Bài 6 - Pytorch - Buổi 1 - Làm quen với pytorch

10 Aug 2019 - phamdinhkhanh

Menu

1. Pytorch là gì?

Trong số những framework hỗ trợ deeplearning thì pytorch là một trong những framework được ưa chuộng nhiều nhất (cùng với tensorflow và keras), có lượng người dùng đông đảo, cộng đồng lớn mạnh. Vào năm 2019 framework này đã vươn lên vị trí thứ 2 về số lượng người dùng trong những framework hỗ trợ deeplearning (chỉ sau tensorflow). Đây là package sử dụng các thư viện của CUDA và C/C++ hỗ trợ các tính toán trên GPU nhằm gia tăng tốc độ xử lý của mô hình. 2 mục tiêu chủ đạo của package này hướng tới là:

- Thay thế kiến trúc của numpy để tính toán được trên GPU.
- Deep learning platform cung cấp các xử lý tốc độ và linh hoạt.

Trước khi đọc bài viết này, bạn đọc có thể ôn lại bài hướng dẫn về tensorflow deeplearning framework (https://www.kaggle.com/phamdinhkhanh/tensorflow-turtorial) để tự rút ra được những so sánh về các đặc điểm chung và các ưu nhược điểm của 2 framework này.

Để sử dụng pytorch trên GPU bắt buộc các bạn phải cài CUDA và tất nhiên phải có GPU. Hướng dẫn cài đặt pytorch có thể xem tại pytorch install (https://pytorch.org/get-started/locally/). Tuy nhiên google colab thì thư viện này và tensorflow đã được tích hợp sẵn cho người dùng.

Bên dưới chúng ta cùng tìm hiểu:

- Định dạng tensor trên pytorch.
- Các toán tử trên torch tensor.
- Xây dựng một mô hình mạng nơ ron trên pytorch.

Bài viết này được mình tổng hợp và lược dịch từ bài viết Deeplearning with Pytorch: A 60 Minute Blitz (https://pytorch.org/tutorials/beginner/deep_learning_60min_blitz.html) có sự hiệu chỉnh và bổ sung về nội dung dựa trên kiến thức của mình.

1.1. Tensor

Là dữ liệu nhiều chiều tương tự như ma trận trong numpy nhưng được thêm các tính chất để có thể hoạt động trên GPU nhằm gia tăng tốc độ tính toán. Các định dạng dữ liệu tensor của pytorch khá giống với tensorflow (https://www.kaggle.com/phamdinhkhanh/tensorflow-turtorial). Tuy nhiên trong quá trình khởi tạo chúng trên pytorch chúng ta không cần phải truyền code vào trong 1 session để tạo tensor như tensorflow. Qua các ví dụ bên dưới chúng ta sẽ cùng làm quen với các dạng tensor chính của pytorch.

```
from __future__ import print_function
import torch
```

```
# Khởi tạo một ma trận rỗng

x = torch.empty(5, 3)
print(x)
```

```
tensor([[1.9225e-36, 0.0000e+00, 3.3631e-44],
        [0.0000e+00,        nan, 0.0000e+00],
        [1.1578e+27, 1.1362e+30, 7.1547e+22],
        [4.5828e+30, 1.2121e+04, 7.1846e+22],
        [9.2198e-39, 0.0000e+00, 0.0000e+00]])
```

```
# Khởi tạo một ma trận ngẫu nhiên

x = torch.rand(5, 3)
print(x)
```

```
tensor([[0.1862, 0.5766, 0.7265],
        [0.5885, 0.8067, 0.5271],
        [0.3040, 0.2556, 0.9610],
        [0.6661, 0.6096, 0.5479],
        [0.3799, 0.8784, 0.8257]])
```

```
# Khởi tạo ma trận 0 với data type là long

x = torch.zeros(5, 3, dtype = torch.long)
print(x)
```

Top

```
tensor([[0, 0, 0],
        [0, 0, 0],
        [0, 0, 0],
        [0, 0, 0],
        [0, 0, 0]])
```

```
# Khởi tạo ma trận từ list

x = torch.tensor([[5, 3.5]])
print(x)
```

```
tensor([[5.0000, 3.5000]])
```

```
# Khởi tạo ma trận có các thuộc tính tương tự như của một ma trận sẵn c

x = x.new_ones(5, 3, dtype = torch.double)
print(x)

x = torch.randn_like(x, dtype = torch.float) #override dtype
# Ma trận mới được khởi tạo ngẫu nhiên có shape tương tự như ma trận c
print(x)
```

```
 tensor([[1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.]], dtype=torch.float64)
 tensor([[ 1.0536, -0.0921, -0.7262],
        [ 0.7677, -0.2225,  0.0847],
        [ 0.4700,  0.6618,  0.3220],
        [-1.3720, -0.9271,  0.3299],
        [ 0.4915, -0.5932, -1.8757]])
```

```
# get size
print(x.size())
```

```
torch.Size([5, 3])
```

Trên thực tế torch.Size() là tuple nên sẽ hỗ trợ các operation dạng tuple.

