LiU-ITN-TEK-A--21/020-SE

Image Synthesis Using CycleGAN to Augment Imbalanced Data for Multi-class Weather Classification

Marcus Gladh

Daniel Sahlin

2021-06-02

Department of Science and Technology
Linköping University
SE-601 74 Norrköping, Sweden

Institutionen för teknik och naturvetenskap
Linköpings universitet
601 74 Norrköping

LiU-ITN-TEK-A--21/020-SE

Image Synthesis Using CycleGAN to Augment Imbalanced Data for Multi-class Weather Classification

The thesis work carried out in Medieteknik
at Tekniska högskolan at
Linköpings universitet

Marcus Gladh
Daniel Sahlin

Norrköping 2021-06-02

Department of Science and Technology
Linköping University
SE-601 74 Norrköping, Sweden

Institutionen för teknik och naturvetenskap
Linköpings universitet
601 74 Norrköping

Abstract

In the last decade, convolutional neural networks have been used to a large extent for image classification and recognition tasks in a number of fields. For image weather classification, data can be both sparse and unevenly distributed amongst labels in the training set. As a way to improve the performance of the classifier, one often use traditional augmentation techniques to increase the size of the training set and help the classifier to converge towards a desirable solution. This can often be met with varying results, which is why this work intends to investigate another approach of augmentation using image synthesis. The idea is to make use of the fact that most datasets contain at least one label that is well represented. In weather image datasets, this is often the sunny label. CycleGAN is a framework which is capable of image-to-image translation (i.e. synthesizing images to represent a new label) using unpaired data. This makes the framework attractive as it does not put any unnecessary requirements on the data collection.

To test the whether the synthesized images can be used as an augmentation approach, training samples in one label was deliberately reduced sequentially and supplemented with CycleGAN synthesized images. The results show adding synthesized images using CycleGAN can be used as an augmentation approach, since the performance of the classifier was relatively unchanged even though the number of images was low. In this case it was as few as 198 training samples in the label that represented foggy weather. Comparing CycleGAN to traditional augmentation techniques, it proved to be more stable as the number of images in the training set decreased. A modification to CycleGAN, which used weight demodulation instead of instance normalization in its generators, removed artifacts that otherwise could appear during training. This improved the visual quality of the synthesized images overall.

Acknowledgments

We would like to thank the Computer Graphics and Image Processing group at Linköping University, for giving us the opportunity to do this thesis work. A special thanks goes out to our supervisor Gabriel Eilertsen for his continuous support and advice throughout. Another special thanks goes out to our examiner Jonas Unger for providing additional support and advice and their combined help with providing the necessary hardware to do our work.

Contents

List of Figures

List of Tables

1 Introduction

Since the launch of the ImageNet Large Scale Visual Recognition Challenge (ILSVRC), a lot of research has been put into developing deep neural networks regarding object recognition in still images and image classification. To achieve image classification with smaller error rates, most methods lean towards making the networks deeper [1]. Such approach requires larger training sets to utilize the networks full potential. Weather classification, which is relatively new topic within computer vision and image classification [2, 3], is a specific case where labelled (categorized) data can often be sparse and unevenly distributed (known as **imbalanced data**). This is mostly due to weather conditions such as rain and snow are rare in comparison to sunny and hazy days. Unevenly distributed data could have consequences on a network, as the representation learning degrades its overall performance. To compensate for inadequate and imbalanced labels in the general sense, while also improving the accuracy of an image classifier, various forms of data augmentation can be used [4, 5].

1.1 Background

Data augmentation is the process of creating fake data, to alter the training process of a neural network. This has proven to enhance the results of the training process, as it creates larger training sets and helps to control the regularization of the network, leading to a better generalization. However, this is under the assumption that the data is evenly distributed. Common strategies for data augmentation of images are rotation, translation, cropping and noise injection. A proposal of augmenting image data, to shift the distribution of labels, is to synthesize images from well represented labels to underrepresented labels.

Generative adversarial network (GAN) is a deep learning method that builds up multiple layers of abstractions, to synthesize images which can be mistaken as real. CycleGAN is an addition to the family of GANs, which introduces a cycle consistency [6]. CycleGAN has been studied in greater detail, but the question whether this model can be used to balance imbalanced training data for weather classification remains somewhat unexplored.

1.2 Aim

The aim is to explore the prospect of augmenting existing image data for multi-weather classification, in terms of creating uniform distributed labels. The focus will be on synthesising images using CycleGAN to generate images containing different weather conditions. To evaluate the results, comparison will be done between an augmented dataset and an existing dataset using a convolutional neural network (CNN) as a multi-class weather classifier.

1.3 Research questions

1. How can CycleGAN be used to enhance the performance of an existing CNN-classifier using an imbalanced training set of weather images for multi-class weather classification?
2. Is it possible to augment images to represent other weather phenomenons from sunny images, such as fog, rain and snow? Are there any significant differences between CycleGANs capability of generating the different weather phenomenons?
3. How does the quality of CycleGAN generated images change when reducing the number of training samples in the target domain?
4. For weather classification: How well does traditional augmentation techniques perform compared to CycleGAN when balancing an imbalanced dataset?

1.4 Delimitations

An existing CNN-model will be used as a multi-weather classifier and its implementation will be done according to previous studies [4]. The size of the images used to train the classifier and the CycleGAN will be limited in size, due to the computational power and time demand to train a deep neural network. As the focus is to see whether CycleGAN can be used to create a more uniform distribution of labels in a dataset, existing datasets related to weather classification will be used. Furthermore, the target labels in the dataset will be limited to the weather conditions fog, rain and snow, as these are typically underrepresented labels in most datasets found online.

2 Theory

In this chapter, theory relevant to convolutional neural networks (CNN) and generative adversarial networks (GANs) will be presented. This chapter will also present how their properties ties into image classification and image synthesis respectively.

2.1 Deep learning

Deep learning belongs to a group of methods based on **artificial neural networks** (ANNs), which tries to find representations needed for feature detection or classification from raw data. ANNs are referred to as neural networks, because their structure is inspired by *biological neural networks* e.g. the human brain [7, 8]. The human brain is composed of *biological neurons* that connects to one another and forms a massive framework, to work in parallel with each other. A biological neuron receives electrochemical stimulations through its *dendrites*, which are connected to other neurons and passes the stimulation through the *axon*, which is a neurons output [9]. A simplified version of this process is depicted in Figure 2.1. A neurons' axon can often be approximated as a threshold function, which determines whether that neuron should be activated. This is done by measuring the net stimulus through the dendrites and determine if it is above a certain threshold.

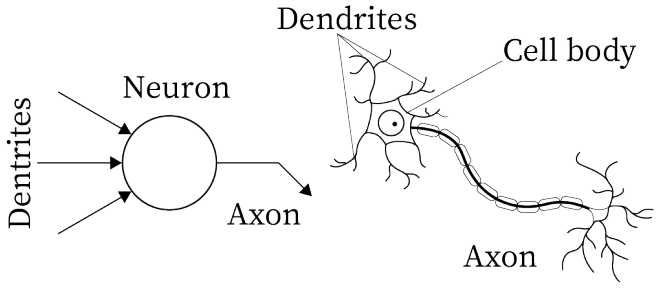


Figure 2.1: A simple depiction of a biological neuron which receives its electrochemical stimulations through its dendrites and passes the stimulation along through its axon.

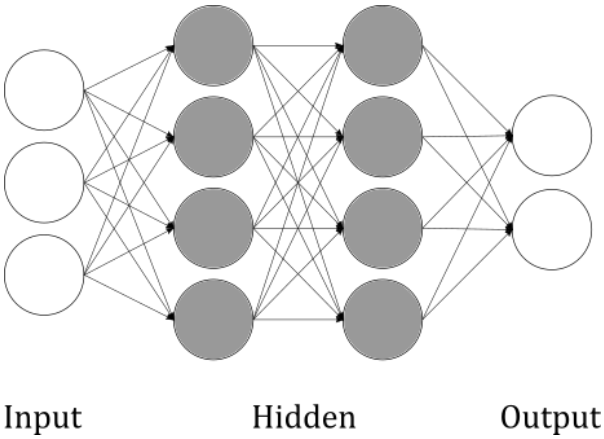


Figure 2.2: An overview of a traditional feedforward neural network. The circles represent the **artificial neurons** e.g. the computational units of the network. The neurons are connected to each other in **dense** layers, meaning that a neuron is connected to all neurons in both the previous and next layer.

Similar to its biological counterpart, the building blocks of an artificial neural network are small computational units known as **artificial neurons** (hereafter referred to as neurons) which receives and produces signals to compute the output of a network. The most common ANN models are **feedforward neural networks**, also known as **feedforward networks** or **multilayer perceptron** (MLP), where information flows through the network in one direction [8]. This process of passing the information forward is known as **forward propagation**. It is worth noting that there are other variations such as *recurrent neural networks* that include feedback connections, but these will not be discussed in the report. Neurons in a feedforward network are often connected to each other in a layer structure (represented as vectors of values) as seen in Figure 2.2. The figure exemplifies a network with **dense** (or **fully-connected**) layers, which means that each neuron receives inputs from all components in the previous layers and proceeds to pass an output to all components of the next layer. For reference, the first layer which receives the input is named **input layer**. This layer is followed by one or more intermediate layers named **hidden layers**, which processes the inputs to generate an output to the last layer known as **output layer**.

The generalized structure of a neuron can be seen in Figure 2.3. A neuron receives a number of n input signals of value x_i from neurons in a previous layer, each with an associated weight θ_i ($1 \leqslant i \leqslant n$) to them and an inclusion of a bias term b and a bias unit x_0 [10]. The weights determine the importance between a set of neurons in regard to specific features in the input, whereas the bias resolves cases where all inputs to the neuron is 0. The bias is also capable of (similar to a linear regression model) shifting the hyperplane in the multi-dimensional solution space, providing the model with more flexibility to find an optimal solution [11]. For the neuron to pass the signal forward to the next layer of the network, the neuron must first be activated by meeting a certain threshold, like the axon in a biological neuron. This is determined by applying an **activation function** α to the input of the neuron (i.e. the weighted sum of all n inputs plus bias). The output of one neuron z of the j:th layer can therefore be described as:

$$z = \alpha(bx_0 + \sum_{i=1}^{n} \theta_{ij} x_i) \tag{2.1}$$

In order for a neural network to extract complex features, non-linearity must be introduced to the network. Otherwise, the network becomes nothing more than a complex logistic regression model. This is often done through the various activation functions the network architect can choose from. Although activation functions can behave differently and the choice(s) will

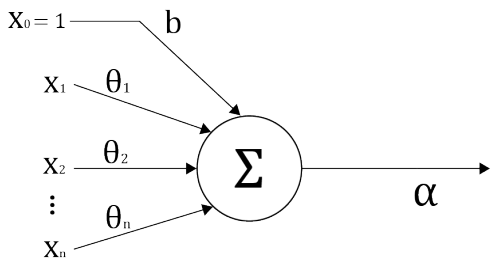


Figure 2.3: Representation of an artificial neuron in a neural network. The neuron receives a number of n inputs of value x_i with an associated weight θ_i ($1 \leqslant i \leqslant n$) and a bias term b. The weighted sum of the input is thereafter applied to the activation function α which determines whether the neuron should pass information to whichever neuron(s) it is connected to.

have great impact on the network, there are some distinct similarities between them. In particular, all activation functions should be continuous and differentiable (as will be explained later, differentiability is an important and necessary property for training a neural network). Another similarity is how the functions choose to map their output, as most activation functions ranges between [-1, 1] or [0, 1].

Feedforward networks can be used in a variety of situations to solve various task, but networks always tries to approximate some function $f^*(x)$ known as the *hyperthesis* where x is the input [10, 12]. Assuming that the task involves classification using a **supervised learning** approach (i.e. the model is trained by providing the sought output for each input sample), a network tries to map an input x to a label y using the function $y = f^*(x)$. For such a task, the network defines a mapping $y = f(x; \theta)$ where weights θ learns the best approximation of the said function $f^*(x)$. This is done though training on a set of input data, where each input has an associated label which represents the **ground truth** or $f^*(x)$. The models are called *networks*, because they are typically represented as a chain of multiple different functions $f^{(n)}$, where $f^{(1)}$ denotes the input layer. The number of functions in the chain determines the **depth** of the network i.e an n layer deep network can be described as $f(x) = f^{(n)}(f^{(n-1)}(...f^{(1)}))$ whereas the **width** is determined by the dimensionality of the hidden layers.

Training a feedforward network requires passing the output from the output layer through a **cost function** $J(\theta)$ (sometimes referred to as **loss function** or **objective function**) [10]. The cost function is used to measure the difference between the output from the output layer and the ground truth. This measurement directly specifies what the output layer must do for each input x with its respective label y. As for the other layers, the behavior is not directly specified. Because the training of an network does not specify a desired output of these layers but rather learn how to use them, is why they have gotten the term *hidden*. During training the cost of $J(\theta)$ is referred to as **empirical loss** (or **training error**), where minimizing the empirical loss will increase the accuracy of the network [Note: here are instances where training a network to achieve as small of an error as possible can lead to **overfitting** as will be explained in Section 2.4] [13]. Minimizing the empirical loss can be done by computing the gradient and update the weights θ according to a given **optimization algorithm** (or **optimizer** for short). One method of iteratively regulating the weights in regard to the gradient during training is called **gradient descent**. Gradient descent initializes the trainable weights θ to some random number (research has shown that initializing random values according to some distribution

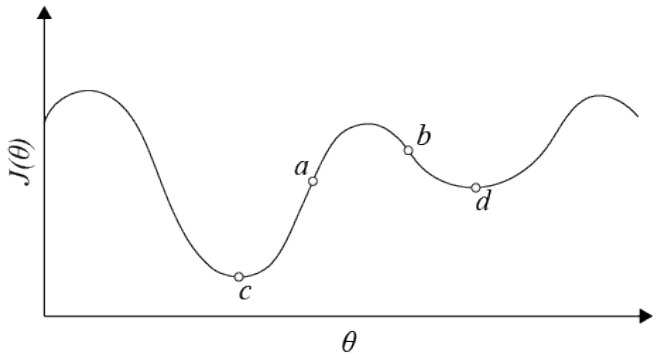


Figure 2.4: Simplified example of gradient descent, where traversing in the negative direction of the gradient, and given a sufficient step size, the algorithm will converge towards a minima. Example: initializing weights θ at a random value a will converge to the global minima c while value b will converge to the local minima d.

or using pre-trained weights [14, 15] can reduce training time and increase accuracy) and change the value by a proportional amount to the negative gradient:

$$\theta(t+1) = \theta(t) - \eta \frac{dJ}{d\theta} \tag{2.2}$$

where η is a small negative number known as **learning rate**. Figure 2.4 shows a simple example of the algorithm in a 1D space, where given a random value a on function $J(\theta)$ and a small learning rate, gradient descent will converge to the global minima c as $t \to \infty$. Note that this is not always the case, as given the initial value b of $\theta(t)$ will converge to the local minima d.

Calculating the gradient for the gradient descent algorithm requires careful application of the chain rule and hard-coding the explicit expression. Therefore, applications uses the **back-propagation algorithm** (or **backprop** for short) which is a generalization of the derivatives of the network [16]. The algorithm is an iterative process, which starts at the output layer and continues till it reaches the input layer. For each iteration, the partial derivative of the cost function in respect to the trainable parameters of the j:th layer is calculated.

2.2 Convolutional neural network

Convolutional neural networks (CNN) are a specialized kind of neural network, which provides superior performance on data that has a grid-like topological structure, like images [10]. The definition of a CNN, in contrast to a traditional feedforward network, is that a CNN employs the mathematical operation of **convolution** instead of the traditional matrix multiplication in at least one of its layers. This makes it possible for the network to extract aspects (or features) from images and differentiate between different input images. Convolutional- along with **pooling** operations reduces the amount of parameters that a feedforward network otherwise would have, while extracting the important information and making the input images smaller [17]. Thus, CNNs makes it possible to compute large-scale color images that a traditional feedforward networks would struggle heavily with, in regard to computational complexity. The parameters in a CNN should not be spatially dependent, which makes it a good fit for image classification. It does not matter where in the image an object can be found. The CNN starts with extracting low level of features (general shapes) and continues to extract higher levels of features (details) deeper into the network. What follows after a series of convolutional and pooling layers are fully connected layers which makes up the classifier of the network. The classifier takes an abstraction of the input images, generated from the various

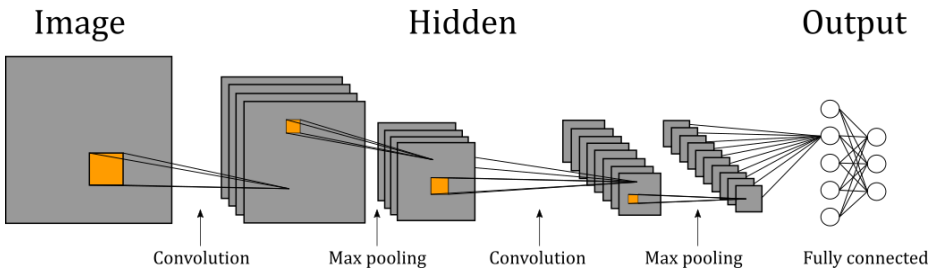


Figure 2.5: An overview of a traditional convolutional neural network (CNN) with a mix of convolutional and pooling layers in the beginning of the network, followed by a fully connected layer.

convolution- and pooling layers, to deduce a label [18]. The structure of the fully connected layers follows the architecture of traditional feedforward networks [19]. An overview of a general CNN can be seen in Figure 2.5.

2.2.1 Convolution operation

Convolution is the mathematical linear operation of generating a function $s(t)$ from two existing functions $x(a)$ and $w(a)$. This can be defined as the integral of the product between two functions, where one function is reversed and shifted:

$$s(t) = (x * w)(t) = \int_{-\infty}^{\infty} x(a)w(t-a)da \tag{2.3}$$

For convolutional networks, the *input* is an image equivalent to input x in Eq. 2.3 and the second argument w is a *kernel*, as can be seen in Figure 2.6. The output s is often called a **feature map** or **activation map** [10]. Taking the integral assumes that measurements are provided at every instant, which is not possible when computers are only capable of handling discrete values. Therefore, the convolution operation is discretized as a summation of samples at regular intervals:

$$s(t) = (x * w)(t) = \sum_{a=-\infty}^{\infty} x(a)w(t-a) \tag{2.4}$$

In machine learning applications, images and other intrinsic structured data are often represented as multidimensional datastructures (tensors) [20]. For images, this refers to the spatial dimensions as well as the depth i.e. the different color channels. Because the elements in the input and kernel needs to be explicitly defined, values which are not within the finite set are assumed to be zero. This means that in practice, the infinite sum in the convolutional operation (Eq. 2.3) can be defined as a summation over a finite number of elements. Performing convolution over two dimensions simultaneously (this would mean the spatial dimensions of an image) between an input image I and a kernel K can be described as:

$$S(i,j) = (I * K)(i,j) = \sum_{m}\sum_{n} I(m,n)K(i-m, j-n) \tag{2.5}$$

The regular steps with the kernel K over image I is referred to as **stride**. The spatial dimensions ($m \times n$) and data of the activation map, in regard to the input image, is dependent on the kernel size and the hyperparameters **zero-padding**, stride and depth used in the convolving process [18]. A 2D example of the convolution process using a 2×2 kernel with unit stride and zero padding can be seen in Figure 2.6.

