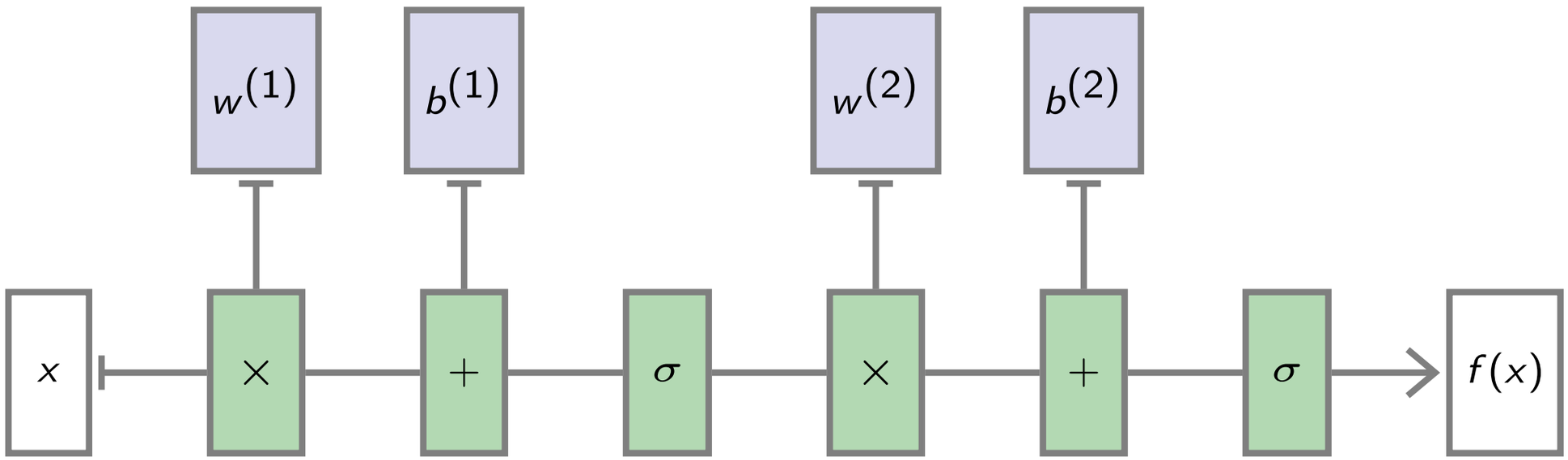
General Computational Networks

MLPs as Graphs

multi layered perceptron drawn as a graph
x input
w weight
b bias
apply activation function
output will be multiplied by another weight and added with another bias

