Aula 14

Treinamento customizado com TensorFlow

Eduardo Lobo Lustosa Cabral

1. Objetivos

Apresentar como realizar treinamento customizado com o TensorFlow

Apresentar um exemplo de treinamento customizado de uma RNA com o TensorFlow.

Importação das bilbiotecas necessárias

```
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
print(tf.__version__)
```

```
2.17.1
```

2. Introdução

Na aula anterior vimos como calcular gradientes com o TensorFlow e resolvemos um problema de ajuste de função.

Nessa aula vamos treinar uma RNA calculando o gradiente da função de custo em relação aos parâmetros da RNA e aplicar esse gradiente para atualizar os parâmetros usando um dos métodos de otimização que já conhecemos.

Para esse exemplo de treinamento customizado vamos desenvolver uma RNA para classificar as imagens de dígitos do conjunto MNIST.

3. Carregar e processar dados

A célula abaixo carrega o conjunto de dados de dígitos MNIST da coleção do Keras.

```
(x_train, y_train), (x_test, y_test) =
tf.keras.datasets.mnist.load_data()

print('Dimensão dos dados de entrada de treinamento =', x_train.shape)
print('Dimensão dos dados de entrada de teste =', x_test.shape)
print('Dimensão dos dados de saída de treinamento =', y_train.shape)
print('Dimensão dos dados de saída de teste =', y_test.shape)
```

```
Dimensão dos dados de entrada de treinamento = (60000, 28, 28)
Dimensão dos dados de entrada de teste = (10000, 28, 28)
Dimensão dos dados de saída de treinamento = (60000,)
Dimensão dos dados de saída de teste = (10000,)
```

Como as imagens do conjunto de dígitos MNIST são pequenas e em tons de cinza, para simplificar vamos transformar as imagens em vetores e usar uma rede com camadas densas.

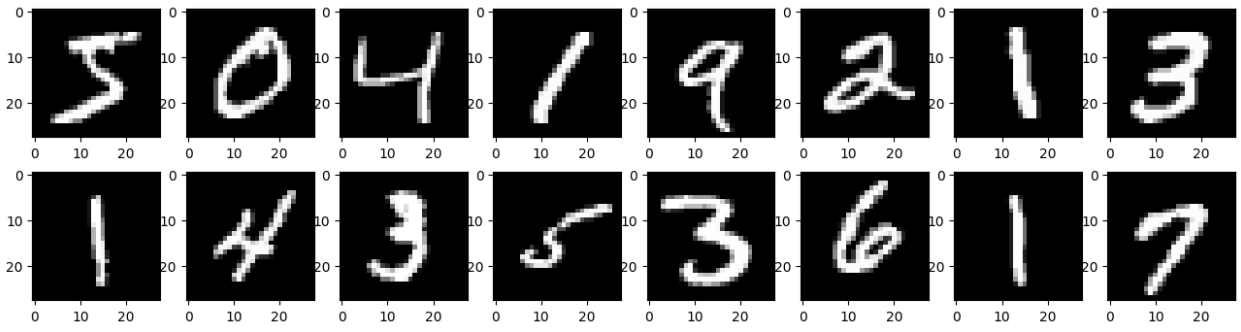
Dessa forma é necessário redimensionar as imagens para torná-las vetores.

```
# Número total de pixels nas imagens
nx = x_train.shape[1]*x_train.shape[2]

# Redimensionamento das imagens
x_train_flat = np.reshape(x_train, (x_train.shape[0], nx))/255.
x_test_flat = np.reshape(x_test, (x_test.shape[0], nx))/255.

print('Dimensão dos dados de entrada de treinamento =',
x_train_flat.shape)
print('Dimensão dos dados de entrada de teste =', x_test_flat.shape)
```

```
Dimensão dos dados de entrada de treinamento = (60000, 784)
Dimensão dos dados de entrada de teste = (10000, 784)
```

Gráficos de alguns exemplos

```
fig, axs = plt.subplots(2,8, figsize=(16,4))
index = 0
for i in range(2):
    for j in range(8):
        axs[i,j].imshow(x_train[index], cmap='gray')
        index += 1
plt.show()
```



Transformação das saídas em vetores one-hot

```
y_train_hot = tf.one_hot(y_train, 10)
y_test_hot = tf.one_hot(y_test, 10)

print('Dimensão dos dados de saída de treinamento =',
```

```
y_train_hot.shape)
print('Dimensão dos dados de saída de teste =',y_test_hot.shape)

Dimensão dos dados de saída de treinamento = (60000, 10)
Dimensão dos dados de saída de teste = (10000, 10)
```

4. Configuração da RNA

Para resolver essa tarefa de classificação multiclasse vamos utlizar uma RNA com camadas densas com a seguinte configuração:

- Primeira camada: 128 neurônios e função de ativação Relu;
- Segunda camada: 64 neurônios e função de ativação Relu;
- Camada de saída: 1 neurônio e função de ativação softmax.

```
# Importa classes
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

# Configura RNA
rna = Sequential()
rna.add(Dense(128, activation='relu', input_shape=(784,)))
rna.add(Dense(64, activation='relu'))
rna.add(Dense(10, activation='softmax'))

# Resumo da RNA
rna.summary()

/usr/local/lib/python3.10/dist-packages/keras/src/layers/core/
dense.py:87: UserWarning: Do not pass an `input_shape`/`input_dim`
argument to a layer. When using Sequential models, prefer using an
`Input(shape)` object as the first layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer,
**kwargs)

Model: "sequential_1"

┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳
━━━━━━━━━━━━━━━━━┓
┃ Layer (type)                         ┃ Output Shape                ┃
Param # ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇
━━━━━━━━━━━━━━━━━┩
│ dense_3 (Dense)                      │ (None, 128)                 │
100,480 │
├──────────────────────────────────────┼─────────────────────────────┼
─────────────────┤
│ dense_4 (Dense)                      │ (None, 64)                  │
8,256 │
├──────────────────────────────────────┼─────────────────────────────┼
```

```
──────────────────┤
| dense_5 (Dense)                      | (None, 10)                  |
650 |
└──────────────────────────────────────┴─────────────────────────────┴
──────────────────┘

 Total params: 109,386 (427.29 KB)

 Trainable params: 109,386 (427.29 KB)

 Non-trainable params: 0 (0.00 B)
```

5. Compilação da RNA

A compilação da RNA segue a seguinte configuração:

- Otimizador: Adam com taxa de aprendizado de 0.001;
- Função de custo: Categorical Crossentropy
- Métrica: Accuracy

Para realizar um treinamento customizado devemos criar objetos com o otimizador, com a função de custo e com a métrica.

Para criar esses objetos deve-se usar as versões na forma de classes de todas essas funções, conforme realizado na célula abaixo.

```
# Define objeto otimizador usando tf.keras.optimizer.Addam
optimizer = tf.keras.optimizers.Adam(learning_rate=0.001)

# Define objeto função de custo usando
tf.keras.losses.BinaryCrossentropy
loss_object = tf.keras.losses.CategoricalCrossentropy()

# Define objeto métrica usando tf.keras.metrics.BinaryAccuracy
metric_object = tf.keras.metrics.CategoricalAccuracy()
```

6. Teste da RNA, da função de custo e da métrica

Vamos avaliar a RNA não treinada, juntamente com a função de custo e a métrica, para verificar se estão corretas e também para termos uma base do resultado esperado do treinamento.

```
# Calcula previsões da RNA não treinada
outputs = rna.predict(x_test_flat)

# Calcula função de custo
loss_value = loss_object(y_true=y_test_hot, y_pred=outputs)
print("Custo antes do treinamento =", loss_value.numpy())

# Calcula métrica
```

```
accuracy = metric_object(y_true=y_test_hot, y_pred=outputs)
print("Métrica antes do treinamento =", accuracy.numpy())
```

```
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step
Custo antes do treinamento = 2.3464541
Métrica antes do treinamento = 0.1078
```

```
# Identifica as classes previstas
y_pred = tf.math.argmax(outputs, axis=1)

print('Classes previstas do primeiros 5 exemplos:',
y_pred[:5].numpy())
```

```
Classes previstas do primeiros 5 exemplos: [1 1 8 5 5]
```

