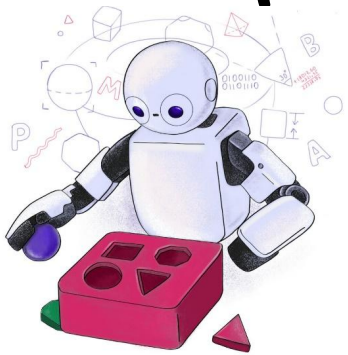
TP558 - Tópicos avançados em Machine Learning: ***Generative Adversarial Networks (GANs)***

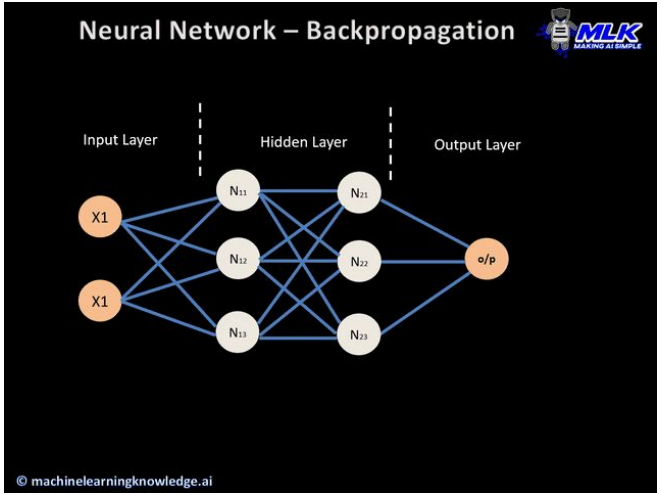


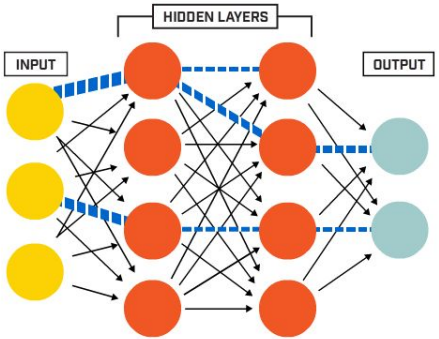
Inatel

Isabel Mendes
isabel.f@dtel.inatel.br

Contexto histórico - até de 2014

- Deep learning conseguia criar modelos capazes de entender e representar padrões complexos em diferentes tipos de dados, como imagens, fala ou linguagem escrita.
- Os maiores avanços ocorreram nos **modelos discriminativos** — aqueles que recebem uma grande quantidade de dados de entrada e os classificam em categorias.
- Esses progressos foram possíveis graças a técnicas como **backpropagation** e **dropout**, que funcionam muito bem com certos tipos de funções de ativação.





Contexto histórico - até de 2014

- Por outro lado, os **modelos generativos**, que têm como objetivo **criar novos dados** (e não apenas classificá-los), enfrentaram mais dificuldades, exigindo cálculos probabilísticos muito complicados, difíceis de resolver diretamente.
- Para contornar esse problema, os autores propuseram um novo método chamado **redes adversariais**. Nesse esquema, o **modelo gerado**r compete contra um adversário, o **modelo discriminador**. O gerador tenta criar exemplos falsos que pareçam reais, enquanto o discriminador tenta identificar o que é falso e o que é verdadeiro.

Ian J. Goodfellow, Jean Pouget-Abadi , Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, Yoshua Bengio

Universidade de Montreal

Trabalhos relacionados

- RBMs, DBMs, DBNs (máquinas de Boltzmann restritas ou profundas e deep belief networks) → **distribuições probabilísticas complexas** e intratáveis, exigindo métodos aproximados como o cadeias de **Markov ou Monte Carlo**, que sofrem com lentidão no processo de mistura, tornando o **treinamento difícil**.
- NCE (noise-contrastive estimation) → usa treinamento discriminativo para estimar modelos generativos, mas **desacelera consideravelmente** após o modelo aprender parcialmente a distribuição.
- GSNs (Redes generativas estocástica) → precisam de cadeias de Markov

Cadeia de Markov >> necessita do estado anterior para treinar a rede
Backpropagation >> atualizar os pesos da rede

Exemplo:

🎭 Os Personagens:

1. **O Falsificador (Gerador):** Ele quer criar *imitações perfeitas*.
2. **O Detetive (Discriminador):** Ele é muito esperto e quer descobrir se a obra é falsa ou verdadeira.

⚔️ O Jogo:

- O falsificador cria uma obra de arte.
- O detetive olha e tenta decidir: **"Isso é real ou falso?"**
- No começo, o falsificador é péssimo: desenha nuvens quadradas.O detetive dá risada e pega na hora.
- Mas o falsificador aprende com seus erros e melhora a cada tentativa.
- Enquanto isso, o detetive também fica cada vez melhor, pegando detalhes mais sutis.

