Bagian III

Neural Networks

11

Artificial Neural Network

> "If you want to make information stick, it's best to learn it, go away from it for a while, come back to it later, leave it behind again, and once again return to it - to engage with it deeply across time. Our memories naturally degrade, but each time you return to a memory, you reactivate its neural network and help to lock it in."
>
> Joshua Foer

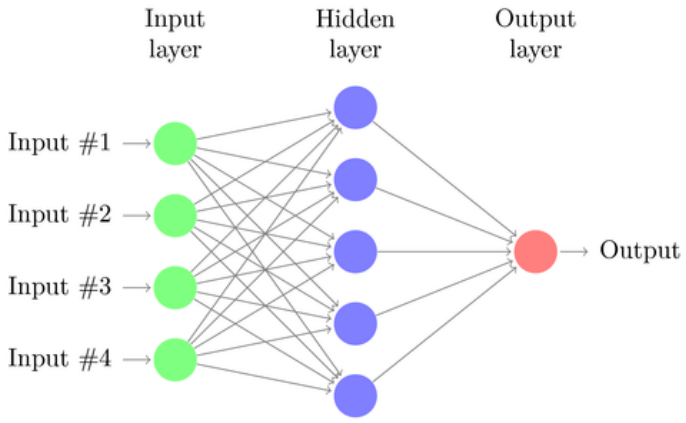
Bab ini membahas salah satu algoritma *machine learning* yang sedang populer belakangan ini, yaitu *artificial neural network*. Pembahasan dimulai dari hal-hal sederhana sampai yang lebih kompleks. Bab ini lebih berfokus pada penggunaan *artifical neural network* untuk *supervised learning*.

11.1 Definisi

Masih ingatkah Anda materi pada bab-bab sebelumnya? *Machine learning* sebenarnya meniru bagaimana proses manusia belajar. Pada bagian ini, peneliti ingin meniru proses belajar tersebut dengan mensimulasikan jaringan saraf biologis (*neural network*) [39, 40, 41, 42]. Kami yakin banyak yang sudah tidak asing dengan istilah ini, berhubung *deep learning* sedang populer dan banyak yang membicarakannya (dan digunakan sebagai trik pemasaran). *Artificial neural network* adalah salah satu algoritma *supervised learning* yang populer dan bisa juga digunakan untuk *semi-supervised* atau *unsupervised learning* [40, 42, 43, 44, 45]. Walaupun tujuan awalnya adalah untuk mensimulasikan jaringan saraf biologis, jaringan tiruan ini sebenenarnya simulasi

yang terlalu disederhanakan, artinya simulasi yang dilakukan tidak mampu menggambarkan kompleksitas jaringan biologis manusia[1].

Artificial Neural Network (selanjutnya disingkat ANN), menghasilkan model yang sulit dibaca dan dimengerti oleh manusia karena memiliki banyak layer (*multilayer perceptron*) dan sifat *non-linearity* (fungsi aktivasi). Pada bidang riset ini, ANN disebut agnostik–kita percaya, tetapi sulit membuktikan kenapa konfigurasi parameter yang dihasilkan *training* bisa benar. Konsep matematis ANN itu sendiri cukup *solid*, tetapi *interpretability* model rendah menyebabkan kita tidak dapat menganalisa proses inferensi yang terjadi pada model ANN. Secara matematis, ANN ibarat sebuah graf. ANN memiliki neuron/*node* (*vertex*), dan sinapsis (*edge*). Topologi ANN akan dibahas lebih detil subbab berikutnya. Sebagai gambaran, ANN berbentuk seperti Gambar 11.1. Walaupun memiliki struktur seperti graf, operasi pada ANN paling baik dan mudah dijelaskan dalam notasi aljabar linear.



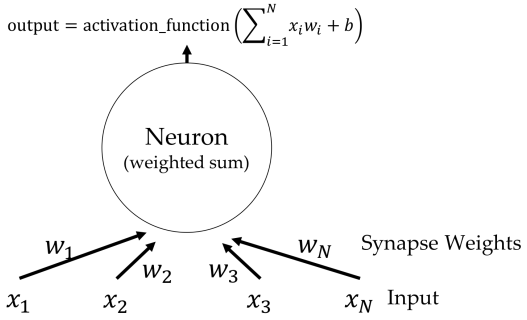
Gambar 11.1. *Multilayer Perceptron*

11.2 Single Perceptron

Bentuk terkecil (minimal) sebuah ANN adalah *single perceptron* yang hanya terdiri dari sebuah neuron. Sebuah neuron diilustrasikan pada Gambar 11.2. Secara matematis, terdapat *feature vector* $\mathbf{x}$ yang menjadi *input* bagi neuron tersebut. Neuron akan memproses *input* $\mathbf{x}$ melalui perhitungan jumlah perkalian antara nilai *input* dan *synapse weight*, yang dilewatkan pada **fungsi non-linear** [46, 47, 4]. Pada *training*, yang dioptimasi adalah nilai *synapse*

[1] `Quora: why is Geoffrey Hinton suspicious of backpropagation and wants AI to start over`

weight (*learning parameter*). Selain itu, terdapat juga bias b sebagai kontrol tambahan (ingat materi *steepest gradient descent*). *Output* dari neuron adalah hasil fungsi aktivasi dari perhitungan jumlah perkalian antara nilai *input* dan *synapse weight*. Ada beberapa macam fungsi aktivasi, misal ***step function***, ***sign function***, ***rectifier*** dan ***sigmoid function***. Untuk selanjutnya, pada bab ini, fungsi aktivasi yang dimaksud adalah jenis ***sigmoid function***. Silahkan eksplorasi sendiri untuk fungsi aktivasi lainnya. Salah satu bentuk tipe ***sigmoid function*** diberikan pada persamaan 11.1. Bila di-*plot* menjadi grafik, fungsi ini memberikan bentuk seperti huruf S.



Gambar 11.2. *Single Perceptron*

$$\sigma(u) = \frac{1}{1 + e^{-u}} \tag{11.1}$$

Perhatikan kembali, Gambar 11.2 sesungguhnya adalah operasi aljabar linear. Single perceptron dapat dituliskan kembali sebagai 11.2.

$$o = f(\mathbf{x} \cdot \mathbf{w} + b) \tag{11.2}$$

dimana o adalah *output* dan f adalah fungsi **non-linear yang dapat diturunkan secara matematis** (*differentiable non-linear function* – selanjutnya disebut "fungsi non-linear" saja.). Bentuk ini tidak lain dan tidak bukan adalah persamaan model linear yang ditransformasi dengan fungsi non-linear. Secara filosofis, ANN bekerja mirip dengan model linear, yaitu mencari *decision boudary*. Apabila beberapa model non-linear ini digabungkan, maka kemampuannya akan menjadi lebih hebat (subbab berikutnya).

Untuk melakukan pembelajaran *single perceptron*, *training* dilakukan menggunakan ***perceptron training rule***. Prosesnya sebagai berikut [4, 46, 47]:

1. Inisiasi nilai *synapse weights*, bisa *random* ataupun dengan aturan tertentu. Untuk heuristik aturan inisiasi, ada baiknya membaca buku referensi [1, 11].

2. Lewatkan input pada neuron, kemudian kita akan mendapatkan nilai *output*. Kegiatan ini disebut ***feedforward***.
3. Nilai *output* (*actual output*) tersebut dibandingkan dengan *desired output*.
4. Apabila nilai *output* sesuai dengan *desired output*, tidak perlu mengubah apa-apa.
5. Apabila nilai *output* tidak sesuai dengan *desired output*, hitung nilai *error* kemudian lakukan perubahan terhadap *learning parameter* (*synapse weight*).
6. Ulangi langkah-langkah ini sampai tidak ada perubahan nilai *error*, nilai *error* kurang dari sama dengan suatu *threshold* (biasanya mendekati 0), atau sudah mengulangi proses latihan sebanyak T kali (*threshold*).

Error function diberikan pada persamaan 11.3 dan perubahan *synapse weight* diberikan pada persamaan 11.4, dimana y melambangkan *desired output*[2], $f(\mathbf{x}, \mathbf{w})$ melambangkan *actual output* untuk $\mathbf{x}$ sebagai input. η disebut sebagai *learning rate* [3].

$$E(\mathbf{w}) = (y - f(\mathbf{x}, \mathbf{w}))^2 \tag{11.3}$$

$$\Delta w_i = \eta(y - o)x_i \tag{11.4}$$

Hasil akhir pembelajaran adalah konfigurasi *synapse weight*. Saat klasifikasi, kita melewatkan *input* baru pada jaringan yang telah dibangun, kemudian tinggal mengambil hasilnya. Pada contoh kali ini, seolah-olah *single perceptron* hanya dapat digunakan untuk melakukan *binary classification* (hanya ada dua kelas, nilai 0 dan 1). Untuk *multi-label classification*, kita dapat menerapkan berbagai strategi. Metode paling umum adalah melewatkan *output neurons* pada fungsi softmax[4], sehingga mendapatkan nilai distribusi probabilitas untuk tiap kelas.

11.3 Permasalahan XOR

Sedikit sejarah, *perceptron* sebenarnya cukup populer sekitar tahun 1950-1960. Entah karena suatu sebab, *perceptron* menjadi tidak populer dan digantikan oleh model linear. Saat itu, belum ada algoritma yang bekerja dengan relatif bagus untuk melatih *perceptron* yang digabungkan (*multilayer perceptron*). Model linear mendapat popularitas hingga kira-kira dapat disebut sekitar tahun 1990'an atau awal 2000'an. Berkat penemuan *backpropagation* sekitar awal 1980[5], *multilayer perceptron* menjadi semakin populer. Perlu dicatat, komunitas riset bisa jadi seperti cerita ini. Suatu teknik yang baru belum

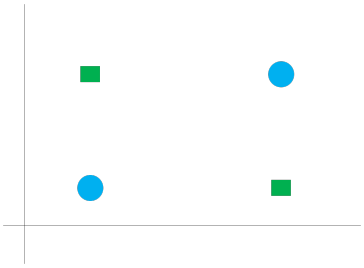
[2] Pada contoh ini, kebetulan jumlah neuron output hanyalah satu.
[3] Pada umumnya, kita tidak menggunakan satu data, tetapi *batch-sized*.
[4] Sudah dibahas pada model linear.
[5] `http://people.idsia.ch/~juergen/who-invented-backpropagation.html`

tentu bisa segera diimplementasikan karena beberapa kendala (misal kendala kemampuan komputasi).

Pada bab-bab sebelumnya, kamu telah mempelajari model linear dan model probabilistik. Kita ulangi kembali contoh data yang bersifat *non-linearly separable*, yaitu XOR yang operasinya didefinisikan sebagai:

- $\text{XOR}(0, 0) = 0$
- $\text{XOR}(1, 0) = 1$
- $\text{XOR}(0, 1) = 1$
- $\text{XOR}(1, 1) = 0$

Ilustrasinya dapat dilihat pada Gambar 11.3. Jelas sekali, XOR ini adalah fungsi yang tidak dapat diselesaikan secara langsung oleh model linear.



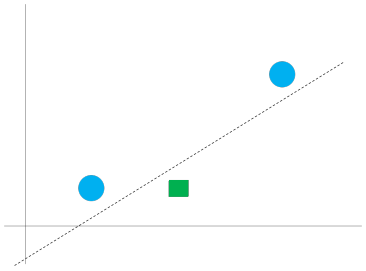
Gambar 11.3. Permasalahan XOR

Seperti yang diceritakan pada bab model linear, solusi permasalahan ini adalah dengan melakukan transformasi data menjadi *linearly-separable*, misalnya menggunakan fungsi non-linear pada persamaan 11.5 dimana (x, y) adalah absis dan ordinat. Hasil transformasi menggunakan fungsi ini dapat dilihat pada Gambar 11.4. Jelas sekali, data menjadi *linearly separable*.

$$\phi(x, y) = (x \times y, x + y) \tag{11.5}$$

Sudah dijelaskan pada bab model linear, permasalahan yang ada tidak sesederhana ini (kebetulan ditransformasikan menjadi data dengan dimensi yang sama). Pada permasalahan praktis, kita harus mentransformasi data menjadi dimensi lebih tinggi (dari 2D menjadi 3D). Berbeda dengan ide utama linear model/*kernel method* tersebut, **prinsip ANN adalah untuk melewatkan data pada fungsi non-linear** (*non-linearities*). Sekali lagi penulis ingin tekankan, ANN secara filosofis adalah ***trainable non-linear mapping functions***. ANN mampu mentransformasi data ke *space*/ruang konsep yang berbeda (bisa pada dimensi lebih tinggi atau lebih rendah), lalu mencari *non-linear decision boundary* dengan *non-linear functions*. Interaksi antar-fitur juga dapat dimodelkan secara non-linear.

Perlu dicatat, pemodelan non-linear inilah yang membuat ANN menjadi hebat. ANN mungkin secara luas didefinisikan mencakup *perceptron* tetapi

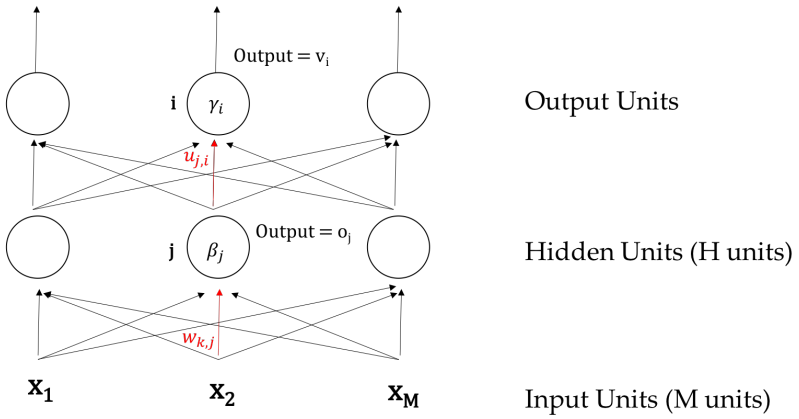


Gambar 11.4. XOR ditransformasi. Segiempat berwarna hijau sebenarnya melambangkan dua instans (yang setelah ditransformasi kebetulan berlokasi pada tempat yang sama)

secara praktis, **ANN sebenarnya mengacu pada *multilayer perceptron* dan arsitektur lebih kompleks** (dijelaskan pada subbab berikutnya). Pada masa ini, hampir tidak ada lagi yang menggunakan *single perceptron*. Untuk bab-bab kedepan, ketika kami menyebut ANN maka yang diacu adalah *multilayer perceptron* dan arsitektur lebih kompleks (*single perceptron* di-*exclude*). Hal ini disebabkan oleh *single perceptron* tidak dapat mempelajari *XOR function* secara independen tanpa *feature engineering*, sementara *multilayer perceptron* bisa [48].

11.4 Multilayer Perceptron

Kamu sudah belajar *training* untuk *single perceptron*. Selanjutnya kita akan mempelajari *multilayer perceptron* (MLP) yang juga dikenal sebagai ***feed forward neural network***. Kami tekankan sekali lagi, **istilah "ANN" selanjutnya mengacu pada MLP dan arsitektur lebih kompleks**. Seperti ilustrasi pada Gambar 11.5, *multilayer perceptron* secara literal memiliki beberapa *layers*. Pada *lecture note* ini, secara umum ada 3 *layers*: *input*, *hidden*, dan *output layer*. *Input layer* menerima *input* (tanpa melakukan operasi apapun), kemudian nilai *input* (tanpa dilewatkan ke fungsi aktivasi) diberikan ke *hidden units*. Pada *hidden units*, *input* diproses dan dilakukan perhitungan hasil fungsi aktivasi untuk tiap-tiap neuron, lalu hasilnya diberikan ke *layer* berikutnya. Hasil dari *input layer* akan diterima sebagai input bagi *hidden layer*. Begitupula seterusnya *hidden layer* akan mengirimkan hasilnya untuk *output layer*. Kegiatan ini dinamakan *feed forward* [40, 4]. Hal serupa berlaku untuk *artificial neural network* dengan lebih dari 3 *layers*. Parameter neuron dapat dioptimisasi menggunakan metode *gradient-based optimization* (dibahas pada subabb berikutnya, ingat kembali bab 5). Perlu diperhatikan, MLP adalah gabungan dari banyak fungsi non-linear. Seperti yang disampaikan pada subbab sebelumnya, gabungan banyak fungsi non-linear ini lebih hebat dibanding *single perceptron*.



