



Building Neural Networks with PyTorch

Setup

Este projeto requer Python 3.10 ou superior:

```
In [1]: #!pip install -q pandas matplotlib seaborn scikit-Learn
```



Isso demora 9 minutos =/

```
In [2]: #!pip install torch>=2.8.0
```

```
In [3]: import sys  
assert sys.version_info >= (3, 10)
```

Também requer Scikit-Learn $\geq 1.6.1$:

```
In [4]: from packaging.version import Version  
import sklearn  
assert Version(sklearn.__version__) >= Version("1.6.1")
```

E é claro que precisamos do PyTorch, especificamente do PyTorch $\geq 2.8.0$:

```
In [5]: import torch  
  
assert Version(torch.__version__) >= Version("2.8.0")
```

Como fizemos nos capítulos anteriores, vamos definir os tamanhos de fonte padrão para deixar as figuras mais bonitas:

```
In [6]: import matplotlib.pyplot as plt  
  
plt.rc('font', size=14)  
plt.rc('axes', labelsize=14, titlesize=14)  
plt.rc('legend', fontsize=14)  
plt.rc('xtick', labelsize=10)  
plt.rc('ytick', labelsize=10)
```

Sobre o Pytorch

PyTorch é uma biblioteca open source de deep learning (originada do Torch/Lua) mantida pelo laboratório FAIR (Meta AI). Em Python, oferece **tensores**,

autodiferenciação (autograd), aceleração por GPU/TPU e um ecossistema de **camadas, perdas e otimizadores prontos**, tornando o desenvolvimento de redes neurais ágil e produtivo. Conceitualmente, lembra o NumPy, mas com suporte nativo a **cálculo em hardware acelerado e gradientes automáticos**.

"PôfêSô, por que Pt? eu gostava do TF!" Em 2016, o **TensorFlow** dominava o cenário por desempenho e escalabilidade, porém seu modelo de programação era mais **estático e complexo**. O PyTorch foi criado com uma proposta **mais "pythônica" e dinâmica**, usando **grafos computacionais dinâmicos (define-by-run)**, o que facilita a **depuração, a exploração interativa e a pesquisa**. Além do design limpo e documentação robusta, a comunidade open source cresceu rapidamente; em 2022, a governança migrou para a **PyTorch Foundation** (Linux Foundation), consolidando o ecossistema. O resultado prático foi a **adoção massiva na academia** e, por consequência, **migração gradual da indústria**.

O que você aprenderá neste capítulo

- **Fundamentos:** como trabalhar com **tensores** e **autograd** no PyTorch.
- **Primeiros passos de modelagem:** construir e treinar um **modelo de regressão linear** para entender o pipeline básico.
- **Evolução para redes profundas:** ampliar para **redes multicamadas (MLP)**:
 - **Regressão** com redes neurais.
 - **Classificação** com redes neurais.
- **Arquiteturas personalizadas:** criar modelos com **múltiplas entradas ou múltiplas saídas**.
- **Ajuste de hiperparâmetros:** usar **Optuna** para **tunar** modelos automaticamente.
- **Otimização e exportação:** técnicas para **otimizar desempenho e salvar/exportar** modelos para uso em produção.

Resumo da ideia central: PyTorch equilibra **simplicidade, flexibilidade e alto desempenho** graças aos **grafos dinâmicos** e ao **ecossistema maduro**. Este capítulo guia você do **básico (tensores/autograd)** à **produção (tuning, otimização e exportação)** passando por exemplos práticos de **regressão e classificação**.

1. Fundamentos do PyTorch

O **tensor** é a estrutura de dados central do PyTorch: um **array multidimensional** com **forma e tipo de dado**, usado para computação numérica. Ele é semelhante a um array do **NumPy**, mas tem duas vantagens fundamentais: pode **residir em GPU** (ou outros aceleradores) e **suporta auto-diferenciação**. A partir deste ponto, **todos os modelos**

trabalharão **recebendo e produzindo tensores**, de modo análogo a como modelos do Scikit-Learn usam arrays NumPy. O próximo passo é **criar e manipular tensores**.

1.1 PyTorch Tensors

Você pode criar um tensor PyTorch da mesma forma que criaria um array NumPy. Por exemplo, vamos criar um array 2×3 :

```
In [7]: import torch

X = torch.tensor([[1.0, 4.0, 7.0], [2.0, 3.0, 6.0]])
X
```

```
Out[7]: tensor([[1., 4., 7.],
               [2., 3., 6.]])
```

Assim como um array do NumPy, um **tensor** pode conter **floats, inteiros, booleanos** ou **números complexos** — **apenas um tipo por tensor**. Se você o inicializa com valores de tipos mistos, o PyTorch escolhe o **mais geral** segundo a hierarquia: **complexo > float > inteiro > bool**. Também é possível **definir o tipo explicitamente** na criação (por exemplo, `dtype=torch.float16` para floats de 16 bits). **Tensores de strings ou objetos não são suportados**.

Você pode **inspecionar a forma (shape) e o tipo (dtype)** de um tensor diretamente:

```
In [8]: X.shape
```

```
Out[8]: torch.Size([2, 3])
```

```
In [9]: X.dtype
```

```
Out[9]: torch.float32
```

A indexação funciona exatamente como para matrizes NumPy:

```
In [10]: X[0, 1]
```

```
Out[10]: tensor(4.)
```

```
In [11]: X[:, 1]
```

```
Out[11]: tensor([4., 3.])
```

Você pode aplicar **diversas operações numéricas** diretamente em tensores, com uma API **muito semelhante à do NumPy**: `torch.abs()`, `torch.cos()`, `torch.exp()`, `torch.max()`, `torch.mean()`, `torch.sqrt()`, entre outras. Quase todas também existem como **métodos do próprio tensor**, permitindo escrever `X.exp()` em vez de `torch.exp(X)`. O próximo trecho demonstra algumas dessas operações na prática.

```
In [12]: 10 * (X + 1.0) # item-wise addition and multiplication
```

```
Out[12]: tensor([[20., 50., 80.],
   [30., 40., 70.]])
```

```
In [13]: X.exp()
```

```
Out[13]: tensor([[ 2.7183,  54.5981, 1096.6332],
   [ 7.3891, 20.0855, 403.4288]])
```

```
In [14]: X.mean()
```

```
Out[14]: tensor(3.8333)
```

```
In [15]: X.max(dim=0)
```

```
Out[15]: torch.return_types.max(
    values=tensor([2., 4., 7.]),
    indices=tensor([1, 0, 0]))
```

```
In [16]: X @ X.T
```

```
Out[16]: tensor([[66., 56.],
   [56., 49.]])
```

Você também pode converter um tensor em uma matriz NumPy usando o método `numpy()` e criar um tensor a partir de uma matriz NumPy:

```
In [17]: import numpy as np
```

```
X.numpy()
```

```
Out[17]: array([[1., 4., 7.],
   [2., 3., 6.]], dtype=float32)
```

```
In [18]: torch.tensor(np.array([[1., 4., 7.], [2., 3., 6.]]))
```

```
Out[18]: tensor([[1., 4., 7.],
   [2., 3., 6.]], dtype=torch.float64)
```

No PyTorch, o **padrão de floats é 32 bits (float32)**, enquanto no NumPy é **64 bits (float64)**. Em deep learning, **float32 costuma ser preferível**: consome **metade da RAM, acelera os cálculos** e a rede **não precisa** da precisão extra de 64 bits. Ao converter um array NumPy com `torch.tensor()`, **indique** `dtype=torch.float32`. Como alternativa, `torch.FloatTensor()` **já converte** automaticamente para **32 bits**.

```
In [19]: torch.tensor(np.array([[1., 4., 7.], [2., 3., 6.]]), dtype=torch.float32)
```

```
Out[19]: tensor([[1., 4., 7.],
   [2., 3., 6.]])
```

```
In [20]: torch.FloatTensor(np.array([[1., 4., 7.], [2., 3., 6.]]))
```

```
Out[20]: tensor([[1., 4., 7.],
   [2., 3., 6.]])
```

```
In [21]: # extra code: demonstrate torch.from_numpy()
X2_np = np.array([[1., 4., 7.], [2., 3., 6.]])
X2 = torch.from_numpy(X2_np) # X2_np and X2 share the same data in memory
```

```
X2_np[0, 1] = 88
X2
```

```
Out[21]: tensor([[ 1., 88.,  7.],
   [ 2.,  3.,  6.]], dtype=torch.float64)
```

Você também pode modificar um tensor no local usando indexação e fatiamento, como com uma matriz NumPy:

```
In [22]: X[:, 1] = -99
X
```

```
Out[22]: tensor([[ 1., -99.,  7.],
   [ 2., -99.,  6.]])
```

A API do PyTorch oferece diversas operações **in-place** (terminadas com `_`), como `abs_()`, `sqrt_()` e `zero_()`, que **modificam o próprio tensor**. Elas podem **economizar memória e aumentar a velocidade** em alguns casos. Exemplo: `relu_()` aplica a **ReLU** diretamente, **substituindo valores negativos por 0** no mesmo tensor.

```
In [23]: X.relu_()
X
```

```
Out[23]: tensor([[1., 0., 7.],
   [2., 0., 6.]])
```

Os tensores do PyTorch realmente se assemelham a matrizes NumPy. Na verdade, eles têm mais de 200 funções comuns!

```
In [24]: # extra code: List functions that appear both in NumPy and PyTorch
functions = lambda mod: set(f for f in dir(mod) if callable(getattr(mod, f)))
", ".join(sorted(functions(torch) & functions(np)))
```

```
Out[24]: '__getattr__', 'abs', 'absolute', 'acos', 'acosh', 'add', 'all', 'allclose', 'amax', 'amin', 'angle', 'any', 'arange', 'arccos', 'arccosh', 'arcsin', 'arcsinh', 'arctan', 'arctan2', 'arctanh', 'argmax', 'argmin', 'argsort', 'argwhere', 'asarray', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'atleast_1d', 'atleast_2d', 'atleast_3d', 'bincount', 'bitwise_and', 'bitwise_left_shift', 'bitwise_not', 'bitwise_or', 'bitwise_right_shift', 'bitwise_xor', 'broadcast_shapes', 'broadcast_to', 'can_cast', 'ceil', 'clip', 'column_stack', 'concat', 'concatenate', 'conj', 'copysign', 'corrcoef', 'cos', 'cosh', 'count_nonzero', 'cov', 'cross', 'cumprod', 'cumsum', 'deg2rad', 'diag', 'diagflat', 'diagonal', 'diff', 'divide', 'dot', 'dsplit', 'dstack', 'dtype', 'einsum', 'empty', 'empty_like', 'equal', 'exp', 'exp2', 'expm1', 'eye', 'finfo', 'fix', 'flip', 'fliplr', 'flipud', 'float_power', 'floor', 'floor_divide', 'fmax', 'fmin', 'fmod', 'frexp', 'from_dlpack', 'frombuffer', 'full', 'full_like', 'gcd', 'gradient', 'greater', 'greater_equal', 'heaviside', 'histogram', 'histogramdd', 'hsplit', 'hstack', 'hypot', 'i0', 'iinfo', 'imag', 'inner', 'isclose', 'isfinite', 'isinf', 'isnan', 'isneginf', 'isposinf', 'isreal', 'kron', 'lcm', 'ldexp', 'less', 'less_equal', 'linspace', 'load', 'log', 'log10', 'log1p', 'log2', 'logaddexp', 'logaddexp2', 'logical_and', 'logical_not', 'logical_or', 'logical_xor', 'logspace', 'matmul', 'max', 'maximum', 'mean', 'median', 'meshgrid', 'min', 'minimum', 'moveaxis', 'multiply', 'nan_to_num', 'nanmean', 'nanmedian', 'nanquantile', 'nansum', 'negative', 'nextafter', 'nonzero', 'not_equal', 'ones', 'ones_like', 'outer', 'positive', 'pow', 'prod', 'promote_types', 'put', 'quantile', 'rad2deg', 'ravel', 'real', 'reciprocal', 'remainder', 'reshape', 'result_type', 'roll', 'rot90', 'round', 'row_stack', 'save', 'searchsorted', 'select', 'set_printoptions', 'sign', 'signbit', 'sin', 'sinc', 'sinh', 'sort', 'split', 'sqrt', 'square', 'squeeze', 'stack', 'std', 'subtract', 'sum', 'swapaxes', 'tak', 'tan', 'tanh', 'tensordot', 'tile', 'trace', 'transpose', 'trapezoid', 'trapz', 'tril', 'tril_indices', 'triu', 'triu_indices', 'true_divide', 'trunc', 'typename', 'unique', 'unravel_index', 'vander', 'var', 'vdot', 'vsplit', 'vstack', 'where', 'zeros', 'zeros_like'']'
```

1.2 Hardware Acceleration

Copiar tensores para a GPU é simples no PyTorch, desde que sua máquina tenha GPU compatível e as bibliotecas necessárias estejam instaladas. No **Google Colab**, basta usar um **runtime com GPU** (Menu *Runtime* → *Change runtime type* → selecionar uma GPU, como **Nvidia T4**). Esse ambiente já vem com o PyTorch compilado com suporte a GPU, drivers e bibliotecas requeridas (por exemplo, **CUDA** e **cuDNN**). Se preferir rodar localmente, instale **drivers** e **bibliotecas** apropriadas seguindo as instruções: <https://homl.info/install-p>.

O PyTorch tem excelente suporte a **GPUs Nvidia** e também a outros aceleradores:

- **Apple MPS**: aceleração em **Apple Silicon** (M1, M2, posteriores) e alguns **Intel Macs** compatíveis.
- **AMD**: **Instinct** e **Radeon** via **ROCM** (Linux) ou **DirectML** (Windows).
- **Intel**: **GPUs e CPUs** (Linux/Windows) via **oneAPI**.
- **Google TPUs**: integração via **torch_xla**.

```
In [25]: if torch.cuda.is_available():
    device = "cuda"
elif torch.backends.mps.is_available():
    device = "mps"
else:
    device = "cpu"

device
```

```
Out[25]: 'cpu'
```

Em um Colab GPU Runtime, o dispositivo será igual a "cuda". Agora, vamos criar um tensor nessa GPU. Para isso, uma opção é criar o tensor na CPU e copiá-lo para a GPU usando o método **to()**:

```
In [26]: M = torch.tensor([[1., 2., 3.], [4., 5., 6.]])
M = M.to(device)
M.device
```

```
Out[26]: device(type='cpu')
```

Alternativamente, podemos criar o tensor diretamente na GPU usando o argumento do dispositivo:

```
In [27]: M = torch.tensor([[1., 2., 3.], [4., 5., 6.]], device=device)
```

Quando o tensor estiver na GPU, podemos executar operações nele normalmente, e todas elas ocorrerão na GPU:

```
In [28]: R = M @ M.T # run some operations on the GPU
R
```

```
Out[28]: tensor([[14., 32.],
 [32., 77.]])
```

Quando um tensor é processado na **GPU**, o resultado também permanece na GPU. Isso permite **encadear várias operações** sem copiar dados entre CPU e GPU, evitando um gargalo comum de desempenho.

Quanto a GPU acelera? Depende do **modelo da GPU** (as mais caras podem ser **dezenas de vezes** mais rápidas) e do **throughput de dados**:

- **Modelos compute-heavy** (ex.: redes muito profundas): o **poder de cálculo** da GPU e a **quantidade de RAM** tendem a ser os fatores críticos.
- **Modelos rasos / datasets grandes**: o **envio contínuo de dados** para a GPU pode virar o **gargalo** principal.

O próximo trecho realiza um **teste comparando a multiplicação de matrizes na CPU vs GPU** para ilustrar essas diferenças de **tempo de execução e banda de dados**.

```
In [29]: M = torch.rand((1000, 1000)) # on the CPU
M @ M.T # warmup
%timeit M @ M.T

M = M.to(device)
M @ M.T # warmup
%timeit M @ M.T
```

3.28 ms ± 93.5 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
3.72 ms ± 120 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)

No teste, a **GPU (Nvidia T4 no Colab)** proporcionou um **ganho de ~26x** na multiplicação de matrizes. Com GPUs mais potentes, o **speedup** tende a ser ainda maior. Porém, **em matrizes pequenas** (ex.: 100×100), o ganho cai para algo como **~2x**. Isso ocorre porque **GPUs paralelizam tarefas grandes** (quebrando-as em muitas subtarefas para milhares de núcleos). **Tarefas pequenas** não geram paralelismo suficiente, reduzindo o benefício — e, em cenários com **muitas tarefas minúsculas**, a **CPU pode ser até mais rápida** devido ao overhead de orquestração na GPU.

Com a base de **tensores e execução em CPU/GPU** estabelecida, o próximo passo é explorar o **autograd** do PyTorch (auto-diferenciação).

1.3 Autograd

PyTorch implementa **auto-diferenciação em modo reverso** (*autograd*), permitindo calcular **gradientes automaticamente**. A ideia é simples de usar: dado, por exemplo, $f(x) = x^2$, o cálculo diferencial diz que $f'(x) = 2x$. Avaliando em $f(5)$, obtemos $f(5) = 25$ e $f'(5) = 10$. O próximo trecho verifica esses valores com o **autograd** do PyTorch.

```
In [30]: x = torch.tensor(5.0, requires_grad=True)
f = x ** 2
f
```

```
Out[30]: tensor(25., grad_fn=<PowBackward0>)
```

```
In [31]: f.backward()
x.grad
```

```
Out[31]: tensor(10.)
```

Obtivemos os valores corretos: $f = 25$ e ($x.\text{grad}=10$). A chamada `backward()` calculou automaticamente $f'(x)$ no ponto ($x=5.0$). Eis a lógica, linha a linha:

- **Definição de variável com gradiente** Criamos $x = 5.0$ com `requires_grad=True`. Assim, o PyTorch **rastreia todas as operações** envolvendo x , construindo o **grafo de computação** necessário para executar o **backpropagation**. Nesse grafo, x é um **nó-folha**.
- **Cálculo de $f = x * x^2$** O resultado é 25.0 . Além do valor, f carrega o atributo `grad_fn` (ex.: `PowBackward0`), que **representa a operação** que gerou f e **informa como retropropagar** os gradientes por essa operação. É assim que o PyTorch **mantém o grafo**.
- **Retropropagação** `f.backward()` **propaga gradientes** a partir de f **até os nós-folha** (aqui, apenas x).
- **Leitura do gradiente** `x.grad` contém a **derivada de f em relação a (x)**, computada no backprop (no exemplo, **10**).

Grafos dinâmicos (define-by-run)

O PyTorch **cria o grafo on-the-fly** a cada *forward pass*, conforme as operações são executadas, o que suporta **modelos dinâmicos** com **loops** e **condicionais**.

Passo de qradiente (gradient descent) com `torch.no_grad()`

Após obter gradientes, normalmente fazemos **descida do gradiente**, subtraindo uma fração do gradiente das variáveis do modelo. No exemplo de $f(x) = x^2$, isso **empurra (x) em direção a 0** (o minimizador).

Importante: **desative o rastreamento de gradiente** durante a atualização (por exemplo, usando `with torch.no_grad():`), pois **não queremos registrar** a própria atualização no grafo — e operações *in-place* em variáveis rastreadas podem **levantar exceção**.

```
In [32]: learning_rate = 0.1
with torch.no_grad():
    x -= learning_rate * x.grad # gradient descent step
```

```
In [33]: x
```

Out[33]: tensor(4., requires_grad=True)

No passo de **descida do gradiente**, (x) é decrementado por $0,1 \times 10,0 = 1,0$, indo de **5,0 para 4,0**.

Outra forma de **evitar cálculo de gradientes** é usar `detach()`: ele cria um **novo tensor desacoplado do grafo** (`requires_grad=False`), **apontando para os mesmos dados em memória**. Assim, você pode **atualizar esse tensor desacoplado** sem registrar a operação no grafo de computação.

In [34]:

```
x_detached = x.detach()
x_detached -= learning_rate * x.grad
```

Como x_{detached} e x **compartilham a mesma memória**, alterar x_{detached} também **altera** x .

- `detach()`: útil quando você precisa **executar computações sem afetar os gradientes** (ex.: avaliação, logging) ou quando deseja **controle fino** sobre **quais operações** entram no cálculo de gradientes. Cria um **tensor desacoplado do grafo** (`requires_grad=False`), apontando para os **mesmos dados**.
- `no_grad()`: geralmente **preferido** para **inferência** ou durante o **passo de descida do gradiente**, pois fornece um **contexto** conveniente que **desativa o rastreamento** de gradientes para tudo que estiver dentro do bloco.

Antes de repetir o ciclo ($\text{forward} \rightarrow \text{backward} \rightarrow \text{atualização}$), é **essencial zerar os gradientes de todos os parâmetros** do modelo, pois o PyTorch **acumula** gradientes por padrão. Não é necessário `no_grad()` para isso, já que os tensores de gradiente têm `requires_grad=False`.

In [35]:

```
x.grad.zero_()
```

Out[35]: tensor(0.)

Juntando tudo, o ciclo de treinamento fica assim:

In [36]:

```
learning_rate = 0.1
x = torch.tensor(5.0, requires_grad=True)
for iteration in range(100):
    f = x ** 2 # forward pass
    f.backward() # backward pass
    with torch.no_grad():
        x -= learning_rate * x.grad # gradient descent step
    x.grad.zero_() # reset the gradients
```

In [37]:

Out[37]: tensor(1.0185e-09, requires_grad=True)

Operações **in-place** podem **economizar memória** e **evitar cópias**, mas **nem sempre** combinam bem com o **autograd**:

1. Proibição em nós-folha Você **não pode** aplicar uma operação *in-place* em um **nó-folha** (tensor com `requires_grad=True`). O PyTorch não saberia **onde armazenar** as informações do grafo. Exemplos que geram `RuntimeError`:

- `x.cos_()`
- `x += 1`

2. Risco de sobrescrever valores necessários ao backward Mesmo fora de nós-folha, operações *in-place* podem **apagar valores intermediários** que o autograd precisa para calcular gradientes, quebrando o **backpropagation**.

Exemplo a seguir no código: calcular $z(t) = \exp(t) + 1$ em $t = 2$ e tentar obter os gradientes — veremos como certas escolhas *in-place* podem levar a erros durante o `backward()`.

