Example title

Masterarbeit

von

Johannes Baßler, B.Sc.

am

Institut für Industrielle Informationstechnik

Zeitraum:	01. 01. 2010 – 30. 06. 2010
Hauptreferent:	Prof. Dr.-Ing. Michael Heizmann
Betreuer:	M.Sc. Max Mustermann
	Dr.-Ing. Hans Maier[1]
	Dipl.-Ing. John Doe[1]

[1] XYZ AG - Karlsruhe

Erklärung

Ich versichere hiermit, dass ich meine Masterarbeit selbstständig und unter Beachtung der Regeln zur Sicherung guter wissenschaftlicher Praxis im Karlsruher Institut für Technologie (KIT) in der aktuellen Fassung angefertigt habe.
Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt und wörtlich oder inhaltlich übernommene Stellen als solche kenntlich gemacht.

Karlsruhe, den 30. Juni 2010

Kurzfassung

Hier könnte eine deutsche Kurzfassung kommen.

Abstract

Here comes an english abstract. (This is optional: If not needed, please delete this environment)

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Motivation

Hier kann viel aus Papern übernommen werden. Einige Industriebeispiele wären ganz cool. –> TODO Beispiele, die zitierfähig sind, finden.
Wird aber wahrscheinlich eines der letzten Kapitel, die angegangen werden wird. Aktuell keine hohe Priorität (11. Oktober)

1.1 Anomaliedetektion in der Industrie

Hier wird die Relevanz von Anomaliedetektion in der Industrie erläutert. Wie bereits erklärt –> Industriebeispiele aus der Ëchten Welt-> Muen etc. fragen für Tips.

1.2 Unüberwachte Anomaliedetektion

Hier wird ausgeführt warum überwachte Methoden an ihre Grenzen stoßen und warum unüberwachte Methoden sinnvoll sind. Auch hierfür werden noch Quellen gesucht.

1.3 Laufzeitoptimierung und Ressourcenbeschränktheit

Erster Abschnitt aus efficientad paper kann hier gut zitiert werden.
Real-world anomaly detection applications frequently put constraints on the computational requirements of a method. There are cases where detecting an anomaly too late can cause substantial economic damage, such as metal objects in a crop field entering the interior of a combine harvester. In other cases, even human health is at risk, for example, if a limb of a machine operator approaches a blade. Furthermore, industrial settings commonly involve strict runtime limits caused by high production rates [4]. Not adhering to these limits would decrease the production rate of the respective application and thus its economic via-bility. It is therefore essential to pay attention to the compu-tational and economic cost of anomaly detection methods to keep them suitable for real-world applications. Dann wird ausgeführt, warum eine Laufzeitoptimierung sinnvoll ist und warum es auch heute noch sinnvoll sein kann oder nicht

ansders möglich ist, als auf teure GPUs zu verzichten. Beispiele für edge devices –> mobile Roboter, oder Drohnen, die autonom agieren sollen, aber nur begrenzte Ressourcen haben und vor allem keine GPUs.
Können wir hier den Bogen spannen zu Implementierung auf FPGAs? –> Muen fragen und recherchieren.

Kapitel 2

Grundlagen

Hier wird kurz aufgezählt was erläutert wird. Mehr nicht. Es sollen hier nur Elemente erläutert werden, die für mindestens zwei der Methoden relevant sind. k-center greedy (PatchCore), Autoencoder (efficientad) oder Backpropagation (simplenet) also zB nicht.

2.1 Datensatz: MVTecAD[4]

Das „MVTec Anomaly Detection Dataset“ (MVTec AD) ist ein am 6. Januar 2021 veröffentlichter, umfangreicher Datensatz für die Anomalieerkennung in Bildern. Dieser bildet die Evaluationsgrundlage aller hier in dieser Arbeit vorkommendenen Entwicklungen und Methoden. Treibende Kraft hinter der Entwicklung des Datensatzes ist die MVTec Software GmbH - ein deutsches Unternehmen, das sich auf industrielle Bildverarbeitung spezialisiert hat. Dieser Datensatz ist entwickelt worden, um einen internationalen Benchmark zu schaffen, der die Entwicklung von Algorithmen für die Unüberwachte Anomalieerkennung in Bildern vorantreibt und Methoden quantitativ vergleichbar macht. Betrachtet man die Anzahl an Veröffentlichungen fällt auf, dass seit der Veröffentlichung des Datensatzes immer mehr Methoden auf diesem Datensatz evaluiert werden. Waren es 2020 22 Veröffentlichungen stieg die Anzahl streng monoton bis auf bereits 80 im laufenden Jahr 2023 (Stand 11.10.2023).[8]
Der Datensatz besteht aus insgesamt 5354 Bildern, die 15 verschiedene Klassen von Objekten enthalten. Die folgende Tabelle gibt einen Überblick über die Klassen und die Anzahl an Bildern pro Klasse. Diese 15 Klassen lassen sich in zehn Objektklassen und fünf Texturklassen unterteilen. Die fünf ersten Klassen der Tabelle (Carpet, Grid, Leather, Tile, Wood) sind Texturen bzw. Strukturen, die weiteren zehn Klassen (Bottle, Cable, Capsule, Hazelnut, Metal nut, Pill, Screw, Toothbrush, Transistor, Zipper) sind Objekte.
Wie bereits erwähnt, handelt es sich um einen Datensatz für Unüberwachte Anomaliedetekion, was sich daran erkennen lässt, dass in den Trainingsdaten ausschließlich Bilder ohne Defekte (nominal) enthalten sind. Die Testdaten sind in zwei Klassen unterteilt: „Good“ und „Defective“. Zwar liegen in den meisten Fällen mehrere mögliche Defektklassen vor, die auch eindeutig gelabelt werden, allerdings handelt es sich um einen Binärklassifikationsdatensatz, was bedeutet, dass das Erkennen der Art des Defektes keine Zielstellung ist. Durch diese Vielzahl an Defekten kann aber eine gewisse Generalisierungsfähigkeit getestet werden.
Alle anomalen Testbilder habem eine pixelweise Annotation, die die Defekte markiert. Diese

Tabelle 2.1: Übersicht über Anzahl an Bildern, Auflösung und Defektgruppen des Datensatzes

Kategorie	#Training	#Test (nominal)	#Test (anomal)	#Defekt Gruppen	#Defekt Regionen	Seitenlänge
Teppich	280	28	89	5	97	1024
Gitter	264	21	57	5	170	1024
Leder	245	32	92	5	99	1024
Fließen	230	33	84	5	86	840
Holz	247	19	60	5	168	1024
Flasche	209	20	63	3	68	900
Kabel	224	58	92	8	151	1024
Kapsel	219	23	109	5	114	1000
Haßelnuss	391	40	70	4	136	1024
Metallmutter	220	22	93	4	132	700
Pille	267	26	141	7	245	800
Schraube	320	41	119	5	135	1024
Zahnbürste	60	12	30	1	66	1024
Transistor	213	60	40	4	44	1024
Reißverschluss	240	32	119	7	177	1024
Total	3629	467	1258	73	1888	-

Annotationen sind in Form von Binärbildern gegeben, wobei die Pixel der Defekte mit 1 und die Pixel der nominalen Regionen mit 0 markiert sind. In dieser Arbeit liegt zwar der Fokus auf der Instanzklassifizierungsgenauigkeit und nicht auf der Segmentierung. Dennoch ist diese qualitativ hochwertige Annotation ein wohl wesentlicher Grund für die Beliebtheit des Datensatzes. Nicht nur lässt sich damit schlicht die Segmentierungsgenauigkeit testen, sondern der Vergleich einer von einer Methode zu einem Testbild festgestellte Anomaliekarte mit der Annotation ist ein wertvolles Werkzeug, um die Funktionsweise einer Methode zu verstehen und etwaige Schwachstellen zu identifizieren.
Anzumerken ist, dass es sich bei den vorliegenden Defekten um keine logischen Defekte handelt, sonder um lokale, strukturelle Abweichungen von der Norm. Die grundsätzliche Gestalt ist auch im Falle eines anomalen Bildes erhalten. Möchte man zum Beispiel ein Anomaliedetekionsverfahren entwickeln, das zu lang geratene oder stark gekrümmte Schrauben erkennt, so ist der Datensatz nicht optimal geeignet. Hierzu sei auf den neueren Datensatz aus dem Hause MVTec, MvTec LOCO AD verwiesen. [14] Beisple für nominale und anomale Bilder aus dem Datensatz finden sich in Abbildung 2.1.

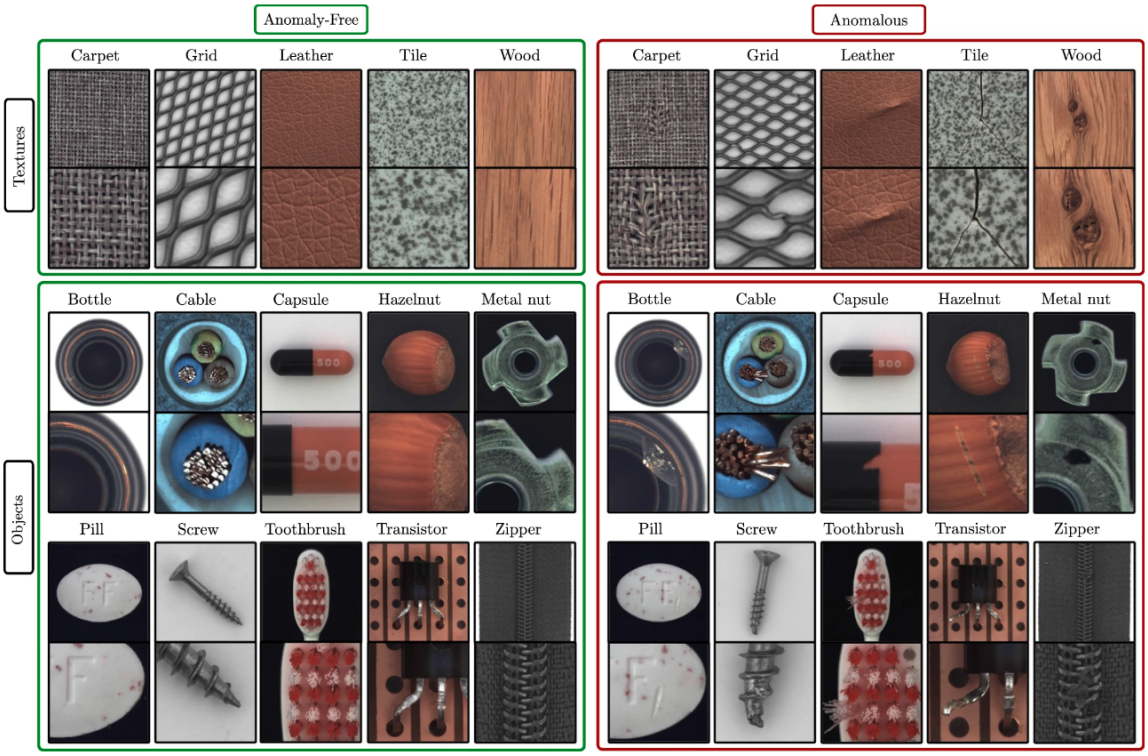


Abbildung 2.1: Beispiele für nominale und anomale Bilder aus dem MVTec AD Datensatz

2.2 Eigener Datensatz (Granulat)

Details und Beispiele zu eigenem Datensatz.

2.3 AUROC[11]

In dieser Arbeit, genauso wie in beinahe allen anderen Arbeiten im Bereich der binären Anomalieerkennung, wird die „Area Under the Receiver Operating Characteristic Curve“ (AUROC) als Leistungsmaß verwendet. Die AUROC ist ein Maß für die Fähigkeit eines binären Klassifikators, zwischen zwei Klassen zu unterscheiden. Nachfolgend wird schrittweise zum Begriff des AUROC hingeführt.

2.3.1 Logistische Regression

Die logistische Regression ist eine Art verallgemeinertes lineares Modell, das üblicherweise für binäre Klassifizierungsprobleme verwendet wird. Bei der logistischen Regression besteht das Ziel darin, die Wahrscheinlichkeit eines binären Ergebnisses (z. B. nominal oder anomal) auf der

Grundlage einer Reihe von Eingangsmerkmalen vorherzusagen. Das logistische Regressionsmodell verwendet eine logistische Funktion („Sigmoidfunktion“), um die Eingabemerkmale auf die vorhergesagte Wahrscheinlichkeit abzubilden.

Die logistische Funktion ist definiert als:

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

wobei z eine lineare Kombination aus den Eingangsmerkmalen und ihren zugehörigen Gewichten ist:

$$z = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_p x_p$$

Dabei ist β_0 der Bias-Term und $\beta_1, \beta_2, \ldots, \beta_p$ sind die Koeffizienten für die Eingangsmerkmale $x_1, x_2, \ldots bzw. x_p$.

Das logistische Regressionsmodell wird trainiert, indem eine Verlustfunktion minimiert wird, die die Differenz zwischen den vorhergesagten Wahrscheinlichkeiten und den wahren binären Kennzeichnungen misst. Eine gängige Verlustfunktion für logistische Regression ist der binäre Kreuzentropie (‚Cross-Entropy“), die wie folgt definiert ist:

$$\mathcal{L}(\beta) = -\frac{1}{n} \sum_{i=1}^{n} y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i)$$

wobei β die Modellparameter (d.h. den Achsenabschnitt und die Koeffizienten) darstellt, n die Anzahl der Trainingsbeispiele, y_i das wahre binäre Label für das i-te Beispiel und $\hat{y}_i$ die vorhergesagte Wahrscheinlichkeit für das i-te Beispiel ist.

Das logistische Regressionsmodell kann mithilfe des Gradientenabstiegs trainiert werden, bei dem die Modellparameter iterativ in Richtung des negativen Gradienten der Verlustfunktion aktualisiert werden. Der Gradient der Verlustfunktion in Bezug auf die Modellparameter kann mithilfe der Kettenregel der Infinitesimalrechnung berechnet werden. [7] (Kapitel 4) [18]

2.3.2 Konfusionsmatrix

Die Konfusionsmatrix ist eine Tabelle, die die Leistung eines binären Klassifikationsmodells zusammenfasst. Sie besteht aus vier Einträgen: wahr-positive (TP), falsch-positive (FP), wahr-negative (TN) und falsch-negative (FN). TP und TN stehen für die Anzahl der richtig klassifizierten positiven bzw. negativen Beispiele, während FP und FN für die Anzahl der falsch klassifizierten positiven bzw. negativen Beispiele stehen.

Hier stehen die Zeilen für die vorhergesagten Kennzeichnungen und die Spalten für die wahren Kennzeichnungen. Die Einträge in der Diagonale stehen für die richtigen Vorhersagen, während

Tabelle 2.2: Konfusionsmatrix

	Tatsächlich Positive	**Tatsächlich Negative**
Prädizierte Positive	Wahre Positive (TP)	Falsche Positive (FP)
Prädizierte Negative	Falsche Negative (FN)	Wahre Negative (TN)

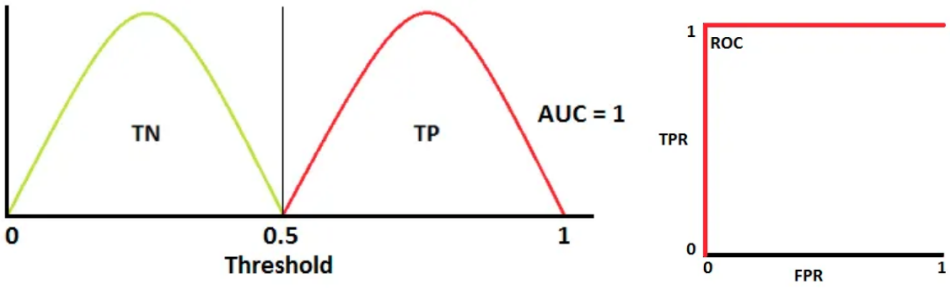
die Einträge außerhalb der Diagonale die falschen Vorhersagen darstellen. Die „True Positive Rate“(TPR), die auch als Sensitivität oder Recall bezeichnet wird, ist definiert als TP / (TP + FN), d. h. der Anteil der positiven Beispiele, die richtig klassifiziert wurden. Die „False Positive Rate“ (FPR) ist definiert als FP / (FP + TN), d. h. der Anteil der negativen Beispiele, die falsch klassifiziert werden. Die Konfusionsmatrix bietet eine Möglichkeit, die Leistung eines binären Klassifikationsmodells im Hinblick auf seine Fähigkeit, positive und negative Beispiele richtig zu klassifizieren, zu bewerten. Sie kann zur Berechnung verschiedener Leistungsmetriken verwendet werden, wie z. B. Genauigkeit, Präzision, Wiedererkennung, F1-Score und die Fläche unter der „Receiver Operating Characteristic“ (ROC)-Kurve (AUC), die im nächsten Abschnitt erläutert wird.

2.3.3 ROC-Kurve

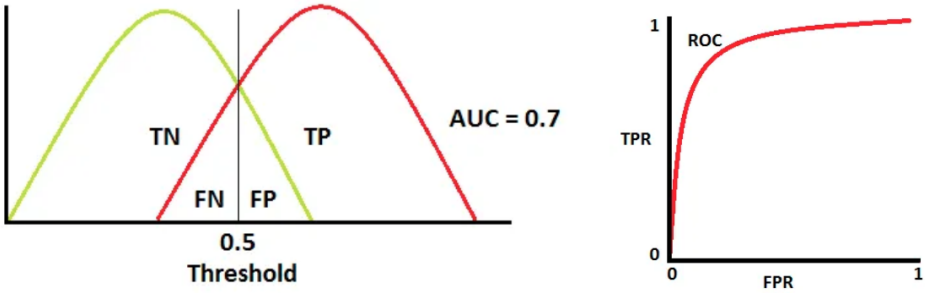
Die Receiver-Operating-Characteristic-Kurve (ROC-Kurve) ist eine grafische Darstellung der Leistung eines binären Klassifikationsmodells, wenn der Schwellenwert variiert wird. In der ROC-Kurve wird die Rate der richtig positiven Beispiele (TPR) gegen die Rate der falsch positiven Beispiele (FPR) für verschiedene Schwellenwerte aufgetragen. Die TPR ist der Anteil der positiven Beispiele, die richtig klassifiziert werden, während die FPR der Anteil der negativen Beispiele ist, die falsch klassifiziert werden.
Berechnet man die Fläche unter der ROC-Kurve erhält man das **„Area Under the Curve“ (AUC)** Maß. Es ist eines der aussagekräftigsten Maße, die es für eine binäre Klassifikation wie die Anomalieerkennung gibt. Die AUC ist ein Wert zwischen 0 und 1, wobei 1 für eine perfekte Klassifikation steht, $0{,}5$ für eine zufällige Klassifikation und 0 für eine gänzlich falsche Klassifikation steht. Nachfolgend sind mögliche Ausbabeverteilungen einer logistischen Regression, markiert mit der tatsächlichen Klassenzugehörigkeit, skizziert. Diese Veranschaulichen den Zusammenhang zwischen der Ausgabe einer logistischen Regression, der ROC-Kurve und dem AUROC-Maß.

2.4 Residuale Netzwerke

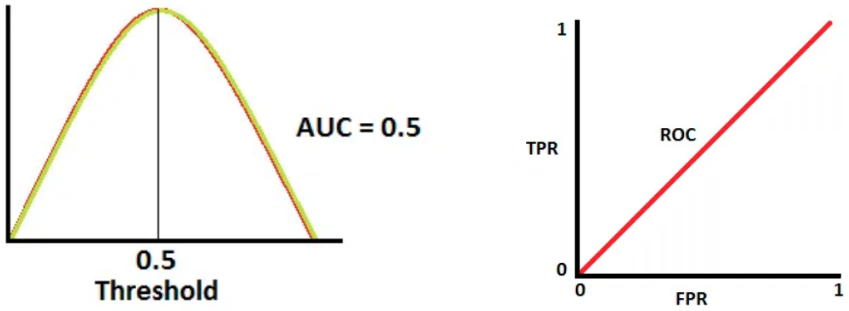
In diesem Abschnitt werden die Grundlagen von Residual Networks (textbf„ResNets“) erläutert. Diese spielen für die Merkmalsextraktoren („Feature Extractors“) in den im Haupteil der Arbeit (TODO –> Link) eine wichtige Rolle. Der Schwerpunkt hierbei liegt auf der grundlegenden Idee, der Rolle, die ResNets historisch in der Entwicklung von Deep Learning gespielt haben und



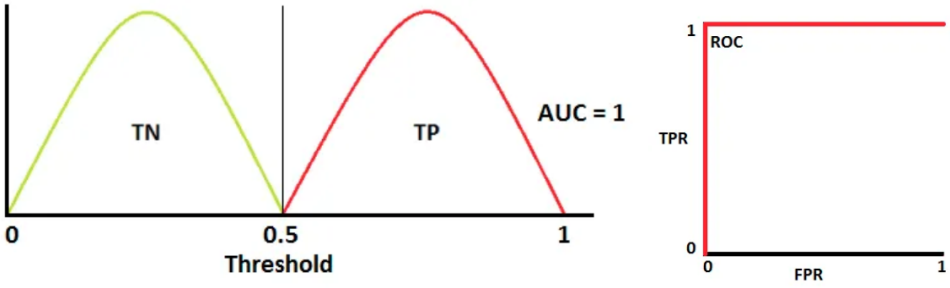
(a) $\mathbf{AUC} = 100\%$: Die beiden Verteilungen der Klassen sind vollständig getrennt. Eine perfekte Klassifikation ist möglich.



(b) $\mathbf{AUC} = 70\%$: Eine Überlappung der Verteilungen lässt mit keinem Schwellwert eine fehlerfreie Klassifikation zu. Ein Großteil der Beispiele kann aber richtig klassifiziert werden.



(c) $\mathbf{AUC} = 50\%$: Die Verteilungen der Klassen überlappen sich vollständig. Eine sinnvolle Klassifikation ist somit nicht möglich. Eine Zuordnung würde zufällig geschehen.



(d) $\mathbf{AUC} = 0\%$: Dieser Spezialfall lässt keine korrekte Klassifikation zu, obwohl die Klassen eindeutig getrennt sind. Dies liegt daran, dass die Verteilung der positiven Klasse vollständig links von der Verteilung der negativen Klasse liegt. Durch eine geeignete Abbildung kann dieses Problem gelöst werden.

Abbildung 2.2: Veranschaulichung verschiedener Verteilungen positiver und negativer Klassen und den dazugehörenden ROC-Kurven und AUC-Werten.

vor allem den Aspekten, die für die Anwendung in dieser Arbeit relevant sind. Für detailierte Informationen zu ResNets wird auf das Paper von Kaiming He et al. [16] und die zahlreichen Erläuterungen in der Literatur verwiesen.

2.4.1 Hintergrund & Idee hinter „ResNets“

Tiefe neuronale Netze (Deep Neural Networks, DNNs) eignen sich hervorragend für das Lernen hierarchischer Darstellungen aus Daten, aber sie stehen vor Herausforderungen, wenn sie „tiefer“ werden.„Tiefe“ bezeichnet in diesem Zusammenhang die Anzahl an Schichten, die sequentiell durchlaufen werden, um das Endergebnis bzw. die Ausgabe zu erhalten. Tiefere Netze können komplexere Merkmale in Daten erfassen, was für Aufgaben wie die Bildklassifizierung, bei der Objekte und Muster komplizierte Details aufweisen können, entscheidend ist. Einer der großen Herausforderungen bei tiefen Netzen ist das Problem der „verschwindenden Gradienten“ (*engl.* **Vanishing Gradients**). Diese Gradienten sind entscheidend für das Training eines Neuronalen Netzes, insofern, als dass das Optimierungsverfahren des Gradientenabstiegs die Grundlage auch moderner Optimierer wie „Adam“ (TODO –> Ref) ist. Dieses Problem lässt sich anschaulich dadurch erklären, dass frühe Gradienten, was sich leicht mithilfe der Kettenregel zeigen lässt, eine Multiplikation von allen nachfolgenden (im Sinne der Infernenzrichtung - „forward pass“) Gradienten darstellt. Betrachtet man nun ein sehr tiefes Netz, das heißt, viele Gradienten, die miteinander multipliziert werden und den wahrscheinliche Fall von Gradienten, die kleiner als 1 sind, so wird schnell klar, dass frühe Schichten einen sehr kleinen Gradienten haben können. Dies macht es schwierig, die Gewichte der frühen Schichten zu aktualisieren, was den Lernprozess behindert. Dabei ist es vor allem die Tiefe, die Neuronalen Netzen das Generalisieren von komplexen Zusammenhängen ermöglicht („Deep Learning “)
Residuale Netze, allgemein bekannt als ResNets und im Folgenden auch so bezeichnet, wurden von Kaiming He et al. (TODO –> Ref) in ihrem Paper von 2015 vorgestellt und bieten eine einfache und dennoch effektive Methode an, wie dieses Problem angegangen werden kann. Die Grundidee besteht darin, Verknüpfungen zwischen den Schichten einzuführen, die es dem Netz ermöglichen, eine oder mehrere Schichten zu „überspringen“. Anstatt die gewünschte Abbildung also direkt zu lernen, lernen ResNets die residuale Abbildung, was der Differenz zwischen Ein- und Ausgabe entspricht. Die Eingabe wird über sogenannte „Shortcut (Skip) Connections“, also einfach die Identitätsabbildung, vom Eingang zur residualen Ausgabe weitergeleitet, um dann durch Summation wieder miteinander verknüpft zu werden. Das Problem der verschwindenden Gradienten wird dadurch entschärft, dass die Gradienten direkt in frühere Schichten zurückfließen können. Anschaulich lässt sich das durch die Tatsache erklären, dass die Identitätsabbildung einen konstanten Gradienten von 1 besitzt. Weil sich die Summenbildung am Ausgang eines nachfolgend noch im Detail besprochenen Residual-Blocks auch bei der Gradientenbildung als Summation widerspiegelt, ist der Gradient einer jeden Schicht nicht mehr das Produkt, sondern vielmehr die Summe aller nachfolgenden Gradienten. (Optional TODO: Formel) Das zugrundeliegende Paper ist eines der meistzitierten Paper im Bereich des Deep Learning und hat die Entwicklung von Deep Learning maßgeblich beeinflusst.

2.4.2 Residual Block & Architektur

Der Grundbaustein eines ResNet ist der Residualblock. Er besteht aus zwei Hauptpfaden: dem Identitätspfad (der Abkürzungsverbindung) und dem Residualpfad (dem Hauptfaltungspfad). Mathematisch wird die Ausgabe eines Residualblocks wie folgt berechnet:

$$\text{Output} = F(\text{Input}) + \text{Input}$$

wo F die Residualabbildung ist, die durch die Hauptfaltungsschichten gelernt wird. Nachfolgende Abbildung zeigt eine vereinfachte Darstellung eines „Building Blocks“ bzw. Residual Block.

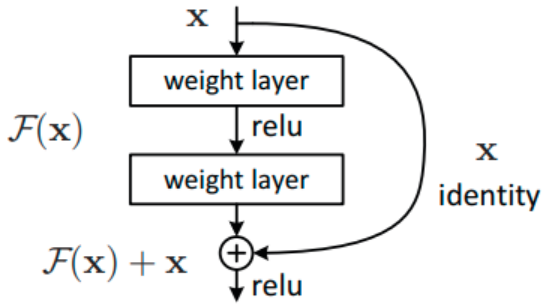


Abbildung 2.3: Residual Block (TODO –> Ref)

Ein ResNet besteht aus mehreren Residualblöcken, die sequentiell durchlaufen werden. Es ist also eine „Feed-Forward“-Architektur, weil die Daten zur Inferenz nur in eine Richtung durch das Netz fließen. Durch das „Stapeln “ können dann sehr tiefe Netze erstellt werden, die sich dennoch aufgrund der oben beschriebenen Eigenschaften gut optimieren bzw. trainieren lassen. Es existieren zahlreiche verschiedene Varianten von ResNets, die sich in der Art und Anzahl ihrer Residual Blöcke unterscheiden. Nachfolgende Tabelle gibt einen Überblick über die drei, in dieser Arbeit vor allem verwendeten Varianten: ResNet18, ResNet34 und WideResNet50

Tabelle 2.3: Vergleich verschiedener ResNet Varianten (TODO –> Ref)

Netzwerk	Tiefe	# Parameter	Top-1 Fehlerrate[1]	Inferenz auf RBP4[2]
ResNet18	18	$11{,}7 \cdot 10^6$	30,24%	0, 82s
ResNet34	34	$21{,}8 \cdot 10^6$	26,70%	1, 45s
WideResNet50	50	$68{,}9 \cdot 10^6$	22,53%	3, 00s

Zu erkennen ist eindeutig, dass die Anzahl der Parameter und die Inferenzzeit mit der Tiefe des Netzes steigt. Gleichzeitig lässt sich aber auch eine Verbesserung der Top-1 Fehlerrate erkennen, je tiefer bzw. mächtiger das Netz ist.
Einer der Hauptvorteile von ResNets ist die Fähigkeit, Merkmale pyramidenförmig durch das

[1] in ILSVRC (TODO –> Ref)
[2] Raspberry Pi 4B 8GB. Betrachtet wurde die Laufzeit von einem Bild der Auflösung 224x224 und 3 (Farb-)Kanälen

Netz zu verbreiten. Das bedeutet, dass das Netz Merkmale auf verschiedenen Abstraktionsebenen erfassen kann, von Low-Level-Merkmalen wie Kanten und Ecken bis zu High-Level-Merkmalen wie Objektteilen und semantischen Konzepten. Mit zunehmender Tiefe wird also die Auflösung der „Feature Maps“ (TODO –> Ref) reduziert, während die Anzahl der Kanäle und die Komplexität der zugrundeliegenden Merkmale zunimmt. Dargestellt ist das in folgender Abbildung:

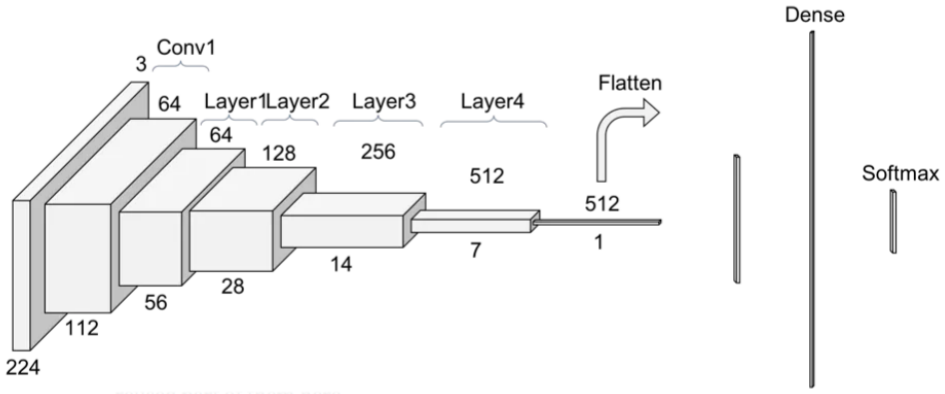


Abbildung 2.4: Pyramidale Merkmalsverteilung in ResNets (TODO –> Ref)

Die Quader in oben stehendem Netz stehen symbolhaft für die Auflösung der Feature Maps. Während die oben stehenden Zahlen die Anzahl der Kanäle repräsentieren, stehen die unten aufgeführten Zahlen für die räumliche Auflösung. Letztere hängt proportional von der Auflösung des Eingangsbildes ab und gilt für alle drei Modelle. Die Anzahl an Kanälen ist konstant für alle Auflösungen und für ResNet18 und ResNet34. Für das WideResNet50 gilt im Wensentlichen der gleiche Aufbau, allerdings sind die Kanäle „geweit“ gegenüber den anderen beiden Architekturen, was sich an der Vervierfachung der Anzahl an Kanäle durch Verwendung eines komplexeren Residual Blocks („Bottleneck“[37])zeigt. ResNet18 und ResNet34 unterscheiden sich ausschließlich in der Anzahl der Residual Blöcke bzw. der Tiefe des Netzes.
Alle der drei hier vorgestellten Architekturen lassen sich in fünf Faltungsschichten („Conv1“, „Layer1“, „Layer2“, „Layer3“, „Layer4“) unterteilen. Dem schließt sich eine „Average Pooling“-Schicht an, die die Auflösung der Feature Maps auf 1 reduizert („Flatten“). Dieser 1D-Vektor wird schließlich durch eine „Fully Connected“-Schicht („Dense“) auf die Anzahl der Klassen (1000) abgebildet. Schließlich wird durch eine „Softmax“-Aktivierungsfunktion Pseodo-Wahrscheinlichkeiten erzeugt, die den Axiomen von Kolmogorov entsprechen und somit als Auftrittswahrscheinlichkeiten interpretiert werden können.
In dieser Arbeit werden die ResNets als Feature-Extraktoren verwendet, weshalb die letzten beiden Schichten, also „Flatten“ und „Dense“ nicht verwendet werden. Die Begriffe „Layer1 - Layer4“ werden im Folgenden in dem hier beschriebenen Kontext verwendet.

