Relatório Técnico

Estratégias, Metodologias, Arquiteturas e Resultados

Análise Exploratória de Dados

▼ Primeiros Passos

- Ao abrir a pasta data, foi visto que as imagens se encontravam todas no mesmo diretório.
- A diferença entre as imagens era o label _tree para imagens de Pinheiros, e _soil para imagens de solo.
- Para que fosse possível trabalhar com as classes separadas, foi criado duas pastas no diretório Data:
  - tree
  - soil

A partir disso, foi executado os seguintes comandos no git bash:

```
$ mv *tree* tree/
```

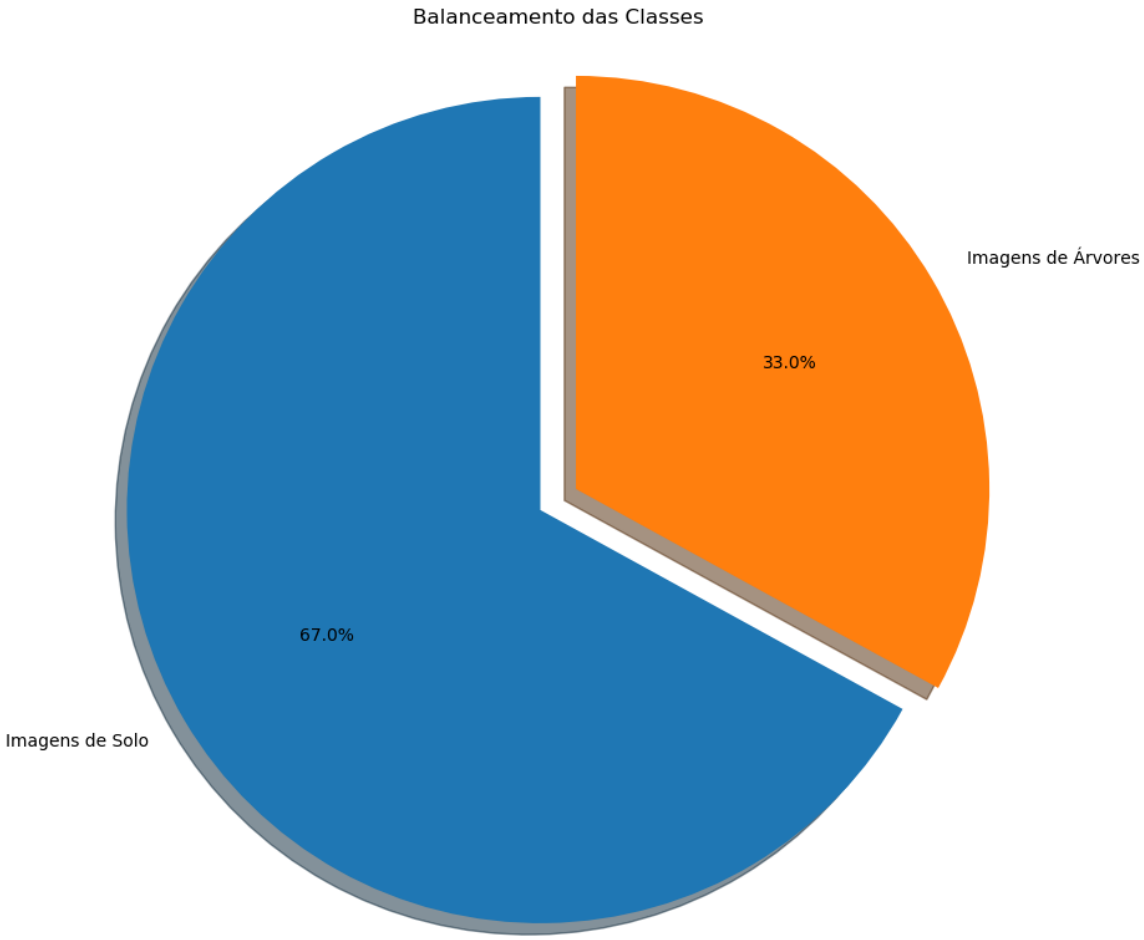
```
$ mv *soil* soil/
```

▼ Balanceamento das Classes

Com as classes separadas, foi possível verificar o número de imagens de cada classe:

- 2224 imagens de Árvores
- 4448 imagens de Solo

Através desses dados foi gerado o gráfico de balanceamento das Classes:



Com isso conclui-se que o dataset está desbalanceado. Provavelmente se utilizado dessa maneira, os modelos terão uma tendência em predizer com melhor acurácia as imagens da classe solo.

