Improving image sensor performance by developing on-chip machine learning algorithms

Anders Grinden Vestengen

Thesis submitted for the degree of Master in Electronics, Informatics and Technology
(Microelectronics and Sensor Technology)
60 credits

University of Oslo

Oslo, Norway. Autumn 2023

Supervised by:

Johannes Sølhusvik,
Associate professor

Hemin Qadir,
Research Scientist

Contents

List of Figures

List of Tables

Abstract

Machine learning models are outperforming humans on an increasing number of tasks. However, development and implementation in hardware remains a smaller but pressing issue. In this project we present a machine learning model based upon Generative Adversarial Networks and a base encoder-decoder architecture similar to U-net, aimed at tackling larger defects in image sensors. We show that our model can outperform a conventional median filter on larger defects. Improving PSNR by 10dB, and outperforming the filter by 38% when scored on defective clusters of size $[7 \times 7 - 12 \times 12]$, using the SSIM metric. Furthermore, we present a custom objective function to help guide machine learning models when tasked with solving smaller problems within larger input spaces. We call this the latent loss objective and demonstrate comparative results with non-latent implementations on a large dataset. The model is also developed with eventual hardware implementation in mind, and we discuss the efficacy and practicality of this.

Dedication

I dedicate this work to my late father and grandparents. People of remarkable strength, warmth, and ingenuity who did not get to see this work to its completion.

Acknowledgements

I would like to thank and acknowledge my thesis advisors Johannes Sølhusvik and Hemin Qadir who helped guide me through the project. I thank my girlfriend Yini, without whom this thesis might not exist. Finally, I would like to thank friends and family.

Chapter 1

Introduction

Machine learning as a subject has exploded over the least decade. Entering mainstream fascination with the public release of the ChatGPT large language model. As a student of microelectronics I was intrigued by the prospect of using machine learning models on previously impractical problems within low-level hardware design. One of the ideas I came up with alongside my eventual thesis advisor Johannes Sølhusvik was on defect correction in image sensors. Most image sensor produced today usually have several defective pixels, which is not a problem as algorithms exist that blend information from neighbouring pixels creating a fake datapoint of what should have been. This is done as a low-level software implementation on the sensor. The problem is that when these defects reach a certain size f.ex $[1 \times 3]$ pixels in a row, those handcrafted algorithms stop working well. We wondered if a small and efficient machine learning model could be used to better solve this issue. It should be noted that we also planned on implementing the model on an FPGA. This turned out to be too ambitious, and the hardware implementation has been relegated to a feasibility study instead.

1.1 Objectives

- Train or develop a machine learning model to do defect correction in image sensors.
- Create real or synthetic training images to train the network.
- Implement the model on hardware to study the efficacy of the idea.

1.2 Scope of Work

In this project I have developed generative adversarial machine learning models with new or adapted objective functions based on state-of-the-art research. I have used tools like Pytorch, Numpy, Scikit-learn, Pillow, and matplotlib to aid in model training and development. I have also created a custom framework in Python to facilitate all training, development and prototyping work.

A public epository of the source code has been uploaded to the github instance hosted by UiO here.

The framework currently

- Implements custom functions to update the models during both training and validation.
- Implements custom objective functions like the Latent inpainting loss.
- Runs inference, aggregates PSNR and SSIM scores, and stores inference images
- Creates on the fly, custom synthetic image defects aimed at mimicking real image sensor defective clusters, and can be interfaced to most image training datasets. The defects can vary in size, shape, and type. There can also be a random variable number of defects present

in the image, which is determined by user parameters. The defects themselves will also randomly vary in amplitude, to better mimic non-linear pixel defects.

- Creates, stores, and visualizes model analytics and information for every trained model.
- Implements several loss functions and model types from previous research efforts.
- Implements a custom script to blend and visualize graphs and data of different models.

Chapter 2

Scientific Background

2.1 Introduction to Neural Networks

As the basis for most modern machine learning methods, and certainly in the context of this work, is the concept of the neural network. The objective of a neural network is to take some input and produce an output that fits the desired criteria. This can for example be sentences in one language producing a translation into a different language, or an old black and white picture from before color photography comes out of the network as fully colorized. Networks can perform many tasks and can be thought of in one sense as a large differentiable function that is 'tuned' through training to produce a set of outputs. The types of network investigated in this paper are Supervised networks, which mean they train only on pre-constructed samples which have a clear 'truth' that the network can aim towards. When training is complete the network is used for inference, which means it works just like when training, but now the neurons are static and there is no more learning happening in the background.

2.2 The Neuron

The first notion of a neuron was introduced as a mathematical model in 1943 by Walter Pitts and Warren McCulloch[26]. The idea then was to be able to study a system closely related to neurons in the human brain. The model consists of one or several weighted inputs to a summing function, and finally a threshold or activation function which decides if the neuron should fire its output based on the sum of inputs. This is meant to mimic a real neuron, which collects charges on its input from other connected neurons and at a certain threshold would fire its own charge based upon sufficient input charges.

2.3 The Multilayer Perceptron

The modern neural network however, didn't start to take shape until 1958 with the creation of the Perceptron[26]. In this model several of the Pitts McCulloch neurons are strung together in what's called a layer. The input connects to every neuron and the output in turn depends on every neuron. When you combine several Perceptron layers together in sequence you get a neural network 2.1, a collection of layers of neurons. The neural network has an input layer and an output layer, there are also several in-between layers called hidden layers. Between every layer are connections. Networks are called 'densely connected' when every neuron of the previous layer is connected to every neuron in the following layer. The influence of any neuron on the next is determined by its weight, a 'learnable' parameter which means it will get updated through the training process.

Attached to every neuron is usually a bias which is there to guarantee some output from the neuron at the beginning of training. Every neuron and associated weight gets a random value when the network is first brought about before training. There are many ways to initialize these values, some of which will be discussed later in this chapter, but the goal is to ensure that all neurons fire something and so at the end of training the hopefully contribute to the network output in some way. It can happen that a neuron is silenced through what's called the vanishing gradient problem, a phenomenon where the training process updates a neurons value to be near-zero and at that point the neuron might as well not exist at all.

Another concept to most fundamental neural networks is the bias term. This is a small, fixed (it can also be set as a learnable parameter) addition to the layer term 2.2 which is there to ensure the output of the neuron is not zero at the start of training. This is done to counteract the phenomenon of vanishing gradients and dead neurons early at the early stages of training if initialization of the neuron is set close to zero.

2.4 Forward propagation

$$\mathbf{i}_k^l, \quad \text{Where k denotes neuron and l denotes layers} \tag{2.1}$$

$$\mathbf{I} = \phi(\mathbf{i}), \quad \mathbf{i} = \mathbf{W}^{(1)}\mathbf{x} + \mathbf{B}^{(1)} \tag{2.2}$$

$$\mathbf{H} = \phi(\mathbf{h}), \quad \mathbf{h} = \mathbf{W}^{(l-1)}\mathbf{h}^{(l-1)} + \mathbf{B}^{(l-1)} \tag{2.3}$$

$$\mathbf{O} = \phi(\mathbf{o}), \quad \mathbf{o} = \mathbf{W}^{(l-1)}\mathbf{h}^{(l-1)} + \mathbf{B}^{(l-1)} \tag{2.4}$$

$$\mathbf{L} = (\mathbf{O} - \mathbf{y})^2 \tag{2.5}$$

During forward propagation the input signal travels from the input $\mathbf{x}$ as seen in 2.1, through the layers and activation functions before finally exiting the network. There are about as many ways to arrange the output as there are different problems for the network to solve, but for any network configured for supervised learning there will be an output $\mathbf{y}$' from the network, which is intended to be as close to the 'ground truth' $\mathbf{y}$ as possible. The difference between $\mathbf{y}$' and $\mathbf{y}$ is what's known as the 'cost' or 'loss' and is the objective used to measure how well the network is doing.

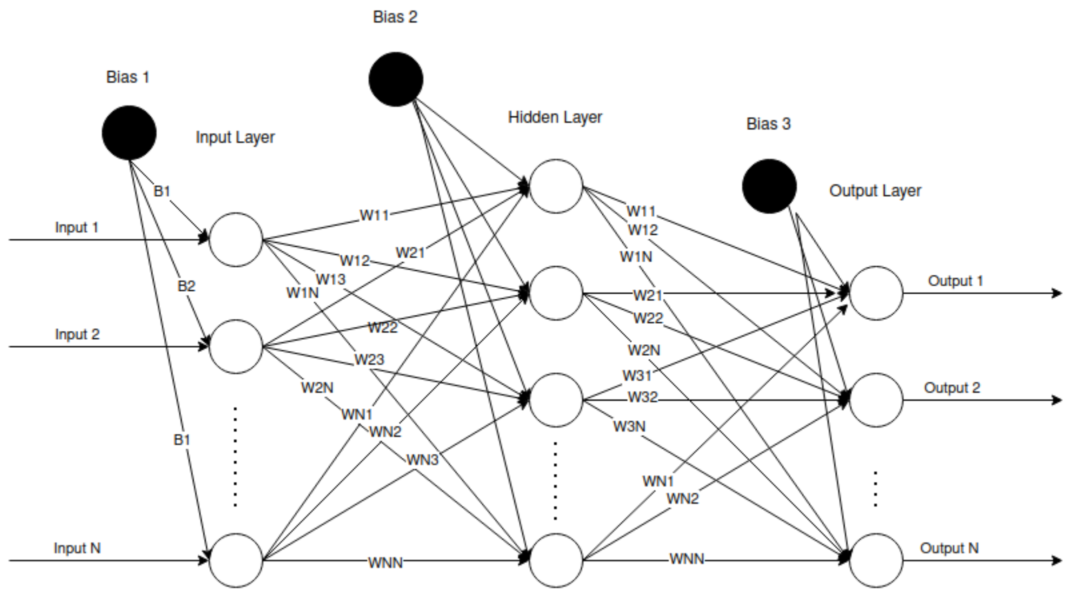


Figure 2.1: Illustration of the Multi Layer Perceptron with the intermediate components shown. Activation functions ϕ_1 to ϕ_3 are omitted for clarity.

2.5 Backpropagation

$$C = \frac{1}{2n} \sum_x ||y(x) - a^L(x)||^2 \tag{2.6}$$

$$\delta_j^l = \frac{\partial C}{\partial \mathbf{h}_j^l} \tag{2.7}$$

$$\delta^L = \nabla_{\mathbf{H}} C \odot \phi'(\mathbf{h}_j^l) \tag{2.8}$$

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \phi'(\mathbf{h}^l) \tag{2.9}$$

$$\frac{\partial C}{\partial b_j^l} = \delta_j^l \tag{2.10}$$

$$\frac{\partial C}{\partial w_{jk}^l} = \mathbf{H}_k^{l-1} \delta_j^l \tag{2.11}$$

Backpropagation is the reverse of forward propagation, and after the network produces and output from its training or inference input we use backpropagation to figure out the gradients. Gradients in this case represent how much each individual neuron in the network influenced the final output. The following breakdown and notation is taken from [28, Ch. 2]. Consider a quadratic loss function 2.6, to find the error in a given layer, as in the difference between the truth values in the training data and the network output, can be represented as 2.7, which is the partial derivative of the Loss function with respect to the given activation of a layer. Representing this in matrix notation 2.8 we see the general error is given by the gradient of C with respect to the hidden layers activation multiplied with the partial derivative of the activation function with respect to that particular layer. From there functions for the error on individual biases 2.10 and weights 2.11 are given. With these its possible to peel back the network activations of the previous training sample and see every neurons influence and error compared to the target.

2.6 Gradient Optimization

Gradient optimization is an algorithm that takes the gradients procured by the backwards pass along with the loss from the objective function (or loss function) and decides by how much the weights and parameters of the network should change before the next iteration of training. Optimization algorithms are usually concerned with minimizing the loss, but this can be changed with a sign value [45, Sec. 12.3.1]. Adams is the a modern optimizer generally regarded as one of the most robust and effective available [45, p. 12.10] . Adams is a popular algorithm combining several of the best parts of previous efforts into one unified update rule[45, Sec. 12.10].

$$\mathbf{v}_t \leftarrow \beta_1 \mathbf{v}_{t-1} + (1 - \beta_1)\mathbf{g}_t \tag{2.12}$$
$$\mathbf{s}_t \leftarrow \beta_2 \mathbf{s}_{t-1} + (1 - \beta_2)\mathbf{g}_t^2 \tag{2.13}$$

$$\hat{\mathbf{v}}_t = \frac{\mathbf{v}_t}{1 - \beta_1^t}, \quad \hat{\mathbf{s}}_t = \frac{\mathbf{s}_t}{1 - \beta_2^t} \tag{2.14}$$

$$\mathbf{g}_t' = \frac{\eta \hat{\mathbf{v}}_t}{\sqrt{\hat{\mathbf{s}}_t} + \epsilon} \tag{2.15}$$

$$\mathbf{x}_t \leftarrow \mathbf{x}_{t-1} - \mathbf{g}_t' \tag{2.16}$$

The procedure starts with updating the two state vectors. Adam uses exponential weighted moving averages to obtain an estimate of both momentums of the gradient $(\mathbf{s}_t, \mathbf{v}_t)$ 2.12. (β_1, β_2) are parameters set before training begins. Next it normalizes these vectors 2.14 before updating the gradients 2.15. The standard value for ϵ is $1 \times 10^{-}8$ in the Pytorch Framework [31]. The network weights are then updated according to 2.16

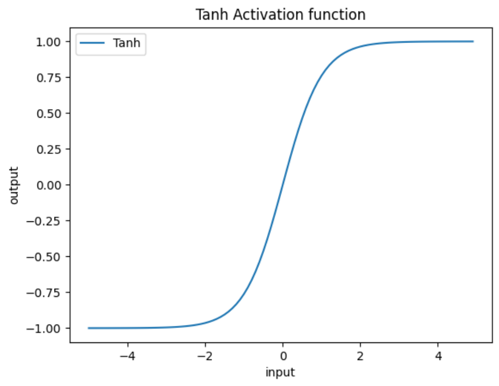
2.7 Activation functions

The representation of a layer without an activation function is given by **h** 2.3. **h** is an affine function, meaning it can only represent other affine functions. This severely limits the potential of the network to approach more complex representations, and is where adding non-linear activation functions come in. Activation functions gives the network layer more complexity and stacking multiple layers, each with their own non-linear activations, yields even more expressive models [45, Sec 5.1.1.3].

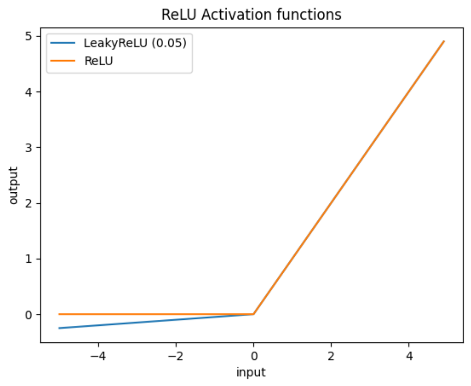
$$\tanh(\mathbf{x}) = \frac{\exp(\mathbf{x}) - \exp(-\mathbf{x})}{\exp(\mathbf{x}) + \exp(\text{-}\mathbf{x})} \tag{2.17}$$

$$\text{ReLU}(\mathbf{x}^{+}) = \max(0, x) \tag{2.18}$$

$$\text{LeakyReLU}(\mathbf{x}^{+}) = \max(0, x) + \lambda * \min(0, x) \tag{2.19}$$



(a) Illustration of the Tanh activation function



(b) Illustration of the ReLU activation functions

Three different activation functions were used for this project, activation functions are usually applied element-wise on the input tensor. The first is the Tanh function 2.17 which in an implementation of the hyperbolic tangent function. Next is the ReLU or rectified linear unit function is simply 2.18, zeroing out any negative values that pass through the layer. Finally, LeakyReLU 2.19 is a deviation allowing some negative gradient through dependent on a parameter λ which is defined before training.

2.8 Initialization

Initialization is the process of giving all parameters in the network a value. Initially it might seem somewhat unnecessary to give these values much thought since they will immediately begin changing once training starts, and that's true. However, these initial values have a huge impact on the early parts of training a network and therefore on convergence. To understand this you simply have to look back at the Backpropagation 2.5 chapter. Any update to a neuron is derived from its original value and so very small initialization will make even smaller gradient updates during training. This can cause the network to get stuck early in a shallow minimum. On the opposite end of the spectrum large values for neurons might make the optimization take larger steps and overshoot a better solution. So the way in which the network is initialized matters a great deal. This section will cover a few techniques that have been used during development on the project

A big problem in machine learning these days are maintaining stable updates for ever-increasing network sizes. Recall in the Backpropagation2.5 section that the gradient of a neuron at the end of the network might only depend on a few parameters, but the neurons in the first few layers depends on a large chain of matrices and vectors of all the layers after. For larger networks unstable gradient updates are a problem which can cause early stopping and poor convergence, GAN style networks are especially prone to this and when they failed to converge its usually referred to as mode collapse. Careful initialization of the network parameters are one of the tools to mitigate this behavior and start the training in a better 'basin' of attraction to the optimal procedure according to the original Xavier initialization paper[11, p. 1 par. 3]

$$\mathcal{N}(\mu, \sigma^2) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}(\frac{x-\mu}{\sigma})^2} \tag{2.20}$$

The first initialization is the normal distribution. It samples values from the normal distribution function 2.20, usually with $mean = 0.0$ and $std = 1.0$.

$$\mathcal{U}(-\sqrt{\frac{6}{n_{in} + n_{out}}}, \sqrt{\frac{6}{n_{in} + n_{out}}}) \tag{2.21}$$

$$\sigma^2 = \frac{2}{n_{in} + n_{out}} \tag{2.22}$$

The second initialization scheme is commonly called Xavier initialization and draws values from a uniform distribution $\mathcal{U}$ 2.21 with mean of zero and variance of 2.22, which both depend on the number of input and output neurons in the given layer (n_{in}, n_{out}). Xavier Initialization is meant to give the network a more stable launchpad from which to start training by maintaining the variance of activations and back-propagated gradients throughout the network [11].

$$\mathcal{U}(-\sqrt{\frac{2}{n_{in}}}, \sqrt{\frac{2}{n_{in}}}) \tag{2.23}$$

The last initialization strategy is a variation of Xavier called 'Kaiming' or sometimes just 'He' initialization. It was created because Xavier works poorly with popular and more recent activation functions like ReLU because it assumes linear activations [16]. Kaiming initialization instead draws from a uniform distribution $\mathcal{U}$ and then normalizes the values with 2.23. This yields better results for networks employing ReLU or LeakyReLU functions [16, p. 5].

2.9 Autograd

Autograd is an automatic differentiation engine that comes with the PyTorch library[30]. It automates differentiation of gradients in the forward and backwards passes, which abstracts this consideration away in practice when working with neural networks. The Autograd engine works by using a DAG or directed acyclic graph of function objects to track every tensor and its operation during the forward pass of the program. It then takes the information in the DAG and computes gradients along the reverse order of the graph, similarly to backpropagation but in a dynamic way and on a much larger scale.

2.10 Convolutional Layers

Convolutional networks or CNN's for short are neural networks characterized by their reliance on convolutional layers to meet their designed objective. A convolutional layer is in essence a convolutional kernel that have changeable values which can be trained just like the neurons in a perceptron. By strict definition the convolutional layer is actually a cross-correlation operator, but since the values of the kernel are learnable the convention is to discard this step[45, Ch. 7.1.3].

