as a human’s face or dog’s ears. Input consists of one or multiple local structures that might be important for learning the more complex structure of the input. If only some of the feature maps are important for learning, they are connected to the feature maps of the following convolutional layer. For comparison, fully-connected layers will see an input as a 1D vector of values, so that the receptive field of fully-connected layers is shallow, while convolutional layer allow for processing information presented as a multi-dimensional tensor.

The pooling layers are commonly used in CNNs. Pooling is performed similarly to convolution; the sliding kernel of predefined size goes over an image and computes some mathematical operation. Several pooling types exist, for example, max-pooling takes the maximum value, average-pooling takes the average of part of an image, to which kernel is applied. According to Goodfellow et al. [24], pooling, as well as convolution, creates representation, that is invariant to the small translations (e.g., object is shifted by a few pixels on the picture), so that it preserves the fact whether some feature of interest is present on the image or not. The downside of pooling is that we are losing some information due to the reduction of the image resolution.

2.2. Generative Adversarial Networks

This section is an introduction to Generative Adversarial Networks and gives an overview of a few GAN variations used in this work.

2.2.1. Vanilla GAN

First proposed by Goodfellow et al. [3], a GAN architecture consists of two networks – generative network G, which tries to match the distribution of real samples, and discriminative network D, which tries to correctly decide whether an input sample is from the real dataset or is a fake one. D and G are adversaries of each other, that constantly try to overcome the opponent. G can be presented as a fraudster that improves their forgery skills to make an expert, D, into thinking that fake art of G is real. D in the meantime improves its ability to recognize fraud. This game is going on until D can not confidently distinguish between fake and real samples [3].

More formally, GAN forms a structure (shown in Figure 2.3) from which G receives updates to increase its capability to produce more „real-looking" samples by learning from the experience of discriminator function $D()$, where $\mathbf{x}$ is a real samples vector, that tries to distinguish between fake and real samples. D assigns a probability $D(\mathbf{x}) \in [0,1]$ to each sample as output and tries to predict $G(\mathbf{z})$ samples correctly, where $\mathbf{z}$ is a random noise vector sampled from Gaussian Distribution. On the other hand, G repeatedly struggles to generate samples $G(\mathbf{z})$ to mislead D. Thus, G will get better at generation, while D will constantly be trying to improve at discrimination. Combining the generator's task to increase the probability of fake example to be labeled as real and the discriminator's task to distinguish between real and fake data as good as possible, the two networks have the following optimization objectives

$$\begin{aligned}
&\max_D \mathbb{E}_{\mathbf{x}\sim p_r(\mathbf{x})}[\log D(\mathbf{x})] + \mathbb{E}_{\mathbf{z}\sim p_z(\mathbf{z})}[\log(1-(D(G(\mathbf{z}))))] \\
&\max_G \mathbb{E}_{\mathbf{z}\sim p_\mathbf{z}(\mathbf{z})}[\log(D(G(\mathbf{z})))]
\end{aligned} \tag{2.10}$$

where $p_r(\mathbf{x})$ is the distribution of real samples and $p_z(\mathbf{z})$ is the distribution of latent vector. Both objectives can be summarized to

$$\min_G \max_D \ \mathbb{E}_{\mathbf{x}\sim p_r(\mathbf{x})}[\log D(\mathbf{x})] + \mathbb{E}_{\mathbf{z}\sim p_z(\mathbf{z})}\left[\log\left(1-\left(D\left(G(\mathbf{z})\right)\right)\right)\right], \tag{2.11}$$

where $\mathbb{E}_{\mathbf{x}\sim p_r(\mathbf{x})}$ is an expectation w.r.t. distribution of real samples and $\mathbb{E}_{z\sim p_z(\mathbf{z})}$ is an expectation w.r.t. distribution of latent vector. The first summand of the loss function represents log-probability that the discriminator predicts real data $\mathbf{x}$ as real correctly, the second summand represents the log-probability that the discriminator predicts generated data $G(\mathbf{z})$ as fake correctly. In this objective the discriminator tries to maximize classification accuracy, while the generator tries to minimize the discriminator's accuracy.

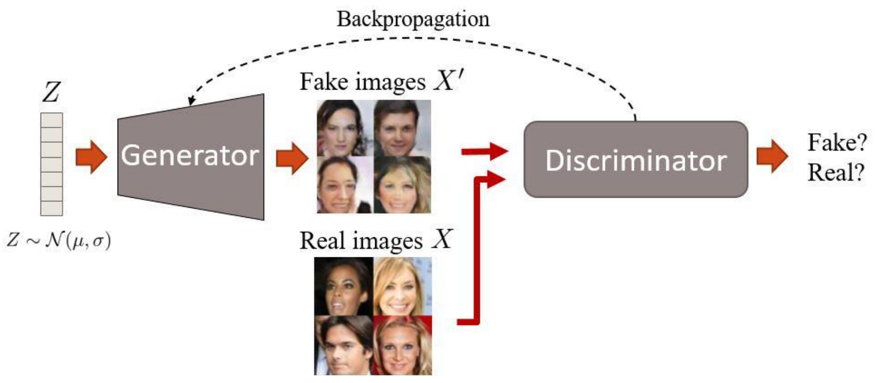

Figure 2.3: Overview of GAN training procedure on the example of face images dataset. Random noise vector **z** serves as an input for *G*. After *G* generates the output *G*(**z**), both real and generated images are passed to *D*, classifying the image as either fake or real. During backpropagation, weights learned by *D* are updated first, and *G* weights are updated based on a gradient from *D* [25].

GAN's loss function computes the similarity between generated data distribution and real data distribution. While minimizing the loss, we reduce the distance between two distributions (Figure 2.4).

GAN is a two-player game, where both players, in order to converge, should reach the state where they have a minimal loss (or a maximum payoff). Nash Equilibrium is a state in which none of the two players can improve their payoff even if one of them changes strategy. In our case, if both networks reached Nash Equilibrium - then both networks are trained to the state, where there is no further improvement possible. According to Salimans et al. [15], despite using optimization strategies such as gradient descent to minimize the loss, it is tough to reach even local Nash Equilibrium. Both players would instead increase each other's loss while trying to minimize their own and will likely settle on the local saddle point.

There are several known problems that can be encountered during the training of GANs [11]:

(1) **Mode collapse**: generator training collapses and it starts to produce the output of the same type (or class). The generated instances lose diversity and the generator itself is not learning the real distribution but rather improves on a small subset of samples as much as possible.

(2) **Vanishing gradient**: an overtrained discriminator does not give much feedback to the generator. Discriminator gets so good at discerning real from fake that the generator's gradient diminishes fast and model stops learning.

(3) **Non-convergence**: two players will usually fail to reach Nash Equilibrium. As previously mentioned, generator's goal is to generate objects that look realistically. Therefore, when the generator is trained well, discriminator starts to give worse feedback, as it might not distinguish between real and fake well. Consequently, the generator might start to adapt to the discriminator's „random coin flips" and degrade.

(4) **Difficult evaluation**: up to the date this thesis was written (Summer, 2021), there is no universal evaluation method that can be used for comparison of different GAN architectures. Existing ones often depend on the problem domain and dataset that GANs were trained on [12].

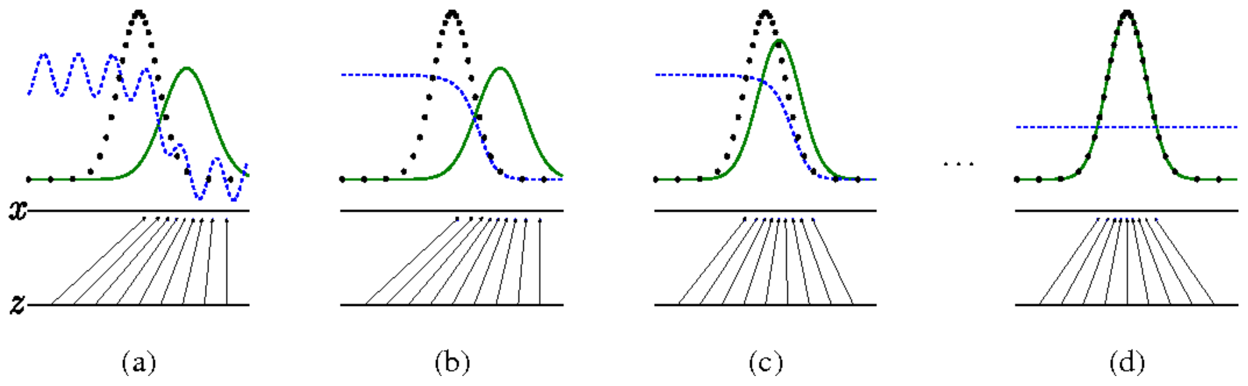

Figure 2.4: Distribution of generated images (green line) gradually matches distribution of real images (black dotted line) as training proceeds. The black dotted line represents the real distribution, the green line is the generated distribution and the blue line is a discriminator's outputs distribution [3].

2.2.2. Deep Convolutional GAN

Deep Convolutional Generative Adversarial Network (DCGAN) architecture was first proposed by Radford et al. [13] to modify the Vanilla architecture. DCGAN consists of convolutional layers only. Convolutional layers offered benefits such as the decreased amount of learnable parameters and increased receptive field (see subsection 2.1.2. for the explanation of those concepts), which in theory could lead to faster training and better image quality. General architectures of discriminator and generator are shown in Figure 2.5.

As stated in the original paper, the discriminator uses convolutions with stride parameters greater than 1, which is used for image downsampling. The discriminator also uses LeakyReLU [26] activation functions (except the output layer, which uses the Sigmoid activation function with formula $f(x) = \frac{1}{1+e^{-x}}$, to label images as fake or real) to downsample and classify input images. The generator uses transposed convolutions (deconvolutions) with ReLU [27] activation functions (except the output layer, which uses the hyperbolic tangent activation function with formula $f(x) = \frac{1-\exp(-2x)}{1+\exp(-2x)}$) to upsample the input to the size of an entire image, say of 64x64 pixels in size (as shown in the Figure 2.5). Both networks use Batch Normalization [28] between convolution layers to stabilize learning and normalize batches to have zero mean and unit variance. The optimization strategy of choice was Adam (adaptive moment estimation) [29], with a learning rate of 2×10^{-4} for both discriminator and generator.

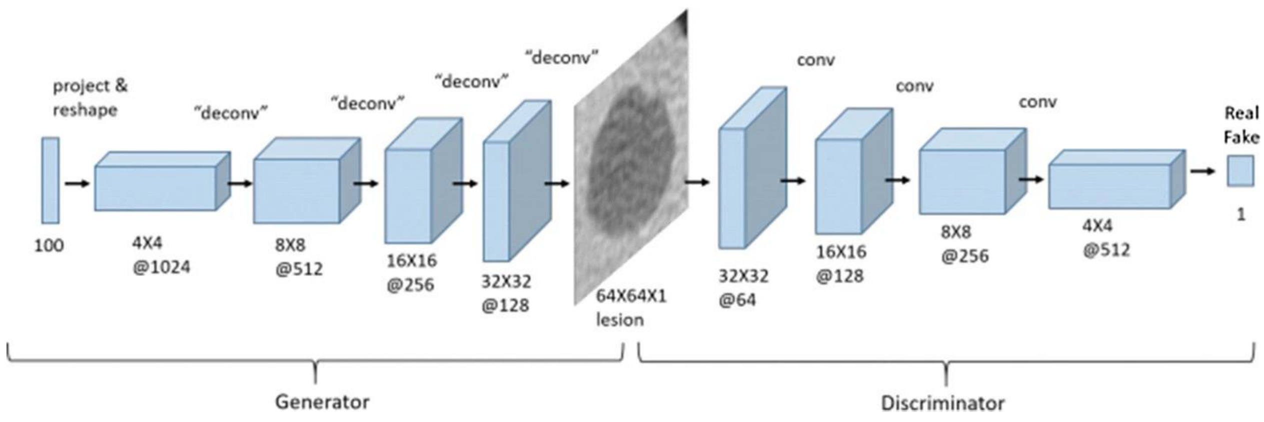

Figure 2.5: Example architecture of DCGAN, used for liver lesion medical image generation [30]. Image from [31].

