Lecture 08: Backprop Part 2

Data C182 (Fall 2024). Week 05. Tuesday Sept 24th, 2024

Speaker: Eric Kim

Rev01: 2024-10-02 7:45 PM PST

Announcements

- HW01: Updated deadline
  - Old: Tues Oct 1st, 11:59 PM PST
  - New: Tues Oct 8th, 11:59 PM PST
- Tip: you'll find this week's discussion (Week 05) very helpful for HW01.
- Keep asking questions on Ed and office hours

Today's lecture

- Deeper dive into the backpropagation algorithm, and how it connects to code
  - (Useful for HW01!)

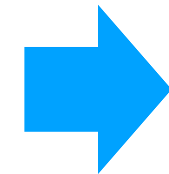
Tip: partial derivative shapes

- When you see a symbol like:

Suppose z has shape=[n], and a has shape=[m]

Tip: When working with partial derivatives (eg debugging HW01 code), it's really helpful to keep track of what all the shapes should be

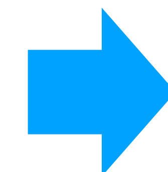
$$\frac{dz}{da}$$



Shape=[dim(a), dim(z)] => [m, n]

This is "denominator" convention, as the shape is [dim(denominator), dim(numerator)]

$$\frac{dLoss}{dz}$$



Shape=[dim(z), 1] => [m, 1]

Since our Loss is a scalar

$$\frac{dz}{dW}$$

Variant 1(flatten W matrix into a vector):
Shape=[dim(W), dim(z)] => [m*n, dim(z)]

Variant 2 (no flattening, work with tensors):
Shape=[n, m, dim(z)]

Suppose W is a matrix with shape=[n,m]

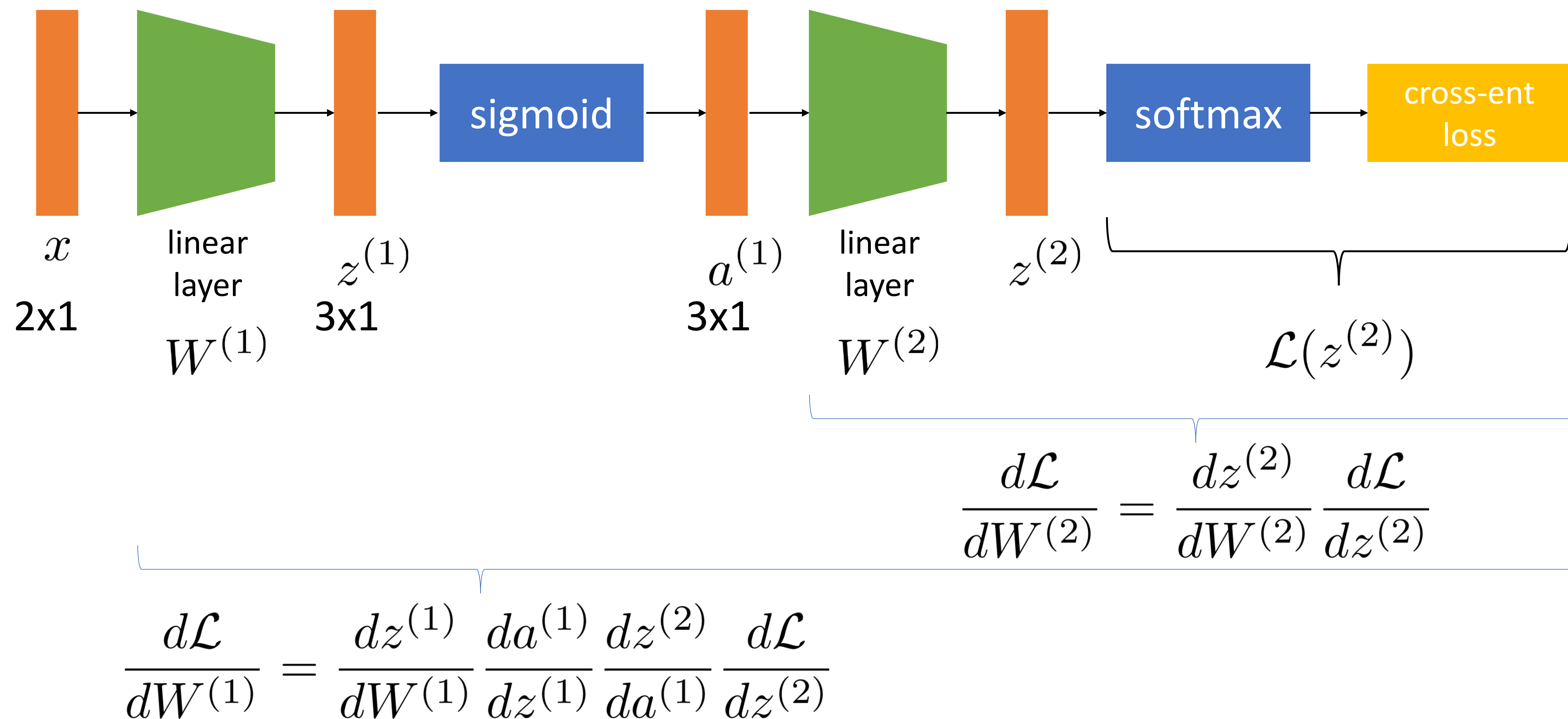
Depending on your computation, one view may be easier than another

Tip: for our backprop calculations, multiplying $\frac{dz}{dW}\frac{dL}{dz}$ means multiplying a matrix with shape=[dim(W), dim(z)] with a vector with shape=[dim(z), 1]. Eg flatten dz/dW!

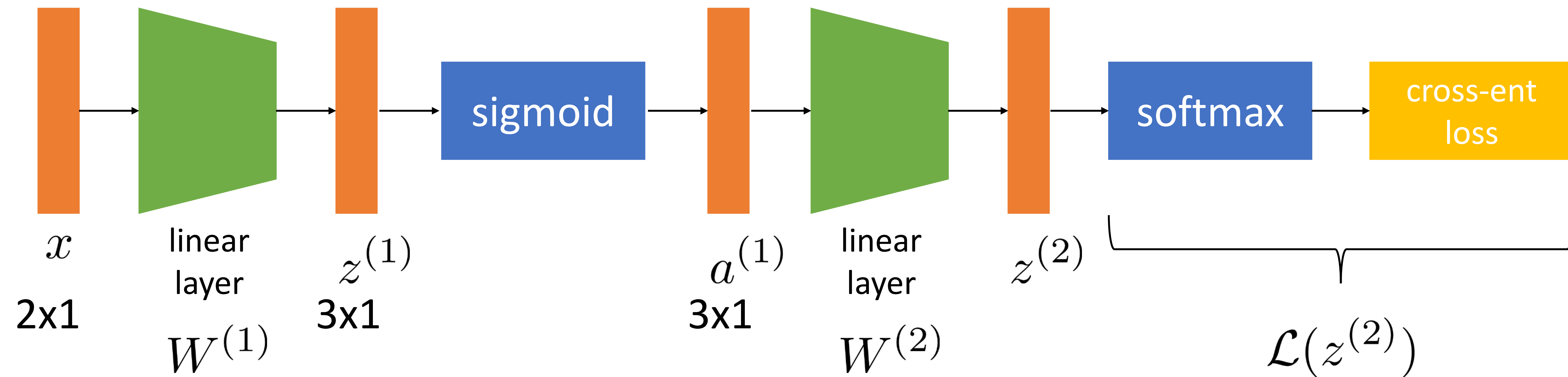
Chain rule for neural networks

A neural network is just a composition of functions

So we can use chain rule to compute gradients!



