# Lecture 04: Neural network basics

Data C182 (Fall 2024). Week 03. Tuesday Sept 10th, 2024

Speaker: Eric Kim

# Announcements

- Welcome to Week 03!
- HW01 released, due Oct 1st!
    - MacOS/Windows: please use Docker to handle setting up the environment (eg installing packages/dependencies). See this Ed post for more details: [link]
    - Linux: feel free to either install the deps yourself, or you can also use Docker
    - Reminder: submit assignments via Gradescope [link]
    - Tip: If you're having trouble getting setup for HW01, please ask in Ed or attend office hours.

# Office hours, discussions

- Office hours are active this week! Full OH schedule: [link]
  - Eric Kim OH: Wednesdays, 3PM – 4 PM [Zoom link]
  - Naveen Ashish OH: Wednesdays 1PM – 2PM (Zoom link TBD)
- Discussions active starting this week! Notes + solutions on website: [link]
  - If you still aren't assigned to a discussion section, or you're unable to make your assigned discussion section (eg due to a conflict), please fill out the "2.0" Google Form in this Ed post: [link]
  - Please raise any discussion section assignment issues in this Ed post: [link]
  - Our aim is to get everyone assigned to a section by Week 04 (Sept 16th)
  - That said: feel free to attend any discussion section you prefer. Seats are reserved for those that are officially enrolled in that section.

# Midterm

- **Midterm**: Thursday October 24th 2024 (Week 09), 6:30 PM – 8:00 PM
  - In-person exam, pencil + paper.
  - **Physical location**: TBD (likely 10 Evans + another location on campus)
  - **Alternate exam times** will only be given for truly unavoidable, extraordinary circumstances. If you truly can't make this midterm time with a good reason, please write on Ed in a private post ASAP.

# Today’s lecture

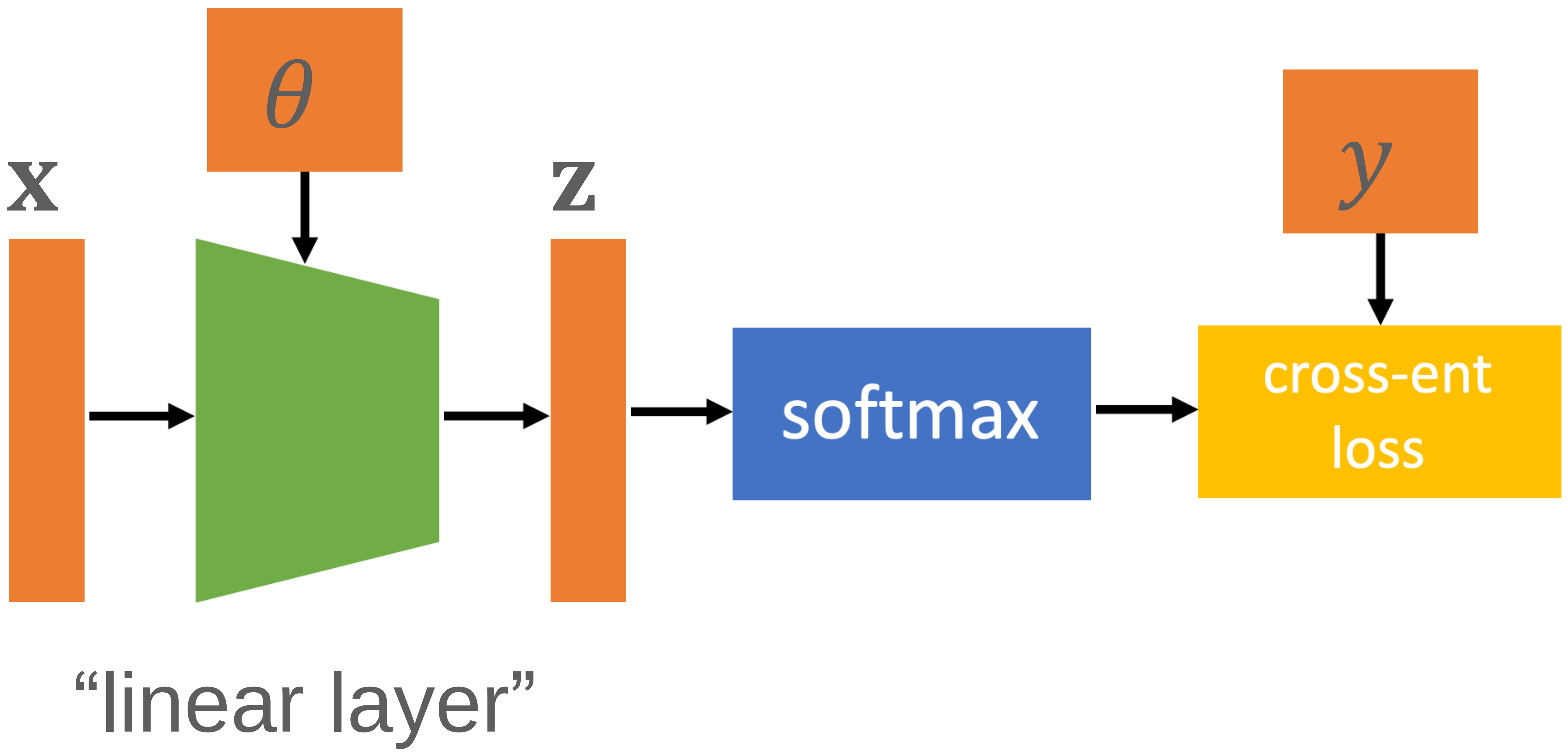
- Some of you may be thinking: “where are the deep neural networks??”
- Today, we’ll start talking about our first basic neural network models
  - We’ll put a full model together in this lecture, mathematically and diagrammatically
- We will then work through the **backpropagation** algorithm for computing gradients of the loss function with respect to the neural network parameters
  - This algorithm relies on reusing gradient values and matrix-vector products
  - Useful to learn and implement once (for the latter, HW1 has you covered)

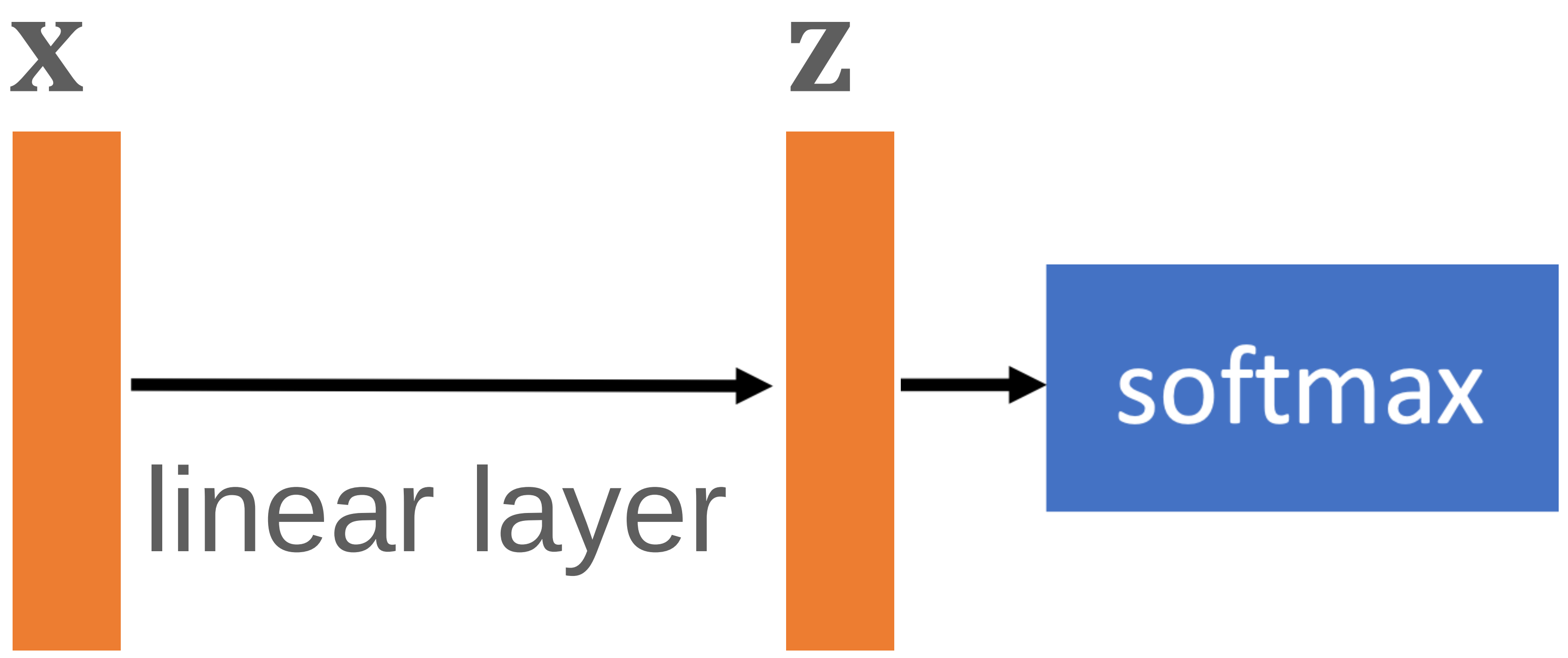
# Recall: logistic regression

- The “linear neural network”
  - **Setup: Multiclass classification**. Suppose we have K classes ("multiclass", K > 2), and each input sample consist of d input features
  - Given $\mathbf{x} \in \mathbb{R}^d$ ,define $f_\theta(\mathbf{x}) = \theta^\top \mathbf{x}$ ,where $\theta$ is a $d \times K$ matrix
  - Then, for class $c \in \{0, \ldots, K-1\}$, we have $p_\theta(y = c|\mathbf{x}) = softmax(f_\theta(\mathbf{x}))_c$
    - Remember: $softmax(f_\theta(\mathbf{x}))_c = \frac{\exp f_\theta(\mathbf{x})_c}{\sum_{i=0}^{K-1} \exp f_\theta(\mathbf{x})_i}$
  - Loss function: $\ell(\theta; \mathbf{x}, y) = -\log p_\theta(y|\mathbf{x})$

For a nice review of logistic regression, and how to generalize from binary classification to multiclass (K>2) classification, see: [link]

