Lecture 17: More on binary vs. multi-class classifiers

(Polychotomizers: One-Hot Vectors, Softmax, and Cross-Entropy)

Mark Hasegawa-Johnson, 3/9/2019.

Modified by Julia Hockenmaier

More on supervised learning

The supervised learning task

Given a **labeled training data set**
of N items $\mathbf{x}_n \in \mathcal{X}$ with labels $y_n \in \mathcal{Y}$

$$\mathcal{D}^{\text{train}} = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_N, y_N)\}$$

(y_n is determined by some unknown target function f($\mathbf{x}$))

Return a model g: $\mathcal{X} \longmapsto \mathcal{Y}$ that is a good approximation of f($\mathbf{x}$)

(g should assign correct labels y to unseen $\mathbf{x} \notin \mathcal{D}^{\text{train}}$)

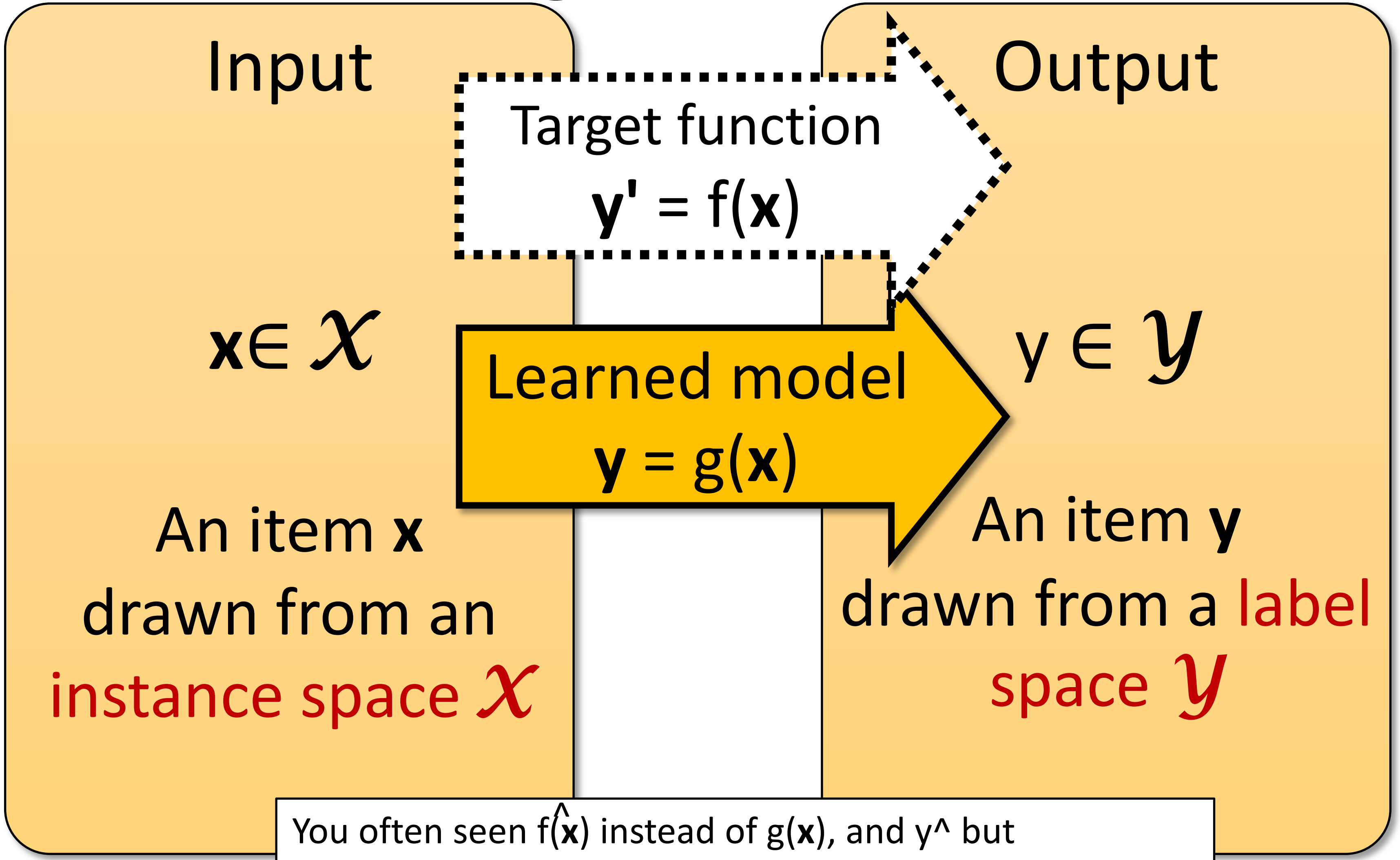
Supervised learning terms

Input items/data points $\mathbf{x}_n \in \mathcal{X}$ (e.g. emails)
are drawn from an **instance space $\mathcal{X}$**

Output labels $y_n \in \mathcal{Y}$ (e.g. 'spam'/'nospam')
are drawn from a **label space $\mathcal{Y}$**

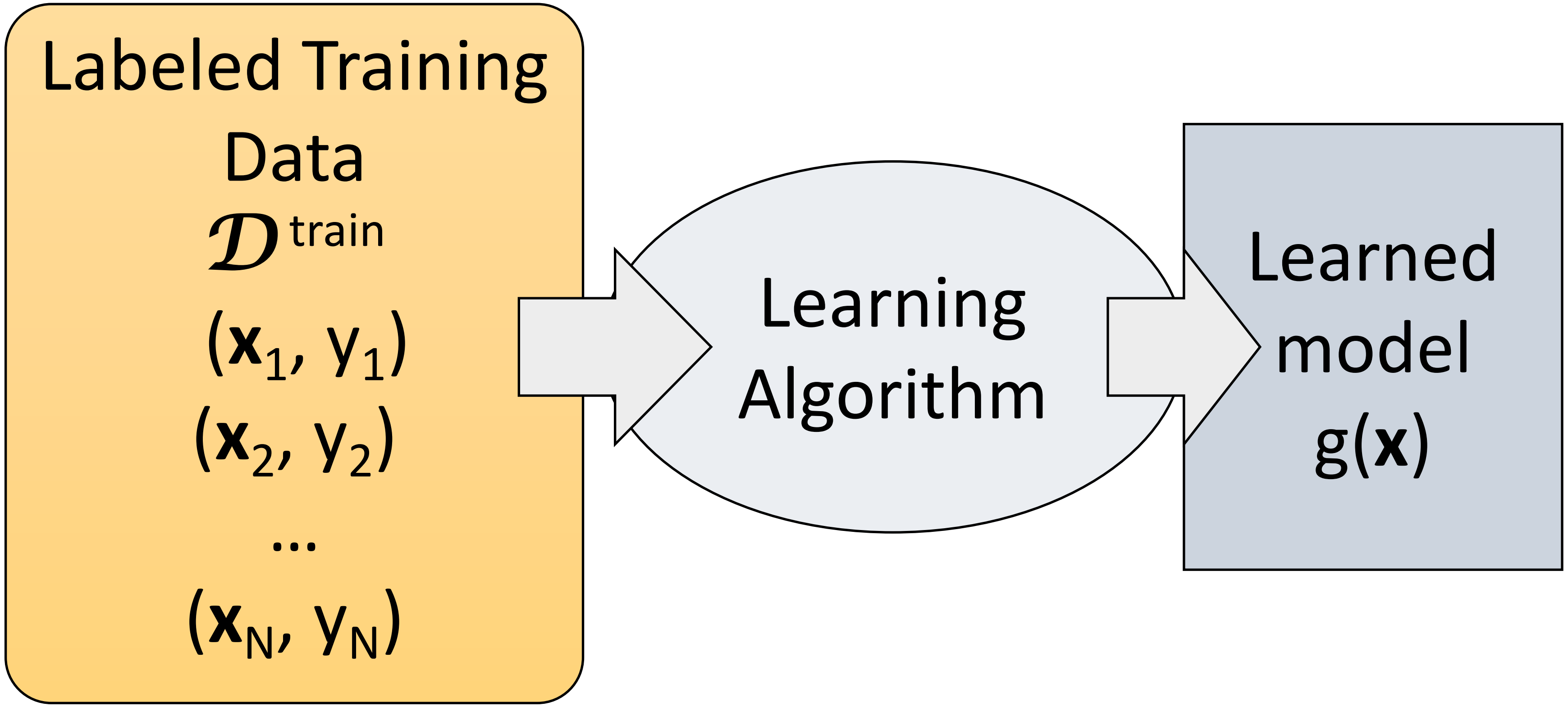
Every data point $\mathbf{x}_n \in \mathcal{X}$ has a *single* correct label $y_n \in \mathcal{Y}$,
defined by an (unknown**) target function** $f(\mathbf{x}) = y$

Supervised learning



You often seen $\hat{f}$(**x**) instead of g(**x**), and y^ but PowerPoint can't really typeset that, so g(**x**) and y' will have to do.

Supervised learning: Training



Give the learner examples in $\mathcal{D}^{\text{train}}$

The learner returns a model $g(\mathbf{x})$

Supervised learning: Testing

Labeled

Test Data

$\mathcal{D}^{\text{test}}$

$(\mathbf{x}'_1, y'_1)$

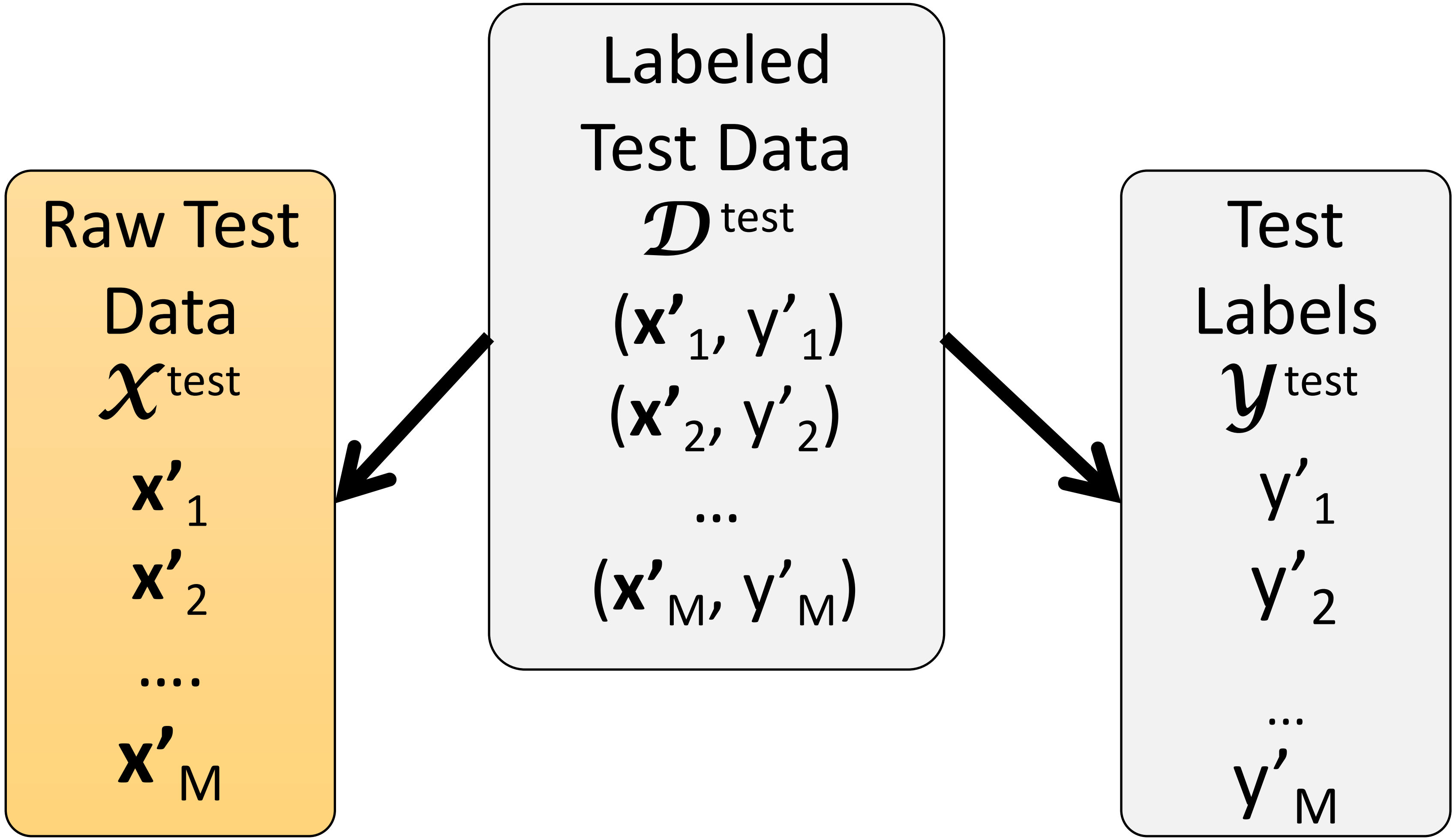
$(\mathbf{x}'_2, y'_2)$

…

$(\mathbf{x}'_M, y'_M)$

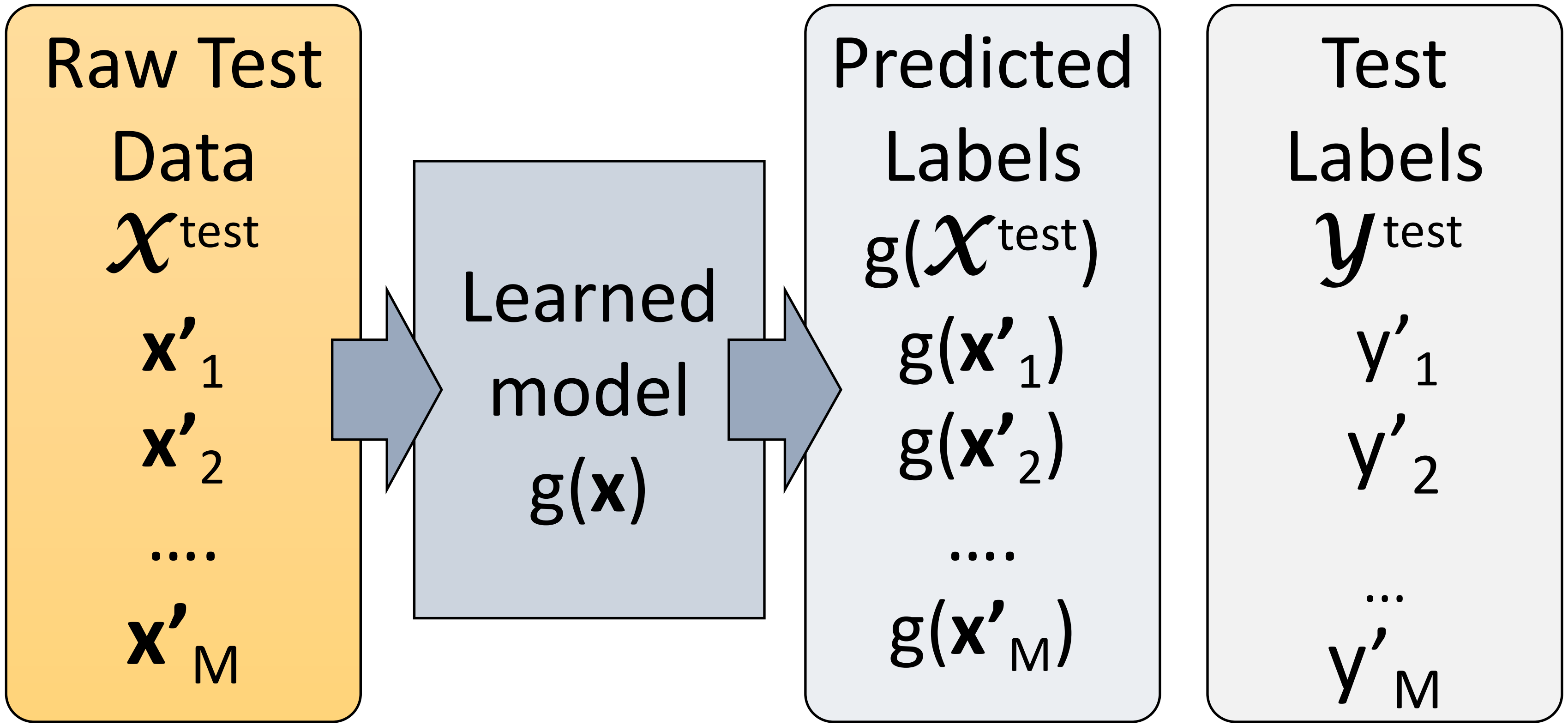
Reserve some labeled data for testing

Supervised learning: Testing



Supervised learning: Testing

Apply the model to the raw test data



Evaluating supervised learners

Use a **test data set** $\mathcal{D}^{\text{test}}$ that is *disjoint* from $\mathcal{D}^{\text{train}}$

$\mathcal{D}^{\text{test}} = \{(\mathbf{x'}_1, y'_1), \ldots, (\mathbf{x'}_M, y'_M)\}$

The learner has not seen the test items during learning.

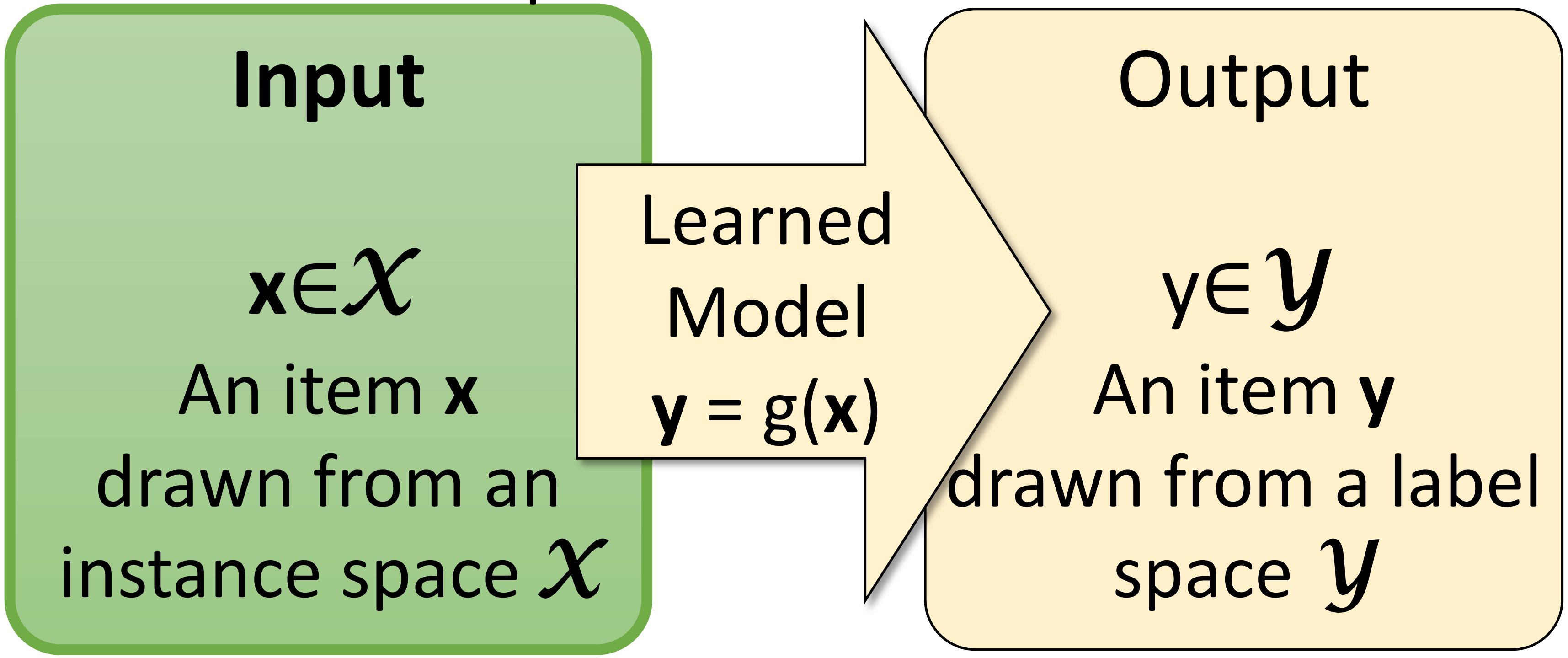
Split your labeled data into two parts: test and training.

Take all items $\mathbf{x'}_i$ in $\mathcal{D}^{\text{test}}$ and compare the predicted $f(\mathbf{x'}_i)$ with the correct y'_i .

This requires an evaluation metric (e.g. accuracy).

1. The instance space

1. The instance space $\mathcal{X}$



Designing an appropriate instance space $\mathcal{X}$
is crucial for how well we can predict y.

1. The instance space $\mathcal{X}$

When we apply machine learning to a task,
we first need to *define* the instance space $\mathcal{X}$.

Instances $\mathbf{x} \in \mathcal{X}$ are defined by **features**:

Boolean features:

Does this email contain the word ‘money’?

Numerical features:

How often does ‘money’ occur in this email?
What is the width/height of this bounding box?

