Neural Networks Basics

Kenjiro Taura

Contents

Contents

What is machine learning?

- input: a set of *training data set*

$$D = \{ \ (x_i, t_i) \mid i = 0, 1, \cdots \ \}$$

  - each x_i is normally a real vector (i.e. many real values)
  - each t_i is a real value (regression), 0/1 (binary classification), a discrete value (multi-class classification), etc., depending on the task

What is machine learning?

- input: a set of *training data set*

$$D = \{\ (x_i, t_i) \mid i = 0, 1, \cdots \ \}$$

  - each x_i is normally a real vector (i.e. many real values)
  - each t_i is a real value (regression), 0/1 (binary classification), a discrete value (multi-class classification), etc., depending on the task
- goal: a supervised machine learning tries to find a function f that "matches" training data well. i.e.

$$f(x_i) \approx t_i \text{ for } (x_i, t_i) \in D$$

- put formally, find f that minimizes an *error* or a *loss*:

$$L(f; D) \equiv \sum_{(x_i, t_i) \in D} \mathrm{err}(f(x_i), t_i),$$

  where $\mathrm{err}(y_i, t_i)$ is a function that measures an "error" or a "distance" between the predicted output and the true value

Machine learning as an optimization problem

- finding a good function from the space of *literally all* possible functions is neither easy nor meaningful
- we thus normally fix a search space of functions ($\mathcal{F}$) to a fixed expression parameterized by w and find a good function $f_w \in \mathcal{F}$ *(parametric models)*

Machine learning as an optimization problem

- finding a good function from the space of *literally all* possible functions is neither easy nor meaningful
- we thus normally fix a search space of functions ($\mathcal{F}$) to a fixed expression parameterized by w and find a good function $f_w \in \mathcal{F}$ *(parametric models)*
- the task is then to find the value of w that minimizes the loss:

$$L(w; D) \equiv \sum_{(x_i,t_i)\in D} \operatorname{err}(f_w(x_i), t_i)$$

Contents

A simple example (linear regression)

- training data $D = \{\ (x_i, t_i) \mid i = 0, 1, \cdots\}$
  - x_i : a real value
  - t_i : a real value

A simple example (linear regression)

- training data $D = \{\ (x_i, t_i) \mid i = 0, 1, \cdots \}$
  - x_i : a real value
  - t_i : a real value
- let the search space be a set of polynomials of degree ≤ 2. a function is then parameterized by $w = (w_0\ w_1\ w_2)$. i.e.

$$f_w(x) \equiv w_2 x^2 + w_1 x + w_0$$

A simple example (linear regression)

- training data $D = \{\ (x_i, t_i) \mid i = 0, 1, \cdots \}$
  - x_i : a real value
  - t_i : a real value
- let the search space be a set of polynomials of degree ≤ 2. a function is then parameterized by $w = (w_0\ w_1\ w_2)$. i.e.

$$f_w(x) \equiv w_2 x^2 + w_1 x + w_0$$

- let the error function be a simple square distance:

$$\mathrm{err}(y, t) \equiv (y - t)^2$$

A simple example (linear regression)

- training data $D = \{ \ (x_i, t_i) \mid i = 0, 1, \cdots \}$
  - x_i : a real value
  - t_i : a real value
- let the search space be a set of polynomials of degree ≤ 2. a function is then parameterized by $w = (w_0 \ w_1 \ w_2)$. i.e.

$$f_w(x) \equiv w_2 x^2 + w_1 x + w_0$$

- let the error function be a simple square distance:

$$\mathrm{err}(y, t) \equiv (y - t)^2$$

- the task is to find $w = (w_0, w_1, w_2)$ that minimizes:

$$L(w; D) = \sum_{(x_i, t_i) \in D} \mathrm{err}(f_w(x_i), t_i) = \sum_{(x_i, t_i) \in D} (w_2 x_i^2 + w_1 x_i + w_0 - t_i)^2$$

Contents

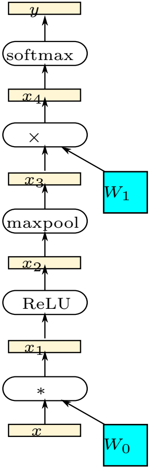
A more realistic example: digit recognition

