Introduction to Machine Learning 8

Deep Learning ii

- Tong Zhang UCAS 2025 fall

Recap

1. What are the neural networks

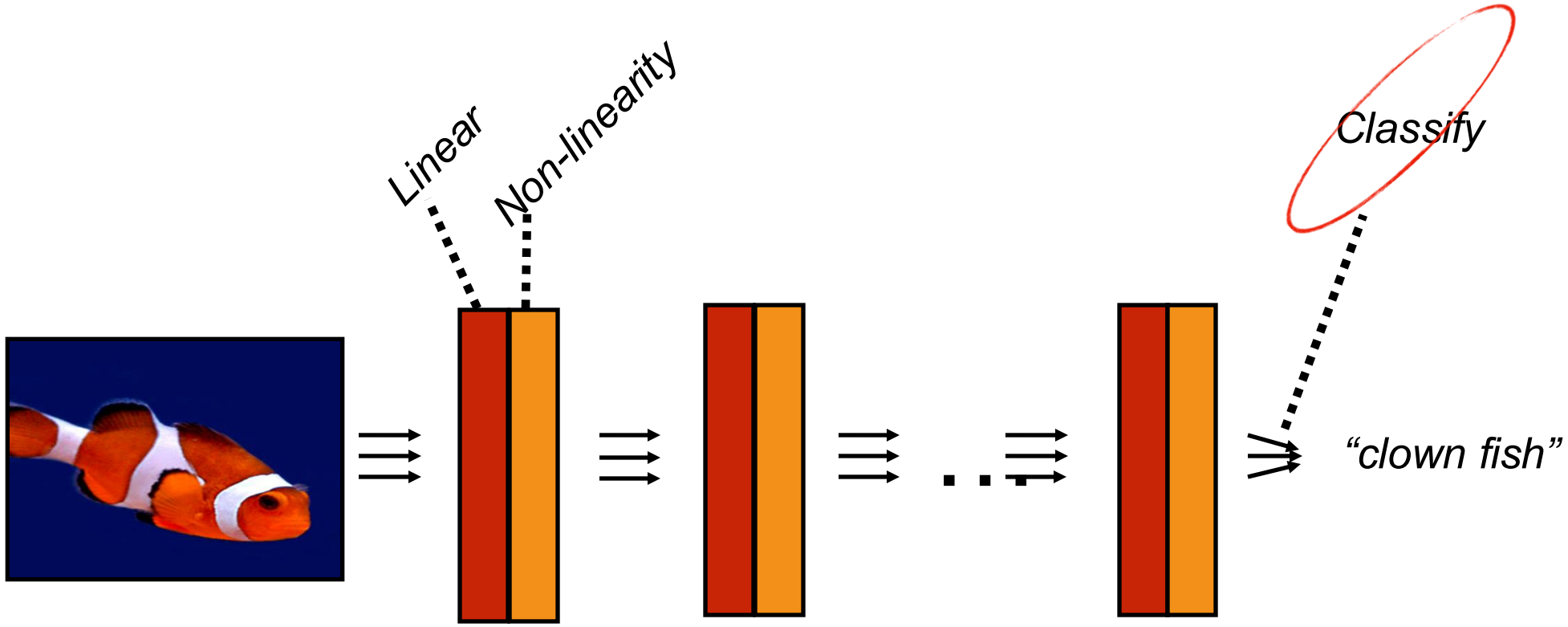
2. Key technique of neural networks:
    - Activation function
    - Optimizer
    - Loss function

Today's plan

1. What are the neural networks

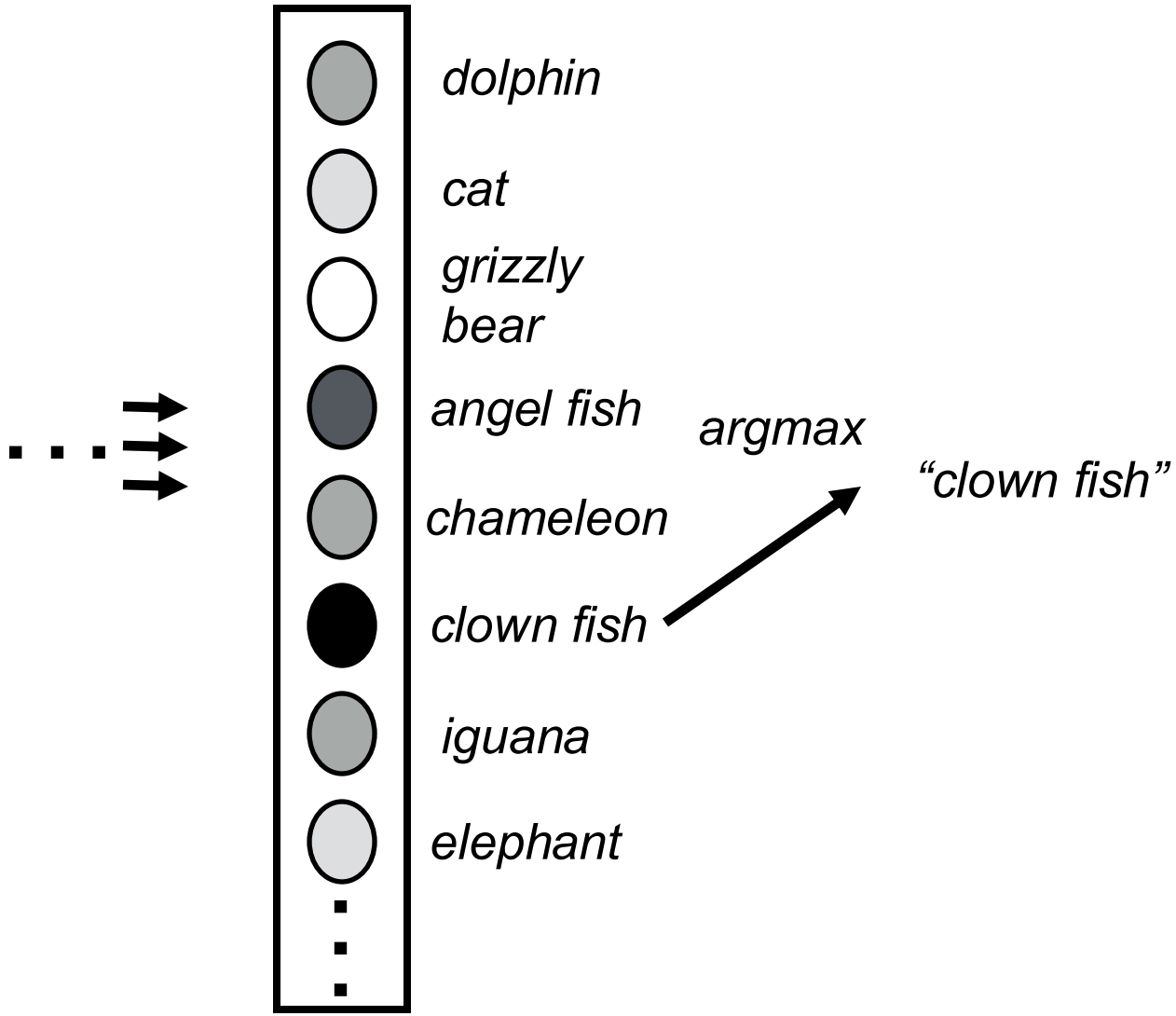
2. Key technique of neural networks:
   - Activation function
   - Optimizer
   - Loss function
   - Backpropagation
   - Regularizer
     - Normalization, dropout, weight decay, early stopping ……
   - Convolutional Neural Networks

Deep Nets

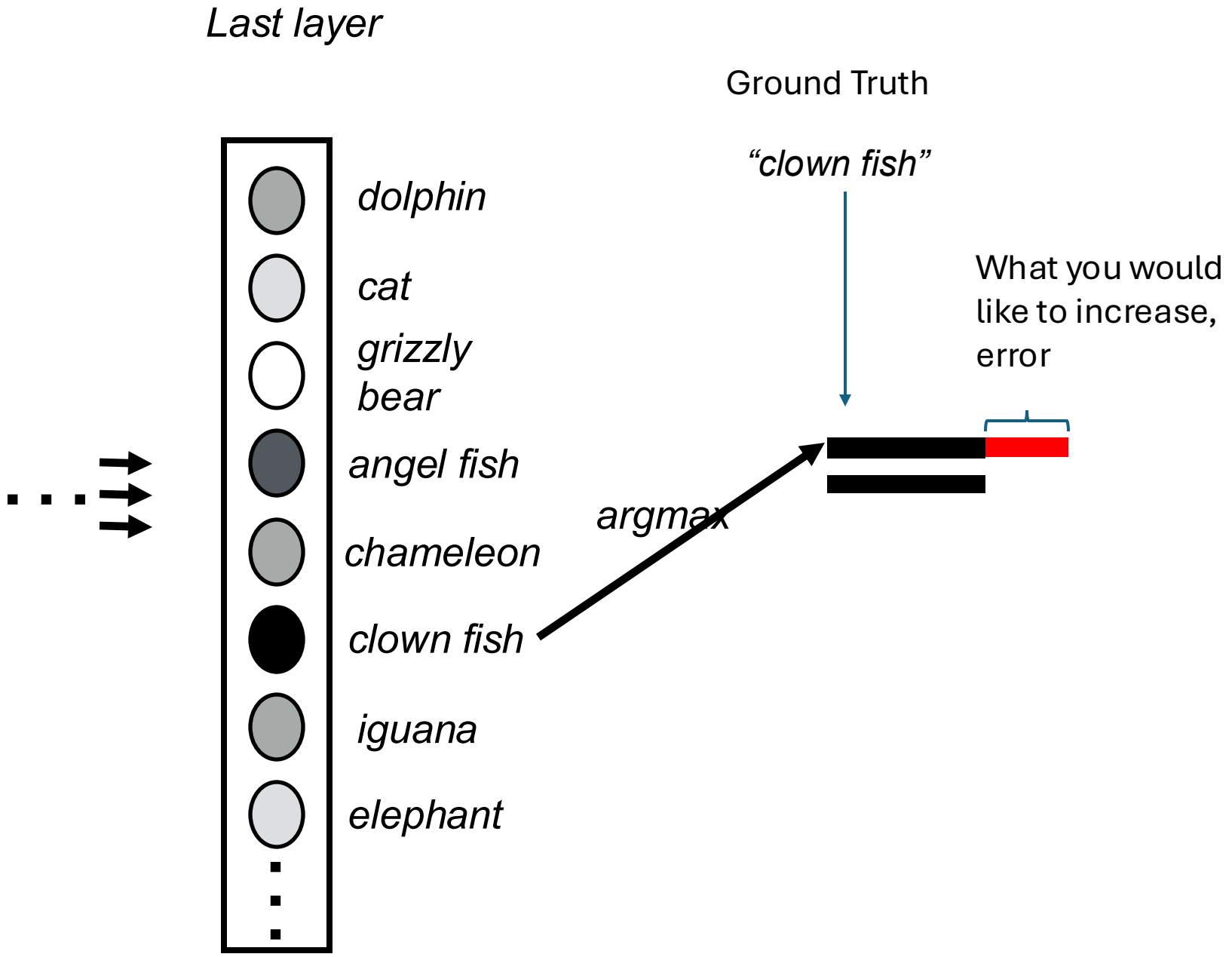


$$f(\mathbf{x}) = f_L(f_{L-1}(\dots f_2(f_1(\mathbf{x}))))$$

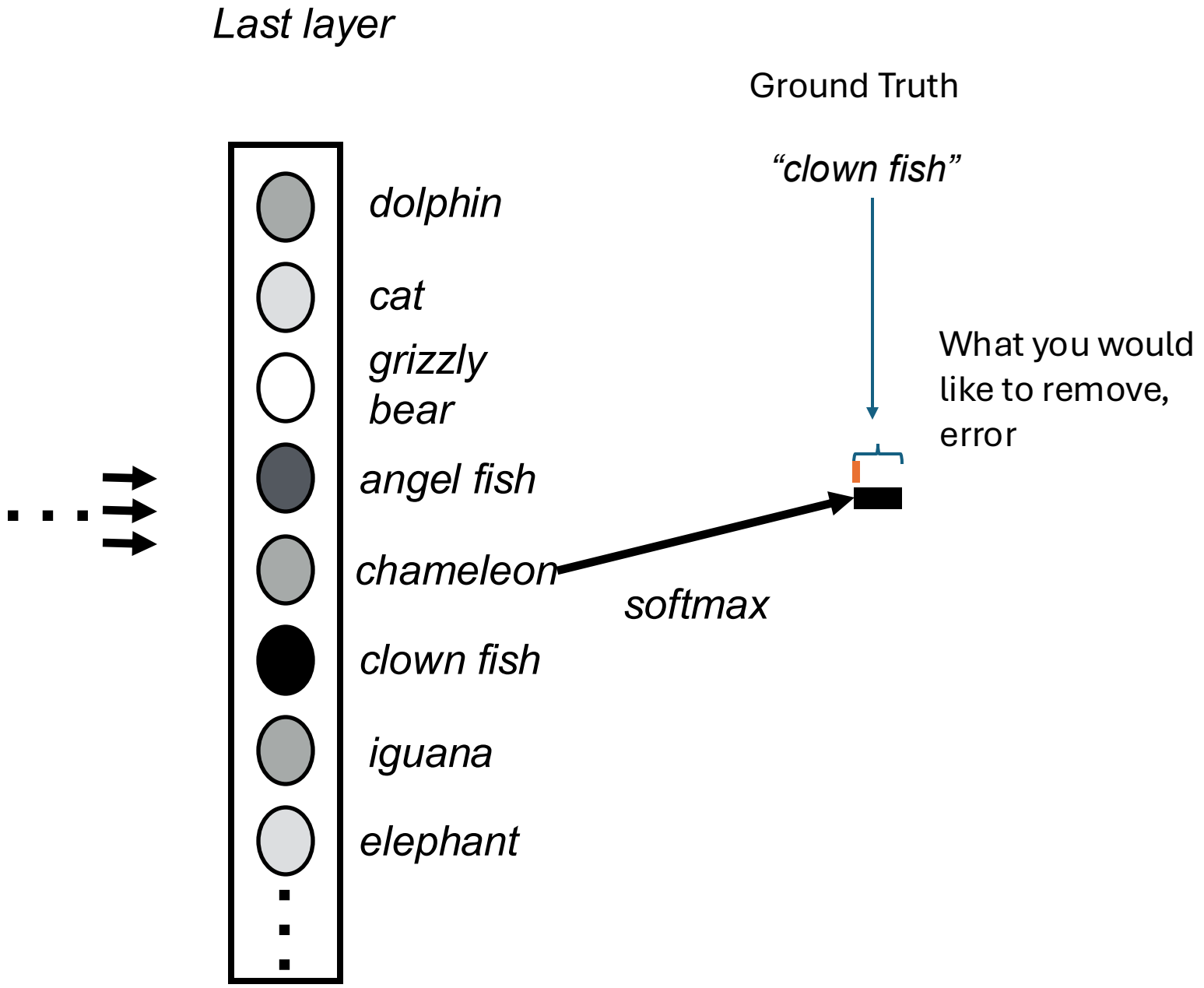
Deep Nets


Last layer
dolphin
cat
grizzly bear
angel fish
chameleon
clown fish
iguana
elephant
argmax
“clown fish”

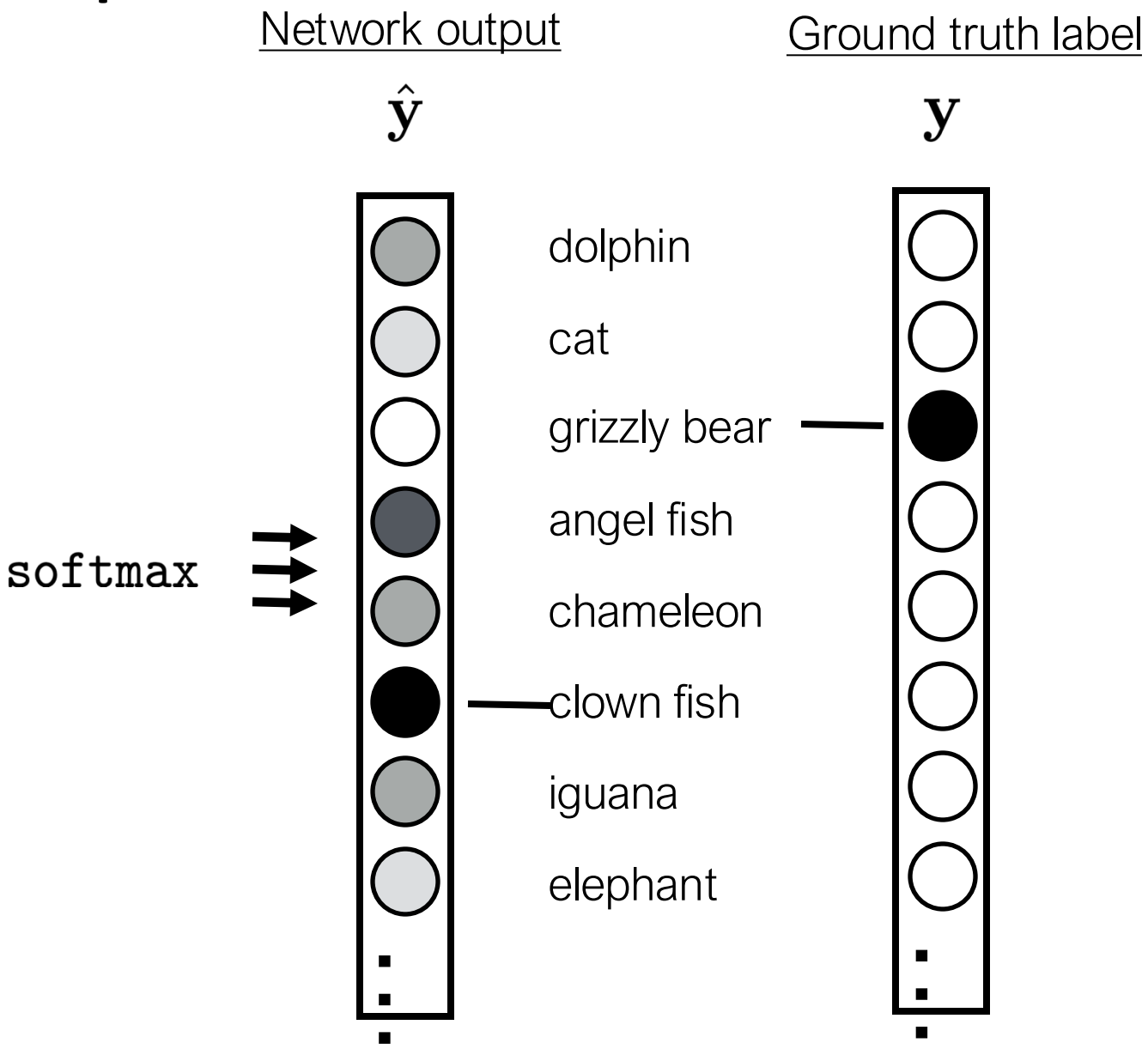
Deep Nets


Last layer
Ground Truth
"clown fish"
What you would like to increase, error
dolphin
cat
grizzly bear
angel fish
chameleon
clown fish
iguana
elephant
argmax

Deep Nets


Last layer
Ground Truth
"clown fish"
dolphin
cat
grizzly bear
angel fish
chameleon
clown fish
iguana
elephant
softmax
What you would like to remove, error