It's also worth talking about what happens when the input has more than one dimension. For projects like this one concerning images there are three color dimensions (alt. channels) in every input image red, blue, and green. Together they constitute the full image, but this also means that information about the image is spread among the color channels and to extract this there needs to be a kernel for every channel of the input. It's also worth pointing out that the kernels of a given layer are referred to interchangeably to as feature maps because the kernels are learned mappings of their input[45, Ch. 7.1.4].

Convolutional layers also have several advantages over their counterparts in MLP's as the layer is now spatially invariant[45, Ch. 7.1.1] since it doesn't matter how big or small the input is (as long as it's at least as big as the kernel). An MLP mandates that the input always be the same size as its input layer. For example in this project the standard training size of an image is $3x128x128$. An MLP would require a one dimensional input array meaning the image would now have flattened to the size 4.9×10^4 and connecting this to just one hidden layer of the same size would mean the weight matrix connecting the two would have to be of size $10^4 \times 10^4 = 10^8$. In Pytorch this single weight matrix would be 400 MB large, and that's just one weight matrix in a two-layer perceptron, not even considering any output layers. By contrast the weight-matrix of a 2-D convolutional layer with 3 input and output channels and a square kernel of size 4 would need 0.000576 MB of space[32]. This should illuminate the immense impracticality that strictly linear networks face. This is why convolutional networks can expand to much larger complexity with respect to their layers and architectures without becoming relatively immense compared to the linear example above. In fact, the complete network developed for this project is only 24 MB when saved to file.

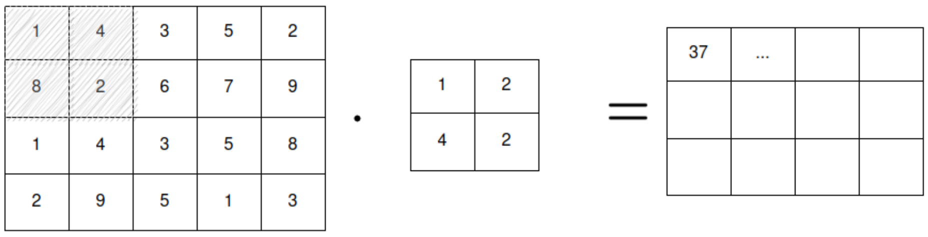


Figure 2.3: Figure showing the cross correlation operation

$$[\mathbf{H}]_{i,j,d} = \sum_{a=-\Delta}^{\Delta} \sum_{b=-\Delta}^{\Delta} \sum_{c} [\mathbf{V}]_{a,b,c,d} [\mathbf{X}]_{i+a,j+b,c} \tag{2.24}$$

Figure 2.3 show's the cross correlation operation intuitively as a sliding windows across an input. This aims to illustrate the operation captured in 2.24 where the output features **H** from a single layer is the product of the layer kernels represented by **V** sliding over the input **X**. It's also time to discuss how input and outputs work in these operations. You can image that for a single input dimensions the kernel operation creates a single output dimension. If our input has several dimensions like an image then we need a corresponding amount of kernels to complete the cross-correlation operation[45, Ch.7.4.1]. The convention for cross-correlation is to then sum all the outputs together to form a single output dimension. This may seem weird at first glance, after all if were working on color images in a neural network why shouldn't we just let there be three output channels, doesn't this just increase computation? And yes, but there are two key points which make it a good option. The first is the fact that information is spread among the channels and adding them together condenses this information down into the one output, so information from the whole image propagates through the network instead of channels descending through their own kernels. The second reason is for flexibility. If we want a different number of output channels than inputs we don't necessarily want those outputs to be different. If we export the three outputs from the example above, but also decide we want four or six channels what should those extra channels now be made up of? It would be an arbitrary selection of different channels, and maybe there are applications where this makes sense, but in general terms this is undesired. Instead, we create a kernel tensor of the input size $C \times H \times W$ for every output channel. This has the benefit of solving the problem previously mentioned, and is what the subscript d in $[\mathbf{H}]_{i,j,d}$ represents. You can now start to look at the 2.24 equation more intuitively. Every layer output $[\mathbf{H}]$ is made up of the sum of the difference between $[\mathbf{V}]$ and $[\mathbf{X}]$. First over the output dimensions d which sums over the spacial dimensions (i, j) and input channels c. The subscript (a, b) represent the kernel size Δ. Usually the kernel is square and so a and b are the same size.

$$\mathbf{W}_{out} = [\frac{\mathbf{W}_{in} + 2 \times padding - dilation \times (kernelsize - 1) - 1}{stride} + 1] \tag{2.25}$$

Equation 2.25 shows the effect padding, kernel size, dilation and stride have on their respective spacial dimension $\mathbf{W}$.

Finally, it's also worth talking about why there would be a need for multiple output channels, what is the virtue of expanding the number of dimensions in a convolutional network? The answer is that one can trade spacial resolution for feature maps[45, Ch. 7.4.1]. Expanding the number of feature maps expands the dimensions and should allow the model to learn more complexity. It's also worth noting that the 'why' of how machine learning algorithms learn is not known. Current Machine learning algorithms are essentially black boxes[12, sect. 1] and while you can train and then test a traditional algorithm on a set of data, you only get an output and not any idea of why that output is. Still it is widely accepted that increasing feature maps allow convolutional networks to reason with multiple features at a time[45, Ch. 7.4.4]. A good example of the power from feature expansion is the original U-Net[35] which relied heavily on increasing feature channels at the cost of spacial resolution in order to learn complex representations while only having access to a smaller set of training data. The network delivered impressive results[35, Table 2] which gives merit to the idea that expanding feature channels results in the model being able to learn more complex representations.

2.11 Generative Adversarial Networks

Generative adversarial networks or GAN for short were introduced in a research paper in 2014 [13]. It's a neural network characterized by the fact that it consists of two networks working against each other in a min/max game. The idea starts with your desired target. Say you want to generate images of dogs, and so you'll start out with a training set of real dogs, this will be you training distribution (ρ_{data}). The first network which is usually referred to as the Generator takes an input of random noise and outputs a tensor of the same dimensions as your images. Finally, there is the adversarial part of the network called the Discriminator. Its job is to take the inputs from the training images and the generator outputs and label them as either real or fake. Over time the output distribution from the Generator (ρ_g) should move towards the distribution of the training images. Theoretically there is a final state for the network if there is sufficient parametric capacity [13, Sec. 3]. In this case ($\rho_g = \rho_{data}$) there would be an equilibrium as the Discriminator would no longer be able to distinguish the two distributions and classifies the images at random i.e. $D(x) = \frac{1}{2}$.

$$\min_G \max_D V(D, G) = \mathbb{E}_{\mathbf{x} \sim p_{data}(\mathbf{x})}[logD(\mathbf{x})] + \mathbb{E}_{\mathbf{z} \sim p_z(\mathbf{z})}[log(1 - D(G(\mathbf{z})))] \tag{2.26}$$

$$\min_G \max_{D \in \mathcal{D}} V(D, G) = \mathbb{E}_{\mathbf{x} \sim \mathbb{P}_r}[D(\mathbf{x})] - \mathbb{E}_{\tilde{\mathbf{x}} \sim \mathbb{P}_g}[D(\tilde{\mathbf{x}})] \tag{2.27}$$

$$\mathcal{L}_{cGAN}(G, D) = \mathbb{E}_{(\mathbf{x},\ \mathbf{y})}[logD(\mathbf{x},\mathbf{y})] + \mathbb{E}_{(\mathbf{x},\ \mathbf{z})}[log(1 - D(G(\mathbf{x},\mathbf{z})))] \tag{2.28}$$

$$\mathcal{L}_{L1}(G) = \mathbb{E}_{\mathbf{x},\ \mathbf{y},\ \mathbf{z}}[||\mathbf{y} - G((\mathbf{x},\mathbf{z}))||_1] \tag{2.29}$$

$$G* = \arg\min_G \max_D \mathcal{L}_{cGAN}(G, D) + \lambda \mathcal{L}_{L1}(G) \tag{2.30}$$

GAN networks are hard to train [13][3][14]. One of the big problems with the original formulation of the GAN objective function2.26 is that it can be prone to something called mode collapse[13, Sec. 6], this is where the Generator finds a very small subset of generated samples which work well to fool the Discriminator and the two tend to get stuck wherein the Generator presents the same few samples and the Discriminator accepts these as the optimal values. Another problem comes when that the Discriminator manages to distinguish samples between the distributions too easily and rejects fake samples with a high accuracy such that the Generator has little to go on in terms of gradient updates. The Discriminator 'figures out' the Generator too fast before it has a chance to evolve better samples and so the training collapses.

To mitigate this issue a different objective function was proposed which is much better behaved called Wasserstein GAN or WGAN[3]. The WGAN objective function 2.27 is constructed using the Kantorovich-Rubenstein duality, where $\mathcal{D}$ is the set of 1-Lipschitz functions[14, Sec. 2.2], $\mathbb{P}_r$ and $\mathbb{P}_g$ are the training and generator distributions respectively. The WGAN function is based on what's called EM (earth-mover) or alternatively Wasserstein-1 distance. Intuitively this distance is the optimal transport plan for 'moving' $\mathbb{P}_g$ closer to $\mathbb{P}_r$ [3, p. 4]. It should also be noted that the Discriminator output is different for WGAN objectives because it does not label true or fake $(1, 0)$, instead the output of the network is now unconstrained and is therefore called a critic. This is done because the original Discriminator will quickly learn to distinguish either real or fake and when it does the binary information becomes less useful, thus unconstrained outputs lead to better gradient information as the samples are given a score rather than a label [3, p. 8].

Another GAN variant is called cGAN which learns a mapping from observed image $\mathbf{x}$ and random noise vector $\mathbf{z}$ to desired output $\mathbf{y}$, $G : \{\mathbf{x},\mathbf{z}\} \to \mathbf{y}$ [20, p. 3]. Although, in practice $\mathbf{z}$ varies in representation, from outright being a sampled noise vector to sometimes just an abstraction based upon layer regularization like Dropout [20, Sec. 3.1]. The objective function for cGAN 2.28 is similar to the original GAN function. Another addition is that cGAN's benefit from adding a second loss function like L1 [20, Sec. 3.1] to the Generator loss 2.29. It doesn't change anything in regard to the Discriminator. The final objective function is then summarized as 2.30.

2.12 Skip Connections

$$\mathcal{F}(\mathbf{x}) := \mathcal{H}(\mathbf{x}) - \mathbf{x},\ \ \mathcal{F}(\mathbf{x}) + \mathbf{x} \tag{2.31}$$

Residual (or more commonly referred to as just 'skip') connections are a recasting of a network layer (or layers) mapping i.e. If the network can be seen as a large differential function then there is an ideal layer 'mapping' of parameters which should lead to it [45, Sec. 8.6.2]. Skip connections recast this problem into a residual mapping, so instead of learning a mapping $\mathcal{H}$, we seek to learn the residual 2.31, where $\mathbf{x}$ is the input vector to the layer. This was originally developed to help mitigate the 'degradation' problem, which is when a network becomes worse as more layers are added and the cause is not overfitting. Recasting the formulation is not strictly better in theory, but they do perform better in practice [15, Sec. 3]. Residual connections in short allows the network to deepen without great penalty which is crucial for more complex architectures.

2.13 Loss Functions

Outside of the GAN objective functions there are two more fundamental loss functions used in the project. The first is L1-loss 2.32 [33] which was also mentioned for cGAN networks 2.29. The second is Mean squared error loss or MSE for short 2.33 [34]. They are used in conjunction with the GAN loss to enforce pixel ground-truth.

$$l(\mathbf{x},\ \mathbf{y}) = L = \{l_1, ..., l_N\}^\top, l_n = |x_n - y_n| \tag{2.32}$$

$$l(\mathbf{x}, \mathbf{y}) = L = \{l_1, ..., l_N\}^\top, l_n = (x_n - y_n)^2 \tag{2.33}$$

2.14 Regularization techniques

$$\sigma(A) := \max_{\mathbf{h}:\mathbf{h}\neq 0} \frac{||A\mathbf{h}||_2}{||\mathbf{h}||_2} = \max_{||\mathbf{h}||_2 \leq 1} ||A\mathbf{h}||_2 \tag{2.34}$$

$$\bar{W}_{SN}(W) := \frac{W}{\sigma(W)} \tag{2.35}$$

$$\sigma(\bar{W}_{SN}(W)) = 1 \tag{2.36}$$

$$||f_1 \odot f_2||_{Lip} \leq ||f_1||_{Lip} \cdot ||f_2||_{Lip} \tag{2.37}$$

Regularization techniques are 'helper' functions or strategies employed at training time to mitigate 'overfitting' which is when a network learns its training set too well and starts performing worse when tasked to do the same thing new data. In essence the network now fails to do what its been trained to do by performing too well during training. The method used for this project is Spectral Normalization. Spectral Normalization is a weight normalization method designed to stabilize training of GAN networks. It works by using the spectral norm of a given layer 2.34, and then utilizing 2.35 and 2.36 given normalization at every layer to impose an upper bound of $||f||_{Lip}$ by 1 given the inequality 2.37, where f_n are layers of network f [27, Sec. 2.1]. This helps ensure Lipschitz continuity which crucially effect the performance of GAN networks [27, Sec. 2.0].

2.15 Attention mechanisms

Attention comes in many forms for machine learning algorithms, but in the context of this paper we narrow the definition closer to the image and inpainting papers used for this project. A large portion of current attention mechanisms borrow from the proposed transformer architecture by Google [9]. This form of attention uses some kind of feature extractor like a simple linear neural net or 1×1 convolutional layer to extract features onto new mappings generally referred to as key and value layers [45, Ch. 11]. Attention can be explained as a way for the network to figure out how much importance is given to any input feature when constructing an output feature. In the book "Dive into deep learning" [45, Ch. 11], the attention mechanism is likened to a database. Imagine you have a number of distinct values in this database, each with a corresponding key to identify it. If you search for a given key you get the corresponding value and so on. Now imagine that you perform an incomplete search with a partial key, and correspondingly the database gives you a 'closest match'. This is how attention layers can be understood to work, and the attention is how strict the search for a corresponding value, given an input. In this way attention layers can be trained to either focus or disregard certain input features when constructing outputs based on specific loss functions. The network can now learn long-term dependencies that convolutional networks struggle with as they are bound by high-resolution but close neighboring focus given the kernel-size, or long range but low resolution given deeper convolutional layers. Attention can break this up and refocus features deemed important to the objective. Some examples are the SAGAN [46] model which uses self-attention mechanisms to pick up long range dependencies for image generation. In their own words previous convolutional networks might generate dogs with realistically textured fur, but the same animal might not have clearly defined paws [46, Sec. 1]. DAM-GAN [8] created attention blocks that aimed to detect and correct 'fake' looking pixels from a preceding 'coarse' inpainting network to better generate high-fidelity features.

Chapter 3

Methodology

3.1 Project Development

The thesis goal was to investigate machine learning applications on hardware, and specifically defect correction in image sensors. Initially the project used an implementation made by my supervisor Hemin Qadir, based upon the pix2pix [20] inpainting model created by Nvidia research. After getting the model and training framework setup it was rebuilt to understand how the model and training schemes worked, and an initial version of the current dataset class was created to generate synthetic defects. However, while inpainting models like pi2pix had published impressive results, it converged poorly on the defect regions 3.4. Therefore, several new prototypes were tested before developing a new architecture with the additional goal of being easy to deploy on hardware. Several new objective functions have also been developed to help train the network on this specific problem. Below is a graph showing the evolution of the project.

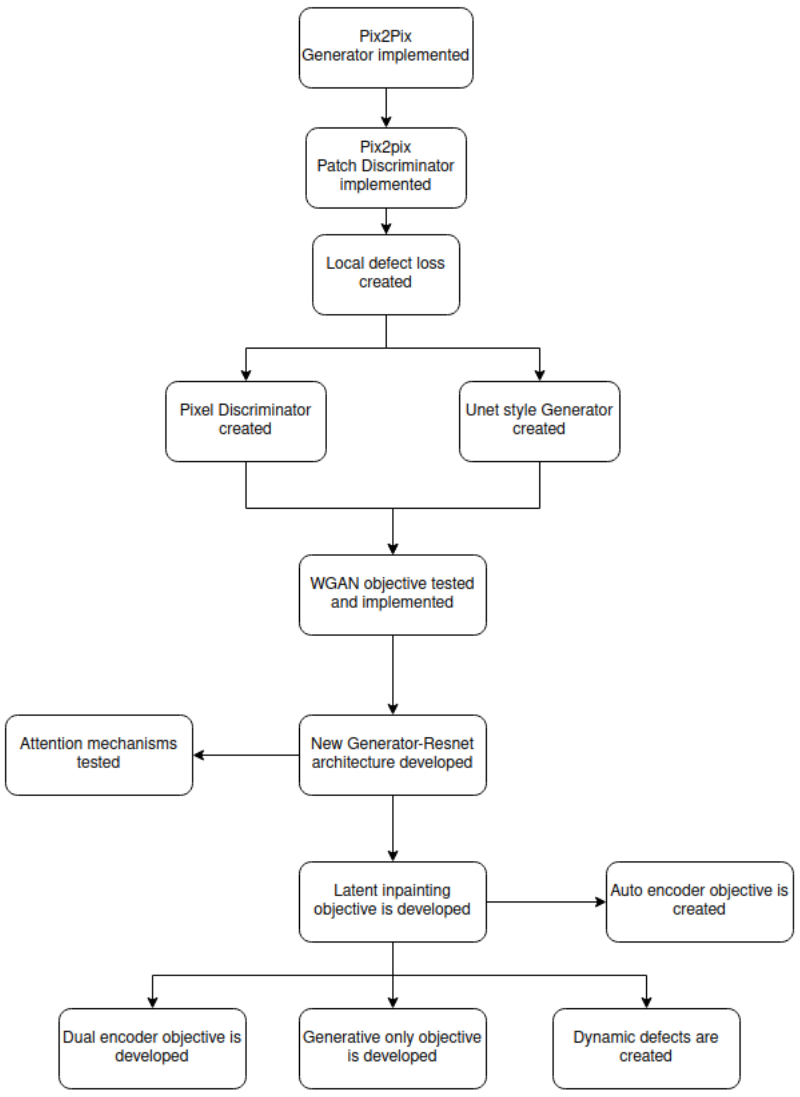

Pix2Pix Generator implemented
Pix2pix Patch Discriminator implemented
Local defect loss created
Pixel Discriminator created
Unet style Generator created
WGAN objective tested and implemented
Attention mechanisms tested
New Generator-Resnet architecture developed
Latent inpainting objective is developed
Auto encoder objective is created
Dual encoder objective is developed
Generative only objective is developed
Dynamic defects are created

Figure 3.1: Flowchart detailing the project evolution

3.2 Generator architecture type

The generator architecture itself is based upon the U-net style of autoencoder architecture, first proposed in [35]. It is a fully convolutional network which means there are no linear layer types to do prediction. It also has the key advantage of converging well on a small dataset [35]. However, It should be mention that this is not an implementation the U-net architecture itself, but takes inspiration from the overall Unet structure, specifically the up- and down-sample configuration. This generative network is also much smaller. The new architecture does not use maxpool, upsample, or consecutive convolutional layers at the same spatial size. This is because it was found through testing that it is not necessary. Furthermore, the model works without these non-learning layers like maxpool and upsample for two reasons. The first is that maxpool can act like a regularizer because it removes some of the information in the feature maps in order to scale down the spatial size. Instead the model uses spectral normalization [27] and dropout [18] to enforce regularization. For the transposed convolutions were chosen for the upsample layers and they can also increase spatial size, but have the advantage that they are learnable parameters. This ties in to the final overall architectural choice which is to change the skip-connections from concatenations of the features across the architecture, and instead use residual [15] ones. In their paper they argue that the residual connections should allow the network to 'zero' activations that don't help the overall training objective. This to be a very interesting idea as it should mean that for a hardware implementation of the network, the residual connections should allow it to be more aggressively pruned 6.2.1, because the network has hopefully learned to reduce neurons that don't contribute. The other reason for using residual connections is for their stated performance. Skip connections are also popular with other autoencoder style inpainters and generators because they allow the passage of more spatial information beyond the bottleneck from the downsample- to the upsample-region [8] [20] [38].