Chain rule and `backwards()`



One view: $$\frac{d\mathcal{L}}{dW^{(1)}} = \frac{dz^{(1)}}{dW^{(1)}} \frac{da^{(1)}}{dz^{(1)}} \frac{dz^{(2)}}{da^{(1)}} \frac{d\mathcal{L}}{dz^{(2)}}$$

Gradient of Linear(1) output wrt params W

Grad. of Sigmoid wrt inputs z(1)

Grad. of Linear(2) wrt inputs a(1)

Grad. of Loss wrt inputs z(2)

Aka: decompose what we want ($\frac{\partial Loss}{\partial W^{(1)}}$) into terms that are local calls to a layer's `backwards()` function (eg in HW01).

```
def layer_backwards(dout, cache):
    # Given: dout (upstream, loss w.r.t. layer's outputs)
    # Compute two things:
    # (1) Gradient of loss w.r.t. layer's inputs
    #    Tip: typically done by computing d_output_d_input
    #      and multiplying with dout
    # (2) Gradient of loss w.r.t. layer parameters (if any,
ex Relu has no trainable params)
    ...
```

Insight: rather than (laboriously) compute gradients for all trainable parameters, instead define backwards() for each of your layers that follows the above interface, and backpropagation will "do the right thing" (aka chain rule) to get you the gradients you need!

Does it work?

$$\frac{d\mathcal{L}}{dW^{(1)}} = \frac{dz^{(1)}}{dW^{(1)}} \frac{da^{(1)}}{dz^{(1)}} \frac{dz^{(2)}}{da^{(1)}} \frac{d\mathcal{L}}{dz^{(2)}}$$

Shapes: **[|W| x 1] or [m x m], if W.shape=[m,m]** [|W| x m] [m x m] [m x n_c] [n_c x 1]

(depends if you like working with tensors or matrices)
Tip: this is "denominator" convention

|W|: num elements in param matrix W
m: hidden dimensionality
n_c: num target classes (assuming classification)

We **can** calculate each of these Jacobians!

Example:

$$z^{(2)} = W^{(2)}a^{(1)}$$

$$\frac{dz^{(2)}}{da^{(1)}} = W^{(2)^T}$$

Why might this be a **bad** idea?

if each $z^{(i)}$ or $a^{(i)}$ has about n dims...

each Jacobian is about $n \times n$ dimensions

matrix multiplication is $O(n^3)$

do we care?

AlexNet has layers with 4096 units...

Doing it more efficiently

this product is cheap: $O(n^2)$

this product is expensive

$$\frac{d\mathcal{L}}{dW^{(1)}} = \frac{dz^{(1)}}{dW^{(1)}} \frac{da^{(1)}}{dz^{(1)}} \frac{dz^{(2)}}{da^{(1)}} \frac{d\mathcal{L}}{dz^{(2)}}$$

$n \times n$ $\quad n \times 1$

this is **always** true because the loss is scalar-valued!

Idea: start on the right

compute $\frac{dz^{(2)}}{da^{(1)}} \frac{d\mathcal{L}}{dz^{(2)}} = \delta$ first

$$\frac{d\mathcal{L}}{dW^{(1)}} = \frac{dz^{(1)}}{dW^{(1)}} \frac{da^{(1)}}{dz^{(1)}} \delta$$

this product is cheap: $O(n^2)$

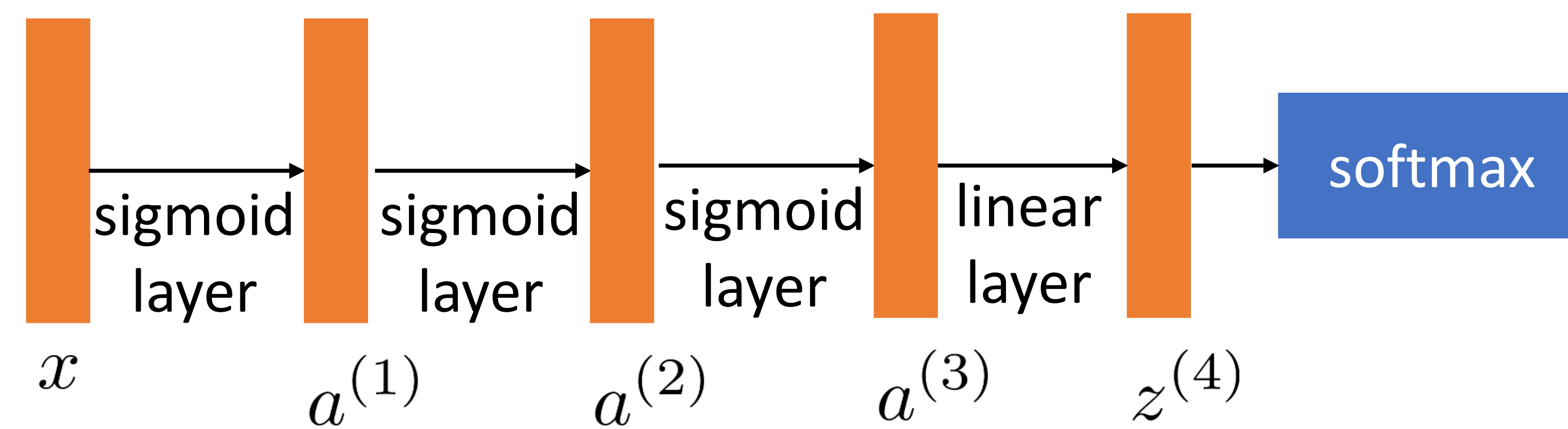
compute $\frac{da^{(1)}}{dz^{(1)}} \delta = \gamma$

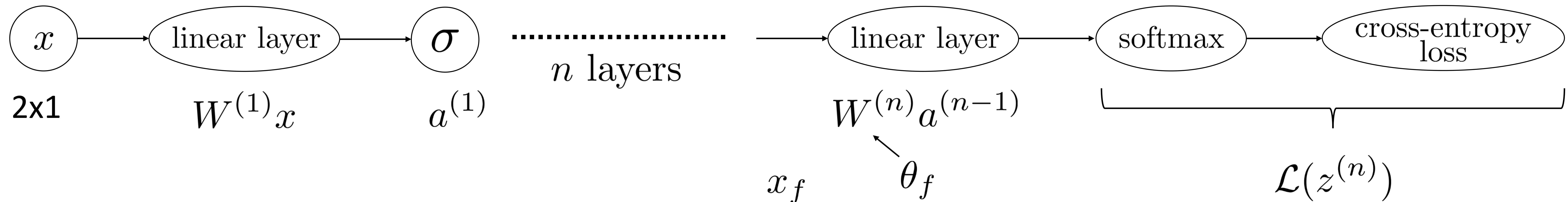
$$\frac{d\mathcal{L}}{dW^{(1)}} = \frac{dz^{(1)}}{dW^{(1)}} \gamma$$

