Reading Notes: *Neural Networks and Deep Learning*

Nielsen, Michael A. *Neural Networks and Deep Learning*. Determination Press, 2015.
http://neuralnetworksanddeeplearning.com

These notes were taken while reading Michael Nielsen's *Neural Networks and Deep Learning*. They record exercise solutions, mathematical derivations, and annotated code excerpts, organised chapter by chapter, as a personal reference for the mathematical foundations and Python implementations discussed in the book.

1 Chapter 1

1.1 Exercise: Sigmoid neurons simulating perceptrons, part I

This holds simply because $h(cx) = h(x)$ for any $c > 0$.

1.2 Exercise: Sigmoid neurons simulating perceptrons, part II

If $z \neq 0$, then $\lim_{c\to\infty} \sigma(cz) = h(z)$. However, if $z = 0$, then $\sigma(cz) = 0.5$ for $c > 0$.

1.3 The `feedforward` function

The feedforward function is the composition of all layer transitions:

$$g^l \colon \mathbb{R}^{n^{l-1}} \to \mathbb{R}^{n^l}, \quad a^{l-1} \mapsto \sigma(w^l a^{l-1} + b^l)$$

where

- n^l is the number of neurons in layer l,
- $w^l \in \mathbb{R}^{n^l \times n^{l-1}}$ is the weight matrix of layer l,
- $b^l \in \mathbb{R}^{n^l}$ is the bias vector of layer l, and
- σ is the sigmoid function.

Thus

$$\text{feedforward}(a) = g^L \circ g^{L-1} \circ \cdots \circ g^1(a)$$

where L is the number of layers (excluding the input layer).

1.4 The `SGD` function

Inputs

- `training_data`: a list of tuples `(x, y)` representing the training examples
- `epochs`: the number of epochs
- `mini_batch_size`: the size of the mini-batches
- `eta`: the learning rate

The function `SGD` trains the network using stochastic gradient descent (SGD).

Loop over the number of epochs: `for j in range(epoch)`. At each iteration, the examples are randomly shuffled and split into mini-batches:

```
random.shuffle(training_data)
mini_batches = [
    training_data[k:k+mini_batch_size]
    for k in range(0, n, mini_batch_size)
]
```

Loop over the number of mini-batches; for each mini-batch, the biases and weights are updated in place using the `update_mini_batch` function (described below):

```
for mini_batch in mini_batches:
    self.update_mini_batch(mini_batch, eta)
```

1.5 The `update_mini_batch` function

Inputs

- `mini_batch`: list of tuples `(x, y)` of training examples
- `eta`: learning rate η

The weights and biases are updated as follows:

$$w' = w - \eta \nabla_w C, \quad b' = b - \eta \nabla_b C$$

Typically, the gradient of the cost function is calculated per example:

$$\nabla C = \frac{1}{N_S} \sum_{(x,y) \in S} \nabla C_{x,y}$$

where S is the mini-batch and N_S is the number of examples in it.

The function initiates the bias vectors b^l and weight matrices w^l with zero values:

```
nabla_b = [np.zeros(b.shape) for b in self.biases]
nabla_w = [np.zeros(w.shape) for w in self.weights]
```

The `backprop` function returns a tuple `(delta_nabla_b, delta_nabla_w)` where `delta_nabla_b` and `delta_nabla_w` are lists of NumPy arrays containing the gradients for each layer.

The function `backprop` is described in Section 2.

In the following code line:

```
delta_nabla_b, delta_nabla_w = self.backprop(x, y)
```

we store the gradient of the cost function for each layer in the lists:

- `delta_nabla_b`: $(\nabla_{b^1} C_{x,y}, \dots, \nabla_{b^L} C_{x,y})$
- `delta_nabla_w`: $(\nabla_{w^1} C_{x,y}, \dots, \nabla_{w^L} C_{x,y})$

Then, the gradients are summed over the mini-batch, layer by layer:

```
nabla_b = [nb+dnb for nb, dnb in zip(nabla_b, delta_nabla_b)]
nabla_w = [nw+dnw for nw, dnw in zip(nabla_w, delta_nabla_w)]
```

So the variables `nabla_b` and `nabla_w` contain the sum of per-example gradients for each layer:

- `nabla_b`: $\left(\sum_{(x,y)\in S} \nabla_{b^l} C_{x,y}\right)$
- `nabla_w`: $\left(\sum_{(x,y)\in S} \nabla_{w^l} C_{x,y}\right)$

Finally, the weights and biases are updated layer by layer:

```
self.weights = [
    w - (eta / len(mini_batch)) * nw
    for w, nw in zip(self.weights, nabla_w)
]
self.biases = [
    b - (eta / len(mini_batch)) * nb
    for b, nb in zip(self.biases, nabla_b)
]
```

The function does not return anything; the weights and biases are updated in place.

