Norwegian University of
Science and Technology

Applying generative adversarial networks for anomaly detection in hyperspectral remote sensing imagery

Aksel Wilhelm Wold Eide

Master of Science in Physics and Mathematics
Submission date: March 2018
Supervisor: Pål Erik Goa, IFY
Co-supervisor: Eilif Solberg, FFI

Norwegian University of Science and Technology
Department of Physics

Contents

1 Problem description and abstract **3**
1.1 Problem description . . . 3
1.2 Abstract . . . 3
1.3 Sammendrag . . . 3

2 Introduction **4**
2.1 Overview . . . 4
2.2 Anomaly detection . . . 4
2.3 Hyperspectral imaging . . . 5
2.4 Machine learning . . . 6
Anomaly detection and binary classification . . . 7
2.5 Generative adversarial networks . . . 7
2.6 Hyperspectral remote sensing imagery . . . 7
2.7 Previous work . . . 8
2.8 Synthesis . . . 8

3 Theory **10**
3.1 Machine learning fundamentals . . . 10
Problem description . . . 10
Accuracy . . . 10
Validation and overfitting . . . 11
3.2 Neural network . . . 13
Artificial neuron . . . 13
Network architecture . . . 14
Gradient descent and backpropagation . . . 15
Deep learning . . . 17
Output layer encoding . . . 18
ROC and AUC . . . 19
Semi-log ROC plot and logAUC . . . 20
3.3 Convolutional neural networks . . . 22
Convolutional layers . . . 22
Padding and stride . . . 22
Rectified linear units . . . 23
Shortcut . . . 24
3.4 GAN . . . 25
Wasserstein GAN with gradient penalty . . . 25
3.5 Anomaly detection with generative adversarial networks . . . 26
Discriminator based anomaly detection . . . 26
Generator based anomaly detection . . . 27
Reconstruction cost . . . 27

4 Materials and methods **29**
4.1 Hardware and software . . . 29
4.2 Data material . . . 29
4.3 GAN implementation . . . 29
Discriminator and generator . . . 30

Recreation cost 31
4.4 Experiments 32

5 Results and discussion **33**
5.1 Single pixel baseline results 33
5.2 Single pixel results for model variations 37
5.3 Discriminator based anomaly detection 40
5.4 Single pixel results with time intensive reconstruction 43
5.5 Larger window sizes 45

6 Conclusion and further work **47**

1 Problem description and abstract

1.1 Problem description

The aim of the project is to test the feasibility of using a machine learning approach called generative adversarial networks (GAN) to detect anomalies in remote sensing imagery. The project will involve reviewing the current literature on GANs and implementing a tried and tested architecture. The GAN will be applied to a hyperspectral dataset provided by the supervisor, which will require adapting the network's architecture, tuning its parameters and testing a variety of different optimizations. The project will evaluate the suitability of GANs for this kind of problem and discuss the significance of different design possibilities and parameter values.

1.2 Abstract

Automatic anomaly detection has previously been implemented on hyperspectral images by use of different statistical methods with good results. We apply a machine learning method using generative adversarial networks (GAN) to a hyperspectral remote sensing image provided by the Norwegian Defence Research Establishment (FFI), using the Wasserstein formulation with gradient penalty. We train a generator network which outputs generated 32×32 pixel image windows with good visual quality, but are only able to make anomaly detection work for 1×1 pixel windows. We show results for both a discriminator based anomaly detection method, which only works properly with very carefully tuned hyperparameters, and for a generator based anomaly detection method making use of a recreation cost, which is robust and achieves useful results that are not much worse than more highly developed methods. Our work demonstrates the possibilities of applying GANs for unsupervised anomaly detection and explores the challenges that this method presents.

1.3 Sammendrag

Automatisk anomalideteksjon har tidligere blitt gjort i hyperspektrale bilder ved bruk av ulike statiske metoder med suksess. Vi anvender en maskinlæringsmetode som nyttegjør seg av generative adversariale nettverk (GAN) på et hyperspektralt fjernmålingsbilde produsert av Forsvarets forskningsinstitutt (FFI), med bruk av Wasserstein-formuleringen og med et kostnadsledd som fungerer som en gradientbetingelse. Vi trener et generatornettverk som produserer kunstige 32×32-pikslers bildevinduer med god visuell kvalitet, men får bare anomalideteksjon til å fungere godt for 1×1-piksel bildevinduer. Vi presenterer resultater både for en diskriminatorbasert anomalideteksjonsmetode, som kun fungerer med nøye tilpassede hyperparemetere, og for en genratorbasert anomalideteksjonsmetode som gjør bruk av en rekonstruksjonskostnad, som er robust og oppnår gode resultater som ikke er stort verre enn mer utviklede metoder. Arbeidet vårt demonstrerer muligheten for å anvende GAN til å gjøre anomalideteksjon uten bruk av noen fasit i læringsprosessen og utforsker utfordringene ved bruk denne metoden.[1]

[1] This is the Norwegian-language version of the abstract.

2 Introduction

2.1 Overview

This project makes use of a specific form of machine learning, namely generative adversarial neural networks, to detect anomalous objects in hyperspectral surveillance images. In this introduction, we will give a brief explanation of each of the involved topics in relatively simple terms, before finally describing how they all fit together in our particular application.

2.2 Anomaly detection

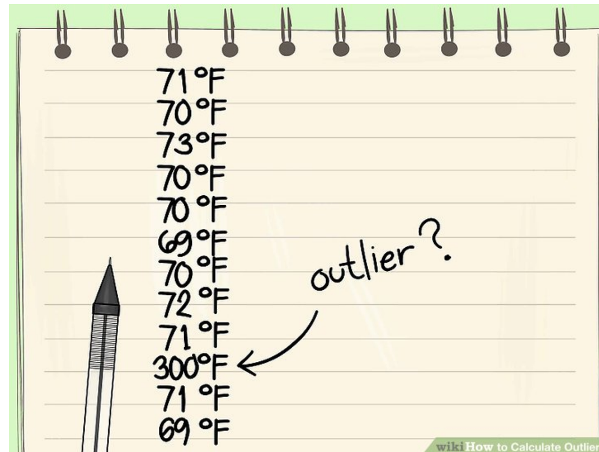


Figure 1: Faux experimental temperature measurements, with a single anomalous measurement flagged as a potential outlier. Reproduced from wikiHow under the Attribution-Noncommercial-Share Alike 3.0 Creative Commons License

It is difficult to give a rigorous definition of anomalies, which are characterized by failing to match with either prior expectations or patterns in the dataset. A familiar example is the process of looking at measurements from a laboratory experiment, spotting an outlier and excluding it when calculating a trend line. A combination of statistical intuition and scientific understanding indicates whether the measured value is physically reasonable or if seems to be due to human error, equipment malfunction, etcetera.

In the simplest cases, such as in figure 1, we can rely on anomalies being obvious — *I know it when I see it.* More generally, we need a framework for distinguishing between data from the tail of the statistical distribution and true anomalies. For the lifespan of humans, 75 years is ordinary and 150 years is unheard of, but there is no sharp threshold anywhere in between that delineates exception values from anomalies. A more formal definition is useful for dealing with edge cases and in domains where we lack intuition. It is also necessary in order to automate anomaly detection, seeing as computers have no intuition to fall back on.

A familiar application of anomaly detection is the system of p-value testing used in all forms of scientific research. It consists of defining a null hypothesis which imposes a statistical expectation on our data, for instance that there is no difference in the mean value of X between groups A and B. Using a statistical test, we can calculate how likely our results are, given that they are sampled from a distribution where our null hypothesis holds. If the observed values of X are highly improbable, using for instance the standard threshold of $p < 0.05$, we conclude that the results are incompatible with our null hypothesis. In p-value testing, we consider this

a refutation of the null hypothesis instead of concluding that the data is anomalous, but the formalism exactly the same.

In its simplest form, anomaly detection is a matter of straightforward statistics, but real world problems are full of complications. One application of anomaly detection is for monitoring bank transactions and trying to pick out illegitimate transactions, such as someone using a stolen credit card. What characterizes the distribution of legitimate bank transactions? We have to deal with large amounts of high dimensional data as well as new credit card holders and sellers that we have little data on. In practice, the approach must be custom tailored to the domain we are dealing with.

2.3 Hyperspectral imaging

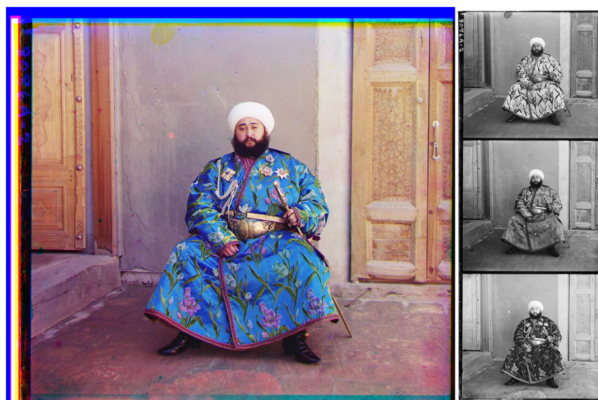


Figure 2: Left: Color photography of Mohammed Alim Khan. Right: Each of the three channels in separate images, with signal intensities represented in monochrome. Reproduced from Wikipedia. The photography, by Sergei Mikhailovich Prokudin-Gorskii, is in the public domain.

The earliest forms of photography were monochrome, producing an image representing an aggregate intensity of visible light. Digital color photography is a more sophisticated technique, which uses separate channels for red, green and blue light and is designed to capture a scene as it would appear to the human eye. This separation into channels stores more information in form of a higher spectral resolution. The additional colors generally make it easier to interpret the image.

Hyperspectral imaging is an technique that does not merely try to mimic natural vision. It goes beyond the limits of the human eye in two important ways: by increasing the spectral resolution such that we have more than just three channels in the visible spectrum, and by also measuring signals outside of the visible spectrum. The first is motivated by how different signal response curves can have the same signature when reduced to three channels. Most color displays make use of this effect, representing a mixed red-green signal the same way as a in-between, single frequency yellow signal. While these appear equivalent to the human eye, they are distinct with a higher spectral resolution. This is particularly relevant for surveillance images, since camouflage is usually custom designed to resemble the natural surroundings as viewed by the *human* eye. A familiar example of the second is the use of infrared vision, that can be used to pick out relatively warmer objects because of differences in thermal radiation. While this is most useful when there is no visible light, the infrared signal is also present in bright environments.

Hyperspectral images encode much more information than normal images, which in principle allow us to discriminate better between objects in the scene. A key challenge is to find a way make use of this information. Hyperspectral images are not themselves human readable — we must choose a visual representation to convert them into actual pictures, which re-introduces the three channel limitation of human vision. A straightforward solution is to produce a monochrome image for each spectral band in the image, as done for a color image in figure 2, but this makes it difficult to synthesize the information from different bands. Computer analysis is an obvious way to sidestep this problem, but computer vision comes with its own set of challenges.

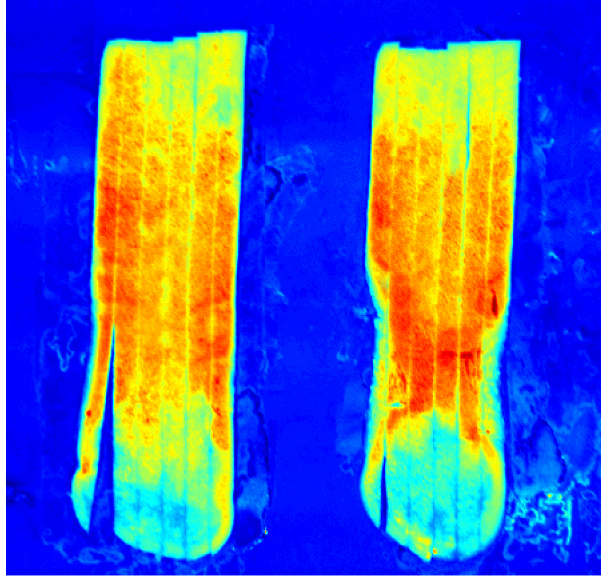


Figure 3: Hyperspectral image of potato strips showing defects not visible in the spectra visible to the human eye. Reproduced from Wikipedia under the Creative Commons Attribution-Share Alike 3.0 Unported license.

2.4 Machine learning

Machine learning is field of study that combines statistical methods and computer programming in order to solve a variety of different problems. The general aim is to automate the task of finding patterns in data, ultimately trying to arrive at a way to map input data to a desired output. While this description is so general that it appears meaningless, this is a reflection of the incredibly wide range of applications of machine learning, which include interpreting handwritten text, playing the abstract board game Go and modifying pictures of faces to make them express different emotions.

A machine learning problem requires a dataset and a problem to be solved. A typical introductory problem is trying to distinguish between handwritten digits in pixelated images. Before we can get to the automated parts of the process, we must choose a machine learning algorithm. In practice, established and pre-packaged algorithms such as convolutional neural networks and random forests are most used. Properly pre-processing the data and manually tuning a number of hyper-parameters is often essential for good results.

The final step, which characterizes machine learning, involves the algorithm automatically tuning its own internal parameters, attempting to make its own mapping conform with the

patterns in the dataset. Ideally, this results in a model that we can apply to real world problems, such as automatically interpreting the letters in a scanned book.

Anomaly detection and binary classification

Superficially, anomaly detection can seem like an ordinary machine learning problem, where we treat normal and anomalous data as two different classes or clusters. In the limiting case where we have plentiful data and are able to model the anomalous class, this holds true. In practice, however, there are two key characteristics that make anomalies special. First, anomalous data is assumed to be sparse, which makes it difficult to learn how it is distributed. Second, the anomalous data is expected to be highly irregular, because *anomaly* is a catch-all term for the different unexpected things that can happen in a system and lead to abnormal data. This motivates a special approach, where we focus on finding a model for the normal data and treat whatever fails to conform to that expectation as anomalies, i.e. a negative definition.

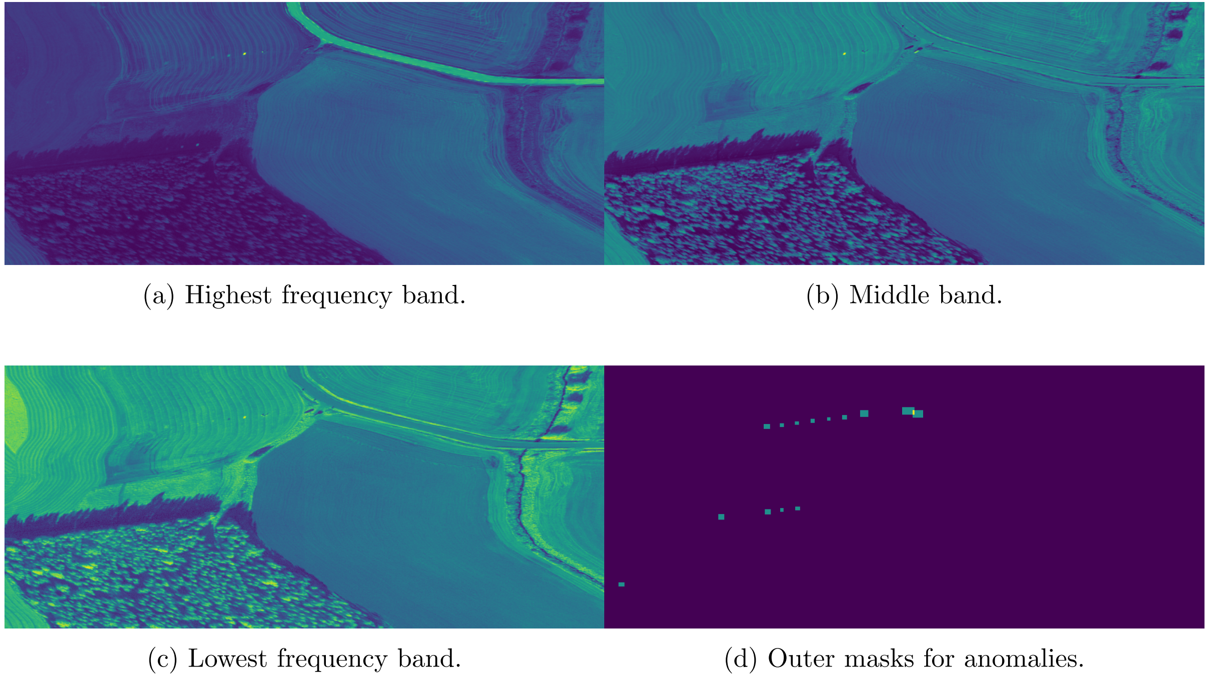
2.5 Generative adversarial networks

A generative adversarial network is a machine learning approach that makes use of two different neural networks. One is tasked with generating data that mimics real data. The second tries to discriminate between synthetic and real data. The two networks are in an adversarial relationship - the generator is trained to generate data that is plausible enough to fool the discriminator, while the discriminator is trained to recognize the ways in which generated data differs from real data. Generative adversarial networks have a number of interesting properties. They allows us to generate synthetic data in domains where this is otherwise difficult, such as nearly photorealistic images. Fiddling with the inputs given to the generator also makes it possible to visualize how the neural network models its domain, which is particularly interesting because neural networks are often criticized for being functional black boxes.