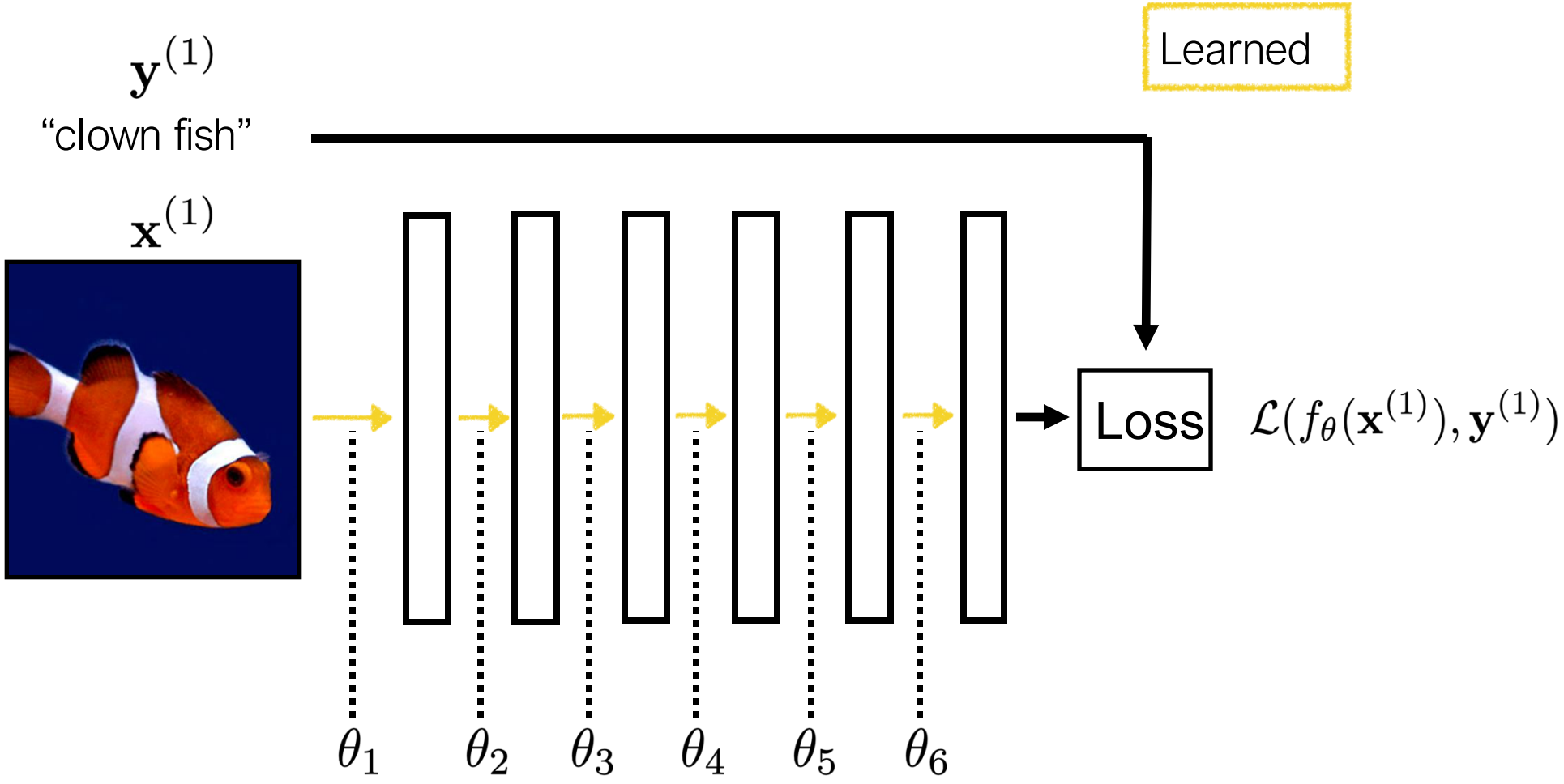
Output



Probability of the observed data under the model

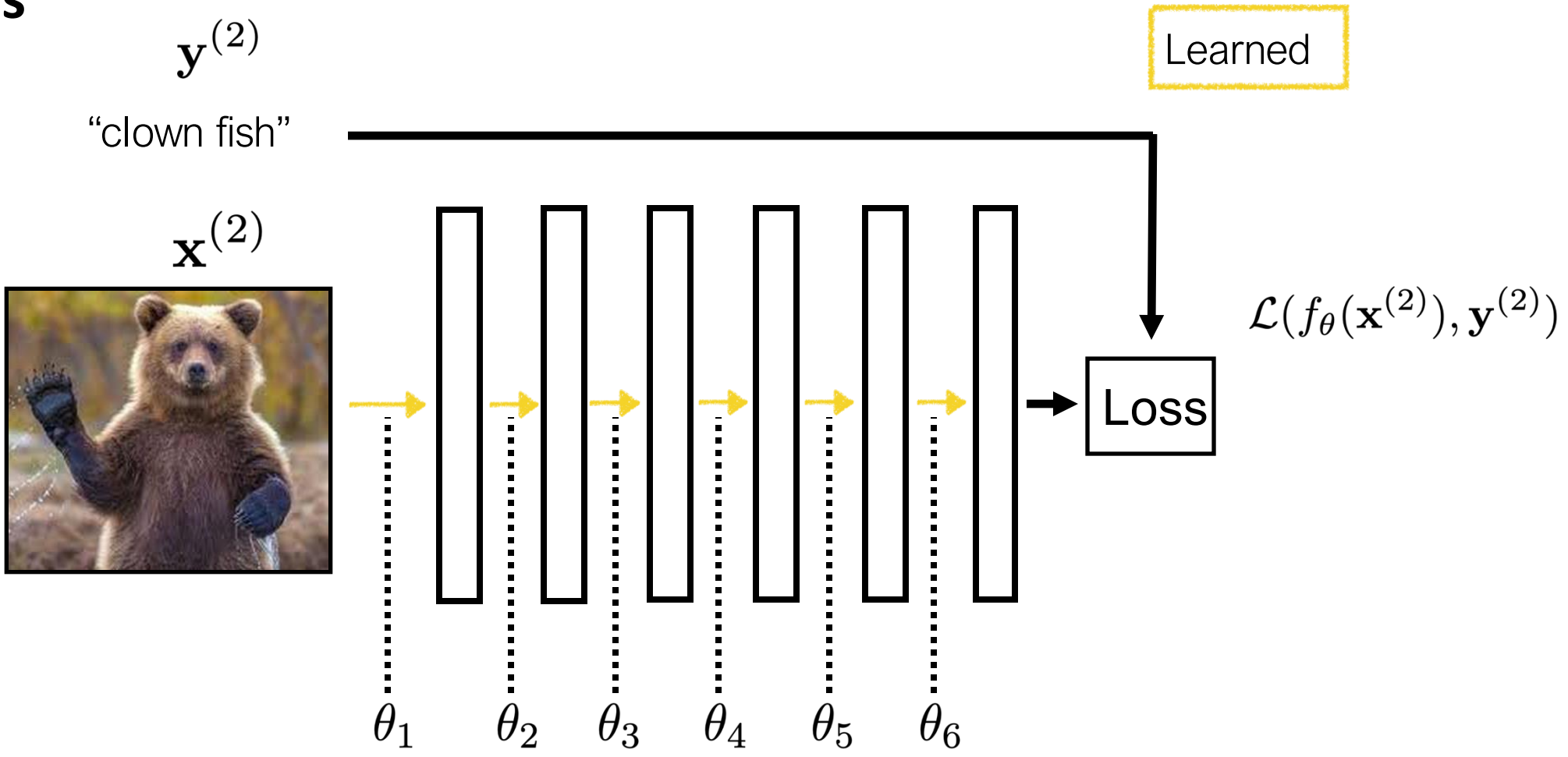
$$H(\mathbf{y}, \hat{\mathbf{y}}) = -\sum_{k=1}^{K} y_k \log \hat{y}_k$$

Deep Nets



$$\theta^* = \arg\min_\theta \sum_{i=1}^{N} \mathcal{L}(f_\theta(\mathbf{x}^{(i)}), \mathbf{y}^{(i)})$$

Deep Nets



$$\theta^* = \arg\min_\theta \sum_{i=1}^{N} \mathcal{L}(f_\theta(\mathbf{x}^{(i)}), \mathbf{y}^{(i)})$$

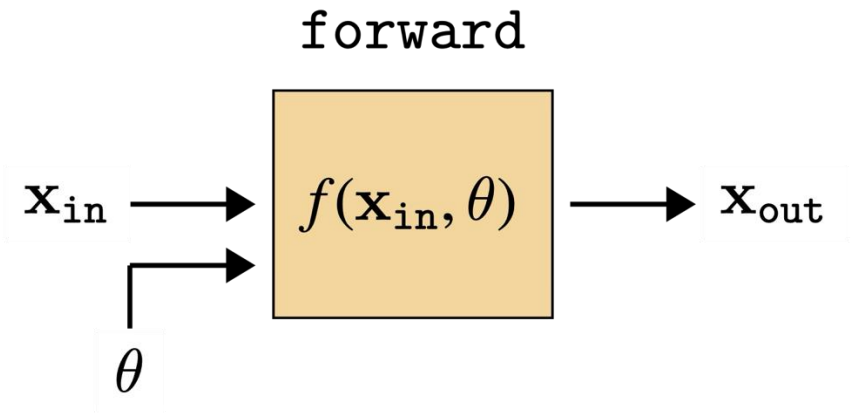
How to pass the gradients layer by layer

Gradient Descent

Backpropagation

Backpropagation

Network parameters $\theta = \{w_1, w_2, \cdots, b_1, b_2, \cdots\}$



$$\mathbf{x}_{out} = f(\mathbf{x}_{in}, \theta)$$

$$\nabla L(\theta) = \begin{bmatrix} \partial L(\theta)/\partial w_1 \\ \partial L(\theta)/\partial w_2 \\ \vdots \\ \partial L(\theta)/\partial b_1 \\ \partial L(\theta)/\partial b_2 \\ \vdots \end{bmatrix}$$

$$\theta^0 \longrightarrow \theta^1 \longrightarrow \theta^2 \longrightarrow \cdots\cdots$$

Compute $\nabla L(\theta^0)$ $\quad \theta^1 = \theta^0 - \eta \nabla L(\theta^0)$

Compute $\nabla L(\theta^1)$ $\quad \theta^2 = \theta^1 - \eta \nabla L(\theta^1)$

Millions of parameters ……

To compute the gradients efficiently, we use ***backpropagation***.

Chain Rule

Case 1

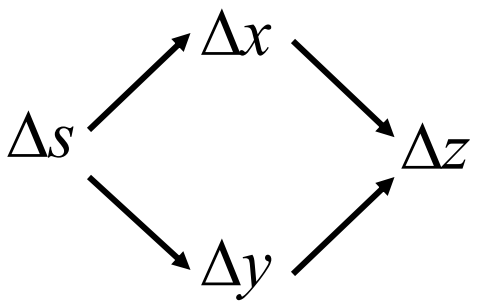
$$y = g(x) \qquad z = h(y)$$

$$\Delta x \to \Delta y \to \Delta z$$

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}$$

Case 2

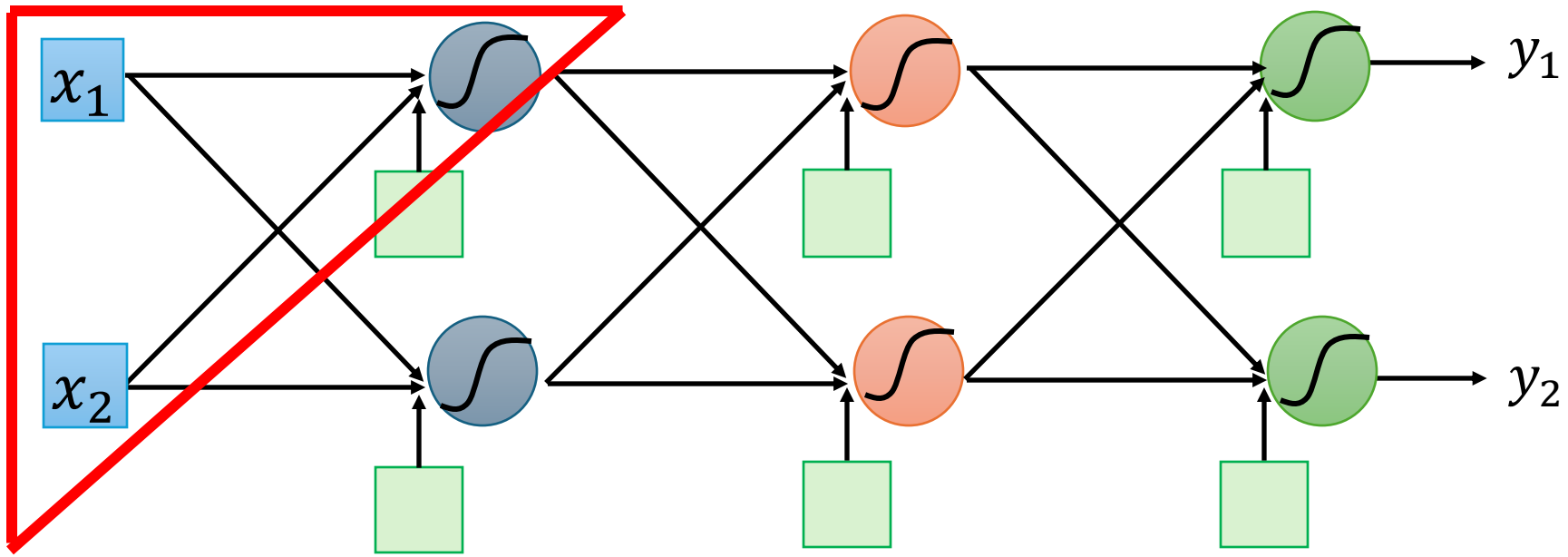
$$x = g(s) \qquad y = h(s) \qquad z = k(x, y)$$



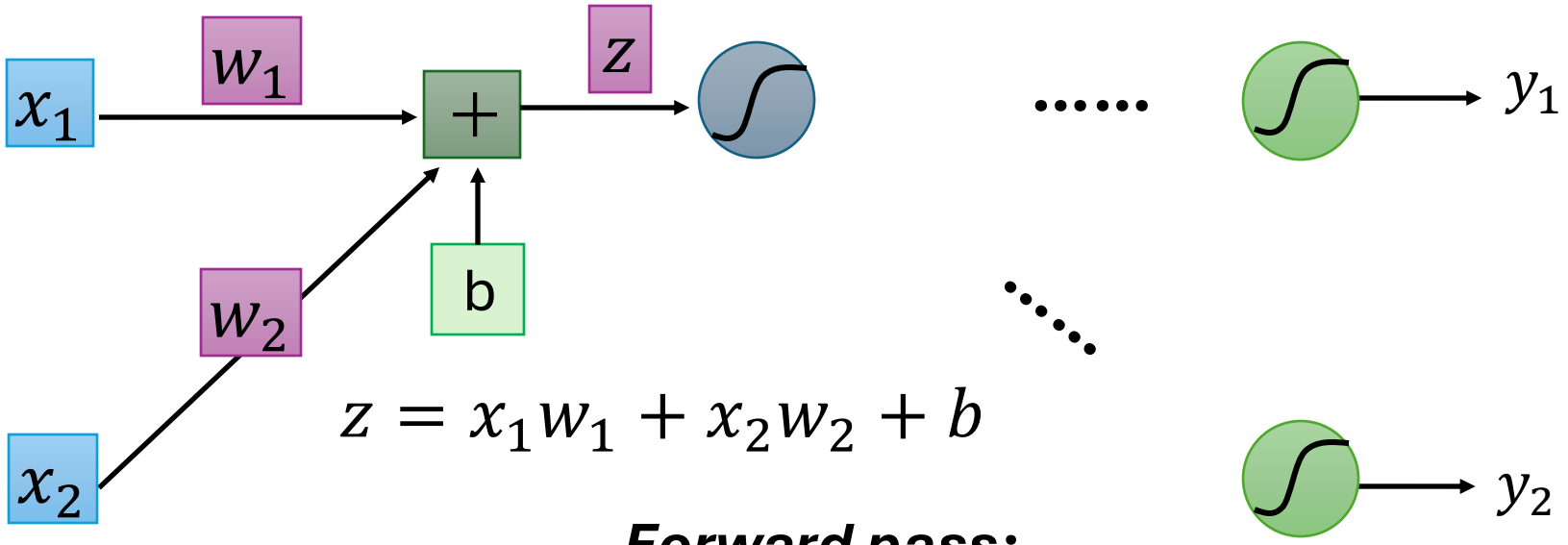
$$\frac{dz}{ds} = \frac{\partial z}{\partial x}\frac{dx}{ds} + \frac{\partial z}{\partial y}\frac{dy}{ds}$$

Backpropagation

$$L(\theta) = \sum_{n=1}^{N} l^n(\theta) \quad \Rightarrow \quad \frac{\partial L(\theta)}{\partial w} = \sum_{n=1}^{N} \frac{\partial l^n(\theta)}{\partial w}$$



Backpropagation



$$z = x_1 w_1 + x_2 w_2 + b$$

Forward pass:

Compute $\partial z / \partial w$ for all parameters

$$\frac{\partial l}{\partial w} = ? \quad \frac{\partial z}{\partial w} \frac{\partial l}{\partial z}$$

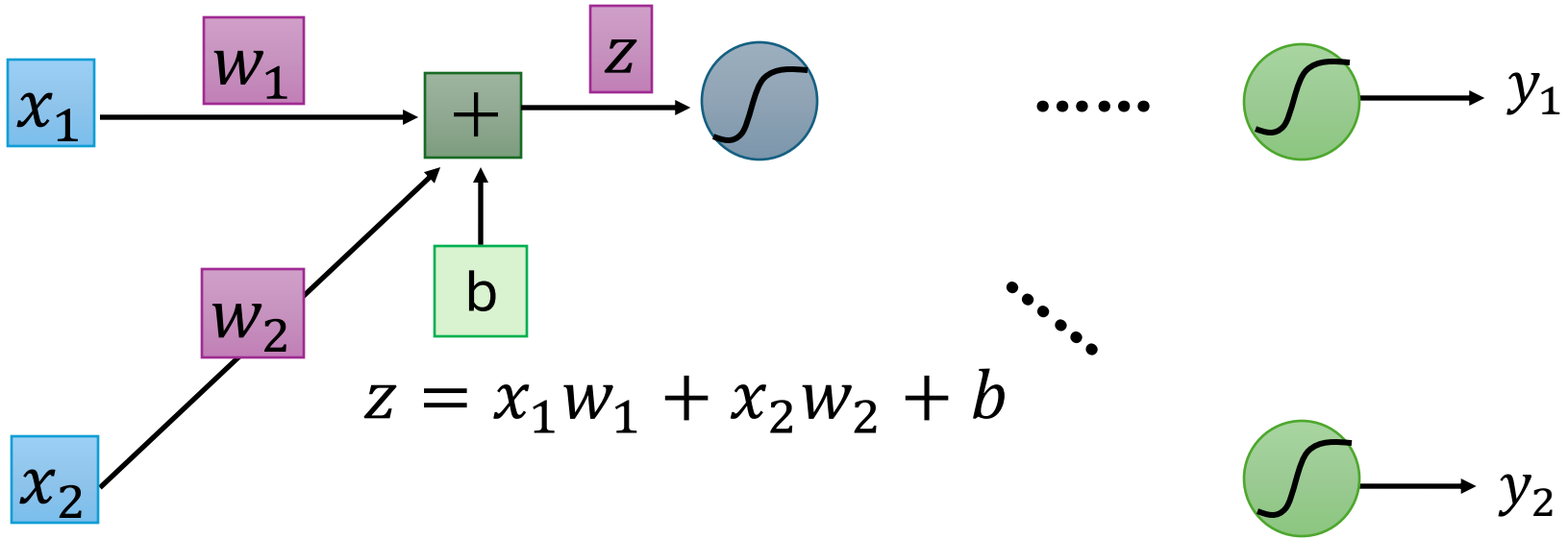
(Chain rule)

Backward pass:

