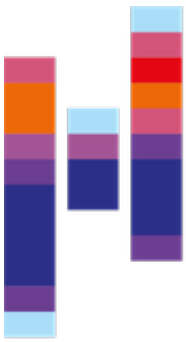

Hochschule Flensburg
University of Applied Sciences

Bewegungserkennung auf mobilen Geräten mit Verwendung von GANs für eine automatische Datensatzgenerierung

Master-Thesis

Florian Hansen
Hochschule Flensburg

2. September 2021

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

Wie können komplexe Machine-Learning-Modelle effizient und in Echtzeit auf mobilen Geräten wie Smartphones ausgeführt werden? Mit dieser Frage soll sich diese Arbeit beschäftigen. Speziell wird sich dabei auf die Erkennung und Analyse von Bewegungen bezogen. Dabei müssen bereits vorhandene Modelle umgewandelt werden, um auf mobilen Geräten eine Echtzeiterkennung durchführen zu können. Künstliche Intelligenz hat bereits in vielen verschiedenen Bereichen eine unterstützende Rolle eingenommen. Dementsprechend ist das Feld in den letzten Jahren stetig gewachsen und hat an Interesse gewonnen. Viele Anwendungen funktionieren nur deshalb, weil sie durch Modelle des Machine-Learnings unterstützt werden. Vor allem in der Computer-Vision findet diese Technologie Anwendung. Beispiele hierfür sind Bildklassifizierer und Objekt-Detektoren, die entsprechend Bilder eine Klasse zuordnen bzw. viele Objekte innerhalb eines Bildes erkennen. Neben der Bildverarbeitung ist die Erkennung von menschlichen Posen bzw. von Bewegungen mit künstlichen neuronalen Netzen (KNN) ein weiteres, aktuelles Forschungsthema. Diese Art von Detektoren werden unter anderem dazu verwendet, um Schlüsselpunkte des menschlichen Körpers zu identifizieren.

Während solche Modelle bereits im Desktopbereich mit weniger Einschränkungen ausgeführt werden können, sind diese eher schwierig auf ressourcenarme Geräte übertragbar. Oft müssen abgewandelte, verkürzte Varianten erstellt werden, um die benötigte Rechenleistung so gering wie möglich zu halten – die meisten Smartphones haben zur Zeit leider nicht die gleichen Rechen- und Speicherkapazitäten wie die meisten Desktopmaschinen, ganz zu schweigen von diversen anderen Geräten des Internet-of-Things (IoT) wie Haushaltsgeräte und Sensoren. Aus diesem Grund soll sich diese Arbeit insbesondere damit beschäftigen, wie die Bewegungserkennung auf mobilen Geräten ausgeführt werden kann. Zusätzlich wird untersucht, welche Anpassungen vorhandene Machine-Learning-Modelle benötigen, um auf mobile Geräte ausgeführt werden zu können.

Kapitel 2 beschäftigt sich mit den Grundlagen der in dieser Arbeit verwendeten Technologien. Dabei wird unter anderem darauf eingegangen, wie Unterschiede zwischen

Distributionen gemessen werden können, um damit den Grundstein für spätere Loss-Funktionen zu schaffen. Diese werden dann vor allem für das Trainieren von Generative-Adversarial-Networks (GANs) verwendet. Auch werden in diesem Kapitel einige Grundbausteine zum Entwickeln von sehr tiefen neuronalen Netzen besprochen. Anschließend wird die Funktionsweise von GANs und entsprechenden Verlustfunktionen zum Trainieren dieser Netzwerke besprochen. Zusätzlich werden Probleme der einzelnen Architekturen besprochen und Lösungen vorgestellt.

Kapitel 3 stellt Methoden vor, die zum Erstellen eines Datensatzes verwendet werden. Dieser Datensatz wird anschließend verwendet, um die Bewegungserkennung aus dem nächsten Kapitel zu implementieren und die Modelle damit zu trainieren. Besonders wird in diesem Kapitel der Aufbau des Datensatzes erläutert und inwiefern dieser mithilfe von GANs erweitert werden kann. Insbesondere wird hier versucht, einen kompletten Datensatz mithilfe eines GANs zu erzeugen, sodass dieses dynamisch anstelle eines echten Datensatzes verwendet werden kann.

Kapitel 4 führt die Bewegungserkennung ein und vergleicht unter anderem verschiedene Ansätze zum Erkennen von menschlichen Schlüsselpunkten. Dabei wird auf Single- und Multi-Pose-Detection eingegangen und mit dessen Hilfe eine neue Netzwerkarchitektur definiert, die in der Lage ist, eine Folge von menschlichen Posen zu klassifizieren und analysieren.

2 Grundlagen

In diesem Kapitel sollen die Grundlagen, die in dieser Arbeit benötigt werden, besprochen werden und bildet den Theorieteil der Thesis. Zuerst wird auf die mathematischen Grundlagen eingegangen, die zum Definieren von Verlustfunktionen für generative Machine-Learning-Modelle essentiell sind. Ziel dieses Teils ist es die Frage zu beantworten, wie weit eine Wahrscheinlichkeitsverteilung von einer anderen entfernt ist, um die Distanz anschließend zu minimieren. Das Minimieren der Entfernung zwischen Verteilungen wird in darauf folgenden Abschnitten verwendet, um Generative-Adversarial-Networks zu trainieren.

Neben den mathematischen Grundlagen werden einige grundlegende Bausteine für die Entwicklung von hochqualitativen Feature-Extraktoren für die Objekterkennung auf mobilen Endgeräten erläutert. Diese werden in späteren Kapiteln verwendet, um neue neuronale Netzwerke zu entwickeln, die unter anderem menschliche Posen aus Bildern sehr genau extrahieren und letztlich Bewegungen erkennen können. Hierfür ist eine hohe Semantik bei einer hohen Qualität bzw. Auflösung unabdingbar, um auch kleine Merkmale wie Hände erkennen zu können. Deshalb werden Feature-Pyramid-Networks (FPNs) erläutert, die mithilfe eines sogenannten Backbones hochauflösende Features mit einer hohen Semantik extrahieren können während einfache Faltungsschichten (Convolutional-Layer) zwar ein hohes Verständnis entwickeln, jedoch eine kleine Auflösung dieser Features besitzen.

Neben den FPNs soll auch die Idee hinter Residual-Neural-Networks (ResNets) erläutert werden.

2.1 Notationen

In dieser Arbeit werden verschiedene Notationen aus der Statistik und dem Machine-Learning-Umfeld verwendet und sollen hier aufgrund der Les- und Verständlichkeit aufgelistet werden.

Erwartungswert. Der Term $\mathbb{E}_{x\sim P}[f(x)]$ stellt den Erwartungswert einer Verteilung P dar und liest sich als *erwarteter Wert von $f(x)$ unter x verteilt als P.*

Berechnung von Gradienten. Der Term $\nabla_w[f(x)]$ stellt die Berechnung der Gradienten von den Parametern w mithilfe der Loss-Funktion f dar.

Euklidische Norm. Der Term $\|v\|$ stellt die euklidische Norm von v dar. Sie ist definiert als die Summer aller Quadrate der Komponenten von v, also $\|v\| = \|v\|_2 = \sqrt{v_1^2 + v_2^2 + ... + v_n^2}$.

2.2 Lipschitzstetigkeit

Definition 1 (K-Lipschitzstetigkeit) *Seien (X, d_X) und (Y, d_Y) metrische Räume. Eine Abbildung $f : X \to Y$ wird als K-lipschitzstetig bezeichnet, wenn*

$$d_Y(f(x_1), f(x_2)) \leq K \cdot d_X(x_1, x_2)$$

für alle $x_1, x_2 \in X$ gilt. K wird hierbei als Lipschitzkonstante bezeichnet und muss immer $K \geq 0$ erfüllen.

2.3 Kullback-Leibler-Divergenz

Die Kullback-Leibler-Divergenz (KL-Divergenz) misst, wie sehr sich zwei Verteilungen voneinander unterscheiden und hat seinen Ursprung in der Informationstheorie.

Definition 2 (Kullback-Leibler-Divergenz [1]) *Seien P und Q zwei Wahrscheinlichkeitsfunktionen über den gleichen Wahrscheinlichkeitsraum X. Dann ist der Abstand bzw. die Divergenz der beiden Verteilungen definiert als*

$$D_{KL}(P||Q) = \sum_{x\in X} P(x) \log \frac{P(x)}{Q(x)}.$$

Dabei gibt $P||Q$ eine Divergenz von der Ausgangsverteilung P zur Zielverteilung Q an. Das Messen der Divergenz zwischen zwei Wahrscheinlichkeitsverteilungen findet insbesondere im Machine-Learning statt, um künstliche neuronale Netze und ihre Ge-

wichte zu trainieren. Deshalb kann die KL-Divergenz auch als Loss-Funktion verwendet werden. Bemerkenswert ist hierbei, dass die KL-Divergenz asymmetrisch ist, also $D_{KL}(P||Q) \neq D_{KL}(Q||P)$. Die Distanz zwischen zwei Verteilungen unterscheidet sich demnach je nach Ausgangsverteilung.

2.4 Jensen-Shannon-Divergenz

Definition 3 (Jensen-Shannon-Divergenz [1]) *Seien P und Q zwei Wahrscheinlichkeitsfunktionen über den gleichen Wahrscheinlichkeitsraum X. Dann ist die Jensen-Shannon-Divergenz der beiden Verteilungen definiert als*

$$D_{JS}(P||Q) = \frac{1}{2}D_{KL}(P||M) + \frac{1}{2}D_{KL}(Q||M) \qquad mit \;\; M = \frac{1}{2}(P+Q)$$

Die Jensen-Shannon-Divergenz kann als Erweiterung der Kullback-Leibler-Divergenz angesehen werden. Im Gegensatz zur Kullback-Leibler-Divergenz ist die Jensen-Shannon-Divergenz (JS-Divergenz) symmetrisch. Das bedeutet, dass der Abstand zwischen zwei Wahrscheinlichkeitsverteilungen gleich groß ist, egal von welchen er beiden Distributionen aus betrachtet wird.

2.5 Wasserstein-Abstand

Eine weitere Methode zum Messen des Abstands zwischen zwei Wahrscheinlichkeitsverteilungen ist die Berechnung des Wasserstein-Abstands. Diese Methode wird besonders wichtig für die folgenden Abschnitten, in denen generative neuronale Netze mithilfe des Wasserstein-Abstandes definiert und umgesetzt werden.

Definition 4 (Wasserstein-Abstand [1]) *Seien P_r und P_g zwei Wahrscheinlichkeitsverteilungen, dann ist der Wasserstein-Abstand definiert als*

$$W(P_r, P_g) = \inf_{\gamma \in \Pi(P_r, P_g)} \mathbb{E}_{(x,y)\sim\gamma} \left[\|x - y\| \right],$$

wobei $\Pi(P_r, P_g)$ die Menge aller gemeinsamen Verteilungen $\gamma(x, y)$ darstellt, dessen Grenzen P_r und P_g sind.

Der Term $\gamma(x, y)$ stellt dabei die *Masse* dar, die von x nach y transportiert wird, um

schließlich die Verteilung P_r in die Verteilung P_g umzuformen. Aus diesem Grund ist der Wasserstein-Abstand auch als *Earth-Mover-Abstand* (EM-Abstand) bekannt.

2.6 Residual-Neural-Networks

Mit [7] wurde ein neues Framework zum Trainieren von tiefen neuronalen Netzen vorgestellt. Ein großes Problem mit sehr tiefen neuronalen Netzen ist, dass diese zu einem größeren Fehler im Training führen. Dies führt ebenfalls zu einem erhöhten Test-Fehler. Diese Fehler werden überraschenderweise immer größer, je tiefer das Netzwerk ist und das Phänomen wird als *Degradation* bezeichnet. Dies ist auch bei der Genauigkeit solcher Netze beobachtbar. Ist die Genauigkeit gesättigt und erhöht man nun die Tiefe des Netzes, so degradiert die Genauigkeit schnell. Laut [7] liegt dies nicht am Overfitting (Überanpassung). Overfitting beschreibt die Überspezialisierung eines Machine-Learning-Modells, welches sich zu sehr an den dahinterliegenden Datensatz angepasst hat. Ein solches Modell kann nicht mehr zuverlässig mit Daten außerhalb des zum Trainieren verwendeten Datensatzes betrieben werden. Anders ausgedrückt besitzt das Netzwerk eine sehr hohe Genauigkeit beim Arbeiten mit dem Trainingsdatensatz, aber eine signifikant niedrigere Genauigkeit beim Arbeiten mit dem Testdatensatz.

Abhilfe für Degradation sollen die Residual-Networks (ResNets) mithilfe von *Building-Blocks* schaffen. Anstatt die Abbildung $F(x)$ bei normalen gestapelten Schichten zu optimieren, wird bei ResNet das Residuum $F(x) + x$ modelliert. Dies kann mithilfe von Shortcut-Connections (Abkürzungsverbindungen) realisiert werden (siehe Abbildung 2.1).

Abhängig von den Anforderungen können nun entsprechend tiefe Netze eingesetzt werden, indem die Building-Blocks gestapelt eingesetzt werden. In [7] werden außerdem ein paar Architekturen vorgestellt, die erfolgreich auf den ImageNet-Datensatz [3] evaluiert wurden. Diese unterscheiden sich hauptsächlich in der Anzahl der verwendeten Schichten und sind in Tabelle 2.1 zu sehen. Jeder dieser Residual-Blocks, egal ob einfacher Building- oder Bottleneck-Block, kann einen Stride besitzen. Dabei ist wichtig, dass der entsprechende Stride nur auf die erste Faltungsschicht des gesamten Blocks angewandt wird. Die restlichen Schichten besitzen demnach immer einen Stride von 1. Die Ergebnisse nach dem Trainieren dieser Netze haben gezeigt, dass das Degradation-Problem gelöst werden konnte, sodass tiefere Netze mithilfe von Residual-Blocks auch einen geringeren Fehler erzeugen, also genauer arbeiten als weniger tiefe Netze.

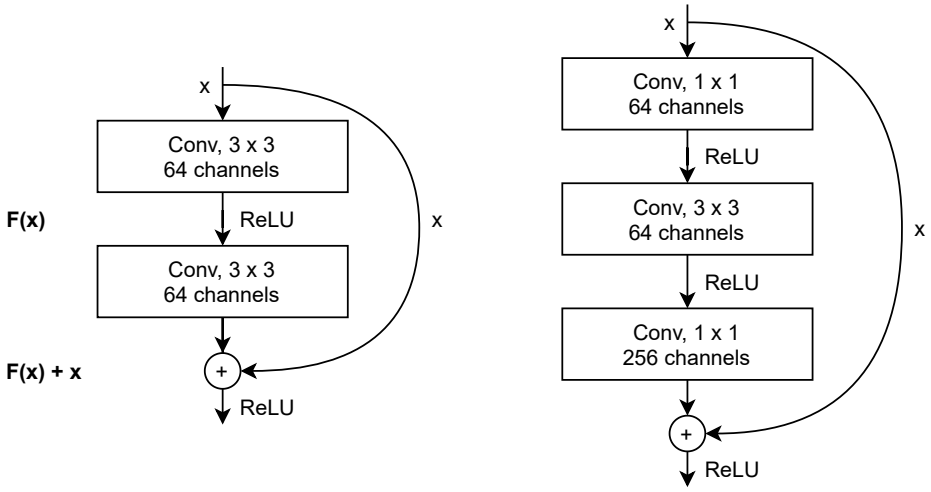

Abbildung 2.1: Schemata für Building-Blocks eines ResNets [7]. Die Shortcut-Connection ist die Identität von dem Eingabeparameter x. Links ist ein einfacher Residual-Block dargestellt während rechts ein tieferer, soganannter Bottleneck-Block dargestellt wird.

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
conv2_x	56×56	3×3, max pool, stride 2				
		$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix} \times 2$	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix} \times 3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix} \times 3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix} \times 3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix} \times 3$
conv3_x	28×28	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix} \times 2$	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix} \times 4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix} \times 4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix} \times 4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix} \times 8$
conv4_x	14×14	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix} \times 2$	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix} \times 6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix} \times 6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix} \times 23$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix} \times 36$
conv5_x	7×7	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix} \times 2$	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix} \times 3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix} \times 3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix} \times 3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix} \times 3$
	1×1	average pool, 1000-d fc, softmax				
FLOPs		$1{,}8{\cdot}10^9$	$3{,}6{\cdot}10^9$	$3{,}8{\cdot}10^9$	$7{,}6{\cdot}10^9$	$11{,}3{\cdot}10^9$

Tabelle 2.1: Architekturen für ImageNet [7]. Es werden die gestapelten Schichten bzw. Building-Blocks für ResNet-18, -34, -50, -101 und -152 aufgelistet. Die Building-Blocks werden hier als Matrix dargestellt und geben pro Zeile die Kernelgröße und Anzahl der Kanäle der Faltungsschichten an. Das Multiplikationszeichen hinter den Matrizen gibt die Anzahl der verketteten Building-Blocks an.

