12

Autoencoder

> "The goal is to turn data into information, and information into insight."
>
> Carly Fiorina

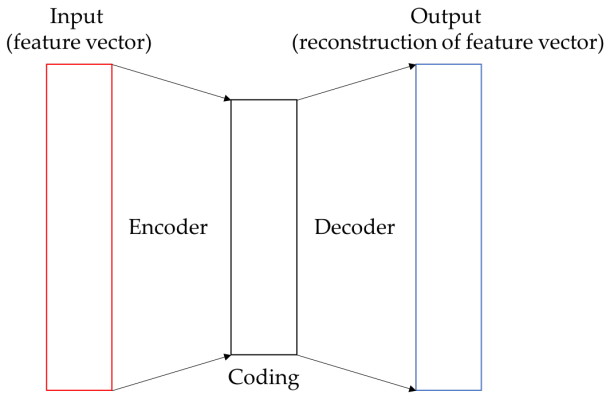
Bab ini memuat materi yang relatif sulit (karena agak *high level*). Bab ini memuat materi *autoencoder* serta penerapannya pada pemrosesan bahasa alami (***natural language processing***–NLP). Berhubung aplikasi yang diceritakan adalah aplikasi pada NLP, kami akan memberi sedikit materi (*background knowledge*) agar bisa mendapat gambaran tentang persoalan pada domain tersebut. Bagi yang tertarik belajar NLP, kami sarankan untuk membaca buku [64]. Teknik yang dibahas pada bab ini adalah ***representation learning*** untuk melakukan pengurangan dimensi pada *feature vector* (*dimensionality reduction*), teknik ini biasanya digolongkan sebagai *unsupervised learning*. Artinya, *representation learning* adalah mengubah suatu representasi menjadi bentuk representasi lain yang ekuvalen, tetapi berdimensi lebih rendah; sedemikian sehingga informasi yang terdapat pada representasi asli tidak hilang/terjaga. Ide dasar teknik ini bermula dari dekomposisi matriks pada aljabar linear.

12.1 Representation Learning

Pada bab model linear, kamu telah mempelajari ide untuk mentransformasi data menjadi dimensi lebih tinggi agar data tersebut menjadi *linearly separable*. Pada bab ini, kamu mempelajari hal sebaliknya, yaitu mengurangi dimensi. *Curse of dimensionality* dapat dipahami secara mendalam apabila kamu membaca buku [65]. Untuk melakukan klasifikasi maupun *clustering*, kita membutuhkan fitur. Fitur tersebut haruslah dapat membedakan satu *instance* dan *instance* lainnya. Seringkali, untuk membedakan *instance* satu dan

instance lainnya, kita membutuhkan *feature vector* yang berdimensi relatif "besar". Karena dimensi *feature vector* besar, kita butuh sumber daya komputasi yang besar juga (bab 9). Untuk itu, terdapat metode-metode ***feature selection***[1] untuk memilih fitur-fitur yang dianggap "representatif" dibanding fitur lainnya. Sayangnya, bila kita menggunakan metode-metode *feature selection* ini, tidak jarang kita kelihangan informasi yang memuat karakteristik data. Dengan kata lain, ada karakteristik yang hilang saat menggunakan *feature selection*.

Pertanyaan yang kita ingin jawab adalah apakah ada cara untuk merepresentasikan data ke dalam bentuk yang membutuhkan memori lebih sedikit tanpa adanya kehilangan informasi? Kita dapat memanfaatkan prinsip *principal component analysis* yang sudah kamu pelajari pada bab 9 untuk mereduksi dimensi data (mengurangi dimensi *input*), pada saat yang bersamaan, menjaga karakteristik data. ***Representation learning*** adalah metode untuk melakukan **kompresi** *feature vector* menggunakan *neural network*.[2] Proses melakukan kompresi disebut ***encoding***, hasil *feature vector* dalam bentuk terkompres disebut ***coding***, proses mengembalikan hasil kompresi ke bentuk awal disebut ***decoding***.[3] *Neural network* yang mampu melakukan proses *encoding* disebut ***encoder***, sedangkan ***decoder*** untuk proses *decoding* [66, 67, 68, 69, 70].



Gambar 12.1: Contoh autoencoder sederhana.

Contoh *representation learning* paling sederhana kemungkinan besar adalah ***autoencoder*** yaitu *neural network* yang dapat merepresentasikan data kemudian merekonstruksinya kembali. Ilustrasi *autoencoder* dapat dilihat pada

[1] http://scikit-learn.org/stable/modules/feature_selection.html
[2] Istilah *representation learning* pada umumnya mengacu dengan teknik menggunakan *neural network*.
[3] Bisa dianggap sebagai proses menginterpretasikan *coding*.

Gambar 12.1. Karena tujuan *encoder* untuk kompresi, bentuk terkompresi haruslah memiliki dimensi lebih kecil dari dimensi *input*. *Neural network* mampu melakukan "kompresi" dengan baik karena ia mampu menemukan *hidden structure* dari data. *Autoencoder* dilatih untuk memilimalkan meminimalkan *loss*. Kamu mungkin berpikir bahwa idealnya, *output* harus sama dengan *input*, yaitu *autoencoder* dengan tingkat *loss* 0%. Akan tetapi, kita sebenarnya tidak ingin *autoencoder* memiliki performa 100% (subbab 12.4).

Contoh klasik lainnya adalah ***N-gram language modelling***, yaitu memprediksi kata y_t diberikan suatu konteks (*surrounding words*) misal kata sebelumnya y_{t-1} (bigram), i.e., $P(y_t \mid y_{t-1})$ Apabila kita mempunyai *vocabulary* sebesar $40,000$ berarti suatu bigram model membutuhkan *memory* sebesar $40,000^2$ (kombinatorial). Apabila kita ingin memprediksi kata diberikan *history* yang lebih panjang (misal dua kata sebelumnya–trigram) maka kita membutuhkan *memory* sebesar $40,000^3$. Artinya, *memory* yang dibutuhkan berlipat secara eksponensial. Tetapi, terdapat strategi menggunakan *neural network* dimana parameter yang dibutuhkan tidak berlipat secara eksponensial walau kita ingin memodelkan konteks yang lebih besar [71].

