Variational Autoencoders: Uma Revisão de Métodos e Aplicações

Ademir Tomaz, Carlos Souza, Juliana Carvalho, Raphael Levy

11 de julho de 2023

Conteúdo

1 Abstract

This paper provides a comprehensive overview of disentangled representations, an essential concept in machine learning, especially with respect to Variational Autoencoders (VAEs). Disentangled representations aim to capture the underlying factors of variation in the data, where different dimensions in the learned representation correspond to distinct generative factors, thereby allowing the data to be better understood and manipulated. In the context of VAEs, this disentanglement is particularly critical, as VAEs function by encoding input data into a lower-dimension latent space, with the goal of capturing these salient factors. By doing so, VAEs not only provide a robust method for data compression, but also pave the way for improved interpretability and performance in downstream tasks. This paper takes a deep dive into this process, exploring both the theoretical underpinnings and practical implications of disentangled representations within VAEs.

We also highlight the application of disentangled representations on the Fashion MNIST dataset. This dataset, composed of 10 categories of clothing items, poses unique challenges and opportunities for disentanglement, with factors of variation including the type of clothing, the style, the presence of patterns, among others. Our exploration underscores the ability of disentangled VAEs to isolate these factors and

generate new samples, enabling greater precision and creativity in fashion design and recommendation systems.

Keywords: Disentangled Representations, Variational Autoencoders, Fashion MNIST, Machine Learning, Data Compression, Interpretability, Generative Models.

2 Resumo

Este artigo fornece uma visão abrangente de representações desemaranhadas, um conceito essencial em aprendizado de máquina, especialmente no que diz respeito a Variational Autoencoders (VAEs). As representações desemaranhadas visam captar os fatores subjacentes de variação nos dados, onde diferentes dimensões na representação aprendida correspondem a distintos fatores generativos, permitindo assim que os dados sejam melhor compreendidos e manipulados. No contexto dos VAEs, esse desemaranhamento é particularmente crítico, pois os VAEs funcionam codificando dados de entrada em um espaço latente de menor dimensão, com o objetivo de capturar esses fatores salientes. Ao fazer isso, os VAEs não apenas fornecem um método robusto para compactação de dados, mas também abrem caminho para melhor interpretabilidade e desempenho em tarefas de downstream. Este artigo mergulha profundamente nesse processo, explorando tanto os fundamentos teóricos quanto as implicações práticas das representações desemaranhadas dentro dos VAEs.

Também destacamos a aplicação de representações desemaranhadas no conjunto de dados Fashion MNIST. Este conjunto de dados, composto por 10 categorias de artigos de vestuário, apresenta desafios únicos e oportunidades de desvendamento, com fatores de variação que incluem o tipo de roupa, o estilo, a presença de padrões, entre outros. Nossa exploração ressalta a capacidade dos VAEs desvendados de isolar esses fatores e gerar novas amostras, permitindo maior precisão e criatividade no design de moda (aplicação no fashionMNIST).

Palavras-chave: Representações Desemaranhadas, Autoencoders Variacionais, Fashion MNIST, Machine Learning, Compressão de Dados, Interpretabilidade, Modelos Gerativos.

3 Introdução

As Disentagled Representations (representações desentrelaçadas/desemaranhadas; DR) surgiram como um conceito crucial no campo do aprendizado de máquina. A capacidade de desentrelaçar os fatores subjacentes de variação em um conjunto de dados, de forma que cada dimensão corresponda a um fator gerativo distinto, tem implicações significativas para a interpretação e manipulação de dados. Este conceito encontrou ampla aplicação em vários domínios, desde visão computacional e processamento de linguagem natural até saúde e condução autônoma.

A importância das DR tornou-se ainda mais proeminente com o aumento da complexidade e diversidade dos dados. À medida que os conjuntos de dados se tornam maiores e mais multidimensionais, surge a necessidade de um mecanismo capaz de isolar e entender os fatores distintos que contribuem para a variância nos dados. As representações desentrelaçadas fornecem uma poderosa solução para este problema, oferecendo um meio de capturar a estrutura inerente dos dados e criar modelos mais robustos, interpretáveis e eficientes.

Os Variational Autoencoders (VAEs) são uma classe de modelos generativos, e se destacam como uma das principais ferramentas nesse empreendimento. Os VAEs operam codificando os dados de entrada em um espaço latente de menor dimensão e decodificando-os de volta, capturando assim efetivamente as características essenciais dos dados. No processo, eles se esforçam para aprender uma representação desentrelaçada no espaço latente, melhorando ainda mais sua utilidade e eficiência.

Neste trabalho, vamos nos aprofundar na teoria e aplicação das representações desentrelaçadas no contexto dos VAEs. Esta exploração será ancorada na aplicação desses conceitos no conjunto de dados Fashion MNIST, uma coleção de 70.000 imagens em escala de cinza de 10 categorias de itens de moda. Como ilustraremos, as representações desentrelaçadas têm o potencial de melhorar significativamente a precisão e criatividade em tarefas como design de moda e sistemas de recomendação, entre outros.

Para garantir que nossa exploração e implementação sejam baseadas em uma sólida base teórica, começaremos com uma revisão abrangente da literatura existente sobre representações desentrelaçadas, VAEs e suas aplicações. Isso nos fornecerá o contexto e a compreensão necessários para implementar e interpretar efetivamente nossos modelos nas etapas subsequentes do nosso trabalho.

4 Fundamentação Teórica

Descrição de Disentangled Representations:

De forma generalizada, podemos definir as disentangled representations como uma técnica de aprendizado não-supervisionado que separa cada feature em variáveis restritas e codifica elas em dimensões distintas, de forma a capturar informações sensíveis e úteis e entendendo relações causais entre as variáveis. Por exemplo, na imagem abaixo é possível visualizar como podemos alterar dimensões específicas e entender como variações específicas modificam a imagem original:

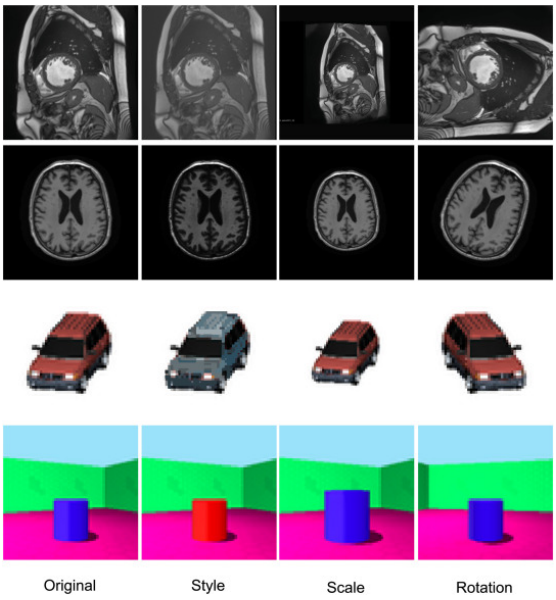

Figura 1: Imagem retirada do artigo "Learning disentangled representations in the imaging domain".

Para definir "desemaranhamento"de uma maneira mais formal, precisamos ter em mente representações latentes, que são os dados comprimidos após serem passados pelo encoder, ou seja, transformando uma informação complexa em uma versão "mais simples", facilitando seu processamento e análise.