2.4.3 ResNets as Feature Extractor für Unüberwachte Lernverfahren

Zusätzlich zu ihrem Erfolg bei der überwachten Bildklassifizierung haben ResNets Anwendungen als leistungsstarke Merkmalsextraktoren bei Unüberwachten Klassifizierungsaufgaben wie der Anomalieerkennung gefunden. In unüberwachten Szenarien wie der Anomalieerkennung stehen oft keine markierten (gelabelten) Daten zur Verfügung, wie bereits in 1.2 beschrieben, um ein Modell explizit bzw. überwacht zu trainieren. Stattdessen verlässt man sich auf das Lernen von Darstellungen normaler Daten und identifiziert dann Abweichungen als Anomalien. ResNets können mit ihrer Fähigkeit, umfangreiche und hierarchische Merkmale zu erfassen, dazu verwendet werden, sinnvolle Merkmale aus den Daten zu extrahieren. Es kann mithilfe dieser Netze eine kompaktere und aussagekräftigere Darstellung der Daten erzeugt werden, die die Grundlage sind, um auch komplexere Anomalien zu erkennen. Allerdings ist festzuhalten, dass die extrahierten Feature in keiner Weise optimiert sind, um Anomalieklassifikationen zu ermöglichen, sondern um auf dem ImageNet-Datensatz Bilder richtig einzuordnen. Vor allem in späteren Schichten ist also zu erwarten, dass die Merkmale einene „Bias “ hin zu ImageNet aufweisen, der sich negativ auf die Anomalieklassifikation auswirken kann.[26] Hierzu werden die Gewichte offizieller Implementierungen von ResNets verwendet, womit das eigentliche Vortraining entfällt und Reproduzierbarkeit, Konsistenz und Vergleichbarkeit der Ergebnisse gewährleistet wird.
Alle hier vorgestellten Methoden verwenden ResNets als Feature-Extraktoren, wenn auch im Ansatz„EffientAD“ (4) nicht während der Inferenz. Gezeigt, dass eine solche Feature Extraktion im Kontext der Unüberwachten Anomalieerkennung sinnvoll ist, wurde erstmals in der Veröffentlichung„Deep Nearest Neighbor Anomly Detection“ von Bergman et al.[3] mit der Methode „DN2“ im Februar 2020. Im nachfolgenden Abschnitt wird die darauf aufbauende Methode „SPADE“, vom gleichen Team entwickelt, vorgestellt.

2.5 Quantisierung von CNNs

In dieser Arbeit wird an vielen Stellen zu sehen sein, dass die Inferenz eines Convolutional Neural Networks (CNN) von enormer Bedeutung ist. Die Inferenzzeit dieser Netze zu verkürzen, ohne dabei die Genauigkeit zu stark zu beeinträchtigen, ist ein Hauptmotiv vieler Adaptionen, die in dieser Arbeit diskutiert werden. Eine sehr effektives Werkzeug ist das sogenannte Quantisieren dieser Netze. Im Folgenden wird das Prinzip dahinter erläutert und gezeigt, wie sich die Inferenzzeit von CNNs durch Quantisierung reduzieren lässt.

2.5.1 Grundlagen der Quantisierung

Im Wesentlichen ist die Quantisierung nichts anderes, als die Reduzierung der numerischen Präzision. In Fällen, in denen diese Präzision nicht von großer Bedeutung ist, kann durch eine Quantisierung eine enorme Komprimierung der Information erreicht werden. So wird man,

gefragt nach der Uhrzeit, nicht mit der Uhrzeit in Sekunden antworten, selbst wenn jemandem diese Information zur Verfügung stünde, sondern großzügig auf 5 Minuten runden. Eine noch genauere Uhrzeit bringt dem Fragenden kein Mehrwert und würde die Antwort nur schwerer verständlich machen. Das ist, sehr vereinfacht formuliert, das Prinzip der Quantisierung. Information wird bewusst reduziert, um die Komplexität zu reduzieren.

Abbildungsfunktion

Es wird zunächst eine Abbildungsfunktion benötigt, die Gleitkommazahlen, konkret „floating point"-Zahlen in $32bit$ aufgelöst, auf ganze Zahlen, also „Integer" abbildet. Diese Integer haben eine Länge von $8bit$ ($n = 8$), also $1Byte$. Es sind aber auch andere Zahlenwerte hier denkbar. In der Regel ist eine solche Abbildungsfunktion Q eine lineare Transformation der Form:

$$\mathcal{Q}(x) = \left\lfloor \frac{x}{S} + Z + \frac{1}{2} \right\rfloor$$

x bezeichnet dabei den Eingangswert als Gleitkommazahl, S die Skalierung, Z den Nullpunkt und $\mathcal{Q}(x) = x_q \in \mathbb{N}$ schließlich der korresponierende qunatisierte Wert zu $x \in \mathbb{R}$. Die Quantisierungsparameter S und Z müssen passend gewählt werden, um eine sinnvolle Abbildung zu erzielen, wie im nächsten Abschnitt zu sehen sein wird.
Um aus einer quantisierierten Ganzzahl wieder eine korresponierende Gleitkommazahl zu erhalten, wird die Umkehrung der Abbildungsfunktion verwendet:

$$\tilde{x} = \mathcal{Q}^{-1}(x_q) = S \cdot (x_q - Z)$$

Es sei angemerkt, dass im Allgemeinen $x \neq \tilde{r}$ gilt. Die Differenz $|r - \tilde{r}|$ wird als Quantisierungsfehler bezeichnet. Der Quantisierungsfehler ist maximal $q/2$ mit $q = \frac{S}{2^n}$ als die Größe einer Quantisierungsstufe.

Skalierung und Nullpunkt

Der Skalierungsparameter S entspricht dem Verhältnis zwischen dem Wertebereich der Gleitkommazahlen und dem Wertebereich der Ganzzahlen. Ersteres lässt sich beispielsweise experimentell mithilfe eines Trainingsdatensatzes aus dem Minimum und Maximum der Eingangswerte x, also den Gleitkommazahlen, bestimmen. Der Wertebereich der Ganzzahlen ist durch die Anzahl der Bits n, die zur Darstellung verwendet werden, gegeben. Es ergibt sich damit folgende Formel:

$$S = \frac{x_{max} - x_{min}}{2^n - 1}$$

Der Nullpunkt Z ist der Wert, der auf die Gleitkommazahlen abgebildet wird, die dem Wert 0 entsprechen. Er stellt sicher, dass die Abbildungsfunktion $\mathcal{Q}$ die Zahl 0 auch auf die quantisierte Ganzzahl 0 abbildet. Der Nullpunkt lässt sich wie folgt berechnen:

$$Z = -\frac{x_{min}}{S}$$

Die Parameter x_{min} und x_{max} werden während des sogenannten Kalibrierung bestimmt. Es wird ein Kalibrierungsdatensatz verwendet, der repräsentativ für die Daten ist, die später quantisiert werden sollen. Es ist dabei nicht zwingend notwendig, dass x_{max} und x_{min} den tatsächlichen Extremwerten entsprechen. Häufig kann es sinnvoll sein, Quantile zu verwenden, um einen negativen Einfluss von Ausreißern zu vermeiden.

2.5.2 Statische Post-Training Quantisierung

Die statische Post-Training Quantisierung ist eine der einfachsten Methoden, um ein CNN zu quantisieren und bietet sich insbesondere dann an, wenn das Verhalten eines bereits trainierten Netzes möglichst gut durch die eine quantisierte Variante dessen approximiert werden soll. Da dies im Falle dieser Arbeit der Fall ist, wie in späteren Kapiteln zu sehen sein wird, wird diese Methode hier vorgestellt.
Die Ausgangssituation ist ein bereits trainiertes Netz ϕ. Zunächst wird die Struktur des Netzes betrachtet und bestimmte Operationen zusammengefasst. So werden zum Beispiel die sequentiellen Elemeneten „Convolution“, „Batch Normalization“ und „ReLU“ in einen zusammenhängenden „Quantisierungsblock“ gefasst. Im quantisierten Netz ϕ_q können diese dann durch die Berechnung eines einzelnen Kernels berechnet werden, was die Inferenzzeit ebenfalls reduziert. Da die Eingabedaten, als auch die gewünschte Ausgabe des Netzes ϕ in aller Regel Gleitkommazahlen sind, müssen außerdem an Eingabe und Ausgabe des Netzes Quantisierungsschichten angefügt werden.
Während der sich anschließenden Kalibrierung werden dann repräsentative Daten aus einem Kalibrationsdatensatz verwendet, um das Verhalten des nicht quantisierten Netzes ϕ zu simulieren und zu analysieren. An den Ein-und Ausgängen werden die statistischen Informationen gesammelt, die zur Bestimmung der Quantisierungsparameter S und Z benötigt werden. Man beachte, dass jeder Quantisierungsblock seine eigenen Parameter S und Z besitzt. Dieser Kalibrierungsdatensatz sollte möglichst repräsentativ sein, zum Beispiel aus der gleiche Domäne stammen. Es reichen aber in aller Regel wenige hundert Beispiele aus, um eine gute Approximation zu erreichen.
Es kann dann die tatsächliche Quantisierung stattfinden. Sämtliche Gewichte werden dann mithilfe des beschrieben Verfahren quantisiert. Schließlich erhält man so ein quantisiertes Netz ϕ_q, das die gleiche Struktur wie das ursprüngliche Netz ϕ besitzt, aber nur noch Ganzzahlen verwendet.

2.6 SPADE

Abgeschlossen: 30.10. v1
Die Methode **SPADE** ist ein wichtiger Meilenstein in der Unüberwachten Anomalieerkennung. Zahlreiche erfolgreiche Methoden bauen auf SPADE auf und übernehmen wichtige Elemente

und Konzepte. Das Paper wurde am 5. Mai 2020 von Niv Cohen und Yedid Hoshen von der Hebräischen Universität von Jerusalem veröffentlicht. Im Folgenden wird SPADE vorgestellt.

2.6.1 Funktionsweise

Feature Extraktion

Wie bereits in 2.4.3 beschrieben, werden ResNets erfolgreich als Feature Extraktoren verwendet. Der Name „SPADE“ ist dabei ein Akronym für „Semantic Pyramid Anomaly Detection“. Betrachtet man 2.4 wird klar warum von einer „Pyramide“ gesprochen werden kann: Es werden die Feature Maps der einzelnen Schichten, konkret der Schichten „Layer1, Layer2 und Layer3“ als Feature unverändert verwendet. Die Hierarchieebnene, die für die Anomalieerkennung verwendet werden, werden im Folgenden als Menge j bezeichnet. Hier ist $j = \{1, 2, 3\}$, da die Schichten Layer1, Layer2 und Layer3 verwendet werden. Dabei wird auf einen Einbettungsprozess, wie bei anderen Methoden üblich, verzichtet. Durch die mit der Tiefe des Netzes abnehmende Auflösung der Feature Maps, entsteht so eine „Pyramidale“ Struktur, die eben die Grundlage für die Namensgebung ist. Zusätzlich wird der 1D-Vektor, der durch die „Flatten“-Schicht erzeugt wird, als Feature verwendet, um instanzweise, also für jedes Bild als Ganzes, zu entscheiden, ob eine Anomalie vorliegt, wie im nächsten Abschnitt beschrieben. Bezeichnen wir das ResNet als Feature-Extraktor mit ϕ, so lassen sich die Feature f_i eines gegebenen Bildes x_i wie folgt beschreiben:

$$f_i = \phi(x_i)$$

Die Feature werden unverändert weiter verwendet, das heißt, es wird kein expliziter Einbettungsprozess, wie bei anderen Methoden üblich, durchgeführt. In der Trainings- bzw. Initialisierungsphase werden die Feature f_i aller Trainingsbilder $x_i \in \mathcal{X}_{train}$, welche alle nominal sind, extrahiert und gespeichert. Diese Menge wird im Folgenden mit F_{train} bezeichnet.

Instanzklassifizierung

Die Instanzklassifizierung ist der zweite Schritt von SPADE, der sich der Feature Exktraktion anschließt. Es wird dabei für jedes Bild x_i aus der Menge an Testbilder $\mathcal{X}_{test}$ entschieden, ob es sich um eine Anomalie handelt oder nicht. Dies geschieht ausschließlich anhand des 1D-Vektors, der durch die „Flatten“-Schicht erzeugt wird. Hierzu wird für ein gegebenes Testbild x_i, die K Nächsten Nachbarn in F_{train} gesucht. Diese Menge an K Vektoren wird fortan als $N_k(f_y)$ bezeichent. Es wird die euklidische Distanz auf die folgende Art und Weise bestimmt:

$$d(x_i) = \frac{1}{K} \sum_{f \in N_k(f_{x_i})} \left|\left| f - f_{x_i} \right|\right|^2$$

Die Distanz $d(x_i)$ enstpricht dann dem, dem Bild zugewiesenen Anomaliegrad und kann dann mit einem Schwellenwert τ verglichen. Ist die Distanz kleiner als der Schwellenwert, so wird

das Bild als nominal klassifiziert, ansonsten als Anomalie. Zur Evaluation wird, wie in dieser Arbeit üblich und in (TODO –> Ref) beschrieben, der AUROC-Wert verwendet.

Segmentierung

Nachdem eine Instanzklassifiziergung ergeben hat, dass es sich um ein anomales Bild handelt, besteht die nächste Aufgabe darin, Anomalien auf dem Bild zu lokalisieren. Dieser Schritt wird übersprungen, wenn das Bild als nominal klassifiziert wurde.
Eine naive Methode, die Anomalie zu lokalisieren, wäre es, die Differenz zwischen dem anomalen Bild und dem ausgerichtete Bild des Nächsten Nachbarn zu berechnen. Dort, wo die Differenz am größten ist, wird die Anomalie vermutet. Die Ausrichtung geschieht derart, dass die Objekte in den zu vergleichenden Bildern sich möglichst vollständig überlappen. Allerdings hat diese Methode einige Schwachstellen. Zum einen kann die Ausrichtung des nächsten Nachbarn so, dass das zu untersuchende Objekt im Bild an der gleichen Stelle ist, fehlschlagen. Auch kann insbesondere bei eienem kleinen Datensatz und Objekten, die komplexe Variationen aufweisen möglicherweise kein passender Nachbar gefunden werden, was zu falsch positiven (anomalen) Klassifizierungen führen kann. Außerdem könnte die Berechnung der Differenz sehr empfindlich auf die verwendete Berechnungsmethode sein.
Um diese Probleme zu überwinden, wird eine Übereinstimmungsmethode präsentiert, die mit vielen „Bildern“ arbeitet („multi-image correspondence method“). Diese Bilder entsprechen den Feature Maps der Schichten Layer1, Layer2 und Layer3, also gewissermaßen den Quader in 2.4. Nun gilt es zu jeder Pixelposition $(h, w) \in \{1, ..., h^*\} \times \{1, ..., w^*\}$ mit h^* und q^* als Kantenlängen des Eingangsbildes,
die korresponierenden Feature $F(x_i, h, w)$ zu bestimmen. Auf jedes Bild $x_i \in \mathcal{X}_{train}$ angewandt und ergibt sich dann die „Gallerie“ zu $G = \{F(x_1, h, w) \cup F(x_2, h, w).. \cup F(x_K, h, w)\}$.
Der Anomaliegrad eines Pixels (h, w) ist dann gegeben durch die durchschnittliche Distanz zwischen dem Feature $F(y_i, h, w)$ des anomalen Bildes y_i und den κ nächsten Featuren aus der Gallerie G. Es ergibt sich also für einen Pixel (h, w) im Testbild y_i folgende Formel:

$$d(y_i, h, w) = \frac{1}{\kappa} \sum_{f \in N_\kappa(F(y_i,h,w))} ||f - F(y_i, h, w)||^2$$

Für einen gegebenen Schwellwert θ wird ein Pixel h, w als anomales Pixel klassifiziert, wenn $d(y_i, p) > \theta$ gilt. Dies ist dann der Fall, wenn wir kein nominales Feature in G finden können, welches dem Feature $F(y_i, p)$ ausreichend ähnlich ist.
Auvh hier gilt, dass ein pixelweiser AUROC aus $d(y_i, h, w)$ bestimmt werden kann, um die Methode zu evaluieren. Dadurch, dass die Gallerie G die Positionsinformation p marginalisiert, wird die Segmentierung robust gegenüber der Ausrichtung. Anschaulich gesprochen, wird also in jedem Bild an allen Positionen nach ähnlichen Features gesucht.

2.6.2 Ergebnisse und Diskussion

Wie bereits erwähnt, markiert diese Arbeit einen wichtigen Entwicklungsschritt in der Forschung zur (Unüberwachten –> TODO) Anomalieerkennung. Folgende Aspekte sollten in diesem Zusammenhang hervorgehoben werden:

- SPADE ist die erste ganzheitliche Methode, die auf ImageNet vortrainierte ResNets als Feature-Extraktoren verwendet. Dieser Ansatz hat sich als sehr erfolgreich erwiesen und wird von zahlreichen Methoden übernommen.
- Das Zusammenfassen aller extrahierten Feature aus den Testbildern zu einer Menge G löst das Problem der Ausrichtung und ermöglicht eine robuste Segmentierung. Auch diese Idee wird uns bei PatchCore (3) wiederbegegnen.
- Das Bestimmen des Anomaliegrades mithilfe einer kNN-Suche ist ein einfacher und effektiver Ansatz, der sich auch in modifizierter Form in vielen anderen Methoden wiederfindet.

Insbesondere in der pixelweisen Klassifikation von Anomalien hat sich SPADE als geeignet erwiesen. Auch in der Instanzklassifiziergung erreicht diese Methode auf dem damals noch jungen Datensatz MVTecAD gute, zum Zeitpunkt der Veröffentlichung der Methode, sogar die besten Ergebnisse.
Für das Ziel dieser Arbeit, welche den Fokus auf die Instanzklassifiziergung und die Laufzeitoptimierung legt, ist SPADE allerdings eher ungeeignet. Zum einen sind 85,5% erreichte Bildklassifizierungsgenauigkeit auf MVTecAD für die Instanzklassifizierung nicht mit neueren, zum Teil aber auf SPADE aufbauenden Methoden, konkurrenzfähig. Zum Anderen ist die notwendige kNN Suche, die für jedes Testbild durchgeführt werden muss, sehr rechenintensiv. Dieser Rechenaufwand skaliert linear mit der Anzahl an Bildern im Trainingsdatensatz und mit der Anzahl der Pixel pro Bild. Kann man auf eine Segmentierung verzichten, verkleinert sich der Rechenaufwand zwar deutlich, es muss aber dennoch eine kNN-Suche über alle Trainingsbilder für jedes Testbild durchgeführt werden, wodurch die Laufzeit wiederum von der Größe des Trainingsdatensatz abhängt, auch wenn die Auflösung der Bilder keinen Einfluss mehr hat. Den Anstoß und die Impulse, die durch diese Veröffentlichung gesetzt wurden, sind jedoch nicht zu unterschätzen.

2.7 PaDiM

Abgeschlossen: 31.10. v1
Die Methode **PaDiM** wurde am 17. November 2020 von Defard et al. (Universität Paris-Saclay) veröffentlicht. Es werden einige Aspekte von SPADE übernommen, vor allem aber Schwächen der Methode erfolgreich addressiert. Einer der wesentlichen Unterschiede zwischen PaDiM und SPADE ist, die Weise, wie der Anomaliegrad bestimmt wird. PaDiM markierte damit die beste Instanzklassifizierungsgenauigkeit auf dem Datensatz MVTecAD zum Zeitpunkt der

Veröffentlichung und hielt dies bis zum Erscheinen der Methode „PatchCore“.[8] Nachfolgend wird der Ansatz von PaDiM vorgestellt und diskutiert.

2.7.1 Funktionsweise

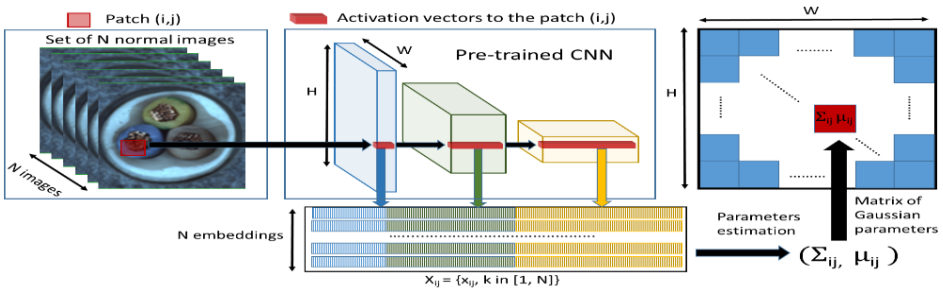


Abbildung 2.5: PaDiM: Funktionsweise (TODO –> Ref)

PaDiM kann in drei Schritte unterteilt werden:

1. Feature Extraktion und Einbettungsprozess mithilfe eines vortrainierten ResNets

2. Bestimmen der Gaußverteilungen durch Schätzen der Parameter μ und Σ für jede Position im Bild

3. Bestimmen des Anomaliegrades durch die Mahalanobis-Distanz

2.5 veranschaulicht den Prozess exklusive der Inferenz.

Feature Extraktion und Einbettungsprozess

Der Prozess des Erzeugens von Featuren ist bei PaDiM sehr ähnlich zu SPADE: Auch hier werden Feature Maps unterschiedlicher Auflösung und Hierarchieebene zu einem Featurevektor zusammengefasst, der mit einer Position bzw. einem Pixel des Eingangsbildes korresponiert. Die Feature Map, welche die höchste Auflösung besitzt, also die Feature Map der ersten zur Feature Extraktion ausgewählten Schicht, definiert die Auflösung der Anomaliesegmentierung. Nehmen wir an diese Feature Map habe eine Auflösung von $w^* \times h^*$, so korresponiert also zu jeder Position $(h, w) \in \{1, ..., h^*\} \times \{1, ..., w^*\}$ ein Featurevektor $x_{h,w}$. Dieser Vektor wird in Einklang mit der Veröffentlichung „Patch Embedding Vektor“ genannt.

Bestimmen der Gaußverteilungen

PaDiM modelliert für jede Position (h, w) eine multivariate Normalverteilung, die durch die Patch Embedding Vektoren aller Trainingsbilder an dieser Position definiert wird.
Zunächst wird folgerichtig die Menge an Patch Embedding Vektoren für eine Position (h, w) aus dem gesamten Trainingsdatensatz gebildet. Diese Menge $X_{h,w} = \{x_{h,w}^k | k \in \{1, ..., N\}\}$ aus den N nominalen Bildern im Trainingsdatensatz wird dann benutzt, um die Normalverteilung für die Position (h, w) zu bestimmen. Es wird also angenommen, $X_{h,w}$ würde von der multivariaten Gaußverteilungen $N(\mu_{h,w}, \Sigma_{h,w})$ erzeugt werden. Die Parameter $\mu_{h,w}$ und $\Sigma_{h,w}$ werden dann wie folgt geschätzt:

$$\mu_{h,w} = \frac{1}{N} \sum_{k=1}^{N} x_{h,w}^k$$

$$\Sigma_{h,w} = \frac{1}{N-1} \sum_{k=1}^{N} (x_{h,w}^k - \mu_{h,w})(x_{h,w}^k - \mu_{h,w})^T + \epsilon I$$

wobei der Regularisierungsterm ϵI hinzugefügt wird, um die Invertierbarkeit bzw. den vollen Rang der Kovarianzmatrix $\Sigma_{h,w}$ zu gewährleisten. Schließlich wird die multivariate Gaußverteilung $N(\mu_{h,w}, \Sigma_{h,w})$ für jede Position (h, w) im Bild definiert.

Bestimmen des Anomaliegrades

Um den Anomaliegrad zu bestimmen, wird die Mahalanobis-Distanz herangezogen [21]. Die Mahalanobis-Distanz ist eine Verallgemeinerung der euklidischen Distanz, die die Korrelation zwischen den Dimensionen der Daten berücksichtigt. In diesem Fall lässt sich die Mahalanobis-Distanz für die Position (h, w) und einem aus einem Testbild extrahierten Patch Embedding Vektor $y_{h,w}$ wie folgt berechnen:

$$M(y_{h,w}) = \sqrt{(y_{h,w} - \mu_{h,w})^T \Sigma_{h,w}^{-1} (y_{h,w} - \mu_{h,w})}$$

Damit kann dann eine Anomaliekarte M für ein Testbild y erzeugt werden:

$$M = (M(y_{h,w}))_{1<=w<=w^*, 1<=h<=h^*}$$

Hierbei deuten hohe Werte auf eine Anomalie hin, während niedrige Werte auf einen nominalen Bildbereich hinweisen. Durch Maximalwerbildung über die Anomaliekarte M kann dann ein Anomaliegrad s für ein Testbild y bestimmt werden:

$$s(y) = \max_{h,w} M(y_{h,w})$$

Basierend auf allen Anomaliegraden $s(y)$ für alle Testbilder $y \in \mathcal{X}_{test}$ kann dann der AUROC bestimmt werden.

2.7.2 Ergebnisse und Diskussion

Wie bereits zu Beginn dieses Kapitels erwähnt, erreicht PaDiM zum Zeitpunkt der Veröffentlichung die beste Instanzklassifizierungsgenauigkeit auf dem Datensatz MVTecAD mit 97,5%. Auch die Segmentierungsergebnisse sind mit 97,9% zum Veröffentlichungszeitpunkt State-of-the-Art.
Ein spannender Aspekt, der in diesem Veröffentlichung untersucht wird, ist das Reduzieren der Anzahl an Kanälen bzw. die Reduzierung der Dimensionalität der Featurevektoren:
Es wird gezeigt, dass im Falle eines ResNet18 als Backbone die Dimensionalität von ursprünglich 448 auf 200 oder sogar 100 reduziert werden kann, ohne dass die Klassifizierungsgenauigkeit signifikant sinkt. Dabei wurden die zu entfernenden Dimensionen zufällig ausgewählt und die Ergenisse über mehrere Durchläufe gemittelt. Im Falle aller 448 Dimensionen wird eine Genauigkeit von 97,1% erreicht, während bei 200 Dimensionen eine Genauigkeit von 97,0% und bei 100 Dimensionen eine Genauigkeit von 96,7% erreicht wird.
Dies ist vor allem deshalb eine sehr relevante Erkenntnis, da die Reduzierung der Dimensionalität einen ganz erhebliche Einfluss auf die Berechnungsgeschwindigkeit der Mahalanobis-Distanz hat. Die Komplexität in der Landau-Notation für die Berechnung der Mahalanobis-Distanz zwischen zwei Vektoren der Länge d ist $\mathcal{O}(d^3)$[7], wobei d die Dimensionalität der Featurevektoren ist, in diesem Beispiel also 448, 200 bzw. 100. Dass der Einbruch der Genauigkeit bei einer Reduzierung der Dimensionalität so gering ist, ist auf die Mahalanobis-Distanz zurückzuführen, die die Korrelation zwischen den Dimensionen der Daten berücksichtigt. Weil die einzelnen Dimensionen teilweise stark korreliert sind, kann die Dimensionalität reduziert werden, ohne dass die Genauigkeit signifikant sinkt. Dies ist ein Vorteil gegenüber der euklidischen Distanz, die die Dimensionen unabhängig voneinander betrachtet, aber auch deutlich weniger komplex und laufzeitkritisch ist, als die Mahalanobis-Distanz.
Ein Verbesserung gegenüber SPADE ist, dass die Instanzklassifiziergung auf Grundlage der Anomaliekarte M erfolgt und durch Maximalwerbildung über alle Positionen im Bild erzeugt wird. Diese Anomaliekarte wird mithilfe der Feature aus den ersten drei von vier Schichten ($j = \{1, 2, 3\}$) des ResNets erzeugt, die, wie in 2.4.3 beschrieben, nur einen eher geringen „Bias“ hin zu ImageNet aufweisen. SPADE wiederum nutzt den 1D-Vektor, der durch die „Flatten“-Schicht erzeugt wird, um die Instanzklassifizierung durchzuführen, was zwei Nachteile mit sich bringt: Es ist der Bias zur Bildklassifikation zu erwarten und durch die durch Mittelwertbildung erreichte Dimensionsreduktion können wichtige, zu einer lokalen Anomalie gehörende Detailinformationen verloren gehen. Bei PaDiM können selbst lokale Anomalien, die nur wenige Pixel groß sind, erkannt werden, was eine enorme Verbesserung darstellt und in allen im Haupteil dieser Arbeit vorgestellten Methoden übernommen wird.

2.8 Raspberry Pi 4B

2.8.1 Allgemeines

Der Raspberry Pi 4, der im Juni 2019 von der Raspberry Pi Foundation veröffentlicht wurde, ist ein kleiner, erschwinglicher und vielseitiger Einplatinencomputer.