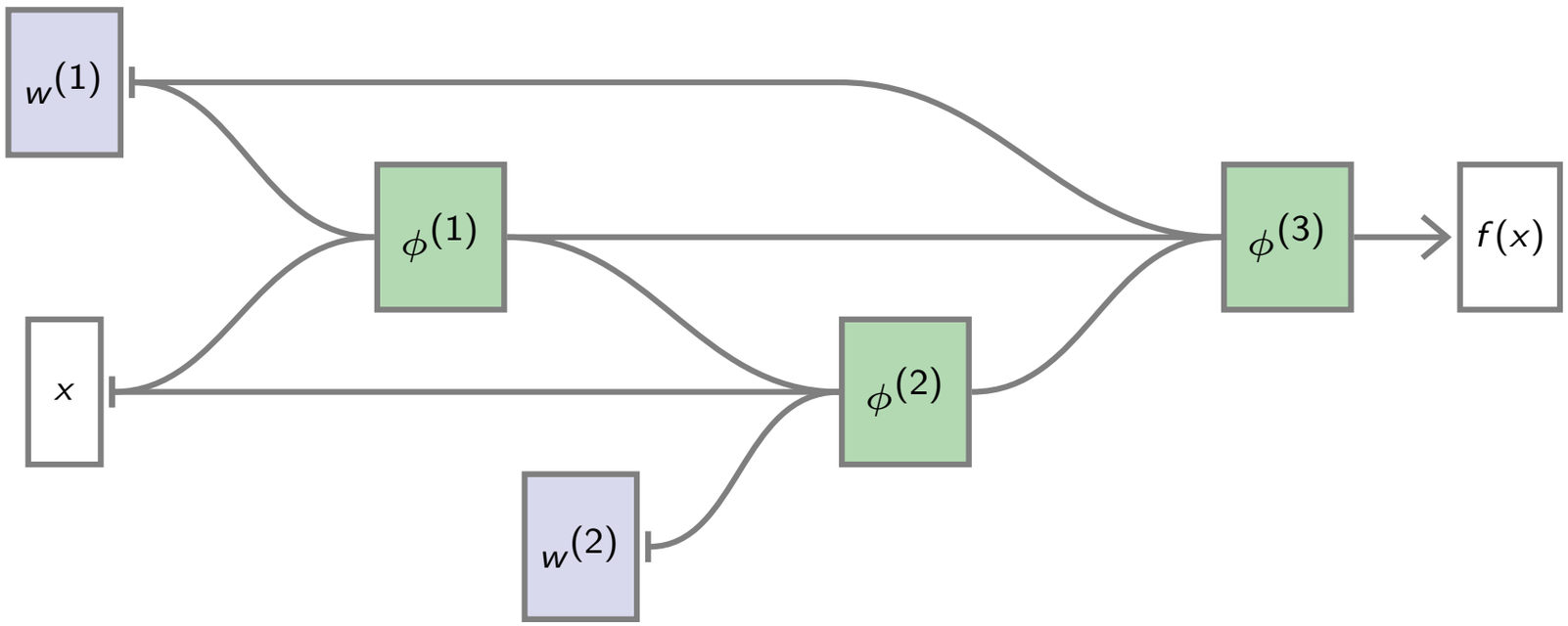


Generalisation to DAGs

not layered, not MLP.

thi1 function doing operation

f(x) as output



Jacobian matrix

putting gradients together.

thi(arguments = b1 to br)
each producing an output eg thi1 producing a1 thiq producing aq
b1 to br have set of input
multiple outputs appearing in the form of a matrix