- training data $D = \{\ (x_i, t_i) \mid i = 0, 1, \cdots\}$
  - x_i : a vector of pixel values of an image:
  - t_i : the class of x_i ($t_i \in \{0, 1, \cdots, 9\}$)

A more realistic example: digit recognition

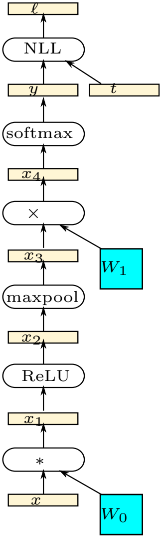
- training data $D = \{ (x_i, t_i) \mid i = 0, 1, \cdots \}$
  - x_i : a vector of pixel values of an image:
  - t_i : the class of x_i ($t_i \in \{0, 1, \cdots, 9\}$)
- the search space: the following composition parameterized by three matrices $W_0$and W_1

$$f_{W_0, W_1}(x) \equiv \mathrm{softmax}(W_1 \mathrm{maxpool}(\mathrm{ReLU}(W_0 * x)))$$



A handwritten digits recognition

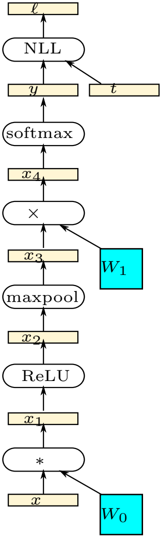
- the output $y = f_{W_0,W_1}(x)$ is a 10-vector representing probabilities that x belongs to each of the ten classes



A handwritten digits recognition

- the output $y = f_{W_0,W_1}(x)$ is a 10-vector representing probabilities that x belongs to each of the ten classes
- a loss function is *negative log-likelihood* commonly used in multiclass classifications

$$\mathrm{err}(y,t) = \mathrm{NLL}(y,t) \equiv -\log y_t$$



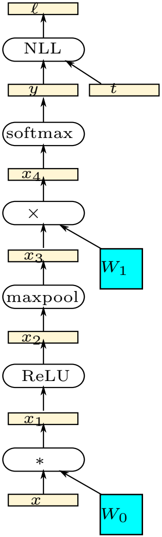
A handwritten digits recognition

- the output $y = f_{W_0,W_1}(x)$ is a 10-vector representing probabilities that x belongs to each of the ten classes
- a loss function is *negative log-likelihood* commonly used in multiclass classifications

$$\mathrm{err}(y,t) = \mathrm{NLL}(y,t) \equiv -\log y_t$$

- the task is to find W_0 and W_1 that minimize:

$$\begin{aligned} & L(W_0, W_1; D) \\ = & \sum_{(x_i,t_i)\in D} \mathrm{NLL}(f_{W_0,W_1}(x_i), t_i) \\ = & \sum_{(x_i,t_i)\in D} \mathrm{NLL}(\mathrm{softmax}(W_1 \mathrm{maxpool}(\mathrm{ReLU}(W_0 * x))), t_i) \end{aligned}$$



Contents

Contents

How to find the minimizing parameter?

- it boils down to minimizing a function that takes *lots of* parameters w

$$L(w; D) = \sum_{(x_i, t_i) \in D} \mathrm{err}(f_w(x_i), t_i),$$

- we compute the derivative of L with respect to w and move w to its opposite direction *(gradient descent; GD)*

$$w = w - \eta^t \frac{\partial L}{\partial w}$$

(η : a small value controlling a learning rate)

- repeat this until $L(w; D)$ converges

Why GD works

- recall
$$L(w + \Delta w; D) \approx L(w; D) + \frac{\partial L}{\partial w}\Delta w$$
- so, by moving w slightly to the direction of gradient (i.e., $\Delta w = -\eta \frac{{}^t\partial L}{\partial w}$ for small η),

$$\begin{aligned} L(w - \eta \frac{{}^t\partial L}{\partial w}; D) &\approx L(w; D) - \eta \frac{\partial L}{\partial w} \frac{{}^t\partial L}{\partial w} \\ &< L(w; D) \end{aligned}$$

