



Fakultät V – Verkehrs- und Maschinensysteme
Institut für Werkzeugmaschinen und Fabrikbetrieb
Fachgebiet Handhabungs- und Montagetechnik
Prof. Dr.-Ing. Franz Dietrich

Masterarbeit

Entwicklung und Optimierung einer skalierbaren und adaptiven Diffusion-Policy für Robotermanipulation

Vorgelegt von:

Tim Goerschel

Matrikelnummer 382201

Studiengang: Produktionstechnik

t.goerschel@campus.tu-berlin.de

Berlin, den 4. Januar 2025

Erstprüfer*in

Dr. Ing. Arne Glodde

Zweitprüfer*in

Prof. Dr.-Ing. Dirk Oberschmidt

Ehrenwörtliche Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und eigenhändig sowie ohne unerlaubte fremde Hilfe und ausschließlich unter Verwendung der aufgeführten Quellen und Hilfsmittel angefertigt habe.

Nutzung von KI-Tools [Variante 1 ohne KI-Tools]

Ich habe keine Outputs von [Text-, Bild-, oder Codegenerierenden] KI Tools in der Ausarbeitung verwendet.

Nutzung von KI-Tools [Variante 2 mit KI-Tools]

Ich habe nur die erlaubten und dokumentierten Hilfsmittel benutzt. Ich verantworte die Auswahl, Übernahme und sämtliche Ergebnisse des von mir verwendeten Outputs vollständig selbst. Ich versichere, dass die Kennzeichnung des KI-Einsatzes vollständig ist. Im Anhang "Übersicht verwendeter Hilfsmittel" habe ich die verwendeten KI-Tools mit ihrem Produktnamen aufgeführt. Des Weiteren werden darin

*die von mir verwendeten Prompts aufgeführt,

[oder]

*sämtliche KI-generierten Outputs einzeln aufgeführt [z.B. Links auf Promptverläufe],

[oder]

* die Nutzung der KI-tools dokumentiert [siehe Beispiel Tabelle], die relevant für die Aufgabe waren.

4. Januar 2025/ *Tim Fackel*

(Datum / Unterschrift)

Inhaltsverzeichnis

Ehrenwörtliche Erklärung	i
Kurzfassung	v
Abstract	vi
Abbildungsverzeichnis	vii
Abbildungsverzeichnis	vii
Tabellenverzeichnis	ix
Tabellenverzeichnis	ix
Symbolverzeichnis	x
Abkürzungsverzeichnis	xiv
1 Einleitung	1
1.1 Herausforderungen	2
1.2 Lösungsansatz	2
2 Theorie	3
2.1 KI und Maschinelles Lernen	3
2.2 Abgrenzung des Imitationslernen zum Reinforcement Learning	3
2.3 Imitationslernen	3
2.3.1 Vorteile des Imitationslernens	4
2.3.2 Nachteile des Imitationslernens	4
2.3.2.1 Nachteil der Fehlerakkumulation über die Zeit	5
2.3.3 Identifiziertes Verbesserungspotential	5
2.4 Reinforcement Learning	6
2.4.1 Vorteile des Reinforcement Learning	6
2.4.2 Vorteile von RL bei Aufgaben mit kritischen Verhaltensschritten . . .	6
2.4.3 Nachteile des Reinforcement Learning	7
2.5 Unterschied	7
2.6 Gemeinsamkeiten	8
2.7 Zusammenfassung der Gegenüberstellung	9
2.8 Diffusionsprozess im Zusammenhang mit der Diffusion Policy	9

2.9	Denoising Diffusion Probabilistic Models (DDPM)	10
2.10	Tricks zur Verbesserung des Modells	19
2.11	Weiterführende Tricks zur Verbesserung des Modells	20
2.12	Architektur und Prozessanpassungen für die Diffusion Policy	24
3	Stand der Technik	33
3.1	Verbesserung durch Rekonstruieren der visuellen Repräsentation	33
3.2	Generalisierung und dateneffizientes Lernen	33
3.3	Finetunen und Verkürzen des Entrauschungsprozesses	34
4	Methodik	37
4.1	Experimentelle Infrastruktur	37
4.2	Diffusion Policy Implementierung	37
4.3	Experimentelle Methodik	37
4.3.1	Aufnahme von Trainingsdaten	37
4.3.2	Evaluationsschleife	39
4.3.3	Evaluationsframework	40
4.4	Technische Implementierung	40
4.5	Datenverarbeitung und Training	40
5	Verbesserungspotentiale und Implementierung	41
5.1	Trainingsdaten und Qualität	41
5.2	Potential 1: Segmentierungsmasken	42
5.2.1	Implementierung	42
5.2.1.1	Datenaufnahme	43
5.2.2	Ergebnisse und Interpretation	44
5.3	Potential 2: Trainingsdaten	45
5.3.1	Potentiale	45
5.3.2	Implementierung der Bewertungsmetriken (2b)	46
5.3.3	Aufnahme des besseren Datensatzes 3	48
5.3.4	Abgeleitetes Potential: Kritische Bereiche erkennen und entsprechend handeln	48
5.3.5	Ergebnisse und Interpretation	49
5.4	Potential 3: Inferenzgeschwindigkeit	51
5.4.1	Potentiale	52
5.5	Implementierung der progressiven Destillation (3a)	52
5.5.1	Ansatz 3a-1 - Klassische Destillation über Inferenzschrittskalierung . .	52
5.5.1.1	Überlegungen zur Parametrisierung der Rauschvorhersage . .	53
5.5.1.2	Warum die Vorhersage in wenigen Schritten problematisch ist	54

5.5.1.3	Optimaler Rekonstruktionsverlust	55
5.5.1.4	Anpassung des Netzwerks an die Destillationsmethode	56
5.5.1.5	Anpassung des Trainingsprozess an die Destillationsmethode	56
5.5.1.6	Erklärung des inferenzzschrittbasierten herkömmlichen De- stillationsalgorithmus	56
5.5.2	Ansatz 3a-2 - Alternative Destillation über Temperaturskalierung . . .	57
5.5.2.1	Erklärung des Algorithmus	58
5.5.3	Ansatz 3a-3 - Destillation der Temperatur und Inferenzschritte	60
5.5.4	Ergebnisse der Tests von Ansatz 3a-2 und 3a-3	60
5.5.5	Anwendung von 3a-1 auf eine zeitkritischen Aufgabe	64
5.5.5.1	Anforderungen an die zeitkritische Aufgabe	64
5.5.5.2	Notwendige Prozess-Anpassungen für die Würfel-Aufgabe . .	65
5.5.5.3	Ergebnisse des zeitkritischen Würfelschiebens	65
6	Fazit und Ausblick	67

Literaturverzeichnis	
----------------------	--

Kurzfassung

Herkömmliche Diffusion Policies zur Robotersteuerungen stoßen bei komplexen zeitkritischen Anwendungen an ihre Grenzen. Der Diffusions-Ansatz überträgt das Konzept der Diffusionsprozesse, das ursprünglich aus der Bildgenerierung stammt, auf die robotische Aktionsgenerierung. Dabei werden Handlungssequenzen durch einen kontrollierten Verrauschungs- und Entrauschungsprozess erzeugt. Die Methode eignet sich besonders für multimodale Lösungsräume, in denen mehrere valide Lösungswege existieren, wie beispielsweise beim Umfahren von Hindernissen. Ein wesentlicher Vorteil besteht darin, dass das System ausschließlich anhand von Bildern und Robotergelenkpositionen lernt, ohne dass komplexe Belohnungsfunktionen definiert werden müssen. Die vorliegende Arbeit untersucht im Wesentlichen drei Potentiale. Lernen aus segmentierten Bildern führt zu einer Verbesserung von 1.3 % gegenüber der Baseline. Zweitens sollten Metriken zur Bewertung aufgenommener Trainingsdaten zur frühzeitigen Aussage über die Modellqualität definiert werden. Diese haben sich als valide erwiesen. D.h. man kann anhand dieser die Erfolgsrate abschätzen. Das dritte Potential betrifft die Inferenzbeschleunigung: Destillation des Modells im Probenraum führt bei der Auswertung in zwei Inferenzschritten zu einer vierfach schnelleren Inferenzzeit bei ähnlicher Erfolgsrate. Das destillierte Modell löst zeitkritische Aufgaben in 57.1 % der Fälle, während die Baseline nur 0.6 % erreicht. Weiterhin kann durch den verbesserten Trainingsalgorithmus ein Modell trainiert werden, was in nur einem Inferenzschritt zusammenhängende Aktionsbahnen generiert, wohingegen die Baseline dies erst ab zwei Schritten schafft.

Abstract

Conventional diffusion policies for robot control reach their limits in complex time-critical applications. The diffusion approach transfers the concept of diffusion processes, which originally comes from image generation, to robotic action generation. Action sequences are generated through a controlled noise and de-noising process. The method is particularly suitable for multimodal solution spaces in which several valid solution paths exist, such as when avoiding obstacles. A major advantage is that the system learns solely from images and robot joint positions without the need to define complex reward functions. The present work essentially investigates three potentials. Learning from segmented images leads to an improvement of 1.3 % compared to the baseline. Secondly, metrics for the evaluation of recorded training data should be defined for early prediction of model quality. These have proven to be valid. In other words, they can be used to estimate the success rate. The third potential concerns inference acceleration: Distillation of the model in the sample space leads to a fourfold faster inference time with a similar success rate when evaluated in two inference steps. The distilled model solves time-critical tasks in 57.1 % of cases, while the baseline only achieves 0.6 %. Furthermore, the improved training algorithm can be used to train a model that generates coherent action trajectories in just one inference step, whereas the baseline only manages this from two steps.

Abbildungsverzeichnis

2.1	Fortpflanzung der Trajektorien-Fehler über die Zeit. H: der Horizont der Aufgabe (abgewandelt nach [RAI-22])	5
2.2	Gegenüberstellung von Imitationslernen und Reinforcement Lernen (abgewandelt nach [RAI-22])	8
2.3	Gegenüberstellung des Entrauschungsprozess bei der Bild-Diffusion und der Aktion-Diffusion. Abgewandelt nach [Chi-23]	10
2.4	Trainings-Algorithmus des DDPM, abgewandelt nach [Ho-20]	14
2.5	Sample-Algorithmus des DDPM, abgewandelt nach [Ho-20]	15
2.6	Geführte Diffusion anhand von zwei Klassen für ein einfaches Beispiel [Pie-23]	18
2.7	Rauschproben aus linearen (oben) und Cosinus (unten) Verrauschungsplänen mit linearen Zeitwerten t von 0 bis T [Nic-21]	20
2.8	Vergleich von Algorithmus 1 (Training des Diffusionsmodells) und Algorithmus 2 (progressive Distillation) nebeneinander, wobei die relativen Änderungen bei der progressiven Destillation grün hervorgehoben sind. (Bildquelle: [Sal-22]) .	22
2.9	Zeitliche Versetzung von Beobachtungen und vorhergesagten Aktionen: Beobachtungsdimension = 2, Aktionshorizont für ausgeführte Aktionen = 8, Aktionshorizont für vorhergesagte Aktionen = 16	25
2.10	Selbsterstellte Darstellung des DP-Prozesses. E_S : State Encoder, $h_{t,img}$: Visuelle Embeddings, h_t : Observations-Kondition, E_A : Action Encoder, D_A : Action Decoder, ϵ_θ : Geschätztes Rauschen, A_t^k : Verrauchte Aktionssequenz, A_t^{k-1} : Entrauchte Aktionssequenz, X_t^k : eine Repräsentation aus dem tiefsten CNN Layer, $X_{t,skip}^k$: Skip Connection Layer	27
2.11	Übersicht der Trainingsarchitektur nach [Zha-24]	28
2.12	Übersicht der U-Net Architektur nach [Li-24]	32
4.1	Prozess der Datenaufnahme	38
4.2	Aufbau der Pybullet Umgebung	39
5.1	Aufbau des Versuchs: Schiebe Würfel zum Ziel.	43
5.2	1000 Trajektorien für den Versuch: Schiebe Würfel zum Ziel. Mit RGB-Bildern (links) und mit segmentierten Bildern (rechts)	45
5.3	199 Trainingsbahnen für den Versuch: Schiebe Würfel zum Ziel.	46
5.4	199 Trajektorienverläufe aus dem Training für die zwei Datensätze. Rote Box: Die Anfangsregion der Würfel.	49

5.5 Vergleich der Lösungs-Trajektorienverläufe der zwei Datensätze. Datensatz 3 (links) und Datensatz 1 (rechts)	51
5.6 Naive Diffusions- und Flow-Matching-Modelle versagen bei der Generierung in wenigen Schritten. Trainings-Pfade werden durch die zufällige Paarung von Daten und Rauschen erzeugt (links). Flow-Matching Modelle lernen eine deter- ministische ODE - dabei sind ihre Pfade nicht gerade, sondern müssen schritt- weise angepasst werden. Die vorhergesagten Richtungen je Schritt weisen auf den Durchschnitt der plausiblen Datenpunkte hin. Je weniger Inferenzschritte, desto mehr werden die Generationen in Richtung des Mittelwerts des Datensat- zes verzerrt. Bei nur einem Inferenzschritt zeigt das Modell auf den Mittelwert des Datensatzes und kann daher keine multimodalen Daten erzeugen (siehe rote Kreise). Abb. angepasst nach [Ano-24].	54

Tabellenverzeichnis

5.1	Gegenüberstellung der DP mit und ohne Bild-Segmentierung. 199 Trainingsepochen, Inferenzdaten aus 1000 Lösungsepochen, Abbruch nach 30 fehlerhaften Aktionsgenerierungen (240 Roboterschritte). GPU: RTX 3080	44
5.2	Gegenüberstellung der DP mit 'guten' (3) und 'schlechten' Daten (1) ausgehend von den aufgestellten Bewertungsmetriken. 199 Trainingsepochen, 1000 Lösungsepochen, Abbruch nach 30 fehlerhaften Aktionsgenerierungen (240 Roboterschritte), $K_{ent}=10$, verwendete GPU: RTX 3080	50
5.3	Trainings-Konfiguration des gemischten Destillationsansatzes	60
5.4	Gegenüberstellung destillierten Modelle mit dem Originalmodell für die Aufgabe des Würfelschiebens. 199 Trainingsepochen, 1000 Lösungsepochen, Abbruch nach 30 fehlerhaften Aktionsgenerierungen (240 Roboterschritte), $K_{ent}=2$, verwendete GPU: RTX 3080. Dest1 wurde mit Alg. 2 mit 3 Destillationsschritten und 50 Epochen trainiert allerdings ohne die Temperatur schrittweise zu skalieren. Dest2 - config 1 wurde mit Alg. 2 trainiert mit der Konfiguration 1 aus 5.3 - analog wurde Dest2 - config 2 mit der Konfiguration 2 trainiert. Dest3 wurde in fünf Destillationsstufen à 15 Epochen durch von Alg. 1 trainiert. Die besten Werte pro Reihe sind verstärkt dargestellt.	61
5.5	Gegenüberstellung des dest3-destillierten Modells mit dem Originalmodell für die Aufgabe des zeitkritischen Würfelschiebens. 400 Trainingsepochen, 1000 Lösungsepochen, Abbruch nach 30 fehlerhaften Aktionsgenerierungen (240 Roboterschritte) oder nach 5 Sekunden, $K_{ent}=1$ bzw. 2, verwendete GPU: RTX 3080. Dest3 - 1 step wurde in sechs Destillationsstufen à 15 Epochen durch von Alg. 1 trainiert wobei der Verlustterm zu 80 % aus dem soften Teacherloss bestand; Dest3 - 2 step in fünf Destillationsstufen. Die besten Werte pro Reihe sind verstärkt dargestellt.	66

Symbolverzeichnis

Lateinische Symbole

<u>Symbol</u>	<u>Einheit</u>	<u>Bedeutung</u>
A_t^k	1	Verrauschte Aktionssequenz
A_t^{k-1}	1	Entrauschte Aktionssequenz
h_t	1	Beobachtungs-Kondition
$h_{t,img}$	1	Visuelle Embeddings
K	1	Anzahl der Rauschschrifte
K_{ent}	1	Anzahl der Entrauschungsschritte
$N_{0,\sigma^2 I}$	1	Gauß'sches Rauschen
N	1	Die Länge der Demonstrationen
O_t	1	Beobachtungsmerkmale
$p(A_t O_t)$	1	Bedingte Verteilung für das Rauschen im modifizierten DDPM
$p_\theta(x_{k-1} x_k)$	1	Bedingte Verteilung zur Rückführung des Rauschens im Standard-DDPM
$q(x_0)$	1	Ursprüngliche Daten (Originaldaten)
$q(\mathbf{x}_k \mathbf{x}_{k-1})$	1	Bedingte Verteilung von \mathbf{x}_k als verrauschter Version von \mathbf{x}_{k-1} wobei β_k die Rauschstärke angibt.
$q(\mathbf{x}_{1:K} \mathbf{x}_0)$	1	Gemeinsame Verteilung der verrauschten Zustände $\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_K$ ausgehend von \mathbf{x}_0 , wobei jeder Schritt $q(\mathbf{x}_k \mathbf{x}_{k-1})$ eine weitere Rauschstufe auf das vorherige Bild anwendet.
T_{DDPM}	1	Inferenzzeit des DDPM
T_{DPM}	1	Inferenzzeit des DPM
p_θ	1	Gelernte Rückführungsverteilung

x_0	1	Entrauschte Probe
x_k	1	Aus gauß'schem Rauschen gesampeltes Beispielbild (verrauschte Probe)
x_{k-1}	1	Beispielbild des zweiten Iterationsschrittes (aus Rauschen gesampelt)
X_t^k	1	Repräsentation aus dem tiefsten CNN-Layer
$X_{t,skip}^k$	1	Skip Connection im CNN-Layer
z	1	Explorativer Hyperparameter beim Entrauschen

Griechische Symbole

<u>Symbol</u>	<u>Einheit</u>	<u>Bedeutung</u>
α	1	Parameter des Rauschzeitplans
β_t	1	Varianz des hinzugefügten Rauschens
γ	1	Parameter des Rauschzeitplans
\hat{x}	1	Generierte synthetische Daten
ε_k	1	Künstlich zugefügte Rauschmenge zum Zeitpunkt k
ε_θ	1	Rauschvorhersagenetzwerk (oder geschätztes Rauschen)
θ	1	Parameter des Rauschvorhersagenetzwerks
η	1	DDIM Hyperparameter - kontrolliert Varianz
σ	1	Parameter des Rauschzeitplans
$\pi\beta$	1	Unbekannte Expertenpolicy

Weitere Symbole

<u>Symbol</u>	<u>Einheit</u>	<u>Bedeutung</u>
$\mathcal{L}_{DPPM,mod}$	1	Modifizierte Trainingsverlustfunktion
$\mathcal{L}_{DPPM,nrom}$	1	Normale Trainingsverlustfunktion
\mathcal{T}	1	Diffusionstrajektorie

Abkürzungsverzeichnis

Abkürzung Bedeutung

BC	Behaviour Cloning = Imitationslernen oder Verhaltensklonen
EA	Aktionsencoder
DA	Aktionsdecoder
DDIM	Denoising Diffusion Implicit Models
DDP	Diffusion Probabilistic Model
DDPM	Denoising Diffusion Probabilistic Model = Entrauschung Diffusion probabilistisches Modell
DPPO	Diffusion Policy Policy Optimization = Diffusions Policy Policy Optimierung
DP	Diffusion Policy = Diffusionsregelwerk
EE	Endeffektor eines Roboters
EDP	Efficient Diffusion Policy = Effiziente Diffusions Policy
ES	Zustandsencoder
FiLM	Feature-wise Linear Modulation
H	Horizont der Aufgabe
HDP	Hierarchical Diffusion Policy = Hierarchische Diffusions Policy
KI	Künstliche Intelligenz
KL	Kullback-Leibler
MB	oder mb = Mega Byte
MDP	Markov Decision Process = Markov'scher Entscheidungsprozess
ML	Machine Learning = maschinelles Lernen
MLP	Multilayer perceptron = Mehrlagiges Perzeptron

MSE	Mean Square Error = Mittlere quadratische Abweichung
ODE	Ordinary differential equation = gewöhnliche Differentialgleichung
PD	Progressive Distillation
RL	Reinforcement Learning
SNR	Signal Rausch Verhältnis
SSL	Self-Supervised Learning = Selbstüberwachtes Lernen
TCN	Temporal Convolutional Networks

1 Einleitung

Um die wachsende Produktnachfrage mit der Entwicklung von Industrie 4.0 zu befriedigen, müssen intelligente Fabriken die verfügbaren Ressourcen auf begrenztem Raum optimal nutzen. Dafür müssen intelligente Produktionslinien aufgebaut werden, welche flexibel und sicher sind. Sie sollten schnell auf eine veränderte Umgebung umgestellt werden können, um neue Produkte aufzunehmen und somit die Produktherstellungszeiten zu verkürzen. Ebenso existiert die Herausforderung, dynamische Prozesse mit hoher Qualität durchzuführen, was die Effizienz der Anlagen erhöht [Dju-16][Sur-20].

Da die Grenzen der Roboteranwendungen immer weiter ausgedehnt werden, steigen die Anforderungen an Industrieroboter. Herkömmliche Steuerungsmethoden reichen für komplexe praktische Anwendungen nicht mehr aus, so dass intelligente Steuerungsmethoden eingeführt werden müssen. Um dieses Problem zu lösen, wurden adaptive Steuerungsmethoden entwickelt. Hierbei kommen Sensoren z.B. Kameras zum Einsatz, um Kenntnisse über die Umgebung zu gewinnen und auf dieser Grundlage entsprechenden Handlungsentscheidungen zu treffen [Alm-16][Fou-13][Dju-16]. Verhaltensketten (BC) ist ein überwachtes Lernverfahren von Roboterhandlungsstrategien. Mit gegebenen Expertendemonstrationen wird ein Modell trainiert, was anhand eines Bildes die nächsten Roboteraktionen vorhersagen kann. Dies hat sich als sehr effektiv erwiesen, insbesondere wenn eine ausreichende Menge an Trainingsdaten zur Verfügung steht. Komplexe Aufgaben können in Sequenzen aufgeteilt werden (Sequenzmodellierung). Das Ziel besteht hier darin, die Wahrscheinlichkeitsverteilung der mehrstufigen Zustands-Aktions-Trajektorie zu modellieren. Dies ermöglicht BC, über eine einstufige Regression hinauszugehen um bessere Daten aus der Historie zu lernen. Eine vielversprechende Methode der Sequenzmodellierung soll in dieser Arbeit adaptiert und verbessert werden. Dabei handelt es sich um die Diffusion Policy [Chi-23]. Der Diffusions-Ansatz überträgt das Konzept der Diffusionsprozesse, das ursprünglich aus der Bildgenerierung stammt, auf die robotische Aktionsgenerierung. Dabei werden Handlungssequenzen durch einen kontrollierten Verrauschungs- und Entrauschungsprozess erzeugt. Die Methode eignet sich besonders für multimodale Lösungsräume, in denen mehrere valide Lösungswege existieren, wie beispielsweise beim Umfahren von Hindernissenchi2023diffusionpolicy. Der Vorteil dieser Methode ist, dass Aufgaben nur anhand von Bildern und Robotergelenkpositionen gelernt werden können. Es müssen also keine komplizierten Belohnungsfunktionen geschrieben werden. Diffusionsprozesse für die Bildgenerierung schalten schrittweise gauß'sches Rauschen auf ein Bild auf. Das entspricht dem schrittweise Vermischen des Tropfens mit dem Wasser. Anschließend wird durch einen Algorithmus gelernt, das verrauchte Bild in einen vorherigen Zeitpunkt zurückzuführen bis schließlich ein erkennbares Bild herauskommt. Nun war die Idee, das Rauschen anstatt auf Bilder, auf Roboterbewegungen anzuwenden. Anhand einer auf Wahrscheinlich-

keiten basierenden Methode werden hierdurch Handlungssequenzen durch das Sampling von Demonstrationsdaten aus einem diffusionsähnlichen Prozess erzeugt. Dies ermöglicht es, komplexe Roboter-Handlungen in einem mehrdimensionalen Raum vorherzusagen.

1.1 Herausforderungen

Diffusionsmodelle erben jedoch Limitierungen aus dem Verhaltensklonen wie suboptimale Leistungen bei unzureichenden Demonstrationsdaten. Um eine Aufgabe mit ausreichender Sicherheit imitieren zu können, müssen abhängig von der Komplexität viele erfolgreiche Demonstrationbahnen aufgenommen werden. Das Aufnehmen von robotergerechten Daten ist mit hohem Aufwand verbunden.

Zweitens hat die Diffusionspolitik höhere Rechenkosten und Inferenzzeiten im Vergleich zu einfacheren Methoden. Da die Diffusion Policy auf dem Sampling-Prozess basiert, wäre eine Optimierung der Sampling-Effizienz entscheidend. Diffusionsmodelle haben oft das Problem, dass sie viele Iterationen benötigen, um zu glatten Lösungen zu konvergieren, was für zeitkritische Steuerungsaufgaben problematisch sein kann. Es können Fortschritte bei der Beschleunigung von Diffusionsmodellen genutzt werden, um die Anzahl der erforderlichen Inferenzschritte zu reduzieren, wie z. B. verbesserte Rauschzeitpläne, Inferenzlöser- oder Konsistenz-Ansätze.

Auch im Bezug Skalierbarkeit lässt sich die DP verbessern, um anhand von gelernten Manipulationsaufgaben an bspw. kleinen Objekten skalierte Handlungssequenzen für die Interaktion mit größeren Objekten zu generieren.

1.2 Lösungsansatz

Die vorliegende Arbeit beschäftigt sich zunächst mit der Einordnung und Erklärung der vorgestellten Diffusion Policy (DP) (2). Anschließend werden bestehende Verbesserungsansätze vorgestellt (3). Drauf aufbauend sollen Lösungsansätze für zuvor genannten Herausforderungen gefunden werden (5).

2 Theorie

Diffusionsmodelle wurden bereits kurz vorgestellt. Sie nehmen iterative Gradientenabstiege anhand der Demonstrationsdaten vor. Dadurch erben sie Eigenschaften aus dem Imitationslernen, obwohl sie zur Gruppe von generativen Modellen gehören. Weiterhin bieten sie viele Vorteile in Situationen wo andere Maschine-Learning-Ansätze an ihre Grenzen stoßen. Um das Verständnis für die größten Herausforderungen beim maschinellen Lernen zu stärken, wird im Folgenden zunächst Imitationslernen vom Reinforcement Lernen abgegrenzt. Der Diffusionsprozess und zusammenhängende Verfahren müssen zunächst erläutert werden, bevor anschließend der Zusammenhang zur Diffusion Policy verstanden werden kann. Wie man ein Diffusionsmodell trainiert und sammelt wird erklärt. Zusätzlich zur Erklärung der DP-Grundlagen stellt dieses Kapitel auch Verbesserungsansätze in Inferenzlösern und Destillationsverfahren vor. Damit wird das notwendige Wissen für den Stand der Technik aufgebaut.

2.1 KI und Maschinelles Lernen

Künstliche Intelligenz bezeichnet das intelligente Verhalten von Maschinen. Sie umfasst die Forschungsrichtung des Maschinellen Lernens. Dieses umfasst eine Vielzahl Art und Weisen, Maschinen das Lösen von Aufgaben basierend auf Daten beizubringen. Ein bekanntes Beispiel sind generative Modelle, welche anhand von Eingabe-Text Ausgabe-Artefakte wie Code oder Bilder generieren. Die Aufgaben werden von sogenannten Agenten durchgeführt.

2.2 Abgrenzung des Imitationslernen zum Reinforcement Learning

Zwei prominente Ansätze zur Lösung von Steuerungs- und Entscheidungsfindungsproblemen sind *Reinforcement Learning* (RL) und *Imitationslernen* (IL). Beide Methoden zielen darauf ab, optimale Handlungen in komplexen Umgebungen zu lernen, unterscheiden sich jedoch grundlegend in der Art und Weise, wie das Lernen erfolgt. Da die später vorgestellte Diffusion Policy auf dem Imitationslernen beruht, soll dies im folgenden Abschnitt gegenüber dem Reinforcement Learning abgegrenzt werden.

2.3 Imitationslernen

Beim *Imitationslernen* (IL) ahmt der Agent das Verhalten eines Experten nach, anstatt durch eigene Interaktionen mit der Umgebung zu lernen. Der Agent erhält eine Menge von Bei-

spielsdaten, die von einem menschlichen oder algorithmischen Experten generiert wurden, und lernt, eine Policy zu approximieren, die den gezeigten Handlungen möglichst ähnlich ist. Policy meint dabei eine Handlungsrichtlinie nach welcher der Agent agiert.

Das Ziel von IL ist es, eine solche Policy $\pi(a|s)$ zu lernen, welche die vom Experten demonstrierten Handlungen a in denselben Zuständen s nachahmt [Pom-91]. Dies kann mit Methoden des überwachten Lernens erreicht werden, wobei der Agent eine *Mapping*-Funktion von Zuständen zu Handlungen lernt, basierend auf den Demonstrationen des Experten. Eine häufig verwendete Technik im Imitationslernen ist das *Behavioral Cloning*, bei dem die gezeigten Aktionen direkt auf den Eingaberaum des Agenten abgebildet werden [Pom-91]. Imitationslernen ist also das Klonen von Verhaltensweisen.