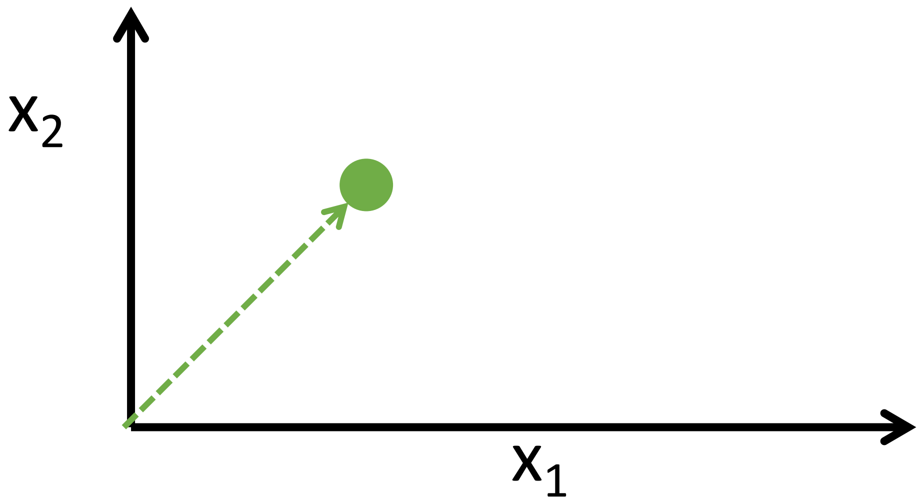
$\mathcal{X}$ as a vector space

$\mathcal{X}$ is an N-dimensional vector space (e.g. $\mathbb{R}^N$)

Each dimension = one feature.

Each **x** is a **feature vector** (hence the boldface **x**).

Think of **x** = $[x_1 \ldots x_N]$ as a point in $\mathcal{X}$:



From feature templates to vectors

When designing features, we often think in terms of templates, not individual features:

What is the 2nd letter?

N **a** oki → [1 0 0 0 …]

A **b** e → [0 1 0 0 …]

S **c** rooge → [0 0 1 0 …]

What is the *i*-th letter?

Abe → [**1** 0 0 0 0… 0 1 0 0 0 0… 0 0 0 0 **1** …]

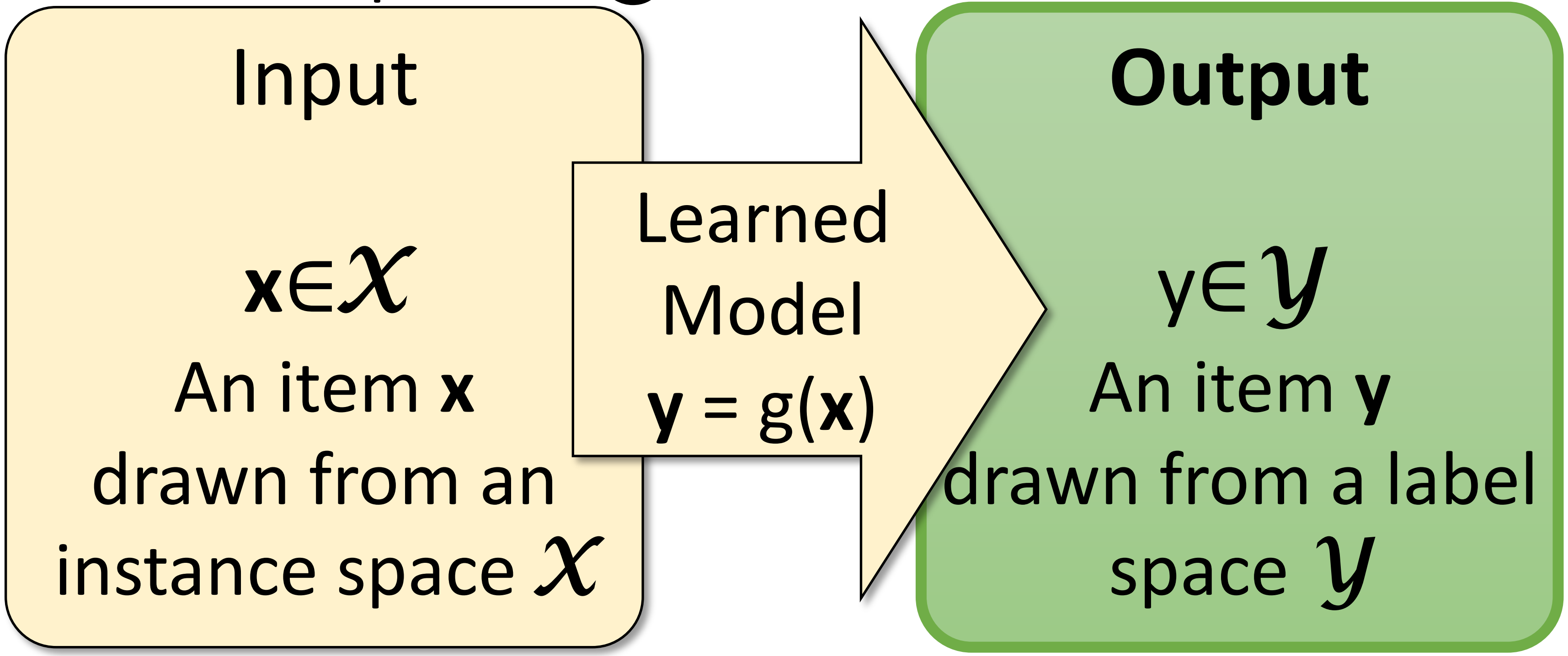
Good features are essential

- The choice of features is crucial
  for how well a task can be learned.
    - In many application areas (language, vision, etc.),
      a lot of work goes into designing suitable features.
    - This requires domain expertise.

- We can’t teach you what specific features
  to use for your task.
    - But we will touch on some general principles

2. The label space

2. The label space $\mathcal{Y}$



The label space $\mathcal{Y}$ determines **what *kind* of supervised learning task** we are dealing with

Supervised learning tasks I

Output labels $y \in \mathcal{Y}$ are **categorical**: CLASSIFICATION

Binary classification: Two possible labels

Multiclass classification: *k* possible labels

Output labels $y \in \mathcal{Y}$ are **structured objects**
(sequences of labels, parse trees, etc.)

Structure learning, etc.

Supervised learning tasks II

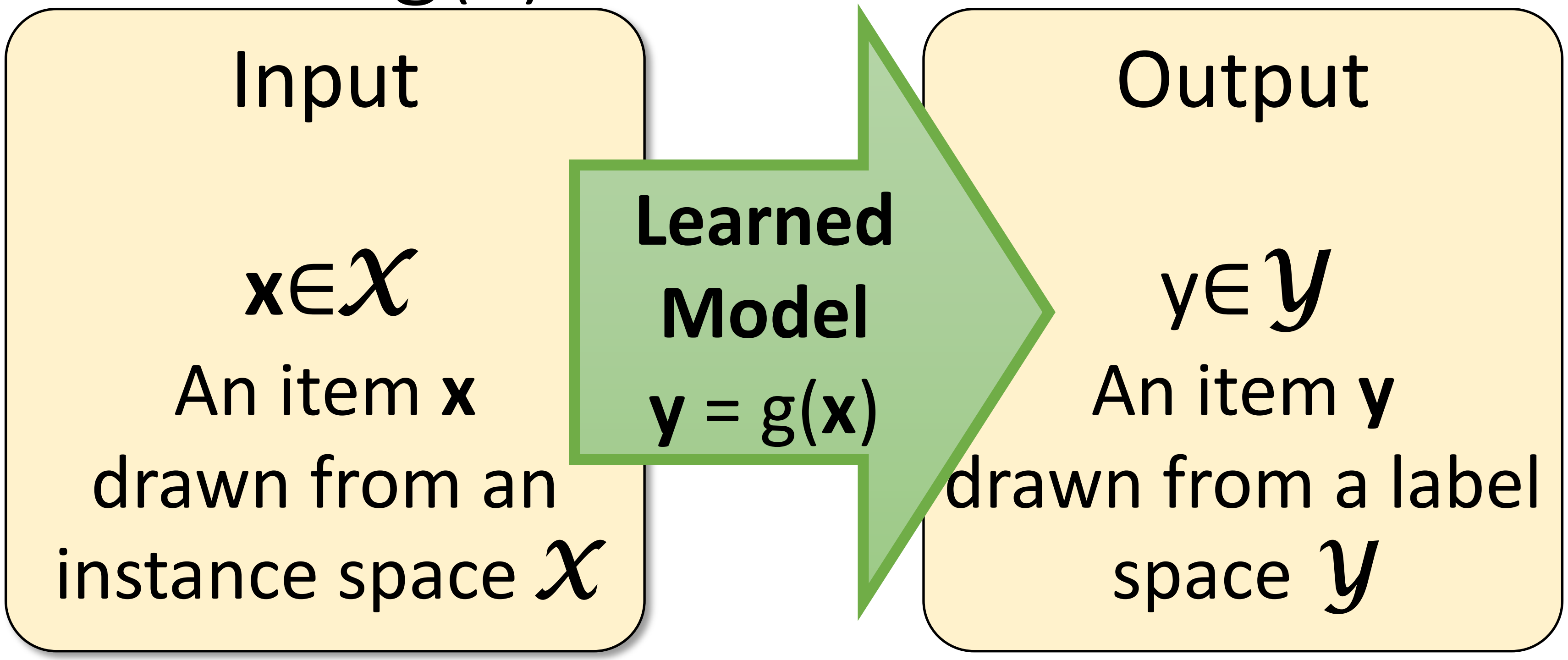
Output labels $y \in \mathcal{Y}$ are **numerical**:

Regression (linear/polynomial):
Labels are continuous-valued
Learn a linear/polynomial function f(x)

Ranking:
Labels are ordinal
Learn an ordering $f(x_1) > f(x_2)$ over input

3. Models
(The hypothesis space)

3. The model g(**x**)



We need to choose what *kind* of model we want to learn

More terminology

For classification tasks ($\mathcal{Y}$ is categorical, e.g. {0, 1}, or {0, 1, …, k}), the model is called a **classifier**.

For **binary classification tasks** ($\mathcal{Y}$ = {0, 1} or $\mathcal{Y}$ = {-1, +1}),

we can either think of the two values of $\mathcal{Y}$ as Boolean or as positive/negative

A learning problem

	x_1	x_2	x_3	x_4	y
1	0	0	1	0	0
2	0	1	0	0	0
3	0	0	1	1	1
4	1	0	0	1	1
5	0	1	1	0	0
6	1	1	0	0	0
7	0	1	0	1	0

A learning problem

Each **x** has 4 bits: $|\mathcal{X}| = 2^4 = 16$

Since $\mathcal{Y} = \{0, 1\}$, each f(**x**)
defines one subset of $\mathcal{X}$

$\mathcal{X}$ has $2^{16} = 65536$ subsets:
There are 2^{16} possible f(**x**)
(2^9 are consistent with our data)

We would need to see all of $\mathcal{X}$ to learn f(**x**)

A learning problem

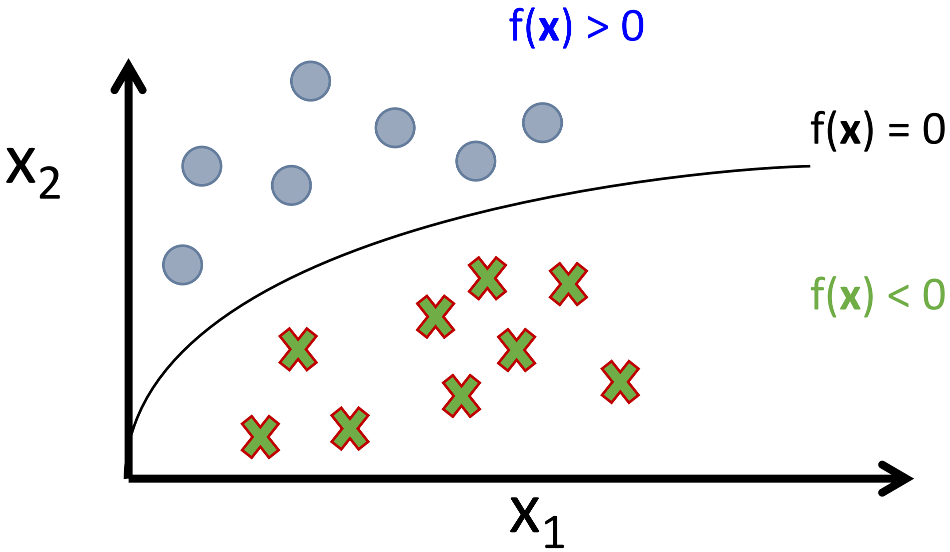
We would need to see all of $\mathcal{X}$ to learn f(**x**)

Easy with $|\mathcal{X}|$=16

Not feasible in general
(for any real-world problems)

Learning = generalization,
not memorization of the training data

Classifiers in vector spaces



Binary classification:

We assume f *separates* the positive and negative examples:

Assign y = 1 to all **x** where f(**x**) > 0

Assign y = 0 (or -1) to all **x** where f(**x**) < 0

Learning a classifier

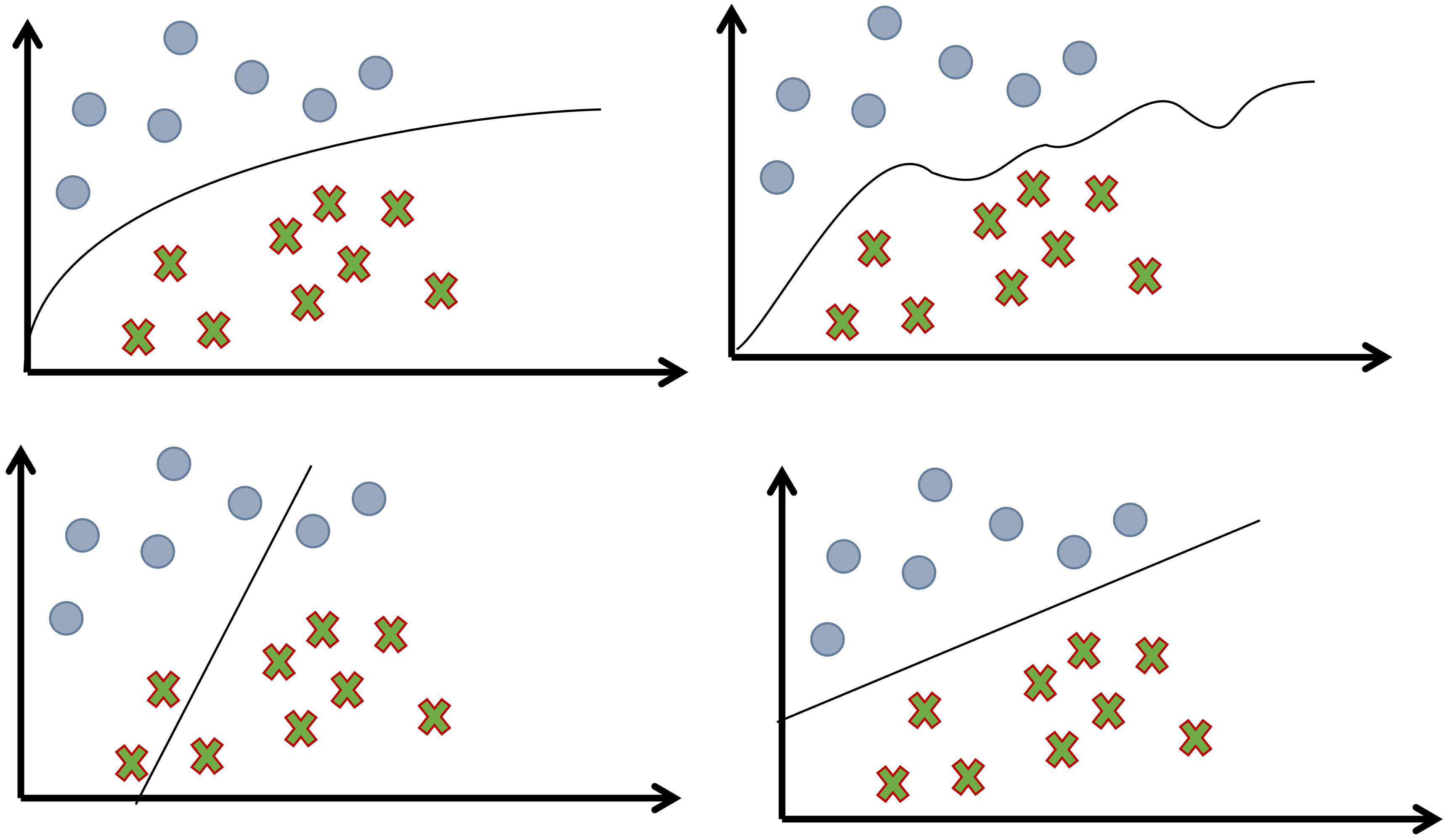
The learning task:
Find a function f(**x**) that best separates the (training) data

What kind of function is f?

How do we define *best*?

How do we find f?

Which model should we pick?



Criteria for choosing models

Accuracy:

Prefer models that make **fewer mistakes**

We only have access to the training data

But we care about accuracy on unseen (test) examples

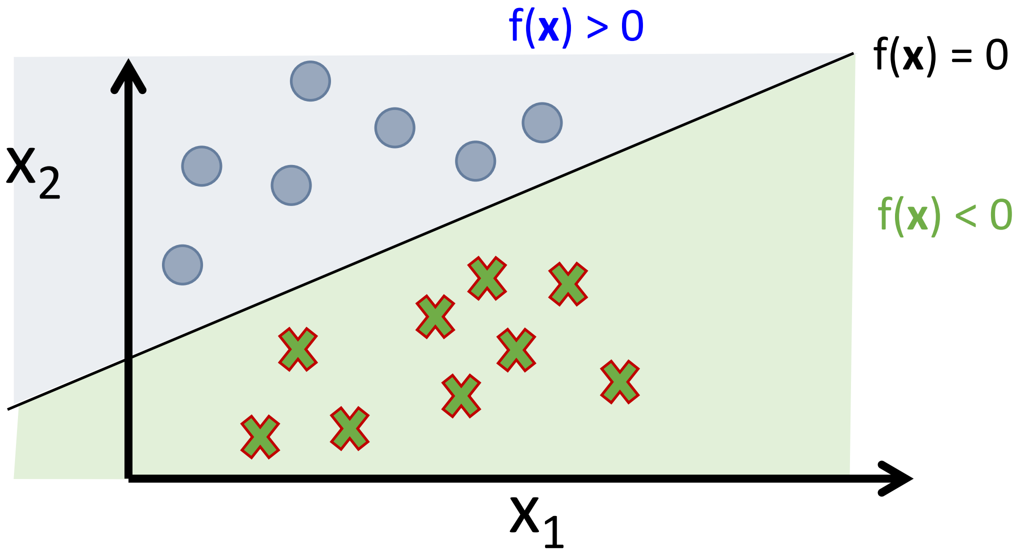
Simplicity (Occam's razor):

Prefer **simpler** models (e.g. fewer parameters).

These (often) generalize better,
and need less data for training.

Linear classifiers

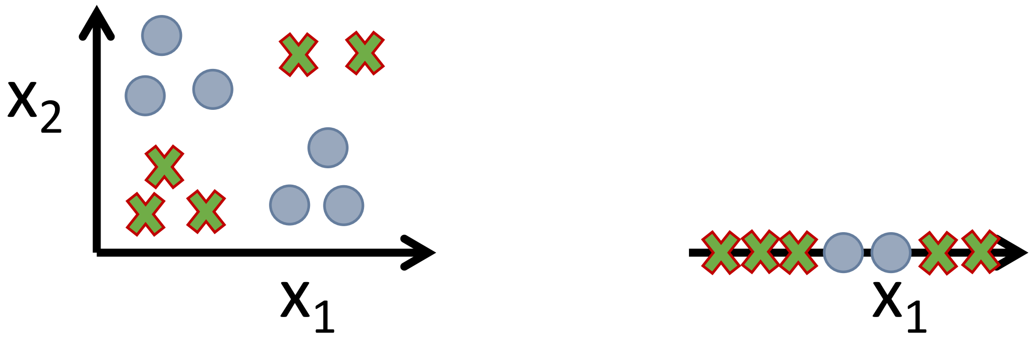
Linear classifiers



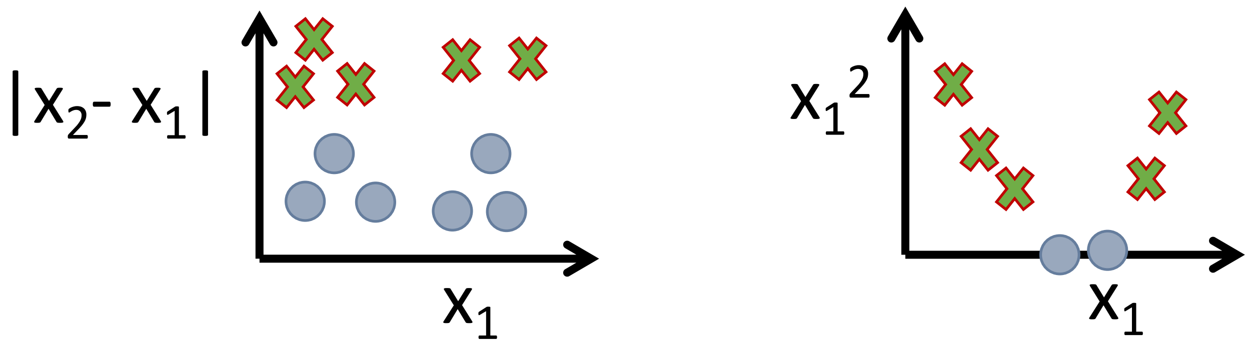
Many learning algorithms restrict the hypothesis space to **linear classifiers**:
$f(\mathbf{x}) = w_0 + \mathbf{wx}$

