No início, te falei sobre a biblioteca TensorFlow.js e disse que ela seria a base para as demonstrações deste módulo

Então para começar, Tensorflow tem este nome, pois trabalha com Tensores, que são vetores, ou listas em JavaScript em que seu conteúdo é representado por números para posteriormente serem usados por algoritmos que aprendem com padrões de dados

Vamos pensar em um exemplo de um array de pessoas, onde tem sua idade, cor preferida, e localização, cada pessoa aqui tem uma categoria. Dado as propriedades de uma nova pessoa, você quer determinar em qual categoria ela mais se encaixa baseado no histórico do banco de dados

(animação)

```
const pessoas = [
  { nome: "Erick", idade: 30, cor: "azul", localizacao: "São Paulo" },
  { nome: "Ana", idade: 25, cor: "vermelho", localizacao: "Rio" },
  { nome: "Carlos", idade: 40, cor: "verde", localizacao: "Curitiba" }
];

const categorias = [
  'Erick', 'premium',
  'Ana', 'medium',
  'Carlos', 'basic'
]
```

Como eu falei, Tensorflow.js não entende objetos ou instâncias primitivas de JavaScript, você necessariamente precisa converter para uma estrutura que contenha apenas números, um Tensor

Para simplificar, vamos trabalhar em cima do array de pessoas, para transformar esse array em um Tensor podemos criar uma tabela e adicionar pontos para propriedades que estão ativas

Batendo o olho, temos 3 cores diferentes, então, Erick tem cor preferida azul, o azul recebe 1 e todo o resto zero, Ana vermelho, vermelho recebe 1 e zero para o resto e asim por diante

	azul	vermelho	verde
Erick	1	0	0
Ana	0	1	0
Carlos	0	0	1

Então, um tensor para cada item seria

Erick = [1, 0, 0]
Ana = [0, 1, 0]
Carlos = [0, 0, 1]

Agora que você entendeu a lógica, vamos avançar, vamos colocar cada um dos campos de pessoa na tabela, assim como a idade e vai ficar dessa forma

	idade	azul	vermelho	verde	São Paulo	Rio	Curitiba
Erick	30	1	0	0	1	0	1
Ana	25	0	1	0	0	1	0
Carlos	40	0	0	1	0	0	1

Em termos práticos, esse processo se chama categorização e repare que seu array de pessoas, agora é convertido por um array de arrays, ou um tensor de duas dimensões, usando apenas números

(animação - antes - flechinha)

```
const pessoas = [
  { nome: "Erick", idade: 30, cor: "azul", localizacao: "São Paulo" },
  { nome: "Ana", idade: 25, cor: "vermelho", localizacao: "Rio" },
  { nome: "Carlos", idade: 40, cor: "verde", localizacao: "Curitiba" }
];
```

depois

```
const tensorPessoas = [
  [30, 1, 0, 0, 1, 0], // Erick
  [25, 0, 1, 0, 0, 1], // Ana
  [40, 0, 0, 1, 1, 0]  // Carlos
];
```

Tensores em termos mais bonitos, são vetores multidimensionais, no caso de pessoas, usamos apenas duas dimensões, ou seja, um array tem um array que cada linha representa uma pessoa

Tá, olhando nosso novo tensor de pessoas, existe um problema, todos os números vão de zero a um, menos o campo de idade, e isso pode ser problemático para os algoritmos, então

precisamos realizar uma normalização, uma função de ativação para transformar essas idades entre o numero 0 e 1

Como fazemos então? Bom, é fácil, usamos essa regrinha básica isso:

idade_normalizada = (idade - idade_min) / (idade_max - idade_min)

Olhando nossa tabela, Ana, é zero pois é a idade mínima, Carlos é um pois tem a idade máxima e Erick ficou em 0.33 pois está entre os dois

	idade	Idade normalizada
Erick	30	(30-25)/(40-25) = 0.33
Ana	25	(25-25)/(40-25) = 0
Carlos	40	(40-25)/(40-25) = 1

Então aplicando a normalização, agora nosso tensor sai do objeto contendo as idades, para as idades normalizadas

Antes

```
const tensorPessoas = [
  [30, 1, 0, 0, 1, 0], // Erick
  [25, 0, 1, 0, 0, 1], // Ana
  [40, 0, 0, 1, 1, 0]  // Carlos
];
```

Depois

```
[
  [0.33, 1, 0, 0, 1, 0], // Erick
  [0,    0, 1, 0, 0, 1], // Ana
  [1,    0, 0, 1, 1, 0]  // Carlos
]
```

Toda vez que você for trabalhar com processamento de máquina ou machine learning, acontece essa etapa de normalização e conversão dos dados para tensores, é dessa forma que a máquina consegue aprender padrões, prever situações e recomendar produtos

Na próxima aula, vamos avançar ainda mais a partir deste ponto, você vai entender o que são redes neurais, porque se diz “treinar uma rede neural” e vou continuar esse exemplo treinando a rede para classificar a categoria de uma pessoa

Efeito fade

Agora que você já entendeu como transformar seus dados em tensores e normalizar tudo para trabalhar entre 0 e 1, chega a parte mais mágica (e divertida!) do machine learning: **as redes neurais**.

https://www.notablecap.com/blog/the-anatomy-of-a-neural-network

Como funciona uma rede neural no TensorFlow.js?

