TTIC 31230, Fundamentals of Deep Learning

David McAllester, Winter 2020

Deep Learning Frameworks

What is a Deep Learning Framework?

A framework provides a high level language for writing models $P_\Phi(y|x)$.

A framework compiles a model into an optimization algorithm.

$$\Phi^* \approx \operatorname*{argmin}_\Phi E_{(x,y)\sim\mathrm{Train}}\; -\ln P_\Phi(y|x)$$

A framework also typically provides support for managing large training sets and pre-trained model parameter values (also called "models").

Some Frameworks

- PyTorch
- Tensorflow
- Keras
- Microsoft Cogntive Toolkit
- Chainer

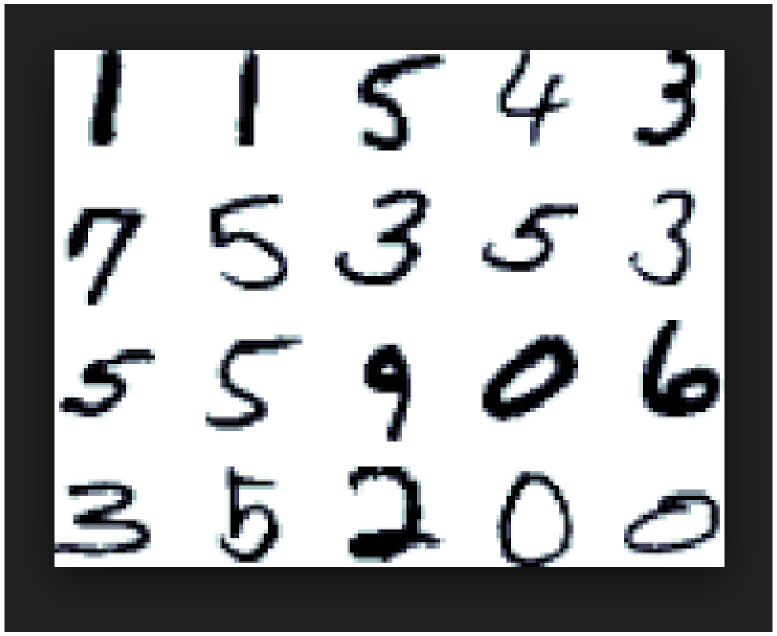
  ⋮

- EDF (Educational Framework in Python/NumPy for earlier versions of this class).

An Example

A Multi-Layer Perceptron (MLP) for MNIST

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 $y \in \{y_1, \ldots, y_k\}$.

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 $k = 10$.

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

A Multi Layer Perceptron (MLP)

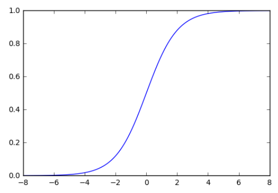
$$\begin{aligned} h &= \sigma\left(W^0 x - b^0\right) \\ s &= \sigma\left(W^1 h - b^1\right) \\ P_\Phi[\hat{y}] &= \operatorname*{softmax}_{\hat{y}}\; s[\hat{y}] \end{aligned}$$

W^1 and W^2 are matrices. b_1 and b_2 are vectors.

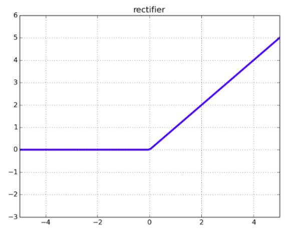
σ is a scalar-to-scalar activation function applied to each component of a vector.

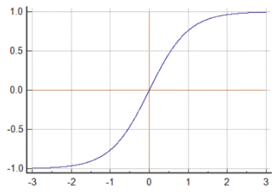
Activation Functions

An activation function $\sigma : \mathbb{R} \to \mathbb{R}$ (scalar-to-scalar) is applied to each component of a vector.

$\sigma(u) = \frac{1}{1+e^{-u}}$ , $\sigma(m) = P(y|m)$ for margin m.

other common activation functions are

$\mathrm{ReLU}(u) = \max(0, u)$ 

, $\tanh(u) = 2\sigma(u) - 1$ 

Stochastic Gradient Descent (SGD)

Once we have specified our model $P_\Phi(y|x)$ in high level equations (such as on the previous two slides) we need to train it.

$$\Phi^* \approx \operatorname*{argmin}_\Phi E_{(x,y)\sim\mathrm{Train}} \;\; -\ln P_\Phi(y|x)$$

The framework generates the training code automatically from the model definition.