2 Chapter 2

2.1 Equation 29

Error in layer l:

$$\delta^l = \nabla_{z^l} C$$

2.2 Equation BP1 and BP1a

Error in the output layer:

$$\begin{aligned}
\delta^L &= \nabla_{z^L} C \\
&= \nabla (C \circ \sigma)(z^L) \\
&= J_\sigma(z^L)^\top \nabla C(a^L(z^L)) \\
&= \operatorname{diag}(\sigma'(z^L)) \nabla_a C \\
&= \nabla_a C \odot \sigma'(z^L)
\end{aligned}$$

For the MSE cost function $C = \frac{1}{2}\sum_j (a_j^L - y_j)^2$ and the sigmoid activation function $\sigma(z) = 1/(1+e^{-z})$ we have:

$$\nabla_a C = a^L - y \quad \text{and} \quad \sigma'(z) = \sigma(z)(1-\sigma(z))$$

2.3 Equation BP2

Equation for the error δ^l in terms of the error in the next layer. Let us start with the penultimate layer $L-1$:

$$\begin{aligned}
\delta^{L-1} &= \nabla_{z^{L-1}} C \\
&= \nabla (C \circ \sigma \circ z^L \circ \sigma)(z^{L-1}) \\
&= J_{C \circ \sigma \circ z^L \circ \sigma}(z^{L-1})^\top \\
&= \left(J_C(a^L) J_\sigma(z^L) J_{z^L}(a^{L-1}) J_\sigma(z^{L-1})\right)^\top \\
&= \operatorname{diag}(\sigma'(z^{L-1})) J_{z^L}(a^{L-1})^\top \operatorname{diag}(\sigma'(z^L)) \nabla_a C \\
&= (w^L)^\top \delta^L \odot \sigma'(z^{L-1})
\end{aligned}$$

Indeed, we have:

$$J_{z^l}(a^{l-1}) = w^l$$

More generally, we have:

$$\begin{aligned}\nabla_{z^l} C &= \nabla(C \circ \sigma \circ z^L \circ \sigma \circ ... \circ z^{l+1} \circ \sigma)(z^l) \\ &= (w^{l+1})^\top \odot \sigma'(z^l) \cdot ... \cdot (w^L)^\top \odot \sigma'(z^{L-1}) \cdot \nabla_a C \odot \sigma'(z^L)\end{aligned}$$

Proceeding by induction, we get:

$$\delta^l = (w^{l+1})^\top \delta^{l+1} \odot \sigma'(z^l)$$

2.4 Equation BP3

We have:

$$\nabla_{b^l} C = J_{z^l}(b^l)^\top \nabla_{z^l} C$$

However, $J_{z^l}(b^l) = I$, and thus:

$$\nabla_{b^l} C = \delta^l$$

2.5 Equation BP4

Let n_l be the number of units in layer l. The gradient of the cost function with respect to the weights w^l can be written as:

$$\nabla_{w^l} C = \begin{bmatrix} \dfrac{\partial C}{\partial w^l_{\cdot,1}} \\ \vdots \\ \dfrac{\partial z^l}{\partial w^l_{\cdot,n^{l-1}}} \end{bmatrix}$$

We calculate it from the gradient of the cost function with respect to z^l:

$$\begin{aligned}\nabla_{w^l} C &= J_{z^l}(w_l)^\top \nabla_{z^l} C \\ &= J_{z^l}(w_l)^\top \delta^l\end{aligned}$$

