Machine Learning for Zero Defect Manufacturing

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Embedded Systems

eingereicht von

José Matías Vásquez Lobos
Matrikelnummer 11742193

an der Fakultät für Elektrotechnik
der Technischen Universität Wien

Betreuung: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Markus Vincze
Mitwirkung: Dr. techn. Amirreza Baghbanpourasl

Wien, 1. Oktober 2022

José Matías Vásquez Lobos	Markus Vincze

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

TECHNISCHE UNIVERSITÄT WIEN
Vienna University of Technology

PROFACTOR®

ACIN
AUTOMATION & CONTROL INSTITUTE
INSTITUT FÜR AUTOMATISIERUNGS- & REGELUNGSTECHNIK

Machine Learning for Zero Defect Manufacturing

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Embedded Systems

by

José Matías Vásquez Lobos
Registration Number 11742193

to the Faculty of Electrical Engineering
at the TU Wien

Advisor: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Markus Vincze
Assistance: Dr. techn. Amirreza Baghbanpourasl

Vienna, 1st October, 2022

José Matías Vásquez Lobos

Markus Vincze

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

José Matías Vásquez Lobos

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 1. Oktober 2022

José Matías Vásquez Lobos

Acknowledgements

I would like to thank Profactor for funding my work through an internship and especially Amir, who followed every step of this thesis and guided me towards a better understanding. I would also like to thank Markus Vincze for his insight and continuous help to finish this project.

I'm extremely grateful to Alex Amesmann and Eva Wilhelm who have supported me through the years and accepted me into their family as one of their own.

Words cannot express my gratitude to Hannah, who has been there for me since the start and has been the greatest partner and support I could have ever asked for. She and her family have been without a doubt an enormous help to achieve my goals and feel at home.

Lastly and with the deepest appreciation, I would like to thank my family, especially my parents. Their belief in me and unconditional support have been crucial to achieving this milestone. I will be forever thankful to them for raising me to value every opportunity given to me and use them to seek a better future.

Matías Vásquez

Vienna, July 2022

Kurzfassung

Die verarbeitende Industrie nutzt wertvolle natürliche Ressourcen, und unabhängig davon, ob niedrigere Produktionskosten oder Umweltschutz im Vordergrund stehen, ist eine bessere Effizienz durch weniger Defekte stets erwünscht.

Der Einsatz neuer Technologien kann helfen, dieses Ziel zu erreichen und die Ressourcen besser zu nutzen. Eine Möglichkeit, Technologie zu diesem Zweck einzusetzen, ist die visuelle Überprüfung der Objekte mit Hilfe von Kameras. Algorithmen für Machine Vision und Machine Learning können eingesetzt werden, um mögliche Fehler zu erkennen.

Eines der Hauptprobleme bei der Verwendung von Machine Learning zur Fehlererkennung ist die Datenerfassung. Selbst wenn einige Tausend Bilder gesammelt werden, reicht dies möglicherweise nicht aus, um ein Modell von Grund auf zu trainieren und die erforderliche Leistung für den Einsatz in der Produktion zu erzielen. Feature Extraction hingegen ermöglicht die Wiederverwendung vortrainierter Modelle, was zu besseren Ergebnissen führt.

Diese Arbeit stellt eine Trainingsstrategie vor, bei der die vortrainierten Modelle vollständig trainiert werden, um ihre Parameter für den gewünschten Datensatz anzupassen, selbst wenn dieser neue Datensatz nicht mit den ursprünglichen Daten übereinstimmt, für die das Modell zuvor trainiert wurde. Üblicherweise werden die ersten Schichten eingefroren[1], um die grobe Feature Extraction zu bewahren, und nur die nachfolgenden Schichten zu trainieren, die sich auf feinere Details konzentrieren könnten. Indem jeder Parameter des Modells mit einer sehr kleinen Lernrate trainiert wird, wurden ähnliche Ergebnisse wie üblicherweise erzielt. Der Mehrwert liegt in der Analyse der Activation Maps der verschiedenen Strategien. Diese werden dann mit Annotated Maps mit den Orten der Defekte verglichen, die beim Training nicht vorhanden sind. Es wurde festgestellt, dass sich das Modell durch das Training aller Parameter stärker auf die möglichen Defekte im Bild konzentrieren kann, wodurch das Modell besser interpretierbar wird.

[1] Die Anzahl der Schichten ist abhängig von der Ähnlichkeit der Datensätze, der Größe des neuen Datensatzes und der Größe des Modells.

Abstract

The manufacturing industry makes use of valuable natural resources and regardless of having lower production costs or the environment as a main reason, better efficiency through less defects is always desired. The use of new technologies can help achieve this goal and make better use of the resources.

One way of applying technology for this purpose is by visually inspecting the objects using cameras. Machine Vision and Machine Learning algorithms can be implemented to detect possible imperfections.

Among the main problems for using Machine Learning for defect detection is the recollection of data. Even when collecting a couple of thousands of images, this might not be enough to train a model from scratch and have the necessary performance to be deployed into production. Feature extraction on the other hand allows the reuse of pretrained models, which leads to better results.

This work presents a training strategy in which the pretrained models are fully trained to accommodate their parameters for the desired dataset, even when this new dataset is dissimilar to the original data for which the model was previously trained. The standard practice is to freeze the first few layers[2] to preserve the coarse feature extractions and only train the later layers which might focus on more finer details. By letting every parameter of the model be trained with a very small learning rate, similar results to the standard practice were achieved. The added value is when analyzing the activation maps of the different strategies. These are then compared to annotated maps with the locations of the defects, which are not present during training. It was determined that training every parameter allows the model to focus more on the possible defects present in the image, making the model more interpretable.

[2]The amount of layers will depend on the similarity of the datasets, the size of the new dataset and the size of the model.

Contents

1 Introduction

Zero Defect Manufacturing (ZDM) is a strategy seeking the reduction and avoidance of failures during production focusing either on the product (analysing the product) or the process (analysing the manufacturing equipment) [1]. Altough this concept was born within the US army during the Cold War to prevent human error in the weapon systems [1], the concept has been adapted to the general industry in order to reduce costs by diminishing any type of failure.

The evolution of Industry 4.0, with the increase in data available for machine learning techniques to work as desired, makes the concept of ZDM implementable [1]. Industry 4.0 has had a great impact on monitoring systems for pattern detection and detection of fault after the occurrence of failures, however, a significant challenge lies in the generation and collection of data [2].

The concept explored here is to apply Machine Learning (ML) for defect detection, such that it can be implemented during manufacturing. Instead of carefully designing Machine Vision algorithms and filters to detect the defects on a specific product, a ML model is trained to distinguish between images with and without defects. This way the model can be retrained when analyzing a new product, instead of manually designing the necessary feature extractions again.

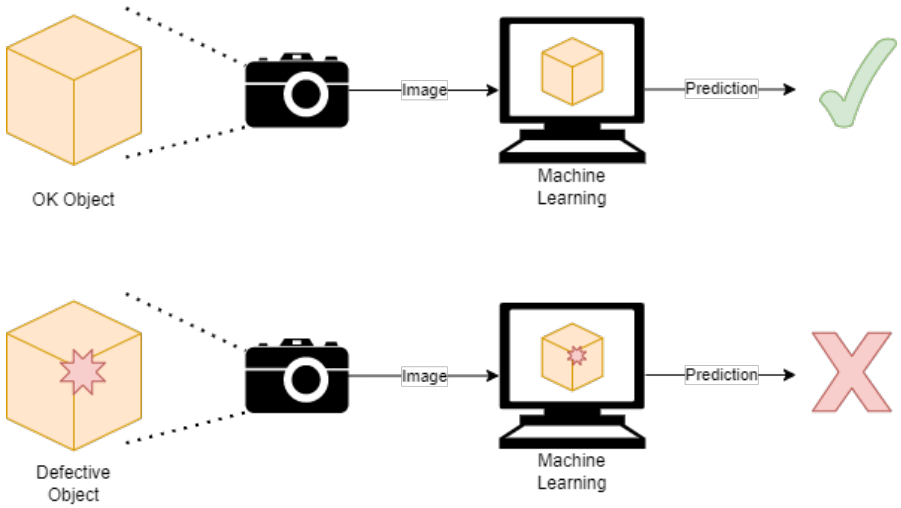


Figure 1.1: Defect detection with Machine Learning

1.1 Challenge

Machine Learning techniques require large amounts of data to generalize and extrapolate to unseen examples. If there is not enough data present, the models can memorize the data, which leads to a poor performance on new data.

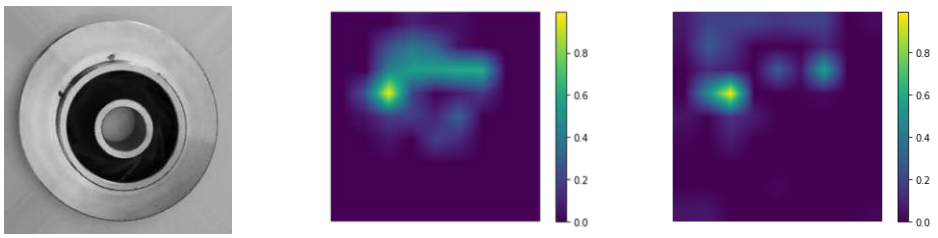
Even with enough data to train a model, the validation process refers to how well the model is trained to predict the result. Nevertheless, this does not justify the model and its predictions. In order to determine the responsible criteria for the decision and be able to justify the model, a way to interpret the model is necessary [3].

Teh et al. [4] propose a way of locating objects in weakly labeled images through the use of Attention Networks. This Neural Network takes a set of candidate regions of the original image and computes "attention scores" by analysing the number of contours inside a bounding box and the ones from the boxes overlapped.

A contemporary work to Teh et al. [4] was proposed by Selvaraju et al. [5], where activation maps show the most relevant sections of the image for the specific class predicted. This allows the analysis of the models in order to interpret what they might be taking into account to make the classification.

Given a small weakly annotated dataset, the challenge is to train a model to classify unseen images correctly, proving that the training causes the model to focus on the defects present in the image. As a baseline, the same models with the pretrained weights are used, when available. The goal is to increase the overlapping ratio between the most active regions of the class activation maps and the manually annotated defects.

In other words, not only do the different models have to learn to classify the images correctly, but the model has to be interpretable in such a way that it can be determined if the defects are key contributors for the classification.



(a) Image of Pump impeller with defects

(b) Before Fine Tuning

(c) After Fine Tuning

Figure 1.2: Example of the key features on a class activation map before and after fine tuning.

1.2 Contribution

The main goal of this study is to facilitate the visual detection of defects while manufacturing in order to take action during production and minimize the loss. For this purpose, many models where analyzed, where one relatively small model was trained from scratch, while the other seven models where fine tuned from publicly available pretrained models.

As a baseline for comparison for the fine tuned models, the same models were trained using the same hyperparameters and training conditions, only changing the amount of parameters frozen.

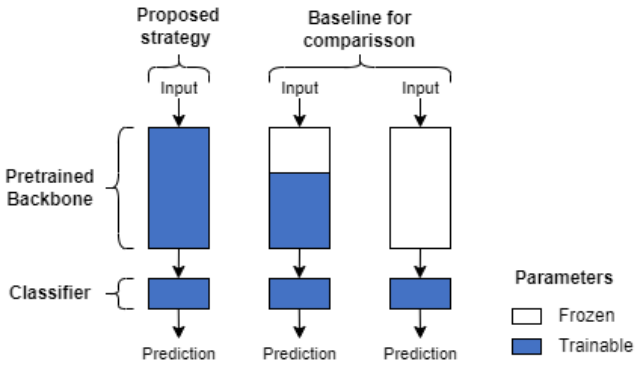


Figure 1.3: Proposed fine tuning strategy for small datasets that are dissimilar to the pretrained backbone's original dataset.

In addition to analyzing the performances of said architectures, the class activation maps are computed. This way, the architectures can be interpreted in order to determine if the defects present on the images are a key factor to classify the images as being defective or not.

As a way to analyze the differences between the different training strategies, two new metrics are introduced (Intersection over Prediction and Intersection over Ground Truth). With these metrics, it can be analyzed with more detail what the strengths and weaknesses of each model are. If one wants to make sure that the visual interpretation of a model recognizes every defect present on an image, then the Intersection over Ground Truth has to be maximized. If, on the other hand, one wants to make sure that the visual interpretation for the prediction of the model lies inside of the defects present, then Intersection over Prediction is the right metric.

$$IoP = \frac{\text{Intersection}}{\text{Prediction}} = \frac{\text{Ground Truth} \cap \text{Prediction}}{\text{Ground Truth} \cup \text{Prediction}} \tag{1.1}$$

$$IoGT = \frac{\text{Intersection}}{\text{Ground Truth}} = \frac{\text{Ground Truth} \cap \text{Prediction}}{\text{Ground Truth} \cup \text{Prediction}} \tag{1.2}$$

1.3 Results

While training, validating and testing the different models with the different training strategies, all of the unfrozen models showed a significant improvement from the fully frozen. From an average 78.45% accuracy among the different models, it increased to 97.78%. During validating this increment was from 82.39% to 99.12% accuracy and for testing from 90.40% to 99.80%. Similar results were observed among every metric computed.

All of the metrics for all of the models remained consistent between the partially frozen and the unfrozen models. Both of these strategies performed better than the fully frozen having excellent metrics when classifying the different images, but neither of them showed a clear advantage over the other. Averaging the different models, every metric remained below a 0.2% difference between partially frozen and unfrozen.

The different models were also interpreted using GradCAM for the CNN models and extracting the Attention Maps for the Transformer models. Both of these methods successfully detected the relevant features for the classification, demonstrating that the models did learn to focus on the defects for the correct classification.

Interpreting the models showed that more freedom to modify the parameters meant more accuracy towards detecting the defects. The Intersection over Union, Intersection over Prediction and Intersection over Ground Truth improved for almost every model when leaving every parameter unfrozen. The only model that did not follow this trend was the DINO architecture. This suggests that unfreezing the parameters allows for an easier adjustment of the parameters to detect the defects, while no decrease in the performance was noticed.

Leaving every parameter unfrozen and using a very small learning rate leads to more interpretable results that have similar performances to partially frozen models. Leaving every parameter unfrozen also alleviates the burden of deciding up until where the parameters shall remain frozen.

1.4 Thesis Outline

The way this thesis is structured is as follows. First, the necessary building blocks for deep learning are explained, so that later the individual architectures can be presented. Here, the explanation is divided into two different types of architectures, Convolutional Neural Networks (CNNs) and Transformers.

The CNN architectures are only superficially explained, since these are the more common and known architectures. These have been explored for a longer time and count with many resources that explain them.

Transformers on the other side, are newer and even though there are many resources explaining them in detail, it was deemed necessary to go into more detail to understand and explain their functionality. Here, the introduction of Transformers to Natural Language Processing (NLP) is discussed, then the adaptation to Machine Vision, as well as one variation of the Vision Transformer.

This is then followed by a description of the dataset used. There is also a brief explanation of how the code might be adapted to other datasets in order to implement

this work. This is done with the intention that the work can be reproduced and used for different datasets.

After introducing the different models and dataset used, the necessary sections for training a model are explained, such as the *Loss*, the *Optimizer* and how the training actually occurs. The training, validation and testing results are presented here.

The *Interpretability* section explains how the different models are analyzed to see if the defects are responsible for the correct classification. This will help describe the functionality of the models, in order not to treat them as a black box, but rather understand what the model "sees".

Finally, the conclusion gives the final remarks on the results shown both during the training and the interpretation of the models.

2 Related Work

The field of Deep Learning has had a trending boom in the recent years. This has been specially the case, since the general public is aware of their implementation in daily activities used such as social media and streaming platforms. However, the founding principles that make this possible date back to the second half of the last century.

Each architecture is composed of different layers that perform different tasks. Some layers contain trainable parameters which need to be tuned in order to achieve the desired task. These have been developed through the years. Alom et al. [6] have encompassed these developments into this brief history of Deep Neural Networks[1]:

- 1943: McCulloch & Pitts show that neurons can be combined to construct a Turing machine (using ANDs, ORs, & NOTs) [8].
- 1958: Rosenblatt shows that perceptron's will converge if what they are trying to learn can be represented [9].
- 1969: Minsky & Papert show the limitations of perceptron's, killing research in neural networks for a decade [10] also known as the first AI winter[2,3].
- 1985: The backpropagation algorithm by Ackley et al. [11] revitalizes the field.
- 1988: Neocognitron: a hierarchical neural network capable of visual pattern recognition [12].
- 1998: CNNs with Backpropagation for document analysis by Yan LeCun [13].
- 2006: The Hinton lab solves the training problem for DNNs [14, 15].
- 2012 : AlexNet by Alex Krizhevesky in 2012 [16]

In this section, the building blocks for creating a Deep Neural Network are presented. The section is then divided into two main subsections, architectures consisting mainly of Convolutions, also known as Convolutional Neural Networks (CNNs) and the other subsection consists of Transformers.

CNNs are a well known concept with many resources that explain them, therefore, the explanation here limits itself to the novelty of what each architecture might have brought

[1] Emmert-Streib et al. [7] have a more complete time line of key developments until the year 2019.
[2] https://towardsdatascience.com/history-of-the-first-ai-winter-6f8c2186f80b
[3] https://towardsdatascience.com/history-of-the-second-ai-winter-406f18789d45

to the field and not digging too deep into them. Transformers on the other hand, are a more recently introduced concept and thus, the explanation goes deeper, describing in detail how they work.

These building blocks are also called layers and can be stacked on top of each other to form architectures.

2.1 Layers used

The layers described here are the necessary building blocks used for the architectures described later. They can be arranged in almost any way that it is needed, as long as the graph does not create a loop. Emmert-Streib et al. [7] compare this to building with Lego blocks.

2.1.1 Linear

The first concept to grasp when trying to understand Neural Networks is the Artificial Neuron. Figure 2.1 shows the mathematical representation of an artificial neuron with multiple inputs and one output.

The main idea of the artificial neuron is that after receiving a vector input, x, each element, x_i, is weighted by a corresponding weight, w_i. The results are added up together along with a bias term, b. The sum is then passed through an activation function ϕ to get the output y [7].

$$y = \phi(w^T \cdot x + b) \tag{2.1}$$

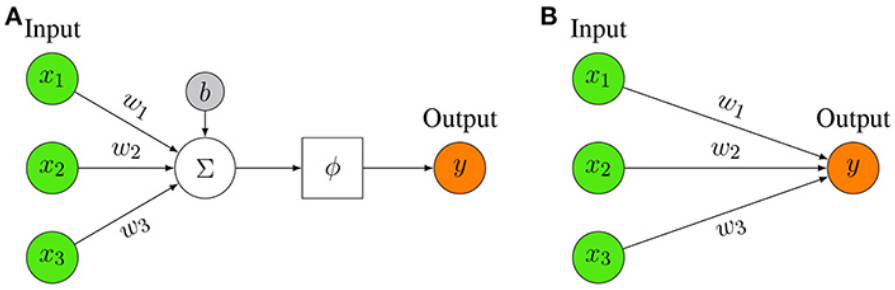


Figure 2.1: **(A)** Representation of a mathematical artificial neuron model (left). **(B)** Simplified representation, depicting only the key elements. [7]

If the activation function, ϕ, is the Heaviside step function, then the artificial neuron corresponds to the original *perceptron* design proposed by Rosenblatt [17].

A linear layer consists of multiple artificial neurons in parallel, each processing the same input with their respective weights and biases.

2.1.2 Multi-Layer Perceptron

The input mentioned in the linear layer might refer to the actual input of the architecture, or just refer to the output of a previous layer. In order to create Neural Networks, the layers have to be connected to each other.

One simple way to create a Neural Network is by connecting multiple linear layers, where the activations can be linear or non-linear [18]. These configurations receive multiple names, such as *Fully Connected (FC)*, *Feedforward Neural Network (FFNN)* or *Multi-Layer Perceptron (MLP)*.

The very first layer is called the *input layer* and the last is the *output layer*, everything in between is a *hidden layer*. The width refers to the amount of parallel neurons in a layer, the depth on the other hand refers to the amount of layers in the network (without counting the input layer) [7].

2.1.3 Convolutional Layer

For *Linear layers* it was mentioned that the input, x, has to be a vector. When analysing images, in order to feed the pixels to the network, the information has to be flattened, thus losing the spatial information. This means that neighbouring pixels might now be far away in the sequence, making it more difficult for the network to infer their connection.

To address this problem, convolutional layers use kernels and slide them across the input. This kernel acts as a window, where the result is a weighted sum of the neighbouring input values and the weight is given by the kernel. Figure 2.2 shows a simple example of an input being multiplied by a kernel.

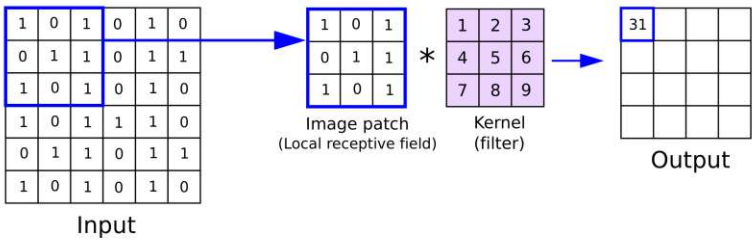


Figure 2.2: Convolutional Layer: Multiplying an input by a kernel. [4]

The size of the kernel determines how many neighbouring features are included in the calculation, this is also known as the *receptive field*. There are also other parameters that can be adjusted to the needs, such as the stride (number of pixels when moving the kernel to get the next calculation) and the padding (extra pixels added on the borders). All of these parameters determine how big the output (from this layer) will be, thus creating an output that is equal in size to the input, or smaller.

2.1.4 Batch normalization

The batch normalization layer computes the mean and variance to normalize the batch along the Channel dimension.

[4] Image taken from `https://anhreynolds.com/blogs/cnn.html`

Algorithm 1 Batch Normalization [19]

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i$$

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

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

$$y_i \leftarrow \gamma \hat{x}_i + \beta$$

2.1.5 Dropout

Dropout is a simple technique to avoid overfitting when training [20]. This regularizer drops the connections to neurons, which are randomly selected on each iteration. This forces the network to not rely on specific paths, but learn to generalize.

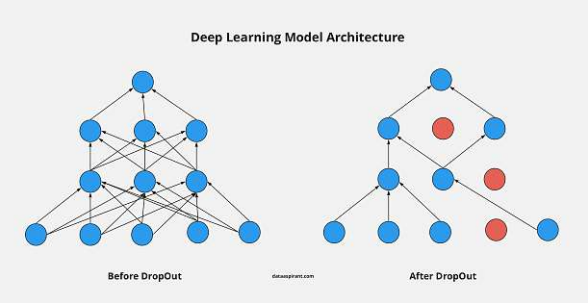


Figure 2.3: Dropout [5]

2.1.6 Max Pool

The max pool layer selects the maximum values inside a windows of a chosen size. The windows is then shifted by a stride, thus selecting a new maximum to create the output [21]. Figure 2.4 shows the function of this layer with a window size of 2 and a stride of 2, so that the windows do not overlapp.