1.2. Operations

Có rất nhiều các operation trên pytorch như: cộng, trừ, nhân, chia, reshape, khởi tạo ngẫu nhiên. Chúng ta sẽ lần lượt tìm hiểu:

- **Khởi tạo ngẫu nhiên**

Top

```
x = torch.randn(3, 3)
print(x)
y = torch.ones(3, 3)
print(y)
print(x+y)
```

```
tensor([[ 0.7770, -1.0313, -0.5739],
        [ 2.2917,  0.4533,  1.9091],
        [-0.1328,  0.2833, -0.9506]])
tensor([[1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.]])
tensor([[ 1.7770, -0.0313,  0.4261],
        [ 3.2917,  1.4533,  2.9091],
        [ 0.8672,  1.2833,  0.0494]])
```

- **Phép cộng**

```
print(torch.add(x, y))
```

```
tensor([[ 1.7770, -0.0313,  0.4261],
        [ 3.2917,  1.4533,  2.9091],
        [ 0.8672,  1.2833,  0.0494]])
```

hoặc cũng có thể đưa kết quả vào một giá trị khởi tạo rỗng.

```
z = torch.empty(3,3)
torch.add(x, y, out = z)
print(z)
```

```
tensor([[ 1.7770, -0.0313,  0.4261],
        [ 3.2917,  1.4533,  2.9091],
        [ 0.8672,  1.2833,  0.0494]])
```

Cách khác: Triển khai tính toán inplace. Tức tính toán và lưu kết quả ngay trên đối tượng được áp dụng.

```
print(y)
y.add_(x)
print(y)
```

```
tensor([[1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.]])
tensor([[ 1.7770, -0.0313,  0.4261],
        [ 3.2917,  1.4533,  2.9091],
        [ 0.8672,  1.2833,  0.0494]])
```

Note: Các biến đổi inplace trên tensor sẽ có suffix là `_` . Chẳng hạn: `x.copy_(y), x.t_()` sẽ thay đổi trên chính giá trị của x.

Top

- **Truy cập index**

Có thể sử dụng các tính chất của numpy 1 cách dễ dàng. Chẳng hạn như indexing.

```
# Truy cập cột thứ 2 của x
print(x[:, 1])
```

```
tensor([-1.0313,  0.4533,  0.2833])
```

- **reshape tensor**

Để resize/reshape tensor ta sử dụng hàm view

```
x = torch.randn(4, 4)
y = x.view(16)
z = x.view(-1, 8) # Giá trị -1 cho biết kích thước của chiều này được
print(x.size(), y.size(), z.size())
```

```
torch.Size([4, 4]) torch.Size([16]) torch.Size([2, 8])
```

Chúng ta có thể chuyển tensor x có 1 phần tử sang 1 numeric python bằng hàm `item()`.

```
x = torch.tensor([1.5])
i = x.item()
print(i)
```

```
1.5
```

1.3. Các kết nối với numpy

Chuyển đổi Torch tensor sang numpy array khá dễ dàng.

Torch tensor và numpy array sẽ sử dụng chung các địa chỉ ô nhớ khi torch tensor hoạt động trên CPU, do đó khi thay đổi giá trị này sẽ thay đổi giá trị kia. Gần giống như phép gán trong pandas nếu không sử dụng `copy()`. Bên dưới ta sẽ thử nghiệm khởi tạo a trên pytorch và b là giá trị numpy của tensor torch a. Thay đổi a và kiểm tra giá trị của b có bị thay đổi tương ứng không?

1.3.1. Chuyển đổi torch tensor sang numpy array

```
a = torch.ones(5)
print(a)
print(type(a))

b = a.numpy()
print(type(b))
```

Top

```
tensor([1., 1., 1., 1., 1.])
<class 'torch.Tensor'>
<class 'numpy.ndarray'>
```

Khi thay đổi a sẽ thay đổi phần tử của b như thế nào?

```
# Thêm 1 phần tử 1 vào a
a.add_(1)
print(a)
print(b)
```

```
tensor([2., 2., 2., 2., 2.])
[2. 2. 2. 2. 2.]
```

Khi a thay đổi thì giá trị numpy của nó là b cũng thay đổi tương ứng và có các phần tử bên trong bằng a.

1.3.2. chuyển đổi numpy array sang torch tensor

Để chuyển 1 numpy array sang pytorch ta sử dụng hàm `from_numpy()`

```
import numpy as np

a = np.ones(5)
b = torch.from_numpy(a)

print(a)
print(b)
print(type(a))
print(type(b))
```

```
[1. 1. 1. 1. 1.]
tensor([1., 1., 1., 1., 1.], dtype=torch.float64)
<class 'numpy.ndarray'>
<class 'torch.Tensor'>
```

Tất cả các tensor CPU ngoại trừ CharTensor hỗ trợ chuyển đổi về numpy và ngược lại.

2. CUDA tensor

là định dạng tensor nhưng được đưa lên device (có thể là cpu, cuda, mkldnn, opengl, …). Hỗ trợ các tính toán nhanh hơn nhờ kiến trúc CUDA. Để đưa 1 tensor lên một thiết bị bất kì ta sử dụng hàm `to()` .

Top

```
# kiểm tra xem có tồn tại CUDA trên máy không. Lưu ý nếu đang sử dụng
if torch.cuda.is_available():
  device = torch.device("cuda") # Khởi tạo một cuda device object
  y = torch.ones_like(x, device = device) # Trực tiếp khởi tạo một ten
  x = x.to(device) # Truyền giá trị tensor vào thiết bị. Có thể truyền
  z = x + y
  print(z)
  print(z.to("cpu", torch.double)) # Trong hàm .to() ta có thể thay đị
```

```
tensor([2.5000], device='cuda:0')
tensor([2.5000], dtype=torch.float64)
```

2.1. Autograd: Tự động tính đạo hàm

Trung tâm của toàn bộ các mạng nơ ron hoạt động trên pytorch là `autograd` package. Hãy tìm hiểu về autograd trước khi thực sự xây dựng một mạng nơ ron trên pytorch.

Chức năng của autograd: Tự động tính toán đạo hàm trên toàn bộ các toán tử của tensors. Nó là một framework được định nghĩa trong quá trình chạy, có nghĩa rằng quá trình lan truyền ngược được xác định khi mà code được chạy, và do đó mỗi vòng lặp có thể có kết quả thay đổi tham số theo lan truyền ngược khác nhau.