In [38]: #Novo

```
"""t = torch.tensor(2.0, requires_grad=True)
z = t.exp() # this is an intermediate result
z += 1 # this is an in-place operation
z.backward() # ! RuntimeError!"""

Out[38]: 't = torch.tensor(2.0, requires_grad=True)\nz = t.exp() # this is an intermediate result\nz += 1 # this is an in-place operation\nz.backward() # ! RuntimeError!'
```

O valor de z foi calculado corretamente, mas o `backward()` lançou `RuntimeError: one of the variables needed for gradient computation has been modified by an in-place operation`. Isso ocorreu porque o resultado intermediário $z = t.\exp()$ foi **sobrescrito** por $z += 1$. No *backward*, ao chegar na `\exp`, o autograd já não tinha o valor necessário para calcular o gradiente.

Correção simples: troque $z += 1$ por $z = z + 1$. Visualmente é parecido, mas **não é in-place**: cria-se **um novo tensor**, preservando o original no **grafo de computação**.

Por que às vezes funciona? (ex.: com `cos()`)

Se você trocar `\exp()` por `\cos()` no exemplo, o gradiente **funciona**. O motivo é **como cada operação é implementada e o que ela guarda** para o *backward*:

- **Operações que salvam a *saída*** (não modifique a **saída in-place antes do backward):** `\exp()`, `\relu()`, `\rsqrt()`, `\sigmoid()`, `\sqrt()`, `\tan()`, `\tanh()`.
- **Operações que salvam a *entrada*** (podem tolerar modificar a **saída, mas não modifique a entrada in-place antes do backward):** `\abs()`, `\cos()`, `\log()`, `\sin()`, `\square()`, `\var()`.
- **Operações que salvam *entrada e saída*** (não modifique **nenhuma delas in-place):** `\max()`, `\min()`, `\norm()`, `\prod()`, `\sgn()`, `\std()`.

- **Operações que não salvam nem entrada nem saída** (seguro modificar *in-place*):

```
ceil(), floor(), mean(), round(), sum().
```

Regra prática: quando usar operações *in-place* para economizar memória, verifique **o que a operação salva** para o *backward*. Se você alterar *in-place* algo que o autograd precisa (entrada, saída ou ambos), o gradiente **quebrará**.

OK, vamos voltar um pouco. Já discutimos todos os fundamentos do PyTorch: como criar tensores e usá-los para realizar todos os tipos de cálculos, como acelerar os cálculos com uma GPU e como usar o Autograd para calcular gradientes para gradiente descendente. Ótimo! Agora, vamos aplicar o que aprendemos até agora, construindo e treinando um modelo de regressão linear simples com o PyTorch.

2. Implementing Linear Regression

Começaremos implementando a regressão linear usando tensores e autograd diretamente, depois simplificaremos o código usando a API de alto nível do PyTorch e também adicionaremos suporte à GPU.

2.1 Linear Regression Using Tensors & Autograd

Usaremos o **California Housing** (como no Cap. 9). Suponha que os dados já foram baixados com `sklearn.datasets.fetch_california_housing()` e divididos com `train_test_split()` em: `X_train`, `y_train`, `X_valid`, `y_valid`, `X_test`, `y_test`.

Agora vamos **converter tudo para tensores** e **normalizar** usando **operações de tensor** (em vez de `StandardScaler`), para praticar PyTorch:

- **Estatísticas no treino:** calcule **média** μ_{train} e **desvio-padrão** σ_{train} apenas em `X_train`.
- **Padronização:** aplique em todos os *splits* a transformação

$$\tilde{X} = \frac{X - \mu_{\text{train}}}{\sigma_{\text{train}}}$$

Isso garante padronização consistente e evita **vazamento de informação**. O próximo trecho mostra a **conversão para tensores** e a **normalização** passo a passo.

```
In [39]: from sklearn.datasets import fetch_california_housing
from sklearn.model_selection import train_test_split

housing = fetch_california_housing()
X_train_full, X_test, y_train_full, y_test = train_test_split(
    housing.data, housing.target, random_state=42)
X_train, X_valid, y_train, y_valid = train_test_split(
    X_train_full, y_train_full, random_state=42)
```

```
In [40]: X_train = torch.FloatTensor(X_train)
X_valid = torch.FloatTensor(X_valid)
X_test = torch.FloatTensor(X_test)
means = X_train.mean(dim=0, keepdims=True)
stds = X_train.std(dim=0, keepdims=True)
X_train = (X_train - means) / stds
X_valid = (X_valid - means) / stds
X_test = (X_test - means) / stds
```

Também precisamos **converter os alvos para tensores**. Como as **preedições** serão **vetores-coluna** (matrizes com **uma única coluna**), os **alvos** devem ter o **mesmo formato**. Porém, no NumPy os alvos vêm **unidimensionais**; portanto, é necessário **reestruturar** para **adicionar uma 2ª dimensão de tamanho 1**, formando vetores-coluna (isto é, shape $[N, , 1]$).

Em resumo: converta `y_*` para tensores e **reshape** para $[N, , 1]$ para alinhar com as **saídas do modelo**.

```
In [41]: y_train = torch.FloatTensor(y_train).view(-1, 1)
y_valid = torch.FloatTensor(y_valid).view(-1, 1)
y_test = torch.FloatTensor(y_test).view(-1, 1)
```

Agora que os dados estão prontos, vamos criar os parâmetros do nosso modelo de regressão linear:

```
In [42]: torch.manual_seed(42)
n_features = X_train.shape[1] # there are 8 input features
w = torch.randn((n_features, 1), requires_grad=True)
b = torch.tensor(0., requires_grad=True)
```

Definimos um vetor de **pesos** w (vetor-coluna com **um peso por atributo de entrada**, aqui 8) e um **viés** escalar b . Os **pesos são inicializados aleatoriamente** e o **viés em zero**. Embora, neste caso simples, também pudéssemos zerar w , é boa prática **inicializar pesos aleatoriamente** para **quebrar a simetria** entre unidades — princípio crucial em **redes neurais** (Cap. 9). Adotar esse hábito desde já facilita a transição para modelos mais profundos.

Em seguida, vamos treinar nosso modelo, de forma muito semelhante à que fizemos no Capítulo 4, exceto que usaremos o autodiff para calcular os gradientes em vez de usar uma equação de forma fechada. Por enquanto, usaremos a descida do gradiente em lote (BGD), utilizando o conjunto de treinamento completo em cada etapa:

Laço de treino: passo a passo (regressão linear)

- **Taxa de aprendizado (learning_rate)** Definimos o hiperparâmetro de **taxa de aprendizado**; experimente valores para equilibrar **convergência** e **precisão**.
- **Épocas (20)** Executamos **20 épocas**. Poderíamos aplicar **early stopping** (Cap. 4), mas aqui mantemos simples.

- **Forward pass** Calculamos as **previsões** y_{pred} e a **loss MSE**:

$$\text{MSE}(y, \hat{y}) = \frac{1}{N} \sum_{i=1}^N (y_i - \hat{y}_i)^2.$$

- **Autograd** `loss.backward()` calcula **gradientes** da loss em relação a **todos os parâmetros**.
- **Passo de descida do gradiente** Usamos `b.grad` e `w.grad` para **atualizar** os parâmetros **dentro de** `with torch.no_grad():`.
- **Zerar gradientes (essencial!)** Após atualizar, **zeramos os gradientes** para não acumulá-los na próxima iteração.
- **Logging** Imprimimos **nº da época** e a **loss**; `item()` extrai o valor escalar.

Regra prática de formatação: **código** → `backticks` ; **fórmulas** → $\$... \$$; se precisar de sublinhado em modo math, use `_`.

In [43]:

```
learning_rate = 0.4
n_epochs = 20
for epoch in range(n_epochs):
    y_pred = X_train @ w + b
    loss = ((y_pred - y_train) ** 2).mean()
    loss.backward()
    with torch.no_grad():
        b -= learning_rate * b.grad
        w -= learning_rate * w.grad
        b.grad.zero_()
        w.grad.zero_()
    print(f"Epoch {epoch + 1}/{n_epochs}, Loss: {loss.item()}")
```

```
Epoch 1/20, Loss: 16.158456802368164
Epoch 2/20, Loss: 4.87936544418335
Epoch 3/20, Loss: 2.2552266120910645
Epoch 4/20, Loss: 1.3307629823684692
Epoch 5/20, Loss: 0.968069851398468
Epoch 6/20, Loss: 0.8142682313919067
Epoch 7/20, Loss: 0.7417048811912537
Epoch 8/20, Loss: 0.7020704746246338
Epoch 9/20, Loss: 0.6765921711921692
Epoch 10/20, Loss: 0.6577967405319214
Epoch 11/20, Loss: 0.6426153779029846
Epoch 12/20, Loss: 0.6297224760055542
Epoch 13/20, Loss: 0.6184943914413452
Epoch 14/20, Loss: 0.6085969805717468
Epoch 15/20, Loss: 0.5998218655586243
Epoch 16/20, Loss: 0.5920187830924988
Epoch 17/20, Loss: 0.5850692391395569
Epoch 18/20, Loss: 0.5788734555244446
Epoch 19/20, Loss: 0.5733454823493958
Epoch 20/20, Loss: 0.5684100985527039
```

Parabéns, você acabou de treinar seu primeiro modelo usando o PyTorch! Agora você pode usar o modelo para fazer previsões para alguns novos dados X_{new} (que devem

ser representados como um tensor do PyTorch). Por exemplo, vamos fazer previsões para as três primeiras instâncias do conjunto de teste:

```
In [44]: X_new = X_test[:3] # pretend these are new instances
with torch.no_grad():
    y_pred = X_new @ w + b # use the trained parameters to make predictions
```

```
In [45]: y_pred
```

```
Out[45]: tensor([[0.8916],
                  [1.6480],
                  [2.6577]])
```

Implementar a regressão linear usando a API de baixo nível do PyTorch não foi tão difícil, mas usar essa abordagem para modelos mais complexos seria muito complicado e trabalhoso. Portanto, o PyTorch oferece uma API de alto nível para simplificar tudo isso. Vamos reescrever nosso modelo usando essa API de alto nível.

2.2 Linear Regression Using PyTorch's High-Level API

PyTorch fornece uma implementação de regressão linear na classe `torch.nn.Linear`, então vamos usá-la

nn = *neural network(s)* ("rede(s) neural(is)").

```
In [46]: import torch.nn as nn

torch.manual_seed(42) # to get reproducible results
model = nn.Linear(in_features=n_features, out_features=1)
```

`nn.Linear` (atalho de `torch.nn.Linear`) é um **módulo** do PyTorch; todo módulo herda de `nn.Module`. Para **regressão linear simples, um único** `nn.Linear` já basta. Em redes neurais reais, você **empilha vários módulos** — pense neles como "**LEGOs matemáticos**".

Parâmetros internos do `nn.Linear`:

- **Viés:** vetor de **bias** com **um termo por neurônio**.
- **Pesos:** **matriz de pesos** com **uma linha por neurônio** e **uma coluna por dimensão de entrada** (ou seja, é a **transposta** da matriz/vetor de pesos usada anteriormente no capítulo e na equação abaixo).

No nosso caso, com `out_features=1`, o modelo tem **um único neurônio**:

- O vetor de **bias** tem **um único termo**.
- A **matriz de pesos** tem **uma única linha**.

Esses parâmetros ficam acessíveis **diretamente** como **atributos** do módulo:

- `linear.weight` → matriz de pesos W

- `linear.bias` → viés b

Intuição: o módulo implementa $y = XW^\top + b$, (com W armazenado como **linha** quando há um neurônio de saída), mantendo os parâmetros **dentro do módulo** para integração com o **autograd** e os **otimizadores**.



In [47]: `model.bias`

Out[47]: Parameter containing:
`tensor([0.3117], requires_grad=True)`

In [48]: `model.weight`

Out[48]: Parameter containing:
`tensor([[0.2703, 0.2935, -0.0828, 0.3248, -0.0775, 0.0713, -0.1721, 0.2076]], requires_grad=True)`

Parâmetros em `nn.Linear`: inicialização, `Parameter` e iteração

- **Inicialização aleatória + reproduzibilidade** Os **dois parâmetros** do módulo (`weight` e `bias`) são **inicializados aleatoriamente** por padrão — por isso utilizamos `manual_seed()` para obter **resultados reproduzíveis**.
- **`torch.nn.Parameter` vs `tensor comum`** Esses parâmetros são instâncias de `torch.nn.Parameter`, que **herda** de `tensor.Tensor`. ➤ Consequência: você pode **usá-los como tensores normais** (mesmas operações), **mas** o PyTorch os **registra automaticamente** como **parâmetros treináveis** do módulo (para autograd e otimizadores). **Diferença central:** um `Parameter` é um **tensor marcado** para ser **rastreado como parâmetro do modelo**; já um **tensor comum**, mesmo com `requires_grad=True`, **não** é listado como parâmetro do módulo.
- **`module.parameters()` (recursivo)** O método `parameters()` retorna um **iterador** sobre **todos os atributos do tipo `Parameter`** do módulo **e de seus submódulos (recursivamente)**. Não retorna **tensores comuns**, mesmo que tenham `requires_grad=True`.

Em resumo: **Parameter = tensor treinável do módulo** (aparece em `parameters()` e nos otimizadores); **tensor comum** pode participar do grafo, mas **não** é tratado como **parâmetro do modelo**.

In [49]: `for param in model.parameters():
print(param)`

```
Parameter containing:
tensor([[ 0.2703,  0.2935, -0.0828,  0.3248, -0.0775,  0.0713, -0.1721,  0.207
6]],

Parameter containing:
tensor([0.3117], requires_grad=True)
```

- **Parâmetros nomeados** Além de `parameters()`, há `named_parameters()`, que retorna um **iterador de pares (nome, parâmetro)**. Útil para **inspeção, debug e logs** (ex.: imprimir gradientes por nome).
- **Chamando o módulo como função** Um **módulo** (`nn.Module`) pode ser **chamado como função**: internamente ele executa o método `forward`. Exemplo: passar as **duas primeiras instâncias** do *training set* ao modelo para obter **predições** iniciais.

Como os **parâmetros ainda estão aleatórios**, as **predições serão ruins** — o comportamento esperado **antes do treino**.

```
In [50]: model(X_train[:2])
```

```
Out[50]: tensor([[-0.4718],
 [ 0.1131]], grad_fn=<AddmmBackward0>)
```

Ao usar um **módulo como função**, o PyTorch chama internamente seu `forward()`. No caso de `nn.Linear`, ele computa: $[X; @; W^T \top; +; b]$ onde X é a entrada, W é `self.weight` e b é `self.bias` — **exatamente** o que precisamos para **regressão linear**.

O tensor de saída traz o atributo `grad_fn`, indicando que o **autograd** rastreou o **grafo de computação** durante as predições.

Próximo passo: com o modelo definido, precisamos criar um **otimizador** (para **atualizar os parâmetros**) e escolher uma **função de perda** (loss).

```
In [51]: optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)
mse = nn.MSELoss()
```

O PyTorch oferece **diversos otimizadores** (detalhes no próximo capítulo). Aqui usamos o **SGD (stochastic gradient descent)**, que atende a **SGD puro, mini-batch GD e batch GD**. Para inicializá-lo, passamos os **parâmetros do modelo** (por exemplo, `model.parameters()`) e a **taxa de aprendizado** (`lr`).

Para a **função de perda**, instanciamos `nn.MSELoss`. Ela também é um **módulo**, então pode ser usada **como função**: recebe **predições** e **alvos** e calcula a **MSE**,

$$\text{MSE}(y, \hat{y}) = \frac{1}{N} \sum_{i=1}^N (y_i - \hat{y}_i)^2.$$

O submódulo `nn` inclui **muitas outras perdas** e utilidades de redes neurais.

A seguir: escreveremos uma **função de treino** compacta para executar o ciclo *forward* → *loss* → *backward* → *update*.

```
In [52]: def train_bgd(model, optimizer, criterion, X_train, y_train, n_epochs):
    for epoch in range(n_epochs):
        y_pred = model(X_train)
        loss = criterion(y_pred, y_train)
        loss.backward()
        optimizer.step()
        optimizer.zero_grad()
        print(f"Epoch {epoch + 1}/{n_epochs}, Loss: {loss.item()}")
```

Comparando os laços de treino: baixo nível vs. alto nível (nn + optim)

O novo laço é **quase idêntico** ao anterior, porém agora usamos **abstrações de nível mais alto** (módulos `nn` e otimizadores `optim`) em vez de manipular diretamente **tensores e autograd**. Pontos-chave:

- **Critério (loss function “objeto”)** Em PyTorch, é comum chamar o **objeto** de função de perda de **criterion** (para distinguir do **valor da loss** calculado a cada iteração). Aqui, o **criterion** é uma instância de `nn.MSELoss`.
- **optimizer.step()** Substitui as duas linhas manuais que **atualizavam** *b* e *w* no código anterior (descida do gradiente).
- **optimizer.zero_grad()** Substitui as duas linhas que **zeravam** *b.grad* e *w.grad*. Observação: **não é necessário** usar `with torch.no_grad():` aqui — o **otimizador** já trata corretamente o **rastro de gradientes** internamente em `step()` e `zero_grad()`.

Próximo passo: chamar a **função de treino** para ajustar o modelo e observar a **queda da loss** ao longo das épocas.

```
In [53]: train_bgd(model, optimizer, mse, X_train, y_train, n_epochs)
```

```
Epoch 1/20, Loss: 4.3378496170043945
Epoch 2/20, Loss: 0.780293345451355
Epoch 3/20, Loss: 0.6253840327262878
Epoch 4/20, Loss: 0.6060433387756348
Epoch 5/20, Loss: 0.5956299304962158
Epoch 6/20, Loss: 0.5873566269874573
Epoch 7/20, Loss: 0.5802990198135376
Epoch 8/20, Loss: 0.5741382241249084
Epoch 9/20, Loss: 0.5687100887298584
Epoch 10/20, Loss: 0.5639079213142395
Epoch 11/20, Loss: 0.5596510767936707
Epoch 12/20, Loss: 0.5558737516403198
Epoch 13/20, Loss: 0.5525193810462952
Epoch 14/20, Loss: 0.5495391488075256
Epoch 15/20, Loss: 0.5468899011611938
Epoch 16/20, Loss: 0.544533908367157
Epoch 17/20, Loss: 0.5424376130104065
Epoch 18/20, Loss: 0.5405716300010681
Epoch 19/20, Loss: 0.5389096736907959
Epoch 20/20, Loss: 0.5374288558959961
```

Tudo bem; o modelo foi treinado e agora você pode usá-lo para fazer previsões simplesmente chamando-o como uma função (de preferência dentro de um contexto `no_grad()`, como vimos anteriormente):

```
In [54]: X_new = X_test[:3] # pretend these are new instances
with torch.no_grad():
    y_pred = model(X_new) # use the trained model to make predictions

y_pred
```

```
Out[54]: tensor([[0.8061],
                  [1.7116],
                  [2.6973]])
```

As **previsões** ficam **semelhantes**, mas **não idênticas** às do modelo anterior porque o `nn.Linear` **inicializa os parâmetros de forma diferente**: usa uma **distribuição uniforme** em um **intervalo específico** tanto para os **pesos** quanto para o **viés** (detalhes sobre inicialização serão tratados no **Cap. 11**).

Com essa **API de alto nível** dominada, estamos prontos para ir além da **regressão linear** e construir um **perceptron multicamadas (MLP)**.

Aula 34

3. Implementing a Regression MLP

`nn.Sequential` no PyTorch

O PyTorch oferece o módulo `nn.Sequential`, que permite encadear múltiplos módulos de forma automática:

- Ao passar uma entrada para um módulo `nn.Sequential`, ela é processada pelo **primeiro módulo**.
- A saída do primeiro módulo é então enviada para o **segundo módulo**, e assim por diante até o último módulo.

Esse recurso é especialmente útil porque muitas redes neurais são essencialmente **pilhas de módulos**, tornando o `nn.Sequential` um dos módulos mais importantes do PyTorch.

No caso de um **MLP (Perceptron Multicamadas)**, ele pode ser construído como uma simples sequência de módulos: duas camadas ocultas seguidas de uma camada de saída.

In [55]:

```
torch.manual_seed(42)
model = nn.Sequential(
    nn.Linear(n_features, 50),
    nn.ReLU(),
    nn.Linear(50, 40),
    nn.ReLU(),
    nn.Linear(40, 4),
    nn.Softmax()
)
```

- **Camada 1 (entrada → oculta 1)**
 - **Entradas:** n_{features} (aqui, 8).
 - **Saídas:** 50 (hiperparâmetro ajustável).
- **Ativação 1 — `nn.ReLU`**
 - **Sem parâmetros;** atua **elemento a elemento**; **não** altera o *shape* (apenas os valores).
- **Camada 2 (oculta 1 → oculta 2)**
 - **Entradas:** 50 (deve **casar** com a saída da camada anterior).
 - **Saídas:** 40 (poderia ser igual à anterior, mas aqui usamos 40 para enfatizar a **compatibilidade entre camadas**).
- **Ativação 2 — `nn.ReLU`**
 - Mesmas observações: **sem parâmetros, itemwise, shape** preservado.
- **Camada de saída (oculta 2 → saída)**
 - **Entradas:** 40 (devem **casar** com a camada anterior).
 - **Saídas:** devem **igualar a dimensionalidade dos alvos**. Como o alvo é **unidimensional**, usamos 1.