In comparison to Vanilla GAN, DCGAN's generator produced pictures of higher quality and offered better memory efficiency due to the usage of convolutional layers. Generally, due to improvements in the architecture, such as using Batch Normalization, ReLU and LeakyReLU activation functions, training became more stable. However, the cost function did not change, so DCGAN still suffered from problems similar to Vanilla GAN (subsection 2.2.1. page 10). Furthermore, the DCGAN authors stated that the discriminator's learned features could be further used for the classification tasks on comparable datasets.

2.2.3. Wasserstein GAN

Wasserstein GAN (WGAN) was first proposed by Arjovsky et al. [14]. This variant of GAN stands out from previously described types with the modified algorithm for learning, which is based upon the Earth Movers (EM) distance concept, also called Wasserstein-1 distance. By definition of [32], EM distance is the minimal cost that must be paid in order to transform one distribution into another. The GAN (equivalently DCGAN) architecture is based on the Jensen-Shannon (JS) divergence, which may fail to converge if distributions are disjoint. On the other hand, EM distance can give a scalar value representing the distance between two disjoint distributions [14,33,34].

In WGAN, the discriminator is called „critic“ and there is no Sigmoid activation after the last layer anymore, so the output of the critic is a scalar value that shows how „real“ the input image is. Another advantage of WGAN over previously described architectures is that WGAN’s training algorithm employs more update steps for the critic than for the generator. Since the generator is slowed down, the discriminator has a chance to learn better data representation so that generator has better feedback in the form of gradients from the discriminator [16].

Given that p_r is a probability distribution of real samples, p_g is a probability distribution of generated samples, new critic‘s task is to maximize the difference between distributions as much as possible (increase the distance), and the generator’s task is to minimize the difference between these distributions as much as possible (decrease the distance). The loss function of WGAN quantitatively shows how near is the generated distribution to the real distribution. Loss function of WGAN, based on Wasserstein-1 distance is

$$W(P_r, P_g) = \sup_{\|D\|_L \leq 1} \underset{\tilde{\boldsymbol{x}} \sim p_g}{\mathbb{E}}[D(\tilde{\boldsymbol{x}})] - \underset{\boldsymbol{x} \sim p_r}{\mathbb{E}}[D(\boldsymbol{x})] \tag{2.12}$$

where $\tilde{\boldsymbol{x}}$ denotes generated sample, and $\boldsymbol{x}$ denotes real sample. $\| D \|_L \leq 1$ is the 1-Lipschitz constraint on the discriminator. We will focus on two approaches to enforce 1-Lipschitz constraint: weight clipping and gradient penalty (gradients of the WGAN model using these methods are shown in Figure 2.6).

Weight clipping: simple approach, which introduces new hyperparameter c and requires us to clip discriminator‘s weights within $[c, -c]$ range. As the authors of the original publication [9] state, weight clipping is not a great option to enforce 1-Lipschitz constraint because if we choose a large clipping value, it might take longer for weights to approach their limit. On the other hand, if we choose a very small clipping value, this can lead to weights being limited to very low values and thus to vanishing gradient. Therefore, c should be appropriately chosen. In addition, weight clipping does not allow generator to learn complex structures [9].

Gradient penalty: a more challenging to implement but more stable approach, which was first introduced by Gulrajani et al. [35] in their publication Improved Training of Wasserstein GANs. The authors showed, that WGAN with Gradient penalty is more stable during training and has more capacity for learning complex structures. Instead of directly applying constraints on the model‘s weights, this technique penalizes any weight values that move away from L2-norm of 1. The point on the line between $\boldsymbol{x}$ and $G(\mathbf{z})$ for a given e in the range [0,1] is defined as

$$\hat{\boldsymbol{x}} = e\boldsymbol{x} + (1-e)G(\mathbf{z}) \tag{2.13}$$

where $\hat{\boldsymbol{x}}$ - is a weighted average between real and fake samples. Loss function with gradient penalty is defined as

$$L = \underset{\tilde{x}\sim\mathbb{P}_g}{\mathbb{E}}[D(\tilde{\boldsymbol{x}})] - \underset{\boldsymbol{x}\sim\mathbb{P}_r}{\mathbb{E}}[D(\boldsymbol{x})] + \lambda \underset{\hat{x}\sim\mathbb{P}_{\hat{x}}}{\mathbb{E}}\left[\left(\|\nabla_{\hat{x}}D(\hat{\boldsymbol{x}})\|_2 - 1\right)^2\right] \tag{2.14}$$

where the second summand is the gradient penalty constraint and the first summand is the WGAN loss.

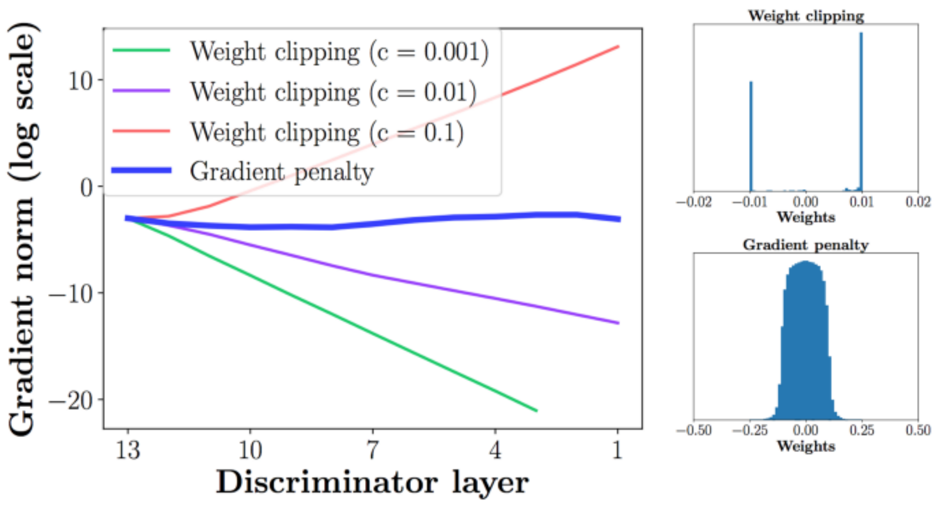

Figure 2.6: Gradient and weights behavior with proposed strategies to enforce 1-Lipschitz constraint. It can be seen, that weight clipping approach requires careful selection of c constant and even then weights’ distribution is mostly accumulated at boundary values of $[c, -c]$ range, while gradient penalty’s weights are normally distributed on the whole range [35].

2.3. Evaluation Methods

The problem of comparing Generative Adversarial Networks architectures is present to this day [12]. An enormous amount of GAN variations have been proposed since the Vanilla version, which improves the training process and quality of the results. However, there are not so many techniques that exist to evaluate the results quantitatively. This section presents two quantitative methods for evaluating generated samples produced by GANs and introduces the Inception network architecture, which is a key component for both evaluation methods.

2.3.1. Evaluation Difficulties

Since we generate completely new samples, there must be a way to compare generated and real samples quantitatively, since qualitative estimation (e.g., visual inspection of the image) does not take into account how well the generated distribution approximates the real distribution. Such a quantitative measure should consider how good the distribution of generated samples represents the real one, estimate quality and diversity of generated samples, and correlate well with human experts' opinions [11]. Borji et al. [12] compared 24 quantitative and five qualitative evaluation methods and analyzed their positive and negative sides. The author states that every method is applicable to a specific case, and there is no universal and reliable metric yet that could be applied to GANs independently of what kind of dataset is used and for which problem domain generation is done. Therefore, even with all available methods, a universal evaluation method for GANs is still to be discovered.

2.3.2. InceptionNet

InceptionNet is a deep neural network architecture proposed by Szegedy et al. [36]. It is used by some evaluation methods for Generative Adversarial Networks, such as Inception Score (IS) [15] and Fréchet Inception Distance (FID) [16]. This architecture uses several techniques that allow increasing the network’s depth and width. For instance, it utilizes Inception Modules, which execute convolutional layers with different filter sizes and/or maximum pooling layers in parallel to stack up to their outputs into one mixed output representation. Mixing the outputs of convolutions of different sizes led to the efficient increase of the perception field throughout the network. However, it is computationally expensive to naively utilize these modules (i.e., concatenating resulting feature maps without using any bottleneck filtering) because the features maps will stack and increase the network's width. Convolution with a 1x1 kernel size is used as a bottleneck to avoid this by significantly reducing the number of learnable parameters.

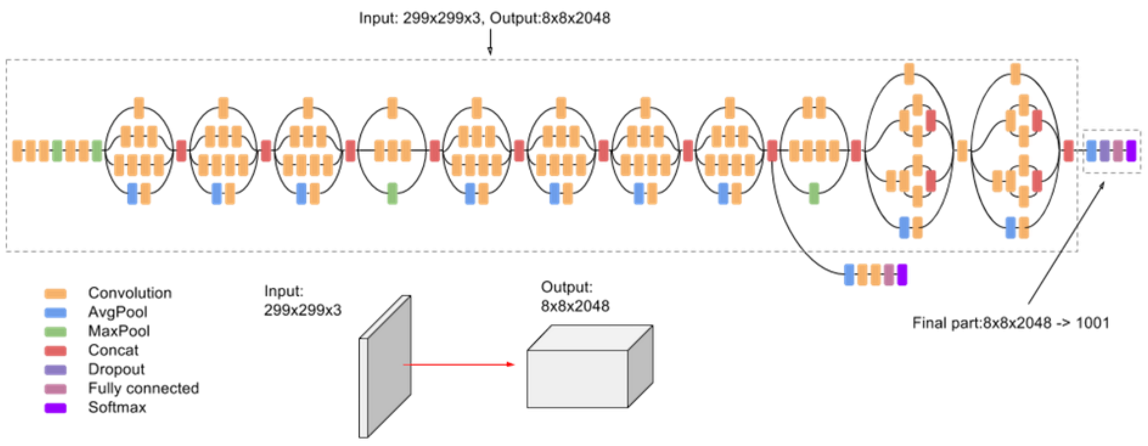

Figure 2.7: InceptionNet architecture. The central part of the network is comprised of blocks of convolution and pooling layers. Final part consists of average pooling, dropout and fully-connected layer with softmax activation function [37].

Inception-v3 [37] architecture is the upgraded version of previous Inception-v2. This architecture is widely known for its usage on ILSVRC (ImageNet Large Scale Visual Recognition Competition) in 2015 [38], where this network architecture was the first runner-up in the image classification task. Since then, Inception-v3 pre-trained on the ImageNet dataset, which features 1000 different classes and 14 million images, can be used for different tasks by transfer learning (i.e., use the network's pre-trained weights for a different task or dataset).

2.3.3. Inception Score

Inception score (IS) is an evaluation method proposed by Salimans et al. [15] that allowed to give a quantitative estimation of how well a generated image resembles properties of a real image. The authors developed IS to automatically assess the quality and diversity of a set of generated images without human expertise. This evaluation method has „Inception“ in its name because of using output probabilities of pre-trained Inception network [36] in its calculation.
To evaluate an images' quality and diversity, we must calculate the entropy of conditional probabilities (that particular image belongs to a particular class) and marginal probability of predicted classes are computed. Lower the entropy of conditional class distribution – higher the fidelity and thus higher the generated image's quality. Higher the entropy of marginal probability of predicted classes – the more diverse generated image set. All forementioned computations are combined into the formula,

$$IS = \exp\left(\mathbb{E}_{x\sim p_g} D_{KL}(p(y|x)||p(y))\right) \tag{2.15}$$

where $\mathbb{E}_{x\sim p_g}$ is an expectation w.r.t. distribution of generated samples. Exponent is intended to increase values‘ magnitude. In general, if generated images are both having realistic features and are diverse, IS should be large.