Theo dõi lịch sử của tensor torch: torch.tensor là package khởi tạo các tensor torch. Mỗi một tensor torch sẽ có 1 thuộc tính là `.requires_grad` , nếu bạn set thuộc tính này về True, các toán tử triển khai trên tensor sẽ được theo dõi. Khi kết thúc quá trình lan truyền thuận (hoặc quá trình tính toán output) bạn có thể gọi `.backward()` và mọi tính toán gradient sẽ được tự động thực hiện dựa trên lịch sử đã được lưu lại. Các gradient cho tensor này sẽ được tích lũy và xem tại thuộc tính `.grad` .

Để dừng theo dõi một tensor chúng ta gọi vào hàm `.detach()` . Khi đó các hoạt động trên tensor sẽ không còn được lưu vết nữa.

Ngoài ra để ngăn tensor lưu lại lịch sử (và sử dụng memory), chúng ta cũng có thể bao quanh code block triển khai tensor với hàm `with torch.no_grad():` nó rất hữu ích trong trường hợp đánh giá model bởi vì khi thuộc tính `requires_grad = True` thì model sẽ có thể được cập nhật tham số. Nhưng quá trình đánh giá model sẽ không cần cập nhật tham số nên chúng ta không cần áp dụng gradient lên chúng. Đơn giản là set `requires_grad = False` .

Bạn đọc tạm chấp nhận lý thuyết nêu trên, phần thực hành bên dưới sẽ giúp giải thích sáng tỏ.

Ngoài ra class Function cũng rất quan trọng trong thực hiện autograd.

Lưu trữ đồ thị tính toán:

2 class `Tensor` và `Function` cùng tương tác và xây dựng một đồ thị chu trình mà đồ thị này mã hóa lại toàn bộ lịch sử tính toán. Mỗi một tensor đều có 1 thuộc tính `grad_fn` trích dẫn đến một `Function` đã tạo ra `Tensor` (trừ trường hợp tensor được tạo ra bởi user được set thuộc tính `grad_fn` là `None`).

Nếu muốn tính toán đạo hàm chúng ta gọi vào hàm `.backward()` của `Tensor` . Nếu `Tensor` là một scalar sẽ không cần xác định bất kì đối số `gradient` nào cho `.backward()` . Tuy nhiên khi `Tensor` có nhiều hơn 1 phần tử cần xác định đối số `gradient` là một tensor có cùng kích thước. Đối số này sẽ qui định tốc độ thay đổi theo `gradient` tại mỗi chiều là bao nhiêu.

Top

Ví dụ về lưu trữ đồ thị tính toán:

Bên dưới ta sẽ khởi tạo một tensor torch có khả năng theo dõi thay đổi theo 2 cách:

- Cách 1: set `requires_grad = True` .

```
import torch

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

y = x+2
print(y)
```

```
tensor([[1., 1.],
        [1., 1.]], requires_grad=True)
tensor([[3., 3.],
        [3., 3.]], grad_fn=<AddBackward0>)
```

Do tại x ta đã theo dõi thay đổi bằng các set tham số `requires_grad = True` nên các tính toán được thực hiện trên x sẽ được theo dõi lại ở thuộc tính `grad_fn` .

```
print(y.grad_fn)
```

```
<AddBackward0 object at 0x7f211d04fc18>
```

Giá trị của `grad_fn` cho thấy ta đã thực hiện một phép cộng để thu được y. Thực hiện tiếp 1 biến đổi nữa sử dụng y.

```
z = y * y * 3
out = z.mean()
print(z)
print(out)
```

```
tensor([[27., 27.],
        [27., 27.]], grad_fn=<MulBackward0>)
tensor(27., grad_fn=<MeanBackward0>)
```

- Cách 2: Sử dụng inplace function.

Hoặc chúng ta có thể thiết lập requires_grad theo cách inplace. Mặc định của `requires_grad` khi khởi tạo 1 tensor torch là False tức là sẽ không ghi lại lịch sử thay đổi.

```
a = torch.ones(3, 3)
print(a)
print(a.requires_grad)
a.requires_grad_(True)
print(a.requires_grad)
b = (a*a).sum()
print(b)
print(b.grad_fn)
```

Top

```
tensor([[1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.]])
False
True
tensor(9., grad_fn=<SumBackward0>)
<SumBackward0 object at 0x7f216932d4a8>
```

2.1.2. Gradients

Bây h ta sẽ thực hiện một lan truyền ngược (backprop) thông qua hàm `out.backward()`.

```
out.backward()
```

Kết quả của gradient `d(out)/dx` sẽ được lưu trong phần tử `x.grad`.

```
print(x.grad)
```

```
tensor([[4.5000, 4.5000],
        [4.5000, 4.5000]])
```

Chúng ta nhận được ma trận chỉ gồm các phần tử là 4.5. Đây chính là đạo hàm của mỗi phần tử của x theo y. $\frac{dy}{dx} = \frac{d(\frac{3(x+2)^2}{4})}{dx} = \frac{3(x+2)}{2}$

Tại $x = 1$ ta suy ra $\frac{dy}{dx} = 4.5$

Hàm `torch.autograd` sẽ là hàm chức năng tính tích giữa vector và ma trận jacobian. Hàm số cho ta biết mức độ thay đổi của các chiều khi đi theo phương gradient.

