Aula 15

Funções de custo e métricas em camadas customizadas

Eduardo Lobo Lustosa Cabral

1. Objetivos

Apresentar como usar funções de custo em camadas customizadas.

Apresentar como inserir incluir funções de custo diretamente em modelos.

Apresentar exemplo completo de uso de camadas customizadas com custo, incluindo o treinamento do modelo.

Importação das bilbiotecas básicas

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

```
2.17.1
```

2. Introdução

Nas aulas anteriores vimos algumas formas de definir funções de custo e métricas customizadas.

Existem outras formas para lidar com funções de custo que não se enquadram nos formatos que vimos nas aulas enteriores.

A grande maioria das funções de custo é calculada a partir das saídas reais e previstas (`y_true` e `y_pred`), mas existem casos onde isso não se aplica. Por exemplo, o cálculo de uma função de custo pode exigir apenas a ativação de uma das camadas da rede, nesse caso não existem valores desejados e essa ativação pode não ser uma saída do modelo.

Nesses casos usam-se camadas customizadas na forma de classe para calcular a função de custo.

3. Função de custo na forma de camada customizada

3.1 Regularização de ativações

Já vimos metódos de regularização de RNAs baseados no controle da amplitude dos pesos das ligações e viéses (regularização L1 e L2). Existe outra forma de regularização que é aplicar regularização nas ativações das camadas da RNA.

Na regularização das ativações tenta-se diminuir o valor das ativações de forma que seja necessária a contribuição de um maior número de neurônios para a obtenção das saídas da RNA. Isso faz com que grupos de neurônios não se expecializem em determinados exemplos e, assim, tende a aumentar a capacidade de generalização da rede.

Uma função de custo que inclui regularização de ativações na l-ésima camada de uma RNA é dada pela seguinte expressão:

$$J_{RA}(W,B) = J(W,B) + \lambda \sum_{k=1}^{n^{[l]}} a_k^{[l]2}$$

onde $J(W,B)$ é a função de custo normal sem regularização, $J_{RA}(W,B)$ é a função de custo com regularização das ativaçõe da l-ésima camada, W e b representam genericamente todos os parâmetros da rede, λ é o fator de regularização, $n^{[l]}$ é o número de neurônios da l-ésima camada e $a^{[l]}$ são as ativações da l-ésima camada.

Nota-se que é possível incluir regularização de ativações em qualquer camada de uma rede. Asim, se for incluida regularização de ativações em outras camadas deve-se incluir os termos correspondentes na equação acima.

Uma forma de incluir o termo de regularização das ativações na função de custo é definir uma camada customizada para calcular a soma das aitvações da camada e adicionar esse termo à função de custo normal.

Para fazer isso, usa-se o método `add_loss(operação)` dentro de uma camada customizada. Observa-se que para usar esse método para incluir o termo de regularização de ativações de uma camada, temos que incluir a operação matemática que define a soma das ativações como argumento do método.

Quando se usa o método `add_loss()`, o valor calculado por esse custo adicional é adicionado à função de custo principal $J(W,B)$ durante o treinamento. Observe que a função de custo principal consiste na função de custo passada na compilação do modelo.

Vamos mostrar como implementar a regularização de ativações em um problema de classificação multiclasse usando o conjunto de dados Fashion MNIST.

Codificação da camada com função de custo de regularização de ativações

Para criar o termo adicional da regularização de ativações na função de custo vamos criar uma camada customizada que somente calcula esse custo.

Observa-se que pode usar uma camada customizada para calcular outras coisas além da função de custo adicional.

```
# Importa classe de camadas do Keras
from tensorflow.keras import layers

# Camada customizada para regularização de ativações
class ActivityRegularizationLayer(layers.Layer):
    # Inicializa classe e parâmetros
    def __init__(self, lamb=0):
        # Inicializa classe
        super(ActivityRegularizationLayer, self).__init__()
        # Inicializa fator de regularização
        self.lamb = lamb

    # Define operações da camada
    def call(self, inputs):
        # Função de custo é a soma das entradas
        self.add_loss(self.lamb * (tf.reduce_sum(inputs**2)))
        return inputs
```

- Observa-se que essa camada retorna a sua entrada sem nenhuma modificação.
- O único parâmetro necessário para a operação da camada é o termo de regularização λ, que é inicializado no método `__init__()`.
- Observa-se que a função de custo adicional é inserida nessa camada com o método `add_loss()`.
- Como a camada não possui nenhum parâmetro, não é necessário incluir o método `build()`.