Linear Separability

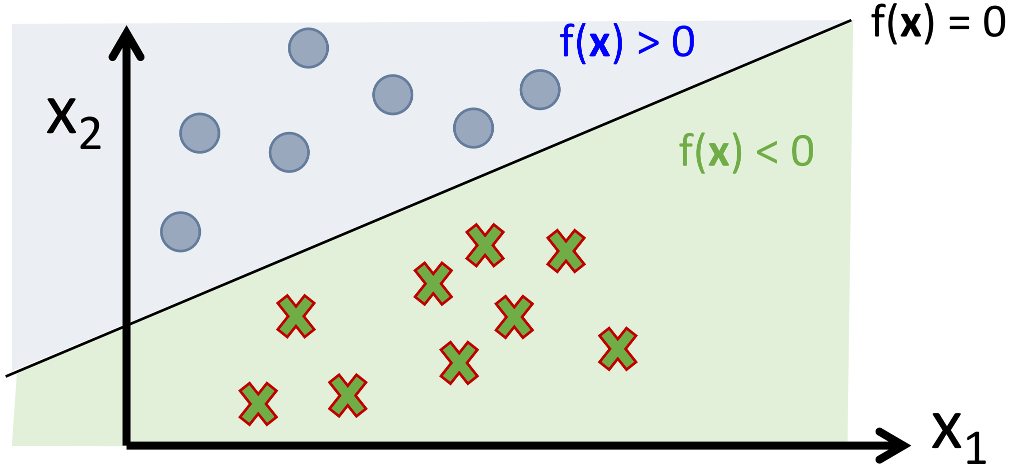
- Not all data sets are linearly separable:



- Sometimes, feature transformations help:



Linear classifiers: $f(\mathbf{x}) = w_0 + \mathbf{wx}$



Linear classifiers are defined over vector spaces

Every hypothesis $f(\mathbf{x})$ is a hyperplane:

$$f(\mathbf{x}) = w_0 + \mathbf{wx}$$

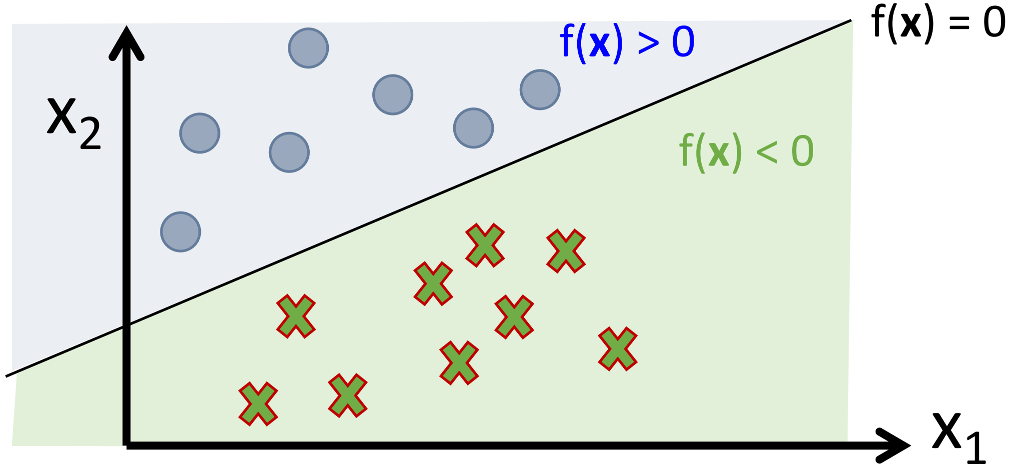
$f(\mathbf{x})$ is also called the decision boundary

Assign $\hat{y} = +1$ to all $\mathbf{x}$ where $f(\mathbf{x}) > 0$

Assign $\hat{y} = -1$ to all $\mathbf{x}$ where $f(\mathbf{x}) < 0$

$$\hat{y} = \text{sgn}(f(\mathbf{x}))$$

$y \cdot f(\mathbf{x}) > 0$: Correct classification



An example ($\mathbf{x}$, y) is **correctly classified** by f($\mathbf{x}$) if and only if $y \cdot f(\mathbf{x}) > 0$:

Case 1 ($y = +1 = \hat{y}$): $f(\mathbf{x}) > 0 \Rightarrow y \cdot f(\mathbf{x}) > 0$

Case 2 ($y = -1 = \hat{y}$): $f(\mathbf{x}) < 0 \Rightarrow y \cdot f(\mathbf{x}) > 0$

Case 3 ($y = +1 \neq \hat{y} = -1$): $f(\mathbf{x}) > 0 \Rightarrow y \cdot f(\mathbf{x}) < 0$

Case 4 ($y = -1 \neq \hat{y} = +1$): $f(\mathbf{x}) < 0 \Rightarrow y \cdot f(\mathbf{x}) < 0$

With a separate bias term w_0: $f(\mathbf{x}) = \mathbf{w}\cdot\mathbf{x} + w_0$

The **instance space** $\mathcal{X}$ is a ***d*-dimensional vector space** (each $\mathbf{x}\in\mathcal{X}$ has *d* elements)

The **decision boundary** $f(\mathbf{x}) = 0$ is **a (*d*−1)-dimensional hyperplane** in the instance space.

The **weight vector w** is **orthogonal (normal)** to the decision boundary $f(\mathbf{x}) = 0$:

For any two points $\mathbf{x}^A$ and $\mathbf{x}^B$ on the decision boundary $f(\mathbf{x}^A) = f(\mathbf{x}^B) = 0$

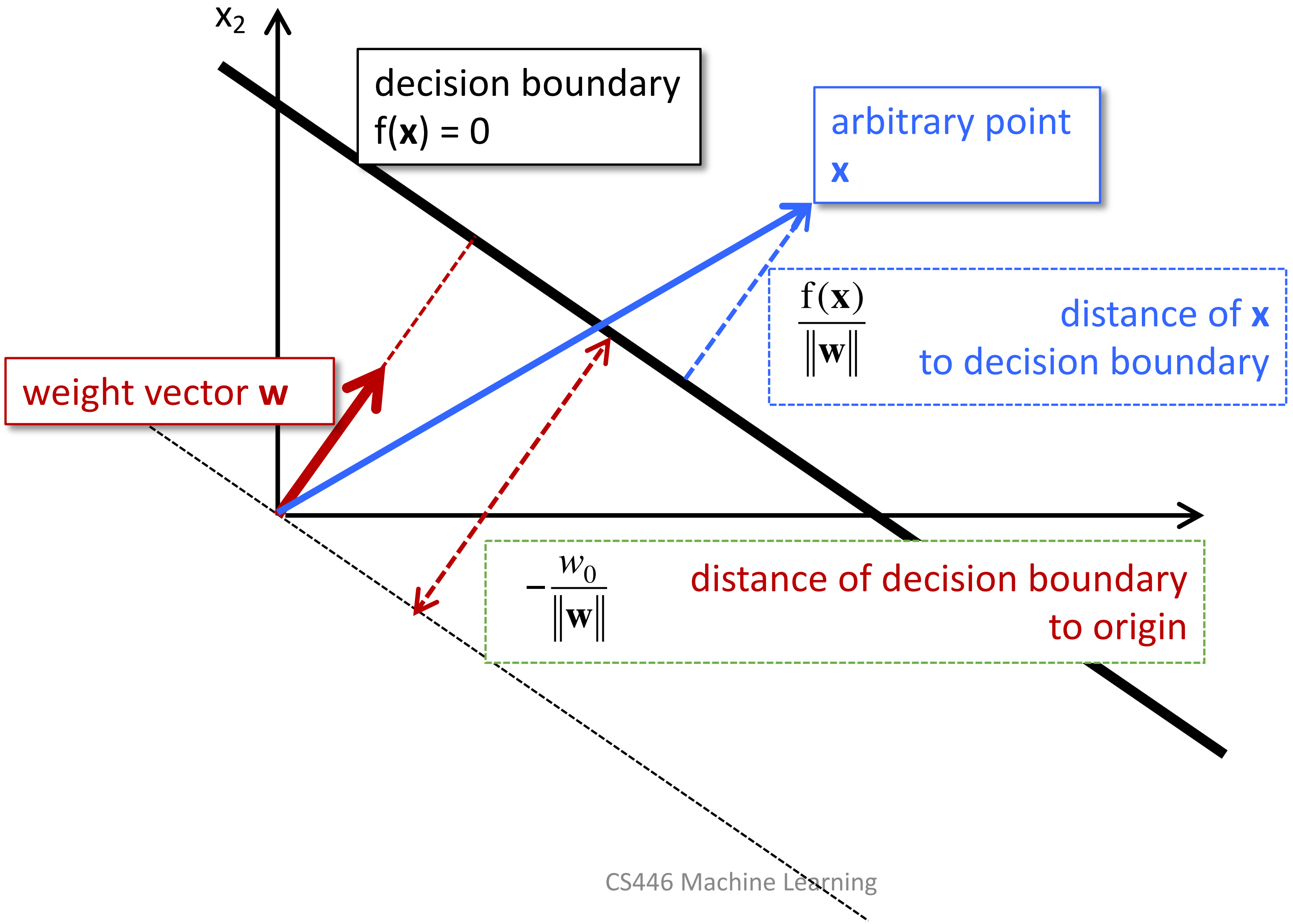
For any vector $(\mathbf{x}^B - \mathbf{x}^A)$ on the decision boundary: $\mathbf{w}(\mathbf{x}^B - \mathbf{x}^A) = f(\mathbf{x}^B) - w_0 - f(\mathbf{x}^A) + w_0 = 0$

The **bias term** w_0 determines the **distance of the decision boundary** from the origin:

For $\mathbf{x}$ with $f(\mathbf{x}) = 0$, the distance to the origin is

$$\frac{\mathbf{w}\cdot\mathbf{x}}{\|\mathbf{w}\|} = -\frac{w_0}{\|\mathbf{w}\|} = -\frac{w_0}{\sqrt{\sum_{i=1}^{d} w_i^2}}$$

With a separate bias term w_0: $f(\mathbf{x}) = \mathbf{w} \cdot \mathbf{x} + w_0$



Canonical representation: getting rid of the bias term

With $\mathbf{w} = (w_1, \ldots, w_N)^T$ and $\mathbf{x} = (x_1, \ldots, x_N)^T$:

$$f(x) = w_0 + \mathbf{wx}$$
$$= w_0 + \sum_{i=1\ldots N} w_i x_i$$

w_0 is called the **bias term.**

The **canonical representation** redefines **w, x** as

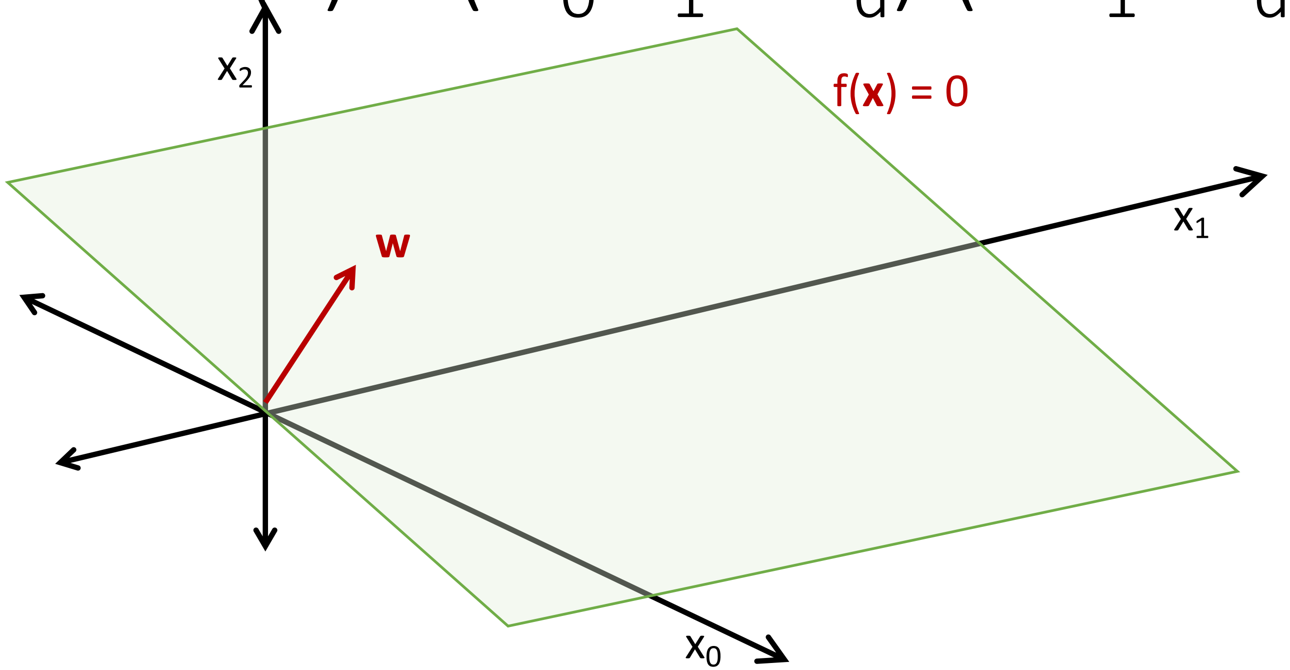
$$\mathbf{w} = (w_0, w_1, \ldots, w_N)^T$$

and $$\mathbf{x} = (1, \quad x_1, \ldots, x_N)^T$$

=> $$f(\mathbf{x}) = \mathbf{w} \cdot \mathbf{x}$$

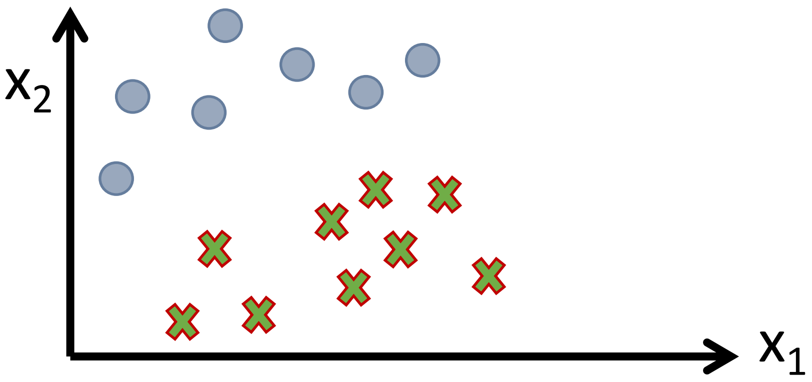
In canonical form (with $x_0 = 1$)

$f(\mathbf{x}) = (w_0 w_1 \ldots w_d) \cdot (1\ x_1 \ldots x_d)$

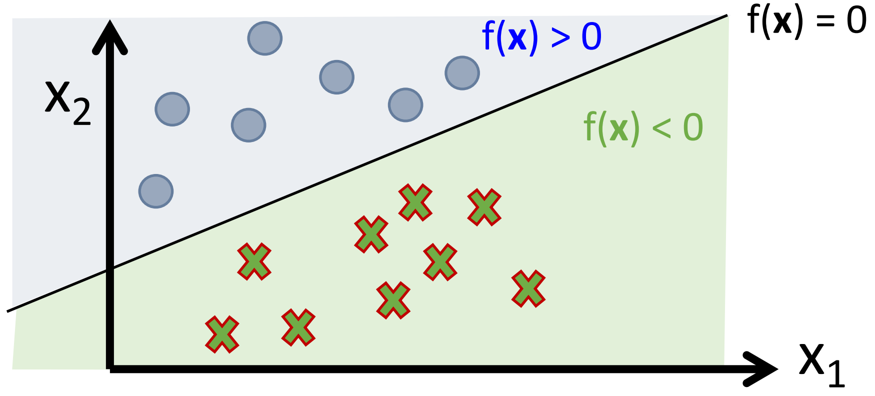


- We now operate **in (*d*+1)-dimensional space**
- The **decision boundary** $f(\mathbf{x}) = 0$ is a *d*-dimensional hyperplane that goes through the origin.
- The **weight vector w** is still orthogonal to the decision boundary $f(\mathbf{x}) = 0$

Learning a linear classifier



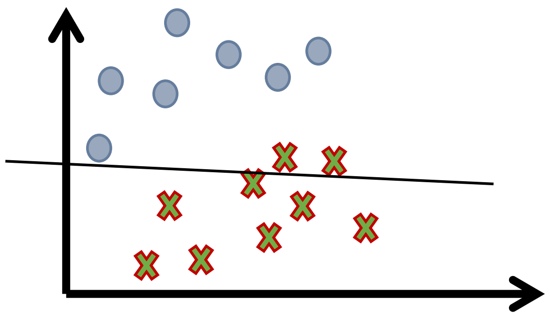
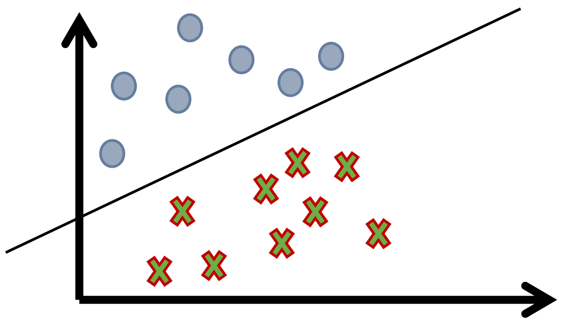
Input: Labeled training data
$\mathcal{D} = \{(\mathbf{x}^1, y^1),\ldots,(\mathbf{x}^D, y^D)\}$
plotted in the sample space $\mathcal{X} = \mathbf{R}^2$
with : $y^i = +1$, : $y^i = 1$



Output: A decision boundary $f(\mathbf{x}) = 0$
that separates the training data

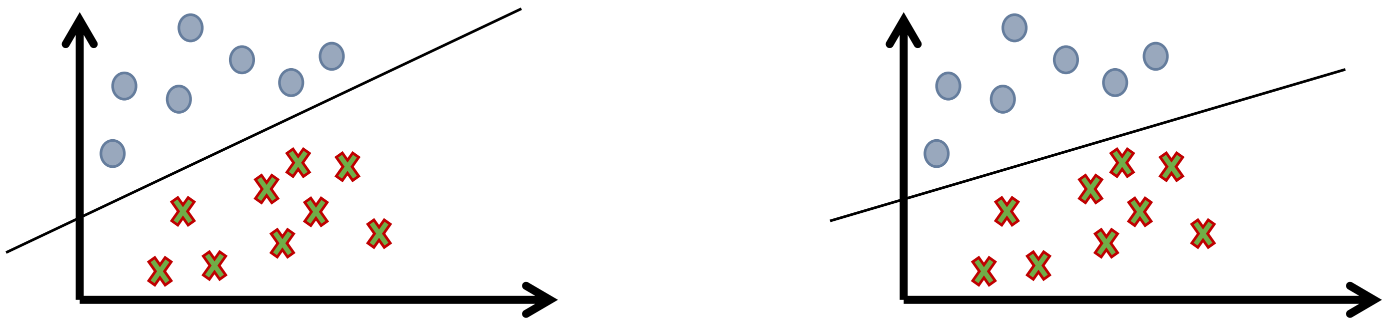
$$y^i \cdot f(\mathbf{x}^i) > 0$$

Which model should we pick?

- We need a metric (aka an objective function)
- We would like to minimize the probability of misclassifying *unseen* examples, but we can't measure that probability.
- Instead: minimize the number of misclassified training examples

Which model should we pick?



- We need a more specific metric:
  There may be many models that are consistent with the training data.

- Loss functions provide such metrics.

4. The learning algorithm

4. The learning algorithm

- **The learning task:**
  Given a labeled training data set
  $$\mathcal{D}^{\text{train}} = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_N, y_N)\}$$
  return a model (classifier) g: $\mathcal{X} \longmapsto \mathcal{Y}$
  from the hypothesis space $\mathcal{H} \subseteq |\mathcal{Y}|^{|\mathcal{X}|}$

Batch versus online training

Batch learning:

The learner sees the complete training data, and only changes its hypothesis when it has seen **the entire training data set**.

Online training:

The learner sees the training data one example at a time,
and can change its hypothesis **with every new example**

Compromise: Minibatch learning (commonly used in practice)

The learner sees **small sets of training examples** at a time,
and changes its hypothesis with every such minibatch of examples

Perceptron

Perceptron

- Simple, **mistake-driven** algorithm for learning linear classifiers

- There are batch and online versions
- We will analyze the online version

- Uses (stochastic) gradient descent, with a particular loss function

Perceptron criterion

We would like a weight vector **w** such that

$f(\mathbf{x}_n) = \mathbf{w}\cdot\mathbf{x}_n > 0$ for $y_n = +1$

$f(\mathbf{x}_n) = \mathbf{w}\cdot\mathbf{x}_n < 0$ for $y_n = -1$

The perceptron tries to minimize the error

$$-\mathbf{w}\cdot\mathbf{x}_n\cdot y_n$$

for any misclassified example ($\mathbf{x}_n$, y_n)

The overall training error of **w** depends on the misclassified items M:

$$\mathrm{E}_{Perceptron}(\mathbf{w}) = -\sum_{n\in M} \mathbf{w}\cdot\mathbf{x}_n\cdot y_n$$

Perceptron

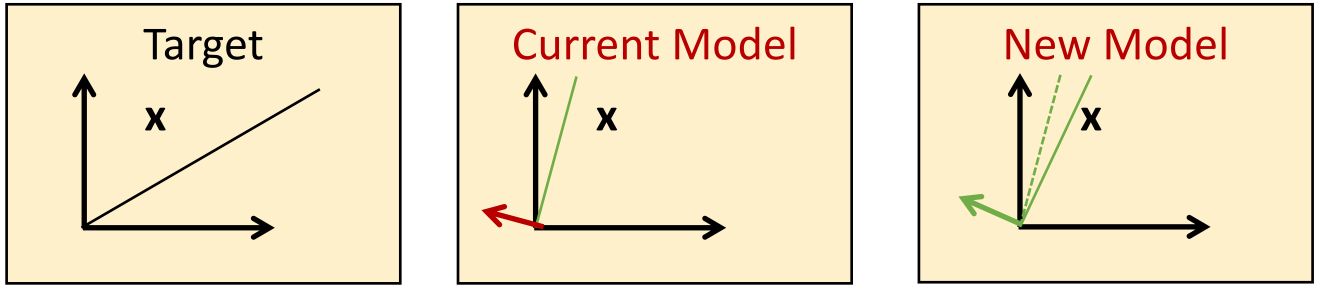
For each training instance $\vec{\boldsymbol{f}}$ with label $y \in \{-1,1\}$:

- Classify with current weights: $y' = \text{sgn}(\vec{w}^T \vec{f})$
  - Notice $y' \in \{-1,1\}$ too.
- Update weights:
  - if $y = y'$ then do nothing
  - if $y \neq y'$ then $\vec{w} = \vec{w} + \eta\, y\, \vec{f}$
  - η (eta) is a "learning rate." More about that later.