this product is cheap: $O(n^2)$

The backpropagation algorithm

"Classic" version





$a^{(n-1)} \longrightarrow f \longrightarrow z^{(n-1)}$

forward pass: calculate each $a^{(i)}$ and $z^{(i)}$

backward pass:

initialize $\delta = \frac{d\mathcal{L}}{dz^{(n)}}$ ← Tip: δ is `dout` in HW01

for each f with input x_f & params θ_f from end to start:

$$\frac{d\mathcal{L}}{d\theta_f} \leftarrow \frac{df}{d\theta_f}\delta$$

$\frac{df}{d\theta_f}$: Deriv. of layer output w.r.t. layer parameters θ_f (if any)

$$\delta \leftarrow \frac{df}{dx_f}\delta$$

$\frac{df}{dx_f}$: Deriv. of layer output w.r.t. layer inputs x_f

$\times$ $=$

Backprop (pseudocode)

```
from typing import Dict
import numpy as np

def backprop_pseudocode(
        model, x: np.ndarray,
        targets: np.ndarray) -> Tuple[float, Dict[int, np.ndarray]]:
    #### Forwards ####
    forward_outs = []
    cur_x = x
    # For simplicity: assume model is just a single path, no branches
    for layer in model.layers:
        out, cache = layer.forward(cur_x)
        forward_outs.append((out, cache))
        cur_x = out
    # Assume that final cur_x is the logits
    logits = cur_x
    loss, d_loss_d_logits = model.softmax_loss(logits, targets)

    #### Backwards ####
    d_loss_d_cur_outputs = d_loss_d_logits
    grads = {}  # map layer index to their d_loss_d_param gradients
    # Begin: backpropagation algorithm
    # iterate in reverse layer order (starting at layer before loss)
    for i, layer in enumerate(model.layers[::-1]):
        ind_layer = len(model.layers) - i - 1
        cache = forward_outs[ind_layer][1]
        d_loss_d_cur_inputs, d_loss_d_params = layer.backward(
            d_loss_d_cur_outputs, cache)
        grads[ind_layer] = d_loss_d_params
        # current layer's inputs becomes outputs of the previous layer
        d_loss_d_cur_outputs = d_loss_d_cur_inputs
    return loss, grads
```

Connecting the previous math equations to our code

d_loss_d_cur_outputs   d_loss_d_logits

$$\text{initialize } \delta = \frac{d\mathcal{L}}{dz^{(n)}}$$

$$\frac{d\mathcal{L}}{d\theta_f} \leftarrow \frac{df}{d\theta_f}\delta$$

d_loss_d_params

layer.backward() computes this product

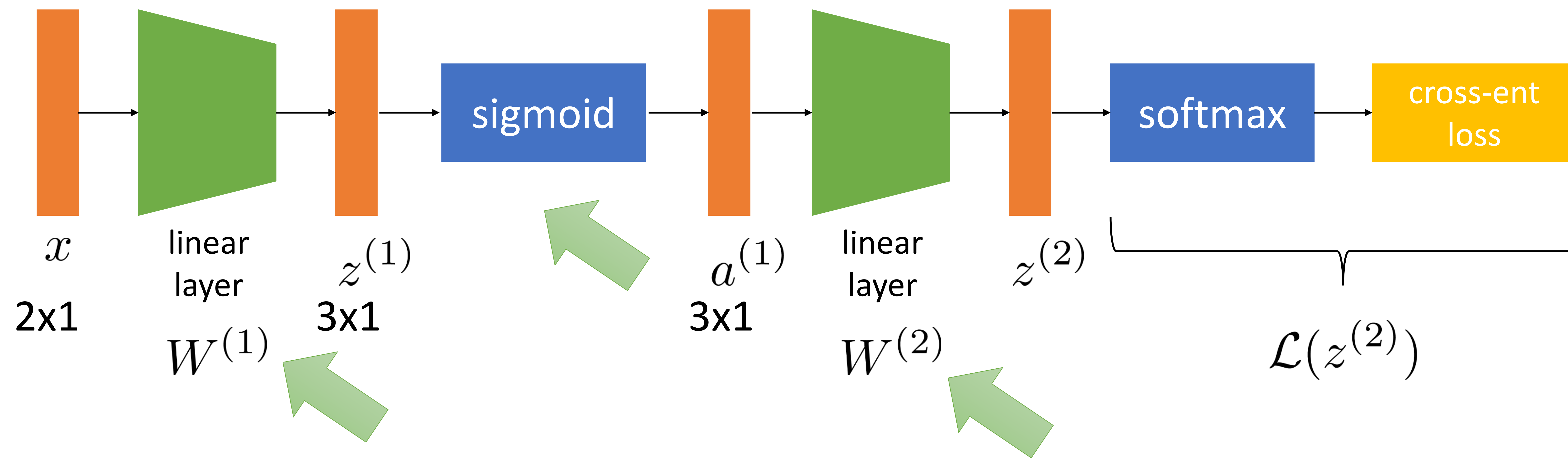
d_loss_d_cur_outputs (pre overwrite)

$$\delta \leftarrow \frac{df}{dx_f}\delta$$

d_loss_d_cur_outputs

layer.backward() computes this product: d_loss_d_cur_inputs

Let's walk through it...



$$\frac{d\mathcal{L}}{dW^{(2)}} = \underbrace{\frac{dz^{(2)}}{dW^{(2)}} \frac{d\mathcal{L}}{dz^{(2)}}}_{\delta}$$

$$\frac{d\mathcal{L}}{dW^{(1)}} = \underbrace{\frac{dz^{(1)}}{dW^{(1)}} \frac{da^{(1)}}{dz^{(1)}} \frac{dz^{(2)}}{da^{(1)}} \frac{d\mathcal{L}}{dz^{(2)}}}_{\delta}$$

forward pass: calculate each $a^{(i)}$ and $z^{(i)}$

backward pass:

initialize $\delta = \frac{d\mathcal{L}}{dz^{(n)}}$

for each f with input x_f & params θ_f from end to start:

$$\frac{d\mathcal{L}}{d\theta_f} \leftarrow \frac{df}{d\theta_f}\delta$$

$$\delta \leftarrow \frac{df}{dx_f}\delta$$

Calling linear_backwards() gives us: $\frac{df}{d\theta_f}$ (aka $\frac{dz^{(2)}}{dW^{(1)}}, \frac{dz^{(2)}}{db^{(1)}}$), and $\frac{df}{dx_f}$ (aka $\frac{dz^{(2)}}{da^{(1)}}$)

Building the model + backprop: in code

Network Architecture: Linear -> Relu -> Linear -> SoftMax -> CrossEntropyLoss

Model parameters ("trainable parameters"):
Question: what are the model parameters?
Answer: Each Linear layer has their own weight, bias parameters.
Thus, with two Linear layers, we have two Weight matrices, and two bias vectors.