3.2 Dados de treinamento

Vamos carregar o conjunto de dados Fashion MNIST diretamento de Keras, conforme realizado na célula a seguir.

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

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-
keras-datasets/train-labels-idx1-ubyte.gz
29515/29515 ━━━━━━━━━━━━━━━━━━━━ 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-
keras-datasets/train-images-idx3-ubyte.gz
```

```
26421880/26421880 ━━━━━━━━━━━━━━━━━━━━ 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/t10k-labels-idx1-ubyte.gz
5148/5148 ━━━━━━━━━━━━━━━━━━━━ 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/t10k-images-idx3-ubyte.gz
4422102/4422102 ━━━━━━━━━━━━━━━━━━━━ 0s 0us/step
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,)
```

3.3 Codificação one-hot das saídas

Como temos um problema de classificação multiclasse é mais adequado transformar as saídas em vetores one-hot.

```
# 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)

# Mostra primeiros 5 exemplos de treinamento
print('Saída categórica:', y_train[:5])
print('Saída one-hot:\n', y_train_hot[:5].numpy())
```

```
Saída categórica: [9 0 0 3 0]
Saída one-hot:
 [[0. 0. 0. 0. 0. 0. 0. 0. 0. 1.]
 [1. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [1. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 1. 0. 0. 0. 0. 0. 0.]
 [1. 0. 0. 0. 0. 0. 0. 0. 0. 0.]]
```

3.4 Configuração da RNA

Para incluir essa função de custo na forma de camada, podemos criar a RNA na forma sequencial ou funcional. Vamos usar a forma funcional.

Vamos criar uma RNA com 3 camadas densas. As duas primeiras com função de ativação ReLu e a camada de saída com função de ativação softmax.

O processamento das imagens será realizado dentro da rede.

Vamos incluir muitos neurônios na rede para termos problema de overfitting.

```
# Importa classe de modelos do Keras
from tensorflow.keras.models import Model

# Define fator de regularização de ativações
lamb = 1.0e-5
```

```python
# Inclui camada de entrada
inputs = layers.Input(shape=(28,28))

# Inclui camada de Flatten
x = layers.Flatten()(inputs)

# Inclui camada Lambda de normalização
norm_layer = layers.Lambda(lambda x: x/255.)
x = norm_layer(x)

# Inclui 1a camada densa
x = layers.Dense(512, activation="relu")(x)

# Insere camada para calcular função de custo de regularização
x = ActivityRegularizationLayer(lamb=lamb)(x)

# Adiciona 2a camada densa
x = layers.Dense(256, activation="relu")(x)

# Insere camada para calcular função de custo de regularização
x = ActivityRegularizationLayer(lamb=lamb)(x)

# Adiciona camada de saída de classificação
outputs = layers.Dense(10, activation='softmax')(x)

# Define modelo
rna = Model(inputs=inputs, outputs=outputs)

# Sumário do modelo
rna.summary()
```

```
Model: "functional"
```

```
┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━┓
┃ Layer (type)                         ┃ Output Shape                ┃ Param #         ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━┩
│ input_layer (InputLayer)             │ (None, 28, 28)              │               0 │
├──────────────────────────────────────┼─────────────────────────────┼─────────────────┤
│ flatten (Flatten)                    │ (None, 784)                 │               0 │
├──────────────────────────────────────┼─────────────────────────────┼─────────────────┤
│ lambda (Lambda)                      │ (None, 784)                 │               0 │
├──────────────────────────────────────┼─────────────────────────────┼─────────────────┤
```

```
──────────────────┤
│ dense (Dense)                        │ (None, 512)                 │
401,920 │
├──────────────────────────────────────┼─────────────────────────────┼
──────────────────┤
│ activity_regularization_layer        │ (None, 512)                 │
0 │
│ (ActivityRegularizationLayer)        │                             │
│
├──────────────────────────────────────┼─────────────────────────────┼
──────────────────┤
│ dense_1 (Dense)                      │ (None, 256)                 │
131,328 │
├──────────────────────────────────────┼─────────────────────────────┼
──────────────────┤
│ activity_regularization_layer_1      │ (None, 256)                 │
0 │
│ (ActivityRegularizationLayer)        │                             │
│
├──────────────────────────────────────┼─────────────────────────────┼
──────────────────┤
│ dense_2 (Dense)                      │ (None, 10)                  │
2,570 │
└──────────────────────────────────────┴─────────────────────────────┴
──────────────────┘