Gambar 11.5. *Multilayer Perceptron* 2

$$o_j = \sigma\left(\sum_{k=1}^{M} x_k w_{k,j} + \beta_j\right) \tag{11.6}$$

$$v_i = \sigma\left(\sum_{j=1}^{H} o_j u_{j,i} + \gamma_i\right) = \sigma\left(\sum_{j=1}^{H} \sigma\left(\sum_{k=1}^{M} x_k w_{k,j} + \beta_j\right) u_{j,i} + \gamma_i\right) \tag{11.7}$$

Perhatikan persamaan 11.6 dan 11.7 untuk menghitung *output* pada *layer* yang berbeda. u, w adalah *learning parameters*. β, γ melambangkan *noise* atau *bias*. M adalah banyaknya *hidden units* dan H adalah banyaknya *output units*. Persamaan 11.7 dapat disederhanakan penulisannya sebagai persamaan 11.8. Persamaan 11.8 terlihat relatif lebih "elegan". Seperti yang disebutkan pada subbab sebelumnya, ANN dapat direpresentasikan dengan notasi operasi aljabar.

$$\mathbf{v} = \sigma(\mathbf{oU} + \gamma) = \sigma((\sigma(\mathbf{xW} + \beta))\mathbf{U} + \gamma) \tag{11.8}$$

Untuk melatih MLP, algoritma yang umumnya digunakan adalah ***backpropagation*** [49]. Arti kata *backpropagation* sulit untuk diterjemahkan ke dalam bahasa Indonesia. Kita memperbaharui parameter (*synapse weights*) secara bertahap (dari *output* ke *input* layer, karena itu disebut *backpropagation*) berdasarkan *error*/*loss* (*output* dibandingkan dengan *desired output*). Intinya adalah mengkoreksi *synapse weight* dari *output layer* ke *hidden layer*, kemudian *error* tersebut dipropagasi ke layer sebelum-sebelumnya. Artinya, perubahan *synapse weight* pada suatu layer dipengaruhi oleh perubahan *synapse weight* pada layer setelahnya[6]. *Backpropagation* tidak lain

[6] Kata "setelah" mengacu *layer* yang menuju *output layer*, "sebelum" mengacu layer yang lebih dekat dengan *input layer*.

dan tidak bukan adalah metode *gradient-based optimization* yang diterapkan pada ANN.

Pertama-tama diberikan pasangan *input* ($\mathbf{x}$) dan *desired output* ($\mathbf{y}$) sebagai *training data*. Untuk meminimalkan *loss*, algoritma *backpropagation* menggunakan prinsip *gradient descent* (ingat kembali materi bab model linear). Kamu akan memperlajari bagaimana cara menurunkan *backpropagation* menggunakan teknik *gradient descent*, yaitu menghitung *loss* ANN pada Gambar 11.5 **yang menggunakan fungsi aktivasi sigmoid**. Untuk fungsi aktivasi lainnya, pembaca dapat mencoba menurunkan persamaan sendiri!

Ingat kembali *chain rule* pada perkuliahan diferensial

$$f(g(x))' = f'(g(x))g'(x). \tag{11.9}$$

Ingat kembali *error*, untuk MLP diberikan oleh persamaan 11.3 (untuk satu data *point*), dimana I adalah banyaknya output neuron dan θ adalah kumpulan *weight matrices* (semua parameter pada MLP).

$$E(\theta) = \frac{1}{I}\sum_{i=1}^{I}(y_i - v_i)^2 \tag{11.10}$$

Mari kita lakukan proses penurunan untuk melatih MLP. *Error/loss* diturunkan terhadap tiap *learning parameter*.

Diferensial $u_{j,i}$ diberikan oleh turunan *sigmoid function*

$$\begin{aligned}\frac{\delta E(\theta)}{\delta u_{j,i}} &= (y_i - v_i)\frac{\delta v_i}{\delta u_{j,i}} \\ &= (y_i - v_i)v_i(1 - v_i)o_j\end{aligned}$$

Diferensial $w_{k,j}$ diberikan oleh turunan *sigmoid function*

$$\begin{aligned}\frac{\delta E(\theta)}{\delta w_{k,j}} &= \sum_{i=1}^{H}(y_i - v_i)\frac{\delta v_i}{\delta w_{k,j}} \\ &= \sum_{i=1}^{H}(y_i - v_i)\frac{\delta v_i}{\delta o_j}\frac{\delta o_j}{\delta w_{k,j}} \\ &= \sum_{i=1}^{H}(y_i - v_i)(v_i(1 - v_i)u_{j,i})(o_j(1 - o_j)x_k)\end{aligned}$$

Perhatikan, diferensial $w_{k,j}$ memiliki $\sum$ sementara $u_{j,i}$ tidak ada. Hal ini disebabkan karena $u_{j,i}$ hanya berkorespondensi dengan satu *output* neuron. Sementara $w_{k,j}$ berkorespondensi dengan banyak *output* neuron. Dengan kata lain, nilai $w_{k,j}$ mempengaruhi hasil operasi yang terjadi pada banyak *output* neuron, sehingga banyak neuron mempropagasi *error* kembali ke $w_{k,j}$.

Metode penurunan serupa dapat juga digunakan untuk menentukan perubahan β dan γ. Jadi proses *backpropagation* untuk kasus Gambar 11.5 dapat

diberikan seperti pada Gambar 11.6 dimana η adalah *learning rate*. Untuk *artificial neural network dengan* lebih dari 3 *layers*, kita pun bisa menurunkan persamaannya. Secara umum, proses melatih ANN (apapun variasi arsitekturnya) mengikuti *framework perceptron training rule* (subbab 11.2).

(2) Hidden to Output

$$v_i = \sigma\left(\sum_{j=1}^{H} o_j u_{j,i} + \gamma_i\right)$$

(3) Output to Hidden

$$\delta_i = (y_i - v_i)v_i(1 - v_i)$$
$$\Delta u_{j,i} = -\eta(t)\delta_i o_j$$
$$\Delta\gamma_i = -\eta(t)\delta_i$$

(1) Input to Hidden Layer

$$o_j = \sigma\left(\sum_{k=1}^{K} x_k w_{k,j} + \beta_j\right)$$

(4) Hidden to Input

$$\varphi_j = \sum_{i=1}^{H} \delta_i u_{j,i} o_j(1 - o_j)$$
$$\Delta w_{k,j} = -\eta(t)\varphi_j x_k$$
$$\Delta\beta_j = -\eta(t)\varphi_j$$

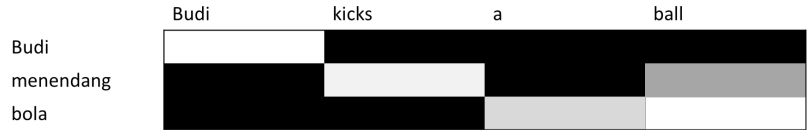
Gambar 11.6. Proses latihan MLP menggunakan *backpropagation*

11.5 Interpretability

Interpretability ada dua macam yaitu *model interpretability* (i.e., apakah struktur model pembelajaran mesin dapat dipahami) dan *prediction interpretability* (i.e., bagaimana memahami dan memverifikasi cara *input* dipetakan menjadi *output*) [50]. Contoh teknik pembelajaran mesin yang mudah diinterpretasikan baik secara struktur dan prediksinya adalah *decision tree* (bab 6.2). Struktur *decision tree* berupa pohon keputusan mudah dimengerti oleh manusia dan prediksi (keputusan) dapat dilacak (*trace*). Seperti yang sudah dijelaskan pada bagian pengantar, ANN (MLP) biasanya dianggap sebagai metode *black box* atau susah untuk dinterpretasikan (terutama *model interpretability*-nya). Hal ini disebabkan oleh kedalaman (*depth*) yaitu memiliki beberapa *layer* dan *non-linearities*. Suatu unit pada *output layer* dipengaruhi oleh kombinasi (*arbitrary combination*) banyak parameter pada *layers* sebelumnya yang dilewatkan pada fungsi non-linear. Sulit untuk mengetahui bagaimana pengaruh bobot suatu unit pada suatu *layer* berpengaruh pada *output layer*, beserta bagaimana pengaruh kombinasi bobot. Berbeda dengan model linear, kita tahu parameter (dan bobotnya) untuk setiap *input*. Salah satu arah riset adalah mencari cara agar keputusan (karena struktur lebih susah, setidaknya beranjak dari keputusan terlebih dahulu) yang dihasilkan oleh ANN dapat dijelaskan [51], salah satu contoh nyata adalah *attention*

mechanism [52, 53] (subbab 13.4.4) untuk *prediction interpretability*. Survey tentang *interpretability* dapat dibaca pada *paper* oleh Doshi-Velez dan Kim [54].

Cara paling umum untuk menjelaskan keputusan pada ANN adalah menggunakan *heat map*. Sederhananya, kita lewatkan suatu data $\mathbf{x}$ pada ANN, kemudian kita lakukan *feed-forward* sekali (misal dari *input* ke *hidden layer* dengan parameter $\mathbf{W}$). Kemudian, kita visualisasikan $\mathbf{x} \cdot \mathbf{W}$ (ilustrasi pada Gambar 11.7). Dengan ini, kita kurang lebih dapat mengetahui bagian *input* mana yang berpengaruh terhadap keputusan di *layer* berikutnya.



Gambar 11.7. Contoh *heat map* (*attention mechanism*) pada mesin translasi. Warna lebih terang menandakan bobot lebih tinggi. Sebagai contoh, kata "menendang" berkorespondensi paling erat dengan kata "kicks"

11.6 Binary Classification

Salah satu strategi untuk *binary classification* adalah dengan menyediakan hanya satu *output unit* di jaringan. Kelas pertama direpresentasikan dengan -1, kelas kedua direpresentasikan dengan nilai 1 (setelah diaktivasi). Hal ini dapat dicapai dengan fungsi non-linear seperti *sign*[7]. Apabila kita tertarik dengan probabilitas masuk ke dalam suatu kelas, kita dapat menggunakan fungsi seperti sigmoid[8] atau tanh[9].

11.7 Multi-label Classification

Multilayer perceptron dapat memiliki *output unit* berjumlah lebih dari satu. Seumpama kita mempunyai empat kelas, dengan demikian kita dapat merepresentasikan keempat kelas tersebut empat *output units*. Kelas pertama direpresentasikan dengan unit pertama, kelas kedua dengan unit kedua, dst. Untuk C kelas, kita dapat merepresentasikannya dengan C *output units*. Kita dapat merepresentasikan data harus dimasukkan ke kelas mana menggunakan

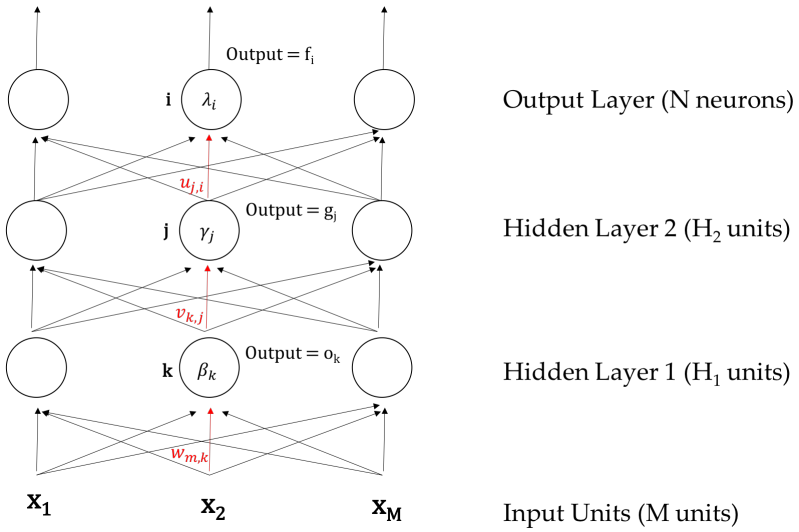
[7] `https://en.wikipedia.org/wiki/Sign_function`
[8] `https://en.wikipedia.org/wiki/Sigmoid_function`
[9] `https://en.wikipedia.org/wiki/Hyperbolic_function`

sparse vector, yaitu bernilai 0 atau 1. Elemen ke-i bernilai 1 apabila data masuk ke kelas c_i, sementara nilai elemen lainnya adalah 0 (ilurasi pada Gambar 11.8).*Output* ANN dilewatkan pada suatu fungsi softmax yang melambangkan probabilitas *class-assignment*; i.e., kita ingin *output* agar semirip mungkin dengan *sparse vector-desired output*. Pada kasus ini, *output* dari ANN adalah sebuah distribusi yang melambangkan *input* di-*assign* ke kelas tertentu. Ingat kembali materi bab 5, *cross entropy* cocok digunakan sebagai *utility function* ketika *output* berbentuk distribusi.

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Kelas pertama
Kelas kedua
Kelas ketiga
Kelas keempat

Gambar 11.8. Ilustrasi representasi *desired output* pada *multi-label classification*



Gambar 11.9. *Deep Neural Network*

11.8 Deep Neural Network

Deep Neural Network (DNN) adalah *artificial neural network* yang memiliki banyak *layer*. Pada umumnya, *deep neural network* memiliki lebih dari 3 *layers*

(*input layer*, *N hidden layers*, *output layer*), dengan kata lain adalah MLP dengan lebih banyak *layer*. Karena ada relatif banyak layer, disebutlah *deep*. Proses pembelajaran pada DNN disebut sebagai *deep learning*[10] [9]. Jaringan *neural network* pada DNN disebut *deep network*.

Perhatikan Gambar 11.9 yang memiliki 4 *layers*. Cara menghitung *final output* sama seperti MLP, diberikan pada persamaan 11.11 dimana β, γ, λ adalah *noise* atau *bias*.

$$f_i = \sigma\left(\sum_{j=1}^{H_2} u_{j,i}\sigma\left(\sum_{k=1}^{H_1} v_{k,j}\sigma\left(\sum_{m=1}^{M} x_m w_{m,k} + \beta_k\right) + \gamma_j\right) + \lambda_i\right) \tag{11.11}$$

Cara melatih *deep neural network*, salah satunya dapat menggunakan *backpropagation*. Seperti pada bagian sebelumnya, kita hanya perlu menurunkan rumusnya saja. **Penurunan diserahkan pada pembaca sebagai latihan**. Hasil proses penurunan dapat dilihat pada Gambar 11.10.