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

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

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

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

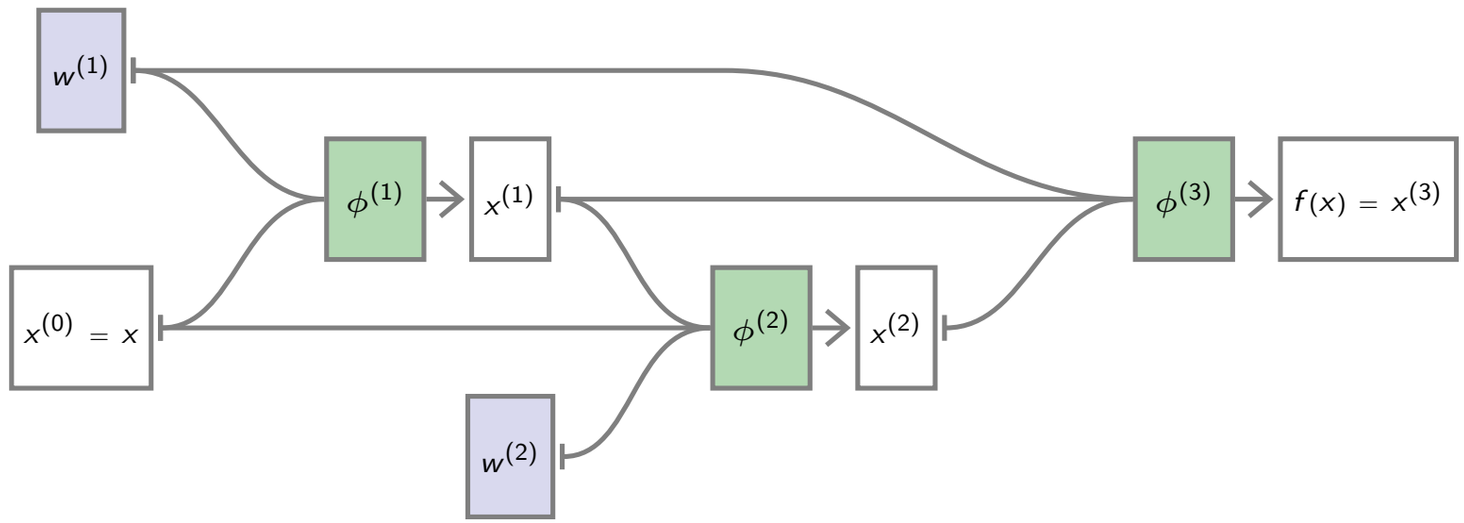
thilc, a subset c of the inputs

thi returning tuple of length q

Forward and Backward passes in general DAGs

have to become a dag become undefined if there is a cycle. where to start? eg in spreadsheets, dont know what forumla to start with