The Jacobian matrix $J_{z^l}(w_l)$ can be written as:

$$J_{z^l} = \begin{bmatrix} \dfrac{\partial z^l}{\partial w^l_{\cdot,1}} & \dfrac{\partial z^l}{\partial w^l_{\cdot,2}} & \cdots & \dfrac{\partial z^l}{\partial w^l_{\cdot,n^{l-1}}} \end{bmatrix}$$

where

$$z^l = w^l a^{l-1} + b^l$$

Element-wise, we have:

$$z^l_j = \sum_{k=1}^{n_{l-1}} w^l_{jk} a^{l-1}_k + b^l_j, \quad j = 1, ..., n_l$$

and the derivative of z^l_j with respect to w^l_{jk} is

$$\frac{\partial z^l_j}{\partial w^l_{ik}} = \begin{cases} a^{l-1}_k, & \text{if} \quad i = j \\ 0, & \text{else} \end{cases}$$

Thus

$$\frac{\partial z^l}{\partial w^l_{\cdot,k}} = \begin{bmatrix} \frac{\partial z^l_1}{\partial w^l_{1,k}} & \cdots & \frac{\partial z^l_1}{\partial w^l_{n^l,k}} \\ \vdots & \ddots & \vdots \\ \frac{\partial z^l_{n_l}}{\partial w^l_{1,k}} & \cdots & \frac{\partial z^l_{n_l}}{\partial w^l_{n^l,k}} \end{bmatrix} = \operatorname{diag}(\begin{bmatrix} a^{l-1}_k & \cdots & a^{l-1}_k \end{bmatrix})$$

So, the Jacobian matrix is:

$$J_{z^l}(w_l) = \begin{bmatrix} \operatorname{diag}(\begin{bmatrix} a^{l-1}_1 & \cdots & a^{l-1}_1 \end{bmatrix}) \\ \vdots \\ \operatorname{diag}(\begin{bmatrix} a^{l-1}_{n^l} & \cdots & a^{l-1}_{n^l} \end{bmatrix}) \end{bmatrix}$$

and the gradient of the cost function with respect to w^l is

$$\nabla_{w^l} C = a^{l-1} \otimes_{\text{Kron}} \delta^l$$

where $\otimes_{\text{Kron}}$ is the Kronecker product.

Element-wise, we have:

$$\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j$$

In the Python implementation, the partial derivative of the cost function with respect to the weights is stored in matrices of the same shape as the weights:

$$\left[\frac{\partial C}{\partial w^l_{jk}}\right]_{jk} = \begin{bmatrix} \frac{\partial z^l}{\partial w^l_{\cdot,1}} & \frac{\partial z^l}{\partial w^l_{\cdot,2}} & \cdots & \frac{\partial z^l}{\partial w^l_{\cdot,n^{l-1}}} \end{bmatrix}$$

These matrices can be calculated with the outer product of the error δ^l and the activations a^{l-1}:

$$\left[\frac{\partial C}{\partial w^l_{jk}}\right] = \delta^l (a^{l-1})^\top$$

2.6 Python implementation of the backpropagation algorithm

```
class Network(object):

    def __init__(self, sizes):
        self.num_layers = len(sizes)
        self.sizes = sizes
        self.biases = [np.random.randn(y, 1) for y in sizes[1:]]
        self.weights = [np.random.randn(y, x)
                        for x, y in zip(sizes[:-1], sizes[1:])]

    def backprop(self, x, y):
        """Return a tuple ``(nabla_b, nabla_w)`` representing the gradient
            for
        the cost function C_x.  ``nabla_b`` and ``nabla_w`` are layer-by-
           layer
        lists of numpy arrays, similar to ``self.biases`` and
        ``self.weights``."""
        nabla_b = [np.zeros(b.shape) for b in self.biases]
        nabla_w = [np.zeros(w.shape) for w in self.weights]
        # feedforward
```

```
        activation = x
        activations = [x] # list to store all the activations, layer by
            layer
        zs = [] # list to store all the z vectors, layer by layer
        for b, w in zip(self.biases, self.weights):
            z = np.dot(w, activation) + b
            zs.append(z)
            activation = sigmoid(z)
            activations.append(activation)
        # backward pass initiation
        delta = (activations[-1] - y) * sigmoid_derivative(zs[-1])
        nabla_b[-1] = delta
        nabla_w[-1] = np.dot(delta, activations[-2].transpose())a
        # backward pass
        for l in range(2, self.num_layers):
            z = zs[-l]
            sp = sigmoid_derivative(z)
            delta = np.dot(self.weights[-l+1].transpose(), delta) * sp
            nabla_b[-l] = delta
            nabla_w[-l] = np.dot(delta, activations[-l-1].transpose())
        return (nabla_b, nabla_w)

def sigmoid(self, z):
    """The sigmoid function."""
    return 1 / (1 + np.exp(-z))

def sigmoid_derivative(self, z):
    """Derivative of the sigmoid function."""
    return sigmoid(z) * (1 - sigmoid(z))
```