Compute $\partial l / \partial z$ for all activation function inputs z

Backpropagation

Compute $\partial z/\partial w$ for all parameters



$$z = x_1 w_1 + x_2 w_2 + b$$

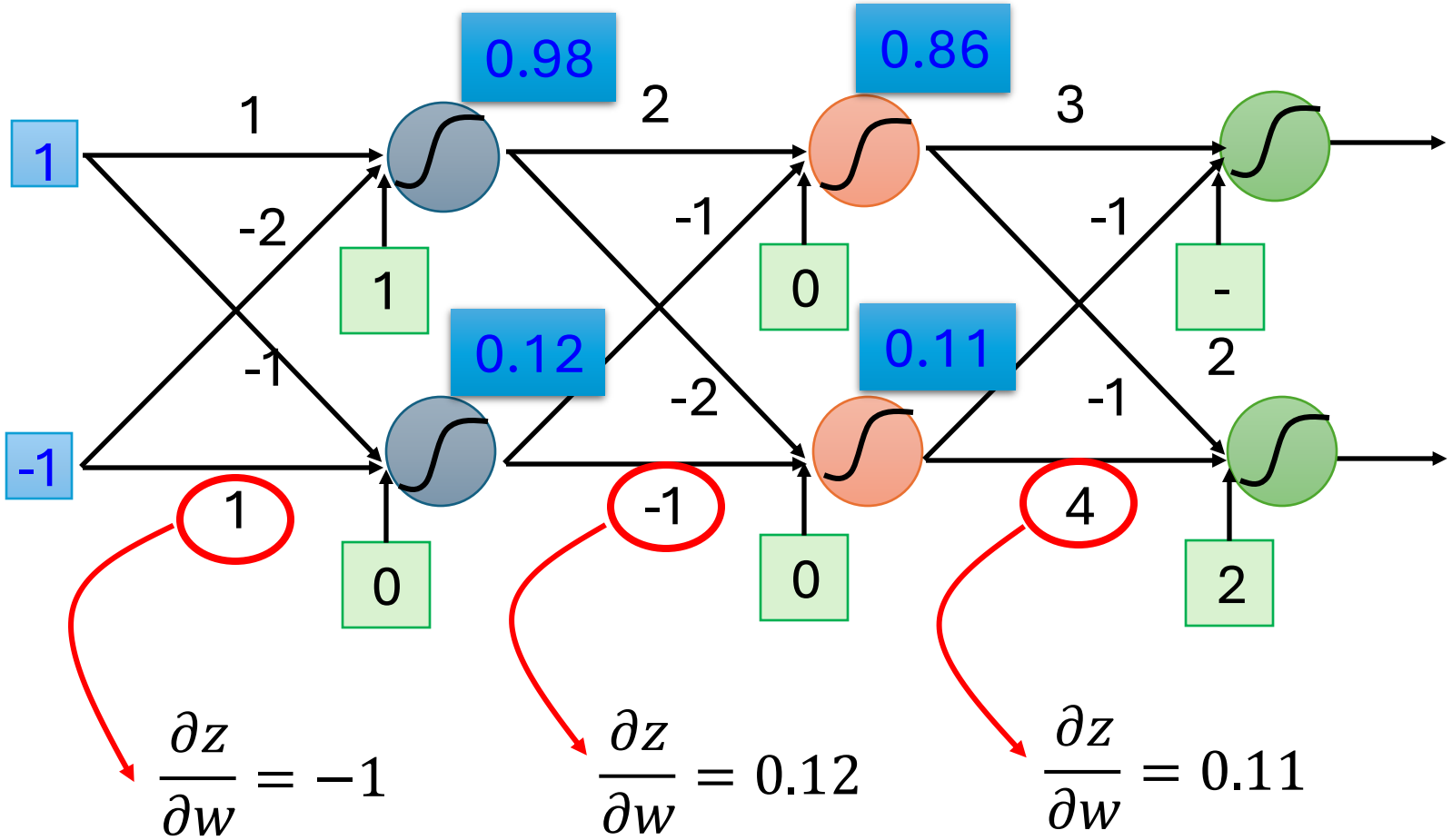
$\partial z/\partial w_1 =?\ x_1$

$\partial z/\partial w_2 =?\ x_2$

The value of the input connected by the weight

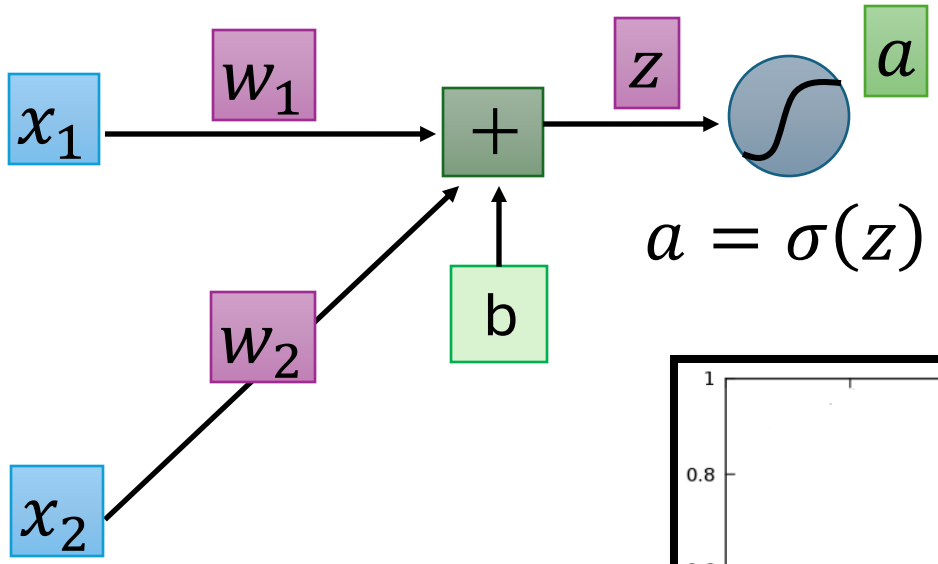
Backpropagation

Compute $\partial z/\partial w$ for all parameters



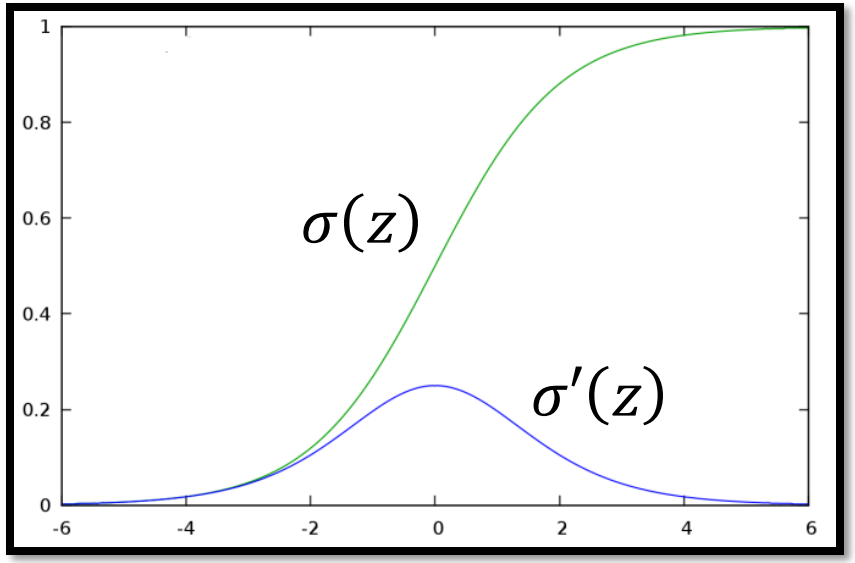
Backpropagation

Compute $\partial l/\partial z$ for all activation function inputs z



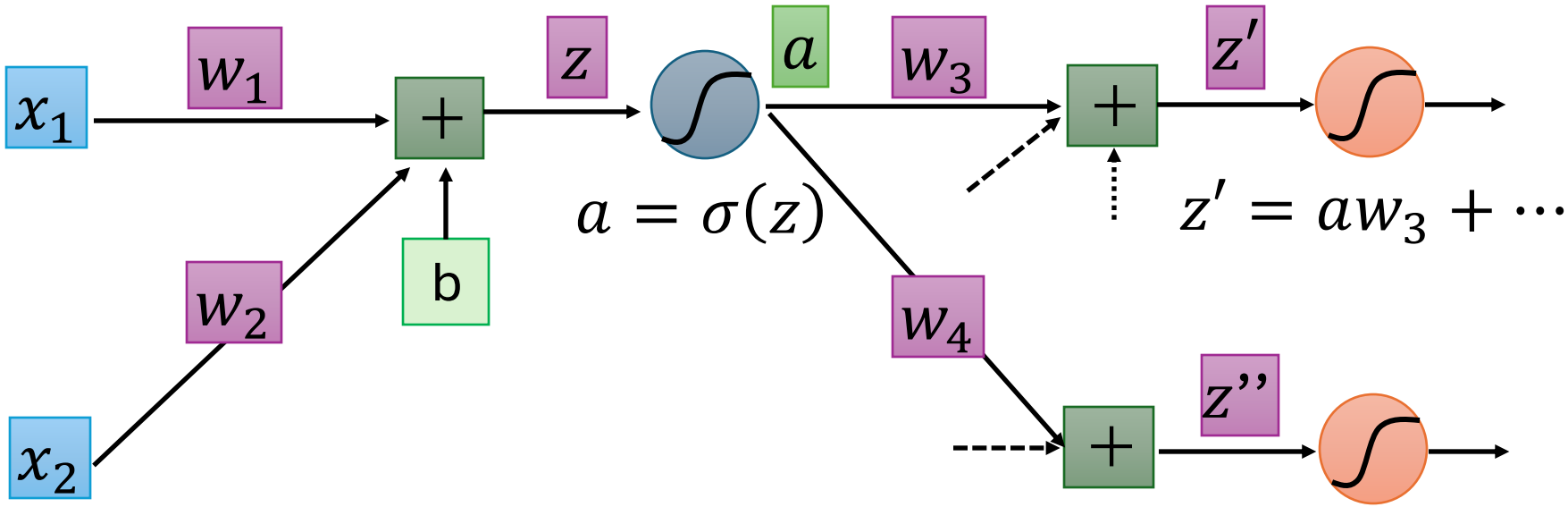
$$\frac{\partial l}{\partial z} = \frac{\partial a}{\partial z}\frac{\partial l}{\partial a}$$

$\Rightarrow \sigma'(z)$



Backpropagation

Compute $\partial l/\partial z$ for all activation function inputs z



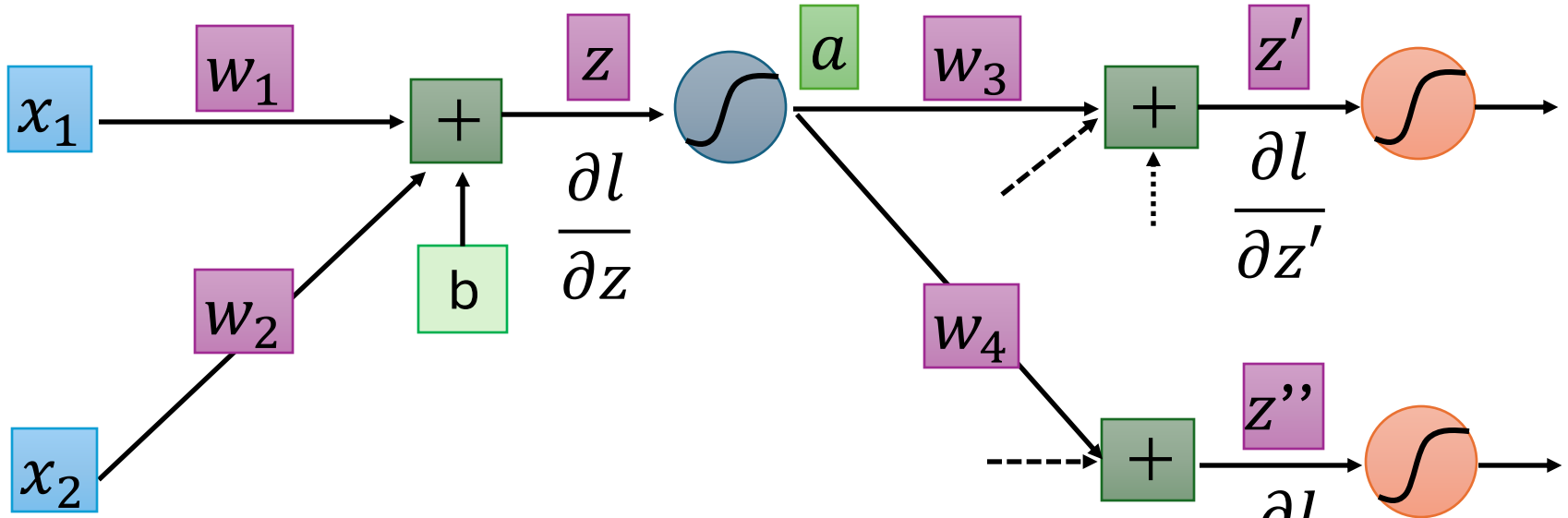
$$\frac{\partial l}{\partial z} = \frac{\partial a}{\partial z}\frac{\partial l}{\partial a} \qquad \frac{\partial l}{\partial a} = \frac{\partial z'}{\partial a}\frac{\partial l}{\partial z'} + \frac{\partial z''}{\partial a}\frac{\partial l}{\partial z''}$$ (Chain rule)

$\frac{\partial z'}{\partial a} = w_3$, $\frac{\partial l}{\partial z'}$ = ?, $\frac{\partial z''}{\partial a} = w_4$, $\frac{\partial l}{\partial z''}$ = ?

Assumed it's known

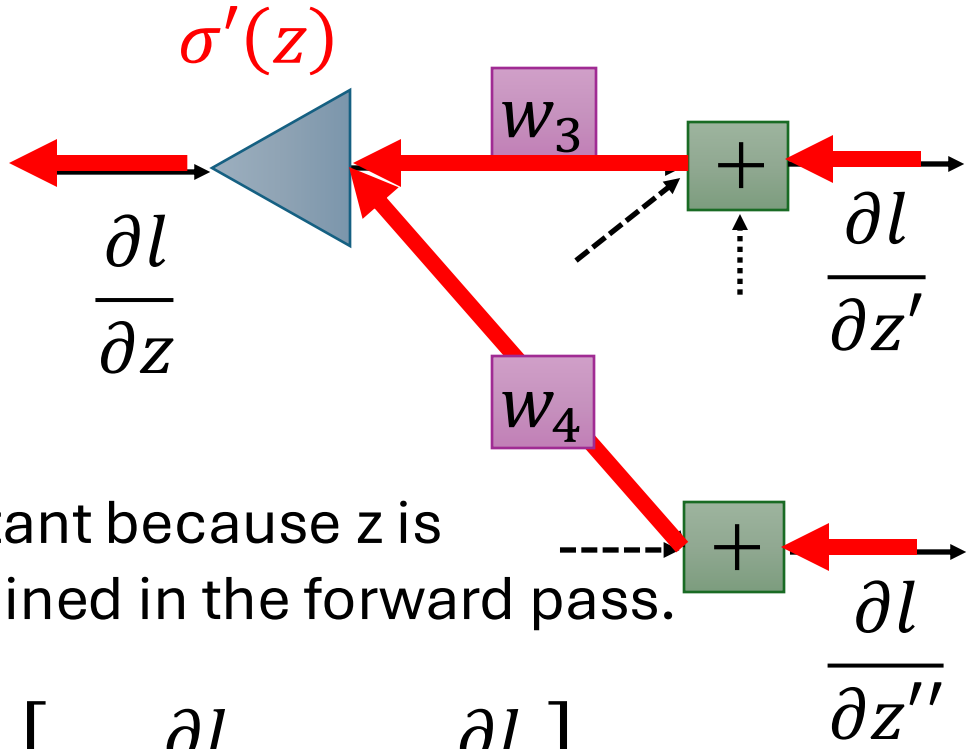
Backpropagation

Compute $\partial l/\partial z$ for all activation function inputs z



$$\frac{\partial l}{\partial z} = \sigma'(z)\left[w_3 \frac{\partial l}{\partial z'} + w_4 \frac{\partial l}{\partial z''}\right]$$

Backpropagation

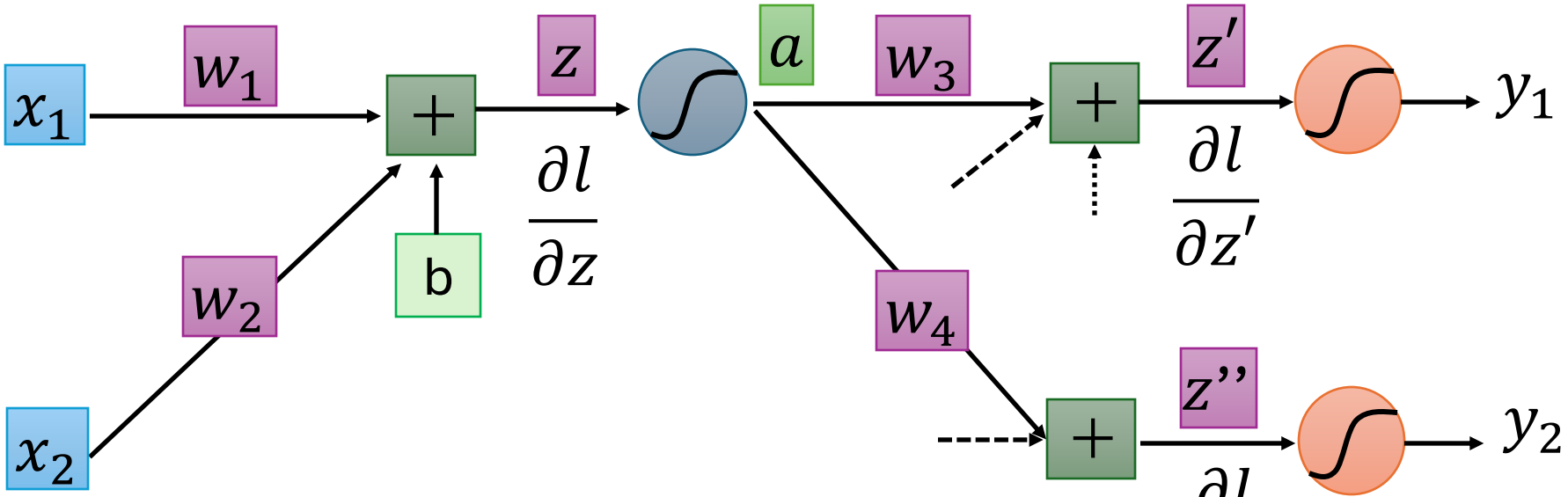


$\sigma'(z)$ is a constant because z is already determined in the forward pass.

$$\frac{\partial l}{\partial z} = \sigma'(z)\left[w_3 \frac{\partial l}{\partial z'} + w_4 \frac{\partial l}{\partial z''}\right]$$

Backpropagation

Compute $\partial l/\partial z$ for all activation function inputs z



Case 1. Output Layer

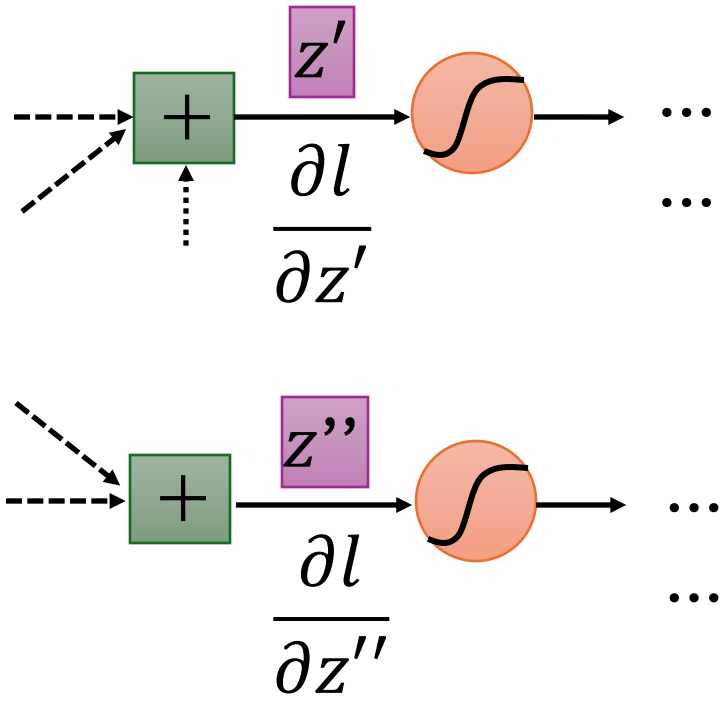
$$\frac{\partial l}{\partial z'} = \frac{\partial y_1}{\partial z'}\frac{\partial l}{\partial y_1} \qquad \frac{\partial l}{\partial z''} = \frac{\partial y_2}{\partial z''}\frac{\partial l}{\partial y_2}$$

Done!

