INSTITUTO FEDERAL
Minas Gerais
Campus Bambuí

YANKY JHONATHA MONTEIRO FONTE BOA

RELATÓRIO 5 – TÓPICOS ESPECIAIS EM ENGENHARIA DE COMPUTAÇÃO

BAMBUÍ-MG

2023

A arquitetura do modelo de rede neural é do seguinte modo:

A entrada do modelo é uma camada de Input que recebe dados com a forma (None, num_features), onde num_features é o número de atributos no conjunto de dados.
Em seguida, há uma camada oculta que eu usei com 128 neurônios, ativada pela função de ativação "sigmoid". A inicialização dos pesos dos neurônios é realizada usando a distribuição "lecun_normal". Após a camada Dense, há uma camada de Batch Normalization para normalizar as ativações.
Em seguida, é aplicada uma camada de Dropout com uma taxa de dropout de 0.1 para regularização.
A arquitetura continua com mais duas camadas ocultas, cada uma com 32 neurônios, ativadas pela função de ativação "sigmoid". Também é aplicada Batch Normalization e Dropout em cada camada.
A camada de saída consiste em uma camada Dense com o mesmo número de neurônios que o número de classes no conjunto de dados. A função de ativação usada também é a "sigmoid".
O modelo é criado utilizando a classe Model do Keras, passando as camadas de entrada e saída como argumentos. A arquitetura do modelo é resumida e plotada usando as funções summary() e plot_model().
Imagens do código:

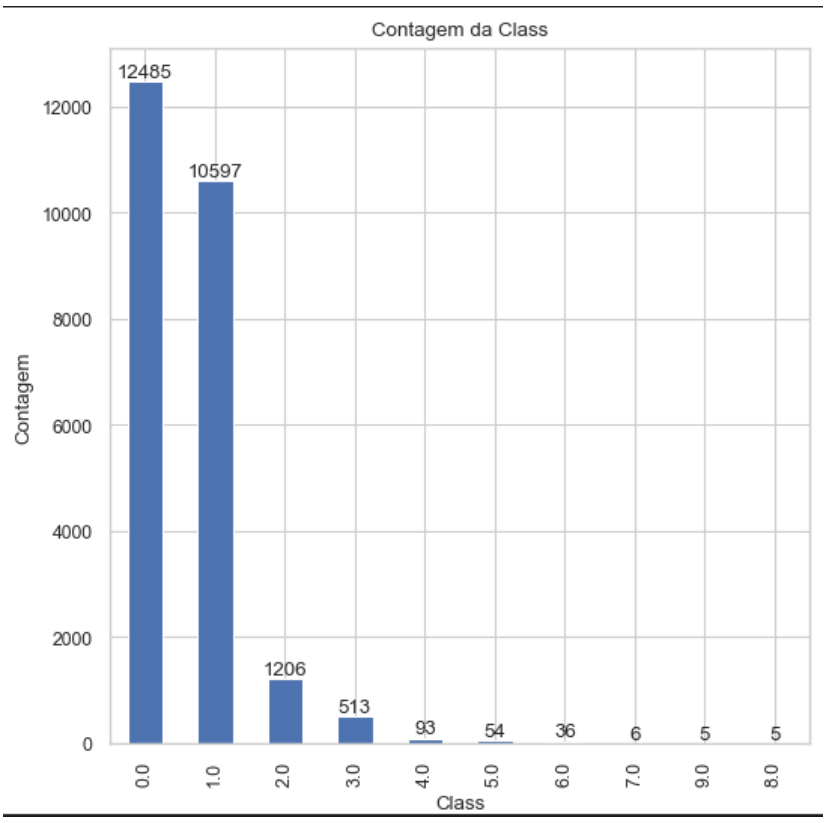
```python
import tensorflow as tf
from tensorflow.keras import layers

act_function = 'sigmoid'
drop = 0.1
initializer = tf.keras.initializers.lecun_normal()

inputs = tf.keras.Input(shape=(x_train.shape[1],), dtype='float32')

layer1 = layers.Dense(128, activation=act_function, kernel_initializer=initializer)(inputs)
layer1 = layers.BatchNormalization()(layer1)
layer1 = layers.Dropout(drop)(layer1)

layer2 = layers.Dense(32, activation=act_function, kernel_initializer=initializer)(layer1)
layer2 = layers.BatchNormalization()(layer2)
layer2 = layers.Dropout(drop)(layer2)

layer3 = layers.Dense(32, activation=act_function, kernel_initializer=initializer)(layer2)
layer3 = layers.BatchNormalization()(layer3)
layer3 = layers.Dropout(drop)(layer3)

layer4 = layers.Dense(32, activation=act_function, kernel_initializer=initializer)(layer3)
layer4 = layers.BatchNormalization()(layer4)
layer4 = layers.Dropout(drop)(layer4)

outputs = layers.Dense(y_train.shape[1], activation=act_function, kernel_initializer=initializer)(layer4)

dnn_pokerhand = tf.keras.Model(inputs, outputs)
```

