Deep Learning: Part 2

東京海洋大学 TUMSAT

竹縄知之 Tomoyuki Takenawa

目次

1 Forward propagation in neural networks

In this section, you will study that a neural network is a network model that outputs a prediction $\hat{y}$ for input data $\mathbf{x} = [x_0, x_1, \cdots, x_{D-1}]$ via several layers of computation, and that the computation in each layer is basically done by linear transformation (or more precisely, affine transformation) and activation function.

1.1 Forward propagation in an affine layer

Each layer of a neural network consists of multiple neurons (vertices of a graph), and the operation in the lth neuron of the ith layer is a linear transformation (or, more precisely, an affine transformation, since there is also the operation of adding a constant) followed by a nonlinear transformation as

$$\begin{cases} a_l^{(i)} &= x_0^{(i-1)} w_{0l}^{(i)} + x_1^{(i-1)} w_{1l}^{(i)} + \cdots + x_{K-1}^{(i-1)} w_{K-1,l}^{(i)} + b_l^{(i)} \\ x_l^{(i)} &= f\left(a_l^{(i)}\right) \end{cases} .$$

Here, $\mathbf{x}^{(i-1)} = (x_0^{(i-1)}, \cdots, x_{K-1}^{(i-1)})$ is the output value of the previous layer, $w_{kl}^{(i)}$ and $b_l^{(i)}$ are called the weight parameters of the model, and $f(a)$ is a nonlinear function called the activation function.

Using (K, L) dimensional matrix $W^{(i)} = \left(w_{kl}^{(i)}\right)$ and L dimensional vector $\mathbf{b}^{(i)} = \left(b_l^{(i)}\right)$, the above equation can be written as

$$\begin{cases} \mathbf{a}^{(i)} &= \mathbf{x}^{(i-1)} W^{(i)} + \mathbf{b}^{(i)} \\ \mathbf{x}^{(\mathbf{i})} &= f\left(\mathbf{a}^{(i)}\right) \end{cases} .$$

Here, it is assumed that the vector is a horizontal vector and the function f acts on each component of $\mathbf{a}^{(i)}$ (such an action is called broadcasting).

In the case of mini-batch

Let's consider the case of mini-batch learning with N data.

Let the set of output values of the previous layer be an array of type (N, K): $X^{(i-1)} = \begin{bmatrix} x_{0,0}^{(i-1)} & \cdots & x_{0,K-1}^{(i-1)} \\ \vdots & \cdots & \vdots \\ x_{N-1,0}^{(i-1)} & \cdots & x_{N-1,K-1}^{(i-1)} \end{bmatrix}$, then the forward propagation of the affine layer can be written as

$$\begin{cases} A^{(i)} &= X^{(i-1)}W^{(i)} + \mathbf{b}^{(i)} \\ X^{(i)} &= f\left(A^{(i)}\right) \end{cases} .$$

Here, again $W^{(i)}$ is an array of type (K, L) and $\mathbf{b}^{(i)}$ is an array of type $(L,)$. In this case, $A^{(i)}$ and $X^{(i)}$ are arrays of type (N, L).

Note that in NumPy, it is broadcasted as

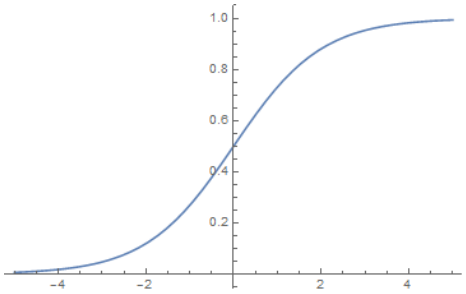
$$(N, L)\text{type array} + (L,)\text{type array} = (N, L)\text{type array}$$

(the second term viewed as a row vector is added to all the rows of the first term). Also note that the activation function acts component-wise.

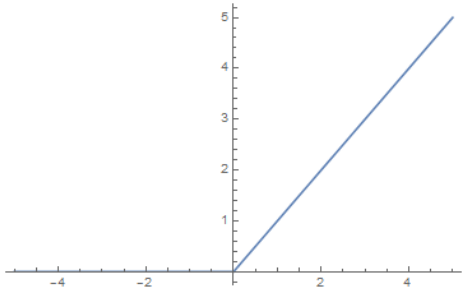
1.2 Activation function

The following two activation functions are typical.

Sigmoid function $\sigma(x) = \dfrac{1}{1+e^{-x}}$

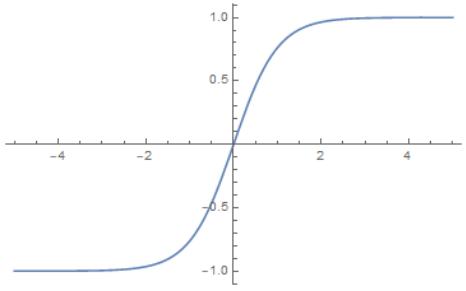


ReLU (Rectified Linear Unit) function
$\mathrm{ReLU}(x) = \max\{0, x\}$

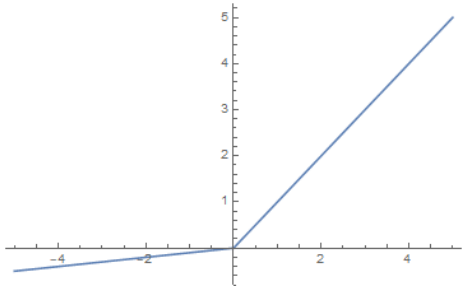


The following functions are also used depending on the situation.

hyperbolic tangent
$\tanh(x) = \dfrac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$



Leaky ReLU function
$$f(x) = \begin{cases} x & (x > 0) \\ 0.1x & (x \leq 0) \end{cases}$$



2 Design of loss function and output layer

The loss function or cost function is a measure of the "badness" of a machine learning model. The loss function is essentially the same as the error function for measuring the accuracy of predictions, but sometimes the loss function can include penalties such as the magnitude of parameters. However, for the sake of simplicity, we will consider only the error function as the loss function for a while.

In order to train a neural network, we need to design the error function appropriately. First, of course, the error function must be large when the accuracy of the prediction is low, and small when it is high. Also, it must be differentiable by the parameters of the model in order to be trainable. In the case of supervised learning, the error function is a function of the correct answer value y and the predicted value $\hat{y}$.

In this section, we introduce a typical output layer and loss function (error function) for supervised learning of regression and classification problems.

2.1 In case of a regression problem

Let the output value be $\hat{y}$ and the correct answer label be y. It is common to use (one half of) the squared error

$$\frac{1}{2}(\hat{y} - y)^2$$

as the error function. No activation function is used in the output layer.

In the case of mini-batch training with N data, when the set of output values is $\hat{Y} = [\hat{y}^{(0)}, \cdots, \hat{y}^{(N-1)}]$ and the set of correct labels is $Y = [y^{(0)}, \cdots, y^{(N-1)}]$, (one half of) the mean squared error

$$\frac{1}{2N} \sum_{n=0}^{N-1} \left(\hat{y}^{(n)} - y^{(n)} \right)^2$$

is used.

2.2 In case of a 0-1 binary classification problem

In the case of a binary problem where the correct answer is 0 or 1, using the sigmoid function as the activation function of the output layer $\hat{y} = \sigma(x) = \dfrac{1}{1+e^{-x}}$ and setting $P(y=1) = \hat{y}$ and $P(y=0) = 1-\hat{y}$, we can consider that output is a Bernoulli distribution (a probability distribution where the probability of 1 is p and the probability of 0 is $1-p$).

It is common to use the cross-entropy of the output value $\hat{y}$ to the correct label y:

$$H(y, \hat{y}) = -y \log \hat{y} - (1-y) \log(1-\hat{y})$$

as the error function.

In the case of mini-batch learning with N data, when the set of output values is $\hat{Y} = [\hat{y}^{(0)}, \cdots, \hat{y}^{(N-1)}]$ and the set of correct answer labels is $Y = [y^{(0)}, \cdots, y^{(N-1)}]$, we use the average cross-entropy

$$H(Y, \hat{Y}) = -\frac{1}{N} \sum_{n=0}^{N-1} \left(y^{(n)} \log \hat{y}^{(n)} + (1-y^{(n)}) \log(1-\hat{y}^{(n)}) \right)$$

as the error function.

2.3 When classifying into K types

If we apply a softmax function

$$\begin{bmatrix}\hat{y}_0 & \cdots & \hat{y}_{K-1}\end{bmatrix} = \text{softmax}\left(\begin{bmatrix}x_0 & \cdots & x_{K-1}\end{bmatrix}\right) = \frac{1}{e^{x_0}+\cdots+e^{x_{K-1}}}\begin{bmatrix}e^{x_0} & \cdots & e^{x_{K-1}}\end{bmatrix}$$

as the activation function of the output layer so that each component is greater than or equal to 0 and the sum is 1, we can consider the output to be a categorical distribution with K elementary events (a distribution in which the probability $p_0, \cdots, p_{K-1}$ of each event is given for mutually exclusive K events).

It is common to use the cross-entropy of this output value $\hat{\mathbf{y}} = \begin{bmatrix}\hat{y}_0 & \cdots & \hat{y}_{K-1}\end{bmatrix}$ to the correct label $\mathbf{y} = \begin{bmatrix}y_0 & \cdots & y_{K-1}\end{bmatrix}$ (one-hot representation, i.e., when the correct answer is l, $y_l = 1$ and $y_k = 0$ for $k \neq l$):

$$H(\mathbf{y}, \hat{\mathbf{y}}) = -\sum_{k=0}^{K-1} y_k \log \hat{y}_k$$

as the error function.

In the case of mini-batch training with N data, the average cross-entropy

$$H(Y, \hat{Y}) = -\frac{1}{N}\sum_{n=0}^{N-1}\sum_{k=0}^{K-1} y_k^{(n)} \log \hat{y}_k^{(n)}$$

is used as the error function.

Review of cross-entropy

Information content

Let $P(A)$ be the probability that an event A occurs. When event A occurs in a trial, the information content is

$$I(A) = -\log P(A).$$

Cross-entropy

Suppose two probability distributions, $P(A_k)$ and $Q(A_k)$ $(k = 0, 1, \cdots, K-1)$, where all events are mutually exclusive K events $A_0, A_1, \cdots, A_{K-1}$. Then,

$$H(P, Q) = -\sum_{k=0}^{K-1} P(A_k) \log Q(A_k)$$

is called the cross-entropy (of Q with respect to P). The cross-entropy is the expected value of the information content about the probability distribution Q with respect to the probability distribution P.

3 Backpropagation

In order to train a neural network (e.g., by stochastic gradient descent), we need to calculate the derivative with respect to the parameters of the error function. Calculating the derivative for a single parameter can be done easily by using numerical differentiation (shifting the parameter slightly and observing how the error changes). However, this method is very time consuming because it requires calculating each variable individually. Neural networks use the chain rule of differentiation to do this efficiently. This calculation is called the backpropagation method because it traces the network backwards from the error.

3.1 Stochastic gradient descent method and differentiation of errors

When the error of a neural network $\hat{\theta} = f(\mathbf{x}, \theta)$ with parameters θ is written as $E(\mathbf{x}, \theta)$, the update of θ by stochastic gradient descent is given by

$$\theta \leftarrow \theta - \lambda \frac{\partial E}{\partial \theta}$$

($\lambda > 0$ is a hyperparameter called the learning coefficient), where

$$\frac{\partial E}{\partial \theta} = \frac{1}{N} \sum_{n=0}^{N-1} \frac{\partial E(\mathbf{x}^{(n)}, \theta)}{\partial \theta}$$

($\mathbf{x}^{(0)}, \cdots, \mathbf{x}^{(N-1)}$ are input data in the mini-batch). Thus, we need to compute the partial derivative of the error with respect to the parameters. The backpropagation method is an efficient way to calculate this using the chain rule.

3.2 Chain rule

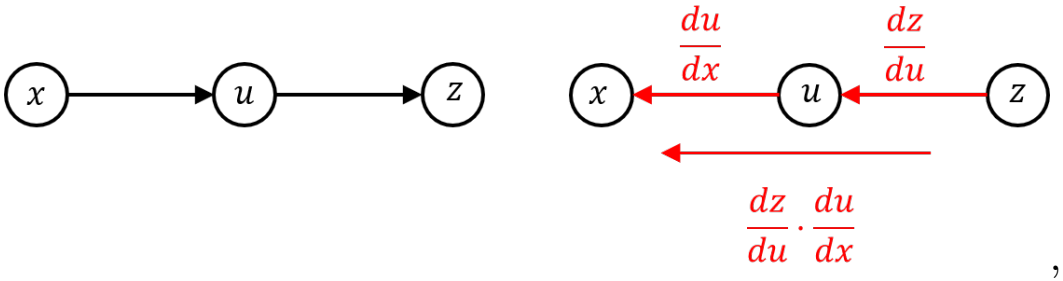
For simplicity, the function $y = f(x)$ is written as $y = y(x)$.

