SEGMENTAÇÃO MULTI-CLASSES DE DADOS GEOFÍSICOS DE ALTA RESOLUÇÃO (SSS) UTILIZANDO *DEEP LEARNING*

Aluno: Cicero Pereira Batista Junior

Orientador: Leonardo Forero Mendoza

Data: 04/01/2021

Trabalho apresentado no curso BI MASTER como pré-requisito para conclusão de pós-graduação e obtenção de crédito na disciplina "Projetos de Sistemas Inteligentes de Apoio à Decisão".

RESUMO

No presente trabalho foram aplicadas e avaliadas técnicas de *Deep Learning* que permitem a segmentação de imagens acústicas submarinas, oriundas dos dados geofísicos de alta resolução (sonar de varredura lateral – SSS), em duas classes distintas: alvo refletivo e fundo lamoso.

A segmentação de imagem foi baseada em Redes Neurais Totalmente Convolucionais (*fully convolutional network*) do tipo *U-Net*.

OBJETIVOS DO TRABALHO

Tendo como *inputs* os dados de SSS, a utilização de modelos de *Deep Learning* objetiva o aperfeiçoamento e automatização do mapeamento de feições naturais no leito marinho, através da segmentação de imagens, que atualmente é realizada de maneira manual.

OPORTUNIDADES E BENEFÍCIOS

A utilização de técnicas de *Deep Learning* têm sido amplamente utilizadas na área da geofísica de exploração de hidrocarbonetos (Fase Exploratória) e tem sido fundamental para aprimorar os resultados.

Diferentemente da grande aplicação dessas técnicas na área da geofísica de exploração, nota-se carência na utilização desses métodos em outras áreas da geofísica, como a que iremos tratar nesse trabalho, que é a geofísica de alta resolução para atendimento a Projetos de Engenharia Submarina relacionadas a Projetos de Desenvolvimento da Produção de Óleo e Gás.

Nesse sentido, a proposta desse trabalho se destina aplicar Redes Neurais Totalmente Convolucionais (*fully convolutional network),* do tipo *U-Net,* em dados geofísicos de alta

resolução, mais especificamente o SSS, para otimização no tempo de execução do mapeamento de feições naturais no leito marinho.

RECURSOS COMPUTACIONAIS

Recursos de *Software*

O desenvolvimento do projeto foi baseada na linguagem de programação *Python*.

Para o carregamento, pré-processamento e visualização dos dados foram utilizadas principalmente as bibliotecas *tifffile* e *scikit-learn*, além das tradicionais *matplotlib*, *numpy*, *skimage* e *opencv*.

Para a divisão dos dados de treinamento, teste e validação, bem como a implementação das redes neurais e verificação de seus desempenhos foram utilizadas o *tensorflow*, *keras* e *scikit-learn.*

Recursos de *Hardware*

Inicialmente o projeto foi desenvolvido na plataforma *online* gratuita chamada *Google Colaboratory*, porém, durante os testes iniciais, percebeu-se que a GPU dessa plataforma não possuía memória suficiente para carregar a imagem original, nem depois do redimensionamento da mesma. Além disso, outras operações não foram possíveis, como por exemplos, a normalização dos dados e divisão em *patchs*.

Devido a essa restrição, o projeto final foi implementado em uma máquina local, utilizando o *Jupyter Notebook* do *Anaconda Navigator.*

ETAPAS GERAIS DO TRABALHO

A execução desse trabalho envolveu as seguintes etapas:

(i) Escolha de uma área piloto para obtenção dos dados SSS;

(ii) Geração do *dataset* (*input*) com o dado SSS processado (imagem RGB georreferenciada no formato **.tiff*);

(iii) Geração do *target* (*masks*) do mapeamento das duas classes objetivas desse trabalho: alvo refletivo e fundo lamoso (imagem RGB georreferenciada no formato **.tiff*);

(iv) Carregamento, pré-processamento e visualização do *dataset* e *target*;

(v) Divisão do *dataset* e *target* em dados de treinamento, teste e validação para entrada no modelo;

(vi) Simulações de redes *U-Net* e verificação de seus desempenhos;

(vii) Escolha do melhor modelo *U-Net*;

(viii) Conclusões;

(ix) Trabalhos Futuros;

(x) Bibliografia;

(xi) Confidencialidade das Informações.

As etapas (i) a (iii) foram realizadas internamente na companhia utilizando *softwares* específicos de geofísica e GIS, portanto não serão evidenciadas neste documento.

O detalhamento das etapas (iv) a (xi) podem ser observadas a seguir, bem como no código **monografia_final_REV=A.ipynb**.

ETAPAS DETALHADAS DO TRABALHO

Carregamento , pré-processamento e visualização do *dataset* e *target*

Para o carregamento das imagens (*dataset* e *target*) no formato **.tiff* foi utilizada a biblioteca *tifffile*.

O pré-processamento dos dados iniciou-se com a necessidade de diminuir o tamanho das imagens (*dataset* e *target*) originais ora carregadas, devido a limitação computacional para execução de tarefas posteriores. Tomou-se o cuidado para que o *shape* do *dataset* fosse igual ao *shape* do *target.*

Após essa etapa, transformou-se as imagens (*dataset* e *target*) recortadas em *patchs* de 128 x 128 *pixels*, utilizando a biblioteca *scikit-learn.*

Por fim, efetuou-se a alteração da 3º dimensão do *target* de 3 para 1 para posterior entrada no modelo.

