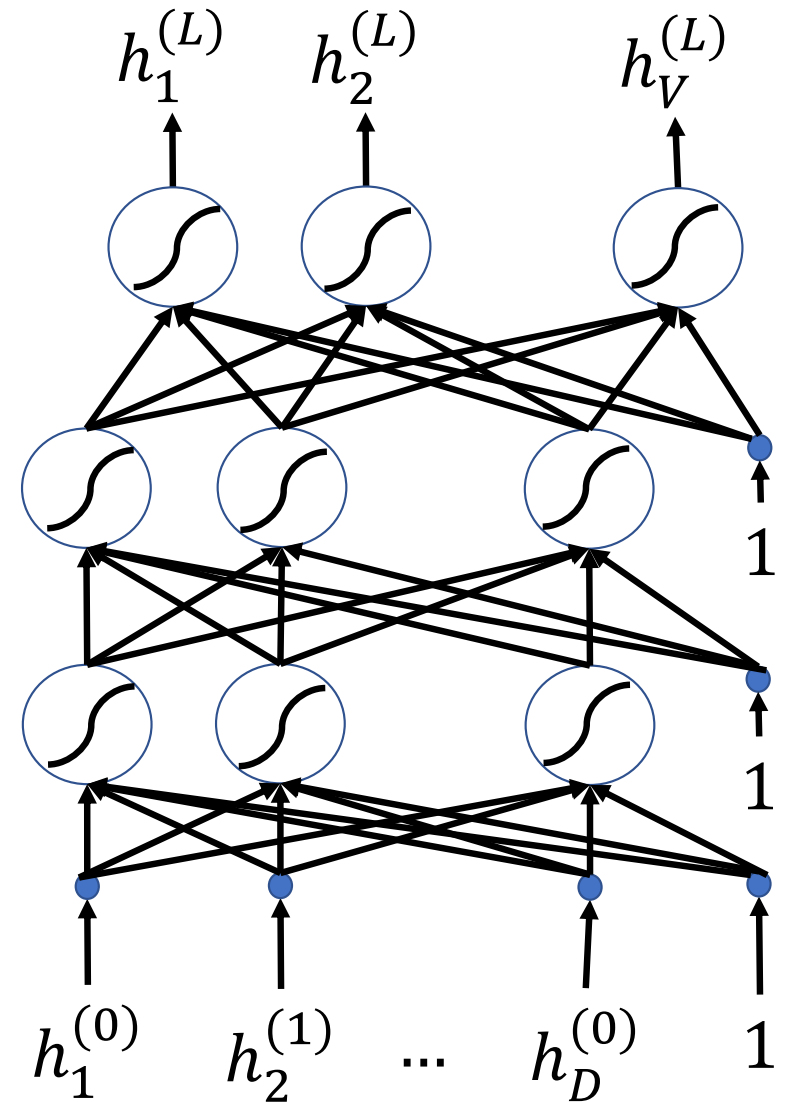


Deep Learning

Mark Hasegawa-Johnson, May 2022

Some slides by Svetlana Lazebnik

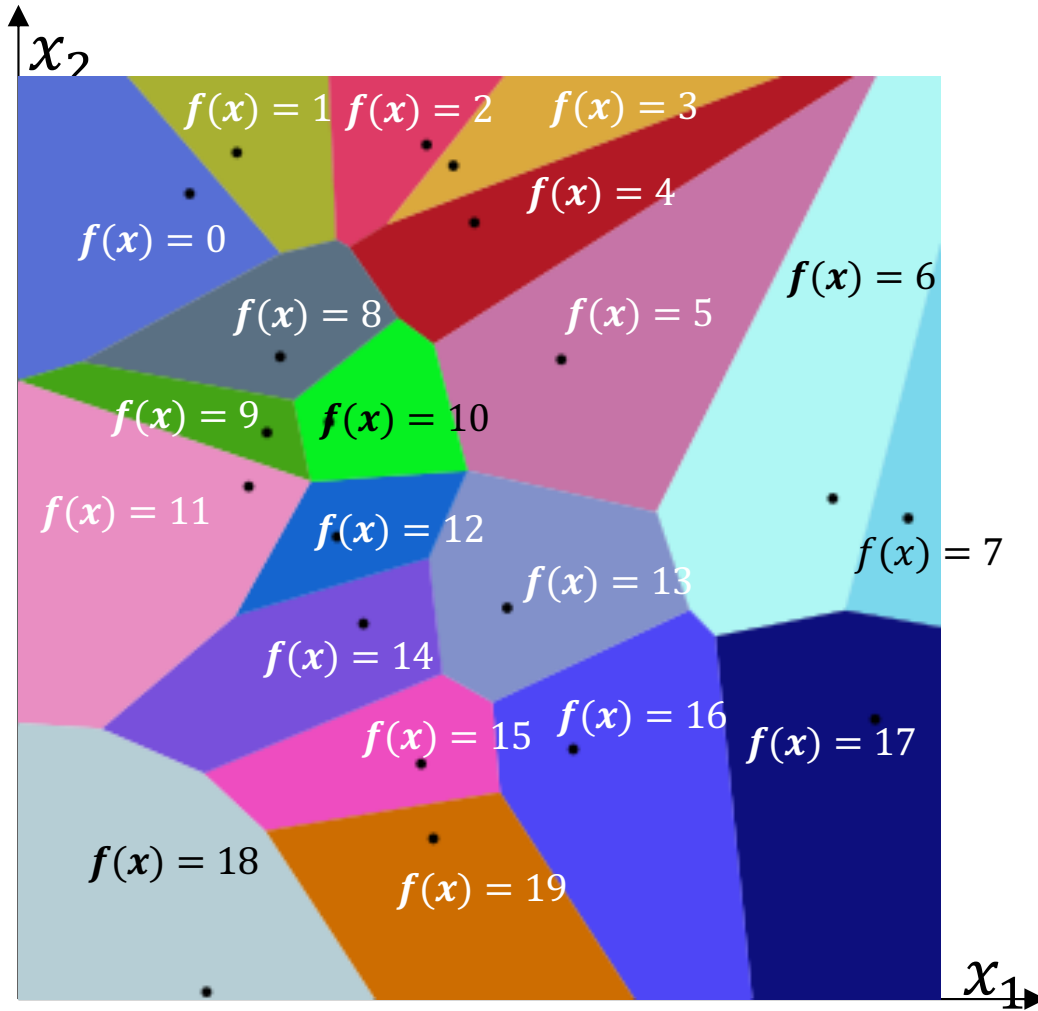
CC-BY 4.0: Copy or modify at will,
but please cite the source



Outline

- Review: multi-class perceptron
- Breaking the constraints of linearity: multi-layer neural nets
- Flow diagram for the XOR problem
- Flow diagram for a multi-layer neural net
- One-hot vectors
- Softmax
- Cross-entropy = negative log probability of the training data
- Stochastic gradient descent
- Training a neural net using numpy
- The same example using pytorch
- torch.nn: standard layers and units

Review: Multi-Class Perceptron



$$f(\vec{x}) = \underset{c}{\operatorname{argmax}}(\vec{w}_c^T \vec{x})$$

$$\vec{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_D \\ 1 \end{bmatrix}, \vec{w}_c = \begin{bmatrix} w_{c,1} \\ \vdots \\ w_{c,D} \\ b_c \end{bmatrix}$$

$$\vec{w}_c^T \vec{x} = b_c + \sum_{j=1}^D w_{c,j} x_j$$

Training Algorithm for the Multi-Class Perceptron

For each training instance \vec{x} with ground truth label $y \in \{-1, 1\}$:

- Classify with current weights: $f(\vec{x}) = \underset{c}{\operatorname{argmax}}(\vec{w}_c^T \vec{x})$

- Update weights:

- If $f(\vec{x}) = y$ then do nothing

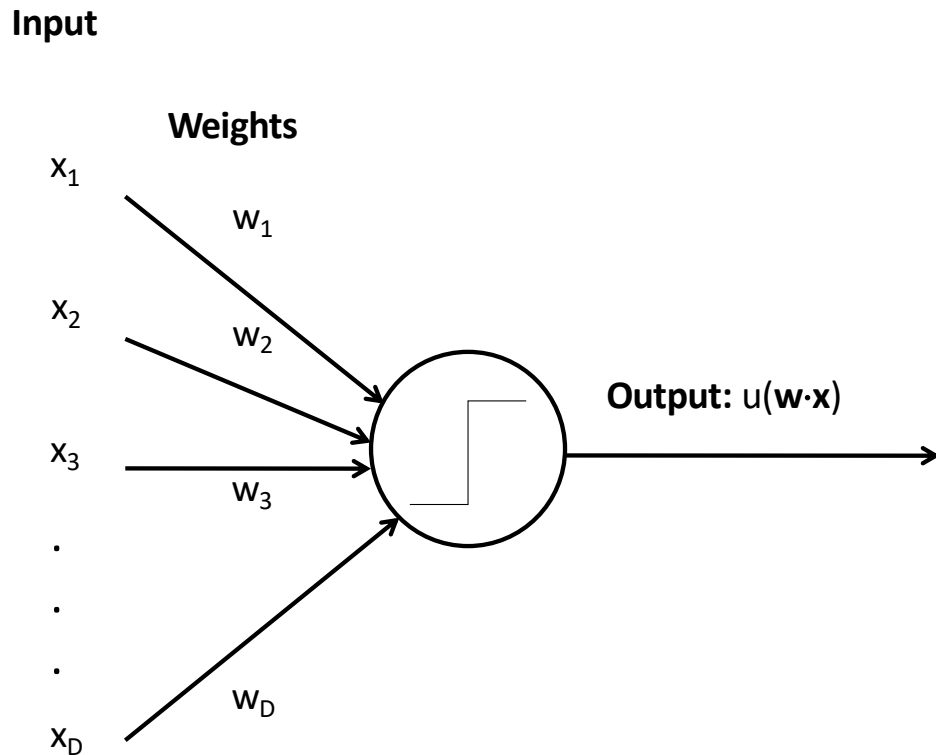
- If $f(\vec{x}) \neq y$ then

$$\begin{aligned}\vec{w}_y &= \vec{w}_y + \eta \vec{x} \\ \vec{w}_{f(\vec{x})} &= \vec{w}_{f(\vec{x})} - \eta \vec{x}\end{aligned}$$

Outline

- Review: multi-class perceptron
- Breaking the constraints of linearity: multi-layer neural nets
- Flow diagram for the XOR problem
- Flow diagram for a multi-layer neural net
- One-hot vectors
- Softmax
- Cross-entropy = negative log probability of the training data
- Stochastic gradient descent
- Training a neural net using numpy
- The same example using pytorch
- torch.nn: standard layers and units

Biological Inspiration: McCulloch-Pitts Artificial Neuron, 1943



- In 1943, McCulloch & Pitts proposed that biological neurons have a nonlinear activation function (a step function) whose input is a weighted linear combination of the currents generated by other neurons.
- They showed lots of examples of mathematical and logical functions that could be computed using networks of simple neurons like this.

Biological Inspiration: Neuronal Circuits

- Even the simplest actions involve more than one neuron, acting in sequence in a neuronal circuit.
- One of the simplest neuronal circuits is a reflex arc, which may contain just two neurons:
 - The **sensor neuron** detects a stimulus, and communicates an electrical signal to ...
 - The **motor neuron**, which activates the muscle.

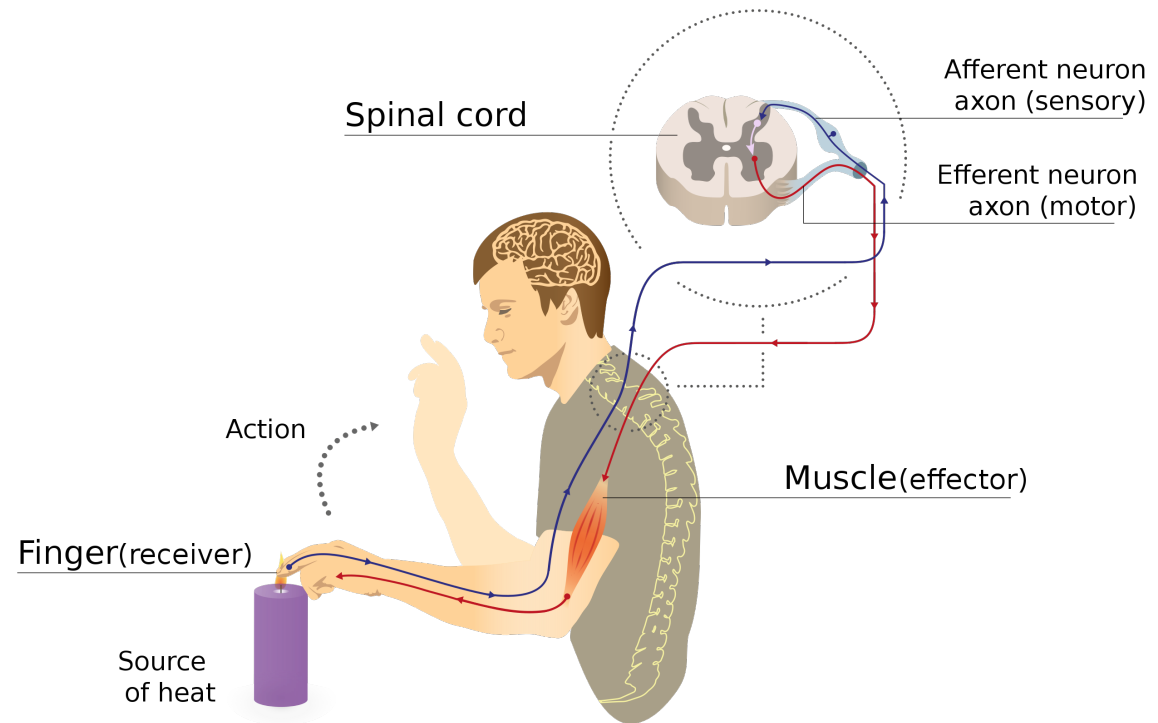


Illustration of a reflex arc: sensor neuron sends a voltage spike to the spinal column, where the resulting current causes a spike in a motor neuron, whose spike activates the muscle.

By MartaAguayo - Own work, CC BY-SA 3.0, <https://commons.wikimedia.org/w/index.php?curid=39181552>

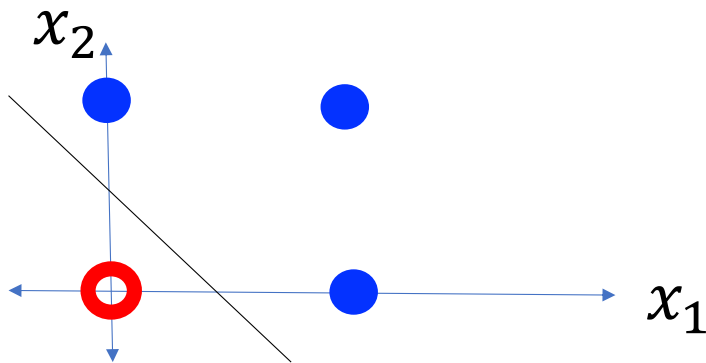
A McCulloch-Pitts Neuron can compute some logical functions...

When the features are binary ($x_j \in \{0,1\}$), many (but not all!) binary functions can be re-written as linear functions. For example, the function

$$f(\vec{x}) = (x_1 \vee x_2)$$

can be re-written as

$$f(\vec{x}) = u(x_1 + x_2 - 0.5)$$

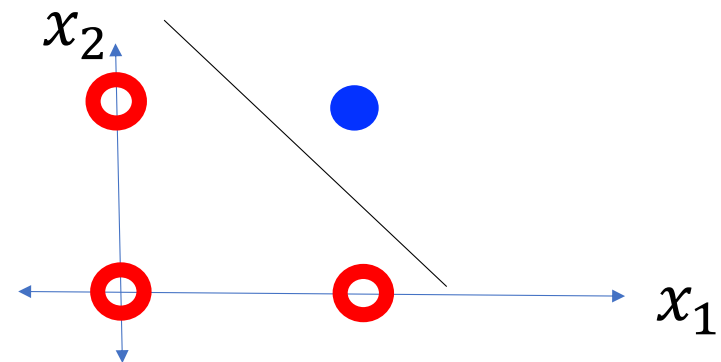


Similarly, the function

$$f(\vec{x}) = (x_1 \wedge x_2)$$

can be re-written as

$$f(\vec{x}) = u(x_1 + x_2 - 1.5)$$

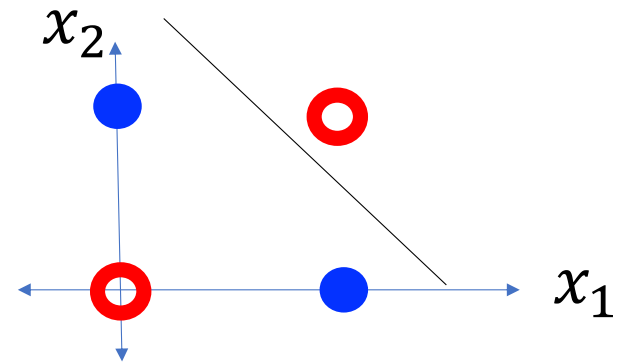


... but not all.

“A linear classifier cannot learn an XOR function.”

- Minsky & Papert, 1969

- ...but a *two-layer neural net* can compute an XOR function!




