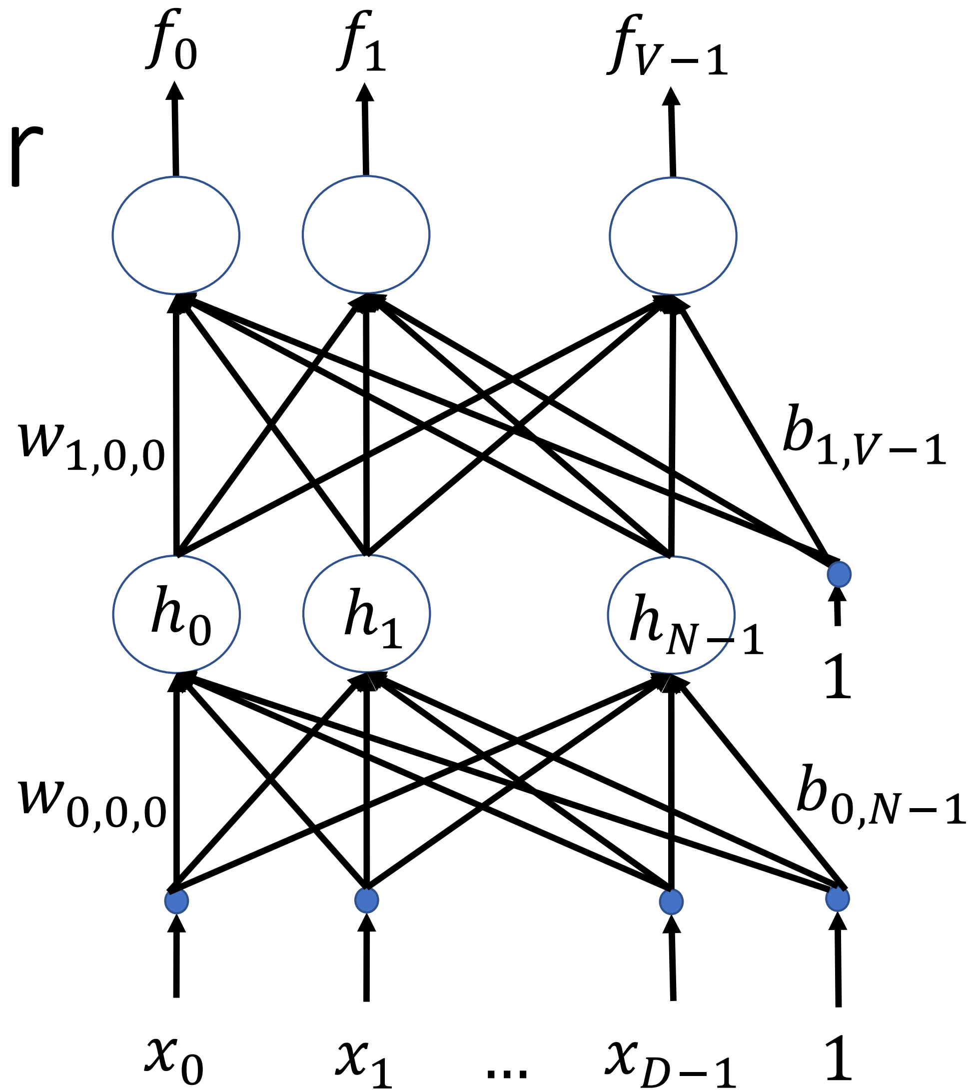
Lecture 9: Two-Layer Neural Nets

Mark Hasegawa-Johnson

2/2022

Lecture slides CC0:



Outline

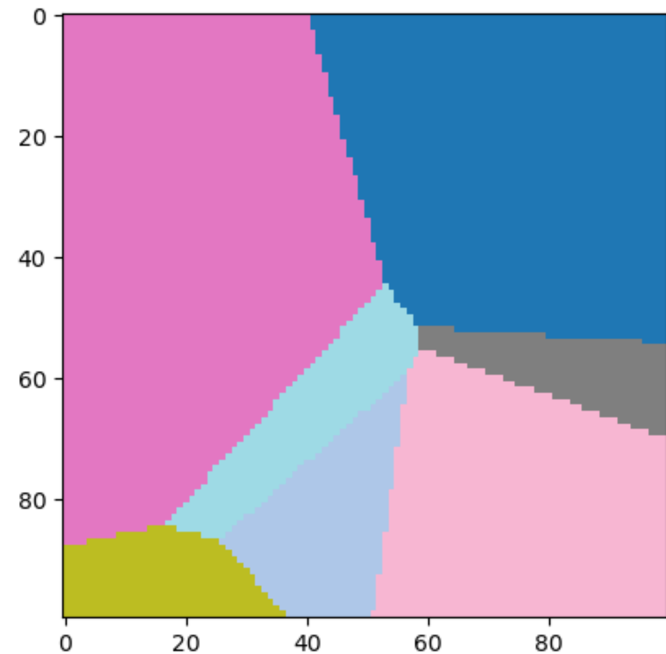
- From linear to nonlinear classifiers
- Training a two-layer network: Back-propagation
- Other nonlinearities