12.2 Singular Value Decomposition

Sebelum masuk ke *autoencoder* secara matematis, penulis akan memberikan sedikit *overview* tentang dekomposisi matriks. Seperti yang sudah dijelaskan pada bab-bab sebelumnya, dataset dimana setiap *input* direpresentasikan oleh *feature vector* dapat disusun menjadi matriks $\mathbf{X}$ berukuran $N \times F$, dimana N adalah banyaknya sampel[4] dan F adalah dimensi fitur. Pada *machine learning*, dekomposisi atau reduksi dimensi sangat penting dilakukan terutama ketika dataset berupa *sparse matrix*. Dekomposisi berkaitan erat dengan *principal component analysis* (PCA) yang sudah kamu pelajari. Teknik PCA (melalui *eigendecomposition*) mendekomposisi sebuah matriks $\mathbf{X}$ menjadi tiga buah matriks, seperti diilustrasikan pada persamaan 12.1. Matriks $\mathbf{A}$ adalah kumpulan eigenvector dan λ adalah sebuah diagonal matriks yang berisi nilai eigenvalue, dimana $\mathbf{U}$ berukuran $N \times N$, $\mathbf{V}$ berukuran $N \times F$, dan $\mathbf{W}^T$ berukuran $F \times F$.

$$\mathbf{X} = \mathbf{A}\lambda\mathbf{A}^{-1} \tag{12.1}$$

PCA membutuhkan matriks yang kamu ingin dekomposisi berbentu simetris. Sedangkan, teknik *singular value decomposition* (SVD) tidak. Dengan konsep yang mirip dengna PCA, matriks $\mathbf{X}$ dapat difaktorisasi menjadi tiga buah matriks menggunakan teknik SVD, dimana operasi ini berkaitan dengan mencari eigenvectors, diilustrasikan pada persamaan 12.2.

$$\mathbf{X} = \mathbf{U}\ \mathbf{V}\ \mathbf{W}^T \tag{12.2}$$

[4] Banyaknya training data.

Perlu diperhatikan, matriks $\mathbf{V}$ adalah sebuah diagonal matriks (elemennya adalah nilai *singular value* dari $\mathbf{X}$). $\mathbf{U}$ disebut *left-singular vectors* yang tersusun atas eigenvector dari $\mathbf{X}\mathbf{X}^T$. Sementara, $\mathbf{W}$ disebut *right-singular vectors* yang tersusun atas eigenvector dari $\mathbf{X}^T\mathbf{X}$.

Misalkan kita mempunyai sebuah matriks lain $\hat{\mathbf{V}}$ berukuran $K \times K$, yaitu modifikasi matriks $\mathbf{V}$ dengan mengganti sejumlah elemen diagonalnya menjadi 0 (analogi seperti menghapus beberapa baris dan kolom yang dianggap kurang penting). Sebagai contoh, perhatikan ilustrasi berikut!

$$\mathbf{V} = \begin{bmatrix} \alpha_1 & 0 & 0 & 0 & 0 \\ 0 & \alpha_2 & 0 & 0 & 0 \\ 0 & 0 & \alpha_3 & 0 & 0 \\ 0 & 0 & 0 & \alpha_4 & 0 \end{bmatrix} \quad \hat{\mathbf{V}} = \begin{bmatrix} \alpha_1 & 0 & 0 \\ 0 & \alpha_2 & 0 \\ 0 & 0 & \alpha_3 \end{bmatrix}$$

Kita juga dapat me-nol-kan sejumlah baris dan kolom pada matriks $\mathbf{U}$ dan $\mathbf{W}$ menjadi $\hat{\mathbf{U}}$ $(N \times K)$ dan $\hat{\mathbf{W}}^T$ $(K \times F)$. Apabila kita mengalikan semuanya, kita akan mendapat matriks $\hat{\mathbf{X}}$ yang merupakan *approximation* untuk matriks asli $\mathbf{X}$, seperti diilustrasikan pada persamaan 12.3.

$$\hat{\mathbf{X}} = \hat{\mathbf{U}}\ \hat{\mathbf{V}}\ \hat{\mathbf{W}}^T \tag{12.3}$$

Suatu baris dari matriks $\mathbf{E} = \hat{\mathbf{U}}\ \hat{\mathbf{V}}$ dianggap sebagai aproksimasi baris matriks $\mathbf{X}$ berdimensi tinggi [1]. Artinya, menghitung *dot-product* $\mathbf{E}_i \cdot \mathbf{E}_j = \hat{\mathbf{X}}_i \cdot \hat{\mathbf{X}}_j$. Artinya, operasi pada matriks $\mathbf{E}$ kurang lebih melambangkan operasi pada matriks asli. Konsep ini menjadi fundamental *autoencoder* yang akan dibahas pada subbab berikutnya. Operasi data pada level *coding* dianggap merepresentasikan operasi pada bentuk aslinya. Matriks aproksimasi ini memanfaatkan sejumlah K arah paling berpengaruh pada data. Dengan analogi tersebut, sama seperti mentransformasi data ke bentuk lain dimana data hasil transformasi memiliki varians yang tinggi.

12.3 Ide Dasar Autoencoder

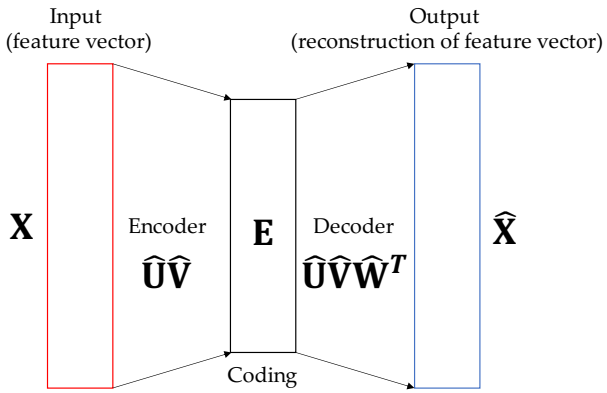
Seperti yang sudah dijelaskan *autoencoder* adalah *neural network* yang mampu merekonstruksi *input*. Ide dasar *autoencoder* tidak jauh dari konsep dekomposisi/*dimentionality reduction* menggunakan *singular value decomposition*. Diberikan dataset $\mathbf{X}$, kita ingin mensimulasikan pencarian matriks $\hat{\mathbf{X}}$ yang merupakan sebuah aproksimasi dari matriks asli. Arsitektur dasar *autoencoder* diberikan pada Gambar 12.1. Kita memberi input matriks $\mathbf{X}$ pada *autoencoder*, kemudian ingin *autoencoder* tersebut menghasilkan matriks yang sama. Dengan kata lain, *desired output* sama dengan *input*. Apabila dihubungkan dengan pembahasan ANN pada bab sebelumnya, *error function* untuk melatih *autoencoder* diberikan pada persamaan 12.4, dimana $\mathbf{y}$ adalah output dari jaringan dan Z adalah dimensi *output*, N adalah banyaknya sampel dan $\mathbf{x}_i$ adalah data ke-i (*feature vector* ke-i).