Agora, tomando as definições necessárias:

$\mathcal{X}$: variáveis observadas
$\mathcal{Z}$: representações latentes
$\mathcal{Y}$: domínio de saídas do modelo

O aprendizado do modelo é dado por uma função $f : \mathcal{X} \to \mathcal{Y}$, sendo que f pode ser separado em dois componentes:

$$f : E_\phi \circ D_\phi,$$

onde $E_\phi : \mathcal{X} \to \mathcal{Z}$ é o encoder e $D_\phi : \mathcal{Z} \to \mathcal{Y}$ é o decoder, mapeando os dados para uma representação latente intermediária e a representação para os outputs, respectivamente.

Assim, o objetivo do aprendizado do modelo é aprender uma boa representação, que terá equivariâncias, invariâncias e simetrias para diferentes alterações. Vamos definir essas operações:

1. Simetria: uma simetria Ω é uma transformação que mantém aspectos dos dados de entrada, como a categoria de um objeto por exemplo, que não se altera quando deslocamos o objeto.

2. Equivariância: um mapeamento E_ϕ é equivariante com respeito a Ω se existe uma transformação $\omega \in \Omega$ de um input $X \in \mathcal{X}$ que afeta o output $Z \in \mathcal{Z}$ da mesma maneira, ou seja, existe um mapeamento $M_\omega : \mathbb{R}^d \to \mathbb{R}^d$ tal que $E_\phi(M_\omega \circ X) = M_\omega \circ E_\phi(X)\ \forall\ \omega \in \Omega$.

3. Invariância: é um caso especial da equivariância, onde M é a identidade, e $E_\phi(M_\omega \circ X) = E_\phi(X)\ \forall\ \omega \in \Omega$.

Em resumo, usamos equivariância caso desejamos que a saída passe pela mesma transformação aplicada ao dado de entrada, enquanto que para a invariância, desejamos que a saída seja a mesma independente de qualquer transformação aplicada aos dados de entrada.

Partindo agora para os fatores geradores, eles são as variáveis subjacentes que caracterizam a variação dos dados de $\mathcal{X}$, que são observadas ou que esperam-se ser observadas. Sendo assim, representações deveriam possibilitar a decomposição, ou desemaranhamento, dos dados de entrada em fatores distintos, ou seja, cada fator corresponderia a uma única variável de interesse do processo que gerou os dados. Outro ponto necessário para uma definição mais formal dessas representações são as alterações de domínio, que ocorrem quando os dados de treino, validação e teste são extraídos de uma distribuição de probabilidade que é distinta da distribuição usada com os modelos de predição.

Sendo assim, definimos uma **Disantagled Representation**:

Seja $W = W_1 \times W_2$ o espaço de estados sujeito às transformações, por exemplo um espaço 2D com movimentações para cima, baixo, direita e esquerda; ou um espaço 3D de rotações de 90° não comutativas,e Ω um conjunto de transformações.

Na prática, temos acesso a O (o conjunto de observações) que são feitas nos "espaço de estados", e não aos próprios "espaços de estados".

Seja $G = G_1 \times G_2$ o grupo de transição de estados.

Aplicação : $G \times W \to W$ será **desemaranhada** quando G_1 preserva W_2 e G_2 preserva W_1.

Supondo a existência de um processo generativo $b : W \to O$ que leva esse espaço para as observações, e um processo de inferência $h : O \to Z$ levando as observações às representações dos agentes, em geral, assumimos W um conjunto e Z um espaço vetorial. Sendo assim, consideramos a composição $f : W \to Z$, que consegue induzir Ω na representação latente $Z \in \mathcal{Z}$ de forma equivariante, i.e, a função $f : W \to Z$ é uma "disentangled representation"se existir uma aplicação: $G \times Z \to Z$ tal que f seja $G - Equivalente$.

Em resumo: é uma representação que não possui elementos entrelaçados ou interdependentes. Isso implica que a função f preserva a estrutura do grupo G na ação em Z, ou seja, quando a ação de G é aplicada a Z, a função f é equivalente em relação a essa ação.

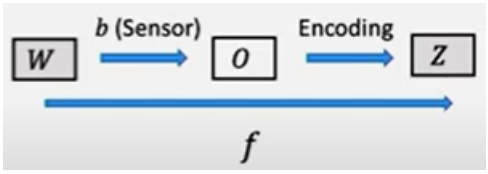

Figura 2: Ilustração

A representação Z é definida como "desemaranhada"caso exista uma decomposição $Z = Z_1 \times \ldots \times Z_n$ tal que uma transformação ω aplicada em Z_i resulta em uma transformação equivalente no domínio $\mathcal{X}$, deixando inalterados quaisquer outros aspectos $Z_j, j \neq i$. Essa definição satisfaz as propriedades desejadas para uma representação desemaranhada:

1. Modularidade: essa propriedade mede se cada dimensão latente codifica não mais do que um único fator gerador. Cada código latente é associado a um único fator gerador e não é afetado por mudanças nos demais, sendo implicitamente necessário que cada variável latente seja separável.

2. Compacidade ("Compactness"): mede se cada fator gerador de dados é codificado por uma única dimensão latente. Se a dimensionalidade do espaço latente for grande demais, informação ruidosa

ou redundante pode ser codificada na representação latente. Essa definição não é bem acordada entre autores, visto que várias das abordagens mais recentes requerem que essa propriedade seja válida, enquanto que outras abordagens aceitam que fatores geradores individuais sejam codificados por múltiplas dimensões latentes.

3. Explicitude: mede se os valores de todos os fatores generativos podem ser decodificados da representação usando uma transformação linear. É a propriedade mais forte das três, dado que engloba dois pontos de extrema importância: que uma representação desemaranhada capture informação sobre todos os fatores generativos (completude), e que essa informação seja linearmente decodificável (informatividade).

A modularidade é o requerimento mais essencial para um modelo de aprendizado de representação desemaranhada, por garantir que cada feature desemaranhada corresponda a um fator generativo que as features desemaranhadas distintas sejam independentes umas das outras. A compacidade requer que o modelo mapeie os dados originais de alta dimensão para features de baixa dimensão, atuando como um compressor dos dados.

Separando a explicitude em seus pontos integrantes, a completude requer que a representação desemaranhada expresse informação sobre todas as variáveis dos dados observados, enquanto que a informatividade se refere ao fato de que features desemaranhadas podem recuperar os valores dos fatores generativos através do modelo.

Descrição de VAEs:

Conceitos preliminares: Autoencoders são uma classe de redes neurais que são normalmente utilizadas para aprender representações de dados de alta dimensão, como imagens, em um espaço de menor dimensão. Esse procedimento é feito em duas etapas:

Enconders: Nessa etapa, nosso dataset entra por varias camadas convolucionais de forma que suas dimensões sejam comprimidas, sem haver perdas significativas (denotamos por f)

Decoders: Nessa etapa vamos reconstruir a imagem original a partir da imagem reduzida (denotamos por h)

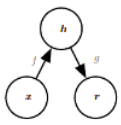

Figura 3: Ilustração de um autoencoder

Denoised Autoenconders: Quando tratamos de imagens com muito ruído, ao tentar reconstruir uma imagem "limpa"sem eles, podemos treinar o autoencoder para reduzir esses ruídos.

