Relatório 27 - Modelos Generativos

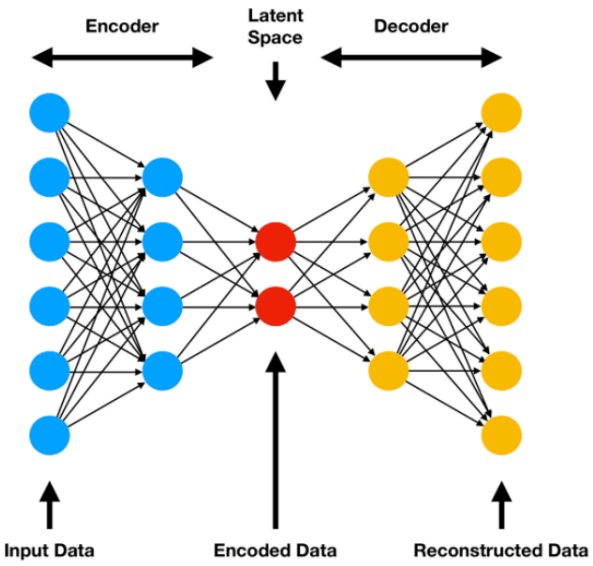
Guilherme Loan Schneider

Descrição da atividade

1. Variational Auto-Encoders (VAE's)

As redes neurais AutoEncoders são um tipo específico treinado para copiar sua entrada para sua saída. A ideia principal desse tipo de rede é aprender uma representação codificada dos dados, chamada de "representação latente", reduzindo a dimensionalidade da entrada no processo. Isso é feito por meio de duas partes principais: o codificador (encoder), que comprime os dados em uma forma reduzida, e o decodificador (decoder), que tenta reconstruir a entrada original a partir dessa representação comprimida.

Autoencoders são úteis para redução de dimensionalidade, remoção de ruído (denoising autoencoders), geração de dados e aprendizado não supervisionado de características. Eles são treinados usando um critério de erro de reconstrução, como o erro quadrático médio entre a entrada e a saída reconstruída, incentivando a rede a aprender os padrões mais relevantes dos dados.

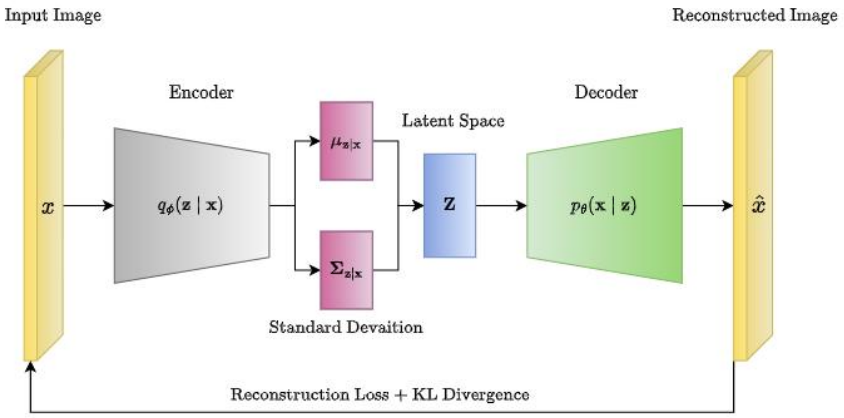


O VAE, ou Autoencoder Variacional, é um tipo de modelo de aprendizado profundo usado principalmente para aprendizado de representação e geração de dados. Ele se destaca por aprender uma representação latente contínua dos dados, permitindo a geração de novas amostras similares às originais.

A diferença fundamental dos VAEs é que, em vez de aprender um único ponto no espaço latente para cada entrada, eles aprendem uma distribuição de probabilidade (a partir da inferência bayesiana). Isso permite que o modelo gere novas amostras aleatoriamente a partir dessa distribuição.

Uma boa forma de entender como o decoder dos VAEs funciona é pensando que o decoder funciona como uma Rede Neural Convolucional ao contrário, onde a CNN tem uma imagem de entrada e reduz a sua dimensionalidade até um vetor de uma

dimensão. No VAE, a ideia é a partir do vetor (Latent Space na Figura abaixo), reconstruir uma imagem de tamanho igual à entrada no encoder.