$$E(\theta) = \frac{1}{N} \sum_{i=j}^{Z} \left(\mathbf{x}_{i[j]} - \mathbf{y}_{i[j]} \right)^2 \tag{12.4}$$

Persamaan 12.4 dapat kita tulis kembali sebagai persamaan 12.5, dimana f melambangkan fungsi aktivasi dan θ adalah ANN (kumpulan *weight matrices*).[5]

$$E(\theta) = \frac{1}{N} \sum_{j=1}^{Z} \left(\mathbf{x}_{i[j]} - f(\mathbf{x}_i, \theta)_{[j]} \right)^2 \tag{12.5}$$

Seperti yang sudah dijelaskan sebelumnya, *desired output* sama dengan *input*. Tetapi seperti yang kamu ketahui, mencapai *loss* sebesar 0% adalah hal yang susah. Dengan demikian, kamu dapat memahami secara intuitif bahwa *autoencoder* melakukan aproksimasi terhadap data asli, tetapi tidak sama persis. Apabila *loss* sebesar 0%, ditakutkan bahwa *autoencoder* semata-mata hanya melakukan translasi (penggesaran) data saja. Gambar 12.2 mengilustrasikan hubungan antara *autoencoder* dan *singular value decomposition*.[6] Perhatikan, ada dua proses *encoding* yaitu merepresentasikan data ke dimensi lebih rendah dan *decoding*–rekonstruksi kembali.



Gambar 12.2: Hubungan autoencoder dan *singular value decomposition* (analogi).

Perhatikan, *hidden layer*/*coding* dapat dianalogikan sebagai $\mathbf{E} = \hat{\mathbf{U}}\ \hat{\mathbf{V}}$. Dengan kata lain, kita dapat melakukan operasi *dot-product* pada *coding* untuk merepresentasikan *dot-product* pada data asli $\mathbf{X}$. Ini adalah ide utama

[5] Pada banyak literatur, kumpulan *weight matrices* ANN sering dilambangkan dengan θ

[6] Hanya sebuah analogi.

autoencoder, yaitu meng-aproksimasi/mengkompresi data asli menjadi bentuk lebih kecil *coding*. Kemudian, operasi pada bentuk *coding* merepresentasikan operasi pada data sebenarnya.

Autoencoder terdiri dari *encoder* (sebuah *neural network*) dan *decoder* (sebuah *neural network*). *Encoder* merubah *input* ke dalam bentuk dimensi lebih kecil (dapat dianggap sebagai kompresi). *Decoder* berusaha merekonstruksi *coding* menjadi bentuk aslinya. Secara matematis, kita dapat menulis *autoencoder* sebagai persamaan 12.6, dimana dec melambangkan *decoder*, enc melambangkan *encoder* dan $\mathbf{x}$ adalah *input*. *Encoder* diberikan pada persamaan 12.7 yang berarti melewatkan *input* pada suatu *layer* di *neural network* untuk menghasilkan representasi $\mathbf{x}$ berdimensi rendah, disebut *coding* $\mathbf{c}$. $\mathbf{U}$ dan α melambangkan *weight matrix* dan *bias*.

$$f(\mathbf{d}, \theta) = \text{dec}(\text{enc}(\mathbf{x})) \tag{12.6}$$

$$\mathbf{c} = \text{enc}(\mathbf{x}) = g(\mathbf{x}, \mathbf{U}, \alpha) \tag{12.7}$$

Representasi $\mathbf{c}$ ini kemudian dilewatkan lagi pada suatu *layer* untuk merekonstruksi kembali *input*, kita sebut sebagai *decoder*. *Decoder* diberikan pada persamaan 12.8 dimana $\mathbf{W}$ dan β melambangkan *weight matrix* dan *bias*. Baik pada fungsi *encoder* dan *decoder*, σ melambangkan fungsi aktivasi.

$$f(\mathbf{d}, \theta) = \text{dec}(\mathbf{c}) = h(\mathbf{c}, \mathbf{W}, \beta) \tag{12.8}$$

Pada contoh sederhana ini, *encoder* dan *decoder* diilustrasikan sebagai sebuah *layer*. Kenyataannya, *encoder* dan *decoder* dapat diganti menggunakan sebuah *neural network* dengan arsitektur kompleks.

Sekarang kamu mungkin bertanya-tanya, bila *autoencoder* melakukan hal serupa seperti *singular value decomposition*, untuk apa kita menggunakan *autoencoder*? (mengapa tidak menggunakan aljabar saja?) Berbeda dengan teknik SVD, teknik *autoencoder* dapat juga mempelajari fitur nonlinear.[7] Pada penggunaan praktis, *autoencoder* adalah *neural network* yang cukup kompleks (memiliki banyak *hidden layer*). Dengan demikian, kita dapat "mengetahui" **berbagai macam representasi** atau transformasi data. *Framework autoencoder* yang disampaikan sebelumnya adalah *framework* dasar. Pada kenyataannya, masih banyak ide lainnya yang bekerja dengan prinsip yang sama untuk mencari *coding* pada permasalahan khusus. *Output* dari *neural network* juga bisa tidak sama *input*-nya, tetapi tergantung permasalahan (kami akan memberikan contoh persoalan *word embedding*). Selain itu, *autoencoder* juga relatif fleksibel; dalam artian saat menambahkan data baru, kita hanya perlu memperbaharui parameter *autoencoder* saja. Kami sarankan untuk membaca *paper* [72, 73] perihal penjelasan lebih lengkap tentang perbedaan dan persamaan SVD dan *autoencoder* secara lebih matematis.

Secara sederhana, *representation learning* adalah teknik untuk mengkompresi *input* ke dalam dimensi lebih rendah tanpa (diharapkan) ada kehilangan

[7] Hal ini abstrak untuk dijelaskan karena membutuhkan pengalaman.

informasi. Operasi vektor (dan lainnya) pada level *coding* merepresentasikan operasi pada bentuk aslinya. Untuk pembahasan *autoencoder* secara lebih matematis, kamu dapat membaca pranala ini.[8] Setelah *autoencoder* dilatih, pada umumnya *encoder* dapat digunakan untuk hal lainnya juga, e.g., klasifikasi kelas gambar.