Outline


- Review: multi-class perceptron
- Breaking the constraints of linearity: multi-layer neural nets
- Flow diagram for the XOR problem
- Flow diagram for a multi-layer neural net
- One-hot vectors
- Softmax
- Cross-entropy = negative log probability of the training data
- Stochastic gradient descent
- Training a neural net using numpy
- The same example using pytorch
- torch.nn: standard layers and units

Example: one way (of many possible ways) to represent the XOR function using a two-layer neural network


For example, consider the XOR problem.

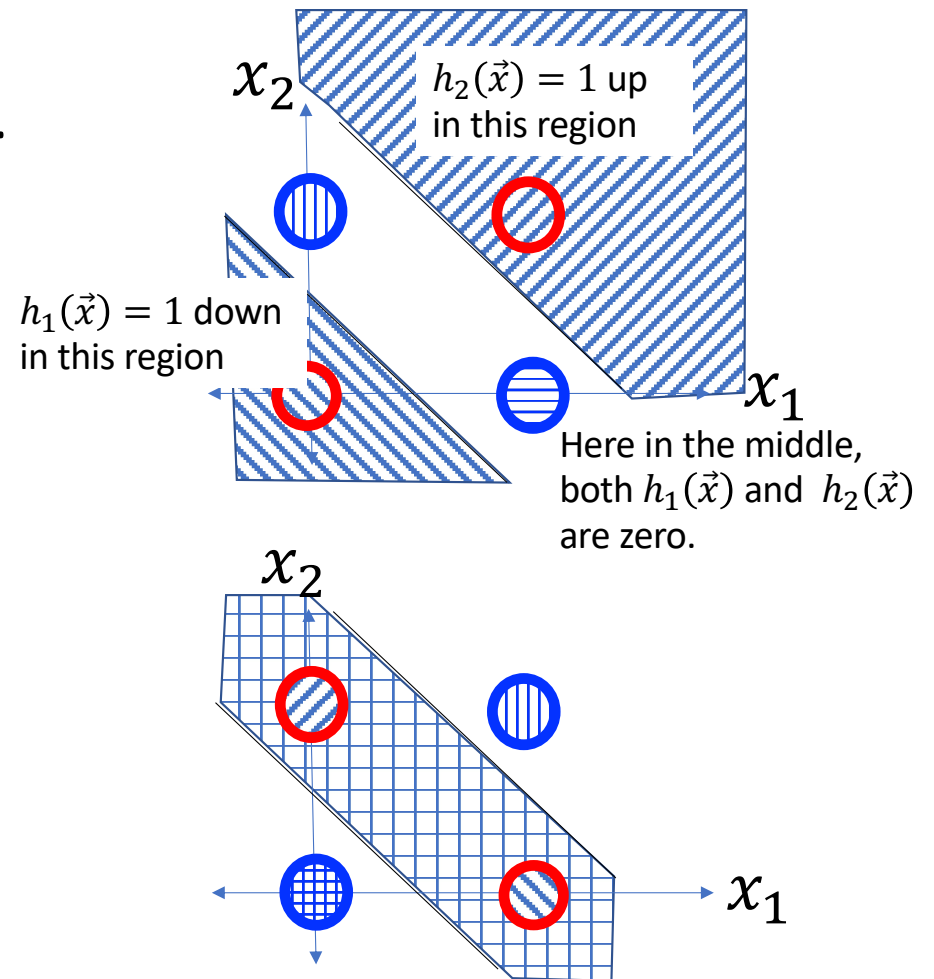
Suppose we create two **hidden nodes**:

 $h_1(\vec{x}) = u(0.5 - x_1 - x_2)$

 $h_2(\vec{x}) = u(x_1 + x_2 - 1.5)$


Then **the XOR function** $f(\vec{x}) = (x_1 \oplus x_2)$ is given by $f(\vec{x}) = \neg(h_1 \vee h_2)$. For example, we could write this as:


 $f(\vec{x}) = u(0.5 - h_1(x) - h_2(x))$



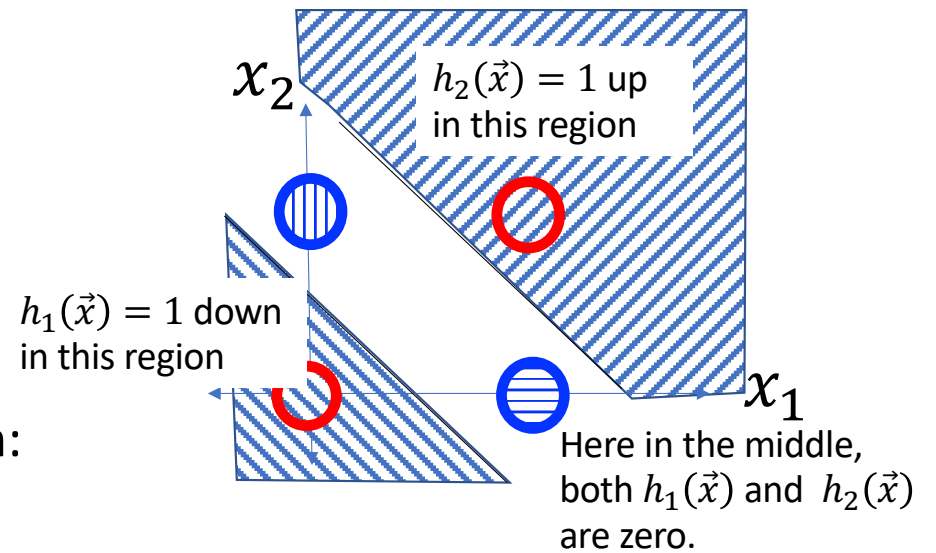
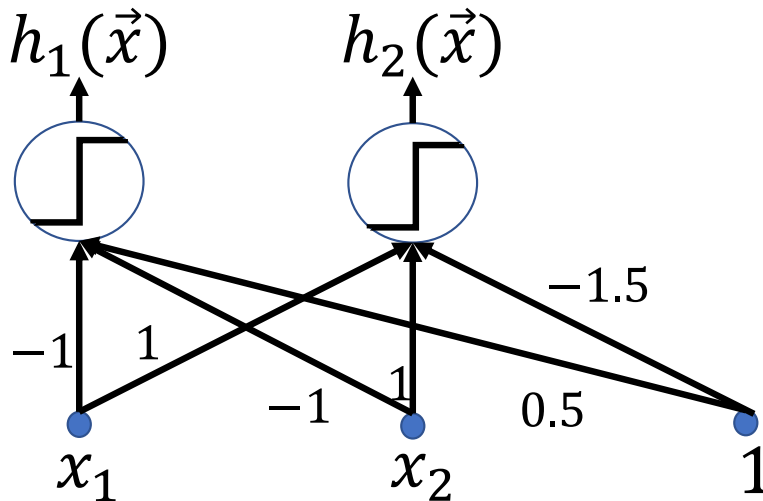
Flow diagrams

Suppose we create two **hidden nodes**:

 $h_1(\vec{x}) = u(0.5 - x_1 - x_2)$

 $h_2(\vec{x}) = u(x_1 + x_2 - 1.5)$

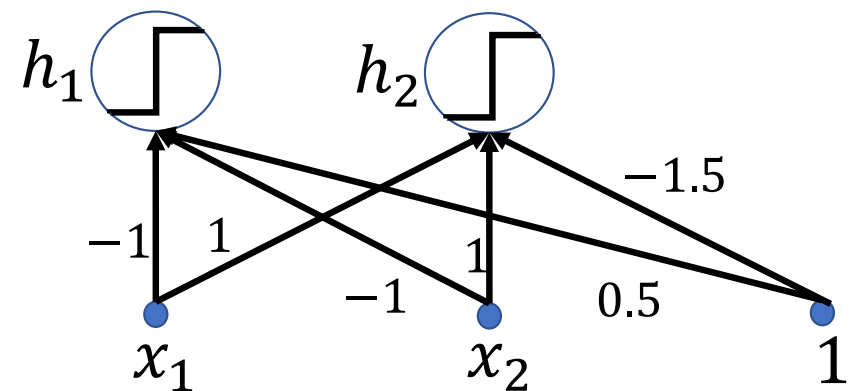
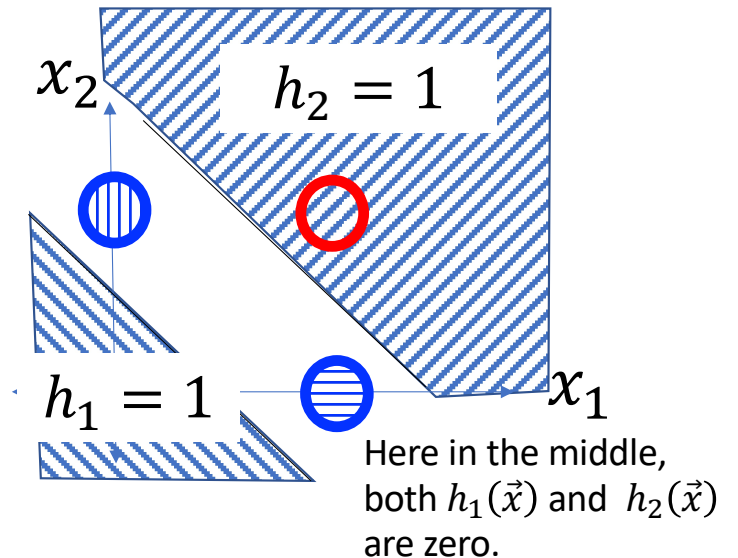
Here is a flow diagram for this computation:



Flow diagrams

A flow diagram is a way to represent the computations performed by a neural network.

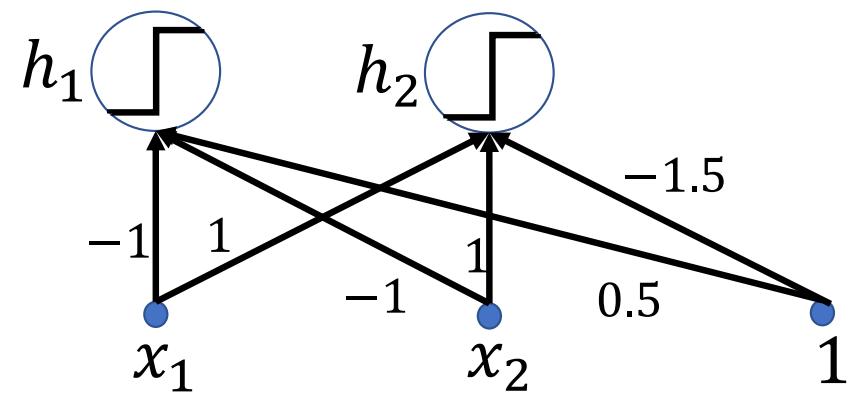
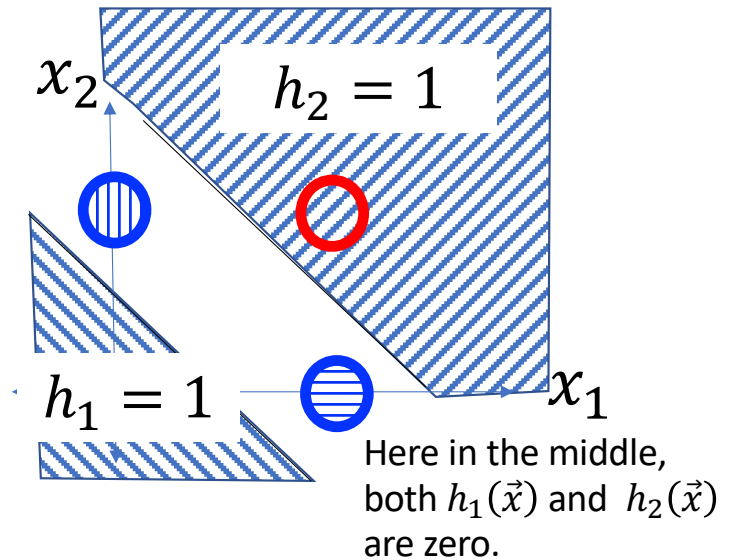
- Circles, a.k.a. “nodes,” a.k.a. “neurons,” represent scalar operations.
 - The circles above x_1 and x_2 represent the scalar operation of “read this datum in from the dataset.”
 - The circles labeled h_1 and h_2 represent the scalar operation of “unit step function.”
- Lines represent multiplication by a scalar.
- Where arrowheads come together, the corresponding variables are added.



Flow diagrams

It's often useful to distinguish two types of hidden variables at each neuron:

- The neural excitation, ξ_j , is the result of adding together all of the inputs to the neuron.
- The neural activation, h_j , is the result of passing ξ_j through a scalar nonlinearity.



Flow diagrams

So in this flow diagram, for example, we can see that:

$$\xi_1 = 0.5 - 1 \cdot x_1 - 1 \cdot x_2$$

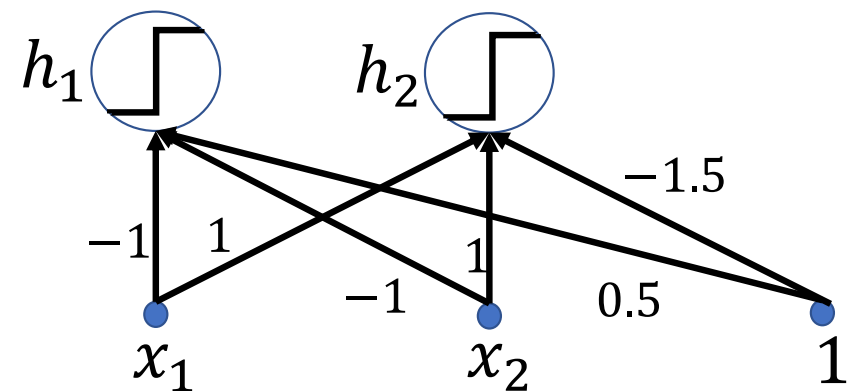
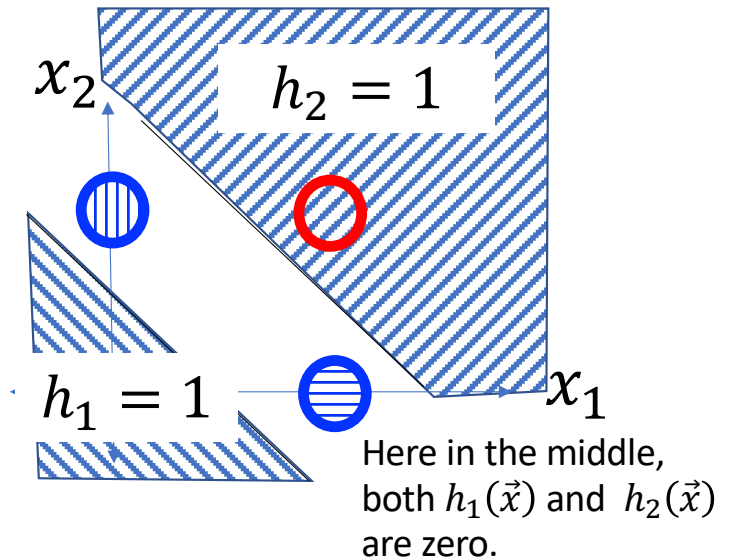
$$\xi_2 = -1.5 + 1 \cdot x_1 + 1 \cdot x_2$$

... and then ...

$$h_1 = u(\xi_1)$$

$$h_2 = u(\xi_2)$$

... where $u(\cdot)$ is the unit step function.