Although the Inception score is well correlated with human expert opinion, it is often not the best choice as an evaluation metric. Barrat et al. [39] questioned the Inception score's accuracy, stating that it has some critical issues. For example, IS only measures the quality and diversity of generated images without comparing them to real images. As a result, IS will be large for a generator that learned to produce diverse and high-quality images instead of a generator that was trying to resemble a real distribution. Additionally, IS is sensitive to random noise, such as slight differences in weights for the same Inception network, implemented in different Deep Learning frameworks. These slight weight differences can significantly affect IS, and the resulting score will differ from one model to another [39]. Finally, if the generator was trained to produce images that are different from the ImageNet dataset, the Inception score can show confusing results. Other datasets may contain images of different domains. For example, CelebA dataset consists of face images, so the domain is quite different from ImageNet. Therefore, generated images from the generator trained on CelebA may be mislabeled by InceptionNet. Due to that, the entropy of distribution conditioned on the random image from CelebA must be higher than one conditioned on the random image from ImageNet [39].

2.3.4. Fréchet Inception Distance

Fréchet Inception Distance (FID) was first introduced by Heusel et al. [16]. FID features a direct comparison of fake and real distributions using the Wasserstein-2 distance. We also seek the Gaussian distribution for both real and generated samples, as it is a prerequisite for the calculation of Fréchet distance. Generally, the distribution of real samples is unknown, so it is better to assume Gaussian for both real and generated distributions, because according to [16], the meaningful way to represent two unknown distributions is to represent them as maximum entropy probability distributions (i.e., Gaussians).

FID formula utilizes Inception Network’s coding layer (Average Pooling’s activations) to compute the distance. For the calculation of FID, one would first need two sample sets from the real and generated distribution. By calculating both sampled distributions‘ statistics, like means and correlation matrices, we calculate:

$$d^2\big((m, C), (m_w, C_w)\big) = \|m - m_w\|_2^2 + \mathrm{Tr}\,\big(C + C_w - 2(CC_w)^{1/2}\big) \tag{2.16}$$

where m, C stand for the mean vector and covariance matrix of generated samples, m_w, C_w are the mean vector and covariance matrix of real data samples, and Tr is a trace operator. Low FID implies high similarity between two distributions. The authors of the original paper showed that FID is able to respond to various disruptions on the images, such as added random noise or blur, with a larger distance, while IS mostly responds poorly or does not change the score much. Furthermore, FID responds with a larger distance to mode collapse (section 2.2.1.) caused by the generator. The main advantage of FID over IS is the possibility to compare real and generated distributions, instead of only estimating quality and diversity of generated images.

2.4. Proteins

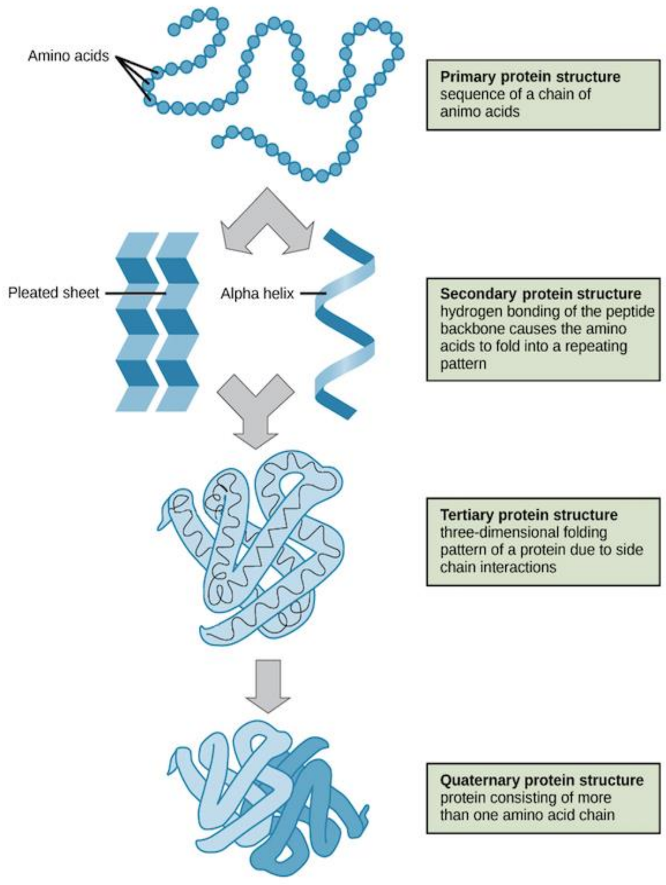
Amino acids
Primary protein structure
sequence of a chain of animo acids
Pleated sheet
Alpha helix
Secondary protein structure
hydrogen bonding of the peptide backbone causes the amino acids to fold into a repeating pattern
Tertiary protein structure
three-dimensional folding pattern of a protein due to side chain interactions
Quaternary protein structure
protein consisting of more than one amino acid chain

Figure 2.8: Four subsequent protein structures. Primary structure is a basic sequence of amino acids formed into a chain, also called polypeptide. Secondary structure comprises several polypeptides to form a so-called alpha helix or beta sheet. Tertiary structure is made of several alpha helices and beta sheets, folded into three-dimensional structure. Finally, quarternary protein structure is a complex of several tertiary proteins [30].

Proteins are the building blocks of living organisms. They play many essential roles, from forming the muscular and skin tissues to delivering nutrients to different parts of the body. They are also in charge of DNA replication, as well as maintenance. Proteins are made up of amino acid chains that are called polypeptides. Polypeptides are specific to each protein, which means that sequence of amino acids that protein is built from is predefined by the gene that codes for a particular protein. Gene regions are three-nucleotide combinations that are called codons, and they serve as specific codes for a particular amino acid. Considering that there are four different nucleotides in the DNA, there are 64 possible combinations of nucleotides, but not all of them code for amino-acid. Some of the codons code for start signal (AUG) and three others that code for stopping signal (UAG, UAA, UGA). There are twenty types of amino acids that can form the polypeptide chain so the possible amount of combinations of amino acids in those chains is enormous - 20^n combinations of amino acids, where *n* is the amount of residues in polypeptide chain [40].

Proteins have four hierarchically ordered structures (Figure 2.8): primary, secondary, tertiary and quaternary. The primary form is the set of amino acids, which describe the unique ordering of amino acids in the particular polypeptide chain. The secondary structure consists of a helical (alpha-helix structure) or a folded (beta-pleated sheet structure) polypeptide chain. The tertiary structure is a structure that consists of alpha helices and beta sheets bonded together by hydrophobic interactions and hydrogen bonding. Finally, quaternary structures are complexes made up of several tertiary structures [41].

In eukaryotic organisms, protein is biosynthesized in two main processes, called transcription and translation. First, enzyme RNA polymerase attaches to the promoter region of DNA and unzips double helix into two single-strands. Then, polymerase reads the template strand and elongates the complementary mRNA strand until the termination phase. That strand is then undergoes preprocessing. After preprocessing, mRNA is bound to several transport proteins in order to transport mRNA out of the nucleus to the cytoplasmic reticulum of the cell [42]. Outside of the nucleus, mRNA should reach the ribosome, which is the natural factory of polypeptides production. When mRNA enters the ribosome, it starts „reading“ the nucleotide triplets. tRNA with a specific amino acid that matches triplet binds to the ribosome complex and adds the amino acid into the chain. This process keeps going until codon coding for the stopping signal is reached. The resulting polypeptide chain is a primary structure of a protein. This structure is then either kept to be combined with other polypeptides or is folded into higher-order structures [43].

2.5. Protein Design

The protein structure design involves first acquiring folded protein structure and then inferring its amino acid sequence. Generally, when designing a new protein, we seek the amino acid sequence as well as the protein backbone structure that will fit some predefined architecture [44]. This process is called *de novo* protein design, meaning that a completely new protein with desired properties is designed. If the protein structure is known, then the task is to design an optimal amino acid sequence that codes for a particular structure. Amino acid sequence is optimal, if it codes for the same (functioning) protein. This sequence can then be reverse-translated to the DNA sequence, and this DNA can be produced artificially. Nowadays, protein design is accomplished using various algorithms and systems. For example, a popular choice for protein structure design is Rosetta@home (Rosetta) [45]. In addition, probabilistic algorithms, such as

Hidden Markov Model (HMM), were used for structure prediction. Furthermore, Deep Learning approaches, such as GANs and recurrent neural networks with Long-Short Term Memory (LSTM) units [46,47], were found to be useful for structure prediction and protein generation tasks.

2.5.1. Known Approaches

Several *in silico* (from Latin „in silicon“, a study performed using computational machines) methods of protein design exist that do not utilize Machine Learning algorithms. One example is Rosetta@home [45], a distributed computational system used to predict and design new protein structures. Rosetta is based upon calculating the protein backbone and conducting a heuristic search of the most probable polypeptide sequences that fit the calculated backbone structure [48]. As of March 25th, 2021, around 87,000 active hosts contribute their idle computational powers to the system. However, even with all the power this system provides, it is time-consuming and resource-demanding to design a new structure using Rosetta because of the algorithm’s heuristic nature, which uses its energy function [49] to match a conformation with the lowest energy for a particular protein. According to the thermodynamic hypothesis proposed by Anfinsen et al. [50], the native tertiary structure of the protein is the one that has the lowest amount of Gibbs free energy, or the lowest amount of the thermodynamic work needed to drive a chemical reaction [51]. We seek for the native structure, as by definition [52] it is an undamaged, stable, and functional protein structure. The search space for such a structure is rather large - for n amino acids in the polypeptide chain coding for a protein, there are 20^n combinations of amino acids. In order to run such search, a vast amount of computational power is required, because amount of possible sequences grows exponentially depending on the amount of residues in the polypeptide.

Probabilistic models, such as TorusDBN [47], also found their place in protein design. Initially, this model was used for protein structure prediction, which is essentially a reversed protein design task, that is, predicting protein structure from amino acid sequence. TorusDBN is a HMM based approach that generates a backbone structure based on torsion angles of alpha carbons, specifically phi-angles and psi-angles. These angles can estimate interactions between residues in the protein structure and therefore predict which secondary structure polypeptide chain will form. Sampled protein distribution is then compared to the real protein distribution by using KL divergence.

When the amino acid sequence is successfully designed, it is then possible to synthetically produce the protein. First, the DNA sequence is reconstructed from polypeptide chain. Second, the DNA is sliced into the smaller overlapping pieces for easier synthesis. Finally, overlapping sequences are produced and connected into larger pieces [53]. This process is called DNA synthesis, the end product of synthesis is recombinant DNA (rDNA), which will serve as a

blueprint for the synthesis of the desired protein. Most of the known *in vitro* (from latin „in the glass“, a study performed in the laboratory conditions) protein synthesis approaches require protein to be naturally reproducible by selected microorganisms or at least coded in DNA, either artificially made or natural. Two of the well-researched methods are cell-based and cell-free.
(1) The cell-based approach uses the protein synthesis system of a particular microorganism to produce the protein of interest. Essentially, rDNA is planted into microorganisms, and it is then going through transcription and translation. After that, the protein is purified and collected. The most known and researched microorganism suitable for that approach is *E. coli* [54].
(2) The cell-free approach requires no microorganisms. Instead, the environment for protein synthesis is created in a vial. This approach allows to fully control protein production, as there is no need to engineer living cells first. This also enables the production of toxic proteins, since there is no more need to control the surrogate cell's viability [55].

2.5.2. Protein Generation using GANs

In the paper proposed by Anand et al. [9], the authors use DCGAN architecture to generate protein contact maps. A protein contact map is a compact representation of a protein structure, consisting of distances between alpha carbons in the protein backbone. Contact maps can be used to predict tertiary protein structure [56]. Specifically, contact maps, since they are 2D matrices, can be used as an input for machine learning methods, such as CNN, which is proven to work well with image data. In order to obtain a protein contact map, several steps required. First, a matrix of pairwise Euclidean distances between residues is calculated. Then, one needs to choose some cut-off value to substitute distance values in the matrix with 1 or 0 to determine if the pair of carbons is related or unrelated. Finally, an encoded distance matrix can be used for tertiary structure reconstruction [57].

Because information about protein structures is packed into generated protein contact maps, it is important to choose efficient algorithms for „unpacking“ of protein maps into a 3D conformation. In [9] the Alternating Direction Method of Multipliers (ADMM) was used, a „divide-and-conquer” optimization method that breaks problem into smaller pieces that are easier to handle. Additionally, Rosetta’s fragment sampling was used, a heuristic algorithm for finding an appropriate 3D structure for 2D representation of the protein. The authors also stated that Rosetta might take much longer to unfold protein maps than ADMM because of Rosetta‘s heuristic algorithm nature.

