PIERIAN DATA

PyTorch Gradients

This section covers the PyTorch **`autograd`** (https://pytorch.org/docs/stable/autograd.html) implementation of gradient descent. Tools include:

- **`torch.autograd.backward()`** (https://pytorch.org/docs/stable/autograd.html#torch.autograd.backward)
- **`torch.autograd.grad()`** (https://pytorch.org/docs/stable/autograd.html#torch.autograd.grad)

Before continuing in this section, be sure to watch the theory lectures to understand the following concepts:

- Error functions (step and sigmoid)
- One-hot encoding
- Maximum likelihood
- Cross entropy (including multi-class cross entropy)
- Back propagation (backprop)

Additional Resources:
PyTorch Notes: (https://pytorch.org/docs/stable/notes/autograd.html) Autograd mechanics

Autograd - Automatic Differentiation

In previous sections we created tensors and performed a variety of operations on them, but we did nothing to store the sequence of operations, or to apply the derivative of a completed function.

In this section we'll introduce the concept of the *dynamic computational graph* which is comprised of all the *Tensor* objects in the network, as well as the *Functions* used to create them. Note that only the input Tensors we create ourselves will not have associated Function objects.

The PyTorch **`autograd`** (https://pytorch.org/docs/stable/autograd.html) package provides automatic differentiation for all operations on Tensors. This is because operations become attributes of the tensors themselves. When a Tensor's `.requires_grad` attribute is set to True, it starts to track all operations on it. When an operation finishes you can call `.backward()` and have all the gradients computed automatically. The gradient for a tensor will be accumulated into its `.grad` attribute.

Let's see this in practice.

Back-propagation on one step

We'll start by applying a single polynomial function $y = f(x)$ to tensor x. Then we'll backprop and print the gradient $\frac{dy}{dx}$.

$$\begin{aligned} Function: \quad & y = 2x^4 + x^3 + 3x^2 + 5x + 1 \\ Derivative: \quad & y' = 8x^3 + 3x^2 + 6x + 5 \end{aligned}$$

Step 1. Perform standard imports

In [1]:

```
import torch
```

Step 2. Create a tensor with `requires_grad` set to True

This sets up computational tracking on the tensor.

In [2]:

```
x = torch.tensor(2.0, requires_grad=True)
```

Step 3. Define a function

In [3]:

```
y = 2*x**4 + x**3 + 3*x**2 + 5*x + 1

print(y)
```

```
tensor(63., grad_fn=<AddBackward0>)
```

Since y was created as a result of an operation, it has an associated gradient function accessible as `y.grad_fn`
The calculation of y is done as:

$$y = 2(2)^4 + (2)^3 + 3(2)^2 + 5(2) + 1 = 32 + 8 + 12 + 10 + 1 = 63$$

This is the value of y when $x = 2$.

Step 4. Backprop

In [4]:

```
y.backward()
```

Step 5. Display the resulting gradient

In [5]:

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

```
tensor(93.)
```

Note that `x.grad` is an attribute of tensor x, so we don't use parentheses. The computation is the result of

$$y' = 8(2)^3 + 3(2)^2 + 6(2) + 5 = 64 + 12 + 12 + 5 = 93$$

This is the slope of the polynomial at the point $(2, 63)$.

Back-propagation on multiple steps

Now let's do something more complex, involving layers y and z between x and our output layer out.

1. Create a tensor

In [6]:

```
x = torch.tensor([[1.,2,3],[3,2,1]], requires_grad=True)
print(x)
```

```
tensor([[1., 2., 3.],
        [3., 2., 1.]], requires_grad=True)
```

2. Create the first layer with $y = 3x + 2$

In [7]:

```
y = 3*x + 2
print(y)
```

```
tensor([[ 5.,  8., 11.],
        [11.,  8.,  5.]], grad_fn=<AddBackward0>)
```

3. Create the second layer with $z = 2y^2$

In [8]:

```
z = 2*y**2
print(z)
```

```
tensor([[ 50., 128., 242.],
        [242., 128.,  50.]], grad_fn=<MulBackward0>)
```

4. Set the output to be the matrix mean

In [9]:

```
out = z.mean()
print(out)
```

```
tensor(140., grad_fn=<MeanBackward1>)
```

5. Now perform back-propagation to find the gradient of x w.r.t out

(If you haven't seen it before, w.r.t. is an abbreviation of *with respect to*)

In [10]:

```
out.backward()
print(x.grad)
```

```
tensor([[10., 16., 22.],
        [22., 16., 10.]])
```

You should see a 2x3 matrix. If we call the final `out` tensor "o", we can calculate the partial derivative of o with respect to x_i as follows:

$o = \frac{1}{6}\sum_{i=1}^{6} z_i$

$z_i = 2(y_i)^2 = 2(3x_i + 2)^2$

To solve the derivative of z_i we use the [chain rule (https://en.wikipedia.org/wiki/Chain_rule)](https://en.wikipedia.org/wiki/Chain_rule), where the derivative of $f(g(x)) = f'(g(x))g'(x)$

In this case

$$\begin{array}{rll} f(g(x)) = 2(g(x))^2, & f'(g(x)) = 4g(x) \\ g(x) = 3x + 2, & g'(x) = 3 \\ \frac{dz}{dx} = 4g(x) \times 3 & = 12(3x + 2) \end{array}$$

Therefore,

$\frac{\partial o}{\partial x_i} = \frac{1}{6} \times 12(3x + 2)$

$\frac{\partial o}{\partial x_i}\bigg|_{x_i=1} = 2(3(1) + 2) = 10$

$\frac{\partial o}{\partial x_i}\bigg|_{x_i=2} = 2(3(2) + 2) = 16$

$\frac{\partial o}{\partial x_i}\bigg|_{x_i=3} = 2(3(3) + 2) = 22$

Turn off tracking

There may be times when we don't want or need to track the computational history.

You can reset a tensor's `requires_grad` attribute in-place using `.requires_grad_(True)` (or False) as needed.

When performing evaluations, it's often helpful to wrap a set of operations in `with torch.no_grad():`

A less-used method is to run `.detach()` on a tensor to prevent future computations from being tracked. This can be handy when cloning a tensor.

A NOTE ABOUT TENSORS AND VARIABLES: Prior to PyTorch v0.4.0 (April 2018) Tensors (`torch.Tensor`) only held data, and tracking history was reserved for the Variable wrapper (`torch.autograd.Variable`). Since v0.4.0 tensors and variables have merged, and tracking functionality is now available through the `requires_grad=True` flag.