🔄 A Competição Infinita:

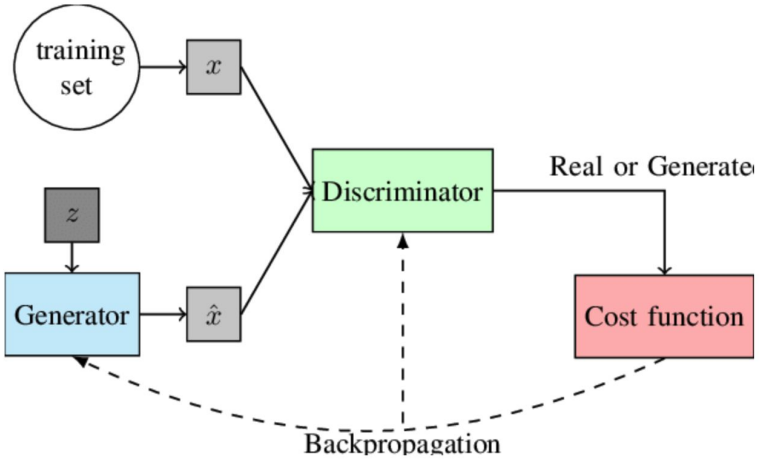
- O falsificador quer enganar o detetive. O detetive não quer ser enganado.
- Essa disputa continua até que o falsificador se torne **tão bom** que o detetive não consegue mais distinguir se algo é real ou gerado.

🎨 Resultado:

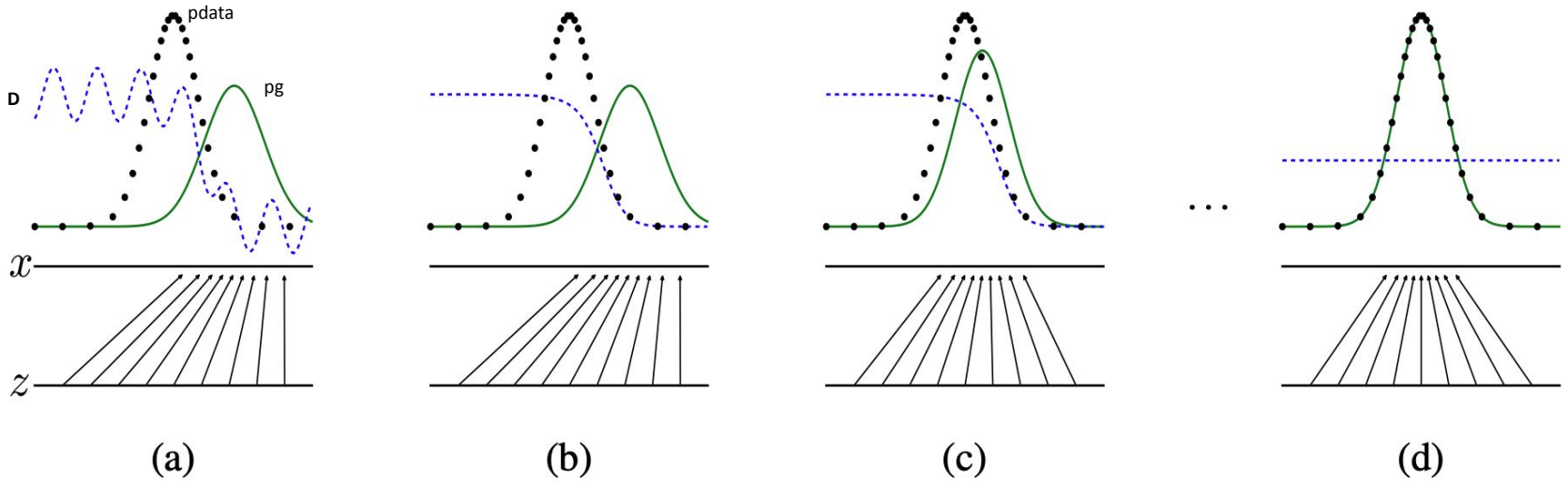
No final, o falsificador aprende a criar obras de arte que parecem **muito reais**, mesmo sem nunca ter visto o “original completo”.

Fundamentação teórica

- Modelos generativos usando processo adversarial
- Gerador (G): Gera dados a partir de uma pequena amostra
- Discriminador (D): Distingue entre dados reais e gerados
- Treinamento = jogo minimax ⟶ Equilíbrio: G imita dados reais,
- Acaba quando D = 0,5



Fundamentação teórica



(a) Considere um par adversarial próximo à convergência: pg é semelhante a pdata e D é um classificador parcialmente preciso.

(b) No loop interno do algoritmo, D é treinado para discriminar amostras de dados, convergindo para D(x) = pdata (x)/[pdata (x)+pg (x)].

(c) Após uma atualização de G, o gradiente de D guiou G(z) para fluir para regiões que têm maior probabilidade de serem classificadas como dados.

(d) Após várias etapas de treinamento, se G e D tiverem capacidade suficiente, eles chegarão a um ponto em que ambos não poderão melhorar.

Fundamentação teórica

Objetivo de D: maximizar a acurácia da predição

Prob. de D corretamente predizer x como um dado real

Prob. de D corretamente predizer G(Z) como um dado gerado

$$\min_G \max_D V(D,G) = \mathbb{E}_{\boldsymbol{x}\sim p_{\text{data}}(\boldsymbol{x})}[\log D(\boldsymbol{x})] + \mathbb{E}_{\boldsymbol{z}\sim p_{\boldsymbol{z}}(\boldsymbol{z})}[\log(1 - D(G(\boldsymbol{z})))].$$

Objetivo de G: minimizar a acurácia da predição, enganando D com suas saídas G(z) tão reais como possíveis.

