ECE448:
Deep Learning

Mark Hasegawa-Johnson, 5/2023

Outline

- Linear Classifiers
- Gradient descent
- Cross-entropy
- Softmax
- From linear to nonlinear classifiers
- Training a deep network: Back-propagation

Linear classifier: Notation

- The observation $x^T = [1, x_1, \ldots, x_d]$ is a real-valued vector (d is the number of feature dimensions)
- The class label $y \in \mathcal{Y}$ is drawn from some finite set of class labels.
- Usually the output vocabulary, $\mathcal{Y}$, is some set of strings. For convenience, though, we usually map the class labels to a sequence of integers, $\mathcal{Y} = \{1, \ldots, v\}$, where v is the vocabulary size

Linear classifier: Definition

A linear classifier is defined by

$$f(x) = \operatorname{argmax} wx$$

where:

$$wx = \begin{bmatrix} b_1 & w_{1,1} & \cdots & w_{1,d} \\ \vdots & \vdots & \ddots & \vdots \\ b_v & w_{v,1} & \cdots & w_{v,d} \end{bmatrix} \begin{bmatrix} 1 \\ x_1 \\ \vdots \\ x_d \end{bmatrix} = \begin{bmatrix} w_1^T x \\ \vdots \\ w_v^T x \end{bmatrix}$$

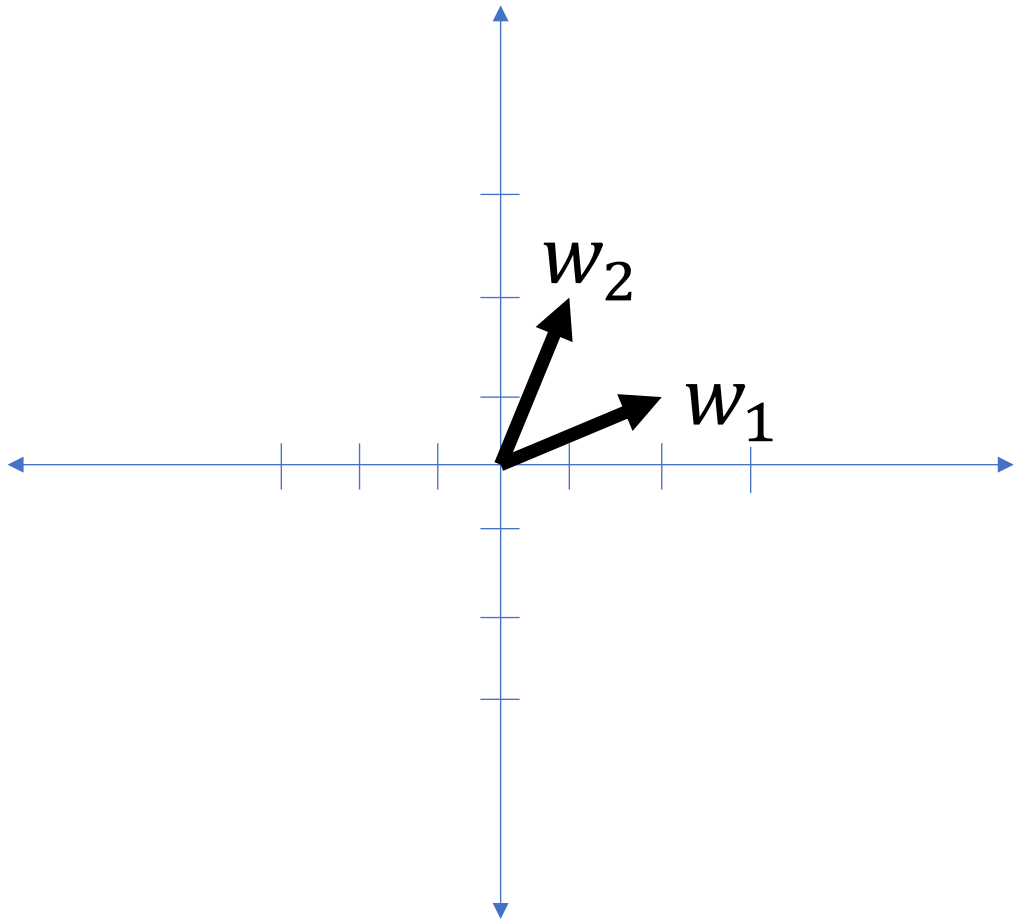
w_k, b_k are the **weight vector** and **bias** corresponding to **class k**, and the argmax function finds the element of the vector wx with the largest value.

There are a total of $v(d+1)$ trainable parameters: the elements of the matrix w.

Example

Consider a two-class classification problem, with

$$w_1^T = [b_1, w_{1,1}, w_{1,2}] = [0,2,1]$$
$$w_2^T = [b_2, w_{2,1}, w_{2,2}] = [0,1,2]$$



Example

Notice that in the two-class case, the equation

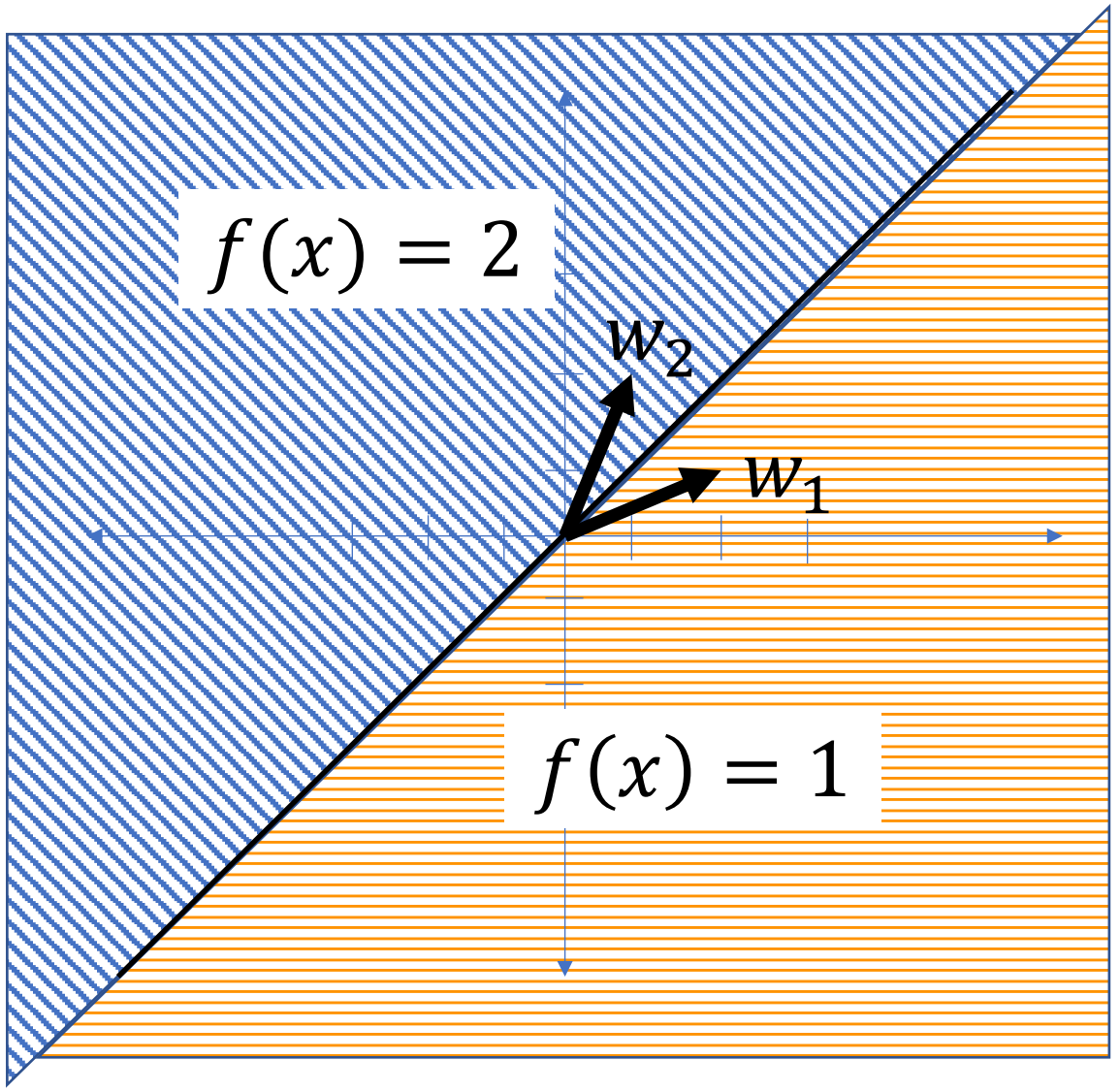
$$f(x) = \text{argmax}\, wx$$

Simplifies to

$$f(x) = \begin{cases} 1 & w_1^T x > w_2^T x \\ 2 & w_1^T x < w_2^T x \end{cases}$$

The class boundary is the line whose equation is

$$(w_2 - w_1)^T x = 0$$



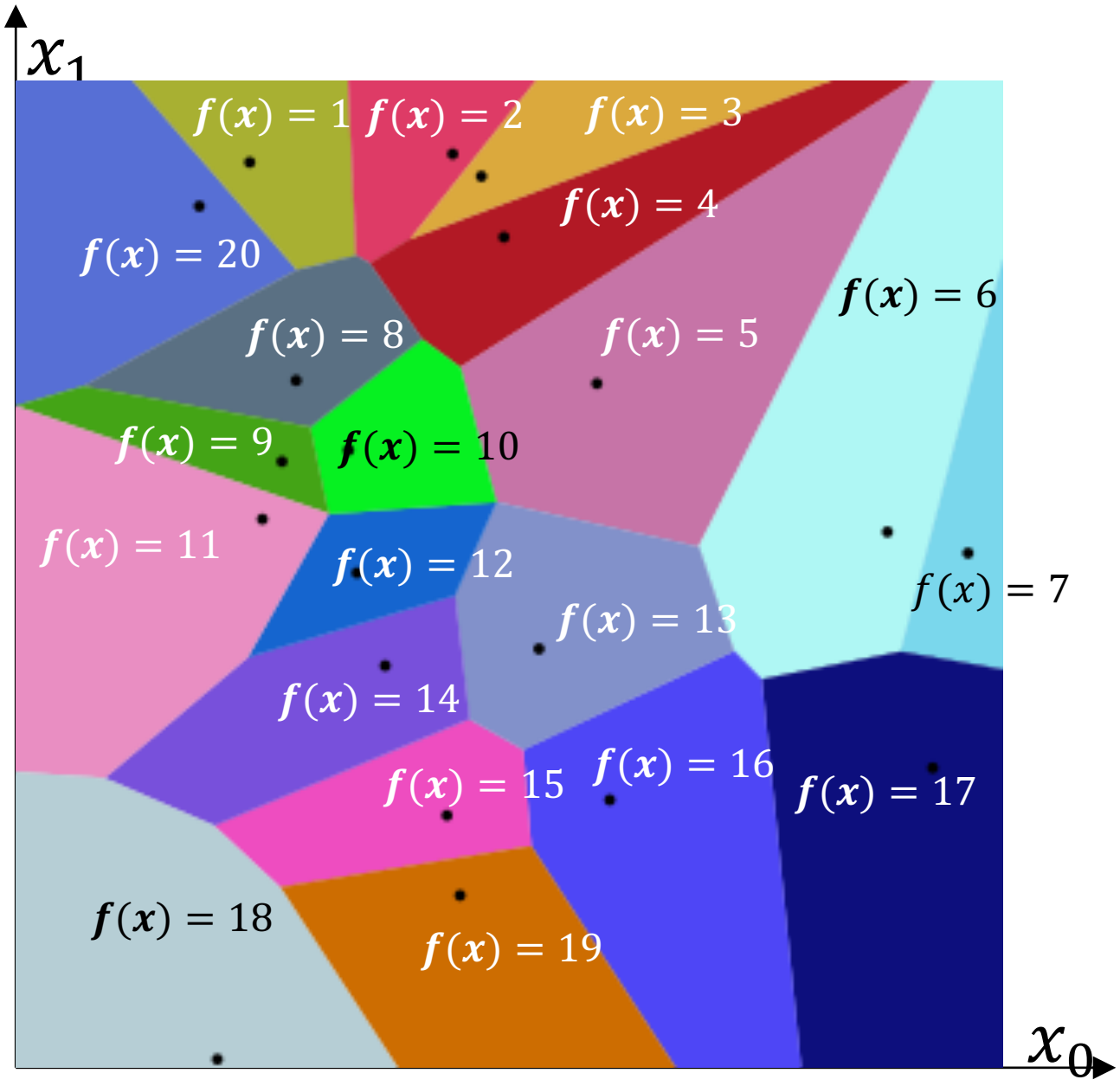
Multi-class linear classifier

In a general multi-class linear classifier,

$$f(x) = \text{argmax}\, wx$$

The boundary between class k and class l is the line (or plane, or hyperplane) given by the equation