(3) Hidden 2 to Output

$$f_i = \sigma\left(\sum_{j=1}^{H_2} g_j u_{j,i} + \lambda_i\right)$$

(4) Output to Hidden 2

$$\delta_i = (y_i - f_i) f_i (1 - f_i)$$
$$\Delta u_{j,i} = -\eta(t)\delta_i g_j$$
$$\Delta\lambda_i = -\eta(t)\delta_i$$

(2) Hidden 1 to Hidden 2

$$g_j = \sigma\left(\sum_{k=1}^{H_1} o_k v_{k,j} + \gamma_j\right)$$

(5) Hidden 2 to Hidden 1

$$\varphi_j = \sum_{i=1}^{H_2} \delta_i u_{j,i} g_j(1 - g_j)$$
$$\Delta v_{k,j} = -\eta(t)\varphi_j o_k$$
$$\Delta\gamma_j = -\eta(t)\varphi_j$$

(1) Input to Hidden Layer

$$o_k = \sigma\left(\sum_{m=1}^{M} x_m w_{m,k} + \beta_k\right)$$

(6) Hidden 1 to Input

$$\mu_k = \sum_{j=1}^{H_1} \varphi_j v_{k,j} o_k (1 - o_k)$$
$$\Delta w_{m,k} = -\eta(t)\mu_k x_m$$
$$\Delta\beta_k = -\eta(t)\beta_k$$

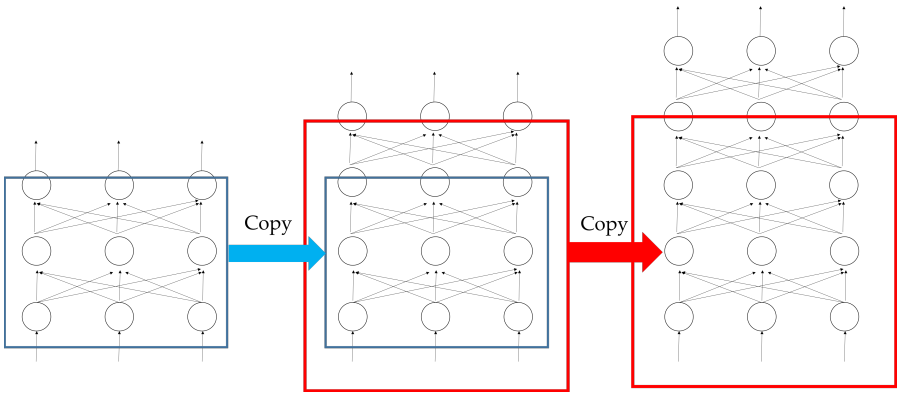
Gambar 11.10. Proses latihan DNN menggunakan *backpropagation*

Deep network terdiri dari banyak *layer* dan *synapse weight*, karenanya estimasi parameter susah dilakukan. Arti filosofisnya adalah susah/lama untuk menentukan relasi antara *input* dan *output*. Walaupun *deep learning* sepertinya kompleks, tetapi entah kenapa dapat bekerja dengan baik untuk

[10] Hanya istilah keren saja, tak ada arti spesial!

permasalahan praktis [9]. *Deep learning* dapat menemukan relasi "tersembunyi" antara *input* dan *output*, yang tidak dapat diselesaikan menggunakan *multilayer perceptron* (3 layers). Banyak orang percaya *deep neural network* lebih baik dibanding *neural network* yang lebar tapi sedikit *layer*, karena terjadi lebih banyak transformasi. Maksud lebih banyak transformasi adalah kemampuan untuk merubah *input* menjadi suatu representasi (tiap *hidden layer* dapat dianggap sebagai salah satu bentuk representasi *input*) dengan langkah *hierarchical*. Seperti contoh permasalahan XOR, permasalahan *non-linearly separable* pun dapat diselesaikan apabila kita dapat mentransformasi data (representasi data) ke dalam bentuk *linearly separable* pada ruang yang berbeda. Keuntungan utama *deep learning* adalah mampu merubah data dari *non-linearly separable* menjadi *linearly separable* melalui serangkaian transformasi (*hidden layers*). Selain itu, *deep learning* juga mampu mencari *decision boundary* yang berbentuk non-linear, serta mengsimulasikan interaksi non-linear antar fitur.

Karena memiliki banyak parameter, proses latihan ANN pada umumnya lambat. Ada beberapa strategi untuk mempercepat pembelajaran menggunakan deep learning, misalnya: regularisasi, ***successive learning***, dan penggunaan ***autoencoder*** [9]. Sebagai contoh, saya akan menceritakan *successive learning*. Arti *successive learning* adalah jaringan yang dibangun secara bertahap. Misal kita latih ANN dengan 3 *layers*, kemudian kita lanjutkan 3 *layers* tersebut menjadi 4 *layers*, lalu kita latih lagi menjadi 5 *layers*, dst. Hal ini sesuai dengan [55], yaitu mulai dari hal kecil. Ilustrasinya dapat dilihat pada Gambar 11.11. Menggunakan *deep learning* harus hati-hati karena pembelajaran cenderung *divergen* (artinya, *minimum square error* belum tentu semakin rendah seiring berjalannya waktu – *swing* relatif sering).



Gambar 11.11. Contoh *successive learning*

11.9 Tips

Pada contoh yang diberikan, *error* atau *loss* dihitung per tiap data point. Artinya begitu ada melewatkan suatu *input*, parameter langsung dioptimisasi sesuai dengan *loss*. Pada umumnya, hal ini tidak baik untuk dilakukan karena ANN menjadi tidak stabil. Metode yang lebih baik digunakan adalah teknik *minibatches*. Yaitu mengoptimisasi parameter untuk beberapa buah *inputs*. Jadi, update parameter dilakukan per *batch*. Data mana saja yang dimasukkan ke suatu *batch* dalam dipilih secara acak. Seperti yang mungkin kamu sadari secara intuitif, urutan data yang disajikan saat *training* mempengaruhi kinerja ANN. Pengacakan ini menjadi penting agar ANN mampu mengeneralisasi dengan baik. Kita dapat mengatur laju pembelajaran dengan menggunakan *learning rate*. Selain menggunakan *learning rate*, kita juga dapat menggunakan *momentum* (subbab 5.5).

Pada library/API *deep learning*, *learning rate* pada umumnya berubah-ubah sesuai dengan waktu. Selain itu, tidak ada nilai khusus (*rule-of-thumb*) untuk *learning rate* terbaik. Pada umumnya, kita inisiasi *learning rate* dengan nilai $\{0.001, 0.01, 0.1, 1\}$ [1]. Biasanya, kita menginisiasi proses latihan dengan nilai *learning rate* cukup besar, kemudian mengecil seiring berjalannya waktu[11]. Kemudian, kita mencari konfigurasi parameter terbaik dengan metode ***grid-search***[12], yaitu dengan mencoba-coba parameter secara *exhaustive* (*brute-force*) kemudian memilih parameter yang memberikan kinerja terbaik.

ANN sensitif terhadap inisialisasi parameter, dengan demikian banyak metode inisialisasi parameter misalkan, nilai *synapse weights* diambil dari distribusi binomial (silahkan eksplorasi lebih lanjut). Dengan hal ini, kinerja ANN dengan arsitektur yang sama dapat berbeda ketika dilatih ulang dari awal. Untuk menghindari *bias* inisialisasi parameter, biasanya ANN dilatih beberapa kali (umumnya 5, 10, atau 15 kali). Kinerja ANN yang dilaporkan adalah nilai kinerja rata-rata dan varians (*variance*). Kamu mungkin sudah menyadari bahwa melatih ANN harus telaten, terutama dibanding model linear. Untuk model linear, ia akan memberikan konfigurasi parameter yang sama untuk *training data* yang sama (kinerja pun sama). Tetapi, ANN dapat konfigurasi parameter yang berbeda untuk *training data* yang sama (kinerja pun berbeda). Pada model linear, kemungkinan besar variasi terjadi saat mengganti data. Pada ANN, variasi kinerja ada pada seluruh proses! Untuk membandingkan dua arsitektur ANN pada suatu dataset, kita dapat menggunakan *two sample t-test unequal variance* (arsitektur X lebih baik dari arsitektur Y secara signifikan dengan nilai $p < threshold$).

Apabila kamu pikir dengan seksama, ANN sebenarnya melakukan transformasi non-linear terhadap *input* hingga menjadi *output*. Parameter diper-

[11] Analogi: ngebut saat baru berangkat, kemudian memelan saat sudah dekat dengan tujuan agar tidak kelewat

[12] `https://en.wikipedia.org/wiki/Hyperparameter_optimization`

barahui agar transformasi non-linear *input* bisa menjadi semirip mungkin dengan *output* yang diharapkan. Dengan hal ini, istilah "ANN" memiliki asosiasi yang dekat dengan "transformasi non-linear". Kami ingin kamu mengingat, ANN (apapun variasi arsitekturnya) adalah **gabungan fungsi non-linear**, dengan demikian ia mampu mengaproksimasi fungsi non-linear (*decision boundary* dapat berupa fungsi non-linear).

Deep learning menjadi penting karena banyaknya transformasi (banyaknya *hidden layers*) lebih penting dibanding lebar jaringan. Seringkali (pada permasalahan praktis), kita membutuhkan banyak transformasi agar *input* bisa menjadi *output*. Setiap transformasi (*hidden layer*) merepresentasikan *input* menjadi suatu representasi. Dengan kata lain, *hidden layer* satu dan *hidden layer* lainnya mempelajari bentuk representasi atau karakteristik *input* yang berbeda.

Curriculum learning juga adalah tips yang layak disebutkan (*mention*) [56]. Penulis tidak mengerti detilnya, sehingga pembaca diharapkan membaca sendiri. Intinya adalah memutuskan apa yang harus ANN pelajari terlebih dahulu (mulai dari mempelajari hal mudah sebelum mempelajari hal yang susah).

11.10 Regularization and Dropout

Seperti yang sudah dijelaskan pada model linear. Kita ingin model mengeneralisasi dengan baik (kinerja baik pada *training data* dan *unseen examples*). Kita dapat menambahkan fungsi regularisasi untuk mengontrol kompleksitas ANN. Regularisasi pada ANN cukup *straightforward* seperti regularisasi pada model linear (subbab 5.8). Kami yakin pembaca bisa mengeksplorasi sendiri.

Selain itu, agar ANN tidak "bergantung" pada satu atau beberapa *synapse weights* saja, kita dapat menggunakan *dropout*. *Dropout* berarti me-*nol*-kan nilai *synapse weights* dengan nilai *rate* tertentu. Misalkan kita *nol*-kan nilai 30% *synapse weights* (*dropout rate*$= 0.3$) secara random. Hal ini dapat dicapai dengan teknik *masking*, yaitu mengalikan *synapse weights* dengan suatu *mask*.

Ingat kembali ANN secara umum, persamaan 11.12 dimana $\mathbf{W}$ adalah *synapse weights*, $\mathbf{x}$ adalah *input* (atau secara umum, dapat merepresentasikan *hidden state* pada suatu layer), b adalah *bias* dan f adalah fungsi aktivasi (non-linear). Kita buat suatu *mask* untuk *synapse weights* seperti pada persamaan 11.13, dimana $\mathbf{p}$ adalah vektor dan $p_i = [0, 1]$ merepresentasikan *synapse weight* diikutsertakan atau tidak. $r\%$ (*dropout rate*) elemen vektor $\mathbf{p}$ bernilai 0. Biasanya $\mathbf{p}$ diambil dari *bernoulli distribution* [1]. Kemudian, saat *feed forward*, kita ganti *synapse weights* menggunakan *mask* seperti pada persamaan 11.14. Saat menghitung *backpropagation*, turunan fungsi juga mengikutsertakan *mask* (gradient di-*mask*). Kami sarankan untuk membaca *paper* oleh Srivastava et al. [57] tentang *dropout* pada ANN. Contoh implementasi *dropout* dapat dilihat pada pranala berikut[13].

[13] `https://gist.github.com/yusugomori/cf7bce19b8e16d57488a`

$$o = f(\mathbf{x} \cdot \mathbf{W} + b) \tag{11.12}$$

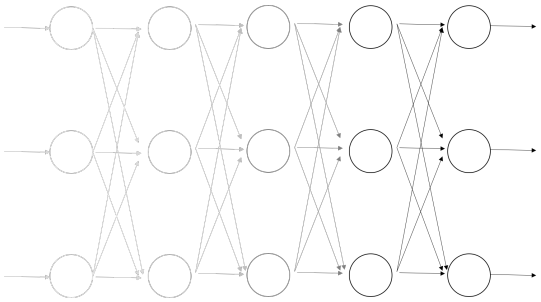
$$\mathbf{W}' = \mathbf{p} \cdot \mathbf{W} \tag{11.13}$$

$$o = f(\mathbf{x} \cdot \mathbf{W}' + b) \tag{11.14}$$

Baik *regularization* maupun *dropout* sudah menjadi metode yang cukup "standar" dan diaplikasikan pada berbagai macam arsitektur (tidak terbatas pada MLP saja).

11.11 Vanishing and Exploding Gradients

Pada beberapa kasus, nilai gradien ($\Delta\mathbf{W}$ - perubahan parameter) sangat kecil (mendekati nol - *vanishing*) atau sangat besar (*explode*). *Vanishing gradient problem* umum terjadi untuk ANN yang sangat dalam (*deep*), yaitu memiliki banyak layer. Hal ini juga terjadi pada arsitektur khusus, seperti *recurrent neural network* saat diberikan input yang panjang [58]. Turunan suatu fungsi bernilai lebih kecil dari fungsi tersebut. Artinya nilai gradient pada *input layer* bernilai lebih kecil dari *output layer*. Apabila kita memiliki banyak *layer*, nilai gradient saat *backpropagation* mendekati nol ketika diturunkan kembali dalam banyak proses. Ilustrasi *vanishing gradient* diberikan pada Gambar 11.12 (analogikan dengan *heat map*). Saat melakukan *backpropagation*, nilai gradien menjadi mendekati nol (warna semakin putih, delta nilai semakin menghilang). Penanganan permasalahan ini masih merupakan topik riset tersendiri. Untuk saat ini, biasanya digunakan fungsi aktivasi *long short term memory* (LSTM) atau *gated recurrent unit* (GRU) untuk menanganinya. Selain nilai gradien, nilai *synapse weights* juga bisa sangat kecil atau sangat besar. Hal ini juga tidak baik!



Gambar 11.12. Ilustrasi *vanishing gradient problem*

11.12 Rangkuman

Ada beberapa hal yang perlu kamu ingat, pertama-tama jaringan *neural network* terdiri atas:

1. *Input layer*
2. *Hidden layer(s)*
3. *Output layer*

Setiap *edge* yang menghubungkan suatu *node* dengan *node* lainnya disebut *synapse weight*. Pada saat melatih *neural network* kita mengestimasi nilai yang "bagus" untuk *synapse weights*.

Kedua, hal tersulit saat menggunakan *neural network* adalah menentukan topologi. Kamu bisa menggunakan berbagai macam variasi topologi *neural network* serta cara melatih untuk masing-masing topologi. Tetapi, suatu topologi tertentu lebih tepat untuk merepresentasikan permasalahan dibanding topologi lainnya. Menentukan tipe topologi yang tepat membutuhkan pengalaman.

Ketiga, proses *training* untuk *neural network* berlangsung lama. Secara umum, perubahan nilai *synapse weights* mengikuti tahapan (*stage*) berikut [9]:

1. *Earlier state*. Pada tahap ini, struktur global (kasar) diestimasi.
2. *Medium state*. Pada tahap ini, *learning* berubah dari tahapan global menjadi lokal (ingat *steepest gradient descent*).
3. *Last state*. Pada tahap ini, struktur detail sudah selesai diestimasi. Harapannya, model menjadi konvergen.

Neural network adalah salah satu *learning machine* yang dapat menemukan *hidden structure* atau pola data "implisit". Secara umum, *learning machine* tipe ini sering menjadi *overfitting/overtraining*, yaitu model memiliki kinerja sangat baik pada *training data*, tapi buruk pada *testing data*/*unseen example*. Oleh sebab itu, menggunakan *neural network* harus hati-hati.

Keempat, *neural network* dapat digunakan untuk *supervised*, *semi-supervised*, maupun *unsupervised learning*. Hal ini membuat *neural network* cukup populer belakangan ini karena fleksibilitas ini. Contoh penggunaan *neural network* untuk *unsupervised* learning akan dibahas pada bab 12. Semakin canggih komputer, maka semakin cepat melakukan perhitungan, dan semakin cepat melatih *neural network*. Hal ini adalah kemewahan yang tidak bisa dirasakan 20-30 tahun lalu.

Soal Latihan

11.1. Turunan

(a) Turunkanlah perubahan *noise*/*bias* untuk *training* pada MLP.
(b) Turunkanlah proses *training deep neural network* pada Gambar 11.10 termasuk perubahan *noise*/*bias*.

