

TTIC 31230, Fundamentals of Deep Learning

David McAllester, Winter 2019

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.

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

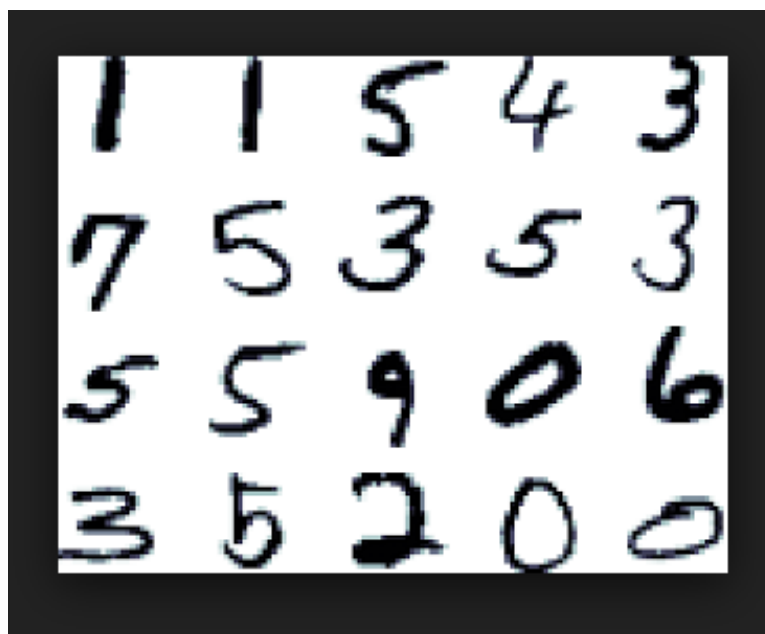
A framework also typically provides support for managing large training sets and pre-trained model parameter values (also called “models”).

Some Frameworks

- Kaffe
 - Tensorflow
 - DyNet
 - Chainer
 - PyTorch
 - EDF (Educational Framework in Python for this class).
- ⋮

An Example: Multi-Layer Perceptron Models 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 $y \in \{y_1, \dots, y_k\}$.

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

A Multi Layer Perceptron (MLP)

$$\Phi = (W^0, b^0, W^1, b^1)$$

$$\textcolor{red}{h} = \sigma \left(W^0 \textcolor{red}{x} - b^0 \right)$$

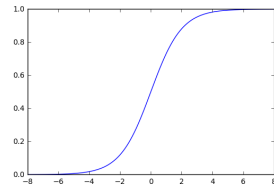
$$\textcolor{red}{s} = \sigma \left(W^1 \textcolor{red}{h} - b^1 \right)$$

$$\textcolor{red}{P}_\Phi[\hat{y}] = \underset{\hat{y}}{\text{softmax}} \textcolor{red}{s}[\hat{y}]$$

Activation Functions

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

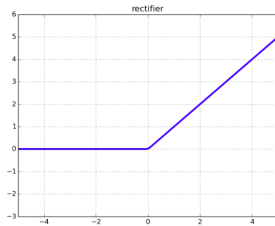
$$\sigma(u) = \frac{1}{1+e^{-u}}$$



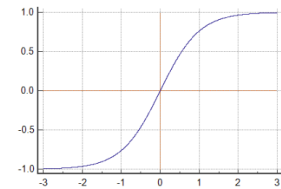
, $\sigma(m) = P(y|m)$ for margin m .

other common activation functions are

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



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



Einstein Notation

$$\textcolor{red}{h} = \sigma \left(W^0 \textcolor{red}{x} - b^0 \right)$$

is an abbreviation for

$$\textcolor{red}{h}[j] = \sigma \left(\left(\sum_i W^0[j, i] \textcolor{red}{x}[i] \right) - b^0[j] \right)$$

Think of this as a separate assignment statement for each $\textcolor{red}{j}$.

Each $\textcolor{red}{h}[j]$ is the output of a “linear threshold unit”.

Einstein notation makes all indices and summations explicit.

Optimization

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.

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

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 back-propagation on the model definition.

Stochastic Gradient Descent (SGD)

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

1. Randomly Initialize Φ (initialization is important and must be done with care).
2. Repeat until “converged”:
 - draw $(x, y) \sim \text{Train}$ at random.
 - $\Phi \leftarrow \Phi - \eta \nabla_{\Phi} \mathcal{L}(x, y, \Phi)$

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 shuffle of the training data before each epoch.

SGD for MLPs

$$\Phi = (W^0, b^0, W^1, b^1)$$

$$\textcolor{red}{h} = \sigma \left(W^0 \textcolor{red}{x} - b^0 \right)$$

$$\textcolor{red}{s} = \sigma \left(W^1 \textcolor{red}{h} - b^1 \right)$$

$$\textcolor{red}{P}_\Phi[\hat{y}] = \operatorname{softmax}_{\hat{y}} \textcolor{red}{s}[\hat{y}]$$

We now need to automatically compute $\nabla_\Phi \mathcal{L}(x, y, \Phi)$.