Suppose we have d input features, n_c target classes, and we want the intermediate activations ("hidden dim") to have dimensionality h.
Question: what are the shapes of each Linear layer's weight, bias?
Answer:
Linear(1): Weight.shape=[d, h], bias.shape=[h]
Linear(2): Weight.shape[h, n_c], bias.shape=[n_c]

```
# Tip: randomly init weight via normal dist (0 mean, 1.0 var)
# Init bias to be all 0's
weight1 = np.random.normal(0, 1.0, size=(d, h))
bias1 = np.zeros(h)

weight2 = np.random.normal(0, 1.0, size=(h, n_c))
bias2 = np.zeros(n_c)

# x.shape=[1, d], weight1.shape=[d, h], bias1.shape=[h]
# matmult-ing a [1, d] with a [d, h] => [1, h]
# => out1.shape=[1, h]
out1 = x.dot(weight1) + bias1

# weight2.shape=[h, n_c], bias2.shape=[n_c]
# matmult-ing a [1, h] with a [h, n_c] => [1, n_c]
# => out2.shape=[1, n_c]
out2 = out1.dot(weight2) + bias2
```

Forward, backwards (in code)

Network Architecture: Linear -> Relu -> Linear -> SoftMax -> CrossEntropyLoss

```
# x.shape=[batchsize, d]
# target_classe.shape=[batchsize]
# Forward
out1, affine1_cache = affine_forward(x, weights1, bias1)
out1, relu_cache = relu_forward(out1)
logits, affine2_cache = affine_forward(out1, weights2, bias2)

# Backwards
loss, d_loss_d_logits = softmax_loss(logits, target_classes)
d_loss_d_affine2_inputs, d_loss_d_w2, d_loss_d_b2 = affine_backward(
    d_loss_d_logits, affine2_cache)
d_loss_d_relu_inputs = relu_backward(d_loss_d_affine2_inputs,
relu_cache)
d_loss_d_affine1_inputs, d_loss_d_w1, d_loss_d_b1 = affine_backward(
    d_loss_d_relu_inputs, affine1_cache)

grads = {
    'W1': d_loss_d_w1, 'b1': d_loss_d_b1,
    'W2': d_loss_d_w2, 'b2': d_loss_d_b2,
}
```

Solving for $\frac{dLoss}{dW^{(2)}}$

forward pass: calculate each $a^{(i)}$ and $z^{(i)}$

backward pass:

initialize $\delta = \frac{d\mathcal{L}}{dz^{(n)}}$ *d_loss_d_logits*

for each f with input x_f & params θ_f from end to start:

$$\frac{d\mathcal{L}}{d\theta_f} \leftarrow \frac{df}{d\theta_f}\delta$$

$\frac{df}{d\theta_f}$: Deriv. of layer output w.r.t. layer parameters (if any)

$$\delta \leftarrow \frac{df}{dx_f}\delta$$

$\frac{df}{dx_f}$: Deriv. of layer output w.r.t. layer input

d_loss_d_logits (\`dout\` in affine_backward())

$$\frac{d\mathcal{L}}{dW^{(2)}} = \frac{dz^{(2)}}{dW^{(2)}}\frac{d\mathcal{L}}{dz^{(2)}}$$

Shape: [h, n_c]

Product is computed in affine_backward()

```
# Backwards
loss, d_loss_d_logits = softmax_loss(logits, target_classes)
d_loss_d_affine2_inputs, d_loss_d_w2, d_loss_d_b2 = affine_backward(
    d_loss_d_logits, affine2_cache)
d_loss_d_relu_inputs = relu_backward(d_loss_d_affine2_inputs, relu_cache)
d_loss_d_affine1_inputs, d_loss_d_w1, d_loss_d_b1 = affine_backward(
    d_loss_d_relu_inputs, affine1_cache)
```

Solving for $\frac{dLoss}{dW^{(1)}}$

forward pass: calculate each $a^{(i)}$ and $z^{(i)}$

backward pass:

initialize $\delta = \frac{d\mathcal{L}}{dz^{(n)}}$ *d_loss_d_logits*

for each f with input x_f & params θ_f from end to start:

$$\frac{d\mathcal{L}}{d\theta_f} \leftarrow \frac{df}{d\theta_f}\delta$$

$\frac{df}{d\theta_f}$: Deriv. of layer output w.r.t. layer parameters (if any)

$$\delta \leftarrow \frac{df}{dx_f}\delta$$

$\frac{df}{dx_f}$: Deriv. of layer output w.r.t. layer input

$$\frac{d\mathcal{L}}{dW^{(1)}} = \frac{dz^{(1)}}{dW^{(1)}}\frac{da^{(1)}}{dz^{(1)}}\frac{dz^{(2)}}{da^{(1)}}\frac{d\mathcal{L}}{dz^{(2)}}$$

`d_loss_d_logits`

`d_loss_d_affine2_inputs`

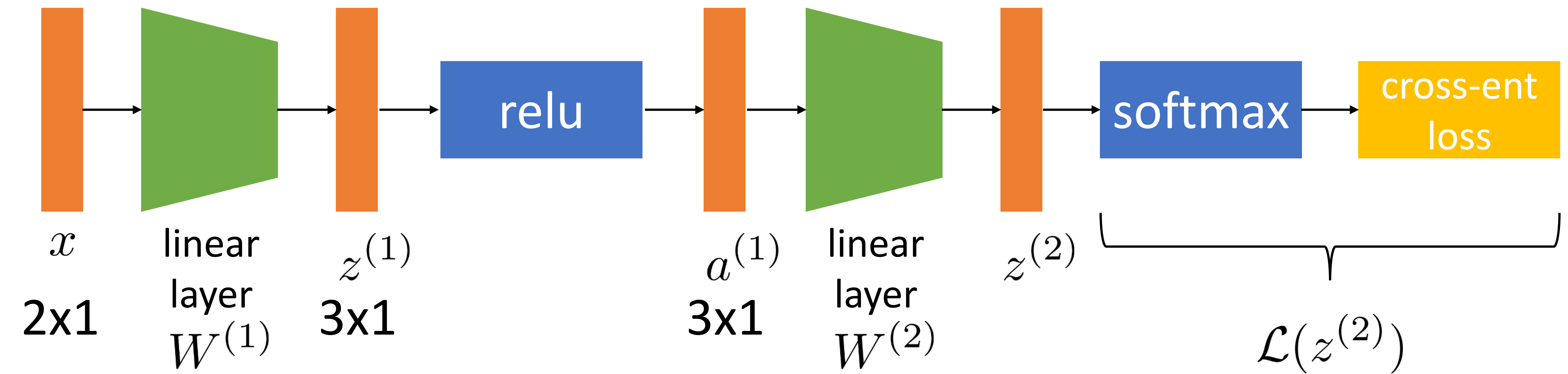
`d_loss_d_relu_inputs`

`d_loss_d_w1`

```
# Backwards
loss, d_loss_d_logits = softmax_loss(logits, target_classes)
d_loss_d_affine2_inputs, d_loss_d_w2, d_loss_d_b2 = affine_backward(
    d_loss_d_logits, affine2_cache)
d_loss_d_relu_inputs = relu_backward(d_loss_d_affine2_inputs, relu_cache)
d_loss_d_affine1_inputs, d_loss_d_w1, d_loss_d_b1 = affine_backward(
    d_loss_d_relu_inputs, affine1_cache)
```

https://simpsons.fandom.com/wiki/Woo-hoo!?file=Woo_hoo%21_poster.jpg

Aside: what are these terms?



