Aprendizado Profundo

Frameworks de Desenvolvimento de Redes Neurais - PyTorch

Professor: Lucas Silveira Kupssinskü

Agenda

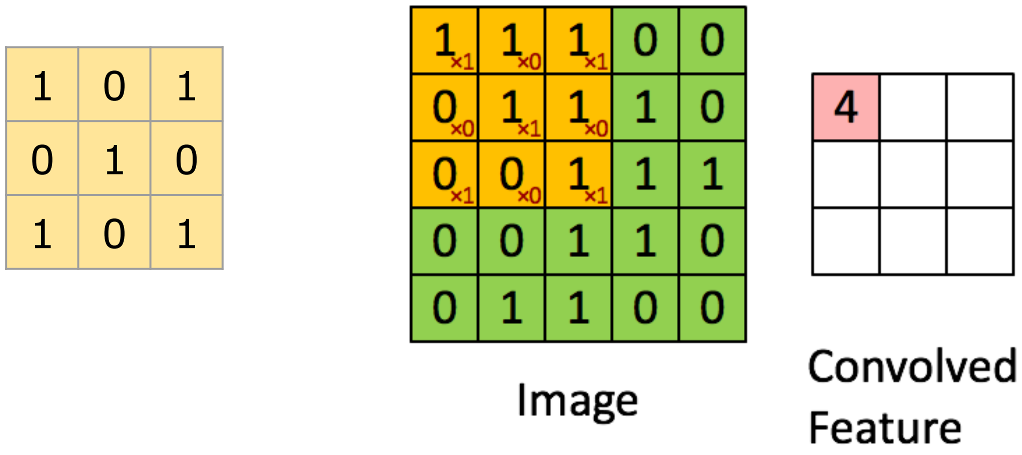
- Redes Convolucionais
  - Camadas Convolucionais
  - Camadas de Pooling
- De Jupyter Notebook para *script* de treinamento
- Utilitários monitorar o *hardware* durante o experimento

Redes Convolucionais

- São redes que incluem operações de convolução
  - Usadas principalmente em imagens
- Criadas em 1989(!) por Yann LeCun (LeNet - 5)
  - LECUN, Yann et al. Handwritten digit recognition with a back-propagation network. Advances in neural information processing systems, v. 2, 1989.
- Explodiram em popularidade pós 2012
  - AlexNet
    - KRIZHEVSKY, Alex; SUTSKEVER, Ilya; HINTON, Geoffrey E. Imagenet classification with deep convolutional neural networks. Advances in neural information processing systems, v. 25, 2012.
    - Treinamento em GPUs
    - +Dados
    - +Poder Computacional

Redes Convolucionais

- Antigas conhecidas nossas
  - Tamanho do Kernel
  - Número de filtros
  - *Stride*
  - *Padding*
  - *Dilation*



Camadas Convolucionais

- Temos diversas camadas convolucionais disponíveis no PyTorch

Convolution Layers

`nn.Conv1d`	Applies a 1D convolution over an input signal composed of several input planes.
`nn.Conv2d`	Applies a 2D convolution over an input signal composed of several input planes.
`nn.Conv3d`	Applies a 3D convolution over an input signal composed of several input planes.
`nn.ConvTranspose1d`	Applies a 1D transposed convolution operator over an input image composed of several input planes.
`nn.ConvTranspose2d`	Applies a 2D transposed convolution operator over an input image composed of several input planes.
`nn.ConvTranspose3d`	Applies a 3D transposed convolution operator over an input image composed of several input planes.

https://pytorch.org/docs/stable/nn.html

Camadas Convolucionais

CLASS `torch.nn.Conv2d(in_channels, out_channels, kernel_size, stride=1, padding=0, dilation=1, groups=1, bias=True, padding_mode='zeros', device=None, dtype=None)` [SOURCE]

Applies a 2D convolution over an input signal composed of several input planes.

In the simplest case, the output value of the layer with input size (N, C_{in}, H, W) and output $(N, C_{\text{out}}, H_{\text{out}}, W_{\text{out}})$ can be precisely described as:

$$\text{out}(N_i, C_{\text{out}_j}) = \text{bias}(C_{\text{out}_j}) + \sum_{k=0}^{C_{\text{in}}-1} \text{weight}(C_{\text{out}_j}, k) \star \text{input}(N_i, k)$$

where $\star$ is the valid 2D cross-correlation operator, N is a batch size, C denotes a number of channels, H is a height of input planes in pixels, and W is width in pixels.

https://pytorch.org/docs/stable/nn.html

Camadas de Pooling

`nn.MaxPool1d`	Applies a 1D max pooling over an input signal composed of several input planes.
`nn.MaxPool2d`	Applies a 2D max pooling over an input signal composed of several input planes.
`nn.MaxPool3d`	Applies a 3D max pooling over an input signal composed of several input planes.
`nn.MaxUnpool1d`	Computes a partial inverse of `MaxPool1d`.
`nn.MaxUnpool2d`	Computes a partial inverse of `MaxPool2d`.
`nn.MaxUnpool3d`	Computes a partial inverse of `MaxPool3d`.
`nn.AvgPool1d`	Applies a 1D average pooling over an input signal composed of several input planes.
`nn.AvgPool2d`	Applies a 2D average pooling over an input signal composed of several input planes.

https://pytorch.org/docs/stable/nn.html

Camadas de Pooling

CLASS torch.nn.MaxPool2d(*kernel_size*, *stride=None*, *padding=0*, *dilation=1*, *return_indices=False*, *ceil_mode=False*) [SOURCE]

Applies a 2D max pooling over an input signal composed of several input planes.

