TTIC 31230, Fundamentals of Deep Learning

David McAllester, Winter 2018

Multiclass Logistic Regression

Multilayer Perceptrons

Stochastic Gradient Descent (SGD)

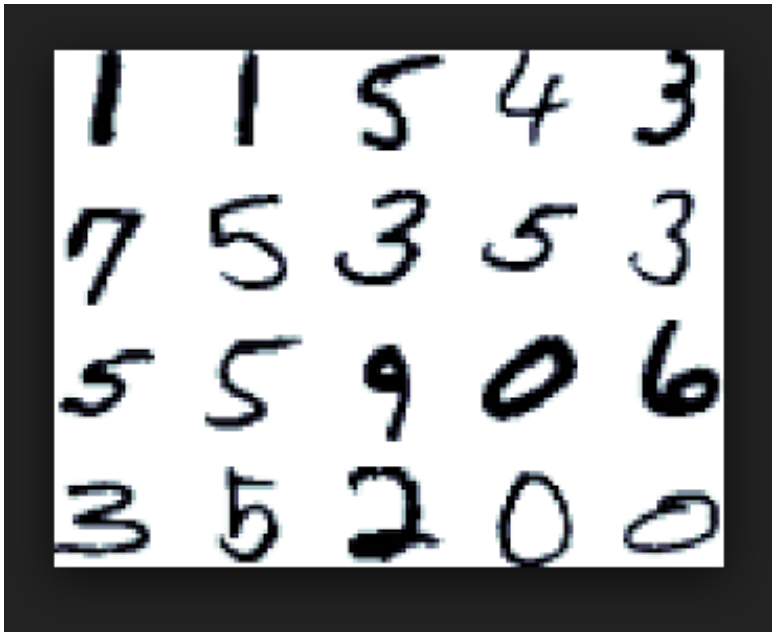
Computation Graphs and Backpropagation

The Educational Framework (EDF)

Minibatching

Multiclass Classification

We consider the problem of taking an input x (such as an image of a hand written digit) and classifying it into some small number of classes (such as the digits 0 through 9).



Multiclass Classification

Assume a population distribution on pairs (x, y) for $x \in \mathbb{R}^d$ and $y \in \mathcal{C}$.

For MNIST x is a 28×28 image which we take to be a 784 dimensional vector giving $x \in \mathbb{R}^{784}$.

For MNIST $\mathcal{C}$ is the set $\{0, \ldots, 9\}$.

Assume a sample $(x_0, y_0), \ \ldots, \ (x_{N-1}, y_{N-1})$ drawn IID from the population.

We want to use the sample to construct a rule for predicting y given x when we draw new pairs from the population.

Multiclass Logistic Regression

Assume a sample $(x_0, y_0), \ \ldots, \ (x_{N-1}, y_{N-1})$ drawn IID from the population with $x \in \mathbb{R}^d$ and $y \in \{0, \ldots, K\}$.

For a new x we compute a score $s(\hat{y})$ for each possible label $\hat{y}$.

$$s = Wx + b$$

Multiclass Logistic Regression for MNIST

j — image pixel

$\hat{y}$ — possible image label (0 through 9)

$$s(\hat{y}) = \sum_j W[\hat{y}, j]\ x[j] + b[\hat{y}]$$

Note that $W[\hat{y}, :]$ is an image.

Softmax

Softmax converts scores (or energies or logits) to probabilities.

$$Q(\hat{y}) = \frac{1}{Z} e^{s(\hat{y})}$$

$$Z = \sum_{\hat{y}} e^{s(\hat{y})}$$

In vector notation

$$Q = \text{softmax } s$$

Log Loss and Logistic Regression

Let $Q_\Phi(\hat{y}|x)$ be defined by a model with parammeters Φ.

In logistic regression Φ is the pair (W, b).

