Lecture 18: Wrapping up classification

Mark Hasegawa-Johnson, 3/9/2019.

Modified by Julia Hockenmaier

Aliza Aufrichtig @alizauf · Mar 4

Garlic halved horizontally = nature's Voronoi diagram?

en.wikipedia.org/wiki/Voronoi_d...

Today’s class

- Perceptron: binary and multiclass case
- Getting a distribution over class labels: one-hot output and softmax
- Differentiable perceptrons: binary and multiclass case
- Cross-entropy loss

Recap: Classification, linear classifiers

Classification as a supervised learning task

- **Classification tasks:** Label data points $\mathbf{x} \in \mathcal{X}$ from an n-dimensional vector space with discrete categories (classes) $y \in \mathcal{Y}$
  Binary classification: Two possible labels $\mathcal{Y}$ = {0,1} or $\mathcal{Y}$ = {-1,+1}
  Multiclass classification: *k* possible labels $\mathcal{Y}$ = {1, 2, …, *k*}

- **Classifier:** a function $\mathcal{X} \rightarrow \mathcal{Y}$ f(**x**) = y
  - **Linear classifiers** f(**x**) = sgn(**wx**) [for binary classification] are parametrized by (n+1)-dimensional weight vectors

- **Supervised learning:** Learn the parameters of the classifier (e.g. **w**) from a labeled data set $\mathcal{D}^{\text{train}} = \{(\mathbf{x}^1, y^1),\ldots,(\mathbf{x}^D, y^D)\}$

Batch versus online training

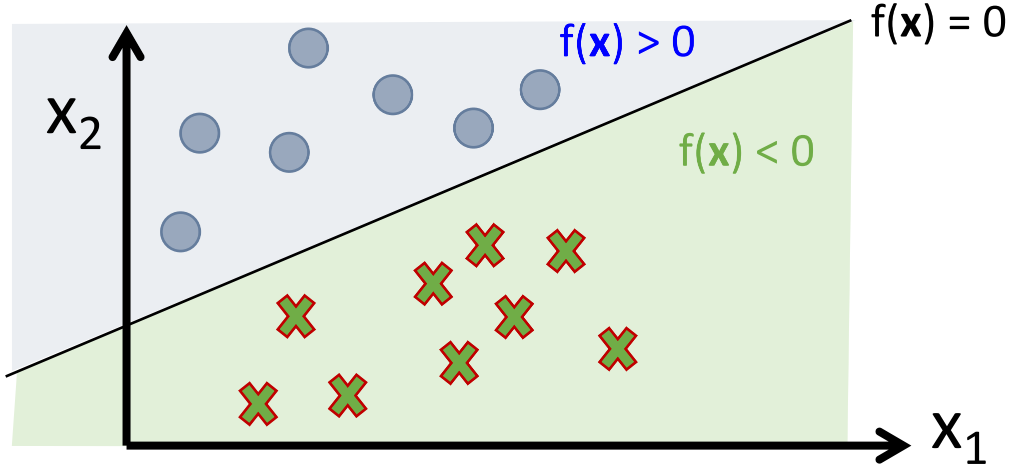
Batch learning: The learner sees the complete training data, and only changes its hypothesis when it has seen **the entire training data set**.

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

For minibatch and online example: randomize the order of examples for each epoch (=complete run through all training examples)

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



Linear classifiers are defined over vector spaces

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

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

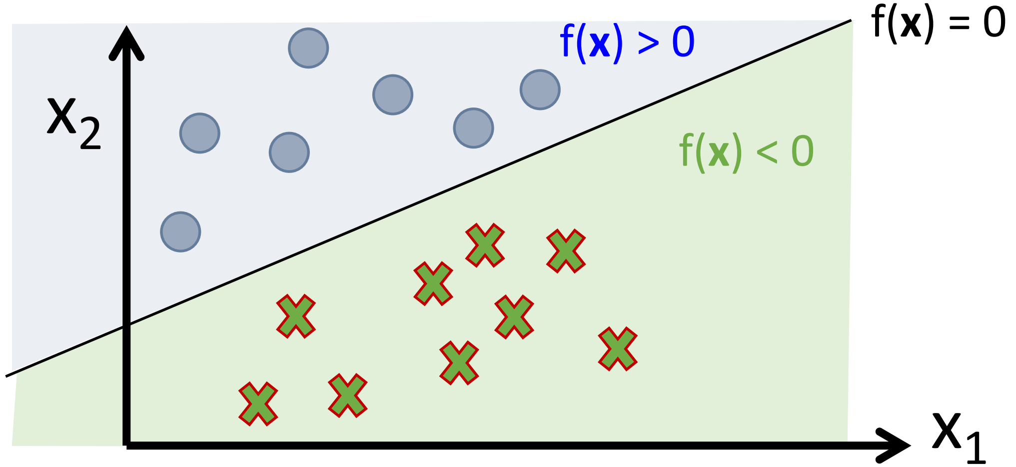
$f(\mathbf{x})$ is also called the decision boundary

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

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

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

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



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

Case 1 correct classification of a positive example ($y = +1 = \hat{y}$):
predicted $f(\mathbf{x}) > 0 \Rightarrow y \cdot f(\mathbf{x}) > 0$ ✓

Case 2 correct classification of a negative example($y = -1 = \hat{y}$):
predicted $f(\mathbf{x}) < 0 \Rightarrow y \cdot f(\mathbf{x}) > 0$ ✓

Case 3 incorrect classification of a positive example ($y = +1 \neq \hat{y} = -1$):
predicted $f(\mathbf{x}) > 0 \Rightarrow y \cdot f(\mathbf{x}) < 0$ ✗

Case 4 incorrect classification of a negative example ($y = -1 \neq \hat{y} = +1$):
predicted $f(\mathbf{x}) < 0 \Rightarrow y \cdot f(\mathbf{x}) < 0$ ✗

Perceptron

Perceptron

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

- Classify with current weights: $y' = \text{sgn}(\vec{w}^T \vec{\boldsymbol{x}})$
  - Predicted labels $y' \in \{-1,1\}$.
- Update weights:
  - if $y = y'$ then do nothing
  - if $y \neq y'$ then $\vec{w} = \vec{w} + \eta\, y\, \vec{x}$
  - η (eta) is a “learning rate.”

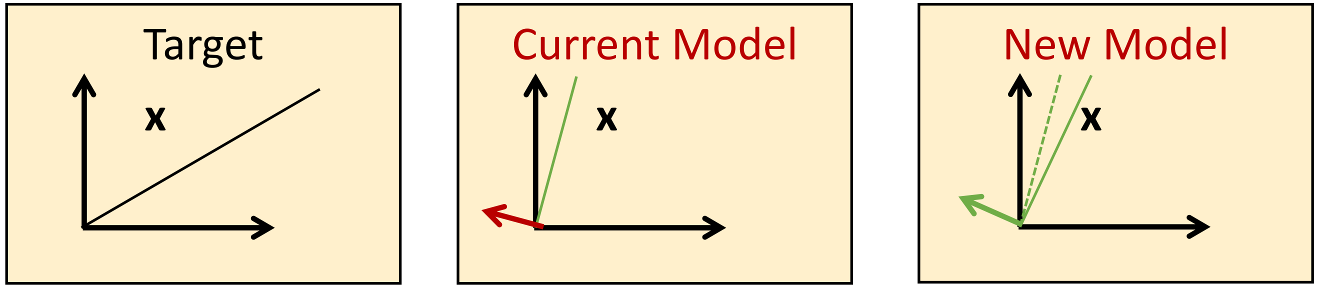
Learning rate η**:** determines how much **w** changes.

Common wisdom: η should get smaller (decay) as we see more examples.

The Perceptron rule

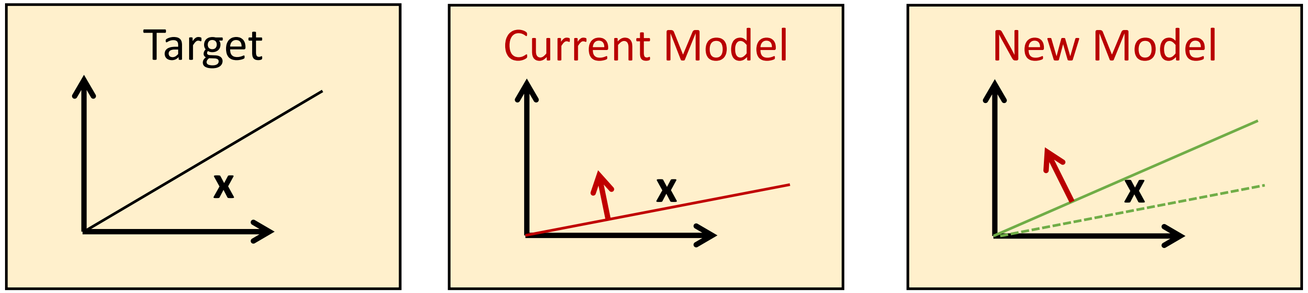
If target y = +1: x should be **above** the decision boundary

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: Proof of Convergence

If **the data are linearly separable** (if there exists a $\vec{w}$ vector such that the true label is given by $y' = \text{sgn}(\vec{w}^T\vec{x})$), then **the perceptron algorithm is guaranteed to converge**, even with a constant learning rate, even $\eta=1$.

> 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 $\vec{w}$ cycles, then no.

If the **data are not linearly separable**, **then perceptron converges (trivially) iff the learning rate decreases**, e.g., $\eta=1/n$ for the n’th training sample.

Multi-class classification

(Ab)using binary classifiers for multiclass classification

- One vs. all scheme:
  K classifiers, one for each of the K classes
  Pick the class with the largest score.

- All vs. all scheme:
  $K(K-1)$ classifiers for each pair of classes
  Pick the class with the largest #votes.

- For both schemes, the classifiers are trained independently.