Die zentrale Recheneinheit (CPU) des Raspberry Pi 4 ist eine 64-bit Quad-Core-ARM-Cortex-A72-CPU, die mit 1,8 GHz (ältere Versionen mit 1,5 GHz) taktet. Er ist in drei Speicherkonfigurationen erhältlich: 1 GB, 2 GB, 4 GB und 8 GB LPDDR4 RAM mit 3200 MHz. Die Integration eines Broadcom VideoCore VI-Grafikprozessors verbessert die Multimedia-Fähigkeiten und ermöglicht eine flüssige Videowiedergabe und 3D-Grafik-Rendering.

Ein bemerkenswertes Merkmal des Raspberry Pi 4 sind die Anschlussmöglichkeiten. Er verfügt über zwei USB 3.0-Ports und zwei USB 2.0-Ports, die den Anschluss verschiedener Peripheriegeräte ermöglichen. Dualband-Wi-Fi (2,4GHz und 5GHz) und Gigabit-Ethernet sorgen für eine zuverlässige Netzwerkverbindung. HDMI- und Audioausgänge unterstützen hochauflösende Bildschirme und Audiogeräte. So können beispielsweise zwei 4K-Displays angeschlossen werden.

Das Gerät ist mit mehreren Betriebssystemen kompatibel, darunter Raspberry Pi OS (früher Raspbian), Linux-Distributionen und sogar Windows 10, je nach Vorlieben und Anforderungen des Benutzers. In dieser Arbeit wurde Raspberry Pi OS in der 64-bit Version verwendet. Das auf Debian basierende Betriebssystem ist für die Hardware des Raspberry Pi optimiert und ist kompatibel mit allen notwendigen Softwarepaketen.

Die 40 GPIO-Pins des Raspberry Pi 4 sind eine vielseitige Hardwareschnittstelle, die zahlreiche Anwendungen in verschiedenen Bereichen ermöglicht. Die Anwendungen des Raspberry Pi 4 reichen von Bildungs- und Hobbyprojekten bis hin zu professionellen Unternehmungen. Er kann für Aufgaben wie Heimautomatisierung, Robotik, Webserver und Softwareentwicklung verwendet werden. Dank seines geringen Stromverbrauchs von etwa 2,7 W (Idle) bis maximal 6,4 W (Volllast, keine Peripherie)[13] eignet er sich für eingebettete Systeme und Internet-of-Things-Anwendungen (IoT).[12][33]

2.8.2 Ressourcenbeschränktheit

Die Rechenkapazität des Raspberry Pi 4 ist für viele Anwendungen ausreichend. Dennoch muss die Leistungsfähigkeit realisitsch eingeordnet werden: Während der Paspberry Pi 4 eine Rechenleistung von 13,5 GLFOPS (Gleitkommaoperationen pro Sekunde) erreicht, ist ein auf der gleichen Architektur (ARM) beruhender Apple M1 Prozessor aus dem Jahr 2020, der für einfache Consumer Tätigkeiten konzipiert ist, mit 154 GFLOPS mehr als 11 mal so schnell. [22] Auch muss erwähnt werden, dass auf die Verwendung von modernen GPUs für die Inferenz in dieser Arbeit verzichtet wird, wie in Kapitel 1.3 bereits beschrieben. Setzt man die Leistung

des Raspberry Pi 4 in Relation zu modernen GPUs, die viele Entwicklungen im Bereich *Deep Learning* überhaupt erst ermöglichten, so wird schnell klar, warum bei der Verwendung eines Raspberry Pi 4 von „Ressourcenbeschränktheit“ gesprochen werden kann. Auch wenn Rechenkapazität in FLOPS gemessen keine eindeutigen Schlüsse auf die Laufzeit eines konkreten Algorithmus zulässt, so kann trotzdem festgehalten werden, dass eine moderne GPU eine enorm höhere Rechenleistung besitzt. So erreichen moderne GPUs, wie Nvidia's Geforce RTX4090 Ti 82 600 GFLOPS.[25]

Um die Ressourcenbeschränktheit weiter zu verdeutlichen, sind in 2.4 die Laufzeiten von in 2.4 vorgestellten Modellen auf dem Raspberry Pi 4 B mit der Laufzeit auf einem AMD Ryzen R5 5600X und einer Nvidia GeForce RTX3060Ti verglichen. Die Laufzeiten wurden für ein Bild der Auflösung 224x224 und 3 (Farb-)Kanälen gemessen. Es handelt sich hierbei um Hardware der gehobenen Mittelklasse aus dem Jahr 2020, die auch für dieser Arbeit verwendet wurde, womit die Ergebnisse leicht selbst erzeugt werden konnten.
Es ist deutlich zu erkennen, dass die Laufzeit auf dem Raspberry Pi 4 B im Vergleich zu den anderen Plattformen um mehrere Größenordnungen höher ist. Die Laufzeit auf dem Raspberry Pi 4 B ist im Mittel und Vergleich zu der Laufzeit auf dem AMD Ryzen R5 5600X um den Faktor 61 und im Vergleich zur Nvidia GeForce RTX3060Ti um den Faktor 602 höher, wie in letzter Zeile der Tabelle 2.4 zu sehen. Anzufügen ist, dass dies nicht auf alle Szenarien zutrifft. Bei dem hier angebrachten Beispiel, der Laufzeit eines ResNets, ist eine GPU aufgrund der Parallelisierbarkeit von Faltungsschichten und der damit verbundenen hohen Anzahl an FLOPS deutlich im Vorteil. Wie im weiteren Verlauf dieser Arbeit zu sehen, ist dies aber ein durchaus representatives Beispiel, da die alle Modelle in dieser Arbeit zumindest teilweise auf eben jene Faltungsschichten (CNNs) zurückgreifen.

Tabelle 2.4: Laufzeiten der Modelle auf verschiedenen Plattformen

Netzwerk	**Raspberry Pi 4B**	**AMD Ryzen R5 5600X**	**Nvidia GeForce RTX3060Ti**
ResNet18	820,0 ms	11,68 ms	1,46 ms
ResNet34	1450,0 ms	20,00 ms	2,58 ms
WideResNet50	3000,0 ms	74,83 ms	4,40 ms
Faktor zu RPB4[3]	1 (Referenz)	61	602

[3] Raspberry Pi 4B

Kapitel 3

PatchCore [26]

3.1 Einleitung

Abgeschlossen: 01.11. v1

Die Methode **PatchCore** wurde erstmals am 15. Juni 2021 in Zusammenarbeit der Universität Tübingen und Amazon AWS im Paper „Towards Total Recall in Industrial Anomaly Detection“ veröffentlicht. In seiner zweiten Fassung vom 5. Mai 2022 wurde das Paper bei der Konferenz CVPR 2022 (Computer Vision and Pattern Recognition) akzeptiert und zählt mit über 290 Zitierungen zu einem der populärsten Paper im Bereich der (Unüberwachten –> TODO) Anomaliedetektion.

Die Grundlage dieses Ansatzes sind wiederum „Einbettungen“ (Embeddings) von Merkmalen (Feature), die aus den Eingabebildern mithilfe eines auf „ImageNet“ vortrainiertem „Convolutional Neural Network (CNN)“ erzeugt werden. Damit ähnelt sich die Methode PatchCore sowohl SPADE (2.6), also auch PaDiM (2.7) und greift die in 2.4.3 beschriebene Vorgehensweise auf. Wie wir später sehen werden, unterscheidet sich der Einbettungsprozess jedoch recht deutlich von den bisherigen Methoden. Weiter wird die eigentliche Anomaliedetektion, wie bereits bei der Methode SPADE mithilfe einer Nächsten Nachbar Suche (Nearest Neighbor Search; NN) in einer „Memory-Bank“ $\mathcal{M}$ durchgeführt. Die wesentliche Weitereentwicklung gegenüber SPADE liegt vor allem in der Methode, wie die Memory-Bank aufgebaut wird. Durch die Auswahl möglichst representativer Elemente in der Memory Bank, kann die Anzahl der Elemente in der deutlich reduziert werden, was einer Reduzierung der Laufzeit bedeutet.

Auch gut 2 Jahre nach Veröffentlichung ist die PatchCore Methode insbesondere auf dem MVTecAD-Datensatz (2.1) mit einer Genauigkeit (Auccuracy) von maximal 99,6% („PatchCore Ensemble“) absolut konkurrenzfähig und wird in vielen Veröffentlichungen als „State-of-the-Art“ Methode verwendet.

Im Laufe dieses Kapitels soll zunächst die Funktionsweise der Methode PatchCore erläutert werden. Anschließend evaluieren wir die Originalmethode im Hinblick auf Laufzeit und Genauigkeit. Im sich dann anschließenden Teil werden zahlreiche Modifikationen besprochen, die versuchen, die Laufzeit auf zu Reduzieren und dabei möglichst viel der Genauigkeit zu erhalten.

3.2 Funktionsweise

Abgeschlossen: 01.11. v1

Zunächst kann zwischen zwei Phasen unterschieden werden: Der Trainingsphase und der Testphase. In der Trainingsphase werden die „(locally aware) **Patch Features**“ aus den Trainingsbildern („Nominal Samples“) extrahiert. Hierzu wird ein „Pretrained Encoder“ verwendet, analog zu 2.4.3. Anschließend findet eine Unterabtastung bzw. eine Auswahl der Patch Features statt, die in der Memory Bank gespeichert werden. Dieser Vorgang wird als „Coreset Subsampling“ bezeichnet. Ist diese Memory Bank erzeugt, ist die Methode initialisiert und das Training abgeschlossen. In der Testphase werden die Patch Features auf die gleiche Weise aus den „Test Samples“ extrahiert, wie in der Trainingsphase. Jedes dieser Patch Features wird nun mit den Patch Features in der Memory Bank verglichen. Dies geschieht mit einer „Nearest Neighbor Search“ (NN). Aus den Distanzen zum Nächsten Nachbarn kann dann, wie in 2.7 eine räumlich aufgelöste Anomaliekarte erzeugt werden. Auf Grundlage dieser Anomaliekarte geschieht dann die Instanzklassifizierung als nominales oder anomales Bild. Nachfolgende Abbildung (3.1), die aus der Veröffentlichung übernommen wurde, zeigt die Funktionsweise der Methode PatchCore.

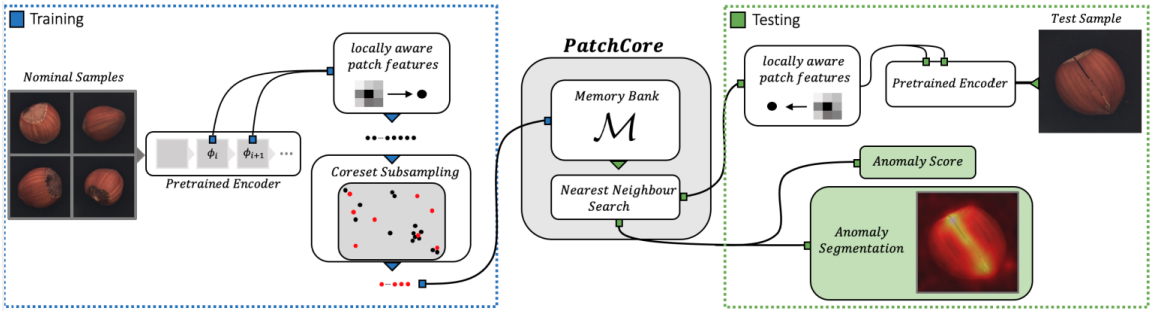


Abbildung 3.1: Skizze zur Funktionsweise von PatchCore. [26]

3.2.1 Erzeugen der Patch Features

Zunächst werden einige Notationen definiert, die im Folgenden verwendet werden und in analoger Form auch anderen Stellen in dieser Arbeit verwendet werden. So wird die Menge aller Trainingsbilder als $\mathcal{X}_{train}$ bezeichnet. Die Menge aller Testbilder als $\mathcal{X}_{test}$. Für den Trainingsdatensatz gilt im Sinne der Unüberwachtheit (TODO), dass es sich um ausschließlich nominale Samples handelt. Im Testdatensatz können sowohl nominale als auch anomale Samples enthalten sein. Bezeichnen wir die wahre Klassenzugehörigkeit eines Bildes x als y_x, so kann diese entweder 0 (nominal) oder 1 (anomal) sein. Für den Trainingsdatensatz gilt dann: $\forall x \in \mathcal{X}_{train} : y_x = 0$ und für den Testdatensatz $\forall x \in \mathcal{X}_{test} : y_x \in \{0, 1\}$.
Den bereits in 2.6 und 2.7 angetroffenen „Pretrained Encoder“ wird als ϕ bezeichnet. Es wird dabei, wie bereits gesehen, nicht der Ausgang dieses Netzwerkes benutzt, sondern die Feature

Maps aus einer oder mehreren Schichten j des Netzwerkes. Im Falle von ResNets, die auch in dieser Veröffentlichung hauptsächlich verwendet werden, ist $j \in \{1, 2, 3, 4\}$. j wird folgend auch als „Hierarchielevel“ bezeichnet und spielt eine wichige Rolle. $\phi_{i,j} = \phi_j(x_i)$ bezeichnet die Feature Map des Bildes $x_i \in \mathcal{X}$ aus den Hierarchieleveln j. Wie bereits in 2.6.2 diskutiert, ist eine sinnvolle Auswahl der Hierarchielevel eine wichtige Voraussetzung für gute Ergebnisse und wird in 3.5.2 genauer untersucht. Auch die Autoren von PatchCore weisen auf diese Problemstellung hin. Man könne, wie bei SPADE, die letzte Ebene in der Merkmalshierarchie des Netzes verwenden. Dies bringe aber die folgenden zwei Probleme mit sich. Erstens gehe dabei mehr lokalisierte nominale Informationen verloren. Das sei während der Trainingsphase kritisch, weil die Arten von Anomalien, die zum Testzeitpunkt auftreten, nicht im Voraus bekannt seien und die möglichst vollständige Erfassung des Normals notwendig sei. Zweitens seien die sehr tiefen und abstrakten Merkmale in den vortrainierten ImageNet-Netzwerken auf die Aufgabe der Klassifizierung natürlicher Bilder ausgerichtet, welche nur wenig direkte Überschneidungen mit der hier vorliegenden Aufgabe der industriellen Anomaliedetekion aufweise. Es wird deshalb vorgeschlagen, Merkmale aus den mittleren Hierarchielevels zu verwenden. Das entspricht bei ResNets $j = \{2, 3\}$. Wie in 2.4 zu erkennen, handelt es sich bei $\phi_{i,j}$ um einen dreidimensionalen Tensor: $\phi_{i,j} \in \mathbb{R}^{c^* \times h^* \times w^*}$ mit c^* als Tiefe der Feature Maps, h^* als Höhe und w^* als Breite. $\phi_{i,j}(h, w) \in \mathbb{R}^{c^*}$ bezeichnet dann den zur Position $h \in \{1, ..., h^*\}$ und $w \in \{1, ..., w^*\}$ gehörenden Vektor der Länge c^*. Unter der Annahmen, dass die Größe des Feldes im Originalbild x_i, das Einfluss auf ein $\phi_{i,j}(h, w)$ nimmt („Receptive Field“), so groß ist, um einen ausreichenden räumlichen Kontext zu erfassen, eignet sich dieser Vektor als „Patch Feature“ für eine gegenüber räumlichen Variationen robusten Anomaliedetektion.
Um diese wünschenswerte Annahme zu erfüllen, wird eine Aggregation der lokal umliegenden Regionen („local Neighborhood Aggregation“) durchgeführt, das nachfolgend vorgestellt wird und die Größe des rezeptiven Feldes steuert.
Dafür wird die oben eingeführte Notation für $\phi_{i,j}(h, w)$ um eine ungerade Feldgröße („patchsize“) p erweitert, die die benachbarten Feauture Vektoren mit einbezieht. Zunächst wird diese Nachbarschaft wie folgt definiert:

$$\mathcal{N}_p^{(h,w)} = \left\{(a, b)|a \in \left[h - \left\lfloor \frac{p}{2} \right\rfloor, ..., h + \left\lfloor \frac{p}{2} \right\rfloor\right], b \in \left[w - \left\lfloor \frac{p}{2} \right\rfloor, ..., w + \left\lfloor \frac{p}{2} \right\rfloor\right]\right\}$$

Damit ergeben sich schließlich ein „Patch Feature“ zu

$$\phi_{i,j}\left(\mathcal{N}_p^{(h,w)}\right) = f_{agg}\left(\{\phi_{i,j}(a, b)|(a, b) \in \mathcal{N}_p^{(h,w)}\}\right),$$

wobei f_{agg} eine Aggregationsfunktion ist. Die Aggregationsfunktionsfunktion, die in der PatchCore Methode verwendet wird, ist ein adptives „Average Pooling“, in einer Dimension, die unabhängig von der Länge der Eingangsfeature, immer eine feste Länge d ausgibt.
Da diese Operation für alle Paare von (h, w) mit $h \in \{1, ..., h^*\}$ und $w \in \{1, ..., w^*\}$ durchgeführt wird, wird die Auflösung der Feature Map erhalten. Für einen gesamten Feature Map Tensor ergibt sich dementsprechend:

$$\mathcal{P}_p\left(\phi_{i,j}\right) = \left\{\phi_{i,j}\left(\mathcal{N}_p^{(h,w)}\right)|h \in \{1, ..., h^*\}, w \in \{1, ..., w^*\}\right\}$$

Wie bereits erwähnt, kann diese Operation für verschiedene Hierarchielevel geschehen. Weil die Auflösung der Feature Maps mit steigendem Hierarchielevel abnimmt, wird $\mathcal{P}_p\left(\phi_{i,j+1}\right)$ berechnet und anschließend auf die Auflösung von $\mathcal{P}_p\left(\phi_{i,j}\right)$ linear interpoliert. Jedes Element wird dann mit dem korrespondieren Element, also dem Element an der gleichen Stelle, aggregiert. Es ist also notwendig, dass für jedes j die gleiche Featurelänge d gewählt wird.
Würde auf eine Auswahl der Patch Feature, wie im folgenden Abschnitt erläutert, verzichtet, würde sich folgende Memory-Bank ergeben:

$$\mathcal{M} = \bigcup_{x_i \in \mathcal{X}_{train}} \mathcal{P}_{s,p}\left(\phi_{i,j}\right)$$

3.2.2 Coreset Subsampling

Insbesondere, wenn $\mathcal{X}_{train}$ eine große Kardinalität hat, also viele Bilder enthält, wird die Memory-Bank $\mathcal{M}$ sehr groß. Wie bereits in 2.6 festgestellt ist diese Kardinalität besonders laufzeitkritisch, weil die Nächste Nachbar Suche in der Memory-Bank mit einer Komplexität von $\mathcal{O}(n)$ berechnet wird. Wie bereits in 2.7 zu sehen, ist ein „patchbasierter“ Vergleich zwischen allen Elementen in $\mathcal{M}$ und allen Elemnten in $\mathcal{P}_{s,p}\left(\phi_{i,j}\right)$ ein notwediger Schritt - nicht nur um die Anomaliekarte zu erstellen, die in dieser Arbeit ohnehin von keinem großen Interesse ist, sondern auch um eine robuste und präzise Instanzklassifizierung durchzuführen. Für die Laufzeitoptimierung ist es also wünschenswert, die Kardinalität der Memory-Bank zu reduzieren.
Auf der anderen Seite müssen die Elemente in $\mathcal{M}$ möglichst gut nominale Eigenschaften abbilden, um eine gute Anomaliedetektion zu ermöglichen. Wie in der Veröffentlichung gezeigt wird, (§4.4.2 - Importance of Coreset Subsampling) führt der naive Ansatz, zufällig Elemente aus $\mathcal{M}$ auszuwählen, nicht zu zufriedenstellenden Ergebnissen.
Das von PatchCore zugrundeliegenden Konzept setzt genau hier an. Es soll eine Teilmenge $\mathcal{S} \subset \mathcal{A}$ gefunden werden, bei der die Problemlösung über $\mathcal{A}$ am ehesten und vor allem schneller durch die über $\mathcal{S}$ berechnete Lösung approximiert werden kann. Dabei ist die Methode, die zu einer solchen Teilmenge führt, problemspezifisch. Im Falle von PatchCore wird eine Berechnung von Nächsten Nachbarn durchgeführt, weswegen gemäß [28] eine „MiniMax-Funktion“ sich anbietet, um eine annähernd ähnliche Abdeckung der $\mathcal{M}$ mit $\mathcal{M}_\mathcal{C}$ zu erreichen. Dies kann wie folgt gelöst werden:

$$\mathcal{M}_\mathcal{C}^* = \underset{\mathcal{M}_C \subset \mathcal{M}}{\arg\min} \max_{m \in \mathcal{M}} \min_{n \in \mathcal{M}_\mathcal{C}} \|m - n\|_2$$

Die exakte Berechnung von $\mathcal{M}_\mathcal{C}^*$ ist NP-schwer, also nicht in polynomieller Zeit berechenbar. Es handelt sich zwar um einen Prozess, der nicht während der Inferenz durchgeführt werden muss, sondern einmalig während der Trainingsphase, aber es muss dennoch mit iterativen, approximierenden Verfahren gearbeitet werden.
Aus [28] wird ein „Greedy Algorithmus“ übernommen, der iterativ Elemente aus $\mathcal{M}$ auswählt, die die größte Distanz zu allen bereits ausgewählten Elementen haben. Um die Laufzeit des Subsamplings weiter zu reduzieren, wird das „Johnson-Lindenstrauss Lemma“[10] verwendet, um die Dimensionalität der Elemente $m \in \mathcal{M}$ durch zufällige lineare Projektion $\psi : \mathbb{R}^d \to \mathbb{R}^{d^*}$

mit $d^* < d$ zu reduzieren. Anschaulich kann dies durch eine Punktwolke erklärt werden, die aus zufälligen Blickwinkeln betrachtet wird, wodurch die 3-D-Struktur als 2-D Struktur angenähert wird.

3.2.3 Bestimmen des Anomaliegrades

Wie bereits in der Einleitung zu diesem Katpitel erwähnt, ist die Grundlage der Bestimmung des Anomaliegrades die Distanz zu den Nächsten Nachbarn in der Memory-Bank. Das gilt sowohl für die Anomaliekarte bzw. die Segmentierung, als auch für die Instanzklassifizierung. Zunächst muss während der Inferenz aus einem Testbild $x_i \in \mathcal{X}_{test}$ die Patch Features extrahiert werden. Dies geschieht auf die gleiche Weise, wie in der Trainingsphase:

$$\mathcal{P}(x_i) = \mathcal{P}_p\left(\phi_j(x_i)\right)$$

Diese Menge $\mathcal{P}(x_i)$ enthält nun die Patch Features m^{test} des Testbildes x_i. Nun gilt es zu jedem Patch Feature in $\mathcal{P}(x_i)$ den Nächsten Nachbarn in der Memory Bank $\mathcal{M}$ zu finden:

$$m^* = \arg\min_{m \in \mathcal{M}} \left\| m - m^{test} \right\|_2, \forall m^{test} \in \mathcal{P}(x_i)$$

Es gilt zu beachten, dass jedes m^{test} zu einer Position (h, w) im Bild x_i gehört. So kann eine räumlich aufgelöste Anomaliekarte M erzeugt werden, die die Distanz zu den Nächsten Nachbarn in der Memory-Bank enthält:

$$M = \left(\left\| m^*_{h,w} - m^{test}_{h,w} \right\|_2 \right)_{h,w}, \forall (h, w) \in \{1, ..., h^*\} \times \{1, ..., w^*\}$$

Diese Anomaliekarte M kann schließlich mittels bilinearer Interpolation und einer anschließenden Glättung auf die Auflösung des Originalbildes x_i gebracht werden, wodurch eine pixelweise Klassifikationskarte möglich wird. Weil dies in dieser Arbeit nicht von Interesse ist, wird darauf nicht weiter eingegangen.
Die Instanzklassifizierung könnte nun analog zu PaDiM (2.7.1) durch Maximalwerbildung durchgeführt werden. Die Autoren von PatchCore gehen ähnlich vor, fügen jedoch noch einen Gewichtungsschritt hinzu. Zunächst wird ganz analog vorgegangen und die maximale Distanz zu den Nächsten Nachbarn herangezogen, was nichts anderes als der Maximalwert der Anomaliekarte M ist.

$$m^{test,*}, m^\star = \arg\max_{m^{test} \in \mathcal{P}(x_i)} \arg\min_{m \in \mathcal{M}} \left\| m - m^{test} \right\|_2$$

$$s^* = \left\| m^{test,*} - m^\star \right\|_2 = \max\{M\}$$

Die Gewichtung schließt die nächsten b Patch Features in M zu $m^\star$ mit ein. Diese Menge notieren wir als $\mathcal{T}_b(m^\star)$. Der Gedanke hinter dieser Gewichtung ist, den Anomaliegrad s dann zu erhöhen, wenn die Feature Patches in $\mathcal{T}_b(m^\star)$ selbst weit entfernt vom Anomaliekandidaten

Patch Feature $m^\star$ sind und es sich somit ohnehin um seltene nominale Patch Features handelt. Der finale, für die Instanzklassifiziergung entscheidende Anomaliegrad s ergibt sich dann zu:

$$s = \left(1 - \frac{e^{\left\|m^\star - m^{test,*}\right\|_2}}{\sum_{m \in \mathcal{T}_b(m^\star)} e^{\left\|m - m^{test,*}\right\|_2}}\right) \cdot s^*$$

Durch diese Gewichtung wird die Instanzklassifizierung robuster und erhöht die Instanzklassifiziergungsgenauigkeit.

3.3 Ergebnisse und Diskussion der Originalmethode

Abgeschlossen: 01.11. v1

In diesem Abschnitt werden die Ergebnisse der Originalmethode PatchCore vorgestellt und diskutiert. Dies soll vor allem im Hinblick auf die Eignung für eine Implementierung auf einem ressourcenbeschränkten Gerät, nämlich einem RaspberryPi 4B (2.8) geschehen. Es wird sich dabei auf den hier verwendeten Datensatz MVTecAD (2.1) bezogen. Weitere Datensätze, die in der Veröffentlichung verwendet wurden, werden mit dem Verweis auf die Veröffentlichung nicht weiter betrachtet. Die zur Evaluation herangezogene Metrik ist AUROC (2.3).

Wie bereits in der Einleitung erwähnt, ist die Methode PatchCore grundsätzlich in der Lage, sehr hohe Instanzklassifiziergungsgenauigkeiten zu erreichen, die auch zwei Jahre nach Veröffentlichung noch zu den besten gehören. Erreicht wird dies teilweise durch ein Ensemble von verschiendenen Feature Extraktoren bzw. Backbones und höhere Auflösungen. So wird ein „DenseNet 201“[17], ein „ResNext 101“[34] und ein bereits bekanntes Wide ResNet mit 101 Schichten [37] verwendet. Steht eine GPU zur Verfügung, welche, wie in 2.4 zu sehen, die Laufzeit von CNNs deutlich reduziert, können selbst mit einer solchen Konfiguration, bestehend aus vielen Backbones, einigermaßen schnelle Inferenzzeiten erreicht werden. Da die Zielsetzung dieser Arbeit jedoch die Implementierung auf einem RaspberryPi 4B ist und bereits gezeigt wurde, dass das Ausführen von CNNs auf einem solchen Gerät äußerst laufzeitkritisch ist, ist eine solche Konfiguration im Kontext dieser Arbeit nicht sinnvoll. Beschränken wir uns auf Konfigurationen, mit nur einem Backbone und auf die Auflösung von 256×256 Pixeln, so erreicht PatchCore mit einem Wide ResNet-50 Backbone eine Instanzklassifiziergungsgenauigkeit (AUROC) von 99,1 mit einer Reduktion der Memory Bank auf 25%. Wird die Größe der Memory Bank auf 1% reduziert, verschlechtert sich der AUROC nur leicht auf 99,0%. Wie im Laufe dieses Kapitels noch zu sehen sein wird, ist die Reduzierung der Kardinalität der Memory Bank ein wichtiger Schritt, um die Laufzeit zu reduzieren. Die hier angewandte Methode des Coreset Subsamplings ist also ein wichtiger Bestandteil und eine Errungenschaft der Methode PatchCore.

Neben dem Erzeugen der Patch Feature ist die Nächste Nachbar Suche in der Memory Bank besonder laufzeitkritisch. In der Veröffentlichung lediglich eine kleine Randnotiz im Anhang, ist diese Suche mit „FAISS“ implementiert. FAISS ist eine Bibliothek, entwickelt von Facebook's AI Research Department, die die Nächste Nachbar Suche enorm beschleunigt und im Laufe

dieses Kapitels (Abbildung 3.3) nochmal genauer erläuert und analysiert wird.
Insgesamt ist die Methode PatchCore zwar in ihrer Originalform eine ausgezeichnete Basis, aber für die Implementierung auf einem RaspberryPi 4B nur eingeschränkt geeignet. Es werden im Folgenden zahlreiche Adaptionen vorgestellt und evaluiert, die das Ziel haben, die Laufzeit zu reduzieren und dabei möglichst viel der Genauigkeit zu erhalten.

3.4 Testaufbau

In diesem Abschnitt wird die Testumgebung vorgestellt, die für die Evaluation der PatchCore Methode verwendet wurde. Dabei sind viele der hier aufgeführten Vorgehensweisen auf 4 und 5 übertragbar.

Hardware

Es stehen grundsätzlich zwei Testumgebungen zur Verfügung. Das Ziel ist es zwar, die Implementierung auf einem RaspberryPi 4B zu ermöglichen, auf dem Weg dorthin, ist eine potentere Hardware aber notwendig.
Die Entwicklungsgeschwindigkeit hängt zum einen auch von der Laufzeit der Trainingsphase ab, die auf einem RaspberryPi 4B sehr lange dauert. Außerdem sind Ergebnisse, die auf der Desktop-Hardware erzeugt wurden, im Falle der Genauigkeit ganzheitlich übertragbar, nehmen, aber nur einen Bruchteil der Zeit in Anspruch. So spielt die Hardware, auf der eine identische Methode ausgeführt wird, für die Instanzklassifiziergung keine Rolle. Sind also lediglich die Instanzklassifiziergungsgenauigkeiten in einem Abschnitt von Relevanz, weil keine oder bekannte Unterschiede in der Laufzeit bestehen, kann auch auf eine GPU zurückgegriffen werden.
Laufzeitmessungen werden über weite Teile dieser Arbeit auf der Desktop-Hardware durchgeführt. Zwar sind nicht nur die reine Rechenleistung zwischen den Prozessoren der beiden Geräten unterschiedlich, auch die CPU-Architektur (ARM vs. x86) spielt eine Rolle. Jedoch sind die Ergebnisse qualitativ übertragbar. Aufgrund der Vielzahl an Adaptionen, die im Laufe dieser Arbeit getestet werden, wurde sich dazu entschlossen, die Laufzeitmessungen nur dann auf dem RaspberryPi 4B durchzuführen, wenn entweder große relative Abweichungen zu den Messungen auf der Desktop-Hardware zu erwarten sind oder es sich um eine finale Konfiguration handelt.
Im Folgenden werden einige relevante Informationen zum Desktop-System aufgeführt.