2.7 MobileNetV2

Dieser Abschnitt soll nun eine Architektur vorstellen, die geeignet ist, um Modelle des maschinellen Lernens auf mobilen Geräten auszuführen. Diese soll als Grundlage zum Verständnis der MoveNet-Architektur aus [20] dienen, die später für die Bewegungserkennung wichtig wird. Die Rede ist hierbei von MobileNetV2 [15] und den damit eingeführten Inverted-Residual-Blocks. Diese besitzen einen verringerten Speicherverbrauch bei einer Inferenz, was für mobile Anwendungen sehr wichtig ist und sind wie folgt aufgebaut. Die Eingabe x in einem solchen Block wird zuerst in eine Faltungsschicht mit einem 1×1 Kernel und einer ReLU6-Aktivierung gegeben. Danach wird eine Depthwise-Separable-Convolution [8] durchgeführt, die eine Kernelgröße von 3×3 verwendet. Auch hier wird eine ReLU6-Aktivierung vorgenommen. Die nächste Schicht ist wieder eine 1×1 Faltungsschicht mit keiner, also einer linearen Aktivierung. Zum Schluss wird die Ausgabe der letzten Faltungsschicht mit der Identität der Eingabe x addiert und bildet ein Residuum. Der gesamte Ablauf ist in Abbildung 2.2 zu sehen und ähnelt sehr stark den Residual-Blocks aus Abschnitt 2.6. Neben den üblichen Parametern wurden ein paar wenige hinzugefügt. So beschreibt der Expansionsfaktor t einen Skalar, der die Anzahl der internen Filter des Blocks skaliert. Dieser hat keinen Einfluss auf die Eingabe- oder Ausgabegröße Das beudetet bei einer Eingabegröße von $h \times w \times c$ besitzen die erste und zweite Faltungsschicht $c \cdot t$ Ausgabefilter. Auf die dritte und letzte Schicht eines Inverted-Residual-Blocks hat der Expansionsfaktor keinen direkten Einfluss. Hier wird lediglich von $c \cdot t$ auf c' Filter projeziert. Bei dem Stride eines kompletten Blocks verhält es sich ähnlich. Dieser wird lediglich auf die zweite also Depthwise-Separable-Convolution-Schicht angewandt. Die restlichen Schichten besitzen einen Stride von 1.

Der Aufbau von MobileNetV2 kann nun einfach erläutert werden. Die erste Schicht ist eine Faltungsschicht mit 32 Filtern gefolgt von 17 Inverted-Residual-Blocks und einer weiteren Faltungsschicht mit 1280 Filtern. Der gesamte Aufbau soll durch Tabelle 2.2 dargestellt werden. Wie schon eingangs erwähnt, wird dieses Modell später für die Bewegungs- bzw. Posenerkennung verwendet. Genauer gesagt wird das MoveNetV2 als Backbone für einen Feature-Extraktor verwendet. Dies soll jedoch in dem nächsten Abschnitt näher erläutert werden.

Input	Layer	t	c	n	s
$224 \times 224 \times 3$	Conv2D, 3×3	-	32	1	2
$112 \times 112 \times 32$	Bottleneck	1	16	1	1
$112 \times 112 \times 16$	Bottleneck	6	24	2	2
$56 \times 56 \times 24$	Bottleneck	6	32	3	2
$28 \times 28 \times 32$	Bottleneck	6	64	4	2
$14 \times 14 \times 64$	Bottleneck	6	96	3	1
$14 \times 14 \times 96$	Bottleneck	6	160	3	2
$7 \times 7 \times 160$	Bottleneck	6	320	1	1
$7 \times 7 \times 320$	Conv2D, 1×1	-	1280	1	1

Tabelle 2.2: Aufbau von MobileNetV2 nach [15]. Der Bottleneck-Block entspricht einem Inverted-Residual-Block (siehe Abbildung 2.2). Der Parameter t beschreibt den Skalierungsfaktor, c die Anzahl der Filter, n die Anzahl der aneinander geketteten Blöcke und s den Stride eines Block-Verbundes mit n Blöcken.

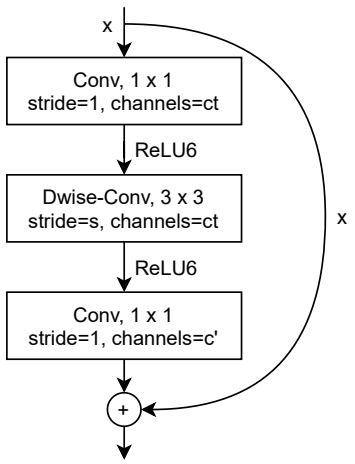

Abbildung 2.2: Darstellung eines Inverted-Residual-Blocks. Der *Dwise*-Block stellt eine Depthwise-Separable-Convolution aus [8] dar. Der gesamte Block wird auch als Bottleneck-Block in der MobileNetV2-Architektur genannt. Nicht zu verwechseln mit dem Bottleneck-Block aus dem ResNet-Kontext.

2.8 Feature-Pyramid-Networks

Bei der Objekterkennung ist beim Erlernen der Features eines Bildes die Auflösung dieser Features sehr gering. Feature-Pyramid-Networks (FPN) haben die Aufgabe, eine hohe Semantik bei einer hohen Auflösung zu generieren. Häufig sind diese Netzwerke nur Teil eines Backbones bzw. werden dahinter geschaltet. Als Backbone-Modell kann zum Beispiel AlexNet, MobileNet und ResNet dienen. In Kombination mit einem FPN werden diese Netze damit zu einem Feature-Extractor umgewandelt, der eine hohe Semantik bei einer hohen Auflösung der Features erlernen kann. Der Weg von der Eingabe über das Backbone-Modell wird auch als *Bottom-Up-Pathway* bezeichnet, während der Weg vom Backbone über die Feature-Pyramid als *Top-Down-Pathway* bezeichnet wird [10].

Anfänglich wurden FPNs in Verbindung mit ResNet-Backbones eingeführt (siehe Abbildung 2.3). Die Eingabe ist ein $224 \times 224 \times 3$ Bild und wird mithilfe der ersten ResNet-Schicht (C1) in $112 \times 112 \times 64$ Filter mithilfe eines Convolutional-Layers mit einem Stride von 2 und einem 7×7 Kernel umgeformt. Anschließend wird die Ausgabe in ein Max-Pooling-Block gegeben, welcher die Eingabe in $56 \times 56 \times 128$ Filter umwandelt. Die nächsten Blöcke sind für die Einbettung von FPNs am wichtigsten. Der folgende Block (C2) besteht aus mehreren Residual-Blocks, welche zusammen einen Stride von 1 ergeben und 256 Filter erzeugen. Diese Blöcke werden zusammengefasst auch *Bottlenecks* genannt. Darauf folgen drei weitere Bottleneck-Blöcke (C3, C4, C5) mit einem Stride von 2. Nach C5 ensteht somit eine Ausgabe von $7 \times 7 \times 2048$. Soweit zum Aufbau des Bottom-Up-Pathways. Der Top-Down-Pathway wird mithilfe von Verbindungsschichten (Laterals) mit dem Backbone verbunden. Diese haben die Aufgabe, die Anzahl der Filter aus dem Bottom-Up-Pathway so umzuformen bzw. anzugleichen, sodass diese im Top-Down-Pathway miteinander addiert werden können. Hierfür werden Convolutional-Layer mit einem 1×1 Kernel und 256 Filtern verwendet, sodass lediglich die Anzahl der Filter transformiert werden. Im Konkreten bedeutet dies, dass die Ausgabe von C5 von $7 \times 7 \times 2048$ auf die Form $7 \times 7 \times 256$ gebracht wird. Die Größe dieser Ausgabe wird nun mithilfe von Upsampling (M5) verdoppelt und mit der Ausgabe der Lateralverbindung von C4 addiert (M4). Dies wird mit den übrigen Lateralverbindungen und Bottleneck-Blöcken verkettet wiederholt, sodass das FPN schließlich vier Ausgaben mit den Größen $56 \times 56 \times 256$, $28 \times 28 \times 256$, $14 \times 14 \times 256$ und $7 \times 7 \times 256$ (M5, M4, M3, M2) ausgibt. Betrachtet man nun die letzte Ausgabe M5 relativ zum Eingabeformat, so besitzt diese Architektur einen Stride von 4. Das Problem beim Vergrößern der Filter ist, dass dabei ein Alias-Effekt auftritt, das Bild also verschwommen wirkt. Hierfür die-

nen Smoothing-Layer, welche wiederum nichts weiter als Convolutional-Layer mit einem 3×3 Kernel und einem Stride von 1 sind. Entsprechend werden die Ausgaben M5 - M2 verschärft und ergeben Features mit einer hohen Auflösung und einer hohen Semantik, die nun wahlweise für die Objekterkennung verwendet werden können.

Ähnlich verhält sich der Vorgang beim Verwenden eines MobileNetV2-Backbones, wie in Abbildung 2.4 dargestellt. Da das MobileNetV2 mehr Blöcke als das ResNet besitzt, die Anzahl der Pyramidenstufen aber gleich bleiben soll, werden einige Blöcke übersprungen und sind nicht direkt Teil des Top-Down-Pathways. Im Prinzip werden lediglich die Blöcke mit Lateralverbindungen mit dem Top-Down-Pathway verknüpft, welche die letzte Stufe vor der Halbierung der Filtergröße darstellen. Also immer dann, wenn der Stride 2 ist. Ausnahme ist C5. Hier wird einfach die letzte Schicht des Netzwerks verwendet. Der Einbau deines FPN erfolgt ansonsten wie vorher erläutert. Die Kombination mit MobileNetV2 und Feature-Pyramid-Network ist in dieser Arbeit deshalb wichtig, da dieser Feature-Extraktor mit für die Posenerkennung zuständig ist. Besonders, weil die Features eine hohe Semantik und eine hohe Auflösung besitzen, können besonders kleine Objekte innerhalb eines Bildes erkannt und benötigte Informationen extrahiert werden. In Kapitel 4 wird die Posenerkennung genauer erläutert.

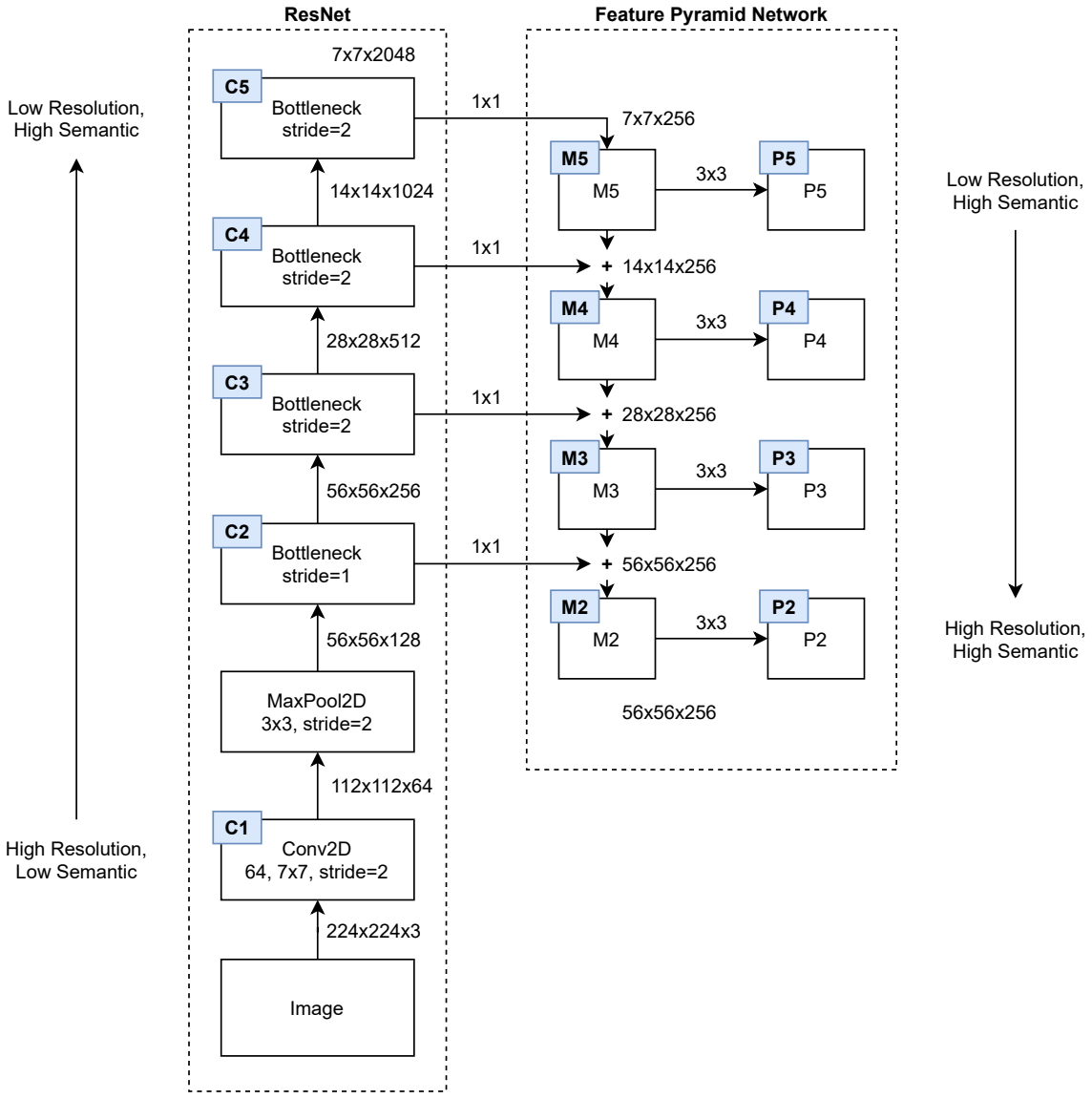

Abbildung 2.3: Architektur eines Feature-Extractors als Feature-Pyramid-Network mit ResNet als Backbone.

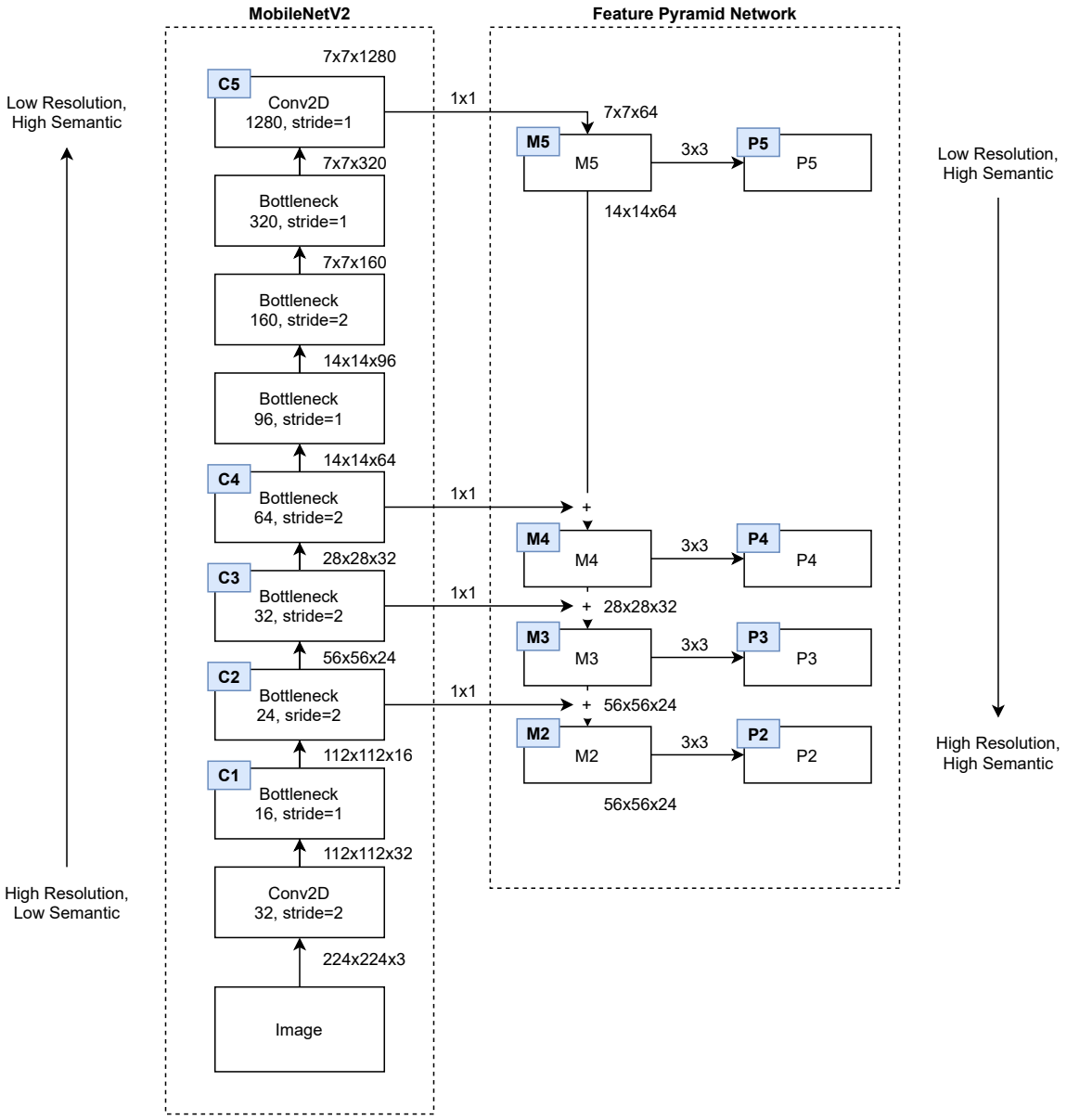

Abbildung 2.4: Architektur eines Feature-Extractors als Feature-Pyramid-Network mit MobileNetV2 als Backbone.