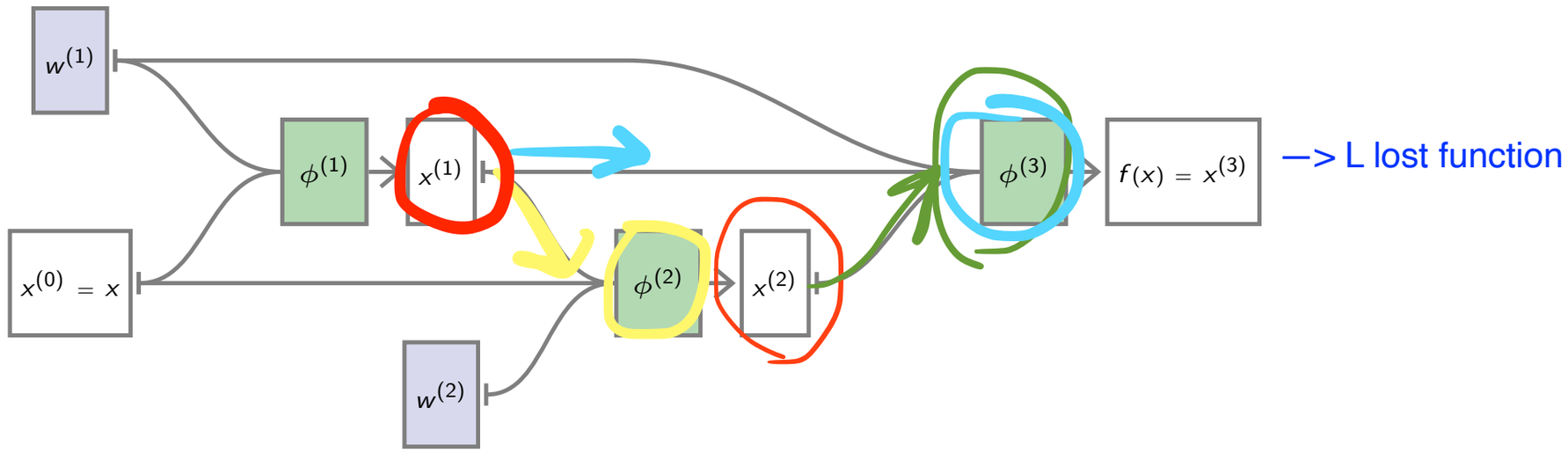
Forward pass



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

Forward and Backward passes in general DAGs

Backward pass, derivatives w.r.t activations



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

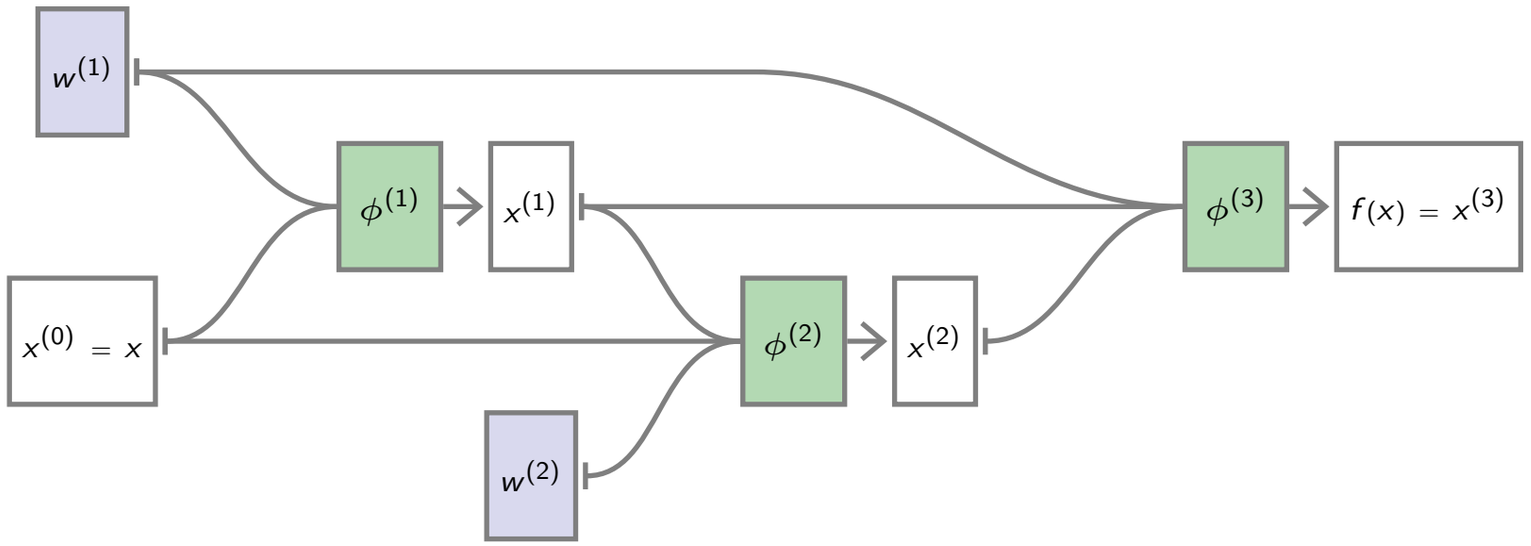
$$\left[\frac{\partial \ell}{\partial x^{(1)}}\right] = \left[\frac{\partial x^{(2)}}{\partial x^{(1)}}\right]\left[\frac{\partial \ell}{\partial x^{(2)}}\right] + \left[\frac{\partial x^{(3)}}{\partial x^{(1)}}\right]\left[\frac{\partial \ell}{\partial x^{(3)}}\right] = J_{\phi^{(2)}|x^{(1)}}\left[\frac{\partial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