- CPU: AMD Ryzen 5 5600X (6 Kerne, 12 Threads, @ 3,7 GHz)
- RAM: 32 GB DDR4 @ 3200 MHz
- GPU: Nvidia GeForce RTX 3060Ti (8 GB GDDR6)
- OS: Ubuntu 23.10 (Linux, Kernel 6.2.0-generic)

Detailierte Information zur Hardware des RaspberryPi finden sich in 2.8.

3.4.1 Software

Wie in der Forschung weit verbreitet und auch in allen Veröffentlichungen, die in dieser Arbeit verwendet werden, wird die Programmiersprache **Python** verwendet. Es kommt dabei die Version *3.10* zum Einsatz.
Ebenfalls der Konvention in der Forschung entsprechend, wird das **PyTorch** Framework in der Version *2.0.1* verwendet. PyTorch ist ein Open-Source-Framework für maschinelles Lernen, das von Facebooks AI Research Lab (FAIR) entwickelt wurde. Es hat aufgrund seiner Flexibilität, seines dynamischen Berechnungsgraphen und seiner Benutzerfreundlichkeit in der Community für maschinelles Lernen und Deep Learning große Beliebtheit erlangt. PyTorch bietet eine Python-basierte Schnittstelle für die Entwicklung neuronaler Netze und anderer Modelle für maschinelles Lernen. kann mit PyTorch effizient abgebildet werden.[24]
Die Nächste Nachbar Suche wird mit der Bibliothek **FAISS** durchgeführt. FAISS ist eine Bibliothek, die von Facebooks AI Research Lab (FAIR) entwickelt wurde. Wie bereits im vorherigen Abschnitt erwähnt, wird die Nächste Nachbar Suche jedoch in den meisten Fällen mit der Bibliothek **FAISS** durchgeführt. FAISS (Facebook AI Similarity Search) ist eine leistungsstarke Bibliothek, die ebenfalls vom KI-Forschungsteam von Facebook (FAIR) für die effiziente und skalierbare Ähnlichkeitssuche und die Suche nach dem nächsten Nachbarn in großen Datensätzen entwickelt wurde. FAISS wurde insbesondere für die Verarbeitung hochdimensionaler Daten entwickelt und eignet sich daher besonders gut für Aufgaben, die Merkmalsvektoren beinhalten, wie z. B. Einbettungen aus Deep-Learning-Modellen. Es nutzt Techniken wie Indexstrukturen, Quantisierung und GPU-Beschleunigung, um Suchvorgänge erheblich zu beschleunigen.[19]
Daneben werden bekannte Bibliotheken wie **numpy** oder **scikit-learn** verwendet. Zur besseren Organisation des Codes wird ein modularer Aufbau verwendet, der durch das Framework **pytorch lightning** ermöglicht wird.

Laufzeit- und AUROC-Messungen

Die Laufzeitmessungen werden mit dem Python-Modul **time** bzw. der Methode **perf_counter()** durchgeführt. Zunächst wird immer nur die Laufzeit für ein einzelnes Bild betrachtet. Erst im Anschluss wird mit dem Durchsatz („Throughput“) die Laufzeit von einem Ensemble (Batch) an Bildern betrachtet (TODO –> Ref).
Ein einzelnes Bild durchläuft, während einer Laufzeitmessung 3 mal einen sogenannten „Warm-Up“ Prozess, der im Wesentlichen dazu dient, mögliche Overheads in Form von Initialisierungsprozessen, die im Hintergrund ablaufen, auszuschließe und zusätzliche, die Hardware in einen

authentischen thermischen Zustand zu bringen. Keiner dieser Prozesse geht in die Laufzeitmessung direkt ein. Diese folgt für jedes Bild einzeln, indem die Inferenz 5 mal durchgeführt wird und die Laufzeiten gemittelt werden. Es werden hierbei 5 Zeitpunkte innerhalb des Prozesses mit *perf_counter()* festgestellt. Diese sind:

- **Start**: Der Zeitpunkt, an dem die Inferenz beginnt.
- **Ende: Feature Extraktion**: Der Zeitpunkt, an dem die Feature Extraktion durch den Backbone abgeschlossen ist (3.2.1).
- **Ende: Einbettungsprozess**: Der Zeitpunkt, an dem die Patch Feature vorliegen (3.2.1).
- **Ende: Nächste Nachbar Suche**: Der Zeitpunkt, an dem die Nächse Nachbar Suche abgeschlossen ist.
- **Ende: Anomalieprozess**: Der Zeitpunkt, an dem der Anomaliegrad berechnet wurde und der damit Prozess abgeschlossen ist (3.2.3).

Aus den Differenzen dieser Zeitstempel lassen sich dann präzise die Laufzeiten für die einzelnen Prozesse und den Gesamtprozess bestimmen. Da es jedoch zu Schwankungen in der Laufzeit kommt, ist eine Mittelwertbildung notwendig. Dies geschieht in zweifacher Hinsicht. Zunächst wird für jedes Bild über die 5 Durchläufe gemittelt. Das wird für jedes Bild wiederholt, sodass für jedes Bild eine Laufzeit vorliegt. Aus diesen Laufzeiten wird dann ebenfalls der Mittelwert gebildet, der als eigentlicher Messwert für eine Konfiguration dient.
Anzumerken ist, dass in dieser Arbeit, wie bereits in 2.3 erläutert, auf eine Segmentierung verzichtet wird. Dementsprechend wird dieser Prozess, soweit nicht für die Instanzklassifiziergung notwendig, übersprungen und insbesondere nicht laufzeittechnisch erfasst.
Ebenfalls wird der Initialisierungsprozess bzw. die Trainingsphase nur in exemplarischen Fällen betrachtet. Der Initialisierungsprozess ist einmalig und kann auch somit auch auf potenter Hardware durchgeführt werden. In einer produktiven Umgebung ist dieser Trainingsprozess ohnehin nicht relevant, weil bereits abgeschlossen.
Wie in (TODO –> ref) zu sehen, hängt die Laufzeit, genauer gesagt, die NN-Suche, stark von der Kardinalität von der Memory Bank $\mathcal{M}$ ab. In der Literatur allgemein üblich ist, dass ein relatives Subsampling stattfindet. Wie wir 2.1 entnehmen können, hat jede Kategorie unterschiedlich viele Bilder im Trainingsdatensatz und teilweise verschiedene Auflösungen, wodurch sich eine unterschiedliche absolute Anzahl an Patch Featuren ergeben. Um die Laufzeitmessungen vergleichbar zu machen, wird die Kardinalität der Memory Bank in den meisten Fällen auf 1000 gesetzt, wenn nicht anders angegeben. Dies dient vor allem dazu, die Laufzeitmessungen für alle Klassen gleichermaßen geltend zu machen. In diesem Sinne werden auch die Auflösungen der Bilder auf 256×256 Pixel gesetzt, was dem Vorgehen der meisten Veröffentlichungen in diesem Bereich entspricht. Der Skalierungsprozess wird dabei nicht als Teil der Laufzeit betrachtet.
Dieser ermöglicht, dass die Laufzeitmessung für eine Konfiguration für eine Klasse ausreichend ist. Dies reduziert den Zeitaufwand für eine Messung einer Konfiguration über alle Klassen deutlich, weil zum einen auf die Wiederholungen verzichtet werden kann, andererseits für die verbleibende Bestimmung der AUROC auch eine deutlich schneller arbeitende GPU verwendet

werden kann.
Wie bereits in 2.3 ausgeführt, ist die AUROC eine ideale Metrik um die Güte eines Anomaliedetektors zu bewerten. Hierzu werden zunächst für alle Bilder im Testdatensatz die Anomaliegrade berechnet. Auf Grundlage dieser Werte, wird dann die AUROC berechnet. Wie in 3.2.2 ausgeführt, wird bei der Berechnung der Elemente, die in die Memory-Bank übernommen werden, ein Algorithmus verwendet, der randomisiert arbeitet. Um diesen zufälligen Einfluss zu minimieren, werden aus den identischen Patch Features 5 Memory Banks $\mathcal{M}$ erzeugt, deren AUROC ebenfalls gemittelt wird, um einen möglichst konsistenten Schätzer zu erhalten. Das geschieht für alle Klassen aus MVTecAD (2.1) und der Klasse des einigen Datensatzes (Granulat, 2.2). Während für den eigenen Datensatz der AUROC explizit angegeben wird, wird für die Klassen aus MVTecAD nur der Mittelwert über alle Klassen angegeben. Die einzelnen Ergebnisse sind aber archiviert und können beim Autor erfragt werden.
Die meisten der hier zu sehenden Plots wurden mit **tikz** und **matplotlib** direkt auf Grundlage der Messergebnisse erzeugt. Die Annotationen sind sinnvoll gerundet, um eine bessere Lesbarkeit zu ermöglichen.
Die Grundlage der Implementierung der Methode PatchCore liefert dabei die offizielle Implementierung [26] und eine inoffizielle Implementierung [15]. Diese wurden jedoch jeweils derart modifiziert, dass nur noch wenige Elemente der ursprünglichen Implementierung übrig geblieben sind. Der gesamte Code mit dem die folgenden Ergebnisse und Messungen erzeugt wurden, ist abrufbar.

3.5 Adaptionen und Messergebnisse

Der hier beginnende Abschnitt ist der umfangreichste dieser Arbeit. Er besteht als vielen Adaptionen, die im Laufe der Arbeit durchgeführt wurden, um die Laufzeit zu reduzieren. Jeder Unterabschnitt beschäftigt sich damit mit einer Adaption oder einer Kombination von Adaptionen. Zunächst wird dabei, die Idee hinter der Adaption erläutert. Messergebnisse werden dann präsentiert und diskutiert. Schließlich findet eine Einordnung und Bewertung statt.

3.5.1 Originalmethode

Zunächst wird die Originalmethode ohne Adaption betrachtet. Es wird also die Methode PatchCore mit einem Wide ResNet-50 Backbone und einer Auflösung von 256×256 Pixeln verwendet. Die Laufzeitmessungen wurden mit der CPU des Desktop-PCs durchgeführt.
In 3.2 ist die Laufzeit für die einzelnen Prozesse und den Gesamtprozess zu sehen, sowie die erreichte AUROC Klassifizierungsgenauigkeit. Es lässt sich erkennen, dass die Angaben aus dem Paper sich verifizieren lassen. Es kann daraus abgeleitet werden, dass die im Rahmen dieser Arbeit erarbeitete Implementierung korrekt ist.
Eine schon besprochene Bemerkung kann ebenfalls abgelesen werden. Die von PatchCore verwendete Subsampling-Methode ist in der Lage, die Kardinalität der Memory Bank $\mathcal{M}$ deutlich

zu reduzieren, ohne die AUROC Klassifizierungsgenauigkeit zu stark zu beeinflussen. Eindeutig ist auch zu erkennen, dass diese Reduktion der Kardinalität ein wesentlicher Bestandteil ist, möchte man eine möglichst geringe Laufzeit erreichen. Trotz Verwendung der State-of-the-Art Methoden, die von FAISS bereitgestellt werden, ist die Nächste Nachbar Suche der laufzeitkritischste Prozess, wenn ein Subsampling $> 10\%$ verwendet werden soll.
Am Granulatdatensatz lässt sich sogar feststellen, dass zumindest in einzelnen Fällen, ein Subsampling der Instanzklassifiziergungsgenauigkeit sogar zuträglich sein kann. Wie bereits ausgeführt, wird im Rahmen dieser Arbeit in den meisten Fällen kein relatives Subsampling durchgeführt, sondern eine absolute Kardinalität von 1000 verwendet. Für manche Klassen mit wenigen Trainingsbeispielen und geringerer Auflösung bedeutet das, es werden mehr Patch Feature in der Memory Bank sein, als bei einem relativen Subsampling mit 1%. In den meisten Fällen läge die relative Subsamplingrate, die eine Kardinalität von 1000 erzeugt, bei $< 1\%$. Berechnet man den Mittelwert über alle Klassen, so ergibt sich eine durchschnittliche relative Subsamplingrate von $\approx 0{,}6\%$. Die erzeugten Ergebnisse entsprechen somit den Erwartungen und den Ergebnissen aus der Veröffentlichung.
Der in der Originalmethode verwendete Einbettungsprozess, der in 3.2.1 beschrieben ist, bietet

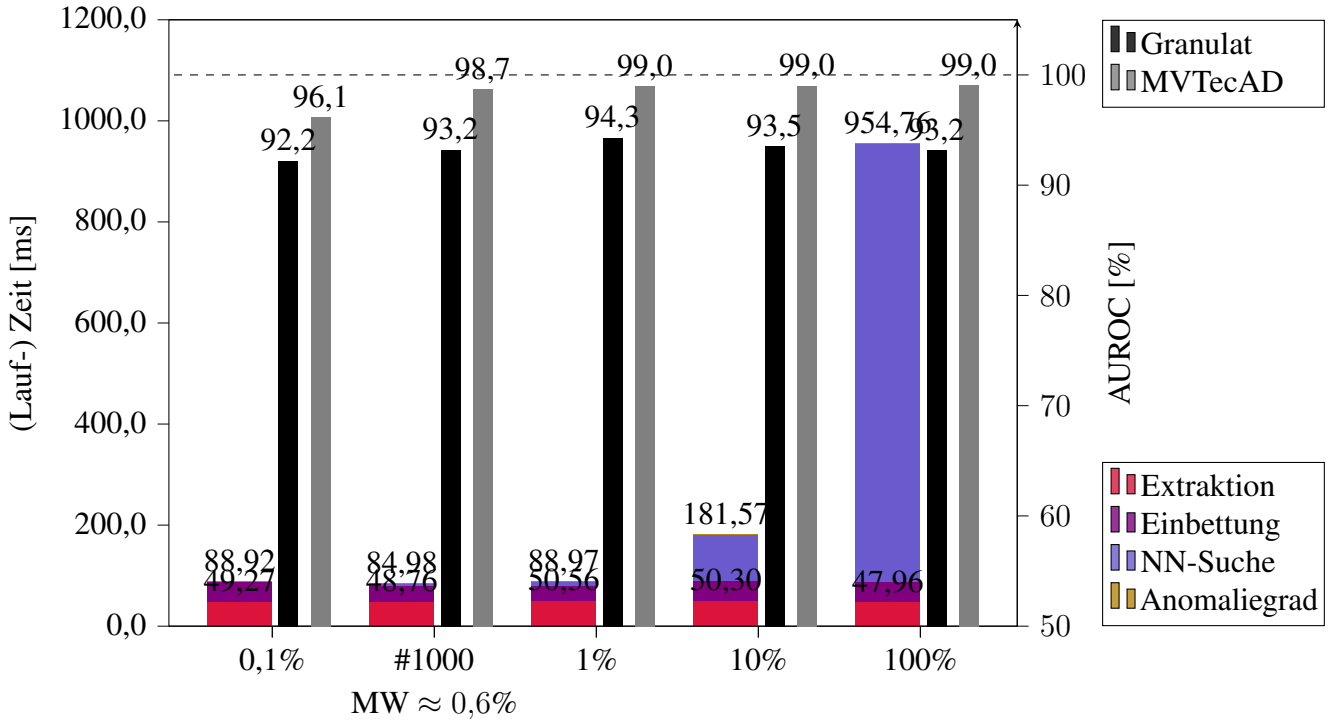


Abbildung 3.2: PatchCore: Originalmethode mit unterschiedlicher Anzahl an Patch Featuren in Memory Bank.

einen Parameter d, der die Länge der Patch Features bestimmt. In der 3.2 wurde keine Dimensionsreduktion durchgeführt, indem d der Länge der aus Layer3 ($j = 3$) extrahierten Feature entspricht, nämlich 1024. Mithilfe eines kleineren Parameters d kann die Laufzeit weiter reduziert werden. In 3.2 ist die Laufzeit für unterschiedliche Werte von d und zwei verschiedenen

Methoden zur Bestimmung der Nächsten Nachbarn und den korrespondierenden Distanzen zu sehen. Diese Abbildung inkludiert bei (1) und (4) Einbettungsmethoden, die nicht in der Originalmethode vorgeschlagen worden sind und in Abschnitt (TODO –> ref) erläutert werden. An dieser Stelle ist wichtig, dass die Merkmalslänge dadurch nochmal größer ausfällt ($d = 1536$) als bei der Originalmethde und dem größten sinvollen Wert für $d = 1024$. Anhand von 3.3 können zweierlei Phänomene erkannt werden.

Zum einen ist die in der Veröffentlichung verwendete Methode zur Bestimmung der Nächsten

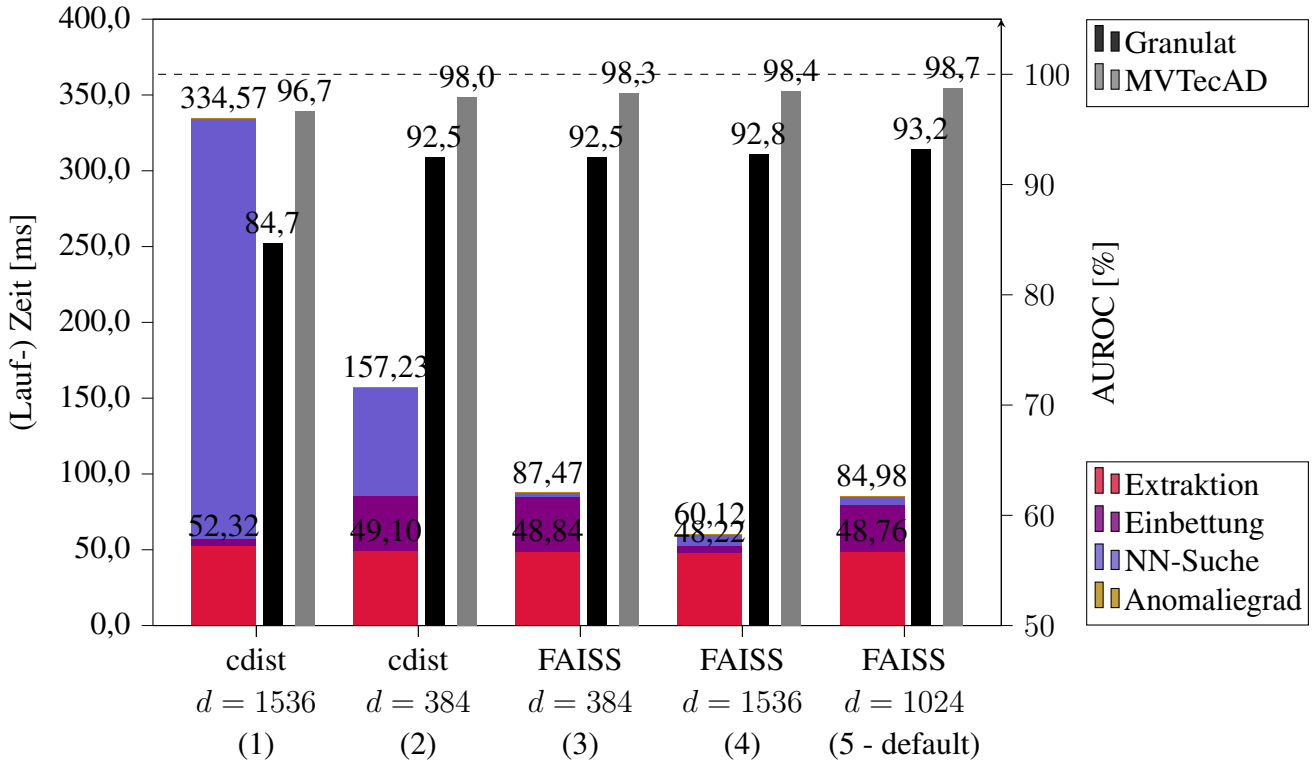


Abbildung 3.3: FAISS im Vergleich mit SciPy's cdist und unterschiedlichen Merkmalslängen d.

Nachbarn, die auf FAISS basiert, deutlich schneller als das Berechnen der Distanzen mit SciPy's Funktion cdist[9]. Dabei ist diese Methode, wie der Name bereits nahelegt, in der Programmiersprache C implementiert und kann somit als sehr performant angesehen werden. Es wird allerdings zu jedem Patch Feature in $\mathcal{P}(x_i)$ die Distanz zu jedem Patch Feature in $\mathcal{M}$ explizit berechnet. FAISS beschleunigt diesen Ansatz enorm durch Methoden wie Quantisierung und Indexstrukturen. Für eine genauere Erläuterung der Funktionsweise von FAISS wird auf [19] verwiesen. Der positive Effekt durch FAISS auf die Laufzeit steigt naheliegenderweise mit der Länge der Patch Features, ist aber auch bei $d = 384$ schon deutlich zu erkennen.

Das zweite Phänomen, das hier kurz besprochen werden soll, ist, dass FAISS deutlich weniger unter dem als „Curse of Dimensionality" (*dt.:* Fluch der Dimensionalität) bezeichneten Problem leidet. Der Fluch der Dimensionalität bezieht sich auf das Phänomen, dass der euklidische Abstand zwischen Punkten in einem hochdimensionalen Raum mit zunehmender Anzahl von Dimensionen an Aussagekraft verliert, so dass es schwierig wird, die Ähnlichkeit oder den

Abstand zwischen Punkten genau zu messen.(TODO –> find ref) Durch Quantisierung bzw. aufteilen in kleinere Vektoren, die dann in Indexstrukturen abgelegt werden, kann FAISS dieses Problem umgehen. Es lässt sich in 3.3 erkennen, dass, je größer d ist, dieser negative Effekt immer stärker sich in der Genauigkeit der Instanzklassifiziergung niederschlägt. So kommen die beiden Suchverfahren bei $d = 384$ noch auf recht ähnliche Ergenisse ((2) und (3)), während bei $d = 1536$ ((1) und (4)) die Instanzklassifiziergungsgenauigkeit bei der Verwendung von cdist deutlich schlechter ist als bei der Verwendung von FAISS.
Es lässt sich also festhalten, dass die Verwendung von FAISS essentiell ist, einerseits um eine präzise Anomaliedetektion auch mit hochdimensionalen Merkmalsvektoren zu ermöglichen, andererseits um die Laufzeit zu reduzieren.

3.5.2 Feature Extraktor - Wahl des Backbones

Es konnte bereits in 3.3 anhand von (5 - default) erkannt werden, dass die Feature Extraktion, also der „forward pass“ durch den Backbone, einen Großteil der Laufzeit für sich in Anspruch nimmt. Naheliegend ist deshalb, auch hier anzusetzten. In diesem Abschnitt wird ein Feature Extraktor gesucht, der einen möglichst guten Kompromiss aus Laufzeit und Genauigkeit bietet. Es steht hierfür eine sehr große Auswahl an möglichen Architekturen zur Auswahl. EfficientNets [29] und DenseNets [17] wurden bereits im Zusammenhang mit PatchCore Ensemble erwähnt. In vielen Veröffentlichung wurden Studien durchgeführt und festgestellt, dass insbesondere EfficientNets eine valide Wahl darstellen, aber gegenüber den hier in dieser Arbeit verwendeten Architekturen keine signifikanten Vorteile bieten. Es wurde sich deshalb entschieden, ausschließlich mit bewährten ResNet Architekturen zu arbeiten und nur Architekturen, die aus anderen Gründen interessant sind, im Rahmen dieser Arbeit quantitativ zu evaluieren.
Die in jüngerer Vergangenheit im Bereich der Künstlichen Intelligenz sehr erfolgreich angewandten „Transformer“ („Attention is all you need“[32]) basierten Architekturen, wie DeiT [30] oder CaiT [31], bieten auch in der Unüberwachten Anomaliedetekion ein spannendes Potential. Die Methoden FastFlow [36] und CAINNFlow [35] haben solche Netzwerke bereits erfolgreich für die Anomaliedetekion nutzbar gemacht. In beiden Veröffentlichungen werden neben diesen Transformer-Architekturen jedoch auch ResNet-Architekturen verwendet, die in ihrer Leistungsfähigkeit kaum schlechter abschneiden. Da Transformer-Architekturen im Allgemeinen, insbesondere bei der Berechnung ohne GPU, deutlich laufzeitkritischer sind, als ResNets, wird in dieser Arbeit auch auf die Verwendung von Transformer-Architekturen verzichtet. Zwar gibt es Bestrebungen, die Laufzeit von Transformer basierten Netzwerken zu reduzieren, die auch durchaus erfolgreich sind. Allerdings ist eine der schnellsten und kompaktesten Varianten von Transformer basierten Netzwerken, die Architektur „MobileViT S“ [23], immer noch deutlich langsamer als gängige CNN Architekturen (Tabelle 3 in [23]). Im Folgenden werden dementsprechend vor allem ResNets behandelt.
Es wird zusätzlich auf eine verhältnismäßig neue Architektur, die „ConvNexts“, eingegangen, weil diese in bislang noch keiner dem Autor bekannten Veröffentlichung als Feature Exktraktor

untersucht wurde. Es handelt sich dabei um eine CNN-Architektur, die für die Bildklassifizierung eine bemerkenswerte Konkurrenz zu Transformern darstellt, dabei aber auch weniger rechenintensiv ist.
Außerdem werden leichtgewichtige Feature Extraktoren, die durch Wissens Distilation (TODO –> Ref) erzeugt wurden, untersucht. Das Konzept hierfür stammt aus der noch ausführlicher besprochenen Methode EfficientAD (4).
Auch die Vielzahl an ResNet Varianten verlangt eine profunde Vorauswahl. In Frage kommen sämtliche Architekturen, die eine deutlich kürzere Laufzeit verprechen, als die in der Originalmethode verwendete Architektur Wide ResNet-50. Das sind, wie bereits in 2.3 zu sehen, ResNet 34 und ResNet 18.

ResNet

In diesem ausführlichen Abschnitt werden die Architekturen ResNet 18 und ResNet 34 untersucht. Diese Netze sind die kompaktesten Netze der ResNet-Familie und haben sich in zahlreichen (TODO –> RN50 inkludieren?) Anwendungen bewährt. Sie bieten außerdem den Vorteil, dass die Anzahl an Ausgabkanälen im Vergleich zu sämtlichen größeren ResNet Varianten um den Faktor 4 geringer ist und exakt den Angaben aus 2.4 entspricht. Neben der Feature Extraktion sind somit auch die Folgeprozesse der Einbettung und der NN-Suche weniger umfangreich und somit schneller ausführbar.
Es wird außerdem untersucht, welche Kombination an Hierarchielevel j sinnvoll sind. Abschließend findet eine Bewertung statt mit dem Ziel, einige wenige Varianten auszuwählen, die als Grundlage für weitere Untersuchungen dienen.
Es sei an dieser Stelle angemerkt, dass, wie bereits in der Implementierung zu PatchCore vorgesehen, die Berechnung der Feature Maps immer nur bis zum größten j durchgeführt wird. Der „forward pass“ wird also terminiert, wenn alle notwendigen Ausgaben erzeugt wurden. Da die Feature Maps frühere Schichten aufgrund der sequentiellen Struktur des Netzwerks zuerst zur Verfügung stehen, ist die Laufzeit der Extraktion stark abhängig von dem größten Hierarchielevel j.
Im Gegensatz zur Originalimplementierung wird das Netz hier zusätzlich auf die benötigten Elemente beschränkt. Wird ein ResNet beispielsweise nur bis Layer2 ausgeführt, so werden sämtliche Gewichte, die für Layer3, Layer4 und den Klassifikator notwendig sind, nicht geladen. Das beschleunigt zwar die Feature Extraktion nicht direkt, spart aber Speicherplatz und steigert somit dennoch die Effizienz der Methode.
Neben den verschiedenen Modellen, die hier das Wide ResNet 50 ersetzen, wird eine weitere Modifiktion vorweggegriffen. Wie bereits in 3.3 zu sehen, ist die Laufzeit, die vom Einbettungsprozess in Anspruch genommen wird, groß. Es wird deshalb in (TODO –> Ref) eine alternative Variante vorgestellt, die den Einbettungsprozess vereinfacht und die Laufzeit deutlich reduziert, wie ebenfalls in 3.3 anhand von (4) zu sehen ist. Aufgrund dieser laufzeittechnischen Überlegenheit wird dieser Prozess in diesem Abschnitt, aber auch für den Rest der Arbeit, als Standard verwendet. Die in diesem Abschnitt herausgearbeiteten Ergebnisse gelten qualitativ

aber auch für den originalen Einbettungsprozess und steheh im Einklang mit Abbildung 4 in der Veröffentlichung [26].

