Lecture 4: Neural network basics

CS 182/282A (“Deep Learning”)

2022/01/31

Today's lecture

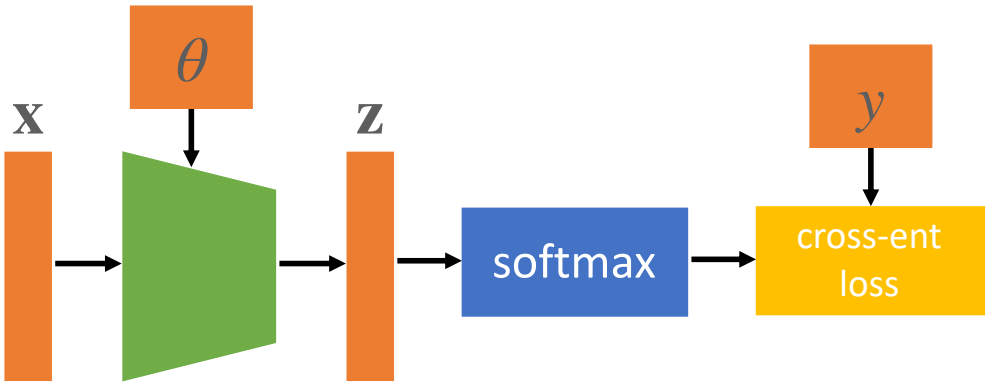
- Some of you may be thinking: "where are the deep neural networks??"
- Today, we'll start talking about our first basic neural network models
  - We'll put a full model together in this lecture, mathematically and diagrammatically
- We will then work through the **backpropagation** algorithm for computing 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), but next lecture you'll hear from Matt Johnson how deep learning libraries do this for you

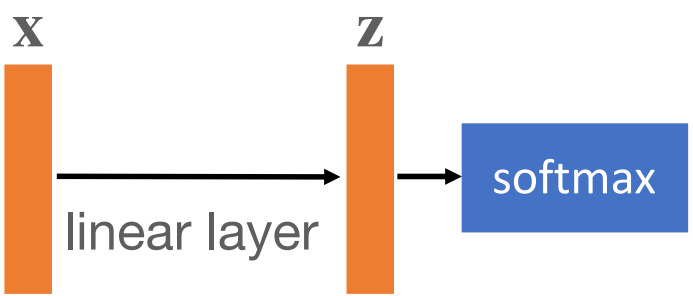
Recall: logistic regression

The “linear neural network”

- Given $\mathbf{x} \in \mathbb{R}^d$, define $f_\theta(\mathbf{x}) = \theta^\top \mathbf{x}$, where θ is a $d \times K$ matrix
- Then, for class $c \in \{0, \ldots, K-1\}$, we have $p_\theta(y = c \mid \mathbf{x}) = \text{softmax}(f_\theta(\mathbf{x}))_c$
  - Remember: $\text{softmax}(f_\theta(\mathbf{x}))_c = \dfrac{\exp f_\theta(\mathbf{x})_c}{\sum_{i=0}^{K-1} \exp f_\theta(\mathbf{x})_i}$
- Loss function: $\ell(\theta; \mathbf{x}, y) = -\log p_\theta(y \mid \mathbf{x})$