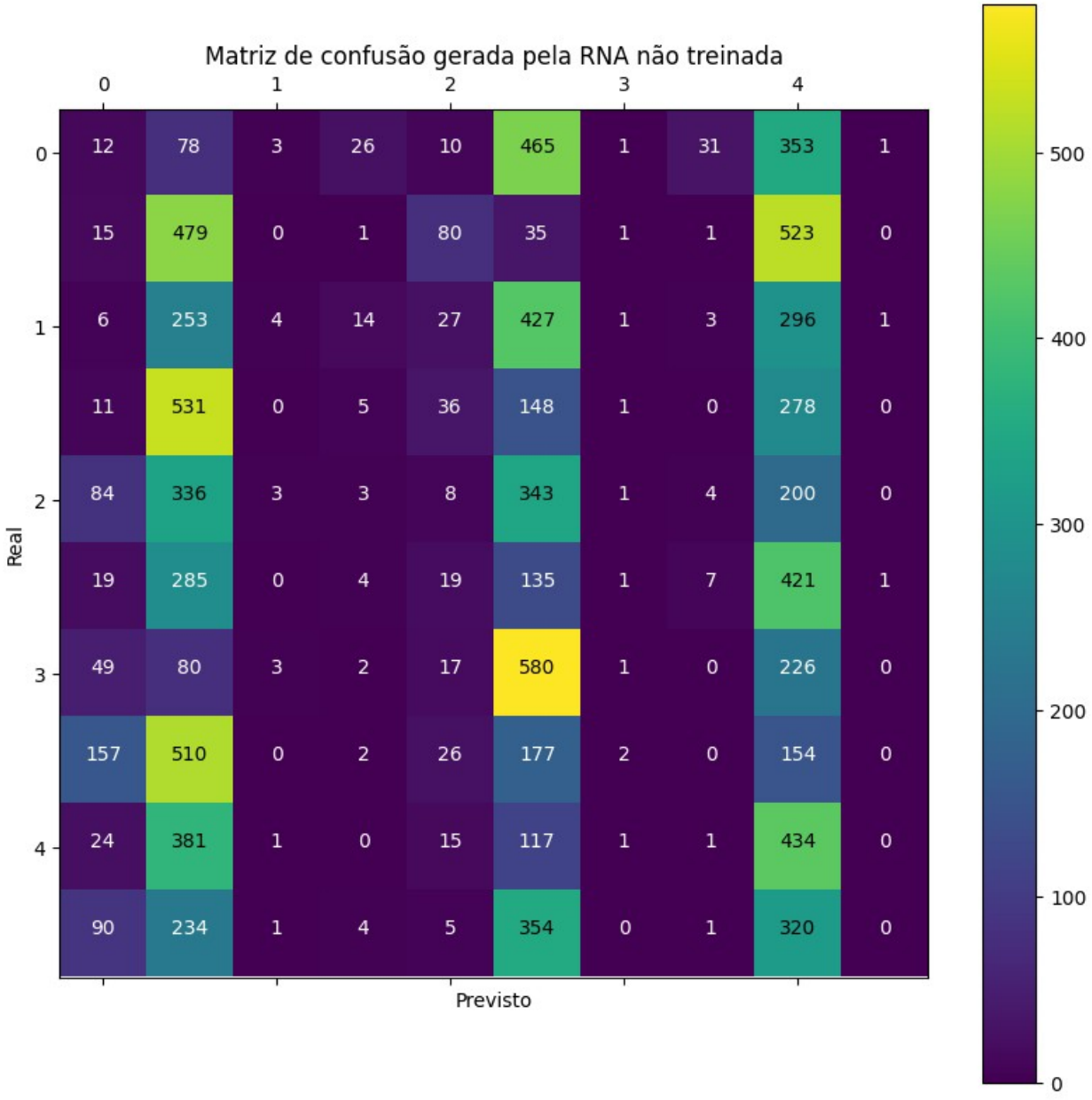
6.1 Matriz de confusão

Para permitir uma visualização melhor do resutado do treinamento vamos usar a matriz de confusão dos resutados previstos pela RNA.

Na célula abaixo é criada uma função para calcular e mostrar a matriz de confusão.

```
# Importa funções para calcula matriz e confusão
from sklearn.metrics import confusion_matrix
import itertools

# Define função para construir matriz de confusão
def plot_confusion_matrix(y_true, y_pred, title='',
labels=[0,1,2,3,4,5,6,7,8,9]):
    cm = confusion_matrix(y_true, y_pred)
    fig = plt.figure(figsize=(10,10))
    ax = fig.add_subplot(111)
    cax = ax.matshow(cm)
    plt.title(title)
    fig.colorbar(cax)
    ax.set_xticklabels([''] + labels)
    ax.set_yticklabels([''] + labels)
    plt.xlabel('Previsto')
    plt.ylabel('Real')
    fmt = 'd'
    thresh = cm.max() / 2.
    for i, j in itertools.product(range(cm.shape[0]),
range(cm.shape[1])):
          plt.text(j, i, format(cm[i, j], fmt),
                  horizontalalignment="center",
                  color="black" if cm[i, j] > thresh else "white")
    plt.show()

plot_confusion_matrix(y_test, y_pred, title='Matriz de confusão gerada
pela RNA não treinada')
```

```
<ipython-input-21-710a0ca9c75d>:13: UserWarning: set_ticklabels()
should only be used with a fixed number of ticks, i.e. after
set_ticks() or using a FixedLocator.
  ax.set_xticklabels([''] + labels)
<ipython-input-21-710a0ca9c75d>:14: UserWarning: set_ticklabels()
should only be used with a fixed number of ticks, i.e. after
set_ticks() or using a FixedLocator.
  ax.set_yticklabels([''] + labels)
```



- Observa-se que os resultados antes do treinamento da RNA são muotruins.

7. Treinamento da RNA

Para treinar essa RNA vamos criar um loop customizado usando a função `tf.GradientTape()`.

7.1 Cálculo do gradiente e atualização dos parâmetros

Para calcular o gradiente da função de custo em relação aos parâmetros da RNA e depois atualizar esses parâmetros usando o otimizador configurado anteriormente, vamos criar uma função.