Anhand von Abb. 2.2 wird das IL wie folgt erklärt: Es gibt eine unbekannte Expertenpolitik, die hier als π_β bezeichnet wird. Diese ist beispielsweise das explizite oder implizite Wissen eines Experten. Dieser agiert durch das Wissen mit der Welt. Die Expertenpolicy trifft Handlungs-Entscheidungen in Form von Aktionen a abhängig von Belohnung r und Zustand s . Anschließend wird die Beziehung zwischen Zuständen, Belohnungen und Aktionen verwendet, um eine Policy zu trainieren, welche die Aktionen des Experten aus Zuständen vorhersagt. Diese wiederhergestellte Policy ist als rote Linie in Abb. 2.2 dargestellt. Dieser Lernprozess wird auch als offline RL bezeichnet; Offline, weil der Roboter beim Lernen nicht aktiv verwendet wird.

2.3.1 Vorteile des Imitationslernens

IL kann schneller und effizienter als RL sein, da der Agent nicht durch einen langwierigen Trial-and-Error-Prozess gehen muss. Anstatt zu experimentieren, lernt er direkt von erfolgreichen Handlungen des Experten. Dies ist besonders vorteilhaft in Situationen, in denen das Explorieren gefährlich oder teuer ist, wie z.B. in der Robotik oder autonomen Fahrzeugsteuerung [Arg-09]. Es ist außerdem recht stabil und leicht zu verifizieren. Ein weiterer Vorteil, ist dass es gut mit großen Datensätzen skaliert.

2.3.2 Nachteile des Imitationslernens

Der erste und offensichtlichste Nachteil ist, dass man nur imitieren kann, was auch als Expertendatensatz demonstriert werden kann. Weiterhin hat IL eine Abhängigkeit von hochwertigen und umfangreichen Demonstrationsdaten. Wenn die Daten unvollständig oder fehlerhaft sind, kann das zu schlechten Policies führen. Darüber hinaus hat IL Schwierigkeiten, in Umgebungen mit hoher Variabilität oder in Situationen, die nicht explizit in den Demonstrationen enthalten sind, zu generalisieren [Ros-11]. Ein weiterer sehr großer Nachteil ist die Fehlerakkumulation über die Zeit. Dieser Nachteil ist sehr wichtig zu verstehen. Deshalb wird er im

folgenden Kapitel weiter ausgeführt.

2.3.2.1 Nachteil der Fehlerakkumulation über die Zeit

Das Hauptproblem liegt darin, dass kleine Fehler im Verhalten der Policy zu einer Abweichung von der Trainingsbahn führen, was zu immer größeren Fehlern führt wie in Abb. 2.1 dargestellt. Das geschieht, weil die Policy sich in Zustände begibt, die sie während des Trainings nicht gesehen hat. Diese Fehler addieren sich und können zu einem deutlichen Leistungsverlust führen.

In einem kontinuierlichen Szenario, in dem bei jeder Abweichung eine Kostenstrafe verhängt wird, skaliert der Fehler quadratisch mit der Länge der Trajektorie h^2 . In einem Szenario, bei dem nur eine Strafe für das Verlassen des optimalen Pfads verhängt wird, skaliert der Fehler linear mit der Trajektorienlänge h . Die Analyse zeigt, dass Imitationslernen in bestimmten Aufgaben (z.B. Erfolg- oder Misserfolg-Aufgaben) nicht optimal ist, da der Fehler mit der Trajektorienlänge skaliert. [Pom-91]

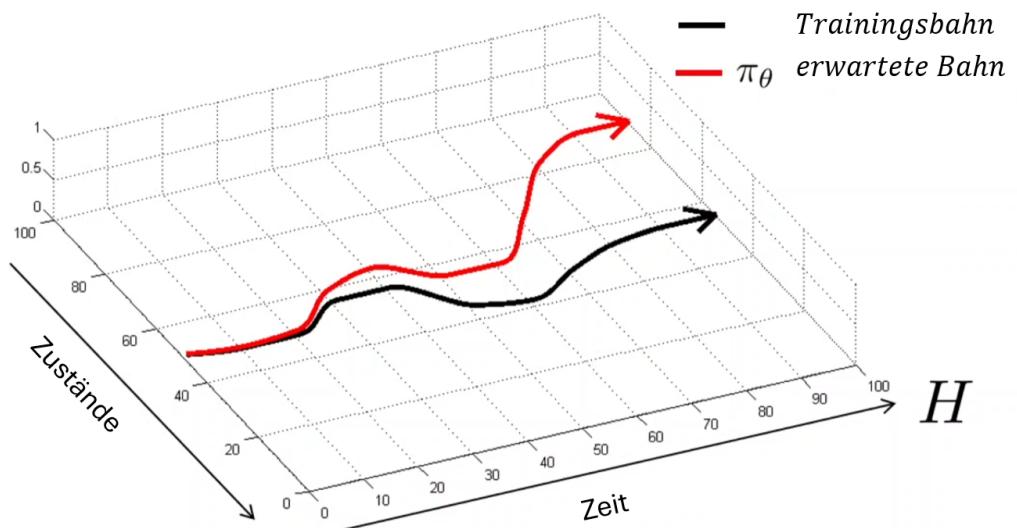


Abbildung 2.1: Fortpflanzung der Trajektorien-Fehler über die Zeit. H: der Horizont der Aufgabe (abgewandelt nach [RAI-22])

2.3.3 Identifiziertes Verbesserungspotential

Die Variabilität der Umgebung kann im Trainingsdatensatz abgebildet werden, um beim Generalisieren zu helfen. Eine große Schwäche ist die Abhängigkeit von guten Demonstrationsdaten. Demnach sollte der Trainingsdatensatz gefiltert werden, sodass nur von den 'guten' Daten gelernt wird.

2.4 Reinforcement Learning

Auf der anderen Seite steht *Reinforcement Learning* (RL). Das ist etwas sehr Gegensätzliches. Dies bezeichnet eine Lernmethode, bei der ein Agent durch Interaktion mit der Umgebung lernt, eine Sequenz von Handlungen auszuführen, um eine kumulative Belohnung zu maximieren. Der Agent handelt basierend auf dem Zustand der Umgebung und erhält nach jeder Aktion eine Belohnung (*reward*). Das Ziel ist es, eine *Policy* $\pi(a|s)$ zu erlernen, die vorgibt, welche Aktion a in einem Zustand s ausgeführt werden soll, um den erwarteten kumulierten Belohnungswert *Return* zu maximieren [Sut-18]. Der Agent lernt durch wiederholte Versuche, welche Aktionen langfristig den größten Erfolg bringen. Das zentrale Problem im RL ist die Balance zwischen *Exploration* (das Erkunden neuer Aktionen) und *Exploitation* (das Ausnutzen bereits gelernter, guter Aktionen). Mathematisch lässt sich der Lernprozess als Markov Decision Process (MDP) modellieren, wobei die optimale Policy durch die Maximierung des erwarteten diskontierten Returns erlernt wird. Bildlich gesprochen kann man sich RL wie Topfschlägen vorstellen. Die Person ist hierbei der Agent und die wärmer- bzw. leiser-Rufe sind die positiven/negativen Belohnungen. Der einzige Unterschied ist, dass der Agent nach dem erfolgreichen finden des Topfes genau weiß, anhand welcher Rufe er erneut zu dem Topf finden kann. In dem Vergleich wären beispielsweise Parameter wie die Bodenbeschaffenheit Teil des Zustands s . Das sind also augmentierende Hinweise. Nur im Zusammenspiel mit der Belohnung führen sie zu einer Handlungsmaxime.

2.4.1 Vorteile des Reinforcement Learning

RL ist besonders leistungsfähig in Umgebungen, in denen das Ziel oder der Weg dahin nicht im Voraus bekannt ist, und in Situationen, in denen der Agent durch kontinuierliche Interaktion mit der Umgebung lernt. Ein gutes Beispiel ist das Balancieren eines Objekts. Der Weg dort hin ist zwar intuitiv bekannt und lässt sich mit komplexen Handlungsfunktionen beschreiben. Es ist jedoch sehr viel einfacher einen RL-Agenten hierfür zu trainieren. RL ist robust gegenüber dynamischen Umgebungen und kann auch in hochdimensionalen Zustands- und Aktionsräumen arbeiten [Mni-15].

2.4.2 Vorteile von RL bei Aufgaben mit kritischen Verhaltensschritten

"Reinforcement Learning" hat Vorteile im Vergleich zu "Imitation Learning" (IL), insbesondere in einem Offline-Setting. Offline-RL kann potenziell bessere Ergebnisse liefern, weil es die Dynamik von Aktionen berücksichtigt und die Konsequenzen des Handelns versteht. Nicht alle Aktionen sind gleich wichtig, und in bestimmten Situationen ist es entscheidend, die

richtigen Handlungen zu wählen (z. B. beim Balancieren auf einem Drahtseil). Imitation Learning behandelt alle Aktionen gleich, was zu Problemen führen kann.

RL hingegen erkennt, dass manche Handlungen wichtiger sind als andere, besonders in kritischen Zuständen, wo eine falsche Entscheidung schwerwiegende Folgen haben kann. Die Gewichtung der Wegpunkte stammt aus der gelernten Belohnung/Bestrafung. In diesen kritischen Zuständen muss eine bestimmte Handlung ausgeführt werden, um Erfolg zu haben, während in weniger kritischen Zuständen die Auswahl der Aktion weniger entscheidend ist. Ein wichtiger Punkt ist, dass Offline-RL sogar von suboptimalen Daten lernen kann, indem es versteht, was nicht zu tun ist, während Imitation Learning durch suboptimale Daten beeinträchtigt wird. Offline-RL kann also in speziellen Szenarien, in denen es kritische Zustände gibt, besser abschneiden kann als Imitation Learning.

2.4.3 Nachteile des Reinforcement Learning

Ein bedeutender Nachteil des RL ist der hohe Rechenaufwand und die oft langen Lernphasen. Da der Agent viele Interaktionen mit der Umgebung benötigt, um optimale Policies zu lernen, kann der Prozess ineffizient und zeitaufwendig sein. Insbesondere in Umgebungen, in denen Belohnungen selten oder schwer zu erreichen sind kann der Zeitaufwand enorm sein [Aru-17].

2.5 Unterschied

Ein Unterschied zwischen den beiden Darstellungen ist die Annahme über die Herkunft der gesammelten Daten. Beim Imitationslernen gehen wir davon aus, dass die Daten von einem Experten gesammelt wurden, d.h. von einer Strategie, die entweder optimal oder nahezu optimal für die wahre Belohnungsfunktion der Aufgabe ist. Auf der anderen Seite treffen wir beim RL nicht diese Annahme, sondern die Trainingsdaten können sehr unterschiedlich und auch suboptimal sein.

Ein weiterer Unterschied zwischen Imitationslernen und Offline Reinforcement Learning (RL) liegt in ihren Zielen:

Beim Imitationslernen geht man davon aus, dass die Verhaltenspolicy optimal ist. Ziel ist es, die Daten generierende Policy wiederherzustellen.

Bei Offline RL ist das Ziel etwas anders. Hier versucht man, Ordnung aus einem chaotischen Datensatz von Trajektorien zu schaffen. Das Ziel ist, eine ideale Policy (rot) zu finden, die besser bei der Aufgabe abschneidet als die einzelnen schwarz markierten Beispieltrajektorien. Idealerweise ist die gelernte Policy also besser als jede einzelne der beobachteten Trajektorien. Reinforcement Learning und Imitationslernen unterscheiden sich grundlegend in ihren Ansätzen:

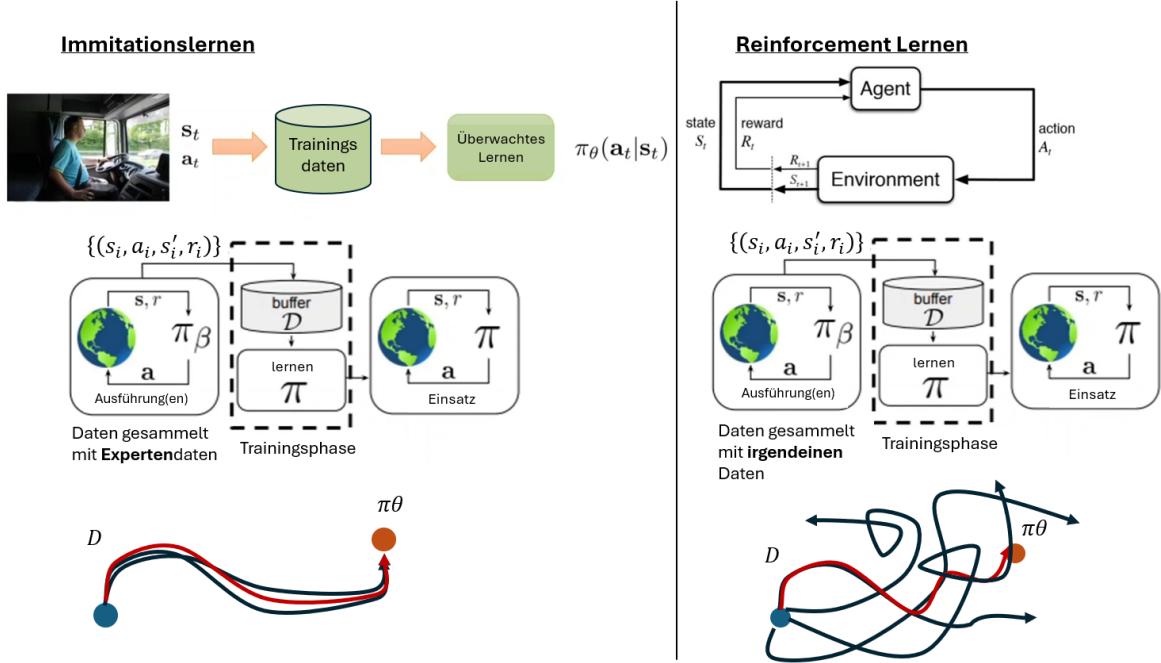


Abbildung 2.2: Gegenüberstellung von Imitationslernen und Reinforcement Lernen (abgewandelt nach [RAI-22])

- **Lernquelle:** Im RL lernt der Agent durch Interaktion mit der Umgebung, während im IL das Lernen von vorgegebenen Demonstrationen des Experten erfolgt.
- **Lernziel:** Im RL ist das Ziel, eine Policy zu finden, die die kumulative Belohnung maximiert, während im IL das Ziel darin besteht, die vom Experten gezeigten Handlungen so genau wie möglich zu imitieren.
- **Effizienz:** IL ist oft schneller, da es keinen langen Trial-and-Error-Prozess erfordert. RL kann jedoch flexibler sein, da der Agent in RL-Szenarien auch in Umgebungen lernen kann, in denen keine Expertendaten verfügbar sind.
- **Exploration vs. Exploitation:** RL balanciert aktiv zwischen Exploration und Exploitation, wohingegen IL keine Exploration erfordert und nur das gelernte Verhalten des Experten nachahmt.

2.6 Gemeinsamkeiten

Was ein Offline RL Algorithmus erfüllen muss:

- **Nähe zu den vorhandenen Daten:** Der Algorithmus sollte sich nicht zu weit von den beobachteten Daten entfernen, da er keine zusätzlichen Daten sammelt. Es wäre

somit riskant, Verhaltensweisen zu lernen, die zu stark von den beobachteten Aktionen abweichen. Das Ziel ist, nur solche Aktionen zu wählen, bei denen man sicher ist, dass man ihren Wert genau abschätzen kann.

- **Maximierung der Belohnung:** Wie bei allen RL-Methoden muss auch im Offline-RL die Belohnung maximiert werden, was bedeutet, dass die erwartete Summe der diskontierten Belohnungen möglichst groß sein sollte.

Es gibt jedoch eine Spannung zwischen diesen beiden Zielen: Je mehr man versucht, die Belohnung zu maximieren, desto eher neigt man dazu, sich von den Daten zu entfernen. Effektive Offline-RL-Algorithmen müssen daher einen intelligenten Kompromiss zwischen der Maximierung der Belohnung und der Nähe zu den Daten finden.

Imitationslernen erfüllt nur die Nähe zu den Daten, aber nicht unbedingt die Belohnungsmaximierung. In manchen Fällen könnte es sogar sinnvoller sein, Imitationslernen statt Offline-RL zu verwenden, insbesondere wenn das Datenset gut ist und das Verhalten der Policy bereits nahe an optimal ist. Wenn jedoch die Verhaltenspolicy schlecht war, könnte man die Belohnung nur maximieren, indem man sich zu stark von den Daten entfernt, was auch riskant sein kann.

2.7 Zusammenfassung der Gegenüberstellung

Reinforcement Learning und Imitationslernen sind zwei unterschiedliche Ansätze zur Lösung von Entscheidungsfindungsproblemen. Während RL durch selbstständige Exploration der Umgebung langfristig bessere Policies lernen kann, ist IL eine effizientere Methode, die jedoch stark von der Qualität der Expertendaten abhängt. Insgesamt muss abgewogen werden, ob es sinnvoll ist, mehr auf Imitationslernen oder auf Offline-RL zu setzen. Es hängt maßgeblich von der Qualität des verfügbaren Datensatzes und der Art der Aufgabe ab. Beide Methoden haben ihre Vor- und Nachteile und werden oft kombiniert, um voneinander zu profitieren [Ng-00].

2.8 Diffusionsprozess im Zusammenhang mit der Diffusion Policy

Die Diffusion Policy basiert auf dem Prinzip der Diffusion, einem iterativen Prozess, bei dem im Vorwärtsschritt Rauschen hinzugefügt und im Rückwärtsschritt entfernt wird. Dieser Ansatz findet häufig in generativen Modellen Anwendung. Ein prominentes Beispiel ist Stable Diffusion, ein Diffusionsmodell für die Bildgenerierung. Dabei wird schrittweise aus reinem Rauschen ein bedeutungsvolles Bild erzeugt, indem der Prozess des „Entrauschens“ erlernt wird.

Im Gegensatz zur Bildgenerierung wird die Diffusion Policy im Bereich der Robotik und Bewegungssteuerung eingesetzt. Sie nutzt den Diffusionsprozess, um Bewegungssequenzen vorherzusagen, indem abstrakte Eingaben in konkrete Bewegungsabfolgen umgewandelt werden. Dabei kommen probabilistische Modelle zum Einsatz. Die Diffusion Policy kann als Analogie zur Bildgenerierung betrachtet werden, wobei anstelle von Pixeln Bewegungsdaten erzeugt werden. Während Stable Diffusion bildbeschreibende Labels nutzt, verarbeitet die Diffusion Policy aktuelle Sensordaten, z. B. Bilder der Robotersituation, als Eingabe. Abbildung 2.3 stellt die Entrauschungsschritte beider Methoden gegenüber. Der von links nach rechts dargestellte Prozess illustriert die schrittweise Generierung von Bewegungssequenzen mithilfe von Denoising Diffusion Probabilistic Models (DDPM), deren Funktionsweise im Folgenden erläutert wird.

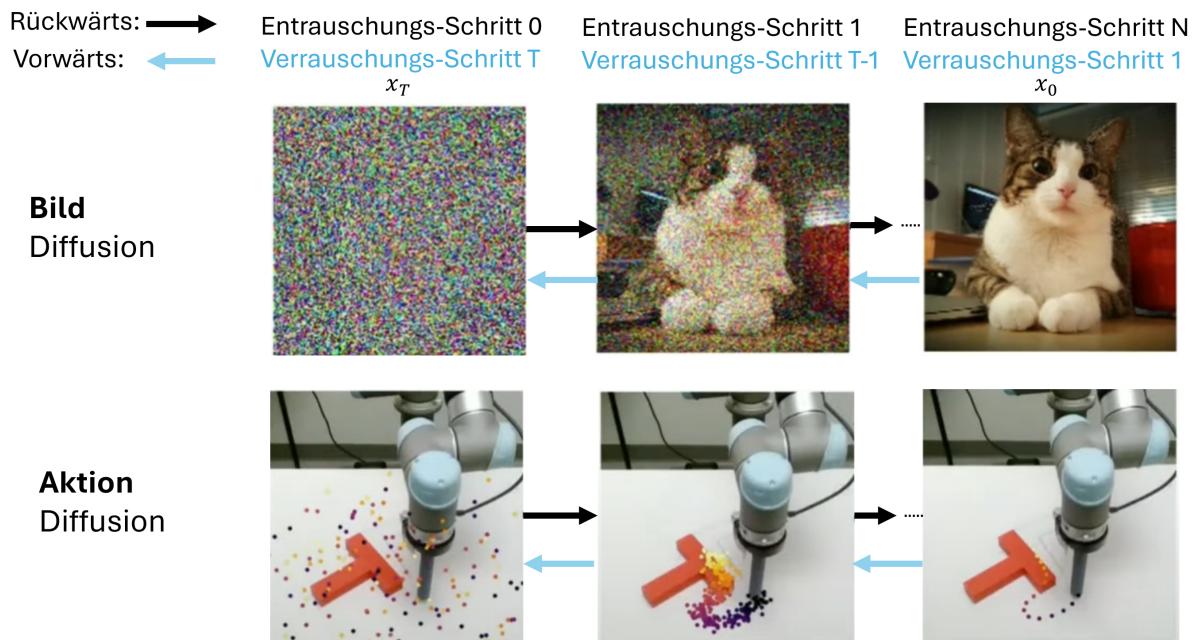


Abbildung 2.3: Gegenüberstellung des Entrauschungsprozess bei der Bild-Diffusion und der Aktion-Diffusion. Abgewandelt nach [Chi-23]

2.9 Denoising Diffusion Probabilistic Models (DDPM)

Denoising Diffusion Probabilistic Model (DDPM) ist ein generatives Modell zum ver- und entrauschen von Daten. Das Modell lässt sich in den Vorwärts- und Rückwärts-Prozess aufteilen.

Vorwärtsprozess

Im Vorwärtsprozess (Abb. 2.3 von rechts nach links) werden Originaldaten, beispielsweise Bilder, schrittweise durch das Hinzufügen von Gauß'schem Rauschen zerstört. Gauß'sches Rauschen wird verwendet, um in jedem Schritt möglichst viele Daten gleichmäßig zu beschädigen. Das macht dann wiederum die vorhergesagten Werte so unabhängig voneinander, wie möglich. Ausgehend von einer sauberen Probe x_0 aus der Datenverteilung $q(x)$ wird zu jedem Zeitpunkt t schrittweise Rauschen mit Varianz β_t hinzugefügt. Ein einzelner Schritt ist wie folgt definiert:

$$\mathbf{x}_t = \sqrt{1 - \beta_t} \mathbf{x}_{t-1} + \sqrt{\beta_t} \boldsymbol{\epsilon}_{t-1} \quad (2.1)$$

wobei $\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$. Dies beschreibt eine Normalverteilung mit Mittelwert 0 und Identitätsvarianz. Je mehr Schritte durchgeführt werden, desto stärker wird die ursprüngliche Information durch Rauschen überlagert. Die Übergangswahrscheinlichkeit eines Markov-Prozesses ist gegeben durch:

$$q(\mathbf{x}_t | \mathbf{x}_{t-1}) = \mathcal{N}(\mathbf{x}_t; \sqrt{1 - \beta_t} \mathbf{x}_{t-1}, \beta_t \mathbf{I}) \quad (2.2)$$

$q(\mathbf{x}_t | \mathbf{x}_{t-1})$ = Bedingte Verteilung von \mathbf{x}_t als verrauschter Version von \mathbf{x}_{t-1} wobei β_t die Rauschstärke angibt.

Die gesamte Zustandsmenge $\mathcal{T} = \{\mathbf{x}_0, \mathbf{x}_1, \dots, \mathbf{x}_T\}$ wird als Diffusionstrajektorie bezeichnet. Aufgrund der Markov-Eigenschaft hängt der Zustand \mathbf{x}_t nur vom unmittelbar vorherigen Zustand ab:

$$q(\mathbf{x}_{1:T} | \mathbf{x}_0) = \prod_{k=1}^T q(\mathbf{x}_t | \mathbf{x}_{t-1}) \quad (2.3)$$

$q(\mathbf{x}_{1:T} | \mathbf{x}_0)$ = Gemeinsame Verteilung der verrauschten Zustände $\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_T$ ausgehend von \mathbf{x}_0 , wobei jeder Schritt $q(\mathbf{x}_t | \mathbf{x}_{T-1})$ eine weitere Rauschstufe auf das vorherige Bild anwendet.

Zusätzlich zum Hinzufügen von Rauschen wird die ursprüngliche Information sukzessive reduziert, sodass letztlich reines Rauschen entsteht. Dieser Zustand dient als Ausgangspunkt für den Rückwärtsprozess [Wen-21] [Pie-23].

Reparametrisierungstrick für x_t zu einem beliebigen Zeitpunkt t

Der Reparametrisierungstrick ermöglicht es, x_t direkt zu berechnen, ohne alle t Schritte durchlaufen zu müssen. Mit $\alpha_t = 1 - \beta_t$ gilt:

$$x_t = \sqrt{\alpha_t} x_{t-1} + \sqrt{1 - \alpha_t} \epsilon_{t-1}. \quad (2.4)$$

Durch rekursive Anwendung kann \mathbf{x}_t direkt als Funktion von \mathbf{x}_0 dargestellt werden:

$$\mathbf{x}_t = \sqrt{\bar{\alpha}_t} \mathbf{x}_0 + \sqrt{1 - \bar{\alpha}_t} \boldsymbol{\epsilon}_0, \quad (2.5)$$

wobei $\bar{\alpha}_t = \prod_{i=1}^t \alpha_i$ das Produkt der α_i -Werte über alle vorherigen Schritte darstellt. Das Signal-Rausch-Verhältnis $\bar{\alpha}_t$ gibt an, wie viel vom ursprünglichen Signal in \mathbf{x}_t noch enthalten ist: Zu Beginn ($t = 0$) entspricht der Zustand \mathbf{x}_0 vollständig dem ursprünglichen Datensatz, und $\bar{\alpha}_0 = 1$. Mit fortschreitender Zeit wird zunehmend Rauschen hinzugefügt, und der ursprüngliche Datensatz wird verwässert. Am Ende der Trajektorie ($t = T$) gilt $\bar{\alpha}_T = 0$, und der Zustand \mathbf{x}_T besteht nur noch aus reinem Rauschen.

Rückwärtsprozess

Der Rückwärtsprozess ist der generative Teil des Modells, bei dem aus verrauschten Daten (x_T) schrittweise eine rauschfreie Probe (x_0) rekonstruiert wird. Der Prozess basiert auf der Annäherung der Verteilung $q(\mathbf{x}_{t-1} | \mathbf{x}_t)$ durch ein trainiertes Modell p_θ , da die direkte Berechnung von q nicht möglich ist. [Ho-20] haben festgestellt, dass bekannte Ausgangsgröße x_0 es möglich macht, eine Rückwärts-Verteilung zu erhalten. Die bedingte Verteilung für einen Rückwärtsschritt ist:

$$p_\theta(\mathbf{x}_{t-1} | \mathbf{x}_t) = \mathcal{N}\left(\mathbf{x}_{t-1}; \tilde{\mu}_\theta(\mathbf{x}_t, t), \tilde{\beta}_t \mathbf{I}\right), \quad (2.6)$$

wobei $\tilde{\mu}_\theta$ und $\tilde{\beta}_t$ vom Modell gelernt werden. Das Modell p_θ entfernt schrittweise das Rauschen aus x_T , bis eine synthetische Ausgabe x_0 entsteht, die den Originaldaten $q(x_0)$ ähnelt. Wir definieren diesen Rückwärtsprozess als:

$$p_\theta(\mathbf{x}_{t-1} | \mathbf{x}_t) = \mathcal{N}\left(\mathbf{x}_{t-1}; \tilde{\mu}_\theta(\mathbf{x}_t, t), \tilde{\beta}_t \mathbf{I}\right) \quad (2.7)$$

Ein Rückwärtsschritt wird durch folgende Gleichung berechnet:

$$x_{t-1} = \sqrt{\frac{1}{\alpha_t}} \left(x_t - \sqrt{\frac{\beta_t}{1 - \bar{\alpha}_t}} \epsilon_\theta(x_t, t) \right) + \sigma_t z, \quad (2.8)$$

wobei $z \sim \mathcal{N}(0, I)$ [Ho-20] und ϵ_θ ein trainiertes Rauschvorhersagenetzwerk ist. Nach K_{ent} Iterationen entsteht die rauschfreie Ausgabe x_0 .

In [Chi-23] wird dies auch beschrieben als:

$$x_{t-1} = \alpha(x_t - \gamma \epsilon_\theta(x_t, t) + N_{0, \sigma^2 I}), \quad (2.9)$$

[Wen-21].

