Backpropagation and Hardware-Aware AI

Generative AI and Foundation Models

Spring 2024

Department of Mathematical Sciences

Ernest K. Ryu

Seoul National University

Backprop $\subseteq$ autodiff

Autodiff (automatic differentiation) is an algorithm that automates gradient computation. In deep learning libraries, you only need to specify how to evaluate the function.
Backprop (back propagation) is an instance of autodiff.

Gradient computation costs roughly $5\times$ the computation cost* of forward evaluation.

To clarify, backprop and autodiff are not

- finite difference or
- symbolic differentiation.

Autodiff $\approx$ chain rule of vector calculus

*Depends on computational structure of function. 5X difference is mostly true for neural networks used in deep learning.

Autodiff example

This complicated gradient computation is simplified by autodiff.

PyTorch demo

In[∘]:=
```
fn = Sin[Cosh[y^2 + x/z] + Tanh[x y z]] / Log[1 + Exp[x]];
D[fn, x]
% /. {x → 3.3, y → 1.1, z → 2.3} // N
D[fn, y]
% /. {x → 3.3, y → 1.1, z → 2.3} // N
D[fn, z]
% /. {x → 3.3, y → 1.1, z → 2.3} // N
```

미분 계수 / 수치

Out[∘]= $-\frac{e^x \mathrm{Sin}\left[\mathrm{Cosh}\left[y^2+\frac{x}{z}\right]+\mathrm{Tanh}[x\,y\,z]\right]}{(1+e^x)\,\mathrm{Log}[1+e^x]^2}+\frac{\mathrm{Cos}\left[\mathrm{Cosh}\left[y^2+\frac{x}{z}\right]+\mathrm{Tanh}[x\,y\,z]\right]\left(y\,z\,\mathrm{Sech}[x\,y\,z]^2+\frac{\mathrm{Sinh}\left[y^2+\frac{x}{z}\right]}{z}\right)}{\mathrm{Log}[1+e^x]}$

Out[∘]= -0.285274

Out[∘]= $\frac{\mathrm{Cos}\left[\mathrm{Cosh}\left[y^2+\frac{x}{z}\right]+\mathrm{Tanh}[x\,y\,z]\right]\left(x\,z\,\mathrm{Sech}[x\,y\,z]^2+2\,y\,\mathrm{Sinh}\left[y^2+\frac{x}{z}\right]\right)}{\mathrm{Log}[1+e^x]}$

Out[∘]= -1.01578

Out[∘]= $\frac{\mathrm{Cos}\left[\mathrm{Cosh}\left[y^2+\frac{x}{z}\right]+\mathrm{Tanh}[x\,y\,z]\right]\left(x\,y\,\mathrm{Sech}[x\,y\,z]^2-\frac{x\,\mathrm{Sinh}\left[y^2+\frac{x}{z}\right]}{z^2}\right)}{\mathrm{Log}[1+e^x]}$

Out[∘]= 0.288027

The power of autodiff

Autodiff is an essential yet often an underappreciated feature of the deep learning libraries. It allows deep learning researchers to use complicated neural networks, while avoiding the burden of performing derivative calculations by hand.

Most deep learning libraries support 2nd and higher order derivative computation, but we will only use 1st order derivatives (gradients) in this class.

Autodiff includes forward-mode, reverse-mode (backprop), and other orders. In deep learning, reverse-mode is most commonly used.

Autodiff by Jacobian multiplication

Consider $g = f_L \circ f_{L-1} \circ \cdots \circ f_2 \circ f_1$, where $f_\ell \colon \mathbb{R}^{n_{\ell-1}} \to \mathbb{R}^{n_\ell}$ for $\ell = 1, \cdots, L$.

Chain rule: $Dg = \underset{n_L \times n_{L-1}}{Df_L} \; \underset{n_{L-1} \times n_{L-2}}{Df_{L-1}} \; \cdots \; \underset{n_2 \times n_1}{Df_2} \; \underset{n_1 \times n_0}{Df_1}$

Forward-mode: $Df_L(Df_{L-1}(\cdots(Df_2 Df_1)\cdots))$

Reverse-mode: $(((Df_L \, Df_{L-1}) \, Df_{L-2}) \cdots) \, Df_1$

Reverse mode is optimal* when $n_L \leq n_{L-1} \leq \cdots \leq n_1 \leq n_0$. The number of neurons in each layer tends to decrease in deep neural networks for classification. So reverse-mode is often close to the most efficient mode of autodiff in deep learning.

*Can be proved with dynamic programming. Cf. “matrix chain multiplication”.

Backprop for MLP

Consider the MLP

$$\mathcal{L} = \frac{1}{2}(y_L - Y_{\text{data}})^2$$

$$y_L = \sigma(A_L y_{L-1} + b_L)$$

$$y_{L-1} = \sigma(A_{L-1} y_{L-2} + b_{L-1})$$

$$\vdots$$

$$y_2 = \sigma(A_2 y_1 + b_2)$$

$$y_1 = \sigma(A_1 x + b_1),$$

where $x \in \mathbb{R}^{n_0}$, $A_\ell \in \mathbb{R}^{n_\ell \times n_{\ell-1}}$, $b_\ell \in \mathbb{R}^{n_\ell}$, and $n_L = 1$.

Backprop can be computed with:

$$\frac{\partial \mathcal{L}}{\partial y_L} = (y_L - Y_{\text{data}})$$

$$\frac{\partial \mathcal{L}}{\partial b_\ell} = \frac{\partial \mathcal{L}}{\partial y_\ell} \mathrm{diag}\left(\sigma'(A_\ell y_{\ell-1} + b_\ell)\right)$$

$$\frac{\partial \mathcal{L}}{\partial A_\ell} = \mathrm{diag}\left(\sigma'(A_\ell y_{\ell-1} + b_\ell)\right) \left(\frac{\partial \mathcal{L}}{\partial y_\ell}\right)^{\intercal} y_{\ell-1}^{\intercal}$$

$$\frac{\partial \mathcal{L}}{\partial y_{\ell-1}} = \frac{\partial \mathcal{L}}{\partial y_\ell} \mathrm{diag}\left(\sigma'(A_\ell y_{\ell-1} + b_\ell)\right) A_\ell$$

for $\ell = L, L-1, \dots, 1$.

Backprop for MLP

Backprop requires two steps.

Forward pass: Evaluate the intermediate node values.

Backward pass: Evaluate gradient in reverse order of computation.

```
# forward pass
y_list = [X_data]
y = X_data
for ell in range(L):
  y = sigma(A_list[ell]@y+b_list[ell])
  y_list.append(y_next)

# backward pass
dA_list = []
db_list = []
dy = y-Y_data
for ell in reversed(range(L)):
  A, b, y= A_list[ell], b_list[ell], y_list[ell]
  db = dy*sigma_prime(A@y+b).T
  dA = (sigma_prime(A@y+b)*dy.T)@y.T
  dy = (dy*sigma_prime(A@y+b).T)@A
  dA_list.insert(0,dA)
  db_list.insert(0,db)
```

General graph-form backprop

Consider a computation graph $G = (V, E)$.

- G is required to be a finite directed acyclic graph (DAG). (No self-loops.)
- $i \in V$ is an *input* node if $v \nrightarrow i$ for all $v \in V$. An input node i has its value y_i provided but has no evaluation function, i.e., $f_i = \emptyset$.
- For a non-input node $v \in V$, the value is y_v and it is computed by f_v, which takes in as input all $u \in V$ such that $u \to v$. I.e., $y_v = f_v\left(\{y_u \,|\, u \to v,\, u \in V\}\right)$.

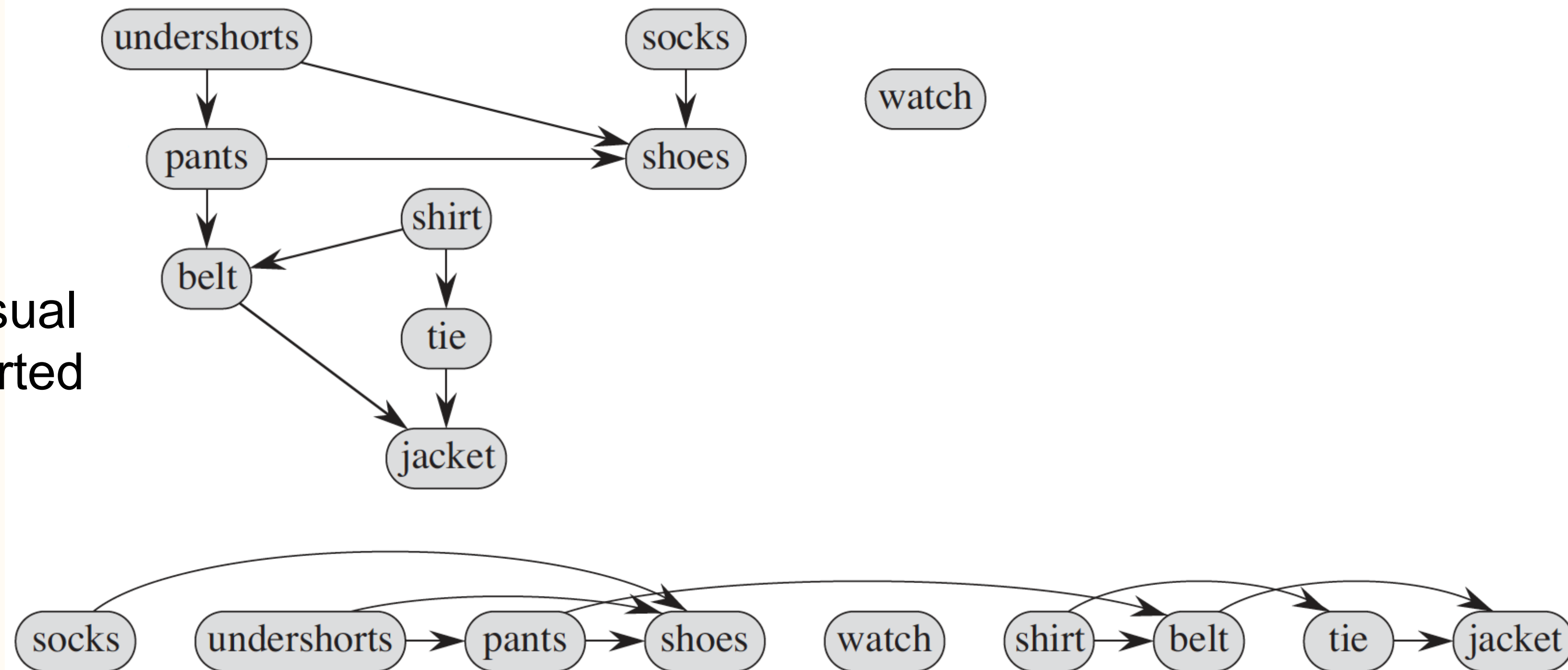
```
# Forward pass given v.value for input nodes
for v in V :
  if v.notInput:
    v.value = v.fn( [u.value for u->v] )
```

