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

Master-Thesis

Florian Hansen
Hochschule Flensburg

10. August 2021

Inhaltsverzeichnis

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.

Kapitel 3 erläutert anschließend die Funktionsweise von GANs und entsprechende Loss-Funktionen zum Trainieren dieser Netzwerke. Zusätzlich werden Probleme der einzelnen Architekturen besprochen und Lösungen vorgestellt.

Kapitel 4 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.

Kapitel 5 führt die Bewegungserkennung ein und vergleicht unter anderem verschiedene Ansätze zum Erkennen von menschlichen Schlüsselpunkten. Dabei wird auf Single- und Multi-Pose-Detection eingangen und mit dessen Hilfe eine neue Netzwerkarchitektur definiert, die in der Lage ist, eine Folge von menschlichen Posen zu klassifizieren und analysieren.

2 Grundlagen

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.

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 [arjovsky2017wasserstein]) *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 Gewichte 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 [arjovsky2017wasserstein]) *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.

Definition 4 (Wasserstein-Abstand [arjovsky2017wasserstein]) *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.

3 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 [**barsoum2017hpgan**] 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 [**goodfellow2014generative**].

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

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.

3.1 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 [**richardson2018gans**]. 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 [**goodfellow2014generative**] hinzugefügt.

3.2 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 [**radford2016unsupervised**] 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 [**springenberg2015striving**] einfach Convolutional-Layer mit erhöhtem Stride verwendet werden, ohne dass die Genauigkeit sinkt. In Bezug zu DCGANs von [**radford2016unsupervised**] 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 [**radford2016unsupervised**].

Um Mode-Collapse zu vermeiden, verwendet [**radford2016unsupervised**] Batch-Normalization-Layer. Dadurch wird das Training stabilisiert und Probleme wie *Internal-Covariate-Shifting* angegangen [**pmlr-v37-ioffe15**]. Vor allem wird dadurch aber auch verhindert, dass der Generator immer die gleichen Ausgaben erzeugt. Das Anwenden der Batch-

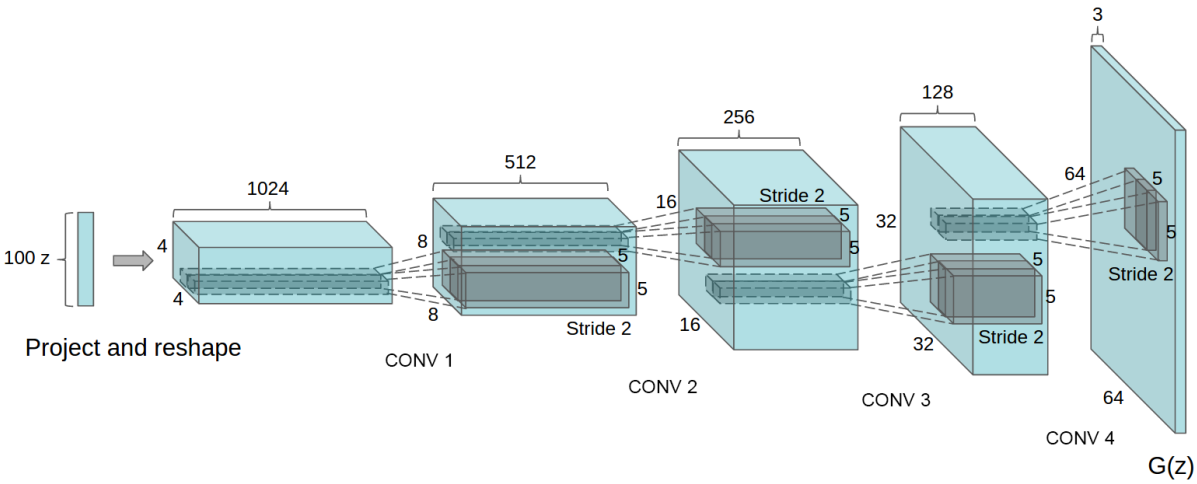

Abbildung 3.1: DCGAN-Architektur des Generators von [**radford2016unsupervised**]. Als Eingabe dient ein 100-dimensionaler Vektor, dessen Elemente zufällig gewählt werden. Dieser wird dann in den ersten Schichten umgeformt und durch vier Convolutional-Layer auf die Form 3×64×64 gebracht. Die Strides geben dabei den Vergrößerungsfaktor pro Convolution-Schicht an, während die Anzahl der Filter den Farbkanälen entsprechen.