3.3 Attention

One of the missing pieces from the final model is the absence of any attention mechanism, which is a prominent feature in many of the more recent image generation and inpainting papers reserached for this project. Several were investigated, including Dynamic Attention Maps [8], Self-Attention GAN [46], Contextual Attention [44], and Non-local networks [38]. However, they all work on adding some kind of attention block, consisting of several convolutional layers. It was decided that the size increase of these blocks to not be worth the additional quality. As an example the proposed generator is about nine convolutional layers in total with 512 feature maps at the deepest, while one block of self attention would require four 1x1 convolutional maps. That's close to a 50% total size increase and the self attention generator uses two of these blocks. A non-local block requires about the same. So, while there's no denying the impressive results, like Dynamic attention maps achieving a 90% SSIM score on its recreated 64×64 pixel block on the Celeb-A-HQ dataset [8, table. 1]. The trade-off in size was found to be too much given the goal of hardware implementation. There is also a practical limitation. Training these networks with two self-attention blocks in the generator and discriminator required more than the 24 GB of GPU RAM available. More prototyping would have required a compromise with the architecture or training setup, or potentially even more time to develop a much smaller attention mechanism.

3.4 Generator

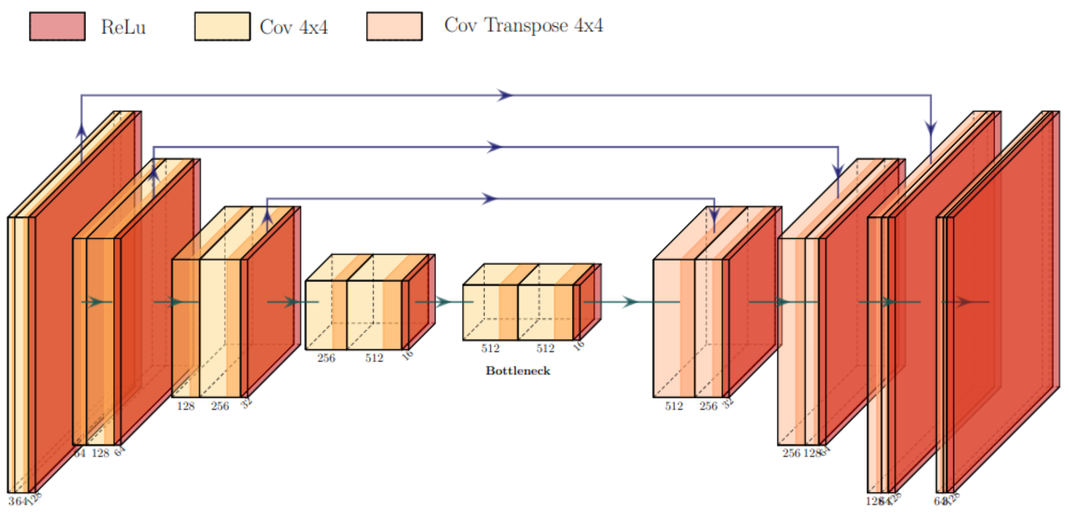


Figure 3.2: Defect Generator architecture. Credit to [1] for the visualizer

The 'Defect Generator' consists of only a few modules. Each module contains a convolutional or transpose convolutional layer which also either halves or doubles the spatial size of the input. Each module also has an activation function which is ReLU for all layers, even the output. ReLU is used because of its performance 5.4c, and also because the generator is supposed to hallucinate pixels, so it's convenient that the activation function outputs only positive values. Furthermore, the ReLU activations will result in more 'zeroed' activations, which should help with sparsity compression 6. The generator uses Residual connections between the first three encoder and decoder blocks.

The final output of the layer is the matrix addition of the residual input with the output of the activation function of that layer. The idea for this is to hopefully increase the spatial information carried through the network to hopefully aid in generation of high resolution features. It should also reduce training error [15]. Furthermore, there is the previously mentioned argument for improved hardware implementation, which is also a motivator in implementing this feature. The design goals for the architecture, is for it to be as simple and small as possible. A simpler network should be easier to implement in hardware, and given that the defect problem is a small part of the image we are confident that simplicity will also mean a larger portion of the network can be removed or more aggresively quantized. The question of hardware implementation is further investigated in chapter 6.

3.5 Discriminator

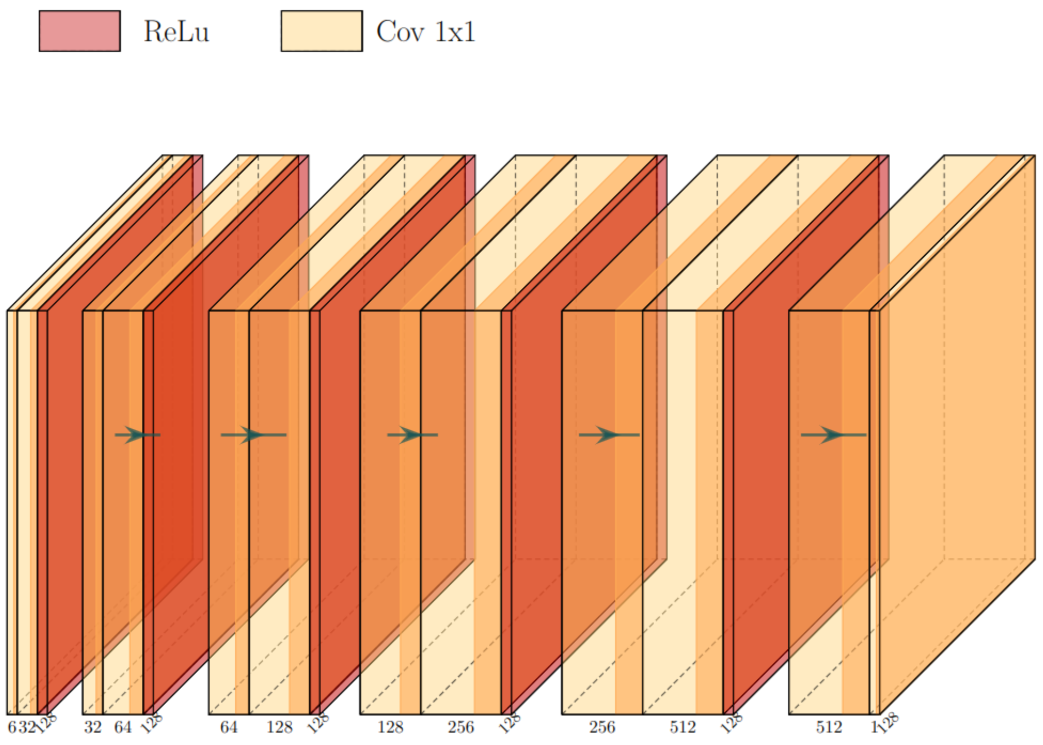


Figure 3.3: Pixel Discriminator architecture. Credit to [1] for the visualizer

The discriminator is a 'Pixel' discriminator based originally on the discriminator proposed in [20], but with some modifications. As the discriminator output needs to be uncapped for the WGAN objective there is no output activation at the final layer, all other layers use LeakyReLU with a parameter of 0.2. There is also no batchnorm due to the mapping problem discussed in [14]. [14] [8] [44] all advocate a discriminator based on resnet with either a global (fully connected final layer) or in addition to a local critique. It was found not to not work very well 5.3, and for this application we argue that the large consistency checks that comes from a global discriminator, and which most other inpainting papers put such emphasis on, are unnecessary because the defects are so small. Instead, the best results came with keeping the original spatial size of the incoming image while also expanding the feature space. The final output of the discriminator in this project is instead an unconstrained critic for every pixel in the original image.

3.6 Local Defect Loss

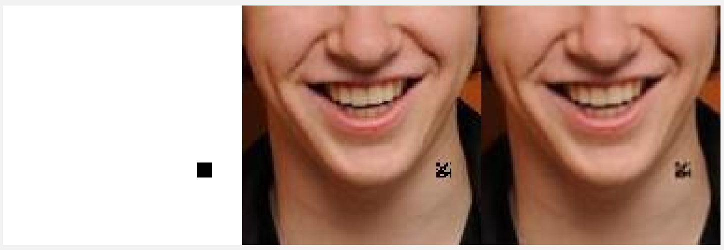
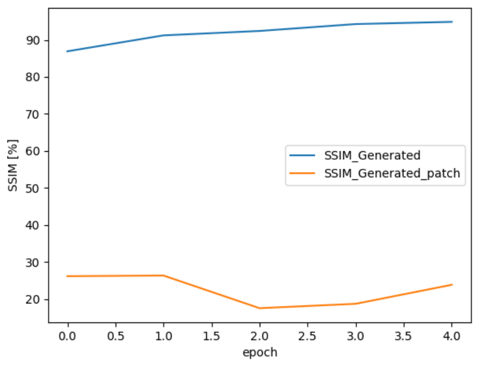
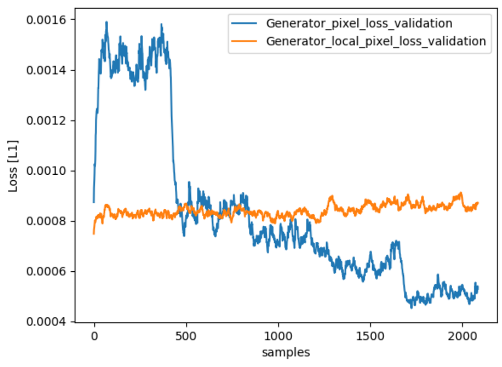


Figure 3.4: Image showing poor convergence towards defect correction with no local objective. From left to right is the **the defect mask**, center shows the **defective image**, and to the right is the **generator output**.

Figure 3.5: WGAN pixel loss with no defect correction



(a) SSIM score of a WGAN trained with only global L1-objective.(Higher is better)



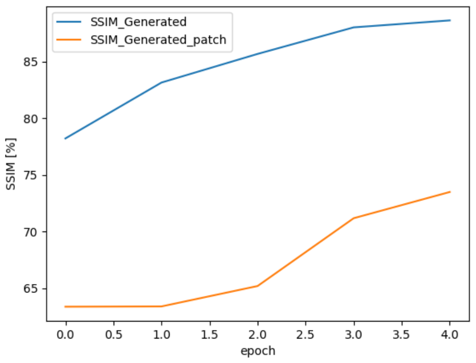
(b) L1 loss curves for WGAN trained with only global L1-objective. (Lower is better)

Testing revealed that a standard cGAN approach like [20] with a global adversarial loss and accompanying regularization term like L1 or L2 would have poor convergence on the defect patch 3.5a. We struggled for a long time to reconcile this because of the impressive results inpainting models have so far achieved, with respect to the large parts of the image they can reconstruct. It would seem trivial then to adapt this for small-scale 'touch ups' like the ones considered for this project. However, it turns out that global objectives alone struggle to teach the generator effectively, this is especially evident in figure 3.5b where there is a global L1-component and you can see the global loss go down which signals an improvement, but around the defect patch its stationary or even increasing. Its the same with the SSIM scores 3.5a. The reason for this could be that the defect patch is so small both in terms of size but also in terms of the loss penalty it creates, and so the model sees no gain when adjusting this region. To remedy this a second regularization term was added for the defect area using the mask created by the dataset, and based on the L2 loss [equation: 3.1], here $(\mathbf{x}_{local}, \mathbf{y}_{local})$ are the generated sample and ground truth respectively. This expands the complete objective function with respect to the generator to equation 3.2, where $G*$ is the complete generator training objective, $\mathcal{L}_{WGAN}(G, D)$ is the original adversarial objective 2.27, and $(\mathcal{L}_{L1}(G), \mathcal{L}_{L1\text{local}}(G))$ are the global and local L1 objective alongside their scalar parameters $(\lambda_{\text{global}}, \lambda_{\text{local}})$.

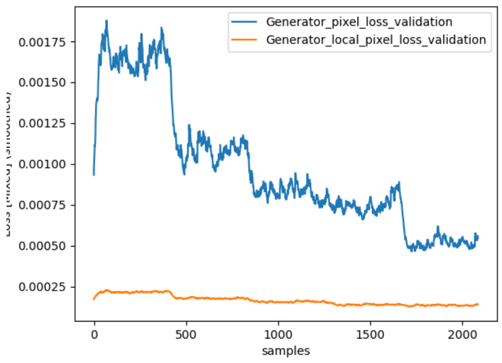
$$\mathcal{L}_{L2\text{local}}(\mathbf{x}_{local}, \mathbf{y}_{local}) = \{l_1, ..., l_N\}^\top, l_n = (x_n - y_n)^2 \tag{3.1}$$

$$G* = \arg\max_{G}\min_{D}\mathcal{L}_{cGAN}(G, D) + \lambda_{\text{global}}\mathcal{L}_{L1}(G) + \lambda_{\text{local}}\mathcal{L}_{L2\text{local}}(G) \tag{3.2}$$

Figure 3.6: WGAN pixel loss with defect correction



(a) SSIM score of a WGAN trained with both global-L1 and defect L2. (Higher is better)



(b) L1 loss curves for WGAN trained with both global-L1 and defect L2. (Lower is better)

There is a clear and marked improvement in the quality of both the loss curves and SSIM scores with the added defect objective. You can also see what was discussed earlier about the defect area, the loss is much smaller for this defect region. Figure 3.6b shows both losses unscaled and their distance from ground truth.

3.7 Latent loss and 'Dual' Encoder Feature

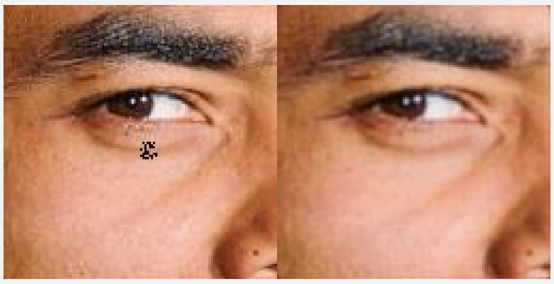


Figure 3.7: Image shows a 'blurring' from the model prediction. From left to right is the **defective image** and the **generator output**.

This is a new objective function and the motivation are based on the results above. The original idea behind latent loss penalties was created for text-to-speech networks [22, p. 2] and suggested to me by my supervisor Hemin. This was then adapted for a new idea where the same architecture is first trained as an autoencoder, which means it seeks to produce the exact same output as the given input. The assumption is that now the network has learned to recreate high fidelity features in the image and we can use this to teach an inpainter to do the same. Since both networks are the same architectures it is easy to now implement this loss for the network, which simply examines the differences of their respective latent features compared to the autoencoder. It should also be mentioned that 'latent features' in the context of this work is the 'bottleneck' layer in the generator 3.2. The goal is that the defect correction network learns to preserve the parts of the image not directly associated with the defect. Intuitively this should result in a higher quality image. The loss is then simply the distance between the two features, measured using either L1 2.32 or L2 2.33, there will be more emphasis on this in chapter 5.

There are two implementations of the latent loss featured in this project. The first is the architecture gets trained as an autoencoder and is then used later as an objective for the same architecture training on the inpainting task. The autoencoder will then simply do inference (static output) and an L2 loss penalty will be applied to the difference of the latent space between the autoencoder and inpainter for every training sample. The second implementation is a parallel scheme called DualEncoder. Here one architecture is traind on both objectives at the same time. The same architecture does two individual predictions, one with the ground truth image and one on the defect image. An L2 loss is then applied to the difference between the two latent spaces and the normal pixel loss as well as adversarial loss for both runs. Finally all gradients are summed for backpropagation. The second implementation is slower to train, but should have the intuitive advantage that the quality of the image could be higher since both objectives continue to train.

Algorithm 1 Latent Feature Loss Algorithm

```
$G_A$ Autoencoder, $G_I$ Inpainter, $D$ Critic, $L$ Latent Features,
  $y \leftarrow$ Ground truth image
  $\hat{y} \leftarrow$ Defected image
  $x, L_A \leftarrow G_A(y)$
  $\hat{x}, L_I \leftarrow G_I(\hat{y})$
  $\text{pred}[\hat{x}] \leftarrow D(G_I(\hat{y}))$

  $\mathcal{L}_{\text{Latent}} = \mathcal{L}_{\text{L1}}(L_I, L_A)$
  $\mathcal{L}_{\text{Pixel}} = \mathcal{L}_{\text{L1}}(\hat{y}, \hat{x})\lambda_{\text{pixel loss}}$
  $\mathcal{L}_{\text{Local pixel}} = \mathcal{L}_{\text{L1}}(\hat{y}_{\text{defect}}, \hat{x}_{\text{defect}})\lambda_{\text{local pixel loss}}$
  $\mathcal{L}_{\text{total}} = \mathcal{L}_{\text{GAN}}(\text{pred}[\hat{x}]) + \mathcal{L}_{\text{Pixel}} + \mathcal{L}_{\text{Local pixel}} + \mathcal{L}_{\text{Latent}}$
```

Algorithm 2 DualEncoder Algorithm

```
$G_D$ DualEncoder, $D$ Critic, $L$ Latent Features,
  $y \leftarrow$ Ground truth image
  $\hat{y} \leftarrow$ Defected image
  $x, L_A \leftarrow G_D(y)$
  $\hat{x}, L_I \leftarrow G_D(\hat{y})$
  $\text{pred}[x] \leftarrow D(G_D(y))$
  $\text{pred}[\hat{x}] \leftarrow D(G_D(\hat{y}))$

  $\mathcal{L}_{\text{Latent}} = \mathcal{L}_{\text{L1}}(L_{G(I)}, L_{G(A)})$
  $\mathcal{L}_{\text{Pixel}} = \mathcal{L}_{\text{L1}}(y, x)\lambda_{\text{pixel loss}}$
  $\mathcal{L}_{\text{Local pixel}} = \mathcal{L}_{\text{L2}}(y_{\text{defect}}, x_{\text{defect}})\lambda_{\text{local pixel loss}}$

  $\mathcal{L}_{\text{total}} = \mathcal{L}_{\text{GAN}}(\text{pred}[x], \text{pred}[\hat{x}]) + \mathcal{L}_{\text{Pixel}}(x, y) + \mathcal{L}_{\text{Pixel}}(\hat{x}, y) + \mathcal{L}_{\text{Local pixel}}(\hat{x}, y) + \mathcal{L}_{\text{Latent}}$
```

3.8 Generative Inpainter objective

A short updater for a purely generative inpainter has also been developed. In this case there is no Discriminator and the training architecture is no longer a 'GAN' as there is no adversarial component to the loss. This is meant as both experimentation, but also as a control to the results of the GAN implementations.