Após isso, os dados (*dataset* e *target*) ficaram com essa forma (Figuras 01 e 02):

```
In [12]: # renomeando a variável para um nome menor
         dataset = patches_dataset
         dataset.shape
Out[12]: (5985, 128, 128, 3)
```

(i)

```
In [54]: target.shape
Out[54]: (5985, 128, 128, 1)
```

(ii)

Figura 01: (i) *shape* do *dataset* e (ii) *shape* do *target*

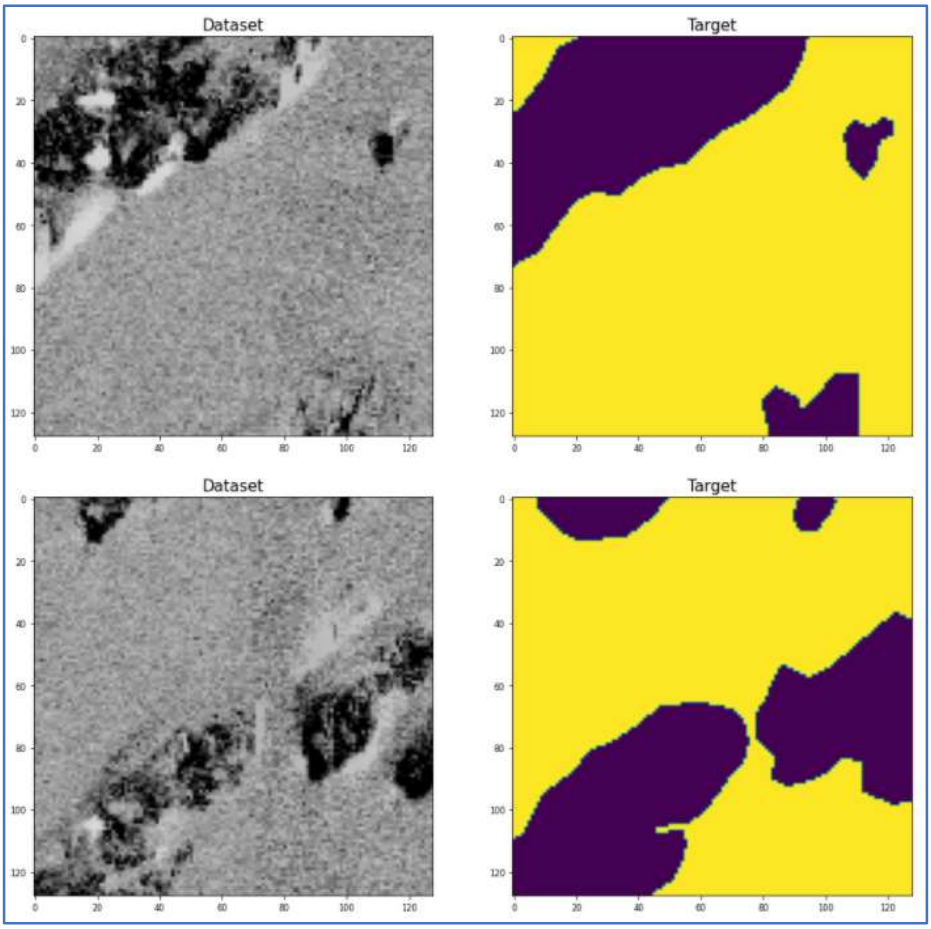

Figura 02: Visualização de 2 *patchs* do *dataset* com seus respectivos *patchs* do *target* (alvo refletivo = roxo e fundo lamoso = amarelo).

Divisão do *dataset* e *target* em dados de treinamento, teste e validação

Para a divisão do *dataset* e *target* em dados de treinamento (80%) e teste (20%) foi utilizada a biblioteca *scikit-learn*.

Para os dados de validação foram extraídas 1.200 amostras dos dados de treinamento.

Feito isso, os dados de treinamento, teste e validação ficaram dessa forma (Figuras 03 e 04):

```
In [95]: # verificando o shape dos dados de treinamento, teste e validação

         print('Shape do x_train : {}'.format(x_train.shape))
         print('Shape do x_test : {}'.format(x_test.shape))
         print('Shape do y_train : {}'.format(y_train.shape))
         print('Shape do y_test : {}'.format(y_test.shape))
         print('Shape do x_val : {}'.format(x_val.shape))
         print('Shape do y_val : {}'.format(y_val.shape))

         Shape do x_train : (3588, 128, 128, 3)
         Shape do x_test : (1197, 128, 128, 3)
         Shape do y_train : (3588, 128, 128, 1)
         Shape do y_test : (1197, 128, 128, 1)
         Shape do x_val : (1200, 128, 128, 3)
         Shape do y_val : (1200, 128, 128, 1)
```

Figura 03: *Shapes* dos dados de treinamento, teste e validação.