`d_loss_d_logits`

$$\frac{d\mathcal{L}}{dW^{(1)}} = \frac{dz^{(1)}}{dW^{(1)}} \frac{da^{(1)}}{dz^{(1)}} \frac{dz^{(2)}}{da^{(1)}} \frac{d\mathcal{L}}{dz^{(2)}}$$

`d_loss_d_affine2_inputs`

`d_loss_d_relu_inputs`

`d_loss_d_w1`

$$\frac{dLoss}{da^{(1)}} = \frac{dz^{(2)}}{da^{(1)}} \frac{dLoss}{dz^{(2)}}$$ (By Chain Rule)

$$\frac{dLoss}{dz^{(1)}} = \frac{da^{(1)}}{dz^{(1)}} \frac{dz^{(2)}}{da^{(1)}} \frac{dLoss}{dz^{(2)}}$$ (By Chain Rule)

$$\frac{dLoss}{dW^{(1)}} = \frac{dz^{(1)}}{dW^{(1)}} \frac{da^{(1)}}{dz^{(1)}} \frac{dz^{(2)}}{da^{(1)}} \frac{dLoss}{dz^{(2)}}$$ (By Chain Rule)

Intuition: This (backprop) works so nicely because
(1) Chain rule naturally decomposes into terms for each layer, and
(2) We can easily (and efficiently!) compute the gradients of each layer with respect to their input(s) and their layer parameters (if any)

Important caveat: your layers must be differentiable w.r.t. inputs and parameters! If not, then you can't train with them (aka "backprop through" them).
Aside: sometimes if your function is "almost" differentiable (eg Relu at 0), you can get away with using subgradients.

Building the model + backprop: in code

forward pass: calculate each $a^{(i)}$ and $z^{(i)}$

backward pass:

initialize $\delta = \frac{d\mathcal{L}}{dz^{(n)}}$ *d_loss_d_logits*

for each f with input x_f & params θ_f from end to start:

$$\frac{d\mathcal{L}}{d\theta_f} \leftarrow \frac{df}{d\theta_f}\delta$$

$$\delta \leftarrow \frac{df}{dx_f}\delta$$

$\frac{df}{d\theta_f}$: Deriv. of layer output w.r.t. layer parameters (if any)

$\frac{df}{dx_f}$: Deriv. of layer output w.r.t. layer input

$$\frac{d\mathcal{L}}{dW^{(1)}} = \frac{dz^{(1)}}{dW^{(1)}} \frac{da^{(1)}}{dz^{(1)}} \frac{dz^{(2)}}{da^{(1)}} \frac{d\mathcal{L}}{dz^{(2)}}$$

`d_loss_d_logits`

`d_loss_d_affine2_inputs`

`d_loss_d_relu_inputs`

`d_loss_d_w1`

```
# Backwards
loss, d_loss_d_logits = softmax_loss(logits, target_classes)
d_loss_d_affine2_inputs, d_loss_d_w2, d_loss_d_b2 = affine_backward(
    d_loss_d_logits, affine2_cache)
d_loss_d_relu_inputs = relu_backward(d_loss_d_affine2_inputs, relu_cache)
d_loss_d_affine1_inputs, d_loss_d_w1, d_loss_d_b1 = affine_backward(
    d_loss_d_relu_inputs, affine1_cache)
```

Note how there is a "chain" that links each backwards() call, starting with `d_loss_d_logits`. This matches up with the following update equation and the above right-to-left reduction:

$$\delta \leftarrow \frac{df}{dx_f}\delta$$

What is this *_cache?

```
# x.shape=[batchsize, d]
# target_classes.shape=[batchsize]
# Forward
out1, affine1_cache = affine_forward(x, weights1, bias1)
out1, relu_cache = relu_forward(out1)
logits, affine2_cache = affine_forward(out1, weights2, bias2)

# Backwards
loss, d_loss_d_logits = softmax_loss(logits, target_classes)
d_affine2_d_inputs, d_loss_d_w2, d_loss_d_b2 = affine_backward(
    d_loss_d_logits, affine2_cache)
d_relu_d_inputs = relu_backward(d_affine2_d_inputs, relu_cache)
d_affine1_d_inputs, d_loss_d_w1, d_loss_d_b1 = affine_backward(
    d_relu_d_inputs, affine1_cache)
```

Implementation detail: many *_backwards() functions needs to know information from the *_forwards() function.

Ex: affine_backwards() needs to know the original input to affine_forwards() to calculate d_loss_d_w, etc.

In HW01, the *_forwards()/*_backwards() functions will make it clear what you need to put in the caches.

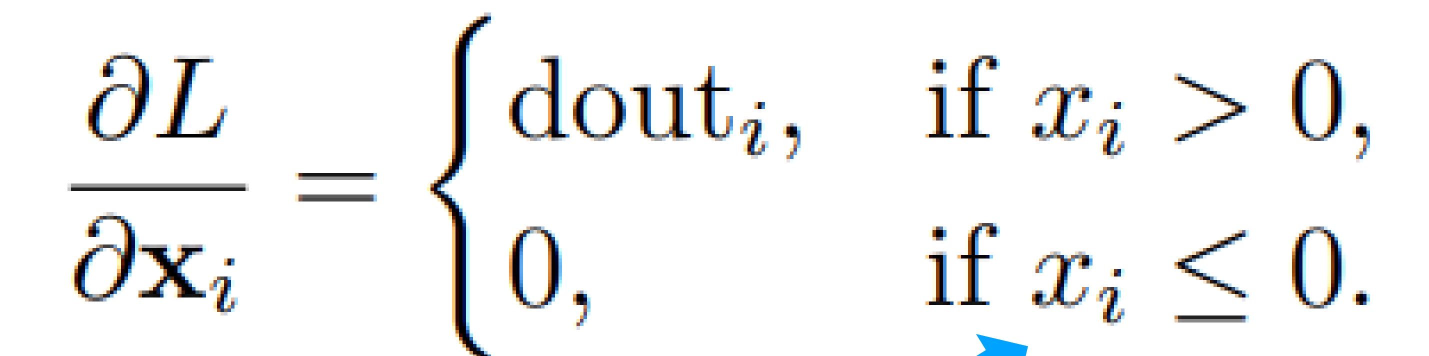
Example: relu cache

Relu forward:

The ReLU function is piecewise-defined:

$$\mathrm{ReLU}(x) = \begin{cases} x, & \text{if } x > 0, \\ 0, & \text{if } x \leq 0. \end{cases}$$

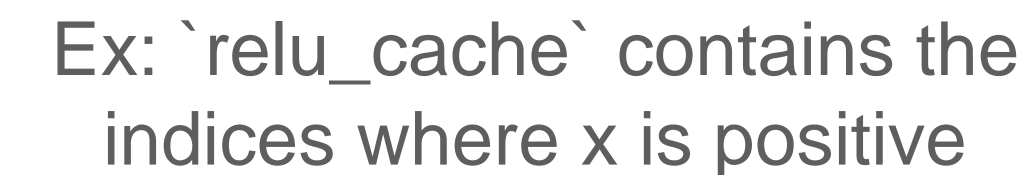
Relu backward:



$$\frac{\partial L}{\partial \mathbf{x}_i} = \begin{cases} \mathrm{dout}_i, & \text{if } x_i > 0, \\ 0, & \text{if } x_i \leq 0. \end{cases}$$

To compute the gradient, relu_backward() needs to know the indices where the relu_forward(x) inputs were positive.