 Total params: 535,818 (2.04 MB)

 Trainable params: 535,818 (2.04 MB)

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

3.5 Compilação e treinamento da rede

Para compilar a rede vamos usar a seguinte configuração:

- Otimizador: Adam
- Taxa de aprendizado: 0.001
- Função de custo: `categorical crossentropy`
- Métrica: `accuracy`

Para treinar a rede vamos usar 40 épocas e lotes com 1024 exemplos.

```python
# Importa classe dos otimizadores
from tensorflow.keras import optimizers

# Compilação da rede
adam = optimizers.Adam(learning_rate=0.001)
rna.compile(optimizer=adam, loss='categorical_crossentropy',
metrics=['accuracy'])
```

```
# treinamento da rede
results = rna.fit(x_train, y_train_hot, batch_size=1024, epochs=40,
validation_data=(x_test, y_test_hot))
```

```
Epoch 1/40
59/59 ━━━━━━━━━━━━━━━━━━━━ 6s 47ms/step - accuracy: 0.6651 - loss:
1.5928 - val_accuracy: 0.8362 - val_loss: 0.7466
Epoch 2/40
59/59 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.8555 - loss:
0.6719 - val_accuracy: 0.8589 - val_loss: 0.5910
Epoch 3/40
59/59 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.8739 - loss:
0.5431 - val_accuracy: 0.8685 - val_loss: 0.5346
Epoch 4/40
59/59 ━━━━━━━━━━━━━━━━━━━━ 1s 5ms/step - accuracy: 0.8863 - loss:
0.4751 - val_accuracy: 0.8712 - val_loss: 0.4918
Epoch 5/40
59/59 ━━━━━━━━━━━━━━━━━━━━ 1s 5ms/step - accuracy: 0.8966 - loss:
0.4238 - val_accuracy: 0.8739 - val_loss: 0.4692
Epoch 6/40
59/59 ━━━━━━━━━━━━━━━━━━━━ 1s 5ms/step - accuracy: 0.9009 - loss:
0.3980 - val_accuracy: 0.8776 - val_loss: 0.4451
Epoch 7/40
59/59 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9045 - loss:
0.3694 - val_accuracy: 0.8752 - val_loss: 0.4440
Epoch 8/40
59/59 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9101 - loss:
0.3479 - val_accuracy: 0.8802 - val_loss: 0.4181
Epoch 9/40
59/59 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9145 - loss:
0.3324 - val_accuracy: 0.8844 - val_loss: 0.4102
Epoch 10/40
59/59 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9135 - loss:
0.3235 - val_accuracy: 0.8818 - val_loss: 0.4081
Epoch 11/40
59/59 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9171 - loss:
0.3114 - val_accuracy: 0.8816 - val_loss: 0.4069
Epoch 12/40
59/59 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9208 - loss:
0.2973 - val_accuracy: 0.8883 - val_loss: 0.3927
Epoch 13/40
59/59 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9222 - loss:
0.2866 - val_accuracy: 0.8813 - val_loss: 0.4056
Epoch 14/40
59/59 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9229 - loss:
0.2833 - val_accuracy: 0.8804 - val_loss: 0.3983
Epoch 15/40
59/59 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9254 - loss:
0.2742 - val_accuracy: 0.8841 - val_loss: 0.3892
```