ResNet 18 Es ist naheliegend einen möglichst leichten Feature Extraktor zu verwenden, um die Laufzeit zu verringern. Wie bereits in 2.3 zu sehen, ist ResNet 18 die leichteste Architektur, die in Frage kommt und somit der erste Kandidat. Betrachtet man zunächst jede Hierarchieebene für sich, ergibt sich 3.4. Es lässt sich leicht erkennen, dass $j = \{4\}$ aus zweierlei Perspektive nicht geeignet ist. Zum einen ist die Laufzeit für die Feature Extraktion im Verhätnis zu den anderen Hierarchieleveln sehr hoch. Zum anderen ist die AUROC Klassifizierungsgenauigkeit deutlich schlechter als bei den anderen Hierarchieleveln.
Die nächst beste Hierarchieebene ist $j = \{1\}$. Es erreicht die niedrigste Laufzeit in diesem Feld mit lediglich $7{,}25\mathrm{ms}$ je Bild auf der Desktop-CPU. Das ist zwar fast doppelt so schnell, wie mit $j = \{4\}$, aber die NN-Suche benötigt deutlich mehr Zeit, als bei den anderen Varianten. Das erscheint zunächst kontraintuitiv, weil die Merkmalslänge hier mit $d = 64$ am geringsten ist. Betrachtet man 2.4 wird aber klar, dass durch die hohe räumliche Auflösung der Feature Map an Stelle $j = 1$ die Anzahl an Patch Featuren, die für ein Testbild mit der Memory Bank verglichen werden, deutlich größer ist. Aufgrund der Halbierung der Seitenlänge pro Hierarchielevel ist die Anzahl an elementweisen Distanzberechnungen für jedes $j = j + 1$ doppelt so groß, was sich auch in etwa in einer Verdoppelung der benötigten Laufzeit für die NN-Suche niederschlägt. Dies gilt analog auch für das ResNet 34 und auch das Wide ResNet 50. Zwar hat das Wide ResNet 50 für jeden Kanal die 4-fache Merkmalslänge, die Verhältnisse zwischen den Hierarchieleveln bleiben aber erhalten.
Für den eigenen Datensatz ist $j = \{3\}$ die beste Wahl, betrachtet man nur einzelne Hierarchielevel. Mit absolut $0{,}9\%$ schlechterer AUROC auf dem Granulat Datensatz, aber mit $97{,}0\%$ auf MVTecAD deutlich präziser, ist $j = \{2\}$ hier aber die insgesamt beste Wahl, die auch mit einer kurzen Laufzeit von $8{,}47\mathrm{ms}$ überzeugt.
Wie bereits in der Erläuterung von PatchCore gelernt, kann eine Aggregation verschiedener Hierarchielevel die Instanzklassifizierungsgenauigkeit verbessern. Dass dies auch tatsächlich der Fall ist, zeigen $j = \{1, 2, 3\}$ und, die auch in PatchCore verwendeten Hierarchieebenen $j = \{2, 3\}$ in 3.5. Es kann aber nicht pauschal davon gesprochen werden, dass eine Aggregation immer zu einer Verbesserung führt. Wir sehen, dass $j = \{1, 2\}$ eine deutlich schlechtere Instanzklassifiziergungsgenauigkeit aufweist, als $j = \{2\}$ alleine.
Ebenfalls gilt es zu beachten, dass der Einbettungsprozess und die NN-Suche in aller Regel mehr Zeit in Anspruch nimmt, verwendet man eine Kombination verschiedener Hierarchielevel. Einmalig betrachten wir in 3.6 die Verwendung von Kombinationen, die $j = 4$ beinhalten. Wie bereits in 3.4 zu sehen, ist $j = 4$ alleine im Vergleich kaum geeignet. Der AUROC kann durch Hinzunahme weiterer Hierarchielevel lediglich marginal verbessert werden. Dies geht einher mit einer erheblichen Laufzeitsteigerung. Wie in PatchCore bereits vermutet, sind die Feature, die aus Layer 4 extrahiert werden, nicht geeignet, um eine präzise Instanzklassifizierung zu ermöglichen, weil sie einen zu hohen Bias hin zur Klassifikationsaufgabe der Objekterkennung

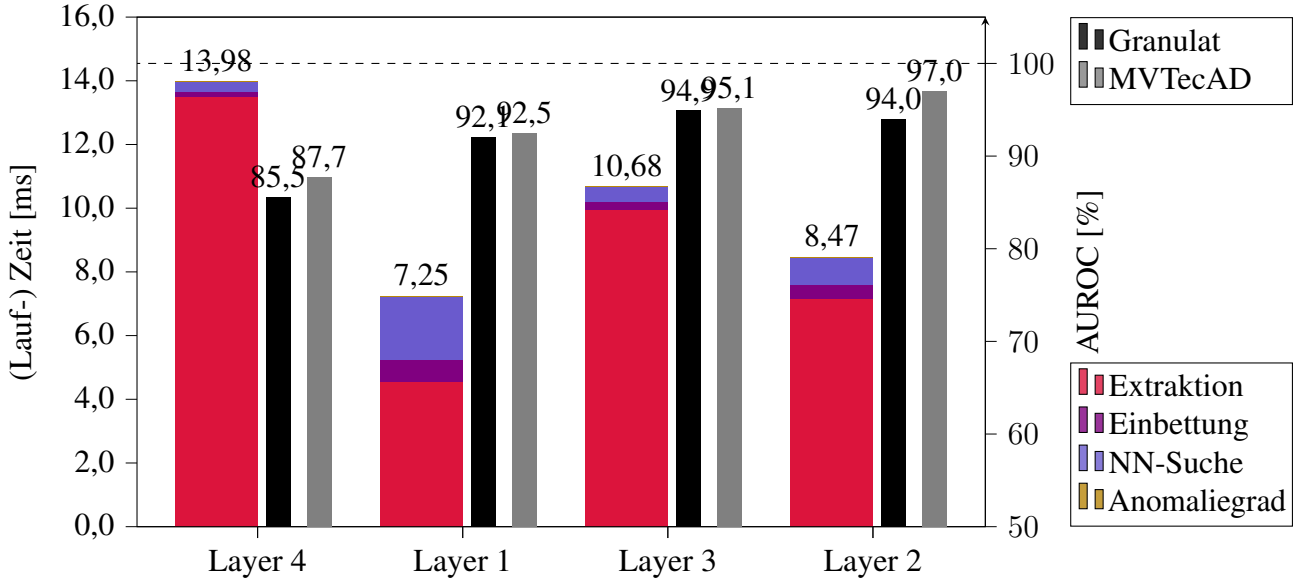


Abbildung 3.4: ResNet 18: Einzelne Hierarchielevel im Vergleich.

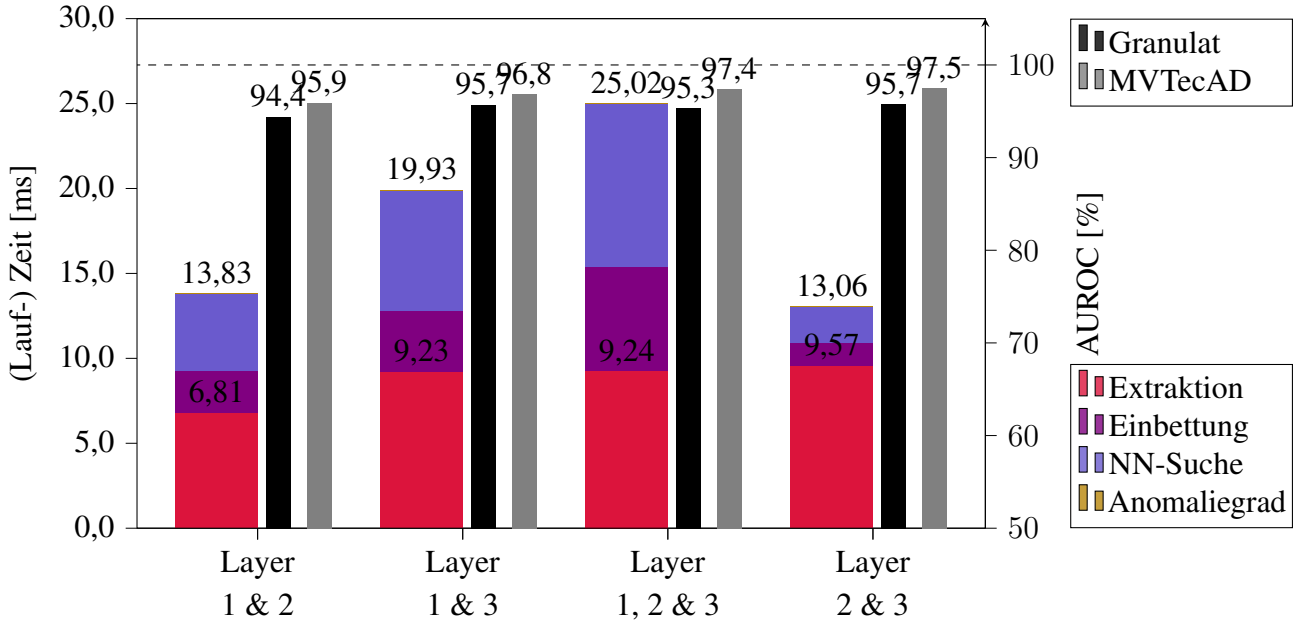


Abbildung 3.5: ResNet 18: Verschiedene Kombinationen an Hierarchieleveln im Vergleich exklusive Layer 4.

aus ImageNet aufweisen. Wie noch zu sehen sein wird, verhalten sich die verschiedenen Hierarchielevel analog im Hinblick auf Laufzeit und Genauigkeit für alternative Netze. Im Folgenden wird dementsprechend auf die Verwendung von $j = 4$ verzichtet.

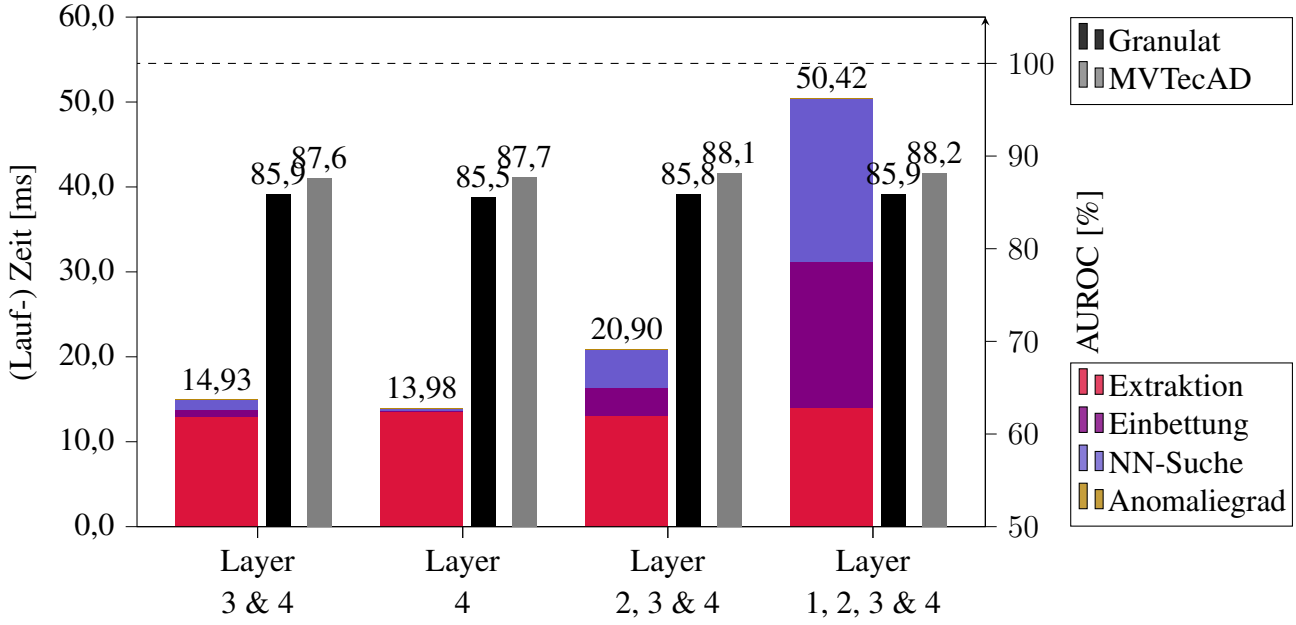


Abbildung 3.6: ResNet 18: Verschiedene Kombinationen an Hierarchieleveln im Vergleich inklusive Layer 4.

ResNet 34 Der nächst größere Kandidat ist das ResNet 34. Analog zu ResNet 18 werden zunächst die einzelnen Hierarchielevel betrachtet.
Es ergibt sich ein zum ResNet 18 sehr ähnliches Bild. Auch hier ist für $j = \{2\}$ die höchste Instanzklassifiziergung festzustellen. Danach folgt $j = \{3\}$ und $j = \{1\}$. Auch hier ist $j = \{4\}$ aufgrund von Laufzeit und Instanzklassifiziergungsgenauigkeit nicht geeignet. Auch die AUROC Werte liegen auf ähnlichem Niveau. Wie zu erwarten ist, sind die Laufzeiten für die Feature Exktraktion höher, als bei ResNet 18. Es gilt aber auch, dass in allen Fällen die Instanzklassifiziergungsgenauigkeit für MVTecAD leicht zunimmt. Durch die gestiegene Tiefe des Netzes können aussagekräftigere Merkmale erzuegt werden, die auch für die Instanzklassifizierung hilfreich sind. Am Ende dieses Abschnittes wird sich eingehender mit dem Vergleich zwischen verschiedenen Backbones beschäftigt.
Auch für die Aggregation verschiedener Hierarchielevel j sind große Parallelen zum bereits besprochenen Fall mit dem ResNet 18 zu erkennen. Ein wesentlicher Unterschied kann aber in der Verbesserung der AUROC Werte bei Verwendung von mehreren Elementen füür j festgestellt werden. Zwar kann im Falle von $j = \{2, 3\}$ eine leichte Steigerung des AUROC festgestellt werden, diese ist aber mit einer absoluten Differenz von $0{,}2\%$ auf MVTecAD sehr gering und geht mit einer beinahen Verdoppelung der Laufzeit einher. Für den Fall $j = \{1, 2, 3\}$ ist eine deutliche Verbesserung auf dem Granulat Datensatz festzustellen. Wie noch zu sehen sein wird, ist es aber nicht zwingend notwendig, Layer1 mit zu inkludieren, um eine guten AUROC für diese Domäne zu erhalten. Einher geht die Hinzunahem von $j = 1$ nämlich auch mit einer deutlichen Erhöhung der Laufzeit.

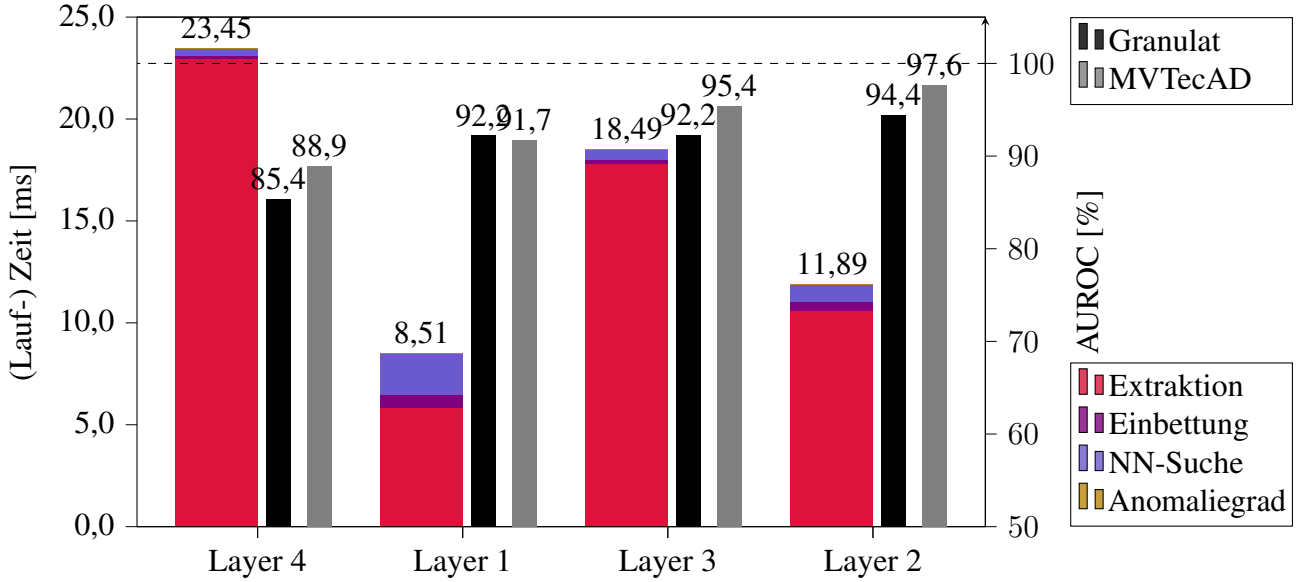


Abbildung 3.7: ResNet 34: Einzelne Hierarchielevel im Vergleich.

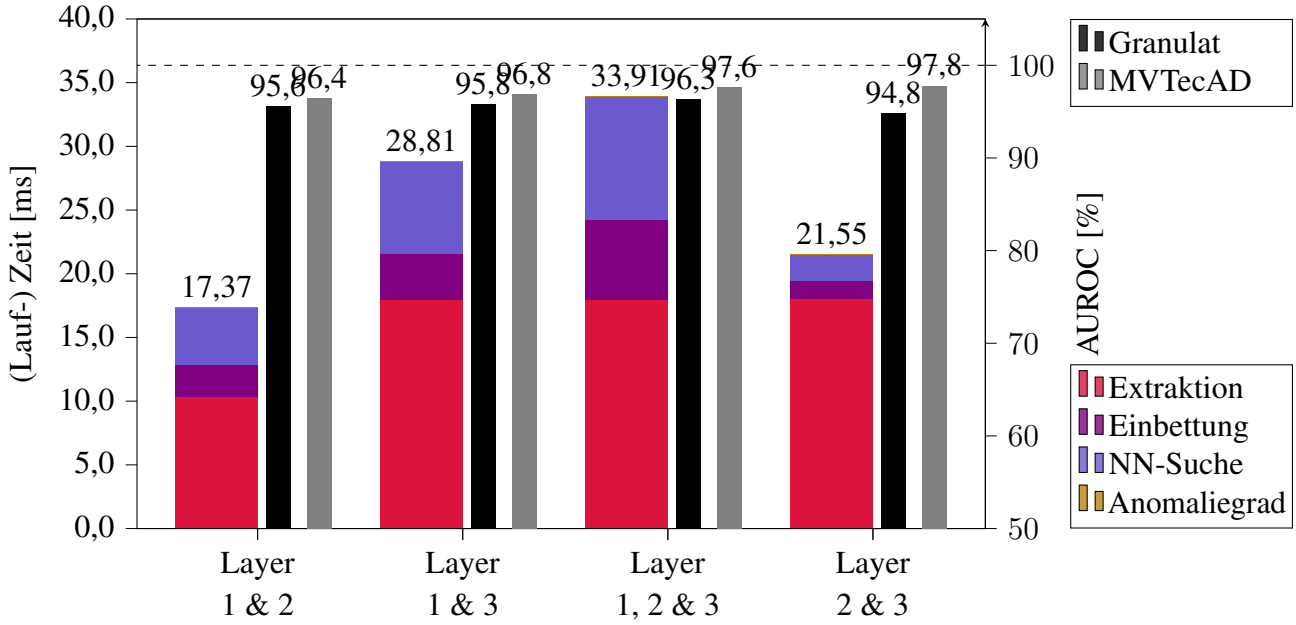


Abbildung 3.8: ResNet 34: Verschiedene Kombinationen an Hierarchieleveln im Vergleich.

Patch Description Network (PDN) - Wissensdistillation mit Wide ResNet 101 und PatchCore Einbettung

Wie in 4 noch ausführlich erläutert wird, kann über Wissensdistillation ein den spezifischen Anforderungen angepasster alternativer Feature Exktraktor erzeugt werden. Hierfür wird ein die Methode PatchCore angewandt, um Trainingsziele zu erzeugen. Ein Wide ResNet 101

wird als Backbone verwendet und der Einbettungsprozess wird mit $d = 384$ durchgeführt. Das Verhalten dieses Patch Feature Exktraktionsprozess von PatchCore wird dann vom PDN imitiert. Weil implizit in der Inferenz des PDNs inkludiert, ist kein expliziter Einbettungsprozess mehr notwendig, lediglich eine Dimensionsanpassung („flatten“ bzw. Auflösen der räumlichen Struktur). Wie wir in 3.9 sehen können, ist das PDN nicht in der Lage die Laufzeit zu reduzieren.

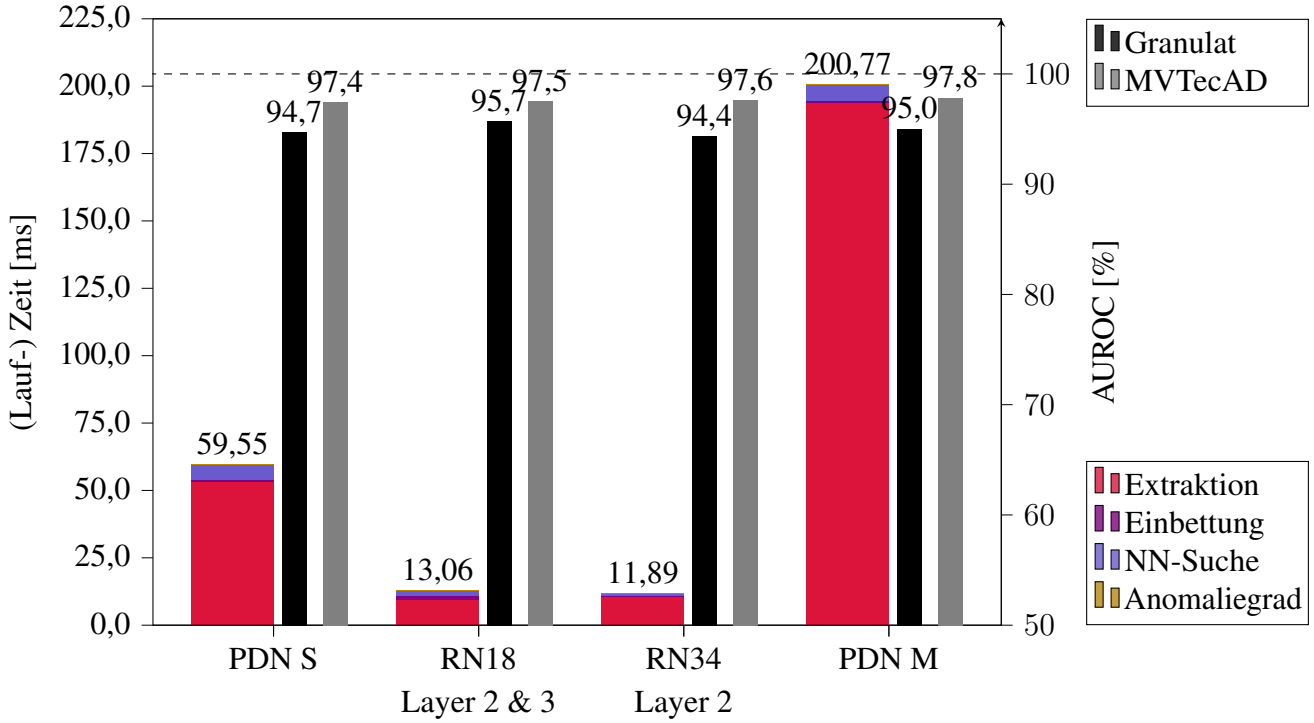


Abbildung 3.9: PDN als Feature Extraktor für PatchCore.

Es ist aber in der Lage, in der Instanzklassifiziergungsgenauigkeit die Leistung auf Augenhöhe mit den ResNet Architekturen zu sein. Das zeigt, dass ein solcher Ansatz grundsätzlich interessant ist.

Obwohl die Anzahl der Parameter für ResNet 18 bis inklusive Layer3 und PDN S mit jeweils etwa 2,7M sehr ähnlich sind, ist die Laufzeit für PDN S deutlich höher. Das liegt vor allem daran, dass bei ResNet Architekturen dirket zu Beginn des „forward pass“ die räumliche Auflösung der Feature Maps deutlich reduziert wird. Schon bei der Eingabe zu Layer1 ist die Kantenlänge um den Faktor 4 reduziert. Zwar wird auch beim PDN, wie in 4 noch ausführlicher besprochen wird, die räumliche Auflösung reduziert, allerdings in einem wesentlich geringeren Maße. Auch die Ausgabe des PDN ist im Verhältnis immer noch recht hochauflösend. Wird ein quadratisches Bild der Kantenlänge 256 Pixel als Eingabe verwendet, so ist die Ausgabe des PDN 64×64 Pixel groß. Das entspricht der Auflösung der Feature Map eines gleich großen Eingabebildes bei einem ResNet 18 nach Layer1 bzw. $j = \{1\}$. Diese Eigenschaft ist sehr nützlich, möchte man eine fein auflösende Segmentierung durchführen, was im Rahmen dieser Arbeit allerdings nur von geringem Interesse ist. Durch eine hohe Auflösung der Feature Map ergeben sich auch eine höhere Anzahl an Berechnungen. (TODO –> Erklärung) Das ist der Grund, warum die

Laufzeit für PDN S höher ist, als für ResNet 18. Diese Berechnungen lassen sich zwar auf einer GPU derart parallelisieren, dass die Laufzeitunterschiede zwischen PDN S und ResNet 18 durchlich kleiner ausfallen würden, wie in **??** noch einmal ausführlicher gezeigt wird. Jedoch ist die Verwendung einer GPU in dieser Arbeit nicht vorgesehen.
Aufgrund der nicht konkurrenzfähigen Laufzeit der PDNs, vor allem der M Variante, wird im weiteren Verlauf die Verwendung von PDNs im Rahmen der Methode PatchCore nicht weiter verfolgt.

ConvNexts

Wie bereits in der Einleitung zu diesem Abschnitt erläutert, wird hier eine weitere, relativ neue Architektur untersucht. In der jüngeren Entwicklung im Bereich der Künstlichen Intelligenz sind, wie bereits thematisiert, Transformer basierte Architekturen sehr erfolgreich und verdrängen CNNs in vielen Anwendungen, wenn es um die Generalisierungsfähigkeit und Präzision geht. Für die Bildklassifizierung gilt das im Grunde ebenso. Liu et al. [**?**] erreichten dennoch mit ihrer CNN-Architektur-Familie der „ConvNexts“ eine Klassifizierungsgenauigkeit, die in einigen Fällen sogar Transformer basierte Architekturen übertrifft. In der der Veröffentlichung entnommenen Abbildung 3.10 ist zu sehen, dass dabei nicht nur die Auccuracy gängiger Transformer Modelle, wie DeiT oder den Swin Transformern [**?**], übertroffen wird, sondern auch der Rechenaufwand (FLOPS) vergleichbar ist. Letzteres ist durch die Größe der Kreise dargestellt. Insbesondere für die kleineren Architekturen gilt das, welche auch hier vorrangig behanderlt werden.
Es stellt sich im Rahmen dieser Arbeit nun die Frage, ob sich diese Genauigkeit, die, wie ebenfalls in 3.10 zu sehen ist ResNets deutlich übertrifft, auch auf aussagekräftigere Feature Maps übertragen lässt. Dies könnte sich schließlich in einer Verbesserung der Instanzklassifiziergungsgenauigkeit auf MVTecAD ausdrücken. Um diese These zu testen, wird schließlich das gleiche Prozedere wie für die ResNet Architekturen durchgeführt.
Die Ergebnisse hierzu sind in 3.11 und 3.12 zu sehen. Es handelt sich hierbei um die „S “ bzw. die „XS“ Varianten. Diese entsprechen den kleinsten Architekturen, die in der Veröffentlichung [**?**] vorgestellt werden und somit den am schnellsten ausführbaren. Sämtliche größere Architekturen kommen im Rahmen dieser Arbeit nicht in Frage, weil sie die Laufzeit deutlich steigern, aber keine signifikante Verbesserung der Instanzklassifiziergungsgenauigkeit auf MVTecAD bieten. Um dies zu belegen, wurden unter allen Architekturen die vier im Hinblick auf den AUROC auf MVTecAD besten Kombinationen an Hierarchieleveln ausgewählt und in 3.13 dargestellt.

Vergleicht man diese Ergebnisse mit denen der ResNet Architekturen, so ist zu erkennen, dass die ConvNexts in der Instanzklassifiziergungsgenauigkeit auf MVTecAD schlechter abschneidet. Die AUROC Werte kommen auch für die günstigsten Kombinationen nicht an die der ResNets heran. Auch die Laufzeut ist in den meisten Fällen höher, als bei den ResNets. Es ist also festzuhalten, dass die ConvNexts für die Anwendung in PatchCore schlechter geeignet sind, als die bislang entworfenen Methoden mit einem ResNet 18 oder ResNet 34. Im Folgenden wird dementsprechend auf eine weitere Untersuchung der ConvNexts verzichtet.

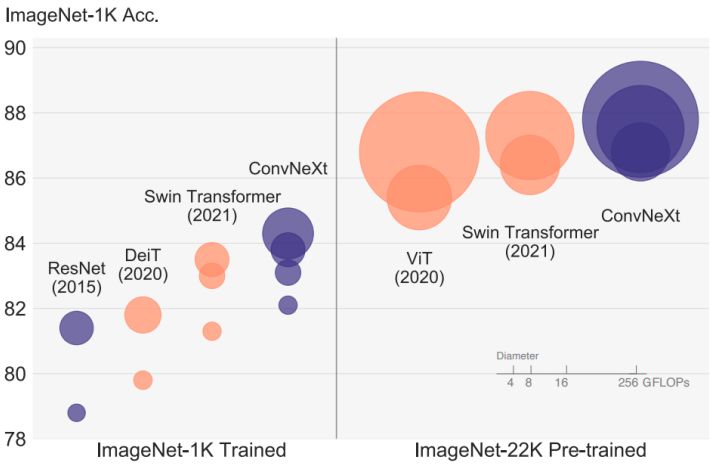


Abbildung 3.10: ConvNexts als Feature Konkurrenz zu Transformern.

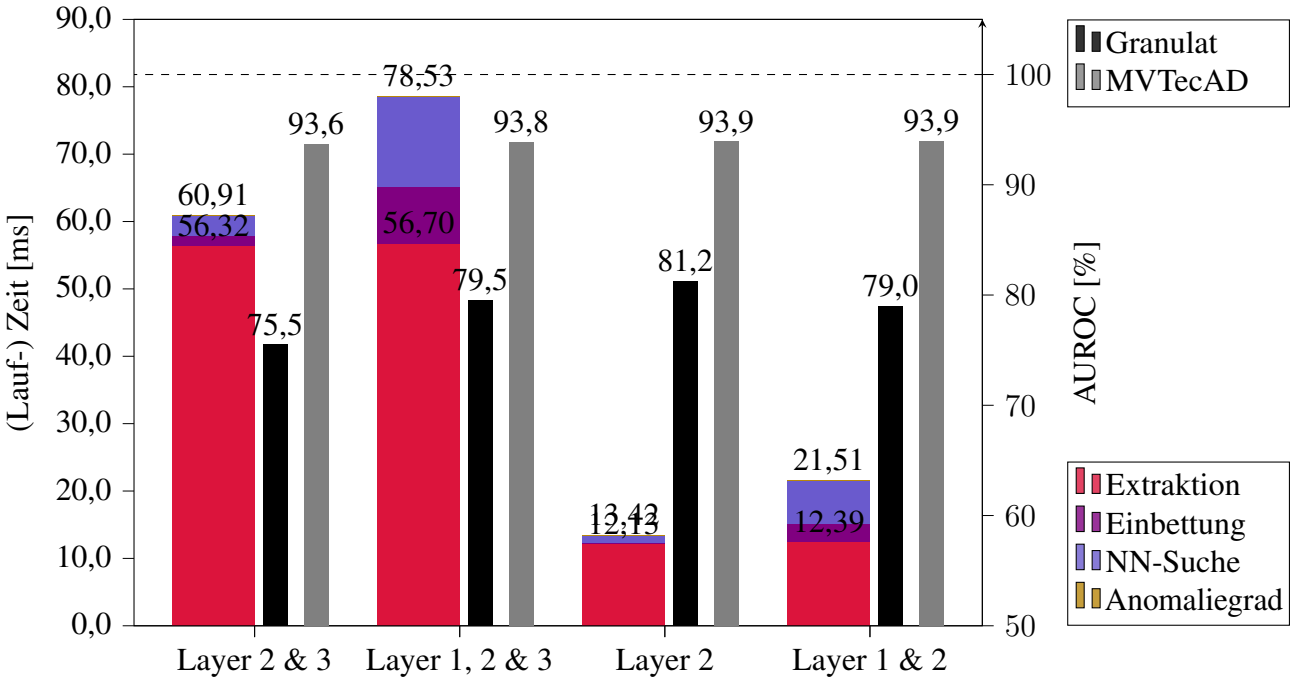


Abbildung 3.11: Die vier besten Hierarchielevelkombinationen eines ConvNext S.