- $o \in V$ is an *output* node if $o \nrightarrow v$ for all $v \in V$. Assume there is only one output node.
- Goal of backprop is to compute $\frac{\partial y_o}{\partial y_i}$ for all input nodes i.

Topological ordering of DAGs

DAGs are used to indicate precedence among events. A *topological sort* (top-sort) of a directed acyclic graph (DAG) $G = (V, E)$ finds an ordering $v_1, \dots, v_{|V|}$ such that $v_i \nrightarrow v_j$ for any $i \geq j$. (All edges go forward with respect to the ordering.) Such an order is called a *linear topological order*.

(Top-sort is different from the usual "sorting" in which an array is sorted under a total order.)



DAG representing the constraints of a dressing routine.

Graph-form forward evaluation

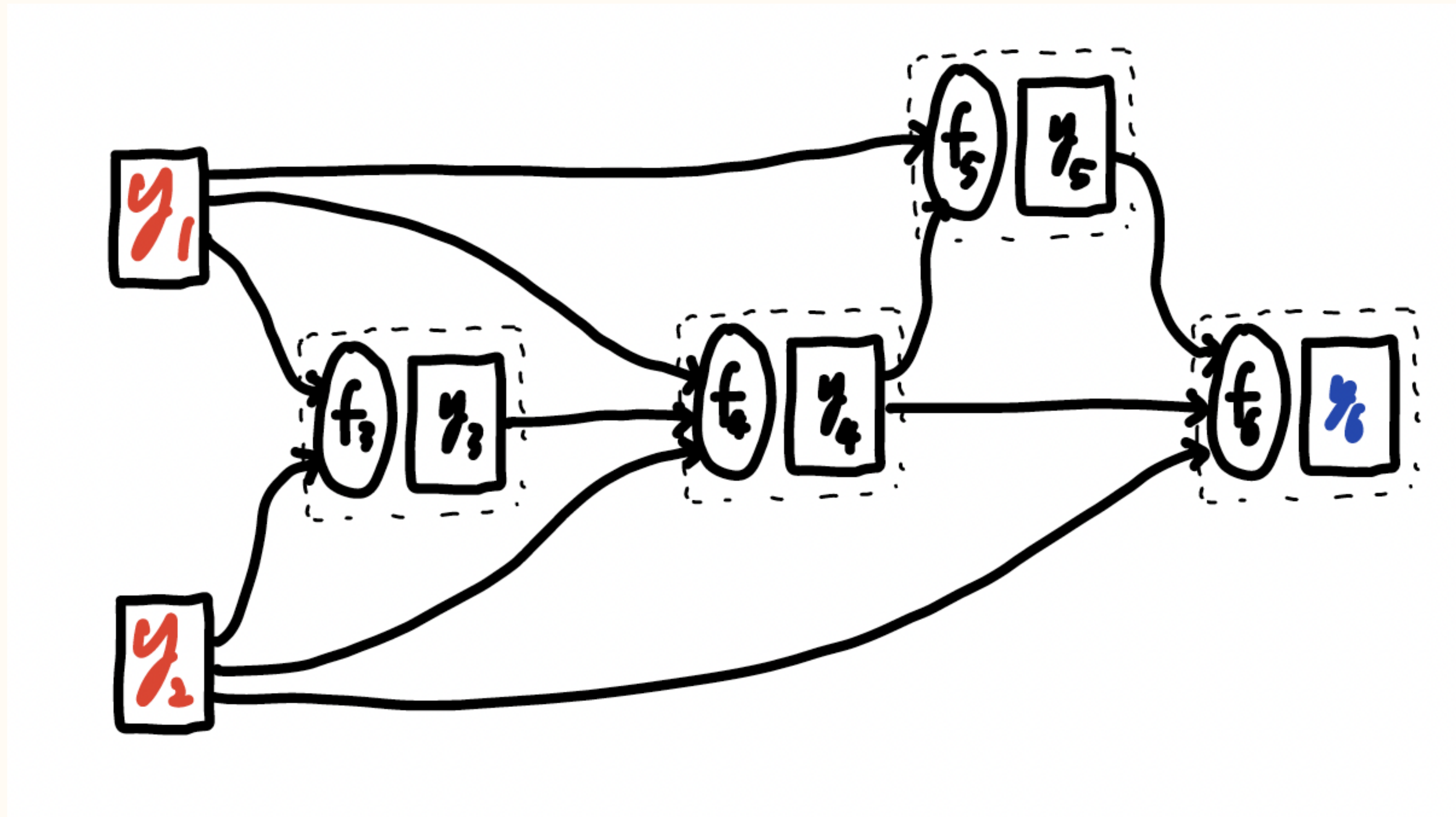
Since G is acyclic, y_v can be sequentially evaluated with a linear topological order.

```
# Forward pass given v.value for input nodes
for v in V : # In linear topological order
  if v.notInput:
    v.value = v.fn( [u.value for u->v] )
```

The values for `[u.value for u->v]` will be ready if we process `for v in V` in a linear topological order.

In practice, the computation graph $G = (V, E)$ is built during the forward evaluation.

Forward evaluation example



Backprop theorem

Backprop theorem) Assume f_v is differentiable for all non-input node v. Let o be the output node and define $\partial y_o / \partial y_o = 1$. Define

$$\frac{\partial y_o}{\partial y_v} = \sum_{w:\, v \to w} \frac{\partial y_o}{\partial y_w} \frac{\partial f_w}{\partial y_v}, \qquad \forall\, v \in V \backslash \{o\}.$$

Then, (i) this formula is well defined when it is evaluated in a reverse topological ordering (ii) it correctly computes the derivative $\partial y_o / \partial y_i$ when $v = i$ is an input node.

For input node i, of course, $\frac{\partial y_o}{\partial y_i}$ is defined by

$$y_o\Big(y_i + h, \{y_j \,|\, \text{input node } j,\, j \neq i\}\Big) = y_o\Big(y_i, \{y_j \,|\, \text{input node } j,\, j \neq i\}\Big) + \frac{\partial y_o}{\partial y_i} h + \text{higher order terms}$$

Backprop theorem

Backprop theorem) Assume f_v is differentiable for all non-input node v. Let o be the output node and define $\partial y_o / \partial y_o = 1$. Define

$$\frac{\partial y_o}{\partial y_v} = \sum_{w:\, v \to w} \frac{\partial y_o}{\partial y_w} \frac{\partial f_w}{\partial y_v}, \qquad \forall\, v \in V \backslash \{o\}.$$

Then, (i) this formula is well defined when it is evaluated in a reverse topological ordering (ii) it correctly computes the derivative $\partial y_o / \partial y_i$ when $v = i$ is an input node.

If v is not an input node, the meaning of $\frac{\partial y_o}{\partial y_v}$ is somewhat tricky. For now, we define $\frac{\partial y_o}{\partial y_v}$ through the formula of theorem.

Later, we will understand $\frac{\partial y_o}{\partial y_v}$ generally through edge severing.

Graph-form backward pass code I

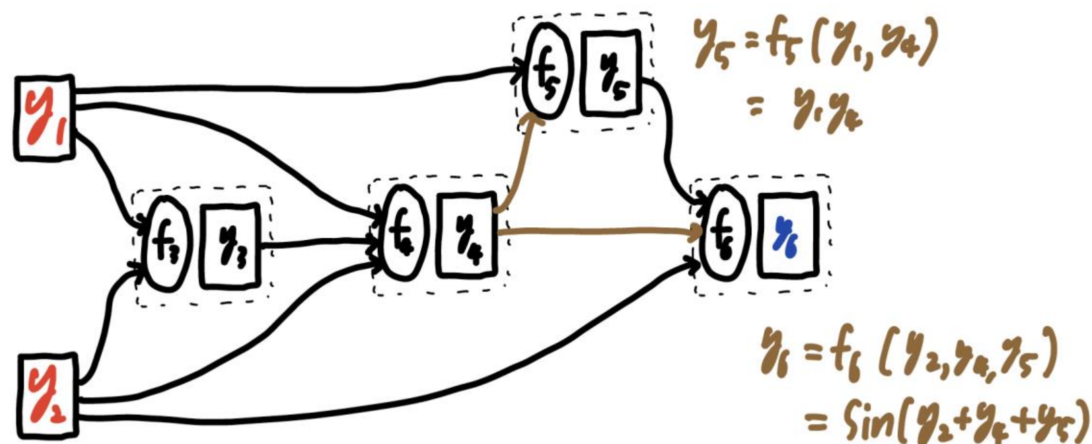
Again, assume there is only one output node. Let `v.grad` correspond to $\partial y_o / \partial y_v$.

```
# Forward pass given u.value for source nodes
for v in V :   # In linear topological order
  v.value = v.fn( [u.value for u->v] )

for v in V :   # .zero_grad()
  v.grad = 0

# Backward pass
for v in V :   # In reversed linear topological order
  for w such that v->w :
    v.grad += w.grad @ w.fn.grad(v)
    # v queries all outgoing edges
```

$$\frac{\partial y_o}{\partial y_v} = \sum_{w:\ v \to w} \frac{\partial y_o}{\partial y_w} \frac{\partial f_w}{\partial y_v}$$

The value of `w.grad` will be ready if we process `for v in V` in a reversed linear topological order.