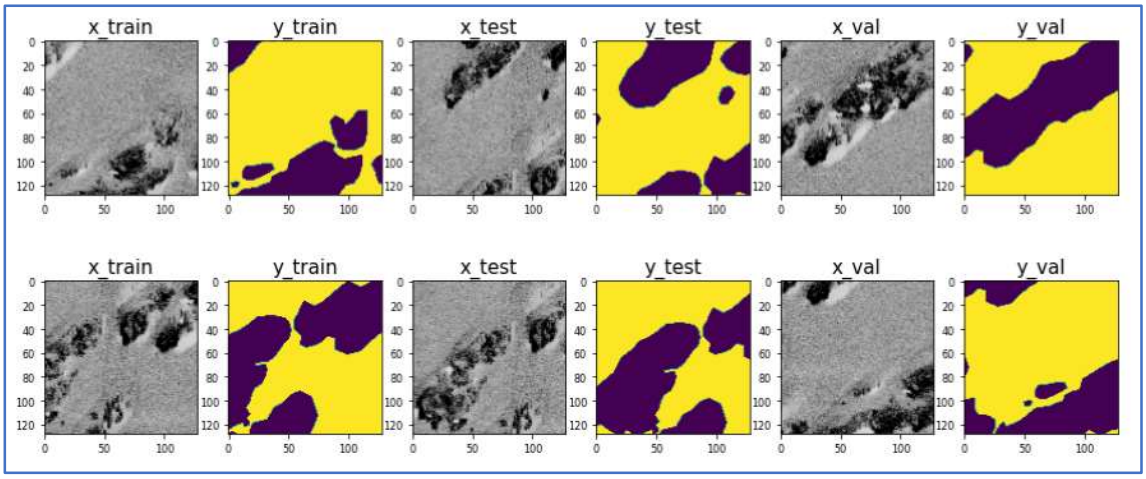

Figura 04: Visualização dos dados de treinamento, teste e validação.

Por fim, aplicou-se a normalização e conversão dos dados de entrada (*x_train, x_test e x_val*) e saída (*y_train, y_test e y_val*) para *float*, sendo que para os dados de saída foram necessárias essas operações para possibilizar utilizar futuramente o *loss* = *binary_crossentropy* no modelo.

Simulações de rede *U-Net* e verificação das performances

A segmentação de imagens é o processo de atribuição de uma classe de objeto para cada *pixel* de uma imagem, como pode ser observado nas Figura 02 e 04.

Como exemplos, a segmentação de imagens tem amplas aplicações em imagens médicas, carros autônomos e imagens de satélite, entre outros.

A tarefa de segmentação de imagens é treinar uma rede neural para produzir uma máscara (denominada nesse trabalho de *target*) da imagem em *pixels*. Isso ajuda a entender a imagem em um nível muito inferior, ou seja, o nível de *pixel*.

Para o desenvolvimento desse trabalho foi escolhida a rede neural totalmente convolucional com arquitetura *U-Net.*

A abordagem de uma *fully convolutional network* (rede totalmente convolucional), a qual a *U-Net* faz parte, foi introduzida por (LONG; SHELHAMER; DARRELL, 2015), em que os autores propuseram uma adaptação de redes popularizadas já existentes, que são utilizadas para classificação de imagens, transformando as camadas totalmente conectadas em camadas de convolução. Isto permite a geração de mapas de características de segmentação para cada imagem.

Em relação à tarefa de segmentação, é necessário que a rede neural seja capaz de combinar a informação da localização com a informação contextual dos mapas de características da imagem a ser predita. A *U-Net* é capaz de juntar as informações contextuais obtidas do caminho de contração (*encoder*) com as informações de localização adquiridas do caminho de expansão (*decoder*) com um bom desempenho (RONNEBERGER; FISCHER; BROX, 2015) (Figura 05).

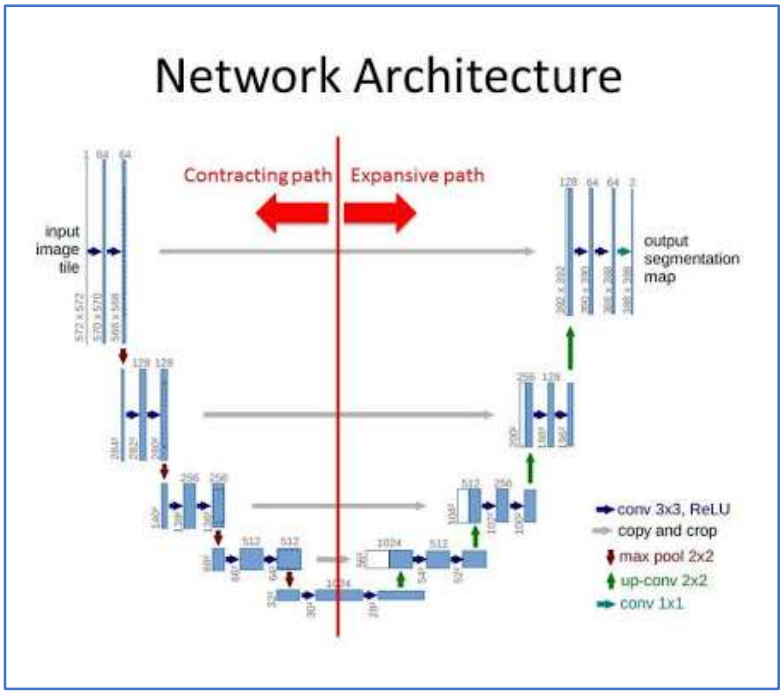

Figura 05: Arquitetura *U-Net*
(fonte: RONNEBERGER; FISCHER; BROX, 2015).