Optimization is almost always done with some form of stochastic gradient descent (SGD) and the gradient is computed by backpropagation on the model definition.

$$\Phi^* = \operatorname*{argmin}_{\Phi} E_{(x,y)\sim \mathrm{Train}}\; \mathcal{L}(x, y, \Phi).$$

1. Randomly Initialize Φ (initialization is important and must be done with care).

2. Repeat until "converged":

   - draw $(x, y) \sim$ Train at random.

   - Φ `-=` $\eta \nabla_\Phi\, \mathcal{L}(x, y, \Phi)$

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.

Backpropagation (backprop)

$$h = \sigma\left(W^0 x - b^0\right)$$

$$s = \sigma\left(W^1 h - b^1\right)$$

$$P_\Phi[\hat{y}] = \operatorname*{softmax}_{\hat{y}} \; s[\hat{y}]$$

We now need to automatically compute $\nabla_\Phi \, \mathcal{L}(x, y, \Phi)$ where $\Phi = (W^0,\ b^0,\ W^1,\ b^1)$.

Computation Graphs (Framework Source Code)

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

$$h = \sigma\left(W^0 x - b^0\right)$$

$$s = \sigma\left(W^1 h - b^1\right)$$

$$P_\Phi[\hat{y}] = \operatorname*{softmax}_{\hat{y}}\ s[\hat{y}]$$

I prefer the term "source code" to the term "graph".

Simpler Source Code

The expression

$$\mathcal{L} = \sqrt{x^2 + y^2}$$

can be transformed to the assignment sequence

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

Source Code

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

For each variable z, the derivative $\partial\mathcal{L}/\partial z$ will get computed in reverse order.

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

A More Abstract Example (Still Scalar Values)

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

For now assume all values are scalars (single numbers rather than arrays).

We will "backpopagate" the assignments the reverse order.

Backpropagation (Scalar Values)

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

$$\partial \mathcal{L} / \partial u = 1$$

Backpropagation (Scalar Values)

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

$\partial\mathcal{L}/\partial u = 1$

$\partial\mathcal{L}/\partial z = (\partial h/\partial z)\ (\partial\mathcal{L}/\partial u)$ (this uses the value of z)

Backpropagation (Scalar Values)

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

$\partial \mathcal{L}/\partial u = 1$

$\partial \mathcal{L}/\partial z = (\partial h/\partial z)\ (\partial \mathcal{L}/\partial u)$

$\partial \mathcal{L}/\partial y = (\partial g/\partial y)\ (\partial \mathcal{L}/\partial z)$ (this uses the value of y and x)

Backpropagation (Scalar Values)

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

$\partial\mathcal{L}/\partial u = 1$

$\partial\mathcal{L}/\partial z = (\partial h/\partial z)\ (\partial\mathcal{L}/\partial u)$

$\partial\mathcal{L}/\partial y = (\partial g/\partial y)\ (\partial\mathcal{L}/\partial z)$

$\partial\mathcal{L}/\partial x = ???$ Oops, we need to add up multiple occurrences.

Backpropagation (Scalar Values)

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

Each framework program variable denotes an object (in the sense of C++ or Python).

x.value and x.grad are attributes of the object x.

Values are computed "forward" while gradients are computed "backward".

Backpropagation (Scalar Values)

$$\begin{aligned} y &= f(x) \\ z &= g(y, x) \\ u &= h(z) \\ \mathcal{L} &= u \\ z.\mathrm{grad} &= y.\mathrm{grad} = x.\mathrm{grad} = 0 \\ u.\mathrm{grad} &= 1 \end{aligned}$$

Invariant: The gradients are correct for the red program.

Backpropagation (Scalar Values)

$$\begin{aligned} y &= f(x) \\ z &= g(y,x) \\ u &= h(z) \\ \mathcal{L} &= u \\ z.\mathrm{grad} &= y.\mathrm{grad} = x.\mathrm{grad} = 0 \\ u.\mathrm{grad} &= 1 \\ z.\mathrm{grad} &\mathrel{+}= (\partial h/\partial z) * u.\mathrm{grad} \end{aligned}$$

Invariant: The gradients are correct for the red program.