L will decrease

A linear regression example

- recall that in the linear regression example:

$$L(w; D) = \sum_{(x_i, t_i) \in D} (w_2 x_i^2 + w_1 x_i + w_0 - t_i)^2$$

A linear regression example

- recall that in the linear regression example:

$$L(w; D) = \sum_{(x_i, t_i) \in D} (w_2 x_i^2 + w_1 x_i + w_0 - t_i)^2$$

- differentiate L by $w = {}^t(w_0 \;\; w_1 \;\; w_2)$ to get:

$$\frac{\partial L}{\partial w} = \sum_{(x_i, t_i) \in D} 2(w_2 x_i^2 + w_1 x_i + w_0 - t_i)(1 \;\; x_i \;\; x_i^2)$$

(remark: we used a chain rule)

A linear regression example

- recall that in the linear regression example:

$$L(w; D) = \sum_{(x_i, t_i) \in D} (w_2 x_i^2 + w_1 x_i + w_0 - t_i)^2$$

- differentiate L by $w = {}^t(w_0 \;\; w_1 \;\; w_2)$ to get:

$$\frac{\partial L}{\partial w} = \sum_{(x_i, t_i) \in D} 2(w_2 x_i^2 + w_1 x_i + w_0 - t_i)(1 \;\; x_i \;\; x_i^2)$$

  (remark: we used a chain rule)

- so you repeat:

$$w = w - \eta \sum_{(x_i, t_i) \in D} 2(w_2 x_i^2 + w_1 x_i + w_0 - t_i) \begin{pmatrix} 1 \\ x_i \\ x_i^2 \end{pmatrix}$$

  until $L(w; D)$ converges

A problem of the gradient descent

- the loss function we want to minimize is normally a summation over *all* training data:

$$L(w; D) = \sum_{(x_i, t_i) \in D} \mathrm{err}(f_w(x_i), t_i)$$

A problem of the gradient descent

- the loss function we want to minimize is normally a summation over *all* training data:

$$L(w; D) = \sum_{(x_i, t_i) \in D} \text{err}(f_w(x_i), t_i)$$

- the gradient descent method just described:
  1. computes $\dfrac{\partial}{\partial w}\text{err}(f_w(x_i), t_i)$ for each training data $(x_i, t_i) \in D$, *with the current value of w*
  2. sum them over *whole data set* and then update w

A problem of the gradient descent

- the loss function we want to minimize is normally a summation over *all* training data:

$$L(w; D) = \sum_{(x_i, t_i) \in D} \mathrm{err}(f_w(x_i), t_i)$$

- the gradient descent method just described:
  1. computes $\dfrac{\partial}{\partial w}\mathrm{err}(f_w(x_i), t_i)$ for each training data $(x_i, t_i) \in D$, *with the current value of w*
  2. sum them over *whole data set* and then update w
- it is commonly observed that the convergence becomes faster when we update w more "incrementally" $\rightarrow$ *Stochastic Gradient Descent (SGD)*

Contents

SGD

repeat:

1. randomly draw a *subset* of training data D' (a mini batch; $D' \subset D$)

SGD

repeat:

1. randomly draw a *subset* of training data D' (a mini batch; $D' \subset D$)
2. compute the gradient of loss *over the mini batch*

$$\frac{\partial L(w; D')}{\partial w} = \sum_{(x_i,t_i) \in D'} \frac{\partial}{\partial w} \mathrm{err}(f_w(x_i), t_i)$$

SGD

repeat:

1. randomly draw a *subset* of training data D' (a mini batch; $D' \subset D$)
2. compute the gradient of loss *over the mini batch*

$$\frac{\partial L(w; D')}{\partial w} = \sum_{(x_i, t_i) \in D'} \frac{\partial}{\partial w} \mathrm{err}(f_w(x_i), t_i)$$

3. update w

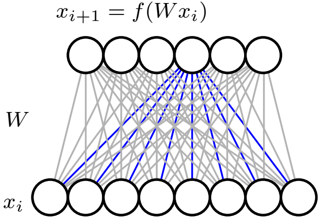
$$w = w - \eta^t \frac{\partial L(w; D')}{\partial w}$$