# A diagram for logistic regression

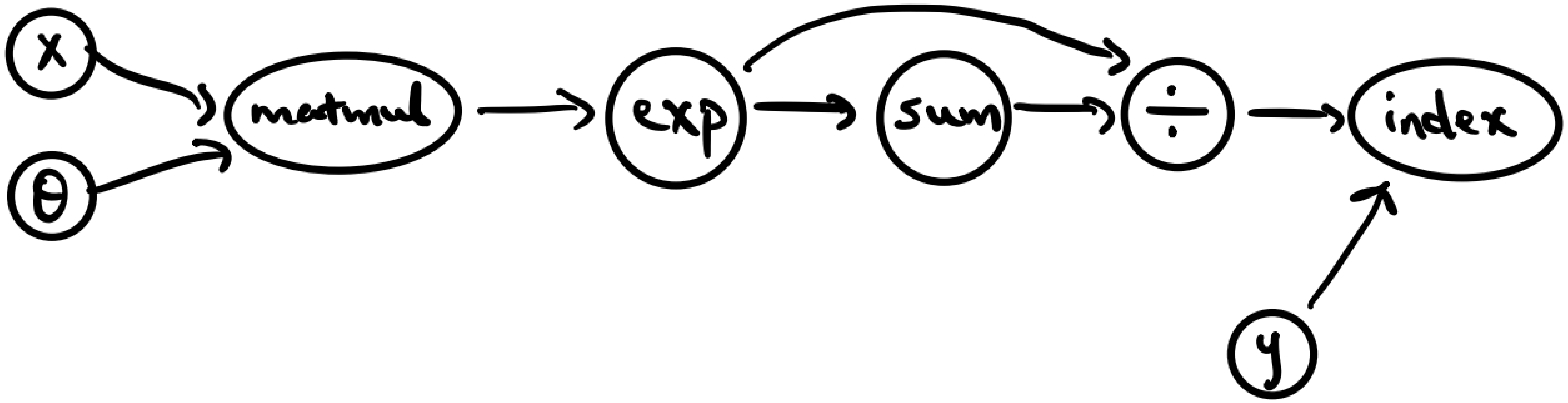




- Often, we will simplify this diagram:
  - Omit the $\theta$ box, the parameters are implicit in the diagram
  - Omit the layer box entirely! Denote it with just the arrow
  - Omit the loss box at the end, if we’re drawing “just the model”

# Another type of drawing: computation graphs

Computation graphs are more detailed, rigorous graphical representations



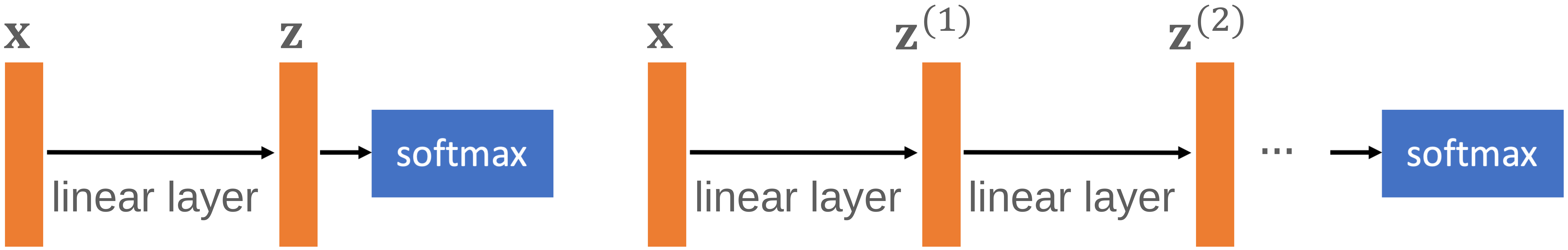
Pictured: the logistic regression model implemented as a series of mathematical "primitive" operations.

you will see variations on the style of drawing, level of detail, etc.

Aside: modern deep learning frameworks "compile" network architectures into a series of "primitive" operators (ex: Tensorflow/pytorch/Caffe/Caffe2). For a glimpse of this, see the "Operators" catalogue for Caffe2: [link]

# Neural networks: attempt #1

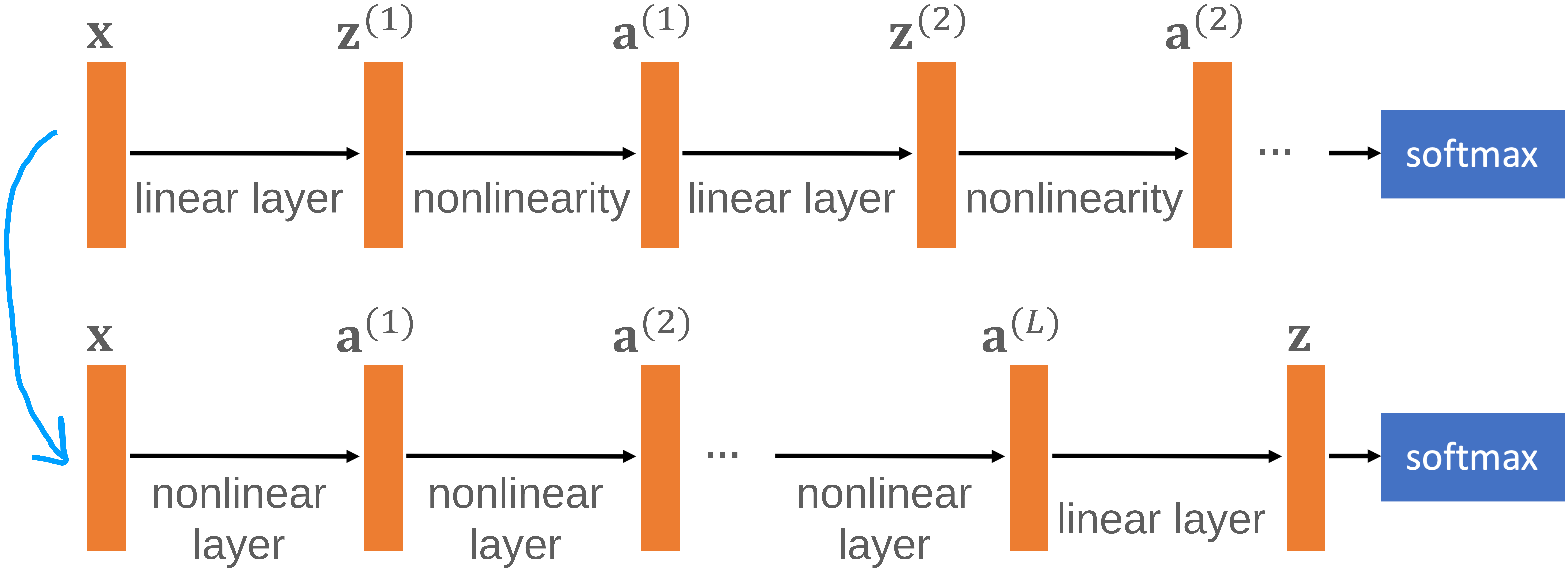
- Our drawing of logistic regression suggests that it is a “single layer model”
  - Are neural networks just more of these layers stacked on top of each other?
  - What’s the issue with this?
  - Composing linear transformations together is still linear!



# Making neural networks nonlinear

- One of the main things that makes neural networks great is that they can represent complex non linear functions
- How? The canonical answer: add **nonlinearities** after every linear layer
  - Also called **activation functions**
  - Basically always *element wise* functions on the linear layer output
- Examples: $tanh(\mathbf{z})$, $sigmoid(\mathbf{z}) = \frac{1}{\exp\{-\mathbf{z}\}+1}$, $ReLU(\mathbf{z}) = max\{0, \mathbf{z}\}$

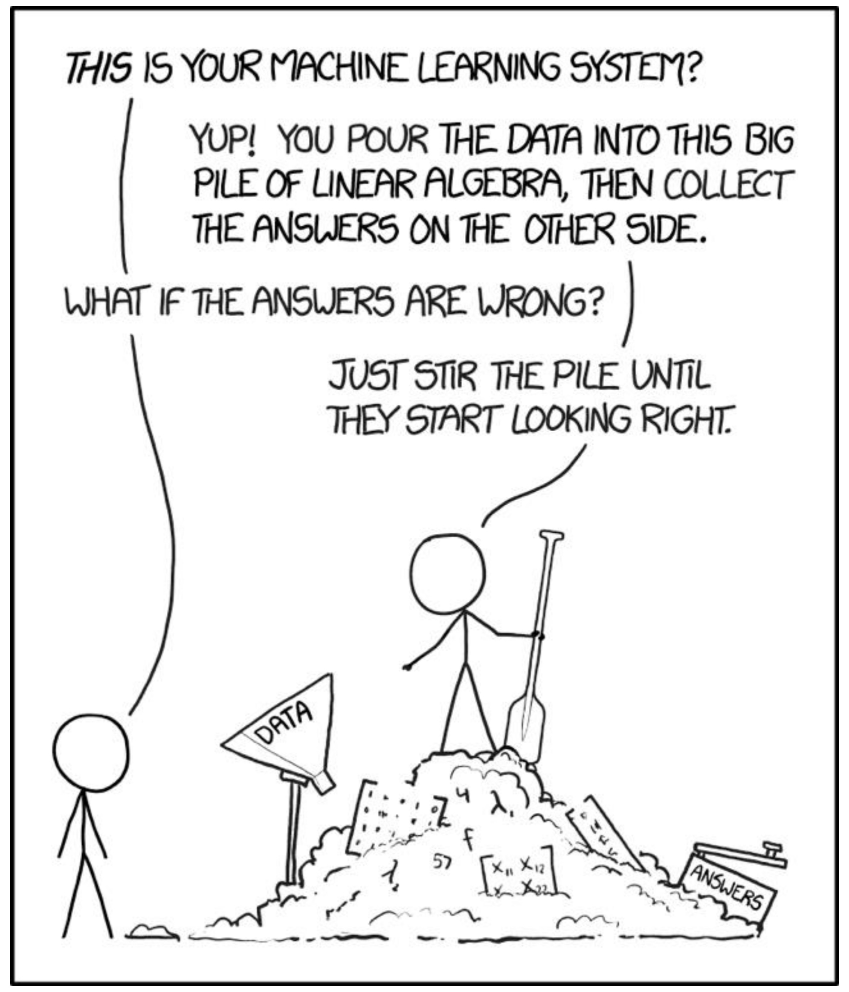
# Neural networks: attempt #2



To simplify the diagram, we often "merge" the linear layer with the nonlinear activation function