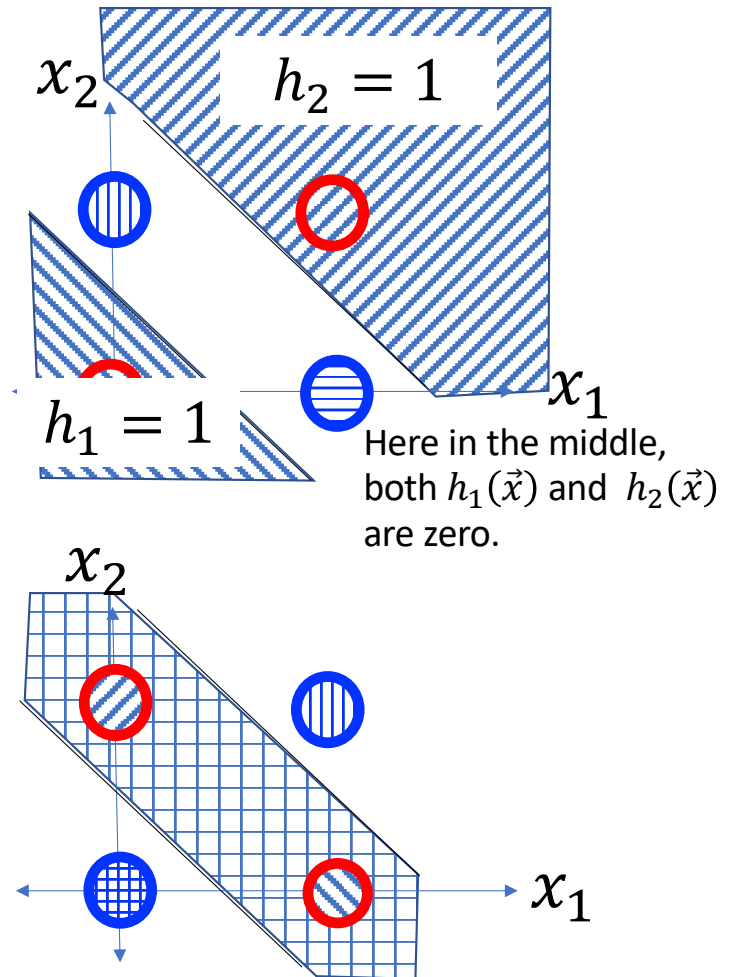
Flow diagrams

Now suppose that we want to compute

$$f(\vec{x}) = (x_1 \oplus x_2).$$

as:

$$f(\vec{x}) = u(0.5 - h_1 - h_2)$$

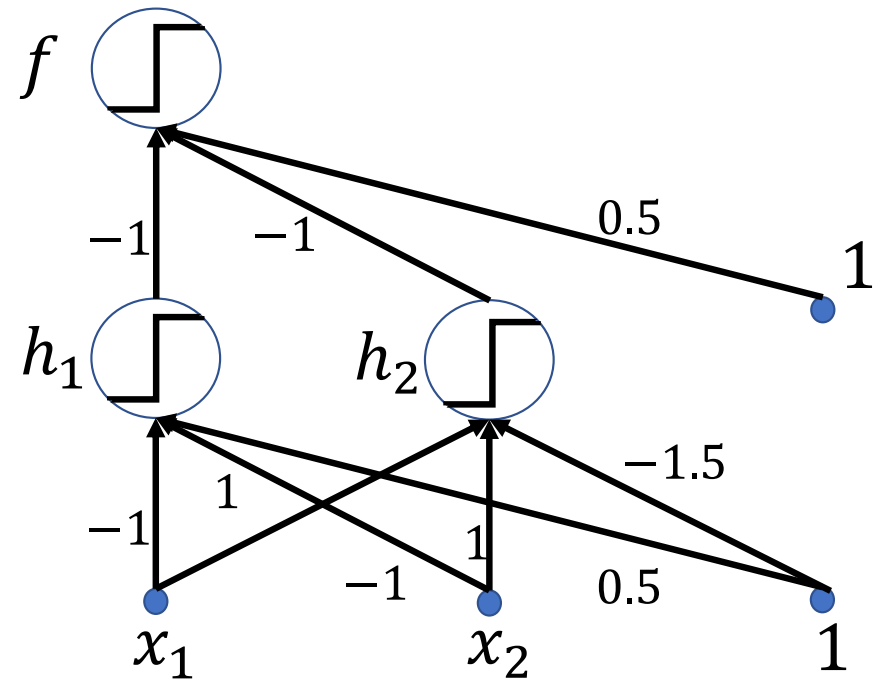
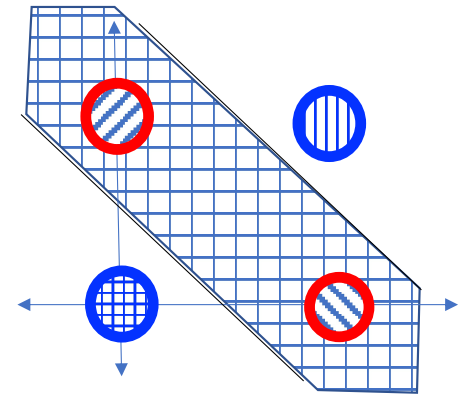


Flow diagrams

We can write the XOR function as:

$$\xi_3 = 0.5 - 1 \cdot h_1 - 1 \cdot h_2$$

$$f(\vec{x}) = u(\xi_3)$$



Flow diagrams

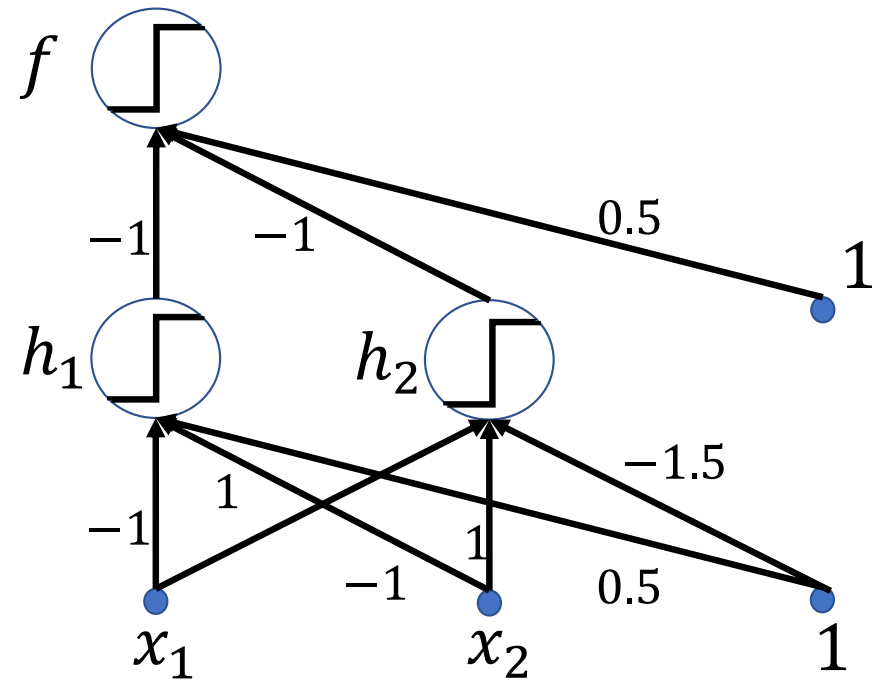
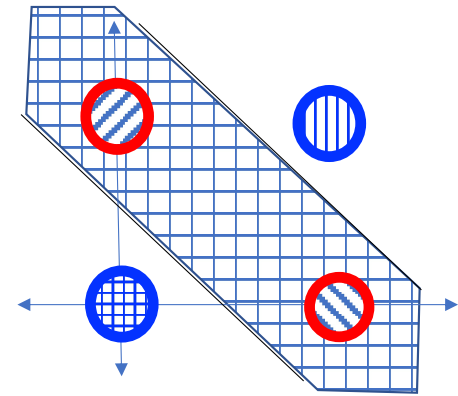
Putting it all together:

$$\xi_1 = 0.5 - 1 \cdot x_1 - 1 \cdot x_2$$
$$\xi_2 = -1.5 + 1 \cdot x_1 + 1 \cdot x_2$$

$$h_1 = u(\xi_1)$$
$$h_2 = u(\xi_2)$$

$$\xi_3 = 0.5 - 1 \cdot h_1 - 1 \cdot h_2$$

$$f(\vec{x}) = u(\xi_3)$$



Outline

- Review: multi-class perceptron
- Breaking the constraints of linearity: multi-layer neural nets
- Flow diagram for the XOR problem
- Flow diagram for a multi-layer neural net
- One-hot vectors
- Softmax
- Cross-entropy = negative log probability of the training data
- Stochastic gradient descent
- Training a neural net using numpy
- The same example using pytorch
- torch.nn: standard layers and units

Multi-layer neural net

- $\xi_j^{(l)}$ = **excitation** of the j^{th} neuron (a.k.a. “node”) in the l^{th} layer
 - Computed by adding together inputs from many other neurons, each weighted by a corresponding connection strength or connection weight, $w_{j,k}^{(l)}$
- $h_j^{(l)}$ = **activation** of the j^{th} node in the l^{th} layer
 - This is computed by just passing the excitation through a scalar nonlinear activation function, thus $h_j^{(l)} = g(\xi_j^{(l)})$. The activation functions in different layers differ, so to be pedantic, sometimes we’ll write $h_j^{(l)} = g^{(l)}(\xi_j^{(l)})$.

Multi-layer neural net

Given: some training token $\vec{x} = [x_1, \dots, x_D, 1]^T$ and its target label y

1. Initialize: $h_k^{(0)} = x_k$
2. Forward propagate: for $l \in \{1, \dots, L\}$:
 - a. Compute the excitations as weighted sums of the previous-layer activations:

$$\xi_j^{(l)} = b_j^{(l)} + \sum_k w_{j,k}^{(l)} h_k^{(l-1)}$$

- b. Compute the activations by applying scalar nonlinearities:

$$h_j^{(l)} = g^{(l)}(\xi_j^{(l)})$$

3. Output: $P(Y = k|x) = h_k^{(L)}$

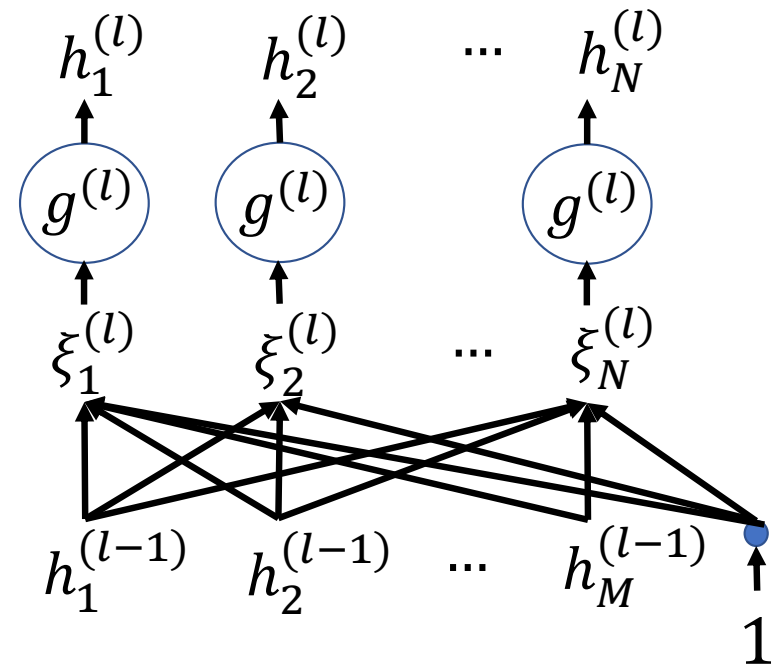
Forward propagation

- From activation to excitation is a matrix multiply:

$$\xi_j^{(l)} = b_j^{(l)} + \sum_k w_{j,k}^{(l)} h_k^{(l-1)}$$

- From excitation to activation is a scalar nonlinearity:

$$h_j^{(l)} = g^{(l)} \left(\xi_j^{(l)} \right)$$



Forward propagation: Matrix multiply

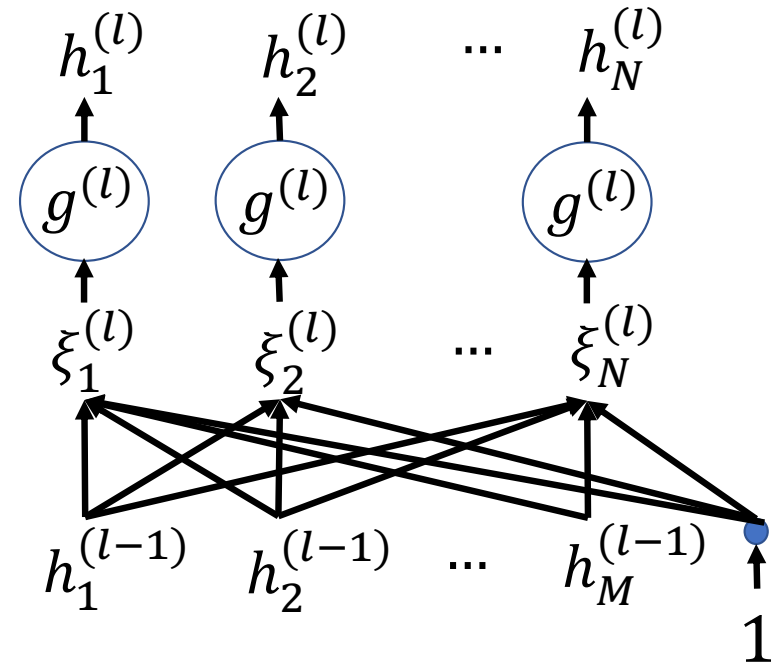
From activation to excitation is a matrix multiply:

$$\vec{\xi}^{(l)} = W^{(l)} \vec{h}^{(l-1)}$$

...where...

$$\vec{\xi}^{(l)} = \begin{bmatrix} \xi_1^{(l)} \\ \vdots \\ \xi_N^{(l)} \end{bmatrix}, \quad \vec{h}^{(l-1)} = \begin{bmatrix} h_1^{(l-1)} \\ \vdots \\ h_M^{(l-1)} \\ 1 \end{bmatrix},$$

$$W^{(l)} = \begin{bmatrix} w_{1,1}^{(l)} & \cdots & w_{1,M}^{(l)} & b_1^{(l)} \\ \vdots & \ddots & \vdots & \vdots \\ w_{N,1}^{(l)} & \cdots & w_{N,M}^{(l)} & b_N^{(l)} \end{bmatrix}$$



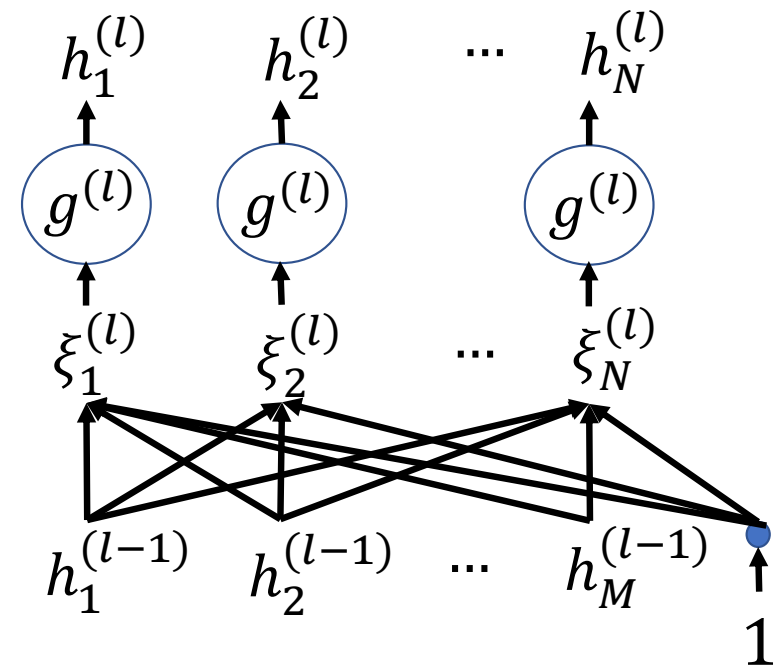
Forward propagation

From excitation to activation is a scalar nonlinearity:

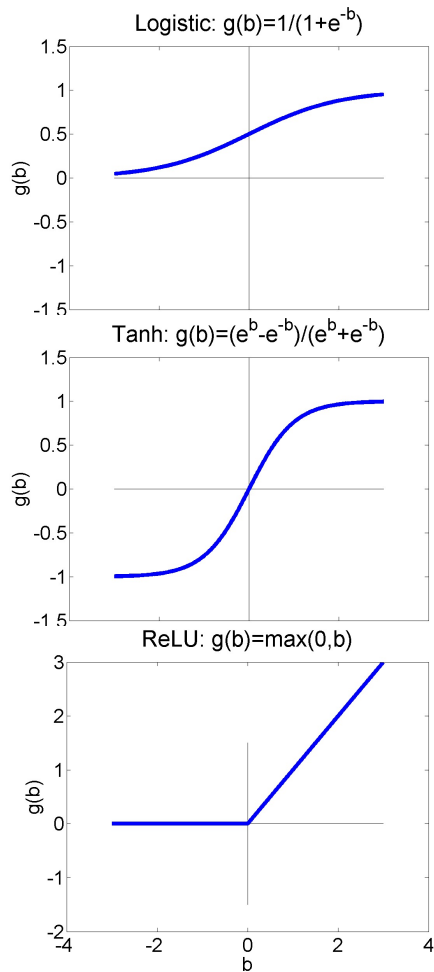
$$h_j^{(l)} = g^{(l)}(\xi_j^{(l)})$$

What type of nonlinearity?

Answer: it depends on what task you want your neural net to learn.