Furthermore, a well-functioning generator-discriminator-network is in practice a model of how real data is distributed - both the generator and discriminator must implicitly learn the distribution of real data in order to achieve their goals. When the discriminator has been trained using non-anomalous data as real data and the generator is stressing it properly, it should be able to recognize real anomalous data in the same way that it recognizes synthetic data, in practice functioning as an anomaly detector.

2.6 Hyperspectral remote sensing imagery

The dataset is a hyperspectral surveillance image, shot by airplane for purposes of internal research and development. Fourteen objects to be detected have been placed at the scene. The image consists of 1600×700 pixels, and each pixel measures signal intensities for 80 different frequency bands, and also exists in a pre-processed form where the spectral resolution is reduced to 20 bands in order to achieve less noise. In addition, an inner and an outer mask is provided, which mark the fourteen objects, with the inner mask excluding all non-object pixels and the outer mask including all object pixels. Because of the relatively small size of the objects, measured in pixels, the edges make up a substantial part of the object pixels, and the pixel- and object boundaries do not generally align.



(a) Highest frequency band.

(b) Middle band.

(c) Lowest frequency band.

(d) Outer masks for anomalies.

Figure 4: Visualization of dataset. Subfigures (a-c) show select single-band intensities of the hyperspectral image. Subfigure (d) shows the outer masks for anomalies in the image, i.e. regions that contain the anomalies in their entirety. Various features are easily identifiable in the single-band images, such as the road in the top right corner (most visible in (a)) and the cluster of trees in the lower right corner (most identifiable in (c)). Some of the anomalous regions in the mask stand out clearly in all three bands, others do not seem visible in any of them.

2.7 Previous work

The same dataset has been studied extensively at the Norwegian Defence Reserach Establishment. The best published results for anomaly detection are based on a multinormal mixture model. In the simplest possible terms, this is a statistical method that describes each individual pixel in the scene as drawn from a distribution that is Gaussian, but with multiple modes — this means that there are multiple peaks in the distribution, such as we get for the body height of humans, which is commonly represented with a bimodal distribution in order to capture the different mean values and deviations for the two sexes. In the surveillance image, the background is composed pixels with signals that fall into different clusters, such as asphalt, fields and trees, which motivates a multimodal model. Anomaly detection is done by finding the distance of individual pixels from this distribution, where a greater distance indicates a more abnormal pixel and a greater likelihood of characterizing an anomaly.

Results based on this model are good and mostly struggle with discrimination in dark parts of the image.

2.8 Synthesis

A brief outline of the anomaly detection procedure is as follows:

- Adapt a residual neural network implementation to work with the hyperspectral surveil-

lance image

- Integrate this neural network in a generative adversarial networks-framework
- Train the network on the dataset such that the networks learn the data distribution
  - Generator network is optimized for producing outputs that the discriminator cannot tell apart from real data
  - Discriminator network is optimized for telling the difference between generated and real data
- Evaluate the training process and compare the generator's outputs with the real image
- Use the trained networks to detect anomalies in the surveillance image
  - Evaluate the results
  - Compare results with previous work

3 Theory

3.1 Machine learning fundamentals

A more complete treatment of machine learning as a general topic is outside of the scope of this project. We will instead cover the most fundamental concepts and definition as briefly as possible, and progressively increase the level of detail as we get closer to the directly relevant subtopics such as convolutional layers and generative adversarial networks. Throughout this section, we will use the task of recognizing hand-written digits in the MNIST database[15] as a go-to example. This is a standard introductory problem that is reasonably accessible to a novice, while still rich enough to still be studied seriously.

Problem description

Machine learning problems can be defined in terms of finding a good mapping from an input domain to an output domain. For the MNIST problem, this means that we try to find a way to map the input, 28×28 pixels grayscale images, to the correct output, namely the digit $0 - 9$ that the image depicts, which in practice means that we make a system capable of interpreting handwritten digits as a human would.

We can consider each image (each *sample* in the general case) to be a point in a high-dimensional space. In the MNIST dataset, the images are characterized by the intensity values for each of its $28 * 28 = 784$ pixels, such that it can be characterized by a 784-dimensional vector, where each dimension, representing the intensity at a particular pixel location, is called a feature. When the output space is a discrete set of categorical variables, such as for the ten different digits, we refer to the different possible outputs as different *classes* and the machine learning problem as a *classification problem*. There are a multitude of other types of machine learning problems, where we for instance do not have any prior knowledge of what the classes of the output space are and determining these is part of the problem (clustering), or where the output is a continuous variable (regression)[17, Chapter 1].

In the most typical and straightforward case, we have a complete dataset available upfront (offline learning) and can make use of a large number of samples with labeled output values during training (supervised learning). In the MNIST case, this means that we know the correct digit class (ground truth) for the images. This way, the training process can make use of the difference between ground truth and the learning algorithm's current outputs to iteratively steer it towards a representation where the outputs match with ground truth. Ideally, this means that the algorithm has *learned* the patterns and relationships in the dataset.

Accuracy

A very basic measure of the quality of a machine learning algorithm is its accuracy, which is the fraction of all samples that are classified correctly:

$$\text{accuracy} = \frac{\text{number of correctly classified samples}}{\text{total number of samples}} \tag{1}$$

In the case of binary classification, where there are only two possible classes, we can label these as *positive* and *negative*. This lets us distinguish between two different kinds of errors: false positives, samples that are negative in ground truth but incorrectly classified as positive; and false negatives, samples that are positive in ground truth, but classified as negative. These are

contrasted against the true positives and true negatives, which are correctly classified positive and negative samples respectively[6]. In these terms, the definition of accuracy becomes:

$$\text{accuracy} = \frac{\text{true positives} + \text{true negatives}}{\text{true positives} + \text{true negatives} + \text{false positives} + \text{false negatives}} \tag{2}$$

While simple and intuitively reasonable, there are a number of problems with optimizing exclusively for high accuracy[7]. This is most easily demonstrated by considering a dataset with highly imbalanced classes. For the purposes of anomaly detection, we expect the vast majority of samples to be non-anomalous (negative), with only a tiny fraction of anomalies. Supposing that the frequency of anomalies is one per thousand, we can find the accuracies of two hypothetical pseudoclassifiers, one that always returns positive and one that always returns negative, without taking into account the sample's feature vector. Making use of equation 2 and denoting the total number of samples N, we get:

$$\text{accuracy}_{\text{all_pos}} = \frac{N * 10^{-3} + 0}{N} = 0.001 \tag{3}$$

$$\text{accuracy}_{\text{all_neg}} = \frac{0 + N * (1 - 10^{-3})}{N} = 0.999 \tag{4}$$

Neither of these classifiers have any ability to distinguish between positive and negative samples and only reflect the different base rates of negative and positive samples. We can also imagine a classifier that detects all the anomalies while only misclassifying one in a hundred of the non-anomalies, which would be an impressive result in some domains. It would have have an accuracy as follows:

$$\text{accuracy}_{\text{useful}} = \frac{N * 10^{-3} + N * (1 - 10^{-2})}{N} = 0.991 \tag{5}$$

Not only does the accuracy metric have an unfortunate base rate dependency that makes its numeric value meaningless in isolation — even for a given dataset, an intuitively stronger classifier can have a worse accuracy. This is a result of *accuracy* weighing anomalous and non-anomalous samples equally, despite the differences in base rate, which in turn rewards a classifier for erring in the direction of the majority class. For the imbalanced dataset we have considered, however, where the anomaly frequency is one per thousand, the expected *exchange rate* between true positives and true negatives are 1 : 1000 when blindly guessing. Any classifier doing better than this has an information content that the accuracy metric fails to appreciate. In section 3.2 we will introduce a more robust metric, the receiver operator characteristic, which avoids these problems.

Validation and overfitting

From working with regression or Fourier series, it is familiar that increasingly complex models with more parameters generally fit better and better with a set of points, regardless of whether the models actually match with the underlying distribution that has produced the points. It it inadvisable to optimize exclusively for a low mean square error or a high value of R^2 — in practice we want to strike a balance between low error and low complexity.

In machine learning applications, we are faced with a similar problem. Most machine learning models, particularly neural nets, have a large enough number of parameters that they can in principle memorize the dataset, an effect called overfitting. This is troublesome, because

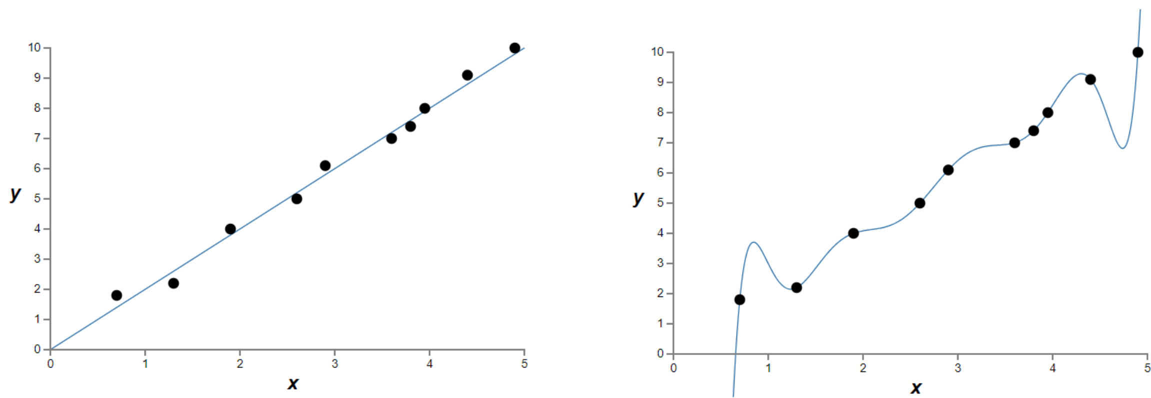


Figure 5: A first-order (left) and ninth-order (right) polynomial fit to the same set of points. The straight line fit on the left has a slightly greater mean square error, due to the greater distance between the line and the points, but in an intuitive sense seems to match the distribution better than the much more complex curve on the right. The curve on the right, with a very small mean square error, but also with shape that does not seem to match with the distribution the points are drawn from, is a very bare-bones visualization of overfitting. Reproduced from Nielsen's book, "Neural Networks and Deep Learning" under the Creative Commons Attribution-NonCommercial 3.0 Unported License.

overfitted models perform exceptionally well during training and most training procedures will prefer to overfit a model given the chance, rather than to learn properly. When training a model, we would expect it to start out with no understanding of the problem and doing no better than chance. Gradually, during training, we hope that it learns meaningful patterns and relationships that improves its ability to classify, and by *meaningful* we mean that this learning is robust and will transfer to other datasets with the same classification problem. However, we also run a risk of the model beginning to pick up on patterns that are specific to the dataset we are training it on, which will improve its results on the data it has memorized during training, but otherwise degrade its performance.

For instance the third image, first row in figure 6 is a sample for which we would expect a good model to output a class that does *not* match ground truth in the dataset, because it is visually much closer to a nine than an eight. However, an overfitted model can memorize peculiarities about this image (and the other images) to improve its results during training, such as increasingly classifying digits with wide strokes as nines.

The most fundamental safeguard against overfitting is to split a dataset into two parts — a training set that we use to train the model, and a validation set. After training, we can run the model on the validation set. Because we have never given the model a chance to memorize the correct classifcation for the samples in the validation set, this serves as a test of whether the model has succeeded in learning relationships that generalize to unseen data. When training is going well, the model will be improving its performance, both for training and validation data. In cases of overtraining, training performance will keep improving, while validation performance stabilizes or even declines[19, Chapter 3].

Unfortunately, the training-validation scheme does not actually prevent the learning process from overfitting. It just gives us an indication of whether it is happening. A number of techniques for reducing the impact of overfitting, even when training complicated models for a large number of iterations, have been developed, such as noising, regularization, batch normalization

and dropout[25].

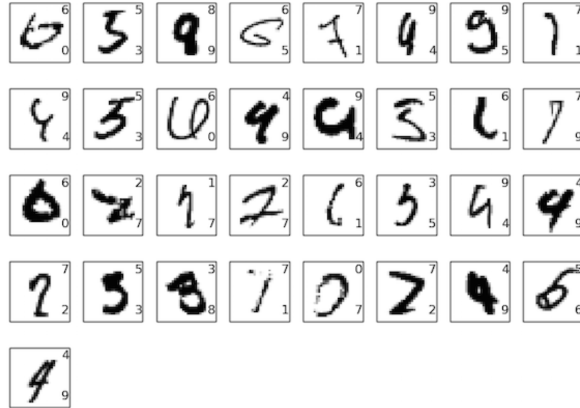


Figure 6: A selection of misclassified images from the MNIST-database, with the correct class in the upper-right corner of each individual image and a neural network classifier's output in the bottom-right corner. Note the sixth image, first row and first image, fourth row, where we can see that the images are ambiguous and that the classifier's choice seem reasonable to a human. Note also third image, first row, where the classifier's output seems more reasonable that ground truth. Reproduced from Nielsen's book, "Neural Networks and Deep Learning" under the Creative Commons Attribution-NonCommercial 3.0 Unported License.

3.2 Neural network

Artificial neuron

The fundamental building block of neural networks is the artificial neuron. It was conceived as a computational model of the biological neurons in the brain. In mathematical terms, a neuron is a simple multivariable function:

$$y = f(b + \sum_{j=0}^{m} w_j x_j) \tag{6}$$

The neuron maps its m input values x_j to a single output y. The parameters w_j and b are specific to the individual neuron and are called the weights and the bias respectively. The enveloping function f is called an activation or a transfer function, and is usually a fairly simple function such as tanh or rectified linear unit. Activation functions are primarily used to limit the range of the neuron's output and to make the partial derivatives $\frac{\partial y}{\partial x_j}$ are convenient to work with during backpropogation.

It should be emphasized that the individual neuron is mathematically simple — essentially a linear equation in m variables. The complexity of neural networks mostly have to do with the overall architecture of the network, its size and the tuning of the neuron parameters. A simple network with roughly a thousand neurons can be trained to achieve a 95% accurate classification of handwritten digits[19, Chapter 1].

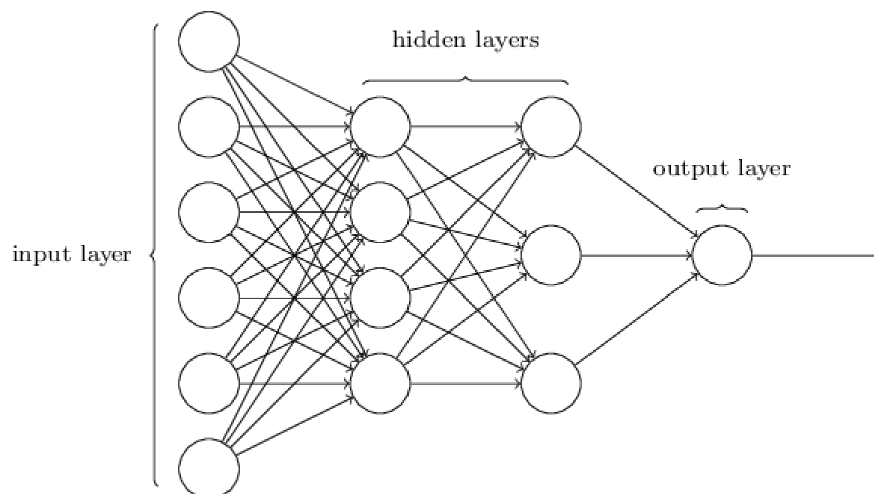


Figure 7: A toy neural network with two hidden layers and fully connected layers. Circles represent neurons. Signal propagates from neuron to neuron in the direction of the arrows. Reproduced from Nielsen's book, "Neural Networks and Deep Learning" under the Creative Commons Attribution-NonCommercial 3.0 Unported License.

Network architecture

As a matter of convenience, neural networks are are generally structured as a series of layers, such that the inputs to a neuron x_j are a subset of the outputs from neurons in the previous layer. The first such layer is called the input layer, which is responsible for feeding the input signal into the network. The individual neurons in this layer are associated with a feature in the input data, such as the intensity value at a particular pixel position, and for each sample they are simply assigned the values in the data. The final layer in a neural network is the output layer. The values of the neurons are the network's suggested outputs for the sample. In between these two layers, there can be any number of layers where the network performs its intermediate calculations[19, Chapter 1]. These are called hidden layers because they are internal to the network and can be abstracted away.

Notably, given values for its input layer, the activations of each subsequent layer can be calculated by applying equation 6 for each individual neuron. This process of chaining the input signal throughout the network is called forward propagation and allows us to calculate the values of the output neurons in the final layer. For a given network with fixed weights and biases, the output values depend only on the inputs.

