Pengantar Machine Learning: Artificial Neural Network

Arya Bangun

August 15, 2022

Contents

Pengantar

Penulis ingin mengucapkan terima kasih kepada Yayasan Nusantara Sejati untuk kesempatan mengajar dan menyediakan konten materi atau bahan pengajaran ini. Dengan berkembangnya kebutuhan teknologi informasi dan data yang masif di jaman ini, pengetahuan akan *data science* dan *machine learning* menjadi ilmu pengetahuan yang penting. Bahan pengajaran ini diharapkan dapat membantu dalam memperkenalkan beberapa metode dalam *data science* dan *artificial neural network* secara singkat sehingga peserta dapat mengerti secara fundamental maupun implementasi dengan bahasa pemrograman python.

Chapter 1

Pendahuluan

Seiring dengan berkembangnya komputer dari masa ke masa, pertanyaan fundamental mengenai kemampuan komputer dalam melakukan tugas seperti manusia menjadi suatu pertanyaan yang sentral dalam perkembangan teknologi informasi. Pioneer awal dalam perkembangan komputer seperti Alan Turing, John von Neuman, Norbert Wiener, dan lainnya [1,2] melihat suatu visi yang bisa dicapai di masa depan dengan penggunaan komputer. Meskipun awalnya terlihat seperti fiksi, fondasi dalam teori komputasi, teori informasi meletakkan suatu dasar matematika fundamental bahwa hal tersebut dapat dicapai.

Perkembangan awal kecerdasan buatan atau artificial intelligence terbatas dengan kemampuan komputer yang belum memiliki kemampuan seperti komputer di masa kini. Kapastias memori, kecepatan komputasi yang terbatas membuat komputer tidak dapat melakukan banyak hal seperti yang manusia lakukan. Di sisi lain dengan terbatasnya data yang bisa diakses untuk melatih komputer menjadi lebih cerdas membuat angan-angan kecerdasan buatan seperti terhambat. Dengan berkembangnya performa komputer saat ini sehingga mampu melakukan kalkulasi bukan hanya pada *Central Processing Unit (CPU)* melainkan juga *Graphics Processing Unit (GPU)* [3] membuat *artificial neural network* semakin mudah untuk diimplementasikan. Di sisi lain, perkembangan internet dan banyaknya data yang diciptakan oleh manusia juga membuat stagnasi dalam perkembangan dan penelitian kecerdasan buatan menjadi hidup kembali. Rata-rata data yang manusia ciptakan selama semenit di tahun 2021 bisa dilihat di gambar 1.1.

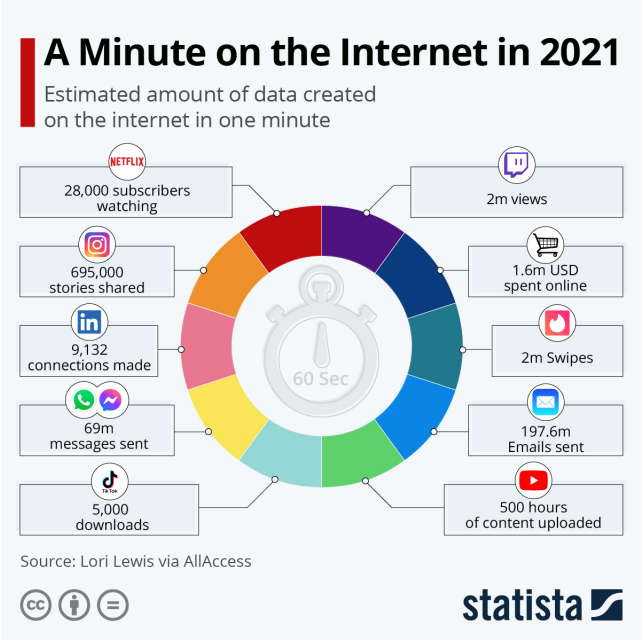

Figure 1.1: Rata-rata data yang diciptakan manusia per menit di tahun 2021 © Copyright Statista

Perkembangan struktur *neural network* baru mulai bermunculan dengan beberapa operasi metodologi seperti *convolutional neural network* [4], yaitu menggunakan operasi konvolusi pada tiap layer, *generative adversarial network* [5], yaitu struktur *neural network* yang bisa merekonstruksi gambar dari distribusi

random, dan bahkan *transformer network* [6] yang memiliki performa yang bagus dalam bidang *natural language processing* untuk aplikasi seperti Apple Siri, Amazon Alexa, maupun Google, lihat gambar 1.2, membuat perkembangan riset dan aplikasi bidang ini semakin maju dalam beberapa tahun terakhir.

Figure 1.2: Contoh aplikasi natural language processing © Amazon, Google, dan Apple

Bahkan dalam aplikasi di ilmu alam sendiri, Deep Mind sebuah perusahaan riset di Inggris yang akhirnya diakuisisi oleh Google, menciptakan struktur *neural network* baru yang bisa memprediksi struktur protein, yaitu Alphafold. Deep mind juga menciptakan neural network untuk bermain Go dan bahkan telah mengalahkan pemain profesional Go sendiri, yaitu AlphaGo [7, 8] seperti pada gambar 1.3. Dua kunci utama dalam perkembangan *artificial neural network* adalah banyaknya data yang digunakan untuk melatih *artificial neural network* dan kekuatan komputasi komputer sendiri.

Figure 1.3: Alphafold dan AlphaGo © DeepMind, Netflix

Dengan perkembangan teknologi *neural network* yang sangat pesat tersebut, sangat penting bagi kita untuk juga mengerti secara konsep sederhana sebelum juga ikut berpartisipasi pada perkembangan teknologi tersebut secara global. Terlebih lagi, untuk juga bisa digunakan di Indonesia dan secara spesifik untuk aplikasi beberapa industri yang bisa disupport dengan teknologi tersebut. Modul pembelajaran pengantar *neural network* ini bisa diharapkan untuk membantu peserta agar bisa mengerti struktur yang ada di dalam *artificial neural network* dan mengimplementasikan secara sederhana melalui bahasa pemrograman python. Disisi lain juga mengenal *modern tools* yang dipakai dalam perkembangan teknologi *artificial neural network*, seperti Pytorch maupun Tensorflow.

Chapter 2

Review Fundamental

Pada bagian ini kita akan mengulang sedikit fundamental matematika dan pemrograman python sehingga kita bisa mengerti operasi matematika yang ada di balik struktur *artificial neural network*. Secara umum data yang direpresentasikan pada komputer merupakan data yang diskrit dan disimpan dalam bentuk array ataupun list. Sehingga perlu operasi matematika yang bisa digunakan untuk mengoptimalkan data tersebut. Oleh karena itu kita akan membahas secara singkat aljabar linear dan teori optimisasi. Secara implementasi, kita akan menggunakan bahasa pemrograman python.

2.1 Aljabar Linear

2.1.1 Matriks dan Vektor

Data, baik itu gambar ataupun suara, yang diproses dalam komputer dapat direpresentasikan dengan matriks dan vektor dalam bilangan real ataupun kompleks. Kecuali gambar dalam bentuk grayscale yang bisa direpresentasikan dengan sebuah matriks, secara umum gambar berwarna memiliki tiga kanal matriks untuk merepresentasikan semua warna yaitu RGB, yang merupakan singkatan dari Red (Merah), Green (Hijau), dan Blue (Biru). Dengan kata lain semua warna yang kita lihat bisa direkonstruksi dengan tiga kombinasi warna tersebut. Lihat gambar 2.1 untuk penjelasan lebih. Disisi lain suara audio juga

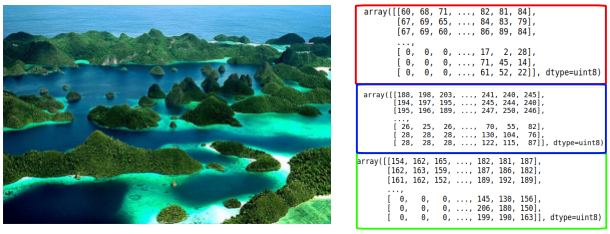

Figure 2.1: Gambar RGB dan representasi dalam python array

dapat direpresentasikan dengan dua kanal vektor untuk suara stereo dan satu kanal vektor untuk suara mono. Gambar 2.2 merupakan contoh representasi vektor pada audio mono.

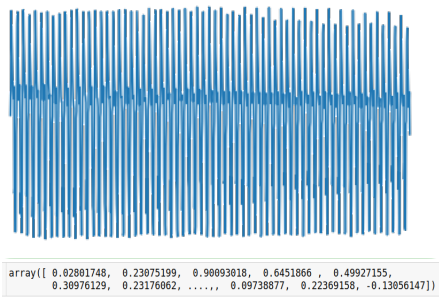

Figure 2.2: Gambar sound dan representasi dalam python array

Secara definisi kita akan menggunakan huruf kecil tebal untuk merepresentasikan vector, misal $\mathbf{x} \in \mathbb{R}^N$. Dengan kata lain vektor $\mathbf{x}$ memiliki sebanyak N element bilangan real, yaitu

$$\mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_N \end{pmatrix}.$$

Konsep N elemen vektor merupakan suatu abstraksi dalam matematika karena kita hanya bisa membayangkan vektor dalam tiga dimensi, seperti yang terlihat di gambar 2.3. Namun konsep ini sangat penting karena dengan membuat variabel N kita bisa merepresentasikan banyaknya data dengan variabel yang umum. Dalam gambar 2.3, kita bisa melihat vektor pada titik X memiliki elemen

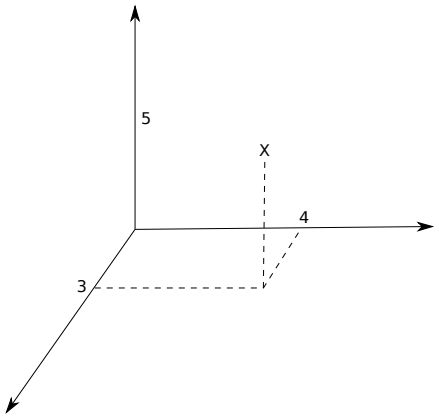

Figure 2.3: Vektor pada koordinat tiga dimensi

$$\mathbf{x} = \begin{pmatrix} 3 \\ 4 \\ 5 \end{pmatrix}.$$

Hal yang sama bisa kita lakukan pada matriks, misal kita bisa menulis matriks $\mathbf{A} \in \mathbb{R}^{m \times N}$, yang berarti kita memiliki matriks dengan total element $m \times N$ bilangan real.

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1N} \\ a_{21} & a_{22} & \dots & a_{2N} \\ \vdots & \vdots & \dots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mN} \end{pmatrix},$$

Secara tidak langsung, kita juga bisa melihat matriks sebagai kumpulan vektor, dimana dalam contoh kita matriks $\mathbf{A}$ merupakan kumpulan N elemen vektor sebanyak m.

2.1.2 Operasi pada Matriks dan Vektor

Kita juga dapat melakukan operasi matematika pada matriks dan vektor, misal penjumlahan, perkalian, perkalian element, dan lain sebagainya. Sebagai contoh penjumlahan dua matriks dengan dimensi yang sama dapat ditulis sebagai berikut $\mathbf{C} = \mathbf{A} + \mathbf{B} \in \mathbb{R}^{m \times N}$.

$$\mathbf{C} = \begin{pmatrix} a_{11} + b_{11} & a_{12} + b_{12} & \dots & a_{1N} + b_{1N} \\ a_{21} + b_{21} & a_{22} + b_{22} & \dots & a_{2N} + b_{2N} \\ \vdots & \vdots & \dots & \vdots \\ a_{m1} + b_{m1} & a_{m2} + b_{m2} & \dots & a_{mN} + b_{mN} \end{pmatrix},$$

Perlu dicatat penjumlahan dua matriks dengan dimensi yang tidak sama tidak berlaku. Hal yang sama juga berlaku pada penjumlahan dua vektor yang memiliki dimensi yang berbeda.

Perkalian dot atau perkalian skalar pada dua vektor $\mathbf{x}, \mathbf{y} \in \mathbb{R}^N$ yang memiliki dimensi yang sama dapat ditulis dengan

$$\langle \mathbf{x}, \mathbf{y} \rangle = \sum_{i=1}^{N} x_i y_i$$

Perkalian dua matriks hanya berlaku jika kolom pada matriks pertama dan baris pada matriks ke dua memiliki dimensi yang sama, misal $\mathbf{C} = \mathbf{AB} \in \mathbb{R}^{m \times m}$ untuk matriks $\mathbf{A} \in \mathbb{R}^{m \times N}$ dan matriks $\mathbf{B} \in \mathbb{R}^{N \times m}$. Perlu diingat bahwa hasil akhir perkalian memiliki dimensi yang sama dengan dimensi baris pada matriks pertama dan dimensi kolom pada matriks kedua.

$$\begin{aligned}
\mathbf{C} = \begin{pmatrix} c_{11} & c_{12} & \dots & c_{1N} \\ c_{21} & c_{22} & \dots & c_{2N} \\ \vdots & \vdots & \dots & \vdots \\ c_{m1} & c_{m2} & \dots & c_{mN} \end{pmatrix} &= \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1N} \\ a_{21} & a_{22} & \dots & a_{2N} \\ \vdots & \vdots & \dots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mN} \end{pmatrix} \begin{pmatrix} b_{11} & b_{12} & \dots & b_{1m} \\ b_{21} & b_{22} & \dots & b_{2m} \\ \vdots & \vdots & \dots & \vdots \\ b_{N1} & b_{N2} & \dots & b_{Nm} \end{pmatrix} \\
&= \begin{pmatrix} \sum_{j=1}^{N} a_{1j} b_{j1} & \sum_{j=1}^{N} a_{1j} b_{j2} & \dots & \sum_{j=1}^{N} a_{1j} b_{jm} \\ \sum_{j=1}^{N} a_{2j} b_{j1} & \sum_{j=1}^{N} a_{2j} b_{j2} & \dots & \sum_{j=1}^{N} a_{2j} b_{jm} \\ & \vdots & \vdots & \dots & \vdots \\ \sum_{j=1}^{N} a_{mj} b_{j1} & \sum_{j=1}^{N} a_{mj} b_{j2} & \dots & \sum_{j=1}^{N} a_{mj} b_{jm} \end{pmatrix}
\end{aligned}$$

Jika dilihat secara sekilas perkalian matriks merupakan kumpulan dari perkalian dot atau perkalian skalar, sebagai contoh elemen matriks $c_{11} = \sum_{j=1}^{N} a_{1j} b_{j1}$ merupakan perkalian antara vektor di baris pertama matriks $\mathbf{A}$ dengan vektor di kolom pertama matriks $\mathbf{B}$. Perlu diingat bahwa perkalian matriks sendiri bukan operasi yang komutatif, artinya perkalian $\mathbf{AB}$ tidak sama dengan $\mathbf{BA}$.

Perkalian matriks berbeda dengan perkalian elemen matriks, perkalian elemen matriks atau juga disebut perkalian Hadamard, dinamakan karena pertama kali notasinya diperkenalkan oleh Jacques Hadamard seorang matematikawan Prancis, yang mana memiliki notasi $\mathbf{C} = \mathbf{A} \circ \mathbf{B}$

$$\begin{aligned}
\mathbf{C} = \begin{pmatrix} c_{11} & c_{12} & \dots & c_{1N} \\ c_{21} & c_{22} & \dots & c_{2N} \\ \vdots & \vdots & \dots & \vdots \\ c_{m1} & c_{m2} & \dots & c_{mN} \end{pmatrix} &= \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1N} \\ a_{21} & a_{22} & \dots & a_{2N} \\ \vdots & \vdots & \dots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mN} \end{pmatrix} \circ \begin{pmatrix} b_{11} & b_{12} & \dots & b_{1m} \\ b_{21} & b_{22} & \dots & b_{2m} \\ \vdots & \vdots & \dots & \vdots \\ b_{N1} & b_{N2} & \dots & b_{Nm} \end{pmatrix} \\
&= \begin{pmatrix} a_{11} b_{11} & a_{12} b_{12} & \dots & a_{1m} b_{1m} \\ a_{21} b_{21} & a_{22} b_{22} & \dots & a_{2m} b_{2m} \\ & \vdots & \vdots & \dots & \vdots \\ a_{N1} b_{N1} & a_{N2} b_{N2} & \dots & a_{Nm} b_{Nm} \end{pmatrix}
\end{aligned}$$

Sama dengan penjumlahan matriks, perkalian elemen matriks harus memiliki dimensi yang sama. Berbeda dengan perkalian matriks yang tidak komutatif, perkalian elemen matriks merupakan operasi yang komutatif, yang berarti $\mathbf{A} \circ \mathbf{B} = \mathbf{B} \circ \mathbf{A}$.

Norm vektor merupakan fungsi yang dapat didefinisikan sebagai berikut

$$\|\mathbf{x}\|_p^p = \sum_{i=1}^{N} |x_i|^p$$

Norm vektor merupakan suatu fungsi yang berguna sebagai parameter untuk mengukur di ruang vektor. Norm yang paling umum sering di pakai adalah norm Euklidean, yaitu ketika $p = 2$. Norm ini sangat berguna nanti ketika kita masuk ke teori optimisasi atau ketika kita mau melatih *artifisial neural network* kita, karena kita bisa mengukur *cost function* atau fungsi loss, atau error prediksi pada *artificial neural network* kita dengan menggunakan Euklidean norm. Dengan kata lain kita mau mengoptimalkan variabel pada *artificial neural network* kita yang meminimalkan error pada prediksi kita. Sebagai contoh, kita bisa mengukur perbedaan antara dua vektor $\mathbf{x}, \mathbf{y} \in \mathbb{R}^N$ dengan menggunakan norm.

$$\|\mathbf{x} - \mathbf{y}\|_2^2 = \sum_{i=1}^{N} |x_i - y_i|^2$$

2.1.3 Operasi Konvolusi

Selain menggunakan operasi perkalian matriks dan vektor kita juga akan membahas operasi konvolusi. Operasi ini berguna dalam implementasi *convolutional neural network*. Misal kita mempunyai sebuah matriks $\mathbf{A} \in \mathbb{R}^{I \times J}$ dan kernel filter $\mathbf{F} \in \mathbb{R}^{M \times N}$. Operasi konvolusi dapat ditulis sebagai berikut

$$\mathbf{R} = \mathbf{A} \circledast \mathbf{F}$$

Proses dalam tiap elemen dapat ditulis sebagai berikut

$$R_{ij} = \sum_{m=1}^{M} \sum_{n=1}^{N} A_{i+m,j+n} F_{m,n},$$

dimana $i \in [I]$ dan $j \in [J]$ merupakan element pada matriks. Jika dilihat secara sekilas maka operasi konvolusi tidak lain merupakan perkalian dan penjumlahan antara elemen dari kernel filter dan data. Perlu dicatat disini definisi konvolusi yang kita gunakan merupakan definisi korelasi dalam konvensi matematika namun definisi ini yang digunakan dalam konvensi di komunitas *machine learning*. Disisi lain kedua operasi tersebut saling berhubungan satu dengan yang lain tergantung dari pergeseran yang dilakukan. Operasi konvolusi akan kita pelajari lebih lanjut di bab *convolutional neural network*.