Variational Autoenconders: Autoencoders Variacionais são profundamente enraizados em métodos variacionais da estatística Bayesiana, em que, diferente do Autoencoder tradicional, ao invés de mapear o input para um vetor fixo, queremos mapeá-lo para uma distribuição parametrizada por θ (p_θ). Dessa forma, ele não possui apenas um "gargalo"como no método tradicional, possuindo agora dois vetores, um para média e outro para o desvio padrão como mostrado na figura.

Assim, definido uma priori $p_\theta(z)$, a verossimilhança $p_\theta(x|z)$ e a posteriori $p_\theta(z|x)$, sendo x nosso input e z o vetor codificador latente, e assumindo que conhecemos o parâmetro real θ^* dessa distribuição, podemos gerar uma amostra próxima de um data point real $x^{(i)}$ seguindo esses passos:

1. Primeiro, amostramos $z^{(i)}$ da priori $p_{\theta^*}(z)$
2. Depois, gerar um valor $x^{(i)}$ através da condicional $p_\theta(x|z = z^{(i)})$

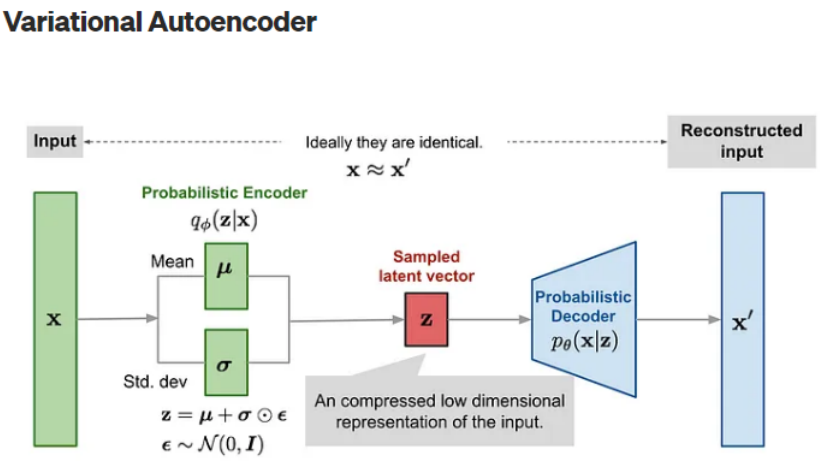

Figura 4: Autoenconder Variacional

No caso, θ^* é o parâmetro ótimo que maximiza a probabilidade de gerar amostras de dados reais:

$$\theta^* = \text{argmax}_\theta \prod_{i=1}^{n} p_\theta(x^{(i)}) = \text{argmax}_\theta \sum_{i=1}^{n} \log p_\theta(x^{(i)})$$

Com isso, temos:

$$p_\theta(x^{(i)}) = \int p_\theta(x^{(i)}|z) p_\theta(z) dz$$

Para facilitar o cálculo de $p_\theta(x^{(i)})$, podemos introduzir uma função de aproximação que tem como saída um código possível dado o input x, parametrizado por ϕ ($q_\phi(z|x)$). Com isso, temos uma estrutura similar a de um autoencoder: a probabilidade condicional define um modelo generativo, similar ao decodificador, então podemos chamar $p_\theta(x|z)$ de decodificador probabilístico, enquanto que a função de aproximação $q_\phi(z|x)$ é o codificador probabilístico.

Outra forma de definir esse tipo de representações é feito por Huang et al., através de um método chamado de "Sufficient and disentangled learning" (SDRL). Considerando um par de vetores $(x, y) \in \mathbb{R}^p \times \mathbb{R}^q$, onde x é o vetor de variáveis de entrada e y o vetor de labels, desejamos encontrar uma representação suficiente e desemaranhada de x.

Assim, dizemos que uma função $g : \mathbb{R}^p \to \mathbb{R}^d,\ d \leq p$ será uma representação suficiente de x se $y \perp\!\!\!\perp x \,|\, g(x)$, ou seja, x e y são condicionalmente independentes dado $g(x)$. Essa condição vale se e somente se a distribuição condicional de y dado x e y dado $g(x)$ forem iguais. Sendo assim, a informação sobre y em x está totalmente codificada por $g(x)$. Essa formulação é uma generalização não-paramétrica da condição básica em redução suficiente de dimensionalidade, em que assumimos $g(x) = B^T x,\ B \in \mathbb{R}^{p \times d},\ B^T B = I_d$. Note que sempre existe tal g, já que tomando $g(x) = x$, a formulação contará com a trivialidade. As classes de representações suficiente que satisfazem a condição acima são dadas por

$$\mathcal{F} = \{g : \mathbb{R}^p \to \mathbb{R}^d, g \text{ mensurável e que satisfaça } y \perp\!\!\!\perp x \,|\, g(x)\},$$

sendo $\mathcal{F}$ chamada uma classe de Fisher dada sua conexão próxima com o conceito de estatística suficiente. Dada uma transformação injetiva e mensurável $T : \mathbb{R}^d \to \mathbb{R}^d$ e $g \in \mathcal{F}$, $T \circ g(x)$ também é suficiente pela propriedade básica da probabilidade condicional. Sendo assim, a classe $\mathcal{F}$ é invariante, dado que se T é injetiva, $T \circ \mathcal{F} = \mathcal{F}$, onde $T \circ \mathcal{F} = T \circ g : g \in \mathcal{F}$. Definimos assim a condição de suficiência.

Agora, dentre as representações suficientes, veremos quais são desemaranhadas. Sendo assim, começamos com as funções dos dados de entrada que são representações suficientes na classe $\mathcal{F}$. Para qualquer representação $g(x)$ suficiente e desemaranhada, vamos definir $\Sigma_g = Var(g(x))$. Dado que os componentes de $g(x)$ são desemaranhados (independentes), Σ_g é uma matriz diagonal e $\Sigma_g^{-1/2} g(x)$ também possui componentes independentes. Dessa maneira, podemos sempre redimensionar $g(x)$ tal que tenha uma matriz

de covariância que é a identidade. Para simplificar a estrutura estatística da representação g, também necessitamos que seja invariante a rotação, ou seja, $Qg(x) = g(x)$ em distribuição para qualquer matriz ortogonal $Q \in \mathbb{R}^{d\times d}$. A classe $\mathcal{F}$ é invariante a rotação em termos de independência condicional, mas nem todos seus elementos são invariantes a rotação em distribuição.

Através da caracterização de Maxwell para distribuições Gaussianas, um vetor de dimensão maior ou igual a 2 com componentes independentes é invariante a rotação em distribuição se e somente se for uma Gaussiana com média 0 e matriz de covariância esférica, ou seja, é diagonal e todos os elementos na diagonal são iguais. Assim, depois de adicionar o fator de escala, para que uma representação suficiente seja desemaranhada e invariante a rotação, ela deve ser distribuída como $\mathcal{N}_d(0, I_d)$. Com isso, seja $\mathcal{M}$ a classe de Maxwell de funções $g : \mathbb{R}^d \to \mathbb{R}^d$, onde $g(x)$ é desemaranhada e invariante a rotação em distribuição. Escrevemos então

$$\mathcal{M} = \{g : \mathbb{R}^p \to \mathbb{R}^d, g(x) \sim \mathcal{N}(0, I_d)\}.$$

Agora, o nosso problema de achar um map de aprendizado suficiente e desemaranhado (SDRM) se torna o problema de achar uma representação em $\mathcal{F} \cap \mathcal{M}$, que é possível garantir que é não-vazia através do lema a seguir:

Lema: seja μ uma medida de probabilidade em $\mathbb{R}^d$. Supondo que tem segundo momento finito e é absolutamente contínua com respeito a uma medida Gaussiana padrão, definida por γ_d, então μ admite um mapeamento de transporte único e ótimo $T : \mathbb{R}^d \to \mathbb{R}^d$ tal que $T_{\#}\mu = \gamma_d \equiv \mathcal{N}(0, I_d)$, onde $T_{\#}\mu$ denota a distribuição pushforward de μ sobre T.