Regra prática: a **saída de uma camada** deve **combinar** com a **entrada da próxima**; apenas a **última camada** é **restrita pelos alvos**. A seguir: **treinar o modelo** exatamente como antes (definir *criterion*, *optimizer*, laço de épocas).

```
In [56]: learning_rate = 0.1
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)
mse = nn.MSELoss()
train_bgd(model, optimizer, mse, X_train, y_train, n_epochs)
```

```
Epoch 1/20, Loss: 4.697454929351807
Epoch 2/20, Loss: 4.697449684143066
Epoch 3/20, Loss: 4.697444438934326
Epoch 4/20, Loss: 4.697439670562744
Epoch 5/20, Loss: 4.697434425354004
Epoch 6/20, Loss: 4.697429656982422
Epoch 7/20, Loss: 4.69742488861084
Epoch 8/20, Loss: 4.697420120239258
Epoch 9/20, Loss: 4.697415351867676
Epoch 10/20, Loss: 4.697410583496094
Epoch 11/20, Loss: 4.697405815124512
Epoch 12/20, Loss: 4.69740104675293
Epoch 13/20, Loss: 4.697396755218506
Epoch 14/20, Loss: 4.697392463684082
Epoch 15/20, Loss: 4.6973876953125
Epoch 16/20, Loss: 4.697383880615234
Epoch 17/20, Loss: 4.6973795890808105
Epoch 18/20, Loss: 4.697375774383545
Epoch 19/20, Loss: 4.697371482849121
Epoch 20/20, Loss: 4.697366714477539
```

```
c:\Users\estev\OneDrive\Área de Trabalho\NCIA\.ncia\Lib\site-packages\torch\nn\modules\module.py:1775: UserWarning: Implicit dimension choice for softmax has been deprecated. Change the call to include dim=X as an argument.
    return self._call_impl(*args, **kwargs)
c:\Users\estev\OneDrive\Área de Trabalho\NCIA\.ncia\Lib\site-packages\torch\nn\modules\loss.py:634: UserWarning: Using a target size (torch.Size([11610, 1])) that is different to the input size (torch.Size([11610, 4])). This will likely lead to incorrect results due to broadcasting. Please ensure they have the same size.
    return F.mse_loss(input, target, reduction=self.reduction)
```

Parabéns: você **treinou sua primeira rede neural no PyTorch!** 🎉 Mas ainda estamos usando **batch gradient descent**: a cada iteração, calculamos **gradientes sobre todo o conjunto de treino**. Isso funciona para **datasets pequenos**, porém **não escala bem para dados/modelos grandes**.

Próximo passo: migrar para **mini-batch gradient descent**, isto é, **particionar o treinamento em lotes menores** (mini-batches). Benefícios principais:

- **Eficiência computacional**: melhor uso de **GPU/CPU e memória**.
- **Atualizações mais frequentes**: gradientes **mais ruidosos**, porém **mais rápidos**, acelerando a convergência prática.
- **Escalabilidade**: viabiliza treinar com **datasets massivos e modelos maiores**.

Ideia central: trocamos “um gradiente caro por época” por “**vários gradientes baratos por época**”, mantendo bom progresso e reduzindo o custo por iteração.

4. Implementing Mini-Batch Gradient Descent using DataLoaders

Carregando Mini-Batches com DataLoader

Para facilitar a implementação do **Gradient Descent em mini-batches**, o PyTorch oferece a classe `DataLoader`, disponível em `torch.utils.data`.

Ela permite:

- Carregar os dados em **lotes (batches)** de forma eficiente;
- **Embaralhar** os dados a cada época, quando desejado.

O `DataLoader` precisa receber um **dataset** que possua duas funções:

- `__len__()` → retorna o número de amostras;
- `__getitem__(index)` → retorna uma amostra específica (entrada + alvo).

Como nossos dados já estão nos tensores `X_train` e `y_train`, podemos usar a classe `TensorDataset`, que encapsula os tensores e fornece a interface necessária.

Após isso, utilizamos o `DataLoader` para extrair os mini-batches do dataset, incluindo a opção `shuffle=True` para embaralhar as amostras durante o treinamento.

```
In [57]: from torch.utils.data import TensorDataset, DataLoader
train_dataset = TensorDataset(X_train, y_train)
train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)
```

Com um modelo maior e o treinamento sendo feito em mini-batches, é recomendável utilizar **aceleração por hardware** (GPU).

A abordagem é direta:

- O modelo deve ser **movido para a GPU**, usando o método `.to()`, o que transfere todos os seus parâmetros para a memória da GPU.
- A cada iteração de treinamento, o **batch também deve ser copiado para a GPU** antes do processamento.

Além disso, após mover o modelo, podemos definir a **função de perda** e o **otimizador** normalmente, porém com um **learning rate menor** (por exemplo, `0.02`), já que modelos maiores exigem ajustes mais cuidadosos.

```
In [58]: # extra code - build the model just like earlier
torch.manual_seed(42)
model = nn.Sequential(
    nn.Linear(n_features, 50),
    nn.ReLU(),
    nn.Linear(50, 40),
    nn.ReLU(),
    nn.Linear(40, 1))
```

```
)  
  
model = model.to(device)  
  
# extra code - build the optimizer and loss function, as earlier  
learning_rate = 0.02  
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate, momentum=0)  
mse = nn.MSELoss()
```

Para treinar o modelo utilizando **mini-batch Gradient Descent**, definimos uma função `train()` responsável por:

- Iterar sobre os batches fornecidos pelo `DataLoader`;
- Enviar os dados para a GPU antes do processamento;
- Executar as etapas do treinamento em cada batch:
 - Forward pass
 - Cálculo da perda
 - Backpropagation
 - Atualização dos parâmetros

Essa função organiza o processo de treinamento de forma clara e reutilizável, mantendo o fluxo eficiente com uso de mini-batches.

```
In [59]: def train(model, optimizer, criterion, train_loader, n_epochs):  
    model.train()  
    for epoch in range(n_epochs):  
        total_loss = 0.  
        for X_batch, y_batch in train_loader:  
            X_batch, y_batch = X_batch.to(device), y_batch.to(device)  
            y_pred = model(X_batch)  
            loss = criterion(y_pred, y_batch)  
            total_loss += loss.item()  
            loss.backward()  
            optimizer.step()  
            optimizer.zero_grad()  
  
        mean_loss = total_loss / len(train_loader)  
        print(f"Epoch {epoch + 1}/{n_epochs}, Loss: {mean_loss:.4f}")
```

Durante o treinamento, a função percorre todo o conjunto de treinamento **época por época**, processando cada batch de forma sequencial, como já visto.

A primeira linha da função é:

- `model.train()`
 - Coloca o modelo (e todos os seus submódulos) em **modo de treinamento**.
 - Por enquanto, isso não muda nada no comportamento das camadas que estamos usando.
 - Porém, será essencial quando utilizarmos camadas como:
 - `nn.Dropout`
 - `nn.BatchNorm1d`
 - Essas camadas **se comportam de maneira diferente** no treinamento e na avaliação.

Para usar o modelo fora do treinamento, como em avaliação ou previsão:

- `model.eval()` deve ser chamado primeiro.
 - O atributo `model.training` indica se o modelo está no modo de treinamento (`True`) ou avaliação (`False`).
-

Observação Importante

O PyTorch **não fornece** um loop de treinamento pronto.

Você precisa implementá-lo manualmente — o que oferece:

- Mais controle
- Maior clareza
- Flexibilidade total

Caso prefira um loop mais completo e automatizado, existem bibliotecas baseadas em PyTorch que fornecem:

- suporte a multi-GPU
- callbacks
- logging de métricas
- treinamento estruturado

📦 Exemplos:

- **PyTorch Lightning**
 - **FastAI**
 - **Catalyst**
 - **Keras** (desde a versão 3, também com suporte a PyTorch e JAX)
-

Após definir a função `train()`, basta chamá-la para iniciar o treinamento do modelo na GPU.

```
In [60]: train(model, optimizer, mse, train_loader, n_epochs)
```

```

Epoch 1/20, Loss: 0.5900
Epoch 2/20, Loss: 0.4046
Epoch 3/20, Loss: 0.3801
Epoch 4/20, Loss: 0.3629
Epoch 5/20, Loss: 0.3529
Epoch 6/20, Loss: 0.3520
Epoch 7/20, Loss: 0.3408
Epoch 8/20, Loss: 0.3427
Epoch 9/20, Loss: 0.3406
Epoch 10/20, Loss: 0.3378
Epoch 11/20, Loss: 0.3304
Epoch 12/20, Loss: 0.3267
Epoch 13/20, Loss: 0.3244
Epoch 14/20, Loss: 0.3221
Epoch 15/20, Loss: 0.3186
Epoch 16/20, Loss: 0.3149
Epoch 17/20, Loss: 0.3123
Epoch 18/20, Loss: 0.3111
Epoch 19/20, Loss: 0.3088
Epoch 20/20, Loss: 0.3072

```

5. Model Evaluation

Para avaliar o desempenho do modelo em um conjunto de dados específico, é criada uma função que:

- Recebe o **modelo** e um **DataLoader** contendo os dados de avaliação.
- Utiliza uma **função de métrica** para calcular o desempenho em cada batch.
- Utiliza uma **função de agregação** (por padrão, a média) para combinar as métricas dos batches e obter o valor final.

Essa estrutura permite avaliar o modelo de maneira eficiente, processando os dados em mini-batches e garantindo que o cálculo das métricas seja flexível e reutilizável.

```
In [61]: def evaluate(model, data_loader, metric_fn, aggregate_fn=torch.mean):
    model.eval()
    metrics = []
    with torch.no_grad():
        for X_batch, y_batch in data_loader:
            X_batch, y_batch = X_batch.to(device), y_batch.to(device)
            y_pred = model(X_batch)
            metric = metric_fn(y_pred, y_batch)
            metrics.append(metric)
    return aggregate_fn(torch.stack(metrics))
```

Para calcular o **MSE (Mean Squared Error)** no conjunto de validação:

1. Criar um **TensorDataset** a partir dos tensores de validação `X_valid` e `y_valid`.
2. Criar um **DataLoader** para iterar sobre o dataset de validação em mini-batches.
3. Passar o DataLoader para a função `evaluate()` junto com a métrica desejada.

Dessa forma, podemos calcular eficientemente a métrica de validação sem afetar o modelo treinado.

```
In [62]: valid_dataset = TensorDataset(X_valid, y_valid)
valid_loader = DataLoader(valid_dataset, batch_size=32)
valid_mse = evaluate(model, valid_loader, mse)
valid_mse
```

Out[62]: tensor(0.4080)

Se quisermos utilizar o **RMSE (Root Mean Squared Error)** ao invés do MSE:

- O PyTorch não fornece uma função pronta para RMSE.
- No entanto, podemos implementar facilmente a função de métrica personalizada:
 - Calcular o MSE de cada batch
 - Aplicar a **raiz quadrada** do resultado

Isso permite obter uma métrica mais interpretável, já que o RMSE está na mesma escala dos valores previstos.

```
In [63]: def rmse(y_pred, y_true):
    return ((y_pred - y_true) ** 2).mean().sqrt()

evaluate(model, valid_loader, rmse)
```

Out[63]: tensor(0.5668)

O RMSE deveria ser igual à raiz quadrada do MSE. No entanto, ao calcular a raiz quadrada do MSE obtido anteriormente, podemos perceber que o resultado é diferente do esperado.

Isso acontece porque o MSE pode ter sido calculado como a média das perdas de cada batch, e tirar a raiz dessa média não é o mesmo que calcular o RMSE de cada batch individualmente e depois agrregar.

Portanto, é importante definir claramente como a métrica será agregada ao usar o RMSE para garantir resultados consistentes.

Is this conversation helpful so far?

```
In [64]: valid_mse.sqrt()
```

Out[64]: tensor(0.6388)

O motivo da discrepância é que, em vez de calcular o RMSE sobre **todo o conjunto de validação**, calculamos o RMSE de **cada batch** e depois fizemos a média desses valores.

Essa abordagem **não é matematicamente equivalente** a calcular o RMSE considerando todas as amostras do conjunto de validação de uma só vez.

Para resolver isso:

- Podemos usar o **MSE como função de métrica** (`metric_fn`);

- É definir a função de agregação (`aggregate_fn`) para calcular a **raiz quadrada da média dos MSEs**, obtendo assim o RMSE correto para todo o conjunto.

```
In [65]: evaluate(model, valid_loader, mse,
                aggregate_fn=lambda metrics: torch.sqrt(torch.mean(metrics)))
```

Out[65]: tensor(0.6388)

Para simplificar o cálculo de métricas e garantir precisão, podemos usar a biblioteca **TorchMetrics**, criada pela equipe do PyTorch Lightning. Ela oferece métricas de **streaming**, que permitem atualizar os resultados **batch a batch** sem perder o acompanhamento global da métrica.

Principais vantagens:

- Evita implementar métricas manualmente;
- Mantém um registro contínuo das métricas ao longo de todo o conjunto de dados;
- Funciona de forma eficiente em mini-batches.

No Colab, a biblioteca não vem instalada por padrão, então é necessário executar:

```
In [66]: import torchmetrics

def evaluate_tm(model, data_loader, metric):
    model.eval()
    metric.reset() # redefina a métrica no início.
    with torch.no_grad():
        for X_batch, y_batch in data_loader:
            X_batch, y_batch = X_batch.to(device), y_batch.to(device)
            y_pred = model(X_batch)
            metric.update(y_pred, y_batch) # atualize-o a cada iteração.
    return metric.compute() # calcule o resultado final no fim
```

Com a TorchMetrics instalada, podemos:

- Criar uma métrica de **RMSE em streaming**, que acompanha o desempenho ao longo de todos os batches;
- **Mover a métrica para a GPU** para acelerar os cálculos;
- Usá-la para **avaliar o conjunto de validação**, garantindo que o RMSE seja calculado de forma correta e eficiente para todo o dataset.

```
In [67]: rmse = torchmetrics.MeanSquaredError(squared=False).to(device)
evaluate_tm(model, valid_loader, rmse)
```

Out[67]: tensor(0.6388)

```
In [68]: def train2(model, optimizer, criterion, metric, train_loader, valid_loader, n_epochs):
    history = {"train_losses": [], "train_metrics": [], "valid_metrics": []}
    for epoch in range(n_epochs):
        total_loss = 0.
        metric.reset()
        for X_batch, y_batch in train_loader:
            model.train()
            X_batch, y_batch = X_batch.to(device), y_batch.to(device)
```

```

y_pred = model(X_batch)
loss = criterion(y_pred, y_batch)
total_loss += loss.item()
loss.backward()
optimizer.step()
optimizer.zero_grad()
metric.update(y_pred, y_batch)
mean_loss = total_loss / len(train_loader)
history["train_losses"].append(mean_loss)
history["train_metrics"].append(metric.compute().item())
history["valid_metrics"].append(evaluate_tm(model, valid_loader, metric))
print(f"Epoch {epoch + 1}/{n_epochs}, "
      f"train loss: {history['train_losses'][-1]:.4f}, "
      f"train metric: {history['train_metrics'][-1]:.4f}, "
      f"valid metric: {history['valid_metrics'][-1]:.4f}")

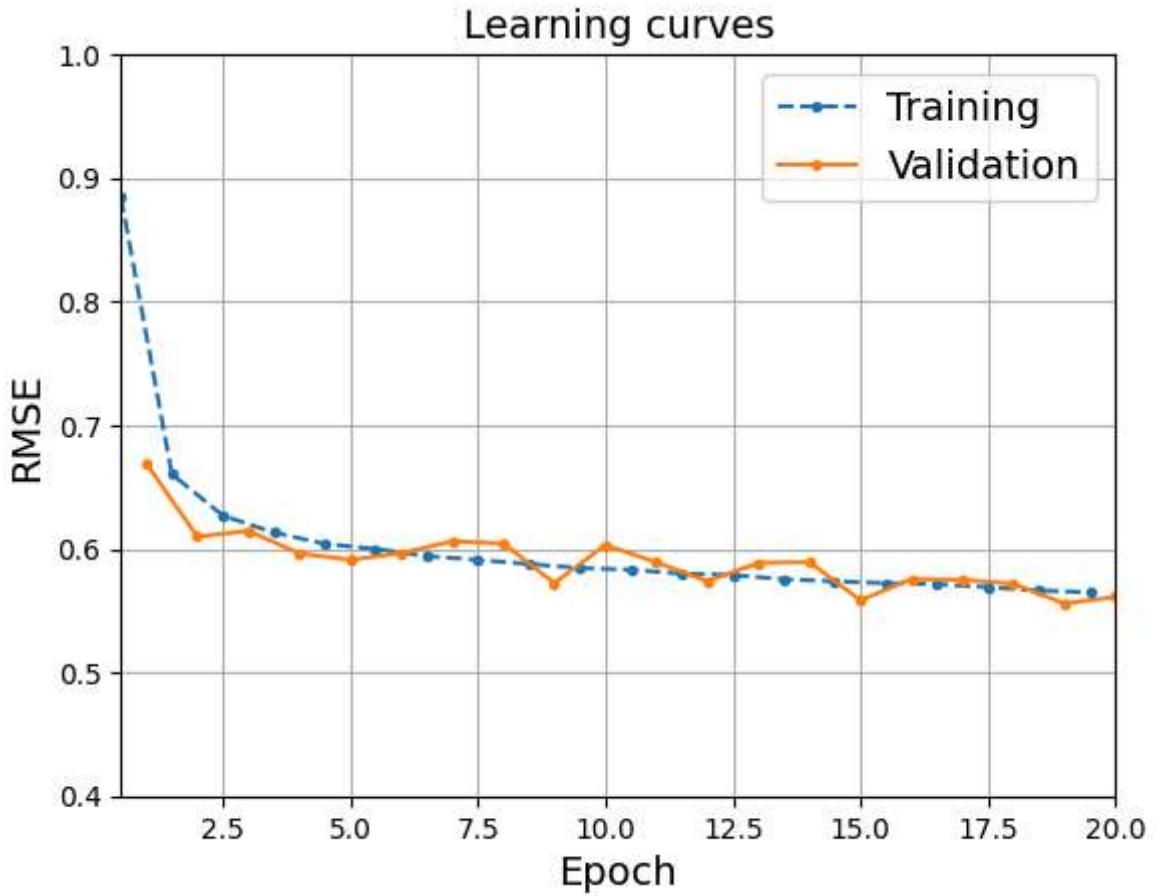
return history

torch.manual_seed(42)
learning_rate = 0.01
model = nn.Sequential(
    nn.Linear(n_features, 50), nn.ReLU(),
    nn.Linear(50, 40), nn.ReLU(),
    nn.Linear(40, 30), nn.ReLU(),
    nn.Linear(30, 1)
)
model = model.to(device)
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate, momentum=0)
mse = nn.MSELoss()
rmse = torchmetrics.MeanSquaredError(squared=False).to(device)
history = train2(model, optimizer, mse, rmse, train_loader, valid_loader,
                  n_epochs)

# Since we compute the training metric
plt.plot(np.arange(n_epochs) + 0.5, history["train_metrics"], ".--",
          label="Training")
plt.plot(np.arange(n_epochs) + 1.0, history["valid_metrics"], "-.",
          label="Validation")
plt.xlabel("Epoch")
plt.ylabel("RMSE")
plt.grid()
plt.title("Learning curves")
plt.axis([0.5, 20, 0.4, 1.0])
plt.legend()
plt.show()

```

Epoch 1/20, train loss: 0.7826, train metric: 0.8847, valid metric: 0.6690
 Epoch 2/20, train loss: 0.4362, train metric: 0.6605, valid metric: 0.6099
 Epoch 3/20, train loss: 0.3930, train metric: 0.6269, valid metric: 0.6145
 Epoch 4/20, train loss: 0.3759, train metric: 0.6132, valid metric: 0.5963
 Epoch 5/20, train loss: 0.3649, train metric: 0.6040, valid metric: 0.5911
 Epoch 6/20, train loss: 0.3598, train metric: 0.5999, valid metric: 0.5965
 Epoch 7/20, train loss: 0.3530, train metric: 0.5941, valid metric: 0.6061
 Epoch 8/20, train loss: 0.3495, train metric: 0.5911, valid metric: 0.6043
 Epoch 9/20, train loss: 0.3455, train metric: 0.5877, valid metric: 0.5723
 Epoch 10/20, train loss: 0.3415, train metric: 0.5844, valid metric: 0.6031
 Epoch 11/20, train loss: 0.3402, train metric: 0.5831, valid metric: 0.5891
 Epoch 12/20, train loss: 0.3362, train metric: 0.5799, valid metric: 0.5738
 Epoch 13/20, train loss: 0.3352, train metric: 0.5788, valid metric: 0.5886
 Epoch 14/20, train loss: 0.3310, train metric: 0.5754, valid metric: 0.5893
 Epoch 15/20, train loss: 0.3292, train metric: 0.5737, valid metric: 0.5584
 Epoch 16/20, train loss: 0.3273, train metric: 0.5722, valid metric: 0.5756
 Epoch 17/20, train loss: 0.3264, train metric: 0.5714, valid metric: 0.5749
 Epoch 18/20, train loss: 0.3237, train metric: 0.5690, valid metric: 0.5718
 Epoch 19/20, train loss: 0.3207, train metric: 0.5662, valid metric: 0.5556
 Epoch 20/20, train loss: 0.3190, train metric: 0.5649, valid metric: 0.5611



Agora que conseguimos calcular métricas corretamente, podemos atualizar a função `train()` para:

- Avaliar o desempenho do modelo **durante o treinamento** no conjunto de treino a cada época;
- Avaliar o desempenho no **conjunto de validação** ao final de cada época.

Observações importantes:

- Se a performance no conjunto de treino for muito melhor que no conjunto de validação, isso pode indicar:
 - **Overfitting;**
 - Algum **erro nos dados**, como desajuste entre o conjunto de treino e validação.
 - É recomendável **plotar as curvas de aprendizado** para analisar essas diferenças, usando:
 - **Matplotlib;**
 - Ou ferramentas de visualização como **TensorBoard**.
-

Próximos Passos

Até agora, aprendemos a:

- Construir, treinar e avaliar um **MLP de regressão** com PyTorch;
- Utilizar o modelo treinado para fazer previsões.

No entanto, tratamos apenas de **modelos sequenciais simples**, formados por camadas lineares e funções de ativação ReLU.

Para construir **modelos mais complexos e não sequenciais**, será necessário criar **módulos personalizados**.

6. Building Nonsequential Models Using Custom Modules

Alguns modelos de redes neurais precisam receber **várias entradas** que não podem ser facilmente combinadas em um único tensor. Isso ocorre, por exemplo, quando as entradas possuem **diferentes números de dimensões**, como no caso de se alimentar a rede neural com **imagens e textos** ao mesmo tempo.