Activation functions



The “activation function,” $g^{(l)}(\cdot)$, can be any scalar nonlinearity. Common ones that you should know include the unit step and signum functions, and:

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):

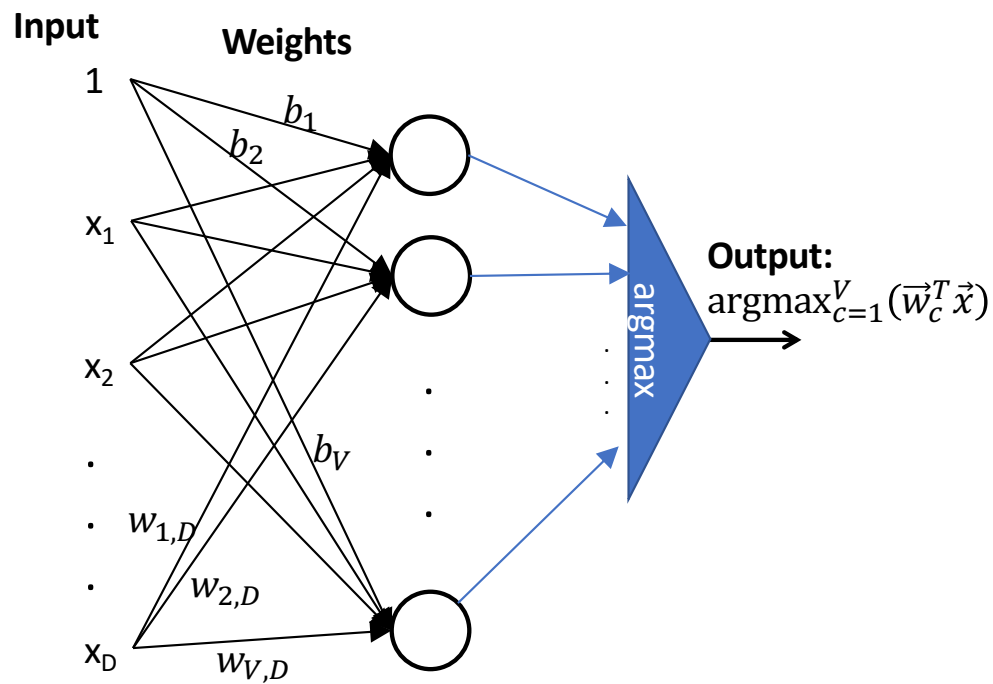
$$\text{ReLU}(\beta) = \max(0, \beta)$$

Outline

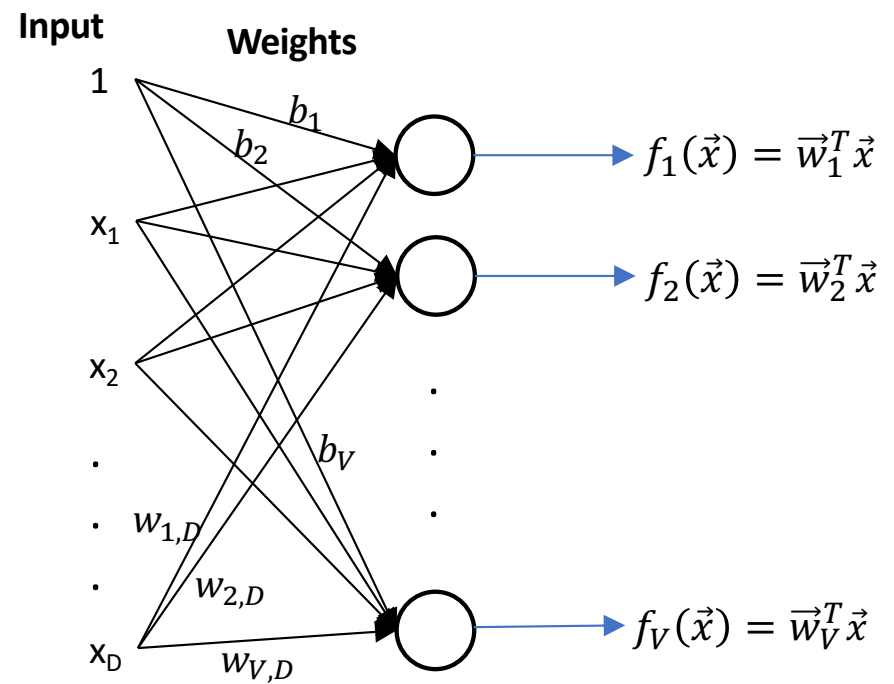
- Review: multi-class perceptron
- Breaking the constraints of linearity: multi-layer neural nets
- Flow diagram for the XOR problem
- Flow diagram for a multi-layer neural net
- One-hot vectors
- Softmax
- Cross-entropy = negative log probability of the training data
- Stochastic gradient descent
- Training a neural net using numpy
- The same example using pytorch
- torch.nn: standard layers and units

Comparison of Multi-Class Perceptron to Multiple Linear Regression

Multi-Class Perceptron



Multiple Linear Regression



Here's a weird question:

Can we come up with some new notation that can be used to write both the multi-class perceptron AND the linear regression algorithm?

New notation: Don't change the multi-class perceptron algorithm, but make it easier to write

- Instead of defining y_i as an integer, let's define \vec{y}_i to be a vector:

$$\vec{y}_i = \begin{bmatrix} y_{i,1} \\ \vdots \\ y_{i,V} \end{bmatrix}$$

- For a multi-class perceptron, this only makes sense if \vec{y}_i is what's called a **one-hot** vector:

$$y_{i,c} = \begin{cases} 1 & c = \text{true class label of the } i^{\text{th}} \text{ token} \\ 0 & \text{otherwise} \end{cases}$$

New notation: Don't change the multi-class perceptron algorithm, but make it easier to write

- Let's also define the output to be a one-hot vector:

$$f(\vec{x}_i) = \begin{bmatrix} f_1(\vec{x}_i) \\ \vdots \\ f_V(\vec{x}_i) \end{bmatrix}$$

... where ...

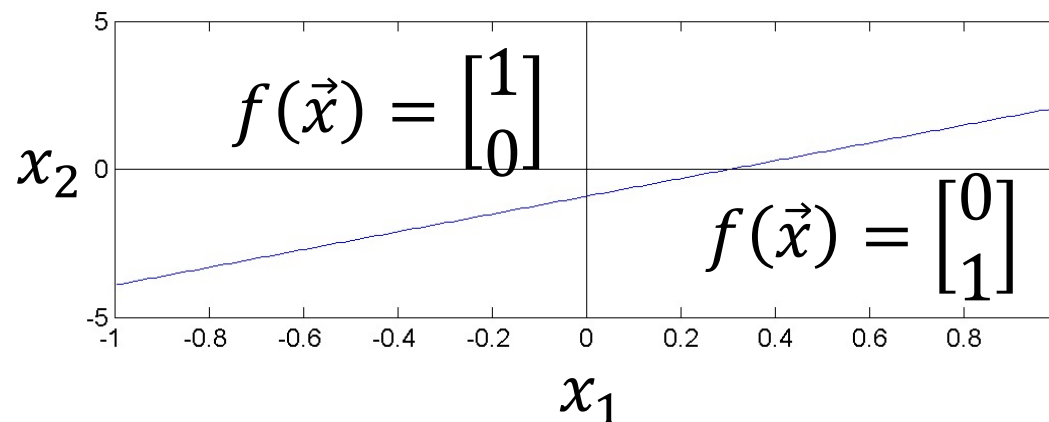
$$f_c(\vec{x}_i) = \begin{cases} 1 & c = \operatorname{argmax} \vec{w}_c^T \vec{x} \\ 0 & \text{otherwise} \end{cases}$$

Example: Binary classifier

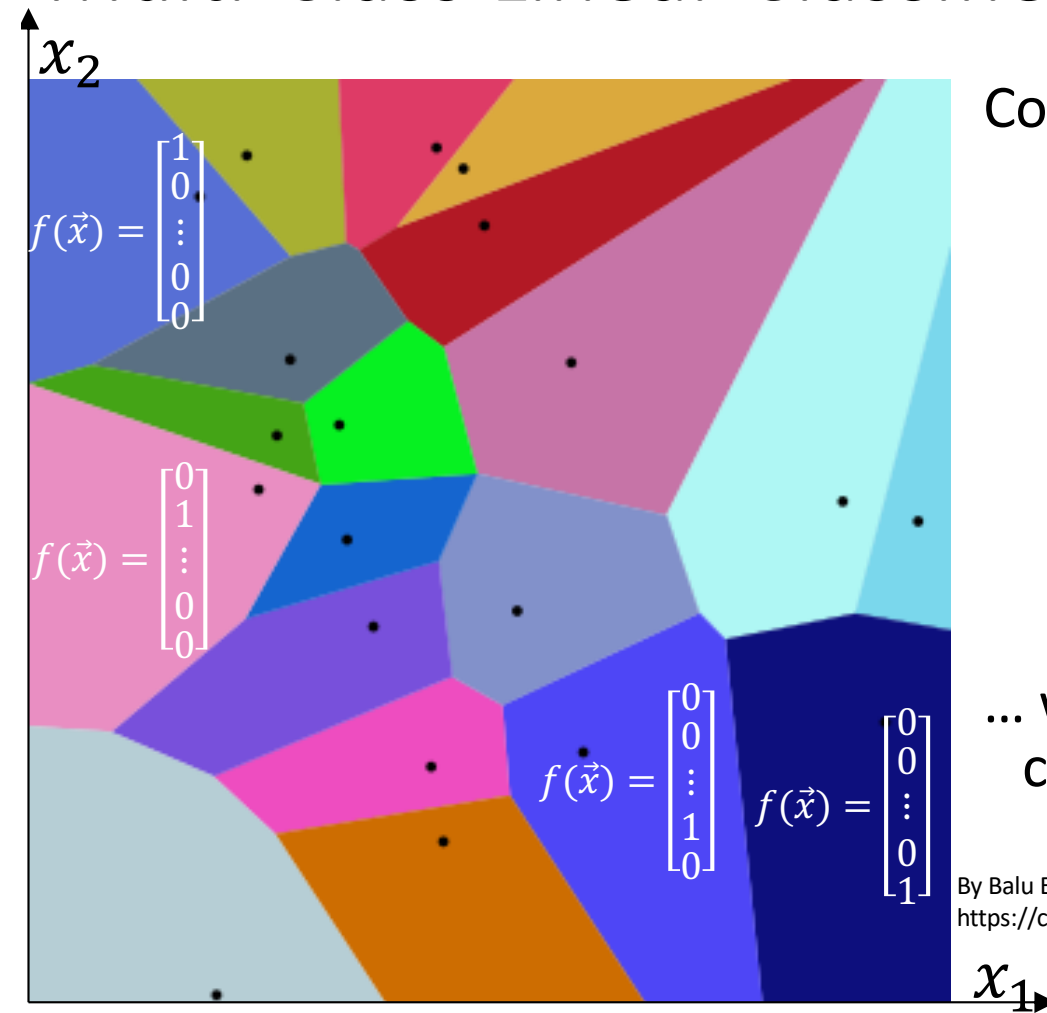
Consider the classifier

$$f(\vec{x}_i) = \begin{bmatrix} f_1(\vec{x}_i) \\ f_2(\vec{x}_i) \end{bmatrix}, \quad f_c(\vec{x}_i) = \begin{cases} 1 & c = \operatorname{argmax} \vec{w}_c^T \vec{x} \\ 0 & \text{otherwise} \end{cases}$$

... with only two classes. Then the classification regions might look like this:



Multi-Class Linear Classifiers



Consider the classifier

$$f(\vec{x}_i) = \begin{bmatrix} f_1(\vec{x}_i) \\ \vdots \\ f_V(\vec{x}_i) \end{bmatrix},$$

$$f_c(\vec{x}_i) = \begin{cases} 1 & c = \operatorname{argmax} \vec{w}_c^T \vec{x} \\ 0 & \text{otherwise} \end{cases}$$

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

By Balu Ertl - Own work, CC BY-SA 4.0,
<https://commons.wikimedia.org/w/index.php?curid=38534275>

Now the perceptron has a vector error, just like linear regression

Now we can define an error term for every output:

$$\vec{\epsilon}_i = \begin{bmatrix} \epsilon_{i,1} \\ \vdots \\ \epsilon_{i,V} \end{bmatrix}, \quad \epsilon_{i,c} = f_c(\vec{x}_i) - y_{i,c}$$

- If c was the correct class label ($y_{i,c} = 1$), but the network didn't get it right ($f_c(\vec{x}_i) = 0$), then it **undershot**:

$$\epsilon_{i,c} = -1$$

- If the network thought the correct answer was c ($f_c(\vec{x}_i) = 1$), but it wasn't ($y_{i,c} = 0$), then it **overshot**

$$\epsilon_{i,c} = +1$$

- Otherwise,

$$\epsilon_{i,c} = 0$$

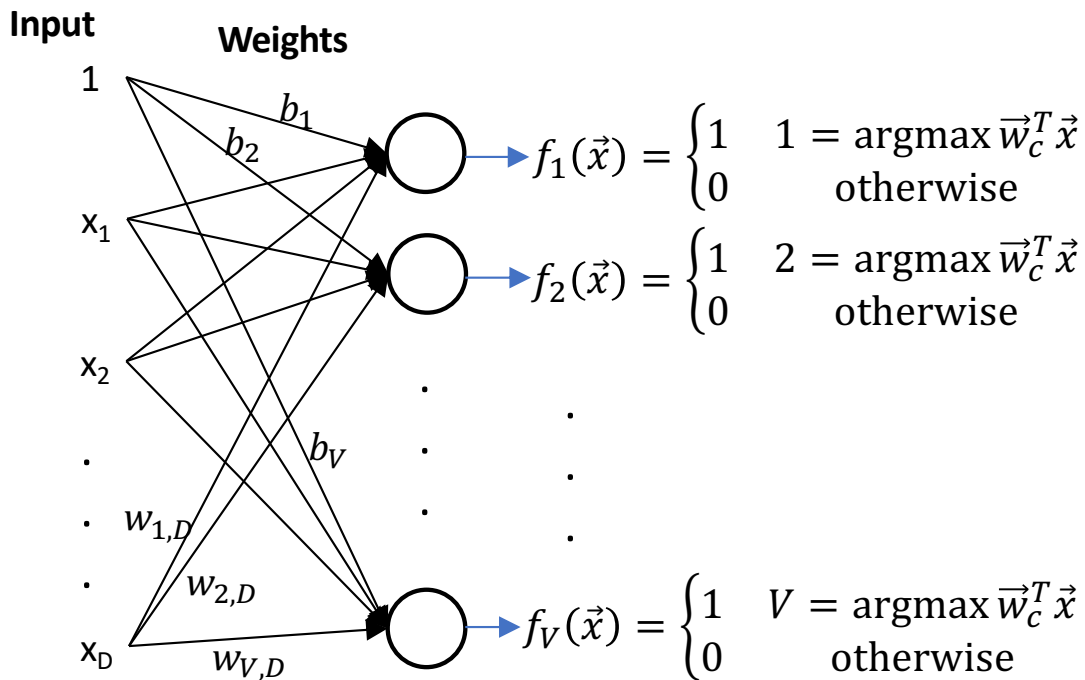
Multi-class perceptron, written in terms of one-hot vectors

But with this definition, we can write the perceptron update the same as the linear regression update:

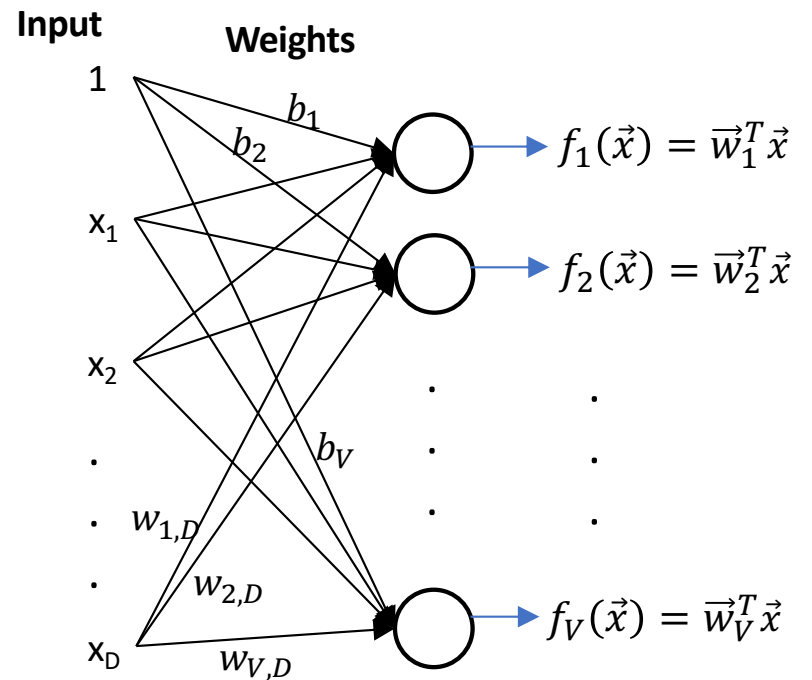
$$\vec{w}_c \leftarrow \vec{w}_c - \eta \epsilon_{i,c} \vec{x}_i = \begin{cases} \vec{w}_c + \eta \vec{x}_i & \epsilon_{i,c} = -1 \\ \vec{w}_c - \eta \vec{x}_i & \epsilon_{i,c} = +1 \\ \vec{w}_c & \epsilon_{i,c} = 0 \end{cases}$$

Comparison of Multi-Class Perceptron to Multiple Regression

Multi-Class Perceptron: One-hot output



Multiple Regression: Real-valued Output

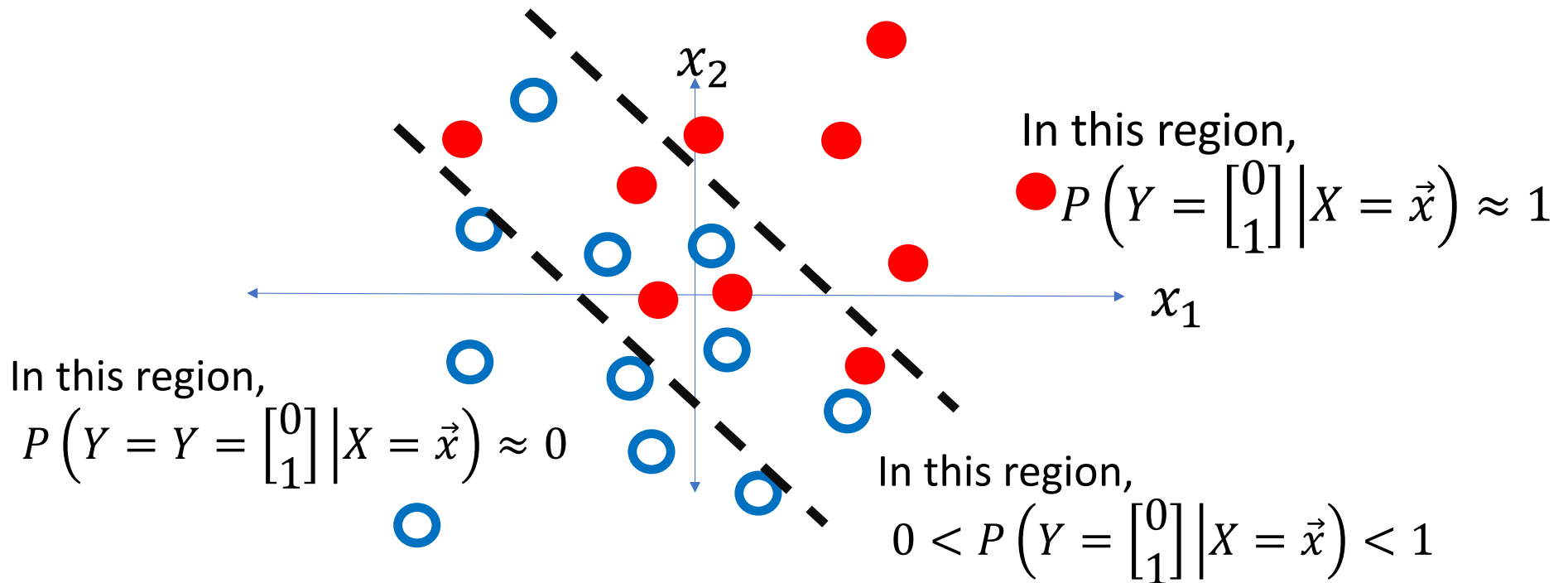


Outline

- Review: multi-class perceptron
- Breaking the constraints of linearity: multi-layer neural nets
- Flow diagram for the XOR problem
- Flow diagram for a multi-layer neural net
- One-hot vectors
- Softmax
- Cross-entropy = negative log probability of the training data
- Stochastic gradient descent
- Training a neural net using numpy
- The same example using pytorch
- torch.nn: standard layers and units

Softmax: Probabilistic boundaries

Instead of trying to find the exact boundaries, let's model the **probability** that token \vec{x} belongs to class \vec{y} .