2.2 Teori Optimisasi

Seperti sudah disinggung pada bab sebelumnya, tujuan utama dalam pembelajaran atau melatih *artificial neural network* merupakan problem optimisasi. Dengan kata lain, dengan menggunakan data atau training yang banyak kita berusaha untuk mencari variabel optimal bagi *neural network* kita sehingga memiliki prediksi, atau klasifikasi yang memiliki error yang sangat kecil. Oleh karena itu teori optimisasi merupakan hal yang sangat krusial dalam implementasi *artificial neural network*.

Secara umum teori optimisasi dapat kita tulis sebagai berikut

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & f_0(x) \\ \text{subject to} \quad & f_i(x) \leq b_i, \ i = 1, \ldots, m. \\ & h_i(x) = c_i, \ i = 1, \ldots, m. \end{aligned}$$

Fungsi $f_0(x)$ disebut juga fungsi objektif dan variabel x merupakan variabel yang harus dioptimalkan sehingga fungsi objektif menjadi minimal. Misalkan x^* merupakan variabel optimal yang meminimalkan fungsi $f_0(x)$, variabel ini disebut juga *optimizer* atau *minimizer*. Perlu diingat jika problem optimisasi ingin memaksimalkan suatu fungsi, variabel optimal disebut juga *maximizer*. Selain fungsi objektif, kita memiliki juga fungsi batasan, yaitu fungsi $f_i(x)$ disebut fungsi pertidaksamaan dan $h_i(x)$ disebut juga fungsi persamaan. Kedua fungsi ini berguna untuk mengurangi ruang solusi dalam problem optimisasi. Semua variabel x yang memenuhi fungsi batasan disebut juga variabel yang feasibel. Sehingga secara tidak langsung nilai optimal dari sebuah problem optimisasi merupakan salah satu dari variabel feasibel tersebut. Secara intuisi dapat dilihat dari gambar 2.4 Dari gambar 2.4, dapat dilihat bahwa variabel x_1, x_2, x_3 merupakan variabel feasibel karena memenuhi batasan pada problem optimisasi, namun variabel x_2 merupakan variabel optimal karena menghasilkan solusi yang optimal.

Sebagai contoh kita bisa lihat aplikasi optimisasi yang umum yaitu optimisasi *least square* sebagai berikut

$$\underset{\mathbf{x}}{\text{minimize}} \quad \|\mathbf{y} - \mathbf{A}\mathbf{x}\|_2^2$$

Misalkan kita memiliki vektor $\mathbf{y} \in \mathbb{R}^m$ dan matriks $\mathbf{A} \in \mathbb{R}^{m \times N}$ dan $m > N$. Kita ingin mencari vektor $\mathbf{x} \in \mathbb{R}^N$ yang meminimalkan Euklidean norm antara vektor $\mathbf{y}$ dengan perkalian matrix $\mathbf{A}$ dan vektor $\mathbf{x}$. Problem *least square* merupakan problem yang memiliki solusi analitik bahkan digunakan untuk regresi linear. Secara umum tidak banyak problem optimisasi yang memiliki solusi analitik. Solusi untuk problem optimisasi tersebut adalah

$$\mathbf{x}^* = \left(\mathbf{A}^T\mathbf{A}\right)^{-1} \mathbf{A}^T\mathbf{y}$$

Disini notasi $\mathbf{A}^T$ merupakan transpose dari matriks $\mathbf{A}$ dan $\mathbf{A}^{-1}$ merupakan inverse dari matriks $\mathbf{A}$.

Secara konsep, notasi yang kita gunakan dalam teori optimisasi hanya mencoba merumuskan permasalahan yang ada. Secara implementasi, ada beberapa cara yang dilakukan untuk mencari nilai optimal, yaitu dengan metode order pertama atau gradient. Pada materi kali ini kita akan fokus pada

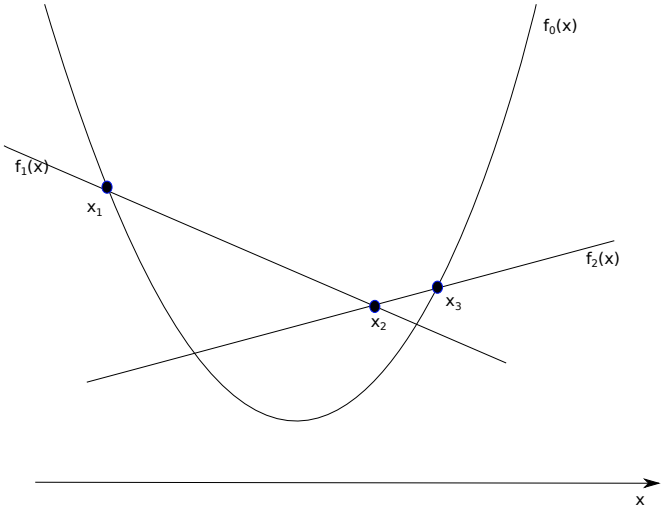

Figure 2.4: Fungsi objektif dan fungsi batasan

metoda gradient karena digunakan dalam *artificial neural network*, secara umum terdapat metoda yang lain, seperti metoda Newton-Rhapson, dengan menggunakan order kedua, atau Hessian.

Metoda gradient merupakan metoda yang sangat simpel namun perlu mengetahui turunan pertama dari fungsi objektif dari problem optimisasi kita. Metoda gradient merupakan metoda iteratif yang terus update solusi sehingga kita mendapat point kritis, atau ketika gradient sama dengan nol. Secara intuisi divisualisasikan di gambar 2.5. Disini terlihat bahwa metode gradient berusaha untuk mencari nilai

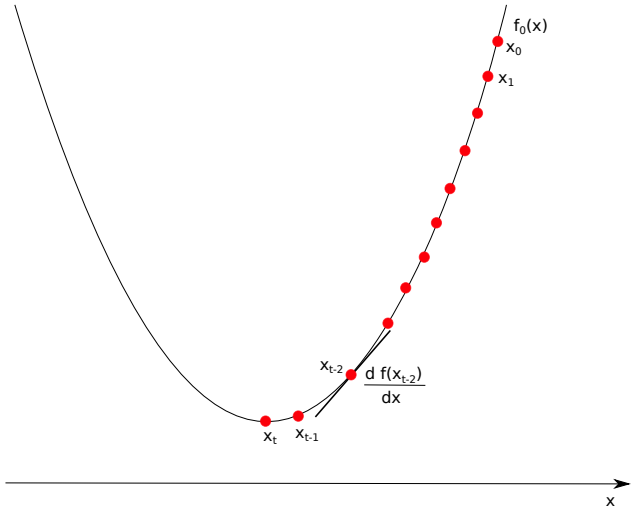

Figure 2.5: Metode gradient

optimal dengan update turunan fungsi objektif pada variabel awal. Dari kalkulus, kita bisa melihat bahwa turunan merupakan perbedaan diantara dua titik pada satu fungsi ketika limitnya mendekati nol. Variabel yang berguna untuk menentukan langkah turunan disebut *step size* atau *learning rate*. Secara singkat implementasi metode gradient dirangkum di Algoritma 1. Dalam perkembangan *artificial neural network* sendiri maupun *machine* atau *deep learning* ada banyak variasi dari metode gradient yang diciptakan untuk menolong mempercepat ditemukannya solusi optimal beberapa metode gradient yang terkenal misalnya ADAM [9], AdaGrad [10], AdaDelta [11], dan lain sebagainya. Perlu diperhatikan bahwa *neural network* sendiri merupakan sistem yang sangat kompleks sehingga fungsi objektifnya tidak sesimpel yang kita diskusikan disini. Kita akan mempelajarinya di bab-bab berikutnya.

Contoh jika kita mempunyai fungsi objektif $f_0(x) = x^2$, maka $\frac{\mathrm{d}f_0(x)}{\mathrm{d}x}) = 2x$. Sehingga kita bisa menulis update pada metode gradient sebagai

$$x^{t+1} = x^t - \alpha 2x^t$$

Algorithm 1 Metoda gradient

1: **Inisial:**
- Variabel inisial x_0
- Step size α
- Banyaknya iterasi T

2: **while** $\frac{\mathrm{d}f_0(x_t)}{\mathrm{d}x} \neq 0$ atau $t < T$ **do**
3: $x_{t+1} = x_t - \alpha \frac{\mathrm{d}f_0(x_t)}{\mathrm{d}x}$
4: $t = t + 1$
5: **end while**

Salah satu teknik di optimisasi yang sering digunakan untuk melatih *neural network* adalah dengan menggunakan regularisasi. Regularisasi sangat berguna untuk mencegah problem *overfitting* yang akan muncul. Problem ini muncul ketika kita melatih *neural network* kita atau parameter hipotesis kita sehingga sangat spesifik optimal terhadap spesifik dataset. Hal ini akan membuat *neural network* kita sangat cenderung spesifik ke arah satu dataset saja, sehingga ketika kita melatih dataset baru *neural network* kita gagal untuk memprediksi atau mengklasifikasi secara benar. Regularisasi dapat diimplementasikan dengan menambahkan fungsi pada problem optimisasi kita. Sebagai contoh mari kita lihat kembali problem optimisasi *least square*

$$\underset{\mathbf{x}}{\text{minimize}} \quad \|\mathbf{y} - \mathbf{A}\mathbf{x}\|_2^2$$

Misalnya kita ingin menambahkan batasan pada hasil optimisasi kita sehingga Euklidean norm pada solusi kita memiliki nilai yang minimal. Maka kita bisa menambahkan batasan seperti berikut

$$\underset{\mathbf{x}}{\text{minimize}} \quad \|\mathbf{y} - \mathbf{A}\mathbf{x}\|_2^2 + \lambda \|\mathbf{x}\|_2^2$$

Perlu diketahui disini kita menambahkan skala λ untuk mengontrol sejauh apa kita inginkan regularisasi kita, jika nilai λ sangat besar maka pengaruhnya akan cukup besar pada optimisasi kita. Disisi lain menambahkan regularisasi akan membuat fungsi turunan berbeda karena ada fungsi baru. Regularisasi akan kita bahas secara spesifik di bab selanjutnya

2.2.1 Tugas Teori Optimisasi

- Seperti yang telah dijelaskan bahwa suara dapat direpresentasikan dengan menggunakan vektor dan gambar dapat direpresentasikan dengan matriks untuk gambar grayscale dan tiga kanal matriks pada gambar RGB. Bagaimana menurut anda data video?
- Seperti yang kita lihat untuk problem *least square* kita memiliki solusi

  $$\mathbf{x}^* = \left(\mathbf{A}^T\mathbf{A}\right)^{-1}\mathbf{A}^T\mathbf{y}$$

  Dapatkah anda menurunkan formula tersebut? Hint: Pakai persamaan gradient sama dengan nol
- Secara umum contoh kita diatas merupakan fungsi konveks, yaitu dimana nilai minimumnya hanya ada satu. Dengan kata lain jika fungsi objektif kita adalah fungsi konveks maka solusi optimal global sama dengan solusi optimal lokal yang ada. Secara khusus kita hanya memiliki satu solusi. Menurut anda bagaimana jika problem optimisasi tidak konveks? Secara gambaran problem optimisasi secara umum di *artificial neural network* merupakan problem tidak konveks!
- Kita telah belajar bahwa kita perlu mendefinisikan *step size* atau *learning rate* α untuk metode gradient. Menurut anda apa pengaruh jika α terlalu besar dan jika α terlalu kecil?

2.2.2 Solusi Tugas Teori Optimisasi

- Secara umum video merupakan kumpulan dari frame gambar bergantung pada waktu. Jadi kita kita memiliki gambar grayscale maka video merupakan data tiga dimensional atau kumpulan dari matriks gambar yang bergantung terhadap waktu. Jika data RGB maka data merupakan tiga dimensional yang bergantung pada waktu dari kumpulan tiga matriks untuk kanal RGB.

- Mari kita tulis persamaan *least square*

$$\begin{aligned}\|\mathbf{y}-\mathbf{A}\mathbf{x}\|_2^2 &= (\mathbf{y}-\mathbf{A}\mathbf{x})^T(\mathbf{y}-\mathbf{A}\mathbf{x}) \\ &= \mathbf{y}^T\mathbf{y}-\mathbf{y}^T\mathbf{A}\mathbf{x}-\mathbf{x}^T\mathbf{A}^T\mathbf{y}+\mathbf{x}^T\mathbf{A}^T\mathbf{A}\mathbf{x}\end{aligned}$$

  Jika kita hitung turunan persamaan tersebut kita mendapat

$$\frac{\mathrm{d}\,\|\mathbf{y}-\mathbf{A}\mathbf{x}\|_2^2}{\mathbf{x}} = -2\mathbf{A}^T\mathbf{y}+2\mathbf{A}^T\mathbf{A}\mathbf{x}=0$$

  Sehingga kita mendapatkan

$$\mathbf{A}^T\mathbf{y}=\mathbf{A}^T\mathbf{A}\mathbf{x}$$

  Dengan kata lain kita mendapat

$$\mathbf{x}^*=\left(\mathbf{A}^T\mathbf{A}\right)^{-1}\mathbf{A}^T\mathbf{y}$$

- Jika problem tidak konveks maka akan banyak minimum lokal pada fungsi objektif, sehingga kita bisa terjebak pada lokal optimal dan bukan pada global optimal. Karena metode gradient akan terjebak pada salah satu problem lokal optimal. Dalam riset di *artificial neural network* ada banyak metode yang diciptakan untuk menghindari problem ini, misalkan menambah momentum pada metode gradient atau bahkan mencoba merelaksasi problem tidak konveks menjadi konveks.
- Jika alpha terlalu kecil maka akan perlu waktu yang cukup lama untuk mencapai solusi global, atau dengan bahasa matematikanya kita mendapatkan konvergensi yang sangat lambat. Namun jika alpha terlalu besar problem yang ada yaitu kita bisa jadi melewatkan solusi global dan tidak terjadi konvergensi pada metode gradient.

2.3 Pemrograman Python

Dalam bagian ini kita akan belajar sedikit pemrograman python yang bisa langsung digunakan untuk implementasi sederhana. Untuk pembelajaran secara utuh diharapkan bisa belajar langsung di tutorial resmi python: https://docs.python.org/3/tutorial/index.html. Disini kita akan belajar beberapa hal yang berguna untuk kursus ini, seperti module numpy, scipy, fungsi, loop. Terkhusus bagaimana representasi matriks dan vektor pada python.

2.3.1 Modul Numpy

Modul numpy sangat berguna dalam merepresentasikan matriks dan vektor pada python. Numpy sendiri merupakan singkatan dari *Numeric Python*. Perlu diingat bahwa numpy sendiri bukan modul yang ada secara langsung di python sehingga para pengguna harus install terlebih dahulu. Didalam bahasa pemrograman tidak mengenal yang disebut matriks dan vektor, namun istilah yang sering digunakan yaitu array. Perlu diingat bahwa berbeda dengan MATLAB, python memulai index element dari nol. Berikut merupakan contoh untuk membuat matriks dan vektor.

```
# Import digunakan untuk mengambil modul yang sudah ada
# Disini kita mau import module numpy dengan singkatan sebagai np
# Sehingga kita bisa langsung menggunakan np
import numpy as np

# Disini kita mau buat array dengan semua element nol dengan dimensi 5 x 10
# Dengan tipe data numpy array dan elementya float 32
A = np.zeros((5,10), dtype = np.float32)

# Cek tipe data
A.dtype

# Perlu diingat bahwa jika kita tidak menulis parameter dtype maka akan dialokasikan
    float 64
# Perbedaan mendasar adalah berapa bit yang ingin dialokasikan untuk merepresentasikan
    data,
# misal 32 atau 64 bit

# Kita bisa alokasikan array dengan random distribusi Gaussian atau normal
# Perhatikan perbedaan syntax yang ada!
```

```
B = np.random.randn(5,10)

# Untuk mengalokasikan vektor kita bisa lakukan dengan
x = np.random.randn(10,1)

# Untuk mengetahui dimensi kita bisa menggunakan
A.shape
x.shape

# Untuk mengakses element dimensi kita bisa menggunakan
A.shape[0]
A.shape[1]

# Untuk akses element pada matrix misal element (1,1)
# Ingat indeks python dari 0
B[0,0]

# Untuk akses element baris pertama
B[0,:]
B[0]

# Kedua syntaks diatas sama, sehingga kita bisa cek dengan menggunakan
# Jika sama maka akan tertulis True
np.allclose(B[0,:],B[0])
```

Listing 2.1: Contoh membuat array

Untuk operasi aljabar linear pada python ada beberapa syntaks yang bisa digunakan seperti berikut

```
import numpy as np

# Buat dua array
A = np.random.randn(5,10)
B = np.random.randn(10,6)

# Operasi perkalian matriks
C = A@B
D = A.dot(B)

# Ingat perkalian matriks merupakan kumpulan perkalian dot atau skalar
# Operasi diatas sama, namun kebanyakan menggunakan operator @ secara langsung
np.allclose(C,D)

# Perlu diingat bahwa @ merupakan perkalian matriks dan untuk perkalian element atau Hadamard
# Menggunakan operasi .

A = np.random.randn(7,9)
B = np.random.randn(7,9)

# Operasi perkalian elemen
C = A*B
D = B*A

# Cek bahwa perkalian elemen matriks merupakan komutatif
np.allclose(C,D)


# Penjumlahan dalam matriks dan vektor
E = A + B

x = np.random.randn(5,1)
jumlah_x = x[0] + x[1] + x[2] + x[3] + x[4]
jumlah_x_2 = np.sum(x)

# Cek apakah sama
np.allclose(jumlah_x, jumlah_x_2)

# Penjumlahan semua elemen di matriks atau vektor dengan fungsi numpy
jumlah_A = np.sum(A)

# Penjumlahan elemen kolom matriks dan baris, index 1 menyatakan aksis kolom dan 0 aksis baris
jumlah_A_kolom = np.sum(A,1)
```

```
jumlah_A_baris = np.sum(A,0)
```

Listing 2.2: Contoh operasi python

Seperti yang sudah kita bahas pada review aljabar linear, kita bisa juga memanggil fungsi norm yang ada di numpy untuk mengukur di ruang vektor.

```
import numpy as np

x = np.random.randn(5,1)

# Ingat definisi norm-2 atau Euklidean norm di bab sebelumnya
norm_x_original = np.sqrt(np.sum(np.abs(x)**2))
norm_x_numpy = np.linalg.norm(x)

# Cek kesamaan
np.allclose(norm_x_original, norm_x_numpy)

# Cek kesamaan dengan menggunakan norm
# Alokasikan vektor dengan semua elemen 1 dan 0
a = np.ones((5,1))
b = np.zeros((5,1))

# Secara langsung perbedaan dua vektor ini adalah akar 5
error = np.linalg.norm(a - b)
print(error)
np.allclose(error, np.sqrt(5))
```

Listing 2.3: Contoh norm

2.3.2 Fungsi

Dalam bahasa pemrograman untuk mempermudah kita mendokumentasikan program dan membuat program kita terlihat lebih rapi kita bisa menggunakan fungsi seperti berikut

```
import numpy as np
def fungsi_penjumlahan(input_vektor):
    """
    Ini adalah docstring, yaitu membantu menjelaskan apa itu parameter dari fungsi dan
    apa keluarannya

    Parameters
    ----------
    input_vektor : array


    Return
    ------
    total: int atau float

    """

    # Alokasi variabel
    total = 0

    # Mulai loop
    for idx in range(len(input_vektor)):
      total += input_vektor[idx]

    return total

# Untuk di python syntaks dibawah berguna untuk mengetahui dimana fungsi utama
if __name__ == '__main__':
    a = np.random.randn(100,1)
    total_jumlah = fungsi_penjumlahan(input_vektor=a)

    # Cek kesamaan dengan numpy
    assert np.allclose(total_jumlah, np.sum(a))
    print(total_jumlah, np.sum(a))
```

Listing 2.4: Contoh fungsi

Perlu diketahui biasanya kita menulis syntaks dan kumpulan fungsi pada suatu file yang memiliki ekstensi `.py`, pada contoh ini misalnya kita buat fungsi bernama `fungsi_penjumlahan.py` Untuk mengeksekusi fungsi tersebut biasa kita bisa menggunakan diterminal linux `python3 fungsi_penjumlahan.py`. Dalam pembelajaran kita kali ini kita akan banyak menggunakan fungsi untuk implementasi *artificial neural network*.

