TTIC 31230, Fundamentals of Deep Learning

David McAllester, Winter 2019

Deep Learning Frameworks

What is a Deep Learning Framework?

A Deep learning framework is a package of tools that support the construction and training of deep models.

- Kaffe
- Tensorflow
- DyNet
- Chainer
- PyTorch
- EDF (Educational Framework written 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).

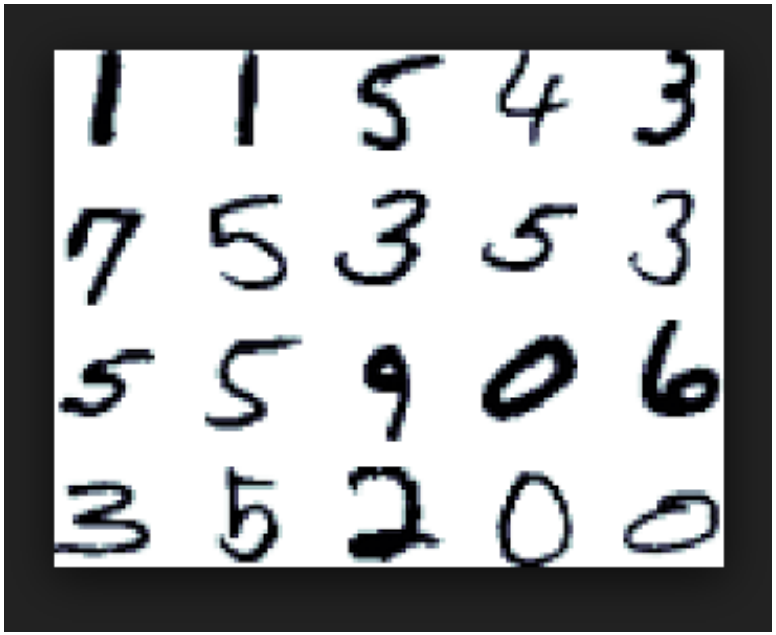
It must be able to compile a model into an optimization algorithm.

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

It typically also provides support for managing large data sets (such as "Train" in the above equation). We will not discuss data management in 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 $y \in \{y_1, \ldots, y_k\}$.

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

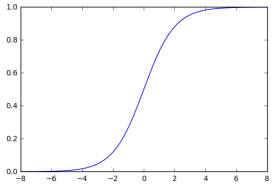
For MNIST $k = 10$.

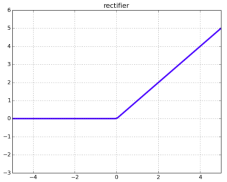
Let Train be a sample $(x_0, y_0),\ \ldots,\ (x_{N-1}, y_{N-1})$ drawn IID from the population.

Multi Layer Perceptrons (MLPs)

Activation functions:

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





An MLP in Einstein Notation (Explicit Indices)

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

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

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

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

$$P_\Phi(\hat{y}) = \underset{\hat{y}}{\text{softmax}}\ s[\hat{y}]$$

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.

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

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.

Gradients with Respect to Systems of Parameters

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

We can think of Φ as a single vector with

$$(\nabla_\Phi \mathcal{L}(x, y, \Phi))[i] = \partial \mathcal{L}(x, y, \Phi)/\partial \Phi[i]$$

But typically Φ is actually some system of multi-dimensional arrays (tensors).

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

$$\nabla_\Phi \mathcal{L}(x, y, \Phi) = (W^0.\text{grad}[j, i],\ b^0.\text{grad}[j],\ W^1.\text{grad}[\hat{y}, j],\ b^1.\text{grad}[\hat{y}]).$$

Total Gradient Descent

$$\mathcal{L}_{\text{Train}}(\Phi) = \frac{1}{N} \sum_{n} \mathcal{L}(\Phi, x_n, y_n)$$

We want: $\Phi^* = \text{argmin}_{\Phi} \mathcal{L}_{\text{Train}}(\Phi)$

$$\Phi \quad \texttt{-=} \quad \eta \nabla_{\Phi} \mathcal{L}_{\text{Train}}(\Phi)$$

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 $(x, y) \sim$ Train at random.
   - Φ `-=` $\eta \nabla_\Phi \mathcal{L}(x, y, \Phi)$

Stochastic Gradient Descent (SGD) on the Training set.

