

# Missing Data Imputation mit Variational Autoencodern

## Projektarbeit

Im Rahmen der Vorlesung  
Stochastic Machine Learning  
vorgelegt am 01. Februar 2021



Albert-Ludwigs-Universität Freiburg

Institut für Mathematik

Niklas Brunn  
Ben Deitmar  
Sebastian Hahn  
Jannis Klingler  
Clemens Schächter

# Inhaltsverzeichnis

<b>1 Einleitung</b>	<b>2</b>
<b>2 Theoretische Ausarbeitung</b>	<b>4</b>
2.1 Auto Encoding Variational Bayes . . . . .	4
2.1.1 AEVB-Algorithmus . . . . .	6
2.1.2 Variational Autoencoder . . . . .	8
2.2 ODE <sup>2</sup> VAE . . . . .	12
2.3 SDE <sup>M</sup> VAE . . . . .	19
2.3.1 Notation . . . . .	20
2.3.2 Lernen von SDEs . . . . .	23
2.3.3 Modell . . . . .	25
<b>3 Empirische Untersuchung</b>	<b>28</b>
3.1 Datensätze . . . . .	28
3.1.1 Rotating MNIST . . . . .	28
3.1.2 Bouncing Balls . . . . .	28
3.1.3 AVI-Motion . . . . .	29
3.1.4 SDE Ball . . . . .	29
3.2 Ergebnisse . . . . .	30
3.2.1 Methode 1: Lineare Interpolation im latenten Raum . . . . .	30
3.2.2 Methode 2: Vanilla-VAE . . . . .	31
3.2.3 Methode 3: ODE <sup>2</sup> VAE . . . . .	32
3.2.4 Methode 4: SDE <sup>M</sup> VAE . . . . .	34
<b>4 Modellstrukturen</b>	<b>38</b>
<b>5 Quellenangabe</b>	<b>42</b>

# 1 Einleitung

Der *Variational Autoencoder* (kurz: VAE) ist ein unsupervised Deep-Learning Algorithmus, welcher es ermöglicht niedrigdimensionale latente Darstellungen aus hochdimensionales Daten zu erlernen. Nach der Trainingsphase kann der VAE schließlich als probabilistisches Generatives Modell zum erzeugen neuer Daten verwendet werden.

Da das Modell selbst erlernen muss, wie die Datenpunkte am besten gruppiert werden, wird auch das Erkennen unbekannter Eigenschaften ermöglicht.

Beim generieren von Bildern wie menschlichen Gesichtern kann die Performance dabei durchaus mit state-of-the-art Modellen wie Generative Adversarial Networks mithalten. Die Anwendungsmöglichkeiten des Modells sind vielseitig, so werden VAE's unter anderem zum Modellieren chemischer Moleküle, zur Verarbeitung menschlicher Sprache oder zum erkennen beschädigter Daten und Schätzung von Unsicherheit eingesetzt.

Im Rahmen dieser Projektarbeit wollen wir mit Variational Autoencodern beschädigte Daten rekonstruieren und konzentrieren uns dabei ausschließlich auf Serien von Bildern mit einer zeitlichen Abhängigkeit, also Videosequenzen, oder kurz *time series*.

Da dem *Variational Autoencoder* die Annahme zugrunde liegt, dass die Verteilung der Datenpunkte unabhängig und identisch verteilt ist, ist eine Anwendung auf unsere Problemstellung nicht ohne weitere Anpassungen sinnvoll.

Ein erster naiver Ansatz hierfür ist es, die zeitliche Komponente in den Daten zu vernachlässigen und die gesamte *time series* dem VAE zusammengefasst als einen Datenvektor zu übergeben. Wenn der Trainingsdatensatz unbeschädigt ist, wird der beschädigten *time series* die latente Darstellung zugewiesen, die einer unbeschädigten *time series* am nächsten kommt. Vom Decoder wird die *time series* nun vollständig rekonstruiert, da ihm beschädigte Daten nach dem Trainingsprozess unbekannt sind.

Da die Darstellung der beschädigten Sequenz im latenten Raum jedoch relativ weit entfernt von der Darstellung der unbeschädigten Sequenz ist, sind hier die Rekonstruktionen entsprechend abweichend. Deshalb bietet es sich an das Modell auf den beschädigten Daten zu trainieren und dann mithilfe der Lossfunktion den Decoder auf die unbeschädigten Daten anzupassen.

Ein weiterer Ansatz ergibt sich, wenn man die zeitlichen Zusammenhänge der *time series* im Latenten Raum linear interpoliert. Hier bilden die einzelnen Frames der *time series* unseren Trainingsdatensatz. Durch lineare Interpolation lassen sich die Übergänge nun auch in stetiger Zeit modellieren.

Wie wir sehen werden, können beide Ansätze jedoch auf etwas komplexeren Datensätzen keine zufriedenstellende Ergebnisse mehr liefern. Eine Möglichkeit diese komplexen Zusammenhängen zwischen den Zeitschritten in *time series* besser zu erlernen bietet der

*Second Order ODE-Variational Auto-Encoder* (kurz: ODE<sup>2</sup>VAE).

Die zeitlichen Abhangigkeiten unter den Datenpunkten werden hier mit Differentialgleichungen zweiter Ordnung modelliert, welche im Trainingsprozess erlernt werden. Schon nur aus drei Inputbildern kann die gesamte *time series* in stetiger Zeit rekonstruiert werden.

Um auch 'chaotischere' *time series* lernen zu konnen, bietet es sich an mit SDEs anstatt ODEs zu arbeiten. Dabei geht allerdings die eindeutige Rekonstruierbarkeit der latenten Darstellungen aus den Anfangswerten verloren. Wir umgehen dieses Problem, indem wir die Originalen encodierten latenten Darstellungen direkt in die Encoder eingeben und mit weiteren Lossfunktionen die Darstellung als SDE erzwingen.

In dieser Arbeit werden wir in einem theoretischen Teil die mathematischen Grundlagen einfuhren, auf denen der Variational Autoencoder, der ODE<sup>2</sup>VAE, sowie der SDE<sup>M</sup>VAE aufbauen.

In einem praktischen Teil werden schlielich Anwendungsmglichkeiten der beiden Modelle auf unsere Problemstellung prasentiert.

Dabei wird sowohl auf die Implementierung der theoretischen Resultate, als auch auf die experimentellen Ergebnisse eingegangen.

Die wichtigsten in dieser Arbeit verwendeten Quellen sind die Paper *Auto-Encoding Variational Bayes* [1], sowie *An Introduction to Variational Autoencoders*, [2] beide von Diederik P. Kingma und Max Welling und *ODE<sup>2</sup>VAE: Deep generative second order ODEs with Bayesian neural networks* [3] von C. Yıldız, M. Heinonen und H. Lahdesmaki.

## 2 Theoretische Ausarbeitung

*Einführung der Notation:* Wir orientieren uns in den ersten beiden Kapiteln an den gängigen Notationen in der Machine-Learning Fachliteratur, da hier vor allem die Paper der Autoren zusammengefasst werden. Im Kapitel zu SDE-VAE werden wir uns von dieser Notation aber entfernen, um ein mathematisch genaueres Grundgerüst definieren zu können. Dieser Teil kann eigenständig gelesen werden, wir fassen die wichtigsten Ideen der ersten zwei Kapitel dort noch einmal zusammen.

Beispiel	Erklärung
$\mathbf{z}, \mathbf{x}_i$	Zufallsvariablen oder zufällig verteilte Vektoren als Realisierung einer Zufallsvariablen werden wir mit Kleinbuchstaben in fetter Schriftstärke notieren.
$p(\mathbf{z}), p(\mathbf{x}_i), p(\mathbf{x}_i \mathbf{z})$	Verteilungen/Wahrscheinlichkeitsdichten und bedingte Verteilungen bzw. Wahrscheinlichkeitsdichten dieser Zufallsvariablen. In der Notation wird nicht zwischen Dichtefunktionen zu der zugehörigen Wahrscheinlichkeitsverteilung unterschieden.
$p(.), q(.)$	Unterschiedliche Buchstaben stehen für unterschiedliche Wahrscheinlichkeitsmodelle
$\mathbb{E}_{\mathbf{z} \sim q(.)} [.]$	Erwartungswert bezüglich der angegebenen Verteilung/Dichte
$D_{KL}(q(.)  p(.))$	(Reverse) KL-Divergenz der Verteilung $q(.)$ nach $p(.)$ : $D_{KL}(q(.)  p(.)) := \mathbb{E}_{\mathbf{z} \sim q(.)} \left[ \log \left( \frac{q(.)}{p(.)} \right) \right]$
$\theta, \phi, \psi$	Parameter der betrachteten Modelle
$\mathbf{X}, \mathbf{M}, \mathbf{I}$	Zufällig verteilte Matrizen wie Datensätze notieren wir mit Großbuchstaben in fetter Schriftstärke. $\mathbf{I}$ ist die Einheitsmatrix
$\simeq$	Erwartungstreue Schätzung

### 2.1 Auto Encoding Variational Bayes

Angenommen wir haben einen Datensatz  $\mathbf{X} = \{\mathbf{x}_i\}_{i=1}^N \subset \mathbb{R}^{D \times N}$ , der aus  $N$  unterschiedlichen unabhängig, identisch verteilten Datenpunkten  $\mathbf{x}_i \sim p(\mathbf{x}_i)$  besteht, welche hochdimensionale Realisierungen einer unbekannten Zufallsvariable sind.

In einem latenten Variablen Modell sind wir darum bemüht die Dimension des Datenpunkts zu reduzieren, um den Prozess, der die Daten erzeugt besser modellieren zu können. Dafür werden sogenannte *latenten Variablen*  $\mathbf{z}$  eingeführt. Diese sind stetige und nicht beobachtbare Zufallsvariablen, welche die Eigenschaften des Bildes festlegen, die es charakte-

risiert. Der sogenannte *latente Raum*  $\mathcal{Z} = \mathbb{R}^d$ , über den die latenten Variablen  $\mathbf{z}$  definiert sind, ist nach Wahl von viel geringerer Dimension als der Raum  $\mathcal{X}$  der ursprünglichen Verteilung, d.h.  $d \ll D$ .

Das latente Variablen Modell besteht aus einem generativen Modell  $p(\mathbf{x}_i|\mathbf{z})$ , einer Priorverteilung  $p(\mathbf{z})$  und dem Inference Modell  $q(\mathbf{z}|\mathbf{x}_i)$ , welches die Posteriorverteilung  $p(\mathbf{z}|\mathbf{x}_i)$  approximiert.

Datenpunkte werden erzeugt indem zuerst eine latente Darstellung  $\mathbf{z} \sim p(\mathbf{z})$  gezogen wird, mit welcher dann ein Datenpunkt  $\mathbf{x}_i \sim p(\mathbf{x}_i|\mathbf{z})$  durch das generative Modell ermittelt wird. Nun lässt sich  $p(\mathbf{x}_i)$  über die Marginale Likelihood durch ein Integral über alle latenten Werte beschreiben.

$$p(\mathbf{x}_i) = \int_{\mathbf{z}} p(\mathbf{x}_i, \mathbf{z}) d\mathbf{z} = \int_{\mathbf{z}} p(\mathbf{x}_i|\mathbf{z})p(\mathbf{z}) d\mathbf{z}$$

In einfachen Modellen kann dieser Wert oft analytisch berechnet und maximiert werden. Wir nehmen aber an, dass dieses Integral aufgrund der hohen Dimension des Raumes  $\mathcal{X}$  und der Komplexität des Modells unberechenbar ist und auch nicht sinnvoll approximiert werden kann.

Die Posteriorverteilung  $p(\mathbf{z}|\mathbf{x}_i)$ , mit welcher sich für Datenpunkte  $\mathbf{x}_i$  eine zugehörige latente Darstellung  $\mathbf{z}$  finden lässt, ist ebenfalls nicht berechenbar. Dies und die Unberechenbarkeit der marginalen Wahrscheinlichkeit impliziert sich durch den Satz von Bayes gegenseitig:

$$p(\mathbf{z}|\mathbf{x}_i) = \frac{p(\mathbf{x}_i|\mathbf{z})p(\mathbf{z})}{p(\mathbf{x}_i)}$$

Die gemeinsame Verteilung  $p(\mathbf{x}_i, \mathbf{z}) = p(\mathbf{x}_i|\mathbf{z})p(\mathbf{z})$  lässt sich anhand der Daten approximieren, dabei sind die Datenpunkte Realisierungen dieser Verteilung. Wäre nun  $p(\mathbf{x}_i)$  beobachtbar, was nach Annahme nicht gilt, könnten wir auch  $p(\mathbf{z}|\mathbf{x}_i)$  errechnen. Eine Umkehrung dieser Überlegung gilt ebenfalls.

Die Verteilungen  $p(\mathbf{x}_i|\mathbf{z})$ ,  $p(\mathbf{z}|\mathbf{x}_i)$ ,  $p(\mathbf{z})$  und  $p(\mathbf{x}_i)$  fassen wir als durch  $\boldsymbol{\theta}$  parametisierte Familien auf. Für die optimalen Parameter  $\boldsymbol{\theta}^*$  soll die Verteilung  $p_{\boldsymbol{\theta}^*}(\mathbf{x}_i)$  die wahre Verteilung  $p(\mathbf{x}_i)$  möglichst gut approximieren.

Weiter führen wir wegen der Unberechenbarkeit von  $p_{\boldsymbol{\theta}}(\mathbf{z}|\mathbf{x}_i)$  das Inferenzmodell  $q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x}_i)$  ein und suchen die Parameter  $\boldsymbol{\phi}^*$  für die  $q_{\boldsymbol{\phi}^*}(\mathbf{z}|\mathbf{x}_i) \approx p_{\boldsymbol{\theta}}(\mathbf{z}|\mathbf{x}_i)$  gilt.

In einem Variational Autoencoder werden die Verteilungen  $q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})$  und  $p_{\boldsymbol{\theta}}(\mathbf{x}_i|\mathbf{z})$  durch neuronale Netze dargestellt und approximiert.

Im AEVB-Algorithmus (kurz für *Auto-Encoding Variational Bayes*) können die geeigneten Parameter  $\boldsymbol{\phi}^*$  und  $\boldsymbol{\theta}^*$  gemeinsam erlernt werden.

### 2.1.1 Der AEVB-Algorithmus

Wir betrachten dafür für einen beliebigen Datenpunkt  $\mathbf{x}_i \in \mathbf{X}$  die Reverse KL-Divergenz des Infernce Modells  $q_\phi(\mathbf{z}|\mathbf{x}_i)$  nach  $p_\theta(\mathbf{z})$ :

$$\begin{aligned} D_{KL}[q_\phi(\mathbf{z}|\mathbf{x}_i) || p_\theta(\mathbf{z}|\mathbf{x}_i)] &\stackrel{\text{Def.}}{=} \mathbb{E}_{\mathbf{z} \sim q_\phi} \left[ \log \left( \frac{q_\phi(\mathbf{z}|\mathbf{x}_i)}{p_\theta(\mathbf{z}|\mathbf{x}_i)} \right) \right] \\ &\stackrel{\text{Bayes}}{=} \mathbb{E}_{\mathbf{z} \sim q_\phi} \left[ \log \left( \frac{p_\theta(\mathbf{x}_i) q_\phi(\mathbf{z}|\mathbf{x}_i)}{p_\theta(\mathbf{x}_i|\mathbf{z}) p_\theta(\mathbf{z})} \right) \right] \\ &= \mathbb{E}_{\mathbf{z} \sim q_\phi} \left[ \log(p_\theta(\mathbf{x}_i)) + \log \left( \frac{q_\phi(\mathbf{z}|\mathbf{x}_i)}{p_\theta(\mathbf{z})} \right) - \log(p_\theta(\mathbf{x}_i|\mathbf{z})) \right] \\ &= \log(p_\theta(\mathbf{x}_i)) + D_{KL}[q_\phi(\mathbf{z}|\mathbf{x}_i) || p_\theta(\mathbf{z})] - \mathbb{E}_{\mathbf{z} \sim q_\phi} [\log(p_\theta(\mathbf{x}_i|\mathbf{z}))] \end{aligned}$$

Durch Äquivalenzumformung erhalten wir schließlich:

$$\log(p_\theta(\mathbf{x}_i)) - \underbrace{D_{KL}[q_\phi(\mathbf{z}|\mathbf{x}_i) || p_\theta(\mathbf{z}|\mathbf{x}_i)]}_{\geq 0} = \underbrace{\mathbb{E}_{\mathbf{z} \sim q_\phi} [\log(p_\theta(\mathbf{x}_i|\mathbf{z}))]}_{=: \text{ELBO}} - D_{KL}[q_\phi(\mathbf{z}|\mathbf{x}_i) || p_\theta(\mathbf{z})] = -\mathcal{L}(\theta, \phi, \mathbf{x}_i)$$

Die rechte Seite wird auch *Evidence Lower Bound* (kurz: ELBO) genannt und dient als (negative) Verlustfunktion des Variational Autoencoder. Als untere Schranke der Loglikelihood führt eine Maximierung der ELBO dazu, dass zum einen die Reverse KL-Divergenz der Approximation zur wahren Verteilung verringert wird, was eine immer besser werdennde Beschreibung von  $p_\theta(\mathbf{z}|\mathbf{x}_i)$  durch  $q_\phi(\mathbf{z}|\mathbf{x}_i)$  impliziert und zum anderen die Likelihood Daten des Datensatzes zu erzeugen groß wird.

Wir haben in  $\mathcal{L}(\theta, \phi, \mathbf{x}_i)$  also eine zur Minimierung geeignete Funktion gefunden und wollen einen geeigneten Algorithmus verwenden, der uns das Finden der optimalen Parameter

$$\theta^*, \phi^* = \arg \min_{\theta, \phi} \mathcal{L}(\theta, \phi, \mathbf{x}_i).$$

in einer angemessenen Berechnungsdauer ermöglicht. Dazu müssen während des Lernprozesses die Gradienten bezüglich der Parameter berechnet werden.

$$\nabla_{\theta, \phi} \mathcal{L}(\theta, \phi, \mathbf{x}_i) = \nabla_{\theta, \phi} \left[ D_{KL}[q_\phi(\mathbf{z}|\mathbf{x}_i) || p_\theta(\mathbf{z})] \right] - \nabla_{\theta, \phi} \left[ \mathbb{E}_{\mathbf{z} \sim q_\phi} [\log(p_\theta(\mathbf{x}_i|\mathbf{z}))] \right]$$

Die KL-Divergenz ist in vielen Fällen mit analytischen Methoden berechen- und differenzierbar, genauer betrachtet werden muss aber der Gradient des Erwartungswertes bezüglich  $\phi$ .