Implementation Idea: let's have relu_forward(x) emit not only the relu output, but also any additional info that relu_backwards() may need, aka a "cache":




```
relu_out, relu_cache = relu_forward(x)
...
bkwd_out = relu_backward(dout, relu_cache)
```

Example: relu cache

Relu forward:

The ReLU function is piecewise-defined:

$$\text{ReLU}(x) = \begin{cases} x, & \text{if } x > 0, \\ 0, & \text{if } x \leq 0. \end{cases}$$

Relu backward:

$$\frac{\partial L}{\partial \mathbf{x}_i} = \begin{cases} \text{dout}_i, & \text{if } x_i > 0, \\ 0, & \text{if } x_i \leq 0. \end{cases}$$

```
def relu_forward(x: np.ndarray) -> Tuple[np.ndarray, Dict[str, Any]]:
    # mask_positive: shape=x.shape, True/False entries
    mask_positive = x > 0
    out = np.copy(x)
    out[~mask_positive] = 0
    cache = {"x_mask_positive": mask_positive}
    return out, cache
```

Store positive indices of input x...

```
def relu_backward(dout: np.ndarray, cache: Dict[str, Any]) -> np.ndarray:
    x_mask_positive = cache["x_mask_positive"]
    # np.where(...) turns the bool mask into an int 0/1 mask
    dx = dout * np.where(x_mask_positive, 1, 0)
    return dx
```

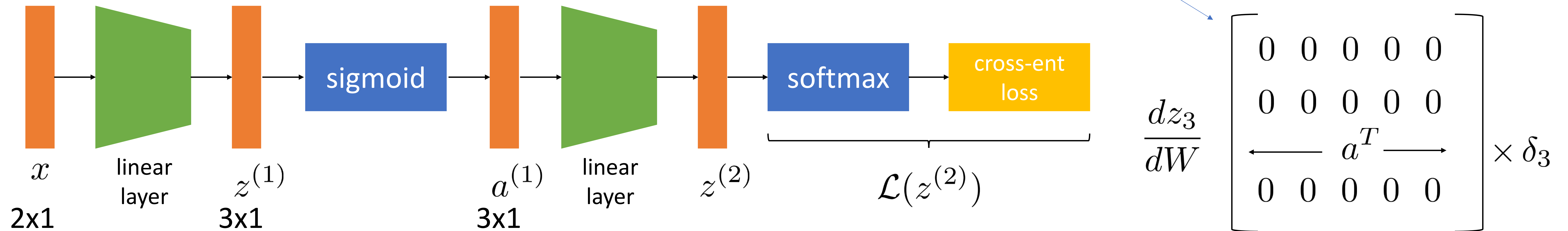
...so that backwards() can use it!

Note: In HW01, the cache is implemented as a tuple, not as a dict, but the same idea holds

Common layer recipies

- Recall that, when defining a new NN layer to use in backpropagation, we must compute the loss with respect to (1) the layer's inputs, and (2) the layer's trainable parameters (if any)
- Here are steps on how to derive (1) and (2) for some common layers
- Some general tips on doing this yourself
  - Try writing out the scalar terms, and see if you can identify a pattern that lets you "generalize" to the final vector (or matrix!)
  - Sanity check matrix/vector shapes to make sure things are right!

Backpropagation recipes: linear layer



Aka: dz_dw[:, :, 2], shape=[m, k]

$$\frac{dz_3}{dW} \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ \longleftarrow & & a^T & & \longrightarrow \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \times \delta_3$$

for each function, we need to compute: $\frac{df}{d\theta_f}\delta \quad \frac{df}{dx_f}\delta$

linear layer: $z^{(i+1)} = W^{(i)}a^{(i)} + b^{(i)}$

$z = Wa + b$ (just to simplify notation!)

x_f, θ_f

[m x k]

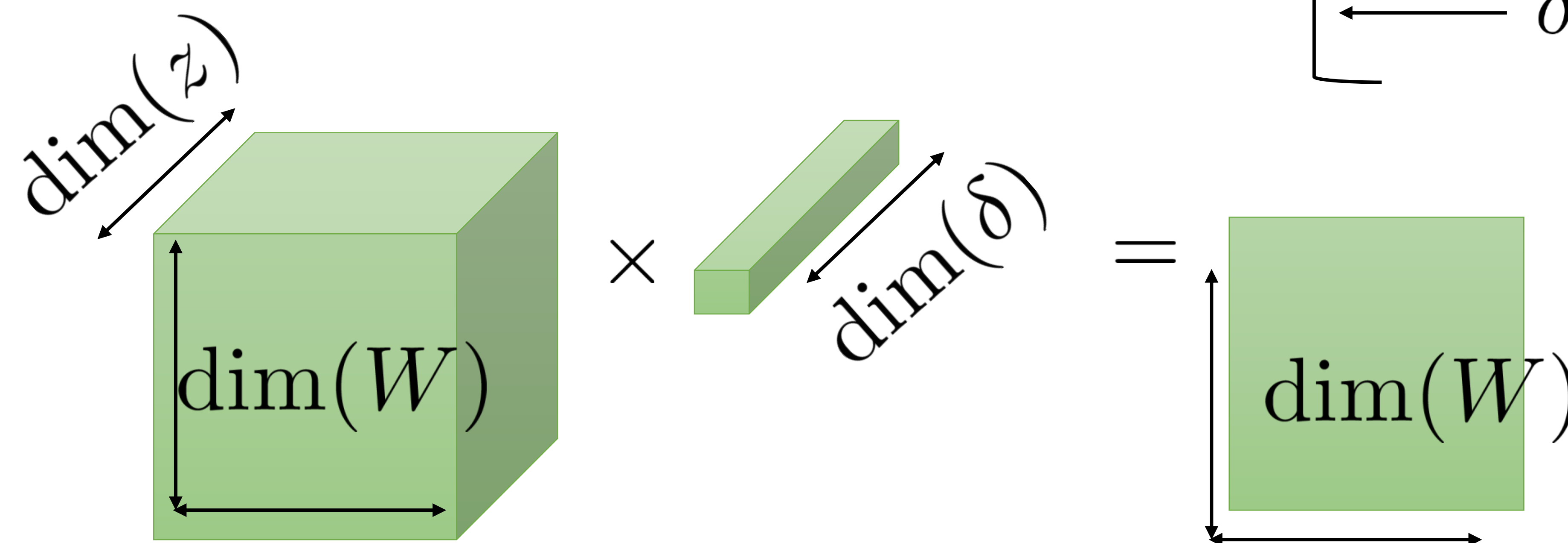
Neat trick: much easier to compute this than explicitly forming $\frac{dz}{dW}$