Learnable kernels (i.e. kernels which weights are altered in the training process) are used in convolutional layers to find features specific to the input image. What kind of feature a kernel picks up depends on the kernels structure, examples of this can be seen in Figure 2.7 where

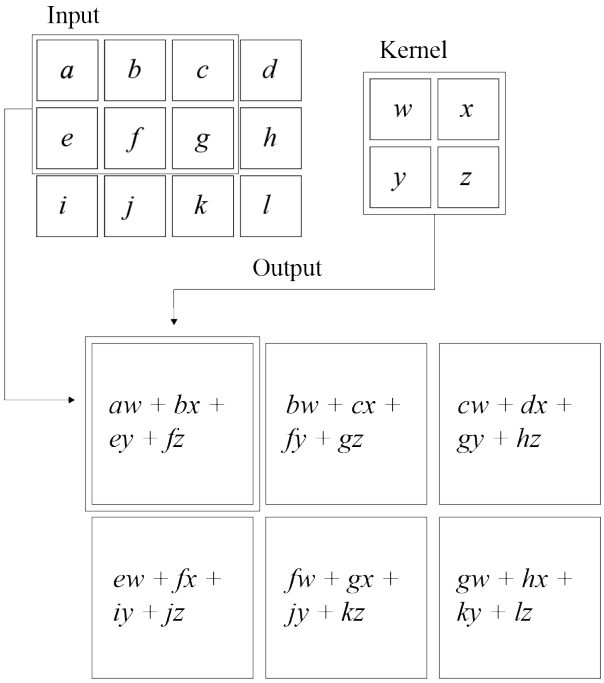


Figure 2.6: An example of 2D convolution with unit stride and a 2×2 kernel, where the output is restricted to the area in which the entire kernel lies within the image. Note this process is without flipping the kernel. Boxes represents how upper-left area of the output tensor is formed by applying the kernel to the corresponding area in the input image.

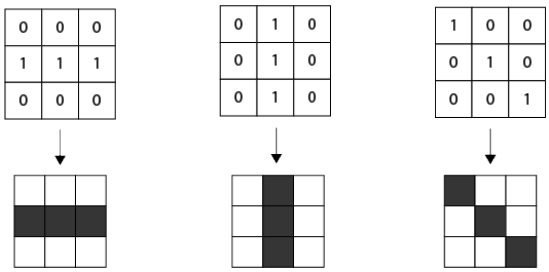


Figure 2.7: Three examples of edge detecting kernels of size 3×3. (*Left*) Kernel is capable of detecting horizonal edges. (*Middle*) Kernel is capable of detecting vertical edges and (*right*) kernel is capable of detecting diagonal edges.

the kernels are capable of picking up horizontal-, vertical- and diagonal edges respectively. The weights of the learnable kernels are adjusted similar to feedforward networks through back-propagation [21], but the key difference being that the amount of learnable parameters in each neuron are significantly reduced. This is accomplished by letting the kernels be significantly smaller than the input [18]. The difference can be significantly large considering two interactive layers consisting of m outputs and n inputs would require $m \times n$ parameters whilst k outputs ($k \ll m$) would only require $k \times n$ parameters. This lesser interaction between neurons in different layers are referred to as **sparse interaction** (or **sparse connectivity**) where the connectivity to a neuron from the previous layer is its **receptive field**. In order to avoid having a set of weights for each position in the input, the convolution layers utilizes **parameter sharing** to just learn a set of parameters. Parameter sharing refers to using the same parameters for more than one function in a model since a feature can be repeated several times in an image [10]. This is made possible since a kernel traverses the entire image.

2.2.2 Pooling

A convolution layer in a CNN can be broken down into three stages as seen in Figure 2.8. The first stage involves performing a series of convolution operations in parallel to produce a set of linear acivations [10]. In the second stage, the linear activations are passed through a non-linear activation function (introduced in Section 2.1) where the most common one is the **ReLU** (Rectified Linear Unit) function:

$$f(x) = max(0, x) \tag{2.6}$$

The reason ReLU is so common is that its first derivative is equal to 1 everywhere the unit is active. Additionally, it does not generate any second derivative effects when using a gradient based optimization algorithm (as the second derivative is equal to 0) and it is fast to compute [10]. The final stage involves using **pooling function** to tweak the output of the layer further.

The pooling function is an approach of downsampling feature maps by summarizing the information within small regions (i.e. pooling size) in a decisive manner. The operation is similar to the one of convolution, where a rectangular box (similar to the kernel) defines a region (usually 2×2 or 3×3) which slides over the feature map to generate a value for each cell in the output of the convolution layer. For example, the **max pooling** operation extracts the maximum value within the region and discards the rest of the information. Other pooling operations includes **average pooling**, which generates a single value from the average of the entire region or the taking the L^2 norm of the region. Due to the destructive nature of the pooling function, the regions are kept small and seldom overlap i.e. the stride is set to the grids size.

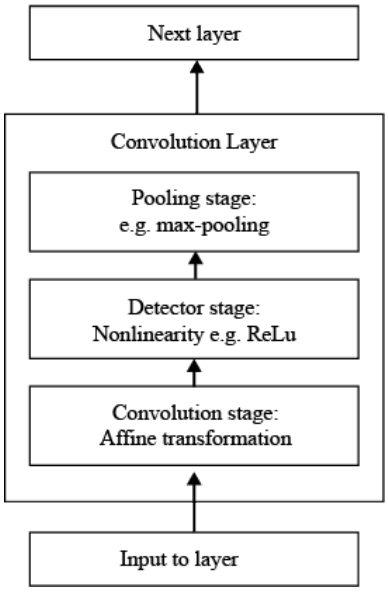


Figure 2.8: An overview of the complex layer terminology for a convolutional neural network layer. This terminology views a layer to be composed of several "stages" as opposed to viewing each stage as an entirely different layer. The stages include: Convolution, nonlinearity through an activation function and pooling.

2.2.3 Classification

What follows after a series of convolutional layers are dense layers, which is a "cheap" way of learning non-linear combinations from the high level features generated as the result of the last feature map [10]. Section 2.2.2 mentioned that the ReLU function is commonly used

as an activation function, another familiar activation function that is worth mentioning is the **Sigmoid** function:

$$S(x) = \frac{1}{1 + e^{-x}} \tag{2.7}$$

A non-linear compressive function which generates an output in the range of 0 to 1. In contrast to the ReLU function, the logistic Sigmoid function does have a non-zero second-derivative everywhere and thus introduces second-derivative effects for gradient based optimization algorithm. The major benefit of using the function somewhere in the network is that it normalizes the signal. This function can be found towards the very end of a series of convolutions or used at the end of the network in the output layer.

The dimensions of the output layer in combination with a suitable activation function is directly connected to the task which the network tries to solve. As hinted at towards the end of Section 2.1, neural networks are often used to solve classification problems. On a general level, classification tasks can be broken down into three types:

1. **Binary classification**: Where there are two classes in the target pool and the network needs to predict the input from both classes. Theses classes are often represented using a positive (1) and a negative (-1) integer value. When the network predicts the input, it produces a scalar value between [0, 1] which is done through the Sigmoid activation function. The scalar value represents the probability of the image belonging to the positive class [22].

2. **Multi-class classification**: Where there are more than two classes in the target pool. Rather than producing a scalar value, the network outputs a vector, containing the probability distribution of the different classes. The class with highest probability is selected as the predicted class [23, 24]. The values in the output vector are often generated from the *Softmax* activation function:

$$softmax(z)_i = \frac{e^{z_i}}{\sum_j e^{z_j}} \tag{2.8}$$

   The Softmax activation function is a generalization of the Sigmoid function but in multiple dimensions. The function is often used in the output layer of a neural network as it normalizes the output into a probability distribution i.e. the sum of the output is equal to 1.

3. **Multi-label classification**: Compared to binary- and multi-class classification, multi-label classification data is associated with two or more class labels. This type of classification problem is more complex. One approach of solving the classification is to make multiple binary classifications for each data sample (remember that this requires the Sigmoid activation function in the last layer) where a prediction follows a Bernoulli probability distribution [25, 26].

2.3 Optimizing a CNN

Optimizing a convolutional neural network (or any other deep network for that matter) starts at the cost function, which is a measurement of how well the network is able to match an input to the ground truth (Section 2.1). Consider the quadratic cost function $J = \frac{1}{2n} \sum_x \left\| y(x) - a^L \right\|^2$ where n is the total number of training inputs, $y(x)$ is the approximated function over training samples x and $a^L = a^L(x)$ is the vectored output from the network with L layers. In order for the cost function to be used in conjunction with backpropagation (Section 2.1) we assume it satisfies two properties [27]:

1. The cost function must be able to be written as an average $J = \frac{1}{n}\sum_x J_x$ over cost functions J_x for individual training samples x, as the backpropagation algorithm requires partial derivatives for a single training example. This requirement is met by the quadratic function where the cost of a single training sample can be written as: $J_x = \frac{1}{2}\left\|y - a^L\right\|^2$.
2. The cost function must be able to be written as a function of the outputs of the neural network. For a single training example, the quadratic cost function may be written as: $J_x = \frac{1}{2}\left\|y - a^L\right\|^2 = \frac{1}{2}\sum_i (y_i - a_i^L)^2$ and thus is a function of the output activations.

A third "unofficial" assumption is that the cost function should also not be dependent on any activation values of the network besides the output values, hence the output in the equations above is denoted as a^L. Technically, a cost function can be dependent on any activated layer a_i^j or neuron z_i^j. However, if the cost function is dependent on anything other than the values of the output layer a^L, the idea of "traversing backwards" would no longer apply to the entire network [27].

One of the major issues with the quadratic cost function is that learning can take a while before there is a rapid change in cost, if the weight initialization is far off from the target value. This is somewhat suboptimal and not related to how humans learn, as it would be more ideal to force a change on the learnable parameters early on and decrease the rate of change as the weights and biases comes closer to the optimum. This is the reason for using **cross-entropy** in most cases [10, 28]. For the different classification tasks mentioned in Section 2.2.3, cross-entropy can be broken down into two variants: **binary cross-entropy** and **categorical cross-entropy** which compares a predicted probability distribution $\hat{y}$ to a target probability distribution y. Binary cross-entropy is used in binary classification tasks (tasks which asks yes or no questions) and multi-label classification as well. Categorical cross-entropy is used in multi-class classification and can be written as:

$$J(y, \hat{y}) = -\sum_i y_i \cdot \log \hat{y}_i \tag{2.9}$$

where the predicted probability distribution represented as a vector of values ranging from 0 to 1 (assuming the last layer uses a Softmax activation function). The target probability distribution is a often represented as a vector where the target class has a probability of 1, and other classes 0.

2.3.1 Gradient descent methods

There are three variants of performing gradient descent, based on how the data is processed when computing the gradient of the cost function. Depending on the amount of data and version, the trade-off is accuracy of the updated parameters in regard to the computation time it takes to perform the update [29].

Stochastic gradient descent

Previous sections have described the optimization process as computing the gradient and updating the weights and biases for each training sample at the time. This method is called **stochastic** or **online** gradient descent (SGD):

$$\theta_{t+1} = \theta_t - \eta \nabla J(\theta_t; x^{(i)}; y^{(i)}) \tag{2.10}$$

Performing updates one at the time does introduce the possibility of **online** learning, which means new training samples can be added during training. The frequent updates also means that the learning happens fast but with higher variance and with the expense of causing the cost function to fluctuate heavily.

Batch gradient descent

On the other end of the spectrum of SGD we have **Batch gradient descent** (BGD), where updates happen once the gradient for all training samples has been computed:

$$\theta_{t+1} = \theta - \eta \nabla J(\theta) \tag{2.11}$$

Since the gradient needs to be calculated for all training samples, batch gradient is slow compared to the two other versions. The version also requires the **epoch** (one iteration over all training samples) to be fixed in size, meaning online training is not applicable to BGD. It does however come with the benefit having a steady update to the cost function.

Mini-batch gradient descent

Mini-batch gradient descent falls somewhere in between SGD and BGD, where the training set is divided into smaller groups of size n (referred to as **batch size**):

$$\theta_{t+1} = \theta_t - \eta \nabla J(\theta_t; x^{(i:i+n)}; y^{(i:i+n)}) \tag{2.12}$$

What makes the mini-batch gradient descent so common is that the method reduces the variance from updating the trainable parameters in comparison to SGD. The smaller size also makes it possible to use optimized matrix operations to speed up the calculations of the gradients. Common batch-sizes varies between 16 and 256, depending on the training set and hardware [10, 29].

2.3.2 Optimization algorithms

Optimization algorithms (optimizers) or learning algorithms are the schemes which tries to minimize the cost function over time t. As mentioned in Section 2.1 and further discussed in 2.3.1, gradient descent is an iterative optimization algorithm which updates the weights and biases according to the negative direction of the gradient, using some small learning rate. Traversing in the negative direction of the gradient of the cost function ensures that each step goes towards the local minimum. However, gradient descent is slow and often does not arrive at a critical minima point at any time. This is more applicable to deep networks than other machine learning algorithms, as the dimensions of the solution space increases due to the density and amount of trainable parameters in a deep network. According to Goodfellow et al. [10] the expected ratio between saddle points (points with zero gradient) and local minima points increases exponentially with the number of n dimensions. If the learning rate is too small, this can also cause the issue of ending up in "small" local minimas or get stuck in plateaus. To avoid the slower training process and getting stuck, one could opt to use a larger learning rate in gradient descent [Equation (2.2)] but then there is the issue of overshooting. Overshooting means that the solution does not align with the critical point (or points of a flat region in a local minima) and misses the minima, and thus continues to traverse the solution space. Both scenarios where the learning rate is either too low or too large are illustrated in Figure 2.9.

Due to these issues, various optimizers have been developed which uses **momentum** and (or) an **adaptive learning rate** to increase the possibility of converging towards an optimum fast without overshooting. One such scheme is the **Adam optimizer** [30]. Adam is a first-order gradient-based optimization algorithm that uses decaying momentum to achieve an adaptive learning rate with good performance. The update rule for Adam can be described as in Equation (2.13):

$$\theta_{t+1} = \theta_t - \frac{\eta \cdot \hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} \tag{2.13}$$

where

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t} \tag{2.14}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t} \tag{2.15}$$

and where

$$m_t = (1 - \beta_1)\nabla J(\theta_t) + \beta_1 m_{t-1} \tag{2.16}$$

$$v_t = (1 - \beta_2)\nabla^2 J(\theta_t) + \beta_2 v_{t-1} \tag{2.17}$$

where η is the learning rate depending on the momentum terms $\hat{m}_t$ and $\hat{v}_t$ described in Equation (2.14) and (2.15). Each momentum term depends on a decay rate β_1 and β_2 initialized to be 0.9 and 0.999, that decreases as t increases. The relationship between the learning rate and the two momentum terms (both initialized to < 1) ensures that the learning rate has momentum in the beginning, but starts to slow down as time increases. This helps the network of getting out of plateaus and local minimas early on in the training process [30]. Because both momentum terms also depend on the gradient, the momentum will also decrease as the network starts to reach an optimum. Both momentum terms are also bias-corrected according to Equation (2.16) and (2.17). To ensure division with none-zero in Equation (2.13), the authors of the Adam paper [30] included a small epsilon term (10^{-8} in most implementations).

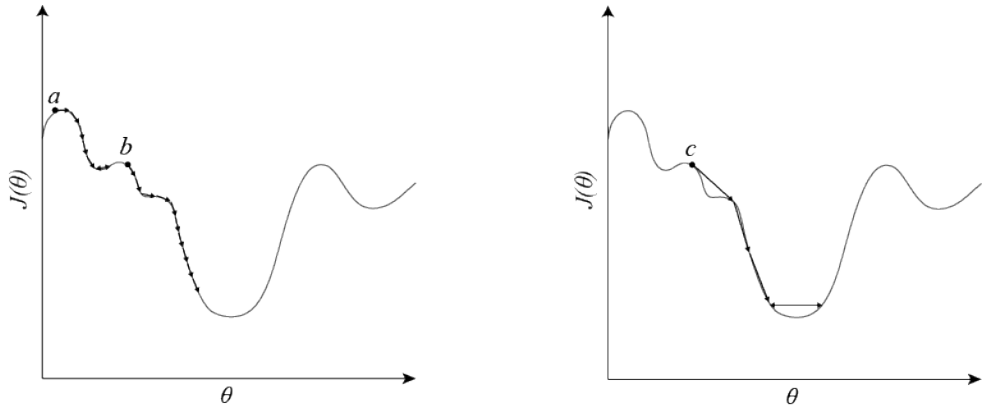


Figure 2.9: Two possible effects of choosing a sub-optimal learning rate for gradient descent. (Left) When choosing too small of a learning rate: Starting from point a, the solution can end up in a small local minima or for b, the training will take a long time and in the worst case the model will never reach a local minima in time. (Right) When choosing too large of a learning rate, the model can overshoot the local minima and miss the optimal solution. As is the case if the solution started from point c.

2.4 Optimizing a CNN until convergence

The goal of optimizing a CNN is to approximate the relationship between an input image and its label (recall how a deep neural network tries to approximate a function f^*) such that the

network can predict labels for unseen samples. The optimization process (or training process) using the methods described in previous sections can be summarized as follows:

1. Initialize the weights of the network. This is often done at random according to some distribution or using predetermined (pre-trained) values from a trained network (known as **transfer learning**) [14, 15].
2. Forward propagate data through the network. Pass an input through the various layers of the network till the signal reaches the output layer. The output of the network will be its prediction.
3. Measure the prediction using a cost function. Determine the empirical loss by passing the predicted output through a loss function.
4. Update the weights and biases with back-propagation. Compute the gradient of the cost function using back-propagation and update the weights and biases according to an optimization algorithm.

Repeating steps 2 to 4 for a set number of times will minimize the empirical loss of a network over the given training samples. The lower the empirical loss, the higher the accuracy will be over the training samples. When a network has converged, it means that a network has tweaked its trainable parameters to the points where they are no longer updated with new values i.e. the network has found a minimum of the cost function.

Although the goal should be to minimize the empiric loss, a higher accuracy on the training samples does not necessarily translate well to unseen samples. This it what can cause **overfitting**, which implies that a model does not generalize its solution well to unseen samples. This happens when a model has altered its trainable parameters such that it creates a solution that is specific to the data in the training set. On the other hand, when a model does not have the complexity to detect features in the input or if the input is lacking certain features, **underfitting** may occur. Underfitting is when the model is not capable of classifying the training data. To evaluate the generalization of the model, it is good practice to test the model on unseen data as the model starts to converge. This must be done on a separate set of data that the model has not trained on, often referred to as **test set**. In most scenarios, the models' capability of generalizing is continuously monitored during training using a third subset called **validation set**. Goodfellow et al. [10] suggests that the training- and test set should be split using a ration of 8:2 on all available data and that the training set can be split further into a training- and validation set using a ratio of 9:1. The error which the validation set yields during training is often referred to as **generalization error** (or **validation error**). An example of a typical relationship between the training error and validation error can be seen in Figure 2.10. Unless more data is provided, once the generalization error starts to deviate from the training error is the optimal time to stop the training.