4. "update sooner rather than later"

Computing the gradients

- in neural networks, a function is a composition of many stages each represented by a lot of parameters

$$\begin{aligned} x_1 &= f_1(w_1; x) \\ x_2 &= f_2(w_2; x_1) \\ &\cdots \\ y &= f_n(w_n; x_n) \\ \ell &= \mathrm{err}(y, t) \end{aligned}$$

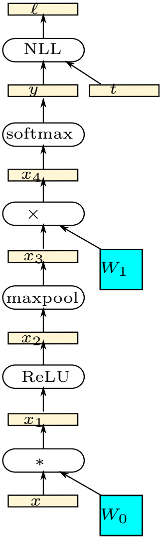


- we need to differentiate ℓ by $w_1, \cdots, w_n$

The digit recognition example

$$\begin{aligned}
x_1 &= W_0 * x \\
x_2 &= \mathrm{ReLU}(x_1) \\
x_3 &= \mathrm{maxpool}(x_2) \\
x_4 &= W_1 x_3 \\
y &= \mathrm{softmax}(x_4) \\
\ell &= \mathrm{NLL}(y, t)
\end{aligned}$$

you need to differentiate ℓ by W_0 and W_1



Contents

Differentiating multivariable functions

- $x = {}^t(x_0 \; \cdots \; x_{n-1}) \in R^n$ (a column vector)
- $f(x)$: a scalar
- **definition:** the gradient of f with respect to x, written $\dfrac{\partial f}{\partial x}$, is a row n-vector s.t.

$$\begin{aligned} \Delta f &\equiv f(x + \Delta x) - f(x) \\ &\approx \frac{\partial f}{\partial x} \Delta x \\ &= \sum_{i=0}^{n-1} \left(\frac{\partial f}{\partial x} \right)_i \Delta x_i \end{aligned}$$

- when it exists,

$$\frac{\partial f}{\partial x} = \left(\frac{\partial f}{\partial x_0} \; \cdots \; \frac{\partial f}{\partial x_{n-1}} \right),$$

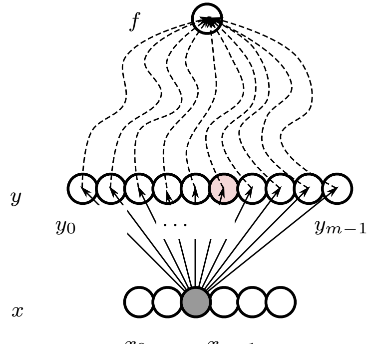
so

$$\Delta f \approx \sum_{i=0}^{n-1} \frac{\partial f}{\partial x_i} \Delta x_i$$

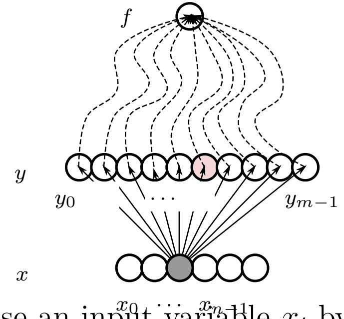
The Chain Rule

- consider a function f that depends on $y = (y_0, \cdots, y_{m-1}) \in R^m$, each of which in turn depends on $x = (x_0, \cdots, x_{n-1}) \in R^n$
- the chain rule (math textbook version):

$$\frac{\partial f}{\partial x_i} = \sum_{0 \le j < m} \frac{\partial f}{\partial y_j} \frac{\partial y_j}{\partial x_i} \quad (0 \le i < n)$$



The Chain Rule : intuition



- say you increase an input variable x_i by Δx_i, each y_j will increase by

$$\approx \frac{\partial y_j}{\partial x_i} \Delta x_i,$$

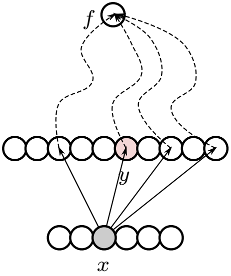
which will contribute to increasing the final output (f) by

$$\approx \frac{\partial f}{\partial y_j} \frac{\partial y_j}{\partial x_i} \Delta x_i$$