Differentiation of composite functions of one variable

If the functions $z = z(u)$ and $u = u(x)$ are both differentiable, then it holds that

$$\frac{dz}{dx} = \frac{dz}{du}\,\frac{du}{dx} = z'(u)\,u'(x)$$

When the dependency of the variables is represented by the diagram



,

the derivative formula of the composite function can be interpreted as multiplying the derivatives corresponding to the arrows.

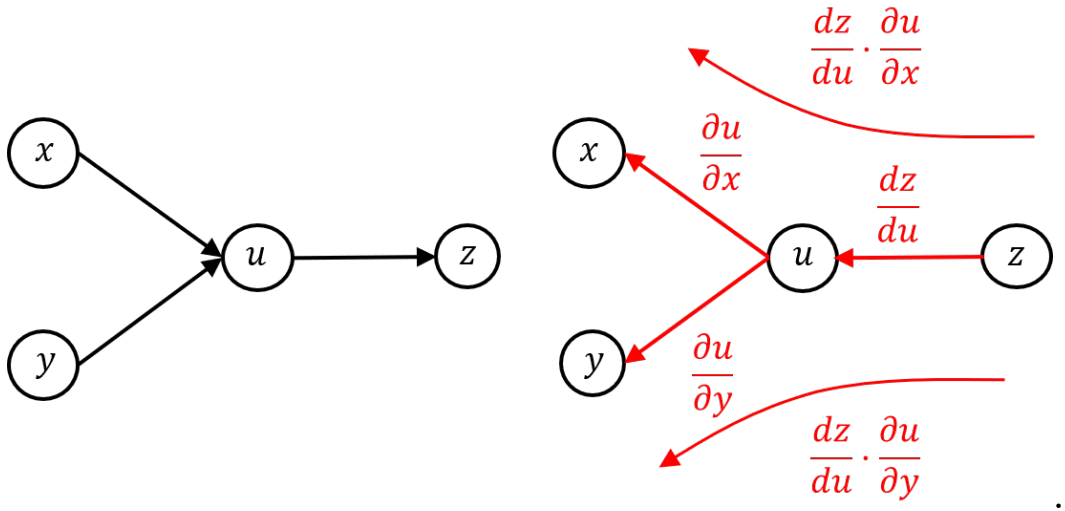
Differentiation of composite functions of two variables

Type 1:

If the function $z = z(u)$ is differentiable and $u = u(x, y)$ is partial differentiable, it holds that

$$\frac{\partial z}{\partial x} = \frac{dz}{du}\,\frac{\partial u}{\partial x} = z'(u)\,u_x(x,y) \quad \text{および} \quad \frac{\partial z}{\partial y} = \frac{dz}{du}\,\frac{\partial u}{\partial y} = z'(u)\,u_y(x,y).$$

The dependency of the variables is represented by the diagram

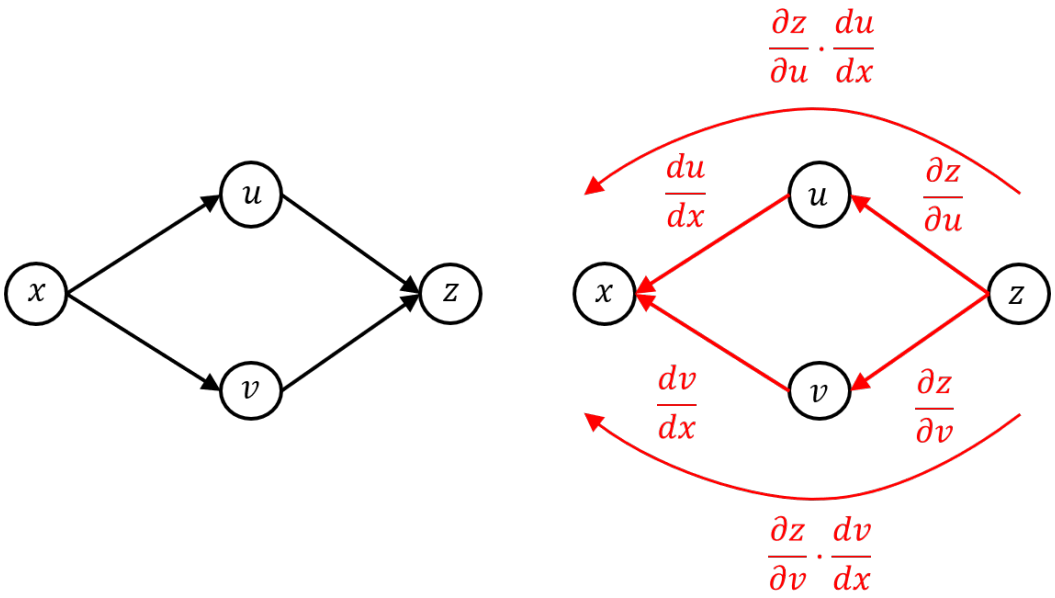


Again, the derivative formula for the composite function can be interpreted as multiplying the derivative along the arrows.

Type 2:
If the function $z = z(u, v)$ is totally differentiable and $u = u(x)$ and $v = v(x)$ are differentiable, it holds that

$$\frac{dz}{dx} = \frac{\partial z}{\partial u}\,\frac{du}{dx} + \frac{\partial z}{\partial v}\,\frac{dv}{dx} = z_u(u, v)\,u'(x) + z_v(u, v)\,v'(x).$$

The dependency of the variables is represented by the diagram



In this case, there are two routes that follow the arrow from x to z, and the derivative formula for the composite function can be interpreted as the sum of the derivatives along the arrow for each route.

Type 3:

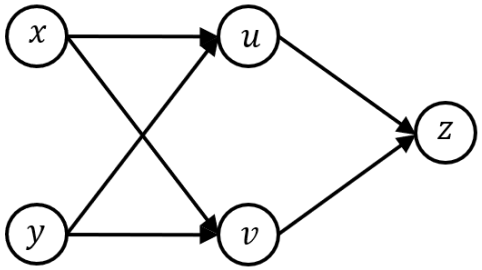
If the function $z = z(u, v)$ is totally differentiable and $u = u(x, y)$ and $v = v(x, y)$ are partial differentiable, it holds that

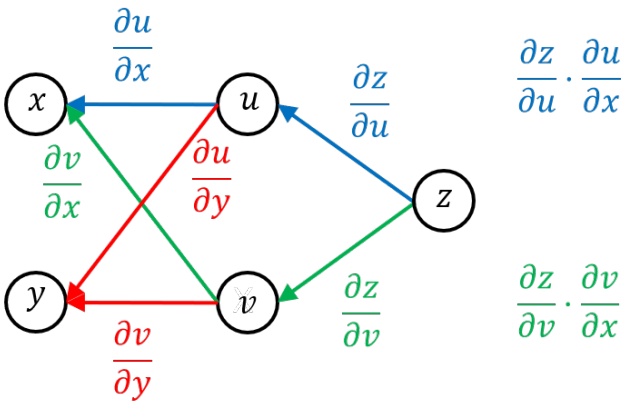
$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial u}\,\frac{\partial u}{\partial x} + \frac{\partial z}{\partial v}\,\frac{\partial v}{\partial x} = z_u(u, v)\,u_x(x, y) + z_v(u, v)\,v_x(x, y)$$

and

$$\frac{\partial z}{\partial y} = \frac{\partial z}{\partial u}\,\frac{\partial u}{\partial y} + \frac{\partial z}{\partial v}\,\frac{\partial v}{\partial y} = z_u(u, v)\,u_y(x, y) + z_v(u, v)\,v_y(x, y).$$

The dependency of the variables is represented by the diagram





In this case, there are two routes from x or y to z following the arrows, and the derivative formula for the composite function can still be interpreted as the sum of the derivatives along the arrows for each route.

3.3 Exercise on the chain rule ♯

(Do it if it's not too hard.)

(1) When $z = z(u)$, $u = \dfrac{1}{1+e^{-x}}$ (sigmoid function), represent $\dfrac{dz}{dx}$ using $\dfrac{dz}{du}$.

(2) In (1), when $z = -t \log u - (1-t)\log(1-u)$ (0-1 value cross-entropy), represent $\dfrac{dz}{dx}$ using t, u.

(3) When $z = z(u, v)$, $u = 1 + e^{x}$, $v = 1 + e^{-x}$ (input x is separated into u and v and then merges at z), represent $\dfrac{dz}{dx}$ using $\dfrac{\partial z}{\partial u}$ and $\dfrac{\partial z}{\partial v}$.

(4) When $z = z(u, v)$, $u = \dfrac{e^{x}}{e^{x}+e^{y}}$ and $v = \dfrac{e^{y}}{e^{x}+e^{y}}$ (softmax functions of two variables), represent $\dfrac{\partial z}{\partial x}$ and $\dfrac{\partial z}{\partial y}$ using $\dfrac{\partial z}{\partial u}$ and $\dfrac{\partial z}{\partial v}$.

(5) In (4), when $z = -s \log u - t \log v$, where $s + t = 1$ and $u + v = 1$ (cross-entropy of two variables), represent $\dfrac{\partial z}{\partial x}$ and $\dfrac{\partial z}{\partial y}$ using s, t, u, v.

(6) For real numbers w_{00}, w_{10}, w_{01}, w_{11}, b_0, b_1 and $z = z(y_0, y_1)$, $(y_0, y_1) = (x_0 w_{00} + x_1 w_{10} + b_0,\ x_0 w_{01} + x_1 w_{11} + b_1)$, represent $\dfrac{\partial z}{\partial x_0}$ and $\dfrac{\partial z}{\partial x_1}$ using $\dfrac{\partial z}{\partial y_0}$ and $\dfrac{\partial z}{\partial y_1}$.

(Correct answer)

(1) $\dfrac{dz}{dx} = \dfrac{e^{-x}}{(1+e^{-x})^2}\dfrac{dz}{du} = u(1-u)\dfrac{dz}{du}$

(2) Substituting $\dfrac{\partial z}{\partial u} = -\dfrac{t}{u} + \dfrac{1-t}{1-u}$ to the result of (1), we obtain $\dfrac{dz}{dx} = -t(1-u)+(1-t)u = u-t$

(3) $\dfrac{dz}{dx} = e^{x}\dfrac{\partial z}{\partial u} - e^{-x}\dfrac{\partial z}{\partial v}$

(4) From $\dfrac{\partial u}{\partial x} = \dfrac{\partial v}{\partial y} = \dfrac{e^{x+y}}{(e^x+e^y)^2} = uv$, $\dfrac{\partial u}{\partial y} = \dfrac{\partial v}{\partial x} = \dfrac{e^{x+y}}{(e^x+e^y)^2} = -uv$, we obtain
$\dfrac{\partial z}{\partial x} = uv\left(\dfrac{\partial z}{\partial u} - \dfrac{\partial z}{\partial v}\right)$, $\dfrac{\partial z}{\partial y} = uv\left(-\dfrac{\partial z}{\partial u} + \dfrac{\partial z}{\partial v}\right)$

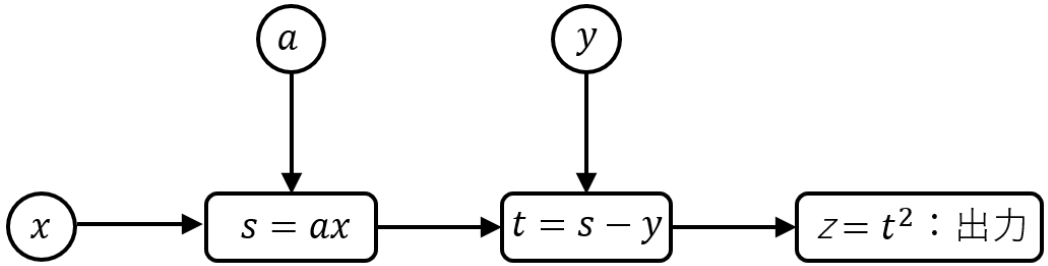
(5) Substituting $\dfrac{\partial z}{\partial u} = -\dfrac{s}{u}$, $\dfrac{\partial z}{\partial v} = -\dfrac{t}{v}$ to the result of (4) and using $s+t=1$ and $u+v=1$, we obtain $\dfrac{\partial z}{\partial x} = -vs+ut = -(1-u)s+u(1-s) = u-s$ and $\dfrac{\partial z}{\partial y} = vs-ut = v(1-t)-(1-v)t = v-t$

(6) $\dfrac{\partial z}{\partial x_0} = w_{00}\dfrac{\partial z}{\partial y_0} + w_{01}\dfrac{\partial z}{\partial y_1}$, $\dfrac{\partial z}{\partial x_1} = w_{10}\dfrac{\partial z}{\partial y_0} + w_{11}\dfrac{\partial z}{\partial y_1}$

3.4 Computational graph

The derivative of a complex function can be calculated using the chain rule by composing simple functions over and over again. The graph that represents this composition is called a computational graph. Each vertex (node) in the computational graph represents a function (and its output), and the arrows indicate that the output from the original vertex is passed on as input to the destination vertex. Tensorflow, PyTorch, and other deep learning frameworks provide the function to automatically apply the chain rule using a computational graph (automatic differentiation).