Argmax versus Softmax

Remember that for the perceptron, we have

$$f(\vec{x}_i) = \begin{bmatrix} f_1(\vec{x}_i) \\ \vdots \\ f_V(\vec{x}_i) \end{bmatrix}, \quad f_c(\vec{x}_i) = \begin{cases} 1 & c = \operatorname{argmax} \vec{w}_c^T \vec{x} \\ 0 & \text{otherwise} \end{cases}$$

For softmax, we have

$$f(\vec{x}_i) = \begin{bmatrix} f_1(\vec{x}_i) \\ \vdots \\ f_V(\vec{x}_i) \end{bmatrix}, \quad f_c(\vec{x}_i) = \frac{e^{\vec{w}_c^T \vec{x}}}{\sum_{k=1}^V e^{\vec{w}_k^T \vec{x}}}$$

The softmax function

- This is called the softmax function:

$$\text{softmax}(\vec{x}_i) = \begin{bmatrix} \text{softmax}_1(W^T \vec{x}) \\ \vdots \\ \text{softmax}_V(W^T \vec{x}) \end{bmatrix}, \quad \text{softmax}_c(W^T \vec{x}) = \frac{e^{\vec{w}_c^T \vec{x}}}{\sum_{k=1}^V e^{\vec{w}_k^T \vec{x}}}$$

- ...where the matrix W is defined to be

$$W = [\vec{w}_1, \dots, \vec{w}_V]$$

Argmax and Softmax

$$f_c(\vec{x}_i) = \begin{cases} 1 & c = \operatorname{argmax} \vec{w}_c^T \vec{x} \\ 0 & \text{otherwise} \end{cases}, \quad f_c(\vec{x}_i) = \frac{e^{\vec{w}_c^T \vec{x}}}{\sum_{k=1}^V e^{\vec{w}_k^T \vec{x}}}$$

In both cases, we have:

- $f_c(\vec{x}_i) \geq 0$
- $f_c(\vec{x}_i) \leq 1$
- $\sum_{c=1}^V f_c(\vec{x}_i) = 1$

Argmax and Softmax

$$f_c(\vec{x}_i) = \begin{cases} 1 & c = \operatorname{argmax} \vec{w}_c^T \vec{x} \\ 0 & \text{otherwise} \end{cases}, \quad f_c(\vec{x}_i) = \frac{e^{\vec{w}_c^T \vec{x}}}{\sum_{k=1}^V e^{\vec{w}_k^T \vec{x}}}$$

In both cases, we can interpret these as probabilities:

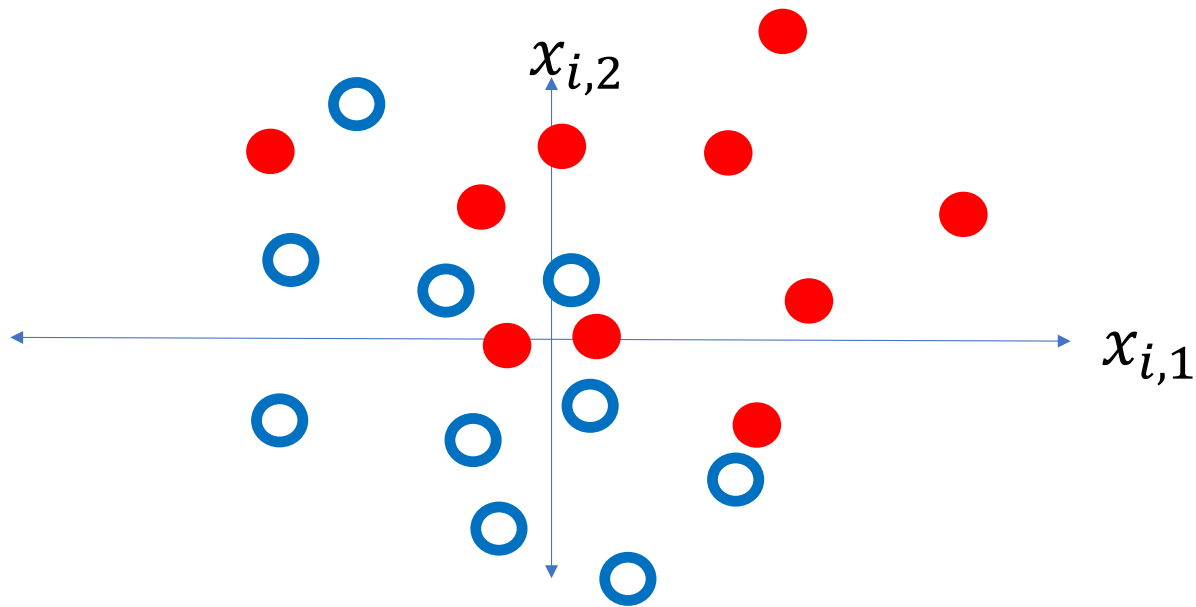
$$f_c(\vec{x}) = P(\text{Class} = c | X = \vec{x})$$

Outline

- Review: multi-class perceptron
- Breaking the constraints of linearity: multi-layer neural nets
- Flow diagram for the XOR problem
- Flow diagram for a multi-layer neural net
- One-hot vectors
- Softmax
- Cross-entropy = negative log probability of the training data
- Stochastic gradient descent
- Training a neural net using numpy
- The same example using pytorch
- torch.nn: standard layers and units

Classifier Learning using a Softmax Layer

- Suppose we have some data.
- We want to learn vectors $\vec{w}_c = [w_{c,1}, \dots, w_{c,D}, b_c]^T$ so that $P(\text{Class} = c | X = \vec{x}) = \text{softmax}_c(W^T \vec{x})$.

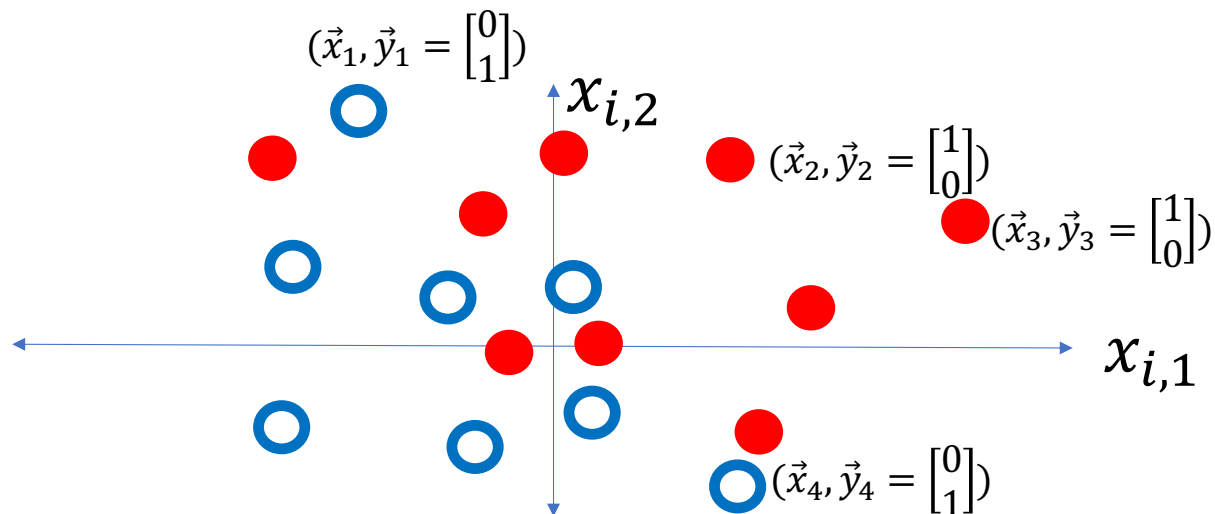


Learning a Softmax: Training data

Data:

$$\mathcal{D} = \{(\vec{x}_1, c_1), (\vec{x}_2, c_2), \dots, (\vec{x}_n, c_n)\}$$

where each $\vec{x}_i = [x_{i,1}, \dots, x_{i,D}, 1]^T$ is a vector, and each $c_i \in \{1, \dots, V\}$ is a integer encoding the true class label.



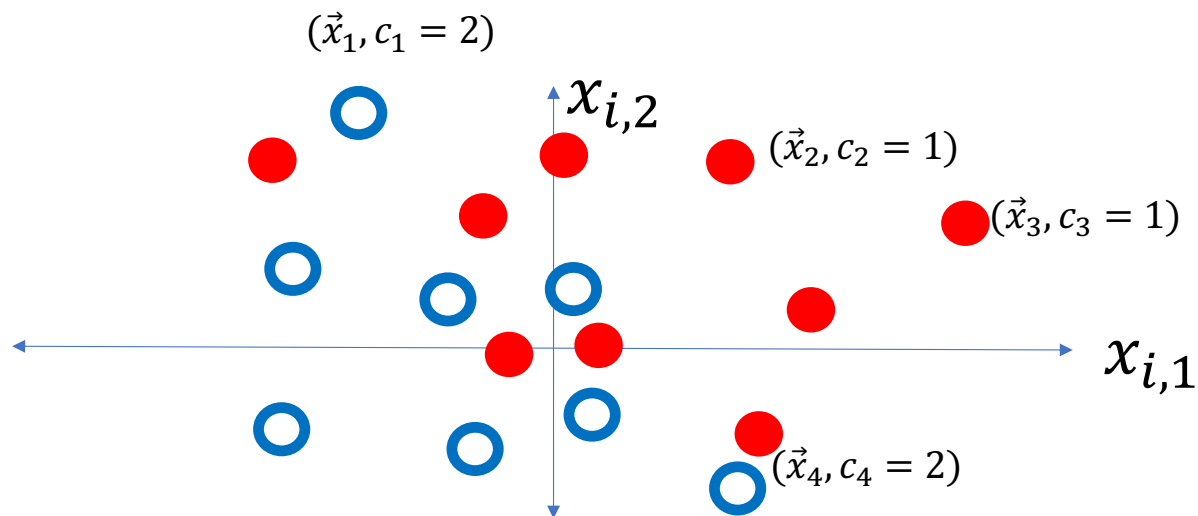
Learning a Softmax: Model parameters

We want to learn the model parameters

$$W = [\vec{w}_1, \dots, \vec{w}_V]$$

so that

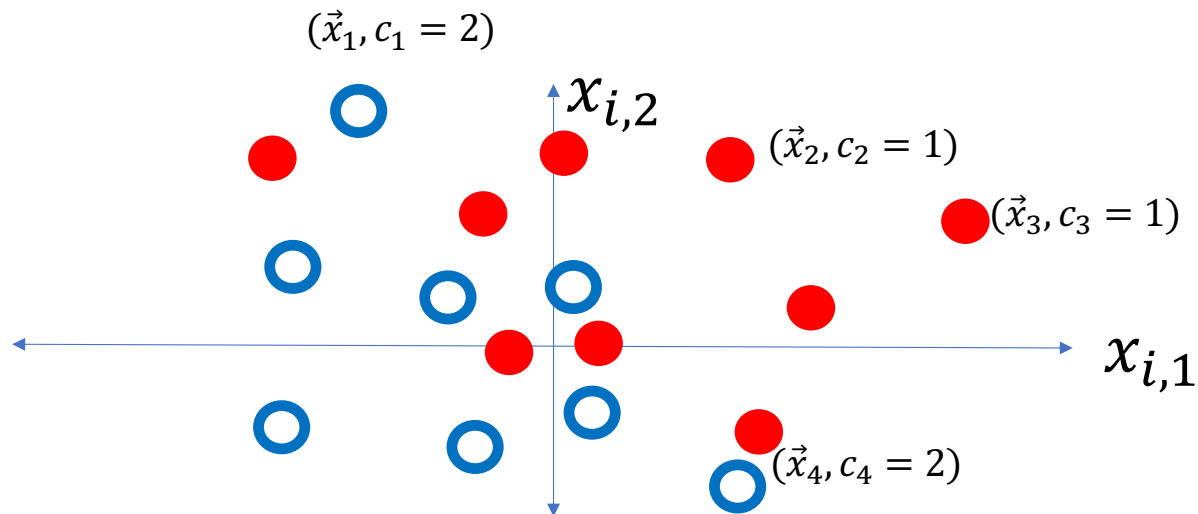
$$P(C = c_i | X = \vec{x}_i) = \underset{c_i}{\text{softmax}}(W^T \vec{x}_i)$$



Learning a Softmax: Training criterion

We want to learn the model parameters, $W = [\vec{w}_1, \dots, \vec{w}_V]$, in order to maximize the probability of the observed data:

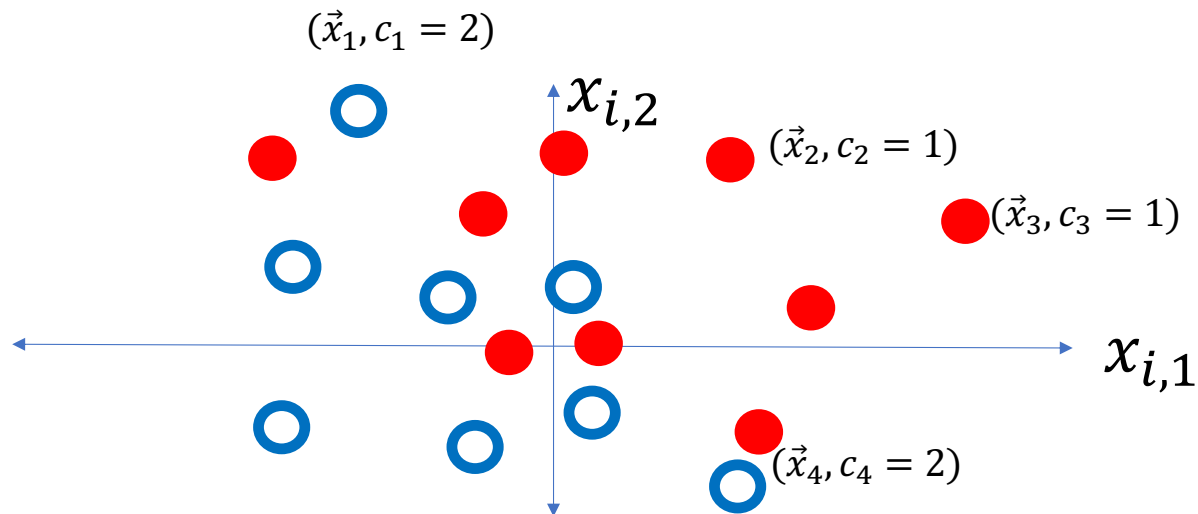
$$P(\mathcal{D}|W) = \prod_{i=1}^n P(C = c_i | X = \vec{x}_i)$$



Learning a Softmax: Training Criterion

We want to learn the model parameters, $W = [\vec{w}_1, \dots, \vec{w}_V]$, in order to maximize the probability of the observed data:

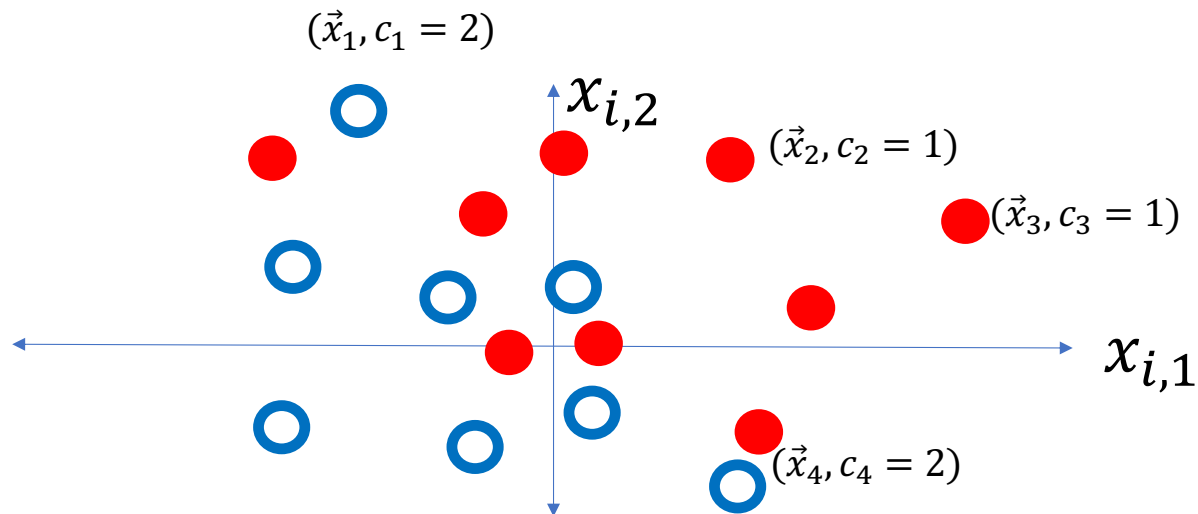
$$P(\mathcal{D}|W) = \prod_{i=1}^n \text{softmax}_{c_i}(W^T \vec{x}_i)$$



Learning a Softmax: Training Criterion

We want to learn the model parameters, $W = [\vec{w}_1, \dots, \vec{w}_V]$, in order to maximize the probability of the observed data:

$$P(\mathcal{D}|W) = \prod_{i=1}^n \frac{e^{\vec{w}_{c_i}^T \vec{x}_i}}{\sum_{k=1}^V e^{\vec{w}_k^T \vec{x}_i}}$$



How do you maximize a function?