Como o dataset era muito grande eu selecionei apenas 25 mil dados e fiz o treinamento da RNA, também utilizei o one hot encoding para converter as colunas para o binário que foi preciso, resultando em 25mil rows x 86 colunas.

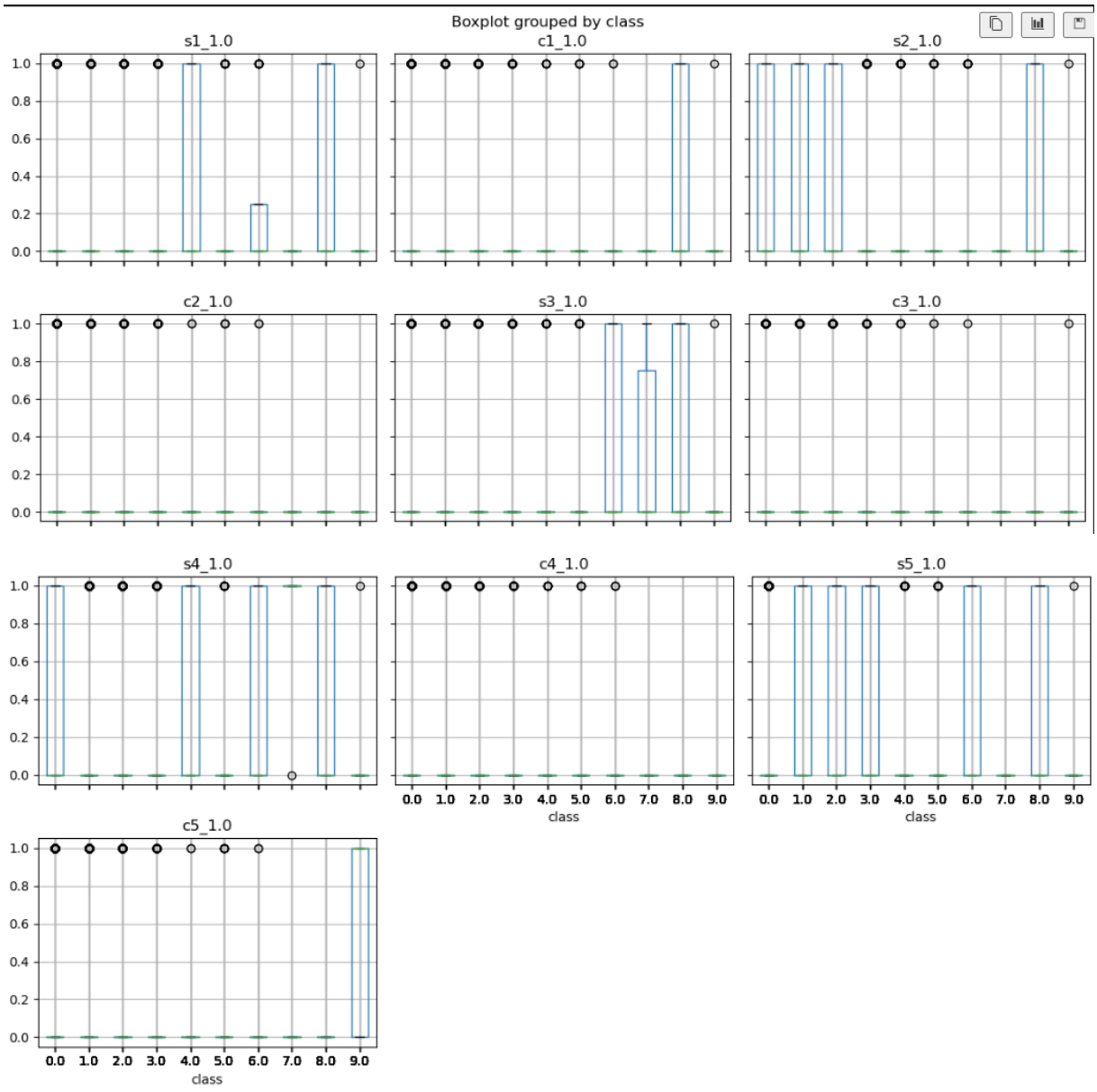
	class	s1_1.0	s1_2.0	s1_3.0	s1_4.0	c1_1.0	c1_2.0	c1_3.0	c1_4.0	c1_5.0	...	c5_4.0	c5_5.0	c5_6.0	c5_7.0	c5_8.0	c5_9.0	c5_10.0	c5_11
8170	0.0	0	0	1	0	0	0	0	0	0	...	0	1	0	0	0	0	0	
8279	0.0	0	1	0	0	0	0	0	1	0	...	0	0	0	0	0	0	1	
19068	0.0	0	0	0	1	0	0	0	1	0	...	0	0	0	0	0	0	0	
13267	0.0	0	1	0	0	0	0	0	0	0	...	1	0	0	0	0	0	0	
10883	1.0	0	1	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	
16023	0.0	0	1	0	0	0	0	0	0	0	...	0	0	0	0	0	0	1	
11363	1.0	0	0	0	1	1	0	0	0	0	...	1	0	0	0	0	0	0	
14423	1.0	1	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	
21962	1.0	0	0	1	0	0	0	0	0	0	...	0	0	0	0	0	0	1	
4426	0.0	0	0	0	1	0	0	0	0	0	...	0	0	0	0	1	0	0	