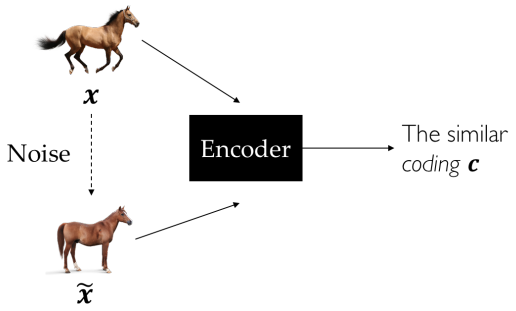
12.4 Resisting Perturbation

Pada subbab sebelumnya, telah dijelaskan bahwa mencapai performa 100% (100% rekonstruksi) pada *autoencoder* adalah hal yang tidak diinginkan. Hal ini disebabkan karena kita ingin menghindari *autoencoder* semata-mata hanya mempelajari *trivial identity function* [11], memiliki analogi dengan *one-to-one mapping*. Misalnya, suatu gambar kuda dipetakan ke *coding* $\mathbf{c}$, kemudian gambar kuda lainnya dipetakan ke *coding* $\hat{\mathbf{c}}$, dan $\hat{\mathbf{c}}$ tidak mirip dengan $\mathbf{c}$, i.e., *cosine similarity*-nya jauh. Artinya kita ingin *autoencoder* merepresentasikan dua hal yang mirip ke dalam bentuk representasi *coding* yang mirip juga! Walaupun kita ingin performa *autoencoder* tidak mencapai 100%, tapi kita masih ingin performanya dekat dengan 100%.

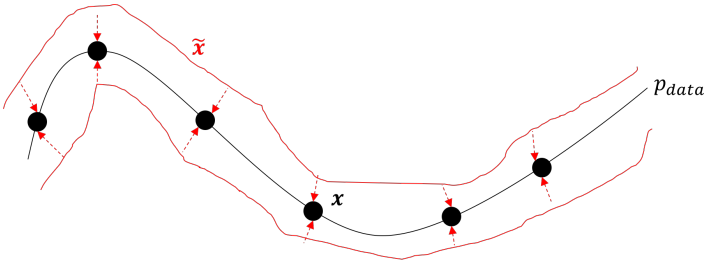
Tujuan utama *autoencoder* adalah mengekstraksi informasi penting tentang data yang ada (seperti *principal components*), bukan replikasi semata. Dengan alasan ini, *coding* pada umumnya memiliki dimensi lebih rendah dibanding *input*. Kita sebut arsitektur ini sebagai ***undercomplete autoencoder***. Apabila *coding* memiliki dimensi lebih besar dari *input*, disebut sebagai ***overcomplete autoencoder***–yang kemungkinan besar hanya mempelajari *trivial identity function* [11]. Kita dapat menggunakan teknik regularisasi pada *autoencoder* untuk memastikan tujuan kita tercapai, misal *sparse autoencoder*, *denoising autoencoder* dan *penalizing derivaties* [11]. Untuk mengilustrasikan permasalahan, buku ini membahas ***denoising autoencoder*** (silahkan baca buku [11] untuk teknik regularisasi lainnya).

Diberikan suatu *input* $\mathbf{x}$, kemudian kita lakukan *noise-injection* terhadap *input* tersebut, menghasilkan $\tilde{\mathbf{x}}$. Perhatikan Gambar 12.3, kita ingin *encoder* memberikan bentuk *coding* yang mirip bagi $\mathbf{x}$ dan $\tilde{\mathbf{x}}$. Kita ingin memaksa *autoencoder* untuk mempelajari sebuah fungsi yang tidak berubah terlalu jauh ketika *input* sedikit diubah. Hal ini disebut sebagai sifat ***resistance to perturbation***. Performa *autoencoder* yang bernilai 100% berbahaya karena *autoencoder* tersebut belum tentu mampu mempelajari sifat data, melainkan mampu "mengingat" *training data* saja (*mapping table*). Objektif *denoising autoencoder* diberikan pada persamaan 12.9, yaitu kemampuan merekonstruksi kembali data tanpa *noise*.

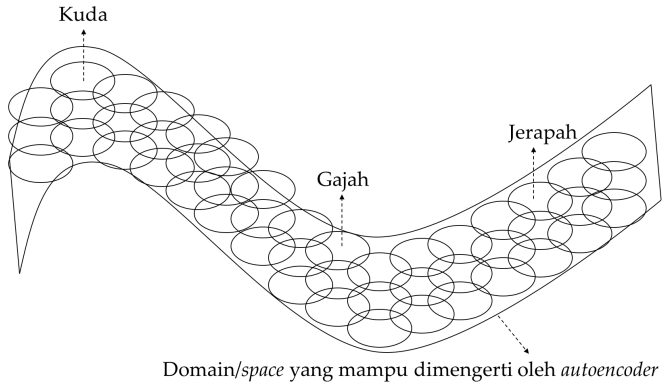
$$E(\theta) = \frac{1}{N}\sum_{j=1}^{Z}\left(\mathbf{x}_{i[j]} - f(\tilde{\mathbf{x}}_i, \theta)_{[j]}\right)^2 \tag{12.9}$$



Gambar 12.3: *Resisting Perturbation.*



Gambar 12.4: *Autoencoder* yang memiliki sifat *resistance to perturbation*, yaitu invarian terhadap sedikit perubahan.



Gambar 12.5: *Manifolds.*

Implikasi atau tujuan dari persamaan 12.9 diberikan pada Gambar 12.4 yang mengilustrasikan *invariant to slight changes*. Diberikan data dengan dis-

[8] https://jaan.io/what-is-variational-autoencoder-vae-tutorial/

tribusi asli p_{data}, dan data yang sudah terkena *noise* $\tilde{\mathbf{x}}$, *autoencoder* mampu mengembalikan $\tilde{\mathbf{x}}$ ke bentuk asli $\mathbf{x}$. Sebagai ilustrasi yang lebih "global", perhatikan Gambar 12.5 dimana suatu elips melambangkan ***manifolds***, dimana data yang mirip (e.g., kuda dengan berbagai macam pose) berada pada ruang yang cukup dekat satu sama lain. Kamu dapat memahami *resistance to perturbation* membuat *autoencoder* membentuk semacam "ruang lokal" yang merepresentasikan suatu data dan variannya.