The design of the input and output layers are mostly determined by the problem we are studying, in that the neurons in these layers have to represent the data we are working with. The only real choice is how to encode the data. As for the hidden layers, we are only limited by imagination, computational feasibility and the practical results. In principle, we can use any number of hidden layers, and for every layer there is a choice of how many neurons to use and how they should be connected. Early work with neural networks used only a single hidden layer, and theoretical considerations show that simple, single-layer networks can represent any function with arbitrary precision[19, Chapter 4]. Some modern, high-performing deep learning architectures such as ResNet-152 make use of more than a hundred layers, with roughly a hundred million parameters (neuron weights). There is considerable variation in classification accuracy between different network architectures, even for networks with the same computa-

tional complexity[5]. The representational power of a network is not the relevant limitation, but rather its ability to learn to a good representation during training.

Gradient descent and backpropagation

Training is the process by which a network's parameters — the weights and biases of the individual neurons — are adjusted, such that the network produces the correct output values. The notion of a *desired output* is formalized by means of a cost function, such as the mean squared error[19, Chapter 1]:

$$C(w, b) = \frac{1}{2n} \left\| \sum_x y(x) - a(x, w, b) \right\|^2 \tag{7}$$

Here, the cost C is half the average of the square error for each individual sample x. y and a denote the correct output and the neural network output for the given sample respectively. Seeing as the output a depends on on the set of all neuron weights and biases, w and b, the cost is a function of these network parameters, as well as the values of x and y, which are given by the dataset and not subject to optimization. Training the network is equivalent to minimizing C with respect to w and b, where the ideal network that perfectly matches the correct values achieves a minimal cost of 0. Even in a toy problem scenario with a small neural network, this optimization problem in thousands of dimensions is analytically intractable. The standard approach is to instead approximate the minimum by means of gradient descent, a straightforward iterative algorithm that makes use of first derivatives as follows[19, Chapter 1]:

$$p_i = p_{i-1} - \eta \nabla C \tag{8}$$

Here, the parameter set p_i (the weights and biases, collectively) are given by moving from the previous set, p_{i-1} in the opposite direction of the gradient of the cost function. The gradient, by construction, is a vector that points in the direction of greatest rate of *increase* for the function — for the purposes of minimization, we find the largest negative gradient in the opposite direction. The constant η denotes a step-size. In machine learning contexts, this is also called the learning rate, because it regulates how much each update will change the parameters during training.

Gradient descent intuitively feels highly computationally intensive, because finding the gradient involves calculating the partial derivative for each of possibly millions of parameters. Fortunately, there is a standard implementation of gradient descent for neural networks, a technique called backpropagation, that is efficient and allows neural networks to be trained in reasonable time. The method has been rediscovered a number of times[16] and an early paper describing the algorithm, making use of the now standard term *backpropagation* was published in 1986[22].

Using the previously described forwards propagation, we can calculate the network's output for a given input, and a cost function such as in equation 7 serves as a measure for the distance between the actual and ideal outputs. For any given neuron in the output layer, its value given by equation 6. The values of this neuron's inputs is a numeric quantity calculated during forward propagation, such that the partial derivatives $\frac{\partial C}{\partial w_j}$ and $\frac{\partial C}{\partial b}$ for the weights and bias associated with this particular neuron can be calculated, given that the cost function and the neuron's activation function can be differentiated and given that the gradient for the entire training set can be written as an average of gradients for each individual sample[19, Chapter 2].

From here on, mirroring the way the weights and biases of neurons in each layer allows us to propagate an input through a network to find its output, the cost can be propagated through the network in the reverse direction, starting at the output layer and working backwards, by use of the chain rule[16]. Thus, backpropagation allows us to calculate the gradient of the cost function with respect to all weights and biases, which is as computationally efficient as calculating the output of the network for a given input. By means of this process, we can start out with randomly initialized weights and biases in the network and iteratively find better values such that the cost descreases and the network approximates an ideal mapping from inputs and outputs. This step-wise process is called training the network and is where the automated learning takes place.

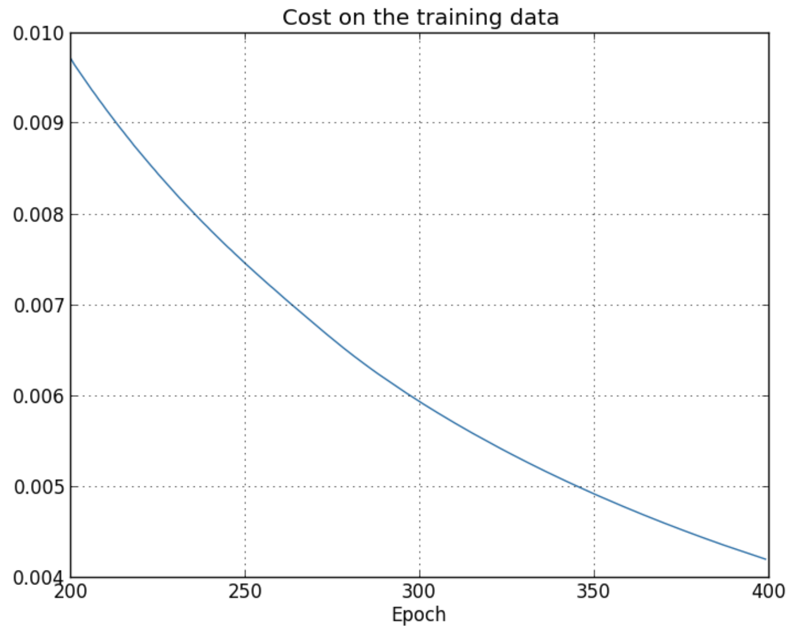


Figure 8: A typical plot of cost as a function of training steps. In this case, the minimization procedure is working as intended, with cost approaching $C = 0$ in a manner resembling exponential decay. An *epoch* refers to gradient descent being performed once for each sample in the training data. Reproduced from Nielsen's book, "Neural Networks and Deep Learning" under the Creative Commons Attribution-NonCommercial 3.0 Unported License.

Unfortunately, gradient descent does not guarantee that parameters p will converge to their ideal value. Two failure modes familiar from single-variable calculus are getting stuck in a local minimum, resulting in a suboptimal solution, or repeatedly overshooting the minimum, resulting in dampened oscillations that slow down convergence, or exploding oscillations that cause catastrophic divergence. In practice, the first failure mode seems to be of limited practical importance, in part due to sparsity of local minima in high-dimensional spaces, and neural networks tend to converge towards similar trained states regardless of initial values. As formulated in a 2015 review article[16]:

> In practice, poor local minima are rarely a problem with large networks. Regardless of the initial conditions, the system nearly always reaches solutions of very similar quality. Recent theoretical and empirical results strongly suggest that local minima

> are not a serious issue in general. Instead, the landscape is packed with a combinatorially large number of saddle points where the gradient is zero, and the surface curves up in most dimensions and curves down in the only a few downward curving directions are present in very large numbers, but almost all of them have very similar values of the objective function. Hence, it does not much matter which of these saddle points the algorithm gets stuck at. (Yann LeCun et al., Nature, 2015)

The second failure mode has been studied extensively, including a number of challenges that show up in higher-dimensional spaces such as saddle points and wildly different gradient magnitudes in different dimensions. A simplistic first refinement to the gradient descent procedure in equation 8 is to introduce a momentum term, such that our update procedure makes use of the history of previous update steps[19, Chapter 3]:

$$m_i = \mu m_{i-1} - \eta \nabla C \tag{9}$$

$$p_i = p_{i-1} + m_i \tag{10}$$

The constant μ regulates to which degree we retain the value of previous update steps — for $\mu = 0$ the procedure has no memory and this set of equations is equivalent to 8, while as μ approaches 1, the dynamics are dominated by the history of updates. This modified gradient descent procedure is motivated by how it allows the updates to build momentum, i.e. proceed with larger steps, in a dimension where the gradient consistently points in the same directions, whereas steps will be relatively smaller in a dimension where the gradient keeps changing direction. A sophisticated optimization method, Adaptive Moment Estimator or Adam, was introduced in 2014[14] and is frequently used to implement gradient descent for neural networks. It makes use of both first order momentum, as introduced above, as well as second order momentum and bias corrections. While considered a robust algortihm — according to its authors: "The hyper-parameters have intuitive interpretations and typically require little tuning.' — trial and error is still best practice for finding a suitable learning rate. While the more sophisticated optimizers are generally more stable, they also have extra parameters that may be tuned, such as the exponential decay rates for the moments.

Deep learning

The term *deep learning* refers to neural networks with a considerable number of hidden layers. In the past, computational limitations and a variety of technical challenges such as vanishing and exploding gradients greatly limited the usefulness of deep neural networks[19, Chapter 5]. In very simple terms, there is a tendency for gradients to differ by orders of magnitude in the earlier and later layers in a network, such that the backpropogation algorithm fails to train the network as a whole. Improvements in network architecture as well as vastly increased hardware performance over the last decades (both in form of overall faster computers as well as the adoption of GPU-based parallelization) have led to a self-reinforcing cycle of increased interest and steadily improved results. The following excerpt from a 2015 review article is a good example of the optimistic, inside view on deep learning[16]:

> Representation learning is a set of methods that allows a machine to be fed with raw data and to automatically discover the representations needed for detection or classification. Deep-learning methods are representation-learning methods with multiple levels of representation, obtained by composing simple but non-linear modules that each transform the representation at one level (starting with the raw

> input) into a representation at a higher, slightly more abstract level. With the composition of enough such transformations, very complex functions can be learned. For classification tasks, higher layers of representation amplify aspects of the input that are important for discrimination and suppress irrelevant variations. An image, for example, comes in the form of an array of pixel values, and the learned features in the first layer of representation typically represent the presence or absence of edges at particular orientations and locations in the image. The second layer typically detects motifs by spotting particular arrangements of edges, regardless of small variations in the edge positions. The third layer may assemble motifs into larger combinations that correspond to parts of familiar objects, and subsequent layers would detect objects as combinations of these parts. The key aspect of deep learning is that these layers of features are not designed by human engineers: they are learned from data using a general-purpose learning procedure.
>
> Deep learning is making major advances in solving problems that have resisted the best attempts of the artificial intelligence community for many years. It has turned out to be very good at discovering intricate structures in high-dimensional data and is therefore applicable to many domains of science, business and government. In addition to beating records in image recognition and speech recognition, it has beaten other machine-learning techniques at predicting the activity of potential drug molecules, analysing particle accelerator data, reconstructing brain circuits, and predicting the effects of mutations in non-coding DNA on gene expression and disease. Perhaps more surprisingly, deep learning has produced extremely promising results for various tasks in natural language understanding, particularly topic classification, sentiment analysis, question answering and language translation.
>
> (Yann LeCun et al., Nature, 2015)

Output layer encoding

In section 3.2, we briefly discussed the design of the output layer in a neural network. In a classification problem like MNIST, where the expected output is one of the ten possible classes, we are faced with a problem — neural networks inherently deal with real valued numbers, not discrete numbers. An intuitive solution would be have just a single neuron in the output layer, using *floor* as an activation function. There are a number of reasons why this design is not used in practice. The simplest explanation is that *floor* is a discontinuous function with a derivative of zero almost everywhere, except for the unit steps where the derivative is undefined. Combining this sort of ill-behaved function with gradient descent would be highly counterproductive.

A partial solution would be to replace *floor* with a continuous approximation. This would however still leave us with a more fundamental problem. In equation 7, we defined a cost function based on the mean square error. If we use a single numeric integer value between 0 and 9 as the output, we implicitly define 1 as far away from 7, because the mean square error is very large, even though these two digits are very similar visually. Mixing up 1 and 7 seems much more reasonable from a human point of view than confusing 7 and 8 — it is unfortunate for sensible mistakes to make disproportionally large contributions to the cost.

The standard way to design the output layer in classifcation problems is to instead use a one-hot encoding[2] for all the classes. This means that each possible class, for MNIST the ten

[2]One-hot encoding is the most common term in machine learning contexts. It is also known as *dummy encoding* or *1-of-n encoding*.

different digits, have one neuron in the output layer assigned to it, and the one-hot representation of any given digit is a ten-dimensional vector with values 0 at all indices except the index that matches with the digit we are representing — i.e. $3 \equiv [0, 0, 0, 1, 0, \ldots, 0]$. Usually, these neurons will all be connected to the last hidden layer in the exact same way. We can use an activation function that constrains the domain of values for the neurons to a well-defined range, such as a sigmoid function with a range $[0, 1]$. If we want to extract a prediction from the network, we choose the class corresponding to the neuron with the largest output value, but otherwise, both for the purposes of computing the cost and during gradient descent, we instead use the continuous one-hot encoded neuron values.

This output encoding has a number of convenient properties. It solves the previously mentioned problems of digits having different distances between them. With a one-hot encoding, cost is calculated as a vector based distance in n-dimensional space, where n is the number of classes. Unlike the integer-based encoding, this makes all the possible classes equidistant. Furthermore, it allows gradual transitions between two different classifications — for instance, the output value for the wrong digit 1 can slowly decrease, while the output value for correct digit 7 increases, and in this case the cost will decrease smoothly throughout the whole process. This is a crucial property, because gradient descent requires non-zero gradients in order to choose a directions for the updates in the parameter space.

Finally, in an integer-based encoding, in order for the classification to change from 1 to 7, gradual updates will have to pass through in-between values corresponding to entirely different digits, such as 2 or 5. A network that is struggling to decide between a classification of 7 and 1 might well produce a nonsensical in-between output of 4. With the one-hot encoding we can directly read out how likely and unlikely the networks considers the different classes for a sample[17, Chapter 4].

ROC and AUC

In section 3.1 we introduced a basic metric for classifier performance, *accuracy*, as discussed some of its shortcomings. An alternative to accuracy is the receiver operating characteristics (ROC) graph and the associated area under curve (AUC) score, which has been in use in signal detection and medical decision making for a long time, and is also widespread in the machine learning community[6]. The key motivation for using ROC over accuracy is the realization that false negatives and false positives can be of wildly different importance. For a medical screening procedure, a large number of false positives can be acceptable if very few positives as misclassified as negative — subsequent, more sophisticated tests can be used to weed these out. In other contexts, we want the opposite sort of classifier, such as in criminal justice where false negatives are far more acceptable than false positives.

> It is better that ten guilty persons escape than that one innocent suffer.
> (Sir William Blackstone, Commentaries on the Law of England)[3]

With a learning algorithm that can produce *scores* rather than just discrete classifications, such as a neural network using the output encoding described in section 3.2, we can interpret these scores as probabilities and evaluate the classification for different probability thresholds. This is most straightforward in the case of binary classification, where we can gradually decrease the positive classification threshold from the greatest score in the dataset to the lowest. This way, we start out with the most conservative classifier possible, which errs in the direction of false negatives, and wind up with a liberal classifier that errs in the direction of false positives[6].

In the general case, such as with the *always positive* and *always negative* classifiers discussed in section 3.1, two different classifiers will simultaneously differ in the number of both false positives *and* false negatives. Without defining the relative importance of false positives and false negatives, we cannot tell which classifier is better. The ROC curve plots the true positive rate against the false positive rate across all possible thresholds[3]. This visualization has a number of useful properties. Working with *rates* normalizes away the class imbalance issue demonstrated by the *always positive* and *always negative* classifiers[6].

> Thus a random classifier will produce a ROC point that *slides* back and forth on the diagonal based on the frequency with which it guesses the positive class. In order to get away from this diagonal into the upper triangular region, the classifier must exploit some information in the data.
>
> (Tom Fawcett, An Introduction to ROC analysis, Pattern Recognition Letters 27)[6]

In an ROC plot, the straight line diagonal from $(\mathrm{fpr}, \mathrm{tpr}) = (0, 0)$ to $(\mathrm{fpr}, \mathrm{tpr}) = (1, 1)$ represents all possible models that do no better than guessing at random. A better ROC curve rises more steeply than this diagonal — that is, its true positive rate increases faster than its false positive rate. An ideal classifier would be able to pick out all positives without mislabeling any of the negatives, which is equivalent to producing an ROC plot that passes through $(\mathrm{fpr}, \mathrm{tpr}) = (0, 1)$[4]. In addition to evaluating the overall steepness of a single model, ROC curves are also useful for drawing comparisons between different models. If one model has a higher true positive rate for all possible false positive rates, it is a better model. In the general case, where their ROC curves intersect, the best classifier depends on the threshold we choose, i.e. the trade-off between false positives and false negatives.

An obvious downside of ROC curves is that they are more complicated to work with than numeric scores, such as we get from accuracy. The single-number representation of an ROC curve is called AUC. This is a shorthand for *area under curve* and can be expressed as the definite integral of the ROC curve. It can be evaluated numerically by use of the trapezoidal rule[6]. A guessing classifier achieves $\mathrm{AUC}_{\mathrm{guessing}} = 0.5$, whereas an ideal classifier approaches the optimal $\mathrm{AUC}_{\mathrm{ideal}} = 1.0$.