The Perceptron rule

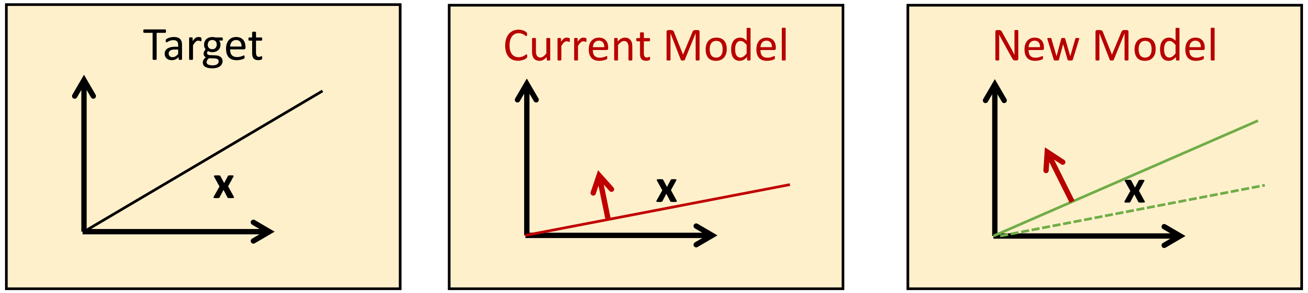
If target y = +1: x should be **above** the decision boundary

Lower the decision boundary's slope: $\mathbf{w}^{i+1} := \mathbf{w}^{i} + \mathbf{x}$

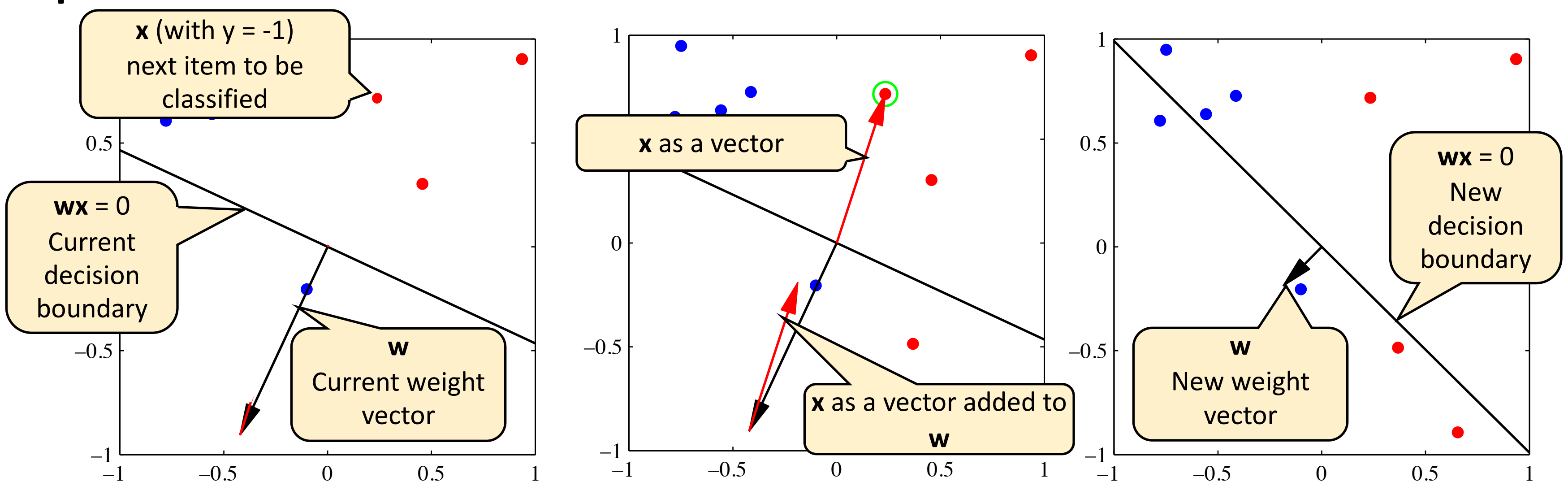


If target y = -1: **x** should be **below** the decision boundary

Raise the decision boundary's slope: $\mathbf{w}^{i+1} := \mathbf{w}^{i} - \mathbf{x}$

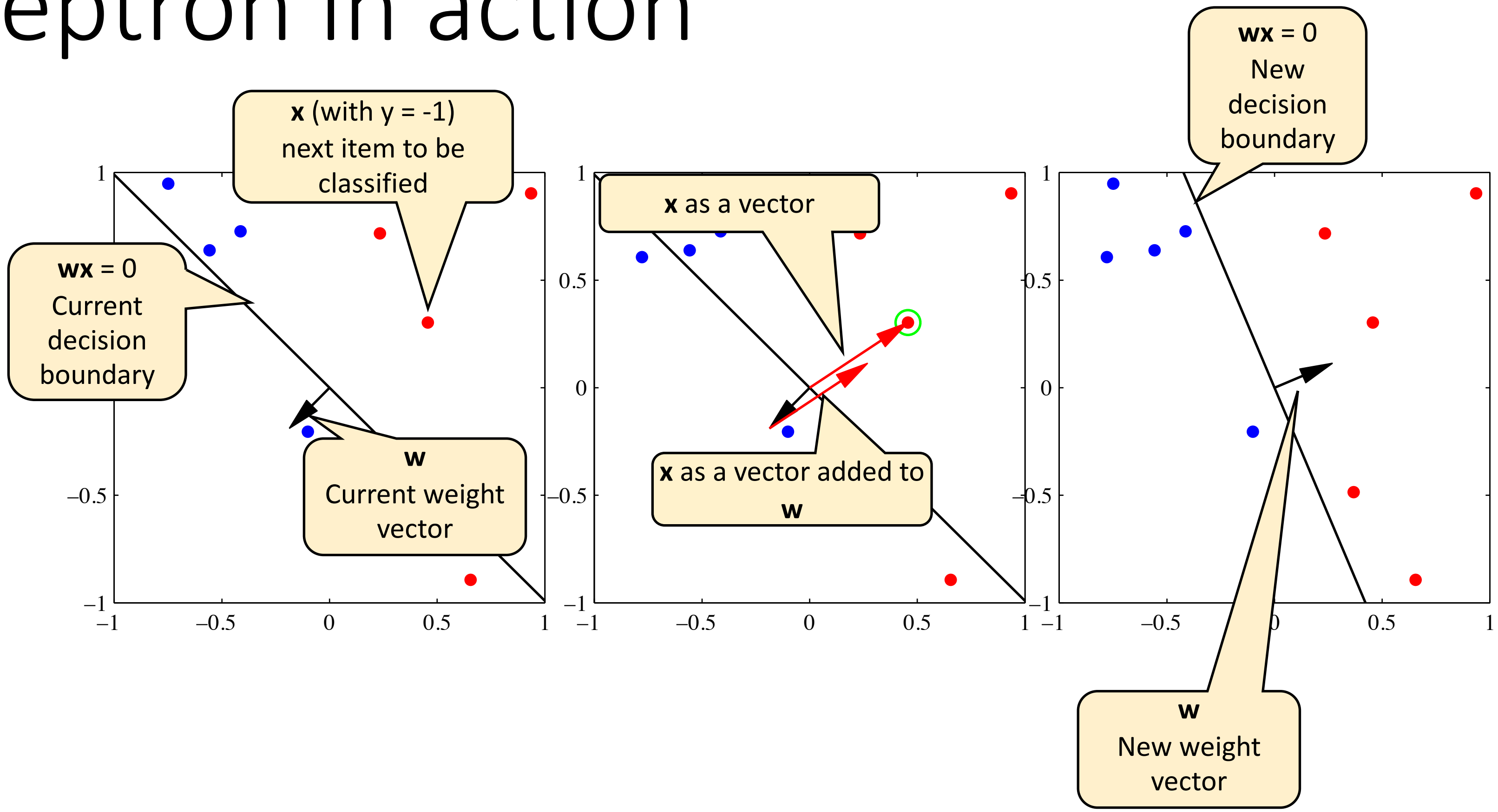


Perceptron in action



(Figures from Bishop 2006)

Perceptron in action



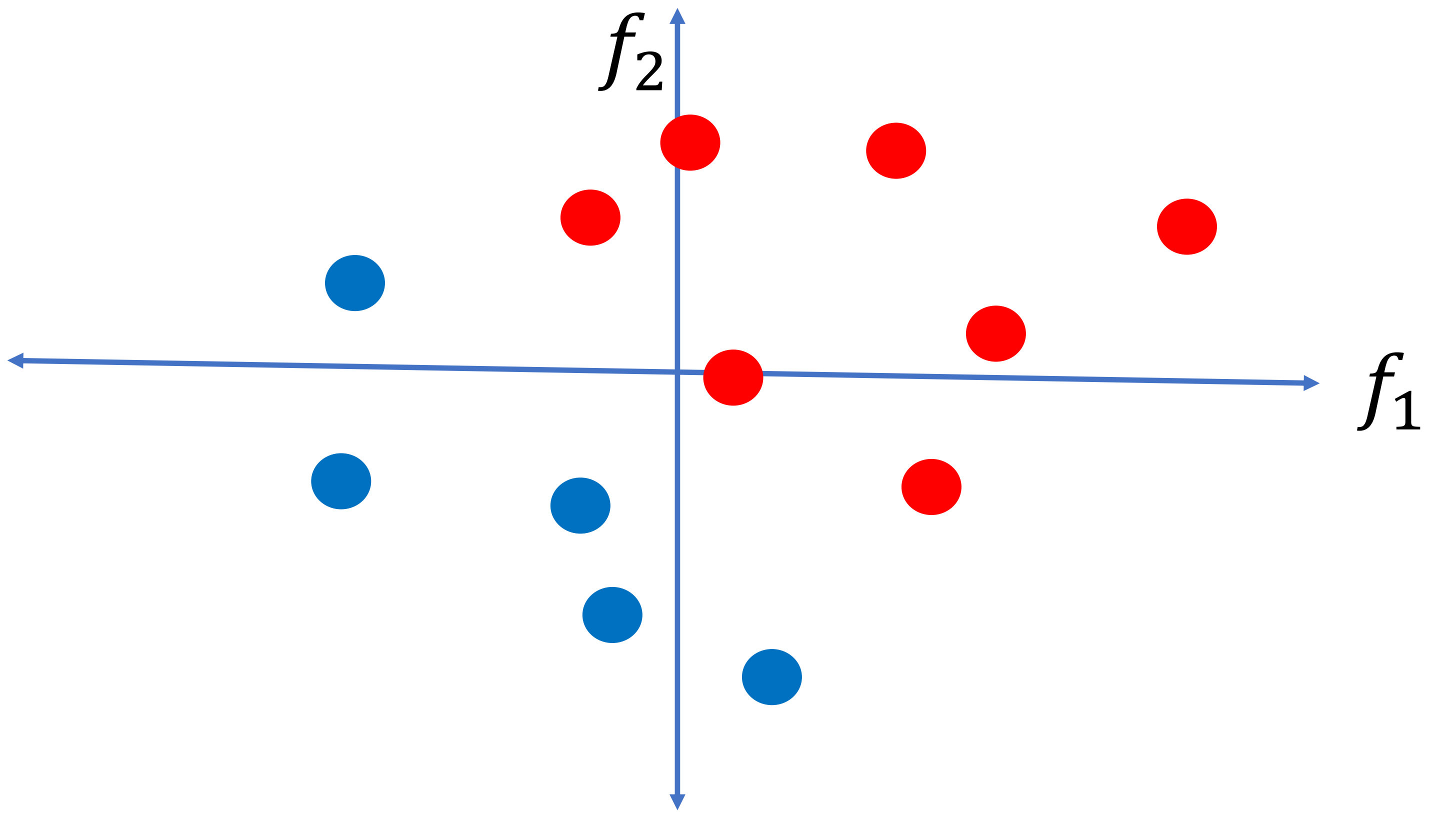
(Figures from Bishop 2006)

Perceptron: Proof of Convergence

- If the data are linearly separable (if there exists a $\vec{w}$ vector such that the true label is given by $y' = \text{sgn}(\vec{w}^T \vec{f})$), then the perceptron algorithm is guarantee to converge, even with a constant learning rate, even $\eta=1$.
- In fact, training a perceptron is often the fastest way to find out if the data are linearly separable. If $\vec{w}$ converges, then the data are separable; if $\vec{w}$ diverges toward infinity, then no.
- If the data are not linearly separable, then perceptron converges iff the learning rate decreases, e.g., $\eta=1/n$ for the n'th training sample.

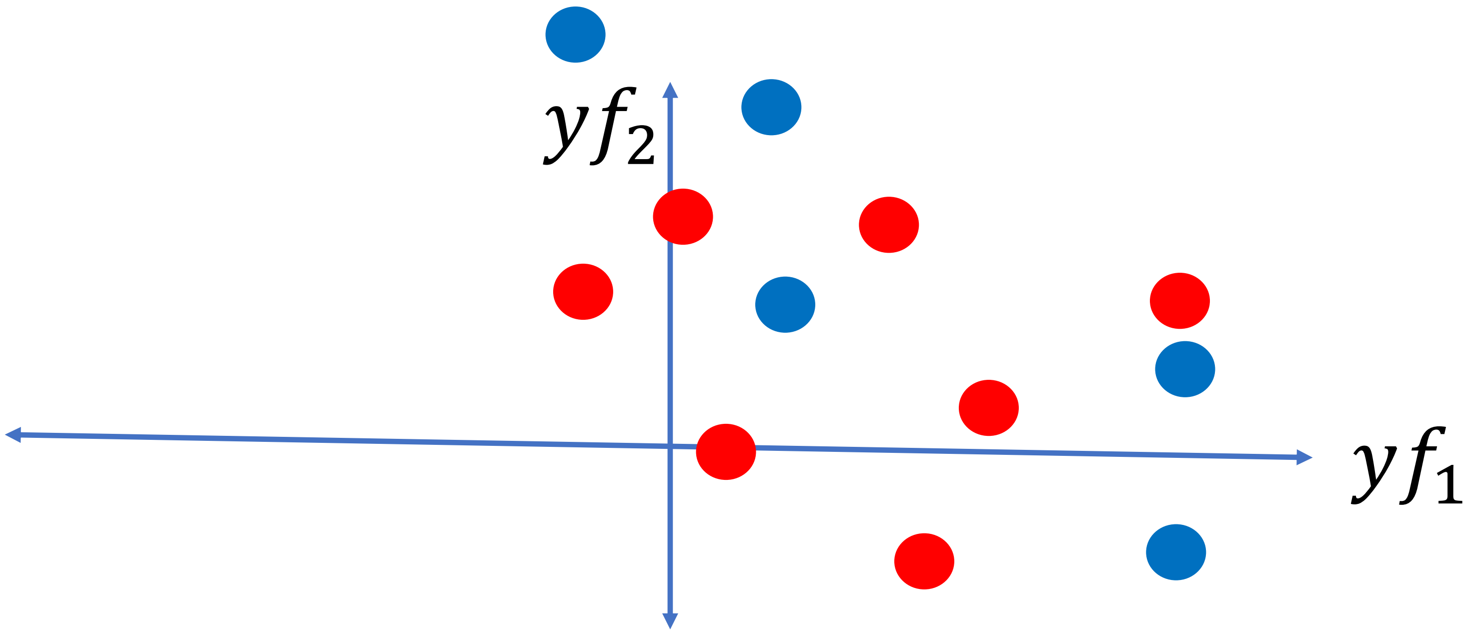
Perceptron: Proof of Convergence

Suppose the data are linearly separable. For example, suppose red dots are the class y=1, and blue dots are the class y=-1:



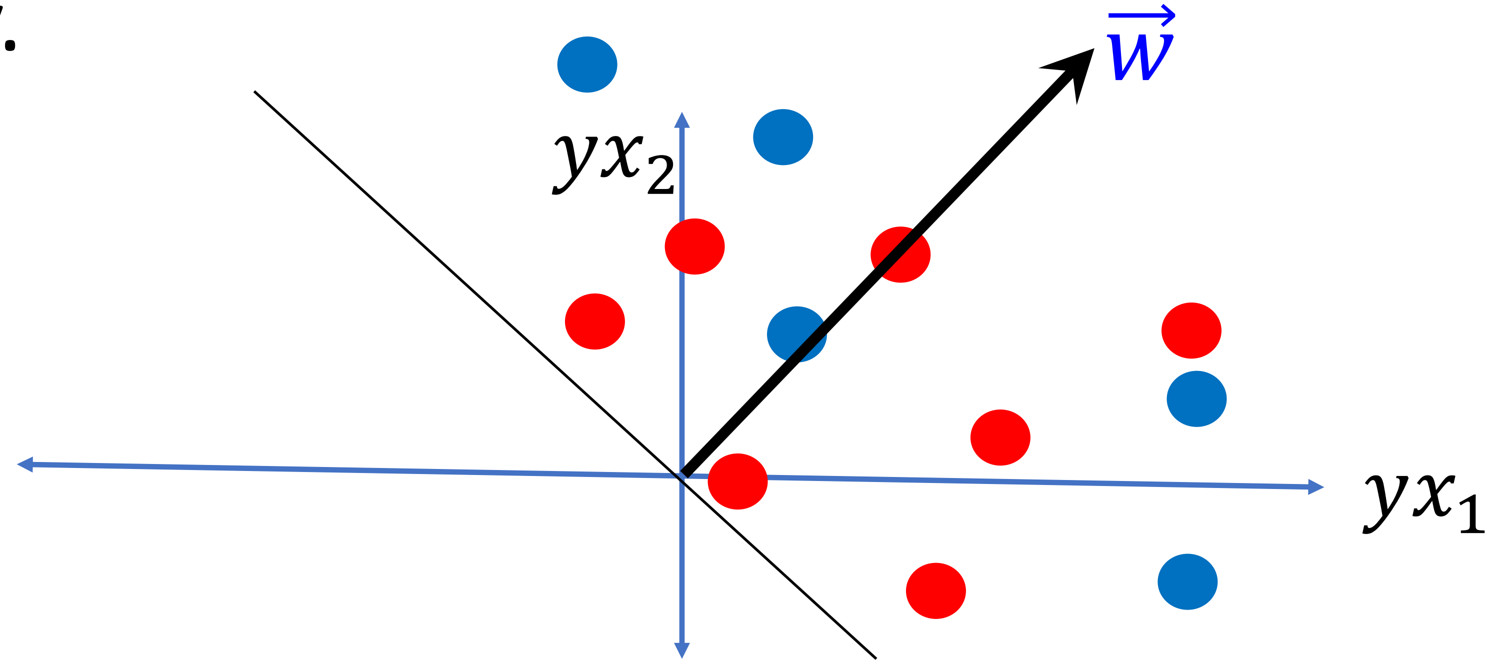
Perceptron: Proof of Convergence

- Instead of plotting $\vec{f}$, plot $y\times\vec{f}$. The red dots are unchanged; the blue dots are multiplied by -1.
- Since the original data were linearly separable, the new data are all in the same half of the feature space.



Perceptron: Proof of Convergence

- Remember the perceptron training rule: if any example is misclassified, then we use it to update $\vec{w} = \vec{w} + y\,\vec{f}$.
- So eventually, $\vec{w}$ becomes just a weighted average of $y\vec{f}$.
- … and the perpendicular line, $\vec{w}^T\vec{f} = 0$, is the classifier boundary.



Perceptron: Proof of Convergence: Conclusion

- If the data are linearly separable, then the perceptron will eventually find the equation for a line that separates them.
- If the data are NOT linearly separable, then perceptron converges iff the learning rate decreases, e.g., $\eta=1/n$ for the n'th training sample. .... In this case, convergence is trivially obvious, because y and $\vec{f}$ are finite, therefore the weight updates $\eta\, y\, \vec{f}$ approach 0 as η approaches 0.

Implementation details

- Bias (add feature dimension with value fixed to 1) vs. no bias
- Initialization of weights: all zeros vs. random
- Learning rate decay function
- Number of epochs (passes through the training data)
- Order of cycling through training examples (random)

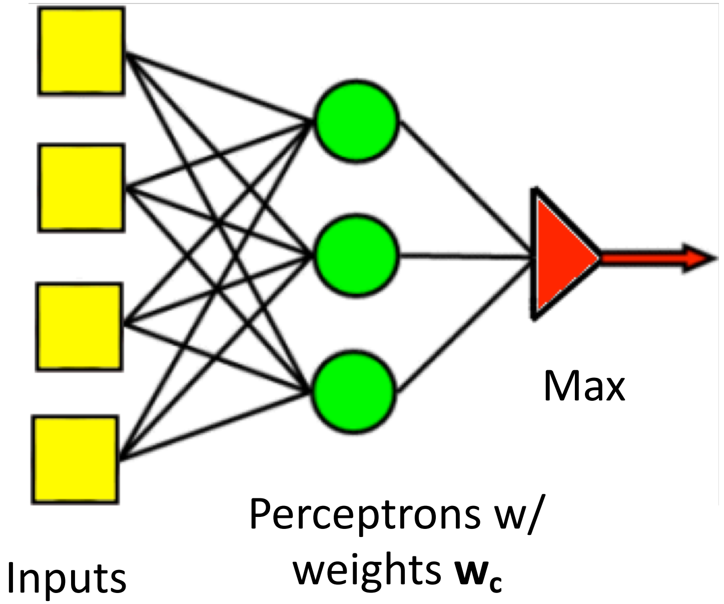
Multi-class Perceptrons

Multi-class perceptrons

- *One-vs-others* framework: Need to keep a weight vector $\mathbf{w}_c$ for each class c
- Decision rule: $y = \text{argmax}_c\ \mathbf{w}_c \cdot \mathbf{f}$
- Update rule: suppose example from class c gets misclassified as c'
  - Update for c: $\mathbf{w}_c \leftarrow \mathbf{w}_c + \eta\mathbf{f}$
  - Update for c': $\mathbf{w}_{c'} \leftarrow \mathbf{w}_{c'} - \eta\mathbf{f}$
  - Update for all classes other than c and c': no change

Multi-class perceptrons

- *One-vs-others* framework: Need to keep a weight vector $\mathbf{w}_c$ for each class c
- Decision rule: $y = \text{argmax}_c\ \mathbf{w}_c \cdot \mathbf{f}$



One-Hot Vector

- Example: if the first example is from class 2 (red), then $\vec{y}_1 = [0,1,0]$

$$y_{ij} = \begin{cases} 1 & \text{i}^{\text{th}} \text{ example is from class j} \\ 0 & \text{i}^{\text{th}} \text{ example is NOT from class j} \end{cases}$$

Call y_{ij} the **reference label**, and call $\hat{y}_{ij}$ the **hypothesis**. Then notice that:

- $y_{ij} =$ True value of $P(class\ j\ |\vec{f}_i)$, because the true probability is always either 1 or 0!
- $\hat{y}_{ij} =$ Estimated value of $P(class\ j\ |\vec{f}_i)$, $\ 0 \le \hat{y}_j \le 1, \quad \sum_{j=1}^{c} \hat{y}_j = 1$

Wait. Dichotomizer is just a Special Case of Polychotomizer, isn't it?