2.9 Generative-Adversarial-Networks

In Machine-Learning existieren viele verschiedene Modelle, die vorhandene Datensätze analysieren und anhand der Daten lernen, Strukturen in den Datensätzen zu erkennen. Besitzt man beispielsweise einen Datensatz bestehend aus Fotoaufnahmen von Tieren, so kann ein Klassifizierer trainiert werden, um einem Bild eine Tierklasse zuzuweisen. Aus diesem Grund fässt man diese Modelle unter dem Begriff *Bildklassifizierung* zusammen.

Wesentlich interessanter ist das Erkennen von vielen Objekten innerhalb eines Bildes, anstatt das gesamte Bild nur einer einzigen Klasse zuzuweisen. In der *Objekterkennung* entwickelt man Modelle, welche mehr als nur eine Klasse erkennen können. Sie liefern zusätzlich zu den erkannten Klassen ihre Position und Größe innerhalb des Bildes. Diese Modelle treffen also keine Aussage über das Gesamtbild, sondern treffen Aussagen über einzelne Objekte innerhalb des Bildes.

Neben Modellen, die zu einem bestimmten Sachverhalt eine Aussage treffen können, existieren auch Modelle, welche in der Lage sind, neue Sachverhalte zu erzeugen. Diese fallen unter dem Begriff *Generative Adversarial Networks* (GANs) und bilden das Hauptthema dieses Abschnitts. Das interessante an diesen generativen Modellen ist, dass sie nicht nur die Strukturen eines Datensatzes lernen, sondern darüber hinaus neue Elemente der Ausgangsdistribution erzeugen können. Trainiert man also ein generatives Modell auf einen Datensatz, welcher Bilder von verschiedenen Tieren enthält, können neue Bilder der gleichen Art erzeugt werden.

Aber nicht nur zum Erzeugen von Bildern kann diese Art von Modellen verwendet werden. Auch bei Aufgaben, bei denen eine Voraussagung getroffen werden soll, werden generative Modelle eingesetzt. Beispielsweise wurde in [2] gezeigt, wie zu bereits getätigten menschlichen Bewegungen unterschiedliche, darauf folgende Bewegungssequenzen aussehen können. Hier hat man also versucht, eine Vorhersage zur Entwicklung von menschlichen Bewegung zu tätigen.

Die Funktionsweise von GANs ist im Prinzip ziemlich simpel. Während beim klassischen supervised-learning in der Regel nur ein Modell beim Training involviert ist, verhält sich das bei generativen Modellen etwas anders. Zum Einen wird ein Generator definiert, welcher, wie sein Name andeutet, Ausgaben selbst erzeugt. Zum Anderen wird ein Diskriminator in das Training eingebaut, welcher zwischen künstlich erzeugten und reellen Daten unterscheidet. Diese beiden Modelle werden dann gleichermaßen trainiert. Während der Generator versucht, Fälschungen immer genauer zu erzeugen, versucht

der Diskriminator immer besser zwischen Fälschung und Realität zu unterscheiden. Die Ausgabe des Diskriminators ist dementsprechend entweder 0 für Fälschung und 1 für Realität. Mit anderen Worten, die beiden Komponenten spielen Spiel, in welchem die eine Partei versucht, die andere zu täuschen [5].

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

Der Eingabeparameter $z \in \mathbb{R}^n$ stellt einen n-dimensionalen Vektor dar, welcher auch als latenter Vektor bezeichnet wird. Dementsprechend wird der Vektorraum auch als latenter Raum bezeichnet. Ein solcher Vektor repräsentiert verschiedenste Objekte mithilfe seiner n Komponenten. In Bezug zu GANs wurde festgestellt, dass ein GAN nicht nur lernt, ähnliche Daten zu einem bestehenden Datensatz zu erzeugen, sondern auch den Zusammenhang zwischen latenten Komponenten und einer Ausgabe zu verstehen. Das schöne daran ist, dass diese latenten Vektoren mithilfe der Gesetze aus der linearen Algebra analysiert und entsprechende Operationen mit ihnen ausgeführt werden können. Hierzu kann folgendes, sehr einfaches Experiment durchgeführt werden. Man wählt zwei latente Vektoren $a, b \leftarrow \mathbb{R}^n$, wobei die Komponenten dieser Vektoren gleich sind, also $a_i = b_i,\ 0 \leq i < n$. Nun wählt man einen zufälligen Komponentenindex $j \in \mathbb{N}$ mit $j \in [0, n[$ und wählt einen zufälligen Wert für die Komponenten beider Vektoren $a_j \leftarrow \mathbb{R},\ b_j \leftarrow \mathbb{R}$, wobei $a_j \neq b_j$. Betrachtet man nun die Ausgaben des Generators mit a und b als Eingabe, dann unterscheiden sich diese um die Eigenschaft, die durch die Komponente an der Stelle j beeinflusst wird. Das gleiche Prinzip kann auch rückwärts durchgeführt werden. Erzeugt der Generator zum Beispiel ein Bild, worauf ein Gesicht mit Brille zu sehen ist und ein weiteres Bild mit Gesicht ohne Brille, dann kann man die beiden Eingabevektoren voneinander subtrahieren und damit die Komponenten herausfinden, die für die entsprechenden Eigenschaften zuständig sind (in diesem Fall, ob das Gesicht eine Brille trägt oder nicht).

Im Verlauf des Trainings entwickelt sich damit ein Generator, welcher im Idealfall so gute Fälschungen erzeugt, sodass sich diese nicht mehr von Daten der Ausgangsdistribution unterscheiden lassen. Der Diskriminator kann hier bestenfalls nur raten, kann also eine Genauigkeit von höchstens 50% erreichen. Ist dies nicht der Fall, d.h. der Diskriminator kann Fälschungen mit einer höheren Wahrscheinlichkeit von realen Daten unterscheiden, so entsteht ein Ungleichgewicht. Aus diesem Grund sollten die Lernparameter sorgfältig ausgewählt und untersucht werden, damit ein stabil laufendes GAN trainiert wird.

2.9.1 Das Mode-Collapse-Problem

Ein großes Problem beim Trainieren von generativen neuronalen Netzen ist, dass sich der Generator sehr häufig auf bestimmte Merkmale der Ausgangsdistribution des Datensatzes fixiert. Das Ergebnis sind signifikant erhöht wiederkehrende Ergebnisse, die sich kaum bis gar nicht von anderen Ausgaben unterscheiden. Man erwartet jedoch, dass das jeweilige GAN eine vielseitige Variation aus allen Elementen des Datensatzes erzeugt. Mit anderen Worten, bei einer zufälligen Eingabe in das Netz, soll immer eine unterschiedliche Ausgabe erzeugt werden. Bei einem Mode-Collapse ist dies nicht der Fall. Es kann beispielsweise passieren, dass wenn das Netz auf das Erzeugen von neuen Gesichtern trainiert wird, dass dieses ausschließlich weibliche Gesichter erzeugt, weil das Netz herausgefunden hat, dass es einfacher ist, weibliche Gesichtszüge zu generieren, als männliche [14]. Dies lässt sich damit erklären, dass der Generator beim Trainingsvorgang mehr Erfolg beim Generieren von weiblichen Gesichtern hatte und der Diskriminator es schwerer hatte, Fälschung von Realität zu unterscheiden. Um das Problem zu beseitigen wurden einige Erweiterungen an dem Standardmodell des GAN von [5] hinzugefügt.

2.9.2 Deep-Convolution-GAN

Das *Deep Convolution GAN* (DCGAN) ist ein Versuch, *Convolutional Neural Networks* (CNNs) mit GANs zu verknüpfen. Nach vielen Fehlschlägen in der Entwicklung von GANs mit CNNs ist die Version von [13] stabil und auf viele unterschiedliche Datensätze anwendbar. Dafür wurden viele verschiedene Kombinationen von Schichten untersucht und es wurde dabei eine Architektur ausgearbeitet, die in ein stabiles Training über verschiedenste Datensätze resultierte. Zusätzlich können mithilfe dieser Architektur höhere Auflösungen und tiefere Netze erreicht werden.

Zusätzlich zur eigentlichen Architektur von DCGAN werden moderne Techniken verwendet, um CNN-Architekturen zu vereinfachen. Damit der Generator über mehrere Schichten hinweg die räumliche Darstellung von Objekten lernen kann, werden Convolutional-Layer verwendet. Anstatt, dass sogenannte Max-Pooling-Layer zum Einsatz kommen, können nach [19] einfach Convolutional-Layer mit erhöhtem Stride verwendet werden, ohne dass die Genauigkeit sinkt. In Bezug zu DCGANs von [13] werden solche Schichten verwendet, um dem Generator das Erlernen vom räumlichen Upsampling zu ermöglichen. Auch der Diskriminator wird mit solchen CNN-Layer ausgestattet, um räumliches Downsampling zu erlernen.

Neben dem Auslassen von Max-Pooling-Layer folgt DCGAN auch dem Trend, Fully-Connected-Layer vor jedem Convolutional-Feature zu vermeiden. Dabei wurde festgestellt, dass die Verknüpfung von Fully-Connected-Layer und der Eingabe des Generators bzw. mit der Ausgabe des Diskriminators am besten funktionieren. Die erste Schicht des Generators ist also ein Fully-Connected-Layer (1-dimensional), jedoch wird die Ausgabe der Schicht in einen 4-dimensionalen Tensor umgewandelt. Im Falle des Diskriminators wird die Ausgabe des letzen Convolutional-Layers (4-dimensional) abgeflacht und in eine 1-dimensionale Schicht mit einer Sigmoid-Aktivierungsfuntion gefüttert [13].

Um Mode-Collapse zu vermeiden, verwendet [13] Batch-Normalization-Layer. Dadurch wird das Training stabilisiert und Probleme wie *Internal-Covariate-Shifting* angegangen [9]. Vor allem wird dadurch aber auch verhindert, dass der Generator immer die gleichen Ausgaben erzeugt. Das Anwenden der Batch-Normalisierung in allen Schichten des Netzwerks führt jedoch zur Stichprobenoszillation und Instabilität des Modells. Aus diesem Grund wird auf Batch-Normalization in der Ausgabesschicht des Generators und in der Eingabeschicht des Diskriminators verzichtet.

Als letzte Beobachtung stellt [13] fest, dass das Hinzufügen von ReLU-Aktivierungsfunktionen in allen Schichten des Generators zu schnellerem Lernen und Abdeckung der Farbräume der Trainingsdistribution führt. In der Ausgabeschicht wird jedoch anstatt von ReLU-Aktivierung eine Tanh-Aktivierung verwendet. Innerhalb des Diskriminators werden schließlich Leaky-ReLU-Aktivierungen angewandt.

2.9.3 Wasserstein-GAN

Anders als andere GAN-Varianten verwendet das Wasserstein-GAN (WGAN) die Wasserstein-Distanz anstelle der JS- oder KL-Divergenz, um die Gewichte von generativen neuronalen Netzen zu optimieren. Da sich die Berechnung aller möglichen gemeinsamen Verteilungen $\gamma \sim \Pi(P_r, P_\theta)$ etwas schwierig gestaltet, formt [1] die Definition unter Berücksichtigung der Kontorovich-Rubinstein-Dualität um, sodass

$$W(P_r, P_\theta) = \sup_{\|f\|_L \leq 1} \mathbb{E}_{x \sim P_r}[f(x)] - \mathbb{E}_{x \sim P_\theta}[f(x)]$$

gilt, wobei das Supremum über alle 1-Lipschitz-Funktionen $f : X \to \mathbb{R}$ ist. Zusätzlich wird ein kleiner Trick angewendet, um das Problem weiter zu vereinfachen, indem K-

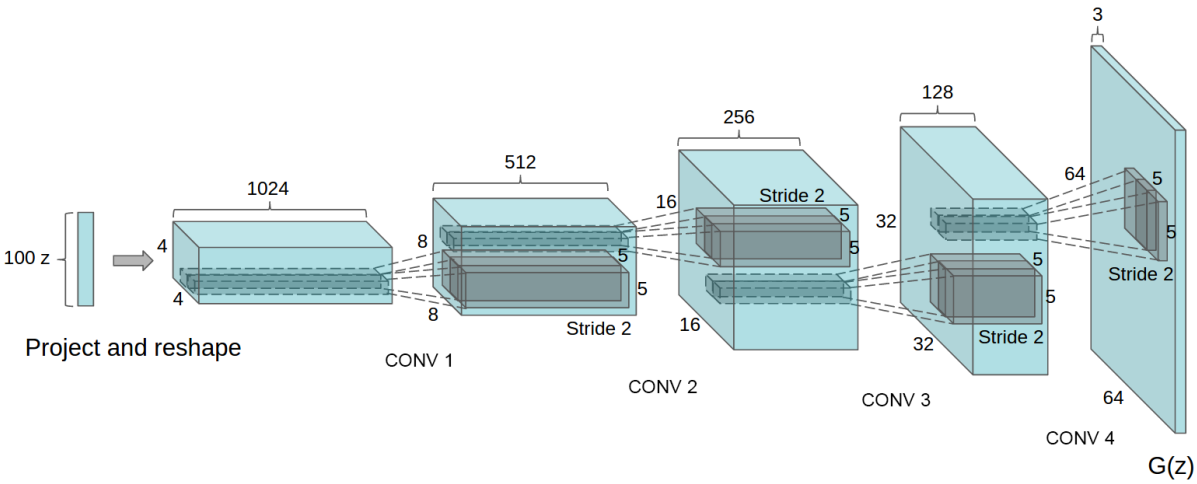

Abbildung 2.5: DCGAN-Architektur des Generators von [13]. Als Eingabe dient ein 100-dimensionaler Vektor, dessen Elemente zufällig gewählt werden. Dieser wird dann in den ersten Schichten umgeformt und durch vier Convolutional-Layer auf die Form 3×64×64 gebracht. Die Strides geben dabei den Vergrößerungsfaktor pro Convolution-Schicht an, während die Anzahl der Filter den Farbkanälen entsprechen.

Lipschitz-kontinuierliche Funktionen verwendet werden.

$$K \cdot W(P_r, P_\theta) = \sup_{\|f\|_L \leq K} \mathbb{E}_{x \sim P_r}[f(x)] - \mathbb{E}_{x \sim P_\theta}[f(x)]$$

Nehmen wir nun an, dass die Abbildung $f \in \{f_w\}_{w \in W}$ parametrisiert durch w existiert, wobei W die Menge aller möglichen Parameter darstellt, so können die Parameter w und damit die Abbildung f_w von einem neuronalen Netz erlernt werden, um so die Wasserstein-Distanz effizient abzuschätzen. Hier bildet der Wasserstein-Abstand also gleichzeitig die Loss-Funktion des Kritisierer mit

$$W(P_r, P_\theta) = \max_{w \in W} \mathbb{E}_{x \sim P_r}[f_w(x)] - \mathbb{E}_{z \sim P_r(z)}[f_w(g_\theta(z))]\,.$$

Trotzdem darf nicht vergessen werden, dass dies nur gültig ist, falls die Funktion 1-Lipschitz-kontinuierlich ist. Um dies zu erzwingen, werden die Werte der aktualisierten Gewichte des Kritisierer zwischen $[-c; c]$ gehalten. Dabei muss laut [1] c relativ klein sein.

Zusätzlich ist bei Wasserstein-GANs von einem Kritisierer (Critic) anstatt eines Diskriminators die Rede. Der Grund dafür ist, dass ein Diskriminator zwischen *fake* und *real* unterscheidet, mehr nicht. Der Kritisierer führt diese Unterteilung der Eingabeparameter

Algorithmus 1 : Wasserstein GAN nach [1]. Standardwerte für die Eingabeparameter sind $\alpha = 5 \cdot 10^{-5}, c = 0.01, m = 64$ und $n_{critic} = 5$.

Input : Lernrate α, Clipping-Parameter c, Batch-Größe m, Anzahl von Kritisierer-Iterationen n_{critic}.