Backpropagation

Compute $\partial l / \partial z$ for all activation function inputs z

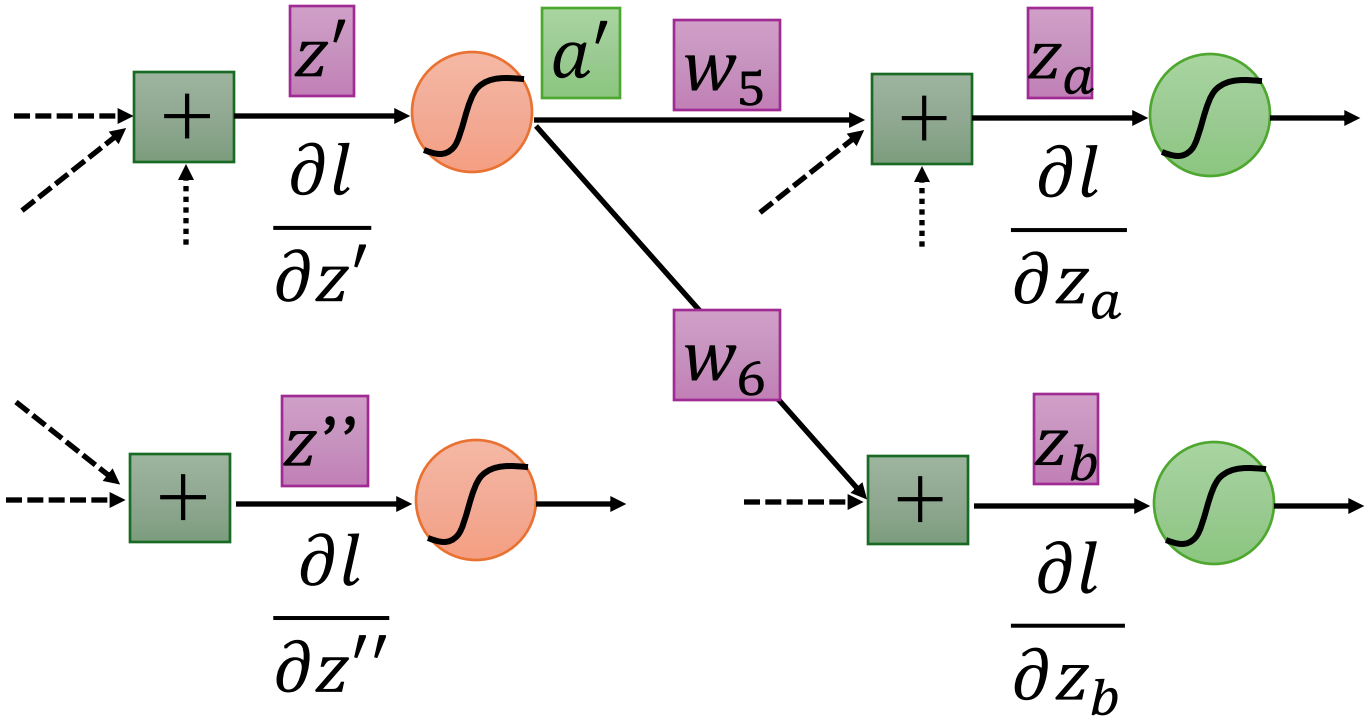
Case 2. Not Output Layer



Backpropagation

Compute $\partial l / \partial z$ for all activation function inputs z

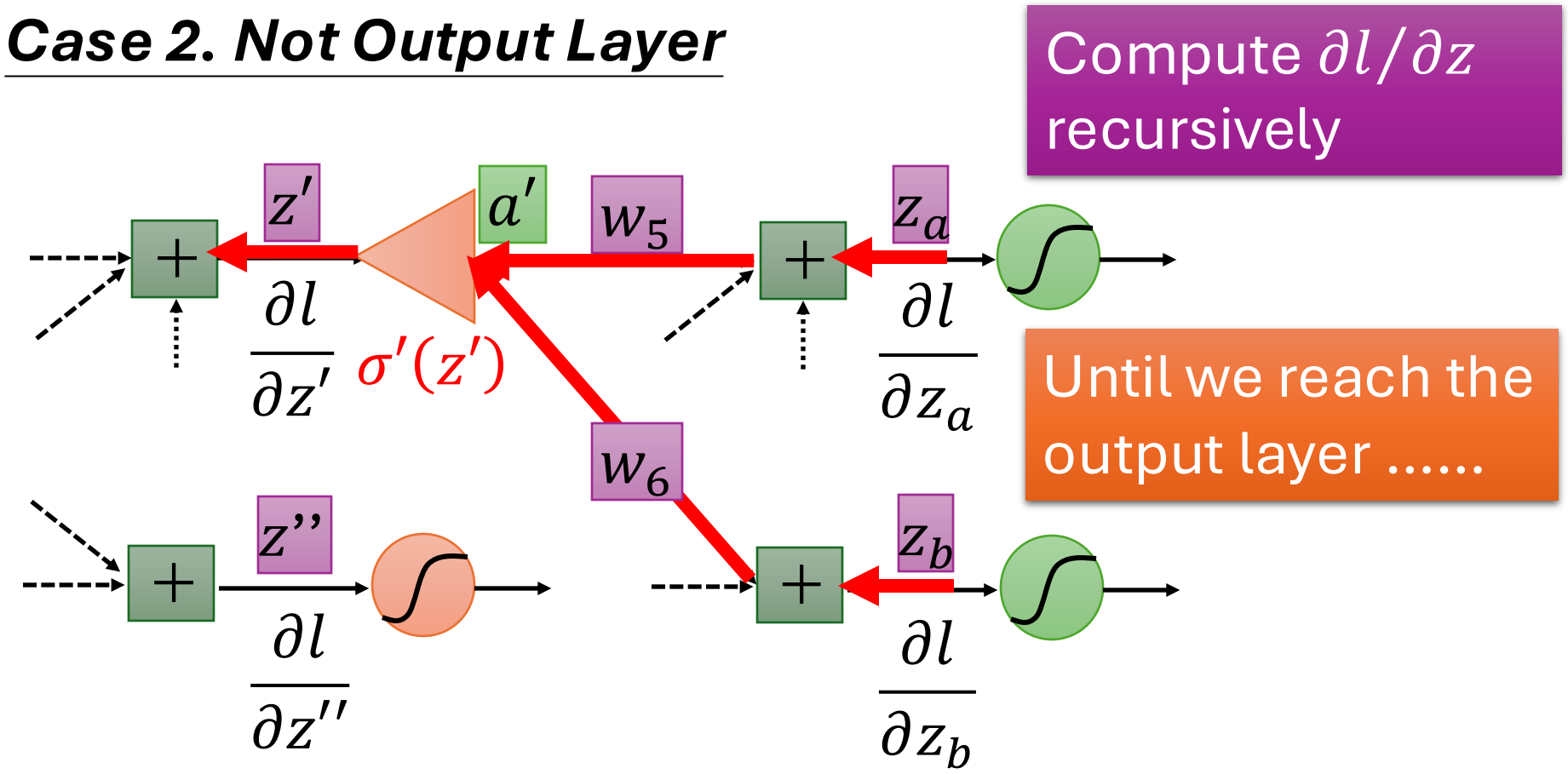
Case 2. Not Output Layer



Backpropagation

Compute $\partial l/\partial z$ for all activation function inputs z

Case 2. Not Output Layer



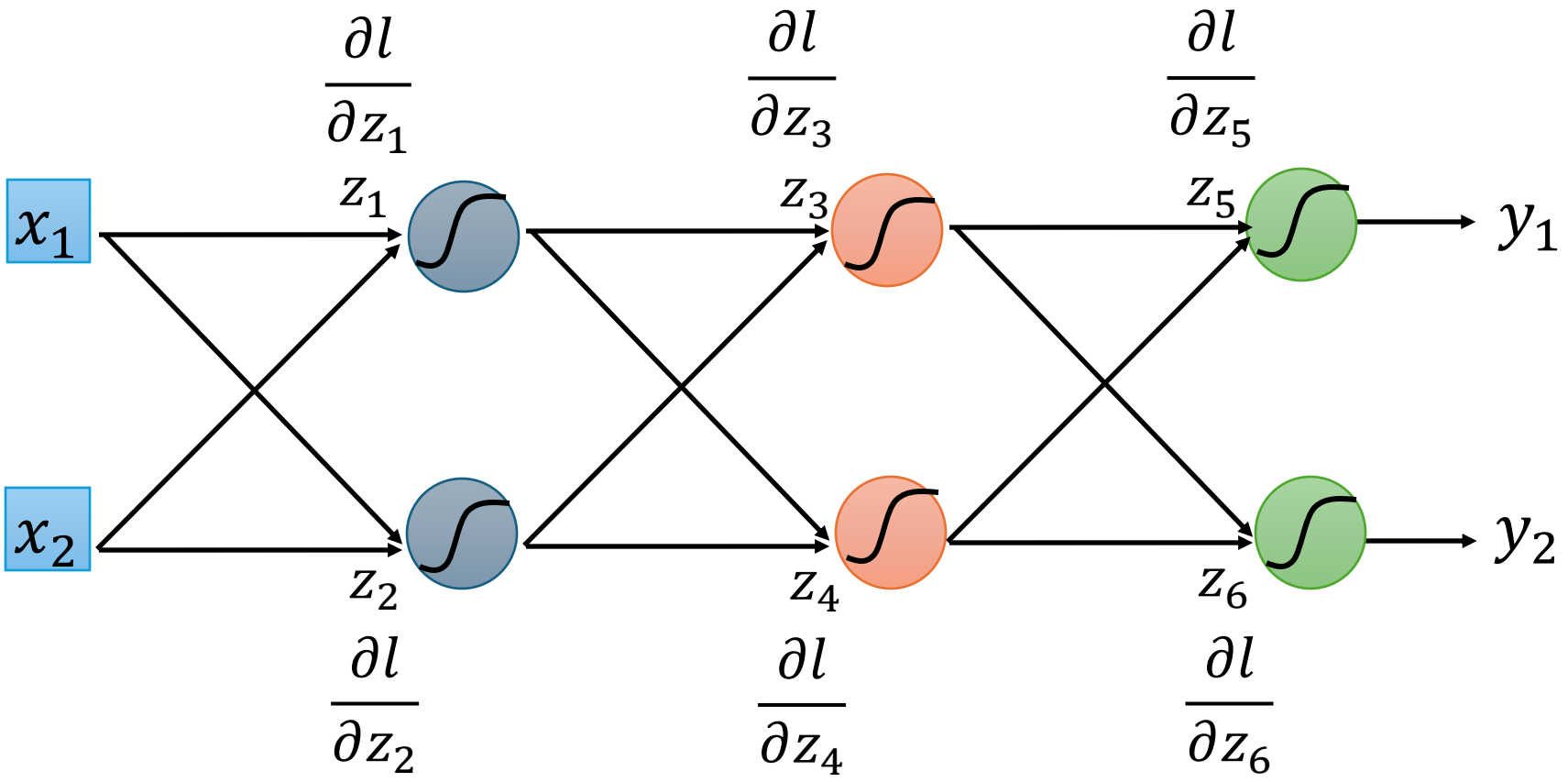
Compute $\partial l/\partial z$ recursively

Until we reach the output layer ……

Backpropagation

Compute $\partial l/\partial z$ for all activation function inputs z

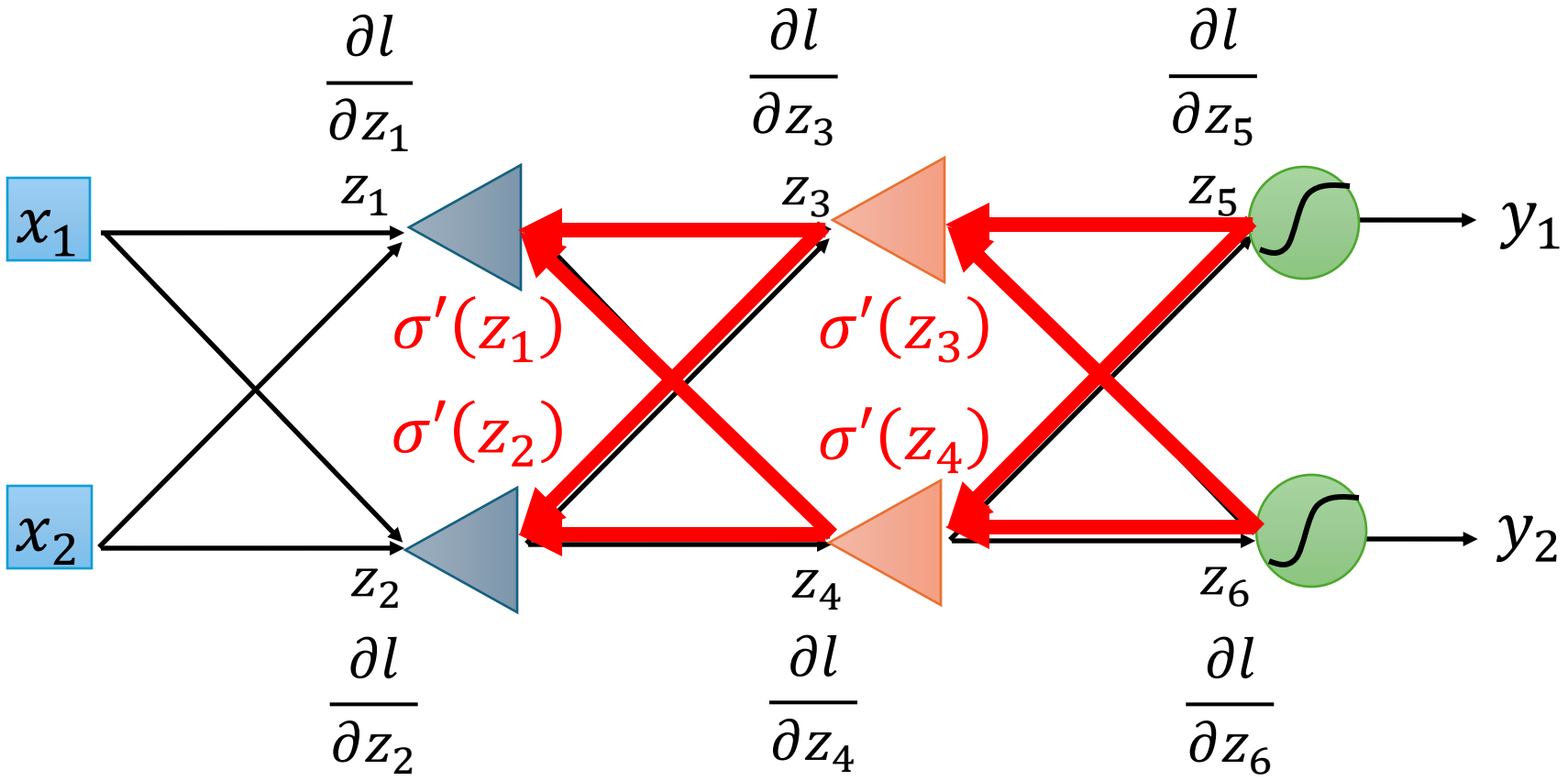
Compute $\partial l/\partial z$ from the output layer



Backpropagation

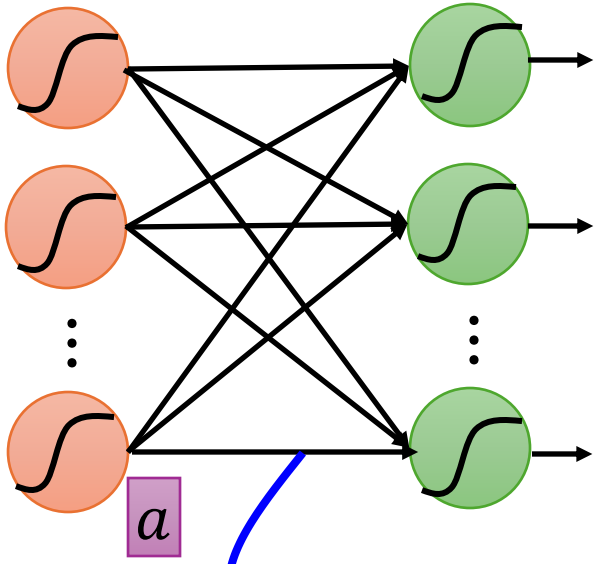
Compute $\partial l/\partial z$ for all activation function inputs z