3 Chapter 3

3.1 Single neuron which take the input 1 to input 0

Since $x = 1$ and $y = 0$, we have:

- feedforward function: $a = \sigma(wx + b) = \sigma(w + b)$
- cost function: $C = \frac{1}{2}(a - y)^2 = \frac{1}{2}a^2$

The gradient of the cost function with respect to w and b is:

$$\nabla_{w,b} C = a\,\sigma'(w + b) \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

Thus, w and b are updated as follows:

$$w' = w - \eta\, a^2(1 - a), \quad b' = b - \eta\, a^2(1 - a)$$

Python implementation

```
# single_neuron.py

import numpy as np
import matplotlib.pyplot as plt

class single_neuron(object):

```

```
    def __init__(self, weight, bias):
        self.weight = weight
        self.bias = bias
        # Initialize lists to store the history of parameters and outputs
        self.weights = []
        self.biases = []
        self.costs = []
        self.outputs = []

    def feedforward(self):
        return self.sigmoid(self.weight + self.bias)

    def cost(self):
        return 0.5 * self.feedforward() ** 2

    def sigmoid(self, x):
        return 1 / (1 + np.exp(-x))

    def update_parameters(self, eta):
        a = self.feedforward()
        delta = eta * a ** 2 * (1 - a)
        self.weight = self.weight - delta
        self.bias = self.bias - delta

    def SGD(self, eta, epochs):
        for i in range(epochs):
            # Store the history of weights, biases, costs, and outputs
            self.weights.append(self.weight)
            self.biases.append(self.bias)
            self.costs.append(self.cost())
            self.outputs.append(self.feedforward())
            self.update_parameters(eta)

    def plot(self):
        epochs = range(1, len(self.weights) + 1)

        # Create subplots
        fig, axs = plt.subplots(2, 2, figsize=(12, 8))

        # Plot Weights
        axs[0, 0].plot(epochs, self.weights, label='Weights')
        axs[0, 0].set_title('Weights vs Epochs')
        axs[0, 0].set_xlabel('Epochs')
        axs[0, 0].set_ylabel('Weights')
        axs[0, 0].legend()

        # Plot Biases
        axs[0, 1].plot(epochs, self.biases, label='Biases', color='orange'
            )
        axs[0, 1].set_title('Biases vs Epochs')
        axs[0, 1].set_xlabel('Epochs')
        axs[0, 1].set_ylabel('Biases')
        axs[0, 1].legend()

        # Plot Costs
        axs[1, 0].plot(epochs, self.costs, label='Costs', color='green')
        axs[1, 0].set_title('Costs vs Epochs')
        axs[1, 0].set_xlabel('Epochs')
        axs[1, 0].set_ylabel('Costs')
```

```
        axs[1, 0].legend()

        # Plot Outputs
        axs[1, 1].plot(epochs, self.outputs, label='Outputs', color='red')
        axs[1, 1].set_title('Outputs vs Epochs')
        axs[1, 1].set_xlabel('Epochs')
        axs[1, 1].set_ylabel('Outputs')
        axs[1, 1].legend()

        # Adjust layout and display the plots
        plt.tight_layout()
        plt.show()

# Initialize the first neuron and perform SGD
sn1 = single_neuron(0.6, 0.9)
sn1.SGD(0.15, 300)

# Initialize the second neuron and perform SGD
sn2 = single_neuron(2.0, 2.0)
sn2.SGD(0.15, 300)

# Plot the data stored in the objects
sn1.plot()
sn2.plot()
```