Maiores detalhes da arquitetura *U-Net* podem ser verificados no trabalho de *RONNEBERGER, O.; FISCHER, P.; BROX, T. U-net: Convolutional networks for biomedical image segmentation. In: SPRINGER. International Conference on Medical image computing and computer-assisted intervention. [S.l.], 2015. p. 234–241*.

Nesse trabalho foram utilizadas 3 redes *U-Net* que foram consultadas na comunidade *Kaggle*.

Importante destacar, que foram realizadas dezenas de simulações nas 3 redes, alterando principalmente o otimizador, indicador de perda , número de épocas e função de ativação. As configurações finais podem ser observadas nas Figuras 06, 07 e 08.

In [285]:
```
# Construção do modelo

inputs=tf.keras.layers.Input(shape=(128,128,3))

# Contração

c1=Conv2D(16, kernel_size=(3,3),padding="same", kernel_initializer='he_normal',activation="relu")(inputs)
c1=Conv2D(16, kernel_size=(3,3),padding="same", kernel_initializer='he_normal',activation="relu")(c1)
p1=MaxPooling2D((2,2),strides=2)(c1)

c2=Conv2D(32, kernel_size=(3,3),padding="same", kernel_initializer='he_normal',activation="relu")(p1)
c2=Conv2D(32, kernel_size=(3,3),activation="relu", kernel_initializer='he_normal',padding="same")(c2)
p2=MaxPooling2D((2,2),strides=2)(c2)

c3=Conv2D(64, kernel_size=(3,3),padding="same", kernel_initializer='he_normal',activation="relu")(p2)
c3=Conv2D(64, kernel_size=(3,3),activation="relu", kernel_initializer='he_normal',padding="same")(c3)
p3=MaxPooling2D((2,2),strides=2)(c3)

c4=Conv2D(128, kernel_size=(3,3),padding="same", kernel_initializer='he_normal',activation="relu")(p3)
c4=Conv2D(128, kernel_size=(3,3),activation="relu", kernel_initializer='he_normal',padding="same")(c4)
p4=MaxPooling2D((2,2),strides=2)(c4)

c5=Conv2D(256, kernel_size=(3,3),padding="same", kernel_initializer='he_normal',activation="relu")(p4)
c5=Conv2D(256, kernel_size=(3,3),activation="relu", kernel_initializer='he_normal',padding="same")(c5)

# Expansão

U6=Conv2DTranspose(128, kernel_size=(2,2),activation="relu", kernel_initializer='he_normal',padding="same",strides=(2,2))(c5)
U6=Concatenate()([U6, c4])
c6=Conv2D(128, kernel_size=(3,3),padding="same", kernel_initializer='he_normal',activation="relu")(U6)
c6=Conv2D(128, kernel_size=(3,3),activation="relu", kernel_initializer='he_normal',padding="same")(c6)


U7=Conv2DTranspose(64, kernel_size=(2,2),activation="relu", kernel_initializer='he_normal',padding="same",strides=(2,2))(c6)
U7=Concatenate()([U7, c3])
c7=Conv2D(64, kernel_size=(3,3),padding="same", kernel_initializer='he_normal',activation="relu")(U7)
c7=Conv2D(64, kernel_size=(3,3),activation="relu", kernel_initializer='he_normal',padding="same")(c7)


U8=Conv2DTranspose(32, kernel_size=(2,2),activation="relu", kernel_initializer='he_normal',padding="same",strides=(2,2))(c7)
U8=Concatenate()([U8, c2])
c8=Conv2D(32, kernel_size=(3,3),padding="same", kernel_initializer='he_normal',activation="relu")(U8)
c8=Conv2D(32, kernel_size=(3,3),activation="relu", kernel_initializer='he_normal',padding="same")(c8)


U9=Conv2DTranspose(16, kernel_size=(2,2),activation="relu", kernel_initializer='he_normal',padding="same",strides=(2,2))(c8)
U9=Concatenate()([U9, c1])
c9=Conv2D(16, kernel_size=(3,3),padding="same", kernel_initializer='he_normal',activation="relu")(U9)
c9=Conv2D(16, kernel_size=(3,3),activation="relu", kernel_initializer='he_normal',padding="same")(c9)


output=Conv2D(1, kernel_size=(1,1),activation="sigmoid")(c9)

modelo_1=tf.keras.Model(inputs=[inputs],outputs=[output])
```

In [258]:
```
# compilação do modelo

loss=tf.keras.losses.BinaryCrossentropy()

modelo_1.compile(optimizer="adam",loss=loss,metrics=["accuracy"])

modelo_1.summary()
```

In [55]:
```
# checkpoint

checkpoint=tf.keras.callbacks.ModelCheckpoint("model-SONAR-REV=H.h5", verbose=1, save_best_only=True)

# callbacks

Callbacks=[tf.keras.callbacks.EarlyStopping(patience=2, monitor="val_loss"),checkpoint]


# treinamento do modelo

Results_1=modelo_1.fit(x_train,y_train, validation_data = (x_val,y_val), batch_size=16, epochs=10, callbacks=Callbacks)
```

Figura 06: Rede *U-Net* 1.