- Định nghĩa về ma trận jacobian: giả sử $\mathbf{f}$ là một hàm số ánh xạ từ vector $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ lên vector hàm số $\mathbf{y} = (y_1, y_2, \ldots, y_m) : \mathbb{R}^n \to \mathbb{R}^m$. Khi đó ma trận Jacobian của hàm số $\mathbf{f}$ chính là ma trận đạo hàm bậc nhất của vector hàm số y theo các chiều của vector $\mathbf{x}$.

$$\mathbf{J} = \nabla_{\mathbf{x}}\mathbf{y} = \begin{bmatrix} \frac{\nabla y_1}{\nabla x_1} & \frac{\nabla y_1}{\nabla x_2} & \cdots & \frac{\nabla y_1}{\nabla x_n} \\ \frac{\nabla y_2}{\nabla x_1} & \frac{\nabla y_2}{\nabla x_2} & \cdots & \frac{\nabla y_2}{\nabla x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\nabla y_m}{\nabla x_1} & \frac{\nabla y_m}{\nabla x_2} & \cdots & \frac{\nabla y_m}{\nabla x_n} \end{bmatrix}$$

Cụ thể hơn chúng ta có thể tham khảo ở link sau: Ma trận jacobian - wiki (https://en.wikipedia.org/wiki/Jacobian_matrix_and_determinant)

Cho một vector $\mathbf{v} = (v_1, v_2, \ldots, v_m)^T$ bất kì. Nếu $\mathbf{v}$ là ma trận gradient của hàm loss function $l = g(y)$ thì $\mathbf{v} = (\frac{\nabla l}{\nabla y_1}, \frac{\nabla l}{\nabla y_2}, \ldots, \frac{\nabla l}{\nabla y_m})^T$, do đó theo công thức chain rule thì tích vector-jacobian sẽ là gradient của hàm l tương ứng với vector $\mathbf{x}$:
$\nabla_{\mathbf{x}} l = \nabla_{\mathbf{x}} g(\mathbf{y}) = \nabla_{\mathbf{x}} \mathbf{y}^T \nabla_{\mathbf{y}} g(\mathbf{y}) = \mathbf{J}^T \mathbf{v}$

Đây chính là giá trị của tích vector-jacobian được tính toán dựa trên hàm số `torch.autograd`. Công thức này giúp ta dễ dàng truyền các gradient bên ngoài vào `backward()` để tùy biến gradient của mô hình theo gradient truyền vào. Cụ thể hơn xem ví dụ bên dưới:

top

Bây h chúng ta cùng xét ví dụ về tích vector-jacobian.

```
import torch

x = torch.randn(3, requires_grad = True)
yhat = torch.randn(3, requires_grad = True)*2
y = x*2
l = ((y-yhat)**2).mean()
print(l)
```

```
tensor(0.4746, grad_fn=<MeanBackward0>)
```

Trong TH này $\mathbf{y}$ sẽ không còn là 1 scalar. Hàm `torch.autograd` sẽ không tính toán ma trận jacobian trực tiếp mà thay vào đó sẽ tính tích vector-jacobian theo vector $\mathbf{v}$ truyền vào đối số `.backward()`.

```
# Khởi tạo một vector gradient tự do v
v = torch.tensor([0.1, 1.0, 0.001], dtype = torch.float)
# Tính ma trận Jacobian (đạo hàm của y theo v)
l.backward(v)
# Tính tích vector-jacobian chính là đạo hàm của
print(x.grad)
```

```
tensor([-1.5380,  0.7883, -0.2852])
```

Để dừng autograd theo dõi các thay đổi lịch sử trên tensor, chúng ta có thể thiết lập `.requires_grad = True` hoặc đặt các biến đổi tensor trong block code `torch.no_grad()`.

```
print(x.requires_grad)
print((x*x).requires_grad)
with torch.no_grad():
  print((x*x).requires_grad)
```

```
True
True
False
```

3. Xây dựng mạng neural network

3.1. Kiến trúc mạng CNN

Các mạng neural sẽ được xây dựng dựa trên package torch.nn. Dựa trên `autograd` model sẽ xác định đạo hàm bậc 1 theo các chiều dữ liệu. Một nn.Module sẽ bao gồm các layers và một phương thức `forward(input)` để trả ra kết quả `output`.

Xây dựng mạng neural network sẽ trả qua các bước sau:

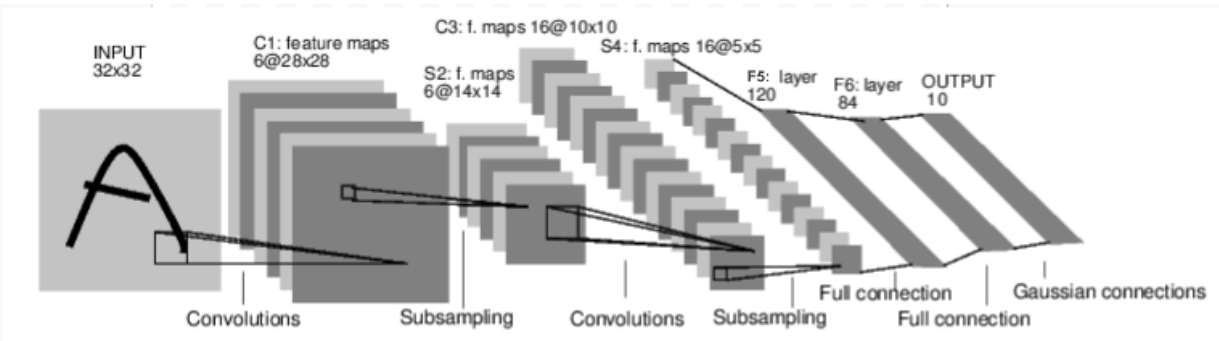
- Xây dựng kiến trúc mạng nơ ron.
- Phân chia dữ liệu train, test.
- Xác định phương pháp optimization để cập nhật gradient descent và hàm loss function.

Top

- Huấn luyện model.
- Hậu kiểm model.

Về convolution layer xem tại: Convolution layer (https://www.kaggle.com/phamdinhkhanh/convolutional-neural-network-p1)

Tiếp theo chúng ta sẽ xây dựng kiến trúc mạng Lenet để phân biệt hình ảnh các đồ vật và loài vật. Sơ đồ kích thước các layers của mạng như bên dưới:



Hình 1 Kiến trúc mạng Lenet sử dụng các convolutional neural network.

Để xây dựng mạng neural chúng ta sẽ kế thừa object `nn.Module`. Object này sẽ cho phép thực hiện quá trình lan truyền thuận và lan truyền ngược thông qua 2 hàm `forward()` và `backward()`.