Essa função processa e atualiza os parâmetros da rede para um lote de dados.

Para acessar os parâmetros de um modelo do TensorFlow basta usar `model.trainable_weights`.

```
# Função que calcula gradientes e atualiza parâmetros da RNA
@tf.function()
def train_step(optimizer, loss_object, model, x, y):
    '''
    Função para calcular o gradiente e atualizar os parâmetros da RNA

    Argumentos:
        optimizer: otimizador configurado para atualizar os parâmetros
        loss_object: função de custo configurada anteriormente
        model: RNA que está sendo treinada
        x: tensor com os dados de entrada de treinamento
        y: saídas desejadas dos dados de treinamento

    Retorna:
        y_pred = saídas previstas pela RNA
        loss_value = valor da função de custo
    '''

    with tf.GradientTape() as tape:
        y_pred = model(x)
        loss_value = loss_object(y_true=y, y_pred=y_pred)

    gradients = tape.gradient(loss_value, model.trainable_weights)
    optimizer.apply_gradients(zip(gradients, model.trainable_weights))

    return y_pred, loss_value
```

7.2 Cálculo do custo e da métrica para os dados de validação

No final de cada época de treinamento, temos que validar a RNA no conjunto de dados de validação. Para isso vamos criar uma função que calcula a função de custo e a métrica para os dados de validação.

```
# Função para calcular custo e métrica dos dados de validação
@tf.function()
def perform_validation(model, x_val, y_val):
    # Calcula saídas previstas dos dados de validação
    val_prev = model(x_val)

    #Calcula custo dos dados de validação
    val_loss = loss_object(y_true=y_val, y_pred=val_prev)

    # Calcula métrica para dados de validação
    val_accuracy = metric_object(y_val, val_prev)

    return val_loss, val_accuracy
```

7.3 Lote de dados

Para dividirmos os dados em lotes de treinamento temos duas opções:

1. Dividir os dados durante o treinamento usando indexação dos tensores;
2. Colocar os dados em um `dataset` do TensorFlow.

Nesse exemplo, vamos utilizar um dataset.

```
# Define tamanho do lote
batch_size = 1024

# Cria dataset
train_dataset = tf.data.Dataset.from_tensor_slices((x_train_flat,
y_train_hot)).cache().prefetch(tf.data.experimental.AUTOTUNE).batch(ba
tch_size)

for x, y in train_dataset.take(1):
  print('Dimensão do lote de entrada:', x.shape)
  print('Dimensão do lote de saída:', y.shape)
```

```
Dimensão do lote de entrada: (1024, 784)
Dimensão do lote de saída: (1024, 10)
```

7.4 Loop de treinamento customizado

Usando a função `apply_gradient()` vamos criar um loop de treinamemto customizado. Vamos utilizar 30 épocas de treinamento.

```
# Loop de treinamento customizado

# Define número de épocas
num_epocas = 30

# # Número de lotes por época
# num_lotes = len(y_train) // batch_size
```

```
# Incializa vetores de custo e métricas
train_loss = np.zeros(num_epocas)
train_accuracy = np.zeros(num_epocas)
val_loss = np. zeros(num_epocas)
val_accuracy = np.zeros(num_epocas)

# Loop de treinamento
for i in range(num_epocas):

    # Inicializa custo e métrica no início de cada época
    accuracy_batch = []
    loss_batch = []

    # Percorre os lotes de dados
    for x, y in train_dataset:
        # Calcula gradientes e atualiza parâmetros da RNA
        y_pred, loss_value = train_step(optimizer, loss_object, rna,
x, y)

        # Calcula métrica para dados de treinamento
        metric = metric_object(y, y_pred)

        # Acumula custo e métrica de cada lote para ter valor médio no
final dde cada época
        loss_batch.append(loss_value)
        accuracy_batch.append(metric)

    # Calcula custo e métrica médios dos dados de treinamento
    loss = np.mean(np.array(loss_batch))
    accuracy = np.mean(np.array(accuracy_batch))

    # Guarda valores do custo e métrica dos dados de treinamento para
fazer gráfico posteriomente
    train_loss[i] = loss
    train_accuracy[i] = accuracy

    # Calcula função de custo e métrica para todos os dados de
validação
    val_loss[i], val_accuracy[i] = perform_validation(rna,
x_test_flat, y_test_hot)

    # Imprime resultado da função de custo e métrica da época
    print('Época:', i, '-', 'custo =', loss, '-', 'exatidão =',
accuracy, '-', 'custo_val =', val_loss[i], '-', 'val_exatidão =',
val_accuracy[i])

# Imprime resultado final
print('\nCusto final =', loss)
print('Exatidão final=', accuracy)
```

```
print('\nCusto final de validação =', val_loss[num_epocas-1])
print('Exatidão final de validação =', val_accuracy[num_epocas-1])
```