12.5 Representing Context: Word Embedding

Subbab ini menceritakan salah satu aplikasi *autoencoder*. Pada domain NLP, kita ingin komputer mampu mengerti bahasa selayaknya manusia mengerti bahasa. Misalkan komputer mampu mengetahui bahwa "meja" dan "kursi" memiliki hubungan yang erat. Hubungan seperti ini tidak dapat terlihat berdasarkan teks tertulis, tetapi kita dapat menyusun kamus hubungan kata seperti WordNet.[9] WordNet memuat ontologi kata seperti hipernim, antonim, sinonim. Akan tetapi, hal seperti ini tentu sangat melelahkan, seumpama ada kata baru, kita harus memikirkan bagaimana hubungan kata tersebut terhadap seluruh kamus yang sudah dibuat. Pembuatan kamus ini memerlukan kemampuan para ahli linguistik.

Oleh sebab itu, kita harus mencari cara lain untuk menemukan hubungan kata ini. Ide utama untuk menemukan hubungan antarkata adalah ***statistical semantics hypothesis*** yang menyebutkan pola penggunaan kata dapat digunakan untuk menemukan arti kata [74]. Contoh sederhana, kata yang muncul pada "**konteks**" yang sama cenderung memiliki makna yang sama. Perhatikan "konteks" dalam artian NLP adalah kata-kata sekitar (*surrounding words*);[10] contohnya kalimat "budi menendang bola", "konteks" dari "bola" adalah "budi menendang". Kata "cabai" dan "permen" pada kedua kalimat "budi suka cabai" dan "budi suka permen" memiliki kaitan makna, dalam artian keduanya muncul pada konteks yang sama. Sebagai manusia, kita tahu ada keterkaitan antara "cabai" dan "permen" karena keduanya bisa dimakan.

Berdasarkan hipotesis tersebut, kita dapat mentransformasi kata menjadi sebuah bentuk matematis dimana kata direpresentasikan oleh pola penggunaannya [64]. Arti kata ***embedding*** adalah transformasi kata (beserta konteksnya) menjadi bentuk matematis (vektor), i.e., mirip/sama dengan *coding*. "Kedekatan hubungan makna" (***semantic relationship***) antarkata kita harapkan dapat tercermin pada operasi vektor. Salah satu metode sederhana untuk merepresentasikan kata sebagai vektor adalah ***Vector Space Model***. Konsep *embedding* dan *autoencoder* sangatlah dekat, tapi kami ingin menakankan bahwa *embedding* adalah bentuk representasi konteks.

[9] https://wordnet.princeton.edu/

[10] Selain *surrounding words*, konteks dalam artian NLP dapat juga berupa kalimat, paragraph, atau dokumen.

	Dokumen 1	Dokumen 2	Dokumen 3	Dokumen 4	⋯
King	1	0	0	0	⋯
Queen	0	1	0	1	⋯
Prince	1	0	1	0	⋯
Princess	0	1	0	1	⋯
⋯					

Tabel 12.1: Contoh 1-of-V encoding.

Semantic relationship dapat diartikan sebagai ***attributional*** atau ***relational similarity***. *Attributional similarity* berarti dua kata memiliki atribut/sifat yang sama, misalnya anjing dan serigala sama-sama berkaki empat, menggongong, serta mirip secara fisiologis. *Relational similarity* berarti derajat korespondensi, misalnya *anjing* : *menggongong* memiliki hubungan yang erat dengan *kucing* : *mengeong*.

12.5.1 Vector Space Model

Vector space model (VSM)[11] adalah bentuk *embedding* yang relatif sudah cukup lama tapi masih digunakan sampai saat ini. Pada pemodelan ini, kita membuat sebuah matriks dimana baris melambangkan kata, kolom melambangkan dokumen. Metode VSM ini selain mampu menangkap hubungan antarkata juga mampu menangkap hubungan antardokumen (*to some degree*). Asal muasalnya adalah *statistical semantics hypothesis*. Tiap sel pada matriks berisi nilai 1 atau 0. 1 apabila $kata_i$ muncul di $dokumen_i$ dan 0 apabila tidak. Model ini disebut ***1-of-V/1-hot encoding*** dimana V adalah ukuran kosa kata. Ilustrasi dapat dilihat pada Tabel 12.1.

Akan tetapi, *1-of-V encoding* tidak menyediakan banyak informasi untuk kita. Dibanding sangat ekstrim saat mengisi sel dengan nilai 1 atau 0 saja, kita dapat mengisi sel dengan frekuensi kemunculan kata pada dokumen, disebut ***term frequency*** (TF). Apabila suatu kata muncul pada banyak dokumen, kata tersebut relatif tidak terlalu "penting" karena muncul dalam berbagai konteks dan tidak mampu membedakan hubungan dokumen satu dan dokumen lainnya (***inverse document frequency***/IDF). Formula IDF diberikan pada persamaan 12.10. Tingkat kepentingan kata berbanding terbalik dengan jumlah dokumen dimana kata tersebut dimuat. N adalah banyaknya dokumen, $|d \epsilon D; t \epsilon d|$ adalah banyaknya dokumen dimana kata t muncul.

$$IDF(t, D) = \log\left(\frac{N}{|d \epsilon D; t \epsilon d|}\right) \tag{12.10}$$

Dengan menggunakan perhitungan TF-IDF yaitu $TF \times IDF$ untuk mengisi sel pada matriks Tabel 12.1, kita memiliki lebih banyak informasi. TF-IDF

[11] Mohon bedakan dengan VSM (*vector space model*) dan SVM (*support vector machine*)

sampai sekarang menjadi *baseline* pada ***information retrieval***. Misalkan kita ingin menghitung kedekatan hubungan antar dua dokumen, kita hitung ***cosine distance*** antara kedua dokumen tersebut (vektor suatu dokumen disusun oleh kolom pada matriks). Apabila kita ingin menghitung kedekatan hubungan antar dua kata, kita hitung *cosine distance* antara kedua kata tersebut dimana vektor suatu kata merupakan baris pada matriks. Tetapi seperti intuisi yang mungkin kamu miliki, mengisi *entry* dengan nilai TF-IDF pun akan menghasilkan *sparse matrix*.

Statistical semantics hypothesis diturunkan lagi menjadi empat macam hipotesis [74]:

1. *Bag of words*
2. *Distributional hypothesis*
3. *Extended distributional hypothesis*
4. *Latent relation hypothesis*

Silakan pembaca mencari sumber tersendiri untuk mengerti keempat hipotesis tersebut atau membaca *paper* Turney dan Pantel [74].

