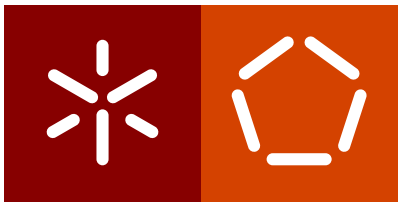
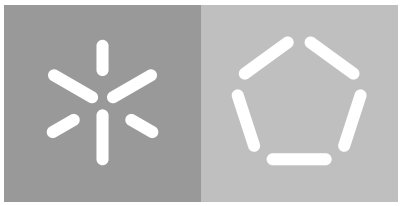


Universidade do Minho
Escola de Engenharia
Departamento de Informática

Tiago Rico Gonçalves

Geração de Imagens de Faces com Alta Resolução utilizando Variational Autoencoders

Outubro 2023



Universidade do Minho
Escola de Engenharia
Departamento de Informática

Tiago Rico Gonçalves

Geração de Imagens de Faces com Alta Resolução utilizando Variational Autoencoders

Dissertação de Mestrado
em Engenharia Informática

Dissertação supervisionada por
Professor António Joaquim André Esteves

Outubro 2023

DIREITOS DE AUTOR E CONDIÇÕES DE UTILIZAÇÃO DO TRABALHO POR TERCEIROS

AGRADECIMENTOS

Ao longo da realização desta dissertação recebi bastante apoio de forma direta e indireta, a que estou seriamente grato.

Ao meu orientador Professor António Joaquim André Esteves agradeço o seu empenho, a disponibilidade e todo o acompanhamento prestado no decorrer do processo de desenvolvimento da presente dissertação.

À minha família agradeço o apoio financeiro e o esforço em proporcionar as melhores condições possíveis à realização deste trabalho, assim como, durante toda o meu percurso académica.

De forma generalizada exprimo aqui um obrigado a todos aqueles que, de algum modo, fizeram parte do meu trajeto até aqui e contribuíram para o meu desenvolvimento académico e pessoal.

RESUMO

Atualmente, a Inteligência Artificial, e em especial a subárea da aprendizagem automática e profunda, é alvo de enorme interesse por parte das comunidades académica e empresarial. Das áreas aplicacionais da aprendizagem automática, a par com o processamento de língua natural, a visão por computador é aquela que tem suscitado mais interesse e gerado mais resultados científicos. A geração de imagens é um dos problemas que se enquadra na visão por computador.

Com esta dissertação pretende-se estudar os modelos geradores de imagens, e de entre as alternativas para atacar este problema, o foco do trabalho são os modelos *Variational Autoencoders*. Na fase inicial da dissertação é feito um levantamento bibliográfico do estado da arte do tema do trabalho, visando adquirir os conhecimentos necessários para concretizar a parte experimental.

Os conhecimentos adquiridos na fase de levantamento bibliográfico foram aplicados na fase seguinte, onde se desenvolveu, treinou e avaliou modelos capazes de gerar imagens novas com faces humanas. O foco foi a arquitetura *Vector Quantized Variational Autoencoder* (VQ-VAE), auxiliada por um modelo autorregressivo PixelCNN. No entanto, foram também explorados outros modelos geradores, tendo em mente complementar o estudo em causa, e consequentemente, poder tirar conclusões mais abrangentes.

Após a implementação dos modelos, foi possível concluir que dentro de todos os modelos testados o VQ-VAE apresentou o melhor desempenho, quer seja a nível qualitativo através da inspeção visual das faces geradas, quer seja a nível quantitativo com a aplicação da métrica *Frechet Inception Distance*. Além do VQ-VAE, o outro modelo que se destacou foi o *Vector Quantized Generative Adversarial Network*, comprovando assim o potencial da aplicação da quantização de vetores nos modelos geradores.

Palavras-chave: modelo gerador de imagens, *variational autoencoder*, VQ-VAE, PixelCNN, VQ-GAN.

ABSTRACT

Nowadays, the artificial intelligence, particularly the subarea of machine and deep learning, is the subject of enormous interest from the academy and the companies. Among the application areas of machine learning, along with natural language processing, computer vision is the one that has aroused the most interest and generated the most scientific results. Image generation is one of the problems that fall within computer vision.

With this dissertation, it is intended to study image generation models, and among the alternatives to address this problem, the main focus of the present work are the *Variational Autoencoders* models. In the initial phase of the dissertation, a bibliographic review of the state of the art of the subject of the work was carried out, aiming to acquire the necessary knowledge to carry out the experimental phase.

The knowledge acquired with the bibliographic review was applied in the next phase, where models capable of generating new images of human faces were developed, trained and evaluated. The main focus was the *Vector Quantized Variational Autoencoder* (VQ-VAE) architecture, aided by an autoregressive PixelCNN model. However, other types of generative models were also explored, in order to complement the study at hand and consequently reach more comprehensive conclusions.

After implementing the models, it was possible to conclude that among all the evaluated models, VQ-VAE was the one that presented the best performance, both qualitatively, visually inspecting the generated faces, and quantitatively, applying the *Frechet Inception Distance* metric. In addition to VQ-VAE, the other model to stand out was VQ-GAN, thus justifying the potential of applying the concept of vector quantization to generative models.

Keywords: image generation model, variational autoencoder, VQ-VAE, PixelCNN, VQ-GAN.

CONTEÚDO

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE LISTAGENS

SIGLAS

CNN Convolutional Neural Network. 1

DCT Discrete Cosine Transform. 1, 32, 33

FFHQ Flickr-Faces-HQ. 1, 38
FID Frechet Inception Distance. 1, 9, 14–16, 103–106, 108

GAN Generative Adversarial Network. 1, 7, 8, 10, 14–17, 26, 27, 30–34, 63–65, 68, 69, 72, 74, 79, 82, 97, 99, 104, 105, 107
GPT Generative Pre-trained Transformer. 1, 68, 74

IntroVAE Introspective Variational Autoencoder. 1, 7, 11, 16, 31, 32, 79, 80, 82, 84, 107
IS Inception Score. 1, 104

JPEG Joint Photographic Experts Group. 1, 33

LSTM Long Short-Term Memory. 1

MNIST Modified National Institute of Standards and Technology database. 1, 37, 38, 86

NIST National Institute of Standards Technology. 1, 37

PGGAN Progressive Growing of Generative Adversarial Network. 1, 32
PixelCNN Pixel Convolutional Neural Network. 1, 7, 14, 24–26, 28, 32, 33, 41, 47, 48, 52–54, 57, 58, 62, 68, 74, 92, 104
PixelRNN Pixel Recurrent Neural Network. 1, 32

RNN Recurrent Neural Network. 1, 32

VAE Variational Autoencoder. 1, 7, 13–15, 17, 18, 20–23, 31, 32, 41–47, 50, 63, 65, 79, 80, 82, 86–91, 104, 108, 109
VQ-GAN Vector Quantized Generative Adversarial Network. 1, 7, 8, 10, 11, 14, 16, 68–72, 74, 75, 100–102, 104, 106
VQ-VAE Vector Quantized Variational Autoencoder. 1, 7, 8, 11, 14, 15, 17, 23–25, 31–34, 41, 46–48, 50, 51, 55, 57–59, 62, 68, 69, 72, 74, 75, 78, 89, 91–95, 100, 102, 105, 106
VQ-VAE2 Vector Quantized Variational Autoencoder 2. 1, 8, 10, 15, 31, 33, 34, 55, 57, 59, 60, 62, 94–98, 104

W&B Weights and Biases. 1, 36, 105

1

INTRODUÇÃO

Neste capítulo apresenta-se a contextualização e motivação do tema da dissertação, os objetivos que se pretendem atingir ao longo do trabalho, os resultados que se espera obter, e ainda, os métodos e técnicas a explorar durante o processo de desenvolvimento do trabalho.

1.1 CONTEXTO E MOTIVAÇÃO

Esta dissertação enquadra-se na unidade curricular final do segundo ano, do Mestrado em Engenharia Informática, da Escola de Engenharia da Universidade do Minho. O seu principal objetivo é explorar redes neuronais capazes de gerar novas imagens de faces humanas com alta resolução, ou seja, faces nunca antes vistas pelo modelo, sendo de prever que algumas faces geradas sejam ligeiras variações de imagens reais.

Num mundo tecnológico onde a Inteligência Artificial tem cada vez mais reconhecimento e relevância, o desenvolvimento de um trabalho nesta área é pertinente e motivador. A utilização de modelos geradores para criar novas imagens tem contribuído para o aumento do interesse global sobre este tipo de redes neuronais. Os modelos geradores são bastante importantes porque permitem explorar a capacidade das redes neuronais, produzindo resultados visualmente fascinantes, através da manipulação e geração de imagens, tornando-se muito apelativo não só para a comunidade científica, mas também para o público em geral. Os contributos deste tipo de modelos para fins criativos é algo cujo impacto já se vai podendo constatar, chamando ainda mais atenção ao tópico. Em suma, sendo esta uma área de estudo em crescimento, não só em popularidade, como também no desenvolvimento de tecnologias, é de realçar a importância de todo e qualquer envolvimento para a evolução da mesma, com a contribuição de novas ideias e perspetivas, e em simultâneo aprofundar e alargar conhecimentos.

No caso particular desta dissertação, o foco principal é os modelos geradores do tipo Variational Autoencoder (VAE) e as suas variantes. Estes modelos são menos explorados que outros tipos de modelos geradores, nomeadamente os modelos do tipo Rede Adversária Geradora (GAN) e os modelos de difusão. Como tal, a investigação dos modelos VAE, bem como das suas variantes, representa uma motivação acrescida para este trabalho.

A implementação das redes neuronais será feita com a linguagem `Python`, com recurso à biblioteca `TensorFlow` e, sempre que possível, utilizando a API `Keras`, explorando as potencialidades do ambiente computacional interativo `Jupyter notebook`, que combina execução de código, comentários, fórmulas matemáticas e gráficos. Este ambiente é atualmente dos mais utilizados por empresas e investigadores na área de Inteligência Artificial.

1.2 OBJETIVOS DA DISSERTAÇÃO

Este trabalho de dissertação tem como objetivo principal estudar e implementar modelos geradores de imagens, em particular para gerar imagens de faces humanas. Para esse efeito, será necessário atingir outros objetivos complementares, nomeadamente:

- Estudar a evolução dos *autoencoders*, passando pelos VAEs, até aos VAEs de vetores quantizados (VQ-VAE) que utilizam modelos autorregressivos PixelCNN;
- Analisar outros tipos de modelos geradores, tais como Generative Adversarial Network (GAN), modelos de fluxo e modelos de difusão;
- Efetuar o levantamento dos trabalhos relacionados, para identificar e compreender as diferentes arquiteturas propostas para os modelos geradores;
- Explorar e analisar o conjunto de dados de faces humanas escolhido para treinar os modelos a desenvolver;
- Modelar e treinar os modelos principais VAE, Vector Quantized Variational Autoencoder (VQ-VAE) e VQ-VAE2;
- Modelar e treinar um modelo GAN e um modelo Vector Quantized Generative Adversarial Network (VQ-GAN), para complementar a avaliação dos diferentes tipos de modelos geradores;
- Comparar os VAEs com outros modelos geradores implementados nomeadamente o modelo GAN e o modelo VQ-GAN;
- Modelar e treinar um modelo *Introspective* VAE, ou seja, um modelo híbrido que inclua uma rede VAE e uma rede GAN;
- Aplicar a métrica Frechet Inception Distance (FID) aos modelos implementados
- Analisar os resultados obtidos.

1.3 RESULTADOS ESPERADOS

Com base no levantamento bibliográfico sobre modelos geradores, e na implementação de alguns desses modelos, espera obter-se um maior conhecimento teórico e prático acerca dos mesmos. Posteriormente, pretende-se tirar conclusões sobre as potencialidades dos vários modelos implementados, com maior incidência nos modelos que são variantes do VAE.

1.4 MÉTODOS E TÉCNICAS

A inicial fase do trabalho de dissertação consistirá numa leitura e análise da bibliografia indicada na proposta de dissertação. De seguida, é feita a pesquisa, leitura e análise de bibliografia adicional. Este estudo bibliográfico é materializado na escrita do estado da arte.

A fase seguinte do trabalho consistirá em pesquisar um ou mais conjuntos de dados para treinar os modelos a desenvolver. Posteriormente, é feita uma análise mais profunda (EDA) ao conjunto de dados escolhido, de modo a tratar e compreender os dados a utilizar. Estando o conjunto de dados escolhido e pronto a usar no treino dos modelos, inicia-se o processo de implementação dos modelos definidos, em linguagem `Python`, recorrendo à biblioteca `TensorFlow` e à API `Keras`. Após implementar os modelos, serão efetuadas as comparações dos modelos, de acordo com a principal métrica com que se costumam avaliar os modelos geradores, a FID.

O trabalho a desenvolver terá o acompanhamento do Orientador, através de uma reunião semanal, com o objetivo de garantir o cumprimento do planeamento definido e manter uma revisão e aconselhamento constantes ao longo do trabalho.

1.5 ORGANIZAÇÃO DO DOCUMENTO

O presente documento encontra-se dividido em 5 capítulos. Sendo este o capítulo de introdução.

O capítulo 2 documenta o levantamento efetuado sobre o estado da arte dos modelos geradores de imagens. O capítulo inclui ainda a explicação dos vários componentes que fazem parte do principal modelo a estudar: o VQ-VAE. Nesta secção aborda-se também os modelos *autoencoder* e VAE. Adicionalmente, contém um resumo sobre os principais modelos geradores de imagens alternativos aos VAEs: GANs, modelos de difusão e modelos de fluxo.

O capítulo 3 documenta a implementação dos modelos. Primeiramente, introduzem-se as tecnologias usadas, apresentando também os conjuntos de dados usados, incluindo uma breve explicação do processo de tratamento desses mesmos dados. O grosso do capítulo é dedicado à implementação dos modelos geradores VAE, VQ-VAE, Vector Quantized Variati-

onal Autoencoder 2 (VQ-VAE2), GAN, VQ-GAN e Introspective Variational Autoencoder (IntroVAE).

O capítulo 4 contém os resultados obtidos com os modelos implementados e inclui também a análise do desempenho dos vários modelos. Inicialmente é feita uma avaliação qualitativa, onde são apresentadas e avaliadas as imagens de faces humanas que cada um dos modelos é capaz de gerar. Depois efetua-se uma análise quantitativa dos modelos, comparando todos os modelos implementados quanto ao tamanho, ao tempo de treino e ao tempo de inferência. Por fim, a análise quantitativa termina com a aplicação da métrica FID aos modelos desenvolvidos, acompanhada com uma avaliação global dos modelos tendo em conta todos os resultados introduzidos ao longo do capítulo.

O capítulo 5, é a conclusão, e nele é feita uma reflexão sobre o trabalho realizado e são discutidos os objetivos concretizados. Este capítulo menciona ainda algumas dificuldades encontradas ao longo do trabalho e elenca ideias para eventual trabalho futuro. Termina com uma pequena apreciação final do trabalho realizado.

2

ESTADO DA ARTE

O principal objetivo deste capítulo é apresentar modelos e conceitos relacionados com a área de Inteligência Artificial, para efeitos de geração de imagens, com especial foco nos modelos que são variantes do *Variational Autoencoder*. Com a análise do estado da arte pretende-se contextualizar o trabalho a desenvolver ao longo da dissertação.

Este capítulo encontra-se organizado nas seções 2.1 a 2.7. A primeira seção (2.1) apresenta uma explicação do modelo *Autoencoder*, uma vez que ele é a base dos modelos VAE e VQ-VAE, que por sua vez são analisados nas seções 2.2 e 2.3, respetivamente.

Após a apresentação dos modelos fulcrais deste trabalho, é alargado o alvo de estudo a três outros tipos de modelo. Deste modo, as seções 2.4, 2.5 e 2.6 correspondem aos modelos GAN, aos modelos de fluxo e aos modelos de difusão, respetivamente. Por fim, a seção 2.7 apresenta um resumo de trabalhos relacionados com a presente dissertação, cobrindo diferentes abordagens aos modelos geradores.

2.1 AUTOENCODER

Um *autoencoder* consiste em duas partes, um codificador e um descodificador. O codificador recebe como entrada uma imagem, um sinal áudio, ou texto codificado, e reduz essa entrada a uma representação latente das suas características. O vetor que contém essa representação latente, que na arquitetura está posicionado entre o codificador e o descodificador, chama-se *bottleneck*. A partir desta codificação, o descodificador tem como função reconstruir a entrada original, com o objetivo de ser o mais fiel possível (figura 1).

O *bottleneck* pode ter qualquer tamanho, sendo uma questão de aproveitamento do método que se está a aplicar. Com a implementação de um *autoencoder* pretende-se que o descodificador seja capaz de fazer reconstruções, com base na menor quantidade de informação possível. Assim sendo, quanto maior for o tamanho do *bottleneck*, menor será o aproveitamento, mas melhor será a saída da reconstrução, uma vez que se fornece mais informação ao descodificador e é assim mais fácil efetuar uma reconstrução fiel ao original. Quanto menor for o tamanho do *bottleneck*, maior será o aproveitamento do método, mas pior será o resultado da reconstrução.

No caso de estudo de imagens com faces humanas, o codificador reduz a imagem a uma representação latente das suas características. De seguida, o pequeno vetor contendo essa informação da imagem é usado como entrada do descodificador, cuja tarefa é recriar a imagem original.

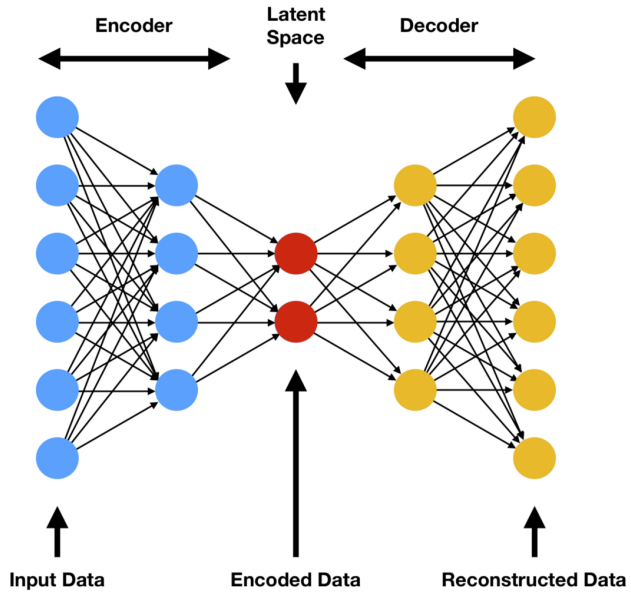


Figura 1: Arquitetura do modelo *autoencoder*.

O principal objetivo deste tipo de modelos é aprender codificações eficientes, revelando uma boa capacidade para funcionar como um método de compressão da informação. Uma possível utilidade destes modelos é a redução da necessidade de utilização de banda larga nas comunicações de dados. Por exemplo, caso um utilizador queira descarregar uma imagem, do lado do servidor recorre-se a um modelo *autoencoder* pré-treinado, utilizando-se o codificador para comprimir a imagem e enviando apenas o vetor com a codificação da imagem. Do lado do utilizador, depois de receber o vetor a informação é fornecida a um descodificador que recria a imagem pretendida. Desta forma, em vez de passar todo o conteúdo da imagem pela rede, transporta-se somente um pequeno vetor, a partir do qual será possível reconstruir a imagem original no lado do recetor. Contudo, até ao momento a maior contribuição dos *autoencoders* é na área de estudo desta dissertação, ou seja nos modelos geradores, servindo de base a outros modelos, como por exemplo o VAE.

2.1.1 *Codificador*

O codificador é uma rede neuronal composta por várias camadas, sendo maioritariamente de dois tipos: completamente ligadas e convolucionais. Esta rede neuronal tem como objetivo reduzir a entrada a uma representação latente das suas características. Em termos estruturais consiste num número opcional de camadas, com um mínimo de duas, onde a primeira terá o número de unidades necessárias para representar cada elemento da entrada, que no caso de uma imagem correspondem aos seus pixeis. A última camada terá um número de unidades igual à dimensão do vetor em que se pretende reduzir a imagem. O número de camadas adicionais entre os pontos de partida e chegada é uma escolha da implementação.

2.1.2 *Descodificador*

O descodificador, tal como o codificador, é uma rede neuronal composta por várias camadas, sendo maioritariamente de dois tipos: completamente ligadas e convolucionais. O objetivo desta rede neuronal é reconstruir a entrada original que o codificador transformou numa representação latente das suas características. A sua estrutura é simétrica da do codificador, sendo por isso composta por um número opcional de camadas, com um mínimo de duas. A primeira camada terá o mesmo número de unidades que a dimensão do vetor latente de entrada. A última camada terá o número de unidades necessárias para representar cada elemento da entrada original do codificador. Caso a entrada seja uma imagem será, o número de unidades corresponde ao número de pixeis.

2.1.3 *Denoising Autoencoder*

Um *denoising autoencoder* é uma variante de *autoencoder* criada com o objetivo de combater o risco de sobreajuste[1]. Este modelo altera o *autoencoder* simples introduzindo uma quantidade de ruído nos dados. Sucintamente, os dados de entrada são parcialmente corrompidos e o modelo deve continuar a ser capaz de reconstruir a entrada original não corrompida. Por conseguinte, o modelo é forçado a aprender representações latentes mais robustas, prevenindo assim, um possível sobreajuste aos dados de treino.

Um modelo completamente treinado de *denoising autoencoder* consegue reconstruir a informação original a partir de informação corrompida. No caso de imagens, isto traduz-se numa ferramenta útil para restaurar imagens danificadas ou com ruído.

A modificação introduzida pelo *denoising autoencoder* assemelha-se bastante ao uso de *dropout* em redes neuronais. Uma vez que o *dropout* só foi proposto à comunidade académica

1 *Overfitting*, na terminologia inglesa.

quatro anos após o surgimento dos *denoising autoencoders*, pode-se considerar que os *denoising autoencoders* são uma das inspirações por trás do *dropout*.

2.1.4 *Autoencoder Esparso*

Um *autoencoder* esparso modifica o *autoencoder* original aplicando uma restrição de esparsidade à ativação das unidades das camadas que compõem o modelo. Tendo novamente em vista a prevenção de sobreajuste no modelo, limita-se o número de unidades que estão ativas em cada momento. Essencialmente, este tipo de *autoencoder* acrescenta mais um termo à função de perda. Este termo penaliza o modelo em treino quando a quantidade de unidades simultaneamente ativas está longe do desejado. Um modelo *autoencoder* esparso consegue uma melhor resposta às características únicas dos dados de treino. A redução do número de unidades utilizadas em simultâneo por cada camada, facilita o foco da aprendizagem do modelo nos aspetos mais relevantes dos dados.

Uma variante de *autoencoder* esparso é o *k-sparse autoencoder*, que escolhe apenas as k unidades que possuem os maiores valores de ativação. O modelo percorre o codificador para encontrar essas k unidades neuronais mais importantes, mantendo apenas aquelas com os valores mais altos e definindo todas as outras com saída 0, o que na prática corresponde a desativá-las.

2.1.5 *Contractive Autoencoder*

Um *contractive autoencoder* propõe igualmente aumentar a restrição aplicada ao modelo original, a fim de manter a representação latente num espaço mais reduzido. O principal objetivo desta variante também é prevenir o sobreajuste e melhorar a robustez do modelo. Resumidamente, este modelo adiciona um novo termo à função de perda, que penaliza as reações demasiado sensíveis a pequenas variações dos dados de treino. O resultado é um modelo com uma representação latente mais genérica, mas que responde melhor a dados de entrada novos.

2.2 VARIATIONAL AUTOENCODER

O *variational autoencoder* (VAE) é uma evolução do *autoencoder* simples apresentada pelos autores Kingma and Welling (2014). Este modelo adota a mesma estrutura codificador-descodificador, alterando apenas a fase intermédia entre os dois blocos principais. Em vez de um simples vetor, entre o codificador e o descodificador o VAE utiliza funções de distribuição de probabilidade para representar o espaço latente, das quais se retirá uma amostra para reconstruir uma dada entrada (figura 2).

O propósito de um VAE é aprender codificações eficientes e utilizá-las para gerar informação nova, enquanto que o *autoencoder* simples se limita a reconstruir a informação de entrada. Tendo este objetivo em mente, o treino é feito pelo descodificador em conjunto com o codificador, através de retro-propagação. O processo de geração inicia com a criação de informação de entrada, resultante da amostragem das distribuições de probabilidade treinadas, a partir da qual o descodificador deve ser capaz de obter uma saída nova do mesmo género dos dados de treino (imagem, áudio, ou texto.), ou seja, deve ser capaz de gerar dados novos.

No presente trabalho os dados utilizados para treinar os modelos VAE são imagens com faces humanas. O codificador e o descodificador são treinados tal como num *autoencoder* simples, tentando reconstruir os exemplos de treino o melhor possível. Finalizado o treino, temos um descodificador capaz de criar imagens de faces humanas a partir dum vetor de entrada selecionado. Devido à diferença estrutural do VAE em relação ao *autoencoder* básico, torna-se possível restringir as entradas fornecidas ao descodificador. A partir de amostras obtidas das distribuições de probabilidade treinadas, espera-se que os valores de entrada do descodificador resultem na construção de uma imagem de uma face humana. Assim, é possível gerar novas imagens de faces humanas recorrendo a amostras das distribuições de probabilidade.

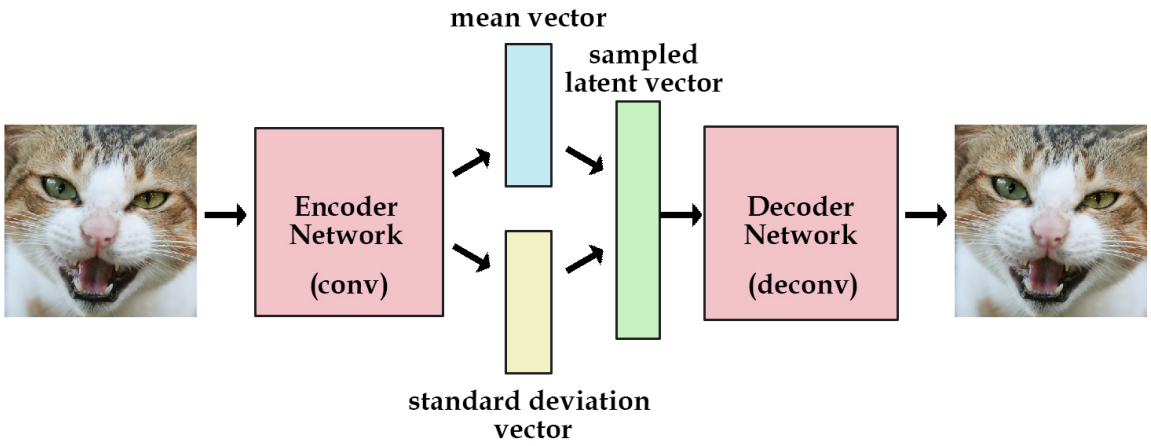


Figura 2: Arquitetura do modelo VAE.

2.2.1 *Distribuição e Amostragem*

O *autoencoder* não pode ser considerado um modelo gerador, em virtude de não oferecer garantia de gerar uma nova imagem com uma face humana, uma vez que não tem forma

de restringir a escolha do vetor de entrada no descodificador. De modo, ao transformar um *autoencoder* simples num modelo gerador é fundamental que haja a possibilidade de escolher o valor que será aplicado na entrada do descodificador, sabendo de antemão, que este valor resultará na geração de uma nova saída do género pretendido. Para tal, no VAE introduziram-se as funções de distribuição de probabilidade de onde se podem obter amostras, para substituir o *bottleneck*. Ao treinar as funções de distribuição de probabilidade, ou abreviadamente distribuições, em conjunto com o resto do modelo, garante-se que qualquer vetor obtido por amostragem dessas distribuições treinadas resultará numa imagem com uma face humana.

De acordo com os autores Kingma and Welling (2014), a função de perda utilizada para treinar o VAE é dada pela equação 1.

$$\mathcal{L}(\theta, \phi) = \mathbb{E}_{q_\phi(z|x)}[\log p_\theta(x|z)] - D_{KL}(q_\phi(z|x)||p_\theta(z)) \tag{1}$$

O primeiro termo representa o erro de reconstrução, que indica o quão longe os dados reconstruídos estão dos dados originais. Este termo é o que permite ao modelo aprender a reconstruir os dados originais, garantindo a capacidade de gerar o tipo de dados pretendidos. O segundo termo é uma divergência de Kullback-Leibler, que essencialmente calcula a distância entre duas funções de distribuição de probabilidade. Este termo funciona como um regulador com o qual se pretende assegurar que a distribuição treinada se mantém próxima de uma distribuição normal. Este termo é fundamental para impedir que o modelo aprenda distribuições estreitas, ou seja com pouca ou nenhuma variância, o que essencialmente tornaria o modelo num *autoencoder* simples, dado que a amostragem da distribuição devolveria quase sempre o mesmo valor.

2.2.2 *Otimização*

A aprendizagem do modelo acontece com a otimização da função de perda. Sendo $q_\phi(z|x)$ a saída do codificador e $p_\theta(x|z)$ a saída do descodificador, o objetivo da otimização do modelo é encontrar os valores ótimos para os parâmetros que influenciam estas duas saídas. Neste caso, os parâmetros do codificador e do descodificador estão representados por ϕ e θ, respetivamente.

O processo de otimização recorre ao método de retro-propagação para encontrar os valores ótimos dos parâmetros. Primeiramente, são definidos dois vetores que caraterizam as funções de distribuição de probabilidade, μ e σ, os quais dependem dos parâmetros θ e, por isso, são treinados em conjunto com o resto do modelo. Estes vetores representam a média (μ) e desvio padrão (σ) das funções de distribuição das variáveis latentes. No entanto, existe um problema na otimização do primeiro termo da função de perda. Aplicar o gradiente a um valor esperado, ou médio, numa amostragem aleatória é problemático.

Deste modo, a amostragem aleatória bloqueia a retro-propagação dos gradientes. A solução encontrada para contornar este problema é chamada de **truque de reparametrização** e consiste em alterar a forma como se faz a amostragem. Em vez de se obter uma amostra z de uma distribuição $\mathcal{N}(\mu, \sigma)$, obtém-se uma amostra ϵ a partir duma distribuição normal $\mathcal{N}(0, I)$. A amostra ϵ assim obtida é depois escalada do valor da variância σ e deslocada do valor da média μ, de acordo com a fórmula apresentada na equação 2, resultando na amostra z.

$$z = \mu + \sigma \odot \epsilon, \quad \text{onde } \epsilon \sim \mathcal{N}(0, I) \tag{2}$$

A aplicação desta técnica permite a retro-propagação atravessar esta fase de amostragem das funções de distribuição, ao tratar da parte estocástica do processo fora do caminho do fluxo da retro-propagação. A figura 3 apresenta as diferenças entre a forma original e a forma reparametrizada. Na forma original, a amostragem das funções de distribuição é feita diretamente sobre um nó estocástico, sendo que esse fator aleatório não permite o cálculo da derivada do gradiente, consequentemente, bloqueando o fluxo de retro-propagação. Na forma reparametrizada, a amostragem é feita sobre uma distribuição normal e, posteriormente, escalada e deslocada de acordo com a variância e média correspondentes. Esta alteração move o fator estocástico da amostragem, para o nó da distribuição normal auxiliar, criando um caminho de nós determinísticos, nos quais é possível calcular as derivadas dos gradientes e, assim, fluir a retro-propagação entre o descodificador e o codificador.