1. Log é uma função monotônica crescente: X aumenta, f(x) aumenta ou fica constante.
2. Log(D(X)) é maximizado quando D(X)=1 e minimizado quando D(X)=0. >>> **maximizar globalmente a função de perda acima significa obter perda igual a 0.**
3. Log(1-D(G(Z))) é maximizado quando D(G(Z))=0 e minimizado quando D(G(Z))=1. >>> **minimizar globalmente a função de perda acima.**

Arquitetura

```
def build_generator():

    model = Sequential([

        Dense(32*32*256, input_dim=NOISE_DIM),
        LeakyReLU(alpha=0.2),
        Reshape((32,32,256)),

        Conv2DTranspose(128, (4, 4), strides=2, padding='same'),
        LeakyReLU(alpha=0.2),

        Conv2DTranspose(128, (4, 4), strides=2, padding='same'),
        LeakyReLU(alpha=0.2),

        Conv2D(CHANNELS, (4, 4), padding='same', activation='tanh')
    ],
    name="generator")
    model.summary()
    model.compile(loss="binary_crossentropy", optimizer=OPTIMIZER)

    return model
```

- Modelo Keras sequencial. Um modelo sequencial é uma pilha linear de camadas.
- Dense(32*32*256, input_dim=NOISE_DIM): camada densa (totalmente conectada) que recebe uma entrada de tamanho NOISE_DIM (que é 100, o tamanho do vetor de ruído aleatório) e gera um tensor de tamanho 32*32*256.
- LeakyReLU(alpha=0.2): função de ativação que permite um pequeno gradiente quando a entrada é negativa, o que pode ajudar a prevenir "ReLUs moribundas".
- Reshape((32,32,256)): remodela a saída da camada densa em um tensor 3D com dimensões 32x32 e 256 canais. Isso prepara os dados para serem processados por camadas convolucionais.

- Conv2DTranspose(128, (4, 4), strides=2, padding='same'): Esta é uma camada convolucional transposta (também conhecida como camada deconvolucional). É usada para aumentar a resolução da imagem. Ela possui 128 filtros de tamanho 4x4, um stride de 2 (que dobra as dimensões espaciais) e padding='same' (tamanho da saída igual da entrada).
- Conv2D(CHANNELS, (4, 4), padding='same', activation='tanh'): Esta é uma camada convolucional padrão com filtros CHANNELS (que são 3 para RGB) e uma função de ativação 'tanh' que é frequentemente usada na camada de saída de um gerador para gerar valores na faixa de -1 a 1, o que é comum para imagens quando normalizadas para essa faixa.

- loss="binary_crossentropy": Especifica a função de perda, adequada para tarefas de classificação binária (embora aqui seja usada para tentar fazer com que o gerador produza imagens que o discriminador classifique como reais).
- optimizer=OPTIMIZER: Especifica o otimizador a ser usado para treinamento, definido como um otimizador de Adam com uma taxa de aprendizado de 0,0002 e um beta1 de 0,5 na variável OPTIMIZER.

Arquitetura

```
[ ]  def build_discriminator():

       model = Sequential([

            Conv2D(64, (3, 3), padding='same', input_shape=(WIDTH, HEIGHT, CHANNELS)
            LeakyReLU(alpha=0.2),

            Conv2D(128, (3, 3), strides=2, padding='same'),
            LeakyReLU(alpha=0.2),

            Conv2D(128, (3, 3), strides=2, padding='same'),
            LeakyReLU(alpha=0.2),

            Conv2D(256, (3, 3), strides=2, padding='same'),
            LeakyReLU(alpha=0.2),

            Flatten(),
            Dropout(0.4),
            Dense(1, activation="sigmoid", input_shape=(WIDTH, HEIGHT, CHANNELS))
        ], name="discriminator")
        model.summary()
        model.compile(loss="binary_crossentropy",
                            optimizer=OPTIMIZER)

        return model
```

- Várias camadas Conv2D com números crescentes de filtros. Essas camadas aprendem características das imagens aplicando filtros convolucionais.
- Flatten(): Esta camada converte a saída 3D das camadas convolucionais em um vetor 1D, preparando-a para a camada densa final.
- Dropout(0,4): Esta camada define aleatoriamente uma fração das unidades de entrada como 0 a cada atualização durante o treinamento, o que ajuda a evitar overfitting.
- Dense(1, activation="sigmoid"): Esta é a camada de saída. Ela possui uma unidade e utiliza uma função de ativação sigmóide. A função sigmóide gera um valor entre 0 e 1, representando a probabilidade de a imagem de entrada ser real (mais próxima de 1) ou falsa (mais próxima de 0).

O treinamento

```
for epoch in range(10): # Changed range to 10 epochs
    # Display progress bar only for specified epochs
    if (epoch + 1) in [1, 5, 10]:
        print(f"Epoch {epoch + 1}/{10}") # Changed total epochs in print stateme
        batch_iterator = tqdm(range(STEPS_PER_EPOCH))
    else:
        batch_iterator = range(STEPS_PER_EPOCH)

    for batch in batch_iterator:
        noise = np.random.normal(0,1, size=(BATCH_SIZE, NOISE_DIM))
        fake_X = generator.predict(noise)

        idx = np.random.randint(0, X_train.shape[0], size=BATCH_SIZE)
        real_X = X_train[idx]

        X = np.concatenate((real_X, fake_X))

        disc_y = np.zeros(2*BATCH_SIZE)
        disc_y[:BATCH_SIZE] = 1

        d_loss = discriminator.train_on_batch(X, disc_y)

        y_gen = np.ones(BATCH_SIZE)
        g_loss = gan.train_on_batch(noise, y_gen)

    print(f"EPOCH: {epoch + 1} Generator Loss: {g_loss:.4f} Discriminator Loss:
    # Sample images only at specified epochs
    if (epoch + 1) in [1, 5, 10]:
        noise = np.random.normal(0, 1, size=(10,NOISE_DIM))
        sample_images(noise, (2,5))
```