Para lidar com essas situações, precisamos **adaptação o método `forward()`** do modelo para aceitar e processar múltiplas entradas separadamente.

No caso do modelo **Wide & Deep**, que combina diferentes tipos de dados (wide features e deep features), essa adaptação permite que cada entrada seja processada adequadamente antes de serem combinadas na rede.



In [69]:

```
class WideAndDeep(nn.Module):
    def __init__(self, n_features):
        super().__init__()
        self.deep_stack = nn.Sequential(
            nn.Linear(n_features, 50), nn.ReLU(),
            nn.Linear(50, 40), nn.ReLU(),
            nn.Linear(40, 30), nn.ReLU(),
```

```

        )
        self.output_layer = nn.Linear(30 + n_features, 1)

    def forward(self, X):
        deep_output = self.deep_stack(X)
        wide_and_deep = torch.concat([X, deep_output], dim=1)
        return self.output_layer(wide_and_deep)

```

No PyTorch, podemos incorporar **qualquer módulo** dentro de um módulo personalizado. No modelo **Wide & Deep**, a parte *deep* é uma MLP construída com `nn.Sequential`, e a **camada de saída** fica separada porque precisa receber a **concatenação da entrada original `X` com a saída da parte `deep`**. Como no código a MLP termina em 30 neurônios, a camada final recebe `30 + n_features` entradas.

No `forward()` fazemos:

1. Passar `X` pela stack *deep* (MLP) para obter `deep_output`.
2. **Concatenar** `X` e `deep_output` ao longo do eixo de features (`dim=1`).
3. Enviar essa concatenação para a **camada de saída**.

Depois de definir o módulo, seguimos o fluxo usual: **instanciar, mover para GPU** (se disponível) e **treinar/avaliar** como qualquer outro modelo. Essa abordagem combina um caminho *wide* (features originais, bom para padrões lineares/co-ocorrências) com um caminho *deep* (interações **não lineares**), mantendo um pipeline simples e flexível para lidar com **entradas múltiplas e combinações de módulos**.

In [70]:

```

torch.manual_seed(42)
model = WideAndDeep(n_features).to(device)
learning_rate = 0.002 # the model changed, so did the optimal Learning rate

```

`momentum` é um “impulso” no **SGD**: em vez de atualizar só com o gradiente do passo atual, o otimizador mantém uma **velocidade** (v) que acumula gradientes recentes (média exponencial). As regras são ($v \leftarrow \mu v + \nabla_{\theta} L$) e ($\theta \leftarrow \theta - lr, v$), onde (μ) é o `momentum` (ex.: 0.9). Isso **suaviza ruído**, acelera na direção consistente e **reduz zigue-zague** em vales alongados. Quando você usa `momentum=0`, vira **SGD puro** (sem memória do passado). Valores típicos: **0.9** ou **0.95**; às vezes **0.99** em problemas muito ruidosos. Se ativar `nesterov=True`, aplica a variante **Nesterov**, que “antecipa” o passo usando a velocidade (pode convergir um pouco melhor).

In [71]:

```

# extra code: train the model, exactly our previous models
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate, momentum=0)
mse = nn.MSELoss()
rmse = torchmetrics.MeanSquaredError(squared=False).to(device)
history = train2(model, optimizer, mse, rmse, train_loader, valid_loader,
                  n_epochs)

```

```
Epoch 1/20, train loss: 1.7802, train metric: 1.3344, valid metric: 0.8690
Epoch 2/20, train loss: 0.6201, train metric: 0.7875, valid metric: 0.9492
Epoch 3/20, train loss: 0.5900, train metric: 0.7682, valid metric: 0.7331
Epoch 4/20, train loss: 0.5607, train metric: 0.7488, valid metric: 0.7771
Epoch 5/20, train loss: 0.5408, train metric: 0.7353, valid metric: 0.7967
Epoch 6/20, train loss: 0.5244, train metric: 0.7241, valid metric: 0.7098
Epoch 7/20, train loss: 0.5070, train metric: 0.7119, valid metric: 0.7419
Epoch 8/20, train loss: 0.4941, train metric: 0.7030, valid metric: 0.6750
Epoch 9/20, train loss: 0.4798, train metric: 0.6928, valid metric: 0.6762
Epoch 10/20, train loss: 0.4657, train metric: 0.6825, valid metric: 0.6678
Epoch 11/20, train loss: 0.4538, train metric: 0.6736, valid metric: 0.6617
Epoch 12/20, train loss: 0.4441, train metric: 0.6665, valid metric: 0.6651
Epoch 13/20, train loss: 0.4328, train metric: 0.6580, valid metric: 0.6803
Epoch 14/20, train loss: 0.4232, train metric: 0.6506, valid metric: 0.6288
Epoch 15/20, train loss: 0.4139, train metric: 0.6434, valid metric: 0.6202
Epoch 16/20, train loss: 0.4063, train metric: 0.6374, valid metric: 0.6216
Epoch 17/20, train loss: 0.3991, train metric: 0.6317, valid metric: 0.6129
Epoch 18/20, train loss: 0.3937, train metric: 0.6274, valid metric: 0.6031
Epoch 19/20, train loss: 0.3879, train metric: 0.6228, valid metric: 0.6092
Epoch 20/20, train loss: 0.3834, train metric: 0.6193, valid metric: 0.5957
```

Em alguns casos, pode ser necessário enviar **subconjuntos diferentes de features** por caminhos distintos do modelo, como:

- Algumas features passam pelo **caminho wide**.
- Outras features (possivelmente sobrepostas) passam pelo **caminho deep**.

Para isso, uma abordagem comum é **dividir as entradas dentro do método `forward()`**.

Isso permite controlar exatamente quais features alimentam cada parte da rede, garantindo maior flexibilidade no processamento dos dados.

```
In [72]: class WideAndDeepV2(nn.Module):
    def __init__(self, n_features):
        super().__init__()
        self.deep_stack = nn.Sequential(
            nn.Linear(n_features - 2, 50), nn.ReLU(),
            nn.Linear(50, 40), nn.ReLU(),
            nn.Linear(40, 30), nn.ReLU(),
        )
        self.output_layer = nn.Linear(30 + 5, 1)

    def forward(self, X):
        X_wide = X[:, :5]
        X_deep = X[:, 2:]
        deep_output = self.deep_stack(X_deep)
        wide_and_deep = torch.concat([X_wide, deep_output], dim=1)
        return self.output_layer(wide_and_deep)
```

`WideAndDeepV2` divide as features em dois ramos: **wide** (5 primeiras colunas) e **deep** (todas as colunas a partir da 3^a). No `__init__`, a MLP (`deep_stack`) recebe `n_features - 2` entradas (bate com `X_deep = X[:, 2:]`, que ignora as duas primeiras colunas) e produz um vetor de tamanho **30**. A camada final (`output_layer`) recebe a **concatenação** de `X_wide` (tamanho **5**) com `deep_output` (tamanho **30**), totalizando **30 + 5** entradas, e gera uma saída escalar.

No `forward`:

1. `X_wide = X[:, :5]` pega as 5 primeiras features (caminho "memorizações lineares").
2. `X_deep = X[:, 2:]` pega do índice 2 em diante (ou seja, descarta as duas primeiras) para o caminho não linear.
3. `deep_output = self.deep_stack(X_deep)` processa no MLP.
4. `torch.concat([X_wide, deep_output], dim=1)` junta por **colunas** (mesmo batch).
5. `self.output_layer(...)` produz a predição.

Pré-condições e cuidados: **(a)** `n_features ≥ 5` (senão quebra); **(b)** as dimensões precisam bater: `nn.Linear(n_features - 2, 50) ⇔ X[:, 2:]`; **(c)** a **ordem/seleção** das colunas tem de ser estável do treino à inferência; **(d)** se mudar o nº de colunas no ramo wide, ajuste o `in_features` da `output_layer`. Para classificação binária, use `BCEWithLogitsLoss` (e `sigmoid` na inferência); para regressão, `MSELoss` / `RMSE`.

Embora seja possível dividir os inputs dentro do método `forward()`, muitas vezes é **mais conveniente que o modelo aceite dois tensores separados** como entrada.

Vantagens dessa abordagem:

- **Mais claro e organizado**: cada tensor representa explicitamente um subconjunto de features.
- **Evita confusões** ao separar manualmente os dados dentro do `forward`.
- **Facilita a reutilização** do modelo em diferentes contextos, como combinar imagens e textos ou diferentes tipos de dados estruturados.

```
In [73]: torch.manual_seed(42)
model = WideAndDeepV2(n_features).to(device)
```

```
In [74]: # extra code: train the model, exactly our previous models
learning_rate = 0.002
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate, momentum=0)
mse = nn.MSELoss()
rmse = torchmetrics.MeanSquaredError(squared=False).to(device)
history = train2(model, optimizer, mse, rmse, train_loader, valid_loader,
                  n_epochs)
```

```
Epoch 1/20, train loss: 1.8482, train metric: 1.3598, valid metric: 0.9100
Epoch 2/20, train loss: 0.6282, train metric: 0.7927, valid metric: 0.8028
Epoch 3/20, train loss: 0.5763, train metric: 0.7591, valid metric: 0.7567
Epoch 4/20, train loss: 0.5413, train metric: 0.7356, valid metric: 0.7290
Epoch 5/20, train loss: 0.5099, train metric: 0.7142, valid metric: 0.7011
Epoch 6/20, train loss: 0.4841, train metric: 0.6958, valid metric: 0.6816
Epoch 7/20, train loss: 0.4656, train metric: 0.6824, valid metric: 0.6670
Epoch 8/20, train loss: 0.4526, train metric: 0.6728, valid metric: 0.6576
Epoch 9/20, train loss: 0.4438, train metric: 0.6662, valid metric: 0.6539
Epoch 10/20, train loss: 0.4380, train metric: 0.6618, valid metric: 0.6498
Epoch 11/20, train loss: 0.4326, train metric: 0.6577, valid metric: 0.6470
Epoch 12/20, train loss: 0.4284, train metric: 0.6546, valid metric: 0.6447
Epoch 13/20, train loss: 0.4253, train metric: 0.6521, valid metric: 0.6452
Epoch 14/20, train loss: 0.4216, train metric: 0.6494, valid metric: 0.6468
Epoch 15/20, train loss: 0.4190, train metric: 0.6473, valid metric: 0.6484
Epoch 16/20, train loss: 0.4169, train metric: 0.6458, valid metric: 0.6452
Epoch 17/20, train loss: 0.4144, train metric: 0.6438, valid metric: 0.6459
Epoch 18/20, train loss: 0.4118, train metric: 0.6417, valid metric: 0.6470
Epoch 19/20, train loss: 0.4101, train metric: 0.6404, valid metric: 0.6475
Epoch 20/20, train loss: 0.4082, train metric: 0.6389, valid metric: 0.6493
```

6.1 Building Models with Multiple Inputs

Alguns modelos de redes neurais precisam receber **mais de uma entrada** que não podem ser facilmente combinadas em um único tensor. Isso acontece, por exemplo, quando as entradas possuem **diferentes números de dimensões**, como ao alimentar a rede com **imagens e textos simultaneamente**.

Para que o modelo **Wide & Deep** aceite **duas entradas separadas** (como ilustrado na Figura abaixo), é necessário **adaptação** o método `forward()` do modelo para processar cada entrada individualmente.



```
In [75]: class WideAndDeepV3(nn.Module):
    def __init__(self, n_features):
        super().__init__()
        self.deep_stack = nn.Sequential(
            nn.Linear(n_features - 2, 50), nn.ReLU(),
            nn.Linear(50, 40), nn.ReLU(),
            nn.Linear(40, 30), nn.ReLU(),
        )
        self.output_layer = nn.Linear(30 + 5, 1)

    def forward(self, X_wide, X_deep):
        deep_output = self.deep_stack(X_deep)
        wide_and_deep = torch.concat([X_wide, deep_output], dim=1)
        return self.output_layer(wide_and_deep)
```

Aqui, o `WideAndDeepV3` passa a **receber duas entradas separadas** no `forward(X_wide, X_deep)`: o ramo `deep` processa `X_deep` com a MLP (`deep_stack`) e o ramo `wide` usa `X_wide` **diretamente**. Depois, concatenamos os dois vetores ao longo do eixo de features (`dim=1`) e mandamos para a `output_layer`.

Assim, você pode alimentar o modelo com **tensores distintos** (ex.: campos categóricos one-hot no *wide* e embeddings/numéricos no *deep*), sem forçar tudo para um único tensor. Pré-condições: `X_wide.shape[1]` precisa ser **5** (como definido na `output_layer`) e `X_deep.shape[1]` precisa ser `n_features - 2` (como definido na primeira `Linear`). No treino/validação, o seu `DataLoader` deve devolver **triplas** (`X_wide`, `X_deep`, `y`) e a chamada vira `y_pred = model(X_wide, X_deep)`; o resto do loop (`loss`, `backward`, `step`, `zero_grad`) permanece igual. Exemplo mínimo de uso: `pred = model(X[:, :5], X[:, 2:])` (desde que essas fatias respeitem as dimensões esperadas).

Para trabalhar com múltiplas entradas no modelo Wide & Deep, precisamos criar **datasets que retornem separadamente as entradas para o caminho wide e para o caminho deep**.

Isso garante que cada subconjunto de features seja fornecido ao caminho correto da rede, mantendo a consistência e a clareza do fluxo de dados durante o treinamento e a avaliação.

```
In [76]: torch.manual_seed(42)
# Divisão da Base de dados (tensores) para Wide
#Train
train_data_wd = TensorDataset(X_train[:, :5], X_train[:, 2:], y_train)
train_loader_wd = DataLoader(train_data_wd, batch_size=32, shuffle=True)
#Valid
valid_data_wd = TensorDataset(X_valid[:, :5], X_valid[:, 2:], y_valid)
valid_loader_wd = DataLoader(valid_data_wd, batch_size=32)
#Test
test_data_wd = TensorDataset(X_test[:, :5], X_test[:, 2:], y_test)
test_loader_wd = DataLoader(test_data_wd, batch_size=32)
```

Esse trecho cria **datasets e data loaders** para o modelo Wide & Deep com **duas entradas por amostra**. `TensorDataset(X_train[:, :5], X_train[:, 2:], y_train)` empacota três tensores: o **ramo wide** (`X[:, :5]`, primeiras 5 colunas), o **ramo deep** (`X[:, 2:]`, da 3^a coluna em diante) e o **alvo y**. O `DataLoader(..., batch_size=32, shuffle=True)` entrega lotes embaralhados de 32 amostras; nos valid/test não embaralhamos. Assim, no loop você recebe **triplas** (`X_wide`, `X_deep`, `y`) e chama `model(X_wide, X_deep)`. O `torch.manual_seed(42)` fixa a aleatoriedade (por exemplo, a ordem de embaralhamento). Atenção: as fatias usadas aqui precisam **bater** com as dimensões esperadas pelo modelo (`X_wide` com 5 colunas; `X_deep` com `n_features-2`) e devem ser **consistentes** em train/valid/test.

```
In [77]: def evaluate_multi_in(model, data_loader, metric):
    model.eval()
    metric.reset() # reset the metric at the beginning
    with torch.no_grad():
        for X_batch_wide, X_batch_deep, y_batch in data_loader:
            X_batch_wide = X_batch_wide.to(device)
            X_batch_deep = X_batch_deep.to(device)
            y_batch = y_batch.to(device)
            y_pred = model(X_batch_wide, X_batch_deep)
            metric.update(y_pred, y_batch) # update it at each iteration
    return metric.compute() # compute the final result at the end
```

```

def train_multi_in(model, optimizer, criterion, metric, train_loader,
                   valid_loader, n_epochs):
    history = {"train_losses": [], "train_metrics": [], "valid_metrics": []}
    for epoch in range(n_epochs):
        total_loss = 0.
        metric.reset()
        for *X_batch_inputs, y_batch in train_loader:
            model.train()
            X_batch_inputs = [X.to(device) for X in X_batch_inputs]
            y_batch = y_batch.to(device)
            y_pred = model(*X_batch_inputs)
            loss = criterion(y_pred, y_batch)
            total_loss += loss.item()
            loss.backward()
            optimizer.step()
            optimizer.zero_grad()
            metric.update(y_pred, y_batch)
        mean_loss = total_loss / len(train_loader)
        history["train_losses"].append(mean_loss)
        history["train_metrics"].append(metric.compute().item())
        history["valid_metrics"].append(
            evaluate_multi_in(model, valid_loader, metric).item())
        print(f"Epoch {epoch + 1}/{n_epochs}, "
              f"train loss: {history['train_losses'][-1]:.4f}, "
              f"train metric: {history['train_metrics'][-1]:.4f}, "
              f"valid metric: {history['valid_metrics'][-1]:.4f}")
    return history

torch.manual_seed(42)
learning_rate = 0.01
model = WideAndDeepV3(n_features).to(device)
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate, momentum=0)
mse = nn.MSELoss()
rmse = torchmetrics.MeanSquaredError(squared=False).to(device)
history = train_multi_in(model, optimizer, mse, rmse, train_loader_wd,
                         valid_loader_wd, n_epochs)

```

Epoch 1/20, train loss: 0.8366, train metric: 0.9148, valid metric: 0.6892
 Epoch 2/20, train loss: 0.4627, train metric: 0.6803, valid metric: 0.6455
 Epoch 3/20, train loss: 0.4319, train metric: 0.6572, valid metric: 0.6374
 Epoch 4/20, train loss: 0.4259, train metric: 0.6525, valid metric: 0.6512
 Epoch 5/20, train loss: 0.4120, train metric: 0.6420, valid metric: 0.6305
 Epoch 6/20, train loss: 0.4040, train metric: 0.6356, valid metric: 0.6287
 Epoch 7/20, train loss: 0.4005, train metric: 0.6330, valid metric: 0.6252
 Epoch 8/20, train loss: 0.3976, train metric: 0.6306, valid metric: 0.6158
 Epoch 9/20, train loss: 0.3883, train metric: 0.6230, valid metric: 0.7407
 Epoch 10/20, train loss: 0.3866, train metric: 0.6218, valid metric: 0.6063
 Epoch 11/20, train loss: 0.3752, train metric: 0.6125, valid metric: 0.5974
 Epoch 12/20, train loss: 0.3704, train metric: 0.6087, valid metric: 0.5887
 Epoch 13/20, train loss: 0.3677, train metric: 0.6063, valid metric: 0.5982
 Epoch 14/20, train loss: 0.3604, train metric: 0.6004, valid metric: 0.5841
 Epoch 15/20, train loss: 0.3592, train metric: 0.5994, valid metric: 0.6401
 Epoch 16/20, train loss: 0.3640, train metric: 0.6034, valid metric: 0.5747
 Epoch 17/20, train loss: 0.3536, train metric: 0.5947, valid metric: 0.5686
 Epoch 18/20, train loss: 0.3461, train metric: 0.5884, valid metric: 0.5679
 Epoch 19/20, train loss: 0.3436, train metric: 0.5862, valid metric: 0.5623
 Epoch 20/20, train loss: 0.3415, train metric: 0.5844, valid metric: 0.5696

- `model.eval()` — coloca o modelo em **modo avaliação**: desativa comportamentos de treino (ex.: *dropout*) e faz *BatchNorm* usar **estatísticas fixas**.
- `metric.reset()` — **zera o estado interno** da métrica do `torchmetrics` (contadores/somas) para começar uma nova medição limpa.
- `X_batch_inputs = [X.to(device) for X in X_batch_inputs]` — move **todas as entradas do batch** (os dois ramos: `X_wide`, `X_deep`, etc.) para o **mesmo device** do modelo (GPU/CPU), mantendo a ordem; depois elas são passadas como `model(*X_batch_inputs)`.
- `metric.update(y_pred, y_batch)` — **acumula** o resultado desse batch dentro da métrica (sem calcular ainda o valor final). No fim da época, `metric.compute()` usa tudo que foi acumulado para devolver a métrica **global**.

Quando o modelo possui **múltiplas entradas**, é fácil confundir a ordem dos tensores, o que pode gerar **erros difíceis de depurar**.

Para evitar isso, uma boa prática é **nomear cada entrada**.

Uma abordagem é criar um **dataset personalizado** que retorne um **dicionário**, onde as chaves são os nomes das entradas e os valores são os tensores correspondentes.

Isso torna o código mais legível e reduz a chance de erros ao alimentar o modelo.

```
In [78]: class WideAndDeepDataset(torch.utils.data.Dataset):
    def __init__(self, X_wide, X_deep, y):
        self.X_wide = X_wide
        self.X_deep = X_deep
        self.y = y

    def __len__(self):
        return len(self.y)

    def __getitem__(self, idx):
        input_dict = {"X_wide": self.X_wide[idx], "X_deep": self.X_deep[idx]}
        return input_dict, self.y[idx]
```

Após definir como as entradas serão retornadas (separadas ou nomeadas), o próximo passo é **criar os datasets e os data loaders**.

- **Datasets**: fornecem os dados ao modelo de forma estruturada, retornando os tensores correspondentes a cada entrada.
- **Data Loaders**: permitem iterar sobre os datasets em **mini-batches**, embaralhar os dados e facilitar o treinamento eficiente no PyTorch.