12.5.2 Sequential, Time Series dan Compositionality

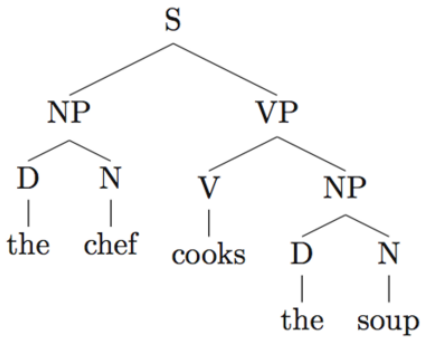
Bahasa manusia memiliki dua macam karakteristik yaitu adalah data berbentuk ***sequential data*** dan memenuhi sifat ***compositionality***. *Sequential data* adalah sifat data dimana suatu kemunculan $data_i$ dipengaruhi oleh data sebelumnya ($data_{i-1}, data_{i-2}, ...$). Perhatikan kedua kalimat berikut:

1. Budi melempar bola.
2. Budi melempar gedung bertingkat.

Pada kedua kalimat tersebut, kalimat pertama lebih masuk akal karena bagaimana mungkin seseorang bisa melempar gedung bertingkat. Keputusan kita dalam memilih kata berikutnya dipengaruhi oleh kata-kata sebelumnya, dalam hal ini "Budi melempar" setelah itu yang lebih masuk akal adalah "bola". Contoh lain adalah data yang memiliki sifat ***time series*** yaitu gelombang laut, angin, dan cuaca. Kita ingin memprediksi data dengan rekaman masa lalu, tapi kita tidak mengetahui masa depan. Kita mampu memprediksi cuaca berdasarkan rekaman parameter cuaca pada hari-hari sebelumnya. Ada yang berpendapat beda *time series* dan *sequential* (sekuensial) adalah diketahuinya sekuens kedepan secara penuh atau tidak. Penulis tidak dapat menyebutkan *time series* dan sekuensial sama atau beda, silahkan pembaca menginterpretasikan secara bijaksana.

Data yang memenuhi sifat ***compositionality*** berarti memiliki struktur hirarkis. Struktur hirarkis ini menggambarkan bagaimana unit-unit lebih kecil berinteraksi sebagai satu kesatuan. Artinya, interpretasi/pemaknaan unit yang lebih besar dipengaruhi oleh interpretasi/pemaknaan unit lebih kecil

(subunit). Sebagai contoh, kalimat "saya tidak suka makan cabai hijau". Unit "cabai" dan "hijau" membentuk suatu frasa "cabai hijau". Mereka tidak bisa dihilangkan sebagai satu kesatuan makna. Kemudian interaksi ini naik lagi menjadi kegiatan "makan cabai hijau" dengan keterangan "tidak suka", bahwa ada seseorang yang "tidak suka makan cabai hijau" yaitu "saya". Pemecahan kalimat menjadi struktur hirarkis berdasarkan *syntactical role* disebut ***constituent parsing***, contoh lebih jelas pada Gambar 12.6. *N* adalah *noun*, *D* adalah *determiner*, *NP* adalah *noun phrase*, *VP* adalah *verb phrase*, dan *S* adalah *sentence*. Selain bahasa manusia, gambar juga memiliki struktur hirarkis. Sebagai contoh, gambar rumah tersusun atas tembok, atap, jendela, dan pintu. Tembok, pintu, dan jendela membentuk bagian bawah rumah; lalu digabung dengan atap sehingga membentuk satu kesatuan rumah.



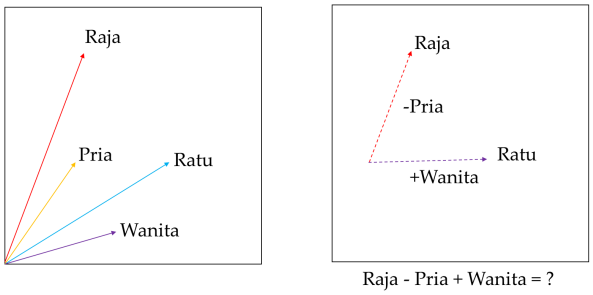
Gambar 12.6: Contoh constituent tree.[12]

12.5.3 Distributed Word Representation

Seperti yang disebutkan pada bagian sebelumnya, kita ingin hubungan kata (yang diinferensi dari konteksnya) dapat direpresentasikan sebagai operasi vektor seperti pada ilustrasi Gambar 12.7. Kata "raja" memiliki sifat-sifat yang dilambangkan oleh suatu vektor (misal 90% aspek loyalitas, 80% kebijaksanaan, 90% aspek kebangsaan, dst), begitu pula dengan kata "pria", "wanita", dan "ratu". Jika sifat-sifat yang dimiliki "raja" dihilangkan bagian sifat-sifat "pria"-nya, kemudian ditambahkan sifat-sifat "wanita" maka idealnya operasi ini menghasilkan vektor yang dekat kaitannya dengan "ratu". Dengan kata lain, raja yang tidak maskulin tetapi fenimin disebut ratu. Seperti yang disebutkan sebelumnya, ini adalah tujuan utama *embedding* yaitu merepresentasikan "makna" kata sebagai vektor sehingga kita dapat memanipulasi banyak hal berdasarkan operasi vektor. Hal ini mirip (tetapi tidak sama)

[12] source: Pinterest

dengan prinsip *singular value decomposition* dan *autoencoder* yang telah dijelaskan sebelumnya.

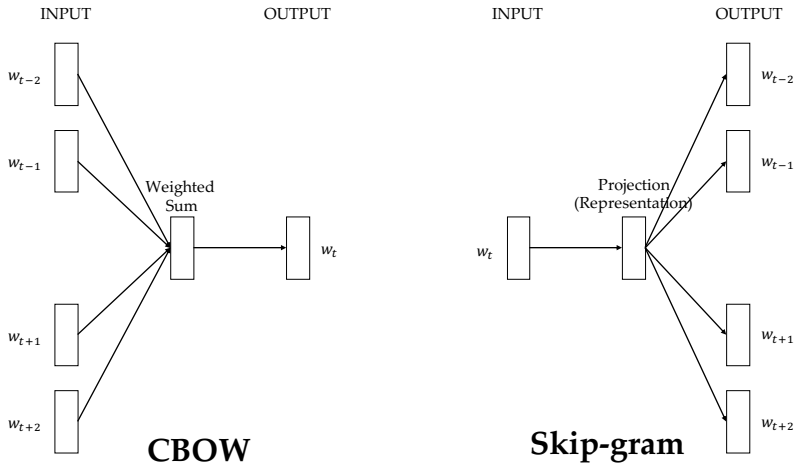


Gambar 12.7: Contoh operasi vektor kata.

