»Лекционен курс

»Въведение в генеративния ИИ

VAE ›

Увод

- През 2013 г. Дидерик П. Кингма и Макс Уелинг публикуваха статия, която положи основите на тип невронна мрежа, известна като вариационен автоенкодер (VAE).
- Сега това е една от най-фундаменталните и добре познати ДУ архитектури за ГМ.
- Отлично начало за изучаване на генеративно дълбоко учене.

Увод

- Ще започнем с изграждането на стандартен автоенкодер.
- След това ще видим как можем да го разширим към вариационен автоенкодер.
- Междувременно ще разгледаме двата типа модели за да разберем как работят на детайлно ниво.
- До края на лекцията трябва да имаме пълно разбиране за това как да изграждаме и обработваме базирани на автоенкодери модели.
- По-специално, как да изграждаме вариационен автоенкодер от нулата за да генерираме изображения въз основа на собствен набор от данни.

Въвеждащ пример

Виртуален гардероб за дрехи

- Нека започнем с една проста история, която ще ни помогне да обясним основния проблем, който се опитва да реши един автокодер.
- Да си представим, че на пода пред нас са купчина от всички дрехи, които притежаваме - панталони, обувки, палта … всички в различни стилове.
- Имаме стилист, който е недоволен поради това, че му отнема много време да осигури необходимите ни облекла.
- Той измисля хитър план.

Виртуален гардероб за дрехи

- Предлага да организираме дрехите си в гардероб, който е с достатъчно големи размери.
- Когато искаме конкретна дреха, просто трябва да му кажем местоположението ѝ и той ще я ушие.
- Скоро става очевидно, че ще трябва да поставим подобни дрехи близо една до друга, така че стилистът да може точно да я пресъздаде, като използва само нейното местоположение.

Виртуален гардероб за дрехи

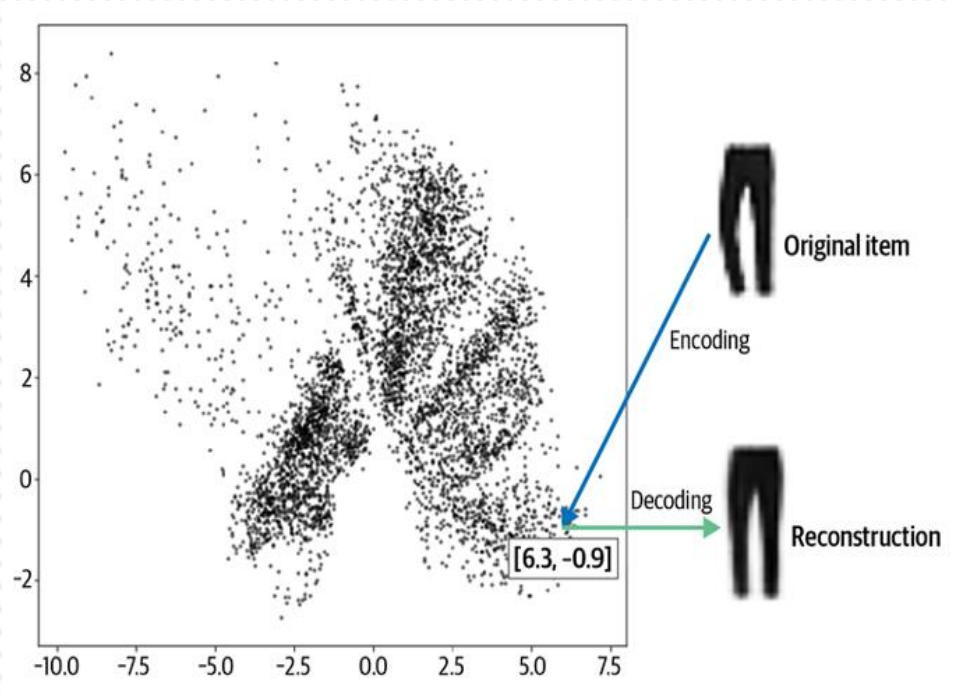
- След няколко седмици практика ние и стилистът се приспособяваме към разбиранията си за оформлението на гардероба.
- Вече е възможно да кажем на стилиста местоположението на всяка дреха, която желаем, и той може точно да я ушие.

Виртуален гардероб за дрехи

- Какво би се случило, ако посочим на стилиста място в гардероба, което е празно?
- За наше учудване откриваме, че той е в състояние да ушие изцяло нови дрехи, които не са съществували преди.
- Процесът не е перфектен, но сега имаме неограничени възможности за генериране на ново облекло, просто като изберем празно място в безкрайния гардероб и оставим стилиста да работи.
- Нека сега научим как тази история е свързана със създаването на автоенкодери.

Автоенкодери

Кодиране - декодиране



Диаграма на процеса, описан в историята, е показана на фигурата.

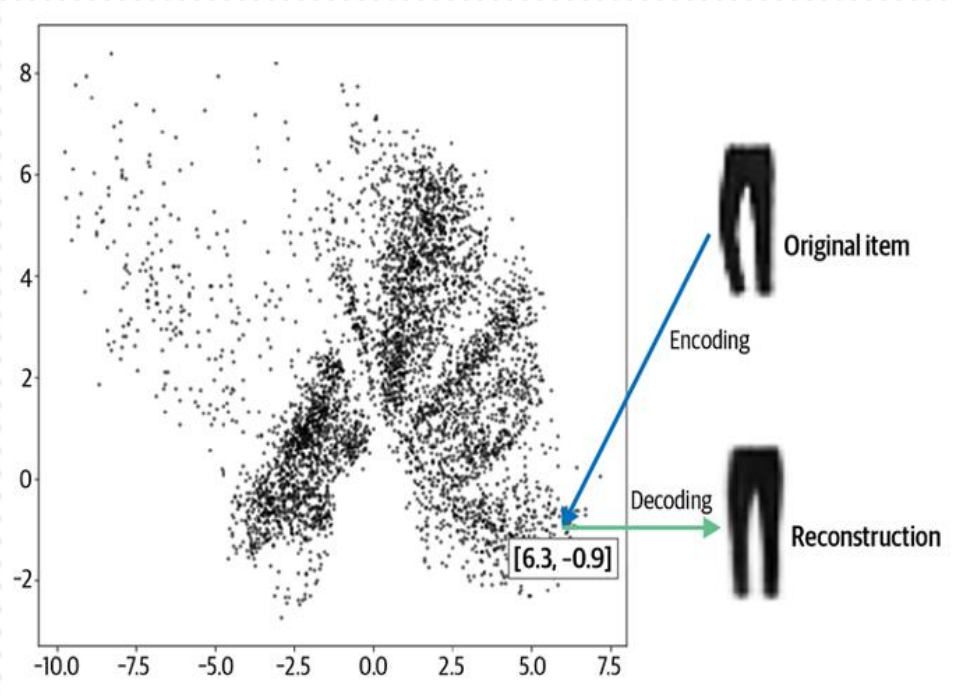
Ние играем ролята на енкодера, поставяйки всеки елемент от облеклото на място в гардероба.

Този процес се нарича кодиране.

Стилистът играе ролята на декодера, като заема място в гардероба и се опитва да пресъздаде артикула.

Този процес се нарича декодиране.

Вграждане



Всяко място в гардероба е представено от две числа (т.е. 2D вектор).

Например, панталоните на фигурата са кодирани до точка [6.3, –0.9].

Този вектор е известен също като вграждане, тъй като енкодерът се опитва да вгради възможно най-много информация в него, така че декодерът да може да произведе точна реконструкция.

Определение

- Автокодерът е просто невронна мрежа, която е обучена да изпълнява задачата за кодиране и декодиране на елемент, така че изходът от този процес да е възможно най-близо до оригиналния елемент.
- Най-важното е, че може да се използва като ГМ, понеже можем да декодираме всяка точка в 2D пространството, която искаме (по-специално тези, които не са вграждания на оригинални елементи) за да произведем нов елемент от облеклото.

Fashion-MNIST набор данни

```
from tensorflow.keras import datasets
(x_train,y_train), (x_test,y_test) = datasets.fashion_mnist.load_data()
```

Като пример ще използваме набора от данни Fashion-MNIST – колекция от изображения в сива скала на облекла, всяко с размер 28 × 28 пиксела.

Някои примерни изображения от набора от данни са показани на фигурата.

Наборът от данни е предварително опакован с TensorFlow, така че може да бъде изтеглен, както е показано.

Предварителна обработка

```
def preprocess(imgs):
    imgs = imgs.astype("float32") / 255.0
    imgs = np.pad(imgs, ((0, 0), (2, 2), (2, 2)), constant_values=0.0)
    imgs = np.expand_dims(imgs, -1)
    return imgs

x_train = preprocess(x_train)
x_test = preprocess(x_test)
```

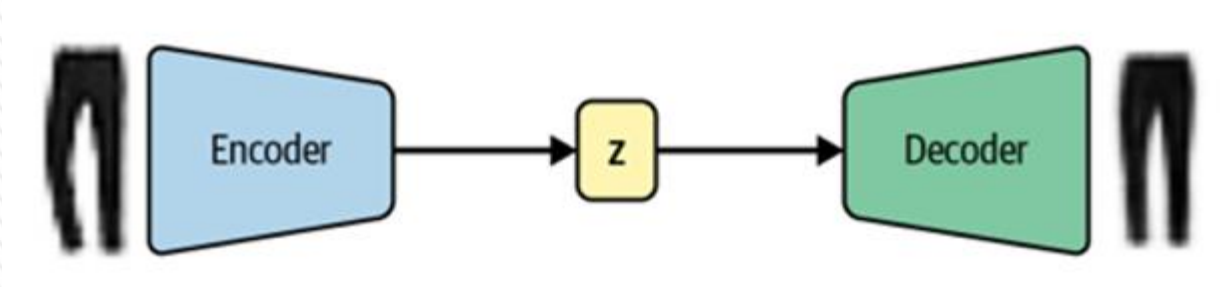
Това са 28 × 28 изображения в нива на сивото (стойности на пикселите между 0 и 255), които трябва да обработим предварително за да гарантираме, че стойностите на пикселите са мащабирани между 0 и 1.

