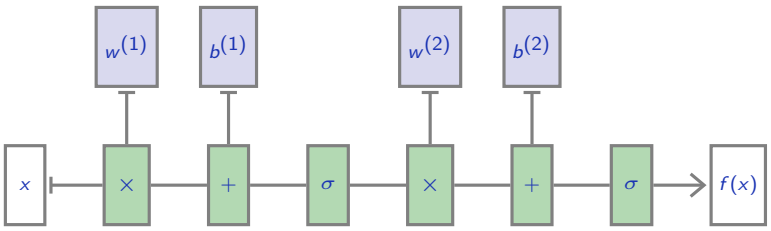
Deep learning

4.1. DAG networks

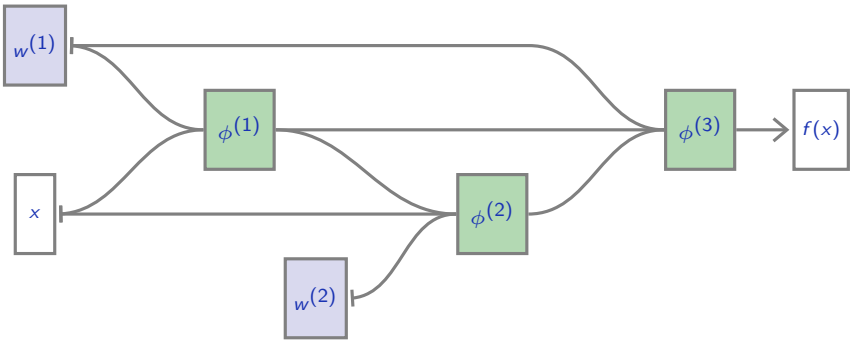
François Fleuret

https://fleuret.org/dlc/

We can generalize an MLP



to an arbitrary "Directed Acyclic Graph" (DAG) of operators



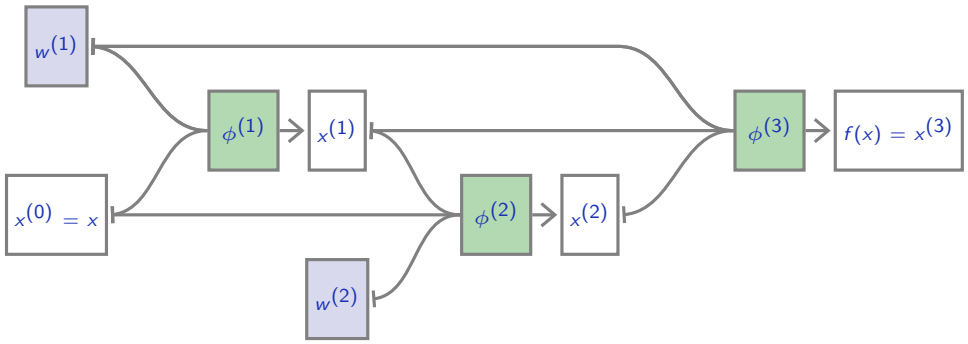
Notes

As before,

- activations (inputs, intermediate activations, outputs) are in white,
- model parameters are in blue,
- operations are in green.

In a directed acyclic graph, there is an ordering of the operations made along the edges that allows to compute the final outputs from the inputs.

Forward pass



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

Notes

DAGs are a simple and straight forward generalization of the forward pass: starting from the inputs, the computation is done in a forward manner to obtain all the inputs required to each operation nodes, until the output is reached.

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

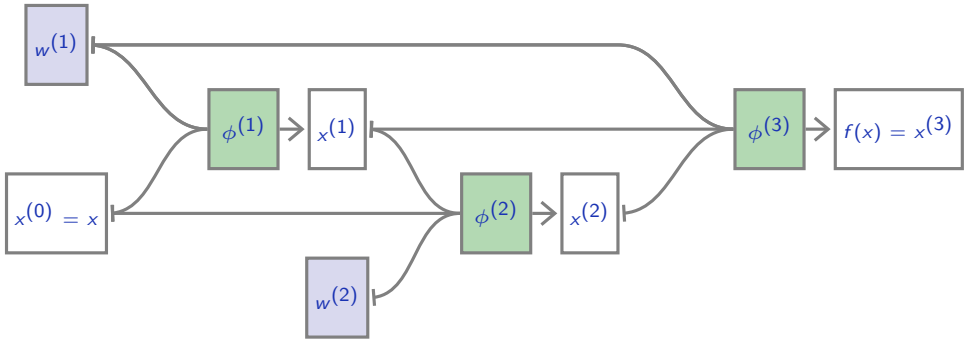
$$\left[\frac{\partial a}{\partial b}\right] = J_\phi^\top = \begin{pmatrix} \frac{\partial a_1}{\partial b_1} & \cdots & \frac{\partial a_Q}{\partial b_1} \\ \vdots & \ddots & \vdots \\ \frac{\partial a_1}{\partial b_R} & \cdots & \frac{\partial a_Q}{\partial b_R} \end{pmatrix}.$$