Multiclass classifier

- A *single* K-class discriminant function, consisting of K linear functions of the form

$$f_k(\mathbf{x}) = \mathbf{w}_k\mathbf{x} + w_{k0}$$

- Assign class *k* if $f_k(\mathbf{x}) > f_j(\mathbf{x})$ for all $j \neq k$
- I.e.: Assign class $k^* = argmax_k(\mathbf{w}_k\mathbf{x} + w_{k0})$
- We can combine the *K* different weight vectors into a single vector **w**:
  - $\mathbf{w} = (\mathbf{w}_1 \ldots \mathbf{w}_k \ldots \mathbf{w}_K)$

NB: Why **w** can be treated as a single vector

Your classifier could map the *n*-dimensional feature vectors **x =** $(x_1,\ldots,x_n)$ to (sparse) **K×*n*-dimensional vectors** $F(y,\mathbf{x})$ in which each class corresponds to *n* dimensions:

$$Y = \{1\ldots K\}, \quad X = R^n \quad F: X\times Y \rightarrow R^{Kn}$$

$$F(1,\mathbf{x}) = [x_1,\ldots,x_n, \ \ldots, \qquad 0, \ldots,0]$$
$$F(i,\mathbf{x}) = [0, \ldots,0, x_1,\ldots,x_n, 0, \ldots,0]$$
$$F(K,\mathbf{x}) = [0, \ldots,0, \quad \ldots, \qquad x_1,\ldots,x_n]$$

Now $\mathbf{w} = [\mathbf{w}_1;\ldots; \mathbf{w}_K]$, and $\mathbf{w}F(y,\mathbf{x}) = \mathbf{w}_y\mathbf{x}$

Multiclass classification

Learning a multiclass classifier:
Find $\mathbf{w}$ such that for all training items $(\mathbf{x}, y_i)$

$$y_i = \operatorname{argmax}_y \mathbf{w}F(y, \mathbf{x})$$

Equivalently, for all $(\mathbf{x}, y_i)$ and all $k \neq i$:

$$\mathbf{w}F(y_i, \mathbf{x}) > \mathbf{w}F(y_k, \mathbf{x})$$

Linear multiclass classifier decision boundaries

Decision boundary between C_k and C_j :

The set of points where $f_k(\mathbf{x}) = f_j(\mathbf{x})$.

$$f_k(\mathbf{x}) = f_j(\mathbf{x})$$

Spelling out $f(\mathbf{x})$:

$$\mathbf{w}_k\mathbf{x} + w_{k0} = \mathbf{w}_j\mathbf{x} + w_{j0}$$

Reordering the terms:

$$(\mathbf{w}_k - \mathbf{w}_j)\mathbf{x} + (w_{k0} - w_{j0}) = 0$$

Multi-class Perceptrons

Multi-class perceptrons

- ***One-vs-others* framework:**
  We keep one weight vector $\mathbf{w}_c$ for each class c
- **Decision rule**: $y = \text{argmax}_c\ \mathbf{w}_c \cdot \mathbf{x}$
- **Update rule:** suppose example from class c gets misclassified as c’
  - **Update for c** [the class we *should* have assigned]: $\mathbf{w}_c \leftarrow \mathbf{w}_c + \eta\,\mathbf{x}$
  - **Update for c’** [the class we *mistakenly* assigned]: $\mathbf{w}_{c'} \leftarrow \mathbf{w}_{c'} - \eta\,\mathbf{x}$
  - **Update for all classes other than c and c’:** no change

One-Hot vectors as target representation

One-hot vectors:

Instead of representing labels as k categories {1,2, …,k}, represent each label as a **k-dimensional vector** where *one element is 1*, and *all other elements are 0*:

class 1 => [1,0,0,…,0]   class 2 => [0,1,0,….,0]   … class k => [0,0,0,…,1]

Each example (in the data) is now a vector $\vec{y}_i$ = [y_{i1},…y_{ij},..,y_{ik}] where

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

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

From one-hot vectors to probabilities

Note that we can interpret $\vec{y}$ as a **probability over class labels**:

For the correct label of $\mathbf{x}_i$: y_{ij} = True value of $P(class\ j\ |\vec{x}_i)$, because the true probability is always either 1 or 0!

Can we define a classifier such that our hypotheses form a distribution?
i.e. $\hat{y}_{ij}$ = Estimated value of $P(class\ j\ |\vec{x}_i)$, $0 \leq \hat{y}_j \leq 1$, $\sum_{j=1}^{C} \hat{y}_j = 1$

Note that the perceptron defines a real vector $[\mathbf{w}_1\mathbf{x}_i, \ldots., \mathbf{w}_k\mathbf{x}_i] \in R^k$

We want to turn $\mathbf{w}_j\mathbf{x}_i$ into a probability $P(\mathbf{class}_j \mid \mathbf{x}_i)$ that is large when $\mathbf{w}_j\mathbf{x}$ is large.

Trick: **exponentiate and renormalize**! This is called the **softmax** function:

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

Added benefit: this is a differentiable function, unlike argmax

Softmax defines a distribution

The softmax function is defined as:

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

Notice that this gives us

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

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

Differentiable Perceptrons (Binary case)

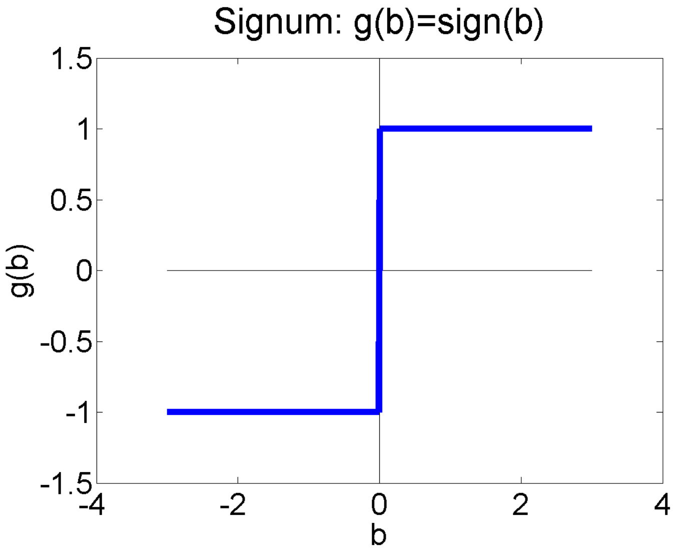
Differentiable Perceptron

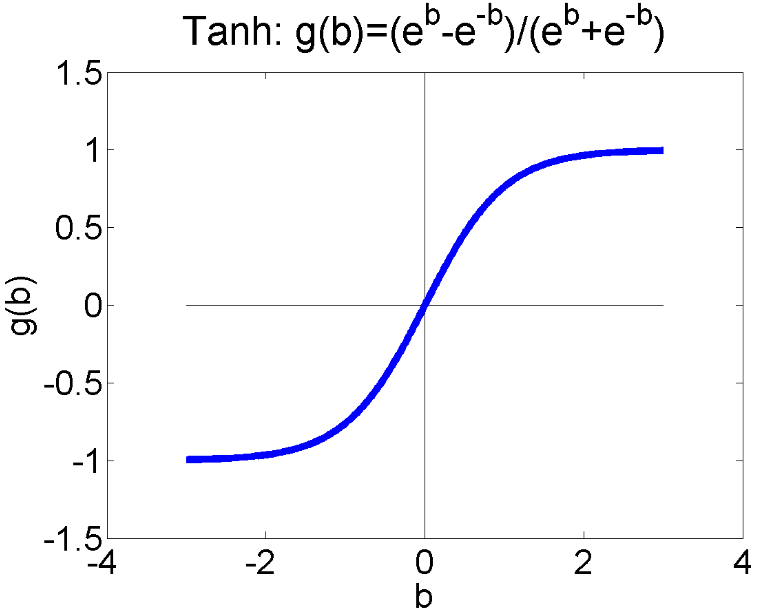
- Also known as a "one-layer feedforward neural network," also known as "logistic regression." Has been re-invented many times by many different people.
- Basic idea: replace the non-differentiable decision function

$$y' = \mathrm{sign}(\vec{w}^T\vec{x})$$

with a differentiable decision function

$$y' = \tanh(\vec{w}^T\vec{x})$$





Differentiable Perceptron

- Suppose we have n training vectors, $\vec{x}_1$ through $\vec{x}_n$. Each one has an associated label $y_i \in \{-1,1\}$. Then we replace the true loss function,

$$L(y_1, \dots, y_n, \vec{x}_1, \dots, \vec{x}_n) = \sum_{i=1}^{n} (y_i - \text{sign}(\vec{w}^T \vec{x}_i))^2$$

with a differentiable error

$$L(y_1, \dots, y_n, \vec{x}_1, \dots, \vec{x}_n) = \sum_{i=1}^{n} (y_i - \tanh(\vec{w}^T \vec{x}_i))^2$$

Why Differentiable?