Result : Trainieren der Kritisierer-Parameter w und Generator-Parameter θ.

while θ ist nicht konvergiert **do**
  for $t = 0, ..., n_{critic}$ **do**
    Erzeuge Batch $\{x_i \mid 1 \leq i \leq m\} \sim \mathbb{P}_r$ aus realen Daten;
    Erzeuge Batch $\{z_i \mid 1 \leq i \leq m\} \sim \mathbb{P}_z$ aus latenten Vektoren;
    $g_w \leftarrow \nabla_w \left[\frac{1}{m} \sum_{i=1}^{m} f_w(x_i) - \frac{1}{m} \sum_{i=1}^{m} f_w(g_\theta(z_i))\right]$;
    $w \leftarrow w + \alpha \cdot \text{RMSProp}(w, g_w)$;
    $w \leftarrow \text{clip}(w, -c, c)$;
  end
  Erzeuge Batch $\{z_i \mid 1 \leq i \leq m\} \sim \mathbb{P}_z$ aus latenten Vektoren;
  $g_\theta \leftarrow -\nabla_\theta \left[\frac{1}{m} \sum_{i=1}^{m} f_w(g_\theta(z_i))\right]$;
  $\theta \leftarrow \theta - \alpha \cdot \text{RMSProp}(\theta, g_\theta)$;
end

nicht durch, sondern bewertet bzw. kritisiert diese viel mehr. Mathematisch ausgedrückt sprechen wir hierbei von einer linearen Ausgabe in $\mathbb{R}$ im Falle des Kritisierers, während der Diskriminator eine binäre Ausgabe erzeugt. Zu Beginn von Algorithmus 1 werden die Parameter w für den Kritisierer und θ für den Generator initialisiert. Anschließend werden m Datenpunkte bzw. ein Batch aus dem reellen Datensatz (Verteilung $\mathbb{P}_r$) gezogen. Dies muss nicht unbedingt zufällig sein. Auch werden m zufällige Vektoren erzeugt, die als Eingabe für den Generator dienen, welcher wiederum Fake-Daten erzeugt. Dabei bilden die Ausgaben des Generators eine eigene Verteilung $\mathbb{P}_\theta$. Ziel des Generators ist es nun, die Distanz zwischen den beiden Verteilungen $\mathbb{P}_r, \mathbb{P}_\theta$ zu minimieren, um möglichst realitätsnahe Ausgaben erzeugen zu können. Als nächstes werden die Gradienten g_w für Parameter w mithilfe von Gradient-Descent, dargestellt als ∇_w, berechnet. Hierfür wird die Wasserstein-Distanz als Loss-Funktion verwendet. Der nächste Schritt besteht daraus, die Parameter w des Kritisierer-Netzwerks mithilfe des RMSprop-Algorithmus zu aktualisieren und die aktualisierten Gewichte so gering wie möglich zu halten, um die K-Lipschitz-Kontinuität zu gewährleisten. Dies wird mithilfe der Funktion clip umgesetzt, welche die Parameterwerte in einem bestimmten Intervall $[-c; c]$ festsetzt. Die bis hier erläuterten Schritte werden n_{critic}-mal durchgeführt, sodass das Kritisierer-Netzwerk immer öfter trainiert wird, als der Generator. Dieser wird nun optimiert, indem wieder m

Abbildung 2.6: Vergleich von WGAN und Standard-GAN [1]. Links sind Ausgaben vom WGAN-Algorithmus zu sehen während rechts Ausgaben eines Standard-GANs dargestellt sind. In beiden Generator-Modellen wurden Batch-Normalization-Layer entfernt. Klar zu erkennen ist, dass WGAN immer noch interpretierbare Ergebnisse liefert während bei Standard-GANs Probleme erkennbar sind.

Vektoren zufällig erzeugt und als Eingabe für das Generator-Netzwerk verwendet werden. Der Generator erzeugt damit m zufällige Ausgaben, die wiederum als Eingaben in das Kritisierer-Netzwerk gegeben werden. Aus den Ausgaben wird dann der Mittelwert gebildet und zum Bestimmen der Gradienten von den Generator-Parametern θ verwendet.

Im direkten Vergleich zu dem originalen GAN [5] werden einige Änderungen in der Architektur vorgenommen. Während in dem originalen Anstatz fast nach jeder Schicht eine Batch-Normalization vorgenommen wird, können diese bei WGANs entfallen. Standard-GANs würden hierbei kaum interpretierbare Resultate erzeugen, WGANs hingegen produzieren trotzdem gute Ergebnisse, wie Experimente von [1] zeigen (siehe Abbildung 2.6). Das Wasserstein-GAN hat zusätzlich noch einige nützliche Eigenschaften. So wird unter anderem durch Annäherung des Wasserstein-Abstandes zwischen Generator- und Ausgangsdistribution das Problem des Mode-Collapse gelöst. Durch den Wasserstein-Abstand wird der Abstand zwischen den Verteilung wesentlich besser minimiert (im Falle des Generator-Modells) als bei der KL- oder JS-Divergenz. Die Ausgabe eines Kritisierers stellt damit eine Bewertung der Eingabe dar, anstatt diese einer Klasse zuzuweisen und besitzt deshalb mehr Aussagekraft.

2.9.4 Wasserstein-GAN mit Gradient-Penalty

Ein großes Problem von Wasserstein-GANs ist das Clippen der Gewichte in ein fest definiertes Intervall, um die 1-Lipschitzstetigkeit zu erfüllen. Das dies keine elegante Lösung ist, liegt auf der Hand. In [6] wurde speziell dieses Problem genauer untersucht und es wurde festgestellt, dass das Beschneiden der Gewichte den Kritisierer dazu verleitet, nur extrem einfache Funktionen zu erlernen, wie der Vergleich in Abbildung 2.7 zeigt.

Um das Problem des Weight-Clippings anzugehen, stellt [6] eine alternative Lösung vor,

die auf anderem Wege die 1-Lipschitzstetigkeit in WGANs sicherstellen soll. Hierbei soll Gradient-Penalty helfen und wird als

$$(\|\nabla_{\hat{x}} D(\hat{x})\| - 1)^2$$

berechnet, wobei $\hat{x} = x\epsilon + \tilde{x}(1-\epsilon)$ eine zufällige Gewichtung zwischen realen (x) und generierten Daten ($\tilde{x}$) darstellt. Das ϵ wird dabei zufällig aus $[0, 1]$ gewählt. Daraus resultierend gestaltet sich die neue Loss-Funktion des Kritisierers wie folgt.

$$L = \mathbb{E}_{\tilde{x}\sim\mathbb{P}_g}\left[D(\tilde{x})\right] - \mathbb{E}_{x\sim\mathbb{P}_r}\left[D(x)\right] + \lambda \cdot \mathbb{E}_{\hat{x}\sim\mathbb{P}_{\hat{x}}}\left[(\|\nabla_{\hat{x}} D(\hat{x})\| - 1)^2\right]$$

Der Kritisierer ist durch diese Änderung nun wesentlich besser dazu in der Lage, komplexere Verteilungen zu erlernen.

Algorithmus 2 : WGAN mit Gradient-Penalty [6].

Input : Gradient-Penalty-Koeffizient λ, Anzahl von Kritisierer-Iterationen n_{critic}, Batch-Größe m, Adam-Hyperparameter α, β_1, β_2.

Result : Trainieren der Kritisierer-Parameter w und Generator-Parameter θ.

while θ ist nicht konvergiert **do**
  for $t = 1, ..., n_{critic}$ **do**
    for $i = 1, ..., m$ **do**
      Wähle reale Probe $x \sim \mathbb{P}_r$, latenten Vektor $\vec{z} \sim \mathbb{P}_z$, zufällige Zahl $\epsilon \in [0, 1]$;
      $\tilde{x} \leftarrow G_\theta(\vec{z})$;
      $\hat{x} \leftarrow x\epsilon + \tilde{x}(1-\epsilon)$;
      $L_i = D_w(\tilde{x}) - D_w(x) + \lambda(\|\nabla_{\hat{x}} D_w(\hat{x})\| - 1)^2$;
    end
    $w \leftarrow \text{Adam}(\nabla_w \frac{1}{m} \sum_{i=1}^{m} L_i, w, \alpha, \beta_1, \beta_2)$;
  end
  Wähle einen Batch aus latenten Vektoren $\{\vec{z}_i\}_{i=1}^{m} \sim \mathbb{P}_z$;
  $\theta \leftarrow \text{Adam}(\nabla_\theta \frac{1}{m} \sum_{i=1}^{m} -D_w(G_\theta(\vec{z}_i)), \theta, \alpha, \beta_1, \beta_2)$;
end

2.9.5 Conditional-Wasserstein-GAN mit Gradient-Penalty

Generative Modelle wie das WGAN oder DCGAN erzeugen bei einem zufälligen Eingabevektor immer ein zufälliges, interpoliertes Element aus der Trainingsdistribution. Besitzt der Trainingsdatensatz verschiedene Klassen, so werden diese Modelle auch

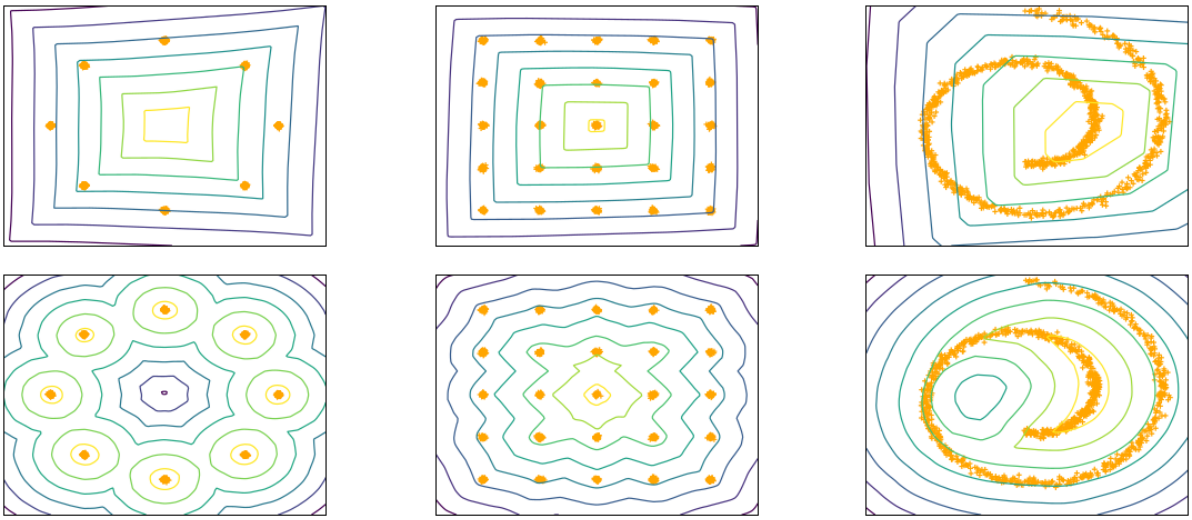

Abbildung 2.7: Vergleich zwischen Weight-Clipping (oben) und Gradient-Penalty (unten). Man erkennt deutlich, dass die Separierung der Ausgangsverteilung, dargestellt durch orangene Punkte, durch Weight-Clipping in sehr vereinfachte Funktionen resultiert. Gradient-Penalty lässt hingegen komplexere Strukturen von Verteilungen zu [6].

zwischen den Klassen interpolierte Ergebnisse erzeugen. Aber was ist, wenn wir ein zufälliges Element einer bestimmten Klasse erzeugen wollen? Nehmen wir als Beispiel einen Datensatz bestehend aus Bildern von Hunden und Katzen. Die bisher vorgestellten DCGANs und WGANs würden auf diesen Datensatz trainiert werden und würden bei einem zufälligen Eingabevektor Hunde oder Katzenbilder bzw. eine Mischung aus beidem erzeugen. Möchte man nun jedoch den Generator der GANs dazu bringen, nur zufällige Hundebilder zu erzeugen, so muss aufwändig die dafür zuständige Komponente des Eingabevektors extrahiert werden, die bestimmt, ob der Generator ein Hunde- oder ein Katzenbild erzeugen soll. Eine einfachere Möglichkeit besteht darin, dem Generator zusätzlich zum latenten Vektor die Klasse der zu erzeugenden Ausgabe zu übergeben. Diese Idee wurde erstmals mit Conditional-Generative-Adversarial-Networks (cGANs) [12] umgesetzt und stellt eine Erweiterung zu den bisher besprochenen GANs dar.

Wie bereits erwähnt, wird bei cGANs zusätzlich zum latenten Vektor x das Label bzw. die Klasse y als Eingabe verwendet. Wie dies technisch umgesetzt wird, soll das Schema in Abbildung 2.8 verdeutlichen. Die beiden Eingaben, latenter Vektor und Label, werden zuerst mithilfe einer Vollverbindungsschicht (Dense) und Umformungsschicht (Reshape) auf die gleiche Form gebracht, da diese nicht unbedingt gleich aufgebaut sein müssen. Anschließend werden die nun zueinander passenden Eingaben mithilfe einer Verkettungsschicht (Concat) aneinandergehängt. Der restliche Verlauf verhält sich wie

bei den anderen vorgestellten GANs, also das Hochskalieren der kleinen Eingabegröße mithilfe von mehreren aneinandergereihten transponierten Faltungsschichten (Conv2D Transpose). Die Verlustfunktion für das Trainieren des cGANs muss aufgrund der Verkettung beider Eingaben ebenfalls angepasst werden.

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

Der Operator | stellt dabei die Verkettungsoperation dar, sodass $x|y$ bedeutet, dass y an x gehängt wird. Aus diesem Grund müssen x und y auch dieselbe Form aufweisen. Andernfalls kann die Verkettung nicht durchgeführt werden. Das komplette Netzwerk lernt damit eine Ausgabe zu erzeugen, die auf das eingegebene Label passt. Soweit zur Idee von Conditional-Generative-Adversarial-Networks. Dies soll nun auf Wasserstein-GANs übertragen werden, sodass in späteren Implementationen Conditional-Wasserstein-GANs (cWGAN) zur Verfügung stehen, um automatisiert Datensätze zu erzeugen. Die angepasste Verlustfunktion für den Kritisierer, welche die Verkettung des Labels berücksichtigt, bildet sich wie folgt.

$$L_w(x, \tilde{x}, \hat{x}, y) = \mathbb{E}_{\tilde{x} \sim \mathbb{P}_g} \left[D(\tilde{x}|y)\right] - \mathbb{E}_{x \sim \mathbb{P}_r} \left[D(x|y)\right] + \lambda \cdot \mathbb{E}_{\hat{x} \sim \mathbb{P}_{\hat{x}}} \left[(\|\nabla_{\hat{x}} D(\hat{x}|y)\| - 1)^2\right]$$

Die Verlustfunktion für den Generator ist entsprechend

$$L_\theta = -\mathbb{E}_{\tilde{x} \sim \mathbb{P}_g} \left[D(\tilde{x}|y)\right].$$

Für ein Training, in welchem die Daten in Batches vorliegen, wird einfach der arithmetische Mittelwert berechnet. Zur Erinnerung, $x \sim \mathbb{P}_r$ entspricht einer Probe x aus einem realen Datensatz. Das x ist damit ebenfalls real. Zudem sind $\tilde{x} = G(z|y)$ und $\hat{x} = x\epsilon + \tilde{x}(1 - \epsilon)$ generierte bzw. interpolierte Proben. Mithilfe dieser Verlustfunktionen kann nun ein Conditional-Wasserstein-GAN mit Gradient-Penalty implementiert werden. Besonders wird dies im nächsten Kapitel besprochen, wenn ein Generator zum Erzeugen von ganzen Datensätzen mit verschiedenen Klassen umgesetzt werden soll.

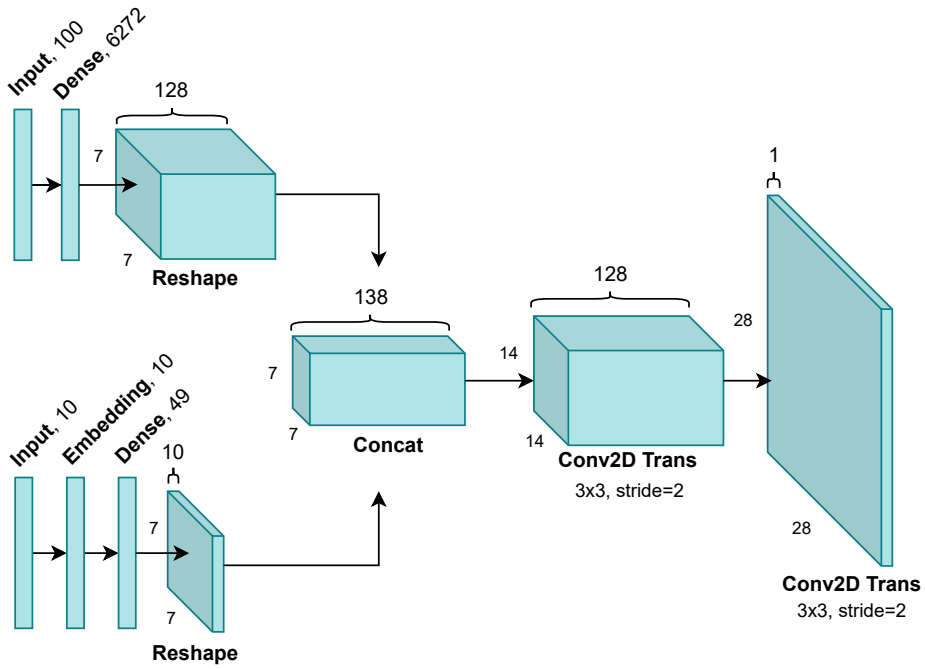

Abbildung 2.8: Schematische Darstellung eines einfachen Conditional-GANs für den MNIST-Datensatz. Die Eingabe sind ein latenter Vektor mit 100 Komponenten und ein Labelvektor mit 10 Komponenten (entsprechend der Anzahl der Klassen).