Nevertheless, even though ADMM is faster, Rosetta’s unfolded local structures have fewer structural errors [9]. Results after generation and folding were also compared to other probabilistic models, such as FB5-HMM [58], which generates coordinates for alpha-carbons in 3D space, TorusDBN, mentioned earlier, and two other implementations of GAN: Torsion GAN [9], which generates torsion angles like TorusDBN and Full-atom GAN [9], which generates full-atom peptide backbones (Figure 2.9). The method described in [9] is limited w.r.t. protein maps' sizes because only 16-, 64- and 128-residue generated maps were giving the best results. Additionally, it was observed that generated maps with higher amount of residues were highly erroneous. It is also possible that the authors decided to limit maps due to Rosetta taking a large amount of time to unfold generated maps. Overall, the generator was able to learn representations of proteins in the form of protein maps. Generated and real protein contact map samples look visually similar but not identical (**Error! Reference source not found.**).

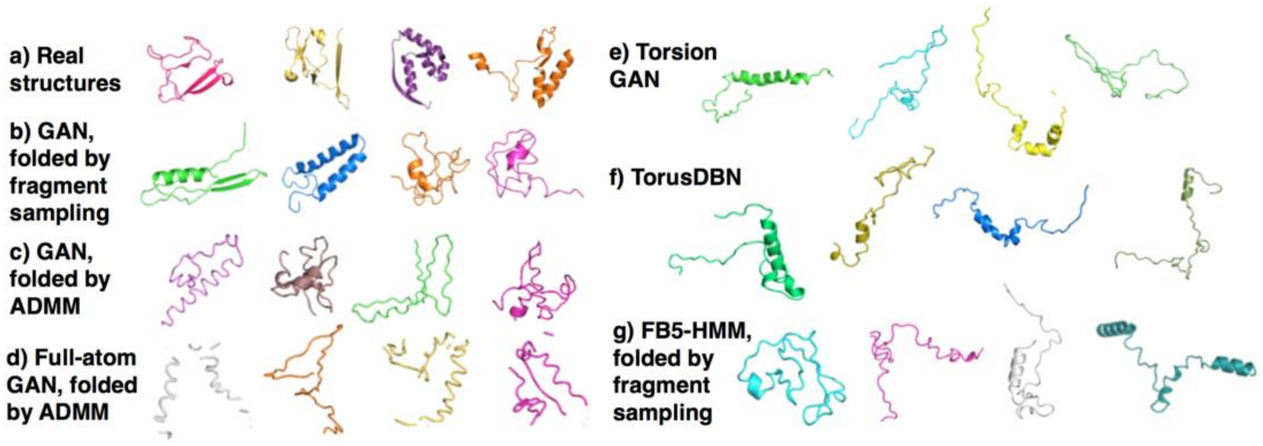
a) Real structures
b) GAN, folded by fragment sampling
c) GAN, folded by ADMM
d) Full-atom GAN, folded by ADMM
e) Torsion GAN
f) TorusDBN
g) FB5-HMM, folded by fragment sampling

Figure 2.9: Comparison of generated samples folded into 3D structures by ADMM and Rosetta. Picture is a Figure 3 from [7].

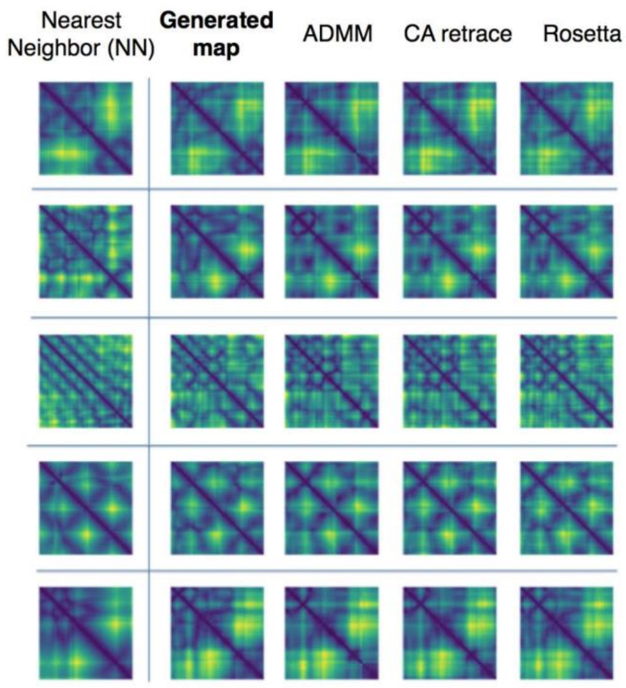

Figure 2.10: Example of protein maps generated by DCGAN, along with the contact maps from real dataset, taken by Nearest Neighbor principle for comparison. Picture is a part of the Figure 2 from [9].

3. Experimental Setup

3.1. Dataset

The dataset of choice was CIFAR-10 (Canadian Institute for Advanced Research) [59], which consists of 32x32 pixels sized images of 10 classes, each class uniformly distributed images (5000 per class in the train set, 1000 per class in the test set, 60000 images in total). This dataset is often used to evaluate classification algorithms‘ performance, and it is used in this thesis because of its relatively small size, which is beneficial for faster training. Before training, images from the dataset were normalized to the mean $\mu = 0.5$ and standard deviation $\sigma = 0.5$ for each RGB channel. Specific values for mean and standard deviation are estimations taken from [60]. Normalization is essential, as both DCGAN and WGAN architectures employ Sigmoid and Tanh activation functions that are sensitive to the input values range.

3.2. Models

DCGAN was implemented according to the guidelines from the original paper [13]. Discriminator uses strided convolutions combined with leaky rectified linear units (LeakyReLU) to downsample images and Sigmoid output activation function in the end to output probability of an image being either real or fake. The generator uses transposed convolutions combined with rectified linear units (ReLU) with hyperbolic tangent output activation function to upsample random noise to a generated sample. Both discriminator and generator use batch normalization layers as well. The optimization strategy of choice for DCGAN was Adam, with beta one and beta two being chosen manually ($b_1 = 0.5$, $b_2 = 0.999$).

WGAN has a slightly different architecture than DCGAN. There is no sigmoid activation function in the output layer of the discriminator, and the optimization strategy is also different – WGAN uses root mean square propagation (RMSprop) instead of Adam. The loss function is different, too, as the discriminator’s output is not probability, but the distance (see Chapter 2.2.3. for details on Wasserstein distance).

To justify picking weight-clipping approach instead of gradient penalty for WGAN, weight clipped WGAN has shown a performance comparable to WGAN using gradient penalty on CIFAR-10 dataset [61]. Additionally, weight clipping is easier to implement than gradient penalty.

3.3. Hyperparameters

In order to estimate difference between the two models, learning rates and batch sizes for both GAN types were selected and a hyperparameter grid search was conducted. Selected learning rates lay within the range [10^{-2}, 10^{-5}] and two distinct batch sizes were added to the search (Table 1). This specific range of learning rates and batch sizes were picked based on the choices that different authors made in their previously published papers [9,13,16].

learning rates for discriminator and generator (DCGAN)	$\{10^{-2}, 10^{-3}, 10^{-4}, 10^{-5}\}$
learning rates for critic and generator (WGAN)	$\{10^{-2}, 10^{-3}, 10^{-4}, 10^{-5}\}$
batch sizes	{64, 128}
coefficient beta 1 (Adam; DCGAN)	0.5
coefficient beta 2 (Adam; DCGAN)	0.999
weight clipping constant *c* (WGAN)	0.01

Table 1: Hyperparameter search space for DCGAN and WGAN.

Heusel et al. [16] showed that different learning rates for generator and discriminator lead to better training results with DCGAN and WGAN. Preferring higher learning rate for discriminator or generator improves results. Thus, it will be interesting to see how different learning rates will affect weight clipped WGAN and original DCGAN.

3.4. Evaluation Method

In order to evaluate DCGAN and WGAN performance, FID was calculated after every epoch and recorded. The fixed sample size of fake and real images for FID calculation was 1000 samples.

3.5. Hardware and Software

All experiments were conducted on the dedicated server provided by the Institute for Machine Learning at Johannes Kepler University in Linz, Austria. The server is running on Linux operating system and is powered by CPU Intel Xeon(R) E5-2660 v3 with 20 cores and a base frequency of 2.60 GHz, GPU NVIDIA Tesla K40 with 12GB of VRAM and 256GB of RAM.

The code for experiments was written in Python 3.7. Models of choice for experiments were implemented in PyTorch 1.2.0 [62]. Additionally, NumPy 1.2.0 [63], SciPy 1.7.0 [64], Pandas 1.3.0 [65] and Matplotlib 3.4.2 [66] frameworks were used for the calculation of FID and visualization of the results. Code is available at: https://github.com/Bollo7/GAN_project.

4. Results and Discussion

In this section, the results of training on the CIFAR-10 using different hyperparameter choices from Table 1 will be presented. Every combination of learning rates and batch size listed in Table 1 was trained with five random initializations for 25 epochs. Every run, both models were initialized with normally distributed random weights and zero biases. More specifically, batch normalization layers are initialized with weights sampled from a normal distribution with mean $\mu = 1.0$ and standard deviation $\sigma = 0.02$, and convolutional layers are initialized with mean $\mu = 0.0$ and standard deviation $\sigma = 0.02$. These specific μ and σ parameters for initial weights distribution are shown to improve training of deep neural networks [67] and are taken from [13].

Results are represented as plots of FID over 25 epochs and tables that show the best FID over the whole run and mean FID from the last epoch for each specific set of hyperparameters. Means and standard deviations in the result tables were calculated by training the models with particular hyperparameters with five different random initializations and averaging FID over these five initializations. In order to select the best performing architecture with identical hyperparameters, the Mann-Whitney-U test was conducted using the last epoch FIDs and best FIDs over the whole run (one set of hyperparameters, five random initializations). The intention to include both the last and best epochs is to compare the models' end performance and compare the best-achieved performance by simulating an early stopping of training.