Yes. Yes, it is.

- Polychotomizer: $\vec{y}_i = [y_{i1}, \dots, y_{ic}],\ y_{ij} = P(class\ j\ |\vec{f}_i)$.
- Dichotomizer: $y_i = P(class\ 1\ |\vec{f}_i)$
- That's all you need, because if there are only two classes, then $P(other\ class\ |\vec{f}_i) = 1 - y_i$
- (One of the two classes in a dichotomizer is always called "class 1." The other might be called "class 2," or "class 0," or "class -1"…. Who cares. They all mean "the class that is not class 1.")

Outline

- Dichotomizers and Polychotomizers
  - Dichotomizer: what it is; how to train it
  - Polychotomizer: what it is; how to train it
- One-Hot Vectors: Training targets for the polychotomizer
- Softmax Function
  - A differentiable approximate argmax
  - How to differentiate the softmax
- Cross-Entropy
  - Cross-entropy = negative log probability of training labels
  - Derivative of cross-entropy w.r.t. network weights
- Putting it all together: a one-layer softmax neural net

OK, now we know what the polychotomizer should compute. How do we compute it?

Now you know that

- y_{ij} = reference label = True value of $P(class\ j\ |\vec{f}_i)$, given to you with the training database.
- $\hat{y}_{ij}$ = hypothesis = value of $P(class\ j\ |\vec{f}_i)$ estimated by the neural net.

How can we do that estimation?

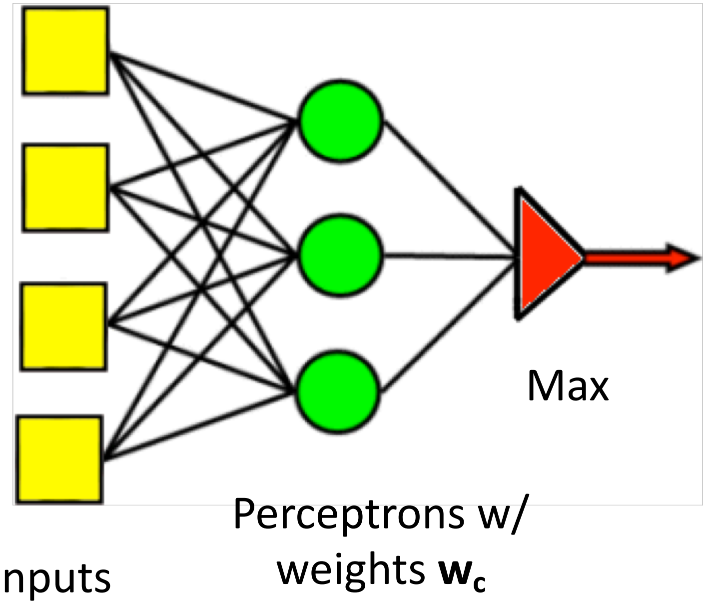
OK, now we know what the polychotomizer should compute. How do we compute it?

$\hat{y}_{ij}$ = value of $P(class\ j\ |\vec{f}_i)$ estimated by the neural net.

How can we do that estimation?

Multi-class perceptron example:

$$\hat{y}_{ij} = \begin{cases} 1 & \text{if } j = \underset{1\le\ell\le c}{\operatorname{argmax}} \vec{w}_\ell \cdot \vec{f}_i \\ 0 & \text{otherwise} \end{cases}$$



Differentiable perceptron: we need a differentiable approximation of the argmax function.

Softmax = differentiable approximation of the argmax function

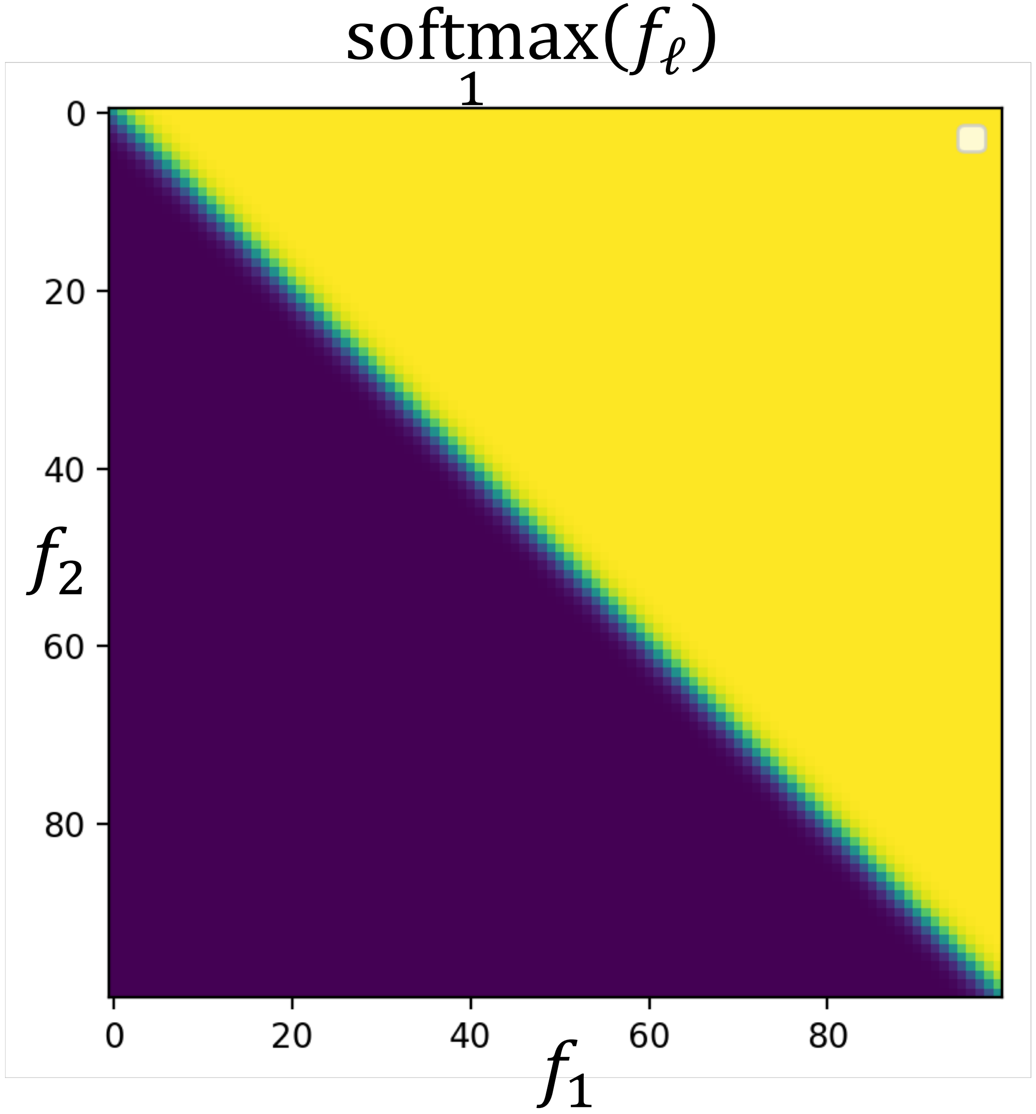
The softmax function is defined as:

$$\hat{y}_{ij} = \underset{j}{\text{softmax}}\left(\vec{w}_\ell \cdot \vec{f}_i\right) = \frac{e^{\vec{w}_j \cdot \vec{f}_i}}{\sum_{\ell=1}^{c} e^{\vec{w}_\ell \cdot \vec{f}_i}}$$

For example, the figure to the right shows

$$\hat{y}_1 = \underset{1}{\text{softmax}}(f_\ell) = \frac{e^{f_1}}{\sum_{\ell=1}^{2} e^{f_\ell}}$$

Notice that it's close to 1 (yellow) when $f_1 = \max f_\ell$, and close to zero (blue) otherwise, with a smooth transition zone in between.



Softmax = differentiable approximation of the argmax function

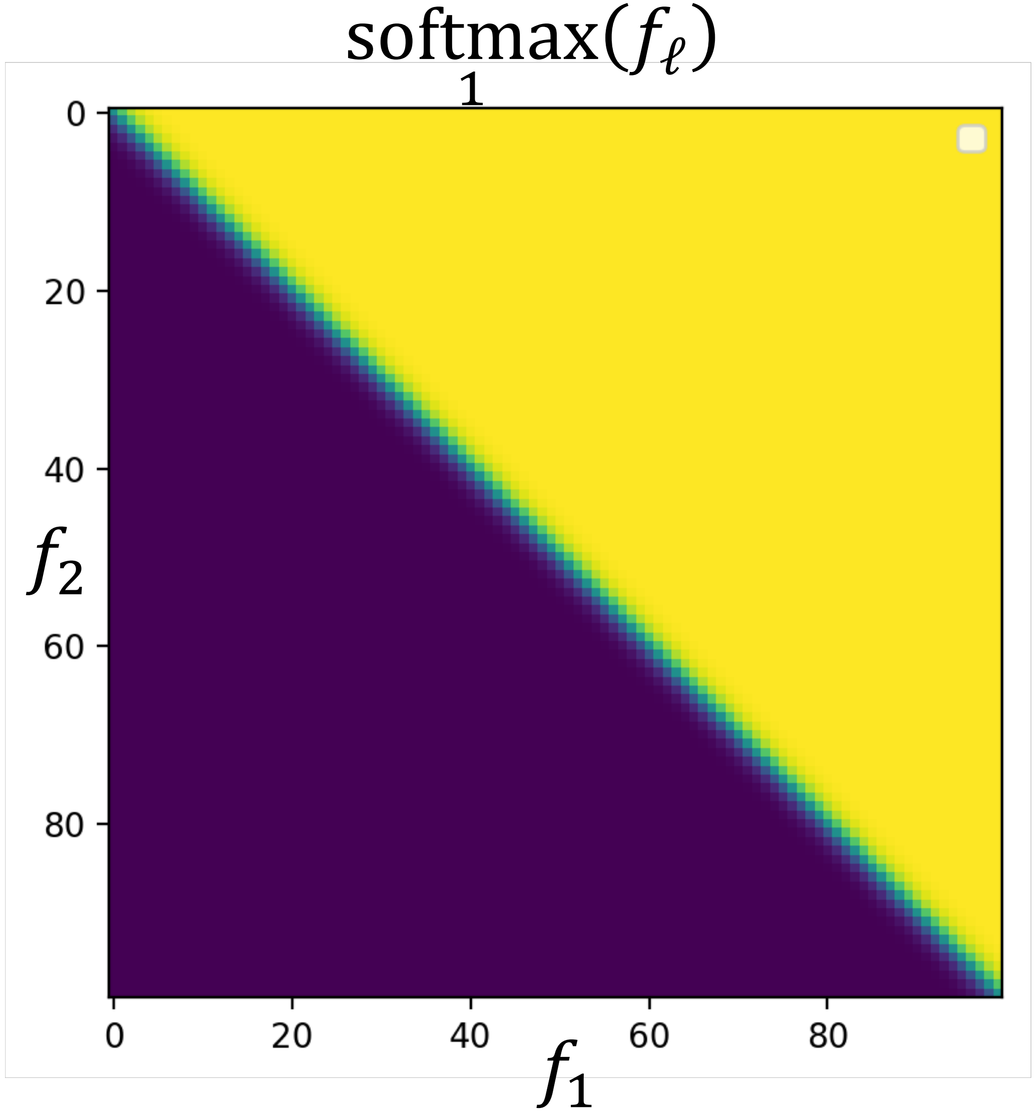
The softmax function is defined as:

$$\hat{y}_{ij} = \underset{j}{\text{softmax}}\left(\vec{w}_\ell \cdot \vec{f}_i\right) = \frac{e^{\vec{w}_j \cdot \vec{f}_i}}{\sum_{\ell=1}^{c} e^{\vec{w}_\ell \cdot \vec{f}_i}}$$

Notice that this gives us

$$0 \leq \hat{y}_{ij} \leq 1, \qquad \sum_{j=1}^{c} \hat{y}_{ij} = 1$$

Therefore we can interpret $\hat{y}_{ij}$ as an estimate of $P(class\ j\ |\vec{f}_i)$.



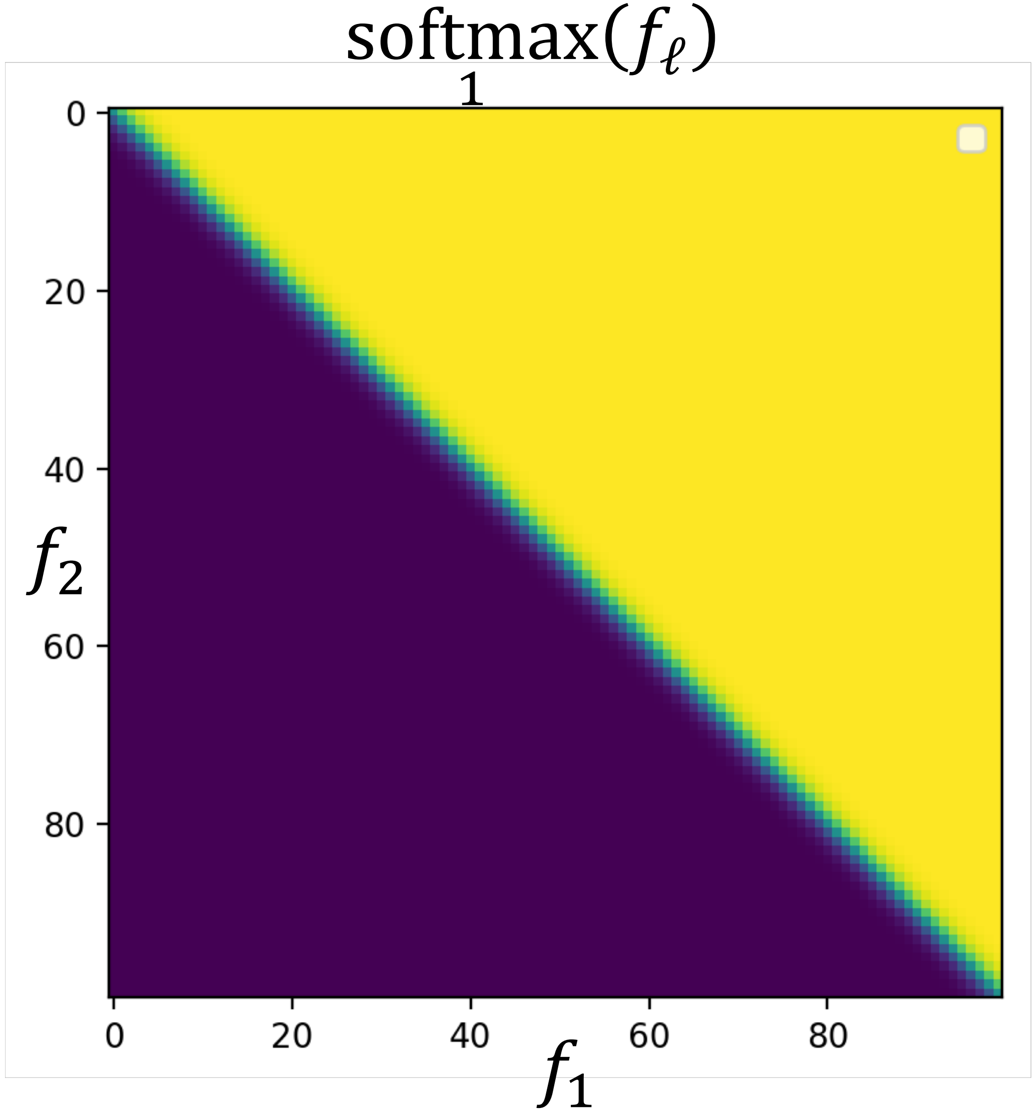
Outline

- Dichotomizers and Polychotomizers
  - Dichotomizer: what it is; how to train it
  - Polychotomizer: what it is; how to train it
- One-Hot Vectors: Training targets for the polychotomizer
- Softmax Function
  - A differentiable approximate argmax
  - How to differentiate the softmax
- Cross-Entropy
  - Cross-entropy = negative log probability of training labels
  - Derivative of cross-entropy w.r.t. network weights
- Putting it all together: a one-layer softmax neural net

How to differentiate the softmax: 3 steps

Unlike argmax, the softmax function is differentiable. All we need is the chain rule, plus three rules from calculus:

1. $\frac{\partial}{\partial w}\left(\frac{a}{b}\right) = \left(\frac{1}{b}\right)\frac{\partial a}{\partial w} - \left(\frac{a}{b^2}\right)\frac{\partial b}{\partial w}$
2. $\frac{\partial}{\partial w}(e^a) = (e^a)\frac{\partial a}{\partial w}$
3. $\frac{\partial}{\partial w}(wf) = f$



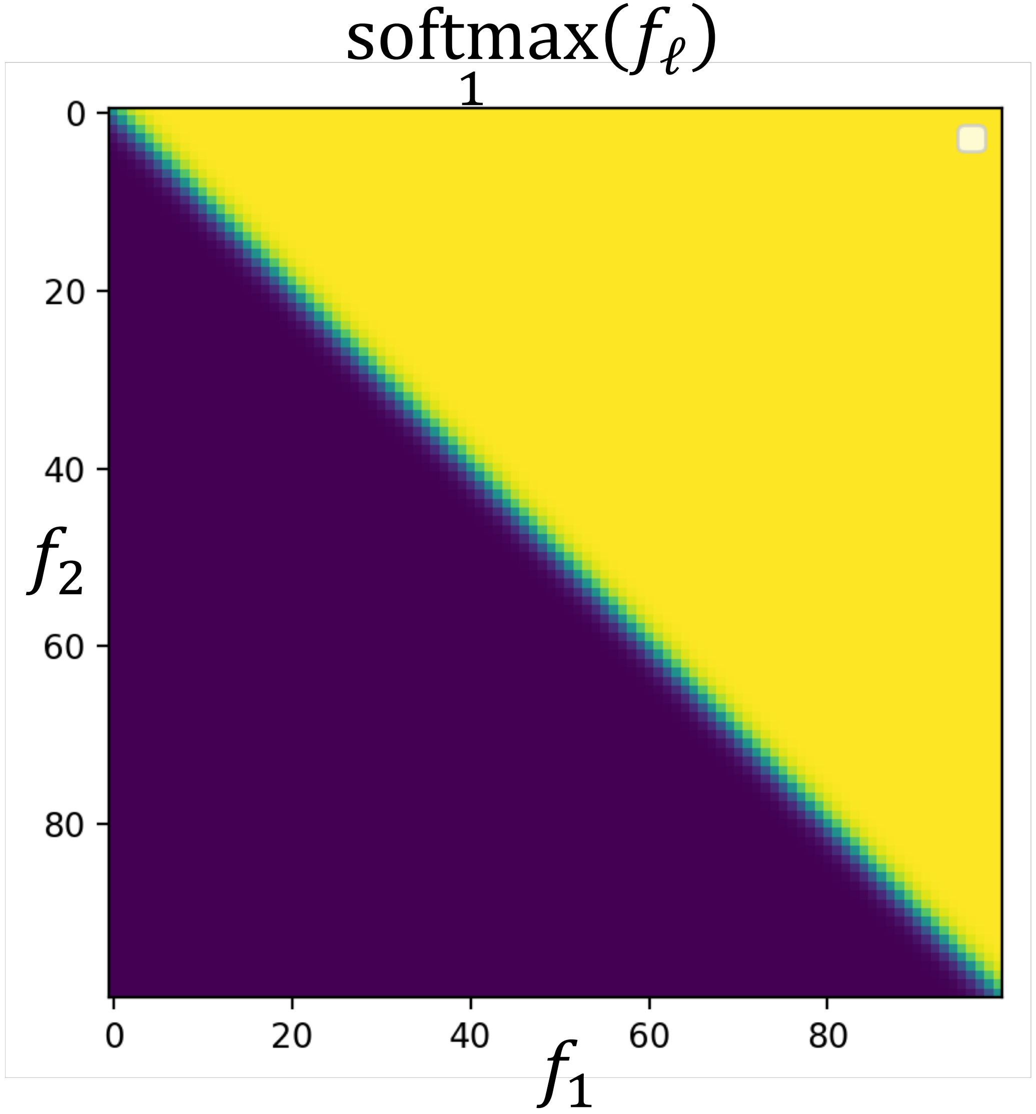
How to differentiate the softmax: step 1