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

expression that involves that Jacobian

Forward and Backward passes in general DAGs

Backward pass, derivatives w.r.t parameters



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

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

Forward and Backward passes in general DAGs

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

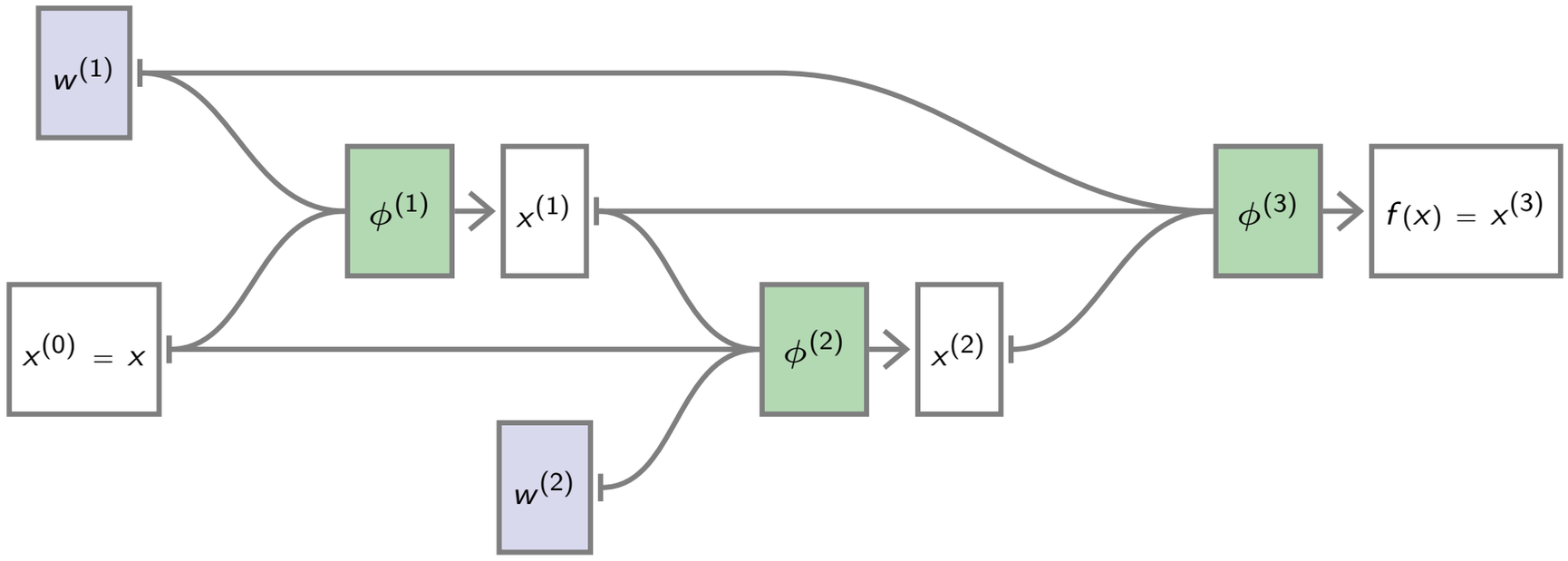
$$
\begin{aligned}
& (x_1, \ldots, x_d, w) \mapsto \phi(x_1, \ldots, x_d; w) \\
\forall c,\ & (x_1, \ldots, x_d, w) \mapsto J_{\phi|x_c}(x_1, \ldots, x_d; w) \\
& (x_1, \ldots, x_d, w) \mapsto J_{\phi|w}(x_1, \ldots, x_d; w),
\end{aligned}
$$

we can build 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.

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**.

Tensors

A tensor is a generalized matrix, a finite table of numerical values indexed along several discrete dimensions.

- A 0d tensor is a scalar,
- A 1d tensor is a vector (*e.g.* a sound sample),
- A 2d tensor is a matrix (*e.g.* a grayscale image),
- A 3d tensor can be seen as a vector of identically sized matrix (*e.g.* a multi-channel image),
- A 4d tensor can be seen as a matrix of identically sized matrices, or a sequence of 3d tensors (*e.g.* a sequence of multi-channel images),
- etc.

Tensors are used to encode the signal to process, but also the internal states and parameters of models.

Manipulating data through this constrained structure allows to use CPUs and GPUs at peak performance.

Tensors

PyTorch is a Python library built on top of Torch's THNN computational backend.

Its main features are:

- Efficient tensor operations on CPU/GPU,
- automatic on-the-fly differentiation (autograd),
- optimizers,
- data I/O.

"Efficient tensor operations" encompass both standard linear algebra and, as we will see later, deep-learning specific operations (convolution, pooling, etc.)

A key specificity of PyTorch is the central role of autograd to compute derivatives of *anything!* We will come back to this.

Tensors

```
>>> x = torch.empty(2, 5)
>>> x.size()
torch.Size([2, 5])
>>> x.fill_(1.125)
tensor([[ 1.1250,  1.1250,  1.1250,  1.1250,  1.1250],
        [ 1.1250,  1.1250,  1.1250,  1.1250,  1.1250]])
>>> x.mean()
tensor(1.1250)
>>> x.std()
tensor(0.)
>>> x.sum()
tensor(11.2500)
>>> x.sum().item()
11.25
```

In-place operations are suffixed with an underscore, and a 0d tensor can be converted back to a Python scalar with `item()`.