Über die genauen Gründe, warum dies der Fall ist, kann an dieser Stelle nur spekuliert werden. Geht man dazu ein Schritt zurück und schaut sich an, wie die Verbesserungen der ConvNexts gegenüber den ResNets in der Bildklassifizierung erreicht werden, so ist zu erkennen, dass als Vorbild für diese Architetktur-Familie an vielen Stellen Transformer dienen. Ein Baustein von vielen ist dabei, dass Transformer den Datenfluss (bzw. „Feature Maps“) seltener und in weniger Ganzheitlich normalisiert werden. Dies wird auch für die ConvNexts übernommen und steigert die Bildklassifizierungsgenauigkeit. Es werden die in ResNet pro Block dreimal vorkommenden „Batch Normalization Layers“ durch nur noch eine ersetzt. In einem weitere Schritt werden diese

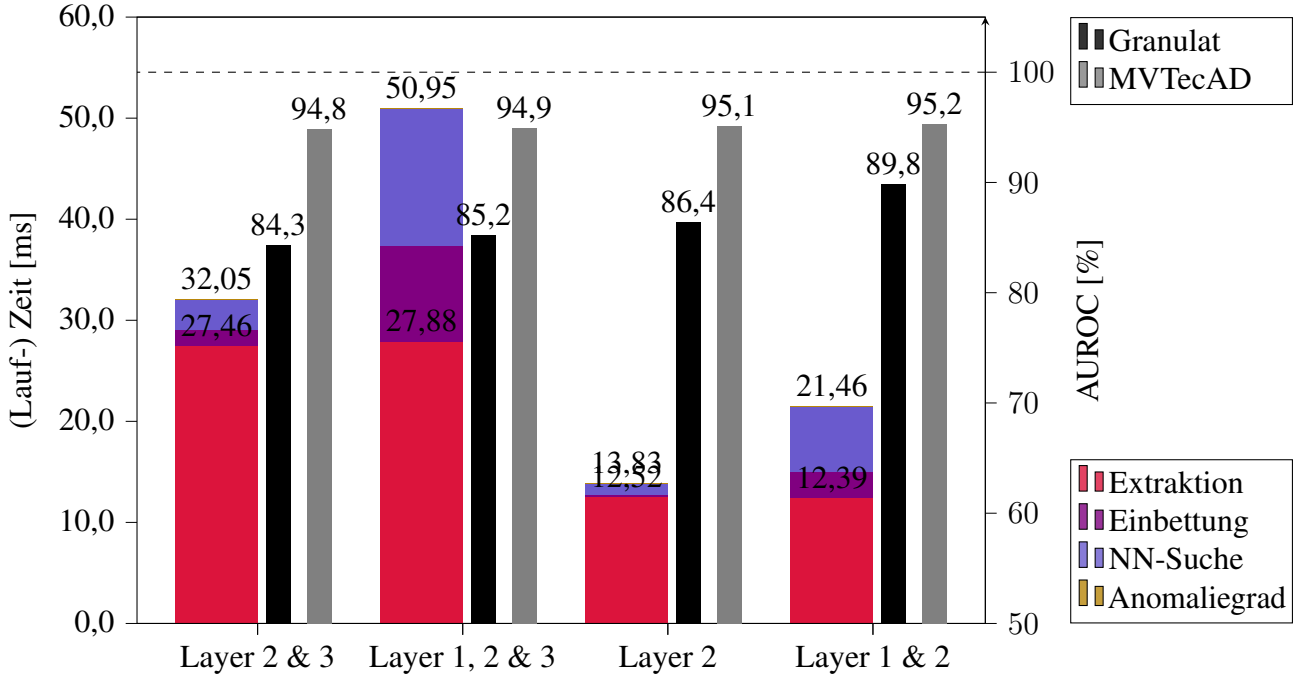


Abbildung 3.12: Die vier besten Hierarchielevelkombinationen eines ConvNext XS.

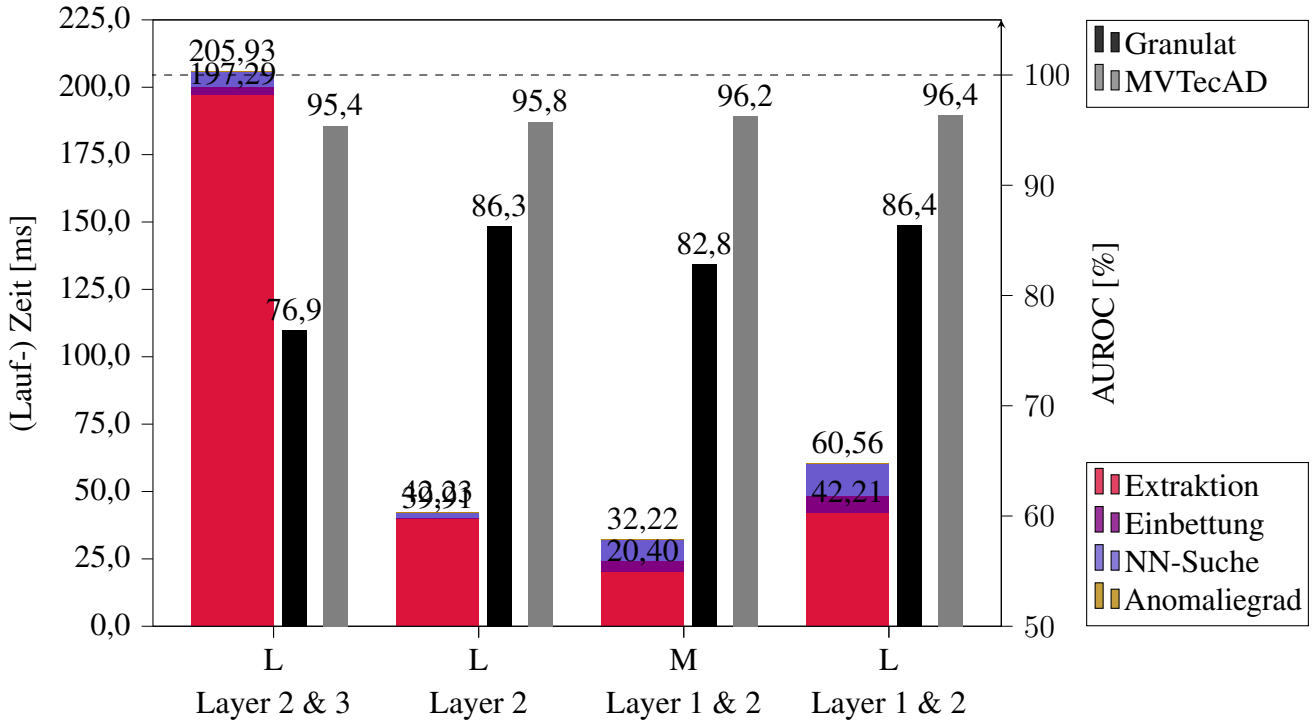


Abbildung 3.13: Die vier besten Hierarchielevelkombinationen aller ConvNexts.

durch weniger stark regularisierende „Layer Normalizations“ ersetzt. Der Datenfluss in einem ResNet ist also deutlich stärker regularisiert und normalisiert als in einem ConvNext. Dieser Datenfluss ist im Falle der Feature Exktraktion aber die entscheidende Größe und es kann angenommen werden, dass die fehlende Regularisierung zu einer breiteren, weniger aussagekräftigen Verteilung der Feature Maps führt und damit in letzter Konsequenz zu einer weniger robusten Anomaliedetekion. Eine mögliche Abhilfe könnte sein, durch spezielle Aktivierungsfunktionen (z.B SoftMin) oder einer nachträglichen kanalweisen Normalisierung diese Regularisierung „nachzuholen“. Dies ist aber aufgrund des ohnehin schon sehr großen Suchraum in dieser Arbeit und der schließlich relativ deutlich schlechteren Ausgangsperformance nicht für eine weitere Untersuchung vorgesehen.

Fazit

Zunächst sollten festgehalten werden, dass durch sowohl ResNet 18 als auch ResNet 34 die Laufzeit der Feature Extraktion sich deutlich reduziert. Wie in 3.2 zu sehen ist, nimmt die Feature Extrakion mit einem Wide ResNet 50 etwa Für die gleichen Hierarchielevel benötigt ein ResNet 34 als Backbone in etwaDas sind enorme Verbesserungen, die sich analog auch auf dem RaspberryPi 4B zeigen würden, wie später noch zu sehen sein wird.
Dies geht einher mit einer Verschlechterung der Instanzklassifiziergungsgenauigkeit auf MVTecAD. Die Referenz liegt bei einem AUROC von Das wir von keiner Konfiguration mit ResNet 18 oder 34 erreicht. Den höchsten AUROC wird mit einem ResNet 34 und $j = \{2, 3\}$ erzielt und liegt bei $97{,}8\%$. Auf dem eigenen Granulat Datensatz zeigt sich hingegen durchweg eine recht deutliche Verbesserung im AUROC. Die kompaktereen Backbones sind also für diese Domäne sogar besser geeignet.
Für beide Backbones lässt sich gleichermaßen feststellen, dass $j = \{2, 3\}$ eine gute Wahl ist. Ebenfalls für eine weitere Betrachtung äußerst interessant ist $j = \{2\}$. Es erreicht fast die gleiche AUROC wie $j = \{2, 3\}$, benötigt aber deutlich weniger Zeit für die Feature Extraktion. AUßerdem wird der sich anschließende Einbettungsprozess und die NN-Suche beschleunigt.
Für die weitere Untersuchung werden deshalb die folgenden Backbones verwendet:

- ResNet 18 mit $j = \{2\}$, 97,0% AOROC MVTecAD, 94,0% AUROC Granulat, 8,5ms je Bild
- ResNet 34 mit $j = \{2\}$, 97,6% AOROC MVTecAD, 94,4% AUROC Granulat, 11,9ms je Bild
- ResNet 18 mit $j = \{2, 3\}$, 97,5% AOROC MVTecAD, 95,7% AUROC Granulat, 13,0ms je Bild
- ResNet 34 mit $j = \{2, 3\}$, 97,8% AOROC MVTecAD, 94,8% AUROC Granulat, 21,5ms je Bild

Die Angaben zu den Laufzeiten beziehen sich, wie alle Angaben diesbezüglich, auf eine Ausführung auf der Desktop CPU.

Einbettungsprozess

Es kann in 3.3 bereits gesehen werden, dass ein wesentlicher Teil der Laufzeit für den Prozess der Einbettung der bereits extrahierten Feature Maps verwendet wird und das es dafür Abhilfe gibt. Während die Varianten (3) und (4) knapp $40\mathrm{ms}$ benötigen, um die Feature einzubetten, ist die Variante, die im Folgenden vorgestellt wird, um ein Vielfaches schneller. Die Laufzeit hängt dabei zwar von verschiedenen Faktoren ab, wie zum Beispiel den gewählten Hierarchieleveln j, aber auch von der Anzahl der Ausgangskanäle.

Funktionsweise des angepassten Einbettungsprozesses Der Einbetungsprozess, der in dieser Arbeit verwendet wird, ist deutlich weniger komplex, als der in PatchCore vorgestellte. (3.2.1) Die Ausgangssituation ist dennoch die gleiche. Es liegen die Feature Maps des Backbones ϕ für ein Bild x_i und Hierarchieebene j vor:

$$\phi_j(x_i) \in \mathbb{R}^{h_j \times w_j \times d_j}$$

Die Auflösung der Feauture Map hängt dabei von der Dimension des Eingangsbildes, vom verwendeten Backbone und vom Hierarchielevel j ab. Für die hier verwendeten Backbones ResNet 18 und 34 ergeben sich für ein Eingangsbild $x_i \in \mathbb{R}^{256\times256\times3}$ $\phi_2(x_i) \in \mathbb{R}^{32\times32\times128}$ und $\phi_3(x_i) \in \mathbb{R}^{16\times16\times256}$.
Es wird zunächst jede Feature Map für sich betrachtet, also $j = \{2\}$ und $j = \{3\}$. Auf eine Feature Map wird in einem ersten Schritt ein Pooling angewandt:

$$\phi_{pooled,i,j} = \mathtt{pool}_{\mathtt{k},\mathtt{s},\mathtt{p}}(\phi_j(x_i))$$

Die Funktion `pool` ist entweder ein Max-Pooling oder ein Average-Pooling mit den Parametern `k` für die Größe des Kernels, `s` für die Schrittweite und `p` für die Padding Größe. In dieser Arbeit wird ausschließlich Zero-Padding verwendet. Auch zur Art des Paddings hätten Versuche angestellt werden können. Da der Suchraum allerdings bereits sehr groß ist und keine signifikanten Verbesserungen durch andere Padding-Methoden zu erwarten sind, wird auf diese Versuche verzichtet.
Für die Dimensionen ergibt sich unabhängig von der Art des Poolings folgende Formeln:

$$w_{j,pooled} = \left\lfloor \frac{w_j - \mathtt{k} + 2\mathtt{p}}{\mathtt{s}} \right\rfloor + 1$$

$$h_{j,pooled} = \left\lfloor \frac{h_j - \mathtt{k} + 2\mathtt{p}}{\mathtt{s}} \right\rfloor + 1$$

$$c_{j,pooled} = c_j$$

Im Unterschied zur PatchCore Methode findet ein Pooling also zunächst nur in der räumlichen Dimension statt. Die Anzahl der Kanäle bleibt also gleich. Für eine optimale Wahl der Parameter $\mathtt{k}$, $\mathtt{s}$ und $\mathtt{p}$ wurde ein Suchraum durchlaufen. Sinnvolle Werte sind für $\mathtt{k}$ sind dabei die ungeraden Zahlen $3, 5, 7, 9$. Für $\mathtt{s}$ sind die Werte $1, 2, 3$ sinnvoll und für $\mathtt{p}$ die Werte $0, 1, 2$. Größere Werte versprechen in Anbetracht der folgenden Ergebnisse, aber vor allem der Größe der Feature Map und des korresponderenden rezeptiven Feldes keine Verbesserung.

Versuche zur Wahl der Pooling Parameter Für die Versuche wurden dementsprechend alle Kombinationen getestet. Das geschah für die Hierarchieebenen $j = \{2\}$ und $j = \{3\}$, sowie für die beiden Backbones ResNet 18 und ResNet 34. Die beiden Backbones verhalten sich dabei, bis auf die bereits bekannten Unterschiede in AUROC und Laufzeit, sehr ähnlich. Deswegen wurde sich auf die Präsentation der Ergebnisse für ResNet 18 mit $j = \{3\}$ und ResNet 34 mit $j = \{2\}$ beschränkt.
Betrachtet man nun die Ergebnisse aus 3.14 und 3.15, so ist zu erkennen, dass ein Average-Pooling mit $\mathtt{k} = 3$, $\mathtt{s} = 1$ und $\mathtt{p} = 1$ in allen Fällen die beste Variante ist. Weiter lässt sich erkennen, dass ein Pooling insgesamt sinnvoll ist, denn insbesondere bei der Feature Map mit $j = \{2\}$ zeigt sich ein recht deutlicher, positiver Effekt bezüglich der AUROC, vergleicht man die Pooling Varianten mit der Identitätsabbildung.
Größere Kernel-Größen führen immer zu einer Verschlechterung der AUROC. Das gilt bereits für $\mathtt{k} = 5$, insbesondere aber für alle größeren Werte. Eine erhöhte Schrittweite hat zwar ebenfalls einen negativen Effekt, dieser ist aber deutlich geringer. Das Padding spielt insgesamt eine ungeordnete Rolle und hat werder auf die AUROC, noch auf die Laufzeit einen signifikanten Einfluss.
Ein positiver Einfluss auf die Laufzeit ist insbesondere für eine erhöhte Schrittwete $\mathtt{s}$ zu erkennen. Die Laufzeit für die Einbettung und die NN-Suche ist gegenüber der Feature Exktraktion in diesem Fall fast vernachlässigbar klein. Betrachten wir die Ausgangsauflösung nach oben angeführten Formeln wird schnell klar, warum das der Fall ist. Während sich für $\mathtt{k} = 3$, $\mathtt{s} = 1$ und $\mathtt{p} = 1$ die Auflösung der Feature Map nicht verändert und somit für $j = \{2\}$ 1024 Patch Feature Vektoren entstehen, ist mit $\mathtt{s} = 3$ diese Anzahl bei lediglich $6 \times 6 = 36$.
Spannend ist, dass ein Maximum-Pooling stets dem Average-Pooling unterlegen ist. Man könnte vermuten, dass das Gegenteil der Fall ist, weil man bei der Anomaliedetektion schließlich gerade an Extremstellen interessiert ist. Das Ergebnis legt aber nahe, dass die Struktur eines Patch Feature Vektors entscheidend ist und nicht etwa einzelne Einträge und dessen Maxima. Diese Struktur wird durch ein Average-Pooling besser erhalten.
Allgemein lässt sich festhalten, dass das Pooling keinen nennenswerten Einfluss auf die Laufzeit hat. Abgesehen von umgebungsbedingten Schwankungen lässt sich keine signifikante Änderung für die Laufzeit der Einbettung feststellen, die nicht auf die Auflösung der resultierenden, gepoolten Feature Map zurückzuführen ist.

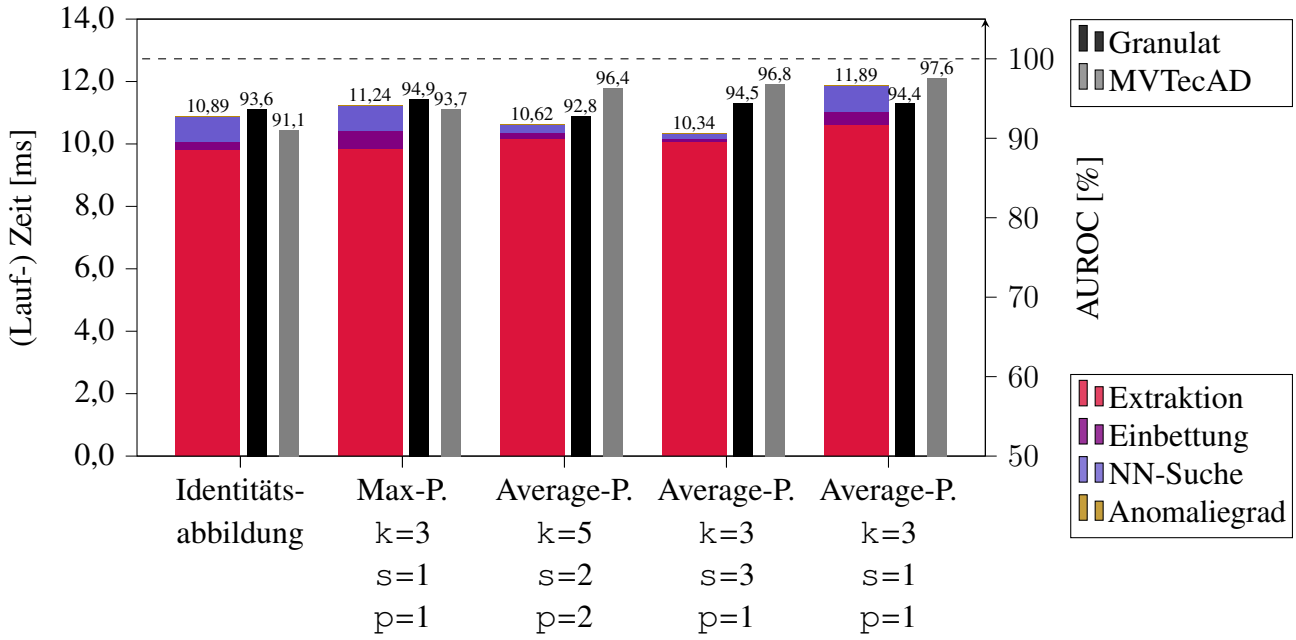


Abbildung 3.14: Exemplarische Ergebnisse für verschiedene Pooling Varianten im Einbettungsprozess anhand von ResNet 34 und $j = \{2\}$.

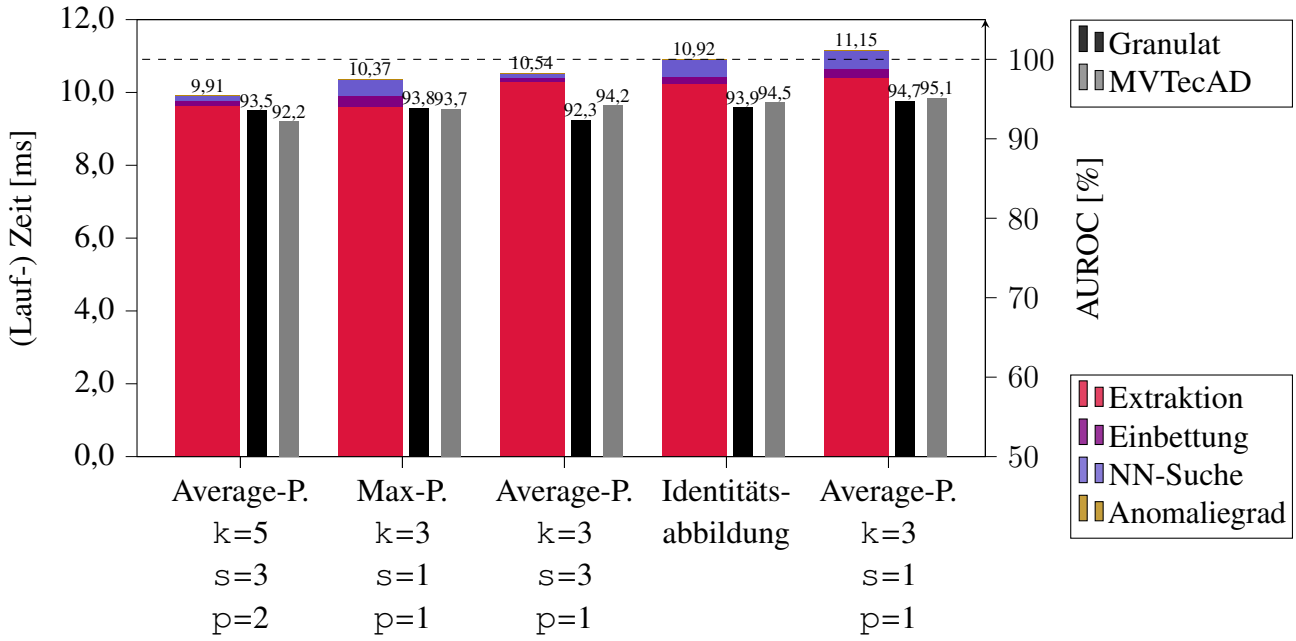


Abbildung 3.15: Exemplarische Ergebnisse für verschiedene Pooling Varianten im Einbettungsprozess anhand von ResNet 18 und $j = \{3\}$.

Kombination von Pooling Operationen Ein für den Autor einleuchtendes Konzept ist die Kombination von Pooling Operationen. Wie wir in beinahe allen in diesem Kapitel vorkommenden Abbildungen sehen können, ist die Feature Exktraktion der laufzeittechnisch kritischste Schritt. Das Pooling an sich ist mit Blick auf die Laufzeit hingegen vernachlässigbar. Es kann also verschiedene Pooling Operationen ausgeführt werden und zu einem Feature Vektor zusammengefasst werden, ohne, dass ein großer Einfluss auf die Laufzeit zu erwarten ist.
So kann versucht werden, ein Average Pooling mit einem Max Pooling zu kombinieren. Die Anzahl der Kanäle verdoppelt sich dadurch. Naheligend ist es, mit gleichen Parametern für das Average Pooling und das Max Pooling zu arbeiten. Das sorgt für identische Dimensionen beider gepoolten Feature Maps und erleichtert die Konkatenation.

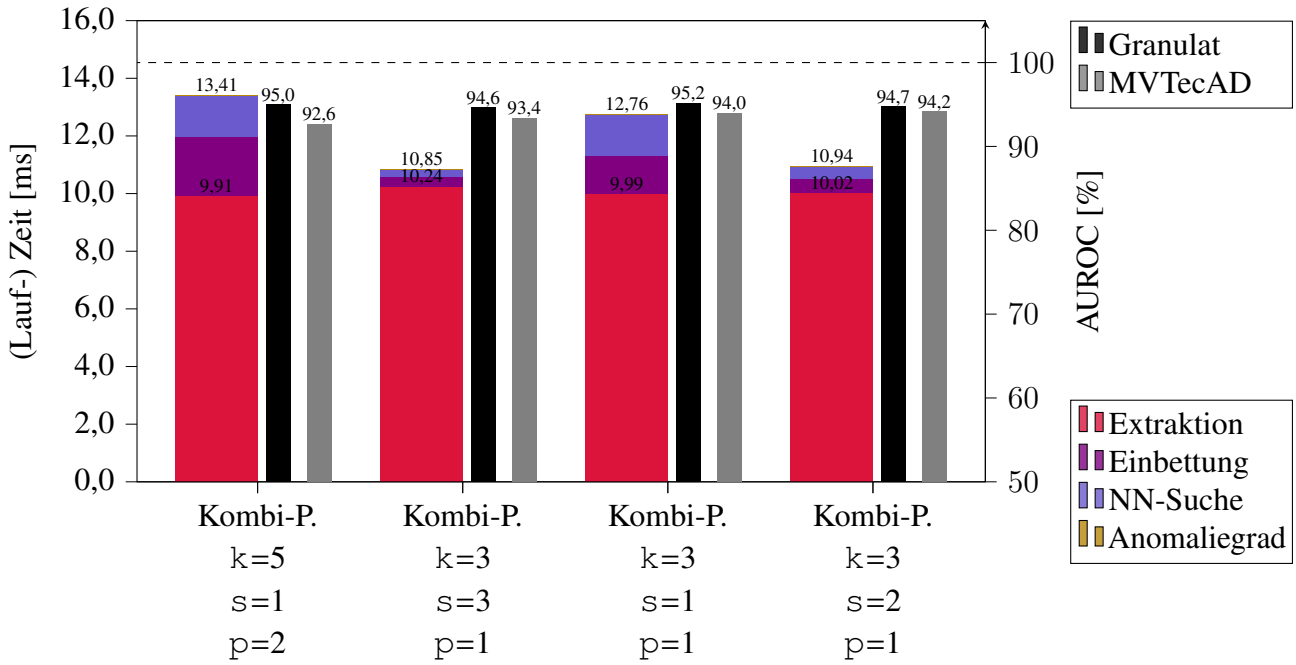


Abbildung 3.16: Exemplarische Ergebnisse für eine Kombination von Max- und Average-Pooling anhand von ResNet 34 und $j = \{2\}$.

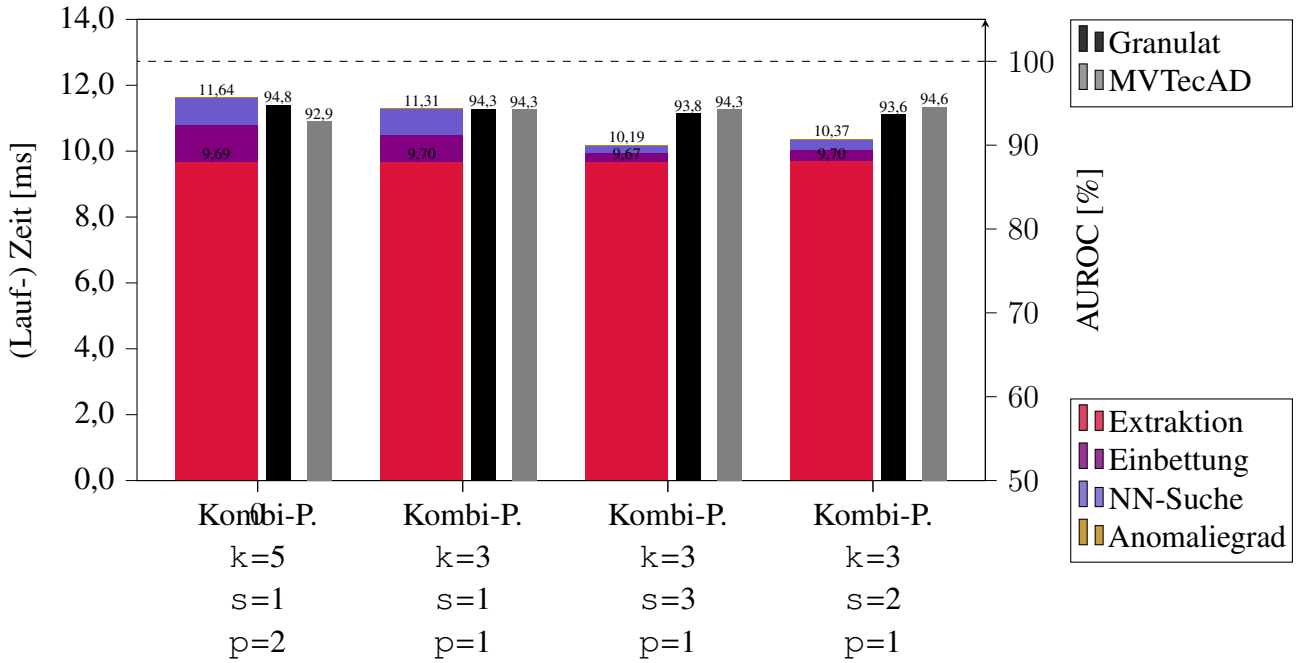


Abbildung 3.17: Exemplarische Ergebnisse für eine Kombination von Max- und Average-Pooling anhand von ResNet 18 und $j = \{3\}$.

Kapitel 4

EfficientAD

4.1 Einleitung

Abgeschlossen: 30.10. v1

Das im Zusammenhang mit dem hier verwendeten Datensatz prominent vertretene Unternehmen MVTec GmbH aus München entwickelte die Methode **EfficientAD** und veröffentlichte diese am 25. März 2023. Es handelt sich also um eine junge Methode, die sich durch geringe Laufzeiten auf einer GPU und gleichzeitig hoher Genauigkeit auszeichnet. Dies ist in 4.1 dargestellt. Rot markiert sind hierbei die Methoden, die in dieser Arbeit implementiert wurden. Mit bis zu 99,8%

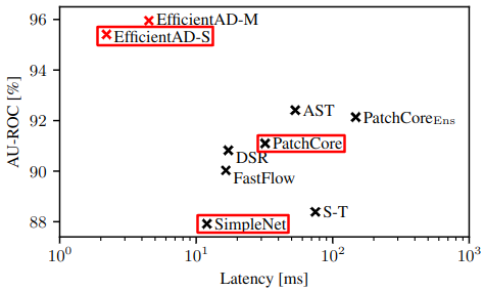


Abbildung 4.1: Übersicht über AUROC und Laufzeit verschiedener Methoden ausgeführt auf Nvidia RTX A6000 GPU. [2]

Instanzklassifiziergungsgenauigkeit auf dem Datensatz MvTecAD ist es die Methode mit der höchsten Genauigkeit auf diesem Datensatz. [8]
Es kombiniert dabei verschiedene Methoden, die in anderen Veröffentlichungen schon erfolgreich eingesetzt wurden. So wird ein zweiteiliger „Student-Teacher“-Ansatz verwendet. Zum einen erkennen wir eine modifizierte Variante der Exktraktion von Featuren durch ein auf ImageNet implizit vortrainiertem Netz (Wissens-Distillation). Der andere Teil setzt auf einen Student-Teacher-Ansatz auf Basis eines Autoencoder, der die Aufgabe hat, nominale Merkmale zu rekonstruieren. Beide Konzepte werden in diesem Kapitel noch genauer erläutert.
Die Implementierung erfolgt dabei beinahe ausschließlich mithilfe von CNNs. Nur die finale Bestimmung des Anomalie-Scores und der Anomaliekarte, die, wie gezeigt werden wird, kaum laufzeitkritisch ist, wird ohne die Verwendung von CNNs realisiert. Da moderne GPUs äußerst schnelle Inferenz von CNNs ermöglichen, gehört diese Methode zu den am schnellsten ausführbaren Methoden im Bereich der Unüberwachten Anomaliedetekion.[2][8]

Wie bereits gesehen werden konnte, lassen sich solche Aussagen über die Laufzeit nicht einfach übertragen, liegt keine GPU vor, wie in dieser Arbeit.

