Neural Networks
basics and parallelization/vectorization

Kenjiro Taura

Contents

Contents

What is machine learning?

- input: a set of *training data set*

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

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.

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

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

Machine learning as an optimization problem

- finding a good function from the space of literally all possible functions is neither easy nor meaningful
- we normally fix a search space of functions ($\mathcal{F}$) 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 normally fix a search space of functions ($\mathcal{F}$) 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:

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

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 somewhat more realistic example: image (digits) 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 : a "one hot" vector representing the class $\in \{0, 1, \cdots, 9\}$ (e.g. $\sim {}^t(0\ 0\ 0\ 0\ 1\ 0\ 0\ 0\ 0\ 0)$ represents "4")
  - we write $\mathbf{i}$ to mean the hot vector v having $v_i = 1$

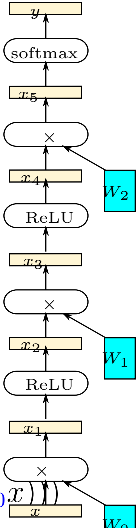
$D = \{($  $, \mathbf{4}), ($  $, \mathbf{9}), \ldots\}$

A somewhat more realistic example: image (digits) 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 : a "one hot" vector representing the class $\in \{0, 1, \cdots, 9\}$ (e.g. $\sim {}^t(0\ 0\ 0\ 0\ 1\ 0\ 0\ 0\ 0\ 0)$ represents "4")
  - we write $\mathbf{i}$ to mean the hot vector v having $v_i = 1$

$D = \{($$, \mathbf{4}), ($$, \mathbf{9}), \ldots\}$

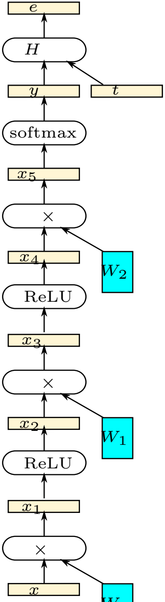
- the search space: the following composition parameterized by three matrices W_0, W_1 and W_2



$$f_{W_0,W_1,W_2}(x) \equiv \mathrm{softmax}(W_2\,\mathrm{ReLU}(W_1\,\mathrm{ReLU}(W_0 x)))$$

A handwritten digits recognition

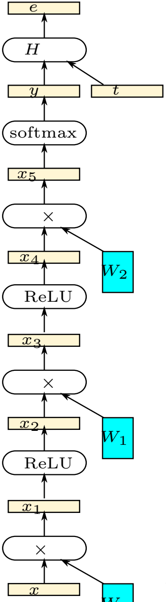
- the value $f_{W_0,W_1,W_2}(x)$ is a 10-vector representing probabilities that x belongs to each of the ten classes



A handwritten digits recognition

- the value $f_{W_0,W_1,W_2}(x)$ is a 10-vector representing probabilities that x belongs to each of the ten classes
- a loss function is the cross entropy commonly used in multiclass classifications ($\cdot$: a dot product)

$$\mathrm{err}(y,t) = H(t,y) \equiv -t \cdot \log y$$



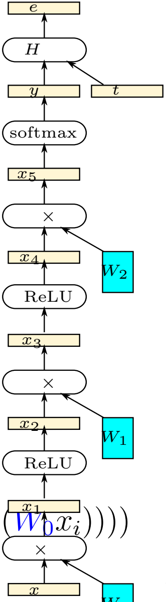
A handwritten digits recognition

- the value $f_{W_0,W_1,W_2}(x)$ is a 10-vector representing probabilities that x belongs to each of the ten classes
- a loss function is the cross entropy commonly used in multiclass classifications ($\cdot$: a dot product)

$$\mathrm{err}(y, t) = H(t, y) \equiv -t \cdot \log y$$

- the task is to find W_0, W_1 and W_2 that minimizes:

$$\begin{aligned} & L({\color{blue}W_0, W_1, W_2}; D) \\ = & \sum_{(x_i,t_i)\in D} H(\mathrm{softmax}(t_i, {\color{blue}W_2}\ \mathrm{ReLU}({\color{blue}W_1}\ \mathrm{ReLU}({\color{blue}W_0}\, x_i)))) \end{aligned}$$



Contents

Contents

How to find the minimizing parameter?