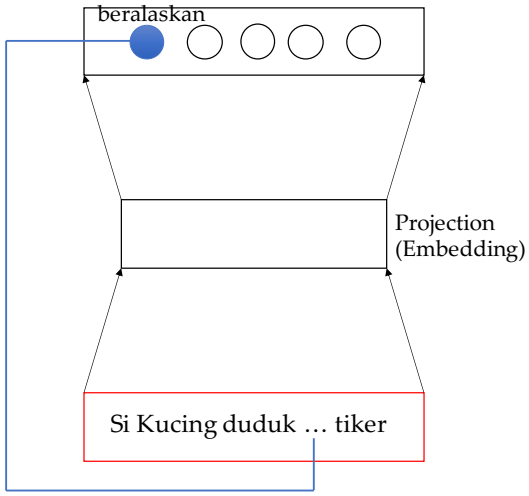
Selain *vector space model*, apakah ada cara lain yang mampu merepresentasikan kata dengan lebih baik? Salah satu kekurangan VSM adalah tidak memadukan sifat sekuensial pada konstruksi vektornya. Cara lebih baik ditemukan oleh [48, 49] dengan ekstensi pada [70]. Idenya adalah menggunakan teknik *representation learning* dan prinsip *statistical semantics hypothesis*. Metode ini lebih dikenal dengan sebutan *word2vec*. Tujuan *word2vec* masih sama, yaitu merepresentasikan kata sebagai vektor, sehingga kita dapat melakukan operasi matematis terhadap kata. *Encoder*-nya berbentuk ***Continous bag of words*** **(CBOW)** atau **Skip-gram**. Pada CBOW, kita memprediksi kata diberikan suatu "konteks". Pada arsitektur "Skip-gram" kita memprediksi konteks, diberikan suatu kata. Ilustrasi dapat dilihat pada Gambar 12.8. Bagian *projection layer* pada Gambar 12.8 adalah *coding* layer. Kami akan memberikan contoh CBOW secara lebih detil. Kedua arsitektur ini dapat dilatih menggunakan *one-hot encoding*, i.e., w_i merepresentasikan *one-hot encoding* untuk kata ke-i.

Perhatikan Gambar 12.9. Diberikan sebuah konteks "si kucing duduk ... tiker". Kita harus menebak apa kata pada "..." tersebut. Dengan menggunakan teknik *autoencoder*, *output layer* adalah distribusi probabilitas $kata_i$ pada konteks tersebut. Kata yang menjadi jawaban adalah kata dengan probabilitas terbesar, misalkan pada kasus ini adalah "beralaskan". Dengan arsitektur ini, prinsip sekuensial atau *time series* dan *statistical semantics hypothesis* terpenuhi (*to a certain extent*). Teknik ini adalah salah satu contoh penggunaan *neural network* untuk *unsupervised learning*. Kita tidak perlu mengkorespondensikan kata dan *output* yang sesuai karena *input vektor* didapat dari statistik penggunaan kata. Agar lebih tahu kegunaan vektor kata, kamu dapat mencoba kode dengan bahasa pemrograman Python 2.7 yang disediakan penulis.[13] Buku ini telah menjelaskan ide konseptual *word embedding* pada

[13] https://github.com/wiragotama/GloVe_Playground



Gambar 12.8: CBOW (*Continous bag of words*) vs Skip-gram, rekonstruksi [49].



Gambar 12.9: CBOW.

level abstrak, yaitu merepresentasikan kata dan konteksnya menjadi bentuk vektor. Apabila kamu tertarik untuk memahami detilnya secara matematis, kamu dapat membaca berbagai penelitian terkait.[14] Silahkan baca *paper* oleh Mikolov [48, 49] untuk detil implementasi *word embedding*.

12.5.4 Distributed Sentence Representation

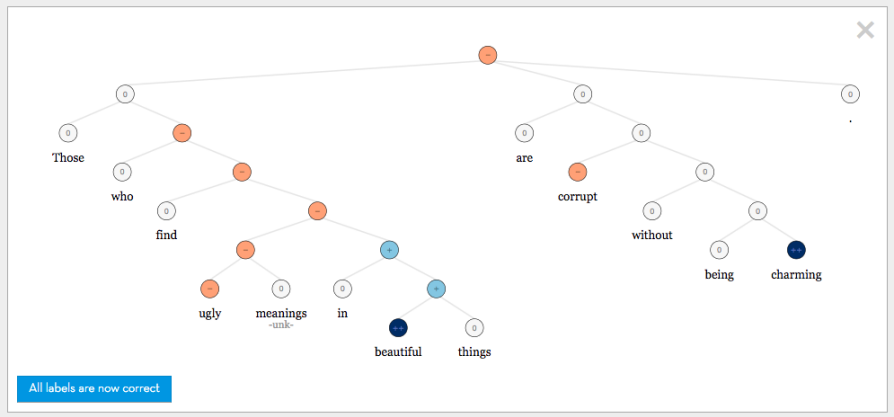
Kita sudah dapat merepresentasikan kata menjadi vektor, selanjutnya kita ingin mengonversi unit lebih besar (kalimat) menjadi vektor. Salah satu cara paling mudah adalah menggunakan nilai rata-rata representasi *word embedding* untuk semua kata yang ada pada kalimat tersebut (*average of its individual word embeddings*). Cara ini sering digunakan pada bidang NLP dan cukup *powerful*, sebagai contoh pada *paper* oleh Putra dan Tokunaga [75]. Pada NLP, sering kali kalimat diubah terlebih dahulu menjadi vektor sebelum dilewatkan pada algoritma *machine learning*, misalnya untuk analisis sentimen (kalimat bersentimen positif atau negatif). Vektor ini yang nantinya menjadi *feature vector* bagi algoritma *machine learning*.

