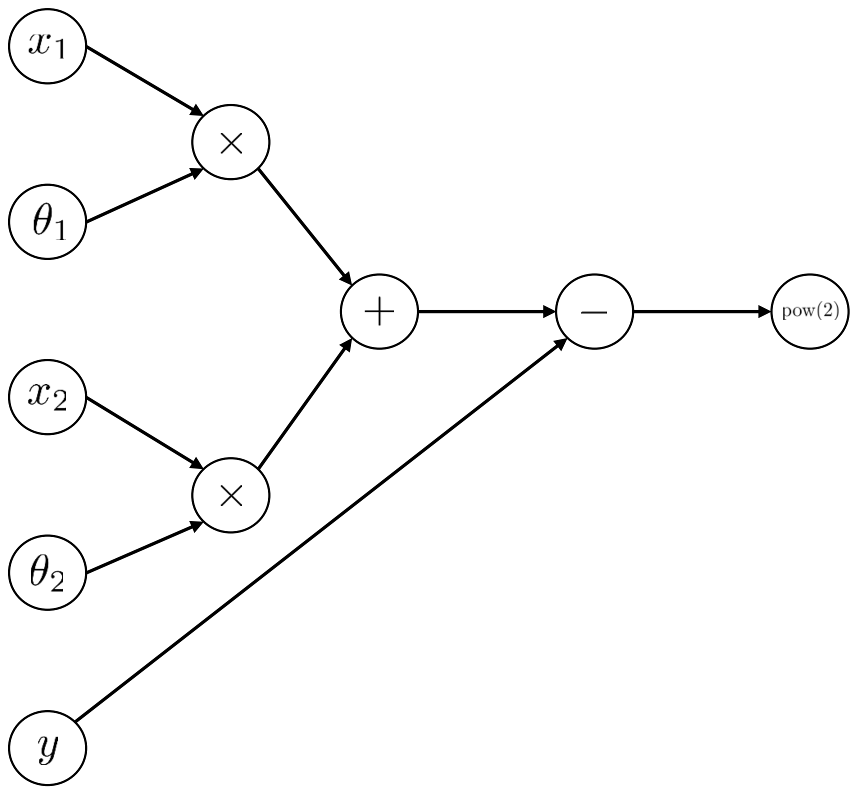
Neural networks

Drawing computation graphs



what **expression** does this compute?

equivalently, what **program** does this correspond to?

$$||(x_1\theta_1 + x_2\theta_2) - y||^2$$

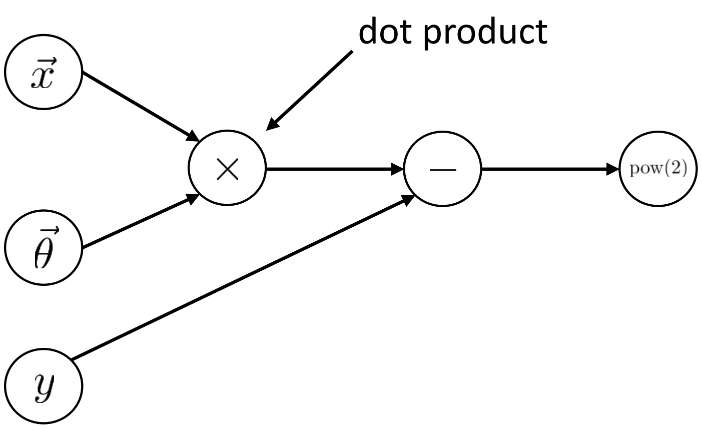
this is a **MSE loss** with a **linear regression** model

neural networks are **computation graphs**

if we design **generic tools** for computation graphs, we can train **many kinds** of neural networks

Drawing computation graphs

a simpler way to draw the same thing:



I'll drop the $\vec{}$ decorator from now on...

what **expression** does this compute?

equivalently, what **program** does this correspond to?

$$||(x_1\theta_1 + x_2\theta_2) - y||^2$$

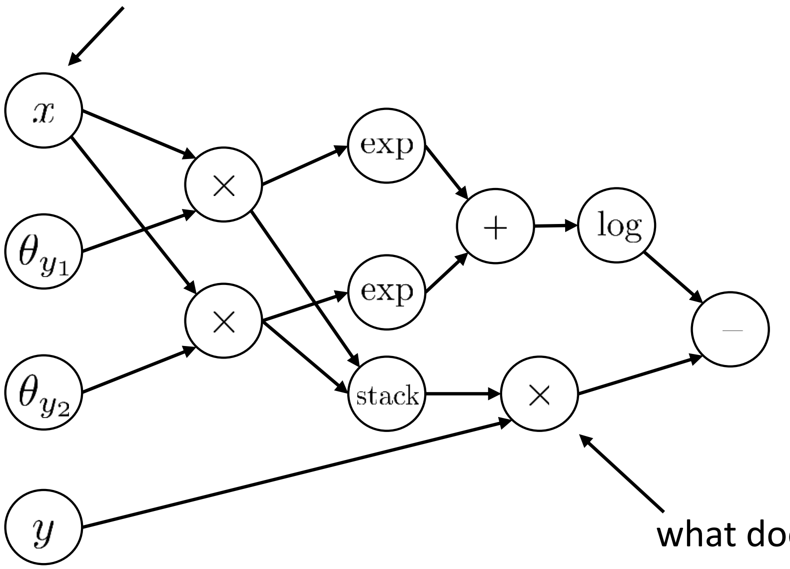
this is a **MSE loss** with a **linear regression** model

neural networks are **computation graphs**

if we design **generic tools** for computation graphs, we can train **many kinds** of neural networks

Logistic regression

remember this is a vector!



what does this produce?

$x^T \theta_y$

"one-hot" vector $\begin{bmatrix} 1 \\ 0 \end{bmatrix}$ or $\begin{bmatrix} 0 \\ 1 \end{bmatrix}$

let's draw the computation graph for **logistic regression**

with the negative log-likelihood loss

$$p_\theta(y|x) = \frac{\exp(x^T \theta_y)}{\sum_{y'} \exp(x^T \theta_{y'})}$$

$$-\log p_\theta(y|x) = -x^T \theta_y + \log \sum_{y'} \exp(x^T \theta_{y'})$$

Logistic regression

$$p_\theta(y|x) = \frac{\exp(x^T\theta_y)}{\sum_{y'}\exp(x^T\theta_{y'})}$$

a simpler way to draw the same thing:

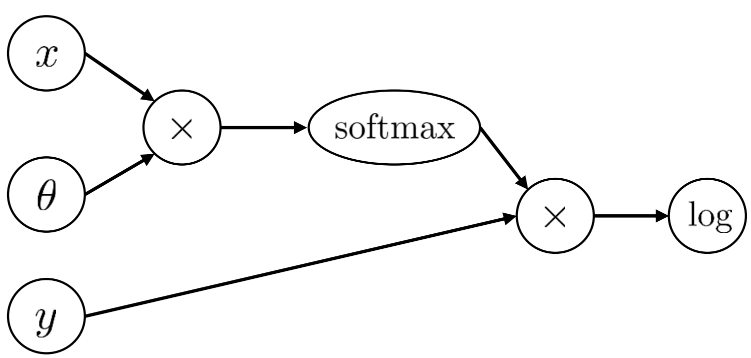
$$-\log p_\theta(y|x) = -x^T\theta_y + \log\sum_{y'}\exp(x^T\theta_{y'})$$

$$f_\theta(x) = \begin{bmatrix} x^T\theta_{y_1} \\ x^T\theta_{y_2} \\ \dots \\ x^T\theta_{y_m} \end{bmatrix}$$

$$f_\theta(x) = \theta x$$

matrix

$$\begin{bmatrix} \theta_{y_1} \\ \theta_{y_2} \\ \theta_{y_3} \end{bmatrix} \times \begin{bmatrix} x \end{bmatrix} = \begin{bmatrix} x^T\theta_{y_1} \\ x^T\theta_{y_2} \\ \dots \\ x^T\theta_{y_m} \end{bmatrix}$$



$$p_\theta(y = i|x) = \text{softmax}(f_\theta(x))[i] = \frac{\exp(f_{\theta,i}(x))}{\sum_{j=1}^m \exp(f_{\theta,j}(x))}$$

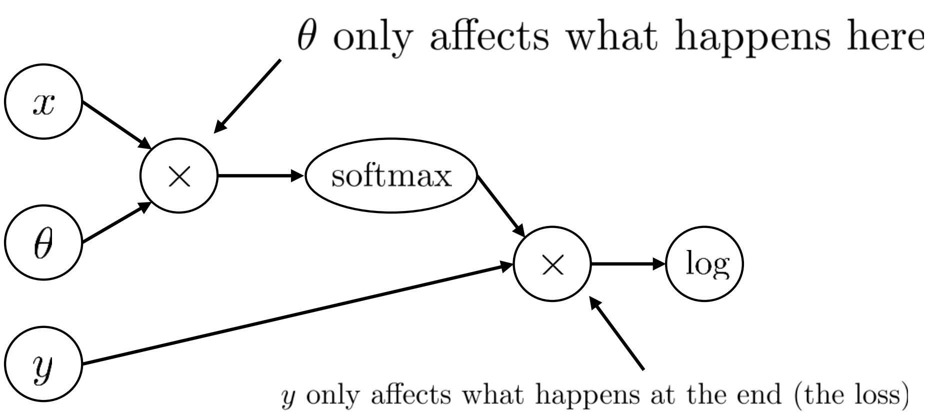
Drawing it even *more* concisely

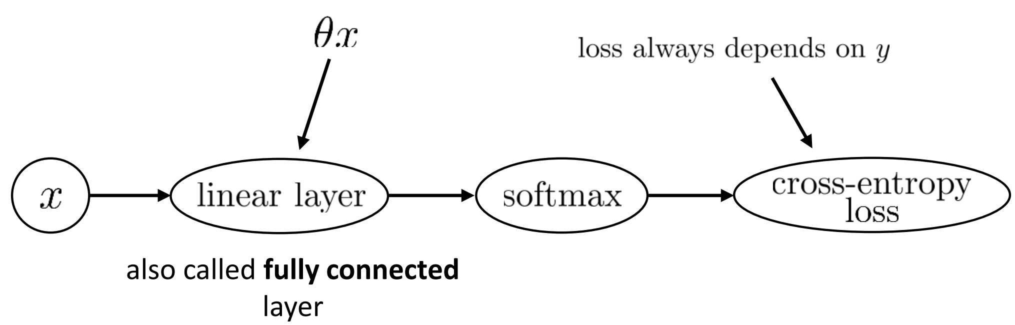
Notice that we have **two types** of variables:

data (e.g., x, y), which serves as input or target output

parameters (e.g., θ) the parameters *usually* affect one specific operation

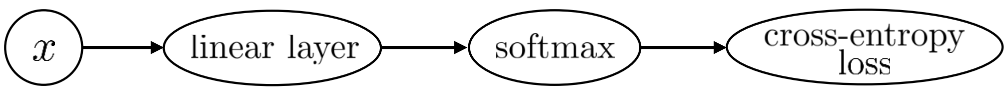
(though there is often *parameter sharing*, e.g., conv nets – more on this later)



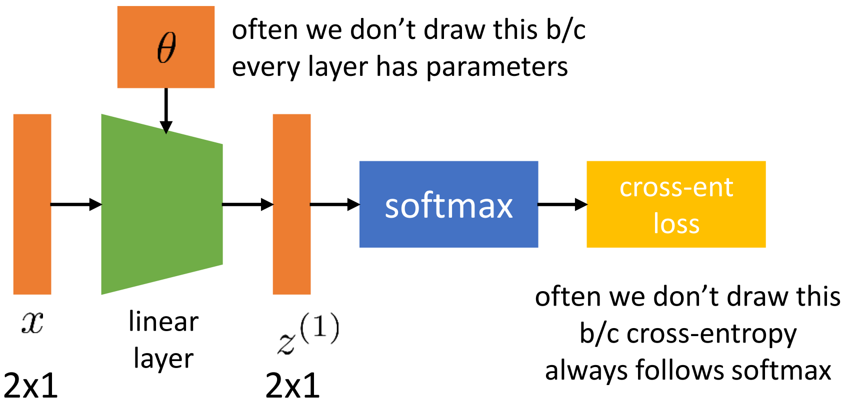


Neural network diagrams

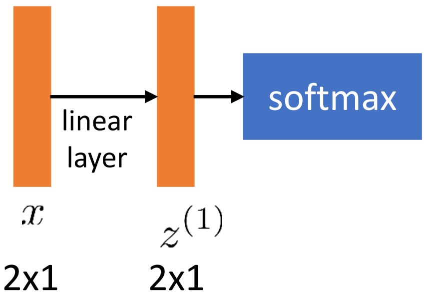
(simplified) computation graph diagram



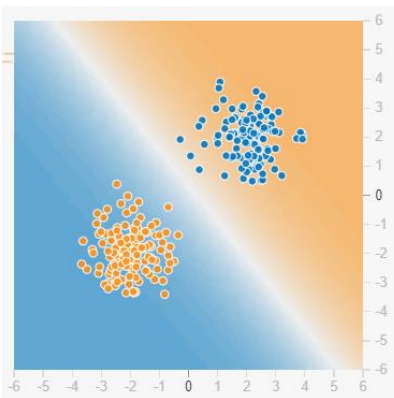
neural network diagram



simplified drawing:



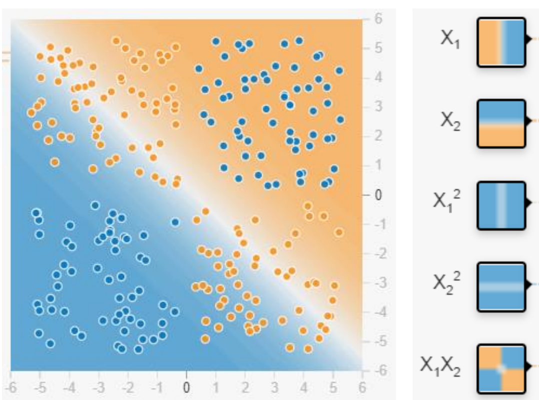
Logistic regression with features



pop quiz: what is the dimensionality of θ?

$$\text{softmax}(x^T\theta)$$

$$\phi(x) = \begin{pmatrix} x_1 \\ x_2 \\ x_1^2 \\ x_2^2 \\ x_1 x_2 \end{pmatrix}$$



$$\text{softmax}(\phi(x)^T\theta)$$

Learning the features

which layer

$$w_1^{(1)}$$

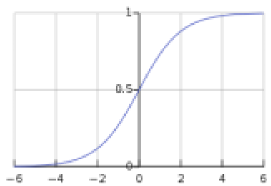
which feature
= rows of weight **matrix**

Problem: how do we represent the learned features?