2.3.3 Tugas Pemrograman Python

- Seperti contoh untuk optimisasi fungsi $f_0(x) = x_1^2 + x_2^2 + 0.5$, bisakah anda mengimplementasikan metode gradient pada python? Hint: Tulis dua fungsi `def run_gradient` dan `def gradient`. Perhatikan apa yang terjadi jika kita menaruh angka yang berbeda pada *step size* dan iterasi?

- Meskipun kita memiliki solusi pada problem *least-square*, bisakah teman-teman implementasikan metode *least square* dengan metode gradient? Bandingkan hasilnya dengan metode analitik.

2.3.4 Solusi Tugas Pemrograman Python

```
import numpy as np
def gradient(x1, x2):
    """


    Parameters
    ----------
    x : float


    Return
    ------
    grad: float

    """
    grad_1 = 2*x1 + 0.5
    grad_2 = 2*x2 + 0.5

    return grad_1, grad_2

def run_gradient(x1 ,x2, iter, alpha):

    temp_x1 =[]
    temp_x2 = []
    for it in range(iter):
        # Calculate gradient
        grad_1, grad_2 = gradient(x1, x2)
        # Update
        x1 = x1 - alpha*grad_1
        x2 = x2 - alpha*grad_2
        # Cek kondisi critical point
        if abs(2*x1 + 0.5) <= 1e-6 and abs(2*x2 + 0.5) <= 1e-6 :
            print('Solusi x1 dan x2', str(x1), str(x2))
            break
        # Append
        temp_x1.append(x1)
        temp_x2.append(x2)

        print(x1,x2)
    return temp_x1,temp_x2


if __name__ == '__main__':
    x = np.random.rand(2,1)*10
    # Coba ganti parameter alpha dari 0.1, 0.01 dan 5
    alpha = 5
    iter = 500
    all_x1, all_x2 = run_gradient(x[0], x[1], iter, alpha)
```

Listing 2.5: Solusi python optimisasi

```python
import numpy as np
def gradient(A,x,y):
    """


    Parameters
    ----------
    x : float


    Return
    ------
    grad: float

    """
    grad = 2*A.T@A@x - 2*A.T@y

    return grad

def run_gradient(x, A,y, iter, alpha):

    temp_x =[]

    for it in range(iter):
        # Calculate gradient
        grad = gradient(A,x,y)
        # Update
        x  = x  - alpha*grad
        # Cek kondisi critical point
        print('Error fungsi objective :', str(np.linalg.norm(y - A@x)**2))
        if np.linalg.norm(y - A@x)**2 <= 1e-6:

            break

        # Append



    return x


if __name__ == '__main__':
    x_asli = np.ones((10,1))
    x_inisiasi = np.random.randn(10,1)
    A = np.random.randn(20,10)
    y = A@x_asli
    alpha = 0.01
    iter = 100
    x_gradient = run_gradient(x_inisiasi, A,y,iter, alpha)
    print('Error nilai x', str(np.linalg.norm(x_gradient - x_asli)**2))

    # Hitung dengan numeric
    x_analitik = np.linalg.pinv(A.T@A)@A.T@y

    print('Perbedaan analitik dan gradient ', str(np.linalg.norm(x_analitik - x_gradient
    )**2))
```

Listing 2.6: Solusi python optimisasi least square

Chapter 3

Regresi Linear

Pada bab ini kita akan belajar regresi linear dan aplikasinya dalam prediksi. Regresi linear merupakan suatu metode yang sebenarnya sudah ada dan merupakan cabang dari statistik. Metode ini selain di statistik sendiri juga banyak digunakan di keilmuan lain, seperti pengolahan sinyal, aktuaria, dan lain sebagainya. Secara umum regresi linear merupakan suatu pendekatan optimasi dengan menggunakan fungsi linear. Dalam bab ini kita akan belajar regresi linear dari satu fitur hingga ke banyak fitur.

3.1 Regresi Linear dengan satu fitur

Sebelum memulai dalam formula matematika, ada baiknya kita melihat dari contoh yang ada dan bisa membuat mengerti problem di regresi linear lebih dalam. Misalkan kita mempunyai data harga rumah di daerah Jabodetabek seperti yang dideskripsikan di Tabel 3.1.

Table 3.1: Tabel harga rumah

Parameters	**Luas (m^2)**	**Harga (Miliar)**
Rumah 1	100	2
Rumah 2	245	4
Rumah 3	120	2.5
Rumah 4	300	8
Rumah 5	520	10
Rumah 6	123	3
⋮	⋮	⋮

Regresi linear satu fitur dapat dilihat sebagai fungsi mapping dari fitur rumah, yaitu luas, dan harganya. Secara umum data tersebut dapat divisualisasikan sebagai gambar 3.1.

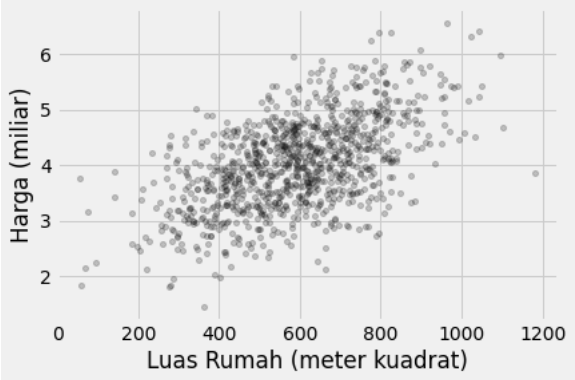

Figure 3.1: Visualisasi distribusi harga rumah

Dari visualisasi tersebut kita bisa melihat bahwa luas rumah memiliki korespondensi ke harga sehingga secara matematika kita bisa melihat bahwa dalam kasus ini kita memiliki satu fitur, yaitu luas rumah,

yang kita sebut $x \in \mathbb{R}$ dan implikasinya terhadap harga rumah $y \in \mathbb{R}$. Problem dalam regresi linear kita berusaha untuk mencari trend dari distribusi tersebut dengan persamaan linear

$$h_\theta(x) = \theta_0 + \theta_1 x$$

Sehingga dalam regresi linear satu fitur, kita ingin mencari nilai θ_0 dan θ_1 yang meminimalkan prediksi terhadap harga rumah $y \in \mathbb{R}$. Seperti yang sudah kita bahas di teori optimisasi kita bisa memodelkan dengan memperkenalkan *cost function* dengan menggunakan Euklidean norm,

$$\underset{\theta_0,\theta_1}{\text{minimize}} \quad \frac{1}{2m}\sum_{i=1}^{m}\left(h_\theta(x)_i - y_i\right)^2, \tag{3.1}$$

dimana dalam kasus ini kita memiliki sebanyak m data. Sehingga hasil dari optimisasi kita adalah dua variabel θ_0 dan θ_1 yang meminimalkan problem optimisasi diatas. Seperti yang sudah kita bahas dalam teori optimisasi, kita bisa menggunakan metode gradient untuk mendapatkan nilai optimal. Pada akhirnya kita mendapatkan persamaan garis yang optimal pada distribusi tersebut dan kita bisa memprediksi harga rumah berdasarkan luasnya dari korespondensi dua titik antara luas rumah dan harga rumah seperti yang ada di gambar 3.2

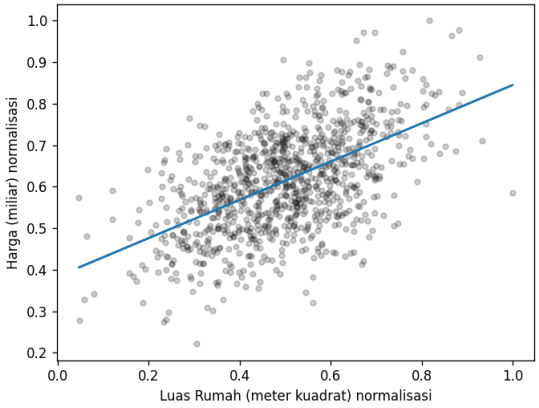

Figure 3.2: Visualisasi distribusi harga rumah dan regresi linear

Perlu diperhatikan bahwa gambar 3.2 merupakan data normalisasi dari gambar 3.1, dibagi terhadap nilai maximum luas tanah dan harga. Normalisasi sangat berguna untuk membantu mempercepat konvergensi pada metode gradient. Hal ini akan saudara jumpai pada tugas bagian bab ini.

3.2 Regresi Linear dengan banyak fitur

Pada bab sebelumnya kita sudah membahas regresi linear dengan satu fitur, namun pada kenyataannya data yang kita punya bisa memiliki lebih dari satu fitur. Kembali pada contoh yang kita punya mengenai data rumah di Jabodetabek, harga rumah tidak hanya ditentukan pada luasnya rumah namun juga misalnya luas pekarangan, total kamar, lokasi, dan lain sebagainya. Sehingga kita memuliki banyak fitur seperti yang tertulis pada Tabel 3.2.

Dengan demikian kita memiliki hipotesa untuk memprediksi harga rumah dengan persamaan

$$h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3 + \ldots + \theta_n x_n.$$

Sehingga dalam regresi linear banyak fitur, kita ingin mencari nilai θ_0, θ_1 sampai θ_n yang meminimalkan prediksi terhadap harga rumah $y \in \mathbb{R}$. Kita memiliki *cost function* seperti berikut

$$\underset{\theta_0,\theta_1,\ldots,\theta_n}{\text{minimize}} \quad \frac{1}{2m}\sum_{i=1}^{m}\left(h_\theta(x)_i - y_i\right)^2, \tag{3.2}$$

dimana dalam kasus ini kita memiliki sebanyak m data dan fitur sebanyak n. Tujuan dari optimisasi ini adalah untuk mendapatkan variabel hipotesa $\theta_0, \theta_1, \ldots, \theta_n$. Sama seperti regresi linear dengan satu fitur, kita juga bisa menggunakan metode gradient untuk mendapatkan nilai optimal.

Table 3.2: Tabel harga rumah dengan banyak fitur

Parameters	**Luas** (m^2)	**Total kamar**	**Luas pekarangan** (m^2)	**Lokasi**	**Harga (Miliar)**
Rumah 1	100	3	20	A	6
Rumah 2	245	4	10	B	3
Rumah 3	120	2	5	A	2.5
Rumah 4	300	4	7	D	4
Rumah 5	520	6	12	C	10
Rumah 6	123	2	7	B	5
⋮	⋮	⋮	⋮	⋮	⋮

3.3 Tugas Regresi Linear

- Seperti yang sudah kita pelajari pada teori optimisasi, bisakah saudara menurunkan gradient pada *cost function* pada problem optimisasi (3.3) untuk satu fitur?
- Jika saudara sudah menurunkan gradient pada persamaan tersebut, coba tuliskan update pada metode gradient untuk regresi linear satu fitur?
- Bisakah saudara implementasikan pada python dan plot hasil optimisasi seperti gambar 3.2?
- Pada optimisasi regresi linear banyak fitur, apa perbedaan dengan optimisasi regresi linear satu fitur?
- Apa problem yang ada jika menggunakan metode gradient dengan update nilai θ_i untuk setiap $i \in \{1, 2, \ldots, n\}$?
- Bisakah saudara memodelkan dalam bentuk produk matriks dan vektor? Tuliskan persamaan gradient dalam bentuk matriks dan vektor!
- Implementasikan metode gradient dengan bentuk matriks dan vektor!
- Apa kesamaan bentuk matriks dan vektor dengan problem least-square? Bisakah teman-teman menghitung nilai optimal θ_i untuk setiap $i \in \{1, 2, \ldots, n\}$ dengan menggunakan solusi analitik dari *least-square* ?

3.4 Solusi Regresi Linear

- Gradient pada persamaan optimisasi (3.1) dapat ditulis seperti berikut

$$\frac{\mathrm{d}}{\mathrm{d}_{\theta_0}} = \frac{1}{m}\sum_{i=1}(\theta_0 + \theta_1 x_i - y_i)$$
$$\frac{\mathrm{d}}{\mathrm{d}_{\theta_1}} = \frac{1}{m}\sum_{i=1}(\theta_0 + \theta_1 x_i - y_i)\, x_i$$

- Sehingga update untuk metode gradient dapat ditulis sebagai berikut

$$\theta_0^{t+1} = \theta_0^t - \frac{\alpha}{m}\sum_{i=1}(\theta_0 + \theta_1 x_i - y_i)$$
$$\theta_1^{t+1} = \theta_1^t - \frac{\alpha}{m}\sum_{i=1}(\theta_0 + \theta_1 x_i - y_i)\, x_i$$

  untuk iterasi ke t dan update $t + 1$.

- Implementasi python dan plot sudah tertulis seperti di contoh
- Pada regresi linear banyak fitur kita memiliki banyak update untuk hipotesa, tidak hanya θ_0 dan θ_1, melainkan juga sampai θ_n.

- Problem dengan menggunakan gradient skalar seperti persamaan *cost function* di (3.3) adalah kita harus menulis banyak update θ_0 sampai θ_n. Salah satu cara untuk menolong dan mempermudah proses gradient adalah dengan bentuk matriks dan vektor.

- Persamaan matriks dan vektor dapat ditulis sebagai berikut dengan menggunakan norm vektor

$$\underset{\boldsymbol{\theta}\in\mathbb{R}^{n+1}}{\text{minimize}} \quad \frac{1}{2m}\left\|\mathbf{A}\boldsymbol{\theta}-\mathbf{y}\right\|_2^2, \tag{3.3}$$

  Dengan matriks $\mathbf{A}\in\mathbb{R}^{m\times n+1}$ dengan element matrix tersebut dapat ditulis sebagai berikut

$$\mathbf{A}=\begin{pmatrix} 1 & x_1^{(1)} & \dots & x_1^{(n)} \\ 1 & x_2^{(1)} & \dots & x_2^{(n)} \\ \vdots & \vdots & \dots & \vdots \\ 1 & x_m^{(1)} & \dots & x_m^{(n)} \end{pmatrix} \in \mathbb{R}^{m\times n+1},$$

  Perlu diingat disini kita memiliki n fitur sehingga kita menulis elemen pada matrix, tanda pangkat diatas merupakan index fitur, misal fitur (1) sampe fitur (n). Sehingga $x_1^{(1)}$ berarti fitur ke (1) element pertama. Dalam contoh kita misalnya rumah ke 1. Selanjutnya gradient least-square bisa ditulis seperti di bab sebelumnya mengenai solusi least square.

Chapter 4

Regresi Logistik

Pada bab sebelumnya kita sudah membahas mengenai regresi linear dan aplikasinya pada prediksi data. Pada bab kali ini kita akan berdiskusi mengenai problem yang umum pada area *data science* dan *machine learning*, yaitu problem klasifikasi. Secara umum problem klasifikasi sangat banyak muncul di dunia nyata, misal bagaimana kita mengklasifikasikan suatu gambar binatang adalah gambar anjing, kucing, ataukah rusa? Problem ini muncul dalam bidang diagnosis medis, misalkan bagaimana kita mengetahui dari hasil gambar *MRI* kita mengdiagnosis seorang pasien mengidap kanker atau tidak. Di era pandemi sekarang juga problem klasifikasi muncl untuk mengklasifikasi apakah seorang pasien terjangkit virus korona atau tidak. Di bab ini kita akan belajar jenis klasifikasi dengan menggunakan regresi logistik.

4.1 Klasifikasi Biner

Dalam proses klasifikasi biner, kita akan mengklasifikasikan suatu data ke dua bagian keluaran, yaitu 1 jika problem klasifikasi sesuai dengan kebenaran yang ada, 0 jika problem klasifikasi tidak sesuai dengan kebenaran yang ada. Misal untuk proses klasifikasi apakah email yang kita terima merupakan spam 1 atau tidak 0. Disisi lain dalam proses klasifikasi kanker pada sebuah gambar observasi, kita bisa mengklasifikasikan jika benar kanker dengan nilai 1 dan jika tidak ada kanker dengan nilai 0. Secara intuisi kita bisa menggunakan *threshold* sebagai alat pembantu kita untuk mengklasifikasikan sesuatu, misal kita mempunyai suatu hipotesa dari data x, yang didefinisikan sebagai $h_\theta(x)$ dan hasil klasifikasi y. Sehingga proses klasifikasi bisa kita lakukan dengan nilai threshold misal 0.5 seperti berikut

$$y = \begin{cases} 1, h_\theta(x) \geq 0.5 \\ 0, h_\theta(x) < 0.5 \end{cases}$$

Secara umum hypotesa $h_\theta(x)$ dapat didefinisikan sebagai berikut

$$h_\theta(x) = \sum_{i=1}^{n} \theta_i x_i = \boldsymbol{\theta}^T \mathbf{x}$$

Hal ini sama dengan hipotesa regresi linear, namun untuk problem klasifikasi biasanya kita mendefinisikan fungsi lain untuk menolong proses klasifikasi, fungsi ini disebut juga fungsi sigmoid atau fungsi logistik. dan didefinisikan sebagai berikut

$$h_\theta(x) = g\left(\boldsymbol{\theta}^T \mathbf{x}\right) = \frac{1}{1 + e^{-\boldsymbol{\theta}^T \mathbf{x}}}$$

Alasan utama menggunakan fungsi yang baru untuk klasifikasi adalah, dengan menggunakan *threshold* yang simpel kita akan memiliki problem nanti untuk menggunakan optimisasi, karena fungsi threshold merupakan fungsi yang tidak kontinu, karena di titik mendekati 0.5 kita memiliki dua kondisi hasil klasifikasi sehingga tidak memiliki turunan di titik tersebut. Berbeda dengan fungsi sigmoid, fungsi ini memiliki kontinuitas sehingga bisa menggunakan gradient, seperti di gambar 4.1. Dapat dilihat bahwa fungsi sigmoid merupakan fungsi yang kontinu dan mencakup klasifikasi biner.