Essa etapa garante que o modelo receba os dados corretamente e de forma eficiente durante o treinamento e avaliação.

```
In [79]: torch.manual_seed(42)
train_data_named = WideAndDeepDataset(
```

```

    X_wide=X_train[:, :5], X_deep=X_train[:, 2:], y=y_train)
train_loader_named = DataLoader(train_data_named, batch_size=32, shuffle=True)
valid_data_named = WideAndDeepDataset(
    X_wide=X_valid[:, :5], X_deep=X_valid[:, 2:], y=y_valid)
valid_loader_named = DataLoader(valid_data_named, batch_size=32)
test_data_named = WideAndDeepDataset(
    X_wide=X_test[:, :5], X_deep=X_test[:, 2:], y=y_test)
test_loader_named = DataLoader(test_data_named, batch_size=32)

```

```

In [80]: def evaluate_named(model, data_loader, metric):
    model.eval()
    metric.reset() # reset the metric at the beginning
    with torch.no_grad():
        for inputs, y_batch in data_loader:
            inputs = {name: X.to(device) for name, X in inputs.items()}
            y_batch = y_batch.to(device)
            y_pred = model(X_wide=inputs["X_wide"], X_deep=inputs["X_deep"])
            metric.update(y_pred, y_batch)
    return metric.compute() # compute the final result at the end

def train_named(model, optimizer, criterion, metric, train_loader,
               valid_loader, n_epochs):
    history = {"train_losses": [], "train_metrics": [], "valid_metrics": []}
    for epoch in range(n_epochs):
        total_loss = 0.
        metric.reset()
        for inputs, y_batch in train_loader:
            model.train()
            inputs = {name: X.to(device) for name, X in inputs.items()}
            y_batch = y_batch.to(device)
            y_pred = model(**inputs)
            loss = criterion(y_pred, y_batch)
            total_loss += loss.item()
            loss.backward()
            optimizer.step()
            optimizer.zero_grad()
            metric.update(y_pred, y_batch)
        mean_loss = total_loss / len(train_loader)
        history["train_losses"].append(mean_loss)
        history["train_metrics"].append(metric.compute().item())
        history["valid_metrics"].append(
            evaluate_named(model, valid_loader, metric).item())
        print(f"Epoch {epoch + 1}/{n_epochs}, "
              f"train loss: {history['train_losses'][-1]:.4f}, "
              f"train metric: {history['train_metrics'][-1]:.4f}, "
              f"valid metric: {history['valid_metrics'][-1]:.4f}")
    return history

torch.manual_seed(42)
learning_rate = 0.01
model = WideAndDeepV3(n_features).to(device)
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate, momentum=0)
mse = nn.MSELoss()
rmse = torchmetrics.MeanSquaredError(squared=False).to(device)
history = train_named(model, optimizer, mse, rmse, train_loader_named,
                      valid_loader_named, n_epochs)

```

Epoch 1/20, train loss: 0.8366, train metric: 0.9148, valid metric: 0.6892
Epoch 2/20, train loss: 0.4627, train metric: 0.6803, valid metric: 0.6455
Epoch 3/20, train loss: 0.4319, train metric: 0.6572, valid metric: 0.6374
Epoch 4/20, train loss: 0.4259, train metric: 0.6525, valid metric: 0.6512
Epoch 5/20, train loss: 0.4120, train metric: 0.6420, valid metric: 0.6305
Epoch 6/20, train loss: 0.4040, train metric: 0.6356, valid metric: 0.6287
Epoch 7/20, train loss: 0.4005, train metric: 0.6330, valid metric: 0.6252
Epoch 8/20, train loss: 0.3976, train metric: 0.6306, valid metric: 0.6158
Epoch 9/20, train loss: 0.3883, train metric: 0.6230, valid metric: 0.7407
Epoch 10/20, train loss: 0.3866, train metric: 0.6218, valid metric: 0.6063
Epoch 11/20, train loss: 0.3752, train metric: 0.6125, valid metric: 0.5974
Epoch 12/20, train loss: 0.3704, train metric: 0.6087, valid metric: 0.5887
Epoch 13/20, train loss: 0.3677, train metric: 0.6063, valid metric: 0.5982
Epoch 14/20, train loss: 0.3604, train metric: 0.6004, valid metric: 0.5841
Epoch 15/20, train loss: 0.3592, train metric: 0.5994, valid metric: 0.6401
Epoch 16/20, train loss: 0.3640, train metric: 0.6034, valid metric: 0.5747
Epoch 17/20, train loss: 0.3536, train metric: 0.5947, valid metric: 0.5686
Epoch 18/20, train loss: 0.3461, train metric: 0.5884, valid metric: 0.5679
Epoch 19/20, train loss: 0.3436, train metric: 0.5862, valid metric: 0.5623
Epoch 20/20, train loss: 0.3415, train metric: 0.5844, valid metric: 0.5696

6.2 Building Models with Multiple Outputs

Existem diversos cenários em que **uma rede neural precisa ter múltiplas saídas**:

1. Tarefas que exigem múltiplos tipos de previsão

- Por exemplo, localizar e classificar o objeto principal em uma imagem.
- Isso combina **regressão** (localização) e **classificação** (tipo do objeto) na mesma rede.

2. Multitask Learning (Aprendizado Multitarefa)

- Quando temos **múltimas tarefas independentes** com base nos mesmos dados.
- Treinar uma rede por tarefa é possível, mas **treinar uma única rede com uma saída por tarefa** geralmente gera melhores resultados.
- A rede aprende **features compartilhadas úteis** para todas as tarefas.
- Exemplo: classificar expressões faciais (sorriso, surpresa) e identificar se a pessoa usa óculos simultaneamente.

3. Regularização com Saídas Auxiliares

- Adicionar uma saída auxiliar ajuda a **reduzir overfitting** e melhorar a generalização.
- A saída auxiliar força a **parte subjacente da rede** a aprender algo útil por si só.
- Exemplo no modelo **Wide & Deep**:
 - A saída da stack deep tem dimensão 40.
 - O alvo tem dimensão 1.
 - Adicionamos uma camada `nn.Linear` para mapear de 40 → 1 para a saída auxiliar.
 - O método `forward()` deve calcular **tanto a saída principal quanto a saída auxiliar**.



```
In [81]: class WideAndDeepV4(nn.Module):
    def __init__(self, n_features):
        super().__init__()
        self.deep_stack = nn.Sequential(
            nn.Linear(n_features - 2, 50), nn.ReLU(),
            nn.Linear(50, 40), nn.ReLU(),
            nn.Linear(40, 30), nn.ReLU(),
        )
        self.output_layer = nn.Linear(30 + 5, 1)
        self.aux_output_layer = nn.Linear(30, 1)

    def forward(self, X_wide, X_deep):
        deep_output = self.deep_stack(X_deep)
        wide_and_deep = torch.concat([X_wide, deep_output], dim=1)
        main_output = self.output_layer(wide_and_deep)
        aux_output = self.aux_output_layer(deep_output)
        return main_output, aux_output
```

WideAndDeepV4 é um **Wide & Deep com saída auxiliar**: o ramo *deep* (MLP) produz um vetor de 30 unidades; dele saem duas cabeças. A **saída principal** (`main_output`) recebe a concatenação do *wide* (5 colunas de entrada direta) com o `deep_output` (30) e passa por `Linear(35→1)`; já a **saída auxiliar** (`aux_output`) conecta **diretamente** o `deep_output` em `Linear(30→1)`. A ideia é fazer **supervisão auxiliar** no ramo profundo para estabilizar e regularizar o treino (o gradiente chega mais “perto” das camadas internas). Na prática, você treina somando perdas, por exemplo: `loss = L_main(main_output, y) + λ * L_aux(aux_output, y)`, com λ pequeno (ex.: 0.1). Em inferência, usa-se só a **saída principal**. Cuidados: `X_wide` deve ter 5 features, `X_deep` deve ter `n_features-2`; as duas saídas têm shape `(batch, 1)` para regressão (ou ajuste a cabeça para classificação conforme o problema).

```
In [82]: import torchmetrics

def evaluate_multi_out(model, data_loader, metric):
    model.eval()
    metric.reset()
    with torch.no_grad():
        for inputs, y_batch in data_loader:
            inputs = {name: X.to(device) for name, X in inputs.items()}
            y_batch = y_batch.to(device)
            y_pred, _ = model(**inputs)
            metric.update(y_pred, y_batch)
    return metric.compute()

def train_multi_out(model, optimizer, criterion, metric, train_loader,
                    valid_loader, n_epochs):
    history = {"train_losses": [], "train_metrics": [], "valid_metrics": []}
    for epoch in range(n_epochs):
        total_loss = 0.
        metric.reset()
        for inputs, y_batch in train_loader:
            model.train()
            inputs = {name: X.to(device) for name, X in inputs.items()}
            y_batch = y_batch.to(device)
            y_pred, y_pred_aux = model(**inputs)
            main_loss = criterion(y_pred, y_batch)
            aux_loss = criterion(y_pred_aux, y_batch)
            loss = 0.8 * main_loss + 0.2 * aux_loss
            total_loss += loss.item()
            loss.backward()
            optimizer.step()
            optimizer.zero_grad()
            metric.update(y_pred, y_batch)
        mean_loss = total_loss / len(train_loader)
        history["train_losses"].append(mean_loss)
        history["train_metrics"].append(metric.compute().item())
        history["valid_metrics"].append(
            evaluate_multi_out(model, valid_loader, metric).item())
        print(f"Epoch {epoch + 1}/{n_epochs}, "
              f"train loss: {history['train_losses'][-1]:.4f}, "
              f"train metric: {history['train_metrics'][-1]:.4f}, "
              f"valid metric: {history['valid_metrics'][-1]:.4f}")
    return history

torch.manual_seed(42)
```

```

learning_rate = 0.01
model = WideAndDeepV4(n_features).to(device)
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate, momentum=0)
mse = nn.MSELoss()
rmse = torchmetrics.MeanSquaredError(squared=False).to(device)
history = train_multi_out(model, optimizer, mse, rmse, train_loader_named,
                           valid_loader_named, n_epochs)

```

Epoch 1/20, train loss: 1.0693, train metric: 0.9506, valid metric: 0.7085
 Epoch 2/20, train loss: 0.5817, train metric: 0.6946, valid metric: 0.6607
 Epoch 3/20, train loss: 0.5010, train metric: 0.6581, valid metric: 0.6425
 Epoch 4/20, train loss: 0.4690, train metric: 0.6497, valid metric: 0.6654
 Epoch 5/20, train loss: 0.4503, train metric: 0.6420, valid metric: 0.6338
 Epoch 6/20, train loss: 0.4387, train metric: 0.6373, valid metric: 0.6563
 Epoch 7/20, train loss: 0.4315, train metric: 0.6330, valid metric: 0.6193
 Epoch 8/20, train loss: 0.4249, train metric: 0.6302, valid metric: 0.6167
 Epoch 9/20, train loss: 0.4116, train metric: 0.6202, valid metric: 0.6450
 Epoch 10/20, train loss: 0.4085, train metric: 0.6198, valid metric: 0.5938
 Epoch 11/20, train loss: 0.4073, train metric: 0.6196, valid metric: 0.5959
 Epoch 12/20, train loss: 0.3914, train metric: 0.6077, valid metric: 0.6074
 Epoch 13/20, train loss: 0.3847, train metric: 0.6033, valid metric: 0.5813
 Epoch 14/20, train loss: 0.3848, train metric: 0.6047, valid metric: 0.6032
 Epoch 15/20, train loss: 0.3745, train metric: 0.5965, valid metric: 0.5729
 Epoch 16/20, train loss: 0.3693, train metric: 0.5929, valid metric: 0.6017
 Epoch 17/20, train loss: 0.3670, train metric: 0.5919, valid metric: 0.6049
 Epoch 18/20, train loss: 0.3612, train metric: 0.5873, valid metric: 0.5677
 Epoch 19/20, train loss: 0.3597, train metric: 0.5863, valid metric: 0.5738
 Epoch 20/20, train loss: 0.3556, train metric: 0.5831, valid metric: 0.6082

- **Entradas nomeadas (dict) e expansão com ** Agora o `DataLoader` entrega `inputs` como **dicionário** (ex.: `{"X_wide": ..., "X_deep": ...}`), não mais tupla. Por isso usamos:

```

inputs = {name: X.to(device) for name, X in inputs.items()}
y_pred, y_pred_aux = model(**inputs) # chaves do dict devem bater
                                    com a assinatura do forward

```

Diferente de antes, quando fazíamos `model(X_wide, X_deep)` diretamente.

- **Duas saídas (principal + auxiliar)** O modelo retorna **duas previsões**: `y_pred` (principal) e `y_pred_aux` (auxiliar). Na avaliação, só usamos a **principal**:

```

y_pred, _ = model(**inputs)
metric.update(y_pred, y_batch)

```

- **Perda composta com pesos** O treino combina as duas perdas:

```

main_loss = criterion(y_pred, y_batch)
aux_loss = criterion(y_pred_aux, y_batch)
loss = 0.8 * main_loss + 0.2 * aux_loss

```

Antes havia uma única perda; aqui há **supervisão auxiliar** ponderada.

- **DataLoaders “named”** Usamos `train_loader_named/valid_loader_named`, que **entregam** `(inputs_dict, y)` já com nomes compatíveis com `forward`.

Antes os loaders entregavam tuplas (`X_wide, X_deep, y`).

Usar **entradas nomeadas** (dict + `**inputs`) evita bugs de ordem dos tensores, deixa o código mais legível, e facilita escalar o modelo (adicionar/remover ramos como texto/imagem/tabular sem refatorar chamadas). Já a **saída auxiliar** com perda ponderada fornece sinal de gradiente mais curto para o ramo deep, atua como regularização, estabiliza o início do treino e permite monitorar melhor onde está o gargalo (comparando `main_loss` e `aux_loss`). Em suma: nomes trazem **manutenibilidade e segurança**; a cabeça auxiliar traz **estabilidade e generalização**.

7. Building an Image Classifier with PyTorch

Neste capítulo, continuaremos trabalhando com o **dataset Fashion MNIST**. A primeira etapa é fazer o download dos dados.

7.1 Using TorchVision to Load the Dataset

A **TorchVision** é uma biblioteca essencial no ecossistema PyTorch para **visão computacional**. Ela oferece:

- Funções utilitárias para **baixar datasets comuns**, como MNIST e Fashion MNIST.
- **Modelos pré-treinados** para diversas tarefas de visão computacional (ver Capítulo 12).
- Funções para **transformar imagens**, incluindo corte, rotação, redimensionamento, entre outras.

Observação: No Google Colab, a TorchVision já vem pré-instalada.

No caso do Fashion MNIST:

- O dataset já vem dividido em:
 - **Treinamento:** 60.000 imagens
 - **Teste:** 10.000 imagens
- Para validação, podemos reservar **5.000 imagens** do conjunto de treinamento usando a função `random_split()` do PyTorch.

```
In [84]: import torchvision
import torchvision.transforms.v2 as T

toTensor = T.Compose([T.ToImage(), T.ToDtype(torch.float32, scale=True)])

train_and_valid_data = torchvision.datasets.FashionMNIST(
    root="datasets", train=True, download=True, transform=toTensor)
test_data = torchvision.datasets.FashionMNIST(
    root="datasets", train=False, download=True, transform=toTensor)

torch.manual_seed(42)
train_data, valid_data = torch.utils.data.random_split(
    train_and_valid_data, [55_000, 5_000])
```

100%	[progress bar]	26.4M/26.4M [00:20<00:00, 1.28MB/s]
100%	[progress bar]	29.5k/29.5k [00:00<00:00, 116kB/s]
100%	[progress bar]	4.42M/4.42M [00:05<00:00, 821kB/s]
100%	[progress bar]	5.15k/5.15k [00:00<00:00, 19.2MB/s]

Esse código baixa o **FashionMNIST** via TorchVision, **converte as imagens para tensor float32 em [0,1]** e separa treino/validation de forma reproduzível. O `toTensor` usa `transforms.v2 : T.ToImage()` garante o formato de imagem e `T.ToDtype(torch.float32, scale=True)` faz o cast para `float32` **escalando** de 0-255 para **0-1**. Em seguida, `FashionMNIST(..., train=True/False, download=True, transform=toTensor)` cria os datasets de **treino (60k)** e **teste (10k)**, salvando em `datasets/`. As imagens são **1×28×28** (tons de cinza) e o transform é aplicado **on the fly** quando você lê cada amostra. Por fim, com `torch.manual_seed(42)` e `random_split(train_and_valid_data, [55_000, 5_000])`, você reserva **5.000** para **validação** e mantém **55.000** para **treino**, com divisão **determinística** por causa da seed. (Se quiser padronizar ainda mais, depois pode adicionar `T.Normalize(mean, std)` ao `Compose`.)

```
In [85]: torch.manual_seed(42)
train_loader = DataLoader(train_data, batch_size=32, shuffle=True)
valid_loader = DataLoader(valid_data, batch_size=32)
test_loader = DataLoader(test_data, batch_size=32)
```

Cada entrada é uma tupla (imagem, alvo):

```
In [86]: X_sample, y_sample = train_data[0]
```

Cada imagem tem um formato [canais, linhas, colunas]. Imagens em tons de cinza, como as do Fashion MNIST, possuem um único canal (enquanto imagens RGB possuem 3, e outros tipos de imagens, como imagens de satélite, podem ter muito mais). Imagens de moda são em tons de cinza e têm 28x28 pixels:

```
In [87]: X_sample.shape
```

```
Out[87]: torch.Size([1, 28, 28])
```

```
In [88]: X_sample.dtype
```

```
Out[88]: torch.float32
```

```
In [89]: train_and_valid_data.classes[y_sample]
```

```
Out[89]: 'Ankle boot'
```

7.2 Building the Classifier

Vamos criar um **módulo personalizado** em PyTorch para uma **MLP (Multi-Layer Perceptron)** com **duas camadas ocultas**.

Este módulo servirá para classificar as imagens do Fashion MNIST, permitindo:

- Definir facilmente o número de **neurônios em cada camada**.
- Aplicar funções de ativação entre as camadas.
- Facilitar o **treinamento e avaliação** do modelo usando o framework PyTorch.

```
In [90]: class ImageClassifier(nn.Module):
    def __init__(self, n_inputs, n_hidden1, n_hidden2, n_classes):
        super().__init__()
        self.mlp = nn.Sequential(
            nn.Flatten(),                                     #Embedding
            nn.Linear(n_inputs, n_hidden1),
            nn.ReLU(),
            nn.Linear(n_hidden1, n_hidden2),
            nn.ReLU(),
            nn.Linear(n_hidden2, n_classes)
        )

    def forward(self, X):
        return self.mlp(X)

torch.manual_seed(42)
model = ImageClassifier(n_inputs=1 * 28 * 28, n_hidden1=300, n_hidden2=100,
                       n_classes=10).to(device)
xentropy = nn.CrossEntropyLoss()
```

`ImageClassifier` é uma MLP para imagens $1 \times 28 \times 28$. A primeira camada é `nn.Flatten()`, que transforma cada imagem de shape (N, C, H, W) em $(N, C*H*W)$ — aqui vira $(N, 784)$. Em seguida, os dados passam por `Linear`→`ReLU`→`Linear`→`ReLU`→`Linear`, mapeando $784 \rightarrow 300 \rightarrow 100 \rightarrow 10$. O método `forward(self, X)` apenas define o **caminho de passagem**: ele aplica `self.mlp(X)` e retorna os **logits** (pontuações não normalizadas) de shape $(batch, 10)$. Já `xentropy = nn.CrossEntropyLoss()` é a **perda de classificação multiclasse** que **espera logits** e rótulos inteiros (`torch.long`, valores 0–9); ela combina **LogSoftmax + NLLLoss** internamente, então **não aplique Softmax antes**. No treino, você passa `(logits, targets)` para a perda; na inferência, use `logits.argmax(dim=1)` (ou `softmax(logits, dim=1)` para probabilidades).

