Haoran's Machine Learning Handbook

Haoran Ni

January 2, 2024

Preface

This handbook was written in spring 2023 by Haoran Ni. He is thrilled about all the resources ~~and free textbooks~~ he can find online, yet getting upset that he always forgets important knowledge in a month after carefully reading through them. That is why he decided to write such a handbook to take notes of all the things he considered important in a brief and clear way. Meanwhile, he thinks structuring and writing down such notes as if he is explaining to somebody else helps him understand the knowledge better, salute to Prof. Richard Feynman, and makes it easier for him to quickly look up for equations and theorems in the future. He also hopes that this handbook could be useful to his friends and colleges in case they want to quickly look up for something related. This handbook is written by referring to online resources, published textbooks, and only a small part, such as some comments, detailed derivations and some python codes, comes from Mr. Haoran Ni's own ideas. So he wants the readers to know that he did not *create* all of this, but rather *restructured* and *paraphrased* them. Respect to all the people who shared their knowledge and notes online to make this handbook possible.

Contents

1 Github workflows

Git is a *distributed version control* system that keeps track of all the changes in a *git repository*. **Github** is instead a user-friendly software(?) that is built upon git.

1.1 Create a repository

A repository is the folder that we work on, the folder that git will keep track of. Suppose we go to a folder `example_repo` in the terminal, then type and enter

```
git init
```

Git will automatically create a hidden folder `.git` inside this folder/repo. Now this folder `example_repo` is officially a git repo!

1.2 Stage files

Suppose we just modified the file `index.html` in a repo, and that was a important modification so that we want to have some kind of *savepoints* just like we do in computer games. We use `git commit` to create a savepoint. But before that, we need to move the files that we want to commit to the **staging area** using

```
    git add <file_names>
```

If we have a lot of files to commit, then we can simpy call

```
    git add .
```

to add all the files in the repo to the staging area.

A good habit to have during this process is to call

```
    git status
```

which shows the modified files, and the staged files. Some people may question that the stage process is not necessary. But consider the case where we modified several important files, and they separately corresponds to different features. A clean workflow would be to stage the first feature, commit the first feature, then stage the second feature and commit the second feature, and so on. The reason why staging exists is that, *sometimes we don't want all the modified files to be committed*.

With all that being said, now we are ready to make a commit!

1.3 Commit a file

Simple, just do

```
    git commit -m 'added index title'
```

Here `-m` stands for message, and the message is the quoted sentence. Keep in mind that the message should be clear and indicative. Do not add meaningless messages.

If we want to check the history of our commits, we can call

```
    git log
```

Or if we want the output to look clean, we can use

```
    git log --oneline
```

1.4 Push a commit

After we make a commit, it only affects our local repo. To publish our local commits to the remote github repo, we should do

```
git push
```

1.5 Summary

The general github workflow is:

- Create a folder on your local machine, which will be used as the repo.
- Go under that folder and call `git init`.
- Create and modify some files for a certain feature.
- Call `git status` to check the names of the modified files and staged files.
- Call `git add <file_names>` to stage the files.
- Call `git commit -m 'message'` to commit the files.
- Call `git log` to check the commit history.
- Call `git push` to publish the commit.

2 Maths, again

2.1 Matrix calculus

2.1.1 Derivative of matrices

Reference: Matrix Calculus.

---

1. If

$$y = Ax, \tag{1}$$

where y is $m \times 1$, A is $m \times n$ and is a constant matrix, x is $n \times 1$, then

$$\frac{\partial y}{\partial x} = A. \tag{2}$$

---

2. If

$$\alpha = \omega^T x, \tag{3}$$

where α is scalar, ω is $n \times 1$, x is $n \times 1$, then

$$\frac{\partial \alpha}{\partial x} = \omega^T. \tag{4}$$

---

3. If

$$\alpha = y^T A x, \tag{5}$$

where α is a scalar, y is $m \times 1$, A is $m \times n$, x is $n \times 1$, then

$$\frac{\partial \alpha}{\partial x} = y^T A. \tag{6}$$

$$\frac{\partial \alpha}{\partial y} = \left(\frac{\partial \alpha}{\partial y^T}\right)^T \tag{7}$$
$$= (Ax)^T \tag{8}$$
$$= x^T A^T. \tag{9}$$

4.★ If

$$\alpha = x^T A x, \tag{10}$$

where α is a scalar, A is $n \times n$, x is $n \times 1$, then

$$\frac{\partial \alpha}{\partial x} = x^T (A + A^T). \tag{11}$$

---

5. If

$$y = Ax, \tag{12}$$

where y is $m \times 1$, A is $m \times n$ and is a constant matrix, x is $n \times 1$, then

$$\frac{\partial y}{\partial z} = A \frac{\partial x}{\partial z}. \tag{13}$$

---

6. If

$$\alpha = y^T x, \tag{14}$$

where α is a scalar, ω is $n \times 1$, x is $n \times 1$, and they are all functions of z, then

$$\frac{\partial \alpha}{\partial z} = x^T \frac{\partial y}{\partial z} + y^T \frac{\partial x}{\partial z}. \tag{15}$$

---

7. If

$$\alpha = y^T A x, \tag{16}$$

where y is $m \times 1$, A is $m \times n$ and is a constant matrix, x is $n \times 1$, then

$$\frac{\partial \alpha}{\partial z} = x^T A^T \frac{\partial y}{\partial z} + y^T A \frac{\partial x}{\partial z}. \tag{17}$$

2.1.2 Jacobian matrix

Suppose we have a function $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^m$, namely this function brings $\mathbf{x} \in \mathbb{R}^n$ to $\mathbf{f}(\mathbf{x}) \in \mathbb{R}^m$. Then the **Jacobian matrix** of $\mathbf{f}$ is an $m \times n$ matrix defined by $\mathbf{J}_{ij} = \partial f_i / \partial x_j$, namely

$$\mathbf{J} = \begin{bmatrix} \dfrac{\partial \mathbf{f}}{\partial x_1} & \cdots & \dfrac{\partial \mathbf{f}}{\partial x_n} \end{bmatrix} = \begin{bmatrix} \nabla f_1 \\ \vdots \\ \nabla f_m \end{bmatrix} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix}, \tag{18}$$

where $\partial \mathbf{f} / \partial x_i$ are column vectors and ∇f_i are row vectors.

2.1.3 Positive-definite matrix and its friends

Positive definite Given a complex matrix H, it is called **positive definite** if

$$\mathcal{R}(v^* H v) > 0 \tag{19}$$

for any non-zero complex vector v, where v^* is the complex conjugate transpose of v. For real matrix the above equation reduces to

$$v^T H v > 0 \tag{20}$$

for any non-zero real vector v, where v^T is the transpose of v.

Positive semi-definite Given a complex matrix H, it is called **positive semi-definite** if

$$\mathcal{R}(v^* H v) \geq 0 \tag{21}$$

for any non-zero complex vector v, where v^* is the complex conjugate transpose of v. For real matrix the above equation reduces to

$$v^T H v \geq 0 \tag{22}$$

for any non-zero real vector v, where v^T is the transpose of v.

Negative definite and negative semi-definite Just change the $>$ sign to $<$ sign.

For convenience, we will only talk about positive-definite in the following parts, other cases can be derived accordingly.

2.1.4 Positive-definite for Hermitian and symmetric matrices

For Hermitian matrix H, we have its eigenfunction

$$H \cdot \vec{a_i} = \alpha_i \, \vec{a_i}. \tag{23}$$

It is easy to prove that all a_i are real numbers. For an arbitrary vector v, we always have

$$\vec{v} = \sum_i c_i \vec{a_i}, \tag{24}$$

because a_i form orthonormal bases. So we have

$$v^* H v = \sum_i \sum_j c_i c_j \vec{a_i}^* H \vec{a_j} \tag{25}$$

$$= \sum_i \sum_j c_i c_j \alpha_j \vec{a_i}^* \vec{a_j} \tag{26}$$

$$= \sum_i \sum_j c_i c_j \alpha_j \delta_{ij} \tag{27}$$

$$= \sum_i c_i^2 \alpha_i. \tag{28}$$

The above result suggests that if H is positive-definite, then all of its eigenvalues α_i should be positive. So we have: **For Hermitian/symmetric matrices, they are positive-definite if and only if all of the eigenvalues are positive**.

Because for any complex matrix A (the steps are similar for real matrices), we can always find its corresponding Hermitian part by doing

$$A_H = \frac{1}{2}(A + A^\dagger). \tag{29}$$

We can prove that **if the Hermitian part of a complex matrix is positive-definite, then this matrix is positive-definite**. This is a second way of telling if a matrix is positive-definite or not. We can also prove that the determinant of a positive definite matrix is always larger than 0, $det(A) > 0$.

2.1.5 Significance of positive-definite

There are many cases where we meet the mathematical form $v^T Hv$. Here we give two examples.

Hessians For a multi-variable function $f(\mathbf{x})$, when we take the Fourier expansion to the second order,

$$f(\mathbf{x}') = f(\mathbf{x}) + (\mathbf{x}' - \mathbf{x})^T \nabla f(\mathbf{x}) + \frac{1}{2}(\mathbf{x}' - \mathbf{x})^T \mathbf{H}(\mathbf{x}' - \mathbf{x}) + \mathcal{O}(\delta \mathbf{x}^3). \tag{30}$$

We naturally see the form $v^T Hv$, and H is called the Hessian matrix of f, which is symmetric. **If the Hessian matrix of a function is positive definite, then this function is convex**.

Observable averages in quantum mechanics In quantum mechanics, $v^T Hv$ is represented as $\langle a|H|a\rangle$, which stands for the average of an observable for a certain state. **For a positive-definite observable, the observation will always be positive**.

2.2 Calculus of variation

We consider the following problem of finding the optimal "path" from point a to point b: We denote

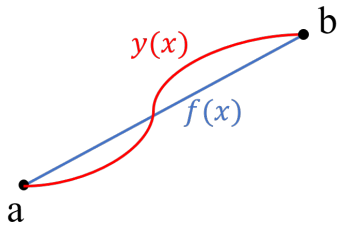


Figure 1: Example to introduce Calculus of variation.

the optimal path as $f(x)$, and any random path as $y(x)$. A **variation** to $f(x)$, or more generally $y(x)$ is denoted as $\epsilon\eta(x)$, where ϵ is infinitesimally small, and $\eta(x)$ is a random function which satisfies $\eta(a) = \eta(b) = 0$. We denote all possible paths from a to b as a **functional**, which stands for the function of a function, $F[y]$. (*A commonly seen functional in machine learning is the entropy $H[x]$ for continuous variables, because it is the function/integral of the probability distribution $p(x)$. $H[x]$ could also be written as $H[p]$.*)

Apparently, $F[y]$ depends on the function form of y, and reaches its minimum when $y(x) = f(x)$. If we now focus on the region near the minimal path $f(x)$, and let $y(x) = f(x) + \epsilon\eta(x)$, then for $\epsilon = 0$, we should have

$$\frac{\partial F[y]}{\partial \epsilon}|_{\epsilon=0} = \frac{\partial F[f]}{\partial \epsilon} = 0. \tag{31}$$

Generally, $F[y]$ could be written as

$$F[y] = \int_a^b G(y, y', x)dx. \tag{32}$$

Therefore, to find the optimal path $f(x)$, we have

$$\frac{\partial F[y]}{\partial \epsilon} = \int_a^b \frac{\partial G(y, y', x)}{\partial \epsilon} dx \tag{33}$$

$$= \int_a^b \left(\frac{\partial G}{\partial y}\frac{\partial y}{\partial \epsilon} + \frac{\partial G}{\partial y'}\frac{\partial y'}{\partial \epsilon} \right) dx \tag{34}$$

$$= \int_a^b \left(\frac{\partial G}{\partial y}\eta(x) + \frac{\partial G}{\partial y'}\eta'(x) \right) dx \tag{35}$$

$$= \int_a^b \frac{\partial G}{\partial y}\eta(x)dx + \int_a^b \frac{\partial G}{\partial y'}d\eta(x) \tag{36}$$

$$= \int_a^b \frac{\partial G}{\partial y}\eta(x)dx + \frac{\partial G}{\partial y'}\eta(x)|_a^b - \int_a^b \eta(x)d\left(\frac{\partial G}{\partial y'}\right) \tag{37}$$

$$= \int_a^b \frac{\partial G}{\partial y}\eta(x)dx - \int_a^b \eta(x)\frac{d}{dx}\left(\frac{\partial G}{\partial y'}\right)dx \tag{38}$$

$$= \int_a^b \left(\frac{\partial G}{\partial y} - \frac{d}{dx}\left(\frac{\partial G}{\partial y'}\right)\right)\eta(x)dx \tag{39}$$

$$= 0. \tag{40}$$

Because the above equation must hold for any function $\eta(x)$, therefore we have

$$\frac{\partial G}{\partial y} - \frac{d}{dx}\left(\frac{\partial G}{\partial y'}\right) = 0, \tag{41}$$

which is called the **Euler-Lagrange equation**. Solving the above equation gives us the optimal path $f(x)$.

2.2.1 Finding the shortest distance between two points on a Euclidean plane

2.2.2 The maximum entropy distribution of a continuous variable is a Gaussian

2.3 Moments and characteristic functions

2.4 Probability theory

Remember by heart the fundamental rules in probability theory:

- Sum rule: $p(X) = \sum_Y p(X, Y)$. $p(X)$ is called marginalization probability. The summing out of another variable is called marginalization.
- Product rule: $p(X, Y) = p(X|Y)p(Y) = p(Y|X)p(X)$
- Sum rule combined with Product rule: $p(X) = \sum_Y p(X|Y)p(Y)$
- $\sum_X p(X|Y) = 1$
- Bayes' theorem (derived from product rule):

$$p(Y|X) = \frac{p(X|Y)p(Y)}{p(X)} = \frac{p(X|Y)p(Y)}{\sum_Y p(X|Y)p(Y)} \tag{42}$$

- X and Y are said to be independent if $p(X, Y) = p(X)p(Y)$. So for two independent variables, $p(X|Y) = p(X)$, $p(Y|X) = p(Y)$.

prior probability and posterior probability The available probability before we make any observation is called the **prior probability**, e.g. there are two boxes and we decide we will select the red one with a 60% chance, so the prior probability is $p(B = red) = 0.6$. The probability after we make observations is called **posterior probability**, e.g. we observed that an apple is selected from one of the boxes, then the probability that we selected the red box is now $p(B = red|F = apple)$, which is a posterior probability. It can be calculated using Bayes' theorem.

Probability density The probability of a continuous variable x falling into the region $(x, x + \delta x)$ is expressed as $p(x)\delta x$, and $p(x)$ is called the probability density of x. Note that it has the inverse unit of x. We have, in analogous to the discrete version,

- Sum rule: $p(x) = \int_y p(x, y)dy$
- Product rule: $p(x, y) = p(x|y)p(y) = p(y|x)p(x)$
- Beyes' theorem: $p(y|x) = \frac{p(x|y)p(y)}{p(x)} = \frac{p(x|y)p(y)}{\int_y p(x|y)p(y)}$
- $\int_x p(x)dx = 1$
- **Cumulative distribution function**: $P(z) = \int_{-\infty}^z p(x)dx$, which satisfies $P'(x) = p(x)$

Expectation The expectation of a function $f(x)$ where x has a probability distribution of $p(x)$ is written as

$$\mathbb{E}[f(x)] = \int_x p(x)f(x)dx \quad or \quad \sum_X p(X)f(X). \tag{43}$$

For multivariate functions, such as $f(x, y)$, it will be indicated on which the expectation is averaged, e.g. $\mathbb{E}_x[f(x, y)]$ will be a function of y.

Conditional expectation It refers to the expectation of a conditional probability distribution

$$\mathbb{E}_x[f(x)|y] = \int_x p(x|y)f(x)dx \quad or \quad \sum_X p(X|Y)f(X) \tag{44}$$

Variance It is defined as

$$var[f(x)] = \mathbb{E}\left[(f(x) - \mathbb{E}[f(x)])^2\right], \tag{45}$$

which can be simplified as

$$var[f] = \mathbb{E}[f(x)^2] - \mathbb{E}[f(x)]^2. \tag{46}$$

Covariance It is defined as

$$cov[x, y] = \mathbb{E}_{x,y}[\{x - \mathbb{E}[x]\}\{y - \mathbb{E}[y]\}] = \mathbb{E}_{x,y}[xy] - \mathbb{E}[x]\mathbb{E}[y], \tag{47}$$

which measures the correlation of two variables. If two variables are independent, then their covariance is 0. **The covariance of the same variable, i.e. two identically sampled variables, is NOT defined**.

The covariance of two vectors is a matrix of the form

$$cov[\mathbf{x}, \mathbf{y}] = \mathbb{E}_{\mathbf{x},\mathbf{y}}[\{\mathbf{x} - \mathbb{E}[\mathbf{x}]\}\{\mathbf{y}^T - \mathbb{E}[\mathbf{y}^T]\}] = \mathbb{E}_{\mathbf{x},\mathbf{y}}[\mathbf{x}\mathbf{y}^T] - \mathbb{E}[\mathbf{x}]\mathbb{E}[\mathbf{y}^T], \tag{48}$$

where the average operation is for each element. We can also measure the covariance of the elements in vector $\mathbf{x}$ by calculating $cov[\mathbf{x}] = cov[\mathbf{x}, \mathbf{x}]$.

2.4.1 The frequentist's interpretation and the Bayesian interpretation

Let's take polynomial curve fitting as an example. Under the frequentist's interpretation, the parameters $\mathbf{w}$ are fixed, it does not have any probability distribution (i.e. we don't consider $p(\mathbf{w}|\mathbf{x}, \mathbf{t})$). And the probability we write in this case brings us **directly** to the likelihood function $p(\mathbf{t}|\mathbf{x}, \mathbf{w}, \beta)$, by which we maximize to obtain the sum-of-squares error function.

A crucial step that brings us to the Bayesian approach is considering the probability distribution of the parameters $\mathbf{w}$, taken as a prior probability, $p(\mathbf{w}|\mathbf{x}, \mathbf{t})$. This is the very term, when multiplied with the likelihood function, that gives us the regularization term, because the probability distribution regularizes the range of $\mathbf{w}$.

The key difference between frequentist's interpretation and Bayesian interpretation is that there are more than one variable with probability distribution in the latter. For example, in polynomial curve fitting, under the freq. approach, the only probability is for the prediction of the target variable, which takes the form of a Gaussian distribution, and the parameters in the model has no probability distribution, it is fixed! The ignored complexity of w is reflected on the fact that we need several such training sets to know the error bar of w.