Ние също ще добавим всяко изображение до 32 × 32 за по-лесно манипулиране на формата на тензора, докато преминава през мрежата, както е показано.

Архитектура на автоенкодер

- Автокодерът е невронна мрежа, съставена от две части:
  - Енкодерна мрежа, която компресира високоразмерни входни данни (напр. изображение) в по-нискоизмерен вектор за вграждане.
  - Декодерна мрежа, която декомпресира даден вектор за вграждане обратно към оригиналния домейн (напр. обратно към изображение).

Архитектура на автоенкодер



Диаграма на мрежовата архитектура е показана на фигурата.
Едно входно изображение се кодира в латентно вграден вектор z, който след това се декодира обратно до оригиналното пикселно пространство.

Принцип на действие

- Автоенкодерите са обучени да реконструират изображения, след като са преминали през енкодера и обратно през декодера.
- Това може да изглежда странно в началото - защо бихте искали да реконструирате набор от изображения, които вече имате на разположение?
- Въпреки това (както ще видим), именно пространството за вграждане (наричано още латентно пространство) е интересната част от автокодера, тъй като вземането на проби от това пространство ще ни позволи да генерираме нови изображения.

Вграждане

- Нека първо да дефинираме какво разбираме под вграждане.
- Вграждането (z) е компресия на оригиналното изображение в по-нискоразмерно латентно пространство.
- Идеята е, че като изберем някоя точка в латентното пространство можем да генерираме нови изображения, като прекараме тази точка през декодера.
  - Понеже декодерът се е научил как да преобразува точки в латентното пространство в жизнеспособни изображения.

Пример за 2D латентно пространство

- В нашия пример ще вградим изображения в двуизмерно латентно пространство.
- Това ще ни помогне да визуализираме латентното пространство, тъй като можем лесно да начертаем точки в 2D.
- На практика латентното пространство на автокодера обикновено има повече от две измерения за да има повече свобода за улавяне на повече нюанси в изображенията.

Използване

- Автоенкодерите могат да се използват за почистване на изображения с шум, тъй като енкодерът научава, че не е полезно да уловя позицията на произволния шум вътре в латентното пространство за да реконструира оригинала.
- За задачи като тази, 2D латентно пространство вероятно е твърде малко за да кодира достатъчно подходяща информация от входа.
- Въпреки това (както ще видим) увеличаването на размерността на латентното пространство бързо води до проблеми, ако искаме да използваме автокодера като ГМ.

Архитектура на енкодер

Layer (type)	Output shape	Param #
InputLayer	(None, 32, 32, 1)	0
Conv2D	(None, 16, 16, 32)	320
Conv2D	(None, 8, 8, 64)	18,496
Conv2D	(None, 4, 4, 128)	73,856
Flatten	(None, 2048)	0
Dense	(None, 2)	4,098

Total params	96,770
Trainable params	96,770
Non-trainable params	0

В един автоенкодер работата на енкодера е да вземе входното изображение и да го картографира към вграден вектор в латентното пространство.

Архитектурата на енкодера, който ще изграждаме, е показана в таблицата.

Архитектура на енкодер

Layer (type)	Output shape	Param #
InputLayer	(None, 32, 32, 1)	0
Conv2D	(None, 16, 16, 32)	320
Conv2D	(None, 8, 8, 64)	18,496
Conv2D	(None, 4, 4, 128)	73,856
Flatten	(None, 2048)	0
Dense	(None, 2)	4,098

Total params	96,770
Trainable params	96,770
Non-trainable params	0

За да постигнем това, първо създаваме входен слой за изображението и го предаваме последователно през три слоя Conv2D, като всеки улавя характеристики от все по-високо ниво.

Използваме стъпка от 2, за да намалим наполовина размера на изхода на всеки слой, като същевременно увеличаваме броя на каналите.

Последният конволюционен слой е сплескан и свързан с плътен слой с размер 2, който представлява нашето двуизмерно латентно пространство.

Програмен код на енкодер

```
encoder_input = layers.Input(
    shape=(32, 32, 1), name = "encoder_input"
) ❶
x = layers.Conv2D(32, (3, 3), strides = 2, activation = 'relu', padding="same")(
    encoder_input
) ❷
x = layers.Conv2D(64, (3, 3), strides = 2, activation = 'relu', padding="same")(x)
x = layers.Conv2D(128, (3, 3), strides = 2, activation = 'relu', padding="same")(x)
shape_before_flattening = K.int_shape(x)[1:]

x = layers.Flatten()(x) ❸
encoder_output = layers.Dense(2, name="encoder_output")(x) ❹

encoder = models.Model(encoder_input, encoder_output) ❺
```

1. Дефинира входния слой на енкодера (изображението).
2. Подрежда Conv2D слоевете последователно един върху друг.
3. Изравнява последния конволюционен слой до вектор.
4. Свързва този вектор към 2D вгражданията с плътен слой.
5. Моделът Keras, който дефинира енкодера - модел, който приема входно изображение и го кодира в 2D вграждане.

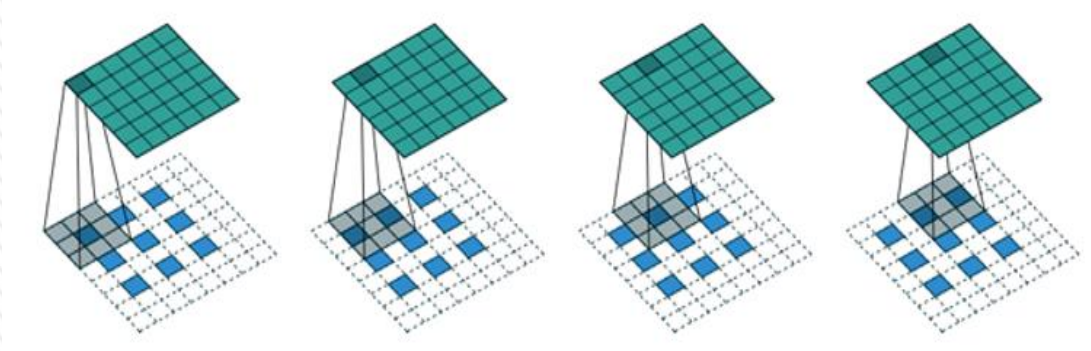
Декодер

Layer (type)	Output shape	Param #
InputLayer	(None, 2)	0
Dense	(None, 2048)	6,144
Reshape	(None, 4, 4, 128)	0
Conv2DTranspose	(None, 8, 8, 128)	147,584
Conv2DTranspose	(None, 16, 16, 64)	73,792
Conv2DTranspose	(None, 32, 32, 32)	18,464
Conv2D	(None, 32, 32, 1)	289

Total params	246,273
Trainable params	246,273
Non-trainable params	0

Декодерът е огледален образ на енкодера - вместо конволюционни слоеве използва конволюционни транспонирани слоеве, както е показано в таблицата.

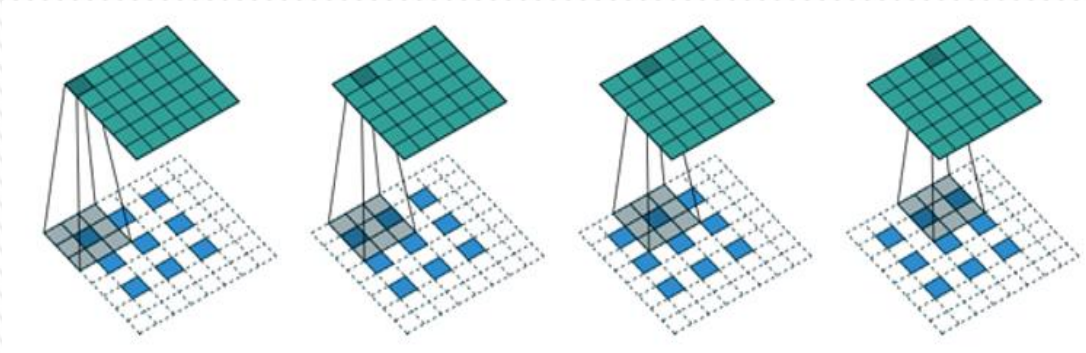
Конволюционни транспонирани слоеве



Стандартните конволюционни слоеве ни позволяват да намалим наполовина размера на входен тензор и в двете измерения (височина и ширина), като зададем крачки = 2.

Конволюционният транспониран слой използва същия принцип като стандартен конволюционен слой (преминаване на филтър през изображението), но е различен по това, че настройката на крачки = 2 удвоява размера на входния тензор и в двете измерения.

Конволюционни транспонирани слоеве



В конволюционен транспониран слой параметърът за крачки определя вътрешната нулева подложка между пикселите в изображението, както е показано на фигурата.

Тук филтър 3 × 3 × 1 (сив) преминава през изображение 3 × 3 × 1 (синьо) със стъпки = 2, за да се получи изходен тензор 6 × 6 × 1 (зелен).

Conv2DTranspose в Keras

- В Keras, слоят Conv2DTranspose ни позволява да извършваме конволюционни транспониращи операции върху тензори.
- Като подреждаме тези слоеве, можем постепенно да разширяваме размера на всеки слой, като използваме крачки от 2, докато се върнем към оригиналното измерение на изображението от 32 × 32.

