

Explicando nossa rede densa linha por linha

Importando bibliotecas

Nas primeiras linhas, estamos apenas importando as bibliotecas que serão utilizadas. Se você já estuda *Python* há algum tempo, provavelmente já fez isso inúmeras vezes, então não há muito o que explicar nessa parte. O uso do **as** serve apenas para facilitar na hora de chamar as funções — basicamente, estamos dando um nome mais curto para essas operações.

```
import torch  
  
import torch.nn as nn  
  
import torch.optim as optim
```

Criando a rede

Aqui vamos criar uma classe chamada `RedeSoma`. Antes que você se pergunte: “Por que não usar o paradigma estruturado?”, eu já adianto — o *PyTorch* funciona bem com os dois paradigmas: o estruturado e o orientado a objetos. Neste caso, estamos optando por criar uma classe porque isso torna o código mais organizado, facilita a alteração dos hiperparâmetros e deixa tudo mais escalável e fácil de entender.

Tudo o que você precisa saber por agora é que estamos criando uma classe chamada `RedeSoma` e que ela herda de `nn.Module`, ou seja, ela se comporta como uma rede neural. Em seguida, usamos o método `__init__()` para construir a arquitetura da rede, com a camada de entrada, camada oculta e por fim a camada de saída.

Não se preocupe com todos os detalhes da programação orientada a objetos (POO) neste momento — o importante é entender que estamos definindo a rede dessa forma para que ela funcione corretamente dentro do *PyTorch*.

```
class RedeSoma(nn.Module):  
  
    def __init__(self, cam_entrada, cam_oculta, cam_saida):  
        super(RedeSoma, self).__init__()
```

Conectando as camadas

No PyTorch, temos a função `nn.Linear()` (ou `torch.nn.Linear()`). Ela cria uma **camada totalmente conectada** da rede neural.

No nosso código, quando fazemos:

```
self.fc1 = nn.Linear(cam_entrada, cam_oculta),
```

estamos criando a **primeira camada da rede** — que conecta todos os neurônios da camada de entrada a todos os neurônios da camada oculta.

A camada de entrada recebe os valores que colocamos e os envia para a camada oculta.

Para entender melhor essa ideia, recomendo que você dê uma olhada no PDF sobre redes densas, que explica isso com mais detalhes.

Quando chegamos na segunda linha,

```
self.fc2 = nn.Linear(cam_oculta, cam_saida),
```

estamos criando a **segunda camada da rede**. Ou seja, a camada oculta vai passar os valores já processados para a camada de saída, que nos retorna o resultado final — no nosso caso, um tensor contendo as respostas das somas.

Em resumo, nessas duas linhas, estamos conectando a camada de entrada à camada oculta e, por fim, conectando a camada oculta à camada de saída.

```
self.fc1= nn.Linear(cam_entrada,cam_oculta)  
self.fc2= nn.Linear(cam_oculta,cam_saida)
```

Função de perda

Quando definimos a função *forward*, estamos explicando como os dados vão passar pela rede.

O parâmetro `x` representa esses dados. Agora, vamos detalhar passo a passo o que acontece:

Primeiro, na linha:

```
x = torch.relu(self.fc1(x))
```

os dados passam pela primeira camada da rede, que faz uma transformação matemática: multiplica os dados por pesos e soma um viés.

A função de ativação *ReLU* é importante para adicionar **não-linearidade** ao modelo. Como aprendemos no módulo 2, isso permite que a rede aprenda padrões mais complexos.

Você pode estar se perguntando: “O que o *ReLU* faz?”

Explicando de forma simples e objetiva: ele transforma todos os valores negativos em zero e mantém os positivos. Isso evita que os neurônios fiquem inativos.

Na próxima linha:

```
x = self.fc2(x)
```

os dados já processados são passados para a camada de saída, que transforma esses valores em uma saída do tamanho definido — no nosso caso, um único número.

Por fim, na última linha da função, usamos o comando:

```
return x
```

que devolve a saída calculada. Essa saída pode ser mostrada ao usuário ou usada na função de perda durante o treinamento para ajustar os pesos da rede.

É importante lembrar que sempre que você chamar `RedeSoma()` com novos dados, o *PyTorch* executa automaticamente essa função para processar esses dados e gerar uma saída.

Entradas e saídas

Criamos a variável “entradas” que vai receber o seguinte tensor:

```
entradas = torch.tensor([[2.0, 2.0],  
                         [4.0, 4.0],  
                         [8.0, 8.0],  
                         [16.0, 16.0]])
```

Cada linha desse tensor tem dois valores. Esses são os números que queremos que a rede aprenda a somar. Por exemplo, na primeira linha temos 2 e 2, ou seja, é como se fosse $2 + 2$.

Você pode estar se perguntando: “Ué, como assim? Não vejo nenhum sinal de soma.”