In [282]:
```
def conv2d_block(input_tensor, n_filters, kernel_size = 3, batchnorm = True):

    x = Conv2D(filters = n_filters, kernel_size =(kernel_size, kernel_size), kernel_initializer = 'he_normal', padding = 'same')

    if batchnorm:
        x = BatchNormalization()(x)
    x = Activation('relu')(x)

    x = Conv2D(filters = n_filters, kernel_size =(kernel_size, kernel_size), kernel_initializer = 'he_normal', padding = 'same')
    if batchnorm:
        x = BatchNormalization()(x)
    x = Activation('relu')(x)
    return x
```

In [283]:
```
def get_unet(input_img,n_filters = 16, dropout = 0.5, batchnorm = True):
    c1 = conv2d_block(input_img,n_filters=n_filters,kernel_size = 3, batchnorm = batchnorm)
    p1 = MaxPooling2D(2,2)(c1)
    p1 = Dropout(dropout*0.5)(p1)

    c2 = conv2d_block(p1, n_filters=n_filters*2, kernel_size=3, batchnorm=batchnorm)
    p2 = MaxPooling2D((2, 2)) (c2)
    p2 = Dropout(dropout)(p2)

    c3 = conv2d_block(p2, n_filters=n_filters*4, kernel_size=3, batchnorm=batchnorm)
    p3 = MaxPooling2D((2, 2)) (c3)
    p3 = Dropout(dropout)(p3)

    c4 = conv2d_block(p3, n_filters=n_filters*8, kernel_size=3, batchnorm=batchnorm)
    p4 = MaxPooling2D(pool_size=(2, 2)) (c4)
    p4 = Dropout(dropout)(p4)

    c5 = conv2d_block(p4, n_filters=n_filters*16, kernel_size=3, batchnorm=batchnorm)

    u6 = Conv2DTranspose(n_filters*8, (3, 3), strides=(2, 2), padding='same') (c5)
    u6 = concatenate([u6, c4])
    u6 = Dropout(dropout)(u6)
    c6 = conv2d_block(u6, n_filters=n_filters*8, kernel_size=3, batchnorm=batchnorm)

    u7 = Conv2DTranspose(n_filters*4, (3, 3), strides=(2, 2), padding='same') (c6)
    u7 = concatenate([u7, c3])
    u7 = Dropout(dropout)(u7)
    c7 = conv2d_block(u7, n_filters=n_filters*4, kernel_size=3, batchnorm=batchnorm)

    u8 = Conv2DTranspose(n_filters*2, (3, 3), strides=(2, 2), padding='same') (c7)
    u8 = concatenate([u8, c2])
    u8 = Dropout(dropout)(u8)
    c8 = conv2d_block(u8, n_filters=n_filters*2, kernel_size=3, batchnorm=batchnorm)

    u9 = Conv2DTranspose(n_filters*1, (3, 3), strides=(2, 2), padding='same') (c8)
    u9 = concatenate([u9, c1], axis=3)
    u9 = Dropout(dropout)(u9)
    c9 = conv2d_block(u9, n_filters=n_filters*1, kernel_size=3, batchnorm=batchnorm)

    outputs = Conv2D(1, (1, 1), activation='sigmoid') (c9)
    modelo_2 = Model(inputs=[input_img], outputs=[outputs])
    return modelo_2
```

In [64]:
```
input_img = Input((128, 128, 3))
modelo_2 = get_unet(input_img, n_filters=16, dropout=0.5, batchnorm=True)

modelo_2.compile(optimizer='adam', loss="binary_crossentropy", metrics=["accuracy"])
modelo_2.summary()
```

In [*]:
```
earlystopper = EarlyStopping(patience=2, monitor="val_loss", verbose=1)
checkpointer = ModelCheckpoint("model-SONAR-REV=I.h5", save_best_only=True)
Results_2 = modelo_2.fit(x_train, y_train,validation_data = (x_val,y_val), epochs=10,
                         callbacks=[earlystopper, checkpointer],verbose=1)
```