Graph-form backward pass example I



$$\frac{\partial y_6}{\partial y_4} = \frac{\partial y_6}{\partial y_6}\frac{\partial f_6}{\partial y_4} + \frac{\partial y_6}{\partial y_5}\frac{\partial f_5}{\partial y_4}$$

$$= \cos(y_2 + y_4 + y_5) + \frac{\partial y_6}{\partial y_5} y_1$$

Note. Cumbersome as we must load y_1, y_2 which are 1-step removed from the connections $y_4 \rightarrow y_5$ or $y_4 \rightarrow y_6$

Graph-form backward pass code II

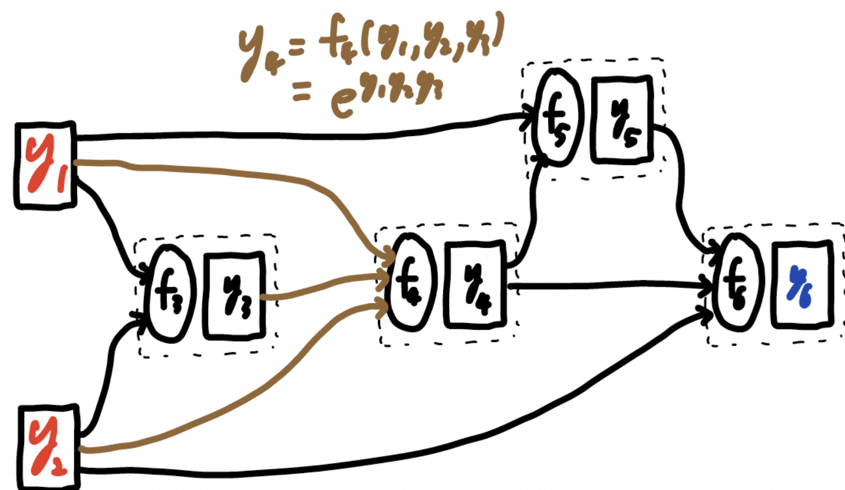
A slightly more efficient alternative is

```
# Forward pass given u.value for source nodes
for v in V :
  v.value = v.fn( [u.value for u->v] )

for v in V :   # .zero_grad()
  v.grad = 0

for v in V :   # In reversed linear topological order
  for u such that u->v :
    u.grad += v.grad @ v.fn.grad(u)
    # v sends grad through all incoming edges
```

$$\frac{\partial y_o}{\partial y_u} = \sum_{v:\ u \to v} \frac{\partial y_o}{\partial y_v} \frac{\partial f_v}{\partial y_u}$$

Formal proof that the two implementations are the same in hw.

Graph-form backward pass example II



$$\frac{\partial y_6}{\partial y_1} = \frac{\partial y_6}{\partial y_4}\frac{\partial f_4}{\partial y_1} + \boxed{\cdots} = \frac{\partial y_6}{\partial y_4} e^{y_1 y_2 y_3} y_2 y_3 + \boxed{\cdots}$$

$$\frac{\partial y_6}{\partial y_2} = \frac{\partial y_6}{\partial y_4}\frac{\partial f_4}{\partial y_2} + \boxed{\cdots} = \frac{\partial y_6}{\partial y_4} e^{y_1 y_2 y_3} y_1 y_3 + \boxed{\cdots}$$

$$\frac{\partial y_6}{\partial y_3} = \frac{\partial y_6}{\partial y_4}\frac{\partial f_4}{\partial y_3} + \boxed{\cdots} = \frac{\partial y_6}{\partial y_4} e^{y_1 y_2 y_3} y_1 y_2 + \boxed{\cdots}$$

Only the y-values of nodes immediately in consideration (connected to ≡) are used.

`no_grad` inputs

In general, however, we cannot or do not want to perform backproparation with respect to all neural network inputs. Examples include:

- Image input to classifier. (Usually. Exceptions exist.)
- Text input into tokenizer. (Tokenizer is not differentiable.)

We want backpropagation to differentiate with respect to a subset of inputs while holding other inputs fixed.

Param vs. fixed-input and graph coloring

Within $G = (V, E)$, further distinguish input nodes into: parameter vs fixed-input.
(In PyTorch, these correspond to `requires_grad=True` and `requires_grad=False`.)

Goal is to compute $\partial y_o / \partial y_p$ for parameter node p.

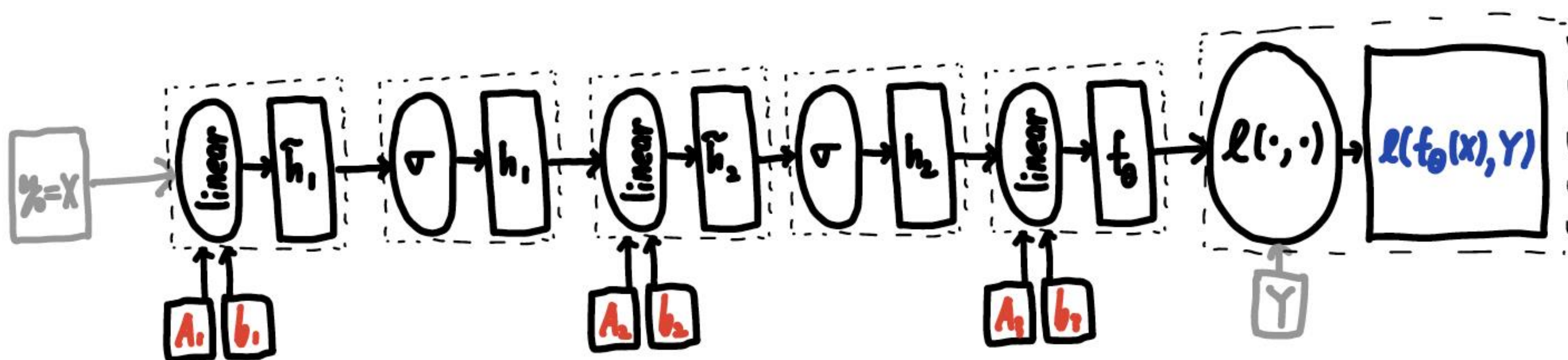
When building a computation graph, color all nodes as follows:

1. parameter is colored BLACK.
2. fixed-input is colored GREY.
3. if f_v depends only on GREY inputs, then v is GREY.
4. if f_v depends on one or more BLACK inputs, then v is BLACK.

Forward evaluation with graph coloring

```
# Forward pass given v.value for input nodes
for v in V : # In linear topological order
  if v.parameter :
    v.color = BLACK
  if v.fixedInput :
    v.color = GREY
  else :
    v.value = v.fn( [u.value for u->v] )
    if all([u.color == GREY for u->v]) :
      v.color = GREY
    else :
      v.color = BLACK
```

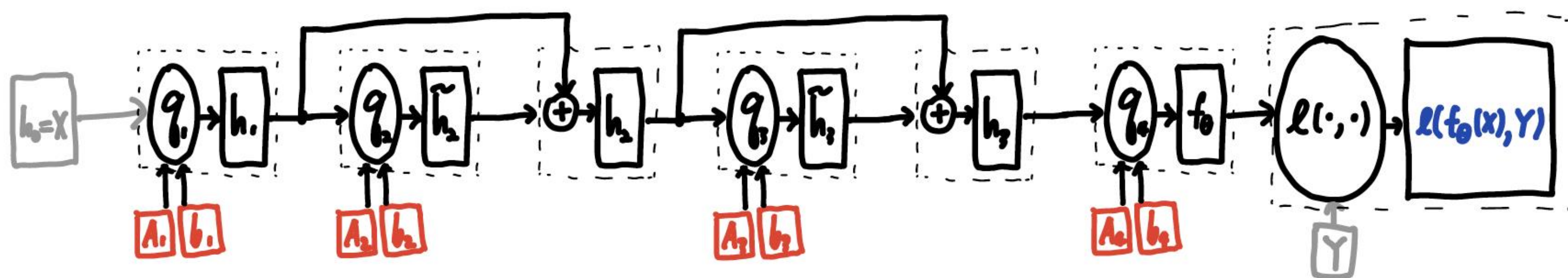
Forward evaluation with MLP



$$h_i = \sigma(\underbrace{A_i h_{i-1} + b_i}_{= \tilde{h}_i}) \quad \text{for } i = 1, \ldots, L-1$$