2.4.1 Optimization using transfer learning

First mentioned in Section 2.1 and later referenced in Section 2.4 is the use of pre-trained weights for weight initialization i.e. transfer learning. As with good ideas, it is often the case that it exists a network that is capable of performing a certain task (or at least a similar task) to the one you are trying to solve. It is therefore common practice to take the trained weights and biases and transfer it over to your model, to decrease the time for convergence [14]. If a network is meant to recognize images that are very much the same to the images the pretrained weights are adjusted to, one can use transfer learning and freeze all the layers but the last one(s). The first layers are often convolutional layers, which has learned all necessary features needed to correctly classify new samples, while the last layers can require different dimensions to fit the desired output.

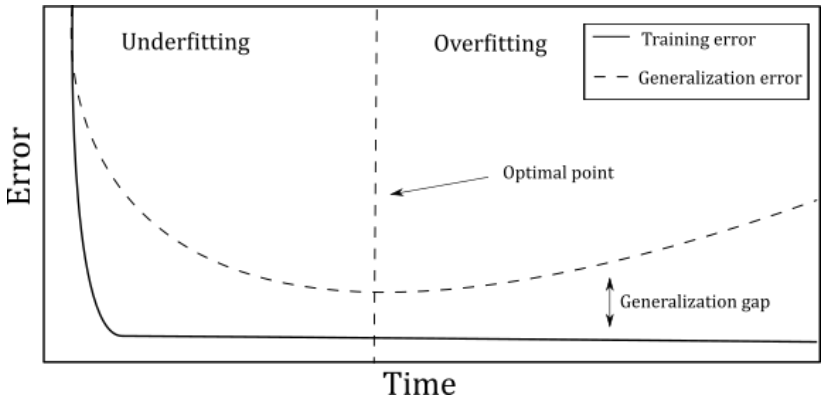


Figure 2.10: Example of training- and test error over time t and model complexity. The generalization error should follow the training error to a certain point, before the model starts to fit too much detail to the solution. This causes the model to stop generalizing and the gap between training- and generalization error starts to increase. The optimal solution in practice is therefore when the generalization error reaches its minimum point.

2.4.2 Optimize learning through hyperparameters

The definition of **hyperparameters** is variables which determines the network structure and variables which determines how the network is trained. More often than not, the networks structure is defined through some API like *Keras* [31] or *PyTorch* [32], with the exception of the last one or two layers, which are modified to fit a desired output. As for the variables which determines how the network is trained are the previously mentioned parameters, learning rate, momentum, number of epochs and batch size. A large part of improving a network's performance is done through changing the variables which determines the training. For example, changing either the learning rate or momentum may help the model to avoid local minima in the early stages of training.

2.5 Generative adversarial network

Generative Adversarial Networks (GAN) is a neural network architecture which belongs to the set of generative models. As the name suggests, the intent of the network is to generate (or produce) new data which is often used for generating images [33]. The network is composed of two sub-networks: A generator G and a discriminator D as seen in Figure 2.11. The generator network G is responsible for producing a distribution of samples $p_g = G(z;\, \theta_g)$ where z is sampled noise from a distribution $p_z(z)$ and θ_g are the learnable parameters of the generator network. Its adversary is the discriminator network, which attempts to differentiate between the samples from the training data p_{data} and generated samples p_g. The discriminator outputs a single scalar given by $D(x;\, \theta_d) \rightarrow (0, 1)$ which represents the probability that the input x comes from the training data. As a result, the two networks D and G plays a two-player minimax game, where D maximizes the probability of assigning the correct labels and G to minimize $\log(1 - D(G(z))$ [10]. This results in a value function describing the *adversarial loss* $\mathcal{L}$:

$$\min_G \max_D V(G, D) = \mathbb{E}_{x \sim p_{\text{data}}} \log D(x) + \mathbb{E}_{x \sim p_z} \log(1 - D(G(z))) \tag{2.18}$$

Theoretically, the competition should continue until the discriminator is not capable of distinguishing generated images p_g from real i.e. the global optimum is fulfilled when $p_g = p_{\text{data}}$. This leads to the discriminator predicts $\frac{1}{2}$ for all samples. However, this is seldom the case as most of the time the discriminator learns to distinguish between real- and generated images better than random guessing.

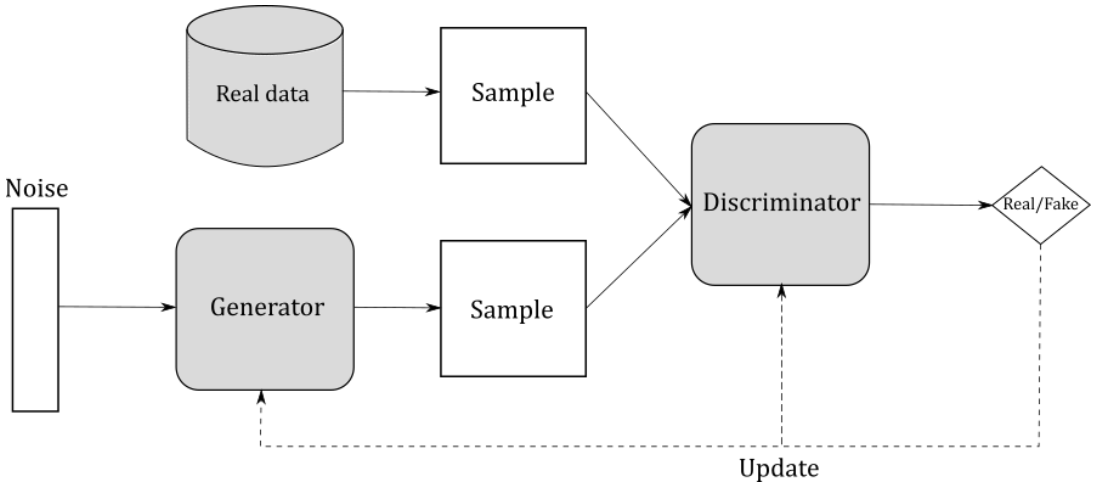


Figure 2.11: An overview of the GAN model. The network is composed of two sub-networks: A generator G and a discriminator D which competes in a minimax game.

When the goal is to generate images it is smart to use a convolutional structure in both the generator- and discriminator network. For the same reasons as stated in Section 2.2, a convolutional structure causes the networks to have a sparse interaction and parameter sharing between the layers of each network. This will let the network have fewer parameters whilst still being capable of producing a desirable accuracy. The discriminator is similar to a CNN with a binary classifier, where only wanted information is kept and the rest discarded. The generator however uses the "transpose" of the convolutional operator [10]. This means that information and detail is added continuously to the sampled noise z, when traversing the generative network. At the output layer, when an image is generated, it has all the detail, textures, lighting and object position that makes it realistic. The main function for discarding information in a CNN happens in the pooling layer, however taking the inverse of the pooling layer is not possible as most pooling functions are not invertible. An approach that has proven to be noteworthy is called "un-pooling" by Dosovitskiy et al. [34]. The approach means to take the inverse of the max-pooling under simplified conditions: For starters, the stride of the max-pooling is constrained by the width of the pooling kernel. Additionally, the maximum input is assumed to be in the upper-left corner within each pooling kernel and all non-max inputs are assumed to be 0. Even though the mentioned conditions are strict and could be considered improbable, they allow the max-pooling operator to be inverted. The layers in the network learns to compensate for the unusual output from the un-pooling approach and the total result generated by the model becomes visually pleasing [10].

The traditional GAN model poses a problem of not guaranteeing that the generated data converges towards a desirable domain, but rather produces a solution which is capable of fooling the discriminator D. Ian Goodfellow et. al. [35] recognized this and proposed an extension of their model in future work, which includes a condition x as input to the generator G, along with a noise vector z, and the discriminator D. The conditional input x can be any kind of auxiliary information [36] but requires to have some correspondence to training examples y i.e. paired data. Multiple problems in image processing can be thought of as "translating" an input image into a corresponding output image. This problem is also known as image-to-image translation where x becomes an input image which determines the loss between the generated image $G(x, z)$ and input from the training data y:

$$\mathcal{L}_{cGAN}(G, D) = \mathbb{E}_{x,y}[\log D(x, y)] + \mathbb{E}_{x,z}[\log(1 - D(x, G(x, z))] \tag{2.19}$$

2.5.1 CycleGAN

CycleGAN is an extension of the GAN architecture but here two generator- and two discriminator models are trained simultaneously, focusing on different domains. The idea is that an image generated by the first generator can serve as input to the second generator and the output from the second generator should look like the original image. The method enables training with unpaired data from different domains and can be applied to many areas of use such as style transfer, object transfiguration, season transfer and photograph enhancement.

A disadvantage with conditional GAN model is that the training data requires matching pairs, $\{x_i, y_i\}_{i=1}^{N}$. For example, this could be photos of one scene but under different weather- or lighting conditions. Zhu et al. [6] proposes an unsupervised image translation model, that learns a mapping $G : X \rightarrow Y$ between the two domains X and Y and couple it with the inverse mapping $F : Y \rightarrow X$. This exploit introduces a "cycle consistency", as seen in Figure 2.12, which enables the use of unpaired data: $\{x_i\}_{i=1}^{N} \in X$ and $\{y_j\}_{j=1}^{M} \in Y$. The cycle consistency also solves another problem. The traditional mapping $G : X \rightarrow Y$ does not guarantee that the input x and output y are paired up in a meaningful way, the same distribution over $\hat{y}$ can be induced by infinitely many mappings G. This can lead to mode collapse i.e. where all input images map to the same output image and the optimization fails to make progress. To solve this problem a cycle-consistency loss is introduced, making the translation cycle consistent. The loss encourages $F(G(x)) \approx X$ and $G(F(y)) \approx Y$, meaning that G and F should be inverses of each other. The cycle consistency loss is then combined with adversarial loss to achieve good translation.

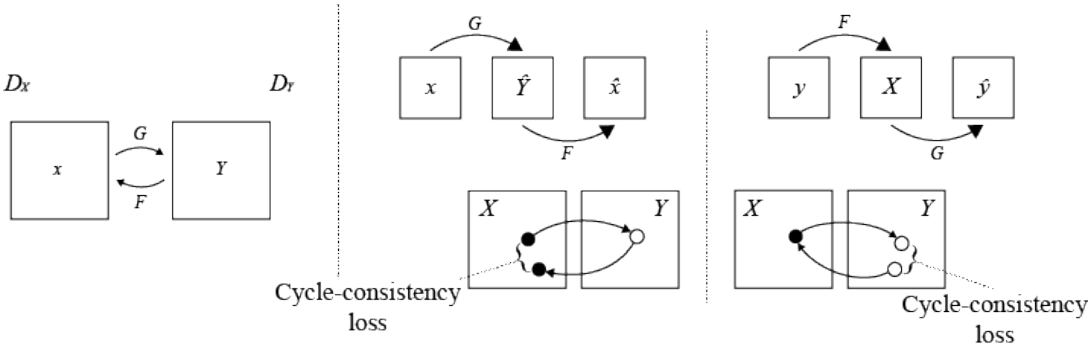


Figure 2.12: An overview of the CycleGAN model. The model is composed of two mappings $G : X \rightarrow Y$ and $F : Y \rightarrow X$ and two discriminators D_Y and D_X. D_Y encourages G to translate X into a indistinguishable result of the target domain Y while D_X tries to do the same for mapping F.

Adversarial loss

The adversarial loss is applied to both mapping functions, to match the distribution of generated images to the distribution of the data in the target domain [6]. The function $G : X \rightarrow Y$ is expressed as:

$$\min_{G} \max_{D_Y} \mathcal{L}_{GAN}(G, D_Y, X, Y) = \mathbb{E}_{y \sim p_{data(y)}}[\log D_Y(y)] + \mathbb{E}_{x \sim p_{data(x)}}[\log(1 - D_Y(G(x))] \quad (2.20)$$

where the discriminator D_Y tries to distinguish between fake samples $G(x)$ and real samples y. This encourages G to produce results that are indistinguishable from the real samples. The same adversarial loss function is also applied to the inverse mapping F discriminator D_X, resulting in $\min_F \max_{D_X} \mathcal{L}_{GAN}(F, D_X, Y, X)$.

Cycle consistency loss

The adversarial loss alone cannot guarantee that the learned mappings can translate an individual sample x_i to a desirable output y_i, which motivates the inclusion of the cycle-consistency. This means for each training sample x_i from domain X should be able to translate and be brought back to the original domain in one cycle i.e. $x \rightarrow G(x) \rightarrow F(G(x)) \approx x$. This motivated Jun-Yan Zhu et al. [6] to include a cycle-consistency loss to help incentivize this behavior:

$$\mathcal{L}_{cyc}(G, F) = \mathbb{E}_{x \sim p_{data(x)}}[\| F(G(x)) - x \|_1] + \mathbb{E}_{y \sim p_{data(y)}}[\| G(F(y)) - y \|_1] \tag{2.21}$$

The overall adversarial loss is the sum of the two equations:

$$\mathcal{L}(G, F, D_X, D_Y) = \mathcal{L}_{GAN}(G, D_Y, X, Y) + \mathcal{L}_{GAN}(F, D_X, Y, X) + \lambda \mathcal{L}_{cyc}(G, F) \tag{2.22}$$

where λ in the last term controls the relative importance between the two loss functions.

2.6 Related works

CNNs are being used in many state-of-the-art applications which requires image classification or object recognition. The reason for CNN's success when it comes to image classification is because of its capability of extracting features in images and connecting these features to the different labels provided. For instance, some CNNs have achieved beyond human-level object classification, as GoogLeNet displayed when winning ILSVRC 2014 [37]. The task presented was to categorise images into one of 1000 leaf-node categories in the ImageNet catalogue. Other areas where CNNs have been successful is in both video surveillance and autonomous driving, where classifying the weather proven to be an important step to achieve a better overall performance of the respective systems [2, 3].

2.6.1 Related works on weather classification

Elhoseiny et al. [38] studied the use of CNNs for weather classification tasks. Their approach of using a CNN outperformed previous state of the art methods, e.g. support vector machine (SVM) and Adaboost, with a normalized classification accuracy of 82.2% instead of 53.1%. In their work they analyzed the recognition performance for both pretrained ImageNet CNN and Weather trained CNN.

Zhu et al. [39] used a CNN to recognize extreme weather conditions on their dataset WeatherDataset with 16 635 images including labels: sunny, rainstorm, blizzard and fog. They split their data by assigning 80% to a training set and 20% of the images to a test set. In their work they experimented on three network structures: GoogLeNet, AlexNet and modified AlexNet, where GoogLeNet performed with an accuracy of 94.5% with fine-tuning parameters.

Guerra et al. [4] explored the possibility of using superpixel masks as a form of data augmentation to improve the performance of a multi-class weather classifier. In their work they also created an open source dataset called RFS, containing images of weather types, cloudy, foggy, rainy, snowy and sunny, as a contribution to future work in the field of computer vision. The images in the dataset contains the Creative Commons licence and are retrieved from Flickr, Pixabay and Wikimedia Commons. In their work they compared ten classification models, including: CaffeNet, PlacesCNN and variations of ResNet and VGG. The classifier model that had the best overall performance for all the settings of their superpixel masks was ResNet50.

Di Lin et al. [40] proposed a deep learning framework called region selection and cuncurrency model (RSCM) which uses regional cues for weather prediction. They evaluated their RSCM

model on a multi-class weather dataset. In their work they used a VGG-16 model pre-trained on ImageNet classification, which serves as the CNN architecture in their model, RSCM. They further mention that without pre-training, their network yields a performance drop of 9.1%.

With the success of using CNN-models for image classification tasks (including weather images) in previous works this type of classifier is used. More specifically, ResNet50 is used as it is a model with good performance overall for weather classification tasks, described in the work by Guerra et al. [4]. Transfer learning with ImageNet is also investigated, to see how the performance of the ResNet50 classifier can be improved.

2.6.2 Related works on image synthesis and imbalanced data

Zhe Li et al. [41] proposed a data augmentation method using deep convolution generative adversarial networks (DCGAN) to balance imbalanced data. They claim that most classification algorithms only perform optimally when the number of samples of each class is roughly the same and that weather datasets often are imbalanced due to sunny days being more common than rainy, snowy or hazy days. To measure the performance of their DCGAN they used a CNN model as a classifier, VGG16. The experiments showed that their GAN-based data augmentation techniques can lead to improvements in distribution integrity and margin clarity between labels.

Giovanni Mariani et al. [42] proposed a balancing GAN (BAGAN) as an augmentation tool to balance imbalanced dataset. They mention that balancing a dataset is a challenge because the few images in underrepresented labels may not be enough to train a GAN, but overcame this by including all available images from the minority and majority labels. Their generative model learns useful features from labels with more images and uses these to generate images for labels with fewer images. The datasets used were MNIST, CIFAR-10, Flowers (different labels of flowers) and GTSRB (traffic signs). For evaluation, they used a ResNet18 classifier, and compared BAGAN to other state-of-the-art GANs and showed that their model generates images with higher quality when trained on an imbalanced dataset.

As for balancing imbalanced datasets, various GANs have been used to tackle this issue. However, DCGAN performs image synthesis from random vector rather than image-to-image translation which can be a limiting factor. DCGAN does not consider image-to-image translation with unpaired data. Since Zhe Li et al. [41] showed success in their work, it is therefore worth investigating CycleGANs capabilities of performing a similar task, as it does not put the same restraints on the data collection. The same reasoning can be said for BAGAN versus CycleGAN, as BAGAN requires the data in both the source domain and target domain to be paired or under alignment [43].

2.7 Evaluation with CNN classifier

The performance of a classifier can be evaluated using a number of metrics, where the most common are *accuracy*, *precision*, *recall* and *F1-score*. The predicted labels from the classifier are directly measured against the actual label of the input and doing so over the entirety of a test set determines the performance of the classifier.

2.7.1 Precision, Recall, F1-Score and Accuracy

The precision of a said label is *the number of true positives* (TP) *divided by the total number of elements labelled as a belonging to the positive class*. Recall in the context of classification is *the number of TP divided by the total number of elements that actually belong to the actual positive class*. This means for classifying sunny weather, true positive are images that contain sunny

weather and false positive are images that are classified as sunny but contain another weather phenomenon. Other two important terminologies are *false positive* (*FP*) and *false negative* (*FN*):

$$Precision = \frac{TP}{TP + FP} \tag{2.23}$$

$$Recall = \frac{TP}{TP + FN} \tag{2.24}$$

To get a balanced estimate of how well each label performs, the F1-score gives a good estimate as it includes the harmonic mean of the two measurements. The F1-score is most useful when dealing with imbalanced samples of data, whilst the opposite is true for accuracy. Accuracy is the sum of true positives and true negatives divided by the total number of samples and is an overall estimate of how well the classifier performs over all classes:

$$F1 = 2 \cdot \frac{precision \cdot recall}{precision + recall} \tag{2.25}$$

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{2.26}$$

3 Method

This chapter conveys the different approaches used to answer the research questions presented Chapter 1. The thesis work include varying the label distribution in the training set for both the classifiers and CycleGAN model, as well as the inclusion of CycleGAN synthesized images in the training set for the classifiers. Furthermore, the thesis work also include insight in an alternative approach of normalizing signal inside both generators G_X and G_Y in the CycleGAN model. This chapter also include a description and motivation behind the use of various training parameters and frameworks as well as information about the dataset used to train all networks.