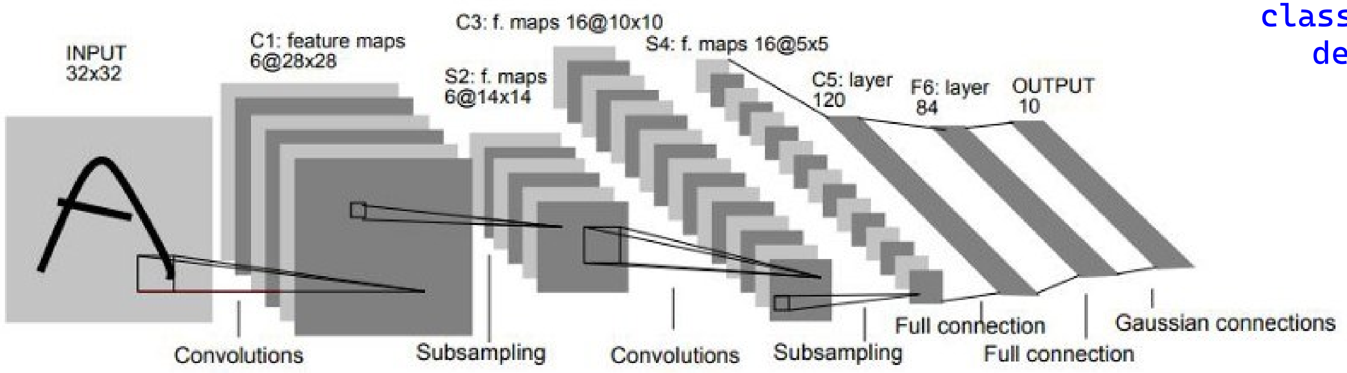
In the simplest case, the output value of the layer with input size (N, C, H, W), output (N, C, H_{out}, W_{out}) and `kernel_size` (kH, kW) can be precisely described as:

$$out(N_i, C_j, h, w) = \max_{m=0,\ldots,kH-1} \max_{n=0,\ldots,kW-1} \text{input}(N_i, C_j, \text{stride}[0] \times h + m, \text{stride}[1] \times w + n)$$

If `padding` is non-zero, then the input is implicitly padded with negative infinity on both sides for `padding` number of points. `dilation` controls the spacing between the kernel points. It is harder to describe, but this link has a nice visualization of what `dilation` does.