Compute $\partial l/\partial z$ from the output layer

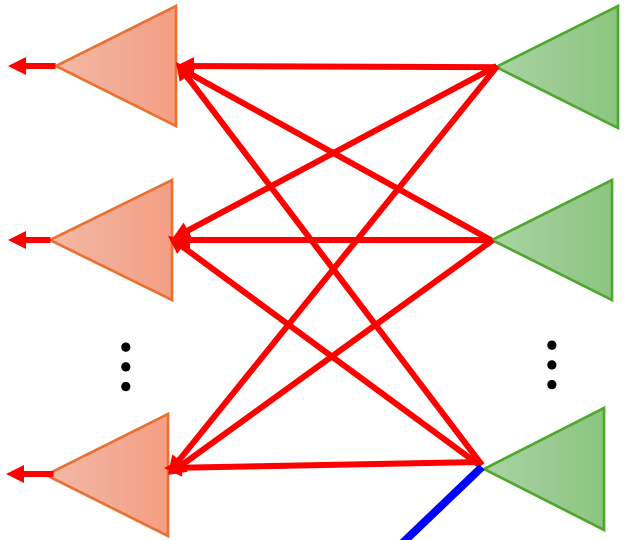


Backpropagation

Forward Pass



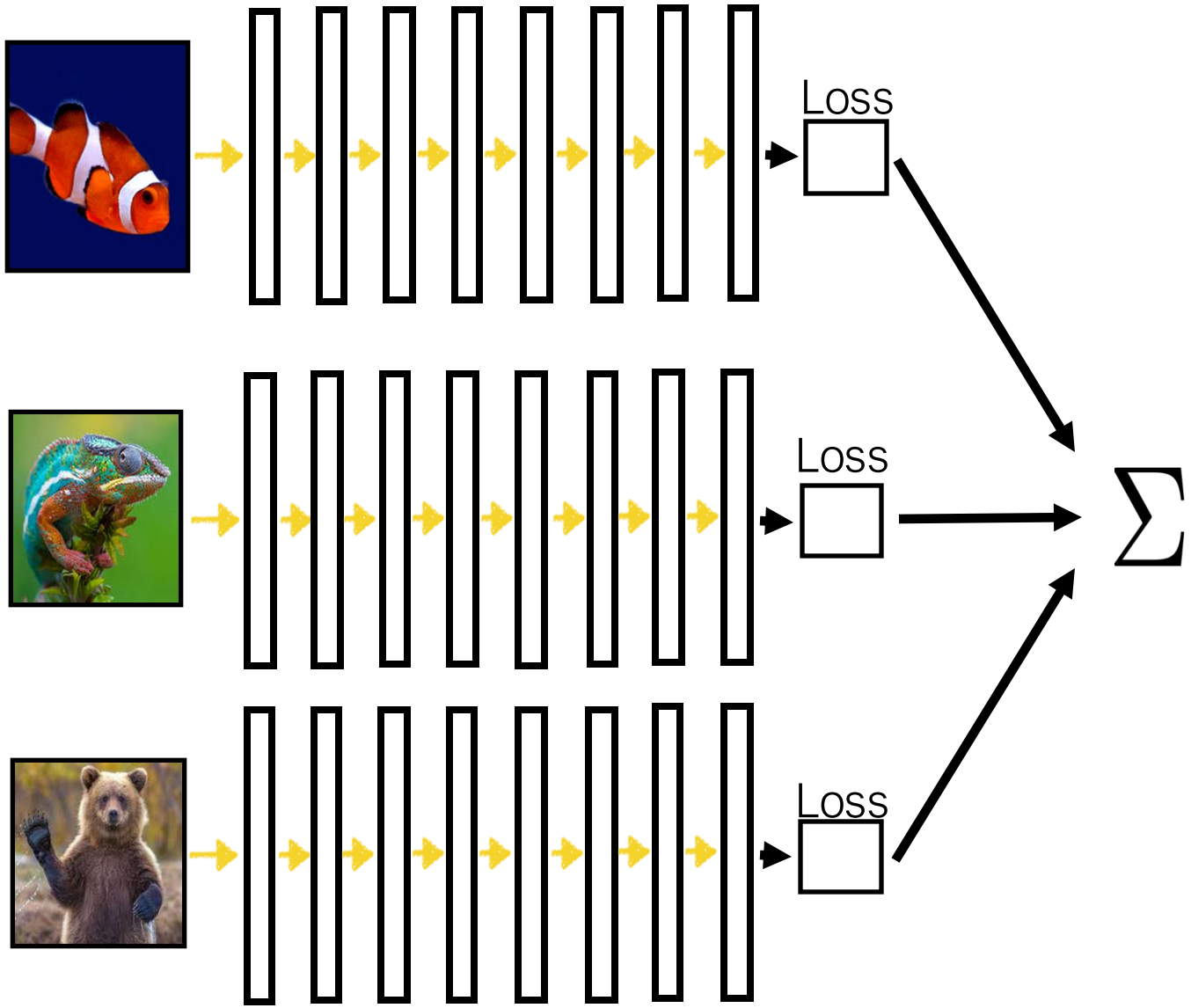
Backward Pass

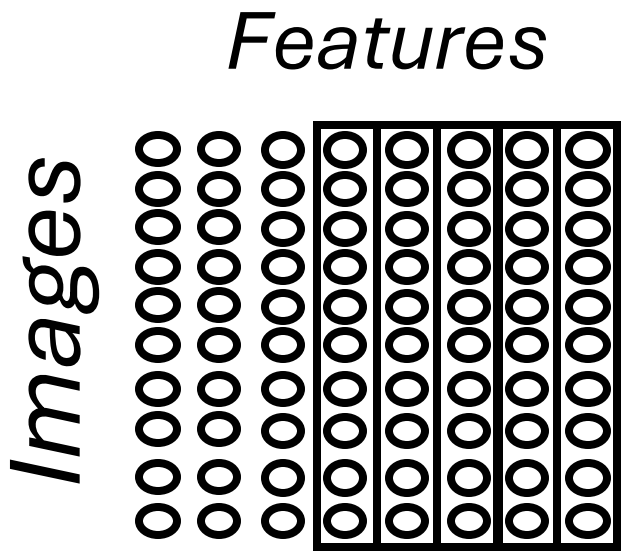


$$\frac{\partial z}{\partial w} = a \quad \text{X} \quad \frac{\partial l}{\partial z} = \frac{\partial l}{\partial w}$$

for all w

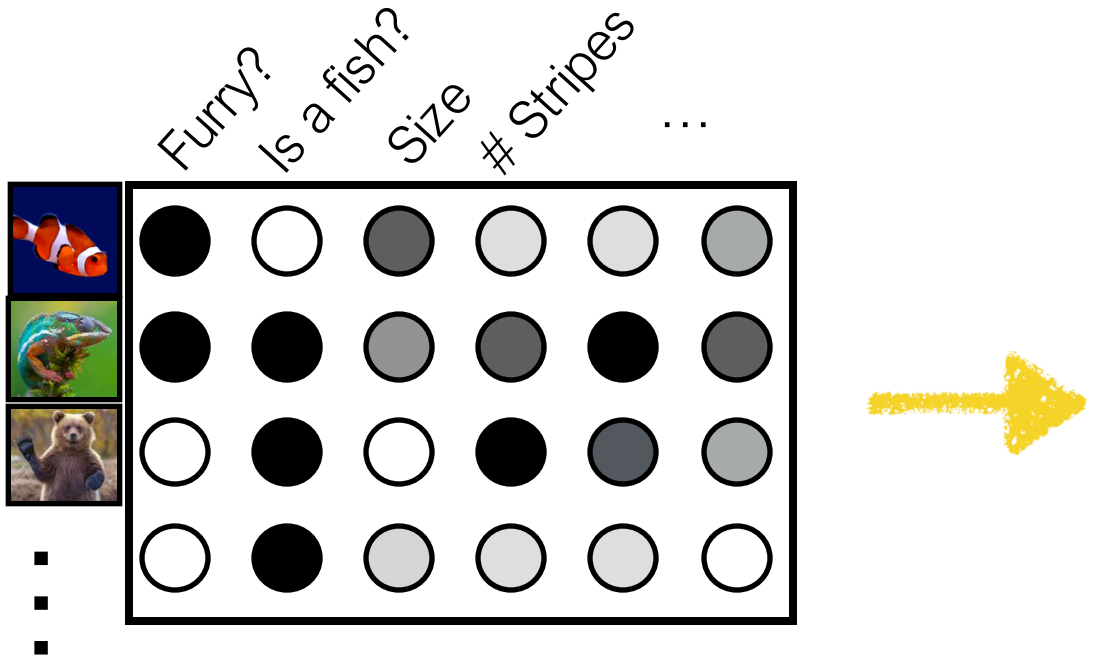
Parallel processing





Tensors

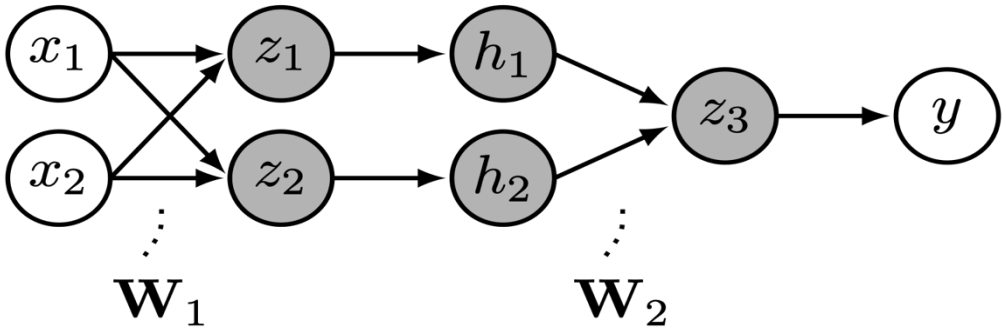
$$\mathbf{h}_1 \in \mathbb{R}^{N_{\text{batch}} \times C_1}$$



neurons
features
units
#
“channels”

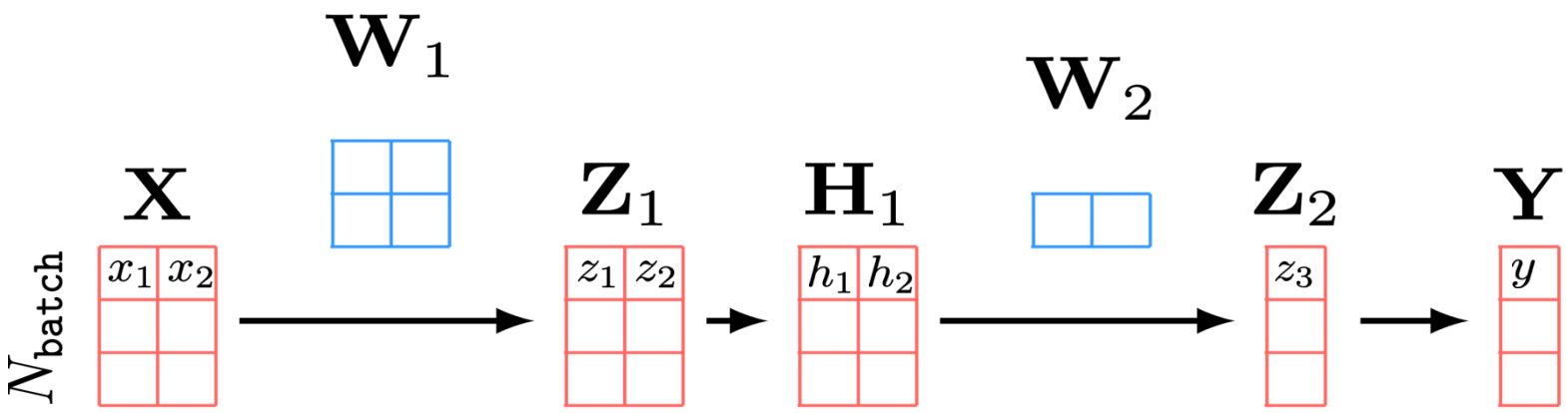
Each layer is a representation of the data

Tensor



$$\mathbf{z} = \mathbf{W}_1\mathbf{x} + \mathbf{b}_1$$
$$\mathbf{h} = g(\mathbf{z})$$
$$z_3 = \mathbf{W}_2\mathbf{h} + b_2$$
$$y = 1(z_3 > 0)$$

Tensor processing with batch size = 3:



Backpropagation over batches

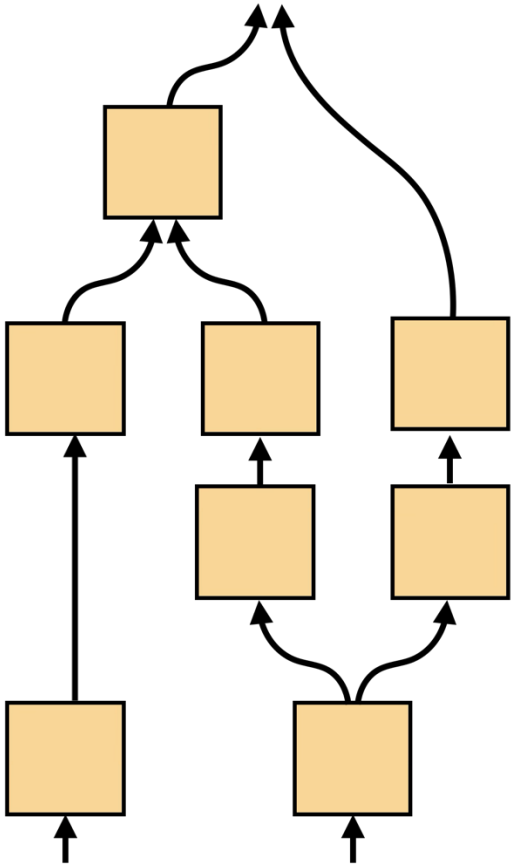
Typically we want to minimize the average cost over lots of datapoints:

$$L = \frac{1}{N}\sum l_\theta(x^i) \; or \; l(x^i;\theta)$$

Then the gradient of the total cost is just the average of all the gradients of all the per-datapoint costs:

$$\frac{\partial L}{\partial \theta} = \frac{1}{N}\sum \frac{\partial l_\theta(x^i)}{\partial \theta}$$

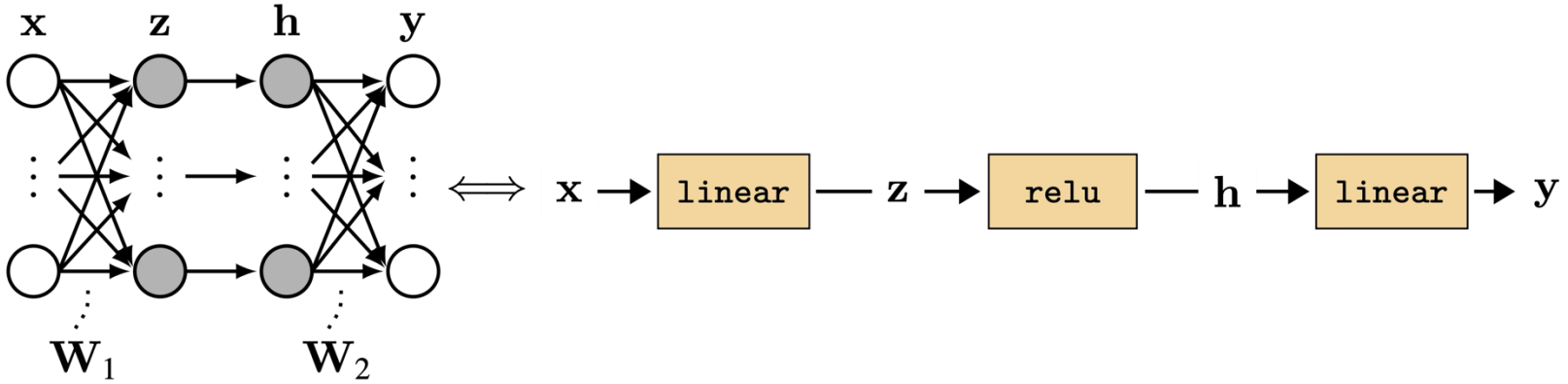
Computation graphs



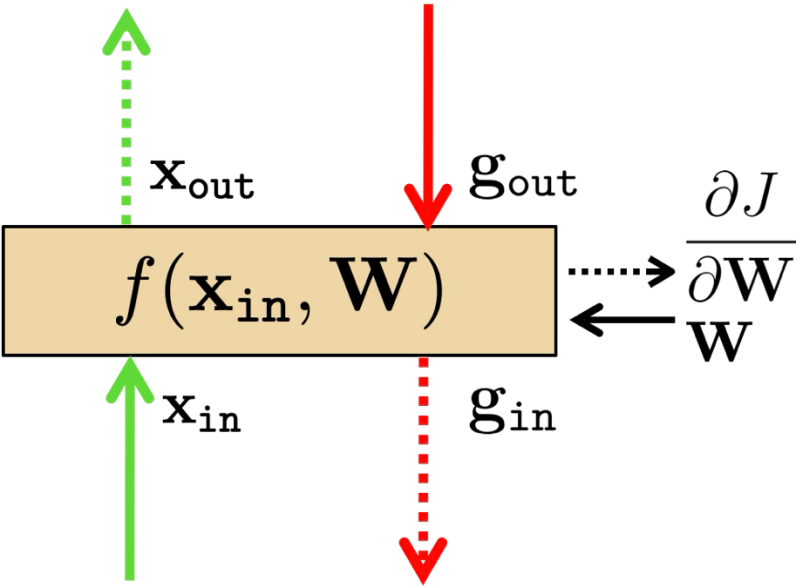
A graph of functional transformations, nodes ( ), that when strung together perform some useful computation.

Deep learning deals (primarily) with computation graphs that take the form of **directed acyclic graphs** (**DAGs**), and for which each node is differentiable.

Computation



Linear layer



- Forward propagation: $\mathbf{x}_{\text{out}} = f(\mathbf{x}_{\text{in}}, \mathbf{W}) = \mathbf{W}\mathbf{x}_{\text{in}}$
- Backprop to input:

$$\mathbf{g}_{\text{in}} = \mathbf{g}_{\text{out}} \cdot \frac{\partial f(\mathbf{x}_{\text{in}}, \mathbf{W})}{\partial \mathbf{x}_{\text{in}}} = \mathbf{g}_{\text{out}} \cdot \frac{\partial \mathbf{x}_{\text{out}}}{\partial \mathbf{x}_{\text{in}}}$$

$$\triangleq \mathbf{g}_{\text{out}} \cdot \mathbf{L}^{\mathbf{x}}$$

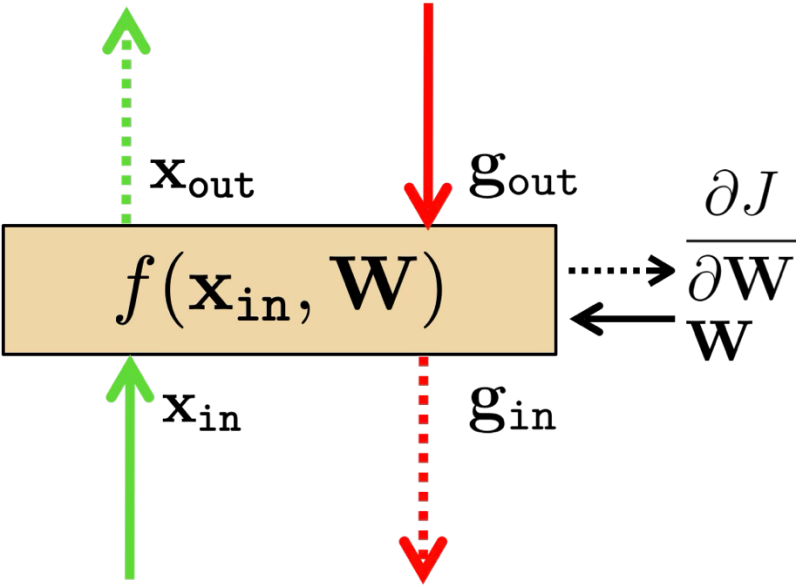
If we look at the i component of output x_{out}, with respect to the j component of the input, x_{in}:

$$\frac{\partial \mathbf{x}_{\text{out}_i}}{\partial \mathbf{x}_{\text{in}_j}} = \mathbf{W}_{ij} \longrightarrow \frac{\partial f(\mathbf{x}_{\text{in}}, \mathbf{W})}{\partial \mathbf{x}_{\text{in}}} = \mathbf{W}$$

Therefore:

$$\mathbf{g}_{\text{in}} = \mathbf{g}_{\text{out}} \cdot \mathbf{W}$$

Linear layer



- Forward propagation: $\mathbf{x}_{\text{out}} = f(\mathbf{x}_{\text{in}}, \mathbf{W}) = \mathbf{W}\mathbf{x}_{\text{in}}$
- Backprop to weights:

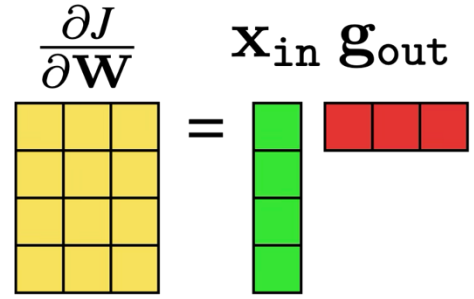
$$\frac{\partial J}{\partial \mathbf{W}} = \mathbf{g}_{\text{out}} \cdot \frac{\partial f(\mathbf{x}_{\text{in}}, \mathbf{W})}{\partial \mathbf{W}} = \mathbf{g}_{\text{out}} \cdot \frac{\partial \mathbf{x}_{\text{out}}}{\partial \mathbf{W}}$$

If we look at how the parameter W_{ij} changes the cost, only the i component of the output will change, therefore:

$$\frac{\partial J}{\partial \mathbf{W}_{ij}} = \frac{\partial J}{\partial \mathbf{x}_{\text{out}_i}} \cdot \frac{\partial \mathbf{x}_{\text{out}_i}}{\partial \mathbf{W}_{ij}} = \frac{\partial J}{\partial \mathbf{x}_{\text{out}_i}} \cdot \mathbf{x}_{\text{in}_j}$$

$$\frac{\partial \mathbf{x}_{\text{out}_i}}{\partial \mathbf{W}_{ij}} = \mathbf{x}_{\text{in}_j}$$