Backpropagation (Scalar Values)

$$
\begin{aligned}
y &= f(x) \\
z &= g(y, x) \\
u &= h(z) \\
\mathcal{L} &= u \\
z.\mathrm{grad} &= y.\mathrm{grad} = x.\mathrm{grad} = 0 \\
u.\mathrm{grad} &= 1 \\
z.\mathrm{grad} &\mathrel{+}= (\partial h/\partial z) * u.\mathrm{grad} \\
y.\mathrm{grad} &\mathrel{+}= (\partial g/\partial y) * z.\mathrm{grad} \\
x.\mathrm{grad} &\mathrel{+}= (\partial g/\partial x) * z.\mathrm{grad}
\end{aligned}
$$

Backpropagation (Scalar Values)

$$\begin{aligned}
y &= f(x) \\
z &= g(y,x) \\
u &= h(z) \\
\mathcal{L} &= u \\
z.\mathrm{grad} &= y.\mathrm{grad} = x.\mathrm{grad} = 0 \\
u.\mathrm{grad} &= 1 \\
z.\mathrm{grad} &\mathrel{\textbf{+=}} (\partial h/\partial z) * u.\mathrm{grad} \\
y.\mathrm{grad} &\mathrel{\textbf{+=}} (\partial g/\partial y) * z.\mathrm{grad} \\
x.\mathrm{grad} &\mathrel{\textbf{+=}} (\partial g/\partial x) * z.\mathrm{grad} \\
x.\mathrm{grad} &\mathrel{\textbf{+=}} (\partial f/\partial x) * y.\mathrm{grad}
\end{aligned}$$

Handling Arrays

$$
\begin{aligned}
h &= \sigma\left(W^0 x - b^0\right) \\
s &= \sigma\left(W^1 h - b^1\right) \\
P_\Phi[\hat{y}] &= \operatorname*{softmax}_{\hat{y}}\; s[\hat{y}]
\end{aligned}
$$

Each array W is an object with attributes W.value and W.grad.

W.grad is an array with the same indeces as W.value.

An array with more than two indeces is called a tensor.

Einstein Notation

$y = Wx$ abbreviates $y[i] = \sum_j W[i,j]x[j]$.

$y = x^\top W$ abbreviates $y[j] = \sum_i W[i,j]x[i]$.

i is a “row index” and j is a “column index” of W.

Linear algebra suppresses indeces.

Einstein notation uses explicit indeces.

Implicit Summation

Capital letter indeces will be used to indicate subtensors (slices) so that, for example, $M[I,J]$ denotes a matrix while $M[i,j]$ denotes one element of the matrix, $M[i,J]$ denotes the ith row, and $M[I,j]$ denotes the jth collumn.

We will adopt the convention, similar to true Einstein notation, that repeated capital indeces in a product of tensors are implicitly summed. For example, we can write the inner product $e[w,I]^\top h[t,I]$ simply as $e[w,I]h[t,I]$ without the need for the (meaningless) transpose operation.

Implicit Summation

$y = Wx$ abbreviates $y[i] = \sum_j W[i,j]x[j] = W[i,J]x[J]$

$y = x^\top W$ abbreviates $y[j] = \sum_i W[i,j]x[i] = W[I,j]x[I]$

An MLP in Einstein Notation

$$h[j] = \sigma\left(W^0[j,I]\;x[I] - b^0[j]\right)$$

$$s[\hat{y}] = \sigma\left(W^1[\hat{y},J]\;h[J] - b^1[\hat{y}]\right)$$

$$P_\Phi[\hat{y}] = \operatorname*{softmax}_{\hat{y}}\; s[\hat{y}]$$

Think of this as a separate assignment for each $h[j]$ and $s[\hat{y}]$.

Why Einstein Notation?

We will need to work with tensors — arrays with more than two indeces.

For 2D CNNS the weight tensor and the data tensor each have four indeces (including the batch index).

For higher order tensors suppressing indeces becomes confusing.

Einstein went back to explicit index notation (Einstein notation) when working with the higher order tensors in his theory of gravitation.

Why Einstein Notation?

Also, the indeces of tensors generally have types such as a “time index”, “x coordinate”, “y coordinate”, “batch index”, or “feature index”.

Writing a matrix as $W[T, I]$ where T is a time index and I is a feature index makes the type of the matrix W clear and clarifies the order of the indeces (disambiguates W from $W^\top$).