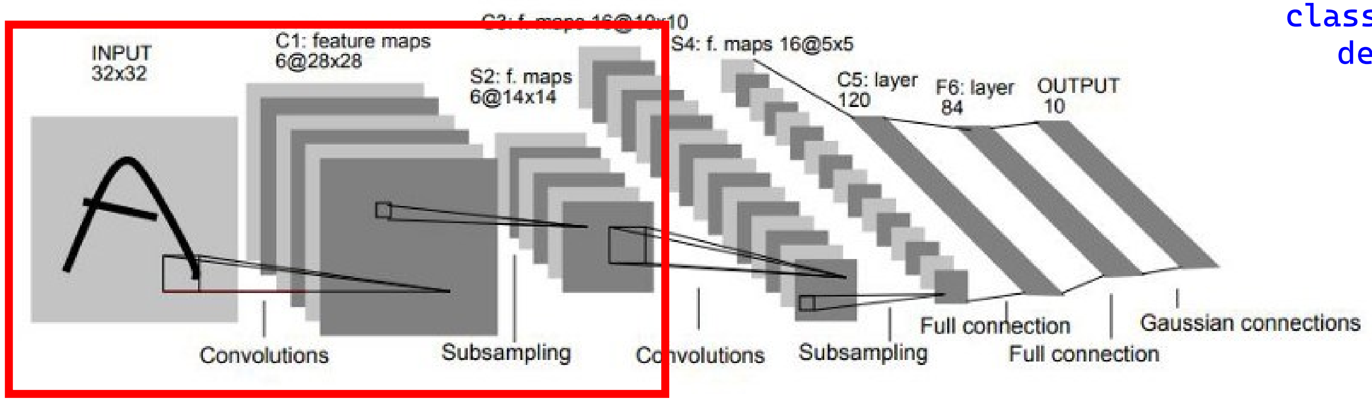
https://pytorch.org/docs/stable/nn.html

LeNet-5



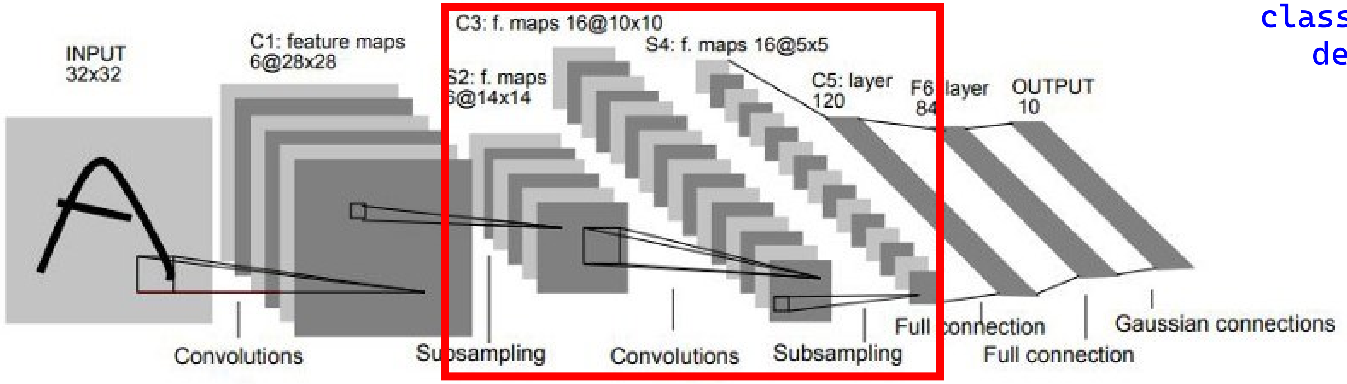

```python
class LeNet5(torch.nn.Module):
   def __init__(self):
      super(LeNet5, self).__init__()
      self.conv1 = torch.nn.Conv2d(
         in_channels=1, out_channels=6, kernel_size=5, padding=2)
      self.conv2 = torch.nn.Conv2d(
         in_channels=6, out_channels=16, kernel_size=5)
      self.fc1 = torch.nn.Linear(16*5*5, 120)
      self.fc2 = torch.nn.Linear(120, 84)
      self.fc3 = torch.nn.Linear(84, 10)

   def forward(self, x):
      x = torch.max_pool2d(torch.relu(self.conv1(x)), 2)
      x = torch.max_pool2d(torch.relu(self.conv2(x)), 2)
      x = torch.flatten(x, 1)
      x = torch.relu(self.fc1(x))
      x = torch.relu(self.fc2(x))
      x = self.fc3(x)
      return x
```

LeNet-5



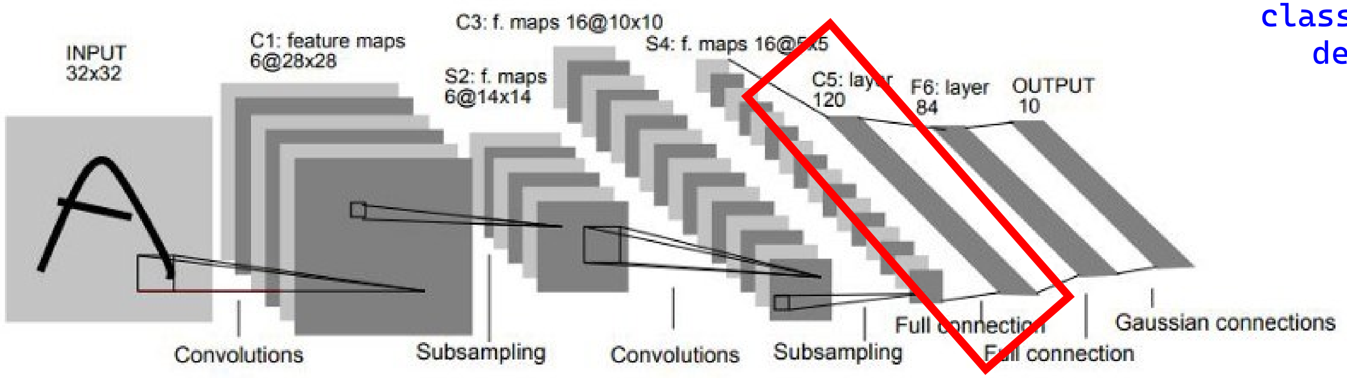

```python
class LeNet5(torch.nn.Module):
    def __init__(self):
        super(LeNet5, self).__init__()
        self.conv1 = torch.nn.Conv2d(
            in_channels=1, out_channels=6, kernel_size=5, padding=2)
        self.conv2 = torch.nn.Conv2d(
            in_channels=6, out_channels=16, kernel_size=5)
        self.fc1 = torch.nn.Linear(16*5*5, 120)
        self.fc2 = torch.nn.Linear(120, 84)
        self.fc3 = torch.nn.Linear(84, 10)

    def forward(self, x):
        x = torch.max_pool2d(torch.relu(self.conv1(x)), 2)
        x = torch.max_pool2d(torch.relu(self.conv2(x)), 2)
        x = torch.flatten(x, 1)
        x = torch.relu(self.fc1(x))
        x = torch.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

LeNet-5



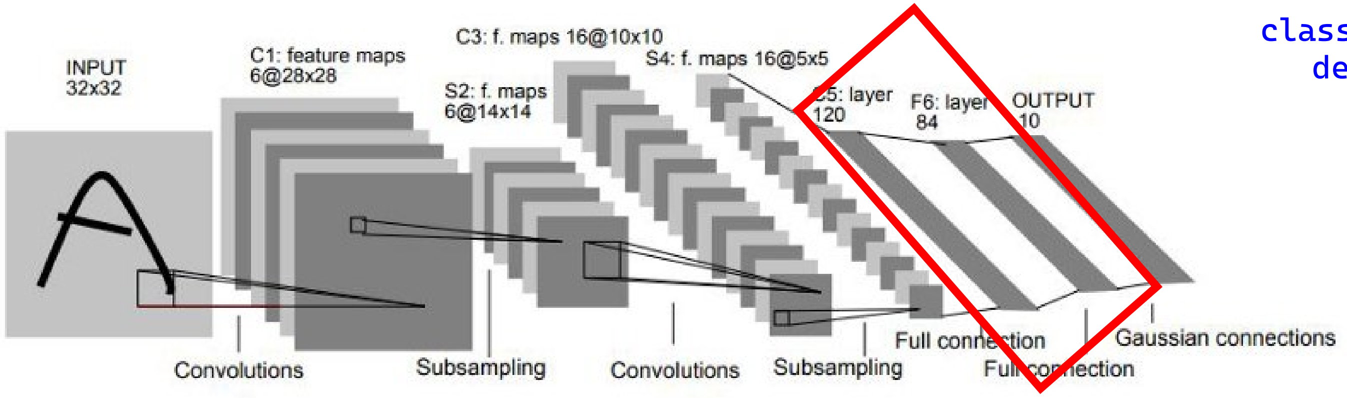