$$(w_k - w_l)^T x = 0$$

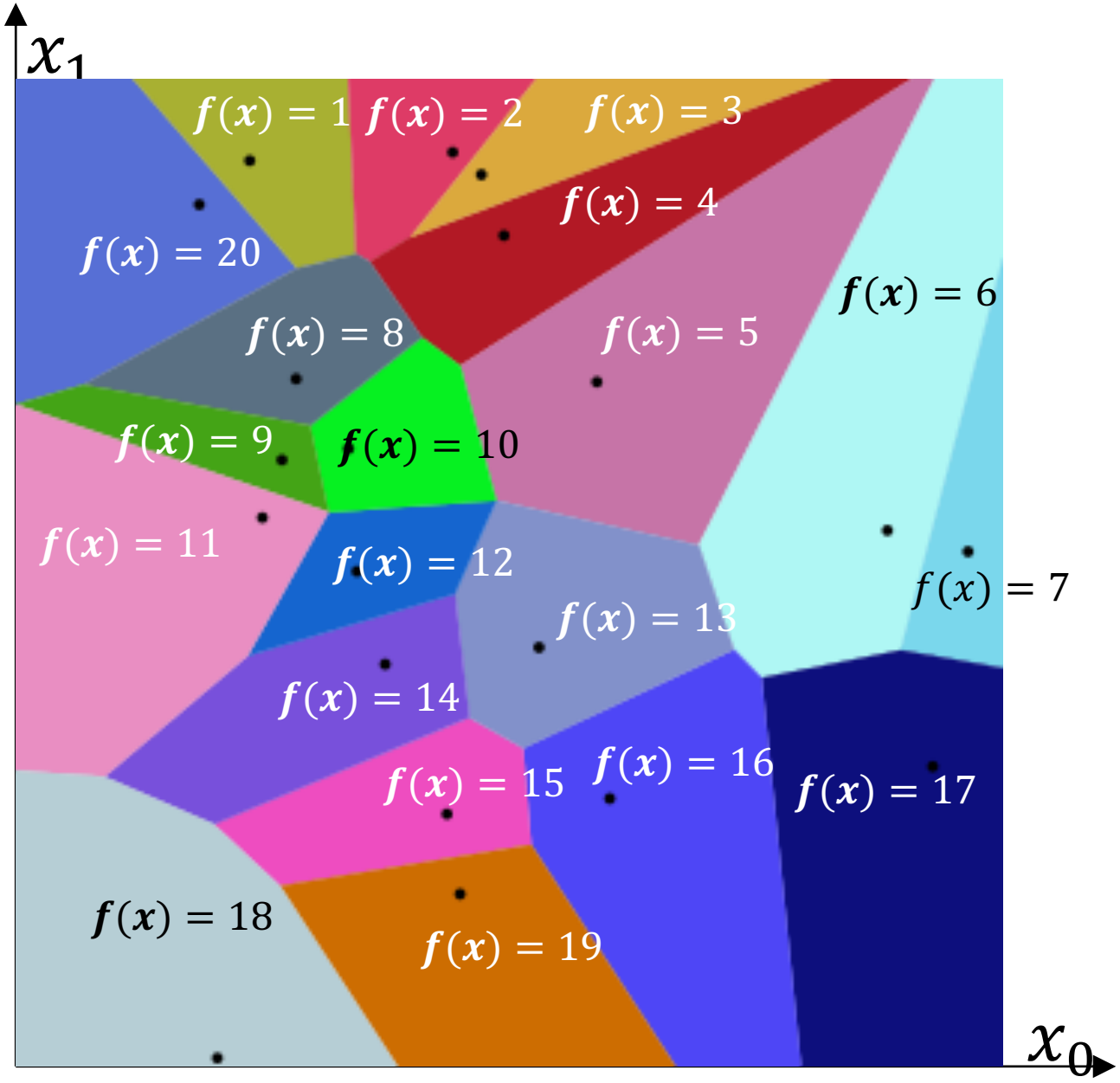


Voronoi regions

The classification regions in a linear classifier are called Voronoi regions.

A **Voronoi region** is a region that is

- Convex (if u and v are points in the region, then every point on the line segment $\overline{uv}$ connecting them is also in the region)
- Bounded by piece-wise linear boundaries



Outline

- Linear Classifiers
- Gradient descent
- Cross-entropy
- Softmax
- From linear to nonlinear classifiers
- Training a deep network: Back-propagation

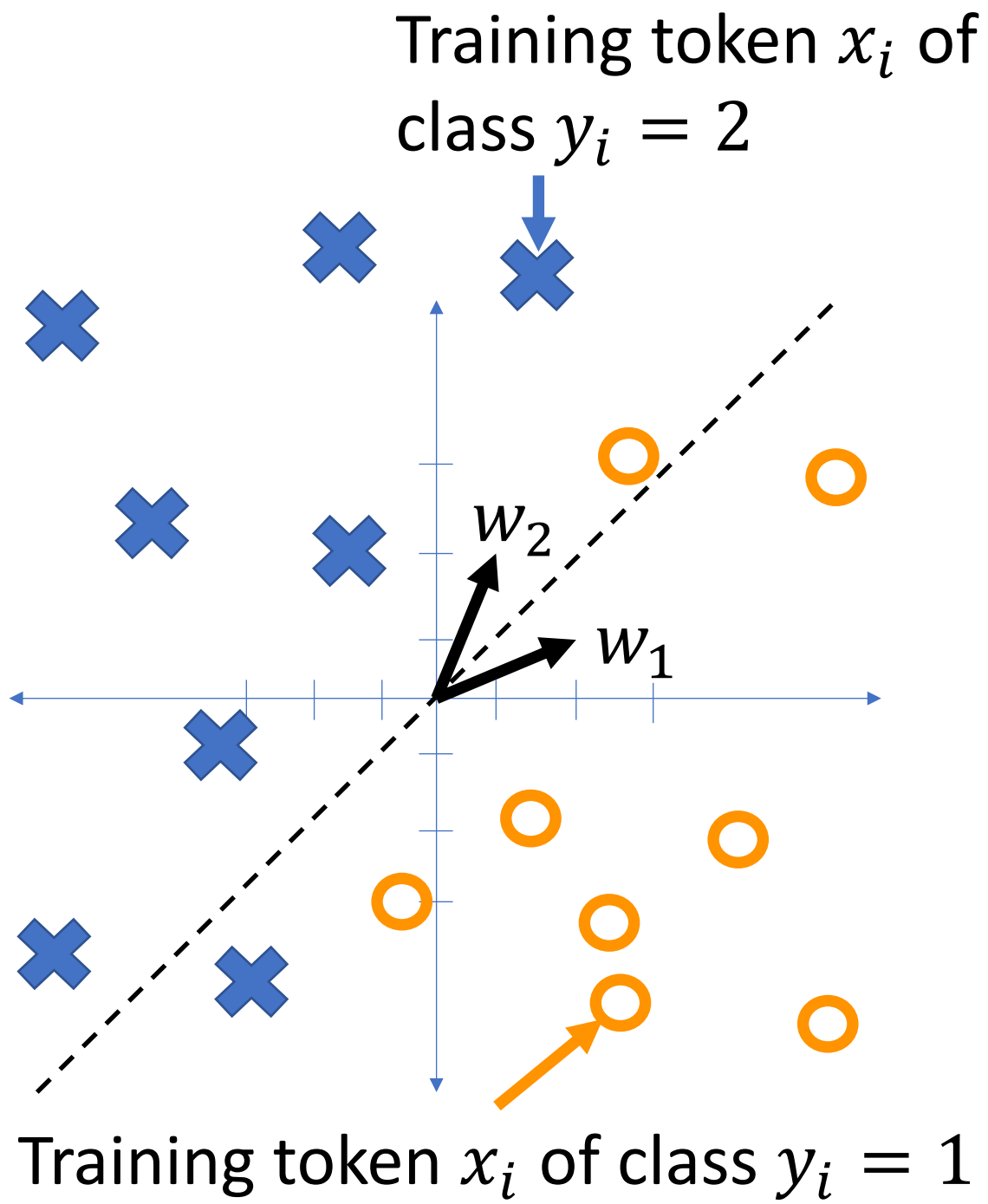
Gradient descent

Suppose we have training tokens (x_i, y_i), and we have some initial class vectors w_1 and w_2. We want to update them as

$$w_1 \leftarrow w_1 - \eta \frac{\partial \mathcal{L}}{\partial w_1}$$

$$w_2 \leftarrow w_2 - \eta \frac{\partial \mathcal{L}}{\partial w_2}$$

…where $\mathcal{L}$ is some loss function. What loss function makes sense?



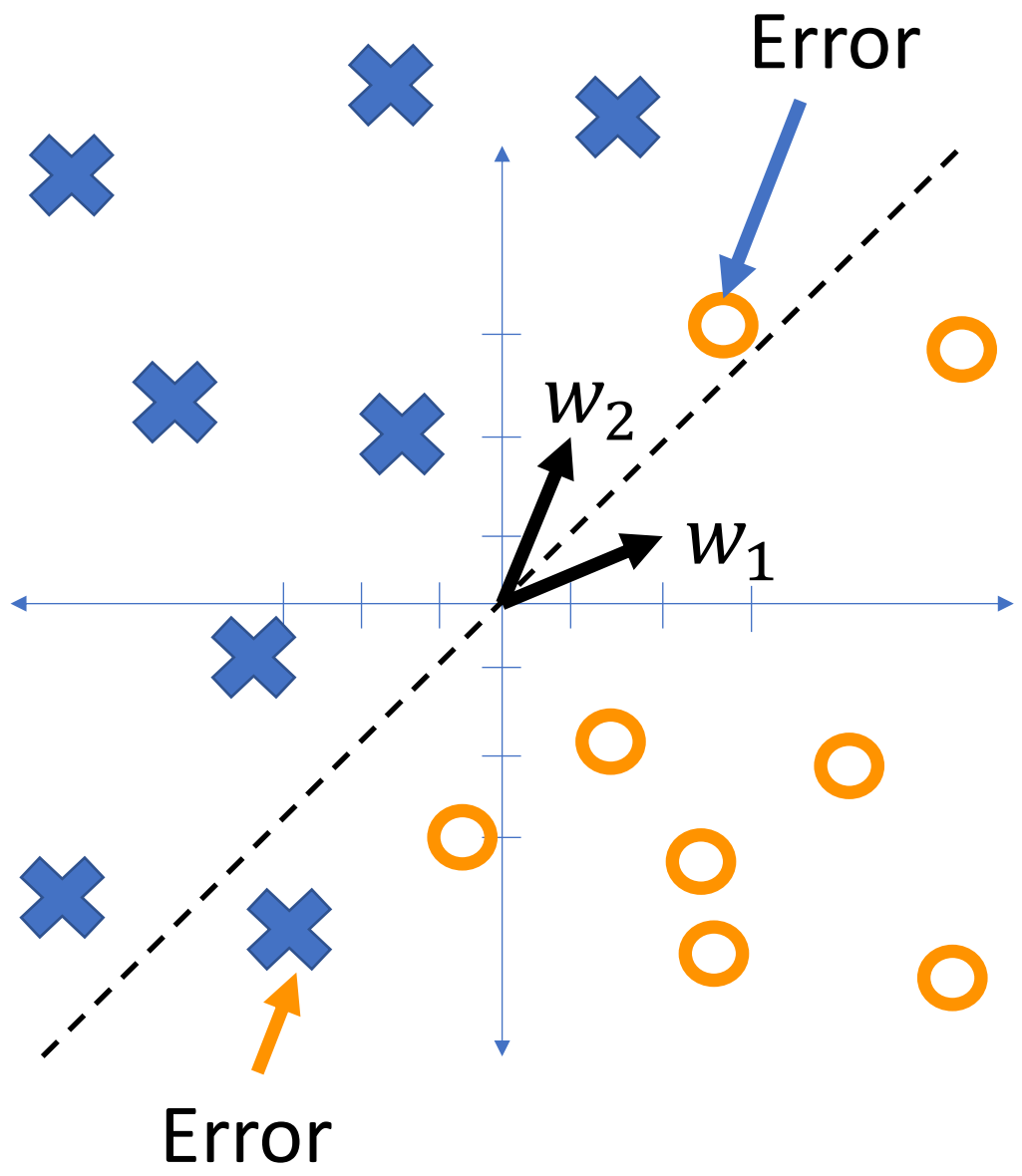
Zero-one loss function

The most obvious loss function for a classifier is its classification error rate,

$$\mathcal{L} = \frac{1}{n}\sum_{i=1}^{n} \ell(f(x_i), y_i)$$

Where $\ell(\hat{y}, y)$ is the zero-one loss function,

$$\ell(\hat{y}, y) = \begin{cases} 0 & \hat{y} = y \\ 1 & \hat{y} \neq y \end{cases}$$

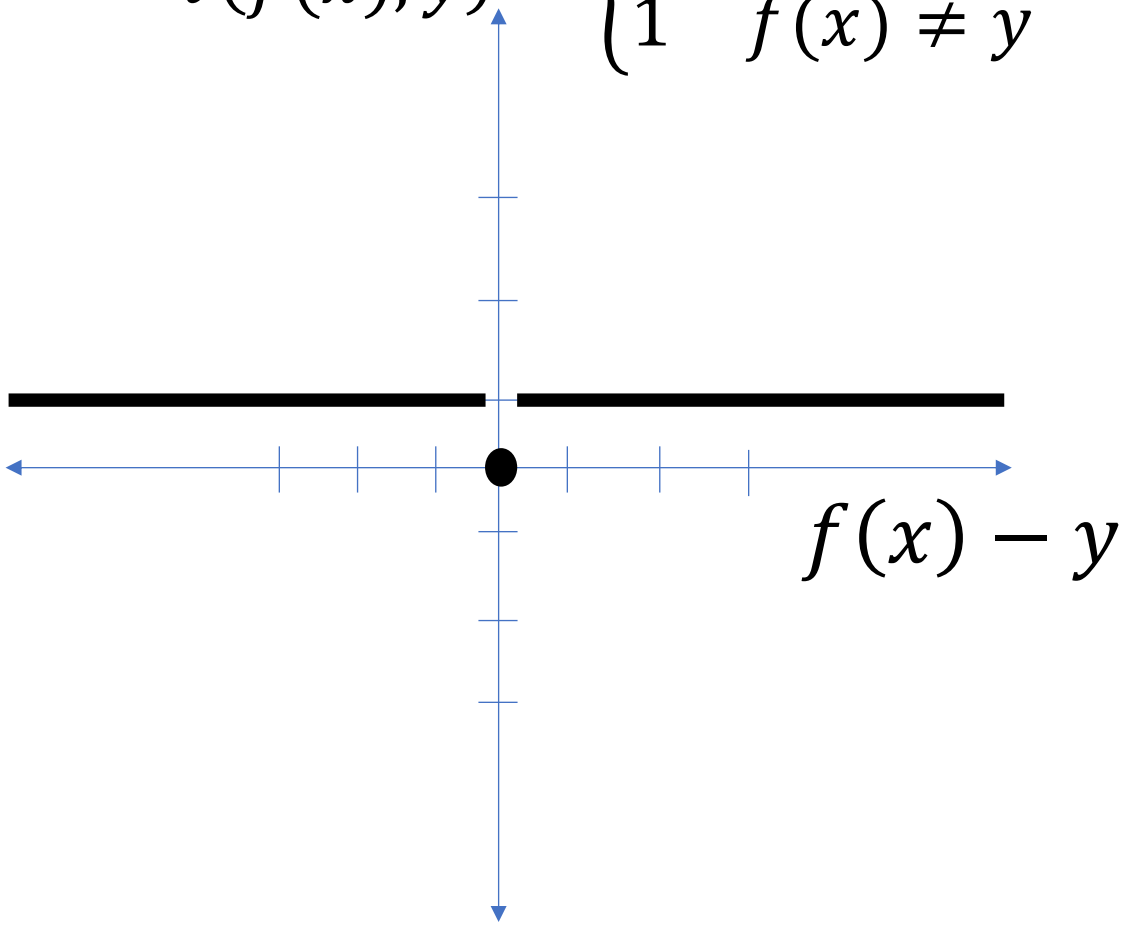


Non-differentiable!