3 Erstellen eines Datensatzes

Ein wichtiger Bestandteil beim Entwickeln von künstlichen neuronalen Netzen ist der unterliegende Datensatz, der zum Trainieren der Parameter der Netze verwendet wird. Ein Machine-Learning-Modell ist nur so gut wie der verwendete Datensatz. Dieser muss deshalb eine große Varianz der Daten repräsentieren. Das bedeutet, dass ausreichend Einträge vorhanden sein müssen, um das Netzwerk auf ähnliche, aber unbekannte Probleme vorzubereiten. Ein beliebter Datensatz ist beispielsweise der MNIST-Datensatz [4], welcher unter anderem Bilder von handgeschriebenen Ziffern bereitstellt. Anhand des MNIST-Beispiels bedeutet eine große Varianz, dass eine Ziffer durch viele verschiedene Bilder repräsentiert wird, die alle eine andere Perspektive des gleichen Kontexts darstellen. Das Problem ist nämlich, dass eine handgeschriebene Ziffer immer anders aussieht. Um ein neuronales Netz also auf dieses Problem vorzubereiten, müssen verschiedenste Varianten zum Training bereitstehen.

In diesem Abschnitt soll nun ein Datensatz eingeführt werden, welcher keine Daten für das Erkennen von Ziffern, aber für die die Bewegungserkennung bereitstellt. Um diesen Datensatz zu erstellen, muss vorab geklärt werden, was eine Bewegung eigentlich ist. Betrachtet man einen Punkt im Raum, dann beschreibt eine Bewegung des Punktes die Änderung des Ortes eines Objekts über die Zeit. Bezogen auf den zu erstellenden Datensatz bedeutet dies, dass einfache Bildaufnahmen von beweglichen (dynamischen) Objekten nicht ausreichen. Um eine Bewegung aufzuzeichnen, müssen zeitlich aufeinander folgende Bilder, also Videos aufgenommen werden. Da der Datensatz später dazu verwendet werden soll, ein neuronales Netz zu trainieren, welches Bewegungen erkennt, müssen die zu erlernenden Bewegungen auch in diesem Datensatz als Videos vertreten sein.

Bevor ein Modell für die Bewegungserkennung und -analyse trainiert wird, soll zusätzlich untersucht werden, ob ein relativ kleiner Datensatz ausreichend ist, um trotzdem gute Resultate beim Trainieren von neuronalen Netzen zu erzeugen. Da diese Netze jedoch auf einen vielseitigen Datensatz zurückgreifen müssen, um entsprechend gute Erken-

nungsraten zu erzeugen, ist die Idee, diesen kleinen Datensatz künstlich zu erweitern. Interessant hierbei ist, ob ein künstlich gedehnter Datensatz genau so gut geeignet ist, wie ein aufwändig erstellter Datensatz bestehend aus realen Daten. Falls sich das Experiment als erfolgreich herausstellt, liegt der Vorteil auf der Hand. Nämlich dass künstlich erzeugte Datensätze wesentlich schneller auszuheben sind als Datensätze mit ausschließlich realen Daten. Dies würde den Arbeitsaufwand für das Erstellen von neuen komplexen Datensätzen um ein vielfaches verringern und gleichzeitig den Fokus auf die eigentliche Forschungsarbeit verbessern.

Nun stellt sich natürlich die Frage, wie ein kleiner Datensatz künstlich vergrößert werden kann. Künstlich bedeutet hier, dass das Vergrößern des Datensatzes mithilfe von Algorithmen geschieht und vom Computer anstatt vom Menschen durchgeführt wird. Die folgenden Beispiele richten sich nach [16]. Eine Möglichkeit ist es, die Daten zu augmentieren. Dies wird bereits häufig während des Trainings von neuronalen Netzen durchgeführt, indem simple Transformationen wie Rotation, Translation und Verzerrung an den Daten durchgeführt werden. Aber auch das Verändern des Gamma-Wertes eines Bildes reicht in einigen Fällen aus, um eine Variation im Datensatz künstlich zu erzeugen. Eine weitere Möglichkeit ist es, anderen neuronalen Netzen diese Arbeit übernehmen zu lassen. Hierbei werden generative Modelle wie GANs verwendet, um neue, bisher unbekannte Samples eines Datensatzes zu erzeugen. Auch diese Idee ist nicht neu, jedoch beziehen sich die meisten Beispiele auf das Generieren von einfachen Bildern. Da es sich bei dem zu erstellenden Datensatz um Videos handelt, müssen die Techniken auf dieses Problem angewandt werden. Neben den Inhalt von einzelnen Bildern muss das GAN als zusätzliche Dimension also die Semantik zwischen einzelnen Frames bzw. Bildern erlernen und neu generieren können, da es keinen Sinn ergibt, dass der Kontext pro Videoframe zufällig wechselt. Stattdessen wird gefordert, dass jeder erzeugte Frame von dem vorherigen abhängt, sodass die Folge der Frames ein sinnvolles Video ergibt.

3.1 GAN für Videos

In diesem Abschnitt wird die Architektur von ViGAN vorgestellt und die Idee dahinter erläutert. ViGAN steht für Video-Generative-Adversarial-Network und wurde im Rahmen dieser Arbeit zum Erzeugen von neuen Videos aus einem bestehendem Datensatz entwickelt. Gerade das Erstellen von Videos und das Labeln der Bewegungen nimmt enorm viel Zeit in Anspruch. Mithilfe von ViGAN soll der Aufwand reduziert

werden. Hierbei wird untersucht, ob ein künstlich erweiterter Datensatz für das Trainieren von neuronalen Netzen geeignet ist. In Kapitel 2.9 werden verschiedene GANs und deren Probleme bzw. Vorteile vorgestellt. Aufgrund der Vorteile von Wasserstein-GANs, nämlich die Interpretationsmöglichkeit des Fehlers während des Trainings und die Mode-Collapse-Resistenz, wird dieses als Grundarchitektur gewählt. Da häufig GANs nur in Verbindung mit einzelnen Bildern verwendet werden, muss das WGAN entsprechend angepasst werden, um Videos anstatt von Bildern zu erzeugen. Wie im vorherigen Abschnitt besprochen, ist es wichtig, dass die Frames des Videos zueinander passen und nicht aus dem Kontext gerissen werden. Wie in Abbildung 3.1 dargestellt unterscheidet sich ViGAN nicht allzu sehr vom WGAN. Der größte Unterschied besteht darin, dass 3D-Convolutional-Transpose- anstelle der 2D-Convolutional-Transpose-Layer verwendet werden. Dies ermöglicht nicht nur ein Hochskalieren des Bildes, sondern auch ein Erhöhen der Frameanzahl. Die Idee dabei ist, dass das Netzwerk dadurch lernt, zusammenhängende Frames zu generieren. Als Eingabe wird ein zufälliger Vektor $\vec{z}$ gewählt. Dann muss das Netzwerk die eindimensionale Eingabe in eine vierdimensionale Struktur umwandeln. Dies geschieht beim ViGAN mithilfe eines Fully-Connected-Layers mit $5{\cdot}6{\cdot}8{\cdot}256$ Neuronen und einem Reshape-Layer, welcher die Ausgabe des Fully-Connected-Layers in einen 4D-Tensor mit den Dimensionen $5 \times 6 \times 8 \times 256$ umstrukturiert. Die erste Dimension beschreibt dabei immer die Anzahl der Frames, die zweite die Höhe, die dritte die Breite und die vierte die Anzahl der Farbkanäle. Anschließend wird der Tensor mithilfe eines Convolutional-Transpose-Layers vergrößert und die Frameanzahl erhöht. Der Stride beträgt dabei 2, sodass eine Verdoppelung der Dimensionen stattfindet. Insgesamt werden vier solcher Schichten hintereinander gereiht. Dabei entsteht ein Ausgabevideo mit 80 Frames, wobei jeder Frame jeweils $96 \times 128 \times 3$ Pixelwerte besitzt.

3.2 Training vom ViGAN

Um ViGAN zu testen, werden einige Experimente durchgeführt, die untersuchen sollen, ob die Ausgaben des GANs brauchbar sind. Unter anderem wird durch Experimente herausgefunden, welcher Aufbau von Generator und Diskriminator (in diesem Fall ist der Diskriminator ein Kritisierer) in einem stabilen Training resultieren. Das Training wird auf eine GeForce RTX 3080 ausgelagert und dauert im Schnitt 12 Stunden für 9000 Epochen bei einem Datensatz von 3 Videoclips mit jeweils 840 Frames. Das größte Problem während des Trainings besteht darin, die Trainingsdaten in den Speicher der Grafikkarte zu laden, da das Modell und die Daten entsprechend groß sind. Daher kann

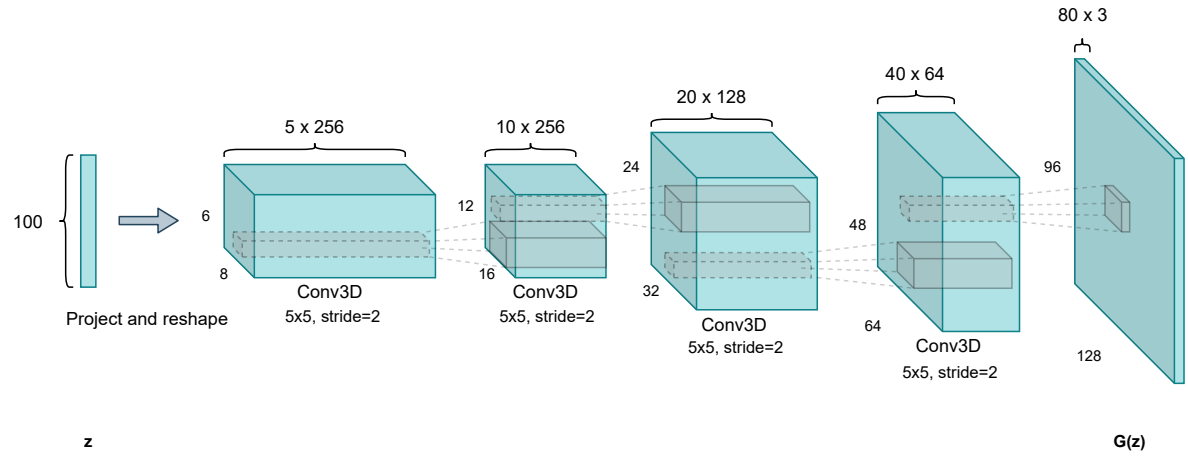

Abbildung 3.1: Architektur von ViGAN. Als Eingabe dient ein zufälliger Vektor $\vec{z} \in \mathbb{R}^{100}$. Dieser wird anschließend in fünf $6 \times 8 \times 256$ Features umgewandelt. Um die relativ kleinen Frames in eine größere Auflösung zu skalieren, werden vier 3D-Convolutional-Transpose-Layer verwendet. Jedes dieser Schichten verwendet eine Kernelgröße von $5 \times 5 \times 5$ und einen Stride von 2. Insgesamt ergibt sich also ein Stride von $2^4 = 16$, sodass die Eingabegröße von $5 \times 6 \times 8 \times 256$ auf 80x96x128x3 hochskaliert wird.

nur mit einer Batch-Größe von 4 gearbeitet werden. Insgesamt wurden 8 Experimente durchgeführt, wobei sich die Architekturen aus Tabelle 3.1 am stabilsten herausstellten. Trotz des stabilen Trainings weisen die Ausgaben des Generator-Netzwerkes einige Makel auf. Es entstehen zwar Videos, in denen eine Bewegung stattfindet, jedoch sind auch Transitionen in der Inneneinrichtung vorhanden. So verschwindet während des Videoclips beispielsweise eine Tür und es erscheint eine Garnitur. Auf jeden Fall entstehen aber Räume, die so in der Art nicht im Datensatz vorhanden sind. Leider wird auch nicht immer eine menschliche Bewegung erzeugt, sodass sich zwar der Raum über die Zeit ändert, die Person aber regungslos in diesem steht. Weitere Experimente haben gezeigt, dass auch zwischen Personen interpoliert wird. Die sich bewegende Person verändert sich also im Laufe der Zeit. Die Ergebnisse von ViGAN sind demnach zwar Videos, dessen Frames voneinander abhängen, jedoch wurde es nicht geschafft, dynamische Bewegungen mit konstanter Umgebung zu generieren.

3.3 GAN für Schlüsselpunktanimationen

Da das ViGAN einige Probleme aufgewiesen hat, wurde eine alternative Methode entwickelt, um Bewegungen effizienter und genauer zu generieren. Weil sich diese Arbeit mit

Input-Shape	Conv1	Conv2	Conv3	Conv4	Output-Shape	Parameter
$5 \times 12 \times 16 \times 256$	c=128 s=2,2,2	c=64 s=2,2,2	c=64 s=2,2,2	c=3 s=1,2,2	$40 \times 192 \times 256 \times 3$	$3{,}07 \cdot 10^7$
$1 \times 3 \times 4 \times 96$	c=96 s=5,5,5	c=48 s=2,3,3	c=16 s=2,4,4	c=3 s=2,2,2	$40 \times 360 \times 480 \times 3$	$0{,}20 \cdot 10^7$
$1 \times 6 \times 8 \times 96$	c=96 s=5,5,5	c=48 s=3,3,3	-	c=3 s=4,4,4	$40 \times 360 \times 480 \times 3$	$0{,}22 \cdot 10^7$
$10 \times 20 \times 45 \times 256$	c=128 s=1,1,1	c=64 s=2,2,2	-	c=3 s=2,2,2	$40 \times 80 \times 180 \times 3$	$2{,}40 \cdot 10^8$
$5 \times 3 \times 4 \times 256$	c=256 s=1,1,1	c=128 s=2,2,2	c=128 s=2,2,2	c=3 s=3,3,3	$60 \times 36 \times 48 \times 3$	$1{,}60 \cdot 10^7$
$5 \times 3 \times 4 \times 512$	c=256 s=2,2,2	c=128 s=2,2,2	-	c=3 s=3,3,3	$60 \times 36 \times 48 \times 3$	$2{,}37 \cdot 10^7$
$5 \times 6 \times 8 \times 256$	c=128 s=2,2,2	c=64 s=2,2,2	c=32 s=2,2,2	c=3 s=2,2,2	$80 \times 96 \times 128 \times 3$	$1{,}17 \cdot 10^7$
$15 \times 12 \times 16 \times 256$	c=128 s=2,2,2	c=64 s=2,2,2	-	c=3 s=2,2,2	$120 \times 96 \times 128 \times 3$	$8{,}03 \cdot 10^7$

Tabelle 3.1: Experimente mit ViGAN. Es wurde untersucht, welche Konfigurationen zu einem stabilen Training führen. Außerdem wurde versucht, die Anzahl der erzeugten Frames so hoch wie möglich zu halten. Die Convolutional-Layer besitzen alle $5 \times 5 \times 5$ Kernels. Die Parameter c und s beschreiben entsprechend die Channels und Strides der Convolutional-Layer.