Algorithm 3 Generative Loss Algorithm

```
$G_I$ Inpainter
  $y \leftarrow$ Ground truth image
  $\hat{y} \leftarrow$ Defected image
  $x, L_A \leftarrow G_I(y)$
  $\hat{x}, L_I \leftarrow G_I(\hat{y})$

  $\mathcal{L}_{\text{Latent}} = \mathcal{L}_{\text{L1}}(L_{G(I)}, L_{G(A)})$
  $\mathcal{L}_{\text{Pixel}} = \mathcal{L}_{\text{L1}}(y, x)\lambda_{\text{pixel loss}}$
  $\mathcal{L}_{\text{Local pixel}} = \mathcal{L}_{\text{L2}}(y_{\text{defect}}, x_{\text{defect}})\lambda_{\text{local pixel loss}}$

  $\mathcal{L}_{\text{total}} = \mathcal{L}_{\text{Pixel}}(\hat{x}, y) + \mathcal{L}_{\text{Pixel}}(x, y) + \mathcal{L}_{\text{Local pixel}}(\hat{x}, y) + \mathcal{L}_{\text{Latent}}$
```

3.9 Early Stopping

This project does not contain an early stopping feature, which is often used to prevent the model performance from degrading as it will eventually start to overfit on the training data. Instead a crude quality metric was employed called the 'model score'. This is simply a score given to the model at the end of every training epoch based on an evaluation of 500 samples. The 'model score' is scalar sum of $(PSNR + SSIM * 100)$ for both the global image and defect area. A function would compare the model score with the score at all other epochs and save the model if this result was better previous ones. While this is not a perfect solution as training runs until completion, even if model performance severely degraded hours before. It will save the best performing model because overfitting models should score worse on the validation data. Below is a graph showing the model score across epochs.

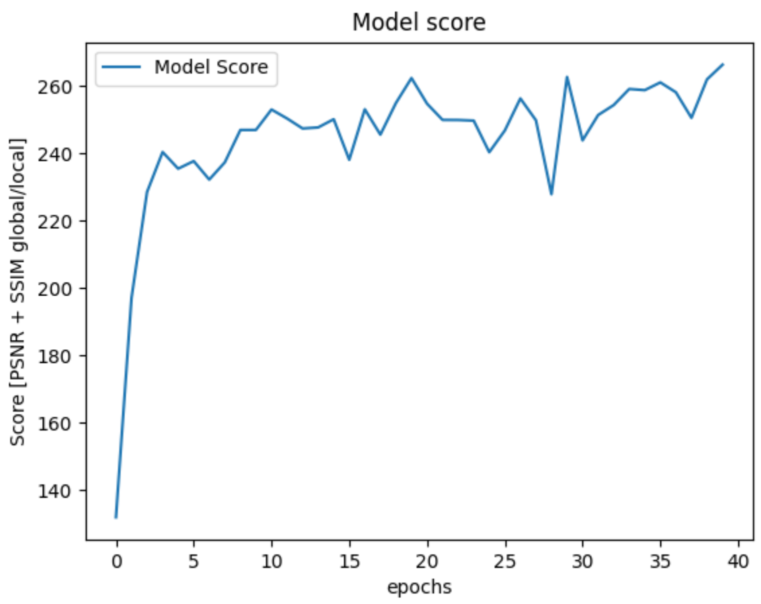


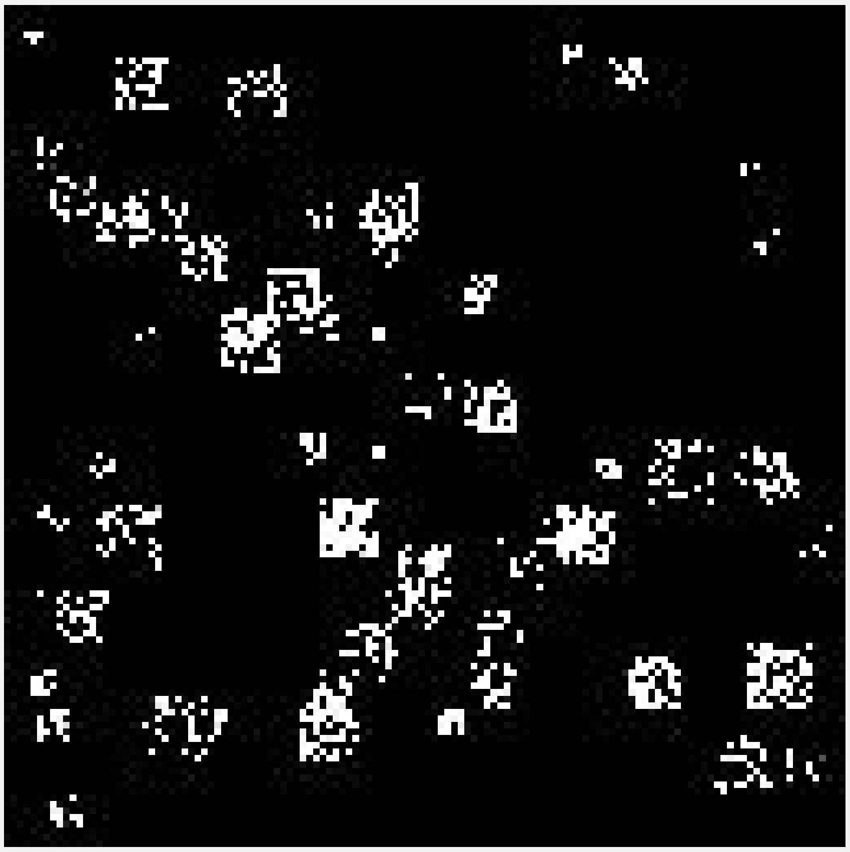
Figure 3.8: Figure showing a model score across training epochs.

3.10 Training Data

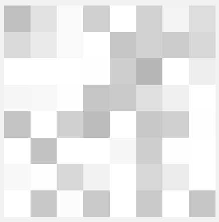
I use the Celeb-A [25] dataset for the project primarily because of its size and use in other inpainting papers [44] [8]. The dataset also had key attributes important for this project which is a good sample size (200K in total). Individual image size is almost always bigger than [256 × 256], which is important for cropping and integration into the dataset class developed. Furthermore, faces and clothes represent a difficult challenge for the model to recreate. As discussed in [3.11] there are many training sets which would probably work well. In fact integrating new training sets is trivial as long as they are in the '.jpg' format and individual samples size is a minimum of [256×256]. This is because training data is an amalgamation of the synthetic defects alongside the image itself. This means the same image can serve as a different sample every time its drawn. An example of the defects is shown below.

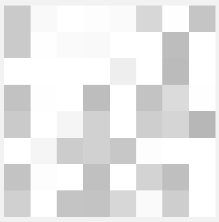
3.11 Fitting the Training data (Dataset class)

Figure 3.9: Defect pattern samples


(a) Figure showing 50 defects implanted on a mask

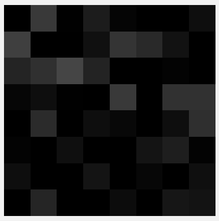
(b) White defect pattern


(c) White defect pattern


(d) White defect pattern

(e) Black defect pattern

(f) Black defect pattern


(g) Black defect pattern

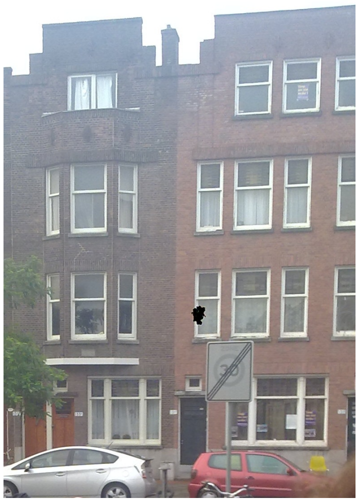


Figure 3.10: picture of a damaged image sensor, credit: [23, Vera de Kok], license CC BY-SA 3.0

No model in the world will converge with any amount of success on bad data. As discussed in 2.11, the generator will attempt to move its distribution closer to the training distributions and therefore the training data better be good. The first real challenge here is that to our knowledge there exists no large dataset on image-sensor defects. There also seemed to be little point in trying to engineer a defect on a real-world sensor as the number of images required, while 'smaller' in terms of relative training requirements, are still much larger than what's practical to photograph. Therefore, an algorithms was made that embeds random defects onto images using boolean masks.

Algorithm 4 Defect Generation Algorithm

```
$n \leftarrow \mathcal{N}$(num defects)
**for** i in range(n) **do**
    $d \leftarrow \mathcal{N}$(Defect size range)
    Defect-type $\leftarrow \mathcal{N}$(white, black)(**bool**)
    $y \leftarrow \mathcal{N}[d, imageHeight - d]$
    $x \leftarrow \mathcal{N}[d, ImageWidth - d]$
    mask $\leftarrow \mathcal{N}$(Defect-type(defect gradient))$[d \times d]$
    defective image $\leftarrow image(mask)[y, x]$
**end for**
**return** defective image, mask
```

The algorithm 4 takes a random integer from a specified range determined pre-training called "Defect size", this determines the size of a square which is filled with a boolean mask drawn from random Gaussian noise. The mask is then imposed over a random set of coordinates within the image creating the example in 3.11b.

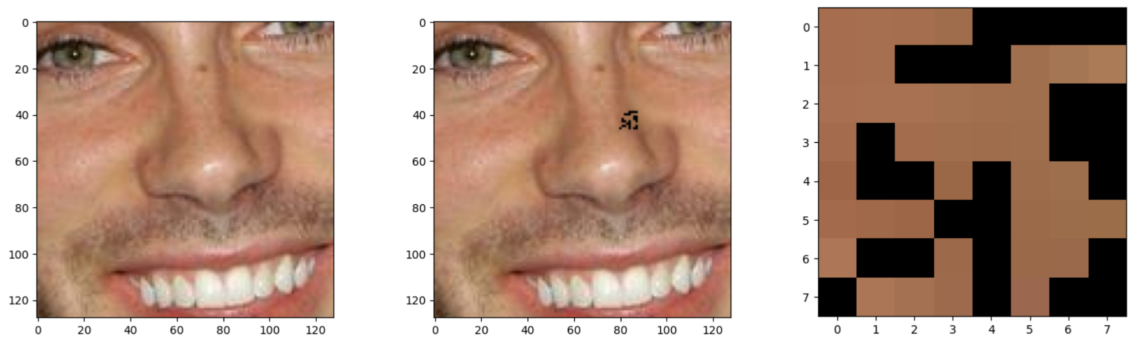


Figure 3.11: Defect implantation

(a) ground truth sample (b) Sample with defect (c) Cutout of the defect area

The Dataset algorithm can generate one or multiple defects of variable size, at random locations, and of different types either white (hot) or black (dead) defects. The nature of the defects will also vary in severity to mimic the real life nature of the sensor defect. It is well known that pixel response and therefore pixel defects can change as a result of dark current and changes in temperature [21], [41]. With this in mind the defects have a random variation in their intensity for both dead and white defect to simulate abnormal but not completely saturated pixel responses. This is meant to mimick the no-linearity of these defects. The benefits of this generative approach is that it will work on any image multiple times and therefore a great number of high quality image datasets become available. Another important point about the chosen training data is the consideration of inductive bias. In a real world example it would be beneficial for the model to make good approximations given the bias of the image sensor itself, rather than learning representations or 'quirks' from many different sensors. The model should also learn to hallucinate pixels from a large number of representations in the scene like clothing, faces, background, texture etc. The project only accounts for the second consideration. Although, eventually there could be merit in exploring the first.

During the conversion process the images are loaded using the PIL library and then converted using a Pytorch function which scales the image values between $[0, 1]$. This is done for training stability. The function also changes the datatype from integer to 32-bit floats. This is to done to increase precision during training. For outputs and inference of the models the RGB dimensions of the image is converted to 8-bit integers ranging from$[0, 255]$.

3.12 Training setup and parameters

This section will focus on most of the training parameters used for the project. Below in table 3.12 are the typical values for a given training run. $\lambda_{\text{Pixel loss}}$ will vary if it is an autoencoder (higher) or inpainter (lower) being trained. $\lambda_{\text{n\_crit}}$ is the parameter for how many discriminator updates per generator update, this is a method that's been used to help train GAN networks. We First encountered it in [3], but there have been various implementation and update rules [14] [27] [46]. It was found through experimentation that between 1-2 works well, which is consistent with the findings in [46] when they employed similar strategies using spectral norm in both networks and a two tiered learning rule [17]. Eventually it settled on a learning rate of 0.0004 for the discriminator and bound the generator rate to be half that to take advantage of [17]. This is a little aggressive compared to most other update rules [14] [27] [44] [14], but again consistent with [46] when using the same regularization schemes. The project uses the Adam optimizer with the same parameters as [46] and [27] chose for image generation.

Table 3.1: Table of training parameters

Parameters list	
Name	Typical value
$\lambda_{\text{Pixel loss}}$	$10 - 100$
$\lambda_{\text{Local pixel loss}}$	100
$\lambda_{\text{Latent loss}}$	1
$\lambda_{\text{n\_crit}}$	2
$\lambda_{\text{learning rate}D}$	0.0004
$\lambda_{\text{learning rate}G}$	$\frac{\lambda_{\text{learning rate}D}}{2}$
Adamβ_1	0
Adamβ_2	0.999

Chapter 4

Results

The results are presented in this chapter. It's important to note that all 'models' are the same architecture, but trained with the different objective functions previously defined. There will also be specific information about what loss functions were used for each. The models are compared across resolutions, defect characteristics, and against more common defect correction like a median filter. Afterwards there is a subjective comparison across different inference images. Additionally, images in the training and validation sets contain a decreasing amount of appropriate images as the resolution increases. This means that results above $[256 \times 256]$ are somewhat skewed as the dataset API will introduce black bars in order to 'produce' a sample at the desired resolution if the image itself is smaller. These black bars will increase the PSNR and SSIM score higher than they should be. For this reason the biggest quantitative results are presented at $[512 \times 512]$.

4.1 Metrics Used

To assess the quality and the performance of the models the project uses two metrics. The first is PSNR (Peak Signal To Noise Ratio), and the second is SSIM (Structural Similarity Index Measure). These were chosen based on their use in other inpainting papers [42][24][8]. In addition, they measure the image in two important way in relation to this work, which is the overall quality measured by the PSNR, and the quality of the pixel reconstruction measured by the SSIM.

SSIM

SSIM was first proposed in 2004 [39] as a metric to more accurately gauge quality in similar terms to humans. The metric scores a desired image against a reference image in a range of $[-1, 1]$, where -1 is complete dissimilarity and 1 is complete similarity. However, a review of the SSIM suggested that negative values are present only when the input image is inverted [6, Note, p. 4]. Since the application of the metric in this case ranges between 0 and 1 they are scaled as a percentage for ease of comparison. Functionally, SSIM is computed by a sliding window kernel. It aims to capture reconstruction by comparing several types of information between the images, defined as luminance4.2, contrast4.3, and structure4.4. For color images the results are summed and then divided by the channels. One downside to this metric is that the implementation requires a minimum window of $[7 \times 7]$ which makes scoring defect corrections of smaller defects more challenging. Therefore, most results are captured with defect regions at a minimum of $[7 \times 7]$. Shown below are the three principal SSIM components, where $\mathbf{x}$ and $\mathbf{y}$ denote the test and reference images. μ is the mean, σ^2 is the variance, and σ the covariance.

$$SSIM(\mathbf{x},\, \mathbf{y}) = \frac{(2\mu_x\mu_y + c_1)(2\sigma_{xy} + c_2)}{(\mu_x^2 + \mu_y^2 + c_1)(\sigma_x^2 + \sigma_y^2 + c_1)} \tag{4.1}$$

$$\text{luminance } l(\mathbf{x},\, \mathbf{y}) = \frac{(2\mu_x\mu_y + c_1)}{(\mu_x^2 + \mu_y^2 + c_1)} \tag{4.2}$$

$$\text{contrast } c(\mathbf{x},\, \mathbf{y}) = \frac{(2\sigma_x\sigma_y + c_2)}{(\sigma_x^2 + \sigma_y^2 + c_2)} \tag{4.3}$$

$$\text{structure } s(\mathbf{x},\, \mathbf{y}) = \frac{\sigma_{xy} + \frac{c_2}{2}}{\sigma_x\sigma_y + \frac{c_2}{2}} \tag{4.4}$$

$$\begin{aligned} c_1 &= (K_1 \cdot \text{data-range})^2 \\ c_2 &= (K_2 \cdot \text{data-ange})^2 \\ \text{data-range} &= \text{Max(image)} - \text{Min(image)} \end{aligned} \tag{4.5}$$

In addition, the values K_1 and K_2 are scalar parameters with default values of 0.01, and 0.03. The image data-range is usually 1 as images are scaled between [0,1] within the framework.

PSNR

PSNR can be defined as a signal processing metric that compares a processed image to its source [10]. Quantitatively it is the maximum possible value of the signal, inversely proportional to the mean squared error, presented in a log scale. It is described by the equations below, where I and K denote the original and processed images, respectively.

$$\text{PSNR} = 20log_10(\frac{\text{MAX}_I^2}{\sqrt{\text{MSE}}}) \tag{4.6}$$

$$\text{MSE} = \sum_{i=0}^{m-1}\sum_{j=o}^{n-1}[I(i,j) - K(i,j)]^2 \tag{4.7}$$

Furthermore, there results are split between global and defect areas to better compare reconstruction. All results are also averaged across 500 random samples to give an estimate of generalized performance. The complete test set constitutes around 40K images.

4.2 Training and inference times

All training and inference were conducted using a single Nvidia RTX 4090 GPU. Since the architecture for all models are the same, the inference times are also the same.

Table 4.1: Table showing model inference times @ 128, 256, 512 and 1024 spatial resolution.

Inference times					
Resolution	128	256	512	1024	unit
Time	23	27	37	76	mS

Table 4.2: Table showing model training times

Inference times			
Name	epochs	time	time per epoch (minutes)
Latent Inpainter	20	3 hours 14 minutes	9.7
Dual encoder	40	9 hours 23 minutes	14.0
Generative inpainter	40	3 hours 36 minutes	5.4
Reference inpainter	20	2 hours 58 minutes	8.9

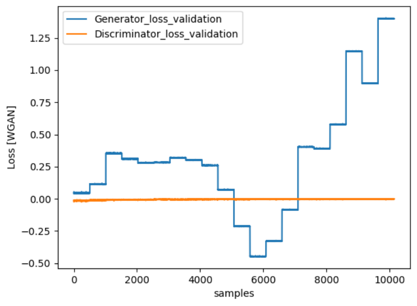
4.3 Latent Inpainter

The latent inpainter was trained using the latent loss objective presented in 3.7. First a model using the same generator architecture 3.4 was trained as an auto encoder, meaning it was given the original un-corrupted image with the objective of recreating it. The auto encoder was trained for 20 epochs. Another instance of the same model architecture was then used to train the latent inpainter, using the auto encoder's latent features 3.2 as a reference for the latent loss objective. Below is the table denoting the models training parameters, and performance.