Logits são as **pontuações não normalizadas** que o modelo produz antes de virar probabilidade. Para classificação multiclasse, o vetor de logits (um por classe) é convertido em probabilidades pelo **softmax**; para binária (saída de 1 unidade), usa-se o **sigmoid**. Eles podem ser quaisquer números reais (negativos/positivos) e indicam “preferência relativa”: quanto maior o logit de uma classe, maior a probabilidade após a normalização. Em PyTorch, `nn.CrossEntropyLoss` já **espera logits** e aplica `log_softmax` + NLL internamente (por isso **não aplique softmax antes**); de forma análoga, `BCEWithLogitsLoss` espera logits e incorpora o sigmoid com melhorias de **estabilidade numérica**. Em inferência, você pega a classe com `logits.argmax(dim=1)` ou transforma em probabilidades com `softmax(logits, dim=1)`.

```
In [91]: optimizer = torch.optim.SGD(model.parameters(), lr=0.1)
accuracy = torchmetrics.Accuracy(task="multiclass", num_classes=10).to(device)
_ = train2(model, optimizer, xentropy, accuracy, train_loader, valid_loader,
n_epochs)
```

```
Epoch 1/20, train loss: 0.6060, train metric: 0.7810, valid metric: 0.8426
Epoch 2/20, train loss: 0.4059, train metric: 0.8493, valid metric: 0.8418
Epoch 3/20, train loss: 0.3628, train metric: 0.8665, valid metric: 0.8540
Epoch 4/20, train loss: 0.3350, train metric: 0.8769, valid metric: 0.8624
Epoch 5/20, train loss: 0.3143, train metric: 0.8845, valid metric: 0.8770
Epoch 6/20, train loss: 0.2984, train metric: 0.8872, valid metric: 0.8636
Epoch 7/20, train loss: 0.2847, train metric: 0.8923, valid metric: 0.8734
Epoch 8/20, train loss: 0.2737, train metric: 0.8970, valid metric: 0.8752
Epoch 9/20, train loss: 0.2624, train metric: 0.9015, valid metric: 0.8834
Epoch 10/20, train loss: 0.2525, train metric: 0.9043, valid metric: 0.8704
Epoch 11/20, train loss: 0.2444, train metric: 0.9077, valid metric: 0.8834
Epoch 12/20, train loss: 0.2360, train metric: 0.9099, valid metric: 0.8854
Epoch 13/20, train loss: 0.2280, train metric: 0.9128, valid metric: 0.8882
Epoch 14/20, train loss: 0.2211, train metric: 0.9159, valid metric: 0.8678
Epoch 15/20, train loss: 0.2170, train metric: 0.9170, valid metric: 0.8802
Epoch 16/20, train loss: 0.2076, train metric: 0.9205, valid metric: 0.8874
Epoch 17/20, train loss: 0.2037, train metric: 0.9229, valid metric: 0.8916
Epoch 18/20, train loss: 0.1991, train metric: 0.9236, valid metric: 0.8854
Epoch 19/20, train loss: 0.1921, train metric: 0.9268, valid metric: 0.8828
Epoch 20/20, train loss: 0.1869, train metric: 0.9292, valid metric: 0.8678
```

Esse trecho configura o treino e o executa. Primeiro, `optimizer = torch.optim.SGD(model.parameters(), lr=0.1)` escolhe o **otimizador SGD** com taxa de aprendizado 0.1 para atualizar os **parâmetros** do `model`. Depois, `accuracy = torchmetrics.Accuracy(task="multiclass", num_classes=10).to(device)` cria a **métrica de acurácia** para classificação multiclasse com 10 classes e a move para o mesmo **device** (CPU/GPU) do modelo. Por fim, `train2(model, optimizer, xentropy, accuracy, train_loader, valid_loader, n_epochs)` roda o laço de treino por `n_epochs`, usando a **perda** `xentropy` (CrossEntropyLoss) e medindo **acurácia** em treino/validação; o retorno é atribuído a `_` para indicar que não será usado.

```
In [92]: model.eval()
X_new, y_new = next(iter(valid_loader))
X_new = X_new[:3].to(device)
with torch.no_grad():
    y_pred_logits = model(X_new)
y_pred = y_pred_logits.argmax(dim=1) # index of the Largest Logit
y_pred
```

Out[92]: tensor([7, 4, 2])

`model.eval()` coloca o modelo em **modo avaliação** (desativa dropout e usa estatísticas fixas de BatchNorm). `X_new, y_new = next(iter(valid_loader))` pega **um batch** do validador; `X_new = X_new[:3].to(device)` mantém só as **3 primeiras amostras** e move para CPU/GPU. Com `torch.no_grad()` desligamos gradientes (mais rápido, sem ocupar memória) e calculamos `y_pred_logits = model(X_new)`, que são os **logits** (pontuações não normalizadas). Por fim, `y_pred =`

`y_pred_logits.argmax(dim=1)` escolhe, para cada amostra, o **índice da classe com maior logit** (não precisa `softmax` para isso), produzindo os rótulos previstos.

```
In [93]: [train_and_valid_data.classes[index] for index in y_pred]
```

```
Out[93]: ['Sneaker', 'Coat', 'Pullover']
```

Vamos verificar se o modelo fez as previsões corretas:

```
In [94]: y_new[:3]
```

```
Out[94]: tensor([7, 4, 2])
```

Perfeito! 😊

```
In [95]: import torch.nn.functional as F
```

```
y_proba = F.softmax(y_pred_logits, dim=1)
if device == "mps":
    y_proba = y_proba.cpu()
y_proba.round(decimals=3)
```

```
Out[95]: tensor([[0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0010, 0.0000, 0.9150, 0.000
0,
                 0.0840],
                [0.0000, 0.0000, 0.0030, 0.0000, 0.9970, 0.0000, 0.0000, 0.0000, 0.000
0,
                 0.0000],
                [0.0010, 0.0000, 0.7710, 0.0000, 0.0720, 0.0000, 0.1550, 0.0000, 0.000
0,
                 0.0000]])
```

Aplica **softmax** para transformar os **logits** em **probabilidades** por classe (cada linha soma 1), movendo para CPU se o device for **MPS** (GPU Apple) e arredondando para 3 casas:

- `F.softmax(y_pred_logits, dim=1)` : converte logits → probabilidades ao longo do eixo das classes (`dim=1`), mantendo shape `(batch, n_classes)`.
- `if device == "mps": y_proba = y_proba.cpu()` : traz o tensor para CPU (útil para exibir/usar com libs que não lidam com MPS).
- `y_proba.round(decimals=3)` : arredonda só para visualização (não use arredondado para cálculos).

```
In [96]: y_top4_values, y_top4_indices = torch.topk(y_pred_logits, k=4, dim=1)
y_top4_probas = F.softmax(y_top4_values, dim=1)
if device == "mps":
    y_top4_probas = y_top4_probas.cpu()
y_top4_probas.round(decimals=3)
```

```
Out[96]: tensor([[0.9150, 0.0840, 0.0010, 0.0000],
                  [0.9970, 0.0030, 0.0000, 0.0000],
                  [0.7720, 0.1550, 0.0720, 0.0010]])
```

Você está pegando o **Top-4 por amostra** e exibindo as probabilidades dessas 4 classes:

- `torch.topk(y_pred_logits, k=4, dim=1)` retorna **dois tensores**:
 - `y_top4_values` : os **4 maiores logits** por linha (amostra)
 - `y_top4_indices` : os **índices das classes** correspondentes
- `F.softmax(y_top4_values, dim=1)` aplica **softmax só nesses 4 logits**, gerando probabilidades **renormalizadas entre as Top-4** (a massa das demais classes é descartada).

Se você fizesse softmax nos **logits completos** e depois selecionasse as Top-4, as probabilidades poderiam ser levemente diferentes; aqui elas são reescaladas para somar 1 **apenas** entre as quatro escolhidas.

- `if device == "mps": y_top4_probas = y_top4_probas.cpu()` move para CPU (útil em Macs com GPU Apple).
- `.round(3)` apenas **arredonda para mostrar** (não use arredondado para cálculos).

Use `y_top4_indices` para mapear para os rótulos de classe (ex.: nomes do Fashion-MNIST).

In [97]: `y_top4_indices`

Out[97]: `tensor([[7, 9, 5, 8],
[4, 2, 6, 0],
[2, 6, 4, 0]])`

In [98]: `sum([param.numel() for param in model.parameters()])`

Out[98]: 266610

Isso retorna o **número total de parâmetros** do modelo.

- `model.parameters()` — iterador sobre **todos os nn.Parameter** do modelo (treináveis por padrão).
- `param.numel()` — quantidade de **elementos** naquele tensor de pesos/viés.
- `sum([...])` — soma tudo → **total de parâmetros** (treináveis + não-treináveis; se quiser só treináveis, filtre).

8. Fine-Tuning Neural Network Hyperparameter with Optuna

Além de definir manualmente os **hiperparâmetros** (como visto no Capítulo 9), é possível realizar uma **busca automática** para encontrar valores mais adequados.

Abordagens possíveis:

1. Converter o modelo PyTorch em um estimador Scikit-Learn:

- Criando uma classe customizada ou usando uma **wrapper library** como [Skorch](#).
- Permite usar métodos como **GridSearchCV** ou **RandomizedSearchCV** para ajustar hiperparâmetros.

2. Usar bibliotecas dedicadas de fine-tuning (geralmente mais eficientes):

- [Optuna](#)
- [Ray Tune](#)
- [Hyperopt](#)

Essas bibliotecas oferecem **estratégias poderosas de busca** e são altamente customizáveis.

Exemplo com Optuna:

1. Instalação (no Colab):

```
%pip install optuna
```

```
In [99]: !pip install optuna
```

```

Collecting optuna
  Downloading optuna-4.6.0-py3-none-any.whl.metadata (17 kB)
Collecting alembic>=1.5.0 (from optuna)
  Downloading alembic-1.17.2-py3-none-any.whl.metadata (7.2 kB)
Collecting colorlog (from optuna)
  Downloading colorlog-6.10.1-py3-none-any.whl.metadata (11 kB)
Requirement already satisfied: numpy in c:\users\estev\onedrive\área de trabalho\ncia\ncia\lib\site-packages (from optuna) (2.3.3)
Requirement already satisfied: packaging>=20.0 in c:\users\estev\onedrive\área de trabalho\ncia\ncia\lib\site-packages (from optuna) (25.0)
Collecting sqlalchemy>=1.4.2 (from optuna)
  Downloading sqlalchemy-2.0.44-cp313-cp313-win_amd64.whl.metadata (9.8 kB)
Requirement already satisfied: tqdm in c:\users\estev\onedrive\área de trabalho\ncia\ncia\lib\site-packages (from optuna) (4.67.1)
Requirement already satisfied: PyYAML in c:\users\estev\onedrive\área de trabalho\ncia\ncia\lib\site-packages (from optuna) (6.0.2)
Collecting Mako (from alembic>=1.5.0->optuna)
  Downloading mako-1.3.10-py3-none-any.whl.metadata (2.9 kB)
Requirement already satisfied: typing-extensions>=4.12 in c:\users\estev\onedrive\área de trabalho\ncia\ncia\lib\site-packages (from alembic>=1.5.0->optuna) (4.1.5.0)
Collecting greenlet>=1 (from sqlalchemy>=1.4.2->optuna)
  Downloading greenlet-3.2.4-cp313-cp313-win_amd64.whl.metadata (4.2 kB)
Requirement already satisfied: colorama in c:\users\estev\onedrive\área de trabalho\ncia\ncia\lib\site-packages (from colorlog->optuna) (0.4.6)
Requirement already satisfied: MarkupSafe>=0.9.2 in c:\users\estev\onedrive\área de trabalho\ncia\ncia\lib\site-packages (from Mako->alembic>=1.5.0->optuna) (3.0.2)
Downloading optuna-4.6.0-py3-none-any.whl (404 kB)
Downloading alembic-1.17.2-py3-none-any.whl (248 kB)
Downloading sqlalchemy-2.0.44-cp313-cp313-win_amd64.whl (2.1 MB)
----- 0.0/2.1 MB ? eta -:---:-
----- 0.3/2.1 MB ? eta -:---:-

```

```
----- 0.5/2.1 MB 216.5 kB/s eta 0:00:08
----- 0.8/2.1 MB 214.1 kB/s eta 0:00:07
----- 1.0/2.1 MB 160.0 kB/s eta 0:00:07
----- 1.3/2.1 MB 178.4 kB/s eta 0:00:05
----- 1.3/2.1 MB 178.4 kB/s eta 0:00:05
----- 1.3/2.1 MB 178.4 kB/s eta 0:00:05
----- 1.6/2.1 MB 200.0 kB/s eta 0:00:03
----- 1.8/2.1 MB 213.1 kB/s eta 0:00:02
----- 2.1/2.1 MB 182.1 kB/s eta 0:00:01
----- 2.1/2.1 MB 181.4 kB/s 0:00:15
```

Downloading greenlet-3.2.4-cp313-cp313-win_amd64.whl (299 kB)

Downloading colorlog-6.10.1-py3-none-any.whl (11 kB)

Downloading mako-1.3.10-py3-none-any.whl (78 kB)

Installing collected packages: Mako, greenlet, colorlog, sqlalchemy, alembic, optuna

```
----- 0/6 [Mako]
----- 1/6 [greenlet]
----- 1/6 [greenlet]
----- 1/6 [greenlet]
```



```

----- 4/6 [alembic]
----- 4/6 [optuna]
----- 5/6 [optuna]
----- 6/6 [optuna]

```

Successfully installed Mako-1.3.10 alembic-1.17.2 colorlog-6.10.1 greenlet-3.2.4 optuna-4.6.0 sqlalchemy-2.0.44

In [100...]

```

import optuna

def objective(trial):
    learning_rate = trial.suggest_float("learning_rate", 1e-5, 1e-1, log=True)
    n_hidden = trial.suggest_int("n_hidden", 20, 300)

    model = ImageClassifier(n_inputs=1 * 28 * 28, n_hidden1=n_hidden,
                           n_hidden2=n_hidden, n_classes=10).to(device)
    optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)
    xentropy = nn.CrossEntropyLoss()
    accuracy = torchmetrics.Accuracy(task="multiclass", num_classes=10)
    accuracy = accuracy.to(device)

    history = train2(model, optimizer, xentropy, accuracy, train_loader,
                     valid_loader, n_epochs=10)
    validation_accuracy = max(history["valid_metrics"])

```

```
    return validation_accuracy
```

1. Métodos para sugerir hiperparâmetros:

- `suggest_float()`: pede um valor float dentro de um intervalo.
- `suggest_int()`: pede um valor inteiro dentro de um intervalo.
- `suggest_categorical()`: pede um valor dentre opções categóricas.

Exemplo para `learning_rate`:

- Intervalo: `10^-5` a `10^-1`
- Uso de `log=True` para explorar múltiplas escalas (log distribution).
 - Sem isso, a amostragem uniforme poderia **ignorar valores muito pequenos**, prejudicando a busca.

2. Criando e rodando a busca:

1. Criar um **Study** com `optuna.create_study()`.
 - Definir `direction="maximize"` se a função objetivo retorna uma métrica que deve ser maximizada (por padrão, Optuna minimiza).
2. Rodar a otimização com `study.optimize()`, passando:
 - A **função objetivo** definida anteriormente.
 - O **número de trials** (quantas vezes Optuna chama a função objetivo).

3. Garantindo reproduzibilidade:

- Definir **random seed do PyTorch**.
- Definir **random seed do sampler do Optuna**.

Essa abordagem permite que Optuna explore de forma eficiente diferentes combinações de hiperparâmetros e encontre as melhores configurações para o modelo

In []:

```
torch.manual_seed(42)
sampler = optuna.samplers.TPESampler(seed=42)
study = optuna.create_study(direction="maximize", sampler=sampler)
study.optimize(objective, n_trials=5)
```

[I 2025-11-18 09:04:34,960] A new study created in memory with name: no-name-2b058bee-648c-48a4-8c36-78a0a0289afa

Epoch 1/10, train loss: 2.2769, train metric: 0.1471, valid metric: 0.1860
 Epoch 2/10, train loss: 2.2093, train metric: 0.2794, valid metric: 0.3500
 Epoch 3/10, train loss: 2.1164, train metric: 0.4109, valid metric: 0.4554
 Epoch 4/10, train loss: 1.9776, train metric: 0.5137, valid metric: 0.5560
 Epoch 5/10, train loss: 1.7867, train metric: 0.5826, valid metric: 0.6026
 Epoch 6/10, train loss: 1.5775, train metric: 0.6184, valid metric: 0.6228
 Epoch 7/10, train loss: 1.3978, train metric: 0.6288, valid metric: 0.6326
 Epoch 8/10, train loss: 1.2605, train metric: 0.6360, valid metric: 0.6372
 Epoch 9/10, train loss: 1.1572, train metric: 0.6467, valid metric: 0.6424

[I 2025-11-18 09:06:30,628] Trial 0 finished with value: 0.6435999870300293 and parameters: {'learning_rate': 0.00031489116479568613, 'n_hidden': 287}. Best is trial 0 with value: 0.6435999870300293.

```
Epoch 10/10, train loss: 1.0782, train metric: 0.6537, valid metric: 0.6436
Epoch 1/10, train loss: 1.1459, train metric: 0.6229, valid metric: 0.7338
Epoch 2/10, train loss: 0.6108, train metric: 0.7841, valid metric: 0.8000
```

Aumentar a acurácia ou reduzir a Loss?

- Se o **objetivo do produto** é "acertar mais", prefira **maximizar a métrica de validação** que representa esse objetivo (**acurácia, F1, AUC, QWK** em tarefas ordinais etc.). Você continuará treinando com **loss** (p/ gradiente), mas o **critério de escolha** no Optuna é a **métrica** que importa no fim.
- **Quando escolher minimizar loss?** Quando a métrica é **instável** em poucos dados (acurácia pulando muito), quando há **muito desbalanceamento** e você ainda não definiu limiar/ métrica final, ou em **regressão** (use **RMSE/MAE** como loss e como objetivo). A loss é **suave e estável**, costuma correlacionar com generalização.
- **Quando escolher maximizar acurácia (ou outra métrica)?** Quando a métrica já está **bem definida e alinhada ao negócio** (ex.: acurácia top-1 em classificação balanceada; **F1** em positivos raros; **AUC-ROC/PR** se você compara curvas; **QWK** em DR ordinal). A seleção por métrica garante que o melhor modelo é o que **de fato performa melhor** do jeito que você será avaliado.
- **Regra prática:** Treine **sempre** com uma loss apropriada (p.ex., CrossEntropy com pesos/focal em classe rara). No Optuna, **otimize a métrica de validação** que você reportará. Exceções: dados muito pequenos/ruidosos → minimize loss; regressão → minimize **RMSE/MAE**; ordinais (DR 0–4) → maximize **QWK**; desbalanceado severo → maximize **F1** ou **AUCPR**.
- **Dica final:** mesmo otimizando por métrica, monitore a **loss** (early stopping) e use a métrica como **critério de melhor checkpoint**.

1. Algoritmo padrão: TPE

- Optuna utiliza por padrão o **Tree-structured Parzen Estimator (TPE)** para otimização de hiperparâmetros.
- TPE é um **algoritmo de otimização sequencial baseado em modelo**:
 - Aprende com os **resultados anteriores** para selecionar melhores hiperparâmetros.
 - Inicialmente começa com valores **aleatórios**.
 - Progressivamente foca a busca nas **regiões mais promissoras** do espaço de hiperparâmetros.

2. Vantagens do TPE:

- Permite encontrar **hiperparâmetros melhores do que a busca aleatória (random search)** no mesmo tempo.
- Explora eficientemente o espaço de busca, evitando gastar muitas tentativas em regiões pouco promissoras.

3. Resultados da otimização:

- Após a execução do estudo (`study.optimize()`), é possível acessar:
 - **Melhores hiperparâmetros encontrados.**
 - **Acurácia de validação** ou outra métrica associada a esses hiperparâmetros.

Essa abordagem torna a busca de hiperparâmetros mais inteligente e eficiente, aproveitando os resultados de testes anteriores para melhorar continuamente a escolha dos próximos parâmetros a serem testados.

```
In [ ]: study.best_params
```

```
Out[ ]: {'learning_rate': 0.008471801418819975, 'n_hidden': 188}
```

```
In [ ]: study.best_value
```

```
Out[ ]: 0.8543999791145325
```

1. Número de trials

- A performance do modelo melhora à medida que aumentamos o número de trials (`n_trials`).
- Exemplo: aumentar `n_trials` para 50 ou mais geralmente gera resultados melhores, mas leva mais tempo para executar.
- Alternativa: rodar `optimize()` repetidamente e interromper quando o desempenho for satisfatório.

2. Paralelização de trials

- Optuna pode executar trials **em paralelo em múltiplas máquinas**, oferecendo um **aumento quase linear na velocidade**.
- Passos para paralelização:
 1. Configurar um **banco de dados SQL** (ex.: SQLite ou PostgreSQL).
 2. Definir o parâmetro `storage` na função `create_study()` apontando para esse banco de dados.
 3. Definir `study_name` e `load_if_exists=True`.
 4. Copiar o script de tuning para cada máquina e executar.
 - Se usar seeds aleatórios, garanta que sejam **diferentes em cada máquina**.
 5. Todos os scripts trabalham em paralelo, lendo e escrevendo resultados no banco de dados.
- Benefício adicional: mantém **um log completo de todos os resultados dos experimentos**.

3. Boas práticas com a função objetivo

- Em exemplos anteriores, a função `objective()` tinha acesso direto ao conjunto de treino e validação via variáveis globais.

- Forma mais limpa: **passar os datasets como argumentos adicionais** à função objetivo, garantindo melhor organização e modularidade do código:

```
In [ ]: def objective(trial, train_loader, valid_loader):
    learning_rate = trial.suggest_float("learning_rate", 1e-5, 1e-1, log=True)
    n_hidden = trial.suggest_int("n_hidden", 20, 300)

    model = ImageClassifier(n_inputs=1 * 28 * 28, n_hidden1=n_hidden,
                           n_hidden2=n_hidden, n_classes=10).to(device)

    optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)

    xentropy = nn.CrossEntropyLoss()
    accuracy = torchmetrics.Accuracy(task="multiclass", num_classes=10)

    accuracy = accuracy.to(device)

    best_validation_accuracy = 0.0
    for epoch in range(n_epochs):
        history = train2(model, optimizer, xentropy, accuracy, train_loader,
                          valid_loader, n_epochs=1)

        validation_accuracy = max(history["valid_metrics"])
        if validation_accuracy > best_validation_accuracy:
            best_validation_accuracy = validation_accuracy

        trial.report(validation_accuracy, epoch)
        if trial.should_prune():
            raise optuna.TrialPruned()
    return best_validation_accuracy
```

O que essa objective faz (bem direto): em cada *trial* do Optuna, ela amostra `learning_rate` (log-uniforme) e `n_hidden`, constrói o `ImageClassifier` (duas camadas ocultas iguais), cria `SGD` e `CrossEntropy`, define a **acurácia** do `torchmetrics` e treina por `n_epochs` **dentro** da própria `objective`. A cada época, chama `train2(..., n_epochs=1)`, lê a **acurácia de validação** (`validation_accuracy`), guarda o **melhor valor** em `best_validation_accuracy`, e informa progresso ao Optuna via `trial.report(validation_accuracy, epoch)`. Se o *pruner* decidir podar (`trial.should_prune()`), a *trial* é interrompida com `TrialPruned()`. No fim, retorna a **melhor acurácia de validação** do *trial* — por isso o seu `study` deve usar `direction="maximize"`.

Por que funciona: o Optuna compara o valor retornado (maior = melhor) entre *trials* com diferentes hiperparâmetros e converge para combinações promissoras, podando cedo as ruins.