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

$$E_n \, \nabla_\Phi \, \mathcal{L}(x_n, y_n, \Phi) = \sum_n P(n) \nabla_\Phi \, \mathcal{L}_{\text{Train}}(x_n, y_n, \Phi)$$

$$= \frac{1}{N} \sum_n \nabla_\Phi \, \mathcal{L}_{\text{Train}}(x_n, y_n, \Phi)$$

$$= \nabla_\Phi \frac{1}{N} \sum_n \, \mathcal{L}_{\text{Train}}(x_n, y_n, \Phi)$$

$$= \nabla_\Phi \, \mathcal{L}_{\text{Train}}(\Phi)$$

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 — input feature $\qquad$ j — hidden feature $\qquad$ $\hat{y}$ — possible label

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

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

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

$$P_\Phi(\hat{y}) = \underset{\hat{y}}{\text{softmax}}\ s[\hat{y}]$$

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

Computation Graphs

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

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

Can be written as the assignment sequence:

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

Computation Graphs

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

For each variable z we can consider $\partial\mathcal{L}/\partial z$.

Gradients are computed in the 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

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

For now assume all values are scalars.

We will "backpopagate" the assignments the reverse order.

Backpropagation

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

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

Backpropagation

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

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

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

Backpropagation

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

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

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

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

Backpropagation

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

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

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

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

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

Backpropagation

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

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 $\partial \mathcal{L}/\partial x$ into x.grad.

Backpropagation

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

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

$u.\text{grad} = 1$

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

Backpropagation

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

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

$u.\text{grad} = 1$

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

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

Backpropagation

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

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

$u.\text{grad} = 1$

Loop Invariant: For any variable w whose definition has not yet been processed we have that $w.\text{grad}$ is $\partial\mathcal{L}/\partial w$ 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

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

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

$u.\text{grad} = 1$

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

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

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

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

The Vector-Valued Case

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

Now suppose the variables can be vector-valued.

The loss $\mathcal{L}$ is still a scalar.

In this case

$$x.\text{grad} = \nabla_x \mathcal{L}$$

These are now vectors of the same dimension with

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

The Vector-Valued Case

$$y = f(x)$$

In this case we consider the Jacobian matrix of f.

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

We then have

$$x.\text{grad}[i] \;\texttt{+=}\; \sum_j y.\text{grad}[j](\nabla_x f(x))[j, i]$$

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.

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.

Minibatching for MLPs

b — batch index $\quad i$ — input feature index

j — hidden feature index $\quad \hat{y}$ — label index

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

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

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

$$P[b,\hat{y}] = \underset{\hat{y}}{\mathrm{softmax}}\ s[b,\hat{y}]$$

$$\mathcal{L}[b] = -\log P[b, y[b]]$$

BackProp for MLPs

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

$$s[b, \hat{y}] \;=\; \sigma(\tilde{s}[b, \hat{y}])$$

$$\tilde{s}.\mathrm{grad}[b, \hat{y}] \;\texttt{+=}\; s.\mathrm{grad}[b, \hat{y}] \;\; \sigma'|_{\tilde{s}[b,\hat{y}]}$$

BackProp for MLPs

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

$$h.\text{grad}[b,j] \;\texttt{+=}\; \sum_{\hat{y}} \tilde{s}.\text{grad}[b,\hat{y}] W^1[\hat{y},j]$$

$$b^1.\text{grad}[\hat{y}] \;\texttt{+=}\; \frac{1}{B}\sum_{b} \tilde{s}.\text{grad}[b,\hat{y}]$$

$$W^1.\text{grad}[\hat{y},j] \;\texttt{+=}\; \frac{1}{B}\sum_{b} \tilde{s}.\text{grad}[b,\hat{y}] h[b,j]$$

By convention parameter gradients are averaged over the batch.

The Swap Rule

$$\tilde{s}[b, \hat{y}] \;=\; \left(\sum_{j} W^1[\hat{y}, j]\; h[b, j] \right) + \ldots$$

$$h.\text{grad}[b, j] \;\texttt{+=}\; \sum_{\hat{y}} \tilde{s}.\text{grad}[b, \hat{y}] W^1[\hat{y}, j]$$

$$W^1.\text{grad}[\hat{y}, j] \;\texttt{+=}\; \frac{1}{B} \sum_{b} \tilde{s}.\text{grad}[b, \hat{y}] h[b, j]$$

When writing a backprop rule for a contraction of tensors one swaps the output with one of the inputs.

END