▼ Comportamento dos Modelos de Redes Neurais Convolucionais no Dataset Proposto

Para encontrar a melhor arquitetura a ser utilizada nesse conjunto de imagens, foi utilizado três diferentes arquiteturas de redes neurais convolucionais presentes na biblioteca *keras.models* do framework *Keras*.

▼ Sumário das Arquiteturas

▼ VGG16

Camada(tipo)	Formato da Saída	Nº Param
input_1 (InputLayer)	[(None, 50, 50, 3)]	0
block1_conv1 (Conv2D)	(None, 50, 50, 64)	1792
block1_conv2 (Conv2D)	(None, 50, 50, 64)	36928
block1_pool (MaxPooling2D)	(None, 25, 25, 64)	0
block2_conv1 (Conv2D)	(None, 25, 25, 128)	73856
block2_conv2 (Conv2D)	(None, 25, 25, 128)	147584
block2_pool (MaxPooling2D)	(None, 12, 12, 128)	0
block3_conv1 (Conv2D)	(None, 12, 12, 256)	295168
block3_conv2 (Conv2D)	(None, 12, 12, 256)	590080
block3_conv3 (Conv2D)	(None, 12, 12, 256)	590080
block3_pool (MaxPooling2D)	(None, 6, 6, 256)	0

block4_conv1 (Conv2D)	(None, 6, 6, 512)	1180160
block4_conv2 (Conv2D)	(None, 6, 6, 512)	2359808
block4_conv3 (Conv2D)	(None, 6, 6, 512)	2359808
block4_pool (MaxPooling2D)	(None, 3, 3, 512)	0
block5_conv1 (Conv2D)	(None, 3, 3, 512)	2359808
block5_conv2 (Conv2D)	(None, 3, 3, 512)	2359808
block5_conv3 (Conv2D)	(None, 3, 3, 512)	2359808
block5_pool (MaxPooling2D)	(None, 1, 1, 512)	0
flatten (Flatten)	(None, 512)	0
dropout (Dropout)	(None, 512)	0
dense (Dense)	(None, 2)	1026

Total params: 14,715,714
Trainable params: 1,026
Non-trainable params: 14,714,688

▼ ResNet50

Camada(tipo)	Formato da Saída	Nº Param
resnet50 (Functional)	(None, 2, 2, 2048)	23587712
flatten (Flatten)	(None, 8192)	0
batch_normalization (BatchNo	(None, 8192)	32768

dense (Dense)	(None, 256)	2097408
dropout (Dropout)	(None, 256)	0
batch_normalization_1 (Batch	(None, 256)	1024
dense_1 (Dense)	(None, 128)	32896
dropout_1 (Dropout)	(None, 128)	0
batch_normalization_2 (Batch	(None, 128)	512
dense_2 (Dense)	(None, 64)	8256
dropout_2 (Dropout)	(None, 64)	0
batch_normalization_3 (Batch	(None, 64)	256
dense_3 (Dense)	(None, 2)	130

Total de Parâmetros: 25,760,962
Parâmetros Treináveis: 17,131,970
Parâmetros não Treináveis: 8,628,992

▼ Inception V3

Camada(tipo)	Formato da Saída	Nº Param
module_wrapper (ModuleWrappe	(1, 1, 1, 2048)	21802784
global_average_pooling2d (Gl	(1, 2048)	0
flatten_1 (Flatten)	(1, 2048)	0
batch_normalization_4 (Batch	(1, 2048)	8192

dense_4 (Dense)	(1, 256)	524544
dropout_3 (Dropout)	(1, 256)	0
batch_normalization_5 (Batch	(1, 256)	1024
dense_5 (Dense)	(1, 128)	32896
dropout_4 (Dropout)	(1, 128)	0
batch_normalization_6 (Batch	(1, 128)	512
dense_6 (Dense)	(1, 64)	8256
dropout_5 (Dropout)	(1, 64)	0
batch_normalization_7 (Batch	(1, 64)	256
dense_7 (Dense)	(1, 2)	130

Total de Parâmetros 22,378,594
Parâmetros treináveis: 570,818
Parâmetros não treináveis: 21,807,776

Todas arquiteturas foram carregadas com transferência de aprendizado baseada no dataset I*mageNet*. As arquiteturas tiveram suas camadas convolucionais impedidas de serem treinadas, para que apenas as camadas totalmente conectadas adicionadas no fim do modelo possam ter seus pesos modificados.