Top

```
import torch
import torch.nn as nn
import torch.nn.functional as F

class Net(nn.Module):
  def __init__(self):
    super(Net, self).__init__()
    # 1 input image channel, 6 output channels, 3x3 square convolutio
    # kernel
    # conv2d (input chanels, output chanels, kernel size)
    self.conv1 = nn.Conv2d(1, 6, 3)
    self.conv2 = nn.Conv2d(6, 16, 3)
    # an affine operation: y = Wx + b
    self.fc1 = nn.Linear(16 * 6 * 6, 120)  # 6*6 from image dimension
    self.fc2 = nn.Linear(120, 84)
    self.fc3 = nn.Linear(84, 10)

  def forward(self, x):
    # Max pooling over a (2, 2) window
    x = F.max_pool2d(F.relu(self.conv1(x)), (2, 2))
    # If the size is a square you can only specify a single number
    x = F.max_pool2d(F.relu(self.conv2(x)), 2)
    x = x.view(-1, self.num_flat_features(x))
    x = F.relu(self.fc1(x))
    x = F.relu(self.fc2(x))
    x = self.fc3(x)
    return x

  def num_flat_features(self, x):
    size = x.size()[1:]  # all dimensions except the batch dimension
    num_features = 1
    for s in size:
        num_features *= s
    return num_features


net = Net()
print(net)
```

```
Net(
  (conv1): Conv2d(1, 6, kernel_size=(3, 3), stride=(1, 1))
  (conv2): Conv2d(6, 16, kernel_size=(3, 3), stride=(1, 1))
  (fc1): Linear(in_features=576, out_features=120, bias=True)
  (fc2): Linear(in_features=120, out_features=84, bias=True)
  (fc3): Linear(in_features=84, out_features=10, bias=True)
)
```

Chúng ta phải xác định trước `forward` function để trả ra kết quả của model ở đầu ra. Dựa trên `forward` function, hàm `backward` function (là nơi mà gradients tại mỗi layers được tính toán) sẽ được tự động xác định khi bạn sử dụng `autograd`. Chúng ta cũng có thể sử dụng bất kì một phép biến đổi toán tử Tensor nào trên hàm forward function.

Các tham số huấn luyện (tham số mà có thể thay đổi được trong huấn luyện) của mô hình được trả về bằng hàm `net.parameters()`.

Top

```
params = list(net.parameters())
print(len(params))
print(params[0].size())  # conv1's .weight
```

```
10
torch.Size([6, 1, 3, 3])
```

Chúng ta thử nghiệm khởi tạo một đầu vào ngẫu nhiên 32x32. Chú ý rằng: Kì vọng đầu vào của mạng lenet là 32x32. Để sử dụng mạng này trên MNIST dataset chúng ta sẽ phải resize kích thước ảnh về 32x32.

```
input = torch.randn(1, 1, 32, 32)
out = net(input)
print(out)
```

```
tensor([[ 0.0451, -0.0784, -0.0997, -0.0572, -0.0218, -0.1198, -0.0557
          0.1927, -0.0887]], grad_fn=<AddmmBackward>)
```

Khi đó kết quả đầu ra thu được là một tensor có 10 phần tử, mỗi phần tử tương ứng với điểm số được phân bố cho class mà nó thuộc về. Bên dưới chúng ta chuyển toàn bộ các gradients trong bộ nhớ đệm về 0 bằng hàm `.zero_grad()` và lan truyền ngược với gradients ngẫu nhiên.

```
net.zero_grad()
out.backward(torch.randn(1, 10))
```

Lưu ý: `torch.nn` chỉ hỗ trợ các mini-batches. Toàn bộ `torch.nn` packages chỉ hỗ trợ đầu vào là mini-batch của mẫu (tức là luôn có 1 chiều trong shape qui định `batch size`), và không tiếp nhận 1 mẫu đơn lẻ.

Chẳng hạn, `nn.Conv2d` sẽ nhận đầu vào là 4D Tensor của `nSamples x nChannels x Height x Width`. Trong đó chiều đầu tiên là kích thước mẫu (`batch size`).

Nếu bạn có một mẫu đơn lẻ, chỉ cần sử dụng `input.unsqueeze(0)` để thêm vào một chiều `batch size` giả mạo.

Tổng kết:

- torch.Tensor: là một mảng nhiều chiều hỗ trợ các biến đổi autograd như `backward()`. Và cũng lưu trữ các gradients của tensor.
- nn.Module: Neural network module. Thuận tiện trong đóng gói các tham số với sự hỗ trợ để đẩy chúng lên GPU, export và loading tham số,….
- nn.Parameter: Là một dạng tensor lưu trữ tham số huấn luyện và được phân bố như một thuộc tính của Module.
- autograd.Function: Kế thừa quá trình lan truyền thuận và lan truyền ngược của một biến đổi autograd. Mọi triển khai `Tensor` tạo ra ít nhất `Function` node kết nối đến function được tạo bởi tensor và mã hóa lịch sử của chúng.

3.1. Hàm loss

Top

Một hàm loss sẽ nhận 1 cặp (output, target) và tính toán giá trị khoảng cách giữa output và giá trị target.

Có một số dạng loss function khác nhau mà chúng ta có thể tham khảo được hỗ trợ trong nn package: Loss function trong nn (https://pytorch.org/docs/stable/nn.html). Dạng đơn giản nhất là `nn.MSELoss` (trung bình bình phương sai số) tính toán trung bình bình phương sai số giữa giá trị output và giá trị target.