```python
class LeNet5(torch.nn.Module):
   def __init__(self):
      super(LeNet5, self).__init__()
      self.conv1 = torch.nn.Conv2d(
         in_channels=1, out_channels=6, kernel_size=5, padding=2)
      self.conv2 = torch.nn.Conv2d(
         in_channels=6, out_channels=16, kernel_size=5)
      self.fc1 = torch.nn.Linear(16*5*5, 120)
      self.fc2 = torch.nn.Linear(120, 84)
      self.fc3 = torch.nn.Linear(84, 10)

   def forward(self, x):
      x = torch.max_pool2d(torch.relu(self.conv1(x)), 2)
      x = torch.max_pool2d(torch.relu(self.conv2(x)), 2)
      x = torch.flatten(x, 1)
      x = torch.relu(self.fc1(x))
      x = torch.relu(self.fc2(x))
      x = self.fc3(x)
      return x
```

LeNet-5



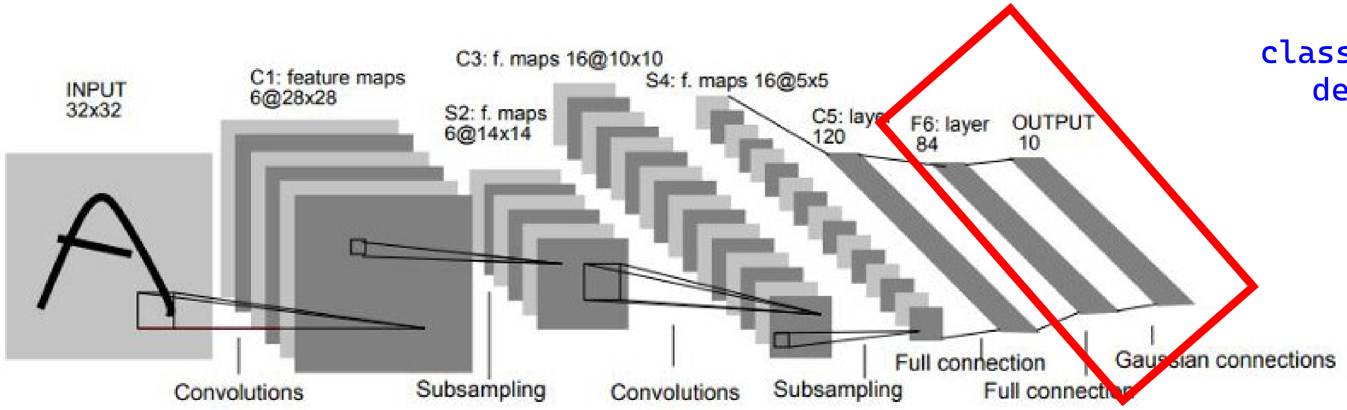

```python
class LeNet5(torch.nn.Module):
    def __init__(self):
        super(LeNet5, self).__init__()
        self.conv1 = torch.nn.Conv2d(
            in_channels=1, out_channels=6, kernel_size=5, padding=2)
        self.conv2 = torch.nn.Conv2d(
            in_channels=6, out_channels=16, kernel_size=5)
        self.fc1 = torch.nn.Linear(16*5*5, 120)
        self.fc2 = torch.nn.Linear(120, 84)
        self.fc3 = torch.nn.Linear(84, 10)

    def forward(self, x):
        x = torch.max_pool2d(torch.relu(self.conv1(x)), 2)
        x = torch.max_pool2d(torch.relu(self.conv2(x)), 2)
        x = torch.flatten(x, 1)
        x = torch.relu(self.fc1(x))
        x = torch.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

LeNet-5




```python
class LeNet5(torch.nn.Module):
   def __init__(self):
      super(LeNet5, self).__init__()
      self.conv1 = torch.nn.Conv2d(
         in_channels=1, out_channels=6, kernel_size=5, padding=2)
      self.conv2 = torch.nn.Conv2d(
         in_channels=6, out_channels=16, kernel_size=5)
      self.fc1 = torch.nn.Linear(16*5*5, 120)
      self.fc2 = torch.nn.Linear(120, 84)
      self.fc3 = torch.nn.Linear(84, 10)

   def forward(self, x):
      x = torch.max_pool2d(torch.relu(self.conv1(x)), 2)
      x = torch.max_pool2d(torch.relu(self.conv2(x)), 2)
      x = torch.flatten(x, 1)
      x = torch.relu(self.fc1(x))
      x = torch.relu(self.fc2(x))
      x = self.fc3(x)
      return x
```

LeNet-5




```
class LeNet5(torch.nn.Module):
    def __init__(self):
        super(LeNet5, self).__init__()
        self.conv1 = torch.nn.Conv2d(
            in_channels=1, out_channels=6, kernel_size=5, padding=2)
        self.conv2 = torch.nn.Conv2d(
            in_channels=6, out_channels=16, kernel_size=5)
        self.fc1 = torch.nn.Linear(16*5*5, 120)
        self.fc2 = torch.nn.Linear(120, 84)
        self.fc3 = torch.nn.Linear(84, 10)

    def forward(self, x):
        x = torch.max_pool2d(torch.relu(self.conv1(x)), 2)
        x = torch.max_pool2d(torch.relu(self.conv2(x)), 2)
        x = torch.flatten(x, 1)
        x = torch.relu(self.fc1(x))
        x = torch.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

LeNet-5

- Essa implementação difere da LeNet-5 original:
  - Função de ativação tanh
  - *Average Pooling*