Reconstruction Loss function Outro tópico interessante de ser abordado é a "Reconstruction Loss function"(Função de Perda de Reconstrução) que é basicamente uma medida que quantifica a diferença entre os dados de entrada originais e os dados reconstruídos por um modelo, como um autoencoder. Ela é usada para avaliar o quão bem o modelo está sendo capaz de reconstruir os dados de entrada.

Definição: Vimos que os VAEs têm como objetivo aprender a verossimilhança marginal dos dados nesse processo generativo:

$$max_{\phi,\theta}\mathbb{E}_{q_\phi(z|x)}[logp_\theta(x|z)]$$

Essa equação pode ser escrita como :

$$log[p_\theta(x|z)] = D_{KL}(q(z|x)||p(z)) + \mathcal{L}(\theta, \phi, x, z),$$

onde D_{KL}* denota a divergência KL entre as distribuições p e q. Sendo assim, note que maximizar $\mathcal{L}(\theta, \phi, x, z)$ é equivalente a estabelecer um limite inferior para equação acima:

$$log[p_\theta(x|z)] \geq \mathcal{L}(\theta, \phi, x, z) = \mathbb{E}_{q_\phi(z|x)}[logp_\theta(x|z)] - D_{KL}(q(z|x)||p(z)),$$

onde o termo $\mathbb{E}_{q_\phi(z|x)}[logp_\theta(x|z)]$ será nossa função de perda de reconstrução. [1]

Truque da Reparametrização:
Antes de começar o nosso treinamento, temos que tratar de um problema: no meio desse procedimento, depois do gargalo, temos uma operação de amostragem. Existe um nó que faz uma amostra de uma distribuição e, em seguida, passa essa amostra pelo decodificador. O problema é que você não pode executar uma "back propagation", também não é possível propagar gradientes por meio de um nó de amostragem. Isso é um problema, e para executar seus gradientes por toda a rede e treinar tudo de ponta a ponta usamos um recurso já visto antes, chamado de **Truque da Reparametrização**:

Se você observar o vetor latente que está sendo amostrado, você pode considerar esse vetor como a soma de vários μ's fixos, que são os parâmetro que você está aprendendo, mais alguns de σ's, que também são parâmetros que você está aprendendo, e então multiplicado por um ϵ, da seguinte forma:

$$Z = \mu + \sigma.\epsilon$$

[1] A divergência de Kullback-Leibler (KL) é uma medida usada para quantificar a diferença entre duas distribuições de probabilidade.

onde $\epsilon \sim \mathcal{N}(0, 1)$.

Agora nosso μ e σ são as únicas coisas que realmente queremos treinar, então agora somos capazes de calcular gradientes e realizar "back propagation"para eles. Mas o ϵ, bem, isso não importa muito, porque não queremos alterar esse ϵ nunca mais, dado que é um nó elástico fixo.

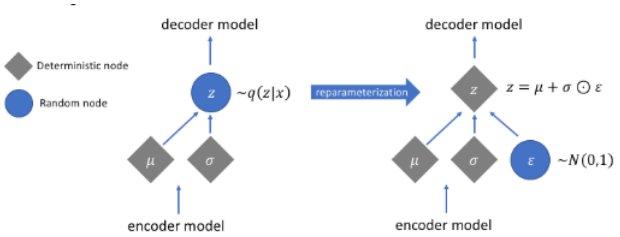

Figura 5: Truque da reparametrização

Desse modo ao invés de ter um nó completamente estocástico que está bloqueando todos os gradientes, dado que não podemos fazer a back propagation nele, nós podemos dividí-lo em duas partes, uma onde podemos fazer a back propagation e outra parte que ainda é estocástica, mas não precismos treiná-la porque ela é fixa.

Disentangled Autoenconder: Com base nos assuntos discutidos acima, sabemos que:

Representação desemaranhada: é um tipo de representação em que diferentes atributos latentes independentes correspondem a diferentes características geradoras dos dados. Por exemplo, em imagens de rostos, uma representação desemaranhada ideal pode ter dimensões separadas para características como cor dos olhos, formato do rosto, idade, expressão facial, etc. A ideia principal é que mudanças em uma única dimensão latente correspondem a mudanças em uma única característica geradora dos dados.

Autoencoder: é uma rede neural que é treinada para tentar copiar sua entrada para sua saída. Internamente, tem uma camada oculta que descreve um código usado para representar a entrada. O autoencoder tem duas partes principais, o **encoder** que transforma a entrada em código, e o **decoder**, que transforma o código de volta na saída. Esta estrutura é útil para aprender representações eficientes dos dados, para compressão de dados e também para redução de ruído.

Agora, vamos ao autoencoder desemaranhado (Disentangled Autoencoder). Este é um tipo de autoencoder projetado para aprender representações desemaranhadas dos dados. Ele faz isso alterando a função de perda do autoencoder para encorajar a separação das características geradoras dos dados nas representações latentes.

Ideia: garantir que os diferentes neurônios em sua distribuição latente sejam não-correlacionados, de forma que todos tentem aprender algo diferente sobre os dados de entrada. Para essa implementação, a única coisa que devemos adicionar é um novo hiperparâmetro β em sua função de perda

$$\mathcal{L}(\theta, \phi, x, z, \beta) = \mathbb{E}_{q_\phi(z|x)}[logp_\theta(x|z)] - \beta D_{KL}(q_\phi(x|z)||p(z))(*)$$

Esse hiperparâmetro β pondera o quanto a divergência KL está presente na função de perda e, consequentemente, na versão desemaranhada do autoencoder. Em geral usamos $\beta > 1$, e portanto essa equação recebe nome de β**-VAE**, enquanto que quando $\beta = 1$ temos o VAE original. O autoencoder só usará uma variável latente específica se ela realmente trouxer algum benefício à compressão. Caso contrário, ele se limitará ao uso normal.