- Why do we want a differentiable loss function?

$$L(y_1, \dots, y_n, \vec{x}_1, \dots, \vec{x}_n) = \sum_{i=1}^{n} (y_i - \tanh(\vec{w}^T \vec{x}_i))^2$$

- Answer: because if we want to improve it, we can adjust the weight vector in order to reduce the error:

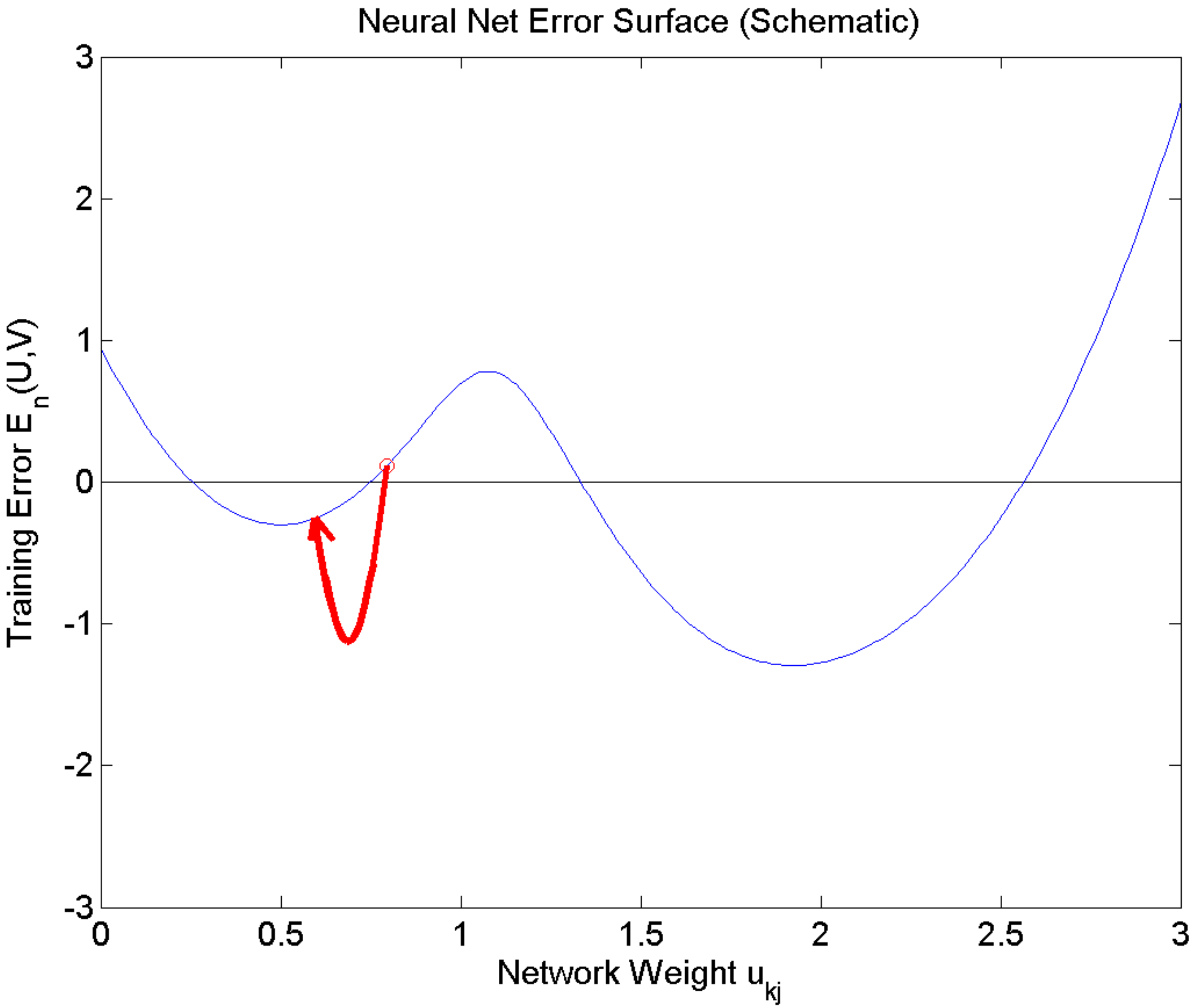
$$\vec{w} = \vec{w} - \eta \nabla_{\vec{w}} L$$

- This is called "gradient descent." We move $\vec{w}$ "downhill," i.e., in the direction that reduces the value of the loss L.

Differential Perceptron

The weights get updated according to

$$\vec{w} = \vec{w} - \eta \nabla_{\vec{w}} L$$



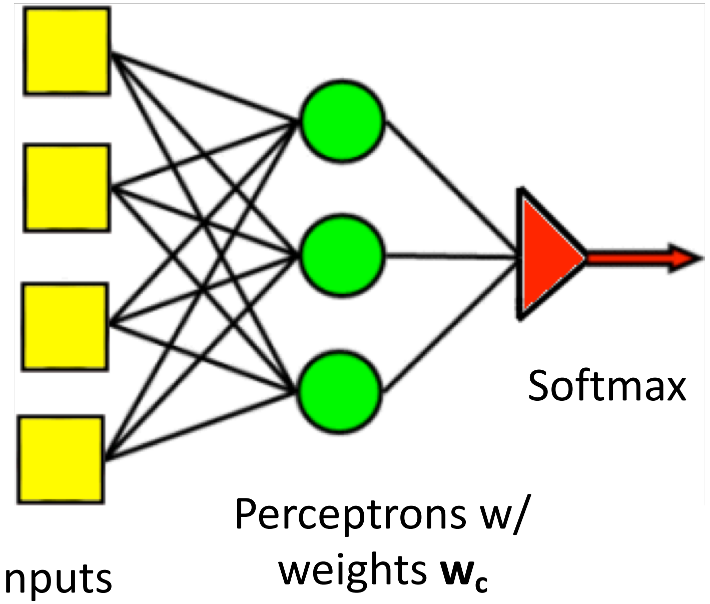
Differentiable Perceptrons (Multi-class case)

Differentiable Multi-class perceptrons

Same idea works for multi-class perceptrons. We replace the non-differentiable decision rule $c = \mathrm{argmax}_c\ \mathbf{w}_c \cdot \mathbf{x}$ with the differentiable decision rule $c = \mathrm{softmax}_c\ \mathbf{w}_c \cdot \mathbf{x}$, where the softmax function is defined as

Softmax:

$$p(c|\vec{x}) = \frac{e^{\vec{w}_c \cdot \vec{x}}}{\sum_{k=1}^{\#\ classes} e^{\vec{w}_k \cdot \vec{x}}}$$



Differentiable Multi-Class Perceptron

- Then we can define the loss to be:

$$L(y_1, \ldots, y_n, \vec{x}_1, \ldots, \vec{x}_n) = -\sum_{i=1}^{n} \ln p(c = y_i | \vec{x}_i)$$

- And because the probability term on the inside is differentiable, we can reduce the loss using gradient descent:

$$\vec{w} = \vec{w} - \eta \nabla_{\vec{w}} L$$

Gradient descent on softmax

$\hat{y}_{ij}$ is the probability of the j^{th} class for the i^{th} training example:

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

Computing the gradient of the loss involves the following term
(for all items i, all classes j and m, and all input features k):

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

w_{mk} is the weight that connects the k^{th} input feature to the m^{th} class label

x_{ik} is the value of the k^{th} input feature for the i^{th} training token

$\hat{y}_{im}$ is the probability of the m^{th} class for the i^{th} training token

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.

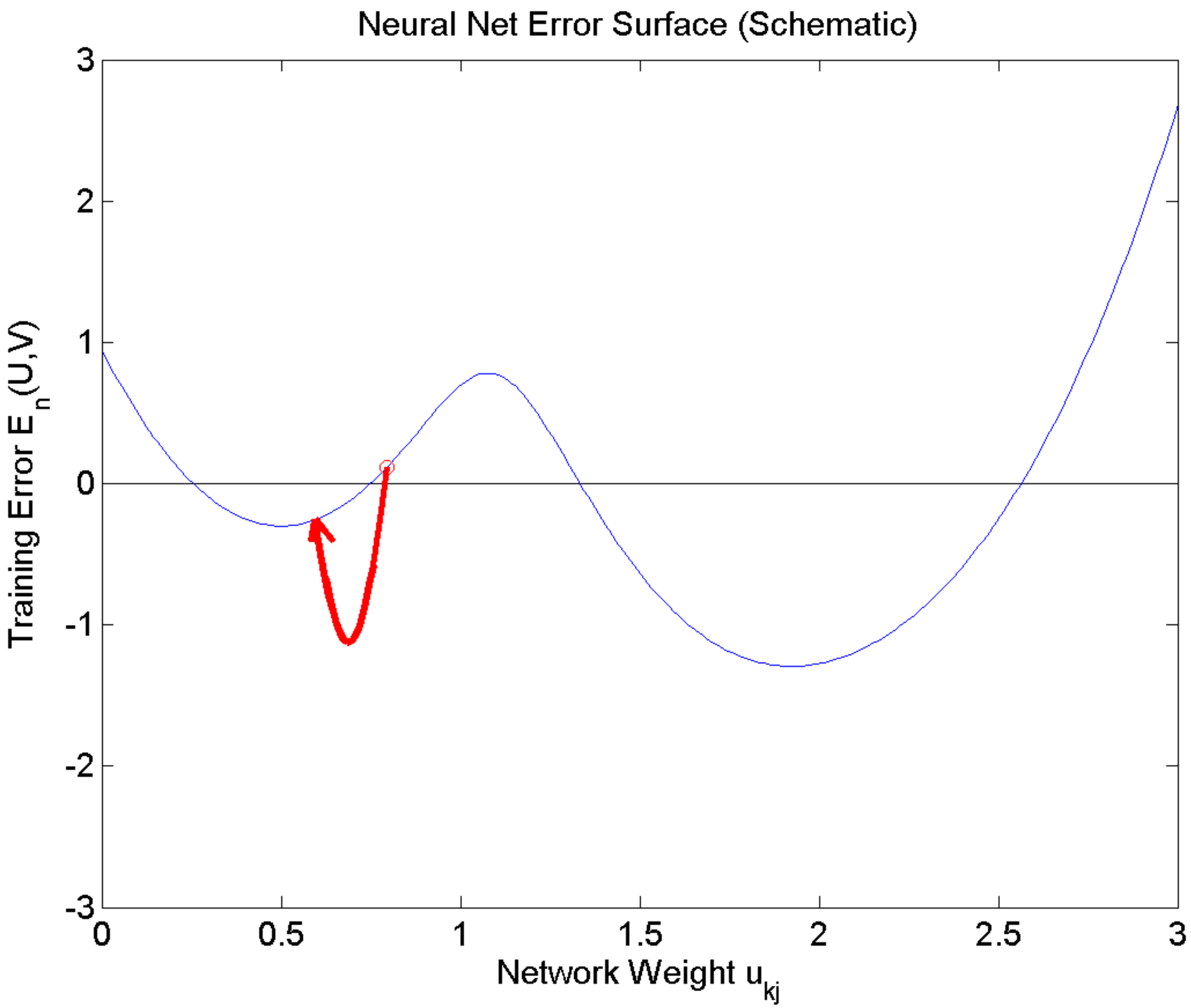
Cross entropy loss

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?



