

PROJETO PRÁTICO

PLAYGROUND TENSORFLOW

<https://playground.tensorflow.org>

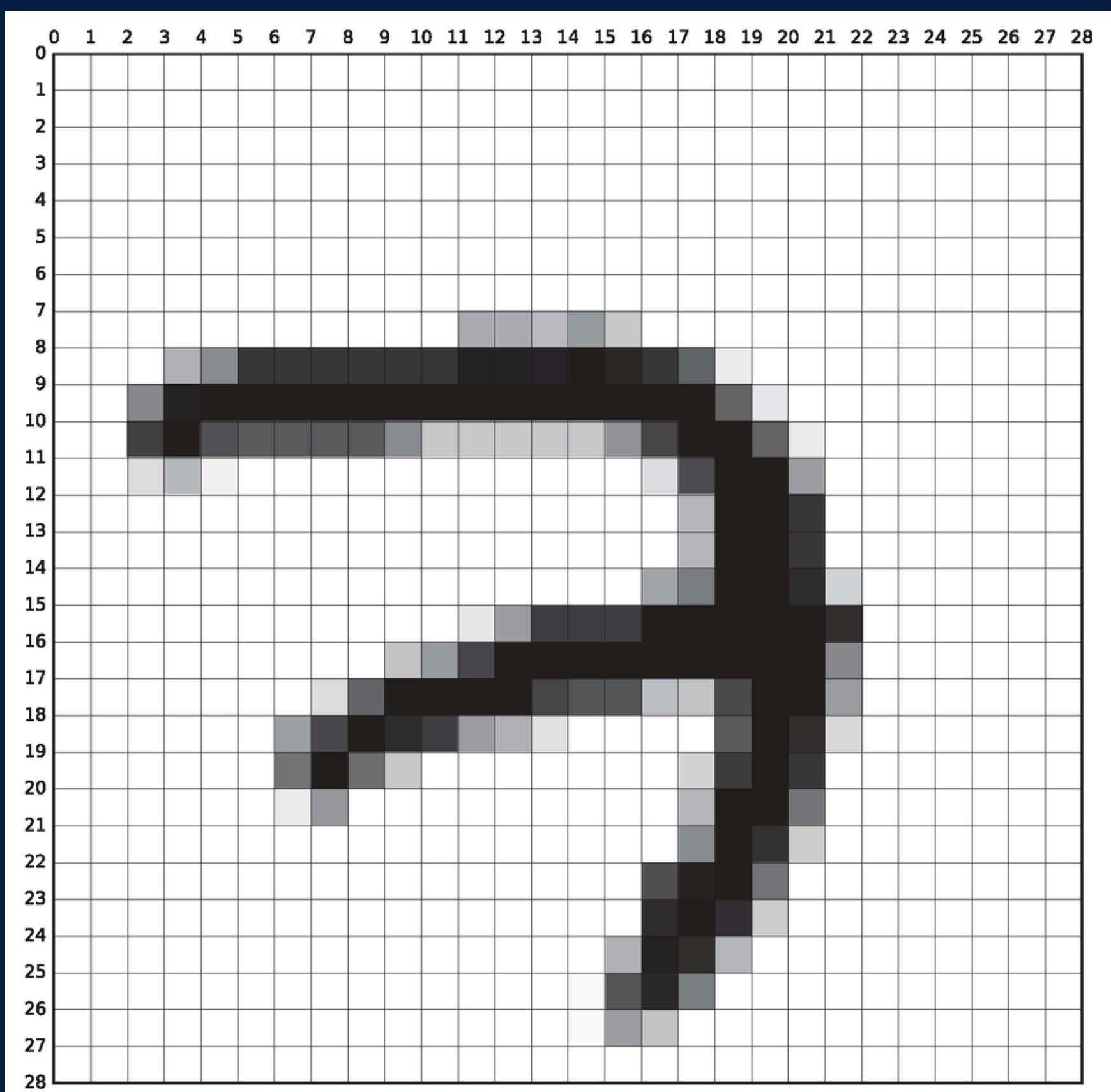
VAMOS CONSTRUIR UMA REDE NEURAL?

O QUE PRECISAMOS?



QUAL O PROBLEMA?