Para ter um entendimento melhor, nas figuras abaixo, podemos ver as ilustrações das arquiteturas de uma rede normal e outra usando uma Disentangled Representation (DR):

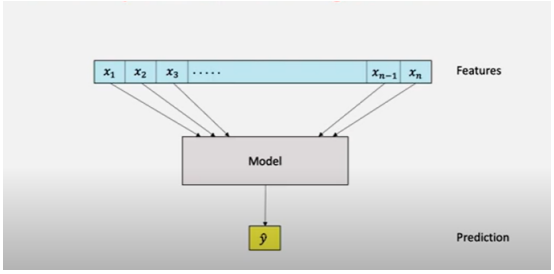

Figura 6: Rede normal

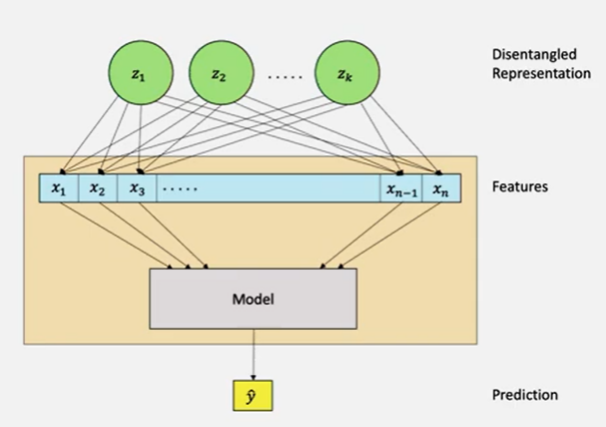

Figura 7: Rede usando disentangled representation

5 Conditional Variational Autoencoder (CVAE)

Por fim, antes de partirmos para a metodologia, devemos definir um outro autoencoder que usamos em nossa implementação: o CVAE. Esse autoencoder tem uma pequena diferença para o VAE padrão, em que nele injetamos a informação da label tanto na fase de codificação quanto na de decodificação. Isso pode ser necessário pois, usando apenas o VAE, nós não temos controle sobre que tipo de dado ele irá gerar. Por exemplo, usando o MNIST e tentando gerar imagens passado $Z \sim \mathcal{N}(0, 1)$ para o decoder, ele pode produzir dígitos aleatórios. As imagens podem ser bem geradas se o autoencoder for bem treinado, mas não podemos ordenar que o VAE gere uma imagem de um dígito específico. Aí entra o CVAE: passando como entrada Y, a label da imagem, queremos como saída X, que será a imagem. Sendo assim, dada a observação y, z é extraído de uma distribuição a priori $P_\theta(z|y)$, e x é gerado da distribuição $P_\theta(x|y, z)$. No VAE simples, a priori é $P_\theta(z)$ e a saída é gerada por $P_\theta(x|z)$.

Assim, temos que o encoder tenta aprender $q_\phi(z|x, y)$, que equivale a aprender representações ocultas dos dados X, ou codificar X na representação oculta condicionada y; enquanto que o decoder tenta aprender $P_\theta(X|z, y)$, que decodifica a representação oculta para o espaço de entrada condicionado por y.

6 Metodologia

Para execução desse projeto vamos fazer comparações computacionais (tempo de processamento), e da qualidade de diferentes implementações de Variational Autoencoders utilizando duas bases de dados:

Para a realização dessa seção realizamos um experimento computacional neste estudo, escolhemos utilizar dois conjuntos de dados bem conhecidos e amplamente utilizados em aprendizado de máquina:

- "Fashion MNIST", o qual consiste em um conjunto de imagens de roupas
- "MNIST", o qual consiste em um conjunto de dígitos escritos manualmente.

Esses conjuntos de dados são eficientes para avaliar modelos de aprendizado de máquina devido à sua simplicidade e, ao mesmo tempo, possibilidade de geração de modelos complexos.

Teste inicial da qualidade de **CVAE** $\times$ β**-VAE**:

Figura 8: MNIST

Figura 9: VAE Mnist

Figura 10: CVAE Fashion Mnist

6.1 Código do Ademir: CVAE

Esta primeira parte do código é dedicada à importação de módulos necessários e à configuração do ambiente de computação.

```
import torch
import torch.nn as nn
import torch.optim as optim
torch.nn.functional as F
from torchvision import datasets, transforms

# Verifique se CUDA está disponível
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print("Using device:", device)
```

Definimos a arquitetura do $CVAE$ na classe $CVAE$, que inclui tanto a **CODER** (codificação) quanto o **DECODER** (decodificação).

Na etapa de CODER, a imagem de entrada é processada primeiro por uma camada convolucional, em seguida, concatenada com as *"labels"* condicionais e, finalmente, passa por uma camada linear totalmente conectada. Isso resulta em duas saídas: a *média* e o *log da variância*, que definem a distribuição normal a partir da qual o espaço latente é amostrado. A amostragem é feita pelo método *'reparametrize'*.

Na decodificação, a amostra latente é concatenada com os labels condicionais e, em seguida, passa por duas camadas lineares totalmente conectadas para produzir a reconstrução da imagem de entrada.

A função de perda usada é uma combinação de perda de entropia cruzada binária (para a reconstrução) e a divergência KL (para forçar a distribuição latente a seguir uma distribuição normal padrão). Isso é típico de VAEs.

O modelo é então treinado por 10 épocas usando o otimizador Adam, onde cada batch de dados é passado pelo modelo, a perda é calculada e os gradientes são propagados de volta através do modelo para atualizar os pesos.

```
# Preprocessamento do MNIST
transform = transforms.Compose([transforms.ToTensor()])

train_dataset = datasets.FashionMNIST(root='./data', train=True, transform=transform,
↪  download=True)
train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=128,
↪  shuffle=True)

# Alguns hiperparâmetros
input_dim = 784  # Imagens 28x28 serão representadas por um array de 784 entradas
label_dim = 10  # Número de labels possíveis (de 0 a 9)

hidden_dim_encoder = 784
hidden_dim_decoder = 64
# Diminuímos a dimensão dos dados no decoder
# caso queiramos que o encoder "capture" mais features, seria interessante aumentar
↪  seu hidden_dim
# o mesmo vale para o decoder, onde poderíamos buscar "gerar" mais complexidade ao
↪  modelo
latent_dim = 10  # dimensão de representação latente
```

A arquitetura do nosso CVAE foi implementada na classe $CVAE$, onde tanto o *encoder* quanto o *decoder* são condicionais a uma variável de interesse, neste caso, às labels das imagens. Isso permite que o modelo aprenda a representação latente dos dados de entrada levando em conta os rótulos, proporcionando maior controle sobre a geração de novas amostras.

O processo de codificação inicia com uma camada convolucional ($conv1$), que tem a finalidade de capturar automaticamente características relevantes da imagem de entrada, substituindo a necessidade de engenharia manual de características. Após essa camada, os dados são concatenados com as labels (a variável condicional), e passam por uma camada totalmente conectada ($fc1$). O output dessa camada é dividido em dois vetores, representando a média e o logaritmo da variância da distribuição latente.

Na fase de decodificação, a arquitetura faz uso do método de reparametrização. Este é um passo crucial que permite que o VAE seja treinado com *backpropagation*. Aqui, amostras aleatórias de uma distribuição normal padrão são ajustadas com base na média e variância aprendidas na fase de codificação.

O resultado do método de reparametrização é então concatenado com os rótulos e passa por uma camada totalmente conectada ($fc2$). A saída dessa camada é processada por outra camada totalmente conectada ($fc3$), que tenta reconstruir a imagem original.

A função de perda usada é composta por duas partes: a perda de reconstrução e a *divergência KL*. A perda de reconstrução é calculada usando a *entrópia cruzada binária*, que mede a diferença entre a imagem de entrada original e a imagem reconstruída. A divergência KL é usada para garantir que a distribuição latente se aproxime de uma distribuição normal padrão, funcionando como um regularizador.