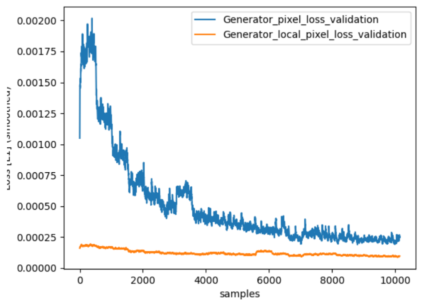
Table 4.3: Table showing the Latent inpainter training parameters

Parameters list	
Name	Value
epochs	20
batch size	16
Defect(s)	black
Defect range	$8-8$
Number of defects	1
$\lambda_{\text{Pixel loss}}$	10
$\lambda_{\text{Local pixel loss}}$	100
$\lambda_{\text{Latent loss}}$	1
n_crit	2
$\lambda_{\text{learning rate}D}$	0.0004
$\lambda_{\text{learning rate}G}$	$\frac{\lambda_{\text{learning rate}D}}{2}$
Adamβ_1	0
Adamβ_2	0.999

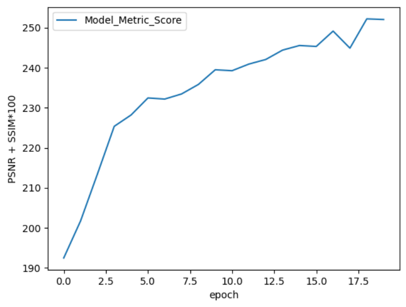
Figure 4.1: Latent inpainter performance graphs



(a) Latent validation losses.



(b) Global and defect pixel validation losses.



(c) Latent inpainter model score. (Higher is better)

Table 4.4: Table of latent inpainter metrics. All metrics are averages across 500 samples.

Latent inpainter score @ 128, 256, 512 image resolution				
Metrics	128	256	512	unit
PSNR global ground truth	34.99	44.08	64.05	dB
PSNR global generated	**38.71**	38.62	37.36	dB
PSNR defect ground truth	11.01	15.24	38.84	dB
PSNR defect generated	31.41	33.19	**42.55**	dB
SSIM global ground truth	99.25	99.82	99.97	%
SSIM global generated	**97.88**	98.03	98.15	%
SSIM defect ground truth	11.09	15.89	40.57	%
SSIM defect generated	79.67	80.56	**86.32**	%

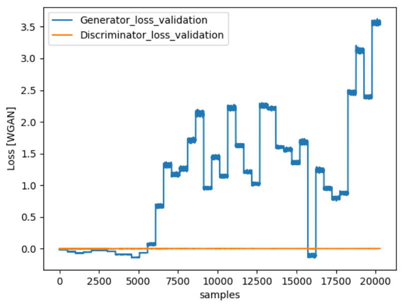
4.4 Dual Encoder

The dual encoder was trained using the alternative objective proposed in 3.7. This means that the autoencoder and inpainter objectives were trained concurrently on the same generator instance. Because of the increasing number of training objectives this objective also traines slower, in terms of convergence per epoch and actual time to train 4.2. To equalize training size the dual encoder was trained for 20 epochs or roughly 48 million samples. This is because the latent inpainter implementation trained its auto encoder for 20 epochs, so the sum of training time is the same.

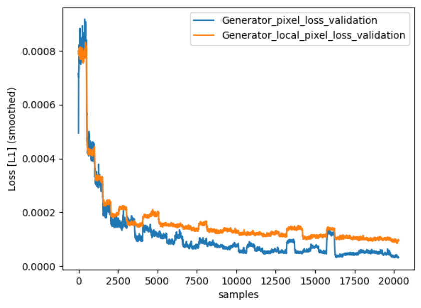
Table 4.5: Table showing the dual encoder training parameters

Parameters list	
Name	Value
epochs	40
batch size	16
Defect(s)	black
Defect range	$8-8$
Number of defects	1
$\lambda_{\text{Pixel loss}}$	100
$\lambda_{\text{Local pixel loss}}$	100
$\lambda_{\text{Latent loss}}$	1
n_crit	2
$\lambda_{\text{learning rate}D}$	0.0004
$\lambda_{\text{learning rate}G}$	$\frac{\lambda_{\text{learning rate}D}}{2}$
Adamβ_1	0
Adamβ_2	0.999

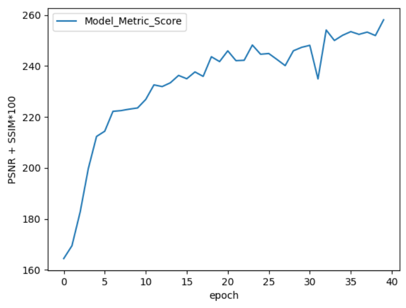
Figure 4.2: DualEncoder performance graphs



(a) Generator and discriminator validation losses.



(b) Global and defect pixel validation losses.



(c) Dual encoder inpainter model score. (Higher is better)

Table 4.6: Table of dual encoder metrics. All metrics are averages across 500 samples.

Dual encoder score @ 128, 256, 512 image resolution				
Metrics	128	256	512	unit
PSNR global ground truth	34.48	43.45	64.81	dB
PSNR global generated	45.46	**46.13**	45.81	dB
PSNR defect ground truth	10.49	14.01	40.10	dB
PSNR defect generated	31.81	32.93	**43.89**	dB
SSIM global ground truth	99.22	99.82	99.97	%
SSIM global generated	99.34	**99.40**	99.37	%
SSIM defect ground truth	10.52	15.71	42.23	%
SSIM defect generated	79.35	79.54	**84.13**	%

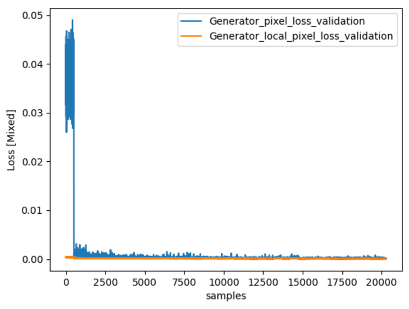
4.5 Generative Inpainter

The generative inpainter was trained using the objective from 3. It is meant as a controling result to check the necessity of the adversarial objective. The generative inpainter also uses the local and latent objectives, with the dual encoder style of implementation during training. Because of the latent objective this model was also trained for 40 epochs. Because there is no need for adversarial training (i.e WGAN) this model trains significantly 4.2 faster than the other models in this project.

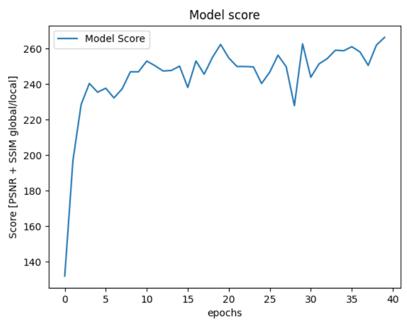
Table 4.7: Table showing the Generative inpainter's training parameters

Parameters list	
Name	Value
epochs	40
batch size	16
Defect(s)	black
Defect range	$8 - 8$
Number of defects	1
$\lambda_{\text{Pixel loss}}$	100
$\lambda_{\text{Local pixel loss}}$	10
$\lambda_{\text{Latent loss}}$	10
n_crit	2
$\lambda_{\text{learning rate}}D$	0.0004
$\lambda_{\text{learning rate}}G$	$\frac{\lambda_{\text{learning rate}}D}{2}$
Adamβ_1	0
Adamβ_2	0.999

Figure 4.3: Generative inpainter performance graphs



(a) Global and defect pixel validation losses.



(b) Generative inpainter model score. (Higher is better)

Table 4.8: Table of generative inpainter metrics. All metrics are averages across 500 samples.

Generative inpainter score @ 128, 256, 512 image resolution				
Metrics	128	256	512	unit
PSNR global ground truth	35.21	44.37	63.29	dB
PSNR global generated	**40.53**	40.41	40.34	dB
PSNR defect ground truth	11.18	15.41	37.35	dB
PSNR defect generated	32.51	34.61	**51.08**	dB
SSIM global ground truth	99.25	99.83	99.97	%
SSIM global generated	**98.69**	98.54	98.59	%
SSIM defect ground truth	11.74	16.45	39.83	%
SSIM defect generated	84.64	83.79	**87.56**	%

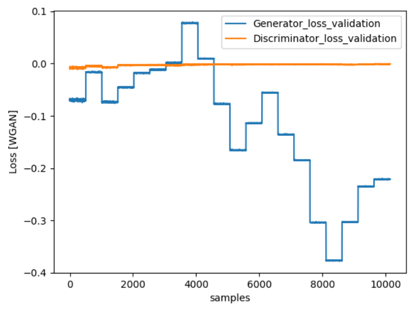
4.6 Inpainter

The Inpainter network is trained without the latent objective to test its efficacy. It has the same objective functions as the Latent Inpainter and Dualencoder networks, except for the latent loss. The network was trained for 20 epochs or roughly 24M training samples.

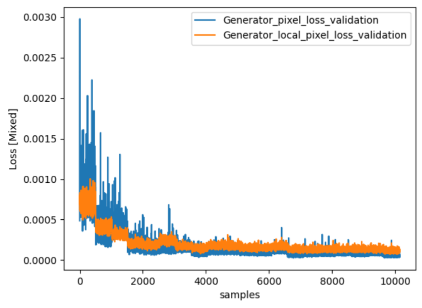
Table 4.9: Table showing the Inpainter's training parameters

Parameters list	
Name	Value
epochs	20
batch size	16
Defect(s)	black
Defect range	$8-8$
Number of defects	1
$\lambda_{\text{Pixel loss}}$	100
$\lambda_{\text{Local pixel loss}}$	100
$\lambda_{\text{Latent loss}}$	0
n_crit	2
$\lambda_{\text{learning rate}D}$	0.0004
$\lambda_{\text{learning rate}G}$	$\frac{\lambda_{\text{learning rate}D}}{2}$
Adamβ_1	0
Adamβ_2	0.999

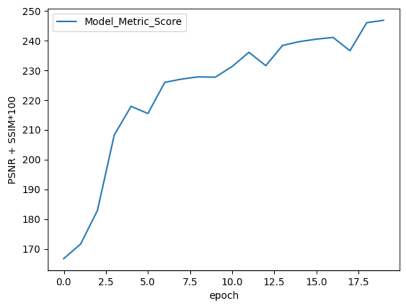
Figure 4.4: Inpainter performance graphs



(a) Generator and discriminator validation losses.



(b) Global and defect pixel validation losses.



(c) Inpainter model score. (Higher is better)

Table 4.10: Table of inpainter metrics. All metrics are averages across 500 samples.

Inpainter score @ 128, 256, 512 image resolution				
Metrics	128	256	512	unit
PSNR global ground truth	34.08	44.25	63.19	dB
PSNR global generated	43.38	44.35	**45.09**	dB
PSNR defect ground truth	10.05	15.30	37.47	dB
PSNR defect generated	29.68	31.62	**40.78**	dB
SSIM global ground truth	99.23	99.83	99.97	%
SSIM global generated	99.35	**99.44**	99.38	%
SSIM defect ground truth	9.73	16.92	39.65	%
SSIM defect generated	72.64	74.03	**79.59**	%

4.7 Comparing model performance

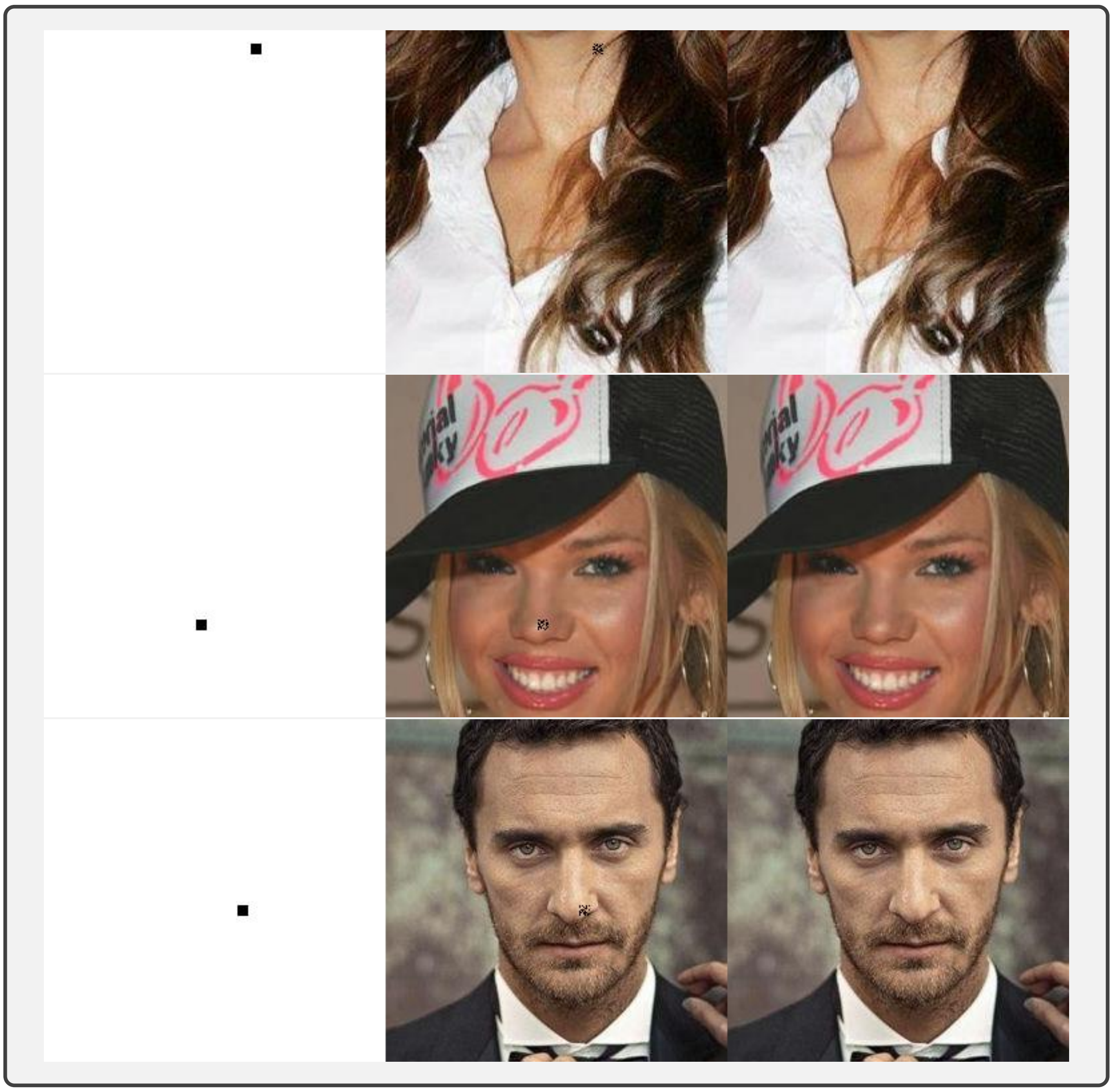
This section provides a comparative overview of the models scores against each other. Below is a table showing the best model scores at any of the tested resolutions compared against eachother.

Table 4.11: Table comparing model performance. All metrics are averages across 500 samples.

Best comparative model score @ any resolution					
Metrics	Latent inpainter	Dual encoder	Generative	Inpainter	unit
PSNR global generated	38.71	**46.13**	40.53	45.09	dB
PSNR defect generated	42.55	43.89	**51.08**	40.78	dB
SSIM global generated	97.88	99.40	98.69	**99.44**	%
SSIM defect generated	86.32	84.13	**87.13**	79.59	%

Comparing all the models in the table above shows that both the Dual encoder and Generative inpainter perform the best. The latent inpainter generally performs the worst, which might be due to its autoencoder being locked after 20 epochs. This reinforces the idea from chapter 3.7, that the theoretical ceiling of quality with the latent objective would perform better if the auto-encoder part continues to train. Although, this idea could be explored further. In subjective terms it is difficult to tell the models a part from one another. Although, the final section in this chapter aims to give some visual indications. Below is also a figure showing some of the inferenced images to give a visual aid alongside the quantifiable results, at this stage of training.

Figure 4.5: DualEncoder inference results @ 256 × 256



From left to right is **the defect mask**, in the center is the **defective image**, and to the right is the **generator output**.

4.8 Dynamic defects and model generalization

Up until now all models have been trained with the dataset set to a fixed size of [8 × 8] and a completely dead blackened pixel showing no response in the image. It's now interesting to test the viability of the models outside their given scope to simulate defects the model may not have been trained for. For the next round of results the models were tested using the previously mentioned dynamic defects 3.11. Both black and white defects are enabled with a gradient for both. Furthermore, there can now be up to three defects in an image and their bounding box can vary from [7 − 12] in size.

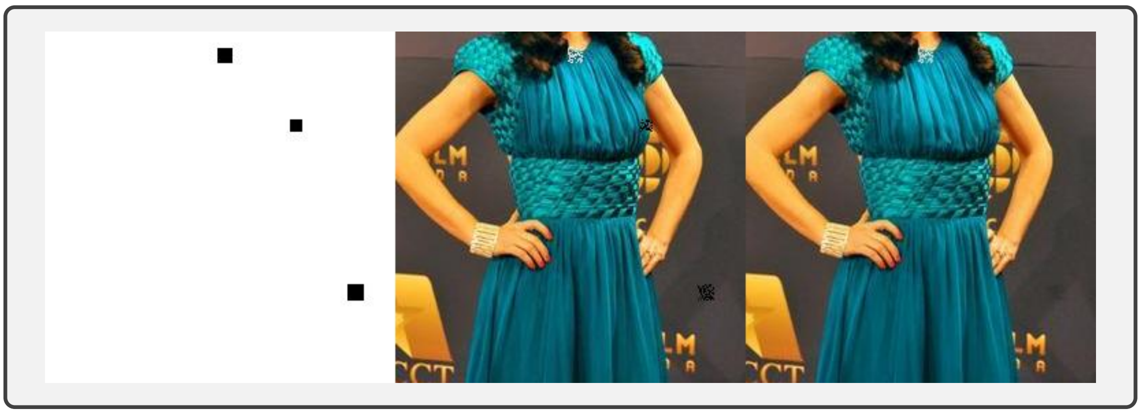
What seems to happen is that since the defects are such a small part of the image most of the models score the same with respect to SSIM on the complete image. For global PSNR its worse and most models seem to drop 3-7dB, which is significant. The worst results come at the defect specific scores. SSIM score drops by around 50% and PSNR plummets by around 13-16dB depending on the model. What seems to happen is that all models have zeroed in on the characteristics of the defect they've trained on, to the point where one could spot the shades of the dark but not quite black defects corrected in the image. In other cases the models seemed to do a very good job when

Table 4.12: Table comparing model performance of dynamic defects without training. All metrics are averages across 500 samples.

Best comparative model score @ [256 × 256] resolution with dynamic defects (no training)						
Metrics	Latent inpainter	Dual encoder	Generative	Inpainter	std. dev.	unit
PSNR global generated	35.39	**39.63**	36.71	38.92	0.14	dB
PSNR defect generated	17.43	16.89	**18.66**	16.89	0.37	dB
SSIM global generated	97.68	99.09	98.24	**99.12**	0.06	%
SSIM defect generated	45.24	41.37	**47.39**	40.92	1.22	%

several and even variable black defects were present, suggesting that the model does to some extent generalize well on the given problem. Where all models fail is with white pixels. This seems to be so far from what they've converged toward that they ignore it along the rest of the image. Below is a typical situation illustrating the results above.

Figure 4.6: Dual encoder inference image @ 256 × 256 on dynamic defects with no prior training



From left to right is **the defect mask**, in the center is the **defective image**, and to the right is the **generator output**.

To further investigate generalization and look at dynamic correction every model was retrained for 5 epochs. The extra training time was again limited by practical contstraints. The below graph shows the training for all models across the five epochs. They all climb up to about the same or slightly below their previous metrics.