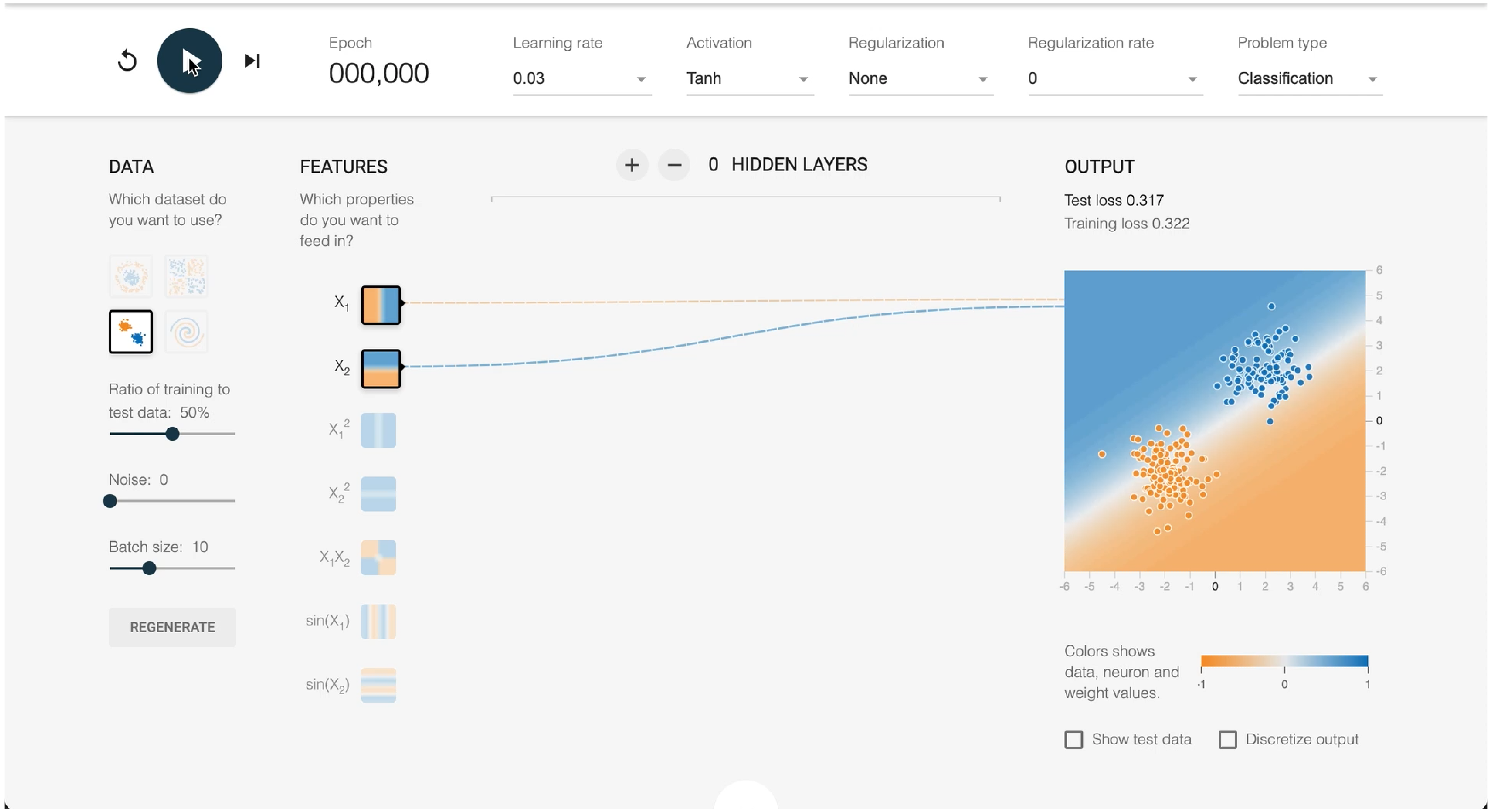
# What function is this?

- $\theta$ represents all our parameters, e.g., $[\mathbf{W}^{(1)}, \mathbf{b}^{(1)}, \dots, \mathbf{W}^{(L)}, \mathbf{b}^{(L)}, \mathbf{W}^{final}, \mathbf{b}^{final}]$
- If our neural network has parameters $\theta$ and $L$ **hidden layers**, then it represents the function $f_\theta(\mathbf{x}) = softmax(A^{final}(\sigma(A^{(L)}(\dots\sigma(A^{(1)}(\mathbf{x}))\dots))))$
  - $\sigma$ is the nonlinearity / activation function
  - $A^i(\mathbf{v}) = \mathbf{W}^i\mathbf{v} + \mathbf{b}^i$ is the $i$-th linear layer
- What can this function represent? Turns out, a lot


THIS IS YOUR MACHINE LEARNING SYSTEM?
YUP! YOU POUR THE DATA INTO THIS BIG PILE OF LINEAR ALGEBRA, THEN COLLECT THE ANSWERS ON THE OTHER SIDE.
WHAT IF THE ANSWERS ARE WRONG?
JUST STIR THE PILE UNTIL THEY START LOOKING RIGHT.
DATA
ANSWERS

# Visualizing neural network functions

- https://playground.tensorflow.org/



Epoch
000,000

Learning rate
0.03

Activation
Tanh

Regularization
None

Regularization rate
0

Problem type
Classification

DATA

Which dataset do you want to use?

Ratio of training to test data: 50%

Noise: 0

Batch size: 10

REGENERATE

FEATURES

Which properties do you want to feed in?

0 HIDDEN LAYERS

OUTPUT

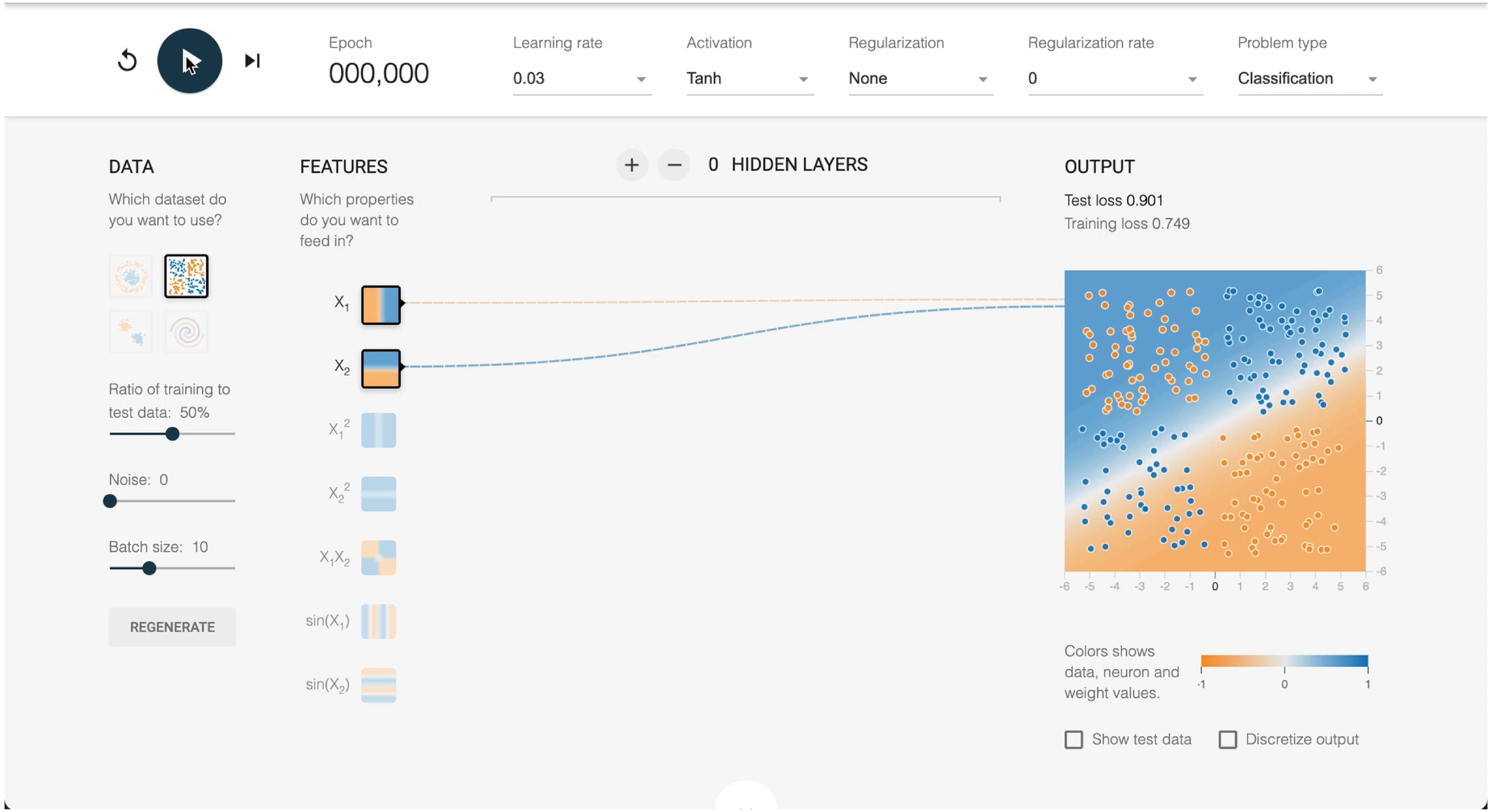
Test loss 0.317
Training loss 0.322

Colors shows data, neuron and weight values.

□ Show test data □ Discretize output

# Visualizing neural network functions

- https://playground.tensorflow.org/



Epoch 000,000

Learning rate 0.03

Activation Tanh

Regularization None

Regularization rate 0

Problem type Classification

DATA

Which dataset do you want to use?

Ratio of training to test data: 50%

Noise: 0

Batch size: 10

REGENERATE

FEATURES

Which properties do you want to feed in?

0 HIDDEN LAYERS

OUTPUT

Test loss 0.901

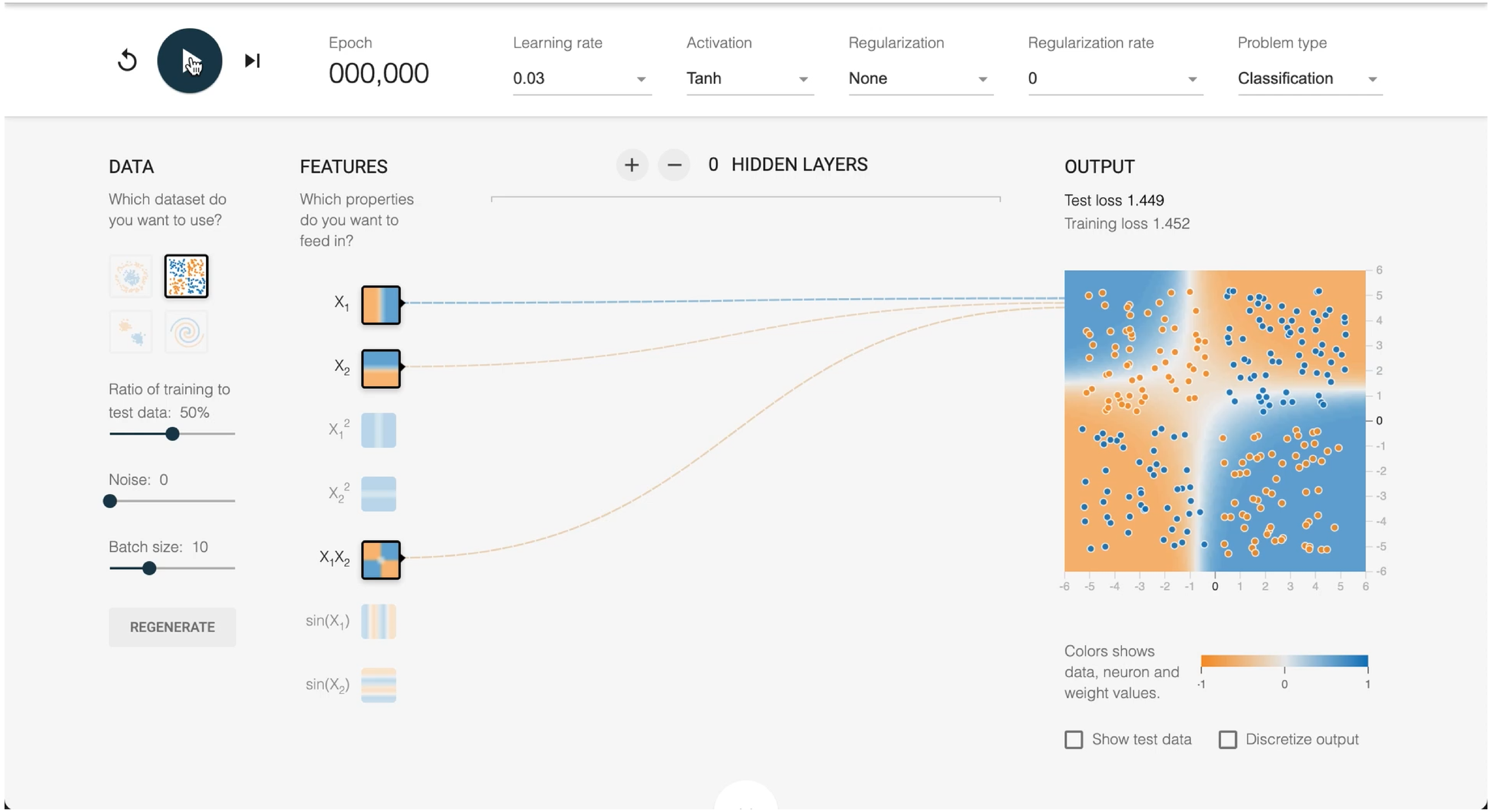
Training loss 0.749

Colors shows data, neuron and weight values.