Idea: what if each feature is a (binary) logistic regression output?

$$W^{(1)} \quad \begin{array}{|c|} \hline w_1^{(1)} \\ \hline w_2^{(1)} \\ \hline w_3^{(1)} \\ \hline \end{array}$$

$$\phi_1(x) = \text{softmax}(x^T w_1^{(1)}) = \frac{1}{1 + \exp(-x^T w_1^{(1)})}$$



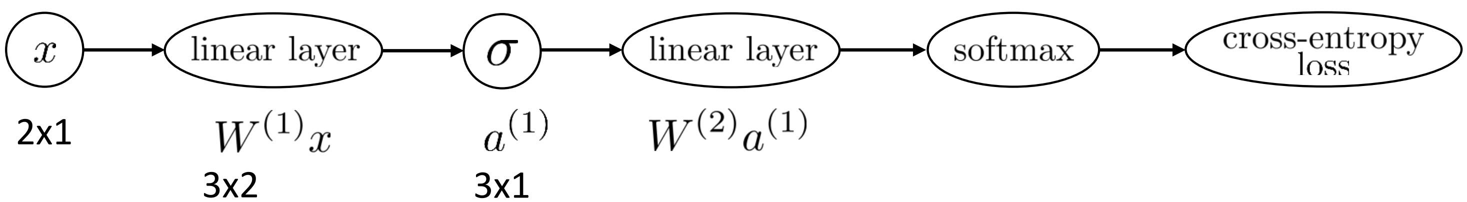
aside: I'll switch to use w or W instead of θ here
θ – *all* parameters of the model
$w_1^{(1)}$ – weights (a.k.a. parameters) of feature 1 at layer 1

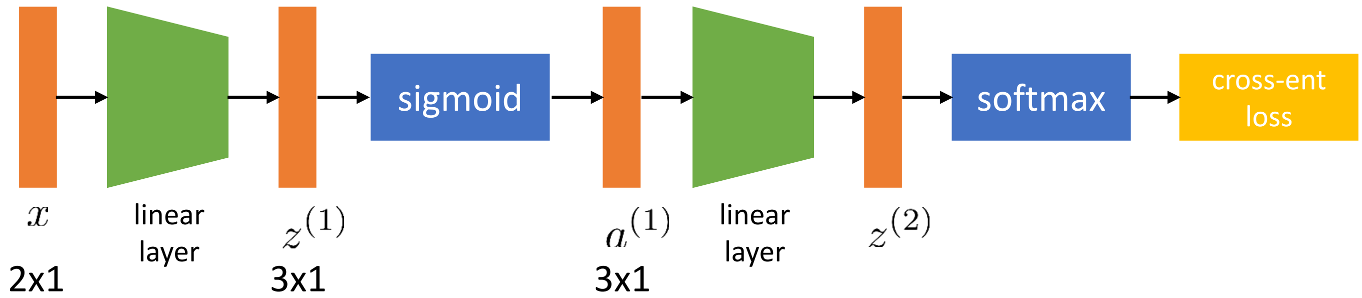
$$\phi(x) = \begin{pmatrix} \text{softmax}(x^T w_1^{(1)}) \\ \text{softmax}(x^T w_2^{(1)}) \\ \text{softmax}(x^T w_3^{(1)}) \end{pmatrix} = \sigma(W^{(1)} x)$$

per-element sigmoid
not the same as softmax
each feature is independent

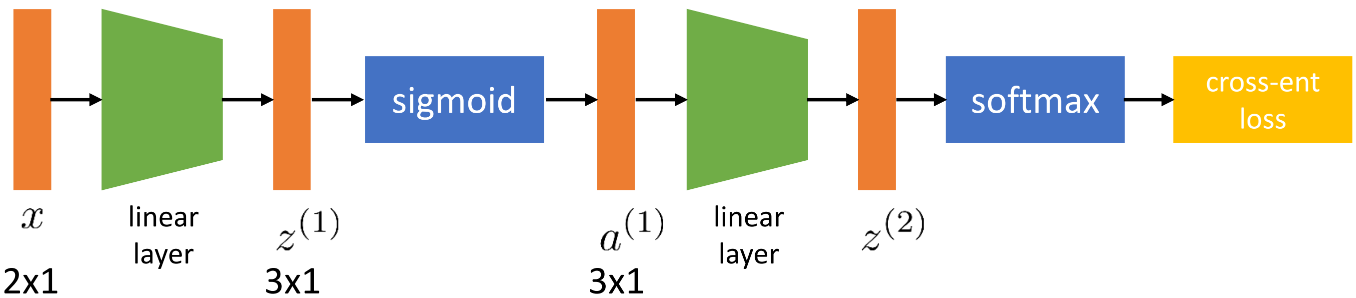
Let's draw this!

$$\phi(x) = \begin{pmatrix} \text{softmax}(x^T w_1^{(1)}) \\ \text{softmax}(x^T w_2^{(1)}) \\ \text{softmax}(x^T w_3^{(1)}) \end{pmatrix} = \sigma(W^{(1)}x) \qquad p(y|x) = \text{softmax}(\phi(x)^T \theta)$$