Chain Rule

- master the following “index-free” version for neural network
- x, y : a scalar (a single component in a vector/matrix/high dimensional array)
- the chain rule (ML practioner’s version):

$$\frac{\partial f}{\partial x} = \sum_{\text{all variables } y \text{ that } x \textit{ directly} \text{ affects}} \frac{\partial f}{\partial y} \frac{\partial y}{\partial x}$$



Chain Rule and "Back Propagation"

- Chain rule allows you to compute

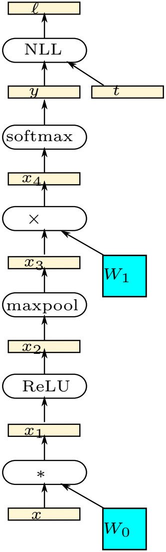
$$\frac{\partial L}{\partial x},$$

the derivative of the loss with respect to a variable, from

$$\frac{\partial L}{\partial y},$$

the derivatives of the loss with respect to upstream variables

$$\frac{\partial L}{\partial x} = \sum_{\text{all variables } y \text{ a step ahead of } x} \frac{\partial L}{\partial y}\frac{\partial y}{\partial x}$$



Contents

Component functions

we use the following functions

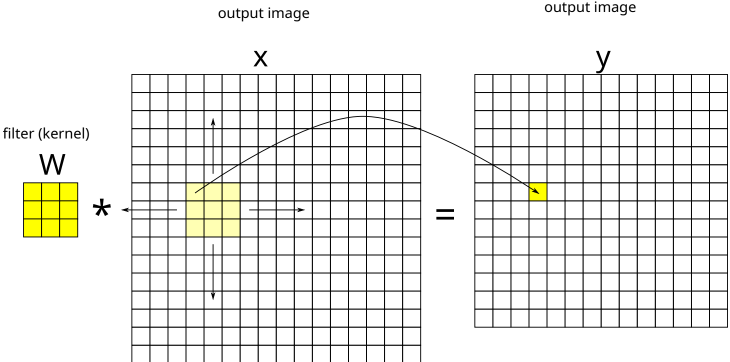
- Convolution$(W; x)$: applies a linear filter
- Linear$(W; x)$: multiplies x by W
- ReLU(x) : zero negative values
- maxpool(x) : replaces each 2x2 patch with 1x1
- dropout(x) : probabilistically zeros some values
- softmax(x) : normalizes x and amplifies large values
- NLL(x, t) : negative log-likelihood

we summarize their definitions and their derivatives

Convolution

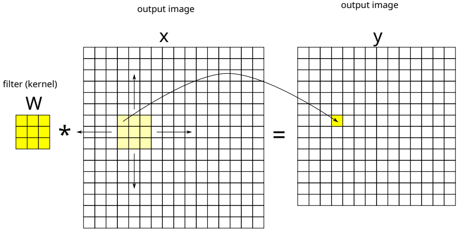
- it takes
  - an image = 2D pixels × a number of channels
  - a "filter" or a "kernel", which is essentially a small image

  and slides the filter over all pixels of the input and takes the local inner product at each pixel
- an illustration of a single channel 2D convolution (imagine a grayscale image)



Convolution (a single channel version)

- $W_{i,j}$: a filter ($0 \leq i < K$, $0 \leq j < K$)
- b : bias
- $x_{i,j}$: an input image ($0 \leq i < H$, $0 \leq j < W$)
- $y_{i,j}$: an output image ($0 \leq i < H - K + 1$, $0 \leq j < W - K + 1$)



$$\forall i, j \quad y_{i,j} = \sum_{0 \leq i' < K, 0 \leq j' < K} w_{i',j'} x_{i+i',j+j'} + b$$

Convolution (multiple channels version)

- say input has IC channels and output OC channels
- $W_{oc,ic,i,j}$: filter ($0 \leq ic < IC$, $0 \leq oc < OC$)
- b_{oc} : bias ($0 \leq oc < OC$)
- $x_{ic,i,j}$: an input image
- $y_{oc,i,j}$: an output image

$$\forall oc, i, j \quad y_{oc,i,j} \;=\; \sum_{ic,i',j'} w_{oc,ic,i',j'} x_{ic,i+i',j+j'} + b_{oc}$$