The problem with the zero-one loss function is that it's not differentiable:

$$\frac{\partial \ell(f(x), y)}{\partial f(x)} = \begin{cases} 0 & f(x) \neq y \\ +\infty & f(x) = y^{+} \\ -\infty & f(x) = y^{-} \end{cases}$$

$$\ell(f(x), y) = \begin{cases} 0 & f(x) = y \\ 1 & f(x) \neq y \end{cases}$$



Outline

- Linear Classifiers: multi-class and 2-class
- Gradient descent
- Cross-entropy
- Softmax
- From linear to nonlinear classifiers
- Training a deep network: Back-propagation

One-hot vectors

A **one-hot vector** is a binary vector in which all elements are 0 except for a single element that's equal to 1.

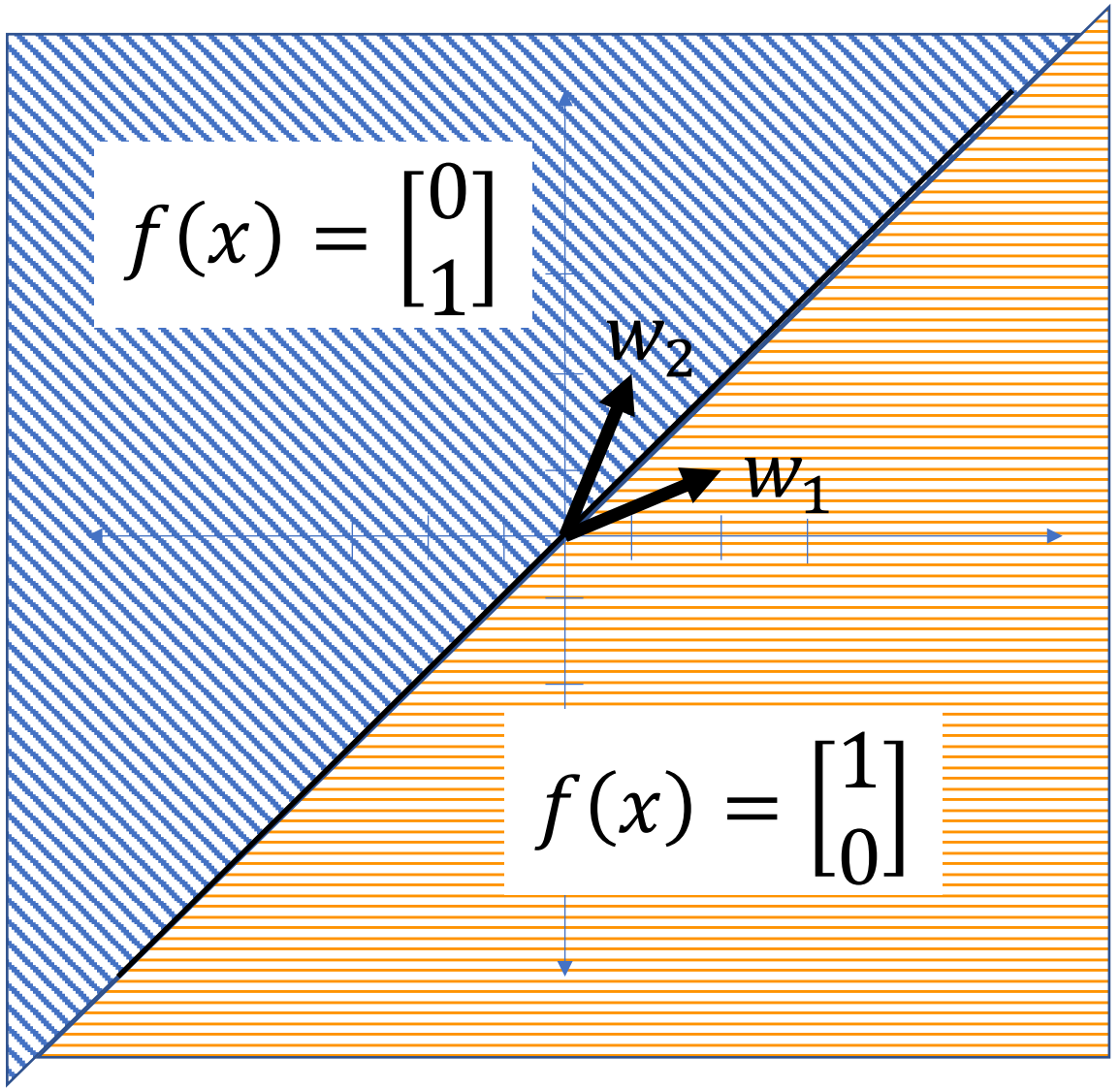
Example: Binary classifier

Consider the classifier

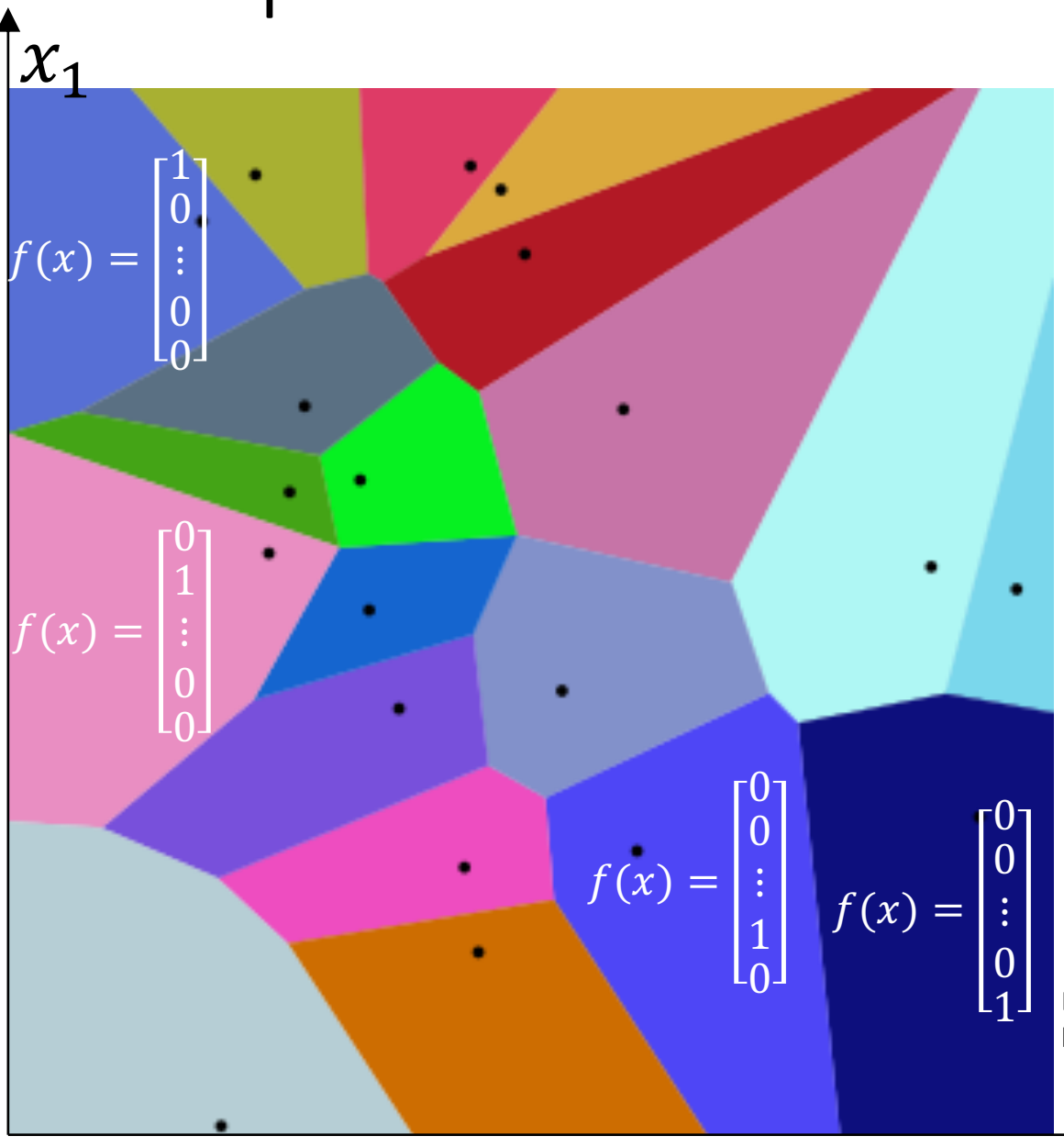
$$f(x) = \begin{bmatrix} f_1(x) \\ f_2(x) \end{bmatrix} = \begin{bmatrix} \mathbb{1}_{\text{argmax } wx=1} \\ \mathbb{1}_{\text{argmax } wx=2} \end{bmatrix}$$

…where $\mathbb{1}_P$ is called the "indicator function," and it means:

$$\mathbb{1}_P = \begin{cases} 1 & P \text{ is true} \\ 0 & P \text{ is false} \end{cases}$$



Example: Multi-Class



Consider the classifier

$$f(x) = \begin{bmatrix} f_1(x) \\ \vdots \\ f_v(x) \end{bmatrix} = \begin{bmatrix} \mathbb{I}_{\text{argmax } wx=1} \\ \vdots \\ \mathbb{I}_{\text{argmax } wx=v} \end{bmatrix}$$

… with 20 classes. Then some of the classifications might look like this.

Using one-hot vectors to calculate the loss

- Suppose that the output is a one-hot vector. Then the goal of the classifier is to set $f_c(x_i) = 1$ for the correct class, and $f_c(x_i) \approx 0$ for all others.
- We can measure this by a formula like:

$$\mathcal{L} = -\frac{1}{n}\sum_{i=1}^{n} \log f_{y_i}(x_i)$$

In words:

- choose the y_ith output of the classifier.
- If that output is $f_{y_i}(x_i) = 1$, then the loss is zero.
- If that output is $f_{y_i}(x_i) < 1$, then the loss is large (∞ if $f_{y_i}(x_i) = 0$).

Cross-entropy

This loss function,

$$\mathcal{L} = -\frac{1}{n}\sum_{i=1}^{n} \log f_{y_i}(x_i)\,,$$

is called cross-entropy. By measuring the negative log-probability of the correct class, we are measuring the **extra uncertainty** that is added to the system by **classification errors.**

CC-SA 4.0, https://en.wikipedia.org/wiki/File:Ultra_slow-motion_video_of_glass_tea_cup_smashed_on_concrete_floor.webm

Cross-entropy of a one-hot vector is still not differentiable!

Consider the classifier

$$f(x_i) = [f_1(x_i), \ldots, f_v(x_i)]^T$$

$$f_c(x_i) = \begin{cases} 1 & c = \text{argmax}\, wx \\ 0 & \text{otherwise} \end{cases}$$

Unfortunately, the cross-entropy of a one-hot vector is still not differentiable!

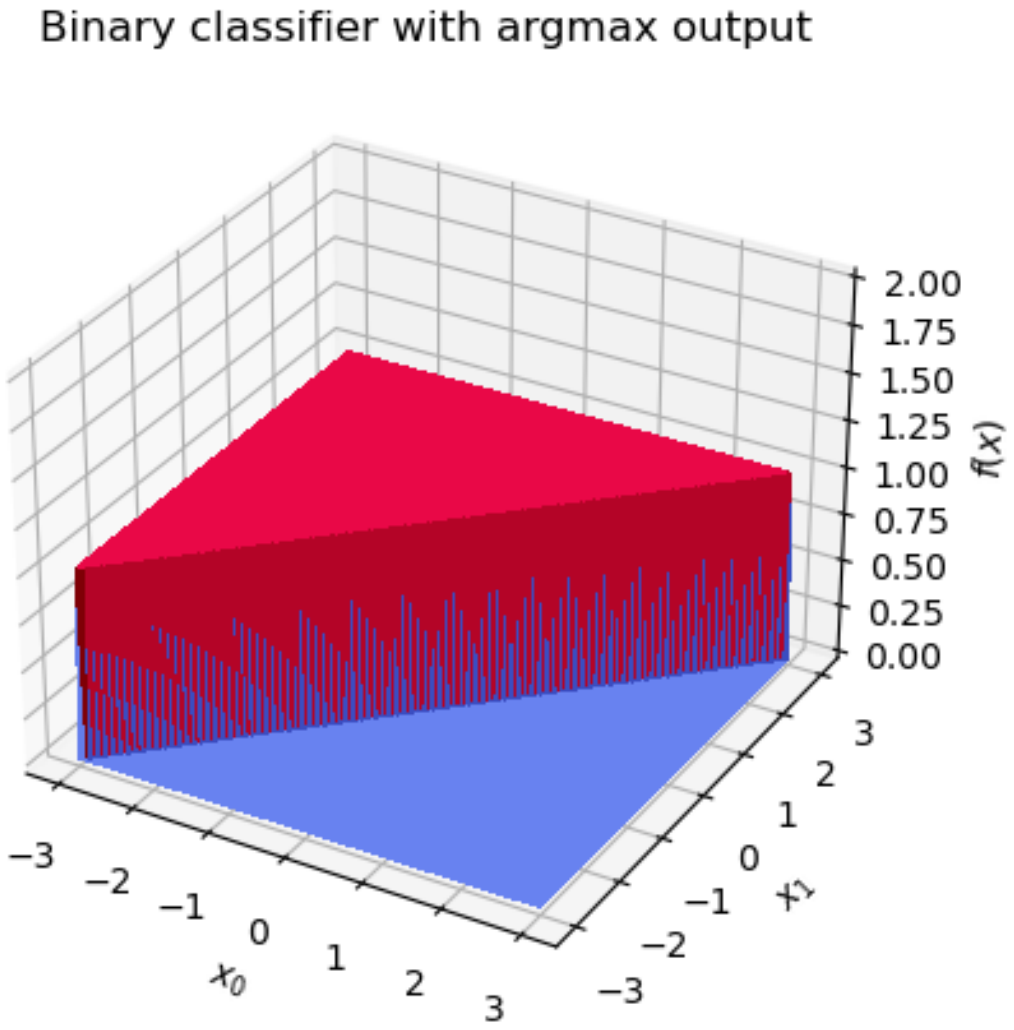
$$\mathcal{L} = -\log f_{y_i}(x_i) = \begin{cases} 0 & f_{y_i}(x_i) = 1 \\ \infty & f_{y_i}(x_i) = 0 \end{cases}$$