□ Show test data □ Discretize output

# Visualizing neural network functions

- https://playground.tensorflow.org/



Epoch 000,000

Learning rate 0.03

Activation Tanh

Regularization None

Regularization rate 0

Problem type Classification

DATA

Which dataset do you want to use?

Ratio of training to test data: 50%

Noise: 0

Batch size: 10

FEATURES

Which properties do you want to feed in?

0 HIDDEN LAYERS

OUTPUT

Test loss 1.449

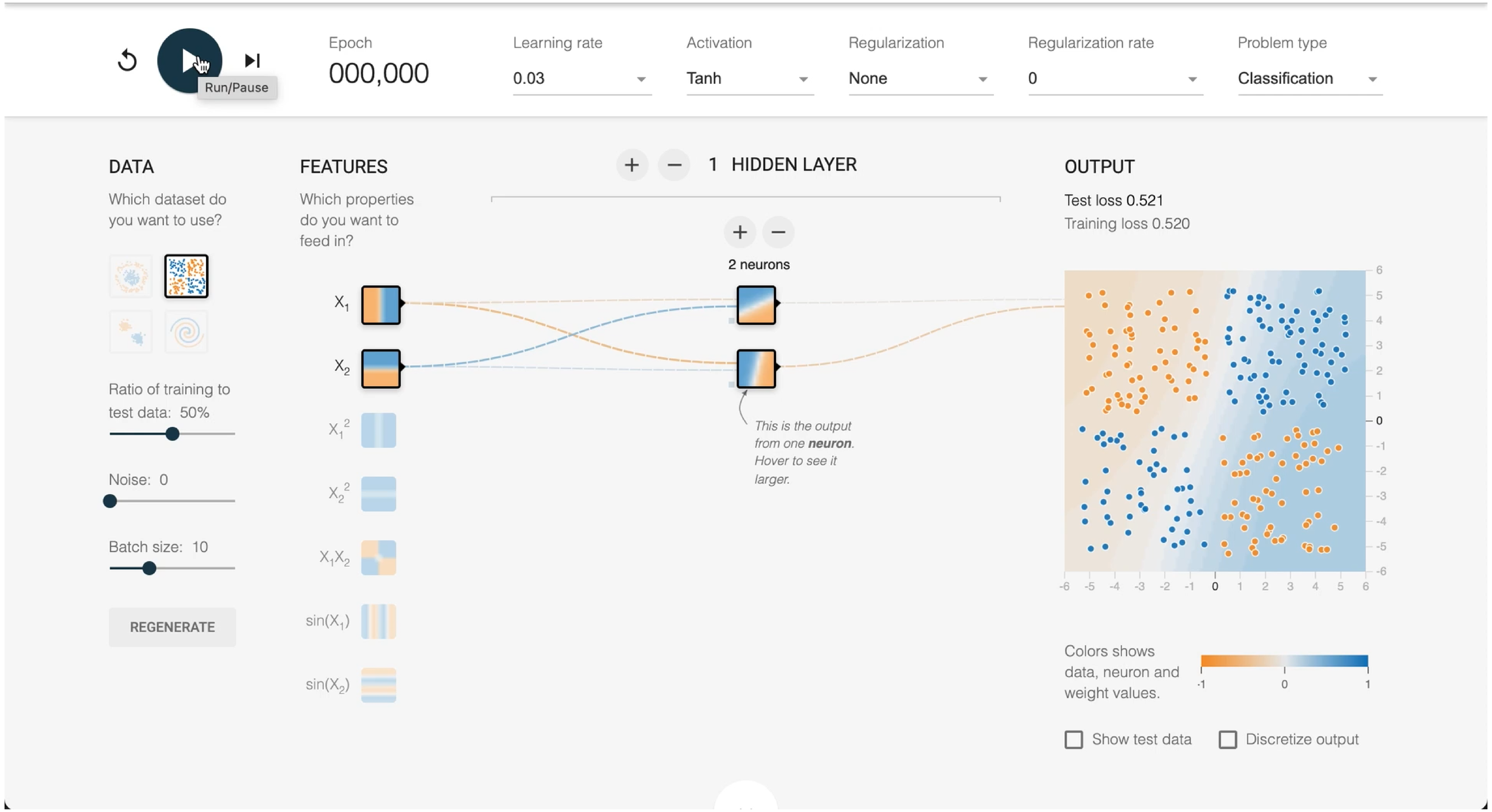
Training loss 1.452

Colors shows data, neuron and weight values.

□ Show test data □ Discretize output

# Visualizing neural network functions

- https://playground.tensorflow.org/



Run/Pause

Epoch 000,000

Learning rate 0.03

Activation Tanh

Regularization None

Regularization rate 0

Problem type Classification

DATA

Which dataset do you want to use?

Ratio of training to test data: 50%

Noise: 0

Batch size: 10

REGENERATE

FEATURES

Which properties do you want to feed in?

1 HIDDEN LAYER

2 neurons

*This is the output from one* ***neuron****. Hover to see it larger.*

OUTPUT

Test loss 0.521

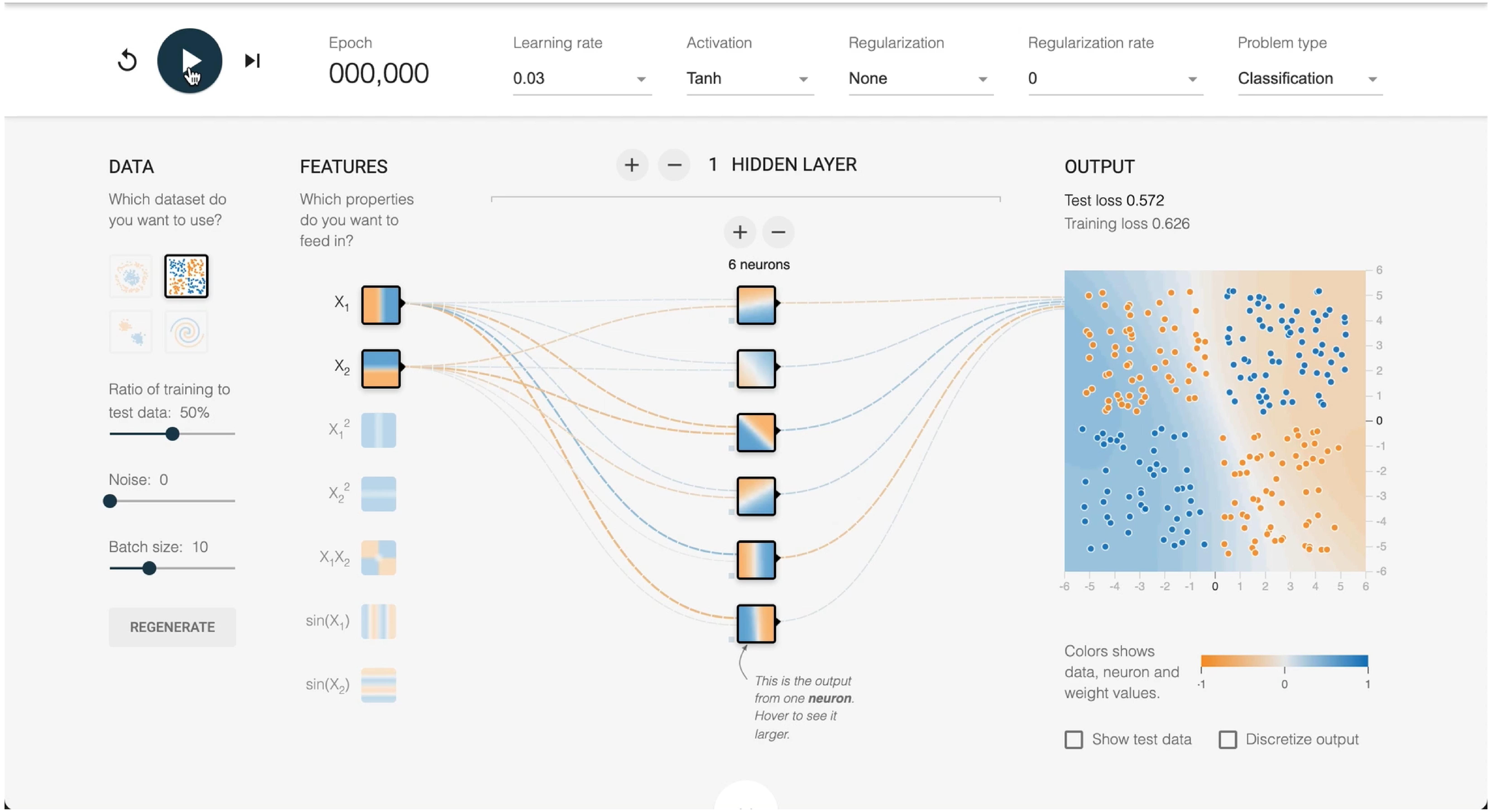
Training loss 0.520

Colors shows data, neuron and weight values.

□ Show test data □ Discretize output

# Visualizing neural network functions

- https://playground.tensorflow.org/



Epoch 000,000

Learning rate 0.03

Activation Tanh

Regularization None

Regularization rate 0

Problem type Classification

DATA

Which dataset do you want to use?

Ratio of training to test data: 50%

Noise: 0

Batch size: 10

REGENERATE

FEATURES

Which properties do you want to feed in?

1 HIDDEN LAYER

6 neurons

*This is the output from one* ***neuron****. Hover to see it larger.*

OUTPUT

Test loss 0.572

Training loss 0.626

Colors shows data, neuron and weight values.

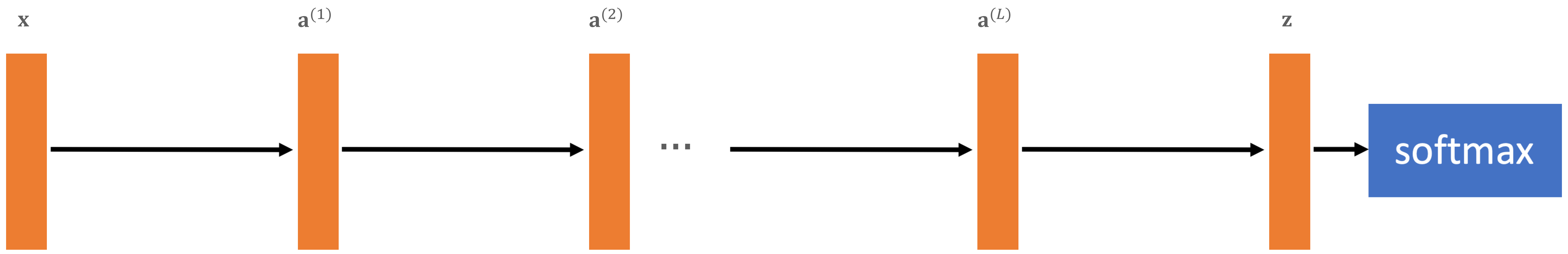
□ Show test data □ Discretize output

# The backpropagation algorithm

# Remember: the machine learning method

- (or, at least, the deep learning method)

1. Define your **model**
2. Define your **loss function**
3. Define your **optimizer**
4. Run it on a big GPU



$\ell(\theta; \mathbf{x}, y) = -\log p_\theta(y|\mathbf{x})$ (“cross-entropy”)

$$\theta \leftarrow \theta - \alpha \nabla_\theta \frac{1}{N} \sum_{i=1}^{N} \ell(\theta; \mathbf{x}_i, y_i)$$

wait… we need gradients!