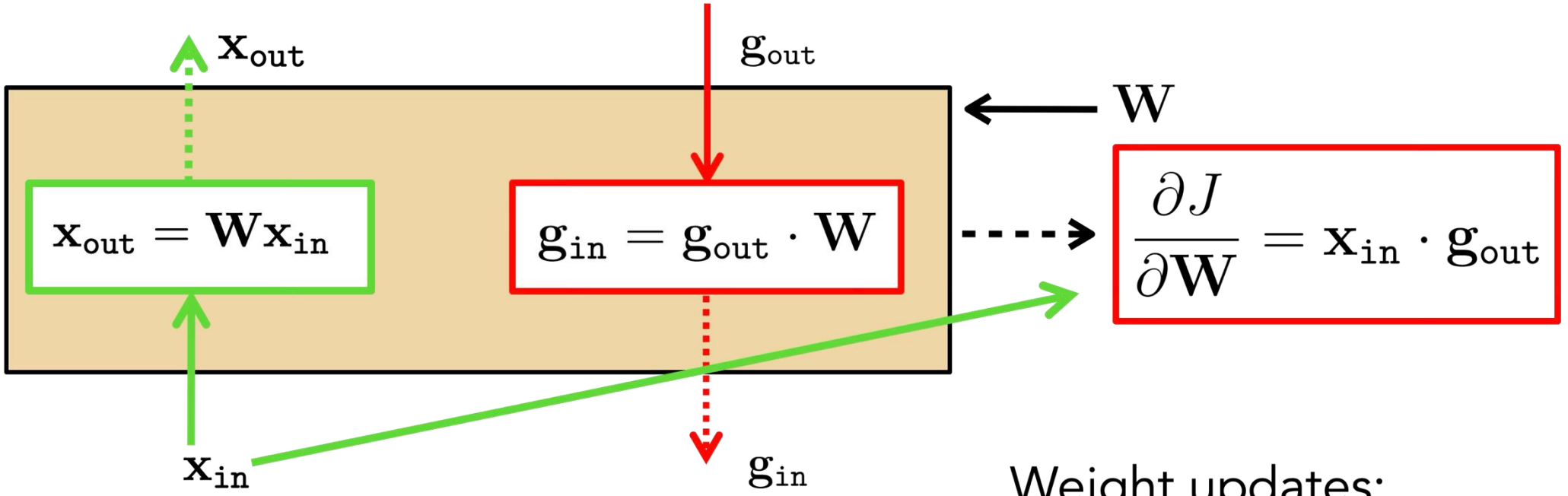
$$\frac{\partial J}{\partial \mathbf{W}} = \mathbf{x}_{\text{in}} \cdot \frac{\partial J}{\partial \mathbf{x}_{\text{out}}} = \mathbf{x}_{\text{in}} \cdot \mathbf{g}_{\text{out}}$$



And now we can update the weights:

$$\mathbf{W}^{k+1} \leftarrow \mathbf{W}^{k} - \eta \left(\frac{\partial J}{\partial \mathbf{W}} \right)^{T}$$

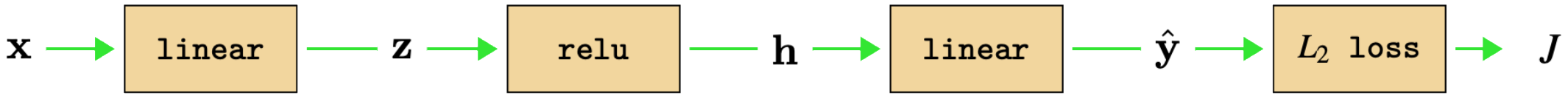
Linear layer



Weight updates:

$$\mathbf{W}^{k+1} \leftarrow \mathbf{W}^{k} - \eta \left(\frac{\partial J}{\partial \mathbf{W}} \right)^{T}$$

MLP: forward



$$\mathbf{z} = \mathbf{W}_1 \mathbf{x}$$

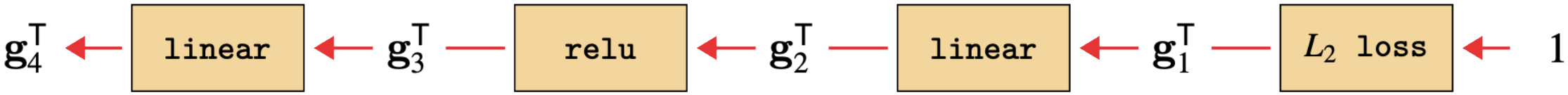
$$\mathbf{h} = \mathtt{relu}(\mathbf{z})$$

$$\hat{\mathbf{y}} = \mathbf{W}_2 \mathbf{h}$$

$$J = \|\hat{\mathbf{y}} - \mathbf{y}\|_2^2$$

MLP: backward

$$\mathbf{g}_{\text{in}}^{\mathsf{T}} = (\mathbf{g}_{\text{out}}\mathbf{W})^{\mathsf{T}} = \mathbf{W}^{\mathsf{T}}\mathbf{g}_{\text{out}}^{\mathsf{T}}$$



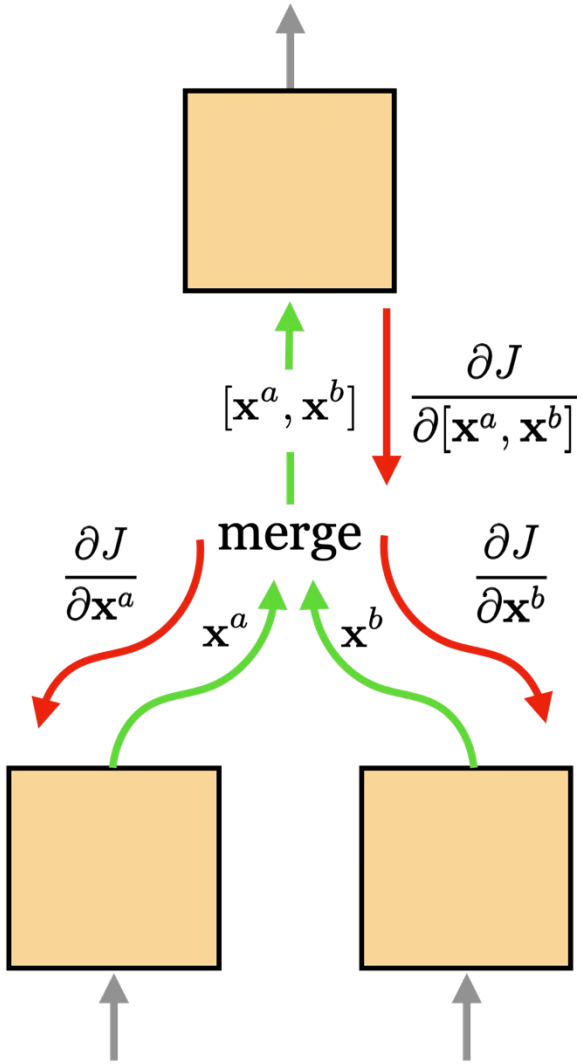
$$\mathbf{g}_4^{\mathsf{T}} = \mathbf{W}_1^{\mathsf{T}}\,\mathbf{g}_3^{\mathsf{T}}$$

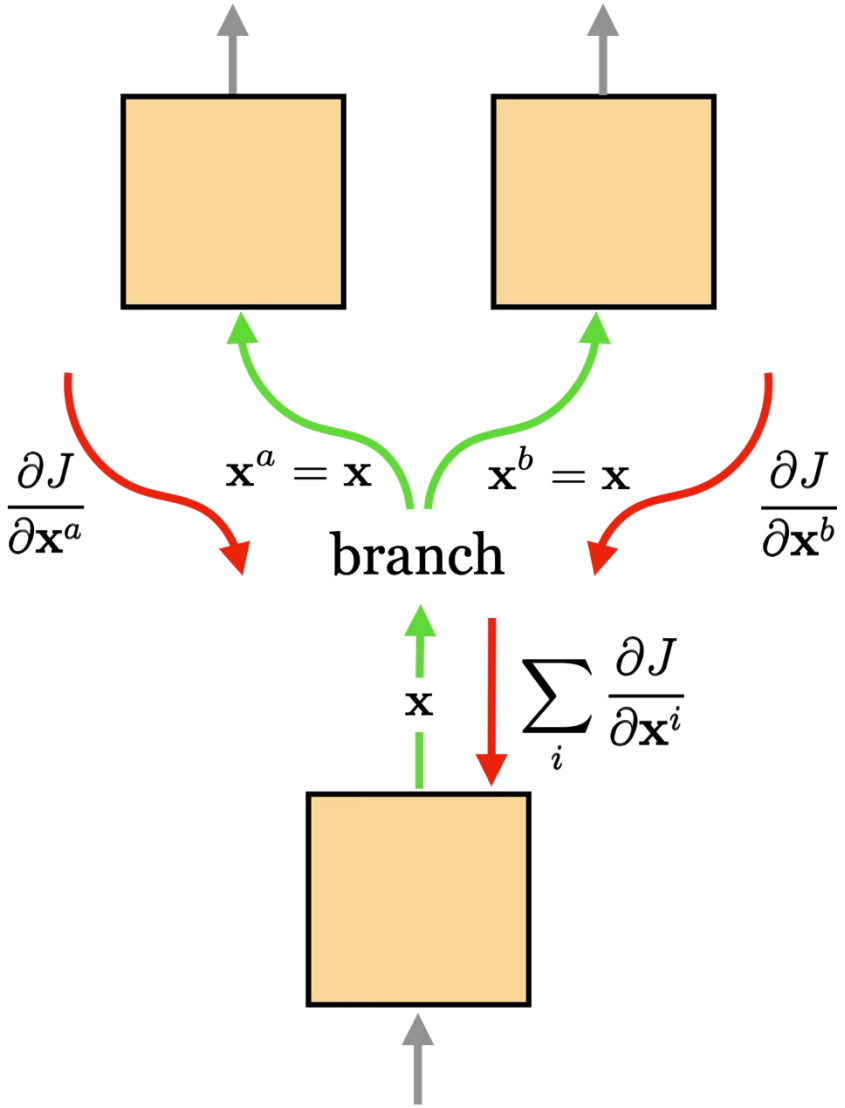
$$\mathbf{g}_3^{\mathsf{T}} = \mathbf{H}'^{T}\,\mathbf{g}_2^{\mathsf{T}}$$

$$\mathbf{g}_2^{\mathsf{T}} = \mathbf{W}_2^{\mathsf{T}}\mathbf{g}_1^{\mathsf{T}}$$

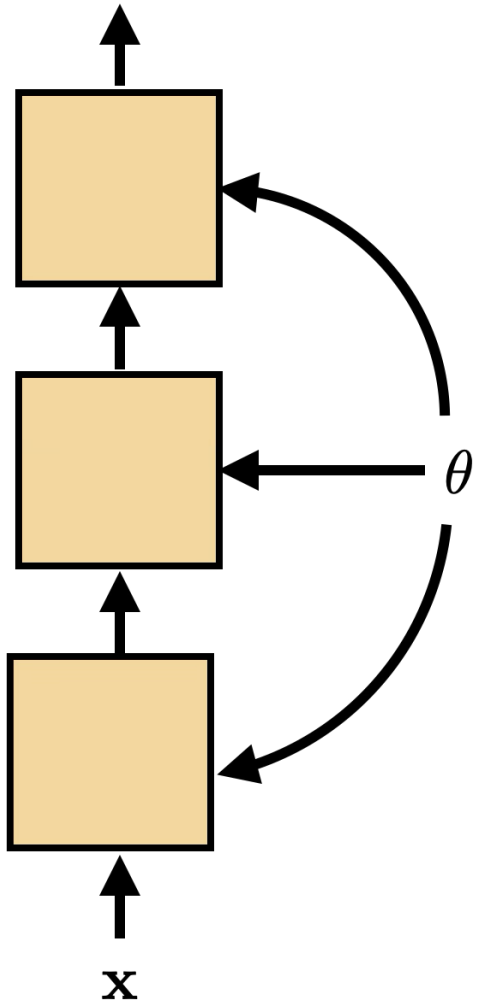
$$\mathbf{g}_1^{\mathsf{T}} = 2(\hat{\mathbf{y}} - \mathbf{y})1$$

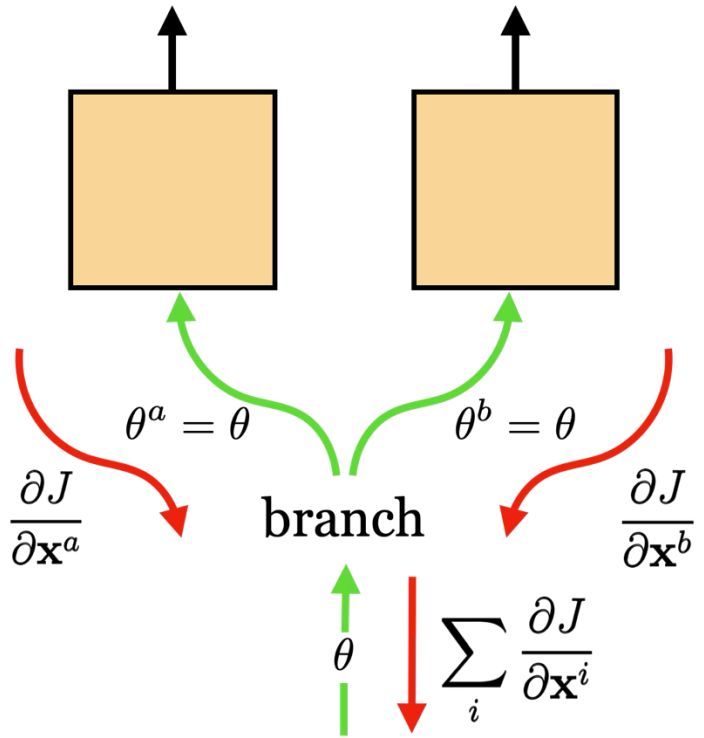
Computational graph





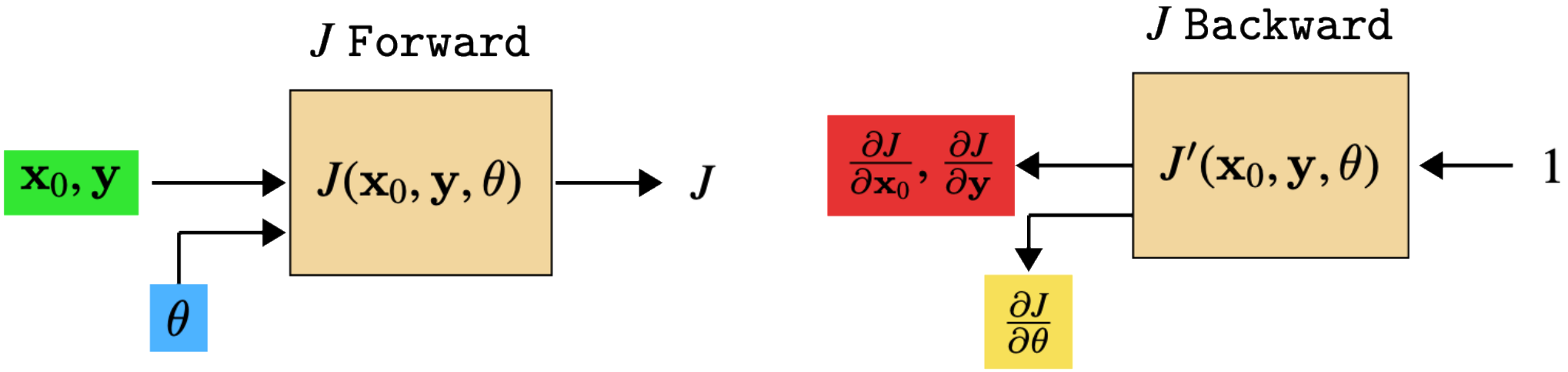
Parameter sharing





Parameter sharing —> sum gradients

Computation module



Higher order

```
import torch

# 1. Setup our input tensor
x = torch.tensor([2.0], requires_grad=True)

# 2. Define the first function (y = x^3)
y = x**3
print(f"y = {y.item()}")

# 3. --- First Derivative (y') ---
# We tell autograd to build a graph OF THE GRADIENT
y.backward(create_graph=True)

# The gradient is now stored in x.grad
# This is y' = 3x^2 = 3*(2^2) = 12.0
grad_1 = x.grad
print(f"First derivative (y') at x=2: {grad_1.item()}")

# 4. --- Second Derivative (y'') ---
# grad_1 is not just a number; it's a tensor that's part of a graph
# that leads back to x. We can now backpropagate from grad_1.
# We need to clear the old .grad on x first.
x.grad.zero_()

# Now, we take the derivative of the first derivative
grad_1.backward()

# The new gradient is the second derivative
# This is y'' = 6x = 6*(2) = 12.0
grad_2 = x.grad
print(f"Second derivative (y'') at x=2: {grad_2.item()}")
```

Pytorch example

1. The "Engine"
torch::autograd::Function
This class is **stateless**. It defines the raw computation and holds no parameters.

- Defines the `static forward(...)` method for the calculation.
- Defines the `static backward(...)` method for the manual gradient (backpropagation).
- This is where we implement the core math.

2. The "Wrapper"
torch::nn::Module
This class is **stateful**. This is the user-facing layer that holds parameters.

- Holds and registers parameters like `weight` and `bias`.
- Its `forward(...)` method calls the "Engine" to perform the calculation.
- This is what you're used to (e.g., `torch::nn::Linear`).

Pytorch example: Engine

```
struct MyLinearFunction : public
torch::autograd::Function<MyLinearFunction> {
  static torch::Tensor forward(
      torch::autograd::AutogradContext* ctx,
      torch::Tensor input,
      torch::Tensor weight,
      torch::Tensor bias) {

    // Save tensors for the backward pass
    ctx->save_for_backward({input, weight});

    // Y = X * W^T + B
    auto output = torch::matmul(input, weight.t()) +
bias;
    return output;
  }
  // ...
};
```

- It's a `static` method.
- It receives a `ctx` (context) object to store data.
- **`ctx->save_for_backward(...)`** is critical. We save `input` and `weight` because we'll need them to calculate the gradients in the backward pass.
- The calculation `torch::matmul(input, weight.t()) + bias` is the standard linear layer formula.

Pytorch example: Engine

This is where we manually implement backpropagation. This method receives `grad_output` (the "blame" from the next layer,) and must return the gradients for each of the inputs to `forward`.

```
static torch::autograd::tensor_list backward(
    torch::autograd::AutogradContext* ctx,
    torch::autograd::tensor_list grad_outputs) {

  // This is dE/dY
  auto grad_output = grad_outputs[0];

  // Load saved tensors
  auto saved = ctx->get_saved_variables();
  auto input = saved[0];   // X
  auto weight = saved[1];  // W

  // --- Gradient Calculations ---
  auto grad_input = torch::matmul(grad_output, weight);
  auto grad_weight = torch::matmul(grad_output.t(), input);

  auto grad_bias = grad_output.sum(0);

  // Return gradients in the *exact* order
  // as the forward() inputs (input, weight, bias)
  return {grad_input, grad_weight, grad_bias};
 }
};
```

Regularization

Regularizing deep nets

Deep nets have millions of parameters!

On many datasets, it is easy to overfit —
we may have more free parameters than data points to constrain them.

How can we prevent the network from overfitting?

