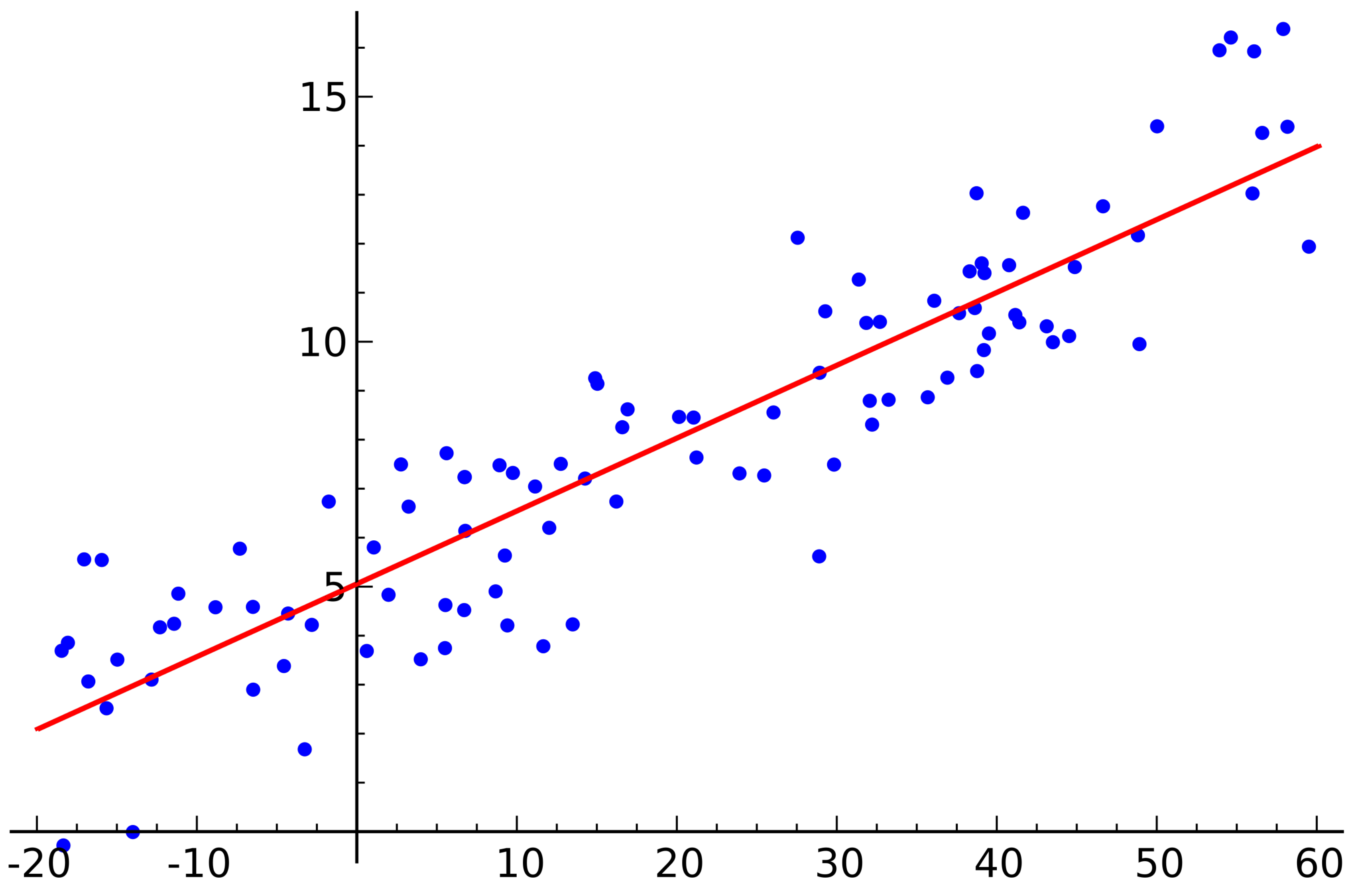
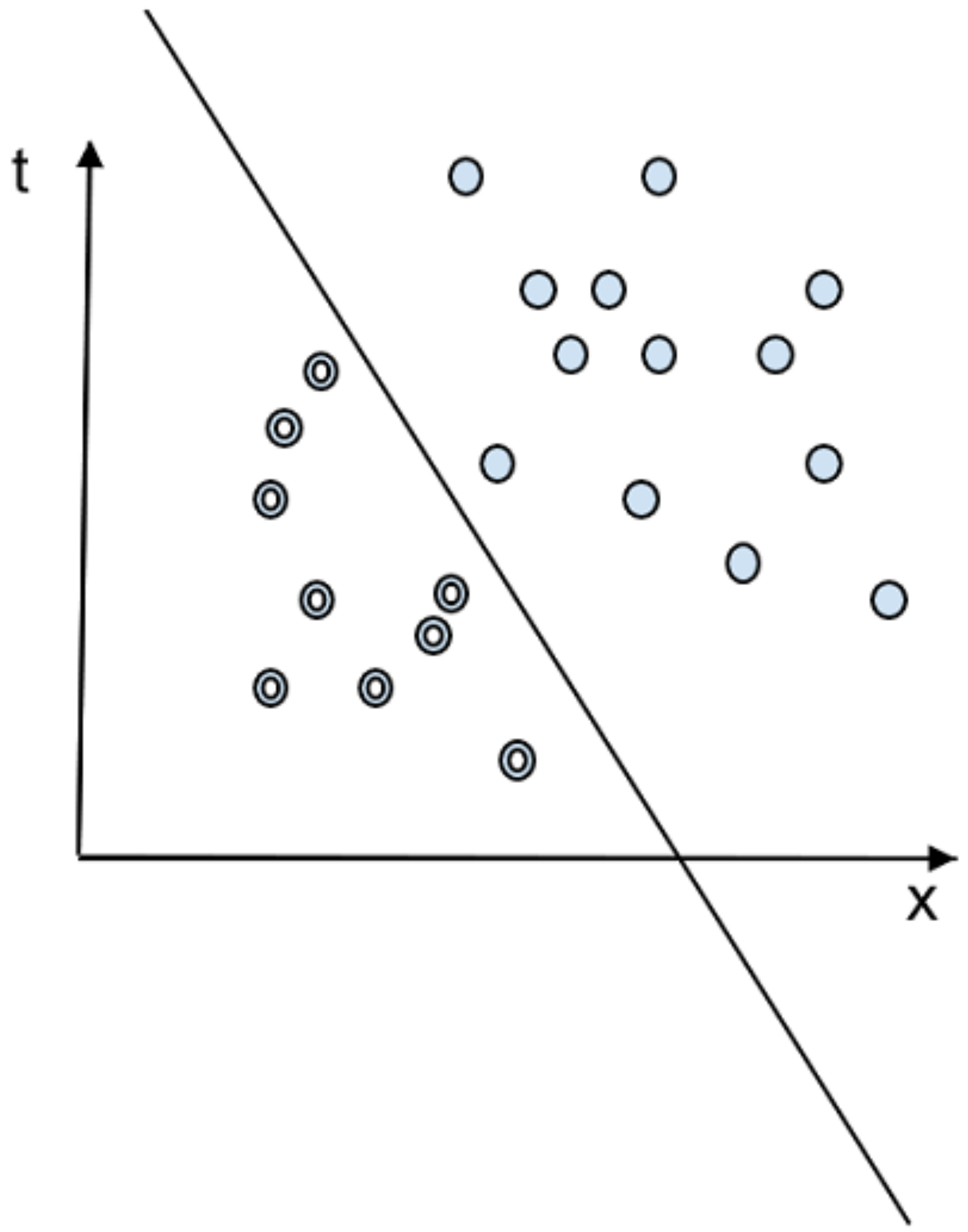
EEECE695D: Efficient ML Systems