Linear classifier

Review: a linear classifier computes

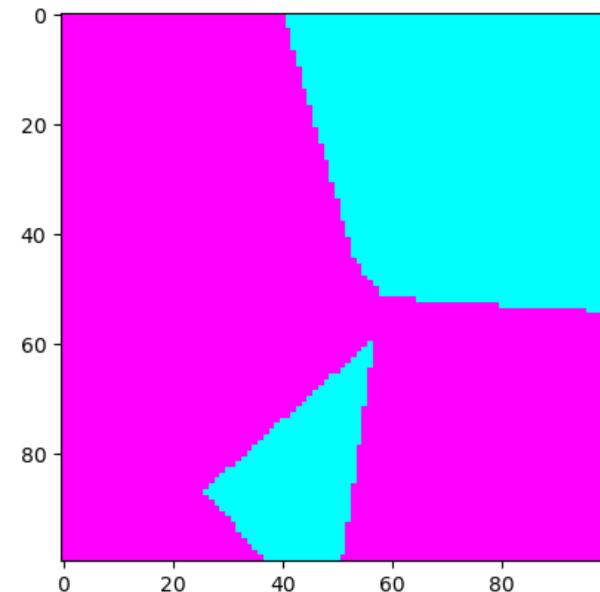
$$f(x) = \underset{k}{\operatorname{argmax}}\, w_k @ x + b_k$$

The resulting classifier divides the x-space into Voronoi regions: convex regions with piece-wise linear boundaries



Nonlinear classifier

- Not all classification problems have convex decision regions with PWL boundaries!
- Here's an example problem in which class 0 (blue) includes values of x near $x = [0.8,0]$, but it also includes some values of x near $x = [0.4,0.9]$
- You can't compute this function using
$$f(x) = \underset{k}{\operatorname{argmax}}\, w_k @ x + b_k$$



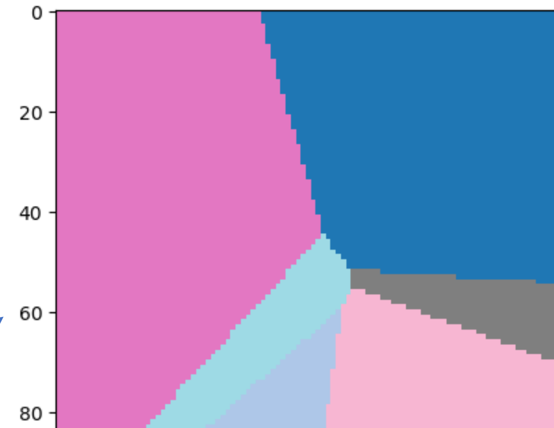
Piece-wise linear classifier

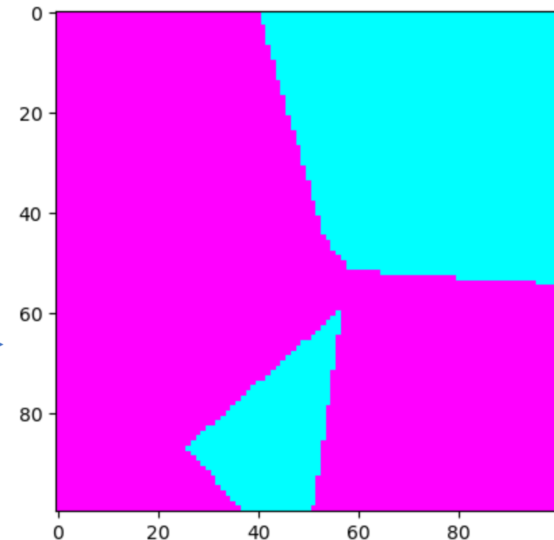
- SOLUTION: Merge the decision regions!
- First, perform a 20-class classification using a formula like

$$h(x) = \underset{k}{\operatorname{argmax}}\, w_k@x + b_k$$

- Then just threshold h(x) to get f(x):

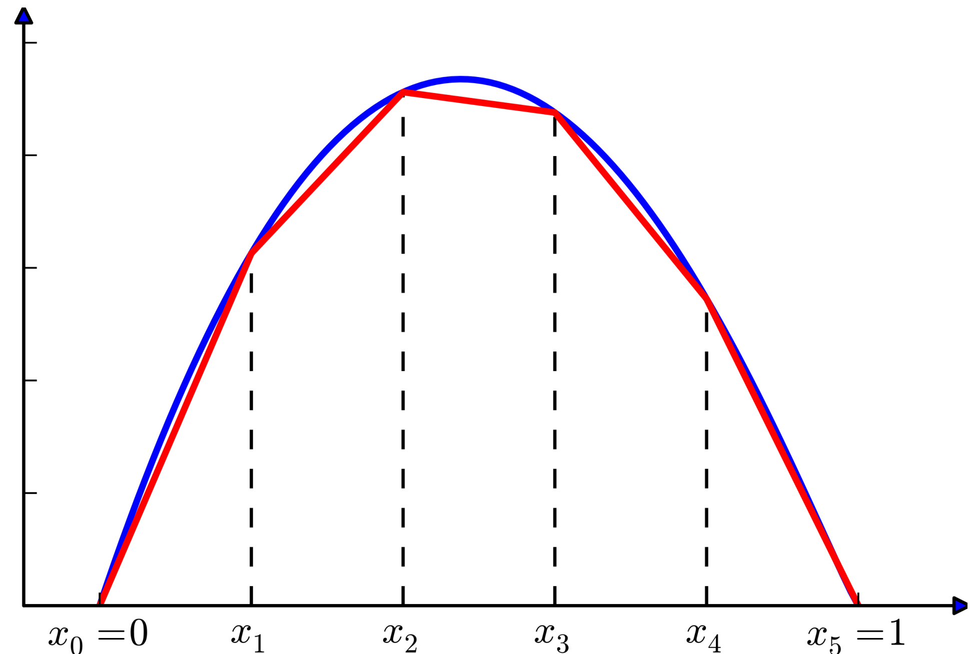
$$f(x) = \begin{cases} 1 & h(x) > 9.5 \\ 0 & h(x) < 9.5 \end{cases}$$