A diagram for logistic regression

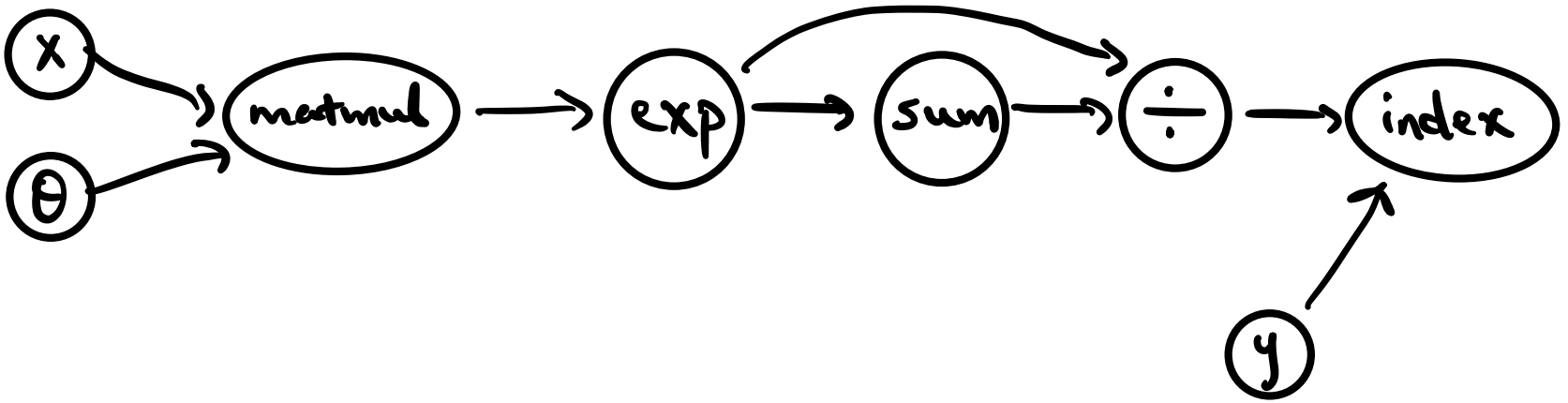




- Often, we will simplify this diagram:
  - Omit the θ box, the parameters are implicit in the diagram
  - Omit the layer box entirely! Denote it with just the arrow
  - Omit the loss box at the end, if we're drawing "just the model"

Another type of drawing: computation graphs

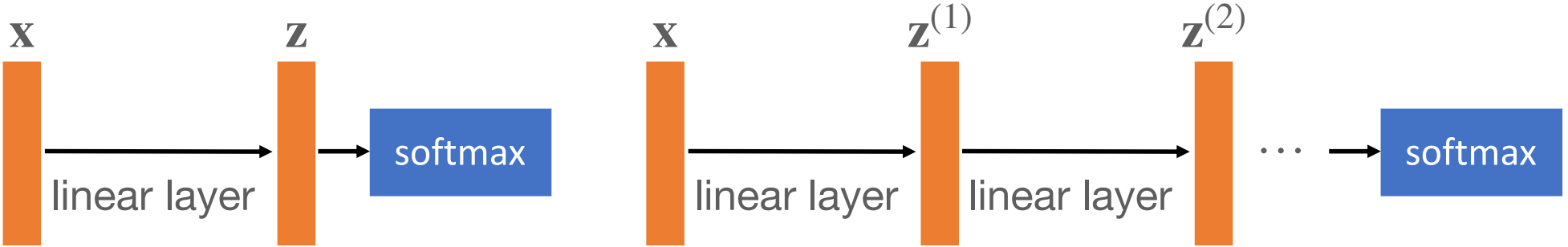
computation graphs are more detailed, rigorous graphical representations