Compute / Memory: Deep Neural Networks

Recap

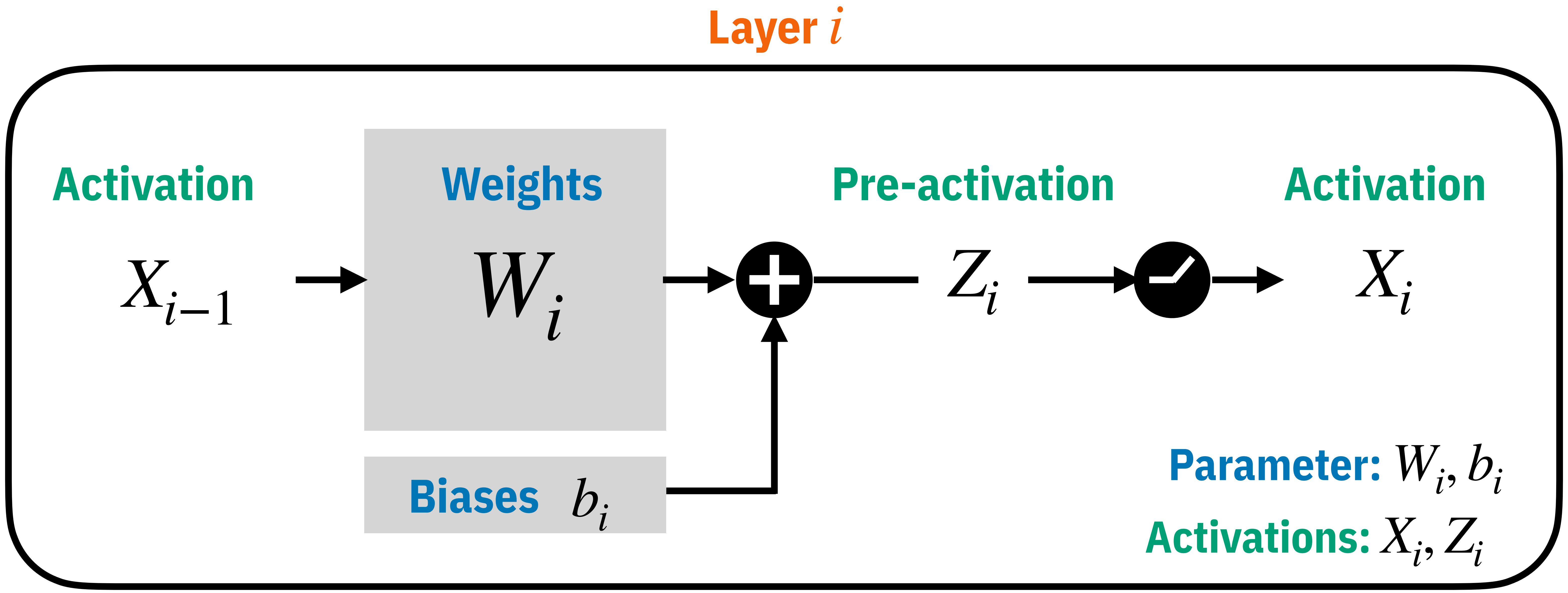
- Compute and memory requirements of linear models (linear regression, perceptron)
    - Training vs. Inference
    - Dependency on optimization methods: Exact vs. Indirect





Deep Neural Networks

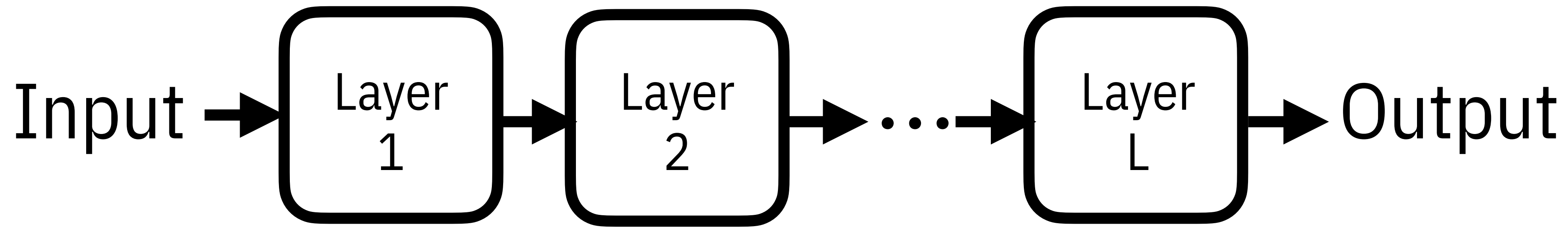
- **Neural networks.** Roughly speaking, a (directed acyclic) graph of layers