From piece-wise-linear to nonlinear

- We can approximate any nonlinear classifier using a PWL classifier
- In the limit, as the number of hidden nodes goes to infinity, the approximation becomes provably perfect



Public domain image, Krishnavedala, 2011

Piece-wise linear classifier by adding argmax nodes

- The hidden layer could use argmax to divide the input space into Voronoi regions

$$h(x) = \underset{k}{\operatorname{argmax}}\, w_k@x + b_k$$

- … then we could add and threshold, to merge those regions

$$f(x) = u(h(x) - 9.5)$$

… where u(x) is called the "unit step function," and is defined as

$$u(x) = \begin{cases} 1 & x > 0 \\ 0 & x < 0 \end{cases}$$

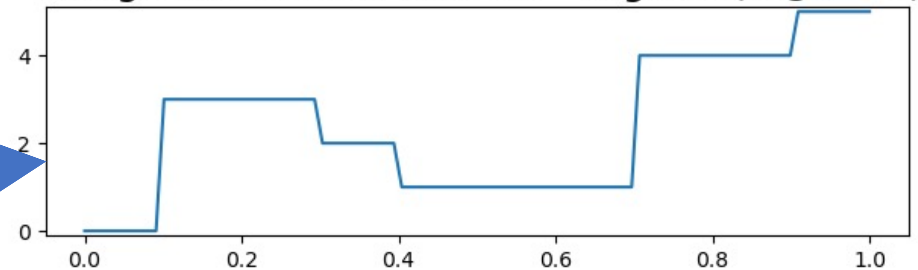
How to add argmax nodes: a 1d example

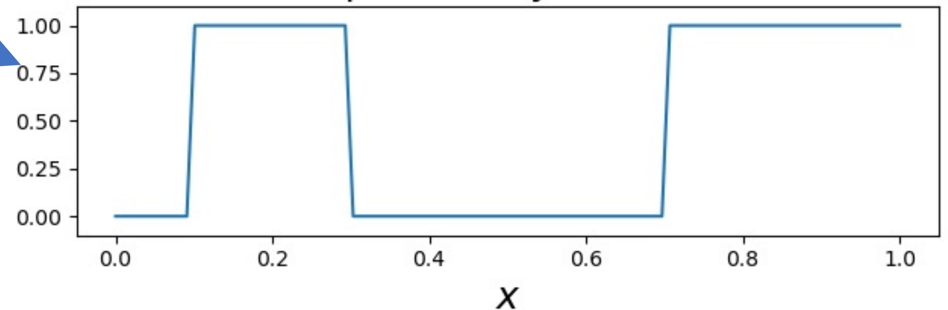
Here's an example with a 1-dimensional x input.

$$h(x) = \underset{k}{\text{argmax}}\, w_k x + b_k$$

$$f(x) = u(h(x) - 1.5)$$

The problem with this method is that neither argmax nor unit step are differentiable, so we can't train this classifier!