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

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

How to find the minimizing parameter?

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

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

- for which we compute a 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 scalar controlling a learning rate)

How to find the minimizing parameter?

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

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

- for which we compute a 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 scalar controlling a learning rate)

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

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:

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

- the gradient descent method just described:
  1. computes $\frac{\partial}{\partial w}\mathrm{err}(f_w(x_i), t_i)$ 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

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 $\frac{\partial}{\partial w} \mathrm{err}(f_w(x_i), t_i)$ 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

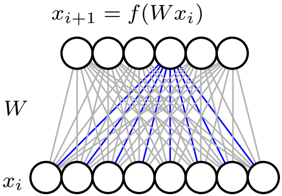
$$w = w - \eta^t \frac{\partial L(w; D')}{\partial w}$$

4. "update sooner rather than later"

SGD and neural networks

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

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

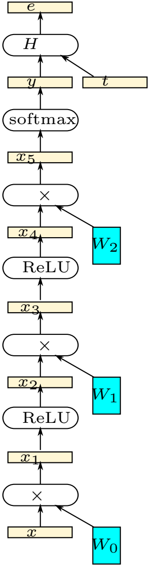


- we need to differentiate e by $w_1, \cdots, w_n$

The digits recognition example

$$
\begin{aligned}
x_1 &= W_0 x \\
x_2 &= \mathrm{ReLU}(x_1) \\
x_3 &= W_1 x_2 \\
x_4 &= \mathrm{ReLU}(x_3) \\
x_5 &= W_2 x_4 \\
y &= \mathrm{softmax}(x_5) \\
e &= H(y, t)
\end{aligned}
$$

you need to get differentiation of e by W_0, W_1 and W_2 done right



Contents

Differentiating multivariable functions

- $x = {}^t(x_0 \ \cdots \ x_{n-1}) \in R^n$ (a column vector)
- $f(x)$: a scalar
- **definition:** a derivative of f with respect to x, written $f'(x)$ or $\dfrac{\partial f}{\partial x}$, is a row n-vector a s.t.

$$\begin{aligned} f(x + \Delta x) &\approx f(x) + a\Delta x \\ &= f(x) + \sum_{i=0}^{n-1} a_i \Delta x_i \end{aligned}$$

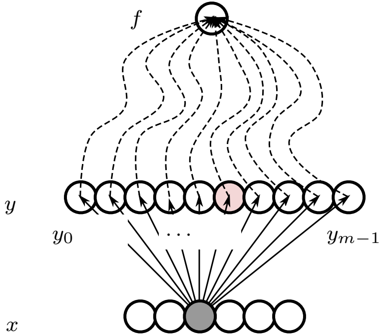
  (a row vector $\times$ a column vector, yielding a 1×1 matrix, identified with a scalar)
- when it exists,

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

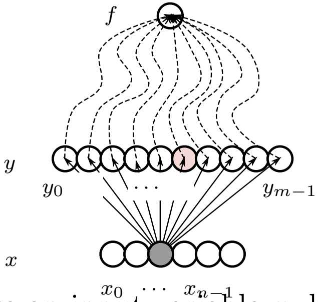
The Chain Rule

- 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

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

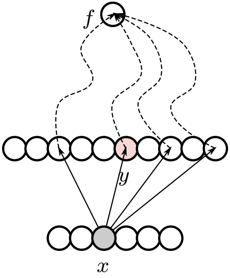
which will contribute to increasing the final output (f) by

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

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

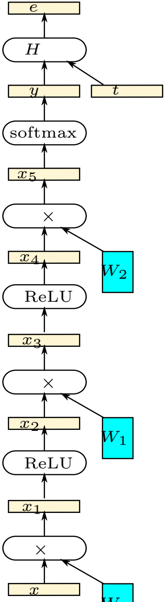
$$\frac{\partial L}{\partial x},$$

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 used the following functions