However, in Bayesian approach, we add the consideration of the probability distribution of the parameters w, such consideration brings us to the case where we can use the sum rule and product rule, which only apply when we have several probabilistic variables. And the following expression applies:

$$p(t|x, \mathbf{x}, \mathbf{t}) = \int p(t|x, \mathbf{w})p(\mathbf{w}|\mathbf{x}, \mathbf{t})d\mathbf{w}, \tag{49}$$

which is just sum rule combined with product rule for continuous variables. The Bayesian approach could be understood using the following diagram:

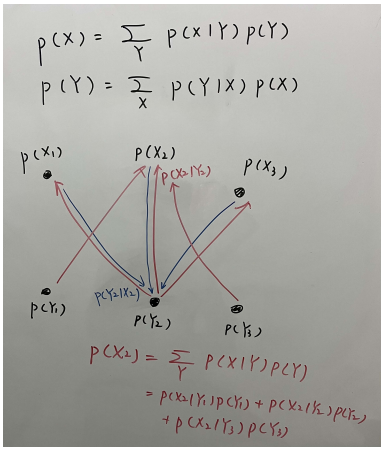


Figure 2: Diagram of Bayesian approach

It is pretty like the idea of path integral, where in order to reach the ultimate and only variable (here is the predicted value t), different paths have to be considered (here is the different possible values of w).

More detailed coverage is on the way.

2.4.2 The Gaussian distribution

Gaussian distribution, or normal distribution is probably the most important probability distribution for **continuous** variables. For a single real-valued variable x obeying Gaussian distribution, the Gaussian distribution is defined as

$$\mathcal{N}(x|\mu,\sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}}\exp\left\{-\frac{(x-\mu)^2}{2\sigma^2}\right\}, \tag{50}$$

where μ is called the **mean**, σ^2 is called the **variance**, σ is called the **standard deviation**. We also define $\beta = 1/\sigma^2$, which is called the **precision**.

Here are some properties of the Gaussian distribution:

- $\int_\infty^\infty \mathcal{N}(x|\mu,\sigma^2)dx = 1$
- $\mathbb{E}[x] = \int_\infty^\infty x\mathcal{N}(x|\mu,\sigma^2)dx = \mu$
- $\mathbb{E}[x^2] = \int_\infty^\infty x^2\mathcal{N}(x|\mu,\sigma^2)dx = \mu^2 + \sigma^2$
- $var[x] = \mathbb{E}[x^2] - \mathbb{E}[x]^2 = \sigma^2$

2.5 A bit of information theory, and a bit of statistical mechanics

2.5.1 Entropy from the view of a data scientist

To measure the amount of information contained in an observation x, the quantity $h(x)$ should satisfy the following properties:

- $h(x)$ measures the "degree of surprise", in a way that the information content is 0 if we observe something that is sure to happen, i.e. $h(x) = 0$ when $p(x) = 1$; the information content is the maximum when we observe something that is very unlikely to happen, i.e. $h(x) = max$ when $p(x) = 0$.
- When we have two independent events, the information content of these two is the sum of their individual information content, i.e. $h(x,y) = h(x) + h(y)$ if $p(x,y) = p(x)p(y)$.

Therefore, $h(x)$ must be a logarithm function of $p(x)$,

$$h(x) = -\log_2 p(x). \tag{51}$$

The minus sign here is to make sure $h(x)$ is positive. The base of the logarithm could be other numbers, and $h(x)$ with base 2 has a unit of **bits**.

Then the average information content of variable x with probability distribution $p(x)$ is

$$H[x] = -\sum_x p(x) \log_2 p(x). \tag{52}$$

This quantity is called the **entropy** of the random variable x.

Uniformly distributed variables Now consider we have 8 possible states of x, all having the same probability 1/8, the entropy of such a distribution is

$$H[x] = -8 \times \frac{1}{8} \log_2 \frac{1}{8} = 3 \; bits. \tag{53}$$

Non-uniformly distributed variables Now consider we still have 8 possible states of x, but now the probability distribution is $\{\frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \frac{1}{16}, \frac{1}{64}, \frac{1}{64}, \frac{1}{64}, \frac{1}{64}\}$. The entropy is now

$$H[x] = \ldots = 2 \; bits. \tag{54}$$

Why is that? Because we can encode each possible state of x using 0, 10, 110, 1110, 111100, 111101, 111110, 111111, the average length of bits is

$$\frac{1}{2} \times 1 + \frac{1}{4} \times 2 + \frac{1}{8} \times 3 + \frac{1}{16} \times 4 + \frac{1}{64} \times 6 \times 4 = 2. \tag{55}$$

Note that the encoding can not be shorter because we need to make sure it is always possible to disambiguate a concatenation of such strings, e.g. 11001110 decodes uniquely to 110, 0, 1110.

The noiseless coding theorem by Shannon The entropy is a lower bound on the number of bits needed to transmit the state of a random variable.

2.5.2 Entropy from the view of a physicist

Hello, and yes I have a Master degree in physics...So in physics the concept of entropy comes from statistical physics, when people were trying to figure out all the possible states of particles moving in a box. This problem could be described as: Suppose we have N particles that we want to randomly put them into a set of bins (imagine that as different positions in space), and we suppose there are n_i particles in the ith bin. The number of different ways that we can allocate those particles is

$$W = C_N^{n_1} C_{N-n_1}^{n_2} C_{N-n_1-n_2}^{n_3} \cdots C_0^0 \tag{56}$$

$$= \frac{N!}{n_1!(N-n_1)!} \frac{(N-n_1)!}{n_2!(N-n_1-n_2)!} \frac{(N-n_1-n_2)!}{n_3!(N-n_1-n_2-n_3)!} \cdots \frac{0!}{0!0!} \tag{57}$$

$$= \frac{N!}{\prod_i n_i!}. \tag{58}$$

The entropy is defined as the logarithm of W scaled by N,

$$H = \frac{1}{N} \ln W = \frac{1}{N} \ln N! - \frac{1}{N} \sum_i \ln n_i!. \tag{59}$$

Now we consider the limit $N \to \infty$ while n_i/N are fixed, and use **Sterling's approximation**

$$\ln N! \simeq N \ln N - N, \tag{60}$$

then

$$H = \frac{1}{N}\ln N! - \frac{1}{N}\sum_i \ln n_i! \tag{61}$$

$$= \frac{1}{N}(N\ln N - N) - \frac{1}{N}\sum_i (n_i \ln n_i - n_i) \tag{62}$$

$$= \ln N - \frac{1}{N}\sum_i n_i \ln n_i \tag{63}$$

$$= \frac{\sum_i n_i}{N}\ln N - \frac{1}{N}\sum_i n_i \ln n_i \tag{64}$$

$$= -\sum_i \frac{n_i}{N}\ln\frac{n_i}{N} \tag{65}$$

$$= -\sum_i p_i \ln p_i. \tag{66}$$

Here p_i is the probability of the particle being assigned to the ith bin.

2.5.3 The continuous case - differential entropy

Mean value theorem This theorem tells us that, if we divide x into bins of width Δ, for each bin, there must exist a value x_i in the bin such that

$$\int_{i\Delta}^{(i+1)\Delta} p(x)dx = p(x_i)\Delta. \tag{67}$$

Thus it is always possible to transform the entropy of a continuous variable as

$$H = -\sum_i p(x_i)\Delta \ln\left(p(x_i)\Delta\right) = -\sum_i p(x_i)\Delta \ln p(x_i) - \ln\Delta. \tag{68}$$

Taking the limit $\Delta \to 0$, and omit the diverging $\ln\Delta$ term, we naturally have the entropy for continuous variables

$$H = -\int p(x)\ln p(x)dx, \tag{69}$$

which is called the **differential entropy**.

2.5.4 Conditional entropy

Now we consider a joint probability distribution $p(x, y)$, which leads to the case of the **conditional entropy**.

If we consider the marginal distribution $p(x)$, then

$$H = -\int p(x)\ln p(x)dx \tag{70}$$

$$= -\int p(x,y)\ln p(x)dxdy, \tag{71}$$

where we have used $\int p(x, y)dy = p(x)$. Therefore, we see that for joint distribution, if we include 2 (or several) integrated variables, the probability distribution outside the logarithm function should be the joint distribution.

So the conditional entropy of y given x is

$$H[y|x] = -\int p(x,y)\ln p(y|x)dxdy, \tag{72}$$

which is just

$$= -\int p(y|x)p(x)\ln p(y|x)dxdy \tag{73}$$

$$= -\int p(y|x)\ln p(y|x)dy. \tag{74}$$

And because $p(x,y) = p(x|y)p(y)$, we naturally have

$$H[x,y] = H[x|y] + H[y] = H[y|x] + H[x]. \tag{75}$$

2.5.5 Maximum entropy

Before we introduce the maximum entropy, we want to mention a very important method, which is called the **Lagrange multiplier**. It is always used when we want to maximize/minimize a function under certain constraints.

Suppose we have a function $f(x)$ that we want to maximize, and there is a constraint for x such that $g(x) = 0$, then we just maximize the following function w.r.t. x

$$h(x) = f(x) + \lambda g(x), \tag{76}$$

where $\lambda g(x)$ is called the Lagrange multiplier. After this step we will have x containing λ. λ can be calculated by substituting x back to $g(x) = 0$.

Discrete case Under the discrete case, we want to maximize

$$\tilde{H} = -\sum_i^M p(x_i)\ln p(x_i) + \lambda(\sum_i^M p(x_i) - 1). \tag{77}$$

Taking the derivative of the above function w.r.t. each $p(x_i)$ and substituting $p(x_i)$ back to $\sum_i p(x_i) = 1$ to solve λ, we obtain

$$p(x_i) = \frac{1}{M}. \tag{78}$$

Therefore, for the discrete case, the distribution that has the highest entropy is the uniform distribution. And the corresponding maximum entropy is

$$H_{max} = \ln M. \tag{79}$$

Continuous case In order to better define the problem, we have to make sure the first and second **moments** of $p(x)$ should be constrained.

$$\int_{-\infty}^{\infty} p(x)dx = 1, \tag{80}$$

$$\int_{-\infty}^{\infty} xp(x)dx = \mu, \tag{81}$$

$$\int_{-\infty}^{\infty} (x-\mu)^2 p(x)dx = \sigma^2. \tag{82}$$

So we have to maximize the following **functional** w.r.t. $p(x)$

$$\tilde{H} = -\int_{\infty}^{\infty} p(x)\ln p(x)dx + \lambda_1\left(\int_{-\infty}^{\infty} p(x)dx - 1\right) + \lambda_2\left(\int_{-\infty}^{\infty} xp(x)dx - \mu\right) + \lambda_3\left(\int_{-\infty}^{\infty} (x-\mu)^2 p(x)dx - \sigma^2\right). \tag{83}$$

Taking the derivative of $\tilde{H}$ w.r.t. $p(x)$ gives

$$\frac{\partial \tilde{H}}{\partial p(x)} = -(\ln p(x) + 1) + \lambda_1 + \lambda_2 x + \lambda_3(x-\mu)^2 = 0. \tag{84}$$

Note that here we are taking the derivative of $\tilde{H}$ w.r.t. to a certain x, so the integral disappears.

We obtain

$$p(x) = e^{\lambda_3(x-\mu)^2+\lambda_2 x+\lambda_1+1}. \tag{85}$$

Substituting this back to the constraints, we obtain

$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp -\frac{(x-\mu)^2}{2\sigma^2}. \tag{86}$$

We see that the probability that maximizes the differential entropy is the Gaussian!

2.6 Message passing

3 Before neural networks

Root mean square error (RMSE) It is defined by

$$E_{RMS} = \sqrt{\frac{\sum_n (y_n - t_n)^2}{N}}, \tag{87}$$

where y_n are the predicted values, and t_n the target values. The N in the denominator here allows us to compare the prediction errors on datasets with different sizes. The square root ensures E_{RMS} is on the same scale and same unit with the target value.

Overfitting Overfitting happens when the model has too many parameters yet the dataset has too small training points. Overfitting can be solved by either applying regularization to the cost function, or incease the size of the training set, or reduce the complexity and flexibility of the model. Roughly speaking, the number of training data should approximately be no less than 5 - 10 times of the number of parameters in the model. In the case of overfitting, the weights $\mathbf{w}$ are very large, in a sense that a slight change of the input parameter could cause a large change in the output. So for the regularization we usually use $\frac{\lambda}{2}\|\mathbf{w}\|^2$ to suppress $\mathbf{w}$. The particular case of a **quadratic regularizer** is called **ridge regression**.

3.1 Support vector machine

Support vector machines (SVMs) are supervised machine learning models that can do classifications by maximizing the margins of hyperplanes.

A typical illustration is

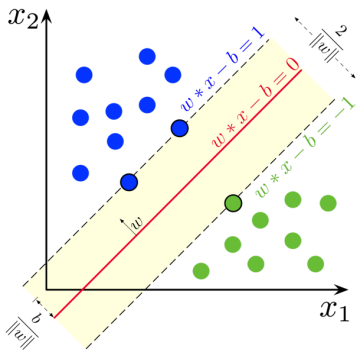


Figure 3: Maximum-margin hyperplane and margins for an SVM trained with samples from two classes. Samples on the margin are called the support vectors.

The key here is the construction of a hyperplane of the form

$$f(\mathbf{x}) = \mathbf{w}\mathbf{x} + b, \tag{88}$$

which is extremely similar to a linear regression model. But the key difference is that, in SVM, $f(\mathbf{x})$ is constructed to be as far from the data points as possible (in order to maximize the margin). While in linear regression, $f(\mathbf{x})$ is constructed to be as close to all data points as possible.

3.2 Linear regression and ridge regression

To better illustrate the definitions, we here consider the problem where we want to find the relations between students' final grades and their ages, genders, homelands and wealth.

In this case, each student is an *observation*, a student's age, gender, homeland and wealth are called *independent variables/features*. It could be written into a *feature matrix* or *design matrix* Φ of size $N \times M$, where N is the number of *samples*, or here we meant the number of students, and M is the number of features, here $M = 4$. Mathematically,

$$\Phi(\vec{x}) = \begin{pmatrix} \phi_1(x_1) & \phi_2(x_1) & ... & \phi_M(x_1) \\ \phi_1(x_2) & \phi_2(x_2) & ... & \phi_M(x_2) \\ ... & ... & ... & ... \\ \phi_1(x_N) & \phi_2(x_N) & ... & \phi_M(x_N) \end{pmatrix}. \tag{89}$$

A student's final grade is called *dependent variable/response*. The real values of students' final grades are called *target values*, denoted as $\vec{t}$. Note that $\vec{t}$ is a vector of size $N \times 1$. The predicted, or estimated values of the target values are denoted as $\vec{y}$, which is also of size $N \times 1$.

Now, the regression task is simply to find the *weights* $\vec{w}$ of the linear model,

$$y_n = \sum_i^M w_i \phi_i(x_n), \tag{90}$$

or

$$\vec{y}(\vec{x}) = \Phi(\vec{x})\vec{w}, \tag{91}$$

that minimize the *loss function*

$$\mathcal{L} = \frac{1}{2N}\sum_i^N (y_i - t_i)^2 + \frac{\lambda}{2}\sum_i^M w_i^2 \tag{92}$$

$$= \frac{1}{2N}\|\vec{y} - \vec{t}\|^2 + \frac{\lambda}{2}\|\vec{w}\|^2. \tag{93}$$

The first term in the above equation is the conventional *least square loss*, the second term is the penalty term to prevent the model from *overfitting*. Ridge regression refers to the linear regression with the penalty term in the loss function.

Linear regression model is analytically solvable, by taking the first order derivative of the loss function with respect to $\vec{w}$, and set it to zero, then we have the expression for the optimized $\vec{w}$.

$$\nabla_{\vec{w}}\mathcal{L} = \frac{1}{N}(\Phi\vec{w} - \vec{t})^T\Phi + \lambda\vec{w}^T = 0, \tag{94}$$

$$\lambda\vec{w}^T = -\frac{1}{N}(\Phi\vec{w} - \vec{t})^T\Phi, \tag{95}$$

$$\lambda\vec{w} = -\frac{1}{N}\Phi^T(\Phi\vec{w} - \vec{t}), \tag{96}$$

$$\lambda\vec{w} = -\frac{1}{N}\Phi^T\Phi\vec{w} + \frac{1}{N}\Phi^T\vec{t}, \tag{97}$$

$$(N\lambda\mathbb{I} + \Phi^T\Phi)\vec{w} = \Phi^T\vec{t}, \tag{98}$$

$$\vec{w} = (\lambda'\mathbb{I} + \Phi^T\Phi)^{-1}\Phi^T\vec{t}, \tag{99}$$

where we have used $(AB)^T = B^T A^T$, and $\lambda' = N\lambda$. One thing to note is that, from Eq.(94) to Eq.(95), we take the transpose of some matrices. Because, the derivative of $\Phi\vec{w}$ w.r.t. $\vec{w}$ is Φ, which is of size $N \times M$, therefore the matrix to the left of Φ must have N columns, so we have to take the transpose of $(\Phi\vec{w} - \vec{t})$.

There may be doubts that how can we be sure we reach the minimal point by taking the derivative to zero, it is also possible that we reach the maximal points. Good question! To erase this doubt, we take the second derivative of the loss function

$$\nabla^2_{\vec{w}}\mathcal{L} = \frac{d}{d\vec{w}}(\Phi^T\Phi\vec{w} - \Phi^T\vec{t} + \lambda\vec{w}) = \Phi^T\Phi + \lambda = \|\Phi\|^2 + \lambda, \tag{100}$$

which is valid.

Reminder: check positive definite matrix, semi positive definite matrix etc....

There are some commonly used linear algebra operations here, such as $(A+B)^T = A^T + B^T$ and $(AB)^T = B^T A^T$. I put them here just in case I forget them AGAIN in the future. And keep this in mind, as long as the expression is still valid, there is no change if we take the transpose of all the terms in that expression.

3.3 Kernel method

3.4 Cross validation

See this webpage. k-fold cross validation is also introduced.

3.5 The coefficient of determination R2

See this webpage for R2 score. It is also implemented in sklearn. R2 = 1 means the model is perfectly accurate, R2 = 0 means the model is purely flat and is learning nothing. R2 could be negative because the model could be arbitrarily bad.

4 Neural networks

4.1 Perceptron

The very elementary version of a neuron in neural networks resembles perceptron, which takes binary inputs and yields binary outputs.

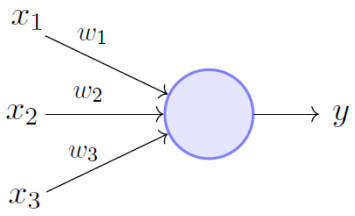


Figure 4: Perceptron

The algebraic expresson of a perceptron is

$$y = \begin{cases} 0 & \text{if } \sum_i \omega_i x_i + b \leq 0 \\ 1 & \text{if } \sum_i \omega_i x_i + b > 0 \end{cases}, \tag{101}$$

where ω are **weights**, and b is the **bias**.

4.2 Sigmoid neurons

When we talk about *learning*, we are in fact referring to the process of adjusting weights and biases in the neural network so that the cost function is minimized. This is achievable only when we have *continuous* cost functions. To be more specific, the weights are biases are optimized as

$$\omega' = \omega - \eta \frac{\partial C(\omega, b)}{\partial \omega}, \tag{102}$$

$$b' = b - \eta \frac{\partial C(\omega, b)}{\partial b}, \tag{103}$$

where $C(\omega, b)$ is the cost function, it's variables are weights and biases, and η is the **learning rate**. Following Eq.(102) and Eq.(103), we can always make sure that the cost function C will decrease once ω and b are stepped.