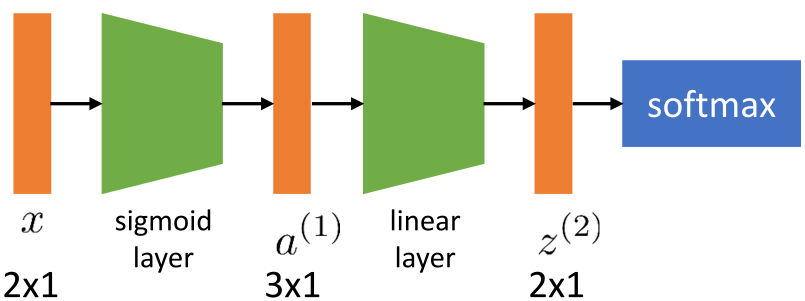




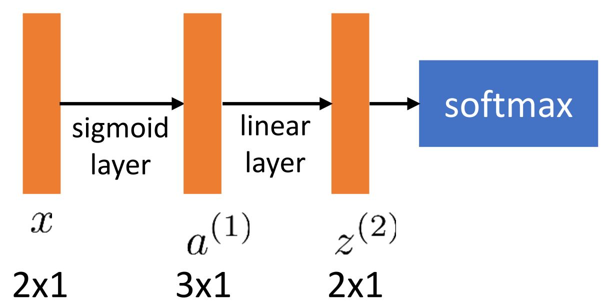
Simpler drawing



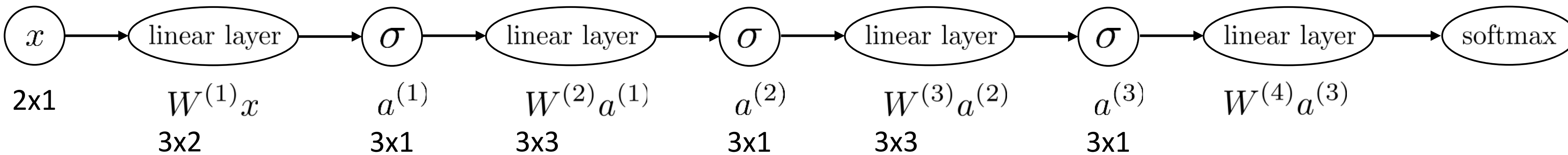
simpler way to draw the same thing:

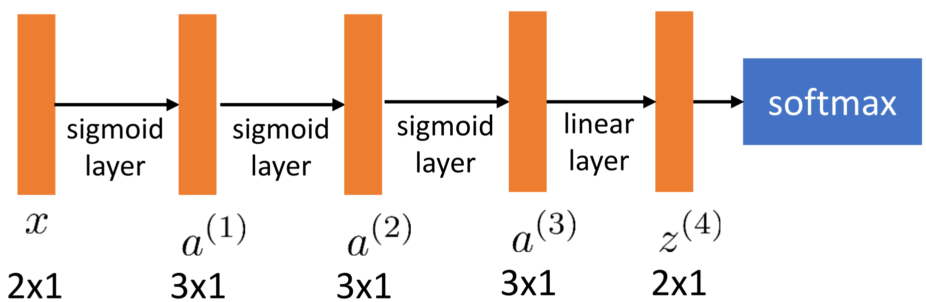


even simpler:



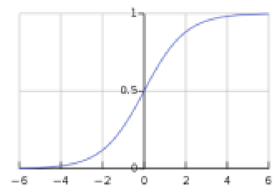
Doing it multiple times





Activation functions

$$\phi_1(x) = \text{softmax}(x^T w_1^{(1)}) = \frac{1}{1 + \exp(-x^T w_1^{(1)})}$$



we don't have to use a **sigmoid**!

a wide range of non-linear functions will work

these are called **activation functions**

we'll discuss specific choices later

why **non-linear?**

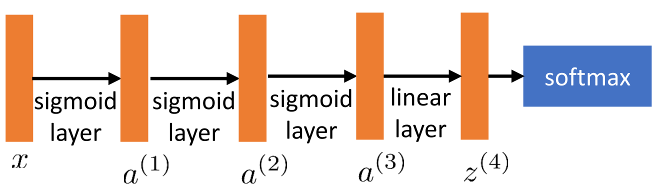
$$a^{(2)} = \sigma(W^{(2)}\sigma(W^{(1)}x))$$

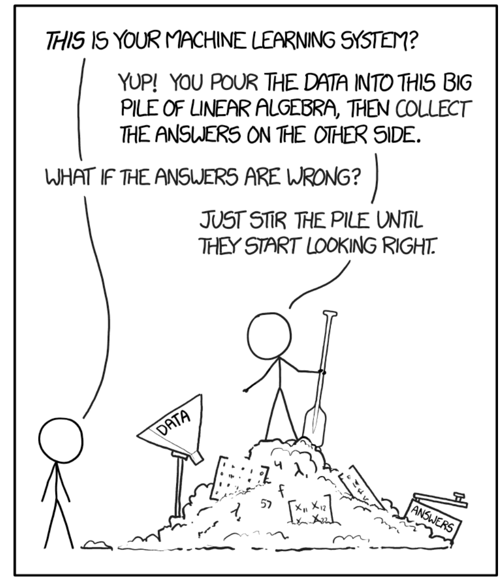
$$\text{if } \sigma(z) = z, \text{ then...}$$

$$a^{(2)} = W^{(2)}W^{(1)}x = Mx$$

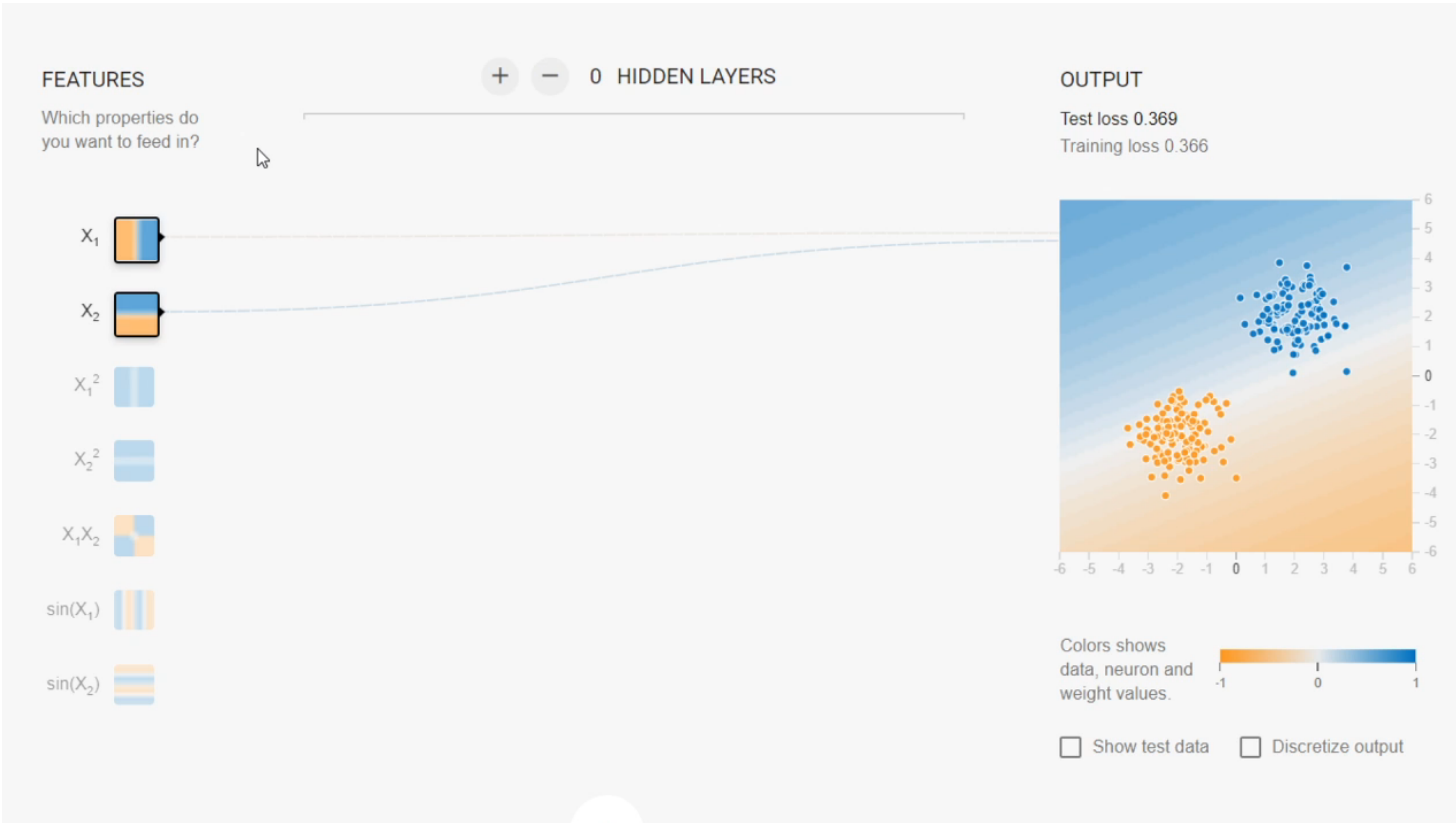
multiple linear layers = one linear layer

