# Automatic Differentiation

## Import `autograd` and create a variable

In [1]:
```
from mxnet import autograd, nd

x = nd.arange(4).reshape((4, 1))
print(x)
```

```
[[0.]
 [1.]
 [2.]
 [3.]]
<NDArray 4x1 @cpu(0)>
```

# Attach gradient to `x`

- It allocates memory to store its gradient, which has the same shape as `x`.
- It also tell the system that we need to compute its gradient.

In [2]:
```
x.attach_grad()
x.grad
```

Out[2]:
```
[[0.]
 [0.]
 [0.]
 [0.]]
<NDArray 4x1 @cpu(0)>
```

# Forward

Now compute

$$y = 2\mathbf{x}^\top \mathbf{x}$$

by placing code inside a `with autograd.record():` block. MXNet will build the according computation graph.

In [3]:
```
with autograd.record():
    y = 2 * nd.dot(x.T, x)
y
```

Out[3]:
```
[[28.]]
<NDArray 1x1 @cpu(0)>
```

# Backward

In [4]:
```
y.backward()
```

# Get the gradient

Given $y = 2\mathbf{x}^\top \mathbf{x}$, we know

$$\frac{\partial y}{\partial \mathbf{x}} = 4\mathbf{x}$$

Now verify the result:

In [5]:
```
print((x.grad - 4 * x).norm().asscalar() == 0)
print(x.grad)
```

```
True

[[ 0.]
 [ 4.]
 [ 8.]
 [12.]]
<NDArray 4x1 @cpu(0)>
```

# Backward on non-scalar

`y.backward()` equals to `y.sum().backward()`

In [6]:
```
with autograd.record():
    y = 2 * x * x
print(y.shape)
y.backward()
print(x.grad)
```

```
(4L, 1L)

[[ 0.]
 [ 4.]
 [ 8.]
 [12.]]
<NDArray 4x1 @cpu(0)>
```

# Training Mode and Prediction Mode

The `record` scope will alter the mode by assuming that gradient is only required for training.

It's necessary since some layers, e.g. batch normalization, behavior differently in the training and prediction modes.

In [7]:

```
print(autograd.is_training())
with autograd.record():
    print(autograd.is_training())
```

```
False
True
```

# Computing the Gradient of Python Control Flow

Autograd also works with Python functions and control flows.

In [8]:
```
def f(a):
    b = a * 2
    while b.norm().asscalar() < 1000:
        b = b * 2
    if b.sum().asscalar() > 0:
        c = b
    else:
        c = 100 * b
    return c
```

# Function behaviors depends on inputs

In [9]:
```
a = nd.random.normal(shape=1)
a.attach_grad()
with autograd.record():
    d = f(a)
d.backward()
```

# Verify the results

`f` is piecewise linear in its input `a`. There exists $g$ such as $f(a) = ga$ and $\frac{\partial f}{\partial a} = g$. Verify the result:

In [10]:
```
print(a.grad == (d / a))
```

```
[1.]
<NDArray 1 @cpu(0)>
```

## Head gradients and the chain rule

We can break the chain rule manually. Assume $\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y}\frac{\partial y}{\partial x}$. `y.backward()` will only compute $\frac{\partial y}{\partial x}$. To get $\frac{\partial z}{\partial x}$, we can first compute $\frac{\partial z}{\partial y}$, and then pass it as head gradient to `y.backward`.

In [11]:

```
with autograd.record():
    y = x * 2
y.attach_grad()
with autograd.record():
    z = y * x
z.backward()  # y.grad = \partial z / \partial y
y.backward(y.grad)
x.grad == 2*x # x.grad = \partial z / \partial x
```

Out[11]:

```
[[1.]
 [1.]
 [1.]
 [1.]]
<NDArray 4x1 @cpu(0)>
```