The problem with perceptrons is that, no matter what the form of the cost function is, it will always be discrete because the output is discrete. So we are not sure to which direction should the parameters of the model be adjusted. That is why we introduce **sigmoid neurons**.

Sigmoid functions are functions with S-shape as below.

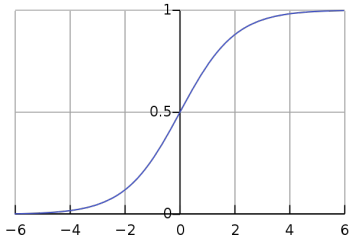


Figure 5: Sigmoid function, using logistic function here as an example

A commonly used sigmoid function is the **logistic function**, which is

$$\sigma(x) = \frac{1}{1 + e^{-kx}}, \tag{104}$$

where k is called the **growth rate**. The logistic function gives us a satisfying smoothed version of the **Heaviside step function**, and it is easy to prove that, as $k \to \infty$, the logistic function approximates the step function.

The structure of a sigmoid neuron is shown in Fig.(6). In this case the **activation function** is just the logistic function. *For perceptrons, the activation function is the Heaviside step function.*

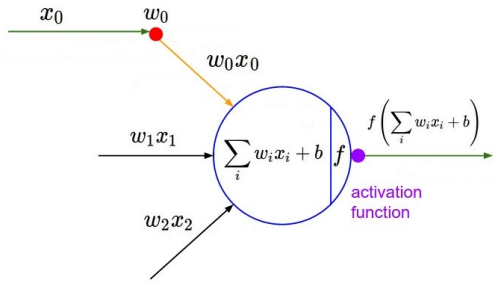


Figure 6: Sigmoid neuron

Now we have continuous outputs, we are ready to introduce the overall structure of neural networks.

4.3 Structure of neural networks

Here is the overall structure of neural networks:

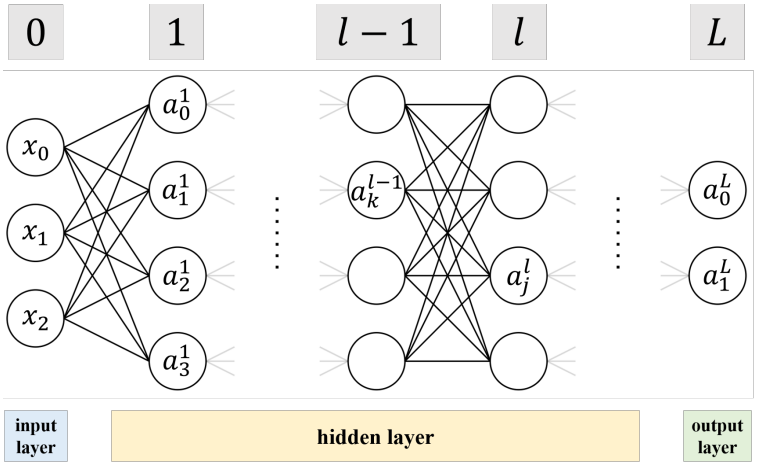


Figure 7: Labels of neural network. We denote the index of each layer in the grey box above. Note that the count starts from 0, respecting Python indexing.

If we zoom in into the l-th layer of the NN,

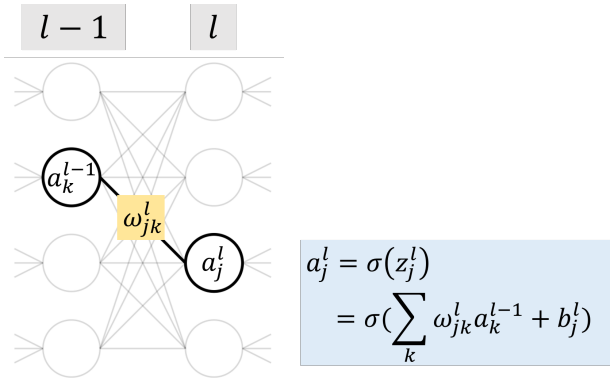


Figure 8: Mechanism of neural networks. $\sigma(x)$ is the sigmoid function, or other activation functions.

So for a NN with L layers ($L+1$ layers if we count the input layer), and n_l neurons in each layer, $l = 0, 1, 2, \cdots, L$, we have the number of parameters

$$N_{weight} = \sum_{l=1}^{L} n_l \cdot n_{l-1}, \tag{105}$$

$$N_{bias} = \sum_{l=1}^{L} n_l, \tag{106}$$

$$N_{total} = \sum_{l=1}^{L} (n_l \cdot n_{l-1} + n_l) = \sum_{l=1}^{L} n_l (n_{l-1} + 1). \tag{107}$$

4.4 Back propagation

4.4.1 Gradient descent and stochastic gradient descent

Previously we mentioned that in order to minimize the cost function $C(\omega, b)$, we have

$$\Delta C = \frac{\partial C}{\partial \omega}\Delta\omega + \frac{\partial C}{\partial b}\Delta b, \tag{108}$$

if we update ω and b as

$$\omega' = \omega - \eta \left.\frac{\partial C}{\partial \omega}\right|_{\omega} \quad \rightarrow \quad \Delta\omega = -\eta \left.\frac{\partial C}{\partial \omega}\right|_{\omega}, \tag{109}$$

$$b' = b - \eta \left.\frac{\partial C}{\partial b}\right|_{b} \quad \rightarrow \quad \Delta b = -\eta \left.\frac{\partial C}{\partial b}\right|_{b}, \tag{110}$$

where η is the learning rate, and the derivative is evaluated at its original ω and b values. Then we always have

$$\Delta C \leq 0. \tag{111}$$

This is called **gradient descent**. Note that for practical training, the cost function $C(\omega, b)$ is the sum of all training samples, namely, $\sum_i^N C_i(\omega, b)$, where N is the size of the training set. The optimization should actually go as

$$\omega' = \omega - \eta \sum_i^N \frac{\partial C_i(\omega, b)}{\partial \omega}, \tag{112}$$

$$b' = b - \eta \sum_i^N \frac{\partial C_i(\omega, b)}{\partial b}. \tag{113}$$

It turns out that this algorithm is not efficient,especially for large training sets. That is why we introduce **stochastic gradient descent** (SGD).

For SGD, Instead of calculating the derivative of the cost function for all the training samples and update the parameters once, we first calculate a subset of the training set, which is called a **mini-batch**, and then update the parameters. Then we take another mini-batch, calculate the derivative of the cost function, update the parameters; change to another mini-batch, calculate the derivative of the cost function, update the parameters and so on. Once we have scanned through the training set once, one **epoch** is finished. For example, we will update ω using SGD as

$$\omega' = \omega - \eta \sum_{batch\ 1} \left.\frac{\partial C_i(\omega, b)}{\partial \omega}\right|_{\omega}, \tag{114}$$

$$\omega'' = \omega' - \eta \sum_{batch\ 2} \left.\frac{\partial C_i(\omega, b)}{\partial \omega}\right|_{\omega'}, \tag{115}$$

$$\omega''' = \omega'' - \eta \sum_{batch\ 2} \left.\frac{\partial C_i(\omega, b)}{\partial \omega}\right|_{\omega''}, \tag{116}$$

$$\cdots\cdots \tag{117}$$

SGD is more efficient than GD because it has updated the parameters several times for one iteration through the training set, even though the descent for each mini-batch may not be in the most optimal direction, it turns out that the general direction is correct (for most of the time). And after several such descents we reach to the optimal point quicker than GD.

4.4.2 Back propagation algorithm

Now the key task is to calculate $\partial C/\partial\omega$ and $\partial C/\partial b$. We introduce **back propagation** to help us quickly calculate the derivatives without performing a lot of extra calculations through the net.

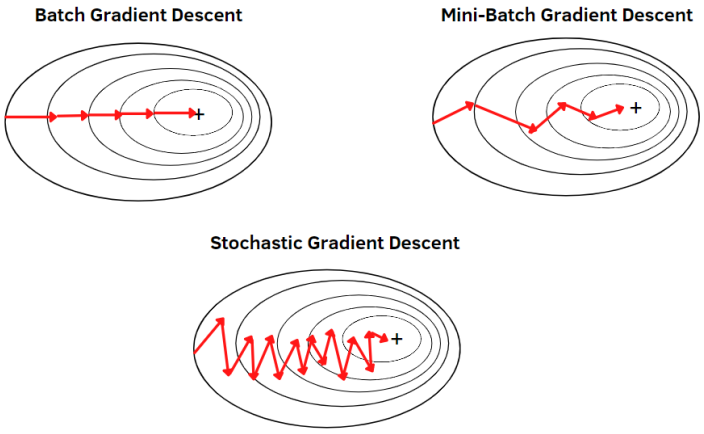


Figure 9: Different kinds of gradient descent.

The Hadamard product Hadamard product refers to the element-wise production of two tensors, and is denoted here as $*$, in agreement with the notation in Python. For example,

$$\begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} * \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} = \begin{pmatrix} 1 \\ 4 \\ 9 \end{pmatrix}. \tag{118}$$

Error We first define the **error** δ_j^l of the j-th neuron in l-th layer as

$$\delta_j^l \equiv \frac{\partial C}{\partial z_j^l}. \tag{119}$$

This quantity can be understood as how much the cost function will change once we change z_j^l a little bit, and yes, this quantity is the *error* in the term *error back propagation*. We could also work with, for example, $\partial C/\partial a_j^l$, but defining the error as Eq.(119) gives simpler mathematical formulation, so we will stick to this working flow.

With that being said, the back propagation is

$$\delta_j^L = \frac{\partial C}{\partial a_j^L}\sigma'(z_j^L) \qquad \delta^L = \frac{\partial C}{\partial a^L} * \sigma'(z^L)$$

$$\delta_j^l = \sum_i \omega_{ij}^{l+1}\delta_i^{l+1}\sigma'(z_j^l) \qquad \delta^l = (\omega^{l+1})^T\delta^{l+1} * \sigma'(z^l)$$

$$\frac{\partial C}{\partial b_j^l} = \delta_j^l \qquad \frac{\partial C}{\partial b^l} = \delta^l$$

$$\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1}\delta_j^l \qquad \frac{\partial C}{\partial w^l} = \delta^l(a^{l-1})^T$$

---

where the left column is the element-wise expression, and the right column is the tensor expression. Namely,

$$\delta^l = \begin{pmatrix} \delta_0^l \\ \delta_1^l \\ \vdots \end{pmatrix}, \quad a^L = \begin{pmatrix} a_0^L \\ a_1^L \\ \vdots \end{pmatrix}, \quad \frac{\partial C}{\partial a^l} = \begin{pmatrix} \frac{\partial C}{\partial a_0^l} \\ \frac{\partial C}{\partial a_1^l} \\ \vdots \end{pmatrix}, \quad \sigma'(z^l) = \begin{pmatrix} \sigma'(z_0^l) \\ \sigma'(z_1^l) \\ \vdots \end{pmatrix}, \tag{120}$$

$$\omega^l = \begin{pmatrix} \omega^l_{00} & \omega^l_{01} & \cdots & \omega^l_{0,n_{l-1}} \\ \omega^l_{10} & \omega^l_{11} & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots \\ \omega^l_{n_l,0} & \omega^l_{n_l,1} & \cdots & \omega^l_{n_l,n_{l-1}} \end{pmatrix}. \tag{121}$$

ω_l is of the size $(n_l+1, n_{l-1}+1)$. The labels of ω here is in accordance with the $\omega_{out,in}$ label. From the above formulae we can summarize the following steps for back propagation (All of these are better implemented in Python using the tensor version):

- **Forward propagation**: Feed the values forward through the net, and record the values of each neuron of this pass. The initial weights are random numbers.
- **Calculate output error**: The output error can be easily calculated because the form of the cost function is known. For example, for the quadratic cost function, the output error is simply $\delta^L = (a^L - t) * \sigma'(z^L)$, where t is the vector of target values.
- **Error back propagation**: Back propagate the error using $\delta^l = (\omega^{l+1})^T \delta^{l+1} * \sigma'(z^l)$.
- **Calculate the derivatives**: Calculate the derivatives using $\frac{\partial C}{\partial b^l} = \delta^l$ and $\frac{\partial C}{\partial w^l} = a^{l-1}(\delta^l)^T$.
- **Update the parameters**.
- **Next batch/epoch starts**...

The proof the these equations are well written in the book "Neural networks and deep learning", please refer there for detailed explanations.

Saturated neurons From the four equations for back propagation we see that when the derivative of the activation function is low, i.e. the activation value of the logistic function is close to either 1 (*high activation*) or 0 (*low activation*), the derivative of the cost function will be small, which means that the parameters will evolve, or *learn*, slowly. So high or low activation neurons are called saturated neurons.

As a result, there will be cases where we use other activation functions and cost functions to avoid this learning slow-down, as will be discussed below.

4.4.3 Activation versus weighted sum

Before we introduce cross-entropy, there are some comments to be made. A natural question is, why do we define $\partial C/\partial z$ as the error? Why not using $\partial C/\partial a$?

Well, it is actually equivalent to use either of them, but in the book they mentioned that defining the error using weighted sum is *mathematically simpler*, in a sense it gives shorter equations. But if we do some simple math and define a new error using activation

$$\zeta^l_i = \frac{\partial C}{\partial a^l_i}, \tag{122}$$

then we can easily prove

$$\delta^l_i = \zeta^l_i \, \sigma'(z^l_i), \tag{123}$$

then the back propagation algorithm can be thus rewritten as

$$\zeta^L = \frac{\partial C}{\partial a^L} \tag{124}$$

$$\zeta^l = (\omega^{l+1})^T \cdot \zeta^{l+1} * \sigma'(z^{l+1}) \tag{125}$$

$$\frac{\partial C}{\partial b^l} = \zeta^l * \sigma'(z^l) \tag{126}$$

$$\frac{\partial C}{\partial \omega^l} = \zeta^l * \sigma'(z^l) \cdot (a^{l-1})^T \tag{127}$$

Yeah, defining the error as $\partial C/\partial z$ does make the equations a bit more simpler...Anyways, there is no special reason for not using the activation in the definition of error other than simper equations, and we will see in the future that using ζ to represent back propagation actually makes it more clear in convolutional neural networks.

4.5 Cross-entropy (cost function)

A intuition for learning is that when the mismatch between the target value and the predicted value is large, the model should learn more quickly. For **quadratic cost function**

$$C = \frac{1}{2n}\sum_i || \ t - a^L \ ||^2, \tag{128}$$

where the summation is over all training samples indexed as i, this is indeed the case because

$$\frac{\partial C}{\partial a^L} = \frac{1}{n}\sum_i (a^L - t). \tag{129}$$

But in $\delta^L = \partial C/\partial a^L * \sigma'(z^L)$, there is another term $\sigma'(z^L)$ affecting δ^L as well, and tend to be very small when the predictions are saturated, i.e. $a^L \approx 0$ or 1. (We consider the target values to be within $[0, 1]$ here).

In order to remove this term, we carefully choose the cost function such that $\sigma'(z^L)$ will not appear when calculating δ^L, and that is **corss-entropy**.

$$C = -\frac{1}{n}\sum_i \left[t \ln a^L + (1-t)\ln(1-a^L)\right]. \tag{130}$$

This is a valid cost function because

- It is non-negative if the target values and predicted values are within the range $[0, 1]$.
- It is minimized when $a^L = t$.

Now we have

$$\frac{\partial C}{\partial a^L} = -\frac{1}{n}\sum_i \left(\frac{t}{a^L} - \frac{1-t}{1-a^L}\right) \tag{131}$$

$$= -\frac{1}{n}\sum_i \frac{t - t * a^L - a^L + a^L * t}{a^L * (1-a^L)} \tag{132}$$

$$= -\frac{1}{n}\sum_i \frac{t - a^L}{a^L * (1-a^L)}. \tag{133}$$

Using the property of the logistic function $\sigma'(z) = \sigma(z) * [1 - \sigma(z)]$, we have

$$\delta^L = \frac{\partial C}{\partial a^L} * \sigma'(z^L) \tag{134}$$

$$= -\frac{1}{n}\sum_i \frac{t - a^L}{a^L * (1-a^L)} * \sigma(z^L) * [1 - \sigma(z^L)] \tag{135}$$

$$= -\frac{1}{n}\sum_i \frac{t - a^L}{a^L * (1-a^L)} * a^L * (1-a^L) \tag{136}$$

$$= \frac{1}{n}\sum_i (a^L - t). \tag{137}$$

Using cross-entropy as the cost function, we manage to make the model *learn* faster when the mismatch is large. But be noted that, this only works when the activation function of the output neurons is the sigmoid function (and softmax, as we will see later). If we use a linear function as the activation for the output neurons, namely $a'(z^L) = const$, there is no need to use cross-entropy. Also note that cross-entropy only improves the situation at the output layer (sure, because we only change the form of the cost function, which only affects the output layer), for neurons living in hidden layers, the optimization still suffers from saturation problem.

4.6 Softmax (output neurons)

Softmax output neuron is another type of neuron using the softmax function as the activation function. For softmax output layer, we still have the weighted inputs $z_j^L = \sum_k \omega_{jk}^L a_k^{L-1} + b_j^L$, but the activation is now

$$a_j^L = \frac{e^{z_j^L}}{\sum_k e^{z_k^L}}. \tag{138}$$

It has the following properties:

- It is normalized. So whenever we can interpret the output as probability distribution, softmax comes in handy.
- It has the same derivative property as sigmoid function: $a'(z) = a(z)(1 - a(z))$, so the formalism of error back propagation using cross-entropy cost can be easily copied here.

$$\begin{aligned} a_j'^L &= \frac{\partial a_j^L}{\partial z_j^L} && (139)\\ &= \frac{e^{z_j^L} \sum_k e^{z_k^L} - (e^{z_j^L})^2}{(\sum_k e^{z_k^L})^2} && (140)\\ &= \frac{e^{z_j^L}}{\sum_k e^{z_k^L}} - (\frac{e^{z_j^L}}{\sum_k e^{z_k^L}})^2 && (141)\\ &= a_j^L - (a_j^L)^2 && (142)\\ &= a_j^L(1 - a_j^L). && (143) \end{aligned}$$

In summary, the biggest advantage of using softmax in the output layer is its probabilistic interpretation, and it can be combined with cross-entropy to solve the problem that the learning is slow when the output neurons are saturated.

Why is it called softmax? Consider a general case, and $c \to \infty$,

$$a(z_j) = \frac{e^{c \cdot z_j}}{\sum_k e^{c \cdot z_k}}. \tag{144}$$

So

$$\frac{a(z_j)}{a(z_k)} = e^{c \cdot (z_j - z_k)}. \tag{145}$$

Suppose z_j is the dominant term, namely z_j is positive and large and thus $a(z_j)$. All other terms will be infinitely small compared with $a(z_j)$ when $c \to \infty$. So here $a(z_j)$ selects the maximum output, giving a probability infinitely close to 1, all other minor terms are infinitely close to 0. By softmax, we mean that $a(z)$ still gives a probabilistic output based on the value of z_j, but it is *softer* in a sense that all other minor terms also have finite values.