# What gradients do we need?

- We want to update our parameters as $\theta \leftarrow \theta - \alpha \nabla_\theta \frac{1}{N} \sum_{i=1}^{N} \ell(\theta; \mathbf{x}_i, y_i)$
- $\theta$ represents all our parameters, e.g., $[\mathbf{W}^{(1)}, \mathbf{b}^{(1)}, \ldots, \mathbf{W}^{(L)}, \mathbf{b}^{(L)}, \mathbf{W}^{final}, \mathbf{b}^{final}]$
- So we need $[\nabla_{\mathbf{W}^{(1)}} \ell, \nabla_{\mathbf{b}^{(1)}} \ell, \ldots, \nabla_{\mathbf{W}^{(L)}} \ell, \nabla_{\mathbf{b}^{(L)}} \ell, \nabla_{\mathbf{W}^{final}} \ell, \nabla_{\mathbf{b}^{final}} \ell]$
- How do we compute these gradients? Let’s talk about two different approaches:
  - numerical (**finite differences**) vs. analytical (backpropagation)

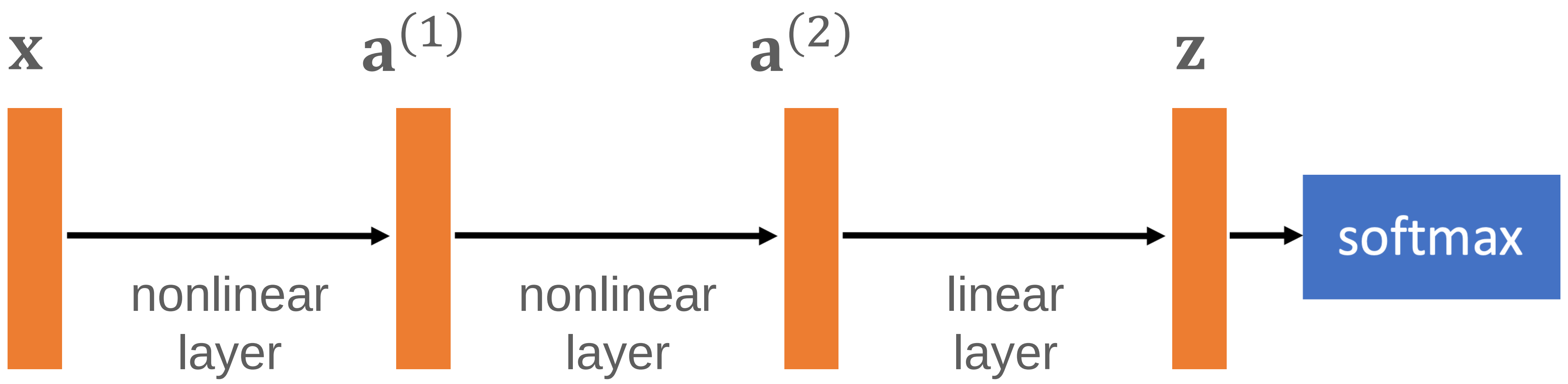
# Finite differences

- The method of finite differences says that, for any sufficiently smooth function $f$ which operates on a vector $\mathbf{x}$, the partial derivative $\frac{\partial f}{\partial x_i}$ is approximated by
- $\frac{\partial f}{\partial x_i} \approx \frac{f(\mathbf{x}+\epsilon \mathbf{e}_i)-f(\mathbf{x}-\epsilon \mathbf{e}_i)}{2\epsilon}$, where $\mathbf{e}_i$ denotes a “one hot” vector
- This is the definition of (partial) derivatives as $\epsilon \rightarrow 0$
- Think about how slow this would be to do for all our network parameters… Nevertheless, it can be useful as a method for checking gradients

# Computing gradients via backpropagation

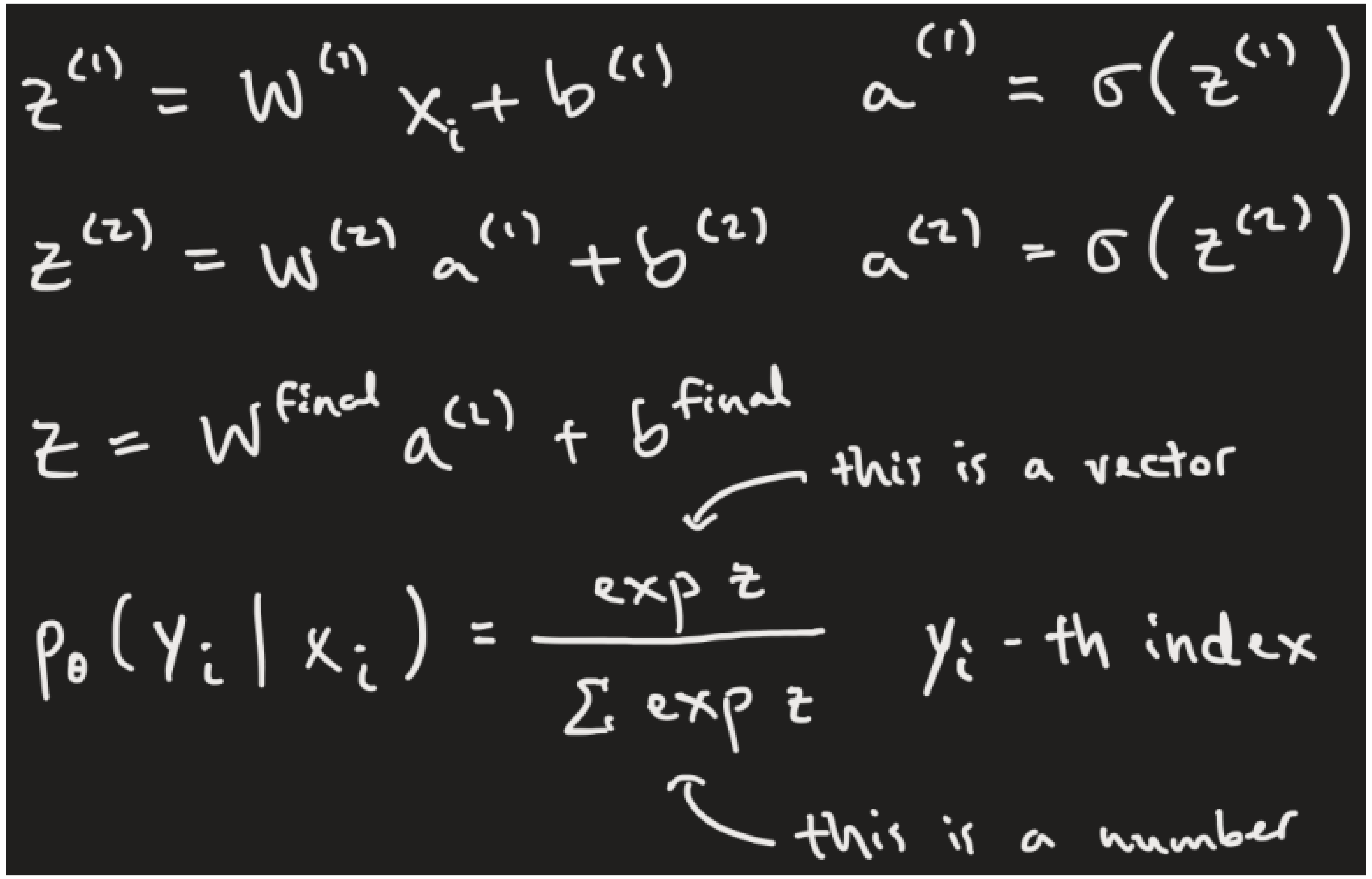
- The backpropagation algorithm is a much faster and more efficient method for computing gradients for neural network parameters
  - It made training large neural networks feasible and practical
- Backpropagation works “backward” through the network, which allows for:
  - reusing gradient values that have already been computed
  - computing matrix-vector products rather than matrix-matrix products, since the loss is a scalar!
- It’s pretty confusing the first (or second, or third, …) time you see it

# Backpropagation: the math



first, let’s do the “forward pass” through our network, from input to prediction

let’s work with two hidden layers, for concreteness



$$z^{(1)} = W^{(1)} x_i + b^{(1)} \qquad a^{(1)} = \sigma(z^{(1)})$$

$$z^{(2)} = W^{(2)} a^{(1)} + b^{(2)} \qquad a^{(2)} = \sigma(z^{(2)})$$

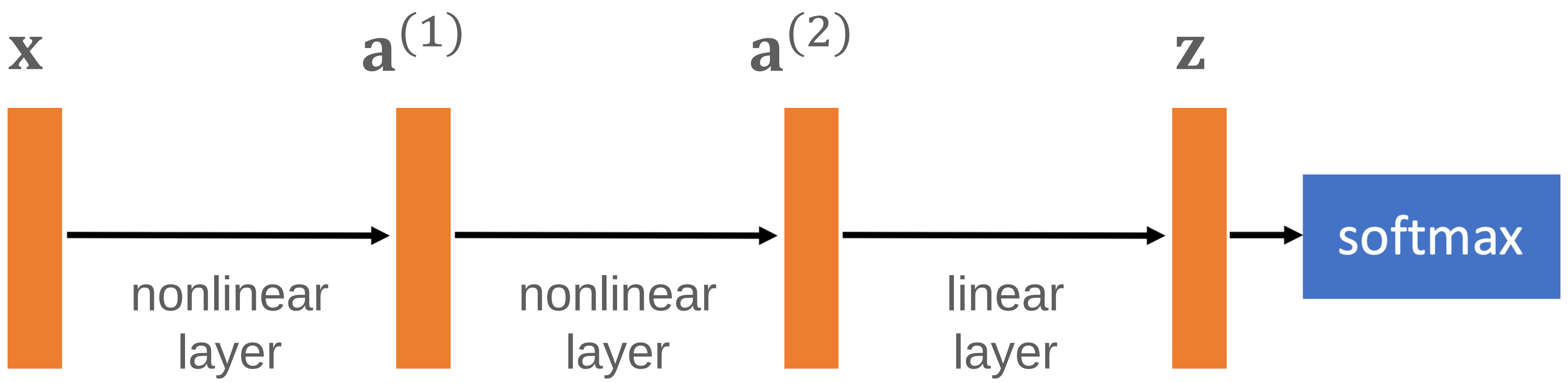
$$z = W^{\text{final}} a^{(2)} + b^{\text{final}}$$