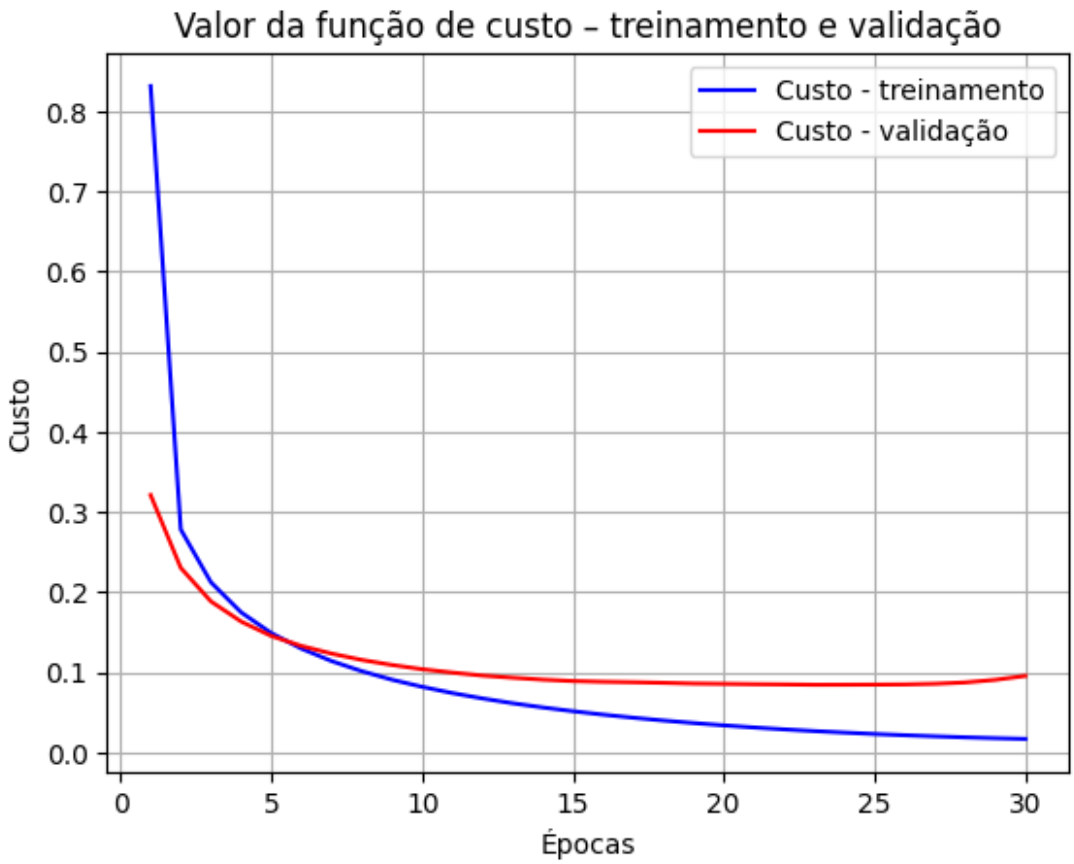
```
Época: 0 - custo = 0.831743 - exatidão = 0.48474398 - custo_val =
0.32132309675216675 - val_exatidão = 0.7109624743461609
Época: 1 - custo = 0.2783336 - exatidão = 0.7638945 - custo_val =
0.23044857382774353 - val_exatidão = 0.809386670589447
Época: 2 - custo = 0.2124474 - exatidão = 0.8299624 - custo_val =
0.18857291340827942 - val_exatidão = 0.8509045243263245
Época: 3 - custo = 0.17494877 - exatidão = 0.86236733 - custo_val =
0.16346493363380432 - val_exatidão = 0.8748344779014587
Época: 4 - custo = 0.14905182 - exatidão = 0.8823222 - custo_val =
0.14548766613006592 - val_exatidão = 0.8907861113548279
Época: 5 - custo = 0.12963827 - exatidão = 0.8962436 - custo_val =
0.13295021653175354 - val_exatidão = 0.9024860262870789
Época: 6 - custo = 0.11411964 - exatidão = 0.90667194 - custo_val =
0.12343964725732803 - val_exatidão = 0.9114760160446167
Época: 7 - custo = 0.10151867 - exatidão = 0.914834 - custo_val =
0.115549236536026 - val_exatidão = 0.9186649322509766
Época: 8 - custo = 0.09086447 - exatidão = 0.9214262 - custo_val =
0.10926245152950287 - val_exatidão = 0.924582839012146
Época: 9 - custo = 0.08200323 - exatidão = 0.9269199 - custo_val =
0.1039867177605629 - val_exatidão = 0.9296042323112488
Época: 10 - custo = 0.0742151 - exatidão = 0.93160105 - custo_val =
0.0997549518942833 - val_exatidão = 0.9338935613632202
Época: 11 - custo = 0.06748493 - exatidão = 0.93565595 - custo_val =
0.09603475034236908 - val_exatidão = 0.9376646876335144
Época: 12 - custo = 0.061489638 - exatidão = 0.93924695 - custo_val =
0.09335608780384064 - val_exatidão = 0.9410337209701538
Época: 13 - custo = 0.056245435 - exatidão = 0.9424349 - custo_val =
0.09093999117612839 - val_exatidão = 0.9440070986747742
Época: 14 - custo = 0.05154146 - exatidão = 0.94526297 - custo_val =
0.08919916301965714 - val_exatidão = 0.9466688632965088
Época: 15 - custo = 0.04727951 - exatidão = 0.9478067 - custo_val =
0.08829066157341003 - val_exatidão = 0.9490717053413391
Época: 16 - custo = 0.043420397 - exatidão = 0.9501082 - custo_val =
0.08771156519651413 - val_exatidão = 0.9512516856193542
Época: 17 - custo = 0.039938573 - exatidão = 0.9521981 - custo_val =
0.08698762953281403 - val_exatidão = 0.9532386064529419
Época: 18 - custo = 0.03678214 - exatidão = 0.954112 - custo_val =
0.08609169721603394 - val_exatidão = 0.9550626873970032
Época: 19 - custo = 0.033896603 - exatidão = 0.9558719 - custo_val =
0.08568870276212692 - val_exatidão = 0.9567510485649109
Época: 20 - custo = 0.031258196 - exatidão = 0.95750105 - custo_val =
0.08525696396827698 - val_exatidão = 0.9583128094673157
Época: 21 - custo = 0.02890583 - exatidão = 0.95901483 - custo_val =
0.08497001975774765 - val_exatidão = 0.9597657918930054
Época: 22 - custo = 0.026770188 - exatidão = 0.9604217 - custo_val =
0.08455566316843033 - val_exatidão = 0.9611148238182068
Época: 23 - custo = 0.024882527 - exatidão = 0.96172804 - custo_val =
```

