Deep learning

4.1. DAG networks

François Fleuret

https://fleuret.org/dlc/
We can generalize an MLP
We can generalize an MLP

\[
\begin{align*}
  \mathbf{x} & \times w^{(1)} + b^{(1)} \\
  & \times w^{(2)} + b^{(2)} \\
  & \sigma \\
  & \times w^{(1)} + b^{(2)} \\
  & \sigma \\
  & f(x)
\end{align*}
\]

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

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

\[ x^{(0)} = x \]
\[ x^{(1)} = \phi^{(1)}(x^{(0)}; w^{(1)}) \]
Forward pass

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

\[ \begin{align*}
x^{(0)} &= x \\
x^{(1)} &= \phi^{(1)}(x^{(0)}; w^{(1)}) \\
x^{(2)} &= \phi^{(2)}(x^{(0)}, x^{(1)}; w^{(2)}) \\
f(x) &= x^{(3)} = \phi^{(3)}(x^{(1)}, x^{(2)}; w^{(1)})
\end{align*} \]
If \((a_1, \ldots, a_Q) = \phi(b_1, \ldots, b_R)\), we use the notation

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

It does not specify at which point this is computed, but it will always be for the forward-pass activations.
If \((a_1, \ldots, a_Q) = \phi(b_1, \ldots, b_R)\), we use the notation

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

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

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

\[ l(x) = f(x) = x(3) \]

\[ \frac{\partial l}{\partial x(0)} = \frac{\partial l}{\partial x(1)} \frac{\partial x(1)}{\partial x(2)} \frac{\partial x(2)}{\partial x(3)} \]

\[ = J^T \phi(1) | x(0) \frac{\partial l}{\partial x(1)} + J^T \phi(2) | x(0) \frac{\partial l}{\partial x(2)} \]
Backward pass, derivatives w.r.t activations

\[
\begin{align*}
\frac{\partial \ell}{\partial x(2)} &= \left[ \frac{\partial x(3)}{\partial x(2)} \right] \left[ \frac{\partial \ell}{\partial x(3)} \right] = J_{\phi(3)}^{\top} | x(2) \left[ \frac{\partial \ell}{\partial x(3)} \right]
\end{align*}
\]
Backward pass, derivatives w.r.t activations

\[
\begin{align*}
\frac{\partial \ell}{\partial x(2)} &= \frac{\partial x(3)}{\partial x(2)} \frac{\partial \ell}{\partial x(3)} = J_{\phi(3)}^T x(2) \frac{\partial \ell}{\partial x(3)} \\
\frac{\partial \ell}{\partial x(1)} &= \frac{\partial x(2)}{\partial x(1)} \frac{\partial \ell}{\partial x(2)} + \frac{\partial x(3)}{\partial x(1)} \frac{\partial \ell}{\partial x(3)} = J_{\phi(2)}^T x(1) \frac{\partial \ell}{\partial x(2)} + J_{\phi(3)}^T x(1) \frac{\partial \ell}{\partial x(3)}
\end{align*}
\]
Backward pass, derivatives w.r.t activations

\[
\begin{align*}
\frac{\partial \ell}{\partial x(2)} &= \frac{\partial x(3)}{\partial x(2)} \frac{\partial \ell}{\partial x(3)} = J_{\phi(3)}^T | x(2) \frac{\partial \ell}{\partial x(3)} \\
\frac{\partial \ell}{\partial x(1)} &= \frac{\partial x(2)}{\partial x(1)} \frac{\partial \ell}{\partial x(2)} + \frac{\partial x(3)}{\partial x(1)} \frac{\partial \ell}{\partial x(3)} = J_{\phi(2)}^T | x(1) \frac{\partial \ell}{\partial x(2)} + J_{\phi(3)}^T | x(1) \frac{\partial \ell}{\partial x(3)} \\
\frac{\partial \ell}{\partial x(0)} &= \frac{\partial x(1)}{\partial x(0)} \frac{\partial \ell}{\partial x(1)} + \frac{\partial x(2)}{\partial x(0)} \frac{\partial \ell}{\partial x(2)} = J_{\phi(1)}^T | x(0) \frac{\partial \ell}{\partial x(1)} + J_{\phi(2)}^T | x(0) \frac{\partial \ell}{\partial x(2)}
\end{align*}
\]
Backward pass, derivatives w.r.t parameters

