Introduction to Deep Learning (CS474)

Lecture 7

Rupam

Outline

- Mechanics of Learning-PART II
    - Pytorch’s autograd

Introduction

- In our little adventure, we just saw a simple example of backpropagation: we computed the gradient of a composition of functions—the model and the loss—with respect to their innermost parameters (**`w`** and **`b`**) by propagating derivatives backward using the chain rule.

- The basic requirement here is that **all functions** we're dealing with can be **differentiated** analytically.

- Writing the analytical expression for the derivatives of a very deep composition of linear and nonlinear functions is <u>not a lot of fun.</u>

Slide credit: E. STEVENS, L. ANTIGA, and T. VIEHMANN

Introduction

- This is when PyTorch tensors come to the rescue, with a PyTorch component called **autograd**.

- We left out one very interesting aspect, however: PyTorch tensors can remember where they come from, in terms of the operations and parent tensors that originated them, and they can automatically provide the chain of derivatives of such operations with respect to their inputs.

- This means we won't need to derive our model by **hand**; given a forward expression, no matter how nested, PyTorch will automatically provide the gradient of that expression with respect to its input parameters.

Slide credit: E. STEVENS, L. ANTIGA, and T. VIEHMANN

Example

```python
import numpy as np
import torch

t_c = torch.tensor([0.5, 14.0, 15.0, 28.0, 11.0, 8.0,
                    3.0, -4.0, 6.0, 13.0, 21.0])
t_u = torch.tensor([35.7, 55.9, 58.2, 81.9, 56.3, 48.9,
                    33.9, 21.8, 48.4, 60.4, 68.4])
t_un = 0.1 * t_u

def model(t_u, w, b):
    return w * t_u + b

def loss_fn(t_p, t_c):
    squared_diffs = (t_p - t_c)**2
    return squared_diffs.mean()
```

Slide credit: E. STEVENS, L. ANTIGA, and T. VIEHMANN

Rupam

Example: Applying AUTOGRAD

Continuing with the earlier notebook:

```
params = torch.tensor([1.0, 0.0], requires_grad=True)
```

- Notice the `requires_grad=True` argument to the tensor constructor? That argument is telling PyTorch to track the entire family tree of tensors resulting from operations on params .
- In other words, any tensor that will have params as an ancestor will have access to the chain of functions that were called to get from params to that tensor.
- In case these functions are differentiable (and most PyTorch tensor operations will be), the value of the derivative will be automatically populated as a grad attribute of the params tensor.
- In general, all PyTorch tensors have an attribute named grad .

Slide credit: E. STEVENS, L. ANTIGA, and T. VIEHMANN

Example: Applying AUTOGRAD

Continuing with the earlier notebook:

```
loss = loss_fn(model(t_u, *params), t_c)
loss.backward()

params.grad
```

```
tensor([4517.2969,   82.6000])
```

- We have called the *model* and compute the *loss*, and then call **backward** on the loss tensor.

Slide credit: E. STEVENS, L. ANTIGA, and T. VIEHMANN

Example: Applying AUTOGRAD

- When we compute our loss while the parameters w and b require gradients, in addition to performing the actual computation, PyTorch creates the **autograd graph** with the operations (in black circles) as nodes, as shown in the figure 1.

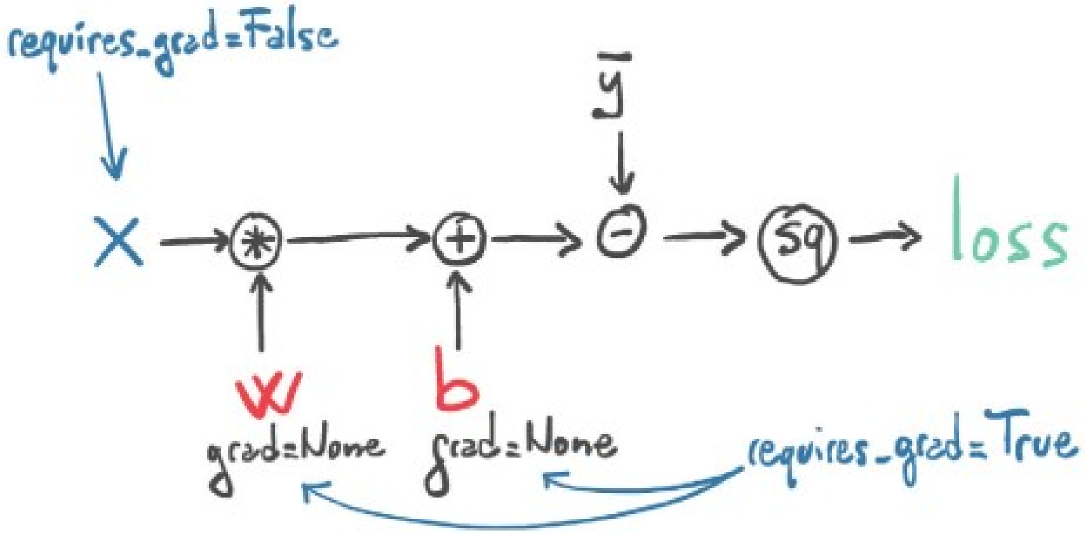

Figure 1

Slide & Image credit: E. STEVENS, L. ANTIGA, and T. VIEHMANN

Example: Applying AUTOGRAD

- When we call **`loss.backward()`**, PyTorch traverses this graph in the ***reverse*** direction to compute the gradients, as shown by the arrows in the figure 2.