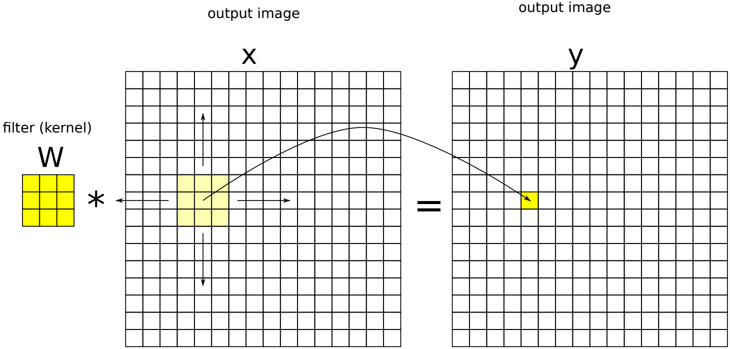
- Convolution$(W; x)$: linear/local transformations to images
- Linear$(W; x)$: linear or "fully connected" layer
- ReLU(x) : rectified linear units
- softmax(x)
- $H(t, x)$: cross entropy

we summarize their definitions and their derivatives

Convolution

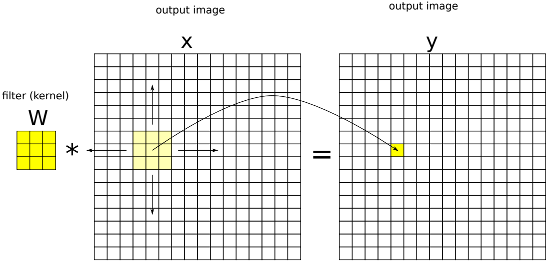
- it takes
  - an image = 2D pixels × a number of channels
  - a "filter" or a "kernel", which is essentially a small image

  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 $(-K \leq i \leq K, -K \leq j \leq K)$
- $x_{i,j}$: an input image $(0 \leq i < H, 0 \leq j < W)$
- $y_{i,j}$: an output image $(0 \leq i < H, 0 \leq j < W)$



- assuming $x_{i,j} = 0$ for underflowed/overflowed indices for brevity,

$$y_{i,j} = \sum_{-K \leq i' \leq K, -K \leq j' \leq K} w_{i',j'} x_{i+i',j+j'}$$

(for each i, j)

Convolution

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

$$y_{oc,i,j} = \sum_{ic,i',j'} w_{oc,ic,i',j'} x_{ic,i+i',j+j'} \quad \text{(for each } oc, i, j\text{)}$$

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

$$y_{b,oc,i,j} = \sum_{ic,i',j'} w_{oc,ic,i',j'} x_{b,ic,i+i',j+j'} \quad \text{(for each } b, oc, i, j\text{)}$$

Convolution (Back propagation)

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

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

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

- **definition:**

$$\begin{aligned} y = \mathrm{Linear}(W; x) &\equiv Wx \\ y_i &= \sum_j W_{ij} x_j \end{aligned}$$

- **derivatives:**
  - by W

$$\frac{\partial y_{i'}}{\partial W_{ij}} = \begin{cases} x_j & (i' = i) \\ 0 & (i' \neq i) \end{cases}$$

  - by x

$$\frac{\partial y_i}{\partial x_j} = w_{ij}$$

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

- **back propagation:**
  - $\frac{\partial L}{\partial W}$

$$\begin{aligned}\frac{\partial L}{\partial W_{ij}} &= \sum_{i'} \frac{\partial L}{\partial y_{i'}} \frac{\partial y_{i'}}{\partial W_{ij}} \\ &= \frac{\partial L}{\partial y_i} x_j \\ \frac{\partial L}{\partial W} &= \frac{\partial L}{\partial y} \times x \text{ (outer product)}\end{aligned}$$

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

$$\begin{aligned}\frac{\partial L}{\partial x_j} &= \sum_{i} \frac{\partial L}{\partial y_i} \frac{\partial y_i}{\partial x_j} \\ &= \sum_{i} w_{ij} \frac{\partial L}{\partial y_i} \\ \frac{\partial L}{\partial x} &= \frac{\partial L}{\partial y} W \text{(vector-matrix product)}\end{aligned}$$