11.2. Neural Network Training

(a) Sebutkan dan jelaskan cara lain untuk melatih *artificial neural network* (selain *backpropagation*) (bila ada)!
(b) Apa kelebihan dan kekurangan *backpropagation*?
(c) Tuliskan persamaan MLP dengan menggunakan momentum! (kemudian berikan juga *backpropagation*-nya)

11.3. Neural Network - Unsupervised Learning

Bagaimana cara menggunakan *artificial neural network* untuk *unsupervised learning*?

11.4. Regularization Technique

(a) Sebutkan dan jelaskan teknik *regularization* untuk *neural network*! (dalam bentuk formula)
(b) Mengapa kita perlu menggunakan teknik tersebut?

11.5. Softmax Function

(a) Apa itu *softmax function*?
(b) Bagaimana cara menggunakan *softmax function* pada *neural network*?
(c) Pada saat kapan kita menggunakan fungsi tersebut?
(d) Apa kelebihan fungsi tersebut dibanding fungsi lainnya?

11.6. Transformasi atribut

Pada bab 4, diberikan contoh klasifikasi dengan data dengan atribut nominal. Akan tetapi, secara alamiah *neural network* membutuhkan data dengan atribut numerik untuk klasifikasi. Jelaskan konversi/strategi penanganan atribut nominal pada *neural network*!

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 [59]. 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 aljabar linear, yaitu dekomposisi matriks.

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 [60]. Untuk melakukan klasifikasi maupun *clustering*, kita membutuhkan fitur. Fitur tersebut haruslah dapat membedakan satu *instance* dan *instance* lainnya. Seringkali, untuk membedakan *instance* satu dan

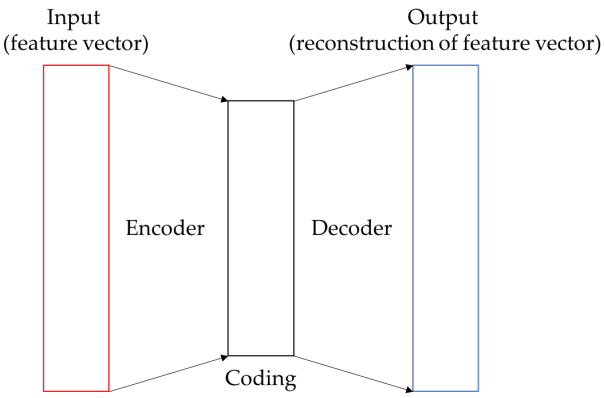
instance lainnya, kita membutuhkan *feature vector* yang berdimensi relatif "besar". Karena dimensi *feature vector* besar, kita butuh sumber daya komputasi yang besar juga (bab 9). Untuk itu, terdapat metode-metode ***feature selection***[1] untuk memilih fitur-fitur yang dianggap "representatif" dibanding 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 (atau secara lebih umum, proses menginterpretasikan *coding*) ***decoding***. *Neural network* yang mampu melakukan hal ini disebut ***encoder*** [61, 62, 63, 64, 65]. Contoh *representation learning* paling sederhana kemungkinan besar adalah ***autoencoder*** yaitu *neural network* yang dapat merepresentasikan data kemudian merekonstruksinya kembali. Ilustrasi *autoencoder* dapat dilihat pada 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. Ukuran *utility function* atau *performance measure* untuk *autoencoder* adalah mengukur *loss*. Idealnya, *output* harus sama dengan *input*, yaitu *autoencoder* dengan tingkat *loss* 0%.

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). 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 [66].

[1] `http://scikit-learn.org/stable/modules/feature_selection.html`

[2] Istilah *representation learning* pada umumnya mengacu dengan teknik menggunakan *neural network*.



Gambar 12.1. Contoh autoencoder sederhana

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 instans direpresentasikan oleh *feature vector* dapat disusun menjadi matriks $\mathbf{X}$ berukuran $N \times F$, dimana N adalah banyaknya instans[3] 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 diaglonal matriks yang berisi nilai eigenvalue.

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

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.

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

dimana $\mathbf{U}$ berukuran $N \times N$, $\mathbf{V}$ berukuran $N \times F$, dan $\mathbf{W}$ berukuran $F \times F$. 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 $\mathbf{X}^T\mathbf{X}$.

[3] Banyaknya training data.

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 $\hat{\mathbf{W}}$ $(K \times F)$. Apabila kita mengalikan semuanya, kita akan mendapat matriks $\hat{\mathbf{X}}$ yang disebut *low rank approximation* dari matriks asli $\mathbf{X}$, seperti diilustrasikan pada persamaan 12.3.

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

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 aproksimasi (walaupun berdimensi lebih rendah), kurang lebih melambangkan operasi pada matriks asli. Konsep ini menjadi fundamental *autoencoder* yang akan dibahas pada subbab berikutnya. 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 (ingat kembali materi PCA).

12.3 Ide Dasar Autoencoder

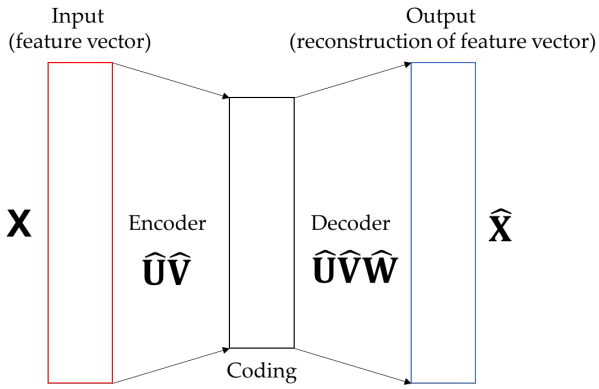
Seperti yang sudah dijelaskan *autoencoder* adalah *neural network* yang mampu merekonstruksi *input*. Ide dasar *autoencoder* tidak jauh dari konsep dekomposisi/*dimentionality reduction* menggunakan *singular value decomposition*. Diberikan dataset $\mathbf{X}$, kita ingin mensimulasikan pencarian matriks $\hat{\mathbf{X}}$ yang merupakan sebuah *low rank approximation* 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 instans 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*)[4].

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

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. Gambar 12.2 mengilustrasikan hubungan antara *autoencoder* dan *singular value decomposition*[5]. Perhatikan, *hidden layer*/*coding* adalah $\mathbf{E} = \hat{\mathbf{U}}\ \hat{\mathbf{V}}$. Dengan kata lain, kita



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

dapat melakukan operasi *dot-product* pada *coding* untuk merepresentasikan *dot-product* pada data asli $\mathbf{X}$. Ini adalah ide utama *autoencoder*, yaitu mengaproksimasi/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

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

[5] Hanya sebuah analogi.

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*.

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

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

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.

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

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 non-linear[6]. 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* [67, 68] perihal penjelasan lebih lengkap tentang perbedaan dan persamaan SVD dan *autoencoder* secara lebih matematis.

Apabila kamu hanya ingin mengerti konsep dasar *representation learning*, kamu dapat berhenti membaca sampai subbab ini. Secara sederhana *representation learning* adalah teknik untuk mengkompresi *input* ke dalam dimensi lebih rendah tanpa (diharapkan) ada kehilangan informasi. Operasi vektor (dan lainnya) pada level *coding* merepresentasikan operasi pada bentuk aslinya. Untuk pembahasan *autoencoder* secara lebih matematis, kamu dapat membaca pranala ini[7]. Apabila kamu ingin mengetahui lebih jauh contoh penggunaan *representation learning* secara lebih praktis, silahkan lanjutkan membaca materi subbab berikutnya. Buku ini akan memberikan contoh

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

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

penggunaan *representation learning* pada bidang *natural language processing* (NLP).

12.4 Representing Context: Word Embedding

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[8]. 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 [69]. 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*)[9]; 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 [59]. Arti kata ***embedding*** adalah transformasi kata (beserta konteksnya) menjadi bentuk matematis (vektor). "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.

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*.

[8] `https://wordnet.princeton.edu/`

[9] 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

12.4.1 Vector Space Model

Vector space model (VSM)[10] 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 12.9. 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.

$$IDF(t, D) = \log\left(\frac{N}{|d \epsilon D; t \epsilon d|}\right) \tag{12.9}$$

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

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

akan menghasilkan *sparse matrix*.

Statistical semantics hypothesis diturunkan lagi menjadi empat macam hipotesis [69]:

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 [69].

12.4.2 Sequential, Time Series dan Compositionality

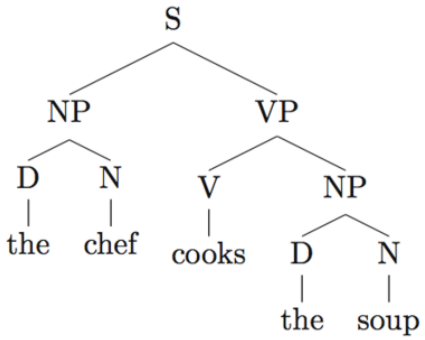
Bahasa manusia memiliki dua macam karakteristik yaitu adalah data berbentuk ***sequential data*** dan memenuhi sifat ***compositionality***. *Sequential data* adalah sifat data dimana suatu kemunculan $data_i$ dipengaruhi oleh data sebelumnya ($data_{i-1}, data_{i-2}, ...$). Perhatikan kedua kalimat 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 12.3. *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.



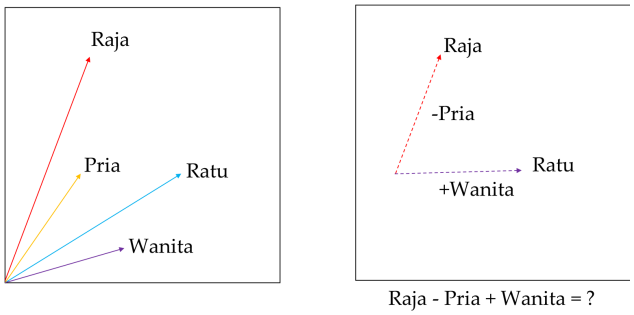
Gambar 12.3. Contoh constituent tree[11]

12.4.3 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 12.4. 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) dengan prinsip *singular value decomposition* dan *autoencoder* yang telah dijelaskan sebelumnya.

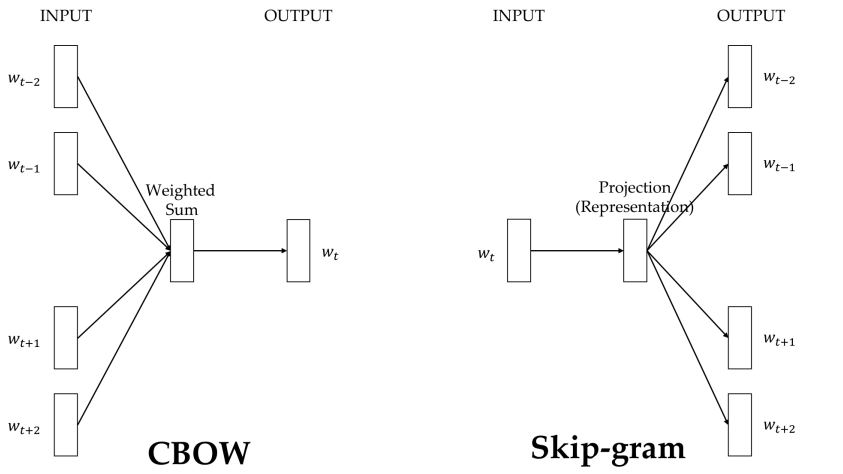
Selain *vector space model*, apakah ada cara lain yang mampu merepresentasikan kata dengan lebih baik? Salah satu kekurangan VSM adalah tidak memadukan sifat sekuensial pada konstruksi vektornya. Cara lebih baik ditemukan oleh [43, 44] dengan ekstensi pada [65]. Idenya adalah menggunakan teknik *representation learning* dan prinsip *statistical semantics hypothesis*. Metode ini lebih dikenal dengan sebutan *word2vec*. Tujuan *word2vec*

[11] source: Pinterest



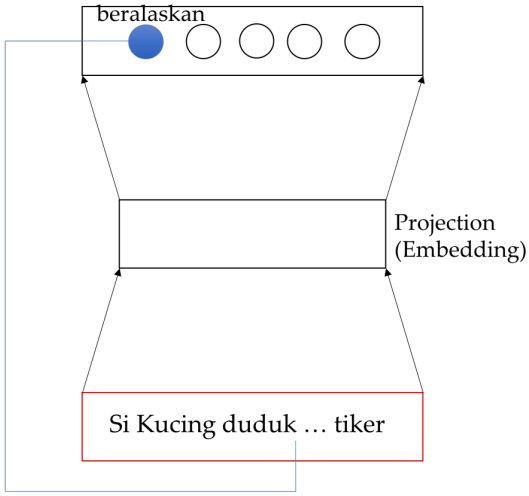
Gambar 12.4. Contoh operasi vektor kata

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 12.5. Bagian *projection layer* pada Gambar 12.5 adalah *coding* layer. Kami akan memberikan contoh CBOW secara lebih detil.



Gambar 12.5. CBOW vs Skip Gram [44]

Perhatikan Gambar 12.6. 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 proba-



Gambar 12.6. CBOW

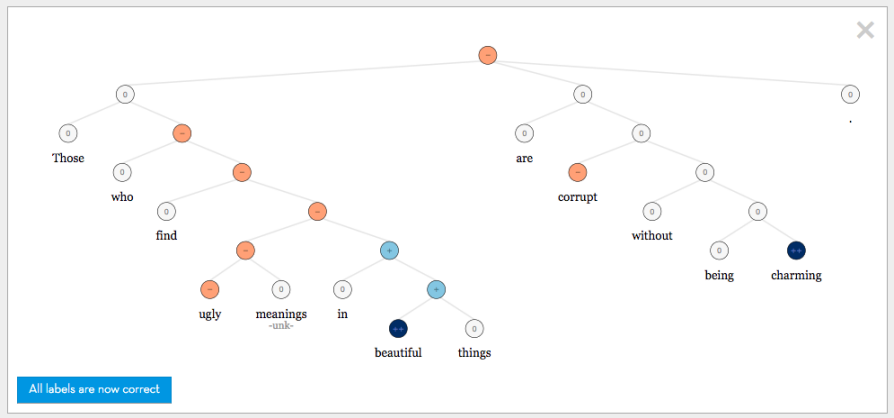
bilitas terbesar, misalkan pada kasus ini adalah "beralaskan". Dengan arsitektur ini, prinsip sekuensial atau *time series* dan *statistical semantics hypothesis* terpenuhi (*to a certain extent*). Teknik ini adalah salah satu contoh penggunaan *neural network* untuk *unsupervised learning*. Kita tidak 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[12]. Buku ini telah menjelaskan ide konseptual *word embedding* pada level abstrak. Apabila kamu tertarik untuk memahami detilnya secara matematis, kamu dapat membaca berbagai penelitian terkait[13]. Silahkan baca *paper* oleh Mikolov [43, 44] untuk detil implementasi *word embedding*.

12.4.4 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 [70]. 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*.

[12] `https://github.com/wiragotama/GloVe_Playground`
[13] Beberapa orang berpendapat bahwa *evil is in the detail*.

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[14]. 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 12.7).