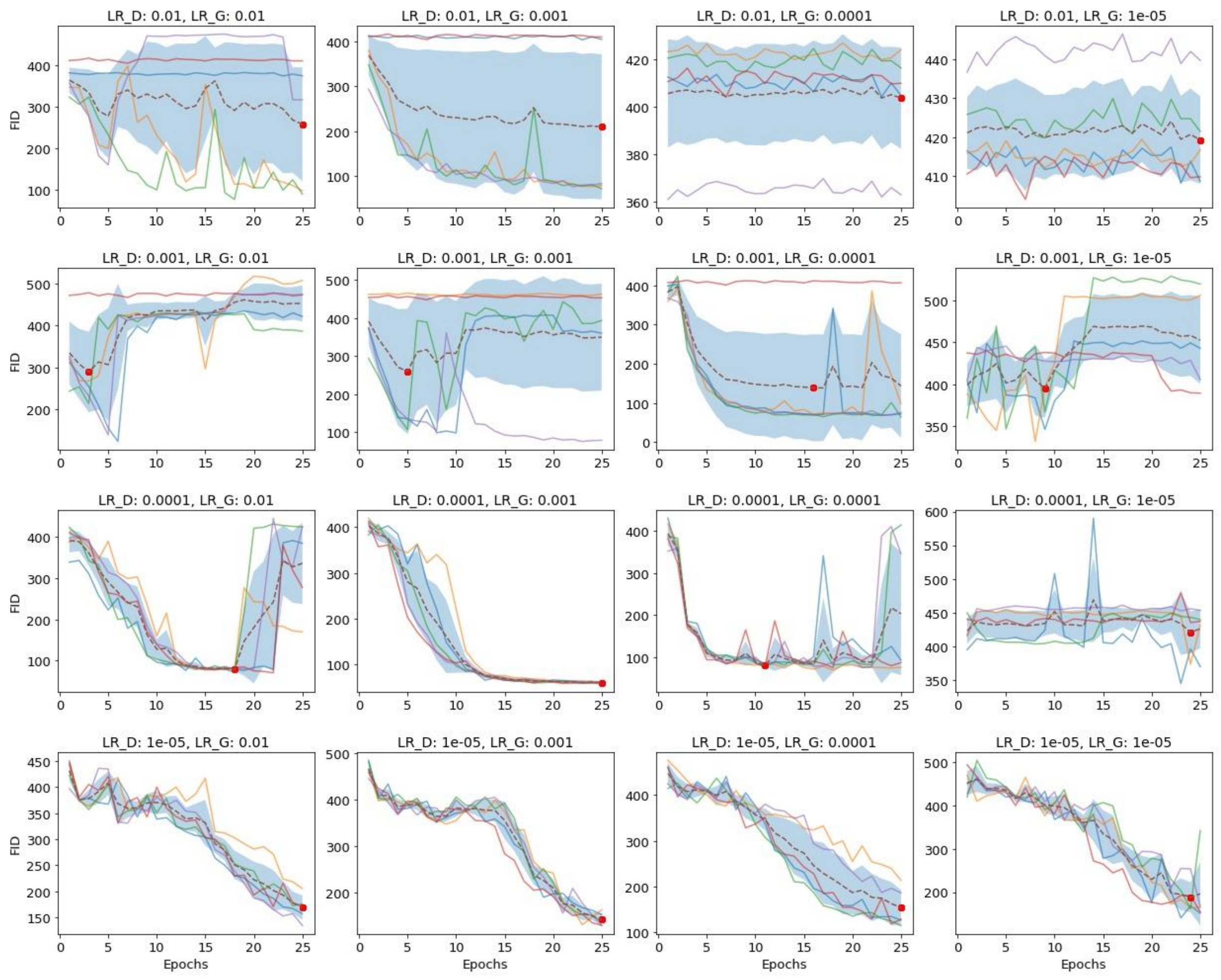

Figure 4.1: Results of training DCGAN with learning rates from Table 1 for 25 epochs with batch size 64. Solid lines represent different random initializations, the dashed line represents the mean FID across all the random initializations, the blue-colored area represents the amount of dispersion and the red point represents the best mean FID from the whole run.

In the Figure 4.1, DCGAN seems to be very unstable during training with learning rate sets {10^{-2}, x} and {10^{-3},x}, where x ∈ {10^{-2}, 10^{-3}, 10^{-4}, 10^{-5}}. I assume that the learning rate of the discriminator is too high, so that it struggles to learn. Consequently, the generator doesn't learn anything as the discriminator's feedback is meaningless. On the opposite side, learning rate sets {10^{-4}, x} and {10^{-5},x}, where x ∈ {10^{-2}, 10^{-3}, 10^{-4}, 10^{-5}}, show better results, as discriminator's learning rate is sufficiently low.

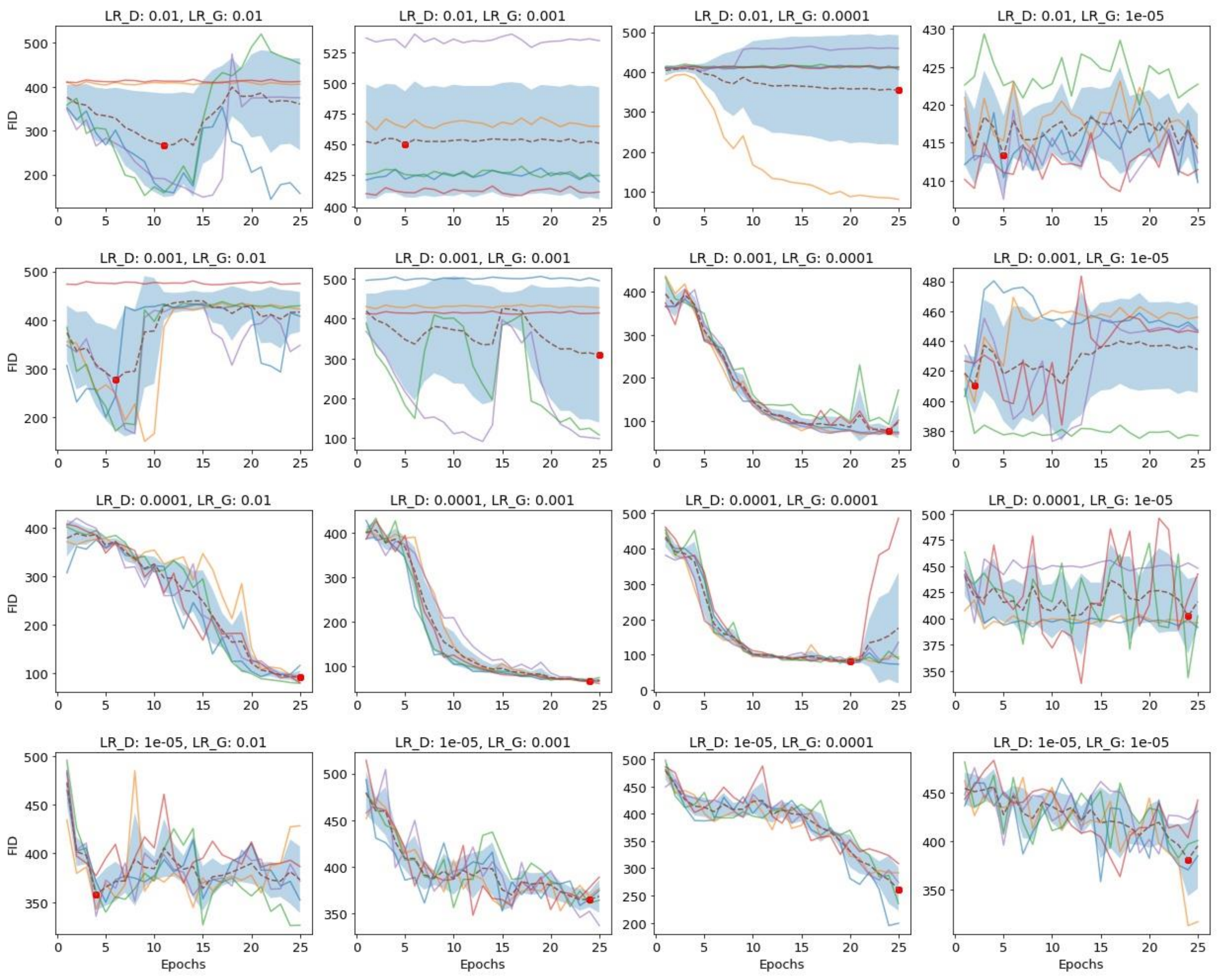

Figure 4.2: Results of training DCGAN with learning rates from Table 1 for 25 epochs with batch size 128. Solid lines represent different random initializations, the dashed line represents the mean FID across all the random initializations, the blue-colored area represents the amount of dispersion and the red point represents the best mean FID from the whole run.

Results from Figure 4.2 show that DCGAN is unstable during training with learning rate sets {10^{-2}, x} and {10^{-3},x}, where x ∈ {10^{-2}, 10^{-3}, 10^{-4}, 10^{-5}}. In general, DCGAN underperforms with batch size 128. However, Figure 4.2 also shows that training DCGAN with learning rates {10^{-3}, 10^{-4}} {10^{-4}, 10^{-2}}, {10^{-4}, 10^{-3}} and {10^{-4}} in combination with batch size 128 give similar or better results in comparison to same learning rate sets combined with batch size 64.

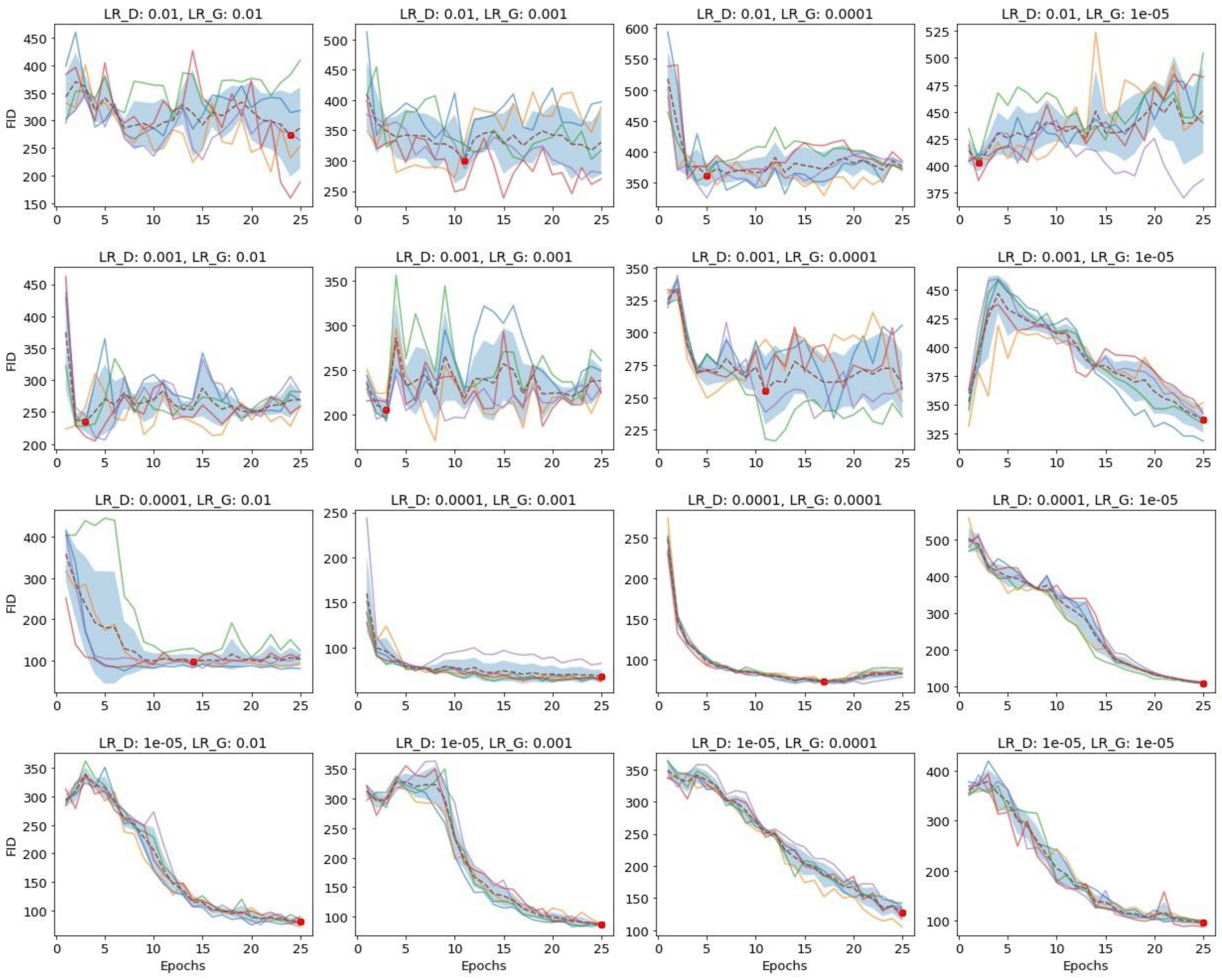

Figure 4.3: Results of training WGAN with learning rates from Table 1 for 25 epochs with batch size 64. Solid lines represent different random initializations, the dashed line represents the mean FID across all the random initializations, the blue-colored area represents the amount of dispersion and the red point represents the best mean FID from the whole run.

The first observation to consider in Figure 4.3 is that WGAN behaves more stable during training in comparison to DCGAN. WGAN with learning rate sets {10^{-2}, x} and {10^{-3},x}, where x ∈ {10^{-2}, 10^{-3}, 10^{-4}, 10^{-5}} also performs much worse than WGAN with the other given learning rate sets, which is also explainable by discriminator learning rate being too high, such that generator does not receive a proper feedback. However, if compared to DCGAN, then WGAN seems more stable, as results from different random initializations do not differ as much as they do in DCGAN’s case.