$$\frac{dz}{dW}\delta = \sum_i \frac{dz_i}{dW}\delta_i = \delta a^T$$

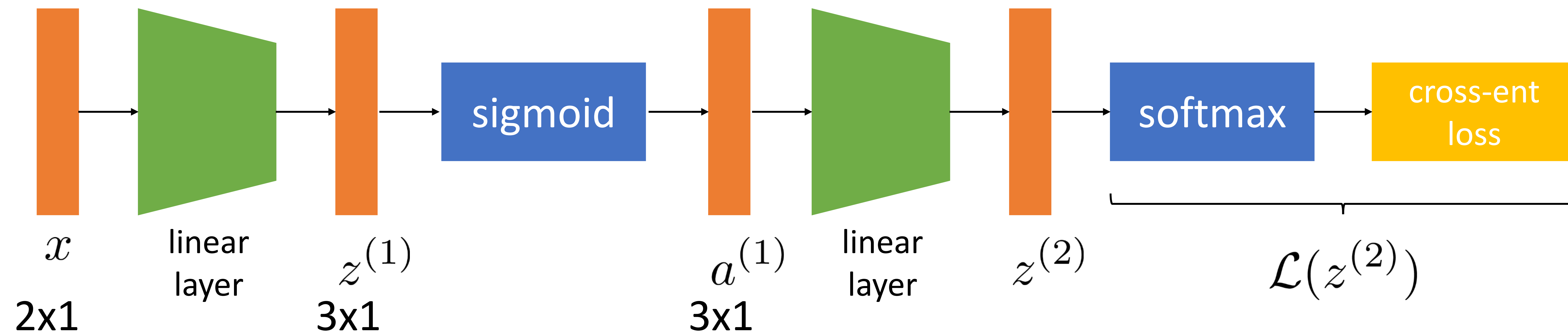
[m x k x dim(z)] [dim(z) x 1]

$$\begin{bmatrix} \longleftarrow & \delta_1 a^T & \longrightarrow \\ \longleftarrow & \delta_2 a^T & \longrightarrow \\ \longleftarrow & \delta_3 a^T & \longrightarrow \\ \longleftarrow & \delta_4 a^T & \longrightarrow \end{bmatrix}$$

$$z_i = \sum_k W_{ik} a_k + b_i \qquad \frac{dz_i}{dW_{jk}} = \begin{cases} 0 \text{ if } j \neq i \\ a_k \text{ otherwise} \end{cases}$$



Backpropagation recipes: linear layer



for each function, we need to compute: $\frac{df}{d\theta_f}\delta \quad \frac{df}{dx_f}\delta$

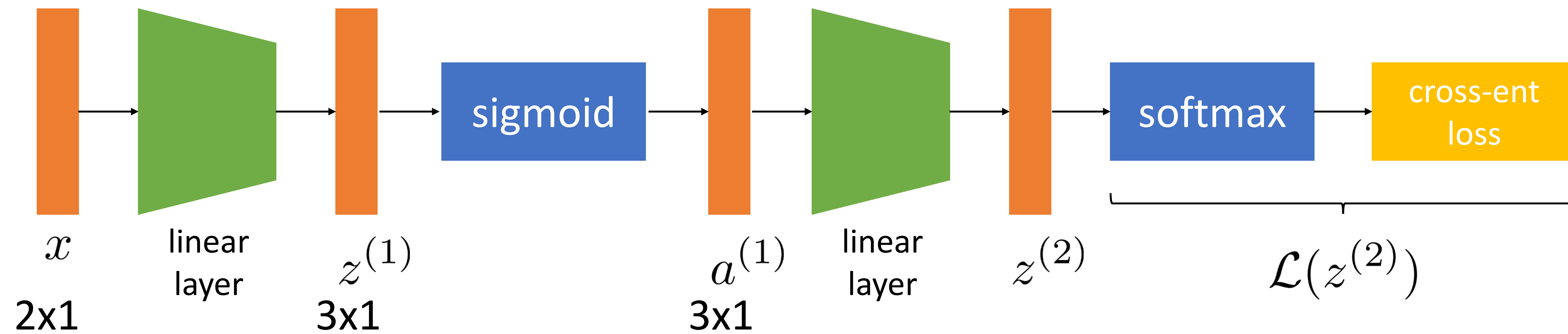
linear layer: $z^{(i+1)} = W^{(i)}a^{(i)} + b^{(i)} \qquad z = Wa + b$ (just to simplify notation!)

$$\frac{dz}{db}\delta = \delta$$

$$z_i = \sum_k W_{ik}a_k + b_i \quad \frac{dz_i}{db_j} = \text{Ind}(i = j) \quad \frac{dz}{db} = \mathbf{I}$$

Interpretation: if I change my bias b_i term by +1, then I expect the output z_i to also change by +1.

Backpropagation recipes: linear layer



for each function, we need to compute: $\frac{df}{d\theta_f}\delta \quad \frac{df}{dx_f}\delta$

linear layer: $z^{(i+1)} = W^{(i)}a^{(i)} + b^{(i)} \qquad z = Wa + b$ (just to simplify notation!)

W.shape=[dim(z), dim(a)]

$$\frac{dz}{da}\delta = W^T\delta$$

[dim(a) x dim(z)] [dim(z) x 1]

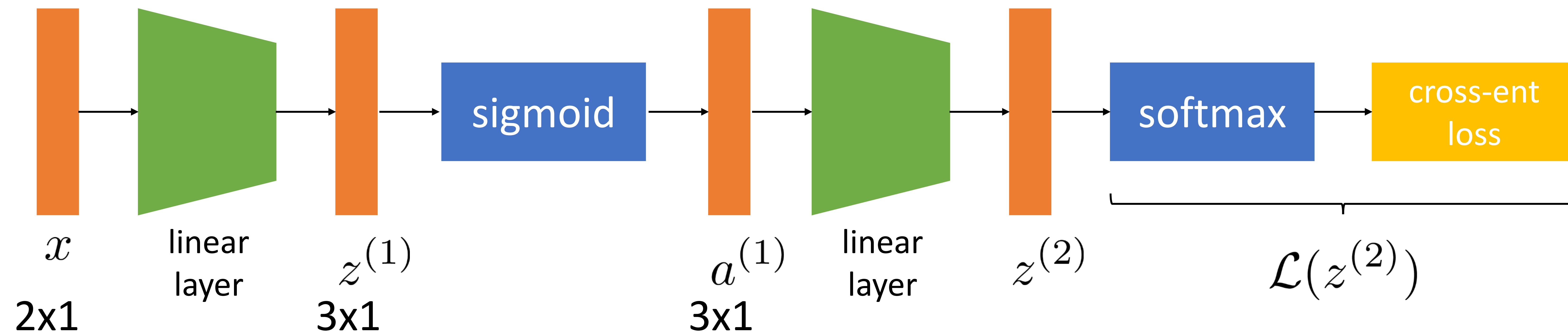
$$z_i = \sum_k W_{ik}a_k + b_i \qquad \frac{dz_i}{da_k} = W_{ik} \qquad \frac{dz}{da} = W^T$$

(1)

$$\left(\frac{dy}{dx}\right)_{ij} = \frac{dy_j}{dx_i}$$

In more detail:
(Using "denominator" format)
Let dz_da.shape=[dim(a), dim(z)]
dz_da[k, :] = W[:, k] # by (1)
Aka: k-th row of dz_da is the k-th column of W
=> dz_da is W.T