R implementation

```
library(dplyr)
library(ggplot2)
library(tidyr)

# Define the functions
sigmoid <- function(x) 1 / (1 + exp(-x))
sigmoid_prime <- function(x) sigmoid(x) * (1 - sigmoid(x))
y <- function(w, b) w + b
a <- function(w, b) sigmoid(y(w, b))
nabla <- function(w, b) a(w, b) * sigmoid_prime(y(w, b))

# Stochastic gradient descent
sgd <- function(w0, b0, eta, epochs) {
    w <- c(w0, rep(0, epochs))
    b <- c(b0, rep(0, epochs))
    for (i in 1:epochs) {
        w[i + 1] <- w[i] - eta * nabla(w[i], b[i])
        b[i + 1] <- b[i] - eta * nabla(w[i], b[i])
    }
    return(rbind(w = w, b = b))
}

# Calculate weights, biases, outputs, and costs
z <- sgd(.6, .9, .15, 300)
z <- rbind(z, out = a(z["w", ], z["b", ]))
z <- rbind(z, cost = z["out", ]^2 / 2)

# Plot the results
as_tibble(t(z)) %>%
    mutate(epochs = (1:n()) - 1) %>%
    pivot_longer(-epochs) %>%
    mutate(name = factor(name, levels = c("w", "b", "out", "cost"), labels
```

```
            =
33          c("weight", "bias", "output", "cost"))) %>%
34      ggplot(aes(epochs, value)) +
35      geom_line() +
36      facet_wrap(~name, scales = "free_y") +
37      labs(x = "Epoch", y = "") +
38      theme_bw()
```

3.2 Cross-entropy loss for binary classification

- y is the true label (typically $y \in \{0, 1\}$),
- a is the predicted probability (output of a model, typically $0 \leq a \leq 1$),
- n is the number of samples.

$$C = -\frac{1}{n} \sum (y \ln(a) + (1 - y) \ln(1 - a))$$

(sum over all samples)

Since $\ln(a) < 0$ and $\ln(1 - a) < 0$, we have $C > 0$.

3.3 Gradient of cross-entropy loss for a single neuron with multiple inputs

$a = \sigma(z)$, where $z = wx + b$.

Thus

$$\begin{aligned} \nabla_w C &= -\frac{1}{n} \sum \left(\frac{y}{a} - \frac{1 - y}{1 - a} \right) a \, (1 - a) \, x \\ &= -\frac{1}{n} \sum \left(y \, (1 - a) - a \, (1 - y) \right) x \\ &= \frac{1}{n} \sum (a - y) \, x \end{aligned}$$

3.4 Exercise: Minimizer of the cross-entropy loss when $y \in [0, 1]$

Since $\frac{dC}{da}(a) = \frac{1}{n} \sum (a - y)$, the cost function is minimized when $a = y$.

3.5 Gradient of cross-entropy loss for many-layer multi-neuron networks

The cross-entropy cost function *(for a single example/observation)* is given by:

$$C = -\sum_j \left(y_j \ln a_j^L + (1 - y_j) \ln(1 - a_j^L) \right)$$

where a_j^L is the output of the jth neuron in the output layer and y_j is the corresponding jth element of the target vector.

The gradient of the cost function with respect to the output layer is:

$$\frac{\partial C}{\partial a_j^L} = -\left(\frac{y_j}{a_j^L} - \frac{1 - y_j}{1 - a_j^L} \right)$$

The derivative of the sigmoid activation function is:

$$\sigma'(z) = \sigma(z) \, (1 - \sigma(z))$$

and thus

$$\sigma'(z^L) = a^L \odot (1 - a^L)$$

The error in the output layer is given by (see Section 2.2):

$$\delta^L = \nabla_a C \odot \sigma'(z^L)$$

Consequently

$$\begin{aligned}\delta_j^L &= \left(\frac{1 - y_j}{1 - a_j^L} - \frac{y_j}{a_j^L}\right) a_j^L (1 - a_j^L) \\ &= \left(a_j^L - y_j\right)\end{aligned}$$

which gives in vector form:

$$\delta^L = (a^L - y)$$

3.6 What does the cross-entropy mean?

Let us take the single-neuron example again: $a = \sigma(z)$, where $z = wx + b$ (x is the input vector and w is the weight matrix).