- for epoch in range(10):: Este é o loop principal que itera pelas épocas de treinamento.
- for batch in batch_iterator:: Este loop itera pelo número definido de passos por época (STEPS_PER_EPOCH), processando efetivamente o número de imagens BATCH_SIZE em cada passo.
- noise = np.random.normal(0,1, size=(BATCH_SIZE, NOISE_DIM)): Isso **gera um lote de vetores de ruído aleatórios.** O ruído é amostrado a partir de uma distribuição normal com média de 0 e desvio padrão de 1. O tamanho do vetor de ruído é determinado por NOISE_DIM (100), e o número de vetores de ruído no lote é BATCH_SIZE (4).
- fake_X = generator.predict(noise): Esta linha usa o modelo do gerador para **criar imagens falsas** a partir do ruído gerado. O método predict gera as imagens geradas com base no ruído de entrada.
- idx = np.random.randint(0, X_train.shape[0], size=BATCH_SIZE): Seleciona um lote aleatório de índices dos dados de treinamento X_train.
- real_X = X_train[idx]: **Extrai o lote de imagens reais de X_train** usando os índices selecionados aleatoriamente.
- X = np.concatenate((real_X, fake_X)): **Concatena o lote de imagens reais e o lote de imagens falsas** em um único array X.
- disc_y = np.zeros(2*BATCH_SIZE): Cria um array disc_y de zeros com tamanho duas vezes maior que o BATCH_SIZE. **Este array armazenará os rótulos para treinar o discriminador**. Zeros normalmente representam "falso" na saída do discriminador.
- disc_y[:BATCH_SIZE] = 1: Define a primeira metade do array disc_y como uns. Esta parte corresponde às **imagens reais no array X concatenado, rotulando-as como "reais".**
- d_loss = **única atualização de gradiente em um lote de dados.** A variável d_loss armazena a perda do discriminador para este lote.
- y_gen = Esses são os rótulos alvo para o gerador. O gerador é treinado para enganar o discriminador, fazendo-o classificar as imagens falsas como "reais" (representadas por 1).
- g_loss = gan.train_on_batch(noise, y_gen): Isso treina o modelo gan combinado. Lembre-se de que, no modelo gan, os pesos do discriminador são congelados (discriminator.trainable = False). **Portanto, treinar o gan apenas atualiza os pesos do gerador.** O gan recebe o ruído como entrada e o alvo é y_gen (todos os valores 1), o que significa que ele tenta fazer com que a saída do discriminador seja 1 para as imagens geradas. A variável g_loss armazena a perda do gerador para este lote.

O treinamento

```
for epoch in range(10): # Changed range to 10 epochs
    # Display progress bar only for specified epochs
    if (epoch + 1) in [1, 5, 10]:
        print(f"Epoch {epoch + 1}/{10}") # Changed total epochs in print stateme
        batch_iterator = tqdm(range(STEPS_PER_EPOCH))
    else:
        batch_iterator = range(STEPS_PER_EPOCH)

    for batch in batch_iterator:
        noise = np.random.normal(0,1, size=(BATCH_SIZE, NOISE_DIM))
        fake_X = generator.predict(noise)

        idx = np.random.randint(0, X_train.shape[0], size=BATCH_SIZE)
        real_X = X_train[idx]

        X = np.concatenate((real_X, fake_X))

        disc_y = np.zeros(2*BATCH_SIZE)
        disc_y[:BATCH_SIZE] = 1

        d_loss = discriminator.train_on_batch(X, disc_y)

        y_gen = np.ones(BATCH_SIZE)
        g_loss = gan.train_on_batch(noise, y_gen)

    print(f"EPOCH: {epoch + 1} Generator Loss: {g_loss:.4f} Discriminator Loss:
    # Sample images only at specified epochs
    if (epoch + 1) in [1, 5, 10]:
        noise = np.random.normal(0, 1, size=(10,NOISE_DIM))
        sample_images(noise, (2,5))
```

