We can generalize an MLP to an arbitrary “Directed Acyclic Graph” (DAG) of operators.
Forward pass

\[ x^{(0)} = x \]
\[ x^{(1)} = \phi^{(1)}(x^{(0)}; w^{(1)}) \]
\[ x^{(2)} = \phi^{(2)}(x^{(0)}, x^{(1)}; w^{(2)}) \]
\[ f(x) = x^{(3)} = \phi^{(3)}(x^{(1)}, x^{(2)}; w^{(1)}) \]

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

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

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

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

\[
\left[ \frac{\partial a}{\partial c} \right] = J_{\phi|c} = \begin{pmatrix}
\frac{\partial a_1}{\partial c_1} & \cdots & \frac{\partial a_1}{\partial c_S} \\
\vdots & \ddots & \vdots \\
\frac{\partial a_Q}{\partial c_1} & \cdots & \frac{\partial a_Q}{\partial c_S}
\end{pmatrix}.
\]
Backward pass, derivatives w.r.t activations

\[
\frac{\partial \ell}{\partial x^{(2)}} = \begin{bmatrix} \frac{\partial x^{(3)}}{\partial x^{(2)}} & \frac{\partial \ell}{\partial x^{(3)}} \end{bmatrix} = J_{\phi^{(3)}|x^{(2)}} \begin{bmatrix} \frac{\partial \ell}{\partial x^{(3)}} \end{bmatrix}
\]

\[
\frac{\partial \ell}{\partial x^{(1)}} = \begin{bmatrix} \frac{\partial x^{(2)}}{\partial x^{(1)}} & \frac{\partial \ell}{\partial x^{(1)}} \end{bmatrix} + \begin{bmatrix} \frac{\partial x^{(3)}}{\partial x^{(1)}} & \frac{\partial \ell}{\partial x^{(3)}} \end{bmatrix} = J_{\phi^{(1)}|x^{(1)}} \begin{bmatrix} \frac{\partial \ell}{\partial x^{(1)}} \end{bmatrix} + J_{\phi^{(2)}|x^{(1)}} \begin{bmatrix} \frac{\partial \ell}{\partial x^{(2)}} \end{bmatrix}
\]

\[
\frac{\partial \ell}{\partial x^{(0)}} = \begin{bmatrix} \frac{\partial x^{(1)}}{\partial x^{(0)}} & \frac{\partial \ell}{\partial x^{(1)}} \end{bmatrix} + \begin{bmatrix} \frac{\partial x^{(2)}}{\partial x^{(0)}} & \frac{\partial \ell}{\partial x^{(2)}} \end{bmatrix} = J_{\phi^{(1)}|x^{(0)}} \begin{bmatrix} \frac{\partial \ell}{\partial x^{(1)}} \end{bmatrix} + J_{\phi^{(2)}|x^{(0)}} \begin{bmatrix} \frac{\partial \ell}{\partial x^{(2)}} \end{bmatrix}
\]

Backward pass, derivatives w.r.t parameters

\[
\frac{\partial \ell}{\partial w^{(1)}} = \begin{bmatrix} \frac{\partial x^{(1)}}{\partial w^{(1)}} & \frac{\partial \ell}{\partial w^{(1)}} \end{bmatrix} = J_{\phi^{(1)}|w^{(1)}} \begin{bmatrix} \frac{\partial \ell}{\partial x^{(1)}} \end{bmatrix}
\]

\[
\frac{\partial \ell}{\partial w^{(2)}} = \begin{bmatrix} \frac{\partial x^{(2)}}{\partial w^{(2)}} & \frac{\partial \ell}{\partial w^{(2)}} \end{bmatrix} = J_{\phi^{(2)}|w^{(2)}} \begin{bmatrix} \frac{\partial \ell}{\partial x^{(2)}} \end{bmatrix}
\]
So if we have a library of “tensor operators”, and implementations of

\[(x_1, \ldots, x_d, w) \mapsto \phi(x_1, \ldots, x_d; w)\]
\[\forall c, \ (x_1, \ldots, x_d, w) \mapsto J_{\phi|_c}(x_1, \ldots, x_d; w)\]
\[(x_1, \ldots, x_d, w) \mapsto J_{\phi|_w}(x_1, \ldots, x_d; w),\]

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.

<table>
<thead>
<tr>
<th>Language(s)</th>
<th>License</th>
<th>Main backer</th>
</tr>
</thead>
<tbody>
<tr>
<td>PyTorch</td>
<td>Python</td>
<td>BSD</td>
</tr>
<tr>
<td>Caffe2</td>
<td>C++, Python</td>
<td>Apache</td>
</tr>
<tr>
<td>TensorFlow</td>
<td>Python, C++</td>
<td>Apache</td>
</tr>
<tr>
<td>JAX</td>
<td>Python</td>
<td>Apache</td>
</tr>
<tr>
<td>MXNet</td>
<td>Python, C++, R, Scala</td>
<td>Apache</td>
</tr>
<tr>
<td></td>
<td>Python, C++</td>
<td>MIT</td>
</tr>
<tr>
<td>Torch</td>
<td>Lua</td>
<td>BSD</td>
</tr>
<tr>
<td>Theano</td>
<td>Python</td>
<td>BSD</td>
</tr>
<tr>
<td>Caffe</td>
<td>C++</td>
<td>BSD 2 clauses</td>
</tr>
</tbody>
</table>

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

\[
\begin{align*}
\phi^{(1)}(x^{(0)}; w^{(1)}) &= w^{(1)} x^{(0)} \\
\phi^{(2)}(x^{(0)}, x^{(1)}; w^{(2)}) &= x^{(0)} + w^{(2)} x^{(1)} \\
\phi^{(3)}(x^{(1)}, x^{(2)}; w^{(1)}) &= w^{(1)} (x^{(1)} + x^{(2)})
\end{align*}
\]

\[
\begin{align*}
w_1 &= \text{tf.Variable(tf.random_normal([5, 5]))} \\
w_2 &= \text{tf.Variable(tf.random_normal([5, 5]))} \\
x &= \text{tf.Variable(tf.random_normal([5, 1]))} \\
x_0 &= x \\
x_1 &= \text{tf.matmul}(w_1, x_0) \\
x_2 &= x_0 + \text{tf.matmul}(w_2, x_1) \\
x_3 &= \text{tf.matmul}(w_1, x_1 + x_2) \\
q &= \text{tf.norm}(x_3) \\
gw_1, gw_2 &= \text{tf.gradients}(q, [w_1, w_2])
\end{align*}
\]

with \text{tf.Session()} as sess:
\[
\text{sess.run(tf.global_variables_initializer())} \\
\_gw_1, \_gw_2 = \text{sess.run([gw_1, gw_2])}
\]
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.