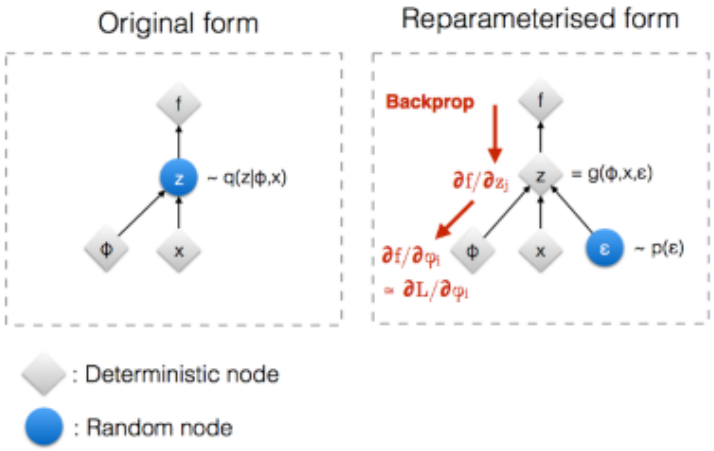


Figura 3: Truque de reparametrização no VAE.

2.3 VECTOR-QUANTIZED VARIATIONAL AUTOENCODER

O modelo *Vector Quantized Variational Autoencoder* (VQ-VAE), introduzido por van den Oord et al. (2018), é uma evolução relativamente recente do VAE, onde é novamente alterada a

ligação entre o codificador e o descodificador. Em vez de se usar funções de distribuição de probabilidade para obter a amostra a aplicar no descodificador, recorre-se a um novo método. No VQ-VAE discretiza-se o vetor de saída do codificador recorrendo a um *codebook*, permitindo uma maior restrição e controlo sobre o que o descodificador vai gerar. Após otimizar o codificador, o descodificador e o *codebook*, a geração de imagens utiliza apenas o descodificador, auxiliado por um modelo autoregressivo PixelCNN.

2.3.1 *Codebook*

O *codebook* consiste num conjunto de vetores treináveis, usados para discretizar o vetor de saída do codificador. O vetor da saída do codificador é usado para encontrar o vetor mais próximo no *codebook*. O índice do vetor mais próximo dentro do *codebook* é então guardado numa tabela. Após mapear todos os vetores de saída do codificador nos índices dos vetores mais próximos do *codebook*, a tabela serve de entrada para o descodificador, que consultando o *codebook*, utiliza os vetores correspondentes aos índices como valores das variáveis latentes (figura 4).

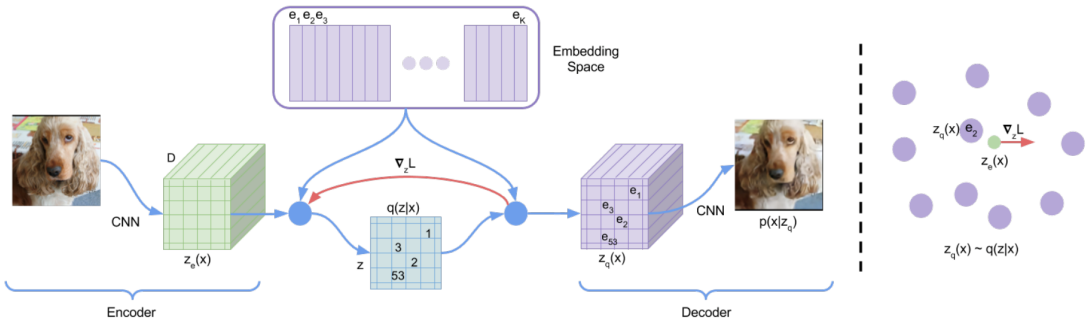


Figura 4: Arquitetura do modelo VQ-VAE.

2.3.2 *Função de Perda*

A função de perda do VQ-VAE é expressa pela equação 3, onde o termo e é a representação latente quantizada correspondente à entrada x, sendo composta por vetores do *codebook*. O operador sg [2] indica que não se aplica o cálculo dos gradientes ao seu operando, transformando este operando numa constante. O termo E representa a função de transferência do codificador e D é a função de transferência do descodificador.

2 *Stop-gradient*, na terminologia inglesa.

$$\mathcal{L}(x, D(e)) = ||x - D(e)||_2^2 + ||sg[E(x)] - e||_2^2 + \beta||E(x) - sg[e]||_2^2 \quad (3)$$

O primeiro termo corresponde ao erro de reconstrução, quantificando a diferença entre os dados reconstruídos e os dados de entrada. A aprendizagem dos vetores do *codebook* é feita efetuando a retro-propagação do gradiente do erro de reconstrução. Esta retro-propagação flui desde a entrada do descodificador até à saída do codificador, recorrendo a um processo de estimativa dos gradientes do tipo *straight-through*, onde simplesmente se copiam os gradientes da entrada do descodificador para a saída do codificador. Este processo corresponde ao fluxo a vermelho na figura 4.

O segundo termo é o erro do *codebook*. Este termo tem como objetivo aproximar o espaço do *codebook* à saída do codificador. A utilização do operador sg faz com que este termo afete apenas os vetores do *codebook* (e).

O terceiro termo é o erro de compromisso. Uma vez que o espaço do *codebook* é adimensional, pode crescer arbitrariamente, caso não treine ao mesmo ritmo que o codificador. Para assegurar que cada saída do codificador se mantenha próximo do vetor do *codebook* escolhido, prevenindo grandes flutuações, o hiper-parâmetro β controla a relação de compromisso entre o segundo e o terceiro termos da função de perda. A utilização do operador sg faz com que o gradiente deste termo afete apenas o codificador.

2.3.3 *Pixel CNN*

Pixel CNN é um modelo probabilístico autoregressivo, ou seja, faz previsões sequencialmente com base nos valores anteriores. No caso de uma imagem, o modelo itera sobre os pixeis, começando pelo canto superior esquerdo, desce linha a linha e prevê cada pixel com base nos últimos x pixeis já previstos. Este modelo também pode ser usado com informação de língua natural, para prever uma palavra, com base apenas nas primeiras letras, algo que é muito útil em teclados inteligentes.

No VQ-VAE, o Pixel Convolutional Neural Network (PixelCNN) é usado na fase de geração de novas imagens, após o modelo VQ-VAE ter sido completamente treinado na reconstrução das entradas. Um modelo PixelCNN é treinado sobre o espaço das variáveis latentes, com o objetivo de permitir escolher a entrada para o processo de descodificação. Após treinar todos os componentes do modelo VQ-VAE, o modelo PixelCNN gera uma nova instância da tabela de índices para que se possam obter os vetores do *codebook* correspondentes a esses índices, os quais funcionarão como os valores das variáveis latentes a utilizar pelo descodificador. Efetivamente, o PixelCNN escolhe as entradas do VQ-VAE quando este está a funcionar como modelo gerador.

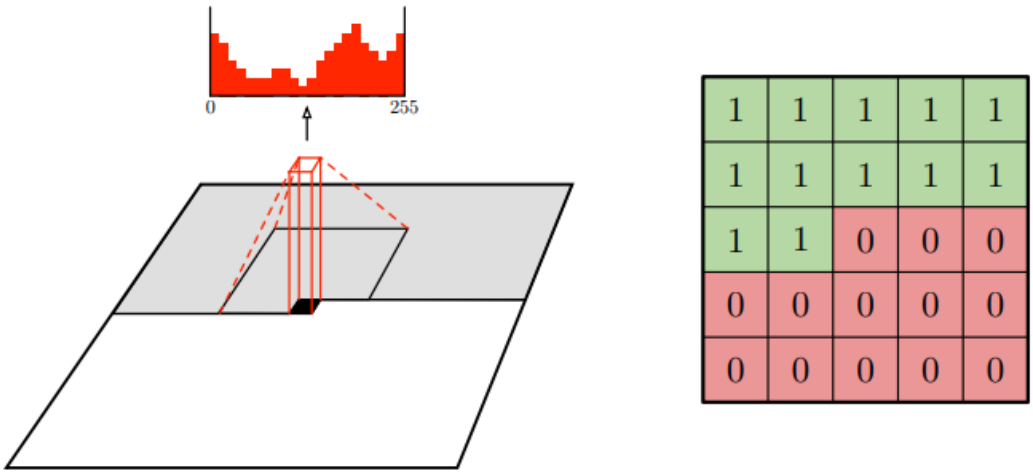


Figura 5: Arquitetura do modelo PixelCNN.

2.4 REDES ADVERSÁRIAS GERADORAS

Redes Adversárias Geradoras, ou GANs na terminologia inglesa, são o tipo de modelo com maior representatividade e número de publicações na área científica da geração de imagens. Por essa razão, estes modelos são utilizados como termo de comparação em boa parte dos artigos desta área. Este tipo de modelos foram propostos pelos autores Goodfellow et al. (2014).

2.4.1 *Arquitetura*

A arquitetura dum modelo GAN é composta por duas redes neuronais: a rede discriminadora e a rede geradora. Relativamente ao seu funcionamento, e tal como o nome indica, a rede geradora tem como objetivo gerar novas instâncias do tipo de imagem pretendida, ou seja, imagens candidatas. A rede discriminadora tem como objetivo distinguir as imagens candidatas das imagens reais obtidas do conjunto de dados de treino (figura 6).

O treino duma GAN segue um processo de otimização MiniMax, como se tratasse de um jogo. A rede geradora tenta maximizar a taxa de erro da rede discriminadora, enquanto a rede discriminadora pretende minimizar o erro. O treino destes modelos pretende atingir, ou aproximar, o equilíbrio de Nash entre as duas redes.

O equilíbrio de Nash é um conceito proveniente da área da teoria dos jogos, onde, num jogo com pelo menos dois jogadores, apesar de não haver cooperação entre os participantes, o resultado ótimo é atingível sem qualquer incentivo para os jogadores alterarem a sua

estratégia inicial. Sucintamente, mesmo sabendo as estratégias dos jogadores adversários, cada jogador não tem nenhum beneficio adicional em unilateralmente desviar-se da sua estratégia, sabendo que é a estratégia ótima.

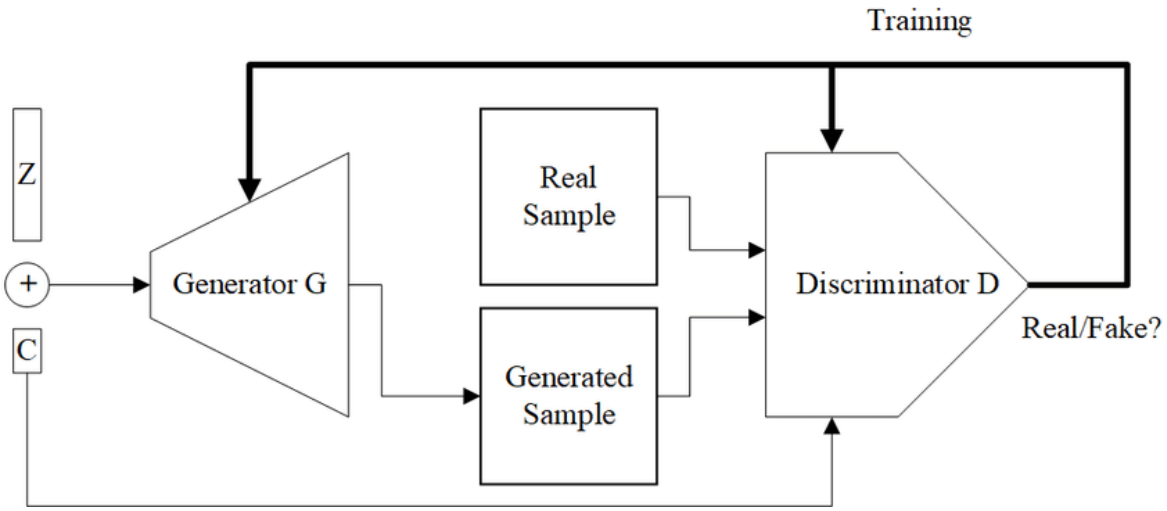


Figura 6: Arquitetura do modelo GAN.

2.4.2 *Problemas*

Apesar dos excelentes resultados apresentados por este tipo de modelo, incluindo as inúmeras variantes, existem algumas falhas significativas que são a principal causa do interesse por outras alternativas. O maior inconveniente dos modelos GAN é a grande dificuldade em os treinar. Um dos maiores obstáculos no treino é o rápido decaimento dos gradientes. Quando o discriminador mostra bons resultados, a função de perda aproxima-se rapidamente de zero, o que impede a evolução pretendida para os gradientes. Esta situação impõe um dilema complicado de conciliar durante o treino, no qual o progresso de treino do discriminador deve ser cuidadosamente equilibrado. Se o discriminador apresentar maus resultados durante muito tempo, não existe retorno necessário para que o treino dos modelos evolua fiavelmente. Se o discriminador apresentar resultados excelentes rapidamente, origina o decaimento dos gradientes.

Outra grande limitação apontada às GANs é a falta de diversidade nas saídas geradas, em consequência de uma falha denominada **colapso dos modos**. Devido à natureza da arquitetura dum GAN, pode acontecer que o modelo gerador se foque em explorar ao máximo uma determinada limitação do modelo discriminador, em vez de se focar em atingir os resultados globais que se pretendiam obter com o modelo. O colapso dos modos acontece quando o gerador encontra uma forma consistente de enganar o discriminador com sucessivas imagens que são parecidas entre si, e neste cenário o treino do modelo foca-se

apenas numa parte extremamente limitada do espaço de distribuição das imagens. Por exemplo, ao pretender-se um modelo para gerar imagens de animais felinos, o colapso dos modos pode criar uma rede geradora capaz de enganar o discriminador só com imagens de tigres, resultando num modelo final limitado e que não cumpre os objetivos pretendidos.

2.5 MODELOS DE FLUXO

Modelos de fluxo consistem na utilização de uma importante ferramenta da área da estatística, os fluxos normalizadores, na conceção de modelos geradores. Conforme a figura 7, a arquitetura de modelos de fluxo tenta encontrar o espaço latente dos dados de treino, fazendo o mapeamento dos dados às suas representações latentes com um funções de transformação invertíveis $f()$ e, consequentemente, reconstruindo os dados com as funções de transformação inversa $f^{-}1()$.

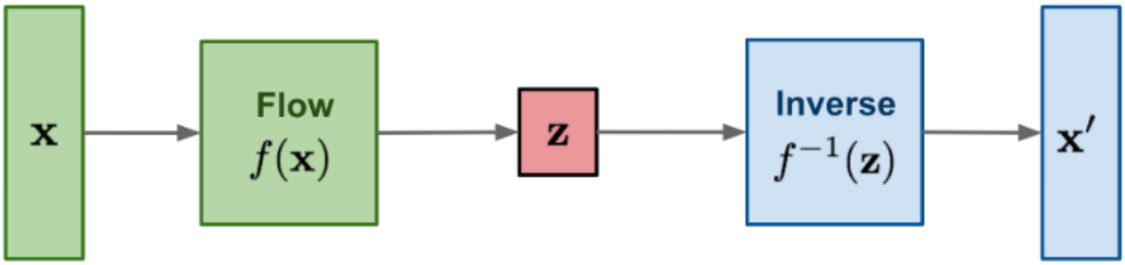


Figura 7: Arquitetura dos modelos de Fluxo.

2.5.1 *Fluxos Normalizadores*

Em modelos de fluxo, os fluxos normalizadores são responsáveis pela aplicação da sequência de funções de transformação invertíveis para encontrar o espaço latente. Com recurso ao teorema de mudança de variáveis, os fluxos normalizadores permitem transformar uma função de distribuição simples numa mais complexa. Esta transformação ocorre ao aplicar uma sequência de funções de transformação invertíveis, criando uma cadeia de transformações, até chegar à função de distribuição de probabilidade pretendida.

Exemplos de modelos de fluxo são o NICE (Dinh et al., 2015), o RealNVP (Dinh et al., 2017) e o Glow (Kingma and Dhariwal, 2018).

2.5.2 *Fluxos Autorregressivos*

Por fim, é também possível, a junção de fluxos normalizadores com modelação autorregressiva, os chamados fluxos autorregressivos. Tal como no caso do PixelCNN, um modelo

autorregressivo modela uma sequência de dados usando apenas a dependência entre um evento e os eventos anteriores na sequência. Sucintamente, fluxos autorregressivos são a modelação da função de distribuição que descreve o espaço latente como um modelo autorresgressivo, essencialmente, interpretando a sequência de transformações que visa inferir o espaço latente da imagem como um problema autorregressivo.

Exemplos de modelos de fluxo autorregressivo são o MAF (Papamakarios et al., 2018) e o IAF (Kingma et al., 2017).

2.6 MODELOS DE DIFUSÃO

Na área de geração de imagens, os modelos que mais atenção têm recebido nos últimos dois a três anos são modelos de difusão. A utilização deste tipo de modelo, por grandes empresas como Google, Adobe e OpenAI, catapultou a popularidade destes modelos muito para além do mundo académico.

Sucintamente, um modelo de difusão é definido por uma cadeia de Markov de passos de **difusão** que, lentamente e aleatoriamente, vão acrescentando ruído Gaussiano aos dados. Concluído este processo inicial, é feita a aprendizagem do modelo de modo a realizar o processo inverso ao da adição de ruído, ou seja, reconstruir os dados originais a partir da imagem com ruído, como se mostra na figura 8.

Figura 8: Arquitetura do processo de difusão.

Alguns exemplos de modelos de difusão são o modelo probabilístico de difusão com base em termodinâmica de não-equilíbrio (Sohl-Dickstein et al., 2015), DDPM (Ho et al., 2020), NCSN (Song and Ermon, 2020) e DDIM (Song et al., 2021). Nos últimos dois anos foram desenvolvidos modelos de difusão de elevado potencial para a geração de imagens de alta qualidade, nomeadamente os modelos DALLE-2 (Saharia et al., 2022), Imagen (Ramesh et al., 2022) e Stable Diffusion (Rombach et al., 2022).

2.6.1 *Processo de Difusão*

A primeira tarefa do treino de um modelo de Difusão corresponde a um processo em que repetitivamente se vai adicionando ruído Gaussiano aos dados de treino, em que em cada passo deste processo se acrescenta uma quantidade diminuta de ruído aos dados, já com ruído, resultantes do passo anterior, até que a informação original é completamente irreconhecível nos dados. Assim, os dados vão perdendo as características distintivas e tornam-se essencialmente ruído.

2.6.2 *Processo Inverso da Difusão*

Após transformar as imagens de treino em ruído puro, a principal aprendizagem dos modelos de difusão acontece no processo de reversão do ruído. Novamente, esta tarefa também decorre iterativamente em pequenos passos, onde a diferença entre uma imagem ruidosa (do passo anterior) e uma imagem ligeiramente menos ruidosa (do passo atual), pode ser descrita a partir de uma distribuição gaussiana. Assim sendo, a aprendizagem dos modelos de difusão consiste em aprender as médias e variâncias dessas distribuições gaussianas. No final, um modelo treinado será capaz de transformar uma imagem contendo apenas ruído gaussiano numa imagem clara.

2.6.3 *Geração Condicionada*

A fim de obter imagens geradas mais próximas do que se pretende, é possível condicionar o conteúdo gerado a uma determinada classe de imagens. Este método já foi implementado pelos autores Dhariwal and Nichol (2021), que treinaram um classificador auxiliar. Recorrendo a gradientes, este classificador foi usado como auxílio no treino, para conduzir o processo de difusão em direção às classes pretendidas. Comparativamente aos modelos GAN, este modelo apresentou bons resultados, demonstrando enorme potencial na geração de imagens de determinado tipo.

2.6.4 *Espaço Latente de Difusão*

Até à data, o grande avanço nos modelos de difusão aconteceu com a introdução do modelo Stable Diffusion. A sua principal característica é a capacidade de manter a qualidade da geração, com um peso computacional bastante reduzido em comparação com os outros modelos que geram imagens de qualidade semelhante. A solução apresentada pelos autores do modelo Stable Diffusion consiste em aplicar o processo de difusão num espaço latente, em vez de o aplicar diretamente no espaço dos pixeis da imagem. A proposta do modelo em

causa é a introdução de um modelo *autoencoder* pré-treinado, no processo de transformação de amostras nas imagens finais. Por conseguinte, é feita uma separação de responsabilidades entre o *autoencoder* e o modelo de difusão. O *autoencoder* trata da codificação e descodificação entre as imagens e a sua representação no espaço latente, uma tarefa em que é mais eficaz e mais eficiente, e representa a maior parte do peso computacional do modelo global Stable Diffusion. O modelo de difusão é aplicado na tarefa que é o seu ponto forte, isto é, na geração de amostras a partir dum espaço percetualmente equivalente ao espaço da imagem, mas com uma complexidade computacional bastante menor.

2.7 TRABALHO RELACIONADO

Nesta secção iremos resumir as publicações mais relevantes relacionadas com o tema da dissertação, ou seja, a geração de imagens com modelos baseados em *variational autoencoders*.

A versão mais recente do modelo VQ-VAE, denominada VQ-VAE2, adota uma organização hierárquica multi-escala de VQ-VAE e foi desenvolvida pelos mesmos autores da arquitetura original (van den Oord et al., 2019). No que respeita à qualidade das imagens geradas, a nova versão demonstra capacidade de rivalizar com as GAN, sem sofrer das mesmas lacunas: o colapso dos modos e a falta de diversidade nas imagens geradas. Resumidamente, o principal destaque do VQ-VAE2 consiste na implementação de uma estrutura hierárquica, que se diferencia da estrutura linear do modelo VQ-VAE original. Esta variante tem como objetivo separar a modelação da informação local (como a textura), da informação global (como a forma ou a geometria dos objetos). A camada superior da hierarquia foca-se na informação global, enquanto que a camada inferior, que é condicionada pela camada superior, é responsável pela representação dos detalhes locais. Os resultados apresentados pelo modelo VQ-VAE2 demonstram melhor acurácia, quando comparados com o modelo BigGAN. Os autores afirmam ainda que o BigGAN tem melhor precisão, enquanto que o VQ-VAE2 tem melhor *recall*. Perante estes resultados, os autores chegam à conclusão de que o VQ-VAE2 é um competidor direto com os modelos GAN de estado de arte, sendo até superior no que respeita à diversidade, uma das principais deficiências das GANs .

Os autores Huang et al. (2018) introduziram um modelo denominado IntroVAE, que consiste numa implementação híbrida de VAE e GAN, na qual se utilizam aspetos de ambos os modelos. Mais concretamente, é usada a arquitetura do VAE, composta por codificador e descodificador, com recurso a distribuições de probabilidade, e utilizado o treino adversário, que é uma abordagem das GANs. O IntroVAE tem como objetivo treinar um modelo VAE capaz de reconstruir e gerar imagens, recorrendo a aprendizagem adversária. O codificador atua como componente discriminador e o descodificador como o componente gerador, combinando-se desta forma as arquiteturas VAE e GAN. O treino do modelo IntroVAE

começa com a redução e consequente reconstrução de uma imagem real, tal como no modelo VAE. Em simultâneo, é gerada uma imagem através do descodificador. As duas imagens, reconstruída e gerada, são avaliadas pelo codificador, na tentativa de as distinguir. Desta forma, a função de perda irá aplicar as abordagens tanto do VAE como das GANs para treinar o modelo. Os resultados desta proposta são comparados com os de Progressive Growing of Generative Adversarial Network (PGGAN)(dos autores Karras et al. (2018)), em que o IntroVAE revela níveis de qualidade e de diversidade semelhantes ou mesmo superiores. Em conclusão, os autores afirmam que o modelo tem capacidade para obter resultados similares, ou até melhores, que os principais modelos geradores, possuindo o IntroVAE uma arquitetura mais simples e eficiente. Sugerem ainda, a possibilidade de aplicar esta abordagem a outras variantes do VAE.

Num trabalho anteriormente publicado pelos autores do modelo VQ-VAE, van den Oord et al. (2016), é demonstrada a aplicação do modelo autorregressivo PixelCNN na geração de imagens. Nesse trabalho é explorada uma arquitetura convolucional, em alternativa ao modelo Pixel Recurrent Neural Network (PixelRNN) que utiliza uma Recurrent Neural Network (RNN). O principal destaque é a introdução de camadas convolucionais fechadas (*gated*), com uma função de ativação fechada, criada pelos autores. Esta função de ativação fechada tem como propósito aproximar o PixelCNN do PixelRNN em relação a capacidade de modelagem de interações complexas entre os pixeis da vizinhança. O trabalho também explora a utilização de um PixelCNN condicional, que funciona de forma semelhante ao descodificador do VAE, fazendo a modelação da função de distribuição de probabilidade condicional duma imagem, a partir de um vetor latente. Os resultados obtidos pela *gated* PixelCNN são comparados com os do modelo PixelRNN. De acordo com as métricas avaliadas, a arquitetura *gated* PixelCNN apresenta melhores resultados. No entanto, a falta de diversidade dos testes, juntamente com as diferenças mínimas entre a *gated* PixelCNN e a PixelRNN, não demonstra uma clara e evidente melhoria de desempenho sobre o estado de arte dos modelos autorregressivos, na tarefa de geração de imagens. Em termos globais, os autores concluem que a *gated* PixelCNN é uma versão melhorada da arquitetura PixelCNN original, não tanto ao nível do desempenho, mas essencialmente em termos de eficiência computacional. O artigo termina com uma menção à possível utilização do modelo PixelCNN em conjunto com VAEs, algo que viria poucos anos depois a ser concretizado no modelo VQ-VAE.

Os autores Nash et al. (2021) introduzem um modelo gerador de imagens, baseado em representações de imagem, do tipo transformação discreta de cosseno [3]. O modelo usa uma arquitetura autorregressiva, denominada DCTransformer, que é treinada para prever onde adicionar conteúdo numa imagem e qual o conteúdo a adicionar. A implementação

3 Discrete Cosine Transform (DCT), na terminologia inglesa.

deste modelo é baseada no método de compressão de imagens Joint Photographic Experts Group (JPEG), cujas imagens de entrada são comprimidas numa representação esparsa, depois de passar por uma quantização em blocos de 8x8 pixeis. A representação esparsa consiste numa sequência com três canais, sendo um de informação referente à luminosidade e os outros dois relativos a componentes de cor. Por sua vez, cada um destes canais é composto por uma lista esparsa de tuplos, contendo informação do canal DCT, a posição espacial e o valor. A parte autorregressiva do modelo tenta prever cada um dos três elementos dos tuplos, começando pelo canal DCT, depois a posição espacial e por fim o valor. Devido a uma utilização excessiva de memória, que é comum nos modelos com arquitetura *transformer*, a previsão acontece em blocos de 896 tuplos. A geração de novas imagens pelo DCTransformer é idêntica à do PixelCNN, consistindo na previsão sequencial de blocos com base nos resultados anteriormente previstos. Comparando o DCTransformer com diversas arquiteturas GAN, tais como StyleGAN, StyleGAN2, ProGAN e BigGAN-deep, os resultados obtidos não mostram um clara vantagem sobre nenhumas das GANs. O DCTransformer exibe melhores resultados ao nível da métrica *recall*, enquanto que as GANs são superiores ao nível da precisão. Os autores observam que o DCTransformer não apresenta as clássicas falhas das GANs, em termos de diversidade e estabilidade de treino, contudo não é tão fiável como as GANs na geração de imagens coerentes. Os autores apontam ainda a similaridade entre a abordagem adotada no DCTransformer e a que foi seguida no VQ-VAE2, por usar também um modelo autorregressivo e recorrer a representações comprimidas das imagens. Por fim, sugerem a exploração de abordagens baseadas em métodos de compressão clássicos, dados os resultados positivos, particularmente ao nível da diversidade. Além disso, é apontado como o maior desafio desta abordagem a necessidade de recursos computacionais substanciais, para se conseguir gerar imagens complexas e de alta resolução.

Numa abordagem alternativa, os autores Schreyer et al. (2020) demonstram a versatilidade do modelo VQ-VAE, ao aplica-lo à área financeira. Devido à crescente quantidade de extratos bancários e ao elevado número de entradas que os compõem, é impossível uma análise detalhada de todas as entradas. Como tal, num processo de auditoria financeira é essencial uma amostragem representativa dos dados. Com o intuito de melhorar o processo de amostragem de auditorias, aumentando a fiabilidade da representatividade das amostras, recorreu-se a um modelo VQ-VAE. O modelo implementado utiliza como entrada um extrato financeiro, composto por informação do tipo: data, montante, tipo de transação e instituição. Como em qualquer VQ-VAE, o codificador transforma esse tipo de entrada numa representação latente que é então discretizada, com recurso aos vetores do *codebook*. Essa mesma representação discretizada é utilizada como entrada do descodificador para reconstruir a entrada original. O objetivo final do treino deste modelo é, não um modelo gerador, mas sim os vetores do *codebook* que, após a fase de treino, serão um conjunto de representações latentes de amostras representativas dos extratos que se pretende auditar.

Fundamentalmente, o modelo VQ-VAE é usado como um algoritmo de agrupamento [4]. Para além de identificar grupos, também obtém amostras representativas de cada um deles. Os resultados apresentados na publicação são avaliados como positivos, ao nível quantitativo, ao nível qualitativo e em relação ao desenredamento [5] dimensional. Em conclusão, é possível utilizar modelos VQ-VAE para obter amostras de auditorias financeiras, cujo potencial a explorar poderá aumentar e melhorar os recursos disponíveis aos auditores.

Os autores Peng et al. (2021) demonstram a aplicação do modelo VQ-VAE, com uma arquitetura hierárquica, no preenchimento de uma imagem incompleta. Sucintamente, dada uma imagem com uma porção significativa bloqueada por um quadrado cinzento, o modelo implementado devolve várias versões possíveis da reconstrução da imagem original desobstruída. O modelo desenvolvido consiste em três componentes: o VQ-VAE hierárquico, um gerador de estruturas diversas e um gerador de textura. A primeira parte a treinar é o VQ-VAE, que separa a informação relativa às estruturas e às texturas, tal como acontece no VQ-VAE2 original. As características estruturais referem-se a formas e cores, enquanto que as características de textura são relativas a detalhes locais. O principal utilidade do VQ-VAE no modelo global é essa separação, que é feita pelo codificador e que posteriormente é discretizada com recurso a *codebooks*. A segunda parte a ser treinada é o gerador de estruturas, o qual usa um modelo autorregressivo capaz de criar uma distribuição de probabilidade sobre as características estruturais identificadas pelo VQ-VAE. Durante a fase de treino, este gerador recebe uma imagem incompleta como entrada. Depois, cria a distribuição de características a partir da qual se retirarão amostras, gerando assim as características estruturais. Por fim, treina-se o gerador de texturas que utiliza as características de textura, identificadas pelo VQ-VAE e, as características estruturais, geradas pelo gerador de estruturas, para obter o resultado final. Este processo de treino é finalizado com a comparação da imagem original com a imagem gerada, pelo codificador do VQ-VAE já treinado. Os resultados obtidos são comparados com os de outros modelos de preenchimento de imagens incompletas, nomeadamente o modelo UCTGAN, uma arquitetura GAN. A nível qualitativo, este modelo mostra resultados mais realistas, mais detalhados e com uma maior diversidade. De um modo geral, o modelo revela melhor desempenho, particularmente no conjunto de métricas que medem a qualidade percetual das imagens. Em suma, os autores concluem que a aplicação do VQ-VAE na resolução do problema de preenchimento de imagens incompletas é vantajoso. A divisão hierárquica do componente VQ-VAE e do modelo em geral, parece facilitar uma geração estruturada e detalhada da informação em falta, mantendo a qualidade e a diversidade.