4.7 Regularization

Here is the summary of different methods to reduce overfitting:

- Reduce the number of parameters in the model.
- Increase the size of the training set. (such as creating more images by slightly rotating the original images)
- Use regularization methods. (such as L1 regularization, L2 regularization, dropout, momentum gradient descent, etc.)

Let's talk about the third point, regularization. The regularization in neural networks are done pretty much the same way as Sec. 3.2 ridge regression. Namely, we add a regularization term to the original cost function C_0.

4.7.1 L2 regularization

For example, the most used regularization is **L2 regularization**,

$$C = C_0 + \frac{\lambda}{2n}\sum_{\omega}\omega^2, \tag{146}$$

where λ is known as the **regularization parameter**, and n is the size of the training set. By adding this term, we impose penalty on large weights, so that small weights are favored. (Large weights make the model less robust to small changes in inputs.)

Using L2 regularization, the derivative of the cost function is now

$$\frac{\partial C}{\partial \omega} = \frac{\partial C_0}{\partial \omega} + \frac{\lambda}{n}\omega, \tag{147}$$

$$\frac{\partial C}{\partial b} = \frac{\partial C_0}{\partial b}. \tag{148}$$

So the updating scheme for the weights and biases is simply

$$\omega' = \omega - \eta\frac{\partial C}{\partial \omega} = \omega - \eta\frac{\partial C_0}{\partial \omega} - \frac{\eta\lambda}{n}\omega = (1 - \frac{\eta\lambda}{n})\omega - \eta\frac{\partial C_0}{\partial \omega}, \tag{149}$$

$$b' = b - \eta\frac{\partial C_0}{\partial b}. \tag{150}$$

In this case, L2 regularization is also called **weight decay** due to the $1 - \eta\lambda/n$ coefficient. Note that the back propagation algorithm is not affected.

4.7.2 L1 regularization

The cost function using L1 regularization is written as

$$C = C_0 + \frac{\lambda}{n}\sum_{\omega}|\omega|. \tag{151}$$

The derivative of the regularized cost function is

$$\frac{\partial C}{\partial \omega} = \frac{\partial C_0}{\partial \omega} + \frac{\lambda}{n}\mathrm{sgn}(\omega), \tag{152}$$

where

$$\mathrm{sgn}(\omega) = \begin{cases} 1 & \text{if } \omega > 0 \\ 0 & \text{if } \omega = 0 \\ -1 & \text{if } \omega < 0 \end{cases}. \tag{153}$$

In this case the updating scheme for the weights is

$$\omega' = \omega - \frac{\eta\lambda}{n}\mathrm{sgn}(\omega) - \eta\frac{\partial C}{\partial \omega}. \tag{154}$$

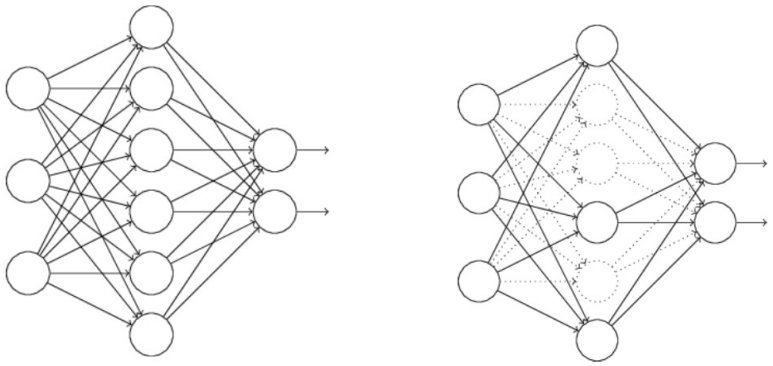
Compared with L2 regularization, the weights in L1 regularization shrink in a constant amount toward 0, while the weights in L2 regularization shrink proportionally to ω.

4.7.3 Dropout

The illustration of the dropout method is shown in Fig.10. The steps are:

- Temporarily deactivate half the hidden neurons in the network, as shown in Fig.10(b).
- Forward-propagate the inputs through the modified network.
- Back propagate the errors through the modified network.
- Update the parameters of the modified network.

- Repeat the above steps for a mini-batch.
- After a mini-batch, restore the network and select another random half of hidden neurons to deactivate.
- Do the forward prop, back prop and update for the newly modified network's parameters for another mini-batch.
- The loop goes on until all mini-batches are done.
- When we finish the training and restore the full network, we have to halve the weights outgoing from the hidden neurons because we use only half of them during the training.



(a) Before dropout. (b) Dropout.

Figure 10: Dropout method

Different from L2 or L1 regularization where we modify the cost function, for dropout method we modify the structure of the network itself. The regularization could be understood as:

- Averaging over several networks to reduce overfitting.
- or, by taking out some neurons, we expect our model to be robust to the loss of individual piece of evidence, so that it is more regularized.

Dropout has been very helpful in training large, deep neural networks.

4.8 Weights initialization

Consider the weighted inputs $z_j^l = \sum_k \omega_k^l a_k^{l-1} + b_j$. Previously we initialize the weights using Gaussians with mean 0 and variance 1, which means that z_j^l will have variance n_{in}, where n_{in} is the number of input weights associated with z_j^l. Recall that the variance of a sum of independent random variables is the sum of the variances of the individual random variables, and variance is the square of the standard deviation.

If the number of input weights is, say, 10, then the variance of z_j^l is 10, and the value of z_j^l could go easily large such that $\sigma(z_j^l)$ is close to 0 or 1, i.e. that neuron is saturated.

As can be seen from back propagation algorithm, saturated neurons will slow down the learning of the parameters. We can adress the learning slow-down problem of the *output layer* by alternating the cost functions, e.g. the cross-entropy cost function. However, that will not solve the learning slow-down for *hidden layers*.

So we should initialize the weights using Gaussians of mean 0 and variance $1/n_{\text{in}}$, or $1/(n_{\text{in}}+1)$ if you also want to consider the bias, so that the variance of z_j^l is still 1, and most neurons will not saturate.

4.9 The vanishing/exploding gradient problem

The vanishing/exploding gradient problem is one of the many reasons that *deep neural networks* are hard to train. To see why that comes, we recall the back propagation algorithm

$$\begin{aligned}
\delta^L &= \frac{\partial C}{\partial a^L} * \sigma'(z^L) \\
\delta^l &= (\omega^{l+1})^T \delta^{l+1} * \sigma'(z^l) \\
\frac{\partial C}{\partial b^l} &= \delta^l \\
\frac{\partial C}{\partial w^l} &= \delta^l (a^{l-1})^T
\end{aligned}$$

If we consider the standard gradient descent using sigmoid neurons, here is the derivative of the sigmoid function So $\sigma'(z)$ is always smaller than ~ 0.25. And since we initialize the weights with Gaussians

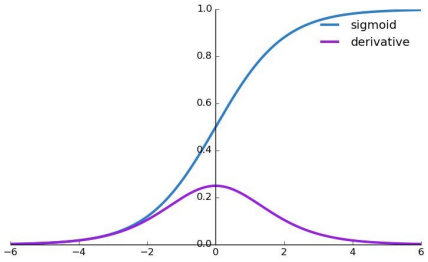


Figure 11: Derivative of the sigmoid function

of variance 1, so in general $|\omega| < 1$, which means that for each layer forward, δ^l will be at least 0.25 of the outer layer, which means that $\partial C/\partial \omega^l$ will be smaller. So in deep neural networks, the weights in the first several layers learn extremely slowly compared with outer layers. This is the **vanishing gradient problem**, as shown in Fig.(12).

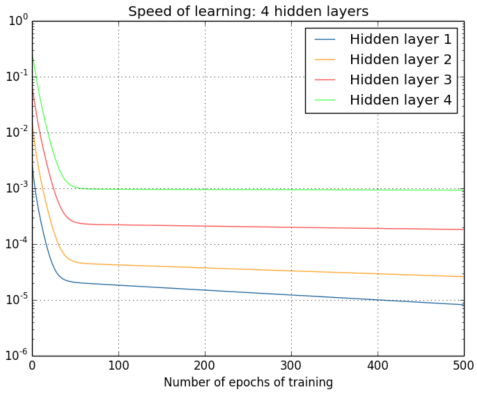


Figure 12: Demonstration of the vanishing gradient problem

On the other hand, if the weights are set by chance very large, e.g. $|\omega| > 100$, then as the error back propagates, it gets larger and larger, so do $\partial C/\partial \omega^l$. In this case we have the **exploding gradient problem**.

In summary, the gradients, i.e. the learning speed $\partial C/\partial \omega$, in deep neural networks tend to be very unstable, either vanishing or exploding gradient can be summarized as the **unstable gradient problem**. And we have ways to ameliorate this problem by altering some components in standard gradient descend, e.g. we can use ReLU activation instead of sigmoid activation.

4.10 Miscellaneous

4.10.1 Hessian optimization

Using Taylor's expansion, the cost function can be approximated near a point ω by

$$C(\omega+\Delta\omega) = C(\omega) + \sum_j \frac{\partial C}{\partial \omega_j}\Delta\omega_j + \frac{1}{2}\sum_{jk}\Delta\omega_j \frac{\partial^2 C}{\partial\omega_j\partial\omega_k}\Delta\omega_k + \cdots, \tag{155}$$

which could be written as

$$C(\omega+\Delta\omega) = C(\omega) + \nabla C^T\cdot\Delta\omega + \frac{1}{2}\Delta\omega^T H\Delta\omega + \cdots \tag{156}$$

where H is called the **Hessian matrix**

$$H_{jk} = \frac{\partial^2 C}{\partial\omega_j\partial\omega_k}. \tag{157}$$

If we omit the higher order terms of Eq.(156), and consider the terms to the second order

$$C(\omega+\Delta\omega) = C(\omega) + \nabla C^T\cdot\Delta\omega + \frac{1}{2}\Delta\omega^T H\Delta\omega, \tag{158}$$

then we let

$$\frac{\partial C(\omega+\Delta\omega)}{\partial\Delta\omega} = \nabla C^T + \frac{1}{2}\Delta\omega^T(H+H^T) \tag{159}$$

$$= \nabla C^T + \Delta\omega^T H = 0, \tag{160}$$

where we have used the symmetric property of H. So

$$\Delta\omega = -H^{-1}\nabla C. \tag{161}$$

By updating ω in this way, we are able to minimize the cost function as well (if H is positive-definite), and there are theories showing that Hessian optimization converges to the minimum in fewer steps than standard gradient descent. There are also back propagation algorithms used to calculate the Hessian.

However, one big drawback to calculating the Hessians comes from its size, because we need to square the number of weights are parameters!

4.10.2 Momentum gradient descent

For momentum gradient descent, the updating scheme for the parameters is

$$\omega' = \omega + v', \tag{162}$$

$$v' = \mu v - \eta\nabla C, \tag{163}$$

where μ is the hyper-parameter called the **momentum coefficient**, which controls the *velocity* or *momentum* of gradient descent (though it is not really the momentum in physics). To be more specific,

$$\begin{aligned}
v_0 &= 0 \\
v_1 &= \mu v_0 - \eta\nabla C|_{\omega_0} \\
&= -\eta\nabla C|_{\omega_0} \\
v_2 &= \mu v_1 - \eta\nabla C|_{\omega_1} \\
&= -\mu\eta\nabla C|_{\omega_0} - \eta\nabla C|_{\omega_1} \\
v_3 &= \mu v_2 - \eta\nabla C|_{\omega_2} \\
&= -\mu^2\eta\nabla C|_{\omega_0} - \mu\eta\nabla C|_{\omega_1} - \eta\nabla C|_{\omega_2}
\end{aligned}
\qquad
\begin{aligned}
\omega_0 &= \omega_0 \\
\omega_1 &= \omega_0 + v_1 \\
&= \omega_0 - \eta\nabla C|_{\omega_0} \\
\omega_2 &= \omega_1 + v_2 \\
&= \omega_0 - (1+\mu)\eta\nabla C|_{\omega_0} - \eta\nabla C|_{\omega_1} \\
\omega_3 &= \omega_2 + v_3 \\
&= \omega_0 - (1+\mu+\mu^2)\eta\nabla C|_{\omega_0} \\
&\quad - (1+\mu)\eta\nabla C|_{\omega_1}
\end{aligned}$$

$$
\begin{aligned}
v_4 &= \mu v_3 - \eta \nabla C|_{\omega_3} \\
&= -\mu^3 \eta \nabla C|_{\omega_0} - \mu^2 \eta \nabla C|_{\omega_1} - \mu \eta \nabla C|_{\omega_2} - \eta \nabla C|_{\omega_3} \\
&\cdots\cdots
\end{aligned}
\qquad
\begin{aligned}
\omega_4 &= \omega_3 + v_4 \\
&= \omega_0 - (1 + \mu + \mu^2 + \mu^3)\eta \nabla C|_{\omega_0} \\
&\quad - (1 + \mu + \mu^2)\eta \nabla C|_{\omega_1} \\
&\quad - (1 + \mu)\eta \nabla C|_{\omega_2} \\
&\quad - \eta \nabla C|_{\omega_3} \\
&\cdots\cdots
\end{aligned}
$$

If $\mu = 0$, we go back to the standard gradient descent scenario, and the descent will not build up, so we have *full* friction here. If $\mu = 1$, we see that for each step, the gradient of the previous step will add upon itself, so we have a strong *velocity* building up from previous steps, so $\mu = 1$ corresponds to no friction at all.

The advantage of using momentum method is that we exploit the information from previous points to help us descend quicker to the minimal point, as if we gain momentum/velocity when we gradient descend.

The drawback of this method is that, if the function space is changing quickly, we may easily overshoot the minimal point. Because for rapidly changing function space, the direction information of the gradient from previous points may be irrelevant to what we are at right now. In this case, we prefer a small value of μ.

The momentum method also has somewhat the effect of *learning rate scheduling*. Because as we descend down the hill, the learning rate of those initial steps are large, and that of the recent steps towards the minimum are small.

4.10.3 Other models of neuron/activation

tanh Tanh neuron replaces the sigmoid function by the hyperbolic tangent function, given by

$$\tanh(\omega \cdot x + b), \tag{164}$$

where

$$\tanh x = \frac{e^x - e^{-x}}{e^x + e^{-x}}. \tag{165}$$

Note that tanh function does not have the derivative property of sigmoid function, therefore the

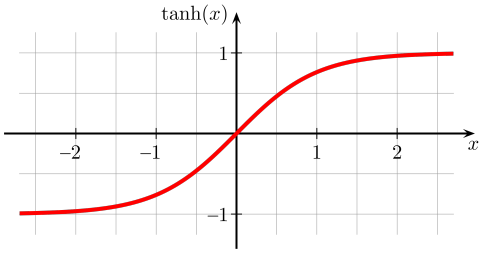


Figure 13: Tanh function

cross-entropy cost will not solve the learning slow-down problem for tanh neurons.

ReLU ReLU stands for **rectified linear unit**. It is defined as

$$\max(0, \omega \cdot x + b). \tag{166}$$

For ReLU neurons, there is no such problem as learning slow-down, because the derivative of this activation will be a constant regardless of the value of the weighted inputs, as long as they are positive. On the other hand, the learning completely stops if the weighted input is negative.

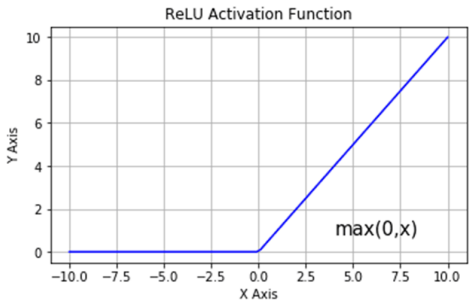


Figure 14: ReLU function

Which one is better? There is no universal conclusion on which activation function to use for which kind of problem. But there are some heuristics on which one performs (maybe slightly) better than others.

4.10.4 On optimizing the hyper-parameters

To get an NN that is learning non-trivial signal from a training set for the first time can sometimes be extremely challenging, if we don't know what are the optimal hyper-parameters. Below are some tips on the general strategy:

- **Fast experimentation is the key!** Use only a very small subset of the training data to do experimentation! If we are learning well, move on to the full data set.
- **Increase the frequency of monitoring!** Do not only look at the final overall accuracy, but plot the learning curve every, say, 5 epochs.
- **Using a small subset of validation data to check the accuracy**. Again, the smaller the data set, the faster the calculation. Avoid using test data to check the performance when tuning hypers, unless you do not have validation data.
- **Guess, trial, error, adjust**.
- **Grid search!** If hand-optimization gives you no clue, try doing grid search for the hypers using small data sets.
- In practice, there are relationships between the hypers. You may experiment with η, feel that you got it just right, then start to optimize for λ, only to find it is messing up your optimization for η. In practice, it helps to bounce backward and forward, gradually closing in good values.

Here are some tips on optimizing specific hypers:

- **Learning rate** η: To optimize learning rate we need to plot the learning curve (cost vs epochs) of the *training set*. 1. Give a random guess of the learning rate, then plot the learning curve of the training set. 2. If the learning curve is flat, reduce η. 3. If the learning curve is decreasing but has oscillations (overshooting at the minimal) when converging, reduce η. 4 The optimal η should give a smoothly converging learning curve.
- **Training epochs**: Use *early stopping* to determine the number of epochs. For example, if the classification accuracy of the validation data has no improvement for 10 consecutive epochs, terminate.
- **Regularization parameter** λ: Start with no regularization. Optimize η first. When you observe overfitting, the suggested value for the regularization parameter λ is such that $1 - \eta\lambda/n$ is of order $0.001 \sim 0.01$.

4.10.5 Others

Early stopping Once the accuracy on the validation data has saturated, e.g. no improvement in 10 epochs, we stop training.

skip-layer A two-layer (1 input, 1 hidden, 1 output) neural network with a skip-layer means the hidden layer will act as the skip-layer, and the information will go directly from the input layer to the output layer. Skip-layer can be understood as a linear transformation layer, such that successive linear transformations can be combined as one linear transformation, thus having the name "skip". Any sigmoidal hidden layer can mimic the behavior of skip-layer by using small weights. Because the inputs will be centered near 0, where the behavior of the sigmoid function is linear.

Neural networks are also called **universal approximators**.

> "A two-layer network with linear outputs can uniformly approximate any continuous function on a compact input domain to arbitrary accuracy provided the network has sufficiently large number of hidden units."

There is some confusion regarding the terminology to count the number of layers. A 3-layer network (1 input layer, 1 hidden layer and 1 output layer) can also be called a single-hidden-layer network. But we will call it a two-layer network, for the consideration that the number of layers of the adaptive weights are important for the neural network.