Outline

- Linear Classifiers: multi-class and 2-class
- Gradient descent
- One-hot vectors
- Softmax
- From linear to nonlinear classifiers
- Training a deep network: Back-propagation

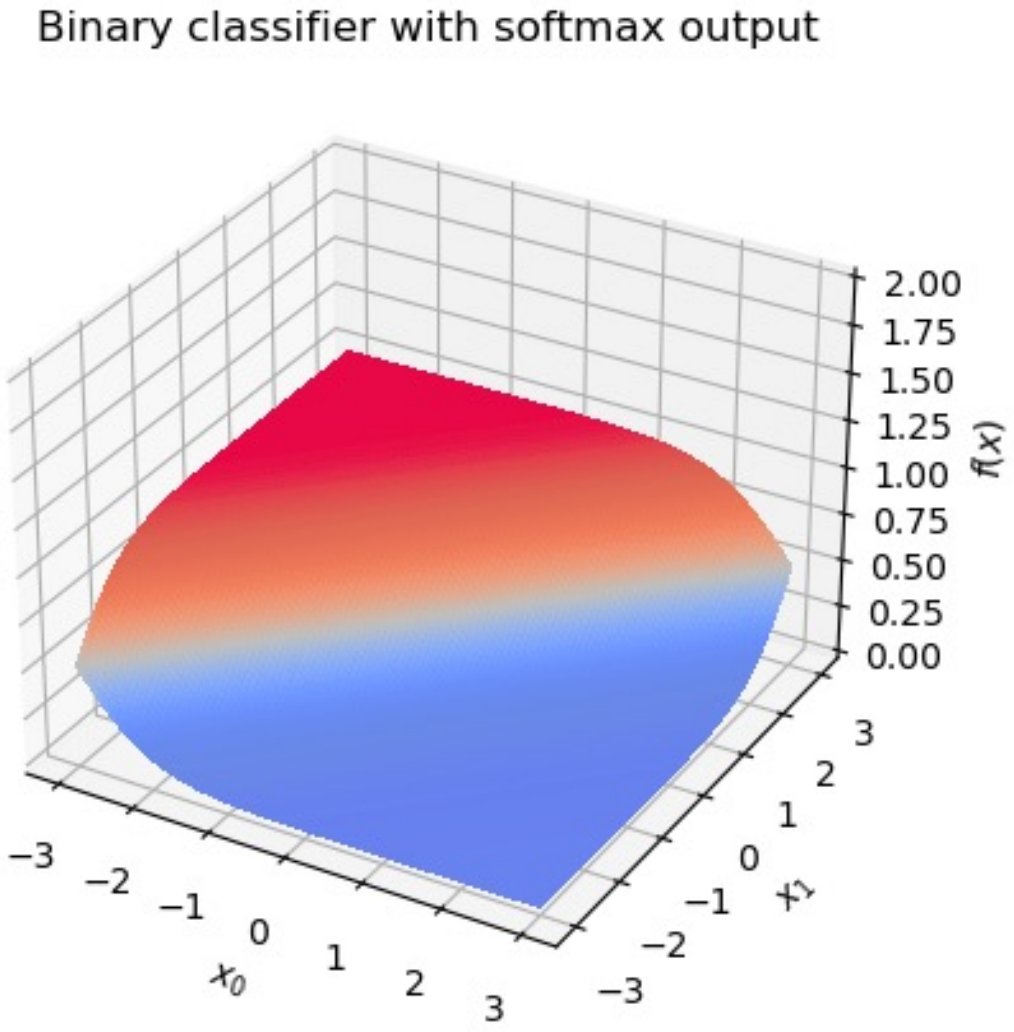
The problem with cross-entropy: $-\log 0 = \infty$

- Cross-entropy is a great loss function because $-\log 1 = 0$, so it measures no loss if the classifier has the right answer
- The problem is that $-\log 0 = \infty$, so if the classifier has the wrong answer, the loss function is unmeasurably huge



The solution: avoid 0-valued outputs

- The solution is to modify $f(x)$ so that it never outputs exactly 0
- Instead, we want $f(x)$ to approach 0 as the classifier gets more confident, but it should never actually reach zero



Argmax versus Softmax

The argmax version of the classifier is

$$f(x) = [f_1(x), \ldots, f_v(x)]^T, \qquad f_c(x) = \begin{cases} 1 & c = \operatorname{argmax} wx \\ 0 & \text{otherwise} \end{cases}$$

We can smooth it by using the softmax function, defined as

$$f(x) = [f_1(x), \ldots, f_v(x)]^T, \qquad f_c(x) = \frac{\exp(w_c^T x)}{\sum_{k=0}^{V-1} \exp(w_k^T x)}$$

The softmax function

This is called the softmax function. We can write it with the Greek letter sigma:

$$\sigma(wx) = [\sigma_1(wx), \ldots, \sigma_v(wx)]^T$$

$$\sigma_c(wx) = \frac{\exp(w_c^T x)}{\sum_{k=1}^{v} \exp(w_k^T x)}$$

…where w_k^T is the kth row of the matrix w.

Key features of the softmax

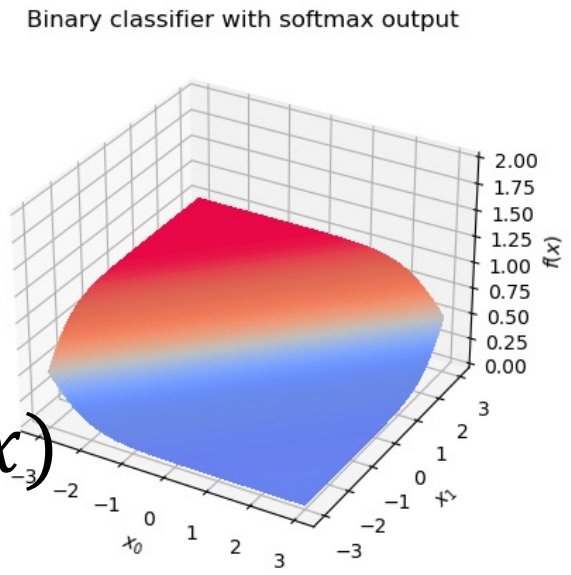
$$\sigma_c(wx) = \frac{\exp(w_c^T x)}{\sum_{k=1}^{v} \exp(w_k^T x)}$$

Notice that the softmax function is (1) smooth, and (2) behaves like a probability distribution:

- $0 < \sigma_c(wx) < 1$
- $\sum_{c=1}^{v} \sigma_c(wx) = 1$

So we can interpret it as a probability!

$$\sigma_y(wx) = P(Y = y|X = x)$$



Logistic sigmoid

Notice that the two-class softmax is special:

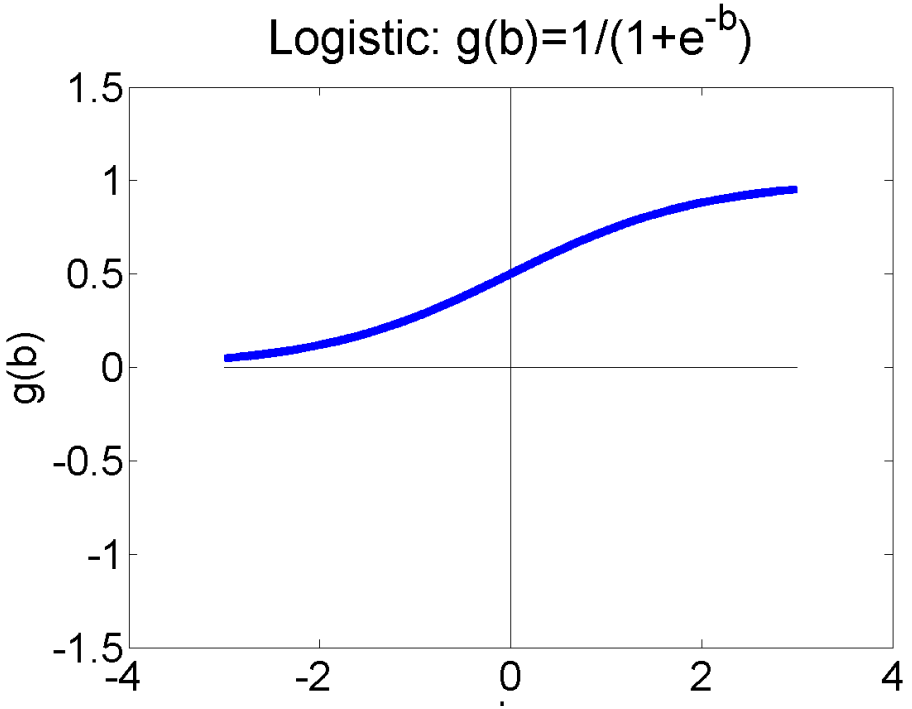
$$\sigma_1(wx) = \frac{e^{w_1^T x}}{e^{w_1^T x} + e^{w_2^T x}} = \frac{1}{1 + e^{(w_2 - w_1)^T x}}$$

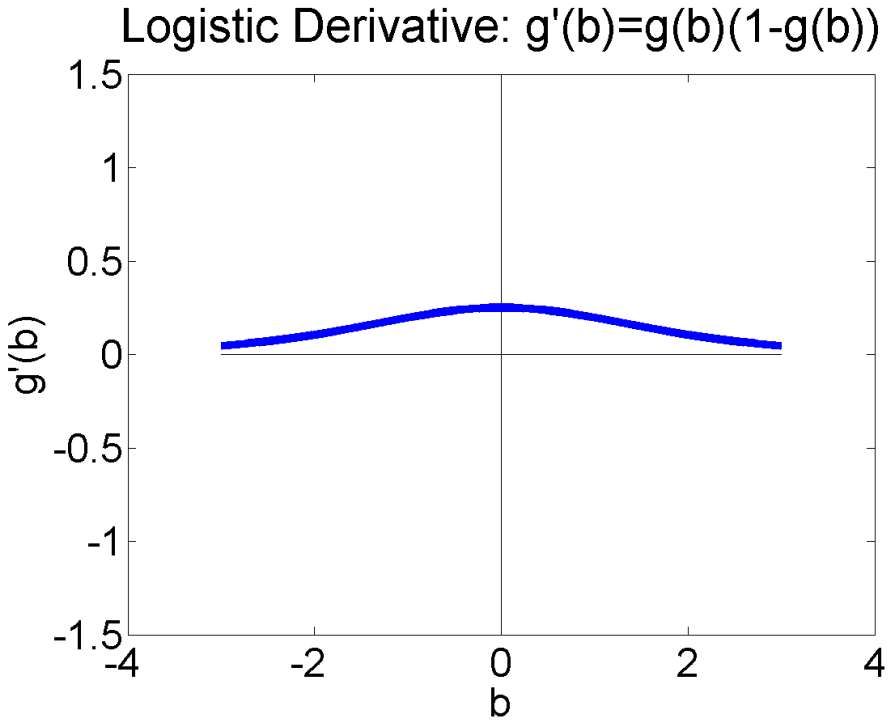
Since it only depends on $w_1 - w_2$, we're free to just arbitrarily set $w_2 = 0$. Then we get

$$\sigma_1(wx) = \frac{1}{1 + e^{-w_1{}^T x}}$$

This is called the logistic sigmoid function. It has many cool properties. Among other things, its derivative is

$$\frac{\partial \sigma_1(z)}{\partial z} = \sigma_1(z)\big(1 - \sigma_1(z)\big)$$





Derivative of the softmax

$$\sigma_y(z) = \frac{e^{z_y}}{\sum_{k=1}^{v} e^{z_k}}$$

$$\frac{\partial \sigma_y(z)}{\partial z_c} = \frac{1}{\sum_{k=1}^{v} e^{z_k}} \frac{\partial e^{z_y}}{\partial z_c} - \frac{e^{z_y}}{(\sum_{k=1}^{v} e^{z_k})^2} \frac{\partial \sum_{k=1}^{v} e^{z_k}}{\partial z_c}$$

$$= \frac{e^{z_c}}{\sum_{k=1}^{v} e^{z_k}} \mathbb{1}_{y=c} - \frac{e^{z_y} e^{z_c}}{(\sum_{k=1}^{v} e^{z_k})^2}$$

$$= \sigma_y(z) \left(\mathbb{1}_{y=c} - \sigma_c(z) \right)$$

Gradient of the cross-entropy of the softmax

But suppose we have a cross-entropy loss function:

$$\mathcal{L} = -\ln \sigma_y(wx)$$

Then:

$$\frac{\partial \mathcal{L}}{\partial w} = -\frac{\partial \ln \sigma_y(wx)}{\partial \sigma_y(wx)} \times \frac{\partial \sigma_y(wx)}{\partial wx} \times \frac{\partial wx}{\partial w} = \left(\sigma_c(wx) - \mathbb{1}_{y=c}\right)x$$

…is the same as the gradient of MSE for linear regression!