For the MSE cost function $C = \frac{1}{2}(a - y)^2$, the gradient with respect to w and b is given by:

$$\frac{\partial C}{\partial w_j} = (a - y)\,\sigma'(z)\,x_j, \quad \frac{\partial C}{\partial b} = (a - y)\,\sigma'(z)$$

To avoid the learning slowdown induced by $\sigma'(z)$, we would like to choose a cost function such that the gradient with respect to w and b would be given by:

$$\frac{\partial C}{\partial w_j} = (a - y)\,x_j, \quad \frac{\partial C}{\partial b} = a - y$$

For an arbitrary cost function C, we have

$$\frac{\partial C}{\partial w_j} = \frac{\partial C}{\partial a}\sigma'(z)\,x_j, \quad \frac{\partial C}{\partial b} = \frac{\partial C}{\partial a}\sigma'(z)$$

and, for the sigmoid activation function, we have:

$$\sigma'(z) = a\,(1 - a)$$

Hence, the cost function should satisfy:

$$\frac{\partial C}{\partial a}\,a\,(1 - a) = a - y$$

So, we have to integrate

$$\frac{a - y}{a\,(1 - a)}$$

Since:

$$\frac{a - y}{a\,(1 - a)} = \frac{1 - y}{1 - a} - \frac{y}{a}$$

we have

$$\begin{aligned}C &= \int \frac{a - y}{a\,(1 - a)}\,da \\ &= \int \frac{1 - y}{1 - a}\,da - \int \frac{y}{a}\,da \\ &= -\left[(1 - y)\ln(1 - a) + y\ln(a)\right] + \text{constant}\end{aligned}$$

3.7 Softmax layer

The outputs of a softmax layer are given by:

$$z^l = w^l a^{l-1} + b^l, \quad a^l_j = \frac{e^{z^l_j}}{\sum_k e^{z^l_k}}$$

Monotonicity of the softmax function

$$\begin{aligned}
\frac{\partial a_j}{\partial z_k} &= \frac{\partial}{\partial z_k} \frac{e^{z_j}}{\sum_{j'} e^{z_{j'}}} \\
&= \frac{1}{Z} \frac{\partial}{\partial z_k} e^{z_j} - \frac{1}{Z^2} e^{z_j} \frac{\partial}{\partial z_k} \sum_{j'} e^{z_{j'}} \\
&= \frac{1}{Z} \left(\mathbb{1}_{j=k}\, e^{z_j} - \frac{1}{Z}\, e^{z_j} e^{z_k} \right) \\
&= \frac{e^{z_j}}{Z} \left(\mathbb{1}_{j=k} - \frac{e^{z_k}}{Z} \right) \\
&= a_j \left(\mathbb{1}_{j=k} - a_k \right)
\end{aligned}$$

where $Z = \sum_k e^{z_k}$.

Thus

$$\frac{\partial a_j}{\partial z_k} = \begin{cases} a_j\,(1 - a_j), & \text{if} \quad j = k \\ -a_j\, a_k, & \text{if} \quad j \neq k \end{cases}$$

Inverse of the softmax function

We have:

$$e^{z_j} = a_j Z$$

thus

$$z_j = \ln a_j + \ln Z$$

Note: $\ln Z$ does not depend on the **index** j. However, Z depends on all the z_k.

Log-likelihood cost function

If the desired output is $y = e^{(j)}$ where $e^{(j)}$ is a unit vector with 1 in the jth position and 0 elsewhere, then the likelihood of the observation x is given by

$$L(x) = P(y = e^{(j)} \mid x) = a^L_j(x)$$

The cost function is defined as the negative log-likelihood:

$$C = -\ln(a^L_j)$$

The gradient of the cost function with respect to the weights w^L and biases b^L are computed as

follows:

$$\begin{aligned}
\frac{\partial}{\partial b_i^L}\left(-\ln(a_j^L)\right) &= -\frac{1}{a_j^L}\frac{\partial}{\partial b_i^L}\left(a_j^L\right) \\
&= -\frac{1}{a_j^L}\left(\nabla_{z^L} a_j^L\right)^\top \frac{\partial z^L}{\partial b_i^L} \\
&= -\frac{1}{a_j^L}\sum_{j'}\frac{\partial a_j^L}{\partial z_{j'}^L}\frac{\partial z_{j'}^L}{\partial b_i^L} \\
&= -\frac{1}{a_j^L}\sum_{j'} a_j^L(\mathbb{1}_{j=j'} - a_{j'}^L)\mathbb{1}_{i=j'} \\
&= a_i^L - \mathbb{1}_{i=j}
\end{aligned}$$