Gegeben  $\mathbf{x}_i$  ist die latente Darstellung  $\mathbf{z}$  kein deterministisch festgelegter Wert, sondern wird durch die von diesen Parametern abhängige Verteilung  $\mathbf{z} \sim q_\phi(\mathbf{z}|\mathbf{x}_i)$  bestimmt. Da der Gradient später durch Backpropagation berechnet wird, müssen auch die Layer durch-

schriften werden, die diesen Zufallsprozess enthalten. Monte-Carlo Schätzungen sind außerdem nicht immer mit zufriedenstellender Genauigkeit möglich, denn der Gradient des Erwartungswertes lässt sich in der Regel nicht zu einem neuen Erwartungswert umschreiben, da die Dichtefunktion von  $\phi$  abhängig ist.

$$\begin{aligned}\nabla_{\phi} \left[ \mathbb{E}_{\mathbf{z} \sim q_{\phi}} [\log (p_{\theta}(\mathbf{x}_i | \mathbf{z}))] \right] &= \nabla_{\phi} \int_{\mathbf{z}} q_{\phi}(\mathbf{z} | \mathbf{x}_i) \log (p_{\theta}(\mathbf{x}_i | \mathbf{z})) d\mathbf{z} \\ &= \int_{\mathbf{z}} \nabla_{\phi} q_{\phi}(\mathbf{z} | \mathbf{x}_i) \log (p_{\theta}(\mathbf{x}_i | \mathbf{z})) d\mathbf{z}\end{aligned}$$

Der *Reparametrisierungs Trick* bietet eine Lösung für diese Problematik. [4] Wir reparametrisieren die latente Variable  $\mathbf{z} = g_{\phi}(\boldsymbol{\epsilon}, \mathbf{x}_i)$  mit einer differenzierbaren, durch  $\phi$  parametrisierten Funktion  $g$  und einer Zufallsvariablen  $\boldsymbol{\epsilon} \sim p(\boldsymbol{\epsilon})$ , deren Dichtefunktion jedoch nicht von  $\phi$  abhängig ist. Somit erhalten wir nach Anwendung

$$\begin{aligned}\nabla_{\phi} \left[ \mathbb{E}_{\mathbf{z} \sim q_{\phi}} [\log (p_{\theta}(\mathbf{x}_i | \mathbf{z}))] \right] &= \nabla_{\phi} \left[ \mathbb{E}_{\boldsymbol{\epsilon} \sim p(\boldsymbol{\epsilon})} [\log (p_{\theta}(\mathbf{x}_i | g_{\phi}(\boldsymbol{\epsilon}, \mathbf{x}_i)))] \right] \\ &= \nabla_{\phi} \int_{\boldsymbol{\epsilon}} p(\boldsymbol{\epsilon}) \log (p_{\theta}(\mathbf{x}_i | g_{\phi}(\boldsymbol{\epsilon}, \mathbf{x}_i))) d\boldsymbol{\epsilon} \\ &= \int_{\boldsymbol{\epsilon}} p(\boldsymbol{\epsilon}) \nabla_{\phi} \log (p_{\theta}(\mathbf{x}_i | g_{\phi}(\boldsymbol{\epsilon}, \mathbf{x}_i))) d\boldsymbol{\epsilon} \\ &= \mathbb{E}_{\boldsymbol{\epsilon} \sim p(\boldsymbol{\epsilon})} [\nabla_{\phi} \log (p_{\theta}(\mathbf{x}_i | g_{\phi}(\boldsymbol{\epsilon}, \mathbf{x}_i)))] \\ &\stackrel{\text{M.C.}}{\simeq} \frac{1}{L} \sum_{l=1}^L \nabla_{\phi} \log (p_{\theta}(\mathbf{x}_i | g_{\phi}(\boldsymbol{\epsilon}_l, \mathbf{x}_i))).\end{aligned}$$

Was eine Berechnung der Gradienten, sowie Monte-Carlo-Schätzung des Erwartungswertes ermöglicht.

Die negative ELBO soll in unserem gesuchten Algorithmus nicht nur für einen einzigen Datenpunkt  $\mathbf{x}_i$ , sondern vielmehr auf dem gesamten Datensatz  $\mathbf{X}$ , also für alle  $\mathbf{x}_i$  minimiert werden, denn wir haben

$$\begin{aligned}\log (p_{\theta}(\mathbf{X})) &= \log (p_{\theta}(\mathbf{x}_1, \dots, \mathbf{x}_N)) \stackrel{\text{i.i.d.}}{=} \log \left( \prod_{i=1}^N p_{\theta}(\mathbf{x}_i) \right) = \sum_{i=1}^N \log (p_{\theta}(\mathbf{x}_i)) \\ &= \sum_{i=1}^N D_{KL} [q_{\phi}(\mathbf{z} | \mathbf{x}_i) || p_{\theta}(\mathbf{z} | \mathbf{x}_i)] - \mathcal{L}(\boldsymbol{\theta}, \phi, \mathbf{x}_i).\end{aligned}$$

Der kürzeren Berechnungsdauer geschuldet, werden dafür Minibatches  $\mathbf{M} = \{\mathbf{x}_i\}_{i=1}^M \subset \mathbf{X}$  gezogen und die ELBO nur auf diesen optimiert.

$$\mathcal{L}(\boldsymbol{\theta}, \phi, \mathbf{X}) \simeq \frac{N}{M} \mathcal{L}(\boldsymbol{\theta}, \phi, \mathbf{M}) = N \mathbb{E}[\mathcal{L}(\boldsymbol{\theta}, \phi, \mathbf{x}_i)] \simeq \frac{N}{M} \sum_{i=1}^M \mathcal{L}(\boldsymbol{\theta}, \phi, \mathbf{x}_i)$$

$$\simeq \frac{N}{M} \sum_{i=1}^M \left( D_{KL} [q_\phi(\mathbf{z}|\mathbf{x}_i) || p_\theta(\mathbf{z})] - \frac{1}{L} \sum_{l=1}^L \log (p_\theta(\mathbf{x}_i | g_\phi(\boldsymbol{\epsilon}_l, \mathbf{x}_i))) \right)$$

Praktischerweise sorgt eine groß gewählte Batchsize  $|\mathbf{M}| = M$ , z.B.  $M = 100$  dafür, dass der Erwartungswert implizit durch die Batches des Stochastich Gradient Verfahrens geschätzt wird.

Wir können dann die Anzahl der Stichproben von  $\boldsymbol{\epsilon}$  für die Monte-Carlo Schätzung des Erwartungswertes  $\mathbb{E}_{\boldsymbol{\epsilon} \sim p(\boldsymbol{\epsilon})} [\log (p_\theta(\mathbf{x}_i | g_\phi(\boldsymbol{\epsilon}, \mathbf{x}_i)))]$  als  $L = 1$  festlegen und erhalten den AEVB-Algorithmus (*Auto Encoding Variational Bayes*):

---

### **AEVB-Algorithmus**

---

- 1)  $\boldsymbol{\phi}, \boldsymbol{\theta} \leftarrow$  Ziehe die Parameter zufällig.
  - 2)  $\mathbf{M} \leftarrow$  Ziehe ein Minibatch  $\mathbf{M} = \{\mathbf{x}_i\}_{i=1}^M$  mit  $|\mathbf{M}| \geq 100$ .
  - 3)  $\boldsymbol{\epsilon} \leftarrow$  Ziehe einen Wert für  $\boldsymbol{\epsilon} \sim p(\boldsymbol{\epsilon})$ .
  - 4)  $\mathbf{g} \leftarrow$  Berechne die Gradienten  $\nabla_{\boldsymbol{\theta}, \boldsymbol{\phi}} \mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\phi}, \mathbf{M})$ .
  - 5)  $\boldsymbol{\phi}, \boldsymbol{\theta} \leftarrow$  Aktualisiere die Parameter.
  - 6) Wiederhole ab Schritt 2), bis zur Konvergenz des Verfahrens.
- 

#### **2.1.2 Der Variational Autoencoder**

Beim Variational Autoencoder werden die Verteilungen  $q_\phi(\mathbf{z}|\mathbf{x}_i)$  und  $p_\theta(\mathbf{x}_i|\mathbf{z})$  durch Neuronale Netze mit den Parametern  $\boldsymbol{\theta}$  und  $\boldsymbol{\phi}$  approximiert und durch den AEVB-Algorithmus trainiert. In diesem Kontext wird  $p_\theta(\mathbf{x}_i|\mathbf{z})$  auch *probabilistischer Decoder* genannt und  $q_\phi(\mathbf{z}|\mathbf{x}_i)$  *probabilistischer Encoder*.

Nach der Trainingsphase können Decoder und Encoder jedoch voneinander getrennt werden. Mit dem Decoder lassen sich nun neue Daten erzeugen, indem hypothetische latente Darstellungen als Input genutzt werden.

Für das Inferenzmodell  $q_\phi(\mathbf{z}|\mathbf{x}_i)$  wird in einem Variational Autoencoder häufig (aber nicht zwingenderweise) eine Multivariate Normalverteilung  $\mathcal{N}(\boldsymbol{\mu}_i, \boldsymbol{\sigma}_i^2 \mathbf{I})$  mit diagonaler Kovarianzmatrix gewählt.

Hier bildet der Encoder einen Input  $\mathbf{x}_i$  auf den Lageparameter  $\boldsymbol{\mu}_i$  und den Streuungsparameter  $\boldsymbol{\sigma}_i$  ab. Diese Parameter sind deterministische Funktionswerte des Encodernetzwerkes, es gilt also  $\boldsymbol{\mu}_i = \boldsymbol{\mu}_\phi(\mathbf{x}_i)$  und  $\boldsymbol{\sigma}_i = \boldsymbol{\sigma}_\phi(\mathbf{x}_i)$ .

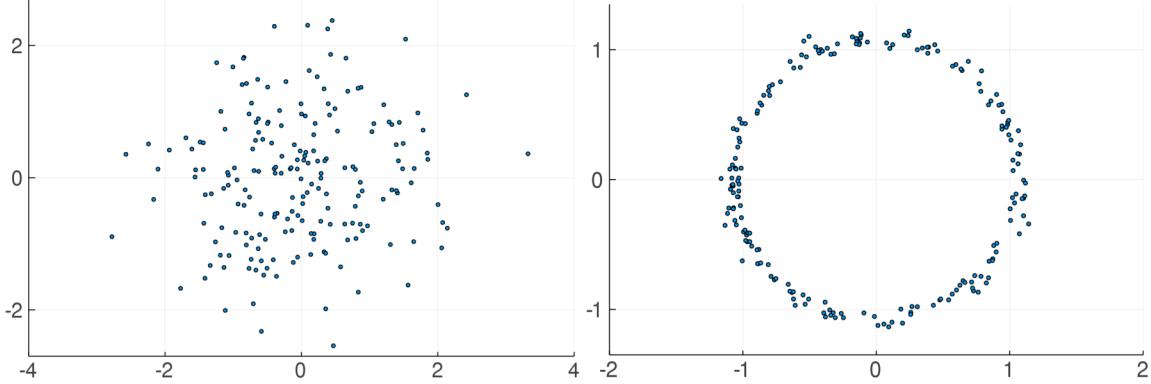
Auf diesen Output wird der Reparametrisation Trick angewandt, um die Berechnung des Gradienten mit Backpropagation zu ermöglichen.

Gegeben ein Datenpunkt  $\mathbf{x}_i$  erhalten wir eine latente Darstellung deshalb durch  $\mathbf{z} = g_\phi(\boldsymbol{\epsilon}, \mathbf{x}_i) = \boldsymbol{\mu}_\phi(\mathbf{x}_i) + \boldsymbol{\sigma}_\phi(\mathbf{x}_i) \odot \boldsymbol{\epsilon}$  mit  $\boldsymbol{\epsilon} \sim p(\boldsymbol{\epsilon}) = \mathcal{N}(\mathbf{0}, \mathbf{I})$ . Hierbei steht  $\odot$  für eine elementweise Multiplikation der beiden Vektoren.

Die Verteilung  $p_\theta(\mathbf{z})$  wird als multivariate Standardnormalverteilung  $\mathcal{N}(\mathbf{0}, \mathbf{I})$  festgelegt

und nicht durch ein Neuronales Netz dargestellt. Interessanterweise ist diese Wahl einer parameterfreien, recht einfachen Verteilung möglich. Es lässt sich durch Abbilden von Realisierungen einer multivariat standardnormalverteilten Zufallsvariable tatsächlich eine beliebige andere Verteilung erzeugen.

Werden komplexere Priorverteilungen gesucht, lassen sich simple Priorverteilungen auch durch erlernbare, bijektive Transformationen verändern. Diese Theorie übersteigt im Umfang aber den Rahmen der Projektarbeit.



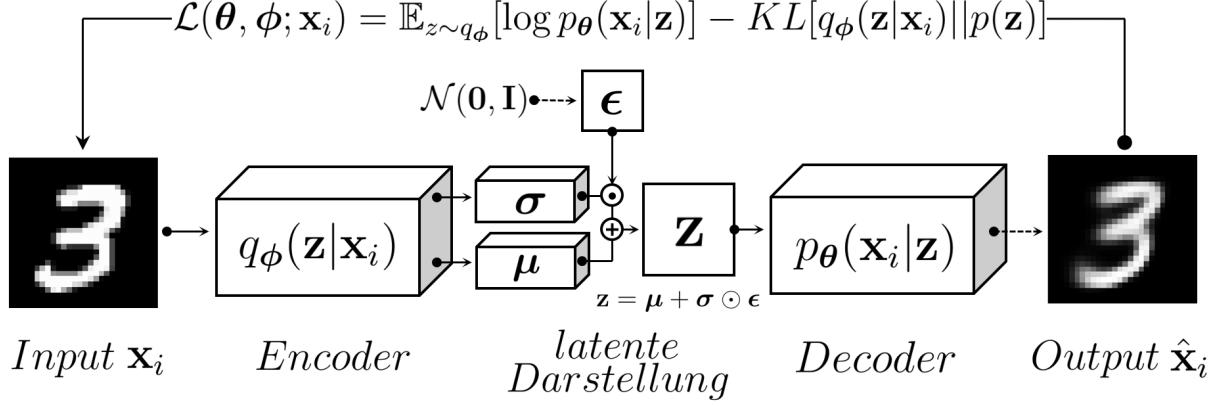
Gegeben die Realisierungen einer Zufallsvariablen  $\mathbf{z} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$  (Links) lässt sich mit Abbildung durch eine Funktion, hier  $g(\mathbf{z}) = \frac{\mathbf{z}}{15} + \frac{\mathbf{z}}{\|\mathbf{z}\|}$  eine ganz andere Verteilung erzeugen (Rechts). Dies nutzt der VAE aus, um aus Realisierungen einer normalverteilten Zufallsvariablen komplexe Verteilungen in großer Dimension zu modellieren. Die Funktion  $g$ , beim VAE der Decoder wird aus den Daten erlernt. [5]

Welche Verteilung für den Decoder  $p_{\theta}(\mathbf{x}_i | \mathbf{z})$  dafür am besten geeignet ist, hängt vom Typ der zu erzeugenden Daten ab.

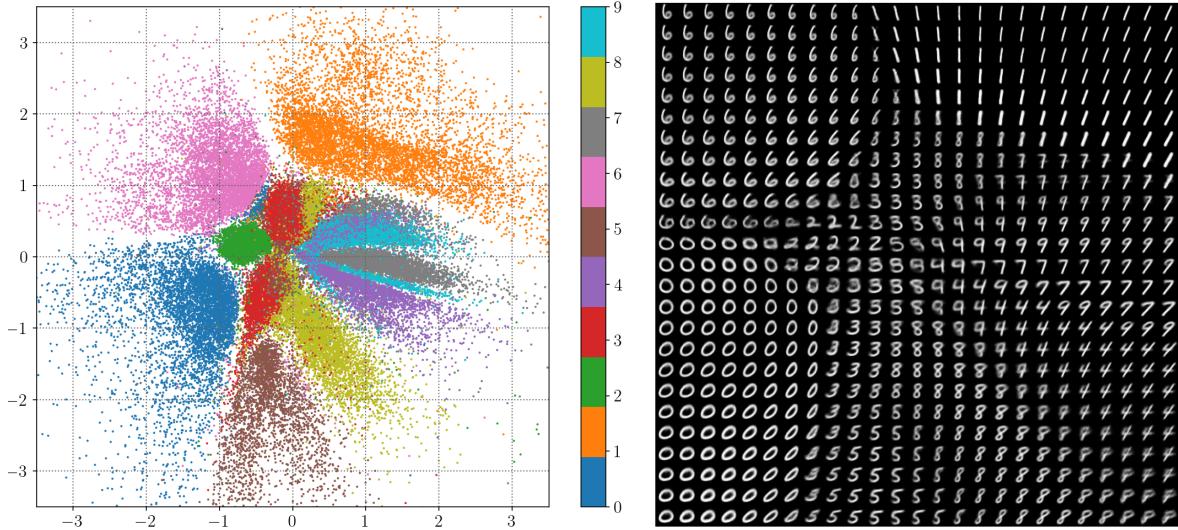
Für binäre Daten  $\mathbf{x}_i \in \{0, 1\}^D$  werden wir eine multivariate Bernoulliverteilung verwenden, also  $p_{\theta}(\mathbf{x}_i | \mathbf{z}) = \mathcal{B}(\mathbf{p}_i)$ . Gemeint ist hiermit eine elementweise Auswertung univariater Bernoulliverteilungen.

Für stetige Daten  $\mathbf{x}_i \in \mathbb{R}^D$  wählen wir eine multivariate Normalverteilung  $p_{\theta}(\mathbf{x}_i | \mathbf{z}) = \mathcal{N}(\hat{\boldsymbol{\mu}}_i, \hat{\boldsymbol{\sigma}}_i^2 \mathbf{I})$ .

Hierbei sind  $\hat{\boldsymbol{\mu}}_i = \hat{\boldsymbol{\mu}}_{\theta}(\mathbf{z})$  und  $\hat{\boldsymbol{\sigma}}_i^2 = \hat{\boldsymbol{\sigma}}_{\theta}^2(\mathbf{z})$  bzw.  $\mathbf{p}_i = \mathbf{p}_{\theta}(\mathbf{z})$  die Ausgaben des Decodernetzwerkes.



Schematische Darstellung eines Variational Autoencoders. Der hochdimensionale Input  $\mathbf{x}_i$  wird durch den Encoder auf einen Erwartungswertvektor  $\mu$  und Streuungsparameter  $\sigma$  abgebildet. Anschließend wird eine niedrigdimensionale latente Darstellung  $\mathbf{z}$  gezogen. Dabei wird der Reparametrisierungstrick  $\mathbf{z} = \mu + \sigma \odot \epsilon$  angewandt. Der Decoder rekonstruiert nun mit dieser Darstellung einen Output  $\hat{\mathbf{x}}_i$ . Der Modellverlust ist durch die negative ELBO gegeben, durch Minimierung dieser kann das Modell optimiert werden.



Erlernte latente Darstellung des MNIST-Trainingsdatensatzes im zweidimensionalen latenten Raum (links) und durch das Decodernetzwerk erzeugte Daten. (rechts) Die latenten Darstellungen  $\mathbf{z}$  wurden mit linearem Abstand im Quadrat  $[-2, 2]^2$  gewählt. Auf die verwendete Modellstruktur und den dazu geschriebenen Code gehen wir im praktischen Teil dieser Arbeit genauer ein.