Seperti halnya pada regresi linear, proses dalam klasifikasi dengan pada regresi logistik juga menggunakan optimisasi pada *cost function*, sebelumnya kita mendefinisikan *cost function* pada regresi logistik,

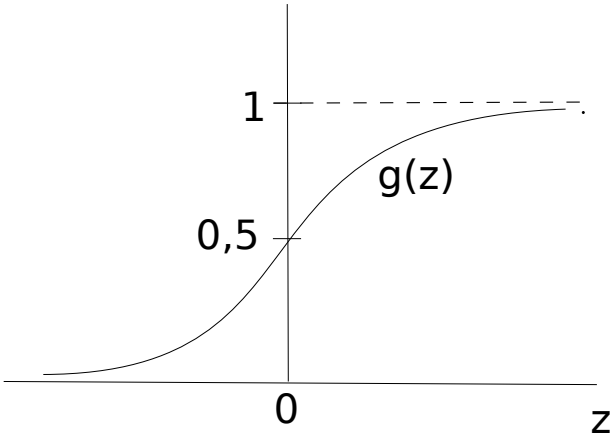

Figure 4.1: Fungsi sigmoid

kita akan melihat *cost function* pada regresi logistik seperti berikut

$$J(\boldsymbol{\theta}) = -\frac{1}{m}\sum_{i=1}^{m} \text{Cost}\left(h_\theta(x^{(i)}), y^{(i)}\right) \tag{4.1}$$

$$= -\frac{1}{m}\sum_{i=1}^{m} \left(y^{(i)} \log h_\theta\left(x^{(i)}\right) + (1-y^{(i)}) \log\left(1 - h_\theta(x^{(i)})\right)\right) \tag{4.2}$$

Secara umum intuisi dari *cost function* dapat dilihat seperti berikut, jika kita benar mengklasifikasikan nilai sebenarnya pada y dengan menggunakan hipotesa atau tebakan kita $h_\theta(x)$ maka *cost function* atau error klasifikasi akan menghasilkan nilai optimal yaitu 0. Secara matematika kita bisa menulis sebagai berikut

- Jika kita memiliki nilai sebenarny $y = 1$ maka kita memiliki *cost function* $= \log h_\theta\left(x^{(i)}\right)$, dan jika kita mengklasifikasikan dengan tepat maka hipotesa kita $h_\theta\left(x^{(i)}\right) = 1$ sehingga *cost function* $=\log 1 = 0$, yang berarti error kita minimal.
- Hal yang sama jika nilai sebenarnya $y = 0$ maka kita memiliki *cost function* $= \log\left(1 - h_\theta(x^{(i)})\right)$. Sehingga jika kita mengklasifikasikan dengan benar maka hipotesa kita $h_\theta(x^{(i)}) = 0$ dan *cost function* kita menjadi $\log 1 = 0$.

Sehingga problem optimisasi pada regresi logistik dapat ditulis seperti berikut

$$\underset{\theta}{\text{minimize}} \quad J(\theta), \tag{4.3}$$

Sama dengan problem regresi linear sebelumnya, kita memiliki metode gradient dengan *update* seperti berikut

$$\theta_j^{t+1} = \theta_j^t - \alpha \frac{\mathrm{d}}{\mathrm{d}\theta_j} J(\theta),$$

untuk $j \in \{1, 2, \ldots, n\}$ dan dimana turunan terhadap *cost function* bergantung pada variabel θ dapat ditulis sebagai berikut

$$\frac{\mathrm{d}}{\mathrm{d}\theta_j} J(\theta) = \frac{1}{m}\sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)}\right) x_j^{(i)}$$

Perlu diingat bahwa kita ingin mencari nilai optimal pada variabel θ yang mana jika kita mempunyai n fitur, kita dapat menulis seperti berikut θ_j untuk $j \in \{1, 2, \ldots, n\}$. Secara utuh metode gradient pada θ ke j dapat ditulis

$$\theta_j^{t+1} = \theta_j^t - \frac{\alpha}{m}\sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)}\right) x_j^{(i)}$$

Sebagai contoh misalnya kita membuat suatu mesin klasifikasi untuk menentukan mahasiswa mana yang lulus dan tidak dari data ujian tengah dan akhir semester seperti di gambar 4.2

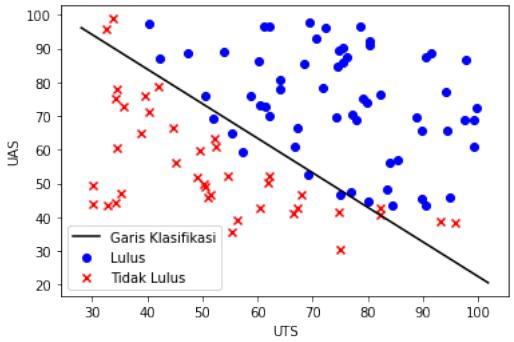

Figure 4.2: Klasifikasi biner

Secara umum visualisasi regresi logistik dapat dilihat di gambar 4.3. Dapat dilihat *forward model* pada struktur ini disebut juga ketika kita menggunakan variabel insial pada fungsi sigmoid, yaitu $\boldsymbol{\theta} \in \mathbb{R}^n$ sehingga mendapatkan suatu klasifikasi inisial $\mathbf{y} \in \mathbb{R}^m$ dan *backward model* merupakan model ketika kita mencoba untuk optimisasi fungsi sigmoid, yaitu mencari nilai $\boldsymbol{\theta}$ yang optimal sehingga nilai prediksi kita dan nilai sebenarnya dari label $\mathbf{y}$ sama.

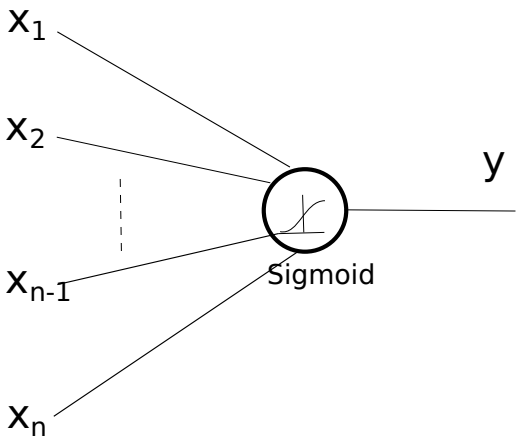

Figure 4.3: Struktur regresi logistik

Di bab berikutnya kita akan melihat struktur yang sama ketika mendefinisikan *artificial neural network* dengan menggunakan berapa input layer.

4.2 Klasifikasi Multikelas: *Satu vs semua*

Pada aplikasi regresi logistik, selain klasifikasi biner kita memiliki beberapa kelas yang mungkin untuk menjadi klasifikasi. Misal daripada mengklasifikasikan bahwa email kita *spam* atau bukan, ada kemungkinan juga untuk mengklasifikasikan suatu email untuk pekerjaan, teman, keluarga, dan grup hobby. Sama seperti halnya dengan cuaca, klasifikasi biner gagal untuk mengklasifikasikan jika kita memiliki beberapa kemungkinan seperti panas, hujan, mendung, atau bahkan salju. Sehingga kita memerlukan klasifikasi multikelas. Seperti yang divisualisasikan oleh gambar 4.4. Perbedaan yang signifikan antara klasifikasi biner dan multikelas adalah, dengan klasifikasi multikelas, kita memiliki banyak kelas tidak hanya 0 dan 1, yaitu misal $0, 1, 2, 3, \ldots$.

Secara prinsip klasifikasi multikelas sama dengan klasifikasi biner, namun di sini hal yang membedakan adalah kita memiliki banyak kelas label. Pendekatan satu vs semua dapat dilihat ketika kita mengoptimasikan variabel $\boldsymbol{\theta}$ pada fungsi hipotesis secara biner, yaitu misal ketika ingin mengoptimisasi kelas bintang, kita beranggapan kelas silang dan kelas kotak menjadi satu kelas. Begitu juga ketika kelas kotak terhadap kelas silang dan bintang, dan yang terakhir kelas silang terhadap kelas bintang dan kotak.

Secara umum kita memiliki kombinasi banyak kelas dengan menggunakan klasifikasi biner kepada semua kelas, dengan filosofi inilah kita juga menyebut sebagai satu vs semua. Langkah klasifikasi multikelas

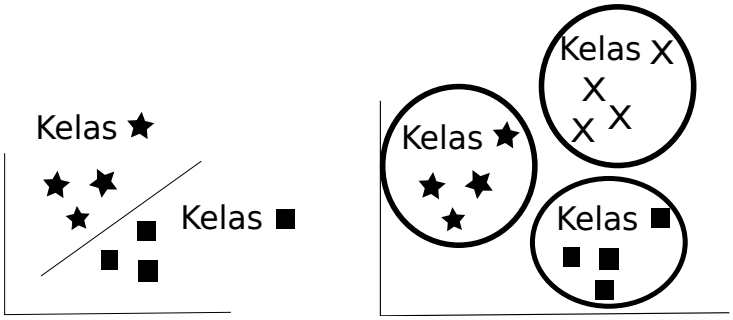

Figure 4.4: Klasifikasi biner dan multikelas

bisa kita rangkum sebagai berikut

- Misalkan kita memiliki k kelas, sehingga kita harus melatih regresi logistik kita, atau mencari nila optimal $\boldsymbol{\theta}$ untuk hipotesis kita ada setiap kelas untuk memprediksi nilai prediksi i sama dengan label kelas y, yaitu $y = i$
- Untuk data yang baru, kita mengklasifikasikan data ini dengan mencari pada kelas yang mana hipotesa kita maksimal

Untuk metode regresi logistik multikelas satu vs semua disini tidak akan diberikan contoh secara jelas karena salah satu aplikasi yang bisa kita gunakan adalah untuk memprediksi tulisan tangan angka dari $0, 1, 2, \ldots, 9$. Sehingga kita memiliki 10 kelas. Problem ini akan kita bahas di bab berikutnya.

4.3 Tugas Regresi Logistik: Klasifikasi Biner

- Kita memiliki fungsi sigmoid seperti berikut

$$h_\theta(x) = \left(\frac{1}{1 + e^{-\boldsymbol{\theta}^T \mathbf{x}}}\right)$$

  Dapatkah saudara menulis turunan dari fungsi sigmoid terhadap θ_j untuk $j \in \{1, 2, 3, \ldots, n\}$?

- Pada bab sebelumnya kita mempunyai problem oprimisasi dari regresi linear dengan menggunakan *least square* atau *mean square error*, yaitu

$$\underset{\theta_0, \theta_1, \ldots, \theta_n}{\text{minimize}} \quad \frac{1}{2m} \sum_{i=1}^{m} \left(h_\theta(x)_i - y_i\right)^2,$$

  Namun untuk regresi logistik kita memiliki

$$\underset{\boldsymbol{\theta}}{\text{minimize}} \quad J(\boldsymbol{\theta}),$$

  dengan *cost function*

$$\begin{aligned} J(\boldsymbol{\theta}) &= -\frac{1}{m} \sum_{i=1}^{m} \text{Cost}\left(h_{\boldsymbol{\theta}}(x^{(i)}), y^{(i)}\right) \\ &= -\frac{1}{m} \sum_{i=1}^{m} \left(y^{(i)} \log h_{\boldsymbol{\theta}}\left(x^{(i)}\right) + (1 - y^{(i)}) \log\left(1 - h_{\boldsymbol{\theta}}(x^{(i)})\right)\right) \end{aligned}$$

  Mengapa kita tidak menggunakan metode yang sama seperti regresi linear? Apa masalahnya ketika kita menggunakan *mean square error* dengan hypothesis menggunakan fungsi sigmoid?

$$h_\theta(x) = g\left(\boldsymbol{\theta}^T \mathbf{x}\right) = \frac{1}{1 + e^{-\boldsymbol{\theta}^T \mathbf{x}}}$$

  Hint: Coba turunkan jika menggunakan *mean-square error* dengan menggunakan fungsi sigmoid!

- Turunan atau gradient dari *cost function* $J(\boldsymbol{\theta})$ adalah

$$\frac{\mathrm{d}}{\mathrm{d}\theta_j} J(\boldsymbol{\theta}) = \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right) x_j^{(i)}$$

  Dapatkah saudara menurunkan persamaan tersebut? Hint: Turunkan *cost function* terhadap θ_j

- Sekarang implementasikan regresi logistik untuk klasifikasi biner seperti gambar 4.2 dataset telah tersedia dengan menggunakan python.

- Setelah mendapatkan nilai optimal untuk $\boldsymbol{\theta}$, gunakan fungsi sigmoid untuk menghitung prediksi, dimana jika nilai sigmoid lebih besar sama dengan 0.5 kita berikan nilai 1, jika sebaliknya kita berikan nilai 0

- Hitung akurasi klasifikasi dari regresi logistik kita, misal nilai label sebenarnya $\mathbf{y} \in \mathbb{R}^m$ dan nilai prediksi kita $\hat{\mathbf{y}} \in \mathbb{R}^m$ dengan persamaan

$$\text{acc} = \left(1 - \frac{1}{m} \sum_{i=1}^{m} (y_i - \hat{y}_i) \right) \times 100$$

4.4 Solusi Tugas Regresi Logistik: Klasifikasi Biner

- Turunan fungsi sigmoid

$$\begin{aligned}
\frac{\mathrm{d}}{\mathrm{d}\theta_j} h_\theta(x) &= \frac{\mathrm{d}}{\mathrm{d}\theta_j} \left(\frac{1}{1 + e^{-\boldsymbol{\theta}^T \mathbf{x}}} \right) \\
&= \left(\frac{e^{-\boldsymbol{\theta}^T \mathbf{x}}}{\left(e^{-\boldsymbol{\theta}^T \mathbf{x}} + 1 \right)^2} \right) x_j \\
&= \frac{1}{e^{-\boldsymbol{\theta}^T \mathbf{x}} + 1} \left(1 - \frac{1}{e^{-\boldsymbol{\theta}^T \mathbf{x}} + 1} \right) x_j \\
&= h_\theta(x) \left(1 - h_\theta(x) \right) x_j
\end{aligned}$$

- Jika kita menggunakan mean squared error, maka gradient yang kita dapatkan seperti berikut

$$\frac{\mathrm{d}}{\mathrm{d}\theta} \frac{1}{2m} \sum_{i=1}^{m} \left(h_\theta(x)_i - y_i \right)^2 = \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(x)_i - y_i \right) \frac{\mathrm{d}}{\mathrm{d}\theta} h_\theta(x)_i$$

  Sekarang kita ingin mencari turunan fungsi sigmoid seperti berikut

$$\begin{aligned}
\frac{\mathrm{d}}{\mathrm{d}\theta_j} h_\theta(x) &= \frac{\mathrm{d}}{\mathrm{d}\theta_j} \left(\frac{1}{1 + e^{-\boldsymbol{\theta}^T \mathbf{x}}} \right) \\
&= \left(\frac{e^{-\boldsymbol{\theta}^T \mathbf{x}}}{\left(e^{-\boldsymbol{\theta}^T \mathbf{x}} + 1 \right)^2} \right) x_j \\
&= \frac{1}{e^{-\boldsymbol{\theta}^T \mathbf{x}} + 1} \left(1 - \frac{1}{e^{-\boldsymbol{\theta}^T \mathbf{x}} + 1} \right) x_j \\
&= h_\theta(x) \left(1 - h_\theta(x) \right) x_j
\end{aligned}$$

  Problem yang ada bahwa fungsi akhirnya adalah tidak konveks. Properti ini tidak akan kita bahas lebih lanjut, namun secara umum dalam teori optimisasi jika problem tidak konveks kita akan memiliki problem untuk mencari nilai optimal. Secara umum syarat untuk melihat konveksitas dari sebuah fungsi dengan menggunakan turunan kedua. Dalam konteks regresi logistik, kita perlu menggunakan *cost function* yang berbeda dari mean square error untuk mendapatkan konveksitas dari *cost function* kita, yaitu dengan cost function yang sudah kita bahas, $J(\theta)$.

- Turunan dari cost function $J(\boldsymbol{\theta})$

$$\frac{\mathrm{d}}{\mathrm{d}\theta_j} J(\boldsymbol{\theta}) = -\frac{1}{m}\sum_{i=1}^{m} \frac{\mathrm{d}}{\mathrm{d}\theta_j}\left(y^{(i)} \log h_\theta\left(x^{(i)}\right) + (1-y^{(i)}) \log\left(1 - h_\theta(x^{(i)})\right)\right)$$

Sebelum menghitung turunan pada fungsi tersebut mari kita lihat dulu beberapa fungsi

$$\log h_\theta\left(x^{(i)}\right) = \log\left(\frac{1}{1+e^{-\boldsymbol{\theta}^T\mathbf{x}^{(i)}}}\right) = -\log\left(1+e^{-\boldsymbol{\theta}^T\mathbf{x}^{(i)}}\right)$$

Kita juga mempunyai

$$\log\left(1-h_\theta\left(x^{(i)}\right)\right) = \log\left(\frac{1+e^{-\boldsymbol{\theta}^T\mathbf{x}^{(i)}}}{1+e^{-\boldsymbol{\theta}^T\mathbf{x}^{(i)}}} - \frac{1}{1+e^{-\boldsymbol{\theta}^T\mathbf{x}^{(i)}}}\right) = -\boldsymbol{\theta}^T\mathbf{x}^{(i)} - \log\left(1+e^{-\boldsymbol{\theta}^T\mathbf{x}^{(i)}}\right)$$

Sehingga kita bisa menulis persamaan *cost function* sebagai berikut

$$\begin{aligned} J(\boldsymbol{\theta}) &= -\frac{1}{m}\sum_{i=1}^{m} -y^{(i)} \log\left(1+e^{-\boldsymbol{\theta}^T\mathbf{x}^{(i)}}\right) + (1-y^{(i)})\left(-\theta^T x^{(i)} - \log\left(1+e^{-\boldsymbol{\theta}^T\mathbf{x}^{(i)}}\right)\right) \\ &= -\frac{1}{m}\sum_{i=1}^{m}\left(y_i\boldsymbol{\theta}^T\mathbf{x}^{(i)} - \boldsymbol{\theta}^T\mathbf{x}^{(i)} - \log\left(1+e^{-\boldsymbol{\theta}^T\mathbf{x}^{(i)}}\right)\right) = -\frac{1}{m}\sum_{i=1}^{m}\left(y_i\boldsymbol{\theta}^T\mathbf{x}^{(i)} - \log\left(1+e^{\boldsymbol{\theta}^T\mathbf{x}^{(i)}}\right)\right), \end{aligned}$$

dimana kita bisa mendapatkan persamaan terakhir dari

$$\boldsymbol{\theta}^T\mathbf{x}^{(i)} - \log\left(1+e^{-\boldsymbol{\theta}^T\mathbf{x}^{(i)}}\right) = -\left(\log e^{\boldsymbol{\theta}^T\mathbf{x}^{(i)}} + \log\left(1+e^{-\boldsymbol{\theta}^T\mathbf{x}^{(i)}}\right)\right) = -\log\left(1+e^{\boldsymbol{\theta}^T\mathbf{x}^{(i)}}\right)$$

Ingat bahwa $\log(x) + \log(y) = \log(xy)$. Sehingga total turunan terhadap *cost function* menjadi

$$\begin{aligned} \frac{\mathrm{d}}{\mathrm{d}\theta_j} J(\boldsymbol{\theta}) &= -\frac{1}{m}\sum_{i=1}^{m} \frac{\mathrm{d}}{\mathrm{d}\theta_j}\left(y^{(i)}\boldsymbol{\theta}^T\mathbf{x}^{(i)} - \log\left(1+e^{\boldsymbol{\theta}^T\mathbf{x}^{(i)}}\right)\right) \\ &== \frac{1}{m}\sum_{i=1}^{m}\left(h_{\boldsymbol{\theta}}(\mathbf{x}^{(i)}) - y^{(i)}\right)x_j^{(i)} \end{aligned}$$

Perlu diingat bahwa $\frac{\mathrm{d}}{\mathrm{d}\theta_j}\log(1+e^{\boldsymbol{\theta}^T\mathbf{x}^{(i)}}) = \frac{\mathbf{x}_j^{(i)}}{1+e^{\boldsymbol{\theta}^T x^{(i)}}} = x_j^{(i)} h_\theta(x^{(i)})$

Chapter 5

Regularisasi

Kali ini kita akan belajar teknik regularisasi pada teori optimisasi untuk mengatasi problem *overfitting*. Seperti yang sudah kita bahas di bab review fundamental, jika problem kita memiliki terlalu banyak fitur maka regresi linear, logistik bahkan *neural network* kita akan sangat bisa mengikuti trend pada data training namun akan gagal untuk memprediksi data baru secara general. Hal ini muncul karena model kita hanya fokus untuk mencari nilai optimal pada data *training* kita dan bukan data secara umum. Gambar 5.1 merupakan contoh kasus dimana regresi linear kita memiliki problem *under* dan

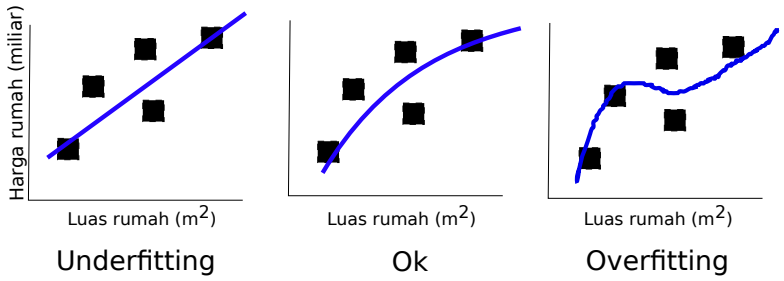

Figure 5.1: Visualisasi problem dalam regresi linear

overfitting. Seperti yang dilihat problem *overfitting* membuat regresi linear sangat mengikuti trend pada data, padahal belum tentu data berikutnya atau data pada umumnya memiliki trend yang sama. Sama halnya dengan regresi logistik yang di perlihatkan pada gambar 5.2, di sini pada problem klasifikasi biner regresi logistik yang kita latih juga cenderung mengikuti spesifik data dan tidak general.