E você está certo! Não tem mesmo. O que acontece é que a rede vai perceber, com o tempo, que esses dois valores devem ser somados porque as saídas esperadas — que estão no tensor `saidas` — mostram exatamente isso, veja abaixo:

```
saidas = torch.tensor([[4.0],  
                         [8.0],  
                         [16.0],  
                         [32.0]])
```

Se você reparar, as **entradas** são os dados que a rede vai receber durante o treinamento, e as **saídas** são os resultados que queremos que ela aprenda a retornar. Ou seja: $2 + 2 = 4$, $4 + 4 = 8$ e assim por diante.

Isso vai ficar mais claro quando chegarmos ao *loop de treinamento* da rede. Nele, a cada *epoch*, a rede compara a resposta que ela previu com a resposta correta que gostaríamos que ela tivesse dado — e essas respostas corretas estão armazenadas na variável `saídas`, que guarda o tensor com os resultados esperados.

Essa comparação é feita por meio da *função de perda*, que usa um critério — uma espécie de verificação entre a saída prevista e a saída esperada.

Mas não se preocupe com isso agora — vamos tratar dessa parte mais adiante na explicação.

Hiperparâmetros

Mais adiante no código, vamos precisar definir os hiperparâmetros da rede neural. Você deve estar se perguntando o que são os tais “hiperparâmetros”, mas não se preocupe, vamos explicar.

Durante o treinamento, a rede ajusta **os pesos** automaticamente — isso acontece sem que precisemos mexer manualmente. Porém, existem certas decisões que a rede **não consegue tomar sozinha**, e aí entra o programador: somos nós que definimos, por exemplo, o número de neurônios em cada camada, quantas épocas de treinamento usar ou qual critério adotar para otimização.

Tente pensar nos hiperparâmetros como um “**ajuste externo**”: são escolhas feitas fora do processo de aprendizado automático da rede. Em resumo, **hiperparâmetros são ajustes definidos pelo programador para guiar o treinamento**.

No nosso código, começamos definindo os hiperparâmetros relacionados ao número de neurônios em cada camada, como mostrado na imagem abaixo:

```
input_camada=2  
  
hidden_camada=10  
  
saida_camada=1
```

Na primeira linha, *input_camada*, definimos que a camada de entrada terá apenas **2 neurônios**, pois iremos somar apenas dois números. Assim, cada neurônio da entrada receberá um valor.

Em seguida, temos a *hidden_camada*, ou **camada oculta**, que será responsável por processar os dados de entrada e aplicar transformações. No código, ela foi definida com **10 neurônios**. É importante tomar cuidado com esse hiperparâmetro: se exagerarmos no número de neurônios, a rede pode “explodir” — não literalmente! Nesse caso, dizemos que a rede “explode” quando não consegue ajustar os pesos, ficando desordenada e retornando um tensor de *NaN* (*Not a Number*).

Para este problema específico, eu recomendaria no máximo uns **20 ou 25 neurônios** na camada oculta. Mas em situações reais, é sempre importante testar diferentes configurações por conta própria.

Por fim, temos a camada de saída, *saida_camada*, com **apenas 1 neurônio**. Isso acontece porque queremos um único valor como resultado: a soma de dois números.

Definindo o Modelo

Continuando no nosso código temos os seguintes hiperparâmetros:

```
Modelo=RedeSoma(input_camada,hidden_camada,saida_camada)

criterio=nn.MSELoss()

otimizador=optim.Adam(Modelo.parameters(), lr=0.1)
```

Quando criamos a variável `Modelo = RedeSoma(input_camada, hidden_camada, saida_camada)`, estamos finalmente **criando nosso modelo** com os hiperparâmetros que definimos anteriormente. Nesse caso, `RedeSoma` é a **arquitetura da rede**.

Logo abaixo, temos `criterio = nn.MSELoss()`, que define o **critério da rede**. Numa rede neural, o critério serve para medir **quanto a rede errou**. No nosso caso, `MSELoss()` (*Mean Squared Error*, ou erro quadrático médio) calcula a diferença entre o valor previsto pela rede e o valor desejado. A rede vai usar essa informação para fazer o **backpropagation**, que é, em termos simples, o processo em que a rede ajusta os pesos para aprender melhor e se aproximar do resultado correto. O `MSELoss` penaliza mais os erros maiores, ajudando a rede a corrigir as previsões mais distantes do esperado.

Por fim, temos o **otimizador da rede**, que no nosso caso é o Adam. O otimizador ajuda a rede a **aprender de maneira mais eficiente**.