```python
class LeNet5(torch.nn.Module):
  def __init__(self):
    super(LeNet5, self).__init__()
    self.conv1 = torch.nn.Conv2d(
      in_channels=1, out_channels=6, kernel_size=5, padding=2)
    self.conv2 = torch.nn.Conv2d(
      in_channels=6, out_channels=16, kernel_size=5)
    self.fc1 = torch.nn.Linear(16*5*5, 120)
    self.fc2 = torch.nn.Linear(120, 84)
    self.fc3 = torch.nn.Linear(84, 10)

  def forward(self, x):
    x = torch.max_pool2d(torch.relu(self.conv1(x)), 2)
    x = torch.max_pool2d(torch.relu(self.conv2(x)), 2)
    x = torch.flatten(x, 1)
    x = torch.relu(self.fc1(x))
    x = torch.relu(self.fc2(x))
    x = self.fc3(x)
    return x
```

LeNet-5

- Podemos encapsular partes da rede em um `nn.Sequential`

```
class LeNet5(torch.nn.Module):
  def __init__(self):
    super(LeNet5, self).__init__()
    self.feature_extractor = torch.nn.Sequential(
      torch.nn.Conv2d(
        in_channels=1, out_channels=6, kernel_size=5, padding=2),
      torch.nn.ReLU(),
      torch.nn.MaxPool2d(kernel_size=2, stride=2),
      torch.nn.Conv2d(
        in_channels=6, out_channels=16, kernel_size=5),
      torch.nn.ReLU(),
      torch.nn.MaxPool2d(kernel_size=2, stride=2)
    )
    self.classifier = torch.nn.Sequential(
      torch.nn.Flatten(1),
      torch.nn.Linear(16*5*5, 120),
      torch.nn.ReLU(),
      torch.nn.Linear(120, 84),
      torch.nn.ReLU(),
      torch.nn.Linear(84, 10)
    )

  def forward(self, x):
    x = self.feature_extractor(x)
    x = self.classifier(x)
    return x
```

MNIST

- Vamos usar a `torchvision.datasets` para carregar os dados
- Essa classe facilita o download de alguns conjuntos de dados bastante utilizados em problemas de visão computacional

CLASS `torchvision.datasets.MNIST(root: str, train: bool = True, transform: Union[Callable, NoneType] = None, target_transform: Union[Callable, NoneType] = None, download: bool = False)` → None [SOURCE]

MNIST Dataset.

Parameters:

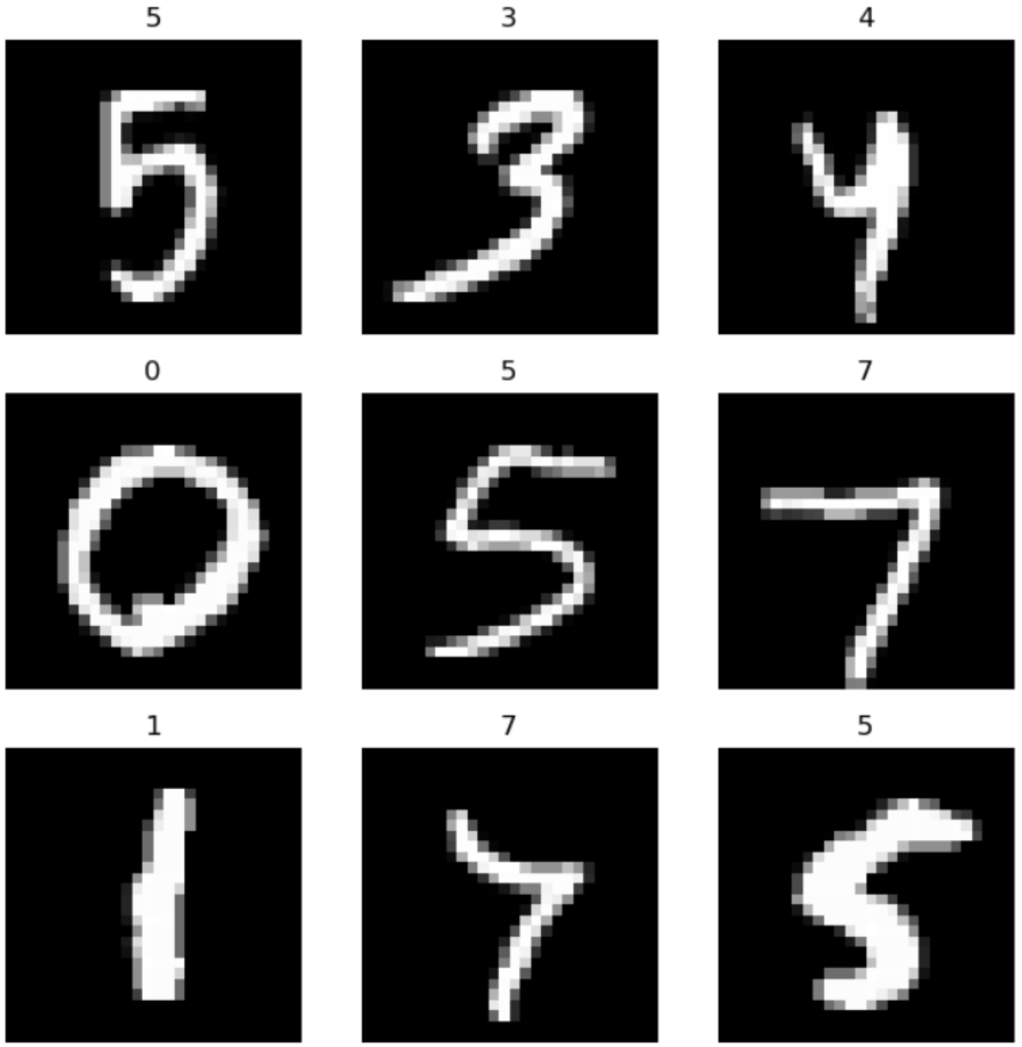
- **root** (*string*) – Root directory of dataset where `MNIST/processed/training.pt` and `MNIST/processed/test.pt` exist.
- **train** (*bool*, *optional*) – If True, creates dataset from `training.pt`, otherwise from `test.pt`.
- **download** (*bool*, *optional*) – If true, downloads the dataset from the internet and puts it in root directory. If dataset is already downloaded, it is not downloaded again.
- **transform** (*callable*, *optional*) – A function/transform that takes in an PIL image and returns a transformed version. E.g, `transforms.RandomCrop`
- **target_transform** (*callable*, *optional*) – A function/transform that takes in the target and transforms it.

torchvision.datasets
CelebA
CIFAR
Cityscapes
+ COCO
DatasetFolder
EMNIST
FakeData
Fashion-MNIST
Flickr
HMDB51
ImageFolder
ImageNet
Kinetics-400
KMNIST
LSUN
MNIST
Omniglot
PhotoTour
Places365
QMNIST
SBD
SBU
STL10
SVHN
UCF101
USPS
VOC

MNIST