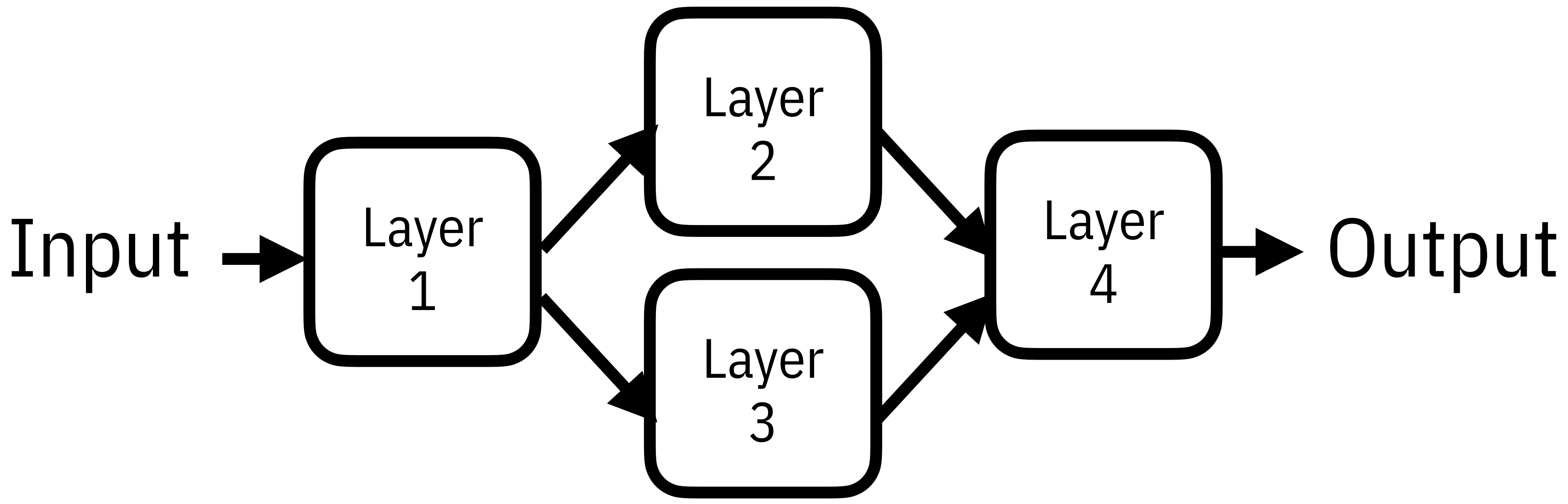
$$Z_i = W_i X_{i-1} + b_i$$

$$X_i = \sigma(Z_i)$$

Feedforward

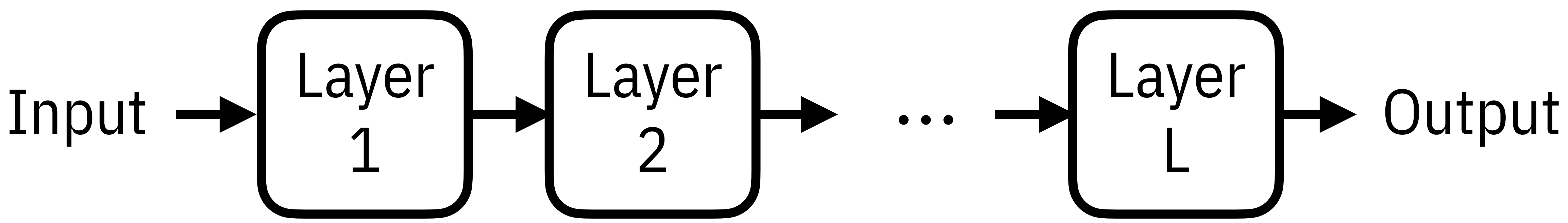


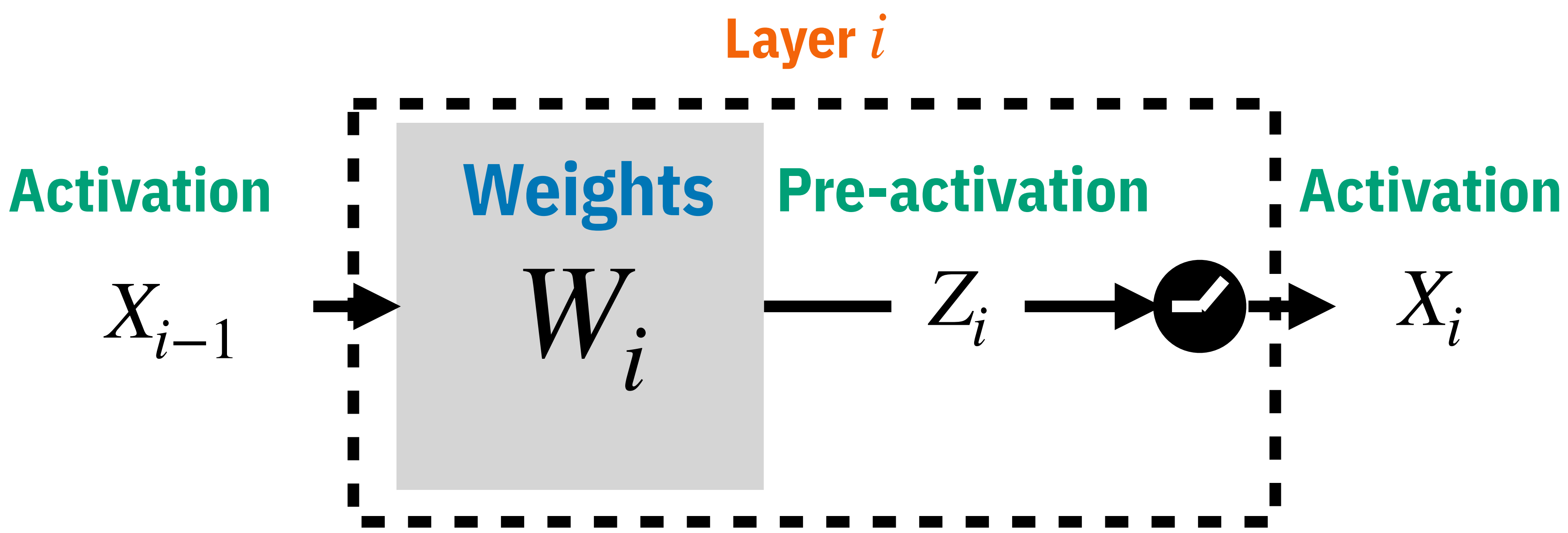
In general



DNN Inference

- Let us consider a simple case: **feedforward net without bias**





- **Routine.** The inference procedure can be simplified as

\# Currently X_{i-1} is on cache

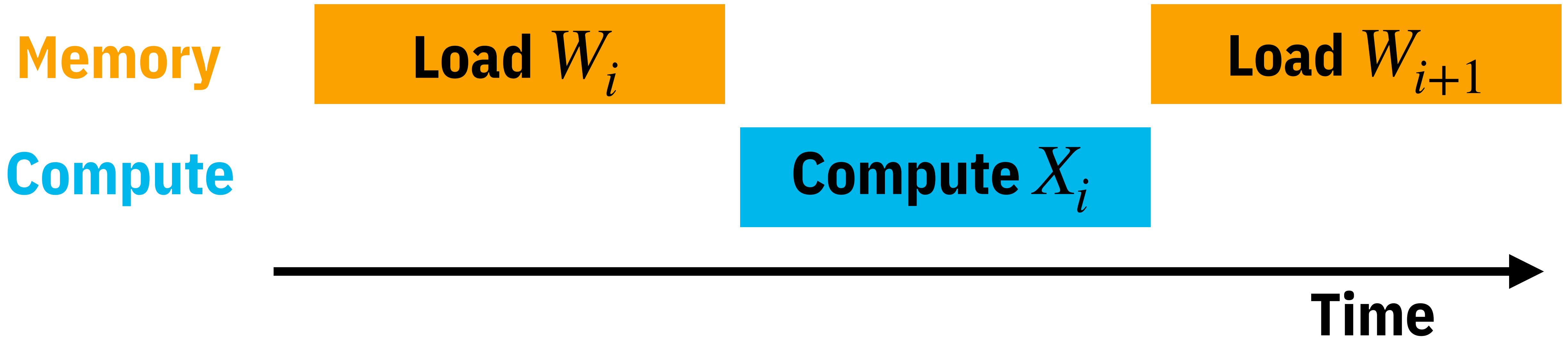
1. Load W_i on cache ← **Memory bandwidth**
2. Compute $X_i = \sigma(W_i X_{i-1})$ ← **Compute**
3. Store X_i on cache #free X_{i-1}
4. Repeat 1—3.

Note. Cache (on-chip) memory is very different from the memory that causes "CUDA out of memory error."
The latter is an external memory— NVIDIA RTX A6000 has 48GB GDDR6 external memory and 6MB L2 cache.

DNN Inference

\# Currently X_{i-1} is on cache

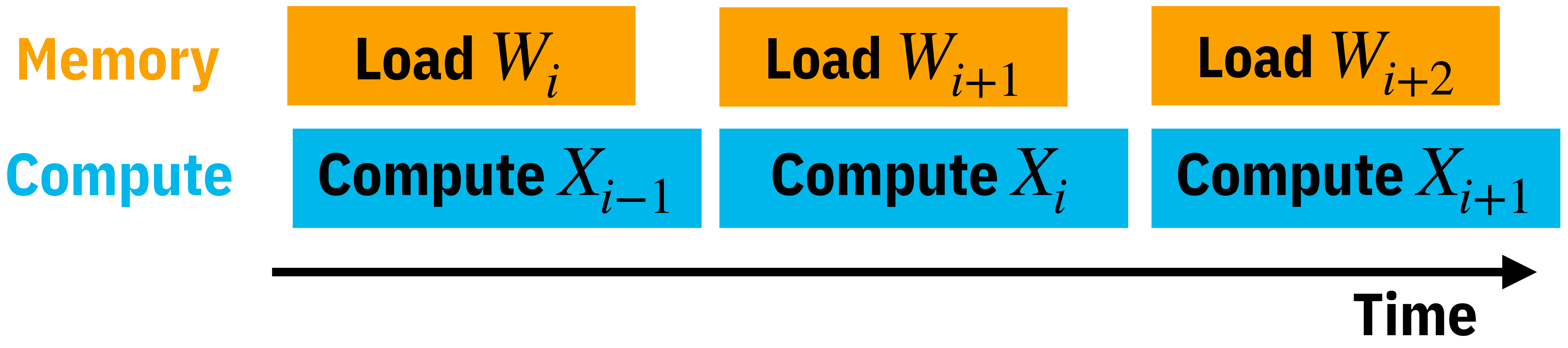
1. Load W_i on cache
2. Compute $X_i = \sigma(W_i X_{i-1})$
3. Store X_i on cache #free X_{i-1}
4. Repeat 1—3.



- **Overlapping.** A common practice is to overlap the loading and processing.