Програмен код на декодера

```
decoder_input = layers.Input(shape=(2,), name="decoder_input") ❶
x = layers.Dense(np.prod(shape_before_flattening))(decoder_input) ❷
x = layers.Reshape(shape_before_flattening)(x) ❸
x = layers.Conv2DTranspose(
    128, (3, 3), strides=2, activation = 'relu', padding="same"
)(x) ❹
x = layers.Conv2DTranspose(
    64, (3, 3), strides=2, activation = 'relu', padding="same"
)(x)
x = layers.Conv2DTranspose(
    32, (3, 3), strides=2, activation = 'relu', padding="same"
)(x)
decoder_output = layers.Conv2D(
    1,
    (3, 3),
    strides = 1,
    activation="sigmoid",
    padding="same",
    name="decoder_output"
)(x)

decoder = models.Model(decoder_input, decoder_output) ❺
```

1. Дефинира входния слой на декодера (вграждането).
2. Свързва входа към плътен слой.
3. Преоформя този вектор в тензор, който може да бъде подаден като вход към първия Conv2DTranspose слой.
4. Подрежда Conv2DTranspose слоеве един върху друг.
5. Моделът Keras, който дефинира декодера - модел, който приема вграждане в латентното пространство и го декодира в оригиналния домейн на изображението.

Обучение

- За да обучим енкодера и декодера едновременно, трябва да дефинираме модел, който ще представлява потока на изображение през енкодера и обратно през декодера.
- За щастие, Keras прави това изключително лесно, както можем да видим в следващия код.
- Забележете начина, по който уточняваме, че изходът от автокодера е просто изход от енкодера, след като е преминал през декодера.

Свързване на енкодера и декодера (пълен автоенкодер)

```
autoencoder = Model(encoder_input, decoder(encoder_output)) ❶
```

1. Моделът на Keras дефинира пълния автоенкодер, който взема изображение и го прекарва през енкодера и обратно през декодера за да генерира реконструкция на оригиналното изображение.

Компилиране на модела

```
# Compile the autoencoder
autoencoder.compile(optimizer="adam", loss="binary_crossentropy")
```

След като дефинирахме нашия модел, просто трябва да го компилираме с функция за загуба и оптимизатор, както е показано.

Функцията на загубата обикновено се избира да бъде средна квадратична грешка (RMSE) или двоична кръстосана ентропия между отделните пиксели на оригиналното изображение и реконструкцията.

Избор на функция на загуба

- Оптимизирането за RMSE означава, че генерираният изход ще бъде симетрично разпределен около средните стойности на пикселите (тъй като надценяването се санкционира еквивалентно на подценяването).
- От друга страна, двоичната загуба на кръстосана ентропия е асиметрична - тя наказва грешките към крайностите по-силно, отколкото грешките към центъра.

Избор на функция на загуба

- Например, ако истинската стойност на пиксела е висока (напр., 0.7), тогава генерирането на пиксел със стойност 0.8 се наказва по-тежко от генерирането на пиксел със стойност 0.6.
- Ако истинската стойност на пиксела е ниска (напр., 0.3), тогава генерирането на пиксел със стойност 0.2 се наказва по-тежко от генерирането на пиксел със стойност 0.4.

Избор на функция на загуба

- Това има ефект при двоична кръстосана ентропия, създаваща малко по-мъгливи изображения в сравнение с RMSE (тъй като има тенденция да тласка прогнозите към 0.5), но понякога това е желателно, тъй като RMSE може да доведе до явно пикселизирани ръбове.
- Няма правилен или грешен избор - трябва да изберем това, което работи най-добре за нашия случай на употреба след експериментиране.

Обучение

```
autoencoder.fit(
    x_train,
    x_train,
    epochs=5,
    batch_size=100,
    shuffle=True,
    validation_data=(x_test, x_test),
)
```

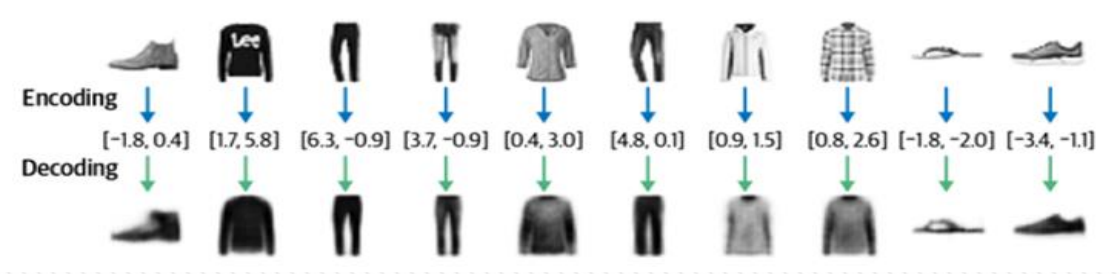
Вече можем да обучим автокодера, като подаваме входните изображения като вход и изход, както е показано.

Сега, когато нашият автоенкодер е обучен, първото нещо, което трябва да проверим, е дали той може точно да реконструира входни изображения.

Реконструиране

```
example_images = x_test[:5000]
predictions = autoencoder.predict(example_images)
```

Можем да тестваме способността да реконструираме изображения, като предаваме изображения от тестовия набор през автокодера и сравняваме изхода с оригиналните изображения.
Кодът за това е показан за пример.



На фигурата са дадени примери за оригинални изображения (горния ред), 2D векторите след кодиране и реконструираните елементи след декодиране (долния ред).

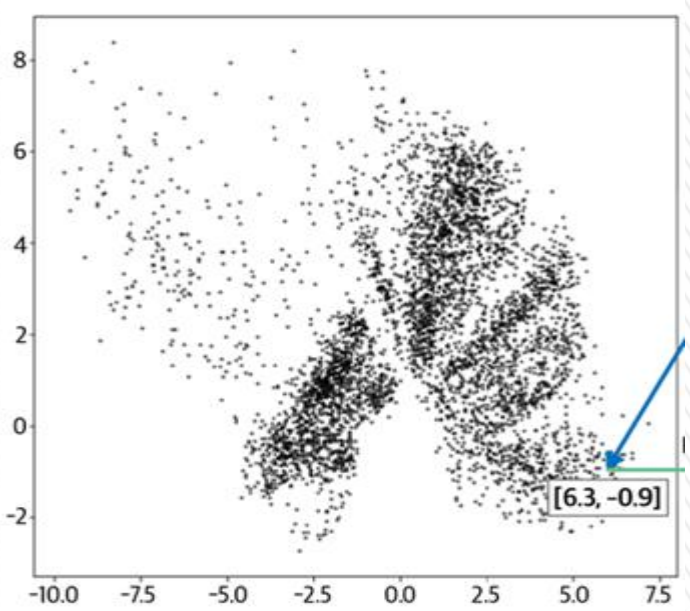
Реконструиране

- Можем да забележим, че реконструкцията не е перфектна - все още има някои детайли от оригиналните изображения, които не са уловени от процеса на декодиране (например лога).
- Това е така, защото като намалим всяко изображение само до две числа, естествено губим част от информацията.

Визуализиране на латентното пространство

```
embeddings = encoder.predict(example_images)

plt.figure(figsize=(8, 8))
plt.scatter(embeddings[:, 0], embeddings[:, 1], c="black", alpha=0.5, s=3)
plt.show()
```

Можем да визуализираме как изображенията се вграждат в латентното пространство, като прекараме тестовия набор през енкодера и начертаем получените вграждания, както е показано в примера.



Получената диаграма е точковата диаграма, показана на фигурата – всяка черна точка представлява изображение, което е вградено в латентното пространство.

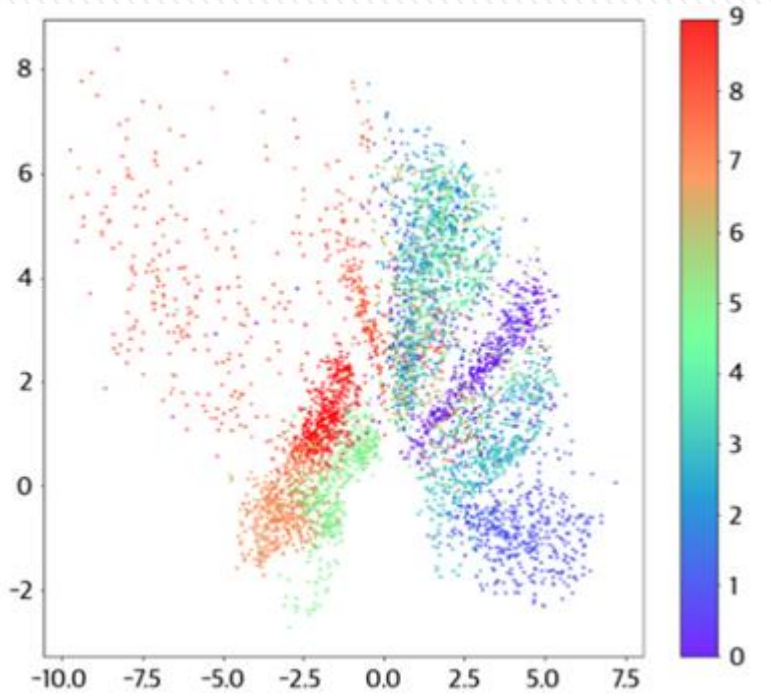
Етикети

ID	Clothing label
0	T-shirt/top
1	Trouser
2	Pullover
3	Dress
4	Coat
5	Sandal
6	Shirt
7	Sneaker
8	Bag
9	Ankle boot

За да разберем по-добре как е структурирано това латентно пространство, можем да използваме етикетите, които идват с набора от данни Fashion-MNIST, описвайки типа артикул във всяко изображение.

Има общо 10 групи, показани в таблицата.

Цветно представяне



Можем да оцветим всяка точка въз основа на етикета на съответното изображение, за да създадем графиката на фигурата.

