UNIVERSIDADE FEDERAL DO RIO GRANDE - FURG

ENGENHARIA DE COMPUTAÇÃO

LINGUAGENS DE PROGRAMAÇÃO - TURMA U

Amanda Jorge Mendes - 149344

Eduardo Augusto Duarte Evangelista - 149382

Kauã Ortiz Silveira - 149367

João Victor Ferrari Mello do Amaral - 149356

GANAPP

Aplicativo para geração de imagens sintéticas

Rio Grande, RS

Julho/2023

SUMÁRIO

1 INTRODUÇÃO

Este trabalho descreve o desenvolvimento de um aplicativo que possibilita a geração de imagens sintéticas de diversas categorias, que compreendem desde animais até objetos do cotidiano. O aplicativo permite que o usuário capture uma foto com seu celular ou que ele use uma imagem de sua galeria. Essa imagem será enviada para uma *Application Programming Interface* (API) intermediária, a qual utiliza um serviço de inteligência artificial (IA), desenvolvido pelo grupo, responsável por processar a imagem. Primeiramente, a imagem é classificada dentre mil diferentes classes de objetos através de uma rede neural convolucional para classificação de imagens. Depois, uma GAN condicional [2] é empregada para produzir uma imagem sintética dessa mesma classe.

De forma sucinta, a tecnologia empregada no aplicativo de celular é Java. Para a API intermediária, a linguagem de programação utilizada foi Golang, ao passo que o serviço de IA foi desenvolvido em *JavaScript*.

2 METODOLOGIA

A aplicação desenvolvida possui três principais camadas, conforme comentado anteriormente. Essa divisão foi assim feita pois para atingir o resultado desejado, foi preciso utilizar três linguagens de programação distintas, de forma que cada uma implementa uma camada do sistema. O diagrama principal do projeto pode ser visto abaixo, na Figura 1.

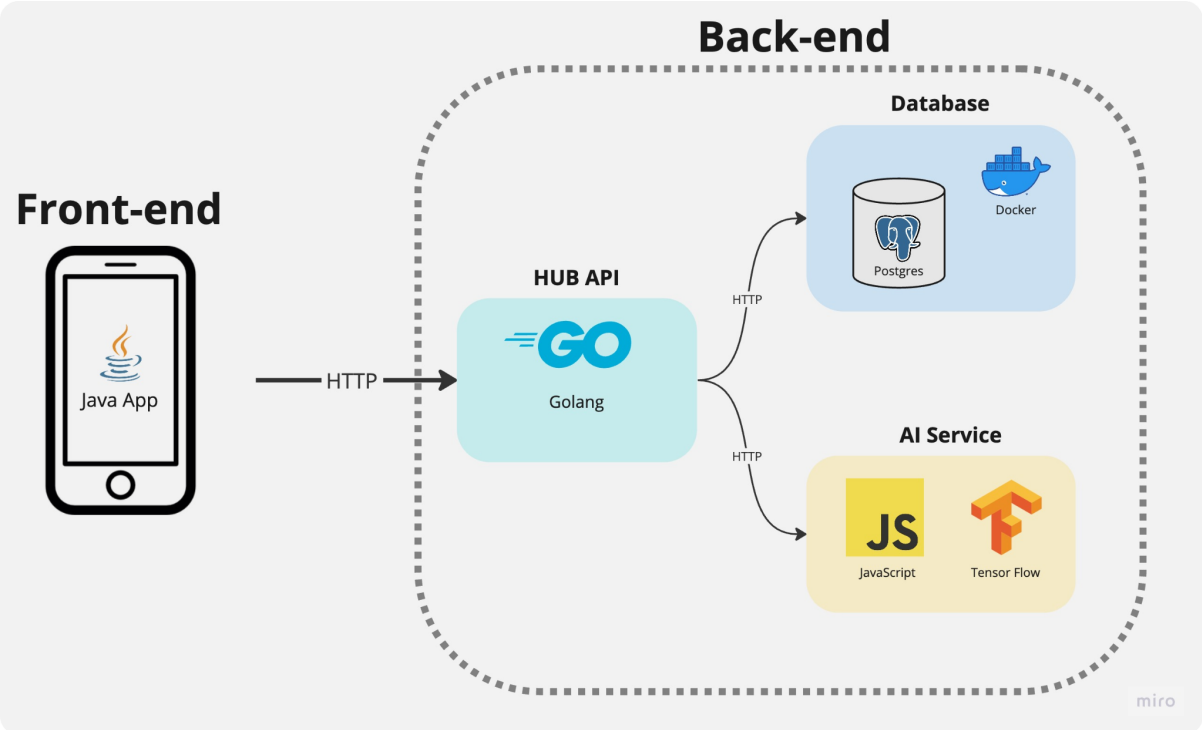

Figura 1. Diagrama estrutural da aplicação

Como pode ser observado, o *Back-end* da aplicação envolve a API em Golang, o serviço de Inteligência Artificial e uma instância do Docker que roda uma imagem Postgres, responsável por guardar as informações dos usuários. Do lado do *Front-end*, o aplicativo é a interface com o usuário final, ou seja, tudo que ele vê é responsabilidade do código escrito