[5] Image taken from: `https://dataaspirant.com/8-deep-learning-dropout/`

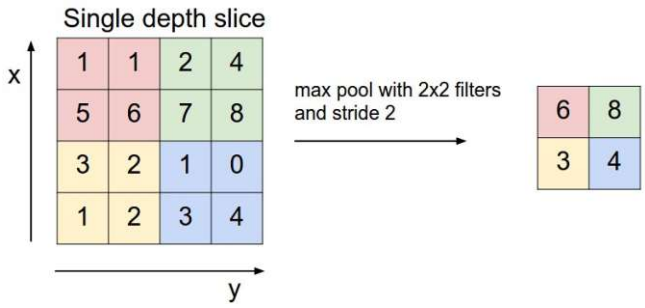


Figure 2.4: MaxPool [6]

2.1.7 Activation Functions

Sigmoid

The sigmoid function maps every input to a value between 0 and 1. The highest rate of change, and thus where the function is most sensitive, is in the inflection point, where the input is 0.

$$Sigmoid(x) = \frac{1}{1+e^{-x}}, \qquad i = 1, ..., J \tag{2.2}$$

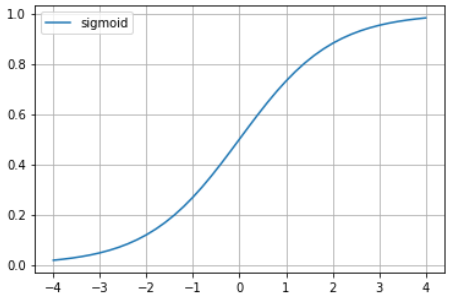


Figure 2.5: Sigmoid

Softmax

Given a vector $\vec{x}$, where each element x_i represents a class each element is normalized by the probability function given in Equation 2.3. Here, each element is taken as an exponent to the base e and divided by the sum of these new values. The exponentiation takes care of negative values, resulting in probabilities weighted by their original values, where the sum of the outputs is 1

$$Softmax(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{J} e^{x_j}}, \qquad i = 1, ..., J \tag{2.3}$$

[6] Image taken from: https://cs231n.github.io/convolutional-networks/#pool

ReLU

Rectified Linear Units is an activation function. Every positive input remains unchanged, whereas all the negative inputs are mapped to 0 [22].

$$f(x) = max(0, x) \tag{2.4}$$

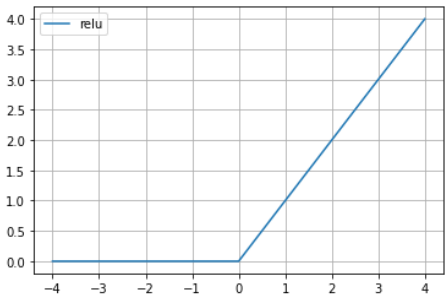


Figure 2.6: ReLU

GeLU

The Gaussian Error Linear Units[23] is an activation function that uses the standard Gaussian cumulative distribution.

$$GELU(x) = xP(X \leq x) = x\Phi(x) = x \cdot \frac{1}{2}\left[1 + erf\left(x/\sqrt{2}\right)\right] \tag{2.5}$$

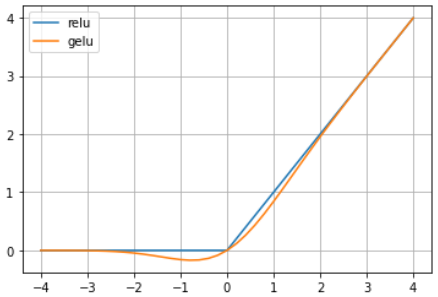


Figure 2.7: GeLU

2.2 Convolutional Neural Network

As the name suggests, the defining feature of these architectures is the (nonexclusive) use of Convolutional Layers. Convolutional Neural Networks have been the *go-to* architecture for image processing in the Deep Learning field since their breakthrough in 2015 with AlexNet. Although the concepts have been there for a longer time, the improvement in computational power was needed to get the ball rolling.

2.2.1 Luna Model

In order to get familiar with the framework used in this project (pytorch), the book "Deep Learning with PyTorch" by Stevens et al. [24] was used as a starting point. This book is divided into three main parts. First, an introduction to Deep Learning while explaining the core concepts of the framework to solve common tasks. Second, a specific task of image classification is solved, namely the use of CT scans from the LUNA (LUng Nodule Analysis) Grand Challenge in order to detect cancer. Finally, the book describes how to deploy the models.

While exploring a solution for the LUNA challenge, Stevens et al. [24] propose a Convolutional Neural Network and decided to call it the Luna Model. The dataset used consists of 3D scans of lungs with and without the presence of nodules and the position and diameter of these when present. Since the nature of the task is to classify images which share the overall structure and ultimately categorized by a defining feature, it was deemed relevant to adapt the architecture for the casting task. Figure 2.8 shows a general representation of the Luna Model. The original model was designed to process 3D images, thus the *BatchNorm*, *Convolutional* and *MaxPool* layers also are defined specifically for 3D data. For the adaptation it was only necessary to change these layers to the 2D version as well as the number of input features for the *Linear* layer.

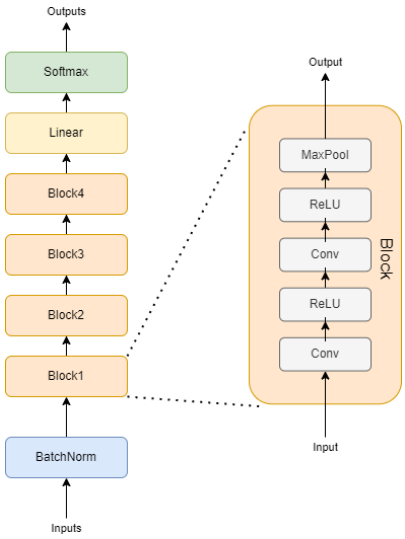


Figure 2.8: Luna Model architecture [24]

Figure 2.8 shows the main workflow of the architecture along with a definition for the *Block* used in it. It consists mainly of typical layers used for image analysis. After performing the corresponding adaptations, this model consists of 115,088 parameters to be trained. This was one of the key features for experimenting with this architecture, since it is significantly lower than other models and allows for training from scratch.

2.2.2 AlexNet

Krizhevsky et al. [16] introduced the AlexNet architecture, which won the 2012 ImageNet

Large Scale Visual Recognition Challenge (ILSVRC)[7]. The goal of this competition is to classify images from 10,000+ different classes from a subset of the 10M labeled images.

As it can be seen in Figure 2.9, the results from this architecture greatly improved on the results of past years by 10%. This image shows the winning architectures for the *top-5 error rate*. This means that the correct class must be among the top-5 predictions made by the model. However, the architecture did not only outperform the architectures from previous years, but also the ones competing in the same year. The second place for the 2012 competition achieved a *top-5 error rate* of 26% and was not based on a deep network.

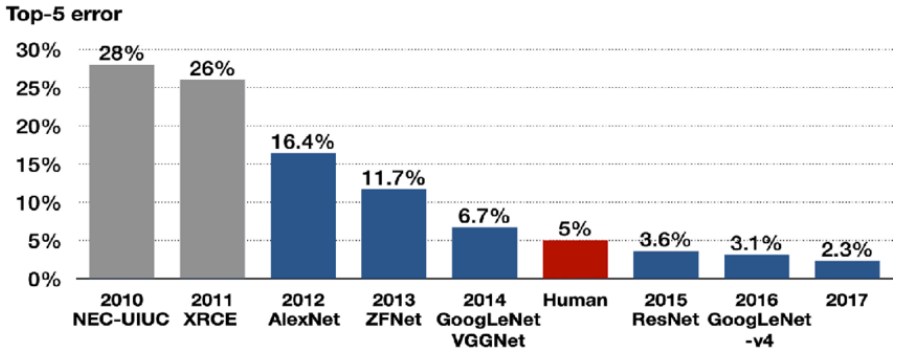


Figure 2.9: Algorithms that won the ILSVRC between 2010-2017. [25]

What Krizhevsky et al. [16] proposed was to make to make a deeper and wider CNN than LeNet. LeNet was introduced in the 1990's by Lecun et al. [26], but due to limited computation it was very complicated to implement until 2010. LeNet consisted of 2 convolutional layers, 2 sub-sampling layers, 2 fully connected layers and an output layer with Gaussian connection [6].

Figure 2.10 shows the architecture as presented in the paper. The first stage has one convolutional layer with a kernel of size $11x11$ followed by max-pooling operations with $3x3$ filters and stride of 2. The second stage performs the same operations, only the convolutional layer has a kernel of size $5x5$. The next three stages use convolutional layers with kernels of size $3x3$. This is then followed by two fully connected layers with dropout, ending with softmax[8].

This deeper architecture achieved state of the art results, setting a defining moment in the history of computer vision. Thanks to this improvement, many researchers shifted towards Deep Learning.

[7] https://www.image-net.org/challenges/LSVRC/2012/index.php

[8] Here is a good comparison between LeNet and AlexNet: https://en.wikipedia.org/wiki/AlexNet#/media/File:Comparison_image_neural_networks.svg

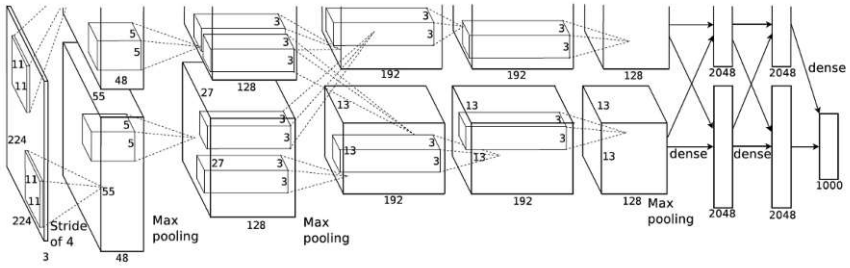


Figure 2.10: AlexNet architecture [16]

2.2.3 ResNet

After the accomplishment of AlexNet, the focus shifted towards Machine Learning to solve the ILSVRC and other Machine Vision tasks. The main idea was to make deeper and wider architectures, while trying to optimize the number of parameters. One clear example is *VGG-16*, which has more Convolutional layers than AlexNet, but only using kernels of size 3. Using exclusively small kernels increases the speed of computation [27].

However, stacking layers on top of layers resulted ultimatively in the degradation of accuracy. At some point the performance started to saturate er even decrease with the additional layers. The main cause was **vanishing gradient effect**. This effect occurs when the gradients are so small that during backpropagation the weights are not updated, meaning the networks stops training [28, 29].

In order to address this problem, He et al. [30] introduced the *identity mapping* or also known as *skip connections*. Figure 2.11 shows the *identity mapping*. Here a value is stored while other computations are performed on the same value, after the computation, the stored value is added to the output.

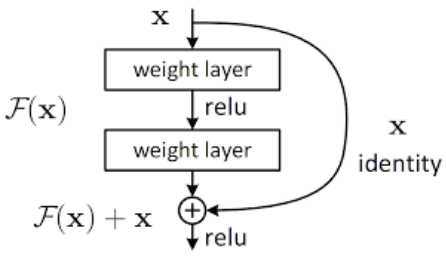


Figure 2.11: ResNet skip connection [30]

Figure 2.12 shows the *ResNet-34* architecture compared to the same architecture without the *identity mappings* and the *VGG-19* architecture.

This technique makes sure that deeper networks are able to keep learning without the *vanishing gradient effect* problem. He et al. [30] proposed several architectures of different depths with these properties and compared them to other SOTA (for that time) architectures, showing a significant improvement.

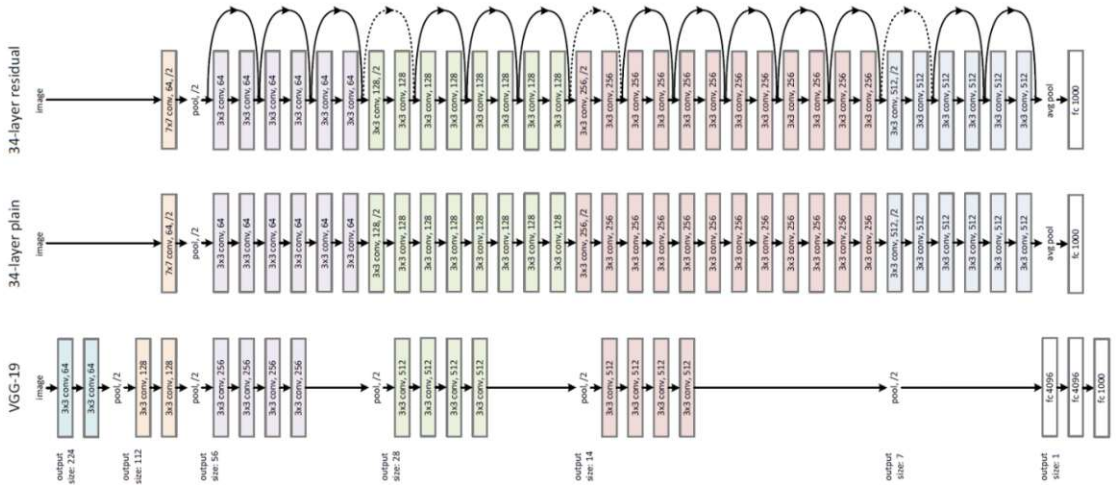


Figure 2.12: *ResNet* architecture compared to *plain architecture without identity mappings* and *VGG-19* [30]

2.2.4 EfficientNetV2

In 2019 Tan and Le [31] proposed a way to create models that can be easily scaled when resources are available. They did it in such a way that the depth, width and resolution of the network are carefully balanced when scaling them up. This was meticulously engineered to have a better performance and achieving state of the art results. Figure 2.13(e) shows their proposal to uniformly scale the three parameters mentioned using a fixed ratio.

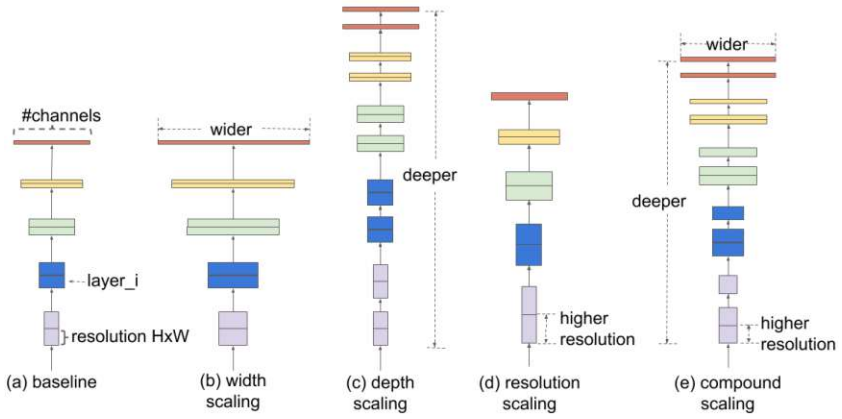


Figure 2.13: EfficientNetV1: Model Scaling [31]

In 2021, the same authors proposed an updated version of this architecture, calling it EfficientNetV2 [32]. With their new proposal they achieved faster training and used fewer parameters than other state of the art architectures with a drop in accuracy. To compensate for this drop in accuracy, they propose a method for progressive learning by adjusting the regularization along the image size.

Their study consisted of discovering the main bottlenecks in their previous architecture and addressing them. They found out that training with large image sizes slows down the process, depthwise convolutions are slow in early stages and that equally scaling up

every stage is sub-optimal. With the insight of which aspects to optimize, they designed a search space to apply a training-aware Neural Architecture Search (NAS)[9], thus obtaining a new architecture design.

Their proposed progressive learning method consists of training with scaled down versions of the images in early epochs and increasingly scaling them up. They also mention that the previous works that used progressive resizing kept a constant regularization, which led to a drop in accuracy. They argue that small image resolutions require weak regularization to not lose too much information, while larger resolutions require strong regularization to avoid overfitting. Therefore, they adapted the regularization to the resolution.

In the end they came up with a new family of scalable architectures which require less parameters and train faster than the state of the art. They also came up with a faster way of training that allows these architectures to achieve better performances on the mainstream datasets used, such as ImageNet and CIFAR.

2.2.5 ConvNeXt

With the increasing trend of research of Transformers[10] for the Machine Vision field, Liu et al. [33] combined gained knowledge from Convolutional Neural Networks and Transformer architectures to create their own architecture.

Figure 2.14 shows the steps taken to gradually improve their performance. It can be seen how they started with a base of ResNet50 (for the smallest version) and arrived to their proposed architecture, ConvNeXt-T (tiny) and compared this to Swin-T.

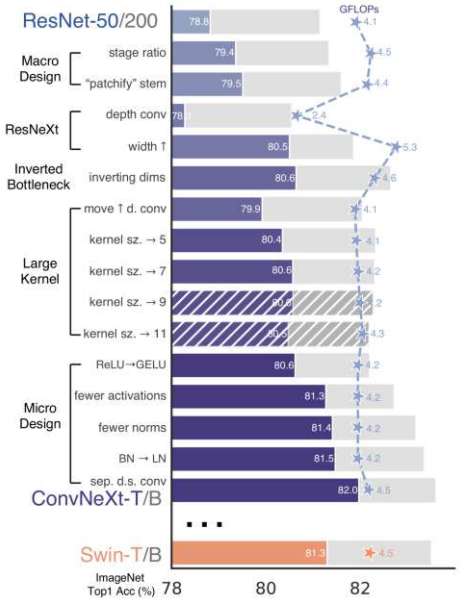


Figure 2.14: ConvNeXt: Approaches taken to create the architecture [33].

[9]NAS is a way to automatically design Artificial Neural Networks instead of designing them ad-hoc. The following link has more information about this topic: https://towardsdatascience.com/what-is-neural-architecture-search-and-why-should-you-care-1e22393de461

[10]The next section explains Transformers in detail.

2.3 Transformers

The attention mechanism was introduced by Bahdanau et al. [34] in 2015 as a way to improve the encoder-decoder based translation systems for Natural Language Processing (NLP). At this point, the state of the art for NLP relied on Recurrent Long Short-Term Memory (LSTM) and Neural Networks (RNN). However these approaches do not perform well when the size of the sentences increase. Thus, in 2017 a team at Google Brain took advantage of the attention mechanism to tackle translation tasks by getting rid of recurrence and convolutions and focusing entirely on attention [35]. The proposed architecture still consists of an Encoder and a Decoder, as did the RNN and LSTM ones. Figure 2.15 shows both encoder and decoder composed mainly of *Multi-Headed Self-Attention mechanisms*, *Skip*, *Add & Norm* and *Fully Connected Feed Forward* layers.

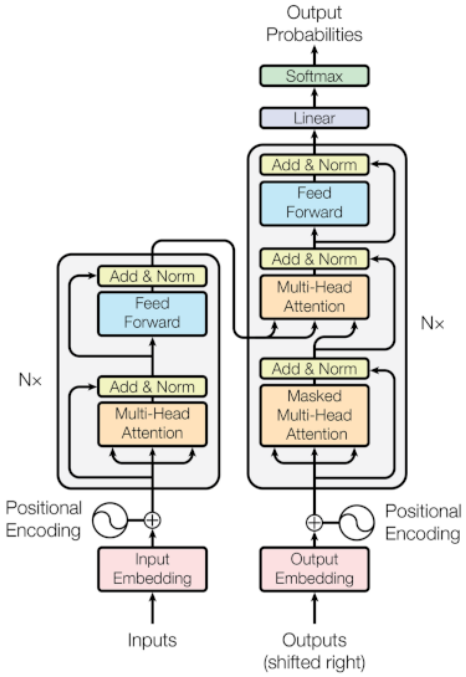


Figure 2.15: The Transformer - model architecture [35]

Figure 2.16 shows how the *Multi-Head Attention* block is composed by *h Scaled Dot-Product Attention* parallel operations, where *h* refers to the number of *heads*.

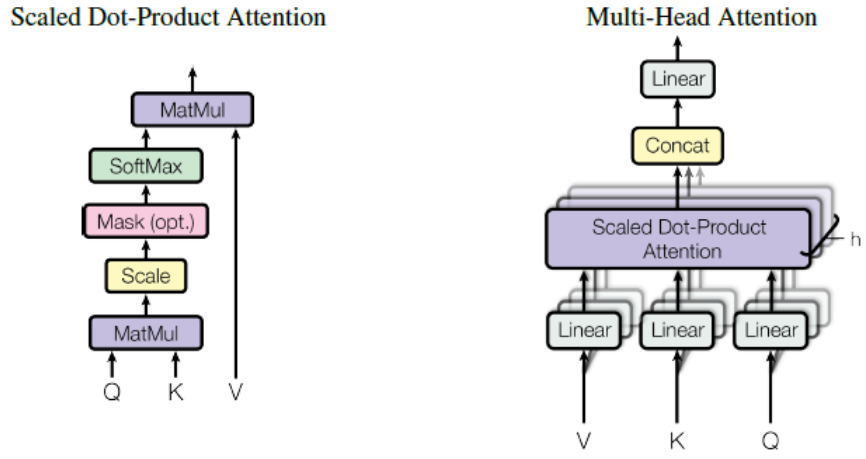


Figure 2.16: (left) Scaled Dot-Product Attention. (right) Multi-Head Attention [35]

Let's look at an example in order to understand how this *Scaled Dot-Product Attention* works and what it produces. For this purpose, let's consider the two following sentences:

Even though she did **not** win, she was happy.
Even though she did win, she was **not** happy.

Both of these sentences contain the same words, however the position of **not** is shifted changing the meaning of the sentence. In the first one, **not** refers to **win**, whereas in the second it makes reference to **happy**.

The first step is to tokenize the sentence. There are several ways of doing this (character, subword, word), for this example we will do it by words. Thus, each word corresponds to one token. Next, each token is vectorized[11], where each vector $\vec{v}_t$ has d dimensions and the values in each dimension carry semantic meaning. In this case the subscript t from $\vec{v}_t$ refers to each token/word in the sentence. When analyzing these vectors, it can be determined that words with similar meaning tend to cluster near each other and dissimilar ones are further apart[12].

It is also important to establish how the dot product of two vectors behaves for similar and dissimilar ones. Considering the simplified version of two dimensional vectors $\vec{p}$, $\vec{q}$ and $\vec{r}$ in Figure 2.17, we can realize the dot product between two close vectors, $\vec{p}$ and $\vec{q}$, and between two that are further apart, $\vec{p}$ and $\vec{r}$, to understand how the more complex multi-dimensional vectorized representation of words react to the dot product.

[11] This blog entry by J. Alammar explains in detail how vectorization can be done: `https://jalammar.github.io/illustrated-word2vec/`

[12] `http://projector.tensorflow.orghasagreatinteractiveviewertoanalyzedifferentwor`

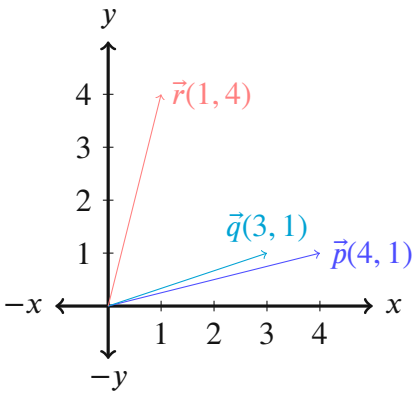


Figure 2.17: Example of two dimensional vectors

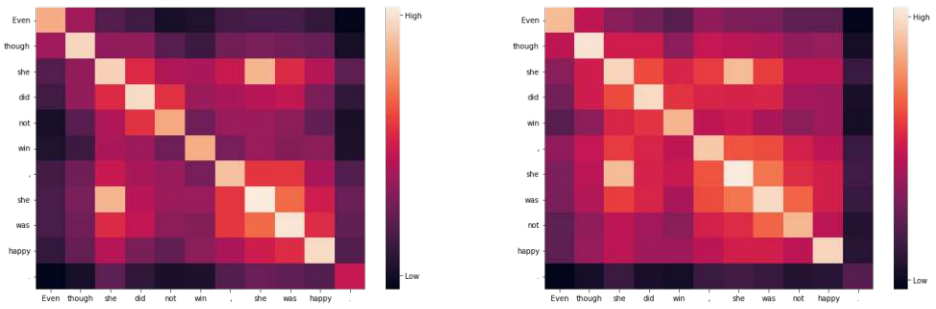
$$\vec{p} \cdot \vec{q} = \begin{pmatrix} x_1 & y_1 \end{pmatrix} \begin{pmatrix} x_2 \\ y_2 \end{pmatrix} = \begin{pmatrix} 4 & 1 \end{pmatrix} \begin{pmatrix} 3 \\ 1 \end{pmatrix} = 12 + 1 = 13$$

$$\vec{p} \cdot \vec{r} = \begin{pmatrix} x_1 & y_1 \end{pmatrix} \begin{pmatrix} x_2 \\ y_2 \end{pmatrix} = \begin{pmatrix} 4 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 4 \end{pmatrix} = 4 + 4 = 8$$

With this small example it can be seen that $\vec{q} \cdot \vec{q}$ produces a larger result than $\vec{q} \cdot \vec{r}$, even though $\vec{r}$ has a larger magnitude than $\vec{q}$. This effect relies solely on the fact that they are pointing in a similar direction. A pair of vectors pointing perpendicular to each other would result in 0 and angles greater than 90° result in negative numbers.