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

Neural networks: attempt #1

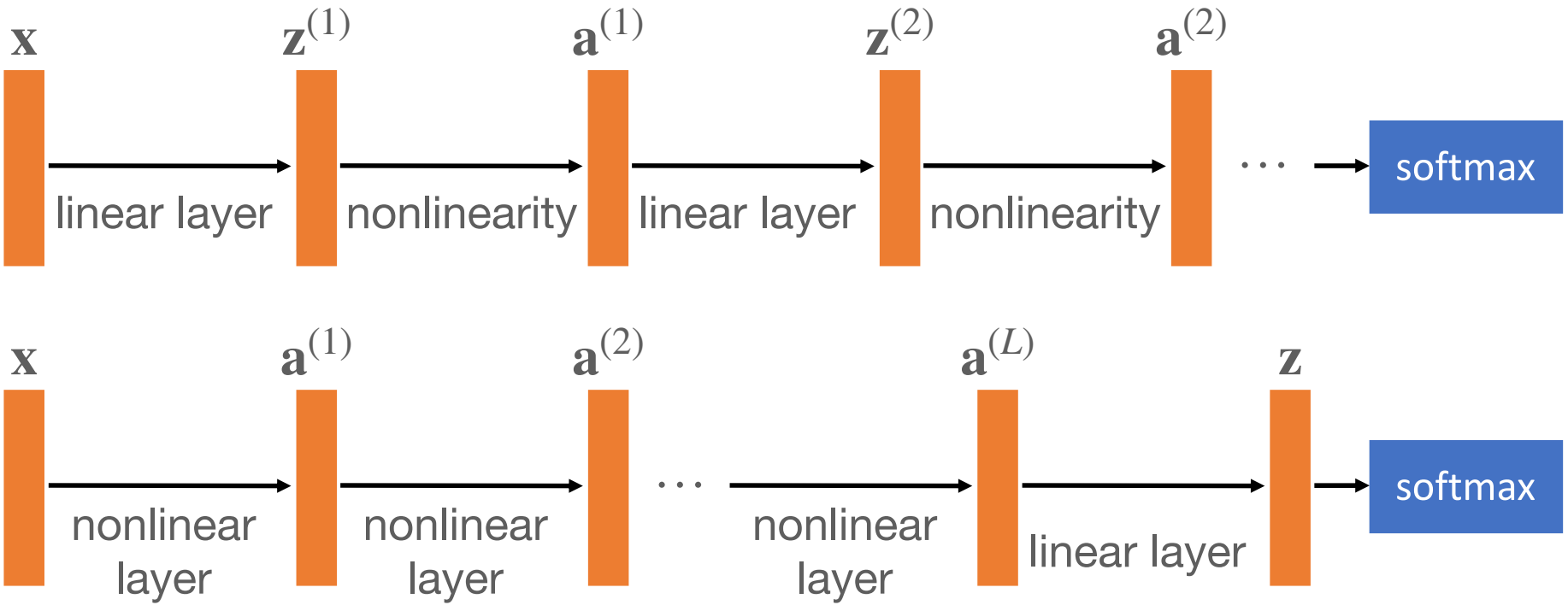
- Our drawing of logistic regression suggests that it is a “single layer model”
  - Are neural networks just more of these layers stacked on top of each other?
  - What’s the issue with this?
  - Composing linear transformations together is still linear!



Making neural networks nonlinear

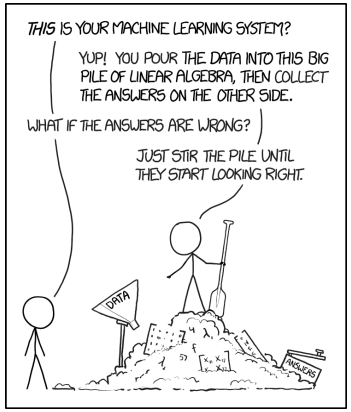