this is a vector

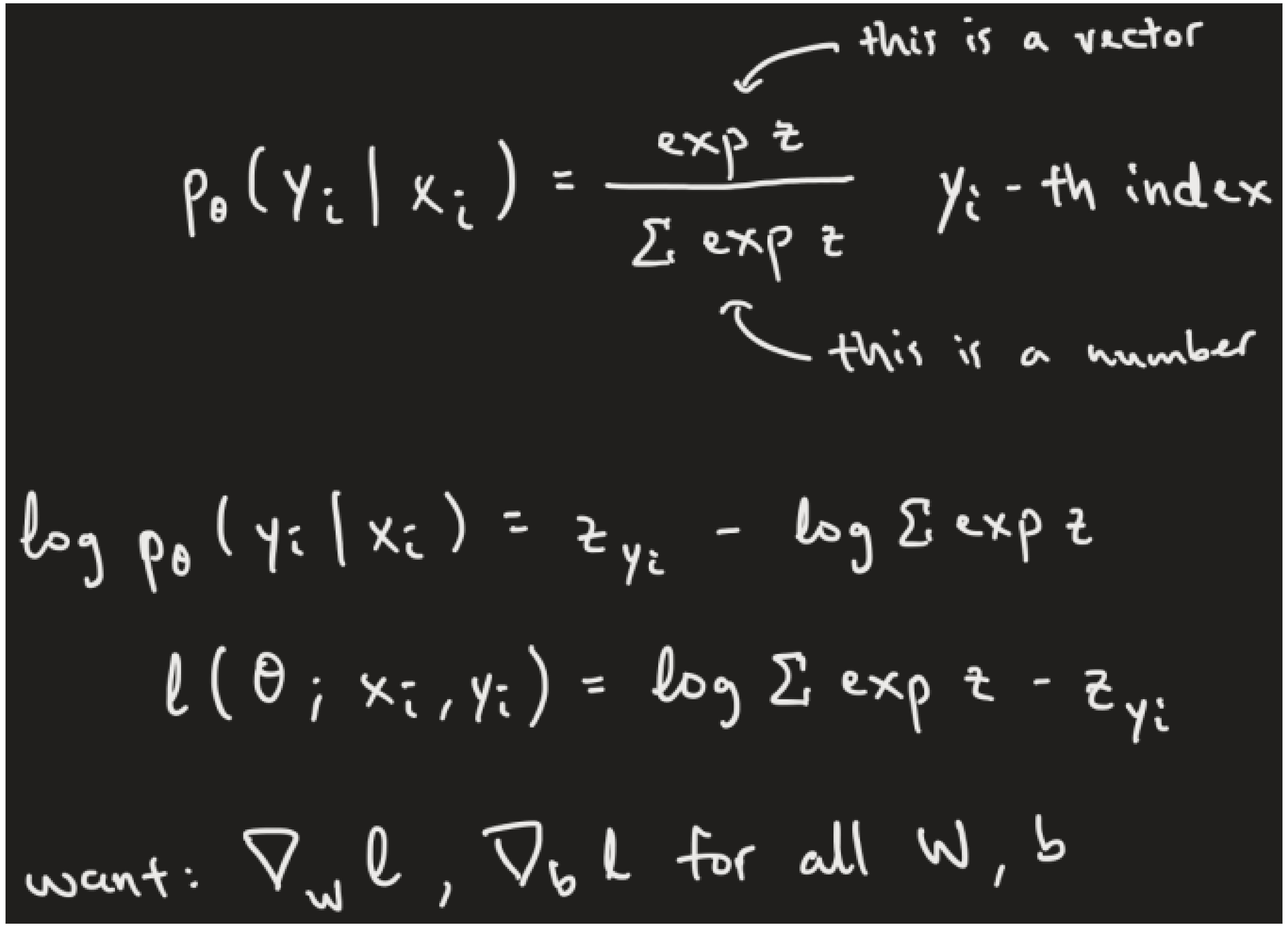
$$p_\theta(y_i \mid x_i) = \frac{\exp z}{\sum_i \exp z} \qquad y_i\text{-th index}$$

this is a number

# Backpropagation: the math



$\mathbf{z} = \mathbf{W}^{final}\mathbf{a}^{(2)} + \mathbf{b}^{final}$ represents our "logits" (aka inputs to softmax)



this is a vector

$$p_\theta(y_i | x_i) = \frac{\exp z}{\sum \exp z} \quad y_i\text{-th index}$$

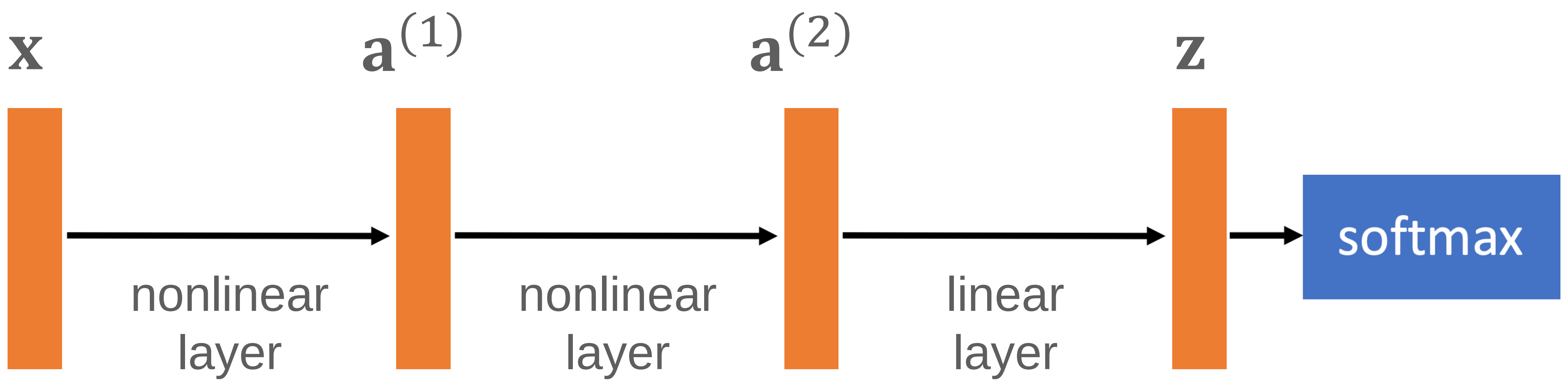
this is a number

$$\log p_\theta(y_i | x_i) = z_{y_i} - \log \sum \exp z$$

$$\ell(\theta; x_i, y_i) = \log \sum \exp z - z_{y_i}$$

want: $\nabla_W \ell$, $\nabla_b \ell$ for all $W$, $b$

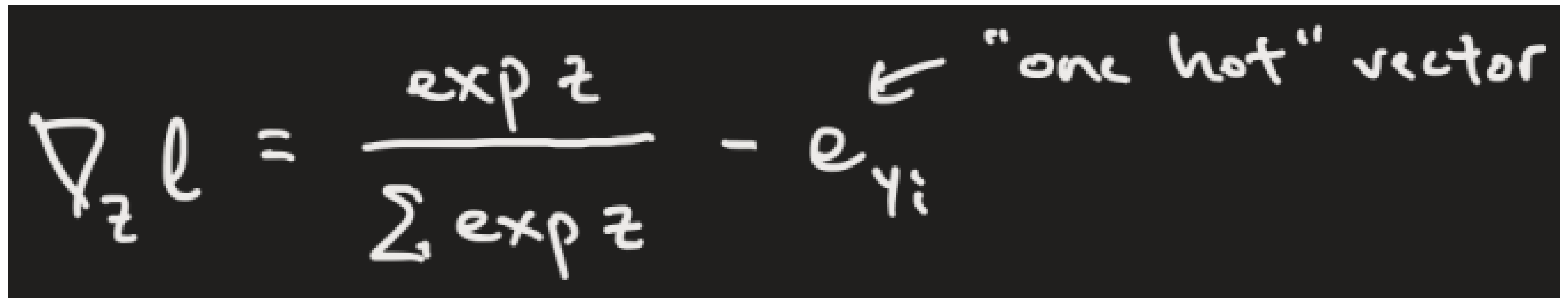
# Backpropagation: the math



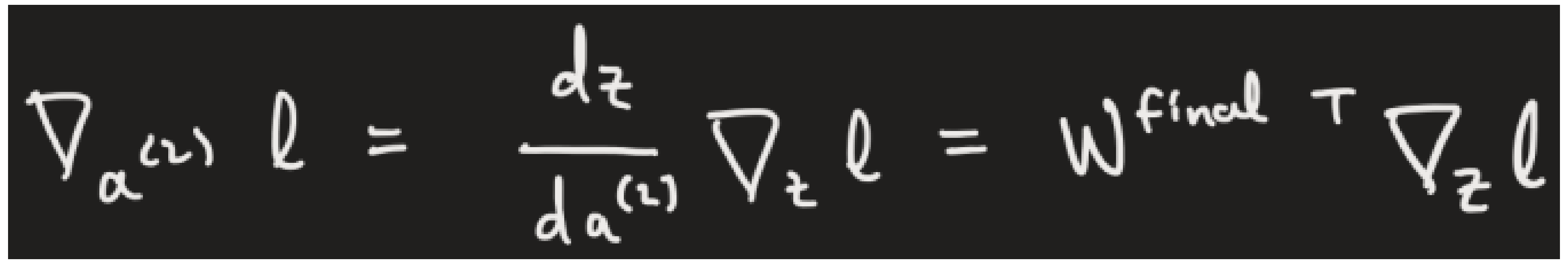
first let's look at $\nabla_{\mathbf{W}^{final}} \ell$ and $\nabla_{\mathbf{b}^{final}} \ell$

remember: $\ell = \log\sum\exp\mathbf{z} - \mathbf{z}_{y_i}$ ,and also $\mathbf{z} = \mathbf{W}^{final}\mathbf{a}^{(2)} + \mathbf{b}^{final}$