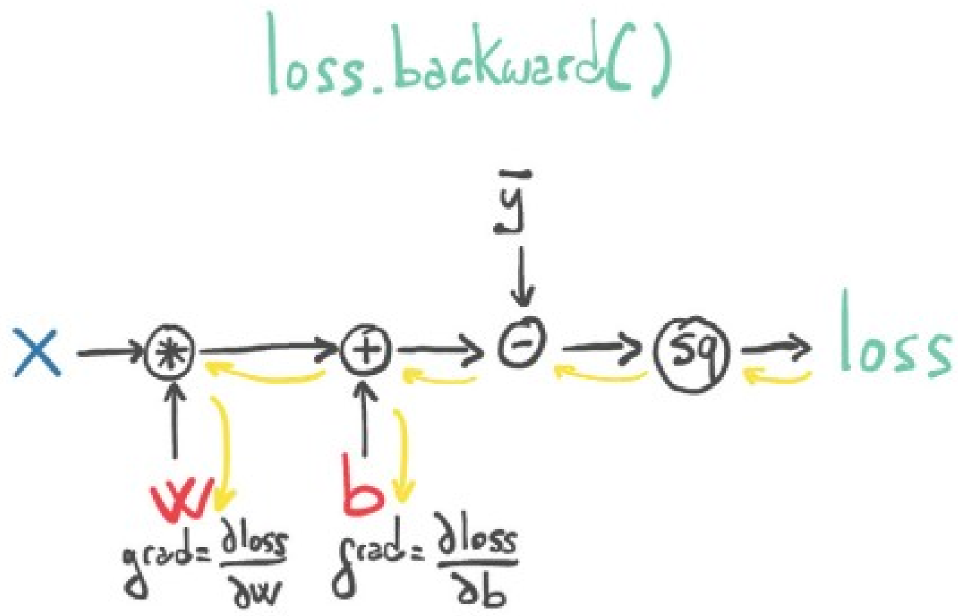

Figure 2

Slide credit: E. STEVENS, L. ANTIGA, and T. VIEHMANN

Rupam

Example: Applying AUTOGRAD

- Calling backward will lead derivatives to accumulate at leaf nodes. We need to zero the gradient explicitly after using it for parameter updates.

- We can do this easily using the in-place zero_ method.

- Therefore, continuing with the earlier *notebook.*

```
if params.grad is not None:
    params.grad.zero_()
```

Slide credit: E. STEVENS, L. ANTIGA, and T. VIEHMANN

Rupam

Example: Applying AUTOGRAD

```
# Our autograd-enabled training code

def training_loop(n_epochs, learning_rate, params, t_u, t_c):
    for epoch in range(1, n_epochs + 1):
        if params.grad is not None:  # <1>
            params.grad.zero_()

        t_p = model(t_u, *params)
        loss = loss_fn(t_p, t_c)
        loss.backward()

        with torch.no_grad():  # <2>
            params -= learning_rate * params.grad

        if epoch % 500 == 0:
            print('Epoch %d, Loss %f' % (epoch, float(loss)))

    return params
```

Slide credit: E. STEVENS, L. ANTIGA, and T. VIEHMANN

Rupam

Example: Applying AUTOGRAD

```python
# Invoking training loop

training_loop(
    n_epochs = 5000,
    learning_rate = 1e-2,
    params = torch.tensor([1.0, 0.0], requires_grad=True),
    t_u = t_un,
    t_c = t_c)
```

```
Epoch 500, Loss 7.860116
Epoch 1000, Loss 3.828538
Epoch 1500, Loss 3.092191
Epoch 2000, Loss 2.957697
Epoch 2500, Loss 2.933134
Epoch 3000, Loss 2.928648
Epoch 3500, Loss 2.927830
Epoch 4000, Loss 2.927679
Epoch 4500, Loss 2.927652
Epoch 5000, Loss 2.927647
tensor([  5.3671, -17.3012], requires_grad=True)
```

Slide credit: E. STEVENS, L. ANTIGA, and T. VIEHMANN

References

- All the contents present in the slides are taken from various online resources. Due credit is given in the respective slides.These slides are used for *academic* purposes only.