Our goal is to find $W = [\vec{w}_1, \dots, \vec{w}_V]$ in order to maximize

$$P(\mathcal{D}|W) = \prod_{i=1}^n \frac{e^{\vec{w}_{c_i}^T \vec{x}_i}}{\sum_{k=1}^V e^{\vec{w}_k^T \vec{x}_i}}$$

Here are some useful things to know:

1. Logarithm turns products into sums
2. Maximizing $f(W)$ is the same thing as minimizing $-f(W)$

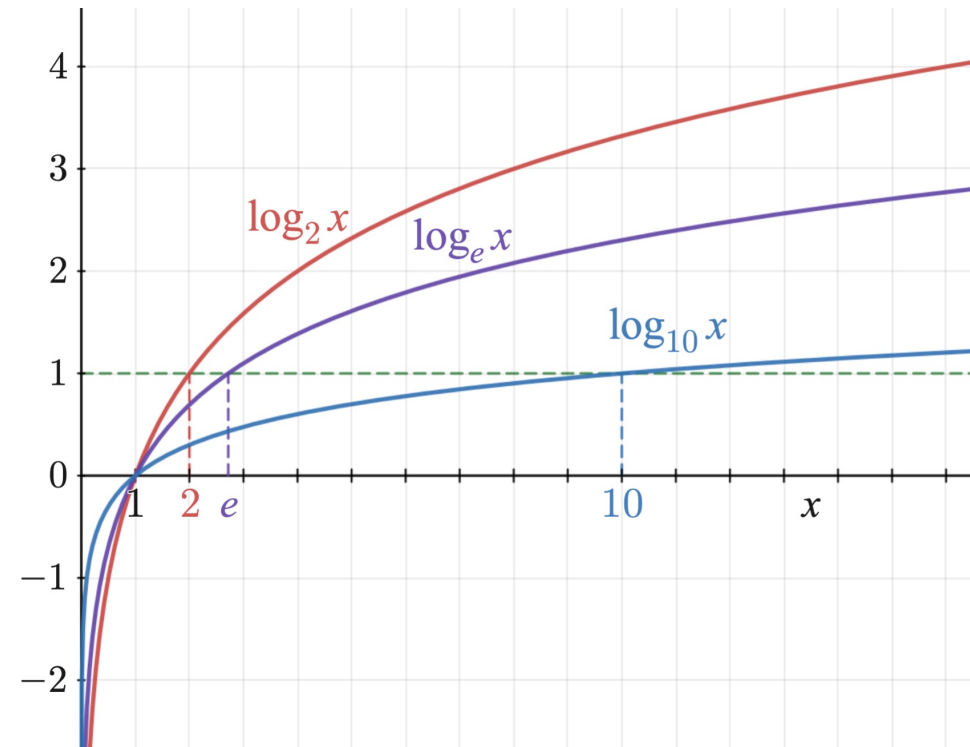
1. Logarithms turn products into sums

$\ln x$ (the natural logarithm of x , shown as $\log_e x$ in the plot at right) is a monotonically increasing function of x .

Since it's monotonically increasing,

$$\operatorname{argmax}_W P(\mathcal{D}|W) = \operatorname{argmax}_W \ln P(\mathcal{D}|W)$$

Almost always, maximizing the log probability is easier than maximizing the probability, because logarithms turn products into sums.



Logarithm_plots.png, CC-SA 3.0, Richard F. Lyon, 2011

1. Logarithms turn products into sums

Our goal is to find $W = [\vec{w}_1, \dots, \vec{w}_V]$ in order to maximize

$$\begin{aligned}\ln P(\mathcal{D}|W) &= \ln \prod_{i=1}^n \frac{e^{\vec{w}_{c_i}^T \vec{x}_i}}{\sum_{k=1}^V e^{\vec{w}_k^T \vec{x}_i}} \\ &= \sum_{i=1}^n \ln \frac{e^{\vec{w}_{c_i}^T \vec{x}_i}}{\sum_{k=1}^V e^{\vec{w}_k^T \vec{x}_i}} \\ &= \sum_{i=1}^n \left(\vec{w}_{c_i}^T \vec{x}_i - \ln \sum_{k=1}^V e^{\vec{w}_k^T \vec{x}_i} \right)\end{aligned}$$

2. Maximizing $f(W)$ is the same thing as minimizing $-f(W)$.

Our goal is to find $W = [\vec{w}_1, \dots, \vec{w}_V]$ in order to maximize

$$\ln P(\mathcal{D}|W) = \sum_{i=1}^n \left(\vec{w}_{c_i}^T \vec{x}_i - \ln \sum_{k=1}^V e^{\vec{w}_k^T \vec{x}_i} \right)$$

Choosing W to maximize $\vec{w}_{c_i}^T \vec{x}_i$ is kind of obvious: just set $\vec{w}_{c_i} = A\vec{x}_i$, where A is a scalar that's as big as possible. Maximizing $-\ln \sum_{k=1}^V e^{\vec{w}_k^T \vec{x}_i}$, is not obvious.

2. Maximizing $f(W)$ is the same thing as minimizing $-f(W)$.

To emphasize the hard part of the problem, there is a convention that, instead of maximizing $\ln P(\mathcal{D}|W)$, we minimize $-\ln P(\mathcal{D}|W)$:

Our goal is to find $W = [\vec{w}_1, \dots, \vec{w}_V]$ in order to minimize

$$\mathcal{L} = -\ln P(\mathcal{D}|W) = \sum_{i=1}^n \left(\ln \sum_{k=1}^V e^{\vec{w}_k^T \vec{x}_i} - \vec{w}_{c_i}^T \vec{x}_i \right)$$

The curly \mathcal{L} is a symbol we use to denote a “loss function”. A loss function is something you want to minimize.

Some details: Cross entropy

- The loss function is called “cross entropy,” because it is similar in some ways to the entropy of a thermodynamic system in physics.
- When you implement this in software, it’s a good idea to normalize by the number of training tokens, so that the scale is easier to understand:

$$\mathcal{L} = -\frac{1}{n} \log P(\mathcal{D}|W) = -\frac{1}{n} \sum_{i=1}^n \log P(C = c_i | X = \vec{x}_i)$$

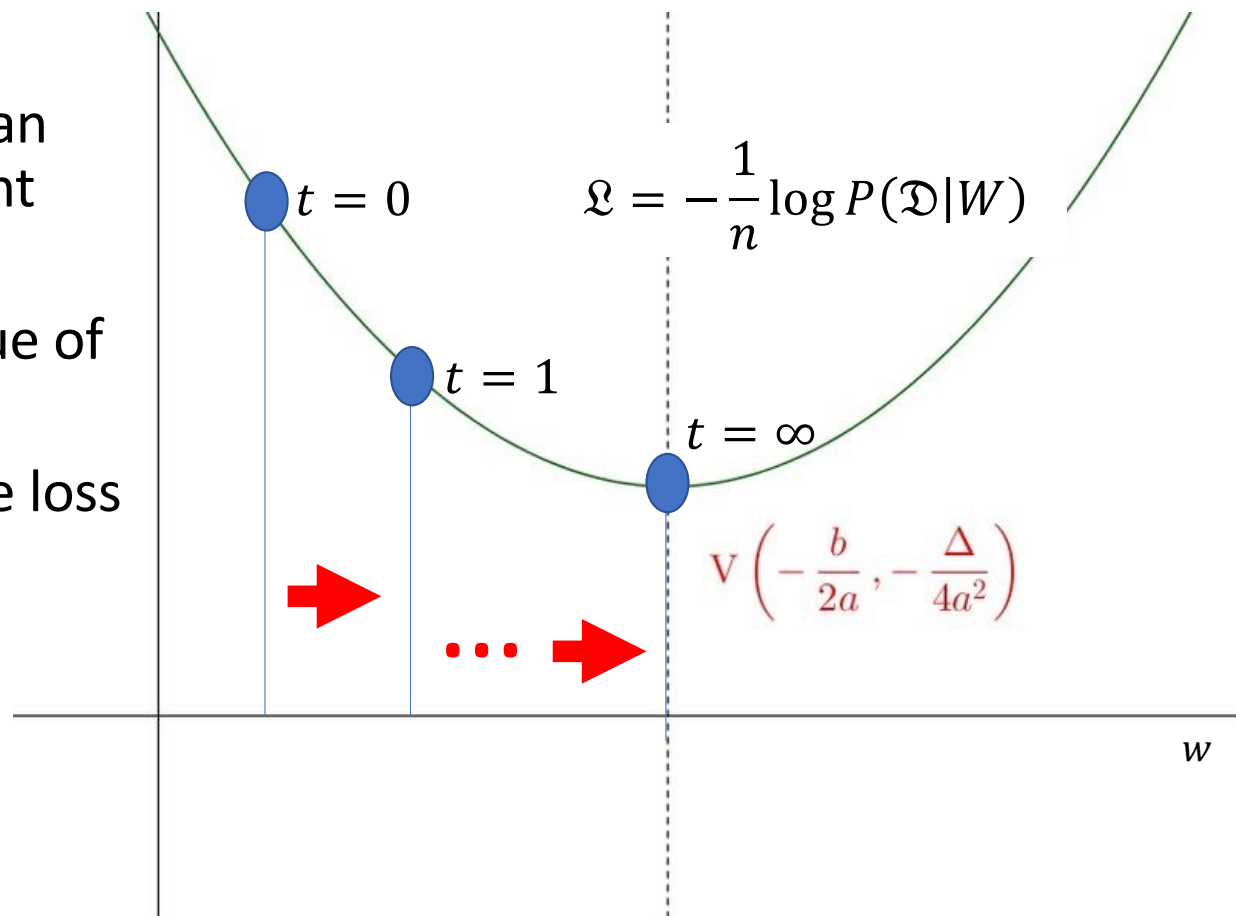
Outline

- Review: multi-class perceptron
- Breaking the constraints of linearity: multi-layer neural nets
- Flow diagram for the XOR problem
- Flow diagram for a multi-layer neural net
- One-hot vectors
- Softmax
- Cross-entropy = negative log probability of the training data
- **Stochastic gradient descent**
- **Training a neural net using numpy**
- **The same example using pytorch**
- **torch.nn: standard layers and units**

The iterative solution to training a neural net

Instead of minimizing the loss in closed form, we're going to use an iterative algorithm called gradient descent. It works like this:

- Start from a random initial value of \vec{w} (at $t = 0$).
- Adjust \vec{w} in order to reduce the loss ($t = 1$).
- Repeat until you reach the optimum ($t = \infty$).



The gradient descent algorithm

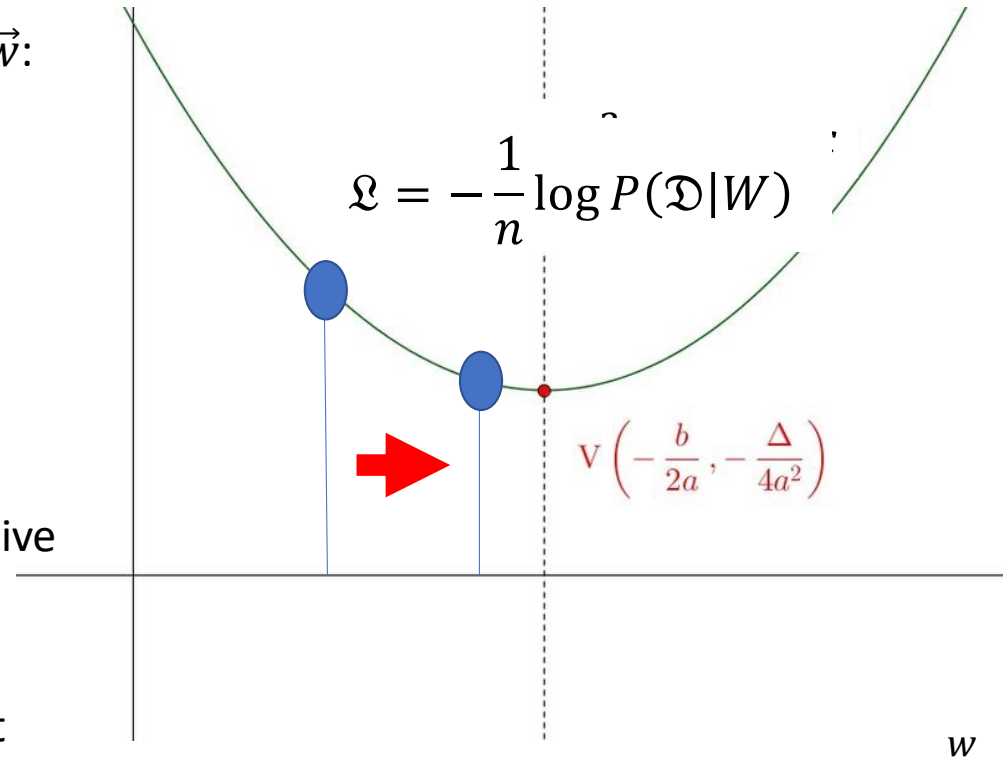
- Start from a random initial value of \vec{w} .
- Calculate the derivative of the loss with respect to \vec{w} :

$$\nabla_{\vec{w}} \mathcal{L} = \begin{bmatrix} \frac{\partial \mathcal{L}}{\partial w_1} \\ \vdots \\ \frac{\partial \mathcal{L}}{\partial w_D} \\ \frac{\partial \mathcal{L}}{\partial b} \end{bmatrix}$$

- Take a step “downhill” (in the direction of the negative gradient)

$$\vec{w} \leftarrow \vec{w} - \frac{\eta}{2} \nabla_{\vec{w}} \mathcal{L}$$

...where η is a constant called the “learning rate,” that determines how big of a step you take. Usually, you need to adjust η in order to get optimum performance on a dev set, but often $\eta \approx 0.001$.



Stochastic gradient descent

- If n is large, computing or differentiating the loss for the entire training dataset, all at once, can be expensive.
- The stochastic gradient descent algorithm picks one training token (\vec{x}_i, y_i) at random ("stochastically"), and adjusts \vec{w} in order to reduce the error a little bit for that one token:

$$\vec{w} \leftarrow \vec{w} - \frac{\eta}{2} \nabla_{\vec{w}} \mathcal{L}_i$$

...where

$$\mathcal{L}_i = -\ln P(C = c_i | X = \vec{x}_i, W)$$

... in other words, the part of the loss function that only depends on the i^{th} token.