25000 rows × 86 columns

Também fiz a contagem dos diagnósticos da variável “class” no conjunto de dados e plotei um gráfico de barras mostrando a contagem de cada classe:



Também criei um boxplot para cada atributo na lista x_names, agrupados pela variável 'class'. Totalizando uma figura com 12 boxplots.



Utilizei a função train_test_split do scikit-learn para realizar a divisão dos dados em conjuntos de treinamento, validação e teste.

```
x_train_val, x_test, y_train_val, y_test = train_test_split(x_frames, y_frames, test_size=0.3, shuffle=True)
x_train, x_val, y_train, y_val = train_test_split(x_train_val, y_train_val, test_size=0.3, shuffle=True)

print(x_train.shape)
print(y_train.shape)
print(x_val.shape)
print(y_val.shape)
print(x_test.shape)
print(y_test.shape)
```

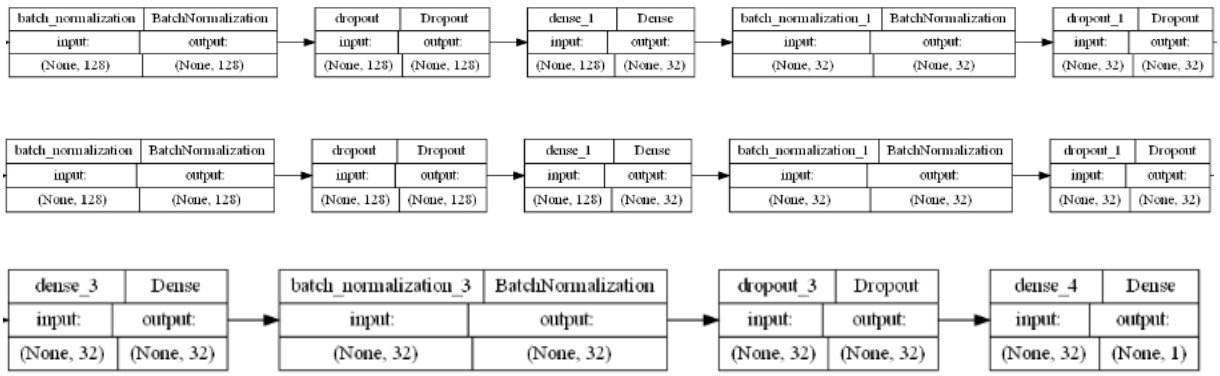
✓ 0.0s

```
(12250, 10)
(12250, 1)
(5250, 10)
(5250, 1)
(7500, 10)
(7500, 1)
```