\# Currently X_{i-1} and W_i on cache

1. Compute $X_i = \sigma(W_i X_{i-1})$ + Load W_{i+1} on other cache
2. Store X_i on cache
3. Repeat 1—2.



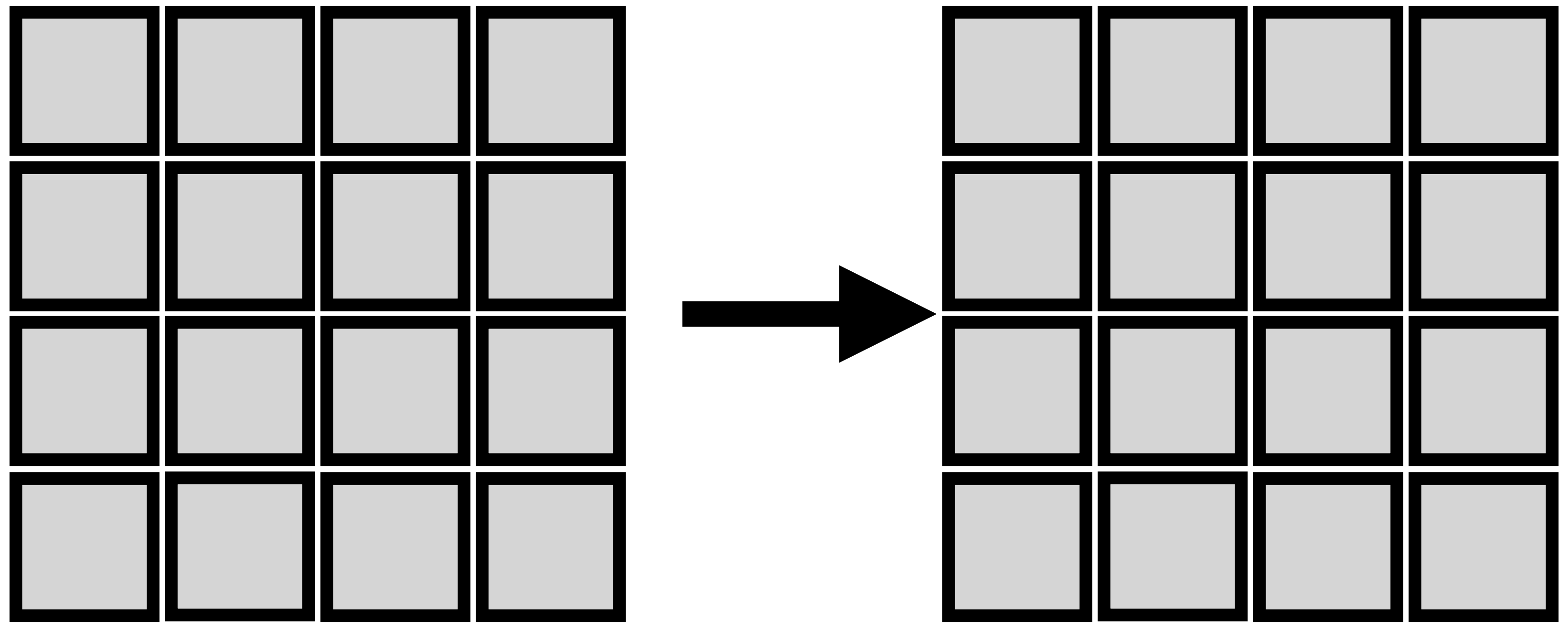
Note. An example of compute-bound case

Note. Cache should be able to store two weights and two activations—double buffering

DNN Inference

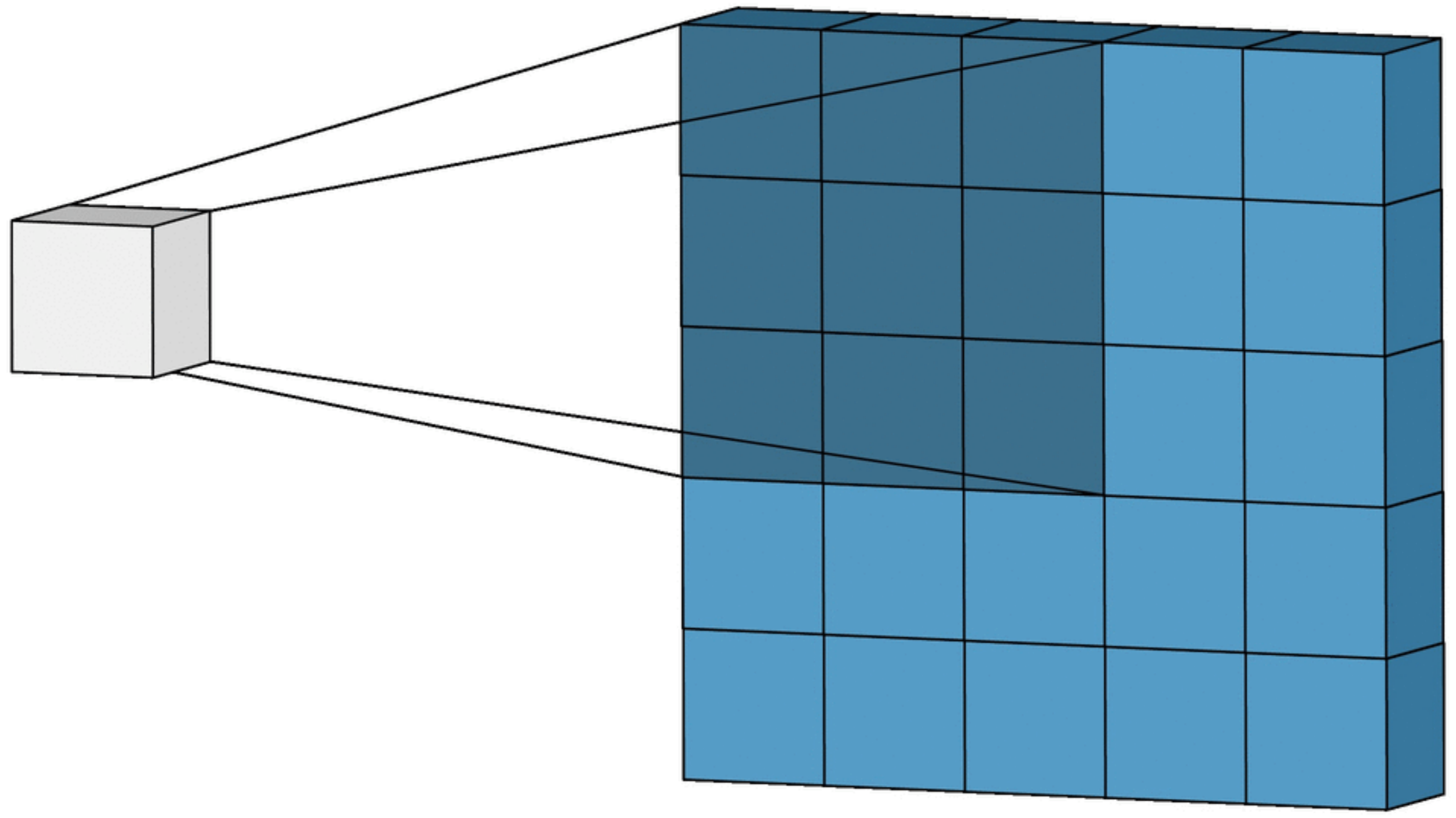
- **Convolutional Layer.** Parameter sharing—reduced compute, and much more reduced memory.

(More likely to be compute-bound than fully-connected layers)



Activation $X_{i-1} \in \mathbb{R}^{d\times d\times c}$

Activation $X_i \in \mathbb{R}^{d\times d\times c}$



Option 1) Fully-connected Layer

Params. Requires c^2d^4 parameters.

Compute. Requires $2c^2d^4$ FLOPs

Option 2) Convolutional layer (3×3)

Params. Requires $9c^2$ parameters

Compute. Requires $18c^2d^2$ FLOPs

Note 1. Scaling-up usually done via #channels.

Note 2. ConvNets usually uses more layers, too.

Note 3. Reduction in memory for parameters, but not for activations!
(i.e., works well for inference, but for training...?)

+ Also a thing in Transformers, but across layers!

GIF source: https://towardsdatascience.com/intuitively-understanding-convolutions-for-deep-learning-1f6f42faee1