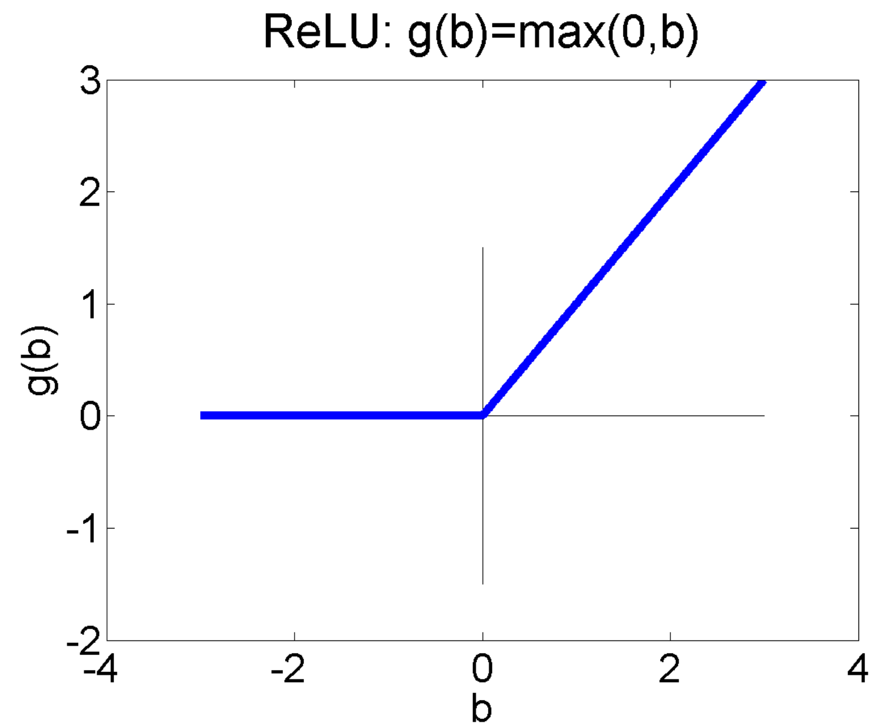




A differentiable and simple alternative: ReLU

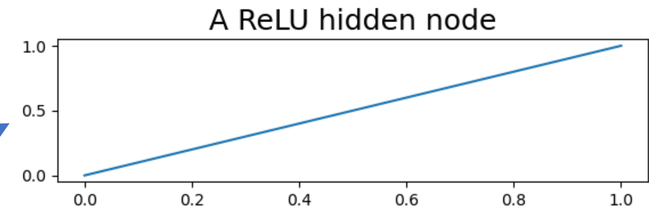
If the goal is PWL classification boundaries, we can achieve that by using hidden nodes that are the simplest possible PWL function: a rectified linear unit, or ReLU:

$$ReLU(b) = \max(0, b)$$

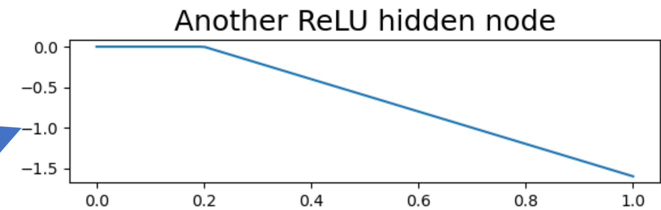


ReLU hidden nodes

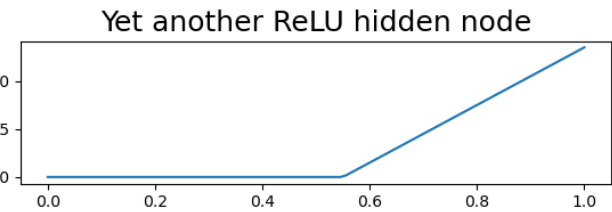
$$w_{1,0}h_0(x) = ReLU\left(w_{0,0}x + b_{0,0}\right)$$



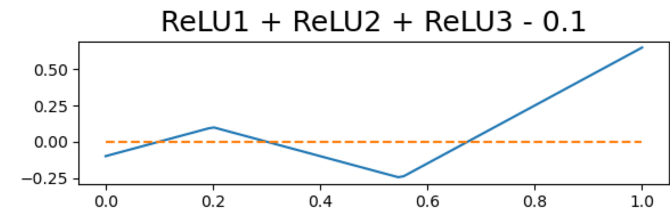
$$w_{1,1}h_1(x) = -2ReLU\left(w_{0,1}x + b_{0,1}\right)$$