Chúng ta có thể xem ví dụ như bên dưới:

```
output = net(input)
target = torch.randn(10) # a dummy target, for example
target = target.view(1, -1) # make it same shape as output
criterion = nn.MSELoss()

loss = criterion(output, target)
print(loss)
```

```
tensor(1.1689, grad_fn=<MseLossBackward>)
```

Tiến trình backward của hàm loss function sẽ sử dụng thuộc tính `.grad_fn` của nó để lần tìm trên đồ thị tính toán quá trình biến đổi tensor như bên dưới:

`input -> conv2d -> relu -> maxpool2d -> conv2d -> relu -> maxpool2d -> view -> linear -> relu -> linear -> relu -> linear -> MSELoss -> loss`

Khi ta gọi vào hàm `loss.backward()`, toàn bộ graph sẽ tính toán đạo hàm của loss function, các tensors trong graph có thuộc tính `requires_grad = True` thì sẽ có tensor `.grad` được cập nhật gradient theo trình tự lũy tiến.

Để minh họa chúng ta có thể sử dụng một vài bước backward:

```
print(loss.grad_fn) # MSELoss
print(loss.grad_fn.next_functions[0][0]) # Linear
print(loss.grad_fn.next_functions[0][0].next_functions[0][0]) # Relu
```

```
<MseLossBackward object at 0x7f32a2e7ef28>
<AddmmBackward object at 0x7f32a2e7e518>
<AccumulateGrad object at 0x7f32a2e7ef28>
```

3.2. Lan truyền ngược (backpropagation)

Để lan truyền ngược chúng ta sử dụng hàm `loss.backward()`. Nhưng trước đó chúng ta cần xóa nhưng gradients đang có và các gradient khác sẽ tích lũy vào gradient hiện có.

Bên dưới chúng ta sẽ cùng gọi vào hàm `loss.backward()`, và chúng ta phải nhìn vào hệ số chệch của conv1 gradient trước và sau khi backward.

Top

```
net.zero_grad() # chuyển về 0 toàn bộ các gradient trong bộ nhớ đệm củ
print('conv1.bias.grad before backward')
print(net.conv1.bias.grad)

loss.backward()

print('conv1.bias.grad after backward')
print(net.conv1.bias.grad)
```

```
conv1.bias.grad before backward
tensor([0., 0., 0., 0., 0., 0.])
conv1.bias.grad after backward
tensor([-0.0063, -0.0041,  0.0004,  0.0066,  0.0064,  0.0063])
```

3.3. Cập nhật trọng số.

Công thức đơn giản để cập nhật trọng số là:

```
weight = weight - learning_rate * gradient
```

Chúng ta có thể triển khai bằng sử dụng python code đơn giản như sau:

```
learning_rate = 0.01
for f in net.parameters():
  f.data.sub_(f.grad.data * learning_rate)
```

Trong đó `_sub()` là một hàm inplace của phép trừ.

Tuy nhiên khi sử dụng mạng neural networks, bạn muốn sử dụng đa dạng các phương pháp cập nhật gradient descent khác nhau như SGD, Nesterov-SGD, Adam, RMSProp,…. Do đó sử dụng package torch.optim chúng ta có thể thực hiện được toàn bộ những phương pháp gradient descent này một cách đơn giản.

```
import torch.optim as optim

# Create your optimizer
optimizer = optim.SGD(net.parameters(), lr = 0.001)

# in your training loop
optimizer.zero_grad() # zero gradients buffers
output = net(input)
loss = criterion(output, target)
loss.backward()
optimizer.step() #Does update
```

Hàm `criterion()` được sử dụng để tính loss function. `loss.backward()` sẽ thực hiện quá trình lan truyền ngược và `optimizer.step()` được dùng để cập nhật gradients theo phương pháp optimization.

3.4. Huấn luyện một mô hình phân lớp

Top

Phần này được tham khảo từ code của bài viết gốc: Hướng dẫn phân loại ảnh pytorch (https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html)

Như vậy chúng ta đã hình dung cơ bản được cách nào để xây dựng một mạng neural và làm thế nào để tính toán loss function và cập nhật trọng số.

Nhưng bước quan trọng nhất của mô hình đó là chuyển hóa dữ liệu từ raw data sang numpy array của python để model có thể đọc hiểu được.

Các định dạng dữ liệu thông thường bạn làm việc sẽ là hình ảnh, âm thanh, đoạn text, đoạn video. Bạn có thể sử dụng các packages của python để đọc những dữ liệu này dưới dạng numpy và sau đó convert những array này sang torch tensor.

- Đối với hình ảnh, packages có thể sử dụng là pillow, opencv.
- Đối với âm thanh chúng ta có thể sử dụng scipy hoặc librosa.
- Đối với định dạng text NLTK và Spacy có thể hữu ích.

Để sử dụng chuyên biệt cho đọc và xử lý ảnh trên pytorch chúng ta có thể sử dụng một packages là `torchvision`. Package này có thể load được các bộ ảnh lớn như CIFAR10, MNIST, … và biến đổi dữ liệu ảnh thông qua các module torchvision.datasets, torchvision.utils.data.DataLoader hay visualization.

3.5. Huấn luyện một model phân loại ảnh

Chúng ta sẽ đi qua các step sau đây:

- Load hình ảnh và chuẩn hóa tập dữ liệu hình ảnh CIFAR10 sử dụng torchvision.
- Xác định kiến trúc mạng neural.
- Xác định hàm loss function.
- Huấn luyện model trên tập training.
- Đánh giá model trên tập testing.

3.5.1. Loading và chuẩn hóa CIFAR10

Sử dụng torchvision chúng ta có thể dễ dàng load các hình ảnh trong CIFAR10. Đầu ra của torchvision dataset là các hình ảnh PILImage nằm trong khoảng [0, 1]. Chúng ta sẽ biến đổi chúng thành các Tensors chuẩn hóa về khoảng [-1, 1].