This concept can be scaled up to the $\mathscr{A}$ dimensions of the vectorized tokens $\vec{v}_t$ and get a confusion matrix when multiplying each word with each of the words in the sentence. Figures 2.18(a) and 2.18(b) show these confusion matrices for the examples with a representation of the resulting scalar. The way to get these scores (s) is by multiplying each row vector ($\vec{v}_m$) by the corresponding column vectors ($\vec{v}_n$) of each word. Here, the darker colors represent lower numbers and the lighter ones higher numbers, meaning the higher the value, the *closer* the words at that *row* x *column* are. As it was expected, the diagonal show high values, since the vectors are the same for *row* and *column*. The same can be observed for the word *she*, which appears twice in the sentences, thus resulting in higher values.

$$s_{mn} = \vec{v}_m \cdot \vec{v}_n^T \tag{2.6}$$



(a) Not win (b) Not happy

Figure 2.18: Confusion matrix for both sentences.

The *x axis* can be seen as the Keys (K) and the *y axis* as the Queries (Q), meaning the same confusion matrix S can be expressed by the following dot product:

$$S = QK^T \tag{2.7}$$

This value is then scaled by the square root of d_k, where d_k refers to the number of dimensions in the vectors in K. The results are then normalized by their probability distribution using the *softmax* function. We can call this W.

$$W(Q,K) = softmax\left(\frac{QK^T}{\sqrt{d_K}}\right) \tag{2.8}$$

It is important to mention that each of the weights W_{mn} has been influenced by its neighbours. These weights are then multiplied by the Values (V), which are each of the words in the given sentence, generating a contextualized representation of the given input vectors. This results in the Self-Attention via Scaled Dot-Product proposed by Bahdanau et al. [34] and shown on the left side of Figure 2.16

$$Attention(Q,K,V) = softmax\left(\frac{QK^T}{\sqrt{d_K}}\right)V \tag{2.9}$$

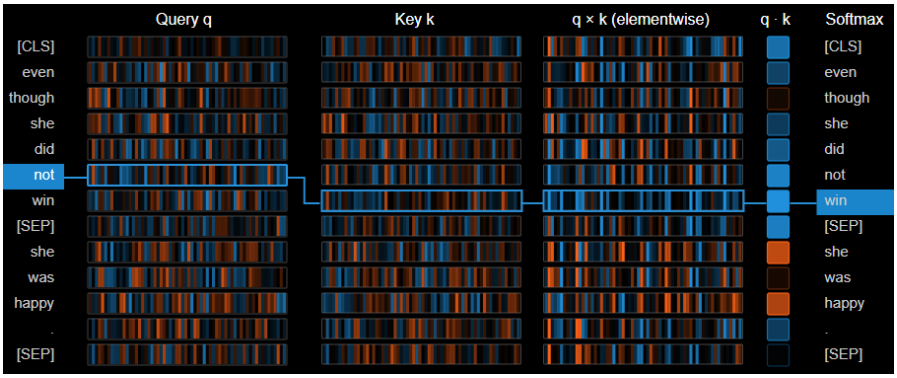
Now any chosen query Q_m can be selected in order to compute that specific contextualized self-attention. However, the values are fixed and no learnable parameters have been introduced. For this purpose new parameter matrices W_i^Q, W_i^K, W_i^V and W_i^O are introduced. Where $W_i^Q \in \mathbb{R}^{d_{model} \times d_k}$, $W_i^K \in \mathbb{R}^{d_{model} \times d_k}$, $W_i^V \in \mathbb{R}^{d_{model} \times d_v}$ and $W_i^O \in \mathbb{R}^{hd_v \times d_{model}}$. These matrices are implemented through Linear Layers. This is then used for each input for every head of the multi-head attention as well as after concatenating all of the results from each head as shown on the right side of Figure 2.16. The dot products (Equations:2.6, 2.7) are now matrix multiplications, but the result is the same.

$$MultiHead(Q,\,K,\,V) = Concat(head_1, ..., head_h)W^O, where \tag{2.10}$$

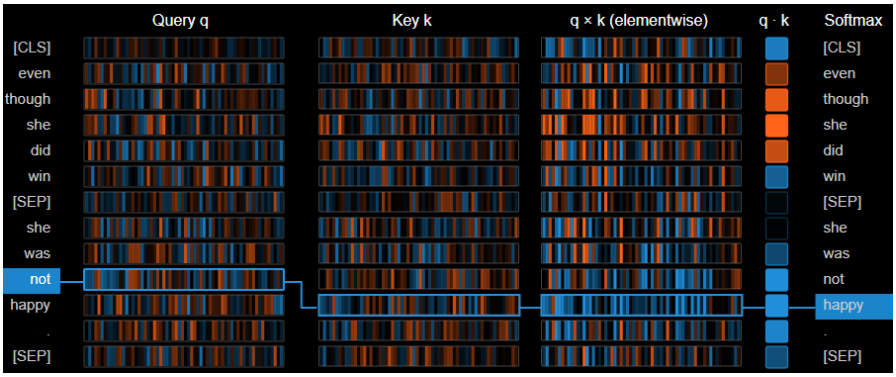
$$head_i = Attention(QW_i^Q, KW_i^K, VW_i^V)$$

Having multiple heads allows the architecture to pay attention to different parts of the sentence when training. Figures 2.19(a) and 2.19(b) were created using the pretrained BERT[13] [36] model from HuggingFace [37] and the BertViz [38] library. They show how one of the multiple heads behaves when selecting **not** as the query. It can be clearly seen how by introducing the (pretrained) weights, the focus of the sentence (for this head) is directed towards the word that the negation refers to.

[13] BERT (Bidirectional Encoder Representations from Transformers) was trained using the entire English Wikipedia.



(a) Not win



(b) Not happy

Figure 2.19: Head 2 of Multi-Head Attention: Attention matrix focused on the word **not** for both example sentences.

2.3.1 Vision Transformers (ViT)

Inspired by the great success of the Transformer in NLP, some researchers decided to experiment by incorporating them in the field of Machine Vision. Facebook AI[39] proposed an architecture called *DEtection TRansformer* (DETR), that consists of a convolutional backbone, feeding the output (positional encoding is also added) to a transformer encoder-decoder pair and finally a feed forward network. They trained this model with the COCO[14][40] dataset obtaining significant results.

Another team from Google Brain[41] introduced the Vision Transformer (ViT), which also relies mostly on the Attention Mechanism. Figure 2.20 shows the Vision Transformer architecture and Figure 2.26 shows a detailed view of the Transformer Encoder shown as a blackbox in the previous image.

[14]COCO(Common Objects in COntext): is a dataset containing >200K labeled images with object segmentation.

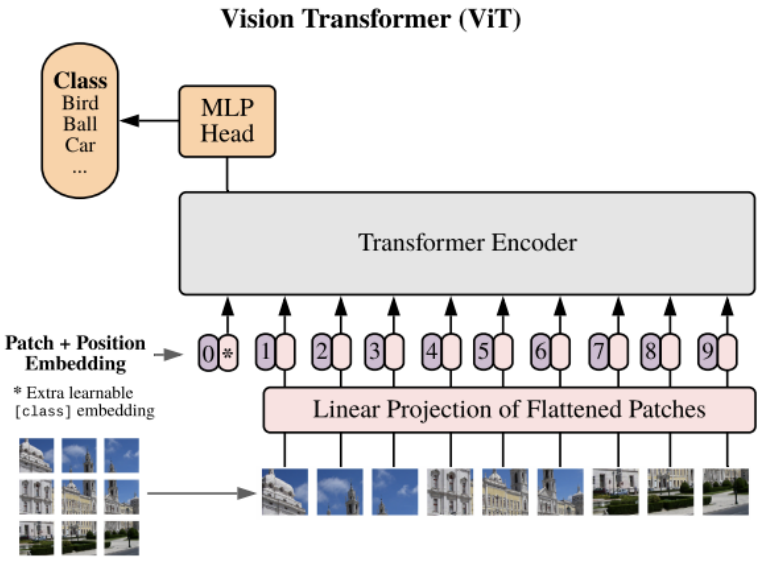


Figure 2.20: Vision Transformer - model architecture [41]

Before diving into the encoder, here is a quick overview of the steps taken to get a classification from the ViT as shown in Figure 2.20:

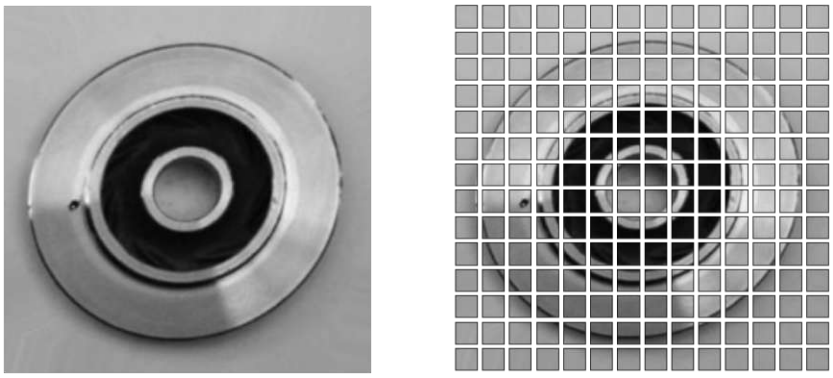
1. Split original image into patches
2. Flatten the patches
3. Add positional embeddings (prepended CLS (Class) Token included)
4. Feed everything to the encoder
5. The output for the CLS Token contains a distribution for the probabilities of each class

Split original image into patches

As Figure 2.21 shows, the original image[15] is split into equally sized patches (N[16]). Here the batch (B) and channels (C) are kept the same size, only the height (H) and width (W) are split to match the desired size of the patches. In this case, the patches are of size 16x16 pixels, thus resulting in 14x14 patches that do not overlap.

[15] The original paper[41] uses a resolution of 3x224x224 pixels. From this point forward, the images will be resized to match this, since it will be necessary to have one of the *default* resolutions to have access to available pretrained weights.

[16] N-number of patches



(a) Original image: 1x3x224x224 $BxCxH_{img}xW_{img}$

(b) 14x14 Patches, each Patch: 1x3x16x16 $BxCxH_{patch}xW_{patch}$

Figure 2.21: Dividing images into patches.

Figure 2.21 is just a representation of what the intent is. The actual way this is done is by the use of a Convolutional Layer with stride equal to the kernel size (to avoid overlapping) and enough output channels to fit an entire patch ($CH_{patch}W_{patch}$). The actual split image looks like Figure 2.22, since the Convolutional layer has two 16x16 kernels with randomly initialized values that have to be trained.

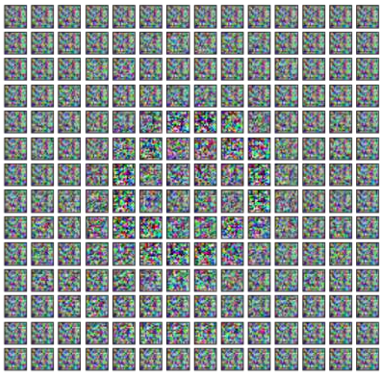


Figure 2.22: 14x14 Patches with random kernel

Flatten the patches

The objective of these steps is to obtain vector tokens that can serve as input to the encoder, similar to the word vectors in the previous NLP example. Equation 2.11 shows the expected size of each image x, where there are N vectors, each corresponding to each patch.

$$x \in \mathbb{R}^{CxH_{img}xW_{img}} \Rightarrow x \in \mathbb{R}^{NxCH_{patch}W_{patch}} \tag{2.11}$$

For the specific case where the images are of dimension 3x224x224 ($CxHxW$) and the patches are of size 16x16 the dimensions are as follows:

$$x \in \mathbb{R}^{3x224x224} \Rightarrow x \in \mathbb{R}^{14*14x3*16*16} = x \in \mathbb{R}^{196x768} \tag{2.12}$$

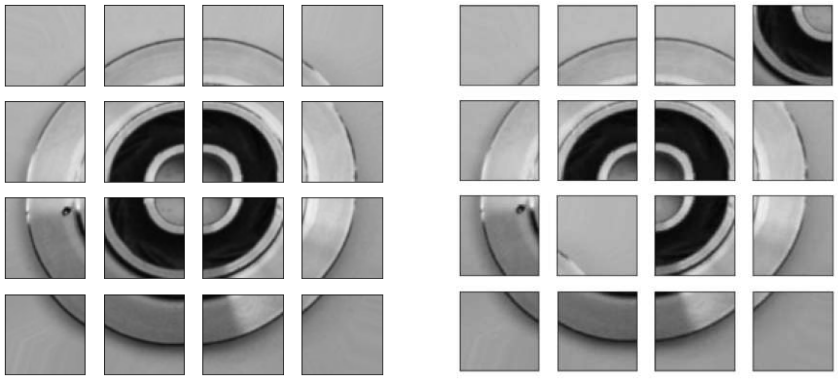
After splitting the image with the Convolutional Layer, the resulting shape of the image is $x \in \mathbb{R}^{768x14x14}$. This means each patch is already flat ($CH_{patchx}W_{patch} = 768$), however, the patches are still distributed into two dimensions ($14x14$). By flattening the tensor, $x \in \mathbb{R}^{768x196}$ is obtained. The last step is to transpose the tensor, resulting in the desired dimension depicted in Equation 2.12.

Add positional embeddings (prepended CLS (Class) Token included)

The Class Token (CLS) was introduced by Devlin et al. [36] when they proposed the BERT architecture for NLP. In the example for NLP shown in Figure 2.19, there are both [SEP] and [CLS] Tokens. They defined CLS as a special symbol added in front of every input sample, and SEP as a special separator token. For the Vision Transformer, the CLS token is used, appended at the beginning of each sample x. For this to be possible, the CLS token has to be of a compatible dimension, thus, $[CLS] \in \mathbb{R}^{1x768}$ in our case. When preppending this token to x we get:

$$x_{CLS} = \begin{pmatrix} [CLS] \\ x \end{pmatrix} \qquad \begin{aligned} x_{CLS} &\in \mathbb{R}^{197x768} \\ [CLS] &\in \mathbb{R}^{1x768} \\ x &\in \mathbb{R}^{196x768} \end{aligned} \tag{2.13}$$

After appending the [CLS] token at the beginning of the input image x, an additional positional embedding has to be added. Without the positional embedding, Figures 2.23(a) and 2.23(b) would represent the same input to the ViT, since the data is processed in parallel, rather than sequentially. There are many ways of doing positional encoding, absolute or relative, learned or fixed [42].



(a) No patches switched (b) Patches switched

Figure 2.23: Split image using bigger patches

The explanation given here of how Transformers for NLP work, focused primarily on the Attention Mechanism. However, Dosovitskiy et al. [41] used positional embedding for the Transformer, which included a way to establish the positional relation of the words into the model. The way this is done for Transformers in NLP is by the use of frequencies dependent on the position of the word and the dimension of the word vector, and independent on the meaning of the input word. The value is alternated between

sin and *cos* for even and uneven position respectively. This result is then added to the *word vector*, thus shifting its semantic meaning by a small value (-1,1) in the various dimensions of the vector. The Transformer then learns that these *constants* that shift every word, independent of the actual meaning, correspond to the position of the word in the sentence.

$$PE_{(pos,2i)} = sin\left(pos/10000^{2i/d_{model}}\right) \tag{2.14}$$

$$PE_{(pos,2i+1)} = cos\left(pos/10000^{2i/d_{model}}\right) \tag{2.15}$$

Figure 2.24 shows the values to be added to the first 5 dimensions of a 10 word sentence. During training the model learns that these values are added to these positions, and can then identify which words are closer or further to each other, or are even set apart by a [SEP] token.

Figure 2.24: First 5 dimensions for 10 words, where $d_{model} = 512$

For the Vision Transformer, there were different approaches considered[17] and it was established that adding a positional embedding improved the performance, however, there was no significant difference between the different methods evaluated. The actual implementation used is by defining a layer with trainable parameters that will act as the positional embedding, this layer is then added to the input image x_{CLS}. This positional embedding layer (PE) has the same dimensions as x_{CLS}.

$$x_{pos\_emb} = x_{CLS} + PE \qquad \begin{aligned} x_{pos\_emb} &\in \mathbb{R}^{197x768} \\ x_{CLS} &\in \mathbb{R}^{197x768} \\ PE &\in \mathbb{R}^{197x768} \end{aligned} \tag{2.16}$$

17

- No positional embedding
- 1-dimensional positional embedding (sequence of patches, e.g. 1-N)
- 2-dimensional positional embedding (grid of patches, e.g. [1-N_x, 1-N_y])
- Relative positional embedding

Figure 2.25 shows three different positional embeddings, trained with different hyperparameters. These only represent one of the three RGB channels for each of the different training methods. Here we can see how the learnt parameters range from −1 to 1, similarly to the NLP solution.

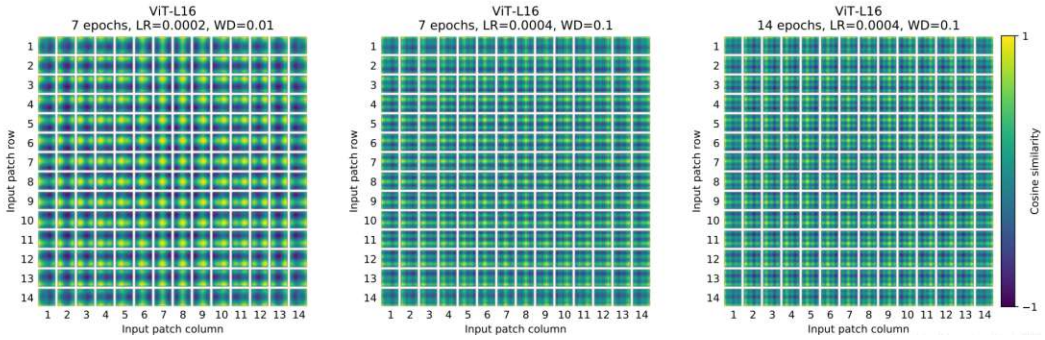


Figure 2.25: Positional embeddings of models trained with different hyperparameters [41]

Feed everything to the encoder

At this point, the image to feed the encoder has been split into patches, flattened, appended a class token and added a positional embedding. Now, as Figure 2.20 shows, this result will be passed through the encoder. The original paper described Figure 2.26 to be the Transformer Encoder, but there is a little bit more to understand. Let's call this structure "block". On the upper left corner is an "L x", signifying there are various of these blocks.

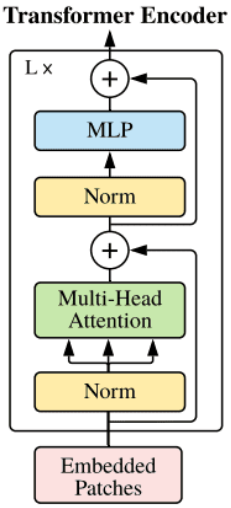


Figure 2.26: Vision Transformer Encoder (Block) [41]

Let's look at one block. The first step of the block is to normalize the input. This input (Embedded Patches) has all of the patches and the class token, meaning a single block receives all the data. Then the normalized data is fed to a Multi-Head Self-Attention Mechanism (MSA). This result is added to the original input through a Skip-Connection, then it is normalized again before feeding it to a Multi-Layer Perceptron (MLP) and

added again through another Skip-Connection[18]. Before going further to what happens to the output of this single block, let's look at the more relevant sections of this block, namely the MSA and MLP layers.

Multi-Head Self-Attention Mechanism

Returning to Equation 2.16, we can redefine x_{pos_emb} as z, which will be the input for this stage, after normalizing it. The Multi-Head Self-Attention Mechanism is the same as the one described for Transformers in NLP. Here, each patch token (CLS token included) will be analogous to the vector word in NLP. These are then represented as queries (q), keys (k) and values (v). We now know that these values are multiplied by a matrix (Linear Layer - U_{qkv}) before performing the self-attention. Then q and k matrices are multiplied, scaled by a factor ($\sqrt{D_h}$) and the softmax is computed. This result is then multiplied by v, constituting the self-attention for a single head.

$$[q, k, v] = zU_{qkv} \qquad z \in \mathbb{R}^{NxD}, \tag{2.17}$$

$$A = softmax\left(qk^T/\sqrt{D_h}\right) \qquad U_{qkv} \in \mathbb{R}^{Dx3D_h}, \tag{2.18}$$

$$SA(z) = Av \qquad A \in \mathbb{R}^{NxN} \tag{2.19}$$

In the same way as for NLP, there are multiple heads, each receiving the same input and doing the same calculations, where each of them has a matrix U_{qkv} with learnable parameters that distinguishes them from each other. The resulting matrices from each head are then stacked together and multiplied by another matrix with learnable parameters (U_{msa}) to return to the original dimensions.

$$MSA(z) = [SA_1(z); SA_2(z); ...; SA_k(z)]U_{msa} \qquad U_{msa} \in \mathbb{R}^{k*D_hxD}, \tag{2.20}$$

where k represents the number of heads. This process is done in parallel for each head.

Multi-Layer Perceptron

The Multi-Layer Perceptron (MLP) used consists of only two linear layers and one GeLU activation function. The shape of the output remains unchanged in respect to the input given to the MLP.

After passing the embedded patches through a single block, the result is then used for the next block. This means that this step is done sequentially for the L number of blocks, rather than parallel. Figure 2.27 shows a representation of how this should work.

[18]There are usually Dropout Layers scattered all over the Block to help generalize during training if specified.

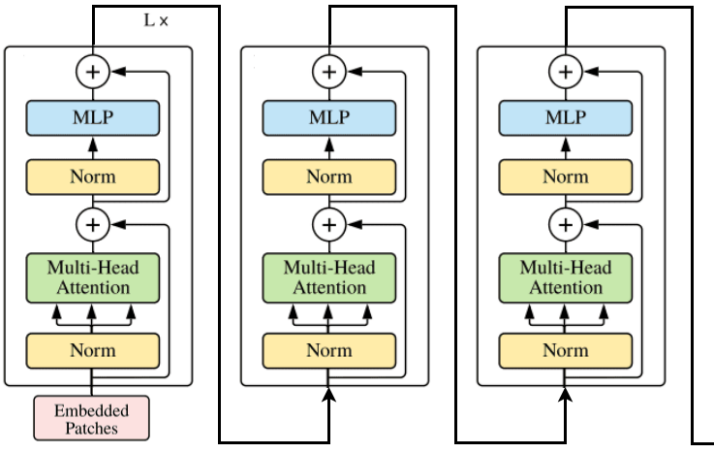


Figure 2.27: Vision Transformer Encoder (Block) [41]

The output for the CLS Token contains a distribution for the probabilities of each class

As explained in the previous section, the output of the encoder has the same shape as the input $\mathbb{R}^{NxD}$, where N is the number of patches plus the class token in the first position and D is the size of these flat patches containing 16 by 16 pixels, all with RGB channels. From this output, the first dimension corresponding to the class token is taken and passed through another MLP with the number of output features equal to the number of classes to classify the image.

Breaking the whole process down, it looks like this.

$$z_0 = \left[x_{class}E; x_p^1E; x_p^2E; ...; x_p^NE\right] + E_{pos}, \quad E \in \mathbb{R}^{(P^2 \cdot C)xD}, E_{pos} \in \mathbb{R}^{(N+1)xD} \tag{2.21}$$

The image is split into N patches $\left(x_p^1 - x_p^N\right)$, each patch is flattened and multiplied by a matrix with learnable parameters (E). These are then concatenated along with a class token (x_{class}) in the first position. Finally, the positional embedding (E_{pos}) is added, resulting in the input for the encoder (z_0).

$$z'_\ell = MSA(LN(z_{\ell-1})) + z_{\ell-1}, \qquad \ell = 1...L \tag{2.22}$$

Now in order to obtain the intermediate value (z'_ℓ), the input value $(z_{\ell-1})$, which for the first block corresponds to z_0, is normalized (LN), then put trough a multi-head self-attention mechanism (MSA) and added with the original input $(z_{\ell-1})$ through a skip layer.

$$z_\ell = MLP(LN(z'_\ell)) + z'_\ell, \qquad \ell = 1...L \tag{2.23}$$

The second half of the block consists of taking this intermediate value (z'_ℓ), normalizing it (LN), performing a feed forward through a multi layer perceptron (MLP) and

adding it to the intermediate value (z'_ℓ) through a skip layer. This output is then fed to the next block performing Equations 2.22 and 2.23 back and forth for L iterations.

$$y = MLP(LN(z_L^0)) \tag{2.24}$$

The output classification is then done by taking the output of the last block of the encoder (z_L) and selecting only the first row, corresponding to the class token (z_L^0). This is then normalized and passed through a last multi-layer perceptron (MLP) that outputs the probabilities for each class.