Meu modelo de treinamento ‘model’ ficou definido como:

```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 10)]              0

 dense (Dense)               (None, 128)               1408

 batch_normalization (BatchN  (None, 128)              512
 ormalization)

 dropout (Dropout)           (None, 128)               0

 dense_1 (Dense)             (None, 32)                4128

 batch_normalization_1 (Batc  (None, 32)               128
 hNormalization)

 dropout_1 (Dropout)         (None, 32)                0

 dense_2 (Dense)             (None, 32)                1056

 batch_normalization_2 (Batc  (None, 32)               128
 hNormalization)

...
Total params: 8,577
Trainable params: 8,129
Non-trainable params: 448
_________________________________________________________________
```

E o boxplot do modelo ficou assim:

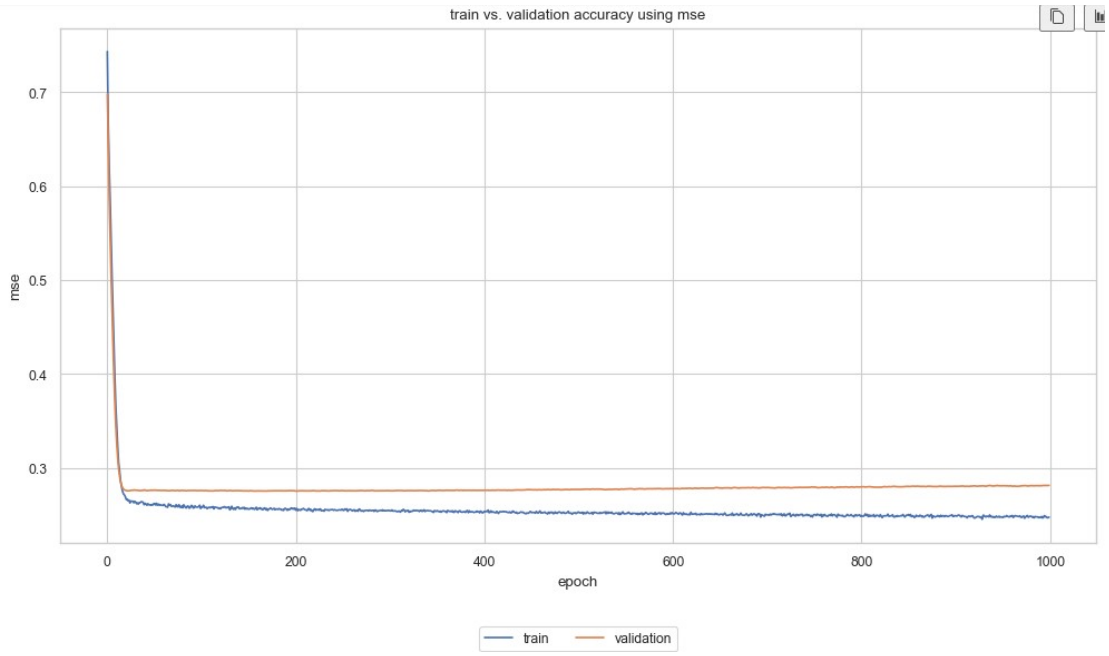


Criação do modelo:

```python
if new_model ==False:
    tf.keras.backend.set_epsilon(1)
    opt = Adam(learning_rate=0.0001)

    dnn_pokerhand.compile(optimizer=opt, loss='binary_crossentropy', metrics=['accuracy'])

    cp = ModelCheckpoint(
        filepath='models/model_weights.h5',
        save_weights_only=True,
        monitor='loss',
        mode='min',
        save_best_only=True
    )

    es = EarlyStopping(monitor='loss', mode='min', patience=100)

    history = dnn_pokerhand.fit(
        x_train,
        y_train,
        validation_data=(x_val, y_val),
        epochs=1000,
        verbose=1,
        callbacks=[es, cp],
        batch_size=64,
        shuffle=False
    )

    np.save('models/history_model.npy', history.history)
    dnn_pokerhand.save('models/dnn_pokerhand.h5')
```

O modelo é compilado com um otimizador Adam, uma função de perda de entropia cruzada binária e métricas de acurácia. Durante o treinamento, são utilizados callbacks para salvar os melhores pesos do modelo com base na diminuição da perda e para interromper o treinamento antecipadamente caso a perda não melhore após um determinado número de épocas. Após o treinamento, o modelo é salvo em um arquivo h5 junto ao histórico de treinamento em um arquivo numpy. Caso um modelo pré-treinado esteja disponível, ele pode ser carregado a partir do arquivo h5 para uso posterior.