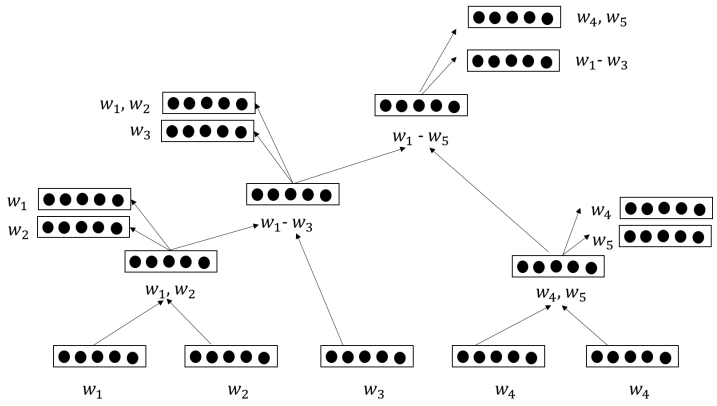


Gambar 12.7. Contoh analisis sentimen (Stanford)[15]

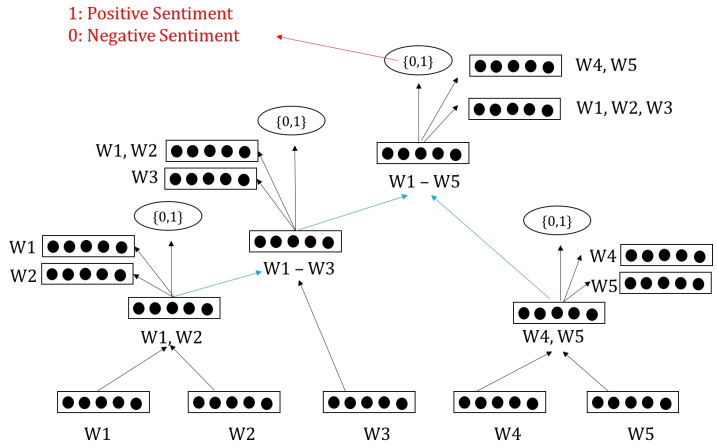
Cara lainnya adalah meng-*encode* kalimat sebagai *vektor* menggunakan *recursive autoencoder*. *Recursive* berarti suatu bagian adalah komposisi dari bagian lainnya. Penggunaan *recursive autoencoder* sangat rasional berhubung data memenuhi sifat *compositionality* yang direpresentasikan dengan baik 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 12.8. 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] Karena ini *recurrent neural network* bagus untuk *language modelling*.
[15] `http://nlp.stanford.edu:8080/sentiment/rntnDemo.html`



Gambar 12.8. Contoh recursive autoencoder



Gambar 12.9. Contoh recursive autoencoder dengan sentiment[62]

Gambar 12.9. 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 [64] yang lebih dikenal dengan nama ***paragraph vector*** atau ***doc2vec***.

12.5 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 (*visual processing*). 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. LSI dan LDA

(a) Jelaskanlah Latent Semantic Indexing (LSI) dan Latent Dirichlet Allocation (LDA)!
(b) Apa persamaan dan perbedaan antara LSI, LDA, dan *autoencoder*?

12.2. Variational Autoencoder

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

13

Arsitektur Neural Network

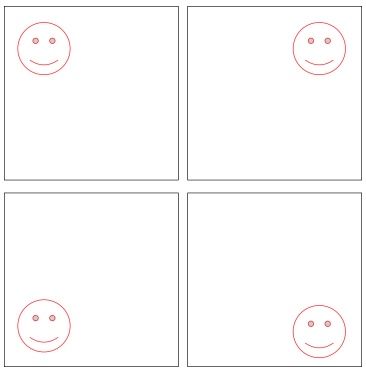
> As students cross the threshold from outside to insider, they also cross the threshold from superficial learning motivated by grades to deep learning motivated by engagement with questions. Their transformation entails an awakening–even, perhaps, a falling in love.
>
> John C. Bean

Seperti yang sudah dijelaskan pada bab 12, data memiliki karakteristik (dari segi *behaviour*) misal *sequential data*, *compositional data*, dsb. Terdapat arsitektur khusus *artificial neural network* (ANN) untuk menyelesaikan persoalan pada tipe data tertentu. Pada bab ini, kami akan memberikan beberapa contoh variasi arsitektur ANN yang cocok untuk tipe data tertentu. Penulis akan berusaha menjelaskan semaksimal mungkin ide-ide penting pada masing-masing arsitektur. Tujuan bab ini adalah memberikan pengetahuan konseptual (intuisi). Pembaca harus mengeksplorasi tutorial pemrograman untuk mampu mengimplementasikan arsitektur-arsitektur ini.

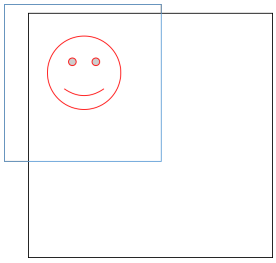
13.1 Convolutional Neural Network

Subbab ini akan memaparkan **ide utama** dari *convolutional neural network* (CNN) berdasarkan *paper* asli dari LeCun dan Bengio [71] (sekarang (2018) sudah ada banyak variasi). CNN memiliki banyak istilah dari bidang pemrosesan gambar (karena dicetuskan dari bidang tersebut), tetapi demi mempermudah pemahaman intuisi CNN, diktat ini akan menggunakan istilah yang lebih umum juga.

Sekarang, mari kita memasuki cerita CNN dari segi pemrosesan gambar. Objek bisa saja dterlatak pada berbagai macam posisi seperti diilustrasikan oleh Gambar. 13.1. Selain tantangan variasi posisi objek, masih ada juga tantangan lain seperti rotasi objek dan perbedaan ukuran objek (*scaling*). Kita ingin mengenali (memproses) objek pada gambar pada berbagai macam posisi yang mungkin (***translation invariance***). Salah satu cara yang mungkin adalah dengan membuat suatu mesin pembelajaran (ANN) untuk regional tertentu seperti pada Gambar. 13.2 (warna biru) kemudian meng-*copy* mesin pembelajaran untuk mampu mengenali objek pada regional-regional lainnya. Akan tetapi, kemungkinan besar ANN *copy* memiliki konfigurasi parameter yang sama dengan ANN awal. Hal tersebut disebabkan objek memiliki informasi prediktif (*predictive information – feature vector*) yang sama yang berguna untuk menganalisisnya. Dengan kata lain, objek yang sama (*smiley*) memiliki bentuk yang sama. ANN (MLP) bisa juga mempelajari prinsip *translation invariance*, tetapi memerlukan jauh lebih banyak parameter dibanding CNN (subbab berikutnya secara lebih matematis) yang memang dibuat dengan prinsip *translation invariance* (*built-in*).



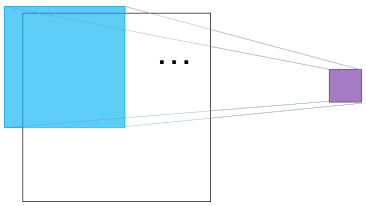
Gambar 13.1. Motivasi *convolutional neural network*



Gambar 13.2. Motivasi *convolutional neural network*, solusi regional

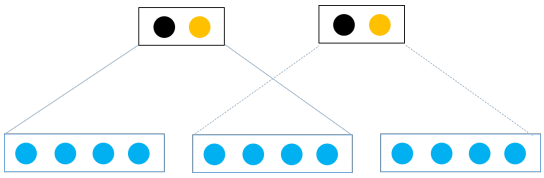
13.1.1 Convolution

Seperti yang sudah dijelaskan, motivasi CNN adalah untuk mampu mengenali aspek yang informatif pada regional tertentu (lokal). Dibanding meng-*copy* mesin pembelajaran beberapa kali untuk mengenali objek pada banyak regional, ide lebih baik adalah untuk menggunakan ***sliding window***. Setiap operasi pada *window*[1] bertujuan untuk mencari aspek lokal yang paling informatif. Ilustrasi diberikan oleh Gambar. 13.3. Warna biru merepresentasikan satu *window*, kemudian kotak ungu merepresentasikan aspek lokal paling informatif (disebut ***filter***) yang dikenali oleh *window*. Dengan kata lain, kita mentransformasi suatu *window* menjadi suatu nilai numerik (*filter*). Kita juga dapat mentransformasi suatu *window* (regional) menjadi d nilai numerik (d-*channels*, setiap elemen berkorespondensi pada suatu *filter*). *Window* ini kemudian digeser-geser sebanyak T kali, sehingga akhirnya kita mendapatkan vektor dengan panjang $d \times T$. Keseluruhan operasi ini disebut sebagai ***convolution***[2].



Gambar 13.3. *Sliding window*

Agar kamu lebih mudah memahami prinsip ini, kami berikan contoh dalam bentuk 1-D pada Gambar. 13.4. Warna biru merepresentasikan *feature vector* (regional) untuk suatu *input* (e.g., regional pada suatu gambar, kata pada kalimat, dsb). Pada contoh ini, setiap 2 *input* ditransformasi menjadi vektor berdimensi 2 (2-*channels*); menghasilkan vektor berdimensi 4 (2 *window* $\times$ 2).



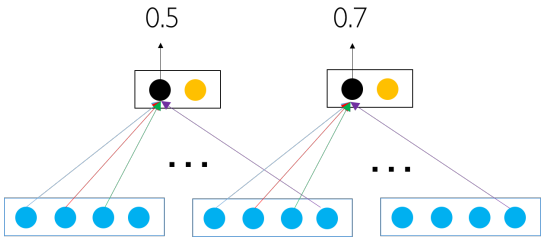
Gambar 13.4. *1D Convolution*

[1] Dikenal juga sebagai *receptive field*.

[2] Istilah *convolution* yang diterangkan pada konteks *machine learning* memiliki arti yang berbeda pada bidang *signal processing*.

Pada contoh sebelumnya, kita menggunakan *window* selebar 2, satu *window* mencakup 2 data; i.e., $window_1 = (x_1, x_2)$, $window_2 = (x_2, x_3)$, $\cdots$. Untuk suatu *input* $\mathbf{x}$. Kita juga dapat mempergunakan ***stride*** sebesar s, yaitu seberapa banyak data yang digeser untuk *window* baru. Contoh yang diberikan memiliki *stride* sebesar satu. Apabila kita memiliki *stride*$= 2$, maka kita menggeser sebanyak 2 data setiap langkah; i.e., $window_1 = (x_1, x_2)$, $window_2 = (x_3, x_4)$, $\cdots$.

Selain *sliding window* dan *filter*, *convolutional layer* juga mengadopsi prinsip *weight sharing*. Artinya, *synapse weights* untuk suatu filter adalah sama walau *filter* tersebut dipergunakan untuk berbagai *window*. Sebagai ilustrasi, perhatikan Gambar. 13.5, warna yang sama pada *synapse weights* menunjukkan *synapse weights* bersangkutan memiliki nilai (*weight*) yang sama. Tidak hanya pada *filter* hitam, hal serupa juga terjadi pada *filter* berwarna oranye (i.e., *filter* berwarnya oranye juga memenuhi prinsip *weight sharing*). Walaupun memiliki konfigurasi bobot *synapse weights* yang sama, unit dapat menghasilkan *output* yang berbeda untuk *input* yang berbeda. Konsep *weight sharing* ini sesuai dengan cerita sebelumnya bahwa konfigurasi parameter untuk mengenali karakteristik informatif untuk satu objek bernilai sama walau pada lokasi yang berbeda. Dengan *weight sharing*, parameter *neural network* juga menjadi lebih sedikit dibanding menggunakan *multilayer perceptron* (*feed-forward neural network*).



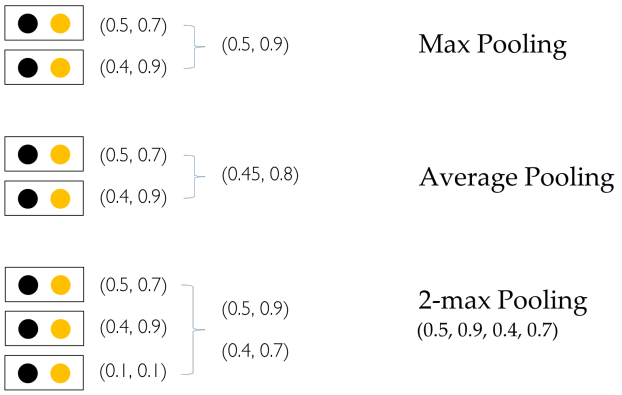
Gambar 13.5. Konsep *weight sharing*

13.1.2 Pooling

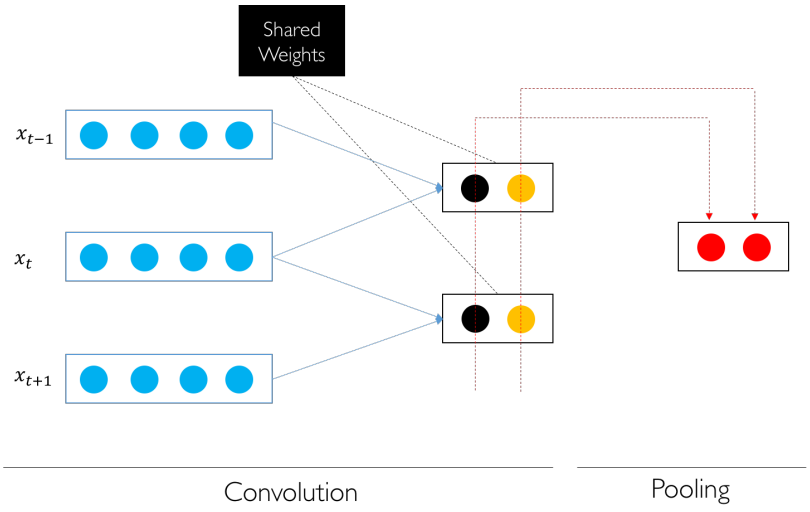
Pada tahap *convolution*, kita merubah setiap *k-sized window* menjadi satu vektor berdimensi d (yang dapat disusun menjadi matriks $\mathbf{D}$). Semua vektor yang dihasilkan pada tahap sebelumnya dikombinasikan (*pooled*) menjadi satu vektor $\mathbf{c}$. Ide utamanya adalah mengekstrak informasi paling informatif (semacam meringkas). Ada beberapa teknik *pooling*, diantaranya: *max pooling*, *average pooling*, dan *K-max pooling*[3]; diilustrasikan pada Gambar. 13.6. *Max pooling* mencari nilai maksimum untuk setiap dimensi vektor. *Average pooling*

[3] Kami ingin pembaca mengeksplorasi sendiri *dynamic pooling*.

mencari nilai rata-rata tiap dimensi. *K-max pooling* mencari K nilai terbesar untuk setiap dimensinya (kemudian hasilnya digabungkan). Gabungan operasi *convolution* dan *pooling* secara konseptual diilustrasikan pada Gambar. 13.7.

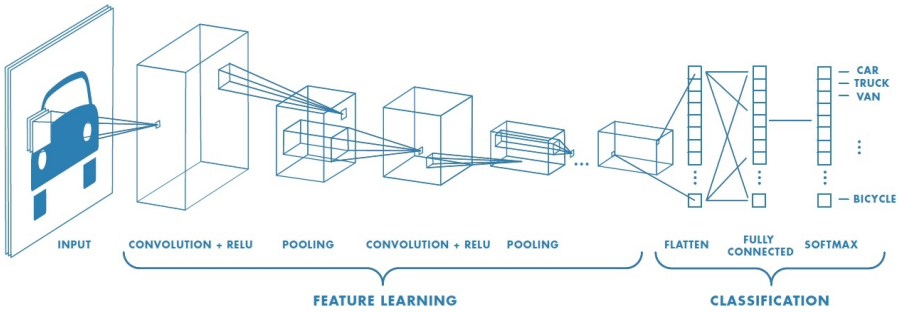


Gambar 13.6. Contoh *pooling*



Gambar 13.7. *Convolution* dan *pooling*

Setelah melewati berbagai operasi *convolution* dan *pooling*, kita akan memiliki satu vektor yang kemudian dilewatkan pada *multilayer perceptron* untuk melakukan sesuatu (tergantung permasalahan), misal klasifikasi gambar, klasifikasi sentimen, dsb (Ilustrasi pada Gambar. 13.8).