Kullback-Leibler Divergence

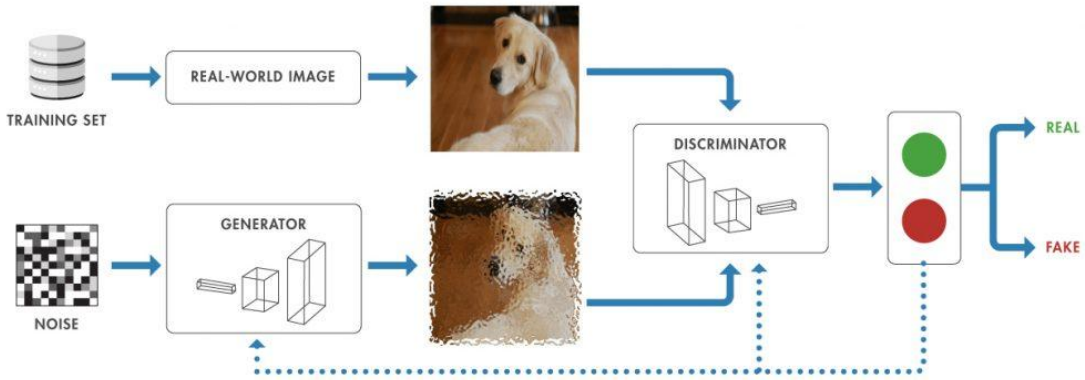
O KL Divergence é uma forma que podemos calcular a distância entre distribuições de probabilidade entre os dados originais e os reconstruídos, ou seja, entender o quão parecidos são os dados gerados. Ela é considerada como uma métrica de erro.

O autor comenta um pouco sobre o cálculo matemático envolvido para analisar essa métrica, mas cita também que pode ser equivalente a entropia relativa, onde o cálculo utilizado nessa é o valor de CrossEntropy(p,q) – Entropy(p).

2. Generative Adversarial Networks (GANs)

Esse tipo de rede neural é utilizado como base para gerar deepfakes, que consistem em gerar imagens falsas na internet, como gerar rostos de pessoas que não existem. Claro, essa é apenas uma das milhares de aplicações.

A ideia das GANs funciona da seguinte maneira, existem duas redes neurais em que uma delas é a geradora, então a geradora das imagens fictícias, e uma discriminadora, que tem o objetivo de verificar as imagens gerada pela rede geradora e dizer se ela é verdadeira ou falsa.



A GAN geradora é treinada com imagens do que o usuário quer gerar, como rostos, pinturas, objetos, etc. Por outro lado, a discriminadora precisa ser treinada com imagens que são reais e falsas, indicando qual é qual.

O objetivo por trás das duas é o seguinte: A geradora deve melhorar até chegar a certo ponto em que a discriminadora não consegue mais afirmar com boa precisão se aquela imagem é real ou falsa (chegando em 50% de precisão). Quando esse valor é atingido, o treinamento é interrompido e a discriminadora é então é treinada para conseguir discernir com precisão as imagens geradas, dizendo se elas são falsas ou verdadeiras. A ideia é manter esse ciclo, sempre melhorando ambas as redes.

O grande problema enfrentado é manter o balanceamento entre as duas, ou seja, a geradora não pode deixar a discriminadora com uma precisão muito ruim, e vice-versa, pois pode ocorrer do discriminador não conseguir recuperar e entender as imagens geradas, ou o gerador não conseguir mais criar imagens boas o suficiente para o discriminador classificar como verdadeira.

Let's build GPT: from scratch, in code, spelled out.

A ideia dessa aula é criar um modelo de processamento de linguagem natural que criará obras de Shakespeare continuamente.

O treinamento do modelo é feito utilizando um documento com todas as obras do Shakespeare em sequência, onde o método que utilizaremos para realizar esse treinamento é com base em caracteres individuais, não palavras inteiras. Rodando um código simples conseguimos ver que o texto possui mais de um milhão de caracteres.

```
print(f'Tamanho do texto: {len(text)} caracteres')
✓ 0.0s

Tamanho do texto: 1115394 caracteres
```

O autor da aula até comenta que existem outros métodos, como o que o ChatGPT usa, que utiliza subwords, então não é nem uma palavra por completo, e nem por caracteres individuais.