Für eine konkrete Berechnung der ELBO betrachten wir zuerst die KL-Divergenz der nun parameterfreien Verteilung  $p(\mathbf{z})$  nach  $q_\phi(\mathbf{z}|\mathbf{x}_i)$ .  $d$  notiert hierbei die Dimension der latenten Darstellung und  $p_{\mathcal{N}(\mu_i, \sigma_i^2 \mathbf{I})}(\mathbf{z})$  bzw.  $p_{\mathcal{N}(\mathbf{0}, \mathbf{I})}(\mathbf{z})$  die Auswertung der jeweiligen Dichtefunktion an der Stelle  $\mathbf{z}$ .

$$D_{KL}(q_\phi(\mathbf{z}|\mathbf{x}_i) || p(\mathbf{z})) = \mathbb{E}_{\mathbf{z} \sim q_\phi(\mathbf{z}|\mathbf{x}_i)} \left[ \log \left( \frac{p_{\mathcal{N}(\mu_i, \sigma_i^2 \mathbf{I})}(\mathbf{z})}{p_{\mathcal{N}(\mathbf{0}, \mathbf{I})}(\mathbf{z})} \right) \right].$$

$$= \mathbb{E}_{\mathbf{z} \sim \mathcal{N}(\boldsymbol{\mu}_i, \boldsymbol{\sigma}_i^2 \mathbf{I})} [\log(p_{\mathcal{N}(\boldsymbol{\mu}_i, \boldsymbol{\sigma}_i^2 \mathbf{I})}(\mathbf{z})) - \log(p_{\mathcal{N}(\mathbf{0}, \mathbf{I})}(\mathbf{z}))].$$

Die jeweiligen Dichtefunktionen sind durch

$$\begin{aligned} p_{\mathcal{N}(\boldsymbol{\mu}_i, \boldsymbol{\sigma}_i^2 \mathbf{I})}(\mathbf{z}) &= \frac{1}{\sqrt{2\pi^d \det(\boldsymbol{\sigma}_i^2 \mathbf{I})}} \exp\left(-\frac{1}{2}(\mathbf{z} - \boldsymbol{\mu}_i)^T (\boldsymbol{\sigma}_i^2 \mathbf{I})^{-1} (\mathbf{z} - \boldsymbol{\mu}_i)\right) \\ \log(p_{\mathcal{N}(\boldsymbol{\mu}_i, \boldsymbol{\sigma}_i^2 \mathbf{I})}(\mathbf{z})) &= -\frac{1}{2}(\log(2\pi^d \det(\boldsymbol{\sigma}_i^2 \mathbf{I})) + (\mathbf{z} - \boldsymbol{\mu}_i)^T (\boldsymbol{\sigma}_i^2 \mathbf{I})^{-1} (\mathbf{z} - \boldsymbol{\mu}_i)) \\ \log(p_{\mathcal{N}(\mathbf{0}, \mathbf{I})}(\mathbf{z})) &= -\frac{1}{2}(\log(2\pi^d) + \mathbf{z}^T \mathbf{I} \mathbf{z}) \end{aligned}$$

gegeben. Einsetzen und aufteilen des Erwartungswertes liefert uns

$$\begin{aligned} \mathbb{E}_{\mathbf{z} \sim \mathcal{N}(\boldsymbol{\mu}_i, \boldsymbol{\sigma}_i^2 \mathbf{I})} \left[ -\frac{1}{2} \left( \log(\det(\boldsymbol{\sigma}_i^2 \mathbf{I})) + (\mathbf{z} - \boldsymbol{\mu}_i)^T (\boldsymbol{\sigma}_i^2 \mathbf{I})^{-1} (\mathbf{z} - \boldsymbol{\mu}_i) - \mathbf{z}^T \mathbf{I} \mathbf{z} \right) \right] \\ = -\frac{1}{2} \left( \log(\det(\boldsymbol{\sigma}_i^2 \mathbf{I})) + \mathbb{E}_{\mathbf{z} \sim \mathcal{N}(\boldsymbol{\mu}_i, \boldsymbol{\sigma}_i^2 \mathbf{I})} [(\mathbf{z} - \boldsymbol{\mu}_i)^T (\boldsymbol{\sigma}_i^2 \mathbf{I})^{-1} (\mathbf{z} - \boldsymbol{\mu}_i)] - \mathbb{E}_{\mathbf{z} \sim \mathcal{N}(\boldsymbol{\mu}_i, \boldsymbol{\sigma}_i^2 \mathbf{I})} [\mathbf{z}^T \mathbf{I} \mathbf{z}] \right). \end{aligned}$$

Da  $\mathbf{z} \sim \mathcal{N}(\boldsymbol{\mu}_i, \boldsymbol{\sigma}_i^2)$  gilt für die Quadratischen Formen [6]:

$$\begin{aligned} \mathbb{E}_{\mathbf{z} \sim \mathcal{N}(\boldsymbol{\mu}_i, \boldsymbol{\sigma}_i^2 \mathbf{I})} [\mathbf{z}^T \mathbf{I} \mathbf{z}] &= \boldsymbol{\mu}_i^T \mathbf{I} \boldsymbol{\mu}_i + \text{Tr}(\boldsymbol{\sigma}_i^2 \mathbf{I}) \\ \mathbb{E}_{\mathbf{z} \sim \mathcal{N}(\boldsymbol{\mu}_i, \boldsymbol{\sigma}_i^2 \mathbf{I})} [(\mathbf{z} - \boldsymbol{\mu}_i)^T (\boldsymbol{\sigma}_i^2 \mathbf{I})^{-1} (\mathbf{z} - \boldsymbol{\mu}_i)] &= \text{Tr}((\boldsymbol{\sigma}_i^2 \mathbf{I})^{-1} \boldsymbol{\sigma}_i^2 \mathbf{I}). \end{aligned}$$

Nun lässt sich mit  $\boldsymbol{\sigma}_i^2 \mathbf{I} = \begin{pmatrix} \sigma_{i,1}^2 & & \\ & \ddots & \\ & & \sigma_{i,d}^2 \end{pmatrix}$  die KL-Divergenz ausrechnen.

$$\begin{aligned} &- \frac{1}{2} \left( \log(\det(\boldsymbol{\sigma}_i^2 \mathbf{I})) + \text{Tr}((\boldsymbol{\sigma}_i^2 \mathbf{I})^{-1} \boldsymbol{\sigma}_i^2 \mathbf{I}) - \boldsymbol{\mu}_i^T \mathbf{I} \boldsymbol{\mu}_i - \text{Tr}(\boldsymbol{\sigma}_i^2 \mathbf{I}) \right) \\ &= -\frac{1}{2} \left( \log \left( \prod_{j=1}^d \sigma_{i,j}^2 \right) + d - \sum_{j=1}^d \mu_{i,j}^2 - \sum_{j=1}^d \sigma_{i,j}^2 \right) \\ &= -\frac{1}{2} \sum_{j=1}^d \left( \log(\sigma_{i,j}^2) + 1 - \mu_{i,j}^2 - \sigma_{i,j}^2 \right). \end{aligned}$$

Auch  $p_{\boldsymbol{\theta}}(\mathbf{x}_i | \mathbf{z})$  kann mit unserer Wahl der Verteilungen explizit ausgerechnet werden. Mit  $D$  wird dazu die Dimension des Inputs und Outputs notiert. Für die Verwendung einer multivariaten Normalverteilung im Decoder ergibt sich:

$$\begin{aligned} \log(p_{\boldsymbol{\theta}}(\mathbf{x}_i | \mathbf{z})) &= \log(p_{\mathcal{N}(\hat{\boldsymbol{\mu}}_i, \hat{\boldsymbol{\sigma}}_i^2 \mathbf{I})}(\mathbf{x}_i)) \\ &= -\frac{1}{2} (\log(2\pi^D \det(\hat{\boldsymbol{\sigma}}_i^2 \mathbf{I})) + (\mathbf{x}_i - \hat{\boldsymbol{\mu}}_i)^T (\hat{\boldsymbol{\sigma}}_i^2 \mathbf{I})^{-1} (\mathbf{x}_i - \hat{\boldsymbol{\mu}}_i)) \\ &= -\frac{1}{2} \left( \log(2\pi^D) + \log \left( \prod_{p=1}^D \hat{\sigma}_{i,p}^2 \right) + \begin{pmatrix} \frac{x_{i,1} - \hat{\mu}_{i,1}}{\hat{\sigma}_{i,1}^2} & \dots & \frac{x_{i,D} - \hat{\mu}_{i,D}}{\hat{\sigma}_{i,D}^2} \end{pmatrix} \cdot (\mathbf{x}_i - \hat{\boldsymbol{\mu}}_i) \right) \end{aligned}$$

$$\begin{aligned}
&= -\frac{1}{2} \left( D \log(2\pi) + \sum_{p=1}^D \log(\hat{\sigma}_{i,p}^2) + \sum_{p=1}^D \frac{(x_{i,p} - \hat{\mu}_{i,p})^2}{\hat{\sigma}_{i,p}^2} \right) \\
&= -\frac{1}{2} \sum_{p=1}^D \left( \log(2\pi\hat{\sigma}_{i,p}^2) + \frac{(x_{i,p} - \hat{\mu}_{i,p})^2}{\hat{\sigma}_{i,p}^2} \right)
\end{aligned}$$

Für eine bernoulliverteilte Zufallsvariable ist die Dichtefunktion durch  $p_{\mathcal{B}(p_i)}(x_i) = p_i^{x_i}(1-p_i)^{1-x_i}$  definiert. Somit ergibt sich für die Verwendung der multivariaten Bernoulliverteilung im Decoder:

$$\begin{aligned}
\log(p_\theta(\mathbf{x}_i | \mathbf{z})) &= \log(p_{\mathcal{B}(\mathbf{p}_i)}(\mathbf{x}_i)) \\
&= \log(p_{\mathcal{B}(p_1)}(x_{i,1}) \cdot \dots \cdot p_{\mathcal{B}(p_D)}(x_{i,D})) \\
&= \log \left( \prod_{p=1}^D (p_{i,p})^{x_{i,p}} (1-p_{i,p})^{1-x_{i,p}} \right) \\
&= \sum_{p=1}^D x_{i,p} \log(p_{i,p}) + (1-x_{i,p}) \log(1-p_{i,p})
\end{aligned}$$

## 2.2 Second Order ODE-Variational Autoencoder

Gegeben sei eine *time series*  $\mathbf{x}_{0:T} = (\mathbf{x}_0, \mathbf{x}_1, \dots, \mathbf{x}_T) \in \mathbb{R}^{D \times T}$  zu observierten Zeitpunkten  $t_0, t_1, \dots, t_T$  mit zugehörigen latenten Zuständen  $\mathbf{z}_{0:T} = (\mathbf{z}_0, \mathbf{z}_1, \dots, \mathbf{z}_T) \in \mathbb{R}^{d \times T}$  geringerer Dimension.

Wir nehmen an, dass die zeitliche Änderung der latenten Zustände durch eine Differentialgleichung beschrieben werden kann. Das Vektorfeld der Differentialgleichung wird durch ein Neuronales Netzwerk  $f_\psi$ , ein *ODE-Netzwerk* in stetiger Zeit  $t$  modelliert.

$$\begin{aligned}
\frac{\partial \mathbf{z}(t)}{\partial t} &= f_\psi(\mathbf{z}(t), t), \\
\mathbf{z}(t_0) &= \mathbf{z}_0
\end{aligned}$$

Durch einen gegebenen Anfangswert  $\mathbf{z}_0$  erhalten wir für lipschitzstetige Funktionen in der ersten Variablen immer eine eindeutige Lösung des Anfangswertproblems

$$\mathbf{z}_T = \mathbf{z}_0 + \int_0^T f_\psi(\mathbf{z}(t), t) dt.$$

Das Integral lässt sich numerisch mit ODE-Solvieren berechnen.

Wir benutzen einen VAE zur Dimensionsreduzierung. Der Encoder bestimmt den Anfangswert  $\mathbf{z}_0$  der Differentialgleichung, welcher mit dem ODE-Netzwerk eine eindeutig

bestimmte Kurve im latenten Raum liefert. Der Decoder rekonstruiert anschließend zu den Zeitpunkten  $t_0, t_1, \dots, t_T$  die Inputsequenz  $\mathbf{x}_{0:T}$ .

ODEs erster Ordnung sind jedoch nicht in der Lage Datensätze zufriedenstellend zu modellieren, bei denen sich auch höhere Momente ändern. Wir werden daher ODEs zweiter Ordnung verwenden, welche sich äquivalent als ein System bestehend aus zwei Differentialgleichungen erster Ordnung notieren lassen.

Der latente Zustand  $\mathbf{z}_t = (\mathbf{s}_t, \mathbf{v}_t)^T$  zum Zeitpunkt  $t$ , wird dazu in eine Positionskomponente  $\mathbf{s}_t$  und eine Geschwindigkeitskomponente  $\mathbf{v}_t$  zerlegt, deren Anfangswerte  $\mathbf{s}_0$  und  $\mathbf{v}_0$  von zwei unterschiedlichen Encoder-Netzwerken, dem *Position-Encoder* und dem *Velocity-Encoder*, erlernt werden.

Der Position-Encoder erhält zur Bestimmung der latenten Positionskomponente die erste Komponente  $\mathbf{x}_0$  der Inputsequenz, während der Velocity-Encoder die ersten  $m \geq 2$  Komponenten  $\mathbf{x}_{0:m}$  der Inputsequenz erhält. Dies ermöglicht uns das Encodieren der höheren Momente.

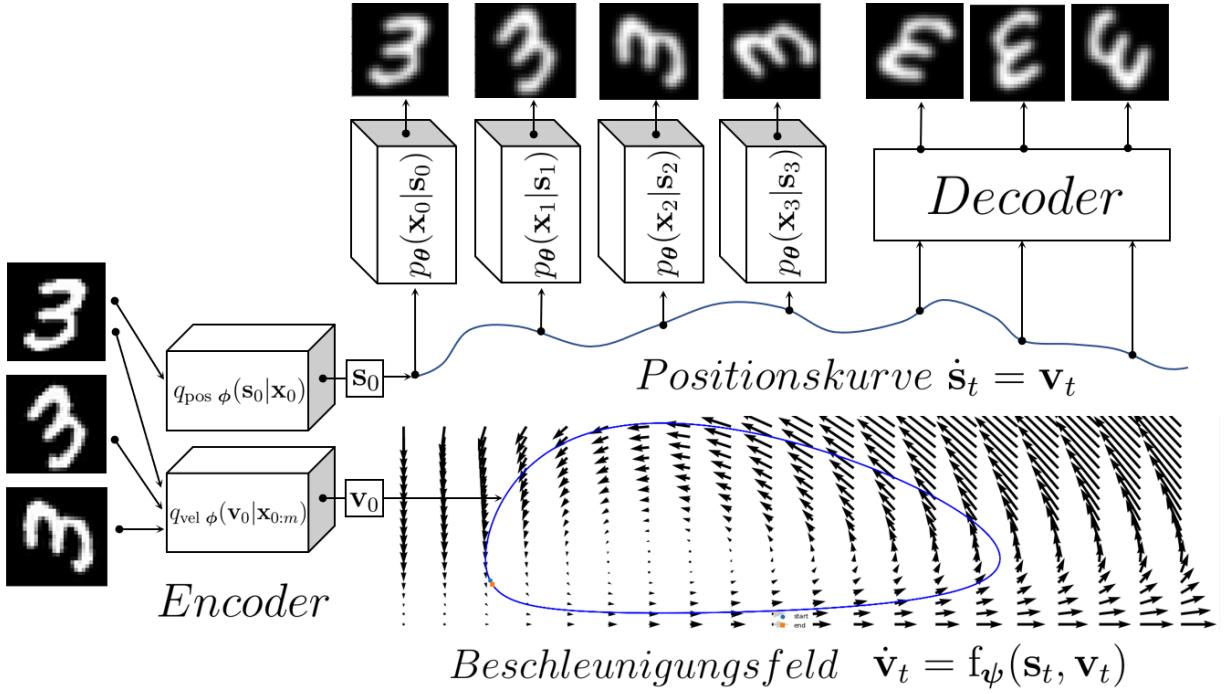
$$\begin{aligned} \frac{\partial^2 \mathbf{z}(t)}{\partial t^2} &= f_{\psi} \left( \mathbf{z}(t), \frac{\partial \mathbf{z}(t)}{\partial t}, t \right), \\ \begin{cases} \frac{\partial \mathbf{s}(t)}{\partial t} = \mathbf{v}(t) \\ \frac{\partial \mathbf{v}(t)}{\partial t} = f_{\psi}(\mathbf{s}(t), \mathbf{v}(t), t) \end{cases}, \quad \begin{pmatrix} \mathbf{s}_T \\ \mathbf{v}_T \end{pmatrix} &= \begin{pmatrix} \mathbf{s}_0 \\ \mathbf{v}_0 \end{pmatrix} + \int_0^T \begin{pmatrix} \mathbf{v}(t) \\ f_{\psi}(\mathbf{s}(t), \mathbf{v}(t), t) \end{pmatrix} dt. \end{aligned}$$

Die Autoren des Papers verwenden ein Bayesisches Neuronales Netzwerk zur Modellierung der Differentialgleichung um Unsicherheit besser darstellen zu können. Wir werden aber auf ein gewöhnliches Neuronales Netzwerk zurückgreifen und im nächsten Kapitel auf die Modellierung Stochastischer Differentialgleichung eingehen.

Als Trainingskriterium des ODE<sup>2</sup>VAE verwenden wir erneut die negative ELBO. Wir erhalten:

$$\begin{aligned} \log(p_{\theta}(\mathbf{x}_{0:T})) &\geq \mathbb{E}_{\mathbf{z}_{0:T} \sim q_{\phi,\psi}} [\log(p_{\theta}(\mathbf{x}_{0:T}|\mathbf{z}_{0:T}))] - D_{KL}[q_{\phi,\psi}(\mathbf{z}_{0:T}|\mathbf{x}_{0:m})||p_{\theta,\psi}(\mathbf{z}_{0:T})] \\ &= \underbrace{\mathbb{E}_{\mathbf{z}_0 \sim q_{\text{enc } \phi}} [\log(p_{\theta}(\mathbf{x}_0|\mathbf{z}_0))]}_{\text{Vanilla-VAE ELBO}} - D_{KL}[q_{\text{enc } \phi}(\mathbf{z}_0|\mathbf{x}_{0:m})||p_{\theta}(\mathbf{z}_0)] \\ &\quad + \underbrace{\sum_{t=1}^T \mathbb{E}_{\mathbf{z}_t \sim q_{\text{ode } \psi}} [\log(p_{\theta}(\mathbf{x}_t|\mathbf{z}_t))]}_{\text{dynamic loss}} - D_{KL}[q_{\text{ode } \psi}(\mathbf{z}_t|\mathbf{x}_{0:m})||p_{\theta,\psi}(\mathbf{z}_t)] \end{aligned}$$

Der erste Teil ist die ELBO eines Standard-VAE's. Als dem latenten Raum zugrundeliegenden Verteilung  $p_{\theta}(\mathbf{z}_0)$  wählen wir eine parameterfreie Standardnormalverteilung  $\mathcal{N}(\mathbf{0}, \mathbf{I})$ .



Schematische Darstellung des ODE<sup>2</sup>VAE. Eine Inputsequenz wird von den Encodermodellen auf die Startwerte einer Beschleunigungs- und Positionskurve abgebildet. Die Änderung der Geschwindigkeitskurve (blau) wird durch ein ODE-Netzwerk modelliert und mit einem ODE-Solver berechnet. Die Änderung der Positionskurve ergibt sich durch Integration über die Geschwindigkeitskurve. Zu festgelegten Zeitpunkten rekonstruiert der Decoder aus den Positions倣en  $s_t$  die Inputsequenz.

Die Encoder-Verteilung ist als Normalverteilung mit  $\mathbf{z}_0 = (s_0, v_0)^T$  durch

$$q_{\text{enc } \phi}(\mathbf{z}_0 | \mathbf{x}_{0:m}) = \mathcal{N} \left( \begin{pmatrix} \boldsymbol{\mu}_{\text{pos } \phi}(\mathbf{x}_0) \\ \boldsymbol{\mu}_{\text{vel } \phi}(\mathbf{x}_{0:m}) \end{pmatrix}, \begin{pmatrix} \text{diag}(\boldsymbol{\sigma}_{\text{pos } \phi}(\mathbf{x}_0)) & \mathbf{0} \\ \mathbf{0} & \text{diag}(\boldsymbol{\sigma}_{\text{vel } \phi}(\mathbf{x}_{0:m})) \end{pmatrix} \right)$$

gegeben. Wie erläutert hängt der Output des Velocity-Encoders dabei von den ersten  $m \geq 2$  Komponenten der Inputsequenz ab, während der Output des Position-Encoders nur durch die erste Komponente bestimmt wird.

Die Berechnung des *reconstruction loss* ist von der Wahl der im Decoder verwendeten Verteilung abhängig. Für unsere Datensätze werden wir eine Bernoulliverteilung verwenden. Die analytische Berechnung ist nun mit den Resultaten des letzten Kapitels möglich. Die Berechnung des *dynamic loss* ist komplizierter.

Der *reconstruction loss*  $\mathbb{E}_{\mathbf{z}_t \sim q_{\text{ode } \psi}} [\log(p_{\theta}(\mathbf{x}_t | \mathbf{z}_t))]$  lässt sich zwar mit den Resultaten des ersten Kapitels berechnen, für die approximative Berechnung der KL-Divergenz

$$\begin{aligned} D_{KL}[q_{\text{ode } \psi}(\mathbf{z}_t | \mathbf{x}_{0:m}) || p_{\theta, \psi}(\mathbf{z}_t)] &= \mathbb{E}_{\mathbf{z}_t \sim q_{\text{ode } \psi}} [\log(q_{\text{ode } \psi}(\mathbf{z}_t | \mathbf{x}_{0:m})) - \log(p_{\theta, \psi}(\mathbf{z}_t))] \\ &\stackrel{\text{M.C.}}{\simeq} \frac{1}{L} \sum_{i=1}^L \log(q_{\text{ode } \psi}(\mathbf{z}_t | \mathbf{x}_{0:m})) - \log(p_{\theta, \psi}(\mathbf{z}_t)) \end{aligned}$$

benötigen wir jedoch die log-Dichten der ODE-Verteilung  $q_{\text{ode } \psi}(\mathbf{z}_t | \mathbf{x}_{0:m})$  und die der Prior-Verteilung  $p_{\theta, \psi}(\mathbf{z}_t)$ .