Computation Graphs (Framework Source Code)

A computation graph (sometimes called a “computation^{al} graph”) is a sequence of assignment statements.

$$\textcolor{red}{h} = \sigma \left(W^0 \textcolor{red}{x} - b^0 \right)$$

$$\textcolor{red}{s} = \sigma \left(W^1 \textcolor{red}{h} - b^1 \right)$$

$$\textcolor{red}{P}_\Phi[\hat{y}] = \operatorname{softmax}_{\hat{y}} \textcolor{red}{s}[\hat{y}]$$

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

$$u = x^2$$

$$v = y^2$$

$$r = u + v$$

$$\mathcal{L} = \sqrt{r}$$

Source Code

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

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

- (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) $\partial\mathcal{L}/\partial x = (2x) * (\partial\mathcal{L}/\partial u)$

A More Abstract Example (Still Scalar Values)

$$y = f(x)$$

$$z = g(y, x)$$

$$u = h(z)$$

$$\mathcal{L} = u$$

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

We will “backpropagate” the assignments the reverse order.

Back-Propagation (Scalar Values)

$$y = f(x)$$

$$z = g(y, x)$$

$$u = h(z)$$

$$\mathcal{L} = u$$

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

Back-Propagation (Scalar Values)

$$y = f(x)$$

$$z = g(y, x)$$

$$u = h(\textcolor{red}{z})$$

$$\mathcal{L} = u$$

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

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

Back-Propagation (Scalar Values)

$$y = f(x)$$

$$z = g(\textcolor{red}{y}, x)$$

$$u = h(z)$$

$$\mathcal{L} = u$$

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

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

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

Back-Propagation (Scalar Values)

$$y = f(\textcolor{red}{x})$$

$$z = g(y, \textcolor{red}{x})$$

$$u = h(z)$$

$$\mathcal{L} = u$$

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

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

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

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

Back-Propagation (Scalar Values)

$$y = f(\textcolor{red}{x})$$

$$z = g(y, \textcolor{red}{x})$$

$$u = h(z)$$

$$\mathcal{L} = u$$

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”.

Back-Propagation (Scalar Values)

$$y = f(x)$$

$$z = g(y, x)$$

$$u = h(z)$$

$$\mathcal{L} = u$$

We initialize $x.\text{grad}$ to zero: $z.\text{grad} = y.\text{grad} = x.\text{grad} = 0$

We initialize the loss gradient to 1: $u.\text{grad} = 1$

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

Back-Propagation (Scalar Values)

$$y = f(x)$$

$$z = g(y, x)$$

$$u = h(z)$$

$$\mathcal{L} = u$$

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

$$u.\text{grad} = 1$$

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

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

Back-Propagation (Scalar Values)

$$y = f(x)$$

$$z = g(y, x)$$

$$u = h(z)$$

$$\mathcal{L} = u$$

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

$$u.\text{grad} = 1$$

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

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

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

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

Back-Propagation (Scalar Values)

$$y = f(x)$$

$$z = g(y, x)$$

$$u = h(z)$$

$$\mathcal{L} = u$$

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

$$u.\text{grad} = 1$$

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

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

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

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

Handling Arrays

$$\textcolor{red}{h} = \sigma \left(W^0 \textcolor{red}{x} - b^0 \right)$$

$$\textcolor{red}{s} = \sigma \left(W^1 \textcolor{red}{h} - b^1 \right)$$

$$\textcolor{red}{P}_\Phi[\hat{y}] = \underset{\hat{y}}{\text{softmax}} \textcolor{red}{s}[\hat{y}]$$

Each array $\textcolor{red}{W}$ is an object with attributes $\textcolor{red}{W.value}$ and $\textcolor{red}{W.grad}$.

$\textcolor{red}{W.grad}$ is an array with the same indices as $\textcolor{red}{W.value}$.

An array with more than two indices is called a $\textcolor{red}{\text{tensor}}$.

Einstein Notation

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

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

$$\textcolor{red}{h}[j] = \sigma \left(\left(\sum_i W^0[j, i] \textcolor{red}{x}[i] \right) - b^0[j] \right)$$

$$\textcolor{red}{s}[\hat{y}] = \sigma \left(\left(\sum_j W^1[\hat{y}, j] \textcolor{red}{h}[j] \right) - b^1[\hat{y}] \right)$$

$$\textcolor{red}{P}_\Phi[\hat{y}] = \operatorname{softmax}_{\hat{y}} \textcolor{red}{s}[\hat{y}]$$

Loop Notation

We will define back-propagation on tensor computations by first compiling them to loop notation. Loop notation assumes all computed tensors are initialized to zero.

Vector : $h = \sigma(Wx - B)$

Einstein : $h[j] = \sigma((\sum_i W[j, i] x[i]) - B[j])$

Loop : for i, j $\tilde{h}[j] += W[j, i]x[i]$

 for j $\tilde{h}[j] -= B[j]$

 for j $h[j] = \sigma(\tilde{h}[j])$

Back-Propagation on Loop Notation

We backpropagate the body of the loop.

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

The body of the loop is just a product of scalars. A product of scalars has a trivial back-propagation:

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

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

For 2D CNNs the weight tensor and the data tensor each have four indices (including the batch index). Vector notation becomes unwieldy while loop notation remains trivial.

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.

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 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

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

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

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

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

Back-Propagation with Minibatching

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

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

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

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

Framework Summary

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

A framework compiles a model into an optimization algorithm.

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

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

The Educational Framework (EDF)

The educational framework (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 (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.  
    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

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

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

$$\frac{dy}{dx} = \frac{e^{-x}}{(1 + e^{-x})^2}$$

$$= y(1 - y)$$

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

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

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

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

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

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

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

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

```
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