- the actual code does this for each sample in a batch

$$\forall s, oc, i, j \quad y_{s,oc,i,j} \;=\; \sum_{ic,i',j'} w_{oc,ic,i',j'} x_{s,ic,i+i',j+j'} + b_{oc}$$

Convolution (Back propagation 1)

- $\frac{\partial L}{\partial x}$

$$
\begin{aligned}
\frac{\partial L}{\partial x_{s,ic,i+i',j+j'}} &= \sum_{s',oc,i,j} \frac{\partial L}{\partial y_{s',oc,i,j}} \frac{\partial y_{s',oc,i,j}}{\partial x_{s,ic,i+i',j+j'}} \\
&= \sum_{oc,i,j} \frac{\partial L}{\partial y_{s,oc,i,j}} w_{oc,ic,i',j'}
\end{aligned}
$$

Convolution (Back propagation 2)

- $\frac{\partial L}{\partial w}$

$$\begin{aligned}\frac{\partial L}{\partial w_{oc,ic,i',j'}} &= \sum_{s,oc',i,j} \frac{\partial L}{\partial y_{s,oc',i,j}} \frac{\partial y_{s,oc',i,j}}{\partial w_{oc,ic,i',j'}} \\ &= \sum_{s,i,j} \frac{\partial L}{\partial y_{s,oc,i,j}} x_{s,ic,i+i',j+j'}\end{aligned}$$

- $\frac{\partial L}{\partial b}$

$$\begin{aligned}\frac{\partial L}{\partial b_{oc}} &= \sum_{s,oc',i,j} \frac{\partial L}{\partial b_{oc}} \frac{\partial y_{s,oc',i,j}}{\partial b_{oc}} \\ &= \sum_{s,i,j} \frac{\partial L}{\partial y_{s,oc,i,j}}\end{aligned}$$

Linear (a.k.a. Fully Connected Layer)

- **definition:**

$$\begin{aligned} y = \text{Linear}(W; x) &\equiv Wx + b \\ \forall i \quad y_i &= \sum_j W_{ij} x_j + b_i \end{aligned}$$

Linear (Back Propagation 1)

- $\frac{\partial L}{\partial x}$

$$
\begin{aligned}
\frac{\partial L}{\partial x_j} &= \sum_{i'} \frac{\partial L}{\partial y_{i'}} \frac{\partial y_{i'}}{\partial x_j} \\
&= \sum_{i'} \frac{\partial L}{\partial y_{i'}} w_{i'j}
\end{aligned}
$$

Linear (Back Propagation 2)

- $\frac{\partial L}{\partial W}$

$$\begin{aligned}\frac{\partial L}{\partial W_{ij}} &= \sum_{i'} \frac{\partial L}{\partial y_{i'}} \frac{\partial y_{i'}}{\partial W_{ij}} \\ &= \frac{\partial L}{\partial y_i} x_j\end{aligned}$$

- $\frac{\partial L}{\partial b}$

$$\begin{aligned}\frac{\partial L}{\partial b_i} &= \sum_{i'} \frac{\partial L}{\partial y_{i'}} \frac{\partial y_{i'}}{\partial b_i} \\ &= \frac{\partial L}{\partial y_i}\end{aligned}$$

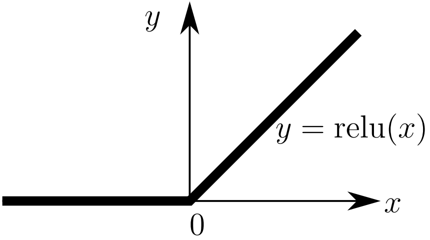
ReLU

- **definition (scalar ReLU):** for $x \in R$, define

$$\mathrm{relu}(x) \equiv \max(x, 0)$$

- **derivatives of relu:** for $y = \mathrm{relu}(x)$,

$$\frac{\partial y}{\partial x} = \begin{cases} 1 & (x > 0) \\ 0 & (x \leq 0) \end{cases} = \max(\mathrm{sign}(x), 0)$$