em Java. Um detalhe que vale destacar é que o módulo “Database” não foi considerado como uma camada do sistema, pois ele acaba fazendo parte da API central.

Antes de explicarmos como cada um dos três módulos funciona e foi desenvolvido, um fluxo será dado como exemplo: sempre que o usuário deseja classificar e obter uma imagem sintética, ele envia a imagem original para a API em Golang. Então, a API comunica com o serviço de AI, o qual classifica a imagem original e gera a imagem sintética. A API central, uma vez que o processamento do serviço de AI ocorreu de forma correta, armazena, no banco de dados, qual a imagem sintética que foi gerada para o usuário (depois podemos retornar o histórico de imagens do usuário). Por fim, a API retorna, após salvar no banco de dados, a imagem sintética e a classificação, para que estes dados sejam exibidos em tela.

Para finalizar a introdução das metodologias, cabe destacar dois pontos. Primeiro, todas as comunicações entre os módulos são realizadas por meio do protocolo HTTP e segundo, a API é responsável, também, por prover outros recursos, que nem sempre envolvem o serviço de AI. A respeito destes outros fluxos, será dada uma explicação na subseção ***Back-End* em *Go***.

2.1 Módulo de Inteligência Artificial (IA) em *JavaScript*

O módulo de inteligência artificial foi desenvolvido na linguagem *JavaScript,* uma vez que ela conta com uma biblioteca chamada *TensorFlow.js,* que disponibiliza ferramentas necessárias para utilizar os modelos de classificação e de geração de imagens. A sequência de operações que permite a geração de uma imagem sintética de uma dada classe é descrita na Figura 2.

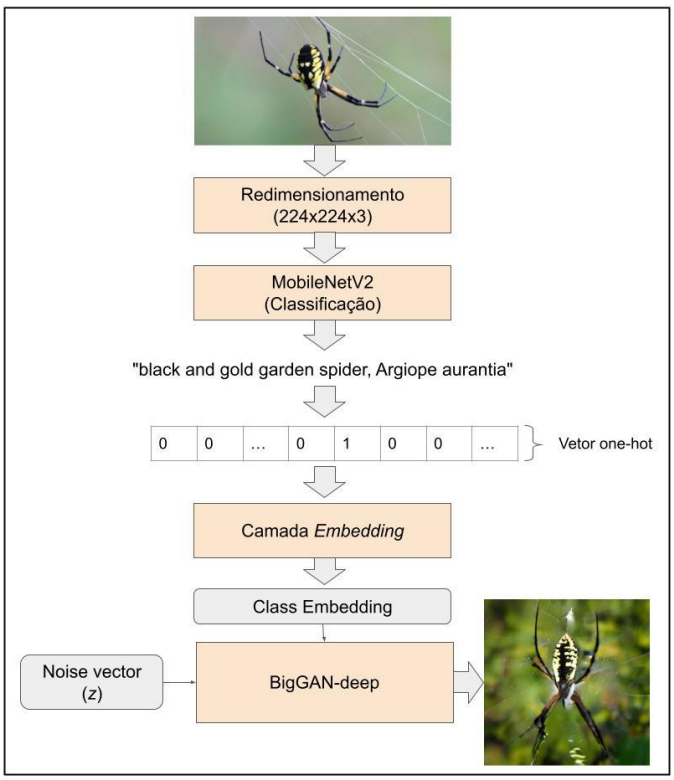

Figura 2. Sequência de operações do módulo de IA.

Primeiramente, a imagem selecionada pelo usuário é redimensionada para o tamanho requerido pela rede neural de classificação de imagens. Após a imagem ser classificada, é formado um vetor com codificação *one-hot* que representa a categoria da imagem. A representação *one-hot* consiste em um vetor em que cada posição corresponde a uma classe. Ele é formado pelo valor 1 na posição da classe de interesse e 0 nas demais. Esse vetor *one-hot* é transformado, através de uma camada de *embedding,* em um outro vetor que representa a classe de forma contínua. O vetor *embedding* que representa a classe e um vetor de ruído, denominado *z*, são passados como entrada para a rede generativa que produz as imagens sintéticas.