5 Convolutional neural networks

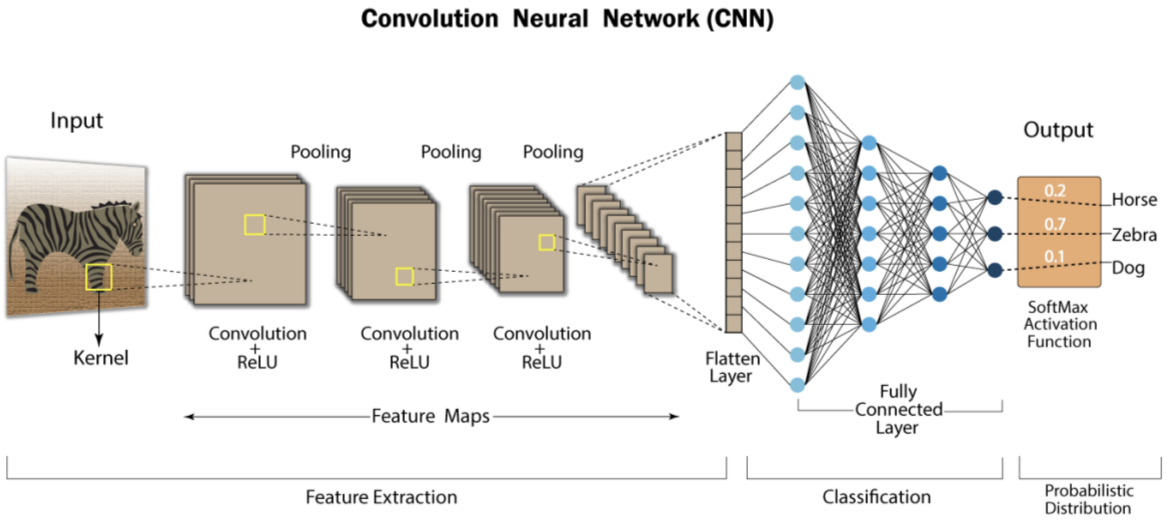


Figure 15: Structure of convolutional neural networks[2]

5.1 What is convolution?

5.2 Structure of CNN

The structure of CNN is very different from the fully connected neural networks introduced before, in a sense that CNN is more adapted to *two dimensional* images. For the following subsections, we should think of the input layer, the convolution layer and the pooling layer as *two dimensional* layer, and change our indices to two dimension as well.

5.2.1 Input layer

Different from fully connected neural networks where we switch the image into an one dimensional vector as the input vector, in CNN we keep the image as two dimensional as it was. As shown in Fig.(16).

5.2.2 Convolutional layer

The convolution layer is formed by convolving the **local receptive field** with the **filter/kernel**, and input to an activation function. The **local receptive field** is the local field (that 5×5 pixels in

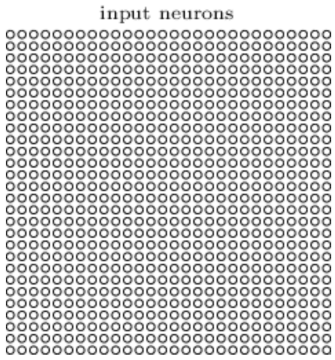


Figure 16: Input layer in CNN

Fig.(17)) from the input layer, the **kernel**, or **filter**, are the *weights* associated with each pixel in the local receptive field.

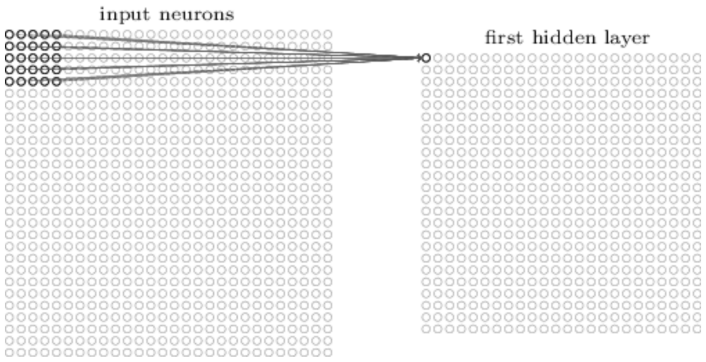


Figure 17: Convolution of the local receptive field with the kernel/filter

For each convolutional hidden layer, the weighted inputs are shown in Fig.(18). Note that we still apply activation functions in the convolutional hidden layer. The expression for the activation in the convolutional hidden layer is

$$a_{ij}^c = \sigma(z_{ij}^c) = \sigma\left(b^c + \sum_{m=0}\sum_{n=0} \omega_{m,n}^c x_{i+m,j+n}\right), \tag{167}$$

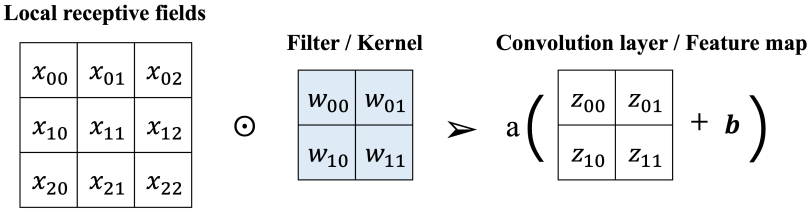
where the superscript c stands for convolution layers, z_{ij}^c is the i,j-th weighted input, m,n are indices for the local receptive field, note that the counting starts from 0. Note that for each feature map we only have 1 bias. Namely b is optimized along with the specific kernel we used. Note that here we take the default stride $= 1$.

In Fig.(17), each time we move 1 pixel to the next receptive field, but sometimes a different **stride length** is used, which is a hyper-parameter that could be optimized.

For each convolutional hidden layer, or **feature map**, the **weights are shared**. Namely for each receptive field going into the same feature map, we use the *same* kernel to convolve. For different feature maps in the same *batch* (as shown in Fig.(19), in practice we use different kernels to create many feature maps in one convolutional layer), we use *different* kernels, with the aim to learn different features from the input images. From this perspective, different kernels are like different feature extractors, and we will use them to determine if a certain feature is existent anywhere in the input image.

5.2.3 Padding[1]

One problem with standard convolution is that when we stride through the input image, the pixels at the corners and on the edges will naturally be sampled less than pixels near the center, e.g. in Fig.(20)



$$z_{00} = w_{00}x_{00} + w_{01}x_{01} + w_{10}x_{10} + w_{11}x_{11}$$
$$z_{01} = w_{00}x_{01} + w_{01}x_{02} + w_{10}x_{11} + w_{11}x_{12}$$
$$z_{10} = w_{00}x_{10} + w_{01}x_{11} + w_{10}x_{20} + w_{11}x_{21}$$
$$z_{11} = w_{00}x_{11} + w_{01}x_{12} + w_{10}x_{21} + w_{11}x_{22}$$

Figure 18: Mechanism of the convolutional hidden layer. a stands for activation function. b stands for biases of the convolutional hidden layer. Or you could think of b as another parameter of the kernel.

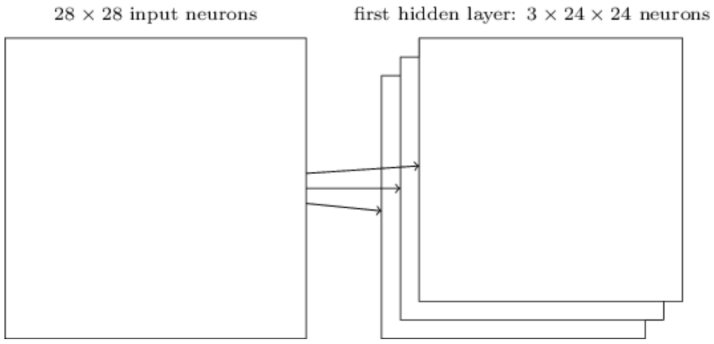


Figure 19: We create many feature maps in one convolutional hidden layer

the pixels at the corners will be sampled only once, but the center pixel will be sampled nine times.

This leads to the result that the extracted feature maps contain less information of pixels at the corners and edges, and could forbid us from constructing deeper networks. As a solution to this problem, we can manually add pixels of value 0 around the original input images to increase their sizes, so that pixels at corners and edges are sampled as frequently as center pixels. This is called **padding**.

Valid padding No padding at all.

Same padding Pad the input image such that the feature map is of the same size as the original input image. Yes, sometimes we want to keep the size.

5.2.4 Pooling layer

Pooling layers are used immediately after convolutional layers to further **condense the information from convolutional layers**. Basically we first select small regions from the convolutional layers, then *condense* the values in some ways, such as max pooling, average pooling, L2 pooling, etc. The size of the step that we move forward to the next selected region is also called **stride**, the default value is usually the size of the region, namely there is no overlap between those regions.

Max pooling In max pooling, we first select each, say, 2×2 region from the convolutional layer, and let the maximum value of those 4 neurons to be the value of the neuron in the pooling layer. Namely,

$$a^p_{ij} = \text{Max}_{m,n}(a^c_{i+m,j+n}), \tag{168}$$

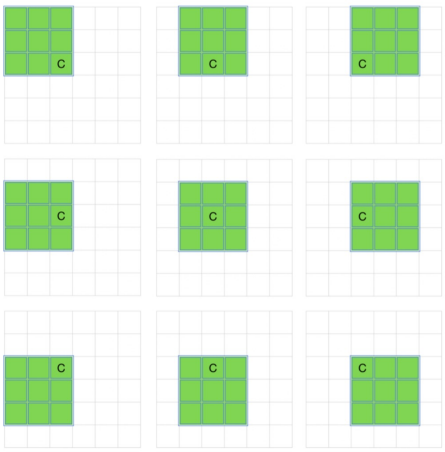


Figure 20: The corner pixels are sampled only once, while the center pixel is sampled nine times.

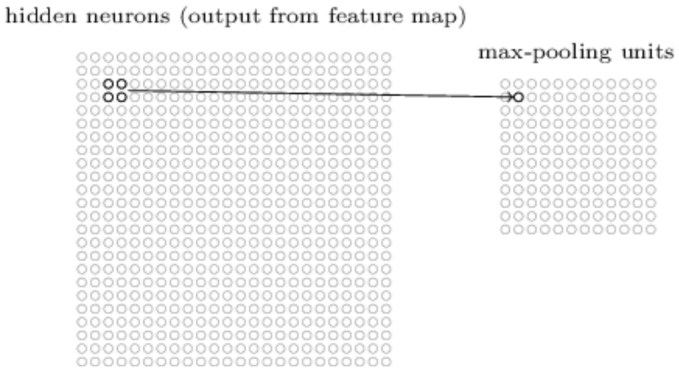


Figure 21: Max pooling

where a^p stands for the activation/output value of the neuron in the pooling layer, a^c stands for the activation/output value of the neuron in the convolutional layer, m and n scan through the selected region in the convolutional layer.

Average pooling As suggested by its name, the values of the neurons in the average pooling layers are taken to be the average of the neurons in the selected regions from the convolutional layers.

Namely,

$$a_{ij}^p = \frac{1}{N} \sum_{m=0} \sum_{n=0} a_{i+m,j+n}^c, \tag{169}$$

where N is the total number of neurons in the selected region.

L2 pooling The values of the neurons in the L2 pooling layers are taken to be the L2 norm of the neurons from the selected region in the convolutional layer.

Namely,

$$a_{ij}^p = \sqrt{\frac{\sum_{m=0} \sum_{n=0} (a_{i+m,j+n}^c)^2}{n}}, \tag{170}$$

where n is the total number of neurons in the selected region.

There are some resources saying that pooling preserves the symmetry invariance properties, find more about it and add them here.

5.2.5 Flatten layer

Flatten layer is used to turn the two dimensional pooling layer into a one dimensional vector, to be fully connected with the neurons of the next layer.

5.2.6 Fully connected layer

Here we go back to the classical fully connected neural networks. Usually by now the features of the image have been fully extracted and the size of layers have been greatly reduced, so that we can train fully connected layers without too much efforts.

5.2.7 LeNet and AlexNet

LeNet[3] was proposed by Yann LeCun et al. in 1998. It used back propagation to train the parameters, and outperformed all existing models back then on classifying hand written digits. LeNet is one of the very important landmarks on the way to better CNN models.

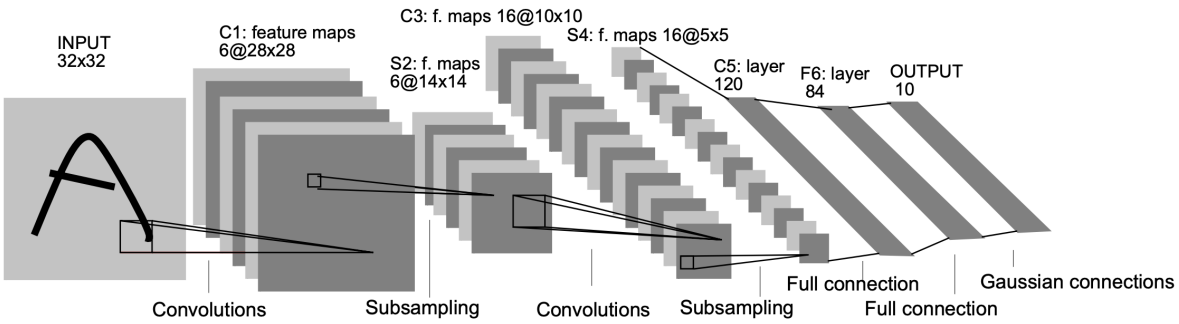


Figure 22: Structure of LeNet. C_x stands for convolutional layers, S_x stands for sub-sampling layers, or pooling layers.

Input layer LeNet has in total 7 layers (not counting the input layer). The input size is set to be 32×32, which is larger than the size of images from the training set. *Padding* is used to prevent loss of information at the edges and corners for images of size smaller than 32×32.

C_1 layer The first layer of convolution C_1 selects a receptive field of size 5×5, and a stride of 1, so the size of the feature map is $32 - 5 + 1 = 28$. A total of 6 feature maps are extracted, so the number of parameters in the first convolutional layer is $5 \times 5 \times 6 + 6 = 156$. The activation function is

S_2 layer The pooling layer S_2 selects non-overlapping neighbourhoods of size 2×2, and each layer is connected to the corresponding C_1 layer. Then the neurons are summed in the neighbourhood, multiplied with a trainable weight, plus a trainable bias, then passed through a sigmoid function. There are in total $6 \times 2 = 12$ trainable parameters in this layer.

C_3 layer The connection between C_3 and S_2 is a bit complicated. See Fig.(23). Each layer in C_3 are connected differently to S_2. The first six layers of C_3 are connected to contiguous three layers of S_2, and the receptive fields in each layer in S_2 have the same positions and sizes (5×5). The weights are different for each receptive field in each layer in S_2. Layer $6 - 11$ are connected to four contiguous layers in S_2, and the last four layers in C_3 are connected to S_2 in different ways. So the total number of parameters of C_3 is

$$5 \times 5 \times 3 \times 6 + 6 + 5 \times 5 \times 4 \times 9 + 9 + 5 \times 5 \times 6 \times 1 + 1 = 1516. \tag{171}$$

S_4 layer The pooling layer S_4 selects non-overlapping neighbourhoods of size 2×2, and each layer is connected to the corresponding C_3 layer. The values are determined in the same way as S_2. There are in total $16 \times 2 = 32$ trainable parameters in this layer.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	X				X	X	X			X	X	X	X		X	X
1	X	X				X	X	X			X	X	X	X		X
2	X	X	X				X	X	X			X		X	X	X
3		X	X	X			X	X	X	X			X		X	X
4			X	X	X			X	X	X	X		X	X		X
5				X	X	X			X	X	X	X		X	X	X

Figure 23: Connections of S_2 and C_3 in LeNet. The left column contains the indices of S_2, and the upper row contains the indices of C_3. X stands for the two layers are connected.

C_5 layer C_5 selects a 5×5 receptive field from ALL 16 layers (so it is a full connection) in S_4, and extracts in total 120 feature maps. Since S_4 itself is of size 5×5, so C_5 is one dimensional. There are in total $5 \times 5 \times 16 \times 120 + 120 = 48120$ trainable parameters in C_5.

F_6 layer It is a one dimensional network with 84 neurons fully connected with C_5. There are in total $120 \times 84 + 84 = 10164$ trainable parameters.

Output layer The output layer uses something special, I would suggest reading the original paper. There are just too many details to cover in this handbook...

LeNet works pretty well on MNIST, but when we have many outputs, say, 1000 outputs, then the training for the fully connected networks near the output gets very expensive. And we need alternative networks for such complex classification tasks.

ALEXNET!!

I will add more content about alexnet in the future...LeNet already took many more hours, which I did not expect.. have to go on for other chapters..

There are several key factors making AlexNet, a deeper and larger CNN than LeNet, possible:

- Dropout allows the design of much more deeper networks. It turns out that not only the regularization of the model parameters are important, the regularization of the network structure itself is also very important.
- ReLU was introduced to ameliorate the vanishing gradient problem, because you have half a space where the gradient is constant and stable.
- Max pooling makes the model invariant to small translations, because max pooling will still select the key feature even if you move it a little bit, unlike average pooling where the pooling layer is responsive to every change in the convolutional layer.

5.2.8 The advantage of using CNN

In the field of image recognition, there are training tasks where the size of the training set is of the order of millions of images. To have a fully connected neural network capable of learning such large number of images, a huge amount of parameter is required. Yet, we have seen before that deep neural networks are inherited with the problem of unstable gradient, making such training tasks formidable.

On the other hand, sharing weights and biases in CNN greatly reduces the number of parameters, and at the same time capturing the key properties to learn. For an image of 256×256 pixels, suppose the receptive field is of size 8×8 and we extract 20 feature map, then for CNN we need $8 \times 8 \times 20 + 20 = 1300$ parameters. In contrast, for a fully connected NN, even if we use a hidden layer of only 30 neurons, we have a daunting number of $256 \times 256 \times 30 + 30 = 1966110$ parameters!

5.3 Back propagation in CNN

The key to deriving back propagation algorithm is the *chain rule*, which is reflected in Fig.(24).

We have to know what is the starting point of the perturbation, e.g. z_i^l, what are the routes, e.g. all the neurons in the next layer $z_{\dots}^{l+1}$, and what is the end point, e.g. the cost function C. This is the

logic to write

$$\frac{\partial C}{\partial z_i^l} = \sum_j \frac{\partial C}{\partial z_j^{l+1}} \frac{\partial z_j^{l+1}}{\partial z_i^l}. \tag{172}$$

Layer by layer, we reach to the output layer, and put everything backward to get back propagation algorithm.

This is the very same logic that brings us back propagation in CNN. However, due to the fact that the activation function is not clearly defined for some layers in CNN, such as the pooling layer, it is better and clearer to use ζ, as defined in Sec.(4.4.3), to represent back propagation, in order to avoid messing up the notations. And only in this section, we call ζ the *error*. Keep in mind that activation value is the actual value passed out from a neuron.

5.3.1 Error in fully connected layers

Neurons in fully connected layers look like Fig.(24).

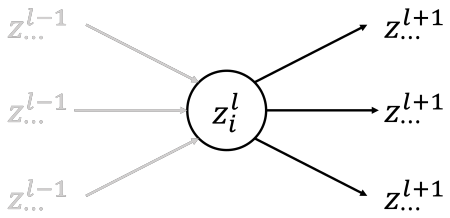


Figure 24: Neurons in fully connected layers. Note that using a^l, the activation, is equivalent to using z^l, since $a^l = \sigma(z^l)$.

