

TTIC 31230, Fundamentals of Deep Learning

David McAllester, Winter 2019

The EDF Framework

The Educational Framework (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 (subclass of **CompNode**).

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()
```

$$y = F(x)$$

```
class  $F$ (CompNode):
```

```
    def __init__(self, x):
```

```
        CompNodes.append(self)
```

```
        self.x = x
```

```
    def forward(self):
```

```
        self.value = ... compute the value ...
```

```
    def backward(self):
```

```
        self.x.addgrad(... compute the gradient ...)
```

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 = Sigmoid(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.

The Sigmoid Class

$$y[b, i] = \sigma(x[b, i])$$

$$y = \frac{1}{1 + e^{-x}}$$

$$\begin{aligned} \frac{dy}{dx} &= \frac{e^{-x}}{(1 + e^{-x})^2} \\ &= y(1 - y) \end{aligned}$$

$$x.\text{grad}[b, i] += y.\text{grad}[b, i]y.\text{value}[b, i](1 - y.\text{value}[b, i])$$

The Sigmoid Class

```
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.addgrad(self.grad*self.value*(1.-self.value))
```

The Affine Class

$$\tilde{y}[b, j] = \sum_i W[j, i] x[b, i]$$
$$y[b, j] = \tilde{y}[b, j] + B[j]$$

$$\tilde{y}.\text{grad}[b, j] = y.\text{grad}[b, j] \text{ internal to } \mathbf{Affine}$$

$$B.\text{grad}[j] += \sum_b y.\text{grad}[b, j]$$

$$x.\text{grad}[b, i] += \sum_j y.\text{grad}[b, j] W[i, j]$$

$$W.\text{grad}[i, j] += \sum_b y.\text{grad}[b, j] x[b, i]$$

```
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, :])
```

Procedures in EDF

```
def MLP(Phi,x)

    if len(Phi) = 0
        return x

    return Sigmoid(Affine(Phi[0],MLP(Phi[1:],x)))
```

END