$$f_\theta(x) = A_L h_{L-1} + b_L$$

Forward evaluation with ResNet

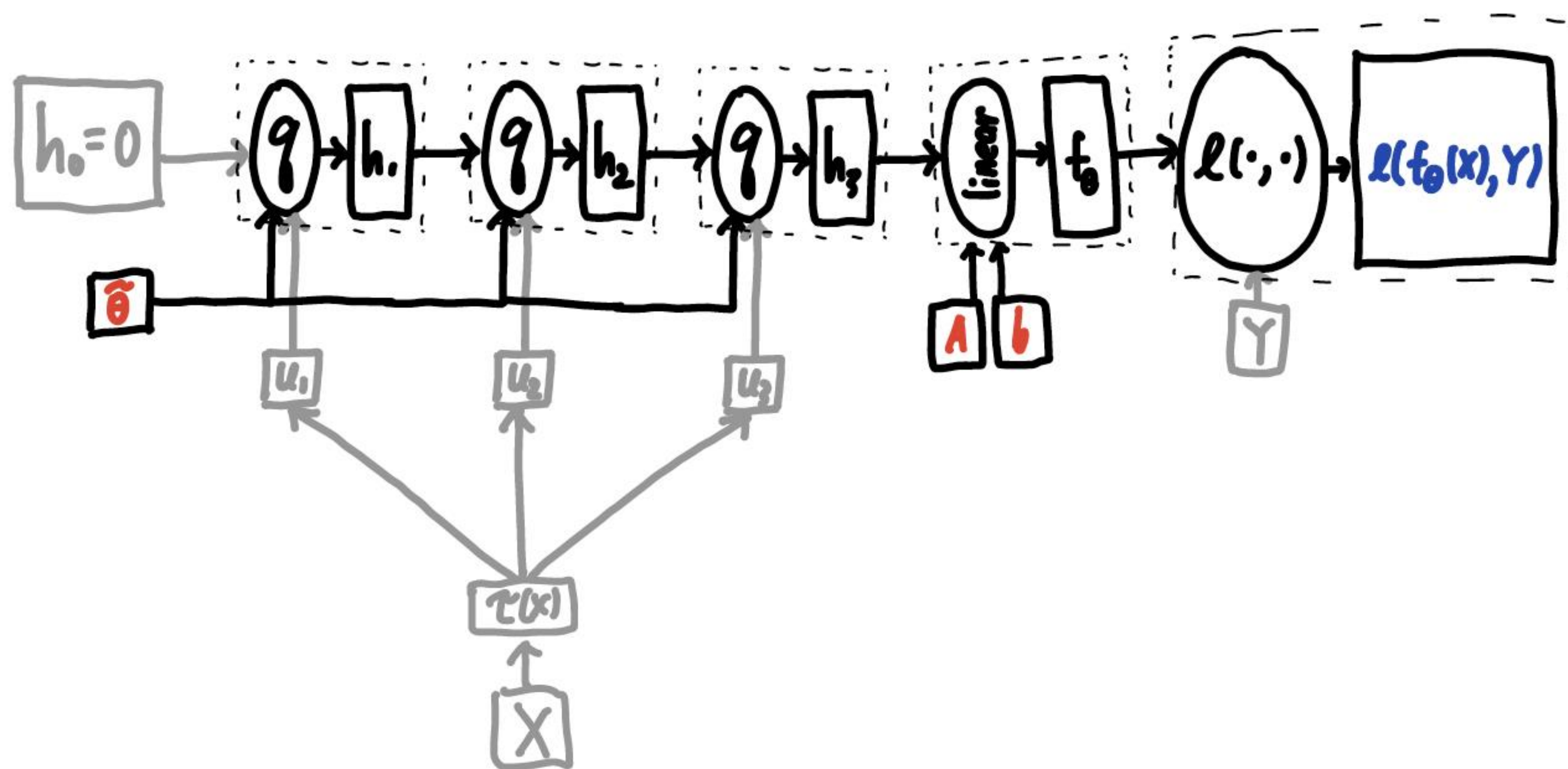


$$h_1 = g_1(A_1, b_1, h_0)$$

$$h_i = \underbrace{g_i(A_i, b_i, h_{i-1})}_{= \tilde{h}_i} + h_{i-1} \quad \text{for } i = 2, \ldots, L-1$$

$$f_\theta(X) = g_L(A_L, b_L, h_{L-1})$$

Forward evaluation with RNN



$$h_0 = 0$$

$$h_t = g_{\tilde{\theta}}(h_{t-1}, u_t) \quad t = 1, \dots, T$$

$$f_\theta(x) = A h_T + b$$

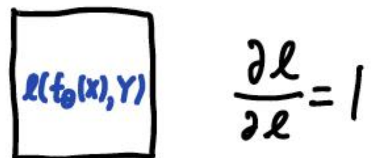
Backprop theorem

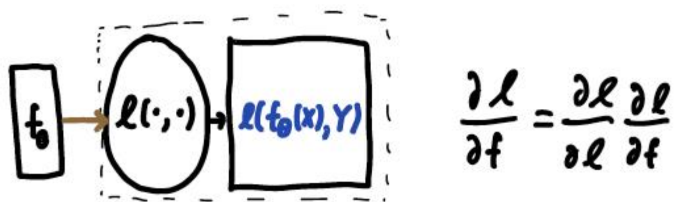
Backprop theorem) Assume that if $u \to v$ and u and v are BLACK, then f_v is differentiable with respect to y_u, i.e., $\partial f_v / \partial y_u$ exists. Let o be the output node and define $\partial y_o / \partial y_o = 1$. Define

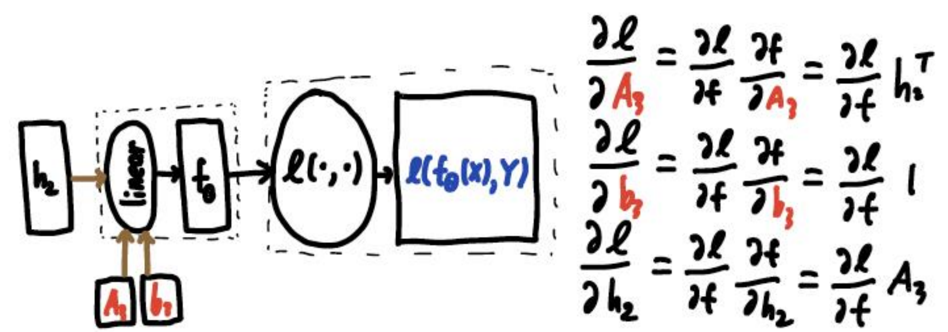
$$\frac{\partial y_o}{\partial y_v} = \sum_{w:\ v \to w} \frac{\partial y_o}{\partial y_w} \frac{\partial f_w}{\partial y_v}, \qquad \forall\, v \in V \setminus \{o\} \text{ and } v \text{ is BLACK.}$$

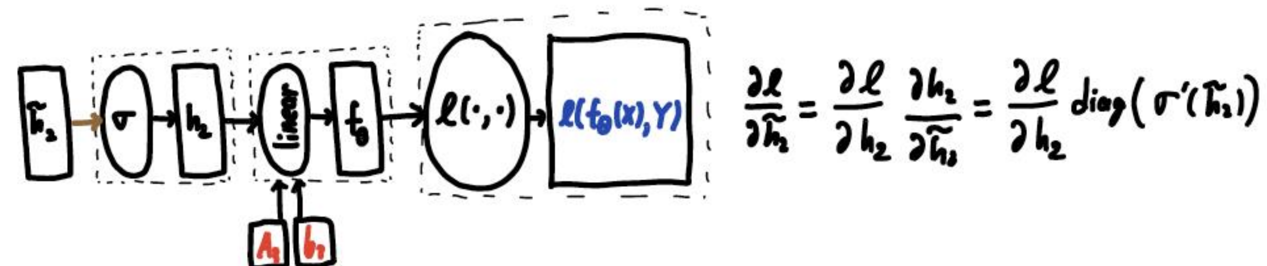
Then, (i) this formula is well defined when it is evaluated in a reverse topological ordering (while skipping GREY nodes) (ii) it correctly computes the derivative $\partial y_o / \partial y_p$ when $v = p$ is a parameter input node.