So we have

$$\zeta_i^l = \frac{\partial C}{\partial a_i^l} \tag{173}$$

$$= \sum_j \frac{\partial C}{\partial a_j^{l+1}} \frac{\partial a_j^{l+1}}{\partial a_i^l} \tag{174}$$

$$= \sum_j \zeta_j^{l+1} \omega_{ji}^{l+1} \sigma'(z_j^{l+1}), \tag{175}$$

where we have used $a_j^{l+1} = \sigma(\sum_k \omega_{jk}^{l+1} a_k^l + b_j^{l+1})$. The above equation can be written in matrix form

$$\zeta^l = (\omega^{l+1})^T \zeta^{l+1} * \sigma'(z^{l+1}). \tag{176}$$

For the output layer, we have

$$\zeta^L = \frac{\partial C}{\partial a^L}. \tag{177}$$

From above we see that the back propagation for fully connected layers remain the same, which is of course expected.

5.3.2 Error in pooling layers

Pooling layer is connected to flatten layer, which is the first layer in fully connected layers. The neurons in flatten layers have same values as their counter-parts in the pooling layer, thus

$$\zeta_{ij}^p = \zeta_{mi+j}^f, \tag{178}$$

where i and j are indices of the pixel in the pooling layer, counted from 0. m is the number of columns in the pooling layer, namely the size of the pooling layer is supposed to be $m \times n$. The superscript p denotes the pooling layer, and f denotes the flatten layer.

5.3.3 Error in convolutional layers

The value of errors in the convolutional layer depends on how it is connected to the pooling layer.

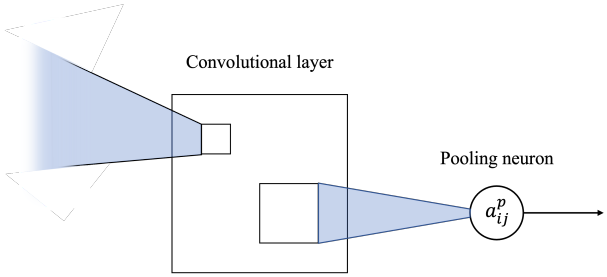


Figure 25: Connection between the convolutional layer and the pooling layer.

Max pooling For max pooling, only the maximum value in the selected neighbourhood is passed to the pooling layer

$$a_{ij}^p = \text{Max}_{m,n}(a_{i+m,j+n}^c), \tag{179}$$

where m and n scan through the selected neighbourhood in the convolutional layer.

As a result, making changes to minor values causes no change to the cost function, so we have

$$\zeta_{i+m,j+n}^c = \begin{cases} \zeta_{ij}^p & \text{if } i+m, j+n \text{ the indices for local maximum} \\ 0 & \text{if } i+m, j+n \text{ not the indices for local maximum} \end{cases}. \tag{180}$$

Average pooling In average pooling, we have

$$a_{ij}^p = \frac{1}{N} \sum_{m=0} \sum_{n=0} a_{i+m,j+n}^c, \tag{181}$$

where N is the total number of neurons in the selected region.

So we have

$$\zeta_{i+m,j+n}^c = \frac{\partial C}{\partial a_{i+m,j+n}^c} \tag{182}$$

$$= \frac{\partial C}{\partial a_{ij}^p} \frac{\partial a_{ij}^p}{\partial a_{i+m,j+n}^c} \tag{183}$$

$$= \frac{1}{N} \zeta_{ij}^p. \tag{184}$$

So for average pooling, the error in the pooling layer is equally distributed to all the neurons in the selected region.

5.3.4 Error propagation through the convolutional layer

The convolution operation can be represented as

$$a_{ij}^c = \sigma(z_{ij}^c) = \sigma\left(b^c + \sum_{m=0} \sum_{n=0} \omega_{m,n}^c x_{i+m,j+n}\right), \tag{185}$$

where the superscript c stands for convolution layers, z_{ij}^c is the i,j-th weighted input, m,n are indices for the local receptive field, note that the counting starts from 0. Note that for each feature map we only have 1 bias. Namely b is optimized along with the specific kernel we used. Note that here we take the default stride $= 1$.

The flow of message can be represented in Fig.(26).

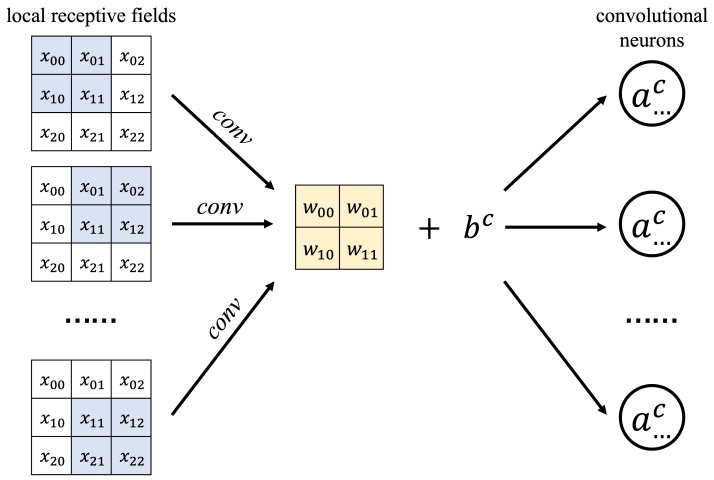


Figure 26: Message passing through the convolutional layer

To update the parameters $\omega^c_{m,n}$ and b^c, we have

$$\frac{\partial C}{\partial \omega^c_{m,n}} = \sum_i \sum_j \frac{\partial C}{\partial a^c_{ij}} \frac{\partial a^c_{ij}}{\partial \omega^c_{m,n}} \tag{186}$$

$$= \sum_i \sum_j \zeta^c_{ij} \frac{\partial a^c_{ij}}{\partial \omega^c_{m,n}} \tag{187}$$

$$= \sum_i \sum_j \zeta^c_{ij} \sigma'(z^c_{ij})\ x_{i+m,j+n} \tag{188}$$

$$= \sum_i \sum_j \delta^c_{ij}\ x_{i+m,j+n}, \tag{189}$$

which is nothing but

$$\frac{\partial C}{\partial \omega^c} = \text{conv}(X, \delta^c), \tag{190}$$

where X is the local receptive field, it is of the same size as the kernel, and its indices, which determine the local receptive field to use, depend on the indices m, n of $\omega^c_{m,n}$, as explicitly shown in Eq.(189). Here we use δ^c instead of ζ^c for convenience, but always keep in mind that these two are equivalent.

$$\frac{\partial C}{\partial b^c} = \sum_i \sum_j \frac{\partial C}{\partial a^c_{ij}} \frac{\partial a^c_{ij}}{\partial b^c} \tag{191}$$

$$= \sum_i \sum_j \frac{\partial C}{\partial a^c_{ij}}\ \sigma'(z^c_{ij}) \tag{192}$$

$$= \sum_i \sum_j \zeta^c_{ij} \sigma'(z^c_{ij}) \tag{193}$$

$$= \sum_i \sum_j \delta^c_{ij}, \tag{194}$$

which is nothing but

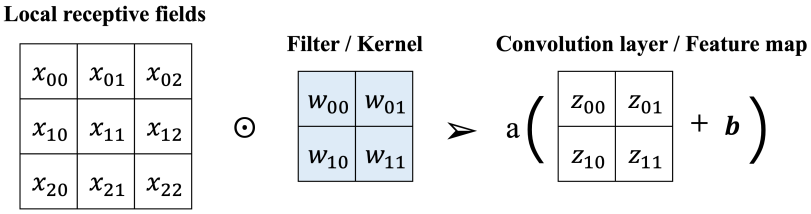
$$\frac{\partial C}{\partial b^c} = \text{sum}(\delta^c). \tag{195}$$

Yeah we successfully find $\partial C/\partial \omega$ and $\partial C/\partial b$ in CNN! But Can we call this an end?

5.3.5 Keep going backward!

Now, if our CNN is very simple and do the convolution only once, the above sub-section would have been the end of the story. But in practice, CNN has several convolutional layers, and the error has to keep going backward after passing through the convolutional layer for the first time! In this case, the input x_{ij} in Fig.(26) would be the activation value from the previous layer, namely from the last pooling layer. We will continue using x_{ij} here to represent the input/activation from the layer before the convolutional layer, feel free to remind yourself that x could the the activation a^p.

Now we have to calculate $\partial C/\partial x$ to propagate the error backward. We will use the simple example in Fig.(27) to reveal the equations.



$$
\begin{aligned}
z_{00} &= w_{00}x_{00} + w_{01}x_{01} + w_{10}x_{10} + w_{11}x_{11} \\
z_{01} &= w_{00}x_{01} + w_{01}x_{02} + w_{10}x_{11} + w_{11}x_{12} \\
z_{10} &= w_{00}x_{10} + w_{01}x_{11} + w_{10}x_{20} + w_{11}x_{21} \\
z_{11} &= w_{00}x_{11} + w_{01}x_{12} + w_{10}x_{21} + w_{11}x_{22}
\end{aligned}
$$

Figure 27: This figure is the same as Fig.(18).

$$
\begin{aligned}
\zeta_{00} &= \frac{\partial C}{\partial x_{00}} && (196)\\
&= \sum_i \sum_j \frac{\partial C}{\partial z^c_{ij}} \frac{\partial z^c_{ij}}{\partial x_{00}} && (197)\\
&= \delta^c_{00}\omega_{00}, && (198)
\end{aligned}
$$

where we have used z^c instead of a^c for convenience. Similarly we have

$$
\begin{aligned}
\zeta_{01} &= \delta^c_{00}\omega_{01} + \delta^c_{01}\omega_{00} && (199)\\
\zeta_{02} &= \delta^c_{01}\omega_{01} && (200)\\
\zeta_{10} &= \delta^c_{00}\omega_{10} + \delta^c_{10}\omega_{00} && (201)\\
\zeta_{11} &= \delta^c_{00}\omega_{11} + \delta^c_{01}\omega_{10} + \delta^c_{10}\omega_{01} + \delta^c_{11}\omega_{00} && (202)\\
\zeta_{12} &= \delta^c_{01}\omega_{11} + \delta^c_{11}\omega_{01} && (203)\\
\zeta_{20} &= \delta^c_{10}\omega_{10} && (204)\\
\zeta_{21} &= \delta^c_{10}\omega_{11} + \delta^c_{11}\omega_{10} && (205)\\
\zeta_{22} &= \delta^c_{11}\omega_{11} && (206)
\end{aligned}
$$

which turns out to be a *padded* δ^c convoluted with a 180° rotated kernel. *Note that a* 180° *rotation is not a transpose!* See Fig.(28). So we have

$$\zeta = \text{conv}(\ \text{Padded}(\delta^c),\ \text{Rotated}(K)\ ), \tag{207}$$

where K stands for the kernel.

5.3.6 Summary

We remind ourselves about the parameters to optimize:

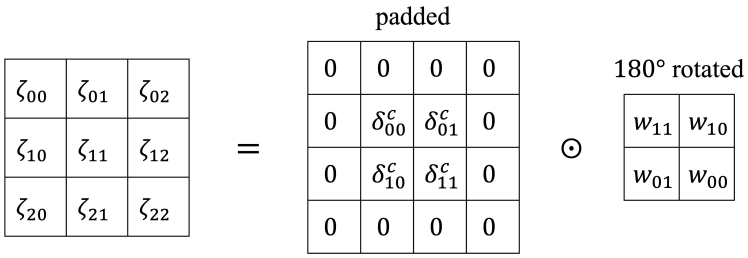


Figure 28: Error in the pooling layer before the convolutional layer

- Weights and biases in fully connected layers
- Weights (Kernel) and the bias for each feature map
- Any other trainable parameters related to specific structures of the CNN

Now let us go backward through the CNN!

$$\text{Output layer:} \quad \zeta^L = \frac{\partial C}{\partial a^L} \tag{208}$$

$$\text{Fully connected layers:} \quad \zeta^l = (\omega^{l+1})^T \zeta^{l+1} * \sigma'(z^{l+1}) \tag{209}$$

$$\underline{\text{Update weights:}} \quad \frac{\partial C}{\partial \omega^l} = \delta^l (a^{l-1})^T \tag{210}$$

$$\underline{\text{Update biases:}} \quad \frac{\partial C}{\partial b^l} = \delta^l \tag{211}$$

$$\text{Average Pooling layer:} \quad \zeta^p_{ij} = \zeta^f_{mi+j} \tag{212}$$

$$\text{Convolutional layer:} \quad \zeta^c_{i+m,j+n} = \frac{1}{N} \zeta^p_{ij} \tag{213}$$

$$\underline{\text{Update weights:}} \quad \frac{\partial C}{\partial \omega^c} = \text{conv}(X, \delta^c) \tag{214}$$

$$\underline{\text{Update bias:}} \quad \frac{\partial C}{\partial b^c} = \text{sum}(\delta^c) \tag{215}$$

$$\text{Pooling layer:} \quad \zeta = \text{conv}(\ \text{Padded}(\delta^c),\ \text{Rotated}(K)\ ) \tag{216}$$

$$\text{Convolutional layer:} \quad \cdots = \cdots \tag{217}$$

where δ can be easily obtained by $\delta = \zeta * \sigma'(z)$.

6 PyTorch

The content of this section is based on the tutorial https://www.learnpytorch.io/.

6.1 Fundamentals

```
import torch

print(torch.__version__)
```

```
# Scalar

scalar = torch.tensor(3)
print(scalar.ndim)
print(scalar.shape)

# Get the Python number within a tensor (only works with one-element tensors)
scalar.item()
```

```
# Vector

vector = torch.tensor([3,4])
print(vector.ndim)
print(vector.shape)
```

In the cotext of machine learning, we usually name scalars and vectors in lower cases, and matrices and tensors in upper cases.

```
# MATRIX

MATRIX = torch.tensor([[1,2],
                       [3,4]])
print(MATRIX.ndim)
print(MATRIX.shape)
```

```
# Tensor

TENSOR = torch.tensor([[[1, 2, 3],
                        [3, 6, 9],
                        [2, 4, 5]]])
print(TENSOR.ndim)
print(TENSOR.shape)
```

```
# Random tensor

rand_tensor = torch.rand(size=(3,4))
print(rand_tensor.ndim, rand_tensor.dtype)
```

```
# Zeros and ones

zeros = torch.zeros(size=(3,4))
ones = torch.ones(size=(3,4))
```

```
# Range

zero_to_ten = torch.arange(0,11)
zero_to_ten = torch.arange(start=0,end=11,step=1)
```

```
# Tensor-like

ten_zeros = torch.zeros_like(input=zero_to_ten)
```

```
# Default datatype for tensors is float32

float_32_tensor = torch.tensor([3.0, 6.0, 9.0],
dtype=None, # defaults to None, which is torch.float32 or whatever datatype is passed
device=None, # defaults to None, or 'cpu', 'cuda' etc.
requires_grad=False) # if True, operations performed on the tensor are recorded.
```

```
# Getting tensor attributes

print(tensor.dtype)
print(tensor.shape) # = tensor.size()
print(tensor.device)
```

```
# Change the dtype of a tensor

float_16_tensor = float_32_tensor.type(torch.float16)
```

```
# Tensor multiplication *

float_16_tensor * float_32_tensor
```

Note that the above line actually gives no error, but just in case, always keep the dtypes the same, since they might give an error. And the operation * is not a true matrix multiplication, **because * is done element-wise**.

Tensor operations There are in general five operations for PyTorch tensors:

- Addition
- Subtraction
- Multiplication, * (element-wise), sequence does not matter
- Division
- Matrix multiplication, @ (or dot product), sequence matters!

```
# Tensor operations

tensor = torch.rand(size=(3,4))

tensor + 10  # element-wise
torch.add(tensor, 10)  # element-wise

tensor * 10  # element-wise
torch.mul(tensor, 10)  # element-wise

tensor * tensor  # also element-wise
torch.mul(tensor, tensor)  # element-wise

tensor @ tensor  # Matrix multiplication
torch.matmul(tensor, tensor)  # Matrix multiplication
torch.mm(tensor, tensor)  # alias for tensor.matmul()
```

Note that before we perform a matrix multiplication, we must make sure the **inner dimensions** of those two matrices match, **`@` or `torch.matmul` will not automatically transpose the matrix for us. We have to transpose the matrix by ourselves**.

```
# Transpose a tensor

a = torch.rand(3,4)

b = torch.transpose(a,0,1)
b = torch.t(a)
b = a.t()
b = a.T
```

`torch.transpose(a,0,1)` is equivalent to `torch.transpose(a,1,0)`. The sequence of the latter two numbers does not matter, as they specify the two dimensions that will be switched.

`torch.t(a)` is equivalent to `a.t()`, the dimension of the input tensor in this case should be ≤ 2, because by default `torch.t()` is the short for `torch.transpose(input,0,1)`.

`a.T` gives the same output as `a.t()` for `tensor.ndim <=2`. It is just that `a.T` could be applied to tensors with arbitrary dimensions and reverse the dimensions.

