

# EE-559 – Deep learning

## 4.1. DAG networks

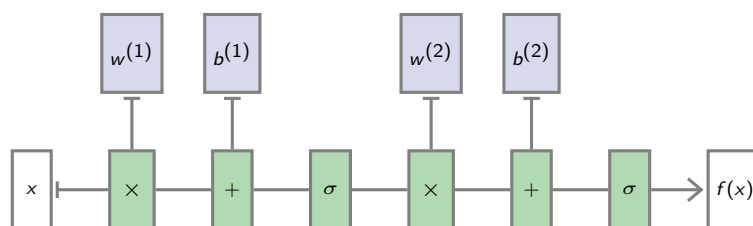
François Fleuret

<https://fleuret.org/ee559/>

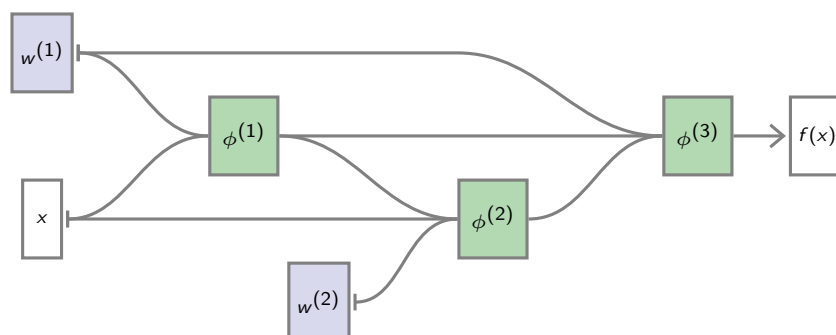
Dec 23, 2019



We can generalize an MLP



to an arbitrary “Directed Acyclic Graph” (DAG) of operators



If  $(a_1, \dots, a_Q) = \phi(b_1, \dots, b_R)$ , we use the notation

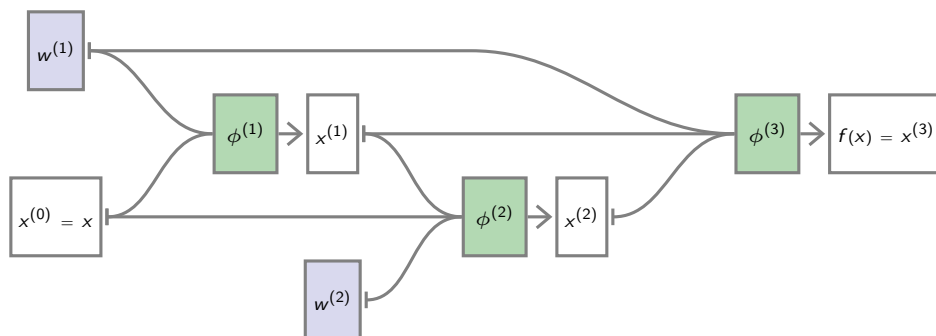
$$\left[ \frac{\partial \mathbf{a}}{\partial \mathbf{b}} \right] = J_\phi = \begin{pmatrix} \frac{\partial a_1}{\partial b_1} & \cdots & \frac{\partial a_1}{\partial b_R} \\ \vdots & \ddots & \vdots \\ \frac{\partial a_Q}{\partial b_1} & \cdots & \frac{\partial a_Q}{\partial b_R} \end{pmatrix}.$$

It does not specify at which point this is computed, but it will always be for the forward-pass activations.

Also, if  $(a_1, \dots, a_Q) = \phi(b_1, \dots, b_R, c_1, \dots, c_S)$ , we use

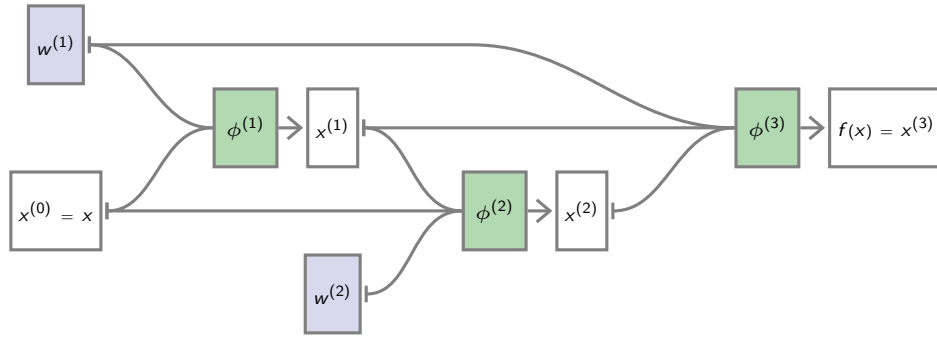
$$\left[ \frac{\partial \mathbf{a}}{\partial \mathbf{c}} \right] = J_{\phi|c} = \begin{pmatrix} \frac{\partial a_1}{\partial c_1} & \cdots & \frac{\partial a_1}{\partial c_S} \\ \vdots & \ddots & \vdots \\ \frac{\partial a_Q}{\partial c_1} & \cdots & \frac{\partial a_Q}{\partial c_S} \end{pmatrix}.$$