Training

Das Training eines Diffusionsmodells basiert auf der Minimierung des Unterschieds zwischen der tatsächlichen und der vorhergesagten Rückwärtsdiffusion. Die Evidence Lower Bound (ELBO), eine untere Schranke für die Log-Wahrscheinlichkeit der Daten, wird wie folgt dargestellt:

$$\log p(\mathbf{x}) \leq L_T + \sum_{t=2}^T L_{t-1} - L_0, \quad (2.10)$$

wobei:

- L_T : KL-Divergenz (Kullback-Leibler-Divergenz, ein Maß für den Unterschied zwischen zwei Verteilungen) zwischen der Prior-Verteilung $p(\mathbf{x}_T)$ und der posterioren Verteilung $q(\mathbf{x}_T | \mathbf{x}_0)$,
- $\sum_{t=2}^T L_{t-1}$: Differenz zwischen gewünschtem und vorhergesagtem Entrauschungsschritt,
- L_0 : Rekonstruktionsverlust zur Überprüfung der Genauigkeit von x_0 .

[Ho-20] vereinfachten die Verlustfunktion, indem sie Gewichtungstermen vernachlässigten. Die vereinfachte Funktion lautet:

$$L_{\text{simple},t} = \mathbb{E}_{t,\mathbf{x}_0,\epsilon} [\|\boldsymbol{\epsilon}_t - \boldsymbol{\epsilon}_\theta(\mathbf{x}_t, t)\|^2], \quad (2.11)$$

wobei t gleichmäßig aus $[1, T]$ gezogen wird. Hier wird der mittlere quadratische Fehler (Mean Squared Error, MSE) zwischen dem echten Rauschen $\boldsymbol{\epsilon}_t$ und der Vorhersage des Modells $\boldsymbol{\epsilon}_\theta$ minimiert.

Der Trainingsablauf ist wie folgt: 1. Bilder x_0 aus dem Datensatz werden mit Rauschen $\boldsymbol{\epsilon}_t$ versehen. 2. Das Rauschvorhersagenetzwerk $\boldsymbol{\epsilon}_\theta$ versucht, das aufgebrachte Rauschen zu schätzen. 3. Der Trainingsverlust wird berechnet als:

$$\mathcal{L}_{\text{DDPM}} = \text{MSE}(\boldsymbol{\epsilon}_t, \boldsymbol{\epsilon}_\theta(x_t, t)). \quad (2.12)$$

Das Ziel ist, dass das Netzwerk lernt, das Rauschen präzise zu entfernen und so synthetische Daten zu erzeugen, die den Originaldaten ähneln. Dieser Prozess wird iterativ mit Gradient Descent (einer Optimierungsmethode zur Anpassung der Modellparameter) verbessert. Eine schematische Darstellung findet sich in Abb. 2.4 [Wen-21].

Algorithmus 1 Training

```

1: repeat
2:  $x_0 \sim q(x_0)$  (Bild ohne Rauschen)
3:  $t \sim Uniform(\{1, \dots, T\})$  (Zeitschritt)
4:  $\epsilon \sim N(0, I)$  (Rauschen aus der Normalverteilung)
5: nehme Gradienten-Abstieg auf
   
$$\nabla_{\theta} \left\| \epsilon - \underbrace{\epsilon_{\theta}(\sqrt{\bar{\alpha}_t}x_0 + \sqrt{1 - \bar{\alpha}_t}\epsilon, t)}_{x_t \text{ aus dem Vorwärtsprozess}} \right\|^2$$

6: until konvergiert

```

Abbildung 2.4: Trainings-Algorithmus des DDPM, abgewandelt nach [Ho-20]

Brücke zu Offline-RL Methoden

Offline-RL Methoden haben das Ziel, nah an den Demonstrationsdaten zu sein und die Belohnung zu maximieren. Die Diffusion Policy verfolgt das erste Ziel. Allerdings wird sie belohnt, wenn ihre vorhergesagtes Rauschen möglichst stark dem Originalrauschen entspricht. D.h. ihre Motivation ist eigentlich nur die Nähe zu den Demonstrationsdaten. Da sie jedoch Schritte durch ein diverses Gradientenfeld nimmt, können die Lösungswege variieren. Sie sind jedoch immer nah an den Demonstrationsdaten!

Sampling (Inferenz)

Nach dem Training des Rauschvorhersagenetzwerks kann dieses zur Bildgenerierung genutzt werden. Diesen Lösungsprozess nennt man auch 'Inferenz'. Der grundlegende Sampling Algorithmus aus [Ho-20] ist in Abb. 2.5 dargestellt. Er beginnt mit einer zufälligen verrauschten Probe x_T , die aus einer Normalverteilung $\mathcal{N}(0, I)$ entnommen wird. Anschließend werden K_{ent} schrittweise Entrauschungen durchgeführt, wobei der Parameter z ebenfalls aus der Normalverteilung zur Kontrolle des Entrauschungsgrades hinzugefügt wird. Der Parameter z bringt zusätzliche Varianz ein und ermöglicht explorativere Ergebnisse. Für $z = 0$ wird der Sampling-Prozess deterministisch.

Algorithmus 2 Sampeln

```

1:  $x_T \sim N(0, I)$  (Rauschen aus der Normalverteilung)
2: for  $t = T, \dots, 1$  do (Bild ohne Rauschen)
3:    $z \sim N(0, I)$  if  $t > 1$ , else  $z = 0$  ( $z$  aus Normalverteilung)
4:    $x_{t-1} = \frac{1}{\sqrt{\alpha_t}} \left( x_T - \frac{1-\alpha_t}{\sqrt{1-\alpha_t}} \underbrace{\epsilon_\theta(x_t, t)}_{\text{Entrauschen aus dem Rückwärtsprozess}} \right) + \sigma_t z$  (Entrauschen aus dem Rückwärtsprozess)
5: end for Vorhergesagtes Rauschen aus
6: return  $x_0$  dem neuronalen Netzwerk

```

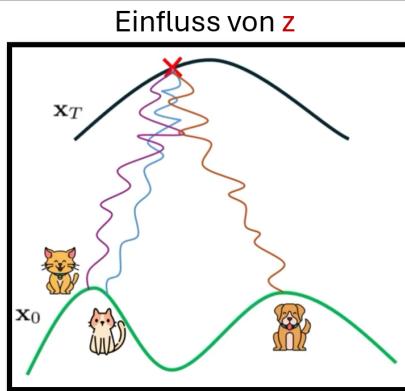


Abbildung 2.5: Sample-Algorithmus des DDPM, abgewandelt nach [Ho-20]

Die grundlegende Methode hierfür ist das diskrete zeitliche *ancestral sampling*, bei dem der Rückwärtsprozess durch die Gleichung:

$$x_{t-1} = \sqrt{\frac{1}{\alpha_t}} \left(x_t - \sqrt{\frac{\beta_t}{1-\bar{\alpha}_t}} \epsilon_\theta(x_t, t) \right) + \sigma_t z, \quad (2.13)$$

definiert wird. Hierbei ist $z \sim \mathcal{N}(0, I)$ zufälliges Rauschen, das für den explorativen Teil des Samplings verantwortlich ist.

Alternativ kann der Entrauschungsprozess deterministisch erfolgen, indem die *Probability Flow ODE* (gewöhnliche Differentialgleichung zur probabilistischen Flusssteuerung) numerisch gelöst wird [Son-21]. Diese ist gegeben durch:

$$\frac{dz_t}{dt} = f(z_t, t) - \frac{1}{2} g^2(t) \nabla_z \log \hat{p}_\theta(z_t), \quad (2.14)$$

wobei:

$$\nabla_z \log \hat{p}_\theta(z_t) = \frac{\alpha_t \hat{x}_\theta(z_t) - z_t}{\sigma_t^2}, \quad f(z_t, t) = \frac{d \log \alpha_t}{dt} z_t, \quad g^2(t) = \frac{d \sigma_t^2}{dt} - 2 \frac{d \log \alpha_t}{dt} \sigma_t^2.$$

Da $\hat{x}_\theta(z_t)$ durch ein neuronales Netzwerk parametrisiert ist, stellt diese Gleichung eine spe-

zielle Form einer neuronalen ODE (gewöhnliche Differentialgleichung, deren Dynamik durch ein neuronales Netzwerk gelernt wird) dar. Numerische Lösungsverfahren wie die Euler- oder Runge-Kutta-Methode können angewandt werden. Der DDIM-Sampler (Deterministic Diffusion Implicit Sampler) kann als Integrationsansatz dieser Gleichung verstanden werden. Details hierzu folgen in späteren Kapiteln.

Konditionierte Diffusionsmodelle

Ein leistungsfähiger Anwendungsfall von Diffusionsmodellen ist ihre Verwendung für die bedingte Generierung. In diesem Abschnitt werden wir uns ansehen, wie Diffusionsmodelle für die bedingte Generierung verwendet werden können. Bisher haben wir nur Bilder aus einer verauschten Probe x_k und dem Entrauschungsschritt t generiert. Die meisten Bildgeneratoren lassen sich jedoch durch einen Textprompt steuern, der die gewünschte Bilddarstellung beschreibt. Dafür erhält das neuronale Netz beispielsweise den Text 'Katze auf Bike' als zusätzlichen Eingabekanal in jedem Generationsschritt. Konditionierte DDPMs werden auf Paaren von Bildern und ihren zugehörigen Textbeschreibungen trainiert, die normalerweise aus Bild-Alt-Text-Tags im Internet stammen. So wird sichergestellt, dass das generierte Bild etwas darstellt, für das der 'Katze' und 'Bike' eine plausible Beschreibung wäre. Prinzipiell kann man generative Modelle an jede beliebige Eingabe konditionieren, solange man geeignete Trainingsdaten findet.

Geführte Diffusion

Die Steuerung des Diffusionsvorgangs nennt man auch 'guided Diffusion'. Wir können bei jedem Schritt des Diffusionsprozesses Konditionierungsinformationen y hinzufügen:

$$p_{\theta}(x_{0:T} | y) = p_{\theta}(x_T) \prod_{t=1}^T p_{\theta}(x_t | x_{t-1}, y)$$

Allgemein zielen geführte Diffusionsmodelle darauf ab, $\nabla \log p_{\theta}(x_t | y)$ zu lernen. Mithilfe der Bayes'schen Regel können wir die obige Gleichung umschreiben und vereinfachen zu:

$$\nabla_{x_t} \log p_{\theta}(x_t | y) = s \cdot \nabla_{x_t} \log p_{\theta}(y | x_t) + \nabla_{x_t} \log p_{\theta}(x_t)$$

Dabei ist s ein Führungs-Skalaterm [Wen-21]. [Chi-23] benutzen diese Klassifikator-gestützte Führung, um die Aktionsgenerierung anhand von latenten Eingangsdaten zu konditionieren. **Kernidee:** Das Diffusions-Modell lernt, Aktionen basierend auf Beobachtungen vorherzusagen. Durch die Konditionierung, kann das Modell Aktionssequenzen basierend auf visuellen und anderen Zustandsdaten erzeugen.

Anhand von Abb. 2.6 lässt sich die geführte Diffusion gut erklären. Der Vorwärtsprozess

verschiebt die klassifizierten Punkte zufällig in x- und y-Richtung durch aufgebrachtes Rauschen. Das Modell lernt durch zusätzliche Klassifizierer 1 bzw. 2 für jede Klasse ein Gradientenfeld. Im Rückwärtsprozess leiten die Felder schrittweise die Punkte zu ihrer jeweiligen Ausgangsposition zurück. Die Punkte werden durch Konditionierung geführt. Das ist jedoch ein schrittweiser Prozess und die Gradientenfelder verschieben sich abhängig vom Zeitschritt. Beim Sampeln schieben die Gradientenfelder die Punkte schrittweise zu einem Ziel. Das ist ein sehr wichtiger Punkt. Das Gradientenfeld aus Schritt 50 muss nicht dem aus Schritt 1 ähneln. Das ist sehr wichtig, um später zu verstehen, warum weniger Schritte oder größere Schrittweiten zu schlechteren Lösungen führen.

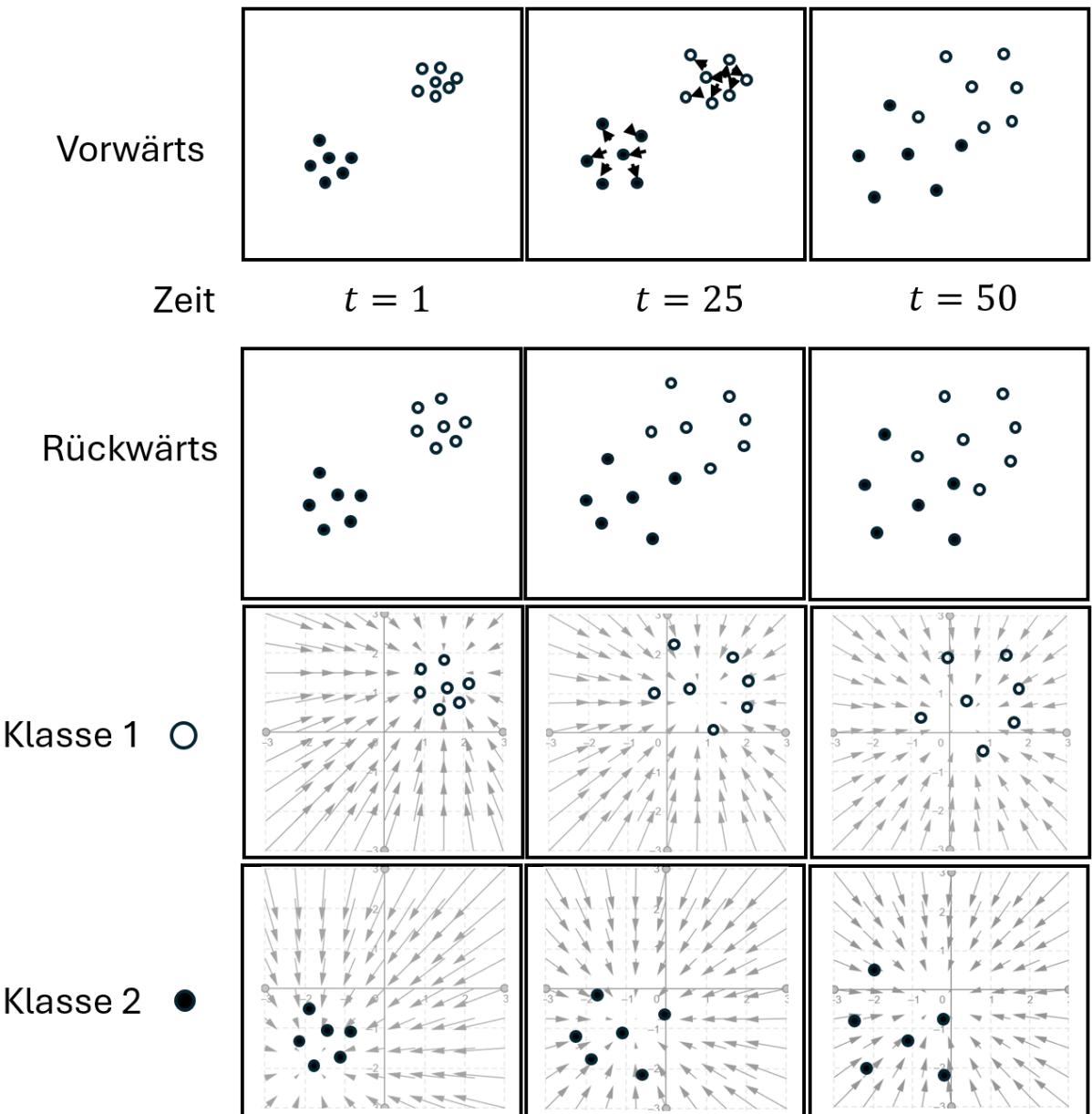


Abbildung 2.6: Geführte Diffusion anhand von zwei Klassen für ein einfaches Beispiel [Pie-23]

Konditionierung auf visuelle Observationen

In der Robotik soll dieses Diffusionsmodell nun für die Generierung von Aktionssequenzen verwendet werden. Weiterhin wurde das Rauschvorhersagenetzwerk $\varepsilon_\theta(x_k, k)$ bisher nur mit dem Bild x_t und dem Iterationsschritt t konditioniert. Es kann also nur aus einem Bild das Rauschen vorhersagen. Wir wollen jedoch aus mehreren Bildern und Roboterinformationen das Rauschen vorhersagen. D.h. der Entrauschungsprozess muss anhand von Beobachtungen O_t konditioniert werden. Die Beobachtung ist dabei zusammengesetzt aus den zwei letzten

Bildern zum Zeitpunkt vom Roboter O_t und den 16 nächsten Zustandspositionen A_t^k zum Zeitpunkt t . Anstatt der zwei Klassen aus Abb. 2.6, lernt das Modell aus sehr vielen latenten Observations-Klassen.

Wir verwenden einen Diffusionsansatz, um die bedingte Wahrscheinlichkeitsverteilung der Endeffektorposition in Abhängigkeit von Roboterbildern zu approximieren:

$$p(A_t | O_t) \text{ statt } p(A_t, O_t) \quad (2.15)$$

Damit ändert sich die Diffusionsgleichung 2.9 zu:

$$A_t^{k-1} = \alpha \left(A_{k,t} - \gamma \varepsilon_\theta(O_t, A_t^k, k) + N_{0,\sigma^2 I} \right) \quad (2.16)$$

Dabei ist A_t^{k-1} die entrauschte Robotersequenz. Die Trainingsverlustfunktion (Gl. 2.12) verändert sich nun. Das Rauschvorhersagenetz ε_θ bekommt latente Observationen O_t , die verrauschte Aktionssequenz $A_t^0 + \varepsilon_t$ sowie den aktuellen Zeitschritt t als Parameter. Abhängig von diesen Parametern erfolgt das Training des Netzwerks.

$$\mathcal{L}_{DPPM,mod} = \text{MSE} \left(\varepsilon_t, \varepsilon_\theta(O_t, A_t^0 + \varepsilon_t, t) \right) \quad (2.17)$$

2.10 Tricks zur Verbesserung des Modells

Parametrisierung der hinzugefügten Varianz β_t

Der Entrauschungsprozess des Denoising Diffusion Probabilistic Model (DDPM) erfolgt schrittweise gemäß der Gleichung 2.2. Wie viel Rauschen dabei pro Schritt hinzugefügt wird, bestimmt der Rauschzeitplan. Würde man in jeder Stufe eine konstante Menge an Rauschen auf das Bild aufschalten, wäre bei der Hälfte der Durchläufe das Bild bereits fast vollständig verrauscht (siehe Abb. 2.7 oben). Folgt man jedoch dem quadratischen Kosinus-Zeitplan verhält sich das Rauschen sehr viel gleichmäßiger [Ho-20]. Die Wahl des Rauschschemas, das während des Trainings verwendet wird, wirkt sich auf die Leistung des Diffusionsmodells aus. Wenn sich die Leistung zum Testzeitpunkt nicht wesentlich ändert, wenn also Trainingsschritte übersprungen werden, bedeutet dies, dass das Modell für einen Bruchteil der Trainingsschritte nichts gelernt hat. Es wird daher versucht, einen Zeitplan zu finden, bei dem sich jeder Trainingsschritt positiv auf die Bildqualität zum Testzeitpunkt auswirkt. Der Kosinus-Zeitplan findet hier den besten Mittelweg.

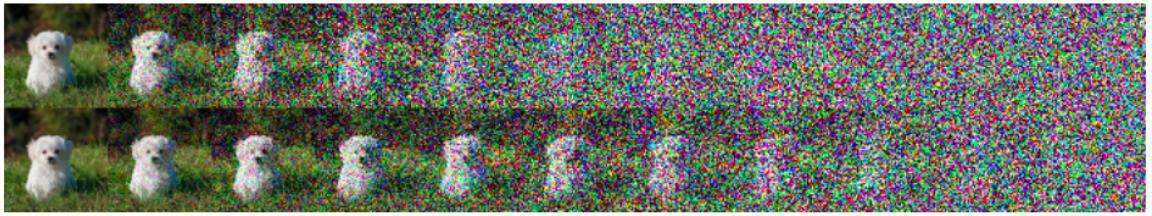


Abbildung 2.7: Rauschproben aus linearen (oben) und Cosinus (unten) Verrauschungsplänen mit linearen Zeitwerten t von 0 bis T [Nic-21]

Parametrisierung durch Lernen

In [Ho-20] wurde entschieden, die Rauschstärke im Vorwärtsprozess β_t als Konstante zu fixieren, anstatt sie lernbar zu machen. Dabei setzen sie $\Sigma_\theta(x_t, t) = \sigma_t^2 I$ aus der Gl. 2.7, wobei σ_t nicht gelernt, sondern entweder auf β_t oder $\tilde{\beta}_t = \frac{1-\alpha t-1}{1-\alpha t} \beta_t$ gesetzt wird. Dies führt laut den Autoren zu instabiler Optimierung und schlechterer Stichprobenqualität. [Nic-21] schlagen stattdessen vor, $\Sigma_\theta(x_t, t)$ als Interpolation zwischen β_t und $\tilde{\beta}_t$ zu modellieren, indem sie einen Mischvektor v vorhersagen: $\Sigma_\theta(x_t, t) = \exp(v \log \beta_t + (t - v) \log \tilde{\beta}_t)$

Dies ist eine Erweiterung, die in späteren Arbeiten entwickelt wurde, um die Leistung weiter zu verbessern. Die Idee dahinter ist, dass anstatt eine feste Varianz zu verwenden, eine gelernte Varianz-Schätzung verwendet wird. Dabei wird ein zusätzlicher Faktor v eingeführt, der es erlaubt, die Gewichtung zwischen den beiden Termen $\log \beta_t$ und $\log \tilde{\beta}_t$ dynamisch anzupassen.

2.11 Weiterführende Tricks zur Verbesserung des Modells

Denoising Diffusion Implicit Models (DDIM)

Denoising Diffusion Implicit Models (DDIM) ist ein schnelleres Lösverfahren der Gl. 2.14. Dieser Ansatz entkoppelt die Anzahl der Entrauschungsschritte beim Training und bei der Inferenz, wodurch der Algorithmus weniger Iterationen K_{ent} für die Inferenz verwendet. Er verfolgt ebenfalls eine deterministische Sampling-Strategie [Nic-21]. Beim Rückwärtsprozess wird ein gestufter Stichprobenplan verwendet. Stichprobenaktualisierung werden nur alle $[T/S]$ Schritte durchgeführt, um den Prozess von T auf S Schritte zu verkürzen. Der neue Stichprobenplan für die Erzeugung lautet $\{\tau_1, \dots, \tau_S\}$ wobei $\tau_1 < \tau_2 < \dots < \tau_S \in [1, T]$ und $S < T$.

Die bedingte Rückführungswahrscheinlichkeit kann durch eine gewünschte Standardabwe-

chung σ_t parametrisiert werden. Damit lautet die Rückführung

$$q(\mathbf{x}_{t-1} | \mathbf{x}_t, \mathbf{x}_0) = \mathcal{N}(\mathbf{x}_{t-1}; \tilde{\boldsymbol{\mu}}(\mathbf{x}_t, \mathbf{x}_0), \tilde{\beta}_t \mathbf{I})$$

Die Varianz β_t ist die quadrierte gewünschte Standardabweichung:

$$\tilde{\beta}_t = \sigma_t^2 = \frac{1 - \bar{\alpha}_{t-1}}{1 - \bar{\alpha}_t} \cdot \beta_t$$

Sie wird durch einen Hyperparameter $\eta \in \mathbb{R}^+$ kontrolliert:

$$\sigma_t^2 = \eta \cdot \tilde{\beta}_t = \eta \cdot \frac{1 - \bar{\alpha}_{t-1}}{1 - \bar{\alpha}_t} \cdot \beta_t \quad (2.18)$$

Der Spezialfall von $\eta = 0$ macht den Sampleprozess deterministisch. Ein solches Modell wird als Denoising Diffusion Implicit Model [Son-22] bezeichnet. Das DDIM hat die gleiche marginale Rauschverteilung, bildet aber das Rauschen deterministisch auf die ursprünglichen Datenproben ab. Die Generierung muss nicht der gesamten Kette folgen, sondern nur einer Teilmenge von Schritten. Bezeichnet man $s < t$ als zwei Schritte in dieser beschleunigten Trajektorie, dann ist der DDIM-Aktualisierungsschritt:

$$q_{\sigma, s < t}(x_s | x_t, x_0) = \mathcal{N}\left(x_s; \alpha_s \left(x_{t-1} - \frac{\alpha_t \epsilon_\theta(t)(x_t)}{\alpha_t}\right) + (1 - \alpha_s - \sigma_t^2) \epsilon_\theta(t)(x_t), \sigma_t^2 I\right) \quad (2.19)$$

In [Son-22] wurden Modelle mit 1000 Diffusionsschritten trainiert. DDIM-Modelle mit $\eta = 0$ liefern die beste Proben-Qualität wenn S klein ist. Dagegen schneiden DDPM-Modelle mit $\eta = 1$ bei kleinen S viel schlechter ab. Wenn man es sich leisten kann, die vollständigen Reverse-Markov-Diffusionsschritte durchzuführen, schneiden DDPM-Modelle mit $\eta = 1$ jedoch besser ab. DDIM ist demnach sinnvoll, wenn der Rückwärtsprozess so schnell wie möglich durchlaufen werden soll.

Progressive Destillation

Um Diffusionsmodelle während der Samplingsphase effizienter zu machen, kann *progressive Destillation (PD)* verwendet werden: ein Algorithmus, der die Anzahl der erforderlichen Sampling-Schritte iterativ halbiert, indem ein langsames Lehrermodell in ein schnelleres Schülermodell distilliert wird. Algorithmus 1 und Algorithmus 2 in Abb. 2.8 zeigen das Training eines Diffusionsmodells und die progressive Destillation nebeneinander, wobei die Änderungen bei der progressiven Destillation in Grün hervorgehoben sind.

Um das zu erreichen, wird zuerst ein Schülermodell aus dem Lehrermodell initialisiert. Das neue Student-Modell entrauscht auf ein Ziel, bei dem ein Schüler-DDIM-Schritt zwei Schritte des Lehrers entspricht. Dabei wird nicht die ursprüngliche Rauschprobe als Ziel für

die Rauschunterdrückung verwendet, sondern der Schüler soll in einem Schritt entrauschen, für was der Lehrer zwei Schritte braucht. In jeder progressiven Destillationsiteration können somit die notwendigen Entrauschungsschritte K_{ent} halbiert werden. Im Paper zeigen die Autoren, wie durch wiederholtes Distillieren mehrere tausend Inferenzschritte auf acht reduziert werden können ohne große Qualitätseinbußen. Diese Einsparung soll es dem RL-Agenten ermöglichen, sich effizient an neue Umgebungen anzupassen. Pseudocodes für beide Trainingsprozesse werden in Abb. 2.8 dargestellt. Hierbei ist links das Standardtraining und rechts das Destillations-Training:

Algorithm 1 Standard diffusion training

Require: Model $\hat{\mathbf{x}}_\theta(\mathbf{z}_t)$ to be trained
Require: Data set \mathcal{D}
Require: Loss weight function $w()$

```

while not converged do
     $\mathbf{x} \sim \mathcal{D}$                                  $\triangleright$  Sample data
     $t \sim U[0, 1]$                                  $\triangleright$  Sample time
     $\epsilon \sim N(0, I)$                                  $\triangleright$  Sample noise
     $\mathbf{z}_t = \alpha_t \mathbf{x} + \sigma_t \epsilon$      $\triangleright$  Add noise to data

     $\tilde{\mathbf{x}} = \mathbf{x}$        $\triangleright$  Clean data is target for  $\hat{\mathbf{x}}$ 
     $\lambda_t = \log[\alpha_t^2 / \sigma_t^2]$            $\triangleright$  log-SNR
     $L_\theta = w(\lambda_t) \|\tilde{\mathbf{x}} - \hat{\mathbf{x}}_\theta(\mathbf{z}_t)\|_2^2$      $\triangleright$  Loss
     $\theta \leftarrow \theta - \gamma \nabla_\theta L_\theta$      $\triangleright$  Optimization
end while

```

Algorithm 2 Progressive distillation

Require: Trained teacher model $\hat{\mathbf{x}}_\eta(\mathbf{z}_t)$
Require: Data set \mathcal{D}
Require: Loss weight function $w()$
Require: Student sampling steps N

```

for  $K$  iterations do
     $\theta \leftarrow \eta$                                  $\triangleright$  Init student from teacher
    while not converged do
         $\mathbf{x} \sim \mathcal{D}$ 
         $t = i/N, i \sim Cat[1, 2, \dots, N]$ 
         $\epsilon \sim N(0, I)$ 
         $\mathbf{z}_t = \alpha_t \mathbf{x} + \sigma_t \epsilon$ 
        # 2 steps of DDIM with teacher
         $t' = t - 0.5/N, t'' = t - 1/N$ 
         $\mathbf{z}_{t'} = \alpha_{t'} \hat{\mathbf{x}}_\eta(\mathbf{z}_t) + \frac{\sigma_{t'}}{\sigma_t} (\mathbf{z}_t - \alpha_t \hat{\mathbf{x}}_\eta(\mathbf{z}_t))$ 
         $\mathbf{z}_{t''} = \alpha_{t''} \hat{\mathbf{x}}_\eta(\mathbf{z}_{t'}) + \frac{\sigma_{t''}}{\sigma_{t'}} (\mathbf{z}_{t'} - \alpha_{t'} \hat{\mathbf{x}}_\eta(\mathbf{z}_{t'}))$ 
         $\tilde{\mathbf{x}} = \frac{\mathbf{z}_{t''} - (\sigma_{t''}/\sigma_t) \mathbf{z}_t}{\alpha_{t''} - (\sigma_{t''}/\sigma_t) \alpha_t}$      $\triangleright$  Teacher  $\hat{\mathbf{x}}$  target
         $\lambda_t = \log[\alpha_t^2 / \sigma_t^2]$ 
         $L_\theta = w(\lambda_t) \|\tilde{\mathbf{x}} - \hat{\mathbf{x}}_\theta(\mathbf{z}_t)\|_2^2$ 
         $\theta \leftarrow \theta - \gamma \nabla_\theta L_\theta$ 
    end while
     $\eta \leftarrow \theta$                                  $\triangleright$  Student becomes next teacher
     $N \leftarrow N/2$                                  $\triangleright$  Halve number of sampling steps
end for

```

Abbildung 2.8: Vergleich von Algorithmus 1 (Training des Diffusionsmodells) und Algorithmus 2 (progressive Destillation) nebeneinander, wobei die relativen Änderungen bei der progressiven Destillation grün hervorgehoben sind. (Bildquelle: [Sal-22])

Um die Destillation starten zu können, wird ein Trainer benötigt. Dieser löst den Entrauschungsprozess in N Entrauschungsschritten. Sei $N = 100$. Nun wird für K Destillations-Schritte distilliert. D.h. K mal wird $N = 100$ halbiert.