Por fim, a otimização dos parâmetros do modelo é feita usando o algoritmo Adam, com um termo de decaimento de peso adicionado para implementar a regularização L2, ajudando a prevenir um *overfitting* [2].

```python
class CVAE(nn.Module):
    def __init__(self, input_dim, label_dim, hidden_dim_encoder, hidden_dim_decoder,
    ↪  latent_dim):
        super(CVAE, self).__init__()

        # Encoder
        self.conv1 = nn.Sequential(
            nn.Conv2d(1, 16, kernel_size=3, stride=2, padding=1),
            nn.BatchNorm2d(16),
            nn.ReLU(),
        )
        self.fc1 = nn.Linear(16 * 14 * 14 + label_dim, hidden_dim_encoder)
        self.fc_mean = nn.Linear(hidden_dim_encoder, latent_dim)
        self.fc_logvar = nn.Linear(hidden_dim_encoder, latent_dim)

        # Decoder
        self.fc2 = nn.Linear(latent_dim + label_dim, hidden_dim_decoder)
        self.fc3 = nn.Linear(hidden_dim_decoder, input_dim)

    def encode(self, x, y):
        x = self.conv1(x)
        x = x.view(x.size(0), -1)
        input = torch.cat((x, y), dim=1) # aqui que fazemos a parte "condicional" do
        ↪  VAE
        hidden = F.relu(self.fc1(input))
        mean = self.fc_mean(hidden)
        logvar = self.fc_logvar(hidden)
        return mean, logvar
```

[2] Um cenário de overfitting ocorre quando, nos dados de treino, o modelo tem um desempenho excelente, porém quando utilizamos os dados de teste o resultado é ruim.

```
    def reparametrize(self, mean, logvar):
        std = torch.exp(0.5 * logvar)
        epsilon = torch.randn_like(std)
        z = mean + epsilon * std
        return z

    def decode(self, z, y):
        input = torch.cat((z, y), dim=1)  # aqui que fazemos a parte "condicional" do
        ↪  VAE
        hidden = F.relu(self.fc2(input))
        reconstructed = torch.sigmoid(self.fc3(hidden))
        return reconstructed

    def forward(self, x, y):
        mean, logvar = self.encode(x, y)
        z = self.reparametrize(mean, logvar)
        reconstructed = self.decode(z, y)
        return reconstructed, mean, logvar

def loss_function(reconstructed, x, mean, logvar):
    # loss (binary cross-entropy)
    reconstruction_loss = F.binary_cross_entropy(reconstructed.view(-1, input_dim),
    ↪  x.view(-1, input_dim), reduction='sum')

    # loss (divergencia KL)
    kl_loss = -0.5 * torch.sum(1 + logvar - mean.pow(2) - logvar.exp())

    total_loss = reconstruction_loss + kl_loss
    return total_loss

model = CVAE(input_dim, label_dim, hidden_dim_encoder, hidden_dim_decoder,
↪  latent_dim)
model = model.to(device)

# podemos mudar o otimizador também / adicionamos weight_decay para implementar
↪  regularização L2
optimizer = optim.Adam(model.parameters(), lr=0.001, weight_decay=1e-5)
```

Com certeza, a escolha dos **hiperparâmetros** [3] é uma parte crucial para o desempenho de qualquer modelo de aprendizado de máquina.

No caso deste CVAE, a dimensão de entrada é definida com base no tamanho da imagem, que é de 28×28 pixels, resultando em um vetor unidimensional de 784 elementos. A dimensão dos rótulos é definida como 10, representando as 10 classes possíveis no FashionMNIST.

Para as camadas ocultas do encoder e do decoder, a dimensão foi definida como 784 e 644, respectivamente. Essas dimensões influenciam na capacidade do modelo em aprender representações complexas. O encoder, tendo uma dimensão oculta maior, tem mais capacidade para aprender a representação latente dos dados. O decoder, por outro lado, tem uma dimensão oculta menor, o que pode ajudar a evitar overfitting, já que uma rede com menos parâmetros tem menos chance de memorizar os dados de treinamento.

A dimensão latente, que define o tamanho do espaço latente, foi definida como 10. Esta é uma escolha arbitrária e pode ser ajustada para melhor se adaptar ao problema em questão. Um espaço latente de maior dimensão pode ser capaz de capturar uma maior complexidade nos dados, mas também pode levar a overfitting se a dimensão for muito grande em relação à quantidade de dados disponíveis.

O otimizador *Adam* foi escolhido com uma taxa de aprendizado de 0.001. Adam é uma escolha po-

[3] Hiperparâmetros são parâmetros ajustáveis que permitem controlar o processo de treinamento do modelo.

pular para muitos problemas de aprendizado profundo, pois combina os benefícios de outros métodos de otimização avançados. A taxa de aprendizado é um hiperparâmetro crucial que determina o quão grandes serão os ajustes nos pesos do modelo durante o treinamento. Uma taxa de aprendizado muito alta pode fazer com que o modelo não consiga convergir, enquanto uma taxa de aprendizado muito baixa pode fazer com que o treinamento seja muito lento. A taxa de aprendizado de 0.001 é um valor comumente utilizado, mas deve ser ajustada se o modelo não estiver aprendendo adequadamente.

Além disso, o Adam foi usado com um decaimento de peso (ou regularização $L2$) de $1e-5$. A regularização L2 ajuda a prevenir o overfitting adicionando uma penalidade aos pesos na função de custo, o que encoraja o modelo a ter pesos menores. Isso torna o modelo menos propenso a ajustar ruído nos dados de treinamento, melhorando sua capacidade de generalização.

Aqui realizamos a etapa de treinamento do modelo:

Primeiramente, a função train é responsável por iterar sobre várias épocas de treinamento e atualizar os pesos do modelo usando o otimizador escolhido. Durante cada época, a função itera sobre cada lote de dados no *train_loader*. Para cada lote, os dados e rótulos são movidos para o dispositivo de computação apropriado (CPU ou GPU) e as labels são convertidas em representações de **"one-hot"** [4] , que é a representação apropriada para usar com um CVAE, já que estamos lidando com um problema de classificação **multiclasse**.

Depois, o otimizador é reiniciado para garantir que não haja acumulação de gradientes de iterações anteriores. Em seguida, os dados e os rótulos são passados pelo modelo, resultando em uma reconstrução dos dados de entrada e os parâmetros da distribuição latente (média e logaritmo da variância).

A função de perda é então calculada usando a função $loss_function$ que foi definida anteriormente. Essa função de perda é uma combinação da perda de reconstrução (usando a entropia cruzada binária) e a perda KL, que garante que a distribuição latente segue uma distribuição normal.

A função **backward**() é então chamada na perda, o que efetivamente calcula os gradientes da perda com respeito a todos os parâmetros do modelo. O otimizador então atualiza os parâmetros do modelo usando esses gradientes.

Este processo é repetido para cada lote de dados, acumulando a perda total para a época ao longo do caminho. No final de cada época, a perda média é impressa.