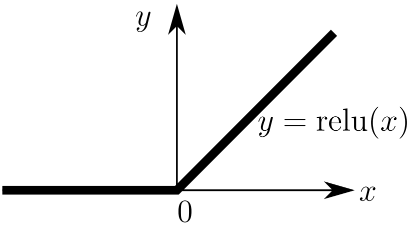
ReLU

- **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:**

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

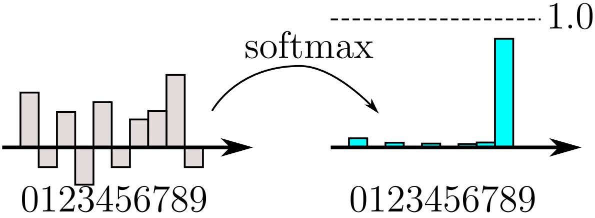
softmax

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

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

it is a vector whose:

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



Derivative of softmax

- for $y = \text{softmax}(x)$, $\left(\dfrac{\partial y}{\partial x}\right)$ is an $n \times n$ matrix whose elements are given by:
  - (diagonal elements)

$$\begin{aligned}\left(\frac{\partial y}{\partial x}\right)_{i,i} &= \frac{\partial}{\partial x_i}\frac{\exp(x_i)}{\sum_k \exp(x_k)}\\ &= \frac{\exp(x_i)\sum_k \exp(x_k) - \exp(x_i)^2}{\left(\sum_k \exp(x_k)\right)^2}\\ &= {\color{blue}y_i(1-y_i)}\end{aligned}$$

  - (non-diagonal elements) for $i \neq j$,

$$\begin{aligned}\left(\frac{\partial y}{\partial x}\right)_{i,j} &= \frac{\partial}{\partial x_j}\frac{\exp(x_i)}{\sum_k \exp(x_k)}\\ &= -\exp(x_i)\frac{\exp(x_j)}{\left(\sum_k \exp(x_k)\right)^2}\\ &= {\color{blue}-y_i y_j}\end{aligned}$$

Cross entropy

- **definition:** for $y \in R^n$,

$$H(t, y) \equiv -\sum_{i=0}^{n-1} t_i \log y_i$$

- **derivative of H:** for $z = H(t, y)$, $\dfrac{\partial z}{\partial y}$ is an n-vector

$$\frac{\partial z}{\partial y} = -\begin{pmatrix} \frac{t_0}{y_0} & \cdots & \frac{t_{n-1}}{y_{n-1}} \end{pmatrix}$$

- if t is a one hot vector, so is this vector; if $t = \mathbf{c}$,

$$\begin{aligned} \frac{\partial z}{\partial y} &= -\begin{pmatrix} 0 & \cdots & 0 & \frac{1}{y_c} & 0 & \cdots & 0 \end{pmatrix} \\ &= -\frac{\mathbf{c}}{y_c} \end{aligned}$$

Composition of softmax and cross entropy

In particular, composition of softmax and H enjoy a remarkable simplification when differentiated:

- for $z = H(t, y) = H(t, \mathrm{softmax}(x))$,

$$
\begin{aligned}
\frac{\partial z}{\partial x} &= \frac{\partial z}{\partial y}\frac{\partial y}{\partial x} \\
&= -\frac{\mathbf{c}}{y_c}\frac{\partial y}{\partial x} \\
&= -\frac{1}{y_c}\frac{\partial y_c}{\partial x} \\
&= \frac{1}{y_c}\begin{pmatrix} y_0 y_c & y_1 y_c & \cdots & y_c(1-y_c) & y_{c+1} y_c & \cdots & y_{n-1} y_c \end{pmatrix} \\
&= \begin{pmatrix} y_0 & y_1 & \cdots & (y_c - 1) & y_{c+1} & \cdots & y_{n-1} \end{pmatrix} \\
&= {}^t(y - t)
\end{aligned}
$$