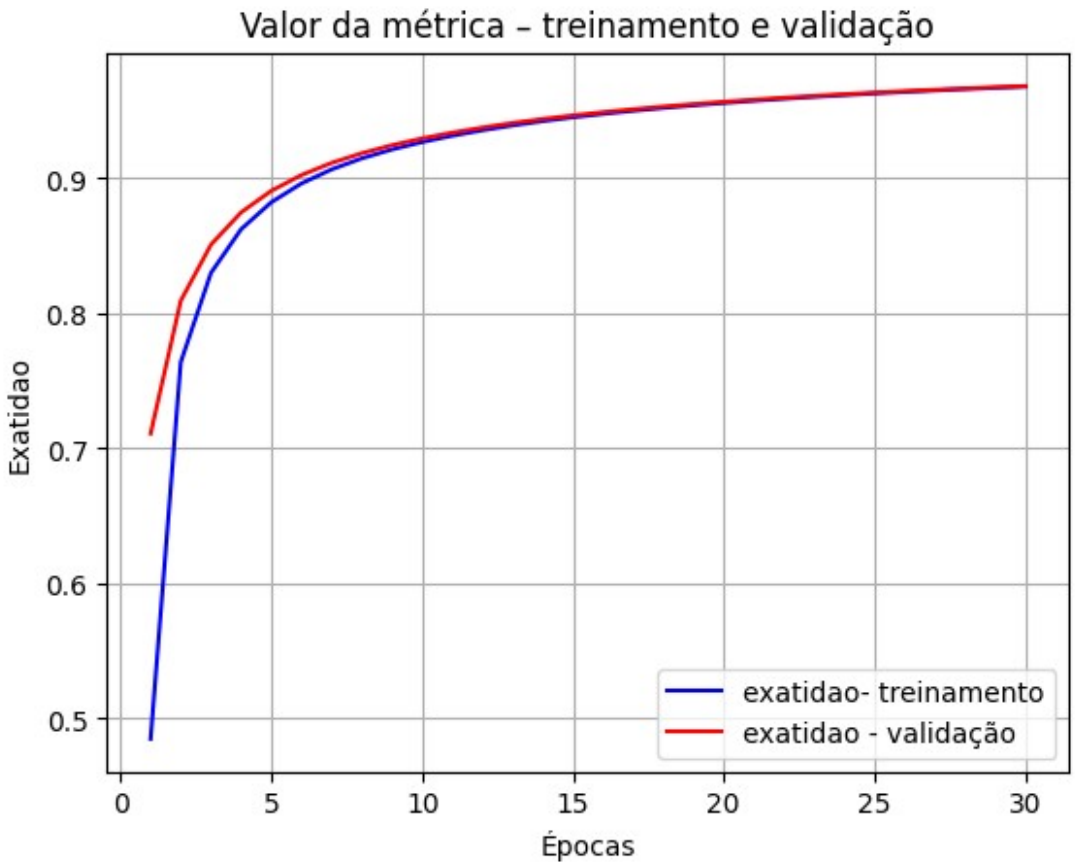
```
0.08461333066225052 - val_exatidão = 0.9623763561248779
Época: 24 - custo = 0.023138206 - exatidão = 0.96295035 - custo_val =
0.08473814278841019 - val_exatidão = 0.9635539650917053
Época: 25 - custo = 0.021589668 - exatidão = 0.9640924 - custo_val =
0.08496887981891632 - val_exatidão = 0.9646524786949158
Época: 26 - custo = 0.02021842 - exatidão = 0.9651626 - custo_val =
0.08577559888362885 - val_exatidão = 0.9656863212585449
Época: 27 - custo = 0.018964915 - exatidão = 0.9661697 - custo_val =
0.08743192255496979 - val_exatidão = 0.9666543006896973
Época: 28 - custo = 0.017895147 - exatidão = 0.96711105 - custo_val =
0.09069347381591797 - val_exatidão = 0.9675657153129578
Época: 29 - custo = 0.016950991 - exatidão = 0.9679965 - custo_val =
0.09563121944665909 - val_exatidão = 0.9684151411056519

Custo final = 0.016950991
Exatidão final= 0.9679965

Custo final de validação = 0.09563121944665909
Exatidão final de validação = 0.9684151411056519
```