Como a tokenização que estamos utilizando é com base em caracteres, nosso espaço de tokens não será muito grande, dado que nesse texto existem caracteres limitados ao alfabeto e alguns caracteres especiais, totalizando 65 caracteres únicos. Na imagem abaixo conseguimos ver a tokenização por caracteres, que sempre respeitará o intervalo de 0 a 64.

```
# Criando um mapeamento de caracteres para inteiros e vice-versa
stoi = { ch:i for i,ch in enumerate(chars) } # palavra para índice
itos = { i:ch for i,ch in enumerate(chars) } # índice para palavra
encode = lambda s: [stoi[c] for c in s] # conversão de caracteres para lista de índices
decode = lambda l: ''.join([itos[i] for i in l]) # conversão de lista de índices para caracteres

print(encode("hii there"))
print(decode(encode("hii there")))
✓ 0.0s

[46, 47, 47, 1, 58, 46, 43, 56, 43]
hii there
```

Temos também o caso abaixo que já é com todo o texto tokenizado.

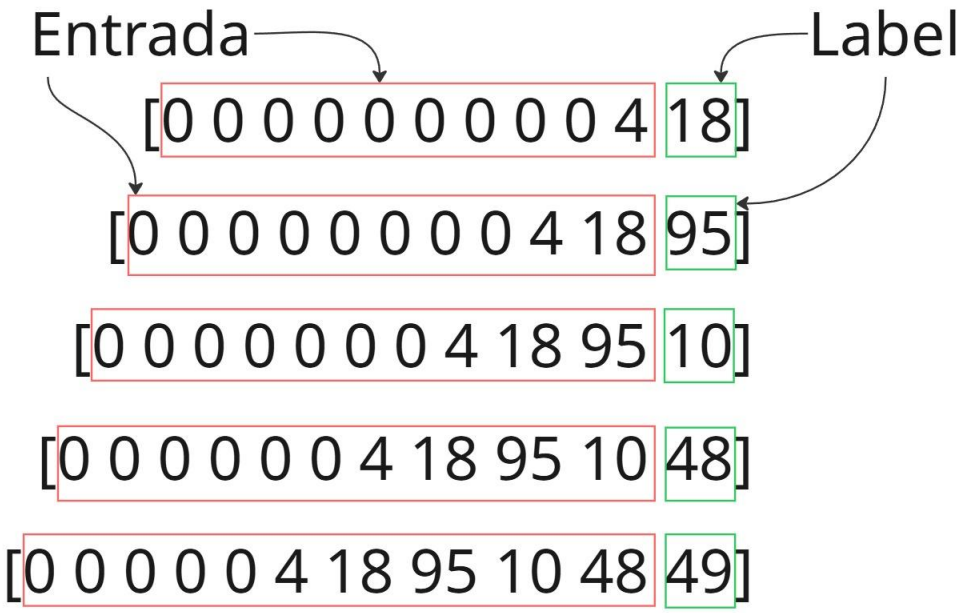
```
import torch
data = torch.tensor(encode(text), dtype=torch.long)
print(data.shape, data.dtype)
print(data[:1000])
```