4.2 Grundlage - Uninformed Students

Abgeschlossen: 02.11. v1
Die Veröffentlichung „Uninformed Students: Student-Teacher Anomaly Detection with Discriminative Latent Embeddings“, welche am 18. März 2020 vorgestellt wurde, präsentiert ein zentrales Konzept, welches auch in EfficientAD verwendet wird. Die ebenfalls von MVTec GmbH entwickelte Methode setzt somit auch auf eine Student-Teacher-Architektur.

4.2.1 Funktionsweise

In diesem Bereich wird genauer auf die Funktionsweise von „Uninformed Students“ eingegangen. Analog zu den bisher beschriebenen Methoden, wird zunächst der aus nominalen Bilder bestehende Trainingsdatensatz als $\mathcal{X}_{train} = x_1, x_2, ..., x_n$ ($\forall x \in \mathcal{X}_{train} : y_x = 0$) definiert. Das Ziel ist es, ein Ensemble an „Studenten“ S_i zu erzuegen, die später in der Lage sind, Anomalien eines anomalen Testbildes $x_i \in \mathcal{X}_{test}$ und $y_x = 1$ mit $\forall x \in \mathcal{X}_{test} : y_x = \{0, 1\}$ zu erkennen. Um eine solche Aussage zu treffen, wird die Abweichung für ein Testbild x_i von der Ausgabe der Studenten S_i und der des „Lehrers“ (Teacher) T bestimmt. Große Abweichungen deuten dabei auf eine Anomalie hin. Der Teacher T ist dabei ein CNN, welches auf einem großen Datensatz, wie ImageNet, vortrainiert wurde. Es kommen dabei keine ResNets diekt zum Einsatz, sondern einfache CNNs, die von den Autoren selbst entwickelt wurden. Die Architektur von Teacher T und Studenten S_i ist dabei identisch.

Lernen von lokalen Patch Deskriptoren

In diesem Abschnitt wird sich damit beschäftigt, wie ein Teacher T in die Lage versetzt wird, deskriptive und lokal aufgelöste Merkmale zu extrahieren. $\hat{T}$ kann für beliebige $p \in \mathbb{N}$ aus einem Bild oder Bildausschnitt $\mathbf{p} \in \mathbb{R}^{p \times p \times C}$ einen eindimensionalen Feature Vektor erzeugen. Drei verschiedene Wege des Lernens des Teachers werden im Folgenden beschrieben.

Wissens-Distillation Wie bereits ausführlich in 2.4.3 und in den vorangegangenen Methoden beschrieben, dienen CNNs, die auf großen Datensätzen vortrainiert wurden, als Feature-Extraktoren für aussagekräftige und kompakte Merkmalsextraktoren. Um ein leichtgewichteren Featre Extraktor zu erhalten, wird das CNN $\hat{T}$ trainiert um das Verhalten eines großen, auf ImageNet vortrainierten CNNs ϕ zu imitieren. Es werden dazu Bilder aus ImageNet auf die Größe $p \times p$ ausgeschnitten und dienen als Eingangsgröße. Das Label könnte nun direkt aus

einem Einbettungsprozess wie bei 3.2.1 erzeugt werden, es ergeben sich aber Probleme durch unterschiedliche Größen der jeweiligen Ausgaben. Deshalb wird zusätzliche ein einfaches vollvernetztes neuronales Netz (MLP) eingesetzt, dass die Ausgaben des Netzwerkes ϕ auf die gewünschte Größe d abbildet. Folgende Verlustfunktion wird verwendet, um das Netzwerk T zu trainieren:

$$\mathcal{L}_{KD} = \left\| D(\hat{T}(\mathbf{p})) - \phi(\mathbf{p}) \right\|_2^2$$

$\hat{T}$ ist jedoch auf eine Eingabe von $p \times p$ Pixeln beschränkt, soll ein eindimensionaler Feature Vektor erzeugt werden. Das Netzwerk $\hat{T}$ muss dementsprechend noch in die Lage versetzt werden, mit größeren Bildern umzugehen. Die Autoren dieser Veröffentlichung gehen dabei nach [1] vor und erhalten so einen effizienten Feature Extraktor T. Für $\hat{T}$ werden drei verschiedene Architekturen verwendet, die sich in ihrer Komplexität und der Größe des rezeptiven Feldes ($p \in \{17, 33, 65\}$) unterscheiden. Für weitere Details wird an dieser Stelle auf die Veröffentlichung [1] verwiesen, weil es hier vor allem um das Konzept und dessen Erläuterung gehen soll.
Das für die Wissens-Distillation verwendete Netzwerk ϕ ist ein ResNet-18, welches auf ImageNet vortrainiert wurde. Es wird dabei der 1D-Feature-Vektor aus der letzten Schicht des ResNets („flatten“ in 2.4) verwendet. Dieser 512 Einträge fassender 1D-Vektor wird mithilfe von D auf $d = 128$ Einträge komprimiert. Es sei angemerkt, dass es sich hierbei um die Werte aus der Originalveröffentlichung handelt und andere Wertekombinationen ebenfalls möglich sind.

Metrisches Lernen Beim metrischen Lernen wird zunächst ein Triplet aus Patches $(\mathbf{p}, \mathbf{p}^+, \mathbf{p}^-)$ erzeugt. Das Ausgangspatch $\mathbf{p}$ wird dabei durch einen zufällig ausgeschnittenen Bildausschnitt aus einem Bild aus dem Trainingsdatensatz erzeugt. $\mathbf{p}^+$ ist eine mit Gauß'schen Rauschen und veränderte Helligkeit augmentierte Version von $\mathbf{p}$. Bei $\mathbf{p}^-$ handelt es sich um einen Bildausschnitt aus einem anderen Bild aus dem Trainingsdatensatz. Die Verlustfunktion für das metrische Lernen ist dann gegeben durch:

$$\mathcal{L}_M = \max\left\{0, \delta + \delta^+ - \delta^-\right\}$$

$$\delta^+ = \left\| \hat{T}(\mathbf{p}) - \hat{T}(\mathbf{p}^+) \right\|_2^2$$

$$\delta^- = \min\left\{ \left\| \hat{T}(\mathbf{p}) - \hat{T}(\mathbf{p}^-) \right\|_2^2, \left\| \hat{T}(\mathbf{p}^+) - \hat{T}(\mathbf{p}^-) \right\|_2^2 \right\}$$

Dabei bezeichnet $\delta > 0$ einen Randparameter, der einen Hyperparameter bzgl. des Trainings darstellt. Minimiert man diese Verlustfunktion, so wird das Netzwerk $\hat{T}$ in die Lage versetzt, ähnliche Patches in einem Feature Raum nahe beieinander zu platzieren und die distinkten Patches weiter voneinander zu entfernen. Es wird also die diskriminative Fähigkeit des Netzwerkes $\hat{T}$ gestärkt.

Kompaktheit der Merkmale Um eine kompakte und möglichst wenige redundante Representation der Merkmale zu erhalten, wird ein weitere Verlustfunktion eingeführt.

$$\mathcal{L}_C\left(\hat{T}\right) = \sum_{i \neq j} c_{ij}$$

Die skalaren Parameter c_{ij} bezeichnen dabei die Einträge der Korrelationsmatrix über alle $\hat{T}(\mathbf{p})$ für sämtliche $\mathbf{p}$ im aktuellen Mini-Batch des Trainings. Indem die Einträge auf der Diagonalen nicht minimiert werden, die Einträge außerhalb der Diagonalen jedoch schon, wird eine Darstellung gefördert, die die einzelnen Merkmale, welche d-fach im Feature Vektor gebündelt sind, entkoppelt, also unabhängig voneinander macht.
Die finale Verlustfunktion ergibt sich dann aus einer Linearkombination dieser drei Verlustfunktionen:

$$\mathcal{L}_T = \lambda_{KD}\mathcal{L}_{KD} + \lambda_M\mathcal{L}_M + \lambda_C\mathcal{L}_C, \quad \lambda_{KD}, \lambda_M, \lambda_C \in \mathbb{R}^+$$

Die Faktoren $\lambda_{KD}, \lambda_M, \lambda_C$ sind dabei Hyperparameter, die das Training beeinflussen.

Lernen der Studenten

Nun wird sich mit dem Training der Studenten S_i beschäftigt. Diese sollen in die Lage versetzt werden, nominale Patches in der selben Weise, wie der Teacher T zu extrahieren, also dessen Ausgabe auf nominalen Bildern zu imitieren. Liegt ein anomales Bild vor, so soll die Ausgabe der Studenten möglichst stark von der des Teachers abweichen.
Zunächst wird die Ausgabe des trainierten Teachers T über alle Bilder in $\mathcal{X}_{train}$ erzeugt. Aus den so entstandenen Merkmalsvektoren wird dann der Mittelwert $\boldsymbol{\mu} \in \mathbb{R}^d$ und die Standardabweichung $\sigma \in \mathbb{R}^d$ komponentenweise bestimmt. Für jede mögliche Position (h, w) mit $h \in \{1, ..., h^*\}$ und $w \in \{1, ..., w^*\}$ wird dann die Ausgabe y der Studenten S_i bestimmt und als Gaußverteilung $P\left(\mathbf{y}|\mathbf{p}_{h,w}\right) = \mathcal{N}\left(\mathbf{y}|\boldsymbol{\mu}_{h,w}^{S_i}, s\right)$ mit konstanter Kovarianz $s \in \mathbb{R}$ modeliert. $\boldsymbol{\mu}_{h,w}^{S_i}$ ist hierbei die Prädiktion von S_i zur Stelle (h, w). Sei nun $\mathbf{y}_{h,w}^T$ die von den Studenten zu prädizierende Zielgröße, also die Ausgabe des trainierten Teachers, ergibt sich damit folgende Verlustfunktion für das Training des Studenten:

$$\mathcal{L}(S_i) = \frac{1}{h^* w^*} \sum_{(h,w)\in\left\{1,...,h^*\right\}\times\left\{1,...,w^*\right\}} \left\| \boldsymbol{\mu}_{h,w}^{S_i} - \left(\mathbf{y}_{h,w}^T - \boldsymbol{\mu}\right) diag\left(\boldsymbol{\sigma}^{-1}\right) \right\|_2^2$$

$diag(\boldsymbol{\sigma})^{-1}$ ist dabei die Inverse der Diagonalen der Matrix, die mit den Werten aus $\boldsymbol{\sigma}$ gefüllt ist.

Bestimmen des Anomaliegrades

Nach dem Training der Studenten bis zur Konvergenz, wird für jeden Position (h, w) eines Bildes eine Gauß-Mixtur bestimmt. Dies geschieht über alle Studenten des Ensembles, die alle gleichgewichtet eingehen. Damit kann auf zwei verschiedene Wege ein Anomaliegrad bestimmt werden.
Zunächst lässt sich der Regressionsfehler zwischen der Ausgabe des Teachers und dem Mittelwert der Ausgaben der Studenten bestimmen.

$$e_{h,w} = \left\| \boldsymbol{\mu}_{h,w} - \left(\mathbf{y}_{h,w}^T - \boldsymbol{\mu}\right) diag\left(\boldsymbol{\sigma}^{-1}\right) \right\|_2^2$$

$\boldsymbol{\mu}_{h,w}$ ist hierbei der Mittelwert der Ausgaben der Studenten an der Position (h, w) ($\boldsymbol{\mu}_{h,w} = \frac{1}{M}\sum_{i=1}^{M} \boldsymbol{\mu}_{h,w}^{S_i}$). Die Idee ist nun, dass ein hoher Regressionsfehler auf eine Anomalie hinweist, weil die Studenten nicht in der Lage sind, die Ausgabe des Teachers zu imitieren.
Eine andere Möglichkeit der Bestimmung eines Anomaliegrades ist es, die Unsicherheit der Prädiktionen der Studenten zu betrachten. Dies basiert auf der Annahme, dass die Augaben der Studenten auf nominalen Bildern eine geringere Varianz aufweisen, als auf anomalen Bildern, weil diese Ungesehenes beinhalten, für das das jeder Student S_i eine andere, zufällig streuende Prädiktion ausgibt. Quantitativ kann das wie folgt formuliert werden:

$$v_{h,w} = \frac{1}{M}\sum_{i=1}^{M} \left\| \boldsymbol{\mu}_{h,w}^{S_i} \right\|_2^2 - \left\| \boldsymbol{\mu}_{h,w} \right\|_2^2$$

Mit einer Normalisierung der jeweiligen Größen $e_{h,w}$ und $v_{h,w}$ mithilfe der Mittelwerte und Standardabweichung über alle Positionen (h, w) folgt schließlich das für ein Studeten-Teacher-Paar finale Anomaliemaß für die Position (h, w):

$$\tilde{s}_{h,w} = \tilde{e}_{h,w} + \tilde{v}_{h,w} = \frac{e_{h,w} - e_\mu}{e_\sigma} + \frac{v_{h,w} - v_\mu}{v_\sigma}.$$

Weitergehend kann nur ein Ensemble an Student-Teacher-Paaren verwendet werden, um die Anomalieklassifikation zu verbessern. Sinnvoll ist es, solche Paare zu trainieren, die unterschiedliche rezeptive Felder p aufweisen. Dann können Anoamlien unterschiedlicher Größe detektiert werden, ohne dass ein Hoch oder Runterskalieren des Bildes notwendig ist, was immer mit einem Informationsverlust einhergeht. Für L Student-Teacher-Paare ergibt und die die jeweiligen Anomliegrade $\tilde{s}_{h,w}^{(l)}$, ergibt sich dann

$$s_{h,w} = \frac{1}{L}\sum_{l=1}^{L} \tilde{s}_{h,w}^{(l)}.$$

4.2.2 Ergebnisse und Diskussion

Es handelt sich bei dieser Methode um die älteste in dieser Arbeit beschriebene Methode. Viele Erkentnisse, die in den darauffolgenden Jahren durch zahlreiche Veröffentlichungen gewonnen wurden, fehlten den Autoren dieser Veröffentlichung. So wurde als Lernziel für den Teacher T und die Wissensdistillation ein ResNet 18 verwendet, dessen Feature dann mittels PCA in die richtige Dimension gebracht wurde. Heute ist bekannt, dass tiefere Netze bessere Feature Extraktoren sind und PCA keine geeignete Methode ist, um Patch Feature zu reduzieren. Letzteres wurde z.B. in PaDiM festgestellt, aber auch in qualitativen Untersuchungen in dieser Arbeit. Auch das metrische Lernen erweist sich als nicht hilfreich, wie bereits die Autoren selbst in ihrer Veröffentlichung feststellen.
Vor diesem Hintergrund ist die erzielte Genauigkeit dennoch beachtlich und entspricht in etwa den Ergebnissen von SPADE. Der Fokus in dieser Arbeit lag nicht auf einer Instanzklassifiziergung, sondern auf der Segmentierung, weswegen hierfür weder eine Methode, noch Ergebnisse

veröffentlicht wurden. Aufgrund der pixelweisen Metriken, die in der Veröffentlichung angeben sind und an dieser Stelle nicht weiter besprochen werden, lässt sich diese Aussage treffen.
Auch Aussagen zur Laufzeit müssen qualitativ erfolgen. Es ist nicht davon auszugehen, dass durch die Vielzahl an Elementen ($M = 3$ und $L = 3$) die Laufzeit geringer ist, als bei den anderen Methoden. Insgesamt müssten bei einem solchen Konfiguration, wie sie in der Veröffentlichung angegeben ist, 9 CNNs auf einem Bild ausgeführt werden.
Viel wichtiger als die tatsächlichen Ergebnisse, sind aber die Impulse, die durch diese Arbeit gesetzt wurden. Folgende Erkentnisse sind hierbei besonders hervorzuheben:

- Eine Feature Exktrattion durch Wissensdistillation ist möglich und ermöglicht spezifisch angepasste Feature Extraktoren, die auf die Aufgabe zugeschnitten sind.
- Ein Student-Teacher Ansatz ist eine geeignete Methode, um Anomalien zu detektieren.
- Verschieden aufgelöste Feature Maps können zu einer Verbesserung der Anomaliedetektion führen.

In nachfolgendem Abschnitt zur Funktionsweise von EfficientAD werden einige dieser Erkentnisse aufgegriffen und weiterentwickelt.

4.3 Funktionsweise von EfficientAD

Abgeschlossen: 02.11. v1
In diesem Abschnitt wird die Funktionsweise der Methode EfficientAD beschrieben. Es wird hauptsächlich das Konzept des Student-Teacher Designs, das in 4.2 beschrieben wurde, verwendet. Dabei gibt es zwei, konzeptionell unterschiedliche Student-Teacher Paare, die jeweils unterschiedliche Aufgaben erfüllen.

4.3.1 Feature Extraktion

Ein wesentlicher Unterschied zu PatchCore und SimpleNet ist, dass für die Feature Exktration nicht auf explizit auf ein vortrainiertes ResNet gesetzt wird. Zwar wird diese Grundidee immer noch beibehalten, insofern, als dass ein solches ResNet verwendet wird, um den eigentlichen Feature Extraktor zu trainieren. Während der Inferenz wird jedoch kein ResNet verwendet. Stattdessen kommt ein sogenannter **Patch Description Network (PDN)** zum Einsatz, welches mithilfe von Wissens-Distillation trainiert wird. Es besteht aus lediglich vier Convolutional-Layern, die in 4.2 dargestellt sind.
In 4.2 zu sehen, ist, dass die Ausgabe eines PDN 384-dimensionale Feature-Vektoren sind, deren rezeptives Feld exakt $p = 33$ Pixel beträgt. Dies entspricht der Idee, die bereits drei Jahre zuvor in „Uninformed Students“ [6] vorgestellt wurde. Im Falle eines ResNets als Feature Exktraktor kann das rezeptive Feld selten grenzscharf bestimmt werden, wodurch sich das PDN für Segmentierungsaufgaben in dieser Hinsicht besser eignet.

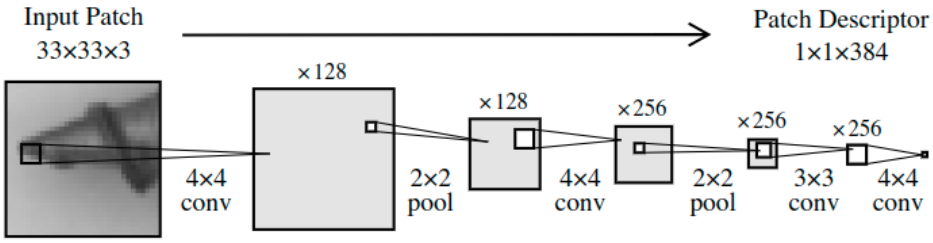


Abbildung 4.2: Skizze der Architektur des Patch Description Networks (PDN). [2]

Eine räumliche Dimensionsreduktion findet durch ein schrittweises Average-Pooling nach den ersten beiden Convolutional-Layers statt. Die geringere Dimension der Feature Maps sorgen für eine schnellere Laufzeit. Wie noch zu sehen sein wird, ist diese Dimensionsreduktion allerdings deutlich zurückhaltender, als zum Beispiel bei ResNet-Architekturen.
Durch den vollständig aus Convolutional- und Pooling-Layern bestehende Aufbau des PDN, ist es möglich, alle Feature Vektoren eines Bildes in einem Durchlauf zu extrahieren. Es stehen zwei verschiende PDNs zur Verfügung, welche sich in ihrere Größe und daraus resultierend, ihrer Laufzeit unterscheiden. Insbesondere die Anzahl an Filter in den Convolutional-Layern unterscheidet sich, also die Anzahl an Channels. Diese ist im Falle des größeren Netzes deutlich höher. Für Details hierzu wird auf die Veröffentlichung [2] und Tabelle 5 bzw. 6 verwiesen.
Wie bereits erwähnt, erfolgt das Training des PDNs mithilfe von Wissens-Distillation von einem vortrainierten ResNet. In 4.2.1 wurde bereits beschrieben, wie ein solcher Distillationsprozess aussieht. Definieren wir den PDN als $T : \mathbb{R}^{3\times256\times256} \rightarrow \mathbb{R}^{384\times64\times64}$ benötigen wir einen vortrainierten Feature Extraktor $\phi : \mathbb{R}^{3\times W\times H} \rightarrow \mathbb{R}^{384\times64\times64}$, der auf ImageNet vortrainiert wurde. Die Kantenlängen W und H sind dabei so zu wählen, dass die Ausgabe des Feature Extraktors ϕ 64×64 Pixel groß ist. Hierzu eignet sich ein Feature Extraktor, wie in 3.2.1 bei der Methode PatchCore verwendet, ideal. Es wird auch in der Veröffentlichung der Einbettungsprozess aus 3.2.1 verwendet und ein Wide ResNet 101 [37]. Die Verlustfunktion für das Training des PDNs auf mit einem Bild $x \in \mathcal{X}_{train}$ ergibt sich dann einfach zu:

$$\mathcal{L} = \|T(x) - \phi(x)\|_2^2$$

$\mathcal{X}_{train}$ ist hierbei aus dem ImageNet Datensatz entnommen. Es handelt sich also um einen $L2$-Loss. Die Auflösung $H \times W$ der Bilder aus ImageNet beträgt in diesem Fall, also mit einem Wide ResNet 101, 512×512 Pixel.

4.3.2 Reduzierter Student-Teacher Ansatz

Die in 4.2 beschriebene Methode „Uninformed Students“ wird in EfficientAD in einer reduzierten Form verwendet. Es wird kein Ensemble an Studenten verwendet, sondern nur ein einzelner Student, was $M = 1$ in 4.2.1 entspricht. Ebenfalls kommt lediglich eine Student-Teacher Paarung in diesem Schritt zum Einsatz. Gegenüber der Veröffentlichung „Uninformed Students“[6] ist das eine weitere Vereinfachung. Diese dienen in erster Linie einer kürzeren Laufzeit.
Der Teacher T ist dabei durch das PDN, trainiert durch Wissensdistillation, gegeben. Für den Studenten S wird die Architektur identisch übernommen.
Es wird somit auch auf eine asymetrische Architektur verzichtet, wie sie erfolgreich in [27] angewandt wurde. Stattdessen wird auf eine angepasste Verlustfunktion zurrückgegriffen, die im Folgenden beschrieben wird.
Grundsätzlich besteht die Schwierigkeit beim Trainieren eines Student-Teacher Paares darin, den Studenten zwar soweit zu trainieren, dass er in der Lage ist, das Verhalten des Teachers auf nominalen Beispielen möglichst genau zu imitieren, aber dennoch nicht genug generalisiert

zu haben, dies auch in einem anomalen Fall zu tun. Es entsteht ein Trade-Off, der in dieser Veröffentlichung clever gelöst wird.
So wird zunächst ein Verfahren eingeführt, dass nur Bereiche im Bild für das Anpassen der Parameter mithilfe von Backpropagation verwendet, welche besonders große Abweichungen zwischen der Ausgabe des Teachers und des Studenten aufweisen. Geringfügige Abweichungen werden ignoriert.
Formal wird dazu ein Trainingsbild $x \in \mathcal{X}_{train}$ verwendet, um die Ausgabe sowohl des Studenten $S(x)$, als auch des Teachers $T(x)$ zu bestimmen. Dabei ist $T(x), S(x) \in \mathbb{R}^{C \times H \times W}$. Daraus wird dann die quadratische Differenz $D_{c,w,h} = \big(T(x)_{c,w,h} - (S(x)_{c,w,h})\big)^2$ für jedes Element (c, h, w) bestimmt. Der Idee folgend, nur die größten Differenzen eingehen zu lassen, wird anschließend ein Schwellwert d_{hart} berechnet, der von einem Hyperparameter $p_{hart} \in [0, 1]$ abhängt. Dieser bestimmt, wie groß die Menge an Elementen aus $D_{c,w,h}$ ist, die für das Training verwendet werden. Es gilt, dass d_{hart}-Quantil d_{hart} ist. Für $p_{hart} = 0{,}999$ werden laut [2] etwa 10% der Elemente aus $D_{c,w,h}$ für das Training verwendet. Würde $p_{hart} = 0$ gewählt, so würden alle Elemente $D_{c,w,h}$ für das Training verwendet. Konkret wird dann die Verlustfunktion $\mathcal{L}_{hart}$ zum Durchschnitt über alle Werte aus D bestimmt, für die gilt, dass $D_{c,w,h} \geq d_{hart}$.
In der Praxis führt dieser Ansatz dazu, dass nur wesentliche Bereiche im Bild für das Training verwendet werden. Veranschaulicht ist das in nachfolgender Abbildung, welche der Veröffentlichung entnommen wurde (4.3). Es ist zu erkennen, dass Objektbereiche in $\mathcal{L}_{hart}$ eingehen, der Hintergrund hingegen bereits vom Studenten gut imitiert wird und somit nicht mehr in die Verlustfunktion eingeht.
Zusätzlich zu dieser speziellen Verlustfunktion, die für das Training des Studenten verwendet

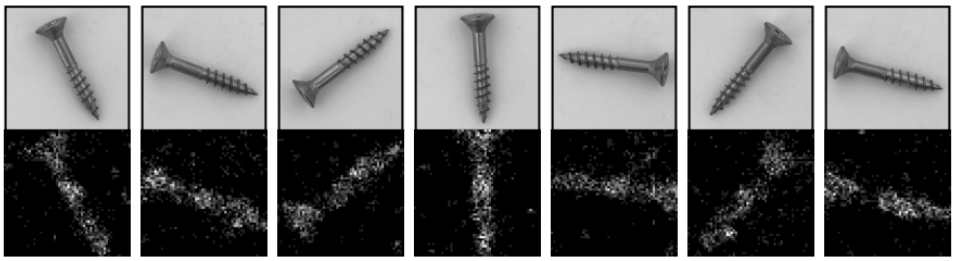


Abbildung 4.3: Veranschaulichung der „Hard Loss“-Filterung für $\mathcal{L}_{hart}$. Obere Reihe stellt Trainingsbilder dar, Untere die Masken. Alle dunklen Bereiche werden ignoriert, helle Bereiche werden zum Training verwendet. [2]

wird, wird ein Strafterm eingeführt, der den Studenten daran hindert, den Teacher auf Bildern, die nicht Teil der nominalen Trainingsdaten sind, zu imitieren. In einem klassischen Setup würde der Student vom Teacher nur mit Bildern lernen, die nominal und aus der Zieldomäne stammend sind. Der Teacher hingegen würde auf einem größeren, breter gefassten Datensatz trainiert werden, zum Beispiel mit ImageNet. In dieser Veröffentlichung wird der Student zusätzlich auf Bildern trainiert, die aus dem ImageNet Datensatz stammen. Konkret wird in jedem Trainingsschritt ein Bild $x_{imagenet}$ aus ImageNet zufällig ausgewählt und mit der Verlustfunktion $\mathcal{L}_{Strafterm} = \frac{\sum_c \|S(x_{imagenet})_c\|_F^2}{HWC}$ ein Strafterm bestimmt. Es wird hierfür die Frobenius-Norm

über alle Kanäle verwendet. Dieser Strafterm verhindert, dass der Student auf Bildern, die nicht aus der Zieldomäne stammen, generalisiert.
Die finale Verlustfunktion für den Studenten ergibt sich dann zu:

$$\mathcal{L}_{Student} = \mathcal{L}_{hart} + \mathcal{L}_{Strafterm}$$

Eine detailierte Aufschlüsslung des gesamten Trainingsprozesses ist in [2] im Appendix A.1 und Algorithus 1 zu finden.

4.3.3 Erkennen logischer Anomalien

Wie bereits in der Einleitung zu diesem Kapitel erwähnt, ist EfficientAD in der Lage, neben strukturellen Anomalien, auch logische Anomalien zu erkennen. Weil bei PatchCore und SimpleNet keine globale Strukturinformationen verwendet werden, sondern immer nur bereichsweise analysiert wird, ist es nur sehr eingeschränkt möglich mit diesen Methoden logische Anomalien zu erkennen. In 2.1 wurde bereits erwähnt, dass der MVTec AD Datensatz im Wesentlichen keine logischen Anomalien enthält und die Zielsetzung dieser Arbeit nicht die Erkennung logischer Anomalien ist. Deshalb wird diese interessante Fähigkeit von EfficientAD hier nicht weiter betrachtet. Auf dem Datensatz MVTec LOCO AD, der ebenfalls in 2.1 Erwähnung findet und hautpsächlich logische Anomalien enthält, ist die Methode EfficientAD die beste bislang veröffentlichte Methode (Stand Oktober 2023). [14]
Um solche Anomalien zu erkennen, wird ein Autoencoder verwendet, um die logischen Zusammenhänge nominaler Bilder zu erlernen. Es wird sich dabei vor allem an der Methode „GCAD“ orientiert [5], die sich mit dem Erkennen logischer Anomalien mithilfe von Autoencodern beschäftigt. Weil dies nicht Schwerpunkt dieser Arbeit ist, wird auf eine Erläuterung der Methode GCAD hier verzichtet. Die wesentlichen Aspekte von GCAD finden sich ohnehin in dem hier Beschriebenen wieder.
Ebenfalls wird ein Student-Teacher Paar gebildet. Auch hier wird der Student A, der durch den Autoencoder implementiert wird, trainiert, um die Ausgabe des Teachers T zu imitieren. Formal ergibt sich die Verlustfunktion für das Training des Autoencoders für ein Bild x zu:

$$\mathcal{L}_{AE} = \frac{\sum_c \|A(x)_c - T(x)_c\|_F^2}{HWC}$$

Auch hier ist $T(x), A(x) \in \mathbb{R}^{C \times H \times W}$ und die Frobenius-Norm wird über alle Kanäle verwendet. Das Bild x ist dabei ein Bild aus dem Trainingsdatensatz, der in diesem Fall ausschließlich aus nominalen Bildern der Zieldomäne besteht.
Im Gegensatz zu den PDNs, die immer nur ein endlich großes rezeptives Feld haben, welches im Falle der PDNs sogar exakt definiert ist, wird vom Autoencoder das gesamte Bild betrachtet. Es muss möglichst viel der Bildinformation durch ein „Bottleneck“ (dt.: Flaschenhals) bringen, um eine Rekonstruktion des Bildes zu ermöglichen. Dieses Bottleneck ist dabei lediglich 64 Kanäle fassend, wodurch eine Komprimierung erzwungen wird.