```
training_data = datasets.MNIST(
   root="data",
   train=True,
   download=True,
   transform=ToTensor(),
)
test_data = datasets.MNIST(
   root="data",
   train=False,
   download=True,
   transform=ToTensor(),
)
```

```
figure = plt.figure(figsize=(8, 8))
cols, rows = 3, 3
for i in range(1, cols * rows + 1):
   sample_idx = torch.randint(len(training_data),
size=(1,)).item()
   img, label = training_data[sample_idx]
   figure.add_subplot(rows, cols, i)
   plt.title(label)
   plt.axis("off")
   plt.imshow(img.squeeze(), cmap="gray")
plt.show()
```

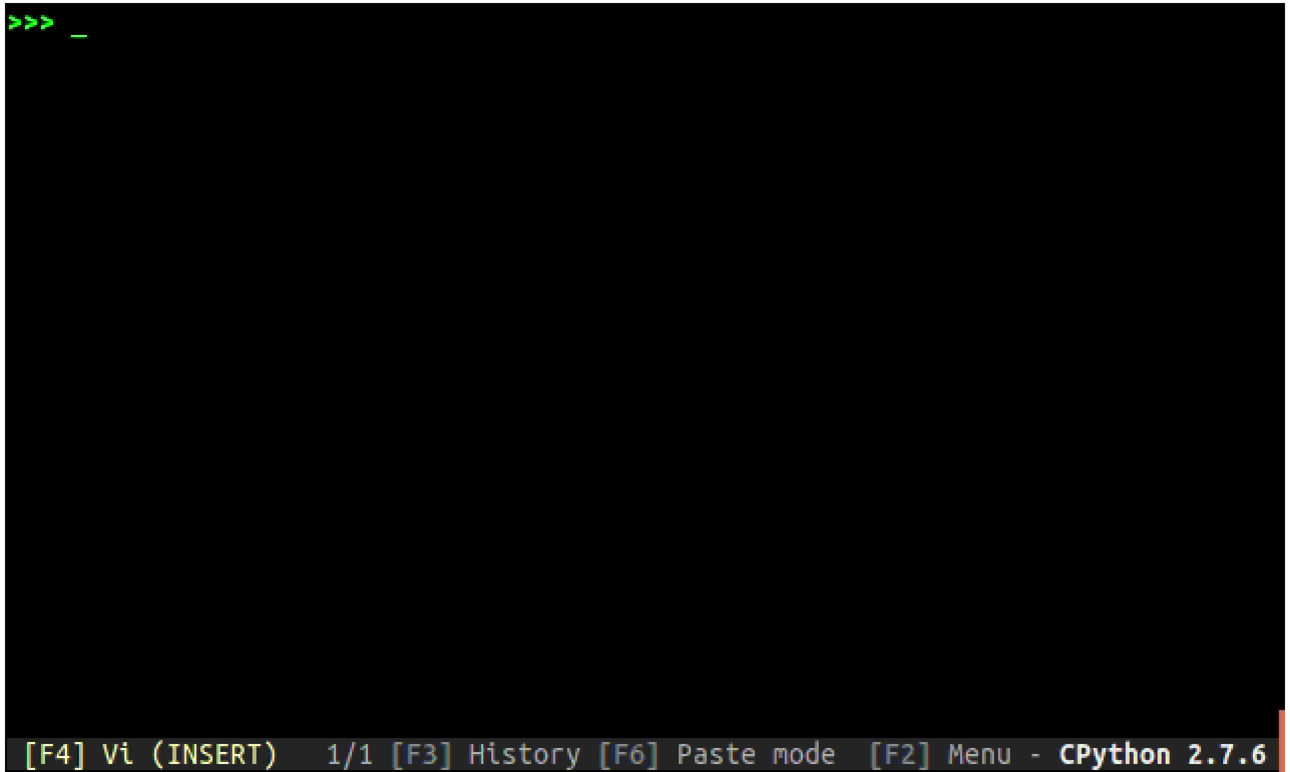


Loop de Treino e Validação