Etapas do algoritmos de treinamento

1. Amostrar vetor de ruído z.
2. Amostrar dado real x.
3. Atualizar D.
4. Atualizar G: minimizar $\log(1-D(G(z)))$

O experimento

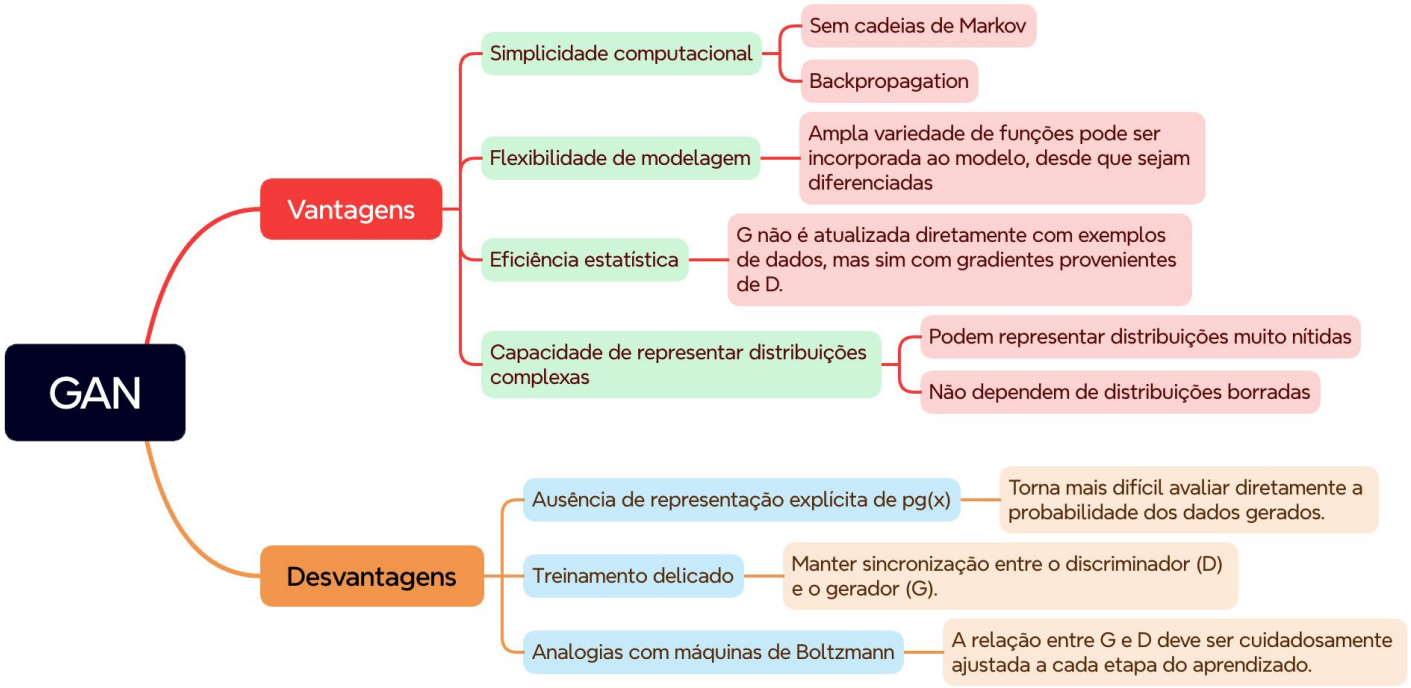
- Datasets: MNIST, TFD, CIFAR-10
- G: ReLU + Sigmoid,
- D: Maxout + Dropout