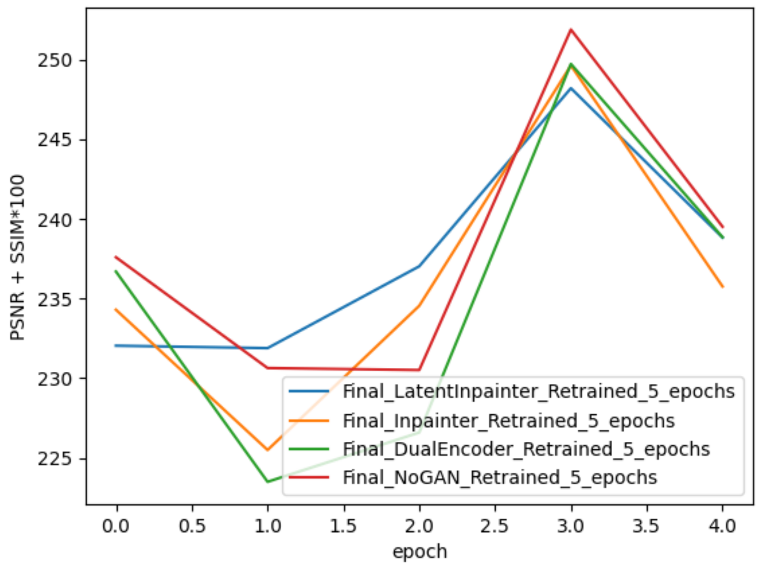


Figure 4.7: Figure showing model scores across epochs of the different models.

What's interesting is to see that they all seem to degrade after an additional epoch of training, before finally starting to figure it out. There is another drop at the end, but this wouldn't be out of the norm for later stages of training when they reach what seems to be the limit at a score of around 250-265, and then start to slightly oscillate around what may be the performance peak. Below is another comparison of inference results comparing performance after re-training.

Table 4.13: Table comparing model performance on dynamic defects after re-training. All metrics are averages across 500 samples.

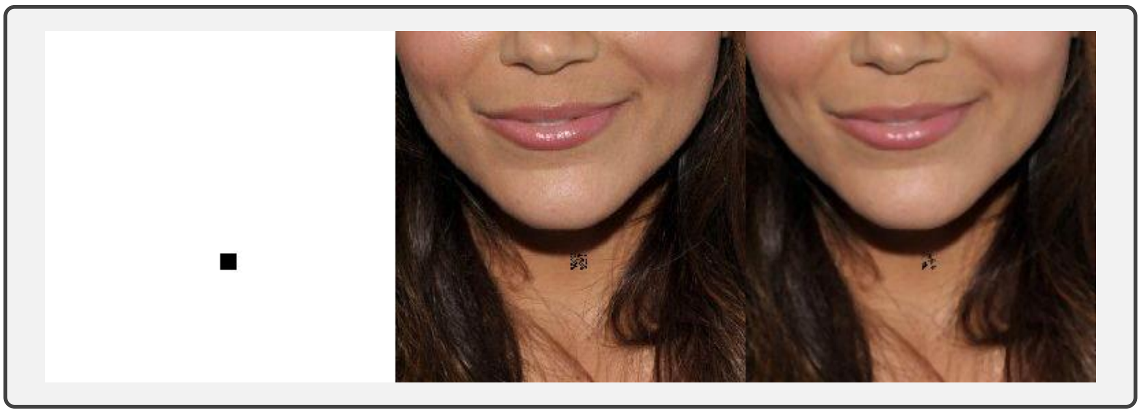
Best comparative model score @ [256 × 256] resolution with dynamic defects (re-trained)					
Metrics	Latent inpainter	Dual encoder	Generative	Inpainter	unit
PSNR global generated	36.74	**42.14**	35.79	42.05	dB
PSNR defect generated	28.67	26.52	**29.52**	27.49	dB
SSIM global generated	96.61	**98.96**	95.49	98.85	%
SSIM defect generated	79.06	71.25	**80.49**	74.54	%

Again the generator models using the dual encoder scheme do better on the dynamic defects as well. The latent inpainter scores the worst, even below the 'vanilla' inpainting model in most metrics. Another thing to note, is that the generative inpainter scores considerabley higher on the defected regions than on the global image. At the same time the GAN based dual encoder and inpainter score better on the overall image. This leads to some suprising conclusions. In general the dual encoder implementation of the latent loss works better. In addition, the Generative inpainter works the best on local defects, but performs worse in overall quality. The regular inpainter does well across the board, this indicates that the contribution of the latent loss is not big, or maybe even positive. It may also be a case of the loss being poorly utilized, but that's for future research to uncover. It is also prudent to reiterate that while training was set for 20 and 40 epochs, which was limited by practical constraints. Therefore, some of the slower training models might have a higher theoretical ceiling, especially for the dynamic defects. Although, this remains speculation.

4.9 Comparing the model to a traditional correction filter.

In order to compare the model performance with a more conventional correction methods, we selected a median filter. This is because median filters are often used to detect and correct pixel defects in image sensors [37][43]. For the comparison the 'median blur' filter from the open source opencv project was used. The median blur filter calculates the median value of the pixels within the chosen kernel area and replaces the central pixel. After some testing a kernel of size [3 × 3] as larger filters gives the overall image an unrealistic drop in quality. Below is an image showing a typical correction result from the median filter, as well as the results compared against the dual encoder model.

Figure 4.8: Image corrected using a median filter @ 256 × 256 resolution



From left to right is **the defect mask**, in the center is the **defective image**, and to the right is the **corrected output**.

Table 4.14: Table comparing model performance against a median filter operation

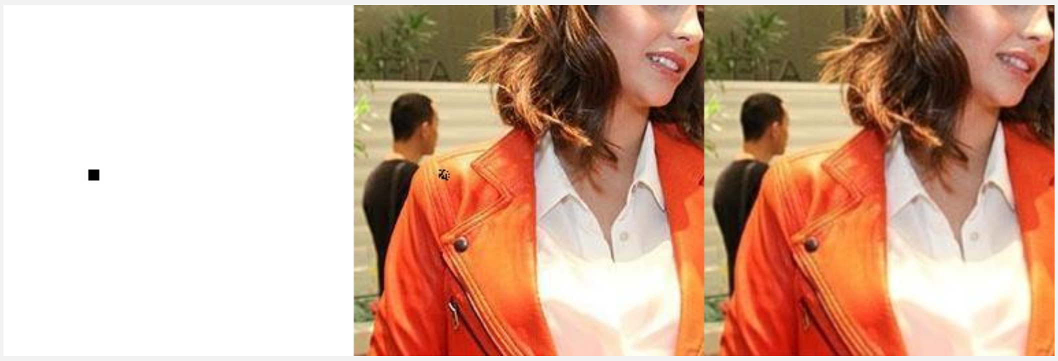
Model vs median filter correction scores @ [256 × 256] resolution with dynamic defects			
Metrics	Dual encoder	median filter	unit
PSNR global generated	**42.14**	32.67	dB
PSNR defect generated	**26.52**	14.14	dB
SSIM global generated	**98.96**	93.31	%
SSIM defect generated	**71.25**	33.25	%

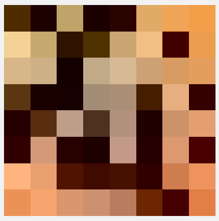
The dual encoder performs significantly better than the filter across all metrics, and especially in the defect region. Overall PSNR is around 10dB higher for the model, and for defect correction the model performs 38% better. There is probably a good argument for only running the median filter in the defect regions, as this would save computation time and increase overall quality. However, The dual encoder would still outperform the filter in this context. The model also has the advantage of flexibility, as it can be re-trained and therefore deployed in different stages of the image sensor pipeline, as needed. Moreover, the model works well on the whole image for several of the resolutions. With this in mind the model can use the entire image as input, which can give the designers more flexibility in terms of hardware implementation.

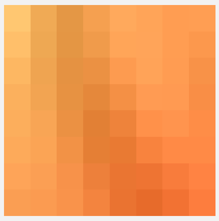
4.10 Inference images and subjective comparison

Below are a few images from the inference runs intended to give some subjective measure of the model performance. First is the complete image and then below is the original defect area, ground truth defect, and finally the various model predictions.

Figure 4.9: Inference comparison 1


(a) Inference image 1


(b) original defect


(c) ground truth

(d) Dual encoder

(e) Latent inpainter

(f) Generative

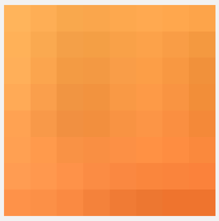
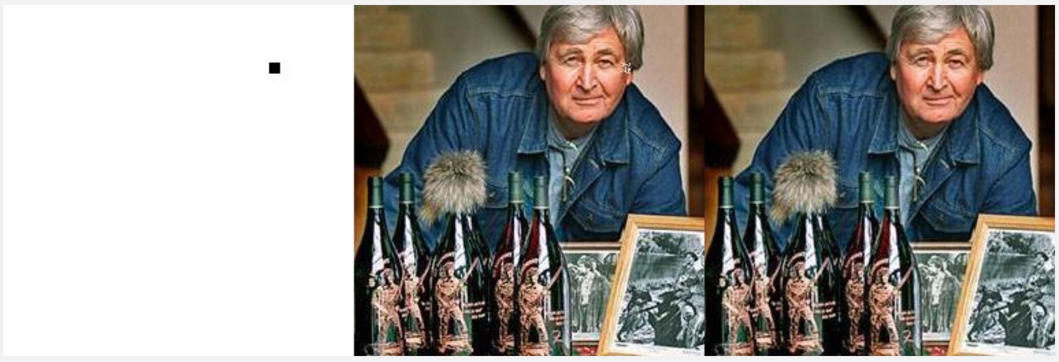
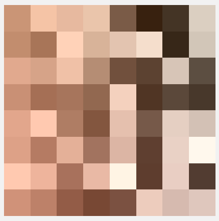
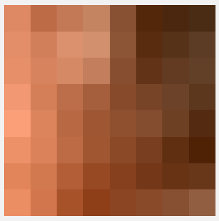
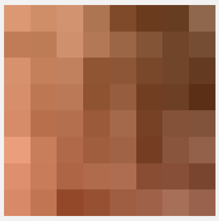

(g) Inpainter

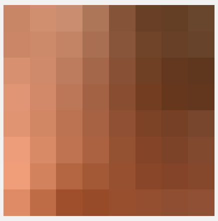
Figure 4.10: Inference comparison 2

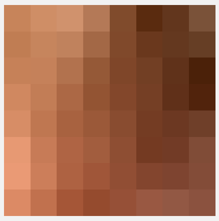

(a) Inference image 2


(b) original defect


(c) ground truth


(d) Dual encoder


(e) Latent inpainter


(f) Generative

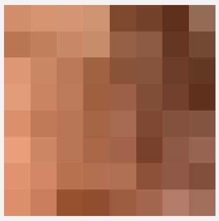
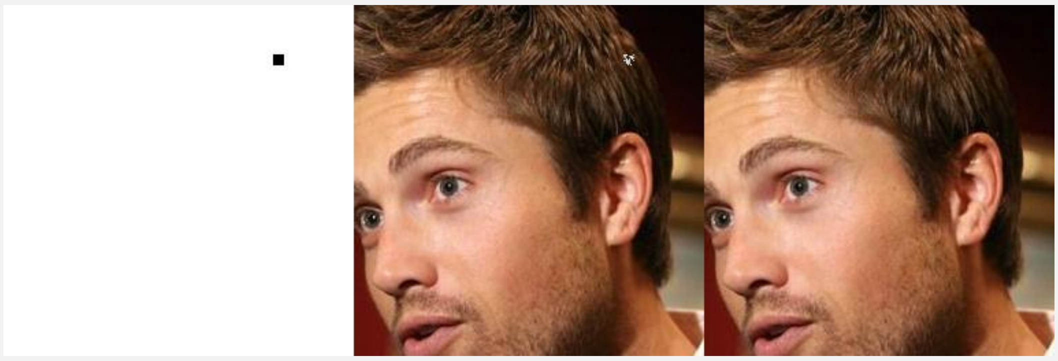
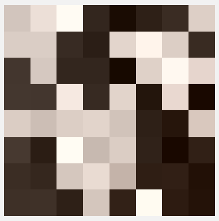
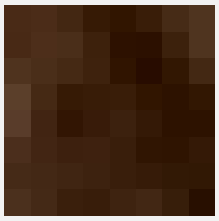
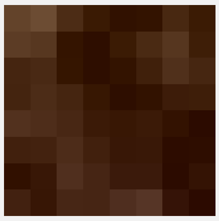

(g) Inpainter

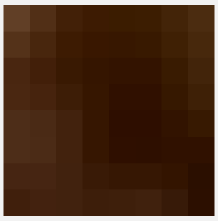
Figure 4.11: Inference comparison 3

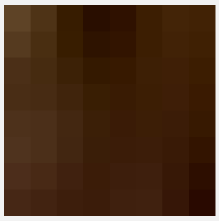

(a) Inference image 3

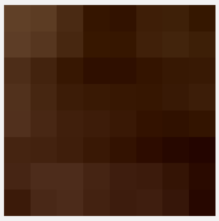

(b) original defect


(c) ground truth


(d) Dual encoder


(e) Latent inpainter


(f) Generative


(g) Inpainter

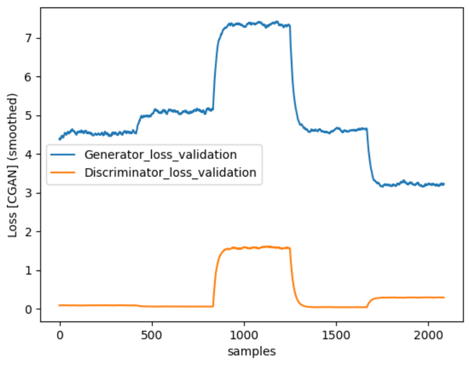
Chapter 5

Discussion

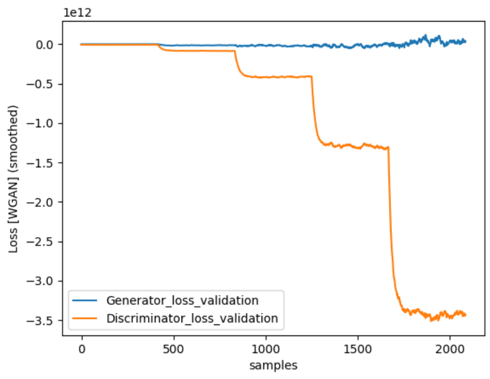
5.1 CGAN to WGAN

Originally work started with the model proposed in [20] as a foundation, which used the CGAN objective to calculate the adversarial loss. This proved very difficult to train and make converge, and this is not something unique to this project, GAN networks are notoriously hard to train and often suffer from what's called mode collapse. This can happen if the generator falls into a small subset of output samples which do very well at fooling the discriminator, but they aren't necessarily good in terms of their quality. Another issue can happen is if the discriminator figures out the generator samples early in training and starts to reject all of them, in this case the GAN stops learning as the generator never manages to fool the discriminator. Therefore, it was very exciting to come across the the original WGAN paper [3] which stated that generated samples using this objective had better diversity and that the objective function helps prevent mode collapse [3, p. 8, Fig. 2]. There are other benefits to switching objective, including the fact that the discriminator loss can be seen as a distance metric between the generated and target distributions. In addition, the objective provides better gradients as the networks trains for longer which is different from the original GAN formulation because the Jenson-Shannon divergence of which the objective is built upon will locally saturate and generate vanishing gradients [3, p. 8]. Furthermore, the WGAN paper claims improved stability in the training process [3, p. 9]. It should also be noted that the CGAN loss is using the alternate objective from [13] where the generator is instead trained to maximize $\mathbb{E}[log(D(G(\mathbf{x})))]$.

Figure 5.1: CGAN and WGAN validation losses



(a) CGAN generator and discriminator validation loss.



(b) WGAN generator and discriminator validation loss.

To test the efficacy of the two objectives the same architecture was trained on both. With the same additional L1 component, and without any form of regularization for 500K samples. The first thing to notice is that both models seem to diverge in the sense that one side of the GAN posts much larger losses than the other. For the CGAN losses it seems like the generator is successfully fooling the discriminator the vast majority of the time, a more ideal curve here would see the discriminator and generator losses converging, which would signal closer competition and possibly a better training run. For the WGAN loss the Discriminator predictions are both uncapped and with no regularizer, and so it increases substantially. However, even with the diverging predictions you'll see in figure 5.2 that the WGAN holds an impressive advantage over the CGAN. This result alongside the experience of training large numbers of GAN networks for this project convinced us that WGAN is the best objective function for this application.

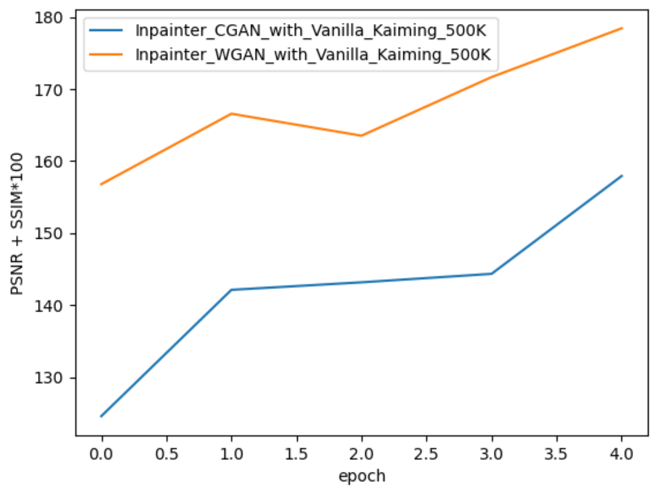
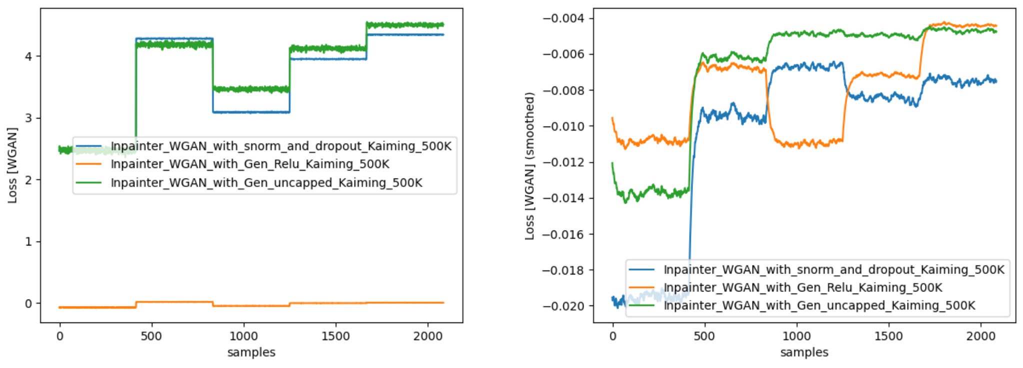


Figure 5.2: Graph showing model score for WGAN and CGAN models. (Higher is better)

5.2 Generator Development

In terms of generator output the last layer should be adapted to the current task. Three different functions were tested based on their use in previous research. These were ReLU, Tanh, and finally an unrestricted output. Originally the generator used the Tanh function for outputs throughout development because it was used for the generator architecture in [20].