A rede neural convolucional (CNN) usada para a classificação de imagens é a *MobileNetV2* [2], a qual foi treinada no *dataset ImageNet* [3]. Esse *dataset* conta com mil classes de diversas categorias, que podem ser conferidas neste *link*. Para a arquitetura utilizada, é necessário fornecer uma imagem no formato 224x224x3. Como saída, ela retorna um vetor de tamanho 1000, representando as 1000 classes do *ImageNet*.

Já a rede responsável por gerar uma imagem sintética dada uma determinada classe, é uma *BigGAN-deep* [4], que é uma arquitetura de GAN condicional. Essa rede, assim como a de classificação, foi previamente treinada com o *dataset ImageNet. Conditional Generative Adversarial Networks* (cGANs) [1] são modelos de inteligência artificial que, assim como as GANs convencionais, são compostos por um gerador e um discriminador. O gerador é uma rede neural responsável por produzir uma imagem que se assemelha a uma imagem natural. Já o discriminador é uma rede neural que deve aprender a distinguir imagens reais de imagens produzidas pelo gerador. Essas redes são treinadas em conjunto para obter bons resultados na geração de imagens sintéticas.

GANs convencionais, normalmente, recebem como entrada apenas um vetor de ruído, usualmente denominado como *z,* que permite que a rede seja não determinística, ou seja, que não produza sempre os mesmos resultados. Já as GANs condicionais recebem, além do vetor *z*, uma informação auxiliar que condiciona a saída. No caso da *BigGAN-deep* essa informação é um vetor que representa uma determinada classe. O vetor *z*, representado na Figura 2, que é fornecido para a *BigGAN-deep*, é gerado a partir de uma distribuição normal truncada, ou seja, uma distribuição em que valores acima de um determinado *threshold* são substituídos. O nível de truncamento influencia diretamente nos resultados obtidos. Quanto maior esse valor, mais variadas e menos realistas serão as imagens geradas. Para este trabalho, o nível de truncamento utilizado é de 0.6.

A *MobileNet* foi obtida diretamente através do *TensorFlow Hub*, que é uma plataforma que contém diversos modelos de aprendizado de máquina desenvolvidos com *TensorFlow*. Já a *BigGAN-deep*, desde a camada que produz o vetor *embeddings* de uma dada classe até o gerador e o discriminador, não puderam ser obtidos diretamente do *TensorFlow Hub* por falta de compatibilidade. Então, primeiramente, um modelo implementado em *Python*, disponível no seguinte repositório do *GitHub*, foi convertido para o formato exigido pelo *TensorFlow.js*, usando uma função disponibilizada pela própria biblioteca. Depois desse processo, foram gerados os modelos necessários para realizar inferência em *JavaScript*.

2.2 Back-End em *Go*

A API intermediária, escrita na linguagem Golang, implementa diversas funcionalidades, como por exemplo utilizar o serviço de IA, controlar e armazenar as

imagens de cada usuário que está utilizando o app e prover todos os recursos que o aplicativo necessita para seu correto funcionamento. Essa camada da aplicação foi escrita em Goland por motivos de performance e velocidade de processamento, pois como se sabe, se trata de uma linguagem puramente compilada, de extrema eficiência.

Adentrando aspectos mais técnicos, o código em Golang é, na verdade, um servidor, que fica aguardando por requisições no protocolo HTTP por parte do aplicativo. Para armazenar dados de usuários e quais as imagens que cada um enviou para o processamento, um banco de dados Postgres foi utilizado. A Figura 3 mostra um diagrama estrutural mais detalhado da API central.