Reading a coefficient also generates a 0d tensor.

```
>>> x = torch.tensor([[11., 12., 13.], [21., 22., 23.]])
>>> x[1, 2]
tensor(23.)
```

Tensors

PyTorch provides operators for component-wise and vector/matrix operations.

```
>>> x = torch.tensor([ 10., 20., 30.])
>>> y = torch.tensor([ 11., 21., 31.])
>>> x + y
tensor([ 21.,  41.,  61.])
>>> x * y
tensor([ 110.,  420.,  930.])
>>> x**2
tensor([ 100.,  400.,  900.])
>>> m = torch.tensor([[ 0., 0., 3. ],
...                   [ 0., 2., 0. ],
...                   [ 1., 0., 0. ]])
>>> m.mv(x)
tensor([ 90.,  40.,  10.])
>>> m @ x
tensor([ 90.,  40.,  10.])
```

Pytorch and autograds

Conceptually, the forward pass is a standard tensor computation, and the DAG of tensor operations is required only to compute derivatives.

When executing tensor operations, PyTorch can automatically construct on-the-fly the graph of operations to compute the gradient of any quantity with respect to any tensor involved.

on the fly, it computes derivative for you, you just write forward pass

This "autograd" mechanism (Paszke et al., 2017) has two main benefits:

- Simpler syntax: one just need to write the forward pass as a standard sequence of Python operations,
- greater flexibility: since the graph is not static, the forward pass can be dynamically modulated.

Pytorch and autograds

A `Tensor` has a Boolean field `requires_grad`, set to `False` by default, which states if PyTorch should build the graph of operations so that gradients with respect to it can be computed.

The result of a tensorial operation has this flag to `True` if any of its operand has it to True.

```
>>> x = torch.tensor([ 1., 2. ])
>>> y = torch.tensor([ 4., 5. ])
>>> z = torch.tensor([ 7., 3. ])
>>> x.requires_grad
False
>>> (x + y).requires_grad
False
>>> z.requires_grad = True
>>> (x + z).requires_grad
True
```

Pytorch and autograds

Only floating point type tensors can have their gradient computed.

```
>>> x = torch.tensor([1., 10.])
>>> x.requires_grad = True
>>> x = torch.tensor([1, 10])
>>> x.requires_grad = True
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
RuntimeError: only Tensors of floating point dtype can require gradients
```

The method `requires_grad_(value = True)` set `requires_grad` to `value`, which is `True` by default.

Pytorch and autograds

numpy think of as data storage
tensor - a needs to know its history, not just the data stored in a, but also where a came from.

`torch.autograd.grad(outputs, inputs)` computes and returns the gradient of `outputs` with respect to `inputs`.

```
>>> t = torch.tensor([1., 2., 4.]).requires_grad_()
>>> u = torch.tensor([10., 20.]).requires_grad_()
>>> a = t.pow(2).sum() + u.log().sum()
>>> torch.autograd.grad(a, (t, u))
(tensor([2., 4., 8.]), tensor([0.1000, 0.0500]))
```

a = sum(t^2) + sum(u.log)

a = 1^2 + 2^2 + 4^2 + log(10) + log(20)

da/dt =

`inputs` can be a single tensor, but the result is still a [one element] tuple.

If `outputs` is a tuple, the result is the sum of the gradients of its elements.