2.3.2 Swin Transformer

We have now seen how the original Transformer and the Vision Transformer work. However, the Vision Transformer scales poorly for larger images, because every pixel "attends" to every other pixel. This means that every time the self-attention is computed, every patch is multiplied by every other patch, even though they might not be close.

Liu et al. [43] propose a way to start these multiplications locally, and then scale up to a more global view of the image. This pyramid form of doing the multiplications speeds up the process and allows for a finer view of the details before scaling up to a broader overview.

The first concept to understand how the Swin Transformer differs from the original transformer is that there are local windows as shown in Figure 2.30. The right side shows the original ViT transformer, where each patch of the image is multiplied by every other patch in order to get the Self-Attention. This is marked with a red border along the entire image, this can be seen as the window where the operations are done. The flow is from bottom to top, where each window remains the same size.

On the left side is the Swin Transformer, there are several sizes of these windows. The top representation of how the Self-Attention is done is the same as in ViT. Stepping backwards into the flow, the windows are smaller and patches are also smaller. This means that the Self-Attention operations are done only within the compounds of the red windows, reducing the complexity of computations needed.

As the arrow shows, the flow goes from smaller windows to bigger ones. This allows for smaller patches with higher resolution to interact with each other, gaining high resolution information of patches attending to near neighbours. The windows are then scaled up in next iterations to allow for a more general view, letting bigger patches (which already contain information of the smaller patches interacting with each other) interact.

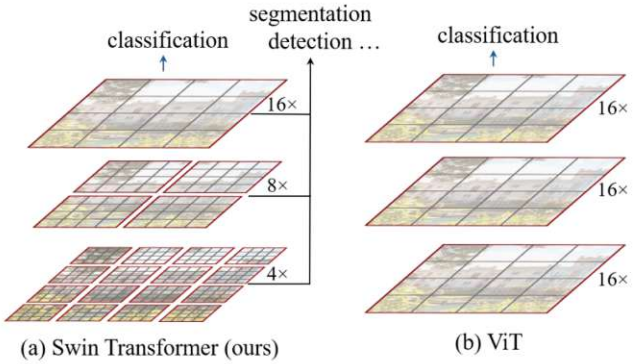


Figure 2.28: Swin Transformer [43]

However, if this process is done like as shown above, the information exchange is done only locally, relying on the last blocks to exchange information from distant patches. In order to avoid this, shifted windows are introduced. Figure 2.29 shows an example for the case where there are four windows dividing the original image. These windows are then shifted in the x and y directions by half of the window's size. This configuration allows the shifted blocks to exchange new information from neighbouring windows from the previous configuration.

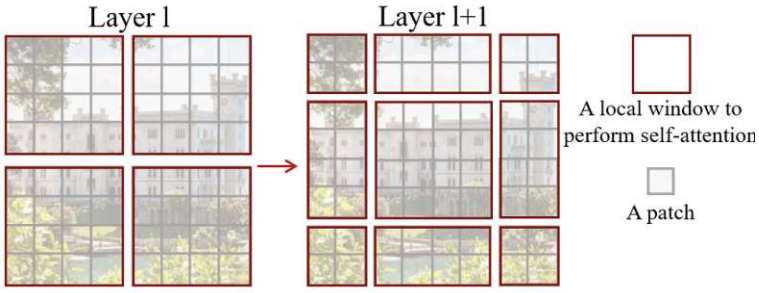


Figure 2.29: Swin Transformer [43]

The shifting is done always in pairs, meaning that the first block will have a *normal* configuration and the next one will have a *shifted* configuration. Figure 2.30 shows two successive Swin Transformer blocks with their Window-Multi-Head Self-Attention (W-MSA) and the Shifted Window-Multi-Head Self-Attention (SW-MSA) respectively.

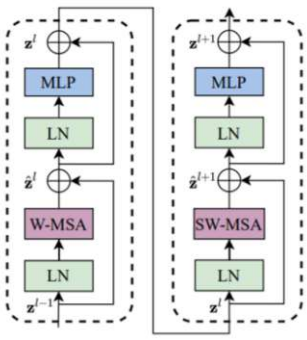


Figure 2.30: Swin Transformer: Two Successive Swin Transformer Blocks[43]

The Swin Architecture is then composed by these concepts. As Figure 2.31 shows, it consists of 4 stages. As with the ViT, the input image is split into patches and flattened before feeding it into the block. At the first stage, the image is divided into patches of 4 pixels each, and after doing the linear embedding, it forwards it through two consecutive Swin Transformer Blocks. As explained earlier, these blocks come in pairs, shifting the windows back and forth as shown in Figures 2.29 and 2.30. The amount of successive transformer blocks depends on the stage and the size of the architecture as shown in Table 2.1. This process is then repeated, scaling up the size of the patches (including more pixels per patch) and also scaling the window accordingly to fit the new patches.

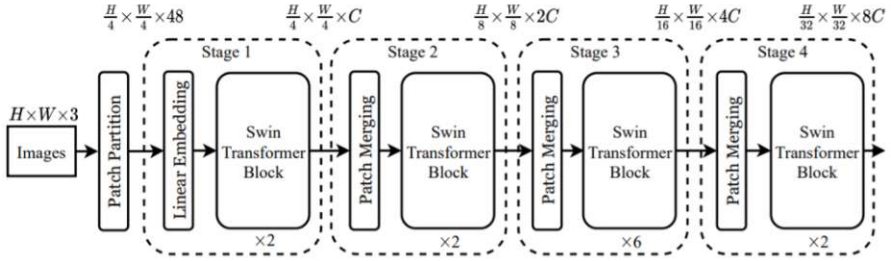


Figure 2.31: Swin Transformer Architecture [43]

Model	C	Successive Transformer Blocks			
		Stage 1	Stage 2	Stage 3	Stage 4
Swin-T	96	2	2	6	2
Swin-S	96	2	2	18	2
Swin-B	128	2	2	18	2
Swin-L	192	2	2	18	2

Table 2.1: Architecture parameters for the different Swin Transformer sizes [43].

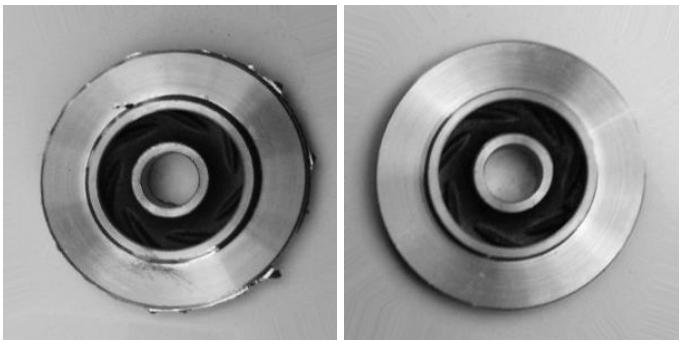
After this section, the reader should now have a basic understanding of the basic building blocks and how they are connected to form complex architectures. Different architectures are introduced and some were even explained in detail how they work. However, there are still some steps until these architectures can classify the desired dataset correctly.

Next we need to specify the dataset that is going to be used, then we can move on to training these models.

3 Dataset: Images of Pump Impellers

This project is intended to be reproduced in the manufacturing field with images that might present visible defects. The goal is to train a model to correctly classify images with a small dataset, such that the decisions made by the model can be justified via visual inspection of the regions of interest selected by the model. For this purpose a publicly available dataset was selected to train eight models and verify their performances. The dataset used in this project was uploaded to Kaggle by R. Dabhi. It contains images provided by PILOT TECHNOCAST, a manufacturing company from India. The images show the top view of stainless steel casted submersible pump impellers as depicted in Figure 3.1. Pump impellers accelerate the pumped fluid radially outwards, transforming energy from the motor into pressure.

Casting refers to the manufacturing process, where a liquid is poured into a mould containing the negative impression of the shape desired and later allowed to solidify. This process allows the manufacture of complex shapes in an inexpensive way. According to Rundman [44], there are several sources of imperfections when performing this procedure with metals.



(a) Defect (b) Ok

Figure 3.1: Top view of Pump Impellers

The first source mentioned are gases in the materials. If gas is present in the liquid, the gas will be trapped resulting in porosity. The next source mentioned are hot tears, resulting from residual stresses due to variations in the cooling rate. Shrinkage is also to be considered during the process, which leads to the use of risers. The riser is an excess of material which provides the actual piece a source to draw material from when cooling, since the density is not the same in the liquid and solid form, thus shrinking up to 5-7%.

All of this leads to various sources of shortcomings in the manufacturing of these pieces. The result of such errors can be seen in Figure 3.1(a). Hence, this dataset was gathered to identify through Computer Vision algorithms if the images contain a visible defect and classify the object as defective or not.

The dataset consists of one folder containing the original images with 512px by 512px of resolution and another folder with augmented data as shown in the tree in Figure 3.2. The first folder (casting_512x512) contains two sub-folders, which divide the data into defective and ok. In these folders are 718 images for the defective class and 519 for the correct one.

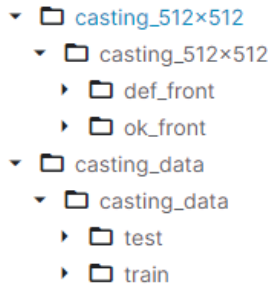


Figure 3.2: Dataset Folder Structure

The other folder containing augmented data has two sub-folders dividing the data into training and testing data. For the training data, there are 3758 defective images and 2875 ok ones. For testing, there are 453 defective images and 262 ok ones.

	Def	Ok	Total	Def (%)	Ok (%)
Original (512x512px)	718	519	1237	58.04	41.96
Train ($\mathcal{T}'$)	3758	2875	6633	56.66	43.34
Test ($\mathcal{R}$)	453	262	715	63.36	36.64
Total ($\mathcal{T}' \cup \mathcal{R}$)	4211	3137	7348	57.31	42.69

Table 3.1: Images provided in the dataset

Table 3.1 shows that the data is fairly well balanced and does not contain a class that heavily outnumbers the other one. Let us denote the dataset as $\mathcal{D}$ and its instances as $\{(x_i, y_i)\}_{i=1}^{N}$ with N as the size of the dataset, $\{x_i\}_{i=1}^{N}$ the images and $\{y_i\}_{i=1}^{N}$ the class they belong to, also referred to as labels. In this case the dataset would be the union of the subsets Train ($\mathcal{T}'$) and Test ($\mathcal{R}$). Where each subset has unique instances.

$$\mathcal{D} = \mathcal{T}' \cup \mathcal{R} \tag{3.1}$$

$$\mathcal{T}' \cap \mathcal{R} = \emptyset \tag{3.2}$$

As the names suggest, $\mathcal{T}'$ will serve to train the different models and $\mathcal{R}$ will serve to analyze the performances after the training. However, in order to validate the model during training, there has to be another subset in order to validate the performance of the training. The Validation subset ($\mathcal{V}$) is taken from $\mathcal{T}'$, thus creating two new subsets, one for the actual training $\mathcal{T}$ and one for validating $\mathcal{V}$.

$$\mathcal{T}' = \mathcal{T} \cup \mathcal{V} \tag{3.3}$$

This means that:

$$\mathcal{D} = \mathcal{T}' \cup \mathcal{R} = \mathcal{T} \cup \mathcal{V} \cup \mathcal{R} \tag{3.4}$$

$$\mathcal{T} \cap \mathcal{R} = \emptyset, \mathcal{V} \cap \mathcal{R} = \emptyset, \mathcal{T} \cap \mathcal{V} = \emptyset \tag{3.5}$$

Usually $\mathcal{D}$ is divided in a way that $\mathcal{T}$ comprises 80%, $\mathcal{V}$ 10% and $\mathcal{R}$ the remaining 10%. This dataset is already divided into $\mathcal{T}'$ and $\mathcal{R}$, with $\mathcal{R}$ being around 9.7% of $\mathcal{D}$ (715 images out of 7348). By splitting $\mathcal{T}'$ by 90% for $\mathcal{T}$ (5970 images) and 10% for $\mathcal{V}$ (663 images), $\mathcal{V}$ results in 9% of $\mathcal{D}$.

3.1 Preprocessing

The state of the art libraries for machine learning development (such as Pytorch and TensorFlow) count with built-in techniques to prepare datasets in order to feed them to the models. Many of these datasets are readily available online to download and the libraries provide easy ways to access them. In the case of image processing, the vision package of pytorch, `torchvision`, provides typically used datasets to download. The example code in Listing 1 shows how with three lines of code the MNIST[45][1] dataset is downloaded to the specified directories and split into train and validation.

Listing 1 Example on how to download the MNIST dataset

```
import torchvision

ds_train = torchvision.datasets.MNIST(
            root='datasets/MNIST/train/',
            train=True,
            download=True
        )
ds_val = torchvision.datasets.MNIST(
            root='datasets/MNIST/val/',
            train=False,
            download=True
        )
```

With these lines of code the images are available for use in the form of a subclass of `torch.utils.data.Dataset`. However, the use of custom datasets is crucial for Machine Learning, thus pytorch offers a simple way to define these datasets. In order to do so, it is necessary to create a class that inherits from the abstract class `torch.utils.data.Dataset` and implements the methods `__len__` and `__getitem__`. As the name of the methods suggest, the first one should return the number of items in the dataset and the second should return an item corresponding to the index passed to it. For this method, the loading and handling of the data must be done by the user.

[1] MNIST(Modified National Institute of Standards and Technology) contains grayscale images of handwritten numbers.

As with the ready to use datasets, the `torchvision` package provides a solution to handle the data and create a custom dataset automatically either for generic files with the class `DatasetFolder` or explicitly for images with `ImageFolder` which inherits from `DatasetFolder`.

When creating an object of the `DatasetFolder` class, the following parameters have to be provided:

- **root***(string)* - Root directory path.
- **loader***(callable)* - A function to load a sample given its path.
- **extensions***(tuple[string])* - A list of allowed extensions. Both extensions and is_valid_file should not be passed.
- **transform***(callable, optional)* - A function/transform that takes in a sample and returns a transformed version. E.g, `transforms.RandomCrop` for images.
- **target_transform***(callable, optional)* - A function/transform that takes in the target and transforms it.
- **is_valid_file** - A function that takes path of a file and checks if the file is a valid file (used to check of corrupt files) both extensions and is_valid_file should not be passed.

The parameter `root` should specify the path to a directory like the one shown in Figure 3.2, which already divides the different classes into sub-folders. It must also be taken into consideration that this is a generic solution for every type of file defined in either the `extensions` or the `is_valid_file` parameters, thus a way to read the files from the directories must be specified within the `loader` parameter. Since this project deals only with images, the subclass `ImageFolder` handles the loading of the images and has a predefined list of allowed extensions. These are the parameters for `ImageFolder`[2]:

- **root** *(string)* – Root directory path.
- **transform** *(callable, optional)* – A function/transform that takes in an PIL image and returns a transformed version. E.g, `transforms.RandomCrop`
- **target_transform** *(callable, optional)* – A function/transform that takes in the target and transforms it.
- **loader** *(callable, optional)* – A function to load an image given its path.
- **is_valid_file** – A function that takes path of an image file and checks if the file is a valid file (used to check of corrupt files)

[2]The scope of this project is for it to be reproducible with other objects within the ZDMP, hence the importance to explain how to create the corresponding dataset objects.

As the description of the parameters show, only the path is needed to create a valid dataset. The optional parameters allow for a customized preprocessing of the dataset when loading it, either by the way the data is loaded, the filtering of undesired data or by applying transformations to the images.

`Transform` refer to a set of common image transformations that either prepare the data to fit the needs of the models or distort the data, thus creating new versions of the original data. This technique is further discussed in Data Augmentation.

Neural Networks work usually with floating-point data as input, showing best training performance when ranging from 0 to 1 or -1 to 1. Typically the original pixel values range from 0 to 255, so they have to be adjusted to either range mentioned. Ioffe and Szegedy [19] proposed a way to reduce the *internal covariance shift* by normalizing each training mini-batch inside the model architecture as Algorithm 2 shows.

Algorithm 2 Batch Normalization [19]

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i$$

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

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

$$y_i \leftarrow \gamma \hat{x}_i + \beta$$

The first steps are to get the mean (μ) and variance (σ^2) for the input batch $\mathcal{B}$. Then each datapoint x_i is normalized by shifting it by μ towards zero mean and dividing it by the sqare root of σ plus a constant ϵ resulting in new normalized datapoints $\hat{x}_i$. These new $\hat{x}_i$ are then used to train the parameters γ and β. Usually the constant ϵ is a small number for numerical stability that can be neglected, approximating the denominator to the standard deviation σ.

If the full training subset $\mathcal{T}$ is available, $\mu_{\mathcal{T}}$ and $\sigma_{\mathcal{T}}$ can be determined and used to normalize the full dataset $\mathcal{D}$. These normalizing values can be used not only for training, but also during validating, testing and in production. It is important to mention that only the training data $\mathcal{T}$ and not the validation $\mathcal{V}$ or testing data $\mathcal{R}$ are to be used in order to avoid data leakage [46].

Another option is to used readily available μ and σ values from large datasets, provided they belong to a pertinent dataset. ImageNet [47], for example is composed of millions of images of *natural scenes*, such as animals, household items and food. The normalizing values are as follow:

- $\mu_{ImageNet}$: 0.485, 0.456, 0.406

- $\sigma_{ImageNet}$: 0.229, 0.224, 0.225

For each channel RGB respectively. For the case of the MNIST dataset, which has grayscale images of handwritten numbers, the values are:

- μ_{MNIST}: 0.1307
- σ_{MNIST}: 0.3081

Since the casting dataset in question is very specific and $\mathcal{T}$ is available, it makes sense to work out the corresponding values. For $\mathcal{T}$ the values are:

- $\mu_{\mathcal{T}}$: 0.5643
- $\sigma_{\mathcal{T}}$: 0.2386

Figure 3.3 shows the pixel distribution along the different grayscale values before normalizing the data for the subset $\mathcal{T}$. It also shows the original $\mu_{\mathcal{T}}$ and $\sigma_{\mathcal{T}}$ before dividing by 255. If the same diagram was plotted for the MNIST dataset, the expected result would be two big bars, one at position 0 and one at 255, since the dataset consists of a dark background and a white number, and then some orders of magnitude lower the rest of values. Whereas here we clearly see a dark concentration of pixels for the center of the pump impellers and then a larger consolidation at higher values for the different tones that make up the rest of the objects as well as the background.

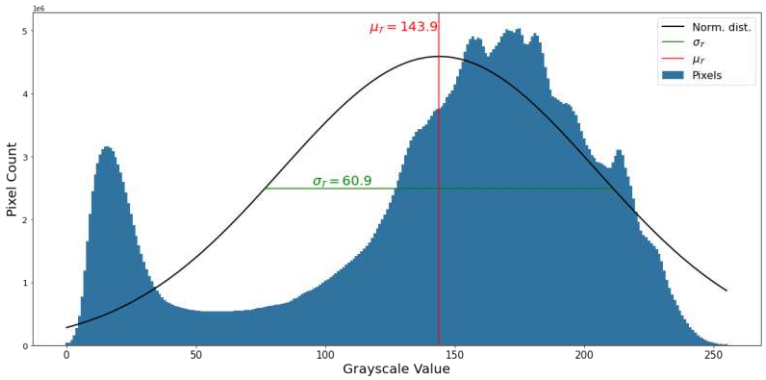


Figure 3.3: Grayscale Histogram for training subset $\mathcal{T}$ of the Casting Dataset

3.2 Data Augmentation

An under-trained model tends to over-generalize (underfitting), whereas a model extensively trained to perform one task is prone to under-generalize (overfitting) [48]. Thus, overfitted models tend to perform well on the training set and very poorly on unseen data [49].

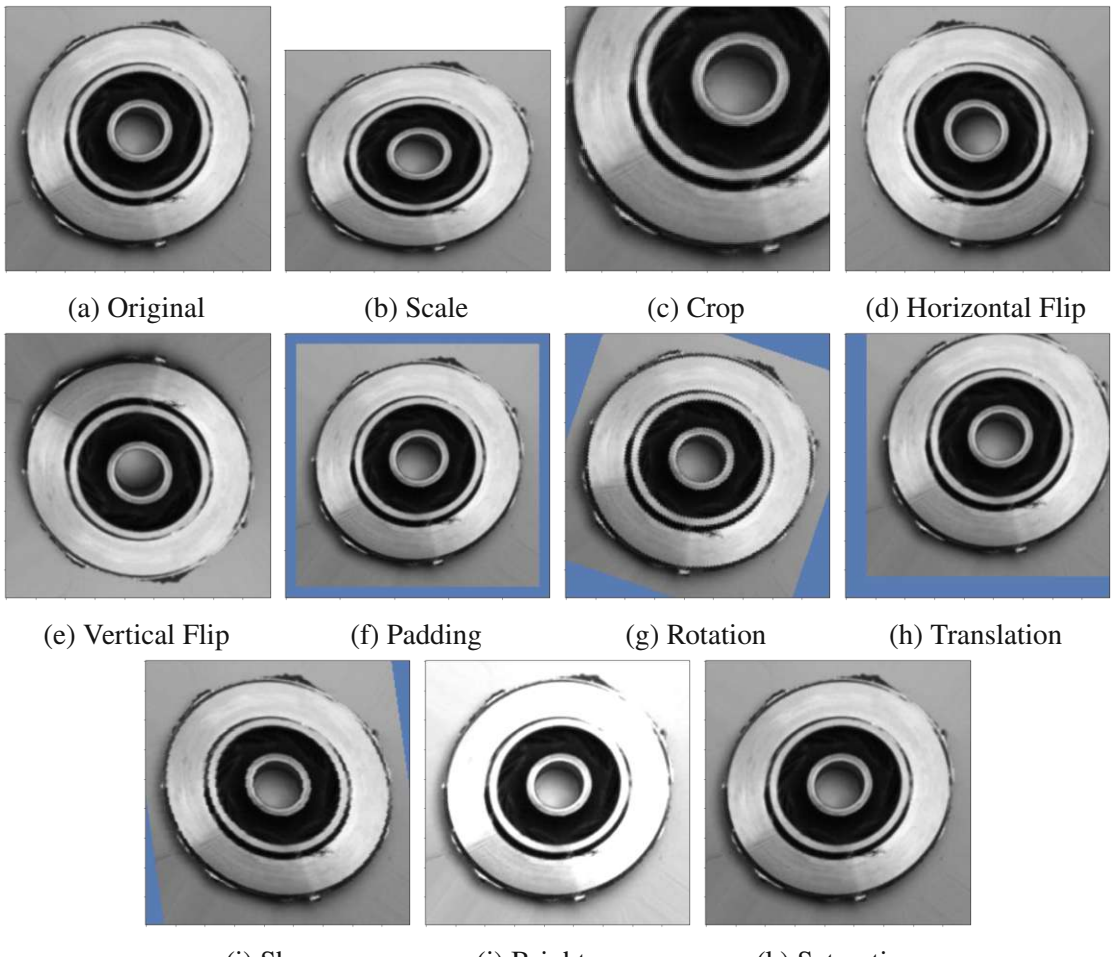
One of the main reasons that overfitting occurs is due to noise learning [50]. An example of this would be an architecture trained to predict the ages of different images of faces. A good fit would try to look at signifiers like wrinkles, gray hair, hairstyle, clothing, etc. An overfitted model would remember each person and their age [24]. Machine

Learning architectures have grown larger in size, which results millions of parameters to be trained, thus making it easier to remember each input and which output it should generate.

There are many approaches to avoid overfitting, such as early stopping, noise reduction, regularization and expansion of the dataset. The expansion of the dataset might be done either through the acquisition of more datapoints or the creation of new ones artificially based on the existing ones. This last element is also called data augmentation [50]. The main goal of data augmentation is to synthetically increase the training set size beyond what the model is capable of memorizing, thus generalizing [24].