Die progressive Destillation beginnt mit einem Lehrermodell, das auf herkömmliche Weise

trainiert wurde. In jeder Iteration der Destillation initialisieren wir das Schülermodell als Kopie des Lehrers, wobei dieselben Parameter und dieselbe Modelldefinition verwendet werden. Wie beim Standard-Training werden anschließend Daten aus dem Trainingsset gesampelt und mit Rauschen versehen. Der Trainingsverlust wird durch Anwendung des Schülermodells auf die verrauschten Daten z_t berechnet. Der Hauptunterschied bei der progressiven Destillation liegt jedoch in der Zielsetzung für das Denoising-Modell: Statt die ursprünglichen Daten x zu verwenden, wird das Schülermodell darauf trainiert, ein Ziel \tilde{x} vorherzusagen, das einen einzelnen DDIM-Schritt des Schülers mit zwei DDIM-Schritten des Lehrers abgleicht.

Das Ziel \tilde{x} wird berechnet, indem zwei DDIM-Sampling-Schritte mit dem Lehrermodell durchgeführt werden, beginnend bei z_t und endend bei $z_{t-1/N}$, wobei N die Anzahl der Sampling-Schritte des Schülers ist. Durch Umkehrung eines einzigen DDIM-Schritts wird dann der Wert berechnet, den das Schülermodell vorhersagen müsste, um in einem Schritt von z_t zu $z_{t-1/N}$ zu gelangen (siehe Details in Anhang G). Das resultierende Ziel $\tilde{x}(z_t)$ ist vollständig durch das Lehrermodell und den Startpunkt z_t bestimmt, was es dem Schülermodell ermöglicht, eine präzise Vorhersage bei z_t zu treffen. Im Gegensatz dazu ist der ursprüngliche Datenpunkt x nicht eindeutig durch z_t bestimmt, da verschiedene x -Werte dasselbe verrauschte z_t erzeugen können. Das ursprüngliche Denoising-Modell sagt daher einen gewichteten Durchschnitt möglicher x -Werte voraus, was zu unscharfen Vorhersagen führt. Durch schärfere Vorhersagen kann das Schülermodell beim Sampling schneller Fortschritte machen.

Nach der Destillation zu einem Schülermodell mit N Sampling-Schritten kann die Prozedur mit $N/2$ Schritten wiederholt werden: Das Schülermodell wird zum neuen Lehrermodell, und ein neues Schülermodell wird durch Kopieren dieses Modells initialisiert.

Im Gegensatz zum ursprünglichen Training, wird progressive Destillation immer in diskreter Zeit durchgeführt: Die diskreten Zeitpunkte werden so gewählt, dass der höchste Zeitindex einem Signal-Rausch-Verhältnis von null entspricht, d. h. $\alpha_1 = 0$. Dies stimmt genau mit der Verteilung des Eingangsrauschens $z_1 \sim \mathcal{N}(0, I)$ überein, die zur Testzeit verwendet wird [Sal-22].

Die Kullback-Leibler-Divergenz in Diffusion Policies

Die **Kullback-Leibler-Divergenz (KL-Divergenz)** ist eine zentrale Metrik in distillierten Diffusion Policies, die den Unterschied zwischen der *Lehrerpolitik* $p(a|s)$ und der *Schülerpolitik* $q_\theta(a|s)$ misst. Sie spielt eine entscheidende Rolle bei der Optimierung und Anpassung der Schülerpolitik an die Zielverteilung der Lehrerpolitik was bei progressiver Destillation angewandt wird.

Die KL-Divergenz zwischen zwei Wahrscheinlichkeitsverteilungen $p(a|s)$ und $q_\theta(a|s)$ wird wie folgt definiert:

$$D_{\text{KL}}(p\|q_\theta) = \int p(a|s) \log \frac{p(a|s)}{q_\theta(a|s)} da.$$

Hier beschreibt:

- $p(a|s)$: Wahrscheinlichkeitsverteilung der Lehrerpolitik,
- $q_\theta(a|s)$: Wahrscheinlichkeitsverteilung der Schülerpolitik, die durch die Diffusionspolitik approximiert wird.

Rolle in Diffusion Policies

1. **Anpassung der Schülerpolitik:** Die KL-Divergenz wird minimiert, um $q_\theta(a|s)$ möglichst eng an $p(a|s)$ anzupassen:

$$\mathcal{L}_{\text{KL}} = \mathbb{E}_{p(a|s)} \left[\log \frac{p(a|s)}{q_\theta(a|s)} \right].$$

2. **Score-Matching:** In Diffusionsmodellen wird die KL-Divergenz genutzt, um die Zielverteilung im Rahmen des Rückwärtsprozesses (*Reverse Process*) zu approximieren. Dies stellt sicher, dass die generierten Aktionen konsistent mit der Lehrerpolitik sind [Ho-20].
3. **Stochastische Steuerung:** Da Diffusion Policies stochastische Prozesse modellieren, bewahrt die Minimierung der KL-Divergenz die Vielfalt der Lehrerpolitik und ermöglicht flexible, adaptive Steuerungsentscheidungen [Son-22].

Konsistenzmodelle

Konsistenzmodelle [Son-23b] sind eine Familie von Ein-Schritt-Generierungsmodellen, die lernen, jeden teilweise verrauschten Datenpunkt in den finalen Datenpunkt zu überführen. Solche Modelle können als Schülermodell für die Destillation verwendet werden. Jedoch wurde Konsistenztraining auch vorgeschlagen, um Konsistenzmodelle von Grund auf in einem End-to-End-Verfahren zu trainieren [Son-23b, Son-23a]. Konsistenztraining erzwingt Konsistenz zwischen empirischen x_t - und x_{t+d} -Beispielen, was jedoch bei jedem Diskretisierungsschritt zu einem irreduzierbaren Bias aufgrund von Ambiguität führt.

2.12 Architektur und Prozessanpassungen für die Diffusion Policy

Geschlossene Regelschleife während des Einsatzes

Die angelernte Diffusion Policy arbeitet in einer geschlossenen Regelschleife. Zum Zeitpunkt 1 werden Zustandsdaten ausgelesen und eine Aktionssequenz von der Länge 8 bestimmt. Das

sind z.B. die nächsten acht Positionen des Endeffektors. Anschließend werden diese acht Positionen angefahren. Dann werden erneut zwei Beobachtung aufgenommen. Nun wiederholt sich der Prozess. DP generiert fortlaufend neue Aktionen basierend auf aktuellen Beobachtungen wie dargestellt in Abb. 2.9.

In der CNN (Convolutional Neural Network) Methode, zieht ein visual Encoder visuelle Features aus den zwei aufgenommenen Bildern. Diese zwei visuellen Einbettungen werden mit zwei niedrig dimensionalen Zuständen zu einem latenten Beobachtungs-Tensor verknüpft. Es werden zwei Bilder genommen, um dem Modell einen zeitlichen Kontext zu geben. Wenn der Roboter im letzten Bild weiter rechts ist, wird dadurch diese Bewegungsrichtung bekannt. Der vorhergesagte Handlungshorizont bietet einen Blick in zukünftige Handlungsabläufe. Die in Abb. 2.9 genannten Horizontwerte sind empirisch ermittelt und erzielen das beste Roboterverhalten [Chi-23]. Anstatt die komplette Robotersequenz vorherzusagen, wird das Problem in Teilsequenzen heruntergebrochen. Dadurch entgeht die DP der Fehlerakkumulation über die Zeit, welche bei BC normalerweise ein großes Problem ist.

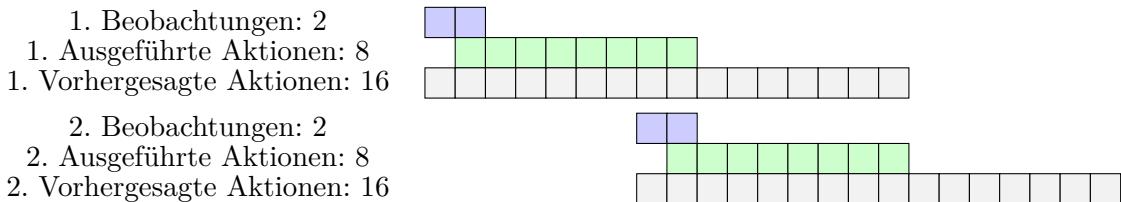


Abbildung 2.9: Zeitliche Versetzung von Beobachtungen und vorhergesagten Aktionen: Beobachtungsdimension = 2, Aktionshorizont für ausgeführte Aktionen = 8, Aktionshorizont für vorhergesagte Aktionen = 16

Trainingsdaten für das visuelle Training

In dieser Sektion wird der Aufbau der Trainingsdaten erklärt. Da das visuelle DDPM Daten der Form aus Abb. 2.9 braucht, müssen diese entsprechend generiert werden.

Aufnahme von Trainingsdaten

1. Zunächst wird eine Trainingsepisode gestartet. Dabei ist eine Episode eine Sequenz aus Roboteraktionen, die zur Lösung der Aufgabe führt.
2. Zu jedem Zeitpunkt wird ein Bild des Roboters, sowie eine Aktion in Form eines Vektors aufgenommen und in einem Array gespeichert.
3. Episodenstart- und Ende werden in einer separaten Datei 'Episoden-Längen' gespeichert. Endet eine Demonstration, wird das Episodenende gespeichert, aber Bilder und Aktionen werden weiterhin aufgenommen.

4. Hat man genügend Episoden aufgenommen, endet der Prozess.
5. Man hat nun ein Fortlaufendes Bild-Array mit N Bildern (img). N: Die Länge der Demonstrationsschritte. Des Weiteren hat man ein fortlaufendes Aktions-Array (actions) mit N gesammelten Positionen, sowie ein Episode-Ende-Array (episode-lengths), welches alle Episodenlängen beinhaltet.
6. Die Bild- und Aktionsdaten werden in jeweils einer Datenreihe aufgenommen, damit sie anschließend einfacher normalisiert werden können. Hierdurch wird der Trainingsprozess stabiler.

Sampeln von Sequenzen aus diesen Trainingsdaten

1. Um nun wie in Abb. 2.9 geformte Trainingsdaten aus dem Rohdatensatz zu beziehen, gibt es Hilfsfunktionen, die anhand des Arrays 'Episoden-Längen' Trainingssequenzen der o.g. Form aus den Episoden beziehen. Hierbei wird darauf geachtet, dass keine Episodengrenzen überschritten werden, und dass die Anfangsschritte der Episode vermieden werden.
2. Der sog. dataloader bezieht dann eine batch von $64 * 2$ Beobachtungen, $64 * 8$ ausgeführten Aktionssequenzen und $64 * 16$ vorhergesagten Aktionssequenzen
3. Die 64 batches werden anschließend normalisiert

Prozessablauf

DP setzt sich aus einem Diffusionsmodell (DDPM) mit Aktionsen- und Decoder, sowie einem State Encoder zusammen. Der Prozessablauf ist in Abb. 2.10 dargestellt.

Komponenten

1. Der Zustandsencoder (ES) verarbeitet Bilddaten mit einem CNN und $S_{t,img}$ und andere Zustandsinformationen $S_{t,low-dim}$ - kombiniert diese
2. Der Aktionsencoder (EA) verarbeitet verrauschte Aktionssequenzen A_t^k und die Observations-Kondition aus Schritt k h_t und produziert eine Repräsentation aus dem tiefsten CNN Layer X_t^k sowie der Skip Connection $X_{t,skip}^k$
3. Der Aktionsdecoder (DA):
 - Schätzt das hinzugefügte Rauschen anhand von A_t^k , X_t^k und $X_{t,skip}^k$
 - Entfernt dieses Rauschen schrittweise in K_{ent} Entrauschungsschritten
 - Erzeugt die finale entrauschte Aktionssequenz A_t^{k-1}

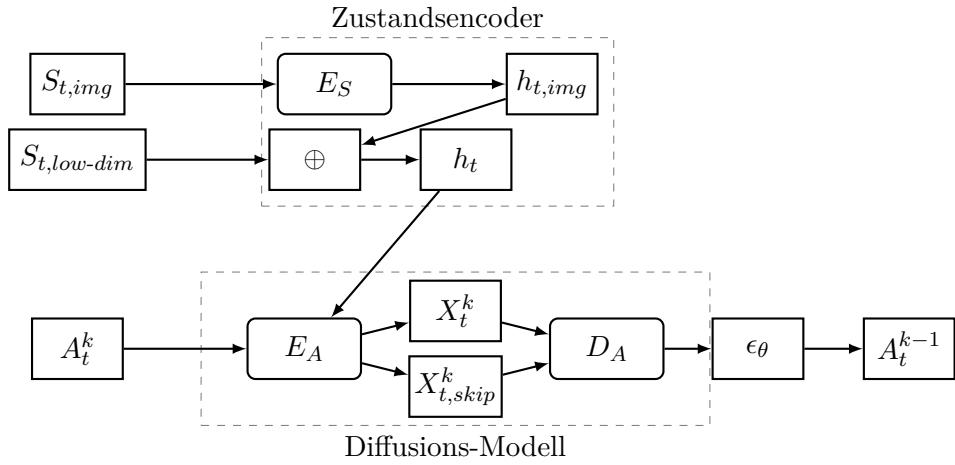


Abbildung 2.10: Selbsterstellte Darstellung des DP-Prozesses. E_S : State Encoder, $h_{t,img}$: Visuelle Embeddings, h_t : Observations-Kondition, E_A : Action Encoder, D_A : Action Decoder, ϵ_θ : Geschätztes Rauschen, A_t^k : Verrauschte Aktionssequenz, A_t^{k-1} : Entrauschte Aktionssequenz, X_t^k : eine Repräsentation aus dem tiefsten CNN Layer, $X_{t,skip}^k$: Skip Connection Layer

[Chi-23] [Li-24]

Zustandsencoder

In Diffusion Policies ist der Zustandsencoder (oder State Encoder) ein wichtiger Baustein, der den aktuellen Zustand einer Umgebung in eine latente Repräsentation transformiert. Diese Repräsentation wird dann von der Diffusionsmodellstruktur genutzt, um die nächsten Schritte oder Aktionen zu planen. Der Zustandsencoder ist also entscheidend für die Fähigkeit des Modells, den aktuellen Kontext zu erfassen und in den Entscheidungsprozess einfließen zu lassen.

Der Zustandsencoder fasst den möglicherweise sehr komplexen und hochdimensionalen Zustandsraum zusammen, z. B. in Form von Bilddaten, Sensorwerten oder anderen Umweltinformationen. Dadurch wird die Menge an Informationen reduziert, die das Modell verarbeiten muss, ohne relevante Details zu verlieren.

Ein gut trainierter Zustandsencoder extrahiert die wichtigsten Merkmale des Zustands, die für die Policy-Entscheidungen relevant sind. Das Modell lernt so, unwichtige Details zu ignorieren und sich auf die Schlüsselinformationen zu konzentrieren, die für die Zielerreichung nötig sind.

Durch die Transformation in eine latente Darstellung kann der Zustandsencoder helfen, Rau-

schen und Unsicherheiten im Eingaberaum (z. B. durch verrauschte Sensordaten) abzufangen, was die Stabilität der Policy-Entscheidungen erhöht.

Insgesamt trägt der Zustandsencoder dazu bei, den Zustand der Umgebung effektiv zu interpretieren und eine stabile, gut verallgemeinerbare Grundlage für die nachfolgenden Entscheidungsprozesse zu schaffen. Diese Verarbeitungsweise ähnelt dem Ansatz bei anderen neuronalen Netzwerken, die Zustandsinformationen in Entscheidungsprozesse einfließen lassen, nutzt aber die spezifischen Vorteile der Diffusionsmodellarchitektur.

Modellarchitektur U-Net

Für die Rauschvorhersage zeitlicher Daten wird eine U-Net Architektur verwendet. Diese Struktur erlaubt es, den zeitlichen Kontext zu erlernen. Wie dieses U-Net verwendet wird, geht aus Abb. 2.11 hervor.

Architekturprinzip

Es wird eine U-Net-ähnliche Struktur verwendet, welche die hierarchische Merkmalsextraktion ermöglicht. Dieses kombiniert lokale und globale zeitliche Kontextinformationen. Die CNN-basierte Architektur wurde speziell für die temporale Verarbeitung von Aktionssequenzen optimiert, wobei der Fokus auf der robusten Generierung von Aktionen unter Berücksichtigung des Beobachtungskontexts liegt. Hierbei können für dieselbe Aufgabe auch gleichzeitig Bilder aus mehreren Perspektiven aufgenommen werden. Dabei sollten Bilder aus jeder Perspektive einen eigenen Vision Encoder haben. [Chi-23]

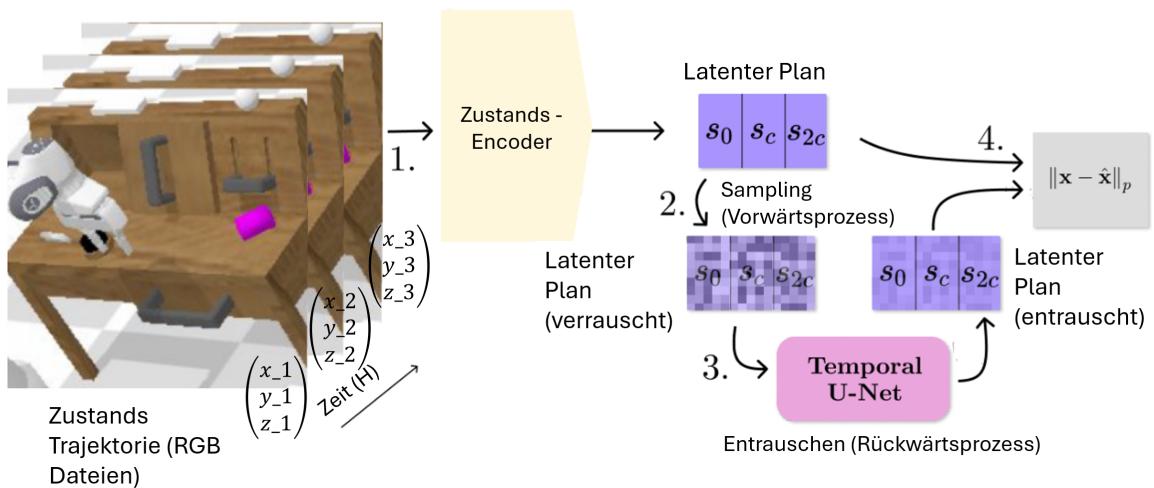


Abbildung 2.11: Übersicht der Trainingsarchitektur nach [Zha-24]

Temporal Convolutional Network

Tiefe neuronale Netze haben sich im Bereich des maschinellen Lernens und der künstlichen Intelligenz als leistungsstarke Lerntechnik erwiesen, indem sie verschiedene Anwendungen von der visuellen Erkennung und Spracherzeugung bis hin zum autonomen Fahren und dem Gesundheitswesen revolutioniert haben. Eine zentrale Komponente ihres Erfolges kann auf die umfangreichen Lernfähigkeiten von Convolutional Neural Networks (CNN) zurückgeführt werden, die bei der Bewältigung von Aufgaben, die Bilder, Graphen und Sequenzen beinhalten, außergewöhnlich effektiv sind.

In jeder CNN-Architektur finden wir einen Faltungsoperator, der darauf abzielt, eine Art von Korrelation (entweder zeitlich oder räumlich) auszunutzen, um Merkmale auf hoher Ebene zu lernen. Auf einer hohen Ebene können wir uns die Faltung als einen kleinen Filter (auch als Kernel bezeichnet) vorstellen, der über ein Eingabebild oder eine Sequenz gleitet und an jeder Position lokale Merkmale erfasst. Diese lokalen Merkmale werden dann kombiniert, um eine Merkmalskarte zu erstellen, die als Input für die nächsten Schichten des Netzwerks dient. Temporal Convolutional Networks (TCNs) sind auf die Analyse von Sequenzen spezialisiert. Sie werden aber auch für die Videogenerierung verwendet. Zweck ist es, sequentiellen Daten über längere Zeiträume zu erfassen. Ein TCN ist eine spezielle Form eines neuronalen Netzwerks für Zeitreihenanalyse. Die Faltung (Convolution) erfolgt nur in einer Dimension (1D). Es sollen glatte Aktionssequenzen generiert werden. TCN ist inspiriert von neueren Convolutional Networks für sequentielle Daten und kombiniert Einfachheit, autoregressive Vorhersage und sehr langes Gedächtnis. Das TCN ist aus zwei Grundprinzipien aufgebaut: Die Faltungen sind kausal, d.h. sie bauen zeitlich aufeinander auf. Informationen fließen von der Vergangenheit in die Zukunft. Die Architektur kann eine Sequenz beliebiger Länge nehmen und auf eine Outputsequenz derselben Länge abbilden. Um den ersten Punkt zu erreichen, verwendet das TCN kausale Faltungen, d.h. Faltungen, bei denen eine Ausgabe zum Zeitpunkt t nur mit Elementen aus dem Zeitpunkt t und früher in der vorhergehenden Schicht gefaltet wird. Um den zweiten Punkt zu erreichen, verwendet das TCN eine 1D Netzwerkarchitektur, bei der jeder Hidden Layer die gleiche Länge wie der Input Layer hat.

Dilatierte (gedehnte) Faltungen Einfache kausale Faltungen haben den Nachteil, dass sie nur in der Tiefe des Netzes linear in die Geschichte zurückblicken, d.h. das rezeptive Feld wächst linear mit jeder weiteren Schicht. Um diesen Umstand zu umgehen, verwendet die Architektur dilatierte Faltungen, die ein exponentiell großes rezeptives Feld ermöglichen. Die Dilatation ist gleichbedeutend mit der Einführung einer festen Stufe zwischen jeweils zwei benachbarten Filterabgriffen. Nach jeder Faltung wird eine Downsampling-Schicht angewendet. Hier wird Strided Convolution von 2 verwendet. D.h. bei jedem Schritt der Faltung über ein Input wird ein Pixel übersprungen.

Konditionierung des U-Nets

Das Rauschvorhersagenetzwerk der DP wird an dem Diffusionsschritt, der Zustandsinformationen, sowie der verrauschten Actionssequenz konditioniert. Das geht aus Gl. 2.17 hervor: $\varepsilon_\theta(O_t, A_t^0 + \varepsilon_k, k)$. Der Diffusions-Timestep k wird durch sinusoidale Positionenkodierung und ein Mehrlagiges Perzepron (Multilayer perceptron = MLP) verarbeitet. Globale Konditionierung (z.B. Beobachtungen) aus dem Zustandsencoder werden mit dem Timestep-Embedding konkateniert. Diese kombinierte Konditionierung wird in allen Blöcken via FiLM (Feature-wise Linear Modulation) angewandt. Die Eingabedaten sind typischerweise Sequenzen von Zeitpunkten (Bilder und Zustände).

Conditional Residual Blocks

Ein weiteres architektonisches Element eines TCNs sind residuale Verbindungen. Anstelle eines Convolutional Layers verwenden TCNs ein generisches residuales Modul. Jeder residuale Block enthält einen Zweig, der zu einer Reihe von Transformationen führt. Deren Outputs werden zum Input x des Blocks addiert.

Dies erlaubt es den Schichten effektiv, Modifikationen der Identitätsabbildung zu lernen, anstatt der gesamten Transformation, was sich als vorteilhaft für tiefe neuronale Netze erwiesen hat. Gerade bei sehr tiefen Netzen wird eine Stabilisierung wichtig, z.B. wenn die Vorhersage von einer großen Historiengröße mit einer hochdimensionalen Inputsequenz abhängt.

Ein residualer Block hat zwei Lagen dilatierter kausaler Faltungen (Conv1D) und Mish Aktivierung als Nichtlinearitäten. In diesen Blöcken wird FiLM verwendet, um die Generierung anhand der Zustandsinformationen zu konditionieren.

Downsampling-Pfad (Encoder)

Die Daten werden in mehreren Stufen heruntergesampelt, um auf höheren Ebenen abstrakte Merkmale zu erfassen. Jede Stufe besteht aus einem oder mehreren Residualblöcken (TCN-Blöcke), die 1D-Faltung verwenden, um das Merkmal im Zeitfenster zu erweitern. Es werden gedehnte Faltungen angewendet, um das Rezeptive Feld zu erweitern.

Mittelschicht ('Engpass')

Die Mittelschicht besteht aus TCN-Blöcken, die als 'Engpass' dienen. Hier werden die repräsentativsten Merkmale des gesamten Zeitfensters extrahiert. Diese Schicht erweitert das Rezeptive Feld noch weiter, sodass Muster über längere Zeiträume hinweg erkannt werden können.

Upsampling-Pfad (Decoder)

In diesem Teil wird das Signal schrittweise in die ursprüngliche zeitliche Auflösung gebracht. Jede Stufe des Upsampling-Pfades besteht ebenfalls aus TCN-Blöcken, die ähnliche Merkmale wie im Encoder wiederherstellen. In jeder Upsampling-Schicht werden Merkmale aus dem Encoder (Skip-Connections) kombiniert, was detailliertere Vorhersagen ermöglicht. Hier wird Transponierte Convolution (Stride=2) verwendet.

Endschicht

Die letzte Schicht führt die Ausgabe des Upsampling-Pfades wieder zu einer Zeitreihen-Prognose zusammen. Typischerweise ist dies ein Conv1D-Layer, der die Ausgabe auf die gewünschte Form (wie z.B. die Anzahl der Kanäle für eine Zeitreihe) bringt.

Skip Connections

Verbinden entsprechende Ebenen von Encoder und Decoder. Sie konkatenieren Features entlang der Kanal-Dimension und helfen bei der Erhaltung räumlicher Details.

Diese Architektur ist speziell für die Diffusion von zeitlichen Sequenzen optimiert, wobei die Konditionierung es ermöglicht, den Generierungsprozess anhand der gegebenen Bilder/Zustände zu steuern.

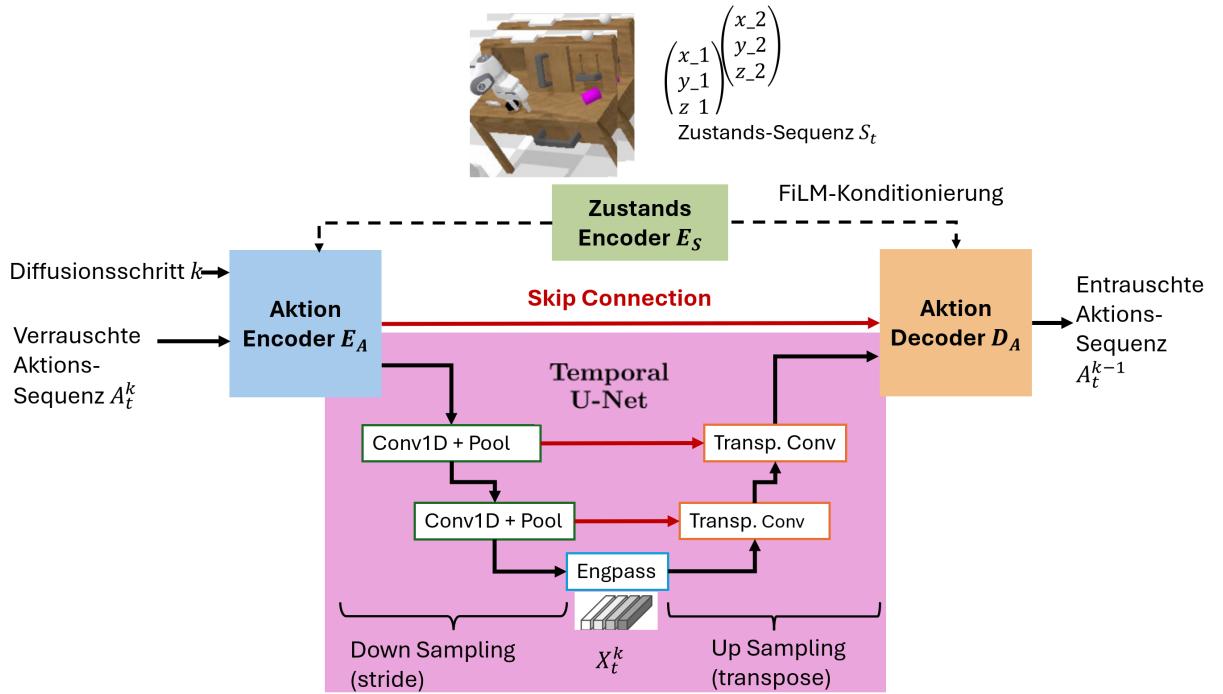


Abbildung 2.12: Übersicht der U-Net Architektur nach [Li-24]

Hauptmodifikationen

Die Aktionsgenerierung wird wie bereits erwähnt durch Beobachtungsmerkmale O_t konditioniert. Dabei kommt Feature-wise Linear Modulation (FiLM) zum Einsatz. Der Entrauschungsschritt k wird als zusätzliche Bedingung einbezogen. Es werden mehrere Bilder über das CNN interpretiert, wodurch die Verarbeitung von zeitlichen Sequenzen ermöglicht wird.