## Forward pass



$$\begin{aligned} x^{(0)} &= x \\ x^{(1)} &= \phi^{(1)}(x^{(0)}; w^{(1)}) \\ x^{(2)} &= \phi^{(2)}(x^{(0)}, x^{(1)}; w^{(2)}) \\ f(x) = x^{(3)} &= \phi^{(3)}(x^{(1)}, x^{(2)}; w^{(1)}) \end{aligned}$$

## Backward pass, derivatives w.r.t activations

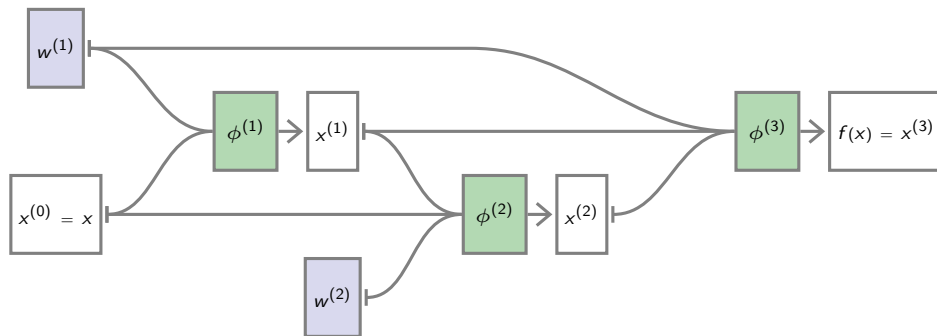


$$\left[ \frac{\partial \ell}{\partial x^{(2)}} \right] = \left[ \frac{\partial x^{(3)}}{\partial x^{(2)}} \right] \left[ \frac{\partial \ell}{\partial x^{(3)}} \right] = J_{\phi^{(3)} | x^{(2)}} \left[ \frac{\partial \ell}{\partial x^{(3)}} \right]$$

$$\left[ \frac{\partial \ell}{\partial x^{(1)}} \right] = \left[ \frac{\partial x^{(2)}}{\partial x^{(1)}} \right] \left[ \frac{\partial \ell}{\partial x^{(2)}} \right] + \left[ \frac{\partial x^{(3)}}{\partial x^{(1)}} \right] \left[ \frac{\partial \ell}{\partial x^{(3)}} \right] = J_{\phi^{(2)} | x^{(1)}} \left[ \frac{\partial \ell}{\partial x^{(2)}} \right] + J_{\phi^{(3)} | x^{(1)}} \left[ \frac{\partial \ell}{\partial x^{(3)}} \right]$$

$$\left[ \frac{\partial \ell}{\partial x^{(0)}} \right] = \left[ \frac{\partial x^{(1)}}{\partial x^{(0)}} \right] \left[ \frac{\partial \ell}{\partial x^{(1)}} \right] + \left[ \frac{\partial x^{(2)}}{\partial x^{(0)}} \right] \left[ \frac{\partial \ell}{\partial x^{(2)}} \right] = J_{\phi^{(1)} | x^{(0)}} \left[ \frac{\partial \ell}{\partial x^{(1)}} \right] + J_{\phi^{(2)} | x^{(0)}} \left[ \frac{\partial \ell}{\partial x^{(2)}} \right]$$

## Backward pass, derivatives w.r.t parameters



$$\left[ \frac{\partial \ell}{\partial w^{(1)}} \right] = \left[ \frac{\partial x^{(1)}}{\partial w^{(1)}} \right] \left[ \frac{\partial \ell}{\partial x^{(1)}} \right] + \left[ \frac{\partial x^{(3)}}{\partial w^{(1)}} \right] \left[ \frac{\partial \ell}{\partial x^{(3)}} \right] = J_{\phi^{(1)} | w^{(1)}} \left[ \frac{\partial \ell}{\partial x^{(1)}} \right] + J_{\phi^{(3)} | w^{(1)}} \left[ \frac{\partial \ell}{\partial x^{(3)}} \right]$$

$$\left[ \frac{\partial \ell}{\partial w^{(2)}} \right] = \left[ \frac{\partial x^{(2)}}{\partial w^{(2)}} \right] \left[ \frac{\partial \ell}{\partial x^{(2)}} \right] = J_{\phi^{(2)} | w^{(2)}} \left[ \frac{\partial \ell}{\partial x^{(2)}} \right]$$

So if we have a library of “tensor operators”, and implementations of

$$\begin{aligned} (x_1, \dots, x_d, w) &\mapsto \phi(x_1, \dots, x_d; w) \\ \forall c, (x_1, \dots, x_d, w) &\mapsto J_{\phi|_{x_c}}(x_1, \dots, x_d; w) \\ (x_1, \dots, x_d, w) &\mapsto J_{\phi|_w}(x_1, \dots, x_d; w), \end{aligned}$$

we can build an arbitrary directed acyclic graph with these operators at the nodes, compute the response of the resulting mapping, and compute its gradient with back-prop.