The transpose of a vector is meaningless in PyTorch.

```
# Tensor aggregation (to its min, max, mean, sum, etc.)

a = torch.arange(0,100,10)

torch.min(a)
a.min()

torch.max(a)
a.max()

torch.mean(a.type(torch.float32))
a.type(torch.float32).mean()
```

```
torch.sum(a)
a.sum()
```

```
# Finding the positional min, max, etc.
# It returns the index of that position corresponding to that value of the tensor

torch.argmin(a)
a.argmin()

torch.argmax(a)
a.argmax()
```

```
# Reshape a tensor
# Note that the new shape should be compatible with the old shape

a = torch.arange(1,11)
a_reshaped = a.reshape(2,5)
```

```
# Change the view of a tensor

a_view = a.view(2,5)

# Note that a_view and a share exactly the same memory
# Any changes made to a_view will also reflect on a
```

```
# Stack tensors

a_stacked = torch.stack([a,a,a], dim=0)
```

```
# Squeeze a tensor
# The returned tensor is a VIEW of the original tensor

a_squeezed = torch.squeeze(a) # Remove all dims = 1
a_squeezed = a.squeeze() # Remove all dims = 1
a_squeezed = a.squeeze(dim=0) # Remove the first dimension if it is 1
a_squeezed = a.squeeze(dim=1) # Remove the second dimension if it is 1
```

```
# Unsqueeze a tensor
# The returned tensor is a VIEW of the original tensor

a_unsqueezed = a.unsqueeze(dim=0) # Add dim = 1 at the first dimension
a_unsqueezed = a.unsqueeze(dim=1) # Add dim = 1 at the second dimension
```

```
# Permute a tensor
# Returns a view of the original tensor with its dimensions permuted

a = torch.rand(size=(224,224,3))
a_permuted = a.permute(2,0,1) # a_permuted.shape = [3, 224, 224]
```

```
# Changing between PyTorch and NumPy

import torch
import numpy as np

array = np.arange(1,11)
tensor = torch.from_numpy(array)
# Warning: the default dtype in NumPy is float64
# While the default dtype in PyTorch is float32
# This would be better: tensor = torch.from_numpy(array).type(torch.float32)

tensor = torch.arange(1,11)
array = tensor.numpy()
# Warning: In this case the dtype of array would be float32

tensor_grad = torch.rand(2,2,requires_grad=Ture)
```

```
array_grad = tensor_grad.detach().numpy()
# Use Tensor.detach().numpy() to convert tensors with grad to numpy

# The above arrays and tensors DO NOT share the same memory
```

```
# Reproducible randomness
# Note that torch.manual_seed() only works for one cell

torch.manual_seed(1998)
a = torch.rand(3,4)

torch.manual_seed(1998)
b = torch.rand(3,4)

print(a == b) # Return a tensor of True
```

```
# Device agnostic code

device = 'cuda' if torch.cuda.is_available() else 'cpu'

a = torch.rand(3,4)
a_deviced = a.to(device)

a_to_numpy = a_deviced.numpy() # Will throw an error if on cuda
a_to_numpy = a_deviced.cpu().numpy() # Works!
# This is an example of device error
```

6.2 A simple linear model

Here we will demonstrate the workflow by fitting to a very simple linear model.

```
import torch
from torch import nn
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
```

```
# Prepare the dataset

weight = 0.7
bias = 0.3

X = torch.arange(0, 1, 0.02).unsqueeze(dim=1)
y = weight*X + bias

X_train, X_test, y_train, y_test = train_test_split(X, y, train_size=0.8)

print(X_train.shape, X_test.shape, y_train.shape, y_test.shape)
```

```
# Create a linear regression model
class LinearRegressionModel(nn.Module):
  def __init__(self):
    super().__init__()

    # Initialize model parameters
    self.weight = nn.Parameter(torch.randn(1, # or size=[1]
                                           dtype=torch.float32,
                                           requires_grad=True))
    self.bias = nn.Parameter(torch.randn(1, # size
                                         dtype=torch.float32,
                                         requires_grad=True))

  def forward(self, x: torch.Tensor) -> torch.Tensor:
    return self.weight*x + self.bias
```

The above model is a very simple linear regression model built using PyTorch. Here are several things to keep in mind:

- Every model built using PyTorch should inherit from the `nn.Module` class, which contains the building blocks for ML models.
- The initial parameters are random parameters. Here we are initializing w.r.t. the weight and bias we defined before, it could also be an entire layer, etc.
- The `nn.Parameter` class is a subclass of `torch.Tensor` that has a very special property when used with `nn.Module`: when they're assigned as `nn.Module` attributes, e.g. here `self.weight`, they are automatically added to the list of its parameters, and will appear e.g. in `parameters()` iterator, see the next code block.
- Any subclass of `nn.Module` needs to overwrite the `forward()` method, which defines the computation performed at every call.

And `x : torch.Tensor` indicates the datatype of `x`, `- > torch.Tensor` indicates the output datatype of this function.

```
# Equivalent and simpler way of defining a linear model

class LinearRegressionModel(nn.Module):
    def __init__(self):
        super().__init__()
        # Also called: linear transform, probing layer, fully connected layer, dense layer
                                                                ...
        self.linear_layer = nn.Linear(in_features=1,
                                      out_features=1)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        return self.linear_layer(x)
```

```
# Check the parameters of the model

model = LinearRegressionModel()

print(list(model.parameters())) # Without name
print(model.state_dict()) # With name
```

```
# Device agnostic codes
device = 'cuda' if torch.cuda.is_available() else 'cpu'

print(next(model.parameters()).device)

model.to(device)

print(next(model.parameters()).device)

# All the data should also be on the same device as our model
# Otherwise it raises an error
X_train = X_train.to(device)
X_test = X_test.to(device)
y_train = y_train.to(device)
y_test = y_test.to(device)
```

```
# Make predictions with the model

with torch.inference_mode():
    y_pred = model(X_test)
```

The reason why we use `torch.inference_mode()` here is that it will make our predictions faster, especially for a large test set, by ignoring the track of things such as gradient. It is suggested to use this when you are sure that your operation has nothing to do with autograd, e.g. model training.

```
# Set up the loss function and the optimizer

loss_fn = nn.L1Loss() # Here we use mean absolute error
optimizer = torch.optim.SGD(params=model.parameters(), # the parameters to optimize
                            lr=0.01) # lr is learning rate
```

Learning rate is one of the most important hyperparameters. **Parameter** refers to the adjusted parameters in the ML model, while **hyperparameter** refers to the parameters that are set by the ML engineers.

```
epochs = 200

# Track different values
epoch_count = []
loss_values = []
test_loss_values = []

for epoch in range(epochs):
  ## Training
  # Put the model in train mode, this is the default state
  model.train()
  # Forward propagation
  y_pred = model(X_train)
  # Calculate the loss
  loss = loss_fn(y_pred, y_train)
  # Clear the gradient values from previous steps
  optimizer.zero_grad()
  # Calculate the gradient of the loss function w.r.t. the parameters using backprop
  loss.backward()
  # Update the parameters
  optimizer.step()

  ## Testing
  # Evaluate the model at some points
  if epoch%10 == 0:
    # Turn off the settings not needed for evaluation/testing (dropout/batchnorm layers)
    model.eval()
    with torch.inference_mode(): # In outdated codes we may use torch.no_grad()
      # Forward propagation
      test_pred = model(X_test)
      # Calculate the test loss
      test_loss = loss_fn(test_pred, y_test)

    # Append values to list
    epoch_count.append(epoch)
    loss_values.append(loss.detach().numpy())
    test_loss_values.append(test_loss.detach().numpy())
    # Print informaiton
    print(f'Epoch: {epoch} | Train loss: {loss} | Test loss: {test_loss}')
    print(model.state_dict())
```

It is a good habit to call `model.train()` when you start the training part, and call `model.eval()` when you start the evaluation/testing part.

```
# Plot the learning curve

fig, ax = plt.subplots()

ax.plot(epoch_count, loss_values, label='Train loss')
ax.plot(epoch_count, test_loss_values, label='Test loss')
ax.set_title('Learning curve')
ax.set_xlabel('Epochs')
ax.set_ylabel('Loss')
ax.legend()

fig.tight_layout()
fig.show()
```

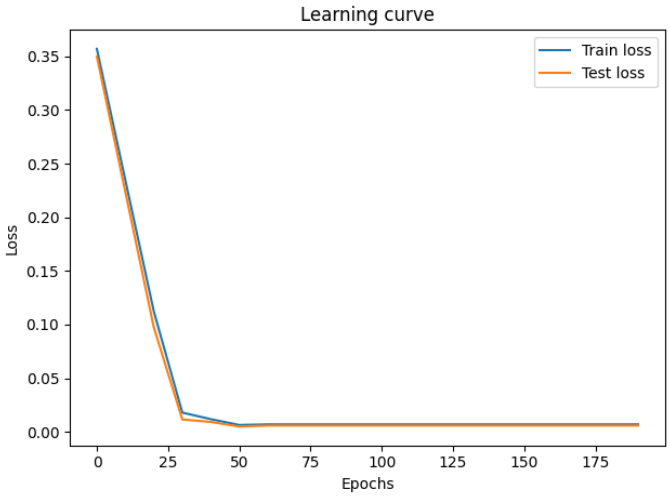


Figure 29: Learning curve

```
# Save the model
# Don't forget the file name (*.pt or *.pth) in PATH

PATH = '/content/drive/MyDrive/Colab_Notebooks/models/model_0.pt'
torch.save(obj=model.state_dict(), f=PATH)
```

```
# Load the model

new_model = LinearRegressionModel()
print(new_model.state_dict())

new_model.load_state_dict(torch.load(f=PATH))
print(new_model.state_dict())
```

```
# Make predictions with the new model
# Don't forget .eval() and .inference_mode()

new_model.eval()
with torch.inference_mode():
  new_preds = new_model(X_test)
```

6.3 Binary classification neural networks

```
# Load classification data
# Note that here data type is numpy.array

from sklearn.datasets import make_circles

n_samples = 1000

X, t = make_circles(n_samples,
                    noise=0.03,
                    random_state=42)

print(f'Input shape: {X.shape}\n')
print(f'Output shape: {t.shape}\n')
print(t[:5])
```

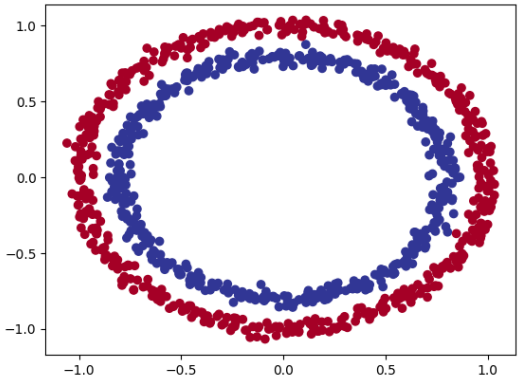
```
Input shape: (1000, 2)

Output shape: (1000,)

[1 1 1 1 0]
```

Hyperparameter	Binary Classification	Multiclass classification
Input layer shape (`in_features`)	Same as number of features (e.g. 5 for age, sex, height, weight, smoking status in heart disease prediction)	Same as binary classification
Hidden layer(s)	Problem specific, minimum = 1, maximum = unlimited	Same as binary classification
Neurons per hidden layer	Problem specific, generally 10 to 512	Same as binary classification
Output layer shape (`out_features`)	1 (one class or the other)	1 per class (e.g. 3 for food, person or dog photo)
Hidden layer activation	Usually ReLU (rectified linear unit) but can be many others	Same as binary classification
Output activation	Sigmoid (`torch.sigmoid` in PyTorch)	Softmax (`torch.softmax` in PyTorch)
Loss function	Binary crossentropy (`torch.nn.BCELoss` in PyTorch)	Cross entropy (`torch.nn.CrossEntropyLoss` in PyTorch)
Optimizer	SGD (stochastic gradient descent), Adam (see `torch.optim` for more options)	Same as binary classification

Figure 30: Architecture of classification neural networks

```
# Pandas comes in handy to deal with data

import pandas as pd

circles = pd.DataFrame({'x1': X[:,0],
                        'x2': X[:,1],
                        'labels': t})

circles.head(5)
```

```
# Visualize the data set

import matplotlib.pyplot as plt

fig, ax = plt.subplots()
ax.scatter(circles['x1'],
           circles['x2'],
           c=t,
           cmap=plt.cm.RdYlBu)
```




```
# Turn data into tensors

import torch

X = torch.from_numpy(X).type(torch.float32)
t = torch.from_numpy(t).type(torch.float32)
```

```
# Split the data set

from sklearn.model_selection import train_test_split

X_train, X_test, t_train, t_test = train_test_split(X,t,
                                                    test_size=0.2,
                                                    random_state=42)
```

```
# Build a NN model

from torch import nn

class ClassificationNN(nn.Module):
    def __init__(self):
        super().__init__()
        self.linear_1 = nn.Linear(in_features=2,
                                  out_features=8)
        self.linear_2 = nn.Linear(in_features=8,
                                  out_features=1)

    def forward(self,x):
        return self.linear_2(self.linear_1(x))

model = ClassificationNN()
```

```
# Equivalent way of creating such a model

model = nn.Sequential(nn.Linear(in_features=2, out_features=8),
                      nn.Linear(in_features=8, out_features=1))
```

```
# Equivalent way of creating such a model

class ClassificationNN_2(nn.Module):
    def __init__(self):
        super().__init__()
        self.two_linear_layers = nn.Sequential(nn.Linear(in_features=2, out_features=8),
                                               nn.Linear(in_features=8, out_features=1))

    def forward(self, x):
        return self.two_linear_layers(x)
```

- Note that here we just built a 2-layer neural network with 1 hidden layer, which has 8 hidden neurons. `self.linear_1` corresponds to the hidden layer, and `self.linear_2` corresponds to the output layer.

- Note that the number of input neurons should match the number of input features, and the number of output neurons should match the number of output features.
- Note that within the NN, the `in_features` of the hidden layer should match the `out_features` of the previous layer.
- Note that there is no activation function applied in any of these layers.
- The raw outputs of the above model are called **logits**. The logit of a probability p is the input value x to a sigmoid function which yields p.

$$p = \sigma(x) = \frac{1}{1 + e^{-x}} \tag{218}$$

$$\text{logit: } x = \sigma^{-1}(p) = \ln \frac{p}{1 - p} \tag{219}$$

  They will be converted to prediction probabilities later on by applying sigmoid activation (or *softmax* for multi-classification), then to prediction labels by rounding the prediction probabilities (or `argmax` for multi-classification).
- logits (raw output) -> prediction probabilities -> prediction labels

```
# Set up loss function and optimizer

loss_fn = nn.BCEWithLogitsLoss()

optimizer = torch.optim.SGD(params=model.parameters(),
                            lr=0.1)
```

The optimal cost function to use is problem-specific.

- For regression problems, you might want to use MSE or MAE.
- For classification problems, you might want to use cross-entropy.

The difference between the `BCEWithLogitsLoss` and `BCELoss` is that, the former one expects logits as its inputs, namely the NN output without the activation function, while the latter one expects probabilities as its inputs. So `nn.BCEWithLogitsLoss( y_logits, y_train )=nn.BCELoss( torch .sigmoid(y_logits), y_train )`

```
# Build the training loop

from sklearn.metrics import accuracy_score

torch.manual_seed(42)

epochs = 500

for epoch in range(epochs):
    model.train()

    # we use squeeze() here to unify the shapes
    y_logits = model(X_train).squeeze()
    # Note that the loss function is BCEWithLogitsLoss()
    loss = loss_fn(y_logits, t_train)
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    # Testing
    if epoch%10 == 0:
        model.eval()
        with torch.inference_mode():
            y_test_logits = model(X_test).squeeze()
            y_test_labels = torch.round(torch.sigmoid(y_test_logits))
            test_loss = loss_fn(y_test_logits, t_test)

            # Note the sequence of input values
```

```
            accuracy = accuracy_score(y_true=t_test,
                                      y_pred=y_test_labels)

        print(f'Epoch: {epoch} | Test loss: {test_loss} | Test accuracy: {100*accuracy}%')
```

```
...
Epoch: 230 | Test loss: 0.6946195363998413 | Test accuracy: 46.0%
Epoch: 240 | Test loss: 0.6946261525154114 | Test accuracy: 46.0%
...
Epoch: 460 | Test loss: 0.6946765184402466 | Test accuracy: 46.0%
Epoch: 470 | Test loss: 0.6946769952774048 | Test accuracy: 46.0%
Epoch: 480 | Test loss: 0.6946772933006287 | Test accuracy: 46.0%
Epoch: 490 | Test loss: 0.6946775913238525 | Test accuracy: 46.0%
```

Hmmm, why is that? Well, in the original tutorial, there is a very good visualization: https://www.learnpytorch.io/02_pytorch_classification/#5-improving-a-model-from-a-model-perspective. Deep down the reason is that the previous model we built has no non-linear activation functions (except for the output layer), so it is not possible for it to fit a circle!

So let us now start over to build a non-linear model!

```
class ClassificationNN_non_linear(nn.Module):
    def __init__(self):
        super().__init__()
        self.linear_1 = nn.Linear(in_features=2, out_features=8)
        self.linear_2 = nn.Linear(in_features=8, out_features=1)
        self.relu = nn.ReLU()

    def forward(self, x):
        return self.linear_2(self.relu(self.linear_1(x)))

model = ClassificationNN_non_linear()
```

```
loss_fn = nn.BCEWithLogitsLoss()
optimizer = torch.optim.SGD(params=model.parameters(),
                            lr=0.2)
```

```
from sklearn.metrics import accuracy_score


epochs = 500

for epoch in range(epochs):
    model.train()
    y_logits = model(X_train).squeeze()
    # Note that the loss function is BCEWithLogitsLoss()
    loss = loss_fn(y_logits, t_train)
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    if epoch%10 == 0:
        model.eval()
        with torch.inference_mode():
            y_test_logits = model(X_test).squeeze()
            y_test_labels = torch.round(torch.sigmoid(y_test_logits))
            test_loss = loss_fn(y_test_logits,t_test)
            test_accuracy = accuracy_score(y_true=t_test, y_pred=y_test_labels)
            print(f'Epoch: {epoch} | Test loss: {test_loss} | Test accuracy: {100*
                                                    test_accuracy}%')
```

```
Epoch: 0 | Test loss: 0.689878523349762 | Test accuracy: 53.0%
Epoch: 10 | Test loss: 0.6865444779396057 | Test accuracy: 54.0%
Epoch: 20 | Test loss: 0.685676634311676 | Test accuracy: 54.50000000000001%
Epoch: 30 | Test loss: 0.6850526332855225 | Test accuracy: 54.50000000000001%
```

```
Epoch: 40 | Test loss: 0.6844508647918701 | Test accuracy: 55.00000000000001%
Epoch: 50 | Test loss: 0.6838477253913879 | Test accuracy: 56.00000000000001%
Epoch: 60 | Test loss: 0.6832348704338074 | Test accuracy: 55.50000000000001%
Epoch: 70 | Test loss: 0.682598888874054 | Test accuracy: 55.00000000000001%
Epoch: 80 | Test loss: 0.6819336414337158 | Test accuracy: 56.00000000000001%
Epoch: 90 | Test loss: 0.6812419891357422 | Test accuracy: 56.49999999999999%
Epoch: 100 | Test loss: 0.6805312633514404 | Test accuracy: 55.50000000000001%
Epoch: 110 | Test loss: 0.6798207759857178 | Test accuracy: 55.50000000000001%
Epoch: 120 | Test loss: 0.6791056990623474 | Test accuracy: 55.50000000000001%
Epoch: 130 | Test loss: 0.6783709526062012 | Test accuracy: 56.00000000000001%
Epoch: 140 | Test loss: 0.6776062250137329 | Test accuracy: 55.50000000000001%
Epoch: 150 | Test loss: 0.6768156290054321 | Test accuracy: 55.50000000000001%
Epoch: 160 | Test loss: 0.6759982109069824 | Test accuracy: 55.50000000000001%
Epoch: 170 | Test loss: 0.6751616597175598 | Test accuracy: 56.49999999999999%
Epoch: 180 | Test loss: 0.6742920875549316 | Test accuracy: 58.5%
Epoch: 190 | Test loss: 0.6733829975128174 | Test accuracy: 59.5%
Epoch: 200 | Test loss: 0.6724340915679932 | Test accuracy: 61.5%
Epoch: 210 | Test loss: 0.6714396476745605 | Test accuracy: 64.5%
Epoch: 220 | Test loss: 0.6704716682434082 | Test accuracy: 65.0%
Epoch: 230 | Test loss: 0.6695387363433838 | Test accuracy: 66.5%
Epoch: 240 | Test loss: 0.668637216091156 | Test accuracy: 69.5%
...
Epoch: 460 | Test loss: 0.633186936378479 | Test accuracy: 84.5%
Epoch: 470 | Test loss: 0.6308820247650146 | Test accuracy: 85.0%
Epoch: 480 | Test loss: 0.6284880638122559 | Test accuracy: 87.5%
Epoch: 490 | Test loss: 0.626177191734314 | Test accuracy: 90.0%
```

It works!

