Pytorch チュートリアル

Shohei Watanabe

2024-05-15

環境構築(雑)

ローカルで動かせなくても聞けるようにはしたつもりですが、
自分で実行したい場合は、

- Poetry があれば、

```
poetry install
```

で必要なライブラリをインストールできます。
(version が合わない⇒次ページ)

- requirements.txt があるので、

```
pip install -r requirements.txt
```

でも大丈夫だと思います。
(Rye は使ったことないので分からないです...)

環境構築(雑)

バージョンが合わない場合は、

```
pyenv install 3.9.{いくつでも大丈夫なはず} # 3.9.13 で動作確認
pyenv local 3.9.{}
```

などで Python のバージョンを変更してください。 その後、

```
poetry install
pip install -r requirements.txt
```

で必要なライブラリをインストールしてください。

topics

tensor の基本から始めてモデル学習の流れまでを説明します。

1. Tensors
2. AutoGrad
3. Datasets & Dataloaders
4. Neural Networks (Linear, Conv1d まで)
5. Optimizing Models
6. おまけ
7. Homework

Tensors

Tensors

- PyTorch では tensor というデータ構造が中心的な役割を果たす
- Numpy の ndarray に似ているが、以下のような特徴がある
  - GPU を使った計算が可能
  - 自動微分が可能
- tensor の初期化は以下に示すような方法がある

Tensor Initialization

```
import torch
import numpy as np
```

- データから直接 tensor を作成する

```
data = [[1, 2], [3, 4]]
x_data = torch.tensor(data)
```

- Numpy の ndarray から tensor を作成する

```
np_array = np.array(data)
x_np = torch.from_numpy(np_array)
```

- 他の tensor から新しい tensor を作成する

```
x_ones = torch.ones_like(x_data)
x_rand = torch.rand_like(x_data, dtype=torch.float)
```

Tensor Initialization

- ランダムな値で初期化された tensor を作成する

```
shape = (2, 3,)
rand_tensor = torch.rand(shape)
ones_tensor = torch.ones(shape)
zeros_tensor = torch.zeros(shape)
```

- device を指定して tensor を作成する

```
dtype = torch.float
device = torch.device('cpu') # or 'cuda' etc.
tensor = torch.ones(shape, dtype=dtype, device=device)
```

Tensor Operations

- tensor は Numpy の ndarray と同様に演算が可能

```
x = torch.ones(2, 2)
y = torch.ones(2, 2)
z = x + y # +, -, *, /, @ など Numpy と同様の演算子が使える
print(z)
# tensor([[2., 2.],
#         [2., 2.]])
z[:, 1] = 0 # slicing も可能
print(z)
# tensor([[2., 0.],
#         [2., 0.]])
```

Sending Tensors to GPU

- tensor は to メソッドを使って GPU に送ることができる

```
tensor = torch.ones(4, 4)
if torch.cuda.is_available():
    tensor = tensor.to('cuda') # send to default GPU
    tensor = tensor.to('cuda:0') # send to GPU 0
```

- tensor は device 属性を使って GPU にあるかどうかを確認できる

```
print(tensor.device)
# cuda:0
```

AutoGrad

AutoGrad

- PyTorch の最も重要な機能の一つは自動微分機能
- tensor は `requires_grad=True` を指定することで、その tensor に対する操作を追跡し、 微分を計算することができる

```
x = torch.ones(2, 2, requires_grad=True)
```

- 微分を計算するには `backward` メソッドを呼び出す
- 微分を計算するためには、
  1. `backward` メソッドを呼び出す tensor がスカラーであるか、
  2. あるいは `backward` メソッドに引数として weight を表す tensor を渡す必要がある

Example

- backward メソッドを呼び出す例 (autograd/ex1.py)

```
x = torch.ones(2, 2, requires_grad=True)
y = x + 2
# tensor([[3., 3.],
#         [3., 3.]], grad_fn=<AddBackward0>)
z = y * y * 3 # = 3(x + 2)^2
# tensor([[27., 27.],
#         [27., 27.]], grad_fn=<MulBackward0>)
out = z.mean() # tensor(27., grad_fn=<MeanBackward0>)
out.backward()
print(x.grad) # d(out)/dx = 6(x + 2)/4 = 4.5
# tensor([[4.5000, 4.5000],
#         [4.5000, 4.5000]])
```