Semi-log ROC plot and logAUC

A regular ROC curve as presented in section 3.2 uses linear axes for both the true and the false positive rates. This is a a good starting point, but not ideal when high false positive rates are of limited interest. It could be that we know a priori that we do not want a classifier with a non-trivial number of false positives, as in the case of Blackstone's rule, or that the ROC curves do not show any behavior of interest when the false positive rate exceeds some particular threshold. This is most obvious for a dataset where most samples are well separated and easy to classify, but where we are trying to improve the classifier's performance for the relatively few difficult, in-between samples.

A simple tweak to the ROC plot which improves resolution for low false positive rates is to use a logarithmic x-axis. The results in a logROC plot, where the smaller values of

[3] Particularly in medical applications, sensitivity $\equiv$ true positive rate and $(1 - \text{specificity}) \equiv$ false positive rate are more commonly used.

[4] It is helpful to keep in mind that ROC curves do not depend on the actual scores or their relative sizes, but only on where the positives show up in a rank ordering of all scores. If the scores for positives and negatives are well separated, there will exist a threshold with a high tpr and low fpr.

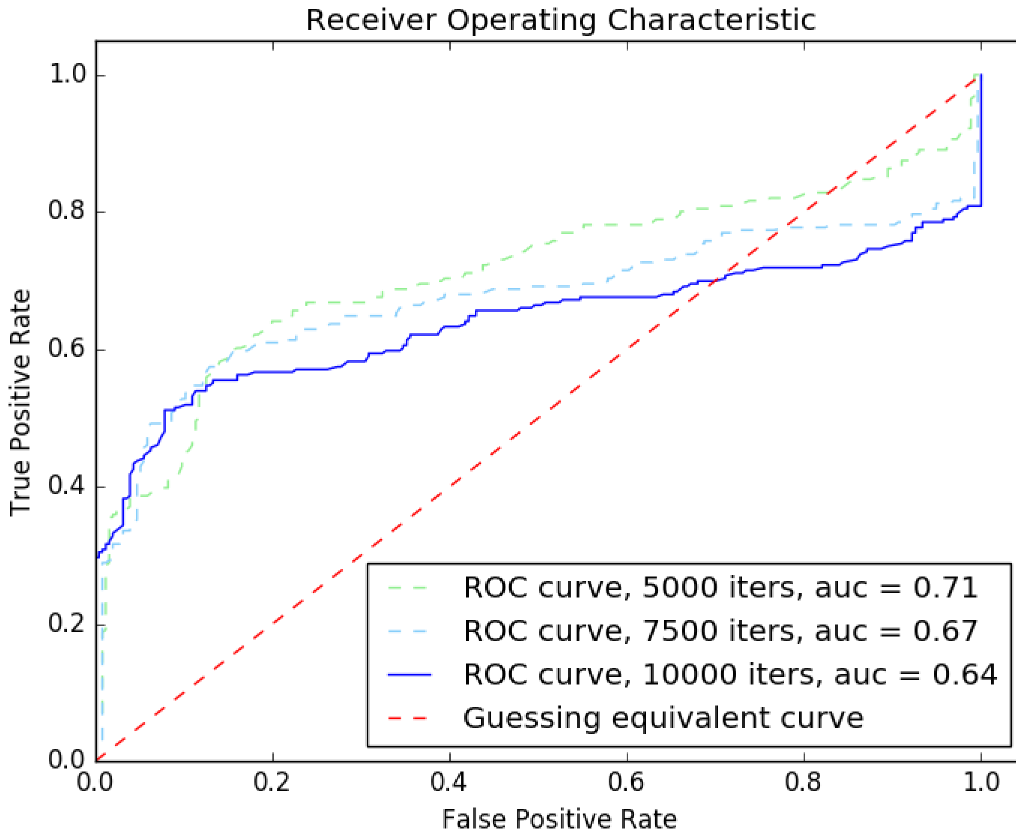


Figure 9: A plot of ROC curves for a GAN-based discriminator classifier. The discriminator's true positive rate is plotted against its false positive rate for all possible treshhold values. The dashed, red line shows the expected performance for a classifier that is blindly guessing — the other three lines shows the performance of the discriminator on a validation set at three different stages in the training process. In this case, classification is not improving during training, and the classifiers exhibit the unusual property of doing worse than chance for fpr > 0.8. Number of training iterations and the AUC-value for each ROC curve is given in the legend.

the false positive rate take up proportionately more space. The area under this curve, the logAUC, assigns relatively greater importance to the early steepness of the ROC curve and thus emphasizes early detection[4, 27].

The logAUC metric introduces a difficulty whenever the highest scored sample corresponds to a true positive. This means that there exists a threshold value such that fpr $= 0$ and tpr > 0, which in turn means that the logAUC is undefined because the definite integral does not converge. A simple, if somewhat cludgy fix for this problem is to treat any threshold which results in zero false positives as if it instead produced half a false positive. The corresponding false positive rate is obtained by dividing by the total number of negatives. This can be partially justified by observing that the false positive rate of zero does not necessarily mean that the classifier would never misclassify a sample as positive, just that this did not happen in the dataset we are working with. Our resolution for the false positive rate is limited by the number of samples in the dataset — the idealized number of false positives, given infinite resolution,

could in principle be anywhere between 0 and 1, and the value of one half is not an unreasonable compromise.

3.3 Convolutional neural networks

With deep neural networks making use of ever greater numbers of hidden layers, it becomes unfeasible to design the individual neurons and connections in the network by hand. Instead, we compose a network by assembling well-defined, specialized layers, or at an ever higher level of abstraction, blocks of layers. A network design for image recognition tasks, ResNet, is named after the residual blocks that constitute its hidden layers. The residual blocks, in turn, makes use of *convolutions*, *rectified linear units* and *shortcuts*, as well as normalizations such as batch normalization. ResNet and other convolution based networks achieve state of the art results in a range of machine vision problems, such as MNIST and ImageNet[5, 13]. In this section, we will explain these concepts in the context of a residual network in detail.

Convolutional layers

Convolutions are motivated by the importance of spatial relationships when processing images. We need some way to make a neural network interpret its inputs as a two-dimensional image, as opposed to just an unordered vector of features. In principle, a fully connected neural network can discover these relationships itself as part of its training process, but this is more wasteful than encoding the spatial relationships directly into the network[19, Chapter 6], resulting in a more computationally demanding and harder to train model.

Convolutional layers work by connecting only a subset of neurons to each neuron in the next layer, and choosing this subset such that it represents a *window* of pixels in the input image, as shown in figure 10. Thus, the values of the neurons in the convolutional layer only depend on a fixed size spatial region in the input layer. Furthermore, the weights of the neurons in the convolutional layer are shared across all the neurons. With a 5×5 convolution, each neuron has 25 inputs from the previous layer, but this set of weights is shared across the whole layer, resulting in just 25 weight parameters that are updated during training and allowing larger and deeper networks to be trained[19, Chapter 6]. The first level of convolution typically represents "the presence or absence of edges at particular orientations and locations in the image"[16]. Sharing these weights across the entire layer means that it will be looking for the same pattern across the whole image. Typically, a large number of convolutions are applied in parallel to the same inputs, such that each of these can detect different patterns.

In addition to capturing the local, spatial relationships in an input image, we can reapply convolution to the outputs from a convolutional layer, building a deeper and deeper network with convolutional layers stacked serially. Holding the size of the convolution window fixed, this means that deeper layers depend on larger windows of the input image at higher levels of abstraction. "The second layer typically detects motifs by spotting particular arrangements of edges, regardless of small variations in the edge positions. The third layer may assemble motifs into larger combinations that correspond to parts of familiar objects, and subsequent layers would detect objects as combinations of these parts."[16]

Padding and stride

As can be seen in figure 10, the use of windowing of the input introduces a problem when close to the edge of the image. We can either choose to just skip all windows that would extend outside of the image, which results in the convolution reducing the overall dimensions as in

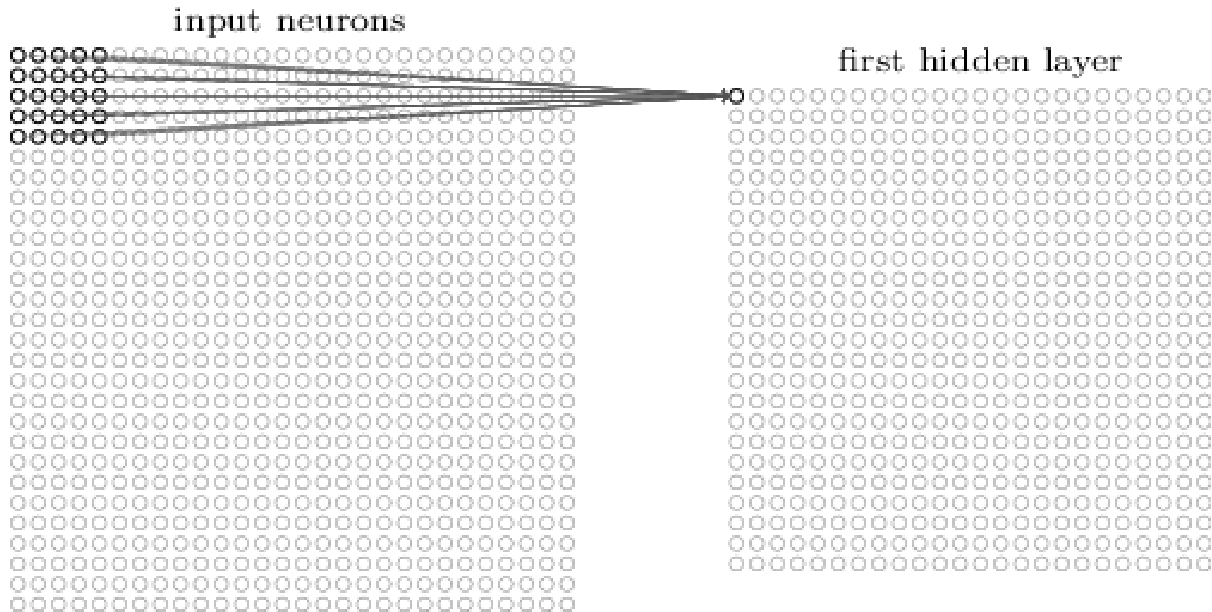


Figure 10: Schematic representation of 28×28 input layer, representing the pixels in an MNIST image, and a subsequent hidden layer of 24×24 neurons which is the result of a convolution. In the hidden layer, all but one neuron is grayed out. In the input layer, all neurons that do not have connections to this neuron are grayed out. Repeating this pattern of connections for the whole hidden layer results in a 5×5 convolution, where the fives denote the width and height of two dimensional windows in the input layer. Reproduced from Nielsen's book, "Neural Networks and Deep Learning" under the Creative Commons Attribution-NonCommercial 3.0 Unported License.

figure 10, or to use a placeholder value for the inputs when the window extends outside of the image. In practice, padding with zero-values is most common, such as in ResNet[13]. For very deep networks in particular, padding is necessary to avoid the constant reduction of dimension from passing the signal through unpadded convolutional, eventually shrinking its size to zero.

While zero-padding is useful for avoiding dimensional reduction as a side effect of convolutional layers, reducing the spatial resolution is part of the design of convolutional neural networks. The general idea is for the deeper layers in the model capture higher level features. For the MNIST problem, we train the network to classify digits, and a high-level feature such as the roundness of the line edges, is not meaningful to analyze at the full 28×28 resolution. ResNet is designed such that the full ImageNet input images are gradually downsampled from their full resolution, 224×224 pixels, to a set of 512 7×7 dimensional feature representations. This downsampling is achieved by the same sort of convolutional layers as previously described, but with the *stride* set to 2[13]. This simply means that the window is shifted by two neuron positions in the two-dimensional grid at a time, such that the convolutional is not applied to every possible window in the input layer. Note that since the window size 3×3, the convolutional layer still makes use of all of the inputs, but with less overlapping.

Rectified linear units

In equation 6, we introduced the activation function f that is applied when calculating the output from a neuron. The most popular activation function for use in deep learning applications

is the rectified linear unit (ReLU)[16]. It is defined as follows[8]:

$$f(z) = \max(z, 0) \tag{11}$$

While seemingly breaking with the biological foundations of neural networks and introducing a number of potentially problematic properties, such as vanishing gradient for negative values of the pre-activation neuron output z, ReLU has been shown to outperform other activation functions, such as *softplus* and *tanh* for use in deep learning networks applied to image data[8] and is used in ResNet[13].

A number of varieties of ReLU have been proposed, such as the leaky rectified linear unit:

$$f(z) = \max(z, -\alpha z) \tag{12}$$

Here, α is a small, non-negative number on the order of 10^-1. Leaky ReLU adresses the problem of vanishing gradients. When using ReLU, some fraction of the neurons will wind up stuck in a state where they have a negative pre-activation output, causing them to output zero, and to fail to respond to training because its output does not respond to any small update of its weights. However, these alternative activation functions are more computationally demanding, introducing extra parameters (such as α) and do not conclusively improve the results.

Shortcut

The design feature that sets ResNet apart from other convolutional neural networks is the use of shortcut connections. With the introduction of convolutional layers, which allow for deeper networks that still have a manageable number of parameters, the trend has been for networks to become ever deeper[5]. Even setting aside computational limitations, training networks with hundreds of layers is difficult. Roughly speaking, the problem is that the parameters in any given layer cannot be optimized independently of the rest of the network, and the training procedure does not work properly for any layer while too many of the layers in the chain are badly tuned.

A number of clever tricks have been discovered to work around this problem, such that networks can be made deeper and learn more sophisticated relationships. One example is the use of unsupervised pre-training[8]. Another approach, more closely related to the method used in ResNet, is to pre-train a network with fewer layers, before inserting additional layers and re-training the modified, deeper network. This sort of incremental bootstrapping procedure makes it easier for a meaningful signal to propagate through the deeper network such that gradient descent optimization works properly.

ResNet makes use of shortcuts, or less colloquially *additive identity mappings*, in its residual blocks. In a residual block, the input is passed through two successive convolutional layers, both with 3×3 convolution filters. However, instead of outputting just the result of this double convolution, it outputs the sum of this result and the unmodified input[13]. The motivation for this is to make sure that a meaningful signal passes through layers despite its convolutional parameters being poorly tuned, ideally resulting in a more stable training procedure even for very deep networks. The reported results show that the networks with shortcut connections perform significantly better without introducing any extra parameters or computational load, and that with this architecture, the classification results for ResNets keeps improving with increased depth[13].

3.4 GAN

The foundational paper for generative adversarial networks was published by Goodfellow in 2014[9]. Most traditional deep learning applications, such as the very successful image classification models, are discriminative in nature, mapping a high-dimensional input to a much simpler output, such as a discrete classification or a single scalar output. A generative network instead tries to generate rich, high-dimensional outputs, for instance an image, from a relatively simpler input vector.

The idea introduced by Goodfellow is to train a discriminative and a generative network simultaneously, by designing their respective cost functions such that their training procedures encourage them to compete against each other. The generator tries to generate new data, such as in the context of the MNIST problem images that look like handwritten digits. The discriminator tries to discriminate between real data, such as actually handwritten digits from the MNIST dataset, and the faux data from the generator. This discrimination is represented by mapping the data inputs to scalars that denote the probability that the corresponding input is real data. In the ideal case — where the networks have sufficient representational power and the training process converges towards the global optimum — the generator will produce samples with a distribution that matches with the distribution of the real data, and the discriminator will detect any difference between the generated and the real distributions[9].

In Goodfellow's original formulation, the discriminiator and generator (D, G) play a minimax game with the following value function V[9]:

$$\min_G \max_D V(D, G) = \mathbb{E}_{x \sim p_{\text{data}}(x)}[\log D(x)] + \mathbb{E}_{z \sim p_z(z)}[\log(1 - D(G(z)))] \tag{13}$$

In this equation, the adversarial relationship between the discriminator and generator is made explicit, in that they try to maximize and minimize the value function respectively. The first term on the right hand side deals with the values of the discriminator when applied to real data x, and the second term with the discriminator values when applied to generated data $G(z)$. The maximization procedure for the discriminator is fulfilled by $D(x) \to 1$ and $D(G(z)) \to 0$, whereas the minimization procedure for the generator counteracts this by making $G(z) \to x$. The Nash equilibrium for the minimax game is achieved when $G(z) \equiv x$ and $D(x) = D(G(z) = 0.5$[23, 9], which means that the generator's outputs are distributed exactly like real data and the discriminator has no ability to tell generated and real data apart.

In the results sections, Goodfellow demonstrates good performance for the GAN implementation on the MNIST dataset, training a generator that can generate images that visually resemble handwritten digits[9, Figure 2]. Furthermore, it is also shown that linear interpolation between the values of z corresponding to different generated digits produce gradual transitions from one digit to the other[9, Figure 3].