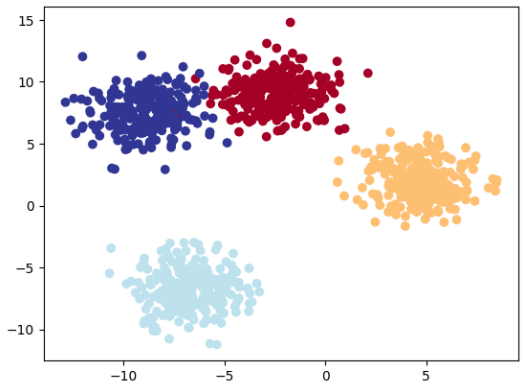
What if we use `y_labels` to calculate the training loss (in this case we use `nn.BCELoss()`)? Well, in that case, the changing of weights may not lead to the change of the outputs (because the outputs are rounded), so in this case the learning completely stops.

6.4 Multi-class classification neural networks

```
import torch
from torch import nn
import matplotlib.pyplot as plt
from sklearn.datasets import make_blobs
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
```

```
# Create the multi-class data

X_blob, t_blob = make_blobs(n_samples=1000,
                            n_features=2,
                            centers=4,
                            cluster_std=1.5,
                            random_state=42)

# Turn data into tensors

X_blob = torch.from_numpy(X_blob).type(torch.float32)

# Well, when calculating cross entropy loss, the dtype should be torch.long
t_blob = torch.from_numpy(t_blob).type(torch.long)


# Split into train and test

X_train, X_test, t_train, t_test = train_test_split(X_blob, t_blob, test_size=0.2,
                                                    random_state = 42)
```

```
# Plot data

fig, ax = plt.subplots()
ax.scatter(X_blob[:,0], X_blob[:,1], c=t_blob, cmap=plt.cm.RdYlBu)
```




```
# Build a multi-class classification model

class MultiClassification(nn.Module):
    def __init__(self, input_features, output_features, hidden_units=8):
        super().__init__()
        self.linear_layer_stack = nn.Sequential(
            nn.Linear(in_features=input_features, out_features=hidden_units),
            nn.ReLU(),
            nn.Linear(in_features=hidden_units, out_features=hidden_units),
            nn.ReLU(),
            nn.Linear(in_features=hidden_units, out_features=output_features)
        )

    def forward(self, x):
        return self.linear_layer_stack(x)



model = MultiClassification(input_features=X_blob.shape[1],
                            output_features=len(torch.unique(t_blob)),
                            hidden_units=8)
```

```
# Loss function and optimizer
# Note the difference from the binary classification problem
# We use cross entropy instead of binary cross entropy (BCE)

# Note that this loss function takes in logits for predictions
loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(params=model.parameters(),
                            lr=0.1)
```

```
# Evaluate the model
# Note the difference from the binary classification problem
# We use softmax instead of sigmoid in the output layer

model.eval()
with torch.inference_mode():
    y_logits = model(X_test)
    y_probs = torch.softmax(y_logits, dim=1)
    y_labels = torch.argmax(y_probs, dim=1)
    print(y_probs[:5])
    print(y_labels[:5])
```

```
tensor([[0.3169, 0.3244, 0.1405, 0.2182],
```

```
        [0.3336, 0.1432, 0.2026, 0.3206],
        [0.3011, 0.1843, 0.2823, 0.2323],
        [0.3078, 0.2766, 0.1836, 0.2320],
        [0.3719, 0.1286, 0.1532, 0.3463]])
tensor([1, 0, 0, 0, 0])
```

```python
torch.manual_seed(42)

epochs = 100

for epoch in range(epochs):
    model.train()

    y_logits = model(X_train)
    # Do not forget dim=1
    y_probs = torch.softmax(y_logits, dim=1)
    y_labels = torch.argmax(y_probs, dim=1)

    loss = loss_fn(y_logits, t_train)

    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    if epoch%10 == 0:
        model.eval()
        with torch.inference_mode():
            y_test_logits = model(X_test)
            y_test_probs = torch.softmax(y_test_logits, dim=1)
            y_test_labels = torch.argmax(y_test_probs, dim=1)

            test_loss = loss_fn(y_test_logits, t_test)
            accuracy = accuracy_score(y_true=t_test,
                                      y_pred=y_test_labels)

            print(f"Epoch: {epoch} | Loss: {loss:.4f} | Test loss: {test_loss:4f} | Test
                                                    accuracy: {accuracy}")
```

```
Epoch: 0 | Loss: 1.1588 | Test loss: 1.075542 | Test accuracy: 0.48
Epoch: 10 | Loss: 0.6448 | Test loss: 0.660687 | Test accuracy: 0.975
Epoch: 20 | Loss: 0.4254 | Test loss: 0.430741 | Test accuracy: 1.0
Epoch: 30 | Loss: 0.2529 | Test loss: 0.245076 | Test accuracy: 0.995
Epoch: 40 | Loss: 0.1123 | Test loss: 0.102285 | Test accuracy: 0.995
Epoch: 50 | Loss: 0.0663 | Test loss: 0.058475 | Test accuracy: 0.995
Epoch: 60 | Loss: 0.0507 | Test loss: 0.042932 | Test accuracy: 0.995
Epoch: 70 | Loss: 0.0430 | Test loss: 0.034910 | Test accuracy: 0.995
Epoch: 80 | Loss: 0.0384 | Test loss: 0.029878 | Test accuracy: 0.995
Epoch: 90 | Loss: 0.0352 | Test loss: 0.026627 | Test accuracy: 0.995
```

```python
# Make predictions with multi-class classification model

model.eval()
with torch.inference_mode():
    y_logits = model(X_test)
    y_probs = torch.softmax(y_logits, dim=1)
    y_labels = torch.argmax(y_probs, dim=1)
    print(y_labels[:5])
    print(t_test[:5])
```

```
tensor([1, 3, 2, 1, 0])
tensor([1, 3, 2, 1, 0])
```

For this specific classification problem, the model still performs well even if we comment out `nn.ReLU()`, namely remove the non-linearity. The reason is that the data we created are linearly separable, and with four output neurons, the model "draws" four different straight lines as the decision boundary. But for the purpose of generality, it is still better to always add non-linearity.

6.5 Gradients in PyTorch [IMPORTANT]

We will figure out the following things in this section:

- `requires_grad`
- `x.grad`, i.e. `torch.Tensor.grad`
- `x.grad_fn`, i.e. `torch.Tensor.grad_fn`
- leaf tensor

`requires_grad` This attribute tells PyTorch whether or not to track the gradient of the tensor, i.e. any mathematical operation on this tensor, so that the gradient of an output w.r.t. this tensor can be easily calculated using back propagation. See this example.

`x.grad` Returns the gradient of a variable on which `.backward()` is applied w.r.t. `x`. If we did not call `.backward()`, `x.grad` is `None`. `x.grad` only works for tensors that are **leaf tensors**. For a non-leaf tensor `y`, `y.grad` only makes sense after we call `y.retain_grad()`, otherwise it gives a warning. Future calls of `.backward()` will accumulate/add gradients on `x.grad`. See here for the original documentation.

leaf tensor All tensors with `requires_grad=False` are leaf tensors. User-created tensors with `requires_grad=True` are leaf tensors. For a leaf tensor `x`, `x.grad` will only have actual values if `x.requires_grad=True`. Otherwise it is `None`. The `.grad` attribute of all leaf tensors will be populated after we call `.backward()`. Note that the value `None` is also considered as *populated*.

`x.grad_fn` This attribute records from which function is the tensor created, under the condition that `requires_grad=True`. This attribute facilitates the process of back propagation because it remembers the derivative of the functions in the previous steps in the chain rule, as demonstrated later in the example. Apparently, for leaf tensors, we always have `x.grad_fn=None`, because leaf tensors either are created by the user with no previous PyTorch operation, or have `requires_grad=False` which do not let PyTorch track the operations.

With the above introduced, we here show a easy example explaining those in detail and show how PyTorch uses back propagation to calculate the gradients.

```
a = torch.rand(2,2,requires_grad=True)
print(a.grad_fn) # None
print(a.is_leaf) # True
print(a.grad) # None
```

```
b = a + 2 # tensor([[2.1163, 2.5686],[2.6716, 2.1990]], grad_fn=<AddBackward0>)
print(b.is_leaf) # False
print(b.grad_fn) # <AddBackward0 object at 0x7fba0cf1d310>
```

```
c = b*b*2
out = c.mean()
print(out.grad_fn) # <MeanBackward0 object at 0x7fba03a65880>
out.backward() # IMPORTANT!
print(a.grad) # tensor([[2.1163, 2.5686],[2.6716, 2.1990]])
```

The back propagation works as follows, let's take the derivative w.r.t. a_{11} for example

$$\frac{d\,(out)}{d\,a_{11}} = \frac{d\,(c_{11}+c_{12}+c_{21}+c_{22})/4}{d\,a_{11}} \tag{220}$$

$$= \frac{1}{4}\left(\frac{d\,c_{11}}{d\,a_{11}} + \frac{d\,c_{12}}{d\,a_{11}} + \frac{d\,c_{21}}{d\,a_{11}} + \frac{d\,c_{22}}{d\,a_{11}}\right) \tag{221}$$

$$= \frac{1}{4}\left(\frac{\partial\,c_{11}}{\partial\,b_{11}}\frac{\partial\,b_{11}}{\partial\,a_{11}} + \frac{\partial\,c_{11}}{\partial\,b_{12}}\frac{\partial\,b_{12}}{\partial\,a_{11}} + \cdots\right) \tag{222}$$

$$\frac{\partial\, c_{12}}{\partial\, b_{11}}\frac{\partial\, b_{11}}{\partial\, a_{11}} + \frac{\partial\, c_{12}}{\partial\, b_{12}}\frac{\partial\, b_{12}}{\partial\, a_{11}} + \cdots \tag{223}$$

$$\frac{\partial\, c_{21}}{\partial\, b_{11}}\frac{\partial\, b_{11}}{\partial\, a_{11}} + \frac{\partial\, c_{21}}{\partial\, b_{12}}\frac{\partial\, b_{12}}{\partial\, a_{11}} + \cdots \tag{224}$$

$$\frac{\partial\, c_{22}}{\partial\, b_{11}}\frac{\partial\, b_{11}}{\partial\, a_{11}} + \frac{\partial\, c_{22}}{\partial\, b_{12}}\frac{\partial\, b_{12}}{\partial\, a_{11}} + \cdots) \tag{225}$$

$$= \frac{1}{4}\frac{\partial\, c_{11}}{\partial\, b_{11}} \tag{226}$$

$$= b_{11}, \tag{227}$$

which agrees with the output of our codes above. Note that during this process, the operation that `.grad_fn` records facilitates the calculations of all the partial derivatives, e.g. $\partial c_{11}/\partial b_{11} = 4b_{11}$, $\partial b_{11}/\partial a_{11} = 1$, etc. It is because the complex mathematical operations consist of simple operations such as Add, Mean, Sum, Power, etc, that we are able to decompose them step by step using chain rule and calculate the gradients of complex functions such as neural network.

Now let's make some variations to the above code and see what happens.

- call `b.grad` → `None` and reports a warning. Because `b` is originally a non-leaf tensor, we have to call `b.retain_grad()` before calling `out.backward()`. The result is as expected, $\partial(out)/\partial b_{ij} = b_{ij}$.
- change `a.requires_grad=False` → `out.backward()` reports a bug: element 0 of tensors does not require grad and does not have a grad_fn.

When I ran the **third code block** above again and again I found something interesting, that instead of giving the correct results as executed for the first time, `a.grad` accumulates the gradient values each time. This has something to do with the internal implementation of PyTorch, and that is why we have to call `optimizer.zero_grad()` function in each epoch of the neural network.

6.6 Modes in PyTorch

- model.train()
- model.eval()
- with torch.no_grad()
- with torch.inference_mode()

`with torch.inference_mode()` Here is a simple example:

```
import torch
from torch import nn

# Define a simple input
X = torch.arange(0,1,0.2).unsqueeze(dim=1)

# Define a simplt linear model
model = nn.Sequential(nn.Linear(in_features=1,out_features=1))

# Pred without inference mode
pred = model(X)
print(pred,'\n')

# Pred with inference model
with torch.inference_mode():
    pred = model(X)
    print(pred)
```

The output is of the above block is (Note the difference at `grad_fn`):

```
tensor([[0.2927],
        [0.4554],
```

```
        [0.6181],
        [0.7808],
        [0.9435]], grad_fn=<AddmmBackward0>)

tensor([[0.2927],
        [0.4554],
        [0.6181],
        [0.7808],
        [0.9435]])
```

There is something going on with requires_grad in nn.Parameter() and torch.randn() and torch.inference_mode() and model.train() and model.eval() and torch.no_grad().

6.7 Miscellaneous

Loss function/Optimizer	Problem type	PyTorch Code
Stochastic Gradient Descent (SGD) optimizer	Classification, regression, many others.	`torch.optim.SGD()`
Adam Optimizer	Classification, regression, many others.	`torch.optim.Adam()`
Binary cross entropy loss	Binary classification	`torch.nn.BCELossWithLogits` or `torch.nn.BCELoss`
Cross entropy loss	Mutli-class classification	`torch.nn.CrossEntropyLoss`
Mean absolute error (MAE) or L1 Loss	Regression	`torch.nn.L1Loss`
Mean squared error (MSE) or L2 Loss	Regression	`torch.nn.MSELoss`

Figure 31: Choosing the loss function and the optimizer

7 Machine Learning Practical Tools

7.1 Use `Pandas` to load the datasets

The basic sentences are

```
import pandas as pd

df = pd.read_csv('path/to/file', header = None)
```

and

```
import pandas as pd

dic = {'col_1':[1,2,3],
       'col_2':{4,5,6}}
df = pd.DataFrame(dic)
```

Note that file `df` here belongs to the class `pandas.DataFrame`. It is a special data format used in pandas. Each column of `DataFrame` belongs to the class `pandas.Series`, which is another one-dimensional data format used in pandas.

Here are some basic commands to show the information of `DataFrame`.

`df.columns` It shows the names/labels of all the columns.

`df.values` It returns a numpy array version of `DataFrame`.

`df.info` It shows the information and content of this `DataFrame`.

`df.head(n)` It shows the first `n` rows of this `DataFrame`.

`df['col_name']` It returns the indicated column as `pandas.Series`.

`df.values[row_id, column_id]` **or** `df['col_name'][row_id]` It returns the indicated element in `DataFrame`.

`df > 10` Doing this will return a `DataFrame` of the same shape where all the elements are booleans. This operation is often used in `df.where()` and `df.mask()`.

7.1.1 pandas.where()

The frequently used format is

```
df.where(cond = df < 90, other = 'A+')
```

Doing this will replace all the elements `if not cond` with `other`. This webpage gives the best examples: Pandas where().

7.1.2 pandas.mask()

The frequently used format is

```
df.mask(cond = df >= 90, other = 'A+')
```

Doing this will replace all the elements `if cond` with `other`. This operation is the opposite of `where()`.

7.1.3 pandas.repalce()

The frequently used format is

```
df.repalce(a,b)
```

and

```
map = {'a': 'b'}
df.repalce(map)
```

Basically, this operation replaces one element, which must be present in `df`, with another user-specified element.

7.1.4 Example

So a basic example of using `pandas` to load the dataset and create the feature matrix and target vector is as follows:

```
import pandas as pd

path = './datasets/iris/iris.data'

df = pd.read_csv(path, header=None)

features = df.drop([4],axis=1)

map = {'Iris-setosa':0, 'Iris-versicolor':1, 'Iris-virginica':2}
```

```
b = df.replace(map)

targets = b[4]
```

8 Miscellaneous

8.1 Python

Dictionary There are two commonly used methods to define a dictionary, one is using the curly braces:

```
MLB_team = {
     'Colorado' : 'Rockies',
     'Boston'   : 'Red Sox',
     'Minnesota': 'Twins',
     'Milwaukee': 'Brewers',
     'Seattle'  : 'Mariners'}
```

Another one is using `dict` method:

```
MLB_team = dict(
     Colorado='Rockies',
     Boston='Red Sox',
     Minnesota='Twins',
     Milwaukee='Brewers',
     Seattle='Mariners')
```

Noted that in the first method, the string-type key names have to be inside the quotation mark, yet for `dict` method, the string-type key names can be directly defined.

Tuple Tuples are immutable! You can not change it!

Set A set can be created in the following two ways:

```
    set_A = set('a','b','c')
    set_B = {'a','b','c'}
```

Note that a set is a collection which is unordered, unchangeable, and unindexed, and it does not allow duplicate terms, which means that set can not have two terms having the same value.

Dunder methods Dunder or magic methods in Python are the methods having two prefix and suffix underscores in the method name. Dunder here means "Double Under (Underscores)". These are commonly used for operator overloading. Few examples for magic methods are: `__init__`, `__add__`, `__len__`, `__repr__` etc[4].

`isinstance()` This method is used to determine whether an object belongs to a certain class, e.g. `isinstance(<instance>,<class>)`. It returns `True` or `False`.

String formatting A good article: https://realpython.com/python-f-strings/

Cardinality A mathematical concept, which means the number of elements in a set.

Inheritance and composition Inheritance expresses the *is a* relationship, e.g. The derived class/-subclass/child class *is a* type of the base class/parent class. Composition expresses the *has a* relationship, e.g. A composite *has a* component.

Everything is python is an object. Everything is python is an object. Modules are objects, class definitions and functions are objects, and of course, objects created from classes are objects too.

`dir()` `dir(<object>)` returns all the members (properties and methods) of that object.

Python object class When we define a python class like this: `class Dog()`, in fact we are calling: `class Dog(object)`. Namely, the class we define inherits from the `object` class. But there is an exception, that is the `Exception` class. When we want to define our own error message and raise it, it has to be inherited from the `BaseException` class or `Exception` class.

```
class MyError(Exception):
    pass
```

***args and **kwargs** See this webpage.

Python decorator @ See this webpage.

Python function annotations See the webpages webpage 1, webpage 2, webpage 3.

8.2 Jupyter notebook

%%time Put %%time at the top of the cell, you will have the wall time of executing this cell in the output.

9 Selected Problems

1. Show that

$$\mathbb{E}[x_n x_m] = \mu^2 + I_{nm}\sigma^2, \tag{228}$$

where x_n and x_m denote data points sampled from a Gaussian distribution with mean μ and variance σ^2, and I_{nm} satisfies $I_{nm} = 1$ if $n = m$ and $I_{nm} = 0$ otherwise. And use the above result to prove

$$\mathbb{E}[\mu_{ML}] = \mu, \tag{229}$$

$$\mathbb{E}[\sigma^2_{ML}] = \frac{N-1}{N}\sigma^2, \tag{230}$$

where μ_{ML} and σ^2_{ML} denote the maximum-likely mean and variance of N independently and identically distributed samples of x.

References

[1] https://www.geeksforgeeks.org/cnn-introduction-to-padding/.

[2] https://developersbreach.com/convolution-neural-network-deep-learning/.

[3] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[4] mohit_negi. Dunder or magic methods in python. https://www.geeksforgeeks.org/dunder-magic-methods-python/, 2018.

Index