Uma rede neural é como se fosse um **monte de operações matemáticas** conectadas, tentando encontrar padrões escondidos nos dados.

Pegando o exemplo da aula anterior, você vai enviar essa lista de pessoas para uma rede neural, vai alimentar ela com qual categoria esses clientes estão inclusos e então vai criar uma nova pessoa e o algoritmo vai tentar determinar em qual categoria essa pessoa tem mais probabilidade de ser

Vamos ver de uma forma mais prática. Voltando à aula anterior, tinhamos essa lista:

```
const pessoas = [
  { nome: "Erick", idade: 30, cor: "azul", localizacao: "São Paulo" },
  { nome: "Ana", idade: 25, cor: "vermelho", localizacao: "Rio" },
  { nome: "Carlos", idade: 40, cor: "verde", localizacao: "Curitiba" }
];
```

Que foi convertida para um tensor normalizado como:

```
const tensorPessoas = [
  [0.33, 1, 0, 0, 1, 0], // Erick
  [0,    0, 1, 0, 0, 1], // Ana
  [1,    0, 0, 1, 1, 0]  // Carlos
]
```

Você pode estar se perguntando, pow, mas a idade ali ficou zero, não vai dar ruim? Como idade é uma faixa de idade, ele vai de zero a um, mas para os outros, não faz sentido a pessoa ter meia cor preferida ou morar em meia localização, por isso para cor e localização fixamos zero e um, a fins de curiosidade, isso é chamado de one hot take, quando só pode ser uma das opções

Agora, precisamos informar o algoritmo, qual é a categoria dos usuários que já temos, e fazemos a normalização, então dado que temos as **categorias**

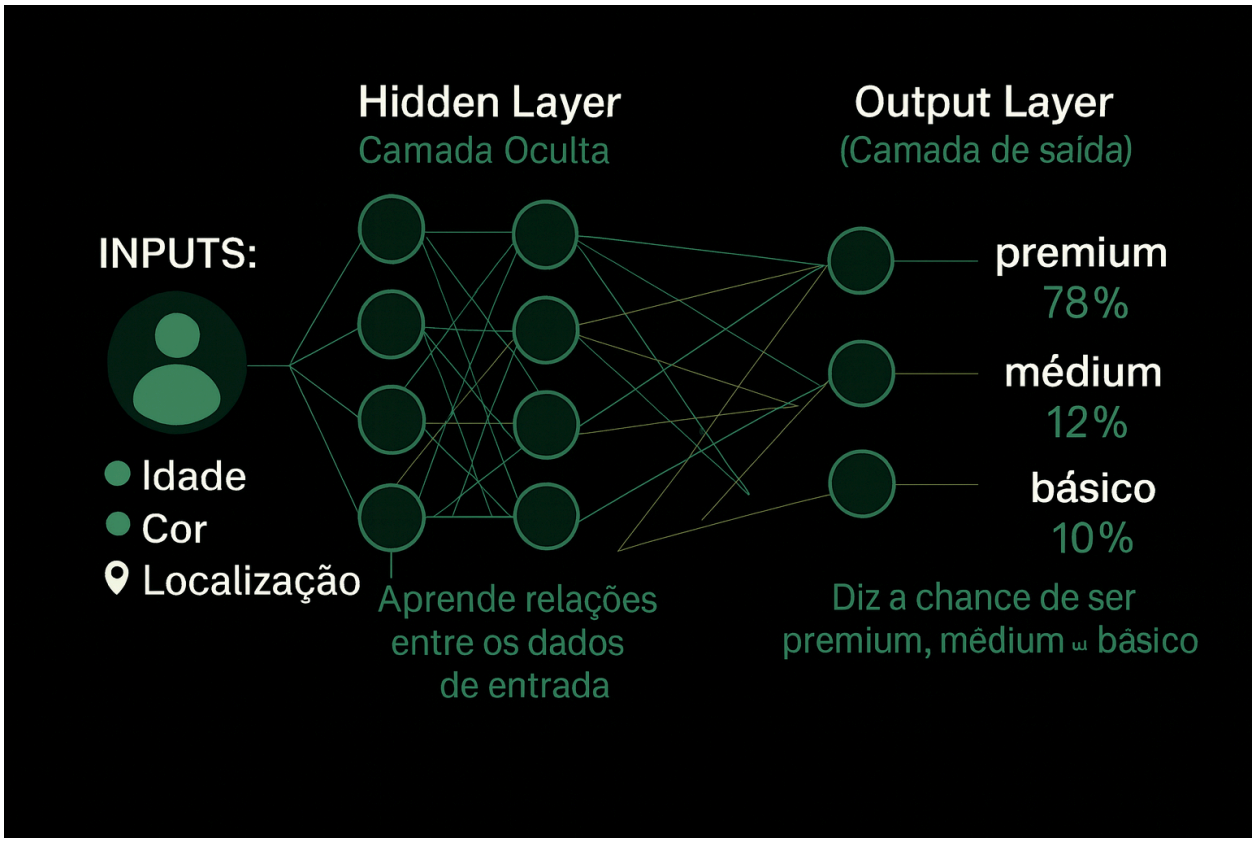
const categorias = ["premium", "medium", "basico"];