Gambar 13.8. *Convolutional Neural Network*[4]

13.1.3 Rangkuman

Kemampuan utama *convolutional neural network* (CNN) adalah arsitektur yang mampu mengenali informasi prediktif suatu objek (gambar, teks, potongan suara, dsb) walaupun objek tersebut dapat diposisikan dimana saja pada *input*. Kontribusi CNN adalah pada *convolution* dan *pooling* layer. *Convolution* bekerja dengan prinsip *sliding window* dan *weight sharing* (mengurangi kompleksitas perhitungan). *Pooling layer* berguna untuk merangkum informasi informatif yang dihasilkan oleh suatu *convolution* (mengurangi dimensi). Pada ujung akhir CNN, kita lewatkan satu vektor hasil beberapa operasi *convolution* dan *pooling* pada *multilayer perceptron* (*feed-forward neural network*), dikenal juga sebagai ***fully connected layer***, untuk melakukan suatu pekerjaan (e.g., klasifikasi). Perhatikan, pada umumnya CNN tidak berdiri sendiri, dalam artian CNN biasanya digunakan (dikombinasikan) untuk arsitektur yang lebih besar.

13.2 Recurrent Neural Network

Ide dasar *recurrent neural network* (RNN) adalah membuat topologi jaringan yang mampu merepresentasikan data *sequential* (sekuensial) atau *time series* [72], misalkan data ramalan cuaca. Cuaca hari ini bergantung kurang lebih pada cuaca hari sebelumnya. Sebagai contoh apabila hari sebelumnya mendung, ada kemungkinan hari ini hujan[5]. Walau ada yang menganggap sifat data sekuensial dan *time series* berbeda, RNN berfokus sifat data dimana instans waktu sebelumnya $(t-1)$ mempengaruhi instans pada waktu berikutnya (t). Intinya, mampu mengingat *history*.

Secara lebih umum, diberikan sebuah sekuens *input* $\mathbf{x} = (x_1, \cdots, x_T)$. Data x_t (i.e., vektor, gambar, teks, suara) dipengaruhi oleh data sebelum-sebelumnya (*history*), ditulis sebagai $P(x_t \mid \{x_1, \cdots, x_{t-1}\})$. Kami harap

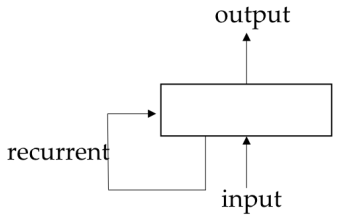
[4] `mathworks.com`

[5] Mohon bertanya pada ahli meteorologi untuk kebenaran contoh ini. Contoh ini semata-mata pengalaman pribadi penulis.

kamu ingat kembali materi *markov assumption* yang diberikan pada bab 8. Pada *markov assumption*, diasumsikan bahwa data x_t (data point) hanya dipengaruhi oleh **beberapa data sebelumnya saja** (analogi: *windowing*). Setidaknya, asumsi ini memiliki dua masalah:

1. Menentukan *window* terbaik. Bagaimana cara menentukan banyaknya data sebelumnya (secara optimal) yang mempengaruhi data sekarang.
2. Apabila kita menggunakan *markov assumption*, artinya kita mengganggap informasi yang dimuat oleh data lama dapat direpresentasikan oleh data lebih baru (x_t memuat informasi dari x_{t-J}; J adalah ukuran *window*). Penyederhanaan ini tidak jarang mengakibatkan informasi yang hilang.

RNN adalah salah satu bentuk arsitektur ANN untuk mengatasi masalah yang ada pada *markov assumption*. Ide utamanya adalah memorisasi[6], kita ingin mengingat **keseluruhan** sekuens (dibanding *markov assumption* yang mengingat sekuens secara terbatas), implikasinya adalah RNN mampu mengenali dependensi yang panjang (misal x_t ternyata dependen terhadap x_1). RNN paling sederhana diilustrasikan pada Gambar. 13.9. Ide utamanya adalah terdapat *pointer* ke dirinya sendiri.



Gambar 13.9. Bentuk konseptual paling sederhana recurrent NN

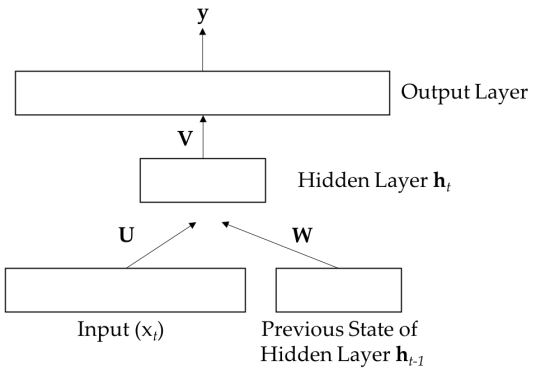
Ilustrasi Gambar. 13.9 mungkin sedikit susah dipahami karena berbentuk sangat konseptual. Bentuk lebih matematis diilustrasikan pada Gambar. 13.10 [72]. Perhitungan *hidden state* pada waktu ke-t bergantung pada *input* pada waktu ke-t (x_t) dan *hidden state* pada waktu sebelumnya (h_{t-1}).

Konsep ini sesuai dengan prinsip *recurrent* yaitu **mengingat** (memorisasi) kejadian sebelumnya. Kita dapat tulis kembali RNN sebagai persamaan 13.1.

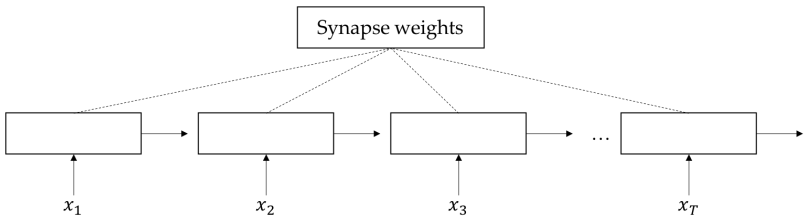
$$\mathbf{h}_t = f(x_t, \mathbf{h}_{t-1}, b) \tag{13.1}$$

dimana f adalah fungsi aktivasi (non-linear, dapat diturunkan). Demi menyederhanakan penjelasan, penulis tidak mengikutsertakan *bias* (b) pada fungsi-fungsi berikutnya. Kami berharap pembaca selalu mengingat bahwa *bias* adalah parameter yang diikutsertakan pada fungsi *artificial neural network*.

[6] Tidak merujuk hal yang sama dengan *dynamic programming*.



Gambar 13.10. Konsep Recurrent Neural Network



Gambar 13.11. Konsep *feed forward* pada RNN

Fungsi f dapat diganti dengan variasi *neural network*[7], misal menggunakan *long short-term memory network* (LSTM) [73]. Buku ini hanya akan menjelaskan konsep paling penting, silahkan eksplorasi sendiri variasi RNN.

Secara konseptual, persamaan 13.1 memiliki analogi dengan *full markov chain*. Artinya, *hidden state* pada saat ke-t bergantung pada semua *hidden state* dan *input* sebelumnya.

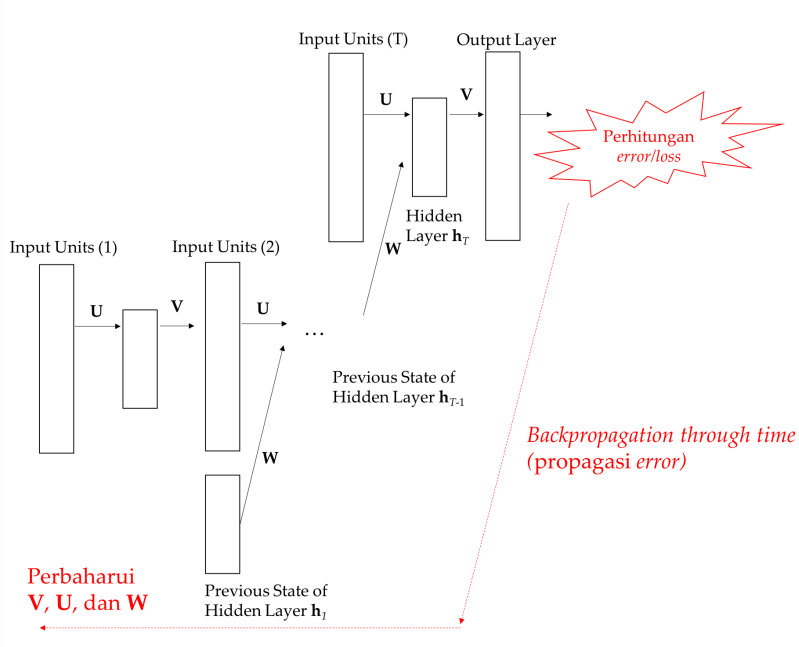
$$\begin{aligned}\mathbf{h}_t &= f(x_t, \mathbf{h}_{t-1}) \\ &= f(x_t, f(x_{t-1}, \mathbf{h}_{t-2})) \\ &= f(x_t, f(x_{t-1}, f(\{x_1, \cdots, x_{t-2}\}, \{\mathbf{h}_1, \cdots, \mathbf{h}_{t-3}\})))\end{aligned} \tag{13.2}$$

Training pada *recurrent neural network* dapat menggunakan metode *backpropagation*. Akan tetapi, metode tersebut kurang intuitif karena tidak mampu mengakomodasi *training* yang bersifat sekuensial *time series*. Untuk itu, terdapat metode lain bernama *backpropagation through time* [74].

Sebagai contoh kita diberikan sebuah sekuens $\mathbf{x}$ dengan panjang T sebagai input, dimana x_t melambangkan input ke-i (**data point** dapat berupa e.g., vektor, gambar, teks, atau apapun). Kita melakukan *feed forward* data

[7] `https://en.wikipedia.org/wiki/Recurrent_neural_network`

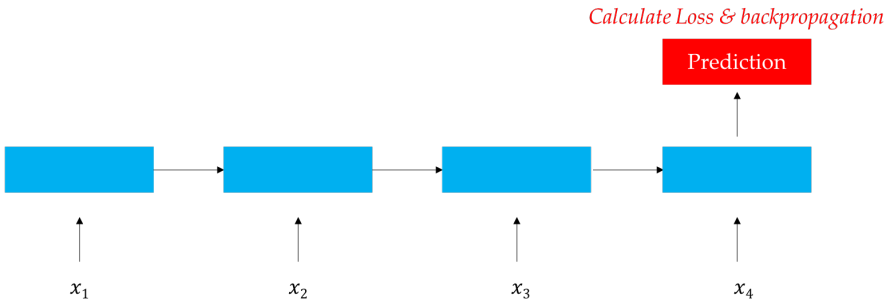
tersebut ke RNN, diilustrasikan pada Gambar. 13.11. Perlu diingat, RNN mengadopsi prinsip *parameter sharing* (serupa dengan *weight sharing* pada CNN) dimana neuron yang sama diulang-ulang saat process *feed forward*. Setelah selesai proses *feed forward*, kita memperbaharui parameter (*synapse weights*) berdasarkan propagasi *error* (*backpropagation*). Pada *backpropagation* biasa, kita perbaharui parameter sambil mempropagasi *error* dari *hidden state* ke *hidden state* sebelumnya. Teknik melatih RNN adalah ***backpropagation through time*** yang melakukan *unfolding* pada *neural network*. Kita mengupdate parameter, saat kita sudah mencapai *hidden state* paling awal. Hal ini diilustrasikan pada Gambar. 13.12[8]. Gambar. 13.12 dapat disederhanakan menjadi bentuk lebih abstrak (konseptual) pada Gambar. 13.13.



Gambar 13.12. Konsep *backpropagation through time* [42]

Kita mempropagasi *error* dengan adanya efek dari *next states of hidden layer*. *Synapse weights* diperbaharui secara *large update*. *Synapse weight* tidak diperbaharui per *layer*. Hal ini untuk merepresentasikan *neural network* yang mampu mengingat beberapa kejadian masa lampau dan keputusan saat ini dipengaruhi oleh keputusan pada masa lampau juga (ingatan). Untuk mengerti proses ini secara praktikal (dapat menuliskannya sebagai pro-

[8] Prinsip ini mirip dengan *weight sharing*.



Gambar 13.13. Konsep *backpropagation through time* [1]. Persegi berwarna merah umumnya melambangkan *multi-layer perceptron*

gram), penulis sarankan pembaca untuk melihat materi tentang ***computation graph***[9] dan disertasi PhD oleh Mikolov [42].

Walaupun secara konseptual RNN dapat mengingat seluruh kejadian sebelumnya, hal tersebut sulit untuk dilakukan secara praktikal untuk sekuens yang panjang. Hal ini lebih dikenal dengan *vanishing* atau *exploding gradient problem* [58, 75, 76]. Seperti yang sudah dijelaskan, ANN dan variasi arsitekturnya dilatih menggunakan teknik *stochastic gradient descent* (*gradient-based optimization*). Artinya, kita mengandalkan propagasi *error* berdasarkan turunan. Untuk sekuens *input* yang panjang, tidak jarang nilai *gradient* menjadi sangat kecil dekat dengan 0 (*vanishing*) atau sangat besar (*exploding*). Ketika pada satu *hidden state* tertentu, gradient pada saat itu mendekati 0, maka nilai yang sama akan dipropagasikan pada langkah berikutnya (menjadi lebih kecil lagi). Hal serupa terjadi untuk nilai gradient yang besar.

Berdasarkan pemaparan ini, RNN adalah teknik untuk merubah suatu sekuens *input*, dimana x_t merepresentasikan data ke-t (e.g., vektor, gambar, teks) menjadi sebuah *output* vektor $\mathbf{y}$. Vektor $\mathbf{y}$ dapat digunakan untuk permasalahan lebih lanjut (buku ini memberikan contoh *sequence to sequence* pada subbab 13.4). Bentuk konseptual ini dapat dituangkan pada persamaan 13.3. Biasanya, nilai $\mathbf{y}$ dilewatkan kembali ke sebuah *multi-layer perceptron* (MLP) dan fungsi softmax untuk melakukan klasifikasi akhir (*final output*) dalam bentuk probabilitas, seperti pada persamaan 13.4.

$$\mathbf{y} = \text{RNN}(x_1, \cdots, x_N) \tag{13.3}$$

$$\text{final output} = \text{softmax}(\text{MLP}(\mathbf{y})) \tag{13.4}$$

Perhatikan, arsitektur yang penulis deskripsikan pada subbab ini adalah arsitektur paling dasar. Untuk arsitektur *state-of-the-art*, kamu dapat membaca *paper* yang berkaitan.

[9] `https://www.coursera.org/learn/neural-networks-deep-learning/lecture/4WdOY/computation-graph`

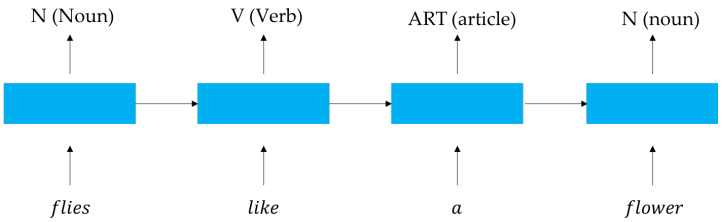
13.3 Part-of-speech Tagging Revisited

Pada bab sebelumnya, kamu telah mempelajari konsep dasar *recurrent neural network*. Selain digunakan untuk klasifikasi (i.e., *hidden state* terakhir digunakan sebagai *input* klasifikasi), RNN juga dapat digunakan untuk memprediksi sekuens seperti persoalan *part-of-speech tagging* (POS *tagging*) [77, 78, 79]. Kami harap kamu masih ingat materi bab 8 yang membahas apa itu persoalan POS *tagging*.