Backprop with MLP

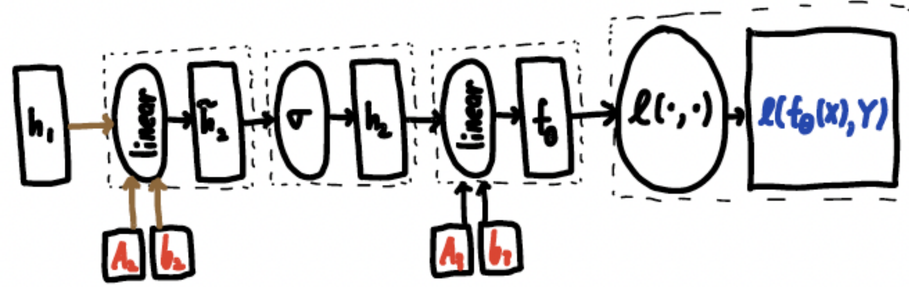








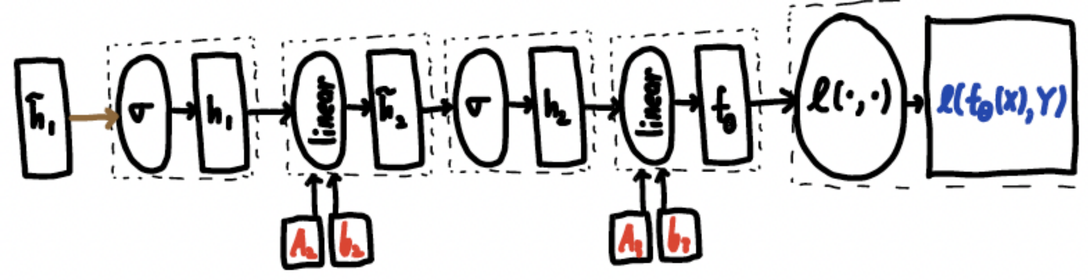
Backprop with MLP



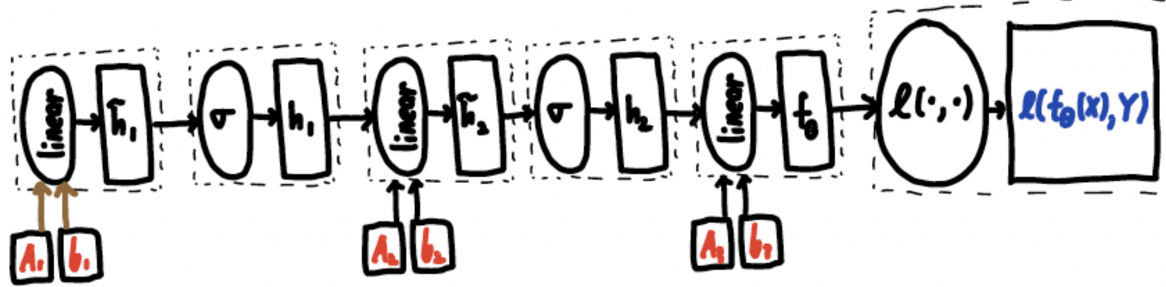
$$\frac{\partial \ell}{\partial A_2} = \frac{\partial \ell}{\partial \tilde{h}_2}\frac{\partial \tilde{h}_2}{\partial A_2} = \frac{\partial \ell}{\partial \tilde{h}_2}^T h_1^T$$

$$\frac{\partial \ell}{\partial b_2} = \frac{\partial \ell}{\partial \tilde{h}_2}\frac{\partial \tilde{h}_2}{\partial b_2} = \frac{\partial \ell}{\partial \tilde{h}_2}$$

$$\frac{\partial \ell}{\partial h_1} = \frac{\partial \ell}{\partial \tilde{h}_2}\frac{\partial \tilde{h}_2}{\partial h_1} = \frac{\partial \ell}{\partial \tilde{h}_2} A_2$$



$$\frac{\partial \ell}{\partial \tilde{h}_1} = \frac{\partial \ell}{\partial h_1}\frac{\partial h_1}{\partial \tilde{h}_1} = \frac{\partial \ell}{\partial h_1} \mathrm{diag}(\sigma'(\tilde{h}_1))$$

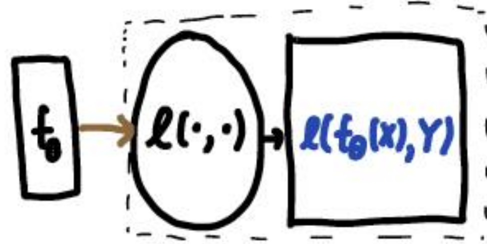


$$\frac{\partial \ell}{\partial A_1} = \frac{\partial \ell}{\partial \tilde{h}_1}\frac{\partial \tilde{h}_1}{\partial A_1} = \frac{\partial \ell}{\partial \tilde{h}_1}^T h_0^T$$

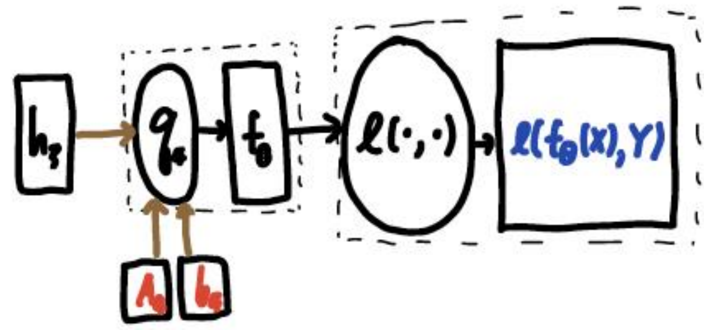
$$\frac{\partial \ell}{\partial b_1} = \frac{\partial \ell}{\partial \tilde{h}_1}\frac{\partial \tilde{h}_1}{\partial b_1} = \frac{\partial \ell}{\partial \tilde{h}_1}$$

Backprop with ResNet

$$\frac{\partial \ell}{\partial \ell} = 1$$



$$\frac{\partial \ell}{\partial f} = \frac{\partial \ell}{\partial \ell}\frac{\partial \ell}{\partial f}$$

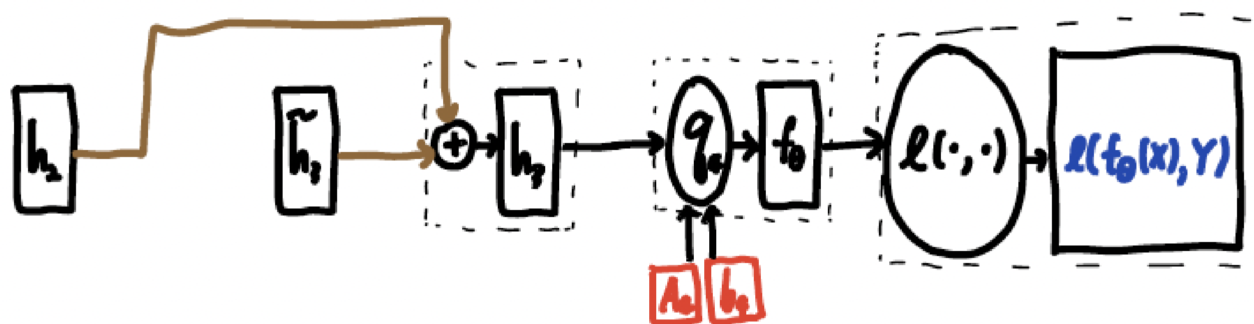


$$\frac{\partial \ell}{\partial A_4} = \frac{\partial \ell}{\partial f}\frac{\partial g_4}{\partial A_4}$$

$$\frac{\partial \ell}{\partial b_4} = \frac{\partial \ell}{\partial f}\frac{\partial g_4}{\partial b_4}$$

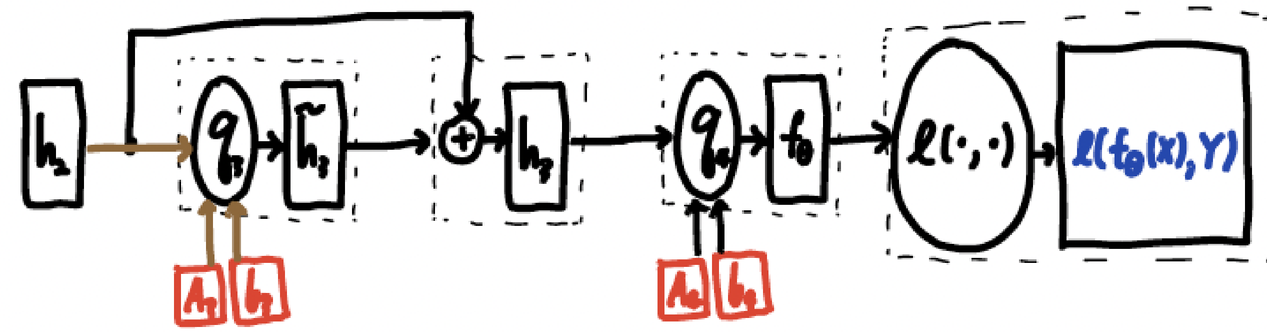
$$\frac{\partial \ell}{\partial h_3} = \frac{\partial \ell}{\partial f}\frac{\partial g_4}{\partial h_3}$$

Backprop with ResNet



$$\frac{\partial \ell}{\partial \tilde{h}_3} = \frac{\partial \ell}{\partial h_3}\frac{\partial h_3}{\partial \tilde{h}_3} = \frac{\partial \ell}{\partial h_3}$$

$$\frac{\partial \ell}{\partial h_2} \mathrel{+}= \frac{\partial \ell}{\partial h_3}\frac{\partial h_3}{\partial h_2} = \frac{\partial \ell}{\partial h_3}$$

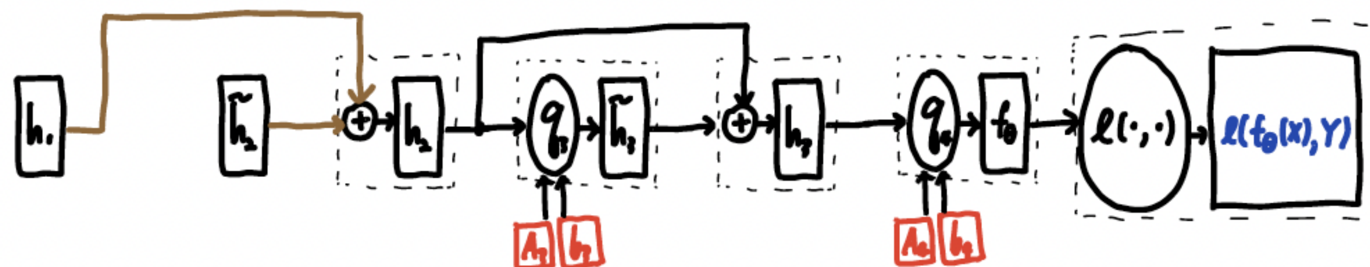


$$\frac{\partial \ell}{\partial A_3} = \frac{\partial \ell}{\partial \tilde{h}_3}\frac{\partial g_3}{\partial A_3}$$

$$\frac{\partial \ell}{\partial b_3} = \frac{\partial \ell}{\partial \tilde{h}_3}\frac{\partial g_3}{\partial b_3}$$