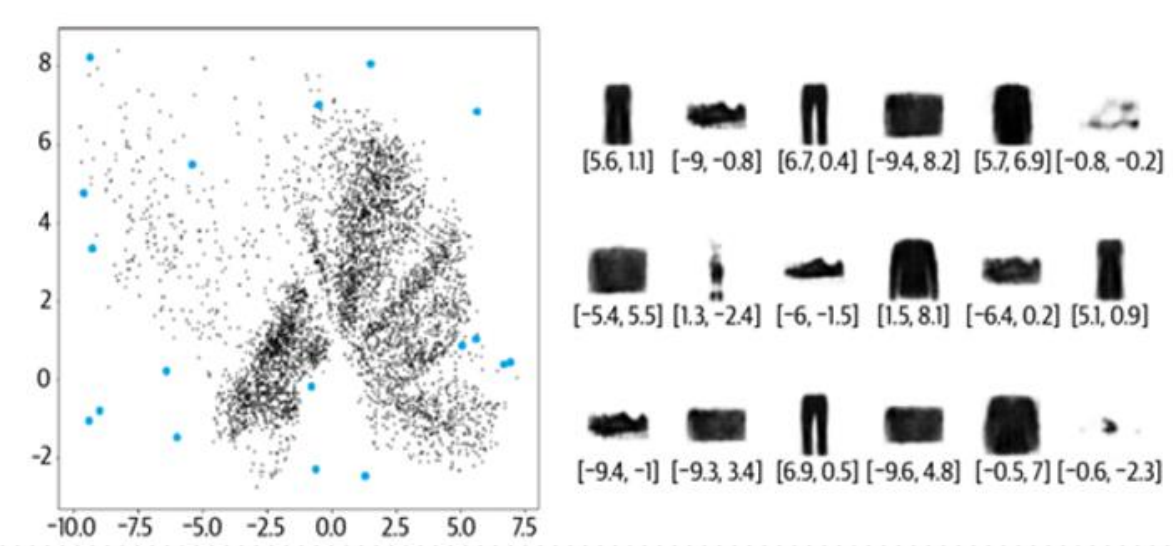
Така структурата става много ясна.

Въпреки че етикетите на дрехите никога не са били показвани на модела по време на обучението, автокодерът естествено е групирал артикули, които изглеждат еднакви в едни и същи части на латентното пространство.

Напр., тъмносиният облак от точки в долния десен ъгъл на скритото пространство са различни изображения на панталони, а червеният облак от точки към центъра са ботуши.

Генериране на нови изображения

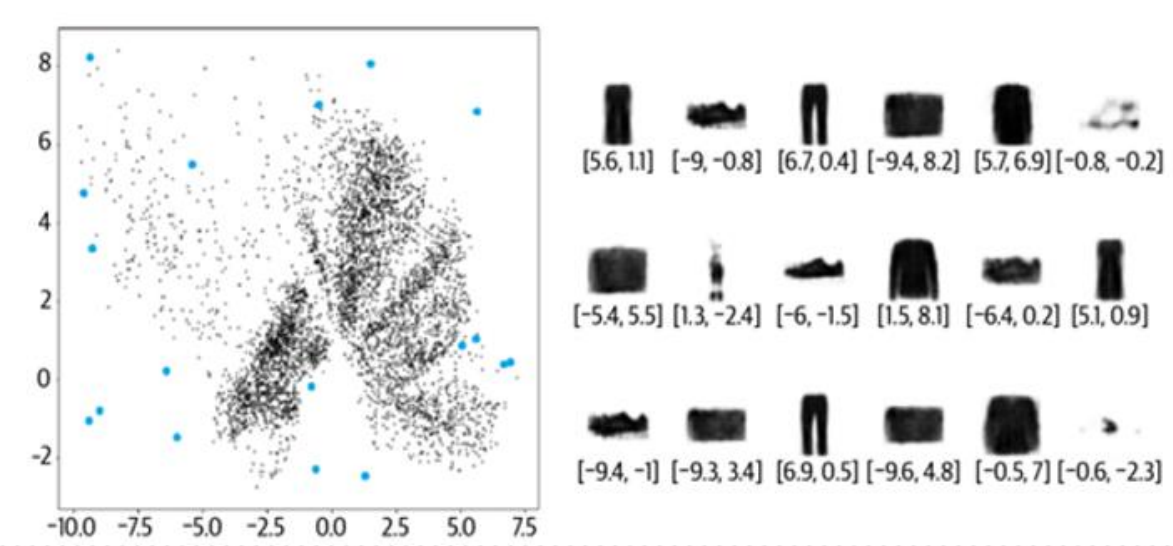
```
mins, maxs = np.min(embeddings, axis=0), np.max(embeddings, axis=0)
sample = np.random.uniform(mins, maxs, size=(18, 2))
reconstructions = decoder.predict(sample)
```



Можем да генерираме нови изображения, като вземем проби от някои точки в латентното пространство и използваме декодера, за да ги преобразуваме обратно в пикселно пространство, както е показано в примера.

Генериране на нови изображения

```
mins, maxs = np.min(embeddings, axis=0), np.max(embeddings, axis=0)
sample = np.random.uniform(mins, maxs, size=(18, 2))
reconstructions = decoder.predict(sample)
```

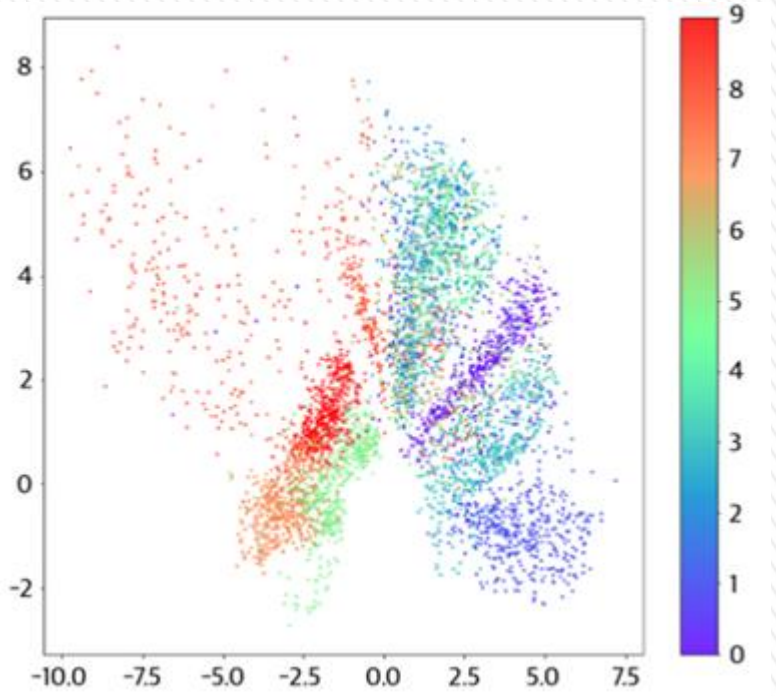


Всяка синя точка се свързва с едно от изображенията, показани вдясно на диаграмата, с вектора за вграждане, показан отдолу.

Забележете как някои от генерираните елементи са по-реалистични от други.

Защо е това?

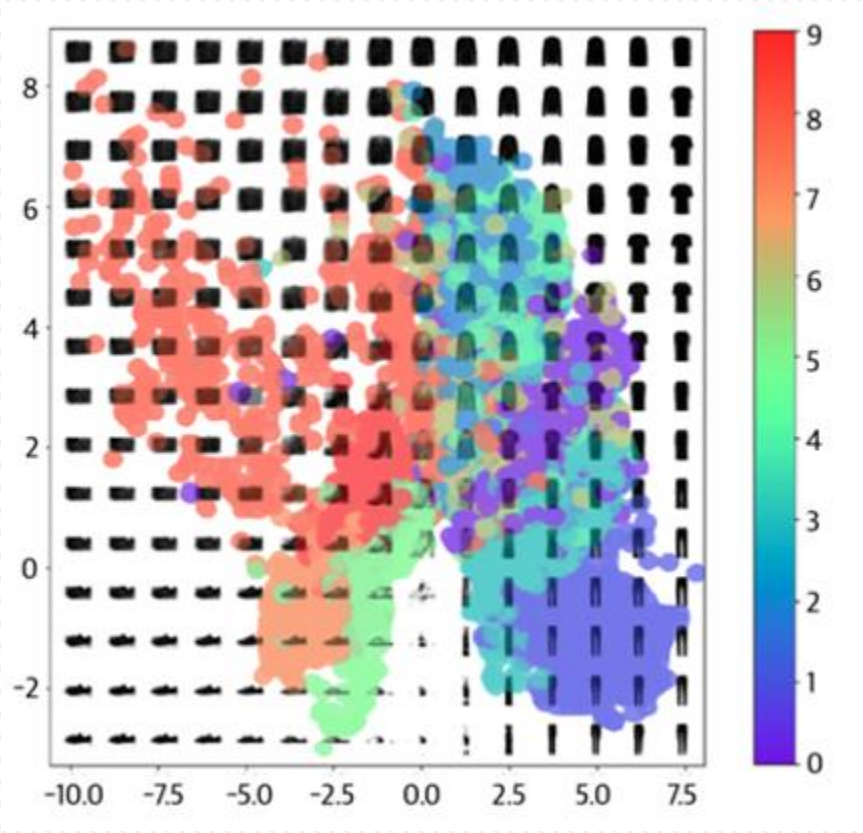
Анализ



За да отговорим на този въпрос, нека първо направим няколко наблюдения относно цялостното разпределение на точките в латентното пространство:

- ✓ Някои облекла са представени на много малка площ, а други на много по-голяма площ.
- ✓ Разпределението не е симетрично спрямо точката (0, 0) или ограничено. Например, има много повече точки с положителни стойности на оста y, отколкото отрицателни, а някои точки дори се простират до стойност на оста y > 8.
- ✓ Има големи празнини между цветовете, съдържащи малко точки.