- Observa-se que usar a estrutura `dataset` do TensorFlow é mais rápido do que usar um gerador de dados customizado.

Gráfico do processo de treinamento.

```
# Cria vetor de épocas
epocas = range(1, num_epocas + 1)

# Gráfico dos valores de custo
plt.plot(epocas, train_loss, 'b', label='Custo - treinamento')
plt.plot(epocas, val_loss, 'r', label='Custo - validação')
plt.title('Valor da função de custo – treinamento e validação')
plt.xlabel('Épocas')
plt.ylabel('Custo')
plt.legend()
plt.grid()
plt.show()

# Gráfico dos valores da métrica
plt.plot(epocas, train_accuracy, 'b', label='exatidao- treinamento')
plt.plot(epocas, val_accuracy, 'r', label='exatidao - validação')
plt.title('Valor da métrica – treinamento e validação')
plt.xlabel('Épocas')
plt.ylabel('Exatidao')
plt.legend()
plt.grid()
plt.show()
```


Valor da função de custo – treinamento e validação
Custo - treinamento
Custo - validação
Custo
Épocas
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0
5
10
15
20
25
30



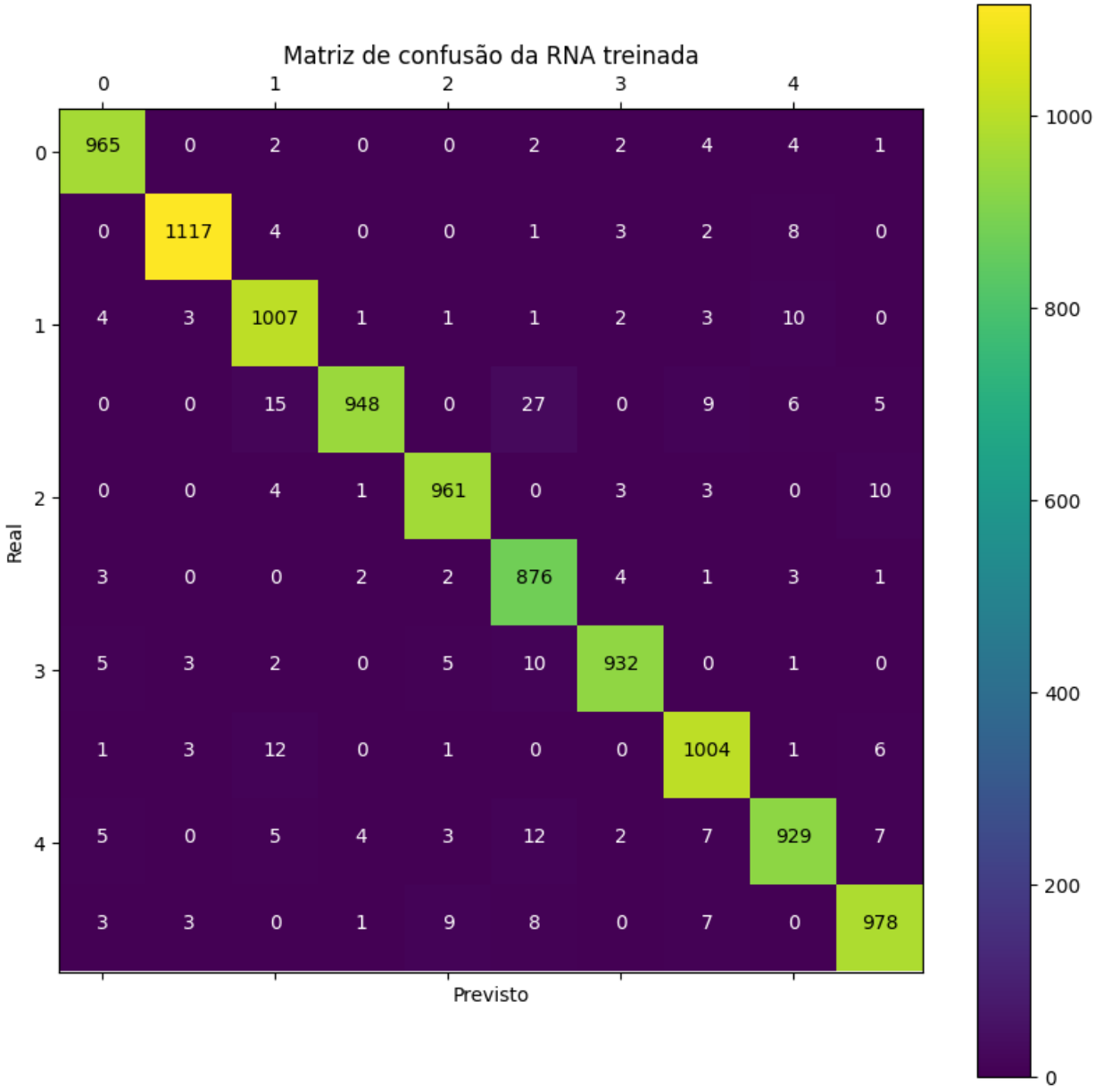
7.4 Avaliação e teste da RNA

Calculo da matriz de confusão para a RNA treinada.

```
# Calcula saída prevista para os dados de teste
outputs = rna.predict(x_test_flat)

# Identifica as classes previstas
y_pred = tf.math.argmax(outputs, axis=1)