3.1 Frameworks and hardware

The scripts used for preprocessing is implemented in Python using the libraries *Tensorflow*[1] and *Keras*[2]. Tensorflow is a library that focuses on machine learning applications, especially deep learning. Keras is a library which provides an interface for Tensorflow in Python for deep learning in neural networks. The CNN classifier used is a pre-built model available from the Keras API called ResNet50 which can run on both the CPU and GPU. The CycleGAN used in this work comes from the developers Zhenliang He and Holly Grimm, who implemented the network[3] in Tensorflow 2 from the original implementations [6]. Tensorflow GPU was used to run CycleGAN on the graphics card, which gives massive parallelism and speedup in the training stage. *CUDA*-toolkit and *cuDNN*, created by NVIDIA, is required to perform deep learning on the graphics card. The training was performed on two computers using an NVIDIA GeForce GTX 1070 and NVIDIA GeForce RTX 3060 Ti.

3.2 Data

The dataset used to train all classifier instances and the CycleGAN was the RFS dataset. The **RFS dataset** is an open source dataset made by Guerra et al. [4] and Lu et al. [44], with the intent to contribute to future efforts in the field of computer vision. The name of the dataset

[1] https://www.tensorflow.org/
[2] https://keras.io/
[3] https://github.com/LynnHo/CycleGAN-Tensorflow-2

(a) Cloudy (b) Foggy (c) Rainy (d) Sunny

Figure 3.1: Sample images from four different labels in the RFS dataset.

is an acronym of the included weather labels rain, fog and snow, but also includes the labels sunny and cloudy as well. All images were retrieved from Flick, Pixabay and Wikimedia Commons under Creative Commons license, using their respective labels as search tags in different languages as well as search terms of various locations. The number of images is 1100 for each of its labels, resulting in a total of 5500 images with varying sizes. Figure 3.1 shows a sample of the images contained in the RFS dataset. The reason for using the RFS dataset is that it contained numerous quality images in various environments and settings while including all targeted labels for this work. All labels but the cloudy label were kept from the dataset for all experiments, as it is a label that could overlap with the other labels, sunny, snowy, rainy and foggy while also not being a target label. The CycleGAN model requires two labels to represent the two domains X and Y. For this work it was decided to use the sunny label as a base class (domain X), as it is often simpler to add fog or snow to a sunny image than predicting how the sky would look behind a thick layer of fog. The sunny label is also often a well represented label in weather datasets found online.

The folder structure in the RFS dataset determines which image correspond to which label. Naturally, the Foggy folder contains images where fog is overwhelmingly present and so on for each label.

```
RFS
├── Foggy
│   ├── 0001.jpg
│   └── 0002.jpg
├── Rainy
├── Snowy
└── Sunny
```

Figure 3.2: The folder structure of the RFS dataset.

3.3 Preprocessing

Prior to any work, the RFS dataset containing 1100 images in each label, were cropped to a size of 256 $\times$ 256 pixels, mainly because the time it takes to train CycleGAN is directly correlated to the number and size of the images in the training set. The dataset was then split into a **training set** and a **test set**. For this work, the split ratio was 8:2 used (as Goodfellow et al. [10] recommended being a good starting point) i.e. the training- and test set contains 80% (880 images) and 20% (220 images) of all images respectively. The test set was then set aside and only used as unseen data, to evaluate the performance of the classifier.

The training set was then used to train both the CycleGAN to synthesize new images and also the classifier under different conditions. To evaluate how CycleGAN performs under different distributions of data, the training set was reduced in steps of 25% (i.e. the training

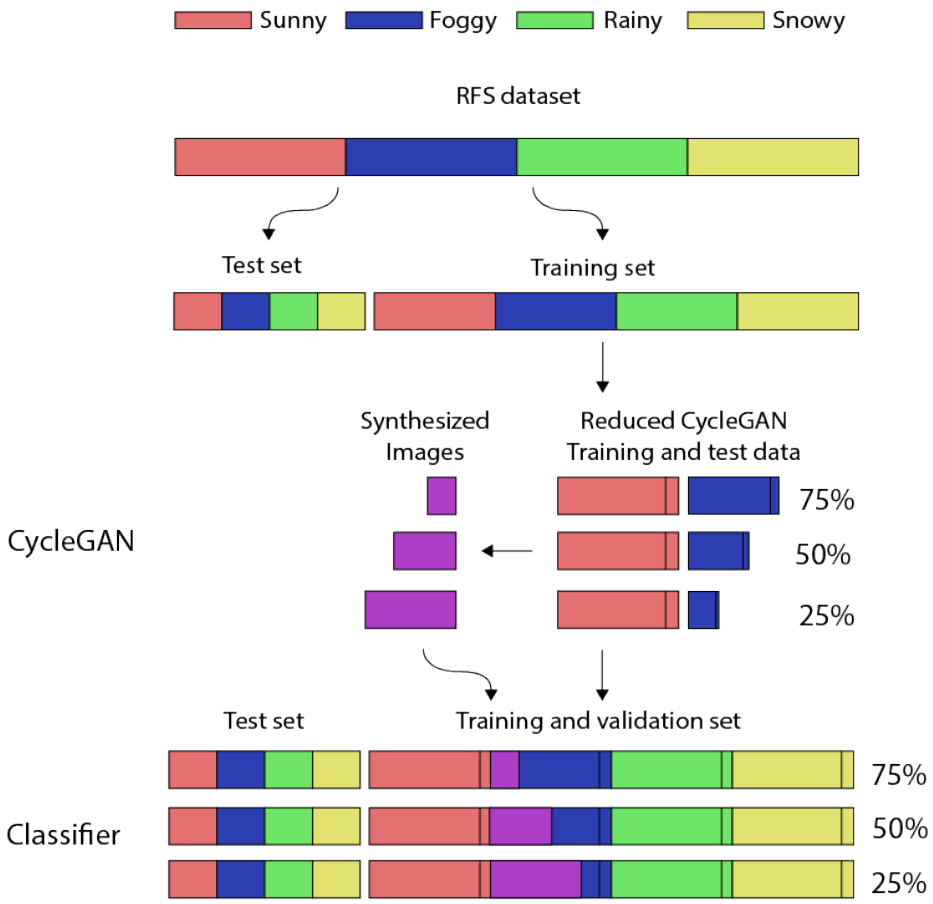


Figure 3.3: Scheme for dividing data into training- test- and validation set for this work. First the RFS dataset was split into a training- and test set using a split ratio of 8:2. The training set was then further divided into training- and test set for CycleGAN and training- and validation set using a ratio of 9:1. The ratios was chosen according to Goodfellows recommendations and how Zhu et al. set up their training in the official CycleGAN paper.

set contained 75%, 50% and 25% of the original data). Removing images for each step was done at random, this also means that the set of images for each step is also a subset to the previous one. To avoid bias, the CycleGAN model and classifier were trained on the same set of images in each step, with the inclusion of augmented images to make up for imbalanced data for the classifier. For each step, the training set was further divided into a training- and test set for CycleGAN and training- and validation set for the classifier using a ratio of 9:1 (similar to how Zhu et al. [6] conducted their training and according to recommendations by Goodfellow et al. [10]). For CycleGAN, the test set exists to ensure that the model is capable of generalizing a solution, and to ensure that the training process proceeds as wanted, without inverting colors or creating artefacts. When training the classifier, the training set was split into mini-bathes of size 16 at random. All mini-batches were also shuffled after each epoch to avoid bias during training of the classifier. The scheme can be seen in Figure 3.3.

3.3.1 Data augmentation

Data augmentation is often used in classification tasks to increase the size of the dataset artificially, which in theory could lead to improved performance as the classifier has more data to train on (overfitting). The technique can also help balance imbalanced datasets, by increasing the amount of images in the underrepresented labels. There exists a number of different

methods which can be used to augment image data. Commonly used image augmentation techniques which transforms the input image includes:

- Zooming, where some portion of the image are enlarged
- Shearing, deforming a rectangle (the image) into a parallelogram
- Rotation, random amount of rotation clockwise or anti-clockwise
- Shifting, moving the image vertically or horizontally
- Flipping, making the image look like it has been mirrored, either vertically or horizontally

All mentioned augmentation techniques were applied to a random number of images (enough to balance the dataset) at random strength, using Keras-Tensorflow's preprocessing-functions. In cases where the image is translated as a result from the augmentation techniques, points outside the boundary of the image appear with no pixel data and must be filled. Different filling modes which exists in `ImageDataGenerator` are: constant, nearest, reflect and wrap. For augmentation in this project, reflect was used, which mirrors the image that is still in the boundary, thus filling the points with pixel values instead of leaving black pixels around the augmented image.

More image augmentation techniques that adjust the pixel values in the image also exists, including: saturation, contrast, hue, brightness and noise; adding random values to random pixels.

3.4 CNN classifier

In order to measure how well the CycleGAN model was capable of producing the targeted weather labels from the sunny label, a CNN classifier was trained using various methods discussed later in this chapter. For multi-weather classification, researchers has tested and compared a number of convolutional neural networks, such as VGG-16 (and 19), AlexNet and ResNet50 (and 110) [4, 45] where ResNet50 seemed like a decent fit for two reasons: First, the ResNet50 model had achieved amongst the best (or best) result on the RFS dataset while also being part of the Keras API[4] (meaning it was not necessary to implement the network from scratch). Second, the ResNet50 model has fewer trainable parameters (27M) in comparison to its closest competitors VGG-16[5] (140M) and AlexNet (62M), although its much deeper. The reason for ResNets success is because of its architecture.

The ResNet50 architecture proposed by He et al. [46] is a deep convolutional neural network, which has 48 convolutional layers along with 1 max pooling layer and 1 average pooling layer. The 50 layer deep network is one out of a family of residual networks, which uses shortcut connections to help train deep networks that can extend up to thousands of layers and help tackle the problem of *degradation*. A notorious problem for deep networks in general, are *vanishing* (or *exploding*) gradients [47, 14]. These issues were addressed to a larger degree using normalized initialization and intermediate normalization layers (layers which normalizes the activation between hidden layers) [15, 48]. This however exposed the problem of degradation: When training accuracy gets saturated and decreases as the depth increases. This implies and that adding more layers to an already deep model leads to a *higher* training error [46, 49]. This is an indication that all systems does not optimize similarly. Consider two networks: A shallower network and its deeper counterpart. There should exist a solution where the added layers of the deeper network are **identity mappings** and the rest are

[4]https://keras.io/api/applications/resnet/#resnet50-function
[5]https://keras.io/api/applications/

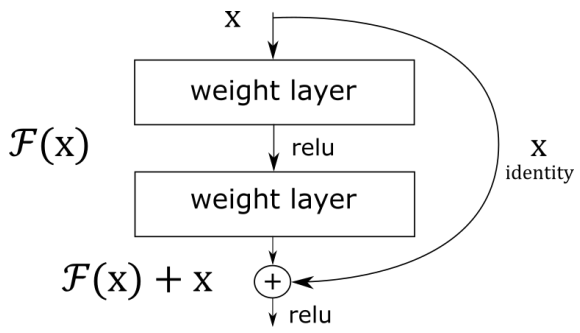


Figure 3.4: Deep residual framework: Shortcut connection which performs identity mapping.

the learned layers of the shallower network, which should prevent the deeper network from producing a higher training error.

To find a desired underlying mapping from a stack of layers, He et al. [46] proposed a **deep residual learning framework**, which explicitly lets the stacked layers to fit a residual mapping: Introducing shortcut connections which performs identity mappings. He et al. denotes the residual mapping as $\mathcal{H}$ and let the nonlinear layers $\mathcal{F}(x)$, in a stack of layers, to fit the mapping $F(x) := \mathcal{H}(x) - x$. The original mappings then becomes $\mathcal{H}(x) := F(x) + x$ as seen in Figure 3.4. Their motivation is that it is easier to optimize the residual mapping (as the residual can be pushed down to 0) than optimizing the original using none-linear layers. The benefit of their solution is that shortcut connections does not require additional parameters nor affect computational complexity while still being able to train a network end-to-end by SGD using backpropagation.

3.4.1 Implementation of the classifier

The base model for ResNet50 was found in the Keras API. The weights in the network was initialized to either ImageNet weights or random weights. An average pooling layer was added in the output layer of the network with the call `GlobalAveragePooling2D()(x)` followed by a fully connected layer with ReLu as the activation function `Dense(2048, activation='relu')(x)`. Finally, a logistic layer was added with the four classes: Foggy, Rainy, Snowy, Sunny with the call `Dense(4, activation='softmax')(x)`. The model was compiled with the categorical cross entropy loss as the loss function, because the RFS dataset include images in several labels. The adam optimizer was chosen as the optimizer with a learning rate of either 0.0005 or 0.001 for ImageNet weights and random weights respectively. See 3.1 for a table of the hyperparameters and their corresponding values.

Table 3.1: Hyperparameters used when training ResNet50 models.

Hyperparameters ResNet50	
Learning rate	0.0005 or 0.001
Optimizer	Adam
Output activation	softmax
Batch size	16
Epochs	25
Trained parameters	$\sim$ 27M
Runs for μ/σ calculations	5
Weight initialization	ImageNet or random

3.5 CycleGAN

The hyperparameters in CycleGAN were set to values according to Table 3.2. Those values were default in the tensorflow implementation and some of the values were also found in the appendix of the CycleGAN report, in the section *Training details* [6].

Table 3.2: Hyperparameters used when training CycleGAN models.

Hyperparameters CycleGAN	
Learning rate	0.002
Epochs	200
Linear decay of learning rate after # epochs	100
Beta	0.5
Adversarial loss mode	lsgan
Cycle loss weight	10
Identity loss weight	0
Pool size to store fake samples	50

3.5.1 Noise and droplet artefacts in generated images

A problem that often occurred when training CycleGAN was that the generated images contained artifacts in the form of noise and extremely bright circles (droplets). Tero Karrass et al. at NVIDIA proposed in StyleGAN2 [50], to remove instance normalization defined in 3.1 and use weight demodulation in the convolution layers as a fix to the problem.

$$y_{tijk} = \frac{x_{tijk} - \mu_{ti}}{\sqrt{\sigma_{ti}^2 + \epsilon}} \tag{3.1}$$

where k, j is the spatial dimensions, i the color channel and t the index of the image in the batch. μ is a mean value and σ the standard deviation. ϵ is a small number [51]. Karass et al. hypothesize that by creating a strong spike that dominates the image, in the form of a droplet, the generator can scale the signal as it wants elsewhere. In other words, the artifact is a result of the generator trying to sneak signal information past the instance normalization step. They further support their hypothesis by finding that the droplet artifacts are completely gone when removing the instance normalization step from the generator.

A block in StyleGAN consists of modulation, convolution and normalization. The modulation scales the convolution weights, $w' = s \cdot w$, meaning that each input feature map of the convolution is scaled based on the incoming style [50]. Assumptions are made that in CycleGAN there exists no style so therefore the scale $s = 1 \Rightarrow w' = w$, making the scaling/modulation step obsolete. Karass et al. further explain that the purpose of instance normalization is to remove the effect of s from the convolutions output feature maps. However, they mention that this can be done more directly, after the modulation and convolution step, by restoring the outputs back to unit standard deviation, presuming that the input activations are independent and identically distributed random variables. They achieve this by scaling (demodulating) each output feature map j by $1/\sigma_j$. In the implementation this is done directly onto the convolution weights as follows:

$$w''_{i,j,k} = w'_{i,j,k} \Big/ \sqrt{\sum_{i,k} w'^2_{i,j,k} + \epsilon} \tag{3.2}$$

where ϵ is a small number to avoid numerical issues, $1e-8$ in this implementation.

In the CycleGAN implementation in tensorflow, in the file `module.py`, five occurrences of instance normalization was thus removed from the generator and six convolution operations that the generator used were changed to a custom made convolution which used weight demodulation instead. The code for the weight demodulation were added in the `call` function of `Keras.layers.Conv` layer, just before the convolution operation. By making this change there were no longer any occurrences of noise or droplet artifacts which improved the overall visual quality of the generated images drastically. Hereafter, the CycleGAN that uses weight demodulation will be refered to as CycleGANWD.

3.6 Evaluating the CycleGAN images and classifiers

Evaluating the quality of synthesized images is a difficult problem [52]. Using a traditional metric such as per pixel mean-squared error does not take the structures in images into consideration and should therefore not be used as a measurement of determining the realism of images in our case. A metric that does this are CNN classifiers, which predicts labels by learning structures and patterns in the training data. In theory, if a CNN classifier were to train on a combination of both real and synthesized images of poor quality, the performance of the model should decrease as the percentage of synthesized images in the training data increases. If the synthesized images are of good quality i.e. the quality is on par with real images, the performance of the model should be unaffected.

The machine learning library scikit-learn [53] was used to calculate the metrics: precision, recall, f1-score and accuracy, using the function `sklearn.metrics.classification_report`. The function `sklearn.metrics.confusion_matrix` was also used to get an overview of the result for the classifier. To get a more credible result, the mean value was calculated for the metrics, for a chosen amount of classification runs. The standard deviation was also calculated, see Equation 3.3, to further identify if there were substantial change in the values for the metrics between runs.

$$\sigma = \sqrt{\frac{1}{N}\sum_{i-1}^{N}(x_i - \mu)^2}, \quad \text{where } \mu = \frac{1}{N}\sum_{i=1}^{N} x_i \tag{3.3}$$

σ is the standard deviation, x a value of a metric in the ith classification run, N is the amount of runs and μ the mean value for all runs.

3.7 Training

To evaluate how well the CycleGAN generated images can be interpreted as real and perform as a image augmentation technique for imbalanced datasets, tests were conducted. The tests included augmenting the RFS dataset using different image augmentation techniques and comparing the evaluation metrics from the ResNet50 classifier for each case. To compare the results of the various augmentation techniques, the ResNet50 classifier model was trained on the RFS dataset without any augmentations nor removing samples from the training set, to get a reference score for the evaluation metrics. The training set contained 792 images, the validation set 88 images and test set 220 images.

3.7.1 Imbalanced datasets

Three methods were evaluated on their ability to balance imbalanced datasets to help improve performance of an image classifier, classic image augmentation methods, CycleGAN and CycleGANWD.

However, before any classification three test scenarios were created where only 75%, 50% and 25% of the images in a chosen label in the training set were kept, and the rest removed, resulting in 660, 440 and 220 images remaining. The datasets in the three scenarios were then augmented with either classic image augmentation, CycleGAN or CycleGANWD. As a reference to these methods, a final method (hereafter referred to as Method 0) was evaluated where the images were simply duplicated in the chosen label to restore balance. After balancing the datasets for each test scenario, classification was done with ResNet50.

Method 1: Classic image augmentation

The training sets (660, 440, 220 images) were further divided into training sets (594, 396, 198 images) and validation sets (66, 44, 22 images) with a ratio of 9:1, for each test scenario. The M number of images in the new training set were then supplemented with N augmented images using the techniques described in Section 3.3.1, to balance the dataset up to 100% or 792 images. The images in the validation set was not augmented as it should represent real world data.

Method 2: CycleGAN

Similar to Method 1, the images in the training set for the targeted label were supplemented, but here CycleGAN generated images were used instead to balance the underrepresented label. A CycleGAN was trained for each targeted label (foggy, snowy and rainy) and for each test scenario

Method 3: CycleGANWD

The same as Method 2 was done here, with the exception that instance normalization was removed, and the convolution operation in the generator block were replaced with a custom convolution operation which included weight demodulation.

4 Results

This chapter presents the results of the work, showing both the synthesized images and classification results for different distributions of imbalanced data. The training times for the CycleGANs ranged from 22-27 hours while the training time for a set of five classifiers took approximately 1 hour.