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 [**radford2016unsupervised**] 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.

3.3 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 [**arjovsky2017wasserstein**] 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-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 [**arjovsky2017wasserstein**] c relativ klein sein.

Algorithmus 1 : Wasserstein GAN nach [**arjovsky2017wasserstein**]. 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

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 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 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 [**goodfellow2014generative**] 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 [**arjovsky2017wasserstein**] zeigen (siehe Abbildung 3.2). 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

Abbildung 3.2: Vergleich von WGAN und Standard-GAN [**arjovsky2017wasserstein**]. 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.

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.

3.4 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 [**gulrajani2017improved**] 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 3.3 zeigt.

Um das Problem des Weight-Clippings anzugehen, stellt [**gulrajani2017improved**] 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.

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

Der Kritisierer ist durch diese Änderung nun wesentlich besser dazu in der Lage, kom-

plexere Verteilungen zu erlernen.

Algorithmus 2 : WGAN mit Gradient-Penalty [**gulrajani2017improved**].

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

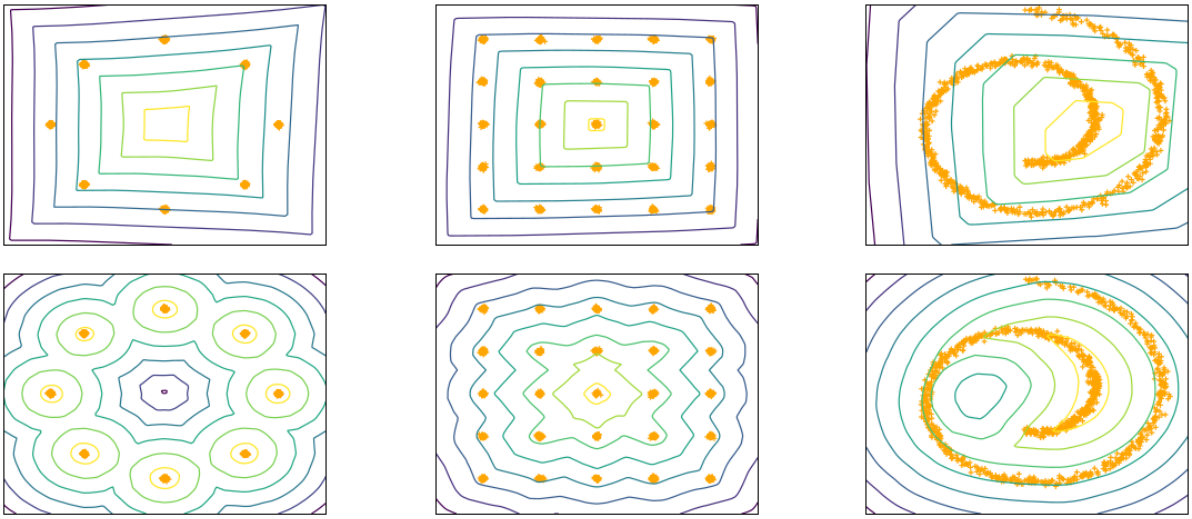

Abbildung 3.3: Vergleich zwischen Weight-Clipping (oben) und Gradient-Penalty (unten). Man erkennt deutlich, dass die Separierung der Ausgangsverteilung, dargestellt durch orangene Punkte, durch Weight-Clipping in sehr vereinfachte Funktionen resultiert. Gradient-Penalty lässt hingegen komplexere Strukturen von Verteilungen zu [**gulrajani2017improved**].

4 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. Eine Machine-Learning-Modell ist nur so gut wie der verwendete Datensatz. Dieser muss deshalb eine große Varianz der repräsentierten Daten besitzen. 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 [**6296535**], 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.

In diesem Abschnitt soll nun ein Datensatz eingeführt werden, welcher Daten für die Bewegungserkennung unterstützt. Um diesen Datensatz zu erstellen, muss vorab geklärt werden, was eine Bewegung eigentlich ist. Betrachtet man einen Punkt im Raum, dann beschreibt eine Bewegung 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 aufgenommen werden, die letzlich Videos darstellen. Da der Datensatz später dazu verwendet werden soll, um 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 Erkennungsraten 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 Expe-

riment 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 geschiet und vom Computer anstatt vom Menschen durchgeführt wird. 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.