Pytorch and autograds

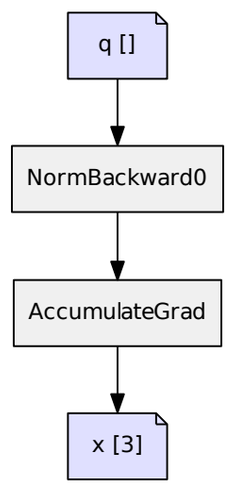
The function `Tensor.backward()` accumulates gradients in the `grad` fields of tensors which are not results of operations, the "leaves" in the autograd graph.

```
>>> x = torch.tensor([ -3., 2., 5. ]).requires_grad_()
>>> u = x.pow(3).sum()
>>> x.grad
>>> u.backward()
>>> x.grad
tensor([27., 12., 75.])
```

This function is an alternative to `torch.autograd.grad(...)` and standard for training models.

autograd graphs

```
x = torch.tensor([1., 2., 2.]).requires_grad_()
q = x.norm()
```



as you are doing operations, the graph is being generated on the fly, get constructed for you automatically

autograd graphs

```
w1 = torch.rand(20, 10).requires_grad_()
b1 = torch.rand(20).requires_grad_()
w2 = torch.rand(5, 20).requires_grad_()
b2 = torch.rand(5).requires_grad_()
```

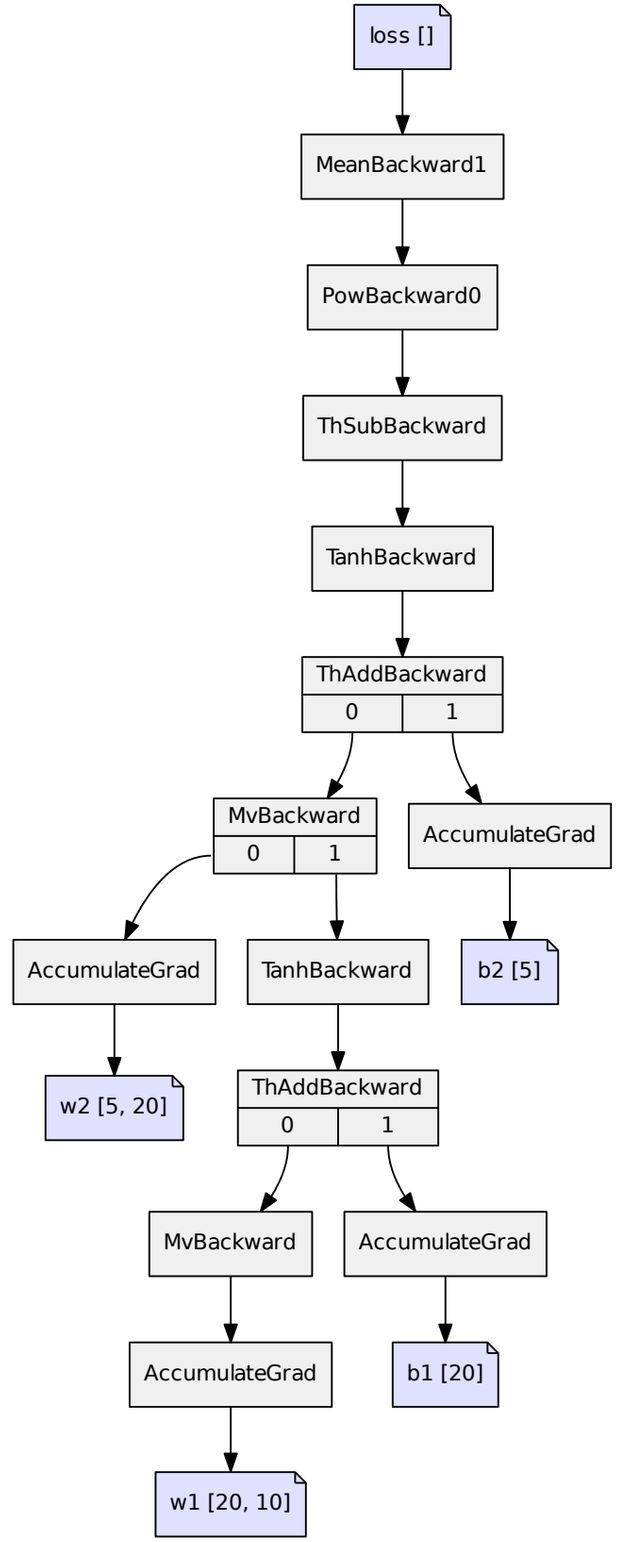
forward pass to describe newwork

```
x = torch.rand(10)
h = torch.tanh(w1 @ x + b1)
y = torch.tanh(w2 @ h + b2)

target = torch.rand(5)
```