- One of the main things that makes neural networks great is that they can represent complex non linear functions
- How? The canonical answer: add **nonlinearities** after every linear layer
  - Also called **activation functions**
  - Basically always *element wise* functions on the linear layer output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

Neural networks: attempt #2



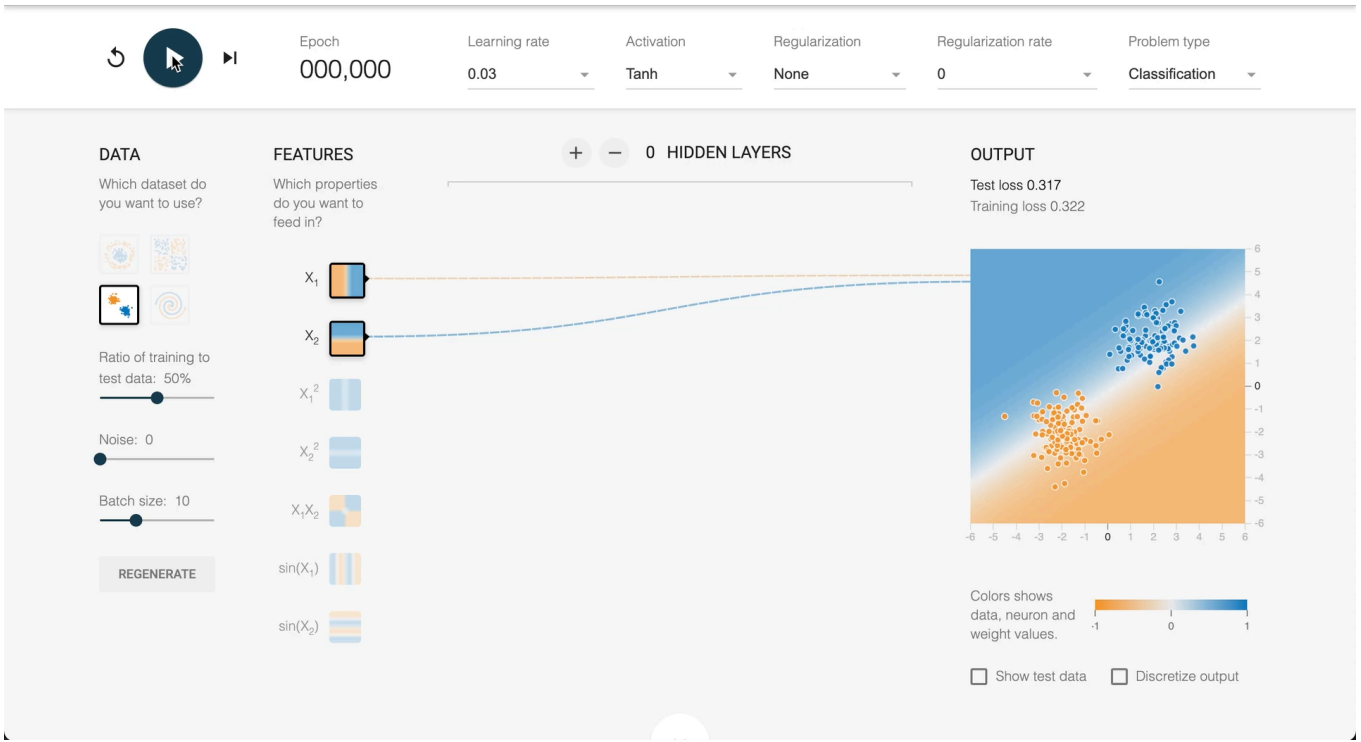
What function is this?