First, we use the rule for $\frac{\partial}{\partial w}\left(\frac{a}{b}\right) = \left(\frac{1}{b}\right)\frac{\partial a}{\partial w} - \left(\frac{a}{b^2}\right)\frac{\partial b}{\partial w}$:

$$\hat{y}_{ij} = \underset{j}{\text{softmax}}\left(\vec{w}_\ell \cdot \vec{f}_i\right) = \frac{e^{\vec{w}_j\cdot\vec{f}_i}}{\sum_{\ell=1}^{c} e^{\vec{w}_\ell\cdot\vec{f}_i}}$$

$$\frac{\partial \hat{y}_{ij}}{\partial w_{mk}} = \left(\frac{1}{\sum_{\ell=1}^{c} e^{\vec{w}_\ell\cdot\vec{f}_i}}\right)\left(\frac{\partial e^{\vec{w}_j\cdot\vec{f}_i}}{\partial w_{mk}}\right) - \left(\frac{e^{\vec{w}_j\cdot\vec{f}_i}}{\left(\sum_{\ell=1}^{c} e^{\vec{w}_\ell\cdot\vec{f}_i}\right)^2}\right)\left(\frac{\partial\left(\sum_{\ell=1}^{c} e^{\vec{w}_\ell\cdot\vec{f}_i}\right)}{\partial w_{mk}}\right)$$

$$= \begin{cases} \left(\frac{1}{\sum_{\ell=1}^{c} e^{\vec{w}_\ell\cdot\vec{f}_i}}\right)\left(\frac{\partial e^{\vec{w}_j\cdot\vec{f}_i}}{\partial w_{mk}}\right) - \left(\frac{e^{\vec{w}_j\cdot\vec{f}_i}}{\left(\sum_{\ell=1}^{c} e^{\vec{w}_\ell\cdot\vec{f}_i}\right)^2}\right)\left(\frac{\partial\left(\sum_{\ell=1}^{c} e^{\vec{w}_\ell\cdot\vec{f}_i}\right)}{\partial w_{mk}}\right) & m = j \\ -\left(\frac{e^{\vec{w}_j\cdot\vec{f}_i}}{\left(\sum_{\ell=1}^{c} e^{\vec{w}_\ell\cdot\vec{f}_i}\right)^2}\right)\left(\frac{\partial\left(\sum_{\ell=1}^{c} e^{\vec{w}_\ell\cdot\vec{f}_i}\right)}{\partial w_{mk}}\right) & m \neq j \end{cases}$$



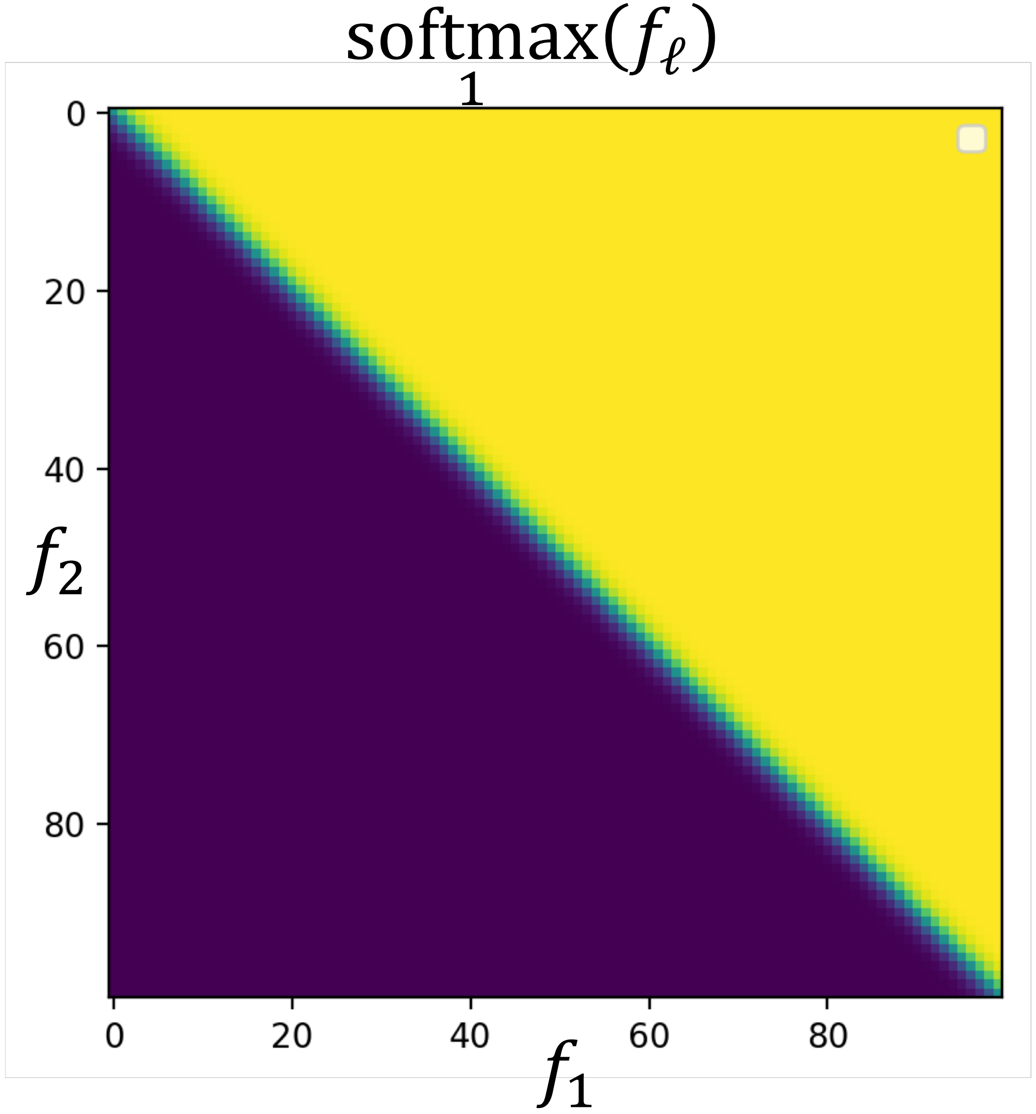
How to differentiate the softmax: step 2

Next, we use the rule $\frac{\partial}{\partial w}(e^a) = (e^a)\frac{\partial a}{\partial w}$:

$$\frac{\partial \hat{y}_{ij}}{\partial w_{mk}} =$$

$$\begin{cases} \left(\frac{1}{\sum_{\ell=1}^{c} e^{\vec{w}_\ell\cdot\vec{f}_i}}\right)\left(\frac{\partial e^{\vec{w}_j\cdot\vec{f}_i}}{\partial w_{mk}}\right) - \left(\frac{e^{\vec{w}_j\cdot\vec{f}_i}}{\left(\sum_{\ell=1}^{c} e^{\vec{w}_\ell\cdot\vec{f}_i}\right)^2}\right)\left(\frac{\partial\left(\sum_{\ell=1}^{c} e^{\vec{w}_\ell\cdot\vec{f}_i}\right)}{\partial w_{mk}}\right) & m = j \\ -\left(\frac{e^{\vec{w}_j\cdot\vec{f}_i}}{\left(\sum_{\ell=1}^{c} e^{\vec{w}_\ell\cdot\vec{f}_i}\right)^2}\right)\left(\frac{\partial\left(\sum_{\ell=1}^{c} e^{\vec{w}_\ell\cdot\vec{f}_i}\right)}{\partial w_{mk}}\right) & m \neq j \end{cases}$$

$$= \begin{cases} \left(\frac{e^{\vec{w}_j\cdot\vec{f}_i}}{\sum_{\ell=1}^{c} e^{\vec{w}_\ell\cdot\vec{f}_i}} - \frac{\left(e^{\vec{w}_j\cdot\vec{f}_i}\right)^2}{\left(\sum_{\ell=1}^{c} e^{\vec{w}_\ell\cdot\vec{f}_i}\right)^2}\right)\left(\frac{\partial(\vec{w}_m\cdot\vec{f}_i)}{\partial w_{mk}}\right) & m = j \\ \left(-\frac{e^{\vec{w}_j\cdot\vec{f}_i}e^{\vec{w}_m\cdot\vec{f}_i}}{\left(\sum_{\ell=1}^{c} e^{\vec{w}_\ell\cdot\vec{f}_i}\right)^2}\right)\left(\frac{\partial(\vec{w}_m\cdot\vec{f}_i)}{\partial w_{mk}}\right) & m \neq j \end{cases}$$

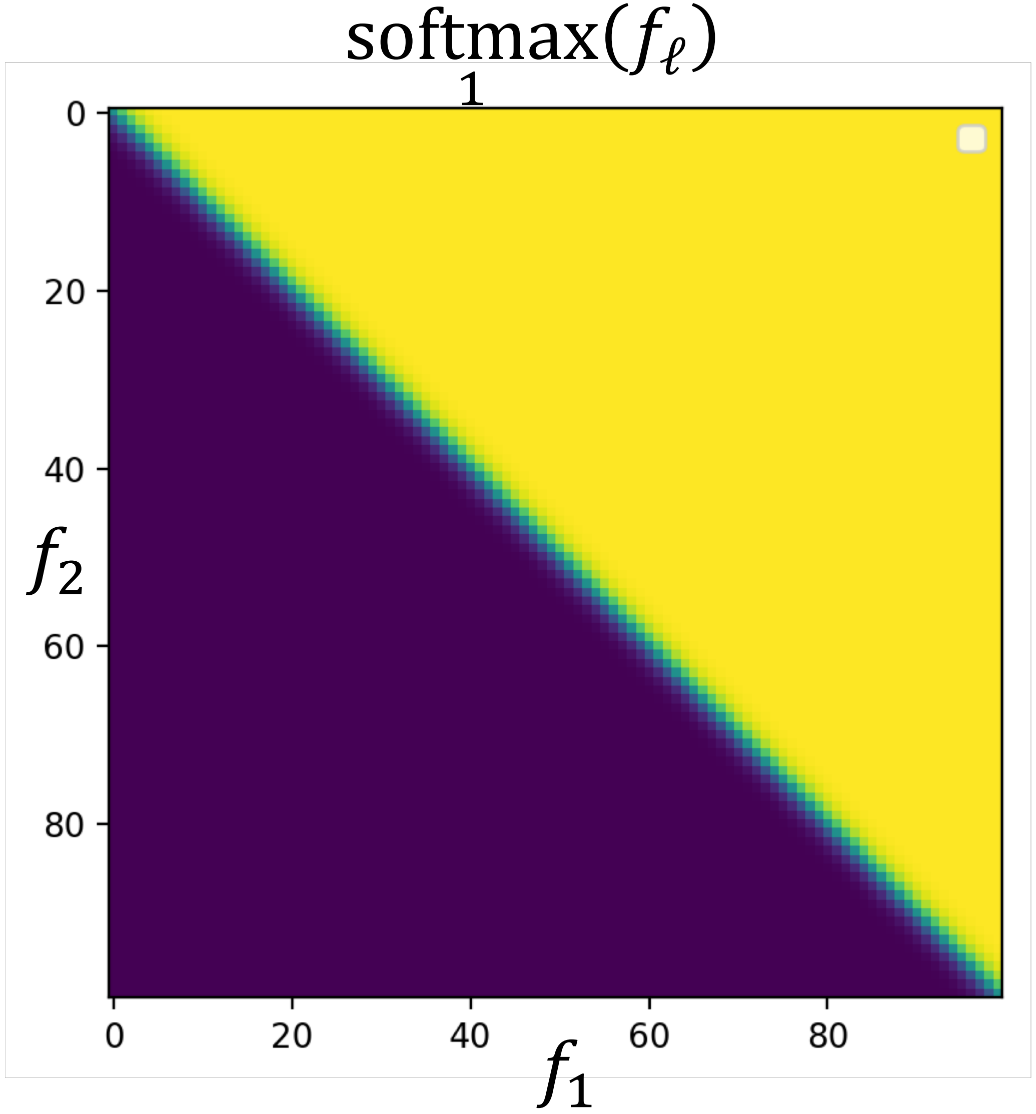


How to differentiate the softmax: step 3

Next, we use the rule $\frac{\partial}{\partial w}(wf) = f$:

$$\frac{\partial \hat{y}_{ij}}{\partial w_{mk}} = \begin{cases} \left(\frac{e^{\vec{w}_j\cdot\vec{f}_i}}{\sum_{\ell=1}^{c} e^{\vec{w}_\ell\cdot\vec{f}_i}} - \frac{\left(e^{\vec{w}_j\cdot\vec{f}_i}\right)^2}{\left(\sum_{\ell=1}^{c} e^{\vec{w}_\ell\cdot\vec{f}_i}\right)^2}\right)\left(\frac{\partial(\vec{w}_m\cdot\vec{f}_i)}{\partial w_{mk}}\right) & m = j \\ \left(-\frac{e^{\vec{w}_j\cdot\vec{f}_i}e^{\vec{w}_m\cdot\vec{f}_i}}{\left(\sum_{\ell=1}^{c} e^{\vec{w}_\ell\cdot\vec{f}_i}\right)^2}\right)\left(\frac{\partial(\vec{w}_m\cdot\vec{f}_i)}{\partial w_{mk}}\right) & m \neq j \end{cases}$$

$$= \begin{cases} \left(\frac{e^{\vec{w}_j\cdot\vec{f}_i}}{\sum_{\ell=1}^{c} e^{\vec{w}_\ell\cdot\vec{f}_i}} - \frac{\left(e^{\vec{w}_j\cdot\vec{f}_i}\right)^2}{\left(\sum_{\ell=1}^{c} e^{\vec{w}_\ell\cdot\vec{f}_i}\right)^2}\right) f_{ik} & m = j \\ \left(-\frac{e^{\vec{w}_j\cdot\vec{f}_i}e^{\vec{w}_m\cdot\vec{f}_i}}{\left(\sum_{\ell=1}^{c} e^{\vec{w}_\ell\cdot\vec{f}_i}\right)^2}\right) f_{ik} & m \neq j \end{cases}$$

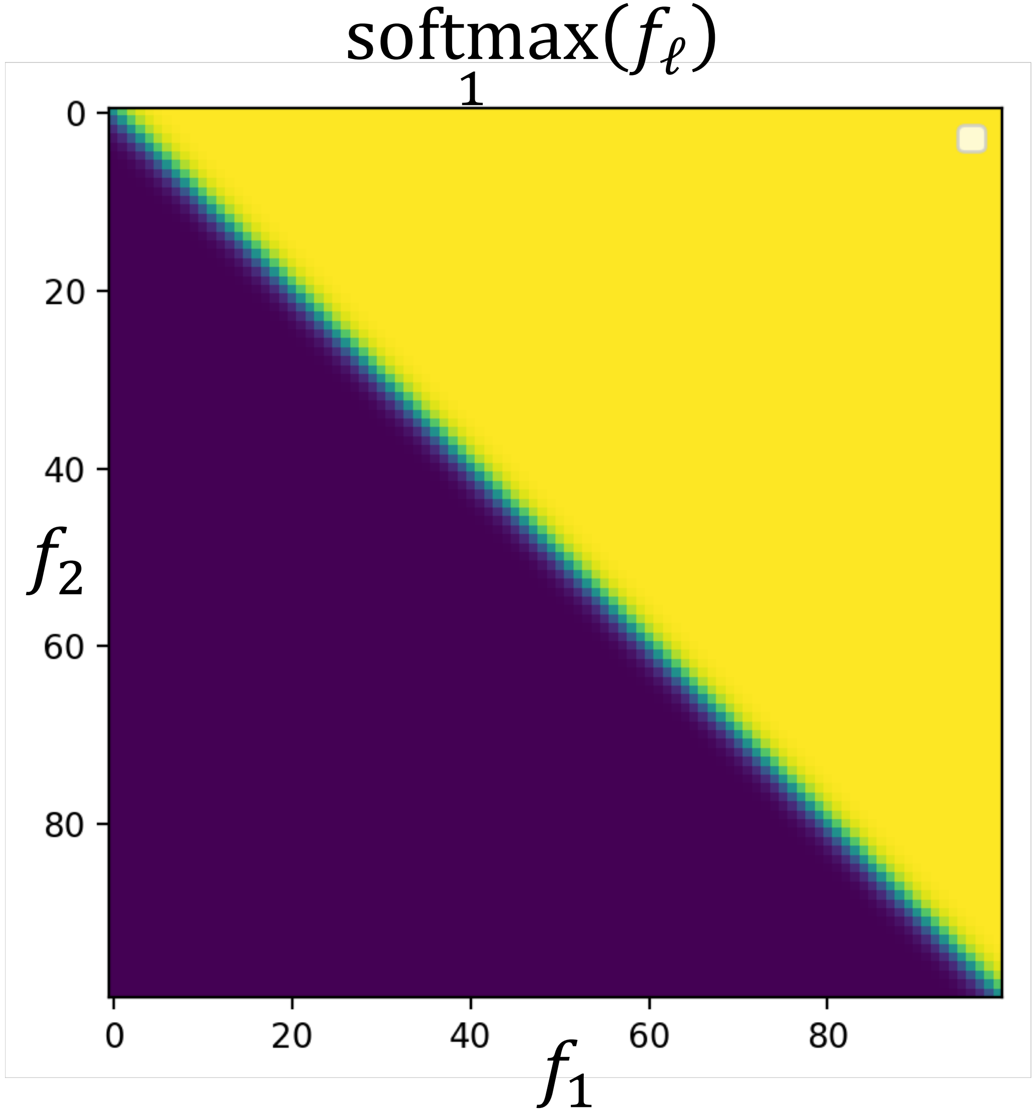


Differentiating the softmax

… and, simplify.

$$\frac{\partial \hat{y}_{ij}}{\partial w_{mk}} = \begin{cases} \left(\frac{e^{\vec{w}_j \cdot \vec{f}_i}}{\sum_{\ell=1}^{c} e^{\vec{w}_\ell \cdot \vec{f}_i}} - \frac{\left(e^{\vec{w}_j \cdot \vec{f}_i}\right)^2}{\left(\sum_{\ell=1}^{c} e^{\vec{w}_\ell \cdot \vec{f}_i}\right)^2} \right) f_{ik} & m = j \\ \left(-\frac{e^{\vec{w}_j \cdot \vec{f}_i} e^{\vec{w}_m \cdot \vec{f}_i}}{\left(\sum_{\ell=1}^{c} e^{\vec{w}_\ell \cdot \vec{f}_i}\right)^2} \right) f_{ik} & m \neq j \end{cases}$$

$$\frac{\partial \hat{y}_{ij}}{\partial w_{mk}} = \begin{cases} \left(\hat{y}_{ij} - \hat{y}_{ij}{}^2\right) f_{ik} & m = j \\ -\hat{y}_{ij} \hat{y}_{im} f_{ik} & m \neq j \end{cases}$$



Recap: how to differentiate the softmax

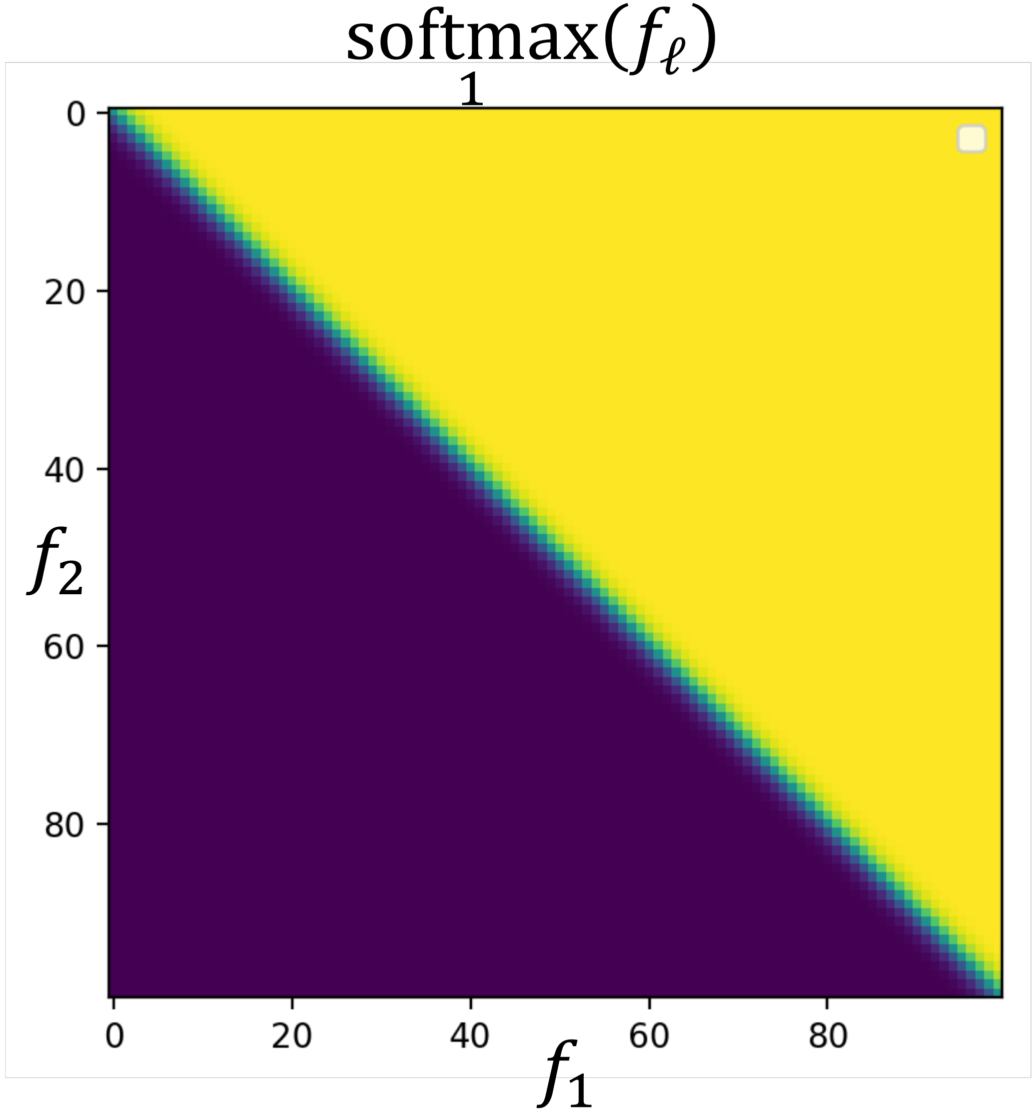
- $\hat{y}_{ij}$ is the probability of the j^{th} class, estimated by the neural net, in response to the i^{th} training token
- w_{mk} is the network weight that connects the k^{th} input feature to the m^{th} class label