De maneira a evitar o *overfitting* de treino do modelo, as camadas totalmente conectadas tiveram a adição de *Dropout* para que pesos aleatórios da camada anterior sejam ignorados, gerando assim camadas com pesos ajustados para que se compense a desativação dos outros nodos.

Nas arquiteturas nas arquiteturas ResNet50 e Inception V3 foi adicionado Batch Normalization. Dessa maneira foi possível adicionar mais camadas densas nas arquiteturas, trazendo regularização para os pesos da rede, tornando os resultados mais generalizáveis

Na arquitetura VGG16 foi utilizada Regularização l2, mantendo os valores dos pesos em escala reduzida.

Para que fosse possível comparar os resultados entre as arquiteturas, alguns dos parâmetros de treino foram mantidos:

▼ Parâmetros Mantidos entre as Arquiteturas

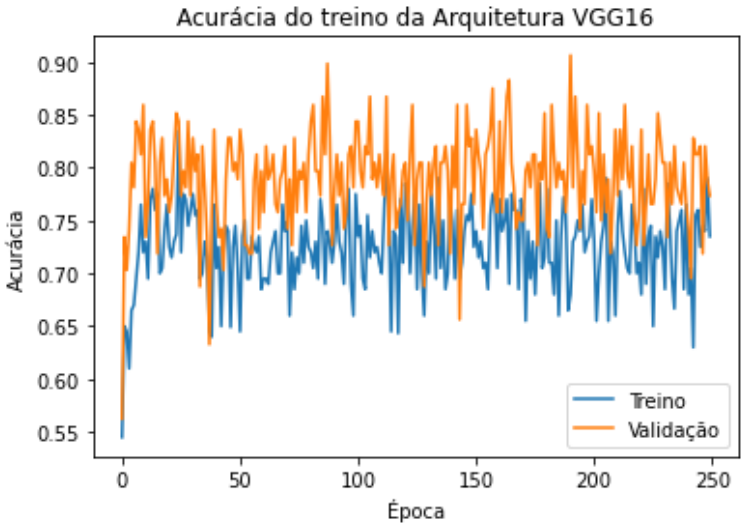
- Épocas : 250
- Otimizador : Adam
- Taxa de Aprendizagem: 0.01
- Função de ativação da camada de saída: Sigmoid
- Função loss: Categorical Cross Entropy
- Métricas Avaliadas no treino: ['categorical_accuracy']
- Checkpoint: val_categorical accuracy.

Durante o treinamento, foi utilizado *Callbacks* de maneira a obter logs dos treinos, e também salvar os pesos do modelo que possuem a melhor acurácia obtida durante a etapa de validação que ocorre no final de cada época. Abaixo está listado os resultados de cada arquitetura.

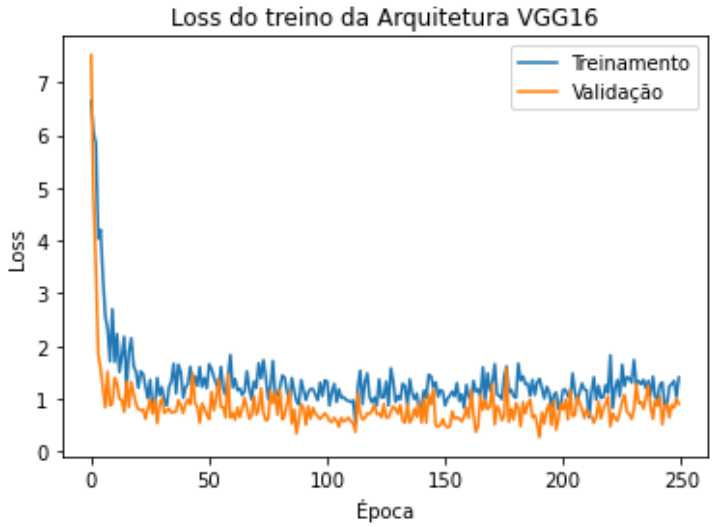
▼ Resultados das Arquitetura VGG16

▼ Gráficos do Treino

▼ Gráfico da Acurácia



▼ Gráfico da Loss



▼ Melhores Épocas do Treino

Época com Maior Acurácia de Treino

Aa Época	# Categorical_Accuracy	# Loss	# Val_Categorical_Accuracy	# Val_Loss
24	0.835	1.008353	0.84375	0.812352