Figura 07: Rede *U-Net* 2.

```
In [293]: inputs = Input((128, 128, 3))


          c1 = Conv2D(16, (3, 3), activation='elu', kernel_initializer='he_normal', padding='same') (inputs)
          c1 = Dropout(0.1) (c1)
          c1 = Conv2D(16, (3, 3), activation='elu', kernel_initializer='he_normal', padding='same') (c1)
          p1 = MaxPooling2D((2, 2)) (c1)

          c2 = Conv2D(32, (3, 3), activation='elu', kernel_initializer='he_normal', padding='same') (p1)
          c2 = Dropout(0.1) (c2)
          c2 = Conv2D(32, (3, 3), activation='elu', kernel_initializer='he_normal', padding='same') (c2)
          p2 = MaxPooling2D((2, 2)) (c2)

          c3 = Conv2D(64, (3, 3), activation='elu', kernel_initializer='he_normal', padding='same') (p2)
          c3 = Dropout(0.2) (c3)
          c3 = Conv2D(64, (3, 3), activation='elu', kernel_initializer='he_normal', padding='same') (c3)
          p3 = MaxPooling2D((2, 2)) (c3)

          c4 = Conv2D(128, (3, 3), activation='elu', kernel_initializer='he_normal', padding='same') (p3)
          c4 = Dropout(0.2) (c4)
          c4 = Conv2D(128, (3, 3), activation='elu', kernel_initializer='he_normal', padding='same') (c4)
          p4 = MaxPooling2D(pool_size=(2, 2)) (c4)

          c5 = Conv2D(256, (3, 3), activation='elu', kernel_initializer='he_normal', padding='same') (p4)
          c5 = Dropout(0.3) (c5)
          c5 = Conv2D(256, (3, 3), activation='elu', kernel_initializer='he_normal', padding='same') (c5)

          u6 = Conv2DTranspose(128, (2, 2), strides=(2, 2), padding='same') (c5)
          u6 = concatenate([u6, c4])
          c6 = Conv2D(128, (3, 3), activation='elu', kernel_initializer='he_normal', padding='same') (u6)
          c6 = Dropout(0.2) (c6)
          c6 = Conv2D(128, (3, 3), activation='elu', kernel_initializer='he_normal', padding='same') (c6)

          u7 = Conv2DTranspose(64, (2, 2), strides=(2, 2), padding='same') (c6)
          u7 = concatenate([u7, c3])
          c7 = Conv2D(64, (3, 3), activation='elu', kernel_initializer='he_normal', padding='same') (u7)
          c7 = Dropout(0.2) (c7)
          c7 = Conv2D(64, (3, 3), activation='elu', kernel_initializer='he_normal', padding='same') (c7)

          u8 = Conv2DTranspose(32, (2, 2), strides=(2, 2), padding='same') (c7)
          u8 = concatenate([u8, c2])
          c8 = Conv2D(32, (3, 3), activation='elu', kernel_initializer='he_normal', padding='same') (u8)
          c8 = Dropout(0.1) (c8)
          c8 = Conv2D(32, (3, 3), activation='elu', kernel_initializer='he_normal', padding='same') (c8)

          u9 = Conv2DTranspose(16, (2, 2), strides=(2, 2), padding='same') (c8)
          u9 = concatenate([u9, c1], axis=3)
          c9 = Conv2D(16, (3, 3), activation='elu', kernel_initializer='he_normal', padding='same') (u9)
          c9 = Dropout(0.1) (c9)
          c9 = Conv2D(16, (3, 3), activation='elu', kernel_initializer='he_normal', padding='same') (c9)

          outputs = Conv2D(1, (1, 1), activation='sigmoid') (c9)

          modelo_3 = Model(inputs=[inputs], outputs=[outputs])
          modelo_3.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
          modelo_3.summary()

In [*]: # Fit model
        earlystopper = EarlyStopping(patience=2, verbose=1)
        checkpointer = ModelCheckpoint("model-SONAR-REV=J.h5", verbose=1, save_best_only=True)


        Results_3 = modelo_3.fit(x_train, y_train, validation_data = (x_val,y_val), epochs=20, batch_size=16,
                                 callbacks=[earlystopper, checkpointer],verbose=1)
```

Figura 08: Rede *U-Net* 3.

As principais diferenças entre as redes, podem ser verificadas abaixo:

Rede *U-Net* 1: camadas de convolução e *maxpolling*. Utilizado *batch size* = 16. Número de épocas de treinamento = 10.

Rede *U-Net* 2: camadas de convolução, *maxpolling, dropout e batchnormalizantion.* Não foi utilizado *batch size*. Número de épocas de treinamento = 10.

Rede *U-Net* 3: camadas de convolução, *maxpolling* e *dropout.* Utilizado *batch size* = 16. Número de épocas de treinamento = 20.

Para análise da melhor rede *U-Net*, foram realizados o treinamento das 3 redes e seus desempenhos e curvas de aprendizagem podem ser observados respectivamente na Tabela 01 e Figura 09.

Tabela 01: Desempenhos das 3 redes *U-Net* na etapa de treinamento.

Modelo	Acurácia Treino	Acurácia Validação	Loss Treino	Loss Validação
U-Net 1	0,9981	0,9979	0,0048	0,0052
U-Net 2	0,9866	0,9866	0,0335	0,0324
U-Net 3	0,9971	0,9975	0,0070	0,0058