Die latenten Werte  $\mathbf{z}_t = (\mathbf{s}_t, \mathbf{v}_t)^T$  sind nun aber nicht mehr von den Encoder-Netzwerken abhängig, sondern werden durch das ODE-Netzwerk  $f_\psi(\mathbf{s}_t, \mathbf{v}_t)$  bestimmt. Zur Berechnung der log-Dichte der ODE-Verteilung lassen sich folgende hilfreiche Theoreme anwenden.

**Theorem 1.** *Sei  $\mathbf{z}(t)$  eine stetige Zufallsvariable mit einer von der Zeit abhängigen Dichtefunktion  $q(\mathbf{z}(t))$ . Sei  $\frac{\partial \mathbf{z}(t)}{\partial t} = f(\mathbf{z}(t), t)$  eine in  $\mathbf{z}$  Lipschitzstetige und in  $t$  stetige Differentialgleichung, welche die Veränderung von  $\mathbf{z}(t)$  mit der Zeit beschreibt. Die Veränderung der Dichte folgt nun ebenfalls einer Differentialgleichung:*

$$\frac{\partial \log(q(\mathbf{z}(t)))}{\partial t} = -\text{Tr}\left(\frac{\partial f}{\partial \mathbf{z}}(t)\right).$$

*Beweis.* Ein ausführlicher Beweis und weitere Erklärungen sind in *Neural Ordinary Differential Equations* [7] zu finden.  $\square$

Die Berechnung der Spur der Jacobimatrix unseres ODE-Netzwerkes  $f_\psi(\mathbf{s}_t, \mathbf{v}_t)$  erfordert entsprechend der gewählten latenten Dimension  $d$  auch eine  $d$ -fache Auswertung des zugrundeliegenden Netzwerkes. Um den Rechenaufwand deutlich zu verringern, verwenden wir zwei Tricks.[8]

Die Berechnung der Spur lässt sich durch einen erwartungstreuen Schätzer  $\boldsymbol{\epsilon}^T \cdot \frac{\partial f}{\partial \mathbf{z}}(t) \cdot \boldsymbol{\epsilon}$  approximieren. Ein Vektor-Matrix Produkt  $\boldsymbol{\epsilon}^T \cdot \frac{\partial f}{\partial \mathbf{z}}(t)$  mit einem Zufallsvektor  $\boldsymbol{\epsilon}$ , lässt sich durch *Reverse-Mode Automatic Differentiation* mit einmaliger Auswertung der Funktion  $f$  berechnen.

**Theorem 2.** *Sei  $\mathbf{M} \in \mathbb{R}^{d \times d}$  eine beliebige Matrix und  $\boldsymbol{\epsilon} \sim p(\boldsymbol{\epsilon})$  eine Zufallsvariable mit  $\mathbb{E}[\boldsymbol{\epsilon}] = \mathbf{0}$  und  $\text{Cov}(\boldsymbol{\epsilon}) = \mathbf{I}$ . Die Spur der Matrix  $\mathbf{M}$  lässt sich erwartungstreu durch ein doppeltes Matrix-Vektorprodukt der Matrix  $\mathbf{M}$  mit dem Zufallvektor  $\boldsymbol{\epsilon}$  schätzen:*

$$\text{Tr}(\mathbf{M}) = \mathbb{E}_{\boldsymbol{\epsilon} \sim p(\boldsymbol{\epsilon})} [\boldsymbol{\epsilon}^T \mathbf{M} \boldsymbol{\epsilon}].$$

*Beweis.* Wir orientieren uns an [9]. Für  $\boldsymbol{\epsilon} \sim p(\boldsymbol{\epsilon})$  mit  $\mathbb{E}[\boldsymbol{\epsilon}] = \mathbf{0}$  und  $\text{Cov}(\boldsymbol{\epsilon}) = \mathbf{I}$  gilt  $\mathbb{E}_{\boldsymbol{\epsilon} \sim p(\boldsymbol{\epsilon})} [\boldsymbol{\epsilon}^T \boldsymbol{\epsilon}] = \mathbf{I}$ . Wir erhalten nun für die Spur einer Matrix  $\mathbf{M}$ :

$$\text{Tr}(\mathbf{M}) = \text{Tr}(\mathbf{M}\mathbf{I}) = \text{Tr}(\mathbf{M}\mathbb{E}_{\boldsymbol{\epsilon} \sim p(\boldsymbol{\epsilon})} [\boldsymbol{\epsilon}^T \boldsymbol{\epsilon}]) = \mathbb{E}_{\boldsymbol{\epsilon} \sim p(\boldsymbol{\epsilon})} [\text{Tr}(\mathbf{M}\boldsymbol{\epsilon}^T \boldsymbol{\epsilon})] = \mathbb{E}_{\boldsymbol{\epsilon} \sim p(\boldsymbol{\epsilon})} [\boldsymbol{\epsilon}^T \mathbf{M} \boldsymbol{\epsilon}].$$

$\square$

Wir wählen für  $p(\boldsymbol{\epsilon})$  eine multivariate Standardnormalverteilung.

Auch hier erlaubt uns eine groß gewählte Batchsize, z.B.  $M \geq 100$  die Anzahl der Ziehungen für Monte-Carlo Schätzung als  $L = 1$  zu wählen.

Mit diesen Resultaten können wir die log-Dichte der ODE-Verteilung mit relativ geringem Rechenaufwand berechnen. Für die Änderung der log-Dichte gilt:

$$\begin{aligned}
\frac{\partial \log(q_{\text{ode } \psi}(\mathbf{z}_t | \mathbf{x}_{0:m}))}{\partial t} &\stackrel{\text{T.1}}{=} -\text{Tr}\left(\frac{\partial f_\psi(\mathbf{z}_t)}{\partial \mathbf{z}_t}\right) \\
&= -\text{Tr}\left(\frac{\frac{\partial \mathbf{v}_t}{\partial \mathbf{s}_t}}{\frac{\partial f_\psi(\mathbf{s}_t, \mathbf{v}_t)}{\partial \mathbf{s}_t}} \frac{\frac{\partial \mathbf{s}_t}{\partial \mathbf{v}_t}}{\frac{\partial f_\psi(\mathbf{s}_t, \mathbf{v}_t)}{\partial \mathbf{v}_t}}\right) \\
&= -\text{Tr}\left(\frac{\partial f_\psi(\mathbf{s}_t, \mathbf{v}_t)}{\partial \mathbf{v}_t}\right) \\
&\stackrel{\text{T.2}}{=} -\mathbb{E}_{\boldsymbol{\epsilon} \sim \mathcal{N}(0, \mathbf{I})} \left[ \boldsymbol{\epsilon}^T \frac{\partial f_\psi(\mathbf{z}_t)}{\partial \mathbf{v}_t} \boldsymbol{\epsilon} \right].
\end{aligned}$$

Durch Integration ergibt sich nun

$$\begin{aligned}
\log(q_{\text{ode } \psi}(\mathbf{z}_t | \mathbf{x}_{0:m})) &= \log(q_{\text{enc } \psi}(\mathbf{z}_0 | \mathbf{x}_{0:m})) - \int_0^t \mathbb{E}_{\boldsymbol{\epsilon} \sim \mathcal{N}(0, \mathbf{I})} \left[ \boldsymbol{\epsilon}^T \frac{\partial f_\psi(\mathbf{s}_t, \mathbf{v}_t)}{\partial \mathbf{v}_t} \boldsymbol{\epsilon} \right] dt \\
&= \log(q_{\text{enc } \psi}(\mathbf{z}_0 | \mathbf{x}_{0:m})) - \mathbb{E}_{\boldsymbol{\epsilon} \sim \mathcal{N}(0, \mathbf{I})} \left[ \int_0^t \boldsymbol{\epsilon}^T \frac{\partial f_\psi(\mathbf{s}_t, \mathbf{v}_t)}{\partial \mathbf{v}_t} \boldsymbol{\epsilon} dt \right] \\
&\stackrel{\text{M.C.}}{\simeq} \log(q_{\text{enc } \phi}(\mathbf{z}_0 | \mathbf{x}_{0:m})) - \frac{1}{L} \sum_{i=1}^L \int_0^t \boldsymbol{\epsilon}^T \frac{\partial f_\psi(\mathbf{s}_t, \mathbf{v}_t)}{\partial \mathbf{v}_t} \boldsymbol{\epsilon} dt.
\end{aligned}$$

Die Randbedingung  $\log(q_{\text{enc } \phi}(\mathbf{z}_0 | \mathbf{x}_{0:m}))$  ist durch die Encoder-Verteilung gegeben. Der Erwartungswert wird durch Monte-Carlo Schätzung approximiert,  $\boldsymbol{\epsilon}$  wird außerhalb des Integrals aus einer Standardnormalverteilung gezogen und das Integral numerisch durch ODE-Integratoren zusammen mit unserer Differentialgleichung berechnet.

$$\begin{aligned}
\frac{\partial h(t)}{\partial t} &= \begin{cases} \frac{\partial \mathbf{s}(t)}{\partial t} = \mathbf{v}(t) \\ \frac{\partial \mathbf{v}(t)}{\partial t} = f_\psi(\mathbf{s}(t), \mathbf{v}(t), t) \\ \frac{\partial(\log \mathbf{q}(t))}{\partial t} = -\boldsymbol{\epsilon}^T \frac{\partial f_\psi(\mathbf{s}, \mathbf{v}, t)}{\partial \mathbf{v}}(t) \boldsymbol{\epsilon} \end{cases} \\
h(t_T) &= \text{ODEintegrate}(h, h(t_0), t_1, t_2, \dots, t_T, \boldsymbol{\epsilon}, f_\psi)
\end{aligned}$$

Der Wert  $\log(p_{\theta, \psi}(\mathbf{z}_t))$ , der ebenfalls für die Berechnung der KL-Divergenz benötigt wird ist nicht bestimmbar. Müsste man doch die log-Dichten mit der unbekannten, "wahren" Funktion  $f_{\psi^*}(\mathbf{s}_t, \mathbf{v}_t)$  mit oben hergeleiteter Formel ausrechnen. Die Autoren des Papers treffen eine vereinfachende Annahme und legen wie bereits für den ersten Zustand geltend  $p_{\theta, \psi}(\mathbf{z}_t) = \mathcal{N}(\mathbf{0}, \mathbf{I})$  als parameterfreie multivariate Normalverteilung fest.

Bei uns hat das aber zu dem Problem geführt, dass der KL-Term den *reconstruction*

*loss* stark übertrifft und das Verfahren im latenten Raum nur die Struktur einer Normalverteilung annehmen will ohne gleichzeitig den *reconstruction loss* zufriedenstellend zu verringern. Wir greifen daher auf eine Abwandlung der im Paper vorgestellten alternativen Lossfunktion zurück. Das Verfahren konvergiert dann nur langsam und die Bilder neigen schnell dazu zu verschwimmen.

$$\begin{aligned}
-\mathcal{L}(\phi, \psi, \theta, \mathbf{x}_{0:T}) = & \underbrace{\mathbb{E}_{\mathbf{z}_0 \sim q_{\text{enc } \phi}} [\log(p_{\theta}(\mathbf{x}_0 | \mathbf{z}_0))] - \gamma D_{KL}[q_{\text{enc } \phi}(\mathbf{z}_0 | \mathbf{x}_{0:m}) || p_{\mathcal{N}(\mathbf{0}, \mathbf{I})}(\mathbf{z}_0)]}_{\gamma\text{-VAE ELBO}} \\
& + \underbrace{\sum_{t=1}^T \mathbb{E}_{\mathbf{z}_t \sim q_{\text{ode } \psi}} [\log(p_{\theta}(\mathbf{x}_t | \mathbf{z}_t))] - \gamma D_{KL}[q_{\text{ode } \psi}(\mathbf{z}_t | \mathbf{x}_{0:m}) || p_{\mathcal{N}(\mathbf{0}, \mathbf{I})}(\mathbf{z}_t)]}_{\text{dynamic loss}} \\
& - \gamma \underbrace{\sum_{t=0}^{T-m} D_{KL}[q_{\text{ode } \psi}(\mathbf{z}_t | \mathbf{x}_{0:m}) || q_{\text{enc } \phi}(\mathbf{z}_t | \mathbf{x}_{t:(t+m)})]}_{\text{KL-Divergenz zur Encoderverteilung}}
\end{aligned}$$

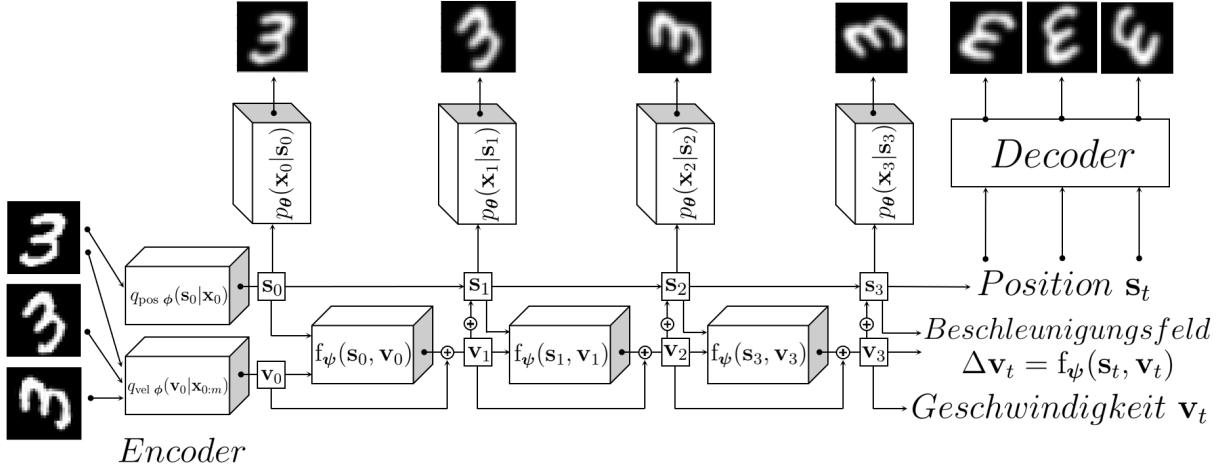
Wir multiplizieren die KL-Divergenz mit einem Faktor  $\gamma$ , der im Verlauf des Trainings schrittweise von 0 auf 1 angehoben wird. So wird zu Beginn des Trainingsprozess das Anfertigen guter Rekonstruktionen priorisiert und die sinnvolle Anordnung der latenten Variablen im latenten Raum erst später im Trainingsprozess vorgenommen.

Außerdem minimieren wir weiter die KL-Divergenz zur Encoderverteilung  $q_{\text{enc } \phi}(\mathbf{z}_t | \mathbf{x}_{t:(t+m)})$ . So wird zusätzlich auch sichergestellt, dass Encodernetzwerk und ODE-Netzwerk der selben Sequenz latente Werte mit ähnlicher Dichte zuweisen.

Diese Lossfunktion lässt sich analytisch berechnen und die Parameter  $\phi, \psi, \theta$  werden anhand dieses Trainingskriterium aktualisiert.

Ebenfalls gute Ergebnisse liefert bei uns die Approximation der Integrale durch eine Summe mit Schrittweite  $dt = 1$  und das Verwenden des *reconstruction loss* als alleiniges Trainingskriterium.

Visuell sind die Rekonstruktionen bei deutlich geringerer Trainingszeit (Bei uns ca. Faktor 30) auf unseren Datensätzen mit denen des Neural-ODE auf unseren Datensätzen vergleichbar. Durch Verwenden einer Summe und Simplifizierung des Trainingskriterium gehen einige praktische Eigenschaften des Modells verloren, wie das Erzeugen neuer Daten durch ziehen aus einer Normalverteilung, das Anfertigen von Rekonstruktionen in stetiger Zeit, die mathematische Interpretation des Modells als Verteilung und somit auch das Schätzen der Wahrscheinlichkeitsdichte und schließlich noch die größere Resistenz des Modells gegenüber Overfitting. Um einige dieser Modelleigenschaften zu erhalten könnte



Das alternative Modell. Die Veränderung in der Zeit wird erneut durch das ODE-Netzwerk modelliert. Wir erhalten den Positionszustand  $\mathbf{s}_{t+1} = \mathbf{s}_t + \mathbf{v}_t + f_\psi(\mathbf{s}_t, \mathbf{v}_t)$  aber durch Addition des Geschwindigkeitszustand und Auswertung des ODE-Netzwerkes, was den Rechenaufwand stark reduziert. Eine stetige Modellierung der *time series* ist nun zwar nicht mehr möglich, Einbußen in der Qualität der Rekonstruktionen konnten wir aber nicht beobachten.

man auch als Trainingskriterium auch den *reconstruction loss* mit der Vanilla-VAE ELBO verwenden.

### 2.3 M-th Order SDE-Variational Autoencoder

Bisher wurden die Pfade der latenten Darstellungen  $z = (s, v) : [0, \tau] \times \Omega \rightarrow \mathbb{R}^d$  als Lösungen von Differentialgleichungen der Form

$$\begin{aligned}\nabla s(\cdot, \omega) &= v(\cdot, \omega) \\ \nabla v(\cdot, \omega) &= \mathbf{f}(s(\cdot, \omega), v(\cdot, \omega))\end{aligned}$$

vorausgesetzt, wobei die 'Differential Function'  $\mathbf{f} = f_\psi$  dann durch ein Neuronales Netzwerk approximiert wurde.

Wir haben die folgenden Möglichkeiten in Betracht gezogen um den ODE<sup>2</sup>VAE zu verallgemeinern:

- Die Pfade der latenten Darstellungen als Lösungen von Differentialgleichungen von Grad  $M \geq 2$  voraussetzen. Z.B. hieße das für  $M = 3$ :

$$\begin{aligned}\nabla s(\cdot, \omega) &= v(\cdot, \omega) \\ \nabla v(\cdot, \omega) &= w(\cdot, \omega) \\ \nabla w(\cdot, \omega) &= \mathbf{f}(s(\cdot, \omega), v(\cdot, \omega), w(\cdot, \omega))\end{aligned}$$

- Die Form der Differentialgleichung weniger einschränken, indem wir beliebige  $M$ -ODE zulassen.

$$\begin{aligned}\nabla s(\cdot, \omega) &= \mathbf{f}_1(s(\cdot, \omega), v(\cdot, \omega), w(\cdot, \omega)) \\ \nabla v(\cdot, \omega) &= \mathbf{f}_2(s(\cdot, \omega), v(\cdot, \omega), w(\cdot, \omega)) \\ \nabla w(\cdot, \omega) &= \mathbf{f}_3(s(\cdot, \omega), v(\cdot, \omega), w(\cdot, \omega))\end{aligned}$$

- Einen Probabilistic Encoder auf jedes Frame  $x_i$  separat anwenden. Bisher wurde der Encoder nur auf die ersten Frames angewendet um  $s(0, \cdot), v(0, \cdot)$  zu finden. Dies geht, wie wir später sehen werden, mit der Betrachtung von Stochastic Differential Equations (SDEs) einher.

### 2.3.1 Einführung der Notation

Zunächst führen wir eine Notation ein, die eher an den Standards der Stochastik und weniger an denen von Machine-Learning orientiert ist. Dazu wiederholen wir auch grob die Idee eines ODE<sup>2</sup>VAE.

#### Idee des Autoencoders:

Sei  $(\mathbb{P}, \Omega, \mathcal{A})$  ein Wahrscheinlichkeitsraum mit der Zufallsvariable  $x : \Omega \rightarrow \mathbb{R}^D$ . Bei einem normalen Autoencoder wird angenommen, dass eine weitere Zufallsvariable  $z : \Omega \rightarrow \mathbb{R}^d$  existiert, sodass  $x = f(z)$  und  $z = g(x)$  für messbare Funktionen  $f, g$ . Die Funktionen  $f$  (Decoder) und  $g$  (Encoder) werden dabei von Neuronalen Netzwerken anhand eines Großen Datensatzes  $x^1, \dots, x^N \sim (x)_*\mathbb{P}$  erlernt. Als Verlustfunktion verwendet man beispielsweise den Reconstruction-Loss oder Mean Squared Error plus eventuelle Regularisierungsterme um Overfitting zu verhindern.