In practice, training generative adversarial networks is difficult, and a number of modifications to have been proposed to improve training stability of GANs since they were first introduced in 2014[2, 11, 18, 20]. Stability is critically important, both in order to achieve good results and to reduce the importance of carefully tuning hyperparameters and is an area of active research.

Wasserstein GAN with gradient penalty

Gulrajani et al. introduce a modification to the original GAN formulation, building on a previous reformulation of the GAN value function by Arjovsky et al[2].

> Generative Adversarial Networks (GANs) are powerful generative models, but suffer from training instability. The recently proposed Wasserstein GAN (WGAN) makes progress toward stable training of GANs, but sometimes can still generate only poor samples or fail to converge. We find that these problems are often due to the use of weight clipping in WGAN to enforce a Lipschitz constraint on the critic, which can lead to undesired behavior. We propose an alternative to clipping weights: penalize the norm of gradient of the critic with respect to its input. Our proposed method performs better than standard WGAN and enables stable training of a wide variety of GAN architectures with almost no hyperparameter tuning, including 101-layer ResNets and language models with continuous generators. We also achieve high quality generations on CIFAR-10 and LSUN bedrooms.
>
> (Gulrajani, Abstract from "Improved Training of Wasserstein GANs", 2017)[11]

We summarize the two key modifications to the training process. First, Goodfellow's value function (equation 13) is replaced by making use of the *Earth-Mover* or *Wassterstein-1* distance, which informally measures the distance between two distributions as the minimum necessary work to transform one distribution to the other, where the work is given as a product of the mass that is moved times the distance it is moved.

$$\min_{G} \max_{D \in \mathbb{D}} [\mathbb{E}_{x \sim \mathbb{P}_r}[D(x)] - \mathbb{E}_{G(z) \sim p_z}[D(G(z))]] \tag{14}$$

Second, a gradient penalty is introduced to the stabilize the training process. The full equation is given in the original paper[11, Eq 3]. We instead explain this gradient penalty term more informally. During a training iteration, we find an interpolated sample drawn randomly from anywhere on the straight line segment between a real sample x and a generated sample $G(z)$. We evaluate the gradient of the discriminator at this point, and penalize the discriminator proportionally to the square deviation from a gradient of 1. This enforces the WGAN requirement that the discriminator respects the 1-Lipschitz constraint[11].

3.5 Anomaly detection with generative adversarial networks

Discriminator based anomaly detection

The most obvious way of applying a generative adversarial network to anomaly detection is to make use of the scalar outputs from the discriminator. As described in section 3.4, the discriminator can map input images to a probability, representing the likelihood that the input image is drawn from real data. The differences between the real data distributions, and the distribution of fake data generated by the generator makes it possible for the discriminator to tell them apart.

The purpose of the discriminator is to provide a signal during training that *steers* the generator towards the outputs that match with real data. Anomalies, however, share this property with imperfectly generated data, that their distribution differs from that of real data. In principle, the discriminator that is used to train the generator on a dataset can be used to detect anomalies in data where the background is similarly distributed, given the reasonable assumption that anomalies will be assigned lower scores than the background.

Unfortunately, directly applying a discriminator network to detect anomalies is not reliable. The key problem is that the discriminator during training does not as much learn the distribution of real data, as it becomes hypersensitive to any *differences* between the real data distribution and the outputs from the generator. Whether this sensitivity also allows is to

detect anomalies depends on details of what sort of errors the generator makes during training and how the training carries on after the generator has improved to the point where the discriminator can no longer discern between real and generated data.

> For GANs, one thing to keep in mind is that the discriminator is not a generalized detector of weird things. It is trying to tell whether a sample came from the real data or *one specific non-data distribution: the generator*. Because of that, it seems like the discriminator would only be useful for anomaly detection if you think you can make your generator resemble the anomalies you expect to need to detect.
>
> (Ian Goodfellow, in an informal Quora answer, 2017)[10]

Generator based anomaly detection

An alternative to the discriminator based method described above is to instead use the trained *generator* for anomaly detection. This is much less straightforward, because the generator does not output any sort of probabilities or scores. The fundamental idea is that the generator ought to be have learned mappings to outputs that resemble real data due to the minimax procedure in eqation 13. We can exploit this property by, for any given sample, making the generator find its best recreation of this sample. Ideally, it will recreate anomalous data with less fidelity, allowing us to use the relative fidelity of the recreation of a set of samples to detect anomalies.

Schlegl et al. implement GAN based anomaly detection based on this intuition[24], first training the network on medical images from healthy people before applying a mixed discriminator-generator based anomaly score function to detect and segment abnormal tissue in unseen data. They report good results for the ROC score and verify that the generated images are *realistic looking* and that the reconstructed images match better with the samples for normal data than for abnormal data.

They define the anomaly score as follows:

$$A(x) = (1 - \lambda)R(x) + \lambda D(x) \tag{15}$$

Where x is the query image (the sample that we are trying to determine whether is an anomaly) and the λ-factors in each term determine a relative weighing of R and D. $R(x)$ is the residual loss, defined as the norm of the difference between the query image x and the reconstructed image $G(z)$, minimized with respect to the generator input z. $D(x)$ is a discrimination score, but based on the norm of the difference between x and $G(z)$ when mapped to a higher-dimensional feature space. This mapping is achieved by extracting the values of the discriminator network in one of its hidden layers[24].

Reconstruction cost

Based on similar ideas as those in the previously discussed paper, we have defined our own reconstruction cost in order to apply the generator as an anomaly detector:

$$R(x, z) = \min_z [C_1 \cdot \|x - G(z)\| + C_2 \cdot \|z\|] \tag{16}$$

Here, x is the target sample and z the generator input. The value of the reconstruction cost is minimized with respect to z by gradient descent. C_1 and C_2 are weight constants and the L2-norm is used. In our formulation, there is no discrimination score. We have instead introduced a new term, the norm of z, such that there is a contribution to the reconstruction

cost from the vector that is input into the generator. With z being normally distributed in each component, this effectively penalizes reconstructions that make use of unlikely inputs for the generator.

4 Materials and methods

4.1 Hardware and software

We have used a computer with 4 modern GPUs (GeForce GTX Titan X) to run the machine learning algorithms. We used the Ubuntu 16.04 as the operating system, running Python 2.7.12[21]. The machine learning algorithm was implemented in TensorFlow 1.4.0[1], an open source software library developed by Google Brain and specialized for implementing neural networks. We also made extensive use of NumPy[26], a library for optimized and convenient numeric computations.

4.2 Data material

All our experiments make use of the Bjoerkelangen dataset provided by the Norwegian Defence Research Establishment. The dataset is not available to the public. The data was provided as a pre-processed TIFF image and has a 1600×700 pixels resolution. For each pixel, there are 20 signal intensities in the range $[0, 256]$, corresponding to different wavelengths in the range [0.410,0.984] µm. The image can be seen in figure 11, captured from above with hyperspectral imaging techniques. There are fourteen different targets (anomalies) of different size and visibility, some of them in shadow. The background is a natural scene with an agricultural region and a small forest near the village of Bjoerkelangen in Norway[4]. A square root transform to normalize the noise levels across the spectral bands had already been applied to the dataset, as described in a previous publication[12].

The dataset includes inner and outer masks for the targets. From the whole image, we create a large number of subimages by extracting $m \times m$-size windows. We partition windows containing no pixels inside of the inner or outer masks into a training and valdiation set, all considered background (non-anomalous). Windows containing only pixels from the outer target masks are ambiguous and left out, while windows containing inner mas target pixels are considered the anomalies to be detected.

4.3 GAN implementation

We base our implementation of the generative adversarial network on a previous implementation by Gulrajani et al. In the paper "Improved Training of Wasserstein GANs", they introduce a GAN model based on the Wasserstein distance metric and making use of a novel gradient penalty term added to the cost function of the discriminator. The paper includes a link to the full code in the form of a public GitHub repository[11]. We chose to use a clone of this repository as a starting point, in part because it is a Python/TensorFlow-based implementation, which we were comfortable working with. In addition, the implementation shows good stability over a wide range of different tasks and hyperparameters, as demonstrated in the paper.

Our approach was to make the minimal possible changes to the ImageNet code in the repository, *gan_64x64.py*, to make it run with our surveillance image data as the input, and then make step by step changes to improve it. Throughout the process, most of the code has been modified. The generator and discriminator networks have been reimplemented, making more use of standard TensorFlow functions and replacing the two-dimensional convolutions with three-dimensional convolutions, treating the spectral domain as a pseudo-spatial dimension. The training logic remains mostly unchanged. A fair amount of logging and evaluation tools have been added, as well as entirely new code for calculating the recreation cost for the the

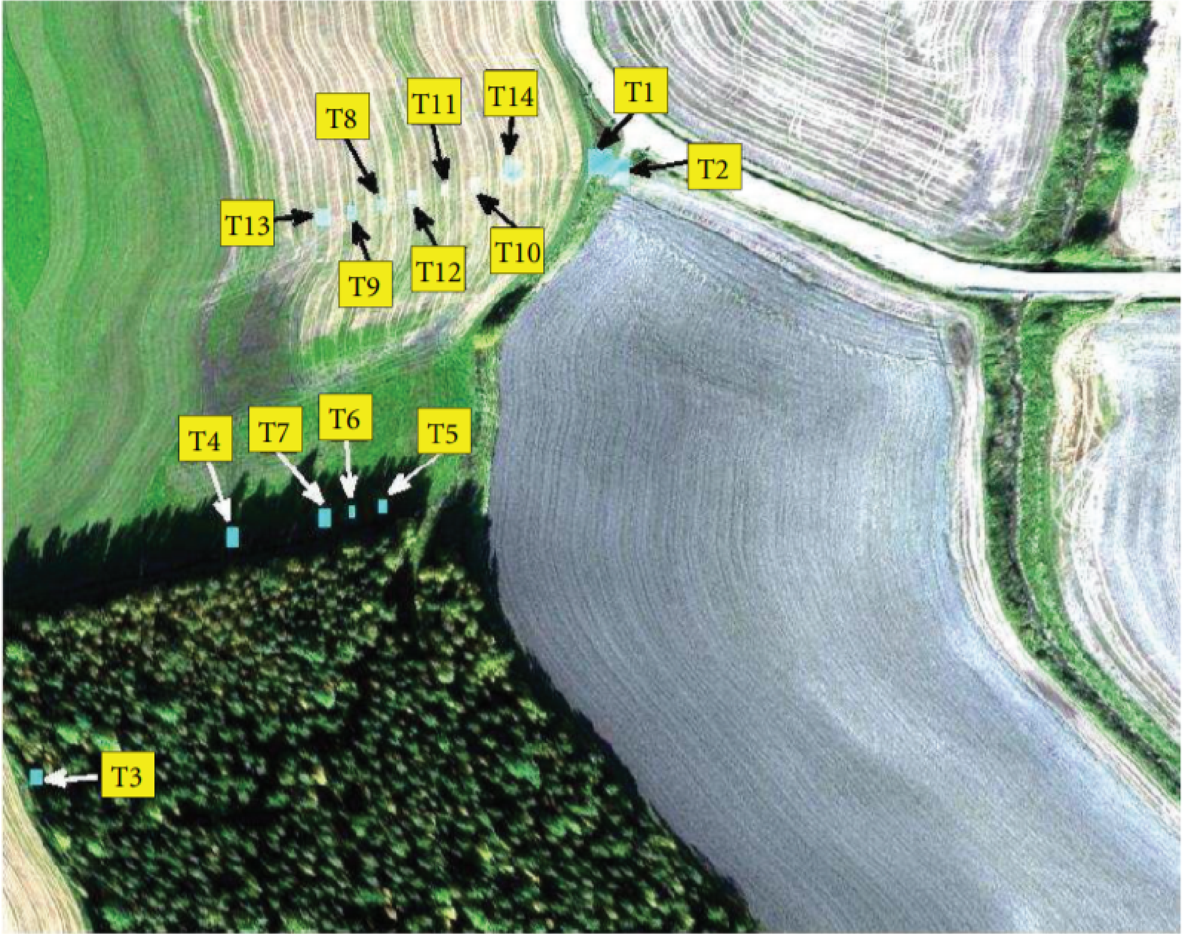


Figure 11: RGB composite of the Bjoerkelangen dataset, with anomalies highlighted and numbered (T1-T14). Aspect ratio is not preserved. Reproduced from "Hyperspectral Anomaly Detection: Comparative Evaluation in Scenes with Diverse Complexity" with permission[4].

generator. The full code is available in a public GitHub repository[5].

Discriminator and generator

We include the part of the code that defines the ResNet-based generator network in TensorFlow.

```
KERNEL = (2,1,1)
STRIDE, STRIDE_SPECIAL = (2,1,1), (1,1,1)

def ResnetGenerator(n_samples, noise=None, dim=DIM, is_training=True):
    with tf.variable_scope('G', reuse=tf.AUTO_REUSE):
        norm = tf.identity
        act = tf.nn.relu
        #Generate noise to seed the generator with
        #If noise passed as argument, use this instead
        #This allows us to re-use the same noise across iterations
        #for visualizing images generated by the Generator
        #Introduce NOISE_DIM constant (length of noise vector)
        if noise is None: noise = tf.random_normal([n_samples, NOISE_DIM], dtype=tf.float32)

        output = tf.layers.dense(noise, 16*dim)
        #Here we need to reshape into 5D in order to work with conv3d
        #Tensorflow expects 'NDHWC' format - n_samples,depth,height,width,channels
        #We map 'd' to wavelength
        output = tf.reshape(output, [-1, 1, 1, 1, 16*dim])
```

[5]Public repository: "https://github.com/awweide/pub-ffi-gan

```
def GenResBlock(input, input_dim, output_dim, kernel, stride):
    shortcut = tf.layers.conv3d_transpose(input, output_dim, kernel, stride, padding='valid')
    conv = input
    conv = NormalizeThenActivate(conv, normalizer=norm, activator=act)
    conv = tf.layers.conv3d_transpose(conv, output_dim, kernel, padding='same')
    conv = NormalizeThenActivate(conv, normalizer=norm, activator=act)
    conv = tf.layers.conv3d_transpose(conv, output_dim, kernel, stride, padding='valid')
    return shortcut + conv

output = GenResBlock(output, 32*dim, 16*dim, kernel=KERNEL, stride=STRIDE)
output = GenResBlock(output, 16*dim, 8*dim, kernel=KERNEL, stride=STRIDE)
output = GenResBlock(output, 8*dim, 4*dim, kernel=KERNEL, stride=STRIDE_SPECIAL)
output = GenResBlock(output, 4*dim, 2*dim, kernel=KERNEL, stride=STRIDE)
output = GenResBlock(output, 2*dim, 1*dim, kernel=KERNEL, stride=STRIDE)

output = NormalizeThenActivate(output, normalizer=norm, activator=act)
output = tf.layers.conv3d_transpose(output, 1, KERNEL, padding='same')
output = tf.nn.tanh(output)

return tf.reshape(output, [-1, CHANNELS, WINDOW, WINDOW])
```

Note in particular the five stacked residual blocks, as described in section 3.3, each returning the sum of the shortcut-term and the result of two sequenced convolutions. Because the generator is a mapping from a relatively low-dimensional input to a richer image domain, it makes use of transpose convolutions, which function like regular convolutions with the directions reversed. The discriminator network has a very similar design, essentially running in the opposite direction, and makes use of regular three-dimensional convolutions in its residual blocks.

Recreation cost

We include the part of the code that calculates the recreation cost for a part of the surveillance image, which is a measure of how well the generator is able to generate an image that matches with the real data image.

```
def batch_cost(batch_real, batch_fake, batch_noise):
    #Partitions data into different chunks for each GPU : GPU-parallelization
    split_batch_real = tf.split(batch_real, len(DEVICES))
    split_batch_fake = tf.split(batch_fake, len(DEVICES))
    split_batch_noise = tf.split(batch_noise, len(DEVICES))

    #Collector for partial results from each GPU
    cost_collector = []

    #Parallelized cost calculations
    for device_index, (device, _real, _fake, _noise) in enumerate(zip(DEVICES, split_batch_real,
         split_batch_fake, split_batch_noise)):
        with tf.device(device), tf.name_scope('device_index'):
            batch_diff = tf.subtract(_real, _fake)
            batch_diff_norm = tf.reduce_sum(tf.square(batch_diff), axis=[1,2,3]) / OUTPUT_DIM
            batch_noise_norm = tf.reduce_sum(tf.square(_noise), axis=[1]) / NOISE_DIM
            cost_collector.append(batch_diff_norm + NOISE_WEIGHT * batch_noise_norm)
    return tf.concat(cost_collector, 0)

def batch_minimize(batch_real):
    #Do minimzation a number of times with different initial values
    for min_run in xrange(MINIMIZE_RUNS):
        #Initialize gen_input with random noise and reset optimizer
        _,_ = session.run([init_op_noise, init_op_minimizer])
        #Minimize noise for a number of iterations
        for min_iter in xrange(MINIMIZE_ITERATIONS):
            _, batch_scores_run, batch_fake_run = session.run([minimize_op, batch_reconstruction_cost,
                 gen_from_noise], feed_dict = {batch_real_data : batch_real})
        #If first run, no comparisons to make
        if min_run == 0:
            batch_scores_min = batch_scores_run
            batch_fake_min = batch_fake_run
        #Otherwise, keep the lowest score and corresponding image for each
        else:
           m = np.argmin([batch_scores_min, batch_scores_run], axis=0)
            for i in xrange(BATCH_SIZE):
                if m[i]: batch_scores_min[i],batch_fake_min[i] = batch_scores_run[i],batch_fake_run[i]
    return batch_scores_min, batch_real, batch_fake_min
```

The reconstruction cost is calculated as a weighted sum of the pixel-wise difference between the real image and the reconstruction image as well as the norm of the noise input that was used for the reconstruction. The best possible reconstruction is found by finding the lowest cost over a number of differently initialized gradient descent based minimizations.