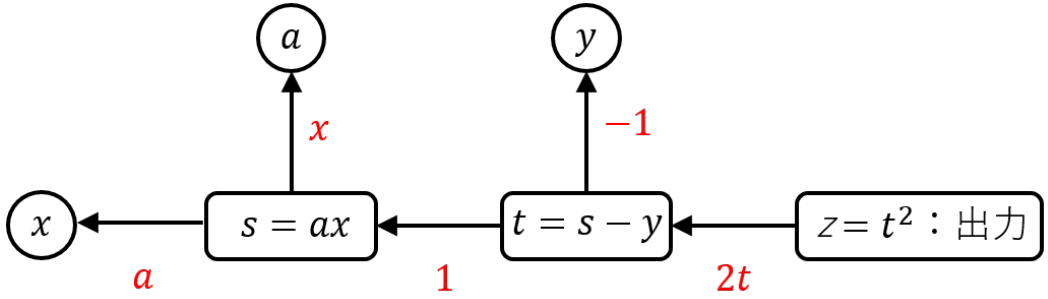
A simple example of a computational graph The function $z = (ax - y)^2$ of three real numbers x, a, and y can be decomposed into the following computational graphs



Here, from

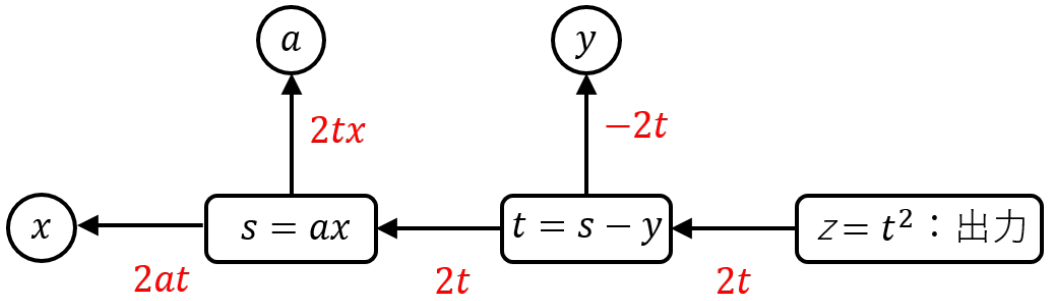
$$\frac{dz}{dt} = 2t, \quad \frac{\partial t}{\partial y} = -1, \quad \frac{\partial t}{\partial s} = 1, \quad \frac{\partial s}{\partial a} = x, \quad \frac{\partial s}{\partial x} = a,$$

we have the following inverse graph:



Here, the equation next to the arrow is the derivative of the variable at the origin of the arrow by the variable at the destination of the arrow. (Continued to the next page)

According to the chain rule, we should be able to get the derivative of the output z for each variable by multiplying them sequentially following the arrows. In fact, if we try this, we get



and

$$\frac{\partial z}{\partial x} = 2at = 2a(s-y) = 2a(ax-y)$$
$$\frac{\partial z}{\partial a} = 2xt = 2x(s-y) = 2x(ax-y)$$
$$\frac{\partial z}{\partial y} = -2t = -2(s-y) = -2(ax-y)$$

which is the correct result.

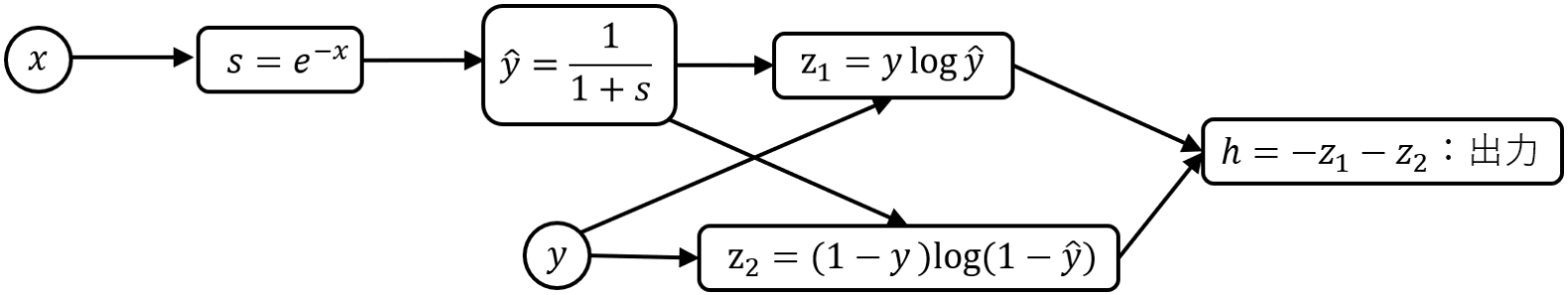
In this way, by using a computational graph, the derivative of the output is propagated backwards through the original graph.

A rather complex example of a computational graph

For two real numbers x, y (where $0 \leq y \leq 1$), consider the (binary) cross-entropy

$$h = -y \log \hat{y} - (1-y) \log(1-\hat{y}), (\text{where } \hat{y} = \frac{1}{1+e^{-x}})$$

with the output of the sigmoid function. This function can be docomposed as the following computational graph

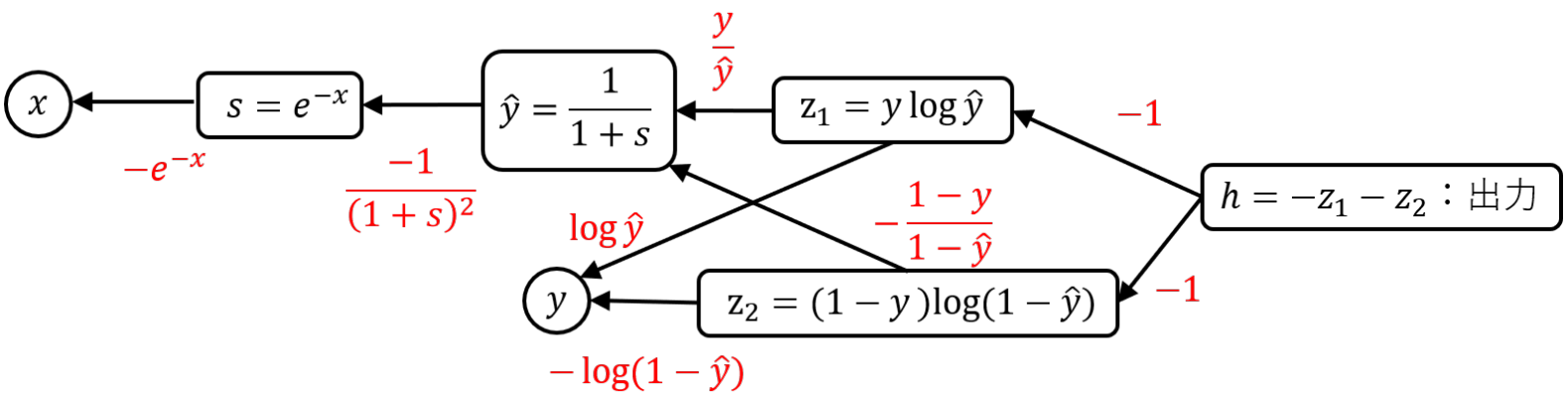


(It can be decomposed more finely, but the graph will be larger).

From

$$\frac{\partial h}{\partial z_1} = \frac{\partial h}{\partial z_2} = -1, \quad \frac{\partial z_1}{\partial y} = \log \hat{y}, \quad \frac{\partial z_1}{\partial \hat{y}} = \frac{y}{\hat{y}}, \quad \frac{\partial z_2}{\partial y} = -\log(1-\hat{y}),$$
$$\frac{\partial z_2}{\partial \hat{y}} = -\frac{1-y}{1-\hat{y}}, \quad \frac{d\hat{y}}{ds} = \frac{-1}{(1+s)^2}, \quad \frac{ds}{dx} = -e^{-x},$$

we have the graph in the opposite direction

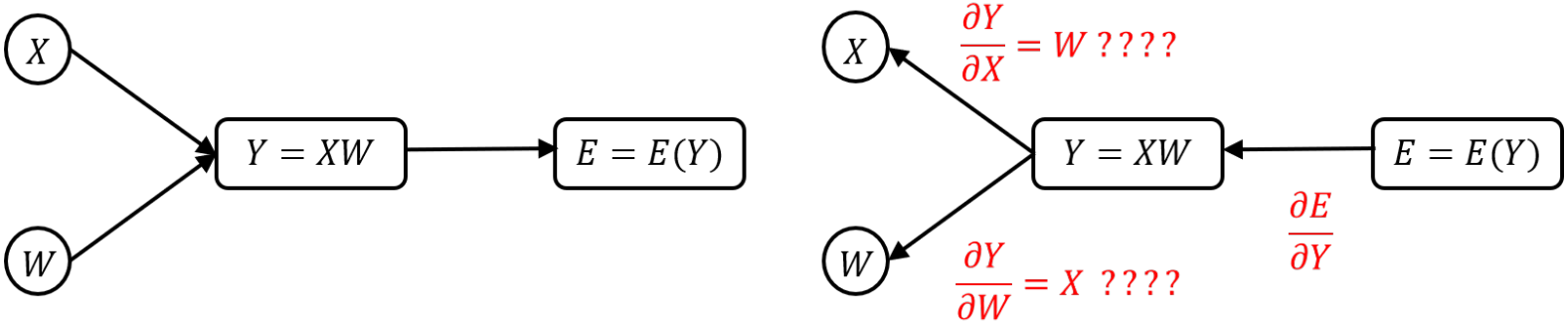


According to the chain rule, if the arrows merge, they should be added. Using $\hat{y} = \dfrac{1}{1+s}$ and $s = \dfrac{1-\hat{y}}{\hat{y}}$, we can calcurate as

$$\begin{aligned}
\frac{\partial h}{\partial y} &= -\log \hat{y} + \log(1-\hat{y}) = \log \frac{1-\hat{y}}{\hat{y}} = \log s = \log e^{-x} = -x \\
\frac{\partial z}{\partial x} &= \left(-\frac{y}{\hat{y}} + \frac{1-y}{1-\hat{y}}\right) \cdot \frac{-1}{(1+s)^2} \cdot (-e^{-x}) = \left(-\frac{y}{\hat{y}} + \frac{1-y}{1-\hat{y}}\right) \cdot \frac{s}{(1+s)^2} \\
&= \left(-\frac{y}{\hat{y}} + \frac{1-y}{1-\hat{y}}\right) \cdot \hat{y}(1-\hat{y}) \quad \text{(this deformation is somewhat technical.)} \\
&= \hat{y} - y.
\end{aligned}$$

3.5 Computational graphs for neural networks

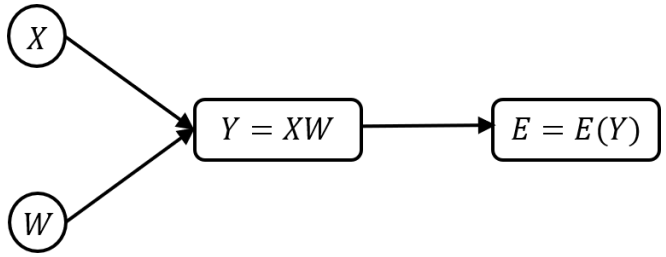
In neural networks, the entire layer can be thought of as a node, or vertex, in a computational graph. In this case, the input and output to the nodes of the computational graph are not a single number, but a higher-order array. Thinking of them together in this way makes it easier to implement backpropagation, but the application of the chaining rules at the nodes becomes more complex.