der Bewegungserkennung auseinandersetzt und ViGAN zu viel bewegungsunabhängige Transitionen in den Videos erzeugt, soll eine neue Methode vorgestellt werden, die direkt Bewegungsinformationen künstlich erzeugen kann. Auch in dieser Methode wird ein GAN verwendet, nur diesmal wird die Information einer Bewegung wesentlich genauer betrachtet. Das Ziel ist ein schnelleres Training des Netzwerks sowie ein geringerer Speicherverbrauch. Ein großer Nachteil beim ViGAN ist, dass viele unnötige Informationen mit generiert werden. Anstatt also Videos herkömmlicher Art zu erzeugen, sollen nun Animationen bestehend aus Schlüsselpunkten des menschlichen Körpers generiert werden. Die Folge dieser Schlüsselpunkte soll schließlich eine Bewegung darstellen. Extrahiert man die Schlüsselpunkte aus entsprechenden Videos, die eine Bewegungsart beinhalten, so wird der eigentliche Datensatz um ein vielfaches reduziert, jedoch bleibt die zu betrachtende Information gleich. Genau wie beim ViGAN wird hier ebenfalls ein WGAN als Grundarchitektur verwendet. Im Grunde verhält sich das Netzwerk ähnlich, außer dass es einzelne Bilder anstatt Videos generiert. Genauer gesagt, wird ein Conditional-Wasserstein-GAN verwendet, sodass das jeweilige Label mit als Eingabe fungieren kann, um den Generator zu zwingen, eine Ausgabe einer bestimmten Klasse zu erzeugen. Dadurch kann gesteuert werden, welche Art von Ausgabe dieser Generator liefert, was unabdingbar für das Erzeugen eines kompletten Datensatzes ist. Andernfalls erhält man zwar künstlich erzeugte Bewegungen, jedoch ist die Klasse bzw. das Label unbekannt. Demnach wär die einzige Alternative das Trainieren von n WGANs, wobei n die Anzahl

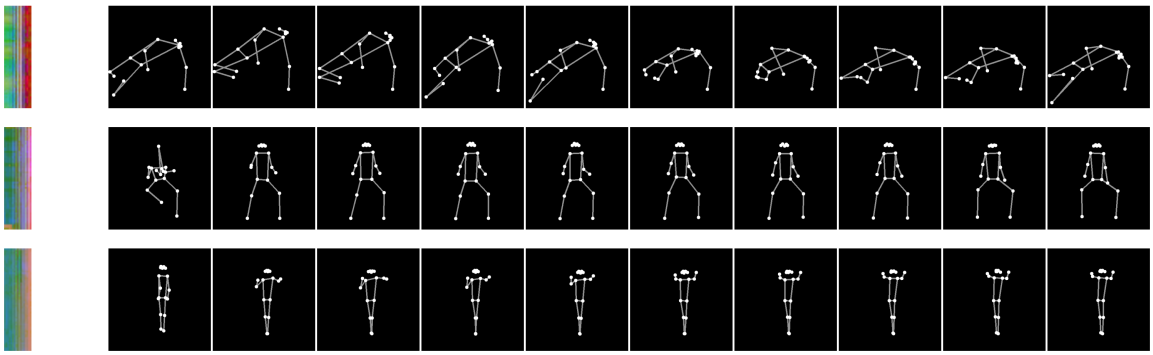

Abbildung 3.2: Die Frames von Schlüsselpunktanimationen können als zweidimensionale Bilder repräsentiert werden. Die erste Reihe stellt eine Liegestütz-, die zweite eine Bankpress- und die dritte eine Hantelbewegung dar. Die erste Spalte zeigt jeweils das kodierte Bild, während die restlichen Spalten die ersten 10 Frames der Bewegung zeigen.

der im System vorkommenden Klassen beschreibt, also ein WGAN pro Klasse. Da dies eine sehr unelegante Lösung wär, wird versucht ein cWGAN entsprechend Abschnitt 2.9.5 zu implementieren, welches Proben aus n verschiedenen Klassen erzeugen kann. Mithilfe eines Labels als Eingabe soll gesteuert werden können, zu welcher Klasse das Erzeugnis gehören soll

Eine Bewegung bestehend aus N Frames und k Schlüsselpunkten, welche jeweils eine Position im zweidimensionalen Raum und eine Auftrittswahrscheinlichkeit besitzen, können mithilfe eines Bildes repräsentiert werden. Ein solches Bild besitzt damit eine Größe von $N \times k \times 3$. Da später Modelle verwendet werden sollen, welche auf dem *COCO Keypoints Dataset* [11] trainiert wurden und somit 17 Schlüsselpunkte von Menschen in Bildern erkennen, wird $k = 17$ gewählt. Zudem wird die Anzahl an Frames auf $N = 60$ gesetzt, es wird also angenommen, dass entsprechende Bewegungen innerhalb von 60 Frames eindeutig erkennbar sind. Die Ausgaben des GANs sind dementsprechend im Format $60 \times 17 \times 3$. In Abbildung 3.2 werden beispielhaft drei Bewegungsarten zu Bilder kodiert und deren ersten 10 Frames dargestellt, die durch die ersten 10 Reihen des jeweiligen Bildes repräsentiert werden. Auf Basis dieser Bilder wird nun ein GAN entwickelt, welches neue Bewegungen des gleichen Typs erzeugen soll. Mit anderen Worten, das GAN soll lernen, ähnliche Bilder zu erzeugen, die wiederum in Schlüsselpunktanimationen dekodiert werden können. Wenn dieses Experiment erfolgreich ist, dann kann der Generator als Datensatzerzeuger für die Bewegungserkennung verwendet werden.

Abbildung 3.3: Beispielausgabe vom KpGAN. Links ist die direkte Ausgabe des Netzwerks. Von links nach rechts werden chronologisch die ersten 10 dekodierten Frames dargestellt.

3.4 Training vom KpGAN

Für das Training wurde der UCF-101-Datensatz [18] verwendet. Dieser enthält sportliche Aktivitäten in Form von Videos, sodass diese vorerst in Schlüsselpunktanimationen umgewandelt werden müssen. Anstatt jeden Frames jedes Videos per Hand zu bearbeiten und Schlüsselpunkte zu annotieren, wird dies mithilfe von MoveNet [20] automatisiert erledigt. Die Schlüsselpunkte der jeweiligen Frames werden in Bilder wie in Abbildung 3.2 kodiert und als Eingabe in das KpGAN gegeben. Die Experimente verlaufen ähnlich wie beim ViGAN, es werden jedoch andere Parameter gewählt, die pro Experiment verändert werden. Im ersten Schritt wurde untersucht, welche Netzkonfiguration stabil beim Training ist. Der finale Aufbau ist in Abbildung 3.4 zu sehen und wurde wie beim ViGAN durch unterschiedliche Versuche herausgefunden. Anschließend wurde untersucht, wie sich das Verhalten während des Trainings ändert, wenn wahlweise Batch-Normalisierung- und Bias-Schichten hinzugefügt werden. Außerdem wurde geschaut, welche Aktivierungsfunktion in der letzten Schicht zu den besten Ergebnissen führt. Dafür wurden Aktivierungen durch Sigmoid- und Tangens-Hyperbolicus-Funktionen realisiert. Konfigurationen mit Bias-Layer, aber ohne Batch-Normalisierung führten zu den besten Ergebnissen. Abbildung 3.3 zeigt eine zufällige Liegestützbewegung, die vom KpGAN erzeugt wurde. Für das Training stand eine GeForce RTX 3080 zur Verfügung und 3000 Epochen dauerten < 1 Minuten. Grundsätzlich sind die Ergebnisse zufriedenstellend, sodass dieses GAN verwendet werden kann, um verschiedene Bewegungen zu erzeugen.

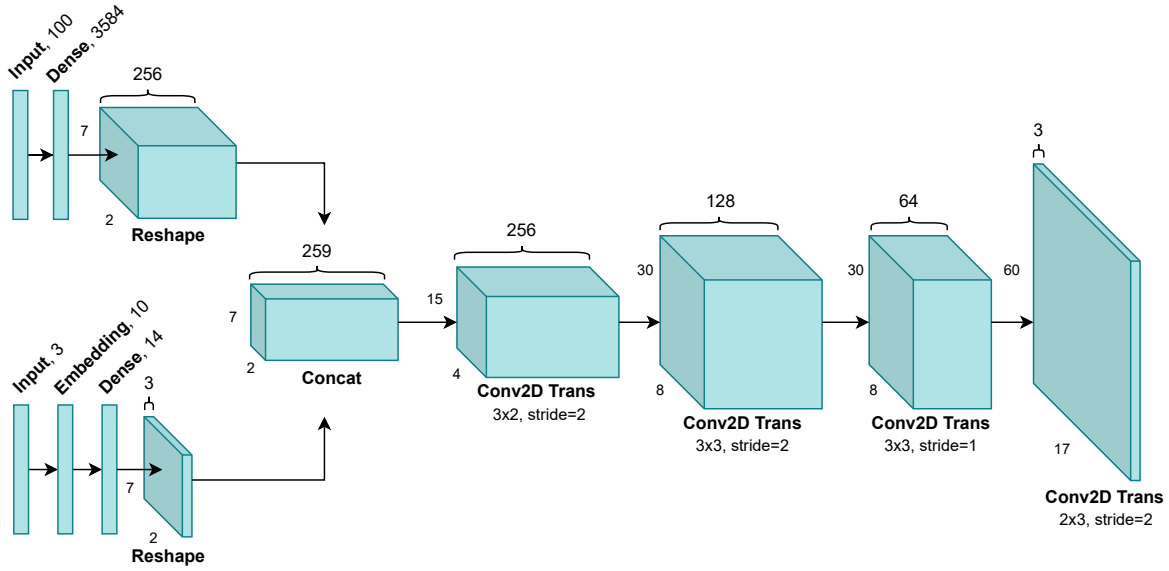

Abbildung 3.4: Architektur von KpGAN zum Generieren von Schlüsselpunktanimationen. Die Eingabe ist ein Vektor, welcher zunächst in das Format $7 \times 2 \times 256$ gebracht wird und das Label der zu erzeugenden Bewegungsart. Anschließend wird mithilfe von insgesamt vier Convolutional-Transpose-Layer eine Ausgabe im Format $60 \times 17 \times 3$ erzeugt. Es werden entsprechend 256, 128, 64 und 3 Filter für die Layer verwendet.

4 Bewegungserkennung

In diesem Kapitel sollen die Ergebnisse aus Kapitel 3 verwendet werden, um eine Bewegungserkennung zu entwickeln. Genauer gesagt soll hier ein Machine-Learning-Modell entworfen werden, um Bewegungen zu erkennen und klassifizieren. Zudem soll ein Ausblick auf Techniken gegeben werden, um weitere bewegungsabhängige Eigenschaften mithilfe von KNNs zu extrahieren. Zum Beispiel ist beim Ausüben von sportlichen Aktivitäten neben der Bewegungsart auch interessant, ob die Bewegung richtig ausgeführt wurde. Aber auch das Vorhersagen von zukünftigen Bewegungen anhand kürzlich getätigten Posen kann für einige Anwendungen hilfreich sein. So auch zum Beispiel beim vorzeitigen Erkennen von aggressiven Verhaltensmustern, sodass eingegriffen werden kann, bevor die eigentliche Tätigkeit ausgeführt werden kann.

Eine Bewegung kann durch die Änderung des Ortes über die Zeit beschrieben werden. Das bedeutet, dass eine einzige Fotoaufnahme einer sich bewegenden Person nicht ausreicht, um eine Bewegung aufzuzeichnen. Entsprechend müssen Bilder über Zeit aufgenommen werden. Die nächste Frage, die sich ergibt ist, wie viele Bildaufnahmen nötig sind, um eine Bewegung erkennen zu können. Dies hängt natürlich von der Art der Bewegung und der Bewegungsgeschwindigkeit ab. Möchte man beispielsweise einen Jumping-Jack aufnehmen und die Person bewegt sich viel zu langsam, dann sind wesentlich mehr Aufnahmen nötig, als wenn diese sich in einem normalen Tempo bewegen würde. In der Informationstheorie wird dies mithilfe der Abtastrate beschrieben, die angibt, wie oft pro Sekunde abgetastet werden soll. Damit in Verbindung kann man die minimale Rate durch das Niquist-Shannon-Abtasttheorem bestimmen, welches aussagt, dass ein Signal exakt rekonstruierbar ist, wenn ein Signal mit der doppelten Abtastrate $f_{abtast} = 2 \cdot f_{max}$ abgetastet wird.

Ein durchschnittlicher Fahrradfahrer schafft eine Trittfrequenz von maximal 60 Umdrehungen pro Minute während Leistungssportler bis zu 110 Umdrehungen pro Minute schaffen [17]. Das würde bedeuten, dass die Abtastrate mindestens 2 Hz betragen muss, um das Signal rekonstruierbar aufzuzeichnen. Betrachtet man nun eine Beinpressbewe-

gung, die ebenfalls mit 60 Tritten pro Minute ausgeführt wird, ergibt sich ebenfalls eine Abtastfrequenz von 2 Hz. Hier wird folgendes Problem ersichtlich. Tastet man die Fußpositionen der Sportler mit 2 Hz ab, so kann man nicht zwischen Kreisbewegung und Linearbewegung unterscheiden. Würde man hingegen mit 3 Hz abtasten, so wären die Bewegungen eindeutig voneinander unterscheidbar. Dies hat unter anderem damit zu tun, dass eine lineare Funktion durch zwei Punkte und ein Kreis stets durch drei sich nicht auf einer Geraden befindenden Punkte beschreibbar ist. Damit also ein neuronales Netz eine akkurate Bewegungserkennung durchführen kann, ist es wichtig, dass dieses mit möglichst vielen Frames pro Sekunde (FPS) ausgeführt wird, um verschiedenste Bewegungen erkennen zu können. Eine Überabtastung wird hierbei als weniger kritisch angesehen, als eine Unterabtastung.

Für die Bewegungserkennung wird in dieser Arbeit die Erkennung von menschlichen Posen eingesetzt, die Schlüsselpunkte des Körpers aus Bildern extrahiert. Aus diesen Informationen können bereits einige Eigenschaften einer Bewegung abgeleitet oder berechnet werden. So ist zum Beispiel die Berechnung des Winkels zwischen Oberschenkel und Wade relativ simpel, wenn die entsprechenden Schlüsselpunkte bekannt sind. Dem Benutzer kann dadurch berechnet werden, ob eine Übung, die abhängig von diesem Winkel ist, richtig ausgeführt wird oder nicht. Grundsätzlich wird folgende Idee zum Erkennen bzw. Analysieren von Bewegungen verfolgt. Die von einer Kamera aufgenommenen Bilder werden als Eingabe in einen Pose-Detektor gegeben, welcher zunächst die in dem Eingabebild enthaltenen Schlüsselpunkte ausgibt. Diese werden dann anschließend in einen angehängten Prediction-Head gegeben, welcher dann aus den Schlüsselpunkten die Bewegungsart erkennt.

4.1 Erkennung von menschlichen Posen

Das Erkennen von menschlichen Posen hat in den letzten Jahren viel Aufmerksamkeit bekommen und wird in 2D- bzw. 3D-Human-Pose-Estimation (HPE) unterschieden. Diese Arbeit beschäftigt sich mit der 2D-HPE, um Schlüsselpunkte des menschlichen Körpers in Bildern zu detektieren. Auch wird sich auf die Erkennung von Einzelpersonen in Bildern konzentriert, anstatt viele verschiedene Personen gleichzeitig zu bestimmen. Hierbei spricht man von Single-Person-Pipelines, die wiederum in zwei unterschiedliche Methoden unterteilt werden: Regressions- und Körperteildetektionsmethoden [21].

Mit Regressionsmethoden werden zu Eingabebildern direkt die Schlüsselpunkte ausge-

geben. Dabei soll ein KNN lernen, Bilder auf Schlüsselpunktkoordinaten abzubilden. Die Ausgabe besitzt dementsprechend $k \cdot n$ Komponenten, wobei k die Anzahl der Schlüsselpunkte und n die Anzahl der Eigenschaften pro Schlüsselpunkt angibt. Das Problem bei dieser Methode ist, dass Körperteile wie Hände, Augen und Füße in einem Bild sehr klein ausfallen können. Dort ist es dann sehr schwer, diese von der Umgebung unterscheiden zu können und damit eine präzise Aussage darüber zu treffen, wo sich deren Schlüsselpunkte befinden. Auch ist es möglich, dass einige Körperteile nicht auf dem Bild zu sehen sind. Das Treffen einer Aussage, ob das entsprechende Körperteil überhaupt im Bild vorhanden ist, gestaltet sich bei dieser Methode eher schwierig. Hierbei kann die folgende Methode helfen.

Bei Körperteildetektionsmethoden verläuft die Erkennung von Schlüsselpunkten ein wenig anders. Anstatt direkt auf Koordinaten der Schlüsselpunkte abzubilden, versucht man hier eine Reihe von Wärmekarten zu erzeugen. Genauer ausgedrückt wird für k Schlüsselpunkte eine Menge aus Wärmekarten $H = \{H_1, ..., H_k\}$ erzeugt. Eine Wärmekarte H_i enthält dabei für jedes Koordinatenpaar (x, y) einen Wahrscheinlichkeitswert $p_{y,x} \in H_i$, welcher angibt, zu welcher Wahrscheinlichkeit dort ein Schlüsselpunkt vorliegt. Für das Training solcher Modelle wird meistens die Abweichung zwischen generierten und tatsächlichen Wärmekarten mit Hilfe des Mean-Squared-Errors (MSE) minimiert. Grundsätzlich besitzen Körperteildetektionsmethoden den Vorteil, dass auch relativ kleine Körperteile gut erkannt werden. Der Unterschied zwischen den Methoden ist in Abbildung 4.1 zu sehen. Es werden in aktuellen Forschungen bezüglich der Posenerkennung häufig Modelle, die Wärmekarten erzeugen, verwendet [21].