4 *Clustering*, na terminologia inglesa.
5 *Disentanglement*, na terminologia inglesa.

3

DESENVOLVIMENTO DE MODELOS

Este capítulo documenta a implementação de modelos geradores de imagens com faces humanas, incluindo as tecnologias e os conjuntos de dados utilizados.

3.1 TECNOLOGIAS

Esta secção apresenta as tecnologias utilizadas, ao longo do desenvolvimento da dissertação, que por sua vez está dividida em subsecções. Cada uma delas corresponde a uma tecnologia.

3.1.1 *Google Colaboratory*

Google Colaboratory é um produto da Google (Google (2017)) que utiliza um ambiente *Jupyter Notebook*. Este ambiente da Google tem duas principais vantagens que agilizam o desenvolvimento do código: a possibilidade do uso de Google Drive, para guardar dados na nuvem e facultar o acesso remoto a *GPUs* externos.

Neste trabalho, o Google Colaboratory foi utilizado principalmente numa fase inicial, com o propósito de experimentar alguns exemplos de modelos. Estes foram usados como base do desenvolvimento das implementações finais. A sua acessibilidade facilitou bastante este primeiro processo de pesquisa e teste de implementações pré-existentes.

3.1.2 *Jupyter Notebook*

Jupyter Notebook é um ambiente computacional Web que permite a criação e execução de documentos *notebook IPython*. Estes documentos são compostos por várias células, que podem conter código *python*, texto com recurso a linguagem *Markdown*, e outros tipos de conteúdos como imagens. Nesta dissertação, o *Jupyter Notebook* foi o principal ambiente de desenvolvimento usado, através da interface interativa *JupyterLab*. A partir deste ambiente, foram usadas as várias bibliotecas e recursos incorporadas no *Jupyter* (Jupyter (2014)), desde

componentes mais gerais de *Python* como *NumPy*, a meios específicos no desenvolvimento de aprendizagem automática, como por exemplo *TensorFlow* e *Keras*.

3.1.3 *TensorFlow*

TensorFlow é uma biblioteca de código aberto para aprendizagem automática/profunda, desenvolvida pela Google (Tensorflow (2015)). Juntamente com Pytorch, são as duas bibliotecas mais utilizadas em trabalhos de investigação publicados e em ambientes empresariais, na área da aprendizagem profunda. É uma ferramenta que pode ser utilizada com diversos objetivos, sendo os principais a implementação e o treino de redes neuronais artificiais.

3.1.4 *Keras*

Keras é uma biblioteca de código aberto com interface em *Python*, focada na área das redes neuronais artificiais (Keras (2015)). Neste momento está integrada no *TensorFlow*, sendo a sua principal API para modelação de redes neuronais artificiais. A biblioteca disponibiliza inúmeras implementações de modelos de aprendizagem profunda. Alguns destes modelos foram utilizados neste trabalho, como ponto de partida para a implementações dos modelos pretendidos.

3.1.5 *Weights and Biases*

A plataforma Weights and Biases (W&B) é uma ferramenta de gestão do processo de treino de modelos de aprendizagem automática. Esta ferramenta permite um registo automático e organizado de cada experiência realizada e a imediata visualização dos dados de treino e dos resultados obtidos. Para além disso, permite alguma automatização do processo de otimização dos modelos.

Devido à dificuldade em manter uma ligação constante ao servidor W&B por longos períodos de tempo, a utilização intensiva da plataforma ocorreu na fase inicial do trabalho, quanto se desenvolviam os modelos mais simples, e na fase de recolha de resultados para a escrita da dissertação. A implementação de modelos mais complexos e a utilização de dados que requeriam mais poder computacional, exigiram tempos de treino elevados, causando demasiadas desconexões com os servidores W&B. Apesar de curta, a utilização do *Weights and Biases* permitiu obter informação valiosa para o desenvolvimento dos modelos.

3.2 CONJUNTOS DE DADOS

Para treinar modelos capazes de gerar imagens de faces humanas, com alta resolução, é preciso dispor de um conjunto de dados com imagens do mesmo tipo que se pretende gerar. No entanto, o manusear um conjunto de dados composto por muitas imagens de alta resolução requer elevados recursos computacionais e de armazenamento, dificultando assim o desenvolvimento dos modelos. Deste modo, a metodologia de desenvolvimento adotada incluiu várias soluções para este problema. Nesta secção são apresentados os dois conjuntos de dados utilizados neste trabalho, assim como algumas funções em Python criadas para manusear esses dados mais facilmente.

3.2.1 *MNIST*

Numa primeira fase, da implementação de cada um dos modelos, utilizou-se um conjunto de dados simples e composto por imagens de menor dimensão. O conjunto de dados em causa é o clássico Modified National Institute of Standards and Technology database (MNIST). Para efeitos de termo de comparação entre modelos e para servir de referencial, o conjunto de dados MNIST é habitualmente usado no treino de diversos modelos na área aplicacional de visão por computador e em inúmeros trabalhos sobre aprendizagem automática. O MNIST foi criada a partir de uma combinação de dois conjuntos de dados do National Institute of Standards Technology (NIST). O conjunto de dados MNIST consiste em 60000 imagens de algarismos entre 0 e 9, escritos à mão. O tamanho das imagens que compõem o conjunto de dados é 28×28 pixeis (figura 9).

Figura 9: Amostras do conjunto de dados MNIST.

A baixa resolução das imagens permite classificá-las com modelos mais simples e utilizando um tempo de treino mais curto, comparativamente a imagens de alta resolução. Além da baixa resolução, a simplicidade das imagens permite uma maior facilidade de otimização dos modelos e até uma facilitada análise visual dos resultados. São estas as razões pelas quais a primeira versão de cada modelo foi treinada com o conjunto de dados MNIST.

3.2.2 *FFHQ*

Com base no estado da arte e em trabalhos relacionados da área, foi possível identificar conjuntos de dados úteis ao presente trabalho. O conjunto de dados escolhido foi o Flickr-Faces-HQ (FFHQ). O conjunto de dados FFHQ consiste em imagens de alta qualidade, disponíveis em várias resoluções. As imagens que compõem este conjunto de dados foram obtidas do sítio Web `Flickr`, através do método *crawl* (figura 10).

Figura 10: Amostras do conjunto de dados FFHQ.

Após treinar os modelos com o conjunto de dados MNIST, o desenvolvimento de modelos evolui para o objetivo da geração de imagens de faces humanas, começando com um conjunto de 70000 imagens de 128×128 pixeis, por forma a permitir tempos de treino mais curtos. O objetivo desta fase do trabalho era estabelecer uma base sobre a qual se pudesse evoluir, de forma faseada, para imagens de maior resolução. Depois de obter uma versão intermédia dos modelos, com uma resolução menor do que a pretendida, realizou-se a transição para imagens de maior resolução, concretamente um conjunto de 52000 imagens com 512×512 pixeis. Como era expectável, este aumento da resolução das imagens resultou em necessidades computacionais significativamente maiores.

A utilização dos dados no formato `TFRecord` permitiu reduzir os requisitos de armazenamento em memória, inerentes a um conjunto de dados com imagens de alta resolução. Este formato possibilita uma leitura e manuseamento dos dados mais eficientes. Para que tal aconteça, foi preciso guardar as imagens originais no formato `TFRecord`. A listagem 3.1 mostra a implementação das funções que convertem os valores das imagens originais para as `features` do formato `TFRecord`, conforme o tipo de dados em causa. As funções apresentadas cobrem os tipos de dados `int`, `float` e `byte`.

A listagem 3.2 mostra a função que transforma uma imagem num elemento `TFRecord`. Uma vez que neste caso se trata de imagens, a representação da imagem no formato `TFRecord` é feita com uma lista de `bytes`.

```
def _bytes_feature(value):
    if isinstance(value, type(tf.constant(0))):
        value = value.numpy()
    return tf.train.Feature(bytes_list=tf.train.BytesList(value=[value]))

def _float_feature(value):
    return tf.train.Feature(float_list=tf.train.FloatList(value=[value]))

def _int64_feature(value):
    return tf.train.Feature(int64_list=tf.train.Int64List(value=[value]))
```

Listagem 3.1: Conversão de valores `int`, `float` e `byte` para as `features` do formato `TFRecord`.

```
def image_example(image_string):
    image_shape = tf.io.decode_jpeg(image_string).shape

    feature = {
        "image_raw": _bytes_feature(image_string),
    }
    return tf.train.Example(features=tf.train.Features(feature=feature))
```

Listagem 3.2: Conversão de uma imagem para um elemento do formato `TFRecord`.

De seguida, percorre-se todo o conjunto de dados sequencialmente, transformando cada imagem num elemento `TFRecord` e guardando a informação no número de ficheiros pretendido (listagem 3.3).

```
sec = tfrecord_split
t = 0
for i in range(n_split):
    train_tfrecords_dir=train_main_dir+str(i+1)+'.tfrecords'
    with tf.io.TFRecordWriter(train_tfrecords_dir) as writer:
        for filename in os.listdir(data_dir):
            if t<sec:
                t+=1
                file=data_dir+'/'+filename
                image_string = open(file, 'rb').read()
                tf.example = image_example(image_string)
                writer.write(tf.example.SerializeToString())
            else:
                break
        sec+=tfrecord_split
```

Listagem 3.3: Escrever os ficheiros `TFRfrecord`.

Quando se lê um ficheiro `TFRecord` é necessário descodificar as imagens nele guardadas. Esse processo começa com a função `decode_jpeg`, disponibilizada pelo `TensorFlow`, a qual devolve uma imagem representada com uma lista de números inteiros. Depois, convertem-se os valores inteiros para o tipo `float` e normalizam-se os valores convertidos, dividindo-os por 255. Finalmente, redimensiona-se a lista de valores `float` para a dimensão original da imagem, antes de esta ter sido convertida para o formato `TFRecord` (listagem 3.4).

```
def decode_image(image):
    image = tf.image.decode_jpeg(image, channels=3)
    image = tf.cast(image, tf.float32)/255
    image = tf.reshape(image, [IMAGE_SIZE, IMAGE_SIZE, 3])
    return image
```

Listagem 3.4: Descodificação de uma imagem a partir do formato `TFRecord`.

O código da listagem 3.5 permite ler um elemento a partir do formato `TFRecord` e devolver uma imagem, recorrendo à função incluída na listagem 3.4.

```
def read_tfrecord(filenames, labeled=False):
    tfrecord_format = (
        {
            "image_raw": tf.io.FixedLenFeature([], tf.string),
            "label": tf.io.FixedLenFeature([], tf.int64),
        }
        if labeled
        else {"image_raw": tf.io.FixedLenFeature([], tf.string),}
    )
    example = tf.io.parse_single_example(example, tfrecord_format)
    image = decode_image(example["image_raw"])
    if labeled:
        label = tf.cast(example["label"], tf.int64)
        return image, image
    return image
```

Listagem 3.5: Leitura de um elemento a partir do formato `TFRecord`.

O código incluído na listagem 3.6 dá acesso a um conjunto de dados completo, guardado em ficheiros `TFRecord`, utilizando a função `read_tfrecord` para mapear cada elemento `TFRecord` numa imagem. A função `load_dataset` devolve um objeto do tipo `Dataset`.

```
def load_dataset(filenames, labeled=False):
    ignore_order = tf.data.Options()
    ignore_order.experimental_deterministic = False
    dataset = tf.data.TFRecordDataset(filenames)
    dataset = dataset.with_options(ignore_order)
    dataset = dataset.map(
        partial(read_tfrecord, labeled=labeled), num_parallel_calls=AUTOTUNE))
    return dataset
```

Listagem 3.6: Acesso a um conjunto de dados guardado em ficheiros `TFRecord`.

Depois de ter acesso a um objeto do tipo `Dataset`, os dados do conjunto de dados podem ser lidos para memória *batch* a *batch*, em que o número de imagens num *batch* é `BATCH_SIZE` (listagem 3.7).

```
def get_dataset(filenames, labeled=False):
    dataset = load_dataset(filenames, labeled=labeled)
    dataset = dataset.shuffle(1024)
    dataset = dataset.prefetch(buffer_size=AUTOTUNE)
    dataset = dataset.batch(BATCH_SIZE)
    return dataset
```

Listagem 3.7: Acesso a um conjunto de dados para ser lido *batch* a *batch*.

A utilização de `TFRecords` não se limitou apenas ao conjunto de dados principal, mas também a outros dados auxiliares. Por exemplo, a representação latente das imagens, que é necessária para treinar os modelos PixelCNN, também foi guardada no formato `TFRecord`. A representação latente das imagens, que é preciso aplicar na entrada do modelo PixelCNN durante o seu treino, é gerada pelo codificador do VQ-VAE, o qual foi previamente treinado. Para facilitar o manuseamento da representação latente, reduzindo as necessidades de armazenamento, esta representação também foi guardada em formato `TFRecord`, recorrendo a um processo análogo ao que se utilizou para guardar o conjunto de dados principal.

3.3 MODELOS

Esta secção documenta a implementação de todos os modelos desenvolvidos ao longo da dissertação.

3.3.1 *Implementação do Variational Autoencoder*

O modelo VAE implementado tem como base o exemplo fornecido pelo `Keras` (Chollet (2020)). Durante o desenvolvimento do modelo VAE, experimentaram-se diferentes tipos de camada no codificador. A utilização de camadas de *pooling* no codificador introduziu um efeito negativo no desempenho do modelo. Também foi testada a inclusão de camadas de *dropout* no codificador e no descodificador, o que revelou efeitos positivos. Outro aspeto importante testado nesta fase, foi a definição das dimensões latentes, entre codificador e descodificador. Dado que a arquitetura dum VAE pressupõe uma redução da dimensão dos dados de entrada para a sua consequente reconstrução, é relevante encontrar o melhor valor para a dimensão latente, mantendo-se fiel ao conceito do modelo. Quanto maior for a dimensão latente, melhores serão os resultados de reconstrução, no entanto, deve-se impor limites para não desvirtuar o modelo que se está implementar.

Codificador

O codificador pode ser divido em três partes:

- A primeira parte é a codificação propriamente dita, sendo composta por blocos de camadas convolucionais 2D e camadas de normalização de *batch*, com o intuito de reduzir a dimensão da imagem de entrada para a dimensão latente pretendida, mantendo uma normalização dos dados;
- A segunda parte do codificador serve de ponte entre os dados codificados e as funções de distribuição de probabilidade. Surgindo depois da fase de codificação, esta parte começa com uma camada de `Dropout`, a fim de reduzir o sobreajuste do processo de codificação. Posteriormente, aplica-se uma camada `Flatten`, que transforma os dados num vetor de uma dimensão, finalizando esta parte com uma camada densamente ligada (`Dense`);
- A última parte da implementação do codificador é a incorporação das funções de distribuição de probabilidade no modelo. Começa com uma bifurcação do grafo da rede neuronal, através de uma camada que calcula a média e outra que calcula a variância. Os vetores média e variância caraterizam o vetor função de distribuição de probabilidade. De seguida, é feita a junção do grafo através de uma camada de amostragem (`Sampling`), que aplica o truque de reparametrização caraterístico do modelo VAE. As saídas do codificador são (i) a média e a variância, as quais caraterizam a função de distribuição de probabilidade, e (ii) uma amostra obtida dessa distribuição, que representa a codificação dos dados de entrada.

A implementação do codificador pode ser consultada na listagem 3.8 e a arquitetura correspondente encontra-se na figura 11.

Descodificador

O descodificador recebe uma amostra como entrada e transforma-a numa imagem. Esta implementação começa com uma camada densamente ligada, que contém um número de unidades igual à dimensão dos dados que saem da camada `Flatten` do codificador. Depois existe uma camada `Reshape`, para redimensionar os dados recebidos para a dimensão pretendida, que neste caso é igual à dimensão dos dados que entram na camada `Flatten` do codificador. A parte central do descodificador consiste em camadas convolucionais 2D transpostas que, passo a passo, transformam a amostra recebida numa imagem com as dimensões da imagem original. Antes da última camada convolucional 2D transposta, insere-se uma camada de `Dropout`, com o objetivo de reduzir o possível sobreajuste e melhorar assim a capacidade de geração do modelo.

A implementação do codificador pode ser consultada na listagem 3.9 e a arquitetura correspondente encontra-se na figura 12.

```
def create_encoder(latent_dim):
    image_dimension = 3
    input_image = Input(shape=(IMG_HEIGHT, IMG_WIDTH, img_dimension))
    encoded = Conv2D(16, 3, activation='relu', strides=2, padding='same')(input_image)
    encoded = tf.keras.layers.BatchNormalization()(encoded)

    encoded = Conv2D(32, 3, activation='relu', strides=2, padding='same')(encoded)
    encoded = tf.keras.layers.BatchNormalization()(encoded)

    encoded = Conv2D(64, 3, activation='relu', strides=2, padding='same')(encoded)
    encoded = tf.keras.layers.BatchNormalization()(encoded)

    encoded = Conv2D(128, 3, activation='relu', strides=2, padding='same')(encoded)
    encoded = Dropout(DROPOUT)(encoded)
    encoded = Flatten()(encoded)
    encoded = Dense(32, activation='relu')(encoded)
    z_mean = tf.keras.layers(latent_dim, name='z_mean')(encoded)
    z_log_sigma = tf.keras.layers(latent_dim, name='z_log_sigma')(encoded)
    z = Sampling()([z_mean, z_log_sigma])
    encoder = tf.keras.Model(input_image, outputs=[z_mean, z_log_sigma, z], name="encoder")
    return encoder
```

Listagem 3.8: Implementação do codificador do modelo VAE.

```
def create_decoder(latent_dim):
    latent_inputs = keras.Input(shape=(latent_dim,))
    decoded = Dense(8 * 8 * 128, activation='relu')(latent_inputs)
    decoded = Reshape((8, 8, 128))(decoded)
    decoded = Conv2DTranspose(128, 3, activation='relu', strides=2, padding='same')(decoded)
    decoded = Conv2DTranspose(64, 3, activation='relu', strides=2, padding='same')(decoded)
    decoded = Conv2DTranspose(32, 3, activation='relu', strides=2, padding='same')(decoded)
    decoded = Conv2DTranspose(16, 3, activation='relu', strides=2, padding='same')(decoded)
    decoded = Dropout(DROPOUT)(decoded)
    decoder_outputs = Conv2DTranspose(3, 3, activation='sigmoid', padding='same')(decoded)

    decoder = tf.keras.Model(latent_inputs, decoder_outputs, name="decoder")
    return decoder
```

Listagem 3.9: Implementação do descodificadordo modelo VAE.

Composição do Modelo

O modelo VAE é construído juntando os componentes codificador e descodificador. Daí que, o processo de composição do modelo consiste na criação desses dois componentes e na definição da interação entre eles. Especifica-se que o codificador recebe uma imagem como

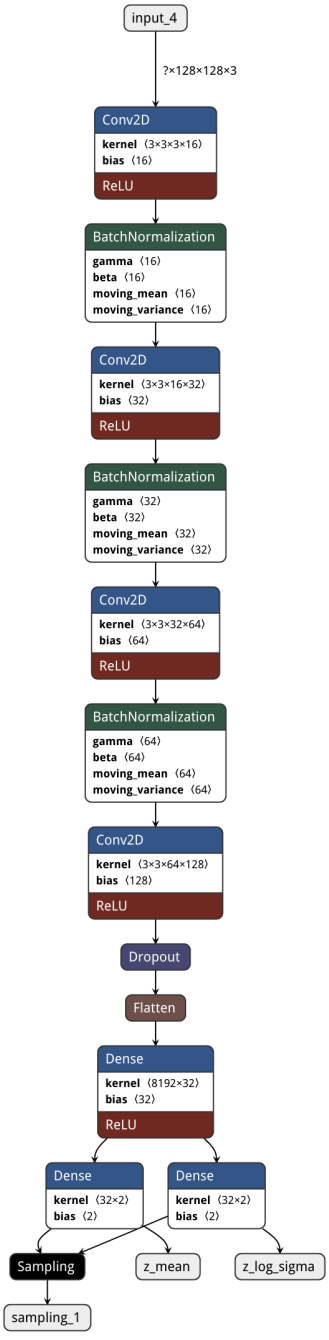


Figura 11: Arquitetura do codificador do modelo VAE implementado.

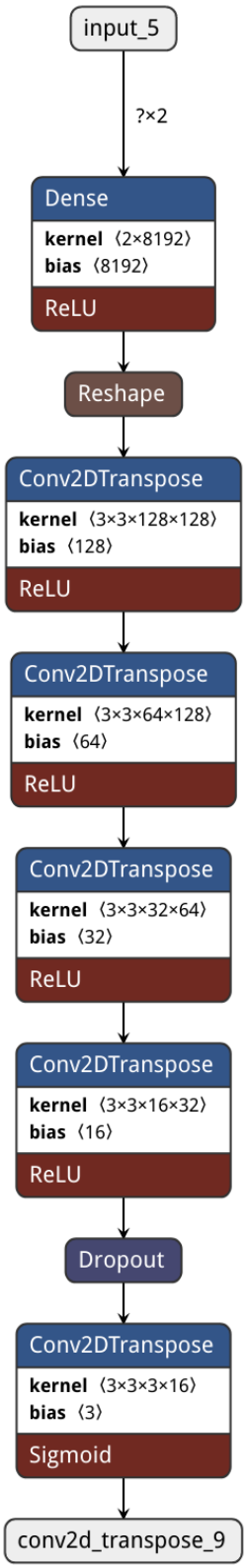


Figura 12: Arquitetura do descodificador do modelo VAE implementado.

entrada e gera três saídas: (i) o vetor com a média da função de distribuição de probabilidade, (ii) o vetor com a variância da mesma distribuição e (iii) a codificação dos dados de entrada, através da amostragem da distribuição, recorrendo ao truque de reparametrização. O papel do descodificador é reconstruir a imagem original, a partir da amostra, ou codificação, que recebe como entrada.

A implementação do VAE encontra-se na listagem 3.10 e a arquitetura correspondente pode ser consultada na figura 13.

```
def get_vae(latent_dim):
    encoder = create_encoder(latent_dim)
    decoder = create_decoder(latent_dim)
    inputs = keras.Input(shape=(IMAGE_SIZE, IMAGE_SIZE, 3))
    z_mean, z_log_var, z = encoder(inputs)
    reconstructions = decoder(z)
    vae = keras.Model(inputs, reconstructions, name="vae")
    return vae
```

Listagem 3.10: Implementação do modelo VAE.

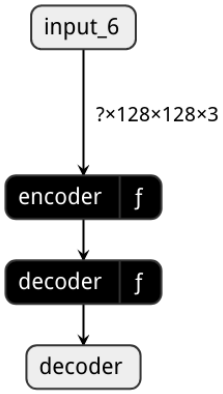


Figura 13: Arquitetura geral do modelo VAE implementado.

3.3.2 *Implementação do Modelo VQ-VAE*

A implementação do modelo VQ-VAE apresentou algumas dificuldades, mas o facto de ter partes comuns ao VAE ajudou no processo. Em grande parte, a implementação baseou-se no exemplo fornecido pelo `Keras` (Paul (2021)). Na fase inicial do desenvolvimento do modelo, foi identificada uma dificuldade computacional para obter a variância dos dados. Por este motivo, numa fase inicial ignorou-se a influência da variância na função de perda. No entanto, foi possível constatar a importância do valor da variância e qual o impacto que tem no treino do modelo. Enquanto a variância não foi introduzida na função de perda, pôde-se verificar que a sua ausência diminuía a perda para valores demasiado próximos de 0, ao

ponto de reduzir a progressão do treino do modelo. Descoberta a importância da variância no treino do modelo, foi necessário obter o seu valor, efetuando o cálculo de forma faseada e obtendo-se como valor final a média dos resultados intermédios. Ao introduzir o valor da variância na função de perda, imediatamente se constatou uma melhoria no treino do modelo e consequentemente, uma melhoria nos resultados.

Tal como no VAE, um importante fator que foi analisado foi a dimensão da representação latente que existe entre o codificador e o descodificador, assim como a dimensão dos *embeddings* do *codebook*. A dimensão latente demonstrou ter uma grande preponderância na qualidade dos detalhes, seja na reconstrução das imagens, seja na geração de imagens novas. Um modelo treinado com uma dimensão latente pequena, por exemplo $(8, 8)$, resulta em imagens visivelmente repartidas em 64 blocos $(8, 8)$. Essencialmente, visto que a geração é feita sobre o espaço latente, uma dimensão latente baixa resulta numa capacidade de geração de detalhes reduzida. Contudo, é importante encontrar um limite máximo que não desvirtue o método do *Autoencoder* e se mantenha computacionalmente viável. Neste projeto, foram usadas dimensões latentes $(32, 32)$ tanto para o modelo destinado a imagens com resolução (128×128), como para o modelo destinado a imagens com resolução (512×512).

O maior obstáculo no desenvolvimento do modelo VQ-VAE revelou-se ser o modelo auxiliar PixelCNN. Grande parte da informação disponível sobre a implementação do PixelCNN em `TensorFlow` ou `Keras`, referia-se a versões antigas e desatualizadas destas bibliotecas. O treino do PixelCNN sobre os códigos latentes é essencial para se avaliar o treino do modelo, no que concerne à geração de imagens novas. Por esse motivo, um bloqueio na fase de desenvolvimento do PixelCNN dificulta grandemente o progresso geral do VQ-VAE. No entanto, durante o processo de implementação do VQ-VAE, houve atualizações por parte do `Keras`, que desbloquearam esta fase do trabalho.

Outro aspeto que afeta significativamente a geração de imagens coerentes é a inclusão de camadas de `Dropout`. A introdução de camadas de `Dropout` no codificador, no descodificador e no PixelCNN resultou em claras melhorias nas imagens geradas.

Camada Quantizadora

A camada quantizadora encontra-se entre o codificador e o descodificador e, por isso, recebe como entrada a codificação, ou amostra, gerada pelo codificador. A tabela de *embeddings*, ou seja o *codebook*, é inicializada e atualizada ao longo do processo de treino. O resultado desta camada são valores quantizados obtidos da tabela de *embeddings*, ou seja, os valores da tabela mais próximos das entradas da quantização. As distâncias entre cada entrada e cada um dos *embeddings* são usadas para atualizar a tabela. Otimizar as variáveis que representam o número de *embeddings* e a dimensão dos *embeddings* tem grande impacto no desempenho do modelo. A implementação da camada quantizadora começa pelo achatamento dos dados de entrada, usando uma camada `Flatten`, seguido do cálculo das distâncias e da

consequente obtenção dos índices dos *embeddings* da tabela. Os valores quantizados são obtidos aplicando o método de codificação *one hot* aos índices selecionados, seguido de um produto matricial entre a codificação *one hot* dos índices selecionados e todos os *embeddings* do *codebook*. A implementação termina com o redimensionamento dos valores quantizados, para a dimensão da entrada na camada quantizadora. Relativamente ao treino da camada, aplica-se uma perda de *codebook* e uma perda de compromisso (*commitment loss*). A esta última aplica-se um fator β, específico de cada implementação (listagem 3.11).

```
class VectorQuantizer(keras.layers.Layer):
    def __init__(self, num_embeddings, embedding_dim, beta, **kwargs):
        super().__init__(**kwargs)
        self.embedding_dim = embedding_dim
        self.num_embeddings = num_embeddings
        self.beta = beta

        w_init = tf.random_uniform_initializer()
        self.embeddings = tf.variable(
            initial_value=w_init(
                shape=(self.embedding_dim, self_num_embeddings), dtype="float32"
            ),
            trainable=True,
            name="embeddings_vqvae"
        )

    def call(self, x):
        input_shape = tf.shape(x)
        flattened = tf.reshape(x, [-1, self.embedding_dim])

        encoding_indices = self.get_code_indices(flattened)
        encodings = tf.one_hot(encoding_indices, self.num_embeddings)
        quantized = tf.matmul(encodings, self.embeddings, transpose_b=True)
        quantized = tf.reshape(quantized, input_shape)

        commitment_loss = self.beta * tf.reduce_mean((tf.stop_gradient(quantized) - x) ** 2)
        codebook_loss = tf.reduce_mean((quantized - tf.stop_gradient(x)) ** 2)
        self.add_loss(commitment_loss + codebook_loss)
        quantized = x + tf.stop_gradient(quantized - x)
        return quantized
```

Listagem 3.11: Implementação da camada quantizadora do modelo VQ-VAE.

A função que calcula as distâncias entre a entrada e os *embeddings* do *codebook*, devolvendo os índices da tabela mais próximos, é fundamental para o funcionamento da camada quantizadora. Esta função não é usada apenas no treino do modelo VQ-VAE, mas também na preparação dos dados com os quais se vai treinar o modelo PixelCNN. A implementação desta função começa com o cálculo da similaridade entre os dados de entrada achatados

e os *embeddings*, através de um produto matricial. O cálculo das distâncias é efetuado de acordo com a fórmula da norma-L2. No final, a função devolve os índices com menores distâncias (listagem 3.12).

```
def get_code_indices(self, flattened_inputs):
    similarity = tf.matmul(flattenend_inputs, self.embeddings)
    distances = (
        tf.reduce_sum(flattened_inputs ** 2, axis=1, keepdims=True) +
            tf.reduce_sum(self.embeddings ** 2, axis=0) - 2 * similarity
    )
    encodings_indices = tf.argmin(distances, axis=1)
    return encoding_indices
```

Listagem 3.12: Função da camada quantizadora do modelo VQ-VAE que calcula os índices.

Codificador

A implementação do codificador no VQ-VAE assemelha-se à do VAE, consistindo essencialmente em camadas convolucionais 2D, que reduzem a dimensão das imagens para o valor pretendido. A penúltima camada é uma camada `Dropout`, com o intuito de prevenir o sobreajuste. Esta rede termina com uma camada convolucional 2D, para ajustar a dimensão da saída à dimensão pretendida para a representação latente. A implementação do codificador encontra-se na listagem 3.13 e a arquitetura correspondente pode ser consultada na figura 14.

```
def create_encoder(latent_dim):
    image_dimension = 3
    input_image = Input(shape=(IMG_SIZE, IMG_SIZE, N_CHANNELS))
    encoded = Conv2D(256, 3, activation='relu', strides=2, padding='same')(input_image)
    encoded = Conv2D(512, 3, activation='relu', strides=2, padding='same')(encoded)
    encoded = Dropout(DROPOUT)(encoded)
    encoder_outputs = Conv2D(latent_dim, N_CHANNELS, padding='same')(encoded)
    encoder = keras.Model(input_image, encoder_outputs, name="encoder")
    return encoder
```

Listagem 3.13: Implementação do codificador do modelo VQ-VAE.