Top

```
import torch
import torchvision
import torchvision.transforms as transforms

# Xây dựng một chuẩn hóa đầu vào cho ảnh

transform = transforms.Compose(
  [transforms.ToTensor(),
  transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))]
)

# Khởi tạo dữ trainset qui định dữ liệu training
trainset = torchvision.datasets.CIFAR10(root = './CIFAR10', train = T
                                        download = True, transform = t

# Khởi tạo trainloader qui định cách truyền dữ liệu vào model theo ba
trainloader = torch.utils.data.DataLoader(trainset, batch_size = 4,
                                          shuffle = True, num_workers =

# Tương tự nhưng đối với test
testset = torchvision.datasets.CIFAR10(root = './CIFAR10', train = Fa
                                       download = True, transform = tr

testloader = torch.utils.data.DataLoader(testset, batch_size = 4,
                                                 shuffle = False, num_wo

# Nhãn cho các class
classes = ('plane', 'car', 'bird', 'cat',
           'deer', 'dog', 'frog', 'horse', 'ship', 'truck')
```

```
0it [00:00, ?it/s]

Downloading https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz to


 98%|█████████▊| 167616512/170498071 [00:11<00:00, 17124241.10it/s]

Files already downloaded and verified
```

```
print(type(trainset[0][0]))
print(trainset[0][0].size())
```

```
<class 'torch.Tensor'>
torch.Size([3, 32, 32])
```

Các object trainset và testset là dữ liệu mà chúng ta sử dụng để huấn luyện model (chính là list các tensor đại diện cho các bức ảnh). Những object còn lại bao gồm trainLoader và testLoader qui định dữ liệu chúng ta lấy từ đâu và cách thức chúng ta truyền dữ liệu vào mô hình theo batch với kích thước bao nhiêu, có thực hiện shuffle các batch sau khi hết một epoch hay không?

Top

Hiển thị một số hình ảnh bằng matplotlib

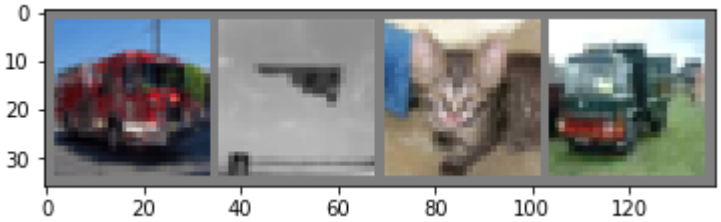
```
import matplotlib.pyplot as plt
import numpy as np

# functions to show an image


def imshow(img):
    img = img / 2 + 0.5     # unnormalize
    npimg = img.numpy()
    plt.imshow(np.transpose(npimg, (1, 2, 0)))
    plt.show()


# get some random training images
dataiter = iter(trainloader)
images, labels = dataiter.next()

# show images
imshow(torchvision.utils.make_grid(images))
# print labels
print(' '.join('%5s' % classes[labels[j]] for j in range(4)))
```



</img> truck plane cat truck

```
print(type(trainloader))
print(images.shape)
```

```
<class 'torch.utils.data.dataloader.DataLoader'>
torch.Size([4, 3, 32, 32])
```

3.5.2. Xác định một mạng neural network

Khởi tạo một mạng neural network thông qua class net như bên dưới. Hàm tạo `__init__()` sẽ chứa những layers của class và hàm `forward()` được sử dụng để ráp nối các layer và trả về một module hoàn chỉnh.

Để hiểu về các layer trong pytorch chúng ta tham khảo tại pytorch layer (https://pytorch.org/docs/stable/nn.html?highlight=nn%20linear#torch.nn.Linear).

Top

```
import torch.nn as nn
import torch.nn.functional as F


class Net(nn.Module):
  def __init__(self):
    super(Net, self).__init__()
    # Conv2d: input nodes, output nodes, kernel size
    self.conv1 = nn.Conv2d(3, 6, 5)
    self.pool = nn.MaxPool2d(2, 2)
    self.conv2 = nn.Conv2d(6, 16, 5)
    self.fc1 = nn.Linear(16 * 5 * 5, 120)
    self.fc2 = nn.Linear(120, 84)
    self.fc3 = nn.Linear(84, 10)

  def forward(self, x):
    x = self.pool(F.relu(self.conv1(x)))
    x = self.pool(F.relu(self.conv2(x)))
    x = x.view(-1, 16 * 5 * 5)
    x = F.relu(self.fc1(x))
    x = F.relu(self.fc2(x))
    x = self.fc3(x)
    return x


net = Net()
```

3.6. Xác định hàm optimizer và hàm loss function

Hàm loss function được sử dụng là cross-entropy thông qua class `nn.CrossEntropyLoss()` và phương pháp optimizer là stochastic gradient descent của module `torch.optim` . Chi tiết về hàm loss function và phương pháp optimize đã quá quen thuộc, các bạn có thể lên google search một vài bài báo để hiểu rõ.

```
import torch.optim as optim

criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr = 0.001, momentum = 0.1)
```

3.7. Huấn luyện model

Chúng ta sẽ khởi tạo một vòng loop bao gồm 2 epochs trong đó mỗi một epochs sẽ truyền toàn bộ các data iterator như đầu vào của mạng nơ ron. Bên trong mỗi epoch chúng ta xác định:

- output của mô hình.
- hàm loss function.
- phương pháp optimize.
- thực hiện quá trình feed forward.