Let X and W be matrices and E a scalar, we want to calculate $\frac{\partial E}{\partial X}$ and $\frac{\partial E}{\partial W}$. However, while we can consider gradients for scalar matrices like $\frac{\partial E}{\partial Y}$, but we can not consider gradients of a matrix with respect to a matrix like $\frac{\partial Y}{\partial X}$ or $\frac{\partial Y}{\partial W}$.

A basic example in neural networks

As an example, Let's consider the node of the matrix product



where X, W, and Y are matrices and $E = E(Y)$ is the final error function of the model in one dimension. The goal of the backpropagation method is to find $\frac{\partial E}{\partial X}$ and $\frac{\partial E}{\partial W}$ when $\frac{\partial E}{\partial Y}$ is given

※ Symbols: For a matrix X and a function $f(X)$, the gradient $\frac{\partial f}{\partial X} = \nabla_X f$ represents a matrix of the same shape as X with (i, j) components as $\frac{\partial f}{\partial x_{ij}}$. The same is applied when X is a vector or tensor.

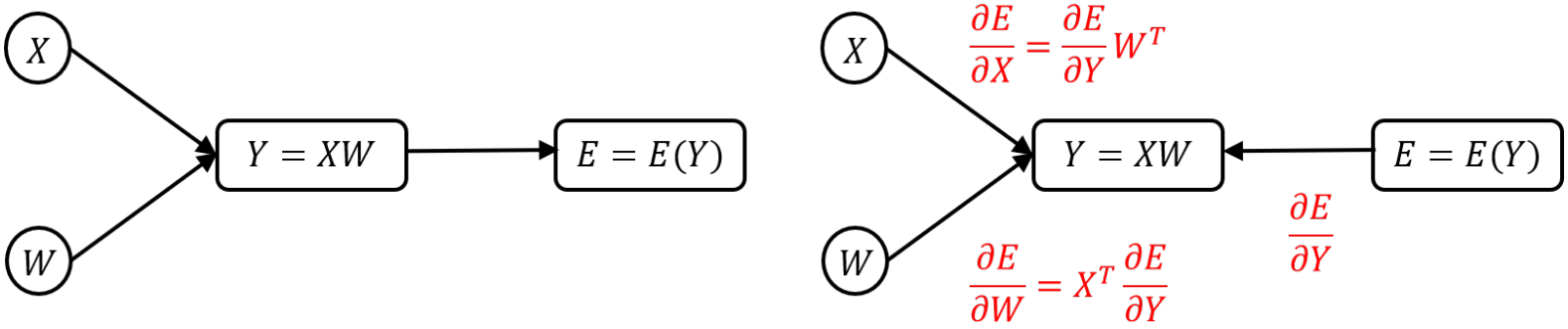
The conclusion is

$$\frac{\partial E}{\partial X} = \frac{\partial E}{\partial Y} W^T, \qquad \frac{\partial E}{\partial W} = X^T \frac{\partial E}{\partial Y}$$

We will derive this formula later.

A way to remember using dimensions

We introduced a formula



This formula is very hard to remember, but if you remember that the derivative of a product is the product of $\dfrac{\partial E}{\partial Y}$ and X or W, then it is automatically determined by just calculating the dimension.

In fact, if X is a matrix of shape (N, K) and W is a matrix of shape (K, L), then Y is of shape (N, L) and

$$\frac{\partial E}{\partial X} : \text{shape } (N, K), \quad \frac{\partial E}{\partial W} : \text{shape } (K, L), \quad \frac{\partial E}{\partial Y} : \text{shape } (N, L)$$

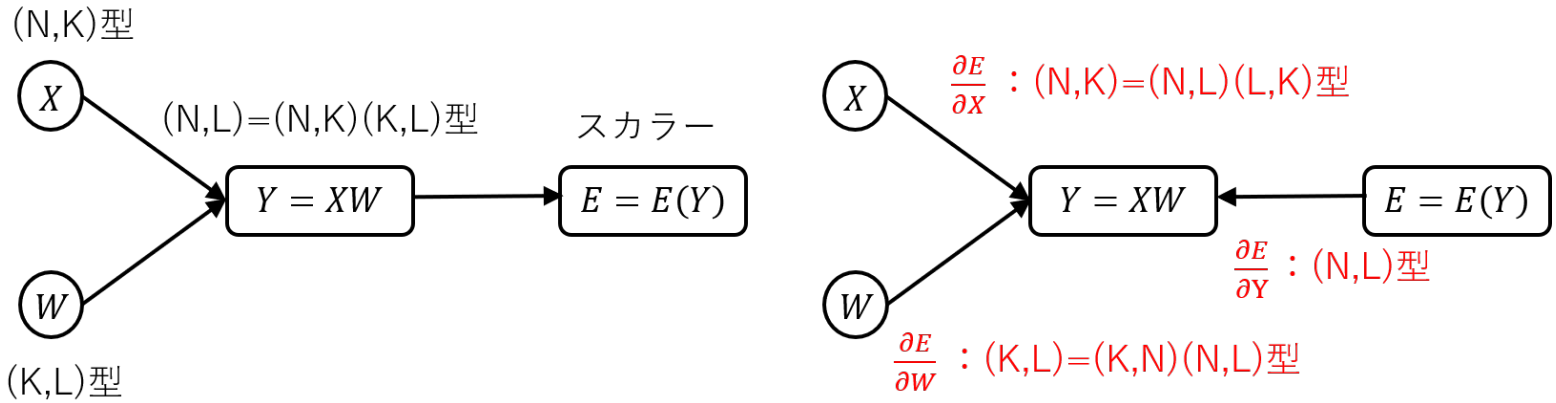
.

A way to remember using dimensions (continued)

For example, to get a matrix of shape (N, K) of $\dfrac{\partial E}{\partial X}$ by taking a product with $\dfrac{\partial E}{\partial Y}$: shape (N, L), since

$$(N, K) = (N, L) \times (L, K),$$

we have to multiply a variable of shape (L, K) from the right, which can only be done with W^T.



Problem Using the above method of dimension calculation, derive $\dfrac{\partial E}{\partial W} = X^T \dfrac{\partial E}{\partial Y}$.

Derivation of the formula

It's too complicated to do it in general shape, so let's calculate it when all matrices are of shape $(2, 2)$. From

$$\begin{bmatrix} y_{00} & y_{01} \\ y_{10} & y_{11} \end{bmatrix} = \begin{bmatrix} x_{00} & x_{01} \\ x_{10} & x_{11} \end{bmatrix} \begin{bmatrix} w_{00} & w_{01} \\ w_{10} & w_{11} \end{bmatrix} = \begin{bmatrix} x_{00}w_{00} + x_{01}w_{10} & x_{00}w_{01} + x_{01}w_{11} \\ x_{10}w_{00} + x_{11}w_{10} & x_{10}w_{01} + x_{11}w_{11} \end{bmatrix},$$

we have

$$\begin{aligned} \frac{\partial E}{\partial X} &= \begin{bmatrix} \frac{\partial E}{\partial x_{00}} & \frac{\partial E}{\partial x_{01}} \\ \frac{\partial E}{\partial x_{10}} & \frac{\partial E}{\partial x_{11}} \end{bmatrix} = \begin{bmatrix} \frac{\partial E}{\partial y_{00}} w_{00} + \frac{\partial E}{\partial y_{01}} w_{01} & \frac{\partial E}{\partial y_{00}} w_{10} + \frac{\partial E}{\partial y_{01}} w_{11} \\ \frac{\partial E}{\partial y_{10}} w_{00} + \frac{\partial E}{\partial y_{11}} w_{01} & \frac{\partial E}{\partial y_{10}} w_{10} + \frac{\partial E}{\partial y_{11}} w_{11} \end{bmatrix} \\ &= \begin{bmatrix} \frac{\partial E}{\partial y_{00}} & \frac{\partial E}{\partial y_{01}} \\ \frac{\partial E}{\partial y_{10}} & \frac{\partial E}{\partial y_{11}} \end{bmatrix} \begin{bmatrix} w_{00} & w_{10} \\ w_{01} & w_{11} \end{bmatrix} = \frac{\partial E}{\partial Y} W^T. \end{aligned}$$

Hence, it holds that

$$\frac{\partial E}{\partial X} = \frac{\partial E}{\partial Y} W^T$$

for $Y = XW$. Calcurating similarly, we have

$$\frac{\partial E}{\partial W} = X^T \frac{\partial E}{\partial Y}.$$

3.6 Exercise on backpropagation

1. Let $X = \begin{bmatrix} 2 & -1 & 1 \end{bmatrix}$, $W = \begin{bmatrix} 2 & 0 \\ 1 & 1 \\ 0 & 4 \end{bmatrix}$, $Y = XW$ and $E = E(Y)$ (E is one-dimensional variable).

   (1) When the error for the correct answer $(t_1, t_2) = (1, 2)$ is $E = \frac{1}{2}\left((y_1 - t_1)^2 + (y_2 - t_2)^2\right)$, find $\frac{\partial E}{\partial X}$ and $\frac{\partial E}{\partial W}$.

   (2) When the error for the correct answer $(t_1, t_2) = (1, 0)$ is $E = -t_1 \log\left(\frac{e^{y_1}}{e^{y_1} + e^{y_2}}\right) - t_2 \log\left(\frac{e^{y_2}}{e^{y_1} + e^{y_2}}\right)$, find $\frac{\partial E}{\partial X}$ and $\frac{\partial E}{\partial W}$.

2. Let $X = \begin{bmatrix} 1 & 3 \\ 2 & -1 \\ 0 & 1 \end{bmatrix}$, $\mathbf{b} = \begin{bmatrix} 2 & 1 \end{bmatrix}$, $Y = X + \mathbf{b} = \begin{bmatrix} 1+2 & 3+1 \\ 2+2 & -1+1 \\ 0+2 & 1+1 \end{bmatrix}$ (calculated by broadcast) and $E = E(Y)$ (E is one-dimensional variable). For $\dfrac{\partial E}{\partial Y} = \begin{bmatrix} 1 & 0 \\ 0 & -1 \ 1 \end{bmatrix}$, find $\dfrac{\partial E}{\partial X}$ and $\dfrac{\partial E}{\partial \mathbf{b}}$.

(Answer)

1. (1) from $Y = \begin{bmatrix} 3 & 3 \end{bmatrix}$, it holds that $\dfrac{\partial E}{\partial Y} = \begin{bmatrix} y_1 - t_1 & y_2 - t_2 \end{bmatrix} = \begin{bmatrix} 2 & 1 \end{bmatrix}$, and hence

$$\frac{\partial E}{\partial X} = \frac{\partial E}{\partial Y} W^T = \begin{bmatrix} 2 & 1 \end{bmatrix} \begin{bmatrix} 2 & 1 & 0 \\ 0 & 1 & 4 \end{bmatrix} = \begin{bmatrix} 4 & 3 & 4 \end{bmatrix}$$

$$\frac{\partial E}{\partial W} = X^T \frac{\partial E}{\partial Y} = \begin{bmatrix} 2 \\ -1 \\ 1 \end{bmatrix} \begin{bmatrix} 2 & 1 \end{bmatrix} = \begin{bmatrix} 4 & 2 \\ -2 & -1 \\ 2 & 1 \end{bmatrix}$$

(2) Exercise of the chain rule 3.3, it holds that $\dfrac{\partial E}{\partial Y} = \begin{bmatrix} \frac{e^{y_1}}{e^{y_1}+e^{y_2}} - t_1 & \frac{e^{y_2}}{e^{y_1}+e^{y_2}} - t_2 \end{bmatrix} = \begin{bmatrix} -\frac{1}{2} & \frac{1}{2} \end{bmatrix}$, and hence

$$\frac{\partial E}{\partial X} = \frac{\partial E}{\partial Y} W^T = \begin{bmatrix} -\frac{1}{2} & \frac{1}{2} \end{bmatrix} \begin{bmatrix} 2 & 1 & 0 \\ 0 & 1 & 4 \end{bmatrix} = \begin{bmatrix} -1 & 0 & 2 \end{bmatrix}$$

$$\frac{\partial E}{\partial W} = X^T \frac{\partial E}{\partial Y} = \begin{bmatrix} 2 \\ -1 \\ 1 \end{bmatrix} \begin{bmatrix} -\frac{1}{2} & \frac{1}{2} \end{bmatrix} = \begin{bmatrix} -1 & 1 \\ \frac{1}{2} & -\frac{1}{2} \\ -\frac{1}{2} & \frac{1}{2} \end{bmatrix}$$