- Fewer neurons, fewer layers
- Weight decay and other regularizers
- Normalization layers

Ridge regression (weight decay)

$$f_\theta(x) = \sum_{k=0}^{K} \theta_k x^k$$

$$R(\theta) = \lambda \|\theta\|_2^2$$

Only use polynomial terms if you really need them! Most terms should be zero

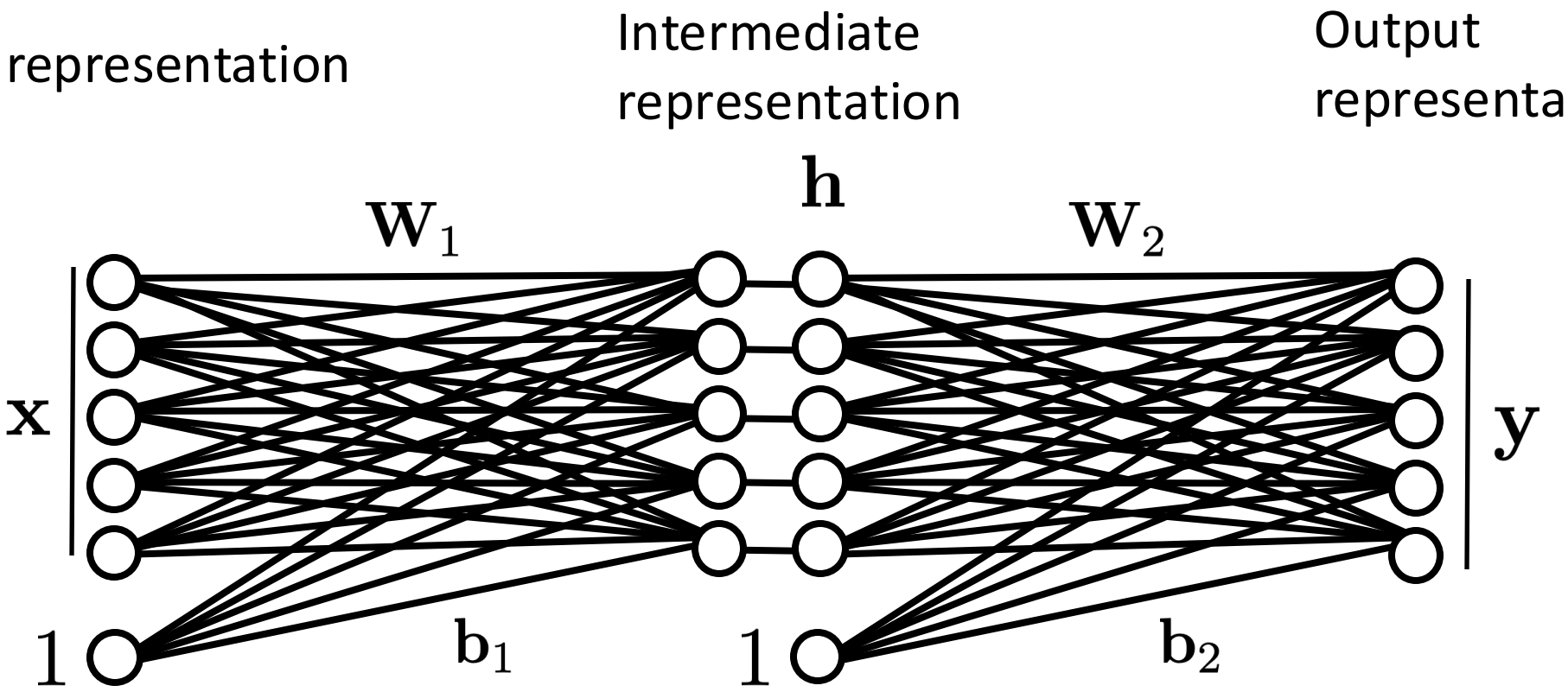
ridge regression, a.k.a., Tikhonov regularization

Ridge regression (weight decay)

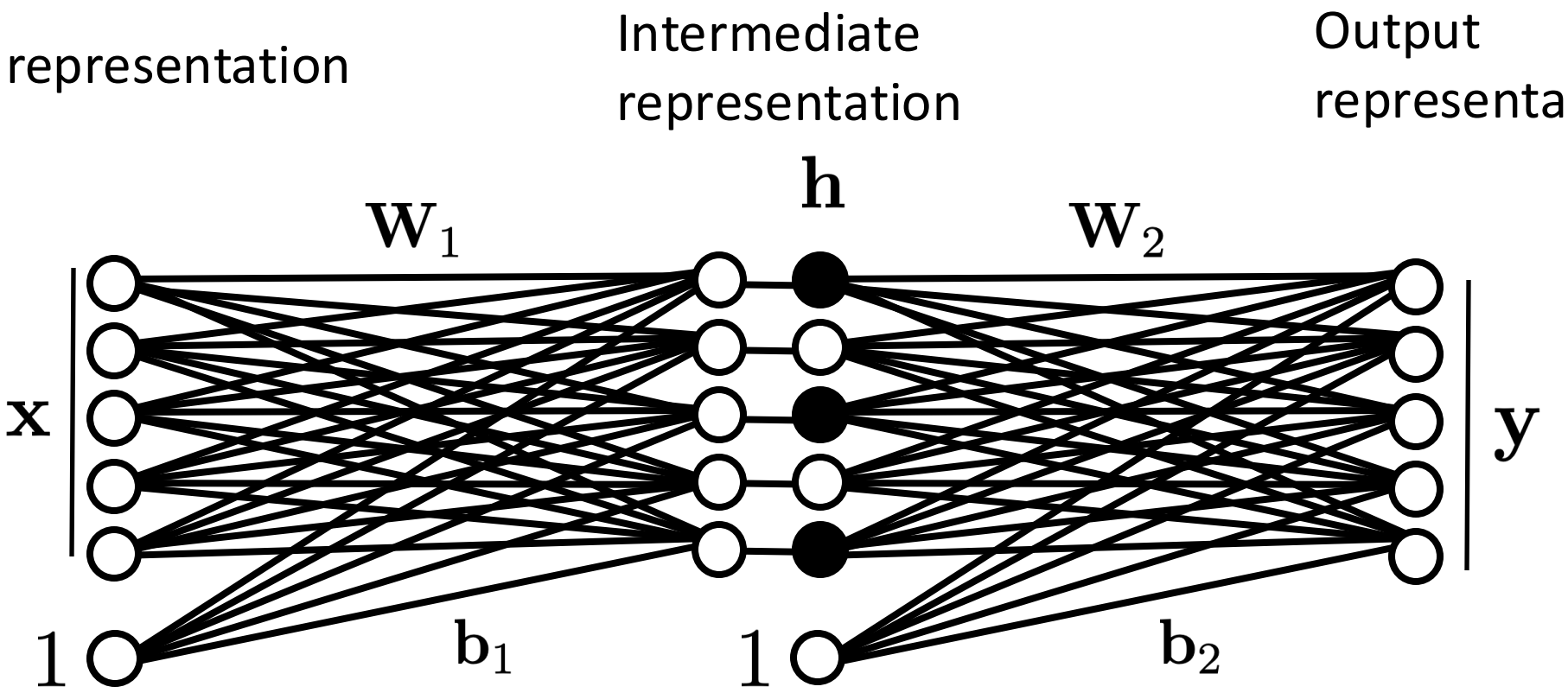
$$\theta^* = \arg\min_{\theta} \sum_{i=1}^{N} \mathcal{L}(f_\theta(\mathbf{x}^{(i)}), \mathbf{y}^{(i)}) + R(\theta)$$

$$R(\mathbf{W}) = \lambda \|\mathbf{W}\|_2^2$$ ⟵ weight decay

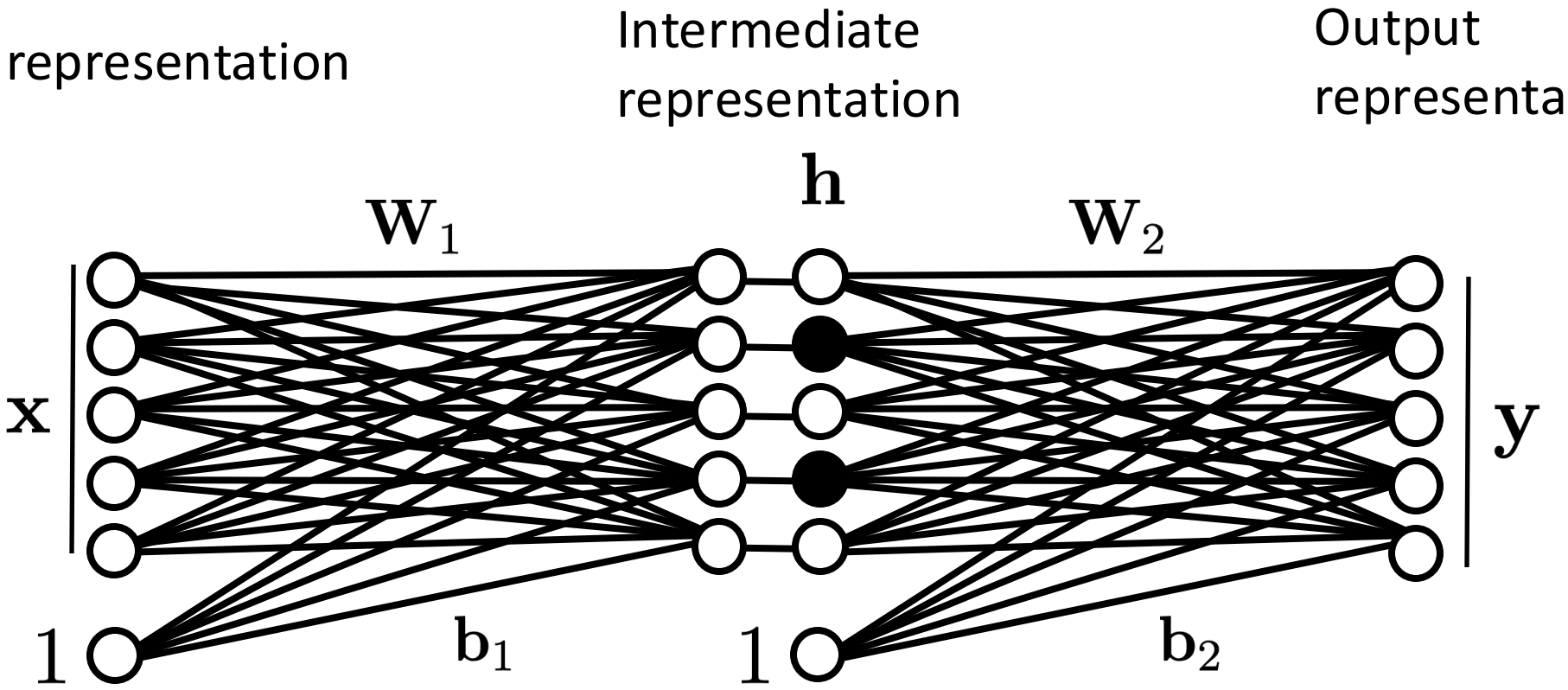
Dropout


Input representation
Intermediate representation
Output representation
$\mathbf{h}$
$\mathbf{W}_1$
$\mathbf{W}_2$
$\mathbf{x}$
$\mathbf{y}$
1
$\mathbf{b}_1$
1
$\mathbf{b}_2$

Dropout


Input representation
Intermediate representation
Output representation
$\mathbf{W}_1$
$\mathbf{h}$
$\mathbf{W}_2$
$\mathbf{x}$
$\mathbf{y}$
1
$\mathbf{b}_1$
1
$\mathbf{b}_2$

Dropout


Input representation
Intermediate representation
Output representation
$\mathbf{h}$
$\mathbf{W}_1$
$\mathbf{W}_2$
$\mathbf{x}$
$\mathbf{y}$
1
$\mathbf{b}_1$
1
$\mathbf{b}_2$

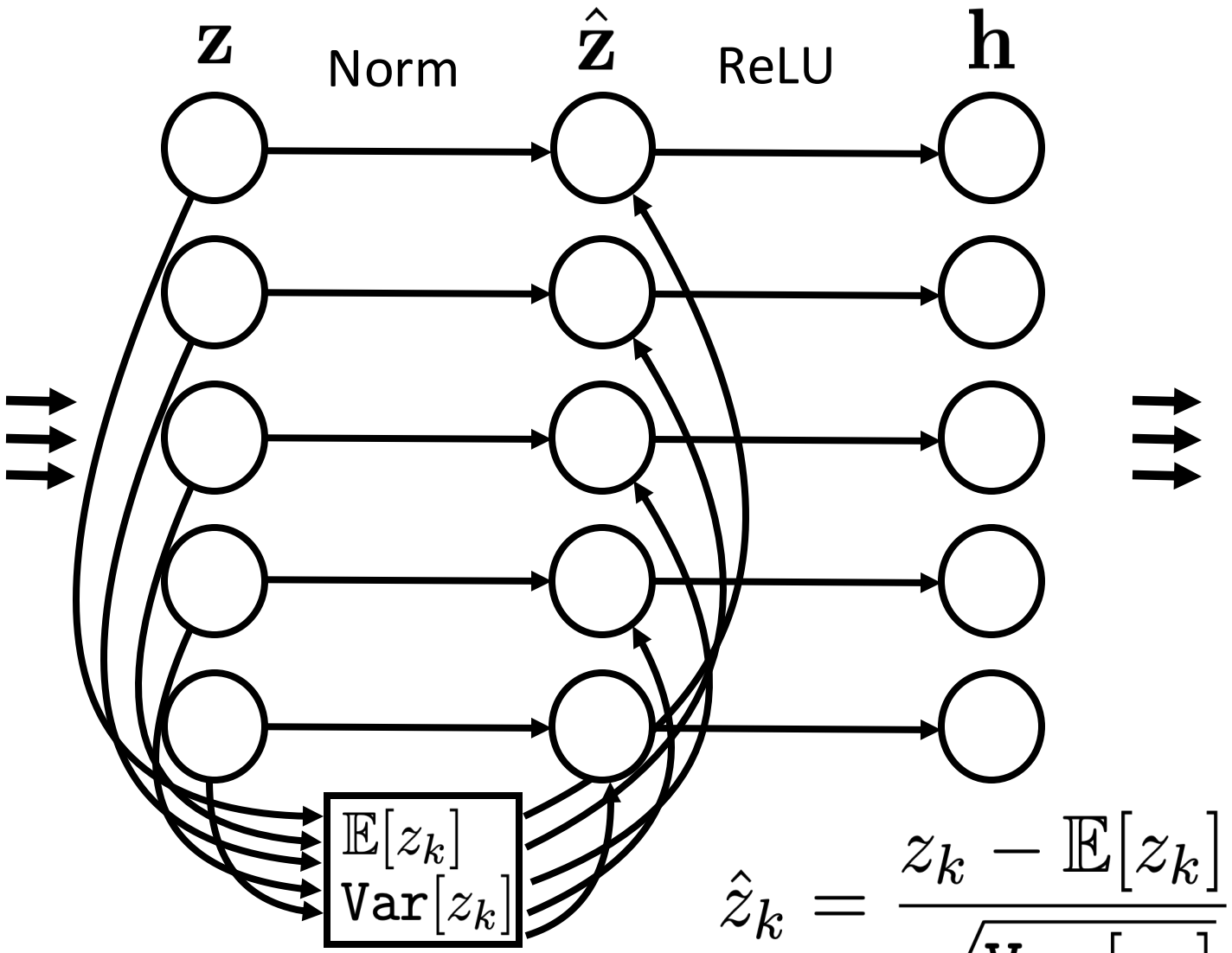
Dropout

Randomly zero out hidden units.

Prevents network from relying too much on spurious correlations between different hidden units.

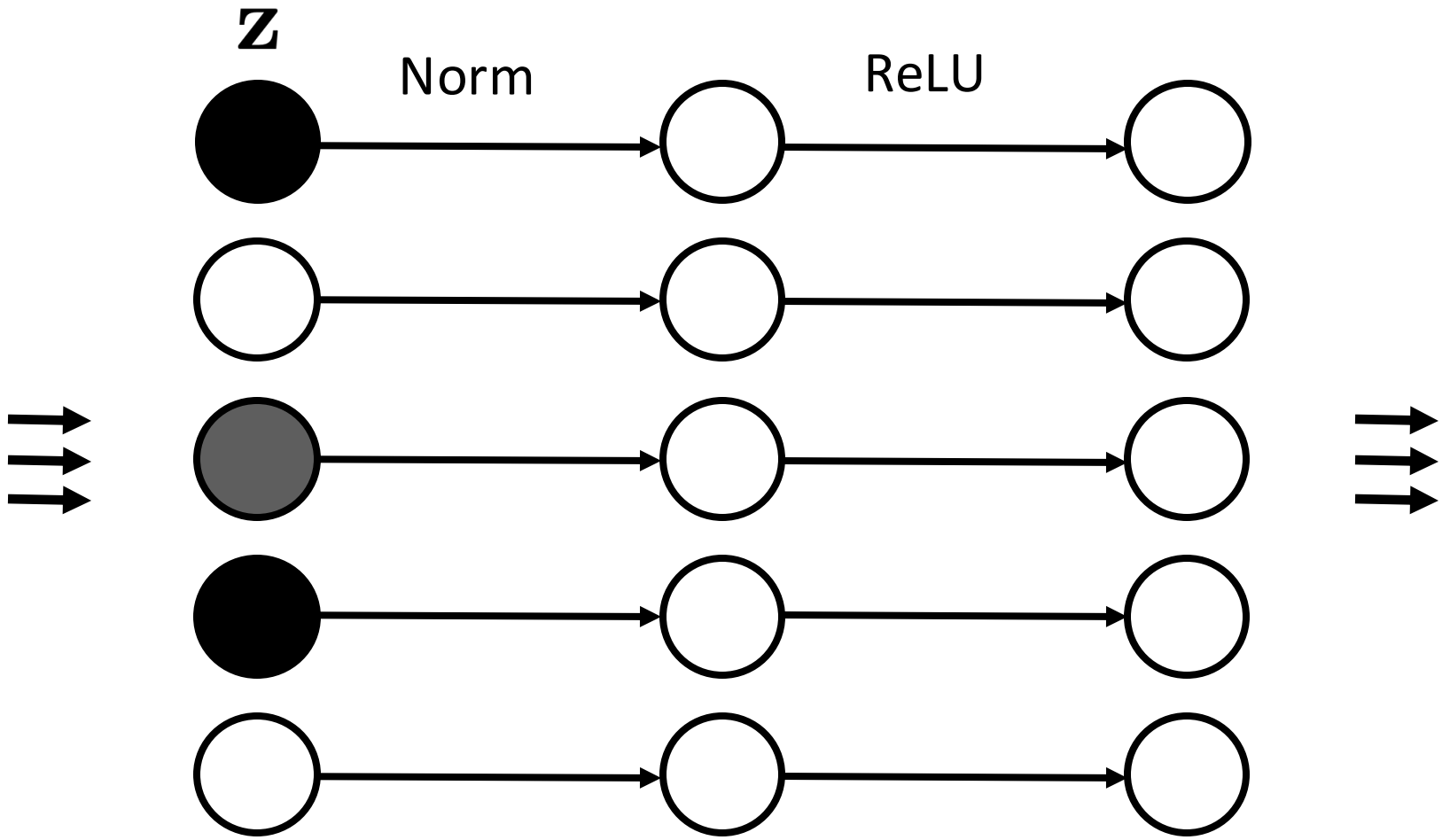
Can be understood as averaging over an exponential ensemble of subnetworks. This averaging smooths the function, thereby reducing the effective capacity of the network.