```
loss_fn = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(params=lenet5.parameters(), lr=0.1)

torch.manual_seed(42)
train_time_start_on_cpu = timer()

for epoch in range(EPOCHS):
   with tqdm(train_dataloader, desc=f'{epoch=}', unit='batch') as tqdm_epoch:
      train_loss = train_step(lenet5, tqdm_epoch, loss_fn, optimizer)
      test_loss, test_acc = test_step(lenet5, loss_fn, test_dataloader, device)

   print(f'Train loss: {train_loss:.5f} | Test loss: {test_loss:.5f}, Test acc: {test_acc*100:.4f}%')

train_time_end_on_cpu = timer()
total_train_time_model_0 = print_train_time(start=train_time_start_on_cpu,
                                            end=train_time_end_on_cpu,
                                            device=str(next(lenet5.parameters()).device))
```

tqdm

- É usual criar barras de processamento durante o treinamento e validação de redes neurais
- Um pacote python muito utilizado para isso é o tqdm



https://tqdm.github.io/

Loop de Treino e Validação

```
def train_step(model, dataloader, loss_fn, optimizer, device: torch.device):
   train_loss = 0
   # Faz loop em todos os dados de treino
   for X, y in dataloader:
      X, y = X.to(device), y.to(device)
      model.train()
      # 1. Forward pass
      y_pred = model(X)
      # 2. Calcula loss por batch
      loss = loss_fn(y_pred, y)
      train_loss += loss
      # 3. zera gradientes anteriores
      optimizer.zero_grad()
      # 4. Backward Pass
      loss.backward()
      # 5. Otimizacao
      optimizer.step()
   train_loss /= len(dataloader)
   return train_loss
```

Loop de Treino e Validação

```
def test_step(model:torch.nn.Module, loss_fn:torch.nn.Module, dataloader:torch.utils.data.DataLoader,
device:torch.device):
   test_loss, test_acc = 0, 0
   model.eval()
   with torch.inference_mode():
      for X, y in dataloader:
         X, y = X.to(device), y.to(device)
         # 1. Forward pass
         test_pred = model(X)

         # 2. Loss
         test_loss += loss_fn(test_pred, y) # accumulatively add up the loss per epoch

         # 3. Computa a acuracia
         test_acc += accuracy(target=y,
                              preds=torch.softmax(test_pred,dim=1),
                              task='multiclass',
                              num_classes=10)

      test_loss /= len(dataloader)
      test_acc /= len(dataloader)
      return test_loss, test_acc
```

Salvando os pesos da rede

- Após o processo de treinamento (ou em algum checkpoint relevante), podemos gravar os pesos (e/ou a arquitetura) da nossa rede
- Usamos `torch.save` e `torch.load` para fazer a serialização e a desserialização da rede neural
  - Usa o módulo pickle

Dessa forma salvamos apenas os pesos

```
torch.save(lenet5.state_dict(), 'lenet5.pth')
```

```
model1 = LeNet5()
model1.load_state_dict(torch.load('lenet5.pth'))
```

Dessa forma salvamos pesos e arquitetura

```
torch.save(lenet5, 'lenet5_model.pth')
```

```
model2 = torch.load('lenet5_model.pth')
```

Jupyter Notebook vs Script

- Embora Jupyter Notebooks (*.ipynb) sejam mais interativos e fáceis para desenvolvimento, usualmente queremos a flexibilidade e facilidade de execução proporcionada por um script (*.py)
- É comum começarmos com jupyter e migrarmos para scripts

Jupyter Notebook vs Script

- Quando portamos nosso código para script precisamos tomar alguns cuidados
  - Modularizar o código separando em scripts diferentes cada uma das funcionalidades
    - `model.py`
    - `data_setup.py`
    - `train.py`
    - `eval.py`
  - Fornecer argumentos nos scripts para parâmetros que podem ser modificados em diferentes execuções do treinamento (argparse ou typedargs são soluções interessantes)
  - Evitar deixar códigos “soltos” dentro do script

Jupyter Notebook vs Script

- Uma chamada para um script de treinamento de uma rede neural pode ser parecido com o seguinte

Target Python script | **How big should the batch size be?** | **Train for how long?**

```
python train.py —model MODEL_NAME —batch_size BATCH_SIZE —lr LEARNING_RATE —num_epochs NUM_EPOCHS
```

Model to train | **What should the learning rate be?**

```
python train.py —model tinyvgg —batch_size 32 —lr 0.001 —num_epochs 10
```

"Train the TinyVGG model with a batch size of 32 and a learning rate of 0.001 for 10 epochs."

Jupyter Notebook vs Script