4.4 Experiments

Our experiments use a large number of different hyperparameters. We here list the name of the parameters we have varied across the different runs and their default value and explain how they affect the implementation:

WINDOW=1: The width and height in pixels of the window used to subdivide the Bjoerkelangen image as described in section 4.2. For the default value of 1, this means that samples have dimensions $[20, 1, 1]$. For larger window sizes, KERNEL and STRIDE must be changed such that the network architecture properly convolves in the spatial dimensions.

DIM=8: This parameter regulates the overall complexity of the model. The number of parallel convolution layers scales with this parameter.

NOISE_DIM=4: The number of components of the z-vector which is the input to the generator. For the default value, the generator input is a vector of length 4, with each component drawn from a normal distribution.

NOISE_WEIGHT=100: Regulates how many orders of magnitude more heavily weighed the reconstruction quality $G(z)$ is than improbability of the generator input z, when calculating the reconstruction cost in equation 16. For NOISE_WEIGHT = 100, the improbability term is simply omitted.

MINIMIZE_ITERATIONS=50: The number of iterations of gradient descent performed when minimizing the reconstruction cost with respect to z.

MINIMIZE_RUNS=4: The number of different initializations of z used when minimizing the reconstruction cost.

5 Results and discussion

5.1 Single pixel baseline results

We first present the results for the GAN model with the default hyperparameters. Throughout this section, we will use these results as the baseline for comparison. With these parameter settings, the model can be trained and tested on the whole image in roughly two hours. The majority of the time is spent on the minimization process in equation 16.

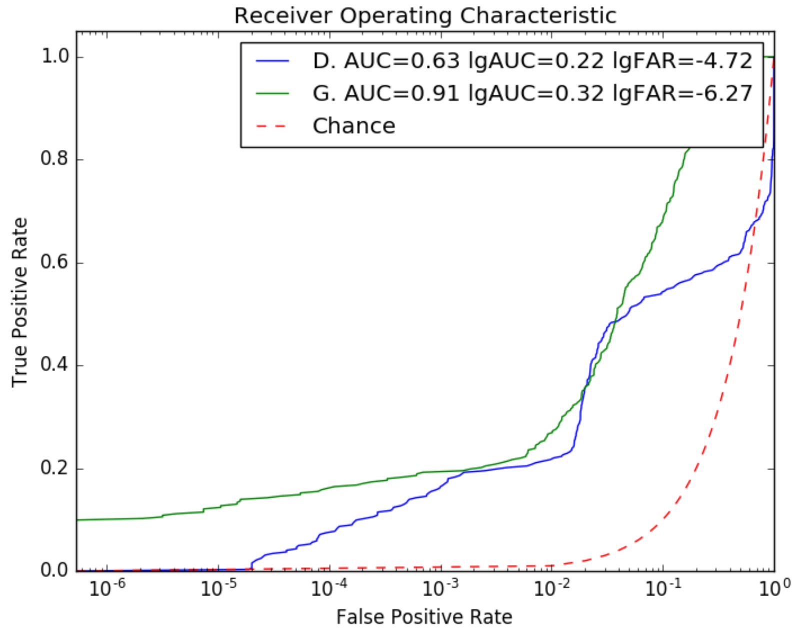
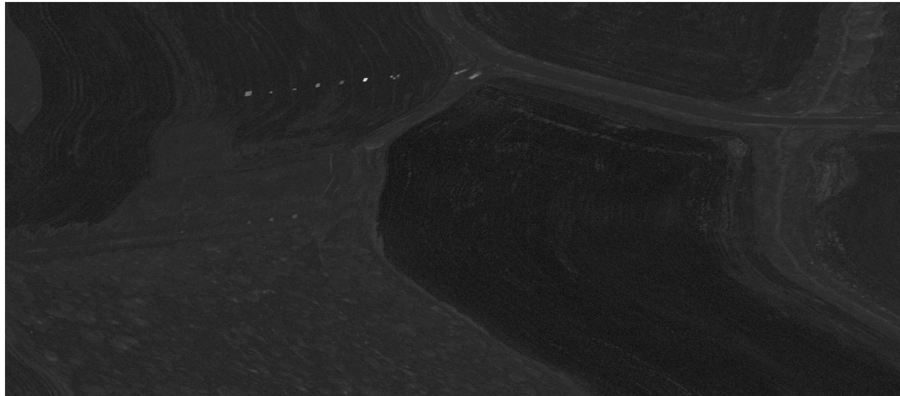


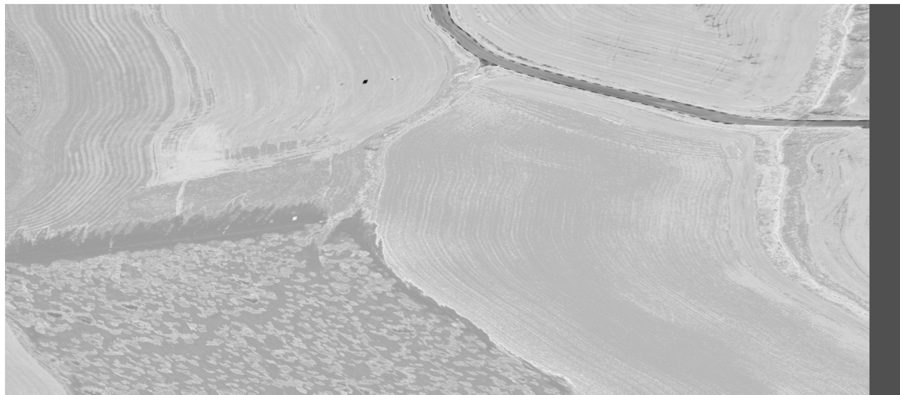
Figure 12: logROC plot for anomaly detection for all targets using the baseline model. *D.* and *G.* represent the discriminator and generator based anomaly detection methods respectively.

Figure 12 shows the logROC plot for both the discriminator and generator-based anomaly detection. Compared to the expected AUC score of 0.50 for a detector that is just guessing, the discriminator detector fares very poorly, particularly considering how much some of the anomalies in the image stand out. The generator detector performs much better. $AUC = 0.91$ is a fairly strong result for many machine learning problems and clearly demonstrates that it is possible to use the generator recreation cost to detect anomalies.

(a) Heatmap for anomaly detection with generator using the baseline model.



(b) Heatmap for anomaly detection with multinormal mixture model[4].



(c) Heatmap for anomaly detection with discriminator using the baseline model.

Figure 13: Heatmaps, where pixel brightness represents the detection models' assigned likelihood that the pixel represents an anomaly. Absolute brightness is not comparable between heatmaps. Our model (a) shows much poorer contrast between anomalies and background than the statistical model (b) and fewer of the targets are readily discernible in the image. Also included (c) is the discriminator based heatmap for the same model.

Subfigures a-b in figure 13 shows that our anoamly detection model is considerably worse than previous results for the same dataset using statistical models[4]. The baseline model's heatmap also demonstrates some of the problems with the model. The road and particularly

the markings on the side of the road (top right quadrant of the image) is much brighter than ideal. The heatmap is fairly noisy overall, particularly in the darker regions of the image, such as the forest (bottom left quadrant) and the brook (oriented top down along right edge of image). Some undesirable behavior is common to both of the heatmaps, such as contours in the image also showing up in the heatmap and smaller, highlighted patches in the forested area.

While the discriminator based detection does not work well, as demonstrated in figure 12, the corresponding heatmap (subfigure c in figure 13) is still of interest. T10, the anomaly that stands out most clearly in both the other heatmaps as well as the RGB image of the scene, is the *darkest* region in the discriminator heatmap. Because this anomaly is fairly large in number of pixels, this has a great effect its ROC and AUC scores. Note also that other targets such as T5 and T6 stand out more clearly in the discriminator heatmap than in the generator heatmap. This suggests that the disciminator detector is not strictly worse than the generator detector, but is more sensitive to some of the anomalies.

There is also an interesting pattern where some of the very bright regions in the generator heatmap (T10, road with markings, small highlighted regions in forested area) are similarly dark in the discriminator heatmap. We will discuss this in more detail later.

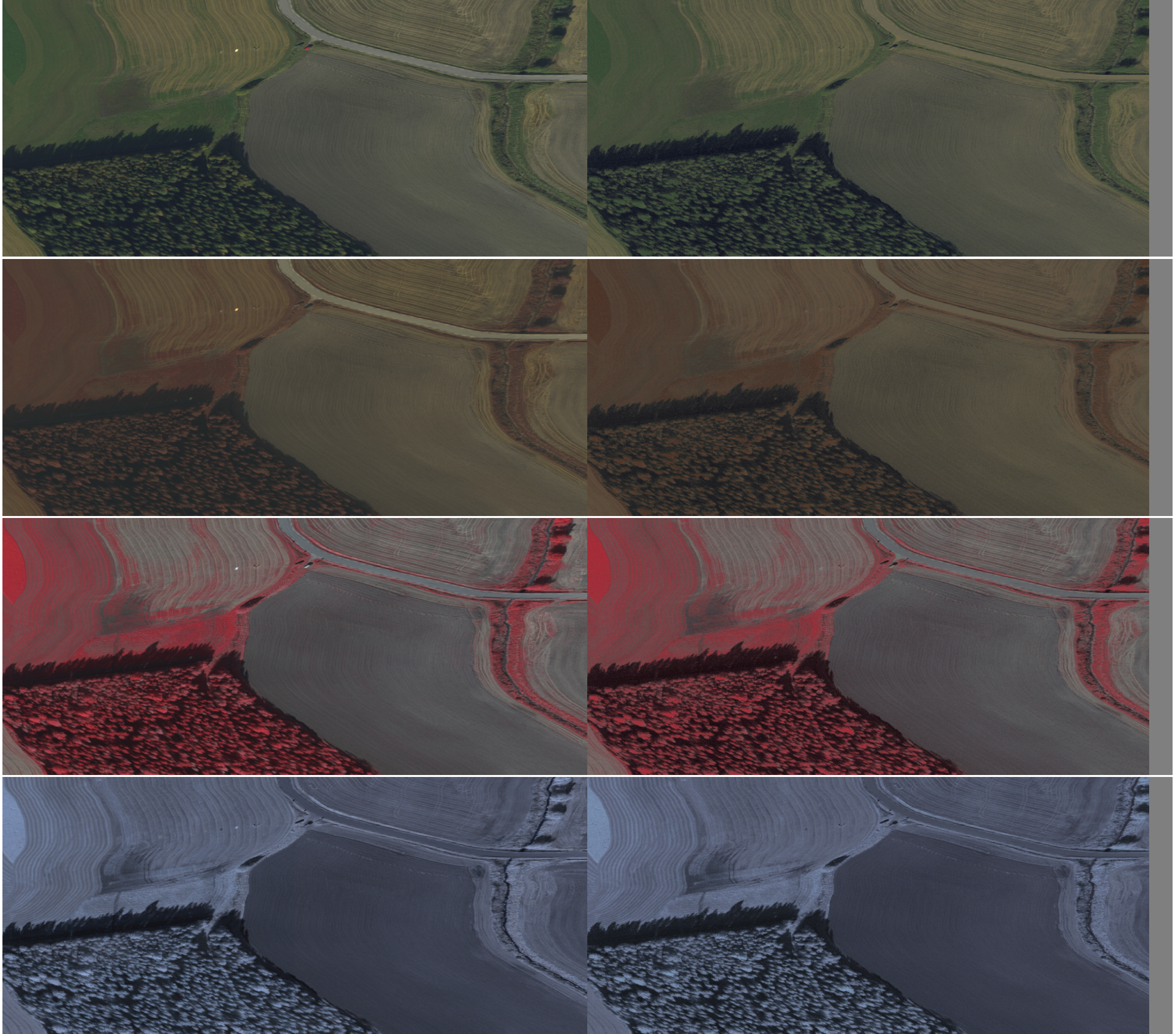


Figure 14: Reconstruction of entire image with generator using baseline model. Left: real image. Right: Reconstruction. Spectra from top to bottom: RGB image and different low-, mid- and high-wavelengths mapped to RGB channels.

Figure 14 shows a side-by-side comparison of the real image and the pixel-wise reconstruction using the baseline model's generator. The general impression is that the reconstruction matches well with the real image. Note in particular how T10 *fades* into the background in the reconstructed image and how T2, which is bright red in the real image has been reconstructed with a poorly matching color that is more common in the real image. This demonstrates very clearly that the idea presented in section 3.5 actually works. The targets are reconstructed less perfectly than the background. In figure 13 we see that this also translates to high values for the reconstruction cost.

Figure 14 also allows us to explain the relative brightness of the road and markings in the generator heatmap in figure 13. In the reconstructed image, we can see that this part of the background has been reconstructed poorly, again making use of somewhat similar colors that are more common in the real image. This means that the generator has not learned to generate data from this part of the image. In the forest region, we can also see that the real image has much more color variety, particularly yellow and brighter greens, than the reconstruction.

The most tempting explanation for this is that the generator inputs only have 4 components or the relatively low complexity of the models and these pixels being too scarce in the dataset for the generator to learn them. We explore these possibilities in later sections.

Observing this behavior from the generator also allows us to explain why the discriminator heatmap is very dark for T10 and for the road markings. As demonstrated in figure 14, the generator is not capable of producing the samples that match the real image samples for these locations. This means that during the training process, whenever the discriminator is passed samples from the road markings, it can safely assume that these are not faked by the generator. Because of this limitation in the generator, the discriminator is rewarded for mapping these pixel signatures to near-zero probability of being generated. Thus, the discriminator concludes that pixels of this kind cannot be anomalies. In the case of the road markings, it is correct, unlike the generator. In the case of T10, it is the other way around. Overall, this seems to match well with theory from section 3.5.

The reconstruction also exhibits noisy behavior, where occasional, randomly scattered pixels seem to be reconstructed badly. This is most visible in the darker regions of the image, such as near T5. Repeated runs shows that this is a random effect, likely due to the minimization procedure failing to find a good approximation of the global optimum. This contributes to false positives and poorer overall ROC, and is part of the explanation for why the generator heatmap looks very noisy.

5.2 Single pixel results for model variations

In this section, we present results for models with minimal variations, for comparison with the baseline results, in order to demonstrate a number of interesting properties of the anomaly detection method.

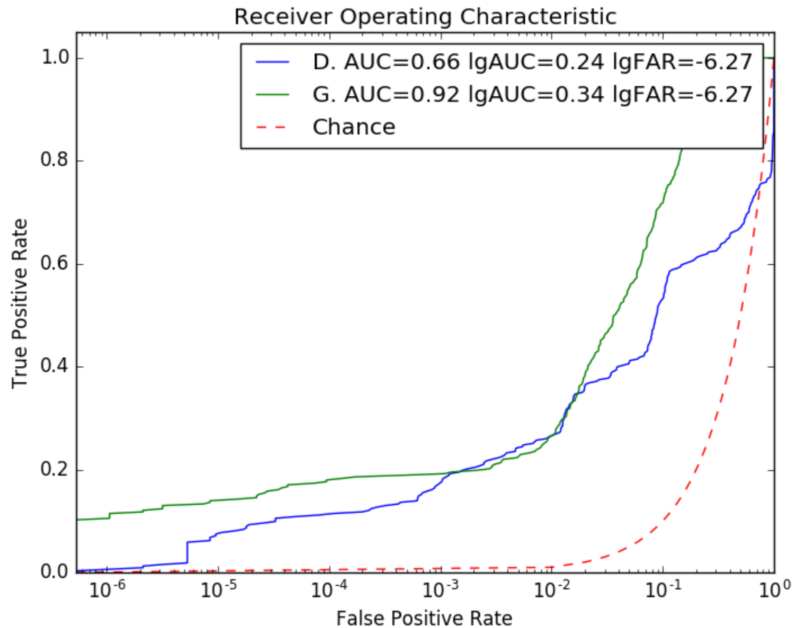
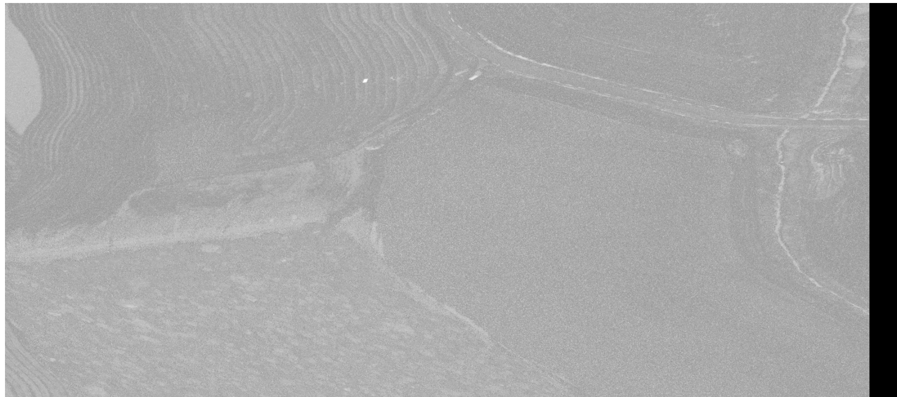


Figure 15: logROC plot for anomaly detection for all targets, where the model is trained on the entire dataset without removing the anomalies.