4.1 Architektur von ViGAN

In diesem Abschnitt wird die Architektur von ViGAN vorgestellt und die Idee dahinter erläutert. ViGAN steht für Video-Generative-Adversarial-Network und dient zum Erzeugen von neuen Videos aus einem bestehendem Datensatz. 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 3 werden verschiedene GANs und deren Probleme bzw. Vorteile vorgestellt. Aufgrund der Vorteile von Wasserstein-GANs, nämlich die Interpretationsmöglichkeit des Losses

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 4.1 dargestellt unterscheidet sich ViGAN nicht allzu sehr vom WGAN. Der größte Unterschied besteht darin, dass 3D-Convolutional-Layer anstelle der 2D-Convolutional-Layer verwendet werden. Dies ermöglicht nicht nur ein Upsampling des Bildes, sondern auch die Anzahl der Frames. Die Idee dabei ist, dass das Netzwerk dadurch lernt, zusammenhängende Frames zu generieren. Als Eingabe wird ein zufälliger Vektor 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 $(5, 6, 8, 256)$ 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-Layers vergrößert und die Frameanzahl erhöht. Der Stride beträgt dabei 2, sodass eine Verdoppelung der Dimensionen stattfindet. Insgesamt werden vier solcher Schichten hinereinander gereiht. Dabei entsteht ein Ausgabevideo mit 80 Frames, wobei jeder Frame jeweils $96 \times 128 \times 3$ Pixelwerte besitzt.

4.2 Training und Diskussion der Ergebnisse

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 Trainigsdaten in den Speicher der Grafikkarte zu laden, da das Modell und die Daten entsprechend groß sind. Daher kann nur mit einer Batch-Größe von 4 gearbeitet werden. Insgesamt wurden 8 Experimente durchgeführt, wobei sich die Architektur aus Tabelle 4.2 am stabilsten herausstellte. 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

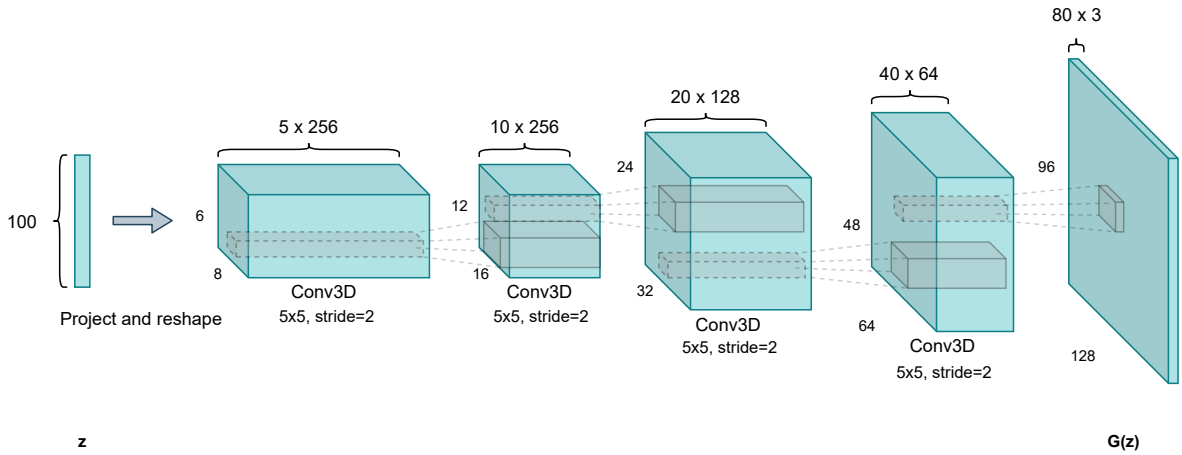

Abbildung 4.1: Architektur von ViGAN. Als Eingabe dient ein zufälliger Vektor $z \in \mathbb{R}^{100}$. Dieser wird anschließend in fünf $6 \times 8 \times 256$ Features umgewandelt. Um die relativ kleinen Frames in eine größere Auflösung zu skalieren, werden vier 3D-Convolutional-Layer verwendet. Jedes 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.

der Inneneinrichtung vorhanden. So verschwindet während des Videoclips beispielsweise eine Tür und es erscheint ein Sofa. 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.

4.3 Architektur von MotionGAN

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 4.1: 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.

5 Bewegungserkennung

5.1 Erkennung von Bewegungsarten

5.2 Erkennung von Anomalien

5.3 Erkennung von Eigenschaften

5.4 Vorhersage von Bewegungen

5.5 Architektur einer mobilen Anwendung

6 Fazit und Ausblick