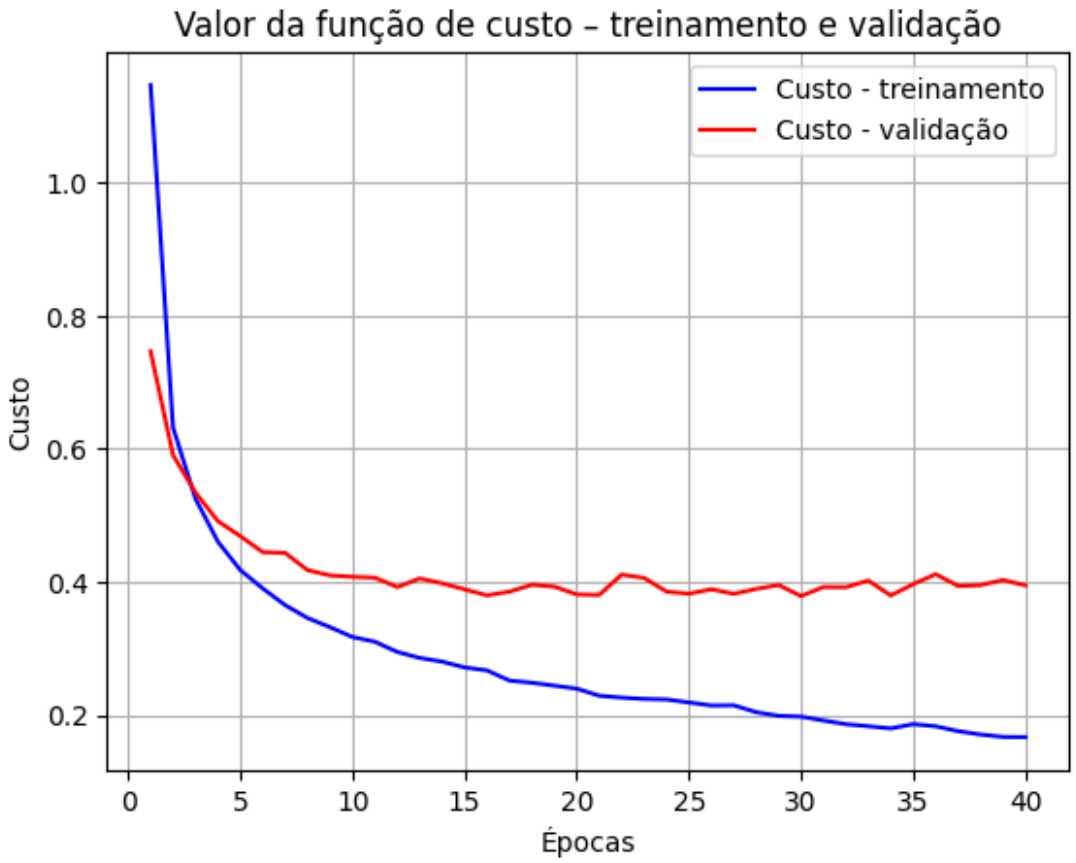
```
Epoch 16/40
59/59 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9261 - loss:
0.2688 - val_accuracy: 0.8866 - val_loss: 0.3803
Epoch 17/40
59/59 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9315 - loss:
0.2508 - val_accuracy: 0.8884 - val_loss: 0.3858
Epoch 18/40
59/59 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9300 - loss:
0.2502 - val_accuracy: 0.8837 - val_loss: 0.3961
Epoch 19/40
59/59 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9334 - loss:
0.2464 - val_accuracy: 0.8808 - val_loss: 0.3934
Epoch 20/40
59/59 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9326 - loss:
0.2424 - val_accuracy: 0.8873 - val_loss: 0.3817
Epoch 21/40
59/59 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9350 - loss:
0.2329 - val_accuracy: 0.8888 - val_loss: 0.3809
Epoch 22/40
59/59 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9380 - loss:
0.2256 - val_accuracy: 0.8800 - val_loss: 0.4114
Epoch 23/40
59/59 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9387 - loss:
0.2242 - val_accuracy: 0.8770 - val_loss: 0.4065
Epoch 24/40
59/59 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9379 - loss:
0.2255 - val_accuracy: 0.8848 - val_loss: 0.3859
Epoch 25/40
59/59 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9404 - loss:
0.2183 - val_accuracy: 0.8871 - val_loss: 0.3827
Epoch 26/40
59/59 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9417 - loss:
0.2100 - val_accuracy: 0.8849 - val_loss: 0.3894
Epoch 27/40
59/59 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9374 - loss:
0.2191 - val_accuracy: 0.8867 - val_loss: 0.3826
Epoch 28/40
59/59 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9447 - loss:
0.2012 - val_accuracy: 0.8843 - val_loss: 0.3899
Epoch 29/40
59/59 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9452 - loss:
0.1992 - val_accuracy: 0.8812 - val_loss: 0.3958
Epoch 30/40
59/59 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9454 - loss:
0.1947 - val_accuracy: 0.8862 - val_loss: 0.3792
Epoch 31/40
59/59 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9487 - loss:
0.1876 - val_accuracy: 0.8828 - val_loss: 0.3928
Epoch 32/40
```

