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.

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

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

Some Frameworks

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

⋮

What Must a Framework Support?

It must provide a high level language for writing models $P_\Phi(y|x)$ (or other forms of models).

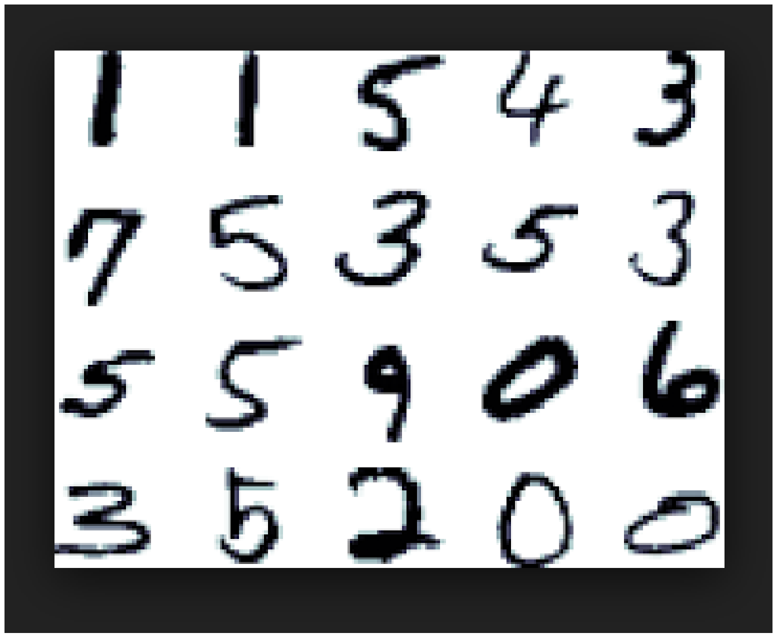
It must be able to compile a model into an optimization algorithm.

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

It typically also provides support for managing large data sets (such as "Train" in the above equation).

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

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)

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

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

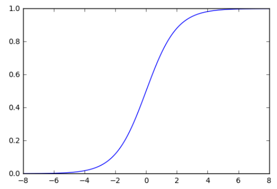
$$s = \sigma\left(W^1 h + b^1\right)$$

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

Activation Functions

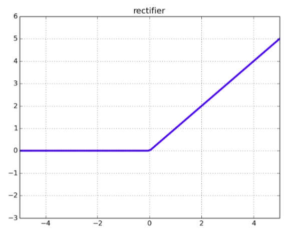
An activation function $\sigma : \mathbb{R} \rightarrow \mathbb{R}$ (scalar-to-scalar) is applied to each component of a vector.

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

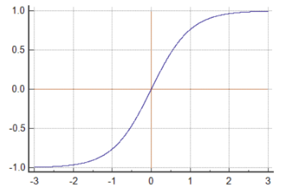


other common activation functions are

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



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



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.

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

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 back-propagation on the model definition.

Stochastic Gradient Descent (SGD)

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

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.

SGD for MLPs

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

$$\begin{aligned} h &= \sigma\left(W^0 x + b^0\right) \\ s &= \sigma\left(W^1 h + b^1\right) \\ P_\Phi[\hat{y}] &= \underset{\hat{y}}{\text{softmax}}\ s[\hat{y}] \end{aligned}$$

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

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.

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

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.

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)

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

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

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

Backpropagation (Scalar Values)

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

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

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

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

Backpropagation (Scalar Values)

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

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

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

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

$\partial\mathcal{L}/\partial x = ???$ 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)

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

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

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

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

Backpropagation (Scalar Values)

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

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

$u.\mathrm{grad} = 1$

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

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

Backpropagation (Scalar Values)

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

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

$u.\mathrm{grad} = 1$

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

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

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

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

Backpropagation (Scalar Values)

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

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

$u.\mathrm{grad} = 1$

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

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

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

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

Handling Vectors, Arrays, and Tensors

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

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

$$P_\Phi[\hat{y}] = \underset{\hat{y}}{\mathrm{softmax}}\ s[\hat{y}]$$

Each array (tensor) W is an object with attributes W.value and a gradient W.grad.

The attribute W.grad is an array (tensor) with the same indeces as W.value.

Einstein Notation

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

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

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

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

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

The Swap Rule

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

$$y[j] = \sigma(\tilde{y}[j] + b[j])$$

$$x.\mathrm{grad}[i] \mathrel{+}= \sum_j \tilde{y}.\mathrm{grad}[j] W[j,i]$$

$$W.\mathrm{grad}[j,i] \mathrel{+}= \tilde{y}.\mathrm{grad}[j] x[i]$$

one swaps the output with one of the inputs and sums over the indeces not occuring on the left.

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

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

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

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

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

The Swap Rule with Minibatching

$$\vdots$$

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

$$\vdots$$

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

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

Summary

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

A framework compiles a model into an optimization algorithm.

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

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

END