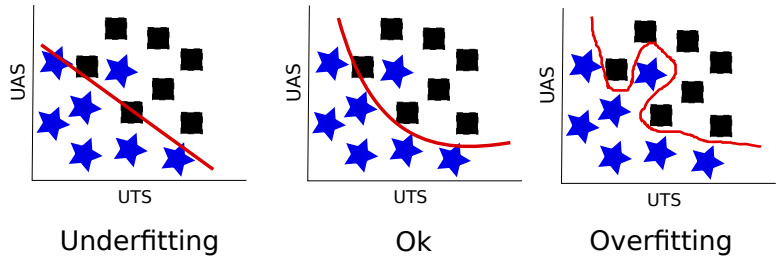

Figure 5.2: Visualisasi problem dalam regresi logistik

5.1 Regularisasi pada regresi linear dan logistik

Ada beberapa cara yang bisa kita lakukan untuk menghindari *overfitting* pada kasus yang memiliki banyak fitur seperti:

- Mengurangi fitur dengan cara memilih secara manual fitur mana yang paling berpengaruh dan menggunakan algoritma seleksi model.

- Regularisasi. Disini kita tidak mengurangi parameter fitur tapi mencoba untuk mengadaptasi variabel optimisasi θ dan sangat memiliki performa yang baik ketika kita memiliki banyak fitur.

Menambahkan regularisasi terjadi pada *cost function* dengan cara menambahkan parameter penalti pada optimisasi kita. Untuk regresi linear kita memiliki optimisasi sebagai berikut

$$\underset{\theta_0,\theta_1,\ldots,\theta_n}{\text{minimize}} \quad \frac{1}{2m}\sum_{i=1}^{m}\left(h_\theta(x)_i - y_i\right)^2 + \frac{\lambda}{m}\sum_{j=1}^{n}\theta_j^2, \tag{5.1}$$

dimana λ merupakan skala penalti yang memberikan pengaruh seberaba besar kita ingin memakai regularisasi. Secara intuisi jika kita ingin mengurangi nilai pada θ kita harus menaruh skala λ dengan nilai yang besar sehingga tetap mendapatkan keseimbangan pada optimisasi. Jika λ sangat kecil atau mendekati 0 maka kita memiliki problem optimisasi awal. Sehingga dalam menggunakan regularisasi kita harus bisa melihat skala λ yang menghasilkan nilai optimal. Sama halnya dengan regularisasi pada regresi logistik kita akan menambahkan parameter pada optimisasi kita. Di mana kita memiliki *cost function*

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}\left(y^{(i)}\log h_\theta\left(x^{(i)}\right) + (1-y^{(i)})\log\left(1-h_\theta(x^{(i)})\right)\right) + \lambda\sum_{j=1}^{m}\theta_j^2 \tag{5.2}$$

Sehingga problem oprimisasi kita menjadi

$$\underset{\theta}{\text{minimize}} \quad J(\theta) \tag{5.3}$$

Perlu diingat disini sebelum kita masuk ke fungsi sigmoid kita selalu menambahkan unit bias pada parameter variabel kita, yaitu θ_0. Namun disini kita tidak meregulasikan parameter ini karena parameter ini tidak masuk dalam fitur kita. Sehingga jika kita lihat parameter regularisasi dimulai dari index $j = 1$. Secara umum update menggunakan metode gradient pada regresi logistik dapat ditulis sebagai berikut

$$\theta_0 = \theta_0 - \frac{\alpha}{m}\sum_{i=1}^{m}\left(h_\theta(x^{(i)} - y^{(i)}\right)x_0^{(i)}$$

Persamaan di atas tidak memiliki regularisasi pada unit bias sehingga kita memiliki persamaan yang sama tanpa regularisasi. Namun ketika parameter untuk fitur berikutnya untuk index $j \neq 0$ kita harus memasukkan parameter regularisasi

$$\theta_j = \theta_j - \alpha\left(\frac{1}{m}\sum_{i=1}^{m}\left(h_\theta(x^{(i)} - y^{(i)}\right)x_j^{(i)} + \frac{\lambda}{m}\theta_j^2\right)$$

Sebagai contoh kita bisa lihat pada gambar 5.3, dimana kita memiliki klasifikasi biner namun dengan pesebaran data yang ada kita tidak bisa melakukan klasifikasi dengan hanya menggunakan persamaan garis.

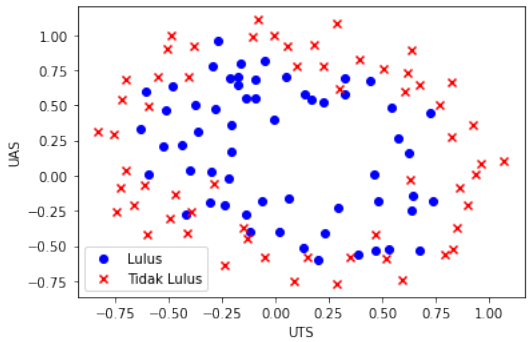

Figure 5.3: Visualisasi problem dalam regresi logistik

Untuk mengklasifikasi data ini, kita harus mengunakan polynomial fitting, namun akan terjadi overfitting ketika kita menggunakan variabel $\lambda = 100$ seperti yang terlihat di gambar 5.4. Berbeda dengan

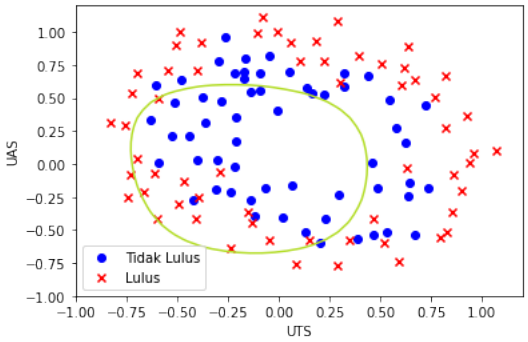

Figure 5.4: Visualisasi problem dalam regresi logistik ($\lambda = 100$)

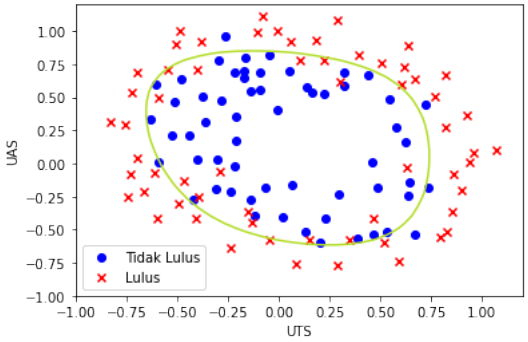

Figure 5.5: Visualisasi problem dalam regresi logistik ($\lambda = 1$)

$\lambda = 100$ kita cukup mendapatkan klasifikasi yang general jika menggunakan parameter $\lambda = 1$ seperti yang diperlihatkan pada gambar 5.5

Dalam teori optimisasi ada berapa teknik regularisasi yang sering dilakukan, apa yang kita lakukan dengan menambahkan persamaan kuadrat disebut juga Tikhonov regularisasi [12] atau ridge regression. Beberapa regularisasi yang lain seperti Lasso [13] menggunakan parameter yang berbeda. Dalam teori optimisasi beberapa aspek yang menarik untuk diperhatikan ketika menggunakan regularisasi, misalnya bagaimana menentukan parameter λ secara optimal tanpa menggunakan metode uji coba dan empirik atau pada saat kapan regularisasi memiliki garansi menghasilkan nilai yang optimal. Namun hal tersebut melampaui konten yang akan kita bahas di tema kali ini.

5.2 Tugas Regularisasi

- Pada tugas kali ini anda diminta untuk mengimplementasikan regresi logistik pada data di gambar 5.3.
- Bisakah saudara mereplikasi garis klasifikasi seperti pada gambar 5.5 dan gambar 5.4?
- Hitung akurasi pada regresi logistik yang saudara implementasikan!

Chapter 6

Neural Network

Pada bagian sebelumnya kita sudah belajar dalam proses prediksi dan klasifikasi menggunakan regresi linear dan regresi logistik. Pada bab kali ini kita akan melihat bahwa struktur dari *artificial neural network* dapat dilihat dari perspektif regresi logistik atau kumpulan dari beberapa layer regresi logistik. Sebelum kita masuk ke dalam *artificiai neural network* ada baiknya kita melihat struktur *brain neural* atau syaraf dalam otak yang menjadi inspirasi asal dari *artificial neural network* [14, 15]. Meskipun sampai saat ini masih banyak diskusi apakah model *artificial neural network* sama seperti fungsi otak karena kenyataannya fungsi otak sangat kompleks dan mungkin tidak bisa dimodelkan secara simpel.

6.1 Otak dan Syaraf

Struktur jaringan otak dalam tubuh manusia terdiri dari beberapa bagian seperti di gambar 6.1. Secara empirik perkembangan otak manusia dan bagaimana kognitif manusia bekerja untuk mempelajari sesuatu merupakan suatu hal yang sangat menarik untuk dipelajari. Ada bukti kuat bahwa otak manusia memiliki mekanisme sendiri dalam proses pembelajaran.

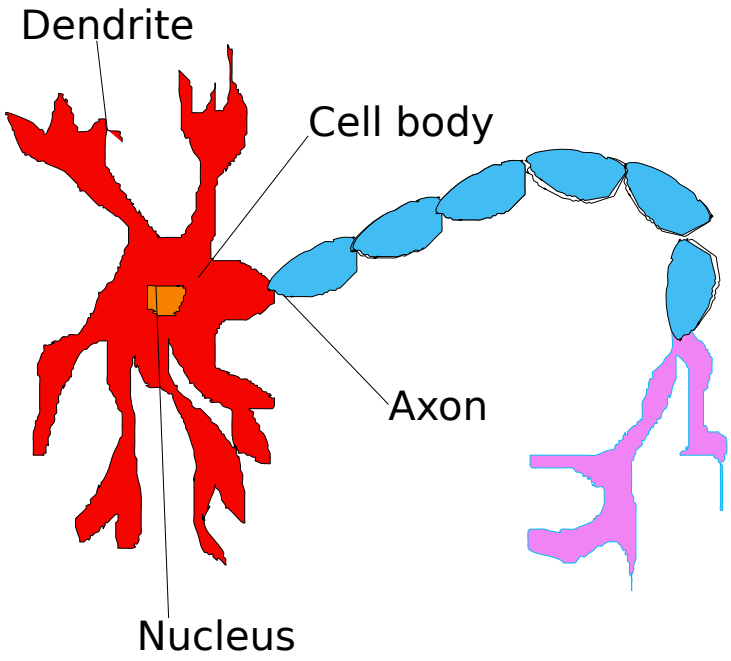

Figure 6.1: Jaringan *neural* pada otak

Dalam simpel model *neuron* merupakan unit komputasi yang menggunakan *dendrites* sebagai input elektrik, disebut juga *spikes*, yang menghasilkan keluaran di sisi *axons*. Sehingga secara tidak langsung model pada jaringan otak ini bisa digunakan dalam merepresentasikan *artificial neural network*. Mari kita lihat kembali model simpel dari regresi logistik.

6.2 Artificial Neural Network

Seperti yang telah kita pelajari pada bab sebelumnya kita sudah melihat visualisasi pada unit logistik yang dijelaskan di gambar 6.2. Dalam model ini kita memiliki fungsi aktivasi yang disebut sebagai fungsi sigmoid atau fungsi logistik.

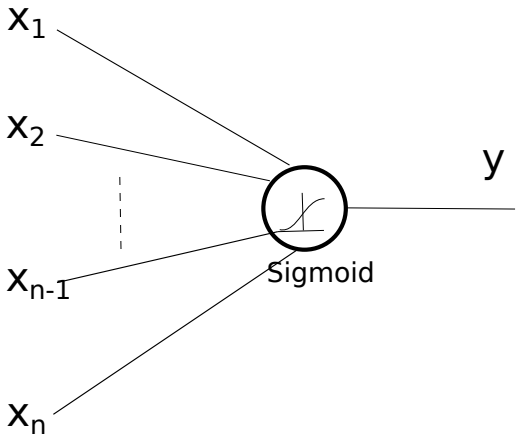

Figure 6.2: Regresi logistik

Sekarang jika kita menambah bagian dalam regresi logistik ini maka kita akan mendapatkan dua layer dari regresi logistik. Model ini juga disebut sebagai *artificial neural network*, seperti yang dijelaskan gambar 6.3. Di sini layer pertama disebut juga sebagai input layer, karena di layer ini kita menggunakan input data sebagai masukan dari *neural network* kita. Input data bisa berupa gambar, suara ataupun video yang direpresentasikan sebagai vektor ataupun matriks. Di layer ke dua kita memiliki *hidden layer* atau disebut juga sebagai *shallow layer*, dimana kita memliki variabel hasil regresi logistik dari layer sebelumnya. Sama seperti contoh di bab sebelumnya dimana kita harus mencari nilai optimal pada variabel hipotesis, dalam neural network kita harus melakukan proses optimisasi. Di layer ke tiga kita memiliki *output layer* yang merupakan akhir dari hipotesa kita. Tiap layer dalam model ini dapat memiliki banyak neuron yang berbeda.

Secara umum proses untuk menentukan nilai hipotesa dari input sampai ke output layer merupakan proses *forward model* atau sering juga disebut sebagai *feed forward model*. Sedangkan proses untuk menentukan nilai optimal dalam variabel hipotesis di tiap layer disebut juga proses *backward model* dimana kita memiliki problem optimisasi.

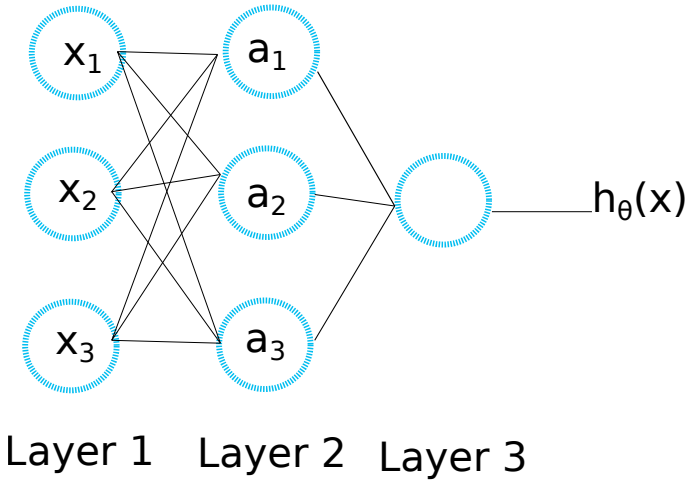

Figure 6.3: *Artificial Neural Network*

Dalam konteks modern *machine learning* jika kita memiliki banyaknya hidden layer layer lebih dari tiga layer maka disebut juga sebagai *deep neural network*. Istilah ini yang sekarang sering muncul atau disebut juga *deep learning*. Pada awal perkembangan komputer, keterbatasan komputasi pada komputer dan sedikitnya data membuat implementasi banyak layer terbatas dikarenakan banyaknya waktu yang

dibutuhkan untuk memproses semua layer. Namun sekarang hal tersebut sudah bisa diatasi dengan kemampuan komputer yang sangat maju dan banyaknya data yang manusia hasilkan untuk melatih *artificial neural network* menjadi lebih baik. Pada bagian berikutnya kita akan mempelajari lebih dalam proses *forward* dan *backward* model.

6.2.1 Forward Model

Sekarang kita akan fokus pada forward model dan menurunkan persamaan pada tiap layer sampai kita memiliki hipotesis. Misalkan kita memiliki input data *single training* secara vektor $\mathbf{x} \in \mathbb{R}^N$. Jika input *network* kita merupakan gambar dengan dimensi $K \times K$, kita bisa vektorisasi gambar tersebut dengan menyusun vektor input dengan dimensi $N = K^2$. Hal ini dapat dicapai dengan menyusun tiap kolom matriks yang merepresentasikan gambar untuk menjadi vektor seperti yang dijelaskan pada gambar 6.4

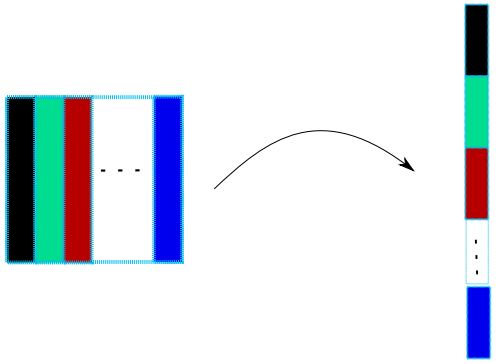

Figure 6.4: Vektorisasi sebuah matriks

Nilai pada layer kedua dapat ditulis seperti keluaran produk matriks dan vektor dan hasil dari fungsi sigmoid. Asumsikan kita memiliki N input vektor dan pada layer berikutnya kita memiliki sebanyak n neuron, seperti yang dijelaskan sebagai berikut.

$$\begin{pmatrix} z_1 \\ z_2 \\ \vdots \\ z_n \end{pmatrix} = \begin{pmatrix} \theta_{10} & \theta_{11} & \theta_{12} & \dots & \theta_{1N} \\ \theta_{20} & \theta_{21} & \theta_{22} & \dots & \theta_{2N} \\ & \vdots & \vdots & \dots & \vdots \\ \theta_{n0} & \theta_{n1} & \theta_{n2} & \dots & \theta_{nN} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ x_N \end{pmatrix}$$

Perlu diingat disini dalam *artificial neural network* biasanya kita menambahkan unit bias, misal variabel θ_0 pada regresi linear. Sehingga dalam kasus ini kita juga menambahkan bagian seperti di warna merah. Sehingga untuk tiap layer dengan n neuron dan N input dari layer sebelumnya, kita memiliki matriks variabel hipotesis θ dengan dimensi $n \times N + 1$. Di sisi lain, jika kita mempunyai m training data maka kita secara total akan memiliki matriks input dengan dimensi $N + 1 \times m$ dan output $n \times m$.