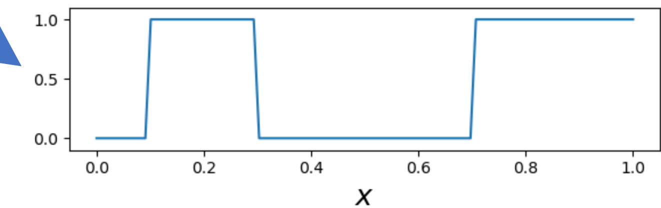


$$w_{1,2}h_2(x) = 3ReLU\left(w_{0,2}x + b_{0,2}\right)$$



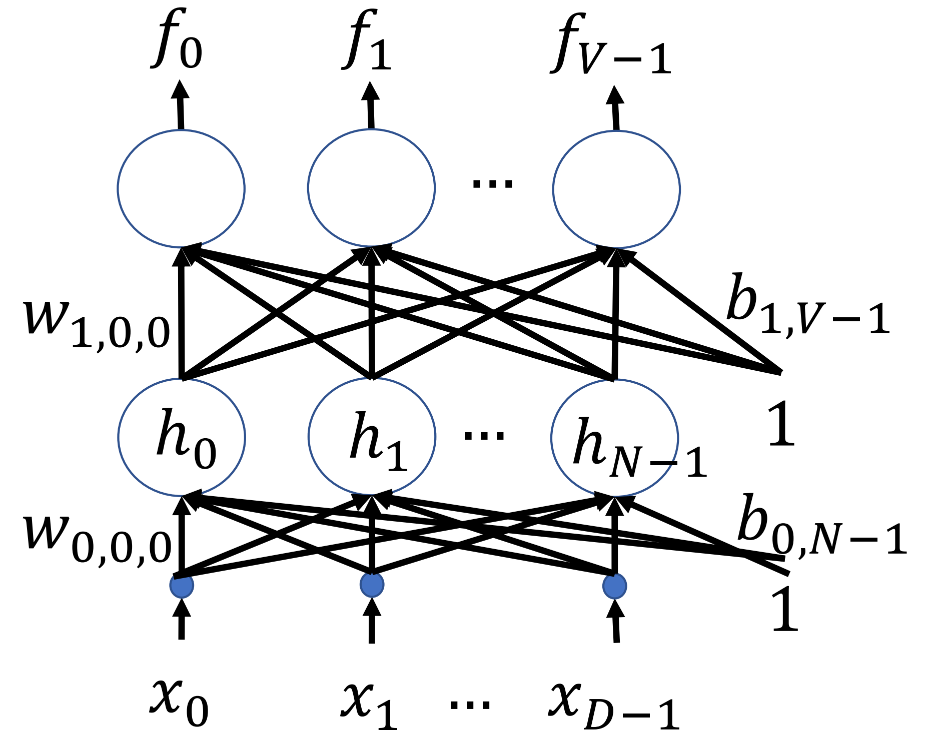
$$f(x) \\ = u\left(w_{1,0}h_0 + w_{1,1}h_1 + w_{1,2}h_2 - 0.1\right)$$





Multi-layer neural net: Basic notation

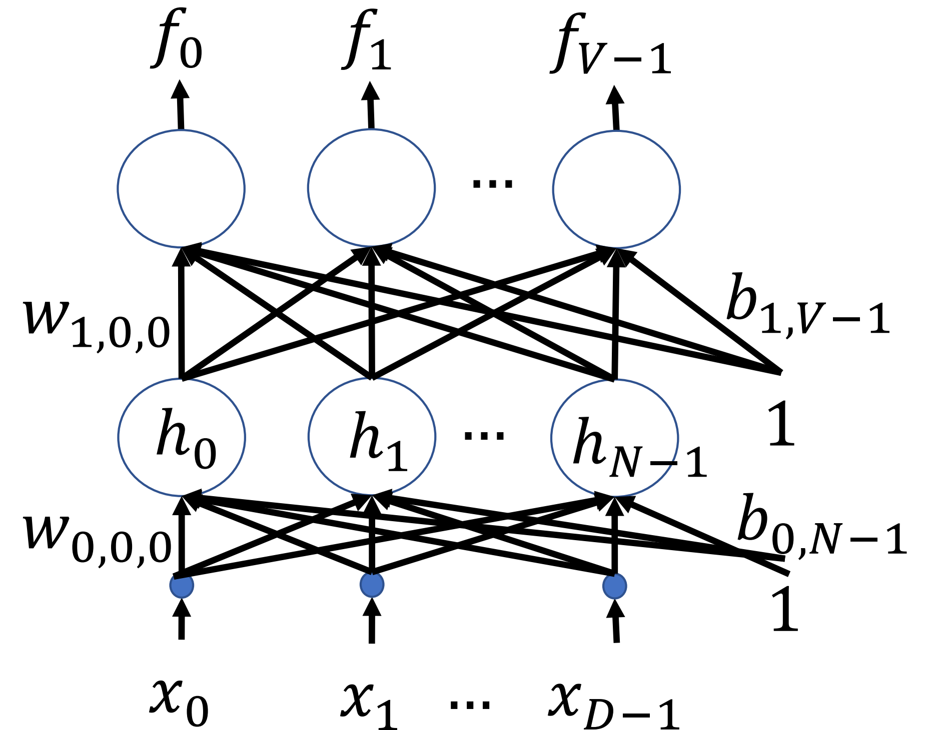
- Output vector:
  $$f = [f_0, \dots, f_{V-1}]$$
- Hidden nodes vector:
  $$h = [h_0, \dots, h_{N-1}]$$
- Input vector:
  $$x = [x_0, \dots, x_{D-1}]$$
- Layer l bias vector:
  $$b_l = \left[b_{l,0}, b_{l,1}, \dots\right]$$
- Layer l weight matrix:
  $$w_l = \begin{bmatrix} w_{l,0,0} & w_{l,0,1} & \cdots \\ w_{l,1,0} & w_{l,1,1} & \cdots \\ \vdots & \vdots & \ddots \end{bmatrix}$$