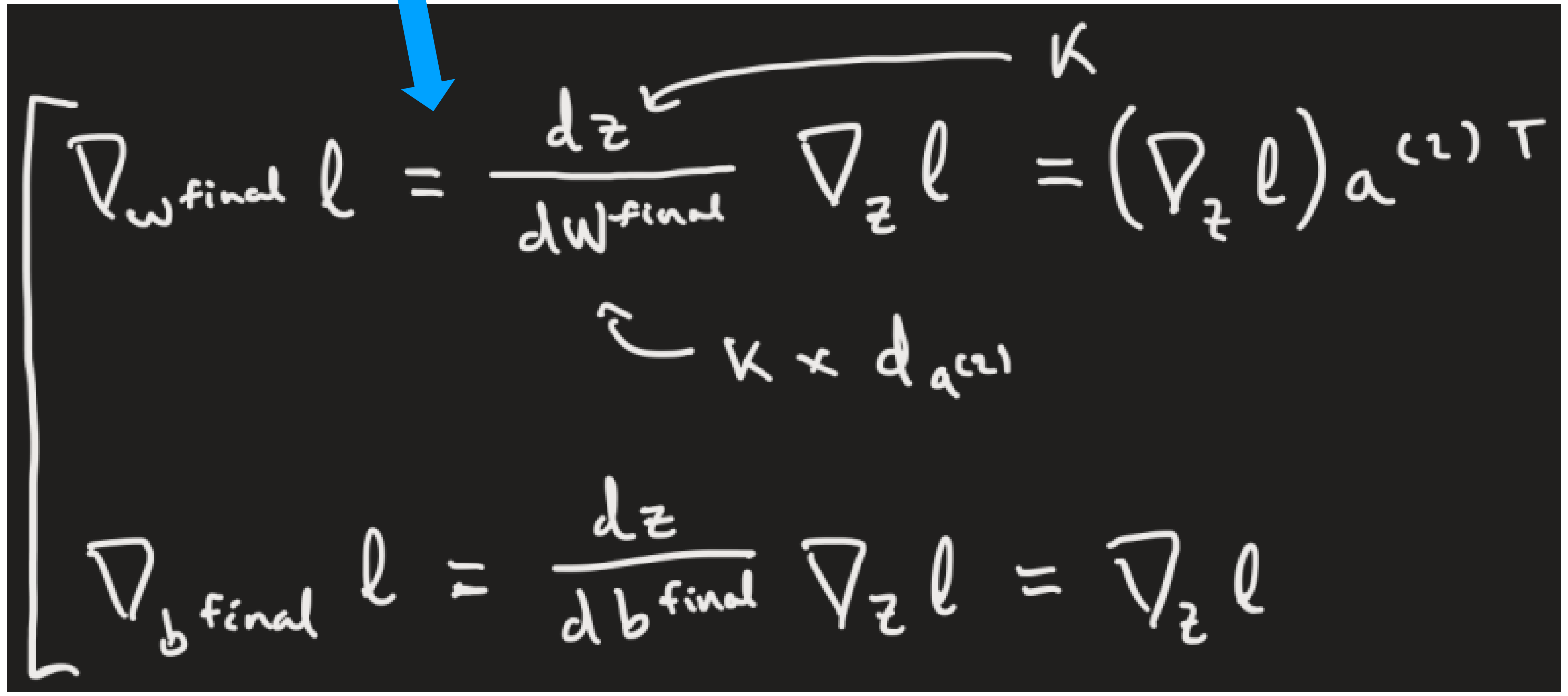
By multivariate chain rule: using the matrix shape conventions defined here: [link]



$$\nabla_z \ell = \frac{\exp z}{\sum \exp z} - e_{y_i} \leftarrow \text{"one hot" vector}$$

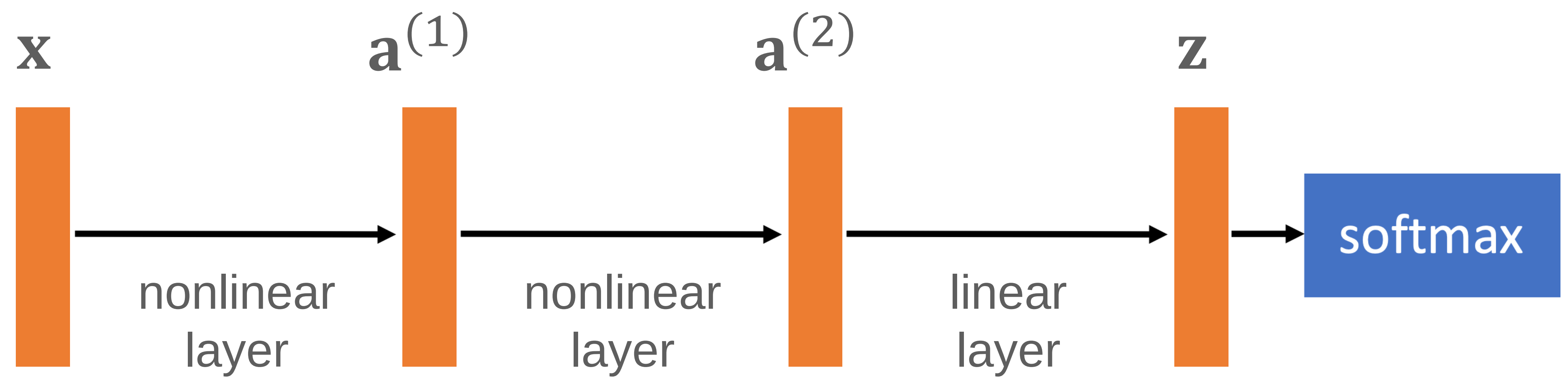


$$\nabla_{a^{(2)}} \ell = \frac{dz}{da^{(2)}} \nabla_z \ell = W^{final\,T} \nabla_z \ell$$



$$\left[\begin{aligned} \nabla_{W^{final}} \ell &= \frac{\overset{K}{dz}}{\underset{K \times d_{a^{(2)}}}{dW^{final}}} \nabla_z \ell = (\nabla_z \ell)\, a^{(2)\,T} \\ \nabla_{b^{final}} \ell &= \frac{dz}{db^{final}} \nabla_z \ell = \nabla_z \ell \end{aligned}\right.$$

# Backpropagation: the math

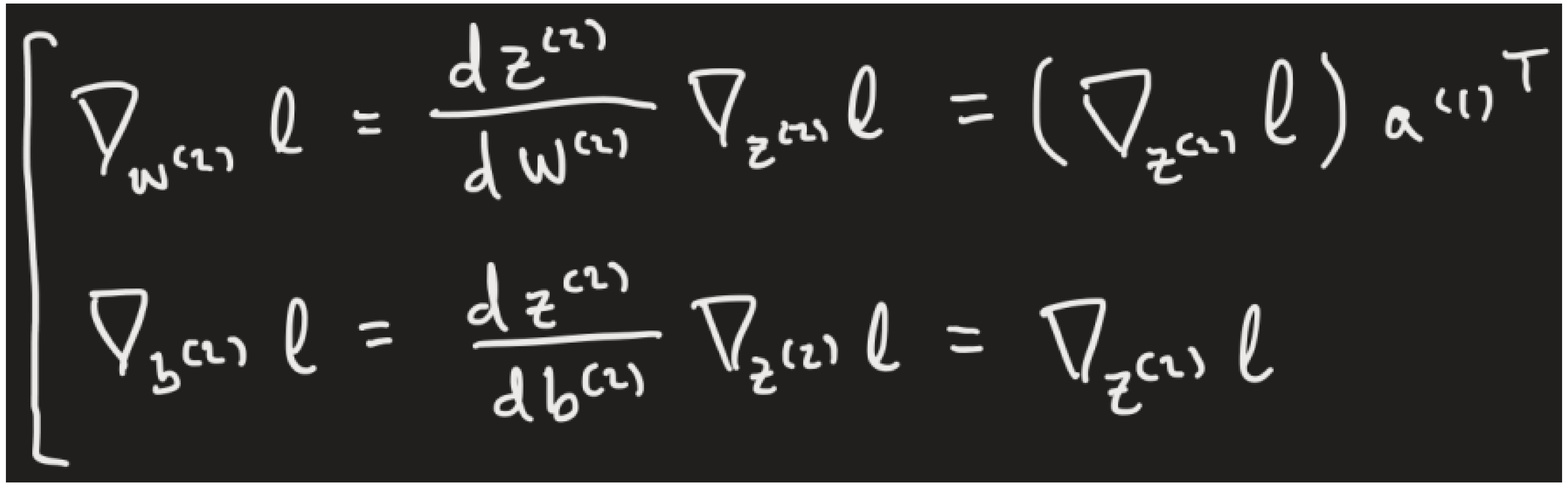


now let's look at $\nabla_{\mathbf{W}^{(2)}}\ell$ and $\nabla_{\mathbf{b}^{(2)}}\ell$

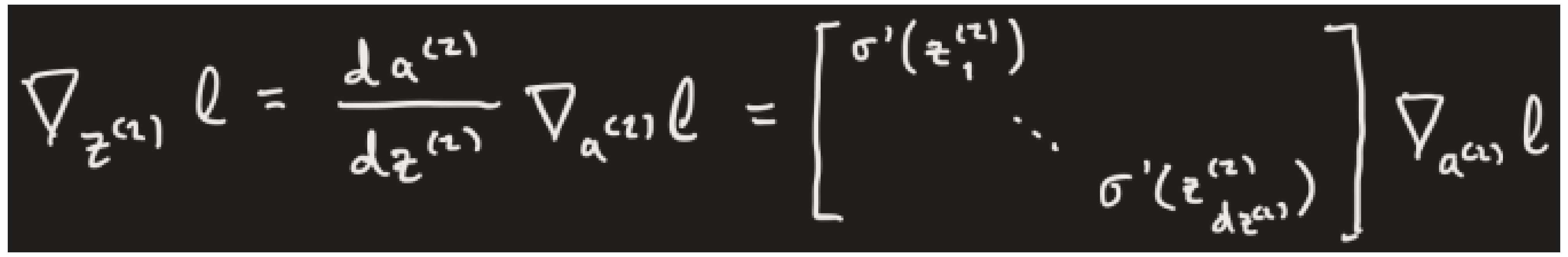
remember: $\mathbf{a}^{(2)} = \sigma(\mathbf{z}^{(2)})$, and also $\mathbf{z}^{(2)} = \mathbf{W}^{(2)}\mathbf{a}^{(1)} + \mathbf{b}^{(2)}$

How does the output of my layer change w.r.t. my layer's parameters?

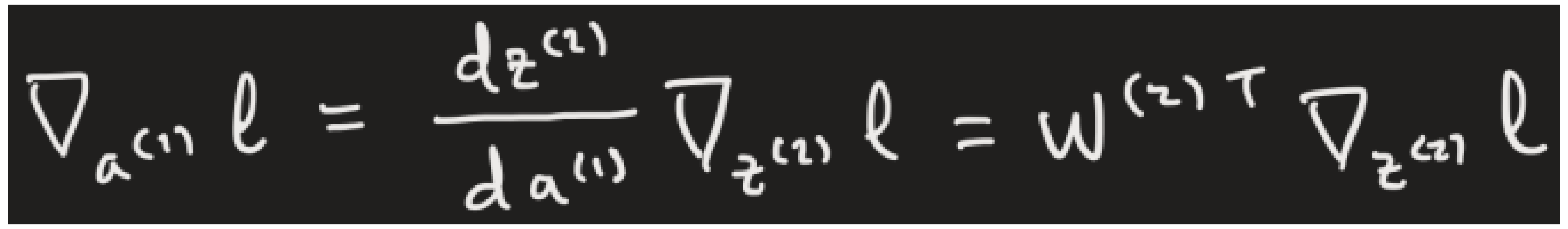
How does the loss change w.r.t. this layer's outputs?



$$\left[\begin{aligned} \nabla_{W^{(2)}} \ell &= \frac{dz^{(2)}}{dW^{(2)}} \nabla_{z^{(2)}} \ell = \left(\nabla_{z^{(2)}} \ell\right) a^{(1)T} \\ \nabla_{b^{(2)}} \ell &= \frac{dz^{(2)}}{db^{(2)}} \nabla_{z^{(2)}} \ell = \nabla_{z^{(2)}} \ell \end{aligned}\right.$$



$$\nabla_{z^{(2)}} \ell = \frac{da^{(2)}}{dz^{(2)}} \nabla_{a^{(2)}} \ell = \begin{bmatrix} \sigma'(z_1^{(2)}) & & \\ & \ddots & \\ & & \sigma'(z_{d_{z^{(2)}}}^{(2)}) \end{bmatrix} \nabla_{a^{(2)}} \ell$$



$$\nabla_{a^{(1)}} \ell = \frac{dz^{(2)}}{da^{(1)}} \nabla_{z^{(2)}} \ell = W^{(2)T} \nabla_{z^{(2)}} \ell$$

Observation: gradients for a given layer are functions of local things (eg inputs to layer during forward pass, and how the layer's outputs affect the loss gradient)
Suggests a dynamic-programming-like way to implement backpropagation in a way that mirrors the computation graph

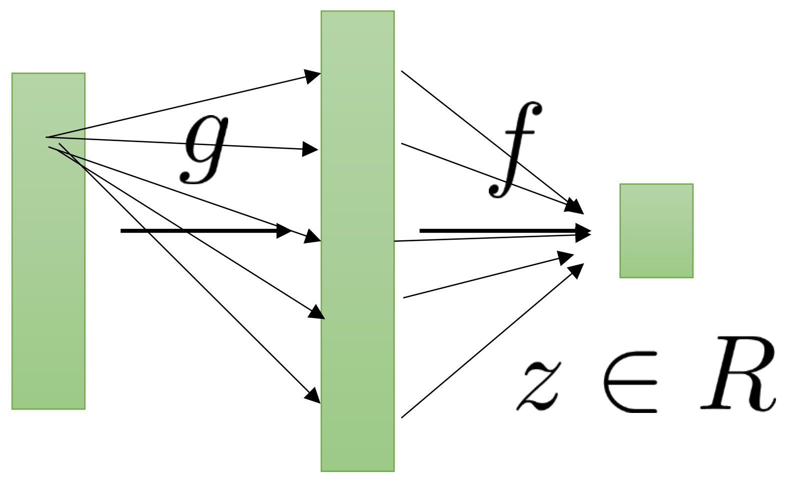
a pattern emerges… do you see it?