Ein Forschungsteam von Google hat ein neues Machine-Learning-Modell namens *MoveNet* [20] präsentiert, das unter anderem auf mobilen Geräten ausgeführt werden kann und dabei eine Echtzeiterkennung von menschlichen Posen ermöglicht. Die Architektur besteht aus einem MobileNetV2-Backbone mit angehängtem Feature-Pyramid-Network (FPN) und insgesamt vier Prediction-Heads, die CenterNets [22] sind. Trainiert wurde das Netzwerk auf den COCO-Datensatz und erreicht laut den Autoren 30 oder mehr FPS auf modernen Computern, Laptops und Smartphones. Auch dieses Modell verwendet Wärmekarten, um mögliche Positionen von menschlichen Schlüsselpunkten zu bestimmen und wird von einem Prediction-Head übernommen. Ein weiterer ist dafür zuständig, den Abstand (Offset) jedes Pixels als Vektor von der geschätzten Schlüsselpunktposition zu bestimmen. Ein anderer Kopf übernimmt die Gruppierung der Schlüsselpunkte und ordnet diesen Personeninstanzen zu. Der vierte Kopf schätzt das Zentrum der Instanzen. Die vollständige Architektur von MoveNet ist in Abbildung 4.2 zu sehen. MoveNet ver-

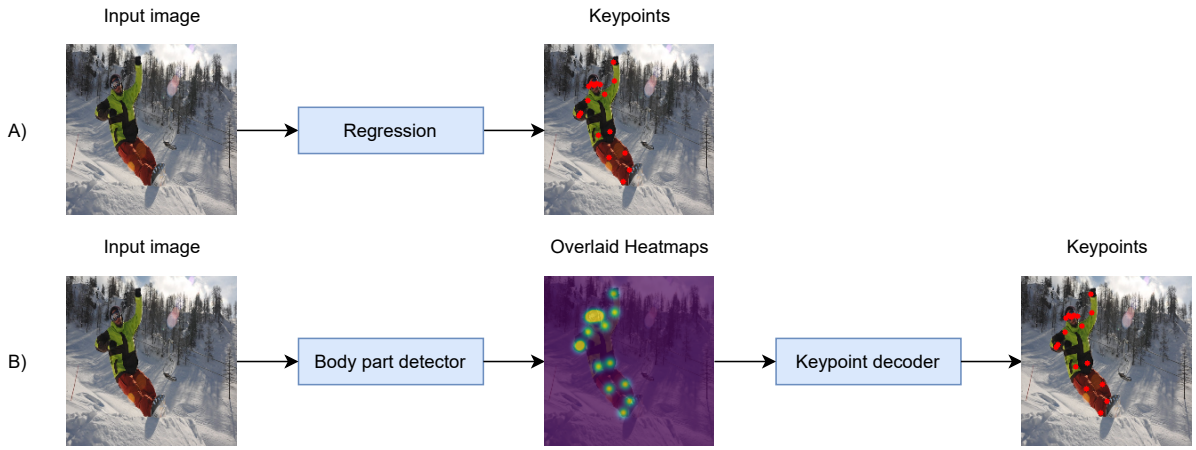

Abbildung 4.1: Regressions- und Körperteildetektions-Verfahren zum Bestimmen von Schlüsselpunkten eines Menschen. A) Ein Regressor versucht direkt die Punkte des menschlichen Körpers als Koordinaten auszugeben. B) Ein Körperteildetektor erzeugt eine Wärmekarte für jeden Schlüsselpunkt. Die Werte in den Karten geben die Wahrscheinlichkeit für den Aufenthalt eines Punktes an. Die Schlüsselpunkte müssen nach der Detektion aus den Karten dekodiert werden.

wendet also neben dem Erzeugen von Körperteildetektionsmethoden zum Erzeugen von Wärmekarten auch Regressionsmethoden, um die Position von Schlüsselpunkten noch genauer angeben zu können.

4.2 Erkennung von Bewegungsarten

Bei der Erkennung von Bewegungsarten handelt es sich um ein Klassifizierungsproblem, bei der es darum handelt, eine Bewegungsaufzeichnung einer Klasse zuzuordnen. Zu einer beliebigen Bewegungssequenz soll also eine Aussage getroffen werden, ob es zum Beispiel Hantelübungen sind. Im Laufe dieses Abschnitts soll dementsprechend ein Machine-Learning-Modell entworfen werden, welches auf diese Aufgabe trainiert wird. Hierzu muss vorerst definiert werden, welche Daten verwendet werden und wie diese für das Training vorbereitet werden können. Damit eine Ortsänderung eines zu betrachtenden Objektes aufgezeichnet werden kann, müssen mehrere Bilder bzw. Videos aufgezeichnet werden. Eine Bewegung anhand eines einzelnen Bildes ist nicht möglich, da die Information über die Ortsänderung dann nicht vorhanden ist. Es gilt also einen Datensatz zu finden, welcher verschiedene Bewegungen im Videoformat bereitstellt. Alternativ muss ein entsprechender Datensatz ausgehoben werden, was einen relativ hohen Zeitaufwand bedeuten würde. Es wird aufgrund seiner Einfachheit der UCF101-Datensatz [18] ver-

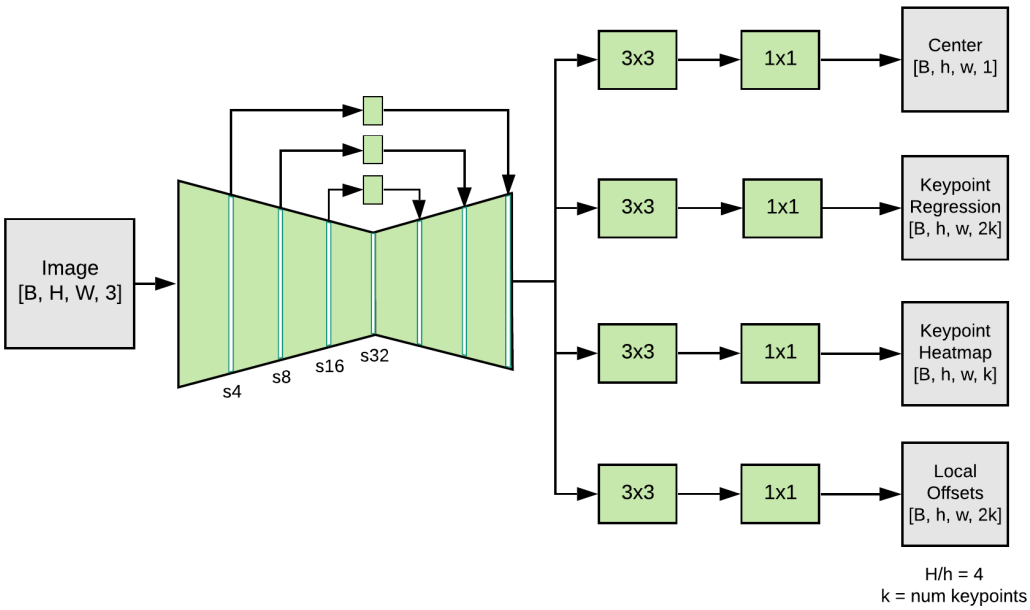

Abbildung 4.2: Architektur von MoveNet [20]. Der Feature-Extractor besteht aus einem vortrainierten MobileNetV2-Backbone mit angehängtem FPN. Insgesamt werden vier Prediction-Heads zum Bestimmen von Schlüsselpunkten verwendet.

wendet. Dieser besteht aus 101 Bewegungsarten in Form von Videos, die öffentlich auf der Plattform YouTube zur Verfügung stehen. Ein Videoclip ist dabei eindeutig einer Bewegungsklasse zugeordnet. Um den Datensatz künstlich zu erweitern bzw. um eine Augmentation durchzuführen, wird das KpGAN aus Kapitel 2.9 verwendet. Dieses erzeugt während des Trainings Schlüsselpunktdaten auf Nachfrage.

Es wird nun ein Modell entworfen, welches n Frames eines Videos einliest und die geschätzte Bewegungsklasse für diese Frames ausgibt. Dabei soll das Netzwerk lernen, menschliche Posen aus den Frames zu extrahieren und anschließend zu klassifizieren. Da das Neutrainieren der Posenerkennung zu lange dauern würde, wird ein vortrainiertes MoveNet als Backbone verwendet. Für die Klassifizierung werden verschiedene Köpfe implementiert, die anschließend miteinander verglichen werden. Ziel ist es herauszufinden, welche Architekturen für den Gebrauch auf mobilen Plattformen geeignet sind. Als Testplattform wird, wie einleitend beschrieben, Android verwendet.

Die zu betrachtenen Metriken sind Genauigkeit und Ausführungsgeschwindigkeit. Als erstes werden einfache Fully-Connected-Layer verwendet und evaluiert. Es wurden dabei Schritt für Schritt Änderungen an der Anzahl der Neuronen und Schichten vorgenommen. Anschließend wird untersucht, ob das Verwenden CNNs ähnliche oder bessere Ergebnisse liefert, besonders in Hinblick auf die Performance. In Tabelle 4.1 sind die durchgeführten Experimente aufgelistet.

Das Ergebnis aus den Trainingsversuchen mit rein-realen Datensätzen ist, dass CNNs die Bewegungen wesentlich besser erkennen können als die Fully-Connected-Modelle. Nicht nur, dass sie hier eine höhere Genauigkeit aufweisen, sie lernen die Verteilung auch wesentlich schneller. Während die Fully-Connected-Modelle 300 Epochen für eine Genauigkeit von 90-97% benötigen, benötigen die CNNs lediglich 9-80 Epochen für bessere Resultate. Der einzige Nachteil besteht in ihrer Ausführungsgeschwindigkeit, die bis zu vier mal kleiner ist.

Erstaunlicherweise verhält sich dies bei den Trainingsversuchen mithilfe eines GANs ein wenig anders. Grundsätzlich erreichten die Fully-Connected-Modelle eine wesentlich höhere Genauigkeit nach sehr viel weniger Trainingsepochen. Die Genauigkeit wurde dabei nicht mithilfe von synthetischen Daten des GANs gemessen, sondern es wurden hierfür wieder echte Daten verwendet. Damit wurde sichergestellt, dass die Genauigkeiten der verschiedenen Trainingsmethoden vergleichbar gehalten werden. Das Verwenden von künstlich erzeugten Daten liefert in diesem Fall also immer bessere Ergebnisse. Dies liegt vermutlich unter anderem daran, dass GANs die Eigenschaft besitzen, zwischen ech-

name	layers	epochs	p_{real}	epochs	p_{gan}	inference speed
linear	Dense, 128 ReLU Dense, 3	300	0.90	82	0.99	$0{,}59{\cdot}10^{-6}$s
	Dense, 256 ReLU Dense, 3	300	0.94	49	0.99	$0{,}61{\cdot}10^{-6}$s
	Dense, 128 ReLU Dense, 256 ReLU Dense, 3	300	0.97	70	0.99	$0{,}83{\cdot}10^{-6}$s
conv	Conv2D, 16, 3×3, stride 2 Conv2D, 32, 3×3, stride 2 Dense, 3	81	0.99	123	0.99	$1{,}25{\cdot}10^{-6}$s
	Conv2D, 16, 3×3, stride 2 Conv2D, 32, 3×3, stride 2 Conv2D, 64, 3×3, stride 2 Dense, 3	44	0.99	121	0.99	$1{,}74{\cdot}10^{-6}$s
	Conv2D, 16, 3×3, stride 2 Conv2D, 32, 3×3, stride 2 Conv2D, 64, 3×3, stride 2 Conv2D, 128, 3×3, stride 2 Dense, 3	9	0.99	126	0.99	$2{,}27{\cdot}10^{-6}$s

Tabelle 4.1: Durgeführte Experimente mit verschiedenen Architekturen für die Bewegungserkennung. Die Eingabe besteht aus 60 Frames einer Bewegungsanimation aus 17 Schlüsselpunkten mit jeweils 3 Eigenschaften (x-, y-Koordinaten und Wahrscheinlichkeitsscore) kodiert als Bild (siehe Abbildung 3.2). Als Ausgabe sollen die Netze die geschätzte Klasse, also das Label liefern. p_{real} gibt die Genauigkeit des jeweiligen Modells an, das mit einem realen Datensatz trainiert wurde. p_{gan} gibt hingegen die Genauigkeit der Modelle an, die mithilfe eines KpGAN-Generators trainiert wurden. Für das Messen der Genauigkeit wurden immer reale Daten verwendet.

ten Daten zu interpolieren und damit automatisch eine Augmentation durchzuführen. Dies führt dazu, dass die Modelle eine höhere Vielfalt lernen können. Diese Vielfalt ist im echten Datensatz nicht so ausgeprägt. Hierbei ist wichtig zu erwähnen, dass das Daten erzeugende GAN auf eben diesen Datensatz trainiert wurde. Der Datensatz wurde also verbessert, indem ein GAN verwendet wurde, die Verteilung zu lernen und zu interpolieren, um eine größere Datenvielfalt zu erzeugen.

5 Entwicklung einer mobilen Anwendung

Auf Basis der ausgearbeiteten Ergebnisse aus Kapitel 4 soll nun eine mobile Anwendung entwickelt werden. Ziel ist es die theoretischen Schlussfolgerungen mit einem praktischen Experiment zu verifizieren. Da sich diese Arbeit vor allem mit dem Problem beschäftigen soll, wie solche Modelle auf mobilen Plattformen überführt werden können, soll eine Android-App entwickelt werden, um die Ergebnisse zusammenzufassend zu präsentieren. Android wird als Plattform gewählt, da es zur Zeit die am häufigsten vertretene mobile Plattform ist und ein entsprechendes Gerät leicht zur Verfügung steht. Zum Vergleich, Android besitzt einen Marktanteil von 72,84%, iOS einen von 26,34% und 0,82% werden von sonstigen Plattformen gehalten[1].

Die Anforderungen an die App sind recht simpel. Es sollen über die Kamera Bewegungen identifiziert werden, wobei die erkannte Bewegungsart angezeigt werden soll. Das Zeitfenster, in welches Bewegungen erkannt werden sollen, wird auf 60 Frames festgelegt. Bei einer Kamera, die 30 Bilder pro Sekunde aufnehmen kann, wird also vorausgesetzt, dass die Bewegung innerhalb von zwei Sekunden eindeutig identifizierbar ist. Zusätzlich muss berücksichtigt werden, dass die Abtastrate für die Bewegungserkennung entsprechend hoch, also die Ausführungsdauer der Machine-Learning-Modelle möglichst gering sein soll. Der Grund dafür ist, dass die Modelle den Flaschenhals der Anwendung darstellen und eine hohe Abtastrate nur mit entsprechend schnellen Netzwerken möglich ist.

5.1 Implementierungsdetails

Der Einfachheit halber werden zwei KNNs verwendet, um eine Bewegung zu identifizieren. Das eine Netzwerk hat die Aufgabe, Schlüsselpunkte des menschlichen Körpers

[1] https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009/ (besucht am 13.08.2021)

in Bildern zu erkennen. Diese werden anschließend in einen Puffer mit maximal 60 Elementen zwischengespeichert. Wird ein Element in den Puffer hinzugefügt, so werden alle anderen Elemente zuerst um eine Position nach hinten (rechts) verschoben. Der erste Slot ist nun dementsprechend leer und wird von dem neuen Element belegt. Ist der Puffer bereits voll, so wird das letzte Element entfernt. Ein Element dieses Puffers sind 17 Schlüsselpunkte des menschlichen Körpers. Dieser Puffer bildet damit die Eingabe für das zweite Netzwerk. Dieses hat die Aufgabe, Schlüsselpunkte aus 60 Kamerabildern einer Bewegungsklasse zuzuordnen.

Für die Schlüsselpunkterkennung wird MoveNets Lightning-Architektur [20] verwendet. Diese wurde speziell für mobile Geräte entwickelt, sodass die Erkennung von Schlüsselpunkten in Echtzeit durchgeführt werden kann. Hier gilt es zu testen, ob die Performance in der Tat ausreichend ist, um auf modernen, aber leistungsärmeren Geräten in Echtzeit zu laufen. Für die Bewegungserkennung werden die verschiedenen Architekturen aus Kapitel 4 verwendet und getestet. Auch hier ist zu überprüfen, ob diese in Zusammenarbeit mit MoveNet zu einer ausreichenden Performance kommen. Eine ausreichende Performance wird dann angenommen, wenn entsprechende Bewegungen über die Kamera bzw. über die entwickelte App richtig erkannt wurden.

Für die Entwicklung der Android-App wurde *Kotlin* verwendet. Zudem wird die *TensorFlow-Lite-API* (TFLite) verwendet, um die exportierten Modelle auf Android auszuführen. Die in dieser Arbeit implementierten Modelle wurden alle mit der *TensorFlow-API* trainiert. Da es sich dabei jedoch um ein Desktop Framework handelt, wurden die trainierten Modelle im *Saved-Model*-Format gespeichert und anschließend in das TFLite-Format umgewandelt.