Duas notas úteis:

- Garanta que `train2` **reseta** corretamente a métrica por época (ex.: `metric.reset()`), senão a acurácia pode acumular indevidamente.
- Para reproduzibilidade, fixe **seeds** (PyTorch e Optuna) e considere um *pruner* (ex.: `MedianPruner`) para acelerar.

```
In [ ]: objective_with_data = lambda trial: objective(
    trial, train_loader=train_loader, valid_loader=valid_loader)
```

Para definir os argumentos extras (os carregadores de conjuntos de dados, neste caso), basta criar uma função lambda quando necessário e passá-la para o método optimize(). Como alternativa, você pode usar a função functools.partial(), que cria uma função wrapper fina em torno do callable fornecido para fornecer valores padrão para qualquer número de argumentos:

```
In [ ]: from functools import partial

objective_with_data = partial(objective, train_loader=train_loader,
                               valid_loader=valid_loader)
```

```
In [ ]: torch.manual_seed(42)
sampler = optuna.samplers.TPESampler(seed=42)
pruner = optuna.pruners.MedianPruner()
study = optuna.create_study(direction="maximize", sampler=sampler,
                             pruner=pruner)
study.optimize(objective_with_data, n_trials=20)
```

[I 2025-11-03 16:05:40,310] A new study created in memory with name: no-name-9575f439-2bcc-4d90-8894-c6db784ec9b6

Epoch 1/1, train loss: 2.2769, train metric: 0.1471, valid metric: 0.1860
 Epoch 1/1, train loss: 2.2093, train metric: 0.2794, valid metric: 0.3500
 Epoch 1/1, train loss: 2.1164, train metric: 0.4110, valid metric: 0.4554
 Epoch 1/1, train loss: 1.9776, train metric: 0.5136, valid metric: 0.5560
 Epoch 1/1, train loss: 1.7867, train metric: 0.5826, valid metric: 0.6026
 Epoch 1/1, train loss: 1.5775, train metric: 0.6184, valid metric: 0.6228
 Epoch 1/1, train loss: 1.3978, train metric: 0.6288, valid metric: 0.6326
 Epoch 1/1, train loss: 1.2605, train metric: 0.6360, valid metric: 0.6372
 Epoch 1/1, train loss: 1.1572, train metric: 0.6468, valid metric: 0.6424
 Epoch 1/1, train loss: 1.0782, train metric: 0.6537, valid metric: 0.6436
 Epoch 1/1, train loss: 1.0162, train metric: 0.6611, valid metric: 0.6530
 Epoch 1/1, train loss: 0.9665, train metric: 0.6689, valid metric: 0.6620
 Epoch 1/1, train loss: 0.9258, train metric: 0.6761, valid metric: 0.6700
 Epoch 1/1, train loss: 0.8919, train metric: 0.6835, valid metric: 0.6782
 Epoch 1/1, train loss: 0.8629, train metric: 0.6897, valid metric: 0.6848
 Epoch 1/1, train loss: 0.8381, train metric: 0.6954, valid metric: 0.6876
 Epoch 1/1, train loss: 0.8166, train metric: 0.7009, valid metric: 0.6932
 Epoch 1/1, train loss: 0.7974, train metric: 0.7073, valid metric: 0.6964
 Epoch 1/1, train loss: 0.7802, train metric: 0.7119, valid metric: 0.7090

[I 2025-11-03 16:12:28,014] Trial 0 finished with value: 0.7089999914169312 and parameters: {'learning_rate': 0.00031489116479568613, 'n_hidden': 287}. Best is trial 0 with value: 0.7089999914169312.

```
Epoch 1/1, train loss: 0.7647, train metric: 0.7196, valid metric: 0.7082
Epoch 1/1, train loss: 1.1485, train metric: 0.6157, valid metric: 0.7330
Epoch 1/1, train loss: 0.6133, train metric: 0.7864, valid metric: 0.8082
Epoch 1/1, train loss: 0.5200, train metric: 0.8180, valid metric: 0.8134
Epoch 1/1, train loss: 0.4783, train metric: 0.8311, valid metric: 0.8232
Epoch 1/1, train loss: 0.4532, train metric: 0.8401, valid metric: 0.8026
Epoch 1/1, train loss: 0.4356, train metric: 0.8464, valid metric: 0.8448
Epoch 1/1, train loss: 0.4208, train metric: 0.8513, valid metric: 0.8274
Epoch 1/1, train loss: 0.4082, train metric: 0.8563, valid metric: 0.8396
Epoch 1/1, train loss: 0.3981, train metric: 0.8604, valid metric: 0.8520
Epoch 1/1, train loss: 0.3882, train metric: 0.8640, valid metric: 0.8586
Epoch 1/1, train loss: 0.3783, train metric: 0.8661, valid metric: 0.8540
Epoch 1/1, train loss: 0.3699, train metric: 0.8696, valid metric: 0.8576
Epoch 1/1, train loss: 0.3632, train metric: 0.8712, valid metric: 0.8568
Epoch 1/1, train loss: 0.3556, train metric: 0.8747, valid metric: 0.8470
Epoch 1/1, train loss: 0.3480, train metric: 0.8765, valid metric: 0.8678
Epoch 1/1, train loss: 0.3423, train metric: 0.8786, valid metric: 0.8526
Epoch 1/1, train loss: 0.3368, train metric: 0.8804, valid metric: 0.8684
Epoch 1/1, train loss: 0.3302, train metric: 0.8819, valid metric: 0.8666
Epoch 1/1, train loss: 0.3249, train metric: 0.8841, valid metric: 0.8670
```

```
[I 2025-11-03 16:19:47,898] Trial 1 finished with value: 0.868399977684021 and parameters: {'learning_rate': 0.008471801418819975, 'n_hidden': 188}. Best is trial 1 with value: 0.868399977684021.
```

```
Epoch 1/1, train loss: 0.3196, train metric: 0.8863, valid metric: 0.8628
Epoch 1/1, train loss: 2.2998, train metric: 0.1078, valid metric: 0.1152
Epoch 1/1, train loss: 2.2923, train metric: 0.1305, valid metric: 0.1432
Epoch 1/1, train loss: 2.2856, train metric: 0.1605, valid metric: 0.1704
Epoch 1/1, train loss: 2.2797, train metric: 0.1872, valid metric: 0.1912
Epoch 1/1, train loss: 2.2744, train metric: 0.2091, valid metric: 0.2114
Epoch 1/1, train loss: 2.2693, train metric: 0.2230, valid metric: 0.2216
Epoch 1/1, train loss: 2.2643, train metric: 0.2332, valid metric: 0.2320
Epoch 1/1, train loss: 2.2591, train metric: 0.2408, valid metric: 0.2380
Epoch 1/1, train loss: 2.2538, train metric: 0.2456, valid metric: 0.2424
Epoch 1/1, train loss: 2.2481, train metric: 0.2493, valid metric: 0.2456
Epoch 1/1, train loss: 2.2422, train metric: 0.2511, valid metric: 0.2464
Epoch 1/1, train loss: 2.2360, train metric: 0.2534, valid metric: 0.2476
Epoch 1/1, train loss: 2.2296, train metric: 0.2538, valid metric: 0.2478
Epoch 1/1, train loss: 2.2229, train metric: 0.2535, valid metric: 0.2482
Epoch 1/1, train loss: 2.2160, train metric: 0.2544, valid metric: 0.2492
Epoch 1/1, train loss: 2.2086, train metric: 0.2549, valid metric: 0.2508
Epoch 1/1, train loss: 2.2009, train metric: 0.2561, valid metric: 0.2522
Epoch 1/1, train loss: 2.1928, train metric: 0.2572, valid metric: 0.2520
Epoch 1/1, train loss: 2.1842, train metric: 0.2582, valid metric: 0.2526
```

```
[I 2025-11-03 16:26:37,422] Trial 2 finished with value: 0.25380000472068787 and parameters: {'learning_rate': 4.207988669606632e-05, 'n_hidden': 63}. Best is trial 1 with value: 0.868399977684021.
```

```
Epoch 1/1, train loss: 2.1751, train metric: 0.2610, valid metric: 0.2538
Epoch 1/1, train loss: 2.3015, train metric: 0.0997, valid metric: 0.1028
Epoch 1/1, train loss: 2.2984, train metric: 0.1009, valid metric: 0.1050
Epoch 1/1, train loss: 2.2953, train metric: 0.1051, valid metric: 0.1124
Epoch 1/1, train loss: 2.2923, train metric: 0.1160, valid metric: 0.1270
Epoch 1/1, train loss: 2.2894, train metric: 0.1347, valid metric: 0.1496
Epoch 1/1, train loss: 2.2865, train metric: 0.1548, valid metric: 0.1652
Epoch 1/1, train loss: 2.2837, train metric: 0.1699, valid metric: 0.1800
Epoch 1/1, train loss: 2.2808, train metric: 0.1800, valid metric: 0.1872
Epoch 1/1, train loss: 2.2780, train metric: 0.1855, valid metric: 0.1916
Epoch 1/1, train loss: 2.2752, train metric: 0.1893, valid metric: 0.1960
Epoch 1/1, train loss: 2.2725, train metric: 0.1950, valid metric: 0.2008
Epoch 1/1, train loss: 2.2697, train metric: 0.2000, valid metric: 0.2026
Epoch 1/1, train loss: 2.2669, train metric: 0.2058, valid metric: 0.2046
Epoch 1/1, train loss: 2.2641, train metric: 0.2124, valid metric: 0.2116
Epoch 1/1, train loss: 2.2613, train metric: 0.2177, valid metric: 0.2166
Epoch 1/1, train loss: 2.2585, train metric: 0.2238, valid metric: 0.2224
Epoch 1/1, train loss: 2.2556, train metric: 0.2289, valid metric: 0.2264
Epoch 1/1, train loss: 2.2528, train metric: 0.2339, valid metric: 0.2314
Epoch 1/1, train loss: 2.2498, train metric: 0.2383, valid metric: 0.2342
```

```
[I 2025-11-03 16:33:42,910] Trial 3 finished with value: 0.23960000276565552 and parameters: {'learning_rate': 1.7073967431528103e-05, 'n_hidden': 263}. Best is trial 1 with value: 0.868399977684021.
```

```
Epoch 1/1, train loss: 2.2469, train metric: 0.2429, valid metric: 0.2396
Epoch 1/1, train loss: 1.8945, train metric: 0.4924, valid metric: 0.6290
Epoch 1/1, train loss: 1.0016, train metric: 0.6582, valid metric: 0.6772
Epoch 1/1, train loss: 0.7747, train metric: 0.7115, valid metric: 0.7248
Epoch 1/1, train loss: 0.6864, train metric: 0.7551, valid metric: 0.7598
Epoch 1/1, train loss: 0.6268, train metric: 0.7833, valid metric: 0.7806
Epoch 1/1, train loss: 0.5830, train metric: 0.7999, valid metric: 0.7862
Epoch 1/1, train loss: 0.5500, train metric: 0.8113, valid metric: 0.8064
Epoch 1/1, train loss: 0.5252, train metric: 0.8185, valid metric: 0.8130
Epoch 1/1, train loss: 0.5061, train metric: 0.8238, valid metric: 0.8202
Epoch 1/1, train loss: 0.4908, train metric: 0.8291, valid metric: 0.8206
Epoch 1/1, train loss: 0.4777, train metric: 0.8337, valid metric: 0.8152
Epoch 1/1, train loss: 0.4681, train metric: 0.8375, valid metric: 0.8304
Epoch 1/1, train loss: 0.4591, train metric: 0.8403, valid metric: 0.8320
Epoch 1/1, train loss: 0.4516, train metric: 0.8430, valid metric: 0.8314
Epoch 1/1, train loss: 0.4447, train metric: 0.8460, valid metric: 0.8346
Epoch 1/1, train loss: 0.4380, train metric: 0.8474, valid metric: 0.8358
Epoch 1/1, train loss: 0.4321, train metric: 0.8496, valid metric: 0.8378
Epoch 1/1, train loss: 0.4272, train metric: 0.8512, valid metric: 0.8368
Epoch 1/1, train loss: 0.4223, train metric: 0.8528, valid metric: 0.8406
```

```
[I 2025-11-03 16:41:16,430] Trial 4 finished with value: 0.8432000279426575 and parameters: {'learning_rate': 0.002537815508265664, 'n_hidden': 218}. Best is trial 1 with value: 0.868399977684021.
```

```
Epoch 1/1, train loss: 0.4173, train metric: 0.8557, valid metric: 0.8432
```

```
[I 2025-11-03 16:41:36,596] Trial 5 pruned.
```

```
Epoch 1/1, train loss: 2.3017, train metric: 0.1007, valid metric: 0.1056
Epoch 1/1, train loss: 0.8584, train metric: 0.7025, valid metric: 0.7978
Epoch 1/1, train loss: 0.5067, train metric: 0.8211, valid metric: 0.8224
Epoch 1/1, train loss: 0.4498, train metric: 0.8405, valid metric: 0.8356
Epoch 1/1, train loss: 0.4182, train metric: 0.8512, valid metric: 0.8510
Epoch 1/1, train loss: 0.3934, train metric: 0.8577, valid metric: 0.8490
Epoch 1/1, train loss: 0.3753, train metric: 0.8661, valid metric: 0.8510
Epoch 1/1, train loss: 0.3601, train metric: 0.8700, valid metric: 0.8644
Epoch 1/1, train loss: 0.3474, train metric: 0.8746, valid metric: 0.8650
Epoch 1/1, train loss: 0.3357, train metric: 0.8780, valid metric: 0.8702
Epoch 1/1, train loss: 0.3263, train metric: 0.8822, valid metric: 0.8722
Epoch 1/1, train loss: 0.3179, train metric: 0.8838, valid metric: 0.8680
Epoch 1/1, train loss: 0.3093, train metric: 0.8871, valid metric: 0.8810
Epoch 1/1, train loss: 0.3016, train metric: 0.8900, valid metric: 0.8680
Epoch 1/1, train loss: 0.2944, train metric: 0.8921, valid metric: 0.8724
Epoch 1/1, train loss: 0.2883, train metric: 0.8945, valid metric: 0.8698
Epoch 1/1, train loss: 0.2820, train metric: 0.8967, valid metric: 0.8688
Epoch 1/1, train loss: 0.2757, train metric: 0.8994, valid metric: 0.8804
Epoch 1/1, train loss: 0.2696, train metric: 0.9014, valid metric: 0.8794
Epoch 1/1, train loss: 0.2640, train metric: 0.9041, valid metric: 0.8756
[I 2025-11-03 16:46:12,222] Trial 6 finished with value: 0.8853999972343445 and parameters: {'learning_rate': 0.021368329072358756, 'n_hidden': 79}. Best is trial 6 with value: 0.8853999972343445.
Epoch 1/1, train loss: 0.2596, train metric: 0.9043, valid metric: 0.8854
[I 2025-11-03 16:46:25,544] Trial 7 pruned.
Epoch 1/1, train loss: 2.2926, train metric: 0.1081, valid metric: 0.1064
[I 2025-11-03 16:46:38,763] Trial 8 pruned.
Epoch 1/1, train loss: 2.2836, train metric: 0.1225, valid metric: 0.1526
[I 2025-11-03 16:46:51,819] Trial 9 pruned.
Epoch 1/1, train loss: 2.2631, train metric: 0.2483, valid metric: 0.3554
Epoch 1/1, train loss: 0.6987, train metric: 0.7437, valid metric: 0.8334
Epoch 1/1, train loss: 0.4543, train metric: 0.8360, valid metric: 0.8288
Epoch 1/1, train loss: 0.4144, train metric: 0.8490, valid metric: 0.8488
Epoch 1/1, train loss: 0.3903, train metric: 0.8575, valid metric: 0.8354
Epoch 1/1, train loss: 0.3719, train metric: 0.8637, valid metric: 0.8454
Epoch 1/1, train loss: 0.3600, train metric: 0.8676, valid metric: 0.8538
Epoch 1/1, train loss: 0.3472, train metric: 0.8711, valid metric: 0.8590
Epoch 1/1, train loss: 0.3400, train metric: 0.8742, valid metric: 0.8546
Epoch 1/1, train loss: 0.3329, train metric: 0.8778, valid metric: 0.8538
Epoch 1/1, train loss: 0.3269, train metric: 0.8790, valid metric: 0.8622
Epoch 1/1, train loss: 0.3217, train metric: 0.8817, valid metric: 0.8616
Epoch 1/1, train loss: 0.3180, train metric: 0.8830, valid metric: 0.8676
Epoch 1/1, train loss: 0.3116, train metric: 0.8848, valid metric: 0.8654
Epoch 1/1, train loss: 0.3094, train metric: 0.8852, valid metric: 0.8508
Epoch 1/1, train loss: 0.3051, train metric: 0.8869, valid metric: 0.8634
Epoch 1/1, train loss: 0.3015, train metric: 0.8883, valid metric: 0.8604
Epoch 1/1, train loss: 0.2993, train metric: 0.8903, valid metric: 0.8706
Epoch 1/1, train loss: 0.2952, train metric: 0.8915, valid metric: 0.8642
Epoch 1/1, train loss: 0.2911, train metric: 0.8916, valid metric: 0.8716
[I 2025-11-03 16:51:29,477] Trial 10 finished with value: 0.8715999722480774 and parameters: {'learning_rate': 0.08165528450509137, 'n_hidden': 21}. Best is trial 6 with value: 0.8853999972343445.
```

```
Epoch 1/1, train loss: 0.2902, train metric: 0.8917, valid metric: 0.8628
Epoch 1/1, train loss: 0.6527, train metric: 0.7611, valid metric: 0.8132
Epoch 1/1, train loss: 0.4509, train metric: 0.8355, valid metric: 0.8452
Epoch 1/1, train loss: 0.4133, train metric: 0.8498, valid metric: 0.8440
Epoch 1/1, train loss: 0.3901, train metric: 0.8582, valid metric: 0.8530
Epoch 1/1, train loss: 0.3744, train metric: 0.8631, valid metric: 0.8478
Epoch 1/1, train loss: 0.3625, train metric: 0.8668, valid metric: 0.8596
Epoch 1/1, train loss: 0.3531, train metric: 0.8704, valid metric: 0.8520
Epoch 1/1, train loss: 0.3470, train metric: 0.8731, valid metric: 0.8592
Epoch 1/1, train loss: 0.3380, train metric: 0.8778, valid metric: 0.8568
Epoch 1/1, train loss: 0.3297, train metric: 0.8785, valid metric: 0.8586
Epoch 1/1, train loss: 0.3267, train metric: 0.8809, valid metric: 0.8564
Epoch 1/1, train loss: 0.3205, train metric: 0.8809, valid metric: 0.8586
Epoch 1/1, train loss: 0.3187, train metric: 0.8836, valid metric: 0.8636
Epoch 1/1, train loss: 0.3119, train metric: 0.8844, valid metric: 0.8594
Epoch 1/1, train loss: 0.3087, train metric: 0.8858, valid metric: 0.8634
Epoch 1/1, train loss: 0.3061, train metric: 0.8865, valid metric: 0.8612
Epoch 1/1, train loss: 0.3013, train metric: 0.8878, valid metric: 0.8446
Epoch 1/1, train loss: 0.2971, train metric: 0.8900, valid metric: 0.8576
Epoch 1/1, train loss: 0.2949, train metric: 0.8903, valid metric: 0.8590
```

```
[I 2025-11-03 16:55:59,506] Trial 11 finished with value: 0.8636000156402588 and
parameters: {'learning_rate': 0.07553503645583182, 'n_hidden': 21}. Best is trial
6 with value: 0.8853999972343445.
```

```
Epoch 1/1, train loss: 0.2925, train metric: 0.8914, valid metric: 0.8584
Epoch 1/1, train loss: 0.6195, train metric: 0.7764, valid metric: 0.8170
Epoch 1/1, train loss: 0.4182, train metric: 0.8470, valid metric: 0.8502
Epoch 1/1, train loss: 0.3726, train metric: 0.8631, valid metric: 0.8438
Epoch 1/1, train loss: 0.3465, train metric: 0.8723, valid metric: 0.8656
Epoch 1/1, train loss: 0.3273, train metric: 0.8787, valid metric: 0.8706
Epoch 1/1, train loss: 0.3116, train metric: 0.8839, valid metric: 0.8756
Epoch 1/1, train loss: 0.2984, train metric: 0.8887, valid metric: 0.8746
Epoch 1/1, train loss: 0.2866, train metric: 0.8917, valid metric: 0.8750
Epoch 1/1, train loss: 0.2754, train metric: 0.8982, valid metric: 0.8502
Epoch 1/1, train loss: 0.2685, train metric: 0.8987, valid metric: 0.8794
Epoch 1/1, train loss: 0.2585, train metric: 0.9040, valid metric: 0.8788
Epoch 1/1, train loss: 0.2498, train metric: 0.9068, valid metric: 0.8786
Epoch 1/1, train loss: 0.2430, train metric: 0.9085, valid metric: 0.8790
Epoch 1/1, train loss: 0.2360, train metric: 0.9114, valid metric: 0.8896
Epoch 1/1, train loss: 0.2302, train metric: 0.9132, valid metric: 0.8856
Epoch 1/1, train loss: 0.2243, train metric: 0.9145, valid metric: 0.8778
Epoch 1/1, train loss: 0.2178, train metric: 0.9171, valid metric: 0.8880
Epoch 1/1, train loss: 0.2136, train metric: 0.9197, valid metric: 0.8810
Epoch 1/1, train loss: 0.2080, train metric: 0.9222, valid metric: 0.8832
```

```
[I 2025-11-03 17:00:11,859] Trial 12 finished with value: 0.8895999789237976 and
parameters: {'learning_rate': 0.08525846269447779, 'n_hidden': 116}. Best is tria
l 12 with value: 0.8895999789237976.
```

```
Epoch 1/1, train loss: 0.2030, train metric: 0.9234, valid metric: 0.8598
Epoch 1/1, train loss: 0.8659, train metric: 0.7035, valid metric: 0.7852
Epoch 1/1, train loss: 0.5120, train metric: 0.8185, valid metric: 0.8192
Epoch 1/1, train loss: 0.4574, train metric: 0.8377, valid metric: 0.8268
Epoch 1/1, train loss: 0.4265, train metric: 0.8485, valid metric: 0.8456
Epoch 1/1, train loss: 0.4036, train metric: 0.8564, valid metric: 0.8468
Epoch 1/1, train loss: 0.3874, train metric: 0.8619, valid metric: 0.8440
Epoch 1/1, train loss: 0.3711, train metric: 0.8670, valid metric: 0.8304
Epoch 1/1, train loss: 0.3581, train metric: 0.8709, valid metric: 0.8574
Epoch 1/1, train loss: 0.3455, train metric: 0.8762, valid metric: 0.8662
Epoch 1/1, train loss: 0.3345, train metric: 0.8794, valid metric: 0.8688
Epoch 1/1, train loss: 0.3256, train metric: 0.8822, valid metric: 0.8658
Epoch 1/1, train loss: 0.3166, train metric: 0.8860, valid metric: 0.8756
Epoch 1/1, train loss: 0.3084, train metric: 0.8880, valid metric: 0.8746
Epoch 1/1, train loss: 0.3016, train metric: 0.8909, valid metric: 0.8814
Epoch 1/1, train loss: 0.2943, train metric: 0.8933, valid metric: 0.8742
Epoch 1/1, train loss: 0.2879, train metric: 0.8954, valid metric: 0.8738
Epoch 1/1, train loss: 0.2818, train metric: 0.8980, valid metric: 0.8712
Epoch 1/1, train loss: 0.2767, train metric: 0.8991, valid metric: 0.8714
Epoch 1/1, train loss: 0.2726, train metric: 0.9003, valid metric: 0.8680
```

```
[I 2025-11-03 17:04:49,365] Trial 13 finished with value: 0.8813999891281128 and
parameters: {'learning_rate': 0.01891149541864801, 'n_hidden': 116}. Best is tria
l 12 with value: 0.8895999789237976.
```

```
Epoch 1/1, train loss: 0.2670, train metric: 0.9022, valid metric: 0.8772
Epoch 1/1, train loss: 0.9440, train metric: 0.6748, valid metric: 0.7728
Epoch 1/1, train loss: 0.5292, train metric: 0.8132, valid metric: 0.8264
Epoch 1/1, train loss: 0.4687, train metric: 0.8343, valid metric: 0.8310
Epoch 1/1, train loss: 0.4347, train metric: 0.8449, valid metric: 0.8280
Epoch 1/1, train loss: 0.4103, train metric: 0.8558, valid metric: 0.8444
Epoch 1/1, train loss: 0.3914, train metric: 0.8613, valid metric: 0.8294
Epoch 1/1, train loss: 0.3759, train metric: 0.8672, valid metric: 0.8608
Epoch 1/1, train loss: 0.3615, train metric: 0.8711, valid metric: 0.8498
Epoch 1/1, train loss: 0.3492, train metric: 0.8755, valid metric: 0.8600
Epoch 1/1, train loss: 0.3373, train metric: 0.8776, valid metric: 0.8692
Epoch 1/1, train loss: 0.3293, train metric: 0.8806, valid metric: 0.8614
Epoch 1/1, train loss: 0.3203, train metric: 0.8847, valid metric: 0.8632
Epoch 1/1, train loss: 0.3116, train metric: 0.8879, valid metric: 0.8724
Epoch 1/1, train loss: 0.3052, train metric: 0.8897, valid metric: 0.8646
Epoch 1/1, train loss: 0.2975, train metric: 0.8924, valid metric: 0.8764
Epoch 1/1, train loss: 0.2929, train metric: 0.8940, valid metric: 0.8740
Epoch 1/1, train loss: 0.2857, train metric: 0.8947, valid metric: 0.8756
Epoch 1/1, train loss: 0.2800, train metric: 0.8978, valid metric: 0.8754
Epoch 1/1, train loss: 0.2748, train metric: 0.8985, valid metric: 0.8796
```

```
[I 2025-11-03 17:09:18,298] Trial 14 finished with value: 0.8830000162124634 and
parameters: {'learning_rate': 0.01635101118825921, 'n_hidden': 127}. Best is tria
l 12 with value: 0.8895999789237976.
```

```
Epoch 1/1, train loss: 0.2703, train metric: 0.9011, valid metric: 0.8830
```

```
[I 2025-11-03 17:09:32,252] Trial 15 pruned.
```

```
Epoch 1/1, train loss: 1.7748, train metric: 0.4454, valid metric: 0.6304
Epoch 1/1, train loss: 0.7612, train metric: 0.7270, valid metric: 0.7950
Epoch 1/1, train loss: 0.4729, train metric: 0.8316, valid metric: 0.8404
Epoch 1/1, train loss: 0.4165, train metric: 0.8509, valid metric: 0.8558
Epoch 1/1, train loss: 0.3841, train metric: 0.8613, valid metric: 0.8618
Epoch 1/1, train loss: 0.3609, train metric: 0.8702, valid metric: 0.8586
Epoch 1/1, train loss: 0.3436, train metric: 0.8745, valid metric: 0.8556
Epoch 1/1, train loss: 0.3294, train metric: 0.8793, valid metric: 0.8720
Epoch 1/1, train loss: 0.3177, train metric: 0.8833, valid metric: 0.8756
Epoch 1/1, train loss: 0.3046, train metric: 0.8876, valid metric: 0.8776
Epoch 1/1, train loss: 0.2960, train metric: 0.8909, valid metric: 0.8570
Epoch 1/1, train loss: 0.2878, train metric: 0.8926, valid metric: 0.8774
Epoch 1/1, train loss: 0.2787, train metric: 0.8981, valid metric: 0.8740
Epoch 1/1, train loss: 0.2712, train metric: 0.8997, valid metric: 0.8770
Epoch 1/1, train loss: 0.2631, train metric: 0.9030, valid metric: 0.8760
Epoch 1/1, train loss: 0.2565, train metric: 0.9044, valid metric: 0.8812
Epoch 1/1, train loss: 0.2518, train metric: 0.9063, valid metric: 0.8780
Epoch 1/1, train loss: 0.2451, train metric: 0.9089, valid metric: 0.8832
Epoch 1/1, train loss: 0.2403, train metric: 0.9110, valid metric: 0.8744
Epoch 1/1, train loss: 0.2352, train metric: 0.9120, valid metric: 0.8810
[I 2025-11-03 17:14:04,071] Trial 16 finished with value: 0.8844000101089478 and
parameters: {'learning_rate': 0.03266629913173288, 'n_hidden': 142}. Best is tria
l 12 with value: 0.8895999789237976.
Epoch 1/1, train loss: 0.2284, train metric: 0.9146, valid metric: 0.8844
[I 2025-11-03 17:14:17,683] Trial 17 pruned.
Epoch 1/1, train loss: 1.5867, train metric: 0.5013, valid metric: 0.6500
Epoch 1/1, train loss: 0.6298, train metric: 0.7719, valid metric: 0.8186
Epoch 1/1, train loss: 0.4236, train metric: 0.8436, valid metric: 0.8508
Epoch 1/1, train loss: 0.3816, train metric: 0.8600, valid metric: 0.8430
Epoch 1/1, train loss: 0.3555, train metric: 0.8697, valid metric: 0.8578
Epoch 1/1, train loss: 0.3372, train metric: 0.8752, valid metric: 0.8616
Epoch 1/1, train loss: 0.3236, train metric: 0.8804, valid metric: 0.8754
Epoch 1/1, train loss: 0.3109, train metric: 0.8847, valid metric: 0.8420
Epoch 1/1, train loss: 0.3003, train metric: 0.8877, valid metric: 0.8658
Epoch 1/1, train loss: 0.2924, train metric: 0.8906, valid metric: 0.8740
Epoch 1/1, train loss: 0.2840, train metric: 0.8949, valid metric: 0.8700
Epoch 1/1, train loss: 0.2771, train metric: 0.8961, valid metric: 0.8710
Epoch 1/1, train loss: 0.2686, train metric: 0.8996, valid metric: 0.8658
Epoch 1/1, train loss: 0.2638, train metric: 0.9019, valid metric: 0.8832
Epoch 1/1, train loss: 0.2591, train metric: 0.9022, valid metric: 0.8724
Epoch 1/1, train loss: 0.2543, train metric: 0.9041, valid metric: 0.8846
Epoch 1/1, train loss: 0.2496, train metric: 0.9056, valid metric: 0.8794
Epoch 1/1, train loss: 0.2432, train metric: 0.9070, valid metric: 0.8782
Epoch 1/1, train loss: 0.2401, train metric: 0.9084, valid metric: 0.8702
Epoch 1/1, train loss: 0.2362, train metric: 0.9095, valid metric: 0.8824
[I 2025-11-03 17:18:39,348] Trial 18 finished with value: 0.8845999836921692 and
parameters: {'learning_rate': 0.09548128419071349, 'n_hidden': 50}. Best is trial
12 with value: 0.8895999789237976.
```

```

Epoch 1/1, train loss: 0.2342, train metric: 0.9102, valid metric: 0.8770
Epoch 1/1, train loss: 0.7349, train metric: 0.7418, valid metric: 0.8136
Epoch 1/1, train loss: 0.4612, train metric: 0.8354, valid metric: 0.8404
Epoch 1/1, train loss: 0.4101, train metric: 0.8515, valid metric: 0.8504
Epoch 1/1, train loss: 0.3796, train metric: 0.8632, valid metric: 0.8580
Epoch 1/1, train loss: 0.3577, train metric: 0.8698, valid metric: 0.8696
Epoch 1/1, train loss: 0.3404, train metric: 0.8755, valid metric: 0.8622
Epoch 1/1, train loss: 0.3252, train metric: 0.8815, valid metric: 0.8696
Epoch 1/1, train loss: 0.3130, train metric: 0.8866, valid metric: 0.8732
Epoch 1/1, train loss: 0.3011, train metric: 0.8889, valid metric: 0.8716
Epoch 1/1, train loss: 0.2907, train metric: 0.8931, valid metric: 0.8540
Epoch 1/1, train loss: 0.2828, train metric: 0.8953, valid metric: 0.8818
Epoch 1/1, train loss: 0.2723, train metric: 0.9006, valid metric: 0.8756
Epoch 1/1, train loss: 0.2664, train metric: 0.9024, valid metric: 0.8736
Epoch 1/1, train loss: 0.2583, train metric: 0.9047, valid metric: 0.8782
Epoch 1/1, train loss: 0.2522, train metric: 0.9069, valid metric: 0.8698
Epoch 1/1, train loss: 0.2456, train metric: 0.9093, valid metric: 0.8800
Epoch 1/1, train loss: 0.2395, train metric: 0.9105, valid metric: 0.8790
Epoch 1/1, train loss: 0.2348, train metric: 0.9130, valid metric: 0.8820
Epoch 1/1, train loss: 0.2289, train metric: 0.9147, valid metric: 0.8804
[I 2025-11-03 17:23:03,848] Trial 19 finished with value: 0.8823999762535095 and
parameters: {'learning_rate': 0.035092247138920854, 'n_hidden': 156}. Best is tri
al 12 with value: 0.8895999789237976.
Epoch 1/1, train loss: 0.2229, train metric: 0.9175, valid metric: 0.8824