✓ 13.5s

```
torch.Size([1115394]) torch.int64
tensor([18, 47, 56, 57, 58,  1, 15, 47, 58, 47, 64, 43, 52, 10,  0, 14, 43, 44,
        53, 56, 43,  1, 61, 43,  1, 54, 56, 53, 41, 43, 43, 42,  1, 39, 52, 63,
         1, 44, 59, 56, 58, 46, 43, 56,  6,  1, 46, 43, 39, 56,  1, 51, 43,  1,
        57, 54, 43, 39, 49,  8,  0,  0, 13, 50, 50, 10,  0, 31, 54, 43, 39, 49,
         6,  1, 57, 54, 43, 39, 49,  8,  0,  0, 18, 47, 56, 57, 58,  1, 15, 47,
        58, 47, 64, 43, 52, 10,  0, 37, 53, 59,  1, 39, 56, 43,  1, 39, 50, 50,
         1, 56, 43, 57, 53, 50, 60, 43, 42,  1, 56, 39, 58, 46, 43, 56,  1, 58,
        53,  1, 42, 47, 43,  1, 58, 46, 39, 52,  1, 58, 53,  1, 44, 39, 51, 47,
        57, 46, 12,  0,  0, 13, 50, 50, 10,  0, 30, 43, 57, 53, 50, 60, 43, 42,
         8,  1, 56, 43, 57, 53, 50, 60, 43, 42,  8,  0,  0, 18, 47, 56, 57, 58,
         1, 15, 47, 58, 47, 64, 43, 52, 10,  0, 18, 47, 56, 57, 58,  6,  1, 63,
        53, 59,  1, 49, 52, 53, 61,  1, 15, 39, 47, 59, 57,  1, 25, 39, 56, 41,
        47, 59, 57,  1, 47, 57,  1, 41, 46, 47, 43, 44,  1, 43, 52, 43, 51, 63,
         1, 58, 53,  1, 58, 46, 43,  1, 54, 43, 53, 54, 50, 43,  8,  0,  0, 13,
```

A pergunta que fica é: como iremos fazer pro modelo entender o contexto das palavras/letras?

A ideia aqui vai ser similar a Aula 24 do Bootcamp, sendo o primeiro caso abordado no relatório o que iremos utilizar. Faremos a relação entre letras ser contextual, então o modelo irá analisar uma letra e ver qual é a seguinte no array de targets, em seguida, irá analisar a letra que viu + a seguinte a ela e analisar qual a próxima letra no array target. Ela respeita a lógica abaixo.



Pense que a relação entre as letras será como uma torre de hanoi, onde uma ordem é respeitada, sendo a base maior por primeiro (nosso vetor de entrada), até chegar na base menor (vetor de label). A relação entre as letras é dada à medida que uma peça é colocada na torre.

Visualizando essa ideia em código, ficaria mais ou menos assim:

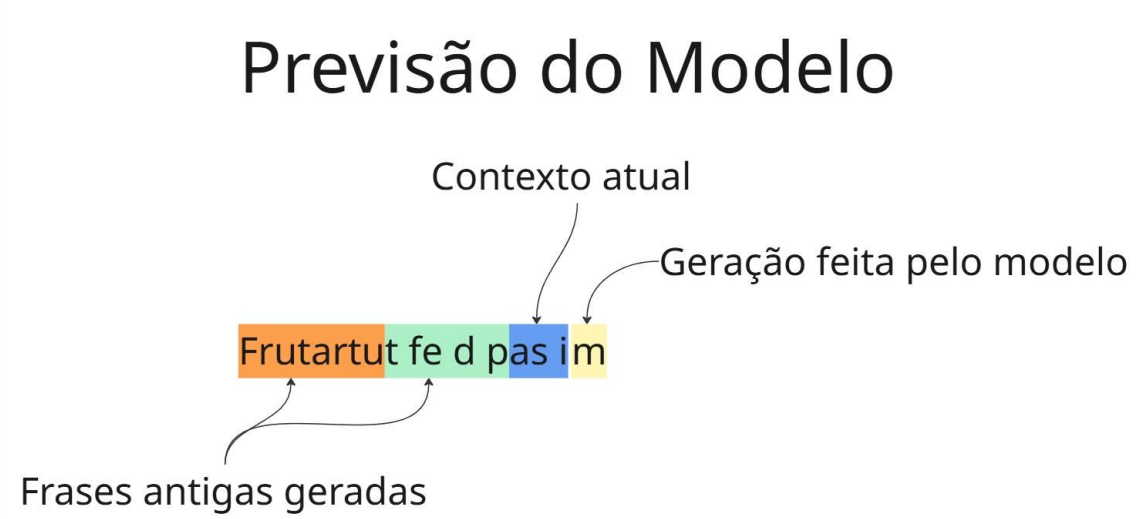
```
Quando o contexto é "tensor([18])" o alvo é "[tensor(47)]"
Quando o contexto é "tensor([18, 47])" o alvo é "[tensor(56)]"
Quando o contexto é "tensor([18, 47, 56])" o alvo é "[tensor(57)]"
Quando o contexto é "tensor([18, 47, 56, 57])" o alvo é "[tensor(58)]"
Quando o contexto é "tensor([18, 47, 56, 57, 58])" o alvo é "[tensor(1)]"
Quando o contexto é "tensor([18, 47, 56, 57, 58,  1])" o alvo é "[tensor(15)]"
Quando o contexto é "tensor([18, 47, 56, 57, 58,  1, 15])" o alvo é "[tensor(47)]"
Quando o contexto é "tensor([18, 47, 56, 57, 58,  1, 15, 47])" o alvo é "[tensor(58)]"
```