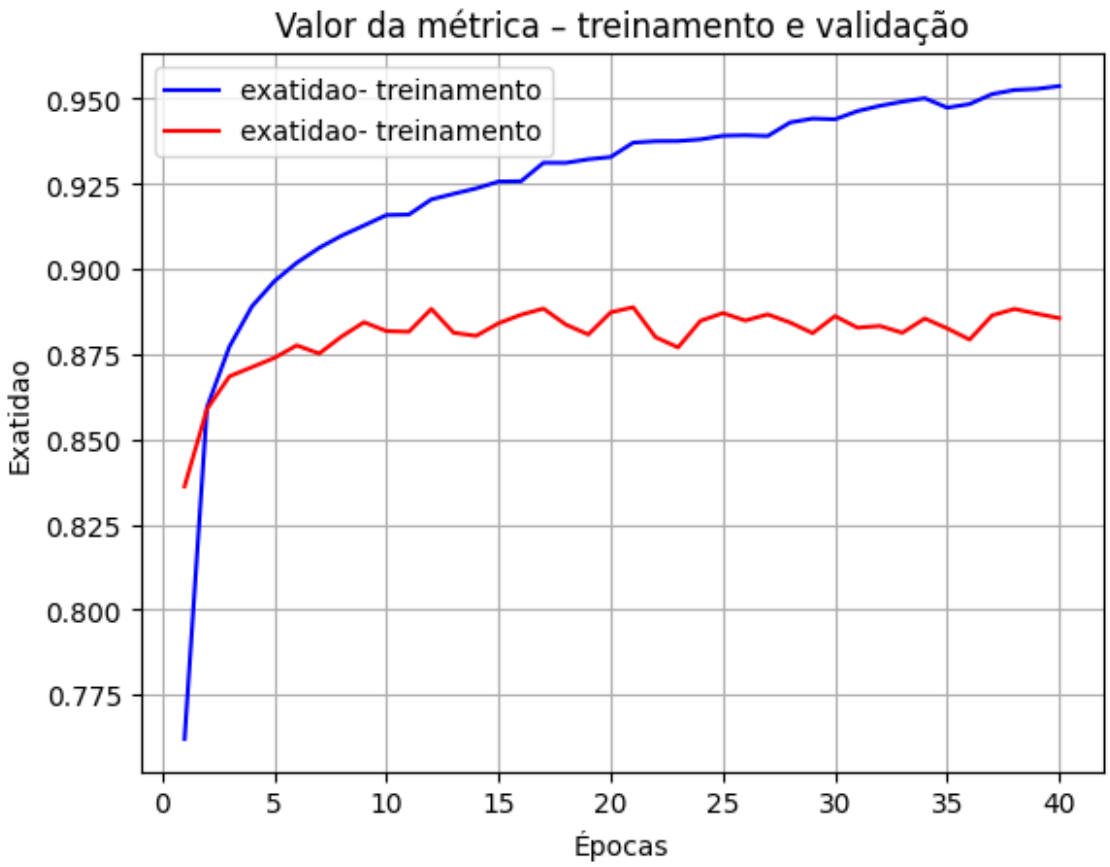
```
59/59 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9471 - loss:
0.1905 - val_accuracy: 0.8833 - val_loss: 0.3926
Epoch 33/40
59/59 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9484 - loss:
0.1847 - val_accuracy: 0.8813 - val_loss: 0.4024
Epoch 34/40
59/59 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.9507 - loss:
0.1796 - val_accuracy: 0.8855 - val_loss: 0.3801
Epoch 35/40
59/59 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9479 - loss:
0.1852 - val_accuracy: 0.8826 - val_loss: 0.3971
Epoch 36/40
59/59 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.9504 - loss:
0.1790 - val_accuracy: 0.8793 - val_loss: 0.4121
Epoch 37/40
59/59 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9514 - loss:
0.1780 - val_accuracy: 0.8864 - val_loss: 0.3944
Epoch 38/40
59/59 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9528 - loss:
0.1713 - val_accuracy: 0.8883 - val_loss: 0.3956
Epoch 39/40
59/59 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9529 - loss:
0.1697 - val_accuracy: 0.8869 - val_loss: 0.4033
Epoch 40/40
59/59 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9561 - loss:
0.1625 - val_accuracy: 0.8856 - val_loss: 0.3954
```