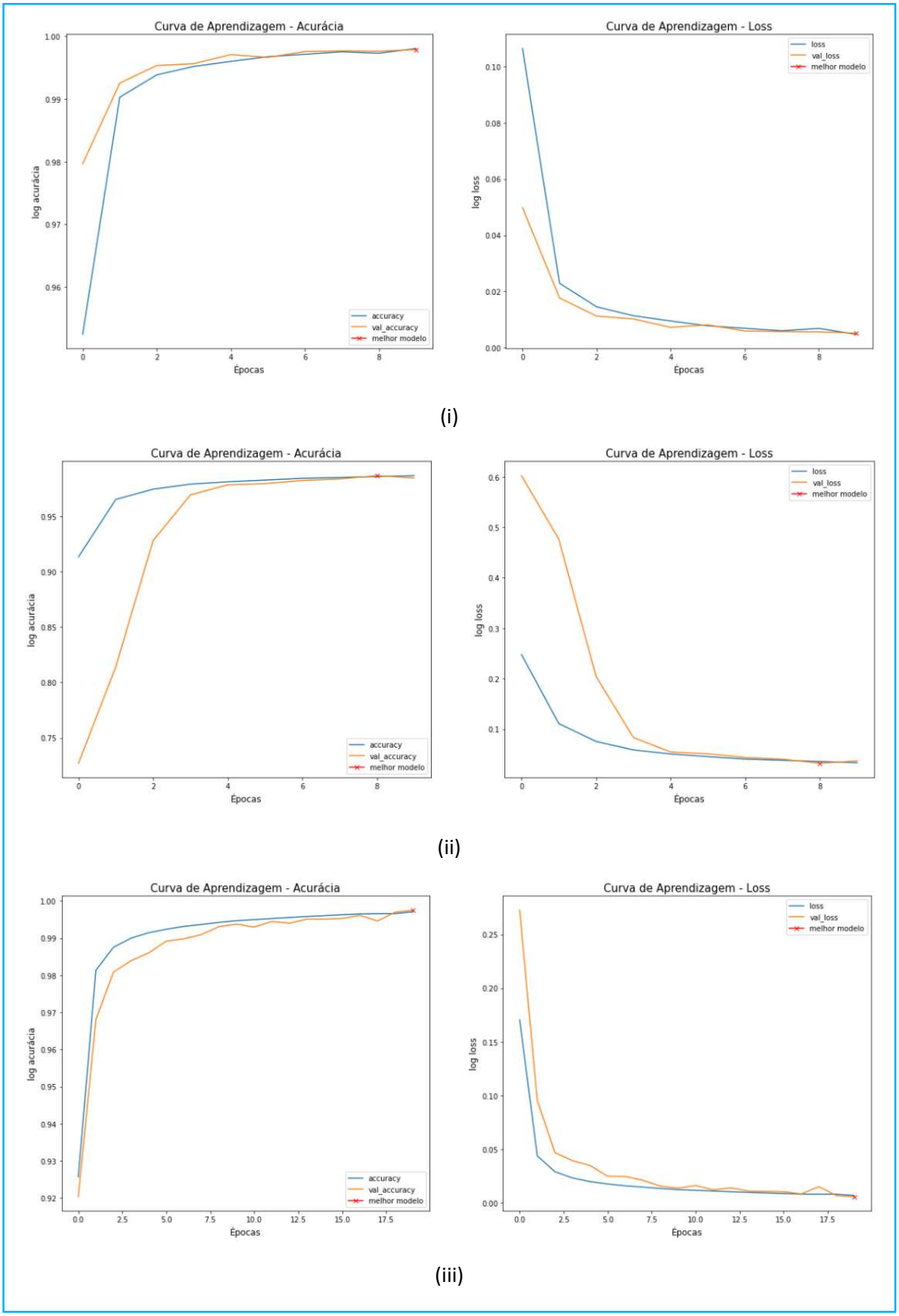

Figura 09: Curvas de aprendizagem da *acurácia* e *loss* para os modelos *U-Net* 1 (i), *U-Net* 2 (ii) e *U-Net 3* (iii).

Após o treinamento, as redes foram apresentadas aos dados novos (*x_test* e *y_test*) para verificar suas capacidades de generalização.

A tabela 02 apresenta os desempenhos das 3 redes *U-Net* aplicados no conjunto de dados testes.

Tabela 02: Desempenhos das 3 redes U-Net na etapa de teste.

Modelo	Acurácia Teste	Loss Teste
U-Net 1	0,9979	0,0051
U-Net 2	0,9867	0,0321
U-Net 3	0,9975	0,0058

Escolha do melhor modelo *U-Net*

Como pode ser observado nas Tabelas 01 e 02, o melhor modelo *U-Net* para os dados apresentados é o *U-Net* 1.

A acurácia do modelo *U-Net* 1 ficou na faixa de 99,8% para os dados de treinamento e teste.

Sendo assim, o modelo *U-Net* 1 conseguiu a melhor previsão e segmentação das imagens originais, atingindo o objeto do trabalho de segmentar as imagens originais nas duas classes – alvo refletivo = roxo e fundo lamoso = amarelo – como pode ser observado na Figura 10.