Época com Maior Acurácia de Validação

Aa Época	# Categorical_Accuracy	# Loss	# Val_Categorical_Accuracy	# Val_Loss
190	0.68	1.380635	0.90625	0.272937

Época com Menor Loss do Treino

Aa Época	# Categorical_Accuracy	# Loss	# Val_Categorical_Accuracy	# Val_Loss
112	0.815	0.524884	0.867188	0.374707

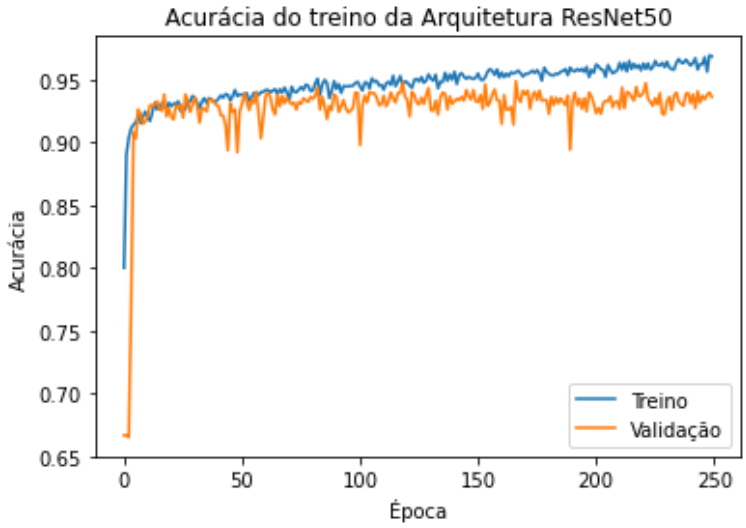
Época com Menor Loss de Validação

Aa Época	# Categorical_Accuracy	# Loss	# Val_Categorical_Accuracy	# Val_Loss
190	0.68	1.380635	0.90625	0.272937

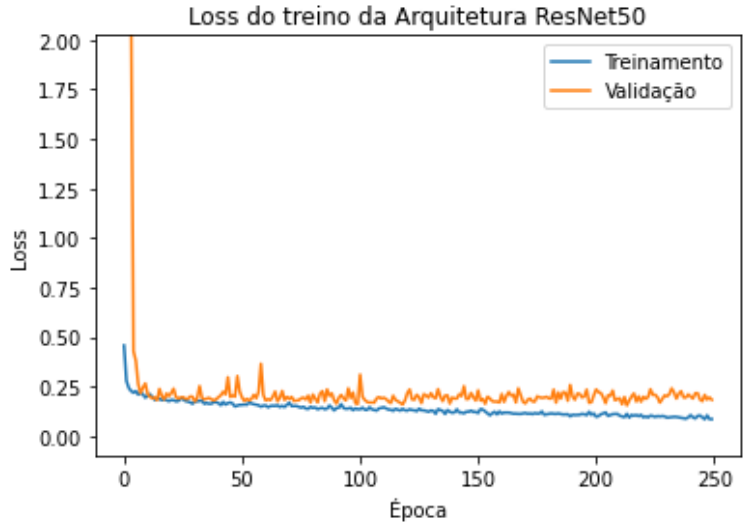
▼ Resultados da Arquitetura ResNet50

▼ Gráficos do Treino

▼ Gráfico da Acurácia



▼ Gráfico da Loss



▼ Melhores Épocas do Treino

Época com Maior Acurácia de Treino

Aa Época	# Categorical_Accuracy	# Loss	# Val_Categorical_Accuracy	# Val_Loss
248	0.969171	0.086476	0.93953	0.195731

Época com Maior Acurácia de Validação

Aa Época	# Categorical_Accuracy	# Loss	# Val_Categorical_Accuracy	# Val_Loss
166	0.955684	0.114065	0.948526	0.170189

Época com Menor Loss do Treino

Aa Época	# Categorical_Accuracy	# Loss	# Val_Categorical_Accuracy	# Val_Loss
246	0.967673	0.085755	0.934033	0.208583