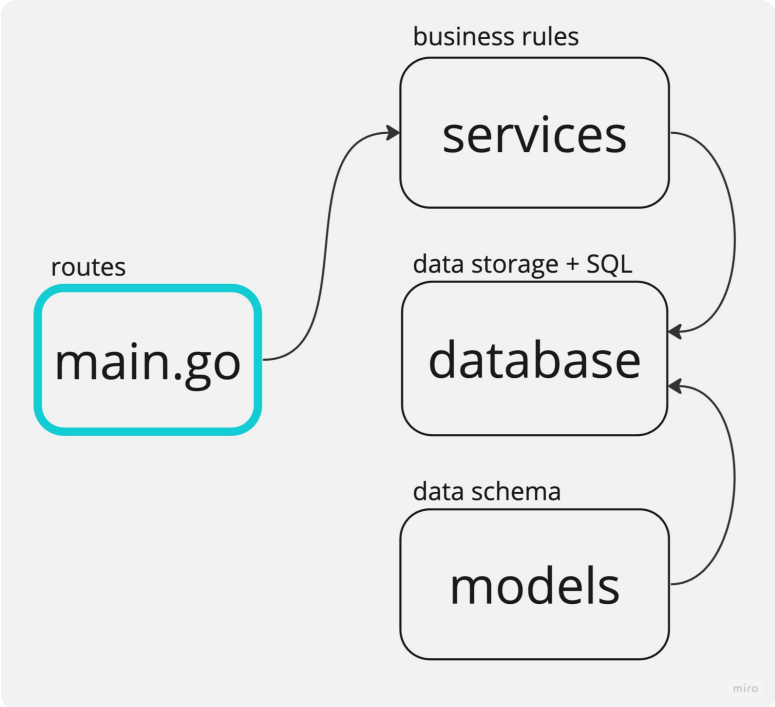

Figura 3. Diagrama estrutural da API em Golang.

O fluxo da API se inicia no arquivo principal, destacado na cor azul. Nele, são mapeados os *endpoints* da aplicação, de forma que sempre que o aplicativo desejar consumir alguma informação da API, o servidor implementado em main.go será o responsável por receber a requisição HTTP.

Em seguida, cada rota mapeada chama um método, implementado em Golang, que está dentro do pacote de serviços - *services*. Cada função desenvolvida implementa uma lógica e aplica regras de negócio para fazer o que deve ser realizado. Aqui são consumidos serviços como o de inteligência artificial.

Cada método da camada de serviço chama um método da próxima camada - *database*, responsável por acessar o banco de dados e realizar alguma operação nele. Esta é a camada final da aplicação, pois a camada *models* apenas guarda modelo de dados utilizados em *database*. Um ponto interessante de se destacar é que dentro desta camada há uma pasta chamada *migrations*, responsável por armazenar o histórico de alterações na estrutura do nosso banco de dados, o que garante compatibilidade e organização do projeto.

Com intuito de disponibilizar as funcionalidades implementadas ao aplicativo, quatro principais rotas (*endpoints*) foram implementados ao todo, conforme a listagem que segue.

- Rota `/createUser`, serve para criar um novo usuário
- Rota `/createUser`, serve para logar um usuário já existente na aplicação
- Rota `/getUserImages`, serve retornar todas as imagens sintéticas geradas para um usuário específico
- Rota `/createImage`, serve para classificar a imagem enviada pelo usuário e gerar uma imagem sintética. Este recurso utiliza o serviço de IA

2.3 Front-End em *Java*

Desenvolver uma aplicação *Android* utilizando Java traz consigo diversas vantagens que transitam entre compatibilidade, disponibilidade de bibliotecas e até mesmo para o suporte ao design de páginas, tudo isso pela característica de ser nativa ao desenvolvimento neste sistema operacional.

O *Front-end* da aplicação é um esforço conjunto tanto de Java quanto de XML para as definições de aparência estática de cada página. Para cada atividade, ou tela, temos um arquivo XML que realiza as devidas marcações das estilizações realizadas para auxiliar a manipulação posterior em Java. Na atual configuração do aplicativo temos duas atividades onde cada uma delas possui seu XML próprio e seu código Java para ser customizado e proporcionar uma experiência mais amigável ao usuário.

Ao iniciar o sistema somos encaminhados para a tela *MenuActivity* na qual se comporta como o menu principal do aplicativo. Nele somos capazes de acessar a câmera do aparelho, a galeria do *smartphone* e até mesmo uma galeria própria da aplicação, que também é uma atividade nova. Nela, ou melhor dizendo *HistoryActivity*, temos uma lista de imagens catalogadas, as quais passaram pelo processo de geração pela rede GAN.

Além das telas, temos duas classes próprias e auxiliares para realizar algumas animações de forma modularizada. A primeira delas - *AnimateWindow* - realiza a configuração da tela inicial, que parte de um design primitivo, até a aparência final da tela, na qual permite ao usuário final realizar a devida utilização do aplicativo. Além dela temos também a *WritingMachine*, na qual contém alguns métodos para realizar a escrita de textos em tela de forma que se assemelha à escrita de uma máquina antiga de escrever.