enough layers = we can represent anything (so long as they're nonlinear)





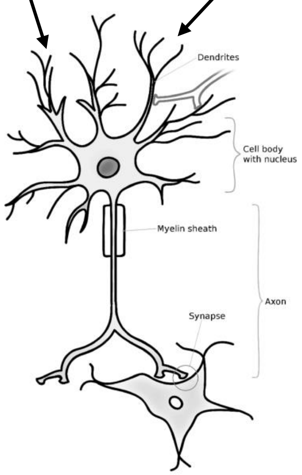
Demo time!



Source: https://playground.tensorflow.org/

Aside: what's so neural about it?

dendrites receive signals from other neurons



neuron "decides" whether to fire based on incoming signals

axon transmits signal to downstream neurons

artificial "neuron" sums up signals from upstream neurons (also referred to as "units")

$$z = \sum_i a_i$$

upstream activations

z

neuron "decides" how much to fire based on incoming signals

$$a = \sigma(z)$$

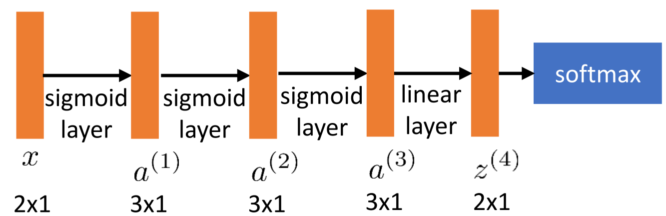
activation function

activations transmitted to downstream units

Training neural networks

What do we need?

1. Define your **model class**



2. Define your **loss function**

negative log-likelihood, just like before

3. Pick your **optimizer**

stochastic gradient descent

what do we need?

$$\nabla_\theta \mathcal{L}(\theta) = \begin{pmatrix} \frac{d\mathcal{L}(\theta)}{d\theta_1} \\ \frac{d\mathcal{L}(\theta)}{d\theta_2} \\ \vdots \\ \frac{d\mathcal{L}(\theta)}{d\theta_n} \end{pmatrix}$$

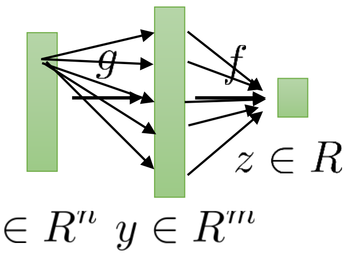
4. Run it on a big GPU

Aside: chain rule

Chain rule: $x \xrightarrow{g} y \xrightarrow{f} z$

$$\frac{d}{dx} f(g(x)) = \frac{dz}{dx} = \frac{dy}{dx}\frac{dz}{dy}$$

Jacobian of g Jacobian of f



High-dimensional chain rule

$$\frac{d}{dx_i} f(g(x)) = \sum_{j=1}^{m} \frac{dy_j}{dx_i}\frac{dz}{dy_j} = \frac{dy}{dx_i}\frac{dz}{dy}$$

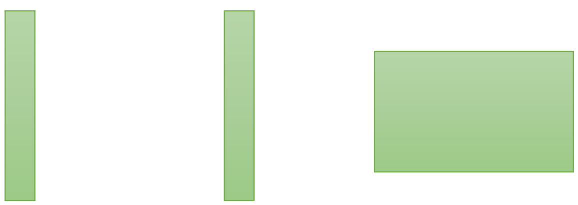
sum over all dimensions of y row $1 \times m$ col $m \times 1$

$$\frac{d}{dx} f(g(x)) = \frac{dy}{dx}\frac{dz}{dy}$$

mat $n \times m$ col $m \times 1$

Row or column?

In this lecture:



$y \in R^m$ $\frac{dz}{dy} \in R^m$ $\frac{dy}{dx} \in R^{n \times m}$

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

In some textbooks:

$y \in R^m$ $\frac{dz}{dy} \in R^m$

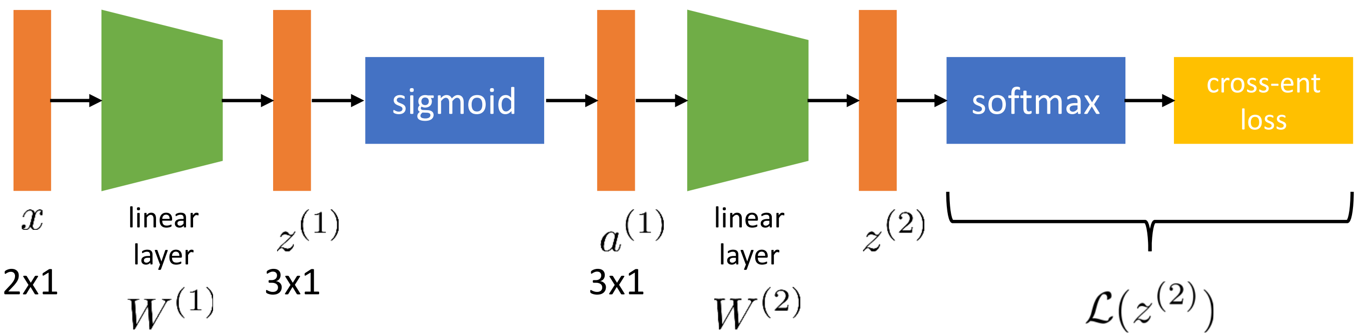
$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}$$

Just two different conventions!

Chain rule for neural networks

A neural network is just a composition of functions

So we can use chain rule to compute gradients!



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

Does it work?