- θ represents all our parameters, e.g., $[\mathbf{W}^{(1)}, \mathbf{b}^{(1)}, \ldots, \mathbf{W}^{(L)}, \mathbf{b}^{(L)}, \mathbf{W}^{\text{final}}, \mathbf{b}^{\text{final}}]$
- If our neural network has parameters θ and L **hidden layers**, then it represents the function $f_\theta(\mathbf{x}) = \text{softmax}(A^{\text{final}}(\sigma(A^{(L)}(\ldots\sigma(A^{(1)}(\mathbf{x}))\ldots))))$
  - σ 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



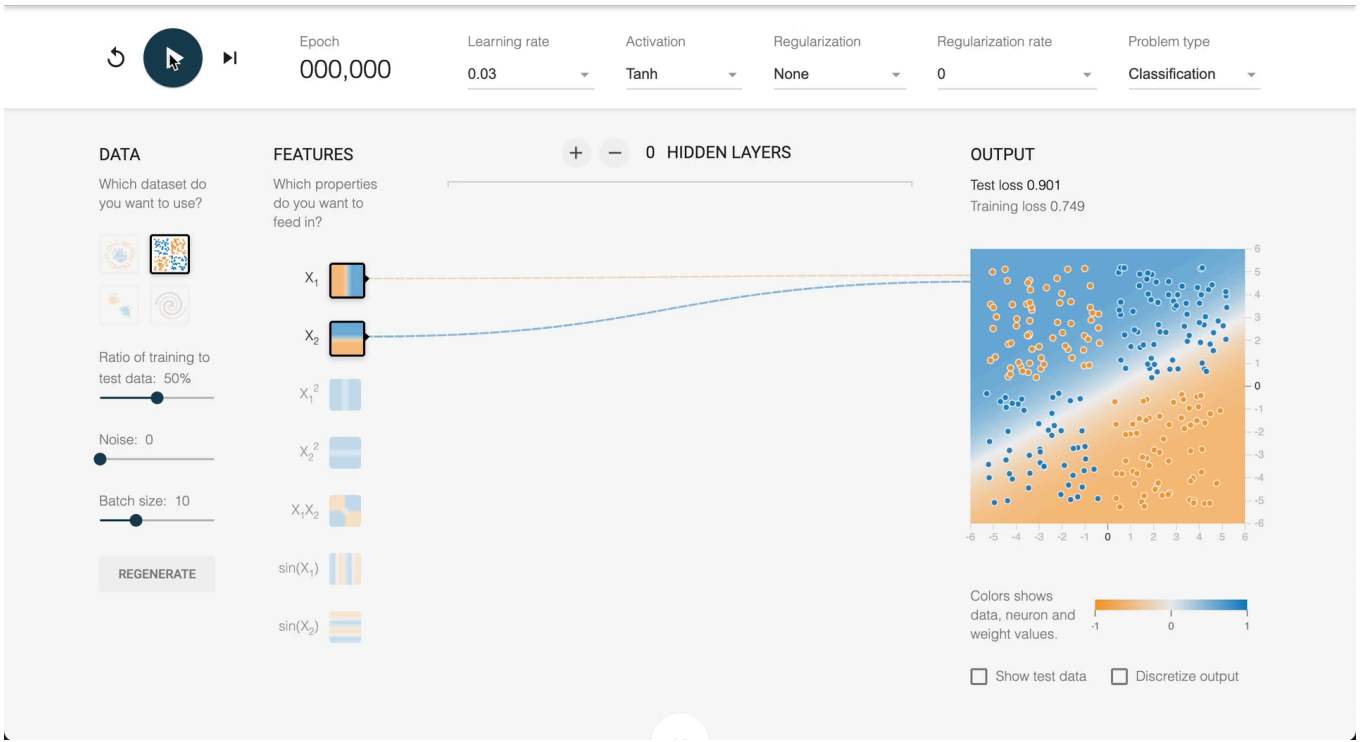