Stochastic gradient descent

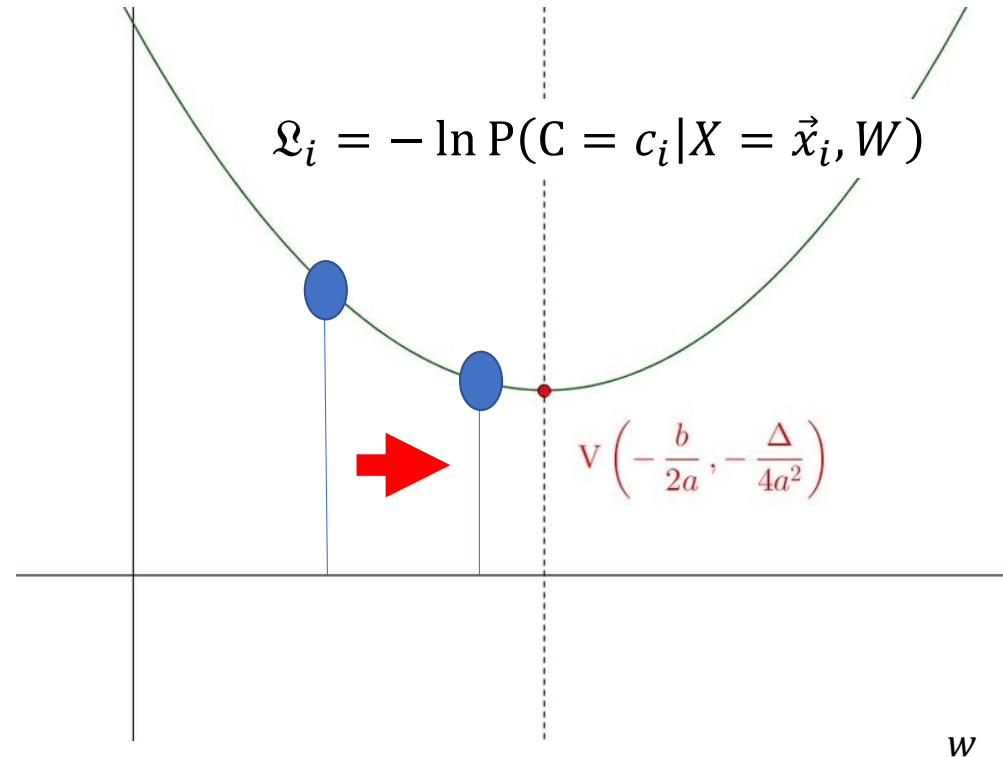
$$\mathcal{L}_i = \ln \sum_{k=1}^V e^{\vec{w}_k^T \vec{x}_i} - \vec{w}_{c_i}^T \vec{x}_i$$

If we differentiate that, we discover that:

$$\nabla_{\vec{w}_c} \mathcal{L}_i = (f_c(\vec{x}) - y_{i,c}) \vec{x}_i$$

So the stochastic gradient descent algorithm is:

$$\vec{w}_c \leftarrow \vec{w}_c - \eta (f_c(\vec{x}) - y_{i,c}) \vec{x}_i$$



Outline

- Review: multi-class perceptron
- Breaking the constraints of linearity: multi-layer neural nets
- Flow diagram for the XOR problem
- Flow diagram for a multi-layer neural net
- One-hot vectors
- Softmax
- Cross-entropy = negative log probability of the training data
- Stochastic gradient descent
- Training a neural net using numpy
- The same example using pytorch
- torch.nn: standard layers and units

Training a neural net using numpy or pytorch

Running example: linear regression

- For example, suppose $y = \sin(x)$
- Suppose that the network can only model functions of the form

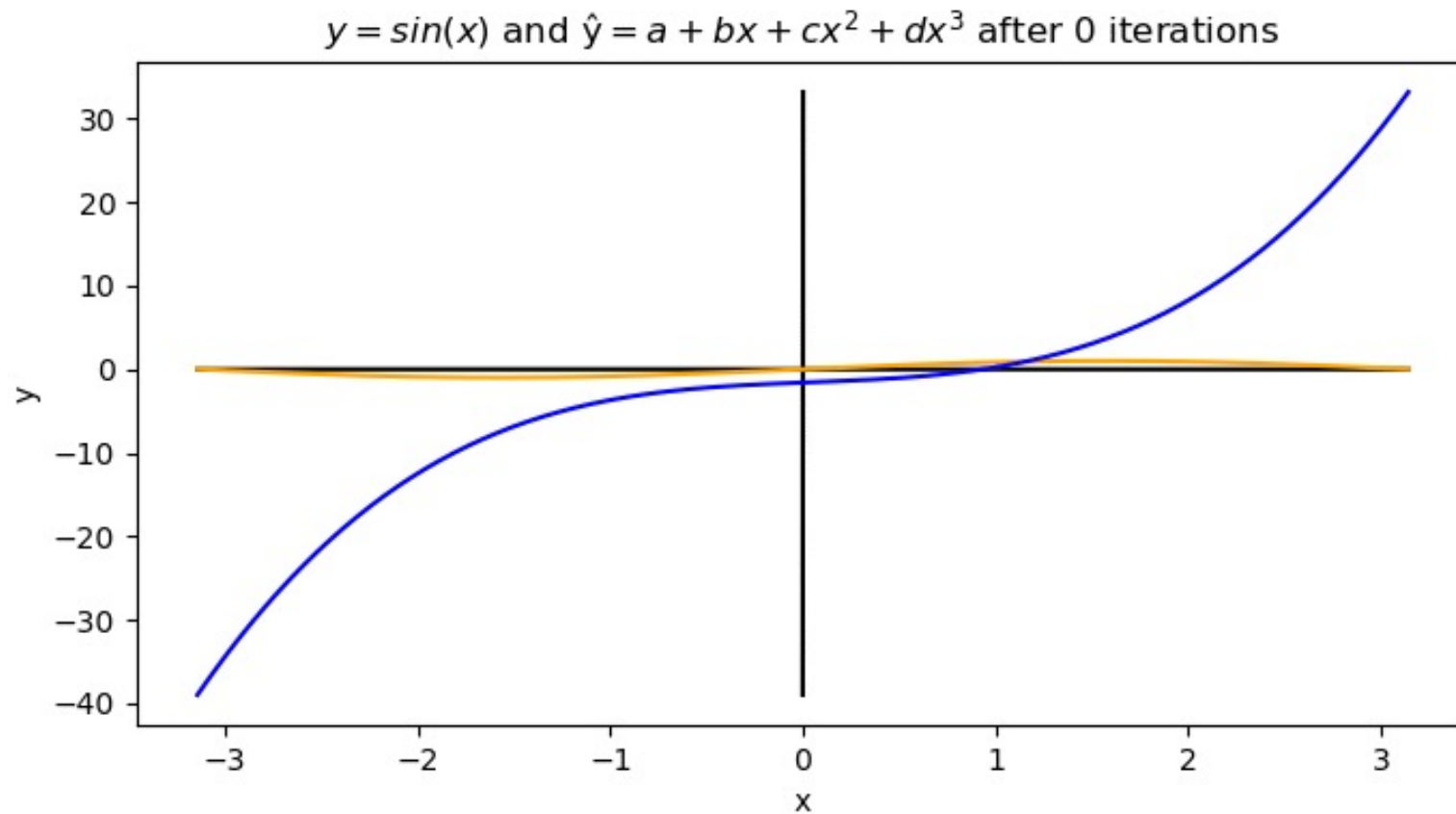
$$f(x) = a + bx + cx^2 + dx^3 = \vec{x}^T \vec{w}$$

...where we're defining...

$$\vec{w} = \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix}, \vec{x} = \begin{bmatrix} 1 \\ x \\ x^2 \\ x^3 \end{bmatrix}$$

- We want to learn a, b, c, d so that $f(x) \approx y$

Running example: neural net regression



Mean-squared error

First, let's define the loss function.

$$f(x_i) = a + bx_i + cx_i^2 + dx_i^3,$$

$$\epsilon_i = f(x_i) - y_i,$$

$$\mathcal{L} = \frac{1}{n} \sum_{i=1}^n \epsilon_i^2$$

Loss gradient

Next, find the derivative of the loss.

$$f(x_i) = a + bx_i + cx_i^2 + dx_i^3, \quad \epsilon_i = f(x_i) - y_i, \quad \mathcal{L} = \frac{1}{n} \sum_{i=1}^n \epsilon_i^2$$

$$\nabla_{\vec{w}} \mathcal{L} = \begin{bmatrix} \frac{d\mathcal{L}}{da} \\ \frac{d\mathcal{L}}{db} \\ \frac{d\mathcal{L}}{dc} \\ \frac{d\mathcal{L}}{dd} \end{bmatrix} = \frac{2}{n} \sum_{i=1}^n \epsilon_i \nabla_{\vec{w}} \epsilon_i = \frac{2}{n} \sum_{i=1}^n \epsilon_i \begin{bmatrix} \frac{df(x_i)}{da} \\ \frac{df(x_i)}{db} \\ \frac{df(x_i)}{dc} \\ \frac{df(x_i)}{dd} \end{bmatrix} = \frac{2}{n} \sum_{i=1}^n \epsilon_i \begin{bmatrix} 1 \\ x_i \\ x_i^2 \\ x_i^3 \end{bmatrix}$$

Gradient update

Now, update the weights by subtracting the gradient.

$$a = a - \frac{\eta}{2} \frac{d\mathcal{L}}{da} = a - \frac{\eta}{n} \sum_{i=1}^n (f(x_i) - y_i),$$

$$b = b - \eta \frac{d\mathcal{L}}{db} = b - \frac{\eta}{n} \sum_{i=1}^n (f(x_i) - y_i)x_i,$$

$$c = c - \eta \frac{d\mathcal{L}}{dc} = c - \frac{\eta}{n} \sum_{i=1}^n (f(x_i) - y_i)x_i^2,$$

$$d = d - \eta \frac{d\mathcal{L}}{dd} = d - \frac{\eta}{n} \sum_{i=1}^n (f(x_i) - y_i)x_i^3$$

How a neural network is trained

Here's Justin Johnson's code for doing those things:

(https://pytorch.org/tutorials/beginner/pytorch_with_examples.html)

```
for t in range(2000):
    # Forward pass: compute predicted y
    #  $y = a + b x + c x^2 + d x^3$ 
    y_pred = a + b * x + c * x ** 2 + d * x ** 3

    # Compute and print loss
    loss = np.square(y_pred - y).sum()
    if t % 100 == 99:
        print(t, loss)

    # Backprop to compute gradients of a, b, c, d with respect to loss
    grad_y_pred = 2.0 * (y_pred - y)
    grad_a = grad_y_pred.sum()
    grad_b = (grad_y_pred * x).sum()
    grad_c = (grad_y_pred * x ** 2).sum()
    grad_d = (grad_y_pred * x ** 3).sum()

    # Update weights
    a -= learning_rate * grad_a
    b -= learning_rate * grad_b
    c -= learning_rate * grad_c
    d -= learning_rate * grad_d
```

© 2021 Pytorch,

https://pytorch.org/tutorials/beginner/pytorch_with_examples.html

Outline

- Review: multi-class perceptron
- Breaking the constraints of linearity: multi-layer neural nets
- Flow diagram for the XOR problem
- Flow diagram for a multi-layer neural net
- One-hot vectors
- Softmax
- Cross-entropy = negative log probability of the training data
- Stochastic gradient descent
- Training a neural net using numpy
- The same example using pytorch
- torch.nn: standard layers and units

Autograd: Main idea

- A neural network is a complicated function $f(x)$, made up of many simple components
- If we try to take all the derivatives, $d\mathcal{L}/dw_{j,k}^{(l)}$, all at once, in a big mass of spaghetti code, then the code will be really ugly.
- HOWEVER: Each of the components is simple to compute. Furthermore, the derivative of its output w.r.t. its input is simple.

Autograd: Tensor objects

The basic idea of autograd is to create a new kind of object that takes responsibility for its own gradient.

- For example, the object might be a network weight, $w_{j,k}^{(l)}$

Autograd: Tensor objects

- In pytorch, variables that take responsibility for their own gradients are called “tensors” (<https://pytorch.org/docs/stable/tensors.html>)
- Here’s how Justin Johnson defines tensors for the polynomial regression problem:

```
# Create random Tensors for weights. For a third order polynomial, we need  
# 4 weights:  $y = a + b x + c x^2 + d x^3$   
# Setting requires_grad=True indicates that we want to compute gradients  
with  
# respect to these Tensors during the backward pass.  
a = torch.randn(), device=device, dtype=dtype, requires_grad=True)  
b = torch.randn(), device=device, dtype=dtype, requires_grad=True)  
c = torch.randn(), device=device, dtype=dtype, requires_grad=True)  
d = torch.randn(), device=device, dtype=dtype, requires_grad=True)
```

Autograd: Overloaded operators

The basic idea of autograd is to create a new kind of object that takes responsibility for its own gradient.

- For example, the object might be a network weight, $w_{j,k}^{(l)}$
- These new objects have overloaded operators, so that any time we use them to compute some output, the input is cached. For example, it might be used to compute

$$\xi_j^{(l)} = b_j^{(l)} + \sum_k w_{j,k}^{(l)} h_k^{(l-1)}$$

Autograd: Overloaded operators

Here's how it gets used:

```
for t in range(2000):  
    # Forward pass: compute predicted y using operations on Tensors.  
    y_pred = a + b * x + c * x ** 2 + d * x ** 3
```

Python overloaded operators: if “b” is an object that has a method named `__mul__`, then the expression “f=b*x” actually calls “f=b.__mul__(x)”.

Autograd: Overloaded operators

The operator overload code looks something like this:

```
class Tensor(torch.autograd.Function):
```

```
    def __init__(self, weight):
```

```
        self.weight = weight
```

```
        self.saved_tensors = ()
```

```
    def __mul__(self, other):
```

```
        self.saved_tensors = (self.saved_tensors[:], other)
```



```
        returnvalue = self.weight * other
```

```
        return Tensor(returnvalue)
```

Cache x in self.saved_tensors, so we can use it later...



Then calculate the output of the multiply operation, ... and cast the return value as a Tensor.



Autograd: Overloaded operators

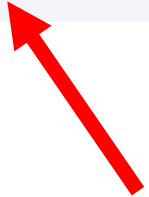
Here's how it gets used:

```
for t in range(2000):  
    # Forward pass: compute predicted y using operations on Tensors.  
    y_pred = a + b * x + c * x ** 2 + d * x ** 3
```

Stores x in b.saved_tensors



Stores x**2 in c.saved_tensors



Python overloaded operators: the expression “b*x” actually calls b.__mul__(x).

Autograd: the Loss tensor

The basic idea of autograd is to create a new kind of object, a tensor, that takes responsibility for its own gradient. Any time we use tensors to compute some output, the input is cached. For example, these operations:

$$f(x_i) = a + bx_i + cx_i^2 + dx_i^3$$

$$\mathcal{L} = \frac{1}{n} \sum_{i=1}^n (f(x_i) - y_i)^2$$

$$f = a + b*x + c*x**2 + d*x**3$$

$$\text{loss} = (f-y).\text{pow}(2)$$

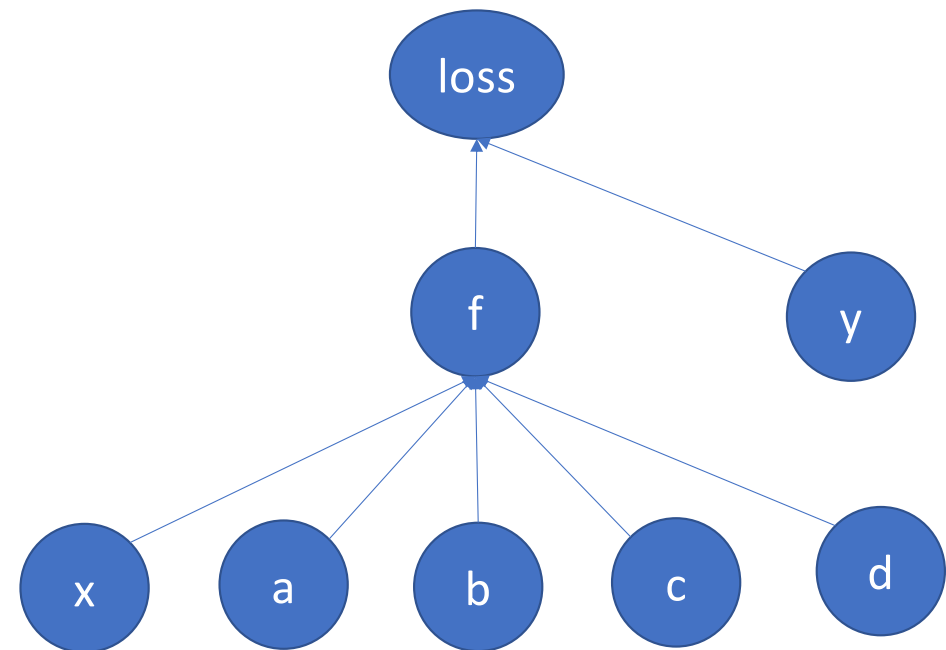
...will calculate the loss, but will also store some extra information in `loss.saved_tensors`, `f.saved_tensors`, `a.saved_tensors`, `b.saved_tensors`, `c.saved_tensors`, `d.saved_tensors`, and `x.saved_tensors`.

Autograd: the Loss tensor

Notice the flow diagram that was implied by those lines of code.