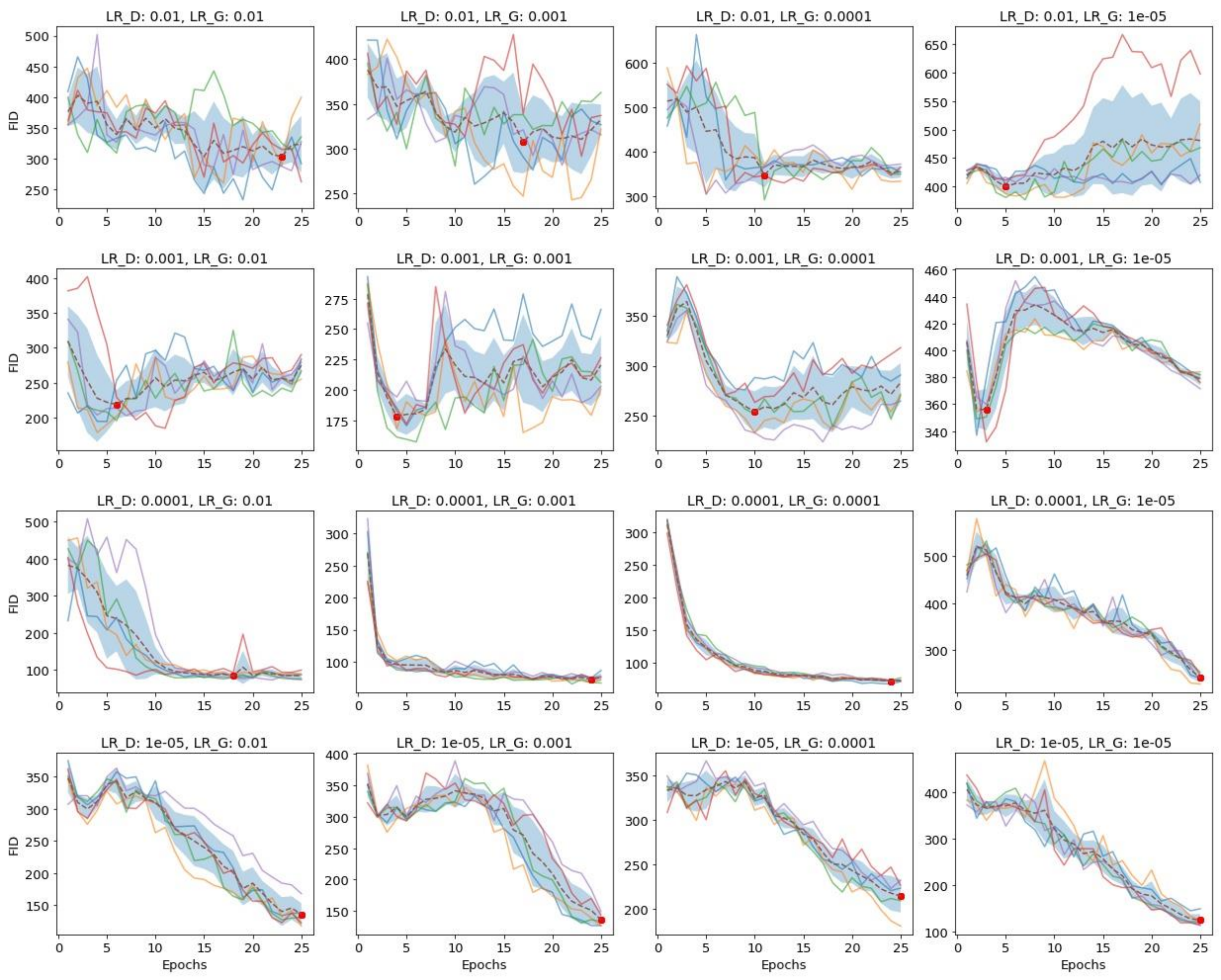

Figure 4.4: Results of training WGAN with learning rates from Table 1 for 25 epochs with batch size 128. Solid lines represent different random initializations, the dashed line represents the mean FID across all the random initializations, the blue-colored area represents the amount of dispersion and the red point represents the best mean FID from the whole run.

In Figure 4.4 WGAN behaves more stable during training in comparison to DCGAN. Batch size 128 mostly worsened stability and the end results.

It can be observed that both models with some learning rates choices behave worse with a batch size of 128, than with a batch size of 64. Interestingly, a worsening of performance can be caused by larger batch size. In the study conducted by Keskar et al. [68], authors show that a large batch size leads to worse testing results for neural networks for classification than with smaller batch sizes. The authors state that the larger batch size creates sharper minima, negatively affecting the travel of parameters to the minima. In the case of GAN architecture, two networks must reach Nash Equilibrium in order to converge. Given that both generator and discriminator are likely to settle on the flat stationary point, sharper minima may make it even more challenging for GAN to converge. Majority of learning rates combined with batch size 128 underperform in comparison to the same learning rates combined with batch size 64 for both DCGAN and WGAN.

WGAN is generally stable during training regardless of the hyperparameters chosen. WGAN is much more stable than DCGAN, judging by the dispersion values and behavior of the models during training. For DCGAN, it was the case when training collapsed, and the model was not able to learn anymore, while for WGAN, training didn't collapse in any run. It is worth noting that, in general, WGAN with a higher learning rate for the generator performs better in terms of FID than WGAN with a higher learning rate for the discriminator. The same tendency can be observed for DCGAN – a higher learning rate for the generator is preferable.

#	$\{lr_D, lr_G\}$	Last epoch DCGAN_64	Last epoch WGAN_64	p-value
1	$\{10^{-2}\}$	257.15 ± 137.32	286.6 ± 73.57	5.00e-01
2	$\{10^{-2}, 10^{-3}\}$	209.47 ± 161.43	330.37 ± 53.23	3.38e-01
3	$\{10^{-2}, 10^{-4}\}$	403.57 ± 21.39	376.75 ± 6.24	7.18e-02
4	$\{10^{-2}, 10^{-5}\}$	419.29 ± 11.16	452.0 ± 40.07	7.18e-02
5	$\{10^{-3}, 10^{-2}\}$	452.62 ± 43.06	**269.89 ± 10.02**	**6.10e-03**
6	$\{10^{-3}\}$	349.52 ± 140.9	238.58 ± 14.14	7.18e-02
7	$\{10^{-3}, 10^{-4}\}$	143.15 ± 132.45	260.29 ± 24.03	7.18e-02
8	$\{10^{-3}, 10^{-5}\}$	452.73 ± 52.19	**336.65 ± 11.0**	**6.10e-03**
9	$\{10^{-4}, 10^{-2}\}$	336.47 ± 99.32	**103.35 ± 15.12**	**6.10e-03**
10	$\{10^{-4}, 10^{-3}\}$	**60.61 ± 0.86**	68.12 ± 7.28	**6.10e-03**
11	$\{10^{-4}\}$	202.93 ± 146.06	83.64 ± 3.89	1.48e-01
12	$\{10^{-4}, 10^{-5}\}$	425.8 ± 29.18	**110.4 ± 1.79**	**6.10e-03**
13	$\{10^{-5}, 10^{-2}\}$	168.21 ± 23.37	**81.29 ± 5.92**	**6.10e-03**
14	$\{10^{-5}, 10^{-3}\}$	141.84 ± 13.23	**86.69 ± 1.08**	**6.10e-03**
15	$\{10^{-5}, 10^{-4}\}$	154.12 ± 38.66	127.03 ± 13.05	3.38e-01
16	$\{10^{-5}\}$	195.91 ± 73.49	**95.8 ± 4.55**	**6.10e-03**

Table 2: Results of Mann-Whitney-U test on FID values from last epoch of WGAN and DCGAN with identical hyperparameters. lr_G corresponds to the learning rate for network G and lr_D corresponds to the learning rate for network D. Values in bold correspond to the significant p-values. All of the learning rates listed in this table are combined with batch size 64.

In Table 2, WGAN significantly outperforms DCGAN with hyperparameter sets number 5, 8, 9, 11, 13, 14 and 15. The only hyperparameter set with which DCGAN significantly outperforms WGAN is set number 10.

#	$\{lr_D, lr_G\}$	Last epoch DCGAN_128	Last epoch WGAN_128	p-value
1	$\{10^{-2}\}$	360.07 ± 104.55	322.8 ± 46.41	1.05e-01
2	$\{10^{-2}, 10^{-3}\}$	451.01 ± 45.41	**332.87 ± 16.49**	**6.10e-03**
3	$\{10^{-2}, 10^{-4}\}$	354.2 ± 138.03	355.06 ± 13.35	7.18e-02
4	$\{10^{-2}, 10^{-5}\}$	414.21 ± 4.55	480.66 ± 68.67	1.05e-01
5	$\{10^{-3}, 10^{-2}\}$	416.48 ± 41.0	**274.91 ± 12.53**	**6.10e-03**
6	$\{10^{-3}\}$	308.53 ± 169.63	220.37 ± 24.63	3.38e-01
7	$\{10^{-3}, 10^{-4}\}$	**97.8 ± 38.68**	283.04 ± 19.63	**6.10e-03**
8	$\{10^{-3}, 10^{-5}\}$	434.46 ± 29.04	**378.52 ± 4.42**	**3.01e-02**
9	$\{10^{-4}, 10^{-2}\}$	90.92 ± 13.25	84.15 ± 9.36	2.02e-01
10	$\{10^{-4}, 10^{-3}\}$	**66.76 ± 4.31**	75.82 ± 6.43	**4.73e-02**
11	$\{10^{-4}\}$	175.09 ± 156.91	**73.76 ± 1.89**	**4.73e-02**
12	$\{10^{-4}, 10^{-5}\}$	416.06 ± 24.17	**239.83 ± 8.93**	**6.10e-03**
13	$\{10^{-5}, 10^{-2}\}$	373.12 ± 34.19	**135.82 ± 17.69**	**6.10e-03**
14	$\{10^{-5}, 10^{-3}\}$	368.34 ± 17.53	**135.93 ± 9.21**	**6.10e-03**
15	$\{10^{-5}, 10^{-4}\}$	261.77 ± 39.62	**214.28 ± 18.41**	**4.73e-02**
16	$\{10^{-5}\}$	394.88 ± 44.11	**124.71 ± 12.87**	**6.10e-03**

Table 3: Results of Mann-Whitney-U test on FID values from last epoch of WGAN and DCGAN with similar hyperparameters. lr_G corresponds to the learning rate for network G and lr_D corresponds to the learning rate for network D. Values in bold correspond to the significant p-values. All of the learning rates listed in this table are combined with batch size 128.

In Table 3, WGAN significantly outperforms DCGAN with hyperparameter sets 2, 5, 8, 11, 12, 13, 14, 15 and 16, which is a majority of combinations. The only hyperparameter sets with which DCGAN significantly outperforms WGAN are sets number 7 and 10.

#	$\{lr_D, lr_G\}$	Best epoch DCGAN_64	Best epoch WGAN_64	p-value
1	$\{10^{-2}\}$	257.15 ± 137.32	273.16 ± 75.29	5.00e-01
2	$\{10^{-2}, 10^{-3}\}$	209.47 ± 161.43	300.08 ± 28.22	3.38e-01
3	$\{10^{-2}, 10^{-4}\}$	403.57 ± 21.39	**362.27 ± 19.35**	**3.01e-02**
4	$\{10^{-2}, 10^{-5}\}$	419.29 ± 11.16	**403.3 ± 8.68**	**1.84e-02**
5	$\{10^{-3}, 10^{-2}\}$	289.24 ± 96.27	234.62 ± 15.89	1.48e-01
6	$\{10^{-3}\}$	259.35 ± 164.95	206.03 ± 11.81	3.38e-01
7	$\{10^{-3}, 10^{-4}\}$	138.24 ± 136.99	255.21 ± 26.35	7.18e-02
8	$\{10^{-3}, 10^{-5}\}$	395.14 ± 34.86	**336.65 ± 11.0**	**1.08e-02**
9	$\{10^{-4}, 10^{-2}\}$	**78.46 ± 4.17**	98.6 ± 16.32	**1.08e-02**
10	$\{10^{-4}, 10^{-3}\}$	**60.61 ± 0.86**	68.12 ± 7.28	**6.10e-03**
11	$\{10^{-4}\}$	80.25 ± 3.51	**72.02 ± 0.78**	**6.10e-03**
12	$\{10^{-4}, 10^{-5}\}$	420.37 ± 31.01	**110.4 ± 1.79**	**6.10e-03**
13	$\{10^{-5}, 10^{-2}\}$	168.21 ± 23.37	**81.29 ± 5.92**	**6.10e-03**
14	$\{10^{-5}, 10^{-3}\}$	141.84 ± 13.23	**86.69 ± 1.08**	**6.10e-03**
15	$\{10^{-5}, 10^{-4}\}$	154.12 ± 38.66	127.03 ± 13.05	3.38e-01
16	$\{10^{-5}\}$	187.79 ± 34.66	**95.8 ± 4.55**	**6.10e-03**

Table 4: Results of Mann-Whitney-U test on best FID values from the whole run of WGAN and DCGAN with identical hyperparameters. lr_G corresponds to the learning rate for network G and lr_D corresponds to the learning rate for network D. Values in bold correspond to the significant p-values. All of the learning rates listed in this table are combined with batch size 64.