Man stellt fest, dass der latente Raum hier eher unstrukturiert in dem Sinne ist, dass das Bildmaß  $(f)_*\mathbb{P}$  auf  $\mathbb{R}^d$  vielleicht nur an ein paar Stellen den Großteil seiner Masse verteilt. An diesen Stellen im latenten Raum könnten wiederum kleine Schwankungen große Unterschiede in den Rekonstruktionen ausmachen. Anschaulich ist das Problem mit Autoencodern, dass sie den latenten Raum  $\mathbb{R}^d$  nicht 'gleichmäßig' nutzen und man keine Kontrolle darüber hat welche Anordnungen angenommen werden.

#### Idee des VAE:

Einem Variational Autoencoder will man im latenten Raum eine Struktur vorgeben.  $(z)_*\mathbb{P}$  soll nun eine festgelegte Verteilung (z.B. Normalverteilung in  $\mathbb{R}^d$ ) sein. Es müssen hier auch der Encoder  $g$  und Decoder  $f$  nicht mehr deterministisch sein, weshalb sie in der Literatur mit  $q(z|x)$  und  $p(x|z)$  beschrieben werden. Das Ziel ist erst mit Daten  $x^1, \dots, x^N \sim (x)_*\mathbb{P}$  den Encoder und Decoder zu trainieren, um dann neue Daten generieren zu können, indem man aus der Normalverteilung  $(z)_*\mathbb{P}$  Werte  $z^1, \dots, z^N$  zieht und dann den Decoder  $f$  darauf anwendet.

Dabei wird der Encoder als Neuronales Netzwerk aufgebaut, das im Falle einer Normalverteilung Erwartungswert  $\mu(x) \in \mathbb{R}^d$  und Covarianzmatrix  $\Sigma(x) \in \text{Diag}(d \times d)$  bestimmt, bevor die latente Darstellung  $z$  nach Reparametrisierung aus einer von  $x$  abhängigen Verteilung, beispielsweise  $g(x) = \mathcal{N}(\mu(x), \Sigma(x))$ , gezogen wird. Damit können einzelne Datenpunkte anschaulich mehr Platz im latenten Raum  $\mathbb{R}^d$  ausfüllen. Dies lässt sich auch als Maßnahme gegen Overfitting des Encoders interpretieren.

Es werden neben dem Reconstruction Loss, der der Log-Likelihood entspricht mit der KL-Divergenz noch zusätzliche Terme zur Verlustfunktion addiert, die dafür zu sorgen, dass  $(z)_*\mathbb{P}$  eine Normalverteilung ist und dass die Verteilung  $(f \circ z)_*\mathbb{P}$  ähnlich der Verteilung  $(x)_*\mathbb{P}$  ist.

### Idee des ODE<sup>2</sup>VAE:

Ein ODE<sup>2</sup>VAE beschäftigt sich mit Zufallsvariablen  $x_0, \dots, x_m : \Omega \rightarrow \mathbb{R}^D$ , bei denen angenommen wird, dass ein stetiger Prozess  $\tilde{x} : [0, \tau] \times \Omega \rightarrow \mathbb{R}^D$  existiert, sodass  $x_k = \tilde{x}_{t_k}$  für  $0 = t_0 < \dots < t_m = \tau$ . Der ODE<sup>2</sup>VAE sucht nun einen latenten Prozess  $\tilde{z} : [0, \tau] \times \Omega \rightarrow \mathbb{R}^d$ , der punktweise die Informationen von  $\tilde{x}$  encodiert, also soll der Decoder ein guter Schätzer für  $\tilde{x}(t, \cdot)$  sein. Wieder wird die Struktur von  $\tilde{z}$  im latenten Raum vorgeschrieben, wobei nun auch eine Vorschrift zur zeitlichen Entwicklung gemacht wird. Für spätere Zeitpunkte  $t > 0$  soll  $\tilde{z}(t, \omega)$  als Lösung der Differentialgleichung

$$\begin{aligned}\nabla s(\cdot, \omega) &= v(\cdot, \omega) \\ \nabla v(\cdot, \omega) &= \mathbf{f}(s(\cdot, \omega), v(\cdot, \omega))\end{aligned}$$

nur von  $s(0, \omega)$ ,  $v(0, \omega)$  und  $\mathbf{f}$  abhängen. In der Literatur wird hier das  $\tilde{z}$  als gemeinsamer Vektor  $(s, v)$  betrachtet. Es wird allerdings nur  $s$  an den Decoder weitergeben, also soll  $f(s(t, \cdot))$  ein guter Schätzer für  $\tilde{x}(t, \cdot)$  sein. Die Anfangswerte  $s(0, \cdot)$  und  $v(0, \cdot)$  werden durch jeweils einen Encoder bestimmt, die als Input  $x_0$ , beziehungsweise mindestens drei anfangsframes für  $v(0, \cdot)$ , erhalten.

Die Verlustfunktion besteht aus Reconstruction-Loss, Terme um Overfitting von  $\mathbf{f}$  zu verhindern, sowie Terme um  $(\tilde{z}(t_k, \cdot))_* \mathbb{P}$  für jedes  $k \leq m$  einer Normalverteilung auf  $\mathbb{R}^d$  näher zu bringen.

Der ODE<sup>2</sup>VAE liefert gute Ergebnisse beim Beschreiben von einfachen Bewegungsabläufen, deren Informationsgehalt nicht mit der Zeit zunimmt. Aber gerade bei chaotischen Systemen ist die Vorschrift, dass sämtliche Information schon zu Beginn (in den ersten drei Frames) vorhanden sein muss eher dem Lernen abträglich.

Besonders ist in einem etwas chaotischerem Setting das generieren von glaubhaften *time series*  $x_0, \dots, x_m$  nicht möglich, da spätere  $x_k, x_{k+1}$  immer deterministisch von einer gemeinsamen Zufallsvariable abhängen.

### Idee des $M$ -th Order SDE-VAE:

Wir beschäftigen uns wieder mit Zufallsvariablen  $x_0, \dots, x_m : \Omega \rightarrow \mathbb{R}^D$ , bei denen angenommen wird, dass ein stetiger Prozess  $\tilde{x} : [0, \tau] \times \Omega \rightarrow \mathbb{R}^D$  existiert, sodass  $x_k = \tilde{x}_{t_k}$  für  $0 = t_0 < \dots < t_m = \tau$ . Wie schon beim ODE<sup>2</sup>VAE suchen wir einen latenten Prozess  $\tilde{z} : [0, \tau] \times \Omega \rightarrow \mathbb{R}^d$ ,

für den der Decoder ein guter Schätzer für  $\tilde{x}(t, \cdot)$  ist. Hier gehen wir jedoch von einer anderen Zeitlichen Struktur aus.

Wir schreiben  $\tilde{z} = \tilde{z}^{(0)}$  und setzen voraus, dass Zufallsvariablen (Startverteilungen)

$$\tilde{z}^{(1)}(0, \cdot), \dots, \tilde{z}^{(M-1)}(0, \cdot) : \Omega \rightarrow \mathbb{R}^d$$

existieren, sodass  $\tilde{z}^{(k)}$  für alle  $k < M$  die  $d$ -Dimensionale SDE

$$\begin{aligned} \forall i \leq d : d(\tilde{z}^{(k),i}(t, \cdot)) = & \boldsymbol{\mu}^{k,i}(\tilde{z}^{(0)}(t, \cdot), \dots, \tilde{z}^{(M-1)}(t, \cdot)) dt \\ & + \sum_{j=1}^n \boldsymbol{\sigma}^{k,i,j}(\tilde{z}^{(0)}(t, \cdot), \dots, \tilde{z}^{(M-1)}(t, \cdot)) dW_t^j \end{aligned}$$

erfüllt. Dies lässt sich als eine gemeinsame SDE im Raum der  $(M \times d)$ -Matrizen, anstatt wie gewohnt im  $\mathbb{R}^d$ , betrachten. Wir betrachten also den Matrix-wertigen Prozess

$$Z_t(\omega) = \begin{pmatrix} \tilde{z}^{(0,1)}(t, \omega) & .. & \tilde{z}^{(0,d)}(t, \omega) \\ : & & : \\ \tilde{z}^{(M-1,1)}(t, \omega) & .. & \tilde{z}^{(M-1,d)}(t, \omega) \end{pmatrix} \in \mathbb{R}^{M \times d}.$$

Es wird nur die Erste Zeile von  $Z$  (analog zu  $s$  im ODE<sup>2</sup>VAE) an den Decoder weitergegeben. Die Funktionen  $\boldsymbol{\mu} : \mathbb{R}^{M \times d} \rightarrow \mathbb{R}^{M \times d}$  und  $\boldsymbol{\sigma} : \mathbb{R}^{M \times d} \rightarrow \mathbb{R}^{M \times d \times n}$  werden mit Neuronalen Netzwerken gelernt.

Das Ziel hierbei ist nicht diese Lösung direkt in einen Encoder einzugeben um die eingegebenen Daten zu rekonstruieren, sondern anhand des gelernten  $\boldsymbol{\mu}$  eine ODE ohne 'Unsicherheit'  $\boldsymbol{\sigma}$  zu simulieren. Diese glatte Lösung der ODE kann dann in die Decoder eingegeben um anschaulich geglättete oder weniger chaotische Daten zu erhalten.

Interessant ist, dass beim Training die originalen, ungeglätteten latenten Darstellungen  $\tilde{z}^{(0)}(t_0, \cdot), \dots, \tilde{z}^{(0)}(t_m, \cdot)$  in die Decoder eingegeben werden. Zusätzliche Verlustfunktionen sorgen dafür, dass  $\boldsymbol{\mu}, \boldsymbol{\sigma}$  erlernt werden sowie dass die En-&Decoder lernen  $z$  als SDEs darzustellen.

Die erlernten  $\boldsymbol{\mu}, \boldsymbol{\sigma}$  können verwendet werden um neue Pfade von  $Z$  zu ziehen (simulieren), mit denen man glaubhaft neue Daten generieren kann.

Der ODE<sup>2</sup>VAE ist nur ein einfacher Spezialfall dieser Darstellung. Wenn wir nämlich  $M = 2$ ,  $\boldsymbol{\sigma} = 0$  und  $\boldsymbol{\mu}^{(0)}(\tilde{z}^{(0)}, \tilde{z}^{(1)}) = \tilde{z}^{(1)}$  setzen, dann erfüllt  $\boldsymbol{\mu}^{(0)}(\tilde{z}^{(0)}, \tilde{z}^{(1)})$  dieselbe Rolle, wie  $\mathbf{f}$  im ODE<sup>2</sup>VAE.

Damit man mit dem Modell besser neue Daten generieren kann, soll  $(\tilde{z}(0, \cdot))_* \mathbb{P}$  wie beim VAE und beim ODE<sup>2</sup>VAE in die Form einer Normalverteilung auf  $\mathbb{R}^d$  gezwungen werden. Leider hatten wir nicht mehr genug Zeit die ELBO in diesem Fall herzuleiten und arbeiten mit provisorischen Verlustfunktionen für das neue Modell. Man könnte daher argumentieren, dass unser Modell teilweise eher einem Autoencoder ähnelt, als einem Variational Autoencoder.

### 2.3.2 Lernen von SDEs

Sei  $\boldsymbol{\mu} : \mathbb{R}^d \rightarrow \mathbb{R}^d$  und  $\boldsymbol{\sigma} : \mathbb{R}^d \rightarrow \mathbb{R}^{d \times n}$ . Für Prozesse  $X : [0, \tau] \times \Omega \rightarrow \mathbb{R}^d$  und  $W^1, \dots, W^n : [0, \tau] \times \Omega \rightarrow \mathbb{R}$  heißt  $(Z, W)$  starke Lösung der Stochastischen Differentialgleichung (SDE)

$$dZ^i = \boldsymbol{\mu}^i(t, Z_t) dt + \sum_{j=1}^n \boldsymbol{\sigma}^{i,j}(t, Z_t) dW_s^j ,$$

falls  $W^1, \dots, W^n$  unabhängige Brownsche Bewegungen sind und die Formel

$$Z_t^i - Z_0^i = \int_0^t \boldsymbol{\mu}^i(s, Z_s) ds + \sum_{j=1}^n \int_0^t \boldsymbol{\sigma}^{i,j}(s, Z_s) dW_s^j \quad (1)$$

für jedes  $i \leq d$  erfüllt ist. Wir sind hier nur an dem Fall, dass  $\boldsymbol{\mu}(t, x)$  und  $\boldsymbol{\sigma}(t, x)$  nicht von  $t$ , sondern nur von  $x$  abhängen, interessiert. Die Lösungen solcher SDEs nennt man Ito-Diffusionen.

Nach [10] Thm. 21.4 existiert unter den Voraussetzungen

$$\begin{aligned} \|\boldsymbol{\mu}(x)\|_2 + \|\boldsymbol{\sigma}(x)\|_2 &\leq C_1 (1 + \|x\|_2) \\ \|\boldsymbol{\mu}(x) - \boldsymbol{\mu}(y)\|_2 + \|\boldsymbol{\sigma}(x) - \boldsymbol{\sigma}(y)\|_2 &\leq C_2 \|x - y\|_2 \end{aligned} \quad (2)$$

für jede Startverteilung  $Z_0$  mit endlichem zweiten Moment und beliebige Brownsche Bewegungen.  $W^1, \dots, W^n$  eine eindeutige Lösung  $Z = Z(W)$  der oben beschriebenen SDE. Mit Hilfe des Euler-Maruyama-Verfahrens lässt sich diese Lösung approximieren. Dazu teilen wir das Intervall  $[0, \tau]$  in  $m$  viele Stücke  $[t_k^m, t_{k+1}^m]$  mit  $\Delta t = \frac{\tau}{m}$  und  $t_k^m = k\Delta t^m = \frac{k\tau}{m}$ . Wir definieren rekursiv

$$Z_{m,t_k^m}^i := Z_{m,t_{k-1}^m}^i + \boldsymbol{\mu}^i(Z_{m,t_{k-1}^m}) \Delta t + \sum_{j=1}^n \boldsymbol{\sigma}^{i,j}(Z_{m,t_{k-1}^m}) \underbrace{(W_{t_k}^j - W_{t_{k-1}}^j)}_{\sim \mathcal{N}(0, \Delta t)}$$

mit  $Z_{m,0} = Z_0$ . Da Brownsche Bewegungen unabhängige Segmente haben, können wir analog  $(\varepsilon_{j,k})_{j \leq n, k \in \mathbb{N}}$  *uiv.*  $\sim N(0, 1)$  definieren und schreiben:

$$\tilde{Z}_{m,t_k^m}^i := \tilde{Z}_{m,t_{k-1}^m}^i + \boldsymbol{\mu}^i(\tilde{Z}_{m,t_{k-1}^m}) \Delta t^m + \sum_{j=1}^n \boldsymbol{\sigma}^{i,j}(\tilde{Z}_{m,t_{k-1}^m}) \sqrt{\Delta t^m} \varepsilon_{j,k} .$$

Solche  $\tilde{X}_m$  lassen sich nun leicht mit dem Computer generieren und wir wissen nach Euler-Maruyama, dass ein  $c > 0$  existiert, sodass für jedes  $t \in [0, \tau]$  gilt:

$$\mathbb{E}[|Z_{m,t} - Z_t|_2] \leq \frac{c}{\sqrt{m}} .$$

Ein einfacher Weg nun die Funktionen  $\boldsymbol{\mu}$  aus Realisierungen von  $(Z_{t_0^m}, \dots, Z_{t_m^m})$  approximativ zu lernen ist nun aus der Sicht des Computers den Prozess  $Z$  mit seiner Euler-Maruyama Approximation  $Z_m$  gleich zu setzen. Man würde also annehmen, dass  $Z_{t_k^m} = Z_{m,t_k^m}$  mit

$$Z_{m,t_k^m}^i = Z_{m,t_{k-1}^m}^i + \boldsymbol{\mu}^i(Z_{m,t_{k-1}^m}) \Delta t^m + \sum_{j=1}^n \boldsymbol{\sigma}^{i,j}(Z_{m,t_{k-1}^m}) \sqrt{\Delta t^m} \varepsilon_{j,k} .$$

Wenn man nun  $\boldsymbol{\mu}$  und  $\boldsymbol{\sigma}$  lernen wollte, indem man versucht mit der rechten Seite der Gleichung die linke Seite vorherzusagen, würde nur  $\boldsymbol{\mu}$  richtig trainiert werden. Um  $\boldsymbol{\sigma}$  richtig zu trainieren nutzen wir etwas mehr Theorie.

Die Quadratische Variation und Covariation von Semimartingalen werden in [10] Prop. 19.28 und Thm. 19.31 definiert. Nach Lemma 19.50 wissen wir, dass

$$\sum_{k=1}^m (Z_{t_k^m}^i - Z_{t_{k-1}^m}^i)(Z_{t_k^m}^l - Z_{t_{k-1}^m}^l) \xrightarrow{m \rightarrow \infty} p [Z^i, Z^l]_\tau$$

Weiter wissen wir nach den Eigenschaften der Covariation und  $Z$ , dass

$$\begin{aligned} [Z^i, Z^l]_\tau &= \left[ \int_0^\tau \boldsymbol{\mu}^i(Z_s) ds + \sum_{j=1}^n \int_0^\tau \boldsymbol{\sigma}^{i,j}(Z_s) dW_s^j, \int_0^\tau \boldsymbol{\mu}^l(Z_s) ds + \sum_{j=1}^n \int_0^\tau \boldsymbol{\sigma}^{l,j}(Z_s) dW_s^j \right] \\ &\stackrel{19.44 \& 19.45}{=} \left[ \sum_{j=1}^n \int_0^\tau \boldsymbol{\sigma}^{i,j}(Z_s) dW_s^j, \sum_{j=1}^n \int_0^\tau \boldsymbol{\sigma}^{l,j}(Z_s) dW_s^j \right] \\ &= \sum_{j,j'=1}^n \left[ \int_0^\tau \boldsymbol{\sigma}^{i,j}(Z_s) dW_s^j, \int_0^\tau \boldsymbol{\sigma}^{l,j'}(Z_s) dW_s^{j'} \right] \\ &\stackrel{19.37}{=} \sum_{j,j'=1}^n \int_0^\tau \boldsymbol{\sigma}^{i,j}(Z_s) \boldsymbol{\sigma}^{l,j'}(Z_s) d \underbrace{[W^j, W^{j'}]}_{=1_{j=j'} s}_s = \sum_{j=1}^n \int_0^\tau \boldsymbol{\sigma}^{i,j}(Z_s) \boldsymbol{\sigma}^{l,j}(Z_s) ds \end{aligned}$$

Indem wir nun den unteren Term durch  $A_{i,l} := \sum_{j=1}^n \Delta t^m \sum_{k=1}^m \boldsymbol{\sigma}^{i,j}(Z_{t_k^m}) \boldsymbol{\sigma}^{l,j}(Z_{t_k^m})$  und den oberen Term  $[Z^i, Z^l]_\tau$  wie in der oberen Überlegung durch  $B_{i,l} := (Z_{t_k^m}^i - Z_{t_{k-1}^m}^i)(Z_{t_k^m}^l - Z_{t_{k-1}^m}^l)$  approximieren, können wir die ganze Matrix  $\boldsymbol{\sigma} \in \mathbb{R}^{d \times n}$  trainieren, indem wir den Abstand zwischen  $A$  und  $B$  bestrafen.

Wir trainieren hier effektiv nur  $\boldsymbol{\sigma} \boldsymbol{\sigma}^* \in \mathbb{R}^{d \times d}$ , aber viel besser kann man  $\boldsymbol{\sigma}$  auch nicht bestimmen, da man z.B. bei zwei unabhängigen Brownschen Bewegungen  $W^1, W^2$  man nicht zwischen  $\sqrt{2}W_1, W_1 + W_2, W_1 - W_2, \sqrt{2}W_2$  anhand ihrer Verteilung unterscheiden kann.