Each tensor's overloaded `__mul__` operator keeps track of the variables used to compute it:

- `loss.saved_tensors` has pointers to `f` and `y`
- `f.saved_tensors` has pointers to `x`, `a`, `b`, `c`, and `d`



Autograd

loss depends on
y_pred, which
depends on a, b,
c, d.

```
for t in range(2000):
    # Forward pass: compute predicted y using operations on Tensors.
    y_pred = a + b * x + c * x ** 2 + d * x ** 3

    # Compute and print loss using operations on Tensors.
    # Now loss is a Tensor of shape (1,)
    # loss.item() gets the scalar value held in the loss.
    loss = (y_pred - y).pow(2).sum()
    if t % 100 == 99:
        print(t, loss.item())

    # Use autograd to compute the backward pass. This call will compute the
    # gradient of loss with respect to all Tensors with requires_grad=True.
    # After this call a.grad, b.grad, c.grad and d.grad will be Tensors holding
    # the gradient of the loss with respect to a, b, c, d respectively.
    loss.backward()

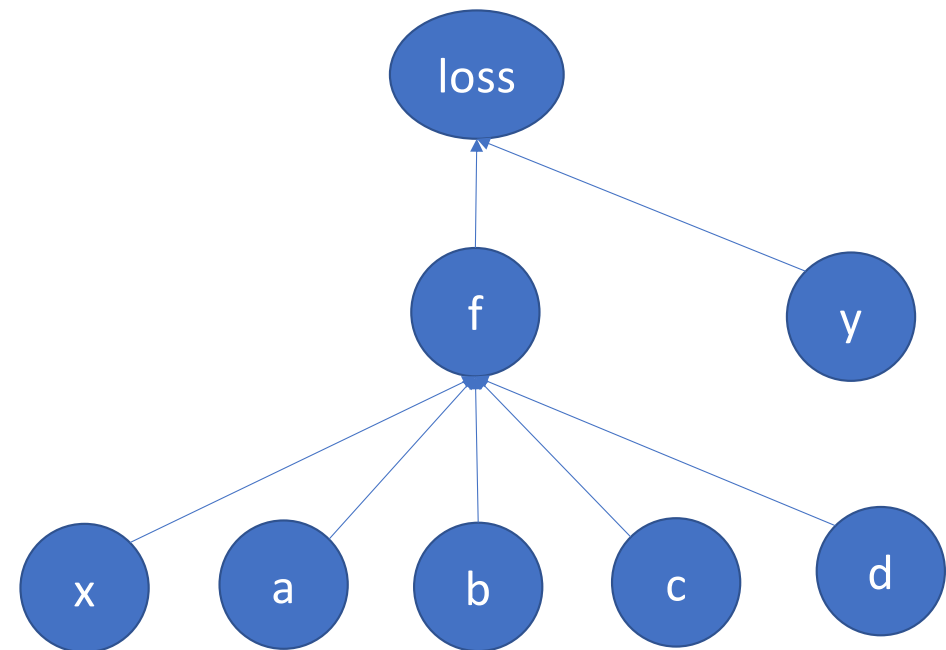
    # Manually update weights using gradient descent. Wrap in torch.no_grad()
    # because weights have requires_grad=True, but we don't need to track this
    # in autograd.
    with torch.no_grad():
        a -= learning_rate * a.grad
        b -= learning_rate * b.grad
        c -= learning_rate * c.grad
        d -= learning_rate * d.grad
```

Autograd: the `backward` function

Every tensor object has a method called `backward()`.

If `backward()` is called with no arguments, it calculates the derivative with respect to the inputs:

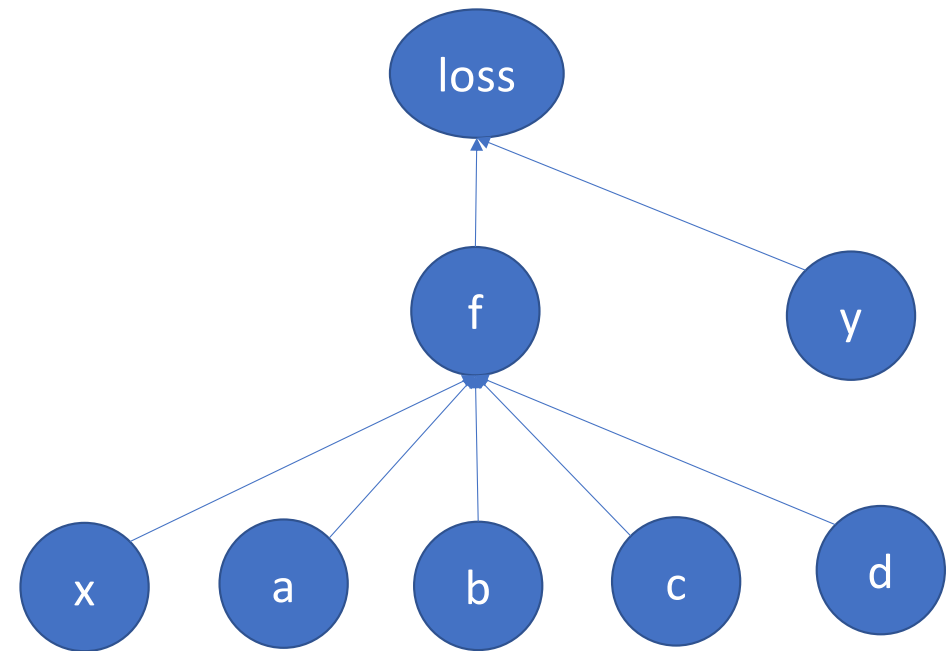
- `loss.backward()` calculates $\text{tmp} = \frac{d\mathcal{L}}{df}$, then calls the method `f.backward(tmp)`.



Autograd: the backward function

If backward() is called with the argument $\text{tmp} = \frac{d\mathcal{L}}{df}$, it does three things:

- Store $\text{f.grad} = \frac{d\mathcal{L}}{df}$
- Calculate derivative w.r.t. each input, for example, $\frac{d\mathcal{L}}{dc} = \frac{d\mathcal{L}}{df} \times \frac{df}{dc}$
- Pass the input derivatives back to the inputs, e.g., call $\text{c.backward}(\text{tmp})$



Autograd

loss depends on `y_pred`, which depends on `a`, `b`, `c`, `d`.

Calculates the derivative of the loss w.r.t. each of its input tensors.

Uses the resulting derivatives to update the weights.

```
for t in range(2000):
    # Forward pass: compute predicted y using operations on Tensors.
    y_pred = a + b * x + c * x ** 2 + d * x ** 3

    # Compute and print loss using operations on Tensors.
    # Now loss is a Tensor of shape (1,)
    # loss.item() gets the scalar value held in the loss.
    loss = (y_pred - y).pow(2).sum()
    if t % 100 == 99:
        print(t, loss.item())

    # Use autograd to compute the backward pass. This call will compute the
    # gradient of loss with respect to all Tensors with requires_grad=True.
    # After this call a.grad, b.grad, c.grad and d.grad will be Tensors holding
    # the gradient of the loss with respect to a, b, c, d respectively.
    loss.backward()

    # Manually update weights using gradient descent. Wrap in torch.no_grad()
    # because weights have requires_grad=True, but we don't need to track this
    # in autograd.
    with torch.no_grad():
        a -= learning_rate * a.grad
        b -= learning_rate * b.grad
        c -= learning_rate * c.grad
        d -= learning_rate * d.grad
```


Details: How to turn off autograd

- As you know, every time you add, subtract, multiply or divide a tensor by anything, the tensor stores data in `self.saved_tensors`, so it can use that information later to compute the gradient
- How do you turn this behavior off?

Dynamically turning off Autograd

These weight
updates are not
part of the neural
network forward
pass.

```
for t in range(2000):
    # Forward pass: compute predicted y using operations on Tensors.
    y_pred = a + b * x + c * x ** 2 + d * x ** 3

    # Compute and print loss using operations on Tensors.
    # Now loss is a Tensor of shape (1,)
    # loss.item() gets the scalar value held in the loss.
    loss = (y_pred - y).pow(2).sum()
    if t % 100 == 99:
        print(t, loss.item())

    # Use autograd to compute the backward pass. This call will compute the
    # gradient of loss with respect to all Tensors with requires_grad=True.
    # After this call a.grad, b.grad, c.grad and d.grad will be Tensors holding
    # the gradient of the loss with respect to a, b, c, d respectively.
    loss.backward()

    # Manually update weights using gradient descent. Wrap in torch.no_grad()
    # because weights have requires_grad=True, but we don't need to track this
    # in autograd.
    with torch.no_grad():
        a -= learning_rate * a.grad
        b -= learning_rate * b.grad
        c -= learning_rate * c.grad
        d -= learning_rate * d.grad
```

How to zero out the gradients

- When you call `backward()` over a tensor, it doesn't zero out any previous gradients
- Instead, it adds the current gradient to the previous gradients
- A very very very common mistake: running 2000 iterations, with the gradient accumulating from each iteration to the next, instead of zeroing it out in between iterations

Manually zeroing out the gradients

Here's the part I didn't show you before.

```
learning_rate = 1e-6
for t in range(2000):
    # Forward pass: compute predicted y using operations on Tensors.
    y_pred = a + b * x + c * x ** 2 + d * x ** 3

    # Compute and print loss using operations on Tensors.
    # Now loss is a Tensor of shape (1,)
    # loss.item() gets the scalar value held in the loss.
    loss = (y_pred - y).pow(2).sum()
    if t % 100 == 99:
        print(t, loss.item())

    # Use autograd to compute the backward pass. This call will compute the
    # gradient of loss with respect to all Tensors with requires_grad=True.
    # After this call a.grad, b.grad, c.grad and d.grad will be Tensors holding
    # the gradient of the loss with respect to a, b, c, d respectively.
    loss.backward()

    # Manually update weights using gradient descent. Wrap in torch.no_grad()
    # because weights have requires_grad=True, but we don't need to track this
    # in autograd.
    with torch.no_grad():
        a -= learning_rate * a.grad
        b -= learning_rate * b.grad
        c -= learning_rate * c.grad
        d -= learning_rate * d.grad

    # Manually zero the gradients after updating weights
    a.grad = None
    b.grad = None
    c.grad = None
    d.grad = None
```

Outline

- Review: multi-class perceptron
- Breaking the constraints of linearity: multi-layer neural nets
- Flow diagram for the XOR problem
- Flow diagram for a multi-layer neural net
- One-hot vectors
- Softmax
- Cross-entropy = negative log probability of the training data
- Stochastic gradient descent
- Training a neural net using numpy
- The same example using pytorch
- **torch.nn: standard layers and units**

Pytorch nn module

- The autograd feature of pytorch allows you to define only the forward propagation of your neural net. As long as all of the component operations are in pytorch's library, the back-propagation will be computed for you.
- Tensors just do multiplication and addition. What about other types of operations?
- General operations are contained in the nn module, using the formalism of a "layer."

Some types of layers

- torch.nn.Linear: a layer that computes $\vec{o} = W\vec{i} + \vec{b}$
- torch.nn.Softmax: a layer that computes $o_j = \frac{\exp(i_j)}{\sum_k \exp(i_k)}$
- torch.nn.Sigmoid: a layer that computes $o_j = \frac{1}{1 + \exp(-i_j)}$
- torch.nn.ReLU: a layer that computes $o_j = \max(0, i_j)$
- torch.nn.Sequential: a model that takes a sequence of layers as its arguments, and applies them, one after the other, in order

`m=torch.nn.Linear(n_1,n_2)`

- This creates a callable object, `m`, such that `o=m(i)` treats each row of `i` as a vector, and generates a corresponding row of `o` using the operation:

$$\vec{o} = W\vec{i} + \vec{b}$$

- `i` can be a tensor of any size, as long as its last dimension (the dimension of each row) is `n_1`
- `o` is then a tensor of the same shape as `i`, except that its last dimension (the row length) is now `n_2`
- `m.weight (W)` is a tensor of size `(n_2,n_1)`
- `m.bias (b)` is a row vector of length `n_2`

Example: Linear, Sigmoid, Softmax

- Here's an example flowgraph. We could create the layers as:

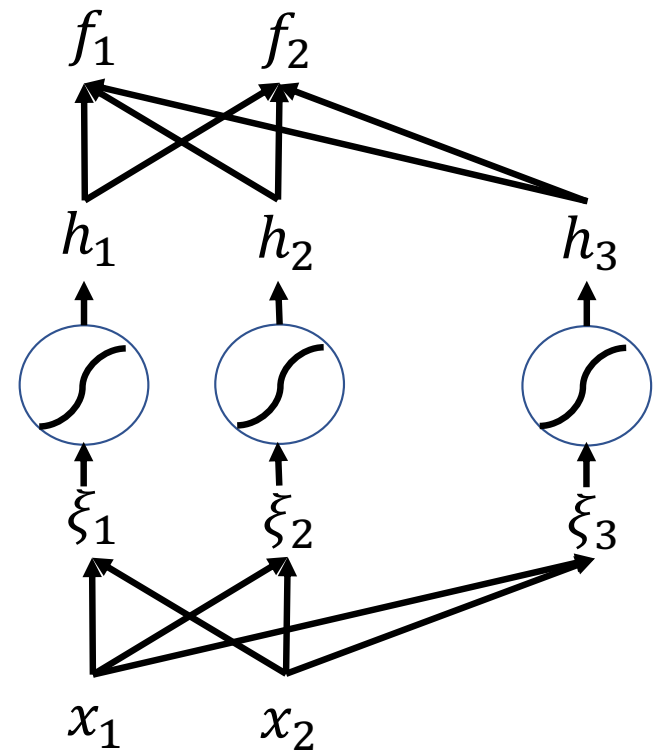
```
linear1 = torch.nn.Linear(2,3)
sigmoid1 = torch.nn.Sigmoid()
linear2 = torch.nn.Linear(3,2)
loss_function = torch.nn.MSELoss()
```

- Having created them, we could then run forward pass as:

```
xi = linear1(x)
h = sigmoid1(xi)
f = linear2(h)
loss = loss_function(f,y)
```

- Then we could calculate all of the gradients by running

```
loss.backward()
```



torch.nn.Sequential

- torch.nn.Sequential is a special module that creates a sequence of layers, where each layer's output is the next layer's input. For example:

```
model = torch.nn.Sequential(  
    torch.nn.Linear(2,3),  
    torch.nn.Sigmoid(),  
    torch.nn.Linear(3,2))  
loss_function = torch.nn.MSELoss()
```

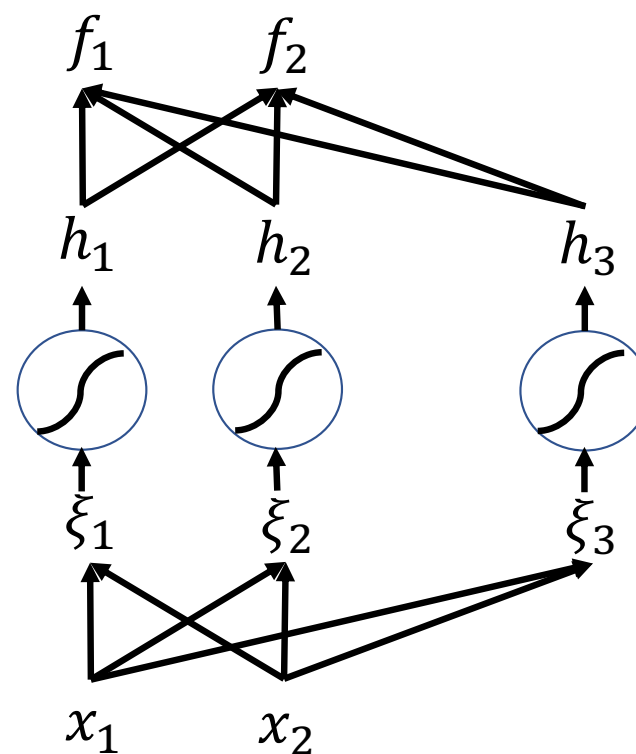
- Then you can run forward pass by just typing:

```
f = model(x)
```

```
loss = loss_function(f,y)
```

- You can still calculate all of the gradients by running

```
loss.backward()
```



torch.nn.Sequential: where are the parameters?

- The layers each have their own parameters, for example, a model created using the commands on the previous slide would have

```
model[0].weight
```

```
model[0].bias
```

```
model[2].weight
```

```
model[2].bias
```

- Accessing them that way requires you to know which layers have weights and biases, and which don't. An easier way is to use the function `model.parameters()`, which iterates through all trainable parameters, regardless of where they are actually stored:

```
for param in model.parameters():
```

```
    param -= learning_rate * param.grad
```

Conclusions

- Neural network forward propagation:

$$\xi_j^{(l)} = b_j^{(l)} + \sum_k w_{j,k}^{(l)} h_k^{(l-1)}, \quad h_j^{(l)} = g^{(l)}(\xi_j^{(l)})$$

- One-hot vectors

$$y_{i,c} = \begin{cases} 1 & c = \text{true class label of the } i^{\text{th}} \text{ token} \\ 0 & \text{otherwise} \end{cases}$$

- Softmax

$$f_c(\vec{x}_i) = \frac{e^{\vec{w}_c^T \vec{x}}}{\sum_{k=1}^V e^{\vec{w}_k^T \vec{x}}} = \text{P}(\text{Class} = c | X = \vec{x}, W)$$

- Stochastic gradient descent w/cross-entropy

$$\mathcal{L}_i = -\ln \text{P}(C = c_i | X = \vec{x}_i, W), \quad \vec{w} \leftarrow \vec{w} - \frac{\eta}{2} \nabla_{\vec{w}} \mathcal{L}_i$$

- Toolkits: https://pytorch.org/tutorials/beginner/pytorch_with_examples.html