The dependence of $\hat{y}_{ij}$ on w_{mk} for $m \neq j$ is weird, and people who are learning this for the first time often forget about it. It comes from the denominator of the softmax.

$$\hat{y}_{ij} = \underset{j}{\text{softmax}}\left(\vec{w}_\ell \cdot \vec{f}_i\right) = \frac{e^{\vec{w}_j \cdot \vec{f}_i}}{\sum_{\ell=1}^{c} e^{\vec{w}_\ell \cdot \vec{f}_i}}$$

$$\frac{\partial \hat{y}_{ij}}{\partial w_{mk}} = \begin{cases} \left(\hat{y}_{ij} - \hat{y}_{ij}^{\,2}\right) f_{ik} & m = j \\ -\hat{y}_{ij}\hat{y}_{im} f_{ik} & m \neq j \end{cases}$$

- $\hat{y}_{im}$ is the probability of the m^{th} class for the i^{th} training token
- f_{ik} is the value of the k^{th} input feature for the i^{th} training token



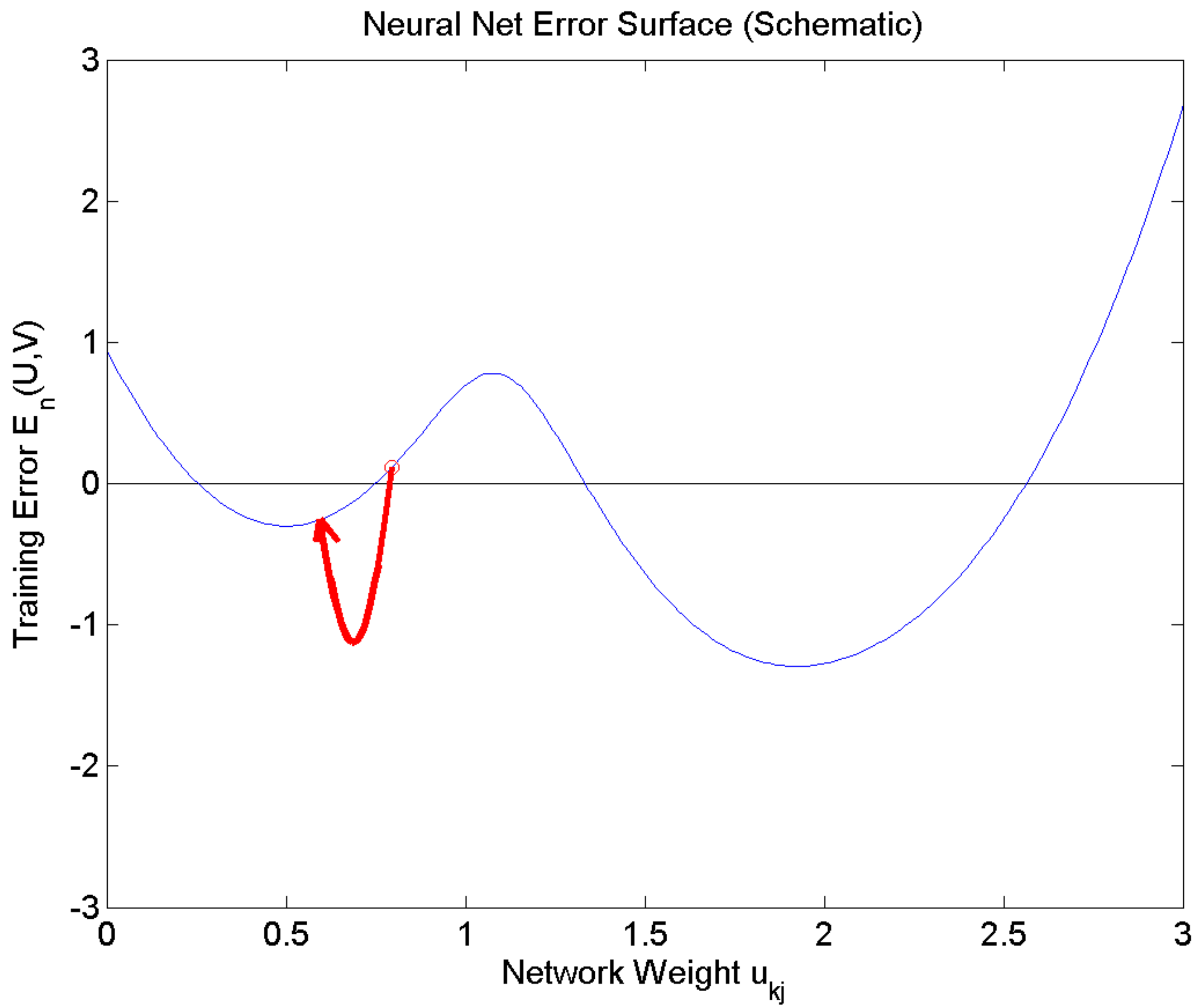
Outline

- Dichotomizers and Polychotomizers
  - Dichotomizer: what it is; how to train it
  - Polychotomizer: what it is; how to train it
- One-Hot Vectors: Training targets for the polychotomizer
- Softmax Function: A differentiable approximate argmax
- Cross-Entropy
  - Cross-entropy = negative log probability of training labels
  - Derivative of cross-entropy w.r.t. network weights
- Putting it all together: a one-layer softmax neural net

Training a Softmax Neural Network

All of that differentiation is useful because we want to train the neural network to represent a training database as well as possible. If we can define the training error to be some function L, then we want to update the weights according to

$$w_{mk} = w_{mk} - \eta \frac{\partial L}{\partial w_{mk}}$$



So what is L?

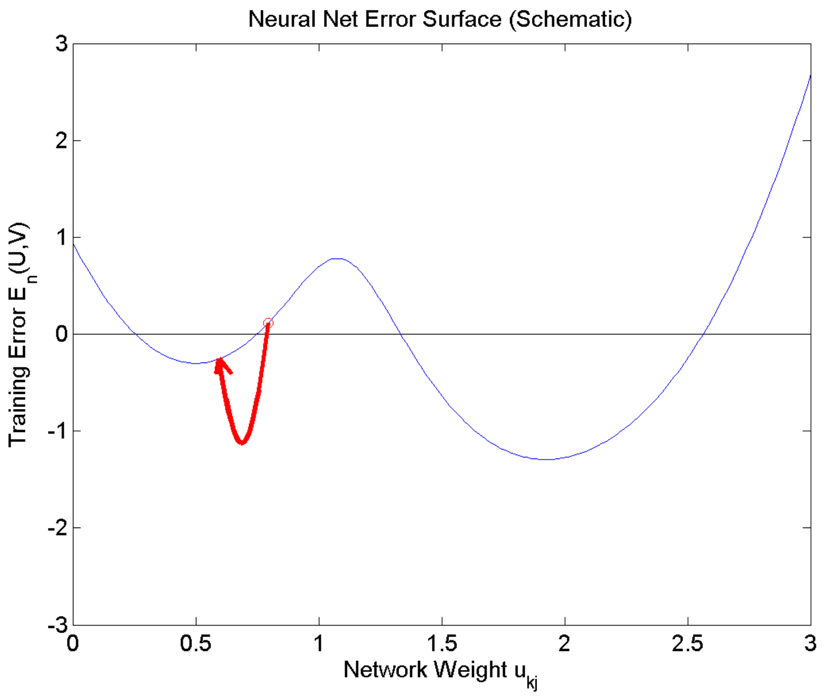
Training: Maximize the probability of the training data

Remember, the whole point of that denominator in the softmax function is that it allows us to use softmax as

$$\hat{y}_{ij} = \text{Estimated value of } P(\text{class } j \mid \vec{f}_i)$$

Suppose we decide to estimate the network weights w_{mk} in order to maximize the probability of the training database, in the sense of

$$w_{mk} = \underset{w}{\operatorname{argmax}}\, P(\text{training labels} \mid \text{training feature vectors})$$



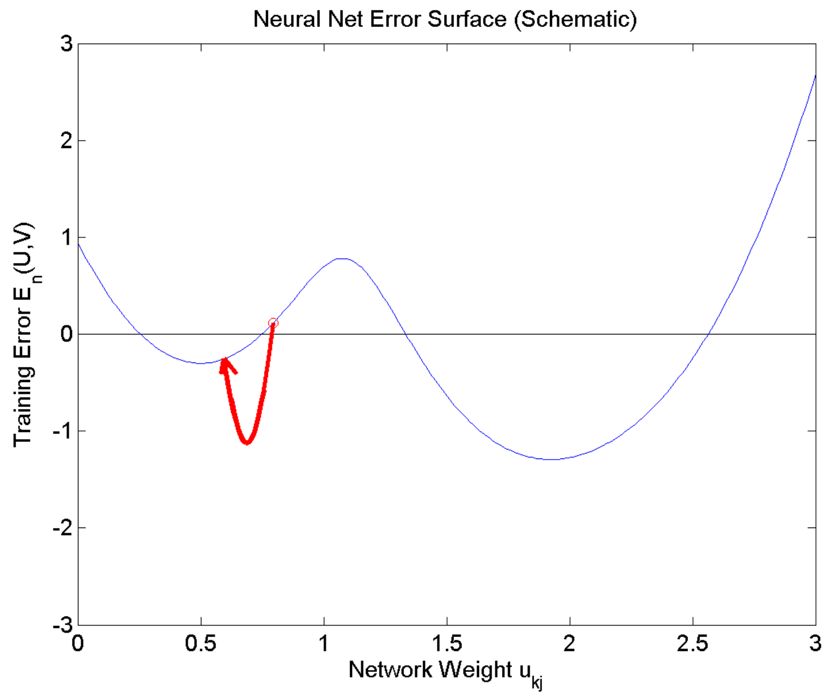
Training: Maximize the probability of the training data

Remember, the whole point of that denominator in the softmax function is that it allows us to use softmax as

$$\hat{y}_{ij} = \text{ Estimated value of } P(\text{class } j \mid \vec{f}_i)$$

If we assume the training tokens are independent, this is:

$$w_{mk} = \underset{w}{\text{argmax}} \prod_{i=1}^{n} P(\text{reference label of the } i^{th} \text{token} \mid i^{th} \text{feature vector})$$



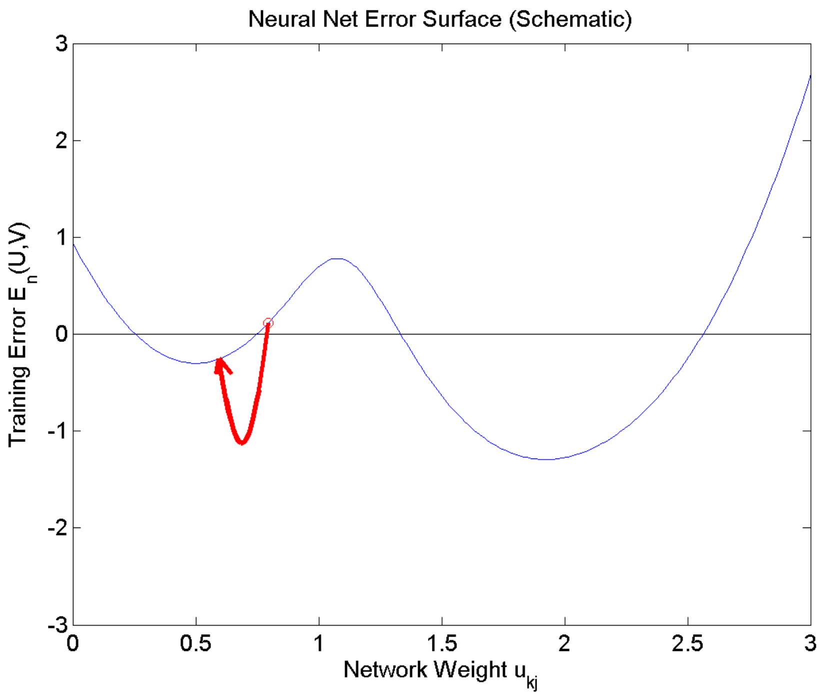
Training: Maximize the probability of the training data

Remember, the whole point of that denominator in the softmax function is that it allows us to use softmax as

$$\hat{y}_{ij} = \text{ Estimated value of } P(\text{class } j \mid \vec{f}_i)$$

OK. We need to create some notation to mean "the reference label for the i^{th} token." Let's call it $j(i)$.

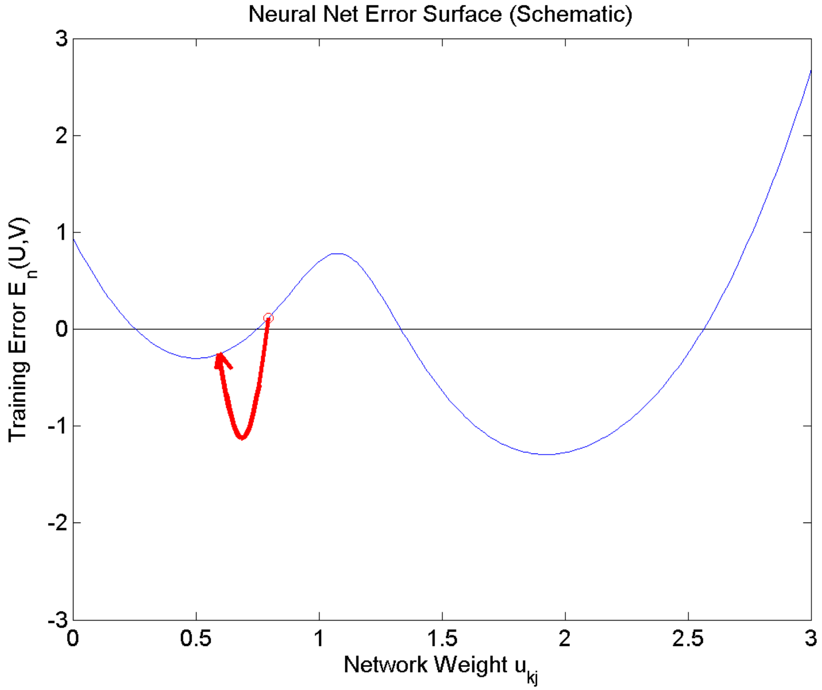
$$w_{mk} = \underset{w}{\text{argmax}} \prod_{i=1}^{n} P(\text{class } j(i) \mid \vec{f})$$



Training: Maximize the probability of the training data

Wow, Cool!! So we can maximize the probability of the training data by just picking the softmax output corresponding to the **correct class** $j(i)$, for each token, and then multiplying them all together:

$$w_{mk} = \underset{w}{\operatorname{argmax}} \prod_{i=1}^{n} \hat{y}_{i,j(i)}$$



So, hey, let's take the logarithm, to get rid of that nasty product operation.

$$w_{mk} = \underset{w}{\operatorname{argmax}} \sum_{i=1}^{n} \ln \hat{y}_{i,j(i)}$$

Training: Minimizing the negative log probability

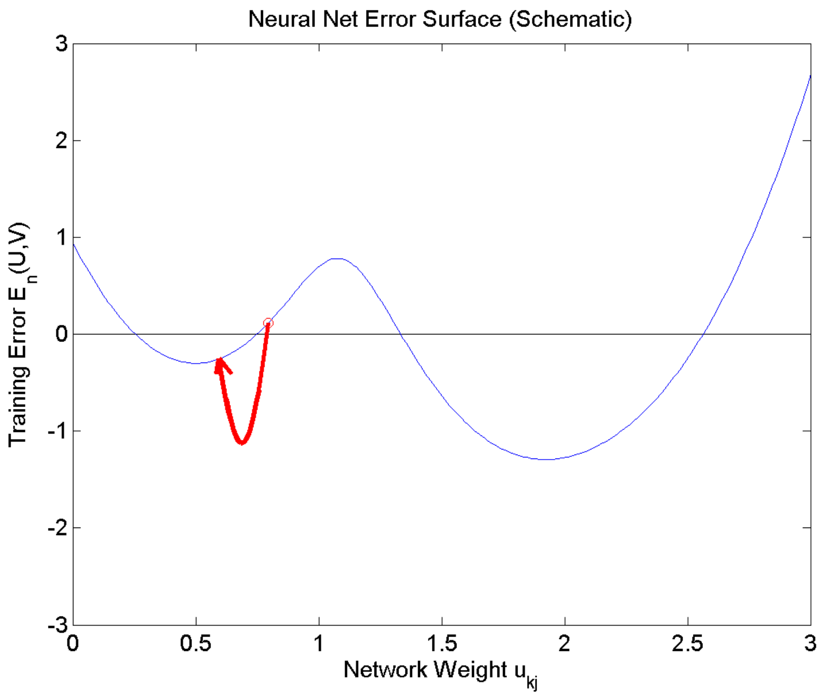
So, to maximize the probability of the training data given the model, we need:

$$w_{mk} = \underset{w}{\operatorname{argmax}} \sum_{i=1}^{n} \ln \hat{y}_{i,j(i)}$$

If we just multiply by (-1), that will turn the max into a min. It's kind of a stupid thing to do---who cares whether you're minimizing L or maximizing $-L$, same thing, right? But it's standard, so what the heck.

$$w_{mk} = \underset{w}{\operatorname{argmin}}\, L$$

$$L = \sum_{i=1}^{n} -\ln \hat{y}_{i,j(i)}$$



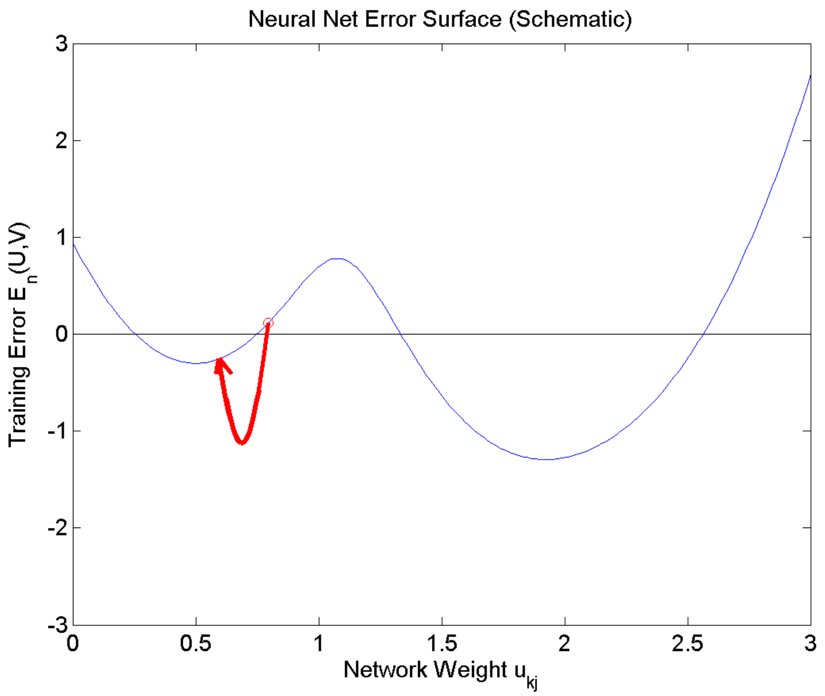
Training: Minimizing the negative log probability

Softmax neural networks are almost always trained in order to minimize the negative log probability of the training data:

$$w_{mk} = \underset{w}{\operatorname{argmin}}\, L$$

$$L = \sum_{i=1}^{n} -\ln \hat{y}_{i,j(i)}$$

This loss function, defined above, is called the **cross-entropy loss**. The reasons for that name are very cool, and very far beyond the scope of this course. Take CS 446 (Machine Learning) and/or ECE 563 (Information Theory) to learn more.