```
if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument('--batch_size', '-b', type=int, default=64,
                        help='batch size for training')
    parser.add_argument('--epochs', '-e', type=int, default=6,
                        help='Training Epochs')
    parser.add_argument('--num_workers', '-w', type=int, default=2,
                        help='Number of workers for dataloader')
    parser.add_argument('--force_cpu', '-c', type=bool, default=False,
                        help='flag to force processing only in cpu')
    args = parser.parse_args()

    main(args.batchsize, args.epoch, args.workers, args.force_cpu)
```

```
(linux-torch) ➜  script_example git:(main) ✗ python main.py --workers 8
```

torch.vision.models

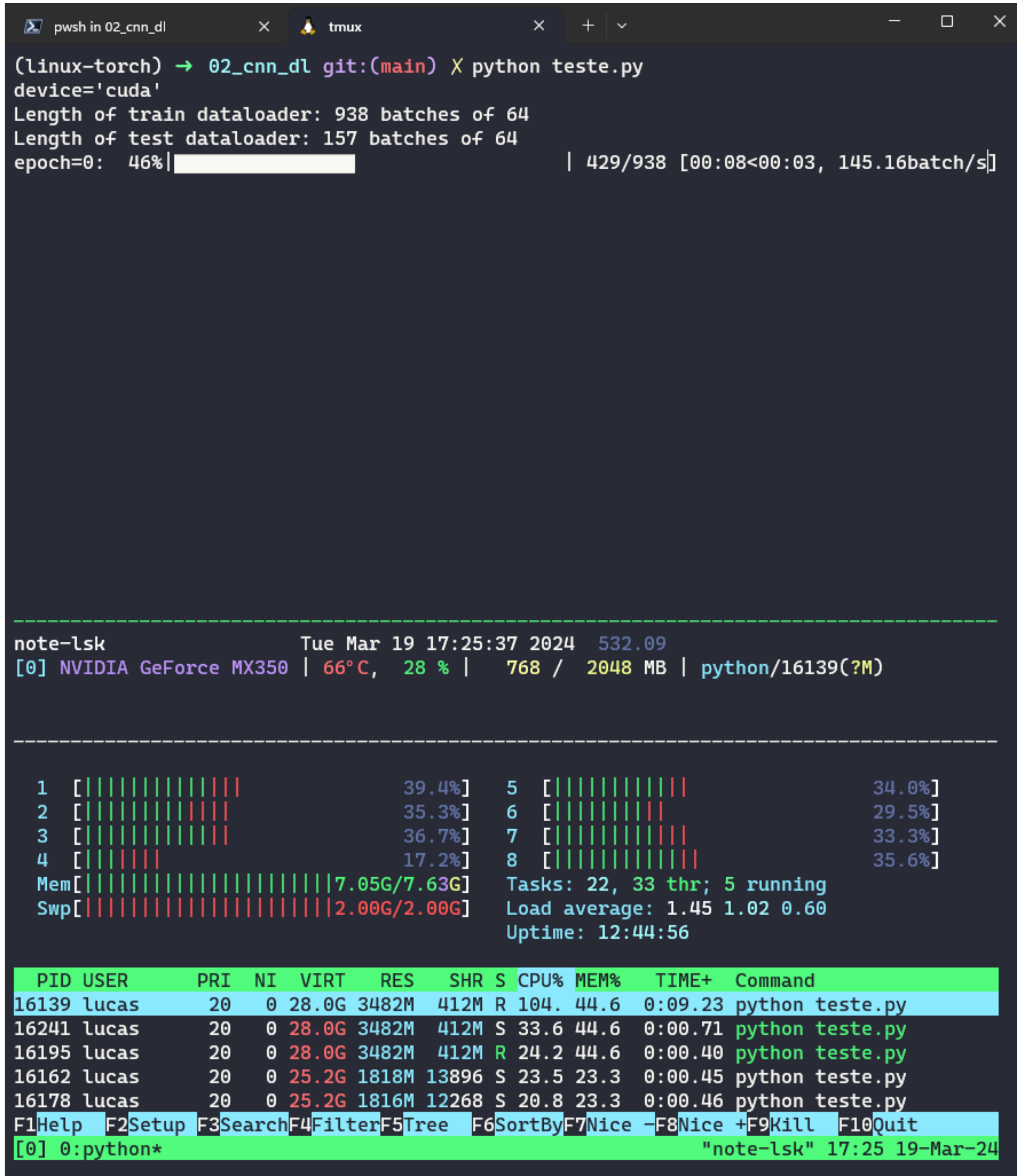
- Temos disponível em torch.vision.models diversas arquiteturas e modelos pré-treinados
- É possível iniciar com um desses modelos e fazer modificações e algumas camadas para possibilitar modificações na tarefa realiza
  - Ex: modificar o número de classes do classificador no final da rede

- AlexNet
- ConvNeXt
- DenseNet
- EfficientNet
- EfficientNetV2
- GoogLeNet
- Inception V3
- MaxVit
- MNASNet
- MobileNet V2
- MobileNet V3
- RegNet
- ResNet
- ResNeXt
- ShuffleNet V2
- SqueezeNet
- SwinTransformer
- VGG
- VisionTransformer
- Wide ResNet

https://pytorch.org/vision/stable/models.html

Algumas ferramentas úteis

- Para uso em terminal
  - **tmux** para permitir execução de diversos terminais simultaneamente e evitar que um problema na conexão finalize um processamento
  - **nvidia-smi** para monitorar as GPUs
  - **gpustat** pacote python para monitorar as GPUs (usa o nvidia-smi)
  - **htop** para avaliar uso de processadores e memória



- https://github.com/tmux/tmux/wiki
- https://github.com/wookayin/gpustat
- https://htop.dev/

Referências

- Documentação Padrão do PyTorch
  - https://pytorch.org/docs/stable/generated/
- PyTorch Fundamentals
  - https://www.learnpytorch.io/00_pytorch_fundamentals/
- tqdm
  - https://tqdm.github.io/
  - https://towardsdatascience.com/training-models-with-a-progress-a-bar-2b664de3e13e