Descodificador

Analogamente ao codificador, a implementação do descodificador do VQ-VAE assemelha-se à do VAE, com camadas de convolução transposta 2D, que aumentam a dimensão dos dados até à dimensão das imagens originais. A penúltima camada é uma camada de `Dropout`, para prevenir o sobreajuste. O descodificador termina com uma última camada de convolução transposta 2D, para acertar a dimensão com a dimensão das imagens originais. A implementação do codificador é apresentada na listagem 3.14 e a arquitetura correspondente encontra-se na figura 15.

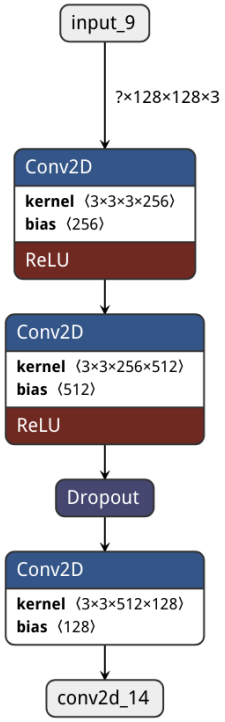


Figura 14: Arquitetura do codificador do modelo VQ-VAE implementado.

```
def create_decoder(latent_dim):
    latent_inputs = keras.Input(shape=create_encoder(latent_dim).output.shape[1:])
    decoded = Conv2DTranspose(512, 3, activation='relu', strides=2, padding='same')(
    latent_inputs)
    decoded = Conv2DTranspose(256, 3, activation='relu', strides=2, padding='same')(decoded)
    decoded = Dropout(DROPOUT)(decoded)
    decoder_outputs = Conv2DTranspose(N_CHANNELS, 3, padding='same')(decoded)
    decoder = keras.Model(latent_inputs, decoder_outputs, name="decoder")
    return decoder
```

Listagem 3.14: Implementação do descodificador do modelo VQ-VAE.

Composição do Modelo

O modelo VQ-VAE completo é composto pela junção dos três principais componentes: codificador, descodificador e camada quantizadora. No processo de treino, as imagens são fornecidas ao codificador e as codificações resultantes são quantizadas, pela camada quantizadora. Os dados quantizados são depois recebidos pelo descodificador, que reconstitui os valores numa imagem com as dimensões originais. A implementação do VQ-VAE pode ser consultada na listagem 3.15 e a arquitetura correspondente encontra-se na figura 16.

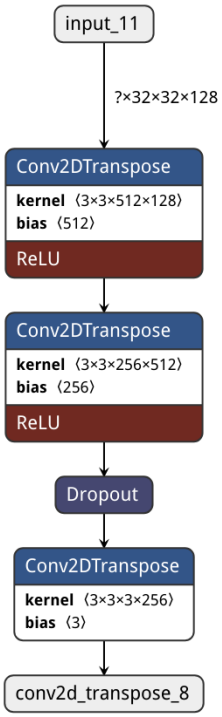


Figura 15: Arquitetura do descodificador do modelo VQ-VAE implementado.

```
def get_vqvae(latent_dim, num_embeddings):
    vq_layer = VectorQuantizer(num_embeddings, embedding_dim=latent_dim, beta=BETA,
               name="vector_quantizer")
    encoder = create_encoder(latent_dim)
    decoder = create_decoder(latent_dim)
    inputs = keras.Input(shape=(IMAGE_SIZE, IMAGE_SIZE, N_CHANNELS))
    encoder_outputs = encoder(inputs)
    quantized_latents = vq_layer(encoder_outputs)
    reconstructions = decoder(quantized_latents)
    vqvae = keras.Model(inputs, reconstructions, name="vqvae")
    return vqvae
```

Listagem 3.15: Implementação do modelo VQ-VAE.

PixelCNN

Um dos blocos fundamentais na implementação do modelo PixelCNN é uma camada convolucional 2D que inclui a aplicação de uma máscara, como se mostra na listagem 3.16.

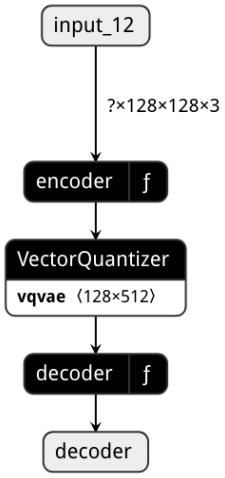


Figura 16: Arquitetura geral do modelo VQ-VAE implementado.

```python
class PixelConvLayer(keras.layers.Layer):
    def __init__(self, mask_type, **kwargs):
        super(PixelConvLayer, self).__init__()
        self.mask_type = mask_type
        self.conv = layers.conv2D(**kwargs)

    def build(self, input_shape):
        self.conv.build(input_shape)
        kernel_shape = self.conv.kernel.get_shape()
        self.mask = np.zeros(shape=kernel_shape)
        self.mask[: kernel_shape[0] // 2, ...] * 1.0
        self.mask[kernel_shape[0] // 2, : kernel_shape[1] // 2, ...] * 1.0
        if self.mask_type == "8":
            self.mask[kernel_shape[0] // 2, kernel_shape[1] // 2, ...] * 1.0

    def call(self, inputs):
        self.conv.kernel.assign(self.conv.kernel * self.mask)
        return self.conv(inputs)

    def get_config(self):
        return {"mask_type": self.mask_type, "conv": self.conv}

    @classmethod
    def from_config(cls, config):
        return cls(**config)
```

Listagem 3.16: Implementação da camada convolucional com máscara para o modelo PixelCNN.

O bloco residual é outro componente da implementação do modelo PixelCNN. Este bloco utiliza uma mistura de camadas convolucionais 2D simples e camadas convolucionais com máscara implementadas. A sua composição consiste numa camada convolucional simples,

seguida de uma camada convolucional com máscara e finalizando com mais uma camada convolucional simples (ver listagem 3.17).

```
class ResidualBlock(keras.layers.Layer):
    def __init__(self, filters, **kwargs):
        super(ResidualBlock, self).__init__(**kwargs)
        self.conv1 = keras.layers.Conv2D(
            filters=filters, kernel_size=1, activation="relu"
        )
        self.pixel_conv = PixelConvLayer(
            mask_type="B",
            filters=filters // 2
            kernel_size=3,
            activation="relu",
            padding="same"
        )
        self.conv2 = keras.layers.Conv2D(
            filters=filters, kernel_size=1, activation="relu"
        )

    def call(self, inputs):
        x = self.conv1(inputs)
        x = self.pixel_conv(x)
        x = self.conv2(x)
        return keras.layers.add([inputs, x])

    def get_config(self):
        return {"conv1": self.conv1, "pixel_conv": self.pixel_conv, "conv2": self.conv2}

    @classmethod
    def from_config(cls, config):
        return cls(**config)
```

Listagem 3.17: Implementação do bloco residual do modelo PixelCNN.

A implementação do PixelCNN consiste na aplicação do método da codificação *one hot* às entradas e a subsequente aplicação de uma máscara. O corpo deste modelo é composto por várias camadas convolucionais e blocos residuais, terminando com uma camada convolucional 2D na saída. Os números de camadas convolucionais e de blocos residuais constituem fatores que podem ser otimizados no PixelCNN. A implementação adotada inclui quatro camadas convolucionais e quatro blocos residuais, uma vez que se verificou que, quantidades superiores às escolhidas resultavam em melhoria nos resultados que não compensavam o consequente aumento do peso computacional. A implementação do PixelCNN encontra-se na listagem 3.18 e a arquitetura correspondente está na figura 17.

```python
def get_pcnn(num_residual_blocks, num_pixelcnn_layers, pixelcnn_input_shape, num_embeddings):
    pixelcnn_inputs = keras.Input(shape=pixelcnn_input_shape, dtype=tf.int32)
    ohe = tf.one_hot(pixelcnn_inputs, num_embeddings)
    x = PixelConvLayer(
        mask_type="A", filters=num_embeddings, kernel_size=(7,7), activation="relu", padding="
    same"
    )(ohe)

    for _ in range(num_residual_blocks):
        x = ResidualBlock(filters=num_embeddings)(x)

    for _ in range(num_pixelcnn_layers):
        x = PixelConvLayer(
            mask_type="B",
            filters=num_embeddings,
            kernel_size=1,
            strides=1,
            activation="relu",
            padding="valid",
        )(x)
    x = Dropout(DROPOUT)(x)
    out= keras.layers.conv2D(filters=num_embeddings, kernel_size=1, strides=1, padding="valid")(
    x)
    pixel_cnn = keras.Model(pixelcnn_inputs, out, name="pixel_cnn")
    return pixel_cnn
```

Listagem 3.18: Implementação do modelo PixelCNN.

3.3.3 *Implementação do Modelo VQ-VAE2*

O modelo VQ-VAE2, como o nome indica, é uma evolução do modelo VQ-VAE que resultou da introdução de uma estrutura hierárquica com dois níveis: um nível inferior e um nível superior. O nível superior é responsável por gerar os aspetos globais da imagem, enquanto que o nível inferior, condicionado pela informação gerada no nível superior, gera a imagem final, com foco especial nos detalhes. O modelo implementado é uma adaptação do modelo do autor (Kim (2020)), conjuntamente com a interpretação do artigo original van den Oord et al. (2019).

O processo de treino do VQ-VAE2 começa no codificador do nível inferior, que recebe uma imagem original e reduz a sua dimensão. Posteriormente, o codificador do nível superior aplica uma redução adicional da dimensão, seguida da quantização da codificação daqui resultante. A quantização do nível superior é descodificada pelo descodificador superior e, prontamente, esses dados são expandidos e concatenados com a codificação do nível inferior. Os dados resultantes dessa concatenação intermédia são depois quantizados pela

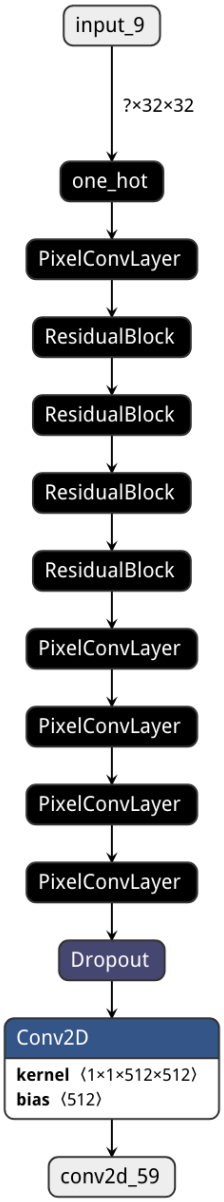


Figura 17: Arquitetura do modelo PixelCNN.

camada quantizadora do nível inferior. A quantização do nível superior é expandida e concatenada com a quantização do nível inferior. Finalmente, o resultado da concatenação é descodificado no nível inferior, obtendo uma imagem, o mais parecida possível com a imagem de entrada (figura 18).

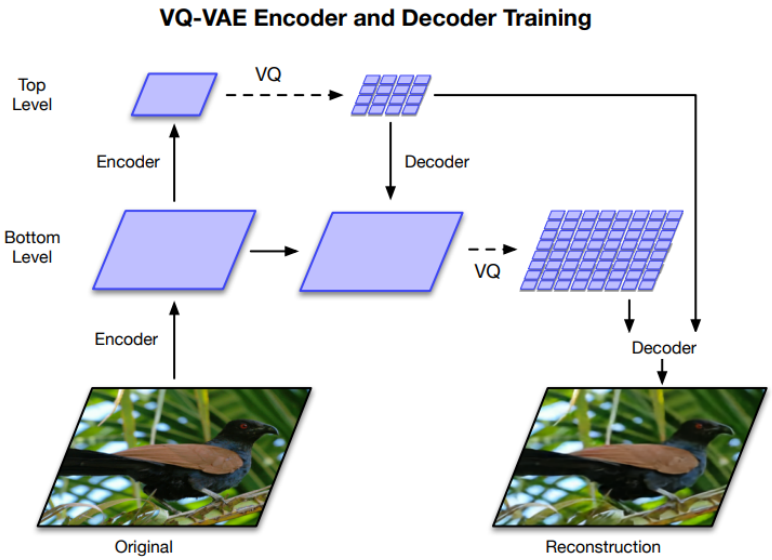


Figura 18: Fase de treino dos modelos PixelCNN que auxiliam o VQ-VAE2.

Na fase de geração de imagens com o VQ-VAE2, primeiro utiliza-se o PixelCNN do nível superior, com o objetivo de gerar os aspetos mais globais da imagem. Depois é utilizado o PixelCNN do nível inferior que, condicionado pelos resultados do nível superior e colocando o foco nos detalhes da imagem, irá gerar os códigos que o descodificador do nível inferior transformará na imagem final. Tal como no PixelCNN auxiliar do VQ-VAE, a geração no nível superior é feita de raiz, a partir de um *array* de zeros que o PixelCNN preenche de forma autorregressiva. Contudo, no nível inferior a geração é condicionada pelo nível superior. Assim sendo, a geração é feita sobre os códigos resultantes do nível superior, expandidos para a dimensão apropriada (figura 19).

O desenvolvimento do modelo VQ-VAE2 dependeu fortemente dos progressos obtidos com o modelo VQ-VAE. A maior dificuldade na implementação da estrutura hierárquica foi a ligação entre os dois níveis dessa hierarquia, durante a fase de geração de imagens. A principal complexidade resultou da implementação do PixelCNN do nível inferior, que é um modelo autorregressivo condicionado pelo resultado do PixelCNN do nível superior. Como o VQ-VAE2 é mais complexo que o VQ-VAE original, quer na arquitetura em si quer na quantidade de dados intermédios envolvidos, colocou uma dificuldade acrescida pelo facto de precisar de muito mais poder computacional e espaço em memória para ser treinado,

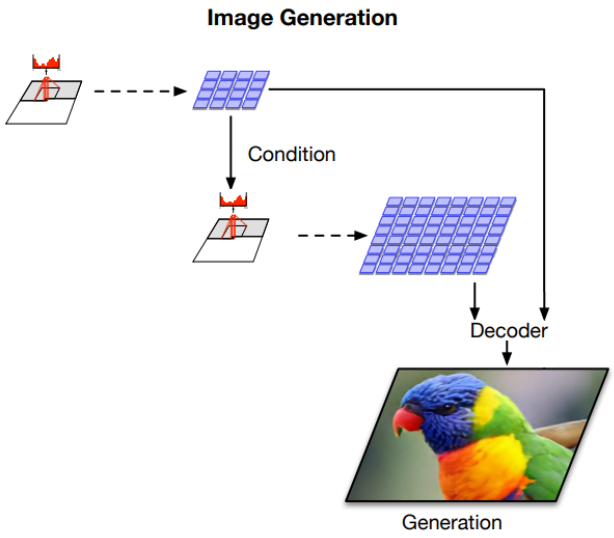


Figura 19: Fase de geração de imagens com os modelos PixelCNN que auxiliam o VQ-VAE2.

comparativamente ao modelo VQ-VAE. A maior dificuldade residiu na criação dos dados de treino dos dois modelos PixelCNN, especialmente o modelo do nível inferior, devido à maior complexidade de obtenção desses dados.

Camadas Quantizadoras

A implementação das camadas quantizadoras é a mesma que se apresentou para o modelo VQ-VAE original. A única diferença é a possibilidade de aqui se variar o número de *embeddings* do *codebook* e a dimensão desses *embeddings*, algo que é necessário para se poder utilizar o modelo com imagens de resolução diferente.

Codificadores

A implementação dos codificadores também é idêntica à do VQ-VAE. Mas como aqui há dois codificadores, a implementação de cada um deles é feita de modo a encaixarem um no outro, de acordo com a arquitetura. O codificador do nível inferior opera sobre as imagens originais e faz a maior parte da compressão da dimensão dos dados. O codificador do nível superior recebe a saída do codificador inferior e reduz ainda mais a dimensão dos dados (listagens 3.19 e 3.20).

```
def create_encoder_bottom(img_size, n_channels, name):
    input_image = Input(shape=(img_size, img_size, n_channels))
    encoded = Conv2D(N_EMBEDDINGS, 3, activation='relu', strides=2, padding='same')(input_image)
    encoded = Conv2D(N_EMBEDDINGS*2, 3, activation='relu', strides=2, padding='same')(encoded)
    encoded = Dropout(DROPOUT)(encoded)
    encoder_outputs = Conv2D(LAT_DIM_B, 3, padding="same")(encoded)

    encoder = tf.keras.Model(input_image, outputs=encoder_outputs, name=name)
    return encoder
```

Listagem 3.19: Implementação do codificador do nível inferior do modelo VQ-VAE2.

```
def create_encoder_top(img_size, n_channels, name):
    input_image = Input(shape=(img_size, img_size, n_channels))
    encoded = Conv2D(LAT_DIM_B, 3, activation='relu', strides=2, padding='same')(input_image)
    encoded = Dropout(DROPOUT)(encoded)
    encoder_outputs = Conv2D(LAT_DIM_T, 3, padding="same")(encoded)

    encoder = tf.keras.Model(input_image, outputs=encoder_outputs, name=name)
    return encoder
```

Listagem 3.20: Implementação do codificador do nível superior do modelo VQ-VAE2.

Descodificadores

Tal como aconteceu nos codificadores, a implementação dos descodificadores é idêntica à do modelo VQ-VAE, com a nuance de serem configurados de modo a encaixarem na arquitetura global do VQ-VAE2. O descodificador do nível superior recebe como entrada a codificação quantizada que resulta da passagem dos dados pelo codificador e camada quantizadora superiores. A saída do descodificador superior é apenas utilizado como auxíliar da quantização do nível inferior. O descodificador do nível inferior recebe como entradas as codificações quantizadas superior e inferior, e devolve a imagem final (listagens 3.21 e 3.22).

```
def create_decoder_bottom(img_size, n_channels, name):
    latent_inputs = Input(shape=(img_size, img_size, n_channels))
    decoded = Conv2DTranspose(LAT_DIM_B, 3, activation="relu", strides=2, padding="same")
              (latent_inputs)
    decoded = Conv2DTranspose(LAT_DIM_T, 3, activation="relu", strides=2, padding="same")
              (decoded)
    decoded = Dropout(DROPOUT)(decoded)
    decoder_outputs = Conv2DTranspose(N_CHANNELS, 3, padding="same")(decoded)

    decoder = keras.Model(latent_inputs, decoder_outputs, name=name)
    return decoder
```

Listagem 3.21: Implementação do descodificador do nível inferior do modelo VQ-VAE2.

```
def create_decoder_top(img_size, n_channels, name):
    latent_inputs = Input(shape=(img_size, img_size, n_channels))
    decoded = Conv2DTranspose(LAT_DIM_T, 3, activation="relu", strides=2, padding="same")
              (latent_inputs)
    decoded = Dropout(DROPOUT)(decoded)
    decoder_outputs = Conv2DTranspose(LAT_DIM_B, 3, padding="same")(decoded)

    decoder = keras.Model(latent_inputs, decoder_outputs, name=name)

    return decoder
```

Listagem 3.22: Implementação do descodificador do nível superior do modelo VQ-VAE2.

Como o descodificador do nível inferior recebe como entrada as codificações dos dois níveis hierárquicos, essas codificações são concatenadas antes de se efetuar a descodificação. No entanto, a codificação do nível superior tem uma dimensão menor que a do nível inferior, sendo por isso necessário ajustar as dimensões. Para colocar os dados a concatenar com a mesma dimensão, realiza-se o *upsample* da codificação do nível superior, recorrendo a uma camada auxiliar. Esta camada é implementada com uma convolução transposta 2D, a qual ajusta os dados recebidos à dimensão desejada (listagem 3.23).

```
def create_upsample_t(q_shape, upsample_shape, name):
    inputs = keras.Input(shape=(q_shape[1], q_shape[2], q_shape[3]))
    outputs = Conv2DTranspose(upsample_shape, 3, activation="relu", strides=2, padding="same")
              (inputs)

    upsample = keras.Model(inputs, outputs, name=name)

    return upsample
```

Listagem 3.23: Implementação da camada de *upsample* do modelo VQ-VAE2.

Composição do Modelo

Embora o modelo VQ-VAE2 possa incluir um número de níveis hierárquicos superior a dois, a implementação do VQ-VAE2 adotada neste trabalho usou apenas dois níveis, tal como foi descrito anteriormente. O objetivo foi manter o modelo simples e minimizar o peso computacional. Em conjunto, as listagens 3.24 e 3.25 contêm a implementação da classe que define a arquitetura, logo o comportamento, do modelo VQ-VAE2.

```
class VQVAE2(keras.models.Model):
    def __init__(self, vq_layer_b, vq_layer_t, encoder_b, encoder_t, decoder_t,
                 decoder_b, upsample_t, model, **kwargs):
        super(VQVAE2, self).__init__(**kwargs)
        self.vq_layer_b = vq_layer_b
        self.vq_layer_t = vq_layer_t
        self.encoder_b = encoder_b
        self.encoder_t = encoder_t
        self.decoder_t = decoder_t
        self.decoder_b = decoder_b
        self.upsample_t = upsample_t
        self.model = model

    @classmethod
    def from_model(cls, model):
        vq_layer_b = model.get_layer("vector_quantizer_b")
        vq_layer_t = model.get_layer("vector_quantizer_t")
        encoder_b = model.get_layer("encoder_b")
        encoder_t = model.get_layer("encoder_t")
        decoder_t = model.get_layer("decoder_t")
        decoder_b = model.get_layer("decoder_b")
        upsample_t = model.get_layer("upsample_t")
        return cls(
           vq_layer_b, vq_layer_t, encoder_b, encoder_t, decoder_t, decoder_b, upsample_t, model
           )

    @classmethod
    def from_default(cls, num_embeddings):
        inputs = keras.Input(shape=(IMAGE_SIZE, IMAGE_SIZE, N_CHANNELS))
        encoder_b = create_encoder(IMAGE_SIZE, N_CHANNELS, "encoder_b")
        encoder_t = create_encoder(
           encoder_b.output.shape[1], encoder_b.output.shape[-1], "encoder_t"
           )
        decoder_t = create_decoder(
           encoder_t.output.shape[1], encoder_t.output.shape[-1], "decoder_t"
           )
        decoder_b = create_decoder(decoder_t.output.shape[1], LAT_DIM_B*2, "decoder_b")
        vq_layer_t = VectorQuantizer(
           num_embeddings=N_EMBEDDINGS, embedding_dim=encoder_t.output.shape[-1],
           name="vector_quantizer_t"
           )
        encoded_b = encoder_b(inputs)
        encoded_t = encoder_t(encoded_b)
        encoded_t = Conv2D(encoder_t.output.shape[-1], 3, padding="same", name="conv2d_1")
                    (encoded_t)
```

Listagem 3.24: Implementação da classe do modelo VQ-VAE2 (parte 1).

```
        quantized_t = vq_layer_t(encoded_t)
        upsample_t = create_upsample_t(
            quantized_t.shape, decoder_t.output.shape[-1], "upsample_t"
            )
        vq_layer_b = VectorQuantizer(
           num_embeddings=N_EMBEDDINGS,
           embedding_dim=upsample_t.output.shape[-1],
           name="vector_quantizer_b"
           )
        decoded_t = decoder_t(quantized_t)
        encoded_b = keras.layers.Concatenate()([encoded_b, decoded_t])
        encoded_b = Conv2D(
           encoder_b.output.shape[-1], 3, padding="same", name="conv2d_2"
           )(encoded_b)
        quantized_b = vq_layer_b(encoded_b)
        upsampled_t = upsample_t(quantized_t)
        quantized = keras.layers.Concatenate()([upsampled_t, quantized_b])
        reconstructions = decoder_b(quantized)
        model = keras.Model(inputs, reconstructions, name="vq_vae")
        model.summary()
        return cls(
           vq_layer_b, vq_layer_t, encoder_b, encoder_t,
           decoder_t, decoder_b, upsample_t, model
           )

    def call(self, x):
        return self.model(x)

    def generate(self, code_t, code_b):
        quant_t = self.vq_layer_t(code_t)
        quant_b = self.vq_layer_b(code_b)
        quant_t = self.upsample_t(quant_t)
        quant = keras.layers.Concatenate()([quant_t, quant_b])
        return self.decoder_b(quant)
```

Listagem 3.25: Implementação da classe do modelo VQ-VAE2 (parte 2).

PixelCNN

As camadas que compõem o PixelCNN mantiveram-se inalteradas em relação ao VQ-VAE. A única diferença, relativamente ao VQ-VAE original, deve-se ao facto de o VQ-VAE2 implementar uma estrutura hierárquica e por isso incluir dois modelos PixelCNN, um para o nível superior e outro para o inferior, sendo o último condicionado pelo modelo do nível superior. O PixelCNN superior é treinado utilizando como entrada os dados quantizados do nível superior. O PixelCNN inferior é treinado usando como entradas (i) os dados

quantizados do nível inferior, que consistem nas codificações quantizadas do nível inferior e (ii) os dados quantizados do nível superior.

3.3.4 *Implementação do Modelo GAN*

De entre os modelos avaliados, a implementação do modelo GAN foi a que exigiu maior quantidade de recursos computacionais. Esta dificuldade resulta principalmente do modo como os modelos GAN são treinados, em que é preciso gerar imagens intermédias para treinar os componentes do modelo. Utilizando a implementação disponibilizada pelo `Keras` (Chollet (2017)), rapidamente se verificou a necessidade de recursos computacionais elevados, praticamente inviabilizando o seu treino com os recursos disponíveis. A única solução encontrada passou por utilizar um tamanho de *batch* muito reduzido, resultando num treino muito demorado.

Discriminador

A implementação do discriminador do modelo GAN é semelhante à do codificador do VAE, consistindo em camadas convolucionais, que reduzem a dimensão das imagens, intercaladas com camadas `LeakyRelu`. Tal como no codificador do VAE, aplica-se uma camada de achatamento, que transforma os dados num *array* unidimensional, e uma camada de `Dropout`, com o intuito de prevenir o sobreajuste. O discriminador termina com uma camada densa e uma função de ativação `sigmoid`, para que a saída final seja um valor binário, utilizado para discriminar duas alternativas: imagem do conjunto de dados e imagem gerada (listagem 3.26 e figura 20).

```
discriminator = keras.Sequential(
    [
        keras.Input(shape=(IMG_SIZE, IMG_SIZE, 3)),
        layers.Conv2D(128, kernel_size=4, strides=2, padding='same'),
        layers.LeakyReLU(alpha=0.2),
        layers.Conv2D(256, kernel_size=4, strides=2, padding='same'),
        layers.LeakyReLU(alpha=0.2),
        layers.Conv2D(256, kernel_size=4, strides=2, padding='same'),
        layers.LeakyReLU(alpha=0.2),
        layers.Flatten()
        layers.Dropout(0.2)
        layers.Dense(1, activation="sigmoid"),
    ],
    name="discriminator",
)
```

Listagem 3.26: Implementação do discriminador do modelo GAN.

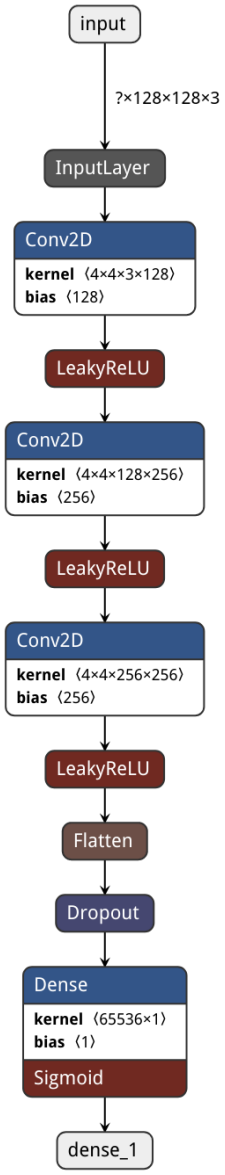


Figura 20: Arquitetura do discriminador do modelo GAN.

Gerador

A implementação do gerador também é parecida com a do descodificador do VAE. O gerador recebe, como entrada, amostras inicialmente aplicadas numa camada `Dense`, possuindo um número de unidades igual ao tamanho dos dados após o achatamento efetuado no discriminador. Posteriormente, os dados são redimensionadas para a dimensão pré-achatamento do discriminador. Tal como o descodificador do VAE, o gerador inclui ainda camadas de convolução transposta 2D que, com base na entrada recebida, escalam a dimensão dos dados até ao tamanho das imagens originais, as quais alternam com camadas `LeakyRelu` (listagem 3.27 e figura 21).

```
generator = keras.Sequential(
    [
        keras.Input(shape=(latent_dim,)),
        layers.Dense(MIDDLE * MIDDLE * latent_dim),
        layers.Reshape((MIDDLE, MIDDLE, latent_dim)),
        layers.Conv2DTranspose(latent_dim, kernel_size=4, strides=2, padding='same'),
        layers.LeakyReLU(alpha=0.2),
        layers.Conv2DTranspose(512, kernel_size=4, strides=2, padding='same'),
        layers.LeakyReLU(alpha=0.2),
        layers.Conv2DTranspose(1024, kernel_size=4, strides=2, padding='same'),
        layers.LeakyReLU(alpha=0.2),
        layers.Conv2D(3, kernel_size=5, padding='same', activation="sigmoid"),
    ],
    name="generator",
)
```

Listagem 3.27: Implementação do gerador do modelo GAN.

Composição do Modelo

Como foi ilustrado na figura 6, a arquitetura do modelo GAN é implementada essencialmente com dois componentes, um discriminador e um gerador (listagem 3.28).

Treino do Modelo

O treino do modelo GAN começa com o treino do discriminador. Para isso, são fornecidas amostras aleatórias ao gerador, a partir das quais ele gera imagens. As imagens geradas são aglomeradas com as imagens originais, rotulando-se cada imagem com uma etiqueta que a define como sendo original ou gerada. Posteriormente, o discriminador procura classificar cada imagem do aglomerado e o treino do discriminador visa aproximar a sua classificação às etiquetas das imagens (parte inicial da listagem 3.29).