# 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 $f$



$z \in R$

$x \in R^n$ $y \in R^m$

## High-dimensional chain rule

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

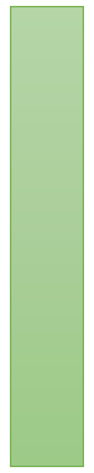
sum over all dimensions of $y$ row $1 \times m$ col $m \times 1$

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

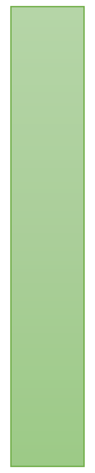
mat $n \times m$ col $m \times 1$

Row or column?

In this lecture:



$y \in R^m$

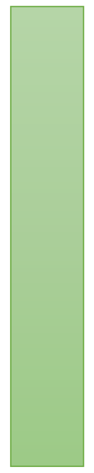


$\frac{dz}{dy} \in R^m$

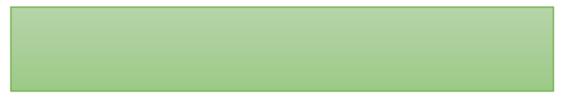
$\frac{dy}{dx} \in R^{n \times m}$

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

In some textbooks:



$y \in R^m$



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

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

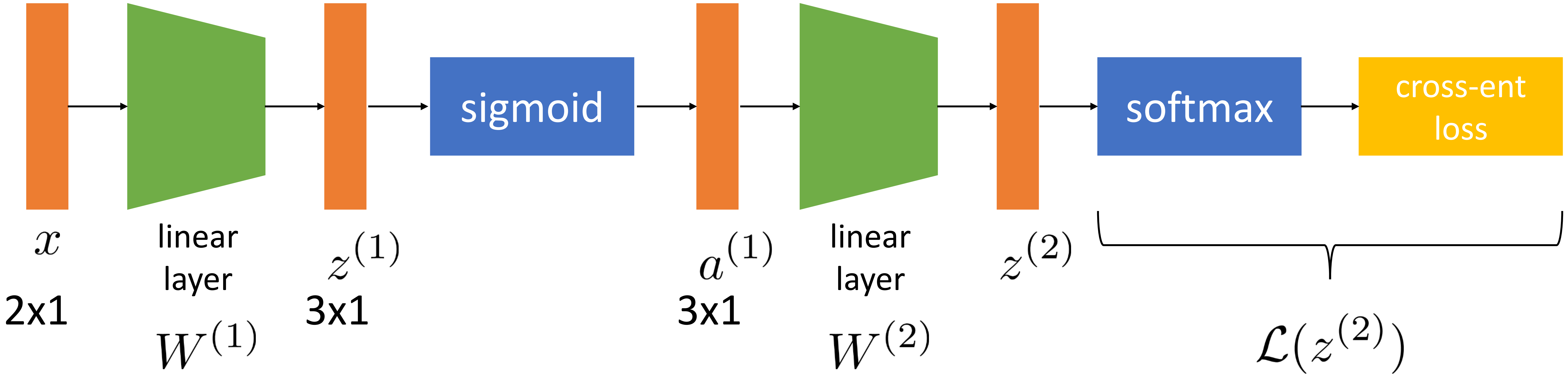
Just two different conventions!

Slides from Prof. Sergey Levine's SP21 CS 182, Lecture 05: [link]

# Chain rule for neural networks

A neural network is just a composition of functions

So we can use chain rule to compute gradients!



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

Slides from Prof. Sergey Levine's SP21 CS 182, Lecture 05: [link]

# 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...

Slides from Prof. Sergey Levine's SP21 CS 182, Lecture 05: [link]

# 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

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

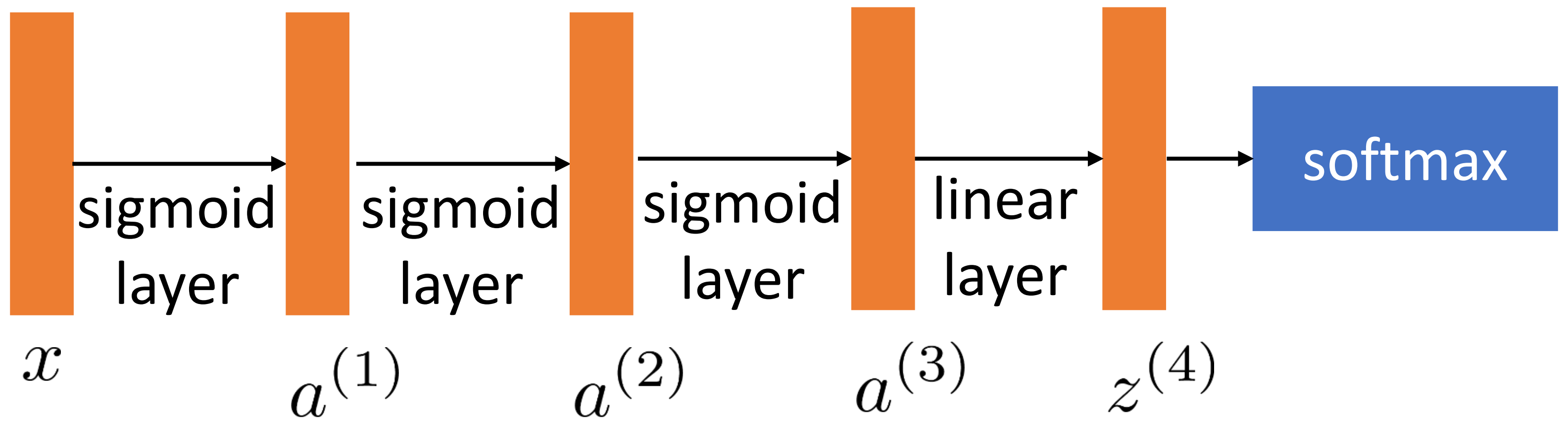
compute $\frac{da^{(1)}}{dz^{(1)}}\delta = \gamma$

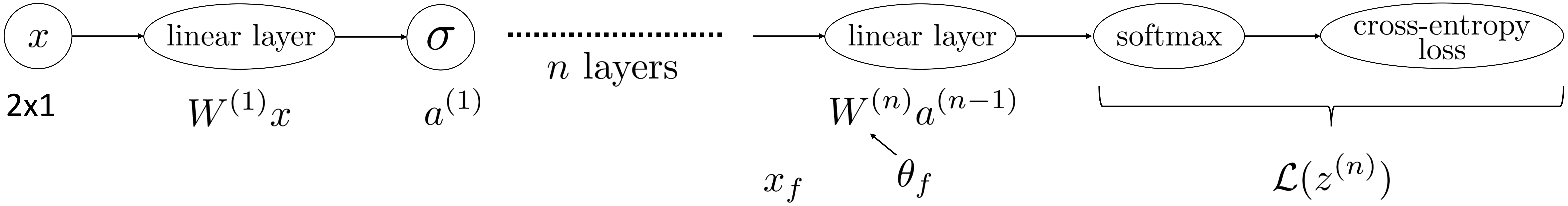
$$\frac{d\mathcal{L}}{dW^{(1)}} = \underbrace{\frac{dz^{(1)}}{dW^{(1)}}\gamma}_{\text{this product is cheap: } O(n^2)}$$

Slides from Prof. Sergey Levine's SP21 CS 182, Lecture 05: [link]

# The backpropagation algorithm

“Classic” version





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

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

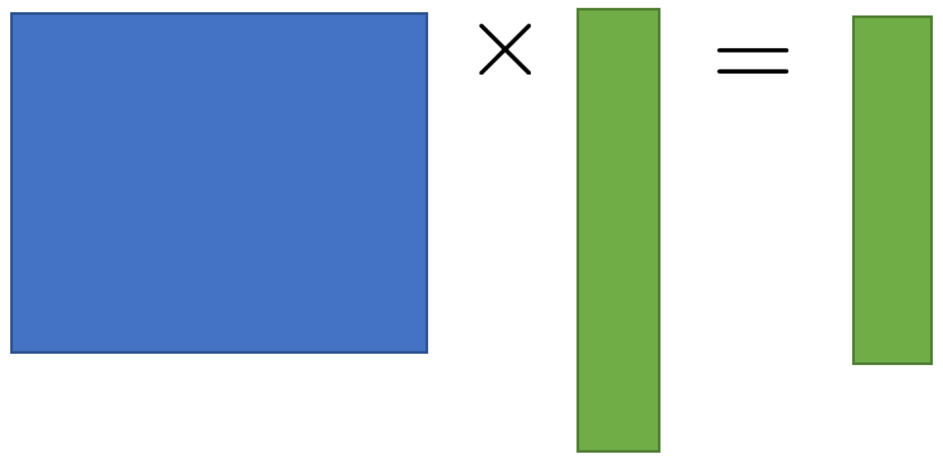
backward pass:

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

for each $f$ with input $x_f$ & params $\theta_f$ from end to start:

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

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



Slides from Prof. Sergey Levine's SP21 CS 182, Lecture 05: [link]

# Backpropagation: the summary

- First, we perform a forward pass and cache all the intermediate $\mathbf{z}^{(l)}$, $\mathbf{a}^{(l)}$
- Then, we work our way backwards to compute all the $\nabla_{\mathbf{W}^{(l)}}\ell$, $\nabla_{\mathbf{b}^{(l)}}\ell$
  - Going backwards allows us to reuse gradients that have already been computed
  - It also results in matrix-vector product computations, which are far more efficient than matrix-matrix product computations
- After all the gradients have been computed, we are ready to take a gradient step
  - Neural network optimization repeats this over and over — more on that next week

# Confused?

- Backpropagation can be tricky and unintuitive
- What can help is trying to work out the math on your own to see the patterns
- Implementing it for HW1 should also help solidify the concept
- But, most importantly: we don’t have to do it ourselves these days!
    - Deep learning libraries do it for us (ex: pytorch, tensorflow)