Claro que existem inúmeras possibilidades em textos, não necessariamente quando o token “4” aparece, irá imediatamente aparecer o token “18”, para isso serve o treinamento do modelo, que fará com que ele consiga gerar previsões mais prováveis da letra certa. Mas note que ainda precisamos adicionar relações entre as letras, no momento elas não se conversam.

A fim de avaliar como o modelo está desempenhando, realizamos um treinamento de teste com dez mil épocas a partir da estrutura criada no vídeo. A geração feita nessa primeira instância é de 300 palavras. É possível perceber que as palavras não possuem sentido algum devido a falta de contexto e mecanismos para que seja possível o modelo entender como as palavras funcionam.

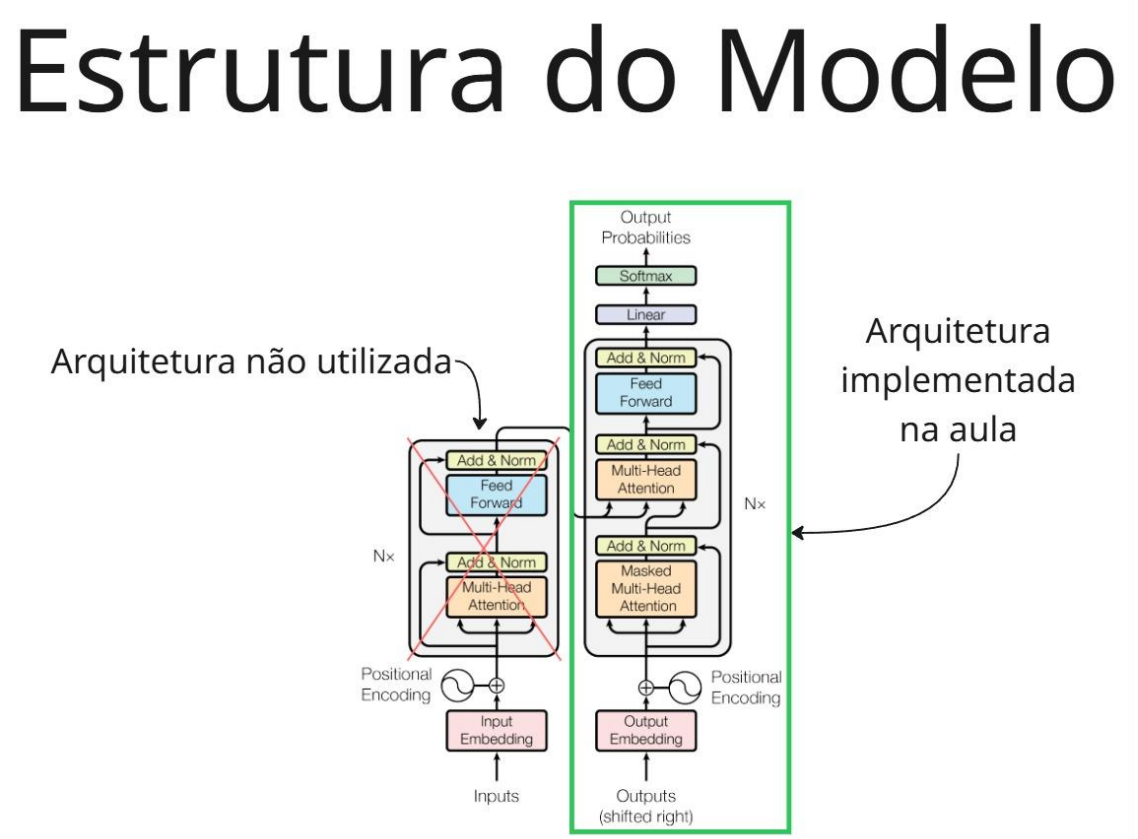
```
TINGAUCAngawind ly fr bu$Howe t
ICEn thy theDWI' alit cofait? 'Becorpllim heouron, tos, thedeeno; me touler'sha ithas, annook ald ow ld.
Whembit d ifene
Frutartut fe d pas imelear the: lak oer ome qullt; hel ofZARENoomeisokns noke y;
He;

PHishoulerotht ir nd shisinomivert, ff h vinpis hod.
Y:
RELLU
```