For Machine Vision there are many standard techniques used to augment the data with common image transformations such as position augmentation and color augmentation.

Figures 3.4(b) to 3.4(i) show examples of position augmentation and Figures 3.4(j) to 3.4(k) show examples of color augmentation on the casting dataset. Some of these examples have been exaggerated to emphasize how they affect the original image.



(a) Original (b) Scale (c) Crop (d) Horizontal Flip

(e) Vertical Flip (f) Padding (g) Rotation (h) Translation

(i) Shear (j) Brightness (k) Saturation

Figure 3.4: Typical positional transformations

4 Training and Fine Tuning the Models for Image Classification

Up until now, the dataset and also the architectures used in this project have been introduced . This section will go step by step explaining how the architectures are trained to classify the images.

First, this section shows how the models are loaded for the training. Then, the loss is introduced. This will be the criteria to minimize in order to determine how well the model is performing. Then the optimizer is explained to know how the model is guided into a better performance (thus, minimizing the loss) and finally the actual steps taken during training are described. Here, the resulting values of training, validation and testing are also presented.

4.1 Creating/Loading the models

The Luna model comes from the book *Deep Learning with PyTorch* by Stevens et al. [24]. This model was originally designed to detect *nodules* in three-dimensional data of CT scans of lungs. Thus, by modifying the architecture to analyse two-dimensional data, there are no pretrained weights available that can be used to initialize the model and then fine tune for the specific task.

This model, its dependencies and its constructor have been defined inside the repository for this project. In order to create a Luna model, either this library has to be imported, or the architecture must be copied. Since this is the only model trained from scratch, it was trained for *50 epochs*.

The rest of the models have been part of publications which have trained their architecture on the *ImageNet* dataset. For many of these architectures, there are several sources to get pretrained weights from (e.g. torchvision, timm, huggingface, kaggle). The main intent was to use *torchvision* as the only source of pretrained weights, however, not all of them are available on *torchvision*.

The model that is **not** available on *torchvision* is **DINO**. This model has to be loaded from the main source (*facebookresearch*). Even other libraries such as *huggingface* load the pretrained weights from this github repository. The workflow to get this model with its weights does not deviate from the other pretrained models.

Listing 2 shows an example to load a pretrained model from *torchvision*. This process is similar with the other libraries, where with a single line of code, the model with

the pretrained weights is loaded. For the model to work with this binary classification problem, the last layer has to be adjusted to converge to only two outputs. How this last layer is accessed will depend on the architecture.

Listing 2 Example on how to load a pretrained model from *torchvision:v0.13.0*

```
model = torchvision.models.efficientnet_v2_s(
        weights="EfficientNet_V2_S_Weights.IMAGENET1K_V1"
    )
model.classifier = torch.nn.Linear(1280, 2)
```

Number of parameters per model

Each architecture is different, thus having a different number of parameters to be tuned. Table 4.1 shows the amount of parameters per model after adjusting them to the right amount of classes in the output layer.

Model	Number of parameters
Luna	115 K
AlexNet	57.0 M
ResNet18	11.2 M
EfficientNetV2	20.2 M
ConvNext	27.8 M
Vision Transformer	85.8 M
Swin Transformer	27.5 M
DINO	21.7 M

Table 4.1: Number of parameters per model

4.2 Loss

In order to observe how well the model is learning, a way to describe its performance is needed. There are many ways to compute this deviation between the ground truth and the prediction, where these functions return a distance metric. The function used depends on the specific implementation.

This is then used as a way to guide the model into better performance. The one used for this classification is the Cross Entropy Loss function. Since there are only two classes (Defective & Ok), the mathematical function automatically turns into a *Binary Cross Entropy Loss* function.

Cross Entropy

Cross Entropy Loss is defined by Equation 4.1, where $\hat{y}_i$ represents the prediction and y_i the ground truth for the i^{th} class and N is the total number of possible classes.

$$\mathcal{L}_{CE} = -\sum_{i=1}^{N} y_i \cdot log\left(\hat{y}_i\right) \tag{4.1}$$

Knowing that y is a *one-hot encoded vector*, means that every class is 0 except for the class i that is currently being evaluated, which is 1. This eliminates every $log(\hat{y}_i)$ that does not correspond to the class i.

As already mentioned, there are only two classes for this task, thus rewriting the equation delivers the formula for the Binary Cross Entropy Loss.

$$\mathcal{L}_{BCE} = -\left[y_1 \cdot log\left(\hat{y}_1\right) + y_2 \cdot log\left(\hat{y}_2\right)\right] \tag{4.2}$$

Provided that the probability $\hat{y}$ has been normalized (via softmax or similar), the probability for one class will be 100% minus the probability of the other class. Taking y and $\hat{y}$ as the ground truth and prediction respectively for a given image, the equation can be rewritten as follows:

$$\mathcal{L}_{BCE} = -\left[\overbrace{y \cdot log\left(\hat{y}\right)}^{Firstclass} + \overbrace{(1-y) \cdot log\left(1-\hat{y}\right)}^{SecondClass}\right] \tag{4.3}$$

For a batch of images $\mathcal{B}$, each individual loss ℓ_j is computed and the result is then averaged.

$$\ell\left(\hat{y}, y\right) = mean\left(\{\ell_1, ..., \ell_B\}^T\right), \qquad \text{where } \ell_j = \mathcal{L}_{BCE}\left(\hat{y}_j, y_j\right) \tag{4.4}$$

Other examples of Loss functions that can be used, are *Mean Squared Error*, *Mean Absolute Error*, *Huber Loss*, etc. The framework used here allows the creation of an instance of the class *torch.nn.modules.loss.CrossEntropyLoss*, which can then be used to compute the loss between a prediction ($\hat{y}$) and the corresponding ground truth (y) as shown on Listing 3.

Listing 3 Pytorch Implementation: Cross Entropy Loss

```
import torch

# Creation of the CrossEntropyLoss iinstance
criterion = torch.nn.CrossEntropyLoss()

# Random values for the example
ŷ = torch.randn(3, 5, requires_grad=True)
y = torch.empty(3, dtype=torch.long).random_(5)

# Cross Entropy Loss computation
output = criterion(ŷ, y)
```

4.3 Optimizer

Once the error is computed, the model needs a way to interpret this in order to update its parameters and head into the right direction. The direction taken will determine how well the model will improve, if it does improve. As in any optimization task, the goal is to find the parameters that minimize the objective function. For the case of ML, the objective function refers to the loss function and the goal is to minimize it.

Here is a step by step description of how the used optimizer in this project (*Adam*) was developed by Kingma and Ba [51] using previous works.

4.3.1 Gradient Descent

Given a starting point for the parameters (θ) of the network, the gradient (∇) of the objective function ($J(\theta)$) is computed. This leads to the direction which will have the biggest effect on the function. The parameters are then updated in the negative direction of the gradient by a single step (η). This process is then repeated until the improvement of the steps falls inside a certain threshold (ϵ). The size of the step taken is called *learning rate* [52, 53].

Algorithm 3 Gradient Descent [52]

$\theta_c \leftarrow$ Initial Values;
$\theta_n \leftarrow \theta_c - \eta \nabla_\theta J_\theta(\theta_c)$;
while $\|J(\theta_n) - J(\theta_c)\| > \epsilon$ **do**
  $\theta_c \leftarrow \theta_n$;
  $\theta_n \leftarrow \theta_c - \eta \nabla_\theta f_\theta(\theta_c)$;
end while

In order to make a robust decision of where the gradient is pointing, a batch of samples is taken into consideration and the result is averaged among the resulting gradients. This is also known as *Batched Gradient Descent* [53].

4.3.2 Learning Rate

The learning rate (in combination with other parameters of the optimizer such as the magnitude of the gradient) defines how much the parameters change in each step taken while searching for the minimum loss. Usually, the learning rate is chosen ad-hoc [52].

If the learning rate is too small, it will take many steps for it to reach its destination, delaying the learning process. However, if the learning rate is too large, it can miss the minima due to overshooting, thus circling back and forth around it. Therefore, the learning rate can also be adaptive, changing the step size accordingly to speed up the converging process [54].

4.3.3 Stochastic Gradient Descent (SGD)

An issue of Gradient Descent (GD) is when encountering local minima. The initialization of the parameters will determine where the optimization will stop. Depending on where

its parameters are initialized, the gradient will move them in the direction of minimum loss, regardless of local or global minima. [55]

Machine learning architectures have many parameters that will determine the prediction which is then fed to the loss function. This high-dimensionality makes it impossible to plot and extremely hard to analyze and determine the positioning of the global minima [52].

As the name suggests, a stochastic fluctuation is introduced by using each sample to update the parameters, instead of an entire batch. This leads to oscillations, which can help overcome local minima and search for a global one [53].

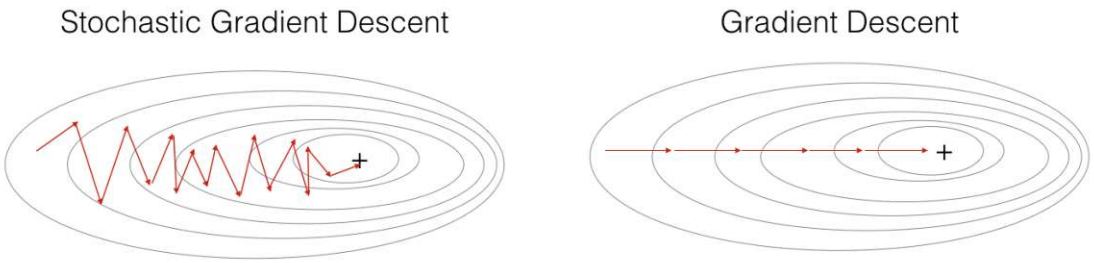


Figure 4.1: Stochastic Gradient Descent vs Gradient Descent [1]

Figure 4.1 shows the main difference between SGD and GD. Here, the black lines denote equal elevation and the black cross is the location of the minima for this concave shape. For GD, the path shown with the red line follows a straight trajectory, whereas SGD shows an erratic movement. Also, it can be seen that SGD requires more steps to converge, however, each step takes less time to compute, as it depends on single inputs and not entire batches to update the parameters [53].

4.3.4 Adagrad

Duchi et al. [56] proposed modifying SGD by adapting the learning rate of the parameters depending on their update frequency. Thus, having greater learning rates for infrequent parameters and smaller rates for frequent ones [53].

This helps sparse features which seldomly update their corresponding parameters. Without this, frequent parameters might converge quickly before the infrequent have had a chance to update enough.

[1] Image taken from: https://datascience-enthusiast.com/DL/Optimization_methods.html

Algorithm 4 Adagrad [56, 53]

$t \leftarrow 1$;
$G_0 \leftarrow$ Zeros;
$\theta_0 \leftarrow$ Initial Values;
$g_{t,i} \leftarrow \nabla_\theta J(\theta_{0,i})$;
$G_t \leftarrow G_0 + g_t^2$;
$\theta_t \leftarrow \theta_c - \frac{\eta}{\sqrt{G_t+\delta}} \times g_t$;
while $\|J(\theta_t) - J(\theta_{t-1})\| > \epsilon$ **do**
  $t \leftarrow t + 1$
  $g_{t,i} \leftarrow \nabla_\theta J(\theta_{t,i})$;
  $G_t \leftarrow G_{t-1} + g_t^2$;
  $\theta_t \leftarrow \theta_{t-1} - \frac{\eta}{\sqrt{G_t+\delta}} \times g_t$;
end while

Where θ_t represents all of the parameters at iteration t, g_t the gradient with respect to the parameters, G_t is a matrix containing the sum of the squares of past gradients along the diagonal, η is the learning rate, ϵ is a small value to avoid zero division and ϵ is the threshold to stop iterating. Variables with the subindex i refer to *per-parameter* operations. Meaning that $g_{t,i} \leftarrow \nabla_\theta J(\theta_{t,i})$ results in each gradient being computed per parameter and stored in its corresponding variable [53].

4.3.5 Adadelta

Zeiler [57] improved Adagrad by limiting the amount of past gradients stored in G. Storing all of the previous squared gradients for a window of size w is inefficient, thus the accumulation is proposed as exponentially decaying. This exponentially decaying running average ($E[g^2]_t$) allows for further optimization without the updating parameters decaying too quickly due to the rapid growth of G_t in frequent parameters [53].

$$E[g^2]_t = \gamma E[g^2]_{t-1} + (1+\gamma)g_t^2 \tag{4.5}$$

Algorithm 5 Adadelta [57, 53]

Require: Decay rate: γ, Constant ϵ
Require: Initial Parameter: θ_1
$E[g^2]_0 \leftarrow 0$;
$E[\Delta\theta^2]_0 \leftarrow 0$;
for $t = 1 : T$ **do**
  $g_{t,i} \leftarrow \nabla_\theta J(\theta_{t,i})$;
  $E[g^2]_t \leftarrow \gamma E[g^2]_{t-1} + (1+\gamma)g_t^2$
  $\Delta\theta_t \leftarrow -\frac{RMS[\Delta\theta]_{t-1}}{RMS[g]_t} g_t$;
  $E[\Delta\theta^2]_t \leftarrow \gamma E[\Delta\theta^2]_{t-1} + (1+\gamma)\Delta\theta_t^2$
  $\theta_{t+1} \leftarrow \theta_t + \Delta\theta_t$
end for

Where

$$RMS[g]_t = \sqrt{E[g^2]_t + \epsilon} \tag{4.6}$$

For this method, it must also be mentioned that it does not require a learning rate (η) as the previous methods did.

4.3.6 Root Mean Square Propogation (RMSprop)

During the same period where Adadelta was being developed, RMS was proposed by G. Hinton[2] independently but was not published. Here the equations are almost identical to Adadelta [53].

$$E[g^2]_t = \gamma E[g^2]_{t-1} + (1-\gamma)g_t^2 \tag{4.7}$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t \tag{4.8}$$

4.3.7 Adam

Adaptive Moment Estimation (Adam) [51] also uses adaptive learning rates per parameter. Similarly to Adadelta and RMSprop, it stores the exponentially decaying average of previous squared gradients (v_t). Additionally, the exponentially decaying average of previous gradients (m_t) is stored, similarly to momentum [53].

$$m_t = \beta_1 m_{t-1} + (1+\beta_1)g_t \tag{4.9}$$

$$v_t = \beta_2 v_{t-1} + (1+\beta_2)g_t^2 \tag{4.10}$$

These represent the estimates for the first (m_t) and second (v_t) moment of gradients. In order to counteract a bias towards zero, the estimates are computed as shown in equations 4.11 and 4.12.

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t} \tag{4.11}$$

$$\hat{v}_t = \frac{v_t}{1-\beta_2^t} \tag{4.12}$$

These values are then used in order to determine the updated parameters (θ_{t+1}) as:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v} + \epsilon}\hat{m}} \tag{4.13}$$

As shown in Figure 4.2, Adam shows the best performance.

[2] http://www.cs.toronto.edu/$\sim$tijmen/csc321/slides/lecture_slides_lec6.pdf

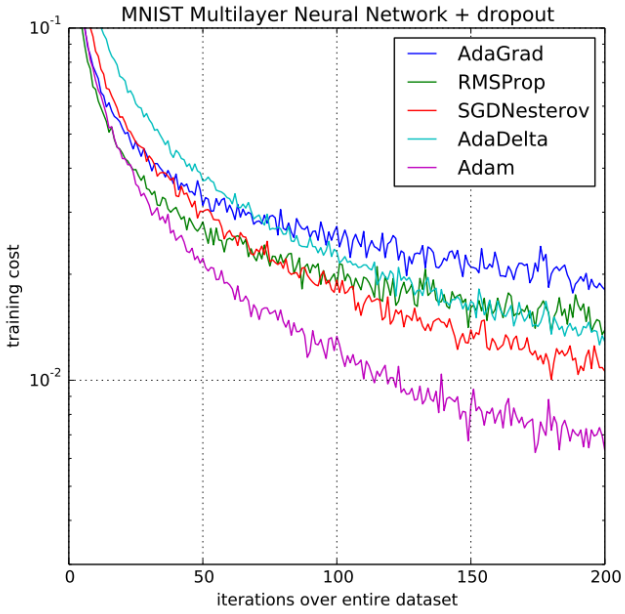


Figure 4.2: Optimizers comparison with neural networks using dropout stochastic regularization on the MNIST dataset. [51]

Pytorch handles the optimization of parameter through an *Optimizer* object bound to the parameters of the *model* to optimize. Listing 4 shows how this is done.

Listing 4 Pytorch Implementation: Adam Optimizer

```
import torch


# Creation of the Adam Optimizer instance
optimizer = torch.optim.Adam(model.parameters(),
                             lr=2e-5,
                             weight_decay=2.5e-4
                             )
```

4.4 Training Steps

In order for the model to train, we need to actually define a loop that takes care of computing the loss and updating the parameters. For this purpose we can divide the training in each iteration into two crucial parts, the forward and the backward step. Also, in order to make sure that the training is working, validation is done to verify the performance of the model with "unseen" images.

Referring back to Equations 3.1 to 3.5 it can be seen that a Subset of the dataset $\mathcal{D}$ has been set aside for training $\mathcal{T}$ and another for validation $\mathcal{V}$.

4.4.1 Training

During training, the subset $\mathcal{T}$ is used in batches. This helps parallelize the process, making better use of the resources, while also helping the optimizer converge faster than using single images. Using the whole subset $\mathcal{T}$ once is called an *epoch*. The architecture can be trained for many epochs until it reaches the desired level of accuracy[3].

Forward Step

In the forward step, a batch of images from $\mathcal{T}$ is run through the architecture. This process yields an output which corresponds to the prediction of the model for each image. This process is also called *forward propagation*. This being a classification problem, the result can be in the form of *logits* or smoothed into probabilities using *softmax* or something similar.

During this process the frameworks collects the gradients making sure to follow the chain of operations done to the input. The framework used in this project is *pytorch*. *Pytorch* takes care of tracking the gradients with *torch.autograd*. This will be needed for the backward step.

Backward Step

After passing the image through the model in the forward pass, the loss must be computed for each result of each image in the batch. Typically the mean loss of the batch is used, but there are other ways it can be used, such as the sum or the median. The loss determines how acceptable the results are and also help determine how each of the parameters has influenced the results.

As mentioned in the forward step, while using *pytorch*, the framework takes care of computing the derivatives during the *forward propagation* and collects these gradients to use during the *backward propagation*[4].

The optimizer is then responsible for taking the computed gradients and update the parameters of the model. If a model is being trained from scratch, then every parameter will be updated. However, if *transfer learning* is being performed, then the amount of parameters updated during the backward propagation will depend on the strategy used.

During transfer Learning, the initial weights of the parameters correspond to a previous training on a large dataset. Figure 4.3 shows the different possible ways to adjust a pretrained model. Strategies one and two correspond to *fine tuning* and strategy number three is also referred to as *feature extraction*. In feature extraction, the pretrained model is used as a backbone and only the classifier is adjusted in order to correctly classify the data. In fine tuning, the unfrozen weights can be adjusted as well, thus changing the pretrained information.

In other words, a strategy needs to be chosen. The weights are frozen or unfrozen according to the strategy. Then during backpropagation only the unfrozen weights are updated according to the optimizer.

[3] Or whatever metric is being used to determine the performance of the model.

[4] Frameworks like *pytorch* require that the gradients are manually reset to zero after using them to update the parameters. If this is not done, the gradients will be accumulated over the course of multiple batches, which might be desirably for Recurrent Neural Networks.

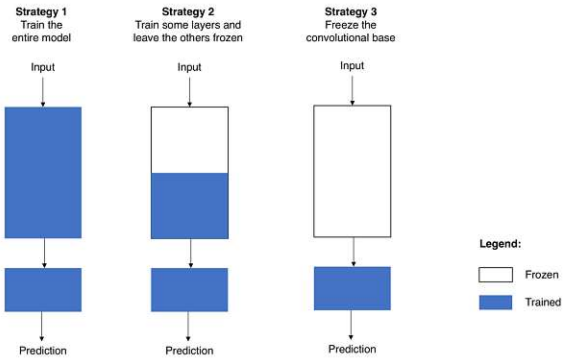


Figure 4.3: Training strategies for transfer learning [58].

Sali et al. [58] present a graphical representation for choosing the right strategy according the dataset analyzed. For manufacturing datasets, it is very common to have a small number of images and they will most likely be dissimilar to other datasets that have publicly available pretrained weights. This is also the case for the present problem, thus positioning the proposed strategy in the third quadrant, where some layers are frozen, but the majority remain unfrozen. There is a high uncertainty as to how much of the model has to be frozen.

Our proposal for the training strategy is to retrain the entire model with a smaller learning rate, even if the dataset has a small number of images and the images are dissimilar to the original dataset used for pretraining the weights. A typical learning rate value used with the Adam optimizer is 0.001. The learning rate used here is $2 \cdot 10^{-5}$.

In order to compare the results and see if there is an improvement with respect to the other strategies, this work was performed for every model using frozen, partially frozen and unfrozen pretrained weights[5]. For the partially frozen weights, the first third of all the layers were frozen as suggested by Figure 4.4.

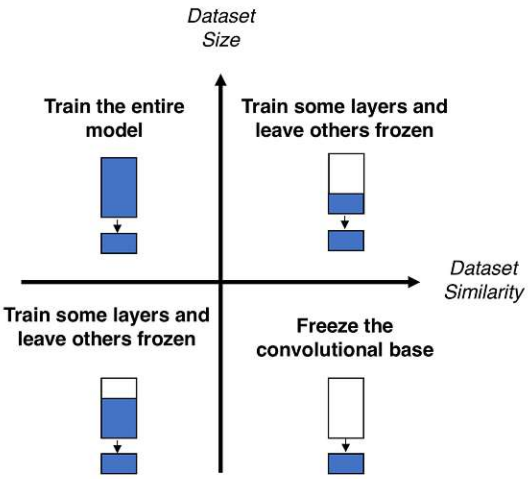


Figure 4.4: Decision map for fine tuning pretrained models [58].

[5]Where the pretrained weights are available. The Luna architecture was used to compare a small model trained from scratch, rather than using transfer learning, since there are no pretrained weights available.

Listing 5 Pytorch Implementation: Train Loop

```
import torch

# Define the model
model = MODEL

# Define loss and optimizer
criterion = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(mode.parameters(), lr=2e-5,
 ↪  weight_decay=2.5e-4)

# Training loop
for epoch in range(max_epochs):
    for images, labels in train_dataloader:
        # Ground Truth and prediction
        y = labels
        ŷ = model(images)

        # Reset the parameter gradients
        optimizer.zero_grad()

        # Compute loss
        output = criterion(ŷ, y)
        output.backward() # Backward propagation

        # Update the parameters
        optimizer.step()
```

As previously mentioned, different training strategies were used for comparison. The following sections (training, validating and testing results) show only the results corresponding to fine tuning unfrozen pretrained weights. The section "Metric Comparison: Frozen vs Partially Frozen vs Unfrozen" in the appendix shows a more complete analysis, comparing the different training strategies for every model.

4.4.2 Training results

The different models were trained on the training subset $\mathcal{T}$. Table 4.2 shows the final results for the different models using different metrics. There is a quick recap on the Metrics used at the end of this section if needed. These metrics are computed during the training loop using the ground truth and the predictions done by the model.

As mentioned before, in the annex are tables comparing the results for the different training strategies. The results shown here correspond only to the strategy proposed here, by fine tuning every parameter.

	Training - $\mathcal{T}$				
	Accuracy (%)	Precision (%)	Recall (%)	F1 Score (%)	Loss (%)
Luna	91.22	90.93	91.22	91.06	$33.18 \cdot 10^{-2}$
AlexNet	97.87	97.81	97.87	97.84	$7.39 \cdot 10^{-4}$
ResNet18	98.76	98.63	98.76	98.69	$4.86 \cdot 10^{-4}$
EfficientNet	98.23	98.21	98.23	98.22	$12.94 \cdot 10^{-4}$
ConvNext	98.98	98.94	98.98	98.96	$1.97 \cdot 10^{-4}$
ViT	98.97	98.90	98.97	98.93	$2.12 \cdot 10^{-4}$
Swin Transformer	98.60	98.57	98.60	98.58	$1.92 \cdot 10^{-4}$
DINO	99.07	99.01	99.07	99.04	$51.91 \cdot 10^{-4}$

Table 4.2: Training metrics

Table 4.3 shows the time needed to train the different models over all layers.