Let n range over training instances.

$$W^*, b^* = \underset{W,b}{\text{argmin}} \frac{1}{N} \sum_{n=1}^{N} -\log Q_{W,b}(y_n|x_n)$$

$$\Phi^* = \underset{\Phi}{\text{argmin}} \frac{1}{N} \sum_{n=1}^{N} -\log Q_{\Phi}(y_n|x_n)$$

Information Theoretic Formulation

Let Φ be the parameters of a probabilistc predictor Q_Φ.

We want $\Phi^* = \text{argmin}_\Phi \ \mathrm{E}_{(x,y)\sim\mathcal{P}}[-\log Q_\Phi(y|x)]$.

This is **cross-entropy** loss:

$$H(\mathcal{P},\mathcal{Q}) = \mathrm{E}_{y\sim\mathcal{P}}[-\log \ \mathcal{Q}(y)]$$

$$H(\mathcal{P}) = H(\mathcal{P},\mathcal{P}) = \mathrm{E}_{y\sim\mathcal{P}}[-\log \ \mathcal{P}(y)]$$

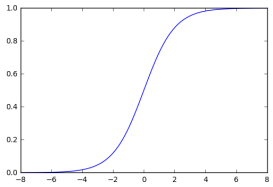
$$H(\mathcal{P},\mathcal{Q}) \geq H(\mathcal{P})$$

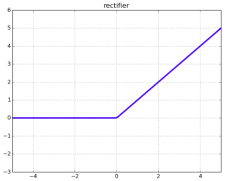
$$\mathrm{E}_{(x,y)\sim\mathcal{P}}[-\log Q_\Phi(y|x)] = \mathrm{E}_{x\sim\mathcal{P}}[H(\mathcal{P}(:|x), Q_\Phi(:|x))]$$

Multi Layer Perceptrons (MLPs)

Activation functions:

$$\sigma(u) = \frac{1}{1+e^{-u}} \qquad \text{Relu}(u) = \max(u, 0)$$





$$L^0 = \text{Relu}(W^0 x + b^0)$$

$$L^1 = \sigma(W^1 L^0 + b^1)$$

$$Q_\Phi = \text{softmax}(L^1)$$

Explicit Index Notation with *Typed Index Variables*

i — pixels

j — image features

$\hat{y}$ — possible image labels

$$L^0[j] = \text{Relu}\left(\left(\sum_i W^0[j,i]\; x[i]\right) + b^0[j]\right)$$

$$L^1[\hat{y}] = \sigma\left(\left(\sum_j W^1[\hat{y},j]\; L^0[j]\right) + b^1[\hat{y}]\right)$$

$$Q_\Phi(\hat{y}) = \frac{1}{Z}\, e^{L^1[\hat{y}]}$$

Loss Vs. Error Rate

While training (gradient descent) is generally done on log loss, performance is often judged by other measures such as error rate.

The "loss" is often used as a synonym for log loss (or whatever loss defined the gradient descent training).

Hence one often reports both "loss" and "error rate".

Note that error rate is not differentiable.

Train Data, Development Data and Test Data

Data is typically divided into **a training set**, **a development set** and **a test set** each drawn IID from the population.

A learning algorithm optimizes training loss.

One then optimizes algorithm design (and hyper-parameters) on the development set. (graduate student descent).

Ultimate performance should be done on a test set not used for development. Test data is often withheld from developers.

Gradients with Respect to Systems of Parameters

$\nabla_\Phi \ell(\Phi, x, y)$ denotes the partial derivative of $\ell(\Phi, x, y)$ with respect to the parameter system Φ.

Here can think of Φ as a single vector with

$$(\nabla_\Phi \ell(\Phi, x, y))_i = \partial \ell(\Phi, x, y)/\partial \Phi_i$$

But in general Φ can be a multi-dimensional array (an ndarray in NumPy). If Φ is four dimensional we can write $\Phi[i, j, k, l]$.