Outline

- Dichotomizers and Polychotomizers
  - Dichotomizer: what it is; how to train it
  - Polychotomizer: what it is; how to train it
- One-Hot Vectors: Training targets for the polychotomizer
- Softmax Function: A differentiable approximate argmax
- Cross-Entropy
  - Cross-entropy = negative log probability of training labels
  - Derivative of cross-entropy w.r.t. network weights
- Putting it all together: a one-layer softmax neural net

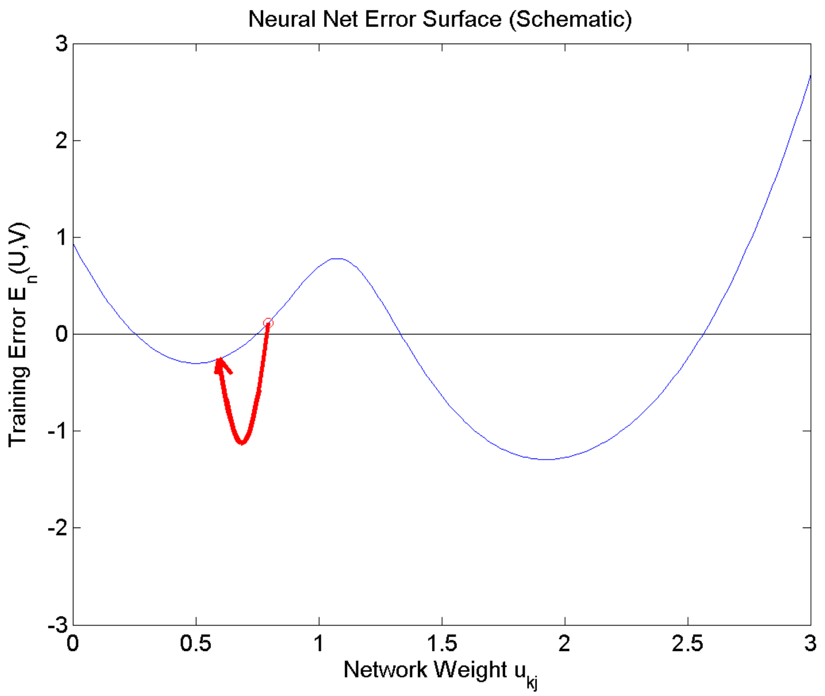
Differentiating the cross-entropy

The cross-entropy loss function is:

$$L = \sum_{i=1}^{n} -\ln \hat{y}_{i,j(i)}$$

Let's try to differentiate it:

$$\frac{\partial L}{\partial w_{mk}} = \sum_{i=1}^{n} -\left(\frac{1}{\hat{y}_{i,j(i)}}\right)\frac{\partial \hat{y}_{i,j(i)}}{\partial w_{mk}}$$



Differentiating the cross-entropy

The cross-entropy loss function is:

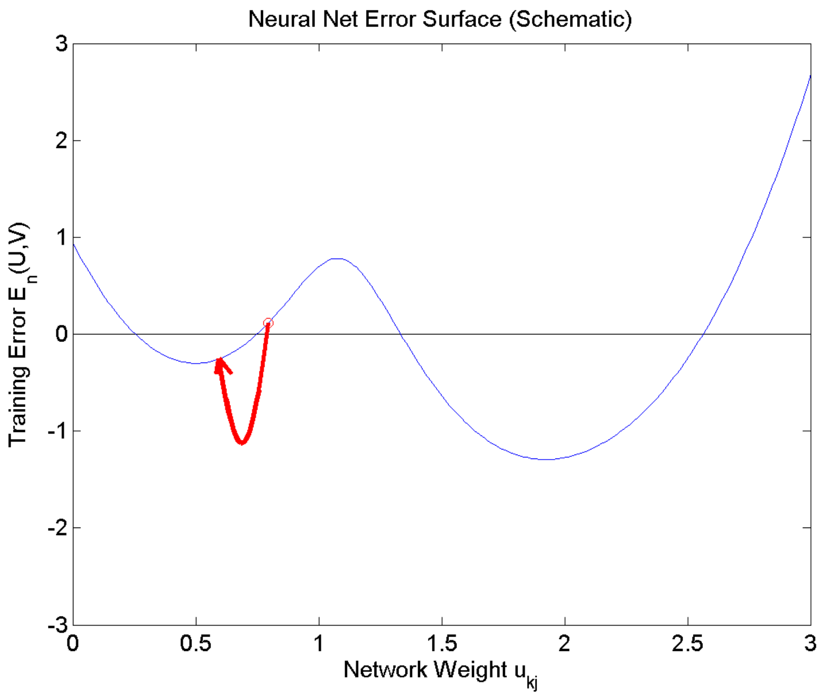
$$L = \sum_{i=1}^{n} -\ln \hat{y}_{i,j(i)}$$

Let's try to differentiate it:

$$\frac{\partial L}{\partial w_{mk}} = \sum_{i=1}^{n} -\left(\frac{1}{\hat{y}_{i,j(i)}}\right)\frac{\partial \hat{y}_{i,j(i)}}{\partial w_{mk}}$$

…and then…

$$\left(\frac{1}{\hat{y}_{i,j(i)}}\right)\frac{\partial \hat{y}_{i,j(i)}}{\partial w_{mk}} = \begin{cases}(1-\hat{y}_{im})f_{ik} & m = j(i)\\ -\hat{y}_{im}f_{ik} & m \neq j(i)\end{cases}$$



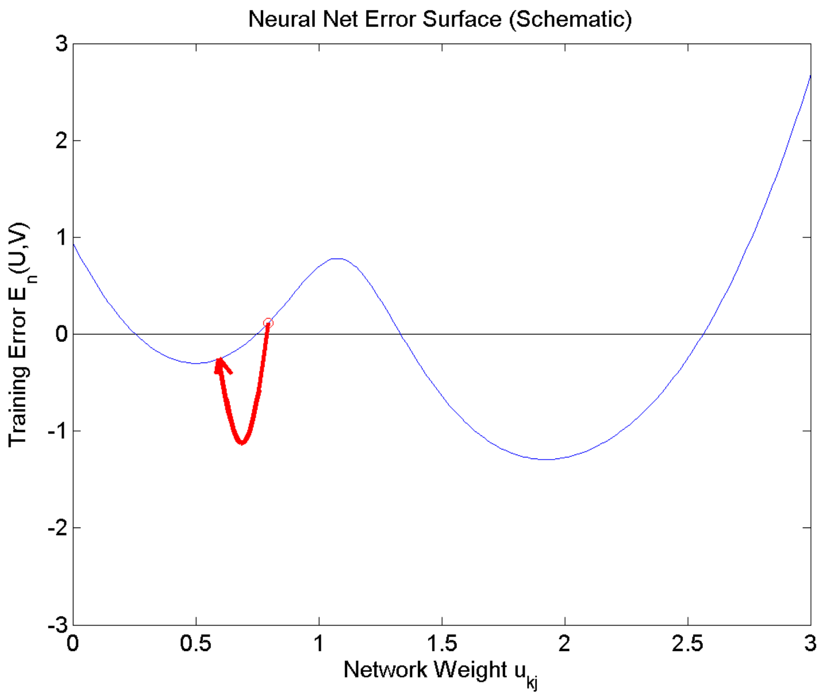
Differentiating the cross-entropy

Let's try to differentiate it:

$$\frac{\partial L}{\partial w_{mk}} = \sum_{i=1}^{n} -\left(\frac{1}{\hat{y}_{i,j(i)}}\right)\frac{\partial \hat{y}_{i,j(i)}}{\partial w_{mk}}$$

…and then…

$$\left(\frac{1}{\hat{y}_{i,j(i)}}\right)\frac{\partial \hat{y}_{i,j(i)}}{\partial w_{mk}} = \begin{cases} (1-\hat{y}_{im})f_{ik} & m = j(i) \\ -\hat{y}_{im} f_{ik} & m \neq j(i) \end{cases}$$



… but remember our reference labels:

$$y_{ij} = \begin{cases} 1 & i^{th} \text{ example is from class j} \\ 0 & i^{th} \text{ example is NOT from class j} \end{cases}$$

Differentiating the cross-entropy

Let's try to differentiate it:

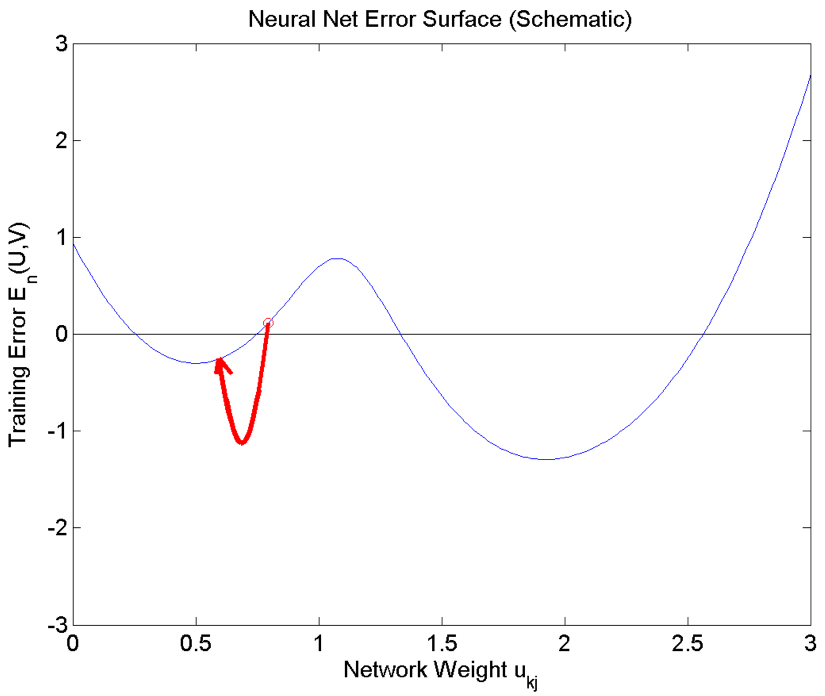
$$\frac{\partial L}{\partial w_{mk}} = \sum_{i=1}^{n} -\left(\frac{1}{\hat{y}_{i,j(i)}}\right)\frac{\partial \hat{y}_{i,j(i)}}{\partial w_{mk}}$$

…and then…

$$\left(\frac{1}{\hat{y}_{i,j(i)}}\right)\frac{\partial \hat{y}_{i,j(i)}}{\partial w_{mk}} = \begin{cases} (y_{im} - \hat{y}_{im})f_{ik} & m = j(i) \\ (y_{im} - \hat{y}_{im})f_{ik} & m \neq j(i) \end{cases}$$

… but remember our reference labels:

$$y_{ij} = \begin{cases} 1 & \text{i}^{\text{th}} \text{ example is from class j} \\ 0 & \text{i}^{\text{th}} \text{ example is NOT from class j} \end{cases}$$



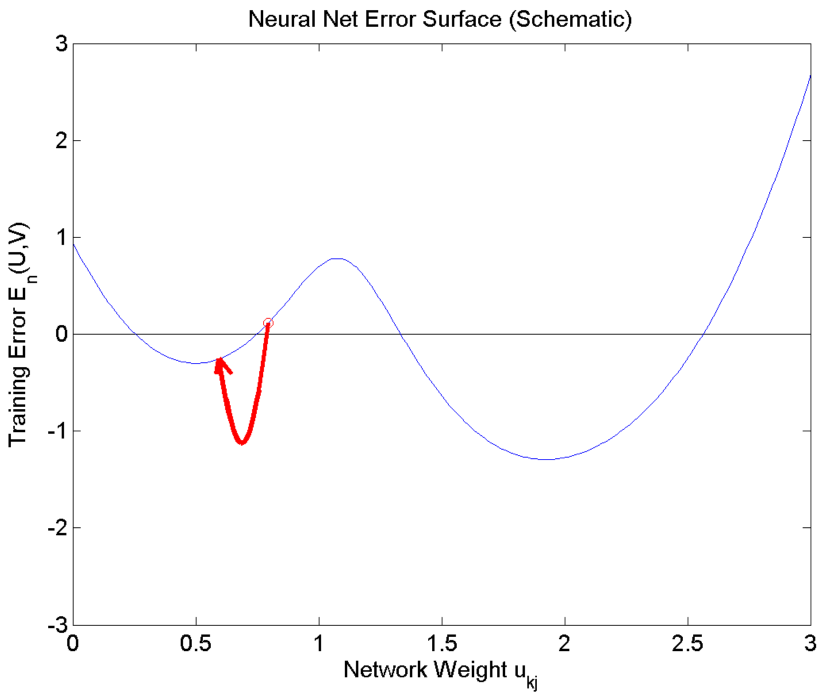
Differentiating the cross-entropy

Let's try to differentiate it:

$$\frac{\partial L}{\partial w_{mk}} = \sum_{i=1}^{n} -\left(\frac{1}{\hat{y}_{i,j(i)}}\right)\frac{\partial \hat{y}_{i,j(i)}}{\partial w_{mk}}$$

…and then…

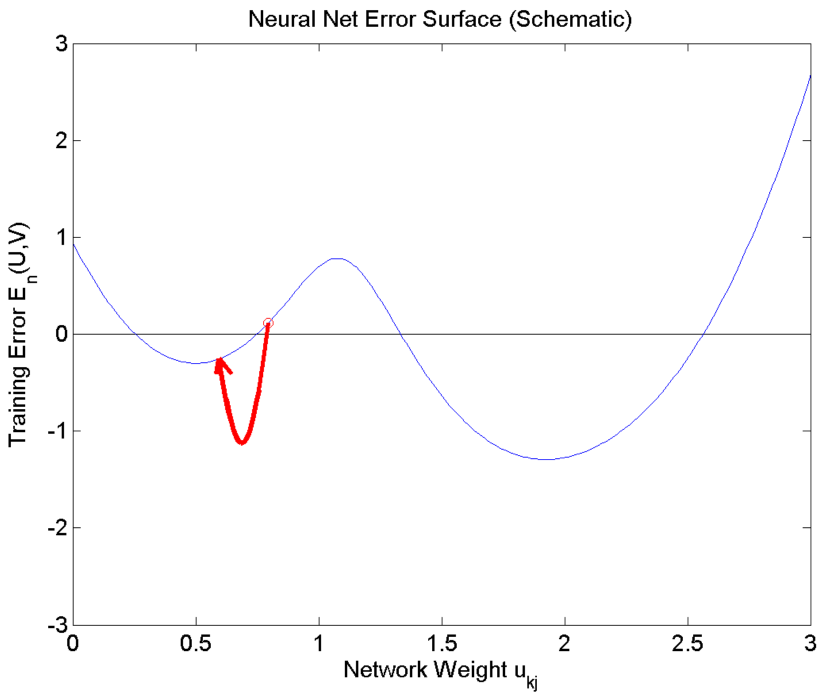
$$\left(\frac{1}{\hat{y}_{i,j(i)}}\right)\frac{\partial \hat{y}_{i,j(i)}}{\partial w_{mk}} = (y_{im} - \hat{y}_{im})f_{ik}$$



Differentiating the cross-entropy

Let's try to differentiate it:

$$\frac{\partial L}{\partial w_{mk}} = \sum_{i=1}^{n} (\hat{y}_{im} - y_{im}) f_{ik}$$



Differentiating the cross-entropy

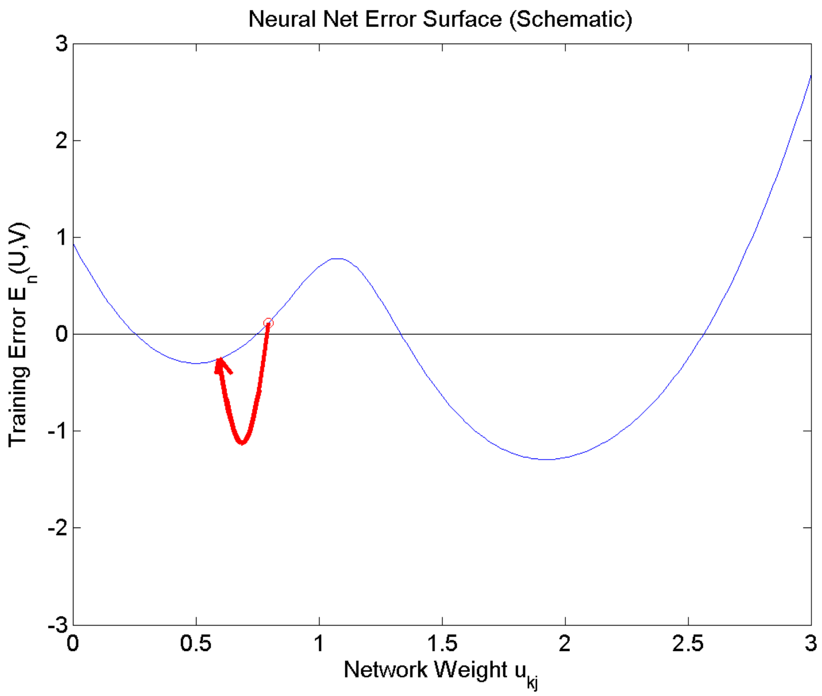
Let’s try to differentiate it:

$$\frac{\partial L}{\partial w_{mk}} = \sum_{i=1}^{n} (\hat{y}_{im} - y_{im}) f_{ik}$$

Interpretation:

Increasing w_{mk} will make the error worse if

- $\hat{y}_{im}$ is already too large, and f_{ik} is positive
- $\hat{y}_{im}$ is already too small, and f_{ik} is negative



Differentiating the cross-entropy

Let's try to differentiate it:

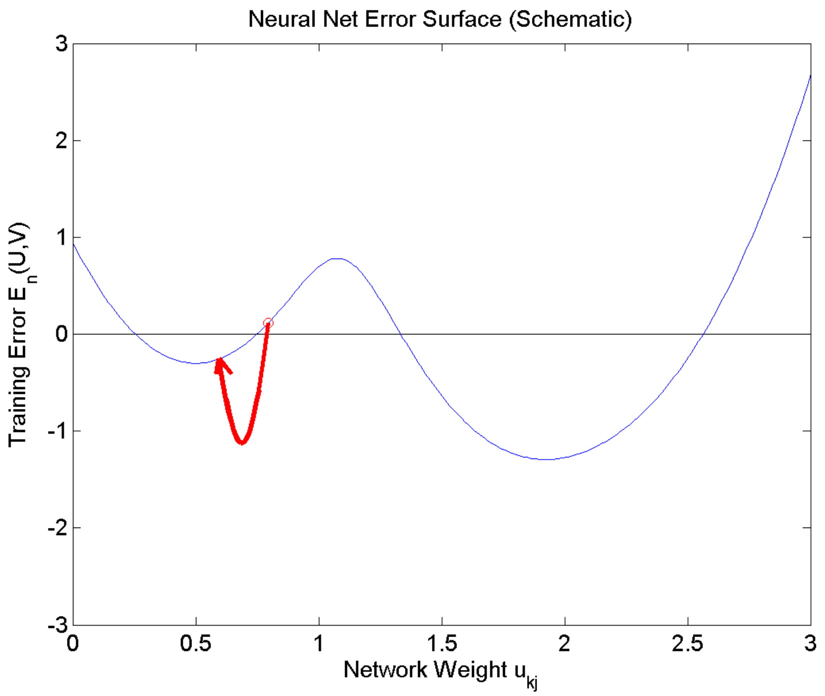
$$\frac{\partial L}{\partial w_{mk}} = \sum_{i=1}^{n} (\hat{y}_{im} - y_{im}) f_{ik}$$

Interpretation:

Our goal is to make the error as small as possible. So if

- $\hat{y}_{im}$ is already too large, then we want to make $w_{mk} f_{ik}$ smaller
- $\hat{y}_{im}$ is already too small , then we want to make $w_{mk} f_{ik}$ larger

$$w_{mk} = w_{mk} - \eta \frac{\partial L}{\partial w_{mk}}$$



Outline

- Dichotomizers and Polychotomizers
  - Dichotomizer: what it is; how to train it
  - Polychotomizer: what it is; how to train it
- One-Hot Vectors: Training targets for the polychotomizer
- Softmax Function: A differentiable approximate argmax
- Cross-Entropy
  - Cross-entropy = negative log probability of training labels
  - Derivative of cross-entropy w.r.t. network weights
- Putting it all together: a one-layer softmax neural net

Summary: Training Algorithms You Know

1. Naïve Bayes with Laplace Smoothing:

$$P(f_k = x|\text{class } j) = \frac{(\#\text{tokens of class } j \text{ with } f_k = x) + 1}{(\#\text{tokens of class } j) + (\#\text{possible values of } f_k)}$$

2. Multi-Class Perceptron: If token $\vec{f}_i$ of class j is misclassified as class m, then

$$\vec{w}_j = \vec{w}_j + \eta\vec{f}_i$$
$$\vec{w}_m = \vec{w}_m - \eta\vec{f}_i$$

3. Softmax Neural Net: for all weight vectors (correct or incorrect),

$$\vec{w}_m = \vec{w}_m - \eta\nabla_{\vec{w}_m}L$$
$$= \vec{w}_m - \eta(\hat{y}_{im} - y_{im})\vec{f}_i$$

Summary: Perceptron versus Softmax

Softmax Neural Net: for all weight vectors (correct or incorrect),

$$\vec{w}_m = \vec{w}_m - \eta(\hat{y}_{im} - y_{im})\vec{f}_i$$

Notice that, if the network were adjusted so that

$$\hat{y}_{im} = \begin{cases} 1 & \text{network thinks the correct class is } m \\ 0 & \text{otherwise} \end{cases}$$

Then we'd have

$$(\hat{y}_{im} - y_{im}) = \begin{cases} -2 & \text{correct class is } m, \text{but network is wrong} \\ 2 & \text{network guesses } m, \text{but it's wrong} \\ 0 & \text{otherwise} \end{cases}$$

Summary: Perceptron versus Softmax

Softmax Neural Net: for all weight vectors (correct or incorrect),

$$\vec{w}_m = \vec{w}_m - \eta(\hat{y}_{im} - y_{im})\vec{f}_i$$

Notice that, if the network were adjusted so that

$$\hat{y}_{im} = \begin{cases} 1 & \text{network thinks the correct class is } m \\ 0 & \text{otherwise} \end{cases}$$

Then we get the perceptron update rule back again (multiplied by 2, which doesn't matter):

$$\vec{w}_m = \begin{cases} \vec{w}_m + 2\eta\vec{f}_i & \text{correct class is } m\text{, but network is wrong} \\ \vec{w}_m - 2\eta\vec{f}_i & \text{network guesses } m\text{, but it's wrong} \\ \vec{w}_m & \text{otherwise} \end{cases}$$

Summary: Perceptron versus Softmax

So the key difference between perceptron and softmax is that, for a perceptron,

$$\hat{y}_{ij} = \begin{cases} 1 & \text{network thinks the correct class is } j \\ 0 & \text{otherwise} \end{cases}$$

Whereas, for a softmax,

$$0 \le \hat{y}_{ij} \le 1, \qquad \sum_{j=1}^{c} \hat{y}_{ij} = 1$$

Summary: Perceptron versus Softmax

…or, to put it another way, for a perceptron,

$$\hat{y}_{ij} = \begin{cases} 1 & \text{if } j = \underset{1\le\ell\le c}{\operatorname{argmax}} \vec{w}_\ell \cdot \vec{f}_i \\ 0 & \text{otherwise} \end{cases}$$

Whereas, for a softmax network,

$$\hat{y}_{ij} = \underset{j}{\operatorname{softmax}}\left(\vec{w}_\ell \cdot \vec{f}_i\right)$$