	Epochs	Time	Time/Epoch
Luna	50	1h 29m 14s	1m 47s
AlexNet	6	49m 42s	8m 17s
ResNet18	6	13m 50s	2m 18s
EfficientNet	6	12m 44s	2m 07s
ConvNeXt	6	25m 51s	4m 19s
ViT	6	13m 32s	2m 15s
Swin Transformer	6	10m 06s	1m 41s
DINO	6	22m 55s	3m 49s

Table 4.3: Training time

After training all of the models with the different strategies, the cosine similarity between the fully frozen and the unfrozen models was computed. For this, all the parameters of every model were vectorized and compared between the different training strategies.

	Cosine Similarity to Unfrozen Fine Tuned Weights		
	Random Initialization (%)	Fully Frozen Pretrained Weights (%)	Partially Frozen Pretrained Weights (%)
AlexNet	0.003	82.41	95.69
ResNet18	23.43	99.76	99.72
EfficientNet	34.11	99.98	99.99
ConvNeXt	29.95	97.40	99.56
ViT	9.67	96.80	97.92
Swin Transformer	21.11	98.64	99.78
DINO	34.83	99.94	99.99

Table 4.4: Cosine Similarity between unfrozen fine tuned weights and

4.4.3 Validating results

After each epoch, the performance of the model is verified. In order to do this, the model has to interact with images that have not been used during training. For this purpose the Subset for validation $\mathcal{V}$ of the Dataset $\mathcal{D}$ is used.

The process of validation looks very similar to the training. Batches of images (from another non-overlapping subset of $\mathcal{D}$) are passed through the model. This yields outputs which are then used in the loss function to determine how good the model is performing.

The main difference is that the gradients are **not** computed and are **not** back propagated with the optimizer. This ensures that the model does not remember the image.

The main purpose of validating is making sure that the model is generalizing and not overfitting to the training data. If the model tends to perform exceptionally well on the training data but poorly on the validating one, then the model is overfitting. This might be a good place to stop the training

When training and validating, there are other metrics that can be used in order to determine how good the model is performing.

Table 4.5, shows the performances during the final epoch of training on the validation subset ($\mathcal{V}$).

	Validation - $\mathcal{V}$				
	Accuracy (%)	Precision (%)	Recall (%)	F1 Score (%)	Loss (%)
Luna	91.35	90.49	91.35	90.77	$33.07 \cdot 10^{-2}$
AlexNet	98.17	97.62	98.17	97.86	$8.38 \cdot 10^{-3}$
ResNet18	99.13	98.98	99.13	99.05	$9.79 \cdot 10^{-3}$
EfficientNet	99.51	99.32	99.51	99.41	$8.59 \cdot 10^{-3}$
ConvNext	99.19	99.32	99.19	99.25	$4.55 \cdot 10^{-3}$
ViT	99.39	99.18	99.39	99.28	$3.07 \cdot 10^{-3}$
Swin Transformer	99.21	99.05	99.21	99.13	$10.95 \cdot 10^{-3}$
DINO	99.21	99.89	99.21	99.08	$6.60 \cdot 10^{-3}$

Table 4.5: Validation metrics

4.4.4 Testing results

After training and validation, these trained models were used with the test subset $\mathcal{R}$. While training and validating are done on the same loop (sharing the number of epochs), the testing is done afterwards. The testing loop is done with the same principle, selecting batches of the subset, getting the predicted output from the model and computing the loss and other metrics.

	Testing - $\mathcal{R}$				
	Accuracy (%)	Precision (%)	Recall (%)	F1 Score (%)	Loss (%)
Luna	99.41	98.69	99.41	98.92	$32.11 \cdot 10^{-2}$
AlexNet	99.80	99.58	99.80	99.67	$7.37 \cdot 10^{-3}$
ResNet18	99.80	99.59	99.80	99.67	$10.47 \cdot 10^{-3}$
EfficientNet	99.80	99.59	99.80	99.67	$10.58 \cdot 10^{-3}$
ConvNext	99.89	99.81	99.89	99.84	$6.38 \cdot 10^{-3}$
ViT	99.80	99.59	99.80	99.67	$8.20 \cdot 10^{-3}$
Swin Transformer	99.80	99.59	99.80	99.67	$12.50 \cdot 10^{-3}$
DINO	99.70	99.40	99.70	99.52	$13.52 \cdot 10^{-3}$

Table 4.6: Test metrics

As it can be seen on Tables 4.2, 4.5 and 4.6, all of the models have great performances in every metric during training, validating and testing.

4.5 Metrics

Here is a quick recap on the metrics used here to quantify the performances. If the reader is familiar with them, then feel free to skip to the next section.

Metrics are needed to quantify the models. They help determine how performant they are and see what the strengths and weaknesses of each model are. Here is a short description of the metrics used in this work[6]. The following table shows the possible combinations between the ground truth and the prediction (classification) given by the model. In statistics the *false positive* term is denoted as *type I* error and *false negative* as *type II*.

		Predicted	
		Negative	Positive
Actual	Negative	True Negative (TN)	False positive (FP)
	Positive	False Negative (FN)	True Positive (TP)

Table 4.7: Relation between atcual labels and predicted ones

4.5.1 Accuracy

Accuracy is a metric that evaluates how many times the model is correct with respect to the entire dataset.

$$Accuracy = \frac{TP + TN}{TP + FP + TN + FN} \tag{4.14}$$

[6]Here is a blog with metrics listed and explained: `https://tinyurl.com/3aur8yhe` `https://tinyurl.com/4d7z9u8d`

4.5.2 Precision

Precision reflects how good the model is at performing a specific task. This metric is also known as *specificity*.

$$Precission = \frac{TP}{TP + FP} \tag{4.15}$$

However, the number of predictions made is irrelevant. The worst case scenario for this metric is cherry picking only the most certain values as positive, avoiding *false positives*, at the expense of predicting many actual positive values as being negative.

4.5.3 Recall

In contrast, recall ensures to not miss any actual positive inputs. This metric is also referred to as *sensibility*.

$$Recall = \frac{TP}{TP + FN} \tag{4.16}$$

The worst case scenario for this metric is classifying everything as positive, regardless of the false negatives that it might predict.

4.5.4 F1-Score

As a way to counteract the shortcomings of both precision and recall, Rijsbergen [59] introduced a metric that uses both of them.

The general equation for the F_β-*Score* has the parameter β, which influences how the precision and recall influence the result.

$$F_\beta = (1 + \beta^2) \cdot \frac{p \cdot r}{\beta^2 \cdot p + r} \tag{4.17}$$

Using β as 1 results in an equal distribution between precision and recall.

$$F_1 = \frac{2 \cdot p \cdot r}{p + r} \tag{4.18}$$

Figure 4.5 shows a visual representation of an equal distribution between precision and recall when using $\beta = 1$[7].

[7] `https://andersource.dev/2019/09/30/f-score-deep-dive.html` has a great visualizer, where beta can be modified to see the effect it has on the curve.

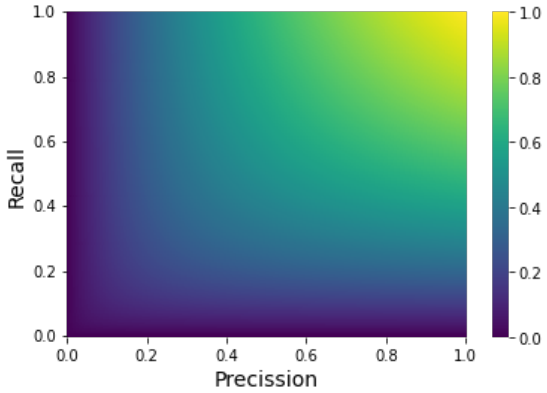


Figure 4.5: Visualization of the F1 score

This metric penalizes wrongful detection (precision - FP) and the lack of detection of positive inputs (recall - FN) equally. A model which makes few mistakes predicting positive inputs and also misses few actual positives will score highly in this metric.

4.5.5 Intersection over Union (IoU)

This metric is used in object detection to determine how much overlap exists between the actual location of the object within the image and the predicted location. As the name suggests, the formula consists of dividing the intersection by the union. This is also known as the *Jaccard Index*.

$$IoU = \frac{\text{Intersection}}{\text{Union}} = \frac{\quad}{\quad} \tag{4.19}$$

This section explained the necessary steps to train/fine tune the different models. This was then used to train the different models and to do the corresponding validation during training and testing afterwards. It was shown that all of the models used achieved near perfect performances across the different metrics used.

Even though the validation and testing could be enough to accept a model as working correctly, this thesis wanted to explore if the models were learning to actually detect the defects on the images. The next section focuses on explaining a visual inspection of the models, in order to determine the relevance of the defects during the classification.

5 Evaluation: Interpreting the predictions of defective images

Interpretability attempts to dissect these models in such a way that they can be understood by a person. Rather than treating the models as a black box, characteristic features that determine which class is being selected are extracted in order to clearly visualize where the architecture focuses and if they actually learn to detect the defects present on the images [60].

An often cited example of the necessity for the understanding of classifiers is given by Ribeiro et al. [61]. One of the examples used here is the classification of images between wolves and huskies. After training their model, they examined the model and found that the model was only focusing on whether there was snow present on the image or not (snow=wolf, no snow=husky).

This section explains the different methods used to interpret the architectures for CNN and for Transformers. These methods are the ones used to determine if the architectures have indirectly learned to localize the defects. In order to quantify the influence of the defects towards the correct classification of defective images, a ground truth is needed. When the actual location of the defects is not present, they have to be manually annotated to have a ground truth to compare to. Here, the annotation is presented, followed by the interpretations of the various models.

5.1 Annotation of defects

This project consists of a weakly supervised dataset, where only the labels of the images are provided and not the location of where the defects are found on the image, they had to be manually annotated. *PixelAnnotationTool*[1] was used for the annotation of the images.

Figures 5.1(a) and 5.1(b) show an image of a defective iron casting with its corresponding mask made manually using *PixelAnnotationTool*. The annotation was done for the 453 images with defect of the Test ($\mathcal{R}$) subset. In all of them, the defects were highlighted with red, the casting with green and the background with purple.

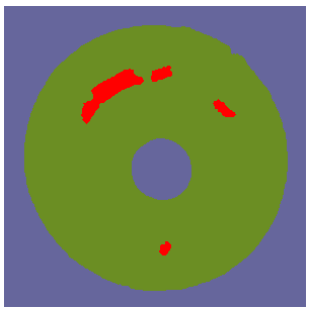
On these images, it can be seen that the annotation of the defects is not perfect. The annotation of the defects is very subjective, since the person annotating has to determine which annotations are important enough to mark and which to leave. In this same example we can see that on the lower part of the middle ring is a defect annotated, however on the

[1] The tool can be found here: `https://github.com/abreheret/PixelAnnotationTool`

upper left side of the innermost ring are some irregularities that could also be selected as such.

For the annotation, only some parts of the images are selected and the tool uses a technique called "watershed" to decide which pixels belong to which label (*background, cast, defect*). So, even though the annotation was done manually and the best effort was made to select only the defects, the annotation is not perfect. This is something to be taken into account when comparing the performances of the models with respect to the annotated images.

(a) Defective iron casting.



(b) Mask of defective iron casting.

Figure 5.1: Defective iron casting and its corresponding annotated mask. Image (***cast_def_0_221.jpeg***) taken from the test subset ($\mathcal{R}$).

5.2 CNN

Many studies have addressed the need to visualize relevant features from CNNs by highlighting the most relevant pixels for the prediction (e.g. [62, 63, 64, 65]), where changes in intensity of said pixels might lead to a change in prediction. However, these methods are not class specific.

5.2.1 CAM - Class Activation Map

Zhou et al. [66] proposed class activation maps as a way to visualize relevant features of an input image with respect to the predicted class. Here the fully connected layers at the end of the architecture are replaced by convolutional networks and global average pooling. The network is then retrained to classify images with the new structure, where each weight that is connected to the classification layer represents the importance of each feature map in the global average pooling stage.

Figure 5.2: CNN with global average pooling at the end for Class Activation Mapping [66].

This can be described through the following equation, where Y^c is the prediction Y for a given class c. The weights w_k^c correspond to the weights connecting each feature map A^k with the predicted class Y^c. A_{ij}^k refers to the activations of the feature map at the location (i, j). Z corresponds to the number of pixels in the feature maps.

$$Y^c = \sum_k \underbrace{w_k^c}_{\text{class feature weights}} \overbrace{\frac{1}{Z}\sum_i \sum_j}^{\text{globalaverage pooling}} \underbrace{A_{ij}^k}_{\text{feature map}} \tag{5.1}$$

The result is shown in Figure 5.2. Here, each feature map learns to distinguish certain aspects of the image and the weights make sure to reference the correct class. Thus if a different class was being inspected, other than the one shown on the image (Australian terrier), then the only thing that would change would be the weights. The feature maps remain the same.

The downside on this approach is having to manipulate the original architecture and having to retrain it in order to get these results.

5.2.2 Grad-CAM - Gradient Class Activation Map

In order to bypass the modification and retrain of the network, Selvaraju et al. [5] proposed a modified version of Class Activation Maps.

Through generalization of CAM they arrived at the conclusion that the necessary values to compute a way to generate visual explanations are already present on the architecture without the need for restructure or retraining.

$$L_{\text{Grad-CAM}}^c = ReLU\left(\sum_k \alpha_k^c A^k\right) \tag{5.2}$$

Where

$$\alpha_k^c = \overbrace{\frac{1}{Z}\sum_i\sum_j}^{\text{global average pooling}} \underbrace{\frac{\partial y^c}{\partial A_{ij}^k}}_{\text{gradients via back-prop}} \tag{5.3}$$

The way these values are retrieved is by first computing the gradients of the score class c (y^c) with respect to the feature map activations (A^k) of a convolutional layer ($\frac{\partial y^c}{\partial A_{ij}^k}$). During backpropagation, these gradients are average pooled over the width and height to obtain the importance weights (α_k^c) as shown on Equation 5.3.

The weights in α_k^c represent a partial linearization of the network downstream from the feature maps A^k for a specific class c. The feature maps (A^k obtained in the forward step) are then multiplied by the weights (α_k^c obtained with the help of the backward step) resulting in the most important features to determine said class. This result is then passed through a *ReLU* layer to only get the positive values, meaning the ones that positively influence the classification.

Figure 5.3 shows this process in more detail. There are additional concepts not relevant for this project (*Guided Grad-CAM*), which are also proposed in the same publication.

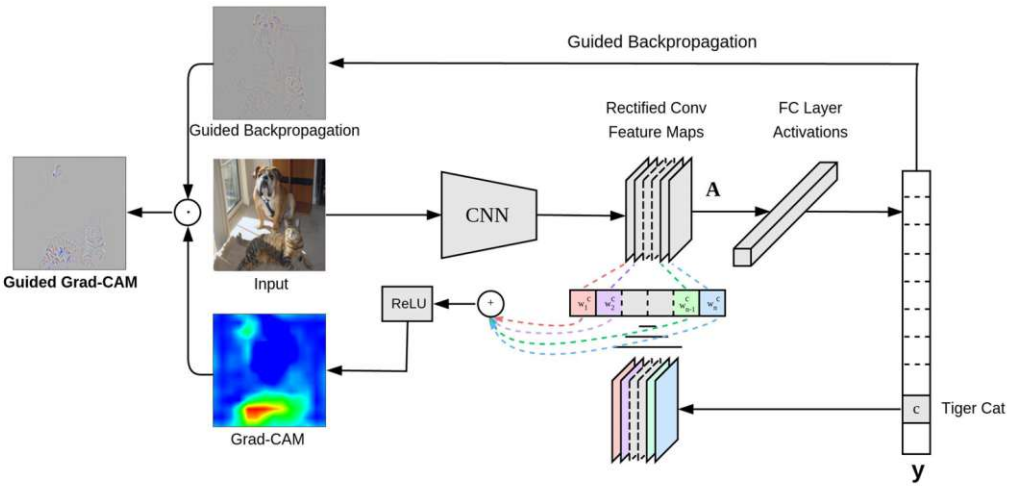


Figure 5.3: GradCAM architecture [5].

5.2.3 GradCAM - Implementation

PyTorch has a way to allow the user to add functionality to an architecture during the *forward / backward* propagation on a user-defined layer, even after the model has already been created. This way to access the model is called ***hook***. During the *forward* propagation, the feature map (A_k) can be extracted and the corresponding gradients with respect to the predicted class ($\frac{\partial y^c}{\partial A_{ij}^k}$) can be obtained during the *backward* propagation.

Listing 6 Registering hooks to extract feature maps and gradients

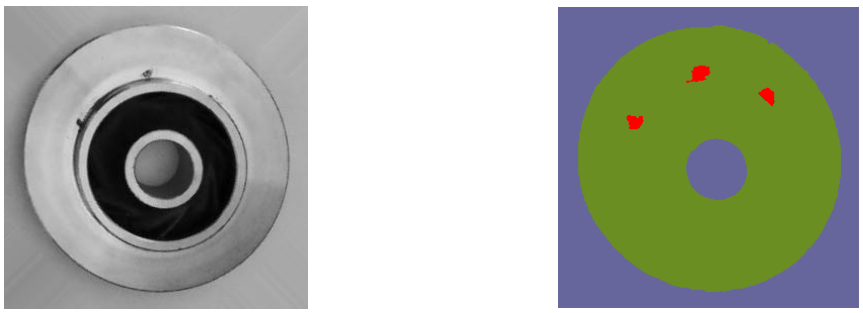
```
class GradCAM():
    def __init__(self, model, target_layer, **kwargs):
        ...
        target_layer.register_forward_hook(
                self.save_feature_map
            )
        target_layer.register_backward_hook(
                self.save_grad
            )

    def save_feature_map(self, module, input, output):
        self.feature_map =  output.detach()
    def save_grad(self, module, grad_in, grad_out):
        self.grad = grad_out[0].detach()

    def __call__(self, x):
        output = self.model(x)
        index = output.argmax(dim=1)
        self.model.zero_grad()
        (F.one_hot(index, 2) * output).sum().backward()

        cam = F.relu(
            (self.feature_map[0] * (self.grad.mean(dim=(2,3))
            [0, :])[:, None, None]).sum(dim=0))
        cam = cv2.resize((cam/cam.max()).cpu().numpy(),
                (x.shape[-2], x.shape[-1]))
        return cam, index
```

In order to demonstrate how the snipped performs in the different CNN architectures, the image ***cast_def_0_1618.jpeg*** from the test subset ($\mathcal{R}$) was used.



(a) Defective iron casting. (b) Annotated mask.

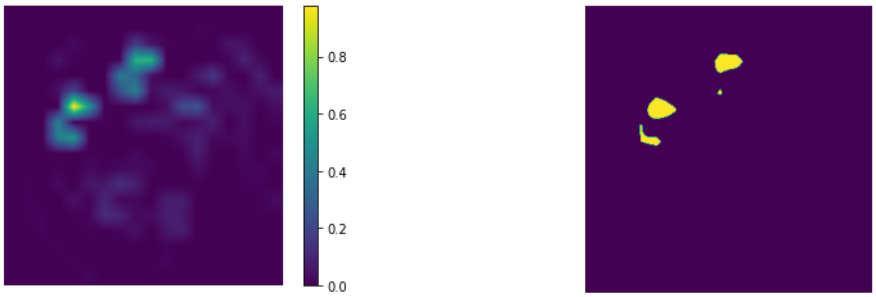
Figure 5.4: Defective iron casting and its corresponding annotated mask.

In order to determine which layer to perform the GradCAM to, it was applied to every possible layer and plotted. This was then visually inspected to determine which one matches the defect locations the best. This process was done for every CNN architecture. All of the GradCAMs have also been normalized between 0 and 1. However, an individual

threshold has to be selected per architecture to only select the most relevant features, trying to minimize the error in the *IoU* computation.

Luna

For the Luna architecture, the ***block4.maxpool*** layer was chosen with a threshold of 0.4. Figures 5.5(a) and 5.5(b) show the result for GradCAM using the example image with a defect. It can be clearly seen that the network has learned to identify the location of the defects.



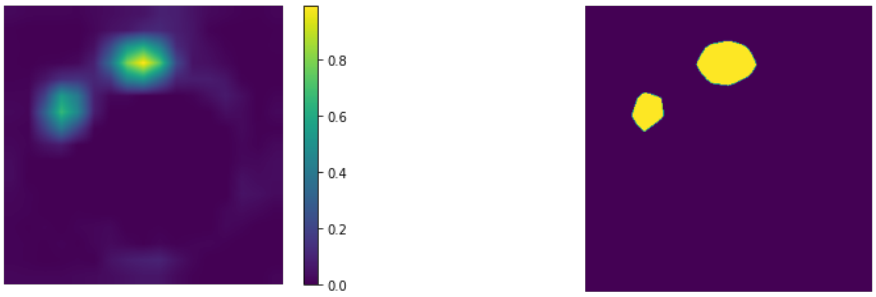
(a) GradCAM for Luna. (b) Thresholded GradCAM

Figure 5.5: Resulting GradCAM for a defective sample using Luna, before and after applying a threshold of 0.4.

The resulting mask is then used alongside the annotated mask to get the intersection and union.

AlexNet

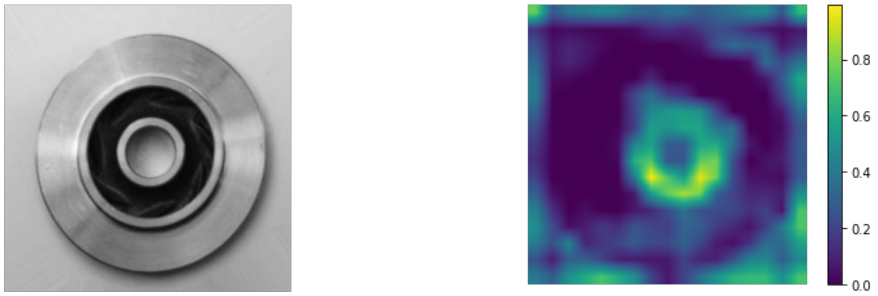
The same process was done for the other architectures. First, GradCAM was applied to every possible layer in the model to visually inspect which one shows the desired feature. After selecting the desired layer to apply GradCAM, a threshold is selected. For Alexnet, the ***features.11*** layer was selected with a threshold of 0.5. Figures 5.6(a) and 5.6(b) show similar results to the previous architecture.



(a) GradCAM for AlexNet. (b) Thresholded GradCAM

Figure 5.6: Resulting GradCAM for defective sample using AlexNet, before and after applying a threshold of 0.4.

Using the same layer and same parameters, an image without defects can also be analyzed to inspect what the model is focusing on. Figure 5.7 shows the resulting GradCAM for the image ***cast_ok_0_235.jpeg***. Here the resulting GradCAM appears to be focusing on some defect that the image might have. However, it is important to mention that the values have been normalized between 0 and 1. Without this normalization, for AlexNet, the maximum values for images **with defect** range between 0.20 and 0.30. The maximum values for images **without defects** are ten times smaller, ranging from 0.015 to 0.25. This GradCAM with the thresholds mentioned, show that the architecture is really focusing on the defect to select the appropriate classification.



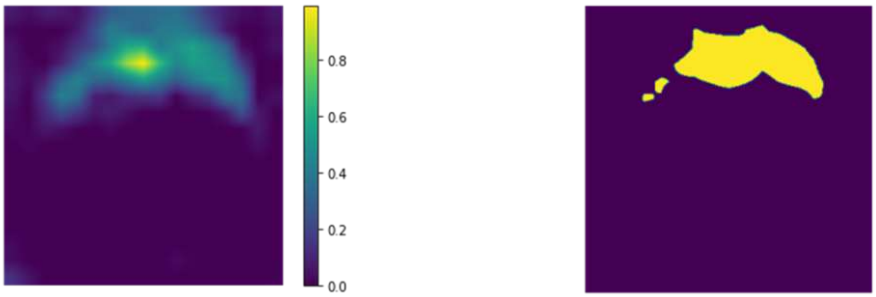
(a) cast_ok_0_235.jpeg (b) GradCAM for cast_ok_0_235.jpeg

Figure 5.7: Iron casting without defect and its resulting GradCAM for AlexNet.