For scalar loss, $\nabla_\Phi \ell(\Phi, x, y)$ has the same shape as Φ.

$$(\nabla_\Phi \ell(\Phi, x, y)) .\text{shape} = \Phi.\text{shape}$$

Total Gradient Descent

$$\ell_{\text{train}}(\Phi) = \frac{1}{N} \sum_n \ell(\Phi, x_n, y_n)$$

We want: $\Phi^* = \text{argmin}_\Phi \, \ell_{\text{train}}(\Phi)$

$$\Phi \quad \texttt{-=} \quad \eta \nabla_\Phi \ell_{\text{train}}(\Phi)$$

Stochastic Gradient Descent (SGD) on the training set.

repeat: Select n at random. Φ `-=` $\eta \, \nabla_\Phi \, \ell(\Phi, x_n, y_n)$

$$\mathrm{E}_n \left[\nabla_\Phi \, \ell(\Phi, x_n, y_n) \right] = \sum_n P(n) \nabla_\Phi \, \ell_{\text{train}}(\Phi, x_n, y_n)$$

$$= \frac{1}{N} \sum_n \nabla_\Phi \, \ell_{\text{train}}(\Phi, x_n, y_n)$$

$$= \nabla_\Phi \frac{1}{N} \sum_n \, \ell_{\text{train}}(\Phi, x_n, y_n)$$

$$= \nabla_\Phi \, \ell_{\text{train}}(\Phi)$$

Epochs

In practice we cycle through the training data visiting each training pair once.

One pass through the training data is called an Epoch.

One typically imposes a random suffle of the training data before each epoch.

SGD for MLPs

i — image pixel $\qquad$ j — image feature $\qquad$ $\hat{y}$ — possible label

$$L^0[j] = \text{Relu}\left(\left(\sum_i W^0[j,i]\, x[i]\right) + b^0[j]\right)$$

$$L^1[\hat{y}] = \sigma\left(\left(\sum_j W^1[\hat{y},j]\, L^0[j]\right) + b^1[\hat{y}]\right)$$

$$P(\hat{y}) = \frac{1}{Z}\, e^{L^1[\hat{y}]}$$

We now need $\frac{\partial \ell(\Phi,x,y)}{\partial \Phi[k]}$ for all k.

Computation Graphs and Backpropagation

Computation Graphs

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

$$L^0[j] = \text{Relu}\left(\left(\sum_i W^0[j,i]\; x[i]\right) + b^0[j]\right)$$

$$L^1[\hat{y}] = \sigma\left(\left(\sum_j W^1[\hat{y},j]\; L^0[j]\right) + b^1[\hat{y}]\right)$$

$$P(\hat{y}) = \frac{1}{Z}\, e^{L^1[\hat{y}]}$$

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.grad be an attribute (as in Python) of node x.

We will initialize x.grad to zero.

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

Backpropagation

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

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

$u.\text{grad} = 1$

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

Backpropagation

$$y = f(x)$$
$$z = g(y, x)$$
$$u = h(z)$$
$$\ell = u$$

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

$u.\text{grad} = 1$

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

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

Backpropagation

$$y = f(x)$$
$$z = g(y, x)$$
$$u = h(z)$$
$$\ell = u$$

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

$u.\text{grad} = 1$

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

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

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

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

Backpropagation

$$y = f(x)$$
$$z = g(y, x)$$
$$u = h(z)$$
$$\ell = u$$

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

$u.\text{grad} = 1$

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

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

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

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

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

$$x.\text{grad} = \nabla_x \ell$$

These are now vectors of the same dimension with

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

The Jacobian Matrix

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

In the vector-valued case the backpropagation equation is

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

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.

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.

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

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

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

$j_1, \ldots, j_k$ 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 "framework" for defining computation graphs and performing backpropagation. 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

The educational frameword (EDF) is a simple Python-NumPy implementation of a "framework" for defining computation graphs and performing backpropagation. 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.

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 * ∇_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,:])
```

END