Diberikan sebuah sekuens kata $\mathbf{x} = \{x_1, \cdots, x_T\}$, kita ingin mencari sekuens *output* $\mathbf{y} = \{y_1, \cdots, y_T\}$ (*sequence prediction*); dimana y_i adalah kelas kata untuk x_i. Perhatikan, panjang *input* dan *output* adalah sama. Ingat kembali bahwa pada persoalan POS *tagging*, kita ingin memprediksi suatu kelas kata yang cocok y_i dari kumpulan kemungkinan kelas kata C ketika diberikan sebuah *history* seperti diilustrasikan oleh persamaan 13.5, dimana t_i melambangkan kandidat POS *tag* ke-i. Pada kasus ini, biasanya yang dicari tahu setiap langkah (*unfolding*) adalah probabilitas untuk memilih suatu kelas kata $t \in C$ sebagai kelas kata yang cocok untuk di-*assign* sebagai y_i.

Ilustrasi diberikan oleh Gambar. 13.14.

$$y_1, \cdots, y_T = \underset{t_1, \cdots, t_T; t_i \in C}{\arg\max}\, p(t_1, \cdots, t_T \mid x_1, \cdots, x_T) \tag{13.5}$$



Gambar 13.14. POS *tagging* menggunakan RNN

Apabila kita melihat secara sederhana (*markov assumption*), hal ini tidak lain dan tidak bukan adalah melakukan klasifikasi untuk setiap *instance* pada sekuens *input* (persamaan 13.6). Pada setiap *time step*, kita ingin menghasilkan *output* yang bersesuaian.

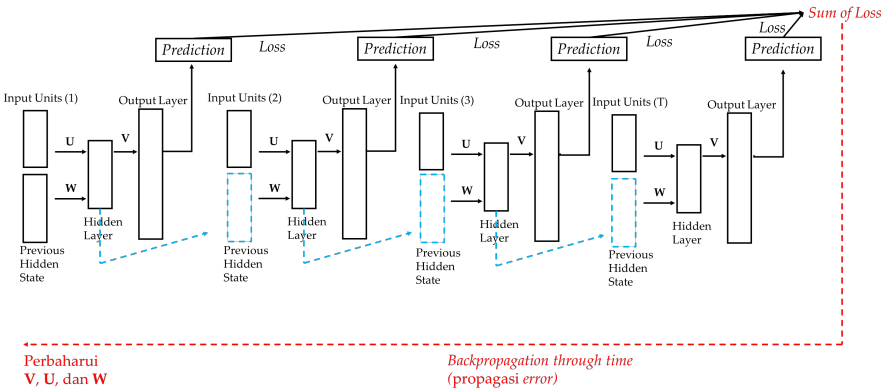
$$y_i = \underset{t_i \in C}{\arg\max}\, p(t_i | x_i) \tag{13.6}$$

Akan tetapi, seperti yang sudah dibahas sebelum sebelumnya, *markov assumption* memiliki kelemahan. Kelemahan utama adalah tidak menggunakan keseluruhan *history*. Persoalan ini cocok untuk diselesaikan oleh RNN karena kemampuannya untuk mengingat seluruh sekuens (berbeda dengan *hidden*

markov model (HMM) yang menggunakan *markov assumption*). Secara teoritis (dan juga praktis[10]), RNN lebih hebat dibanding HMM. Dengan ini, persoalan POS *tagging* (*full history*) diilustrasikan oleh persamaan 13.7.

$$y_i = \arg\max_{t_i \in C} p(t_i | x_1, \cdots, x_T) \tag{13.7}$$

Pada bab sebelumnya, kamu diberikan contoh persoalan RNN untuk satu *output*; i.e., diberikan sekuens *input*, *output*-nya hanyalah satu kelas yang mengkategorikan seluruh sekuens *input*. Untuk persoalan POS *tagging*, kita harus sedikit memodifikasi RNN untuk menghasilkan *output* bagi setiap elemen sekuens *input*. Hal ini dilakukan dengan cara melewatkan setiap *hidden layer* pada RNN pada suatu jaringan (anggap sebuah MLP + softmax). Kita lakukan prediksi kelas kata untuk setiap elemen sekuens *input*, kemudian menghitung *loss* untuk masing-masing elemen. Seluruh *loss* dijumlahkan untuk menghitung *backpropagation* pada RNN. Ilustrasi dapat dilihat pada Gambar. 13.15. Tidak hanya untuk persoalan POS *tagging*, arsitektur ini dapat juga digunakan pada persoalan *sequence prediction* lainnya seperti *named entity recognition*[11]. Gambar. 13.15 mungkin agak sulit untuk dilihat, kami beri bentuk lebih sederhananya (konseptual) pada Gambar. 13.16. Pada setiap langkah, kita menentukan POS *tag* yang sesuai dan menghitung *loss* yang kemudian digabungkan. *Backpropagation* dilakukan dengan mempertimbangkan keseluruhan (jumlah) *loss* masing-masing prediksi.

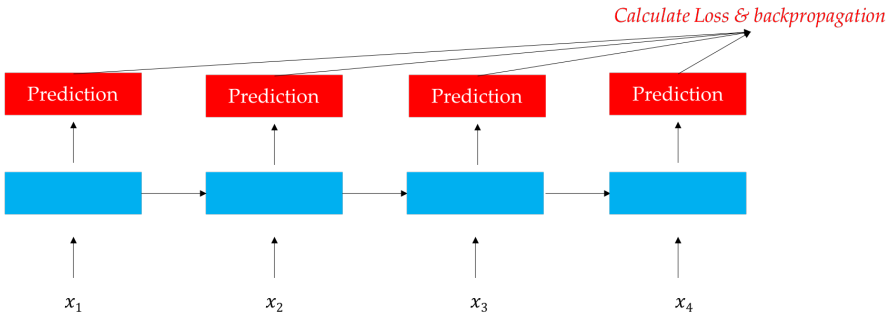


Gambar 13.15. *Sequence prediction* menggunakan RNN

Berdasarkan arsitektur yang sudah dijelaskan sebelumnya, prediksi POS *tag* ke-i bersifat independen dari POS *tag* lainnya. Padahal, POS *tag* lain-

[10] Sejauh yang penulis ketahui. Tetapi hal ini bergantung juga pada variasi arsitektur.

[11] `https://en.wikipedia.org/wiki/Named-entity_recognition`



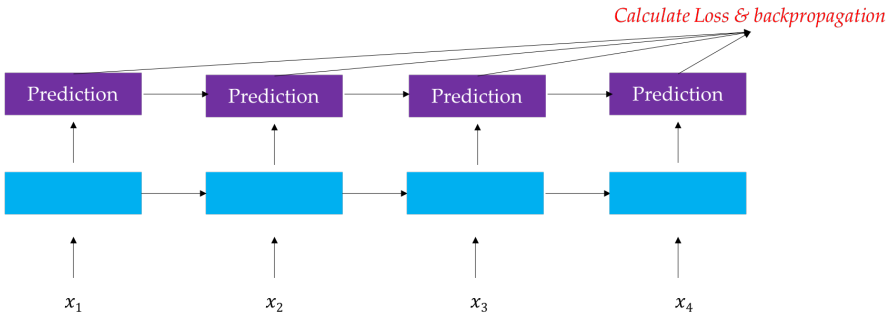
Gambar 13.16. *Sequence prediction* menggunakan RNN (disederhakan) [1]. Persegi berwarna merah umumnya melambangkan *multi-layer perceptron*

nya memiliki pengaruh saat memutuskan POS *tag* ke-i (ingat kembali materi bab 8); sebagai persamaan 13.8.

$$y_i = \arg\max_{t_i \in C} p(t_i \mid y_1, \cdots, y_{i-1}, x_1, \cdots, x_i) \tag{13.8}$$

Salah satu strategi untuk menangani hal tersebut adalah dengan melewatkan POS *tag* pada sebuah RNN juga, seperti para persamaan 13.9 [1] (ilustrasi pada Gambar. 13.17). Untuk mencari keseluruhan sekuens terbaik, kita dapat menggunakan teknik *beam search* (detil penggunaan dijelaskan pada subbab berikutnya). RNN^{x} pada persamaan 13.9 juga lebih intuitif apabila diganti menggunakan *bidirectional RNN* (dijelaskan pada subbab berikutnya).

$$\begin{aligned} &p(t_i \mid y_1, \cdots, y_{i-1}, x_1, \cdots, x_i) = \\ &\text{softmax}(\text{MLP}([\text{RNN}^{\text{x}}(x_1, \cdots, x_i); \text{RNN}^{\text{tag}}(t_1, \cdots, t_{i-1})])) \end{aligned} \tag{13.9}$$



Gambar 13.17. *Sequence prediction* menggunakan RNN (disederhakan). Persegi melambangkan RNN

13.4 Sequence to Sequence

Pertama-tama, kami ingin mendeskripsikan kerangka ***conditioned generation***. Pada kerangka ini, kita ingin memprediksi sebuah kelas y_i berdasarkan kelas yang sudah di-hasilkan sebelumnya (*history* yaitu $y_1, \cdots, y_{i-1}$) dan sebuah *conditioning context* $\mathbf{c}$ (berupa vektor).

Arsitektur yang dibahas pada subbab ini adalah variasi RNN untuk permasalahan *sequence generation*[12]. Diberikan sekuens *input* $\mathbf{x} = (x_1, \cdots, x_T)$. Kita ingin mencari sekuens *output* $\mathbf{y} = (y_1, \cdots, y_M)$. Pada subbab sebelumnya, x_i berkorespondensi langsung dengan y_i; i.e., y_i adalah kelas kata (kategori) untuk x_i. Tetapi, pada permasalahan saat ini, x_i tidak langsung berkorespondensi dengan y_i. Setiap y_i dikondisikan oleh **seluruh** sekuens *input* $\mathbf{x}$ (*conditioning context* dan *history* $\{y_1, \cdots, y_{i-1}\}$. Dengan itu, M (panjang sekuens *output*) tidak mesti sama dengan T (panjang sekuens *input*). Permasalahan ini masuk ke dalam kerangka *conditioned generation* dimana keseluruhan *input* $\mathbf{x}$ dapat direpresentasikan menjadi sebuah vektor $\mathbf{c}$ (*coding*). Vektor c ini menjadi variabel pengkondisi untuk menghasilkan *output* $\mathbf{y}$.

Pasangan *input-output* dapat melambangkan teks bahasa X-teks bahasa Y (translasi), teks-ringkasan, kalimat-*POS tags*, dsb. Artinya ada sebuah *input* dan kita ingin menghasilkan (*generate/produce*) sebuah *output* yang cocok untuk *input* tersebut. Hal ini dapat dicapai dengan momodelkan pasangan *input-output* $P(\mathbf{y} \mid \mathbf{x})$. Umumnya, kita mengasumsikan ada kumpulan parameter θ yang mengontrol *conditional probability*, sehingga kita transformasi *conditional probability* menjadi $P(\mathbf{y} \mid \mathbf{x}, \theta)$. *Conditional probability* $P(\mathbf{y} \mid \mathbf{x}, \theta)$ dapat difaktorkan sebagai persamaan 13.10. Kami harap kamu mampu membedakan persamaan 13.10 dan persamaan 13.5 (dan 13.8) dengan jeli. Sedikit perbedaan pada formula menyebabkan makna yang berbeda.

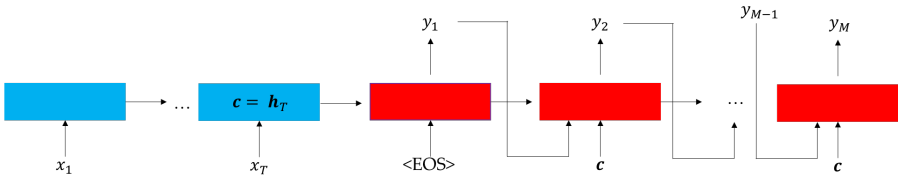
$$P(\mathbf{y} \mid \mathbf{x}, \theta) = \prod_{t=1}^{M} P(y_t \mid \{y_1, \ldots, y_{t-1}\}, \mathbf{x}, \theta), \tag{13.10}$$

Persamaan 13.10 dapat dimodelkan dengan ***encoder-decoder*** model yang terdiri dari dua buah RNN dimana satu RNN sebagai *encoder*, satu lagi sebagai *decoder*. Ilustrasi encoder-decoder dapat dilihat pada Gambar. 13.18. Gabungan RNN *encoder* dan RNN *decoder* ini disebut sebagai bentuk ***sequence to sequence***. Warna biru merepresentasikan *encoder* dan warna merah merepresentasikan *decoder*. "<EOS>" adalah suatu simbol spesial (*untuk praktikalitas*) yang menandakan bahwa sekuens *input* telah selesai dan saatnya berpindah ke *decoder*.

Sebuah *encoder* merepresentasikan sekuens *input* $\mathbf{x}$ menjadi satu vektor $\mathbf{c}$[13]. Kemudian, *decoder* men-*decode* representasi $\mathbf{c}$ untuk menghasilkan (*generate*) sebuah sekuens *output* $\mathbf{y}$. Perhatikan, arsitektur kali ini berbeda dengan arsitektur pada subbab 13.3. *Encoder-decoder* (*neural network*) bertin-

[12] Umumnya untuk bidang pemrosesan bahasa alami.

[13] Ingat kembali bab 12 untuk mengerti kenapa hal ini sangat diperlukan.



Gambar 13.18. Konsep *encoder-decoder* [76]

dak sebagai kumpulan parameter θ yang mengatur *conditional probability*. *Encoder-decoder* juga dilatih menggunakan prinsip *gradient-based optimization* untuk *tuning* parameter yang mengkondisikan *conditional probability* [76]. Dengan ini, persamaan 13.10 sudah didefinisikan sebagai *neural network* sebagai persamaan 13.11. "enc" dan "dec" adalah fungsi *encoder* dan *decoder*, yaitu sekumpulan transformasi non-linear.

$$y_t = \text{dec}(\{y_1, \cdots, y_{t-1}\}, \text{enc}(\mathbf{x}), \theta) \tag{13.11}$$

Begitu model dilatih, *encoder-decoder* akan mencari *output* $\hat{\mathbf{y}}$ terbaik untuk suatu input $\mathbf{x}$, dillustrasikan pada persamaan 13.12. Masing-masing komponen *encoder-decoder* dibahas pada subbab-subbab berikutnya. Untuk abstraksi yang baik, penulis akan menggunakan notasi aljabar linear. Kami harap pembaca sudah familiar dengan representasi *neural network* menggunakan notasi aljabar linear seperti yang dibahas pada bab 11.

$$\hat{\mathbf{y}} = \arg\max_{\mathbf{y}} p(\mathbf{y} \mid \mathbf{x}, \theta) \tag{13.12}$$

13.4.1 Encoder

Seperti yang sudah dijelaskan, *encoder* mengubah sekuens *input* $\mathbf{x}$ menjadi satu vektor $\mathbf{c}$. Suatu data point pada sekuens *input* x_t (e.g., kata, gambar, suara, dsb) umumnya direpresentasikan sebagai *feature vector* $\mathbf{e}_t$. Dengan demikian, *encoder* dapat direpresentasikan dengan persamaan 13.13

$$\begin{aligned} \mathbf{h}_t &= f(\mathbf{h}_{t-1}, \mathbf{e}_t) \\ &= f(\mathbf{h}_{t-1}\mathbf{U} + \mathbf{e}_t\mathbf{W}) \end{aligned} \tag{13.13}$$

dimana f adalah fungsi aktivasi non-linear; $\mathbf{U}$ dan $\mathbf{W}$ adalah matriks bobot (*weight matrices*–merepresentasikan *synapse weights*).

Representasi *input* $\mathbf{c}$ dihitung dengan persamaan 13.14, yaitu sebagai *weighted sum* dari *hidden states* [52], dimana q adalah fungsi aktivasi non-linear. Secara lebih sederhana, kita boleh langsung menggunakan $\mathbf{h}_T$ sebagai $\mathbf{c}$ [76].

$$\mathbf{c} = q(\{\mathbf{h}_1, \cdots, \mathbf{h}_T\}) \tag{13.14}$$

Walaupun disebut sebagai representasi keseluruhan sekuens *input*, informasi awal pada *input* yang panjang dapat hilang. Artinya $\mathbf{c}$ lebih banyak memuat informasi *input* ujung-ujung akhir. Salah satu strategi yang dapat digunakan adalah dengan membalik (*reversing*) sekuens *input*. Sebagai contoh, *input* $\mathbf{x} = (x_1, \cdots, x_T)$ dibalik menjadi $(x_T, \cdots, x_1)$ agar bagian awal $(\cdots, x_2, x_1)$ lebih dekat dengan *decoder* [76]. Informasi yang berada dekat dengan *decoder* cenderung lebih diingat. Kami ingin pembaca mengingat bahwa teknik ini pun tidaklah sempurna.