Example

- backward メソッドに引数を渡す例 1 (autograd/ex2.py)

```
x = torch.ones(2, 2, requires_grad=True)
y = x + 2
z = y * y * 3
out = z # tensor([[27., 27.], [27., 27.]])
out.backward(gradient=torch.ones(2, 2)) # 各要素に対する重みを指定
# d(out)/dx = 6(x + 2) = 18
print(x.grad)
# tensor([[18., 18.],
#         [18., 18.]])
```

Example

- backward メソッドに引数を渡す例 2 (autograd/ex3.py)

```
x = torch.ones(2, 2, requires_grad=True)
y = x + 2
z = y * y * 3
out = z # tensor([[27., 27.], [27., 27.]])
out.backward(gradient=torch.tensor([[1, 2], [3, 4]]))
print(x.grad)
# tensor([[18., 36.],
#         [54., 72.]])
```

非スカラーの Tensor の微分

上述の通り、`backward` メソッドを非スカラーの tensor に対して呼び出す時には、 `gradient` 引数を指定する必要がある
数学的な意味での微分と、異なる動作をすることに注意

非スカラーの Tensor の微分

$$\boldsymbol{x} := \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}, \boldsymbol{W} = \begin{pmatrix} w_{11} & w_{12} & \cdots & w_{1n} \\ w_{21} & w_{22} & \cdots & w_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{m1} & w_{m2} & \cdots & w_{mn} \end{pmatrix}, \boldsymbol{y} = \boldsymbol{W}\boldsymbol{x} := \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{pmatrix}$$ としたとき、

$$y_i = \sum_{l=1}^{n} w_{il} x_l$$

W の各要素 w_{jk} による $\boldsymbol{y}$ の微分を求めると、

$$\frac{\partial y_i}{\partial w_{jk}} = \delta_{ij} x_k$$

という 3 階のテンソルが得られる

非スカラーの Tensor の微分

NN の weight を更新したい場合、上述のような 3 階テンソルではなく、
2 階(weight の次元に一致)テンソルを得る必要がある
実際に autograd で得られるのは、

$$\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{W}} = \begin{pmatrix} x_1 & x_2 & \dots & x_n \\ x_1 & x_2 & \dots & x_n \\ \vdots & \vdots & \ddots & \vdots \\ x_1 & x_2 & \dots & x_n \end{pmatrix}$$

となる。(ref. `diff_tensor.py`)

非スカラーの Tensor の微分

行列積の微分は次のように計算できる。
Loss をL (スカラー)、$\boldsymbol{Y} = \boldsymbol{W}\boldsymbol{X}$ としたとき、

$$
\begin{aligned}
\frac{\partial L}{\partial \boldsymbol{W}} &= \frac{\partial L}{\partial w_{ij}} \\
&= \frac{\partial L}{\partial y_{ik}} \frac{\partial y_{ik}}{\partial w_{ij}} \quad \left(\because y_{ik} = w_{ij} x_{jk} \right) \\
&= \frac{\partial L}{\partial y_{ik}} x_{jk} \quad \left(\because \frac{\partial y_{ik}}{\partial w_{ij}} = x_{jk} \right) \\
&= \frac{\partial L}{\partial \boldsymbol{Y}} \boldsymbol{X}^T
\end{aligned}
$$

非スカラーの tensor に対する微分を計算する際に指定する `gradient` 引数は、
ここでの$\frac{\partial L}{\partial \boldsymbol{Y}}$に相当する部分である

Disabling Autograd

- requires_grad が True の tensor に対しては、その計算は追跡されるが、torch.no_grad ブロック内では追跡を無効にすることができる (autograd/no_grad.py)

```
x = torch.ones(2, 2, requires_grad=True)
print(x.requires_grad) # True
with torch.no_grad():
    print((x ** 2).requires_grad) # False
```