In Table 4, WGAN significantly outperforms DCGAN with hyperparameter sets number 3, 4, 8, 11, 12, 13, 14 and 16, which is the half of hyperparameter sets. The only hyperparameter sets with which DCGAN significantly outperforms WGAN are sets number 9 and 10.

#	$\{lr_D, lr_G\}$	Best epoch DCGAN_128	Best epoch WGAN_128	p-value
1	$\{10^{-2}\}$	266.3 ± 117.98	302.64 ± 14.76	3.38e-01
2	$\{10^{-2}, 10^{-3}\}$	450.12 ± 43.01	**307.37 ± 34.77**	**6.10e-03**
3	$\{10^{-2}, 10^{-4}\}$	354.2 ± 138.03	347.81 ± 28.91	7.18e-02
4	$\{10^{-2}, 10^{-5}\}$	413.36 ± 5.2	400.07 ± 13.61	1.48e-01
5	$\{10^{-3}, 10^{-2}\}$	277.02 ± 103.04	218.41 ± 20.78	2.02e-01
6	$\{10^{-3}\}$	308.53 ± 169.63	178.12 ± 11.9	3.38e-01
7	$\{10^{-3}, 10^{-4}\}$	**77.26 ± 7.91**	253.48 ± 20.92	**6.10e-03**
8	$\{10^{-3}, 10^{-5}\}$	410.5 ± 18.97	**355.78 ± 15.4**	**1.08e-02**
9	$\{10^{-4}, 10^{-2}\}$	90.92 ± 13.25	83.0 ± 5.12	2.65e-01
10	$\{10^{-4}, 10^{-3}\}$	**66.42 ± 1.6**	71.87 ± 3.58	**4.73e-02**
11	$\{10^{-4}\}$	81.71 ± 6.43	**72.4 ± 2.19**	**1.84e-02**
12	$\{10^{-4}, 10^{-5}\}$	402.14 ± 35.64	**239.83 ± 8.93**	**6.10e-03**
13	$\{10^{-5}, 10^{-2}\}$	357.64 ± 15.9	**135.82 ± 17.69**	**6.10e-03**
14	$\{10^{-5}, 10^{-3}\}$	364.95 ± 8.36	**135.93 ± 9.21**	**6.10e-03**
15	$\{10^{-5}, 10^{-4}\}$	261.77 ± 39.62	**214.28 ± 18.41**	**4.73e-02**
16	$\{10^{-5}\}$	380.94 ± 37.85	**124.71 ± 12.87**	**6.10e-03**

Table 5: Results of Mann-Whitney-U test on best FID values from the whole run of WGAN and DCGAN with identical hyperparameters. lr_G corresponds to the learning rate for network G and lr_D corresponds to the learning rate for network D. Values in bold correspond to the significant p-values. All of the learning rates listed in this table are combined with batch size 128.

In Table 5, WGAN significantly outperforms DCGAN with hyperparameter sets number 2, 8, 11, 12, 13, 14, 15 and 16, which is half of hyperparameter sets. The only hyperparameter sets with which DCGAN significantly outperforms WGAN are sets number 7 and 10.

The results of Mann-Whitney-U tests show that WGAN outperforms DCGAN with most of the given hyperparameter sets. Both last epoch and best epoch FIDs are lower for WGAN in most cases. It can be noted that mean FIDs of WGAN often have lower standard deviation values than DCGAN. The only hyperparameter set with which DCGAN constantly outperformed WGAN was hyperparameter set number 10, which seems like a ‘sweet-spot’ for DCGAN on CIFAR-10.

To conclude, both models show good results, but the difference between them is that if hyperparameters for DCGAN are carefully adjusted it will perform as good or even better than WGAN. However, DCGAN was failing to train in many cases – FID was not dropping down, but was rather increasing or stayed the same during training. Meanwhile, WGAN shows general stability during training over wide range of learning rates. It also has better dynamics during training, meaning that FID does not stays the same, but constantly changes – model tries its best to learn.

5. Conclusion

The practical part of this thesis provides a comparison of DCGAN and WGAN architectures using the Frechet Inception Distance evaluation method. DCGAN, in most cases, had heavy perturbations of FID during the training. It was also the case, that DCGAN could not converge and it can be seen by FID being “flat-lined” on the graph. WGAN had much less variance of FID between random initializations and was constantly training without collapsing. Even though selected method to enforce 1-Lipschitz constraint on gradients of WGAN is not superior, it was enough to show that Wasserstein loss is more stable than Jensen-Shannon divergence during training with almost any hyperparameters from Table 1. It can be concluded that given a limited budget of time, it is better to choose WGAN for practical needs.

As a suggestion based on the work of Anand et al. [9], it may be possible to improve results of the generation of protein contact maps. Authors of the paper state that they used the same learning rate for both the generator and the discriminator. Perhaps, results of experiments could be improved by using different hyperparameters, such as higher learning rate for generator, than for discriminator. It might also be beneficial to try GAN architecture other than DCGAN. As the experiment in this thesis shows, DCGAN is often unstable and in some cases fails to learn.

As for future work, experiments could be improved by increasing the amount of iterations that models would be trained for. In this thesis, models were trained for 25 epochs only but over a relatively wide range of hyperparameters. The main focus was on finding out whether Wasserstein loss is better than Jensen-Shannon divergence. The next step is to conduct the same hyperparameter search using the protein contact maps dataset. In my opinion, performance measured using CIFAR-10 might be different from the protein contact maps dataset due to different dataset domains.

6. List of Figures

7. List of Tables

8. Bibliography

[1] Vieira, Armando. “Business Applications of Deep Learning.”. *Deep Natural Language Processing and AI Applications for Industry 5.0*. IGI Global (2021): pp. 39–67.

[2] The Official NVIDIA Blog. “How AI, Machine Learning Advance Academic Research | NVIDIA Blog.”, 2019. URL https://blogs.nvidia.com/blog/2019/03/27/how-ai-machine-learning-are-advancing-academic-research/.

[3] Goodfellow, Ian J., Pouget-Abadie, Jean, Mirza, Mehdi, Xu, Bing, Warde-Farley, David, Ozair, Sherjil, Courville, Aaron, and Bengio, Yoshua. “Generative Adversarial Networks.” (10 June 2014). URL https://arxiv.org/pdf/1406.2661.

[4] Jaydeep T. Chauhan. “Comparative Study of GAN and VAE.” *International Journal of Computer Applications* Vol. 182 No. 22 (2018): pp. 1–5.

[5] Kingma, Diederik P. and Welling, Max. “Auto-Encoding Variational Bayes.” (20 December 2013). URL https://arxiv.org/pdf/1312.6114.

[6] Uzunova, Hristina, Ehrhardt, Jan, Jacob, Fabian, Frydrychowicz, Alex, and Handels, Heinz. “Multi-scale GANs for Memory-efficient Generation of High Resolution Medical Images.” (2 July 2019). URL https://arxiv.org/pdf/1907.01376.

[7] Huang, Fei, Guan, Jian, Ke, Pei, Guo, Qihan, Zhu, Xiaoyan, and Huang, Minlie. “A Text GAN for Language Generation with Non-Autoregressive Generator.” (2020).

[8] Engel, Jesse, Agrawal, Kumar Krishna, Chen, Shuo, Gulrajani, Ishaan, Donahue, Chris, and Roberts, Adam. “GANSynth: Adversarial Neural Audio Synthesis.” *International Conference on Learning Representations* (2018).

[9] Namrata Anand and Possu Huang. “Generative modeling for protein structures.” *Advances in Neural Information Processing Systems 30 (NIPS* (2018).

[10] Hochreiter, Sepp. “The Vanishing Gradient Problem During Learning Recurrent Neural Nets and Problem Solutions.” *International Journal of Uncertainty Fuzziness and Knowledge-Based Systems* Vol. 06 No. 02 (1998): pp. 107–116. DOI 10.1142/S0218488598000094.

[11] Saxena, Divya and Cao, Jiannong. “Generative Adversarial Networks (GANs): Challenges, Solutions, and Future Directions.” (30 April 2020). URL https://arxiv.org/pdf/2005.00065.

[12] Borji, Ali. “Pros and Cons of GAN Evaluation Measures.” (9 February 2018). URL https://arxiv.org/pdf/1802.03446.

[13] Radford, Alec, Metz, Luke, and Chintala, Soumith. “Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks.” (19 November 2015). URL https://arxiv.org/pdf/1511.06434.

[14] Arjovsky, Martin, Chintala, Soumith, and Bottou, Léon. “Wasserstein GAN.” (26 January 2017). URL https://arxiv.org/pdf/1701.07875.

[15] Salimans, Tim, Goodfellow, Ian, Zaremba, Wojciech, Cheung, Vicki, Radford, Alec, and Chen, Xi. “Improved Techniques for Training GANs.” (11 June 2016). URL https://arxiv.org/pdf/1606.03498.

[16] Heusel, Martin, Ramsauer, Hubert, Unterthiner, Thomas, Nessler, Bernhard, and Hochreiter, Sepp. “GANs Trained by a Two Time-Scale Update Rule Converge to a Local Nash Equilibrium.” *Advances in Neural Information Processing Systems 30 (NIPS* (2017).

[17] Payne, Joshua, Activation Functions in Artificial Neural Networks - The Startup - Medium *The Startup* (4 January 2020). URL https://medium.com/swlh/activation-functions-in-artificial-neural-networks-8aa6a5ddf832.

[18] Nielsen, Michael A. “Neural Networks and Deep Learning.”, 2015. URL http://neuralnetworksanddeeplearning.com/chap2.html.

[19] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. “Chapter 5.9 Stochastic Gradient Descent.”. *Deep Learning*. MIT Press (2016): pp. 147–149.

[20] Léon Bottou. “On-line Learning and Stochastic Approximations.”: pp. 9–42. DOI 10.1017/CBO9780511569920.003.

[21] Fukushima, K. “Neocognitron: a self organizing neural network model for a mechanism of pattern recognition unaffected by shift in position.” *Biological Cybernetics* Vol. 36 No. 4 (1980): pp. 193–202. DOI 10.1007/BF00344251.

[22] Patel, Krut, Convolutional Neural Networks — A Beginner’s Guide - Towards Data Science *Towards Data Science* (8 September 2019). URL https://towardsdatascience.com/convolution-neural-networks-a-beginners-guide-implementing-a-mnist-hand-written-digit-8aa60330d022.

[23] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. “Chapters 7.9 Parameter Tying and Parameter Sharing and 7.9.1 Convolutional Neural Networks.”. *Deep Learning*. MIT Press (2016): pp. 246–247.

[24] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. “Chapter 9.3 Pooling.”. *Deep Learning*. MIT Press (2016): pp. 330–334.

[25] “Fantastic GANs and where to find them.”, 2020. URL https://guimperarnau.com/blog/2017/03/Fantastic-GANs-and-where-to-find-them.

[26] Andrew L. Maas. “Rectifier Nonlinearities Improve Neural Network Acoustic Models.” *UR - https://www.semanticscholar.org/paper/Rectifier-Nonlinearities-Improve-Neural-Network-Maas/367f2c63a6f6a10b3b64b8729d601e69337ee3cc* (2013).

[27] Vinod Nair and Geoffrey E. Hinton. “Rectified Linear Units Improve Restricted Boltzmann Machines Vinod Nair.”: pp. 807–814.

[28] Ioffe, Sergey and Szegedy, Christian. “Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift.” (11 February 2015). URL https://arxiv.org/pdf/1502.03167.