raNDOM vector of size 5

```
loss = (y - target).pow(2).mean()
```

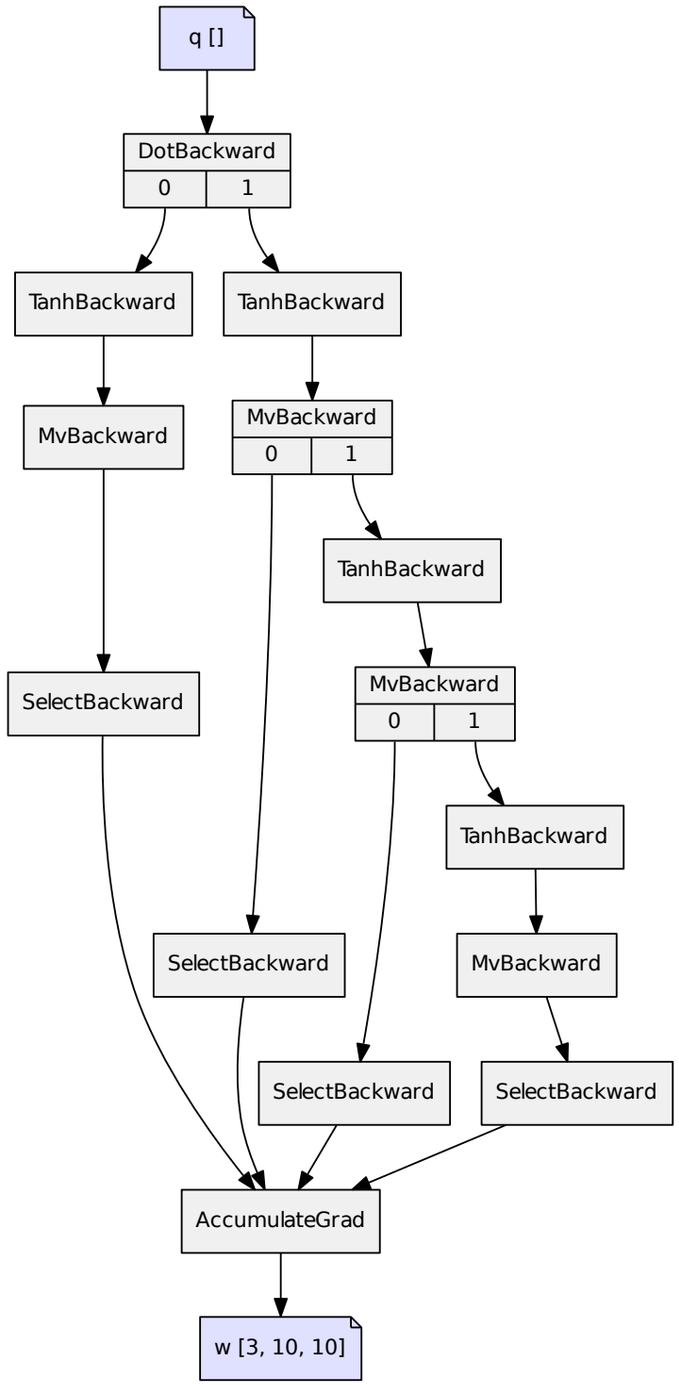


autograd graphs

```
w = torch.rand(3, 10, 10).requires_grad_()

def blah(k, x):
    for i in range(k):
        x = torch.tanh(w[i] @ x)
    return x

u = blah(1, torch.rand(10))
v = blah(3, torch.rand(10))
q = u.dot(v)
```



⚠ Although they are related, **the autograd graph is not the network's structure**, but the graph of operations to compute the gradient. It can be data-dependent and miss or replicate sub-parts of the network.