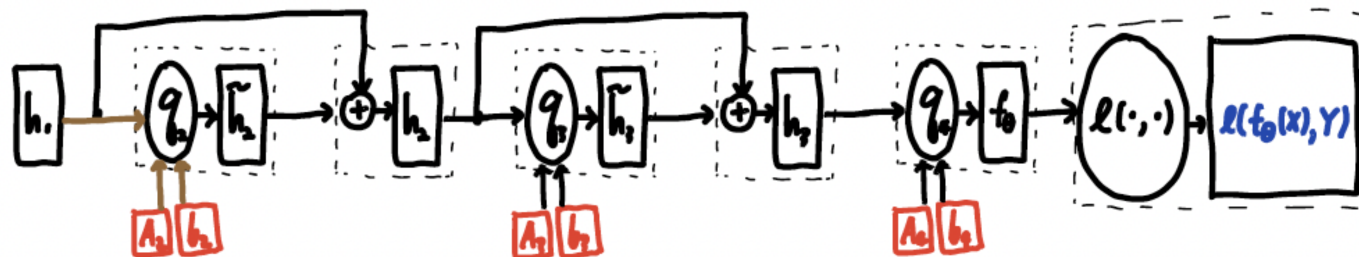
$$\frac{\partial \ell}{\partial h_2} \mathrel{+}= \frac{\partial \ell}{\partial \tilde{h}_3}\frac{\partial g_3}{\partial h_2}$$

Backprop with ResNet



$$\frac{\partial \ell}{\partial \tilde{h}_2} = \frac{\partial \ell}{\partial h_2}\frac{\partial h_2}{\partial \tilde{h}_2} = \frac{\partial \ell}{\partial h_2}$$

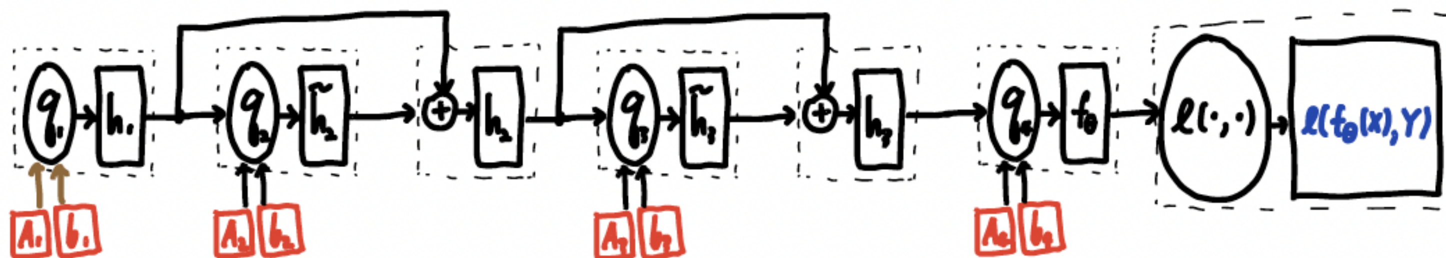
$$\frac{\partial \ell}{\partial h_1} \mathrel{+}= \frac{\partial \ell}{\partial h_2}\frac{\partial h_2}{\partial h_1} = \frac{\partial \ell}{\partial h_2}$$



$$\frac{\partial \ell}{\partial A_2} = \frac{\partial \ell}{\partial \tilde{h}_2}\frac{\partial g_2}{\partial A_2}$$

$$\frac{\partial \ell}{\partial b_2} = \frac{\partial \ell}{\partial \tilde{h}_2}\frac{\partial g_2}{\partial b_2}$$

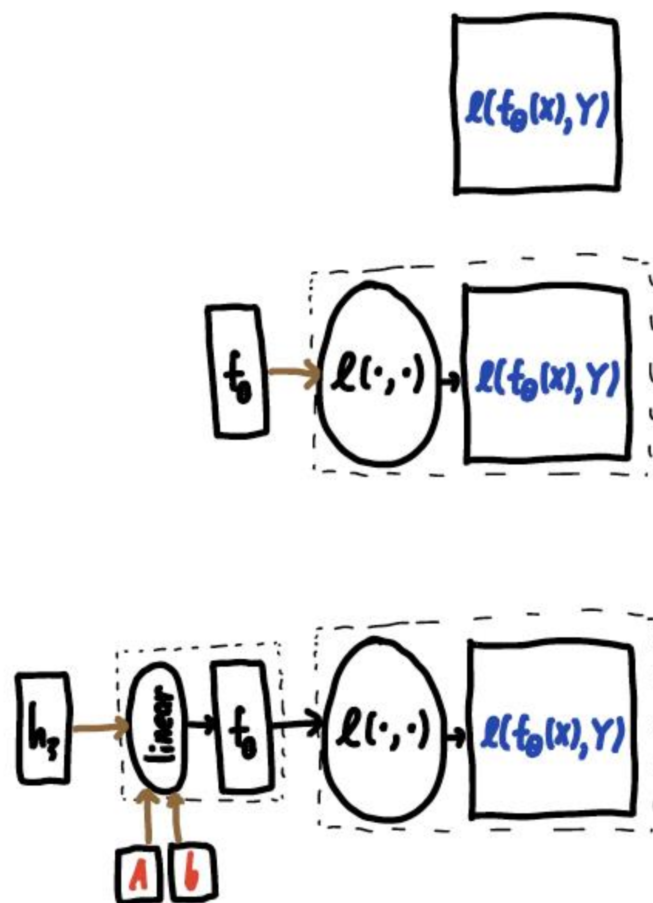
$$\frac{\partial \ell}{\partial h_1} \mathrel{+}= \frac{\partial \ell}{\partial \tilde{h}_2}\frac{\partial g_2}{\partial h_1}$$



$$\frac{\partial \ell}{\partial A_1} = \frac{\partial \ell}{\partial h_1}\frac{\partial g_1}{\partial A_1}$$

$$\frac{\partial \ell}{\partial b_1} = \frac{\partial \ell}{\partial h_1}\frac{\partial g_1}{\partial b_1}$$

Backprop with RNN



$$\frac{\partial \ell}{\partial \ell} = 1$$

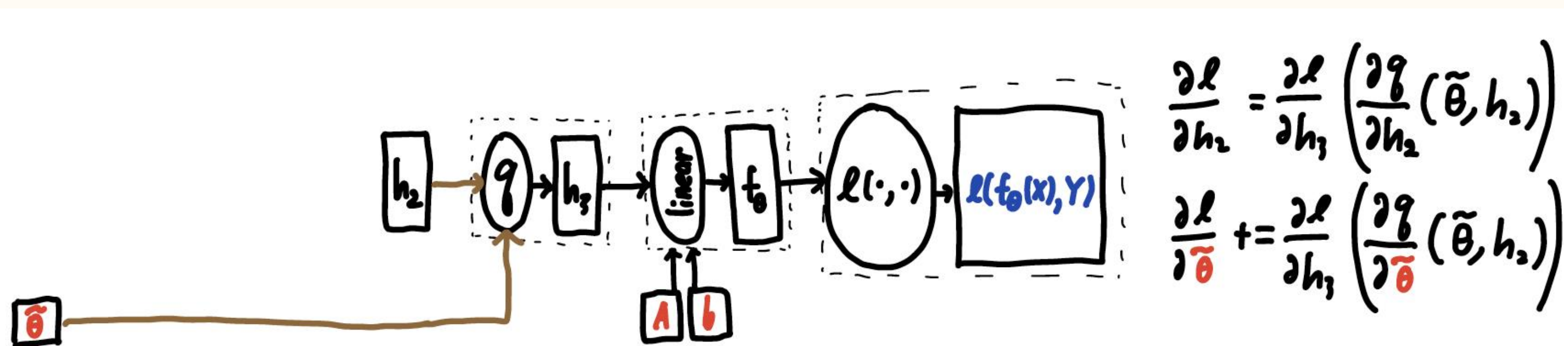
$$\frac{\partial \ell}{\partial f} = \frac{\partial \ell}{\partial \ell}\frac{\partial \ell}{\partial f}$$

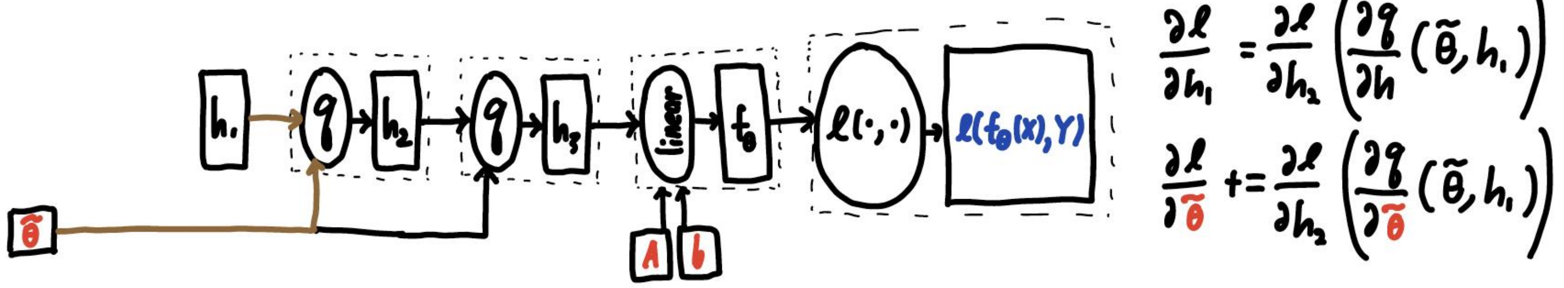
$$\frac{\partial \ell}{\partial A} = \frac{\partial \ell}{\partial f}\frac{\partial f}{\partial A} = \frac{\partial \ell}{\partial f} h_3^T$$