Einschränkungen

Die Architektur hat Schwächen bei schnell wechselnden Aktionensequenzen. Es gibt außerdem Probleme bei scharfen zeitlichen Änderungen (z.B. bei Geschwindigkeitsbefehlen). Die temporale Faltung führt zu einer Tendenz zu niederfrequenten Signalen. Einfach ausgedrückt wirken Temporale Faltungen wie ein natürlicher Tiefpassfilter: Sie glätten hochfrequente (schnelle) Änderungen und bewahren niedrfrequente (langsame) Änderungen. Das hat eine glättende Wirkung: Schnelle Zuckungen werden abgeschwächt und sanfte Bewegungen bleiben erhalten. Die Vorteile sind eine stabilere Roboterbewegung und natürlichere Aktionensequenzen. Es treten also möglicherweise langsamere Reaktion auf schnelle Änderungen auf. Auch der Verlust von feinen Bewegungs-Details ist möglich. Dies ist besonders beachtenswert bei Robotersteuerungen, da zu abrupte Bewegungen vermieden werden sollen. Die Bewegungen werden dadurch flüssiger und natürlicher was auch die mechanische Belastung reduziert.

3 Stand der Technik

Die Diffusion Policy (DP) hat seit ihrer Einführung durch [Chi-23] verschiedene Weiterentwicklungen erfahren. Dieser Abschnitt gibt einen Überblick über die wichtigsten Verbesserungen und Anwendungen.

3.1 Verbesserung durch Rekonstruieren der visuellen Repräsentation

Crossway Diffusion ist eine Methode zur Verbesserung des diffusionsbasierten visuomotorischen Lernens durch einen Zustandsdecoder und ein zusätzliches selbstüberwachtes Lernziel (Self-Supervised-Learning). Der Zustandsdecoder rekonstruiert Rohbildpixel und andere Zustandsinformationen aus den Zwischendarstellungen X_t^k des umgekehrten Diffusionsprozesses. Dabei wird eine neue Verlustfunktion $\mathcal{L}_{Recon.} = \text{MSE}(S_t, \hat{S}_t)$ gebildet. Diese vergleicht den Zustand mit dem rekonstruierten Zustand. Das gesamte Modell wird gemeinsam durch das SSL-Ziel und den ursprünglichen Diffusionsverlust $\mathcal{L}_{DPPM,mod}$ optimiert:

$$\mathcal{L}_{Crossway} = \mathcal{L}_{DDPM} + \alpha \mathcal{L}_{Recon}$$

[Li-24]. In ausgewählten Aufgaben führt das zu einer Verbesserung von 15.7 % gegenüber der Basis-DP.

3.2 Generalisierung und dateneffizientes Lernen

Octo repräsentiert eine innovative Transformer-basierte Diffusion-Policy-Architektur, die verschiedene Modalitäten $M = \{m_1, \dots, m_n\}$ wie visuelle Eingaben, propriozeptive Daten und Sprachanweisungen in einem einheitlichen Framework integriert. Die Policy $\pi_\theta(a_t|s_t, c)$ lernt aus einem umfangreichen Datensatz \mathcal{D} von Roboterinteraktionen und kann Aktionen a_t basierend auf dem aktuellen Zustand s_t und dem Kontext c generieren.

Die Architektur nutzt einen multimodalen Transformer $T(X)$, der Eingaben verschiedener Modalitäten verarbeitet und diese in einen gemeinsamen latenten Raum \mathcal{Z} projiziert. Durch End-to-End-Training auf diversen Aufgaben $\mathcal{T} = \{t_1, \dots, t_k\}$ entwickelt das System eine generalisierte Fähigkeit zur Aufgabenlösung. Die Integration von Demonstrations-Lernen erfolgt durch einen Imitations-Loss \mathcal{L}_{IL} , während die Gesamtperformanz durch einen kombinierten Verlust

$$\mathcal{L}_{total} = \mathcal{L}_{IL} + \lambda \mathcal{L}_{task}$$

optimiert wird, wobei λ die relative Gewichtung zwischen Imitations- und Aufgaben-spezifischem Verlust kontrolliert. Die Policy demonstriert Zero-Shot-Generalisierung auf neue Aufgaben $t_{new} \notin \mathcal{T}$ und kann über verschiedene Roboterplattformen hinweg eingesetzt werden. Diese Generalisierungsfähigkeit wird durch die Transformer-Architektur und das extensi-ve Training auf verschiedenen Aufgabentypen erreicht, was Octo zu einem vielversprechenden Ansatz für generalistische Robotersteuerung macht [Oct-24].

EquiBot erweitert diesen Ansatz durch die Integration von äquivarianten neuronalen Netzen, welche die geometrischen Symmetrien in Roboteraufgaben explizit berücksichtigen. Die Policy π_{equi} ist invariant gegenüber Rotationen und Translationen im 3D-Raum, was die Genera-lisierung auf neue Perspektiven und Objektpositionen verbessert. Die äquivariante Struktur wird durch spezielle Konvolutionsoperationen erreicht: $\mathcal{F}_{equi}(g \cdot x) = g \cdot \mathcal{F}_{equi}(x)$ wobei g eine geometrische Transformation darstellt [Yan-24].

3D Diffusion Actor ergänzt diese Ansätze durch eine direkte Modellierung von Aktionen im 3D-Raum. Die Policy generiert Aktionssequenzen durch einen iterativen Diffusionsprozess: $a_{t+1} = \mu_\theta(a_t, s_t) + \sigma_t \epsilon_t$ wobei μ_θ das gelernte Modell und σ_t den Rauschplan darstellt. Die 3D-Struktur wird durch spezielle Verlustterme erhalten: $\mathcal{L}_{3D} = \mathcal{L}_{recon} + \alpha \mathcal{L}_{geom}$ mit \mathcal{L}_{geom} als geometrischem Regularisierungsterm [Ke-24]. Die Policy demonstriert Zero-Shot-Generalisierung auf neue Aufgaben $t_{new} \notin \mathcal{T}$ und kann über verschiedene Roboterplattformen hinweg eingesetzt werden. Diese Generalisierungsfähigkeit wird durch die Kombination von Transformer-Architekturen, äquivarianten Strukturen und 3D-spezifischen Diffusionsprozes-sen erreicht, was diese Ansätze zu vielversprechenden Methoden für generalistische Roboter-steuerung macht.

HDP Die Hierarchical Diffusion Policy (HDP) faktorisiert eine Manipulationsaufgabe in ei-ne hierarchische Struktur: einen High-Level-Agenten für die Aufgabenplanung, der eine ent-fernte, nächstbeste Endeffektor-Pose vorhersagt, und eine Low-Level-Politik für zielbedingte Diffusion, die optimale Bewegungstrajektorien erzeugt [Ma-24].

3.3 Finetunen und Verkürzen des Entrauschungs-prozesses

Bei dieser Verbesserung liegt der zentraler Fokus auf der Verkürzung der Inferenz- bzw. Trainings-Zeit. **EDP** Die Efficient Diffusion Policy (EDP) adressiert DDPM-Limitierungen durch einen innovativen Ansatz. Im Forward-Diffusionsprozess folgt eine korrumptierte Stich-probe einer vordefinierten Gauß-Verteilung $p(x_t|x_0, t)$, wobei das Rauschvorhersagenetzwerk ε_θ das zur Korruption verwendete Rauschen vorhersagt. Durch Action Approximation (Aktions-Annäherung) wird eine Aktion aus einer korrumptierten Probe konstruiert, wodurch pro Trainingsschritt nur ein einzelner Durchlauf durch das Rauschvorhersagenetzwerk erforderlich ist. Es wird also das DDPM-Training verkürzt. Dies führt zu einer 2-fachen

Beschleunigung ohne Leistungseinbußen. Die Integration des DPM-Solvers als ODE-basierter Sampler beschleunigt zusätzlich sowohl das Training als auch den Sampling-Prozess [Kan-23].

Ein beliebter Ansatz besteht darin, den Diffusionsentrauschungsprozess durch RL zu Lernen. Durch Ziele wie Belohnungssignale oder Zielkonditionierung soll der Diffusionsentrauschungsprozess gesteuert werden [[Aja-23],[Jan-22],[Lia-23],[Ven-23],[Che-24]]. In neueren Arbeiten wurden die Möglichkeiten von Techniken wie Q-Learning und gewichteter Regression erforscht, die entweder auf einer reinen offline Schätzung beruhen [[Che-24],[Din-24],[Din-24]] oder eine Online-Interaktion beinhalten [[Kan-23],[Han-23],[Han-23],[Yan-23]]. Diese grundlegende Idee wird in der **DPPO** Diffusion Policy Policy Optimization (DPPO) verwendet. DPPO stellt eine Weiterentwicklung der klassischen DP dar, die Konzepte des Policy Optimization mit dem Diffusionsprozess vereint. Dieser Ansatz ermöglicht eine effizientere und stabilere Optimierung der Handlungsstrategien in robotischen Systemen. Hier wird der Entrauschungsprozess als MDP betrachtet. Dabei ist das entfernte Rauschen die Aktion, und wie nah die Probe der entrauschten Probe entspricht ist der Reward. Damit kann ein neuronales Netzwerk trainiert werden, was den Entrauschungsprozess durchführt. Im Kern verwendet DPPO einen iterativen Prozess, bei dem die Policy nicht nur durch den Diffusionsprozess gelernt wird, sondern zusätzlich durch einen Policy-Gradienten-Ansatz optimiert wird. Dies erfolgt durch die Integration einer Wertefunktion, welche die Qualität der generierten Aktionen bewertet und zur Anpassung der Policy-Parameter verwendet wird. Die besondere Stärke von DPPO liegt in der Kombination der Explorationsfähigkeit des Diffusionsprozesses mit der zielgerichteten Optimierung des Policy-Gradient-Verfahrens. Während der Diffusionsprozess für eine breite Exploration des Aktionsraums sorgt, gewährleistet die Policy-Optimization-Komponente eine effiziente Verfeinerung der vielversprechendsten Aktionssequenzen. Ein weiterer Vorteil von DPPO ist die verbesserte Samplingeffizienz. Durch die gezielte Optimierung werden weniger Samples benötigt, um eine effektive Policy zu erlernen. Dies macht das Verfahren besonders attraktiv für reale Roboteranwendungen, bei denen die Anzahl der möglichen Trainingsiterationen oft begrenzt ist.

[Ano-24] stellen **One-Step Diffusion Policy** vor. Der vorgestellte Ein-Schritt-Aktionsgenerator generiert Sequenzen in kürzerer Inferenzzeit. Dieser Generator wird durch progressive Distillation anhand der Kullback-Leibler-(KL)Divergenz aus dem vor-trainierten Model destilliert. Das erfordert nur 2 % bis 10 % mehr Trainingszeit. Die Autoren steigern somit die Aktionsgenerierungsfrequenz von ca. 2 Hz (baseline DP) auf bis zu 62 Hz.

[Pra-24] stellen **Consistency Policy** vor. Die Autoren destillieren aus einem trainierten Modell was alle verrauschten Datenpunkte auf der Entrauschungs-Trajektorie der Diffusionsstichprobe direkt auf ihren Ursprung zurück abzubilden. Diese selbst konsistente Eigenschaft wird als Konsistenzmodell bezeichnet, da alle Datenpunkte auf derselben Trajektorie demselben Ursprung zugeordnet werden [Son-23b]. Dadurch wird die Inferenzzeit von 110 ms auf

1-2 Millisekunden verkürzt, wobei ähnliche Performanz erhalten wird.

4 Methodik

4.1 Experimentelle Infrastruktur

Die Untersuchung der Diffusion Policy wurde in einer präzise konfigurierten PyBullet-Simulationsumgebung durchgeführt. Diese Umgebung ermöglicht eine hochgradig realitätsnahe Modellierung robotischer Bewegungen und Interaktionen durch detaillierte physikalische Simulationsparameter. Das verwendete Robotermodell wurde mit spezifischen Eigenschaften implementiert, die eine umfassende aufgabenspezifische Analyse der Bewegungsgeneration erlauben. Pybullet erlaubt ähnlich zu einer Spiele-Engine das Platzieren von steuerbaren Robotern, 3D-Objekten und die Echtzeitinteraktion mit diesen. Sie wurde gewählt, da es sehr viel kostengünstiger und zeiteffektiver ist, einen virtuellen Roboter 1000 mal einen Würfel zu einer Zielposition schieben zu lassen, als einen echten Roboter. Darüber hinaus erlaubt sie die Aufzeichnung von Objekten über den Interaktionszyklus. Will man bspw. die Positionsverläufe eines Interaktionsobjektes plotten, kann man dies sehr einfach umsetzen, wohingegen das selbe in der Realität sehr viel aufwändiger wäre.

4.2 Diffusion Policy Implementierung

Die Diffusion Policy basiert auf einem probabilistischen generativen Modell, welches Bewegungssequenzen durch iterative Rauschreduktion generiert. Die zentrale Herausforderung bestand in der Weiterentwicklung unterschiedlicher Ansätze, die bspw. Inferenzzeit minimieren.

4.3 Experimentelle Methodik

Der experimentelle Ansatz folgte einem systematischen Protokoll zur schrittweisen Evaluierung und Verbesserung der Diffusion Policy. Die Datenerhebung umfasste die Generierung und Vorverarbeitung von Bewegungstrainingsdaten, wobei besonderer Wert auf Datenaugmentation und Normalisierung gelegt wurde.

4.3.1 Aufnahme von Trainingsdaten

Um die DP an eigenen Aufgaben zu trainieren, müssen eigens Trainingsdaten aufgenommen werden. Wie das geschieht, wurde im Theorie-Teil bereits erklärt, soll aber hier noch einmal genauer dargelegt werden.

1. Zunächst wird eine Trainingsepisode gestartet. Dabei ist eine Episode eine Sequenz aus Roboteraktionen, die zur Lösung der Aufgabe führt.

2. Zu jedem Zeitpunkt wird ein Bild des Roboters, sowie eine Aktion in Form eines Vektors aufgenommen und in einem Array gespeichert s. 4.1.
3. Endet eine Episode, wird das Episodenende in einem separaten Array 'Episodenlängen' gespeichert, aber Bilder und Aktionen werden weiterhin aufgenommen.
4. Hat man genügend Episoden aufgenommen, endet der Prozess.
5. Man hat nun ein fortlaufendes Bild-Array mit N Bildern (img). N: Die Länge der Demonstrationsschritte. Des Weiteren hat man ein fortlaufendes Aktions-Array (actions) mit N gesammelten Positionen, sowie ein Episode-Ende-Array der Länge E, welches alle Episodenendpunkte beinhaltet.
6. Die Bild- und Aktionsdaten werden in jeweils einer Datenreihe aufgenommen, damit sie anschließend einfacher normalisiert werden können. Hierdurch wird der Trainingsprozess stabiler.

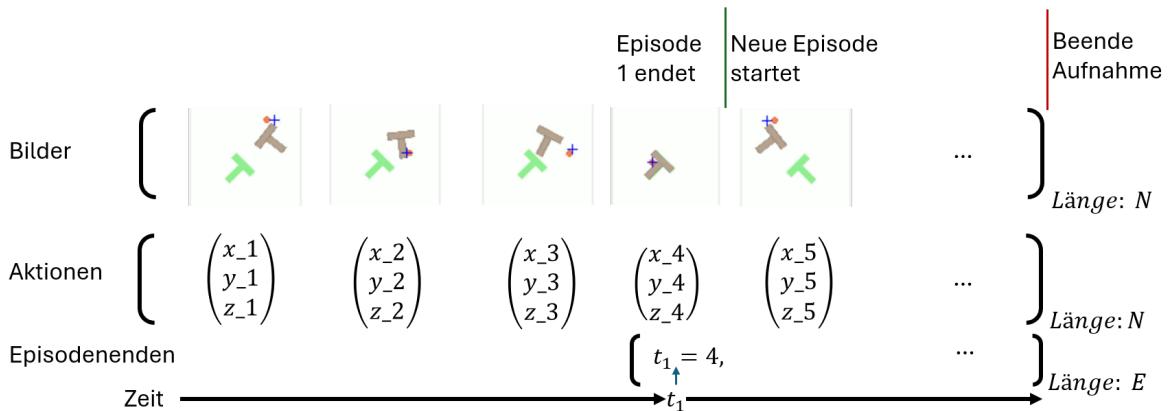


Abbildung 4.1: Prozess der Datenaufnahme

Bilder, Aktionen und Episodenenden müssen nun manuell für die Aufgabe aufgezeichnet werden. Für jede Aufgabe muss eine Roboter-Umgebung programmiert werden, welche aufgabenspezifische Daten und Funktionen definiert. Dazu gehört die Startposition des Roboters, Positionen von interaktiven Objekten, Steuerungslogik des Roboters und vieles mehr. Es ist außerdem wichtig zu definieren, was passiert, wenn eine Episode endet. Im folgenden Beispiel wird dann bspw. ein Würfel zufällig neu platziert und der Roboterarm zurückgesetzt. Die Aufnahmeeumgebung ist in Abb. 4.2 dargestellt. Der Nutzer kann nun den Roboter durch die Episoden steuern und jede Epoche mit einem Tastendruck (r) beenden und alles zurücksetzen.

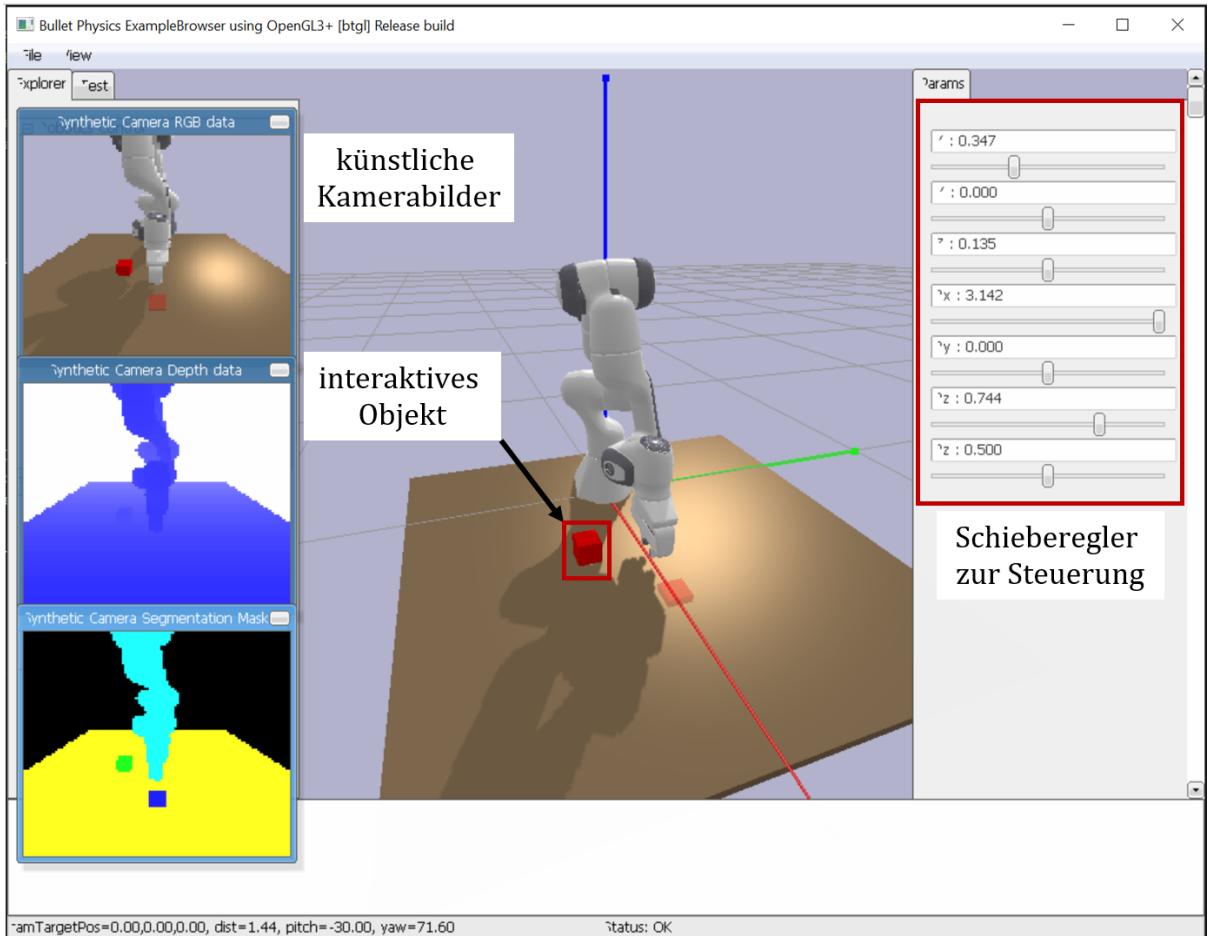


Abbildung 4.2: Aufbau der Pybullet Umgebung

4.3.2 Evaluationsschleife

Um die Qualität eines Modells für eine Aufgabe zu testen, übernimmt dieses nun die Kontrolle über den simulierten Roboter. Es bezieht dabei Umgebungsdaten und generiert entsprechend die nächsten Aktionspfade. Hierbei müssen in der Umgebung Funktionen definiert werden, welche bewerten, wann eine Aufgabe gelöst ist. Man muss außerdem festlegen, in wie vielen Schritten die Aufgabe maximal gelöst werden soll. Das soll verhindern, dass die Policy zu lange versucht eine Epoche zu lösen. Weiterhin können unterschiedliche Aufgaben unterschiedliche Bewertungsmetriken haben. Die push-T Aufgabe bewertet beispielsweise wie viel Fläche der Zielposition von dem geschobenen T verdeckt wird. Gegensätzlich lässt sich eine binäre Aufgabe als gelöst oder ungelöst beschreiben. Variiert eine Aufgabe stark, so muss die Anzahl der ausgewerteten Epochen erhöht werden. [Li-24] veranschlagen einen Durchschnittswert über 3000 Evaluationen. Weiterhin ist das Training des DDPMs selbst von zufälligen Einflussgrößen betroffen. Modelle, welche durch die selbe Trainingsschleife trainiert worden

sind, haben deshalb nicht immer exakt die selbe Leistung. Da batches zufällig aus den Trainingsdaten gewählt werden, kann es sein, dass ein Modell aus besseren Demonstrationen gelernt hat. Deshalb trainieren die Autoren von [Li-24] drei Modelle mit jeweils unterschiedlichen Seeds. Um den Einfluss der Trainingsdatenqualität auf die Erfolgsrate zu zerstreuen, benutzen die Autoren von [Chi-23] 40 % Demonstrationsdaten von 'kompetenten' Experten und 60 % Demonstrationen von mehreren 'normal kompetenten' Menschen.

4.3.3 Evaluationsframework

Die Bewertung der entwickelten Methodik erfolgte über mehrschichtige Evaluationsmetriken:

- Quantitative Analyse der Inferenzzeit-Reduktion, Erfolgsrate, Epochenzeit, Trajektorienlänge, Trainingsverluste und eigener Metriken
- Präzisionsmessung der generierten Roboterbewegungen und plotten der Bewegungsprofile

4.4 Technische Implementierung

Die technische Umsetzung basierte auf mehreren Softwarebibliotheken. Python 3.11 diente als primäre Entwicklungsumgebung, ergänzt durch PyTorch für Deep Learning, NumPy für numerische Berechnungen und PyBullet zur Robotersimulation.

4.5 Datenverarbeitung und Training

Der Trainingsprozess wurde als iterativer Zyklus konzipiert, in dem kontinuierlich Hyperparameter optimiert und Modellvarianten evaluiert wurden. Dabei erfolgte eine schrittweise Validierung der implementierten Verbesserungsstrategien durch vergleichende Analysen.

5 Verbesserungspotentiale und Implementierung

Wie in dem vorigen Kapitel gezeigt, wurden schon viele vielversprechende Verbesserungen der DP identifiziert und umgesetzt. Dennoch können weitere Verbesserungen in Hinsicht auf die Skalierbarkeit, Anpassungsfähigkeit und Inferenzzeit identifiziert werden. Dieses Kapitel stellt diese gefundenen Ansätze vor.

5.1 Trainingsdaten und Qualität

DP wird anhand von Trainingsdaten trainiert und lernt dadurch nicht von selbst. Die Qualität dieser Eingabedaten ist sehr wichtig für die spätere vorhergesagte Trajektorie. Deshalb wird in den meisten Papern zwischen Single Human Input und Multi Human Input unterschieden. DP kann in der Inferenz nur Wege vorhersagen, die ähnlich zu dem gezeigten Datensatz sind. Die Motivation, an guten Trainingsdaten zu lernen ist daher hoch. Dabei hat sich herausgestellt, dass ein möglichst diverses Training die besten Ergebnisse erzielt. Dabei sollte dieses divers in Lösungswegen, Umgebung, Farben, Formen und Wissensstand des Trainers sein. Man nehme an, die Aufgabe sei es, einen Würfel zu einer Zielposition zu schieben. In den Trainingsdaten wird nun gezeigt, wie der Roboter den Würfel von sich selbst zu der Zielposition schieben soll. Es kann nun aber vorkommen, dass der Würfel in einer Ausgangsposition liegt, die von den Trainingsdaten abweicht. Dann kann DP auch keine sinnvolle Lösung generieren. Selbiges für den Fall, dass der Roboter den Würfel aus versehen zu weit schiebt. Dies wäre dann ein Fehler. Nun kann man aber im Trainingsdatensatz auch diese Fehler und den Umgang damit zeigen. Dann kann DP auch während der Inferenz mit diesen Abweichungen umgehen. Ein guter Datensatz zeigt also auch Fehlerhandling. Außerdem sollte ein guter Datensatz idealerweise von vielen Robotern verwendet werden können. Befestigt man die Kamera am Endeffektor, können roboterunabhängige Bilddaten aufgenommen werden. Um die latenten Daten zu vervollständigen, müssen auch Robotergelenkpositionen, Geschwindigkeiten, oder EE-Positionen aufgenommen werden. Um diese so allgemein wie möglich zu gestalten, sollte die Positions-Daten relativ zur aktuellen EE-Position aufgenommen werden. D.h. anstatt die genaue Roboter EE-Position aufzuzeichnen, zeichne die nächste Position in Abhängigkeit von der vorherigen auf. Durch diese beiden Tricks können DP-gerechte skalierbare Daten aufgenommen werden. Man abstrahiert somit die Daten vom Roboter selbst und fokussiert sich nur auf die Position und Bilder aus der Endeffektorperspektive.

5.2 Potential 1: Segmentierungsmasken

Die erste Idee war es, weitere Abstraktion von Farben, Schatten und Lichtverhältnissen zu schaffen. Hierfür kann man sog. Segmentierungsmasken verwenden. Diese färbt jedes Objekt in einer Szene in einer einzigartigen, gleichbleibenden Farbe. Segmentierungsmasken bieten eine vereinfachte Darstellung der Szene auf hoher Ebene. Diese Abstraktion kann dem Modell helfen, sich auf wesentliche Informationen (Objektpositionen, Formen) zu konzentrieren, ohne durch irrelevante Details wie Texturen oder Beleuchtungsvariationen abgelenkt zu werden. Sie haben in der Regel eine geringere Dimensionalität im Vergleich zu vollständigen RGB-Bildern. Diese Verringerung der Eingabekomplexität kann die Lernaufgabe für das Modell erleichtern, was zu einer schnelleren Konvergenz und einer besseren Generalisierung führen kann. Ziel ist eine verbesserte Generalisierung zu schaffen. Durch die Konzentration auf die Form und Position von Objekten anstelle spezifischer visueller Erscheinungen können Modelle, die auf Segmentierungsmasken trainiert wurden, besser auf neue Umgebungen oder Objekte mit anderen visuellen Merkmalen, aber ähnlichen Formen, generalisiert werden. Ein großer Nachteil daran ist, dass visuelle Informationen wie Muster und Texturen verloren gehen. Sind diese jedoch wichtig für die Aufgabe, sollte keine Segmentierung verwendet werden. Für viele Manipulationsaufgaben reicht es jedoch aus, nur die primäre Farbe und Form als Information an das Modell zu geben. Zu den farbunabhängigen Aufgaben gehören: Push-T, Lift Can, Hang Tool, Push Cube, Lift Cube, Transport und Square. Die in [Li-24] erwähnte Aufgabe 'Duck-Collect' beinhaltet das Sortieren von farbigen Enten in die korrekten Container. Hier wäre eine Farbwahrnehmung also sehr wichtig. Bei einer Segmentierung sollte also darauf geachtet werden, dass die Primärfarbe des Objekts in der Segmentierungsmaske enthalten ist.