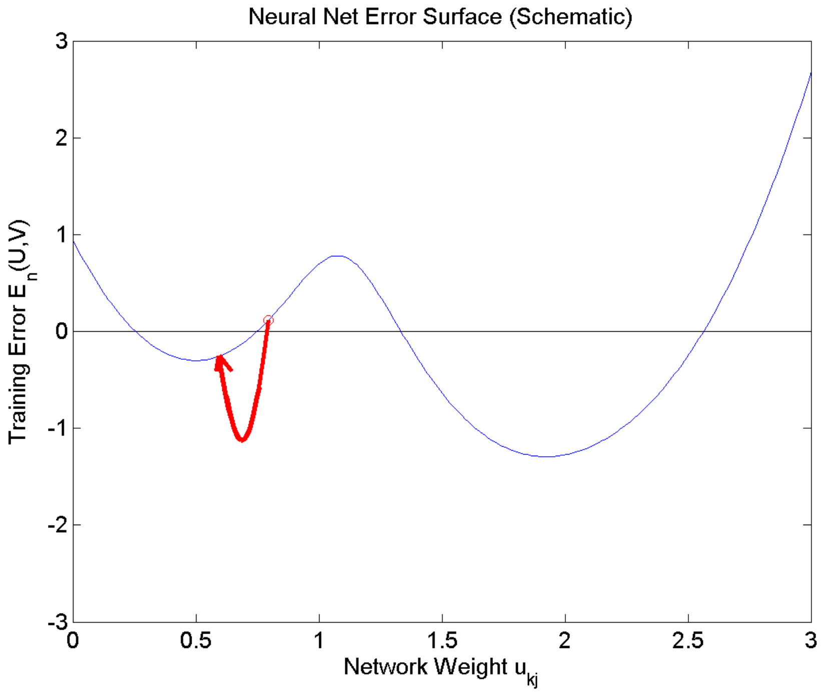
Training: Maximize the probability of the training data

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

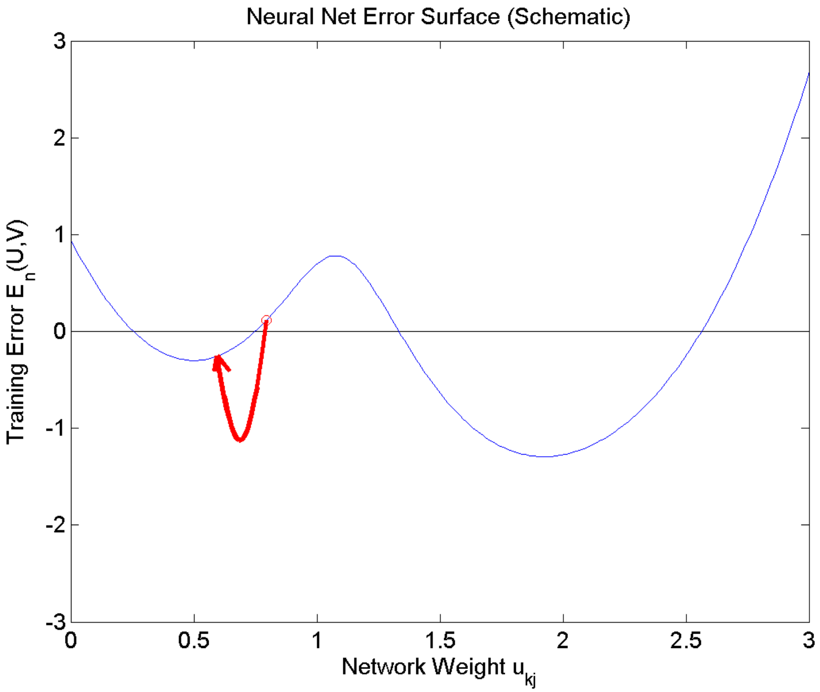
$$\hat{y}_{ij} = \text{Estimated value of } P(\text{class } j \mid \vec{x}_i)$$

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

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

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

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

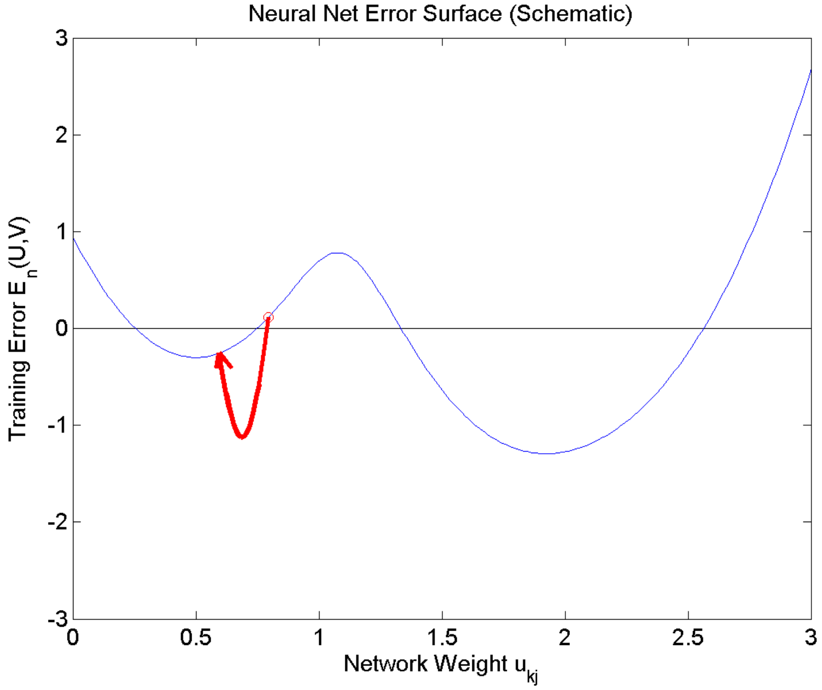
Training: Maximize the probability of the training data

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

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

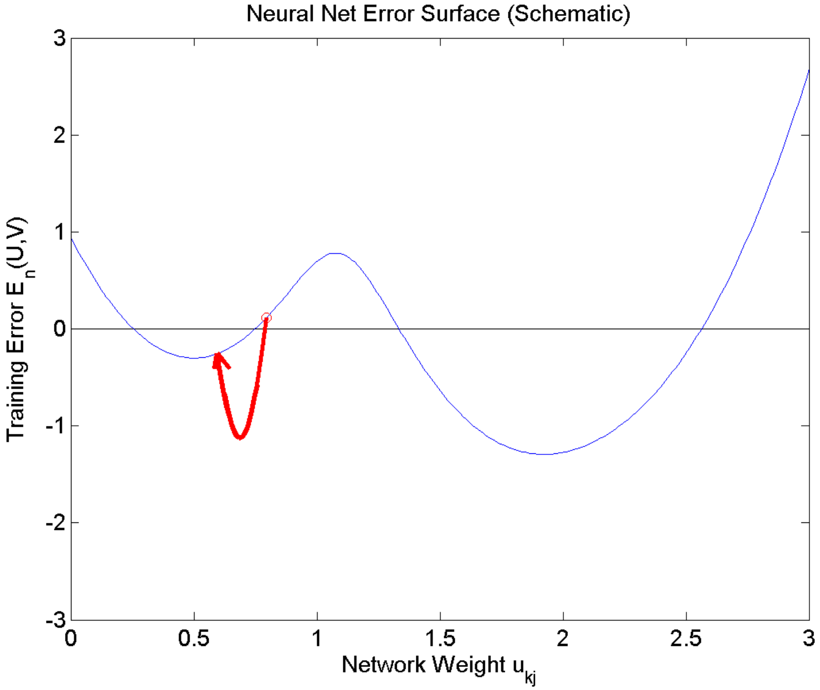
OK. We need to create some notation to mean "the reference label for the i^{th} token." Let's call it $j(i)$.

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



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:

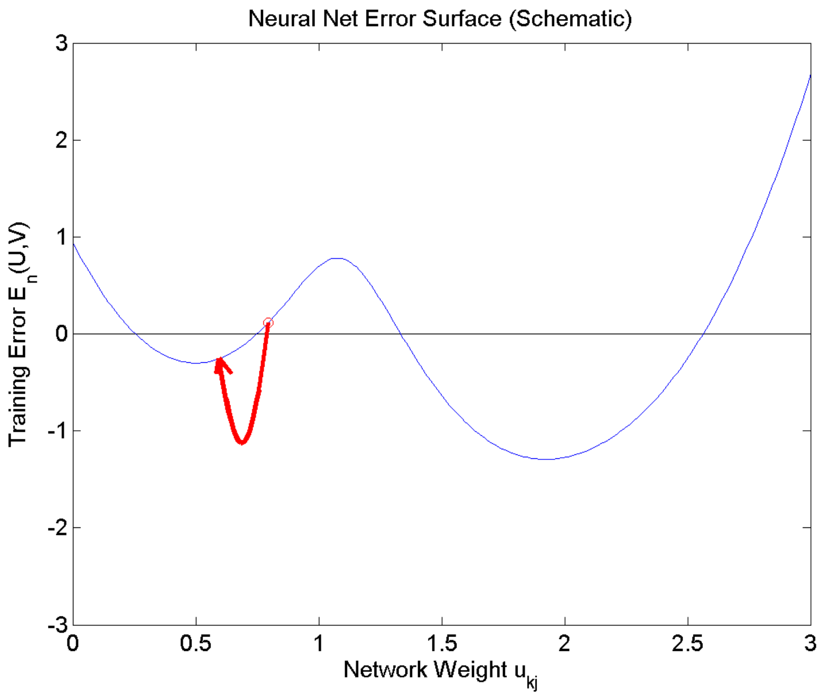


$$w_{mk} = \underset{w}{\operatorname{argmax}} \prod_{i=1}^{n} \hat{y}_{i,j(i)}$$

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.