2. From $\begin{bmatrix} y_{00} & y_{01} \\ y_{10} & y_{11} \\ y_{20} & y_{21} \end{bmatrix} = \begin{bmatrix} x_{00}+b_0 & x_{01}+b_1 \\ x_{10}+b_0 & x_{11}+b_1 \\ x_{20}+b_0 & x_{21}+b_1 \end{bmatrix}$, it holds that

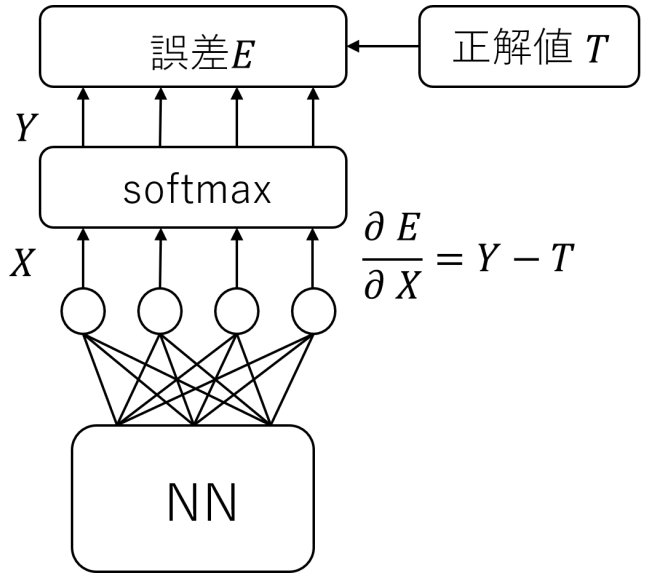
$$\frac{\partial E}{\partial X} = \begin{bmatrix} \frac{\partial E}{\partial x_{00}} & \frac{\partial E}{\partial x_{01}} \\ \frac{\partial E}{\partial x_{10}} & \frac{\partial E}{\partial x_{11}} \\ \frac{\partial E}{\partial x_{20}} & \frac{\partial E}{\partial x_{21}} \end{bmatrix} = \begin{bmatrix} \frac{\partial E}{\partial y_{00}} & \frac{\partial E}{\partial y_{01}} \\ \frac{\partial E}{\partial y_{10}} & \frac{\partial E}{\partial y_{11}} \\ \frac{\partial E}{\partial y_{20}} & \frac{\partial E}{\partial y_{21}} \end{bmatrix} = \frac{\partial E}{\partial Y} = \begin{bmatrix} 1 & 0 \\ 0 & -1 \\ 1 & 1 \end{bmatrix}$$

$$\begin{aligned} \frac{\partial E}{\partial \mathbf{b}} &= \begin{bmatrix} \frac{\partial E}{\partial b_0} & \frac{\partial E}{\partial b_1} \end{bmatrix} = \begin{bmatrix} \frac{\partial E}{\partial y_{00}} + \frac{\partial E}{\partial y_{10}} + \frac{\partial E}{\partial y_{20}} & \frac{\partial E}{\partial y_{01}} + \frac{\partial E}{\partial y_{11}} + \frac{\partial E}{\partial y_{21}} \end{bmatrix} \\ &= \sum_{i=0}^{2} \begin{bmatrix} \left(\frac{\partial E}{\partial Y}\right)_{i0} & \left(\frac{\partial E}{\partial Y}\right)_{i1} \end{bmatrix} = \mathrm{sum}\left(\begin{bmatrix} 1 & 0 \\ 0 & -1 \\ 1 & 1 \end{bmatrix}, \mathrm{axis} = 0 \right) = \begin{bmatrix} 2 & 0 \end{bmatrix} \end{aligned}$$

where, for an array X, $\mathrm{sum}(X, \mathrm{axis} = 0)$ is the sum with respect to the zeroth index (i.e., i when $X = (x_{ij})$) as in the NumPy function.

3.7 Summary of the backpropagation formula

For a single data



backpropagation for softmax with loss

- Output layer for regression: for the one half of squared error: $E = \frac{1}{2}(y-t)^2$, $\frac{\partial E}{\partial y} = y - t$ (t: correct answer value, y: predicted value)
- Output layer for 0-1 value 2-classification (sigmoid + cross entropy): $\frac{\partial E}{\partial x} = y - t$
- Output layer for K-classification (softmax + cross entropy): $\frac{\partial E}{\partial \mathbf{x}} = \mathbf{y} - \mathbf{t}$) ($\mathbf{t}$: vector of one-hot representations of the correct answer values)

- For an affine transformation layer $E = E(\mathbf{y})$, $\mathbf{y} = \mathbf{x}W + \mathbf{b}$:

$$\frac{\partial E}{\partial \mathbf{x}} = \frac{\partial E}{\partial \mathbf{y}} W^T, \quad \frac{\partial E}{\partial W} = \mathbf{x}^T \frac{\partial E}{\partial \mathbf{y}}, \quad \frac{\partial E}{\partial \mathbf{b}} = \frac{\partial E}{\partial \mathbf{y}}$$

- When $E = E(\mathbf{y})$, $\mathbf{y} = \frac{1}{1+e^{-\mathbf{x}}}$ (sigmoid function): $\frac{\partial E}{\partial \mathbf{x}} = \mathbf{y} * (1 - \mathbf{y}) * \frac{\partial E}{\partial \mathbf{y}}$
- When $E = E(\mathbf{y})$, $\mathbf{y} = \max\{0, \mathbf{x}\}$ (ReLU function): $\frac{\partial E}{\partial x_k} = \begin{cases} \frac{\partial E}{\partial y_k} & (\text{if } x_k > 0) \\ 0 & (\text{if } x_k \leq 0) \end{cases}$

For a mini-batch with N of data—just remember this and you can implement NN

- Output layer for regression: for the one half of sum of squared errors: $E = \frac{1}{2}\sum_{n=0}^{N-1}(y_n - t_n)^2$,
  $\frac{\partial E}{\partial \mathbf{y}} = \mathbf{y} - \mathbf{t}$
- Output layer for 0-1 value 2-classification (sigmoid + cross entropy): $\frac{\partial E}{\partial \mathbf{x}} = \mathbf{y} - \mathbf{t}$
- Output layer for K-classification (softmax + cross entropy): $\frac{\partial E}{\partial X} = Y - T$
  (For mini-batches, we usually take the average rather than the sum of the errors, so above three are divided by N.)
- For an affine transformation layer $E = E(Y)$, $Y = XW + \mathbf{b}$:

$$\frac{\partial E}{\partial X} = \frac{\partial E}{\partial Y}W^T, \quad \frac{\partial E}{\partial W} = X^T\frac{\partial E}{\partial Y}, \quad \frac{\partial E}{\partial \mathbf{b}} = \mathrm{sum}\left(\frac{\partial E}{\partial Y}, \mathrm{axis} = 0\right)$$

- When $E = E(Y)$, $Y = \frac{1}{1+e^{-X}}$ (sigmoid function): $\frac{\partial E}{\partial X} = Y * (1 - Y) * \frac{\partial E}{\partial Y}$
- When $E = E(Y)$, $Y = \max\{0, X\}$ (ReLU function): $\frac{\partial E}{\partial x_k^{(n)}} = \begin{cases} \frac{\partial E}{\partial y_k^{(n)}} & (\text{if } x_k^{(n)} > 0) \\ 0 & (\text{if } x_k^{(n)} \leq 0) \end{cases}$

3.8 Exercises on visualization of computational graphs with PyTorch

Notebook: `2-1` **自動微分**

PyTorch is a deep learning library based on Torch, developed by Facebook. (It is said to have inherited the design philosophy of Chainer and made it faster). In 2-1, we will use PyTorch to generate computational graphs and perform automatic differentiation. In particular, we will create computational graphs for matrix computation and the SoftmaxWithLoss layer, and perform automatic differentiation.

- Nodes and Edges in Computational Graphs
- Similarities and differences between PyTorch and Numpy
- Chain rule
- In order for PyTorch to perform automatic differentiation, the last output of the computational graph must be 1D

※ The last point is due to the fact that for example, if X and $Y = f(X)$ are both matrices, then $\dfrac{\partial Y}{\partial X}$ will be a fourth-order array.

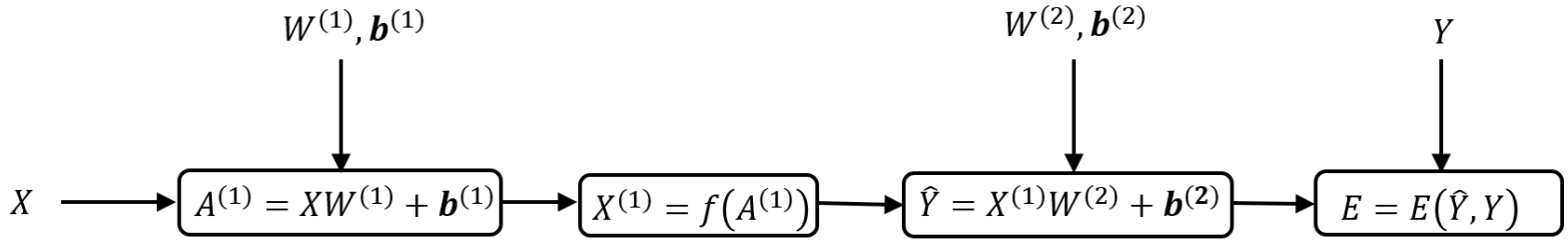
4 Neural network with two layers

A neural network that consists of an input layer (layer 0), a hidden layer (layer 1), and an output layer (layer 2) is called a 2-layer neural network. 3 layers is one way to count the number of layers, but since no operations are performed in the input layer, the number of effective layers is 2.

Since the two-layer neural network is the most basic neural network, let's practice to write the code from scratch.

4.1 Neural network with two layers (Regression)

In a supervised regression problem with input data (a mini-batch with N data) X and corresponding predicted values $\hat{Y}$, a two-layer neural network can be represented as



where when the activation function is ReLU, it holds that

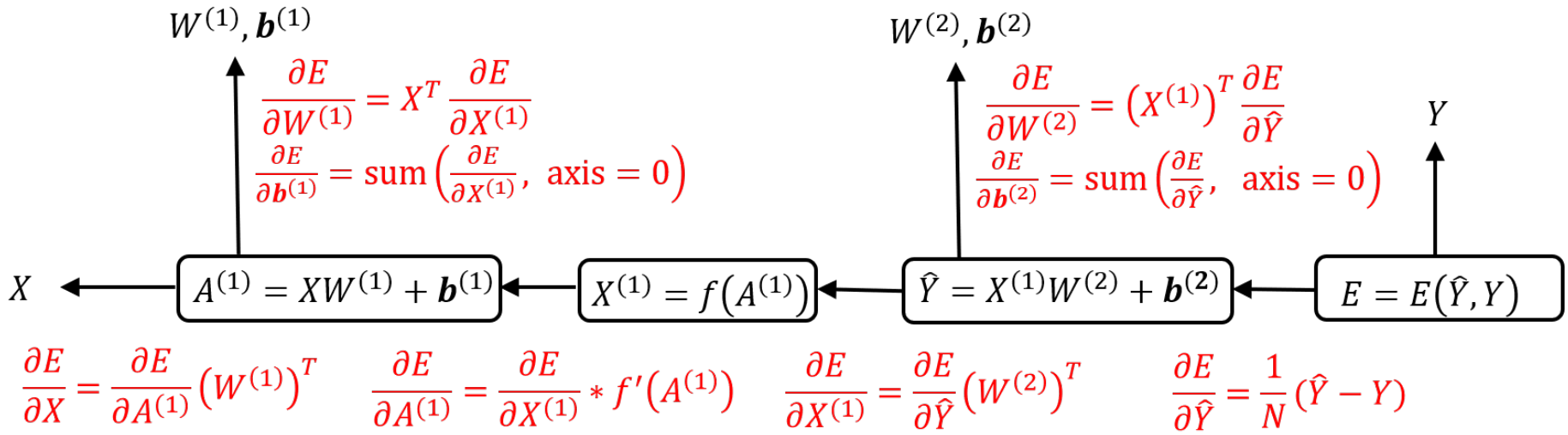
$$X^{(1)} = f(A^{(1)}) = \mathrm{relu}(A^{(1)}) = \max\{A^{(1)}, 0\}$$

and the mean squared error with the prediction Y is

$$E = \frac{1}{2N} \sum_{n=0}^{N-1} (\hat{Y}_n - Y_n)^2$$

Neural network with two layers (Regression) (continued)

On the other haand, from the formula in the previous section, we can caliculate the backprobagation as



where the activation function is ReLU, $\dfrac{\partial E}{\partial A^{(1)}}$ is given by

$$\frac{\partial E}{\partial A^{(1)}} = \frac{\partial E}{\partial X^{(1)}} * (\text{mask of } A^{(1)}), \qquad \left(\text{mask of } A^{(1)} = \textbf{論理式} : A^{(1)} > 0\right)$$

Here “mask” can be written in Python code as

```
# for forward
mask1 = (a1 > 0)
x1 = mask1 * a1
```

```
# for backward
da1 = dx1 * mask1
```

Stochastic gradient decent method

The parameters are updated using the stochastic gradient descent method.