For linear regression, we had

$$\frac{\partial \mathcal{L}_i}{\partial w} = 2\epsilon_i x_i$$

For the softmax classifier with cross-entropy loss, we have

$$\frac{\partial \mathcal{L}_i}{\partial w_c} = 2\epsilon_{i,c} x_i$$

…where $\epsilon_{i,c}$ is the error of the cth output of the classifier:

$$\epsilon_{i,c} = \begin{cases} f_c(x_i) - 1 & c = y_i \text{ (output should be 1)} \\ f_c(x_i) - 0 & \text{otherwise(output should be 0)} \end{cases}$$

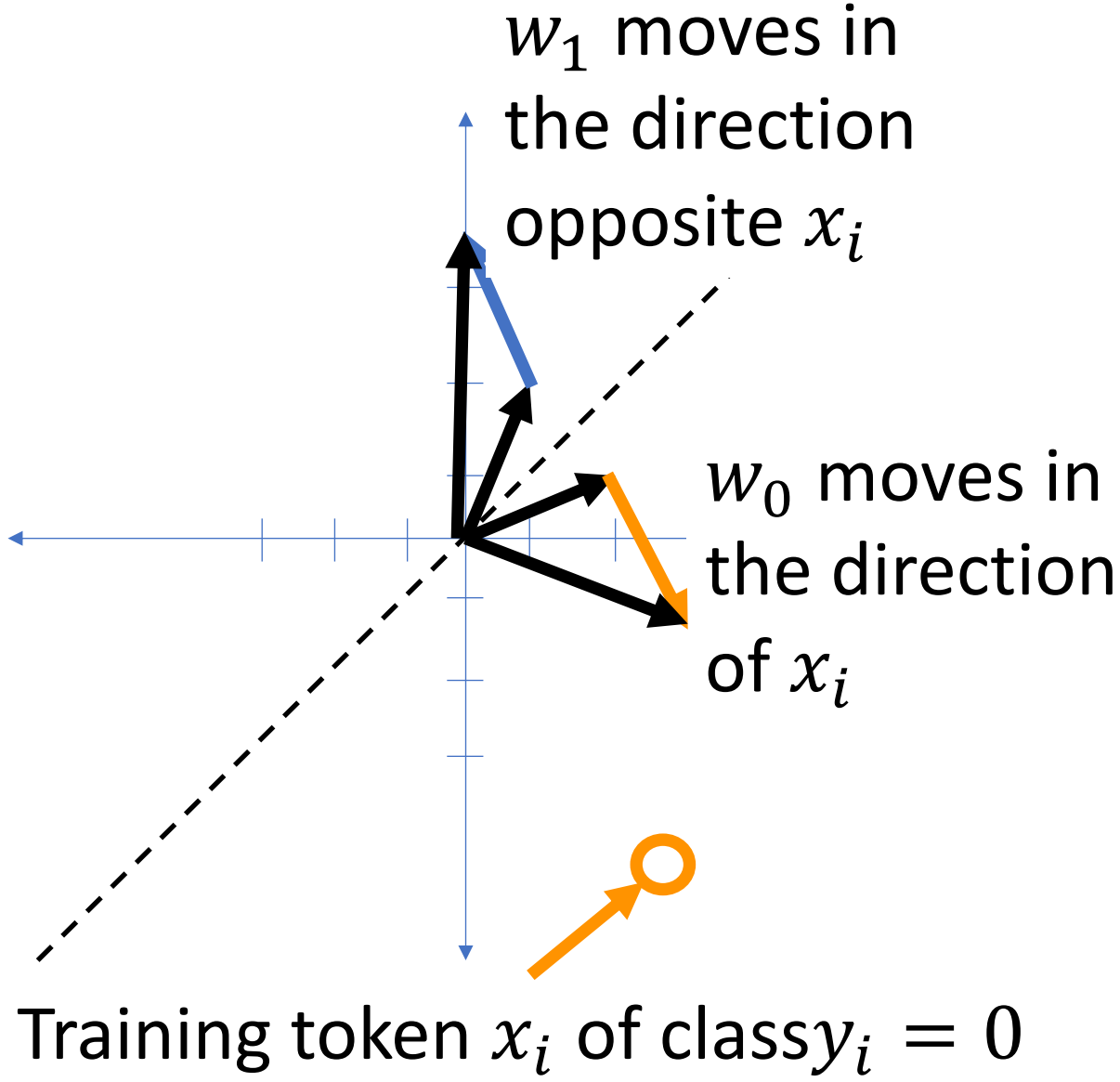
Stochastic gradient descent

Suppose we have a training token (x_i, y_i), and we have some initial class vectors w_c. Using softmax and cross-entropy loss, we can update the weight vectors as

$$w_c \leftarrow w_c - \eta \epsilon_{i,c} x_i$$

…where

$$\epsilon_{i,c} = \begin{cases} f_c(x_i) - 1 & c = y_i \\ f_c(x_i) - 0 & \text{otherwise} \end{cases}$$



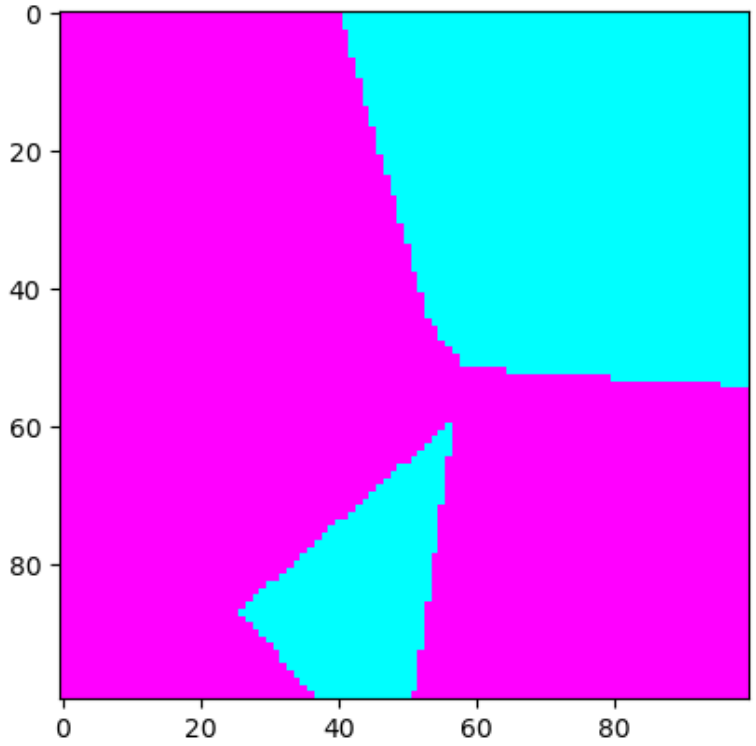
Outline

- Linear Classifiers: multi-class and 2-class
- Gradient descent
- One-hot vectors
- Softmax
- From linear to nonlinear classifiers
- Training a deep network: Back-propagation

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

$$f(x) = \text{argmax}\, wx$$



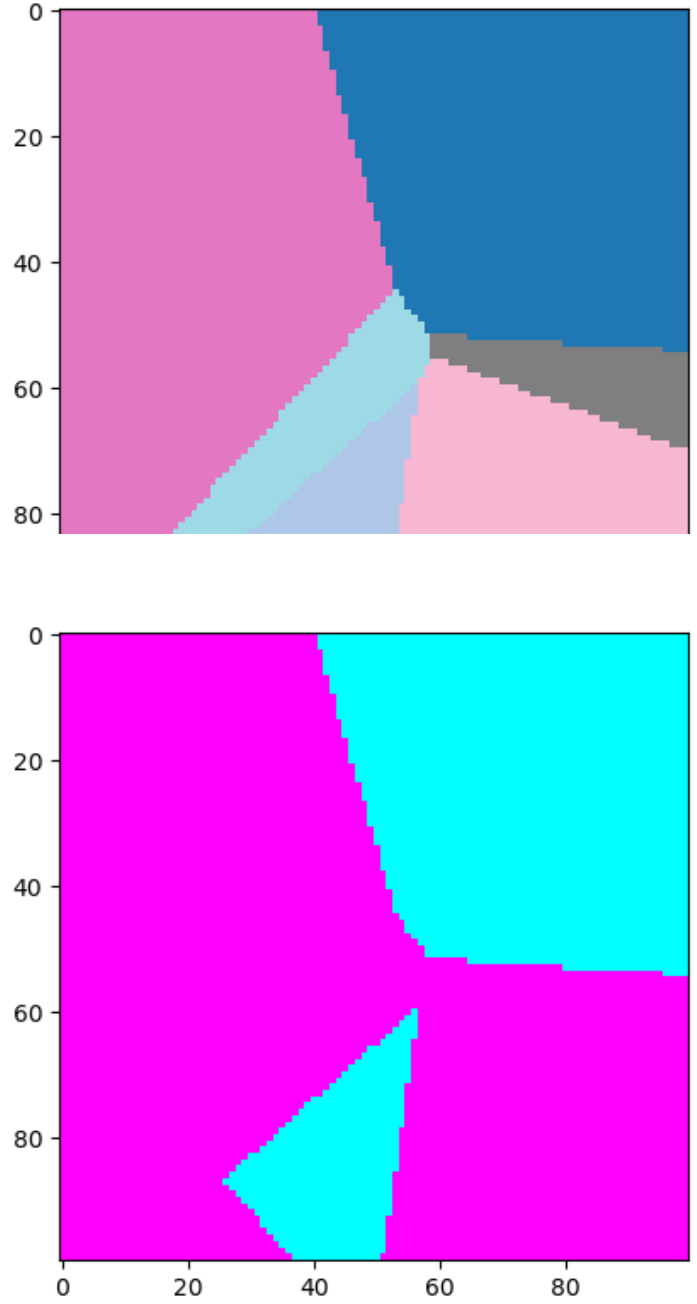
Piece-wise linear classifier

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

$$h(x) = \operatorname{argmax} wx$$

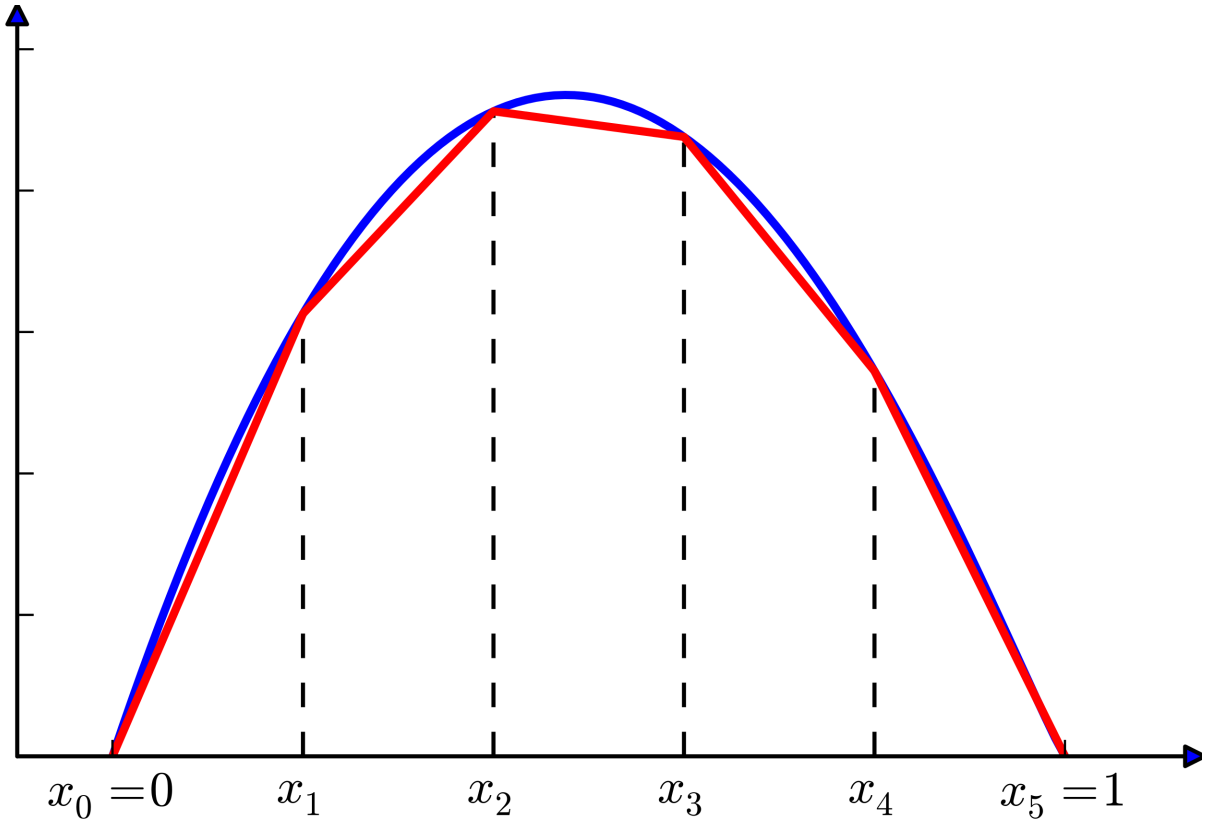
- Then just threshold h(x) to get f(x):