3 RESULTADOS

Os resultados deste trabalho apresentam o desenvolvimento de um aplicativo móvel capaz de gerar imagens sintéticas em diversas categorias. Todos os códigos utilizados estão no repositório do *GitHub*. A Figura 4 exibe as telas do aplicativo, destacando a experiência do usuário ao utilizá-lo. A primeira tela, a de menu, permite que os usuários acessem as diferentes funcionalidades do aplicativo, como capturar fotos ou selecionar imagens da galeria. A segunda tela mostra o histórico de imagens sintéticas geradas, proporcionando aos usuários uma visualização das suas criações juntamente com a sua classificação. Por fim, a terceira imagem ilustra a tela de visualização de resultados, onde é exibida uma imagem sintética gerada. Essas telas refletem a interface amigável do aplicativo e a capacidade de gerar imagens sintéticas de forma personalizada e

diversificada. Importante salientar que o sistema como um todo foi necessário trabalho em equipe, pois havia muitas integrações distintas.

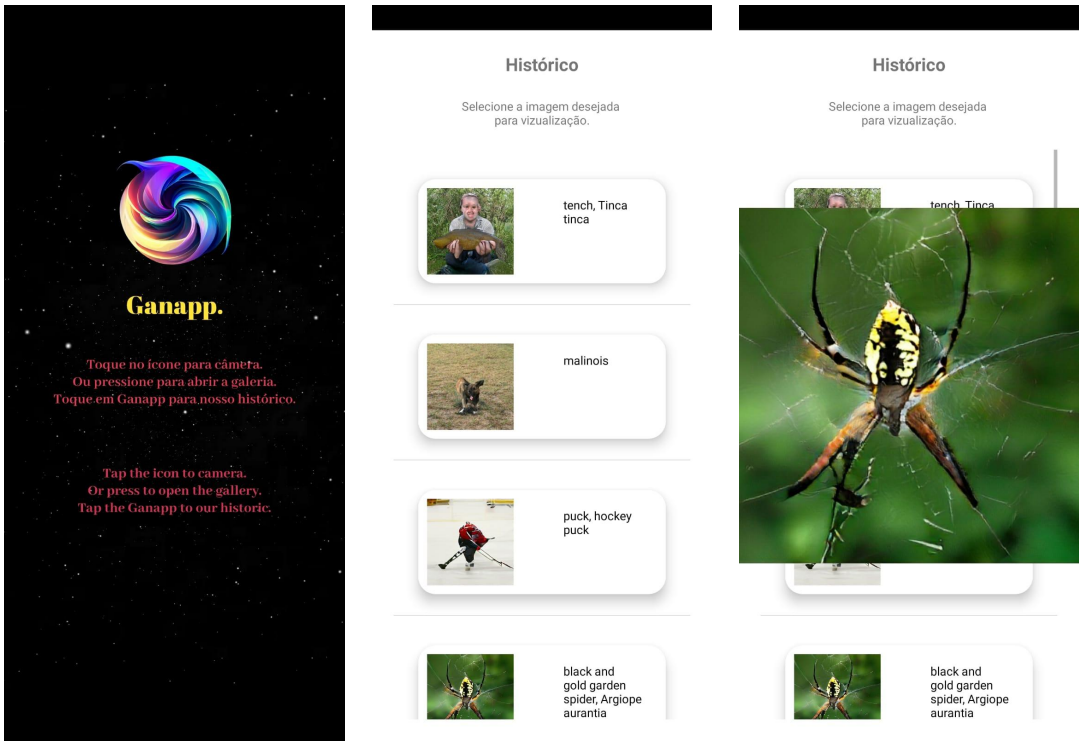

Figura 4. Telas do aplicativo e imagens geradas.

4 CONCLUSÃO

Em resumo, este trabalho descreve o desenvolvimento de um aplicativo móvel que permite a geração de imagens sintéticas em várias categorias. O aplicativo possibilita aos usuários capturar fotos ou selecionar imagens da galeria, utilizando uma API que utiliza redes neurais convolucionais e GANs condicionais para processar e gerar as imagens sintéticas. O back-end centralizador do aplicativo é desenvolvido em Golang, enquanto o front-end é implementado em Java. O resultado é um aplicativo intuitivo, funcional e criativo, oferecendo aos usuários a oportunidade de explorar a geração de imagens sintéticas de forma fácil e personalizada.

5 REFERÊNCIAS BIBLIOGRÁFICAS

[1] Mirza, Mehdi, and Simon Osindero. "Conditional generative adversarial nets." *arXiv preprint arXiv:1411.1784* (2014).

[2] Sandler, Mark, et al. "Mobilenetv2: Inverted residuals and linear bottlenecks." *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2018.

[3] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, “Imagenet: A large-scale hierarchical image database,” in 2009 IEEE conference on computer vision and pattern recognition. Ieee, 2009, pp. 248–255.

[4] Brock, Andrew, Jeff Donahue, and Karen Simonyan. "Large Scale GAN Training for High Fidelity Natural Image Synthesis." *International Conference on Learning Representations*. 2018.