Algorithm for the SGD (revisit):

Assume that we have a set of input data with correct answers.

1. Take arbitrary value for parameter W.
2. Select a few (hundreds) samples randomly (mini-batch) from the set of input data with correct answers
3. Compute a prediction for each of the input data.
4. Calculate the derivative $\dfrac{\partial E}{\partial W}$, where E is the mean of the error from the correct answer in the mini-batch and W is a parameter
5. For a real constant $\lambda > 0$, update W as

$$W \leftarrow W - \lambda \frac{\partial E}{\partial W}$$

and return to 2

4.2 Exercises on implementing a simple two-layer neural network by Numpy

Notebooks:

- 2-2-1_シンプルな 2 層のニューラルネット（回帰）
- 2-2-2_シンプルな 2 層のニューラルネット（2 値分類）
- 2-2-3_シンプルな 2 層のニューラルネット（分類）

Please keep the following points in mind when doing exercises on simple two-layer neural networks by Numpy.

- Design output layers for regression, binary classification, and classification problems respectively.
- back propagation
- How to use Class and its parameters and methods
- Implementation of Sigmoid and ReLU
- How to use Pandas
- Accuracy on training data and on test data

Exercises on implementing a simple two-layer neural network by Numpy No. 2

Following the notebook on the previous page, solve

`2-2-4_Simple2Layers_YearPrediction_問題`

This is an exercise that is essential for understanding this lecture. Make sure you do it.

5 Neural networks with multiple layers

In the previous section, we defined two-layer neural networks and trained them using the stochastic gradient descent method. In this section, we will increase the number of layers.

5.1 Class of layers

- Each layer is implemented as a Python class.
- The activation function is an independent layer.
- We define the following:
  - Affine layer
  - Sigmoid layer
  - ReLU layer
  - Regression error layer
  - Sigmoid and cross-entropy error layer
  - Softmax and cross-entropy error layer

Affine layer

Parameters: Matrxix W, Vector $\mathbf{b}$

Forward propagation:

Input: Matrix X

Output:

$$Y = XW + \mathbf{b}$$

Back propagation:

Input: Matrix dout $= \dfrac{\partial E}{\partial Y}$

Output:

$$\frac{\partial E}{\partial W} = X^T \cdot \text{dout}$$
$$\frac{\partial E}{\partial \mathbf{b}} = \text{Sum of dout with respect to rows (the 0-th index)}$$
$$\frac{\partial E}{\partial X} = \text{dout} \cdot W^T$$

Since we have to use the variable X in the first line of this expression, we store it as `self.x` during the forward calculation. The top two are stored as parameters `self.dW` and `self.db`, and the bottom one is the return value.

Sigmoid layer

Parameters: None

Forward propagation:

Input: matrix X

Output:

$$Y = \text{sigmoid}(X) = \frac{1}{1 + e^{-X}}$$

Back propagation:

Input: Matrix dout $= \dfrac{\partial E}{\partial Y}$

Output:

$$\frac{\partial E}{\partial X} = \text{dout} * Y * (1 - Y)$$

Since the output Y is needed for the backpropagation, it is stored as parameter `self.y` during the forward calculation.

ReLU layer

parameters: None

Forward propagation:

Input: matrix X

Output:

$$Y = \max\{0, X\} = \begin{cases} 0 & (X \leq 0) \\ X & (X > 0) \end{cases}$$

Code: Using broadcast, we can write the following

```
self.mask = (x > 0) # True if x>0  and False if x<=0
y = x * self.mask
```

Back propagation:

Input: Matrix dout $= \dfrac{\partial E}{\partial Y}$

Output:

$$\frac{\partial E}{\partial X} = \begin{cases} 0 & (X \leq 0) \\ \frac{\partial E}{\partial Y} & (X > 0) \end{cases}$$

Code: `dx = dout * self.mask`

Regression error layer

parameters: None

Forward propagation:

Input: Expected value Y, Correct value T

Output:

$$E = \frac{1}{2N}\sum_n (y_n - t_n)^2 \quad (N \text{ is the batch size})$$

Code:

```
batch_size = y.shape[0]
loss = np.sum((y-t) ** 2)/(2 * batch_size)
```

Back propagation:

Input: None Output:

$$\frac{\partial E}{\partial Y} = \frac{1}{N}(Y - T)$$

Code: `dy = ( self.y - self.t )/batch_size`

Sigmoid and cross-entropy error layer

parameters: None

Forward propagation:

Input: Expected value before normalization X, Correct value T (The dimension of each data is one)

Output:

$$Y = \text{sigmoid}(X)$$
$$E = \frac{1}{N}(-T\log(Y) - (1-T)\log(1-Y)) \quad (N \text{ is the batch size})$$

Back propagation:

Input: None Output:

$$\frac{\partial E}{\partial X} = \frac{1}{N}(Y - T)$$

Softmax and cross-entropy error layer

parameters: None

Forward propagation:

Input: Expected value before normalization X, Correct value T

Output:

$$Y = \mathrm{softmax}(X)$$
$$E = -\frac{1}{N}\sum_n \sum_k t_{n,k} \log y_{n,k} \quad (N \text{ is the batch size})$$

Code:

```
y = softmax(x)
batch_size = y.shape[0]
loss =  -np.sum(t * np.log(y))/batch_size
```

Back propagation:

Input: None Output:

$$\frac{\partial E}{\partial X} = \frac{1}{N}(Y - T)$$

5.2 Exercises on Classes of layers

Notebooks:

1. `3-1-1_アファイン層.ipynb`
2. `3-1-2_SoftmaxWithLoss 層_演習問題.ipynb`
3. `3-1-3_その他の層.ipynb`

Notebook 1 defines a Class of Affine layers and uses numerical differentiation to check them. Notebook 2 defines a Class of Softmax and Cross-Entropy Error layers and checks them using numerical differentiation, but some code is missing, please fill in the code to complete it. Notebook 3 defines Classes for the other layers.

Note that in the implementation of the softmax function, as a countermeasure against overflow, instead of

$$\frac{1}{e^{a_0} + \cdots + e^{a_{K-1}}} \left(e^{a_0}, \cdots, e^{a_{K-1}}\right),$$

we use

$$\frac{1}{e^{a_0-c} + \cdots + e^{a_{K-1}-c}} \left(e^{a_0-c}, \cdots, e^{a_{K-1}-c}\right) \qquad (\text{where } c = \max\{a_0, \cdots, a_{K-1}\}).$$

This change does not affect the value of the function.

Remarks on numerical differentiation

In the exercise on the previous page, we checked the correctness of the implementation by comparing the backpropagation of each layer with numerical differentiation. You may consider as this means that:

Isn't it possible to find the gradient by doing numerical differentiation?

The answer is yes. However, since numerical differentiation requires computing forward propagation for each component of the weight parameter, it is very computationally demanding for models with many parameters, and is not used except for testing implementations.

5.3 Exercises on multilayer neural networks

Notebook: `2-4_多層 NN(分類).ipynb`

In this notebook, we will define a class of multilayer neural networks and train them on the MNIST handwritten numbers dataset. Please pay attention to the following points.

- How to store layers: Have layer objects as lists
- How to use the (ordered) dictionary: weighted parameters are stored in an ordered dictionary
- forward propagation
- What happens when the layers become deeper?

Challenges in neural network optimization

In the exercise on the previous page, we learned that simply increasing the depth of the layers will result in a failure to learn. This was the reason why neural networks were considered impractical for a long time, even though they were invented in the 1970s. In the meantime, however, improvements have been made little by little.

In particular, the following are basic:

- Improving the Optimization Method: Alternatives to SGD
- How to set initial values for weight parameters
- Batch Regularization: A method to normalize the output of each layer with respect to the batch

In the next three sections, we will learn more about these so that we can learn as the layers become deeper.

6 Improving the optimization method

In this section, we will learn about variation of the optimization method that improve the stochastic gradient descent (SGD) method to make it easier to converge.

- Momentum: A method that introduces "momentum.
- Nesterov's Momentum: A method based on the "acceleration algorithm" of the gradient method
- AdaGrad: A method that changes the learning coefficient adaptively to the gradient
- RMSProp：An improved version of AdaGrad
- Adam：Further improved versions of AdaGrad and RMSProp (commonly used these days)

In this lecture, we will learn about Momentum and Adam.

Review of SGD

1. From the set of input data with correct answers, randomly select N samples (mini-batches) $\mathbf{x}^{(0)}, \cdots, \mathbf{x}^{(N-1)}$, and set $y^{(0)}, \cdots, y^{(N-1)}$ as the corresponding correct answers
2. For the mean of the cost function $E = \dfrac{1}{N} \displaystyle\sum_{n=0}^{N-1} E\left(f(\mathbf{x}^{(n)}; \theta),\ y^{(n)}\right)$, calculate the gradient

$$\nabla_\theta E = \frac{\partial E}{\partial \theta}$$

   with respect to the parameter θ
3. Update Parameter θ as

$$\theta \leftarrow \theta - \lambda \frac{\partial E}{\partial \theta}$$

The first is the same for all optimization methods, but the second and third vary.

Note: In section 8.3 of [Goodfellow], the cost function is $L(f(\mathbf{x}; \theta),\ y)$, which is very confusing because it uses the same sign as the log likelihood, but the cost function is the log likelihood with the sign reversed (if you ignore the regularization term, etc.). Parameter optimization proceeds in the direction of minimizing the cost $\coloneqq$ maximizing the likelihood.

Note on the magnitude of the learning coefficient

If the learning coefficient λ is too large, learning will become unstable or divergent, while if it is too small, learning will take a long time and you will be stuck in the local optimum.

In practice, it is common to attenuate the coefficients as the learning progresses, which can be thought of as the equivalent of looking for better parameters by looking globally with a rough mesh at the beginning to reduce the cost function, and then looking locally with a fine mesh at the end.

Regardless of the optimization strategy, it is not possible to learn if the gradient converges to zero or diverges. This will be discussed in the next section.

6.1 Momentum

Algorithm of Momentum

α : Hyperparameters satisfying $0 \leq \alpha < 1$

$\mathbf{v}$: An array with the same shape as θ and with initial values of 0: "Momentum"

Execute the following until the termination condition is satisfied:

1. Same as SGD
2. Same as SGD
3. Update Parameter θ and Momentum $\mathbf{v}$ as

$$\mathbf{v} \leftarrow \alpha\mathbf{v} - \lambda\frac{\partial E}{\partial \theta}$$
$$\theta \leftarrow \theta + \mathbf{v}$$

Since the third can be summarized as

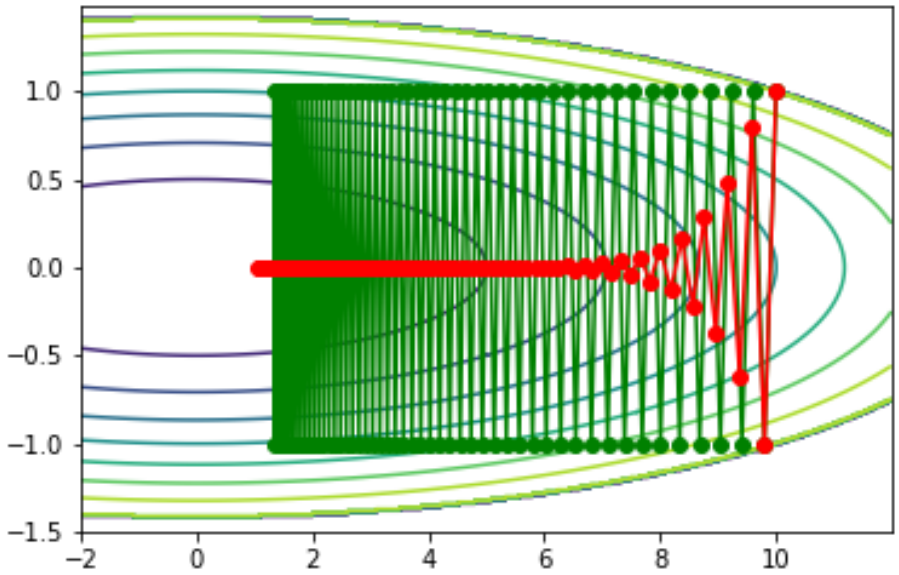
$$\theta \leftarrow \theta + \alpha\mathbf{v} - \lambda\frac{\partial E}{\partial \theta},$$

it is the same as SGD if $\alpha = 0$.