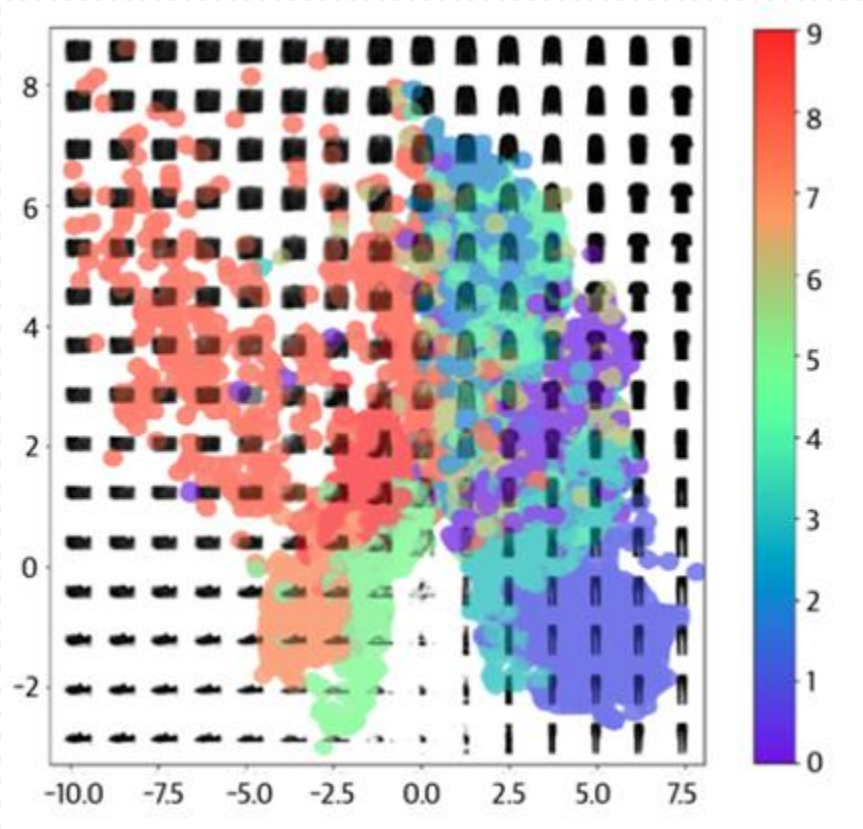
Анализ



Тези наблюдения всъщност правят вземането на проби от латентното пространство доста предизвикателно.

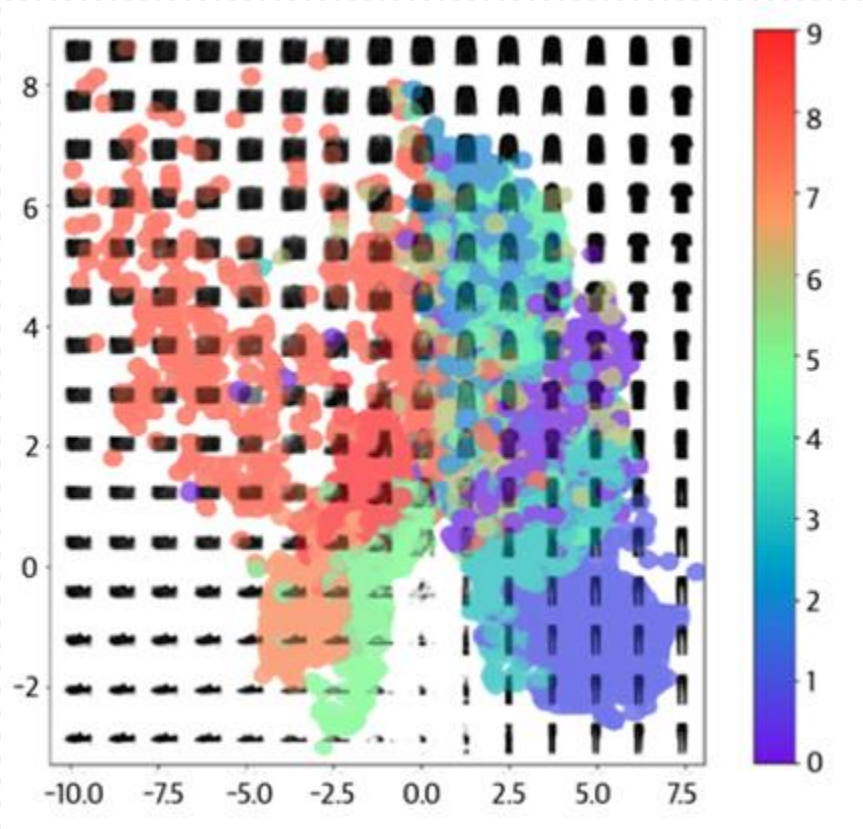
Ако наслагваме латентното пространство с изображения на декодирани точки върху решетка, както е показано на фигурата, можем да започнем да разбираме защо декодерът не може винаги да генерира изображения по задоволителен стандарт.

Анализ



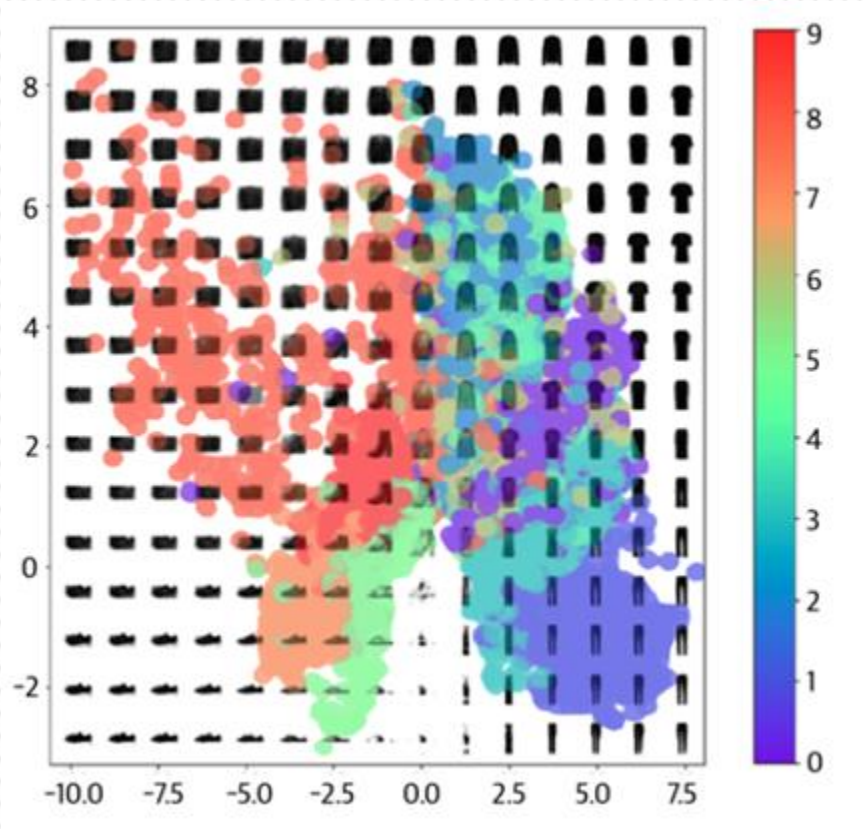
Първо, можем да видим, че ако изберем точки равномерно в ограничено пространство, което дефинираме, е по-вероятно да вземем проба от нещо, което се декодира, за да изглежда като чанта (ID 8), отколкото като ботуш до глезена (ID 9), защото частта от латентното пространство, издълбано за чанти (оранжево), е по-голямо от областта на ботушите на глезена (червено).

Анализ



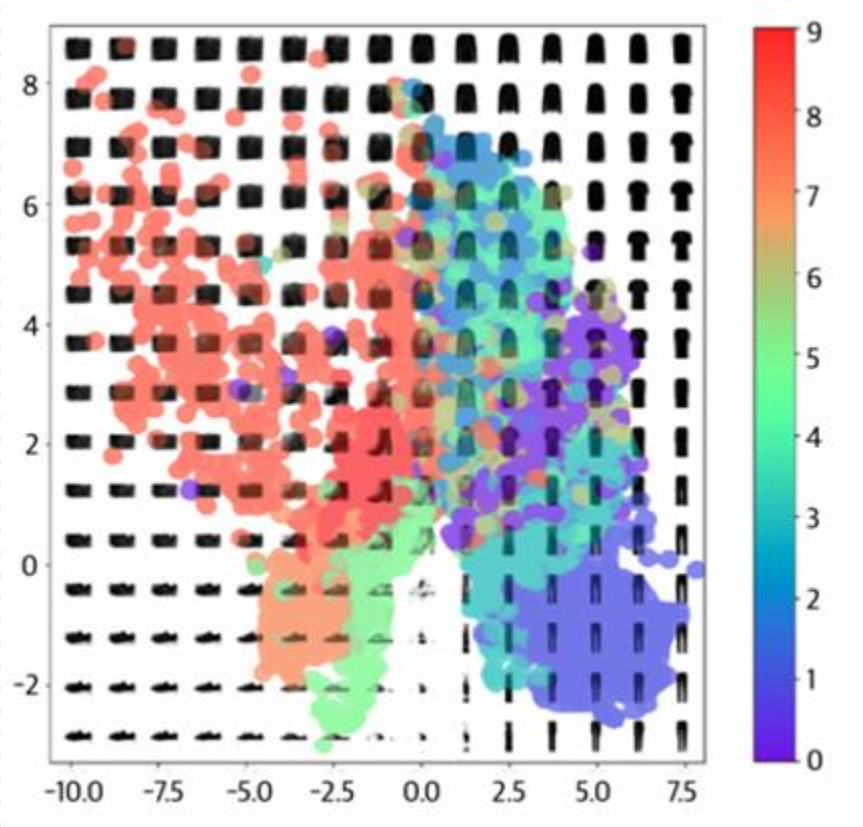
Второ, не е очевидно как трябва да подходим към избора на произволна точка в латентното пространство, тъй като разпределението на тези точки е недефинирано.
Технически бихме били оправдани да изберем която и да е точка в 2D равнината.
Дори не е гарантирано, че точките ще бъдат центрирани около (0, 0).
Това прави вземането на проби от нашето латентно пространство проблематично.