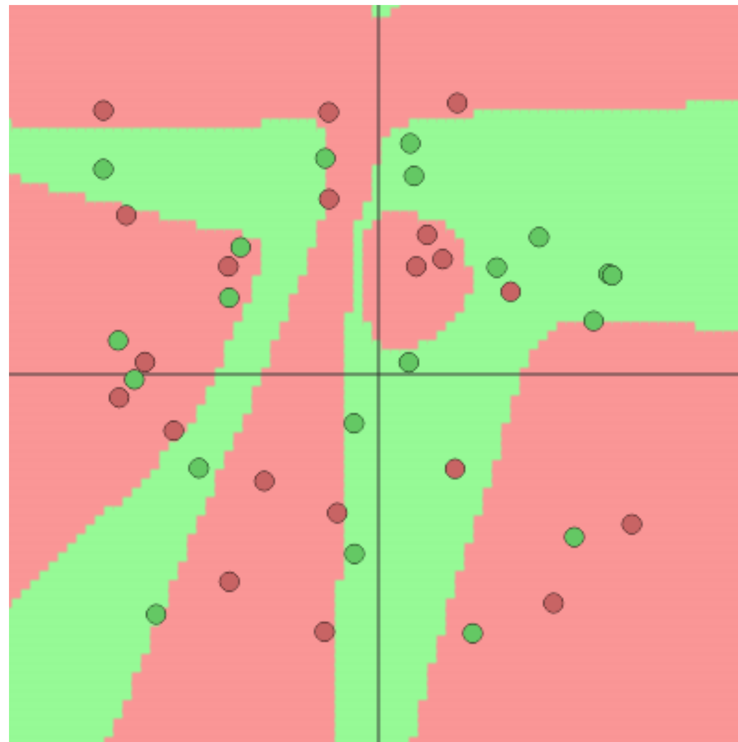
Multi-layer neural network: Each layer is a matrix multiplication followed by a nonlinearity

$$f = \text{softmax}(w_1@h + b_1)$$

$$h = ReLU(w_0@x + b_0)$$



Approximating an arbitrary nonlinear boundary using a two-layer network



https://cs.stanford.edu/people/karpathy/convnetjs/demo/classify2d.html

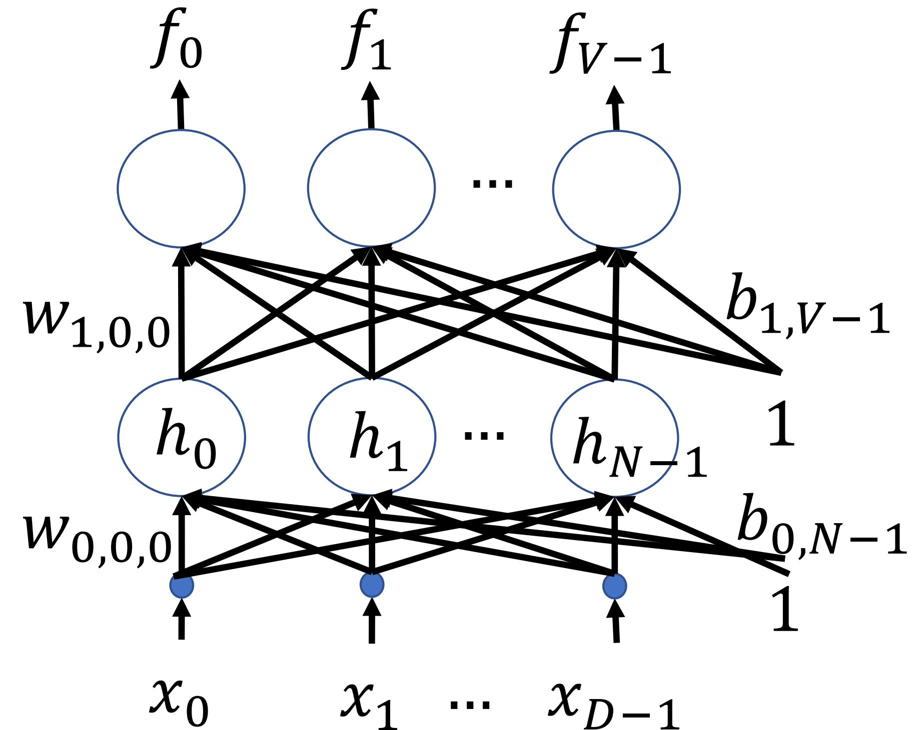
Outline

- From linear to nonlinear classifiers
- Training a two-layer network: Back-propagation
- Other nonlinearities

Training a neural net: Gradient descent

- Suppose we have some scalar loss function, $\mathcal{L}$, that we want to minimize
- Define the gradient of $\mathcal{L}$ w.r.t. any vector of weights, $w_{l,n}$, as:

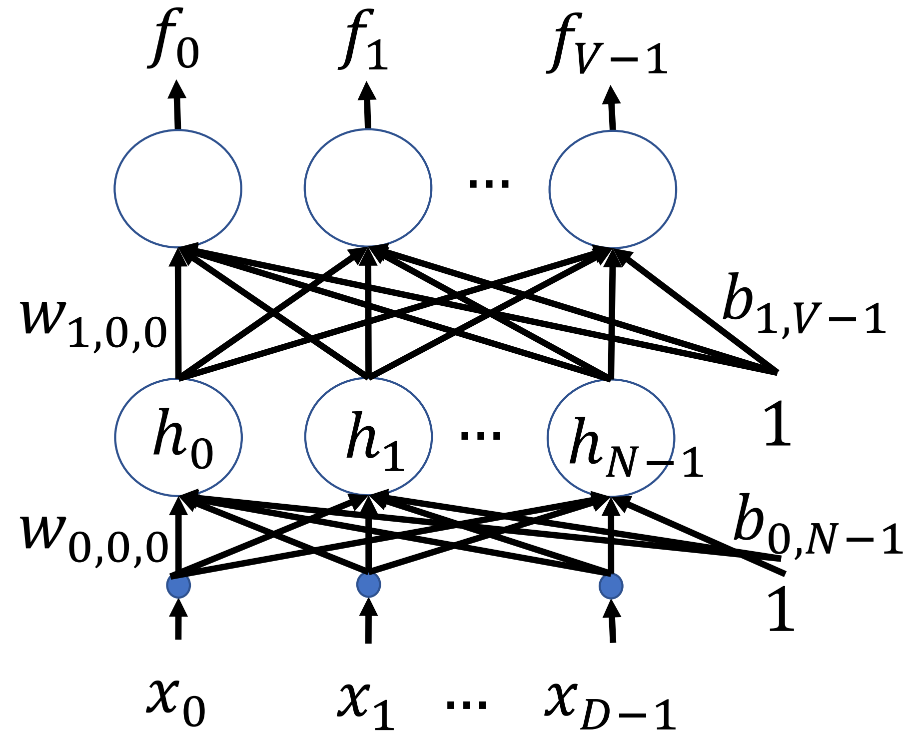
$$\nabla_{w_{l,n}}\mathcal{L} = \left[\frac{\partial \mathcal{L}}{\partial w_{l,n,0}}, \frac{\partial \mathcal{L}}{\partial w_{l,n,1}}, \dots\right]$$