Figure 5.3: Comparing generator last layer activation functions

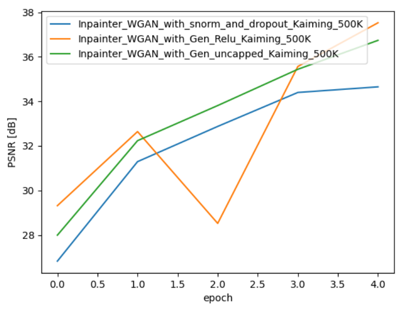


(a) Generator validation loss comparison. (Lower is better)

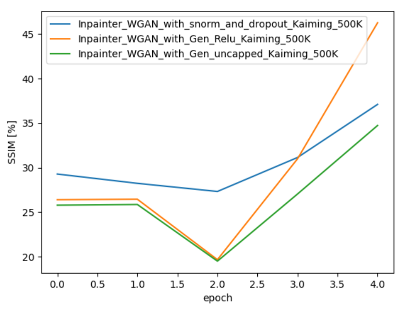
(b) Discriminator validation loss comparison (Higher is better)

The first thing to noticed is the fact that the ReLU output is more well behaved. While all three are similar with respect to their discriminator losses the generator graphs are more interesting. Since both networs try to maximize their respective output here, a Discriminator loss closer to 0 and a generator loss that's higher is better, but a large difference between them could mean that the generator is more easily fooling the discriminator which is bad for training quality. Therefore, the ReLU loss graphs looks better as it shows both networks in closer competition.

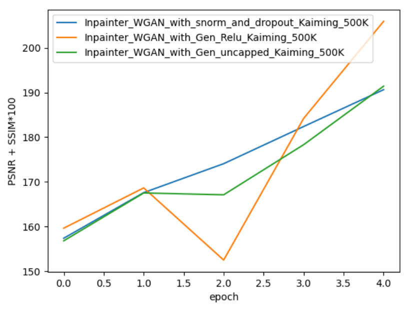
Figure 5.4: Comparing generator score with different last layer activations



(a) Generator activation comparison PSNR score. (Higher is better)



(b) Generator activation comparison SSIM defect score. (Higher is better)



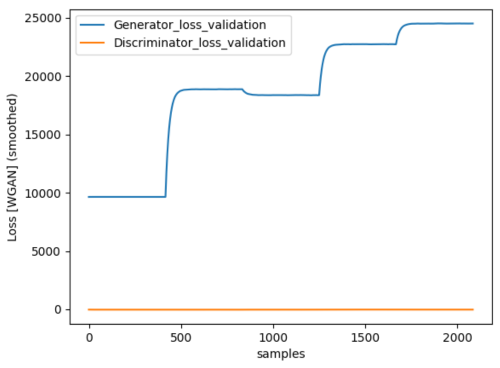
(c) Generator activation comparison model score. (Higher is better)

I pulled three different quality measures for this comparison, image PSNR, defect SSIM, and overall model score. They all show ReLU either in close competition or outright outperforming the other activations. It was especially surprised to see a +10% increase in SSIM defect score which to me implies that ReLU could be very beneficial once a network is fully trained.

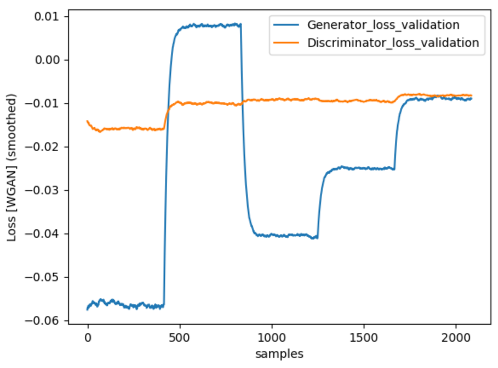
5.3 Discriminator development

As mentioned in 3.5 there are a few considerations with respect to the output of the discriminator. [8] noted that a global enforcement critic like the one from [14] will do better than a local one for irregular hole masks. To test this two training runs where done, where the Discriminator output layer was swapped. The first one is the one shown in 3.5 and the second run featured an additional fully connected layer with a single unconstrained output which is similar to the one in [14], thereby making it a global critic. The results are noted below and show first the generator and discriminator loss curves. With the global discriminator the generator seems to quickly learn to outsmart it. With the pixel discriminator the losses are much closer and are seen to converge towards the end which signals better training. This is finally confirmed below in figure 5.6 which shows the model scores of the two different training runs.

Figure 5.5: Comparing local vs global discriminator outputs



(a) Global discriminator validation loss graphs



(b) Pixel discriminator validation loss graphs.

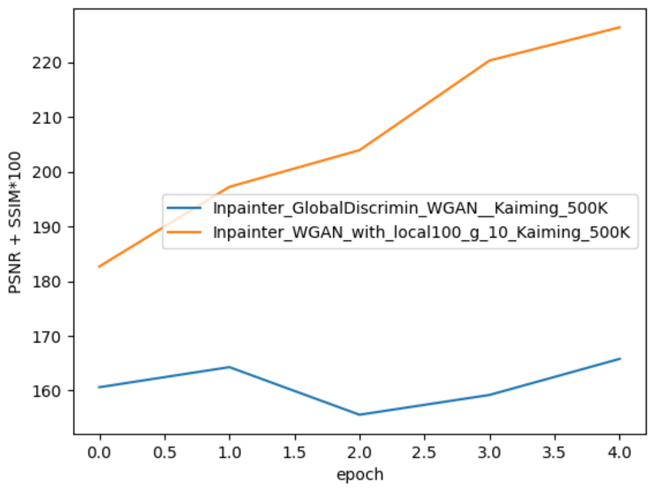
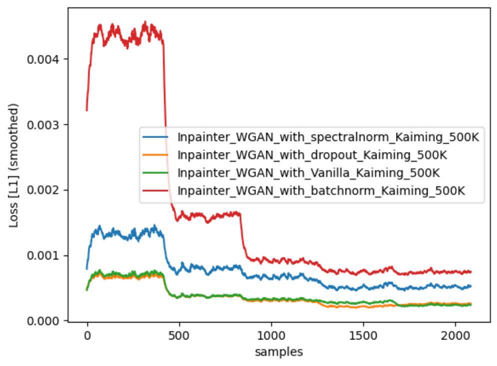


Figure 5.6: Global and pixel discriminator model scores. (Higher is better)

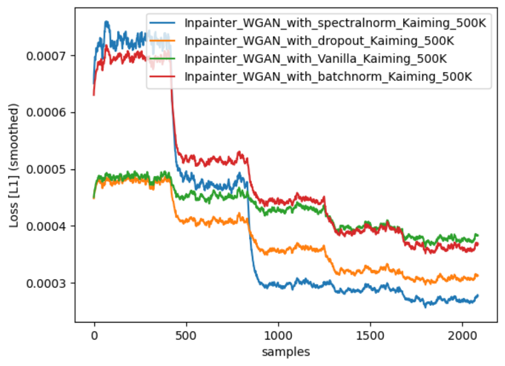
5.4 Regularizations

Regularization techniques were carefully tested and chosen based on performance in other inpainting papers. Suppose you could also make the argument that the defects themselves are a form of regularization as they're closely related to simply injecting noise in an image and tasking the generator with removing that noise. The techniques were chosen because of their reported results [27] [46] [18] [19]. In simple terms Batch normalization normalizes the batch of layer inputs (in this project there's usually 16 images in a batch) which can improve training rates. Spectral normalization is a regularization technique which is made to work more exclusively with WGAN based objective functions. It controls the Lipschitz constant of the network by constraining the spectral norm of each layer [27, Sec. 2]. In this way the Lipschitz constraint of the WGAN objective function is upheld, which should enable the network to converge in a good way. The final regularization is Dropout which simply turns off a random subset of neurons in a given layer. A more thorough explanation of the techniques used in the final model can be found in section 2.14. To test their efficacy three WGAN networks were trained with one regularization at a time for 500K samples. They were then compared to the same architecture with no regularization.

Figure 5.7: Comparing regularization techniques in a WGAN model



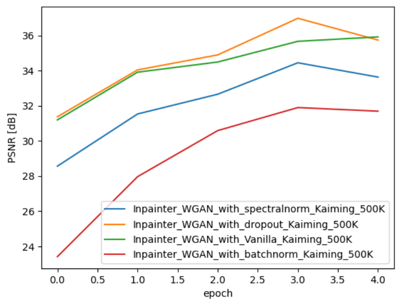
(a) graph comparing a WGAN Generator's pixel loss with the different regularization techniques. (Lower is better)



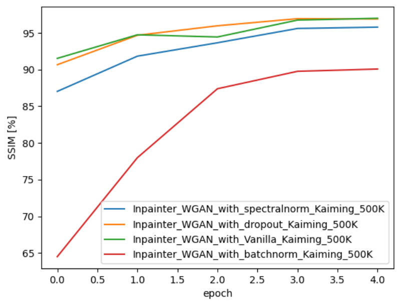
(b) graph comparing a WGAN Generator's defect pixel loss with the different regularization techniques. (Lower is better)

All networks were initialized using Kaiming 2.8 initialization and trained for 500K epochs on the inpainting objective. Spectral normalization is meant to aid training of the networks, in the original paper [27] they apply normalization to the Discriminator, but another paper found it beneficial to add SN to both parts of the GAN [46]. The regularizations have been adopted to both networks based on their recommendation. In figure 5.7 you can see that batchnorm routinely underperforms while dropout and spectral norm perform around or better than the vanilla configuration.

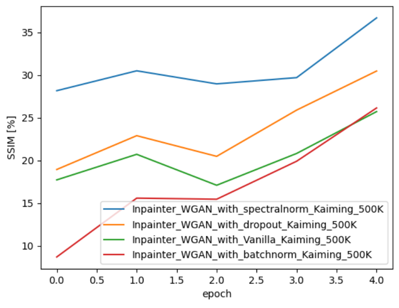
Figure 5.8: Comparing Generator scores with different regularizations.



(a) Generator comparing PSNR score. (Higher is better)



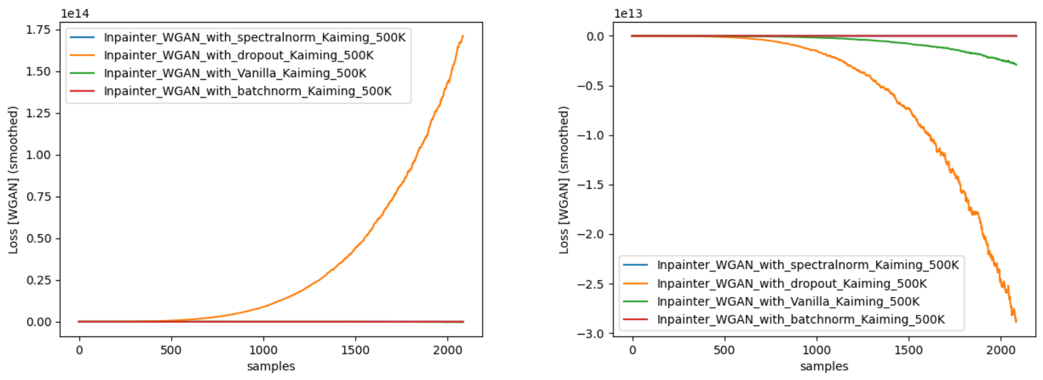
(b) Generator comparing SSIM score. (Higher is better)



(c) Generator comparing SSIM defect score. (Higher is better)

Going over the results shows the same trend as in the pixel loss 5.7 where vanilla and dropout perform the best while spectral norm trails by around 2dB. This is consistent with a trend where spectral normalization will give great results in terms of training stability and convergence, but has a tendency to produce more 'blurry' images. This effect goes away when training for longer, say around 3M samples. Batchnorm, again performs much worse than the others, coming in at around a 3-4dB deficit. Both normalization technique actually exhibit this blurring effect on the images, but the blurring does not completely go away with batch normalization in the same way that it does for spectral normalization. We hypothesize that this is because spectral norrmalization works on the layer constraints, and so this goes away as the model converges and exhibits smaller losses. However, Batchnorm keeps normalizing the input and therefore this blurring might 'move' with the losses. It should be noted again that this is not stated as fact but speculation on our part. WGAN Gradient Penalty [14] which is an improvement on the original Wasserstein objective, was also considered as a regularization method. However, spectral normalization posted better results when tested against GP. In addition, the method introduces additional training time from having to do two discriminator runs each loop to calculate the gradient penalty term itself.

Figure 5.9: Comparing WGAN regularization losses



(a) Generator validation loss comparison.

(b) Discriminator validation loss comparison

Finally, comparing validation losses for the different regularization techniques. In terms of the two normalization techniques, the stated benefit is often training stability and convergence [27] [19] and this holds for both Generator and Discriminator losses 5.9 which shows them both behaving pretty well. The big offender here is instead dropout which causes enormous gradients akin to mode collapse. It's interesting that dropout still performs better than batchnorm given these results. This was one of the reasons for why a purely Generative or 'noGAN' model was developed for the project, in order to investigate what effect the adversarial component had on model performance. Especially, since dropout could perform so well while also seemingly induce large instability to GAN training. Based on these results it was decided to move forward using spectral normalization and dropout as the chosen regularization techniques.

5.5 Creating and evolving the dataset

Initially the algorithm [4] returned the defective image alongside the exact mask to compute the L1-loss. However, during testing it was found that the model converged poorly on this objective. Instead it was found to be much more beneficial to return a boolean mask which was valued 1 at all points in the defect-area and not just the defective-pixels. In this way the model was shown the area it was meant to fix, both defect and original pixels. The masking approach also turned out to be beneficial for the dynamic defect training and inference, as new defects can now just be added to the original mask using tensor slicing. Previously, a crude system would return metadata of the defect size and location. The idea here was that when training it could dynamically vary how much of the surrounding area outside the defect would be shown the generator. Experiments were ran with $[1\times, 2\times, 3\times]$ time the loss regions, but this seemed to only dffuse the correction instead of inpainting it. The metadata approach would also be much less elegant when working with multiple defects as the amount of information would have to grow alongside the new defects, and new functions to parse and select these areas would have to be developed.

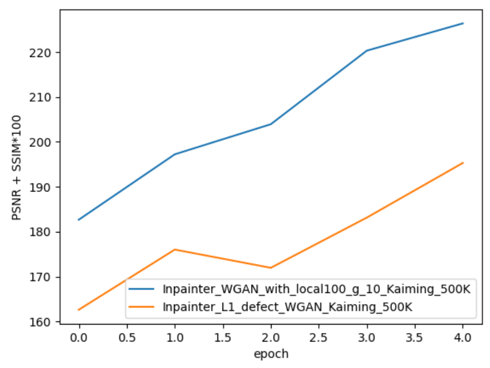
A lot of care was also taken with respect to the speed and throughput of the defect-generation and speed of the dataset-class. The big limiter of this project has been time as there is no shortage of differing techniques, architectures or quirks of statistics to try for the sake of improvement. The time it takes to train a model just to determine if a small feature or layer is better in some way compared to a previus one. A fast dataset-class is imperative for this to function as any sample will need to go through this possible bottleneck. Early in the project I employed different libraries which would need to convert the images between different formats before becoming tensors and training samples. Now the whole process is rewritten to employ only the Pytorch library and offloads data to GPU memory as early as possible.

Another bottleneck in regard to the dataset is the I/O throughput, or simply the act of loading an image from the disk and into memory before processing. This can be a big culprit when it comes to training time as the harddrive usually has the slowest transfer speed of the entire toolchain, which goes [HDD $\rightarrow$ RAM $\rightarrow$ CPU $\rightarrow$ GPU-RAM]. This was especially evident early in the project when trained on the university servers, and while they lacked fast harddrives they had vast amounts of RAM. Therefore, an algorithm was created which would preprocess a number of training samples into a Pytorch file to be stored and when training commenced the dataset would instead load this entire file into RAM. 15-20K samples would take up about 25-30GB of RAM, and since the servers had hundreds of gigabytes it could still leverage a large number of samples. It was determined through testing on my personal machine that this would be an increase in sample-rate of over $\times 7$. The only drawback to this method is that it reduces training diversity because defect and image are now pre-determined, and for every epoch the model would traverse all images but find the same defects. In the 'normal' or previous mode a new defect would be generated for every image and the model could not count on learning just one representation in the same image. Unfortunately, it turned out not to matter, at least for the final parts of the projects when trained locally. At that point it could leverage faster hard drives and the iteration time for forward and backwards passes of the network ended up outweighing the time needed to fetch and process samples.

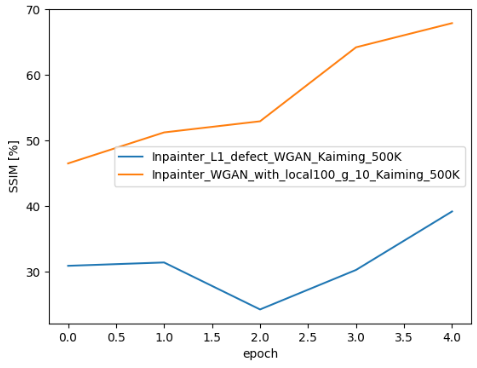
5.6 Defect loss

The use of L1 or L2 loss on the defect pixels was determined experimentally. As with other experiments discussed in this section two equal inpainting architectures were trained for 500K samples and then compared, with the only difference being the defect pixel objective function. The results are shown below and the L1 function actually increases SSIM and PSNR globally compared to the L2 function, but it performs much worse in the defect region. The reason for this could be because L2 punishes outliers more strictly than L1. At the same time the L2 loss won't go all the way to 0 [2] and this could induce some blurriness which is where L1 seems to perform better on a global scale.

Figure 5.10: Comparing L1 vs L2 loss for defect correction



(a) L1 vs L2 inpainter model score. (Higher is better)

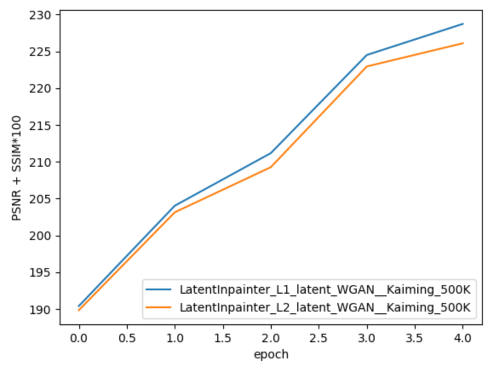


(b) L1 vs L2 inpainter SSIM defect score (Higher is better)

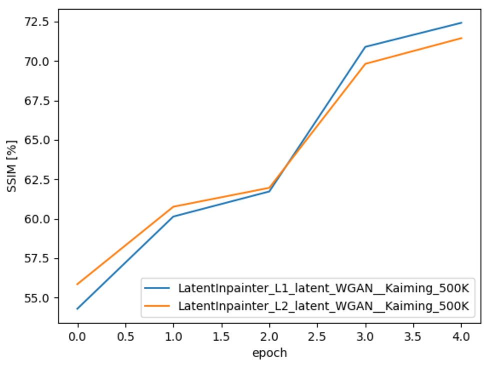
5.7 The Latent Feature Loss

During implementation it was necessary to figure out which loss distance is best between the L1 and L2 metrics. The choice isn't arbitrary either as the L2 distance seems like a logical choice because it more strictly enforces the differences between the two distances. However, [44] noted that WGAN-GP and we believe by extension WGAN works well with the L1 distance metric. It should be beneficial for the different objective functions to use similar metrics, and hopefully this would then lead to better convergence. The test was run in the same way as previous ones where each model was trained for 500K samples. First an autoencoder, then the same inpainting architecture and paramaters swapping only the L1 and L2 metrics. The results show a small but consistent difference in the output quality of the two networks where L1 distance scores better.