Normalization



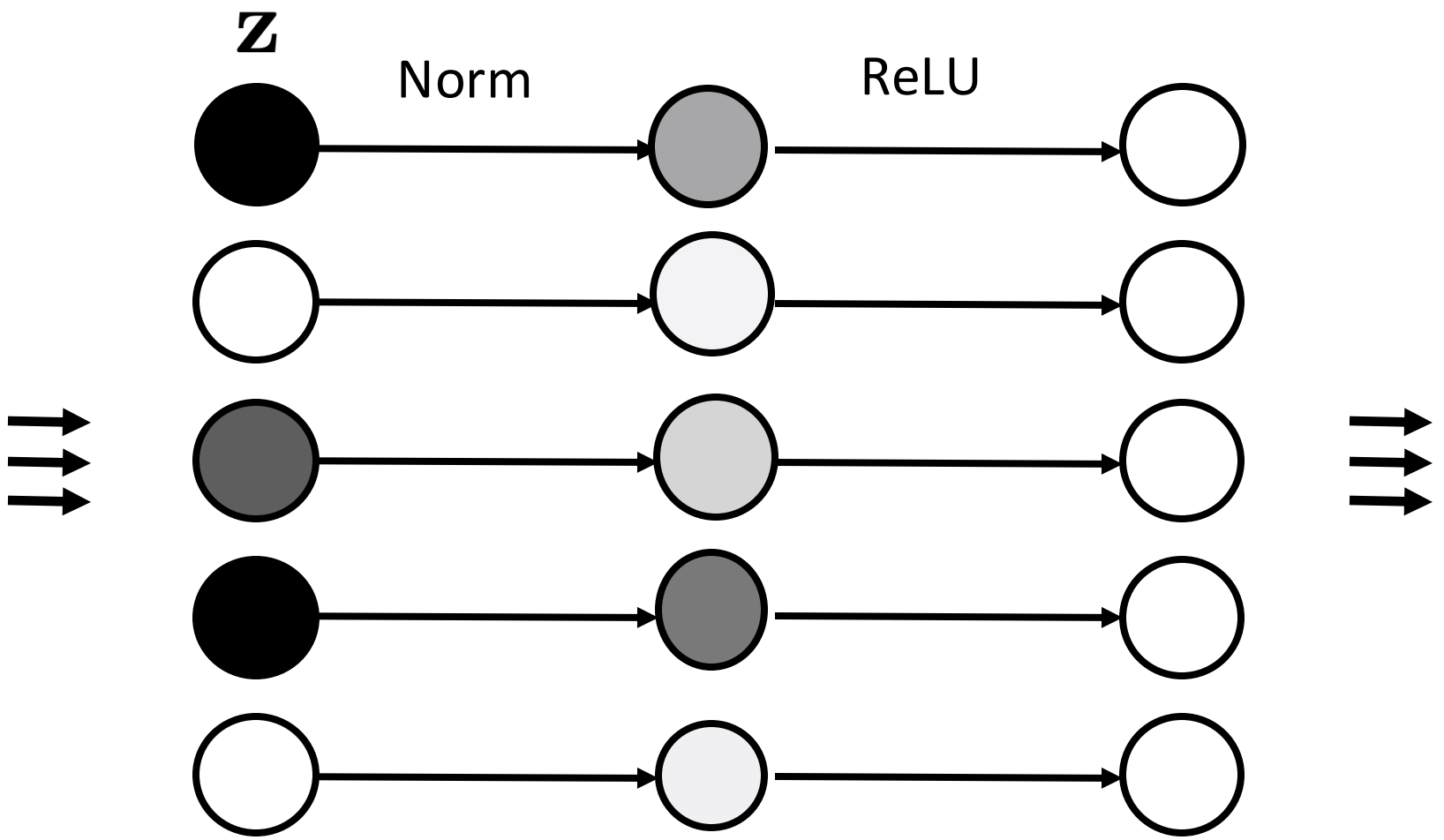
$$\hat{z}_k = \frac{z_k - \mathbb{E}[z_k]}{\sqrt{\mathtt{Var}[z_k]}}$$

Normalization



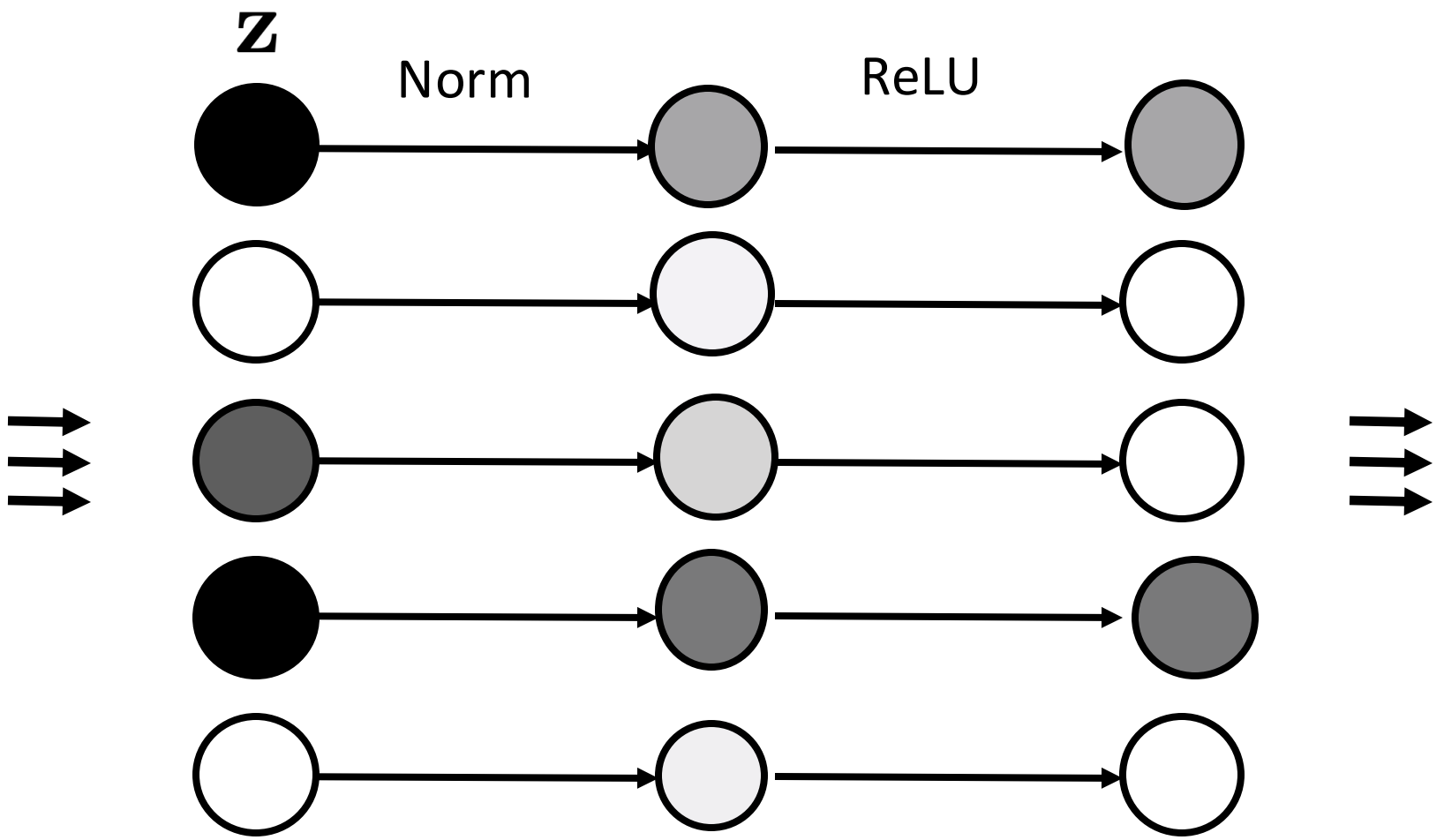
$$\hat{z}_k = \frac{z_k - \mathbb{E}[z_k]}{\sqrt{\mathbf{Var}[z_k]}}$$

Normalization



$$\hat{z}_k = \frac{z_k - \mathbb{E}[z_k]}{\sqrt{\mathrm{Var}[z_k]}}$$

Normalization



$$\hat{z}_k = \frac{z_k - \mathbb{E}[z_k]}{\sqrt{\mathrm{Var}[z_k]}}$$

Why do we need normalization

Keep track of mean and variance of a unit (or a population of units) over time.

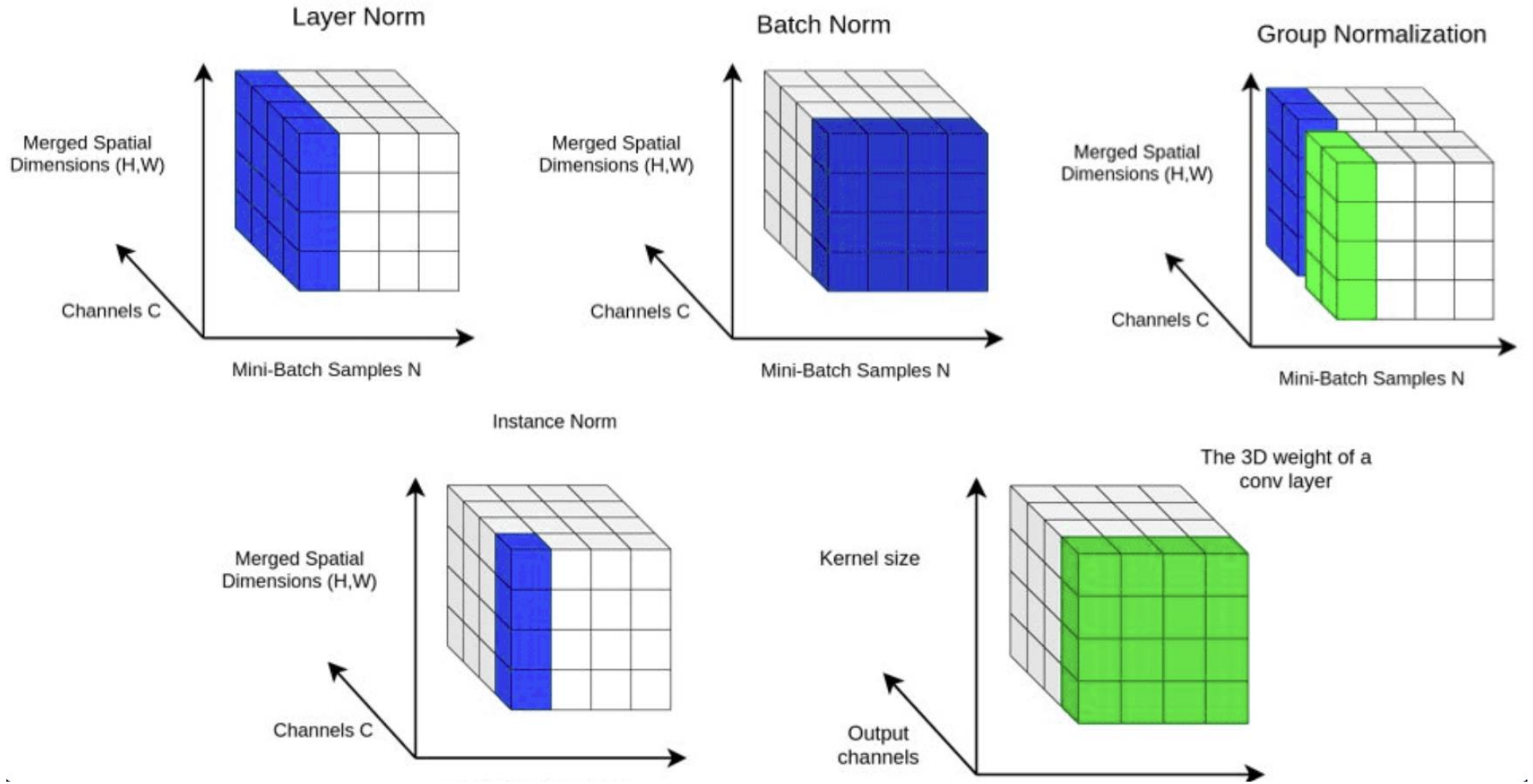
Standardize unit activations by subtracting mean and dividing by variance.

Squashes units into a **standard range**, avoiding overflow.

Also achieves invariance to mean and variance of the training signal.

Both these properties reduce the effective capacity of the model, i.e. regularize the model.

Comparisons of different normalization



https://code.likeagirl.io/different-normalization-method-in-deep-neural-networks-definition-with-tensorflow-keras-codes-fb819edc9f7a?gi=ec4bb1c8af03

BN in practice

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

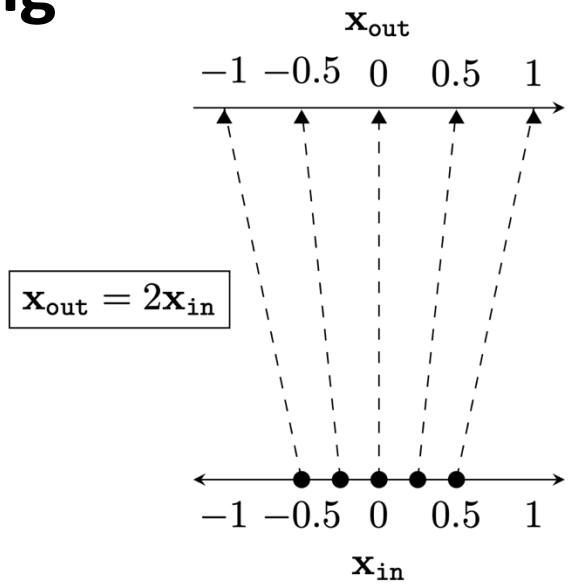
$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

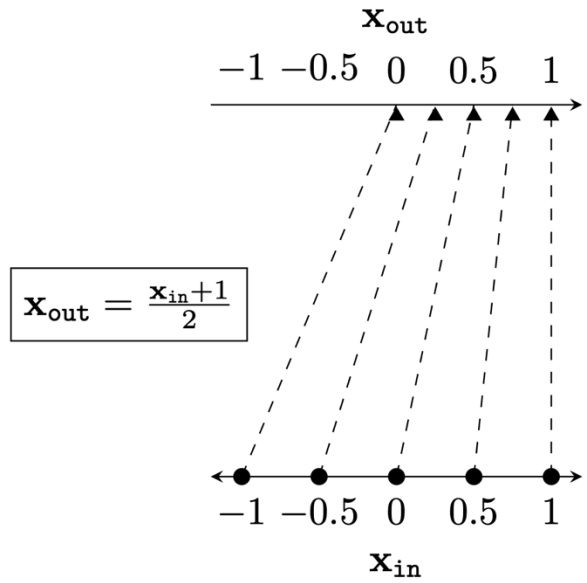
https://arxiv.org/pdf/1502.03167

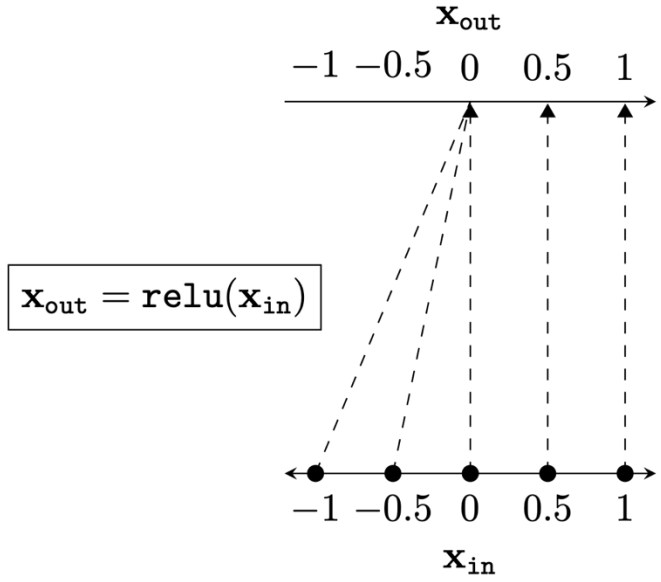
Comparisons of different normalization

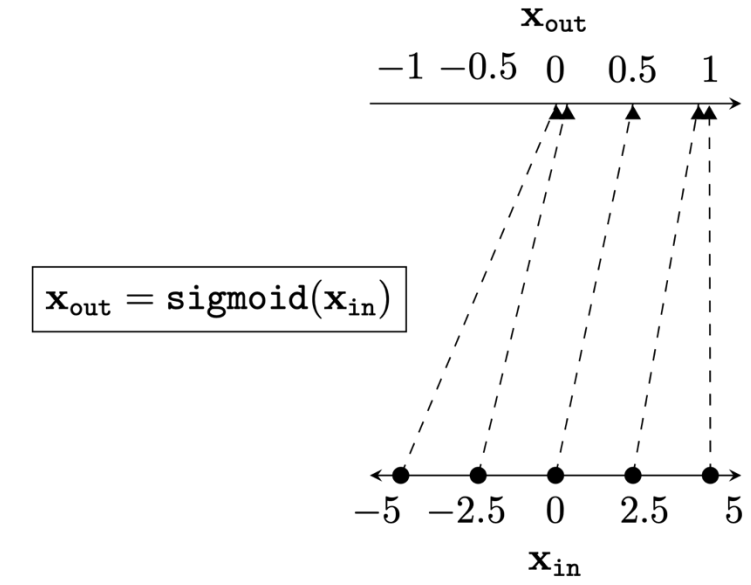
Method	Normalizes Across...	Use Case	Batch-Dependent?
Batch Norm	Batch (N)	CNNs (CV)	✓ Yes
Layer Norm	Features (C, H, W)	Transformers (NLP)	✗ No
Instance Norm	Spatial (H, W) per-channel	Style Transfer (CV)	✗ No
Group Norm	Channel Groups	Small-Batch CV	✗ No

Differentiable programming









Differentiable programming

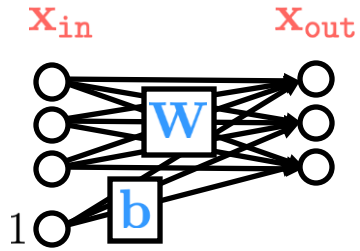
Wiring graph

Equation

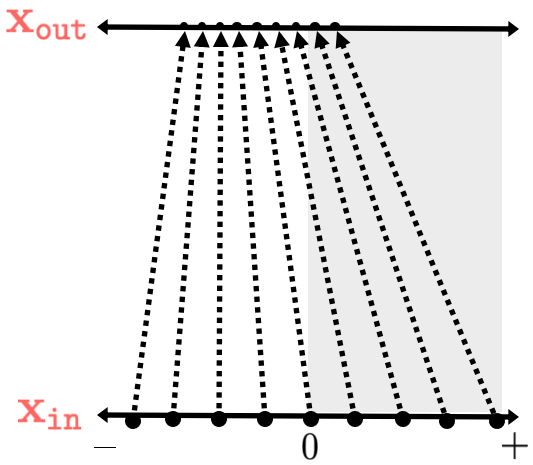
Mapping 1D

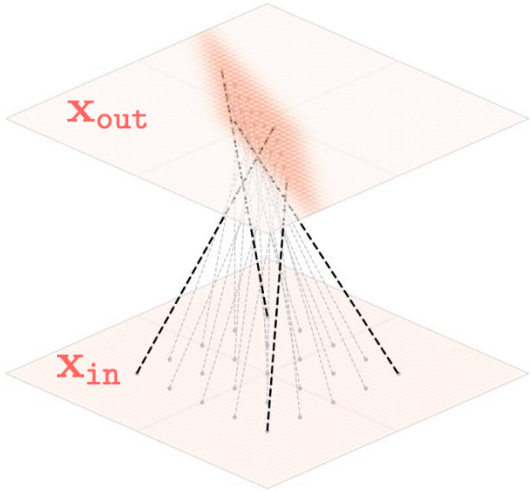
Mapping 2D

linear

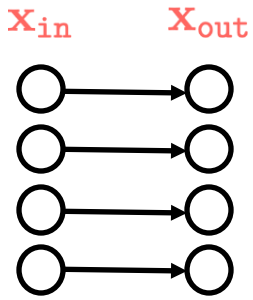


$$\mathbf{x}_{\text{out}} = \mathbf{W}\mathbf{x}_{\text{in}} + \mathbf{b}$$

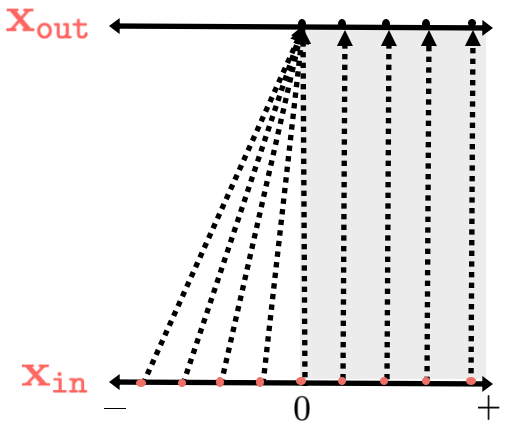


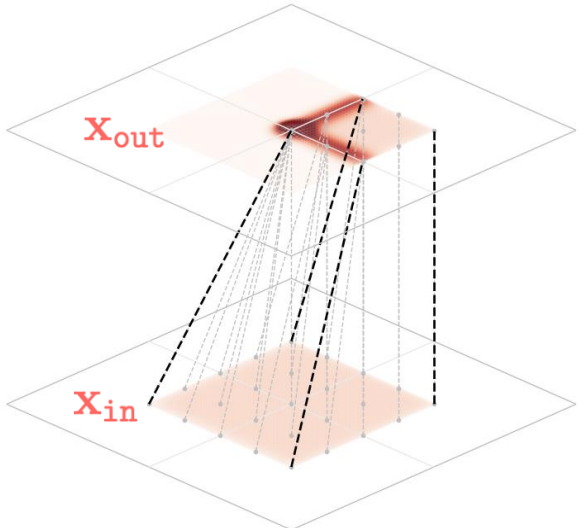


relu



$$x_{\text{out}_i} = \max(x_{\text{in}_i}, 0)$$





Differentiable programming