Анализ



И накрая, можем да видим дупки в латентното пространство, където нито едно от оригиналните изображения не е кодирано.
Например, има големи бели интервали в краищата на домейна - автоматичният енкодер няма причина да гарантира, че точките тук са декодирани до разпознаваеми дрехи, тъй като много малко изображения в набора за обучение са кодирани тук.

Анализ



Дори точки, които са централни, може да не бъдат декодирани в добре оформени изображения.
Това е така, защото автокодерът не е принуден да гарантира, че пространството е непрекъснато.
Например, въпреки че точката (–1, –1) може да бъде декодирана, за да даде задоволително изображение на сандал, няма механизъм, който да гарантира, че точката (–1.1, –1.1) също създава задоволително изображение на сандал.

Обобщение

- В две измерения този въпрос е фин - автоенкодерът има само малък брой измерения, с които да работи, така че естествено трябва да смачка групите дрехи заедно, което води до относително малко пространство между групите дрехи.
- Въпреки това, тъй като започваме да използваме повече измерения в латентното пространство, за да генерираме по-сложни изображения като лица, този проблем става още по-очевиден.

Обобщение

- Ако дадем свобода на автоматичното кодиране как да използва латентното пространство за кодиране на изображения ще има огромни пропуски между групи от подобни точки
  - Без стимул за пространствата между тях да генерират добре оформени изображения.
- За да разрешим тези проблеми, трябва да преобразуваме нашия автоенкодер във вариационен автоенкодер.

Вариационни автоенкодери (VAE)

Регистрация

Registration for: ВГИИ

Четвърта седмица

Short URL:https://tinyurl.com/24j394f2

Start at: 17-05-2024 08:45

Status: CREATED

End at: 17-05-2024 09:45

Registrations: 0

Пренареждане на гардероба

- За да обясним VAE нека прегледаме безкрайния гардероб и направим няколко промени.
- Да предположим сега, че вместо да поставим всяко облекло в една точка в гардероба, решаваме да определим обща зона, където е по-вероятно да бъде намерена дрехата.
- Смятаме, че този подход към местоположението на дрехите ще помогне за решаването на проблема с локалните прекъсвания в гардероба.

Пренареждане на гардероба

- Също така, за да сме сигурни, че няма да станем твърде небрежни с новата система се съгласяваме със стилиста, че ще се опитаме да поставим центъра на зоната на всяка дреха възможно най-близо до средата на гардероба
  - Отклонението от центъра трябва да е възможно най-близо, напр. до един метър (не по-малко и не по-голямо).
- Колкото повече се отклоняваме от това правило, толкова повече трябва да плащаме на стилиста.

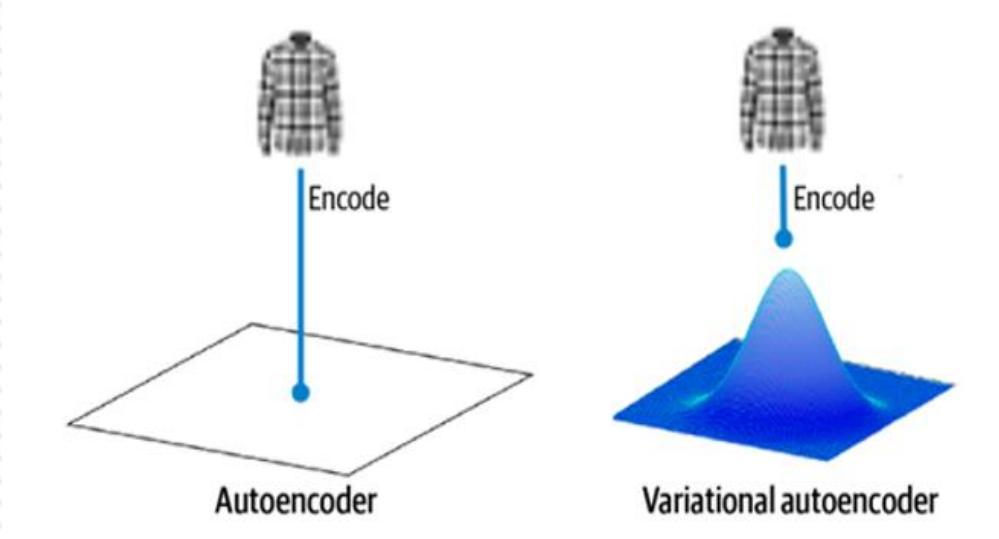
Пренареждане на гардероба

- След няколко месеца работа с тези две прости промени ние се радваме на новото оформление на гардероба, заедно с някои нови дрехи, които стилистът е създал.
- Разнообразието в ушитите дрехи е голямо, като този път няма некачествени дрехи.
- Изглежда двете промени са направили всичко различно.

Вариационен автоенкодер

- Нека сега се опитаме да разберем какво трябва да направим с нашия модел на автоенкодер за да го преобразуваме във вариационен автоенкодер и по този начин да го направим по-сложен генеративен модел.
- Двете части, които трябва да променим, са енкодерът и функцията на загуба.

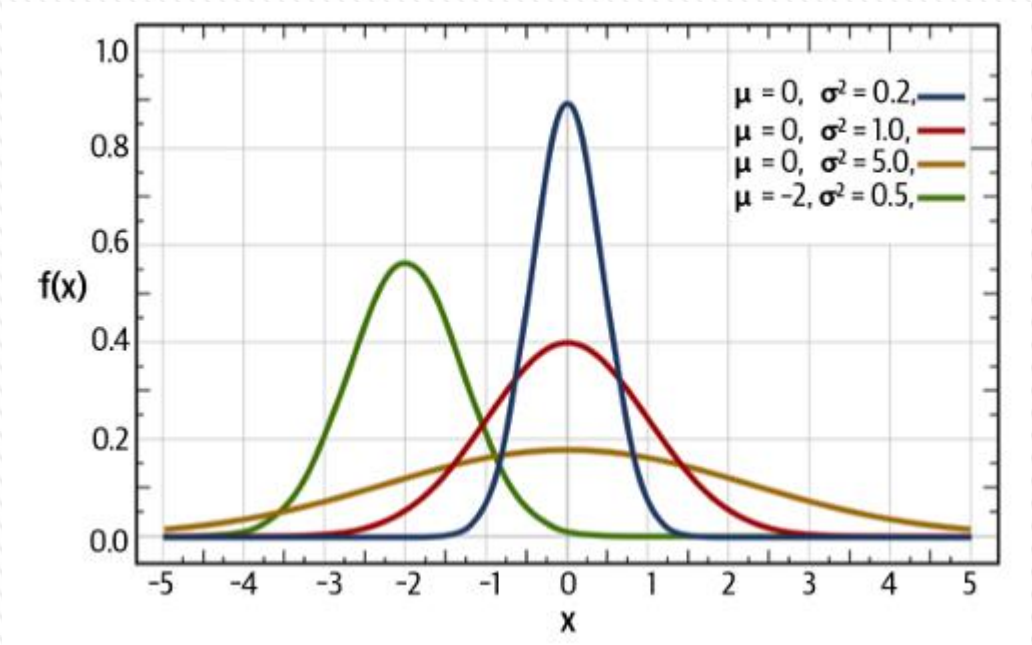
Вариационен енкодер



В автокодера всяко изображение се картографира директно към една точка в латентното пространство.

Във вариационен автоенкодер всяко изображение вместо това се картографира към многовариантно нормално разпределение около една точка в латентното пространство, както е показано на фигурата.

Гаусово разпределение



Многомерното нормално разпределение:
Нормално разпределение (или Гаусово разпределение) N(μ,σ) е вероятностно разпределение, характеризиращо се с отличителна форма на камбановидна крива, дефинирана от две променливи:

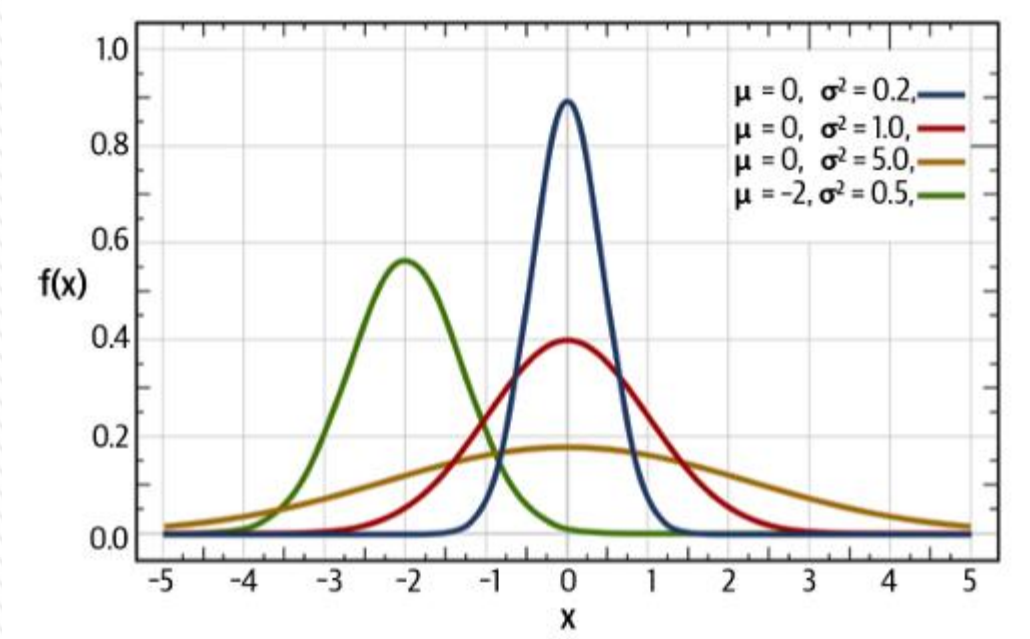
- ✓ Средна стойност (μ)
- ✓ Дисперсия (σ^2).