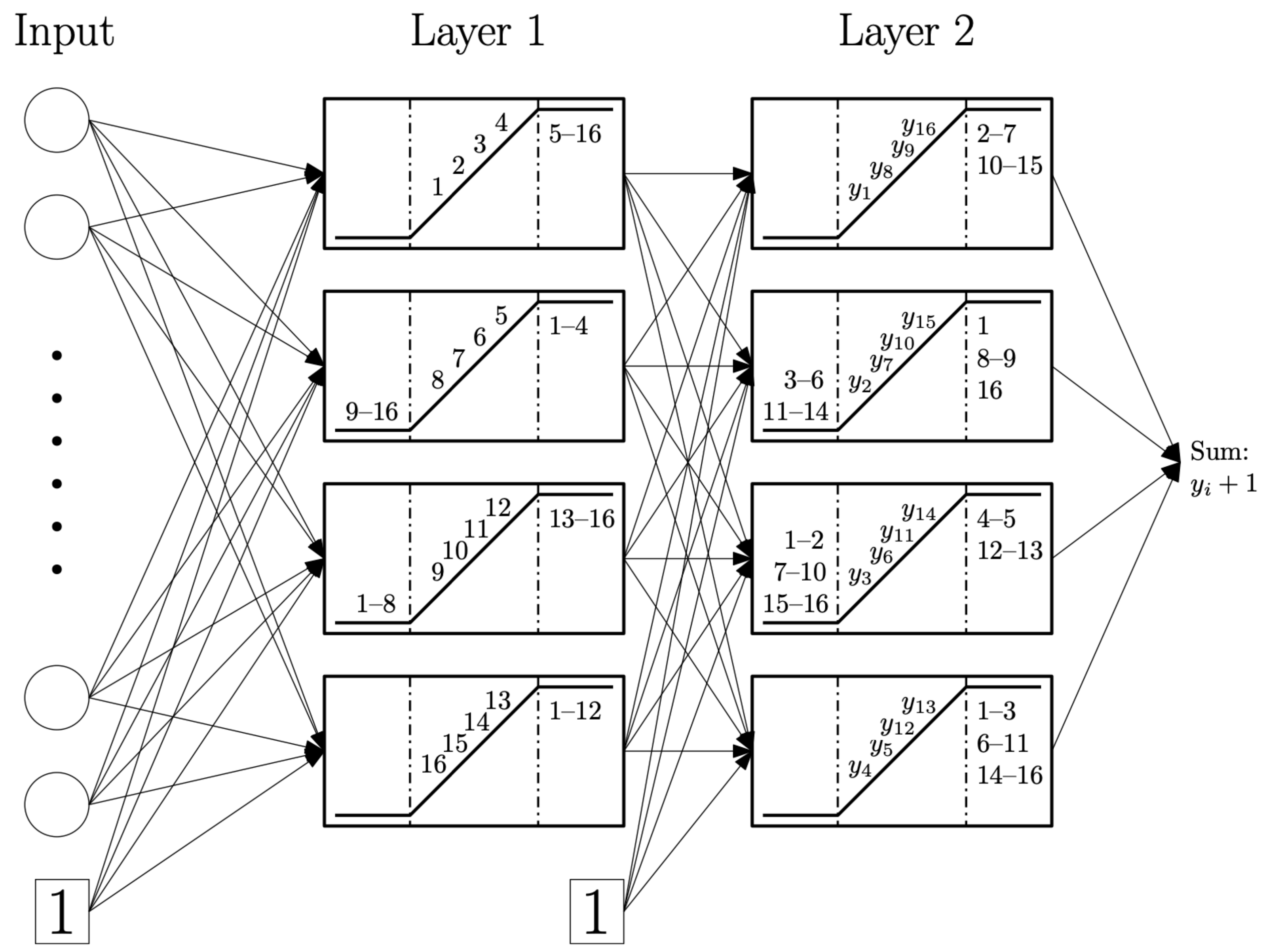
Optimizing DNNs

- **Popular Choice.** We use indirect, iterative methods based on gradient descent (GD)
  Mini-batch GD, Stochastic GD, with Momentum, with Nesterov Momentum,
  AdaGrad, AdaDelta, Adam, AdaMax, AdamP, RAdam, NAdam,
  RMSProp, Shampoo, Lookahead, SAM, …

- **Why Not Direct?** (1) Unique direct solution does not really exist # e.g., ReLU
  (2) Even if it exists, expensive to solve # we've seen $\mathcal{O}(d^3)$ dependency
  (3) Even when easy to solve, the solution is often wiggly and never generalizes…

$$\boldsymbol{W}^1_{j,:} = (-1)^{j-1} \frac{4}{c_{jq} + c_{jq+1} - c_{jq-q} - c_{jq-q+1}} u^T,$$

$$\boldsymbol{b}^1_j = (-1)^j \frac{c_{jq} + c_{jq+1} + c_{jq-q} + c_{jq-q+1}}{c_{jq} + c_{jq+1} - c_{jq-q} - c_{jq-q+1}}.$$



Yun et al., "Small ReLU Networks are powerful memorizers: A tight analysis of memorization capacity," NeurIPS 2019.

Optimizing DNNs

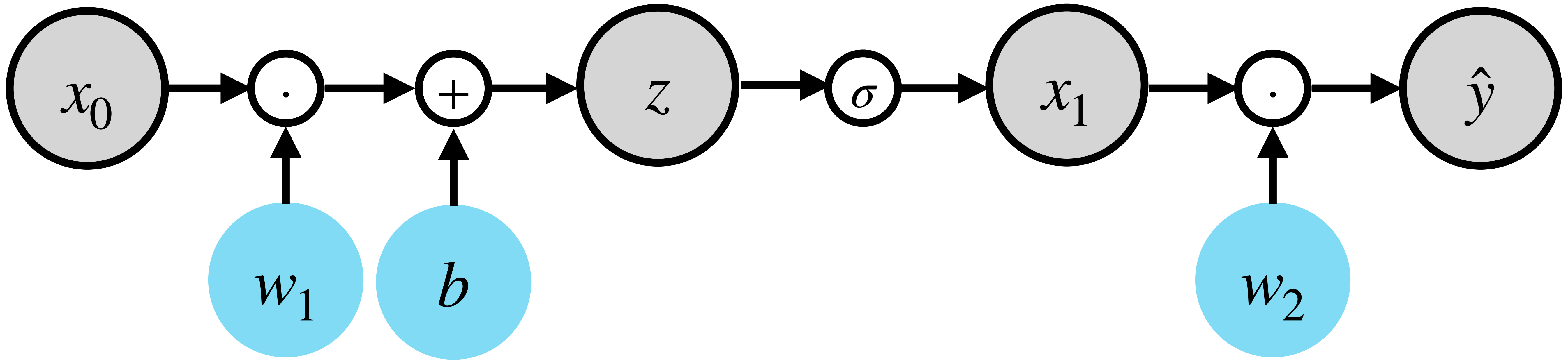
- **Backpropagation.** A "cheap" way to compute gradients (for GD) on multi-layered neural nets.
  Previous approaches required stochastic units & much more compute.
- **Idea.** Computing the gradients can be made less compute-heavy,
  by re-using (1) forward-compute and (2) gradients of succeeding layer activations.
- **Example.** Two-layer ReLU net with 1D parameters

$$z = w_1 \cdot x_0 + b$$

$$x_1 = \sigma(z)$$

$$\hat{y} = w_2 x_1$$

$$L = \frac{1}{2}\left(y - \hat{y}\right)^2$$



Q. Gradients for parameters, w_1, w_2, b?
(evaluated at $w_1^\circ, w_2^\circ, b^\circ$)

$$\left.\frac{\partial L}{\partial w_2}\right|_{w^\circ, b^\circ} =$$

$$\left.\frac{\partial L}{\partial x_1}\right|_{w^\circ, b^\circ} =$$

$$\left.\frac{\partial L}{\partial w_1}\right|_{w^\circ, b^\circ} =$$

$$\left.\frac{\partial L}{\partial b}\right|_{w^\circ, b^\circ} =$$

$$z = w_1 \cdot x_0 + b$$

$$x_1 = \sigma(z)$$

$$\hat{y} = w_2 x_1$$

$$L = \frac{1}{2}\left(y - \hat{y}\right)^2$$

Compare: Forward mode AD

- **Backprop.** An example of reverse mode automatic differentiation.

Backward $\longrightarrow$

$$\hat{y} = f(g(h(x)))$$

$$\frac{\partial \hat{y}}{\partial x} = \frac{\partial \hat{y}}{\partial f} \cdot \frac{\partial f}{\partial g} \cdot \frac{\partial g}{\partial h} \cdot \frac{\partial h}{\partial x}$$

$\longleftarrow$ **Forward**

- **Forward.** $(\text{Mem . Req.}) \quad = \quad \text{Constant}$ # one forward pass for each input, keep one for gradient, one for state
  $\text{Runtime} \quad \propto \quad \#(\text{Input Variables})$

- **Backward.** $(\text{Mem . Req.}) \quad \propto \quad \#(\text{Intermediate Variables})$ # single, monolithic backward pass
  $\text{Runtime} \quad \propto \quad \#(\text{Output Variables})$

- **When to use forward?** Ran out of external memory—very common in meta-learning)
  Want gradient for very small number of variables—e.g., gradient w.r.t. input vector

FLOPs of Backprop

- **Rule of Thumb.** $(\text{Backward FLOPs}) \approx 2 \cdot (\text{Forward FLOPs})$
- **Rough Idea.** Backward FLOPs update both activations and weights, while forward FLOPs are computing activations only.
- **Example.** Three-layer MLP without bias.

$$\hat{y} = W_3\sigma(W_2\sigma(W_1X_0)))$$

$$\hat{y} \in \mathbb{R}^{k\times N} = W_3 \in \mathbb{R}^{k\times h}\,\sigma\Big(W_2 \in \mathbb{R}^{h\times h}\,\sigma\Big(W_1 \in \mathbb{R}^{h\times d}\; X_0 \in \mathbb{R}^{d\times N} \Big)\Big)$$

$$\hat{y} = W_3\sigma(W_2\sigma(W_1X_0))$$

$$\hat{y} \in \mathbb{R}^{k\times N} = W_3 \in \mathbb{R}^{k\times h}\,\sigma\Big(W_2 \in \mathbb{R}^{h\times h}\,\sigma\Big(W_1 \in \mathbb{R}^{h\times d}\; X_0 \in \mathbb{R}^{d\times N}\Big)\Big)$$

Forward FLOPs

Layer 3		Layer 2		Layer 1
$X_3 = W_3X_2$	←	$Z_2 = W_2X_1$	←	$Z_1 = W_1X_0$
$2hkN$		$2h^2N$		$2dhN$