Visualizing neural network functions

https://playground.tensorflow.org/



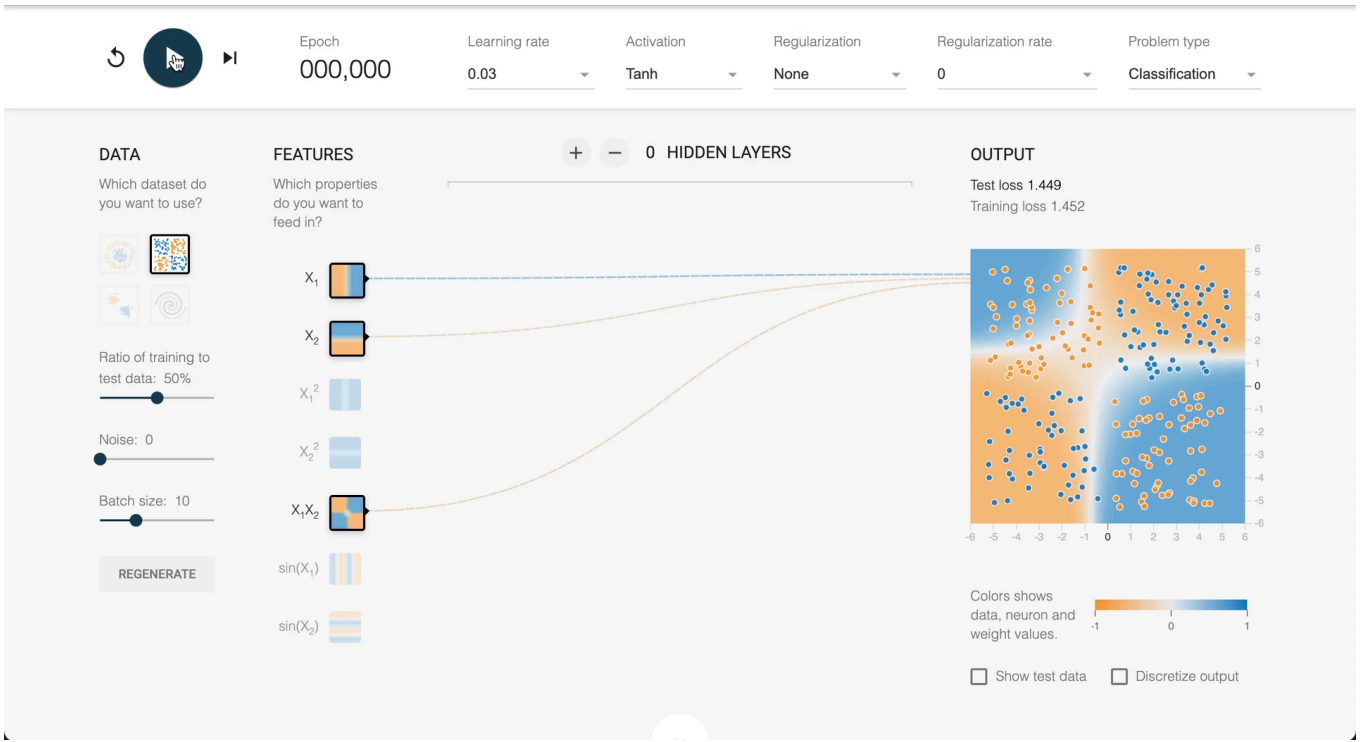
Visualizing neural network functions

https://playground.tensorflow.org/



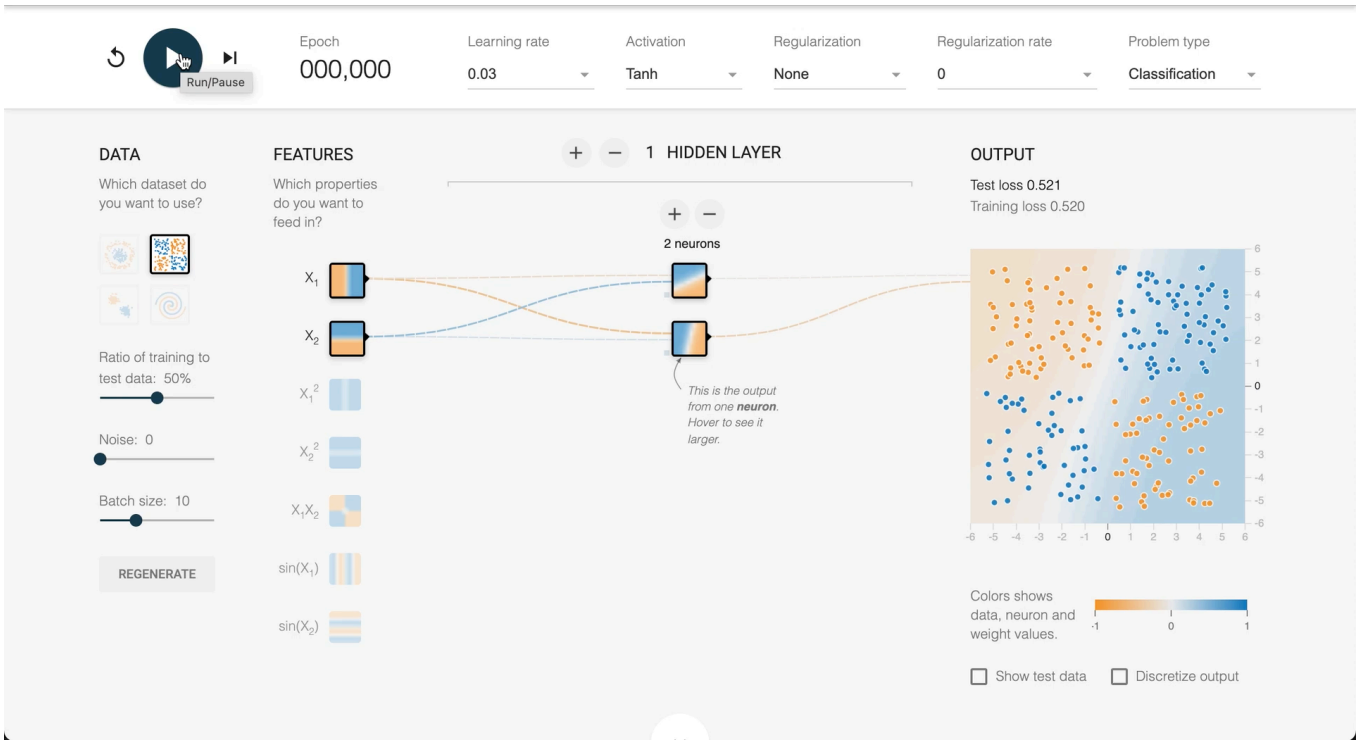
Visualizing neural network functions

https://playground.tensorflow.org/



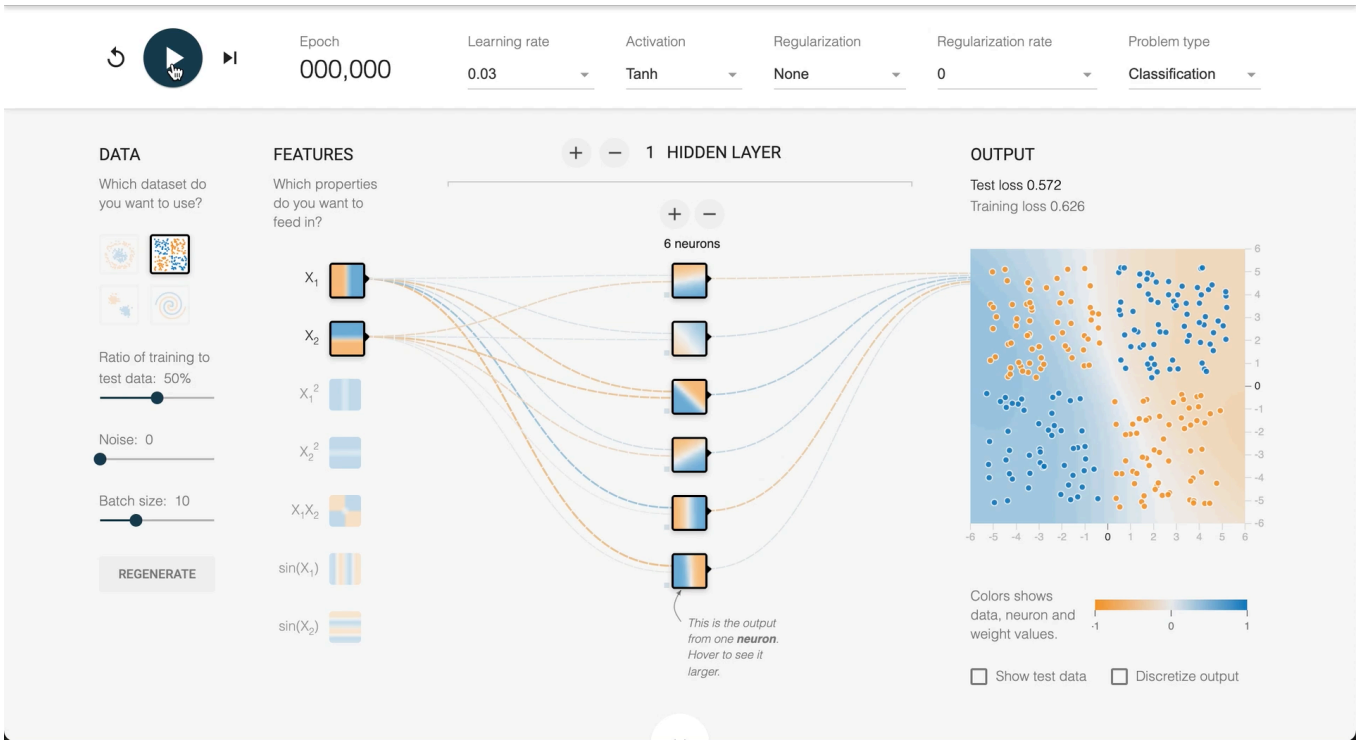
Visualizing neural network functions

https://playground.tensorflow.org/



Visualizing neural network functions

https://playground.tensorflow.org/

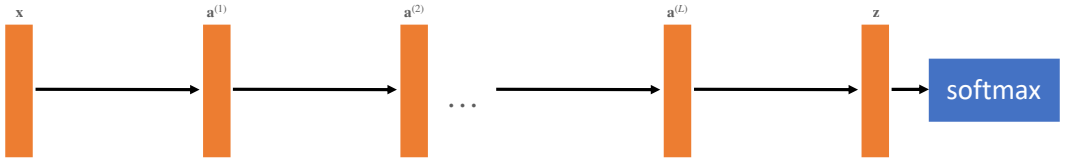


The backpropagation algorithm

Remember: the machine learning method

(or, at least, the deep learning method)

1. Define your **model**



2. Define your **loss function** $\ell(\theta; \mathbf{x}, y) = -\log p_\theta(y \mid \mathbf{x})$ ("cross-entropy")

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

4. Run it on a big GPU

wait… we need gradients!

What gradients do we need?