Imagens reais

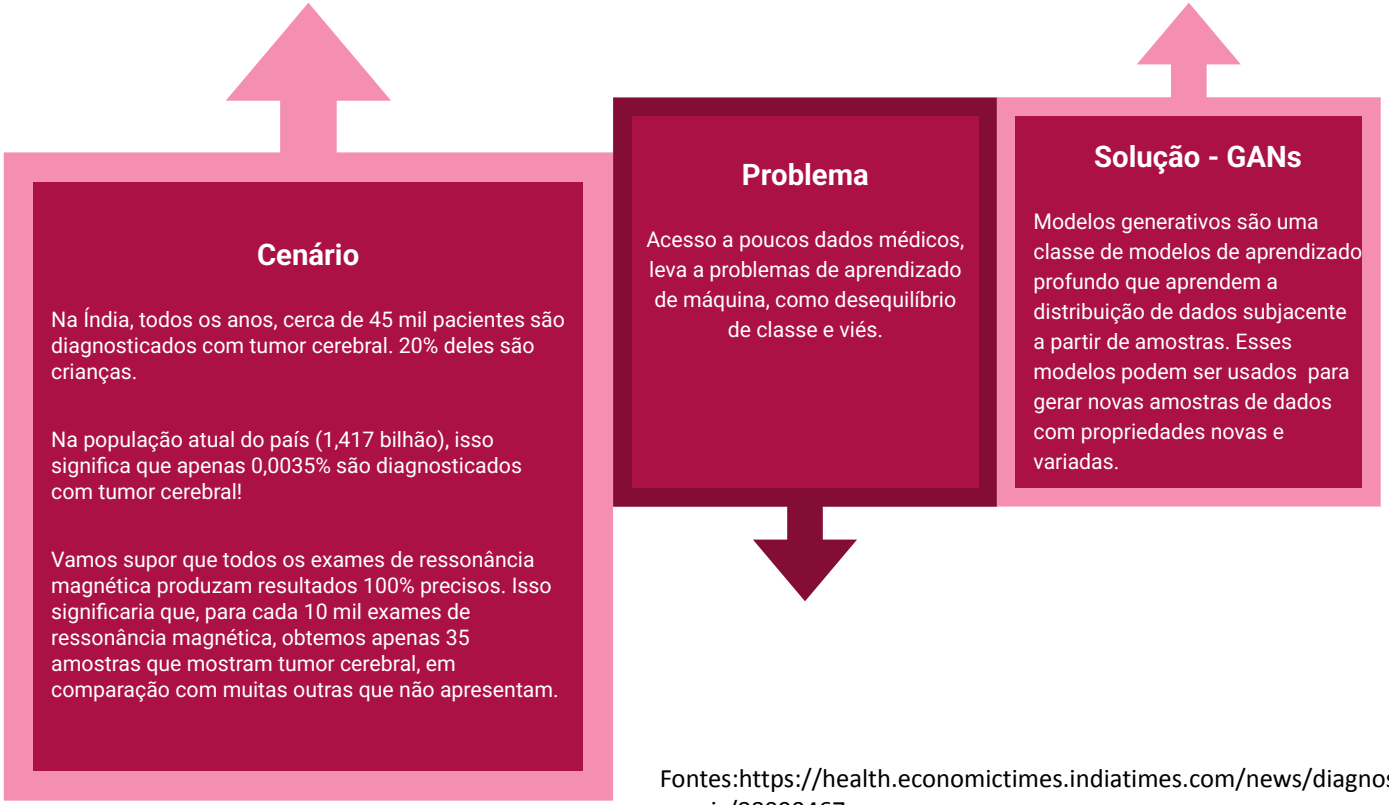
Comparações

	Modelos gráficos dirigidos profundos	Modelos gráficos não dirigidos profundos	Autoencoders generativos	Modelos adversariais
Treinamento	Inferência necessária durante o treinamento.	Inferência necessária durante o treinamento. MCMC necessário para aproximar o gradiente da função de partição.	Compromisso forçado entre mistura e poder de geração de reconstrução.	Sincronizar o discriminador com o gerador.
Inferência	Inferência aproximada aprendida.	Inferência variacional.	Inferência baseada em MCMC.	Inferência aproximada aprendida.
Amostragem	Sem dificuldades.	Requer cadeia de Markov.	Requer cadeia de Markov.	Sem dificuldades.
Avaliação de p(x)	Intratável, pode ser aproximado com AIS.	Intratável, pode ser aproximado com AIS.	Não explicitamente representado, pode ser aproximado com estimação de densidade de Parzen.	Não explicitamente representado, pode ser aproximado com estimação de densidade de Parzen.
Projeto do modelo	Quase todos os modelos apresentam extrema dificuldade.	Projeto cuidadoso necessário para garantir múltiplas propriedades.	Qualquer função diferenciável é teoricamente permitida.	Qualquer função diferenciável é teoricamente permitida.

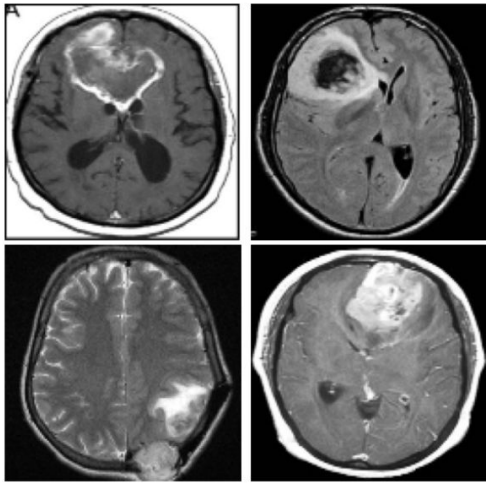
Vantagens & Desvantagens



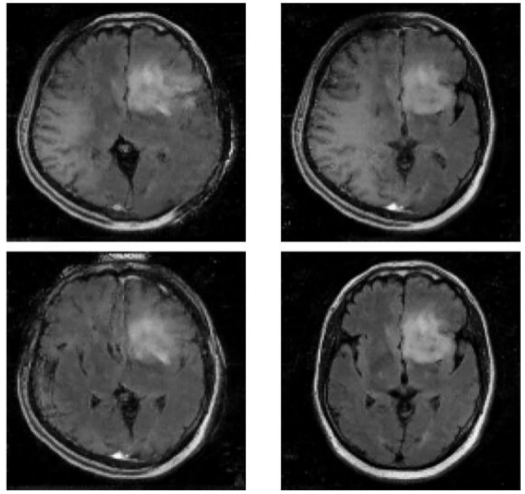
Aplicações - Generating Brain MRI Images with DC-GAN



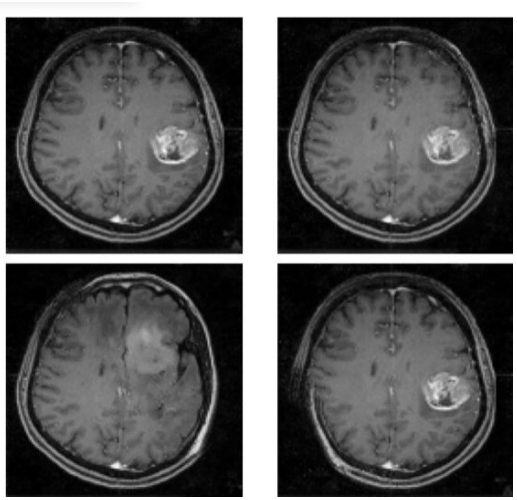
Fontes:https://health.economictimes.indiatimes.com/news/diagnostics/brain-tumors-death-on-diagnosis/88090467