Selanjutnya hasil produk matriks dan vektor ini kita masukkan kepada fungsi sigmoid $g(z)$ sehingga untuk nilai masukan pada layer berikutnya kita memiliki

$$\begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{pmatrix} = \begin{pmatrix} g(z_1) \\ g(z_2) \\ \vdots \\ g(z_n) \end{pmatrix},$$

dimana fungsi sigmuid seperti di bab sebelumnya

$$g\left(z\right) = \frac{1}{1 + e^{-z}}$$

Berikutnya variabel $a_1, a_2, \dots, a_n$ menjadi masukan untuk layer berikutnya ditambah dengan unit bias, sehingga kita memiliki dimensi $m + 1$. Jika kita sampai pada layer terakhir, asumsikan kita hanya memiliki tiga layer, seperti yang divisualisasikan di gambar 6.3, kita memiliki nilai output

$$h_\theta(x) = g(\theta_0 a_0 + \theta_1 a_1 + \ldots + \theta_n a_n)$$

Dengan kata lain $h_\theta(x)$ merupakan output dari layer kita dan prediksi dari *artificial neural network* kita. Pada problem klasifikasi multikelas, dimensi pada prediksi kita memiliki C kelas dengan total m training data. Pada iterasi awal sangat mungkin prediksi kita salah, oleh karena itu kita harus melakukan proses optimisasi untuk mencari nilai optimal dari semua variabel θ di tiap layer. Proses ini disebut juga *backward model* dengan mendefinisikan *cost function*.

6.2.2 Backward Model

Bagian ini merupakan hal yang krusial dalam pembelajaran atau melatih *neural network* kita. Sama seperti bab sebelumnya, kita juga harus mendefinisikan *cost function* agar kita mendapatkan nilai optimal untuk problem yang menggunakan *neural network*. Sebelum kita masuk lebih lanjut, mari kita mendefinisikan variabel L sebagai banyaknya layer di dalam network kita dan untuk problem klasifikasi kita menggunakan variabel C kelas. Jika kita menggunakan klasifikasi biner maka secara tidak langsung kita mempunyai $C = 2$. Banyaknya dataset yang kita gunakan untuk melatih *neural network* kita definisikan dengan variabel m, dimana untuk tiap data $\mathbf{x}^{(i)} \in \mathbb{R}^N$ misalnya kita memiliki vektorisasi dari gambar dengan dimensi matriks $K \times K$ atau $N = K^2$.

Secara umum *cost function* pada *neural network* dapat kita tulis dengan menggunakan *cost function* dari regresi logistik. Misalkan kita mempunyai label atau nilai sebenarnya dari klasifikasi kita $\mathbf{y} \in \mathbb{R}^C$ dan prediksi dari *neural network* kita $h_\theta(\mathbf{x}) \in \mathbb{R}^C$, kita dapat menulis *cost function* seperti berikut

$$J(\theta) = -\frac{1}{m}\left(\sum_{i=1}^{m}\sum_{c=1}^{C} y_c^{(i)} \log\left(h_\theta\left(\mathbf{x}^{(i)}\right)\right)_c + \left(1 - y_c^{(i)}\right)\log\left(1 - h_\theta\left(\mathbf{x}^{(i)}\right)\right)_c\right) + \frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{n_l}\sum_{j=1}^{n_{l+1}}\left(\theta_{ji}^{(l)}\right)^2.$$

Persamaan *cost function* diatas merupakan persamaan ketika kita mengevaluasi secara element dari vektor atau matriks, dengan menggunakan prinsip produk antara dua matriks seperti yang kita pelajari di bab fundamental kita bisa memodelkan persamaan tersebut lebih efisien.

Secara umum untuk proses *backward model* kita ingin meminimalkan *cost function* dan mencari variabel θ yang optimal. Mari kita ulang kembali proses *forward model* untuk $L = 4$ layer seperti di gambar 6.5 Kita bisa menulis proses *forward model* untuk semua training data selangkah demi selangkah

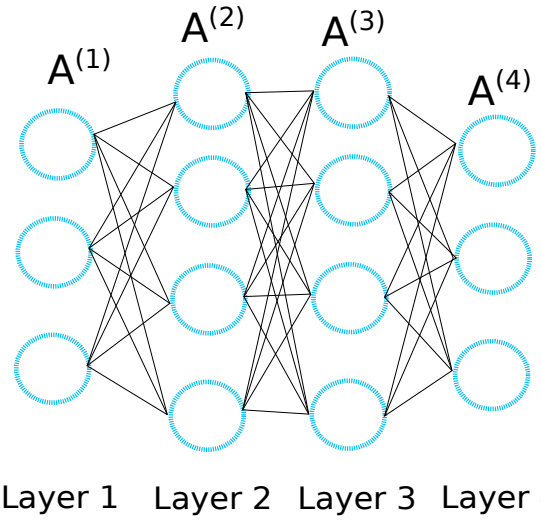

Figure 6.5: *Artificial neural network* dengan 4 layer

sebagai berikut. Berbeda pada *forward model* di sini kita akan menggunakan m training data.

- Untuk layer pertama kita memiliki input data sebanyak m training data sehingga

$$\mathbf{A}^{(1)} = \mathbf{X} \in \mathbb{R}^{N \times m}$$

- Sebagai masukan pada layer kedua, kita harus menggunakan fungsi sigmoid sehingga masukan pada fungsi sigmoid dapat ditulis

$$\mathbf{Z}^{(2)} = \mathbf{\Theta}^{(1)}\mathbf{A}^{(1)},$$

dimana dimensi $\mathbf{\Theta}^{(1)} \in \mathbb{R}^{n_2 \times N+1}$. Perlu diingat di baris pertama pada input data kita tambahkan unit bias sehingga kita memiliki dimensi $n_2 \times N + 1$. Secara praktek penambahan biasa dilakukan dengan menambahkan vektor dengan semua elemen 1 di baris pertama matriks. Layer kedua kita memiliki

$$\mathbf{A}^{(2)} = g(\mathbf{Z}^{(2)}) \in \mathbb{R}^{n_2 \times m}$$

Perlu diingat disini kita menambahkan unit bias sehingga dimensi $\mathbf{A}^{(2)} \in \mathbb{R}^{n_2+1 \times m}$

- Pada layer ketiga kita memiliki

$$\mathbf{Z}^{(3)} = \mathbf{\Theta}^{(2)}\mathbf{A}^{(2)},$$

dimana dimensi $\boldsymbol{\theta}^{(2)} \in \mathbb{R}^{n_3 \times n_2+1}$. Layer ketiga kita memiliki

$$\mathbf{A}^{(3)} = g(\mathbf{Z}^{(3)}) \in \mathbb{R}^{n_3 \times m}$$

Perlu diingat disini kita menambahkan unit bias sehingga dimensi $\mathbf{A}^{(3)} \in \mathbb{R}^{n_3+1 \times m}$

- Pada layer keempat atau layer terakhir, kita memiliki

$$\mathbf{Z}^{(4)} = \mathbf{\Theta}^{(3)}\mathbf{A}^{(3)},$$

dimana dimensi $\mathbf{\Theta}^{(3)} \in \mathbb{R}^{C \times n_3+1}$. Layer keempat kita atau layer keluaran kita memiliki

$$\mathbf{A}^{(4)} = h_\Theta(\mathbf{X}) = g(\mathbf{Z}^{(4)}) \in \mathbb{R}^{C \times m}$$

Dimana layer ini merupakan layer hipotesis kita dengan multikelas C dan total data untuk melatih *artificial neural network* kita sebanyak m.

Setelah kita memiliki hipotesis, kita ingin mengukur sejauh apa prediksi atau klasifikasi kita dengan nilai aslinya yang kita dapatkan dari data label atau disebut juga *label data*. Oleh karena itu kita akan melakukan proses optimisasi pada *cost function*. Tantangan pada proses ini adalah kita mundur kebbelakang pada tiap layer atau disebut juga *backward propagation*, sehingga turunan fungsi yang bergantung pada variabel θ yang ingin kita optimisasi pada tiap layer bergantung satu dengan yang lain. Namun, karena sifat pada turunan atau *chain rule* kita bisa menurunkan persamaan ini seperti berikut dengan asumsi tanpa regularisasi $\lambda = 0$

- Misalkan kita menulis variabel $\boldsymbol{\epsilon}^{(l)}$ sebagai error pada layer l. Sehingga pertama-tama kita bisa menghitung error prediksi atau klasifikasi kita

- Error hipotesis kita pada layer keempat adalah

$$\boldsymbol{\epsilon}^{(4)} = \mathbf{A}^{(4)} - \mathbf{Y} \in \mathbb{R}^{C \times m},$$

dimana matriks $\mathbf{Y}$ merupakan training label untuk m training data.

- Untuk menghitung error pada layer ketiga kita menggunakan error pada layer keempat, variabel inisial $\mathbf{\Theta}^{(3)} \in \mathbb{R}^{C \times n_3+1}$ pada layer ketiga dan turunan fungsi sigmoid kita

$$\boldsymbol{\epsilon}^{(3)} = \left(\mathbf{\Theta}^{(3)}\right)^T \boldsymbol{\epsilon}^{(4)} \circ \frac{\mathrm{d}}{\mathrm{d}\mathbf{Z}} g(\mathbf{Z}^{(3)})$$

Perlu diingat bahwa di sini dimensi $g(\mathbf{Z}^{(3)}) \in \mathbb{R}^{n_3 \times m}$ seperti yang sudah kita turunkan di *forward model* namun $\left(\mathbf{\Theta}^{(3)}\right)^T \boldsymbol{\epsilon}^{(4)}$ memiliki dimensi $\mathbb{R}^{n_3+1 \times m}$ karena kita menambahkan unit bias. Seperti yang sudah kita pelajari produk elemen $\circ$ harus memiliki dimensi yang sama, sehingga kita harus menghilangkan kolom pertama pada matriks $\mathbf{\Theta}^{(3)}$, yaitu unit bias, sehingga kita memiliki matriks baru $\hat{\mathbf{\Theta}}^{(3)} \in \mathbb{R}^{C \times n_3}$. Maka kita memiliki variabel baru

$$\boldsymbol{\epsilon}^{(3)} = \left(\hat{\mathbf{\Theta}}^{(3)}\right)^T \boldsymbol{\epsilon}^{(4)} \circ \frac{\mathrm{d}}{\mathrm{d}\mathbf{Z}} g(\mathbf{Z}^{(3)}) \in \mathbb{R}^{n_3 \times m}$$

- Sama halnya dengan layer kedua kita memiliki persamaan

$$\boldsymbol{\epsilon}^{(2)} = \left(\hat{\mathbf{\Theta}}^{(2)}\right)^T \boldsymbol{\epsilon}^{(3)} \circ \frac{\mathrm{d}}{\mathrm{d}\mathbf{Z}} g(\mathbf{Z}^{(2)}) \in \mathbb{R}^{n_2 \times m},$$

dimana $\hat{\mathbf{\Theta}}^{(2)} \in \mathbb{R}^{n_3 \times n_2}$.

- Sehingga persamaan total dari *backpropagation model* untuk menghitung gradient total dapat dirumuskan seperti berikut

$$\frac{\mathrm{d}}{\mathrm{d}\boldsymbol{\Theta}^{(3)}} J(\boldsymbol{\Theta}) = \frac{1}{m} \boldsymbol{\epsilon}^{(4)} \left(\mathbf{A}^{(3)}\right)^T \in \mathbb{R}^{C \times n_3 + 1}$$
$$\frac{\mathrm{d}}{\mathrm{d}\boldsymbol{\Theta}^{(2)}} J(\boldsymbol{\Theta}) = \frac{1}{m} \boldsymbol{\epsilon}^{(3)} \left(\mathbf{A}^{(2)}\right)^T \in \mathbb{R}^{n_3 \times n_2 + 1}$$
$$\frac{\mathrm{d}}{\mathrm{d}\boldsymbol{\Theta}^{(1)}} J(\boldsymbol{\Theta}) = \frac{1}{m} \boldsymbol{\epsilon}^{(2)} \left(\mathbf{A}^{(1)}\right)^T \in \mathbb{R}^{n_2 \times N + 1}$$

- Jika kita menggunakan regularisasi maka kita harus menambahkan parameter lain untuk gradient. Perlu diingat kita memiliki dimensi dengan menambahkan unit bias, sehingga dimensi selalu bertambah 1. Dalam regularisasi seperti yang kita bahas, kita tidak meregularisasi unit bias sehingga kita memiliki dimensi baru, seperti yang ditulis berikut

$$\frac{\mathrm{d}}{\mathrm{d}\boldsymbol{\Theta}^{(3)}} J(\boldsymbol{\Theta}) = \frac{1}{m} \boldsymbol{\epsilon}^{(4)} \left(\hat{\mathbf{A}}^{(3)}\right)^T + \frac{\lambda}{m} \hat{\boldsymbol{\Theta}}^{(3)} \in \mathbb{R}^{C \times n_3}$$
$$\frac{\mathrm{d}}{\mathrm{d}\boldsymbol{\Theta}^{(2)}} J(\boldsymbol{\Theta}) = \frac{1}{m} \boldsymbol{\epsilon}^{(3)} \left(\hat{\mathbf{A}}^{(2)}\right)^T + \frac{\lambda}{m} \hat{\boldsymbol{\Theta}}^{(2)} \in \mathbb{R}^{n_3 \times n_2}$$
$$\frac{\mathrm{d}}{\mathrm{d}\boldsymbol{\Theta}^{(1)}} J(\boldsymbol{\Theta}) = \frac{1}{m} \boldsymbol{\epsilon}^{(2)} \left(\hat{\mathbf{A}}^{(1)}\right)^T + \frac{\lambda}{m} \hat{\boldsymbol{\Theta}}^{(1)} \in \mathbb{R}^{n_2 \times N}$$

  Perlu diingat kita akan tetap menggunakan unit bias pada gradient sehingga kita harus menambahkan variabel unit bias di awal di gradient total, dengan cara mengambil element pertama pada gradient awal tanpa regularisasi dan menaruhnya pada kolom pertama, sehingga total gradient tetap memiliki dimensi sebagai berikut

$$\frac{\mathrm{d}}{\mathrm{d}\boldsymbol{\Theta}^{(3)}} J(\boldsymbol{\Theta}) \in \mathbb{R}^{C \times n_3 + 1}$$
$$\frac{\mathrm{d}}{\mathrm{d}\boldsymbol{\Theta}^{(2)}} J(\boldsymbol{\Theta}) \in \mathbb{R}^{n_3 \times n_2 + 1}$$
$$\frac{\mathrm{d}}{\mathrm{d}\boldsymbol{\Theta}^{(1)}} J(\boldsymbol{\Theta}) \in \mathbb{R}^{n_2 \times N + 1}$$

Setelah mendapatkan gradient total, maka kita bisa menggunakan metode gradient untuk mencari nilai optimal pada tiap layer untuk meminimalkan estimasi kita.

6.3 Pendekatan praktikal

Secara umum dalam melakukan setting parameter di *artificial neural network* atau *machine learning* untuk supervised model, yaitu jika kita memiliki training data dan nilai sebenarnya dari data atau label, kita tidak menggunakan seluruh dataset untuk melatih atau training *neural network* kita. Ada beberapa

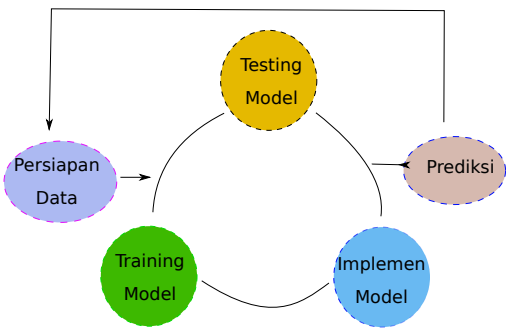

Figure 6.6: Workflow design dalam implementasi *artifiial neural network*

aspek yang harus diperhatikan misalnya kita harus membagi persentase dataset yang kita gunakan. Misalkan kita memiliki total dataset sebanyak m. Dalam praktiknya kita menggunakan sekitar $0.8 \times m$ dataset untuk melakukan training dan $0.2 \times m$ untuk melakukan testing. Dalam prakteknya hal ini berguna untuk melihat akurasi *neural network* dalam melakukan tugas pada training dan test dataset serta melihat apakah kita memiliki performa yang sama pada kedua dataset.

Selain pada pembagian training dan test dataset kita juga perlu melihat parameter pada regularisasi apakah kita mendapatkan parameter yang optimal. Sehingga secara umum pendekatan praktikal dan *workflow design* pada *machine learning* atau *artificial neural network* sebenarnya merupakan problem empirik dan perlu *workflow* yang harus dilakukan dari persiapan data, implementasi model melalui training dan test data, sampai ke prediksi atau klasifikasi, serta kembali lagi ke persiapan data seperti yang digambarkan di gambar 6.6. Di bab berikutnya kita akan melakukan implementasi dan design *regresi logistik* multikelas dan *artificial neural network* untuk mengklasifikasikan tulisan tangan.

Salah satu tips yang penting ketika melakukan implementasi pada python adalah perlunya melihat dimensi data dan *network* kita pada tiap layar sehingga bisa melihat error yang ada. Oleh sebab itu pengetahuan fundamental matriks dan vektor akan sangat berguna ketika kita *debugging* implementasi python kita.

Chapter 7

Convolutional Neural Networks

Seperti yang sudah kita bahas sebelumnya dalam *artificial neural networks* kita menggunakan operasi perkalian matriks dan vektor dalam menghitung relasi antara masukan dan keluaran pada tiap layer ditambah dengan fungsi sigmoid. Berbeda pada implementasi dalam *artificial neural networks* pada *convolutional neural networks* kita menggunakan operasi *convolution* daripada operasi perkalian matriks dan vektor. Keuntungan menggunakan operasi *convolution* adalah kita akan bisa melihat fitur dari gambar seperti *edge*, misal transisi dari area gambar manusia dengan latar belakangnya, dengan jelas sehingga membantu *networks* dalam melakukan tugas klasifikasi atau deteksi.