ReLU

- **definition (vector ReLU):** for a vector $x \in R^n$, define ReLU as the application of relu to each component

$$\mathrm{ReLU}(x) \equiv \begin{pmatrix} \mathrm{relu}(x_0) \\ \vdots \\ \mathrm{relu}(x_{n-1}) \end{pmatrix}$$

- **derivatives of ReLU:**

$$\frac{\partial y_j}{\partial x_i} = \begin{cases} \max(\mathrm{sign}(x_i), 0) & (i = j) \\ 0 & (i \neq j) \end{cases}$$

ReLU

- **back propagation:**

$$\begin{aligned}\frac{\partial L}{\partial x_j} &= \sum_i \frac{\partial L}{\partial y_i}\frac{\partial y_i}{\partial x_j} \\ &= \frac{\partial L}{\partial y_j}\frac{\partial y_j}{\partial x_j} \\ &= \begin{cases} \dfrac{\partial L}{\partial y_j} & (x_j \geq 0) \\ 0 & (x_j < 0) \end{cases}\end{aligned}$$

softmax

- **definition:** for $x \in R^n$

$$y = \text{softmax}(x) \quad \equiv \quad \frac{1}{\sum_{i=0}^{n-1} \exp(x_j)} \begin{pmatrix} \exp(x_0) \\ \vdots \\ \exp(x_{n-1}) \end{pmatrix}$$

it is a vector whose:

- each component > 0,
- sum of all components $= 1$
- largest component "dominates"

softmax 1.0

log softmax

$$
\begin{aligned}
y &= \log(\mathrm{softmax}(x)) \\
&= \begin{pmatrix} x_0 - \log \sum_{i=0}^{n-1} \exp(x_i) \\ \vdots \\ x_{n-1} - \log \sum_{i=0}^{n-1} \exp(x_i) \end{pmatrix}
\end{aligned}
$$

- (recall

$$
\mathrm{softmax}(x) \equiv \frac{1}{\sum_{i=0}^{n-1} \exp(x_j)} \begin{pmatrix} \exp(x_0) \\ \vdots \\ \exp(x_{n-1}) \end{pmatrix}
$$

)

NLL

- **definition:**
  - x : n-vector
  - t : true class of the data

$$\mathrm{NLL}(x, t) \equiv -\log x_t$$

- thus,

$$\begin{aligned} y &= \mathrm{NLL}(\mathrm{softmax}(x), t) \\ &= -x_t + \log \sum_{i=0}^{n-1} \exp(x_i) \end{aligned}$$

NLL softmax (Back propagation)

- $$\begin{aligned}
\frac{\partial L}{\partial x_i} &= \frac{\partial L}{\partial y}\frac{\partial y}{\partial x_i} \\
&= \begin{cases} \frac{\partial L}{\partial y}(-1 + \frac{\exp(x_i)}{\sum_{i=0}^{n-1}\exp(x_i)}) & (i = t) \\ \frac{\partial L}{\partial y}\frac{\exp(x_i)}{\sum_{i=0}^{n-1}\exp(x_i)} & (i \neq t) \end{cases} \\
&= \begin{cases} \frac{\partial L}{\partial y}\ (-1 + \mathrm{softmax}(x_i)) & (i = t) \\ \frac{\partial L}{\partial y}\ \mathrm{softmax}(x_i) & (i \neq t) \end{cases}
\end{aligned}$$

Note: why NLL softmax?

- recall that for n-way classification, the output of $p = \text{softmax}(\dots)$ is an n-vector
- p_i is meant to be the *probability* that a particular sample belongs to the class i
- for that purpose, a loss function could be any function that decreases with p_t (something as simple as $-p_t$), where t is the true label of the particular sample
- we isntead use $\text{NLL}(p, t) = -\log p_t$. why?

Note: why NLL log softmax?

- this is because,
  1. the goal is to maximize the joint probability of the entire data, which is the *product* of probabilities of individual samples:
  $$\Pi_k p_{t_k},$$
  where t_k is the true label of sample k, and
  2. the loss over a mini-batch is the *sum* of losses of individual samples
- they can be reconciled by setting the loss function to $-\log p_t$

$$\sum_k (-\log p_{t_k}) = -\log (\Pi_k p_{t_k})$$