4.1 CycleGAN generated images

This section shows the generated image from both CycleGAN and CycleGANWD using the Sunny label as source domain. Figure 4.1 shows the cycle translation to the Foggy label and back. In Figure 4.2, 4.3 and 4.4 images of better quality are shown when translating to the Foggy label for 75%, 50% and 25% of training data. Figure 4.5 and 4.6 shows the generated images when translating to the Snowy and Rainy label respectivly. Finally, images of worse quality are shown in Figure 4.7, 4.8 and 4.9 when translating to the Foggy label.

Figure 4.1: Illustration of the cycle translation which makes image synthesizing using unpaired data possible. The three images display the original image in the source domain (left), the translated image to the Foggy label (middle) and translated image back to the source domain (right).

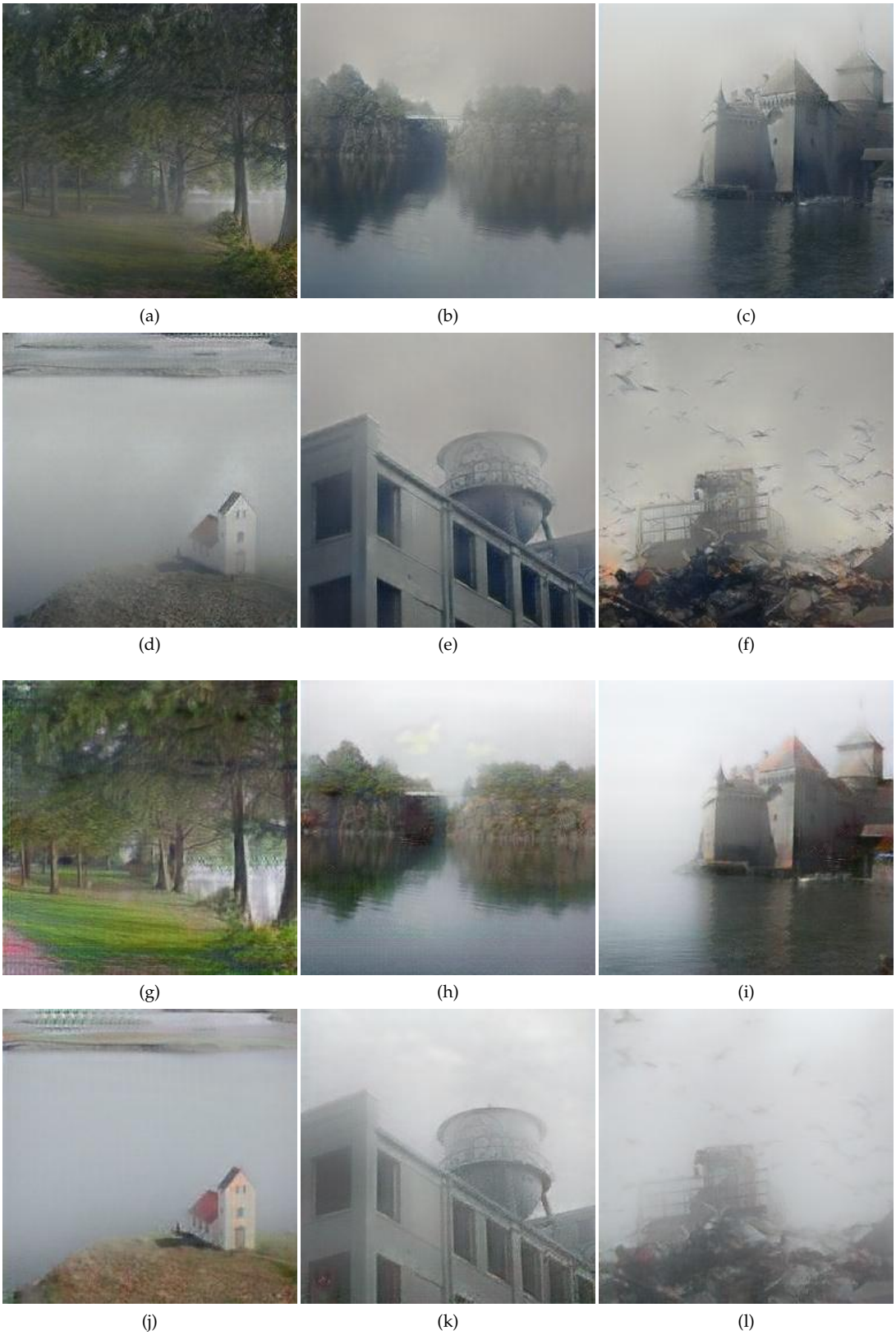


Figure 4.2: Translated images from sunny to foggy using CycleGANWD (a)-(f) and CycleGAN (g)-(l) with 75% of training data. The images are picked to showcase images of better quality.

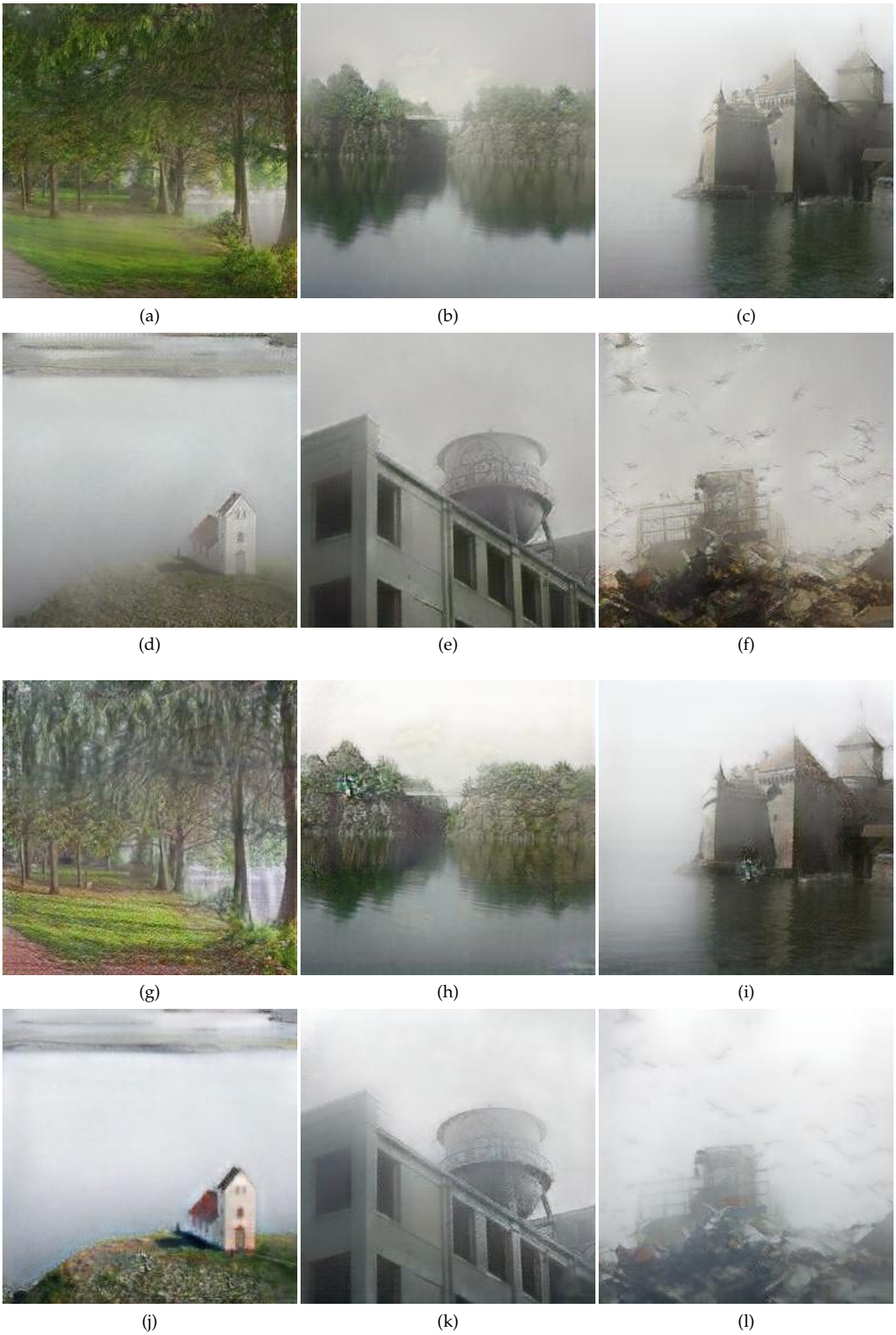
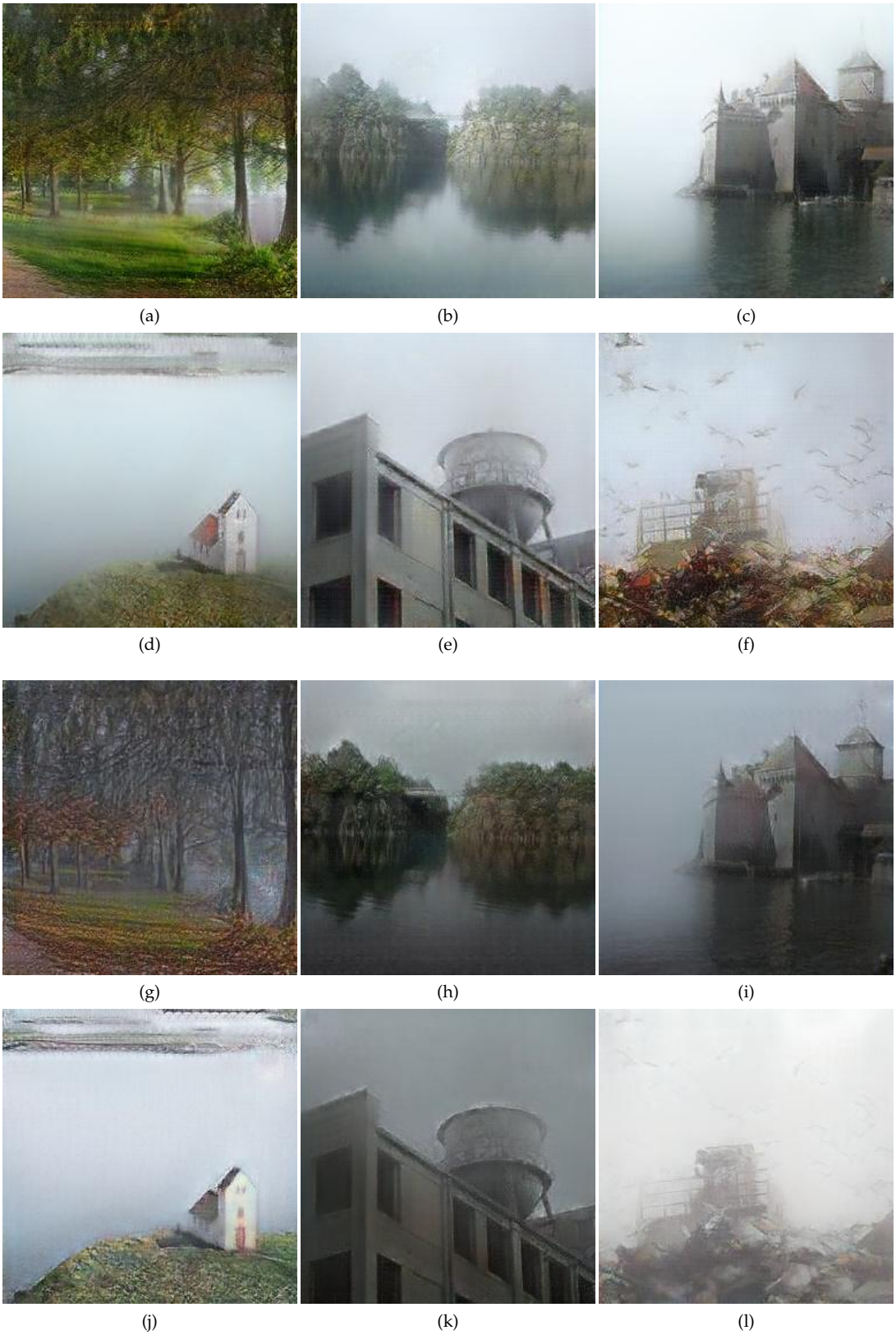


Figure 4.3: Translated images from sunny to foggy using CycleGANWD (a)-(f) and CycleGAN (g)-(l) with 50% of training data. The images are picked to showcase images of better quality.



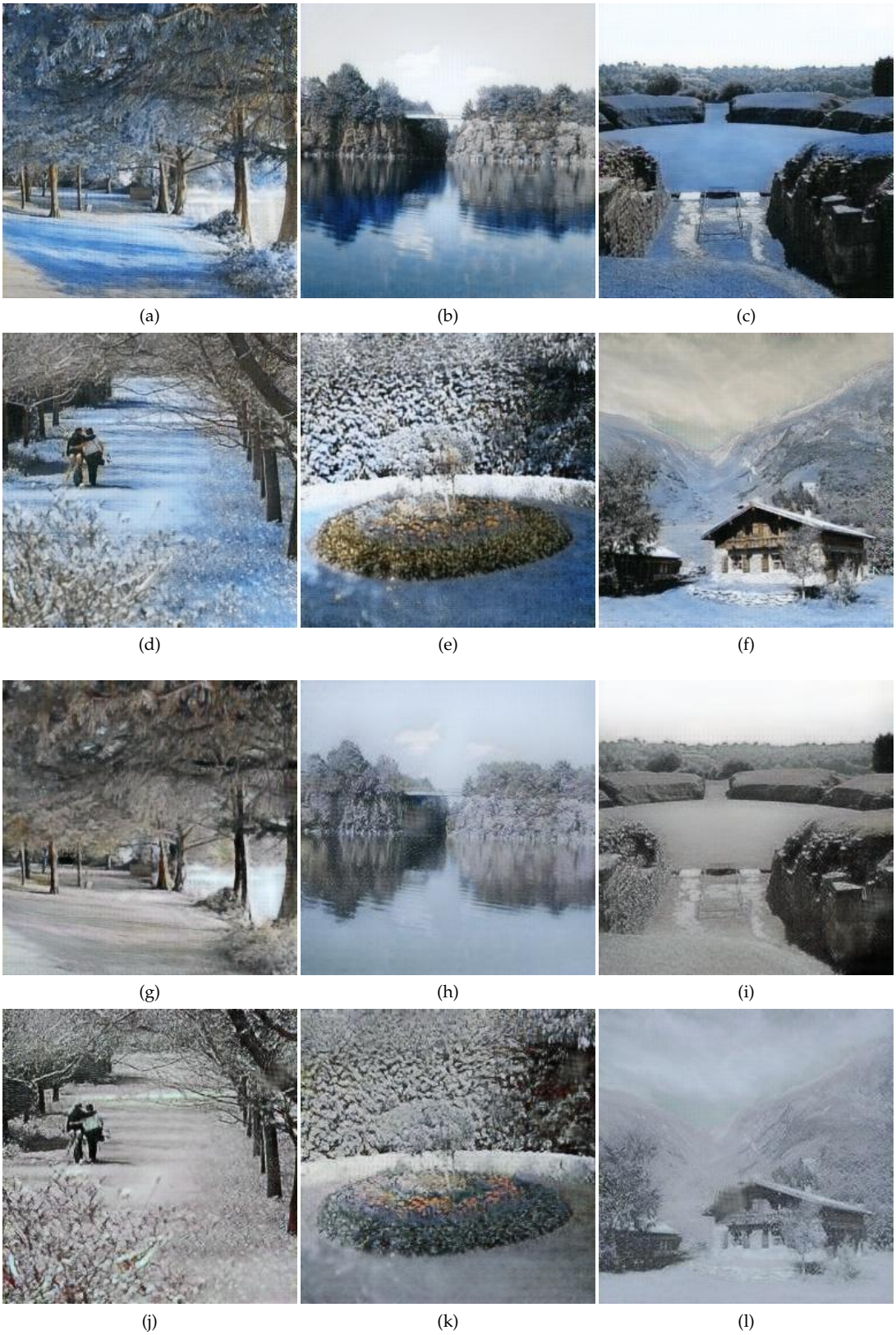
(a) (b) (c)

(d) (e) (f)

(g) (h) (i)

(j) (k) (l)

Figure 4.4: Translated images from sunny to foggy using CycleGANWD (a)-(f) and CycleGAN (g)-(l) with 25% of training data. The images are picked to showcase images of better quality.



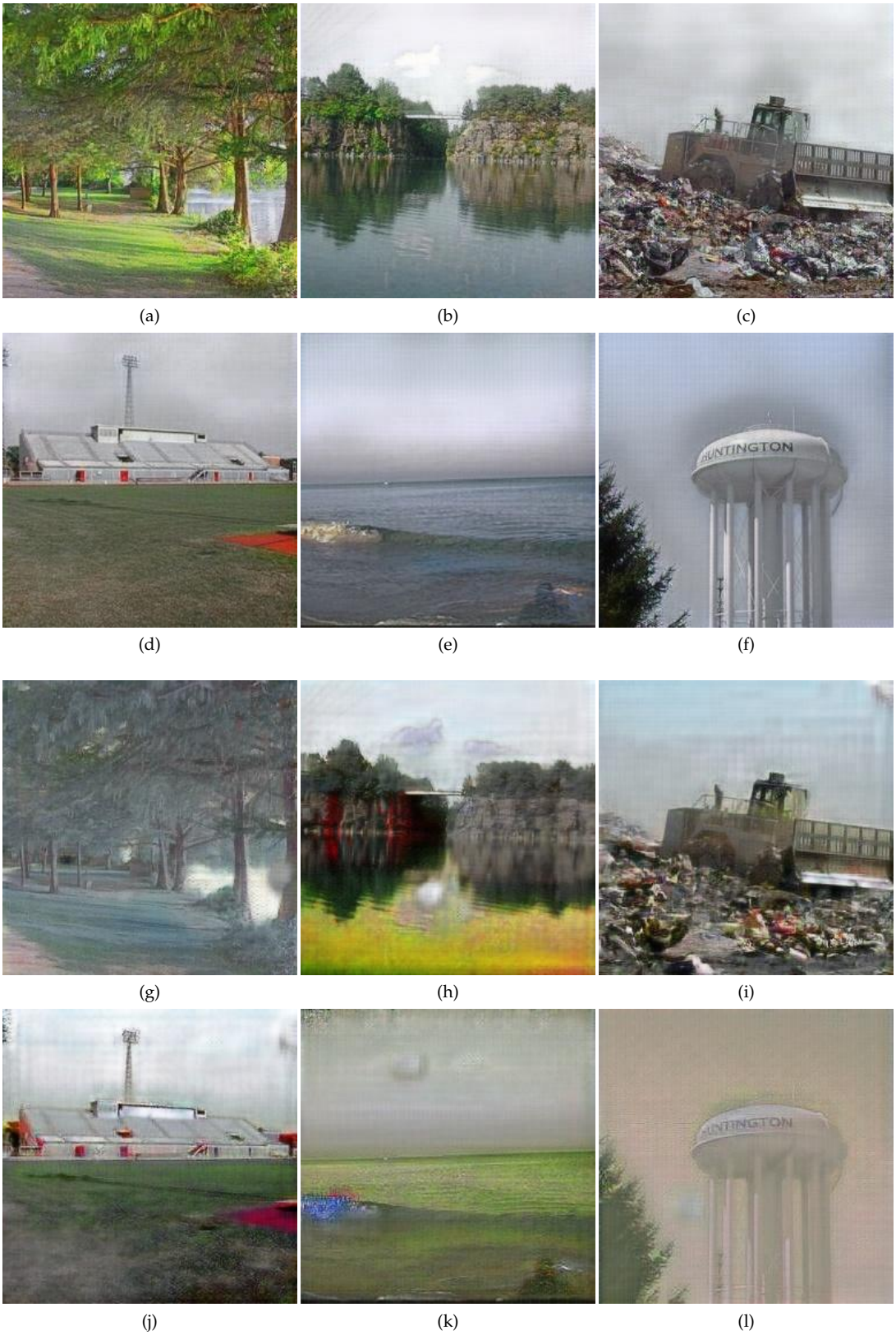
(a) (b) (c)

(d) (e) (f)

(g) (h) (i)

(j) (k) (l)

Figure 4.5: Translated images from sunny to snowy using CycleGANWD (a)-(f) and CycleGAN (g)-(l) with 75% of training data. The images are picked to showcase images of better quality.