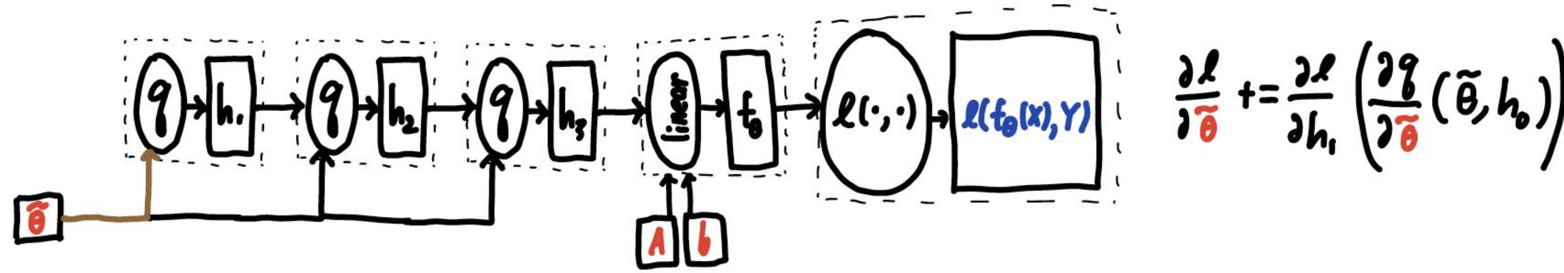
$$\frac{\partial \ell}{\partial b} = \frac{\partial \ell}{\partial f}\frac{\partial f}{\partial b} = \frac{\partial \ell}{\partial f} 1$$

$$\frac{\partial \ell}{\partial h_3} = \frac{\partial \ell}{\partial f}\frac{\partial f}{\partial h_3} = \frac{\partial \ell}{\partial f} A$$

Backprop with RNN





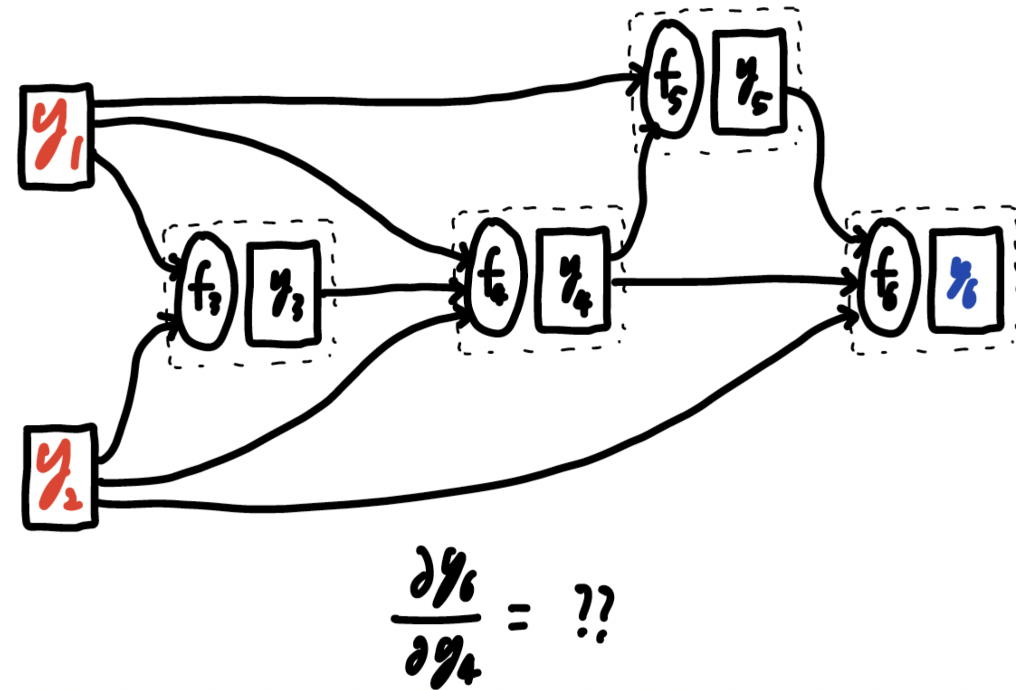


Meaning of intermediate partials

For a v that is not an input node, what does $\frac{\partial y_o}{\partial y_v}$ mean?

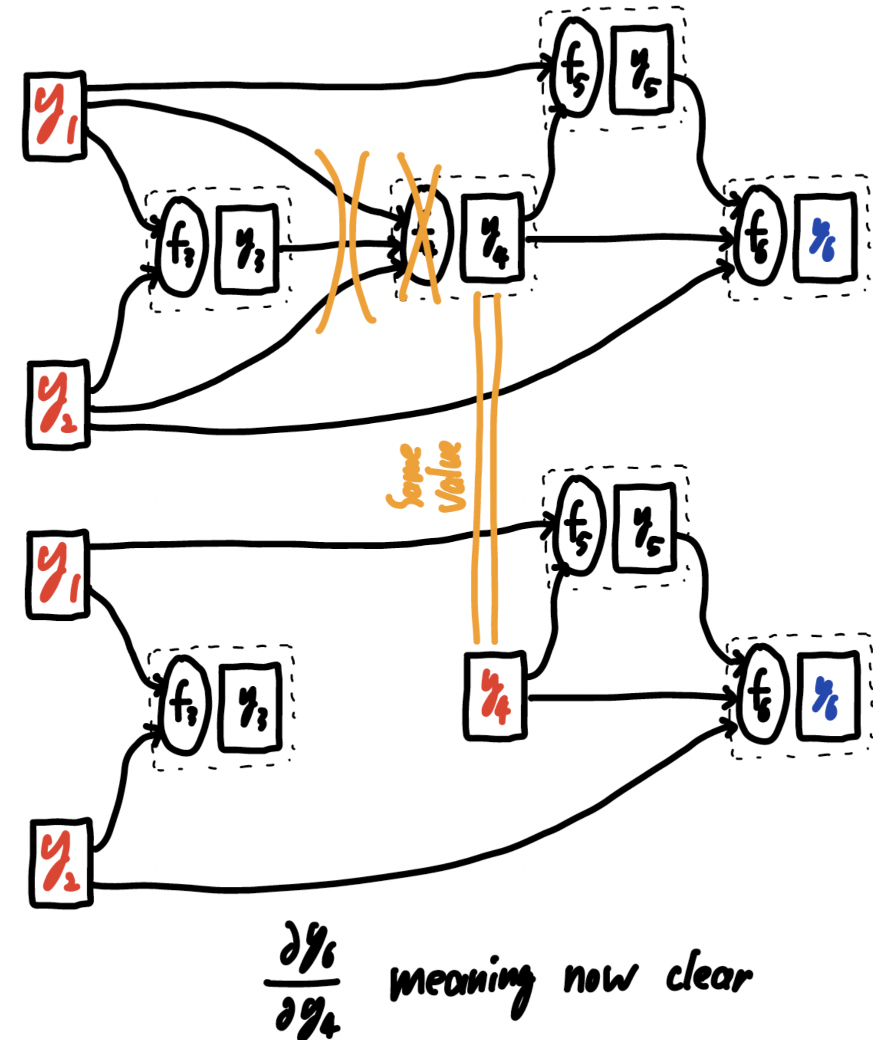
Of course, it means how y_o changes infinitesimally when y_v changes infinitestimally right?

However, what does it mean to “change y_v”
when it is not an indepenent input?
How should the inputs to v change as y_v changes?



Intermediate partials with edge severing

Let $G = (V, E)$ be a computation graph with non-input (BLACK) node $v \in V$. Then $\partial y_o / \partial y_v$ can be understood by severing all inputs to v and considering it an input node.



Intermediate partials with edge severing

More precisely, construct a graph $G' = (V, E')$ as follows.

- G and G' share the same node set V.
- The edge set E is $E' = \{(u, w) \mid (u, w) \in E, w \neq v\}$. (Edges into v are severed.)
- If i is an input node in G, then set its value y_i in G' to be the same as the y_i in G.
- For v, set its value y_v in G' to be the same as the y_i in G.
- For all other nodes u, evaluate its value y_u with the same evaluation funciton f_u.

Then $\frac{\partial y_o}{\partial y_v}$ of G' (with v an input) is the same as $\frac{\partial y_o}{\partial y_v}$ of G (with v not an input).

In this sense, we can understand $\frac{\partial y_o}{\partial y_v}$ as a derivative.