Training a neural net: Gradient descent

Gradient descent updates w_l as:

$$w_{l,n} \leftarrow w_{l,n} - \eta \nabla_{w_{l,n}} \mathcal{L}$$

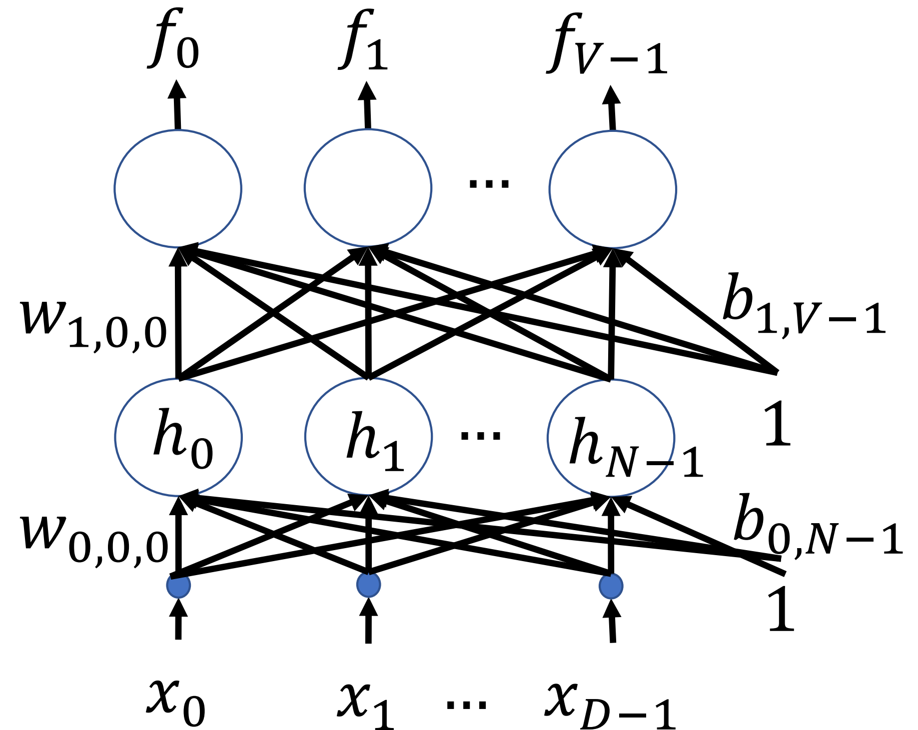


Back-propagation = Chain rule of calculus

- Now here's the big question: how do we find $\nabla_{w_l}\mathcal{L}$?
- Answer: use the chain rule of calculus.

$$\frac{\partial \mathcal{L}}{\partial h_n} = \sum_{c=0}^{V-1} \frac{\partial \mathcal{L}}{\partial f_c} \times \frac{\partial f_c}{\partial h_n}$$

$$\frac{\partial \mathcal{L}}{\partial w_{0,n,d}} = \frac{\partial \mathcal{L}}{\partial h_n} \times \frac{\partial h_n}{\partial w_{0,n,d}}$$



Quiz

The quiz today asks you to compute $\frac{\partial \mathcal{L}}{\partial h_n}$ if you know $\frac{\partial \mathcal{L}}{\partial f_c}$, and if the output layer is linear, so that

$$f_c = \sum_{n=0}^{N-1} w_{1,c,n} h_n$$

Remember that the relevant chain rule is:

$$\frac{\partial \mathcal{L}}{\partial h_n} = \sum_{c=0}^{V-1} \frac{\partial \mathcal{L}}{\partial f_c} \times \frac{\partial f_c}{\partial h_n}$$