Es soll nun der allgemeine Aufbau der mobilen Anwendung besprochen werden. Im ersten Schritt wurde sich Gedanken über mögliche Klassen der Anwendung gemacht. Diese sind im UML-Klassendiagramm in Abbildung 5.2 zu sehen. Als Einstiegspunkt in dieser, sowie in jeder anderen Android-App, dient die `MainActivity`-Klasse. Hier werden vor allem die View-Elemente der App initialisiert. Auch die Modelle für das Erkennen von Bewegungen werden hier geladen. Dies ist zum einen das MoveNet-Lightning-Modell[2] für die Schlüsselpunkterkennung und zum anderen das MotionNet aus Kapitel 4. Warum die beiden Modelle nicht in ein einziges Modell zusammengefügt worden sind, hat den Grund, dass die Schlüsselpunkte in den dazugehörigen Puffer gespeichert werden sollen. Dieser wird dann anschließend vom MotionNet ausgelesen und interpretiert, sodass eine

[2]MoveNet-Lightning, https://tfhub.dev/google/movenet/singlepose/lightning/4

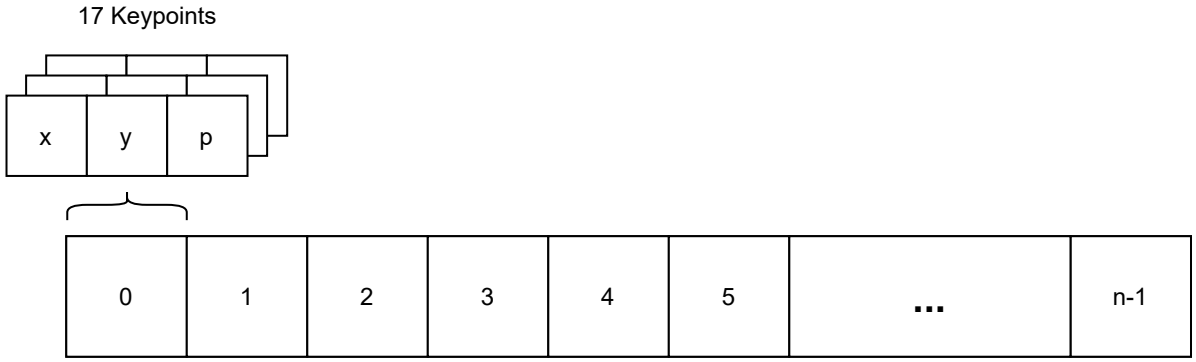

Abbildung 5.1: Schematische Darstellung des Puffers der Bewegungserkennung, welcher $n = 60$ Schlüsselpunkte des menschlichen Körpers speichert.

Bewegungserkennung stattfinden kann. Die Ausgabe von einem Schlüsselpunktdetektor kann demnach nicht direkt als Eingabe für das MotionNet dienen. Dies liegt natürlich auch daran, dass versucht wird, eine Bewegung in Echtzeit zu erkennen. Deshalb ist es notwendig, dass die Bilder der Kamera auch in Echtzeit analysiert werden. Erst wenn diese Abtastung ein bestimmtes Zeitfenster beinhaltet, kann eine zeitabhängige Änderung einer Pose erkannt werden. Die beiden geladenen Modelle werden nun in eine dafür vorgesehene Klasse verwaltet. `MotionDetector` soll die Erkennungslogik implementieren, was unter anderem die Implementierung des Schlüsselpunktpuffers (siehe Abbildung 5.1) bedeutet. Diese Klasse wird abstrakt umgesetzt, damit neue Modelle zum Erkennen von Bewegungen einfach von ihr erben und entsprechende Methoden überschreiben können, um den Weg in die Anwendung leichter finden zu können. Als Beispiel hierfür wurde das `MotionNet` implementiert, welches nun von der abstrakten Klasse erbt. Hier werden die Modelle als *Interpreter* der *TensorFlow-Lite-API* verwaltet. Die Inferenz der Modelle wird schließlich in den zu überschreibenden Methoden ausgeführt. Damit die Ausführung der Modelle überhaupt stattfinden kann, müssen Bilder über die Kamera in die Anwendung geleitet werden. Dies wird mithilfe der implementierten `CameraManager`-Klasse umgesetzt. Diese stellt das aktuelle Bild der Kamera über ein dafür vorgesehenes Steuerelement dar und bietet eine Schnittstelle zum Abfangen einzelner Frames. Diese Frames werden anschließend über die `MainActivity` in die Bewegungserkennung geleitet. Nach der Inferenz wird das User-Interface mithilfe der `MainActivity` aktualisiert und die Ergebnisse dargestellt. Speziell für die Darstellung der Schlüsselpunkte und der Bewegungsart wird die View-Klasse `PersonOverlay` umgesetzt. Diese stellt alle aus der Bewegungserkennung resultierenden Ergebnisse grafisch dar.

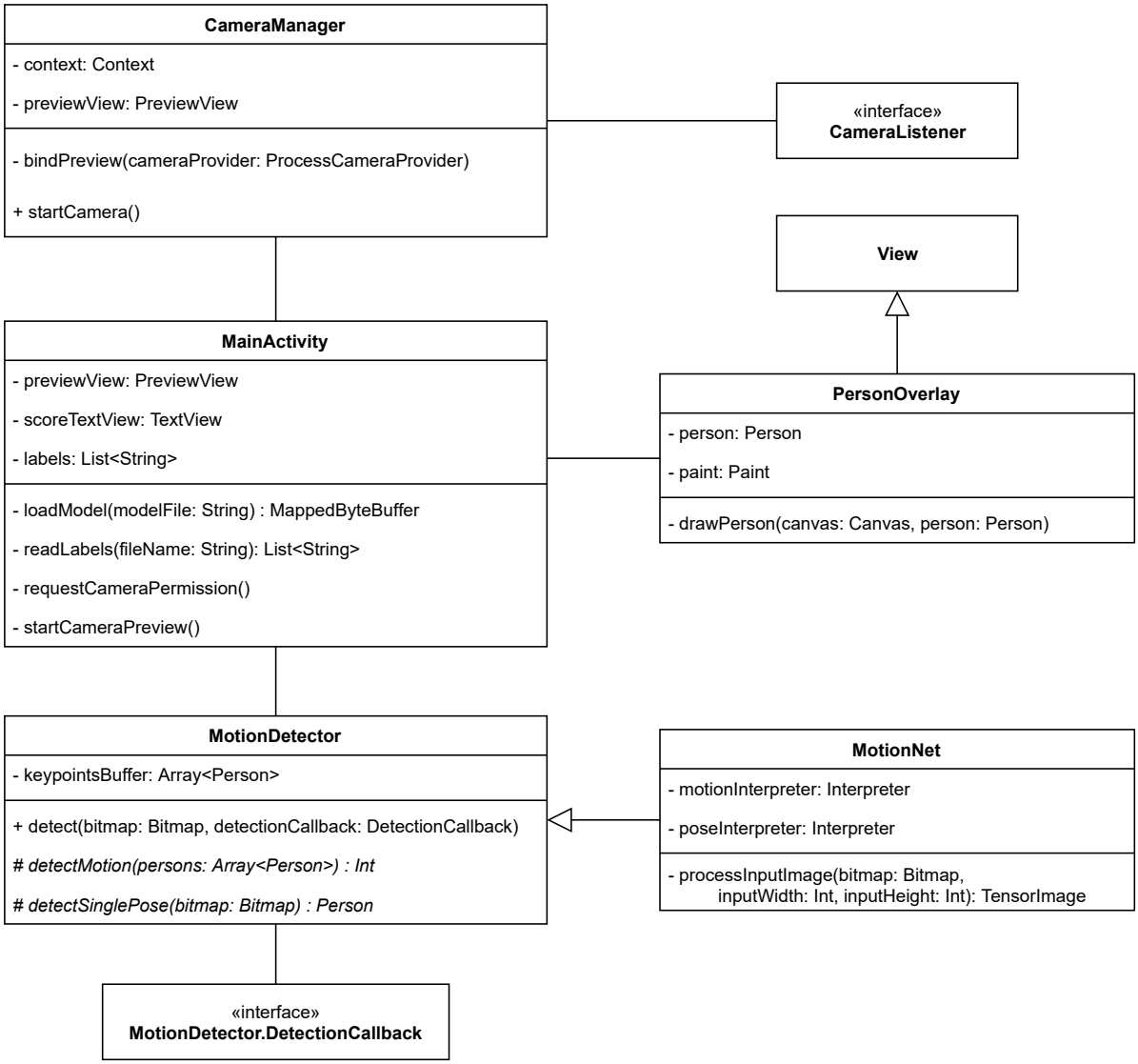

Abbildung 5.2: UML-Klassendiagramm der mobilen Anwendung zum Testen der Machine-Learning-Modelle für die Bewegungserkennung.

5.2 Testaufbau und Ergebnisse

Die Bewegungserkennung aus den vorherigen Abschnitten soll nun mithilfe der implementierten mobilen Applikation in der Praxis getestet werden. Hierfür muss vorerst der Testaufbau besprochen werden, sodass die Ergebnisse nachvollziehbar und reproduzierbar sind. Als Plattform wird Android zusammen mit ausgewählten mobilen Geräten verwendet. Ziel ist es herauszufinden, ob die Bewegungserkennung auf modernen, aber auch auf älterer Hardware in Echtzeit ausführbar ist.

6 Fazit und Ausblick

In dieser Thesis wurden verschiedene Modelle des maschinellen Lernens entwickelt und evaluiert. Vor allem wurden sogenannte Generative-Adversarial-Networks (GANs) erforscht und neue Methoden zum Erzeugen von künstlichen Bewegungen gesucht, um einen kompletten Datensatz mithilfe eines solchen Modells zu erzeugen. Als Ergebnis wurde zuerst ViGAN präsentiert. Ziel von ViGAN ist es, aus einem bestehenden Video-Datensatz neue bisher unbekannte Proben zu erzeugen. Dies gelang auch bis zu einem gewissen Grad. Es wurden zwar neue Videos vom Netzwerk erzeugt, jedoch konnte nicht sichergestellt werden, ob sich eine Person tatsächlich bewegt oder still steht. So sind Personen oft regungslose starre Körper, während sich z.B. ganze Räume über die Zeit verändern. Als Alternative dazu wurde KpGAN entworfen, welches direkt verschiedenste Bewegungsanimationen erzeugen kann. Diese Bewegungsanimationen bestehen dabei aus menschlichen Schlüsselpunkten und werden als Bilder kodiert. Experimente haben gezeigt, dass Datensätze, die komplett mithilfe von KpGAN erzeugt wurden, mindestens genau so gut für das Training anderer Modelle geeignet sind, wie entsprechende reale Datensätze. Dies bringt einen großen Vorteil mit sich, denn der eigentliche Datensatz kann ohne großen Aufwand beliebig erweitert werden, während beim realen Datensatz ein signifikanter Zeitaufwand entstehen würde.

Im Anschluss wurde mithilfe von KpGAN weitere neuronale Netzwerke trainiert, die Bewegungen aus Schlüsselpunktsequenzen erkennen sollen. Die Herausforderung ist dabei die Echtzeiterkennung auf mobilen Geräten. Die verwendeten Metriken waren Genauigkeit und Performanz. Aus denExperimenten resultierte schließlich das MotionNet, welches eine sehr gute Erkennrate und Ausführungsgeschwindigkeit besitzt. Mithilfe des trainiertenMotionNets und eines externen Modells von Google (MoveNet-Lightning) wurde anschließend eine mobile App entwickelt, welche die Anwendbarkeit und Performanz auf mobilen Geräten messen soll. Auf modernen Smartphones wurde eine Ausführungsgeschwindigkeit von bis zu 14 FPS gemessen. Damit wurde eindeutig bewiesen, dass eine Bewegungserkennung auf mobilen Plattformen in Echtzeitausführbar ist.

Weiterführende Arbeiten können sich in Zukunft mit dem Verbessern des ViGAN beschäftigen, um das Problem mit den sich nicht bewegenden Personen in Videos zu lösen. Zudem kann das MotionNet um weitere Bewegungen erweitert werden.

Literatur

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

[2] Emad Barsoum, John Kender und Zicheng Liu. *HP-GAN: Probabilistic 3D human motion prediction via GAN*. 2017. arXiv: 1711.09561 [cs.CV].

[3] Jia Deng u. a. „Imagenet: A large-scale hierarchical image database“. In: *2009 IEEE conference on computer vision and pattern recognition*. Ieee. 2009, S. 248–255.

[4] Li Deng. „The MNIST Database of Handwritten Digit Images for Machine Learning Research [Best of the Web]“. In: *IEEE Signal Processing Magazine* 29.6 (2012), S. 141–142. DOI: 10.1109/MSP.2012.2211477.

[5] Ian J. Goodfellow u. a. *Generative Adversarial Networks*. 2014. arXiv: 1406.2661 [stat.ML].

[6] Ishaan Gulrajani u. a. „Improved Training of Wasserstein GANs“. In: *CoRR* abs/1704.00028 (2017). arXiv: 1704.00028. URL: http://arxiv.org/abs/1704.00028.

[7] Kaiming He u. a. „Deep Residual Learning for Image Recognition“. In: *CoRR* abs/1512.03385 (2015). arXiv: 1512.03385. URL: http://arxiv.org/abs/1512.03385.

[8] Andrew G. Howard u. a. „MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications“. In: *CoRR* abs/1704.04861 (2017). arXiv: 1704.04861. URL: http://arxiv.org/abs/1704.04861.

[9] Sergey Ioffe und Christian Szegedy. „Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift“. In: *Proceedings of the 32nd International Conference on Machine Learning*. Hrsg. von Francis Bach und David Blei. Bd. 37. Proceedings of Machine Learning Research. Lille, France: PMLR, Juli 2015, S. 448–456. URL: http://proceedings.mlr.press/v37/ioffe15.html.

[10] Tsung-Yi Lin u. a. „Feature Pyramid Networks for Object Detection“. In: *CoRR* abs/1612.03144 (2016). arXiv: 1612.03144. URL: http://arxiv.org/abs/1612.03144.

[11] Tsung-Yi Lin u. a. „Microsoft COCO: Common Objects in Context“. In: *CoRR* abs/1405.0312 (2014). arXiv: 1405.0312. URL: http://arxiv.org/abs/1405.0312.

[12] Mehdi Mirza und Simon Osindero. „Conditional Generative Adversarial Nets“. In: *CoRR* abs/1411.1784 (2014). arXiv: 1411.1784. URL: http://arxiv.org/abs/1411.1784.

[13] Alec Radford, Luke Metz und Soumith Chintala. *Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks*. 2016. arXiv: 1511.06434 [cs.LG].

[14] Eitan Richardson und Yair Weiss. *On GANs and GMMs*. 2018. arXiv: 1805.12462 [cs.CV].

[15] Mark Sandler u. a. „Inverted Residuals and Linear Bottlenecks: Mobile Networks for Classification, Detection and Segmentation“. In: *CoRR* abs/1801.04381 (2018). arXiv: 1801.04381. URL: http://arxiv.org/abs/1801.04381.

[16] Connor Shorten und Taghi M. Khoshgoftaar. „A survey on Image Data Augmentation for Deep Learning“. In: *Journal of Big Data* 6.1 (Juli 2019), S. 60. ISSN: 2196-1115. DOI: 10.1186/s40537-019-0197-0. URL: https://doi.org/10.1186/s40537-019-0197-0.

[17] Christian Smolik. *Trittfrequenz*. 2000. URL: http://www.smolik-velotech.de/glossar/tu_TRITTFREQUENZ.htm (besucht am 14. 08. 2021).

[18] Khurram Soomro, Amir Roshan Zamir und Mubarak Shah. „UCF101: A Dataset of 101 Human Action Classes From Videos in The Wild“. In: CRCV-TR-12-01, Nov. 2021.

[19] Jost Tobias Springenberg u. a. *Striving for Simplicity: The All Convolutional Net*. 2015. arXiv: 1412.6806 [cs.LG].

[20] Ronny Votel und Na Li. *Next-Generation Pose Detection with MoveNet and TensorFlow.js*. 2021. URL: https://blog.tensorflow.org/2021/05/next-generation-pose-detection-with-movenet-and-tensorflowjs.html (besucht am 11. 08. 2021).

[21] Ce Zheng u. a. „Deep Learning-Based Human Pose Estimation: A Survey“. In: *CoRR* abs/2012.13392 (2020). arXiv: 2012.13392. URL: https://arxiv.org/abs/2012.13392.

[22] Xingyi Zhou, Dequan Wang und Philipp Krähenbühl. „Objects as Points“. In: *CoRR* abs/1904.07850 (2019). arXiv: 1904.07850. URL: http://arxiv.org/abs/1904.07850.