It must also be mentioned that these models have been fine tuned without freezing the parameter. Thus the layers have the ability to learn new features that might be crucial to classify the data. Freezing the parameters and only training the last layer is also a very useful way to fine tune models, especially when the new dataset is similar to the original training dataset. However, this dataset, as well as many datasets that are needed for zero defect manufacturing, will not resemble the data on *ImageNet* or other datasets where pretrained weights for these models are available.

Figure 5.8 shows the same process being applied to the same architecture, but with the frozen pretrained weights on *ImageNet* for the defective image ***cast_def_0_1618.jpeg***. Here, it can be clearly seen that not freezing the pretrained weights does yield an advantage when trying to interpret the model. Also, using the same hyperparameters, training epochs and overall training workflow, but freezing the previous layers to the last, resulted in training metrics around 70% (*Accuracy, Precision, Recall and F1*), which is clearly inferior to the results presented in table 4.2.

The classification layer might be done using something more complex after the frozen parameters, using multiple layers, also using regularization and more epochs. However, this simple setup of not freezing the pretrained weights and fine tuning has generated great results. Also, as the contrast between Figures 5.6 and 5.8 shows, the unfrozen fine tuned model is more interpretable, highlighting the features that might be crucial to classify the image as defective.

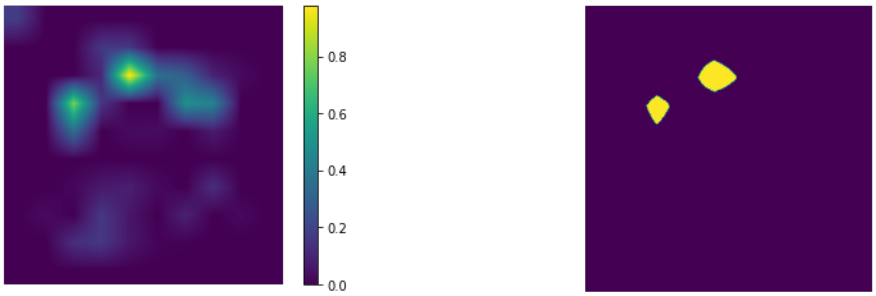


(a) GradCAM AlexNet with frozen weights (b) Thresholded GradCAM

Figure 5.8: GradCAM of defective iron casting using AlexNet with frozen weights and the thresholded result.

ResNet

For ResNet the layer used was ***layer4.0.downsample.0*** with a threshold of 0.4. As Figures 5.9 shows, this model also managed to focus on some of the defects on the image.

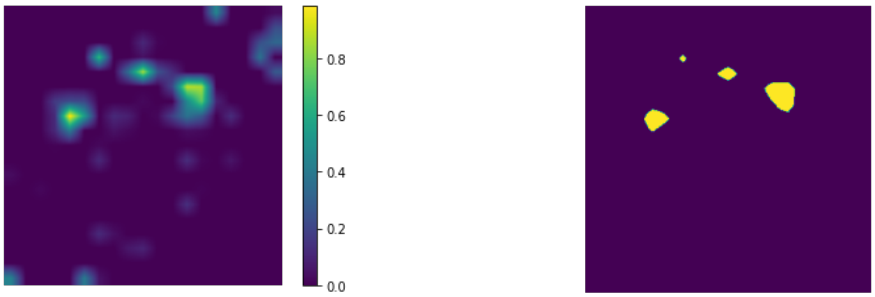


(a) GradCAM for ResNet (b) Thresholded GradCAM

Figure 5.9: Resulting GradCAM for defective sample using ResNet18, before and after applying a threshold of 0.4.

EfficientNet

For the EfficientNet architecture, ***features.6.0.block.0*** with a threshold of 0.4 was used. This is the first architecture to detect all of the three defects originally annotated on the image.

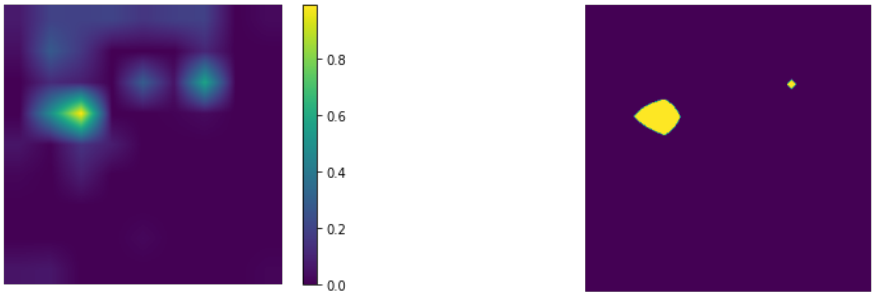


(a) GradCAM for EfficientNetV2 (b) Thresholded GradCAM

Figure 5.10: Resulting GradCAM for defective sample using EfficientNetV2, before and after applying a threshold of 0.4.

ConvNeXt

For the ConvNeXt architecture, the ***features.5.7.block.6*** layer was used with a threshold of 0.4.



(a) GradCAM for ConvNeXt (b) Thresholded GradCAM

Figure 5.11: Resulting GradCAM for defective sample using ConvNext, before and after applying a threshold of 0.4.

5.3 Attention Maps on Transformers

Raghu et al. [67] compare Vision Transformers to Convolutional Neural Networks in order to determine if they "see" similarly. Through the use of representational similarity techniques, they come to significant differences between the models, while also concluding that Vision Transformers do share some CNN properties. Among these similarities lies the importance of local information aggregation at lower layers.

Caron et al. [68] extended on the Vision Transformer by proposing a model trained through self supervised learning, which they denote as self-**di**stillation with **no** labels (DINO). In this paper they analyze the performance of the model by selecting the self-attention of the [CLS] Token with respect to the patches on the heads of the last encoder block from a Vision Transformer.

The result can be seen in Figure 5.12. Here, the architecture learned to extract the relevant features from the image without the use of labels. This exact same technique of

accessing the attention map can be used with any architecture that has an attention layer, such as ViT and Swin-Transformer.

Figure 5.12: Self-attention of the [CLS] Token on the heads of the last layer of a Vision Transformer trained with no supervision[68]

The way their training was done was by using a teacher and a student network as shown in Figure 5.13. Here, both networks are presented with different augmented variations of the same image and the loss is computed according to the difference in the prediction in both networks. Only the students parameters are updated through back propagation. The new student parameters are then used to update the parameters of the teacher using an exponential moving average to make it more robust.

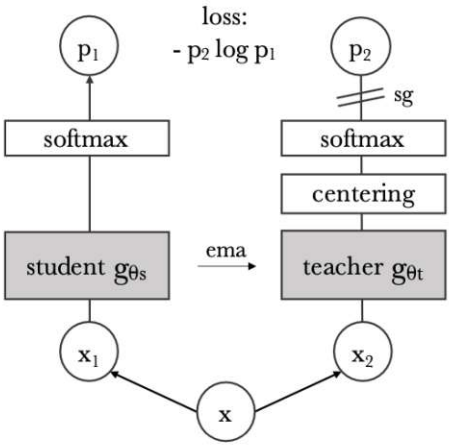


Figure 5.13: Self distillation with no labels[68]

They then proceeded to compare the results with other models trained with supervised data and found a significant improvement.

This process, can also be used to plot how any model using self-attention behaves. This can be used to select any block of said architectures, and even select single heads if needed in order to determine where the architecture is focusing given an input image.

5.3.1 Extracting the Attention Maps

In this section, the different attention maps from the three Transformer architectures used were accessed and plotted.

ViT

In order to get the attention maps from architectures such as ViT and Swin Transformer, pytorch provides a feature extractor functionality, which allows to get intermediate results from the network during the forward pass. Since the predefined architecture for ViT in pytorch does not return the attention maps, the intermediate result before the attention mapping can be extracted to get the self-attention **with** the corresponding attention maps.

Since the architecture counts with twelve blocks, the feature extraction followed by the self-attention was done with everyone to see if any of them shows patterns of recognizing the defects on the images. As Figure 5.14 shows, the seventh encoder block (Encoder Block 6, due to zero indexing) is the one which focuses on the errors the most.

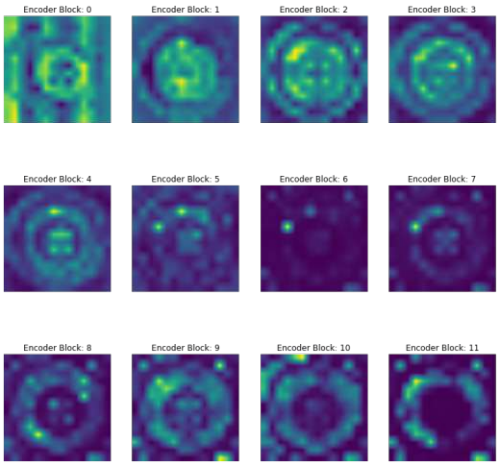


Figure 5.14: ViT: Self-Attention in every Encoder Block for image with defects.

In order to further inspect these results, each individual *Head* from this specific *Encoder Block* was plotted. As Figure 5.15 shows, there are many heads that might be focusing on the errors, such as heads 2, 3, 7 and 9. Some of these were used to compute the IoU to see the performances, but the mean of all the heads yielded the best results, which corresponds to the image of the Encoder Block 6 in Figure 5.14.

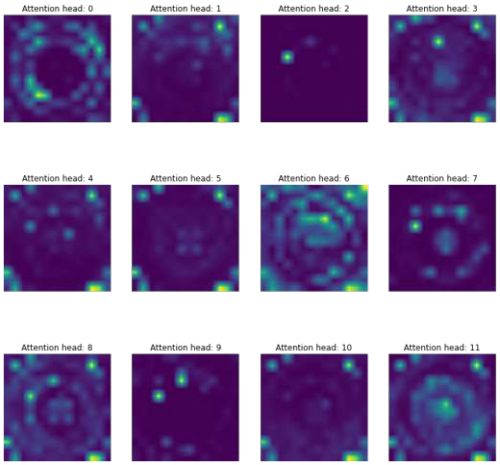


Figure 5.15: ViT: Self-Attention Heads for Encoder Block 6 for image with defects.

Swin Transformer

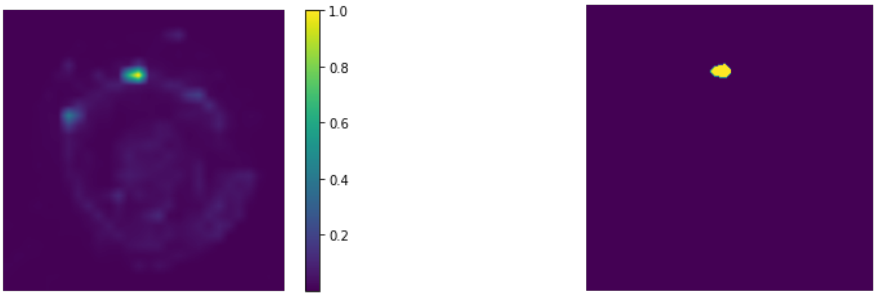
For the Swin Transformer, there were more steps to take than for the Vision Transformer. Here the same procedure as for the ViT was done. First get the values prior to the attention layer in each block. Then the result is passed through the attention function, but the attention is also returned.

The attention used was the one from *Stage 3*, *block 2*. For this, the layer ***features.5.2.norm1*** was computed via the feature extractor and then passed through the attention mechanism. The resulting shape will depend on the stage used. In every stage the windows have a size of $7x7$, however, in earlier stages these windows encompass less pixels and in later stages the field of view grows. What this means is that in later stages there are less patches and in earlier, more. Thus, if there are multiple patches, they have to be concatenated to get the attention of the full image.

DINO

For the DINO model, since the original paper [68] focused on extracting the attention to analyze these maps, the architecture comes with a built-in function to extract the self-attention for any given block or even a specific head of the blocks such as it was manually done in the previous models. This means that with a simple line of code the attention map is available to analyze.

Here, the self-attention corresponding to the last block was used, using the mean of every head inside of this block. The threshold used was 0.4.



(a) GradCAM for DINO (b) Thresholded GradCAM

Figure 5.16: Resulting GradCAM for defective sample using DINO, before and after applying a threshold of 0.4.

5.4 Performances

For the CNN models, the GradCAM was extracted for every image with defects in the *Test* subset ($\mathcal{R}$). These maps where then passed through a threshold and compared with the manually annotated masks. With this information, the intersection and union where computed to get the IoU metric.

For the Transformer models, the same process was done, using the attention maps instead of the GradCAM.

As a way to compare these models to some ground truth, the same process was done using the frozen and partially frozen fine tuned models. Figure 5.17 shows the comparison

for all models, depicting IoU for the fully frozen backbone in light orange, the partially frozen in light blue and the unfrozen in green[2]. This figure clearly shows improvement for every architecture, meaning the models learned to focus more on the defects on the images.

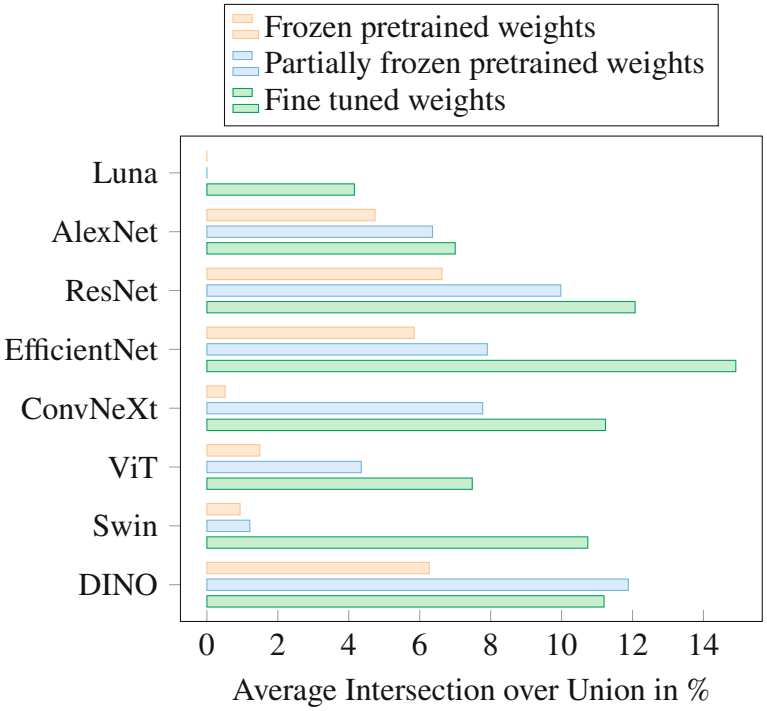


Figure 5.17: Intersection over Union frozen vs. unfrozen weights

As it can be seen, every model indirectly learned to focus on the defects, even though this was not a metric used during training. Using the baseline of frozen pretrained weights as comparison, we can see that every model improved greatly. The values were taken on the same layer for each model for the different training strategies.

The only model that showed better performance when freezing a third of the layers was DINO. All other models benefited from leaving the parameters unfrozen.

To measure the precision of the locations, the Intersection over Prediction was also done. Here, the intersection is compared to what the model is predicting to be the most important features to classify the image. Thus, it can be interpreted as a metric comparing the intersection to the prediction where the defects are, since the defects should be the defining features of the classification. It must be mentioned that this metric was created by the author of this thesis.

Once again, the frozen pretrained weights are taken as a baseline to see how the fine tuning of the parameters affects the location of the defects. Figure 5.19 shows how the different models behave.

[2]As mentioned before, the Luna architecture does not count with pretrained weights, since the architecture was modified from the original source from 3D CT-scans analysis to 2D image analysis.

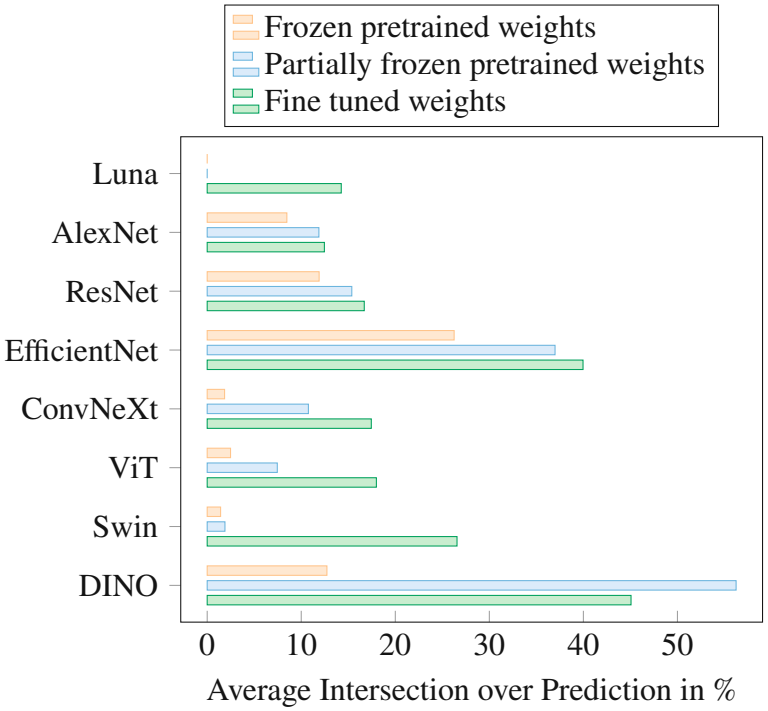


Figure 5.18: Intersection over Prediction frozen vs. unfrozen weights

What Figure 5.19 shows is the percentage of the predicted region of interest that intercepts with the ground truth. The improvement from the last figure to this one means that, although not all of the defects on the image are detected, a big part of the region of interest is located where part of the defects are. This can be specially appreciated for *EfficientNet* and *DINO*, where 40% of the region of interest lies inside the annotated defects.

This metric shows the clear improvement of the models output lying inside of the ground truth. This metric is analogous to computing the precision of the model, where the intersection between ground truth and prediction is the *true positive*, and the sections predicted as being relevant for the classification that do not belong to the annotation of a defect corresponds to the *false positive*.

Lastly, the Intersection over Ground Truth was computed. This is also a new metric that helps understand what happened during the fine tuning of the different models with the different training strategies. While IoP is the analogous to precision, IoGT is the analogous to recall. Here, *false negative* is the area of the manually annotated defects that does not show up on the activation maps.

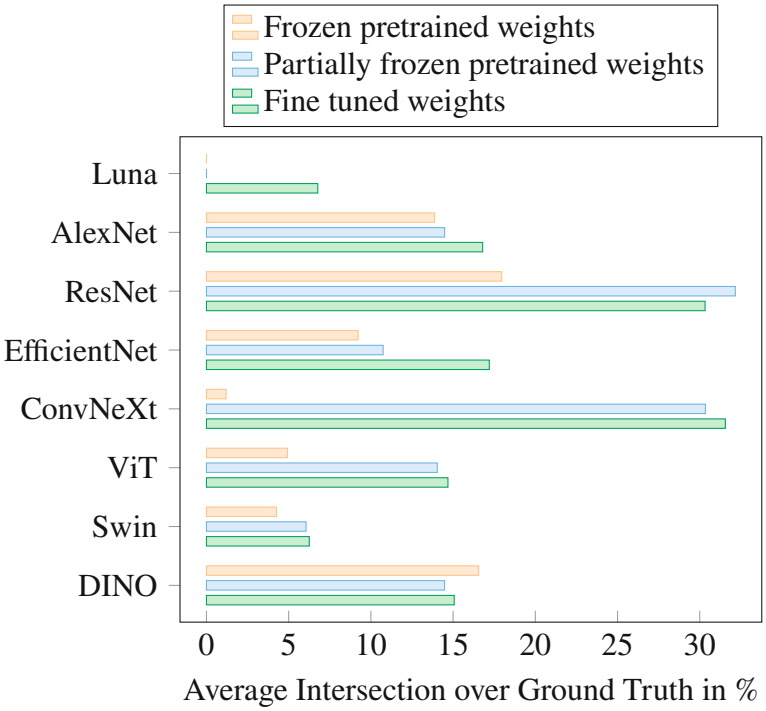


Figure 5.19: Intersection over Ground Truth frozen vs. unfrozen weights

Using both metrics (IoP and IoGT), it can be appreciated how certain models like DINO and EfficientNet learned to detect the defects and were satisfied with encountering some part of them. The predictions made by these models were very precise in comparison to the other models. On the other hand, they covered less area from the ground truth.

It must be mentioned that these results have to be taken with a pinch of salt, since the ground truth annotations are not perfect and are very subjective to what was deemed relevant to mark as a defect at the time of annotation. However, even though the results may vary when revising the ground truth annotations, every model has shown a clear improvement when detecting where the defects are located.

Almost every model followed the same pattern, showing the worst results with the fully frozen models and improving gradually towards the fully unfrozen ones.

6 Conclusion

The increasing automation of the manufacturing industry has led to a higher acquisition of data. This data can be in the form of various sensors measuring different variables during the process. Part of these variables can be observed using cameras on the production line. These images can be then used to detect defects that might be present on the products, leading not only the rejection of imperfect products, but also to the possible correction of the course of action in order to avoid more defects.

However, in order to successfully achieve the classification of images during production, a dataset containing all of the classes is needed. For the purpose of defect detection, it will fall into the category of binary classification, with images *with* and *without* defect. The bigger and the more detail this dataset has (i.e. labeling, annotation, location of the defects), the more expensive it will be. In production environments there is a high emphasis on costs, thus having a curated dataset with perfectly annotated defects will most likely not be the priority.

This thesis looked at existing alternatives to train neural networks with a small dataset which is weakly annotated. Meaning only the labels are present to determine if the images belong to the *defective* or *non-defective* class. For this purpose, one architecture was trained from scratch, while many others where fine tuned from pretrained models. The models with pretrained weights available were trained using three different strategies, freezing everything except the classifier, freezing the first third of the layers and not freezing any parameter.

In this work we proposed the use of transfer learning instead of designing an architecture for every new ZDM task. It was also proposed to fine tune the entire model, even if the data to analyze is dissimilar to the data used for pretraining the weights.

Looking only at the metrics during training, validation and testing it can be seen that the model trained from scratch performed well and could compare to the other models. This model achieved results over 90% for every metric during training, validation and testing, whereas the fine tuned models had results above 97%. However, this requires more work during designing the architecture and more computational power for training, as it needs to train for longer time in order to achieve comparable results to the ones achieved by fine tuning. A bigger architecture and more training time would be needed to close the gap even further, along with more regularization and testing different approaches.

Among the different strategies for fine tuning, there was no significant difference between the partially frozen pretrained weights and the unfrozen pretrained weights. Computing the average of the different metrics along the models, resulted in a difference below 0.2% between partially frozen and unfrozen. Both of these strategies yielded

significantly better results than using the frozen pretrained weights.

Computing the average metrics between every model gives an overview of how the models benefit from allowing the model to adjust its parameters to the specific dataset. Even when sharing a 96.41% cosine similarity between the parameters of the unfrozen and fully frozen models, the unfrozen showed significant improvement. For accuracy during training it went up from 78.45% to 97.78%, for validation from 82.39% to 99.12% and testing from 90.40% to 99.80%. Precision, recall and F1 score showed similar results.

When analyzing the activation maps of the different architectures, it could be determined that the defects do play a significant role for the correct classification of the images. It was also shown that leaving all of the parameters unfrozen leads to a better interpretability of the decisions made by the models. This is a key feature when auditing or justifying the decisions of a model. Although the partially frozen and unfrozen models have a cosine similarity of 98.95% among their parameters, the unfrozen managed to get better results for IoU, IoP and IoGT than the partially frozen. These differences varied greatly among the models, ranging from a 10% IoU increase for AlexNet up to a 60% IoU increase for EfficientNet.

It was also noted during visual inspection that the activation maps do not work properly on every image, especially when the image has many defects. For future work it is recommended to explore different algorithms to show the key features for the decisions[1]. However, this does not impact the results shown here, as every model and training strategy was measured with the same standard and the improvement is demonstrated regardless of the finesse of the activation maps.