[29] Kingma, Diederik P. and Ba, Jimmy. “Adam: A Method for Stochastic Optimization.” (22 December 2014). URL https://arxiv.org/pdf/1412.6980.

[30] Frid-Adar, Maayan, Diamant, Idit, Klang, Eyal, Amitai, Michal, Goldberger, Jacob, and Greenspan, Hayit. “GAN-based Synthetic Medical Image Augmentation for increased CNN Performance in Liver Lesion Classification.” 5 (3 March 2018). URL https://arxiv.org/pdf/1803.01229.

[31] Shorten, Connor and Khoshgoftaar, Taghi M. “A survey on Image Data Augmentation for Deep Learning.” *Journal of Big Data* Vol. 6 No. 1 (2019). DOI 10.1186/s40537-019-0197-0.

[32] Rubner, Yossi. “The Earth Mover's Distance as a Metric for Image Retrieval.” *International Journal of Computer Vision* Vol. 40 No. 2 (2000): pp. 99–121. DOI 10.1023/A:1026543900054.

[33] “Read-through: Wasserstein GAN.”, 2021. URL https://www.alexirpan.com/2017/02/22/wasserstein-gan.html.

[34] Weng, Lilian. “From GAN to WGAN.” (18 April 2019). URL https://arxiv.org/pdf/1904.08994.

[35] Gulrajani, Ishaan, Ahmed, Faruk, Arjovsky, Martin, Dumoulin, Vincent, and Courville, Aaron. “Improved Training of Wasserstein GANs.” (31 March 2017). URL https://arxiv.org/pdf/1704.00028.

[36] Szegedy, Christian, Liu, Wei, Jia, Yangqing, Sermanet, Pierre, Reed, Scott, Anguelov, Dragomir, Erhan, Dumitru, Vanhoucke, Vincent, and Rabinovich, Andrew. “Going Deeper with Convolutions.” (17 September 2014). URL https://arxiv.org/pdf/1409.4842.

[37] Szegedy, Christian, Vanhoucke, Vincent, Ioffe, Sergey, Shlens, Jonathon, and Wojna, Zbigniew. “Rethinking the Inception Architecture for Computer Vision.” (2 December 2015). URL https://arxiv.org/pdf/1512.00567.

[38] Russakovsky, Olga, Deng, Jia, Su, Hao, Krause, Jonathan, Satheesh, Sanjeev, Ma, Sean, Huang, Zhiheng, Karpathy, Andrej, Khosla, Aditya, Bernstein, Michael, Berg, Alexander C., and Fei-Fei, Li. “ImageNet Large Scale Visual Recognition Challenge.” (2 September 2014). URL https://arxiv.org/pdf/1409.0575.

[39] Barratt, Shane and Sharma, Rishi. “A Note on the Inception Score.” (6 January 2018). URL https://arxiv.org/pdf/1801.01973.

[40] Thomas Baldwin, Madeleine Lapointe. “The Chemistry of Amino Acids. The Biology Project.”, 2003. URL http://www.biology.arizona.edu/biochemistry/problem_sets/aa/aa.html.

[41] Khan Academy. “Protein structure: Primary, secondary, tertiary & quatrenary (article) | Khan Academy.”, 2021. URL https://www.khanacademy.org/science/biology/macromolecules/proteins-and-amino-acids/a/orders-of-protein-structure.

[42] Hieronymus, Haley and Silver, Pamela A. “A systems view of mRNP biology.” *Genes & Development* Vol. 18 No. 23 (2004): pp. 2845–2860. DOI 10.1101/gad.1256904.

[43] Mason, Kenneth A., Losos, Jonathan B., Duncan, Todd, Raven, Peter Hamilton, and Johnson, George Brooks. *Biology*. McGraw-Hill Education, New York (Copyright 2020).

[44] Huang, Gao, Yuan, Yang, Xu, Qiantong, Guo, Chuan, Sun, Yu, Wu, Felix, and Weinberger, Kilian. “An empirical study on evaluation metrics of generative adversarial networks.” (2018).

[45] Rohl, Carol A., Strauss, Charlie E.M., Misura, Kira M.S., and Baker, David. “Protein Structure Prediction Using Rosetta.”. *Numerical computer methods*. Elsevier (2000): pp. 66–93.

[46] Hochreiter, S. and Schmidhuber, J. “Long short-term memory.” *Neural computation* Vol. 9 No. 8 (1997): pp. 1735–1780. DOI 10.1162/neco.1997.9.8.1735.

[47] Gao, Wenhao, Mahajan, Sai Pooja, Sulam, Jeremias, and Gray, Jeffrey J. “Deep Learning in Protein Structural Modeling and Design.” *Patterns* Vol. 1 No. 9 (2020): p. 100142. DOI 10.1016/j.patter.2020.100142.

[48] Das, Rhiju and Baker, David. “Macromolecular modeling with rosetta.” *Annual review of biochemistry* Vol. 77 (2008): pp. 363–382. DOI 10.1146/annurev.biochem.77.062906.171838.

[49] Alford, Rebecca F., Leaver-Fay, Andrew, Jeliazkov, Jeliazko R., O'Meara, Matthew J., DiMaio, Frank P., Park, Hahnbeom, Shapovalov, Maxim V., Renfrew, P. Douglas, Mulligan, Vikram K., Kappel, Kalli, Labonte, Jason W., Pacella, Michael S., Bonneau, Richard, Bradley, Philip, Dunbrack, Roland L., Das, Rhiju, Baker, David, Kuhlman, Brian, Kortemme, Tanja, and Gray, Jeffrey J. “The Rosetta All-Atom Energy Function for Macromolecular Modeling and Design.” *Journal of chemical theory and computation* Vol. 13 No. 6 (2017): pp. 3031–3048. DOI 10.1021/acs.jctc.7b00125.

[50] Anfinsen, C. B. “Principles that govern the folding of protein chains.” *Science* Vol. 181 No. 4096 (1973): pp. 223–230. DOI 10.1126/science.181.4096.223.

[51] “G.”. *Dictionary of Energy (Second Edition)*. Elsevier, Boston (2015): pp. 247–273.

[52] Khan Academy. “Globular proteins structure and function (article) | Khan Academy.”, 2021. URL https://www.khanacademy.org/test-prep/mcat/biomolecules/amino-acids-and-proteins1/a/the-structure-and-function-of-globular-proteins.

[53] Hughes, Randall A. and Ellington, Andrew D. “Synthetic DNA Synthesis and Assembly: Putting the Synthetic in Synthetic Biology.” *Cold Spring Harbor Perspectives in Biology* Vol. 9 No. 1 (2017). DOI 10.1101/cshperspect.a023812.

[54] Rosano, Germán L. and Ceccarelli, Eduardo A. “Recombinant protein expression in Escherichia coli: advances and challenges.” *Frontiers in Microbiology* Vol. 5 (2014): p. 172. DOI 10.3389/fmicb.2014.00172.

[55] Jackson, Alison M., Boutell, Joe, Cooley, Neil, and He, Mingyue. “Cell-free protein synthesis for proteomics.” *Briefings in functional genomics & proteomics* Vol. 2 No. 4 (2004): pp. 308–319. DOI 10.1093/bfgp/2.4.308.

[56] Vassura, Marco, Margara, Luciano, Di Lena, Pietro, Medri, Filippo, Fariselli, Piero, and Casadio, Rita. “Reconstruction of 3D structures from protein contact maps.” *IEEE/ACM transactions on computational biology and bioinformatics* Vol. 5 No. 3 (2008): pp. 357–367. DOI 10.1109/TCBB.2008.27.

[57] Emerson, Isaac Arnold and Amala, Arumugam. “Protein contact maps: A binary depiction of protein 3D structures.” *Physica A: Statistical Mechanics and its Applications* Vol. 465 (2017): pp. 782–791. DOI 10.1016/j.physa.2016.08.033.

[58] Li, Shuai Cheng, Bu, Dongbo, Xu, Jinbo, and Li, Ming. “Fragment-HMM: a new approach to protein structure prediction.” *Protein science a publication of the Protein Society* Vol. 17 No. 11 (2008): pp. 1925–1934. DOI 10.1110/ps.036442.108.

[59] “CIFAR-10 and CIFAR-100 datasets.”, 2017. URL https://www.cs.toronto.edu/~kriz/cifar.html.

[60] “DCGAN Tutorial — PyTorch Tutorials 1.9.0+cu102 documentation.”, 2021. URL https://pytorch.org/tutorials/beginner/dcgan_faces_tutorial.html#where-to-go-next.

[61] Lucic, Mario, Kurach, Karol, Michalski, Marcin, Gelly, Sylvain, and Bousquet, Olivier. “Are GANs Created Equal? A Large-Scale Study.” (28 November 2017). URL https://arxiv.org/pdf/1711.10337.

[62] Paszke, Adam, Gross, Sam, Massa, Francisco, Lerer, Adam, Bradbury, James, Chanan, Gregory, Killeen, Trevor, Lin, Zeming, Gimelshein, Natalia, Antiga, Luca, Desmaison, Alban, Köpf, Andreas, Yang, Edward, DeVito, Zach, Raison, Martin, Tejani, Alykhan,

Chilamkurthy, Sasank, Steiner, Benoit, Fang, Lu, Bai, Junjie, and Chintala, Soumith. “PyTorch: An Imperative Style, High-Performance Deep Learning Library.” (3 December 2019). URL https://arxiv.org/pdf/1912.01703.

[63] Harris, Charles R., Millman, K. Jarrod, van der Walt, Stéfan J., Gommers, Ralf, Virtanen, Pauli, Cournapeau, David, Wieser, Eric, Taylor, Julian, Berg, Sebastian, Smith, Nathaniel J., Kern, Robert, Picus, Matti, Hoyer, Stephan, van Kerkwijk, Marten H., Brett, Matthew, Haldane, Allan, Del Río, Jaime Fernández, Wiebe, Mark, Peterson, Pearu, Gérard-Marchant, Pierre, Sheppard, Kevin, Reddy, Tyler, Weckesser, Warren, Abbasi, Hameer, Gohlke, Christoph, and Oliphant, Travis E. “Array programming with NumPy.” *Nature* Vol. 585 No. 7825 (2020): pp. 357–362. DOI 10.1038/s41586-020-2649-2.

[64] Virtanen, Pauli, Gommers, Ralf, Oliphant, Travis E., Haberland, Matt, Reddy, Tyler, Cournapeau, David, Burovski, Evgeni, Peterson, Pearu, Weckesser, Warren, Bright, Jonathan, van der Walt, Stéfan J., Brett, Matthew, Wilson, Joshua, Millman, K. Jarrod, Mayorov, Nikolay, Nelson, Andrew R. J., Jones, Eric, Kern, Robert, Larson, Eric, Carey, C. J., Polat, İlhan, Feng, Yu, Moore, Eric W., VanderPlas, Jake, Laxalde, Denis, Perktold, Josef, Cimrman, Robert, Henriksen, Ian, Quintero, E. A., Harris, Charles R., Archibald, Anne M., Ribeiro, Antônio H., Pedregosa, Fabian, and van Mulbregt, Paul. “SciPy 1.0: fundamental algorithms for scientific computing in Python.” *Nature Methods* Vol. 17 No. 3 (2020): pp. 261–272. DOI 10.1038/s41592-019-0686-2.

[65] “pandas - Python Data Analysis Library.”, 2021. URL https://pandas.pydata.org/about/citing.html.

[66] Hunter, John D. “Matplotlib: A 2D Graphics Environment.” *Computing in Science & Engineering* Vol. 9 No. 3 (2007): pp. 90–95. DOI 10.1109/MCSE.2007.55.

[67] Boulila, Wadii, Driss, Maha, Al-Sarem, Mohamed, Saeed, Faisal, and Krichen, Moez. “Weight Initialization Techniques for Deep Learning Algorithms in Remote Sensing: Recent Trends and Future Perspectives.” (13 February 2021). URL https://arxiv.org/pdf/2102.07004.

[68] Keskar, Nitish Shirish, Mudigere, Dheevatsa, Nocedal, Jorge, Smelyanskiy, Mikhail, and Tang, Ping Tak Peter. “On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima.” (15 September 2016). URL https://arxiv.org/pdf/1609.04836.