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

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

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

Backward pass, derivatives w.r.t activations

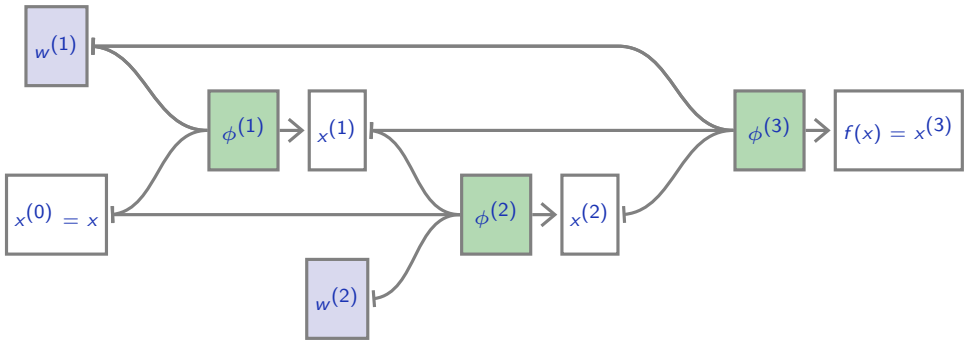


$$\left[\frac{\partial \ell}{\partial x^{(2)}}\right] = \left[\frac{\partial x^{(3)}}{\partial x^{(2)}}\right] \left[\frac{\partial \ell}{\partial x^{(3)}}\right] = J^{\top}_{\phi^{(3)}|x^{(2)}} \left[\frac{\partial \ell}{\partial x^{(3)}}\right]$$

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

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

Backward pass, derivatives w.r.t parameters



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

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

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

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

we can build any directed acyclic graph with these operators at the nodes, evaluate 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**
PyTorch	**Python**, C++	BSD	Facebook
TensorFlow	Python, C++	Apache	Google
JAX	Python	Apache	Google
MXNet	Python, C++, R, Scala	Apache	Amazon
CNTK	Python, C++	MIT	Microsoft
Torch 7	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 (TensorFlow, Torch 7, Caffe, Theano, etc.)

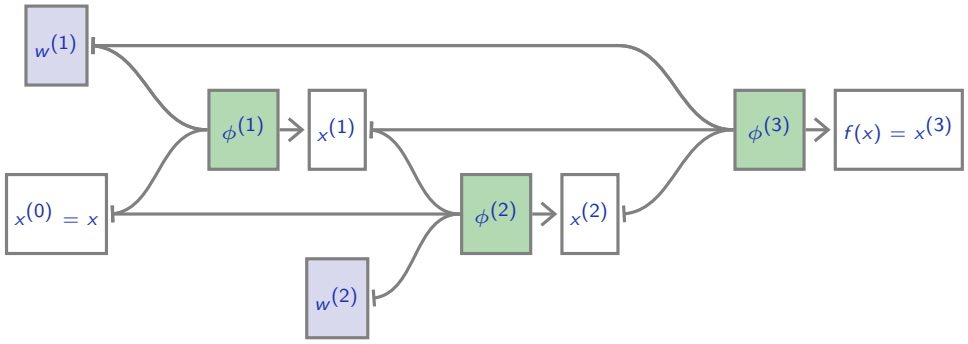
Notes

These frameworks consist of:

- a back end in a low level language (C/C++/CUDA) which directly interacts with the hardware and the libraries which pilot the hardware;
- a front end in a high level language (usually Python because it has its own ecosystem of libraries for plotting, machine learning, signal processing, etc.) which exposes to the AI developer the useful modules.

Note that Torch 7, Theano, and Caffe are not supported anymore.