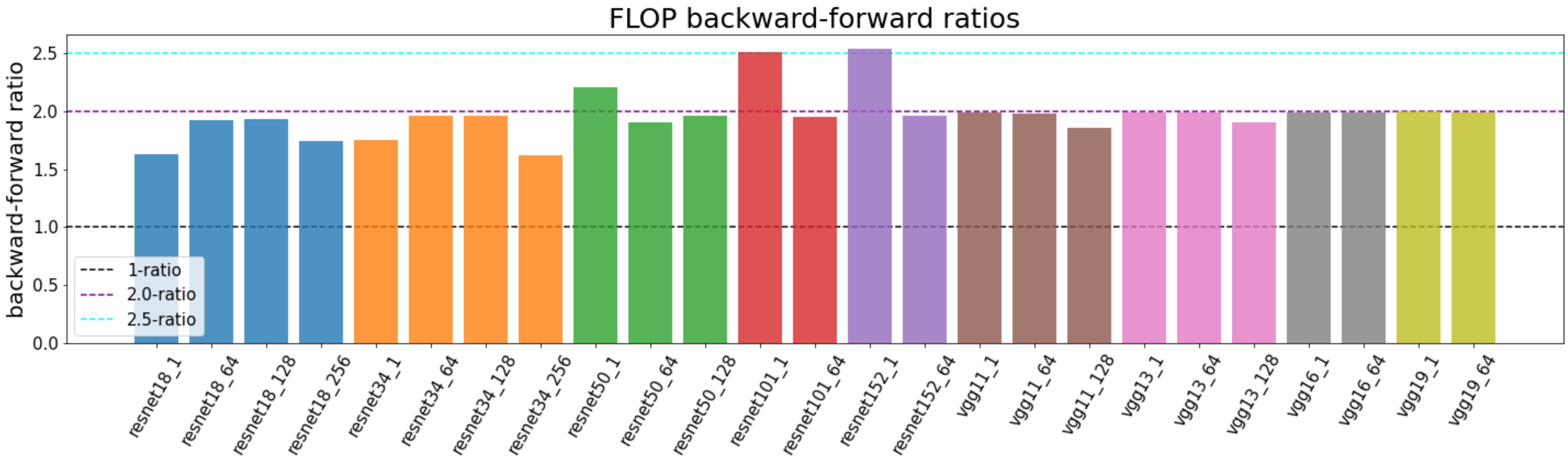
Backward FLOPs

Layer 3		Layer 2		Layer 1
$\frac{\partial L}{\partial W_3} = (\hat{y} - y)X_2^\top$	→	$\frac{\partial L}{\partial W_2} = \frac{\partial L}{\partial X_2}X_1^\top$	→	$\frac{\partial L}{\partial W_1} = \frac{\partial L}{\partial X_1}X_0^\top$
$2hkN$		$2h^2N$		$2dhN$
$\frac{\partial L}{\partial X_2} = W_3^\top(\hat{y} - y)$		$\frac{\partial L}{\partial X_1} = W_2^\top\frac{\partial L}{\partial X_2}$		
$2hkN$		$2h^2N$		

Note. Ignored ReLU!

Note. If memory is large, prepare transposed ones!

Note. Don't forget the FLOPs for the weight update... how big?



Note. Does not mean that optimizing inference == optimizing training!

(Batch issues, steps until convergence, etc)

https://www.lesswrong.com/posts/fnjKpBoWJXcSDwhZk/what-s-the-backward-forward-flop-ratio-for-neural-networks

Memory of Backprop

- Revisit the previous example:

$$\hat{y} \in \mathbb{R}^{k\times N} = W_3 \in \mathbb{R}^{k\times h}\, \sigma\Big(W_2 \in \mathbb{R}^{h\times h}\, \sigma\big(W_1 \in \mathbb{R}^{h\times d}\; X_0 \in \mathbb{R}^{d\times N} \big)\Big)$$

- **Forward pass.** Need to have parameters W_1, W_2, W_3 and data X_0 loaded on memory, and activations $X_1, X_2, \hat{y}$ computed and loaded to memory.

- **Backward pass.** Additionally have parameter updates for $\dfrac{\partial L}{\partial W_1}, \dfrac{\partial L}{\partial W_2}, \dfrac{\partial L}{\partial W_3}$, using intermediates $\dfrac{\partial L}{\partial X_1}, \dfrac{\partial L}{\partial X_2}, \dfrac{\partial L}{\partial \hat{y}}$.

Note. Once you have, $\left(\frac{\partial L}{\partial X_2}, \frac{\partial L}{\partial X_2}\right)$, you no longer need X_2, W_2 on memory.

Memory of Backprop

Forward $Z_1 = W_1 X_0 \rightarrow Z_2 = W_2 X_1 \rightarrow X_3 = W_3 X_2$

Backward

$$\frac{\partial L}{\partial W_3} = (\hat{y} - y) X_2^\top \qquad \frac{\partial L}{\partial X_2} = W_3^\top (\hat{y} - y)$$

$$\rightarrow \quad \frac{\partial L}{\partial W_2} = \frac{\partial L}{\partial X_2} X_1^\top \qquad \frac{\partial L}{\partial X_1} = W_2^\top \frac{\partial L}{\partial X_2}$$

$$\rightarrow \quad \frac{\partial L}{\partial W_1} = \frac{\partial L}{\partial X_1} X_0^\top$$

Brain-teaser. Suppose that all tensors are of uniform size,
and that you have enough cache to store four tensors.
How many tensor movements (in/out) do you need, in order to perform all computations?

What if you have a cache enough to store five tensors?

Side Note: Gradient Checkpointing

- In a sense, backpropagation is a way to use extra memory for less compute
  (for storing activations computed during forward)

Q. Can we use less extra memory,
at the cost of slightly increased compute?

A. Yes—no need to store all activations.

Instead, compute them again with extra
forward operations!

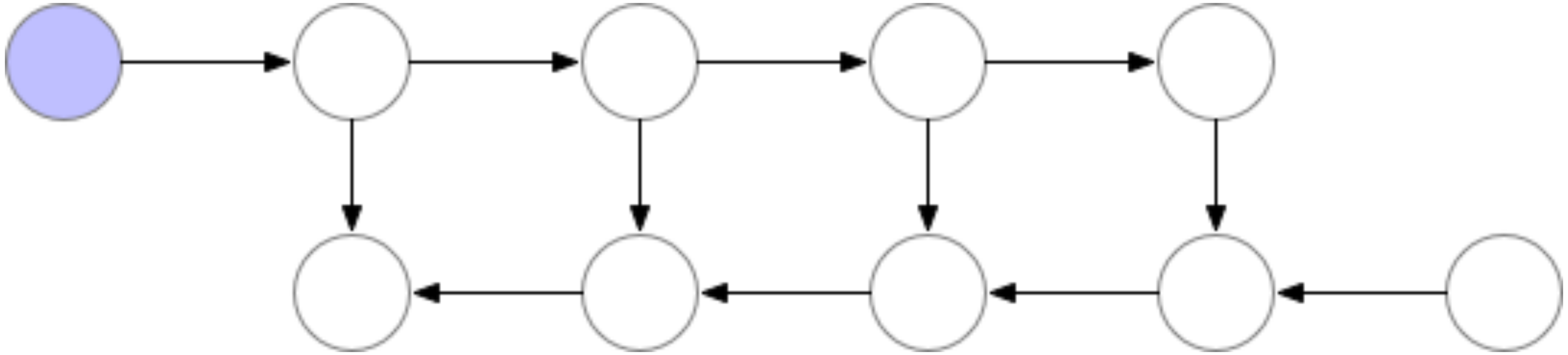
Training Deep Nets with Sublinear Memory Cost

Tianqi Chen [1], **Bing Xu** [2], **Chiyuan Zhang** [3], and **Carlos Guestrin** [1]