### 2.3.3 Aufbau des $\text{SDE}^M\text{VAE}$ -Modells

Sei  $(\mathbb{P}, \Omega, \mathcal{A})$  ein Wahrscheinlichkeitsraum und  $\tilde{x} : [0, \tau] \times \Omega \rightarrow \mathbb{R}^D$  ein stetiger Prozess. Für  $0 = t_0 < \dots < t_m = \tau$  definieren wir die Zufallsvariablen  $x_k := \tilde{x}(t_k, \cdot)$ . Wir gehen der Einfachheit halber davon aus, dass die Abstände  $t_k - t_{k-1}$  alle gleich sind, also  $t_k = k \frac{\tau}{m} = k \Delta t$ .

Gegeben seien Daten  $(x_1^p, \dots, x_m^p)_{p \in \mathbb{N}}$  *uiv.*  $\sim (x_1, \dots, x_m)_* \mathbb{P}$ . Wir definieren einen Encoder  $g : \mathbb{R}^D \rightarrow \mathbb{R}^d$  und einen Decoder  $f : \mathbb{R}^d \rightarrow \mathbb{R}^D$  für  $d << D$ . Dabei ist  $g(x)$  gegeben  $x$  nicht deterministisch, sondern wird (bedingt) unabhängig aus einer  $d$ -dimensionalen Normalverteilung  $\mathcal{N}(\mu(x), \Sigma(x))$  gezogen. Wir schreiben formell  $\mathbb{P}(g(x_k) | x_k) = \mathcal{N}(\mu(x_k), \Sigma(x_k))$  für alle  $k \leq m$ .

Die Funktionen  $\mu : \mathbb{R}^d \rightarrow \mathbb{R}^d$ ,  $\Sigma : \mathbb{R}^d \rightarrow \text{Diag}(d \times d)$  sowie  $f : \mathbb{R}^d \rightarrow \mathbb{R}^D$  werden durch CNNs dargestellt. Für unseren genauen Aufbau für Anwendung auf den rotating-MNIST Datensatz siehe Sektion 4.

Wir definieren die Zufallsvariablen  $z_0, \dots, z_m : \Omega \rightarrow \mathbb{R}^d$  durch  $z_k = z_k^{(0)} := g(x_k)$ . Analog benennen wir die encodierten Daten  $z_k^p(\omega) = z_k^{(0),p}(\omega) := g_\omega(x_k^p(\omega)) \in \mathbb{R}^d$ .

Wir wollen  $z_k^{(0)}$  interpretieren, als seien sie Momentaufnahmen

$$z_k^{(0)} = \tilde{z}^{(0)}(t_k, \cdot) = (\tilde{z}^{(0,1)}(t_k, \cdot), \dots, \tilde{z}^{(0,d)}(t_k, \cdot))$$

eines Prozesses

$$Z_t(\omega) = \begin{pmatrix} \tilde{z}^{(0,1)}(t, \omega) & \dots & \tilde{z}^{(0,d)}(t, \omega) \\ \vdots & & \vdots \\ \tilde{z}^{(M-1,1)}(t, \omega) & \dots & \tilde{z}^{(M-1,d)}(t, \omega) \end{pmatrix} \in \mathbb{R}^{M \times d},$$

der die Lösung einer noch unbekannten SDE ist. Um die SDE zu lernen, definieren wir  $\boldsymbol{\mu} : \mathbb{R}^{M \times d} \rightarrow \mathbb{R}^{m \times d}$  und  $\boldsymbol{\sigma} : \mathbb{R}^{M \times d} \rightarrow \mathbb{R}^{m \times d \times n}$  als Neuronale Netzwerke. Um diese Netzwerke trainieren zu können, benötigen wir auch für  $0 < l < M$  empirische Daten  $(z_1^{(l),p}, \dots, z_m^{(l),p})_{p \in \mathbb{N}}$ . Diese könnten durch weitere Encoder entstehen, wie es bereits im ODE<sup>2</sup>VAE für den Startwert  $v(0, \cdot)$  gemacht wird. Falls  $M$  eher klein ist, oder von einer geringen 'Unsicherheit'  $\boldsymbol{\sigma}$  ausgegangen werden kann, kann man die  $(z_1^{(l),p}, \dots, z_m^{(l),p})_{p \in \mathbb{N}}$  auch als empirische Ableitungen  $z_k^{(l),p} := \frac{1}{\Delta t} (z_{k+1}^{(l-1),p} - z_k^{(l-1),p}) \in \mathbb{R}^d$  definieren. Die Vorstellung als Annäherung einer Ableitung ist hier eigentlich falsch, da Lösungen von SDEs mit positivem  $\boldsymbol{\sigma}$  nirgends differenzierbar sind, aber wie wir schon bei der Theorie zum Lernen von SDEs gesehen hatten, ist  $\mathbb{E}[\frac{1}{\Delta t} (z_{k+1}^{(l-1),p} - z_k^{(l-1),p}) | z_k^{(l-1),p}] \approx \boldsymbol{\mu}(z_k^{(l-1),p})$ , also eignen sich die  $z_k^{(l),p}$  zumindest bei großen Datensätzen um SDEs höherer Ordnung zu lernen.

Für unser Modell definieren wir nun einen 'Reconstructor'  $R_{\mu, \sigma} : \mathbb{R}^{M \times d} \rightarrow \prod_{k=0}^m \mathbb{R}^{M \times d}$ , der zu gegebenen Anfangswerten  $(z_0^0, \dots, z_0^{M-1}) = Z_0$  mit Hilfe der Funktionen  $\mu, \sigma$  das Euler-Maruyama-Verfahren anwendet um  $Z_{m,t_1}, \dots, Z_{m,t_m} \in \mathbb{R}^{M \times d}$  (Notation aus Theorie zum Lernen von SDEs) nacheinander zu konstruieren.

Wir verwenden dabei zur Rekonstruktion besonders  $R_{\mu, 0}$ , da für  $\sigma = 0$  das Euler-Maruyama-Verfahren mit dem Euler-Verfahren übereinstimmt und wir damit  $z_{1,rec}^{(l),p}, \dots, z_{m,rec}^{(l),p} := Z_{m,t_1}, \dots, Z_{m,t_m}$  glatter als die eingegebenen  $z_1^{(l),p}, \dots, z_m^{(l),p}$  rekonstruieren können.

Wir definieren die folgenden Verlustfunktionen:

- Reconstruction-Loss:

$$L_r := H((f(z_0^{(0)}), \dots, f(z_m^{(0)})_* \mathbb{P}), (x_0, \dots, x_m)_* \mathbb{P})$$

Dabei steht  $H$  für die Binäre Kreuzentropie zwischen den Verteilungen.

- Latent-Reconstruction-Loss:

$$L_{lr} := \mathbb{E} \left[ \left( \sum_{k=0}^m \sum_{i=1}^d (z_k^{(0,i)} - z_{k,rec}^{(0,i)})^2 \right)^{\frac{1}{2}} \right]$$

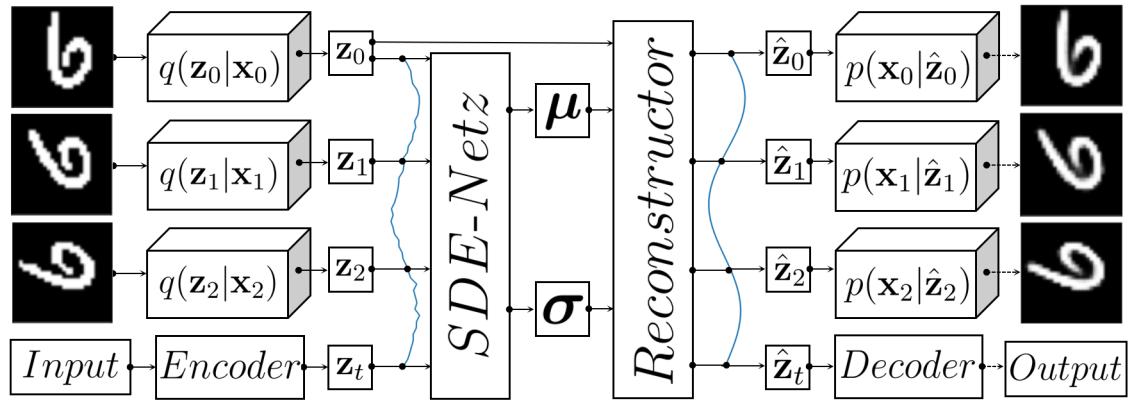
- Latent-Pointwise-Loss:

$$L_{lp} := \mathbb{E} \left[ \left( \sum_{k=0}^{m-1} \sum_{l=1}^{M-1} \sum_{i=1}^d (\mu^{l,i}(z_k) - \Delta t(z_{k+1}^{(l-1,i)} - z_k^{(l-1,i)}))^2 \right)^{\frac{1}{2}} \right]$$

- Latent-Covariance-Loss: Wir betrachten die oben definierten  $Z_{t_k}$  als Vektoren in  $\mathbb{R}^{Md}$ . Seien  $A_{i,l}$  und  $B_{i,l}$  wie sie in der Theorie zum lernen von SDEs definiert werden.

$$L_{lcv} := \mathbb{E} \left[ \left( \sum_{i,l=1}^d (A_{i,l} - B_{i,l})^2 \right)^{\frac{1}{2}} \right]$$

Während des Trainings werden diese Verlustfunktionen, die eigentlich von den Zufallsvariablen  $(x_0, \dots, x_m)$  abhängen, Batch-weise als Mittelwert über die gegebenen Daten  $(x_1^p, \dots, x_m^p)_{p \in \mathbb{N}} \sim (x_1, \dots, x_m)_* \mathbb{P}$  approximiert. Die Gewichtung der Verlustfunktionen muss vom Datensatz abhängig gewählt werden.



Hier sieht man eine vereinfachte Darstellung des Modells für  $M = 1$ . Beim Training werden  $z_i$  direkt als  $\hat{z}_i$  verwendet, wobei der mittlere Block immer noch Einfluss auf die Verlustfunktion hat, damit  $z_0, z_1, \dots$  die Form einer SDE annehmen und diese Gelernt wird.

Für  $M > 1$  müssen zusätzliche Encoder eingefügt werden, die immer 2 oder mehr aufeinanderfolgende Bilder erhalten.

## 3 Empirische Untersuchung

### 3.1 Die verwendeten Datensätze

#### 3.1.1 Rotating MNIST



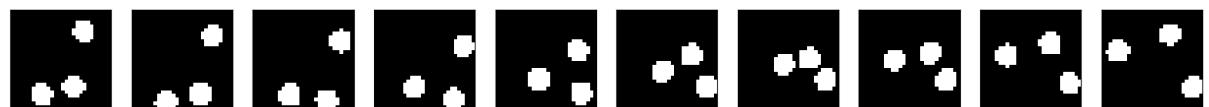
Bei dem Datensatz *rotatingMNIST* handelt es sich um eine Modifikation des bekannten MNIST-Datensatzes. Die Grauwerte der Bilder des MNIST-Datensatzes wurden von uns zunächst binärisiert, um durch eine Bernoulli-Verteilung im Decoder dargestellt werden zu können. Hierbei wurden die Farbwerte 0 bis 127 auf den Wert 0 abgebildet und die Werte 128 bis 255 auf den Wert 1.

Ein Datenpunkt  $\mathbf{x}_i \in \{0,1\}^{28 \times 28 \times 10}$  aus dem Datensatz enthält eine Sequenz von 10 Frames, von einem Frame auf das nachfolgende Frame rotiert die entsprechende Ziffer um  $36^\circ$  gegen den Uhrzeigersinn. Insgesamt wird so eine Rotation um  $360^\circ$  dargestellt.

Der Trainingsdatensatz besteht aus 60000 Sequenzen und der Testdatensatz aus 10000 Sequenzen. Der Code zum erzeugen des Datensatz wurde von uns selbst geschrieben und erlaubt es, falls gewünscht auch eine andere Frameanzahl festzulegen.

Für den SDE<sup>M</sup>VAE benutzen wir Sequenzen bestehend aus 20 Frames anstatt 10.

#### 3.1.2 Bouncing Balls



Der Datensatz *bouncingBalls* besteht aus Bildsequenzen, die Bewegungen dreier gleichgroßer zweidimensionaler Bälle enthalten. Die Bälle befinden sich zum Anfangszeitpunkt in zufälliger Position im Bild und bewegen sich mit normiertem Beschleunigungsvektor in zufällige Richtungen. Dabei können sie miteinander und den Bildrändern kollidieren. Die zugrundeliegenden physikalischen Gesetze einer elastischen Kollisionen zwischen den Bällen und mit den Bildrändern können in [11] nachgelesen werden und sollen von unserem Modell erlernt werden.

Ein Datenpunkt  $\mathbf{x}_i \in \{0,1\}^{28 \times 28 \times 10}$  aus dem Datensatz enthält 10 Frames. Der Wert 1 bedeutet, dass sich an der jeweiligen Stelle auf dem Frame ein Ball befindet.

Obwohl dieser Datensatz dem des Papers zum ODE<sup>2</sup>VAE sehr ähnelt ist er für das Modell komplexer zu erlernen, da bei uns zum einen weniger Frames enthalten sind, zwischen

diesen aber mehr Zeit vergeht.

Der Trainingsdatensatz besteht aus 60000 Sequenzen und der Testdatensatz aus 10000 Sequenzen. Der Code zum erzeugen des Datensatz wurde von uns selbst geschrieben und ermöglicht es, falls erwünscht auch eine andere Anzahl der Bälle, Frames, Auflösung und unterschiedlichen Radius zu wählen.

### 3.1.3 AVI-Motion



Der Datensatz *AVI-Motion* besteht aus Bildsequenzen, die drei menschliche Bewegungen (klatschen, boxen und winken) darstellen. Der ursprüngliche Datensatz enthält zu diesen Bewegungen je 4 Videosequenzen von jeweils 25 unterschiedlichen Personen, die jeweils mehrere Sekunden lang sind und eine Auflösung von  $160 \times 120$  Pixel haben. Der Datensatz kann von der Website [12] heruntergeladen werden. Wir haben die Videosequenzen auf eine Auflösung von  $60 \times 60$  reduziert, indem wir die Personen mittig in einem  $120 \times 120$  großen Ausschnitt zentriert haben und die Auflösung schließlich durch Mean-Pooling verringert haben.

Ein Datenpunkt  $\mathbf{x}_i \in [0, 1]^{60 \times 60 \times 8}$  enthält 8 Frames. Die Farbwerte sind stetige Grauwerte zwischen 0 (schwarz) und 1 (weiß). Wir wollen auf diesem Datensatz testen, ob sich das Modell des ODE<sup>2</sup>VAE dazu eignet, reale Videosequenzen zu modellieren.

Der Trainingsdatensatz besteht dazu aus 17472 Sequenzen und der Testdatensatz aus 112 Sequenzen. Im Gegensatz zu den anderen Datensätzen legen wir die Batchsize hier auf  $M = 112$  fest (da  $112|17472$ ).

### 3.1.4 SDE Ball



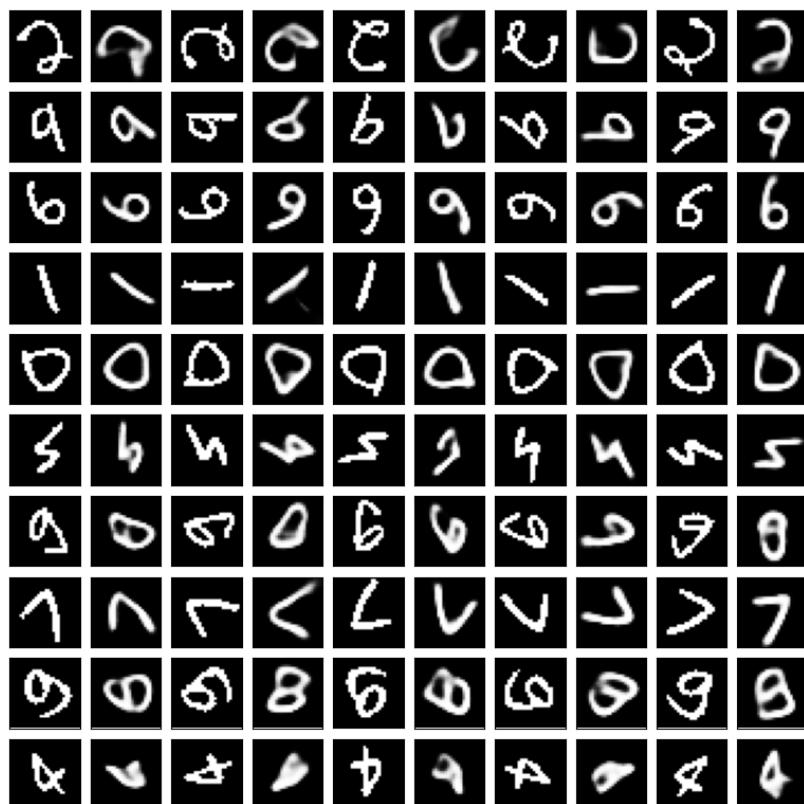
Dieser einfache Datensatz zeigt einen Ball, der sich nach dem Pfad einer SDE mit einer Brownschen Bewegung ( $n = 1$ ) nach oben und unten bewegt. Die SDE von Grad 2 hat die Form

$$\begin{aligned} \boldsymbol{\mu}_{\text{org}} : \mathbb{R}^2 &\rightarrow \mathbb{R}^2 ; \quad \boldsymbol{\mu}_{\text{org}}(x, y) = (y, -x) \\ \boldsymbol{\sigma}_{\text{org}} : \mathbb{R}^2 &\rightarrow \mathbb{R}^{2 \times 1} ; \quad \boldsymbol{\sigma}_{\text{org}}(x, y) = (0.2, 0.1) . \end{aligned}$$

Für die Position des Balls wird nur die erste Koordinate der Lösung der SDE betrachtet. Falls  $\sigma$  gleich 0 wäre, würde es sich hierbei um die Differentialgleichung handeln, deren Lösung durch Sinus und Kosinus gegeben sind. Der Pfad des Balls beschreibt also anschaulich eine 'chaotische' Sinuskurve. Als Startwert der SDE wird immer  $(0, 1)$  gewählt. Ein Datenpunkt  $\mathbf{x}_i \in \{0, 1\}^{28 \times 28 \times 50}$  aus dem Datensatz enthält 50 Frames. Der Trainingsdatensatz besteht aus 3000 Sequenzen und der Testdatensatz aus 1000 Sequenzen.

## 3.2 Ergebnisse

### 3.2.1 Methode 1: Lineare Interpolation im latenten Raum



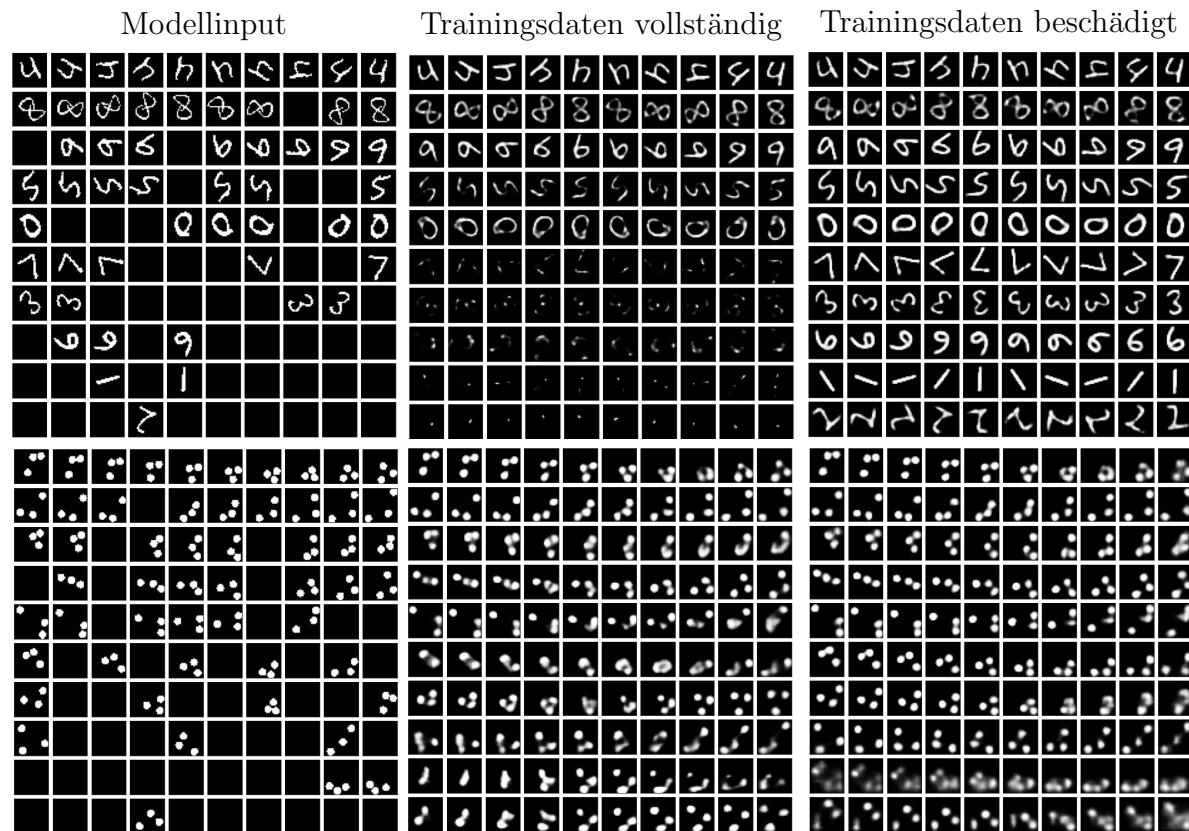
In der Abbildung sehen wir zehn Bildsequenzen aus dem Datensatz rotatingMNIST. Die Bilder zu den Zeitpunkten  $t = 0, 2, 4, 6, 8$  sind Modellinput und die Bilder den Zeitpunkten  $t = 1, 3, 4, 7, 9$  sind Rekonstruktionen der fehlenden Bilder mittels linearer Interpolation im latenten Raum.