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

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

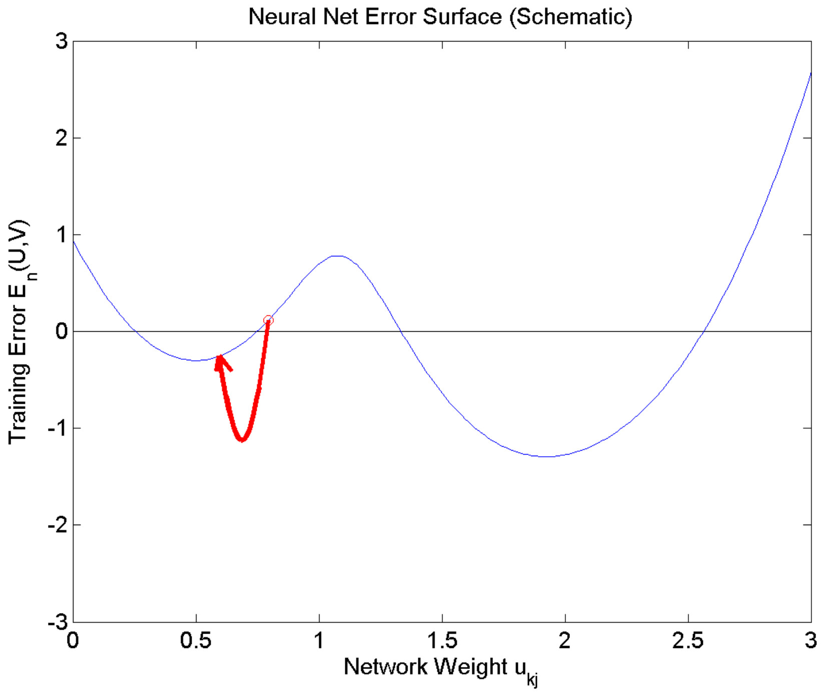
Training: Minimizing the negative log probability

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

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

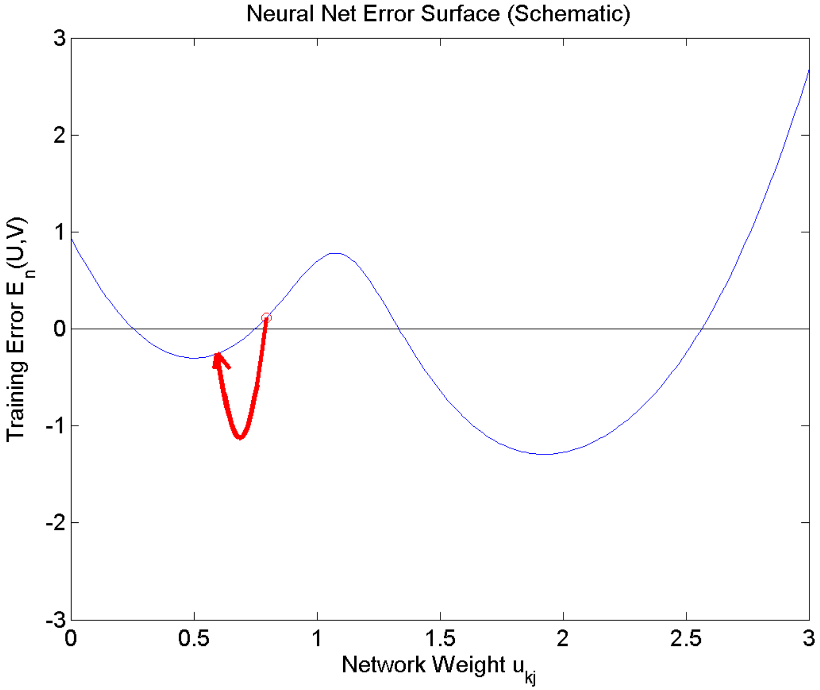
$$L = \sum_{i=1}^{n} -\ln \hat{y}_{i,j(i)}$$

This loss function, defined above, is called the **<u>cross-entropy loss</u>**. 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.



Differentiating the cross-entropy

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



Interpretation:

Increasing w_{mk} will make the error worse if

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

Putting it all together

Summary: Training Algorithms You Know

1. Naïve Bayes with Laplace Smoothing:

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

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

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

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

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

Summary: Perceptron versus Softmax

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

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

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),

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

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):