[1] One possible algorithm to explore would be GradCAM++, which can deal better with multiple objects in an image.

List of Figures

List of Tables

List of Algorithms

Bibliography

[1] F. Psarommatis, G. May, P.A. Dreyfus, and D. Kiritsis. Zero defect manufacturing: state-of-the-art review, shortcomings and future directions in research. *International Journal of Production Research*, 2019. doi: 10.1080/00207543.2019.1605228. URL https://doi.org/10.1080/00207543.2019.1605228.

[2] H. Ahuett-Garza and T. Kurfess. A brief discussion on the trends of habilitating technologies for industry 4.0 and smart manufacturing. *Manufacturing Letters*, 2018. doi: https://doi.org/10.1016/j.mfglet.2018.02.011. URL www.elsevier.com/locate/mfglet.

[3] Christoph Molnar, Giuseppe Casalicchio, and Bernd Bischl. *Interpretable Machine Learning – A Brief History, State-of-the-Art and Challenges*, pages 417–431. Springer, 01 2020. ISBN 978-3-030-65964-6. doi: 10.1007/978-3-030-65965-3_28.

[4] Eu Wern Teh, Mrigank Rochan, and Yang Wang. Attention networks for weakly supervised object localization. In *BMVC*, 2016.

[5] Ramprasaath R. Selvaraju, Michael Cogswell, Abhishek Das, Ramakrishna Vedantam, Devi Parikh, and Dhruv Batra. Grad-CAM: Visual explanations from deep networks via gradient-based localization. *International Journal of Computer Vision*, 128(2):336–359, oct 2019. doi: 10.1007/s11263-019-01228-7. URL https://doi.org/10.1007%2Fs11263-019-01228-7.

[6] Md Zahangir Alom, Tarek M. Taha, Christopher Yakopcic, Stefan Westberg, Paheding Sidike, Mst Shamima Nasrin, Brian C Van Esesn, Abdul A S. Awwal, and Vijayan K. Asari. The history began from alexnet: A comprehensive survey on deep learning approaches, 2018. URL https://arxiv.org/abs/1803.01164.

[7] Frank Emmert-Streib, Zhen Yang, Han Feng, Shailesh Tripathi, and Matthias Dehmer. An introductory review of deep learning for prediction models with big data. *Frontiers in Artificial Intelligence*, 3:4, 02 2020. doi: 10.3389/frai.2020.00004.

[8] Warren Mcculloch and Walter Pitts. A logical calculus of ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 5:127–147, 1943.

[9] F. Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6):386–408, 1958. ISSN

0033-295X. doi: 10.1037/h0042519. URL http://dx.doi.org/10.1037/h0042519.

[10] Marvin Minsky and Seymour Papert. *Perceptrons: An Introduction to Computational Geometry*. MIT Press, Cambridge, MA, USA, 1969.

[11] David H. Ackley, Geoffrey E. Hinton, and Terrence J. Sejnowski. A learning algorithm for Boltzmann machines. *Cognitive Science*, 9:147–169, 1985.

[12] Kunihiko Fukushima. Neocognitron: A hierarchical neural network capable of visual pattern recognition. *Neural Networks*, 1(2):119–130, 1988. URL http://dblp.uni-trier.de/db/journals/nn/nn1.html#Fukushima88.

[13] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998. doi: 10.1109/5.726791.

[14] Geoffrey Hinton, Simon Osindero, and Yee-Whye Teh. A fast learning algorithm for deep belief nets. *Neural Computation*, 18(7):1527–1554, 2006.

[15] G E Hinton and R R Salakhutdinov. Reducing the dimensionality of data with neural networks. *Science*, 313(5786):504–507, July 2006. doi: 10.1126/science.1127647. URL http://www.ncbi.nlm.nih.gov/sites/entrez?db=pubmed&uid=16873662&cmd=showdetailview&indexed=google.

[16] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. 25, 2012. URL https://proceedings.neurips.cc/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf.

[17] F. Rosenblatt. The perceptron, a perceiving and recognizing automaton. Cornell Aeronautical Laboratory, 1957. URL https://blogs.umass.edu/brain-wars/files/2016/03/rosenblatt-1957.pdf.

[18] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016. http://www.deeplearningbook.org.

[19] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. 3 2015. doi: 1502.03167. URL https://arxiv.org/pdf/1502.03167.pdf.

[20] Zhe Li, Boqing Gong, and Tianbao Yang. Improved dropout for shallow and deep learning, 2016. URL https://arxiv.org/abs/1602.02220.

[21] Maximilian Riesenhuber and Tomaso Poggio. Hierarchical models of object recognition in cortex. 11 1999.

[22] Vinod Nair and Geoffrey E. Hinton. Rectified linear units improve restricted boltzmann machines. *null*, 2010. doi: null.

[23] Dan Hendrycks and Kevin Gimpel. Gaussian error linear units (gelus). *arXiv: Learning*, 2016. doi: null.

[24] Eli Stevens, Luca Antiga, and Thomas Viehmann. *Deep Learning with PyTorch*. Manning, 2020. URL https://pytorch.org/assets/deep-learning/Deep-Learning-with-PyTorch.pdf.

[25] Dae-Young Kang, Hieu Duong, and Jung-Chul Park. Application of deep learning in dentistry and implantology. *The Korean Academy of Oral and Maxillofacial Implantology*, 24:148–181, 09 2020. doi: 10.32542/implantology.202015.

[26] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998. doi: 10.1109/5.726791.

[27] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition, 2014. URL https://arxiv.org/abs/1409.1556.

[28] Y. Bengio, P. Simard, and P. Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks*, 5(2):157–166, 1994. doi: 10.1109/72.279181.

[29] Sunitha Basodi, Chunyan Ji, Haiping Zhang, and Yi Pan. Gradient amplification: An efficient way to train deep neural networks. *Big Data Mining and Analytics*, 3 (3):196–207, 2020. doi: 10.26599/BDMA.2020.9020004.

[30] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. pages 770–778, 06 2016. doi: 10.1109/CVPR.2016.90.

[31] Mingxing Tan and Quoc V. Le. Efficientnet: Rethinking model scaling for convolutional neural networks. 2019. doi: 10.48550/ARXIV.1905.11946. URL https://arxiv.org/abs/1905.11946.

[32] Mingxing Tan and Quoc V. Le. Efficientnetv2: Smaller models and faster training. 2021. doi: 10.48550/ARXIV.2104.00298. URL https://arxiv.org/abs/2104.00298.

[33] Zhuang Liu, Hanzi Mao, Chao-Yuan Wu, Christoph Feichtenhofer, Trevor Darrell, and Saining Xie. A convnet for the 2020s, 2022. URL https://arxiv.org/abs/2201.03545.

[34] Dzmitry Bahdanau, Kyunghyun Cho, and Y. Bengio. Neural machine translation by jointly learning to align and translate. *ArXiv*, 1409, 09 2014.

[35] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Kaiser Lukasz, and Illia Polosukhin. Attention is all you need. *31st Conference on Neural Information Processing Systems (NIPS 2017), Long Beach, CA, USA*, 12 2017. doi: 1706.03762v5.

[36] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *null*, 2019. doi: null.

[37] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 38–45, Online, October 2020. Association for Computational Linguistics. URL https://www.aclweb.org/anthology/2020.emnlp-demos.6.

[38] Jesse Vig. A multiscale visualization of attention in the transformer model. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, pages 37–42, Florence, Italy, July 2019. Association for Computational Linguistics. doi: 10.18653/v1/P19-3007. URL https://www.aclweb.org/anthology/P19-3007.

[39] Nicolas Carion, Francisco Massa, Gabriel Synnaeve, Nicolas Usunier, Alexander Kirillov, and Sergey Zagoruyko. End-to-end object detection with transformers. 2020. doi: 10.1007/978-3-030-58452-8_13. URL https://arxiv.org/abs/2005.12872.

[40] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C. Zitnick. Microsoft coco: Common objects in context. 05 2014.

[41] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale. 10 2020. doi: 2010.11929v2.

[42] Jonas Gehring, Michael Auli, David Grangier, Denis Yarats, and Yann N. Dauphin. Convolutional sequence to sequence learning. *ICML*, 2017. doi: 1705.03122. URL https://arxiv.org/abs/1705.03122.

[43] Ze Liu, Yutong Lin, Yue Cao, Han Hu, Yixuan Wei, Zheng Zhang, Stephen Lin, and Baining Guo. Swin transformer: Hierarchical vision transformer using shifted windows. 03 2021. doi: 2103.14030v2. URL https://arxiv.org/abs/2103.14030.

[44] Karl B. Rundman. *METAL CASTING: Quickest and Least Expensive Route to a Near Shape Product*. Michigan Tech. University, 01 2005. URL https://www.refcoat.com/pdf/book-on-metal-casting.pdf. Reference Book for MY4130, Dept. of Materials Science and Engineering Michigan Tech. University.

[45] Li Deng. The mnist database of handwritten digit images for machine learning research. *IEEE Signal Processing Magazine*, 29(6):141–142, 2012.

[46] Shachar Kaufman, Saharon Rosset, and Claudia Perlich. Leakage in data mining: Formulation, detection, and avoidance. *Proceedings of the 17th ACM SIGKDD*

international conference on Knowledge discovery and data mining, pages 556–563, 08 2011. doi: 10.1145/2020408.2020496.

[47] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

[48] Benyamin Ghojogh and Mark Crowley. The theory behind overfitting, cross validation, regularization, bagging, and boosting: Tutorial. 5 2019. doi: 1905.12787v1.

[49] Xue Ying. An overview of overfitting and its solutions. *J. Phys.: Conf. Ser. 1168 022022*, 2019. doi: 10.1088/1742-6596/1168/2/022022.

[50] Sebastien C. Wong, Adam Gatt, and Victor Stamatescu. Understanding data augmentation for classification: when to warp? 11 2016. doi: 1609.08764.

[51] Diederik P. Kingma and Jimmy Lei Ba. Adam: A method for stochastic optimization. 30 2017. doi: 1412.6980v9. URL https://arxiv.org/pdf/1412.6980.pdf.

[52] David W. Jacobs. Lecture notes on gradient descent. URL http://www.cs.umd.edu/~djacobs/CMSC426/GradientDescent.pdf.

[53] Sebastian Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016. URL https://arxiv.org/pdf/1609.04747.pdf.

[54] Tom Schaul, Sixin Zhang, and Yann LeCunn. No more pesky learning rates. *Proceedings of the 30 th International Conference on Machine Learning, Atlanta, Georgia, USA, 2013. JMLR: W&CP volume 28.* URL http://yann.lecun.com/exdb/publis/pdf/schaul-icml-13.pdf.

[55] Herbert Robbins and Sutton Monro. A stochastic approximation method. *The Annals of Mathematical Statistics, 22(3):400–407*, 1951. URL http://www.columbia.edu/~ww2040/8100F16/RM51.pdf.

[56] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research, 12:2121–2159*, 2011. URL https://www.jmlr.org/papers/volume12/duchi11a/duchi11a.pdf.

[57] Matthew D. Zeiler. Adadelta: An adaptive learning rate method. 2012. doi: arXiv:1212.5701. URL https://arxiv.org/pdf/1212.5701.pdf.

[58] Ashik Mohammed Sali, Harish Thampy, Immanuel S Vadakedam, and Sunitha S Pillai. Waste classification using convolutional neural network on edge devices. *International Journal of Innovative Science and Research Technology*, 5, 11 2020. URL https://ijisrt.com/assets/upload/files/IJISRT20NOV654.pdf.

[59] C. J. Van Rijsbergen. *Information Retrieval*. Butterworth-Heinemann, 2nd edition, 1979.

[60] C Molnar. *Interpretable Machine Learning - A guide for making black box models explainable*. Leanpub, 2019.

[61] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. "why should I trust you?": Explaining the predictions of any classifier. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, August 13-17, 2016*, pages 1135–1144, 2016.

[62] Chuang Gan, Naiyan Wang, Yi Yang, Dit-Yan Yeung, and Alexander Hauptmann. Devnet: A deep event network for multimedia event detection and evidence recounting. pages 2568–2577, 06 2015. doi: 10.1109/CVPR.2015.7298872. URL https://www.cv-foundation.org/openaccess/content_cvpr_2015/papers/Gan_DevNet_A_Deep_2015_CVPR_paper.pdf.

[63] Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. Deep inside convolutional networks: Visualising image classification models and saliency maps, 2013. URL https://arxiv.org/abs/1312.6034.

[64] Jost Tobias Springenberg, Alexey Dosovitskiy, Thomas Brox, and Martin Riedmiller. Striving for simplicity: The all convolutional net, 2014. URL https://arxiv.org/abs/1412.6806.

[65] Matthew D Zeiler and Rob Fergus. Visualizing and understanding convolutional networks, 2013. URL https://arxiv.org/abs/1311.2901.

[66] Bolei Zhou, Aditya Khosla, Agata Lapedriza, Aude Oliva, and Antonio Torralba. Learning deep features for discriminative localization, 2015. URL https://arxiv.org/abs/1512.04150.

[67] Maithra Raghu, Thomas Unterthiner, Simon Kornblith, Chiyuan Zhang, and Alexey Dosovitskiy. Do vision transformers see like convolutional neural networks?, 2021. URL https://arxiv.org/abs/2108.08810.

[68] Mathilde Caron, Hugo Touvron, Ishan Misra, Hervé Jégou, Julien Mairal, Piotr Bojanowski, and Armand Joulin. Emerging properties in self-supervised vision transformers, 2021. URL https://arxiv.org/abs/2104.14294.

Metric Comparison: Frozen vs Partially Frozen vs Unfrozen

All of the hyperparameters were kept consistent throughout all of the training strategies and models. For all of them, six epochs were used, with the same learning rate, weight decay, optimizer and loss function. The only thing that changes between the different training strategies are the amount of parameters frozen.

The fully frozen models are shown here with two snowflakes (❄❄) to the right side of the name of the model. The partially frozen models have only one snowflake (❄). The fully unfrozen model, which corresponds to the proposed method here, does not have any snowflakes.

Training

	Training - $\mathcal{T}$				
	Accuracy (%)	Precision (%)	Recall (%)	F1 Score (%)	Loss (%)
AlexNet ❄❄	70.02	71.54	70.02	70.25	0.5223000
AlexNet ❄	97.16	97.10	97.16	97.10	0.0018365
AlexNet	97.87	97.81	97.87	97.84	0.3308000

Table 1: Training metrics for AlexNet using fully frozen ❄❄, partially frozen ❄ and unfrozen pretrained weights.

	Training - $\mathcal{T}$				
	Accuracy (%)	Precision (%)	Recall (%)	F1 Score (%)	Loss (%)
ResNet ❄❄	73.68	75.69	73.68	74.03	0.4180000
ResNet ❄	98.63	98.61	98.63	98.62	0.0009694
ResNet	98.76	98.63	98.76	98.69	0.0004860

Table 2: Training metrics for ResNet using fully frozen ❄❄, partially frozen ❄ and unfrozen pretrained weights.

	Training - $\mathcal{T}$				
	Accuracy (%)	Precision (%)	Recall (%)	F1 Score (%)	Loss (%)
EfficientNet ❄❄	72.52	75.51	72.52	72.85	0.4986000
EfficientNet ❄	97.99	97.95	97.99	97.97	0.0023332
EfficientNet	98.23	98.21	98.23	98.22	0.0012940

Table 3: Training metrics for EfficientNet using fully frozen ❄❄, partially frozen ❄ and unfrozen pretrained weights.

	Training - $\mathcal{T}$				
	Accuracy (%)	Precision (%)	Recall (%)	F1 Score (%)	Loss (%)
ConvNeXt ❄❄	73.55	80.54	73.55	73.81	0.4085000
ConvNeXt ❄	98.80	98.76	98.80	98.78	0.0004571
ConvNeXt	98.98	98.94	98.98	98.96	0.0001970

Table 4: Training metrics for ConvNeXt using fully frozen ❄❄, partially frozen ❄ and unfrozen pretrained weights.

	Training - $\mathcal{T}$				
	Accuracy (%)	Precision (%)	Recall (%)	F1 Score (%)	Loss (%)
ViT ❄❄	83.62	86.10	83.62	84.24	0.2476000
ViT ❄	99.21	99.13	99.21	99.17	0.0000855
ViT	98.97	98.90	98.97	98.93	0.0002120

Table 5: Training metrics for ViT using fully frozen ❄❄, partially frozen ❄ and unfrozen pretrained weights.

	Training - $\mathcal{T}$				
	Accuracy (%)	Precision (%)	Recall (%)	F1 Score (%)	Loss (%)
Swin ❄❄	76.71	78.70	76.71	77.13	0.4249000
Swin ❄	98.63	98.57	98.63	98.60	0.0074219
Swin	98.60	98.57	98.60	98.58	0.0001920

Table 6: Training metrics for Swin Transformer using fully frozen ❄❄, partially frozen ❄ and unfrozen pretrained weights.

	Training - $\mathcal{T}$				
	Accuracy (%)	Precision (%)	Recall (%)	F1 Score (%)	Loss (%)
DINO ❄❄	99.06	99.02	99.06	99.04	0.0003808
DINO ❄	99.11	99.06	99.11	99.09	0.0000017
DINO	99.07	99.01	99.07	99.04	0.0051910

Table 7: Training metrics for DINO using fully frozen ❄❄, partially frozen ❄ and unfrozen pretrained weights.

Validating

	Validating - $\mathcal{V}$				
	Accuracy (%)	Precision (%)	Recall (%)	F1 Score (%)	Loss (%)
AlexNet ❄❄	77.98	80.41	77.98	78.62	0.4569000
AlexNet ❄	98.40	98.27	98.40	98.33	0.0103500
AlexNet	98.17	97.62	98.17	97.86	0.0083800

Table 8: Validating metrics for AlexNet using fully frozen ❄❄, partially frozen ❄ and unfrozen pretrained weights.

	Validating - $\mathcal{V}$				
	Accuracy (%)	Precision (%)	Recall (%)	F1 Score (%)	Loss (%)
ResNet ❄❄	75.31	79.44	75.31	76.03	0.4583000
ResNet ❄	99.00	99.10	99.00	99.05	0.0114200
ResNet	99.13	98.98	99.13	99.05	0.0097900

Table 9: Validating metrics for ResNet using fully frozen ❄❄, partially frozen ❄ and unfrozen pretrained weights.

	Validating - $\mathcal{V}$				
	Accuracy (%)	Precision (%)	Recall (%)	F1 Score (%)	Loss (%)
EfficientNet ❄❄	79.42	80.10	79.42	79.69	0.4957000
EfficientNet ❄	99.33	99.29	99.33	99.31	0.0090118
EfficientNet	99.51	99.32	99.51	99.41	0.0085900

Table 10: Validating metrics for EfficientNet using fully frozen ❄❄, partially frozen ❄ and unfrozen pretrained weights.

	Validating - $\mathcal{V}$				
	Accuracy (%)	Precision (%)	Recall (%)	F1 Score (%)	Loss (%)
ConvNeXt ❄❄	76.35	84.38	76.35	77.27	0.4195000
ConvNeXt ❄	99.21	98.96	99.21	99.08	0.0070286
ConvNeXt	99.19	99.32	99.19	99.25	0.0045500

Table 11: Validating metrics for ConvNeXt using fully frozen ❄❄, partially frozen ❄ and unfrozen pretrained weights.

	Validating - $\mathcal{V}$				
	Accuracy (%)	Precision (%)	Recall (%)	F1 Score (%)	Loss (%)
ViT ❄❄	85.38	87.30	85.38	86.03	0.3327000
ViT ❄	99.19	99.07	99.19	99.13	0.0072708
ViT	99.39	99.18	99.39	99.28	0.0030700

Table 12: Validating metrics for ViT using fully frozen ❄❄, partially frozen ❄ and unfrozen pretrained weights.

	Validating - $\mathcal{V}$				
	Accuracy (%)	Precision (%)	Recall (%)	F1 Score (%)	Loss (%)
Swin ❄❄	83.64	85.79	83.64	84.32	0.3850000
Swin ❄	99.38	99.19	99.38	99.28	0.0088647
Swin	99.21	99.05	99.21	99.13	0.0109500

Table 13: Validating metrics for Swin Transformer using fully frozen ❄❄, partially frozen ❄ and unfrozen pretrained weights.

	Validating - $\mathcal{V}$				
	Accuracy (%)	Precision (%)	Recall (%)	F1 Score (%)	Loss (%)
DINO ❄❄	98.63	98.50	98.63	98.56	0.0071965
DINO ❄	98.02	98.51	98.02	98.25	0.0055777
DINO	99.21	99.89	99.21	99.08	0.0066000

Table 14: Validating metrics for DINO using fully frozen ❄❄, partially frozen ❄ and unfrozen pretrained weights.

Testing

	Testing - $\mathcal{R}$				
	Accuracy (%)	Precision (%)	Recall (%)	F1 Score (%)	Loss (%)
AlexNet ❄❄	89.17	88.70	89.17	88.22	0.4423965
AlexNet ❄	99.80	99.59	99.81	99.67	0.0112395
AlexNet	99.80	99.58	99.80	99.67	0.0073700

Table 15: Testing metrics for AlexNet using fully frozen ❄❄, partially frozen ❄ and unfrozen pretrained weights.

	Testing - $\mathcal{R}$				
	Accuracy (%)	Precision (%)	Recall (%)	F1 Score (%)	Loss (%)
ResNet ❄❄	91.00	91.64	91.00	90.47	0.4223104
ResNet ❄	99.70	99.40	99.70	99.52	0.0119450
ResNet	99.80	99.59	99.80	99.67	0.0104700

Table 16: Testing metrics for ResNet using fully frozen ❄❄, partially frozen ❄ and unfrozen pretrained weights.

	Testing - $\mathcal{R}$				
	Accuracy (%)	Precision (%)	Recall (%)	F1 Score (%)	Loss (%)
EfficientNet ❄❄	79.82	78.21	79.82	73.74	0.5610808
EfficientNet ❄	99.81	99.56	99.81	99.67	0.0098640
EfficientNet	99.80	99.59	99.80	99.67	0.0105800

Table 17: Testing metrics for EfficientNet using fully frozen ❄❄, partially frozen ❄ and unfrozen pretrained weights.

	Testing - $\mathcal{R}$				
	Accuracy (%)	Precision (%)	Recall (%)	F1 Score (%)	Loss (%)
ConvNeXt ❄❄	88.08	85.50	88.08	83.73	0.4445168
ConvNeXt ❄	99.81	99.56	99.81	99.67	0.0076704
ConvNeXt	99.89	99.81	99.89	99.84	0.0063800

Table 18: Testing metrics for ConvNeXt using fully frozen ❄❄, partially frozen ❄ and unfrozen pretrained weights.

	Testing - $\mathcal{R}$				
	Accuracy (%)	Precision (%)	Recall (%)	F1 Score (%)	Loss (%)
ViT ❄❄	92.57	89.80	92.57	90.11	0.3293909
ViT ❄	99.81	99.56	99.81	99.67	0.0094821
ViT	99.80	99.59	99.80	99.67	0.0082000

Table 19: Testing metrics for ViT using fully frozen ❄❄, partially frozen ❄ and unfrozen pretrained weights.

	Testing - $\mathcal{R}$				
	Accuracy (%)	Precision (%)	Recall (%)	F1 Score (%)	Loss (%)
Swin ❄❄	92.36	90.27	92.36	90.20	0.3950927
Swin ❄	99.81	99.56	99.81	99.67	0.0126511
Swin	99.80	99.59	99.80	99.67	0.0125000

Table 20: Testing metrics for Swin Transformer using fully frozen ❄❄, partially frozen ❄ and unfrozen pretrained weights.

	Testing - $\mathcal{R}$				
	Accuracy (%)	Precision (%)	Recall (%)	F1 Score (%)	Loss (%)
DINO ❄❄	99.81	99.59	99.81	99.67	0.0078960
DINO ❄	99.81	99.56	99.81	99.67	0.0100199
DINO	99.70	99.40	99.70	99.52	0.0135200

Table 21: Testing metrics for DINO using fully frozen ❄❄, partially frozen ❄ and unfrozen pretrained weights.