```

- `partial(objective, train_loader=..., valid_loader=...)` cria uma **versão da função** `objective` já “pré-preenchida” com os `loaders` — assim `study.optimize(...)` pode chamá-la **sem argumentos extras** (só o `trial`).
- `torch.manual_seed(42)` e `TPESampler(seed=42)` fixam **seeds** para reproduzibilidade (mesmas amostragens de hiperparâmetros e mesmos pesos iniciais).
- `MedianPruner()` ativa **poda de trials**: se o desempenho parcial do trial ficar **abaixo da mediana** dos anteriores no mesmo passo (época), o Optuna **interrompe cedo** esse trial, economizando tempo.
- `create_study(direction="maximize", sampler=..., pruner=...)` cria o estudo para **maximizar** a métrica retornada pela `objective` (a sua é acurácia de validação).
- `study.optimize(objective_with_data, n_trials=20)` roda **20 trials**, cada um testando combinações diferentes (via **TPE**), com poda automática pelo pruner.

```
In [ ]: study.best_value
```

```
Out[ ]: 0.8867999911308289
```

```
In [ ]: study.best_params
```

```
Out[ ]: {'learning_rate': 0.08525846269447772, 'n_hidden': 116}
```

9. Saving and Loading a PyTorch Model

- **Método mais simples:**

Utilizar `torch.save()` passando o **modelo** e o **caminho do arquivo**.

- **Como funciona:**

1. O objeto do modelo é **serializado** usando o módulo `pickle` do Python, que converte objetos em uma sequência de bytes.
2. Em seguida, o resultado é **compactado (zip)** e salvo no disco.

- **Convenção de arquivos:**

- Extensões comuns: `.pt` ou `.pth`.

```
In [ ]: torch.save(model, "my_fashion_mnist.pt")
```

Definir `weights_only=False` garante que todo o objeto do modelo seja carregado, e não apenas os parâmetros do modelo. Assim, você pode usar o modelo carregado para inferência. Não se esqueça de alternar para o modo de avaliação primeiro usando o método `eval()`:

```
In [ ]: loaded_model = torch.load("my_fashion_mnist.pt", weights_only=False)
```

```
In [ ]: loaded_model.eval()
y_pred_logits = loaded_model(X_new)
```

Embora salvar o modelo completo com `torch.save()` seja fácil e prático, essa abordagem apresenta **problemas sérios**:

1. Segurança:

- O formato de serialização do `pickle` é **notoriamente inseguro**.
- Embora `torch.save()` não salve código personalizado diretamente, o formato pickle suporta isso.
- Um modelo malicioso pode conter **código que será executado automaticamente** ao carregar o modelo.
- **Regra importante:** só carregue modelos de fontes confiáveis.

2. Fragilidade e compatibilidade:

- Pickle pode ser **susceptível a mudanças entre versões do Python** (ex.: grandes mudanças entre Python 3.7 e 3.8).
- Ele salva caminhos de arquivos específicos do código, o que pode **quebrar o carregamento** se a estrutura de pastas for diferente.

Recomendação:

Para evitar esses problemas, é **melhor salvar e carregar apenas os pesos do modelo**, em vez de salvar o objeto completo. Isso mantém a segurança e aumenta a compatibilidade entre diferentes ambientes e versões do Python.

```
In [ ]: torch.save(model.state_dict(), "my_fashion_mnist_weights.pt")
```

```
In [ ]: type(model.state_dict())
```

```
Out[ ]: collections.OrderedDict
```

Para aumentar a segurança e compatibilidade ao salvar modelos PyTorch, é recomendado salvar **apenas os pesos** do modelo, usando o **state dictionary**.

O que é `state_dict()`:

- Retorna um **OrderedDict do Python** contendo:
 - Cada **parâmetro do modelo** (retornado por `named_parameters()`).
 - **Buffers**, se houver, que são tensores registrados via `register_buffer()`.
 - Buffers armazenam dados adicionais necessários ao modelo, mas **não são parâmetros treináveis** (ex.: estatísticas de batch-norm).

Como carregar apenas os pesos:

1. Crie um modelo com a **mesma estrutura** do modelo original.
2. Carregue os pesos usando `torch.load()` com `weights_only=True`.
3. Aplique os pesos ao modelo com `load_state_dict()`.
4. Coloque o modelo em **modo de avaliação** antes da inferência.

```
# Criação do modelo com a mesma arquitetura
model = MyModel()

# Carrega apenas os pesos
weights = torch.load("meus_pesos.pth", weights_only=True)

# Aplica os pesos ao modelo
model.load_state_dict(weights)

# Coloca o modelo em modo de avaliação
model.eval()
```

```
In [ ]: new_model = ImageClassifier(n_inputs=1 * 28 * 28, n_hidden1=300, n_hidden2=100,
                                 n_classes=10)
loaded_weights = torch.load("my_fashion_mnist_weights.pt", weights_only=True)
new_model.load_state_dict(loaded_weights)
new_model.eval()
```

```
Out[ ]: ImageClassifier(
    (mlp): Sequential(
        (0): Flatten(start_dim=1, end_dim=-1)
        (1): Linear(in_features=784, out_features=300, bias=True)
        (2): ReLU()
        (3): Linear(in_features=300, out_features=100, bias=True)
        (4): ReLU()
        (5): Linear(in_features=100, out_features=10, bias=True)
    )
)
```

```
In [ ]: model_data = {
    "model_state_dict": model.state_dict(),
    "model_hyperparameters": {
        "n_inputs": 1 * 28 * 28,
        "n_hidden1": 300,
        "n_hidden2": 100,
        "n_classes": 10,
    }
}
torch.save(model_data, "my_fashion_mnist_model.pt")
```

```
In [ ]: loaded_data = torch.load("my_fashion_mnist_model.pt", weights_only=True)
new_model = ImageClassifier(**loaded_data["model_hyperparameters"])
new_model.load_state_dict(loaded_data["model_state_dict"])
new_model.eval()
```

```
Out[ ]: ImageClassifier(
    (mlp): Sequential(
        (0): Flatten(start_dim=1, end_dim=-1)
        (1): Linear(in_features=784, out_features=300, bias=True)
        (2): ReLU()
        (3): Linear(in_features=300, out_features=100, bias=True)
        (4): ReLU()
        (5): Linear(in_features=100, out_features=10, bias=True)
    )
)
```

10. Compiling and Optimizing a PyTorch Model

O PyTorch possui um recurso poderoso chamado **TorchScript**, que permite transformar o código do modelo em uma versão **estaticamente tipada** do Python.

Benefícios principais:

1. Otimização e desempenho

- TorchScript pode ser **compilado e otimizado**, tornando o modelo mais rápido.
- Exemplos de otimizações:
 - **Fusão de operações**: várias operações podem ser combinadas em uma única operação mais eficiente.
 - **Constant folding**: operações com constantes (ex.: `2 * 3`) são substituídas pelo resultado (`6`).
 - Código não utilizado pode ser removido (pruning).

2. Portabilidade e serialização

- Modelos podem ser **salvos em disco** e carregados tanto em **Python** quanto em **C++** usando a biblioteca **LibTorch**.
- Possibilita a execução em **diversos dispositivos**, incluindo embarcados.

Formas de conversão:

- **Tracing:** O PyTorch executa o modelo com **dados de exemplo**, registra todas as operações e converte esse registro em TorchScript.
 - Implementação: usando a função `torch.jit.trace()`.

```
In [ ]: torchscript_model = torch.jit.trace(model, X_new)
```

oi

O **tracing** funciona bem para modelos **estáticos**, cujo método `forward()` não utiliza **condicionais ou loops**.

- Problemas do tracing em modelos dinâmicos:
 - Se o modelo contém `if` ou `match`, apenas a **ramificação executada** será capturada.
 - Se o modelo contém **loops**, o TorchScript registrará **uma cópia das operações para cada iteração executada**, o que geralmente não é desejado.

Alternativa: Scripting

- O PyTorch **analisa diretamente o código Python** e converte em TorchScript.
- Suporta:
 - `if` e `while` (condições baseadas em tensores).
 - `for` ao iterar sobre tensores.
- Limitações:
 - Apenas um **subconjunto de Python** é suportado:
 - Não pode usar **variáveis globais, generators (yield), list comprehensions complexas, argumentos de função variáveis (*args ou `kwargs`)****, ou `** match**`.
 - Tipos devem ser **fixos** (uma função não pode retornar inteiro em alguns casos e float em outros).
 - Chamadas a funções externas só funcionam se também respeitarem essas regras (sem bibliotecas padrão ou de terceiros).

Apesar das restrições, para a maioria dos modelos reais, é relativamente fácil seguir essas regras.

Modelos podem ser **salvos e reutilizados** usando scripting.

```
In [ ]: torchscript_model = torch.jit.script(model)
```

Independentemente de você ter usado **tracing** ou **scripting** para criar o modelo TorchScript, é possível realizar **otimizações adicionais** para melhorar desempenho e eficiência.

Essas otimizações podem incluir:

- **Fusão de operações adicionais.**
- **Remoção de código não utilizado** (pruning).

- **Constant folding** mais avançado.

Essas melhorias ajudam a acelerar a execução do modelo e a reduzir o consumo de recursos.

```
In [ ]: optimized_model = torch.jit.optimize_for_inference(torchscript_model)
```

- **Finalidade:**

Modelos convertidos para **TorchScript** são destinados **apenas para inferência** (predições).

- **Limitações:**

- Não suportam **rastreio de gradiente** (gradient tracking).
- Não permitem **atualização de parâmetros** durante treinamento.

- **Salvamento de modelos:**

Um modelo TorchScript pode ser salvo usando o método `save()`:

```
In [ ]: optimized_model.save("my_fashion_mnist_torchscript.pt")
```

E então carregue-o usando a função `torch.jit.load()`:

```
In [ ]: loaded_torchscript_model = torch.jit.load("my_fashion_mnist_torchscript.pt")
```

```
In [ ]: y_pred_logits = loaded_torchscript_model(X_new)
y_pred_logits
```

```
Out[ ]: tensor([[ -3.0775,   -0.4287,   -3.8184,   -2.1958,   -2.9602,    1.5626,   -3.2479,
                  9.0668,   -0.9979,    6.8095],
                 [ -0.6670,   -2.8818,   12.1115,   -0.6558,   17.0885,  -13.5468,    8.1562,
                  -12.5179,   -0.4020,   -8.0469],
                 [ -0.7411,   -0.5282,    5.4406,   -0.2547,    3.4654,   -4.5936,    3.5897,
                  -3.1546,    0.3394,   -3.2801]], device='cuda:0')
```

- **TorchScript** não está mais em desenvolvimento ativo:

- Correções de bugs ainda ocorrem, mas **nenhuma nova funcionalidade** é adicionada.
- Continua sendo útil para **executar modelos PyTorch em C++**.
- Com o **PyTorch 2.0 (março de 2023)**, o foco mudou para **novas ferramentas de compilação** baseadas na função `torch.compile()`, que são:
 - Fáceis de usar
 - Mais eficientes
 - Melhor adaptadas para modelos dinâmicos e otimizações automáticas

```
In [ ]: compiled_model = torch.compile(model)
```

```
In [ ]: if device == "cuda":
    y_pred_logits = compiled_model(X_new)
```

```
W1007 08:38:10.662000 360 torch/_inductor/utils.py:1436] [0/0] Not enough SMs to
use max_autotune_gemm mode
```

Após criar um modelo TorchScript, ele pode ser usado **normalmente para inferência** e será **compilado e otimizado automaticamente**.

- Isso é conhecido como **Just-In-Time (JIT) compilation**, em contraste com **Ahead-Of-Time (AOT)**.
- No PyTorch 2.x, `torch.compile()` utiliza internamente:
 - **TorchDynamo**: captura o grafo de computação do modelo a partir do bytecode Python, permitindo lidar corretamente com **condicionais** e **loops**.
 - **TorchInductor**: realiza a compilação e otimização real, gerando código altamente eficiente:
 - GPU Nvidia: usando a linguagem **Triton**
 - CPU: usando **OpenMP**
 - Outros backends: por exemplo, **XLA** para TPUs (configurável com `device="xla"`).

Com essas ferramentas, você pode construir e treinar **redes neurais complexas e eficientes**.

Próximos Passos no Aprendizado com PyTorch

Nos próximos capítulos, você aprenderá a trabalhar com:

- **Redes muito profundas**
- **Arquiteturas populares**:
 - CNNs para imagens
 - RNNs para dados sequenciais
 - Transformers para texto
 - Autoencoders para representação
 - GANs e modelos de difusão para geração de dados
- **Reinforcement learning** para agentes autônomos
- **Deploy e otimização de modelos PyTorch**