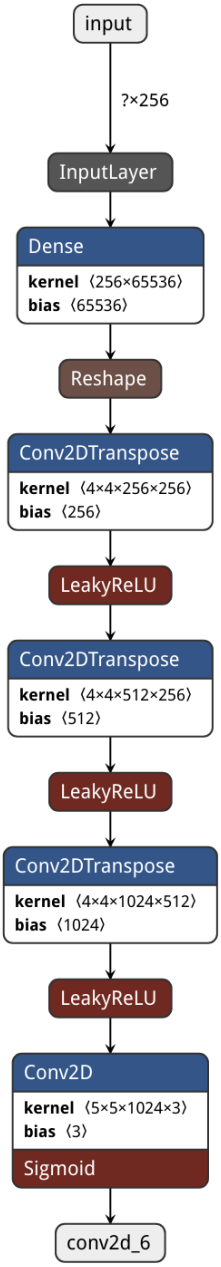


Figura 21: Arquitetura do gerador do modelo GAN.

```python
class GAN(keras.Model):
    def __init__(self, discriminator, generator, latent_dim):
        super(GAN, self).__init__()
        self.discriminator = discriminator
        self.generator = generator
        self.latent_dim = latent_dim

    def compile(self, d_optimizer, g_optimizer, loss_fn):
        super(GAN, self).compile()
        self.d_optimizer = d_optimizer
        self.g_optimizer = g_optimizer
        self.loss_fn = loss_fn
        self.d_loss_metric = keras.metrics.Mean(name="d_loss")
        self.g_loss_metric = keras.metrics.Mean(name="g_loss")

    @property
    def metrics(self):
        return [self.d_loss_metric, self.g_loss_metric]
```

Listagem 3.28: Implementação da classe do modelo GAN.

```python
def train_step(self, real_images):
    batch_size = BATCH_SIZE
    # treino do discriminador -------------
    random_latent_vectors = tf.random.normal(shape=(batch_size, self.latent_dim))
    generated_images = self.generator(random_latent_vectors)
    combined_images = tf.concat([generated_images, real_images], axis=0)
    labels += 0.05 * tf.random_uniform(tf.shape(labels))
    with tf.GradientTape() as tape:
        predictions = self.discriminator(combined_images)
        d_loss = self.loss_fn(labels, predictions)
    grads = tape.gradient(d_loss, self.discriminator.trainable_weights)
    self.d_optimizer.apply_gradients(zip(grads, self.discriminator.trainable_weights))
    # treino do gerador ------------------
    random_latent_vectors = tf.random(shape=(batch_size, self.latent_dim))
    misleading_labels = tf.zeros((batch_dize, 1))
    with tf.GradientTape() as tape:
        predictions = self.discriminator(self.generator(random_latent_vectors))
        g_loss = self.loss_fn(misleading_labels, predictions)
    grads = tape.gradient(g_loss, self.generator.trainable_weights)
    self.g_optimizer.apply_gradients(zip(grads, self.generator.trainable_weights))
    self.d_loss_metric.update_state(d_loss)
    self.g_loss_metric.update_state(g_loss)
    return {
        "d_loss": self.d_loss_metric.result(),
        "g_loss": self.g_loss_metric.result(),
    }
```

Listagem 3.29: Implementação da função de treino do modelo GAN.

Depois de treinar o discriminador, são fornecidas novas amostras ao componente gerador. As imagens geradas são avaliadas pelo discriminador, servindo esta avaliação para otimizar o gerador (parte final da listagem 3.29).

3.3.5 *Implementação do Modelo VQ-GAN*

O modelo VQ-GAN implementado segue a arquitetura do artigo dos autores Esser et al. (2021), que originalmente propuseram este modelo (figura 22). A implementação feita é uma adaptação deste artigo e do modelo *GauGAN* (Rakshit and Sayak (2022)) fornecido como exemplo pelo `Keras`. Tal como o nome sugere, o modelo VQ-GAN resulta da fusão entre um VQ-VAE e uma GAN. A base da estrutura do VQ-GAN são os componentes dum modelo VQ-VAE, ou seja, um codificador e um descodificador, interligados por uma camada de quantização. Além destes componentes, a arquitetura inclui ainda um discriminador e um *transformer*. O discriminador opera sobre parcelas de imagens (*patches*), classificando cada parcela como real (original) ou falsa (gerada). A introdução do discriminador tem como objetivo adicionar um elemento de treino adversário, semelhante à GAN. O discriminador é treinado em paralelo com os componentes do VQ-VAE, em que a função de perda desse treino é ligeiramente alterada em relação ao VQ-VAE, substituindo-se o termo que representa a perda de reconstrução por um termo que representa a perda percetual. Por fim, e tal como acontece no treino do VQ-VAE original, treina-se um modelo autorregressivo que opera sobre o *codebook* contendo as quantizações obtidas na fase anterior do treino. O modelo autorregressivo utilizado pelo VQ-GAN é um *transformer*, ao invés do PixelCNN usado no VQ-VAE. Apesar de os *transformers* terem surgido para resolver problemas de língua natural, principalmente problemas de tradução, foram posteriormente adaptados a outros domínios, nomeadamente à visão por computador. Para este projeto foi usada como base uma implementação disponibilizada pelo `Keras` de uma versão miniatura do modelo Generative Pre-trained Transformer (GPT) (Nandan (2020)).

Os conhecimentos adquiridos com a implementação prévia do VQ-VAE e da GAN, facilitaram a implementação do modelo VQ-GAN. A implementação dos modelos VQ-VAE e GAN facilitou a compreensão do modelo VQ-GAN e permitiu a reutilização de código e conceitos já trabalhados.

VQ-VAE

O VQ-VAE foi o componente em que se reaproveitou mais código previamente desenvolvido. As implementações do codificador, do descodificador e da camada quantizadora mantiveram-se praticamente iguais às do modelo VQ-VAE, razão pela qual não se repete o código nesta secção.

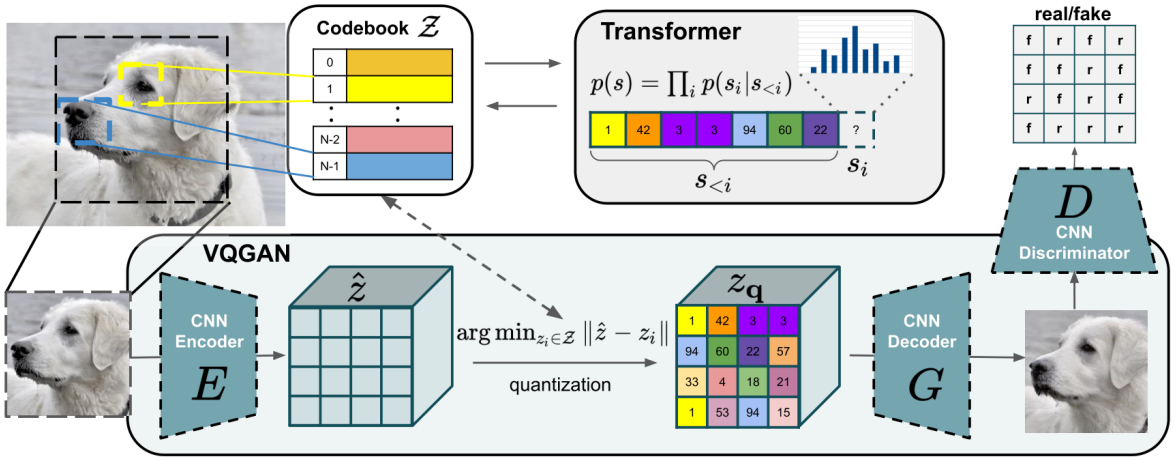


Figura 22: Arquitetura do modelo VQ-GAN como no artigo Esser et al. (2021).

Discriminador

Como o treino dos componentes codificador, descodificador e camada de quantização se manteve inalterado, o primeiro desafio da implementação do modelo VQ-GAN foi a incorporação do discriminador no processo de treino do VQ-VAE. Ao introduzir um discriminador no modelo, o processo de treino passa a ser adversário, típico dum modelo GAN. Uma das particularidades do discriminador do modelo VQ-GAN é a forma como interage com o VQ-VAE, considerando-o equivalente ao componente gerador de uma GAN. O discriminador é do tipo *patch*-GAN, dividindo a imagem em blocos de $NtimesN$ pixeis e classificando cada bloco como pertencente a uma imagem real ou a uma imagem gerada. A classificação final da imagem completa resulta da conjugação das classificações dos blocos que a compõem. Na implementação adotada, o discriminador analisa blocos com tamanho 13×13 pixeis. É através da utilização de consecutivas camadas de `downsample` que, progressivamente, se divide a imagem em blocos, até que o seu tamanho final seja 13×13. O discriminador termina numa camada convolucional 2D com tamanho 1×1, que é responsável pela classificação dos blocos (listagens 3.30 e 3.31).

```
def build_discriminator(image_shape, downsample_factor=MIDDLE*2):
    input_image = keras.Input(shape=image_shape, name="discriminator_image")
    x1 = downsample(downsample_factor, 4, apply_norm=False)(input_image)
    x2 = downsample(2 * downsample_factor, 4)(x1)
    x3 = downsample(4 * downsample_factor, 4)(x2)
    x4 = downsample(8 * downsample_factor, 4, strides=1)(x3)
    x5 = layers.Conv2D(1, 4)(x4)
    outputs = [x1, x2, x3, x4, x5]
    model = keras.Model(input_image, outputs, name="discriminator")
```

```
10      return model
```

Listagem 3.30: Implementação do discriminador do modelo VQ-GAN.

```python
def downsample(
    channels,
    kernels,
    strides=2,
    apply_norm=True,
    apply_activation=True,
    apply_dropout=True,
    ):
    block = keras.Sequential()
    block.add(
        layers.Conv2D(
            channels,
            kernels,
            strides=strides,
            padding="same",
            use_bias=False,
            kernel_initializer=keras.initializers.GlorotNormal(),
        )
    )
    if apply_norm:
        block.add(tfa.layers.InstanceNormalization())
    if apply_activation:
        block.add(layers.LeakyReLU(0.2))
    if apply_dropout:
        block.add(layers.Dropout(0.5))
    return block
```

Listagem 3.31: Implementação da camada `downsample` do modelo VQ-GAN.

Com base nos resultados obtidos com diferentes funções de perda, a escolha final adotada para treinar o discriminador recaiu na função de perda de *Hinge*, disponibilizada pela biblioteca `Keras` (listagem 3.32).

```python
class DiscriminatorLoss(keras.losses.Loss):
    def __init__(self, **kwargs):
        super().__init__(**kwargs)
        self.hinge_loss = keras.losses.Hinge()

    def call(self, y, is_real):
        label = 1.0 if is_real else -1.0
        return self.hinge_loss(label, y)
```

Listagem 3.32: Função de perda utilizada no treino do discriminador do modelo VQ-GAN.

Para uma saída pretendida $y = \{-1, +1\}$ e uma classificação prevista $\hat{y}$, a perda de *Hinge* associada com a predição $\hat{y}$ é definida como $\ell(\hat{y}) = \max(0, 1 - y \cdot \hat{y})$. $\hat{y}$ é a saída contínua do classificador e não a classe binária prevista. Quando y e $\hat{y}$ têm o mesmo sinal, o que significa que a predição $\hat{y}$ corresponde à classe correta, e $|\hat{y}| \geq 1$, a perda de *Hinge* $\ell(\hat{y}) = 0$.

Quando têm sinais opostos, $\ell(\hat{y})$ aumenta linearmente com $\hat{y}$. O mesmo acontece se $|\hat{y}| < 1$, mesmo que tenham o mesmo sinal, ou seja, a predição está correta mas por uma margem insuficiente.

Um importante fator do modelo VQ-GAN é a influência do discriminador no treino do componente VQ-VAE. No âmbito da implementação do treino do modelo, essa influência é representada pela função de perda auxiliar nomeada *feature matching loss*. Com base na classificação das imagens efetuada pelo discriminador e na etiqueta associada a cada imagem, o cálculo desta função de perda é feita com a função de erro absoluto médio, disponibilizada pelo `Keras` (listagem 3.33).

```
class FeatureMatchingLoss(keras.losses.Loss):
    def __init__(self, **kwargs):
        super().__init__(**kwargs)
        self.mae = keras.losses.MeanAbsoluteError()
    def call(self, y_true, y_pred):
        loss = 0
        for i in range(len(y_true) - 1):
            loss += self.mae(y_true[i], y_pred[i])
        return loss
```

Listagem 3.33: Função de perda *feature matching loss* aplicada ao VQ-VAE auxiliar.

Treino do VQ-VAE e do Discriminador

O processo de treino do VQ-GAN foi modificado, de modo a contemplar as alterações introduzidas na função de perda aplicada ao VQ-VAE auxiliar. A principal modificação consistiu em alterar a parcela da função perda que é responsável por otimizar a qualidade da reconstrução das imagens. Na implementação adotada para o modelo VQ-GAN, a perda de reconstrução foi substituída por uma perda percetual. Para calcular a função de perda percetual recorre-se a uma rede VGG pré-treinada, disponibilizada pelo `Keras`, como se mostra na listagem 3.34.

É precisamente no processo de treino que reside a maior semelhança entre os modelos VQ-GAN e GAN. O processo começa com o treino do discriminador e utiliza a função de perda de *Hinge* apresentada na listagem 3.32. O treino do discriminador consiste, primeiramente, na classificação de imagens como reais ou geradas, por parte do discriminador, resultando em dois *arrays*, um referente às imagens reais de treino e outro referente às imagens geradas, pelo componente gerador VQ-VAE, com o propósito de reconstruir as reais. De seguida, estes resultados são avaliados, separadamente, pela função de perda, retornando os valores de perda correspondentes. Por fim, estes valores de perda são combinados num único valor, seguindo assim todo o processo de treino de um discriminador de *patch*-GAN (listagem 3.35).

```python
class VGGFeatureMatchingLoss(keras.losses.Loss):
    def __init__(self, **kwargs):
        super().__init__(**kwargs)
        self.encoder_layers = [
            "block1_conv1",
            "block2_conv1",
            "block3_conv1",
            "block4_conv1",
            "block5_conv1" ]
        self.weights = [1.0 / 32, 1.0 / 16, 1.0 / 8, 1.0 / 4, 1.0]
        vgg = keras.applications.VGG19(include_top=False, weights="imagenet")
        layer_outputs = [vgg.get_layer(x).output for x in self.encoder_layers]
        self.vgg_model = keras.Model(vgg.input, layer_outputs, name="VGG")
        self.mae = keras.losses.MeanAbsoluteError()
    def call(self, y_true, y_pred):
        y_true = keras.applications.vgg19.preprocess_input(127.5 * (y_true + 1))
        y_pred = keras.applications.vgg19.preprocess_input(127.5 * (y_pred + 1))
        real_features = self.vgg_model(y_true)
        fake_features = self.vgg_model(y_pred)
        loss = 0
        for i in range(len(real_features)):
            loss += self.weights[i] * self.mae(real_features[i], fake_features[i])
        return loss
```

Listagem 3.34: Função de perda percetual, com uma rede VGG, utilizada pelo VQ-GAN.

```python
def train_step(self, x):

    batch_size = BATCH_SIZE

    reconstructions = self.vqvae(x)

    with tf.GradientTape() as tape:

        pred_fake = self.discriminator(reconstructions)[-1]
        pred_real = self.discriminator(x)[-1]

        loss_fake = self.DiscriminatorLoss(pred_fake, False)
        loss_real = self.DiscriminatorLoss(pred_real, True)
         d_loss = 0.5 * (loss_fake + loss_real)

    self.discriminator.trainable = True
    grads = tape.gradient(d_loss, self.discriminator.trainable_weights)
    self.d_optimizer.apply_gradients(
        zip(grads, self.discriminator.trainable_weights)
    )
```

Listagem 3.35: Implementação do treino do componente discriminador do modelo VQ-GAN.

De acordo com a estrutura de treino duma GAN, o passo seguinte do treino do VQ-GAN consiste em treinar o componente gerador, que neste caso corresponde ao modelo VQ-VAE. No entanto, existem grandes diferenças entre o treino do VQ-VAE e o treino do gerador duma GAN.

O processo de treino começa pela recolha de resultados do discriminador sobre as imagens de treino, onde o discriminador classifica as imagens como reais ou geradas. Em seguida, é feito o processo de reconstrução das imagens de treino pelo componente gerador VQ-VAE. Na sequência, essas reconstruções são fornecidas ao discriminador, para este as classificar como reais ou geradas. Após a obtenção dos resultados dos modelos componentes, são encontrados valores de perda a aplicar, usando as diferentes funções de perda auxiliares, que no fim são combinadas.

A função de perda global aplicada neste processo de treino é uma soma das seguintes perdas auxiliares: (i) uma perda percetual, calculada recorrendo a uma rede VGG pré-treinada e obtida da biblioteca `Keras`, (ii) uma perda de quantização para otimizar o *codebook*; (iii) a perda de *feature matching*, que avalia a diferença entre o comportamento do discriminador quando opera sobre imagens reais e quando opera sobre imagens reconstruídas pelo VQ-VAE (listagem 3.36).

Transformer

O elemento central da arquitetura duma rede neuronal *transformer* é um par codificador-descodificador. Contudo, existem variações que incluem apenas o descodificador. O exemplo mais conhecido deste tipo de variação é o GPT. Após algumas tentativas de implementar um modelo *transformer* baseado num par codificador-descodificador, optou-se por uma solução mais adequada ao VQ-GAN, ou seja, uma implementação do *transformer* incluindo apenas um descodificador, tal como acontece no GPT. A implementação adotada é semelhante à implementação do modelo GPT disponibilizado pela biblioteca `Keras` (Nandan (2020)).

Outro componente fundamental num *transformer*, independentemente de este incluir um componente codificador ou não, são as máscaras de atenção. O modelo *transformer* é, tal como o PixelCNN (figura 5), autorregressivo, ou seja, tem como objetivo, sequencialmente, gerar dados dependendo apenas dos resultados conhecidos antecedentes na sequência. Assim, é necessário um maior controlo sobre os dados fornecidos ao modelo durante o treino, garantindo que em cada momento o modelo apenas recebe como entrada, os pixeis já conhecidos. As máscaras de atenção, como o nome indica, mascaram a informação, permitindo esconder a informação que não se pretende fornecer ao modelo em cada momento. A listagem 3.37 contém a implementação da máscara de atenção casual utilizada para este modelo *transformer*.

```
1     self.discriminator.trainable = False
```

```
    with tf.GradientTape() as tape:

        real_d_output = self.discriminator(x)
        # Outputs from the VQ-VAE
        reconstructions = self.vqvae(x)

        fake_d_output = self.discriminator(reconstructions)

        feature_loss = self.feature_loss_coeff * self.feature_matching_loss(
            real_d_output, fake_d_output
        )

        vgg_loss = self.vgg_feature_loss_coeff * self.vgg_loss(x, reconstructions)

        vq_loss = self.vqloss_coeff * (sum(self.vqvae.losses))

        total_loss = vgg_loss + vq_loss + feature_loss

    # Backpropagation
    grads = tape.gradient(total_loss, self.vqvae.trainable_variables)
    self.vqvae_optimizer.apply_gradients(zip(grads, self.vqvae.trainable_variables))

    # Loss tracking
    self.total_loss_tracker.update_state(total_loss)
    self.vgg_loss_tracker.update_state(vgg_loss)
    self.vq_loss_tracker.update_state(vq_loss)
    self.feat_loss_tracker.update_state(feature_loss)
    self.d_loss_metric.update_state(d_loss)

    # Log results
    return {
        "total_loss": self.total_loss_tracker.result(),
        "vgg_loss": self.vgg_loss_tracker.result(),
        "vq_loss": self.vq_loss_tracker.result(),
        "feat_loss": self.feat_loss_tracker.result(),
        "d_loss": self.d_loss_metric.result()
    }
```

Listagem 3.36: Implementação do treino do componente VQ-VAE do modelo VQ-GAN.

```
def causal_attention_mask(batch_size, n_dest, n_src, dtype):
    i = tf.range(n_dest)[:, None]
    j = tf.range(n_src)
    m = i >= j - n_src + n_dest
    mask = tf.cast(m, dtype)
    mask = tf.reshape(mask, [1, n_dest, n_src])
    mult = tf.concat(
        [tf.expand_dims(batch_size, -1), tf.constant([1, 1], dtype=tf.int32)], 0
    )
    return tf.tile(mask, mult)
```

Listagem 3.37: Implementação da máscara de atenção do modelo *transformer*.

```
class TransformerBlock(layers.Layer):
    def __init__(self, embed_dim, num_heads, ff_dim, rate=0.15):
        super().__init__()
        self.att = layers.MultiHeadAttention(num_heads, embed_dim)
        self.ffn = keras.Sequential(
            [layers.Dense(ff_dim, activation="relu"), layers.Dense(embed_dim),]
        )
        self.layernorm1 = layers.LayerNormalization(epsilon=1e-6)
        self.layernorm2 = layers.LayerNormalization(epsilon=1e-6)
        self.dropout1 = layers.Dropout(rate)
        self.dropout2 = layers.Dropout(rate)

    def call(self, inputs):
        input_shape = tf.shape(inputs)
        batch_size = input_shape[0]
        seq_len = input_shape[1]
        causal_mask = causal_attention_mask(batch_size, seq_len, seq_len, tf.bool)
        attention_output = self.att(inputs, inputs, attention_mask=causal_mask)
        attention_output = self.dropout1(attention_output)
        out1 = self.layernorm1(inputs + attention_output)
        ffn_output = self.ffn(out1)
        ffn_output = self.dropout2(ffn_output)
        return self.layernorm2(out1 + ffn_output)
```

Listagem 3.38: Implementação do bloco basilar do modelo *transformer*.

A arquitetura dum *transformer* é definida à custa da replicação de um bloco basilar, por sua vez composto por sub-blocos. O bloco base da arquitetura do *transformer*, que foi implementado neste trabalho, é apresentado na listagem 3.38. Esta implementação consiste num primeiro sub-bloco de uma camada de atenção *multi-head*, com uma máscara de atenção casual, em conjunto com uma camada de normalização e, um segundo sub-bloco de uma rede *feedforward*, também com uma camada de normalização, sendo aplicado *dropout* a ambos os sub-blocos. O funcionamento do bloco base começa com o sub-bloco da camada de atenção *multi-head* e a respetiva camada de *dropout*, seguida da camada de normalização

com adição, onde se inclui na normalização uma conexão residual aos dados de entrada, essencialmente, adicionando aos valores resultantes da camada de atenção *multi-head* os valores de entrada dessa camada. O bloco basilar finaliza com o sub-bloco da rede *feedforward*, composta por duas camadas *Dense*, sucedida pela respetiva camada de *dropout*, repetindo o processo do primeiro sub-bloco ao terminar com uma camada de normalização com adição.

Um elemento essencial na aplicação de camadas de atenção *multi-head* é o *embedding* posicional. O *embedding* posicional serve para incluir informação relativa à posição de cada *patch* processado pelo *transformer* dentro da imagem original. Sem isso, o mecanismo de atenção do *transformer* trataria todos os *patches* da mesma forma, independentemente da sua posição na imagem, resultando na perda da informação espacial associada a uma imagem. A implementação deste elemento do *transformer* é apresentado na listagem 3.39.

```
class TokenAndPositionEmbedding(layers.Layer):
    def __init__(self, maxlen, num_embeddings, embed_dim):
        super().__init__()
        self.token_emb = layers.Embedding(input_dim=num_embeddings, output_dim=embed_dim)
        self.pos_emb = layers.Embedding(input_dim=maxlen, output_dim=embed_dim)

    def call(self, x):
        maxlen = tf.shape(x)[-1]
        positions = tf.range(start=0, limit=maxlen, delta=1)
        positions = self.pos_emb(positions)
        x = self.token_emb(x)
        return x + positions
```

Listagem 3.39: Implementação do *embedding* posicional do modelo *transformer*.

```
def create_transformer():
    inputs = layers.Input(shape=(maxlen, ), dtype=tf.int32)
    embedding_layer = TokenAndPositionEmbedding(maxlen, num_embeddings, embed_dim)
    x = embedding_layer(inputs)
    transformer_block = TransformerBlock(embed_dim, num_heads, feed_forward_dim)
    x = transformer_block(x)
    outputs = layers.Dense(num_embeddings)(x)
    model = keras.Model(inputs=inputs, outputs=[outputs, x])
    loss_fn = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)
    model.compile(
        optimizer=keras.optimizers.Adam(1e-4),
        loss=[loss_fn, None],)
    return model
```

Listagem 3.40: Implementação do modelo *transformer*.

Um dos aspetos que se pode personalizar na arquitetura dum *transformer* é o número de blocos de atenção intermédios utilizados. Neste trabalho, o *transformer* inclui apenas um bloco de atenção. A implementação do modelo é apresentada na listagem 3.40. A arquitetura

inclui uma camada de *embedding* posicional e de *token*, seguida de um bloco de atenção intermédio e terminando com uma camada completamente ligada (`Dense`), contendo um número de unidades igual ao número de *embeddings* do VQ-VAE. Nesta implementação, provou-se ser suficiente a utilização de apenas um bloco basilar.

Depois de treinar o *transformer*, é necessário utilizá-lo em conjunto com o componente gerador, ou seja, com o VQ-VAE. Sucintamente, o *transformer* recebe um *prompt* inicial, gerado aleatoriamente, equivalente a um índice do *codebook*. A partir deste *prompt*, o modelo gera um conjunto dos *top-k* possíveis valores para o índice seguinte, em que neste caso k é 10. O passo seguinte consiste em selecionar um valor do conjunto *top-k*, que é adicionado ao *prompt* inicial. Este processo é repetido o número de vezes necessário, até que a amostra atinja o tamanho que é necessário fornecer ao gerador (listagem 3.41).

```
def sample_transformer(model, start_prompt, max_tokens, verbose, logs=None):
    max_tokens = max_tokens - len(start_prompt)
    start_tokens = [_ for _ in start_prompt]
    if(verbose == 2):
        print(f"generated text:\n{start_tokens}\n")
    num_tokens_generated = 0
    tokens_generated = []
    while num_tokens_generated + 1 <= max_tokens:
        pad_len = maxlen - len(start_tokens)
        sample_index = len(start_tokens) - 1
        if pad_len < 0:
            x = start_tokens[:maxlen]
            sample_index = maxlen - 1
        elif pad_len > 0:
            x = start_tokens + [0] * pad_len
        else:
            x = start_tokens
        x = np.array([x])
        y, _ = model.predict(x,  verbose = 0) # model = transformer
        sample_token = sample_from(model, y[0][sample_index], verbose)
        tokens_generated.append(sample_token)
        start_tokens.append(sample_token)
        if(verbose):
            print(f"x:\n{sample_token}\n")
        num_tokens_generated = len(tokens_generated)
    txt = start_prompt + tokens_generated
    if(verbose > 0):
        print(f"generated text:\n{txt}\n")
    return txt
```

Listagem 3.41: Implementação da amostragem com o modelo *transformer*.

O processo de escolha de um índice, a partir do conjunto *top-k*, é muito importante para a geração da próxima amostra a fornecer ao VQ-VAE. Na implementação adotada para

o *transformer*, a escolha é feita de acordo com o valor das probabilidades definidas pelo *transformer* na iteração anterior, como se ilustra na listagem 3.42.

```
def sample_from(model, logits, verbose):
    logits, indices = tf.math.top_k(logits, k=10, sorted=True)
    if(verbose == 2):
        print(f"logits:\n{logits}\n")
    indices = np.asarray(indices).astype("int32")
    if(verbose == 2):
        print(f"indices:\n{indices}\n")
    preds = keras.activations.softmax(tf.expand_dims(logits, 0))[0]
    preds = np.asarray(preds).astype("float32")
    if(verbose == 2):
        print(f"preds:\n{preds}\n")
    return np.random.choice(indices, p=preds)
```

Listagem 3.42: Implementação da seleção do índice da amostra a fornecer ao gerador VQ-VAE.

3.3.6 *Implementação do Modelo IntroVAE*

De acordo com o artigo original dos autores Huang et al. (2018), o modelo IntroVAE adota uma arquitetura codificador-descodificador, caraterística do VAE, e um modo de treino semelhante ao das GANs. Em relação aos outros modelos experimentados neste trabalho, a principal novidade reside na função de perda. Assim, na implementação do IntroVAE reutilizou-se boa parte da estrutura do modelo VAE (adaptado de Chollet (2020)), alterando-se a função de perda e modo de treino, tal como documentado no artigo original do IntroVAE (Huang et al. (2018)) (figura 23).

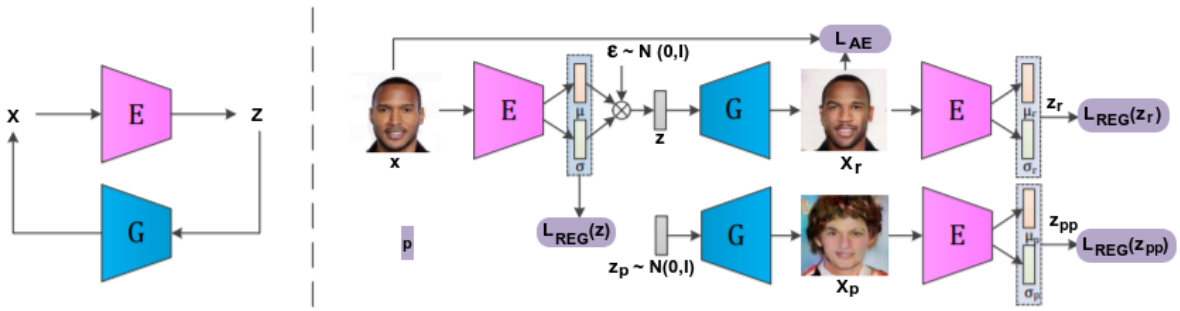


Figura 23: Arquitetura do modelo IntroVAE como no artigo de Huang et al. (2018).

Discriminador

O discriminador do IntroVAE, bloco *E* na figura 23, é muito similar ao codificador do VAE. A sua implementação integra várias camadas convolucionais 2D, que reduzem as dimensões das imagens, seguidas de uma camada de `Dropout`, uma camada de achatamento dos dados

(`Flatten`) e uma camada completamente ligada (`Dense`). O discriminador termina com uma camada que implementa a amostragem de uma distribuição de probabilidades, caraterizada pela média e variância treinadas (listagem 3.43 e figura 24).

```
input_image = Input(shape=(IMG_HEIGHT, IMG_WIDTH, 3))
encoded = Conv2D(128, 3, activation='relu', strides=2, padding='same')(input_image)
encoded = Conv2D(256, 3, activation='relu', strides=2, padding='same')(encoded)
encoded = Conv2D(512, 3, activation='relu', strides=2, padding='same')(encoded)
encoded = Dropout(DROPOUT)(encoded)
encoded = Flatten()(encoded)
encoded = Dense(64, activation='sigmoid')(encoded)
z_mean        = tf.keras.layers(latent_dim, name='z_mean')(encoded)
z_log_sigma   = tf.keras.layers(latent_dim, name='z_log_sigma')(encoded)
z             = Sampling()([z_mean, z_log_sigma])
discriminator = tf.keras.Model(
    input_image, outputs=[z_mean, z_log_sigma, z], name="discriminator")
```

Listagem 3.43: Implementação do discriminador do modelo IntroVAE.

Gerador

A implementação do gerador, bloco G na figura 23, é semelhante à do descodificador do modelo VAE, sendo composta por várias camadas de convolução transposta 2D que, com base na entrada recebida, escalam a dimensão dos dados até ao tamanho das imagens originais. Tal como acontece no descodificador do VAE, também o gerador começa por redimensionar as amostras de entrada através de uma camada `Reshape`, de modo a obter a dimensão pré-definida, a qual pode ser otimizada (listagem 3.44 e figura 25).

```
latent_inputs = keras.Input(shape=(latent_dim,))
decoded = Dense(16 * 16 * 512, activation='relu')(latent_inputs)
decoded = Reshape((16, 16, 512))(decoded)
decoded = Conv2DTranspose(512, kernel_size=4, strides=2, padding='same')(decoded)
decoded = Conv2DTranspose(256, kernel_size=4, strides=2, padding='same')(decoded)
decoded = Conv2DTranspose(128, kernel_size=4, strides=2, padding='same')(decoded)
decoded = Dropout(DROPOUT)(decoded)
generator_outputs = Conv2D(3, kernel_size=5, padding='same', activation="sigmoid")(decoded)
generator = tf.keras.Model(latent_inputs, generator_outputs, name="generator")
```

Listagem 3.44: Implementação do gerador do modelo IntroVAE.