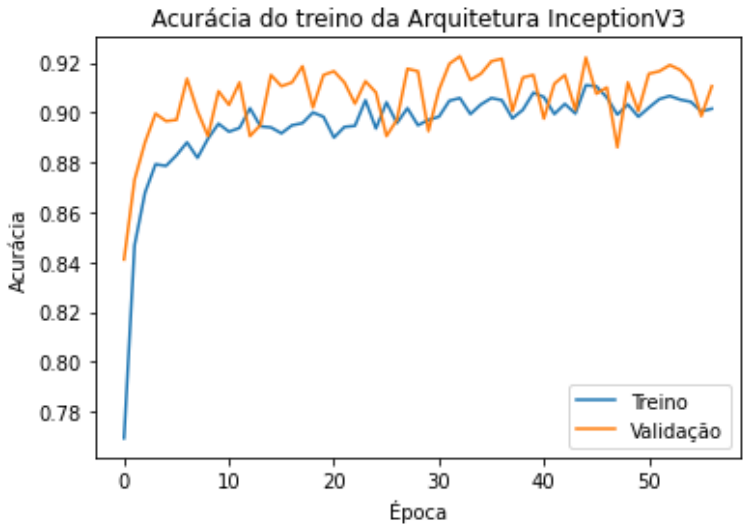
Época com Menor Loss de Validação

Aa Época	# Categorical_Accuracy	# Loss	# Val_Categorical_Accuracy	# Val_Loss
213	0.956326	0.113088	0.94003	0.155527

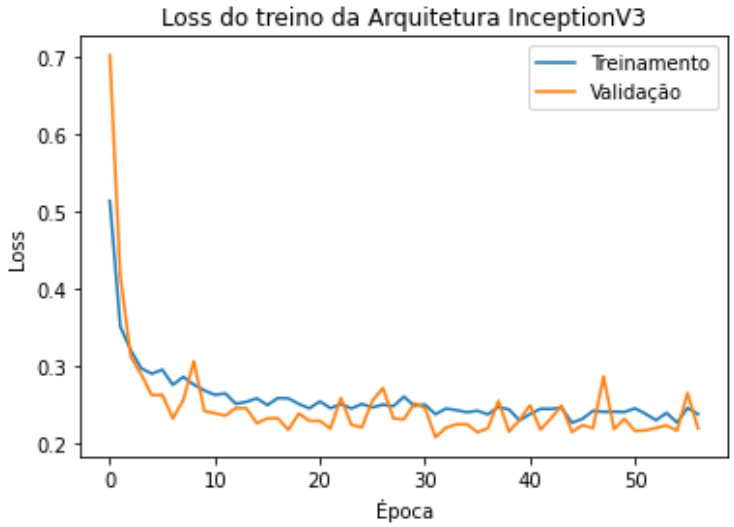
▼ Resultados da Arquitetura InceptionV3

▼ Gráficos do Treino

▼ Gráfico da Acurácia



▼ Gráfico da Loss



▼ Melhores Épocas do Treino

Época com Maior Acurácia de Treino

Aa Época	# Categorical_Accuracy	# Loss	# Val_Categorical_Accuracy	# Val_Loss
44	0.91094	0.226084	0.922039	0.214166

Época com Maior Acurácia de Validação

Aa Época	# Categorical_Accuracy	# Loss	# Val_Categorical_Accuracy	# Val_Loss
32	0.905802	0.2445	0.922539	0.219715

Época com Menor Loss do Treino

Aa Época	# Categorical_Accuracy	# Loss	# Val_Categorical_Accuracy	# Val_Loss
44	0.91094	0.226084	0.922039	0.214166

Época com Menor Loss de Validação

Aa Época	# Categorical_Accuracy	# Loss	# Val_Categorical_Accuracy	# Val_Loss
31	0.904731	0.237154	0.91954	0.207315

Na tabela abaixo, é comparado os resultados de acurácia de treino, acurácia de validação, loss do treino e loss da validação entre as épocas das três arquiteturas utilizadas na estratégia elaborada para o desafio.

Comparação Entre Arquiteturas

Aa Arquitetura	# Maior Acurácia do Treino	# Melhor Acurácia de Validação	# Menor Loss do Treino	# Menor Loss da Validação
VGG16	0.84375	0.90625	0.524884	0.272937
ResNet50	0.969171	0.948526	0.085755	0.155527
InceptionV3	0.91094	0.922539	0.226084	0.207315

De maneira a escolher a melhor arquitetura a ser utilizada para a elaboração da aplicação de predição de imagens, foi analisado qual arquitetura teve os melhores reesultados. Calculando o máximo valor da Melhor Acurácia do Treino e Melhor Acurácia de Validação, e calculando também o mínimo valor de Loss de Treino e Loss de Validação, foi constado que os melhores resultados para as quatro categorias se encontra na Arquitetura ResNet50. Dessa maneira essa arquitetura foi a escolhida para dar continuidade com as elaborações de aplicações para predição de imagem.

