TTIC 31230, Fundamentals of Deep Learning

David McAllester, Winter 2019

The EDF Framework

The Core of EDF

```
def Forward():
    for c in CompNodes: c.forward()

def Backward(loss):
    for c in CompNodes + Parameters: c.grad = 0
    loss.grad = 1/nBatch
    for c in CompNodes[::-1]: c.backward()

def SGD():
    for p in Parameters:
        p.SGD()
```

EDF

The educational frameword (EDF) is a simple Python-NumPy implementation of a deep learning framework.

In EDF we write

$$\begin{aligned} y &= F(x) \\ z &= G(y, x) \\ u &= H(z) \\ \mathcal{L} &= u \end{aligned}$$

This is Python code where variables are bound to objects (inputs, parameters and compnodes).

The EDF Framework

$$\begin{aligned} y &= F(x) \\ z &= G(y, x) \\ u &= H(z) \\ \mathcal{L} &= u \end{aligned}$$

This is Python code where variables are bound to objects.

x is an object in the class `Input`.

y is an object in the class F (subclass of `CompNode`).

z is an object in the class G (subclass of `CompNode`).

u and $\mathcal{L}$ are the same object in the class H (suclass of `CompNode`).

$$y = F(x)$$

```
class $F$(CompNode):

    def __init__(self, x):
        CompNodes.append(self)
        self.x = x

    def forward(self):
        self.value = f(self.x.value)

    def backward(self):
        self.x.addgrad(self.grad $*\ \nabla_x\ f(x)$)          #needs x.value
```

Nodes of the Computation Graph

There are three kinds of nodes in a computation graph — inputs, parameters and computation nodes.

```
class Input:
    def __init__(self):
        pass
    def addgrad(self, delta):
        pass

class CompNode: #initialization is handled by the subclass
    def addgrad(self, delta):
        self.grad += delta
```

```
class Parameter:

    def __init__(self,value):
        Parameters.append(self)
        self.value = value

    def addgrad(self, delta):
        #sums over the minibatch
        self.grad += np.sum(delta, axis = 0)

    def SGD(self):
        self.value -= learning_rate*self.grad
```

MLP in EDF

The following Python code constructs the computation graph of a multi-layer perceptron (NLP) with one hidden layer.

```
L1 = Relu(Affine(Phi1,x))
Q = Softmax(Sigmoid(Affine(Phi2,L1))
ell = LogLoss(Q,y)
```

Here `x` and `y` are input computation nodes whose value have been set. Here `Phi1` and `Phi2` are “parameter packages” (a matrix and a bias vector in this case). We have computation node classes `Affine`, `Relu`, `Sigmoid`, `LogLoss` each of which has a forward and a backward method.

```
class Affine(CompNode):

    def __init__(self,Phi,x):
        CompNodes.append(self)
        self.x = x
        self.Phi = Phi

    def forward(self):
        self.value = (np.matmul(self.x.value,
                                self.Phi.w.value)
                          + self.Phi.b.value)
```

```
def backward(self):

    self.x.addgrad(
       np.matmul(self.grad,
                 self.Phi.w.value.transpose()))

    self.Phi.b.addgrad(self.grad)

    self.Phi.w.addgrad(self.x.value[:,:,np.newaxis]
                       * self.grad[:,np.newaxis,:])
```

forward and backward must handle minibatching

The forward and backward methods must be written to handle minibatching. We will consider some examples.

An MLP

```
L1 = Relu(Affine(Phi1,x))
P = Softmax(Sigmoid(Affine(Phi2,L1))
ell = LogLoss(P,y)
```

The Sigmoid Class

The Sigmoid and Relu classes just work.

```
class Sigmoid:
    def __init__(self,x):
        CompNodes.append(self)
        self.x = x

    def forward(self):
        self.value = 1. / (1. + np.exp(-self.x.value))

    def backward(self):
        self.x.grad += self.grad
                       * self.value
                       * (1.-self.value)
```

```
y = Affine([W,B],x)

 forward:
  y.value[b,j] = x.value[b,i]W.value[i,j]
  y.value[b,j] += B.value[j]

 backward:
    x.grad[b,i] += y.grad[b,j]W.value[i,j]
    W.grad[i,j] += y.grad[b,j]x.value[i]
    B.grad[j] += y.grad[b,j]
```

```
class Affine(CompNode):

    def __init__(self,Phi,x):
        CompNodes.append(self)
        self.x = x
        self.Phi = Phi

    def forward(self):
        # y.value[b,j] = x.value[b,i]W.value[i,j]
        # y.value[b,j] += B.value[j]
        self.value = (np.matmul(self.x.value,
                                self.Phi.W.value)
                      + self.Phi.B.value)
```

```
def backward(self):

    self.x.addgrad(
       # x.grad[b,i] += y.grad[b,j]W.value[i,j]
       np.matmul(self.grad,
                 self.Phi.W.value.transpose()))

    # B.grad[j] += y.grad[b,j]
    self.Phi.B.addgrad(self.grad)

    # W.grad[i,j] += y.grad[b,j]x.value[b,i]
    self.Phi.W.addgrad(self.x.value[:,:,np.newaxis]
                       * self.grad[:,np.newaxis,:])
```

NumPy: Reshaping Tensors

For an ndarray x (tensor) we have that x.shape is a tuple of dimensions. The product of the dimensions is the number of numbers.

In NumPy an ndarray (tensor) x can be reshaped into any shape with the same number of numbers.

NumPy: Broadcasting

Shapes can contain dimensions of size 1.

Dimensions of size 1 are treated as “wild card” dimensions in operations on tensors.

$$\begin{aligned}
x.\mathrm{shape} &= (5,1) \\
y.\mathrm{shape} &= (1,10) \\
z &= x * y \\
z.\mathrm{shape} &= (5,10) \\
z[i,j] &= x[i,0] * y[0,j]
\end{aligned}$$

```
class Affine(CompNode):
    ...
    def backward(self):
        ...
        # W.grad[i,j] += y.grad[b,j]x.value[b,i]
        self.Phi.W.addgrad(self.grad[:,np.newaxis,:]
                           *self.x.value[:,:,np.newaxis])

class Parameter:
    ...
    def addgrad(self, delta):
        self.grad += np.sum(delta, axis = 0)
```

NumPy: Broadcasting

When a scalar is added to a matrix the scalar is reshaped to shape $(1, 1)$ so that it is added to each element of the matrix.

When a vector of shape (k) is added to a matrix the vector is reshaped to $(1, k)$ so that it is added to each row of the matrix.

In general when two tensors of different order (number of dimensions) are added, unit dimensions are prepended to the shape of the tensor of smaller order to make the orders match.

END