O número de épocas para o qual o treinamento é realizado é um hiperparâmetro que deve ser escolhido com base na complexidade do problema e no tempo de treinamento disponível. Aqui, o número de épocas foi definido como 10, o que é um ponto de partida razoável para muitos problemas, mas pode precisar ser ajustado para obter o melhor desempenho.

```
def train(model, optimizer, num_epochs):
    model.train()
    for epoch in range(num_epochs):
        train_loss = 0
        for batch_idx, (data, labels) in enumerate(train_loader):
            data = data.to(device)
            labels = F.one_hot(labels, num_classes=10).type(torch.float).to(device)
            optimizer.zero_grad()

            reconstructed, mean, logvar = model(data, labels)

            loss = loss_function(reconstructed, data.view(-1, input_dim), mean,
            ↪  logvar)

            loss.backward()
            train_loss += loss.item()
            optimizer.step()

            # Print training information
            if batch_idx % 100 == 0:
                print('Epoch [{}/{}], Batch [{}/{}], Loss: {:.4f}'
                      .format(epoch+1, num_epochs, batch_idx+1, len(train_loader),
                      ↪  loss.item()))

        print('Epoch [{}/{}], Average Loss: {:.4f}'
              .format(epoch+1, num_epochs, train_loss/len(train_loader)))
```

[4] É uma técnica usada em machine learning para converter variáveis categóricas em uma forma numérica. Cada valor único da variável categórica é transformado em uma nova variável binária

```
num_epochs = 10
train(model, optimizer, num_epochs)
```

Nesta parte, estamos utilizando o modelo CVAE treinado para gerar novas amostras condicionais. O objetivo é gerar amostras de imagens correspondentes a uma determinada label. A função '*generate_samples*′ recebe o modelo treinado, o número de amostras desejadas e a label específica. Primeiro, colocamos o modelo em modo de avaliação (′*model.eval*()′) para desativar o cálculo de gradientes.

Em seguida, geramos amostras aleatórias na dimensão latente ('z') a partir de uma distribuição normal padrão usando a função '*torch.randn*'. Essas amostras são ajustadas ao dispositivo em uso ('.*to*(*device*)′). Criamos um tensor de labels com tamanho (*num_samples*, *label_dim*) e preenchemos com zeros usando a função '*torch.zeros*'. Em seguida, definimos a label específica como 1 para a dimensão correspondente.

Utilizamos o modelo para decodificar as amostras de latente ('z') e as labels ('y') usando a função '*model.decode*'. Isso nos fornece as amostras geradas ('*generated_samples*').Em seguida, definimos um mapeamento de rótulos ('*clothing_map*') que associa os índices das labels às classes de roupas correspondentes. Imprimimos uma mensagem indicando a classe de roupas das amostras geradas com base na label fornecida.

Por fim, redimensionamos as amostras geradas para o formato de imagem (28x28) usando a função '*view*'. Isso nos permite visualizar as amostras como imagens. Finalmente, exibimos a figura com as amostras geradas usando '*plt.show*()'.

```
import matplotlib.pyplot as plt

# geraçao de novas amostras
def generate_samples(model, num_samples, label):
    model.eval()
    with torch.no_grad():
        z = torch.randn(num_samples, latent_dim).to(device)
        y = torch.zeros(num_samples, label_dim).to(device)
        y[:, label] = 1
        generated_samples = model.decode(z, y)

    clothing_map = {
        0: 'T-shirt/top',
        1: 'Trouser',
        2: 'Pullover',
        3: 'Dress',
        4: 'Coat',
        5: 'Sandal',
        6: 'Shirt',
        7: 'Sneaker',
        8: 'Bag',
        9: 'Ankle boot'
    }

    print(f'Imagens de: {clothing_map[label]}')
    return generated_samples

num_samples = 10
generated_samples = generate_samples(model, num_samples, 7)

generated_samples = generated_samples.view(-1, 28, 28)

fig, axes = plt.subplots(1, num_samples, figsize=(15, 3))
for i in range(num_samples):
    axes[i].imshow(generated_samples[i].cpu().numpy(), cmap='gray')
```

```
    axes[i].axis('off')
plt.show()
```

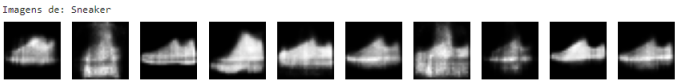

Figura 11: Imagens geradas

6.2 Codigo da Juliana: β-VAE

Antes de avançar vamos fazer uma breve introdução :

7 Beta-Variational Autoencoder (β-VAE)

O β-VAE é um tipo modificado do autoencoder variacional que busca descobrir fatores latentes desembaraçados. Ele modifica os VAEs com um hiperparâmetro ajustável β que equilibra a capacidade dos canais latentes e as restrições de independência com a precisão da reconstrução. A ideia é maximizar a probabilidade de gerar os dados reais enquanto mantém a distância entre as distribuições reais e estimadas pequena, abaixo de um limite. Podemos usar as condições de Kuhn-Tucker [5]para escrever isso como uma única equação representada por $(*)$.

Nós reescrevemos isso novamente usando a suposição de folga complementar para obter a formulação do β-VAE:

$$\mathcal{F}(\theta, \phi, x, z, \beta) \geq \mathcal{L}(\theta, \phi, x, z, \beta) = \mathbb{E}_{q_\phi(z|x)}[logp_\theta(x|z)] - \beta D_{KL}(q_\phi(x|z)||p(z))(*),$$

onde $\mathcal{F}$ representa a função objetivo ou função de perda do β-VAE.

7.1 Arquitetura de Burgues

Enconder
Arquitetura da rede: A arquitetura da rede consiste em 4 camadas convolucionais (cada uma com 32 canais e um kernel de tamanho 4x4 com stride de 2), 2 camadas totalmente conectadas (cada uma com 256 unidades), e uma camada totalmente conectada de 20 unidades que é usada para gerar a média e o log da variância para 10 Gaussianas.
Camadas Convolucionais: As camadas convolucionais são definidas como self.conv1, self.conv2, self.conv3 e $self.conv_6 4$. Cada uma dessas camadas usa a função nn.Conv2d do PyTorch para aplicar uma convolução 2D ao input. A quarta camada convolucional só é aplicada se o tamanho da imagem de entrada for 64x64.
Camadas Totalmente Conectadas: As camadas totalmente conectadas são definidas como self.lin1 e self.lin2, e a camada que gera a média e log variância é definida como$self.mu_logvar_gen$. Essas camadas usam a função nn.Linear do PyTorch para aplicar uma transformação linear ao input.
Forward: O método forward define como a rede processa a entrada e gera uma saída. Primeiro, as ativações ReLU são aplicadas após cada camada convolucional e totalmente conectada para introduzir não linearidades. Depois, a saída é achatada (ou seja, transformada em um vetor unidimensional) antes de passar pelas camadas totalmente conectadas. Finalmente, a média e log variância são geradas e retornadas.
Na maioria dos VAEs tradicionais, a representação latente é modelada como uma única distribuição gaussiana multivariada. No entanto, no caso do modelo -VAE, a arquitetura é modificada para modelar explicitamente vários fatores de variação nos dados. Esses fatores são representados como várias distribuições gaussianas independentes no espaço latente, em vez de uma única distribuição gaussiana. Isso é feito para melhorar a "desenredação"(disentangling) das representações latentes, o que significa que cada dimensão do espaço latente corresponderia a um fator de variação nos dados.
O objetivo é que ao variar uma única dimensão do espaço latente, apenas uma única característica dos dados seja alterada, enquanto todas as outras características permanecem constantes. Por exemplo, em um conjunto de dados de rostos humanos, os fatores de variação podem incluir coisas como a cor do cabelo, o ângulo da cabeça, a expressão facial, etc.
A modificação na arquitetura e no método de treinamento ajuda o modelo a aprender esses fatores de variação de maneira mais explícita e eficaz do que um VAE tradicional.
Em termos de implementação, o EncoderBurgess inclui uma camada adicional de geração de média e log-variança ($self.mu_logvar_gen$) que produz duas vezes o número de saídas latentes para representar a média e a log-variança de várias distribuições gaussianas no espaço latente.