Ao criar `otimizador = optim.Adam(Modelo.parameters(), lr=0.1)`, passamos dois parâmetros:

- `Modelo.parameters()`: diz ao otimizador **quais pesos ele pode ajustar**.
- `lr=0.1`: define o **learning rate**, ou seja, a velocidade com que a rede aprende. Se o valor for muito alto, a rede pode “aprender rápido demais”, pulando passos importantes do treinamento e tornando o modelo impreciso, com alto risco de “explosão da rede” (quando os pesos ficam desordenados e geram *Nan*s).

Por isso, é fundamental ter **cuidado com os hiperparâmetros**. Eles podem parecer pequenos detalhes, mas influenciam muito o desempenho da rede e o sucesso do treinamento. Inteligências artificiais são delicadas — qualquer confusão com esses valores pode gerar problemas difíceis de corrigir.

Treinamento da rede

Estamos na reta final do nosso código, e agora vamos falar sobre o **treinamento da nossa rede neural**.

Partindo da suposição de que você já tenha programado em Python, naturalmente lembrará das **estruturas de repetição**. Aqui, na nossa rede, iremos usar uma estrutura *for*, justamente porque sabemos a **quantidade de epochs** que queremos que a rede complete. Implementamos desta forma:

```
for epoch in range(561):
    saida_predit = Modelo(entradas)
    perda= criterio(saida_predit,saidas)
    otimizador.zero_grad()
    perda.backward()
    otimizador.step()
    if epoch % 100 == 0:
        print(f'Epoch:{epoch} | Perda: {perda.item():.4f}')
```

Agora vamos entender nosso loop de treinamento linha por linha.

Primeiro temos o seguinte comando:

```
saida_predit = Modelo(entradas)
```

Quando definimos a variável `saida_predit` e fazemos com que ela receba `Modelo(entradas)`, estamos **passando os dados de entrada pela rede a cada epoch**. Em termos técnicos, isso é chamado de **forward pass**.

Passando para próxima linha temos o seguinte comando:

```
perda= criterio(saida_predit,saidas)
```

Aqui, vamos calcular **o quanto a rede errou**. Para isso, o critério **compara a saída prevista pela rede com o valor desejado**, realiza o cálculo da distância entre a saída que a rede retornou e a saída correta, e **armazena o resultado na variável perda**.

Seguindo vamos ver o comando:

```
otimizador.zero_grad()
```

Aqui, estamos **zerando os gradientes antigos** para que eles não sejam somados com os novos gradientes. Vamos explicar o que é um gradiente:

O **gradiente** em redes neurais serve para medir a **intensidade e a direção dos ajustes necessários** em cada peso. Para ficar mais fácil de entender, imagine que você está no ponto mais alto de uma montanha e quer descer até o ponto mais baixo. Os ajustes a serem feitos são muitos, e você quer reduzir o número de erros. O gradiente mostra a **direção correta** e indica **quão íngreme** é o caminho.

Quando zeramos o gradiente, estamos basicamente **voltando ao topo da montanha** para buscar outro caminho de descida.

Logo abaixo vamos ver o comando:

```
perda.backward()
```

Como já explicamos no Módulo 2, **backpropagation** é quando a rede pega os erros do **forward pass** e verifica **como cada peso contribuiu para o erro**, ajustando em seguida para melhorar suas previsões.

O próximo comando no nosso *loop* de treinamento é:

```
otimizador.step()
```

Depois que o **backward pass** calcula os gradientes, é o **otimizador.step()** que **ajusta os pesos da rede**, com o objetivo de **diminuir o erro**.

Voltando ao exemplo da montanha: se o **gradiente** indica a direção e a inclinação do caminho, quem realmente **dá o passo nessa direção** é o **otimizador.step()**.

Por fim, a cada 100 *epochs* completadas, vamos **imprimir em qual época de treinamento o modelo está e qual foi a perda**, fazendo isso da seguinte forma:

```
if epoch % 100 == 0:  
    print(f'Epoch:{epoch} | Perda: {perda.item():.4f}')
```

Por fim vamos fazer uma previsão com nossa rede, e vamos fazer dessa maneira:

```
with torch.no_grad():

    pred= Modelo(entradas)

    print('Previsões:')

    print(pred)
```

Quando estamos treinando a rede, precisamos que ela calcule os gradientes, como já explicamos anteriormente. Entretanto, ao fazer uma previsão, não precisamos desses gradientes. Por isso, o `torch.no_grad()` interrompe temporariamente esse cálculo.

Dentro do bloco, passamos os dados de entrada pelo modelo e, em seguida, imprimimos na tela as previsões da rede.

Bônus

Caso você queira passar dados diferentes para o modelo, é igualmente simples, basta criar um tensor dentro do bloco do `torch.no_grad()` e passa-lo para o modelo, dá seguinte maneira:

```
with torch.no_grad():

    novos_dados = torch.tensor([[2.0, 4.0],
                               [4.0, 2.0]])

    pred= Modelo(novos_dados)

    print('Previsões:')

    print(pred)
```