Even when working only with vectors and matrices, Einstein notation is clearer when formulating backpropagation, as we will see.

Backpropagation on Tensor Expressions

Loop Notation

Loop notation assumes all computed tensors are initialized to zero.

$$\begin{array}{lrcl} \text{Einstein}: & \tilde{h}[j] & = & w[j,I]x[I] = \sum_i W[j,i]\,x[i] \\ & h[j] & = & \sigma\left(\tilde{h}[j] - B[j]\right) \end{array}$$

$$\begin{array}{llrcl} \text{Loop}: & \text{for } j & \tilde{h}[j] & = & 0 \\ & \text{for } j,i & \tilde{h}[j] & \mathrel{+}= & W[j,i]x[i] \\ & \text{for } j & h[j] & = & \sigma(\tilde{h}[j] - B[j]) \end{array}$$

Backpropagation on Loop Notation

We backpropagate the body of the loop.

$$\text{for } j, i \;\; \tilde{h}[j] \;\; \texttt{+=} \;\; W[j,i]x[i]$$

The body of the loop is just a product of scalars. A product of scalars has a trivial backpropagation:

$$\text{for } j, i \;\; W.\mathrm{grad}[j,i] \;\; \texttt{+=} \;\; x[i]\tilde{h}.\mathrm{grad}[j]$$

$$x.\mathrm{grad}[i] \;\; \texttt{+=} \;\; W[j,i]\tilde{h}.\mathrm{grad}[j]$$

Minibatching

Training time 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.

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 dimension for each input and computed node.

Parameters do not have a batch index.

Einstein Notation with Minibatching

b— batch index, i — input feature index
j — hidden layer index, $\hat{y}$ — possible label

$$\Phi = (W^0[j,i],\ b^0[j],\ W^1[\hat{y},j],\ b^1[\hat{y}])$$

$$h[b,j] \;=\; \sigma\left(W^0[j,I]\; x[b,I] - b^0[j]\right)$$

$$s[b,\hat{y}] \;=\; \sigma\left(W^1[\hat{y},J]\; h[b,J] - b^1[\hat{y}]\right)$$

$$P_\Phi[b,\hat{y}] \;=\; \operatorname*{softmax}_{\hat{y}}\; s[b,\hat{y}]$$

Backpropagation with Minibatching

$$\text{for } b, i, j \quad \tilde{y}[b,j] \mathrel{+}= W[j,i]\, x[b,i]$$

$$\text{for } b, i, j \quad x.\text{grad}[b,i] \mathrel{+}= W[j,i]\tilde{y}.\text{grad}[b,j]$$

$$W.\text{grad}[j,i] \mathrel{+}= \frac{1}{B}\, x[b,i]\tilde{y}.\text{grad}[b,j]$$

B is the number of batch elements. By convention parameter gradients are averaged over the batch.

The Educational Framework (EDF)

The educational frameword (EDF) is 150 lines of Python-NumPy that implement 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.

The EDF Framework

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

This is Python code.

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`).

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.
    for c in CompNodes[::-1]: c.backward()

def SGD():
    for p in Parameters:
        p.value -= eta*p.grad
```

$$y = F(x)$$

```
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)/nBatch
```

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

$$\begin{aligned} y[b,i] &= \sigma(x[b,i]) \\ y &= \frac{1}{1+e^{-x}} \\ \frac{dy}{dx} &= \frac{e^{-x}}{(1+e^{-x})^2} \\ &= y(1-y) \end{aligned}$$

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

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

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

$$y[b,j] = \tilde{y}[b,j] - B[j] \;\; = \;\; \tilde{y} - B \text{ (broadcasting)}$$

$$\tilde{y}.\mathrm{grad}[b,j] \;\; \texttt{+=} \;\; y.\mathrm{grad}[b,j]$$

$$B.\mathrm{grad}[j] \;\; \texttt{-=} \;\; \frac{1}{B}\sum_b \; y.\mathrm{grad}[b,j]$$

$$x.\mathrm{grad}[b,i] \;\; \texttt{+=} \;\; \sum_j \; \tilde{y}.\mathrm{grad}[b,j] W[i,j] = yW^\top$$

$$W.\mathrm{grad}[i,j] \;\; \texttt{+=} \;\; \frac{1}{B}\sum_b \; \tilde{y}.\mathrm{grad}[b,j] x[b,i] = ???$$

```
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