Ex. Rede Neural simples com Tensor Flow

https://keras.io/examples/

Primeiro vamos entender um pouco o básico do funcionamento do tensorFlow

No tensor flow, os cálculos são organizados em grafos.

No tensorflow 1.0, era necessario criar uma session e executar (run). Ou seja, você criava os grafos de operações e depois tinha que executar esse grafo. Agora, no tensorflow 2.0, a execução é imediata (ager Execution). Não é necessário Session.

Essa biblioteca trabalha com o que chamamos de tensor.

O QUE É UM TENSOR?

Um tensor é basicamente uma matriz de números organizados em uma grade regular com um numero variável de eixos.

- Podem ser 0-dimensional (escalar), 1-dimensional (array), 2-dimensional (matrtiz)
- Um tensor pode ter n dimensões. Para representar imagens, por exemplo, usamos um tensor 4D, devido ao canal de cor (RGB) que deve ser representado.

Para exemplificar:

```
dim1 = tf.constant(10)

dim2 = tf.constant([1, 2, 3, 4, 5])

dim3 = tf.constant([[1, 2], [3, 4]])

print("dimensoes:")

print(dim1.shape)

print(dim2.shape)

print(dim3.shape)
```

Temos como saída:

```
dimensoes:
()
```

```
(5,)
(2, 2)
```

PLACEHOLDERS

No tensorflow, existia o que chamamos de "PlaceHolder", que é basicamente um espaço reservado (uma variavel vazia) para entrada de dados futuramente.

No momento do treinamento, podemos dividir nossos dados em pacotes para um treino mais rápido. Esses pacotes são preenchidos um de cada vez. Eles são os placeholders. Vejamos um exemplo:

```
#Meu banco de dados simulado pelo numpy

dados_x = np.random.randn(4, 8) # 4 linhas e 8 colunas

dados_y = np.random.randn(8,2)


placeH_x = tf.placeholder(tf.float32, shape=(4, 8))

placeH_y = tf.placeholder(tf.float32, shape=(8,2))

# Aqui estou criando meus placeHolders para inserir meus dados (dados_x e
dados_y)
```

Hoje, com o tensorflow 2.0 e o ager execution, os placeholders não são mais utilizados. Mas caso encontre algum conteúdo que cite essas estruturas, você sabe do que se trata.

EXEMPLO DE REDE NEURAL SIMPLES COM TF

```
import tensorflow as tf

tf.random.set_seed(2)
```

Define a semente aleatória para garantir a reprodutibilidade dos resultados.

Isso é importante para garantir que os resultados das operações que envolvem aleatoriedade sejam reprodutíveis.

Quando você define uma semente aleatória, qualquer operação aleatória que você executar (como inicialização de pesos, embaralhamento de dados, ou divisão de dados em lotes) produzirá os mesmos resultados a cada vez que o código for executado.

CRIANDO A ESTRUTURA DA REDE NEURAL SEQUENCIAL

As redes neurais podem ter diferentes tipos de camadas, como:

- Camadas densas (totalmente conectadas), usadas frequentemente em redes neurais feedforward.
- Camadas convolucionais, usadas principalmente em CNNs para tarefas de visão computacional.
- Camadas recorrentes, usadas em RNNs para processamento de sequências, como em NLP.
- E outros tipos de camadas, dependendo da necessidade.

CAMADA DE ENTRADA

Define a camada de entrada, passando uma tupla com o tamanho das entradas.

```
inp = tf.keras.Input((x_train.shape[1],))
```

- x_train.shape[1] é o número de colunas do nosso dataset.

CAMADAS OCULTAS

Camada oculta com 100 neurônios e função de ativação 'relu'

ReLU(x) = max(0, x): Para uma entrada x, a saída será x se x for maior que 0; caso contrário, a saída será 0.

```
hide = tf.keras.layers.Dense(100, activation='relu')(inp)

hide = tf.keras.layers.Dense(100, activation='relu')(hide)
```

CAMADA DE SAÍDA

Camada de saída com 1 neurônio e função de ativação 'sigmoid'

A função sigmoid é usada para transformar a saída em um valor entre 0 e 1.

```
out = tf.keras.layers.Dense(1, activation='sigmoid')(hide)
```

CRIAÇÃO DO MODELO

Cria o modelo especificando as entradas e saídas

```
modelo = tf.keras.Model(inputs=inp, outputs=out)
```

No Keras, existem 2 objetos principais para criar redes neurais: Sequential e Model (para redes mais complexas).

COMPILAÇÃO DO MODELO

Compila o modelo com o otimizador 'adam' e a função de perda 'msle'

'adam' é um otimizador eficiente de primeira ordem.

'msle' (Mean Squared Logarithmic Error) é uma função de perda adequada para dados com grandes variâncias.

```
modelo.compile(optimizer='adam', loss='msle')
```

VISUALIZAÇÃO DO MODELO

Exibe um resumo da arquitetura do modelo

```
modelo.summary()
```

TREINAMENTO DO MODELO

Treina o modelo nos dados de treinamento

```
modelo.fit(
    x_train, y_train,
    validation_data=(x_test, y_test),  # Dados de validação
    epochs=100,                        # Número de épocas
    shuffle=True,                      # Embaralha os dados a cada época
    batch_size=1,                      # Tamanho do lote
    validation_split=0.1               # Fração dos dados de treinamento
usada como validação
)
```

- **Batch size:** Tamanho dos lotes em que os dados são divididos para treinamento.

- **Batches** são pequenos subconjuntos dos dados de treinamento usados para atualizar os pesos da rede neural durante o treinamento. Em vez de usar todo o conjunto de dados (como no gradiente descendente tradicional) ou apenas um único exemplo de cada vez (como no gradiente estocástico), o treinamento com mini batches divide os dados em partes menores e processa essas partes em cada iteração.
- **Epochs:** Número de passadas completas sobre o conjunto de dados.
- **Shuffle:** Torna a ordem dos dados aleatória a cada época para melhorar a generalização.