Give it a try:

https://us.prairielearn.com/pl/course_instance/129874/assessment/2331127

quiz

$f0=8*h0+3*h1$

$f1=4*h0+6*h1$

$dL/dh0 = (dL/df0)*(df0/dh0)+(dL/df1)*(df1/dh0)$

$= 8*0+4*(-0.2)$

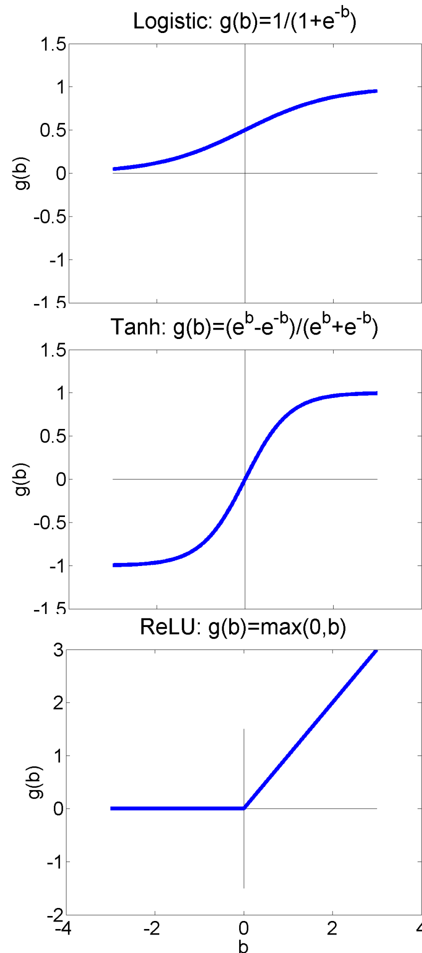
How to train a neural network

- From a very large training dataset, randomly choose a training token (x_i, y_i)
- Calculate the neural net prediction, $f(x_i)$
- Calculate the cross-entropy loss, $\mathcal{L} = -\log f_{y_i}(x_i)$
- Back-propagate to find the gradients, $\frac{\partial \mathcal{L}}{\partial w_{1,c,n}}$ and $\frac{\partial h_n}{\partial w_{0,n,d}}$
- Do a gradient update step, $w_{l,n} \leftarrow w_{l,n} - \eta \nabla_{w_{l,n}} \mathcal{L}$
- Repeat until the loss is small enough.

Outline

- From linear to nonlinear classifiers
- Training a two-layer network: Back-propagation
- Other nonlinearities

Activation functions



The "activation function," $g(\cdot)$, can be any scalar nonlinearity. Common ones that you should know include:

Logistic Sigmoid:

$$\sigma(\beta) = \frac{1}{1 + e^{-\beta}}$$

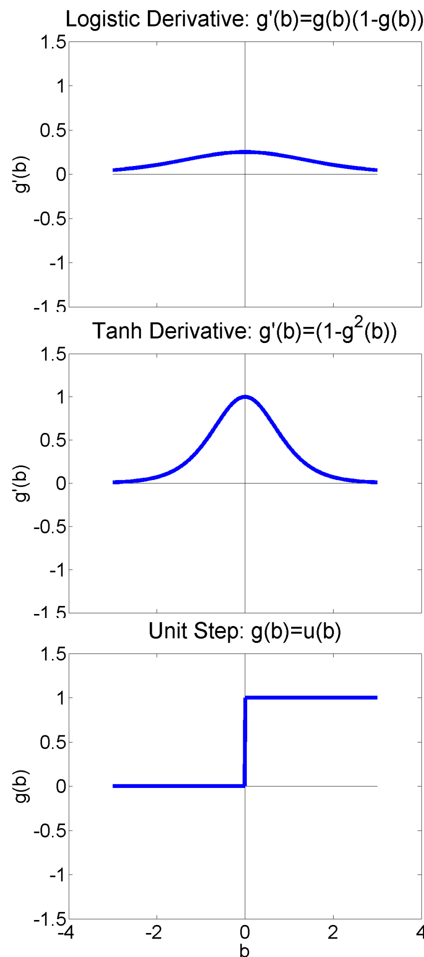
Hyperbolic Tangent (tanh):

$$\tanh(\beta) = \frac{e^{\beta} - e^{-\beta}}{e^{\beta} + e^{-\beta}}$$

Rectified Linear Unit (ReLU):

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

Derivatives of common activation functions



The derivatives of common activation functions are usually things that you can write in terms of the function itself, like this:

Derivative of Sigmoid:

$$\frac{d\sigma(\beta)}{d\beta} = -\frac{e^{-\beta}}{(1+e^{-\beta})^2} = \sigma(\beta)\big(1-\sigma(\beta)\big)$$

Derivative of Tanh:

$$\frac{d\tanh(\beta)}{d\beta} = 1 - \tanh^2(\beta)$$

Derivative of ReLU:

$$\frac{d\mathrm{ReLU}(\beta)}{d\beta} = u(\beta)$$

Why are they used?

- Logistic sigmoid is used if, for some reason, your hidden nodes need to be bounded and non-negative
  - Example: “gate” nodes in a gated recurrent unit. 0 = off, 1 = on.
- Tanh is used if your hidden nodes need to be bounded, but not necessarily non-negative
- ReLU is used for most hidden nodes

Summary

- From linear to nonlinear classifiers
- Training a two-layer network: Back-propagation
- Other nonlinearities