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

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

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

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

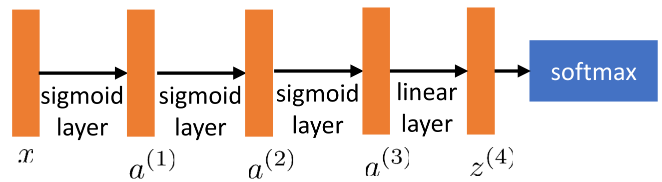
$$\frac{d\mathcal{L}}{dW^{(1)}} = \frac{dz^{(1)}}{dW^{(1)}} \underbrace{\frac{da^{(1)}}{dz^{(1)}} \delta}_{\text{this product is cheap: } O(n^2)}$$

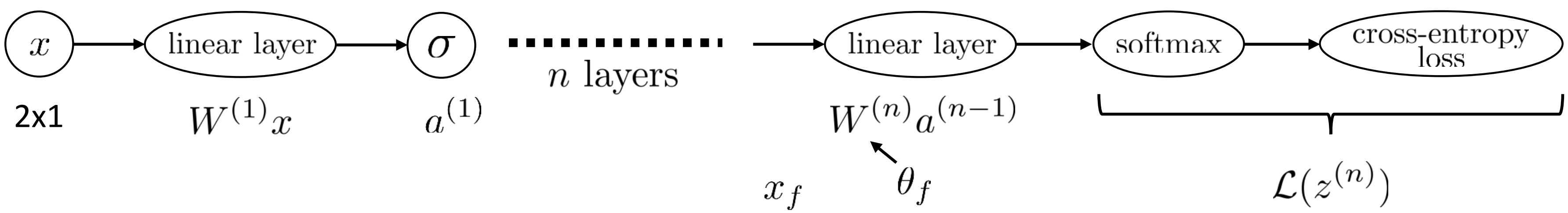
compute $\frac{da^{(1)}}{dz^{(1)}} \delta = \gamma$

$$\frac{d\mathcal{L}}{dW^{(1)}} = \underbrace{\frac{dz^{(1)}}{dW^{(1)}} \gamma}_{\text{this product is cheap: } O(n^2)}$$

The backpropagation algorithm

"Classic" version





x_f

forward pass: calculate each $a^{(i)}$ and $z^{(i)}$ $\quad a^{(n-1)} \longrightarrow f \longrightarrow z^{(n-1)}$

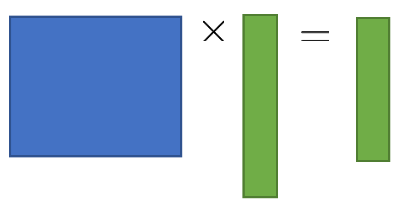
backward pass:

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

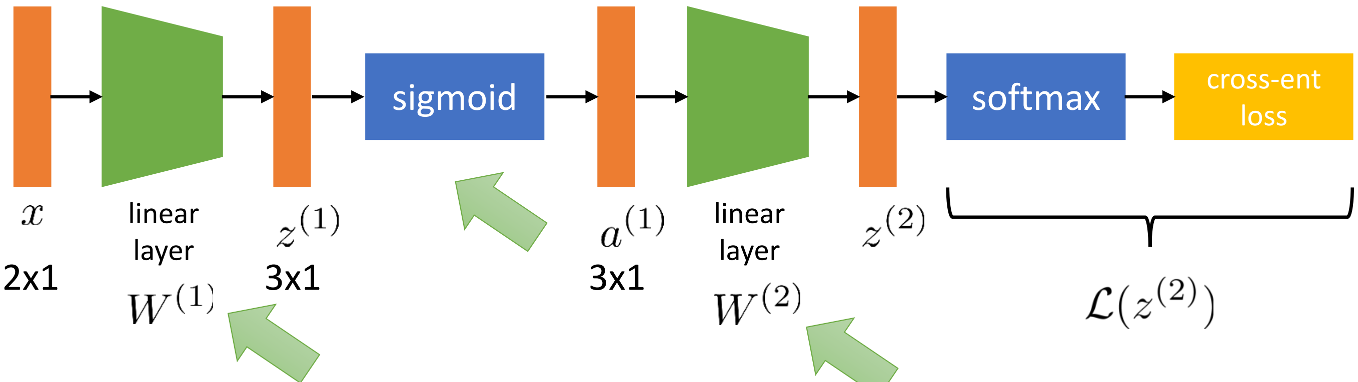
for each f with input x_f & params θ_f from end to start:

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

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



Let's walk through it...



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

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

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

backward pass:

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

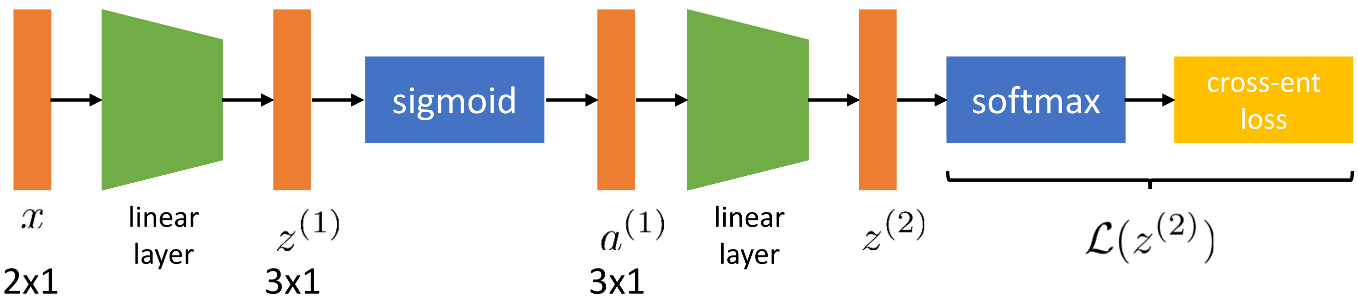
for each f with input x_f & params θ_f from end to start:

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

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

Practical implementation

Neural network architecture details



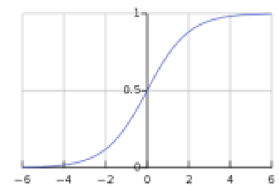
Some things we should figure out:

How many layers?

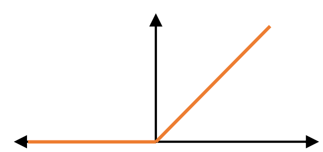
How big are the layers?

What type of **activation function**?

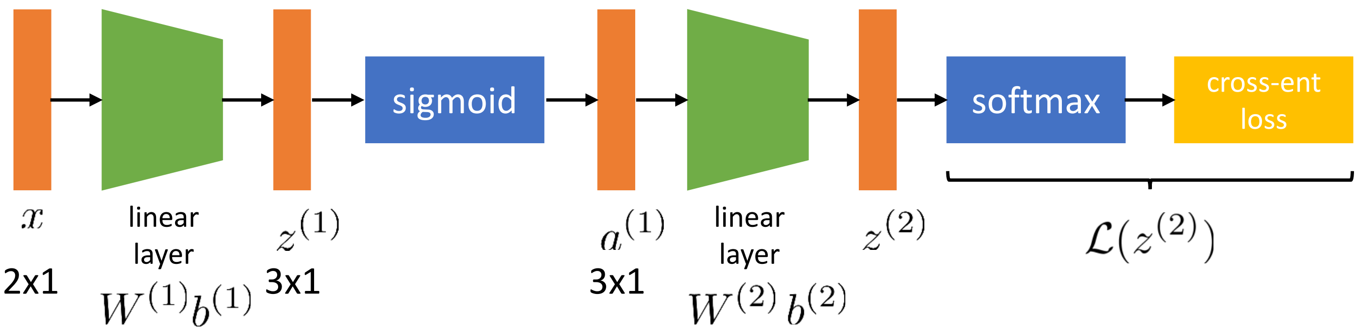
$$\sigma(x) = \frac{1}{1 + \exp(-x)}$$



$$\mathrm{ReLU}(x) = \max(0, x)$$



Bias terms



Linear layer:

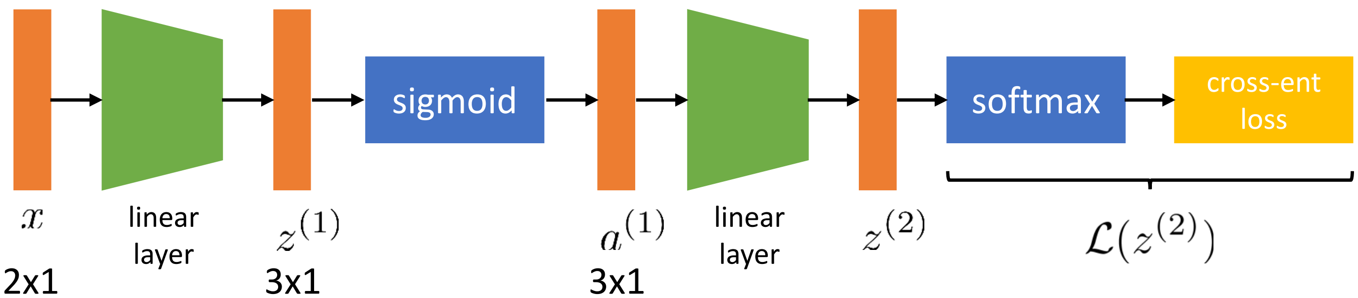
$z^{(i+1)} = W^{(i)} a^{(i)}$ problem: if $a^{(i)} = \vec{0}$, we always get 0...

Solution: add a "bias": has nothing to do with bias/variance bias

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

additional parameters in each linear layer

What else do we need for backprop?



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:

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

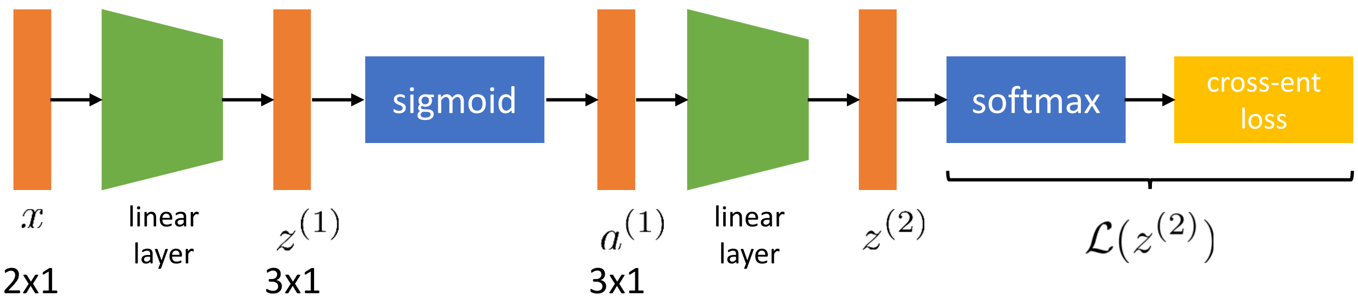
linear layer

softmax + cross-entropy

sigmoid

ReLU

Backpropagation recipes: linear layer



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

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

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

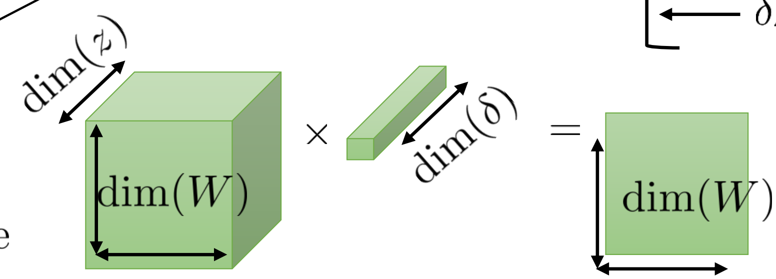
x_f

θ_f

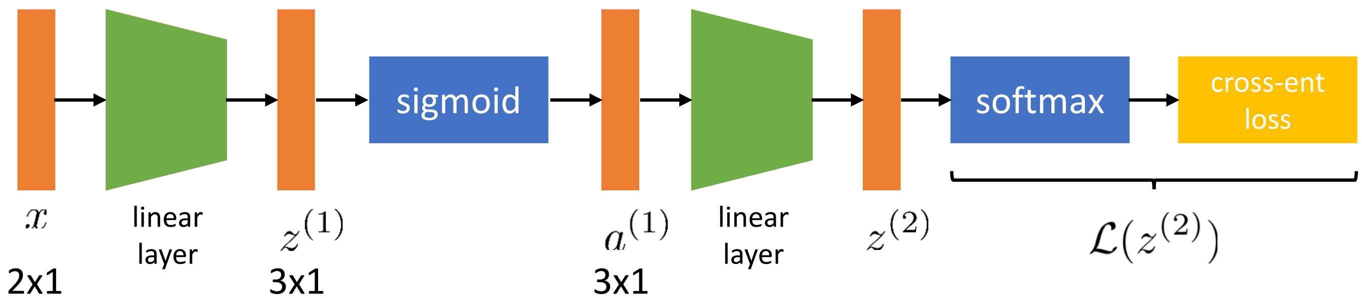
$$\begin{bmatrix} \longleftarrow & \delta_1 a^T & \longrightarrow \\ \longleftarrow & \delta_2 a^T & \longrightarrow \\ \longleftarrow & \delta_3 a^T & \longrightarrow \\ \longleftarrow & \delta_4 a^T & \longrightarrow \end{bmatrix}$$

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

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



Backpropagation recipes: linear layer



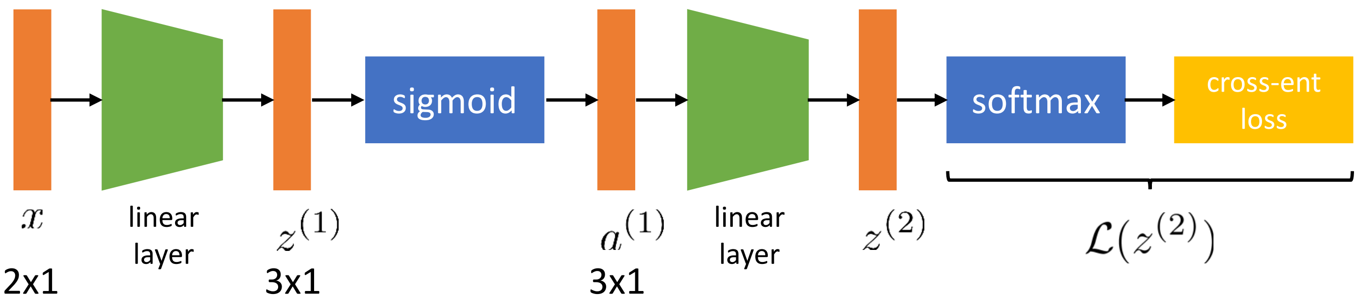
for each function, we need to compute: $\frac{df}{d\theta_f}\delta \quad \frac{df}{dx_f}\delta$