Figure 15 shows the ROC plot for the exact same model as the baseline, but without excluding the anomalies from the dataset during the training. Comparing these with the baseline results in figure 12 shows no clear difference between the ROC plots. This shows that the anomaly detection method works just as well for entirely unsupervised learning — we can use it to detect anomalies in an image without having an anomaly free background to use during training or masks for the targets. This is a very positive result, and is consistent with the interpretation that the failure to reconstruct the road in figure 14 is due to how it makes up a relatively small fraction of the background.

(a) RGB reconstruction of image.



(b) Corresponding heatmap.

Figure 16: Reconstruction and heatmaps for a model that has been trained on a dataset with a boosted fraction of road samples.

In order to test the intuition that it is the scarcity of road samples in the dataset that leads to the generator failing to reconstruct it, we trained the exact same model on a slightly modified dataset, where a portion of the image has been replaced with copies of the pixels from the road. Figure 16 shows the resulting reconstruction and heatmap. Notably, the road is reconstructed with much better fidelity in this case and the road no longer shows up nearly as brightly in the heatmap. This result is a bit of a mixed bag. It is good that there is a relationship between how common a type of samples is in the dataset and whether they are detected as anomalies, but ideally the reconstruction should not fail for something as well-represented in the dataset as road pixels.

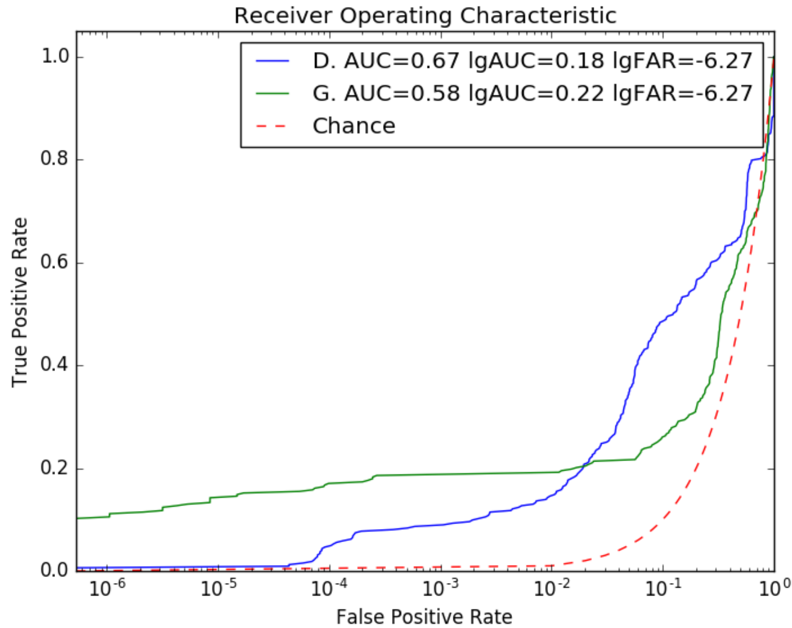


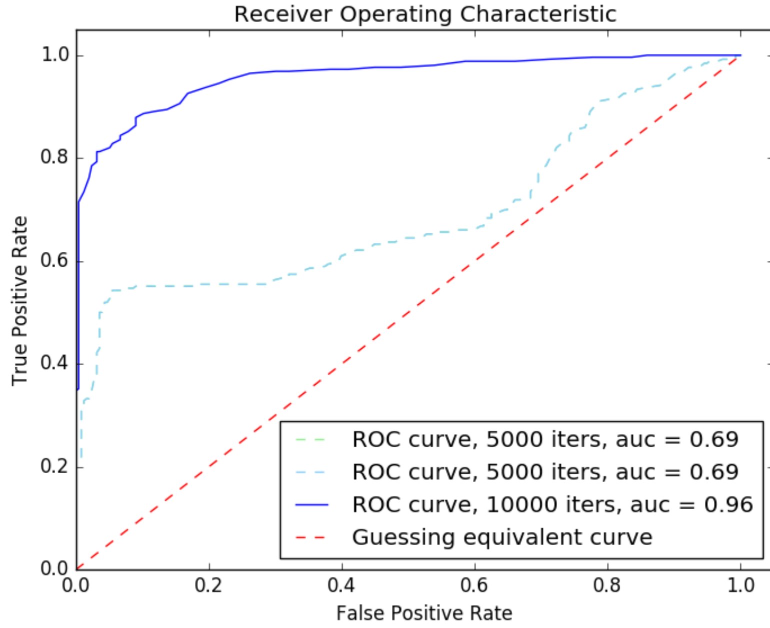
Figure 17: ROC plot for anomaly detection for all targets using a model with NOISE_WEIGHT set to 2.

In the reconstruction cost in equation 16, we included a penalty term for the norm of the generator input z used for the reconstruction. Results reported above have used settings where this penalty term is not used during reconstruction. From theoretical considerations, we expected this term to improve anomaly detection for some value of the constant C_2. Despite testing a large range of values for this constant, we could not find any value that led to better results than removing the term altogether. The only discernible trend was that results monotonically deteriorated as the contributions from this penalty term increased. Figure 17 shows one of these results.

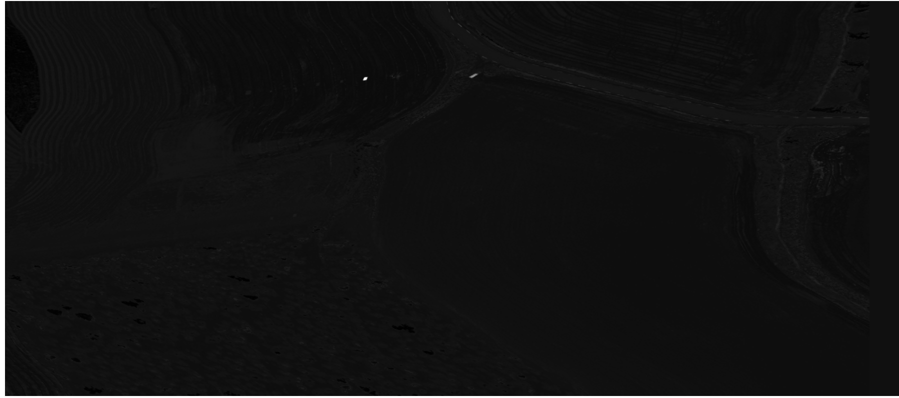
In hindsight, knowing that the unpenalized reconstruction method suffer more from failing to reconstruct samples in the background than from being able to reconstruct anomalies with high fidelity, this behavior makes sense and there is no need for this penalty term. This does not necessarily carry over to more expressive generators, where such a noise unlikelihood term could indeed serve a function.

We also ran a large number of experiments with increased dimension of the generator input z and the overall model complexity (DIM) without any noticeable improvement in the results. There were few discernible patterns in how these parameters affected the anomaly detection process, except that more complicated models increased the training and reconstruction time while increasing the levels of noise from failed reconstruction minimization.

5.3 Discriminator based anomaly detection



(a) Simplified ROC plot. Note the linear x-axis.



(b) Corresponding heatmap.

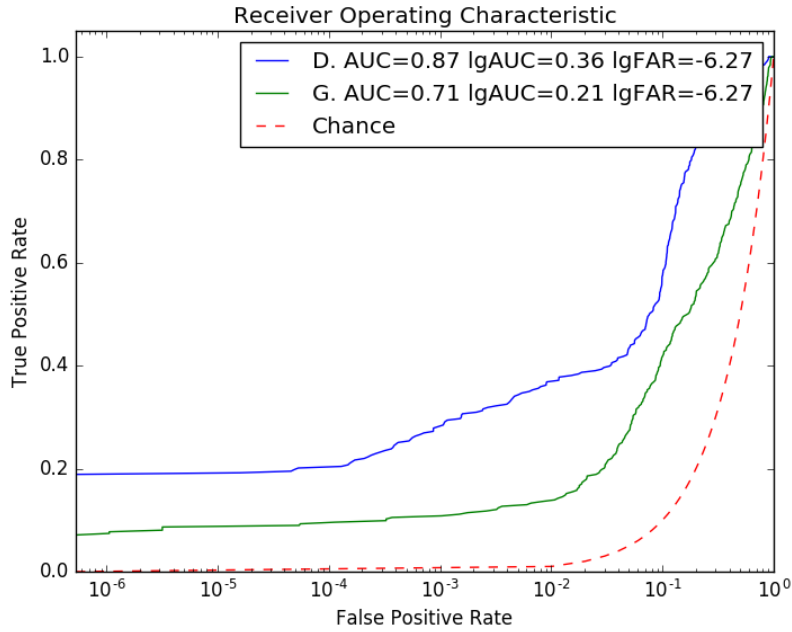
Figure 18: Discriminator based anomaly detection using fine-tuned model and hyperparameters.

There are numerous reasons for why achieving anomaly detection using only the discriminator would be preferable. The reconstruction cost approach introduce a number of complications, such as having to calculate how similar a target image and a reconstructed image are and a method for minimizing with respect to the generator input. Additionally, the reconstruction method is much more time consuming and has the unfortunate property that the most time consuming part of the process is applying the model, not training it.

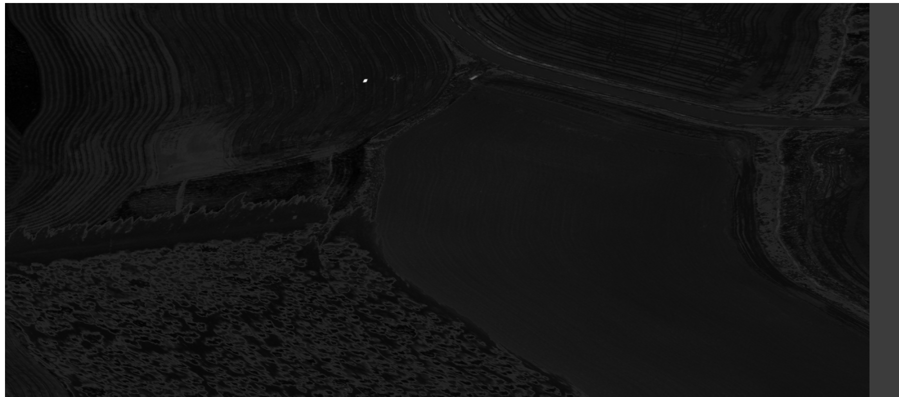
Tuning parameters, the training process and parts of the network design very carefully[6],

[6] Unlike all the generator models reported, this discriminator based detection is achieved by making significant modifications to the training cost function, reducing the number of discriminator training iterations per generator training iteration to 1 and modifying activation functions in the network.

it seems to be possible to obtain good anomaly detection results using just the generator, as shown in figure 18. Nonetheless, we gave up on this approach because we could never make it reliably good. A small change in any of a large number of hyperparameters would cause the entire training process to collapse. The generator based approach, on the other hand, is very stable.



(a) Simplified ROC plot.



(b) Corresponding heatmap.

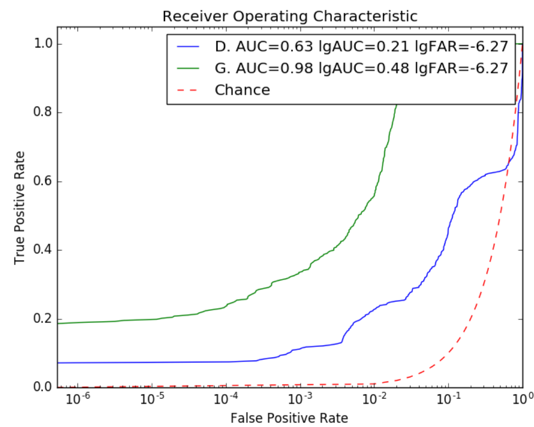
Figure 19: Discriminator based anomaly detection using the baseline model after training for half as many iterations and with NOISE_DIM set to 64.

In figure 19 we show the results for discriminator detection using the baseline model with a shorter training process and much higher dimension for z. As can be seen in the discriminator heatmap, T10 is correctly labeled as an obvious anomaly, unlike in figure 13, which contributes to the much better ROC curve and AUC score for this discriminator. All targets excepts T3 have at least a faint contrast against their local background in the heatmap, but there are a large number of pixels outside of the target regions with comparatively high brightness,

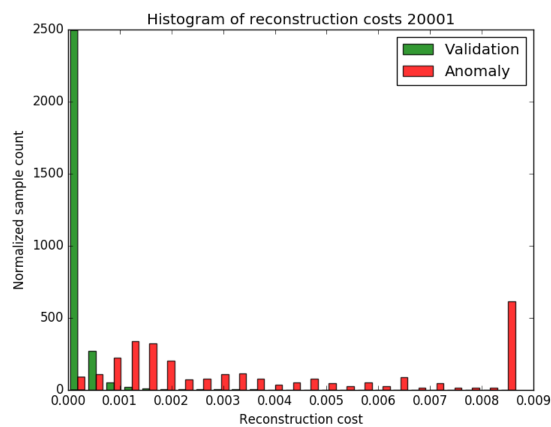
particularly along contours in the image. This is a reasonable failure mode, because edges between different background types will have different mixed contributions from each of the backgrounds, resulting in signal signatures that are unusual in the dataset. $AUC = 0.87$ is considerably worse than the baseline generator method achieving $AUC = 0.91$, but the discriminator method has a *better* value for logAUC due to better early detection. Note also that this detection method is orders of magnitude faster than the generator reconstruction method and can be run in a few minutes.

It is interesting that the discriminator detects anomalies far better early on in the training, at a point where the generator has not yet converged towards its optimum. We speculate that this might be due to effects discussed in section 3.5 — earlier in the training process, the generator is of poorer quality and stresses the discriminator with outputs that differ from the true data distribution in interesting ways, effectively training the discriminator to detect deviations from the true distribution. Later on in training, as the generator converges, the discriminator becomes hyperspecialized to detect the failure modes of the converged generator and loses its ability to serve as a general anomaly detector. In our experience, drawing the generator input z from higher dimensional space helps avoiding this failure mod in part by making the generator more difficult to train. It also make the reconstruction minimization process more unstable.

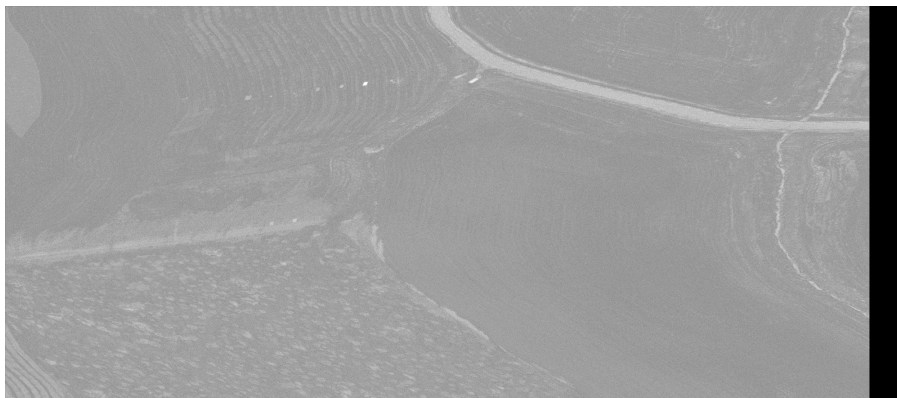
5.4 Single pixel results with time intensive reconstruction



(a) ROC plot



(b) Corresponding histogram of reconstruction costs. Higher values for the reconstruction cost have been truncated to improve resolution for lower values. Samples from outside of targets labeled *validation*.



(c) Corresponding heatmap.

Figure 20: Anomaly detection for all targets using a model with MINIMIZE_RUNS and MINIMIZE_ITERATIONS set to 16 and 100 respectively.