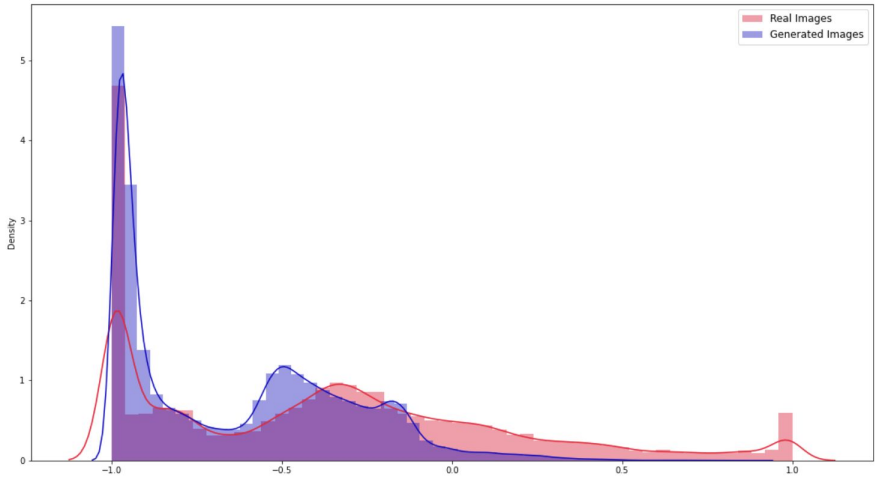
Amostras reais



5 ª época



10 ª época



Runtime

56m 9s · GPU P100

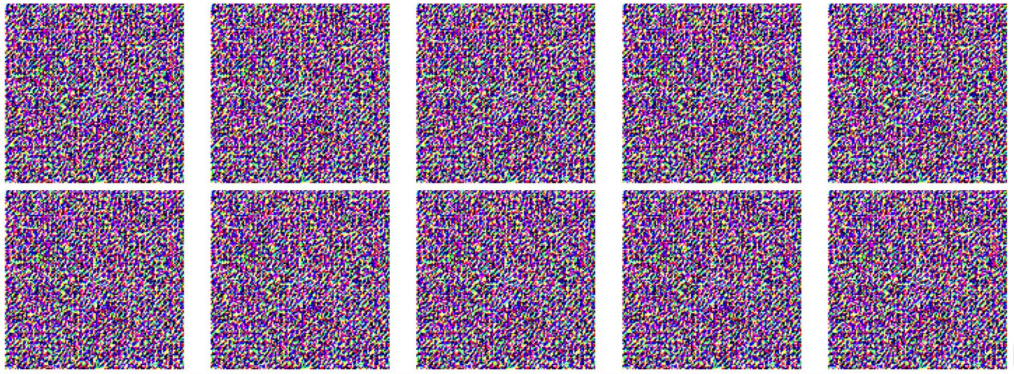
Input

DATASETS

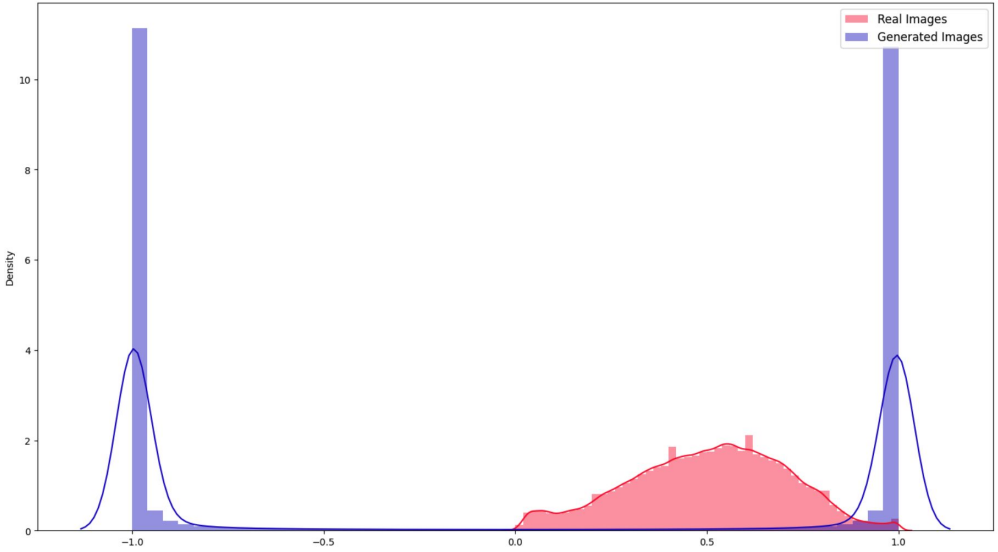
brain-mri-images-for-brain-tumor-de

Amostras reais

```
100%|██████████| 1000/1000 [03:03<00:00,  5.46it/s]
EPOCH: 10 Generator Loss: 0.0067 Discriminator Loss: 4.2091
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 64ms/step
```



10ª época



(GPU) de back-end do Google Compute Engine em Python 3
Mostrando os recursos de 13:08 a 14:00