\[
\begin{align*}
\frac{\partial l}{\partial w^{(1)}} &= \left[ \frac{\partial x^{(1)}}{\partial w^{(1)}} \right] \frac{\partial l}{\partial x^{(1)}} + \left[ \frac{\partial x^{(3)}}{\partial w^{(1)}} \right] \frac{\partial l}{\partial x^{(3)}} \\
\frac{\partial l}{\partial w^{(2)}} &= \left[ \frac{\partial x^{(2)}}{\partial w^{(2)}} \right] \frac{\partial l}{\partial x^{(2)}}
\end{align*}
\]

Francois Fleuret
Deep learning / 4.1. DAG networks
Backward pass, derivatives w.r.t parameters

\[
 \begin{bmatrix}
 \frac{\partial \ell}{\partial w^{(1)}} \\
 \frac{\partial \ell}{\partial w^{(2)}} \\
 \end{bmatrix}
 =
 \begin{bmatrix}
 \frac{\partial x^{(1)}}{\partial w^{(1)}} \\
 \frac{\partial x^{(2)}}{\partial w^{(2)}} \\
 \end{bmatrix}
 \begin{bmatrix}
 \frac{\partial \ell}{\partial x^{(1)}} \\
 \frac{\partial \ell}{\partial x^{(2)}} \\
 \end{bmatrix}
 +
 \begin{bmatrix}
 \frac{\partial x^{(3)}}{\partial w^{(1)}} \\
 \frac{\partial x^{(3)}}{\partial w^{(2)}} \\
 \end{bmatrix}
 \begin{bmatrix}
 \frac{\partial \ell}{\partial x^{(3)}} \\
 \frac{\partial \ell}{\partial x^{(3)}} \\
 \end{bmatrix}
 =
 J_{\phi^{(1)}}^{T} |_{w^{(1)}}
 \begin{bmatrix}
 \frac{\partial \ell}{\partial x^{(1)}} \\
 \frac{\partial \ell}{\partial x^{(3)}} \\
 \end{bmatrix}
 +
 J_{\phi^{(3)}}^{T} |_{w^{(1)}}
 \begin{bmatrix}
 \frac{\partial \ell}{\partial x^{(3)}} \\
 \frac{\partial \ell}{\partial x^{(3)}} \\
 \end{bmatrix}
\]
Backward pass, derivatives w.r.t parameters

\[
\begin{align*}
\frac{\partial \ell}{\partial w^{(1)}} &= \frac{\partial x^{(1)}}{\partial w^{(1)}} \frac{\partial \ell}{\partial x^{(1)}} + \frac{\partial x^{(3)}}{\partial w^{(1)}} \frac{\partial \ell}{\partial x^{(3)}} = J_{\phi^{(1)}|w^{(1)}}^T \frac{\partial \ell}{\partial x^{(1)}} + J_{\phi^{(3)}|w^{(1)}}^T \frac{\partial \ell}{\partial x^{(3)}} \\
\frac{\partial \ell}{\partial w^{(2)}} &= \frac{\partial x^{(2)}}{\partial w^{(2)}} \frac{\partial \ell}{\partial x^{(2)}} = J_{\phi^{(2)}|w^{(2)}}^T \frac{\partial \ell}{\partial x^{(2)}}
\end{align*}
\]
So if we have a library of “tensor operators”, and implementations of

\[(x_1, \ldots, x_d, w) \mapsto \phi(x_1, \ldots, x_d; w)\]

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

\[(x_1, \ldots, x_d, w) \mapsto J_{\phi|_{w}}(x_1, \ldots, x_d; w),\]
So if we have a library of “tensor operators”, and implementations of