Gráfico do processo de trienamento

```python
def plot_train(history):# Salva treinamento na variável history para
visualização
    history_dict = history.history

    # Salva custos, métricas e epocas em vetores
    custo = history_dict['loss']
    acc = history_dict['accuracy']
    val_custo = history_dict['val_loss']
    val_acc = history_dict['val_accuracy']

    # Cria vetor de épocas
    epocas = range(1, len(custo) + 1)

    # Gráfico dos valores de custo
    plt.plot(epocas, custo, 'b', label='Custo - treinamento')
    plt.plot(epocas, val_custo, '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, acc, 'b', label='exatidao- treinamento')
    plt.plot(epocas, val_acc, 'r', label='exatidao- treinamento')
    plt.title('Valor da métrica – treinamento e validação')
    plt.xlabel('Épocas')
    plt.ylabel('Exatidao')
    plt.legend()
    plt.grid()
    plt.show()

plot_train(results)
```





3.6 Avaliação e teste da RNA

Para avaliar o desempenho da RNA vamos calcular a função de custo e a métrica e também calcular as saídas previstas e fazer um gráfico com as saídas reais e previstas para poder compará-las.

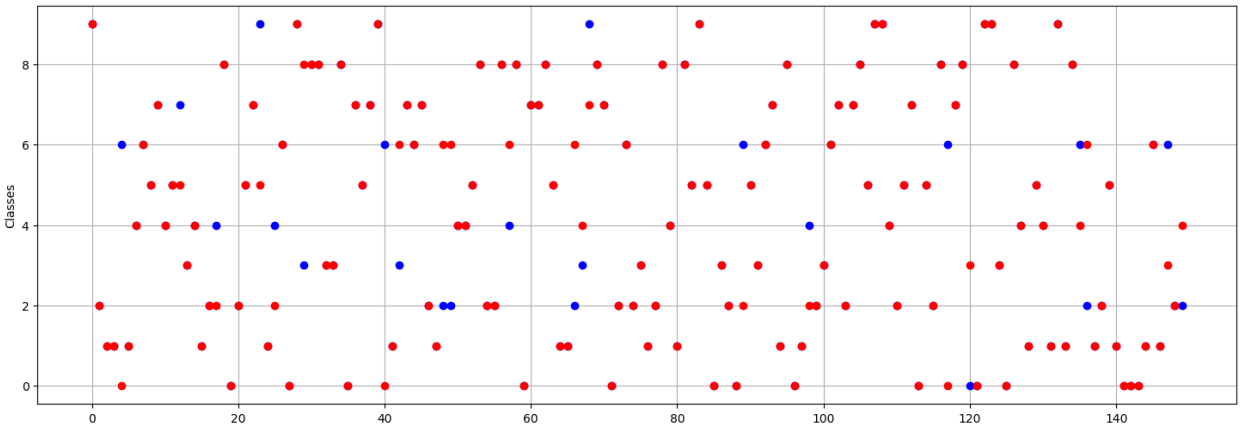
```
# Calcula função de custo e métrica
rna.evaluate(x_train, y_train_hot)
rna.evaluate(x_test, y_test_hot)
```

```
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 5s 2ms/step - accuracy: 0.9566 - loss:
0.1276
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.8834 - loss:
0.3652

[0.36484572291374207, 0.8855999708175659]
```

```
# Calcula saídas previstas
y_prev = rna.predict(x_test)

# Indentifa classe prevista
class_prev = np.argmax(y_prev, axis=1)
```

```
# Gráfico dos resultados
#plt.scatter(x, y)
plt.figure(figsize=(18,6))
plt.plot(y_test[:150], 'bo', label='Saídas reais')
plt.plot(class_prev[:150], 'ro', label='Saídas previstas')
plt.ylabel('Classes')
plt.grid()
plt.show()
```

```
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step
```



Qual é a influência do fator de regularição das ativações λ para eliminar overfitting?