Elaboração das Aplicações

As aplicações seguem o mesmo padrão entre as funções que compõem seus códigos, mas diferem nos argumentos passados como entrada de cada tipo de aplicação.

As aplicações precisam que o modelo escolhido na análise exploratória de dados seja carregado. O método l*oad_model* da biblioteca *Keras* é chamado para que seja carregado o modelo ResNet50 treinado.

A função de classificação em cada aplicação é responsável por receber a imagem carregada e tratá-la para que se adeque a função de predição. Após usar o *model.predict* com a imagem tratada, é obtido um *array* com as probabilidades de cada classe. Para obter a maior probalidade dentre as classes "Pinheiro e "solo", é utilizado o método *argmax* da biblioteca *numpy,* retornando o valor 0 para "solo" e o valor 1 para "pinheiro".

Abaixo está listado as diferenças de cada aplicação.

▼ Aplicação Gráfica

Para fazer a aplicação gráfica, onde fosse possível carregar uma imagem do computador na própria aplicação e receber como resultado a predição da classe, foi utilizado a biblioteca tk, como sugerido nesse tutorial.

A função para carregar as imagens onde é feito o tratamento para que se adequem a para exibição na janela da aplicação.

O objeto do tipo tk é criado, onde são passados os parâmetros da Interface gráfica, como o título, ícone e subtítulos.

Os botões são criados, e são passados como parâmetros o tamanho, a cor e o método que o botão chama.

Com a função *mainloop*, é possível refazer o processo de carregar uma nova imagem e classificá-la.

▼ Aplicação em Linha de Comando

A aplicação em linha de comando recebe a imagem que foi passada como argumento na chamada do programa no terminal.

O método *argv* da biblioteca *sys* é chamado para que o argumento passado seja utilizado para carregar a imagem utilizando a biblioteca *pillow*.

Após tratada a imagem, o método de predição é chamado, e o resultado é impresso no terminal.

▼ Aplicação Web

A aplicação Web utiliza a biblioteca *Flask* para criar uma rest API responsável por criar um server que recebe r*equests* do tipo *POST,* permitindo que arquivos de imagem sejam enviados para o *endpoint* definido no código.

Para que o modelo não seja carregada a cada *request*, é criado uma função específica para carregá-lo, de maneira a que seja chamado na inicialização do *server,* antes que a aplicação seja inicializada.

Dificuldades Encontradas

Quando analisadas as imagens presentes na pasta Data, foi preciso buscar alternativas que separassem elas de acordo com as classes escritas no nome de cada arquivo. Após procura, foi encontrado o método pelo bash.

Para se adequar ao Bitbucket, foi preciso baixar e conhecer a ferramenta SourceTree, de maneira a enviar os commits gerados localmente para o server do desafio.

Como o computador disponível não possui GPU adequada para rodar modelos grandes de redes neurais convolucionais, foi preciso rodar os códigos no ambiente Google Colab. Os códigos e resultados da saída das redes foram commitados no bitbucket do desafio.

Após analisar os resultados das redes, partiu-se para o desenvolvimento das aplicações. A primeira aplicação foi a gráfica, utilizando esse tutorial, foi desenvolvido a aplicação gráfica de predição. Mas ao tentar criar um executável, a biblioteca *pyinstaller* não gerou o executável devido a dependências de bibliotecas.

A aplicação web foi desenvolvida de maneira de a utilizar os conhecimentos aduiridos da biblioteca Flask. Tentou-se fazer o deploy no site heroku, mas novamente houve problemas devido a dependências de bibliotecas, mais especificamente com a biblioteca Pillow. Com slug size reduzido, não foi possível fazer o upload da versão mais atual do Tensorflow.

Conclusões

Com o desafio Pix Force de visão computacional, foi possível colocar em prática, os conceitos de redes neurais convolucionais aprendidos durante os anos de estudos da Universidade. Foi possível colocar em prática novos conceitos que foram necessários serem aprendidos para colocarem as aplicações em prática. Portanto, o desafio foi uma ótimo exercício de aprendizado de atividades diárias para desenvolver aplicações de *Machine Learning*.