7.1 *Stride* dan *Padding*

Seperti yang sudah kita bahas pada bab review fundamental, secara umum *convolution* merupakan proses perkalian dan penjumlahan bergantung pada dimensi kernel filter dan data. Namun pada definisi yang kita gunakan kita mengasumsikan kernel filter bergerak pada data per pixel seperti yang di visualisasikan di gambar 7.1.

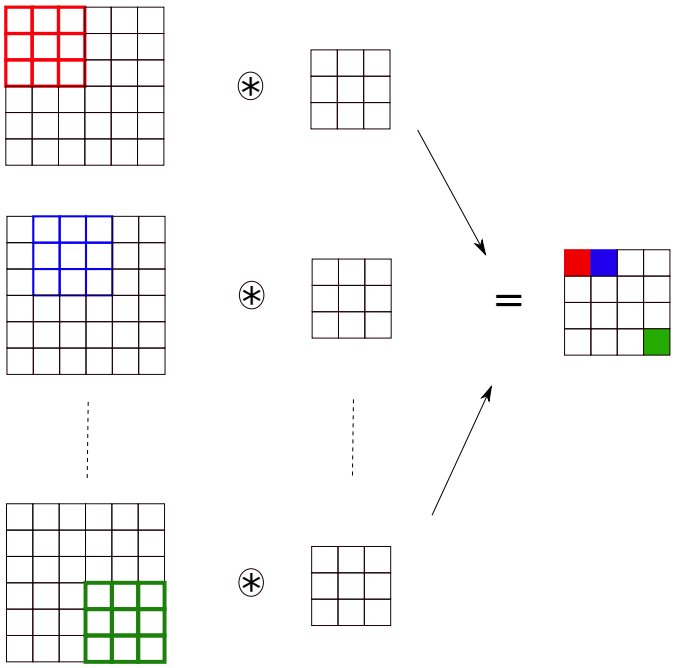

Figure 7.1: Operasi normal *convolution* tanpa *padding* dan *stride*

Dapat dilihat disini bahwa elemen pada hasil akhir matriks merupakan perkalian elemen antara bagian data kita dengan kernel filter dan penjumlahan dari elemen perkalian tersebut. Problem pada *convolution* seperti gambar yang sudah kita bahas adalah hasil dari operasi *convolution* memiliki dimensi yang berbeda dengan data kita. Oleh karena itu agar kita memiliki dimensi yang sama dengan data awal,

kita menggunakan tambahan nilai nol pada data kita yang disebut juga sebagai *padding* seperti yang diperlihatkan di gambar 7.2

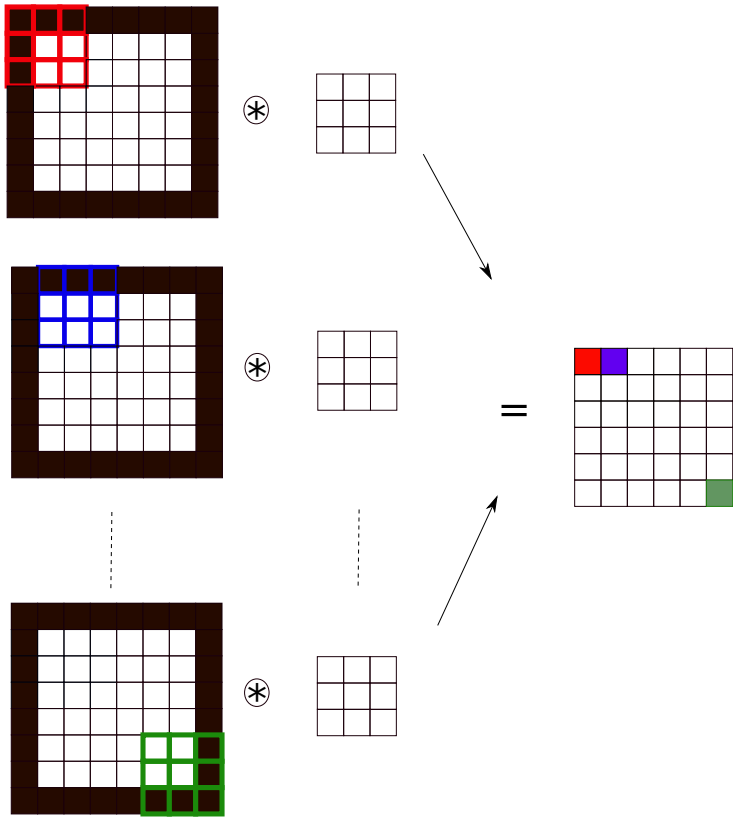

Figure 7.2: Operasi *convolution* *padding* = 1 tanpa *stride*

Pada kenyataannya kita juga bisa menggunakan pergeseran per dua element pada matriks atau lebih. Pergeseran ini disebut disebut *stride* dan divisualisasikan pada gambar 7.3

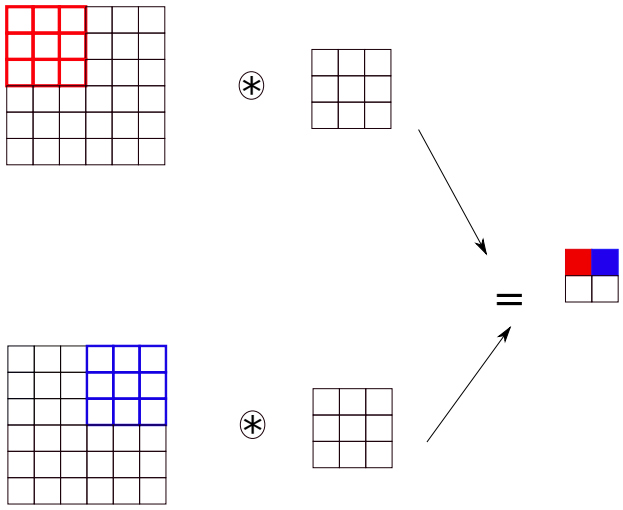

Figure 7.3: Operasi *convolution* dengan *stride* = 3 tanpa *padding*

Secara umum dimensi hasil dari operasi *convolution* jika memiliki variabel *padding* p dan *stride* s pada data $\mathbf{A} \in \mathbb{R}^{n \times n}$ dan kernel filter $\mathbf{F} \in \mathbb{R}^{f \times f}$ dapat ditulis sebagai berikut

$$\mathbf{R} = \mathbf{A} \circledast \mathbf{F}$$

dengan $\mathbf{R} \in \mathbb{R}^{r \times r}$ dan dimensi $r = \lfloor \frac{n+2p-f}{s} + 1 \rfloor$ disini $\lfloor . \rfloor$ merupakan pembulatan kebawah untuk menjadikan integer.

7.2 Satu layer *convolutional neural networks*

Pada umumnya masukan data yang kita gunakan untuk melatih *networks* kita bisa berupa gambar dengan format *RGB*, sehingga kita tidak hanya memiliki matriks namun juga matriks dengan kanal bernilai tiga. Secara umum kita bisa generalisasi nilai kanal pada input kita. Konsekuensi pada informasi ini adalah kita juga memiliki kernel filter dengan banyak channel seperti yang diperlihatkan pada gambar 7.4.

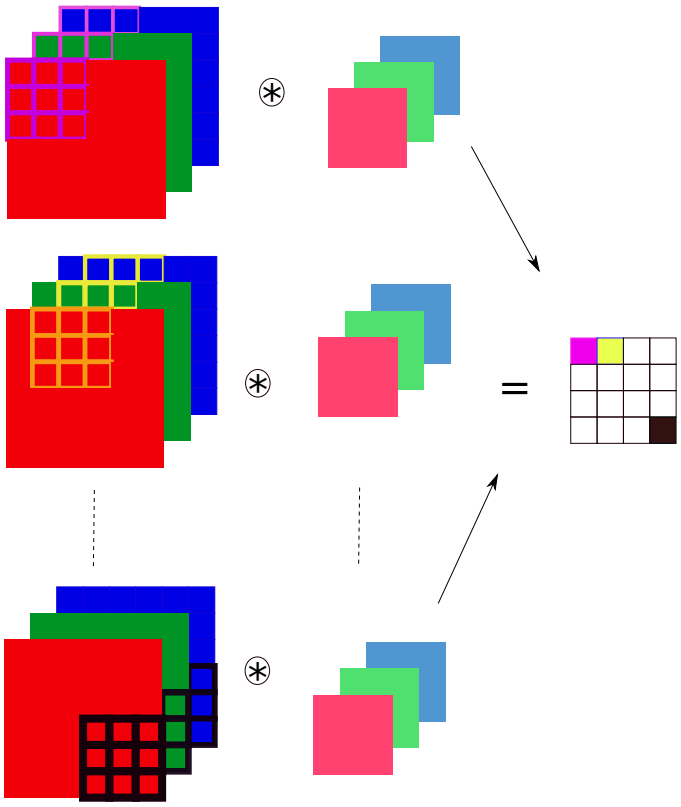

Figure 7.4: Operasi *convolution* dengan 3 kanal dan kernel filter tanpa *stride* dan *padding*

Hasil akhir pada operasi ini adalah selain kita melakukan perkalian dan penjumlahan pada tiap kanal kita juga menjumlahkan semua kanal untuk mendapatkan nilai pada elemen matriks pada hasil akhir. Konsep ini berguna untuk mendefinisikan *convolutional neural networks* dengan menggunakan banyak filter seperti yang divisualisasikan pada gambar 7.5

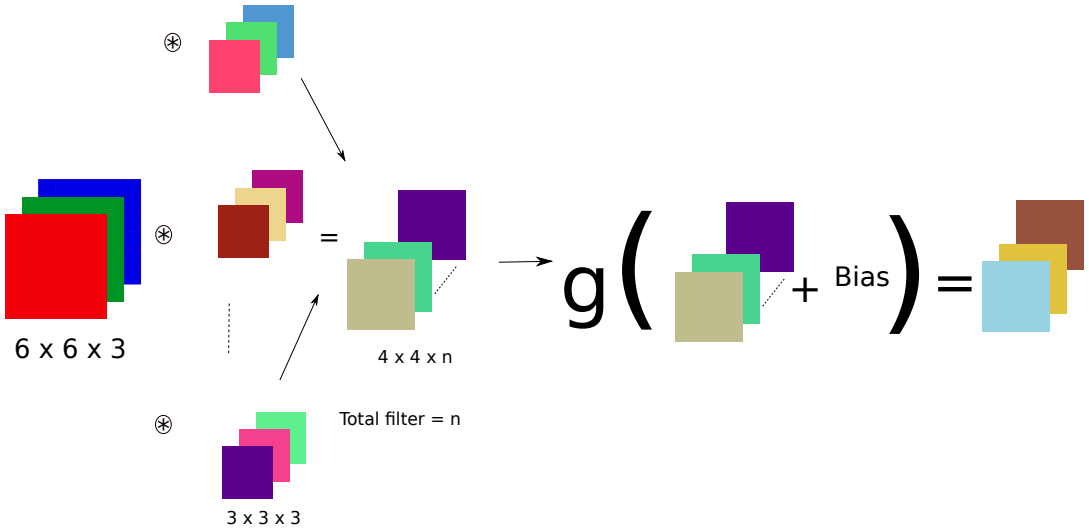

Figure 7.5: Operasi satu layer *convolutional neural networks*

Disini kita memiliki data input dengan dimensi $6 \times 6 \times 3$ dengan kernel filter yang memiliki dimensi $3 \times 3 \times 3$ sebanyak n_f. Perlu diingat disini kita tidak menggunakan *padding* dan *stride* sehingga tiap filter akan menghasilkan matriks dengan dimensi 4×4. Jika kita memiliki filter sebanyak n_f maka kita memiliki hasil dengan dimensi $4 \times 4 \times n_f$. Berikutnya seperti pada *artificial neural networks* kita menggunakan fungsi non-linear $g(.)$ seperti sigmoid ataupun RelU untuk menghasilkan hipotesa kita dengan nilai koreksi bias. Seperti pada *artificial neural networks* kita harus mencari nilai optimal pada kernel filter kita sehingga jika kita memiliki total $n_f = 10$ kernel filter kita harus melakukan proses optimisasi sebanyak $3 \times 3 \times 3 + 1\text{bias} = 28$ dikali dengan $n_f = 10$ sehingga total parameter pada layer ini memiliki 280 parameter yang harus dioptimisasi.

7.3 Convolutional Neural Networks

Secara umum notasi untuk layer *convolution* ke$-l$ dapat ditulis sebagai berikut

- Misalkan kita memiliki data masukan awal dengan dimensi $n^{l-1} \times n^{l-1} \times n_f^{l-1}$. Perlu diingat disini untuk data awal biasa kita memiliki $n_f^0 = 3$. Namun untuk layer ke$-l$ biasanya n_f^l bergantung berapa filter yang kita gunakan di layer tersebut.
- Misal pada layer ke$-l$ kita memiliki kernel filter dengan dimensi $f^l \times f^l \times n_f^{l-1}$. Tiap layer bisa memiliki
- Keluaran dari *convolution* layer ke$-l$ dapat ditulis $n^l \times n^l \times n_f^l$, dimana $n^l = \lfloor \frac{n^{l-1}+2p^l-f^l}{s^l} + 1 \rfloor$.

Untuk mempermudah secara visualisasi kita akan bahas di gambar 7.6 dan kita akan bahas dimensi pada setiap layer yang ada Secara detail forward proses dan dimensi pada tiap layer dapat dijelaskan sebagai

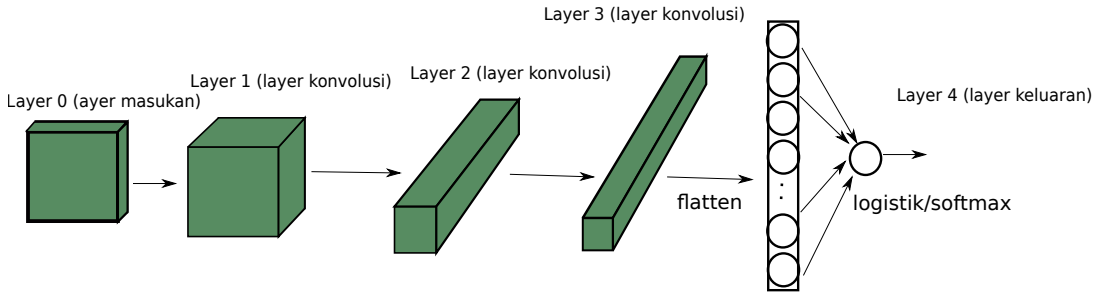

Figure 7.6: Convolutional neural networks

berikut

- Pada layer ke-0 kita memiliki data masukan, misal gambar yang memiliki tiga kanal *RGB* dengan dimensi $n^0 = 64$. Sehingga kita memiliki dimensi data $64 \times 64 \times 3$.
- Pada layer ke-1 kita memiliki layer *convolution* dengan dimensi filter $f^1 = 3$, *stride* $s^1 = 1$ dan tanpa *padding* $p^1 = 0$. Misalkan pada layer ini kita memiliki kernel filter sebanyak $n_f^1 = 20$. Dengan parameter tersebut kita bisa menghitung keluaran dari layer *convolution* ke-1 dengan dimensi $n^1 = \lfloor \frac{64-3}{1} + 1 \rfloor = 62$. Secara total dimensi data pada keluaran layer ke-1 adalah $62 \times 62 \times 20$.
- Pada layer ke-2 kita juga memiliki layer *convolution* dengan parameter $f^2 = 6$, *padding* $p^2 = 0$ dan *stride* $s^2 = 2$. Total filter pada layer ke-2 sebanyak $n_f^2 = 40$. Sehingga pada layer ini kita memiliki $n^2 = \lfloor \frac{62-6}{2} + 1 \rfloor = 29$ dan total dimensi keluaran $29 \times 29 \times 40$.
- Di layer ke-3 kita juga memiliki layer *convolution* misal dengan parameter $f^3 = 3$, *stride* $s^3 = 2$, dan tanpa *padding* $p^3 == 0$ sebanyak $n_f^3 = 60$. Total dimensi keluaran pada layer ini adalah $14 \times 14 \times 60$
- Sebelum layer terakhir kita melakukan vektorisasi pada data kita sehingga total dimensi sebelum masuk fungsi logistik atau yang lainnya adalah 11760. Seperti yang kita pelajari sebelumnya fungsi regresi tidak mengubah dimens. Sehingga kita memiliki hipotesa awal dengan dimensi vektor $\hat{\mathbf{y}}$ adalah 11760.

7.4 Layer pooling

Selain layer *convolution* dalam *convolutional neural networks* kita juga memiliki jenis layer lain seperti pooling layer. Tujuan menggunakan pooling adalah untuk mengurangi dimensi dan mempercepat proses komputasi. Beberapa jenis layer pooling yang terkenal adalah *Max Pooling* dan *Average Pooling*. Secara umum filosofi layer pooling adalah untuk merepresentasikan sub area atau beberapa pixel pada data menjadi satu pixel. *Max Pooling* mencari nilai maximum dari area tersebut dan *Average Pooling* menggunakan persamaan mencari rata-rata seperti yang divisualisasikan pada gambar 7.7. Pada contoh ini

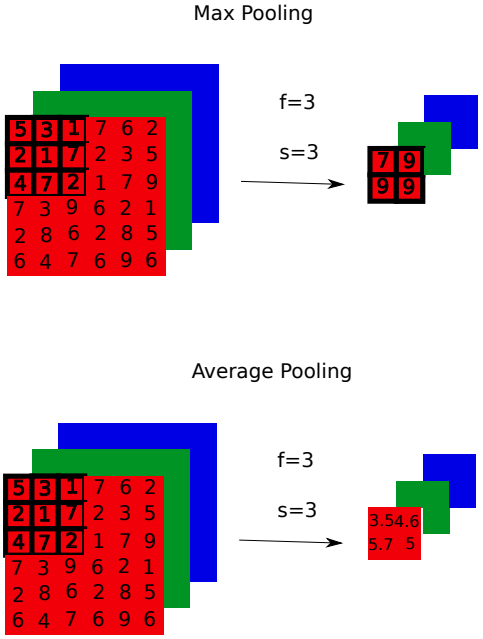

Figure 7.7: Convolutional neural networks

kita menggunakan kernel filter dengan dimensi $f = 3$, *stride* $s = 3$ dan *padding* $p = 0$. Seperti yang dijelaskan pada gambar, *Max Pooling* mencari nilai maximum pada tiap area pada data yang sesuai dengan dimensi kernel filter. Pada *Average Pooling* kita mencari nilai rata-rata dengan cara menjumlahkan semua nilai pixel pada area sesuai dengan dimensi kernel filter dan membaginya dengan jumlah pixel yang ada.

7.5 Contoh CNN

Beberapa jenis *convolutional neural networks* sudah sangat banyak diimplementasikan dengan jenis struktur yang berbeda. Berikut beberapa jenis struktur *convolutional neural networks* yang ada, misal LeNet –5 yang dikembangkan oleh Yann LeCun, Yoshua Bengio dan kolaborator [16] dan AlexNet yang dikembangkan oleh Geoffrey Hinton bersama dengan kolaborator [17].

Sama seperti metoda dalam *artificial neural networks* kita ingin meminimalkan fungsi loss antara hipotesa kita dengan label data yang kita punyai $\mathbf{y}$ dan mencari nilai optimal dalam parameter kernel filter kita.

