PyTorch Tutorials

PyTorch

0.4.0

Search docs

BEGINNER TUTORIALS

⊟ Deep Learning with PyTorch: A 60 Minute Blitz

⊞ What is PyTorch?

⊟ Autograd: automatic differentiation

Tensor

Gradients

⊞ Neural Networks

⊞ Training a classifier

⊞ Optional: Data Parallelism

⊞ PyTorch for former Torch users

⊞ Learning PyTorch with Examples

⊞ Transfer Learning tutorial

⊞ Data Loading and Processing Tutorial

⊞ Deep Learning for NLP with Pytorch

INTERMEDIATE TUTORIALS

⊞ Classifying Names with a Character-Level RNN

⊞ Generating Names with a Character-Level RNN

⊞ Translation with a Sequence to Sequence Network and Attention

⊞ Reinforcement Learning (DQN) tutorial

⊞ Writing Distributed Applications with PyTorch

⊞ Spatial Transformer Networks Tutorial

ADVANCED TUTORIALS

Autograd: automatic differentiation

Central to all neural networks in PyTorch is the `autograd` package. Let's first briefly visit this, and we will then go to training our first neural network.

The `autograd` package provides automatic differentiation for all operations on Tensors. It is a define-by-run framework, which means that your backprop is defined by how your code is run, and that every single iteration can be different.

Let us see this in more simple terms with some examples.

Tensor

`torch.Tensor` is the central class of the package. If you set its attribute `.requires_grad` as `True`, it starts to track all operations on it. When you finish your computation you can call `.backward()` and have all the gradients computed automatically. The gradient for this tensor will be accumulated into `.grad` attribute.

To stop a tensor from tracking history, you can call `.detach()` to detach it from the computation history, and to prevent future computation from being tracked.

To prevent tracking history (and using memory), you can also wrap the code block in `with torch.no_grad():`. This can be particularly helpful when evaluating a model because the model may have trainable parameters with *requires_grad=True*, but we don't need the gradients.

There's one more class which is very important for autograd implementation - a `Function`.

`Tensor` and `Function` are interconnected and build up an acyclic graph, that encodes a complete history of computation. Each variable has a `.grad_fn` attribute that references a `Function` that has created the `Tensor` (except for Tensors created by the user - their `grad_fn is None`).

If you want to compute the derivatives, you can call `.backward()` on a `Tensor`. If `Tensor` is a scalar (i.e. it holds a one element data), you don't need to specify any arguments to `backward()`, however if it has more elements, you need to specify a `gradient` argument that is a tensor of matching shape.

```
import torch
```

Create a tensor and set requires_grad=True to track computation with it

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

Out:

```
tensor([[ 1.,  1.],
        [ 1.,  1.]])
```

Do an operation of tensor:

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

Out:

```
tensor([[ 3.,  3.],
        [ 3.,  3.]])
```

`y` was created as a result of an operation, so it has a `grad_fn`.

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

Out:

```
<AddBackward0 object at 0x7fb9f73ea518>
```

Do more operations on y

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

print(z, out)
```

Out:

```
tensor([[ 27.,  27.],
        [ 27.,  27.]]) tensor(27.)
```

`.requires_grad_( ... )` changes an existing Tensor's `requires_grad` flag in-place. The input flag defaults to `True` if not given.

```
a = torch.randn(2, 2)
a = ((a * 3) / (a - 1))
print(a.requires_grad)
a.requires_grad_(True)
print(a.requires_grad)
b = (a * a).sum()
print(b.grad_fn)
```

Out:

```
False
True
<SumBackward0 object at 0x7fb9f74f1a58>
```

Gradients

Let's backprop now Because `out` contains a single scalar, `out.backward()` is equivalent to `out.backward(torch.tensor(1))`.

```
out.backward()
```

print gradients d(out)/dx

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

Out:

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

You should have got a matrix of `4.5`. Let's call the `out` *Tensor* "o". We have that $o = \frac{1}{4}\sum_i z_i$, $z_i = 3(x_i+2)^2$ and $z_i\bigr\rvert_{x_i=1} = 27$. Therefore, $\frac{\partial o}{\partial x_i} = \frac{3}{2}(x_i+2)$, hence $\frac{\partial o}{\partial x_i}\bigr\rvert_{x_i=1} = \frac{9}{2} = 4.5$.

You can do many crazy things with autograd!

```
x = torch.randn(3, requires_grad=True)

y = x * 2
while y.data.norm() < 1000:
    y = y * 2

print(y)
```

Out:

```
tensor([-590.4467,   97.6760,  921.0221])
```

```
gradients = torch.tensor([0.1, 1.0, 0.0001], dtype=torch.float)
y.backward(gradients)

print(x.grad)
```

Out:

```
tensor([  51.2000,  512.0000,    0.0512])
```

You can also stops autograd from tracking history on Tensors with requires_grad=True by wrapping the code block in `with torch.no_grad():`

```
print(x.requires_grad)
print((x ** 2).requires_grad)

with torch.no_grad():
    print((x ** 2).requires_grad)
```

Out:

```
True
True
False
```

Read Later:

Documentation of `autograd` and `Function` is at http://pytorch.org/docs/autograd

Total running time of the script: (0 minutes 0.138 seconds)

Download Python source code: autograd_tutorial.py

Download Jupyter notebook: autograd_tutorial.ipynb

Gallery generated by Sphinx-Gallery

Previous

Next

Built with Sphinx using a theme provided by Read the Docs.