RAM do sistema
3.3 / 12.7 GB

RAM da GPU
13.7 / 15.0 GB

Disco
39.3 / 112.6 GB

Evolução das GANs

Fig. 1.3

From: Overview of GAN Structure

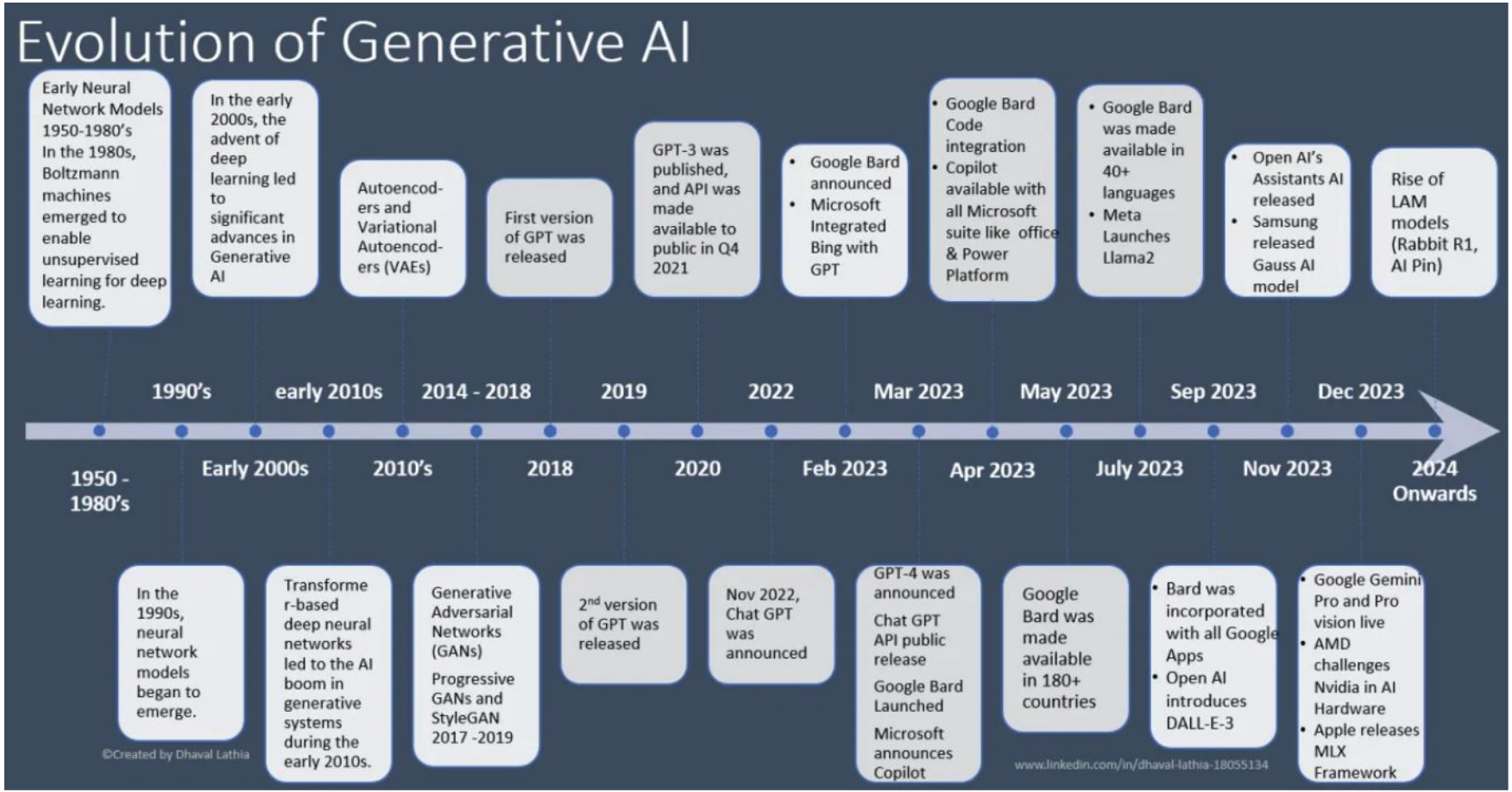
Development of GANs from 2014 to 2023

Fonte: https://doi.org/10.1007/978-3-031-32661-5_1

Tipos de GANs

01	Vanilla GAN	• Estrutura básica: gerador + discriminador • Aprendizado adversarial por competição
02	DCGAN (Deep Convolutional GAN)	• Usa camadas convolucionais • Melhor para imagens • Técnicas: *batch normalization*, *LeakyReLU*
03	cGAN (Conditional GAN)	• Usa informações extras (ex.: rótulos de classe) • Permite geração controlada (objetos/estilos específicos)
04	WGAN (Wasserstein GAN)	• Substitui a função de perda pela distância de Wasserstein • Mais estável, reduz *mode collapse*
05	WGAN-GP (com Gradient Penalty)	• Versão aprimorada da WGAN • Penaliza gradientes → atualizações mais suaves
06	InfoGAN	• Maximiza a informação mútua com variáveis latentes • Saídas mais interpretáveis e estruturadas
07	PGGAN (Progressive Growing GAN)	• Treinamento progressivo (baixa → alta resolução) • Usada para rostos humanos realistas
08	CycleGAN	• Tradução de imagens sem pares de treino • Ex.: fotos ↔ pinturas, estilos diferentes
09	StyleGAN	• Gerador baseado em estilos • Controle refinado de atributos da imagem • Muito usada em rostos hiper-realistas
10	BigGAN	• Modelos e datasets maiores • Imagens altamente detalhadas em larga escala

Fonte: https://ezinsights.ai/generative-adversarial-networks-ai/



Evolution Of Generative AI. source: https://www.linkedin.com/pulse/evolution-generative-ai-dhaval-lathia-cfnie/

Perguntas?

Referências

https://aiplanet.com/notebooks/1959/manish_kc_06/introduction-to-gans-using-fashion-mnist-dataset

https://medium.com/@ibtissam.essadik/the-history-and-evolution-of-generative-ai-from-gan-to-gpt-4-96c1fa69f4e7

https://www.kaggle.com/code/lynchrl/week-5-gan-mini-project-final-execution

https://www.kaggle.com/code/harshsingh2209/generating-brain-mri-images-with-dc-gan/notebook

Kaddoura, S. (2023). Overview of GAN Structure. In: A Primer on Generative Adversarial Networks . SpringerBriefs in Computer Science. Springer, Cham. https://doi.org/10.1007/978-3-031-32661-5_1

Obrigada!