Disabling Autograd

- `detach()` メソッドを使うことでも追跡を無効にすることができる

```
x = torch.ones(2, 2, requires_grad=True)
y = x.detach()
print(y.requires_grad) # False
```

ユースケースとしては、

1. ファインチューニング時に一部のパラメータを固定する場合
2. `forward` の結果のみが必要な場合の高速化

などがある

Computational Graph

PyTorch では AutoGrad のために計算グラフが構築される

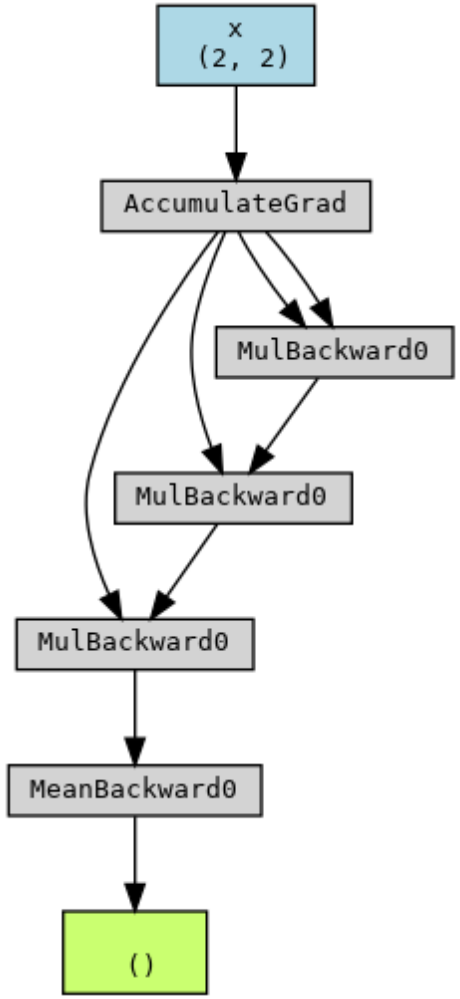
- tensor は `grad_fn` 属性を持ち、その tensor を作成した演算を記録している
- 各演算の微分があらかじめ定義されており、逆伝播時にはそれを使って微分を計算する
- `backward` メソッドを呼ぶと各変数 tensor の `grad` 属性に微分が格納される
- 計算グラフは動的であり、学習を行いながらモデルの構造を変更することが可能

Computational Graph

torchviz を使って計算グラフを可視化することができる (autograd/computational_graph.py)

```
import torchviz
import torch


x = torch.ones(2, 2, requires_grad=True)
y = x * x * x * x
out = y.mean()
out.backward()
dot = torchviz.make_dot(out,
params=dict(x=x, y=y))
dot.render("graph", format="png")
```



Datasets & Dataloaders

Datasets & Dataloaders

- PyTorch では、データセットとデータローダーを使ってデータを扱う
- データセットはデータを格納し、データローダーはデータセットからバッチを取得する
- データセットは `torch.utils.data.Dataset` クラスを継承して作成する
- データローダーは `torch.utils.data.DataLoader` クラスを使って作成する

Example (MNIST)

1. ライブラリのインポート

```
import torch
from torch.utils.data import DataLoader
from torchvision import datasets, transforms
```

Example (MNIST)

2. データセットのダウンロードと変換

```
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.1307,), (0.3081,))
])
train_dataset = datasets.MNIST(
    root='./data',
    train=True,
    download=True, # root にデータがない場合にダウンロードする
    transform=transform
)
test_dataset = datasets.MNIST(
    root='./data',
    train=False,
    download=True,
    transform=transform
)
```

Example (MNIST)

3. データローダーの作成

```
train_loader = DataLoader(
    dataset=train_dataset,
    batch_size=64,
    shuffle=True
)
test_loader = DataLoader(
    dataset=test_dataset,
    batch_size=64,
    shuffle=False
)
```