Depois da criação do modelo, podemos verificar a acurácia e plotar o gráfico para verificar a curva de convergência e olhar seu comportamento. No meu caso, ao observar o gráfico é notório que está sendo satisfeito o modelo, pois obtivemos uma taxa de acerto de 91.97% até 93.24%. Abaixo está o gráfico:



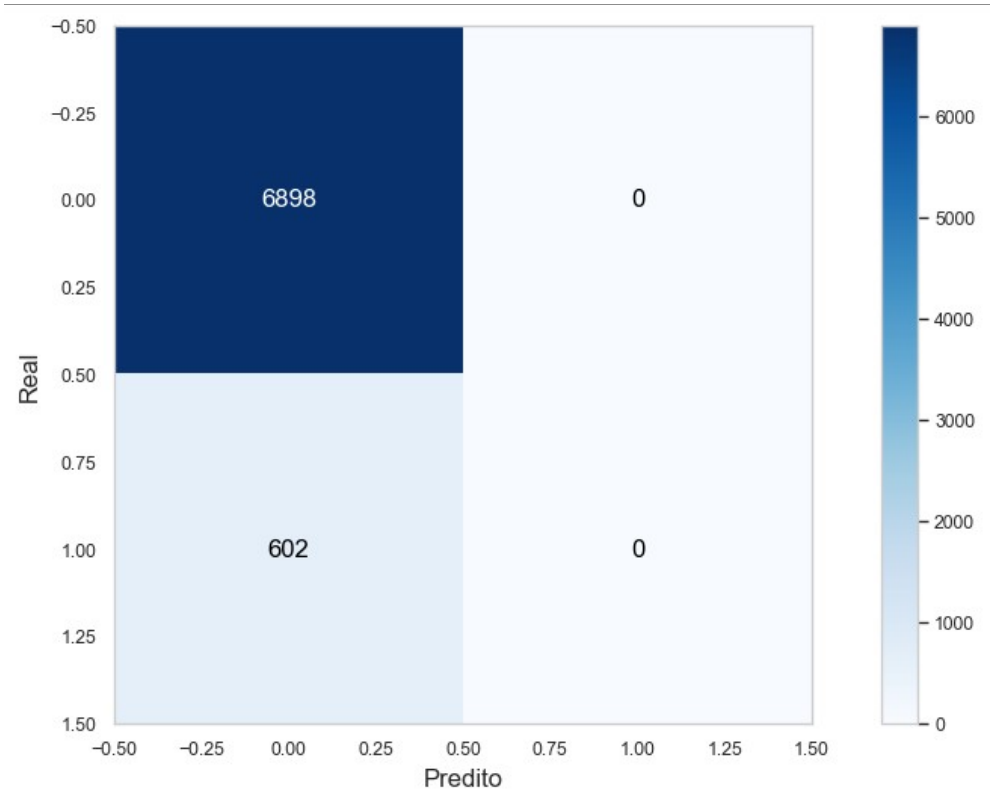
Print da taxa de acerto:

```
from sklearn.metrics import confusion_matrix

# Calculando a matriz de confusão
cm = confusion_matrix(y_test.tolist(), y_hat.tolist())

# Calculando a taxa de acerto
accuracy = (cm[0, 0] + cm[1, 1]) / cm.sum()
accuracy_percent = round(accuracy * 100, 2)

# Exibindo a taxa de acerto
print("Taxa de Acerto: {:.2f}%".format(accuracy_percent))
```

✓ 0.0s

```
Taxa de Acerto: 93.24%
```

E ao plotar a matriz de confusão podemos ver os valores, e quais estão sendo informados corretamente, a matriz adiciona os valores de cada célula como texto, destacando aqueles acima de um certo limiar com uma cor diferente:



Observação: O meu código foi rodado em CPU, mas há parte do código que identifica a GPU, não foi possível rodar nela, pois meu notebook é AMD e sua placa de vídeo também, e não há suporte para ela, então otimizei o máximo para conseguir rodar melhor na CPU. Caso haja necessidade, o código consegue ser convertido para rodar em GPU, aumentando ainda mais a velocidade de processamento e garantindo melhores resultados.