Figure 5.11: Comparing L1 vs L2 loss for the inpainting objective



(a) L1 vs L2 latent model score. (Higher is better)



(b) L1 vs L2 latent SSIM defect score (Higher is better)

Chapter 6

Hardware Study

6.1 The problem of edge DNN's

A big problem facing Deep Neural Networks (DNN) is the immense hardware, memory, and power requirements both during training but also to a smaller degree for inference. Take this specific project for example. To achieve good results we reported that some of our models would have to be trained on around 20 epochs which equates to around 24 million training samples. This took about 3 hours on a modern RTX-4090 GPU with 24GB of VRAM. The GPU alone consumed about 250W while training, as reported by the 'nvidia-smi' tool. In fact the issue of deploying the currently largest crop of models called Large Language Models (LLM) is a pressing issue [40]. Just loading the GPT-175, which consists of 175 billion parameters takes 350GB of RAM requiring at least five Nvidia A100 (80GB VRAM) GPU's [40]. There is also oppurtunity with regard to the fact that most DNN's are somewhat resilient to errors resulted from reductions in precision of the computation. In this way we can leverage techniques like approximate computing in order to make gains in memory size, speed, and power [4].

Deploying most DNN's on smaller hardware is a challenge as most microcrontrollers have just 128KB of RAM and 1MB of flash [4]. This means direct edge deployments of most DNN's on microcontrollers are out of the question. Another important subject is the power requirements. Its been reported that for hardware running DNN's, somewhere between $30 - 80\%$ of the system power draw comes from DRAM because memory movement dominates the system power usage [4]. Another large consumer of power is around the Multiply Accumulate (MAC) operations and data processing [4]. Couple this with the fact that microcrontrollers are orders of magnitude slower than popular GPU's and you soon exceed the power and timing constraints of most edge-device budgets. If the embedded device also ran on battery then continous use could prove prohibitive. In this chapter we aim to introduce some model compression schemes that seek to improve the models speed, memory footprint, and computational compatability with smaller edge devices.

6.2 Solutions to DNN implementation

6.2.1 Pruning

Unstructured Pruning

Pruning is a model compression scheme that reduces model size by removing weights, neurons or even full channels of features from the network based on their (lack of) activation towards a final network prediction. As outlined in [36, Sec. V, A] there are several pruning algorithms. The most common unstructured one is **weight pruning**. It follows a policy outlinining which layers to consider, and what metric to follow when pruning i.e the norm or level of the given layer. The policy can also include considerations of the energy and memory budgets. Weight pruning can give speed ups in inference by $\times 4$ and a reduction in memory usage by $\times 5 - 10$ [36]. Unstructured pruning schemes have the advantage of being easier to implement, and they also require no extra hardware considerations as they simply zero the chosen parts of the input.

Structured Pruning

Structured pruning exists to adress limitations of the unstructured kind. These include the fact that unstructured pruning create irregular sparsity in the weight matrices, and this can introduce variable processing and waste execution time [36, Sec. V, B]. Take for example a weight matrix that is pruned by some unstructured algorithm, there may now be several 'holes' in the matrix where zero-multiplication operations now has to happen every time that weight matrix is used. This leaves obvious room for improvement. The downside to structured sparsity algorithms is that they require special hardware and software support to enable good implementation. **Bayesian compression** is an example of a structured sparsity algorithm. It assumes that there exists hierarchical priors within the activation layers of CNN's, and uses variation inference to approximate a distribution of the weight posterior. This method can reduce parameter count by up to $\times 80$ in unpruned models [36]. Among others outlined are **SIMD-aware weight pruning** that was reported to provide up to $\times 3.5$ speedup and 88% reduction in model size [36].

6.2.2 Quantization

Quantization in machine learning is a model compression scheme which maps high precision 32-bit floating point model parameters down into smaller discrete integer spaces of 16/8/4/3/2 bit widths by applying a function that can be either linear or non-linear. Where the difference between the quantized value x_q and the original value x is the quantization error e_q 6.1 [7, p. 21]. Some of the advantages that come with quantization is the fact that model storage needs are smaller because data types are smaller, and computation is faster for the same reason as well as reducing memory bottlenecks from moving the now smaller parameters between non-volatile memory and low latency cache and/or registries during execution. Finally, quantization removes the need for a floating compute unit (FPU) [29, p. 2] which means model inference can run on more devices that don't have FPU's, or in cases where the device would have to software emulate the floating point operations, creating significant overhead.

$$e_q = x_q - x \tag{6.1}$$

Linear quantization refers to the fact that the quantization intervals are evenly spaced. An example of linear quantization is Fixed Point Encoding. It takes advantage of how floating point representations are stored to change the width of the fractional part of the number using a scaling factor, essentially moving the decimal point towards a wanted accuracy. This also allows for dynamic precision of different layers, keeping it high for sections integral to the network output and lowering precision more aggressively where appropriate [7, Sec. F, 0]. The gains of quantization are large, for example [7, Sec. F, 0] reports that a MAC (Multiply accumulate) performed on an 8-bit fixed point number consumes about $\times 20$ less energy than a MAC performed on a 32-bit floating point number. Moreover, the memory footprint is reported to shrink by$\times 4$ times in the same circumstance. One way to perform linear quantization is by using the Ristretto framework [7, Sec. F, 1]. It uses statistical analysis on the weights and activations to determine the scale factor and bit width, and is further fine-tuned using a re-training step. It was reported that the framework can convert large and complex models to 8-bit inference with less than 1% accuracy loss [7, Sec. F, 1]. One model also achived a $\times 7.6$ speedup compared to its floating point baseline [7, Sec. F, 1]. In fact what is known as quantization-aware training can achieve up to $\times 8$ times higher model compression at the same accuracy compared to whole network fixed point encoding without retraining [36, Sec. A, 1].

A non-linear quantization scheme is logarithmic quantization that approximates the dot product between the weights and activations deriving 6.2 a log scale 6.3. This method was reported to reduce the accuracy loss compared to linear quantization by $\times 10.3$ at a bit width of size 3 on a large model [7, Sec. F, 2].

$$w \cdot x = \sum_{i=0}^{N} w_i x_i \simeq \sum_{i=0}^{N} w_i 2^{\tilde{x}_i} = \sum_{i=0}^{N} w_i \ll \tilde{x}_i \tag{6.2}$$

$$\tilde{x}_i = Int(log_2(x_i)) \tag{6.3}$$

The final quantization scheme covered here is the most extreme called binarized quantization. This method maps the parameters to a single bit. This is reported to increase speeds by $\times 8.5 - 19$ and reduce memory by $\times 8$ [36, Sec. V, A].

6.3 Considering the Defect GAN for hardware

First its important to talk about the choice of activation function in relation to hardware deployment. ReLU is popular with respect to hardware deployment because it zeroes negative values from the layer and does no complex operations on the positive ones. This leaves more redundant activations as sparse-matrices and therefore skipped through hardware optimization techniques [4]. Another paper reports that the number of zeroed activations happening in a network, or simply termed network redundancy is somewhere between $50 - 70\%$ [7], which explains the large gains pruning algorithms can make in network compression. Once the pruning algorithm has finished we are left with another sparse matrix, which is converted using compression techniques that store only the non-null elements for significant memory savings [7].

The Defect GAN was trained with eventual hardware deployment in mind. For inference it contains only two parts, convolutional blocks and ReLU activation functions. This should make hardware deployment easier as there are no exotic layers or model quirks which would need custom hardware implementations or considerations. The complete model has 5.78Mln model parameters and a size of 22.1MB. With some of the reported methods introduced previosly we might be able to reduce the model size by as much as $\times 35 - 49$ times down to 450-650KB without losing accuracy [36]. In fact the compression could be even better when considering the precision requirements of the network. Since this is an inpainter and not a classifier we believe the tolerance for accuracy drop is actually larger as most research papers consider classifier s, which naturally have a low tolerance for error. The eventual drop in inpainting accuracy should manifest as deviating pixel values. People are naturally sensitive to some color spaces, but less sensitive to others [5, Sec. 6] and this could be further exploited to aggresively quantize corresponding feature maps. We believe there also exists room for significant pruning given the sameness of the network coupled with the fact that more complex inpainting networks exist with smaller parameter sizes [44]. This could mean that there exists large inefficiencies in the network that can be identified and removed with pruning while maintaining performance.

Chapter 7

Conclusion & Future Work

The goal of this project was originally to create an optimized machine learning algorithm for hardware, and then implement it as a proof-of-concept on an FPGA platform. Unfortunately, the scope of the work ended up being much larger than anticipated. It was assumed early on that because there already exist a fair number of inpainting networks capable of doing inpainting, or defacto correction of large missing portions in an image [44], that the development path for the machine learning part would be small. However, what we found was that most networks converged poorly on the objective and the eventual development of the network swallowed the majority of the alloted project time. The focus of this work instead shifted towards developing custom a network and objective functions to tackle the specific goal of correcting larger defect-cluster in image sensors. The hardware implementation was eventually relegated to a feasibility and theoretical study.

A custom model, custom loss functions and theory, as well as a versatile defect generation class have all been created specifically for this project. The model convincingly outperforms more conventional defect correction filter when testing 4.9. Achieving a +10dB increase in PSNR and +38% increase in SSIM score on the defective pixel. In addition, the model scores up to 98.96% on global SSIM, beating the filter by close to 7% and approaching the underlying image noise. The latent inpainting objective proved to be less of a factor than initially hypothesize, and some of the variation in score might actually be due to training parameters, suggesting that the objective is less useful in image application, or simply remains unoptimized for this task. The Dual encoder implementation performed better, suggesting it could warrant futher exploration. There is also a good argument based on the results for simply using a generative objective. This would reduce training and prototyping time significantly.

As far as future work is concerned there are a number of things I would have liked to study. The first one is an actual hardware implementation. In chapter 6 several claims were made about model size and expectations, but these are yet to be realized as anything more than a hypothesis. Within the model architecture I would want to further explore developing light-weight attention blocks in the network as texture in certain defect regions can vary and seem blurry. Further study of architectural or objective measures to increase the generated quality in these areas would be appropriate. As for the defects I would want to explore the use of color mapping and/or image conversion functions to better mimic the look of the pixel defects before they reach the inpainter. Right now the defects have a distinctly 'blocky' look to them, and the subject of studying non-linear defect masks is an active topic [8]. A conversion function after the defect is generated could potentially give them a 'smear' or more natural looking presentation closer to a real defect 3.10.

Bibliography

[1] Haris Iqbal (HarisIqbal88). *Plot Neural Net*. 2020. URL: https://github.com/HarisIqbal88/PlotNeuralNet (visited on 08/31/2023).

[2] Neptune AI. *L1 vs L2 regularization*. URL: https://neptune.ai/blog/fighting-overfitting-with-l1-or-l2-regularization (visited on 01/09/2023).

[3] Martin Arjovsky, Soumith Chintala, and Léon Bottou. *Wasserstein GAN*. 2017. arXiv: 1701.07875 [stat.ML].

[4] Giorgos Armeniakos et al. "Hardware Approximate Techniques for Deep Neural Network Accelerators: A Survey". In: *ACM Computing Surveys* 55.4 (Nov. 2022), pp. 1–36. DOI: 10.1145/3527156. URL: https://doi.org/10.1145%2F3527156.

[5] Jenny M Bosten. "Do You See What I See? Diversity in Human Color Perception". eng. In: *Annual review of vision science* 8.1 (2022), pp. 101–133. ISSN: 2374-4642.

[6] Alan C. Brooks, Xiaonan Zhao, and Thrasyvoulos N. Pappas. "Structural Similarity Quality Metrics in a Coding Context: Exploring the Space of Realistic Distortions". In: *IEEE Transactions on Image Processing* 17.8 (2008), pp. 1261–1273. DOI: 10.1109/TIP.2008.926161.

[7] Maurizio Capra et al. "Hardware and Software Optimizations for Accelerating Deep Neural Networks: Survey of Current Trends, Challenges, and the Road Ahead". In: *IEEE Access* 8 (2020), pp. 225134–225180. DOI: 10.1109/access.2020.3039858. URL: https://doi.org/10.1109%2Faccess.2020.3039858.

[8] Dongmin Cha and Daijin Kim. *DAM-GAN : Image Inpainting using Dynamic Attention Map based on Fake Texture Detection*. 2022. arXiv: 2204.09442 [cs.CV].

[9] Alexey Dosovitskiy et al. *An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale*. 2021. arXiv: 2010.11929 [cs.CV].

[10] Fernando A. Fardo et al. *A Formal Evaluation of PSNR as Quality Measurement Parameter for Image Segmentation Algorithms*. 2016. arXiv: 1605.07116 [cs.CV].

[11] 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*. Ed. by Yee Whye Teh and Mike Titterington. Vol. 9. Proceedings of Machine Learning Research. Chia Laguna Resort, Sardinia, Italy: PMLR, 13–15 May 2010, pp. 249–256. URL: https://proceedings.mlr.press/v9/glorot10a.html.

[12] Prashant Gohel, Priyanka Singh, and Manoranjan Mohanty. *Explainable AI: current status and future directions*. 2021. arXiv: 2107.07045 [cs.LG].

[13] Ian J. Goodfellow et al. *Generative Adversarial Networks*. 2014. arXiv: 1406.2661 [stat.ML].

[14] Ishaan Gulrajani et al. *Improved Training of Wasserstein GANs*. 2017. arXiv: 1704.00028 [cs.LG].

[15] Kaiming He et al. *Deep Residual Learning for Image Recognition*. 2015. arXiv: 1512.03385 [cs.CV].

[16] Kaiming He et al. *Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification*. 2015. arXiv: 1502.01852 [cs.CV].

[17] Martin Heusel et al. *GANs Trained by a Two Time-Scale Update Rule Converge to a Local Nash Equilibrium*. 2018. arXiv: 1706.08500 [cs.LG].

[18] Geoffrey E. Hinton et al. *Improving neural networks by preventing co-adaptation of feature detectors*. 2012. arXiv: 1207.0580 [cs.NE].

[19] Sergey Ioffe and Christian Szegedy. *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*. 2015. arXiv: 1502.03167 [cs.LG].

[20] Phillip Isola et al. *Image-to-Image Translation with Conditional Adversarial Networks*. 2018. arXiv: 1611.07004 [cs.CV].

[21] Takeshi Kadono et al. "Reduction of White Spot Defects in CMOS Image Sensors Fabricated Using Epitaxial Silicon Wafer with Proximity Gettering Sinks by CH2P Molecular Ion Implantation". In: *Sensors* 22.21 (2022). ISSN: 1424-8220. DOI: 10.3390/s22218258. URL: https://www.mdpi.com/1424-8220/22/21/8258.

[22] Jaehyeon Kim, Jungil Kong, and Juhee Son. *Conditional Variational Autoencoder with Adversarial Learning for End-to-End Text-to-Speech*. 2021. arXiv: 2106.06103 [cs.SD].

[23] Vera de Kok. *A photograph taken with a damaged image sensor*. Available at https://commons.wikimedia.org/w/index.php?curid=20051738. (Visited on 08/16/2023).

[24] Guilin Liu et al. *Image Inpainting for Irregular Holes Using Partial Convolutions*. 2018. arXiv: 1804.07723 [cs.CV].

[25] Ziwei Liu et al. "Deep Learning Face Attributes in the Wild". In: *Proceedings of International Conference on Computer Vision (ICCV)*. Dec. 2015.

[26] Stephen Marsland. *MACHINE LEARNING, An Algorithmic Perspective*. Machine Learning and Pattern Recognition Series. Chapman & Hall/CRC, 2015. ISBN: 13:978-1-4665-8328-3.

[27] Takeru Miyato et al. *Spectral Normalization for Generative Adversarial Networks*. 2018. arXiv: 1802.05957 [cs.LG].

[28] Michael A. Nielsen. *Neural Networks and Deep Learning*. Determination Press, 2015.

[29] Pierre-Emmanuel Novac et al. "Quantization and Deployment of Deep Neural Networks on Microcontrollers". In: *Sensors* 21.9 (Apr. 2021), p. 2984. DOI: 10.3390/s21092984. URL: https://doi.org/10.3390%2Fs21092984.

[30] Pytorch. *A gentle introduction to torch.autograd*. URL: https://pytorch.org/tutorials/beginner/blitz/autograd_tutorial.html (visited on 08/01/2023).

[31] Pytorch. *Adam*. URL: https://pytorch.org/docs/stable/generated/torch.optim.Adam.html (visited on 08/11/2023).

[32] Pytorch. *Conv2d documentation*. URL: https://pytorch.org/docs/stable/generated/torch.nn.Conv2d.html (visited on 08/01/2023).

[33] Pytorch. *L1loss documentation*. URL: https://pytorch.org/docs/stable/generated/torch.nn.L1Loss.html (visited on 08/08/2023).

[34] Pytorch. *MSEloss documentation*. URL: https://pytorch.org/docs/stable/generated/torch.nn.MSELoss.html (visited on 08/08/2023).

[35] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. *U-Net: Convolutional Networks for Biomedical Image Segmentation*. 2015. arXiv: 1505.04597 [cs.CV].

[36] Swapnil Sayan Saha, Sandeep Singh Sandha, and Mani Srivastava. "Machine Learning for Microcontroller-Class Hardware: A Review". In: *IEEE Sensors Journal* 22.22 (Nov. 2022), pp. 21362–21390. DOI: 10.1109/jsen.2022.3210773. URL: https://doi.org/10.1109%2Fjsen.2022.3210773.

[37] Ghislain Takam Tchendjou and Emmanuel Simeu. "Detection, Location and Concealment of Defective Pixels in Image Sensors". In: *IEEE Transactions on Emerging Topics in Computing* 9.2 (2021), pp. 664–679. DOI: 10.1109/TETC.2020.2976807.

[38] Xiaolong Wang et al. *Non-local Neural Networks*. 2018. arXiv: 1711.07971 [cs.CV].

[39] Zhou Wang et al. "Image quality assessment: from error visibility to structural similarity". In: *IEEE Transactions on Image Processing* 13.4 (2004), pp. 600–612. DOI: 10.1109/TIP.2003.819861.

[40] Guangxuan Xiao et al. *SmoothQuant: Accurate and Efficient Post-Training Quantization for Large Language Models*. 2023. arXiv: 2211.10438 [cs.CL].

[41] Shuang Xie and Albert J. P. Theuwissen. “On-Chip Smart Temperature Sensors for Dark Current Compensation in CMOS Image Sensors”. In: *IEEE Sensors Journal* 19.18 (2019), pp. 7849–7860. DOI: 10.1109/JSEN.2019.2919655.

[42] Jie Yang, Zhiquan Qi, and Yong Shi. *Learning to Incorporate Structure Knowledge for Image Inpainting*. 2020. arXiv: 2002.04170 [cs.CV].

[43] Minghui YANG et al. “Identification and replacement of defective pixel based on Matlab for IR sensor”. In: *Frontiers of Optoelectronics* 4.4, 434 (2011), p. 434. DOI: 10.1007/s12200-011-0177-2. URL: https://journal.hep.com.cn/foe/EN/abstract/article_2138.shtml.

[44] Jiahui Yu et al. *Generative Image Inpainting with Contextual Attention*. 2018. arXiv: 1801.07892 [cs.CV].

[45] Aston Zhang et al. “Dive into Deep Learning”. In: *arXiv preprint arXiv:2106.11342* (2021).

[46] Han Zhang et al. *Self-Attention Generative Adversarial Networks*. 2019. arXiv: 1805.08318 [stat.ML].