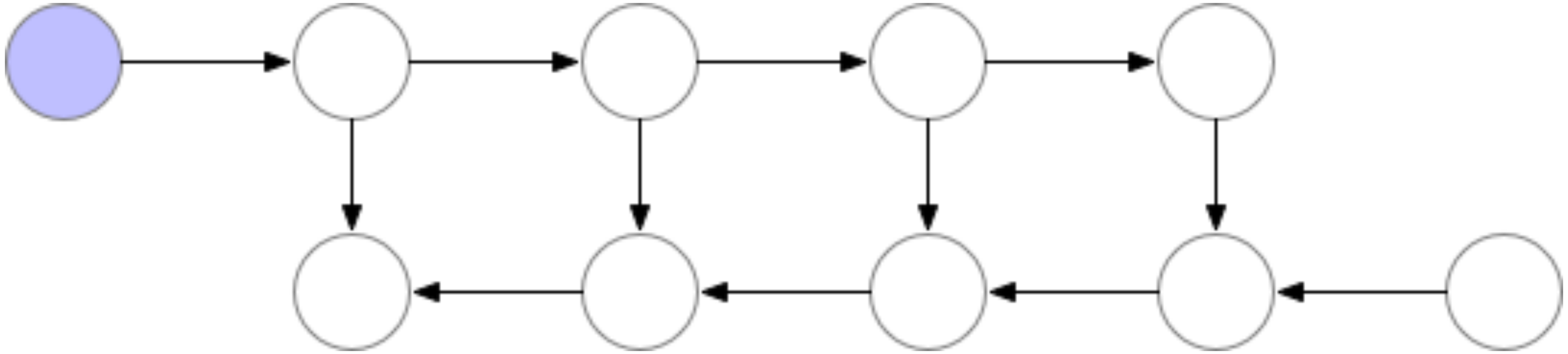
[1] Unveristy of Washington [2] Dato. Inc [3] Massachusetts Institute of Technology

https://arxiv.org/pdf/1604.06174v2.pdf

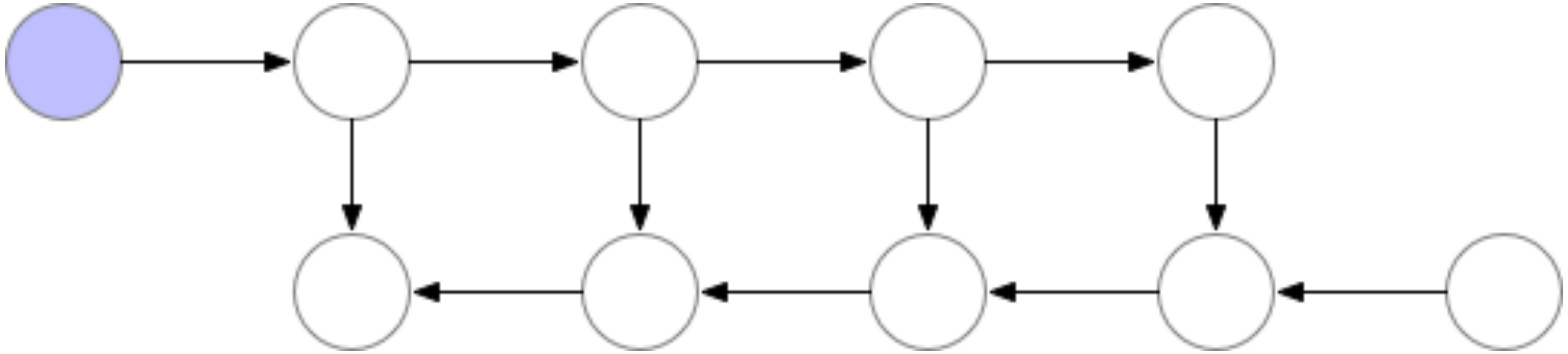
Vanilla



Low-Memory
(w/o checkpoint)



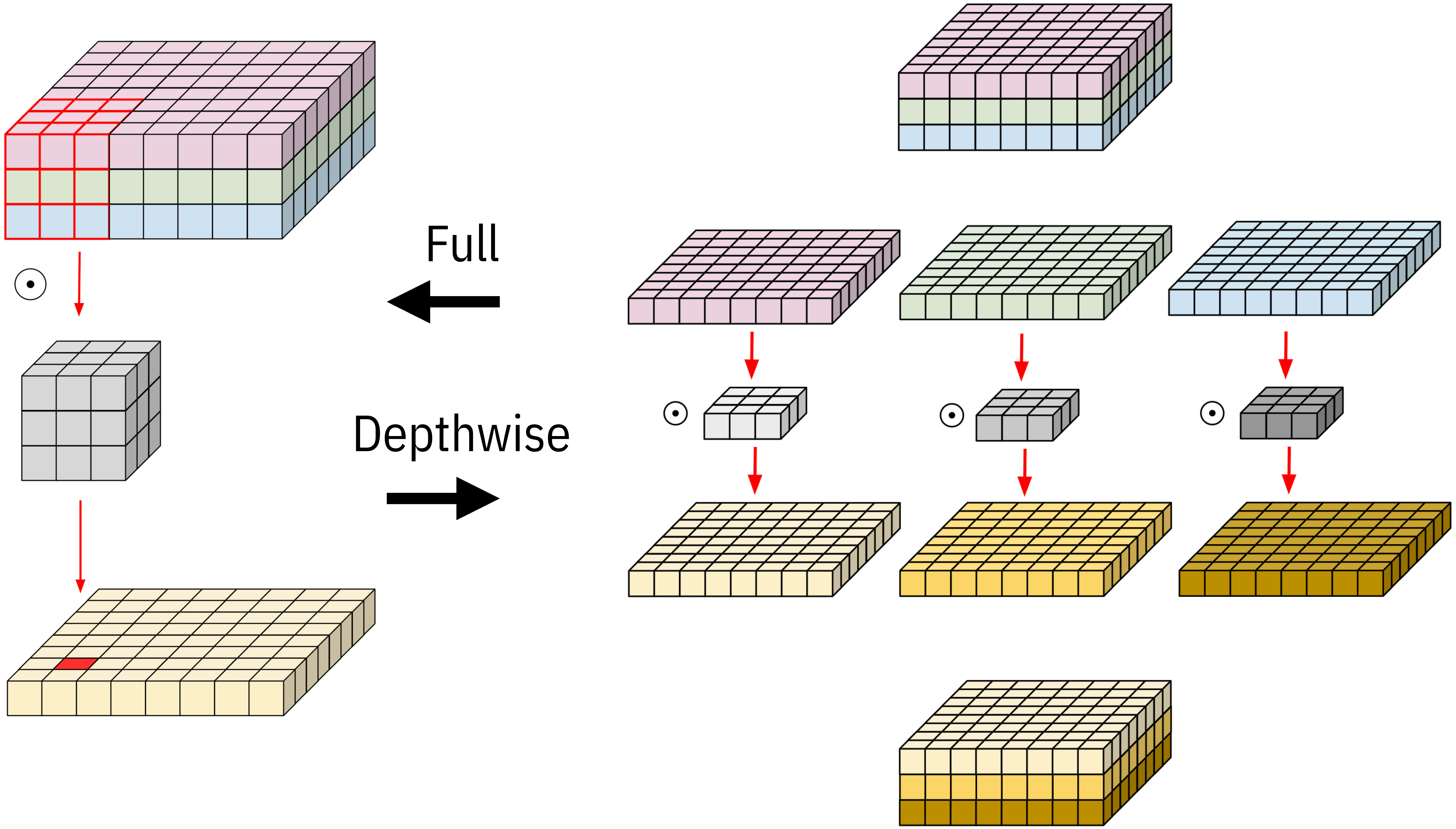
Checkpointing



https://github.com/cybertronai/gradient-checkpointing

Side Note: Depthwise Convolution

- In most modern "efficient" neural network architectures (e.g., MobileNet, EfficientNet), people use **depthwise convolutions** instead of full convolutions.