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

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,

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

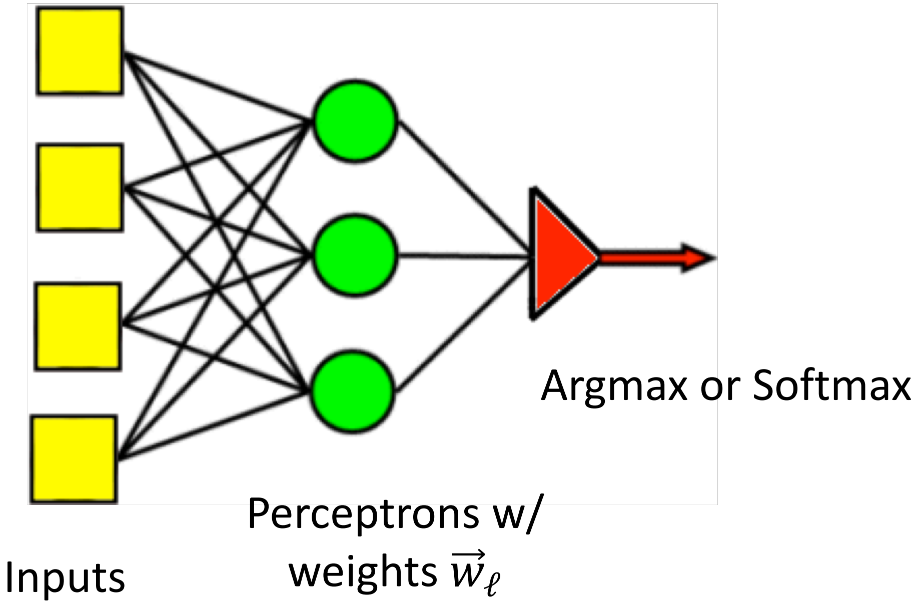
Summary: Perceptron versus Softmax

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

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

Whereas, for a softmax network,

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


Argmax or Softmax

Inputs

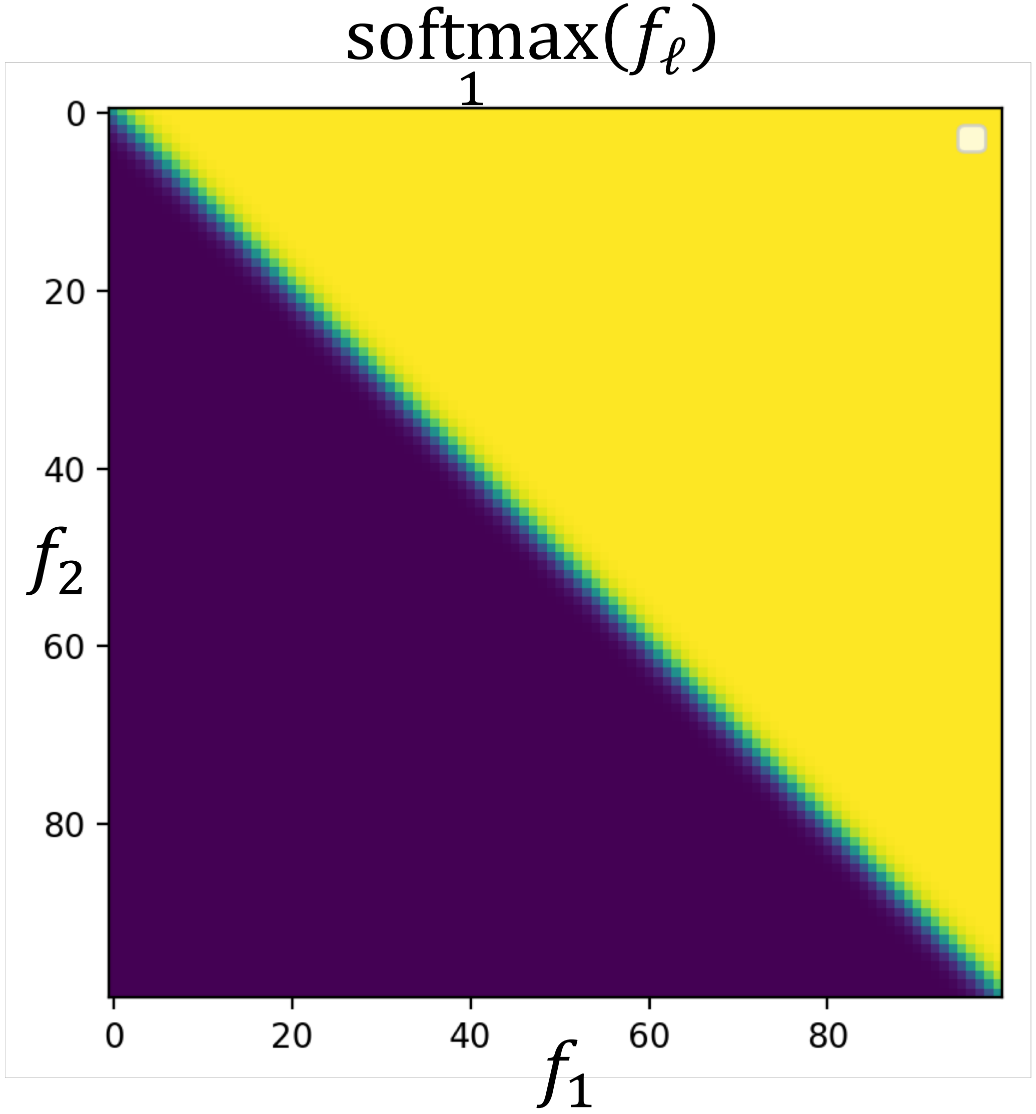
Perceptrons w/ weights $\vec{w}_\ell$

Appendix: How to differentiate the softmax

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



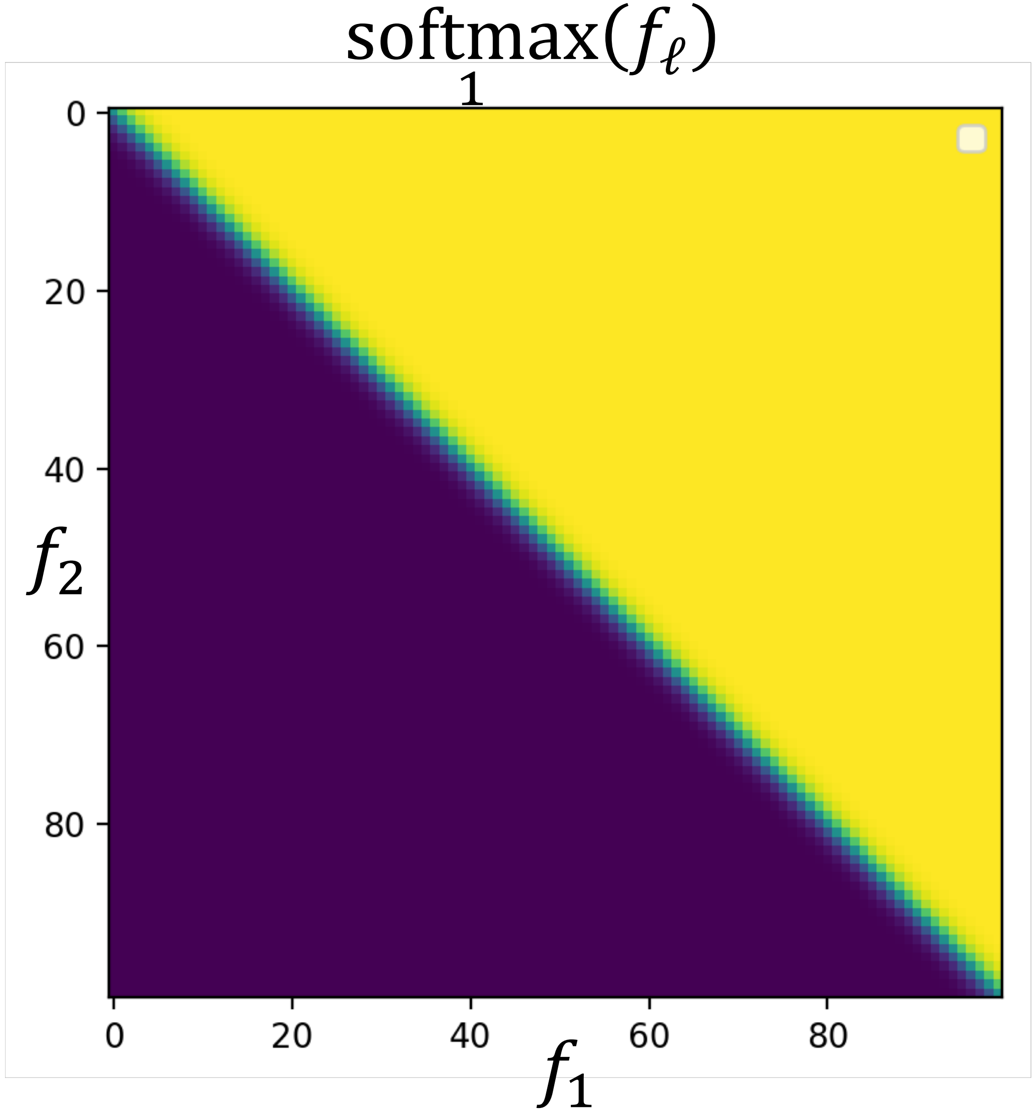
How to differentiate the softmax: step 1

