TTIC 31230, Fundamentals of Deep Learning

David McAllester, Winter 2018

Computation Graphs and Backpropagation

The Educational Framework (EDF)

Minibatching

Explicit Index Notation

Computation Graphs

A computation graph (sometimes called a “computational graph”) is a sequence of assignment statements.

$$
\begin{aligned}
L^0[j] &= \mathrm{Relu}\left(\left(\sum_i W^0[j,i]\,x[i]\right) + b^0[j]\right)\\
L^1[\hat{y}] &= \sigma\left(\left(\sum_j W^1[\hat{y},j]\,L^0[j]\right) + b^1[\hat{y}]\right)\\
P(\hat{y}) &= \frac{1}{Z}\,e^{L^1[\hat{y}]}
\end{aligned}
$$

Computation Graphs

A simpler example:

$$\ell = \sqrt{x^2 + y^2}$$

$$\begin{aligned} u &= x^2 \\ v &= y^2 \\ r &= u + v \\ \ell &= \sqrt{r} \end{aligned}$$

A computation graph defines a DAG (a directed acyclic graph) where the variables are the nodes. Each assignment determines one or more directed edges from the left hand variable to the right hand variables.

Computation Graphs

$$\begin{aligned} 1.\ u &= x^2 \\ 2.\ w &= y^2 \\ 3.\ r &= u + w \\ 4.\ \ell &= \sqrt{r} \end{aligned}$$

For each variable z we can consider $\partial \ell / \partial z$.

Gradients are computed in the reverse order.

$$\begin{aligned} (4)\ \partial \ell / \partial r &= \frac{1}{2\sqrt{r}} \\ (3)\ \partial \ell / \partial u &= \partial \ell / \partial r \\ (3)\ \partial \ell / \partial w &= \partial \ell / \partial r \\ (2)\ \partial \ell / \partial y &= (\partial \ell / \partial w) * (2y) \\ (1)\ \partial \ell / \partial x &= (\partial \ell / \partial u) * (2x) \end{aligned}$$

A More Abstract Example

$$\begin{aligned} y &= f(x) \\ z &= g(y, x) \\ u &= h(z) \\ \ell &= u \end{aligned}$$

For now assume all values are scalars.

We will “backpopagate” the assignments the reverse order.

Backpropagation

$$\begin{aligned} y &= f(x) \\ z &= g(y, x) \\ u &= h(z) \\ \ell &= u \end{aligned}$$

$\partial\ell/\partial u = 1$

Backpropagation

$$\begin{aligned} y &= f(x) \\ z &= g(y, x) \\ u &= h(z) \\ \ell &= u \end{aligned}$$

$\partial \ell / \partial u = 1$

$\partial \ell / \partial z = (\partial \ell / \partial u)\ (\partial h / \partial z)$ (this uses the value of z)

Backpropagation

$$\begin{aligned} y &= f(x) \\ z &= g(y, x) \\ u &= h(z) \\ \ell &= u \end{aligned}$$

$\partial\ell/\partial u = 1$

$\partial\ell/\partial z = (\partial\ell/\partial u)\ (\partial h/\partial z)$

$\partial\ell/\partial y = (\partial\ell/\partial z)\ (\partial g/\partial y)$ (this uses the value of y and x)

Backpropagation

$$\begin{aligned} y &= f(x) \\ z &= g(y, x) \\ u &= h(z) \\ \ell &= u \end{aligned}$$

$\partial \ell / \partial u = 1$

$\partial \ell / \partial z = (\partial \ell / \partial u)\ (\partial h / \partial z)$

$\partial \ell / \partial y = (\partial \ell / \partial z)\ (\partial g / \partial y)$

$\partial \ell / \partial x = ???$ Oops, we need to add up multiple occurrences.

Backpropagation

$$\begin{aligned} y &= f(x) \\ z &= g(y, x) \\ u &= h(z) \\ \ell &= u \end{aligned}$$

We let $x.\mathrm{grad}$ be an attribute (as in Python) of node x.