0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9



VAMOS PARA O CÓDIGO?



```
import numpy as np

class RedeNeuralProfunda():
    """
    Classe para a rede neural profunda.
    """
    def __init__(self, arquitetura, caminho_modelo=None, epochas=10, taxa_aprendizado=0.05):
        """
        Inicializa a rede neural profunda.

        Parâmetros:
        - arquitetura: lista com a arquitetura da rede
        - caminho_modelo: caminho para o arquivo de modelo (matrizes numpy .npz)
        - epochas: número de épocas de treinamento
        - taxa_aprendizado: taxa de aprendizado
        """
        self.arquitetura = arquitetura
        self.num_camadas = len(self.arquitetura)
        self.epochas = epochas
        self.taxa_aprendizado = taxa_aprendizado
        self.parametros = {}
```



```
# Carregar pesos:  
if caminho_modelo:  
    pesos = np.load(caminho_modelo, allow_pickle=True) # pesos em arquivo .npz  
    for chave, valor in pesos.items():  
        self.parametros[chave] = valor  
else:  
    self.inicializarPesos()  
  
# Inicializar matrizes vazias:  
for i in range(self.num_camadas):  
    if not i == 0:  
        self.parametros[f"Z{i}"] = None # armazenar somas ponderadas  
        self.parametros[f"N{i}"] = None # armazenar valores das ativações
```



```
def inicializarPesos(self, semente=42):
    """
    Inicializa os pesos da rede neural.
    """
    np.random.seed(semente)

    for i in range(1, self.num_camadas):
        self.parametros[f"P{i}"] = np.random.randn(self.arquitetura[i], self.arquitetura[i-1]) # inicializa pesos
para cada camada
```



```
def salvarPesos(self, caminho="pesos.npz"):  
    """  
        Salva os pesos da rede neural em um arquivo .npz.  
  
    Parâmetros:  
    - caminho: caminho para o arquivo de saída  
    """  
    np.savez(caminho, **self.parametros)
```



```
def sigmoid(self, x, derivada=False):
    """
    Função de ativação sigmoid.

    Parâmetros:
    - x: entrada
    - derivada: se True, retorna a derivada da função sigmoid
    """
    if derivada:
        return (np.exp(-x))/((np.exp(-x)+1)**2)
    return 1/(1 + np.exp(-x))
```



```
def softmax(self, x, derivada=False):
    """
    Função de ativação softmax.

    Parâmetros:
    - x: entrada
    - derivada: se True, retorna a derivada da função softmax
    """
    exps = np.exp(x - x.max())
    if derivada:
        return exps / np.sum(exps, axis=0) * (1 - exps / np.sum(exps, axis=0))
```



```
def propagar(self, entrada):
    """
    Propaga a entrada para a rede neural.

    Parâmetros:
    - entrada: entrada da rede
    """
    parametros = self.parametros

    parametros["N0"] = entrada

    for cont in range(1, self.num_camadas):
        parametros[f"Z{cont}"] = np.dot(parametros[f"P{cont}"], parametros[f"N{cont-1}"])
        parametros[f"N{cont}"] = self.sigmoid(parametros[f"Z{cont}"])

    return parametros[f"N{self.num_camadas - 1}"]
```



```
def retroPropagar(self, saida, target):
    """
    Retro-propaga o erro para a rede neural.

    Parâmetros:
    - saida: saída da rede
    - target: target da rede
    """
    parametros = self.parametros
    ajustar_pesos = {}

    erro = 2 * (target - saida) / target.shape[0] * self.softmax(self.parametros[f"Z{self.num_camadas - 1}"]),
    derivada=True)
    ajustar_pesos[f"P{self.num_camadas - 1}"] = np.outer(erro, parametros[f"N{self.num_camadas - 2}"])

    for cont in range(self.num_camadas - 1, 2, -1):
        erro = np.dot(parametros[f"P{cont}"].T, erro) * self.sigmoid(parametros[f"Z{cont - 1}"], derivada=True)
        ajustar_pesos[f"P{cont - 1}"] = np.outer(erro, parametros[f"N{cont - 2}"])

    return ajustar_pesos
```



```
def atualizarParametros(self, ajustar_para_pesos):
    """
    Atualiza os pesos da rede neural.

    Parâmetros:
    - ajustar_para_pesos: dicionário com os pesos a serem atualizados
    """
    for chave, valor in ajustar_para_pesos.items():
        self.parametros[chave] -= self.taxa_aprendizado * valor
```



```
def calcularAcuracia(self, x_entrada, y_saida):
    """
    Calcula a acurácia da rede neural.

    Parâmetros:
    - x_entrada: entradas da rede
    - y_saida: saídas
    """
    predicao = []

    for x, y in zip(x_entrada, y_saida):
        saida = self.propagar(x)
        pred = np.argmax(saida)
        predicao.append(pred == np.argmax(y))

    return np.mean(predicao)
```



```
def treinamento(self, x_treino, y_treino, x_teste, y_teste):
    """
    Treina a rede neural.

    Parâmetros:
    - x_treino: entradas de treinamento
    - y_treino: saídas de treinamento
    - x_teste: entradas de teste
    - y_teste: saídas de teste
    """
    for interacao in range(self.epocas):
        for x, y in zip(x_treino, y_treino):
            saida = self.propagar(x)
            ajustar_para_pesos = self.retroPropagar(y, saida)
            self.atualizarParametros(ajustar_para_pesos)

    acuracia = self.calcularAcuracia(x_teste, y_teste)
    print(f"Época: {interacao+1} | Acurácia: {acuracia * 100}")
```

DÚVIDAS?

$$MSE = \frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2$$