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



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

$$h(x) = \operatorname{argmax} wx$$

- … 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

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

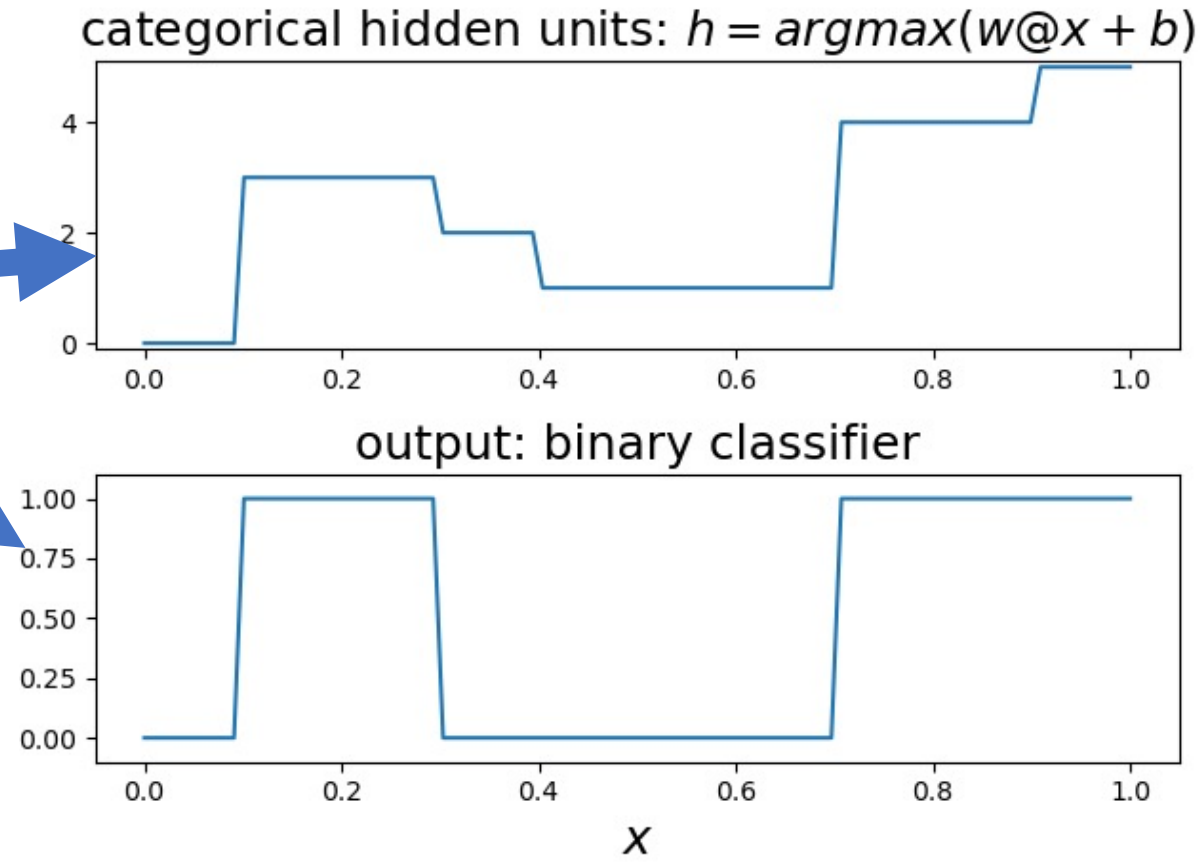
How to add argmax nodes: a 1d example

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

$$h(x) = \text{argmax}\, wx$$

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

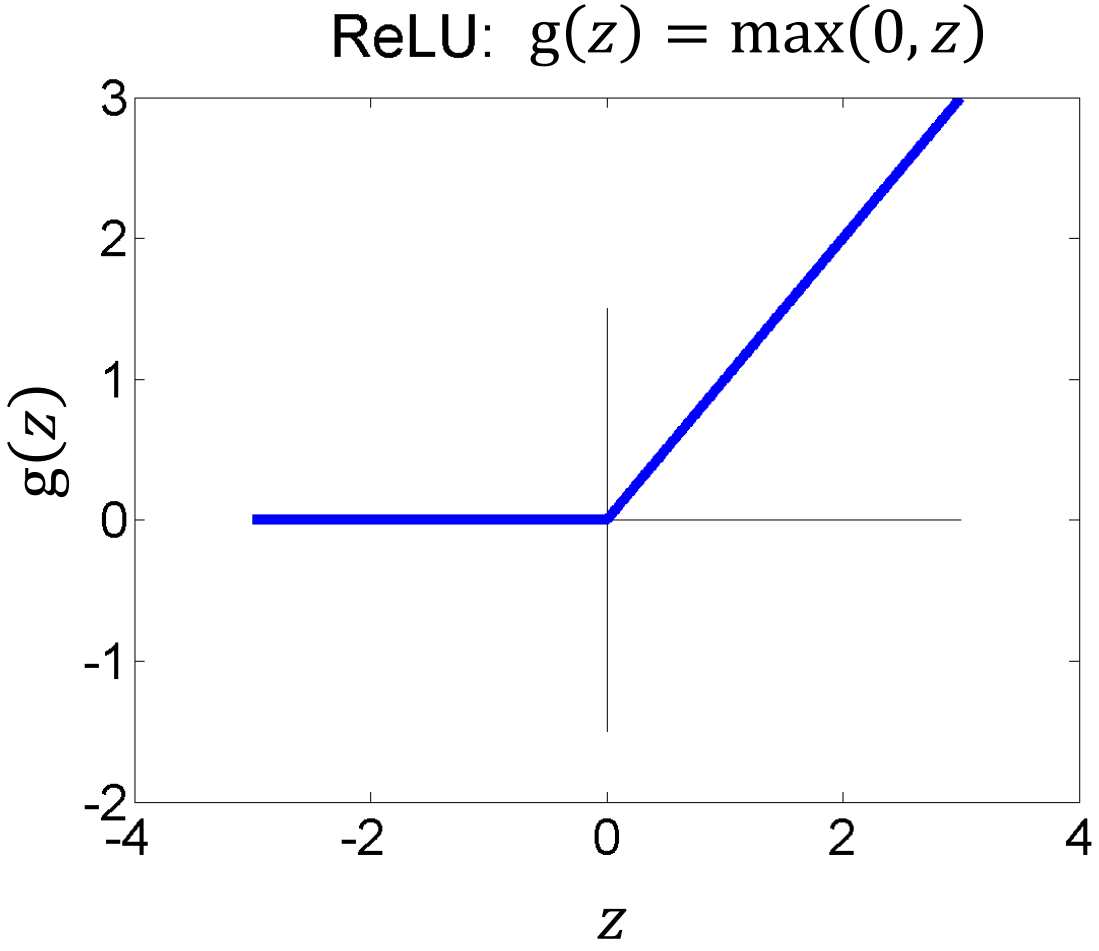
The problem with this method is that neither argmax nor unit step are differentiable, so we can't train this classifier!



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:

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



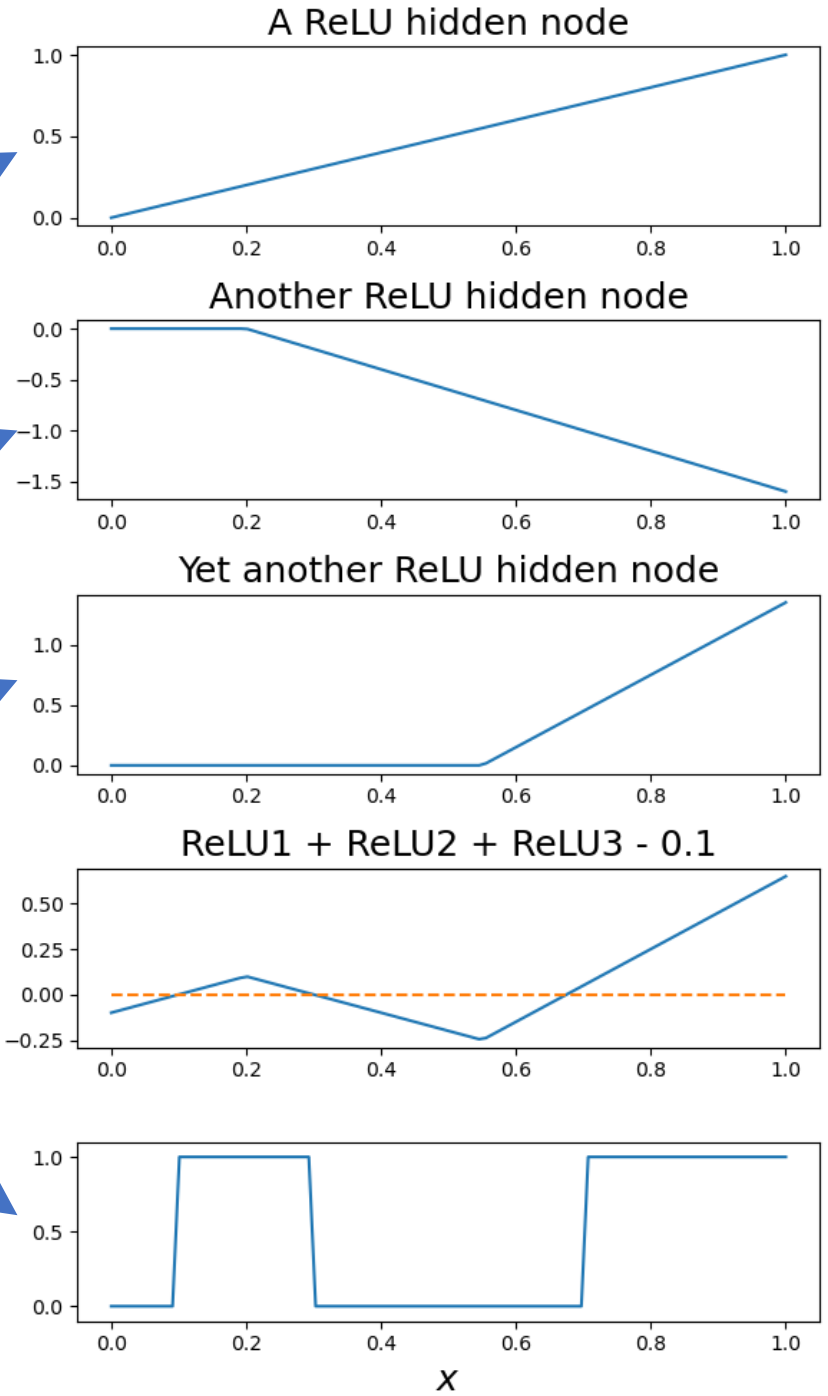
ReLU hidden nodes

$$h_1(x) = ReLU(x)$$

$$w_{2,2}h_2(x) = -2ReLU(x - 0.2)$$

$$w_{2,3}h_3(x) = 3ReLU(x - 0.55)$$

$$f(x) = u(h_1 - 2h_2 + 3h_3 - 0.1)$$



Deep neural net: Basic notation

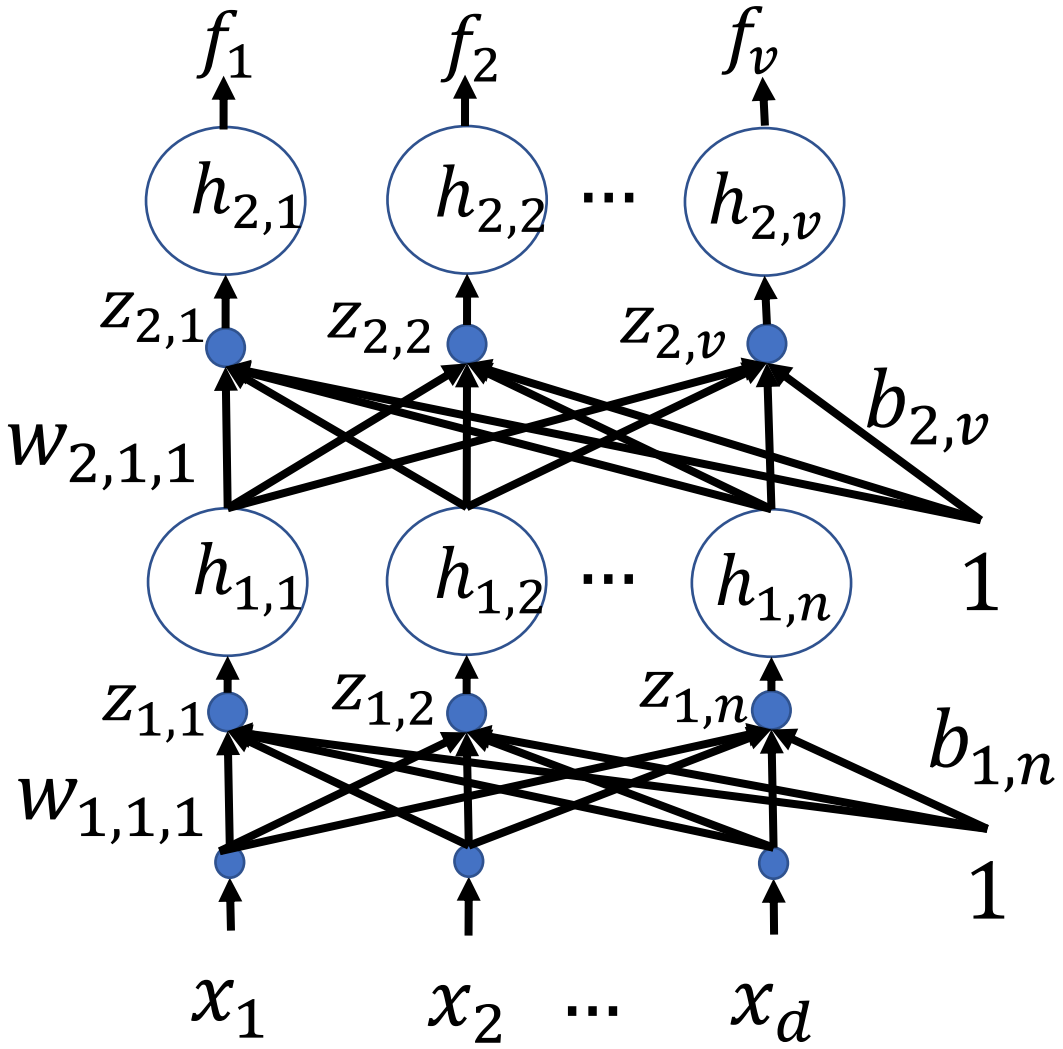
A deep neural net consists of just two steps, repeated over and over:

- Matrix multiplication:

$$z_l = w_l h_{l-1} = \begin{bmatrix} b_{l,1} & w_{l,1,1} & \cdots & w_{l,1,n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{l,n} & w_{l,n,1} & \cdots & w_{l,n,n} \end{bmatrix} \begin{bmatrix} 1 \\ h_{l-1,1} \\ \vdots \\ h_{l-1,n} \end{bmatrix}$$

- Nonlinearity:

$$h_l = g_l(z_l) = \begin{bmatrix} 1 \\ g_l(z_{l,1}) \\ \vdots \\ g_l(z_{l,n}) \end{bmatrix}$$



Example: Two-layer NN with ReLU hidden nodes

- Input: $h_0 = x$
- First linear layer:

$$z_1 = w_1 h_0$$

- First layer nonlinearity: ReLU

$$h_1 = \mathrm{ReLU}(z_1)$$

- Second linear layer:

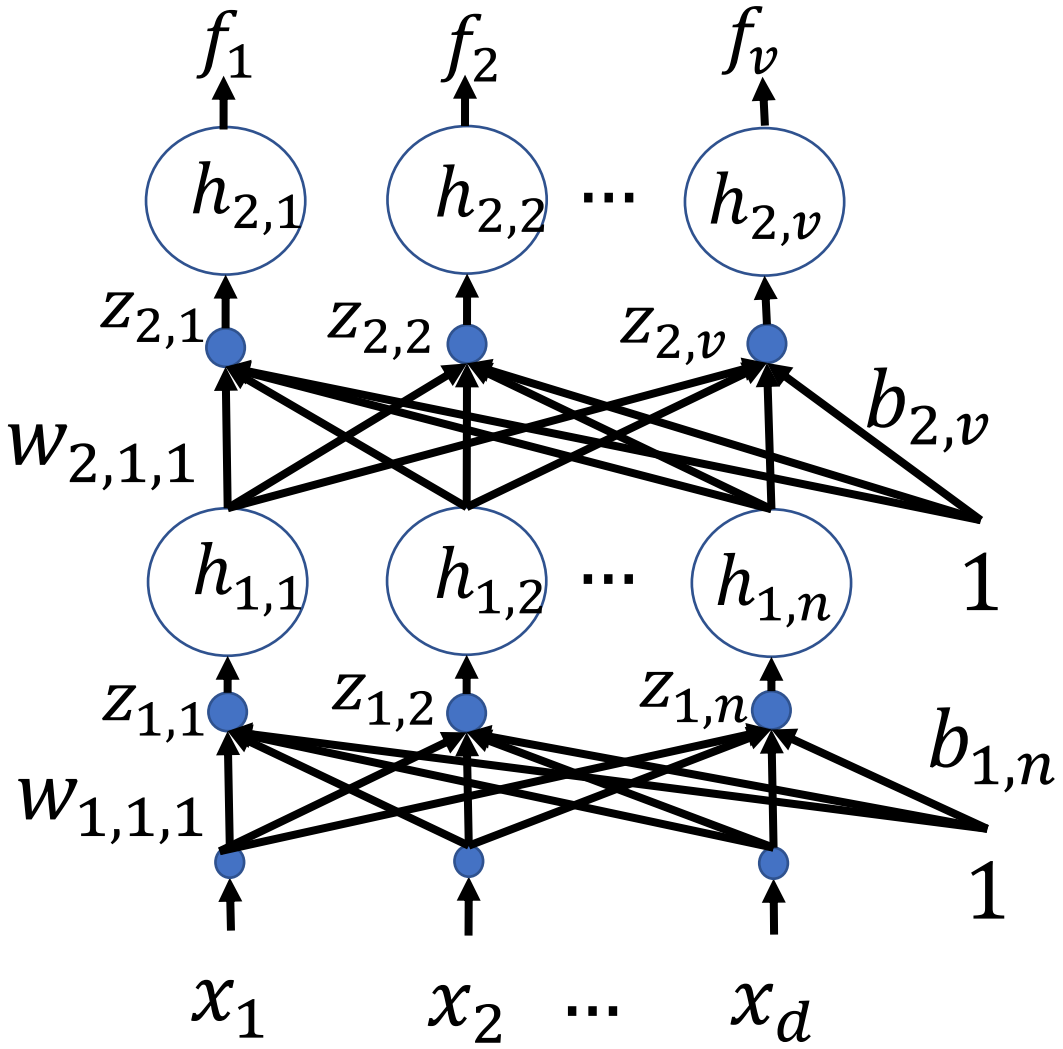
$$z_2 = w_2 h_1$$

- Second layer nonlinearity: softmax

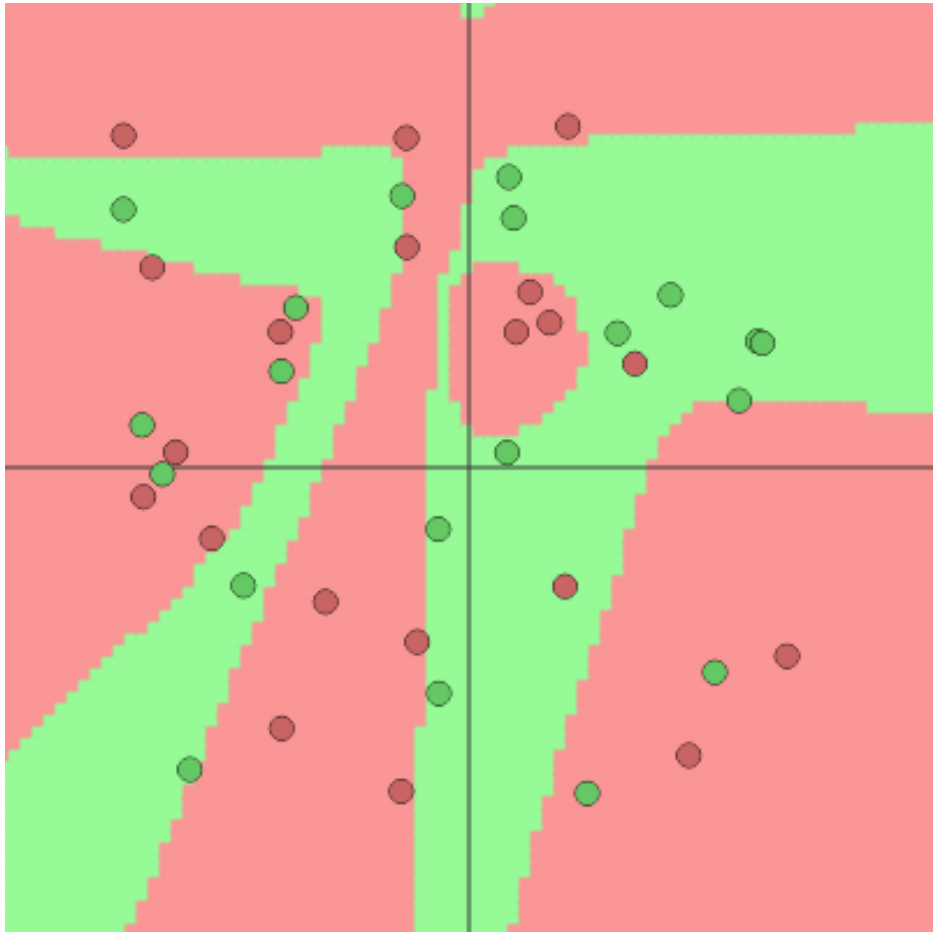
$$h_2 = \sigma(z_2)$$

- Output:

$$f(x) = h_2$$



Approximating an arbitrary nonlinear boundary using a two-layer network



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

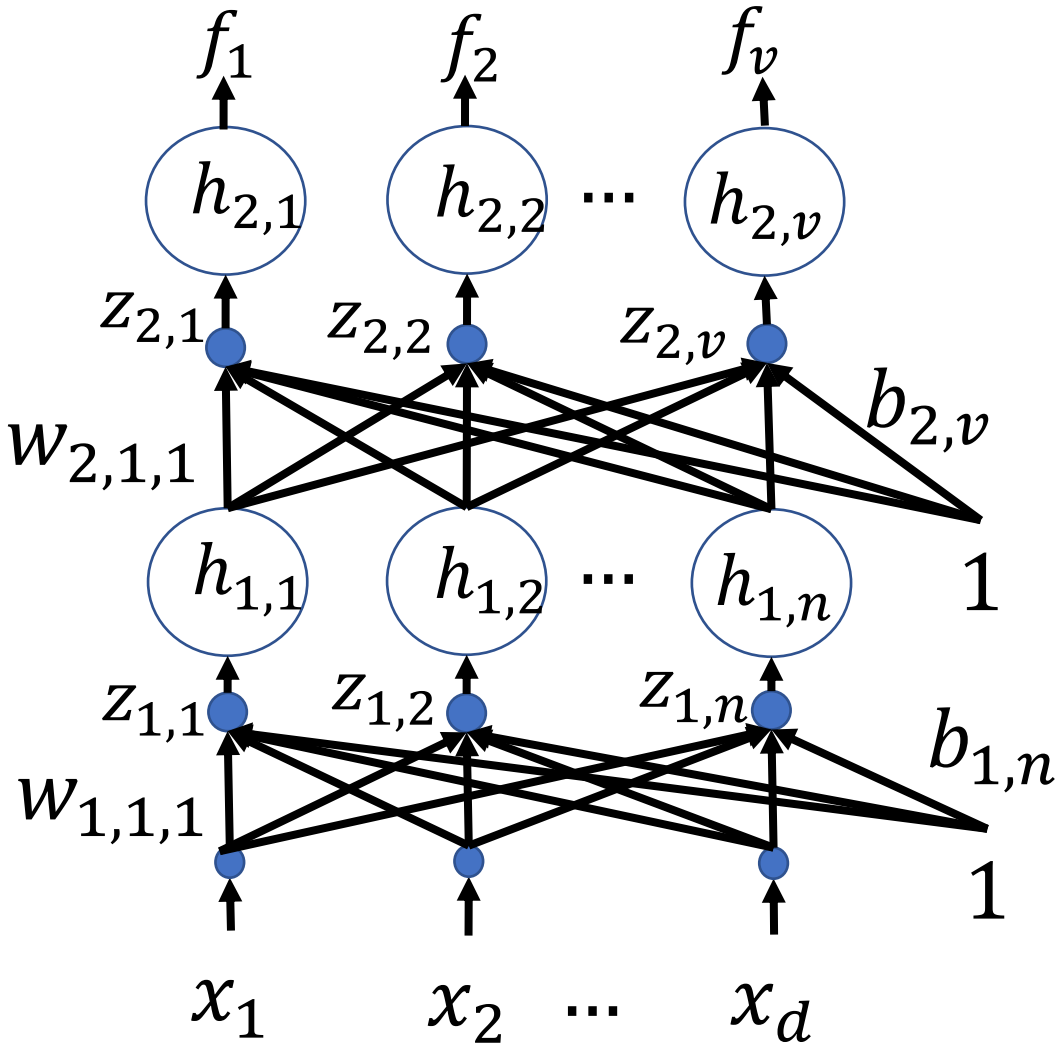
Outline

- Linear Classifiers: multi-class and 2-class
- Gradient descent
- One-hot vectors
- Softmax
- From linear to nonlinear classifiers
- Training a deep network: Back-propagation

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. the layer-l weight matrix, w_l, as:

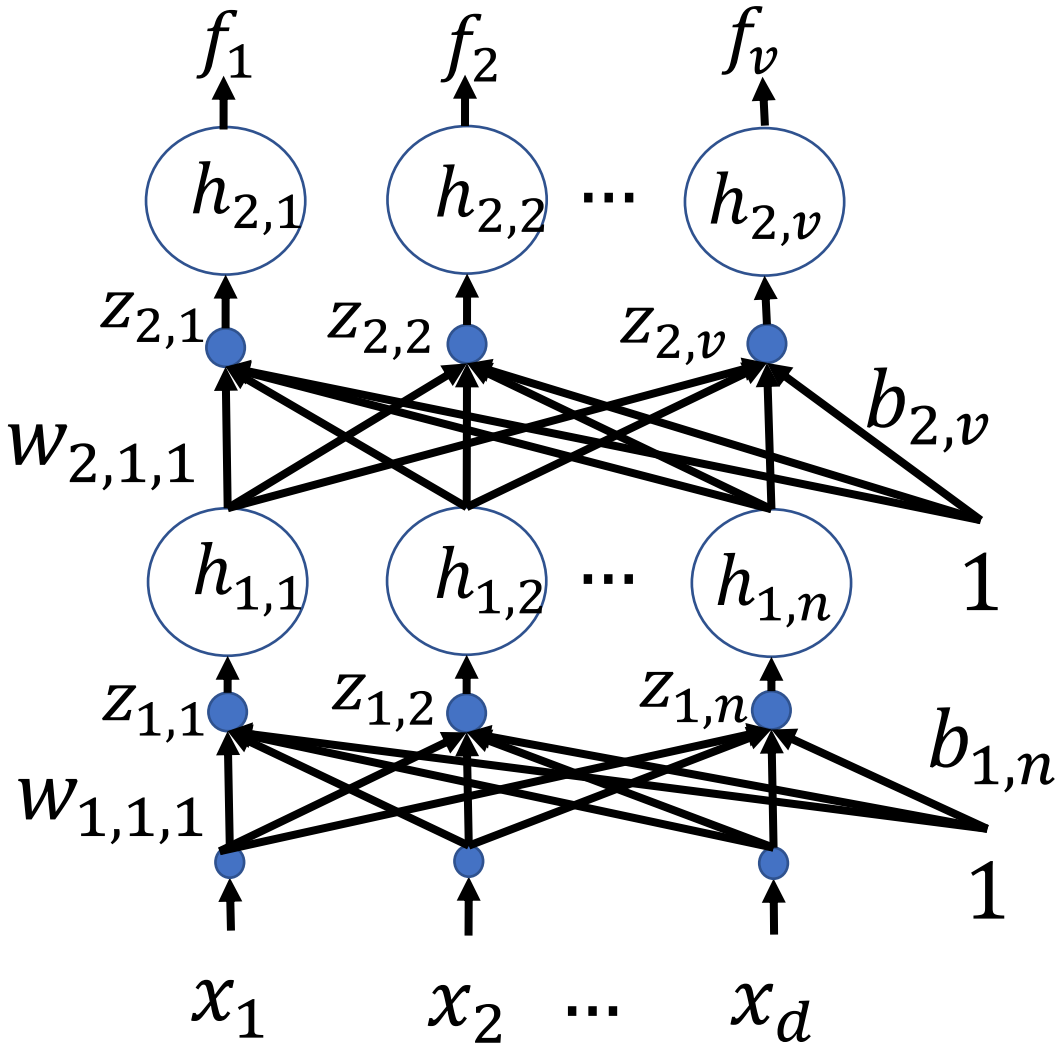
$$\frac{\partial \mathcal{L}}{\partial w_l} = \begin{bmatrix} \frac{\partial \mathcal{L}}{\partial b_{l,1}} & \frac{\partial \mathcal{L}}{\partial w_{l,1,1}} & \cdots \\ \frac{\partial \mathcal{L}}{\partial b_{l,2}} & \frac{\partial \mathcal{L}}{\partial w_{l,2,1}} & \cdots \\ \vdots & \vdots & \ddots \end{bmatrix}$$