Atualmente, a geração sempre leva com base uma única letra, então utilizando o exemplo acima como base, na frase “Frutartut fe d pas i”, a geração feita pelo modelo leva de base apenas a última letra do texto (letra “i” nesse caso), em uma sequência máxima de 8 caracteres.



Para que o modelo conseguisse compreender as relações entre letras e palavras, precisamos aumentar a complexidade, melhorando o nosso Multi-Head Attention, juntamente com a adição de novas camadas, como a FeedForward, uma

Head que é utilizada na Multi-Head e a criação de uma Block que desempenhará o papel de criar vários blocos destacados na figura (bloco indicado pelo Nx).



Como dito anteriormente, o modelo enxerga apenas relações locais: dado um caractere, qual é o próximo? No entanto, isso gera textos sem sentido, porque o modelo não consegue capturar o contexto de longo prazo (por exemplo, formar palavras inteiras).

Para resolver isso, introduzimos o mecanismo de Transformers, que é justamente o que dá ao GPT sua força. A ideia é que cada letra "converse" com as demais, entendendo dependências de contexto.

Partimos então para compreender o que cada bloco está desempenhando no modelo:

- **Head**: representa uma única cabeça de “atenção”. Ela calcula a relação entre tokens por meio de vetores de query, key e value, permitindo que cada posição de uma sequência "preste atenção" em posições anteriores (respeitando a máscara tril, que impede olhar para o futuro).
- **MultiHeadAttention**: combina várias Heads em paralelo. Isso permite que o modelo aprenda diferentes tipos de relações ao mesmo tempo (por exemplo, uma cabeça pode focar em proximidade, outra em rima, outra em padrões gramaticais).
- **FeedForward**: após a etapa de atenção, os embeddings passam por uma rede densa não-linear, que transforma as representações aprendidas e aumenta a expressividade do modelo.
- **Block**: um bloco Transformer completo, formado por:

- Normalização (LayerNorm),
- Autoatenção (MultiHeadAttention),
- Camada densa (FeedForward),
- Conexões residuais (que estabilizam o treinamento).

- **BigramLanguageModel**: é o nosso modelo principal. Ele combina:
  - token_embedding_table: converte tokens em vetores contínuos,
  - positional_embedding_table: adiciona informação de posição (essencial, já que a atenção sozinha não entende ordem),
  - Vários blocos Block empilhados (n_layer),
  - Normalização final + camada linear (lm_head) que retorna as probabilidades para o próximo caractere.

A função generate permite criar texto novo: dado um token inicial (por exemplo, o índice de um espaço em branco), o modelo vai amostrando, caractere por caractere, até formar uma nova sequência.

No início, a geração de texto parece aleatória (exemplo: “Frutartut fe d pas i”), pois o modelo só olha a última letra. Porém, à medida que os blocos de atenção e feedforward são treinados, o modelo começa a capturar relações de contexto maiores: primeiro entre letras, depois entre palavras e, por fim, entre frases inteiras.

Referencias

Machine Learning, Data Science and Generative AI with Python (Seção 11 – Generative Models)

Let's build GPT: from scratch, in code, spelled out.