Стандартното отклонение (σ) е корен квадратен от дисперсия.

$$f\left(x \mid \mu, \sigma^2\right) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

Функция на плътност

Гаусово разпределение



Фигурата показва няколко нормални разпределения в едно измерение за различни стойности на средната стойност и дисперсията.

Червената крива е стандартната норма (или единица норма) N(0,1) — нормалното разпределение със средна стойност 0 и дисперсия 1.

Многовариантно нормално разпределение

- Обикновено използваме изотропни многовариантни нормални разпределения, където ковариационната матрица е диагонална.
- Това означава, че разпределението е независимо във всяко измерение (т.е. можем да вземем вектор на извадка, където всеки елемент е нормално разпределен с независима средна стойност и дисперсия).
- Такъв е случаят с многовариантното нормално разпределение, което ще използваме в нашия вариационен автоенкодер.

Функция на енкодера

- Енкодерът трябва само да картографира всеки вход към среден вектор и вектор на вариация и не трябва да се тревожи за ковариацията между измеренията.
- Вариационните автоенкодери предполагат, че няма корелация между измеренията в латентното пространство.
- Стойностите на дисперсията винаги са положителни, така че всъщност избираме да картографираме логаритъма на дисперсията, тъй като това може да отнеме всяко реално число в диапазона $(-\infty,\infty)$.
- По този начин можем да използваме невронна мрежа като енкодер за извършване на картографиране от входното изображение към векторите на средната стойност и логаритмичната дисперсия.

Обобщение

- За да обобщим, енкодерът ще вземе всяко входно изображение и ще го кодира в два вектора, които заедно определят многовариантно нормално разпределение в латентното пространство:
  - z_mean: Средната точка на разпределението;
  - z_log_var: Логаритъмът на дисперсията на всяко измерение.

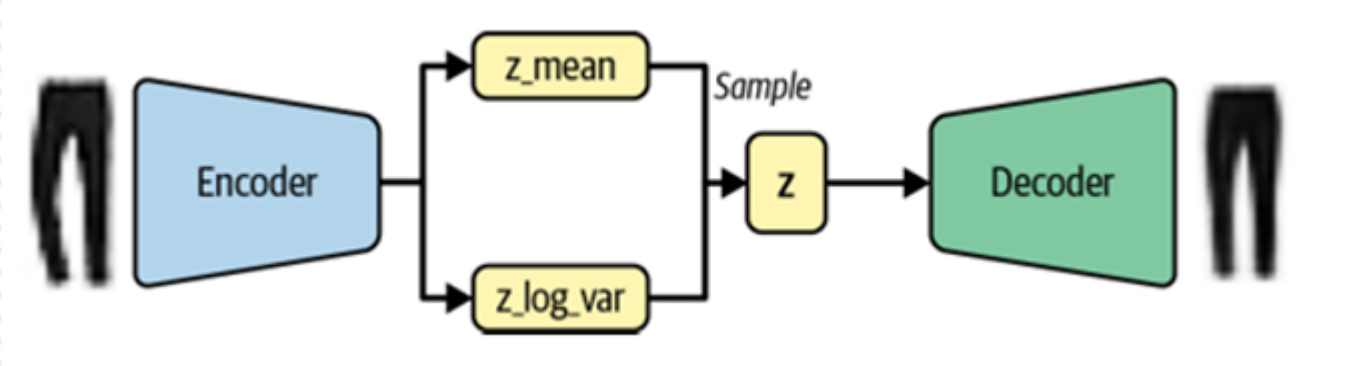
Обобщение

- Можем да извадим точка z от разпределението, дефинирано от тези стойности, като използваме следното уравнение:
  - z = z_mean + z_sigma * epsilon, където:

    z_sigma = exp(z_log_var * 0.5)

    epsilon ~ N(0,I)

Обща схема



Защо тази малка промяна в енкодера помага?

- По-рано видяхме, че няма изискване латентното пространство да бъде непрекъснато – дори ако точката (–2, 2) се декодира до добре оформено изображение на сандал, няма изискване (–2.1, 2.1) да изглежда подобен.
- Понеже вземаме проби от произволна точка от област около z_mean, декодерът трябва да гарантира, че всички точки в същия квартал произвеждат много подобни изображения, когато се декодират, така че загубата на реконструкция да остане малка.
- Това е значимо свойство, което гарантира, че дори когато изберем точка в латентното пространство, която никога не е била виждана от декодера, има вероятност да се декодира до изображение, което е добре оформено.

Създаване на VAE енкодер

```
class Sampling(layers.Layer): ❶
    def call(self, inputs):
        z_mean, z_log_var = inputs
        batch = tf.shape(z_mean)[0]
        dim = tf.shape(z_mean)[1]
        epsilon = K.random_normal(shape=(batch, dim))
        return z_mean + tf.exp(0.5 * z_log_var) * epsilon ❷
```

Първо, трябва да създадем нов тип слой за вземане на проби, който ще ни позволи да вземаме проби от разпределението, дефинирано от z_mean и z_log_var, както е показано в примера.

1. Създаваме нов слой:

- ✓ Можем да създаваме нови слоеве в Keras, като подклас на абстрактния клас Layer
- ✓ Дефинираме метода call, който описва как тензорът се трансформира от слоя.
- ✓ Например във вариационния автоенкодер можем да създадем Sampling слой, който може да обработва вземането на проби от z от нормално разпределение с параметри, дефинирани от z_mean и z_log_var.

Създаване на VAE енкодер

```
class Sampling(layers.Layer): ❶
    def call(self, inputs):
        z_mean, z_log_var = inputs
        batch = tf.shape(z_mean)[0]
        dim = tf.shape(z_mean)[1]
        epsilon = K.random_normal(shape=(batch, dim))
        return z_mean + tf.exp(0.5 * z_log_var) * epsilon ❷
```

2. Използваме препараметризиране за да изградим извадка от нормалното разпределение, параметризирано от z_mean и z_log_var.

- ✓ Вместо да вземаме проби директно от нормално разпределение с параметри z_mean и z_log_var, можем да вземаме проби epsilon от стандартна норма и след това ръчно да коригираме извадката, за да имаме правилната средна стойност и дисперсия.
- ✓ Това е известно като трик за препараметризиране и е важно, тъй като означава, че градиентите могат да се разпространяват обратно свободно през слоя.
- ✓ Чрез запазване на цялата случайност на слоя, съдържащ се в променливата epsilon, частната производна на изхода на слоя по отношение на неговия вход може да се покаже като детерминистична (т.е. независима от произволния epsilon), което е от съществено значение обратното разпространение през слой да е възможно.

Пълен код на енкодера

```
encoder_input = layers.Input(
    shape=(32, 32, 1), name="encoder_input"
)
x = layers.Conv2D(32, (3, 3), strides=2, activation="relu", padding="same")(
    encoder_input
)
x = layers.Conv2D(64, (3, 3), strides=2, activation="relu", padding="same")(x)
x = layers.Conv2D(128, (3, 3), strides=2, activation="relu", padding="same")(x)
shape_before_flattening = K.int_shape(x)[1:]

x = layers.Flatten()(x)
z_mean = layers.Dense(2, name="z_mean")(x) ❶
z_log_var = layers.Dense(2, name="z_log_var")(x)
z = Sampling()([z_mean, z_log_var]) ❷

encoder = models.Model(encoder_input, [z_mean, z_log_var, z], name="encoder") ❸
```

1. Вместо да свързваме слоя Flatten директно с 2D латентното пространство, ние го свързваме със слоевете z_mean и z_log_var.
2. Слоят за вземане на проби взема точка z в латентното пространство от нормалното разпределение, дефинирано от параметрите z_mean и z_log_var.
3. Моделът Keras, който дефинира енкодера – модел, който взема входно изображение и извежда z_mean, z_log_var и точка z на извадка от нормалното разпределение, дефинирано от тези параметри.

Архитектура на енкодера

Layer (type)	Output shape	Param #	Connected to
InputLayer (input)	(None, 32, 32, 1)	0	[]
Conv2D (conv2d_1)	(None, 16, 16, 32)	320	[input]
Conv2D (conv2d_2)	(None, 8, 8, 64)	18,496	[conv2d_1]
Conv2D (conv2d_3)	(None, 4, 4, 128)	73,856	[conv2d_2]
Flatten (flatten)	(None, 2048)	0	[conv2d_3]
Dense (z_mean)	(None, 2)	4,098	[flatten]
Dense (z_log_var)	(None, 2)	4,098	[flatten]
Sampling (z)	(None, 2)	0	[z_mean, z_log_var]

Total params	100,868
Trainable params	100,868
Non-trainable params	0

Функция на загубата

Допълнителен компонент

- Предишната функция на загуба се състоеше само от загуба на реконструкция между изображения и техните опити за копиране след преминаване през енкодера и декодера.
- Загубата на реконструкция също се появява във вариационен автоенкодер, но сега се нуждаем от един допълнителен компонент: дивергенция на Kullback–Leibler (KL).

KL

$$D_{KL}[N(\mu, \sigma \parallel N(0,1)] = -\frac{1}{2}\sum\left(1 + log\left(\sigma^2\right) - \mu^2 - \sigma^2\right)$$

KL дивергенцията е начин за измерване на това колко едно вероятностно разпределение се различава от друго.
В VAE искаме да измерим колко нашето нормално разпределение с параметри z_mean и z_log_var се различава от стандартното нормално разпределение.

В този специален случай може да се покаже, че KL дивергенцията има следната затворена форма:
kl_loss = -0.5 * sum(1 + z_log_var - z_mean ^ 2 - exp(z_log_var))
или в математическа нотация:

KL

- Сумата се взема за всички измерения в латентното пространство.
- kl_loss е сведен до минимум до 0, когато z_mean = 0 и z_log_var = 0 за всички измерения.
- Тъй като тези два термина започват да се различават от 0, kl_loss се увеличава.
- В обобщение, терминът за дивергенция на KL наказва мрежата за кодиране на наблюдения относно променливите z_mean и z_log_var, които се различават значително от параметрите на стандартно нормално разпределение (z_mean = 0 и z_log_var = 0).