Das Modell wird auf den Datenpunkten für 50 Epochen trainiert, indem die *time series*  $\mathbf{x}_{0:T}$  in voneinander unabhängige Datenpunkte  $\{\mathbf{x}_t\}_{t=0}^T$  aufgeteilt werden. Um die fehlenden Bilder zu rekonstruieren, werden die vom Decoder bestimmten latenten Darstellungen durch eine Gerade  $\mathbf{z}_{t+1} = \mathbf{z}_t + a(\mathbf{z}_{t+2} - \mathbf{z}_t)$  verbunden, welche zum gewünschten Zeitpunkt, bei uns  $a = 0.5$  und  $a = 1.5$  für  $t = 9$ , ausgewertet wird. Das Ergebnis wird als latente

Darstellung für das fehlende Bild verwendet und mithilfe des Decoders rekonstruiert. Theoretisch lassen sich so zwischen zwei Bildern beliebig viele Rekonstruktionen anfertigen.

Auffallend ist, dass diese Methode nicht immer gute Rekonstruktionen liefert. Je nach Sequenz schwanken die Rekonstruktionen zwischen minimalen Abweichungen bis hin zu stark verschwommenen oder falsch rekonstruierten Bildern. Hierbei können wir gut sehen, dass die Cluster im latenten Raum nicht immer konvex angeordnet sind. So wird beispielsweise im linken Bild bei der siebten und neunten Sequenz eine 9 zu einer 0 bzw. einer 8 rekonstruiert. Sobald ein Cluster eine annähernd konvexe Form hat erhalten wir, wie beispielweise in der dritten, vierten, fünften oder achten Sequenz, eine gute Rekonstruktion. Die Methode ist daher auf Datensätzen mit vielen Eigenschaften, oder mit Bildern, die in ihren Eigenschaften zu weit auseinanderliegen ungeeignet, da die Interpolationsgerade oft auch durch ein anderes Cluster geht.

### 3.2.2 Methode 2: Vanilla-VAE



In den sechs Abbildungen sehen wir je zehn Bildsequenzen aus den Datensätzen rotatingMNIST und bouncingBalls. Ganz Links ist der Modellinput abgebildet. Wir haben aus diesem jeweils 0 bis 9 Frames entfernt, die von unserem Modell rekonstruiert werden

sollen.

Dafür haben wir einen VAE für 100 Epochen auf den Datensätzen trainiert, wobei die *time series* als gesamtes als Modellinput verwendet.

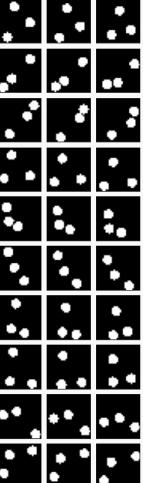
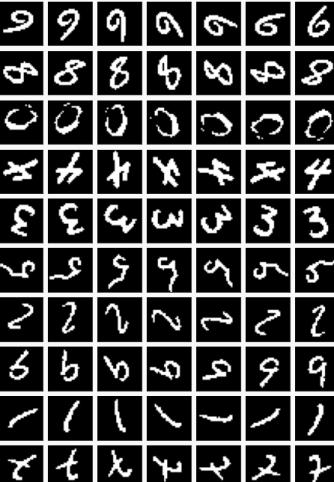
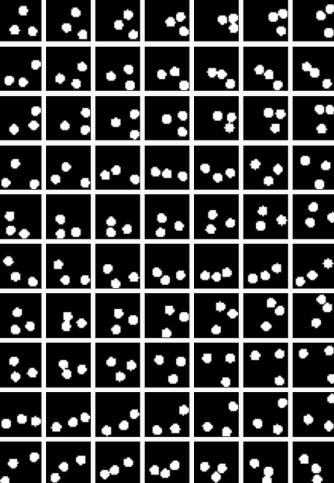
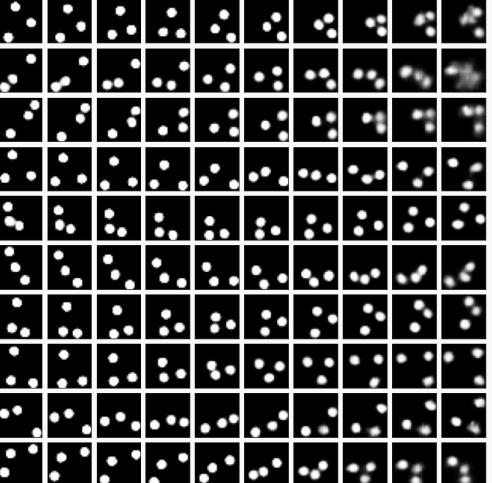
Mittig sind die Rekonstruktionen eines VAE abgebildet, der auf einem unbeschädigten Datensatz trainiert wurde und nach dem Training beschädigte Sequenzen des Testdatensatz rekonstruieren sollte. Die Idee dahinter ist, dass dem Modell diese Daten unbekannt sind und sie nach Abbildung in den latenten Raum daher vom Decoder vollständig rekonstruiert werden. Die Rekonstruktionen werden dabei aber immer schlechter, je mehr Daten aus dem ursprünglichem Bild fehlen.

Wir behelfen uns daher mit einem Trick. Wir beschädigen auch den Trainingsdatensatz, trainieren den VAE aber darauf die gesamte Sequenz und nicht das beschädigte Inputbild zu rekonstruieren. Wie man in den Abbildungen rechts erkennen kann führt das zu einer deutlichen Verbesserung der Modellperformance.

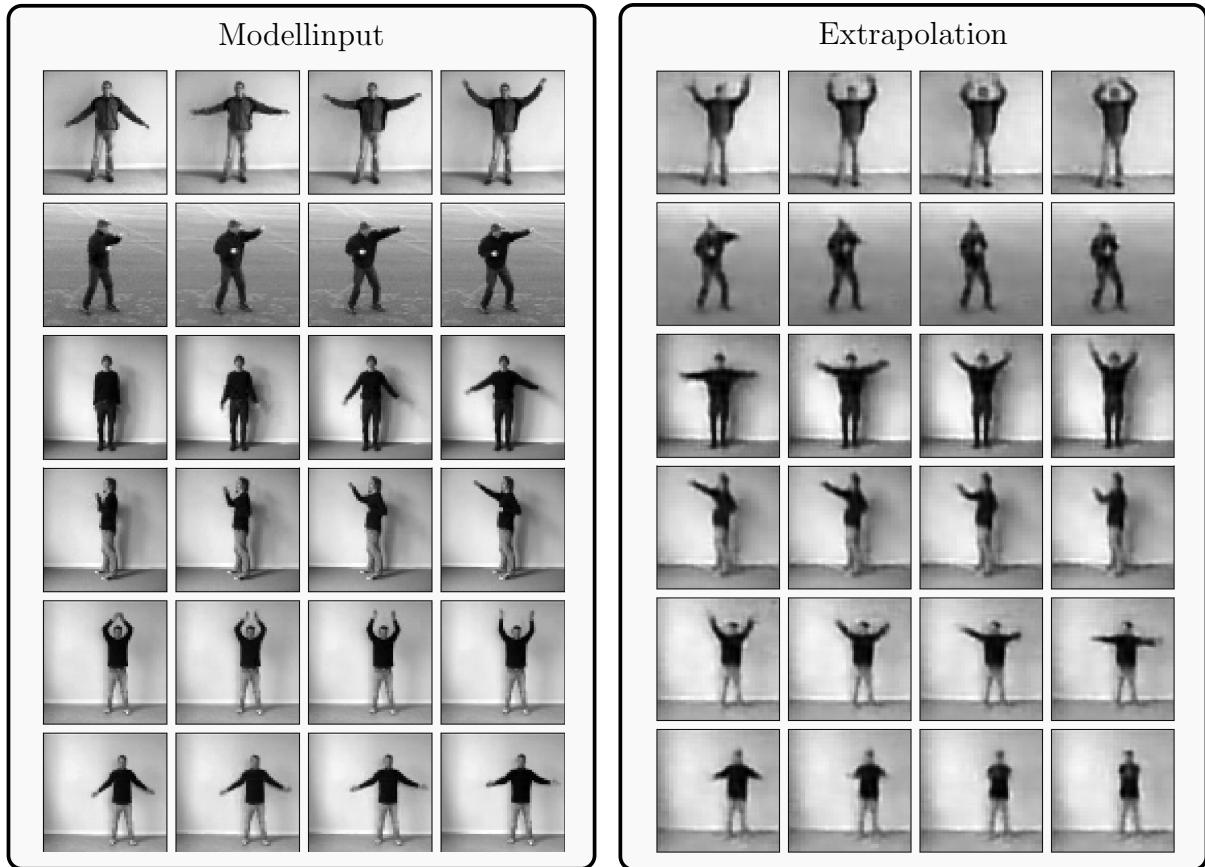
### 3.2.3 Methode 3: ODE<sup>2</sup>VAE

Auf diesen Abbildungen sind die Rekonstruktionen eines ODE<sup>2</sup>VAE, auf den Datensätzen rotatingMNIST und bouncingBalls zu erkennen. Das Modell wurde für 40 Epochen auf einem unbeschädigten Datensatz trainiert und erhält zum anfertigen der Rekonstruktionen aber nur die ersten drei Bilder der *time series*. Dieser Modellinput ist ganz links abgebildet. Daneben ist zum besseren einordnen der Ergebnisse die Groundtruth zu sehen, die das Modell zwar nicht sieht, von ihm aber bei einer perfekten Vorhersage aus den drei Inputdaten errechnet werden soll. Rechts ist schließlich die Rekonstruktion des Modellinputs und ab dem vierten Frame die Extrapolation zu erkennen. Wie man gut sehen kann ist die Modellperformance auf dem Datensatz rotatingMNIST hervorragend, während auf bouncingBalls, der chaotischen Natur des Datensatzes geschuldet, auch einige Vorhersagen in den hinteren Frames nicht so gut aussehen. Dies dürfte daran liegen, dass sich anfänglich kleine Fehler in der Vorhersage mit fortlaufender Zeit immer weiter verschlimmern.

Wir vermuten deswegen, dass das Modell davon profitieren könnte die Priorverteilung beispielsweise durch Inverse Autoregressive Flow von einer Normalverteilung zu einer aussagekräftigeren Verteilung zu transformieren. So könnte sicher gestellt werden, dass die Startposition sehr genau encodiert wird, was einem aufsummieren von Fehlern entgegenwirken würde. Dennoch von allen Methoden liefert dieses Modell auf den Datensätzen bereits auch so die besten Ergebnisse. Sollen nur ein oder zwei Frames rekonstruiert werden und sind die vorherigen drei Frames bekannt ist eine Rekonstruktion annährend fehlerfrei. Auf dem Datensatz AVI-Motion werden ebenfalls gute Ergebnisse erzielt. Die Bewegungen werden vom Modell gut vervollständigt, wir haben im Gegensatz zu den anderen

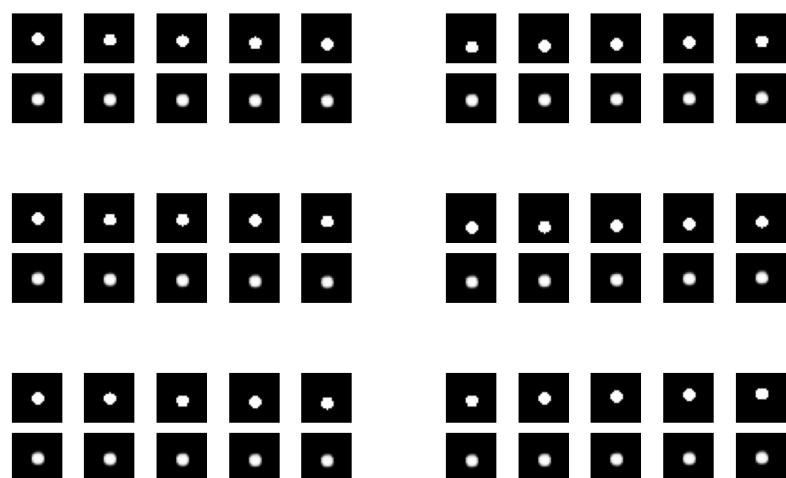
Modellinput	Groundtruth	Rekonstruktion und Extrapolation
 	 	 

Datensätzen dem Modell zur Bestimmung von  $v_0$  jedoch vier Bilder gezeigt. Die Extrapolation ist zwar etwas unscharf, könnte aber durch das Training von beispielsweise Adversarial Variational Autoencodern weiter an Qualität gewinnen.



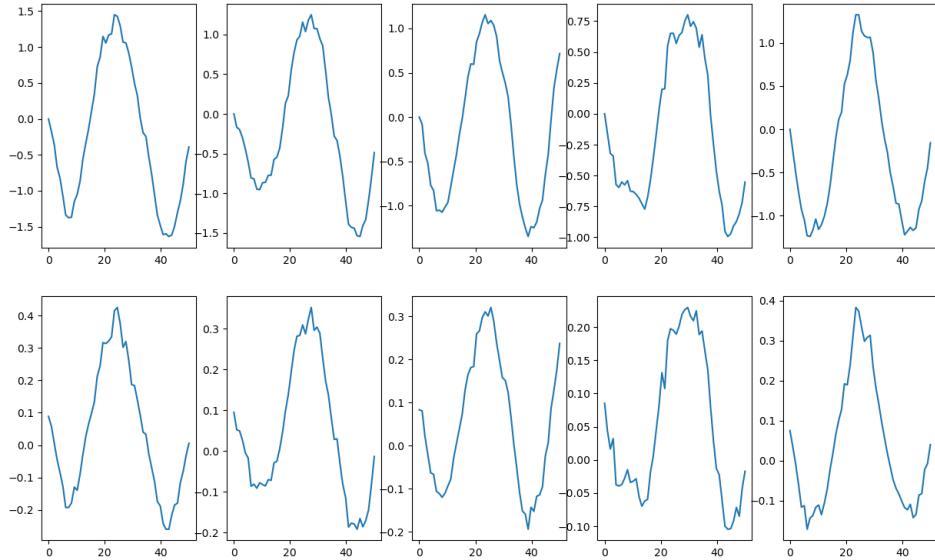
### 3.2.4 Methode 4: SDE<sup>M</sup>VAE

Für den SDE-Ball-Datensatz erhalten wir schon mit einer latenten Dimension ( $d = 1$  und  $M = 2$ ) nach 3 Epochen gemeinsamen Trainings gefolgt von 5 Epochen zusätzlichen Trainings für das SDE-Netzwerk die folgenden Ergebnisse.



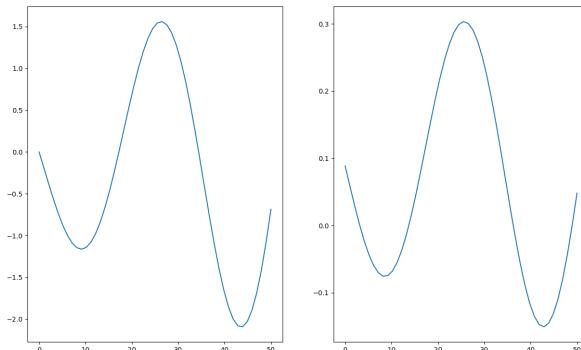
Auf dem Bild sind originale Frames über ihren Rekonstruktionen zu sehen. Links sieht man dabei die ersten 5 und rechts die letzten 5 Frames der *time series*.

Da man die Bewegung des Balls nicht sehr gut erkennen kann, zeigen wir stattdessen die latente Darstellung gegenüber den tatsächlichen Positionen des Balls.



Oben stehen die tatsächlichen Positionen des Balls, nach denen das Bild gezeichnet wurde. Darunter stehen die latenten Darstellungen der einzelnen Frames. Jeder Plot beschreibt genau eine Time-Series.

Wenn wir nun überprüfen wollen, ob wie die SDE auch richtig gelernt haben, können wir die Rekonstruktionen  $R_{\mu_{\text{org}},0}((0, 1))$  und  $R_{\mu,0}((0, 1))$  vergleichen:



Wir beobachten hier eine erstaunlich genaue Approximation der SDE. Vermutlich hätten

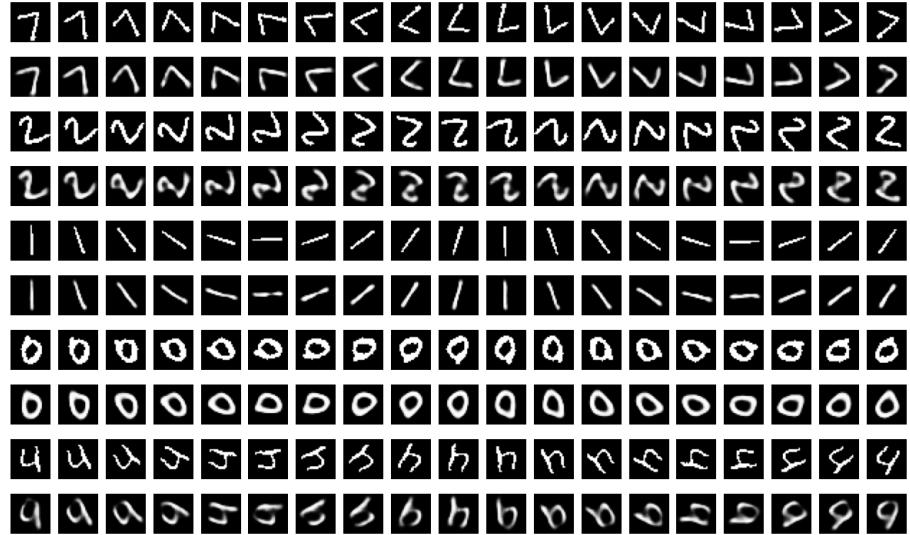
wir weniger als 3000 *time series* zum Trainieren verwenden können. Die Kurven sehen hier nicht ganz wie Sinuskurven aus, da wir die ODE einer Sinuskurve diskret auf 50 Stellen approximieren. Wenn wir mehr Frames (also mehr Approximations-Stellen pro Zeiteinheit) hätten, sähen die Kurven eher wie Sinuskurven aus.