Kamu sudah tahu bagaimana cara mengonversi kata menjadi vektor, untuk mengonversi kalimat menjadi vektor cara sederhananya adalah merata-ratakan nilai vektor kata-kata pada kalimat tersebut. Tetapi dengan cara sederhana ini, sifat sekuensial dan *compositional* pada kalimat tidak terpenuhi. Sebagai contoh, kalimat "anjing menggigit Budi" dan "Budi menggigit anjing" akan direpresentasikan sebagai vektor yang sama karena terdiri dari kata-kata yang sama. Dengan demikian, representasi kalimat sederhana dengan merata-ratakan vektor kata-katanya juga tidaklah sensitif terhadap urutan.[15] Selain itu, rata-rata tidak sensitif terhadap *compositionality*. Misal frase "bukan sebuah pengalaman baik" tersusun atas frase "bukan" yang diikuti oleh "sebuah pengalaman baik". Rata-rata tidak mengetahui bahwa "bukan" adalah sebuah *modifier* untuk sebuah frase dibelakangnya. Sentimen dapat berubah bergantung pada komposisi kata-katanya (contoh pada Gambar 12.10).

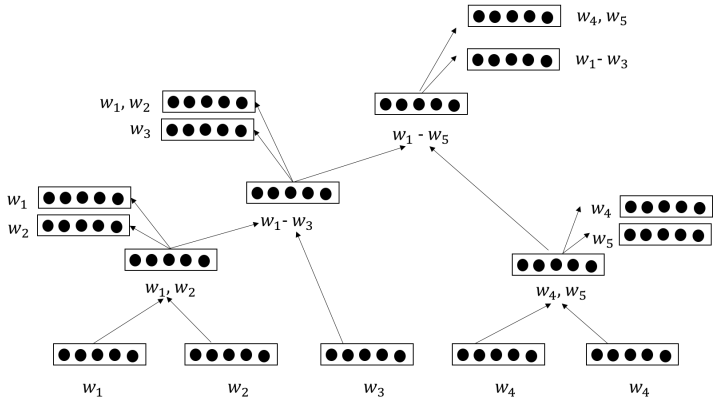
Cara lainnya adalah meng-*encode* kalimat sebagai *vektor* menggunakan *recursive autoencoder*. *Recursive* berarti suatu bagian adalah komposisi dari bagian lainnya. Penggunaan *recursive autoencoder* sangat rasional berhubung data memenuhi sifat *compositionality* yang direpresentasikan dengan baik oleh topologi *recursive neural network*. Selain itu, urutan susunan kata-kata juga tidak hilang. Untuk melatih *recursive autoencoder*, *output* dari suatu layer adalah rekonstruksi *input*, ilustrasi dapat dilihat pada Gambar 12.11. Pada setiap langkah *recursive*, *hidden layer/coding layer* berusaha men-*decode* atau merekonstruksi kembali vektor *input*.

Lebih jauh, untuk sentimen analisis pada kata, kita dapat menambahkan output pada setiap *hidden layer*, yaitu sentimen unit gabungan, seperti pada

[14] Beberapa orang berpendapat bahwa *evil is in the detail*.
[15] Karena ini *recurrent neural network* bagus untuk *language modelling*.
[16] http://nlp.stanford.edu:8080/sentiment/rntnDemo.html



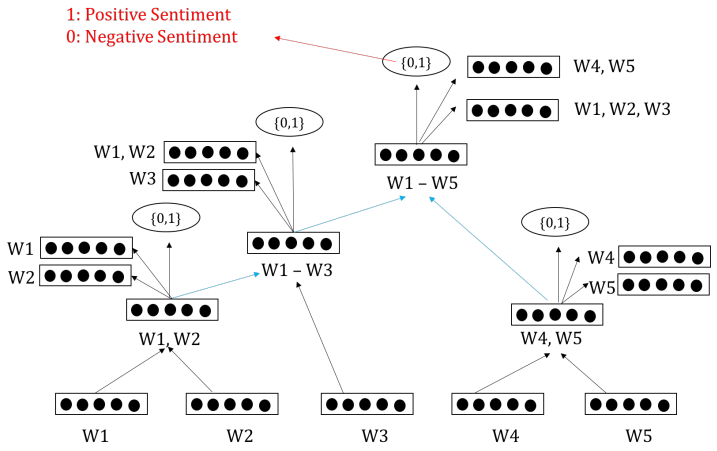
Gambar 12.10: Contoh analisis sentimen (Stanford).[16]



Gambar 12.11: Contoh recursive autoencoder.

Gambar 12.12. Selain menggunakan *recursive autoencoder*, kamu juga dapat menggunakan *recurrent autoencoder*. Kami silahkan pada pembaca untuk memahami *recurrent autoencoder*. Prinsipnya mirip dengan *recursive autoencoder*.

Teknik yang disampaikan mampu mengonversi kalimat menjadi vektor, lalu bagaimana dengan paragraf, satu dokumen, atau satu frasa saja? Teknik umum untuk mengonversi teks menjadi vektor dapat dibaca pada [69] yang lebih dikenal dengan nama ***paragraph vector*** atau ***doc2vec***.



Gambar 12.12: Contoh recursive autoencoder dengan sentiment[67].

12.6 Tips

Bab ini menyampaikan penggunaan *neural network* untuk melakukan *kompresi data* (*representation learning*) dengan teknik *unsupervised learning*. Hal yang lebih penting untuk dipahami bahwa ilmu *machine learning* tidak berdiri sendiri. Walaupun kamu menguasai teknik *machine learning* tetapi tidak mengerti domain dimana teknik tersebut diaplikasikan, kamu tidak akan bisa membuat *learning machine* yang memuaskan. Contohnya, pemilihan fitur *machine learning* pada teks (NLP) berbeda dengan gambar (*computer vision*). Mengerti *machine learning* tidak semata-mata membuat kita bisa menyelesaikan semua macam permasalahan. Tanpa pengetahuan tentang domain aplikasi, kita bagaikan orang buta yang ingin menyetir sendiri!

Soal Latihan

12.1. Penggunaan Autoencoder untuk Arsitektur Kompleks

(a) Pada bab ini, telah dijelaskan bahwa kita dapat menginisialisasi arsitektur *neural network* yang kompleks menggunakan *autoencoder*. Jelaskan pada kasus apa kita dapat melakukan hal tersebut!
(b) Jelaskan mengapa menginisiasi (sebagian) arsitektur kompleks menggunakan *autoencoder* adalah sesuatu yang masuk akal!

12.2. LSI dan LDA

(a) Jelaskanlah Latent Semantic Indexing (LSI) dan Latent Dirichlet Allocation (LDA)!

(b) Apa persamaan dan perbedaan antara LSI, LDA, dan *autoencoder*?

12.3. Variational Autoencoder

Jelaskan apa itu *variational autoencoder*! Deskripsikan perbedaannya dengan *autoencoder* yang sudah dijelaskan pada bab ini?