Custom Dataset

- 自前のデータで Dataset を作ることも可能
- `torch.utils.data.Dataset` クラスを継承して、`__len__`メソッドと`__getitem__`メソッドを実装する

Dataset の作成

ステップ 1: CustomDataset クラスの作成

```
class CustomDataset(Dataset):
    def __init__(self, data, labels, transform=None):
        self.data = data
        self.labels = labels
        self.transform = transform
    def __len__(self):
        return len(self.data)
    def __getitem__(self, idx):
        sample = self.data[idx]
        label = self.labels[idx]
        if self.transform:
            sample = self.transform(sample) # 前処理
        return sample, label
```

DataLoader の使用

ステップ 2: Dataset の初期化

```
data = ... # your data
labels = ... # your labels
dataset = CustomDataset(data, labels)
```

ステップ 3: DataLoader の作成

```
from torch.utils.data import DataLoader
dataloader = DataLoader(dataset, batch_size=4, shuffle=True)
for batch_data, batch_labels in dataloader:
    # training code here
```

__len__

__len__メソッドは、データセットのサイズを返す。

```
class CustomDataset(Dataset):
    def __init__(self, data):
        self.data = data

    def __len__(self):
        return len(self.data)
```

データローダーがデータセットの終了を確認するために使用される。

__getitem__

__getitem__メソッドは、データセットから
特定のインデックスにあるサンプルを取得する。

```
class CustomDataset(Dataset):
    def __init__(self, data):
        self.data = data

    def __getitem__(self, idx):
        return self.data[idx]
```

このメソッドを実装することで、
データセットから特定のデータポイントを取得できる。
データローダーが各バッチのデータを読み込む際に使用される。

Neural Networks

Neural Networks

- Pytorch では `torch.nn` にニューラルネットワークの構築に必要なモジュールが含まれている
- `torch.nn.Module` クラスを継承して、`__init__` メソッドと `forward` メソッドを実装する

Example

```
class NeuralNetwork(nn.Module):
    def __init__(self):
        super(NeuralNetwork, self).__init__()
        self.flatten = nn.Flatten()
        self.linear_relu_stack = nn.Sequential(
            nn.Linear(28*28, 512),
            nn.ReLU(),
            nn.Linear(512, 512),
            nn.ReLU(),
            nn.Linear(512, 10)
        )
    def forward(self, x):
        x = self.flatten(x)
        logits = self.linear_relu_stack(x)
        return logits
```

- `__init__`メソッドでは、
  ネットワークの構造を定義する
- `forward` メソッドでは、
  データがネットワークを通過するときの処理を定義する

Model Parameters

nn.Modlue を subclass に持つモデルのパラメータは、
parameters()や named_parameters()メソッドを使って取得できる
(自前のパラメータは nn.Parameter を使って作成する必要)

```
model = NeuralNetwork()
for name, param in model.named_parameters():
  print(f"Layer: {name} | Size: {param.size()} | Values :
{param[:2]} \n")
# Layer: linear_relu_stack.0.weight | Size: torch.Size([512,
784]) | Values : tensor([[ 0.0022, -0.0021, ...],
# ...
```

nn.Linear

```
torch.nn.Linear(in_features, out_features, bias=True)
```

入力の線形変換を行う $y = xA^T + b$

- in_features: 入力のサイズ
- out_features: 出力のサイズ
- bias: バイアス項を含めるかどうか

```
linear = nn.Linear(20, 30)
input = torch.randn(128, 20)
output = linear(input)
print(output.size())
# torch.Size([128, 30])
```

nn.Conv1d

```
torch.nn.Conv1d(in_channels, out_channels, kernel_size, stride=1,
padding=0, dilation=1, groups=1, bias=True)
```