5.2.1 Implementierung

Um den potentiellen Vorteil von Segmentierung zu testen, wurden für eine Roboteraufgabe zwei Datensätze aufgenommen. Ziel ist es, einen Würfel zu einer Zielposition zu schieben. Am Anfang jeder Epoche wird der Würfel zufällig des Ausgangsbereichs $x_{range} = [0.28, 0.4]$, $y_{range} = [-0.35, 0.35]$ positioniert (vgl. Abb. 5.1). Die Aufgabe wurde mit Pybullet und dem 7 DOF Eimika Panda Roboter in einer selbst erstellten Umgebung simuliert. Die Endeffektorposition kann über Schieberegler gesteuert werden.

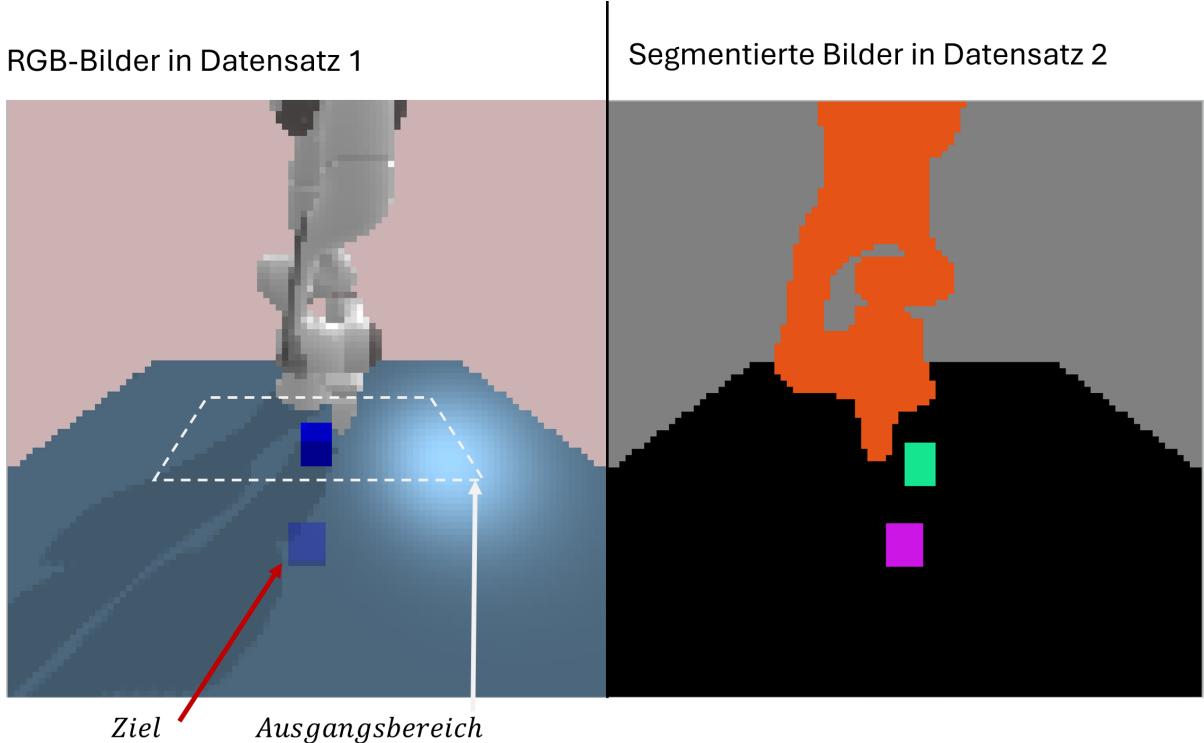


Abbildung 5.1: Aufbau des Versuchs: Schiebe Würfel zum Ziel.

5.2.1.1 Datenaufnahme

Datensatz 1 enthält RGB-Bilder und Datensatz 2 segmentierte Bilder. Anschließend wurden zwei Policies mit den jeweiligen Bildern trainiert. Um die aufgenommenen Erfolgsraten weitestgehend unabhängig von zufälligen Störgrößen zu halten, müssen viele Epochen aufgenommen werden. Jedes Modell wird deshalb für 1000 Epochen ausgeführt und dabei die Erfolgsrate und durchschnittliche Inferenzzeit, Epochenzzeit und Trajektorienlänge aufgenommen. Wenn die Lösungsepoch 240 Roboterschritte übersteigt, ohne zu einer Lösung zu kommen, zählt diese Epoche als nicht erfolgreich. Ist der Würfel während der 240 erlaubten Aktionen weniger als 0.02 Einheiten von der Zielposition entfernt, zählt das als Erfolg.

5.2.2 Ergebnisse und Interpretation

Tabelle 5.1: Gegenüberstellung der DP mit und ohne Bild-Segmentierung. 199 Trainingsepochen, Inferenzdaten aus 1000 Lösungsepochen, Abbruch nach 30 fehlerhaften Aktionsgenerierungen (240 Roboterschritte). GPU: RTX 3080

Metrik	Datensatz 1 (RGB)	Datensatz 2 (Segmentierung)
Erfolgsrate [%]	71	72.3
\varnothing Inferenzzeit [s]	0.154	0.154
Trainingsverlust	0.00513	0.00434
\varnothing Epochenzeit [s]	14.65	14.93
\varnothing Trajektorienlänge	155	150

Aus den Ergebnissen in Tab. 5.1 lässt sich erkennen, dass die Erfolgsrate durch die Segmentierung um 1.3 % zugenommen hat. Der niedrigere Trainingsverlust (0.00434) bei segmentierten Bildern könnte ein Indiz für die minimal höhere Erfolgsrate sein. Das Modell lernt, die Trainingsdaten sehr gut zu approximieren (daher der niedrige Verlust).

RGB-Bilder enthalten jedoch mehr Kontextinformationen (Schatten, Texturen, Farbverläufe), die dem Modell helfen könnten, robuster zu generalisieren und verdeckte Objekte zu erkennen. Segmentierungsmasken reduzieren hingegen die Komplexität der Eingabe, eliminieren aber damit möglicherweise wichtige visuelle Hinweise für die Policy. Die Inferenzzeiten sind bei beiden Methoden annähernd identisch. Die an segmentierten Bildern trainierte DP führt im Schnitt kürzere Trajektorien länger aus. Die langsamere Ausführung könnte auch eine Erklärung für die minimal höhere Erfolgsrate sein. Die Ergebnisse legen nahe, dass für diese spezifische Aufgabe die zusätzlichen Informationen in RGB-Bildern nicht ausschlaggebend für die Performance sind. Deshalb bietet hier eine Segmentierung eine etwas erhöhte Erfolgsrate. Es ist jedoch zu bemerken, dass die Segmentierung an sich auch fehleranfällig sein kann. Pybullet bietet den Vorteil von idealer Segmentierungsmasken. In der Realität müssten diese während der Inferenz separat berechnet werden.

Die Trajektorien beider Modelle und die Endpositionen der Würfel von gescheiterten Episoden sind in Abb. 5.2 dargestellt. Die Fehlerorte häufen sich um den demonstrierten Arbeitsbereich und den Roboter selbst. Hier sind die Würfel entweder nicht mehr zu erreichen oder dieses ist nicht demonstriert worden. Die Fehlerpositionen der segmentierten DP sind gleichmäßiger kreisförmig verteilt. Weiterhin fällt auf, dass die fehlerhaften Bahnen beider Modelle sich in den selben Bereichen häufen. Jedoch verteilen sich die roten Bahnen der segmentierten DP

über eine größere Fläche.

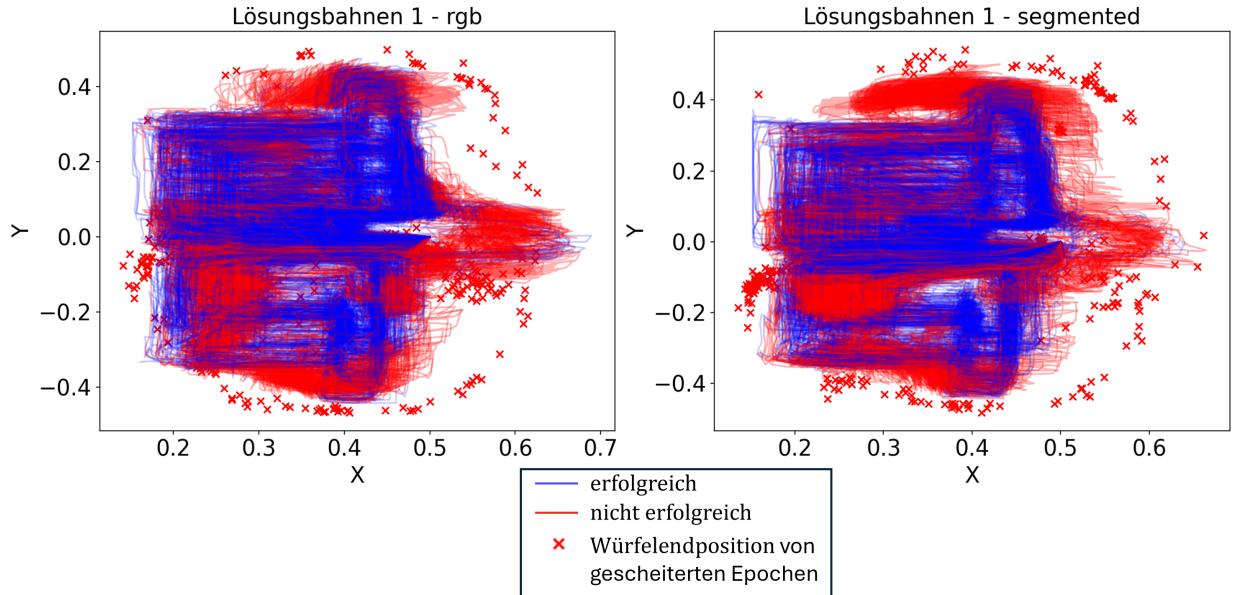


Abbildung 5.2: 1000 Trajektorien für den Versuch: Schiebe Würfel zum Ziel. Mit RGB-Bildern (links) und mit segmentierten Bildern (rechts)

5.3 Potential 2: Trainingsdaten

Das Thema der Trainingsdaten wird in vielen Papern mehr oder weniger stark übergangen. Selten findet man klare Aussagen, auf was bei der Aufnahme von Demonstrationen zu achten ist. Wie viele ideale Lösungswege sind notwendig, um eine stabile DP zu trainieren? Wie sollte man mit Fehlern umgehen? Um die Datenaufnahme besser zu gestalten, sollten Qualitätsmerkmale für das Aufnehmen von Expertendaten identifiziert werden. In diesem Kapitel wird der Einfluss von Trainingsdaten methodisch analysiert. Es soll hauptsächlich gezeigt werden, dass allein die Qualität der Trainingsdaten Einfluss auf die Erfolgsrate hat.

5.3.1 Potentiale

- 2a)** Datensatz filtern/clustern (schlechte Demonstrationen herausfiltern)
- 2b)** Datensatz prüfen (Trajektorien) und Bewertungsmetriken geben: ist die Datenmenge genug? Sind die Lösungswege divers genug? etc.
- 2c)** Datensatz-Bilder aufbereiten (Upscalen, Kanten schärfen, etc.)
- 2d)** Wenn eine Aufgabe mehrere Teilaufgaben beinhaltet, könnten diese in eigene Subaufgaben-Datensätze unterteilt werden. Danach können aus einem Datensatz zwei

Sub-DPs trainiert werden um somit die Generalisierung erhöhen.

5.3.2 Implementierung der Bewertungsmetriken (2b)

Anhand von unterschiedlichen Datensätzen sollen gute Bewertungsmetriken identifiziert werden. Zunächst wird der Trainingsdatensatz 1 betrachtet. Die Eingabemethode über die Schieberegler führt zu einem gitterartigen Trajektorienbild s. Abb. 5.3. Es fällt weiterhin auf, dass in der rechten Hälfte mehr Fehler-Korrektions-Bahnen existieren. Bei Position $[0.5, -0.35]$ jedoch kaum. Tatsächlich sind viele Epochen hier gescheitert, weil die DP den Würfel nicht wieder auf bekannte Bahnen bringen konnte.

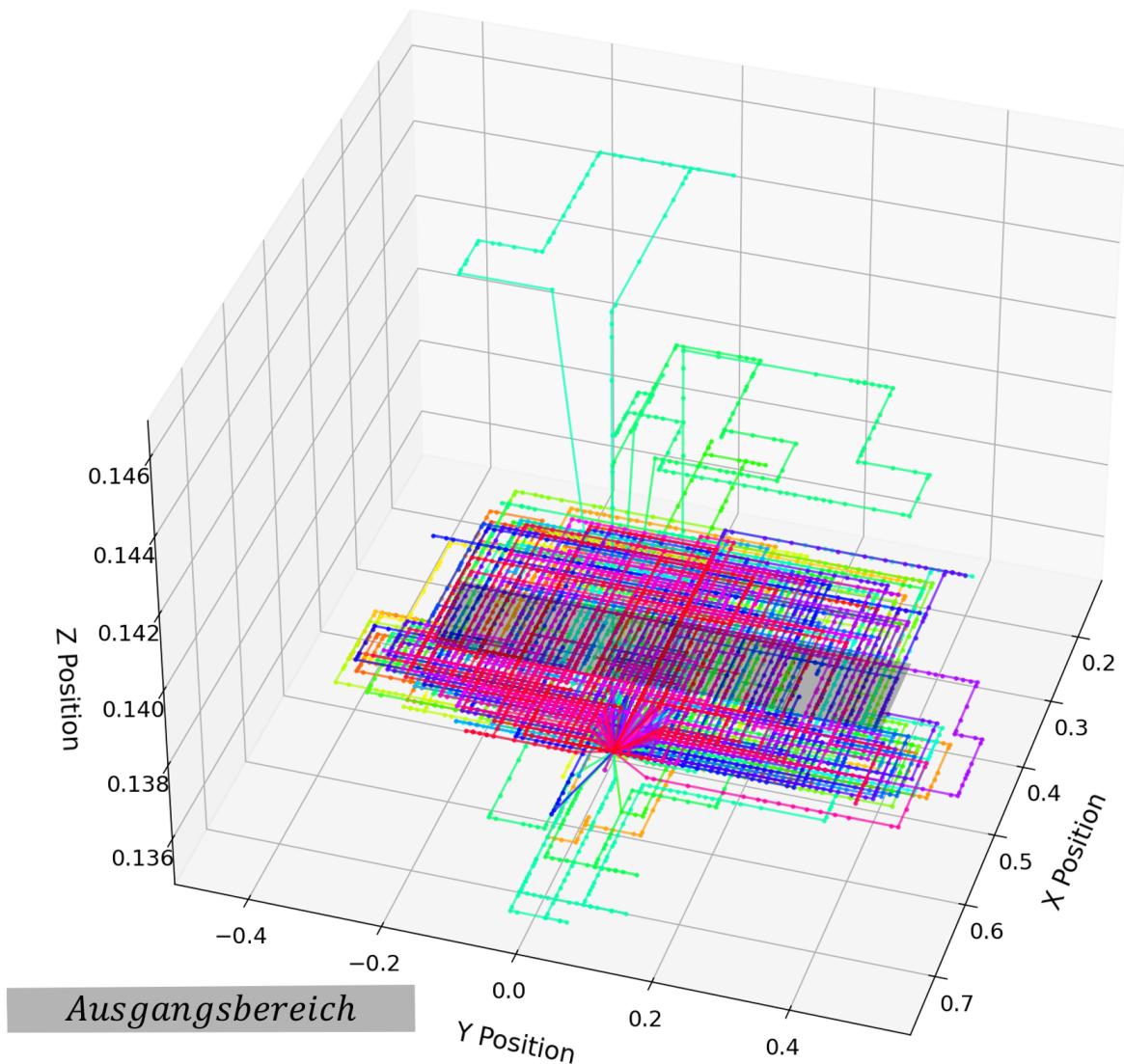


Abbildung 5.3: 199 Trainingsbahnen für den Versuch: Schiebe Würfel zum Ziel.

Zunächst werden Trajektoriendaten und Episode-Ende-Indizes sowie die Startindizes jeder Episode aus dem Datensatz bestimmt. Die folgenden Regionen werden dabei statisch definiert: **Start-Region:** Bereich, in dem die Würfel zufällig platziert werden. **Zielpunkt und Radius:** Zielbereich des Würfels kennzeichnet. Die Bewertungsmetriken werden wie folgt bestimmt:

1. **Smoothness:**

- Positionspunkte entlang einer Trajektorie sollten möglichst äquidistant sein. Sind sie zu weit von einander entfernt, deutet das auf ruckartige Bewegungen hin. Diese sollen vermieden werden. Je gleichmäßiger der Abstand zwischen Punkten, desto besser die Bewertung.

Lösung: Berechne die euklidischen Distanzen zwischen aufeinanderfolgenden Punkten. Dieser Koeffizient gibt ein Maß für die Gleichmäßigkeit der Abstände.

- 'Überschießen': Es sind Bahnen sichtbar, die darauf hindeuten, dass der Roboter über das Ziel hinaus bewegt wurde und anschließend über den selben Weg zurückbewegt worden ist. Diese Bahnen kosten Zeit und sind zu vermeiden. Das führt zu einer niedrigeren Bewertung.

Lösung: Scharfe Wendungen werden über Winkel zwischen den Bewegungsvektoren bestimmt, die durch Punktdifferenzen in der Trajektorie berechnet werden.

2. **Abdeckung:** Ist der gesamte Ausgangsbereich mit Trajektorien bedeckt?

Lösung: Hier wird überprüft, wie gut der gesamte Ausgangsbereich mit Trajektorien abgedeckt ist. Dazu wird ein Gitter über den Ausgangsbereich gelegt und für jeden Gitterpunkt die Dichte der umliegenden Trajektorien-Punkte berechnet. Je höher die durchschnittliche Dichte über alle Gitterpunkte, desto höher die Abdeckung-Bewertung.

3. **Varianz:** Sind die Trajektorien divers?

Lösung: Hierbei wird die Diversität der Trajektorien untersucht. Jede Trajektorie wird mit allen anderen verglichen und die durchschnittliche Distanz zwischen ihnen berechnet. Dies soll Auskunft über die Diversität geben.

4. **Fehlerhandlung:** Gibt es Trajektorien direkt um den Ausgangsbereich und um den Zielbereich? Das deutet auf besseres Fehlerhandling hin.

Lösung: Der Code überprüft, wie viele Punkte einer Trajektorie in den 'Gürtel'-Regionen um den Ausgangs- und Zielbereich liegen. Diese Gürtel-Regionen deuten auf ein besseres Fehlerhandling hin, da der Roboter auch in Bereichen nahe der Ziele operiert. Je mehr Punkte in den Gürtel-Regionen liegen, desto höher die Fehlerhandlung-Bewertung.

Final werden alle Metriken als Prozentwerte (0-100) normalisiert. Ein mehrteiliger Plot präsentiert (1) Trajektorienverläufe und Gürtelbereiche (Start-Region in rot und Gürtelgrenze in blau) zur Fehlerhandhabung, (2) Ein Konturplot, der die Abdeckung des Startbereichs durch die Trajektorien anzeigt, (3) Verteilung der Trajektorien-Diversität: Ein Histogramm, das die Vielfalt der Trajektorien visualisiert, (4) Durchschnittliche Bewertungen: Ein Balkendiagramm der vier Hauptmetriken, um die Gesamtbewertung darzustellen.

5.3.3 Aufnahme des besseren Datensatzes 3

Anhand der festgelegten Bewertungsmetriken (1-4) wird ein besserer Datensatz aufgenommen. Um glätttere Bewegungen zu erzielen, ist ein eigener Robotercontroller für Pybullet programmiert worden. Hiermit lässt sich der EE in x- und y-Richtung per Maus steuern. Beim Aufnehmen der Daten wurde außerdem auf verbessertes Fehlerhandling Wert gelegt. Anhand von Abb. 5.4 kann man die smoothenen Bahnen sowie das verbesserte Fehlerhandling erkennen. Hierdurch hat man mit der selben Anzahl an Demonstrationen eine bessere Abdeckung des Arbeitsbereichs. Dies geschieht mit der Hoffnung, dass die höhere Abdeckung der DP mehr Anhaltspunkte zum Lösen der Aufgabe gibt.

5.3.4 Abgeleitetes Potential: Kritische Bereiche erkennen und entsprechend handeln

eventuell nicht generalisierbar genug umzusetzen Es konnte gezeigt werden, dass smoothere Bahnen und erhöhtes Fehlerhandling zu besseren Erfolgsraten führen. Dennoch, je mehr man den Umgang mit Fehlern zeigt, desto mehr besteht auch das Potential, dass der Würfel um den Bereich des gezeigten Fehlerhandlings landet. Hier kann er wiederum nicht zurückbewegt werden. Es gibt kritische Bereiche um den gezeigten Bereich und den Zielpunkt, wo der Würfel oft liegen bleibt. Aus dieser Erkenntnis entsteht die Idee, Bahnen nahe des ungesesehenen Bereichs als kritisch zu behandeln. Hier sollten die Bewegungen langsam sein und so nah wie möglich am gesehenen Datensatz liegen. Liegt die vorhergesagte Bahn also nahe des kritischen Bereichs, könnte die Inferenzschrittzahl erhöht werden, sowie die Robotergeschwindigkeit herabgesetzt. Basierend auf diesem Konzept könnte man das abgeleitete Potential durch eine dynamische Anpassung der Inferenzparameter in der Diffusion Policy implementieren. Eine **Bahngewichtung** muss die Nähe der aktuellen Trajektorie zum kritischen Bereich berechnen. Es könnte der Vision Encoder zur Merkmalsextraktion genutzt werden, oder ein hardgedcoderter Abgleich mit den ausgelassenen Zonen. Je näher die Bahn an ungesesehenen Gebiete vorbeigeht, desto kritischer ist sie zu bewerten. Darauf basierend muss ein normalisiertes Bewertungskriterium identifiziert werden.

Im zweiten Schritt erhöht die **Dynamische Inferenzanpassung** die Diffusionsschritte in

kritischen Bereichen und reduziert die Robotergeschwindigkeit.

Eine **flexible Konfiguration** soll Basis-Diffusionsschritte und -geschwindigkeit einstellbar machen. Maximale Diffusionsschritte und minimale Geschwindigkeit sollen konfigurierbar sein. Es wird durch die adaptive Behandlung ungewohnter Bereiche eine höhere Erfolgsrate erreicht werden.

5.3.5 Ergebnisse und Interpretation

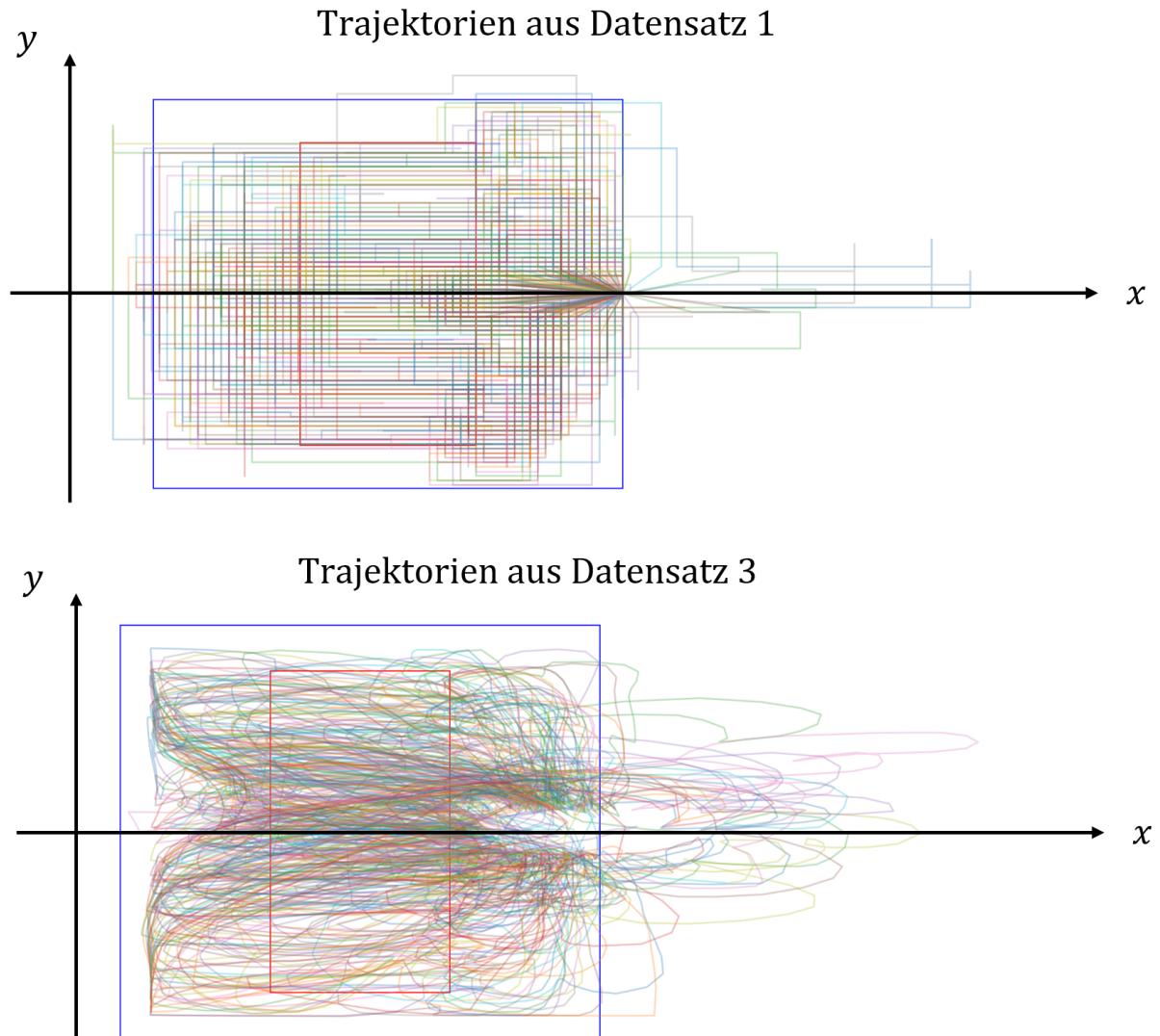


Abbildung 5.4: 199 Trajektorienverläufe aus dem Training für die zwei Datensätze. Rote Box: Die Anfangsregion der Würfel.

Zusätzlich zu den Bewertungsmetriken sind Durchschnittswerte der Inferenzzeit, Epochenzeit (Echtzeit für die Lösung einer Epoche) und Trajektorienlänge aufgezeichnet worden. Da die Scores für Varianz und Abdeckung in beiden Datensätzen gleich waren, sind sie nicht mit aufgenommen worden. Die Ergebnisse sind in Tab. 5.2 dargestellt.

Tabelle 5.2: Gegenüberstellung der DP mit 'guten' (3) und 'schlechten' Daten (1) ausgehend von den aufgestellten Bewertungsmetriken. 199 Trainingsepochen, 1000 Lösungsepochen, Abbruch nach 30 fehlerhaften Aktionsgenerierungen (240 Roboterschritte), $K_{ent}=10$, verwendete GPU: RTX 3080

Metrik	Datensatz gut	Datensatz schlecht
Erfolgsrate [%]	74.5	71
\emptyset Inferenzzeit [s]	0.201	0.154
\emptyset Epochenzeit [s]	12.86	14.93
\emptyset Trajektorienlänge	155	147
Trainingsverlust	0.00678	0.00513
Smoothness (höher ist besser)	63.2	53.8
Fehlerhandlung (höher ist besser)	78.4	72.8

Wie erwartet führt besseres Fehlerhandling sowie glattere Bahnen zu einer erhöhten Erfolgsrate (+ 3.5%) gegenüber der DP, welche mit schlechteren Daten trainiert wurde. Gleichzeitig führen die glatteren Bahnen zu kürzeren und schnelleren Lösungswegen (- 14 %). Der Vergleich mit Erfolgsraten aus Tab. 5.1 lässt darauf schließen, dass bessere Demonstrationsdaten einen positiveren Einfluss haben als die Bild-Segmentierung. Intuitiv wurde angenommen, dass eine Kombination aus beidem ein Verbesserungspotential von 4.8 % hegt. Unabhängig von Segmentierung und Qualität des Trainingsdatensatzes schieben ruckartige Bewegungen den Würfel oft in Regionen außerhalb des gesehenen trainierten Bereichs. In diesen bleibt der Würfel dann einfach liegen (s. Abb. 5.5).