Comparison of gradient descent method's and Momentum's time evolution

The result of applying the Gradient Descent method and (2 and 3 in the algorithms of) Momentum from the initial value $(x, y) = (10, 1)$ to

$$f(x, y) = x^2 + 100y^2.$$



Comparison of GD's and Momentum's time evolution

Green: GD, Red: Momentum

Relationship with equations of motion — why is it called "momentum"?

Momentum can be thought as an algorithm that if one step in the algorithm is considered as one step in time, then θ and $\mathbf{v}$ represent the position and the momentum of the object at time t, $1-\alpha$ represents the friction coefficient, and $-\lambda\frac{\partial E}{\partial \theta}$ represents the. In fact, The similarity between

$$\theta(t+1) - \theta(t) = \mathbf{v}(t+1)$$
$$\mathbf{v}(t+1) - \mathbf{v}(t) = -(1-\alpha)\mathbf{v}(t) - \lambda\frac{\partial E}{\partial \theta}$$

and the momentum

$$\dot{\theta}(t) = \mathbf{v}(t)$$
$$\dot{\mathbf{v}}(t) = -(1-\alpha)\mathbf{v}(t) - \lambda\frac{\partial E}{\partial \theta}$$

may be obvious.

6.2 Adam

There are algorithms such as AdaGrad and RMSProp that adaptively change the learning coefficients λ for each component of the parameters as the learning progresses.

Adam (adaptive momentum) [Kingma-Ba, 2014] is considered to be a further improvement by applying the RMSProp idea to momentum.

However, experience shows that it is often better to force the initial learning coefficients to be changed in the middle of the process, even for Adam.

Adam's parameter update algorithm (original)

$0 < \beta_1 < 1$, $0 < \beta_2 < 1$: hyperparameters ($\beta_1 = 0.9$, $\beta_2 = 0.999$ for example)
$\mathbf{s}$, $\mathbf{r}$: Array of the same shape as the parameter θ with initial values of 0

$$\mathbf{s} \leftarrow \beta_1 \mathbf{s} + (1 - \beta_1) \frac{\partial E}{\partial \theta}$$

$$\mathbf{r} \leftarrow \beta_2 \mathbf{r} + (1 - \beta_2) \frac{\partial E}{\partial \theta} * \frac{\partial E}{\partial \theta}$$

$$\tilde{\mathbf{s}} \leftarrow \frac{\mathbf{s}}{1 - \beta_1^t}$$

$$\tilde{\mathbf{r}} \leftarrow \frac{\mathbf{r}}{1 - \beta_2^t} \quad (t \text{ is the number of updates until then})$$

$$\theta \leftarrow \theta - \alpha \frac{\tilde{\mathbf{s}}}{\sqrt{\tilde{\mathbf{r}}} + \epsilon}$$

It's complicated, and hard to explain what it means.

Update algorithm rewritten to be easier to implement

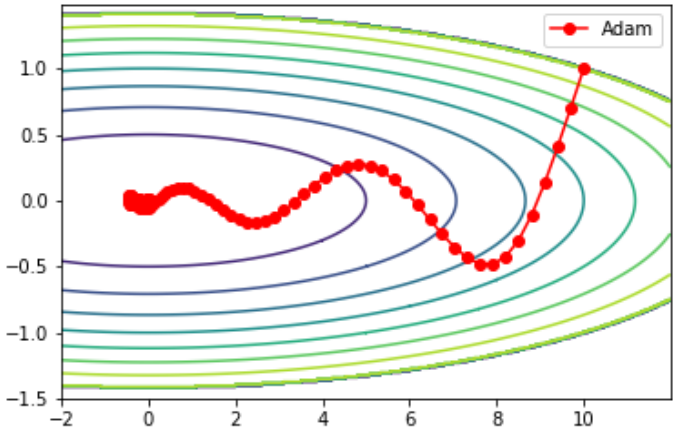
The initial value of λ is set to for example 0.001.

$$\mathbf{s} \leftarrow \beta_1 \mathbf{s} + (1 - \beta_1) \frac{\partial E}{\partial \theta}$$

$$\mathbf{r} \leftarrow \beta_2 \mathbf{r} + (1 - \beta_2) \frac{\partial E}{\partial \theta} * \frac{\partial E}{\partial \theta}$$

$$\lambda \leftarrow \lambda \frac{\sqrt{1 - \beta_2^t}}{1 - \beta_1^t} \quad (t \text{ is the number of updates until then})$$

$$\theta \leftarrow \theta - \lambda \frac{\mathbf{s}}{\sqrt{\mathbf{r}} + \epsilon}$$



Example of time evolution

6.3 Exercises on implementation of optimization methods

Notebooks:

1. `2-5-1_最適化法.ipynb`
2. `2-5-2_最適化法の NN への適用.ipynb`

Notebook 1 defines a class of optimization methods and finds the minimum value of a two-variable quadratic function. Notebook 2 applies it to a multi-layer neural network.

In Notebook 1, there are exercises at the end. Also, please pay attention to the following points.

- Storage of parameters in the form of a dictionary with arrays as elements: for application to NNs
- Differences in the behavior of optimization methods when applied to NNs

7 How to set the initial values of weight parameters

A neural network has many training parameters, and their initial values in training must be generated randomly in advance. For example, in the case of the matrix W and the bias vector $\mathbf{b}$ in the affine transform

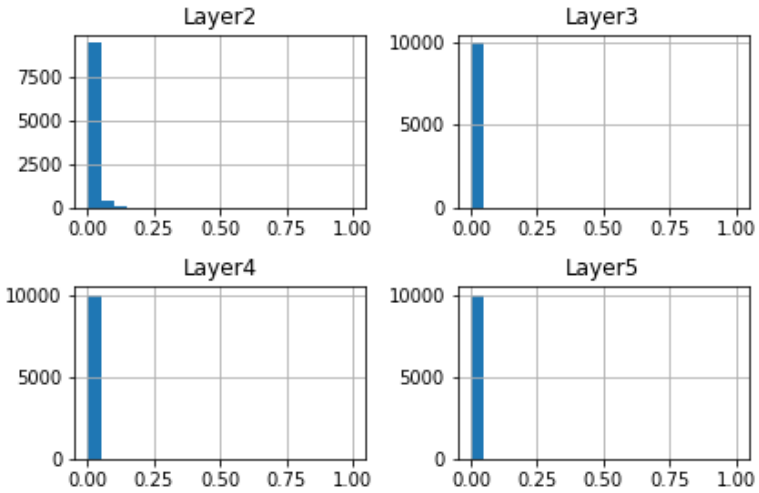
$$\mathbf{y} = \mathbf{x}W + \mathbf{b},$$

each component of W is usually taken according to a normal distribution with mean 0 with variance σ^2, and each component of $\mathbf{b}$ is assumed to be zero.

However, when the network becomes somewhat deep (for example 5 layers or more), simply taking the variance σ^2 without consideration will cause the data values in the deep layers to diverge or become close to zero (the gradient will also converge or diverge in the same way), and learning will not proceed appropriately. In order to avoid this, the variance of the weight parameters must be adjusted appropriately according to the structure of the network.

7.1 Convergence and divergence of data in the middle layers

For example, as in the following exercise, in a 5-layer network, suppose the input values from layer 2 to layer 5 for 10000 data are represented by the following histogram.



This situation is caused by the fact that the standard deviation of the distribution of the first weight parameter is too small, and the values converge to zero as the layers progress. On the other hand, if the standard deviation of the parameter is too large, the values will diverge.

How do you take the standard deviation to be appropriate?

7.2 Parameter initialization strategy

He's initial value

In the method called **He's initial value**, named after the discoverer, when using ReLU as the activation function, if the number of neurons in the previous layer is N_{l-1}, the variance of W in the l-th layer is set as

$$\sigma_l^2 = \frac{2}{N_{l-1}}$$

Note: In the case of convolutional layers, N_{l-1} is replaced by the number of neurons of the previous layer used as input to one neuron in the l-th layer.

Glorot-Bengio's initial value

In the method called **Glorot-Bengios initial value**when using the sigmoid function as the activation function, the variance of W in the l-th layer is set as $\sigma_l^2 = \frac{2}{N_{l-1} + N_l}$.

Why is it good to take it this way?

When the weight parameter W_l follows a distribution with mean 0 and variance V_l, the variance of the data is multiplied by about $N_{l-1} \times V_l$ by the l-th layer, so we can say that if we take V_l as

$$V_l \coloneqq \frac{1}{N_{l-1}},$$

the variance remains the same and is propagated to the next layer.

7.3 Exercises on how to take initial values of parameters

Notebook: `2-6_初期値の取り方.ipynb`

In this notebook, we will experiment with how data propagates through the layers and implement He's initial values into the classes of neural networks we have created.

Please note the followings.

- Creating histograms with Pandas
- How data is propagated through the layers
- Relationship between data propagation and learning
- The way to implement the initialization strategy

8 Batch normalization

Batch normalization [Ioffe, S., Szegedy, C., 2015] is a method to prevent gradient convergence and explosion by normalizing the data for each neuron in each layer for mini-batches. In the previous section, we considered how to take the initial values of the weight parameters, but batch normalization normalizes the output of the layers. Batch normalization is now widely used even though it is a relatively recently proposed method.

8.1 Normalization of datasets

Although we have actually used it many times already, let's summarize dataset normalization.

For a dataset

$$X = [x_0, x_1, x_2, \cdots, x_{N-1}]$$

The mean μ and variance $V = \sigma^2$ are calculated as

$$\mu = \frac{1}{N} \sum_{n=0}^{N-1} x_n$$

$$V = \frac{1}{N} \sum_{n=0}^{N-1} (x_n - \mu)^2$$

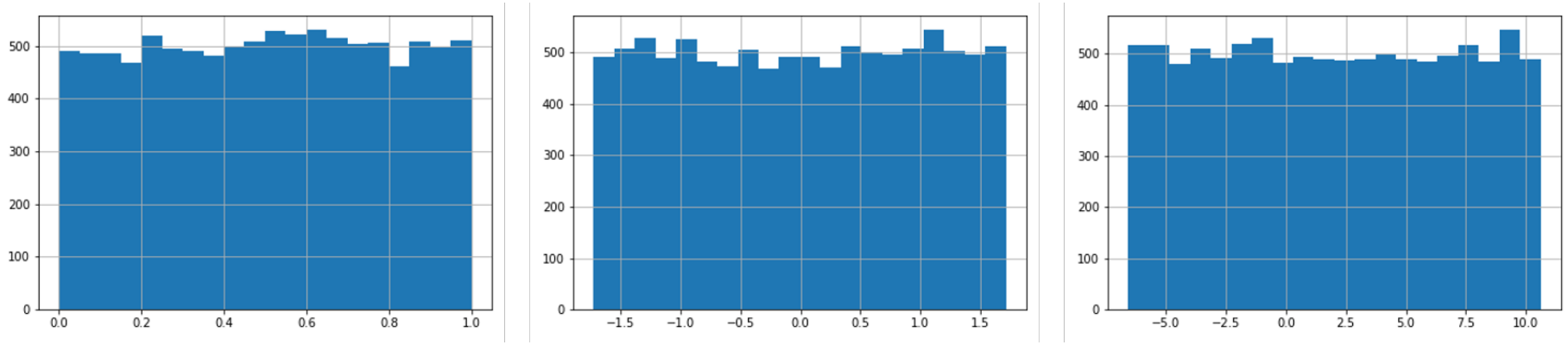
$$\sigma = \sqrt{V}$$

The operation to convert this to a dataset with mean 0 and variance 1 is as follows.

$$\tilde{X} = \frac{X - \mu}{\sigma}$$

Further, we can convert it to mean β and variance γ^2 by setting

$$\tilde{\tilde{X}} = \gamma * \tilde{X} + \beta.$$



10,000 sample data following the uniform distribution. Left: The original data on the $[0, 1]$ interval, Middle: The normalized one, Right: Transformed one to mean 2, variance $5^2 = 25$.