Training a neural net: Gradient descent

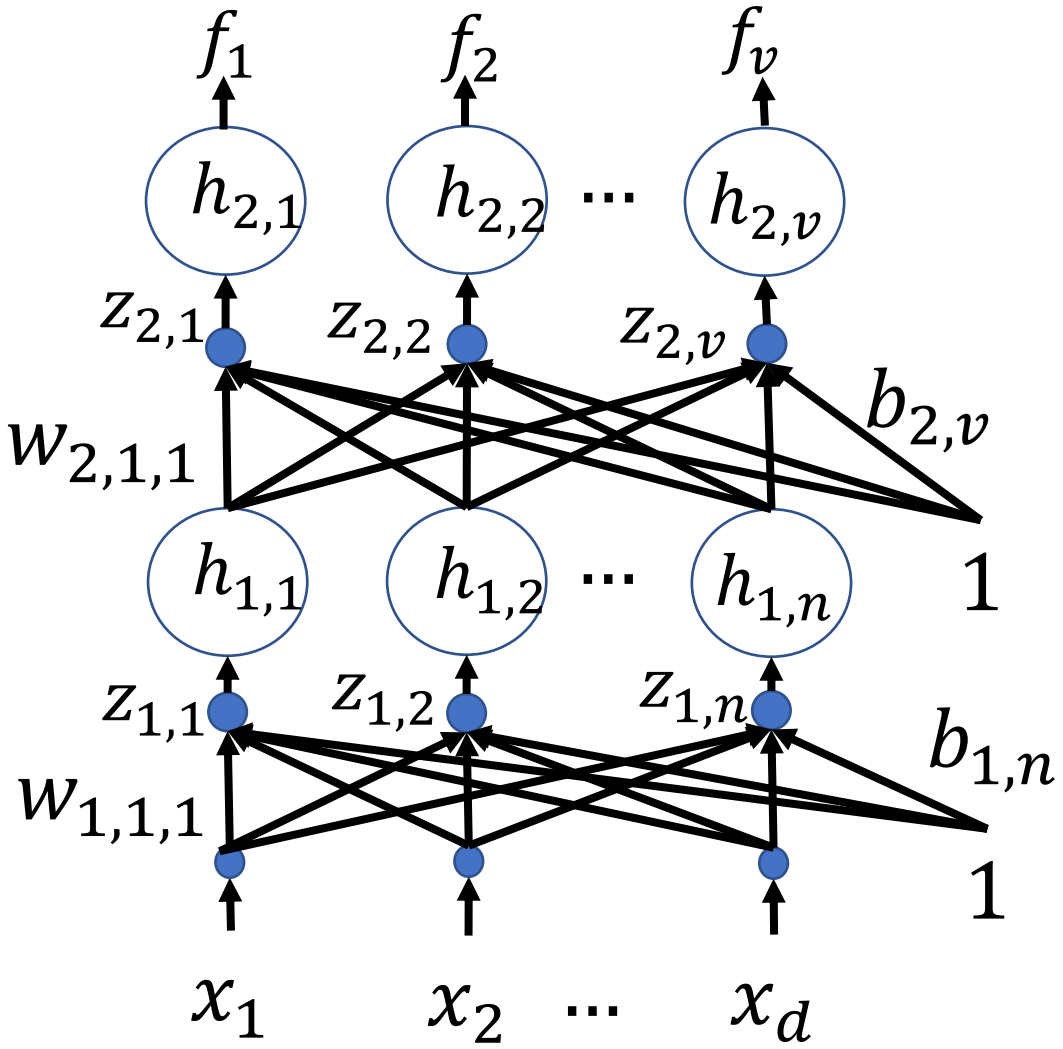
Gradient descent updates w_l as:

$$w_l \leftarrow w_l - \eta \frac{\partial \mathcal{L}}{\partial w_l}$$



Back-propagation = Chain rule

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

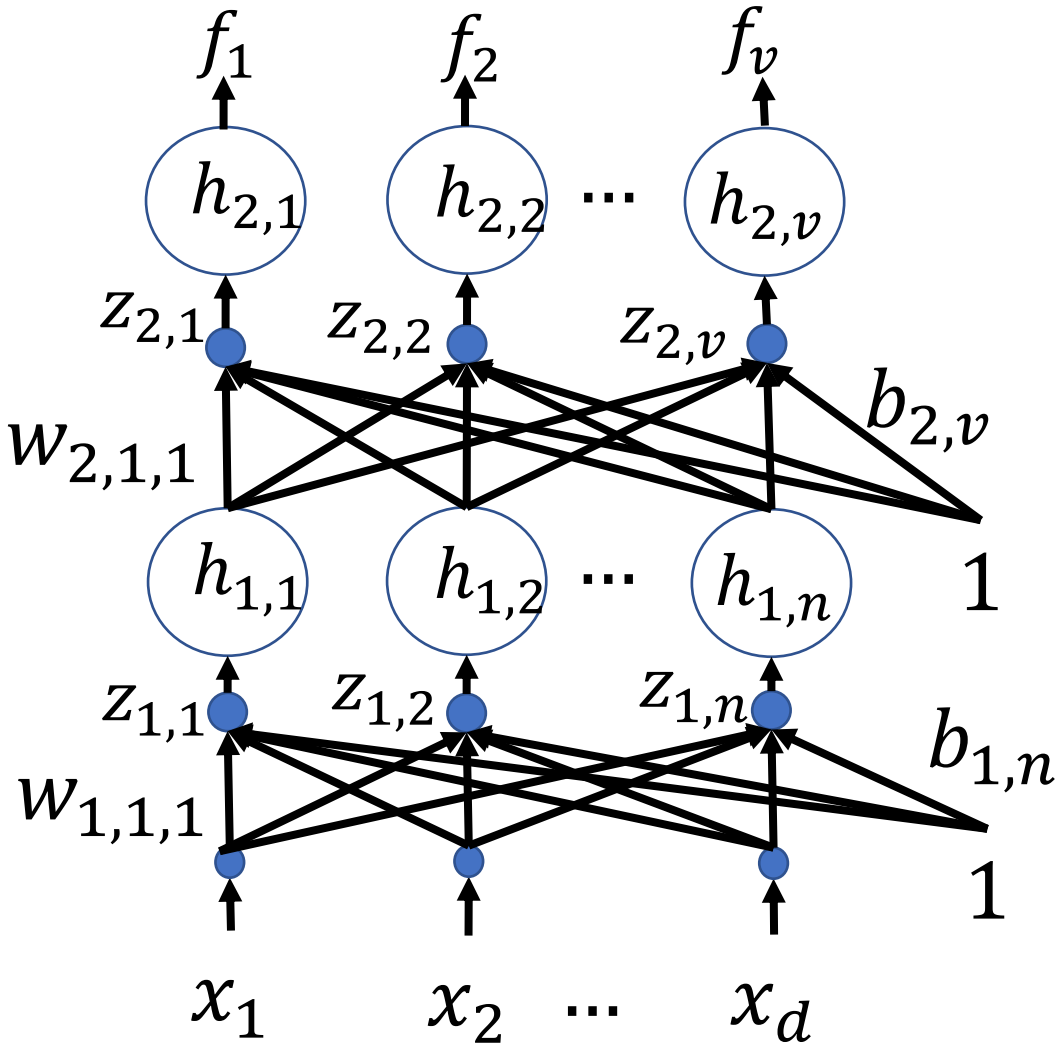


Back-propagation = Chain rule

$$\frac{\partial \mathcal{L}}{\partial h_{1,j}} = \sum_{k=1}^{v} \frac{\partial \mathcal{L}}{\partial z_{2,k}} \times \frac{\partial z_{2,k}}{\partial h_{1,j}}$$

$$\frac{\partial \mathcal{L}}{\partial z_{1,j}} = \frac{\partial \mathcal{L}}{\partial h_{1,j}} \times \frac{\partial h_{1,j}}{\partial z_{1,j}}$$

$$\frac{\partial \mathcal{L}}{\partial w_{1,j,k}} = \frac{\partial \mathcal{L}}{\partial z_{1,j}} \times \frac{\partial z_{1,j}}{\partial w_{1,j,k}}$$



Back-propagation = Chain rule

The key to backpropagation is that each individual derivative is easy. For example, if $\mathcal{L}$ is cross-entropy and $f(x) = \sigma(z_2)$, then you already know that

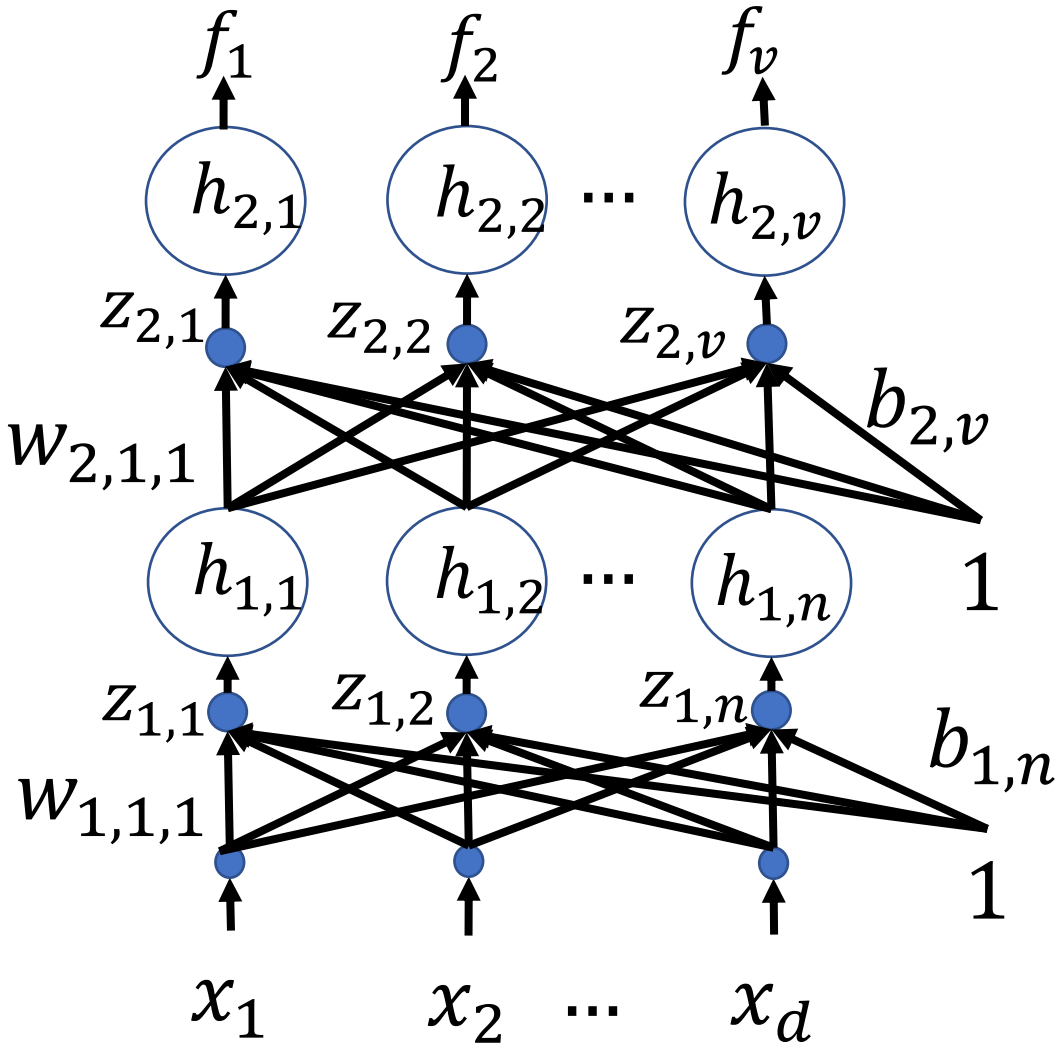
$$\frac{\partial \mathcal{L}}{\partial z_{2,k}} = f_k(x) - \mathbb{1}_{y=k}$$

Since $z_2 = w_2 h_1$ is just a matrix multiplication,

$$\frac{\partial z_{2,k}}{\partial h_{1,j}} = w_{2,k,j}$$

… and the derivative of ReLU is the unit step:

$$\frac{\partial h_{1,j}}{\partial z_{1,j}} = u(z_{1,j})$$



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, $\frac{\partial \mathcal{L}}{\partial w_2}$ and $\frac{\partial h_j}{\partial w_1}$
- Do a gradient update step, $w_l \leftarrow w_l - \eta \frac{\partial \mathcal{L}}{\partial w_l}$
- Repeat until the loss is small enough.

Outline

- Linear Classifiers: $f(x) = \operatorname{argmax} wx$
- Gradient descent: $w_c \leftarrow w_c - \eta \frac{\partial \mathcal{L}}{\partial w_c}$
- Cross-entropy: $\mathcal{L} = -\frac{1}{n}\sum_{i=1}^{n} \log f_{y_i}(x_i)$
- Softmax: $\sigma_c(wx) = \frac{\exp(w_c^T x)}{\sum_{k=1}^{v} \exp(w_k^T x)}$
- Gradient of the cross-entropy of the softmax: $\frac{\partial \sigma_y(z)}{\partial z_c} = \sigma_y(z)\left(\mathbb{1}_{y=c} - \sigma_c(z)\right)$
- Back-propagation: $\frac{\partial \mathcal{L}}{\partial h_{1,j}} = \sum_{k=1}^{v} \frac{\partial \mathcal{L}}{\partial z_{2,k}} \times \frac{\partial z_{2,k}}{\partial h_{1,j}}$, $\frac{\partial \mathcal{L}}{\partial z_{1,j}} = \frac{\partial \mathcal{L}}{\partial h_{1,j}} \times \frac{\partial h_{1,j}}{\partial z_{1,j}}$