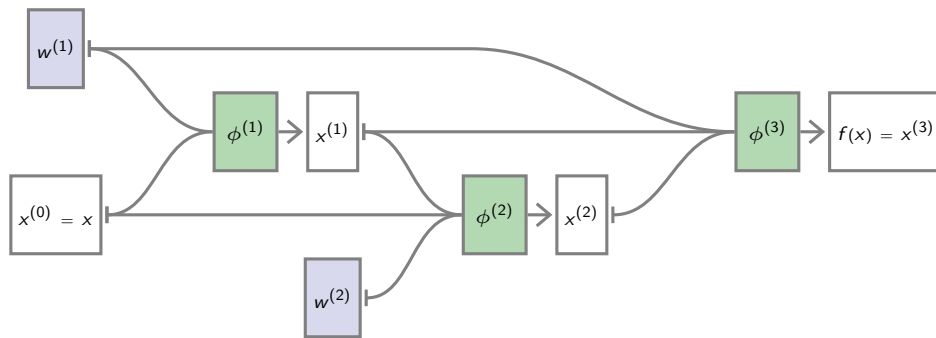
Writing from scratch a large neural network is complex and error-prone.

Multiple frameworks provide libraries of tensor operators and mechanisms to combine them into DAGs and automatically differentiate them.

|                | Language(s)           | License       | Main backer        |
|----------------|-----------------------|---------------|--------------------|
| <b>PyTorch</b> | <b>Python</b>         | BSD           | Facebook           |
| Caffe2         | C++, Python           | Apache        | Facebook           |
| TensorFlow     | Python, C++           | Apache        | Google             |
| MXNet          | Python, C++, R, Scala | Apache        | Amazon             |
| CNTK           | Python, C++           | MIT           | Microsoft          |
| Torch          | Lua                   | BSD           | Facebook           |
| Theano         | Python                | BSD           | U. of Montreal     |
| Caffe          | C++                   | BSD 2 clauses | U. of CA, Berkeley |

One approach is to define the nodes and edges of such a DAG statically (Torch, TensorFlow, Caffe, Theano, etc.)

In TensorFlow, to run a forward/backward pass on



$$\begin{aligned}\phi^{(1)}(x^{(0)}; w^{(1)}) &= w^{(1)}x^{(0)} \\ \phi^{(2)}(x^{(0)}, x^{(1)}; w^{(2)}) &= x^{(0)} + w^{(2)}x^{(1)} \\ \phi^{(3)}(x^{(1)}, x^{(2)}; w^{(1)}) &= w^{(1)}(x^{(1)} + x^{(2)})\end{aligned}$$

```
w1 = tf.Variable(tf.random_normal([5, 5]))
w2 = tf.Variable(tf.random_normal([5, 5]))
x = tf.Variable(tf.random_normal([5, 1]))
x0 = x
x1 = tf.matmul(w1, x0)
x2 = x0 + tf.matmul(w2, x1)
x3 = tf.matmul(w1, x1 + x2)
q = tf.norm(x3)

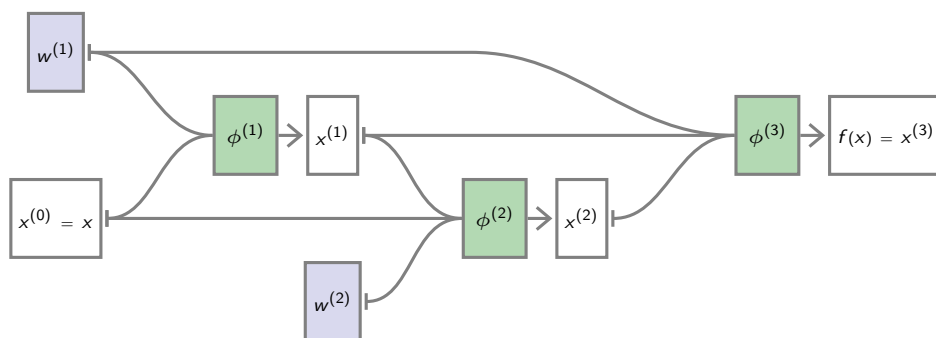
gw1, gw2 = tf.gradients(q, [w1, w2])

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    _gw1, _gw2 = sess.run([gw1, gw2])
```

## Weight sharing

In our generalized DAG formulation, we have in particular implicitly allowed the same parameters to modulate different parts of the processing.

For instance  $w^{(1)}$  in our example parametrizes both  $\phi^{(1)}$  and  $\phi^{(3)}$ .



This is called **weight sharing**.

Weight sharing allows in particular to build **siamese networks** where a full sub-network is replicated several times.