We will initialize $x.\mathrm{grad}$ to zero.

During backpropagation we will accumulate contributions to $\partial\ell/\partial x$ into $x.\mathrm{grad}$.

Backpropagation

$$\begin{aligned} y &= f(x) \\ z &= g(y,x) \\ u &= h(z) \\ \ell &= u \end{aligned}$$

$z.\mathrm{grad} = y.\mathrm{grad} = x.\mathrm{grad} = 0$

$u.\mathrm{grad} = 1$

Loop Invariant: For any variable w whose definition has not yet been processed we have that $w.\mathrm{grad}$ is $\partial\ell/\partial w$ as defined by the set of assignments already processed.

Backpropagation

$$\begin{aligned} y &= f(x) \\ z &= g(y,x) \\ u &= h(z) \\ \ell &= u \end{aligned}$$

$z.\mathrm{grad} = y.\mathrm{grad} = x.\mathrm{grad} = 0$

$u.\mathrm{grad} = 1$

Loop Invariant: For any variable w whose definition has not yet been processed we have that $w.\mathrm{grad}$ is $\partial\ell/\partial w$ as defined by the set of assignments already processed.

$z.\mathrm{grad}$ **+=** $u.\mathrm{grad} * \partial h/\partial z$

Backpropagation

$$\begin{aligned} y &= f(x) \\ z &= g(y,x) \\ u &= h(z) \\ \ell &= u \end{aligned}$$

$z.\mathrm{grad} = y.\mathrm{grad} = x.\mathrm{grad} = 0$

$u.\mathrm{grad} = 1$

Loop Invariant: For any variable w whose definition has not yet been processed we have that $w.\mathrm{grad}$ is $\partial\ell/\partial w$ as defined by the set of assignments already processed.

$z.\mathrm{grad}$ **+=** $u.\mathrm{grad} * \partial h/\partial z$

$y.\mathrm{grad}$ **+=** $z.\mathrm{grad} * \partial g/\partial y$

$x.\mathrm{grad}$ **+=** $z.\mathrm{grad} * \partial g/\partial x$

Backpropagation

$$\begin{aligned} y &= f(x) \\ z &= g(y, x) \\ u &= h(z) \\ \ell &= u \end{aligned}$$

$z.\mathrm{grad} = y.\mathrm{grad} = x.\mathrm{grad} = 0$

$u.\mathrm{grad} = 1$

$z.\mathrm{grad}$ **+=** $u.\mathrm{grad} * \partial h / \partial z$

$y.\mathrm{grad}$ **+=** $z.\mathrm{grad} * \partial g / \partial y$

$x.\mathrm{grad}$ **+=** $z.\mathrm{grad} * \partial g / \partial x$

$x.\mathrm{grad}$ **+=** $y.\mathrm{grad} * \partial f / \partial x$

The Vector-Valued Case

$$\begin{aligned} y &= f(x) \\ z &= g(y, x) \\ u &= h(z) \\ \ell &= u \end{aligned}$$

Now suppose the variables can be vector-valued.

The loss ℓ is still a scalar.

In this case

$$x.\mathrm{grad} = \nabla_x \, \ell$$

These are now vectors of the same dimension with

$$x.\mathrm{grad}[i] = \partial \ell / \partial x[i]$$

The Tensor-Valued Case

$$\begin{aligned} y &= f(x) \\ z &= g(y, x) \\ u &= h(z) \\ \ell &= u \end{aligned}$$

Now suppose the variables can be tensor-valued (the values are multi-dimensional arrays). The loss is still a scalar.

The computation is now a “tensor flow”.

The Tensor-Valued Case

For the tensor case we simply view tensors as a special case of vectors. The indeces of x.grad are the same as the indeces of x.

$$x.\mathrm{grad}.\mathrm{shape} = x.\mathrm{shape}$$

The backpropagation equation for $y = f(x)$ is

$$x.\mathrm{grad}[i_1, \ldots, i_n] = (y.\mathrm{grad})[j_1, \ldots, j_k] \nabla_x f(x)[j_1, \ldots, j_k, i_1, \ldots, i_n]$$

$j_1, \ldots, j_k$ are indeces of y and $i_1, \ldots, i_n$ are indeces of x.