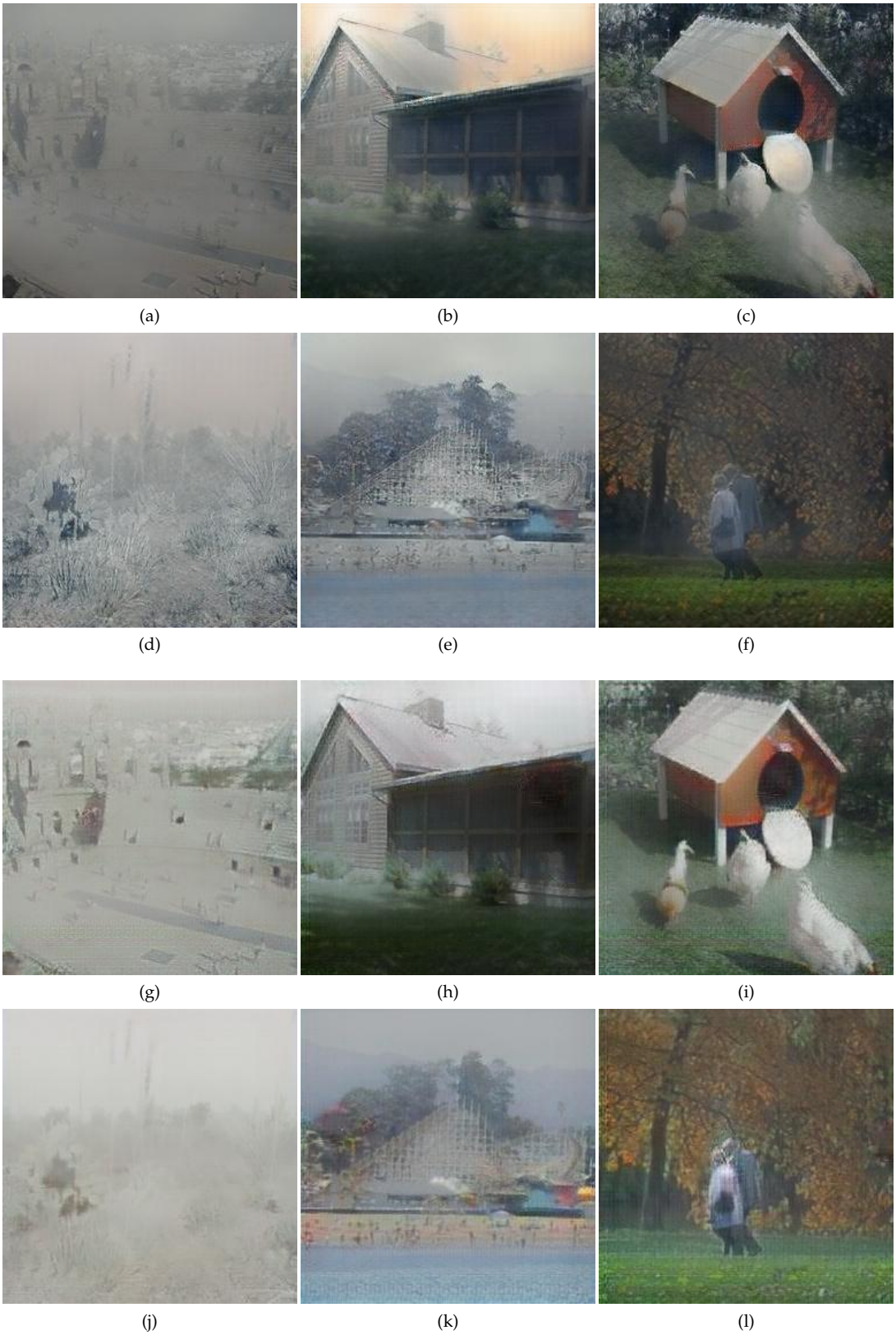
(a) (b) (c)

(d) (e) (f)

(g) (h) (i)

(j) (k) (l)

Figure 4.6: Translated images from sunny to rainy using CycleGANWD (a)-(f) and CycleGAN (g)-(l) with 75% of training data.



(a) (b) (c)

(d) (e) (f)

(g) (h) (i)

(j) (k) (l)

Figure 4.7: Translated images from sunny to foggy using CycleGANWD (a)-(f) and CycleGAN (g)-(l) with 75% of training data. The images are picked to showcase images of worse quality.

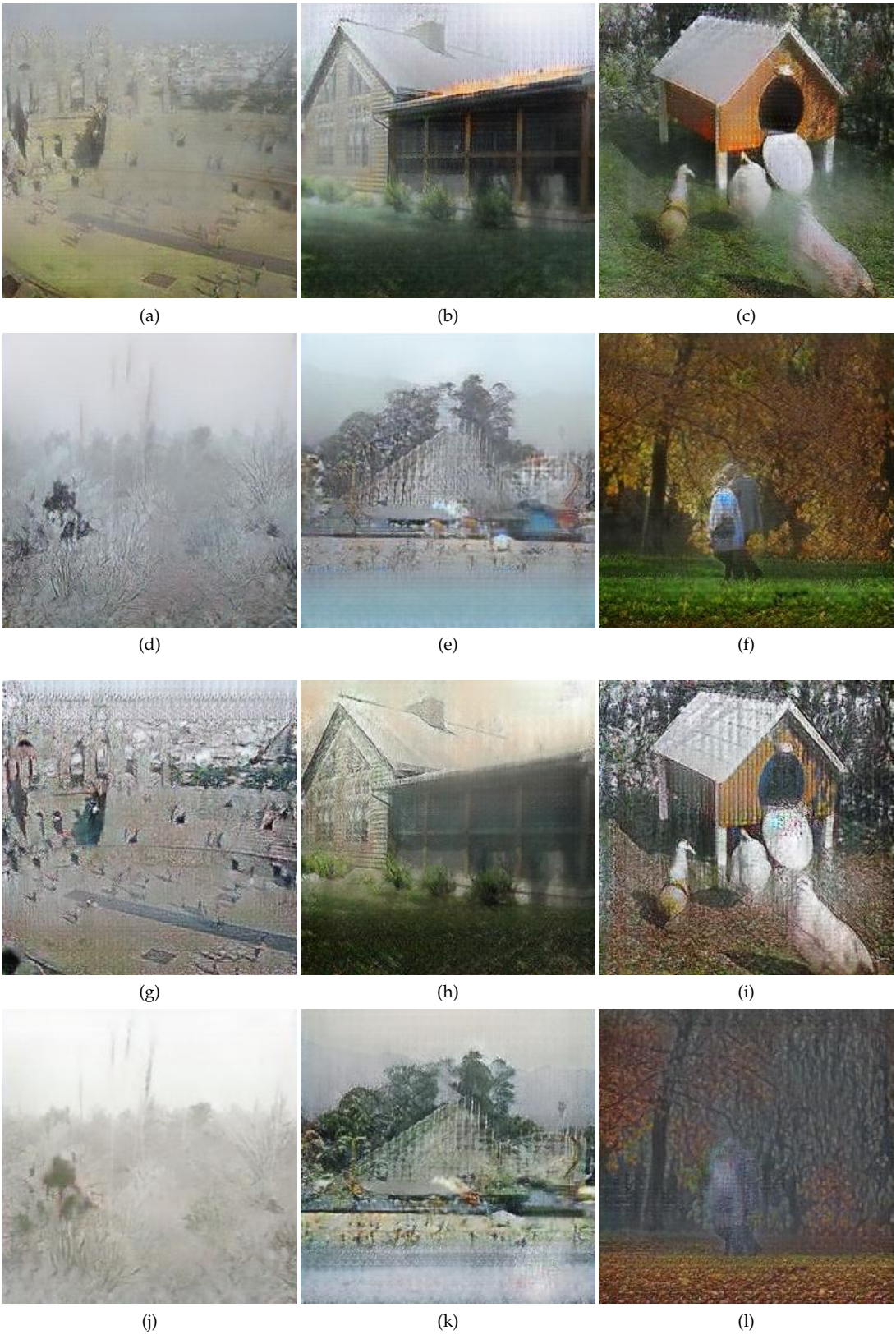
(a) (b) (c)

(d) (e) (f)

(g) (h) (i)

(j) (k) (l)

Figure 4.8: Translated images from sunny to foggy using CycleGANWD (a)-(f) and CycleGAN (g)-(l) with 50% of training data. The images are picked to showcase images of worse quality.



(a) (b) (c)

(d) (e) (f)

(g) (h) (i)

(j) (k) (l)

Figure 4.9: Translated images from sunny to foggy using CycleGANWD (a)-(f) and CycleGAN (g)-(l) with 25% of training data. The images are picked to showcase images of worse quality.

4.2 Classifier results

The following section presents the classification results for all models. Each table presents the average numerical results and standard deviations for varying numbers of real training samples, using the metrics: precision, recall, F1-score and accuracy. Tables 4.1, 4.2, 4.3 and 4.4 shows the performance of translating to the Foggy label when using the full dataset, 75%, 50% and 25% of the training data and when using ImageNet weight initialization. In Table 4.5 and 4.6 the performance is shown when translating to the Snowy and Rainy label using ImageNet weights. Furthermore, Tables 4.7,4.8, 4.9, 4.10, 4.11 and 4.12 shows the performance when using random weight initialization instead. The average accuracy and standard deviation is summarized and displayed in Figure 4.10.

Reference and imbalanced datasets with ImageNet weight initialization

Table 4.1: 100% of training data in all labels

Method	Label	Mean ± Standard deviation			
		Precision	Recall	F1-score	Accuracy
Reference: Full dataset	Foggy	0.894 ± 0.064	0.894 ± 0.062	0.889 ± 0.021	0.852 ± 0.061
	Rainy	0.847 ± 0.076	0.775 ± 0.072	0.805 ± 0.048	
	Snowy	0.927 ± 0.042	0.817 ± 0.196	0.849 ± 0.123	
	Sunny	0.818 ± 0.136	0.921 ± 0.048	0.856 ± 0.070	

Table 4.2: 75% of training data in the Foggy label

Method	Label	Mean ± Standard deviation			
		Precision	Recall	F1-score	Accuracy
Method 0	Foggy	0.941 ± 0.021	0.851 ± 0.038	0.893 ± 0.013	0.881 ± 0.008
	Rainy	0.775 ± 0.018	0.922 ± 0.014	0.842 ± 0.006	
	Snowy	0.931 ± 0.024	0.871 ± 0.036	0.899 ± 0.010	
	Sunny	0.911 ± 0.034	0.879 ± 0.031	0.894 ± 0.009	
Method 1: Strd Augment	Foggy	0.917 ± 0.019	0.879 ± 0.055	0.896 ± 0.025	0.842 ± 0.092
	Rainy	0.735 ± 0.145	0.894 ± 0.034	0.795 ± 0.090	
	Snowy	0.923 ± 0.034	0.753 ± 0.252	0.803 ± 0.206	
	Sunny	0.898 ± 0.054	0.843 ± 0.090	0.867 ± 0.058	
Method 2: CycleGAN	Foggy	0.878 ± 0.043	0.894 ± 0.048	0.884 ± 0.015	0.867 ± 0.023
	Rainy	0.858 ± 0.074	0.793 ± 0.083	0.817 ± 0.033	
	Snowy	0.899 ± 0.064	0.905 ± 0.049	0.898 ± 0.021	
	Sunny	0.869 ± 0.074	0.876 ± 0.106	0.864 ± 0.039	
Method 3: CycleGANWD	Foggy	0.921 ± 0.006	0.873 ± 0.029	0.896 ± 0.016	0.877 ± 0.027
	Rainy	0.787 ± 0.095	0.894 ± 0.070	0.829 ± 0.035	
	Snowy	0.923 ± 0.040	0.895 ± 0.068	0.906 ± 0.030	
	Sunny	0.930 ± 0.050	0.846 ± 0.078	0.882 ± 0.026	

Table 4.3: 50% of training data in the Foggy label

Method	Label	Mean ± Standard deviation			
		Precision	**Recall**	**F1-score**	**Accuracy**
Method 0	Foggy	0.918 ± 0.036	0.844 ± 0.074	0.876 ± 0.026	0.868 ± 0.022
	Rainy	0.820 ± 0.059	0.837 ± 0.078	0.823 ± 0.028	
	Snowy	0.902 ± 0.054	0.894 ± 0.081	0.893 ± 0.032	
	Sunny	0.871 ± 0.079	0.899 ± 0.051	0.880 ± 0.030	
Method 1: Strd Augment	Foggy	0.887 ± 0.070	0.859 ± 0.046	0.870 ± 0.031	0.853 ± 0.052
	Rainy	0.832 ± 0.044	0.810 ± 0.105	0.815 ± 0.049	
	Snowy	0.857 ± 0.132	0.915 ± 0.038	0.876 ± 0.067	
	Sunny	0.898 ± 0.056	0.829 ± 0.124	0.853 ± 0.062	
Method 2: CycleGAN	Foggy	0.875 ± 0.063	0.867 ± 0.072	0.866 ± 0.026	0.847 ± 0.035
	Rainy	0.834 ± 0.091	0.798 ± 0.124	0.803 ± 0.054	
	Snowy	0.891 ± 0.070	0.843 ± 0.131	0.855 ± 0.057	
	Sunny	0.844 ± 0.093	0.879 ± 0.047	0.857 ± 0.044	
Method 3: CycleGANWD	Foggy	0.905 ± 0.024	0.864 ± 0.019	0.883 ± 0.015	0.830 ± 0.008
	Rainy	0.717 ± 0.079	0.906 ± 0.059	0.795 ± 0.026	
	Snowy	0.893 ± 0.109	0.818 ± 0.101	0.842 ± 0.041	
	Sunny	0.918 ± 0.060	0.732 ± 0.111	0.805 ± 0.042	

Table 4.4: 25% of training data in the Foggy label

Method	Label	Mean ± Standard deviation			
		Precision	**Recall**	**F1-score**	**Accuracy**
Method 0	Foggy	0.926 ± 0.063	0.760 ± 0.069	0.831 ± 0.048	0.843 ± 0.049
	Rainy	0.752 ± 0.058	0.877 ± 0.026	0.809 ± 0.045	
	Snowy	0.878 ± 0.085	0.874 ± 0.085	0.870 ± 0.056	
	Sunny	0.880 ± 0.071	0.859 ± 0.149	0.856 ± 0.073	
Method 1: Strd Augment	Foggy	0.844 ± 0.127	0.825 ± 0.085	0.822 ± 0.042	0.785 ± 0.102
	Rainy	0.682 ± 0.159	0.830 ± 0.107	0.743 ± 0.134	
	Snowy	0.844 ± 0.131	0.681 ± 0.292	0.716 ± 0.236	
	Sunny	0.903 ± 0.053	0.806 ± 0.120	0.844 ± 0.050	
Method 2: CycleGAN	Foggy	0.925 ± 0.008	0.850 ± 0.032	0.886 ± 0.014	0.884 ± 0.017
	Rainy	0.861 ± 0.026	0.847 ± 0.035	0.853 ± 0.020	
	Snowy	0.860 ± 0.056	0.941 ± 0.020	0.897 ± 0.023	
	Sunny	0.905 ± 0.030	0.899 ± 0.044	0.901 ± 0.014	
Method 3: CycleGANWD	Foggy	0.902 ± 0.065	0.855 ± 0.045	0.875 ± 0.025	0.865 ± 0.027
	Rainy	0.832 ± 0.051	0.835 ± 0.079	0.829 ± 0.029	
	Snowy	0.888 ± 0.037	0.885 ± 0.082	0.883 ± 0.036	
	Sunny	0.869 ± 0.065	0.885 ± 0.063	0.873 ± 0.025	

Table 4.5: 75% of training data in the Snowy label

Method	Label	Mean ± Standard deviation			
		Precision	**Recall**	**F1-score**	**Accuracy**
Method 2: CycleGAN	Foggy	0.854 ± 0.123	0.836 ± 0.098	0.832 ± 0.053	0.816 ± 0.042
	Rainy	0.823 ± 0.075	0.705 ± 0.137	0.744 ± 0.071	
	Snowy	0.891 ± 0.054	0.809 ± 0.047	0.848 ± 0.048	
	Sunny	0.777 ± 0.098	0.913 ± 0.039	0.834 ± 0.042	
Method 3: CycleGANWD	Foggy	0.904 ± 0.047	0.900 ± 0.036	0.900 ± 0.008	0.863 ± 0.022
	Rainy	0.868 ± 0.049	0.774 ± 0.084	0.813 ± 0.029	
	Snowy	0.883 ± 0.070	0.869 ± 0.106	0.869 ± 0.052	
	Sunny	0.837 ± 0.080	0.911 ± 0.056	0.867 ± 0.028	

Table 4.6: 75% of training data in the Rainy label

Method	Label	Mean ± Standard deviation			
		Precision	**Recall**	**F1-score**	**Accuracy**
Method 2: CycleGAN	Foggy	0.826 ± 0.072	0.869 ± 0.054	0.846 ± 0.054	0.829 ± 0.039
	Rainy	0.830 ± 0.026	0.735 ± 0.085	0.777 ± 0.043	
	Snowy	0.897 ± 0.025	0.821 ± 0.069	0.855 ± 0.027	
	Sunny	0.789 ± 0.097	0.892 ± 0.025	0.838 ± 0.049	
Method 3: CycleGANWD	Foggy	0.864 ± 0.054	0.903 ± 0.039	0.881 ± 0.017	0.865 ± 0.019
	Rainy	0.857 ± 0.016	0.769 ± 0.056	0.809 ± 0.025	
	Snowy	0.869 ± 0.050	0.921 ± 0.027	0.893 ± 0.021	
	Sunny	0.879 ± 0.029	0.865 ± 0.036	0.871 ± 0.021	

Reference and imbalanced datasets with random weight initialization

Table 4.7: 100% of training data in all labels

Method	Label	Mean ± Standard deviation			
		Precision	Recall	F1-score	Accuracy
Reference: Full dataset	Foggy	0.904 ± 0.036	0.854 ± 0.100	0.873 ± 0.049	0.828 ± 0.030
	Rainy	0.830 ± 0.098	0.699 ± 0.168	0.735 ± 0.077	
	Snowy	0.849 ± 0.103	0.882 ± 0.103	0.854 ± 0.051	
	Sunny	0.807 ± 0.095	0.877 ± 0.061	0.836 ± 0.054	