Mọi thứ diễn ra khá đơn giản theo như code bên dưới

Top

```
for epoch in range(2): # loop over the dataset multiple times
  running_loss = 0.0
  for i, data in enumerate(trainloader, 0):
    # get the inputs, data is a list of [inputs, labels]
    inputs, labels = data

    # zero the parameter gradients
    optimizer.zero_grad()

    # forward + backward + optimize
    outputs = net(inputs)
    loss = criterion(outputs, labels)
    loss.backward()
    optimizer.step()

    # print statistics
    running_loss += loss.item()
    if i % 2000 == 1999: # print every 2000 mini-batches
      print('[%d, %5d] loss: %.3f' %
           (epoch + 1, i + 1, running_loss/2000))
      running_loss = 0.0

print('Finished Training')
```

```
[1,  2000] loss: 2.305
[1,  4000] loss: 2.300
[1,  6000] loss: 2.296
[1,  8000] loss: 2.283
[1, 10000] loss: 2.223
[1, 12000] loss: 2.094
[2,  2000] loss: 2.002
[2,  4000] loss: 1.930
[2,  6000] loss: 1.876
[2,  8000] loss: 1.821
[2, 10000] loss: 1.774
[2, 12000] loss: 1.755
Finished Training
```

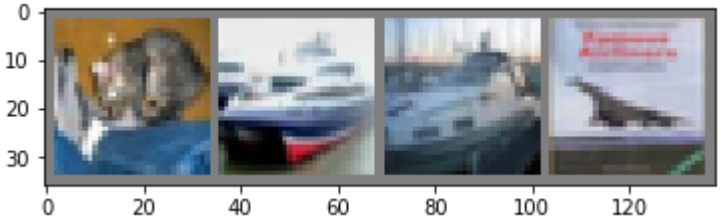
3.8. Kiểm tra network trên tập data test

Như vậy chúng ta đã hoàn thành 2 lượt huấn luyện dữ liệu trên toàn bộ tập training dataset. Sau đây chúng ta cần kiểm tra xem kết quả mô hình sau huấn luyện như thế nào trên dữ liệu test dataset.

```
# Hiển thị một vài dữ liệu
# Sử dụng hàm iter để biến testloader thành 1 iterator, từ đó có thể l
dataiter = iter(testloader)
images, labels = dataiter.next()

# print image
imshow(torchvision.utils.make_grid(images))
print('GroundTruth: {}'.format(' '.join('%5s' % classes[labels[j]] for
```

Top



</img> GroundTruth: cat ship ship plane

Khác với tensorflow khi dự báo chúng ta cần phải sử dụng hàm predict. Để dự báo nhãn cho tập data test chúng ta chỉ cần truyền raw data vào object `net` . Mô hình sẽ tự động thực hiện một quá trình lan truyền thuận và tính ra phân phối xác xuất ở đầu ra.

```
outputs = net(images)
print(type(outputs))
print(outputs.shape)
```

```
<class 'torch.Tensor'>
torch.Size([4, 10])
```

Lấy ra nhãn dự báo dựa vào xác xuất lớn nhất của phân phối xác xuất đầu ra.

```
_,  predicted = torch.max(outputs, 1)
print('Predicted: ', ' '.join('%5s' % classes[predicted[j]]
                               for j in range(4)))
```

```
Predicted:    cat  ship  ship  ship
```

Dự báo trên 4 quan sát đầu tiên cho thấy đúng 3 sai 1. Kiểm tra trên toàn bộ các quan sát.

```
print(type(labels))
print(type(images))

print(labels.shape)
print(images.shape)
```

```
<class 'torch.Tensor'>
<class 'torch.Tensor'>
torch.Size([4])
torch.Size([4, 3, 32, 32])
```

Top

```
correct = 0
total = 0
with torch.no_grad():
    for data in testloader:
        images, labels = data
        outputs = net(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

print('Accuracy of the network on the 10000 test images: %d %%' % (
    100 * correct / total))
```

```
Accuracy of the network on the 10000 test images: 37 %
```

Kiểm tra mức độ chính xác trên từng class một.

```
class_correct = list(0. for i in range(10))
class_total = list(0. for i in range(10))
with torch.no_grad():
    for data in testloader:
        images, labels = data
        outputs = net(images)
        _, predicted = torch.max(outputs, 1)
        c = (predicted == labels).squeeze()
        for i in range(4):
            label = labels[i]
            class_correct[label] += c[i].item()
            class_total[label] += 1


for i in range(10):
    print('Accuracy of %5s : %2d %%' % (
        classes[i], 100 * class_correct[i] / class_total[i]))
```

```
Accuracy of plane : 46 %
Accuracy of   car : 42 %
Accuracy of  bird : 31 %
Accuracy of   cat : 16 %
Accuracy of  deer : 14 %
Accuracy of   dog : 20 %
Accuracy of  frog : 56 %
Accuracy of horse : 53 %
Accuracy of  ship : 39 %
Accuracy of truck : 49 %
```

3.9. Huấn luyện model trên GPU

Đầu tiên xác định cuda device nếu chúng thực sự tồn tại.

Top

```
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu"
print(device)
```

```
cuda:0
```

Để đưa model lên device chúng ta sẽ convert chúng thành parameters và lưu trữ chúng lên buffer của CUDA.

```
net.to(device)
```

```
Net(
  (conv1): Conv2d(3, 6, kernel_size=(5, 5), stride=(1, 1))
  (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ce
  (conv2): Conv2d(6, 16, kernel_size=(5, 5), stride=(1, 1))
  (fc1): Linear(in_features=400, out_features=120, bias=True)
  (fc2): Linear(in_features=120, out_features=84, bias=True)
  (fc3): Linear(in_features=84, out_features=10, bias=True)
)
```

Hãy nhớ rằng bạn phải gửi inputs và targets tại mỗi bước huấn luyện lên GPU:

```
print(data[0].shape)
print(data[1].shape)
```

```
torch.Size([4, 3, 32, 32])
torch.Size([4])
```

```
inputs, labels = data[0].to(device), data[1].to(device)
```

4. Tài liệu tham khảo

1. Pytorch layer (https://pytorch.org/docs/stable/nn.html)
2. Xây dựng mạng convolutional neural network trên pytorch (https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html)
3. Dataset pytorch (https://pytorch.org/docs/stable/data.html)
4. Xử lý trên GPU - pytorch (https://pytorch.org/tutorials/beginner/blitz/data_parallel_tutorial.html)

Top