Figre 20 shows greatly improved results for the baseline model when the reconstruction minimization is done more thoroughly. The resulting heatmap is much less noisy because much fewer pixels wind up reconstructed poorly. The AUC is improved to a respectable 0.98 and all but two targets (T3 and T4) can be discerned against their surroundings The number of initial z vectors tested is multiplied by 4 and the number of minimization iterations multiplied by 2. The resulting running time for the whole image is 20 hours. This demonstrates that the generator has learned more about the data distribution that can be used for anomaly detection than indicated by the results in figure 12, which is held back by the sub-optimal performance of its reconstruction cost minimization. Figure 20 also shows a histogram for the reconstruction costs that demonstrates clear separation between anomalies and background in the vast majority of the cases.

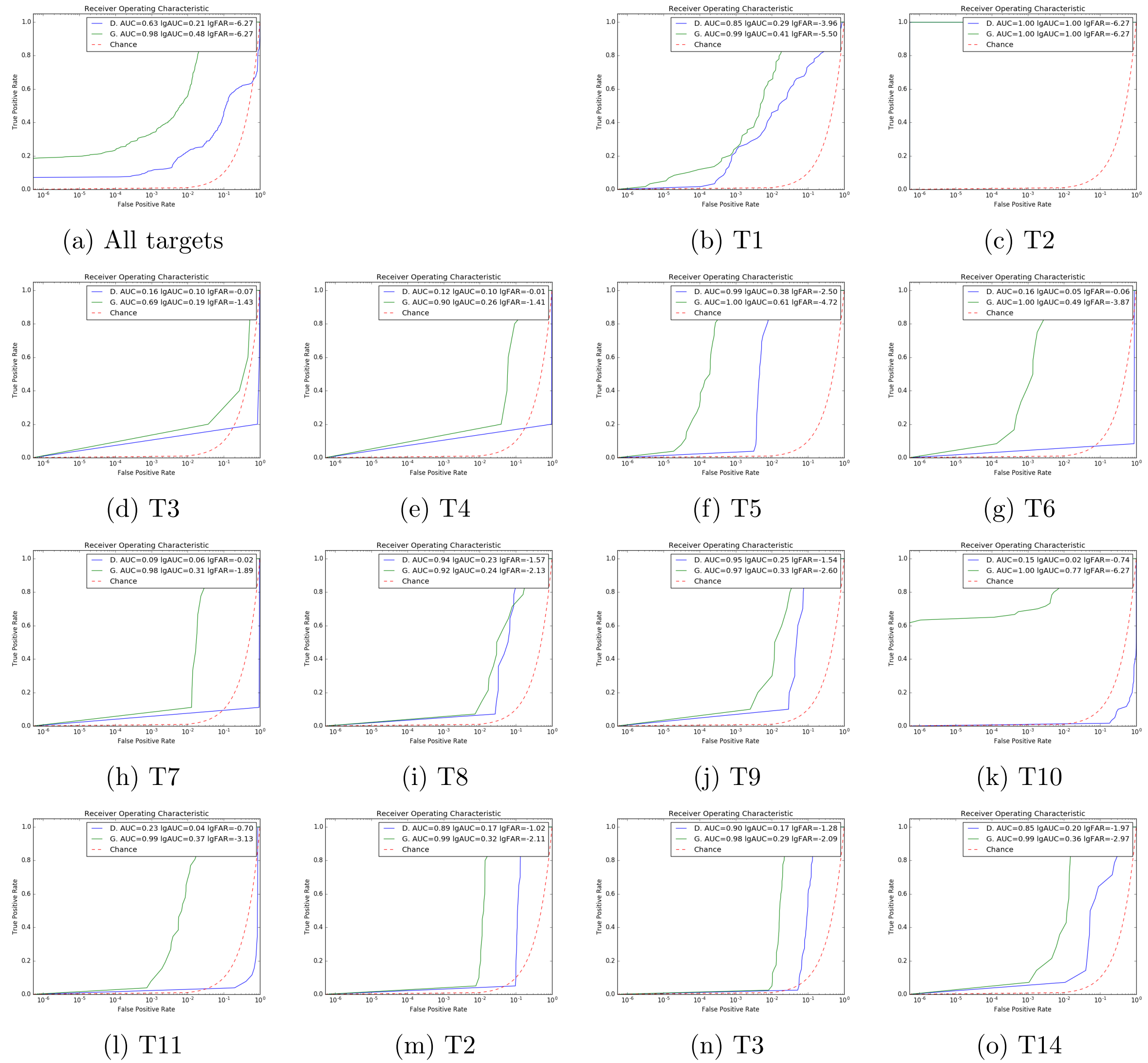
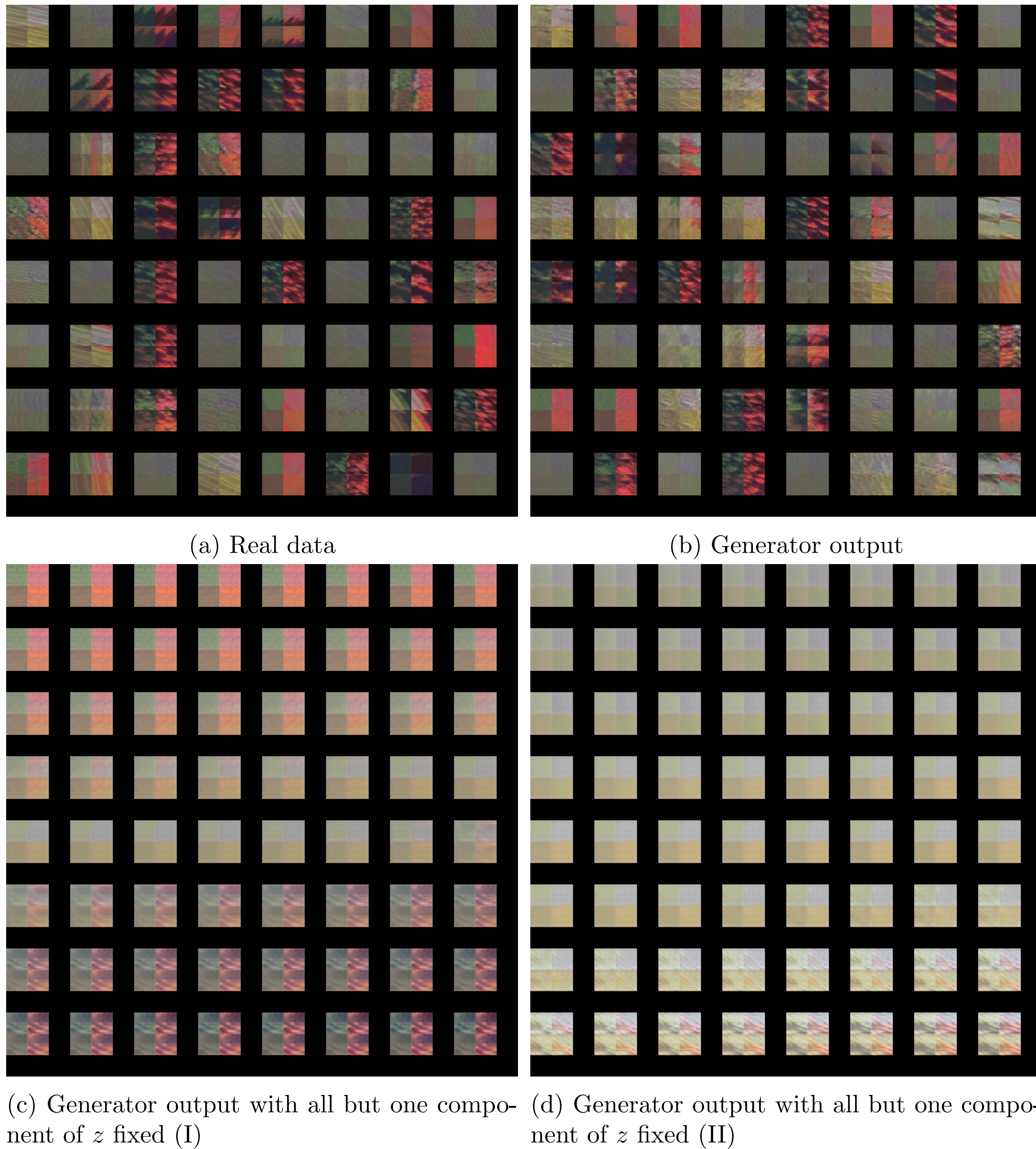


Figure 21: Matrix of ROC plots for each individual target.

We show the anomaly detection results for individual targets in figure 21. The results can be compared with previous work on the same dataset[4, 12]. The only outright poor AUC score is for T3, which is consistently the most difficult target for other methods as well and does not

stand out to the human eye. For most of the other targets, the AUC score is nearly saturated such that we must instead consider the logAUC scores.

5.5 Larger window sizes



(a) Real data

(b) Generator output

(c) Generator output with all but one component of z fixed (I)

(d) Generator output with all but one component of z fixed (II)

Figure 22: Visualization of generator output for 32×32 images with 16 spectral bands. Each subimage consists of four different RGB-representations of the hyperspectral images.

We did some preliminary work with larger spatial resolution, applying the GAN to actual subimages from the hyperspectral dataset rather than single pixels. Figure 22 shows the trained generator, demonstrating good visual fidelity and smooth variations in generator output as we vary a single component of the generator input z. While the generator seems to perform well, we had no success with discriminator based anomaly detection for larger windows, and the

reconstruction process which is time intensive even when working pixel by pixel is infeasible for the whole dataset in this case.

Early tests of reconstruction cost for 5×5 windows achieved $AUC = 0.80$, which is much worse than the pixel by pixel method, and introduce an assortment of problems: we cannot exclude the data outside of the inner but inside the outer masks of the targets; the relative portion of a image that contains an anomaly is small for large windows and depends on how the anomaly lines up with image windows; the reconstruction cost of a subimage seemed more dependent on its visual complexity than the presence or absence of anomalies or the visual quality of the reconstruction, with smoother images having lower cost.

6 Conclusion and further work

We have demonstrated that generative adversarial networks can be trained on the 20-channel hyperspectral Bjoerkelangen dataset. Generators have been shown to produce visually convincing outputs for image sizes of 32×32 pixels, but we have not managed to achieve any worthwhile results for anomaly detection with larger subimages. It is not clear whether this has to do with the trained networks themselves or with the additional challenges introduced by using a larger window size, such as the L2-distance between images being a poor metric for larger images and anomalies making up a relatively smaller portion of the sample images.

We have run extensive tests for single-pixel subimages that demonstrate that the scalar output from the discriminator network is difficult to use for anomaly detection, because the method is highly unreliable and requires very careful tuning of parameters to produce reasonable results. Nevertheless, we have been able to produce some promising results for discriminator based anomaly detection. Aside from tuning the hyperparameters, the discriminator detection method is much simpler to implement and is orders of magnitude faster to apply to a dataset.

We consider it likely that this method can be modified to work better and more reliably and would recommend further study in this vein. One suggestion is to repurpose the discriminator's adversary from a generator to a dedicated *sparring partner* for the discriminator network. We hypothesize that such a sparring partner would need to strike a balance between generating convincing samples and generating samples that differ from the real distribution such that the discriminator does not lose its sensitivity to actual anomalies.

We have also designed and implemented a method that uses the generator's ability to generate close matches to a sample in order to detect anomalies. This method shows very good and highly reliable anomaly detection. While the results are not quite as good as previous, highly developed statistical methods such as the multinormal mixture model, they are better than can be expected from a proof of concept project. It is possible that a similar, more refined version of this method can achieve results for this problem with real world applications. This anomaly detection method also works for entirely unsupervised learning.

It is not meaningful to evaluate the visual quality of samples from the single pixel-generator. Instead, we have demonstrated that the generator is capable of reconstructing the entire input image with reasonable fidelity, and that its fidelity is poorer and the resulting reconstruction cost greater for anomalies than for the background. We have also demonstrated an unfortunate effect, where our generator fails to reconstruct the road in the dataset properly. It is clear that the general principles of reconstruction cost are sound. The generator would work better if it could be made more sensitive to the less common modes in the dataset without simultaneously learning to reproduce the yet less frequent anomalies.

The weakest link in our implementation is the minimization procedure that finds the generator input that results in the output with the best match to a given sample. Our procedure is slow and does not consistently find the global minimum. This makes it impossible to achieve real time generator based anomaly detection. The other key area for improvement is to make the method work for larger subimages instead of single pixels, perhaps as small as 5×5 pixels. This could help with some of the problems with our methods, such as false alarms for image contours. Unfortunately, the current minimization method is much too slow when applied to more complicated models. Another problem that could be studied in more detail is normalization techniques that do not interfere with the possibility of using the generator to reconstruct samples.

The hyperparameters that we have found produce the best result for our anomaly detector are somewhat counter-intuitive. As expected, a more thorough minimization results in better

results. On the other hand, the ideal value for the dimension of the generator input is surprisingly low and we see no returns from increasing the complexity of the relatively simple model. It is likely that the effects of these parameters on the adversarial training of the networks and the minimization procedure is more important than the expressiveness of the networks themselves.

Finally, we have found that the additional term in the reconstruction cost, where we penalize reconstructions that make use of unlikely input vectors for the generator, is not helpful. Visualizations of the reconstructions show that this term in practice is a solution to a problem that does not exist, in that our generator is unable to reconstruct anomalies even without any penalty term at all. This does not necessarily hold true for a more expressive generator.

References

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] M. Arjovsky, S. Chintala, and L. Bottou. Wasserstein GAN. *ArXiv e-prints*, January 2017.

[3] William Blackstone. *Commentaries on the Laws of England Volume I.* Clarendon Press, 1765.

[4] Dirk Borghys, Ingebjørg Kåsen, Véronique Achard, and Christiaan Perneel. Hyperspectral anomaly detection: Comparative evaluation in scenes with diverse complexity. *JECE*, 2012:5:5–5:5, January 2012.

[5] Alfredo Canziani, Adam Paszke, and Eugenio Culurciello. An analysis of deep neural network models for practical applications. *CoRR*, abs/1605.07678, 2016.

[6] Tom Fawcett. Introduction to roc analysis. 27:861–874, 06 2006.

[7] Tom Fawcett and Foster Provost. Adaptive fraud detection. *Data Mining and Knowledge Discovery*, 1(3):291–316, Sep 1997.

[8] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep sparse rectifier neural networks. In Geoffrey Gordon, David Dunson, and Miroslav Dudík, editors, *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, volume 15 of *Proceedings of Machine Learning Research*, pages 315–323, Fort Lauderdale, FL, USA, 11–13 Apr 2011. PMLR.

[9] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio. Generative Adversarial Networks. *ArXiv e-prints*, June 2014.

[10] Ian Goodfellow. Can the generative adversarial network useful for outlier detection and outlier explanation in a high dimentional numerical data? Quora reply.

[11] Ishaan Gulrajani, Faruk Ahmed, Martín Arjovsky, Vincent Dumoulin, and Aaron C. Courville. Improved training of wasserstein gans. *CoRR*, abs/1704.00028, 2017.

[12] Trym Haavardsholm, Amela Kavara, Ingebjorg Kasen, and Torbjorn Skauli. Improving anomaly detection with multinormal mixture models in shadow. pages 5478–5481, 07 2012.

[13] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.

[14] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.

[15] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, Nov 1998.

[16] Yann LeCun, Y Bengio, and Geoffrey Hinton. Deep learning. 521:436–44, 05 2015.

[17] Thomas M. Mitchell. *Machine Learning*. McGraw-Hill, Inc., New York, NY, USA, 1 edition, 1997.

[18] Takeru Miyato, Toshiki Kataoka, Masanori Koyama, and Yuichi Yoshida. Spectral normalization for generative adversarial networks. In *International Conference on Learning Representations*, 2018.

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

[20] Henning Petzka, Asja Fischer, and Denis Lukovnikov. On the regularization of wasserstein GANs. In *International Conference on Learning Representations*, 2018.

[21] Guido Rossum. Python reference manual. Technical report, Amsterdam, The Netherlands, The Netherlands, 1995.

[22] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323:533 EP –, Oct 1986.

[23] Tim Salimans, Ian J. Goodfellow, Wojciech Zaremba, Vicki Cheung, Alec Radford, and Xi Chen. Improved techniques for training gans. *CoRR*, abs/1606.03498, 2016.

[24] Thomas Schlegl, Philipp Seeböck, Sebastian M. Waldstein, Ursula Schmidt-Erfurth, and Georg Langs. Unsupervised anomaly detection with generative adversarial networks to guide marker discovery. *CoRR*, abs/1703.05921, 2017.

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

[26] S. van der Walt, S. C. Colbert, and G. Varoquaux. The numpy array: A structure for efficient numerical computation. *Computing in Science Engineering*, 13(2):22–30, March 2011.

[27] Izhar Wallach, Michael Dzamba, and Abraham Heifets. Atomnet: A deep convolutional neural network for bioactivity prediction in structure-based drug discovery. *CoRR*, abs/1510.02855, 2015.