[5]Condições de Karush-Kuhn-Tucker: (condições KKT) são condições de primeira ordem para que uma solução de um problema de programação não linear seja ótima, desde que valham condições chamadas de condições de qualificação Permitindo restrições de desigualdade, as condições KKT generalizam, na programação não linear, o método de multiplicadores de Lagrange, que permite somente restrições de igualdade.

Deconder
Arquitetura da rede: A arquitetura do decodificador é essencialmente o inverso da do encoder, como é comum nos autoencoders. Ele começa com três camadas totalmente conectadas (self.lin1, self.lin2, self.lin3), seguidas por várias camadas de convolução transposta (self.convT1, self.convT2, self.convT3 e opcionalmente $self.convT_64$ se a imagem de entrada for 64x64). As camadas de convolução transposta são também chamadas de camadas de "deconvolução", e são usadas para aumentar a dimensionalidade dos dados - no caso, para "descompactar"a representação latente em uma imagem.
Forward: O método forward define como a rede processa a entrada e gera uma saída. Ele recebe z, a representação latente. Primeiro, passa z através das camadas totalmente conectadas com ativações ReLU, em seguida, redimensiona a saída para a forma necessária para iniciar as convoluções transpostas. A saída então passa por cada uma das camadas de convolução transposta, também com ativações ReLU. A última camada de convolução transposta usa uma ativação sigmóide, que é comum quando se quer restringir a saída a um intervalo específico (neste caso, entre 0 e 1, o que faz sentido para imagens).

7.2 burgues

Encoder: Aprende a comprimir os dados de entrada em um vetor latente. Na arquitetura proposta por Burgess, o encoder é uma rede convolucional com várias camadas que termina em duas camadas totalmente conectadas. A última camada do encoder produz duas saídas: uma para a média e uma para o log da variância. Estes são os parâmetros da distribuição gaussiana multivariada que a representação latente seguirá.

Decoder: Aprende a reconstruir os dados a partir do vetor latente. Na arquitetura de Burgess, o decoder é essencialmente o espelho do encoder: começa com algumas camadas totalmente conectadas que aumentam a dimensionalidade dos dados, e depois usa camadas de convolução transposta para reconstruir a imagem.
Implementação

A implementação segue a arquitetura acima. A maior diferença em comparação com um VAE tradicional é que a representação latente é modelada como várias distribuições gaussianas independentes, ao invés de uma única distribuição gaussiana. Isso ajuda o modelo a aprender representações mais "desenredadas".
sobre as diferenças Fator : A diferença mais notável é o fator . Em um VAE padrão, a função de perda é a soma da reconstrução e da perda KL (Divergência de Kullback-Leibler), que mede a diferença entre a distribuição latente aprendida e uma distribuição normal padrão. No entanto, no -VAE, a perda KL é multiplicada por um fator , que pode ser maior que 1. Aumentar aplica mais pressão sobre a rede para aprender uma distribuição latente que é semelhante à distribuição normal padrão. Isso pode levar a uma representação latente mais desenredada.

Desenredamento: Uma das principais motivações para o -VAE é aprender representações desenredadas. Em uma representação desenredada, cada dimensão do espaço latente é independente e corresponde a um fator de variação diferente nos dados. Em outras palavras, alterar o valor ao longo de uma única dimensão resultaria em uma alteração correspondente em um único fator de variação na reconstrução. Embora um VAE padrão possa aprender alguma desenredamento, não há garantias. A introdução do fator no -VAE de Burgess é uma tentativa de encorajar mais desenredamento.

Desempenho e utilidade: Em termos de resultados, Burgess et al. mostraram que o β-VAE pode alcançar uma representação mais desenredada dos dados do que um VAE padrão. Isto é, a representação latente aprendida pelo β-VAE é mais interpretável, onde cada dimensão latente representa um fator de variação distinto nos dados. Além disso, eles mostraram que o desenredamento pode ser alcançado sem comprometer significativamente a qualidade da reconstrução, especialmente para valores moderados de . Portanto, o -VAE pode ser particularmente útil em aplicações onde a interpretabilidade da representação latente é importante.

Em termos de implementação, o -VAE é muito semelhante a um VAE padrão - a diferença principal é a adição do fator na função de perda. Portanto, a maioria dos aspectos práticos da implementação de um β-VAE é a mesma de um VAE padrão.

7.3 Comparação entre CVAE e β-VAE

8 Resultados

9 Discussão

10 Conclusão

Diante dos experimentos, se treinarmos um disentagled VAE, o codificador será capaz de reconstruir e criar um mapeamento desses 4 valores latentes para codificar as funções de entrada.

11 Apêndice

12 Referências

1. "Autoencoder Image Interpolation by Shaping the Latent Space", Oring et al. https://arxiv.org/abs/2008.01487
2. "Understanding and Improving Interpolation in Autoencoders via an Adversarial Regularizer", Berthelot et al. https://arxiv.org/abs/1807.07543
3. "Learning disentangled representations in the imaging domain", Liu et al. https://www.sciencedirect.com/science/ar
4. "Invariant-equivariant representation learning for multi-class data", Ilya Feige. https://arxiv.org/abs/1902.03251
5. "Deep Learning – Equivariance and Invariance", Bernhard Kainz. https://www.doc.ic.ac.uk/ bkainz/teaching/DL/n
6. "Domain shift", Marco Taboga. https://www.statlect.com/machine-learning/domain-shift
7. "A Review of Disentangled Representation Learning for Remote Sensing Data", Wang et al. https://www.sciopen.cor
8. "Towards a Definition of Disentangled Representations", Higgins et al. https://arxiv.org/pdf/1812.02230.pdf
9. "From Autoencoder to Beta-VAE", Lilian Weng. https://lilianweng.github.io/posts/2018-08-12-vae/
10. "Structured Disentangled Representations", Esmaeili et al. https://proceedings.mlr.press/v89/esmaeili19a/esmaeili1
11. "Sufficient and Disentangled Representation Learning", Huang et al. https://openreview.net/forum?id=IeuEO1TccZ
12. "DARLA: Improving Zero-Shot Transfer in Reinforcement Learning", Higgins et al. https://arxiv.org/pdf/1707.0847
13. "Auto-Encoding Variational Bayes", Diederik P. Kingma, Max Welling. https://arxiv.org/pdf/1312.6114.pdf
14. "Understanding Conditional Variational Autoencoders", Md Ashiqur Rahman. https://towardsdatascience.com/und conditional-variational-autoencoders-cd62b4f57bf8
15. "Understanding disentangling in β-VAE", Burgess et al. https://arxiv.org/pdf/1804.03599.pdf