Transformamos ela em um tensor e cada linha, vai ser mapeada com cada pessoa existente, ou seja, o primeiro é o Erick, sendo categoria premium, Ana medium e assim por diante

```
const tensorCategorias = [
    [1, 0, 0], // premium - Erick
    [0, 1, 0], // basico - Ana
    [0, 0, 1]  // outro - Carlos
];
```

Pra que fazer isso? Esse mapeamento diz para o algoritmo o que ele deve retornar, você passa os usuários como entrada, diz quais categorias eles pertencem e agora ela deve aprender a retornar a probabilidade de um usuário ser de uma categoria específica neste mesmo formato

(colocar eu lado a lado desta imagem)



Uma rede neural geralmente é representada com imagens como essa.
Cada **input** vira um parâmetro — por exemplo, idade, cor e localização da pessoa — e é enviado para a primeira camada do modelo.

Essa primeira etapa, chamada de **camada oculta**, é composta por vários “neurônios”, que são como minicalculadoras matemáticas.

Cada neurônio recebe todos os valores de entrada, faz suas próprias continhas internas (misturando idade, cor, localização, etc.) e gera um novo número de saída.

Ali por exemplo, cada bolinha de camada oculta, representa um neurônio, neste exemplo, eu usei apenas 4 neurônios, mais por visualização, mas na aula de criar um sistema de recomendação, vamos usar mais de 120 neurônios diferentes, essas quantidades de neurônios, você vai refinando com o tempo, comparando se os resultados que sairam estão de acordo com o que você estava esperando

Essas saídas então viram a entrada para a próxima camada, cada camada vira como se fosse um filtro do passo anterior, se houver mais de uma camada oculta, esse processo se repete, cada vez extraindo padrões mais complexos dos dados

Cada linha colorida mostra como todos os inputs estão conectados a todos os neurônios da camada seguinte, formando esse emaranhado de conexões que permite a rede aprender.

Depois, vem a **camada de saída**, onde cada neurônio representa uma categoria possível:

- Premium
- Medium
- Basic

O papel dessa camada é transformar tudo o que foi aprendido pelas camadas anteriores em **probabilidades** — indicando, por exemplo, que este perfil tem 78% de chance de ser Premium, 12% de ser Medium, e 10% de ser Basic.

(voltar apenas para a câmera)
É assim que a rede decide qual categoria tem mais a ver com aquele perfil.

Resumindo:
O dado entra, passa por um monte de conexões e cálculos, e o modelo te entrega uma resposta — não só dizendo qual é o resultado mais provável, mas também mostrando o quanto ele “acredita” em cada categoria!

Efeito fade

Por que se fala em “treinar” a rede?

Treinar nada mais é do que mostrar **vários exemplos** (como fizemos com Erick, Ana e Carlos), e deixar o algoritmo ajustar sozinho os “pesos” internos dos neurônios, até conseguir acertar o máximo possível nos resultados.

Quanto mais exemplos variados, melhor ela vai aprender.

Print e zoom no **trained on 60 million hours of data**

Olha esse exemplo do SensorLM do google, para gerar esse modelo final, foram necessários mais de 60 milhões de horas de dados de sensores de pessoas para que o modelo fosse possível de determinar se a pessoa teve variação de risco de batimento cardíaco, temperatura e mais

Por isso, pra problemas reais, o segredo está na quantidade **e diversidade de dados**!

Mas para nós, Web developers, na prática, geralmente usamos modelos prontos e podemos até estendê-los, o trabalho de treinar um modelo, gerar o arquivo final para que alguém possa reutilizá-lo depois, é trabalho de engenheiros de machine learning, normalmente uma galera que vem do Python.

Só que nós Web developers, somos a galera do JavaScript e com JavaScript tudo é possível!

Na próxima aula, vou cair na prática em como criar essa rede neural, usando o mesmo contexto que vimos aqui, para ficar mais claro como isso funciona

Vai ser importante para você entender conceitos fundamentais antes de cair para os modelos prontos de machine learning, vem comigo que vai val a pena!