Table 4.8: 75% of training data in the Foggy label

Method	Label	Mean ± Standard deviation			
		Precision	Recall	F1-score	Accuracy
Method 0	Foggy	0.896 ± 0.028	0.846 ± 0.086	0.867 ± 0.042	0.846 ± 0.033
	Rainy	0.798 ± 0.073	0.833 ± 0.070	0.809 ± 0.026	
	Snowy	0.909 ± 0.091	0.801 ± 0.102	0.843 ± 0.059	
	Sunny	0.829 ± 0.057	0.904 ± 0.047	0.863 ± 0.029	
Method 1: Strd Augment	Foggy	0.893 ± 0.038	0.862 ± 0.038	0.876 ± 0.006	0.851 ± 0.014
	Rainy	0.789 ± 0.058	0.808 ± 0.090	0.792 ± 0.028	
	Snowy	0.903 ± 0.036	0.865 ± 0.035	0.882 ± 0.019	
	Sunny	0.852 ± 0.079	0.871 ± 0.035	0.858 ± 0.030	
Method 2: CycleGAN	Foggy	0.870 ± 0.111	0.842 ± 0.075	0.846 ± 0.040	0.822 ± 0.035
	Rainy	0.744 ± 0.038	0.815 ± 0.068	0.775 ± 0.033	
	Snowy	0.933 ± 0.042	0.735 ± 0.189	0.802 ± 0.124	
	Sunny	0.815 ± 0.047	0.895 ± 0.052	0.850 ± 0.012	
Method 3: CycleGANWD	Foggy	0.845 ± 0.079	0.892 ± 0.056	0.863 ± 0.027	0.838 ± 0.039
	Rainy	0.806 ± 0.012	0.782 ± 0.073	0.792 ± 0.038	
	Snowy	0.917 ± 0.026	0.802 ± 0.092	0.853 ± 0.062	
	Sunny	0.838 ± 0.127	0.876 ± 0.050	0.847 ± 0.055	

Table 4.9: 50% of training data in the Foggy label

Method	Label	Mean ± Standard deviation			
		Precision	Recall	F1-score	Accuracy
Method 0	Foggy	0.795 ± 0.142	0.880 ± 0.054	0.824 ± 0.085	0.802 ± 0.092
	Rainy	0.771 ± 0.075	0.725 ± 0.136	0.744 ± 0.101	
	Snowy	0.863 ± 0.085	0.834 ± 0.051	0.846 ± 0.057	
	Sunny	0.866 ± 0.106	0.767 ± 0.239	0.774 ± 0.166	
Method 1: Strd Augment	Foggy	0.876 ± 0.066	0.847 ± 0.089	0.850 ± 0.024	0.845 ± 0.019
	Rainy	0.780 ± 0.056	0.855 ± 0.047	0.813 ± 0.013	
	Snowy	0.873 ± 0.102	0.859 ± 0.091	0.856 ± 0.044	
	Sunny	0.921 ± 0.029	0.819 ± 0.036	0.866 ± 0.010	
Method 2: CycleGAN	Foggy	0.859 ± 0.085	0.833 ± 0.079	0.843 ± 0.065	0.820 ± 0.057
	Rainy	0.780 ± 0.095	0.821 ± 0.060	0.796 ± 0.062	
	Snowy	0.775 ± 0.062	0.900 ± 0.032	0.831 ± 0.037	
	Sunny	0.927 ± 0.026	0.727 ± 0.121	0.807 ± 0.075	
Method 3: CycleGANWD	Foggy	0.918 ± 0.050	0.825 ± 0.069	0.865 ± 0.023	0.840 ± 0.041
	Rainy	0.744 ± 0.126	0.857 ± 0.087	0.782 ± 0.054	
	Snowy	0.908 ± 0.032	0.840 ± 0.069	0.870 ± 0.033	
	Sunny	0.885 ± 0.056	0.838 ± 0.108	0.853 ± 0.049	

Table 4.10: 25% of training data in the Foggy label

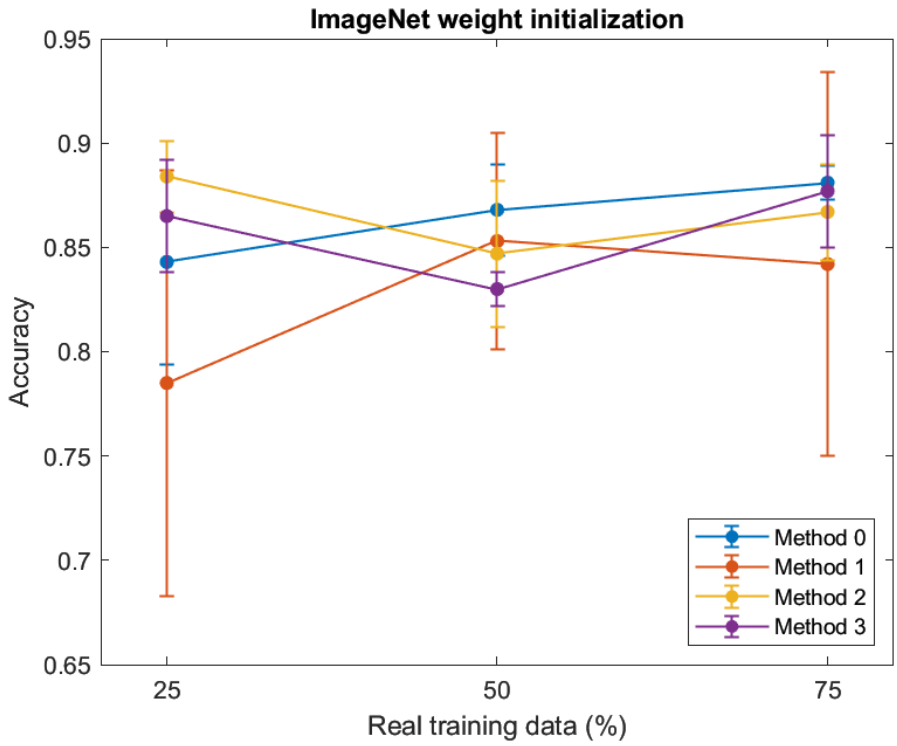
Method	Label	Mean ± Standard deviation			
		Precision	Recall	F1-score	Accuracy
Method 0	Foggy	0.932 ± 0.031	0.715 ± 0.104	0.803 ± 0.058	0.757 ± 0.088
	Rainy	0.662 ± 0.109	0.822 ± 0.156	0.712 ± 0.069	
	Snowy	0.863 ± 0.103	0.683 ± 0.209	0.738 ± 0.118	
	Sunny	0.721 ± 0.114	0.808 ± 0.214	0.758 ± 0.164	
Method 1: Strd Augment	Foggy	0.812 ± 0.115	0.847 ± 0.045	0.821 ± 0.049	0.795 ± 0.076
	Rainy	0.672 ± 0.137	0.747 ± 0.210	0.701 ± 0.166	
	Snowy	0.888 ± 0.073	0.768 ± 0.154	0.814 ± 0.100	
	Sunny	0.861 ± 0.035	0.816 ± 0.095	0.834 ± 0.050	
Method 2: CycleGAN	Foggy	0.918 ± 0.028	0.783 ± 0.072	0.842 ± 0.033	0.840 ± 0.012
	Rainy	0.747 ± 0.056	0.864 ± 0.060	0.797 ± 0.020	
	Snowy	0.856 ± 0.058	0.894 ± 0.057	0.871 ± 0.014	
	Sunny	0.900 ± 0.075	0.821 ± 0.071	0.853 ± 0.029	
Method 3: CycleGANWD	Foggy	0.883 ± 0.061	0.817 ± 0.084	0.843 ± 0.026	0.820 ± 0.049
	Rainy	0.809 ± 0.080	0.745 ± 0.120	0.766 ± 0.061	
	Snowy	0.852 ± 0.147	0.845 ± 0.125	0.829 ± 0.075	
	Sunny	0.815 ± 0.022	0.873 ± 0.102	0.839 ± 0.047	

Table 4.11: 75% of training data in the Snowy label

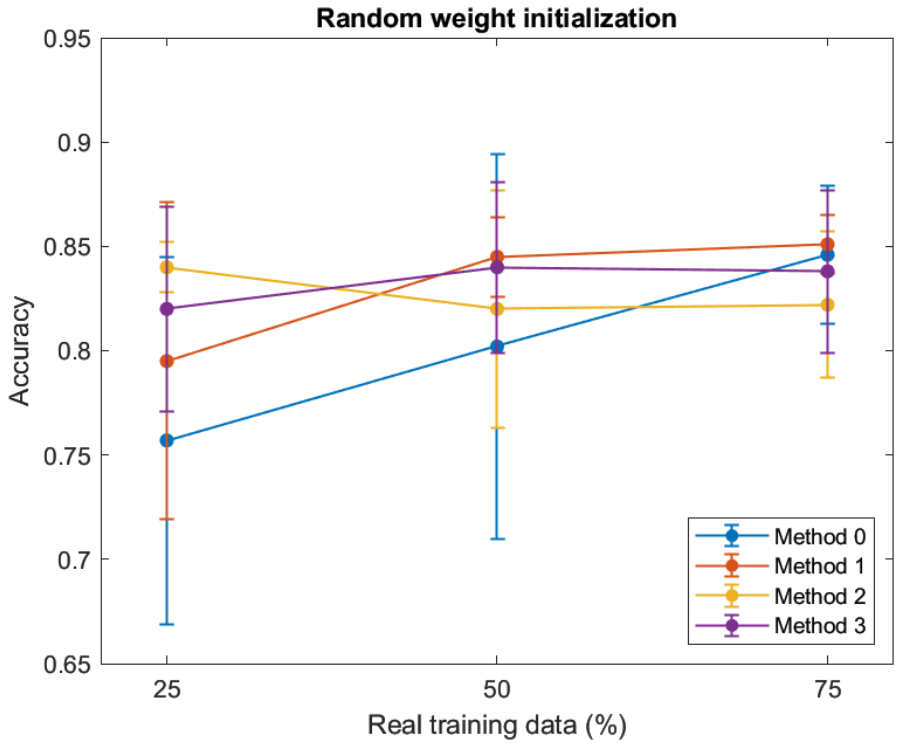
Method	Label	Mean ± Standard deviation			
		Precision	Recall	F1-score	Accuracy
Method 2: CycleGAN	Foggy	0.912 ± 0.034	0.818 ± 0.086	0.859 ± 0.042	0.831 ± 0.032
	Rainy	0.830 ± 0.076	0.769 ± 0.085	0.791 ± 0.033	
	Snowy	0.901 ± 0.034	0.805 ± 0.110	0.844 ± 0.054	
	Sunny	0.755 ± 0.091	0.932 ± 0.023	0.830 ± 0.048	
Method 3: CycleGANWD	Foggy	0.875 ± 0.046	0.866 ± 0.017	0.870 ± 0.024	0.843 ± 0.038
	Rainy	0.748 ± 0.067	0.875 ± 0.055	0.803 ± 0.038	
	Snowy	0.900 ± 0.044	0.832 ± 0.098	0.859 ± 0.050	
	Sunny	0.901 ± 0.053	0.801 ± 0.090	0.843 ± 0.044	

Table 4.12: 75% of training data in the Rainy label

Method	Label	Mean ± Standard deviation			
		Precision	Recall	F1-score	Accuracy
Method 2: CycleGAN	Foggy	0.905 ± 0.016	0.854 ± 0.047	0.878 ± 0.026	0.863 ± 0.021
	Rainy	0.809 ± 0.096	0.844 ± 0.069	0.823 ± 0.026	
	Snowy	0.854 ± 0.066	0.919 ± 0.040	0.883 ± 0.031	
	Sunny	0.905 ± 0.037	0.835 ± 0.031	0.867 ± 0.014	
Method 3: CycleGANWD	Foggy	0.880 ± 0.041	0.887 ± 0.067	0.881 ± 0.018	0.856 ± 0.010
	Rainy	0.818 ± 0.039	0.796 ± 0.078	0.803 ± 0.020	
	Snowy	0.867 ± 0.075	0.882 ± 0.058	0.870 ± 0.018	
	Sunny	0.884 ± 0.032	0.860 ± 0.050	0.870 ± 0.014	



(a)



(b)

Figure 4.10: Plot of the accuracy over the varying percentage of real training samples in the Foggy label. Subfigure (a) shows the average accuracy with standard deviation, as error bars, between the different methods using ImageNet weight initialization. Subfigure (b) shows the average accuracy with standard deviation between the different methods using random weight initialization.

5 Discussion

This chapter contains discussions about the results from using different generator architectures in CycleGAN and how they compare to using traditional augmented techniques when reducing the amount of available data. All discussions on the topic of visual quality of the CycleGAN generated images are based off the authors opinions and may not reflect the opinions of a wider audience. The method will also be discussed as well as work in wider context.

5.1 Result

What made this work interesting to pursue is the fact that CycleGAN does not need paired data in order to perform image synthesis. As mentioned in Section 2.5.1 this is done by creating a cycle i.e. translating an image from the source domain to the target domain, and back to the source domain. This process can be seen in Figure 4.1. The traditional CycleGAN implementation used instance normalization after all but the last convolutional operation in both the generators G_X and G_Y [6]. A contribution of this thesis work is insight in how weight demodulation can be used to remove (or decrease to a large degree) artifacts that appeared during training of CycleGAN. As mentioned, Karras et al. [50] experience similar issues with their StyleGAN which they hypothesized to be the generator sneaking in signal strength information past the instance normalization.

5.1.1 Visual results of CycleGAN images

All images in Chapter 4 are the result of training a CycleGAN model on sunny weather in the source domain to either foggy- rainy- or snowy weather using both instance normalization and weight demodulation in combination with different distributions of data.

5.1.1.1 Comparison between labels and distribution of target data

Both CycleGAN implementations could produce visually realistic images, but the degree of realism varied depending on the label and the number of images in the target domain. Between the different labels, the foggy label performed best, where CycleGAN WD was capable of synthesizing fog-like details to the sunny images, making contours of objects less detailed and giving the images a grayer tone. This can be seen from the hand-picked samples in Figure

4.2 which shows images where the generator tended to do well and Figure 4.7 are examples of images that tended to not do well. It can be noted that scenes which are closer to the camera are often harder to translate, which is understandable as the fog phenomenon are usually more visible at larger views and longer distances. A difficult translation of a scene with very limited view is shown in Subfigure (c) in Figure 4.7.

Comparing the results to the synthesized images from the snowy label (Figure 4.5), the quality of the hand-picked samples was on par with the best samples from the foggy label, where grass and trees gained white details and the images got a bluer and colder tone. However, the overall quality of the snow images were not as good as the foggy images. The same could not be said for synthesized images from the rainy label where all samples had a grainy tone and tended to create artifacts around objects (Figure 4.6), without any addition of rain droplets. When reaching a critical number of training samples, it becomes clear that both models struggle creating fog-like details and instead resorts to manipulating the colors more aggressively. This is most prominent in Subfigure (h) and (i) in Figure 4.9 and Subfigure (g) and (i) in Figure 4.4.

The effect the number of images in the target domain had on the synthesized images came to no surprise. As the number of images in the target domain decreased, so did the visual quality of the synthesized images. This was apparent for both implementations of the CycleGAN model, as can be seen in Figures 4.2 to 4.4 and 4.7 to 4.9. From these results it is possible to see that the fog-like details become less prominent, and checkerboard artifacts starts to appear as the amount of images in the target domain decreases.

5.1.1.2 Comparison between CycleGAN and CycleGANWD

The standard CycleGAN model often had difficulty translating images with fine detail, bright spots or where objects were close to the camera. This often resulted in grainy, noise-like artifacts or even to the extent of droplet effects. The result of noise appearing early in the training can be seen in in the top-right corner of Subfigure (j) in Figure 4.2. Similar artefacts have also appeared in Subfigure (h) and (i) figure 4.3. Finally in Figure 4.9 the translation has failed completely in Subfigure (i) and (l).

Comparing CycleGAN to CycleGANWD, one can draw the conclusion from the images in Chapter 4 that the variant with weight demodulation can handle difficult translations better and does not produce as much noise like artefacts, meaning that the overall visual image quality is higher than the standard CycleGAN. However, the standard CycleGAN can still produce realistic images, under the circumstance that no artefacts appear, shown especially in Subfigure (k) and (l) in Figure 4.2. The biggest difference between the two methods can be seen when synthesizing rain- and snow-like weather. In Figure 4.6, showing generated rainy images, there are clear evidence of discoloring when using CycleGAN, especially in Subfigure (h) and (k) where the water got an unnatural yellow- or green tone. For snow-like weather it is almost the other way around, where CycleGANWD appears to have made the images more vibrant with a blue tone and CycleGAN has made the images more plain. This is most evident when comparing Subfigure (a) and (g) or (b) and (h) to each other in Figure 4.5. In Subfigures (h), (k) and (l) CycleGAN appears to have generated good images overall, with the exception that there exists artefacts in the middle of (h) and (l).

5.1.2 Metric results from classification

It should be pointed out right away that the performance of the full dataset are often outperformed by the methods throughout all data distributions. This should ideally not be possible, as no method should be better than having a larger dataset with real images. This suggests

that the metrics, including precision, recall, f1-score and accuracy, has a rather large window of fluctuations, which is something to keep in mind when comparing the results.

Shown in Section 4.2 are the results from using the ResNet50 classifier initialized with ImageNet weights. By examining the scores for the evaluation metrics for the Foggy label, one can see that all evaluation metrics have similar score for all methods. However, the results when having 25% of the training data stand out, seen in Table 4.4, more specifically the F1-score for Method 0 and Method 1 for the Foggy label seems to have dropped significantly compared to Method 2 and 3. Another difference is seen in the standard deviation of the accuracy, where both CycleGAN methods showed a considerably smaller fluctuation than using Method 0 and 1. This suggests both CycleGAN methods of balancing data could yield better results when data is very scarce. The overall similar scores could be a product of using the pretrained ImageNet weights, meaning that all the methods already are too well traine to differ much in performance.

Shown in Section 4.2 are the results from using the ResNet50 classifier initialized with random weights. In this case Method 0 has a trend to decrease in performance as the real images in the training data decreases for the Foggy label, seen in Table 4.10 when examining the F1-score compared to the other methods. In theory, Method 0 should be the method that performs the worst, especially when the amount of images in the training data decreases, because no new images with new information are added. Even though the dataset is balanced, by duplicating images, the classifier only trains on a small sample of unique images. The standard augmentation techniques, Method 1, has a decent performance throughout all tests except when using 25% of the training data, shown in Table 4.10. Method 2 and 3, CycleGAN and CycleGANWD, both have equal classification performance throughout all tests, seen in Table 4.8, 4.9 and 4.10. Noteworthy is that the F1-score in the Foggy label seem to quite stable throughout the tests with different training data for method 2 and 3, both with the pretrainied ImageNet weights and random initialization. Even though the CycleGAN sometimes generates images containing noise and artifacts it does not seem to impact the performance of the classifier when comparing with CycleGANWDs classification results.

Comparing the results between using pretrained ImageNet weights and random weight initialization, the conclusion can be drawn that using pretrained ImageNet weights generally yields a higher performance for all methods and data distributions, this can be seen in Figure 4.10. However, as a result of using pretrained weights, it is difficult to conclude which method performs the best as a tool to balance imbalanced data. By also training the classifiers with random weight initialization more evidence could be collected that further shows that method 2 and 3 performs the best when training data is scarce. By examining the results from both types of weight initialization it suggests that the CycleGAN methods could be used with an advantage when balancing imbalanced datasets, especially for cases with low real training data. Furthermore, between the two CycleGAN models, in regards to the snowy- and rainy-label, CycleGANWD performed either better than CycleGAN or almost as good but with a smaller standard deviation.