In TensorFlow, to run a forward/backward pass on



$$\phi^{(1)}\left(x^{(0)}; w^{(1)}\right) = w^{(1)}x^{(0)}$$
$$\phi^{(2)}\left(x^{(0)}, x^{(1)}; w^{(2)}\right) = x^{(0)} + w^{(2)}x^{(1)}$$
$$\phi^{(3)}\left(x^{(1)}, x^{(2)}; w^{(1)}\right) = w^{(1)}\left(x^{(1)} + x^{(2)}\right)$$

```
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])
```

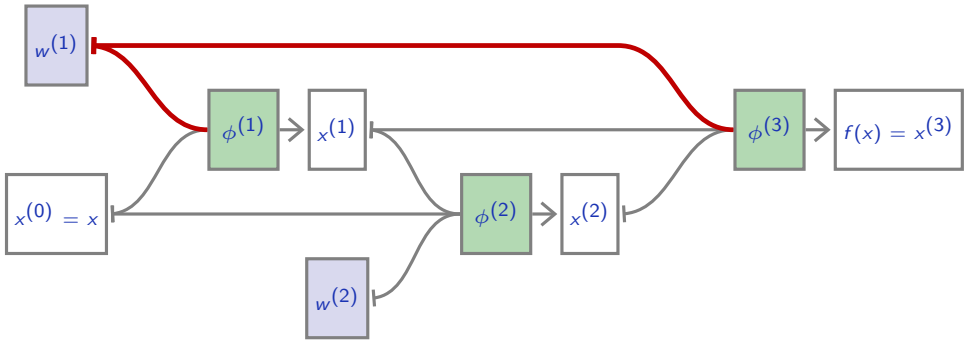
Notes

Note that before the line `with tf.Session() as sess:`, no tensor operations have been executed. The instructions only created a static graph. The operations actually run when `sess.run` is interpreted.

Weight sharing

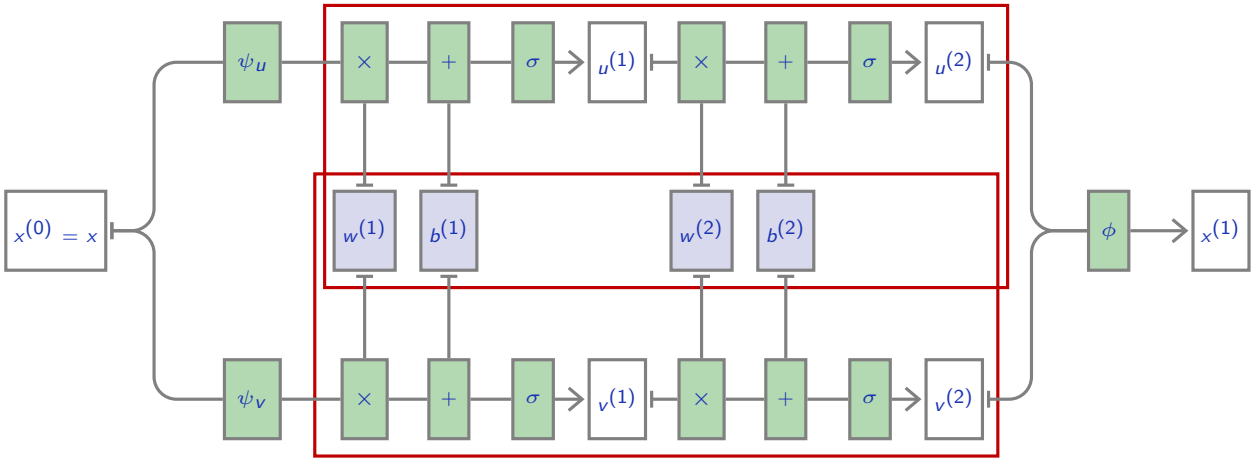
In our generalized DAG formulation, we have allowed the same parameters to modulate different parts of the processing.

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.



Notes

In the case of Siamese networks, which take as input a pair of signals of the same type, the reasoning behind weight sharing is that if a processing is good for one element of the pair, it is also good for the other element.
In the network depicted in that slide, we consider that the initial input is a pair of elements and that ψ_u and ψ_v extract one of the two elements. Then, these two elements go through the same processing before being re-combined by a final processing Φ, that can for instance be a distance measure.