A arquitetura do modelo IntroVAE é definida à custa dos dois componentes mencionados, o discriminador e o gerador, tal como se mostra na listagem 3.45.

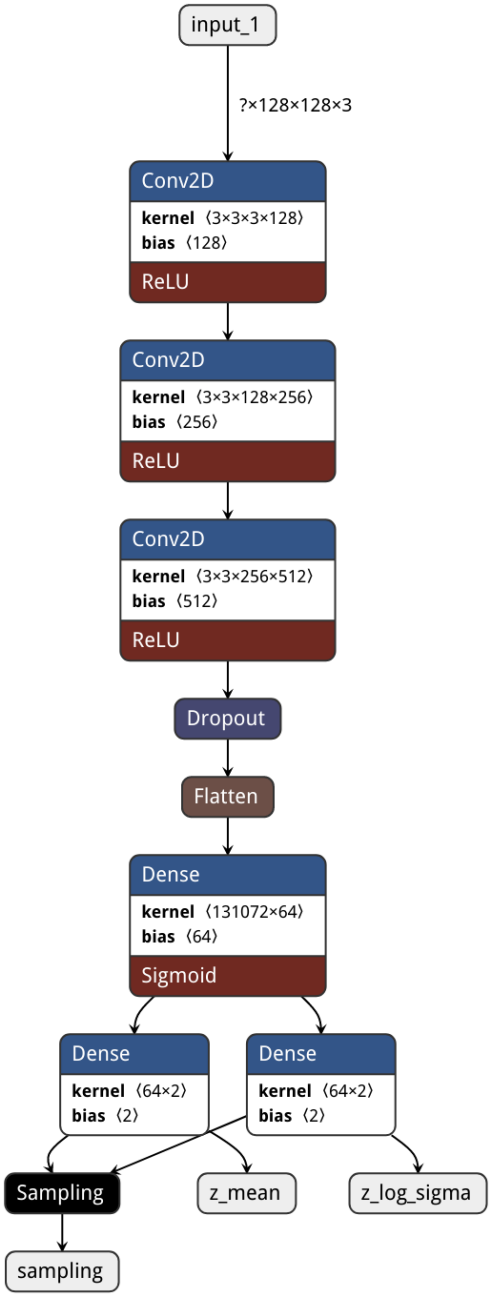


Figura 24: Arquitetura do discriminador do modelo IntroVAE.

```python
class INTROVAE(keras.Model):
    def __init__(self, discriminator, generator, latent_dim):
        super(INTROVAE, self).__init__()
        self.discriminator = discriminator
        self.generator = generator
        self.latent_dim = latent_dim

    def compile(self, d_optimizer, g_optimizer):
        super(INTROVAE, self).compile()
        self.d_optimizer = d_optimizer
        self.g_optimizer = g_optimizer
        self.d_loss_metric = keras.metrics.Mean(name="d_loss")
        self.g_loss_metric = keras.metrics.Mean(name="g_loss")

    @property
    def metrics(self):
        return [self.d_loss_metric, self.g_loss_metric]
```

Listagem 3.45: Implementação da classe do modelo IntroVAE.

Processo de Treino

No treino do modelo IntroVAE destaca-se a inclusão de novas funções de perda, relativamente ao modelo VAE. O algoritmo de treino do modelo IntroVAE á apresentado na figura 26. Os primeiros passos do algoritmo são iguais aos do VAE, começando com a obtenção da codificação z da imagem com o discriminador (ou seja, o codificador do VAE), a extração duma amostra z_p a partir da distribuição intermédia criada pelo discriminador e a consequente reconstrução duma imagem x_r a partir dessa amostra com o gerador (ou seja, o descodificador do VAE). Em paralelo é gerada uma imagem x_p, com o gerador, a partir de uma amostra z_p obtida de uma distribuição normal $\mathcal{N}(0, I)$. Após estes dois passos estarem completos, obtém-se a primeira parcela comum às funções de perda do discriminador e do gerador (L_{AE}), a qual consiste numa perda de reconstrução como resultado da comparação entre a imagem original x e a imagem reconstruída x_r. De seguida, as duas imagens obtidas do gerador, a reconstruída x_r e a nova geração x_p, são fornecidas ao discriminador E, que por sua vez permite criar duas funções de distribuição de probabilidades, uma respetiva à imagem reconstruída (μ_r, σ_r) e outra respetiva à nova imagem gerada (μ_p, σ_p). Os cálculos dessas distribuições incluem o operador $ng()$ que bloqueia a retro-propagação dos gradientes referentes aos dados a que este operador foi aplicado. Neste caso, impede a retro-propagação dos gradientes das primeiras distribuições de probabilidades (z_r e $z_p p$) para o gerador. Destas duas distribuições obtêm-se as amostras z_r e z_{pp}. O próximo passo consiste em obter a segunda parcela da função de perda do discriminador (L^E_{adv}), que aplica o racional utilizado nas GANs, resultando da comparação de três distribuições, a distribuição da imagem reconstruída, a distribuição da imagem gerada e a distribuição, originalmente,

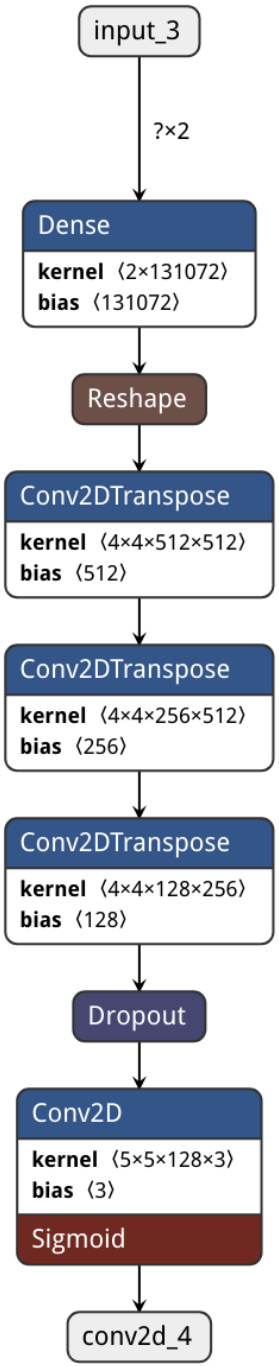


Figura 25: Arquitetura do gerador do modelo IntroVAE.

criada no processo de reconstrução da imagem. De modo a poder comparar as distribuições, as três são regularizadas (L_REG) segundo a fórmula de *Kullback–Leibler*. A função de perda do discriminador é calculada como uma soma pesada entre as duas parcelas, ou seja, $L^E_{adv} + \beta \times L_{AE}$. Esta função é atualizada com o algoritmo Adam, para otimizar os parâmetros Φ_E do discriminador. No passo seguinte, a imagem reconstruída (x_r) e a gerada (x_p) são novamente fornecidas ao discriminador para obter duas funções de distribuição de probabilidades adicionais. A partir destas duas distribuições adicionais são obtidas novas amostras z_r e z_{pp}. Com base nestas amostras é calculada a segunda parcela da função de perda L^G_{adv} do gerador, que resulta da comparação das duas distribuições adicionais. Por fim, a função de perda do gerador é calculada como uma soma pesada entre as duas parcelas, ou seja, $L^G_{adv} + \beta \times L_{AE}$. Esta função é atualizada com o algoritmo Adam, para otimizar os parâmetros θ_G do gerador.

Algorithm 1 Training IntroVAE model

$\theta_G, \phi_E \leftarrow$ Initialize network parameters
while not converged **do**
  $X \leftarrow$ Random mini-batch from dataset
  $Z \leftarrow Enc(X)$
  $Z_p \leftarrow$ Samples from prior $N(0, I)$
  $X_r \leftarrow Dec(Z), X_p \leftarrow Dec(Z_p)$
  $L_{AE} \leftarrow L_{AE}(X_r, X)$
  $Z_r \leftarrow Enc(ng(X_r)), Z_{pp} \leftarrow Enc(ng(X_p))$
  $L^E_{adv} \leftarrow L_{REG}(Z) + \alpha\{[m - L_{REG}(Z_r)]^+ + [m - L_{REG}(Z_{pp})]^+\}$
  $\phi_E \leftarrow \phi_E - \eta\nabla_{\phi_E}(L^E_{adv} + \beta L_{AE})$ ▷ Perform Adam updates for ϕ_E
  $Z_r \leftarrow Enc(X_r), Z_{pp} \leftarrow Enc(X_p)$
  $L^G_{adv} \leftarrow \alpha\{L_{REG}(Z_r) + L_{REG}(Z_{pp})\}$
  $\theta_G \leftarrow \theta_G - \eta\nabla_{\theta_G}(L^G_{adv} + \beta L_{AE})$ ▷ Perform Adam updates for θ_G
end while

Figura 26: Treino do modelo IntroVAE.

A implementação do treino do modelo IntroVAE aplica as funções de perda específicas aos respetivos componentes, atualizando-os iterativamente (listagem 3.46). Seguindo passo a passo o algoritmo de treino, recolhem-se todos os dados necessários para o componente discriminador: a função de distribuição de probabilidades da codificação da imagem real, a reconstrução da imagem real, a imagem gerada a partir da distribuição normal, a distribuição da codificação da imagem reconstruída e a distribuição da codificação da imagem gerada. Com base nestes dados é calculado do valor de perda do discriminador. O treino continua com a recolha de dados relativos às distribuições da codificação da imagem reconstruída e da imagem gerada. Com estes novos dados, e reutilizando os dados previamente recolhidos, calcula-se o valor da perda do gerador.

```
def train_step(self, real_images):
        # Sample random points in the latent space
        batch_size = BATCH_SIZE
        # Train the discriminator
        with tf.GradientTape(persistent=True) as tape_g:
            with tf.GradientTape(persistent=True) as tape_d:
                z_mean, z_log_var, z = self.discriminator(real_images)
                z_p = tf.random.normal(shape=(batch_size, self.latent_dim))
                reconstruction = self.generator(z)
                reconstruction_p = self.generator(z_p)
                z_mean_r, z_log_var_r, z_r = self.discriminator(reconstruction)
                z_mean_p, z_log_var_p, zp_r = self.discriminator(reconstruction_p)
                kl_loss = -0.5 * (1 + z_log_var - tf.square(z_mean) - tf.exp(z_log_var))
                kl_loss = tf.reduce_mean(tf.reduce_sum(kl_loss, axis=1))
                kl_loss_r = -0.5 * (1 + z_log_var_r - tf.square(z_mean_r) - tf.exp(z_log_var_r))
                kl_loss_r = tf.reduce_mean(tf.reduce_sum(kl_loss_r, axis=1))
                kl_loss_p = -0.5 * (1 + z_log_var_p - tf.square(z_mean_p) - tf.exp(z_log_var_p))
                kl_loss_p = tf.reduce_mean(tf.reduce_sum(kl_loss_p, axis=1))
                r_dist = tf.math.maximum(tf.constant([0.]), (MARGIN - kl_loss_r))
                p_dist = tf.math.maximum(tf.constant([0.]), (MARGIN - kl_loss_p))
                LEADV = kl_loss + ALPHA * (r_dist + p_dist)
                reconstruction_loss = tf.keras.losses.mean_squared_error(real_images,
                                      reconstruction)
                LAE = BETA * reconstruction_loss
                d_loss = LEADV + LAE
            grads_d = tape_d.gradient(d_loss, self.discriminator.trainable_weights)
            self.d_optimizer.apply_gradients(
                zip(grads_d, self.discriminator.trainable_weights))
            z_mean_r, z_log_var_r, z_r = self.discriminator(reconstruction)
            z_mean_p, z_log_var_p, zp_r = self.discriminator(reconstruction_p)
            kl_loss_r = -0.5 * (1 + z_log_var_r - tf.square(z_mean_r) - tf.exp(z_log_var_r))
            kl_loss_r = tf.reduce_mean(tf.reduce_sum(kl_loss_r, axis=1))
            kl_loss_p = -0.5 * (1 + z_log_var_p - tf.square(z_mean_p) - tf.exp(z_log_var_p))
            kl_loss_p = tf.reduce_mean(tf.reduce_sum(kl_loss_p, axis=1))
            LGADV = ALPHA * (kl_loss_r + kl_loss_p)
            g_loss = LGADV + LAE
        grads_g = tape_g.gradient(d_loss, self.generator.trainable_weights)
        self.g_optimizer.apply_gradients(
            zip(grads_g, self.generator.trainable_weights))
        # Update metrics
        self.d_loss_metric.update_state(d_loss)
        self.g_loss_metric.update_state(g_loss)
        return {
            "d_loss": self.d_loss_metric.result(),
            "g_loss": self.g_loss_metric.result(),
        }
```

Listagem 3.46: Otimização do discriminador e do gerador do modelo IntroVAE.

4

EXPERIÊNCIAS E RESULTADOS

Este capítulo apresenta as experiências realizadas e os resultados obtidos durante o desenvolvimento da dissertação. As experiências consistem nos diferentes testes feitos com os diferentes modelos, incluindo testes feitos sobre diferentes resoluções das imagens. Primeiramente, os modelos foram treinados com um conjunto de dados com uma resolução de 128×128, sendo posteriormente treinados com um conjunto de dados com resolução de 512×512, com o qual se pretende retirar conclusões referentes a imagens de alta resolução. Dadas as limitações impostas pelo tipo de GPU disponível para treinar os modelos, a resolução máxima testada, 512×512, não foi tão elevada como se desejaria. Contudo, isso não impediu atingir os principais objetivos da dissertação.

Em cada modelo, as primeiras experiências foram realizadas com o conjunto de dados MNIST, tendo em vista avaliar o correto funcionamento dos modelos, uma tarefa mais fácil de concretizar com um conjunto de dados simples. Depois de se confirmar a correção arquitetural dos modelos, treinaram-se os modelos para gerar faces humanas com resolução de 128×128. Este passo intermédio foi essencial para avaliar o comportamento dos modelos na geração de imagens do tipo pretendido, mas mantendo o peso computacional num valor moderado. Depois de terminar a avaliação dos modelos na geração de faces com menor resolução, foram feitos os ajustes necessários nos modelos para terem capacidade de lidar com imagens de resolução 4 vezes maior em cada dimensão. Os modelos implementados e testados nesta terceira fase, são aqueles que permitiram atingir o principal objetivo proposto neste trabalho, a geração de faces com alta resolução.

4.1 VARIATIONAL AUTOENCODER

As experiências feitas com modelos VAE seguiram os parâmetros indicados na tabela 1. A diferença do tamanho dos conjuntos de dados das duas resoluções exploradas é refletida no número de imagens usadas para treino e validação para o modelo respetivo. O otimizador usado para todas as experiências de todos os modelos foi o *adam*. No caso do VAE, para ambas as resoluções de imagens treinadas, a taxa de aprendizagem foi de $3e{-}4$. O número de épocas utilizadas para treinar os modelos foi 100. A principal diferença entre os parâmetros

de treino das duas resoluções é o *batch size*, devido às limitações de recursos, o *batch size* para imagens com resolução 128×128 é de 32, enquanto que para imagens com resolução 512×512 é 16.

Modelo	Nº imagens treino	Nº imagens validação	Batch Size	Nome de Otimizador	Taxa de aprendizagem	Nº de épocas
VAE 128	56000	14000	32	adam	3e−4	100
VAE 512	41600	10400	16	adam	3e−4	100

Tabela 1: Parâmetros de treino dos modelos VAE.

A implementação adotada no modelo VAE revelou consistência na geração de estruturas faciais identificáveis. Após encontrar a configuração ótima para o número de épocas do treino e para a taxa de aprendizagem, foi possível atingir resultados bastante positivos. Contudo, mesmo as melhores versões do modelo são incapazes de representar os detalhes das imagens.

4.1.1 *Imagens com resolução* 128×128

O progresso da função perda ao longo do treino do modelo VAE é apresentado graficamente na figura 27. De acordo com o gráfico, o treino do modelo VAE progride rapidamente nas 20 primeiras épocas. A partir daí a redução da perda é bastante mais lenta.

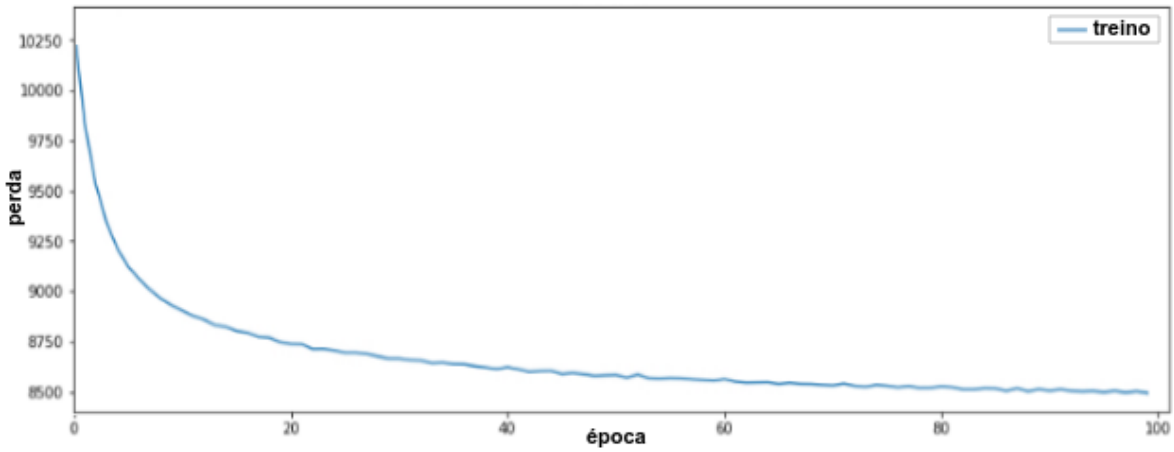


Figura 27: Perda durante o treino do modelo VAE com imagens de resolução 128×128.

Globalmente, o modelo VAE consegue uma reconstrução de imagens bastante positiva. Um dos pontos fortes deste modelo é a excelente capacidade de reconstrução das principais caraterísticas de imagens reais (figura 28). Em termos da geração de imagens, verifica-se uma boa capacidade de gerar faces globalmente corretas (figura 29). Contudo, é notória a incapacidade de representar os detalhes e a dificuldade em manter a consistência das

texturas das imagens. Quanto mais parecidas às imagens de treino forem as imagens geradas, maior é a qualidade das imagens geradas.

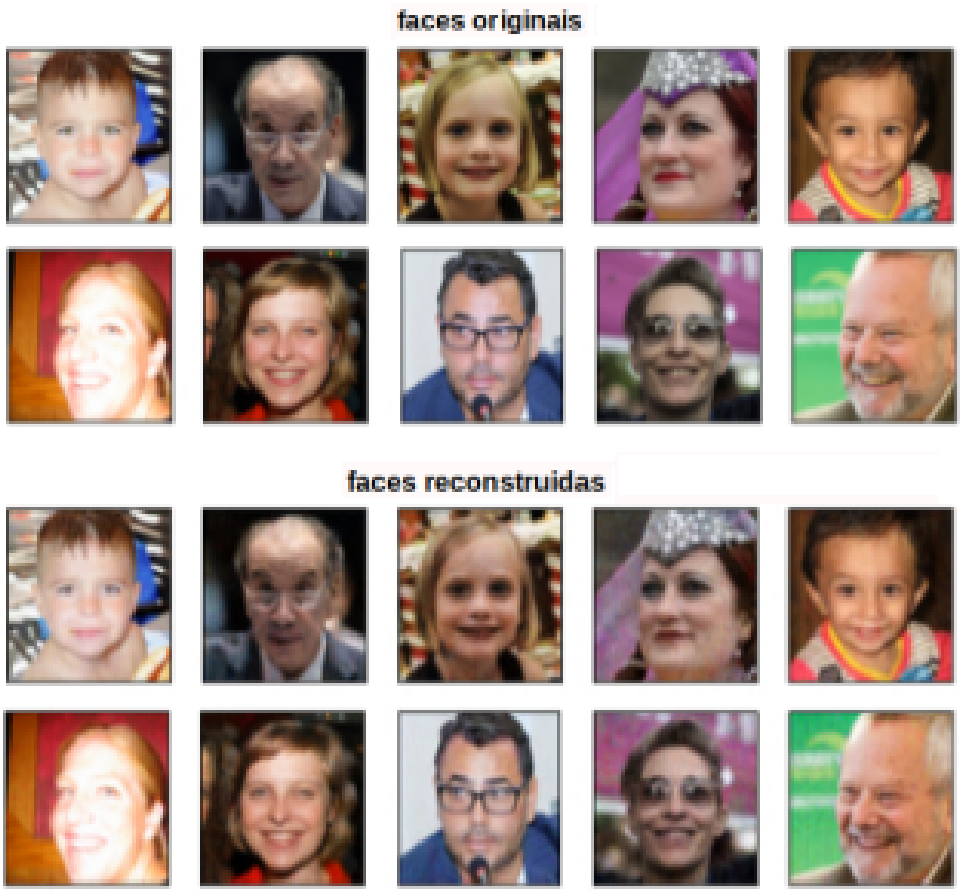


Figura 28: Reconstrução de imagens de resolução 128 × 128 com o modelo VAE.

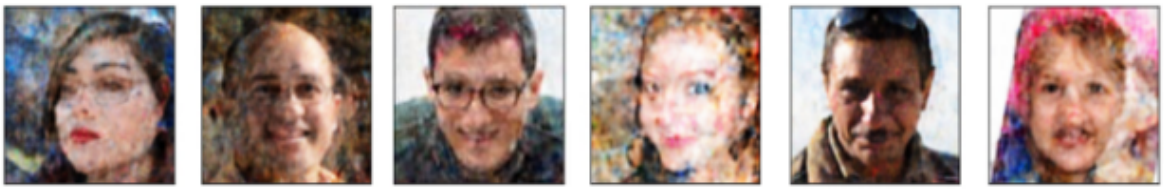


Figura 29: Geração de imagens de resolução 128 × 128 com o modelo VAE.

4.1.2 *Imagens com resolução* 512 × 512

Como seria de esperar, treinar o modelo VAE com imagens de maior resolução resultou numa perda de valor muito superior à que se obteve com o modelo treinado com imagens

de resolução inferior. Não obstante, após 20 épocas o treino do modelo converge para uma solução aceitável (figura 30)

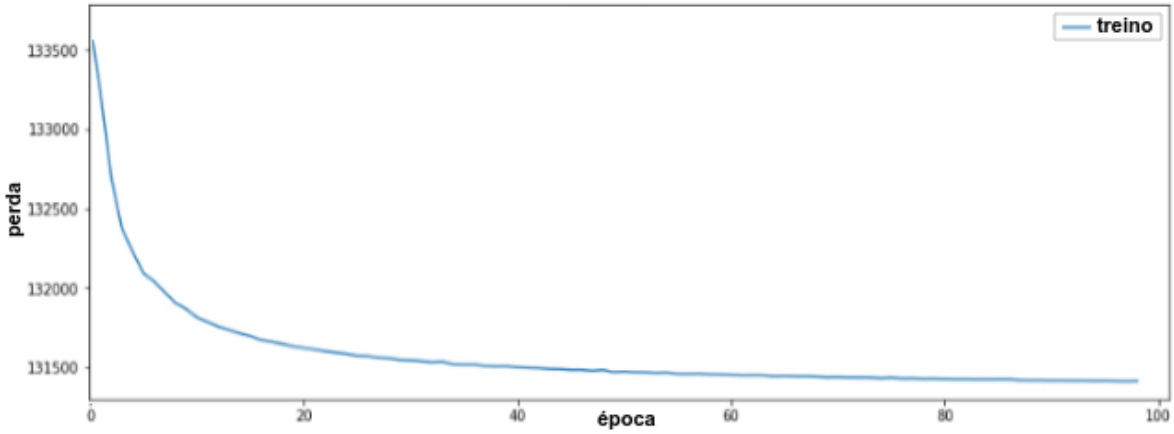


Figura 30: Perda durante o treino do modelo VAE com imagens de resolução 512×512.

Ao utilizar imagens com maior resolução, constata-se que o modelo consegue efetuar uma reconstrução mais fiel às imagens originais (figura 31). Na fase de geração, no essencial o comportamento do modelo mantém-se, notando-se um ligeiro declínio na capacidade de representar os detalhes (figura 32). Tendo em consideração os limitados recursos de *hardware* disponíveis para treinar o modelo, era expectável que o manuseamento de imagens de alta resolução fosse intratável em termos de tempo de cálculo, devido ao maior peso computacional requerido, ou mesmo impossível, por ser impossível armazenar os pesos do modelo e as respetivas derivadas na memória do GPU. Uma das formas de contornar esta limitação passou por reduzir o tamanho do *batch*, mas esta abordagem traduziu-se numa redução da qualidade das imagens geradas. Ainda assim, conseguiu-se obter resultados satisfatórios, revelando que a adoção de imagens maiores tem potencial para melhorar os resultados, desde que se disponha de mais recursos computacionais.

4.2 VECTOR-QUANTIZED VARIATIONAL AUTOENCODER

O treino dos modelos VQ-VAE fizeram uso dos parâmetros indicados na tabela 2. Para ambos os modelos referentes às duas resoluções de imagens treinadas, a taxa de aprendizagem foi de $1e{-}4$. O número de épocas utilizadas para treinar o modelo dedicado a imagens com resolução 128×128 foi 50, ao passo que para as imagens com resolução 512×512 apenas foram usadas 20 épocas. Novamente, devido às limitações de recursos, a principal diferença entre os parâmetros de treino das duas resoluções é o *batch size*, para imagens com resolução 128×128 o valor usado foi 32, enquanto que para imagens com resolução 512×512 só foi possível o valor 8.

Figura 31: Reconstrução de imagens de resolução 512 × 512 com o modelo VAE.

Figura 32: Geração de imagens de resolução 512 × 512 com o modelo VAE.

Modelo	Nº imagens treino	Nº imagens validação	Batch Size	Nome de Otimizador	Taxa de aprendizagem	Nº de épocas
VQVAE 128	56000	14000	32	adam	1e−4	50
VQVAE 512	41600	10400	8	adam	1e−4	20

Tabela 2: Parâmetros de treino dos modelos VQ-VAE.

Contrariamente ao VAE, o VQ-VAE demonstrou uma dificuldade maior para, de forma consistente, gerar faces globalmente corretas. Por outro lado, o modelo demonstrou sempre uma grande capacidade para gerar detalhes de alta qualidade das faces. Em grande parte das gerações, o resultado obtido consistiu em imagens compostas por ruído estático misturado com pequenas partes de faces muito detalhadas e de boa qualidade.

4.2.1 *Imagens com resolução* 128×128

A evolução dos valores da perda durante o treino do modelo VQ-VAE encontra-se na figura 33. A partir desta figura é possível identificar uma rápida convergência da otimização do modelo nas primeiras 10 épocas, seguida de 40 épocas onde se verifica alguma estagnação da otimização. Esta célere evolução da otimização do modelo VQ-VAE nas épocas iniciais resulta, principalmente, de se partir de um valor de perda inicial bastante mais elevado que nas restantes épocas, sendo por isso fácil baixar o valor da perda nas primeiras 10 épocas, atingindo-se um valor próximo de zero.

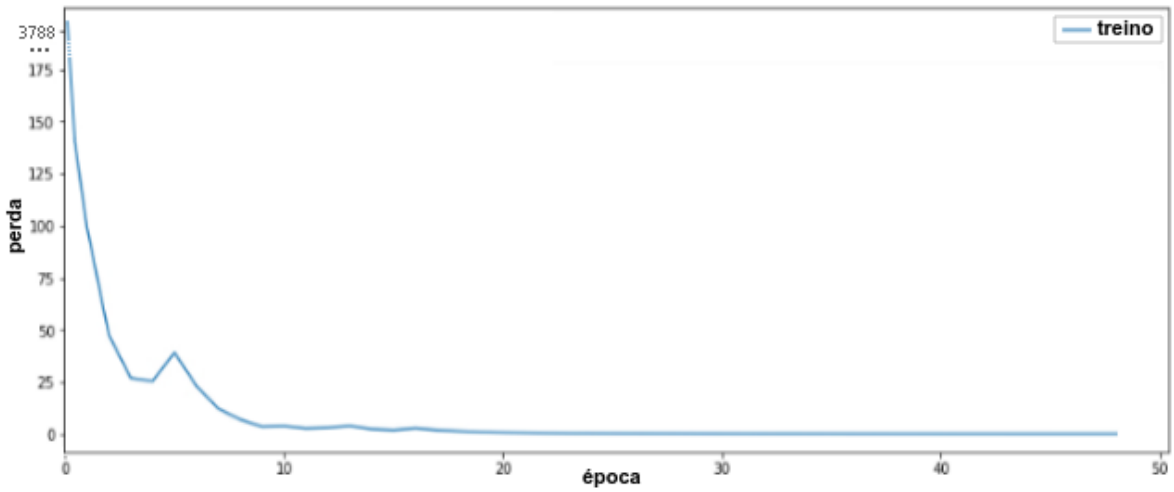


Figura 33: Perda durante o treino do modelo VQ-VAE com imagens de resolução 128×128.

O modelo treinado consegue efetuar reconstruções de excelente qualidade a partir da representação latente das imagens do conjunto de dados. Mesmo nas imagens com faces bastante diferentes do padrão habitual, as reconstruções são sempre muito fiéis às imagens

originais. A elevada qualidade das reconstruções revela que é fácil treinar os componentes codificador e descodificador do modelo VQ-VAE, e obter uma boa sinergia entre eles, independentemente de uma eventual falta de diversidade na representação latente, a qual irá dificultar a tarefa de geração (figura 34).