5.2 Method

The method included collecting and preprocessing data and training a CycleGAN and classifer on varying distributions of real images. Between the different labels, the foggy label was explored more due to thinking that CycleGAN should be able to add fog to images better than rain or snow. Because of time restrains, the other labels were not evaluated to the same extent.

5.2.1 Data

The availability of high quality datasets containing images of different weather types are limited. Large high quality datasets made for similar work are often difficult to access, and the datasets which are more easily accessible often lacks in quality. The quality of the dataset and the size are therefore both large factors when choosing a dataset for the purpose of training neural networks.

The reason for using the RFS dataset was mainly due to its availability and high quality. The dataset was easily accessible through the report by Guerra et al. [4]. The dataset contained five labels of weather and the images in the dataset were of high quality, i.e. the setting in the images were often outdoor with larger views. There existed samples of close up photos of humans, which was not wanted in this work, but those images were in the minority. One disadvantage with the RFS dataset was its size. Only 1100 images were available in each label, which is little in the context of training a neural network. However, using a larger dataset would result in longer training times, especially for the CycleGAN which already took between 22-27 hours to train, depending on the methods used. A larger dataset could potentially lead to better results for both the generated images and classifier runs, however it must be weighted against the longer training times that naturally comes with it.

5.2.1.1 Training- test- and validation set

There is no clear answer as to how large the test set should be, compared to the training set. As mentioned in Section 2.4, Goodfellow et al. [10] suggested using a ratio of 8:2 between training- and test set, which ended up being used in this work. The idea of using a test set is to evaluate the performance of a classifier on data which it has not seen before and act as the ground truth for new examples the classifier might predict in the future. Since what samples appear in the training- and test set should be unbiased, the number of random samples in the test set must be large enough so that it will more likely contain images that represent the said ground truth. Because what images in different weather can contain is broad, the test set had to make up a significant portion of all images, while still keeping enough training samples to train the network.

To monitor the performance of a classifier during training, the training set was split into a training- and validation set using a ratio of 9:1 according to Goodfellow et al. suggestion [10]. Since the number of images to train on started to get small (especially since the tests required removing training samples in a target label) in comparison to what one would normally see in machine learning, it was not plausible to shift the ratio in favor of a higher validation set.

5.2.1.2 Image augmentation

To test whether synthesizing new training samples using CycleGAN is better than traditional augmentation, tests were performed using both methods and their results evaluated afterwards. The image augmentations used in this is described in Section 3.7.1, which includes manipulating brightness, hue, saturation and contrast. It is worth debating whether these techniques should be included at all, since manipulating the sky in an image could mean that the image starts to represent another weather phenomenon.

5.2.2 CNN Classifier

All tests were done using the ResNet50 model. As mentioned in 3.4 and 2.6.1, there were a number of CNN networks to chose from. The purpose of this work was to see what implications CycleGAN synthesized images may have on the training process of an image classifier. The choice was therefore not a matter of using the best performing network as a classifier in this line of field, but rather using a network that performs well enough to compare the

relative difference between the conducted tests in this work and to others work. With that in mind, it was a matter of accessibility and training time. Comparing ResNet50 to other networks such as VGG-16 that has been used in similar works [4, 40], the deciding factor was training time since both were accessible through Keras API.

At first, all classifiers had their weights initialized with ImageNet weights. The idea behind this was that the classifiers would achieve a higher and more stable score. However, after examining the results presented in Section 4.2, it became apparent that no conclusion could be drawn as all results (accuracy, precision and recall) and their standard deviation were somewhat indifferent from each other. For example, replacing images with duplicates (essentially removing images) in a label should not perform better than the full dataset. One hypothesis is that the network becomes too well adjusted to the training set very early on, and thus the network does not explore alternative solutions. So instead of initializing the classifiers with ImageNet weights, random weight initialization was used to see if the classification results would be more conclusive.

Before the actual tests were conducted, there were some tweaking of hyperparameters such as learning rate, epoch and batch size. The tweaking was done by evaluating the plots of empiric loss and generalization error for some of the methods and changing parameters accordingly. The parameters were however not tweaked to a wide extent and one could argue the results could have been better by further tweaking of the hyperparamters.

5.3 Work in wider context

As mentioned in Section 2.6.2, weather classification and recognition can be used as a step in improving video surveillance systems and autonomous driving. In such systems, the performance of the weather classifier is of most importance which, more often than not, is trained on imbalanced data. Take autonomous driving for instance, the environment where data is acquired can consist of mostly sunny weather, which makes data under other weather conditions sparse. Artificially synthesizing different weather conditions could therefore provide more data for unfamiliar cases and thus help systems achieve a better performance for edge cases.

5.4 Source criticism

The theory presented in this thesis work is based to a larger degree of *Deep learning* by Ian Goodfellow et al [10]. Ian Goodfellow is employed by Apple Inc. as of 2019 as its director of machine learning in the Special Proects Group. His book available online has been cited over 25000 times (according to Google Scholar[1]) and is therefore considered a reliable source. The theory presented in Sections 2.5.1 is almost exclusively from the official CycleGAN paper by Zhu et al. [6]. Their paper was first published in 2017 and has science then been cited over 8000 times (according to Google Scholar) and presented their work at [2] conference. Because of these merits, it is also considered a reliable source. Sources cited in related works and papers which the results are compared to are however not cited to a large extent, but considered reliable as they are published articles that has undergone peer review. This is because weather classification is somewhat new (Chapter 1) and so is research in using GAN to help balance imbalanced data, which in itself is a bit of a niche within the field of deep learning.

[1] https://scholar.google.se/
[2] https://www.thecvf.com/

6 Conclusion

This thesis work explored the possibilities of using CycleGAN to help balance imbalanced datasets for multi-class weather classification and compare its performance to other common augmentation methods. CycleGAN can be used to balance imbalanced datasets, especially when the data is scarce, however the time it takes to train the model must be weighted against the performance of the classifier. Artifacts which appeared during training of CycleGAN can be removed by using weight demodulation in the convolution layer, similar to StyleGAN2 [50]. Thus, increasing the overall visual quality of the generated images, which is a contribution of this thesis work.

6.1 Research questions

1. **How can a CycleGAN be used to enhance the performance of an existing CNN-classifier using an imbalanced training set of weather images for multi-class weather classification?**

   The performance of a ResNet50-classifier can be improved by synthesizing new training samples using CycleGAN. This is done by translating images from a well represented label to the underrepresented labels in the training set (assuming there are enough samples such that CycleGAN can learn the features of those particular labels). The sunny label was chosen as the source domain for all translations to the labels foggy, rainy and snowy. The reason being adding other weather phenomenons on top of the image should in theory be easier than predicting what the discarded information should be replaced with. For example, what exists behind a thick layer of fog?

2. **Is it possible to augment images to represent other weather phenomenons from sunny images, such as fog, rain and snow? Are there any significant differences between a CycleGANs capability of generating the different weather phenomenons?**

   It is possible to translate sunny images to foggy, rainy and snowy images, but the visual quality varied depending on what the label was in the target domain. Upon testing, it was evident that translating sunny images to foggy images produced the best results overall. CycleGAN was also capable of producing a handful of visually pleasing snowy

images while the rainy images were of poor quality. Replacing instance normalization with weight demodulation removed artifacts that could appear during training, but both methods could introduce discoloring in the images, depending on the target label. For instance, CycleGAN proved to discolor rainy images while CycleGANWD made snowy images unnaturally vibrant.

3. **How does the quality of CycleGAN generated images change when reducing the number of training samples in the target domain?**

   Reducing the number of images in the target domain lowered the visual quality of the images. However, it did not seem to have a noticeable difference in performance of the ResNet50 classifier, judging by the results of the evaluation metrics. Visually, as the number of images in the target domain decreased, artifacts appeared in the form of noise and checkerboard-looking stripes. CycleGANWD appeared to be superior to CycleGAN in this instance, as CycleGANWD kept most of the details in the images without introducing noise to the same degree.

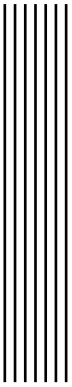
4. **For weather classification: How well does traditional augmentation techniques perform compared to CycleGAN when balancing an imbalanced dataset?**

   Comparing the results between using traditional augmentation techniques and CycleGAN synthesized images to balance a dataset, in particular the f1-score and accuracy, suggests that CycleGAN synthesized images performed better when using fewer real training samples. Between CycleGAN and CycleGANWD, there was no noticeable difference between the two methods in terms of the classifiers' performance. Initializing the ResNet50-classifier with ImageNet-weights made the classifier perform better overall in comparison to random initialized weights, but it was difficult to conclude which method performed better.

6.2 Future work

Previous works in the field of weather classification have contributed with their own datasets, one example is the RFS dataset [4] used in this work. In order to improve the model and get better results, it would be wise to explore the possibility of enlarging the dataset using traditional scraping methods to have more samples to work with.

As mentioned in the discussion, a more in depth fine tuning of the hyperparameters could lead to better performance of the classifier and perhaps yield more stable results between tests. To get a more reliable average score between tests, it would have been preferable to do further testing and take the average over a larger pool of classifier models. Furthermore, removing the synthesized images that are bad according to the discriminator could lead to an overall higher quality training set and thus a better classification results.



Bibliography

[1] Jiuxiang Gu, Zhenhua Wang, Jason Kuen, Lianyang Ma, Amir Shahroudy, Bing Shuai, Ting Liu, Xingxing Wang, Gang Wang, Jianfei Cai, et al. "Recent advances in convolutional neural networks". In: *Pattern Recognition* 77 (2018), pp. 354–377.

[2] Martin Roser and Frank Moosmann. "Classification of weather situations on single color images". In: *2008 IEEE Intelligent Vehicles Symposium*. IEEE. 2008, pp. 798–803.

[3] Xunshi Yan, Yupin Luo, and Xiaoming Zheng. "Weather recognition based on images captured by vision system in vehicle". In: *International Symposium on Neural Networks*. Springer. 2009, pp. 390–398.

[4] Jose Carlos Villarreal Guerra, Zeba Khanam, Shoaib Ehsan, Rustam Stolkin, and Klaus McDonald-Maier. "Weather Classification: A new multi-class dataset, data augmentation approach and comprehensive evaluations of Convolutional Neural Networks". In: *2018 NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*. IEEE. 2018, pp. 305–310.

[5] Sergio Gónzalez, Salvador Garcıéa, Marcelino Lázaro, Anıébal R Figueiras-Vidal, and Francisco Herrera. "Class switching according to nearest enemy distance for learning from highly imbalanced data-sets". In: *Pattern Recognition* 70 (2017), pp. 12–24.

[6] Jun-Yan Zhu, Taesung Park, Phillip Isola, and Alexei A Efros. "Unpaired image-to-image translation using cycle-consistent adversarial networks". In: *Proceedings of the IEEE international conference on computer vision*. 2017, pp. 2223–2232.

[7] S Agatonovic-Kustrin and Rosemary Beresford. "Basic concepts of artificial neural network (ANN) modeling and its application in pharmaceutical research". In: *Journal of pharmaceutical and biomedical analysis* 22.5 (2000), pp. 717–727.

[8] Terrence L Fine. *Feedforward neural network methodology*. Springer Science & Business Media, 2006.

[9] Nicholas T Carnevale and Michael L Hines. *The NEURON book*. Cambridge University Press, 2006.

[10] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. `http://www.deeplearningbook.org`. MIT Press, 2016.

[11] Martin T Hagan, Howard B Demuth, and Mark Beale. *Neural network design*. PWS Publishing Co., 1997.

[12] MY Rafiq, G Bugmann, and DJ Easterbrook. "Neural network design for engineering applications". In: *Computers & Structures* 79.17 (2001), pp. 1541–1552.

[13] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. "Deep learning". In: *nature* 521.7553 (2015), pp. 436–444.

[14] Xavier Glorot and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks". In: *Proceedings of the thirteenth international conference on artificial intelligence and statistics*. JMLR Workshop and Conference Proceedings. 2010, pp. 249–256.

[15] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification". In: *Proceedings of the IEEE international conference on computer vision*. 2015, pp. 1026–1034.

[16] Daniel Svozil, Vladimir Kvasnicka, and Jiri Pospichal. "Introduction to multi-layer feed-forward neural networks". In: *Chemometrics and intelligent laboratory systems* 39.1 (1997), pp. 43–62.

[17] Nadia Jmour, Sehla Zayen, and Afef Abdelkrim. "Convolutional neural networks for image classification". In: *2018 International Conference on Advanced Systems and Electric Technologies (IC_ASET)*. IEEE. 2018, pp. 397–402.

[18] Keiron O'Shea and Ryan Nash. "An introduction to convolutional neural networks". In: *arXiv preprint arXiv:1511.08458* (2015).

[19] Saad Albawi, Tareq Abed Mohammed, and Saad Al-Zawi. "Understanding of a convolutional neural network". In: *2017 International Conference on Engineering and Technology (ICET)*. Ieee. 2017, pp. 1–6.

[20] Vincent Dumoulin and Francesco Visin. "A guide to convolution arithmetic for deep learning". In: *arXiv preprint arXiv:1603.07285* (2016).

[21] Yann LeCun, Bernhard Boser, John S Denker, Donnie Henderson, Richard E Howard, Wayne Hubbard, and Lawrence D Jackel. "Backpropagation applied to handwritten zip code recognition". In: *Neural computation* 1.4 (1989), pp. 541–551.

[22] Hossam Faris, Ibrahim Aljarah, and Seyedali Mirjalili. "Training feedforward neural networks using multi-verse optimizer for binary classification problems". In: *Applied Intelligence* 45.2 (2016), pp. 322–332.

[23] Mohamed Aly. "Survey on multiclass classification methods". In: *Neural Netw* 19 (2005), pp. 1–9.

[24] Rajasekar Venkatesan and Meng Joo Er. "A novel progressive learning technique for multi-class classification". In: *Neurocomputing* 207 (2016), pp. 310–321.

[25] Jesse Read, Bernhard Pfahringer, Geoff Holmes, and Eibe Frank. "Classifier chains for multi-label classification". In: *Machine learning* 85.3 (2011), p. 333.

[26] Dominik Heider, Robin Senge, Weiwei Cheng, and Eyke Hüllermeier. "Multilabel classification for exploiting cross-resistance information in HIV-1 drug resistance prediction". In: *Bioinformatics* 29.16 (2013), pp. 1946–1952.

[27] Michael A Nielsen. *Neural networks and deep learning*. Vol. 25. Determination press San Francisco, CA, 2015.

[28] Douglas M Kline and Victor L Berardi. "Revisiting squared-error and cross-entropy functions for training neural network classifiers". In: *Neural Computing & Applications* 14.4 (2005), pp. 310–318.

[29] Sebastian Ruder. "An overview of gradient descent optimization algorithms". In: *arXiv preprint arXiv:1609.04747* (2016).

[30] Diederik P Kingma and Jimmy Ba. "Adam: A method for stochastic optimization". In: *arXiv preprint arXiv:1412.6980* (2014).

[31] François Chollet. *keras*. https://github.com/fchollet/keras. 2015.

[32] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems 32*. Ed. by H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett. Curran Associates, Inc., 2019, pp. 8024–8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[33] Antonia Creswell, Tom White, Vincent Dumoulin, Kai Arulkumaran, Biswa Sengupta, and Anil A Bharath. "Generative adversarial networks: An overview". In: *IEEE Signal Processing Magazine* 35.1 (2018), pp. 53–65.

[34] Alexey Dosovitskiy, Jost Tobias Springenberg, and Thomas Brox. "Learning to generate chairs with convolutional neural networks". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2015, pp. 1538–1546.

[35] Ian J Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. "Generative adversarial networks". In: *arXiv preprint arXiv:1406.2661* (2014).

[36] Mehdi Mirza and Simon Osindero. "Conditional generative adversarial nets". In: *arXiv preprint arXiv:1411.1784* (2014).

[37] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. "Going deeper with convolutions". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2015, pp. 1–9.

[38] Mohamed Elhoseiny, Sheng Huang, and Ahmed Elgammal. "Weather classification with deep convolutional neural networks". In: *2015 IEEE International Conference on Image Processing (ICIP)*. IEEE. 2015, pp. 3349–3353.

[39] Z. Zhu, L. Zhuo, P. Qu, K. Zhou, and J. Zhang. "Extreme Weather Recognition Using Convolutional Neural Networks". In: *2016 IEEE International Symposium on Multimedia (ISM)*. 2016, pp. 621–625. DOI: 10.1109/ISM.2016.0133.

[40] Di Lin, Cewu Lu, Hui Huang, and Jiaya Jia. "RSCM: Region selection and concurrency model for multi-class weather recognition". In: *IEEE Transactions on Image Processing* 26.9 (2017), pp. 4154–4167.

[41] Zhe Li, Yi Jin, Yidong Li, Zhiping Lin, and Shan Wang. "Imbalanced adversarial learning for weather image generation and classification". In: *2018 14th IEEE International Conference on Signal Processing (ICSP)*. IEEE. 2018, pp. 1093–1097.

[42] Giovanni Mariani, Florian Scheidegger, Roxana Istrate, Costas Bekas, and Cristiano Malossi. "Bagan: Data augmentation with balancing gan". In: *arXiv preprint arXiv:1803.09655* (2018).

[43] Youssef A Mejjati, Christian Richardt, James Tompkin, Darren Cosker, and Kwang In Kim. "Unsupervised attention-guided image to image translation". In: *arXiv preprint arXiv:1806.02311* (2018).

[44] Cewu Lu, Di Lin, Jiaya Jia, and Chi-Keung Tang. "Two-class weather classification". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2014, pp. 3718–3725.

[45] Yin Wang and YingXiang Li. "Research on Multi-class Weather Classification Algorithm Based on Multi-model Fusion". In: *2020 IEEE 4th Information Technology, Networking, Electronic and Automation Control Conference (ITNEC)*. Vol. 1. IEEE. 2020, pp. 2251–2255.

[46] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. "Deep residual learning for image recognition". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778.

[47] Yoshua Bengio, Patrice Simard, and Paolo Frasconi. "Learning long-term dependencies with gradient descent is difficult". In: *IEEE transactions on neural networks* 5.2 (1994), pp. 157–166.

[48] Yann A LeCun, Léon Bottou, Genevieve B Orr, and Klaus-Robert Müller. "Efficient backprop". In: *Neural networks: Tricks of the trade*. Springer, 2012, pp. 9–48.

[49] Kaiming He and Jian Sun. "Convolutional neural networks at constrained time cost". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2015, pp. 5353–5360.

[50] Tero Karras, Samuli Laine, Miika Aittala, Janne Hellsten, Jaakko Lehtinen, and Timo Aila. "Analyzing and improving the image quality of stylegan". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2020, pp. 8110–8119.

[51] Dmitry Ulyanov, Andrea Vedaldi, and Victor Lempitsky. "Instance normalization: The missing ingredient for fast stylization". In: *arXiv preprint arXiv:1607.08022* (2016).

[52] Phillip Isola, Jun-Yan Zhu, Tinghui Zhou, and Alexei A Efros. "Image-to-image translation with conditional adversarial networks". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2017, pp. 1125–1134.

[53] 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". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.