print('Classes previstas do primeiros 5 exemplos:',
y_pred[:5].numpy())

# calcula e mostra matriz de confusão
plot_confusion_matrix(y_test, y_pred, title='Matriz de confusão da RNA
treinada')
```

```
313/313 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step
Classes previstas do primeiros 5 exemplos: [7 2 1 0 4]

<ipython-input-21-710a0ca9c75d>:13: UserWarning: set_ticklabels()
should only be used with a fixed number of ticks, i.e. after
set_ticks() or using a FixedLocator.
```

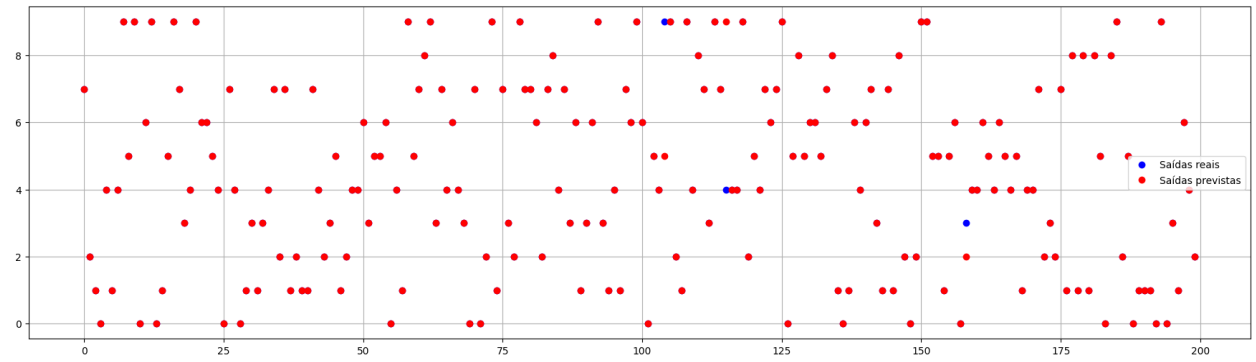
```
  ax.set_xticklabels([''] + labels)
<ipython-input-21-710a0ca9c75d>:14: UserWarning: set_ticklabels()
should only be used with a fixed number of ticks, i.e. after
set_ticks() or using a FixedLocator.
  ax.set_yticklabels([''] + labels)
```



Comparação das saídas previstas e reais

```
# verificação dos 200 primeiros exemplos de teste
plt.figure(figsize=(22,6))
plt.plot(y_test[:200], 'bo', label='Saídas reais')
plt.plot(y_pred[:200], 'ro', label='Saídas previstas')
```

```
plt.legend()
plt.grid()
plt.show()
```




```
val_loss_mean, val_accuracy_mean = perform_validation(rna,
x_test_flat, y_test_hot)
print(val_loss_mean.numpy(), val_accuracy_mean.numpy())

0.09563122 0.96843064
```