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

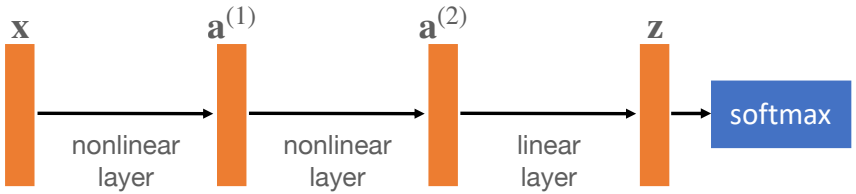
Finite differences

- 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 $\epsilon \to 0$
- 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

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

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

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

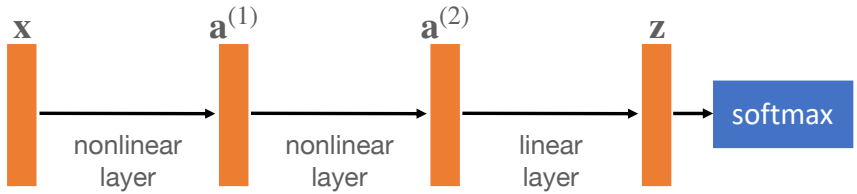
this is a vector

$$p_\theta(y_i \mid x_i) = \frac{\exp z}{\sum \exp z}$$

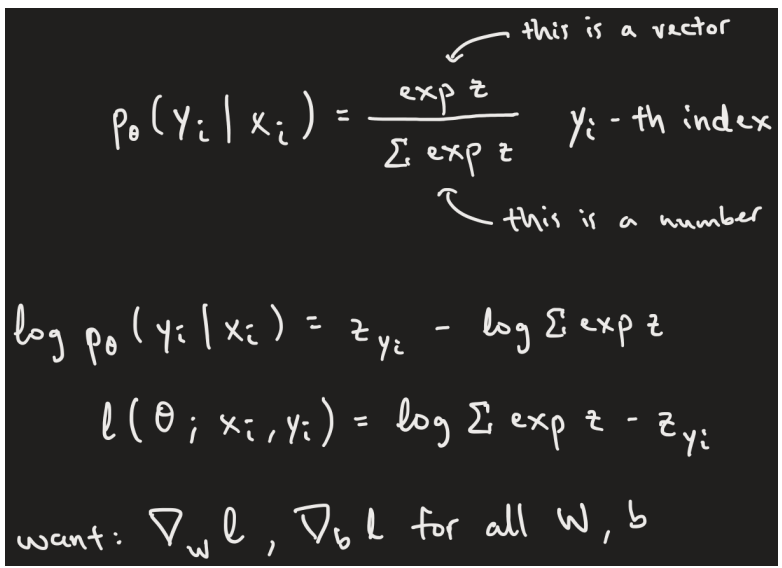
y_i - th index

this is a number

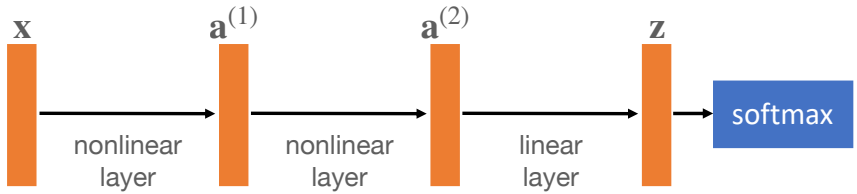
Backpropagation: the math



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



Backpropagation: the math



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

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

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

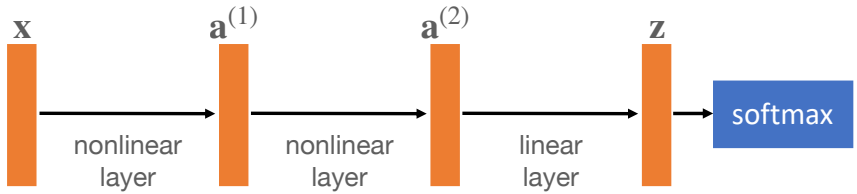
$$\nabla_{a^{(2)}} \ell = \frac{dz}{da^{(2)}} \nabla_z \ell = W^{\text{final}\,T} \nabla_z \ell$$

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

(dz ← K; $K \times d_{a^{(2)}}$)

$$\nabla_{b^{\text{final}}} \ell = \frac{dz}{db^{\text{final}}} \nabla_z \ell = \nabla_z \ell$$

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

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

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

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

$$\nabla_{b^{(2)}} \ell = \frac{dz^{(2)}}{db^{(2)}} \nabla_{z^{(2)}} \ell = \nabla_{z^{(2)}} \ell$$

a pattern emerges… do you see it?

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
  - Next lecture, Matt Johnson will come tell you how that's done in a general and efficient way — a can't miss lecture!