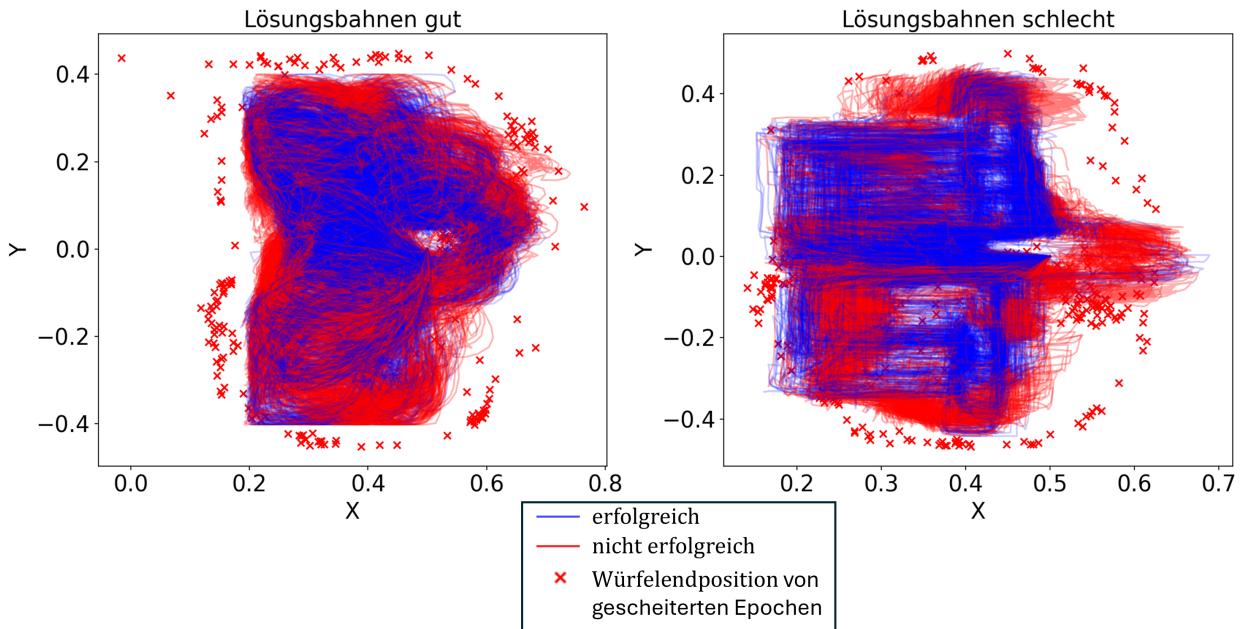


Abbildung 5.5: Vergleich der Lösungs-Trajektorienverläufe der zwei Datensätze. Datensatz 3 (links) und Datensatz 1 (rechts)

5.4 Potential 3: Inferenzgeschwindigkeit

Die Lösungszeit des gesamten Rückwärtsprozesses beeinflusst die Anwendungsbereiche der DP stark. In der Standard DP wird eine potentielle Inferenzgeschwindigkeit von 0.1 s erreicht (Bei 10 Entrauschungsschritten). In dieser Zeit können also die nächsten acht Roboterschritte berechnet werden. In der Realität zeigt sich jedoch, dass es eine Spannung zwischen Lösungszeit und Roboter geschwindigkeit gibt. Es sind theoretisch 80 Schritte = 10 Aktionssequenzen pro Sekunde möglich. Jedoch funktioniert die DP laut Autoren am besten, wenn nur alle 0.5 Sekunden eine Lösung generiert wird. Im Hauptpaper zeigte sich, dass diese Inferenzzeit zu den besten Ergebnissen führt [Chi-23]. Eine weitere Verkürzung ist aber möglich. Es muss gezeigt werden, ob diese jedoch die Erfolgsrate beeinflusst. Es kann davon ausgegangen werden, dass wegen der genannten Gründe keine Motivation bestand, die Inferenzzeit weiter zu verkürzen. In anderen Anwendungen von generativen Diffusionsmodellen wie bspw. der Bildgenerierung geht es jedoch sehr wohl um eine Reduzierung der Zeit. In diesem Kapitel werden zunächst Methoden für die Verkürzung der Inferenzzeit bei gleichbleibender Erfolgsrate vorgestellt. Es soll überprüft werden, ob eine weitere Reduzierung sinnvoll ist und inwiefern sie die Erfolgsrate beeinflusst.

Betrachtet man das zugrunde liegende Paper und seine Codebasis, so sind progressive Destillation und Konsistenzmodelle nicht Teil ihrer Implementierung. Diese Techniken wurden erst später eingeführt. Progressive Destillation wurde in „Progressive Distillation for Fast

Sampling of Diffusion Models“ (Juli 2022) eingeführt, wobei der Schwerpunkt auf der Bildzeugung lag. Konsistenzmodelle wurden noch später mit „Consistency Models“ (März 2023) eingeführt. Sowohl Konsistenzmodelle als auch Destillationsansätze haben ihre eigenen Vor- und Nachteile. Je näher man einem Inferenzschritt kommt, desto mehr neigen beiden Ansätze zu einem Bias in der Vorhersage. Der Grund für die hohe Inferenzzschrittzahl ist, dass dadurch glatte und vielfältige Lösungen probabilistisch aus dem Lösungsdatensatz generiert werden können. Durch den schrittweisen Gradientenabstieg wird die Robotersequenz sukzessive verbessert. Verringert man diesen Prozess nun auf einen Schritt, lauern neue Gefahren welche gelöst werden müssen. Dazu gehören das Aufrechterhalten von multimodalen Lösungswegen.

5.4.1 Potentiale

- 3a)** Progressive Destillation. Destilliere das DP-Modell herunter, sodass nur wenige Entrauschungsschritte notwendig sind.
- 3b)** Adaptive Schrittgrößenanpassung auf Basis des Vertrauens. Passe die Samplingschrittzahl an die Situation an.
- 3c)** Konsistenzmodelle benutzen.

5.5 Implementierung der progressiven Destillation (3a)

Dieses Kapitel beschäftigt sich mit der Umsetzung von Progressive Destillation (PD). Die Grundlagen hierfür werden in Kapitel 2.11 und 2.11 erklärt. DDPM kann Schritte überspringen, erfordert aber weiterhin mehrere Vorwärtsdurchläufe des Netzwerks. Progressive Destillation reduziert die Anzahl dieser Durchläufe und spart so Inferenzzeit. Somit erlaubt sie die Anwendungen auf Edge-Geräten oder in Echtzeit. Um den Destillationsprozess auf Diffusionsmodelle erfolgreich anwenden zu können, sind jedoch einige Vorüberlegungen notwendig. Für die Umsetzung wurden viele unterschiedliche Ansätze betrachtet. Im Folgenden sind die vielversprechendsten Konzepte aufgeführt.

5.5.1 Ansatz 3a-1 - Klassische Destillation über Inferenzschrittskalierung

Der in Kapitel 2.11 vorgestellte Ansatz nach [Sal-22] destilliert ein trainiertes Netzwerk progressiv anhand von trainer-generierten Zwischenzielen der verrauschten Probe.

5.5.1.1 Überlegungen zur Parametrisierung der Rauschvorhersage

Die übliche Parameterisierung dieses Denoising-Modells erfolgt durch direkte Vorhersage vom Rauschen ϵ mittels eines neuronalen Netzes $\hat{\epsilon}_\theta(z_t)$, was implizit $\hat{x}_\theta(z_t) = \frac{1}{\alpha_t}(z_t - \sigma_t \hat{\epsilon}_\theta(z_t))$ setzt. Der Trainingsverlust wird üblicherweise als mittlerer quadratischer Fehler im ϵ -Raum definiert:

$$L_\theta = \|\epsilon - \hat{\epsilon}_\theta(z_t)\|_2^2 = \frac{\alpha_t^2}{\sigma_t^2} \|x - \hat{x}_\theta(z_t)\|_2^2,$$

Einfach ausgedrückt wird beim normalen Training der Verlust zwischen vorhergesagtem Rauschen und eigentlichem Rauschen berechnet. Dieser kann jedoch auch als gewichteter Rekonstruktionsverlust im x -Raum (Proben-Raum) interpretiert werden - Proben-Raum meint, dass hier die Ausgangswerte wie entrauschte Bilder direkt vorhergesagt und verglichen werden. Dabei ist die Gewichtungsfunktion durch $w(\lambda_t) = \exp(\lambda_t)$ definiert, mit dem Signal-Rausch-Verhältnis (SNR) $\lambda_t = \log[\alpha_t^2/\sigma_t^2]$. Das Signal-Rausch-Verhältnis bezieht sich auf die Signalstärke im Vergleich zum Hintergrundrauschen bei verschiedenen Zeitschritten in Difusionsmodellen. Hierbei geht es um den Ausgleich der Rekonstruktionsgenauigkeit. Ziel ist es, einen Verlust zu erzeugen, der das Schülermodell über verschiedene Rauschbedingungen hinweg stabil und effektiv leitet. α_t steuert die Skalierung des Rauschens und σ_t repräsentiert die Standardabweichung des Rauschens.

Obwohl diese Standardparameterisierung gut für das Training des ursprünglichen Modells funktioniert, ist sie für die Destillation ungeeignet. Während der Destillation wird das Modell zunehmend bei niedrigeren Signal-Rausch-Verhältnissen evaluiert. Wenn $\alpha_t \rightarrow 0$, wird der Einfluss kleiner Änderungen in $\hat{\epsilon}_\theta(z_t)$ auf die Vorhersage $\hat{x}_\theta(z_t)$ stark verstärkt. Dies ist besonders problematisch, wenn die Anzahl der Sampling-Schritte reduziert wird, da Fehler nicht mehr korrigiert werden können. Bei einem einzigen Sampling-Schritt, wo das Eingaberauschen $z_t = \epsilon$ reinem Rauschen entspricht, bricht der Zusammenhang zwischen ϵ - und x -Vorhersage vollständig zusammen. Um deshalb stabile Vorhersagen zu gewährleisten schlagen die Autoren einen der folgenden Vorhersage-Parameterisierungen vor:

P1 Direkte Vorhersage von x

P2 Gemeinsame Vorhersage von x und ϵ mit getrennten Ausgabekanälen, kombiniert durch $\hat{x} = \sigma_t^2 \tilde{x}_\theta(z_t) + \alpha_t(z_t - \sigma_t \hat{\epsilon}_\theta(z_t))$

P3 Vorhersage von $v = \alpha_t \epsilon - \sigma_t x$, wodurch $\hat{x} = \alpha_t z_t - \sigma_t \hat{v}_\theta(z_t)$

Alternativ kann man das Modell bei minimal zwei Samplingschritten auswerten und somit dieses Problem elegant umgehen. Im Folgenden werden die destillierten Modelle deshalb nie auf weniger als 50 Inferenzschritte im Training destilliert und mit weniger als zwei ausgewertet.

5.5.1.2 Warum die Vorhersage in wenigen Schritten problematisch ist

In ihrem One Step Diffusion Models Paper besprechen die Autoren das Problem der deterministischen Probability-Flow ODE (vgl. Gl. 2.14) anhand einer Abb. 5.6. Sie behandeln damit ein grundlegendes Problem bei der Generierung von Daten mittels Probability-Flow ODE (Ordinary Differential Equation) in nur einem Schritt. Die Herausforderung liegt in der deterministischen Natur der Probability-Flow ODE und wie diese bei nur einem Inferenzschritt funktioniert. Trainingspfade werden durch zufällige Paarungen von Daten und Rauschen erzeugt, was eine komplexe Struktur impliziert. Bei der Generierung in nur einem Schritt tendiert das Modell dazu, auf den Mittelwert des Datensatzes zu konvergieren. Das bedeutet, die vorhergesagten Richtungen zeigen zum Zentrum der Datenpunkte. Dadurch geht die multimodale Eigenschaft der Diffusionsmodell verloren. Also die eine Eigenschaft, die sie so mächtig macht. Die Generierung wird stark in Richtung des durchschnittlichen Datenpunktes verzerrt. Bei Umfahren eines Hindernisses würde die Policy also den Mittelwert nehmen und direkt kollidieren. Mit zunehmender Reduktion der Inferenzschritte (bis hin zu einem einzigen Schritt) verstärkt sich dieser Bias. Generierte Samples werden zunehmend uninformativ und ähnlich. Die roten Kreise in der Abbildung verdeutlichen, dass bei nur einem Inferenzschritt alle generierten Samples praktisch auf den Mittelpunkt des Datensatzes fallen würden, was eine hochgradig eingeschränkte und uninteressante Generierung bedeutet. Diese Herausforderung unterstreicht die Notwendigkeit mehrschrittiger Generierungsprozesse, um die Vielfalt und Nuancen in komplexen Datensätzen angemessen zu erfassen.

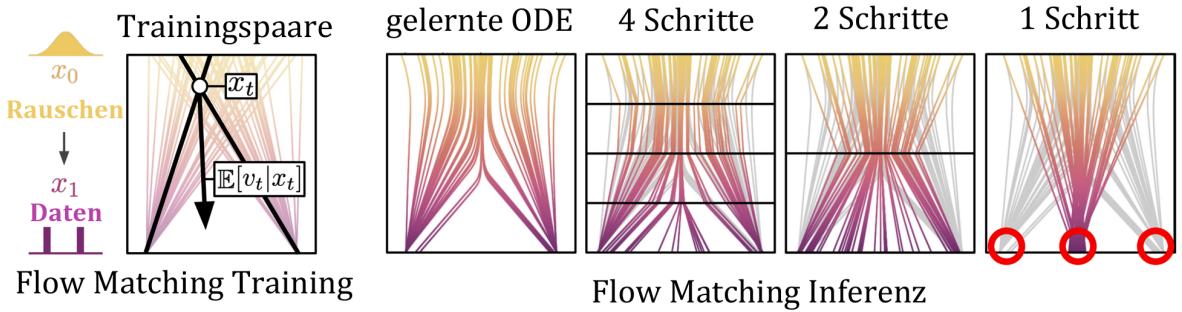


Abbildung 5.6: Naive Diffusions- und Flow-Matching-Modelle versagen bei der Generierung in wenigen Schritten. Trainings-Pfade werden durch die zufällige Paarung von Daten und Rauschen erzeugt (links). Flow-Matching Modelle lernen eine deterministische ODE - dabei sind ihre Pfade nicht gerade, sondern müssen schrittweise angepasst werden. Die vorhergesagten Richtungen je Schritt weisen auf den Durchschnitt der plausiblen Datenpunkte hin. Je weniger Inferenzschritte, desto mehr werden die Generationen in Richtung des Mittelwerts des Datensatzes verzerrt. Bei nur einem Inferenzschritt zeigt das Modell auf den Mittelwert des Datensatzes und kann daher keine multimodalen Daten erzeugen (siehe rote Kreise). Abb. angepasst nach [Ano-24].

5.5.1.3 Optimaler Rekonstruktionsverlust

In der herkömmlichen Methode hat man eine verrauschte Probe und das Rauschvorhersagenetzwerk lernt, dieses Rauschen vorherzusagen. Wie im Theorieteil beschrieben, erfolgt ein sog. Gradientenabstieg anhand einer Verlustgleichung. Das funktioniert sehr gut, um ein normales Diffusionsmodell zu trainieren. Hier versuchen wir jedoch ein bereits trainiertes Diffusions-Modell von einem anderen lernen zu lassen. Sprich wir können nicht mehr einfach nur die Trainingsdaten zu Rate ziehen. Das würde dann wieder nur ein trainiertes DDPM zur Folge haben. Wir wollen ein DDPM trainieren, was sowohl das Rauschen vorhersagt, sich aber auch an der Rauschvorhersage eines anderen DDPMs orientiert. Hierfür muss die Verlustgleichung gewichtet werden. Gewichtung in dem Zusammenhang bedeutet oft, dass mehrere Verlustterme kombiniert werden, um das neue Netzwerk zu trainieren. Da der Schüler versucht, die Lehrervorhersage und den Datensatz zu rekonstruieren, nennt man das auch Rekonstruktionsverlust. Eine geeignete Gewichtung dieses Verlusts ist ausschlaggebend für den Erfolg des Schülermodells. In [Sal-22] geben die Autoren Vorschläge für gewichtete Verlustgleichungen im Probenraum (V1, V2). Zusätzlich werden zwei weitere Verlusttherme vorgestellt, welche eigens formuliert wurden. Sie sind aus iterativem Testen und Verfeinern entstanden:

V1 - Abgeschnittenes Signal-Rausch-Verhältnis: $L_\theta = \max\left(\frac{\alpha_t^2}{\sigma_t^2}, 1\right) \|x - \hat{x}_t\|_2^2$

Verhindert die Überbetonung von frühen verrauschten Zeitschritten

Kappt das Verlustgewicht, um eine übermäßige Bestrafung zu verhindern

V2 - Signal-Rausch-Verhältnis+1: $L_\theta = \|v - \hat{v}_t\|_2^2 = (1 + \frac{\alpha_t^2}{\sigma_t^2}) \|x - \hat{x}_t\|_2^2$

Gleichmäßiger Anstieg des Verlustgewichts mit dem Rauschpegel

Verleiht den stärker beschädigten Zeitschritten mehr Bedeutung

V3 - Student-Lehrer-Rauschvorhersage-gewichtet: $L_\theta = 0.5 \cdot \|\epsilon - \epsilon_{\theta, \hat{Student}}\|_2^2 + 0.5 \cdot \|\frac{\epsilon_{\theta, \hat{Lehrer}}}{\hat{Temperatur}} - \frac{\epsilon_{\theta, \hat{Student}}}{\hat{Temperatur}}\|_2^2$

Kombiniert harten Verlust: MSE zwischen Rauschen und Studentenvorhersage und wei-

chen Verlust: MSE zwischen Studentenvorhersage und Lehrervorhersage des Rausch-
Residuals

V4 - Student-Lehrer-Probenvorhersage-gewichtet: $L_\theta = 0.5 \cdot \|x - x_{\theta, \hat{Student}}\|_2^2 + 0.5 \cdot \|\frac{x_{\theta, \hat{Lehrer}}}{\hat{Temperatur}} - \frac{x_{\theta, \hat{Student}}}{\hat{Temperatur}}\|_2^2$

Kombiniert harten Verlust: MSE zwischen Rauschen und Studentenvorhersage und wei-

chen Verlust: MSE zwischen Studentenvorhersage und Lehrervorhersage der entrauschen-
ten Probe

5.5.1.4 Anpassung des Netzwerks an die Destillationsmethode

Das Netzwerk soll nun direkt die entrauschte Probe vorhersagen, anstatt das Rauschen, welches auf diese aufgetragen wurde. Nur dann lässt sich das Modell verlässlich auf einen Schritt destillieren. Dementsprechend muss der Vorwärtsprozess des Netzwerks eine verrauschte Probe als Input nehmen und eine entrauschte Probe ausgeben. nn.Tanh() wurde als finale Faltung hinzugefügt, um optional die generierte Probe einzuschränken.

5.5.1.5 Anpassung des Trainingsprozess an die Destillationsmethode

Da unser Netzwerk nun direkt Proben vorhersagt, muss auch die Trainingsmethode geändert werden. Beim Training werden nun verrauschte Aktionen als Input in das Netzwerk gespeist. Weiterhin wurde der DDPM Scheduler von Hugging Face angepasst, um nun 'sample' anstatt 'epsilon' vorherzusagen. Der Verlustterm innerhalb der Trainingsfunktion bildet nun den MSE zwischen verrauschter Aktion und entrauschter vorhergesagter Aktion.

5.5.1.6 Erklärung des inferenzzschrittbasierten herkömmlichen Destillationsalgorithmus

Die iterative Modelkompression beginnt mit einem Lehrermodell mit hoher Anzahl von Diffusionsschritten (z.B. 100) und trainiert progressiv Schülermodelle mit halber Schrittzahl. In jeder Iteration lernt der Schüler, das Verhalten des Lehrers zu approximieren. Ziel ist die Aufrechterhaltung eines niedrigen Rekonstruktionsfehlers. Wir starten mit einem vortrainierten Lehrermodell und legen die anfängliche Schrittzahl auf 100 fest. Diese entspricht der Schrittzahl des vortrainierten Modells. Die Schrittzahl für das Schülermodell wird in jeder Iteration halbiert. Es wird der Verlust V4 verwendet. Das Modell wird solange destilliert, bis der Schüler einen Schritt erreicht. Jede Iteration erzeugt ein 'schnelleres' Modell und speichert dieses.

Algorithm 1 Progressive Destillation herkömmlich

Require: $\mathcal{D}_{\text{train}}$ ▷ Training dataset

Require: $\mathcal{M}_{\text{teacher}}$ ▷ Pre-trained teacher model

Require: $N_{\text{epoch}} \leftarrow 15$ ▷ Epochs per compression step

Require: $T_{\text{init}} \leftarrow 100$ ▷ Initial diffusion steps

```

1:  $\mathcal{M}_{\text{current}} \leftarrow \mathcal{M}_{\text{teacher}}$ 
2:  $T_{\text{current}} \leftarrow T_{\text{init}}$ 
3: while  $T_{\text{current}} \geq 1$  do
4:    $\mathcal{M}_{\text{student}} \leftarrow \text{DeepCopy}(\mathcal{M}_{\text{current}})$ 
5:   Optimizer  $\leftarrow \text{AdamW}(\mathcal{M}_{\text{student}})$ 
6:   for  $e \in 1 \dots N_{\text{epoch}}$  do
7:     for batch  $\in \mathcal{D}_{\text{train}}$  do
8:        $x \leftarrow \text{Preprocess}(\text{batch})$ 
9:        $\epsilon \leftarrow \text{SampleNoise}(x)$ 
10:       $t \leftarrow \text{SampleTimesteps}(T_{\text{current}})$ 
11:       $\hat{x}_{\text{student}} \leftarrow \mathcal{M}_{\text{student}}(x, t/2)$  ▷ Vorhersage im Probenraum
12:       $\hat{x}_{\text{teacher}} \leftarrow \mathcal{M}_{\text{current}}(x, t)$ 
13:       $L_{\text{total}} \leftarrow 0.5 \cdot \text{MSE}(\hat{x}_{\text{student}}, x) + 0.5 \cdot \text{MSE}(\hat{x}_{\text{student}}, \hat{x}_{\text{teacher}})$  ▷ V4 Verlustterm
14:      Optimizer.step( $L_{\text{total}}$ )
15:    end for
16:    LogMetrics( $L_{\text{total}}, e$ )
17:  end for
18:  SaveModel( $\mathcal{M}_{\text{student}}$ )
19:   $\mathcal{M}_{\text{current}} \leftarrow \mathcal{M}_{\text{student}}$ 
20:   $T_{\text{current}} \leftarrow \lfloor T_{\text{current}}/2 \rfloor$ 
21: end while

```

5.5.2 Ansatz 3a-2 - Alternative Destillation über Temperaturskalierung

Die Temperatur-Skalierung in der Knowledge Distillation basiert auf dem Konzept der 'softened' Wahrscheinlichkeitsverteilungen, eingeführt in der einflussreichen Arbeit von [Hin-15] 'Distilling the Knowledge in a Neural Network'. Die Temperatur-Skalierung modifiziert die Ausgabeverteilung eines neuronalen Netzes durch Division mit einer Temperatur T . Dabei wird bei einem hohen $T > 1$ die Verteilung weiter und bei $T < 1$ spitzer und konzentrierter. Höhere Temperaturen übertragen mehr weiche Informationen zwischen Modellen und niedrige eher präzise Vorhersagen. Der progressive Ansatz sieht nun vor, schrittweise die Temperatur zwischen Lehrer und Schüler zu reduzieren. Zum jetzigen Zeitpunkt wurde dieser

Ansatz für die Destillation der DP in der Literatur nicht verwendet.

5.5.2.1 Erklärung des Algorithmus

Der wirklich herausfordernde Teil ist nun, die identifizierten Ansätze in stabilen Python-Code zu überführen. Es wurden insgesamt knapp 3000 Zeilen Code verwendet und verschiedenste Lösungen ausprobiert. Dabei haben nur zwei Algorithmen das gewünschte Ergebnis erreicht und erfolgreich ein Schülermodell trainiert. Da diese beiden Codes sehr ähnlich sind, werden sie zusammen anhand des Pseudocodes 2 erklärt.

Ziel ist es, ein großes, komplexes Lehrermodell in ein kleineres, effizienteres Schülermodell zu überführen. Hierzu werden Destillationsschritte über 50 Epochen durchgeführt. Am Anfang jedes Destillationsschritts wird das Schülermodell vom Lehrermodell initialisiert. Dies geschieht durch eine Python-Klasse für Knowledge Distillation in einem neuronalen Netzwerk, die speziell für eine Student-Teacher-Architektur in PyTorch entwickelt wurde. Die Klasse StudentNetwork erbt von ConditionalUnet1D aus der Baseline. Sie ist speziell für ein Machine-Learning-Modell konzipiert, das Knowledge Distillation verwendet. Sie nimmt ein Lehrer-Netzwerk (teacher_network) als Parameter und stellt sicher, dass der Lehrer auf dem gleichen Gerät (Device) wie der Student ist. Anschließend wird der Lehrer in den Evaluationsmodus gesetzt und seine Gewichte eingefroren. Die geerbte Forward-Methode wird so angepasst, dass alle Eingaben auf dasselbe Gerät gebracht werden. Sie berechnet zwei Vorhersagen: a) Vorhersage des Studenten-Netzwerks b) Vorhersage des Lehrer-Netzwerks (ohne Gradientenberechnung) Anschließend wird die Temperatur abhängig vom Schritt bestimmt. Dadurch wird sie in jeder Destillations-Schleife herunterskaliert. Über 50 Epochen werden Batches aus dem Dataloader bezogen. Für jede Batch müssen Vision-Features und Aktionen zu einer latenten Beobachtung zusammengefügt werden. Wie in der Baseline, werden verrauschte Aktionen und Zeitschritte gesampelt. Der neu eingeführte Destillationsverlust besteht aus dem mittleren quadratischen Fehler (MSE) zwischen Student und echtem Rauschen, der gleichgewichtet wird mit dem Schüler-Lehrer-Verlust. Die aus dem angepassten Noise-Prediction-Network stammenden Vorhersagen werden durch die aktuelle Temperatur geteilt. Dieser Schritt kombiniert harte Verluste (Ground Truth) und weiche Verluste (Wissenstransfer) dynamisch über die Zeit. Der Batch-Schritt endet mit der Gradientenberechnung und Optimierung des Studenten-Netzwerks. Abschließend wird der aktuelle Lehrer aus dem aktuellen Schüler initialisiert, und ein neuer Destillationsschritt beginnt.

Algorithm 2 Progressive Destillation mit Temperatur und Schrittskalierung

```

Require:  $\mathcal{M}_{\text{teacher}}$                                 ▷ Vortrainiertes Lehrermodell
Require:  $\mathcal{D}_{\text{train}}$                                ▷ Laden der Trainingsdaten
Require:  $S_{\text{steps}}$                                  ▷ Anzahl der Destillationsschritte
Require:  $N_{\text{epoch}} \leftarrow 50$                   ▷ Epochen pro Schritt
Require:  $T_{\text{init}} \leftarrow 2.0$                    ▷ Initiale Temperatur

1:  $\mathcal{M}_{\text{current}} \leftarrow \mathcal{M}_{\text{teacher}}$ 
2: noise_scheduler.num_train_timesteps  $\leftarrow 100$ 
3: noise_scheduler.beta_schedule  $\leftarrow \text{'squaredcos_cap_v2'}$ 
4: noise_scheduler.clip_sample  $\leftarrow \text{True}$ 
5: noise_scheduler.prediction_type  $\leftarrow \text{'epsilon'}$ 
6: for  $s \in 1 \dots S_{\text{steps}}$  do
7:    $\mathcal{M}_{\text{student}} \leftarrow \mathcal{M}_{\text{current}}$            ▷ Erzeuge ein neues Studentennetzwerk
8:    $T_{\text{current}} \leftarrow \frac{T_{\text{init}}}{s+1}$ 
9:   Optimizer  $\leftarrow \text{AdamW}(\mathcal{M}_{\text{student}}.\text{parameters},$ 
    lr = 1e-4, weight_decay = 1e-6)
10:  for  $e \in 1 \dots N_{\text{epoch}}$  do
11:    for batch  $\in \mathcal{D}_{\text{train}}$  do
12:       $\mathbf{x}_{\text{cond}} \leftarrow \text{Latent Observations Vector aus Batch}$ 
13:       $\mathbf{a}_{\text{noisy}} \leftarrow \text{Zufällige verrauschte Aktion}$ 
14:       $t \leftarrow \text{SampleTimesteps}(64, [0, noise_scheduler.num_train_timesteps))$ 
15:       $\hat{\mathbf{a}}_{\text{student}}, \hat{\mathbf{a}}_{\text{teacher}} \leftarrow \mathcal{M}_{\text{student}}(\mathbf{a}_{\text{noisy}}, t/2, \mathbf{x}_{\text{cond}}),$ 
         $\mathcal{M}_{\text{current}}(\mathbf{a}_{\text{noisy}}, t, \mathbf{x}_{\text{cond}})$ 
16:       $L_{\text{total}} \leftarrow 0.5 \cdot \text{MSE}(\hat{\mathbf{a}}_{\text{student}}, \mathbf{a}_{\text{noisy}}) +$ 
         $0.5 \cdot \text{MSE}\left(\frac{\hat{\mathbf{a}}_{\text{student}}}{T_{\text{current}}}, \frac{\hat{\mathbf{a}}_{\text{teacher}}}{T_{\text{current}}}\right)$ 
17:      Optimizer.zero_grad()                           ▷ Setze Gradienten zurück
18:       $L_{\text{total}}.\text{backward}()$                    ▷ Berechne Gradienten
19:      Optimizer.step()                            ▷ Aktualisiere Parameter
20:    end for
21:  end for
22:   $\mathcal{M}_{\text{current}} \leftarrow \mathcal{M}_{\text{student}}$           ▷ Aktualisiere Lehrer mit Studentennetz
23:  SaveModel( $\mathcal{M}_{\text{current}}$ )
24: end for

```

5.5.3 Ansatz 3a-3 - Destillation der Temperatur und Inferenzschritte

Natürlich liegt es nahe, beide o.g. Ansätze zu verbinden und somit möglicherweise ein besseres Modell zu destillieren. Hierzu muss Alg. 2 abgewandelt werden, sodass zusätzlich zu der progressiven Temperaturskalierung die Schrittzahl reduziert wird. Die grundlegende Logik bleibt bestehen, allerdings beschreibt eine Konfiguration den schrittweisen Abstieg von Inferenzschrittzahl *num_train_timesteps*, Lernrate des Optimierers *lr*, Epochenzahl *epochs* und Temperatur *temperature*.