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

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

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

Backpropagation recipes: linear layer



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

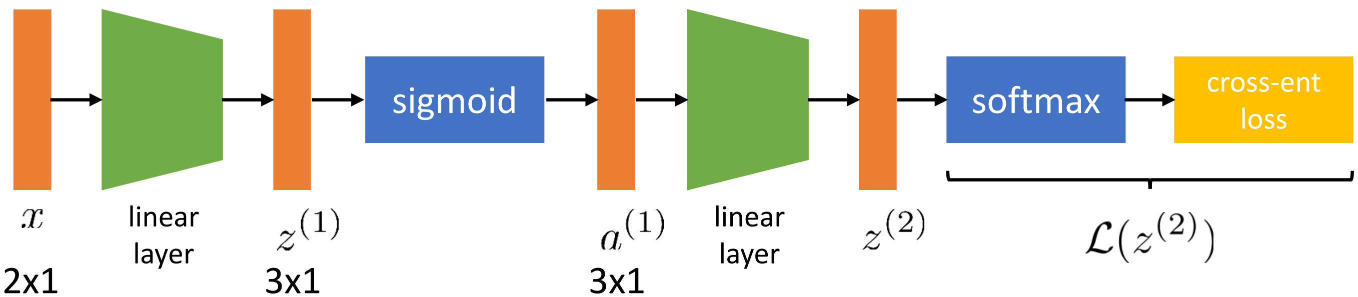
linear layer: $z^{(i+1)} = W^{(i)}a^{(i)} + b^{(i)}$ $\quad z = Wa + b$ (just to simplify notation!)

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

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

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

Backpropagation recipes: linear layer

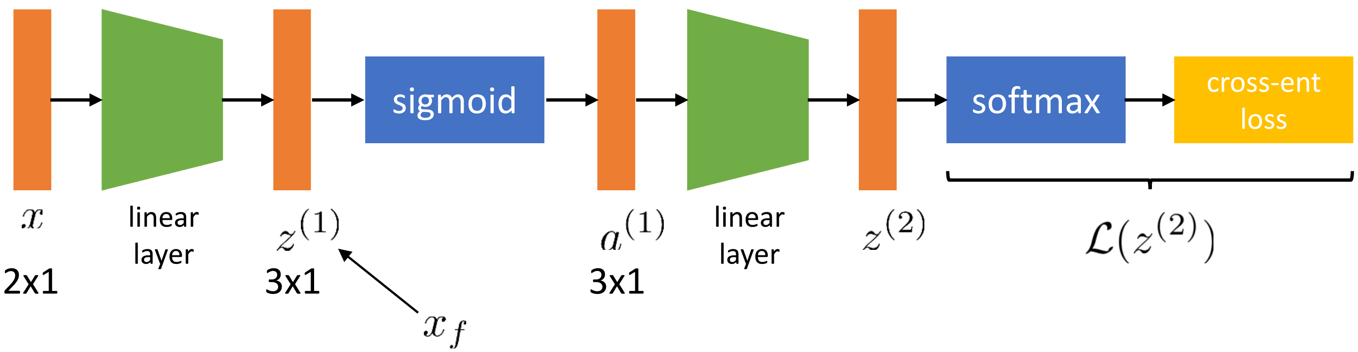


for each function, we need to compute: $\frac{df}{d\theta_f}\delta \quad \frac{df}{dx_f}\delta$

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



$$\frac{df}{dx} \begin{bmatrix} \frac{df_1}{dx_1} & 0 & 0 & 0 \\ 0 & \frac{df_2}{dx_2} & 0 & 0 \\ 0 & 0 & \frac{df_3}{dx_3} & 0 \\ 0 & 0 & 0 & \frac{df_4}{dx_4} \end{bmatrix}$$

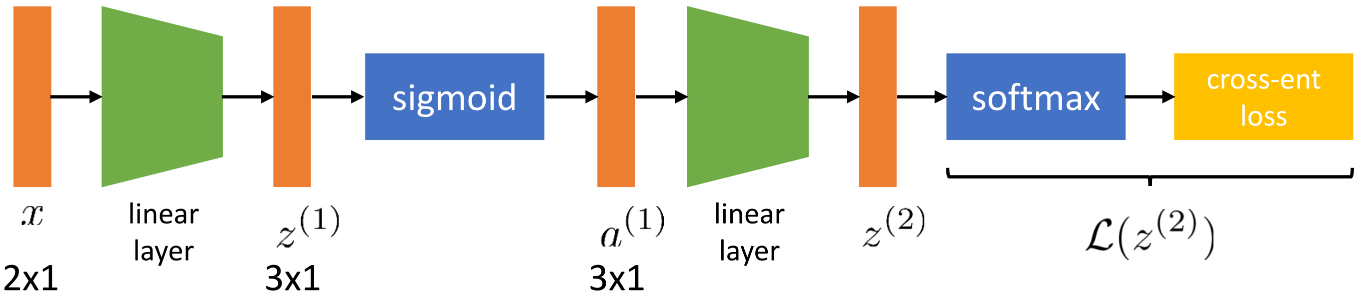
for each function, we need to compute: $\frac{df}{d\theta_f}\delta \quad \frac{df}{dx_f}\delta$

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

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

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

Backpropagation recipes: ReLU

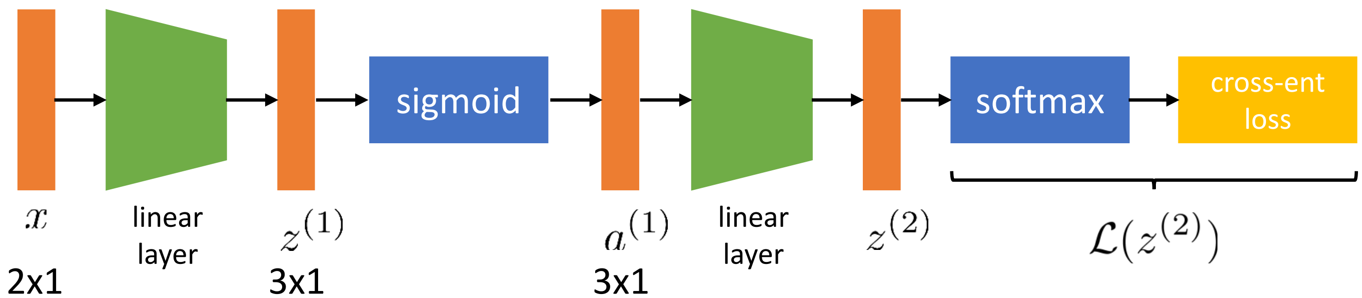


for each function, we need to compute: $\dfrac{df}{d\theta_f}\delta \quad \dfrac{df}{dx_f}\delta$

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

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

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