Für die Verlustfunktion haben wir festgestellt, dass die Verlust-Gewichtung

$$20 L_r + 5 L_{lr} + 10 L_{lp} + 1 L_{lcv}$$

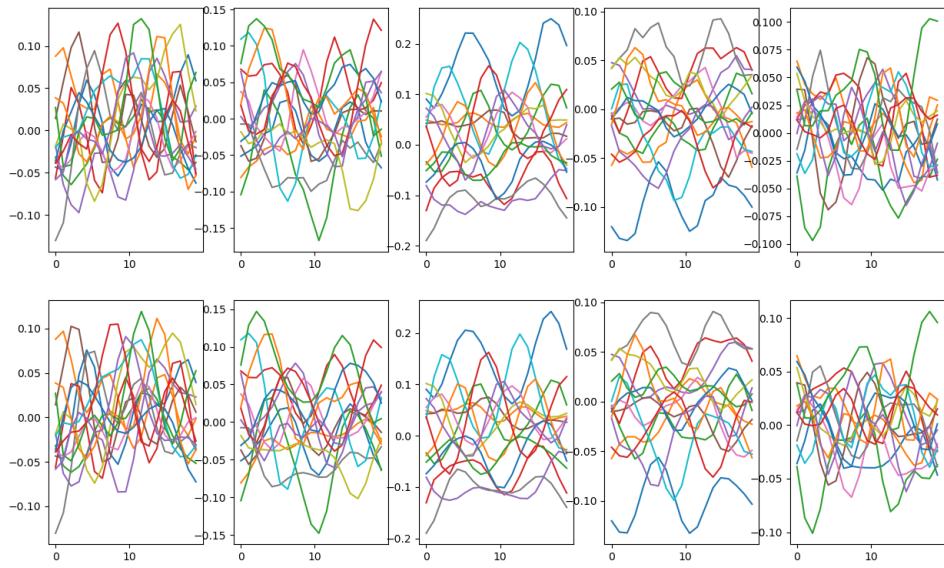
(für Notation vergleiche Seite 26) recht gut funktioniert.

Bei der Anwendung auf den rotating-MNIST-Datensatz mit 20 Frames erhalten wir mit  $d = 15$  und  $M = 2$  nach 5 Epochen gemeinsamen Trainings gefolgt von 10 Epochen Training für das SDE-Netzwerk die folgenden Rekonstruktionen.

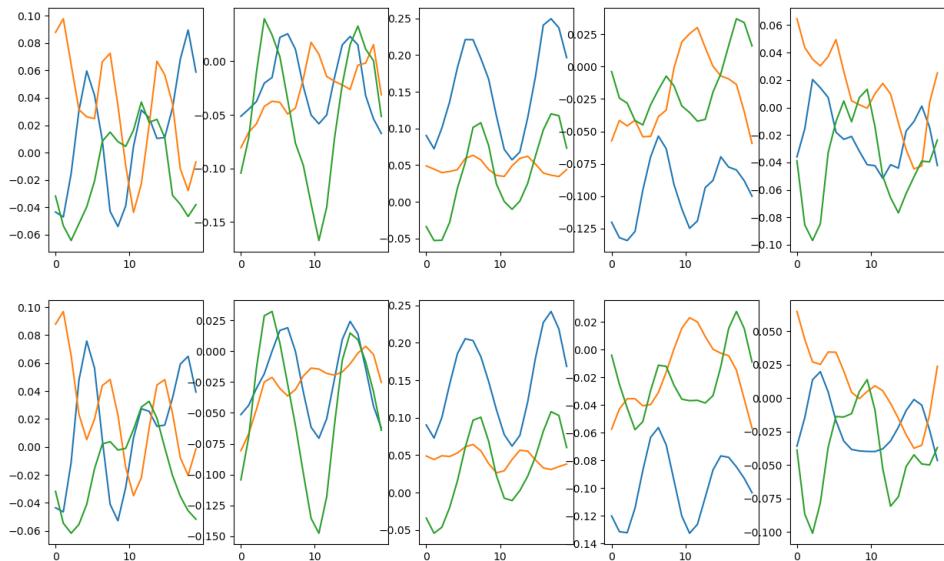


Hier sieht man abwechselnd die originalen *time series* und darunter die Rekonstruktion nach glatter rekonstruierten latenten Darstellungen.

Das Modell scheint also auch seine Aufgabe als Verallgemeinerung des ODE<sup>2</sup>VAE zu erfüllen. Wir können auch die glattere Rekonstruktion der latenten Darstellungen betrachten.



Hier sieht man oben die encodierten 15 latenten Dimensionen und darunter ihre glatten Rekonstruktionen nach der gelernten SDE. Ein Plot steht dabei für genau eine *time series*. Da man mit so vielen geplotteten Linien schlecht die Glättung sehen kann, haben wir hier zufällig drei der Dimensionen ausgewählt.



## 4 Modellstrukturen

Diese Modellstruktur haben wir für die Implementierung des Vanilla-VAE's verwendet. Als Input dient hier die gesamte Videosequenz. Falls nur ein einzelnes Frame betrachtet werden soll, (z.B. auf dem normalen MNIST-Datensatz oder für lineare Interpolation) muss im Input-Layer des Encoders und im Output-Layer des Decoders die Channelsize von 10 auf 1 reduziert werden.

	<b>Encoder</b>	<b>Outputgröße</b>	<b>Parameter</b>	<b>Fenstergröße</b>	<b>Aktivierung</b>
<b>1.</b>	Input-Layer	(28, 28, 10)	0	-	-
<b>2.</b>	Conv2D	(28, 28, 32)	2912	$3 \times 3$ ; padding	relu
<b>3.</b>	Max-Pooling2D	(14, 14, 32)	0	$2 \times 2$	-
<b>4.</b>	Conv2D	(12, 12, 64)	18496	$3 \times 3$	relu
<b>5.</b>	Max-Pooling2D	(6, 6, 64)	0	$2 \times 2$	-
<b>6.</b>	Conv2D	(4, 4, 128)	73856	$3 \times 3$	relu
<b>7.</b>	Flatten	2048	0	-	-
<b>8.</b>	Dense-Layer	256	262272	-	relu
<b>9.</b>	Dense-Layer	20 + 20	10280	-	-
<b>10.</b>	Split-Layer ( $\mu$ , $\log \sigma$ )	(20, 20)	0	-	-
<b>11.</b>	$z = \mu + \epsilon \cdot \exp(\log \sigma)$	20	0	-	-

	<b>Decoder</b>	<b>Outputgröße</b>	<b>Parameter</b>	<b>Fenstergröße</b>	<b>Aktivierung</b>
<b>1.</b>	Input-Layer	20	0	-	-
<b>2.</b>	Dense-Layer	256	5376	-	relu
<b>3.</b>	Dense-Layer	2048	526336	-	relu
<b>4.</b>	Reshape-Layer	(4, 4, 128)	0	-	-
<b>5.</b>	Conv2DTranspose	(6, 6, 64)	73792	$3 \times 3$	relu
<b>6.</b>	Up-Sampling2D	(12, 12, 64)	0	$2 \times 2$	-
<b>7.</b>	Conv2DTranspose	(14, 14, 32)	18464	$3 \times 3$	relu
<b>8.</b>	Up-Sampling2D	(28, 28, 32)	0	$2 \times 2$	-
<b>9.</b>	Conv2DTranspose	(28, 28, 10)	5130	$3 \times 3$ ; padding	sigmoid

Die Modellstruktur des ODE<sup>2</sup>VAE für die Datensätze rotatingMNIST und bouncingBalls. Im ODE-Net verwenden wir als Aktivierungsfunktion den Tangens hyperbolicus, da die ReLu-Funktion zum Modellieren von Differentialgleichungen, aufgrund  $\text{relu}'(x) = 0$ , ungeeignet ist.

	<b>Positionencoder</b>	<b>Outputgröße</b>	<b>Parameter</b>	<b>Fenstergröße</b>	<b>Aktivierung</b>
1.	Input-Layer	(28, 28, 1)	0	-	-
2.	Conv2D	(28, 28, 32)	320	$3 \times 3$ ; padding	relu
3.	Max-Pooling2D	(14, 14, 32)	0	$2 \times 2$	-
4.	Conv2D	(12, 12, 64)	18496	$3 \times 3$	relu
5.	Max-Pooling2D	(6, 6, 64)	0	$2 \times 2$	-
6.	Conv2D	(4, 4, 128)	73856	$3 \times 3$	relu
7.	Flatten	2048	0	-	-
8.	Dense-Layer	256	262272	-	relu
9.	Dense-Layer	20 + 20	10280	-	-
10.	Split-Layer ( $\mu, \log \sigma$ )	(20, 20)	0	-	-
11.	$s = \mu + \epsilon \cdot \exp(\log \sigma)$	20	0	-	-

	<b>Velocityencoder</b>	<b>Outputgröße</b>	<b>Parameter</b>	<b>Fenstergröße</b>	<b>Aktivierung</b>
1.	Input-Layer	(28, 28, 3)	0	-	-
2.	Conv2D	(28, 28, 32)	896	$3 \times 3$ ; padding	relu
3.	Max-Pooling2D	(14, 14, 32)	0	$2 \times 2$	-
4.	Conv2D	(12, 12, 64)	18496	$3 \times 3$	relu
5.	Max-Pooling2D	(6, 6, 64)	0	$2 \times 2$	-
6.	Conv2D	(4, 4, 128)	73856	$3 \times 3$	relu
7.	Flatten	2048	0	-	-
8.	Dense-Layer	256	262272	-	relu
9.	Dense-Layer	20 + 20	10280	-	-
10.	Split-Layer ( $\mu, \log \sigma$ )	(20, 20)	0	-	-
11.	$v = \mu + \epsilon \cdot \exp(\log \sigma)$	20	0	-	-
	Concatenate $\mathbf{z} = (\mathbf{s}, \mathbf{v})$	(2, 20)	0	-	-

	<b>ODE-Net</b>	<b>Outputgröße</b>	<b>Parameter</b>	<b>-</b>	<b>Aktivierung</b>
1.	Input-Layer	(2, 20)	0	-	-
2.	Flatten	40	0	-	-
3.	Dense-Layer	100	4100	-	tanh
4.	Dense-Layer	100	10100	-	tanh
5.	Dense-Layer	20	2020	-	-

	<b>Decoder</b>	<b>Outputgröße</b>	<b>Parameter</b>	<b>Fenstergröße</b>	<b>Aktivierung</b>
1.	Input-Layer	20	0	-	-
2.	Dense-Layer	256	5376	-	relu
3.	Dense-Layer	2048	526336	-	relu
4.	Reshape-Layer	(4, 4, 128)	0	-	-
5.	Conv2DTranspose	(6, 6, 64)	73792	$3 \times 3$	relu
6.	Up-Sampling2D	(12, 12, 64)	0	$2 \times 2$	-
7.	Conv2DTranspose	(14, 14, 32)	18464	$3 \times 3$	relu
8.	Up-Sampling2D	(28, 28, 32)	0	$2 \times 2$	-
9.	Conv2DTranspose	(28, 28, 1)	513	$3 \times 3$ ; padding	sigmoid

Die Modellstruktur des ODE<sup>2</sup>VAE für den Datensatz AVI-Motion.

	<b>Positionencoder</b>	<b>Outputgröße</b>	<b>Parameter</b>	<b>Fenstergröße</b>	<b>Aktivierung</b>
<b>1.</b>	Input-Layer	(60, 60, 1)	0	-	-
<b>2.</b>	Conv2D	(56, 56, 32)	823	$5 \times 5$	relu
<b>3.</b>	Max-Pooling2D	(14, 14, 32)	0	$2 \times 2$	-
<b>3.</b>	Conv2D	(28, 28, 32)	9248	$3 \times 3$ ; padding	relu
<b>4.</b>	Max-Pooling2D	(14, 14, 32)	0	$2 \times 2$	-
<b>5.</b>	Conv2D	(12, 12, 64)	18496	$3 \times 3$	relu
<b>6.</b>	Max-Pooling2D	(6, 6, 64)	0	$2 \times 2$	-
<b>7.</b>	Conv2D	(5, 5, 128)	32896	$2 \times 2$	relu
<b>8.</b>	Flatten	3200	0	-	-
<b>9.</b>	Dense-Layer	384	1229184	-	relu
<b>10.</b>	Dense-Layer	30 + 30	23100	-	-
<b>11.</b>	Split-Layer( $\mu, \log \sigma$ )	(30, 30)	0	-	-
<b>12.</b>	$s = \mu + \epsilon \cdot \exp(\log \sigma)$	30	0	-	-
	<b>Velocityencoder</b>	<b>Outputgröße</b>	<b>Parameter</b>	<b>Fenstergröße</b>	<b>Aktivierung</b>
<b>1.</b>	Input-Layer	(60, 60, 4)	0	-	-
<b>2.</b>	Conv2D	(56, 56, 64)	6464	$5 \times 5$	relu
<b>3.</b>	Max-Pooling2D	(14, 14, 64)	0	$2 \times 2$	-
<b>3.</b>	Conv2D	(28, 28, 64)	36928	$3 \times 3$ ; padding	relu
<b>4.</b>	Max-Pooling2D	(14, 14, 64)	0	$2 \times 2$	-
<b>5.</b>	Conv2D	(12, 12, 128)	73856	$3 \times 3$	relu
<b>6.</b>	Max-Pooling2D	(6, 6, 128)	0	$2 \times 2$	-
<b>7.</b>	Conv2D	(5, 5, 256)	131328	$2 \times 2$	relu
<b>8.</b>	Flatten	6400	0	-	-
<b>9.</b>	Dense-Layer	640	4096640	-	relu
<b>10.</b>	Dense-Layer	30 + 30	38460	-	-
<b>11.</b>	Split-Layer ( $\mu, \log \sigma$ )	(30, 30)	0	-	-
<b>12.</b>	$v = \mu + \epsilon \cdot \exp(\log \sigma)$	30	0	-	-
Concatenate $\mathbf{z} = (\mathbf{s}, \mathbf{v})$		(2, 30)	0	-	-
	<b>ODE-Net</b>	<b>Outputgröße</b>	<b>Parameter</b>	-	<b>Aktivierung</b>
<b>1.</b>	Input-Layer	(2, 30)	0	-	-
<b>2.</b>	Flatten	60	0	-	-
<b>3.</b>	Dense-Layer	120	7320	-	tanh
<b>4.</b>	Dense-Layer	120	14520	-	tanh
<b>5.</b>	Dense-Layer	30	3630	-	-
	<b>Decoder</b>	<b>Outputgröße</b>	<b>Parameter</b>	<b>Fenstergröße</b>	<b>Aktivierung</b>
<b>1.</b>	Input-Layer	20	0	-	-
<b>2.</b>	Dense-Layer	384	11904	-	relu
<b>3.</b>	Dense-Layer	3200	1232000	-	relu
<b>4.</b>	Reshape-Layer	(4, 4, 128)	0	-	-
<b>5.</b>	Conv2DTranspose	(6, 6, 64)	32832	$2 \times 2$	relu
<b>6.</b>	Up-Sampling2D	(12, 12, 64)	0	$2 \times 2$	-
<b>7.</b>	Conv2DTranspose	(14, 14, 32)	18464	$3 \times 3$	relu
<b>8.</b>	Up-Sampling2D	(28, 28, 32)	0	$2 \times 2$	-
<b>9.</b>	Conv2DTranspose	(28, 28, 32)	16416	$3 \times 3$ ; padding	relu
<b>10.</b>	Up-Sampling2D	(56, 56, 32)	0	$2 \times 2$	-
<b>11.</b>	Conv2DTranspose	(60, 60, 1)	801	$5 \times 5$	sigmoid

Die Modellstruktur des SDE-VAE für die Datensätze SDE-Ball und rotatingMNIST.

<b>Encoder</b>	<b>Outputgröße</b>	<b>Parameter</b>	<b>Fenstergröße</b>	<b>Aktivierung</b>
1. Input-Layer	(28, 28, 1)	0	-	-
2. Conv2D	(28, 28, 16)	160	$3 \times 3$ ; padding	relu
3. Max-Pooling2D	(14, 14, 16)	0	$2 \times 2$	-
4. Conv2D	(12, 12, 32)	4640	$3 \times 3$	relu
5. Max-Pooling2D	(6, 6, 32)	0	$2 \times 2$	-
6. Conv2D	(4, 4, 64)	18496	$3 \times 3$	relu
7. Flatten	1024	0	-	-
8. Dense-Layer	128	131200	-	relu
9. Dense-Layer	$d + d$	$2d + 129$	-	-
10. Split-Layer ( $\mu, \log \sigma$ )	(d, d)	0	-	-
11. $z = \mu + \epsilon \cdot \exp(\log \sigma)$	d	0	-	-

<b>Decoder</b>	<b>Outputgröße</b>	<b>Parameter</b>	<b>Fenstergröße</b>	<b>Aktivierung</b>
1. Input-Layer	d	0	-	-
2. Dense-Layer	128	$(d+1)128$	-	relu
3. Dense-Layer	1024	132096	-	relu
4. Reshape-Layer	(4, 4, 64)	0	-	-
5. Conv2DTranspose	(6, 6, 32)	18464	$3 \times 3$	relu
6. Up-Sampling2D	(12, 12, 32)	0	$2 \times 2$	-
7. Conv2DTranspose	(14, 14, 16)	4624	$3 \times 3$	relu
8. Up-Sampling2D	(28, 28, 16)	0	$2 \times 2$	-
9. Conv2DTranspose	(28, 28, 1)	257	$3 \times 3$ ; padding	sigmoid

<b><math>\mu</math>-Netz</b>	<b>Outputgröße</b>	<b>Parameter</b>	<b>Fenstergröße</b>	<b>Aktivierung</b>
1. Input-Layer	(M, d)	0	-	-
2. Flatten	M d	0	-	-
3. Dense-Layer	20 M	$(Md + 1) 20M$	-	tanh
4. Dense-Layer	20 M	$(20M + 1) 20M$	-	tanh
5. Dense-Layer	M d	$(20M + 1) Md$	-	tanh
6. Reshape-Layer	(M, d)	0	-	-

<b><math>\sigma</math>-Netz</b>	<b>Outputgröße</b>	<b>Parameter</b>	<b>Fenstergröße</b>	<b>Aktivierung</b>
1. Input-Layer	(M, d)	0	-	-
2. Flatten	M d	0	-	-
3. Dense-Layer	20 M n	$(Md + 1) 20Mn$	-	tanh
4. Dense-Layer	20 M n	$(20Mn + 1) 20Mn$	-	tanh
5. Dense-Layer	M d n	$(20Md + 1) Mdn$	-	tanh
6. Reshape-Layer	(M, d, n)	0	-	-

## 5 Quellenangabe

### Literatur

- [1] Diederik P. Kingma, Max Welling, *Auto-Encoding Variational Bayes*, 2013, arXiv:1312.6114v10
- [2] Diederik P. Kingma, Max Welling, *An Introduction to Variational Autoencoders*, 2019, arXiv:1906.02691v3
- [3] Çağatay Yıldız, Markus Heinonen, Harri Lähdesmäki, *ODE<sup>2</sup>-VAE: Deep generative second order ODEs with Bayesian neural networks*, 2019, arXiv:1905.10994v2
- [4] <https://gabrielhuang.gitbooks.io/machine-learning/content/reparametrization-trick.html>
- [5] Carl Doersch, *Tutorial on Variational Autoencoders*, 2016, arXiv:1606.05908, Ähnliche Grafik zu finden auf S.5
- [6] Kaare B. Petersen, Michael S. Pedersen, *The Matrix Cookbook*, 2012, <http://matrixcookbook.com>
- [7] Ricky T. Q. Chen, Yulia Rubanova, Jesse Bettencourt, David Duvenaud, *Neural Ordinary Differential Equations*, 2019, arXiv:1806.07366v5
- [8] Will Grathwohl, Ricky T. Q. Chen, Jesse Bettencourt, Ilya Sutskever, David Duvenaud, *FFJORD: Free-Form Continuous Dynamics for Scalable Reversible Generative Models* 2018, arXiv:1810.01367v3
- [9] <http://blog.shakirm.com/2015/09/machine-learning-trick-of-the-day-3-hutchinsons-trick/>
- [10] <https://www.stochastik.uni-freiburg.de/professoren/pfaffelhuber/inhalte/2020zyklus>
- [11] [https://en.wikipedia.org/wiki/Elastic\\_collision](https://en.wikipedia.org/wiki/Elastic_collision)
- [12] <https://www.csc.kth.se/cvap/actions/>