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

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.
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>Framework</th>
<th>Language(s)</th>
<th>License</th>
<th>Main backer</th>
</tr>
</thead>
<tbody>
<tr>
<td>PyTorch</td>
<td>Python</td>
<td>BSD</td>
<td>Facebook</td>
</tr>
<tr>
<td>Caffe2</td>
<td>C++, Python</td>
<td>Apache</td>
<td>Facebook</td>
</tr>
<tr>
<td>TensorFlow</td>
<td>Python, C++</td>
<td>Apache</td>
<td>Google</td>
</tr>
<tr>
<td>JAX</td>
<td>Python</td>
<td>Apache</td>
<td>Google</td>
</tr>
<tr>
<td>MXNet</td>
<td>Python, C++, R, Scala</td>
<td>Apache</td>
<td>Amazon</td>
</tr>
<tr>
<td>CNTK</td>
<td>Python, C++</td>
<td>MIT</td>
<td>Microsoft</td>
</tr>
<tr>
<td>Torch 7</td>
<td>Lua</td>
<td>BSD</td>
<td>Facebook</td>
</tr>
<tr>
<td>Theano</td>
<td>Python</td>
<td>BSD</td>
<td>U. of Montreal</td>
</tr>
<tr>
<td>Caffe</td>
<td>C++</td>
<td>BSD 2 clauses</td>
<td>U. of CA, Berkeley</td>
</tr>
</tbody>
</table>
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>Framework</th>
<th>Language(s)</th>
<th>License</th>
<th>Main backer</th>
</tr>
</thead>
<tbody>
<tr>
<td>PyTorch</td>
<td>Python</td>
<td>BSD</td>
<td>Facebook</td>
</tr>
<tr>
<td>Caffe2</td>
<td>C++, Python</td>
<td>Apache</td>
<td>Facebook</td>
</tr>
<tr>
<td>TensorFlow</td>
<td>Python, C++</td>
<td>Apache</td>
<td>Google</td>
</tr>
<tr>
<td>JAX</td>
<td>Python</td>
<td>Apache</td>
<td>Google</td>
</tr>
<tr>
<td>MXNet</td>
<td>Python, C++, R, Scala</td>
<td>Apache</td>
<td>Amazon</td>
</tr>
<tr>
<td>CNTK</td>
<td>Python, C++</td>
<td>MIT</td>
<td>Microsoft</td>
</tr>
<tr>
<td>Torch 7</td>
<td>Lua</td>
<td>BSD</td>
<td>Facebook</td>
</tr>
<tr>
<td>Theano</td>
<td>Python</td>
<td>BSD</td>
<td>U. of Montreal</td>
</tr>
<tr>
<td>Caffe</td>
<td>C++</td>
<td>BSD 2 clauses</td>
<td>U. of CA, Berkeley</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

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

\[
\begin{align*}
w^{(1)} &= tf.Variable(tf.random_normal([5, 5])) \\
w^{(2)} &= tf.Variable(tf.random_normal([5, 5])) \\
x &= tf.Variable(tf.random_normal([5, 1])) \\
x_0 &= x \\
x_1 &= tf.matmul(w^{(1)}, x_0) \\
x_2 &= x_0 + tf.matmul(w^{(2)}, x_1) \\
x_3 &= tf.matmul(w^{(1)}, x_1 + x_2) \\
q &= tf.norm(x_3) \\
gw^{(1)}, gw^{(2)} &= tf.gradients(q, [w^{(1)}, w^{(2)}]) \\
\end{align*}
\]
In TensorFlow, to run a forward/backward pass on

\[
\phi^{(1)} \left( x^{(0)} ; w^{(1)} \right) = w^{(1)} x^{(0)} \\
\phi^{(2)} \left( x^{(0)} , x^{(1)} ; w^{(2)} \right) = x^{(0)} + w^{(2)} x^{(1)} \\
\phi^{(3)} \left( x^{(1)} , x^{(2)} ; w^{(1)} \right) = w^{(1)} \left( x^{(1)} + x^{(2)} \right)
\]
In TensorFlow, to run a forward/backward pass on

$$\phi^{(1)}(x^{(0)}; w^{(1)}) = w^{(1)} x^{(0)}$$

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

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

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

```python
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.
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.
Weight sharing allows in particular to build **Siamese networks** where a full sub-network is replicated several times.
Weight sharing allows in particular to build **Siamese networks** where a full sub-network is replicated several times.
Weight sharing allows in particular to build **Siamese networks** where a full sub-network is replicated several times.
Weight sharing allows in particular to build **Siamese networks** where a full sub-network is replicated several times.
The end