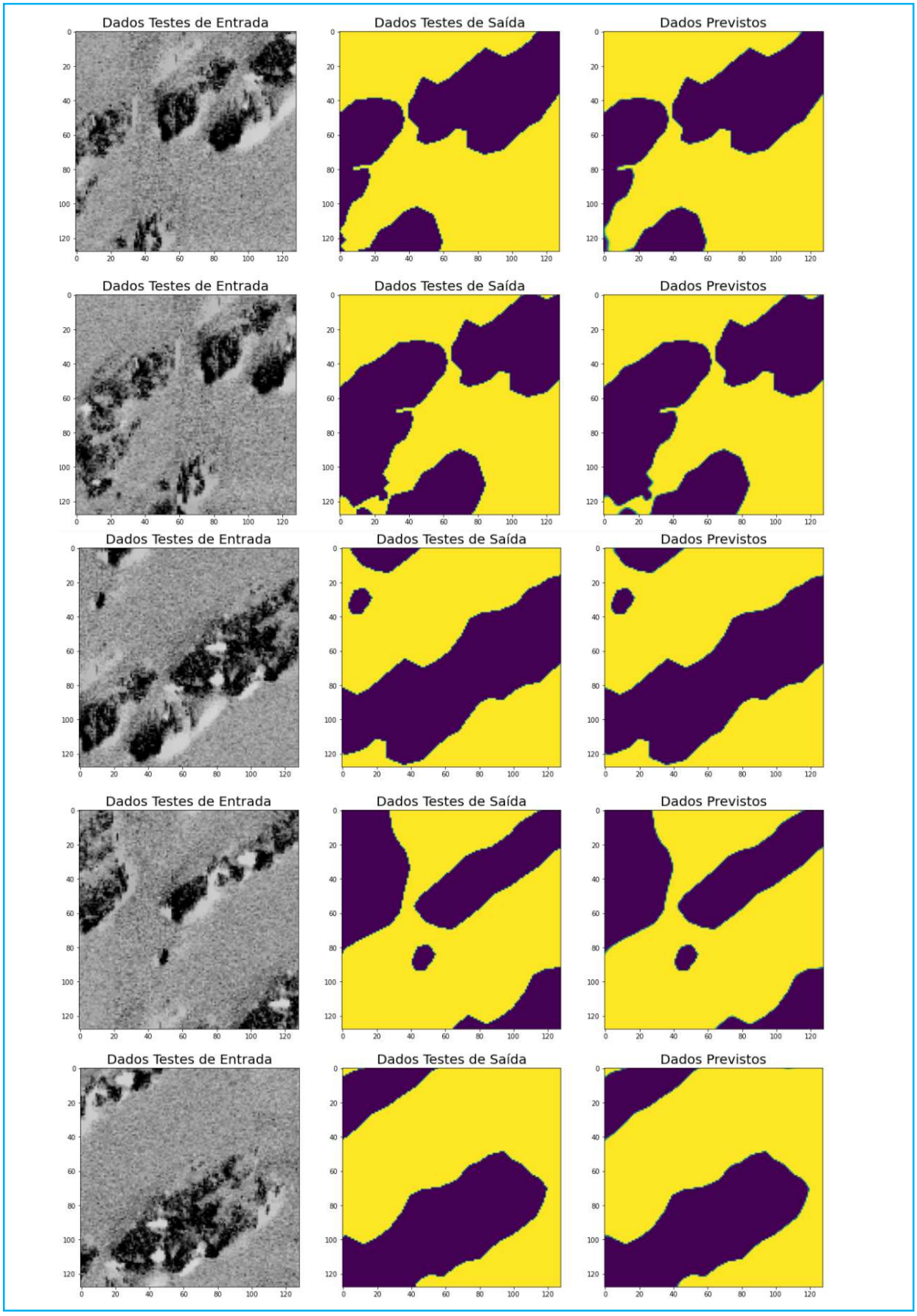

Figura 10: A esquerda são os dados originais, no centro o *target* dos dados originais e a direita as imagens preditas.

Conclusões

O objetivo principal desse trabalho foi aplicar e avaliar técnicas de *Deep Learning* que permitissem a segmentação de imagens acústicas submarinas, oriundas dos dados geofísicos de alta resolução (SSS), em duas classes distintas: alvo refletivo e fundo lamoso, para possibilitar o aperfeiçoamento e automatização do mapeamento de feições naturais no leito marinho, que atualmente é realizada de maneira manual.

Para tal objetivo, foram utilizadas 3 redes *U-Net* onde foram realizadas dezenas de simulações, alterando determinados hiperparâmetros, até que se obtivessem as melhores performances de cada rede.

Toadas as 3 redes *U-Net* obtiveram resultados muito satisfatórios, com acurácia acima de 98%, entretanto a rede *U-Net* 1 foi a que obteve o melhor resultado com acurácia na faixa de 99,8%.

A realização deste trabalho permitiu comprovar a importância das redes *U-Net* na tarefa de segmentação de imagens, atingindo resultados muito satisfatórios.

Trabalhos Futuros

Em trabalhos futuros, as principais oportunidades que podem ser exploradas são:

- colocar o código em produção (interface usuário);

- aumentar o número de classes;

- segmentar a classe alvo refletivo em mais classes;

- utilizar técnicas não supervisionadas.

Bibliografia

LONG, J.; SHELHAMER, E.; DARRELL, T. Fully convolutional networks for semantic segmentation. In: Proceedings of the IEEE conference on computer vision and pattern recognition. [S.l.: s.n.], 2015. p. 3431–3440. Citado 2 vezes nas páginas 17 e 26.

RONNEBERGER, O.; FISCHER, P.; BROX, T. U-net: Convolutional networks for biomedical image segmentation. In: SPRINGER. International Conference on Medical image computing and computer-assisted intervention. [S.l.], 2015. p. 234–241. Citado 2 vezes nas páginas 26 e 27.