Micrograd lecture

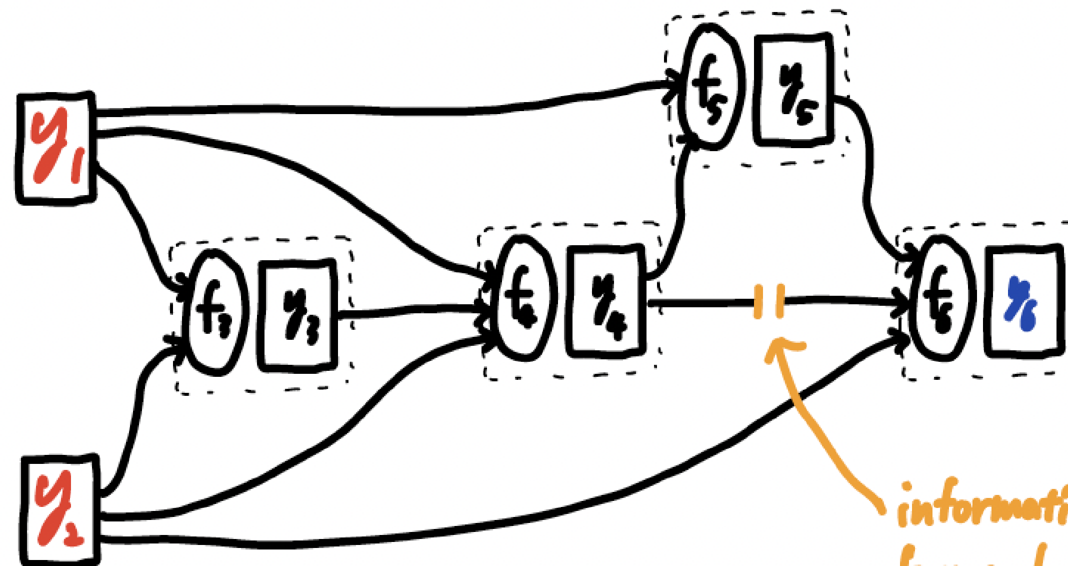
Andrej Karpathy's YouTube lecture:

https://youtu.be/VMj-3S1tku0?si=6I_OUqCp_C9lhFIU

A wonderful lecture spelling out how to implement backpropagation.

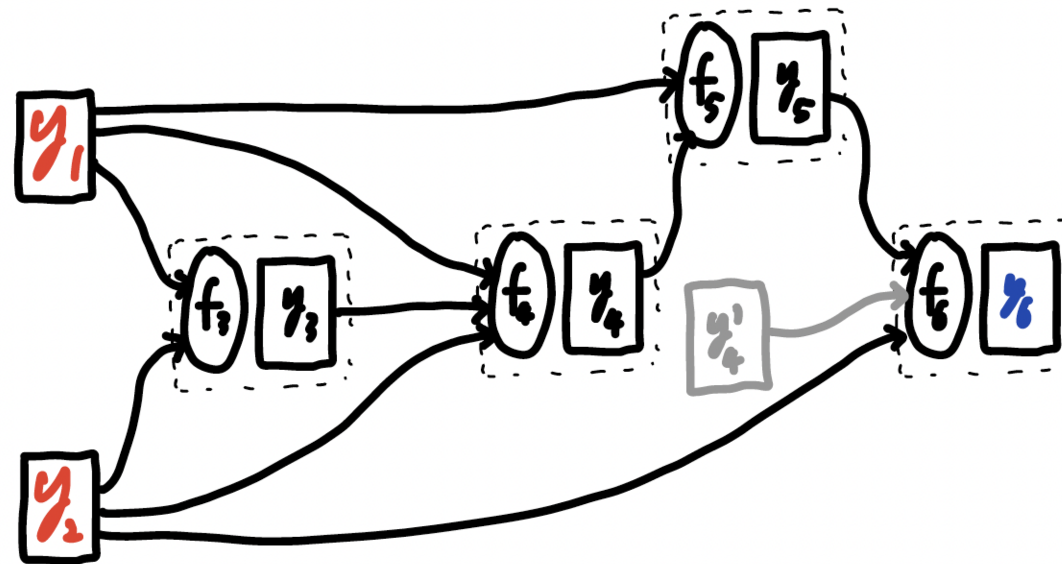
Stop-gradient

The stop-gradient operation allows information to flow through the edge during forward evaluation, but stops the gradient flow during backpropagation.



Stop-gradient

Equivalently, the stop-gradient operation creates a node with fixed input that contains a copy of the numerical value.



Stop-gradient

In PyTorch, the stop-gradient operator is achieved by `detach()`.

In RL, the notation $[\![\cdot]\!]$ is commonly used. (So $y_6 = f_6(y_2, [\![y_4]\!], y_5)$ in previous example.)

Mathematically, the stop gradient operation modifies the backprop formula to

$$\frac{\partial y_o}{\partial y_v} = \sum_{\substack{w:\ v \to w \\ \text{exclude } v \to w \text{ if stop-grad}}} \frac{\partial y_o}{\partial y_w} \frac{\partial f_w}{\partial y_v}, \qquad \forall\, v \in V \backslash \{o\} \text{ and } v \text{ is BLACK.}$$

Computation and memory cost of backprop

Consider an MLP with depth L with uniform width w.

Backprop costs:

- (w^2L) computation in forward pass,
- $\mathcal{O}(wL)$ memory to store intermediate values, and
- $\mathcal{O}(w^2L)$ computation in backward pass.

$$
\begin{aligned}
\mathcal{L} &= \frac{1}{2}(y_L - Y_{\text{data}})^2 \\
y_L &= \sigma(A_L y_{L-1} + b_L) \\
y_{L-1} &= \sigma(A_{L-1} y_{L-2} + b_{L-1}) \\
&\vdots \\
y_2 &= \sigma(A_2 y_1 + b_2) \\
y_1 &= \sigma(A_1 x + b_1),
\end{aligned}
$$

$$
\begin{aligned}
\frac{\partial \mathcal{L}}{\partial y_L} &= (y_L - Y_{\text{data}}) \\
\frac{\partial \mathcal{L}}{\partial b_\ell} &= \frac{\partial \mathcal{L}}{\partial y_\ell} \operatorname{diag}\left(\sigma'(A_\ell y_{\ell-1} + b_\ell)\right) \\
\frac{\partial \mathcal{L}}{\partial A_\ell} &= \operatorname{diag}\left(\sigma'(A_\ell y_{\ell-1} + b_\ell)\right) \left(\frac{\partial \mathcal{L}}{\partial y_\ell}\right)^\intercal y_{\ell-1}^\intercal \\
\frac{\partial \mathcal{L}}{\partial y_{\ell-1}} &= \frac{\partial \mathcal{L}}{\partial y_\ell} \operatorname{diag}\left(\sigma'(A_\ell y_{\ell-1} + b_\ell)\right) A_\ell
\end{aligned}
$$

Gradient checkpointing and recomputation

Gradient checkpointing with interval k:

- Instead of storing $y_0, y_1, y_2, \ldots, y_L$, store $y_0, y_k, y_{2k}, \ldots$.
- To backprop from $y_{(s+1)k}$ to y_{sk}, load y_{sk} and recompute $y_{sk+1}, y_{sk+2}, \ldots, y_{(s+1)k-1}$ and temporarily store them in memory.

Backprop costs:

- $(w^2 L)$ computation in forward pass,
- $\mathcal{O}\left(\frac{wL}{k} + wk\right)$ memory to store intermediate values, and
- $\mathcal{O}(w^2 L)$ computation in backward pass.

$$
\begin{aligned}
\mathcal{L} &= \frac{1}{2}(y_L - Y_{\text{data}})^2 \\
y_L &= \sigma(A_L y_{L-1} + b_L) \\
y_{L-1} &= \sigma(A_{L-1} y_{L-2} + b_{L-1}) \\
&\vdots \\
y_2 &= \sigma(A_2 y_1 + b_2) \\
y_1 &= \sigma(A_1 x + b_1),
\end{aligned}
$$

$$
\begin{aligned}
\frac{\partial \mathcal{L}}{\partial y_L} &= (y_L - Y_{\text{data}}) \\
\frac{\partial \mathcal{L}}{\partial b_\ell} &= \frac{\partial \mathcal{L}}{\partial y_\ell} \operatorname{diag}\left(\sigma'(A_\ell y_{\ell-1} + b_\ell)\right) \\
\frac{\partial \mathcal{L}}{\partial A_\ell} &= \operatorname{diag}\left(\sigma'(A_\ell y_{\ell-1} + b_\ell)\right) \left(\frac{\partial \mathcal{L}}{\partial y_\ell}\right)^{\intercal} y_{\ell-1}^{\intercal} \\
\frac{\partial \mathcal{L}}{\partial y_{\ell-1}} &= \frac{\partial \mathcal{L}}{\partial y_\ell} \operatorname{diag}\left(\sigma'(A_\ell y_{\ell-1} + b_\ell)\right) A_\ell
\end{aligned}
$$

Gradient checkpointing and recomputation

With $k = \sqrt{L}$, we have $\mathcal{O}(w\sqrt{L})$ memory cost.

Trading off computation and memory. The computation cost of backprop doubles, but memory usage is significantly reduced.

(n an MLP, parameters and their gradient require $\mathcal{O}(wL)$ memory to store, which outweighs the $\mathcal{O}(wL)$ memory cost of plain backprop anyway. However, the ratio of intermediate computed values to the number of parameters becomes much larger with convolutional and attention layers, so the reduction to $\mathcal{O}(w\sqrt{L})$ memory cost is much more meaningful.

T. Chen, B. Xu, C. Zhang, and C. Guestrin, Training deep nets with sublinear memory cost, *arXiv*, 2016.
A. Gruslys, R. Munos, I. Danihelka, M. Lanctot, and A. Graves, Memory-efficient backpropagation through time, *NeurIPS*, 2016.

Memory vs. computation

In deep learning, computational and hardware bottleneck primarily comes from memory, rather than computation.

Computation cost includes

- Basic arithmetic and the evaluation of special functions such as $\exp$ or $\log$.

Memory cost includes

- Size of GPU memory required to store neural network weights and other information.
- Memory IO. The reading from and writing to memory, especially HBM.