and

$$\begin{aligned}
\frac{\partial}{\partial w_{ik}^L}\left(-\ln(a_j^L)\right) &= -\frac{1}{a_j^L}\sum_{j'}\frac{\partial a_j^L}{\partial z_{j'}^L}\frac{\partial z_{j'}^L}{\partial w_{ik}^L} \\
&= -\frac{1}{a_j^L}\sum_{j'} a_j^L(\mathbb{1}_{j=j'} - a_{j'}^L)\mathbb{1}_{i=j'}a_k^{L-1} \\
&= a_k^{L-1}(a_i^L - \mathbb{1}_{i=j})
\end{aligned}$$

3.8 Backpropagation (softmax)

For the log-likelihood cost function and a softmax final layer, we have

$$\begin{aligned}
\frac{\partial C}{\partial z_i^L} &= \frac{\partial}{\partial z_i^L}(-\ln(a_j^L)) \\
&= -\frac{1}{a_j^L}\frac{\partial}{\partial z_i^L}a_j^L \\
&= -\frac{1}{a_j^L}a_j^L\left(\mathbb{1}_{i=j} - a_i^L\right) \\
&= a_i^L - \mathbb{1}_{i=j}
\end{aligned}$$

such that,

$$\delta^L = a^L - y$$

For a linear transition function $z^l = w^l a^{l-1} + b^l$ and an activation function $a^{l-1} = \sigma^{l-1}(z^{l-1})$, we have

$$\delta^{l-1} = J_{\sigma^{l-1}}(z^{l-1})^\top (w^l)^\top \delta^l$$

For a softmax layer σ^l, we have *(à vérifier)*

$$\begin{aligned}
J_{\sigma^l}(z^l) &= \left[a_i^l\left(\mathbb{1}_{i=j} - a_j^l\right)\right]_{ij} \\
&= \mathrm{diag}(a^l) - a^l(a^l)^\top
\end{aligned}$$

3.9 Weight decay / L2 regularization

Regularized cross-entropy:

$$C = -\frac{1}{n}\sum_x\sum_j\left[y_j \ln a_j^L + (1-y_j)\ln(1-a_j^L)\right] + \frac{\lambda}{2n}\sum_w w^2$$

(sum over samples and output categories)

where $\lambda > 0$ is the regularization parameter.

We also write:

$$C = C_0 + \frac{\lambda}{2n} \sum_w w^2$$

where C_0 is the unregularized cost function (e.g. cross-entropy).

We compute $\frac{\partial C}{\partial w}$ and $\frac{\partial C}{\partial b}$ to minimize the cost function:

$$\frac{\partial C}{\partial w} = \frac{\partial C_0}{\partial w} + \frac{\lambda}{n} w$$
$$\frac{\partial C}{\partial b} = \frac{\partial C_0}{\partial b}$$

The partial derivatives with respect to w and b are calculated using backpropagation.

The gradient descent learning rule is then:

$$b \leftarrow b - \eta \frac{\partial C_0}{\partial b}$$

and

$$w \leftarrow w - \eta \left(\frac{\partial C_0}{\partial w} - \frac{\lambda}{n} w \right) = \left(1 - \frac{\eta \lambda}{n} \right) w - \eta \frac{\partial C_0}{\partial w}$$

For stochastic gradient descent, we use the following update rule:

$$b \leftarrow b - \frac{\eta}{m} \sum_x \frac{\partial C_x}{\partial b}$$
$$w \leftarrow \left(1 - \frac{\eta \lambda}{n} \right) w - \frac{\eta}{m} \sum_x \frac{\partial C_x}{\partial w}$$

where m is the number of samples in the mini-batch and C_x is the cost function for sample x.

3.10 Weight initialization

Select the initial weights w from a normal distribution with mean 0 and standard deviation $1/\sqrt{n_{\text{in}}}$:

$$w_j \sim \mathcal{N}(0,\, 1/\sqrt{n_{\text{in}}}).$$

The initial biases are still set according to a $\mathcal{N}(0, 1)$.

3.11 How to choose a neural network's hyper-parameters?