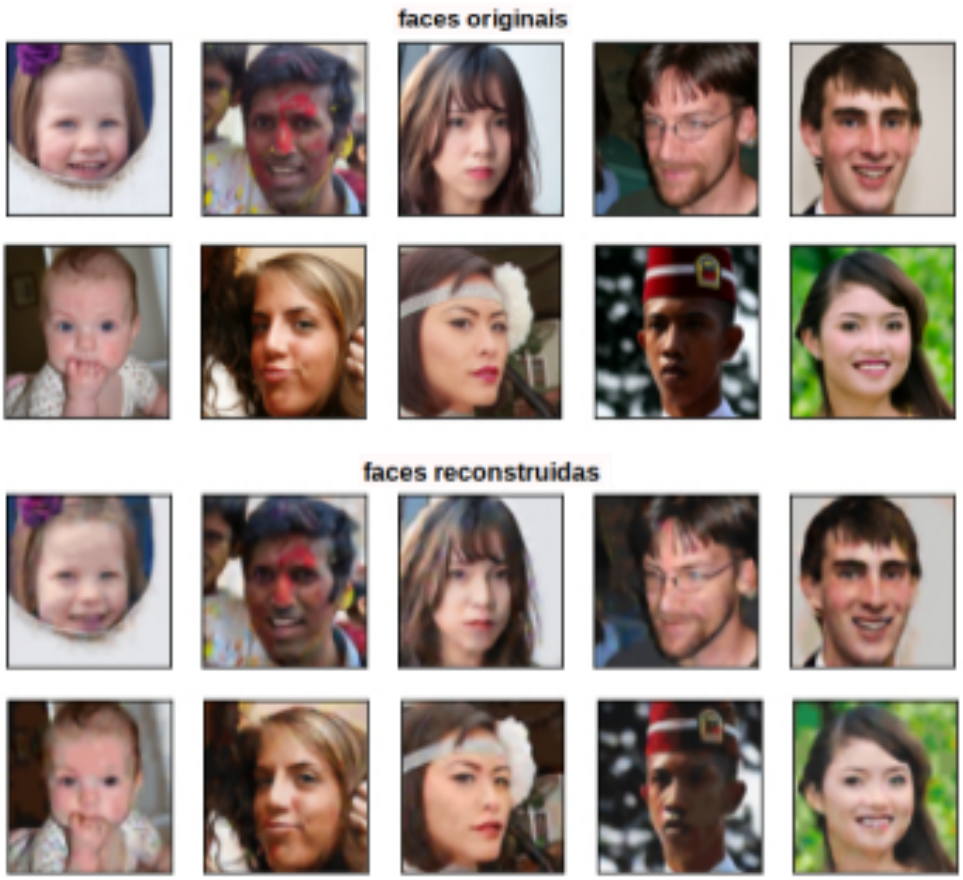


Figura 34: Reconstrução de imagens de resolução 128×128 com o modelo VQ-VAE.

As faces geradas pelo modelo VQ-VAE possuem uma qualidade satisfatória, ainda que haja alguma inconsistência na geração de faces aceitáveis. Os resultados obtidos mostram boa capacidade para representar os detalhes das faces (figura 35). A principal falha a apontar à capacidade de geração do modelo é mesmo a percentagem de exemplos em que se verifica incoerência nas imagens como um todo, mesmo quando partes da imagem apresentam um bom detalhe. Durante as várias iterações do processo de geração de imagens, verificou-se um padrão que se passa a explicar. Na maioria das faces geradas de forma incoerente, a que podemos chamar geração sem sucesso, a parte superior esquerda da imagem, que pode representar uma grande parte da imagem, apresenta boa qualidade. Mas a parte restante da imagem, que termina no canto inferior direito, apresenta menor qualidade e não condiz com o resto da imagem (figura 36). Muito provavelmente, este padrão resulta da forma como o PixelCNN funciona, o qual começa a gerar cada imagem a partir do canto superior

esquerdo, depois evolui ao longo da linha atual até ao limite direito da imagem, e finalmente a geração prossegue na linha da imagem imediatamente abaixo da linha atual. Ou seja, a parte da imagem que é gerada primeiro é coerente, mas a partir de certo ponto o processo de geração autorregressivo falha e o valor de cada pixel gerado deixa de estar coerente com os pixeis anteriormente gerados.

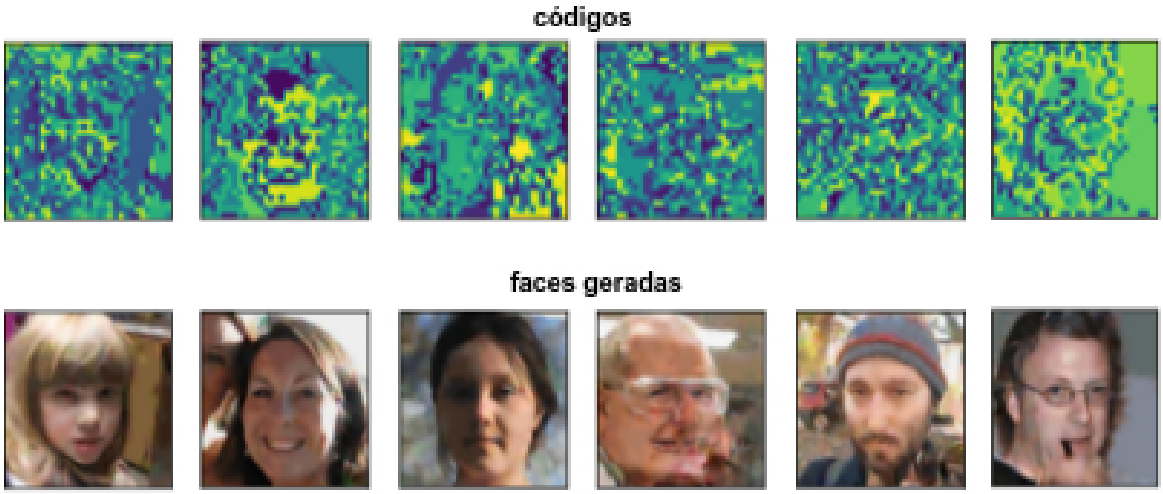


Figura 35: Geração de imagens de resolução 128×128 com o modelo VQ-VAE.

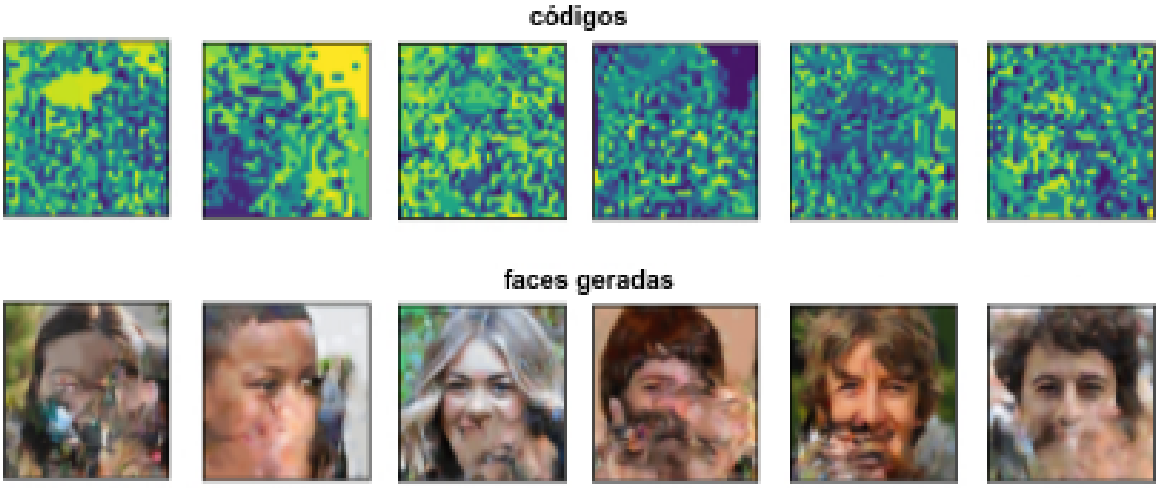


Figura 36: Geração de imagens de resolução 128×128 parcialmente coerentes com o modelo VQ-VAE.

4.2.2 *Imagens com resolução* 512 × 512

Treinar o modelo VQ-VAE com imagens de resolução 512 × 512 revelou-se uma tarefa difícil, particularmente porque este cenário acentua o problema identificado durante o treino do modelo com imagens de menor resolução, e que se materializa num valor inicial da perda muito alto relativamente aos restantes valores verificados ao longo do treino. Este problema é parcialmente visível no gráfico da figura 37, mas em que na realidade ocorre uma redução muito acentuada no valor da perda ao longo das primeiras 5 épocas do treino.

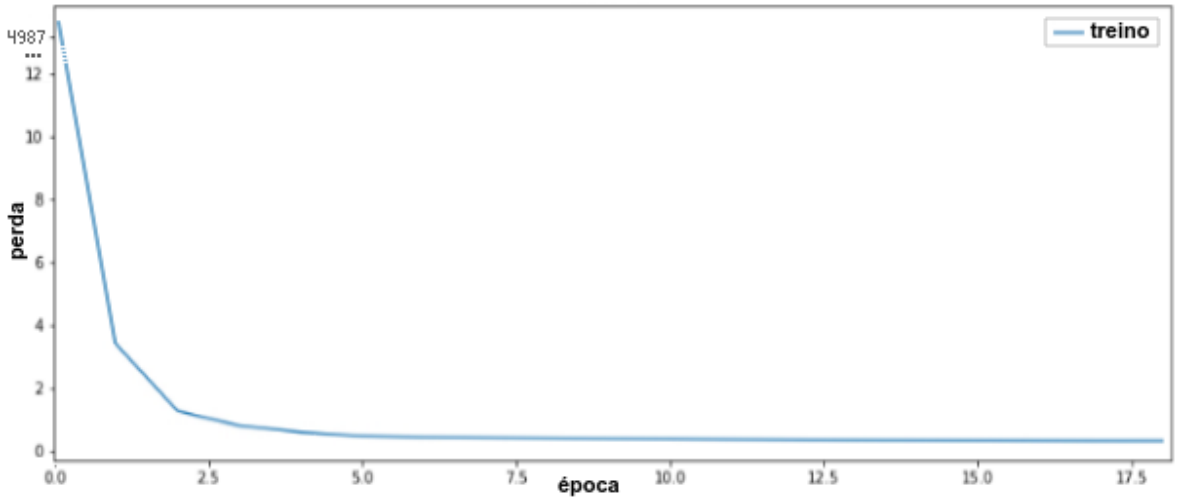


Figura 37: Perda durante o treino do modelo VQ-VAE com imagens de resolução 512 × 512.

O modelo treinado com imagens de resolução 512 × 512 revelou-se ser capaz de gerar faces visualmente satisfatórias, mais até do que o modelo treinado com imagens de menor resolução. A maior dimensão das imagens de treino contém uma melhor representação dos detalhes, o que leva a que as imagens geradas também consigam reproduzir os pormenores que constituem uma face humana. Descontando as limitações impostas pelo peso computacional associado à utilização do modelo com mais informação, a qualidade das imagens geradas é bastante boa (figura 38).

4.3 VECTOR-QUANTIZED VARIATIONAL AUTOENCODER 2

Os parâmetros de treino indicados na tabela 3, são referentes aos modelos do tipo VQ-VAE2 experimentados. Os modelos correspondentes às duas resoluções de imagens treinadas utilizaram uma taxa de aprendizagem de 1e−4. O número de épocas utilizadas para treinar os modelos foi 40. A diferença no *batch size* mantém-se a mesma dos modelos VQ-VAE, para imagens com resolução 128 × 128 o valor usado foi 32, enquanto que para imagens com resolução 512 × 512 foi apenas 8.

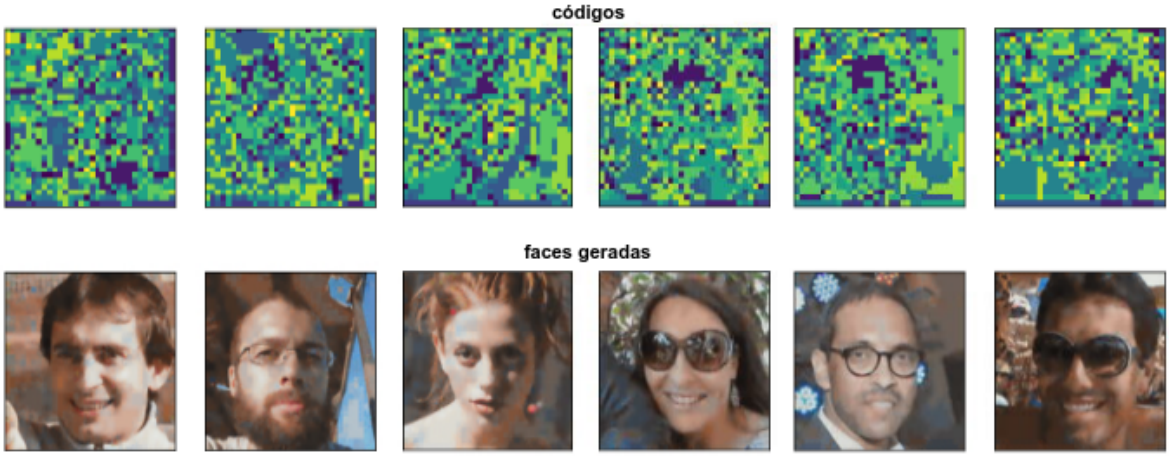


Figura 38: Geração de imagens de resolução 512 × 512 com o modelo VQ-VAE.

Modelo	Nº imagens treino	Nº imagens validação	Batch Size	Nome de Otimizador	Taxa de aprendizagem	Nº de épocas
VQVAE2 128	56000	14000	32	adam	1e−4	40
VQVAE2 512	41600	10400	8	adam	1e−4	40

Tabela 3: Parâmetros de treino dos modelos VQ-VAE2.

Os resultados obtidos com o modelo VQ-VAE2 revelam maior consistência na obtenção de imagens coerentes e detalhadas. Tal como o nome indica, este modelo coloca problemas semelhantes aos do VQ-VAE, embora apresente melhor capacidade de resposta. Assim como o VQ-VAE, o VQ-VAE2 demonstra dificuldade em gerar imagens coerentes com regularidade. No entanto, a estrutura hierárquica permite melhorar o comportamento nesta faceta relativamente ao VQ-VAE.

4.3.1 *Imagens com resolução* 128 × 128

O problema identificado no treino do modelo VQ-VAE, que consistia em obter valores de perda elevadíssimos nas primeiras épocas do treino, também ocorre no treino do modelo VQ-VAE2. A figura 39 evidencia este problema, traduzido numa grande inclinação do gráfico nas primeiras 3 épocas, seguido de uma curva quase plana, resultante de uma fraca otimização do modelo durante o resto do treino.

Tal como o seu antecessor, o VQ-VAE2 demonstra excelente capacidade de reconstrução de imagens (figura 40). Mais uma vez, há que mencionar que o treino do modelo VQ-VAE2 foi efetuado com restrições, sendo por isso compreensível que os resultados obtidos sejam semelhantes aos do VQ-VAE. Apesar disso, os resultados finais obtidos evidenciam que mesmo com uma otimização com restrições é possível gerar imagens de faces plausíveis e em alguns casos bem detalhadas (figura 41).

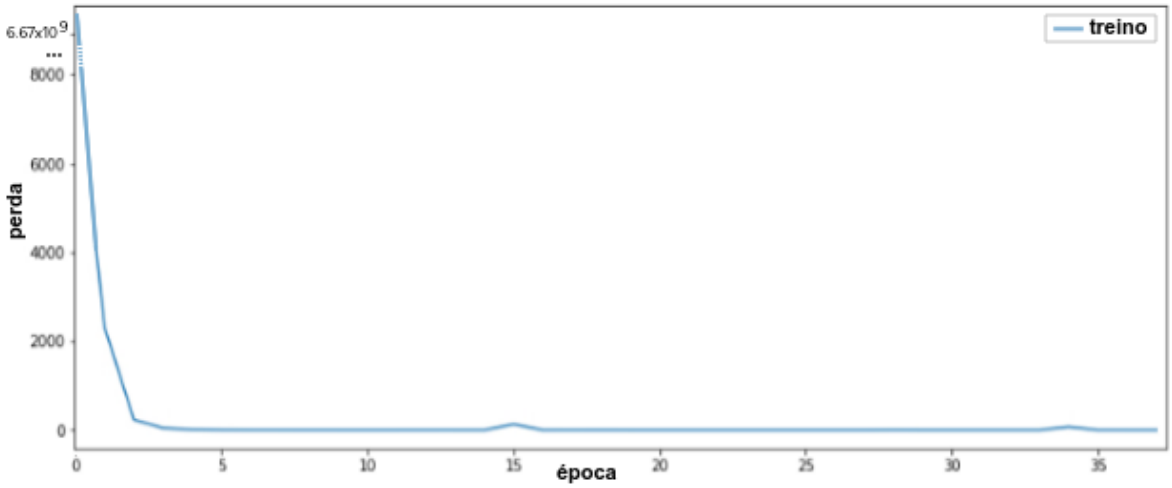


Figura 39: Perda durante o treino do modelo VQ-VAE2 com imagens de resolução 128 × 128.

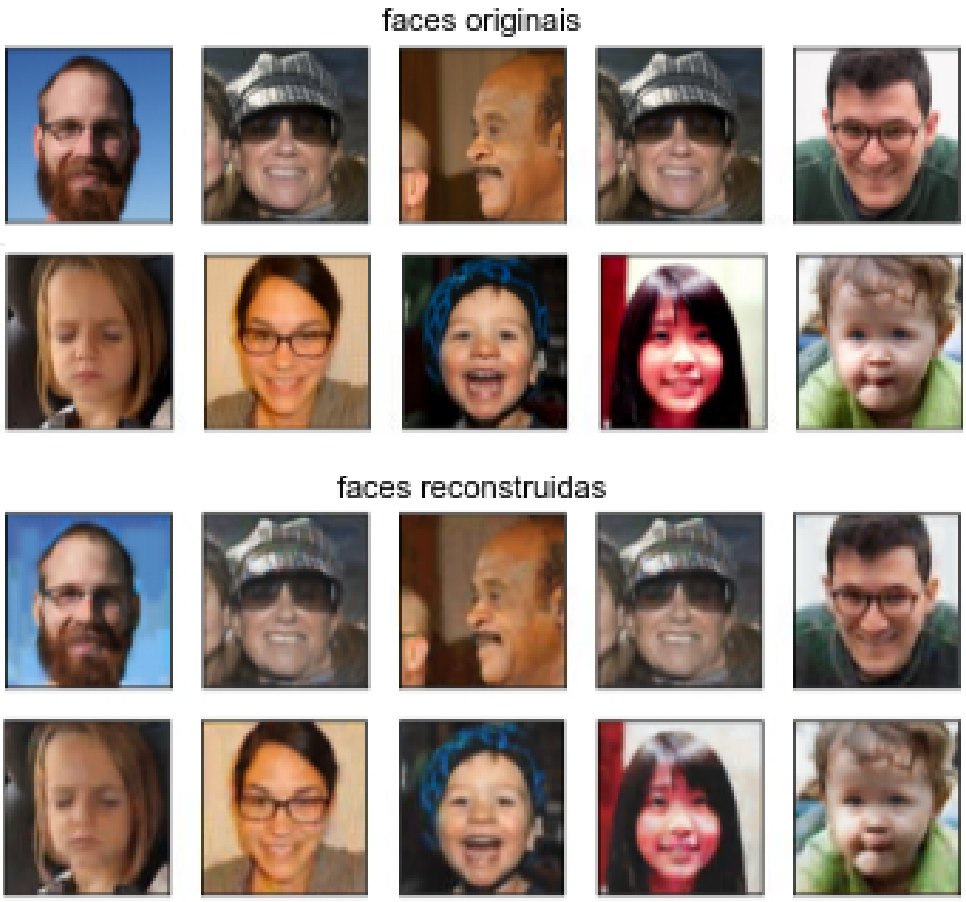


Figura 40: Reconstrução de imagens de resolução 128 × 128 com o modelo VQ-VAE2.

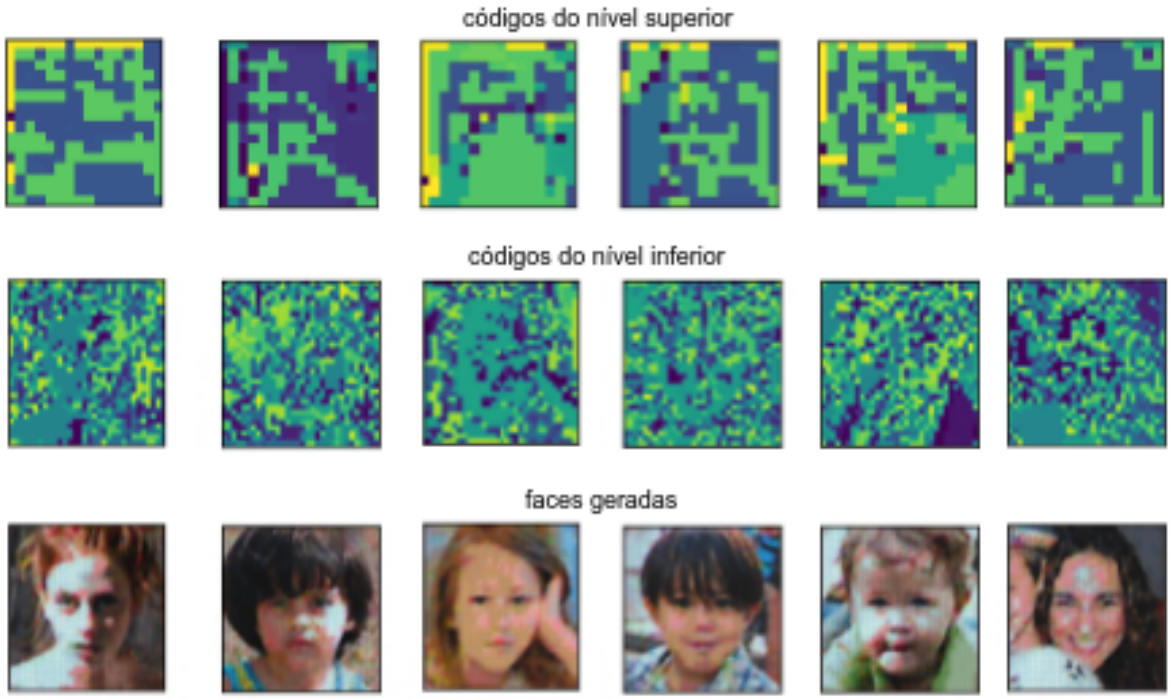


Figura 41: Geração de imagens de resolução 128 × 128 com o modelo VQ-VAE2.

4.3.2 *Imagens com resolução* 512 × 512

A curva que mostra a evolução da perda durante o treino do modelo VQ-VAE2 com imagens de resolução 512 × 512 revela novamente uma derivada enorme nas primeiras épocas, em consequência de o valor inicial da perda ser muito elevado (figura 42).

Os problemas identificados quando o modelo trabalha com imagens de 128 × 128 pixeis são acentuados com imagens de maior resolução. Porém, o modelo continua a ser capaz de gerar imagens claramente representativas de faces humanas, mesmo que com alguma pixelização das mesmas (figura 43).

4.4 Redes Adversárias Geradoras

O treino do modelo GAN recorreu aos parâmetros de treino apresentados na tabela 4. A taxa de aprendizagem foi a mais baixa de todos os modelos implementados com um valor de 1e−5. Em virtude das limitações de recursos presentes no desenvolvimento deste projeto, o modelo GAN apenas pode ser treinado por 50 épocas e com um *batch size* de 8.

O modelo GAN apresentou boa capacidade de geração, particularmente na reprodução de detalhes. O modelo implementado mostrou ser o mais capaz em termos de geração dos

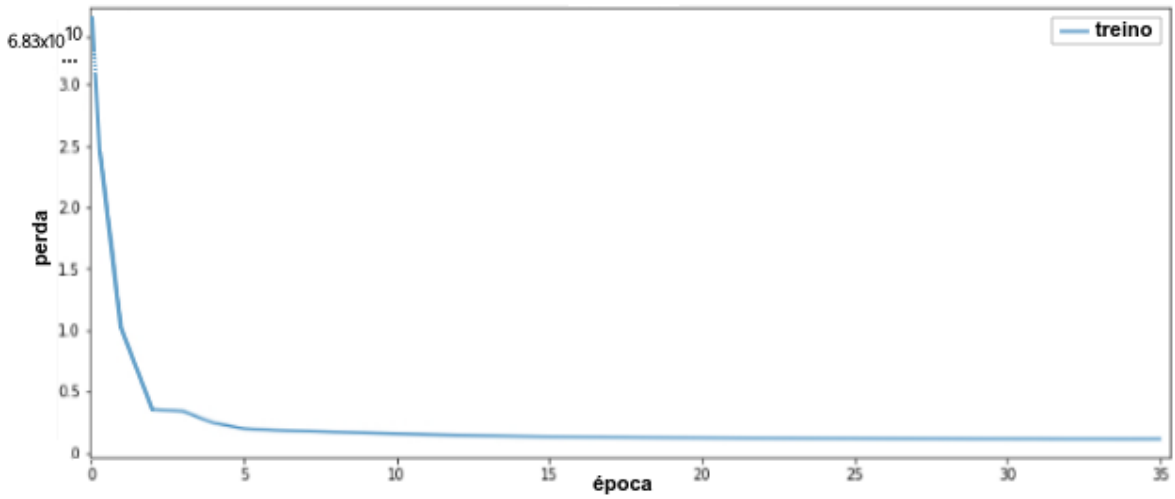


Figura 42: Perda durante o treino do modelo VQ-VAE2 com imagens de resolução 512 × 512.

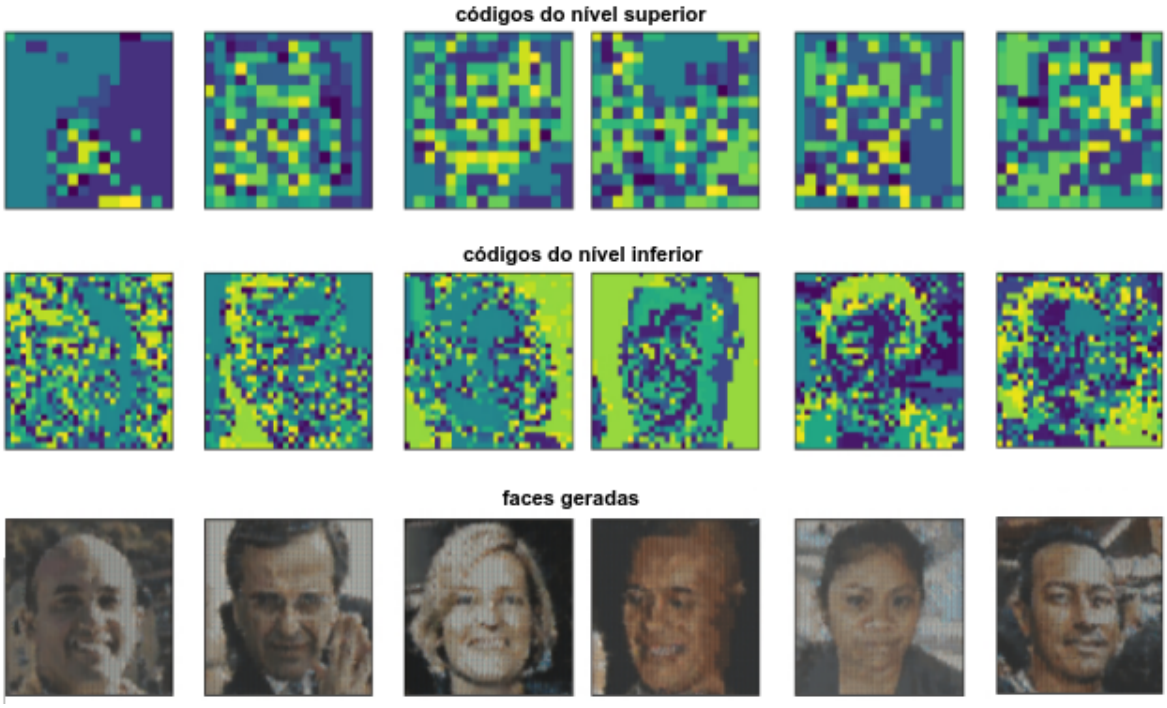


Figura 43: Geração de imagens de resolução 512 × 512 com o modelo VQ-VAE2.

Modelo	Nº imagens treino	Nº imagens validação	Batch Size	Nome de Otimizador	Taxa de aprendizagem	Nº de épocas
GAN 128	56000	14000	8	adam	1e−5	50

Tabela 4: Parâmetros de treino do modelo GAN.

detalhes que compõem uma face, muito embora falhe na composição de faces globalmente coerentes.

4.4.1 *Imagens com resolução* 128×128

Os resultados obtidos pelo modelo GAN treinado demonstram a capacidade do modelo para representar os detalhes faciais. O principal problema das GANs reside no elevado peso computacional necessário para as treinar, sendo necessários fortes recursos computacionais e além disso a otimização dos modelos é pouco estável e demorada. Ou seja, não foi fácil treinar uma GAN para gerar imagens que representassem uma face humana minimamente coerente. As faces tipicamente geradas parecem ser uma colagem de pedaços de faces bem detalhadas. Como em casos raros as faces apresentam maior coerência, pode-se concluir que, sem as restrições de treino, o modelo tem elevado potencial para gerar faces com boa resolução (figura 44).

Pelos motivos explicados, não foi treinado o modelo GAN com imagens de resolução 512×512.

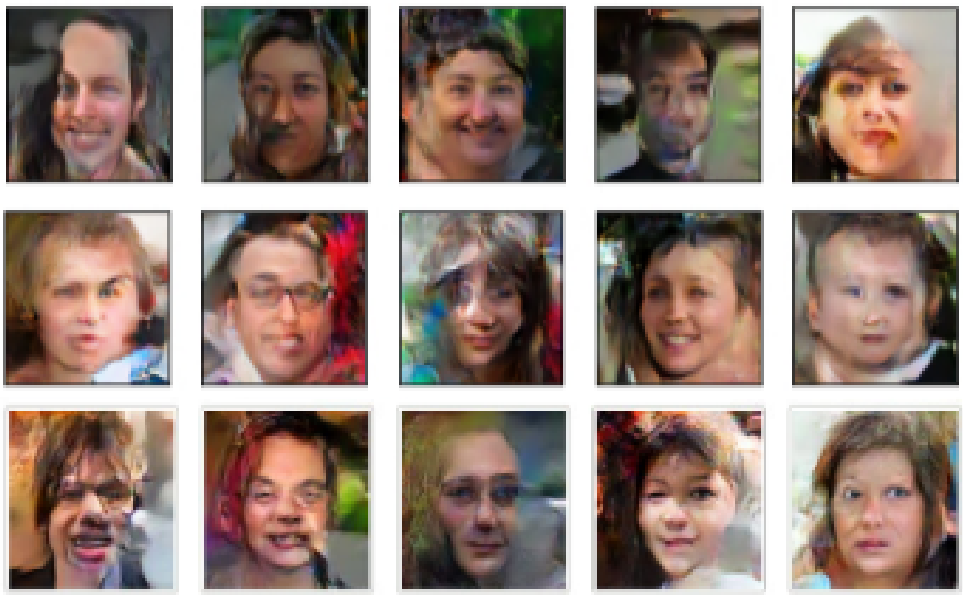


Figura 44: Geração de imagens de resolução 128×128 com o modelo GAN.

4.5 VECTOR-QUANTIZED GENERATIVE ADVERSARIAL NETWORK

As experiências relativas aos modelos do tipo VQ-GAN aplicaram os parâmetros de treino indicados na tabela 5. Todos os modelos testados, com respeito às duas resoluções de imagens treinadas, utilizaram uma taxa de aprendizagem de 2e−4. O número de épocas utilizadas para treinar todos modelos VQ-GAN foi 50. Mais uma vez, as limitações tecnológicas foram a causa da maior diferença entre os conjuntos de parâmetros usados, particularmente, no que concerne ao *batch size*. Os modelos para imagens com resolução 128×128 usaram um *batch size* de 8, enquanto que para imagens com resolução 512×512 foi apenas 4.

Modelo	Nº imagens treino	Nº imagens validação	Batch Size	Nome de Otimizador	Taxa de aprendizagem	Nº de épocas
VQGAN 128	56000	14000	8	adam	2e−4	50
VQGAN 512	41600	10400	4	adam	2e−4	50

Tabela 5: Parâmetros de treino dos modelos VQGAN.