1 次元の畳み込みを行う

- `in_channels`: 入力のチャンネル数
- `out_channels`: 出力のチャンネル数
- `kernel_size`: カーネルのサイズ
- `stride`: ストライド (default: 1)
- `padding`: パディング (default: 0)
- `dilation`: カーネルの間隔 (default: 1)
- `groups`: グループ数 (default: 1)
- `bias`: バイアス項を含めるかどうか (default: True)

nn.Conv1d

`dilation` はカーネルの間隔を指定する
通常のカーネル

```
[1, 2, 3]
```

dilation=2 の場合

```
[1, 0, 2, 0, 3]
```

nn.Conv1d

groups は入力と出力のチャンネルをグループに分割する

in_channels=4, out_channels=8, groups=2 の場合

```
Group 1: Input channels [0, 1] -> Output channels [0, 1, 2, 3]
Group 2: Input channels [2, 3] -> Output channels [4, 5, 6, 7]
```

nn.Conv1d

出力サイズは以下のように計算される

$$O = \left\lfloor \frac{I + 2 \times P - D \times (K - 1) - 1}{S} \right\rfloor + 1$$

$(O : \text{output size}, I : \text{input size}, D : \text{dilation}$

$P : \text{padding}, K : \text{kernel size}, S : \text{stride})$

nn.Conv1d

```
# 入力チャンネル数 3、出力チャンネル数 6、カーネルサイズ 5 の 1 次元畳み込みレイヤー
conv1d_layer
    = nn.Conv1d(in_channels=3, out_channels=6, kernel_size=5)

# ダミー入力データ (バッチサイズ 10, チャンネル数 3, シーケンス長 50)
input_data = torch.randn(10, 3, 50)

# 畳み込みレイヤーを通してデータを渡す
output_data = conv1d_layer(input_data)

print(output_data.shape)  # torch.Size([10, 6, 46])
```

Optimizing Models

Optimizing Models

- PyTorch では、`torch.optim` に最適化アルゴリズムが実装されている
- モデルのパラメータを更新するためには、
  `torch.optim.Optimizer` クラスを使う
- `torch.optim` モジュールには、
  SGD、Adam、RMSprop などの最適化アルゴリズムが含まれている

Example

```
model = NeuralNetwork()
optimizer = torch.optim.SGD(model.parameters(), lr=1e-3)
```

- model.parameters()でモデルのパラメータを取得し、
  lr で学習率を指定して最適化アルゴリズムを初期化する

```
# Inside the training loop
optimizer.zero_grad() # 勾配を初期化
loss_fn = nn.CrossEntropyLoss()
loss = loss_fn(model(data), target)
loss.backward() # 勾配を計算
optimizer.step() # parameter の tensor.grad に基づいてパラメータを更新
```

Optimizer

全体的な学習の流れは `train/training_ex.py` を参照

おまけ (pytorch の内部実装)

おまけ (pytorch の内部実装)

torch/__init__.py には次のような部分がある

```
for name in dir(_C._VariableFunctions):
  if name.startswith('__') or name in PRIVATE_OPS:
      continue
  obj = getattr(_C._VariableFunctions, name)
  obj.__module__ = 'torch'
  # Hide some APIs that should not be public
  if name == "segment_reduce":
      # TODO: Once the undocumented FC window is passed, remove the line bellow
      globals()[name] = obj
      name = "_" + name
  globals()[name] = obj
  if not name.startswith("_"):
      __all__.append(name)
```

おまけ (pytorch の内部実装)

pytorch の実装はほとんどが CUDA C++で書かれており、_C の実体は `torch._C.so` でありコンパイル時に生成される
コンパイル前のソースの多くはここにある。

いろいろ遡っていくと、最終的に非オープンソースの cuDNN の関数が呼ばれているところ(たとえば ここまで辿り着けます。

Homework

Homework

1. 演算子オーバーロードを用いて、autograd が可能な Tensor class を実装する
2. 1 で作った Tensor に対して計算グラフの出力を行えるように `draw_graph` メソッドを実装する (`graphviz` を使うと便利)
3. `nn.Conv2d` の実装

References

References

1. PyTorch Documentation (https://pytorch.org/docs/stable/index.html)
2. PyTorch Tutorials (https://pytorch.org/tutorials/)