Params

Full. c^2k^2

DW. ck^2

FLOPs

Full. $2c^2k^2d^2$

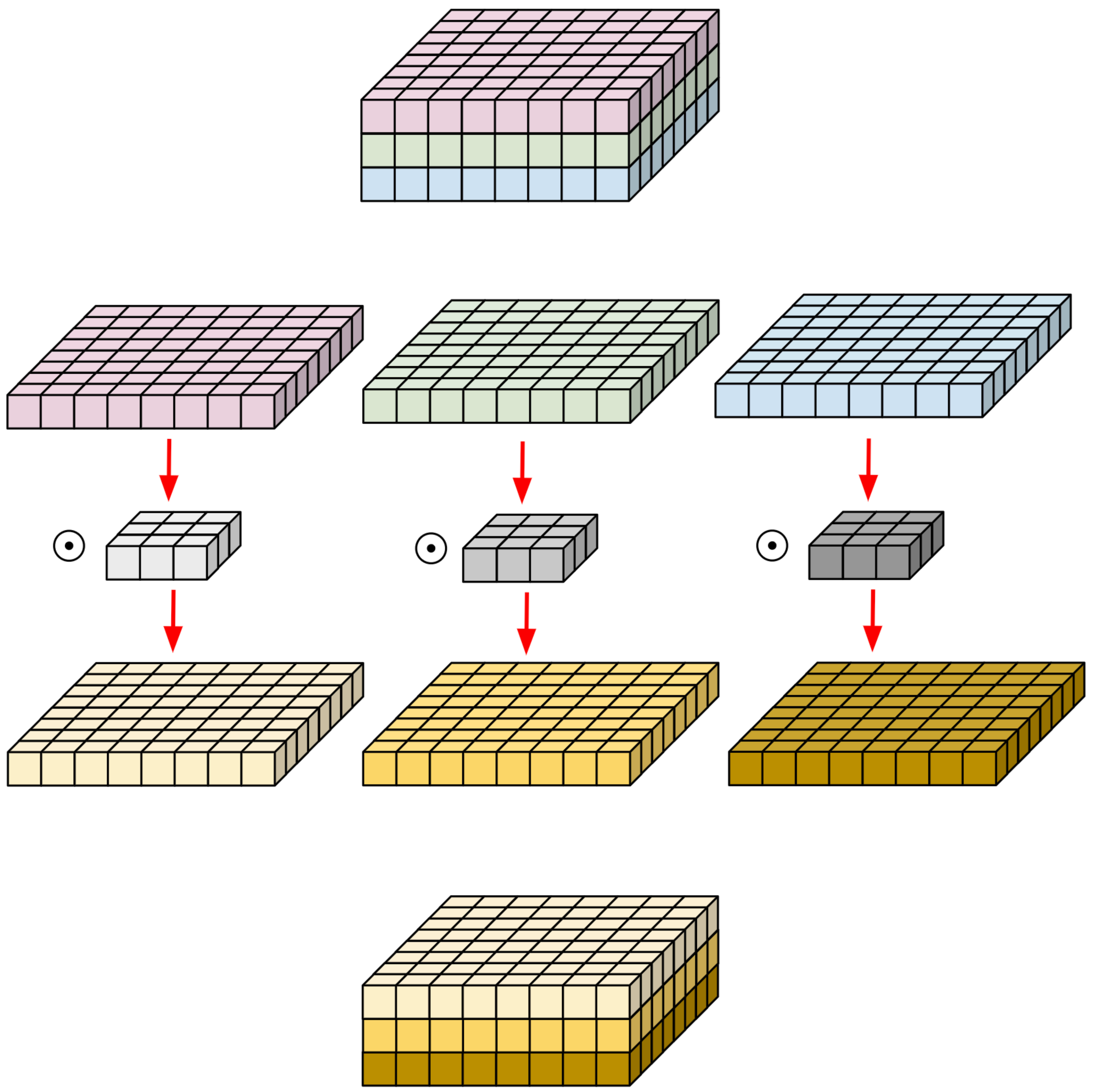
DW. $2ck^2d^2$

Image source: https://eli.thegreenplace.net/2018/depthwise-separable-convolutions-for-machine-learning/

Side Note: Depthwise Convolution

- **Problem.** Channel information cannot be mixed!

  ⇒ Use one-by-one convolution to mix the layers # e.g., Inverted Bottleneck



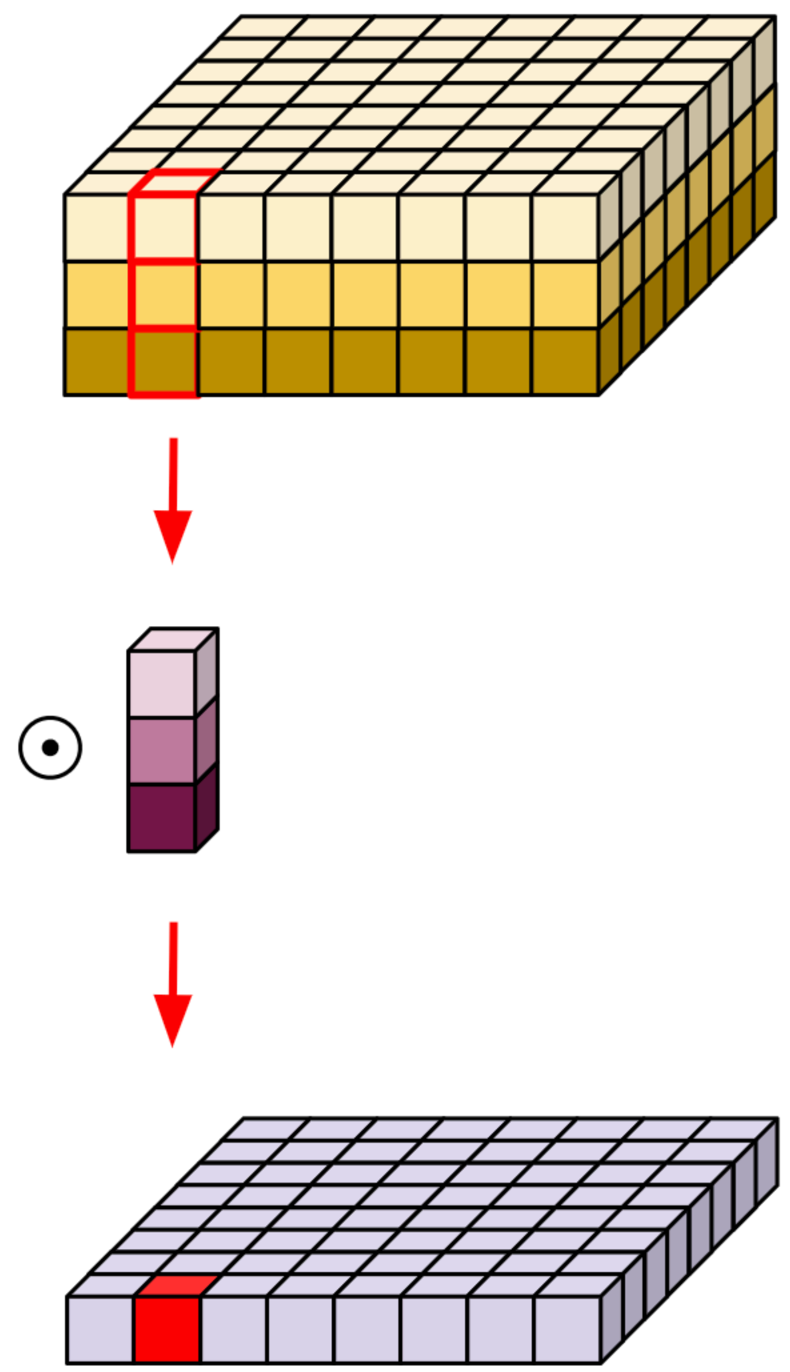


Image source: https://eli.thegreenplace.net/2018/depthwise-separable-convolutions-for-machine-learning/

Side Note: Optimization algorithm!

- Some optimizers have extra "state variables" for various reasons.
  - For instance, we sometimes utilize "momentum" terms.
    Beware of the memory overhead!!

Algorithm 1: *Adam*, our proposed algorithm for stochastic optimization. See section 2 for details, and for a slightly more efficient (but less clear) order of computation. g_t^2 indicates the elementwise square $g_t \odot g_t$. Good default settings for the tested machine learning problems are $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$. All operations on vectors are element-wise. With β_1^t and β_2^t we denote β_1 and β_2 to the power t.

Require: α: Stepsize
Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates
Require: $f(\theta)$: Stochastic objective function with parameters θ
Require: θ_0: Initial parameter vector
  $m_0 \leftarrow 0$ (Initialize 1st moment vector)
  $v_0 \leftarrow 0$ (Initialize 2nd moment vector)
  $t \leftarrow 0$ (Initialize timestep)
  while θ_t not converged **do**
    $t \leftarrow t + 1$
    $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep t)
    $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
    $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
    $\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
    $\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
    $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$ (Update parameters)
  end while
  return θ_t (Resulting parameters)

Kingma and Ba, "Adam: A method for stochastic optimization," ICLR 2015.

Summary

- Deep networks, as a stack of linear models.
- DNN inference: Compute vs. Memory
  - Parameter sharing — Less bandwidth
  - Overlapping — Compute utilization
- DNN training
  - Backprop — Using extra memory for less compute
    (smooth tradeoff via checkpointing)
  - Forward FLOPs vs. Backward FLOPs
- **Next.** A bit more about efficiency criteria
  (compute / memory / peak memory / #params)
  Some case studies…