8.2 Algorithm for batch normalization

X: output of the subject layer of shape $N \times K$ (number of data in mini-batch N, number of neurons in layer K).
β_k: array of shape $(K,)$, average value of the new dataset, initial components are 0.
γ_k: array of shape $(K,)$, standard deviation of the new dataset, initial components are 1.

Forward propagation

$$\mu_k = \frac{1}{N} \sum_{n=0}^{N-1} x_{nk}$$

$$v_k = \frac{1}{N} \sum_{n=0}^{N-1} (x_{nk} - \mu_k)^2$$

$$\tilde{x}_{nk} = \frac{x_{nk} - \mu_k}{\sqrt{v_k + \epsilon}}$$

$$y_{nk} = \gamma_k \, \tilde{x}_{kn} + \beta_k$$

where ϵ is a small quantity to prevent divergence, such as $e = 10^{-8}$.

This transformation can be written in NumPy style as

$$\begin{aligned}
\mu &= \mathrm{mean}(X, \mathrm{axis} = 0) \\
\mathbf{v} &= \mathrm{mean}\big((X - \mu)^2, \mathrm{axis} = 0\big) \\
\tilde{X} &= \frac{X - \mu}{\sqrt{\mathbf{v} + \epsilon}} \\
Y &= \gamma * \tilde{X} + \beta
\end{aligned}$$

Note that β and γ are K-dimensional vectors.

Batch normalization does not work well when the number of data to be evaluated is small (for example, when the number of data is 1, $Y = \beta$). In particular, it is normal to have 1 data when predicting. This problem can be dealt with by using the mean and variance data during training for testing.

Back propagation

Backpropagation can be calculated using chain rules and computational graphs. However, it is very complicated, so we only write the result as follows.

$$\frac{\partial E}{\partial \beta} = \mathrm{sum}\left(\frac{\partial E}{\partial Y},\ \mathrm{axis} = 0\right), \quad \frac{\partial E}{\partial \gamma} = \mathrm{sum}\left(\tilde{X} * \frac{\partial E}{\partial Y},\ \mathrm{axis} = 0\right)$$

and

$$\mathbf{c} = \mathrm{mean}\left(\frac{\partial E}{\partial Y},\ \mathrm{axis} = 0\right), \quad \mathbf{d} = \mathrm{mean}\left(\tilde{X} * \frac{\partial E}{\partial Y},\ \mathrm{axis} = 0\right)$$

$$\frac{\partial E}{\partial X} = \frac{\gamma}{\sigma}\left(\frac{\partial E}{\partial Y} - \mathbf{c} - \tilde{X} * \mathbf{d}\right)$$

Here, although the array shapes are different for $\mathbf{c}$, $\mathbf{d}$ and $\tilde{X}$, Y, they can be calculated by broadcast.

This formula uses the gradient $\frac{\partial E}{\partial Y}$ of the error E with respect to Y to calculate $\frac{\partial E}{\partial X}$.

8.3 Exercises on batch normalization

Notebook: `2-7_バッチ正規化.ipynb`

In this notebook, we will do some exercises on dataset normalization and implement batch normalization.

Please note the followings.

- Normalizing datasets with NumPy
- What is done in batch normalization?
- Batch normalization is one of the layers
- How do we implement the difference between training and testing?

8.4 Exercises on optimization for deep models

Notebook: `2-8_深層モデルのための最適化.ipynb`

In this notebook, we implement and experiment with optimization methods, parameter initialization strategies, and batch normalization in a neural network.

Please note the followings.

- Effectiveness of each method
- How deep can the model go?

9 Regularization for deep models

The deeper the model of the neural network, the more likely it is to overfit. The following methods can be used to avoid overfitting.

- Parameter norm penalty: Regularization by the norm of the weight parameter
- Dataset augmentation: Increase data by creating data that imitates the input dataset
- Noise injection for output values: Round the output values to increase robustness
- Semi-supervised learning: Use unsupervised data for training
- Early stopping: Stop learning depending on the improvement of accuracy on the evaluation dataset
- Ensemble learning: Averaging predictions from multiple models
- Dropout: Neuron output is not used in the next layer with a certain probability

In this section, we will discuss the parameter norm penalty and dropout. The dataset extension will be explained in another section.

9.1 Parameter norm penalty

In supervised machine learning, model regularization is often achieved by changing the original cost function J of the problem to

$$\tilde{J} = J + \alpha\Omega$$

(where $\alpha > 0$) using some norm Ω for the parameters.

In the case of neural networks, for the coefficient matrix $W^{(l)}$ and bias vector $\mathbf{b}^{(l)}$ of the lth layer in an L-layer neural network, it is common to use only $W^{(l)}$ for the calculation of Ω. That is,

$$\tilde{J} = J + \alpha\Omega\left(W^{(1)}, \cdots, W^{(L)}\right).$$

Although it is possible to change α for each layer, it is more usual to use a common α for the entire model since it increases the search range for hyperparameters.

L2 parameter regularization

Regularization

$$\tilde{J} = J + \alpha \sum_{l=1}^{L} ||W^{(l)}||^2$$

using the L2 norm of the coefficient matrix

$$\Omega = \sum_{l=1}^{L} ||W^{(l)}||^2 = \sum_{l=1}^{L} W^{(l)} \left(W^{(l)}\right)^T = \text{sum}(W^{(l)} * W^{(l)})$$

is called L2 parameter regularization (or simply L2 regularization). In other fields, L2 regularization is sometimes referred to as ridge regularization or Tikhonov regularization.

The partial derivative of Ω with respect to $W^{(l)}$ can be easily calculated as

$$\frac{\partial \Omega}{\partial W^{(l)}} = 2\alpha W^{(l)}$$

L1 parameter regularization

Regularization

$$\tilde{J} = J + \alpha \sum_{l=1}^{L} \left| W^{(l)} \right|$$

using the L1 norm of the coefficient matrix

$$\Omega = \sum_{l=1}^{L} \left| W^{(l)} \right|$$

(where $\left| W^{(l)} \right|$ is the sum of the absolute values of all components of the matrix $W^{(l)}$) is called L1 parameter regularization (or simply L1 regularization). Linear regression with L1 regularization is also called LASSO (Least Absolute Shrinkage and Selection Operator).

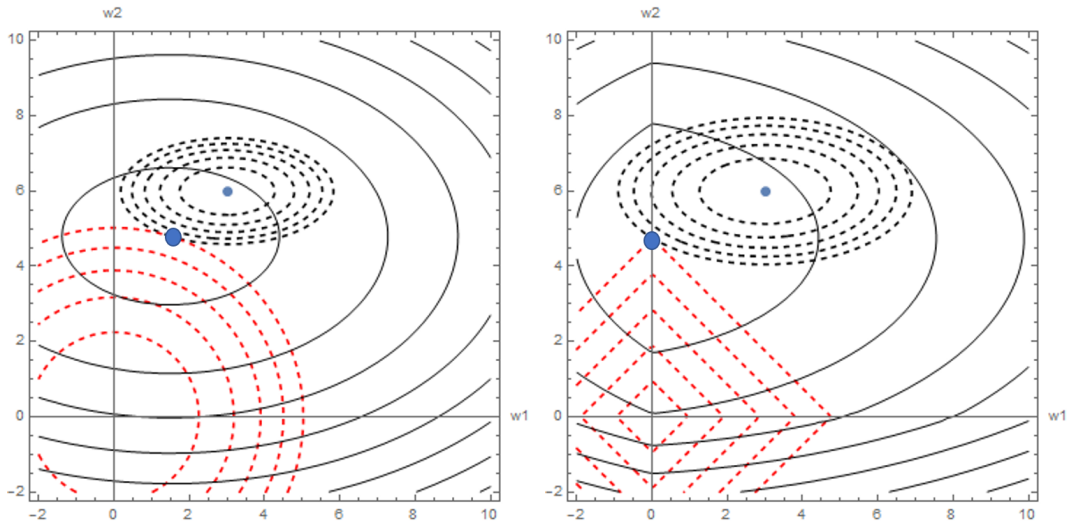
The partial derivative of Ω with respect to $W^{(l)}$ can be calculated as

$$\frac{\partial \Omega}{\partial W^{(l)}} = \alpha \text{ sign } W^{(l)}$$

(where sign x is the function that takes the sign of x, and when x is a matrix, it is applied to each of the components).

Sparsification effect of L1 regularization

A set of values is said to be sparse if many of the values are zero. Comparing the L2 regularization with the L1 regularization, the L1 regularization has the effect of making the parameters sparse. The figure below shows L2 regularization and L1 regularization for a model minimizing the objective function $f(w_1, w_2) = (w_1 - 3)^2 + 4(w_2 - 6)^2$, where the solution of L1 regularization is $w_1 = 0$.



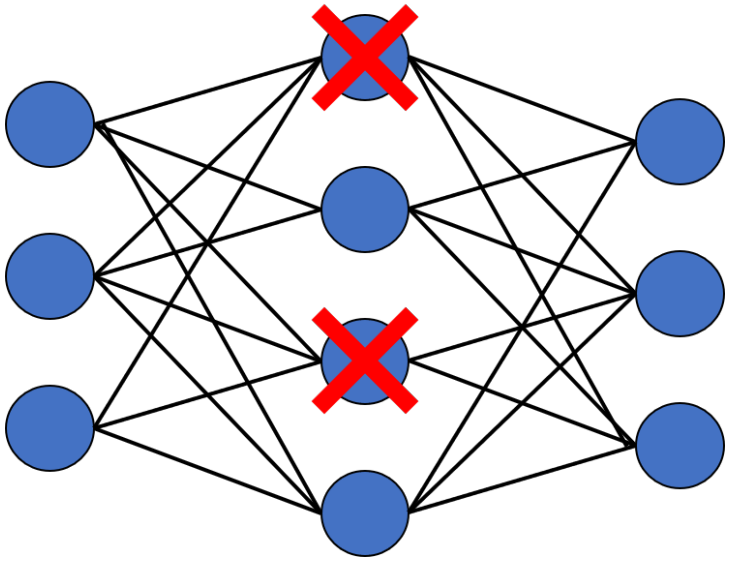
L2 regularization and L1 regularization

Left: Case of adding $w_1^2 + w_2^2$ as a regularization term, Right: Case of adding $10(|w_1| + |w_2|)$,
Black dashed line: Contour line of f, Red dashed line: Contour line of the regularized term,
Solid black line: Contour line of the regularized function

Small point: original minimum point, Large point: minimum point when regularization term is added

9.2 Dropout

Dropout [Srivastava, N. et al., 2014] is a technique to improve the training of neural networks. It is a method where during training, a neuron is erased (masked) for each data in each layer where dropout is applied at a given rate p, and during inference, no mask is applied but instead the output value of each layer where dropout was applied is multiplied by $1-p$. It is thought to imitate ensemble learning by changing the way neurons are eliminated for each data, and is widely used to improve generalization performance.



Algorithm for Dropout

p: probability of setting the output value to zero, one of the hyperparameters

Input: output of the layer $X_{N\times K}$ (N: batch size, K: number of neurons)

Forward propagation

Mask M: Array of the same shape as X with each component taking the value 0 with probability p and 1 with probability $1-p$, independently.

$$Y = \begin{cases} X * M & \text{(during training)} \\ X * (1-p) & \text{(during prediction)} \end{cases}$$

Back propagation

$$\frac{\partial E}{\partial X} = \frac{\partial E}{\partial Y} * \frac{\partial Y}{\partial X} = \frac{\partial E}{\partial Y} * M$$

9.3 Exercises on dropout

Notebook: `2-9_ドロップアウト.ipynb`

In this notebook, we will do some exercises on dropouts.

Please note the followings.

- How to implement the mask M: each component takes 0 with probability p and 1 with probability $1-p$
- Computation for back propagation
- Computation during prediction

9.4 Comprehensive exercises on regularization

Notebook: `2-10_正則化に関する演習.ipynb`

In this notebook, we summarize the previous exercises on neural networks with batch regularization, dropout, L2 regularization, and L1 regularization.

Please note the followings.

1. How to design a network to improve accuracy (this is a difficult question)
2. Why does changing the learning coefficient to a smaller value during training improve accuracy?

Note on point 2: It is considered to be corresponding to searching for good parameters by looking globally with a rough mesh at the beginning so that the cost function becomes smaller, and then looking locally with a fine mesh at the end.