Indeces not on the left of the equation are implicitly summed.

The EDF Framework

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

In EDF we write

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

This is Python code where variables are bound to objects.

The EDF Framework

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

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.

z is an object in the class G.

u and ℓ are the same object in the class H.

$$y = F(x)$$

```
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,:])
```

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()
```

Minibatching

Ther running time of EDF, and of any framework, is greatly improved by minibatching.

Minibatching: We run some number of instances together (or in parallel) and then do a parameter update based on the average gradients of the instances of the batch.

For NumPy minibatching is not so much about parallelism as about making the vector operations larger so that the vector operations dominate the slowness of Python. On a GPU minibatching allows parallelism over the batch elements.

Minibatching

With minibatching each input value and each computed value is actually a batch of values.

We add a batch index as an additional first tensor dimensionfor each input and computed node.

For example, if a given input is a D-dimensional vector then the value of an input node has shape (B, D) where B is size of the minibatch.

Parameters do not have a batch index.

```
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
```

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,:])
```

Explicit Index Notation

$$A[x_1, \ldots, x_n] \mathrel{\texttt{+=}} e[x_1, \ldots, x_n, y_1, \ldots, y_k]$$

abbreviates

for $x_1, \ldots, x_n, y_1, \ldots, y_k :\ A[x_1, \ldots, x_n] \mathrel{\texttt{+=}} e[x_1, \ldots, x_n, y_1, \ldots, y_k]$

$$A[x_1, \ldots, x_n] = e[x_1, \ldots, x_n, y_1, \ldots, y_k]$$

abbreviates

$$\text{for } x_1, \ldots, x_n :\ A[x_1, \ldots, x_n] = 0$$

$$A[x_1, \ldots, x_n] \mathrel{\texttt{+=}} e[x_1, \ldots, x_n, y_1, \ldots, y_k]$$

Affine Transformation with Batch Index

$$\begin{aligned} y[b,i] &\;=\; x[b,j]W[j,i] \\ y[b,i] &\;\texttt{+=}\; B[i] \end{aligned}$$

The Swap Rule for Tensor Products

For backprop swap an input with the output and add “grad”.

$$y[b,i] \;=\; x[b,j]W[j,i]$$

$$x.\mathrm{grad}[b,j] \;\mathsf{+=}\; y.\mathrm{grad}[b,i]W[j,i]$$

$$W.\mathrm{grad}[i,j] \;\mathsf{+=}\; x[b,j]y.\mathrm{grad}[b,i]$$

$$y[b,i] \;\mathsf{+=}\; B[i]$$

$$B.\mathrm{grad}[i] \;\mathsf{+=}\; y.\mathrm{grad}[b,i]$$

The Swap Rule for Functions of Tensor Products

For backprop swap an input with the output and add "grad" and $\dot{f}$.

$$y[b,i] \;=\; f(x[b,j]W[j,i])$$

$$x.\mathrm{grad}[b,j] \;\texttt{+=}\; y.\mathrm{grad}[b,i]\;\dot{f}\;W[j,i]$$

$$W.\mathrm{grad}[i,j] \;\texttt{+=}\; y.\mathrm{grad}[b,i]\;\dot{f}\;x[b,j]$$

Here f is scalar to scalar.

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.

Appendix: The Jacobian Matrix

Consider $y = f(x)$ in vector-valued case.

In the vector-valued case the backpropagation equation is

$$x.\mathrm{grad} \mathrel{\texttt{+=}} (y.\mathrm{grad})^\top \nabla_x f(x)$$

where

$$(\nabla_x f(x))[i,j] = \mathcal{J}[i,j] = \partial f(x)[i]/\partial x[j]$$

The matrix $\mathcal{J}[i,j]$ is the Jacobian of f.

Index Types: j is an x-index and i is a y-index.

END