Защо това допълнение към функцията за загуба помага?

- Първо, сега имаме добре дефинирано разпределение, което можем да използваме за избор на точки в латентното пространство и това е стандартното нормално разпределение.
- Второ, тъй като KL се опитва да принуди всички кодирани разпределения към стандартното нормално разпределение, има по-малък шанс да се образуват големи празнини между точкови клъстери.
- Така енкодерът ще се опита да използва пространството около началото симетрично и ефективно.

Обучение на VAE

VAE като подклас на Keras

```python
class VAE(models.Model):
    def __init__(self, encoder, decoder, **kwargs):
        super(VAE, self).__init__(**kwargs)
        self.encoder = encoder
        self.decoder = decoder
        self.total_loss_tracker = metrics.Mean(name="total_loss")
        self.reconstruction_loss_tracker = metrics.Mean(
            name="reconstruction_loss"
        )
        self.kl_loss_tracker = metrics.Mean(name="kl_loss")

    @property
    def metrics(self):
        return [
            self.total_loss_tracker,
            self.reconstruction_loss_tracker,
            self.kl_loss_tracker,
        ]
```

Примерът показва как изграждаме цялостния VAE модел като подклас на абстрактния клас Keras Model.

Това ни позволява да включим изчисляването на члена на дивергенцията на KL на функцията на загубата в персонализиран метод train_step.

VAE като подклас на Keras

```
def call(self, inputs): ❶
    z_mean, z_log_var, z = encoder(inputs)
    reconstruction = decoder(z)
    return z_mean, z_log_var, reconstruction

def train_step(self, data): ❷
    with tf.GradientTape() as tape:
        z_mean, z_log_var, reconstruction = self(data)
        reconstruction_loss = tf.reduce_mean(
            500
            * losses.binary_crossentropy(
                data, reconstruction, axis=(1, 2, 3)
            )
        ) ❸
        kl_loss = tf.reduce_mean(
            tf.reduce_sum(
                -0.5
                * (1 + z_log_var - tf.square(z_mean) - tf.exp(z_log_var)),
                axis = 1,
            )
        )
        total_loss = reconstruction_loss + kl_loss ❹

    grads = tape.gradient(total_loss, self.trainable_weights)
```

1. Тази функция описва какво бихме искали да върне това, което наричаме VAE на конкретно входно изображение.
2. Тази функция описва една стъпка на обучение на VAE, включително изчисляването на функцията на загубата.
3. Бета стойност от 500 се използва при загубата на реконструкция.
4. Общата загуба е сумата от загубата при реконструкция и загубата от дивергенция на KL.

VAE като подклас на Keras

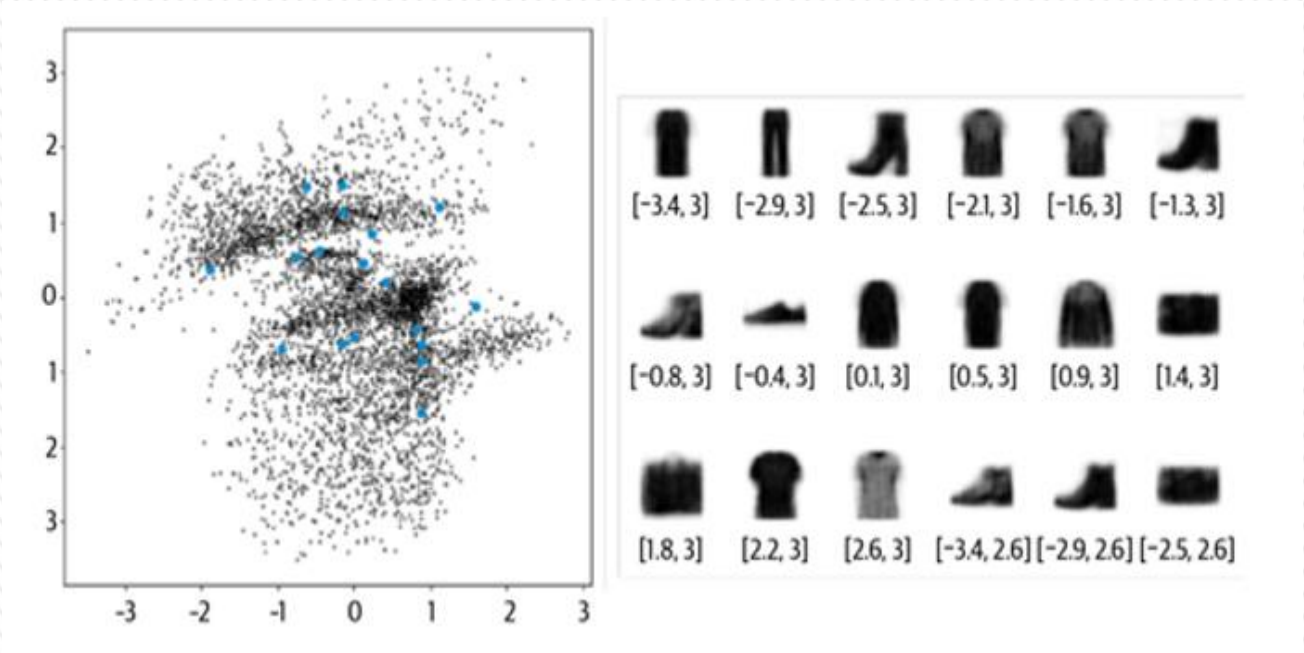
```
        self.optimizer.apply_gradients(zip(grads, self.trainable_weights))

        self.total_loss_tracker.update_state(total_loss)
        self.reconstruction_loss_tracker.update_state(reconstruction_loss)
        self.kl_loss_tracker.update_state(kl_loss)

        return {m.name: m.result() for m in self.metrics}

vae = VAE(encoder, decoder)
vae.compile(optimizer="adam")
vae.fit(
    train,
    epochs=5,
    batch_size=100
)
```

Анализ на VAE

Използване

- Сега, след като обучихме нашия VAE, можем да използваме енкодера, за да кодираме изображенията в тестовия набор и да начертаем стойностите на z_mean в латентното пространство.
- Можем също да вземем проби от стандартно нормално разпределение, за да генерираме точки в латентното пространство и да използваме декодера, за да декодираме тези точки обратно в пространството на пикселите, за да видим как работи VAE.

Използване

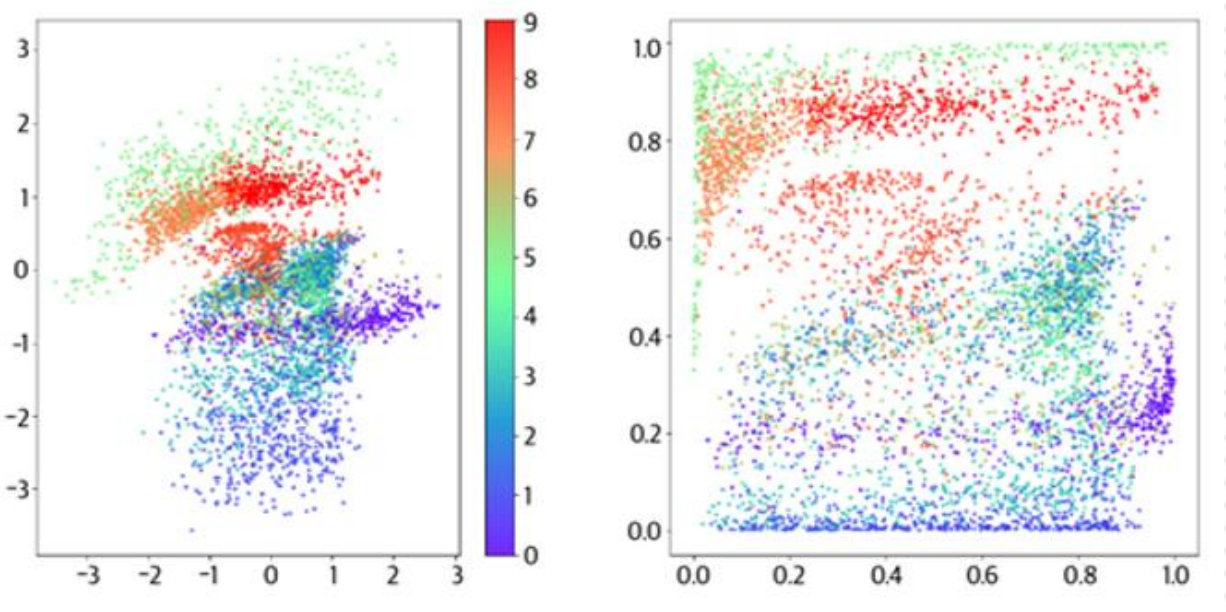


Фигурата показва структурата на новото латентно пространство, заедно с някои избрани точки и техните декодирани изображения.
Веднага можем да видим няколко промени в начина, по който е организирано латентното пространство.

Използване

- Първо, терминът за загуба на дивергенция на KL гарантира, че стойностите z_mean и z_log_var на кодираните изображения никога не се отклоняват твърде далеч от стандартното нормално разпределение.
- Второ, няма толкова много лошо оформени изображения, тъй като латентното пространство сега е много по-непрекъснато, поради факта, че енкодерът вече е стохастичен, а не детерминиран.

Използване



И накрая, чрез оцветяване на точки в латентното пространство по тип облекло (фигурата), можем да видим, че няма преференциално третиране на нито един тип. Дясната графика показва пространството, трансформирано в p-стойности - можем да видим, че всеки цвят е приблизително еднакво представен.
Отново е важно да запомните, че етикетите изобщо не са използвани по време на обучението - VAE е научил сам различните форми на облекло, за да помогне за минимизиране на загубите при реконструкция.


Благодаря за вниманието!