Os resultados obtidos com o modelo VQ-GAN revelam que ele possui boa capacidade de geração, tanto ao nível de uma correta estrutura facial como na representação dos detalhes. Comparando com os outros modelos implementados, o VQ-GAN foi aquele que gerou imagens de melhor qualidade. Apenas o modelo VQ-VAE obteve resultados qualitativos semelhantes aos do VQ-GAN. O ponto mais fraco do VQ-GAN foi a menor originalidade nas imagens que gera. Concretamente, o modelo apresentou uma dificuldade acrescida em gerar imagens que fossem muitos distintas das imagens sobre as quais foi treinado.

4.5.1 *Imagens com resolução* 128×128

A evolução da função de perda durante o treino do modelo VQ-GAN, com imagens de resolução 128×128, encontra-se na figura 45. Verifica-se que a contribuição mais relevante para a perda total é a perda de quantização do *codebook*, representado por `vq_loss` na figura 45. As outras contribuições para a perda total são a perda percetual, calculada recorrendo a uma rede neuronal auxiliar VGG e representada por `vgg_loss` na figura 45, e a perda de *feature matching* referente ao impacto do discriminador no treino do modelo gerador, identificada como `feat_loss` na mesma figura. Estas duas últimas contribuições têm menor influência na perda total, e por conseguinte, na otimização do modelo.

A figura 46 contém alguns exemplos que ilustram a qualidade das imagens geradas pelo modelo VQ-GAN. Estes exemplos correspondem a faces estruturalmente coerentes e suficientemente detalhadas. Apesar de alguma inconsistência na representação dos detalhes mais finos, nas várias iterações do processo de geração, que é visível em alguns pormenores

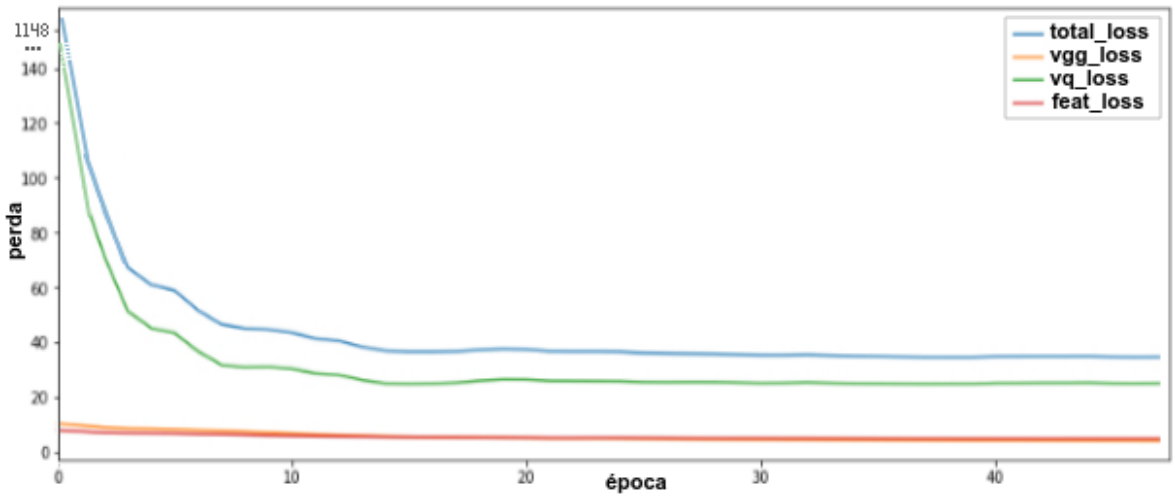


Figura 45: Perda durante o treino do VQ-GAN com imagens de resolução 128 × 128.

um pouco esborratados, a qualidade das imagens geradas é boa e é superior à dos outros modelos implementados.

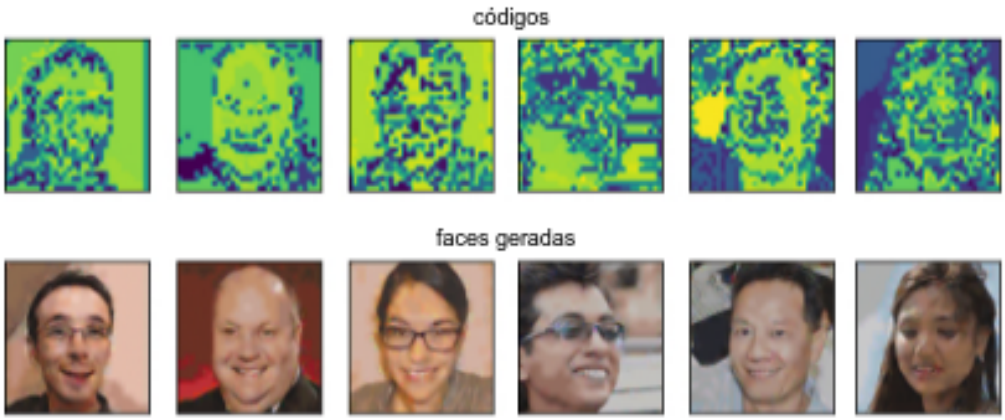


Figura 46: .
Geração de imagens de resolução 128 × 128 com o modelo VQ-GAN.

4.5.2 *Imagens com resolução* 512 × 512

O treino do modelo VQ-GAN com imagens de resolução 512 × 512 não difere significativamente do treino com imagens de menor resolução. De acordo com o gráfico da figura 47, a única diferença notória é o facto da perda *feature matching* (`feat_loss`) ser claramente maior do que a perda percetual (`vgg_loss`), que é calculada com auxílio da rede VGG. Este comportamento pode indicar uma dificuldade ligeiramente superior em conjugar o treino do discriminador com o treino do gerador.

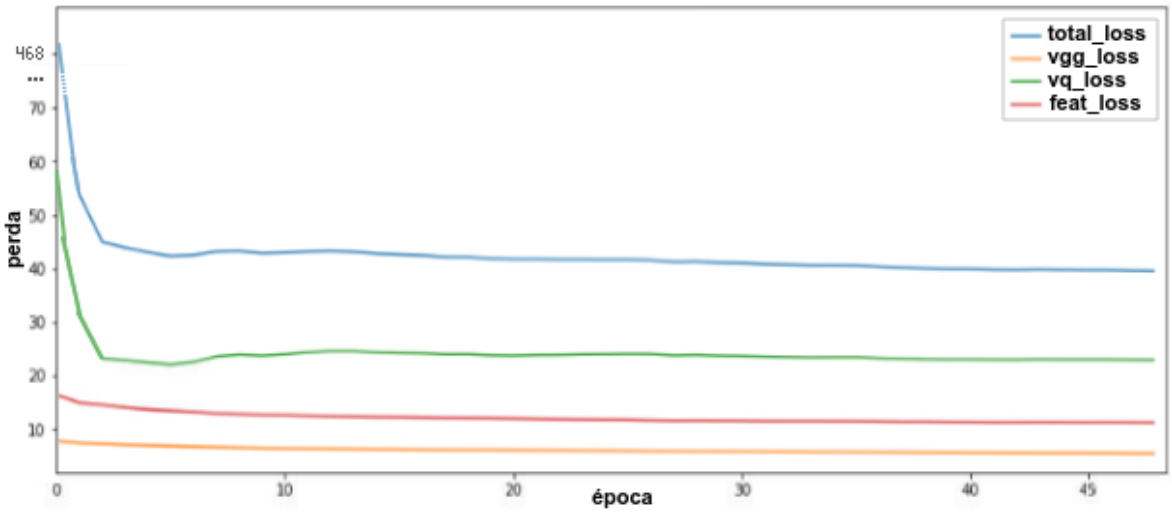


Figura 47: Perda durante o treino do modelo VQ-GAN com imagens de resolução 512 × 512.

O modelo VQ-GAN, treinado com imagens de maior resolução, revelou ser capaz de manter a qualidade das imagens geradas, relativamente às imagens geradas pelo modelo treinado com imagens de menor resolução. As imagens com resolução 512 × 512 geradas pelo VQ-GAN demonstram qualidade nos detalhes e uma coerência global superior às geradas por todos os outros modelos, com exceção do VQ-VAE. O ponto fraco destas imagens traduziu-se numa saturação da cor baixa, dando a sensação de existir uma ligeira névoa sobre as faces (figura 48)

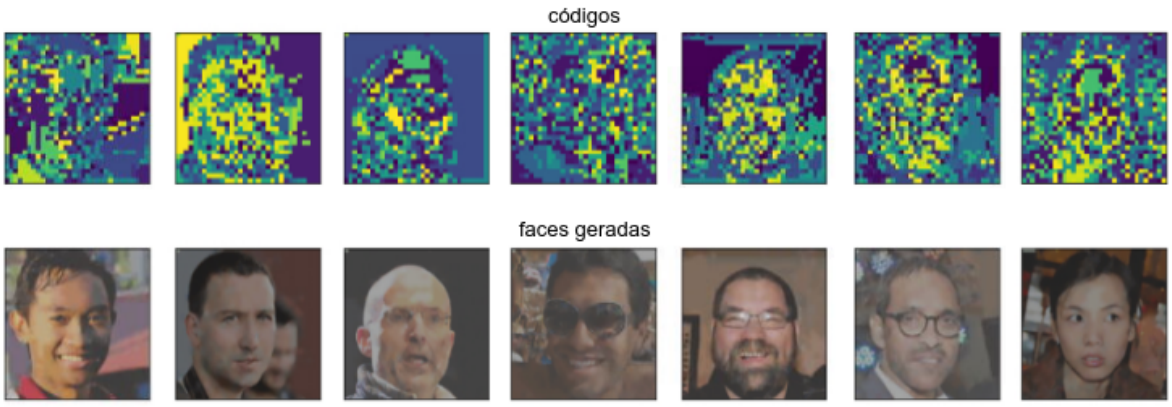


Figura 48: Geração de imagens de resolução 512 × 512 com o modelo VQ-GAN.

4.6 Avaliação Quantitativa dos Modelos

Todo o processo de desenvolvimento dos modelos foi efetuado com a versão 2.11.1 do `TensorFlow`, a versão 2.11.0 do `Keras`, numa máquina com sistema operativo `CentOS 9`

`Stream`, com um GPU NVIDIA GTX 1070 de 8GB de RAM, 64 GB de RAM e 20 cores *Intel Xeon*.

Além da avaliação qualitativa efetuada sobre os resultados obtidos com os modelos implementados, concretizada através da inspeção visual das imagens reconstruídas e geradas, também é importante avaliar quantitativamente o comportamento dos modelos, recorrendo a métricas objetivas. Deste modo, os modelos foram comparados tendo em consideração o seu tamanho, o tempo de treino e o tempo de inferência. Os valores destas métricas para todos os modelos experimentados encontram-se na tabela 6. Adicionalmente, os modelos foram ainda avaliados utilizando a métrica FID.

Modelo	**Número de parâmetros**	**Tempo de treino** (s)	**Tempo de inferência** (s)
VAE 128	6809139	25100	0.129
VAE 512	10000275	70700	0.142
VQVAE 128	3619971	45700	0.070
PixelCNN 128	20455424	18440	66.806
VQVAE 512	4347267	47940	0.091
PixelCNN 512	20455424	21340	72.320
VQGAN 128	3338115	103800	0.080
Transformer 128	2504320	86000	70.975
VQGAN 512	4354947	557350	0.126
Transformer 512	2504320	63960	69.925
VQVAE2 128	11719043	41080	0.029
PixelCNN superior 128	20455424	4460	15.432
PixelCNN inferior 128	20455424	25680	71.367
VQVAE2 512	7446083	59920	0.073
PixelCNN superior 512	20455424	11490	15.283
PixelCNN inferior 512	20455424	30000	70.496
GAN	30100868	362850	0.130
IntroVAE	47966723	22920	0.081

Tabela 6: Dados quantitativos relativos aos modelos implementados.

Primeiramente, e tal como seria previsível, verifica-se uma diferença no número de parâmetros entre os modelos que operam sobre imagens de resolução 128×128 e os que operam sobre imagens de resolução 512×512. Devido à restrição mais crítica enfrentada, imposta pelos 8GB de RAM do GPU utilizado no treino dos modelos, as versões dos modelos que foram implementadas para lidar com imagens de maior resolução não puderam ser tão escaladas (em largura e profundidade) quanto o necessário. Deste modo, a diferença entre os tamanhos dos modelos que operam com maior e menor resolução é bem menor do que deveria ser. Um caso paradigmático, resultante desta limitação, são os modelos VQ-VAE2, onde o modelo que opera sobre imagens menores tem mais parâmetros, em consequência

de o modelo que opera sobre imagens maiores ter sido fortemente restringido para poder ser treinado com o GPU disponível.

Uma das conclusões mais evidentes da análise dos resultados quantitativos recolhidos é o impacto que o tamanho das imagens tem no peso computacional necessário para treinar os modelos. Devido à restrição já mencionada, mais uma vez a diferença entre as métricas, obtidas paras as versões de 128×128 e 512×512 pixeis, não são tão diferentes como seria de esperar. Por exemplo, mesmo possuindo tempos de treino maiores e mais parâmetros, os tempos de inferência para os modelos treinados com as duas resoluções não apresentam diferenças significativas.

Com naturalidade, o modelo mais rápido, tanto no treino como na inferência, é o modelo concetualmente mais simples, ou seja, o VAE. Contrariamente, o modelo que demorou mais a treinar foi o VQ-GAN, essencialmente devido à maior complexidade do próprio processo de treino. Dado o elevado tempo de treino por época requerido, se o modelo GAN fosse treinado integralmente a partir do zero, ele seria o mais demorado a treinar. Como o treino integral da GAN foi abandonado, o modelo VQ-GAN acabou por ser o que exigiu mais tempo de treino. Relativamente ao tempo de interferência, o modelo com pior desempenho foi o VQ-VAE2, principalmente devido à estrutura hierárquica com dois níveis e com os dois PixelCNNs associados, que são responsáveis pela geração das amostras.

No topo das formas de avaliação de modelos geradores deve colocar-se a utilização de métricas específicas deste tipo de modelo. De acordo com o autor Borji (2018), é importante avaliar os prós e os contras das principais métricas, para optar por aquela que conduz a uma comparação mais correta dos diversos modelos.

Uma das métricas mais populares na avaliação de modelos geradores de imagens é a Inception Score (IS). Sucintamente, para obter esta métrica utiliza-se uma rede *Inception* pré-treinada, para classificar as imagens geradas e calcular a entropia das probabilidades de classe previstas. Quanto mais alto for o *Inception Score*, mais diversas e realísticas são as imagens geradas pelo modelo em avaliação.

Nos últimos anos, a FID consolidou-se como a métrica padrão na avaliação de modelos geradores. Baseada no *Inception Score*, também se utiliza uma rede *Inception* pré-treinada para obter os vetores de caraterísticas, tanto para as imagens geradas como para as imagens reais. O vetor de caraterísticas corresponde ao conjunto de valores resultantes da camada de saída da rede *Inception*. A métrica FID resulta do cálculo da distância de *Frechet* entre os dois vetores, o das imagens reais e o das imagens geradas. Quanto menor for a distância, maior é a semelhança entre as imagens geradas pelo modelo em avaliação e as imagens reais. A fórmula da FID é expressa pela equação 4.

$$d^2 = \|\mu_1 - \mu_2\|^2 + Tr(\sigma_1 + \sigma_2 - 2 \times \sqrt{\sigma_1 \times \sigma_2})) \tag{4}$$

Onde os vetores de caraterísticas das imagens reais e geradas são representados pela sua média (μ_1 e μ_2) e covariância (σ_1 e σ_1). O operador $Tr(M)$ refere-se à operação Traço da álgebra linear, que representa a soma dos elementos ao longo da diagonal principal da matriz M.

As métricas precisão e *recall* são basilares em várias outras áreas da computação, nomeadamente em aprendizagem automática. A precisão é a percentagem de amostras que o modelo classifica corretamente como positivas, de entre todas as amostras classificadas como positivas pelo modelo. O *recall* é a percentagem de amostras corretamente classificadas como positivas pelo modelo de entre todas as amostras verdadeiramente positivas.

A ideia fundamental por trás da utilização destas métricas com modelos geradores é que um modelo capaz de igualar a diversidade das imagens reais resultará num *recall* elevado e a geração de amostras realísticas resultará numa precisão alta.

Analisadas algumas das métricas de avaliação mais utilizadas com modelos geradores, e considerando novamente os recursos computacionais disponíveis, foi decidido avaliar os modelos apenas com uma métrica. A métrica escolhida foi a FID, pois é a métrica utilizada nos trabalhos similares ao nosso. A implementação da FID seguiu um exemplo disponibilizado pelo W&B (Thakur (2020)). Os resultados da métrica para os modelos implementados estão expostos na tabela 7.

Modelo	**FID** **(128 × 128)**	**FID** **(512 × 512)**
VAE	1469.5	132.7
VQVAE	1020.8	70.9
VQVAE2	2172.6	139.3
VQGAN	1843.4	81.8
GAN	4167647.4	—

Tabela 7: Métrica FID calculada para os modelos implementados.

Analisando os valores obtidos para a métrica FID, confirmam-se algumas das conclusões retiradas com base nos resultados qualitativos. Comparando o desempenho dos modelos para imagens de resolução (128 × 128) é possível constatar que o modelo VQ-VAE foi o que teve melhor comportamento, ou seja, é o que tem o menor valor de FID. Entre os modelos treinados com imagens de resolução 512 × 512, o modelo VQ-VAE apresentou novamente o melhor desempenho. Tendo em consideração o facto de não ter sido possível treinar de forma completa o modelo GAN e, observando os maus resultados qualitativos obtidos, seria de esperar que a GAN atingisse o pior resultado na métrica FID, como se comprovou. De notar também que a ausência de resultado da métrica FID para modelos GAN treinados com

imagens de resolução 512×512 deve-se ao facto das limitações de temporais e de memória da máquina terem impedido o treino deste tipo de modelos.

Da análise qualitativa dos modelos concluiu-se que o modelo VQ-VAE foi aquele que mais sobressaiu, obtendo imagens mais coerentes e detalhadas que os outros modelos. A partir dos resultados qualitativos, concluiu-se ainda que o modelo VQ-GAN foi o segundo melhor. Estas conclusões são confirmadas pelos valores da métrica FID, especialmente quando calculada para os modelos treinados com imagens de maior resolução (512×512), onde o VQ-VAE e o VQ-GAN se destacam dos demais. Com estes resultados é também possível concluir que a quantização de vetores nos modelos geradores, utilizada para representar o espaço latente de forma discreta, e que é um fator comum aos dois melhores modelos analisados, apresenta um grande potencial que vale a pena explorar.

5

CONCLUSÃO

O presente capítulo avalia o trabalho realizado durante a dissertação, de acordo com os objetivos inicialmente definidos para o trabalho a realizar. Identifica as limitações encontradas no decurso do trabalho, que dificultaram o atingir integral dos objetivos propostos. Também se apresentam ideias para trabalho futuro, que é possível realizar no seguimento do que foi explorado nesta dissertação, com base numa análise da margem de evolução do tema. Por último, é feita uma apreciação final ao trabalho realizado e a todo o processo de desenvolvimento do mesmo.

5.1 OBJETIVOS CONCRETIZADOS

No âmbito dos objetivos estabelecidos para esta dissertação, foram cumpridos quase todos os objetivos dependentes da implementação dos modelos pretendidos e a consequente análise dos resultados provenientes desses modelos. As exceções neste ponto foram a não implementação de um modelo GAN para imagens de maior resolução e a incapacidade de implementar um modelo IntroVAE capaz de resultados visualmente satisfatórios. Ademais, foram obtidos resultados a partir dos quais foi possível tirar conclusões sobre as potencialidades dos vários modelos geradores implementados, atingindo assim os resultados esperados. Por fim, concretizou-se o principal objetivo desta dissertação, o aprofundamento de conhecimentos sobre modelos geradores de imagens com faces humanas.

5.2 LIMITAÇÕES E TRABALHO FUTURO

A limitação que teve mais impacto no desenvolvimento desta dissertação foi a reduzida capacidade computacional disponível, especialmente ao nível dos GPUs. A dissertação foi integralmente feita sobre um único GPU, com 8GB de memória RAM, uma quantidade de memória notoriamente diminuta para treinar vários dos modelos geradores testados. Para além desta limitação, foi também necessário ultrapassar a falta de implementações de referência, para alguns dos modelos geradores avaliados, com a biblioteca `TensorFlow`, a

biblioteca adotada nesta dissertação. Atualmente, a maioria dos trabalhos de investigação nesta área científica implementa os modelos com a biblioteca `PyTorch`.

Um dos problemas identificados durante a implementação e teste dos modelos, especialmente presente quando se pretendeu usar imagens 4 vezes maiores em cada dimensão, foi a incapacidade de os modelos gerarem de forma consistente faces visualmente aceitáveis e a dificuldade dos modelos gerarem faces que fossem visualmente mais divergentes das faces com que foram treinados. Portanto, um dos pontos a explorar como trabalho futuro passaria por, ter acesso a mais GPUs e/ou GPUs com mais capacidade de memória, e escalar os modelos em largura e profundidade, de modo a dotar estes modelos de mais capacidade para gerar faces com alta resolução, consistentemente apelativas e diversificadas. Ainda no tópico da falta de recursos, seria também importante, futuramente, tentar otimizar o máximo possível o processo de treino dos modelos, assim como todo o processo de tratamento de dados computacionalmente pesados.

Acima de tudo, os resultados obtidos nesta dissertação mostram que, se com recursos limitados foi possível obter resultados satisfatórios, as grandes empresas da área, que dispõem de ambientes de investigação e desenvolvimento dedicados e com muito mais recursos humanos e tecnológicos, é possível atingir resultados excelentes e até mesmo revolucionários. O aparecimento de modelos de difusão, como o `Stabble Diffusion`, demonstraram ao público em geral o enorme potencial dos modelos geradores e reforçaram a validade do trabalho desenvolvido nesta dissertação.

O principal foco desta dissertação eram os modelos inspirados no VAE. Relativamente a estes modelos, foi possível identificar as suas melhores características e em que contextos poderão ser utilizados com maior sucesso. O ponto forte dos VAE é a sua capacidade de criar representações latentes, algo que os criadores do modelo `Stable Diffusion` identificaram e incorporaram neste modelo, permitindo assim explorar ao máximo as capacidades dos modelos de difusão. Esta ideia, provada pelo modelo `Stable Diffusion`, pode ser estendida a outros tipos de modelos existentes, em desenvolvimento, ou que apareçam no futuro, continuando a explorar as qualidades dos modelos VAE na geração de imagens. Este projeto permitiu também mostrar que a introdução, nos modelos VAE, da quantização dos vetores que representam o espaço latente é positiva e revela potencial para ser explorado em futuros modelos.

Outra questão a aprofundar em trabalhos posteriores será a utilização de maior quantidade e diversidade de métricas avaliadores da qualidade do modelos desenvolvidos. Neste projeto apenas foram analisadas três métricas, das quais apenas uma (FID) foi aplicada aos modelos implementados. Futuramente, seria interessante e útil para a contínua evolução da área de estudo que houvessem diversas formas de caracterizar e analisar as capacidades de cada modelo gerador desenvolvido.

5.3 APRECIAÇÃO FINAL

A plano de trabalho inicialmente elaborado para esta dissertação foi feito de forma ponderada e de acordo com os conhecimentos do momento. Contudo, no decorrer do trabalho foi necessário efetuar ajustes e alterações a esse planeamento, adaptando os objetivos propostos em face das dificuldades técnicas e tecnológicas encontradas ao longo do trabalho, reenquadrando o projeto no seu ambiente de desenvolvimento. Neste aspeto, foi fundamental a constante comunicação com o orientador da dissertação, por meio de reuniões semanais, que contribuíram para desbloquear vários dos obstáculos inerentes ao desenvolvimento de um projeto desta natureza.

O enorme crescimento nos últimos anos da área de Inteligência Artificial, e em particular dos modelos geradores, sobrepôs-se em boa parte ao período do desenvolvimento desta dissertação, o que serviu de grande motivação para continuar a explorar esta área de estudo. Além do mais, o estudo dos modelos VAE, bem como das suas variantes, proporcionou uma oportunidade de aprender as bases científicas desta área em rápida evolução.

BIBLIOGRAFIA

Ali Borji. Pros and Cons of GAN Evaluation Measures. 2018. URL https://arxiv.org/pdf/1802.03446.pdf.

François Chollet. DCGAN to generate face images. 2017. URL https://keras.io/examples/generative/dcgan_overriding_train_step.

François Chollet. Variational AutoEncoder. 2020. URL https://keras.io/examples/generative/vae.

Prafulla Dhariwal and Alex Nichol. Diffusion Models Beat GANs on Image Synthesis. 2021. URL https://arxiv.org/pdf/2105.05233.pdf.

Laurent Dinh, David Krueger, and Yoshua Bengio. NICE: Non-Linear Independent Components Estimation. 2015. URL https://arxiv.org/pdf/1410.8516.pdf.

Laurent Dinh, Jascha Sohl-Dickstein, and Samy Bengio. Improved Variational Inference with Inverse Autoregressive Flow. 2017. URL https://arxiv.org/pdf/1605.08803.pdf.

Patrick Esser, Robin Rombach, and Björn Ommer. Taming Transformers for High-Resolution Image Synthesis. 2021. URL https://arxiv.org/pdf/2012.09841.pdf.

Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative Adversarial Nets. 2014. URL https://arxiv.org/pdf/1406.2661.pdf.

Google. Google Colaboratory. 2017. URL https://colab.google.

Jonathan Ho, Ajay Jain, and Pieter Abbeel. Denoising Diffusion Probabilistic Models. 2020. URL https://arxiv.org/pdf/2006.11239.pdf.

Huaibo Huang, Zhihang Li, Ran He Zhenan, and Sun Tieniu Tan. IntroVAE: Introspective Variational Autoencoders for Photographic Image Synthesis. 2018. URL https://arxiv.org/pdf/1807.06358.pdf.

Jupyter. Jupyter. 2014. URL https://jupyter.org.

Tero Karras, Timo Aila, Samuli Laine, and Jaakko Lehtinen. Progressive Growing of GANs for Improved Quality, Stability, and Variation. 2018. URL https://arxiv.org/pdf/1710.10196.pdf.

Keras. Keras. 2015. URL https://keras.io.

Seonghyeon Kim. Implementation of Generating Diverse High-Fidelity Images with VQ-VAE-2 in PyTorch. 2020. URL https://github.com/rosinality/vq-vae-2-pytorch.

Diederik P. Kingma and Prafulla Dhariwal. Glow: Generative Flow with Invertible 1×1 Convolutions. 2018. URL https://arxiv.org/pdf/1807.03039.pdf.

Diederik P. Kingma and Max Welling. Auto-Encoding Variational Bayes. 2014. URL https://arxiv.org/pdf/1312.6114.pdf.

Diederik P. Kingma, Tim Salimans, Rafal Jozefowicz, Chen Xi, Ilya Sutskever, and Max welling. Improved Variational Inference with Inverse Autoregressive Flow. 2017. URL https://arxiv.org/pdf/1606.04934.pdf.

Apoorv Nandan. Text generation with a miniature GPT. 2020. URL https://keras.io/examples/generative/text_generation_with_miniature_gpt.

Charlie Nash, Jacob Menick, Sander Dieleman, and Peter W. Battaglia. Generating images with sparse representations. 2021. URL https://arxiv.org/pdf/2103.03841.pdf.

George Papamakarios, Theo Pavlakou, and Iain Murray. Masked Autoregressive Flow for Density Estimation. 2018. URL https://arxiv.org/pdf/1705.07057.pdf.

Sayak Paul. Google Colaboratory. 2021. URL https://keras.io/examples/generative/vq_vae.

Jialun Peng, Dong Liu, Songcen Xu, and Houqiang Li. Generating Diverse Structure for Image Inpainting With Hierarchical VQ-VAE. 2021. URL https://openaccess.thecvf.com/content/CVPR2021/papers/Peng_Generating_Diverse_Structure_for_Image_Inpainting_With_Hierarchical_VQ-VAE_CVPR_2021_paper.pdf.

Soumik Rakshit and Paul Sayak. GauGAN for conditional image generation. 2022. URL https://keras.io/examples/generative/gaugan.

Aditya Ramesh, Prafulla Dhariwal, Alex Nichol, Casey Chu, and Mark Chen. Hierarchical Text-Conditional Image Generation with CLIP Latents. 2022. URL https://cdn.openai.com/papers/dall-e-2.pdf.

Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser, and Björn Ommer. High-Resolution Image Synthesis with Latent Diffusion Models. 2022. URL https://arxiv.org/pdf/2112.10752.pdf.

Chitwan Saharia, William Chan, Mohammad Norouzi, Saurabh Saxena, Lala Li, Jay Whang, Jonathan Ho, David J. Fleet, Emily Denton, Seyed Kamyar Seyed Ghasemipour, Burcu Karagol Ayan, S. Sara Mahdavi, Rapha Gontijo Lopes, and Tim Salimans. Photorealistic Text-to-Image Diffusion Models with Deep Language Understanding. 2022. URL https://arxiv.org/pdf/2205.11487.pdf.

Marco Schreyer, Timur Sattarov, Anita Gierbl, Bernd Reimer, and Damian Borth. Learning Sampling in Financial Statement Audits using Vector Quantised Autoencoder Neural Networks. 2020. URL https://arxiv.org/pdf/2008.02528.pdf.

Jascha Sohl-Dickstein, Eric A. Weiss, Niru Maheswaranathan, and Surya Ganguli. Deep Unsupervised Learning using Nonequilibrium Thermodynamics. 2015. URL https://arxiv.org/pdf/1503.03585.pdf.

Jiaming Song, Chenlin Meng, and Stefano Ermon. Denoising Diffusion Implicit Models. 2021. URL https://arxiv.org/pdf/2010.02502.pdf.

Yang Song and Stefano Ermon. Generative Modeling by Estimating Gradients of the Data Distribution. 2020. URL https://arxiv.org/pdf/1907.05600.pdf.

Tensorflow. Tensorflow. 2015. URL https://www.tensorflow.org.

Ayush Thakur. How to Evaluate GANs using Frechet Inception Distance (FID). 2020. URL https://wandb.ai/ayush-thakur/gan-evaluation/reports/How-to-Evaluate-GANs-using-Frechet-Inception-Distance-FID---Vmlldzo0MTAxOTI.

Aäron van den Oord, Oriol Vinyals, Nal Kalchbrenner, Lasse Espeholt, Alex Graves, and Koray Kavukcuoglu. Conditional Image Generation with PixelCNN Decoders. 2016. URL https://arxiv.org/pdf/1606.05328v2.pdf.

Aäron van den Oord, Oriol Vinyals, and Koray Kavukcuoglu. Neural Discrete Representation Learning. 2018. URL https://arxiv.org/pdf/1711.00937.pdf.

Aäron van den Oord, Oriol Vinyals, and Ali Razavi. Generating Diverse High-Fidelity Images with VQ-VAE-2. 2019. URL https://arxiv.org/pdf/1906.00446.pdf.