5.5.4 Ergebnisse der Tests von Ansatz 3a-2 und 3a-3

Aufgrund der iterativen Vorgehensweise ist es kaum möglich mit einem guten Wurf den perfekten Algorithmus zu entwerfen. Daher sollen hier die vielversprechendsten Ansätze gegenübergestellt werden. Ziel ist es, eine sinnvolle Methode zur weiteren Entwicklung herauszuarbeiten. Ein vortrainiertes Modell wurde als Trainermodell für die o.g. Destillations-Algorithmen verwendet. Die resultierenden Modell sollen anschließend mit nur zwei Inferenzschritten die Würfelaufgabe lösen. Die Ergebnisse werden mit dem Originalmodell verglichen.

Diese Trainings-Konfigurationen für den gemischten Ansatz 3a-3 sind in folgender Tabelle beschrieben:

Tabelle 5.3: Trainings-Konfiguration des gemischten Destillationsansatzes

Schritt	Konfig	<i>num_train_timesteps</i>	<i>temperature</i>	<i>epochs</i>	<i>lr</i>
1	1	100	2	50	10^{-4}
2	1	50	1.5	40	$8 \cdot 10^{-5}$
1	2	100	2	100	10^{-4}
2	2	100	1	80	$8 \cdot 10^{-5}$

Tabelle 5.4: Gegenüberstellung destillierten Modelle mit dem Originalmodell für die Aufgabe des Würfelschiebens. 199 Trainingsepochen, 1000 Lösungsepochen, Abbruch nach 30 fehlerhaften Aktionsgenerierungen (240 Roboterschritte), $K_{ent}=2$, verwendete GPU: RTX 3080. Dest1 wurde mit Alg. 2 mit 3 Destillationsschritten und 50 Epochen trainiert allerdings ohne die Temperatur schrittweise zu skalieren. Dest2 - config 1 wurde mit Alg. 2 trainiert mit der Konfiguration 1 aus 5.3 - analog wurde Dest2 - config 2 mit der Konfiguration 2 trainiert. Dest3 wurde in fünf Destillationsstufen à 15 Epochen durch von Alg. 1 trainiert. Die besten Werte pro Reihe sind **verstärkt** dargestellt.

Metrik	Original	Dest1	Dest2 - config 1	Dest2 - config 2	Dest3
Erfolgsrate [%]	10.1	13.6	35.8	50.2	70.2
\varnothing Inferenzzeit [s]	0.06038	0.06038	0.05827	0.05318	0.05679
\varnothing Epochenzeit [s]	14.82	19.41	17.6	13.69	12.70
\varnothing Trajektorienlänge	221	216	198	192	149
\varnothing Schritte/Sekunde	14.91	11.12	11.25	14.02	11.73
Modellgröße [mb]	356.2	981.2	981.3	981.3	356.2
Destillations-Zeit [min]	-	18.94	17.49	35.83	11.65
Destillations-Verlust	-	0.01917	0.01393	0.01298	0.00006

Erfolgsrate

Der progressive Destillationsansatz mit vermischter Temperatur- und Schrittskalierung im Rausch-Residual-Raum (Dest2) verbessern die Erfolgsrate maximal um 40.1% gegenüber dem undestillierten Originalmodell. Eine Destillation ohne Temperaturskalierung (Dest1) bietet nur eine minimale Verbesserung gegenüber der Baseline. Die herkömmliche Destillationsmethode im Probenraum zeigt hier die stärkste Erfolgsquote, welche nur 4.3 % unter dem 10-Schritt-Modell (74.5 %) liegt.

Inferenzzeit

Die Inferenzzeit ist grundsätzlich sehr ähnlich zwischen den Modellen. Sie liegt bei ca. 25% der Inferenzzeit für 10 Schritte aus den vorherigen Tests. 'Dest2 - config 2' benötigt im Durchschnitt die geringste Zeit, um die nächsten acht Aktionen vorherzusagen.

Epochenzeit, Trajektorienlänge und Geschwindigkeit

'Dest3' unterbietet mit durchschnittlich 12.7 Sekunden pro Epoche sogar das 10-Schritt-Modell. Knapp dahinter liegt 'Dest2 - config 2'. Mit einer Erhöhung der Erfolgsrate sinkt auch die durchschnittliche Epochenzzeit. Ausnahme ist nur 'Dest1'. 'Dest3' benötigt mit im Schnitt 149 die geringsten Roboterschritte, um eine Epoche abzuschließen. Sie ist fast gleichauf mit dem 10-Schritt-Modell (155). Im Gegensatz zur Epochenzzeit sinkt die Trajektorienlänge stetig mit steigender Erfolgsrate. Kombiniert man beide, lassen sich die durchschnittlichen Schritte pro Sekunde des Endeffektors bestimmen. Dabei fällt auf, dass 'Dest2 - config 2' sich schneller bewegt hat als 'Dest3'. Liegen die Aktionsschritte näher aneinander, kann der Roboter mehr vorhergesagte Aktionsschritte pro Epoche anfahren. Je höher also dieser Wert, desto ungleichmäßiger war die Vorhersage des Modells. Die Anfahrgeschwindigkeit wird auch bei einem simulierten Roboter durch die Inverse Kinematik und die Aktoren begrenzt. Schlägt das Modell also eine Zickzackroute vor, wobei jeder Schritt weit entfernt vom vorigen liegt, wird der Roboter in der begrenzten Zeit nicht all diese Punkte anfahren können.

Modellgröße

Die Modellgröße hängt mit der Anzahl von Gewichten im neuronalen Netzwerk zusammen. Dieses besteht aus einem Vision Encoder und einem Vorhersagenetzwerk. Je mehr Gewichte es hat, desto größer fällt auch die Datei aus. Demnach hängt die Modellgröße mit der Trainingsmethode und ganz speziell mit der U-net Architektur zusammen. Für die Alg. 2 Modelle ist sie gestiegen, was möglicherweise auf die Duplikierung von Gewichten zwischen den Destillationsschritten zurückzuführen ist.

Destillationszeit

Destillationszeit meint die Zeit, welche das Modell zusätzlich destilliert wurde. Grundsätzlich lässt sich sagen, dass alle Destillationsmethoden Schritte über die Trainingsdaten in Epochen gehen. Je mehr Schritte sie gehen, desto länger wird auch das zusätzliche Training ausfallen. 'Dest2 - config 2' nimmt exakt doppelt so viele Schritte wie mit 'config 1', was also zu einer Verdopplung der Trainingszeit führt.

Destillations-Verlust

Dieser bezeichnet das Ergebnis des definierten Verlustterms (V3 oder V4) zum Ende des Destillationstrainings. Er besteht zu 50 % aus dem Unterschied zwischen Student-Vorhersage und Rauschen, und zu 50 % aus Student-Vorhersage und Lehrer-Vorhersage alternativ geteilt durch einen Temperaturwert. Je geringer der Wert, desto besser hat das Schülermodell vom Datensatz und vom Lehrer gelernt. Ein Idealverlust wäre 0. Demnach hat das Modell mit dem geringsten Verlust auch die höchste Erfolgsrate. Über die vielen Trainingsschleifen hat

sich gezeigt, dass ein realistischer Wert von 0.005 meist ausreichend ist, um ein gutes Modell zu erhalten.

Implikationen

Die Methodik 'Dest3' bietet mit Abstand das höchste Potential. Sie erbringt fast durch die Bank weg bessere Leistung als die anderen Destillationsverfahren und ist in jedem Fall zu bevorzugen. Weiterhin bietet es als einziges der Modelle die Möglichkeit auch mit nur einem Schritt vernünftige Aktionen vorherzusagen. Alle anderen Methoden sagen in einem Schritt nur Rauschen vorher, da der Vorhersagezusammenhang zwischen ϵ und x wegen zu hohem Eingangsrauschen zusammenbricht. 'Dest3' umgeht dieses Problem jedoch, da es direkt im Probenraum trainiert wurde und keine Noise-Residuale vorhersagt, sondern direkt die entzerrte Probe. Die Ein-Schritt-Auswertung von 'Dest3' birgt lediglich den Nachteil, dass die gelernte Flow-Matching ODE, welche durch das Netzwerk repräsentiert wird, auf den Mittelpunkt aller Trainingsoptionen zielt und deshalb ihr multi-modales Verhalten einbüßt. Wie sich das in der Realität darstellt lässt sich am Würfelbeispiel erklären: Angenommen, der Würfel liegt direkt zwischen Roboter und Ziel. Dann ist kein multio-modales Verhalten notwendig. Der Roboter wird in einem Schritt einfach die Route direkt nach vorne vorhersagen. Nun nehme man an, der Würfel läge links versetzt zwischen Roboter und Ziel. Dann müsste der Roboter entscheiden, ob er den Würfel erst nach vorne und dann nach rechts schiebt, oder umgekehrt. Leider bildet er jetzt aber genau den Zwischenweg dieser beiden Optionen ab und schiebt den Würfel gleichzeitig nach vorne und nach rechts. Das wird dann in jedem Fall zu einem Misserfolg führen. Hätte er jedoch nun zwei Inferenzschritte gemacht, würde er eher zu einer der beiden Lösungen tendieren; weil er eben schon einen Inferenzschritt in diese Entscheidungs-Richtung gemacht hat. Kurz, bei nur einem Schritt fehlt der Policy die Entscheidungskraft und sie entscheidet sich exakt für den Mittelweg. Beschriebene Lösungsansätze verfolgten die Idee, diese Entscheidungsrichtung mit zu lernen und als Geschwindigkeitsvektor in einem Gradientenfeld vorzugeben. Die Policy soll dann den einen Schritt in Richtung dieses gelernten Vektors vornehmen. Man kann sich das so vorstellen, als ob man die Policy mit aller Kraft vom Mittelpunkt in eine der beider Entscheidungsrichtungen beschleunigt und dadurch zwingt, von diesem abzuweichen. Eine andere Idee wäre, die jetzige 50:50 Gewichtung so umzustellen, dass die Policy mehr vom Lehrer als von den ursprünglichen Daten lernt (z.B. 20:80). Dies birgt das Potential, die Ein-Schritt-Vorhersage zu verbessern. Leider könnte es aber damit die Vorhersage in mehreren Schritten verschlechtern. Das muss durch weitere Experimente nachgewiesen werden.

Der Ansatz 3a-3 schien vielversprechend, insbesondere hinsichtlich der erhöhten Erfolgsrate bei minimaler Verringerung der Inferenzeffizienz gegenüber der Baseline. Allerdings ist die angestrebte Erfolgsrate 71% noch einen großen Schritt entfernt. Möglicherweise birgt eine Anpassung der Konfiguration 5.3 weiteres Potential. Logisch wäre eine Erhöhung der Epo-

chen. Damit steigt zwar die Trainingszeit, jedoch könnte das Modell dadurch mehr Wissen vom Lehrer erhalten. Des Weiteren ist die Lernrate des Optimierers pro Schritt statisch gesetzt worden. In der Baseline wird diese jedoch stetig über das Training verringert, was einen geringeren Verlust möglich macht. Ebenso könnte die Temperatur von eingangs 2 direkt auf 1 skaliert werden, um sowohl generalisierte als auch konzentrierte Pfade zu lernen. Diese Verbesserungen wurden in Form von Konfiguration 2 aus Tab. 5.3 in einem weiteren Versuch getestet. Die Ergebnisse sind Spalte 'Dest2 - config 2' von Tab. 5.4 zu entnehmen. Wie erwartet führt die erhöhte Epochenzahl zu einer höheren Erfolgsrate. Erstaunlicherweise führt dies jedoch auch zu einer verringerten Inferenz- und Epochenzzeit sowie Trajektorienlänge. Diese Ergebnisse deuten daraufhin, dass eine erhöhte Epochenzzeit pro Destillationsstufe die Generalisierbarkeit des Modells fördert. Dies könnte jedoch auch auf die Temperaturskalierung von 2 auf 1 zurückzuführen sein. Die methodische Anpassung der Destillationsparameter für Algorithmus Alg. 2 kulminiert mit **Dest2 - config 2** zu einer um 40 % erhöhten Erfolgsrate gegenüber der Baseline bei zwei Inferenzzschritten. Allerdings muss für diese erhöhte Performanz bei zwei Schritten mehr Trainingszeit und Modellgröße investiert werden.

5.5.5 Anwendung von 3a-1 auf eine zeitkritischen Aufgabe

Die bisher implementierten Benchmarks und auch viele in der Literatur sind oft Aufgaben, welche gar keine verringerte Inferenzzeit brauchen, um gelöst zu werden. In anderen Worten eine kürzere Rechenzeit für die nächsten acht Roboterschritte ist nicht ausschlaggebend dafür, ob der Würfel nun an die richtige Position geschoben wird oder nicht. Ebenso wenig ist eine 60 Hertz Robotersteuerung notwendig, um ein T an die gewünschte Position zu schieben. Diese Aufgaben dienen gute Benchmarks, weil für sie bereits viele Vergleichsdaten aufgenommen wurden. Der Vorteil von verringelter Inferenz liegt in der verringerten Reaktionszeit. Es liegt also nahe, eine Aufgabe zu testen, wo Reaktionsgeschwindigkeit über Erfolg oder Misserfolg entscheidet. Beispielsweise soll der Roboter möglichst schnell auf Gefahren im Cobot-Raum reagieren oder ein Laufband möglichst schnell mit Waren bestücken. In solchen Aufgaben könnte die schnellere Policy sogar sehr viel höhere Erfolgsraten erzielen als die Baseline. Deshalb wird in diesem Kapitel eine zeitkritische Aufgabe erstellt, welche dann die verschiedenen Modelle lösen sollen.

5.5.5.1 Anforderungen an die zeitkritische Aufgabe

Anf1 - Zeitkritikalität: Die Aufgabe muss in einem festgelegten Zeitrahmen gelöst werden. Wird das nicht erreicht, schlägt sie fehl.

Anf2 - Multimodalität: Die Aufgabe muss multimodale Lösungswege haben. Ein einfaches Anfahren einer Zielposition ist nicht multimodal.

Anf3 - Trainierbarkeit: Die Aufgabe soll durch einen Menschen zu trainieren sein.

5.5.5.2 Notwendige Prozess-Anpassungen für die Würfel-Aufgabe

Die Würfelschiebaufgabe dient als Vorlage. Erweitert wird sie durch ein Zeitlimit. Dadurch bleibt die Multimodalität erhalten und die Aufgabe wird zeitkritisch. Für die veränderte Aufgabe müssen die folgenden Änderungen vorgenommen werden:

Anp1 - Vorhersage der Aktorenwinkel anstatt der EE-Positionen: Das Netzwerk sollte die Aktorenwinkel der Robotergelenke vorhersagen, anstatt der EE-Position. Somit wird keine zusätzliche Zeit für die inverse Kinematik verbraucht.

Anp2 - Aufnahme der Daten: Demonstrationsdaten werden nicht in Echtzeit aufgenommen. Das geschieht, damit die zeitkritische Aufgabe verlässlich demonstriert werden kann.

Anp3 - Neues Zielziel: Die Demonstrationsdaten zeigen dem Roboter, wie man in maximal fünf Sekunden Simulationszeit die Würfel in die Nähe der Zielposition schiebt. Dieser muss in der Inferenz dieses Zielziels ebenfalls unterschreiten. Nur dann gilt die Aufgabe als gelöst.

Anp4 - Erhöhung der Demonstrationszahl: Die Modelle lernen aus 400 Epochen. Weil diese nun kürzer sind, sollten mehr Episoden aufgenommen werden, sodass möglichst diverse Trainingsdaten zustande kommen.

Anp5 - Neue Gewichtung für den Destillationsprozess dest3: Der Schüler lernt im Verlustterm nun zu 80 % vom Lehrer und nur zu 20 % von den Trainingsdaten.

Anp6 - Auswertung der Modelle in nur einem Inferenzschritt: Das Baseline-, sowie das destillierte Modell inferieren in nur einem Schritt.

5.5.5.3 Ergebnisse des zeitkritischen Würfelschiebens

Die Ergebnisse des Tests für das zeitkritische Würfelschieben sind in Tab. 5.5 dargestellt. Wie vermutet, führt das destillierte Modell die zeitlich anspruchsvolle Aufgabe deutlich zuverlässiger aus. Betrachtet man nur die generierten Aktionspfade, so sind die des Originalmodells nicht glatt und gleichmäßig, sondern entsprechen wie erwartet fast reinem Rauschen was zu erratischen Roboterbewegungen führt. Die 0.6 % Erfolgsrate sind deshalb nur reinem Glück zuzuschreiben. Interessant ist weiterhin, dass die Inferenzzeit für nur einen Schritt sich kaum von der für zwei Schritte unterscheidet. In diesem einen Schritt geht ergo kaum Zeit, aber viel Lösungspotential und Multimodalität verloren. Beide Modelle haben die gleiche Größe (356,6 MB). Dies zeigt, dass die Verbesserung nicht durch eine Vergrößerung des Modells erreicht wurde, sondern durch besseres Training. Während die Verbesserung durch die Destillation

deutlich ist, zeigt die immer noch relativ niedrige Erfolgsrate von 26,5 %, dass die zeitkritische Würfelaufgabe eine echte Herausforderung darstellt. Dies könnte ein Ansatzpunkt für weitere Optimierungen sein. Es werden weitere Untersuchungen durchgeführt:

Unt1 Auswertung mit Dest3 aber mit $K_{ent}=2$

Unt2 Auswertung mit Dest3, mitt $K_{ent}=2$ und unterschiedlichen maximalen Schritt-zahlen. Bisher wurde die maximale Zahl von ausgeführten Aktionsschritten pro Sekunde auf 20 festgelegt. Diese soll nun erhöht werden und die Ergebnisse aufgezeichnet. Möglicherweise adjustiert sich das Modell somit schneller und passt die Aktionswege entsprechend an, was Fehlbewegungen frühzeitig unterbindet.

Die Ergebnisse von **Unt1** sind nun ebenfalls in der Tabelle aufgeführt. Wie erwartet leistet das weniger destillierte 3-Schritt-Modell in zwei Schritten mehr als doppelt so viel wie 'Dest3 - 1 step' bei minimal höherer Inferenzzeit.

Tabelle 5.5: Gegenüberstellung des dest3-destillierten Modells mit dem Originalmodell für die Aufgabe des zeitkritischen Würfelschiebens. 400 Trainingsepochen, 1000 Lösungsepochen, Abbruch nach 30 fehlerhaften Aktionsgenerierungen (240 Roboterschritte) oder nach 5 Sekunden, $K_{ent}=1$ bzw. 2, verwendete GPU: RTX 3080. Dest3 - 1 step wurde in sechs Destillationsstufen à 15 Epochen durch von Alg. 1 trainiert wobei der Verlustterm zu 80 % aus dem soft Teacherloss bestand; Dest3 - 2 step in fünf Destillationsstufen. Die besten Werte pro Reihe sind **verstärkt** dargestellt.

Metrik	Original	Dest3 - 1 step	Dest3 - 2 steps
Erfolgsrate [%]	0.6	26.5	57.7
\varnothing Inferenzzeit [s]	0.05340	0.05341	0.05547
\varnothing Trajektorienlänge	106	94	82
Modellgröße [mb]	356.6	356.6	356.6
Destillations-Zeit [min]	-	17.68	15.18

6 Fazit und Ausblick

Literaturverzeichnis

- [Aja-23] Ajay, A.; Du, Y.; Gupta, A.; Tenenbaum, J.; Jaakkola, T.; Agrawal, P.; *Is Conditional Generative Modeling all you need for Decision-Making?*, 2023.
- [Alm-16] Almusawi, A. R.; Dülger, L. C.; Kapucu, S.; *A new artificial neural network approach in solving inverse kinematics of robotic arm (denso vp6242)*; in *Computational intelligence and neuroscience*; 2016, 2016.
- [Ano-24] Anonymous; *One-Step Diffusion Policy: Fast Visuomotor Policies via Diffusion Distillation*; in *Submitted to The Thirteenth International Conference on Learning Representations*, 2024.
- [Arg-09] Argall, B. D.; Chernova, S.; Veloso, M.; Browning, B.; *A survey of robot learning from demonstration*; in *Robotics and Autonomous Systems*; 57(5), 2009:469–483.
- [Aru-17] Arulkumaran, K.; Deisenroth, M. P.; Brundage, M.; Bharath, A. A.; *Deep reinforcement learning: A brief survey*; in *IEEE Signal Processing Magazine*; 34(6), 2017:26–38.
- [Che-24] Chen, B.; Monso, D. M.; Du, Y.; Simchowitz, M.; Tedrake, R.; Sitzmann, V.; *Diffusion Forcing: Next-token Prediction Meets Full-Sequence Diffusion*, 2024.
- [Chi-23] Chi, C.; Feng, S.; Du, Y.; Xu, Z.; Cousineau, E.; Burchfiel, B.; Song, S.; *Diffusion Policy: Visuomotor Policy Learning via Action Diffusion*; in *Proceedings of Robotics: Science and Systems (RSS)*, 2023.
- [Din-24] Ding, Z.; Jin, C.; *Consistency Models as a Rich and Efficient Policy Class for Reinforcement Learning*, 2024.
- [Dju-16] Djuric, A. M.; Urbanic, R.; Rickli, J.; *A framework for collaborative robot (CoBot) integration in advanced manufacturing systems*; in *SAE International Journal of Materials and Manufacturing*; 9(2), 2016:457–464.
- [Fou-13] Fouz, M.; Bayoumy, A.; Rezeka, S. F.; *Neural-Networks-Based Inverse Kinematics for a Robotic Manipulator*; in *International Conference on Aerospace Sciences and Aviation Technology*; Vol. 15; The Military Technical College, 2013; 1–18.
- [Han-23] Hansen-Estruch, P.; Kostrikov, I.; Janner, M.; Kuba, J. G.; Levine, S.; *IDQL: Implicit Q-Learning as an Actor-Critic Method with Diffusion Policies*, 2023.
- [Hin-15] Hinton, G.; Vinyals, O.; Dean, J.; *Distilling the Knowledge in a Neural Network*, 2015.

- [Ho-20] Ho, J.; Jain, A.; Abbeel, P.; *Denoising Diffusion Probabilistic Models*, 2020.
- [Jan-22] Janner, M.; Du, Y.; Tenenbaum, J. B.; Levine, S.; *Planning with Diffusion for Flexible Behavior Synthesis*, 2022.
- [Kan-23] Kang, B.; Ma, X.; Du, C.; Pang, T.; Yan, S.; *Efficient Diffusion Policies for Offline Reinforcement Learning*, 2023.
- [Ke-24] Ke, T.-W.; Gkanatsios, N.; Fragkiadaki, K.; *3D Diffuser Actor: Policy Diffusion with 3D Scene Representations*, 2024.
- [Li-24] Li, X.; Belagali, V.; Shang, J.; Ryoo, M. S.; *Crossway Diffusion: Improving Diffusion-based Visuomotor Policy via Self-supervised Learning*, 2024.
- [Lia-23] Liang, Z.; Mu, Y.; Ding, M.; Ni, F.; Tomizuka, M.; Luo, P.; *AdaptDiffuser: Diffusion Models as Adaptive Self-evolving Planners*, 2023.
- [Ma-24] Ma, X.; Patidar, S.; Haughton, I.; James, S.; *Hierarchical Diffusion Policy for Kinematics-Aware Multi-Task Robotic Manipulation*, 2024.
- [Mni-15] Mnih, V.; Kavukcuoglu, K.; Silver, D.; Rusu, A. A.; Veness, J.; Bellemare, M. G.; Graves, A.; Riedmiller, M.; Fidjeland, A. K.; Ostrovski, G.; Petersen, S.; Beattie, C.; Sadik, A.; Antonoglou, I.; King, H.; Kumaran, D.; Wierstra, D.; Legg, S.; Hassabis, D.; *Human-level control through deep reinforcement learning*; in *Nature*; 518(7540), 2015:529–533.
- [Ng-00] Ng, A. Y.; Russell, S.; et al.; *Algorithms for inverse reinforcement learning.*; in *Icml*; Vol. 1, 2000; 2.
- [Nic-21] Nichol, A.; Dhariwal, P.; *Improved Denoising Diffusion Probabilistic Models*, 2021.
- [Oct-24] Octo Model Team; Ghosh, D.; Walke, H.; Pertsch, K.; Black, K.; Mees, O.; Dasari, S.; Hejna, J.; Xu, C.; Luo, J.; Kreiman, T.; Tan, Y.; Chen, L. Y.; Sanketi, P.; Vuong, Q.; Xiao, T.; Sadigh, D.; Finn, C.; Levine, S.; *Octo: An Open-Source Generalist Robot Policy*; in *Proceedings of Robotics: Science and Systems*; Delft, Netherlands, 2024.
- [Pie-23] Pierzchlewicz, P. A.; *Defusing Diffusion Models*; in *Perceptron.blog*, 2023.
- [Pom-91] Pomerleau, D. A.; *Efficient Training of Artificial Neural Networks for Autonomous Navigation*; in *Neural Computation*; 3(1), 1991:88–97.
- [Pra-24] Prasad, A.; Lin, K.; Wu, J.; Zhou, L.; Bohg, J.; *Consistency Policy: Accelerated Visuomotor Policies via Consistency Distillation*; in *Robotics: Science and Systems*, 2024.

- [RAI-22] RAIL; *Imitation learning vs. offline reinforcement learning*, 2022.
- [Ros-11] Ross, S.; Gordon, G.; Bagnell, D.; *A Reduction of Imitation Learning and Structured Prediction to No-Regret Online Learning*; in Gordon, G.; Dunson, D.; Dudík, M. (Hrsg.), *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*; Vol. 15 von *Proceedings of Machine Learning Research*; PMLR, Fort Lauderdale, FL, USA, 2011; 627–635.
- [Sal-22] Salimans, T.; Ho, J.; *Progressive Distillation for Fast Sampling of Diffusion Models*, 2022.
- [Son-21] Song, Y.; Sohl-Dickstein, J.; Kingma, D. P.; Kumar, A.; Ermon, S.; Poole, B.; *Score-Based Generative Modeling through Stochastic Differential Equations*, 2021.
- [Son-22] Song, J.; Meng, C.; Ermon, S.; *Denoising Diffusion Implicit Models*, 2022.
- [Son-23a] Song, Y.; Dhariwal, P.; *Improved Techniques for Training Consistency Models*, 2023.
- [Son-23b] Song, Y.; Dhariwal, P.; Chen, M.; Sutskever, I.; *Consistency Models*, 2023.
- [Sur-20] Surdilovic, D.; Bastidas-Cruz, A.; Haninger, K.; Heyne, P.; *Kooperation und Kollaboration mit Schwerlastrobotern–Sicherheit, Perspektive und Anwendungen*; in *Mensch-Roboter-Kollaboration*; Springer, 2020; 91–107.
- [Sut-18] Sutton, R. S.; Barto, A. G.; *Reinforcement Learning: An Introduction*; The MIT Press; second Edn., 2018.
- [Ven-23] Venkatraman, S.; Khaitan, S.; Akella, R. T.; Dolan, J.; Schneider, J.; Berseth, G.; *Reasoning with Latent Diffusion in Offline Reinforcement Learning*, 2023.
- [Wen-21] Weng, L.; *What are diffusion models?*; in *lilianweng.github.io*, 2021.
- [Yan-23] Yang, L.; Huang, Z.; Lei, F.; Zhong, Y.; Yang, Y.; Fang, C.; Wen, S.; Zhou, B.; Lin, Z.; *Policy Representation via Diffusion Probability Model for Reinforcement Learning*, 2023.
- [Yan-24] Yang, J.; ang Cao, Z.; Deng, C.; Antonova, R.; Song, S.; Bohg, J.; *EquiBot: SIM(3)-Equivariant Diffusion Policy for Generalizable and Data Efficient Learning*, 2024.
- [Zha-24] Zhang, E.; Lu, Y.; Huang, S.; Wang, W. Y.; Zhang, A.; *Language Control Diffusion: Efficiently Scaling through Space, Time, and Tasks*; in *The Twelfth International Conference on Learning Representations*, 2024.