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

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

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

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

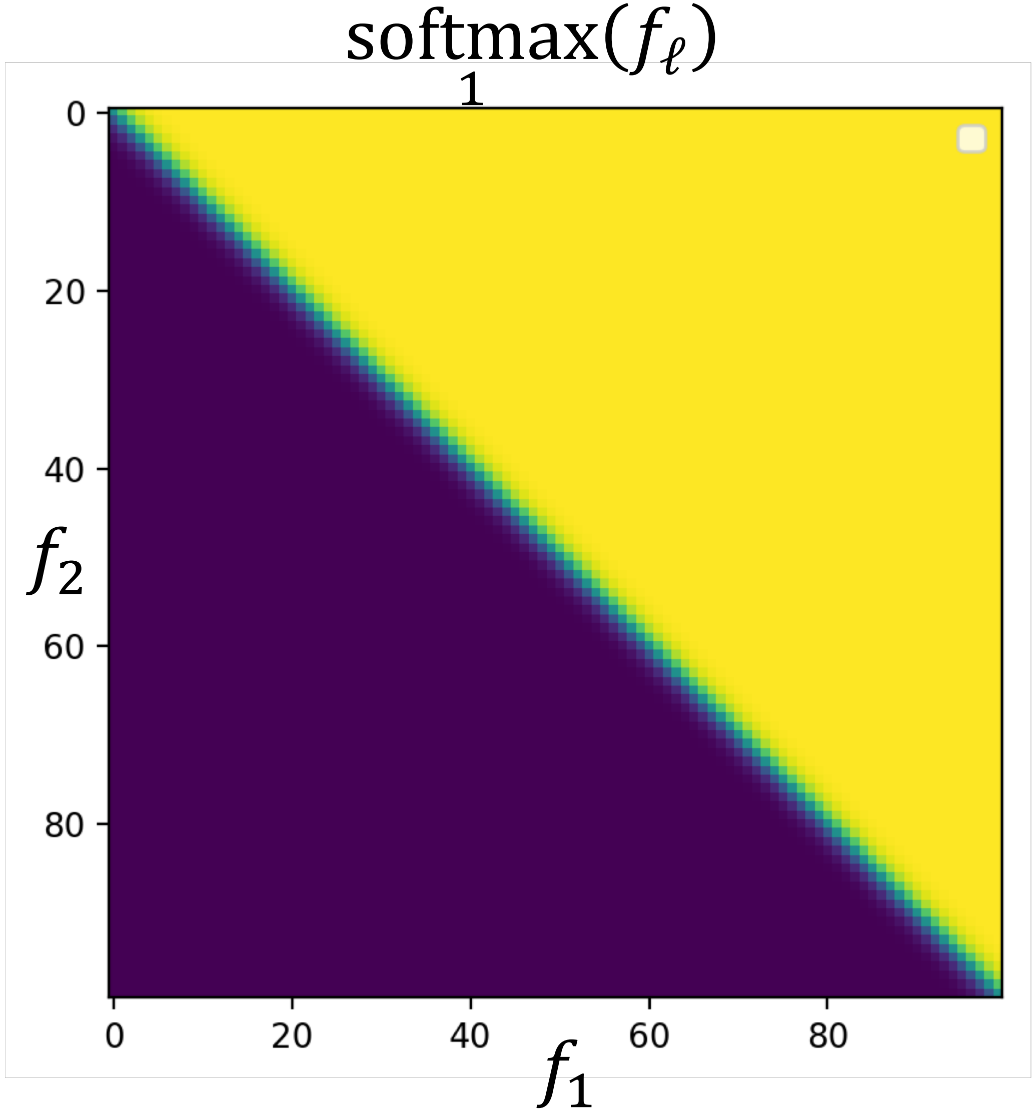


How to differentiate the softmax: step 2

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

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

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

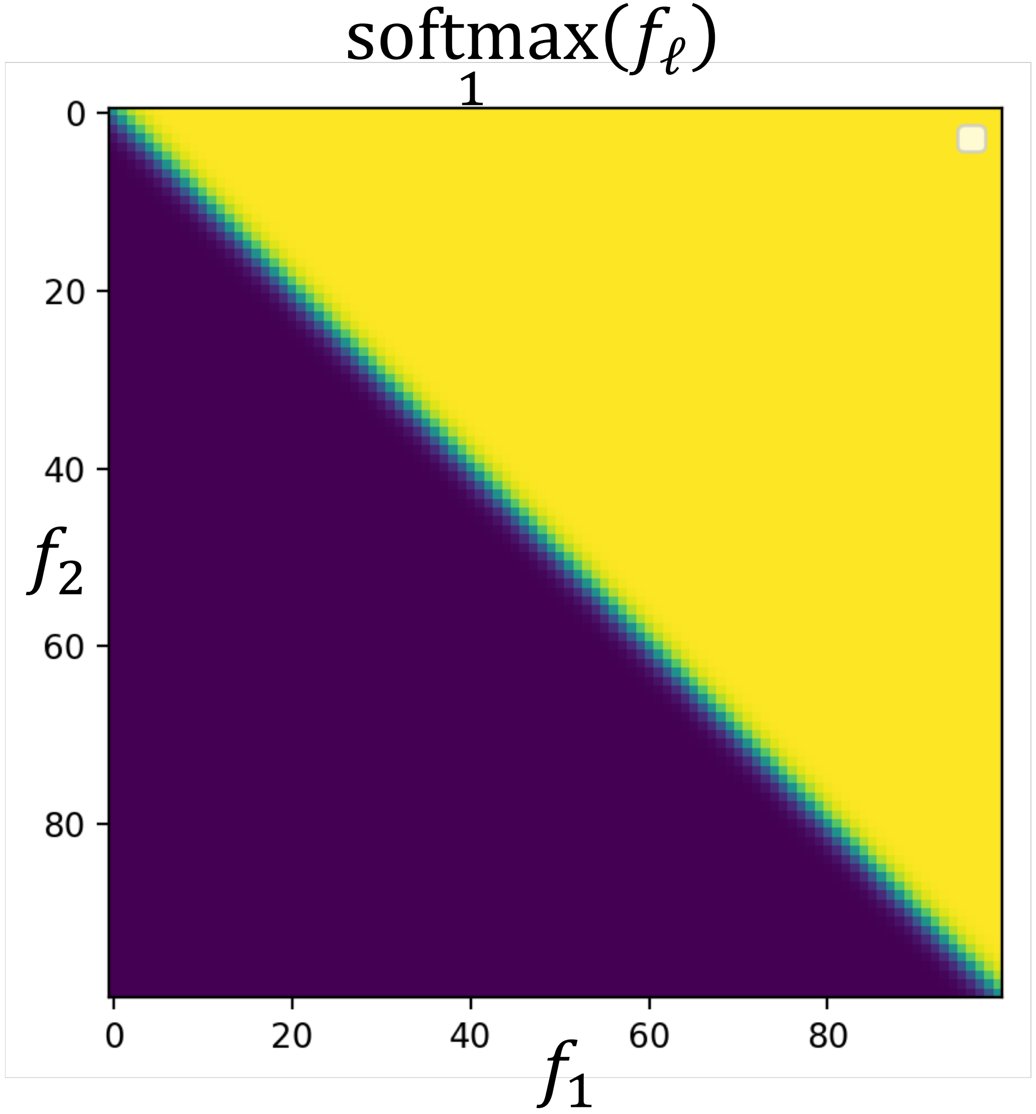


How to differentiate the softmax: step 3

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

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

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

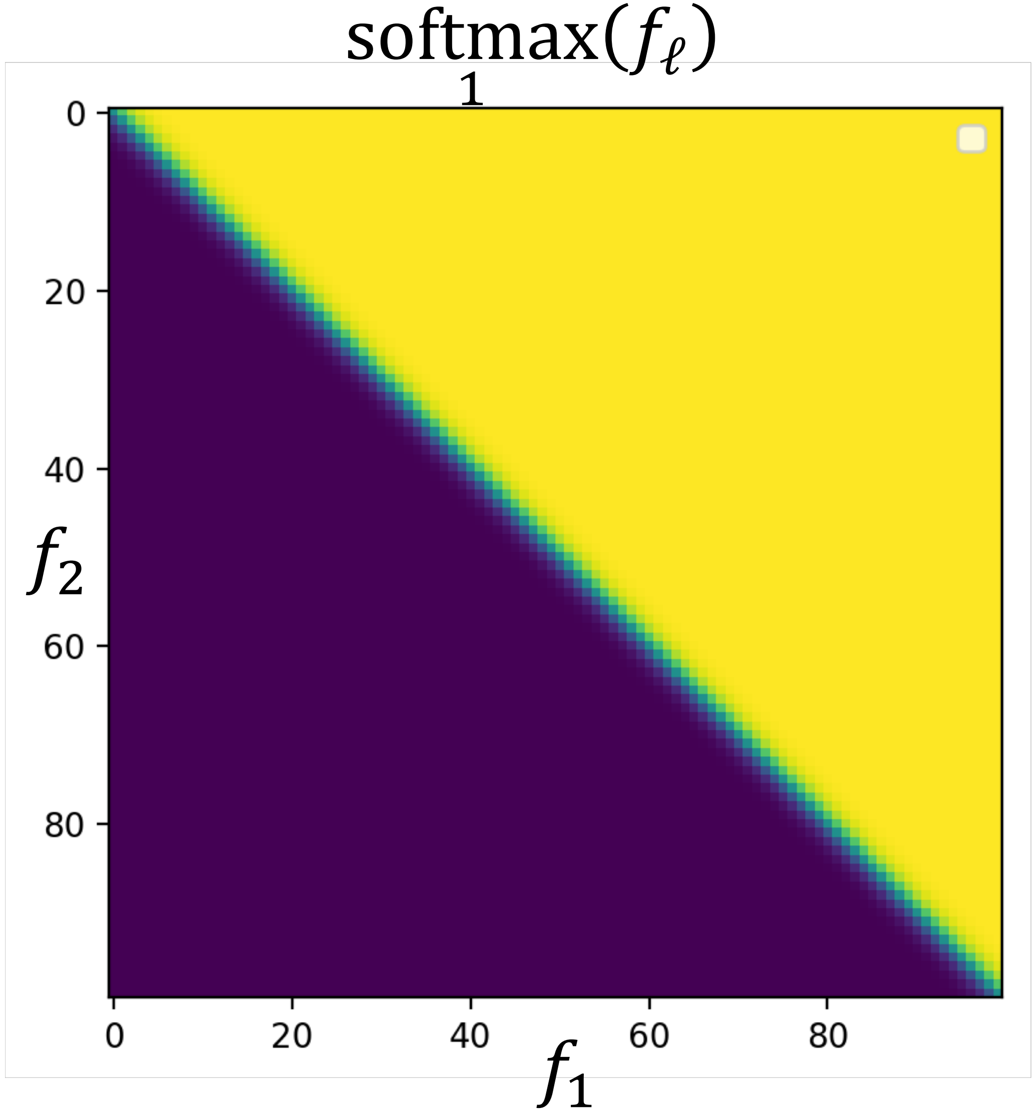


Differentiating the softmax

… 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

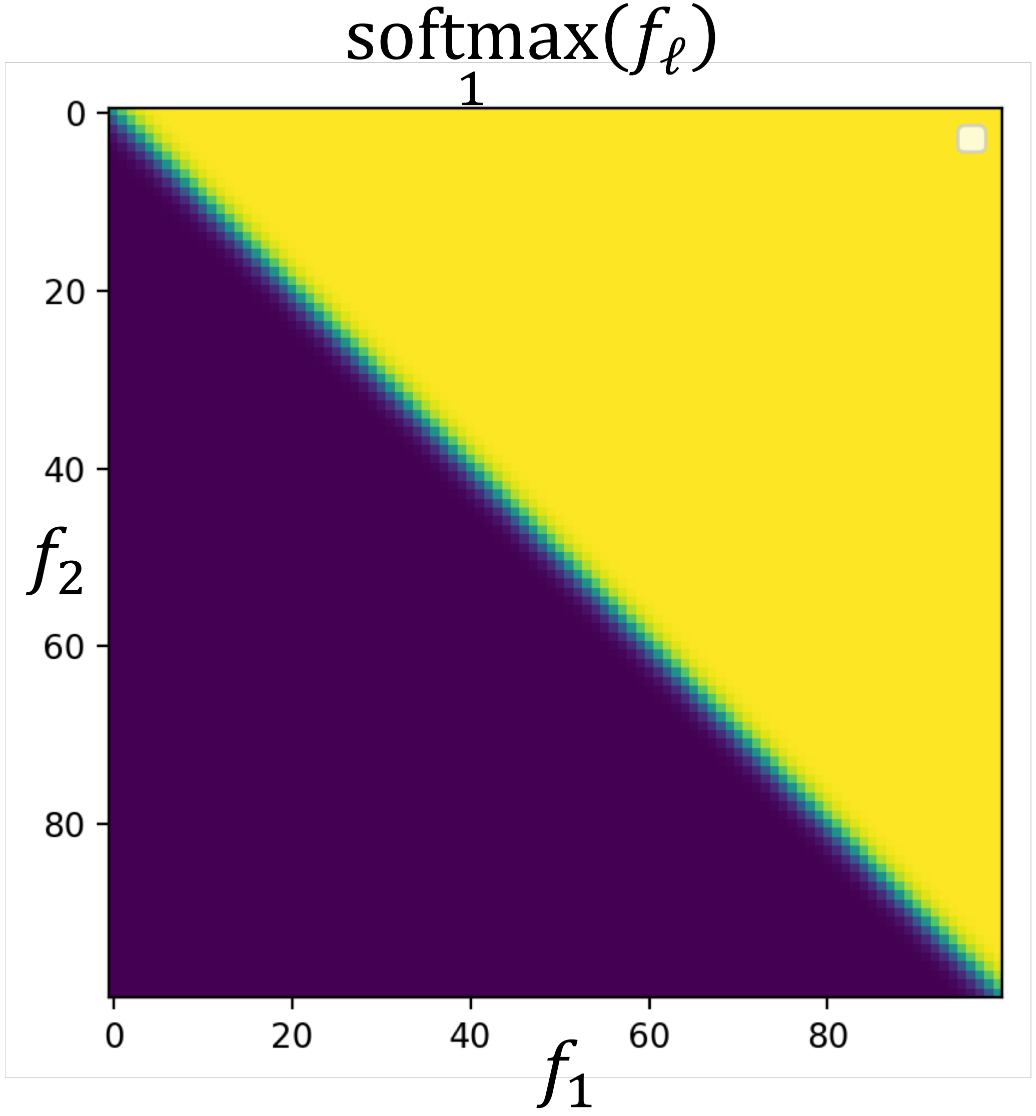
- $\hat{y}_{ij}$ is the probability of the j^{th} class, estimated by the neural net, in response to the i^{th} training token
- w_{mk} is the network weight that connects the k^{th} input feature to the m^{th} class label

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



Appendix: How to differentiate the cross-entropy loss

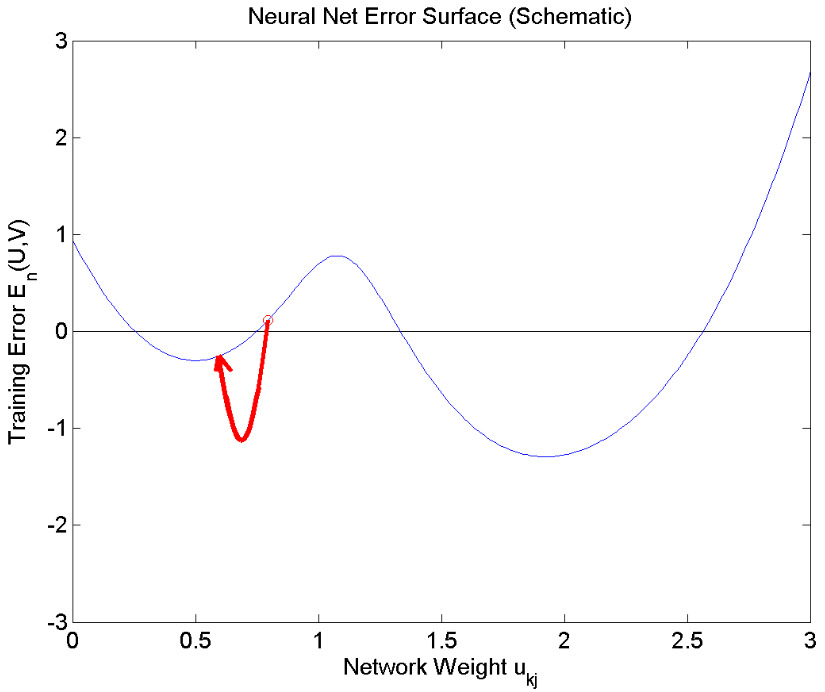
Differentiating the cross-entropy

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:

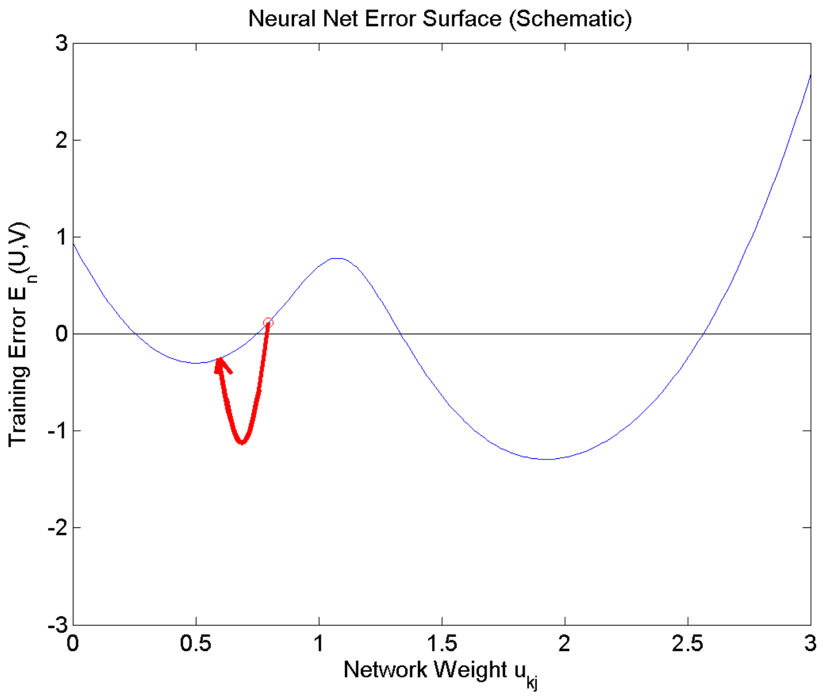
$$L = \sum_{i=1}^{n} -\ln \hat{y}_{i,j(i)}$$

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…

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



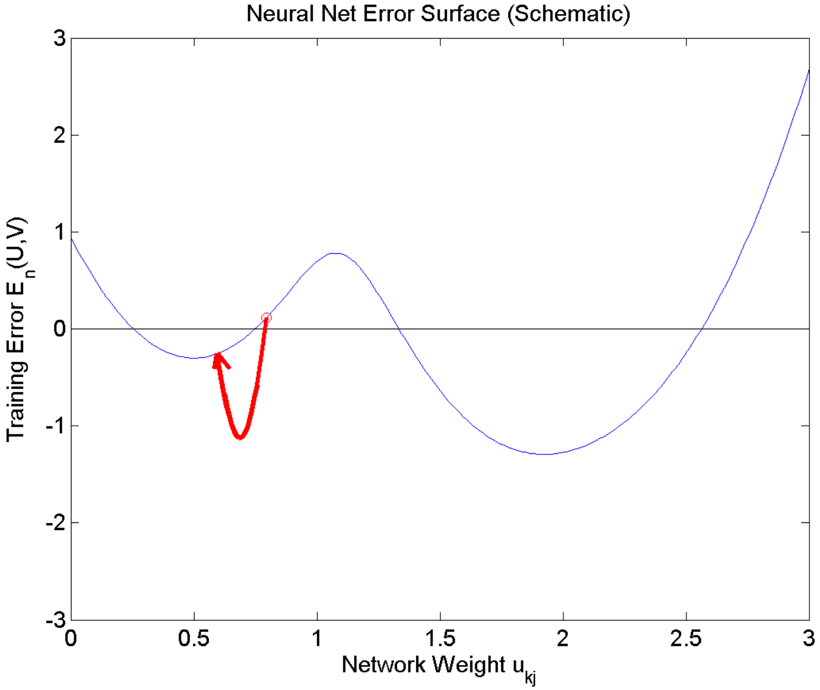
Differentiating the cross-entropy

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:

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

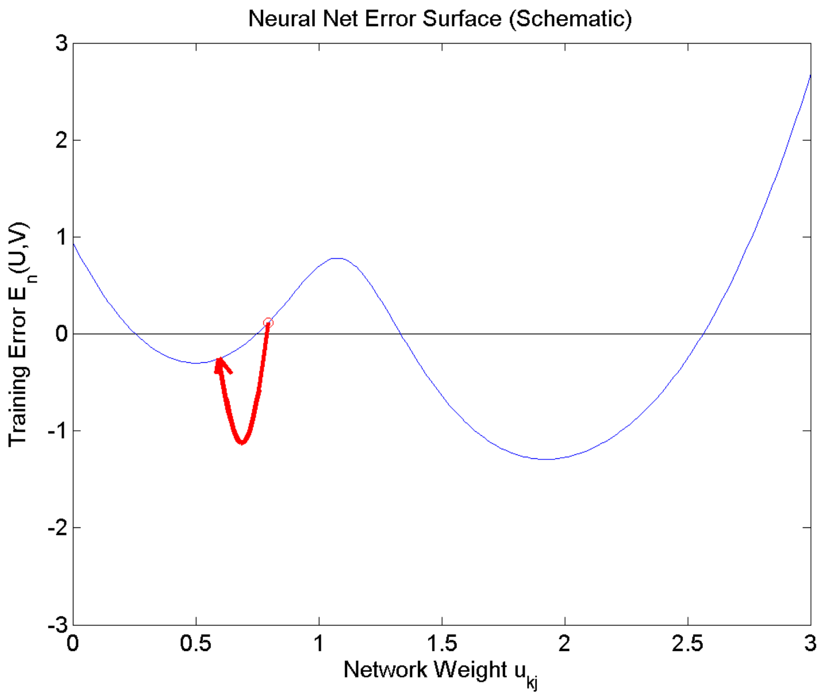
Differentiating the cross-entropy

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…

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



… but remember our reference labels:

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

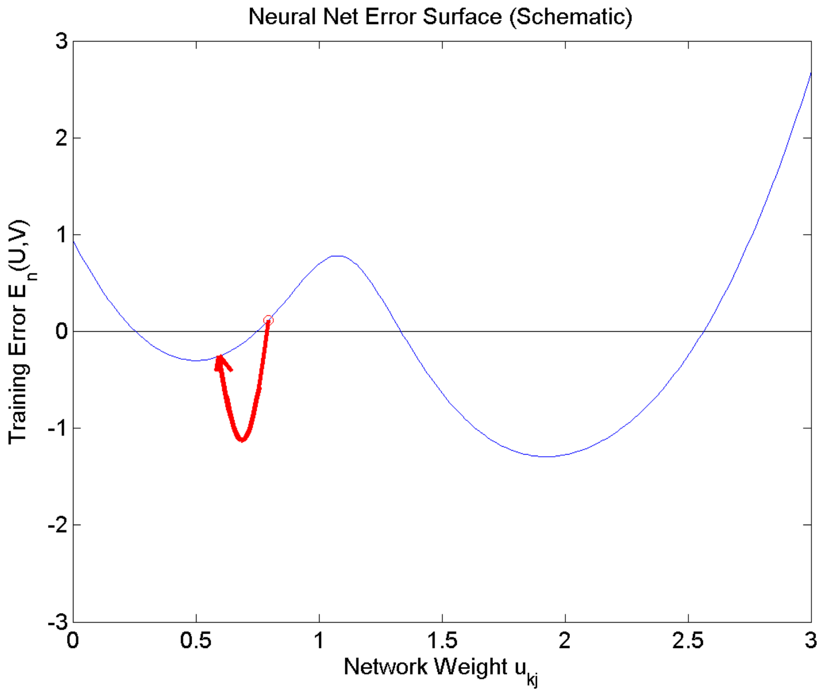
Differentiating the cross-entropy

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…

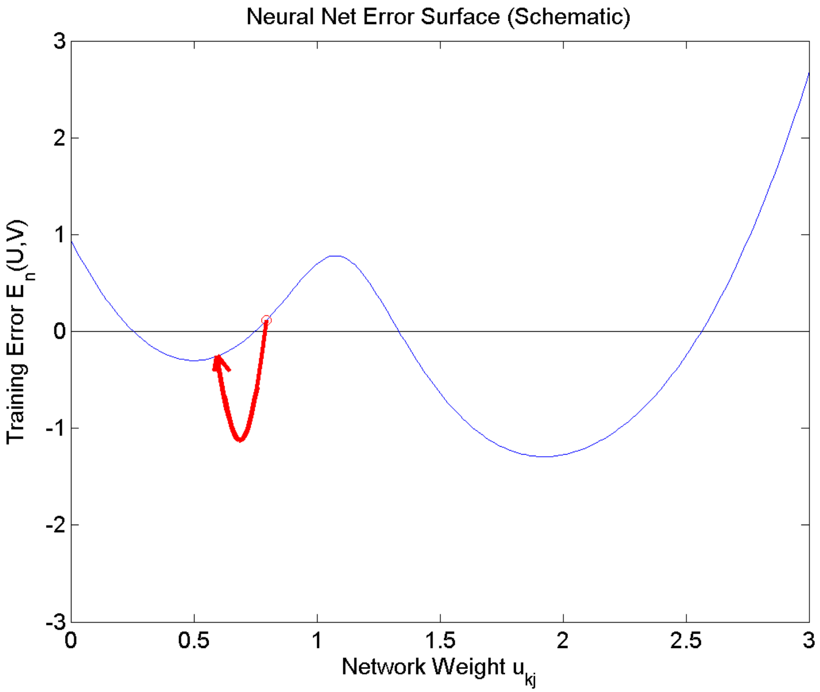
$$\left(\frac{1}{\hat{y}_{i,j(i)}}\right)\frac{\partial \hat{y}_{i,j(i)}}{\partial w_{mk}} = (y_{im} - \hat{y}_{im})f_{ik}$$



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:

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

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