Backpropagation recipes: linear layer

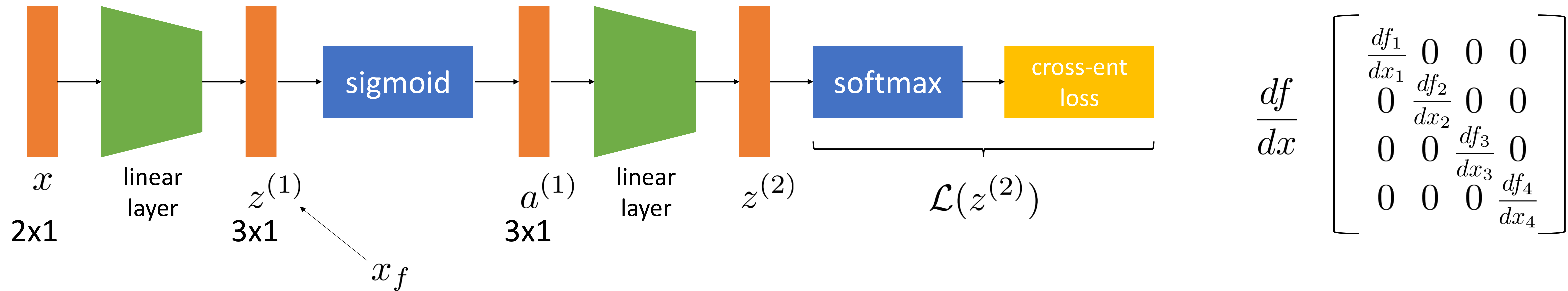


for each function, we need to compute: $\frac{df}{d\theta_f}\delta \quad \frac{df}{dx_f}\delta$

linear layer: $z^{(i+1)} = W^{(i)} a^{(i)} + b^{(i)}$ $\quad z = Wa + b$ (just to simplify notation!)

$$\underbrace{\frac{dz}{da}\delta = W^T \delta}_{\frac{df}{dx_f}\delta} \qquad \underbrace{\frac{dz}{dW}\delta = \delta a^T \qquad \frac{dz}{db}\delta = \delta}_{\frac{df}{d\theta_f}\delta}$$

Backpropagation recipes: sigmoid



for each function, we need to compute: $\frac{df}{d\theta_f}\delta \quad \frac{df}{dx_f}\delta$

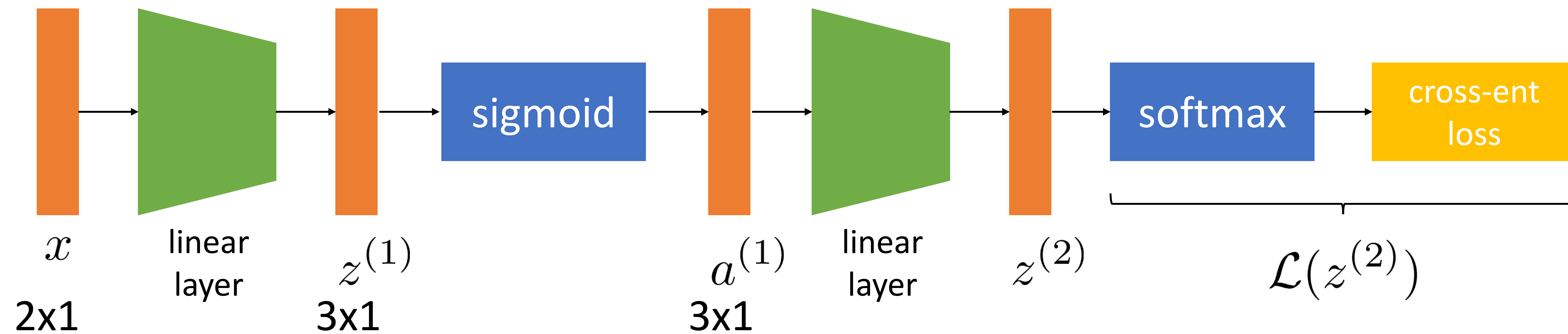
$$\sigma(z_i) = \frac{1}{1+\exp(-z_i)}$$

$$\frac{df_i}{dz_i} = \underbrace{\frac{\exp(-z_i)}{1+\exp(-z_i)}}_{} \underbrace{\frac{1}{1+\exp(-z_i)}}_{\sigma(z_i)} = (1-\sigma(z_i))\sigma(z_i)$$

$$\underbrace{\frac{1+\exp(-z_i)}{1+\exp(-z_i)} - \frac{1}{1+\exp(-z_i)}}_{1-\sigma(z_i)}$$

$$\left(\frac{df}{dz}\delta\right)_i = (1-\sigma(z_i))\sigma(z_i)\delta_i$$

Backpropagation recipes: ReLU



for each function, we need to compute: $\frac{df}{d\theta_f}\delta \quad \frac{df}{dx_f}\delta$

$$f_i(z_i) = \max(0, z_i) \qquad \frac{df_i}{dz_i} = \text{Ind}(z_i \geq 0)$$

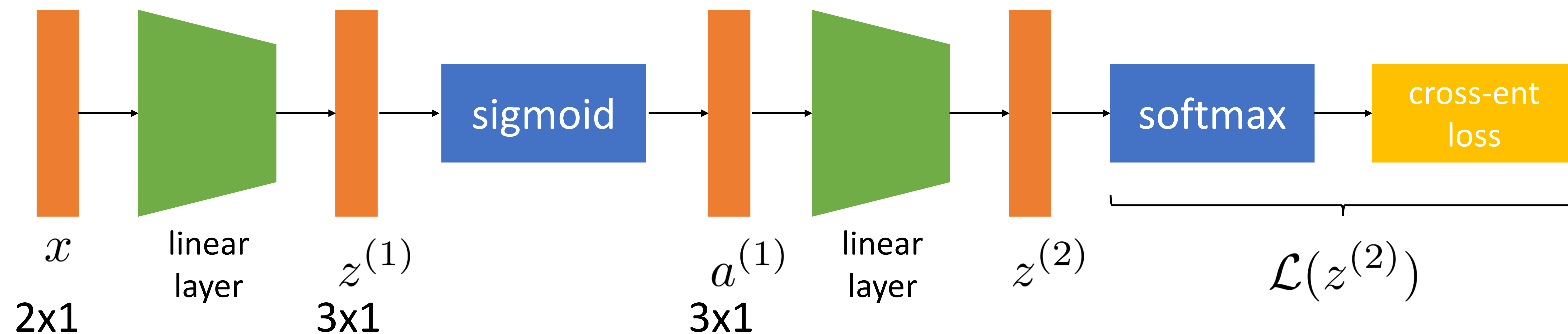
$$\left(\frac{df}{dz}\delta\right)_i = \text{Ind}(z_i \geq 0)\delta_i$$

Interpretation: only the **positive entries of the Relu input** contribute to the gradient.

Question: suppose during a training iteration all of the outputs of this Relu are 0 (eg all Relu inputs were negative). What happens during this training iteration?
Answer: All of the layers preceding this Relu will have 0 gradients, meaning that their trainable parameters will not get updated at all.

Question: can the model ever hope to "recover" from this dead Relu? What are some migitations we can do to avoid this "dead neuron" issue?
Answer: Hope that other batches will "kick" the dead neuron out of the dead negative region. Mitigations: leaky Relu, Gelu, careful weight initialization

Summary



forward pass: calculate each $a^{(i)}$ and $z^{(i)}$

backward pass:

initialize $\delta = \frac{d\mathcal{L}}{dz^{(n)}}$

for each f with input x_f & params θ_f from end to start:

$$\frac{d\mathcal{L}}{d\theta_f} \leftarrow \frac{df}{d\theta_f}\delta$$

$$\delta \leftarrow \frac{df}{dx_f}\delta$$

for each function, we need to compute:

$$\frac{df}{d\theta_f}\delta \qquad \frac{df}{dx_f}\delta$$

linear layer

softmax + cross-entropy

sigmoid

ReLU