Man spricht Im Zuammenhang mit Autoencodern vom Encoder, der die Bildinformation auf einen niedrigdimensionalen Vektor, das Bottleneck, abbildet und dem Decoder, der diesen Vektor wieder auf eine größere Dimension abbildet. Diese Zieldimension entspricht dabei nicht der Auflösung des Eingangsbildes, sondern der Auflösung der Patch Feature Vektoren.
Auf Bildern mit logischen Anomalien ist die grundsätzliche Struktur des Bildes verschieden zur nominalen Struktur. Da der Autoencoder jedoch aus dem Training nur nominale Bilder zu sehen bekommt und diese grundsätzliche Struktur implizit zur möglichst fehlerfreien Rekonstruktion erlernt und ausgenutzt hat, wird der Autoencoder nicht in der Lage sein, eine Abweichung der nominalen Struktur ebenfalls korrekt zu rekonstruieren.
Autoencoder haben im Allgemeinen die Schwäche, dass die Rekonstruktionen von fein aufgelösten Bilddetails nicht gut gelingt. Dies ist auch hier der Fall. Um falsche Positive, also scheinbare Anomalien, die durch eine schlechte Rekonstruktion solcher feiner Strukturen entstehen würden, zu vermeiden, wird die Anzahl der Ausgangskanäle des Studenten S verdoppelt und trainiert, die Ausgabe des Autoencoders zu imitieren. Definiert man nun den Teil des Studenten, der mit dem Autoencoder korrespondert mit S' und $S'(x) \in \mathbb{R}^{C\times H\times W}$ als die Ausgabe dieses Teils des Studenten, ergibt sich die Verlustfunktion für das Training des Studenten S' zu:

$$\mathcal{L}_{AE} = \frac{\sum_c \left\| S'(x)_c - A(x)_c \right\|_F^2}{HWC}$$

Somit lernt der Student S', die systematischen Rekonstruktionsfehler des Autoencoders A mit, wodruch diese nicht mehr zu falschen Positiven führen. Außerdem ist der Student S' Teil des PDNs, welches ein rezeptives Feld von $p = 33$ Pixeln hat. Es ist also nicht in der Lage größere Strukturen zu erkennen und somit in aller Regel nicht beeinfluss von logischen Anoamlien.

4.3.4 Bestimmen des Anomaliegrades

In beiden oben genannten Paaren, also Autoencoder A und Student S' sowie PDN T und Student S, kann die Differenz der Ausgaben als Anomaliemaß verwendet werden. Konkret wird die pixelweise quadratische Differenz der Ausgaben berechnet. Es entstehen somit zwei Anomaliekarten, die im Folgenden als lokale Anomliekarte für die Differenz von S und T und als globale Anomaliekarte für die Differenz von S' und A bezeichnet werden. In folgender, der Veröffentlichung entnommenen, Abbildung ist das unterschiedliche Verhalten dieser beiden Anomaliekarten veranschaulicht.
Es handelt sich hierbei um zwei Beispielbilder aus dem bereits erwähnten Datensatz MVTec LOCO AD. Auf der linken Seite ist ein Testbild zu sehen, welches einen kleinen, strukturellen Defekt aufweist. Die globale Struktur des Bildes ist aber in Ordnung. Alle Baulteile sind an Stellen, an denen man sie erwarten würde. Dementsprechend ist die Differenz, die unten in der „Global Map“ abgebildet ist, sehr gering. Dies lässt also keinen Schluss auf eine Anomalie zu. Die lokale Anomaliekarte hingegen, die oben zu sehen ist, zeigt eine deutlich erhöhte Differenz an der Stelle des Defekts, was wiederum eindeutig auf die tatsächlich existierende, strukturelle

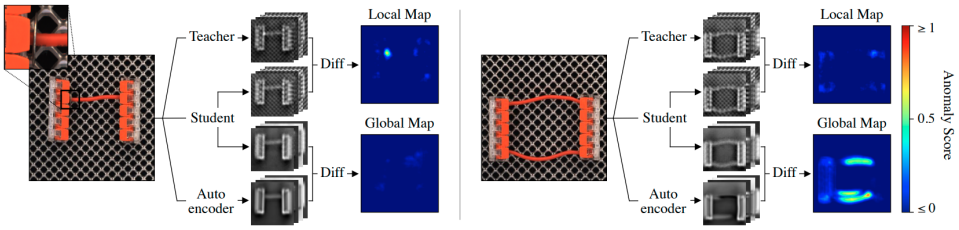


Abbildung 4.4: Veranschaulichung der Unterschiede zwischen lokaler und globaler Anomaliekarte. [2]

Anomalie hinweist.
Auf der rechten Seite ist ein Testbild zu sehen, welches eine logische Anomalie aufweist. Die globale Anomaliekarte zeigt hier eine deutlich erhöhte Differenz, während die lokale Anomaliekarte keine erhöhte Differenz aufweist.
Nun sollen aber nicht logische und strukturelle Anomlien voneinander unterschieden werden, sonder eine resultierende, finale Anomaliekarte soll erstellt werden. Dies geschieht, naheliegenderweise, durch eine Kombination der beiden Anomaliekarten. Es müssen allerdings unterschiedliche Rauschniveaus der beiden Anomaliekarten berücksichtigt werden, um keine falschen Positive zu erzeugen. Um das Ausmaße des Rauschens für beide Karten zu bestimmen, werden ungesehene Bilder aus dem Trainingsdatensatz verwendet. Mit diesen wird eine Menge aller auftretenden Differenzen jeweils für beide Karten bestimmt. Mithilfe dieser Mengen werden dann die Quantile q_a und q_b bestimmt, die das Rauschniveau der jeweiligen Karte beschreiben. Abschließend wird jeweils eine lineare Transformation bestimmt, die die den Wert q_a auf 0 und den Wert q_b auf 0,1 abbildet. Zu den einzelnen Zahlenwerten sind Ablation Studien in [2] zu finden.
Die finale Anomliekarte ergibt sich dann einfach aus der Addition der beiden Anomaliekarten.

4.4 Ergebnisse und Diskussion der Originalmethode

Abgeschlossen: 02.11. v1
Trotz des leichtgewichtigen Aufbaus des PDNs, ist eine Beschleunigung der Feature Extraktion gegenüber einem kleinen ResNets nicht zu erwarten. Dies liegt vor allem daran, dass die Feature Maps in frühen Schichten des ResNets gegenüber dem Eingangsbild stark herunterskaliert werden. Zwar wird dies auch bei den hier verwendeten PDNs getan, aber in einer weniger stark ausgeprägten Weise. Bereits nach der „Layer1“ ist bei einem ResNet, wie 2.4 zu entnehmen, die räumliche Auflösung 16 mal kleiner als die des Eingangsbildes. Beim PDN ist sie nur geringfügig kleiner. Weil somit der Tensor, dessen Größe bei einem CNN maßgeblich die Anzahl an Berechnungen bestimmt, damit bei einem PDN deutlich größer ist, ist eine Beschleunigung der Feature Extraktion nicht zu erwarten. Der Vorteil eines solchen Ansatzes ist, dass die Auflösung der finalen Anomaliekarte auch ohne ein Hochskalieren verhältnismäßig groß bleibt. Insbesondere für eine Segmentierungsaufgaben ist das eine wertvolle Eigenschaft, die für diese Arbeit allerdings keine explizite Rolle spielt. Vergleicht man die Laufzeiten gesamter ResNets aus 2.4 mit denen der PDNs bestätigt sich die Vermutung, dass die Laufzeit nicht geringer ist. Für ein Eingangsbild der Dimension 224×224 ist die Laufzeit auf der Desktop-CPU (AMD Ryzen R5 3600X, mittlere Spalte) mit 42,42ms und 143,83ms für die kleine bzw. große Variante des PDNs sogar jeweils höher, als mit vergleichbaren ResNets. Auf der GPU hingegen (Nvidia RTX3060Ti, rechte Spalte), die das Mehr an Berechnungen durch Parallelisierung kompensieren kann, ist die Laufzeit mit 1,32ms bzw. 5,19ms im Falle der kleinen Variante des PDNs sogar geringer, als für ein ResNet 18 mit 1,46ms. An dieser Stelle ist allerdings zu erwähnen, dass je nach gewählter Hierarchieebene des ResNets, sich die Laufzeit für die Feature Extraktion

verkürzt weil hintere Schichten des ResNets nicht mehr ausgeführt werden müssen. Hinzu kommt für den Fall, dass mehrere Hierarchieebenen verwendet werden, der Einbettungsprozess, der bei den PDNs nicht notwendig ist.
Betrachtet man die Laufzeiten des größereren PDNs, kann bereits jetzt festgestellt werden, dass lediglich das Verwenden des kleineren PDNs für diese Arbeit sinvoll ist. Die Laufzeit des größeren PDNs ist ohne Hardwarebeschleunigung schlicht zu hoch.

Kapitel 5

SimpleNet

5.1 Einleitung

Abgeschlossen: 27.10. v1

SimpleNet ist die zweite Methode, die im Rahmen dieser Arbeit genauer analysiert wird. Das am 28. März 2023 im Rahmen der Konferenz „Computer Vision and Pattern Recognition“ veröffentlichte Paper „SimpleNet: A Simple Network for Real-Time Instance Segmentation“ verspricht eine einfache, schnelle und präzise Methode. Es wird eine Genauigkeit von $99{,}6\%$ AUROC auf MVTecAD bei einer 8-mal schnelleren Laufzeit im Vergleich zur PatchCore Variante, welche mit 10% Subsampling in 3.2 zu sehen ist, propagiert. Es handelt sich also im Hiblick auf die Zielsetzung in dieser Arbeit um eine sehr vielversprechende Methode.

5.2 Funktionsweise

Abgeschlossen: 27.10. v1

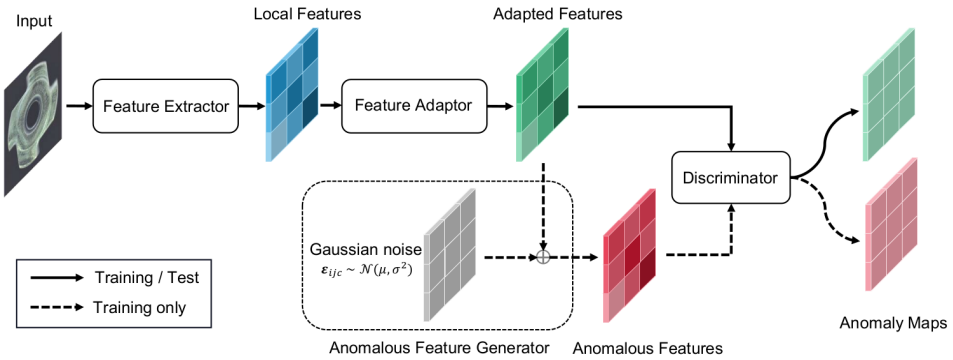


Abbildung 5.1: Funktionsweise von SimpleNet im Überblick [20]

Die grobe Funktionsweise lässt sich anhand von Abbildung 5.1 bereits gut erläutern. Analog zu den bisher vorgestellten Methoden muss auch hier zwischen einer Trainingsphase und der Inferenz- bzw. Testphase unterschieden werden. Die Teile der Abbildung, die mit gestrichelten Linien miteinander verbunden sind, werden nur während der Trainingsphase verwendet. Der

obere Strang ist somit die Pipeline, die sequentiell während der Inferenz ausgeführt wird.
In der Trainingsphase werden nominale Bilder aus dem Trainingsdatensatz in einen vortrainierten **Feature Extraktor** gegeben - analog zu 3.2.1. Dessen Ausgabe wird dann in einem **Feature Adaptor** transformiert, um eine geeingetere Darstellung zu erhalten. Dann werden **Pseudo-Anomalien** aus den so entstandenen Feature erzeugt, indem zufällig Gauß'sches Rauschen aufaddiert wird. Schließlich wird ein **Diskriminator** trainiert, dessen Ziel es ist, diese Pseudo-Anomalien von nicht manipulierten und somit nominalen Feature zu unterscheiden.
Durch dieses Training wird der Diskriminator in die Lage versetzt, auch tatsächliche Anomalien zu erkennen.
Im Folgenden wird dieser Prozess schrittweise noch einmal genauer erläutert.

5.2.1 Erzeugen der Patch Feature

An dieser Stelle kann auf die Ausführungen in 3.2.1 verwiesen werden. Der Prozess des Erzeugens von Patch Feature ist vollständig von der Methode PatchCore übernommen worden. Folglich gilt auch hier folgende Formel für die Menge aller Patch Feature, die aus einem Bild extrahiert werden:

$$\mathcal{P}_p\left(\phi_{i,j}\right) = \left\{\phi_{i,j}\left(\mathcal{N}_p^{(h,w)}\right) | h \in \{1, ..., h^*\}, w \in \{1, ..., w^*\}\right\}$$

mit der „Neighborhood" $\mathcal{N}_p^{h,w}$ und der Patchgröße p:

$$\mathcal{N}_p^{h,w} = \left\{(a,b) | a \in \left[h - \left\lfloor \frac{p}{2} \right\rfloor, ..., h + \left\lfloor \frac{p}{2} \right\rfloor\right], b \in \left[w - \left\lfloor \frac{p}{2} \right\rfloor, ..., w + \left\lfloor \frac{p}{2} \right\rfloor\right]\right\}$$

Dabei beizeichnet i den Index des Eingagsbildes x_i, j die Hierarchieebene des Feature Exktraktors ϕ, welches ein vortrainiertes CNN ist. h und w geben die Ortskoordinate innerhalb einer Feature Map an, die die Größe $h^* \times w^*$ hat.

5.2.2 Feature Adaptor

Ein zusätzliches Element gegenüber PatchCore ist der Feature Adaptor G_θ. Dieser hat die Aufgabe, die Patch Feature in eine kompaktere Darstellung zu überführen. Die Varianz der Patch Feature innerhalb einer Domäne wird dadurch reduziert, um besser in der Lage zu sein anomale Feature zu erkennen. Formal lässt sich der Feature Adaptor wie folgt beschreiben:

$$G_\theta : \mathbb{R}^c \to \mathbb{R}^{c'}$$

mit den Parametern θ und der Anzahl der Kanäle c und c'. In der Veröffentlichung wurde konsequent $c = c'$ gewählt. Bei Versuchen zeigte sich, dass eine einfache lineare Transformation die besten Ergebnisse liefert. Es findet also eine Matrix-Vektor Multiplikation statt zwischen

den einzelnen Patch Feature Vektoren $p_{h,w}$ aus $\mathcal{P}_p\left(\phi_{i,j}\right)$ und einer im Falle von $c = c'$ quadratischen Matrix θ. Die so entstehenden Feature Vektoren $G_\theta(p_{h,w})$ werden im Folgenden als $q_{h,w}$ bezeichnet.

5.2.3 Erzeugen der Pseudo-Anomalien

Die Pseudo-Anomalien werden erzeugt, indem zufällig Gauß'sches Rauschen aufaddiert wird.

$$q^-_{h,w} = q_{h,w} + \epsilon, \epsilon \sim \mathcal{N}(\mu, \sigma^2)$$

Das Rauschen ist dabei unabhängig von den anderen Feature Vektoren (i.i.d.) und Mittelwertfrei ($\mu = 0$). Außerdem gilt naheliegenderweise $\epsilon \in \mathbb{R}^{c'}$. Durch Studie der offiziellen Implementierung werden weitere Details zu diesem Prozess deutlich. Aus einem Bild x_i werden, wie bereits hinlänglich bekannt, die Patch Feature $\mathcal{P}_p\left(\phi_{i,j}\right)$ erzeugt. Für jeden dieser Patch Feature Vektoren $p_{h,w}$ wird jeweils ein zufälliger Vektor $\epsilon_{h,w}$ erzeugt und aufaddiert. Es wird also ein gesamtes Bild an Pseudo-Anomalien erzeugt. Dies unterscheidet sich von einer typischen strukturellen Anomalie, die nur einen kleinen Teil des Bildes betrifft. Weil aber jeder Patch Feature Vektor für sich genommen auf Anomalien geprüft wird und jedem einzelnen Patch Feature Vektor ein Anomliegrad zugeordnet wird, hat diese Abstraktion keinen negativen Einfluss auf die Ergebnisse.

5.2.4 Diskriminator

Der Diskriminator D_ψ hat die Aufgabe, jedem einzelnen Patch Feature Vektor $q_{h,w}$ einen Anomaliegrad zuzuordnen. Negative Ausgaben deuten dabei auf Anomlien hin, während positive Ausgaben nominale Feature implizieren. Somit gilt, dass $D_\psi(q_{h,w}) \in \mathbb{R}$ ist. Es wurde hierfür ein einfaches, zweischichtiges Multi-Layer-Perzeptron verwendet.

5.2.5 Verlustfunktion und Training

Der Disktiminator soll in die Lage versetzte werden, anomale Patch Feature Vektoren den Wert $th^- = 0{,}5$ zuzordnen, während nominale Feature Vektoren den Wert $th^+ = 0{,}5$ erhalten sollen. Die Autoren verwenden dafür einen einfachen $l1$-Loss. Die Ausgabe wird auf positive Werte beschränkt. Es ergibt sich für ein Patch Feature Trainings-Tupel $(q^i_{h,w}, q^{i-}_{h,w})$ folgende Verlustfunktion:

$$l^i_{h,w} = \max\{0, th^+ - D_\psi(q^i_{h,w})\} + \max\{0, D_\psi(q^{i-}_{h,w}) - th^-\}$$

Dieser Vorgang wird dann für alle Positionen h und w und über alle Bilder x_i im Trainingsdatensatz $\mathcal{X}_{train}$ durchgeführt. Der Gesamtverlust ergibt sich dann aus der Summe aller Verluste:

$$\mathcal{L} = \sum_{x_i \in \mathcal{X}_{train}} \sum_{h,w} \frac{l^i_{h,w}}{w^* h^*}$$

Dabei gilt $(h, w) \in \{1, ..., h^*\} \times \{1, ..., w^*\}$. Diese Funktion wird minimiert im Sinne der Parameter θ und ψ. Das bedeutet, dass sowohl der Feature Adaptor G_θ als auch der Diskriminator D_ψ trainiert werden.

5.2.6 Bestimmen des Anomaliegrades (Inferenz)

Wie bereits erwähnt, wird bei der Inferenz ein Bild x_i in Patch Feature $\mathcal{P}_p\left(\phi_{i,j}\right)$ überführt. Diese werden dann in den Feature Adaptor gegeben, um die Feature Vektoren $q_{h,w}$ zu erhalten. Abschließend werden diese Feature Vektoren in den Diskriminator gegeben, um den Anomaliegrad $s_{h,w} = D_\psi(q_{h,w})$ zu erhalten. So kann dann durch das Zusammenführen der räumlichen Information die Anomaliekarte für das gesamte Bild bestimmt werden. Für diese Arbeit relevant ist aber vor allem der Anomliegrad des gesamten Bildes. Dies geschieht einfach über die Maximalwertbildung über die Anomaliekarte.

$$s_i = \max_{h,w} s_{h,w}$$

Abschließend wird, wie bereits bekannt, der AUROC über den gesamten Testdatensatz berechnet.

5.3 Ergebnisse und Diskussion der Originalmethde

Abgeschlossen: 01.11. v1

Wie bereits in der Einleitung erwähnt, sind die Ergebnisse der Veröffentlichung sehr vielversprechend. Die Ergebnisse konnten zum größten Teil auch reproduziert werden.
Mit einem AUROC von $99{,}6\%$ auf MVTecAD ist die Methode sehr präzise. Für dieses Ergebnis ist allerdings ein „Early Stopping“ notwendig, was kritisch hinterfragt werden sollte. Es ist durchaus der Fall, dass ein fallender Verlust mit einer gesteigerten Genauigkeit einhergeht. Diese starke Korrelation ist aber nicht immer gegeben. Es ist also nicht zwingend der Fall, dass ein fallender Verlust auch zu einer gesteigerten Genauigkeit führt. In vielen Fällen ist die höchste, während des Trainings erreichte Genauigkeit, nicht die, die am Ende des Trainings erreicht wird. In der Originalimplementierung ist ein Training über 40 Epochen vorgesehen, was bedeutet, dass der Trainigsdatensatz $\mathcal{X}_{train}$ 40 mal durchlaufen wird. Nach jeder dieser Epochen wird das das Modell nicht etwa auf einem Validierungsdatensatz getestet, sondern auf dem gesamten Testdatensatz. Das Modell, welches die höchste Genauigkeit erreicht, wird dann gespeichert und für die finale Inferenz, also die Berechnung des AUROC, verwendet. Dieses Vorgehen ist

fragwürdig, weil der Testdatensatz für solche Zwecke nicht vorgesehen ist. In der Realität steht ein solcher Testdatensatz, der dann zum Beispiel einer produktiven Produktion entspricht, nicht zur Verfügung. Dieses Problem wird zu späterer Stelle noch einmal aufgegriffen.
Mit Hinblick auf die Laufzeit ist die Methode für eine Inferenz ohne GPU wohl kaum schneller als die Methode PatchCore. Ein Unterschied in der Laufzeit kann nur bei den Bausteinen erzeugt werden, die überhaupt unterschiedlich sind. Um eine gegenüber PatchCore gerinigere Laufzeit zu erreichen, muss also der Feature Adaptor oder der Diskriminator schneller sein, als die NN-Suche und das wenig zeitkritische Berechnen des Anomaliegrades. Berachtet man die Ergebnisse der PatchCore Methode, so ist die Laufzeit der NN-Suche und des Berechnens des Anomaliegrades im Verhältnis sehr gering. Es ist also kaum zu erwarten, dass signifikante Verbesserungen in der Laufzeit gegenüber PatchCore erreicht werden können. Dass die Autoren darauf verweisen, ihre Methode sei 8-mal schneller als PatchCore, ist nur vor dem Hintergrund der Verwendung einer GPU zu verstehen. Die Methode SimpleNet kann komplett mithilfe einer GPU ausgeführt werden, weil nur bekannte Feed-Forward-Operationen verwendet werden (CNN, MLP). Weil sich allerdings auch die NN-Suche durch eine GPU enorm beschleunigen würde, worauf auch die Autoren von PatchCore bereits hinwiesen, und in keiner der Versuche zu PatchCore die NN-Suche der laufzeitkritischste Teil war, ist die Aussage der Autoren von SimpleNet für den Autor nur schwer nachvollziehbar.
Die Methode ist dennoch spannend, weil sie sich nicht nur durch einen tatsächlich sehr einfachen Aufbau auszeichnet, sondern auch Potentiale für weitere Forschung bietet. So ist zum Beispiel denkbar, dass durch ein spezifischeres Design der Pseudo-Anomalien, die Ergebnisse noch weiter verbessert werden können. In jedem Fall liefert SimpleNet den Beweis, dass ein solches Vorgehen prinzipiell funktioniert.

Kapitel 6

Zusammenfassung, Fazit & Ausblick

6.1 Zusammenfassung

Hier quasi eine längere und konkretere Fassung vom Abstract.

6.2 Fazit

Ist das Ziel erreicht worden? Welche Erkentnisse konnten gewonnen werden?

6.3 Ausblick

Was sind noch Potentiale, die zu heben sind? Für was ist das ganze alles gut?

Anhang A

ASCII-Tabelle

…

Literaturverzeichnis

[1] Bailer, Christian, Habtegebrial, Tewodros, varanasi, Kiran und Stricker, Didier: *Fast Feature Extraction with CNNs with Pooling Layers*, 2018.

[2] Batzer, Kilian, Heckler, Lars und König, Rebecca: *EfficientAD: Accurate Visual Anomaly Detection at Millisecond-Level Latencies*, 2023.

[3] Bergman, Liron und Hoshen, Yedid: *Classification-Based Anomaly Detection for General Data*, 2020.

[4] Bergmann, Paul, Batzner, Kilian, Fauser, Michael, Sattlegger, David und Steger, Carsten: *The MVTec Anomaly Detection Dataset: A Comprehensive Real-World Dataset for Unsupervised Anomaly Detection*. International Journal of Computer Vision, 129, 04 2021.

[5] Bergmann, Paul, Batzner, Kilian, Fauser, Michael, Sattlegger, David und Steger, Carsten: *Beyond Dents and Scratches: Logical Constraints in Unsupervised Anomaly Detection and Localization*, Februar 2022.

[6] Bergmann, Paul, Fauser, Michael, Sattlegger, David und Steger, Carsten: *Uninformed Students: Student-Teacher Anomaly Detection with Discriminative Latent Embeddings*. CoRR, abs/1911.02357, 2019.

[7] Bishop, Christopher M: *Pattern Recognition and Machine Learning*. Springer, 2006.

[8] Code (Meta AI), Papers with: *SOTA: Anomaly Detection on MVTec AD*. https://paperswithcode.com/sota/anomaly-detection-on-mvtec-ad, 2023. Accessed: 2023-10-09.

[9] Community, SciPy: *scipy.spatial.distance.cdist Dokumentation*. = https://docs.scipy.org/doc/scipy/reference 2023. Accessed: 2023-10-09.

[10] Dasgupta, Sanjoy und Gupta, Anupam: *An elementary proof of a theorem of Johnson and Lindenstrauss*. Random Structures & Algorithms, 22(1):60–65, 2003.

[11] Fawcett, Tom: *Introduction to ROC analysis*. Pattern Recognition Letters, 27:861–874, 06 2006.

[12] Foundation, The Raspberry Pi: *Specifications for the Raspberry Pi 4 Model B*. https://www.raspberrypi.org/documentation/hardware/raspberrypi/bcm2711/rpi_DATA_2711_1p0_preliminary.pdf, 2023. Accessed: 2023-10-09.

[13] Geerling, Jeff: *Power Consumption Benchmarks | Raspberry Pi Dramble*. https://www.pidramble.com/wiki/benchmarks/power-consumption. Accessed: 2023-10-09.

[14] GmbH, MvTec: *MVTec LOCO: The MVTec logical constraints anomaly detection dataset*. https://www.mvtec.com/company/research/datasets/mvtec-loco, 2023.

[15] Ha, Changwoo: *Unofficial implementation of PatchCore*. https://github.com/hcw-00/PatchCore_anomaly_detection, 2022. Accessed: 2023-10-09.

[16] He, Kaiming, Zhang, Xiangyu, Ren, Shaoqing und Sun, Jian: *Deep Residual Learning for Image Recognition*, 2015.

[17] Huang, Gao, Liu, Zhuang, Maaten, Laurens van der und Weinberger, Kilian Q.: *Densely Connected Convolutional Networks*, 2018.

[18] James, Gareth, Witten, Daniela, Hastie, Trevor und Tibshirani, Robert: *An Introduction to Statistical Learning: With Applications in R*. Springer Publishing Company, Incorporated, 2014.

[19] Johnson, Jeff, Douze, Matthijs und Jégou, Hervé: *Billion-scale similarity search with GPUs*. IEEE Transactions on Big Data, 7(3):535–547, 2019.

[20] Liu, Zhikang, Zhou, Yiming, Xu, Yuansheng und Wang, Zilei: *SimpleNet: A Simple Network for Image Anomaly Detection and Localization*, 2023.

[21] Mahalanobis, Prasanta Chandra: *On the generalized distance in statistics*. Proceedings of the National Institute of Sciences (Calcutta), 2:49–55, 1936.

[22] Maine, The Weaver Computer Engineering Research Group of University of: *FLOPS/Watt of Various CPUs*. https://web.eece.maine.edu/~vweaver/group/green_machines.html, 2023. Accessed: 2023-10-09.

[23] Mehta, Sachin und Rastegari, Mohammad: *MobileViT: Light-weight, General-purpose, and Mobile-friendly Vision Transformer*, 2022.

[24] Paszke, Adam, Gross, Sam, Massa, Francisco, Lerer, Adam, Bradbury, James, Chanan, Gregory, Killeen, Trevor, Lin, Zeming, Gimelshein, Natalia, Antiga, Luca, Desmaison, Alban, Köpf, Andreas, Yang, Edward, DeVito, Zach, Raison, Martin, Tejani, Alykhan, Chilamkurthy, Sasank, Steiner, Benoit, Fang, Lu, Bai, Junjie und Chintala, Soumith: *PyTorch: An Imperative Style, High-Performance Deep Learning Library*, 2019.

[25] PC Games Hardware, Maximilian Holm von: *RTX 4090: Mit starker Übertaktung mehr als 100 TFLOPS Rechenleistung erreicht*. https://www.pcgameshardware.de/Grafikkarten-Grafikkarte-97980/News/RTX-4090-Mit-starker-Uebertaktung-mehr-als-100-TFLOPS-Rechenleistung-erreicht-1405138/, 2022. Accessed: 2023-10-09.

[26] Roth, Karsten, Pemula, Latha, Zepeda, Joaquin, Schölkopf, Bernhard, Brox, Thomas und Gehler, Peter: *Towards Total Recall in Industrial Anomaly Detection*, 2022.

[27] Rudolph, Marco, Wehrbein, Tom, Rosenhahn, Bodo und Wandt, Bastian: *Asymmetric Student-Teacher Networks for Industrial Anomaly Detection*, 2022.

[28] Sener, Ozan und Savarese, Silvio: *Active Learning for Convolutional Neural Networks: A Core-Set Approach*, 2018.

[29] Tan, Mingxing und Le, Quoc V.: *EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks*, 2020.

[30] Touvron, Hugo, Cord, Matthieu, Douze, Matthijs, Massa, Francisco, Sablayrolles, Alexandre und Jégou, Hervé: *Training data-efficient image transformers and distillation through attention*, 2021.

[31] Touvron, Hugo, Cord, Matthieu, Sablayrolles, Alexandre, Synnaeve, Gabriel und Jégou, Hervé: *Going deeper with Image Transformers*, 2021.

[32] Vaswani, Ashish, Shazeer, Noam, Parmar, Niki, Uszkoreit, Jakob, Jones, Llion, Gomez, Aidan N., Kaiser, Lukasz und Polosukhin, Illia: *Attention Is All You Need*, 2023.

[33] Wikipedia contributors: *Raspberry Pi 4 — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=Raspberry_Pi_4&oldid=1178956001, 2023. [Online; accessed 9-October-2023].

[34] Xie, Saining, Girshick, Ross, Dollár, Piotr, Tu, Zhuowen und He, Kaiming: *Aggregated Residual Transformations for Deep Neural Networks*, 2017.

[35] Yan, Ruiqing, Zhang, Fan, Huang, Mengyuan, Liu, Wu, Hu, Dongyu, Li, Jinfeng, Liu, Qiang, Jiang, Jinrong, Guo, Qianjin und Zheng, Linghan: *CAINNFlow: Convolutional block Attention modules and Invertible Neural Networks Flow for anomaly detection and localization tasks*, 2022.

[36] Yu, Jiawei, Zheng, Ye, Wang, Xiang, Li, Wei, Wu, Yushuang, Zhao, Rui und Wu, Liwei: *FastFlow: Unsupervised Anomaly Detection and Localization via 2D Normalizing Flows*, 2021.

[37] Zagoruyko, Sergey und Komodakis, Nikos: *Wide Residual Networks*, 2017.