Chapter 8

Aplikasi Artificial Neural Network

Pada bab kali ini kita akan coba mengimplementasikan aplikasi dari regresi logistik multikelas dengan *artificial neural network*. Kita akan mengklasifikasikan data tulisan tangan atau yang umum disebut dengan MNIST dataset [18]. Dataset ini memiliki sekitar $m = 60000$ training data tulisan tangan dengan dimensi gambar $K \times K$ dengan $K = 28$, atau jika kita gunakan vektorisasi maka vektor gambar kita adalah $N = 28 \times 28 = 784$. Data ini juga dilengkapi dengan label nilai yang benar $\mathbf{y}$ yang memiliki dimensi $m = 60000$. Pada bab kali ini kita akan membuat program regresi logistik dan *artificial neural network* untuk mengklasifikasikan angka pada tiap gambar di MNIST dataset. Secara umum MNIST dataset dapat dilihat pada gambar 8.1.

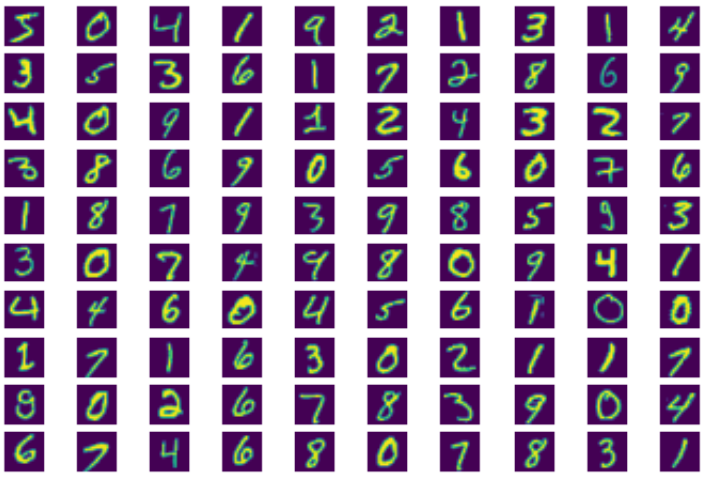

Figure 8.1: Sample pada MNIST dataset

8.1 Klasifikasi Tulisan Tangan: Regresi Logistik

Dalam implementasi regresi logistik multikelas, kita akan menggunakan klasifikasi satu vs semua. Dimana kita akan membandingkan setiap dua kelas sampai semua total kelas dapat dibandingkan. Perlu diingat kita memiliki 10 kelas yaitu angka dari $0, 1, \ldots, 9$. Langkah-langkah dalam implementasi regresi logistik satu vs semua dapat diringkas sebagai berikut

- Dataset kita disimpan dalam format `*.mat` untuk bisa kita gunakan pada python kita harus menggunakan module `scipy.io.loadmat`.
- Buat variabel yang berbeda untuk `'trainX'`, `'trainY'`, `'testX'`, dan `'testY'`. Di sisi lain bagi training data dan validasi data sebanyak 50000 dan 10000. Normalisasi data dengan membagi data masukan `'trainX'` dan `'testX'` dengan nilai maksimum pixel, yaitu 255.
- Buat fungsi python untuk mendefinisikan fungsi sigmoid `def sigmoid`, *cost function* untuk regresi logistik `def costfunction` dan inisial parameter $\boldsymbol{\theta}$.

- Buat fungsi python untuk satu vs semua dan prediksi satu vs semua serta metode gradient untuk update inisial parameter $\boldsymbol{\theta}$. Disini untuk mempermudah metode gradient kita bisa menggunakan module `def scipy.optimize.minimize` sehingga kita tidak perlu mengimplementasi metode gradient dari awal.

- Setelah selesai proses optimisasi, ambil nilai optimal $\boldsymbol{\theta}$ dan buat fungsi baru untuk prediksi. Bandingkan akurasi untuk `training`, `validasi`, dan `test` data

- Test regresi logistik multikelas yang kita implementasikan dengan tulisan tangan sendiri untuk angka diantara 0 sampai 9. Perlu diingat disini ada baiknya kita tetap pada dimensi 28×28 untuk meminimalkan error pada implementasi.

8.2 Klasifikasi Tulisan Tangan: *Artificial Neural Network*

Untuk *artificial neural network* kita akan menggunakan parameter setting dengan satu *hidden layer* dimana total neuron yang kita gunakan adalah 40. Sehingga secara total kita memiliki 3 layer *artificial neural network*, yaitu untuk layer input sebanyak $N + 1 \times m = 785 \times 50000$, untuk *hidden layer* kita memiliki parameter $n_1 + 1 \times N + 1 = 41 \times 785$ dan untuk output layer kita memiliki kelas $C \times n_1 + 1 = 10 \times 41$.

Kita akan menggunakan 50000 dataset untuk training dan 10000 dataset untuk validasi. Berikut adalah langkah langkah untuk melakukan implementasi

- Dataset kita disimpan dalam format `*.mat` untuk bisa kita gunakan pada python kita harus menggunakan module `scipy.io.loadmat`.

- Buat variabel yang berbeda untuk `'trainX'`, `'trainY'`, `'testX'`, dan `'testY'`. Di sisi lain bagi training data dan validasi data sebanyak 50000 dan 10000. Normalisasi data dengan membagi data masukan `'trainX'` dan `'testX'` dengan nilai maksimum pixel, yaitu 255.

- Untuk melatih model kita, kita harus mendefinisikan nilai *ground truth* pada label kita dengan cara untuk setiap nilai dari $0, 1, \ldots, 9$ jika `'trainY` dan `'testY` benar beri nilai `True` dan jika salah beri nilai `False`.

- Setting nilai parameter banyaknya input data, untuk kasus kita $N = 784$ dan satu *hidden layer* dengan total neuron 40 dan multikelas label untuk layer output 10. Kita akan menggunakan regularisasi dengan parameter $\lambda = 0.5$ dan training data dengan total $m = 50000$.

- Buat fungsi python untuk mendefinisikan `def sigmoid`, `def sigmoidgradient`, dan fungsi untuk inisial parameter $\boldsymbol{\Theta}$.

- Buat fungsi python untuk *forward model* dan *backward model* serta metode gradient untuk update inisial parameter $\boldsymbol{\Theta}$. Disini untuk mempermudah metode gradient kita bisa menggunakan module `def scipy.optimize.minimize`

- Setelah selesai proses optimisasi, ambil nilai optimal $\boldsymbol{\Theta}$ dan buat fungsi baru untuk prediksi. Bandingkan akurasi untuk `training`, `validasi`, dan `test` data

- Test *artificial neural network* yang kita implementasikan dengan tulisan tangan sendiri untuk angka diantara 0 sampai 9. Perlu diingat disini ada baiknya kita tetap pada dimensi 28×28 untuk meminimalkan error pada implementasi.

Langkah-langkah implementasi akan lebih mudah dilihat dalam program python yang akan saudara kerjakan.

Chapter 9

Modern Tools: Pytorch

Kita sudah belajar untuk mengimplementasikan *artificial neural network* sederhana dan berhasil untuk melakukan klasifikasi pada tulisan tangan MNIST data. Secara umum implementasi pada python terbagi menjadi dua bagian, yaitu *forward* dan *backward* model. Implementasi secara manual perlu dilakukan dengan cermat dan jika kita mengimplementasikan *deep neural netowrk* kita perlu banyak menulis fungsi serta produk antara matriks dan vektor. Untuk data yang sangat besar perlu keterampilan dalam *high performance computing* agar membuat implementasi kita efisien dan cepat. Terlebih lagi untuk aplikasi yang memerlukan waktu *real time* seperti Apple Siri, Halo Google dan Amazon Alexa.

Terlebih lagi untuk aplikasi *autonomous driving* sehingga mobil bisa mengendarai dan mengenali halangan secara otomatis, kita pelru membuat komputasi secara *real time*. Oleh karena itu implementasi yang efisien merupakan hal yang krusial untuk mencapai komputasi yang cepat. Dengan banyak alasan itu tim peneliti di Google ataupun Facebook menciptakan *modern tools* yang bisa digunakan secara sederhana, yaitu *TensorFlow* [19] dan *PyTorch* [20], dimana kita tidak usah implementasi semua layer secara manual namun tinggal memanggil module yang sudah ada. Di bab kali ini kita akan implementasikan module *PyTorch* untuk klasifikasi MNIST dataset.´

9.1 Regresi logistik dengan Pytoch

Secara umum kita bisa menggunakan Pytorch sebagai berikut. Sama seperti module lain pada python, kita pertama tama harus *import module pytorch* setelah kita melakukan instalasi.

```
1
2 # Import module torch dan torchvision yang memiliki dataset mnist
3 import torch
4 from torch.autograd import Variable
5 import torchvision.transforms as transforms
6 import torchvision.datasets as dsets
```

Listing 9.1: Regresi Logistik dengan Pytorch

Kita akan memisahkan antara dataset untuk training dan dataset untuk test. Perlu diingat di sini pytorch menggunakan format data tensor sehingga `ToTensor()` di sini berguna untuk mengubah format data menjadi tensor.

```
1 train_dataset = dsets.MNIST(root='./data', train=True, transform=transforms.ToTensor(),
      download=True)
2 test_dataset = dsets.MNIST(root='./data', train=False, transform=transforms.ToTensor())
```

Listing 9.2: Regresi Logistik dengan Pytorch

Setelah kita load dataset kita kita akan menggunakan parameter yang harus disediakan untuk melatih regresi logistik kita

```
1 # Definisi variabel dalam optimisasi
2
3 # Batch size merupakan banyaknya data yang di proses misal jika kita mempunyai data
      10000
4 # dengan batch size 200, berarti kita memiliki 50 batch dengan per batch kita
      mengeksekusi 200 data
5 batch_size = 100
6
```

```
# Banyaknya iterasi pada gradient
n_iters = 6000

# Epoch berarti berapa kali algoritma kita akan bekerja untuk memproses semua data/per batch
epochs = n_iters / (len(train_dataset) / batch_size)
# Banyaknya input data kita, ingat kita memiliki gambar MNIST dengan dimensi 28 x 28 sehingga kita memiliki
# Vektorisasi 784
input_dim = 784
# Banyaknya kelas, disini kita ingin mengklasifikasikan multikelas dari 0,1,..9
output_dim = 10
# Learning rate merupakan parameter skala dalam turunan, ingat parameter alpha
lr_rate = 0.001
```

Listing 9.3: Regresi Logistik dengan Pytorch

Di sini kita menggunakan variabel training dan test data untuk load data kita dan fungsi boolean apakah kita mau shuffle data kita atau tidak.

```
# Variabel training data dan test data
train_loader = torch.utils.data.DataLoader(dataset=train_dataset, batch_size=batch_size, shuffle=True)
test_loader = torch.utils.data.DataLoader(dataset=test_dataset, batch_size=batch_size, shuffle=False)
```

Listing 9.4: Regresi Logistik dengan Pytorch

Dalam pytorch kita harus mendefinisikan *class*, ingat definisi *class* di *object oriented program* kita memiliki beberapa atribut, atau metoda, disini kita memiliki *constructor* `init` yang menggunakan fungsi super agar kita mendapatkan properti dari `torch.nn.Module`. Metoda *forward* merupakan *forward model* kita

```

class LogisticRegression(torch.nn.Module):
    def __init__(self, input_dim, output_dim):
        super(LogisticRegression, self).__init__()
        self.linear = torch.nn.Linear(input_dim, output_dim)

    def forward(self, x):
        outputs = self.linear(x)
        return outputs
```

Listing 9.5: Regresi Logistik dengan Pytorch

Setelah mendefinisikan *class* biasanya kita membuat objek atau *instance* dari kelas `LogisticRegresion` dengan input dimensi sebanyak gambar kita dan output dimensi sesuai banyaknya kelas

```
model = LogisticRegression(input_dim, output_dim)
```

Listing 9.6: Regresi Logistik dengan Pytorch

Di sini kita menentukan *cost function* kita, yaitu seperti yang kita pelajari disebut juga *cross entropy.*

```
criterion = torch.nn.CrossEntropyLoss() # computes softmax and then the cross entropy
```

Listing 9.7: Regresi Logistik dengan Pytorch

Berikutnya kita memilih metode gradient yang kita gunakan, di dalam modul yang kita pelajari kita hanya menggunakan simpel gradient. Namun pada kenyataannya sekarang ada banyak variasi metode gradient seperti yang sudah dibahas pada bab fundamental. Di sini kita menggunakan *SGD (Stochastic Gradient Method)* dengan *learning rate* atau *step size* yang sudah kita tentukan.

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

Listing 9.8: Regresi Logistik dengan Pytorch

Setelah kita mendefinisikan semua parameter saat nya kita melakukan iterasi pada network kita seperti berikut

```


# Masuk kedalam iterasi per epoch
iter = 0
for epoch in range(int(epochs)):
    # Load semua data dan label nya
```

```
    for i, (images, labels) in enumerate(train_loader):
        images = Variable(images.view(-1, 28 * 28))
        labels = Variable(labels)

        # Definisikan dan inisialisasi gradient dengan 0
        optimizer.zero_grad()
        # Hitung instance kita dengan input data dan disini
        # Kita hitung forward model
        outputs = model(images)
        # Bandingkan dengan label dengan menggunakan cost function kita
        loss = criterion(outputs, labels)
        # Lakukan prosess backward dengan stepsize
        loss.backward()
        optimizer.step()

        # Tambah iterasi
        iter+=1
        # Condisi agar kita menampilkan nilai tiap iterasi yang habis dibagi 500
        if iter%500==0:
            # Hitung akurasi network kita dan print hasilnya
            # Dengan menggunakan test data
            correct = 0
            total = 0
            for images, labels in test_loader:
                images = Variable(images.view(-1, 28*28))
                outputs = model(images)
                _, predicted = torch.max(outputs.data, 1)
                total+= labels.size(0)
                # fJika kita menggunakan GPU, kita harus membawa hasilya pertama tam ke
    CPU agar kita bisa lihat hasilnya
                correct+= (predicted == labels).sum()
            accuracy = 100 * correct/total
            print("Iteration: {}. Loss: {}. Accuracy: {}.".format(iter, loss.item(),
    accuracy))
```

Listing 9.9: Regresi Logistik dengan Pytorch

Berikut adalah hasil dari regresi logistik kita ketika memperlihatkan hasilnya tiap iterasi ke 500.

```
Iteration: 500. Loss: 1.9379119873046875. Accuracy: 66.80999755859375.
Iteration: 1000. Loss: 1.5509116649627686. Accuracy: 76.0999984741211.
Iteration: 1500. Loss: 1.332176923751831. Accuracy: 78.8499984741211.
Iteration: 2000. Loss: 1.204288363456726. Accuracy: 80.51000213623047.
Iteration: 2500. Loss: 1.0564988851547241. Accuracy: 81.7300033569336.
Iteration: 3000. Loss: 1.0266402959823608. Accuracy: 82.62000274658203.
Iteration: 3500. Loss: 0.9627998471260071. Accuracy: 83.30999755859375.
Iteration: 4000. Loss: 0.8270407915115356. Accuracy: 83.72000122070312.
Iteration: 4500. Loss: 0.8181248307228088. Accuracy: 84.20999908447266.
Iteration: 5000. Loss: 0.7286779880523682. Accuracy: 84.58999633789062.
Iteration: 5500. Loss: 0.8189718127250671. Accuracy: 84.91999816894531.
Iteration: 6000. Loss: 0.7198503613471985. Accuracy: 85.33000183105469.
```

Listing 9.10: Regresi Logistik dengan Pytorch

Sangat simpel bukan? Namun sangat bagus untuk kita mengerti dahulu sistem di dalamnya agar tidak bingung apa yang Pytorch kerjakan secara konsep. Dalam bab ini diharapkan peserta dapat aktif mencari tahu banyaknya variabel yang perlu digunakan dalam implementasi pytorch melalui dokumentasi resmi pytorch: https://pytorch.org/docs/stable/index.html

9.2 Tugas Pytorch

- Pelajari tutorial resmi dari pytroch di https://pytorch.org/docs/stable/index.html
- Bisakah saudara implementasikan *artificial neural network* dengan menggunakan pytorch?
- Pelajari jenis struktur *artificial neural netwok* yang lain seperti *convolutional neural network* dan implementasikan dengan data yang ada!

Bibliography

[1] Heinz Muehlenbein, "Artificial intelligence and neural networks the legacy of alan turing and john von neumann," *Int J Comput*, vol. 5, no. 3, pp. 10–20, 2014.

[2] Norbert Wiener, "Cybernetics," *Scientific American*, vol. 179, no. 5, pp. 14–19, 1948.

[3] Dave Steinkraus, Ian Buck, and PY Simard, "Using gpus for machine learning algorithms," in *Eighth International Conference on Document Analysis and Recognition (ICDAR'05)*. IEEE, 2005, pp. 1115–1120.

[4] Yann LeCun, Yoshua Bengio, et al., "Convolutional networks for images, speech, and time series," *The handbook of brain theory and neural networks*, vol. 3361, no. 10, pp. 1995, 1995.

[5] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio, "Generative adversarial nets," *Advances in neural information processing systems*, vol. 27, 2014.

[6] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[7] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al., "Mastering the game of go with deep neural networks and tree search," *nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[8] Andrew W Senior, Richard Evans, John Jumper, James Kirkpatrick, Laurent Sifre, Tim Green, Chongli Qin, Augustin Žídek, Alexander WR Nelson, Alex Bridgland, et al., "Improved protein structure prediction using potentials from deep learning," *Nature*, vol. 577, no. 7792, pp. 706–710, 2020.

[9] Diederik P. Kingma and Jimmy Ba, "Adam: A Method for Stochastic Optimization," *3rd International Conference for Learning Representations*, Jan. 2017, arXiv: 1412.6980.

[10] John Duchi, Elad Hazan, and Yoram Singer, "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization," *Journal of Machine Learning Research*, vol. 12, no. Jul, pp. 2121–2159, 2011.

[11] Matthew D. Zeiler, "ADADELTA: An Adaptive Learning Rate Method," *arXiv:1212.5701 [cs]*, Dec. 2012, arXiv: 1212.5701.

[12] Andrei Nikolajevits Tihonov, "Solution of incorrectly formulated problems and the regularization method," *Soviet Math.*, vol. 4, pp. 1035–1038, 1963.

[13] Robert Tibshirani, "Regression shrinkage and selection via the lasso," *Journal of the Royal Statistical Society: Series B (Methodological)*, vol. 58, no. 1, pp. 267–288, 1996.

[14] Frank Rosenblatt, "The perceptron: a probabilistic model for information storage and organization in the brain.," *Psychological review*, vol. 65, no. 6, pp. 386, 1958.

[15] Jürgen Schmidhuber, "Deep learning in neural networks: An overview," *Neural networks*, vol. 61, pp. 85–117, 2015.

[16] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[17] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton, "Imagenet classification with deep convolutional neural networks," *Advances in neural information processing systems*, vol. 25, 2012.

[18] Yann LeCun and Corinna Cortes, "MNIST handwritten digit database," 2010.

[19] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, Software available from tensorflow.org.

[20] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds., pp. 8024–8035. Curran Associates, Inc., 2019.