13.4.2 Decoder

Seperti yang sudah dijelaskan sebelumnya, *encoder* memproduksi sebuah vektor $\mathbf{c}$ yang merepresentasikan sekuens *input*. *Decoder* menggunakan representasi ini untuk memproduksi (*generate*) sebuah sekuens *output* $\mathbf{y} = (y_1, \cdots, y_M)$, disebut sebagai proses ***decoding***. Mirip dengan *encoder*, kita menggunakan RNN untuk menghasilkan *output* seperti diilustrasikan pada persamaan 13.15.

$$\begin{aligned} \mathbf{h}'_t &= f(\mathbf{h}'_{t-1}, \mathbf{e}'_{t-1}, \mathbf{c}) \\ &= f(\mathbf{h}'_{t-1}\mathbf{H} + \mathbf{e}'_{t-1}\mathbf{E} + \mathbf{c}\mathbf{C}) \end{aligned} \tag{13.15}$$

dimana f merepresentasikan fungsi aktivasi non-linear; $\mathbf{H}$, $\mathbf{E}$, dan $\mathbf{C}$ merepresentasikan *weight matrices*. *Hidden state* $\mathbf{h}'_t$ melambangkan distribusi probabilitas suatu objek (e.g., POS tag, kelas kata yang **berasal dari suatu himpunan**) untuk menjadi *output* y_t. Umumnya, y_t adalah dalam bentuk *feature-vector* $\mathbf{e}'_t$.

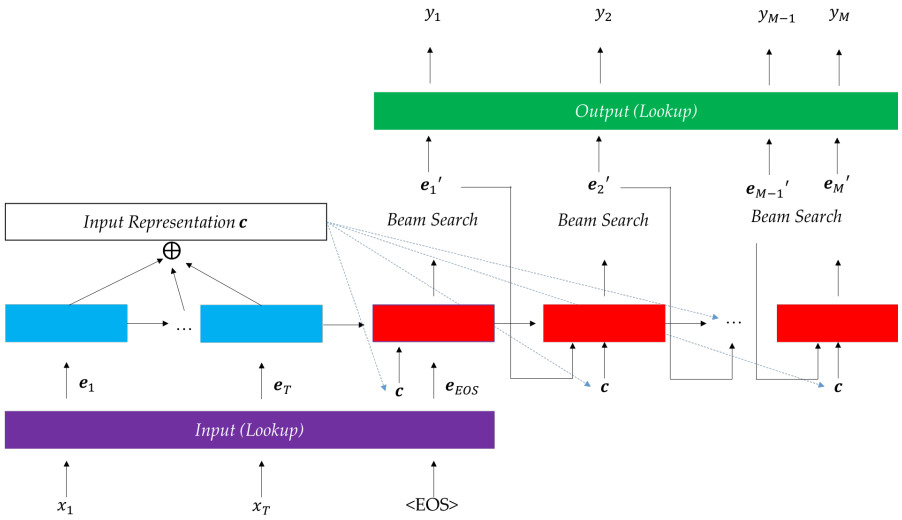
Dengan penjelasan ini, mungkin pembaca berpikir Gambar. 13.18 tidak lengkap. Kamu benar! Penulis sengaja memberikan gambar simplifikasi. Gambar lebih lengkap (dan lebih nyata) diilustrasikan pada Gambar. 13.19.

Kotak berwarna ungu dan hijau dapat disebut sebagai *lookup matrix* atau *lookup table*. Tugas mereka adalah mengubah *input* x_t menjadi bentuk *feature vector*-nya (e.g., *word embedding*) dan mengubah $\mathbf{e}'_t$ menjadi y_t. Komponen "*Beam Search*" dijelaskan pada subbab berikutnya.

13.4.3 Beam Search

Kita ingin mencari sekuens *output* yang memaksimalkan nilai probabilitas pada persamaan 13.12. Artinya, kita ingin mencari *output* terbaik. Pada suatu tahapan *decoding*, kita memiliki beberapa macam kandidat objek untuk dijadikan *output*. Kita ingin mencari sekuens objek sedemikian sehingga probabilitas akhir sekuens objek tersebut bernilai terbesar sebagai *output*. Hal ini dapat dilakukan dengan algoritma *Beam Search*[14].

[14] `https://en.wikipedia.org/wiki/Beam_search`



Gambar 13.19. Konsep *encoder-decoder* (*full*)

```
beamSearch(problemSet, ruleSet, memorySize)
        openMemory = new memory of size memorySize
        nodeList = problemSet.listOfNodes
        node = root or initial search node
        add node to OpenMemory;
        while(node is not a goal node)
                delete node from openMemory;
                expand node and obtain its children, evaluate those children;
                if a child node is pruned according to a rule in ruleSet, delete it;
                place remaining, non-pruned children into openMemory;
                if memory is full and has no room for new nodes, remove the worst
                        node, determined by ruleSet, in openMemory;
        node = the least costly node in openMemory;
```

Gambar 13.20. *Beam Search*[15]

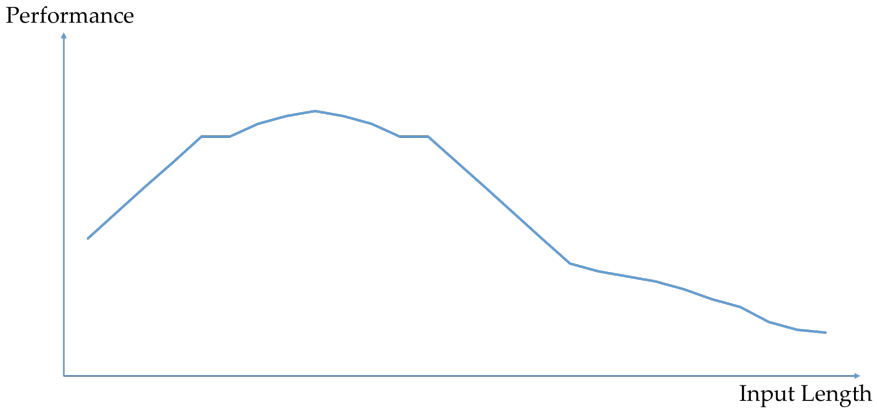
Secara sederhana, algoritma *Beam Search* mirip dengan algoritma Viterbi yang sudah dijelaskan pada bab 8, yaitu algoritma untuk mencari sekuens dengan probabilitas tertinggi. Perbedaannya terletak pada *heuristic*. Untuk menghemat memori komputer, algoritma *Beam Search* melakukan ekspansi terbatas. Artinya mencari hanya beberapa (B) kandidat objek sebagai sekuens berikutnya, dimana beberapa kandidat objek tersebut memiliki probabilitas $P(y_t \mid y_{t-1})$ terbesar. B disebut sebagai *beam-width*. Algoritma *Beam Search* bekerja dengan prinsip yang mirip dengan *best-first search* (*best-B search*) yang sudah kamu pelajari di kuliah algoritma atau pengenalan kecerdasan

[15] `https://en.wikibooks.org/wiki/Artificial_Intelligence/Search/Heuristic_search/Beam_search`

buatan[16]. Pseudo-code *Beam Search* diberikan pada Gambar. 13.20 (*direct quotation*).

13.4.4 Attention-based Mechanism

Seperti yang sudah dijelaskan sebelumnya, model *encoder-decoder* memiliki masalah saat diberikan sekuens yang panjang (*vanishing* atau *exploding gradient problem*). Kinerja model dibandingkan dengan panjang *input* kurang lebih dapat diilustrasikan pada Gambar. 13.21. Secara sederhana, kinerja model menurun seiring sekuens input bertambah panjang. Selain itu, representasi **c** yang dihasilkan *encoder* harus memuat informasi keseluruhan *input* walaupun sulit dilakukan. Ditambah lagi, *decoder* menggunakan representasinya **c** saja tanpa boleh melihat bagian-bagian khusus *input* saat *decoding*. Hal ini tidak sesuai dengan cara kerja manusia, misalnya pada kasus translasi bahasa. Ketika mentranslasi bahasa, manusia melihat bolak-balik bagian mana yang sudah ditranslasi dan bagian mana yang sekarang (difokuskan) untuk ditranslasi. Artinya, manusia berfokus pada suatu bagian *input* untuk menghasilkan suatu translasi.

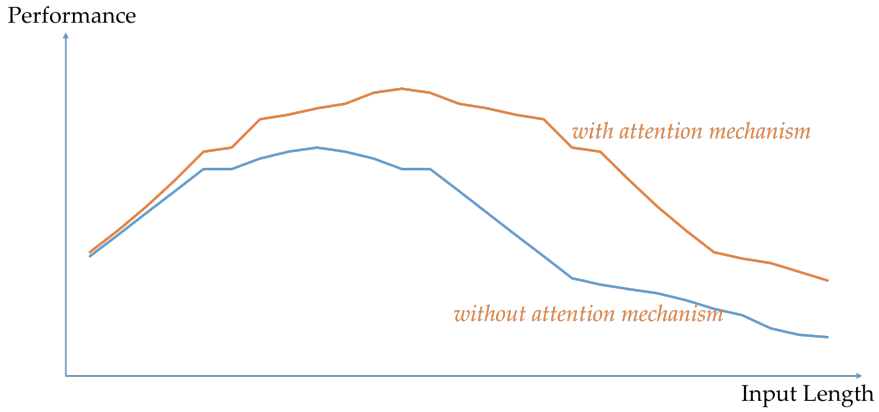


Gambar 13.21. Permasalahan *input* yang panjang

Sudah dijelaskan sebelumnya bahwa representasi sekuens *input* **c** adalah sebuah *weighted sum*. **c** yang sama digunakan sebagai *input* bagi *decoder* untuk menentukan semua *output*. Akan tetapi, untuk suatu tahapan *decoding* (untuk *hidden state* $\mathbf{h}'_t$ tertentu), kita mungkin ingin model lebih berfokus pada bagian *input* tertentu daripada *weighted sum* yang sifatnya generik. Ide ini adalah hal yang mendasari ***attention mechanism*** [52, 53]. Ide ini sangat

[16] `https://www.youtube.com/watch?v=j1H3jAAGlEA&t=2131s`

berguna pada banyak aplikasi pemrosesan bahasa alami. *Attention mechanism* dapat dikatakan sebagai suatu *soft alignment* antara *input* dan *output*. Mekanisme ini dapat membantu mengatasi permasalahan *input* yang panjang, seperti diilustrasikan pada Gambar. 13.22.



Gambar 13.22. Menggunakan vs. tidak menggunakan *attention*

Dengan menggunakan *attention mechanism*, kita dapat mentransformasi persamaan 13.15 pada *decoder* menjadi persamaan 13.16

$$\mathbf{h}'_t = f'(\mathbf{h}'_{t-1}, \mathbf{e}'_{t-1}, \mathbf{c}, \mathbf{k}_t) \tag{13.16}$$

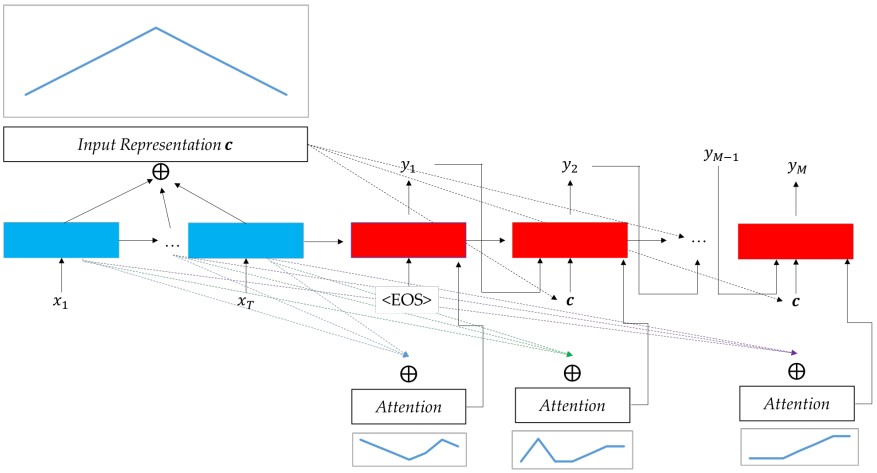
dimana $\mathbf{k}_t$ merepresentasikan seberapa (*how much*) *decoder* harus memfokuskan diri ke *hidden state* tertentu pada *encoder* untuk menghasilkan *output* saat ke-t. $\mathbf{k}_t$ dapat dihitung pada persamaan 13.17

$$\begin{aligned} \mathbf{k}_t &= \sum_{i=1}^{T} \alpha_{t,i} \mathbf{h}_i \\ \alpha_{t,i} &= \frac{\exp(\mathbf{h}_i \cdot \mathbf{h}'_{t-1})}{\sum_{z=1}^{T} \exp(\mathbf{h}_z \cdot \mathbf{h}'_{t-1})} \end{aligned} \tag{13.17}$$

dimana T merepresentasikan panjang *input*, $\mathbf{h}_i$ adalah *hidden state* pada *encoder* pada saat ke-i, $\mathbf{h}'_{t-1}$ adalah *hidden state* pada *decoder* saat ke $t-1$.

Sejatinya $\mathbf{k}_t$ adalah sebuah *weighted sum*. Berbeda dengan $\mathbf{c}$ yang bernilai sama untuk setiap tahapan *decoding*, *weight* atau bobot ($\alpha_{t,i}$) masing-masing *hidden state* pada *encoder* berbeda-beda untuk tahapan *decoding* yang berbeda. Perhatikan Gambar. 13.23 sebagai ilustrasi (lagi-lagi, bentuk *encoder-decoder* yang disederhanakan). Terdapat suatu bagian grafik yang

menunjukkan distribusi bobot pada bagian *input representation* dan *attention*. Distribusi bobot pada *weighted sum* **c** adalah pembobotan yang bersifat generik, yaitu berguna untuk keseluruhan (rata-rata) kasus. Masing-masing *attention* (semacam *layer* semu) memiliki distribusi bobot yang berbeda pada tiap tahapan *decoding*. Walaupun *attention mechanism* sekalipun tidak sempurna, ide ini adalah salah satu penemuan yang sangat penting.



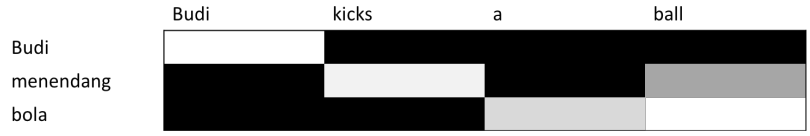
Gambar 13.23. *Encoder-decoder with attention*

Seperti yang dijelaskan pada bab 11 bahwa *neural network* susah untuk dimengerti. *Attention mechanism* adalah salah satu cara untuk mengerti *neural network*. Contoh yang mungkin lebih mudah dipahami diberikan pada Gambar. 13.24 yang merupakan contoh kasus mesin translasi [52]. *Attention mechanism* mampu mengetahui *soft alignment*, yaitu kata mana yang harus difokuskan saat melakukan translasi bahasa (bagian *input* mana berbobot lebih tinggi). Dengan kata lain, *attention mechanism* memberi interpretasi kata pada *output* berkorespondensi dengan kata pada *input* yang mana. Sebagai informasi, menemukan cara untuk memahami (interpretasi) ANN adalah salah satu tren riset masa kini [51].

13.4.5 Variasi Arsitektur Sequence to Sequence

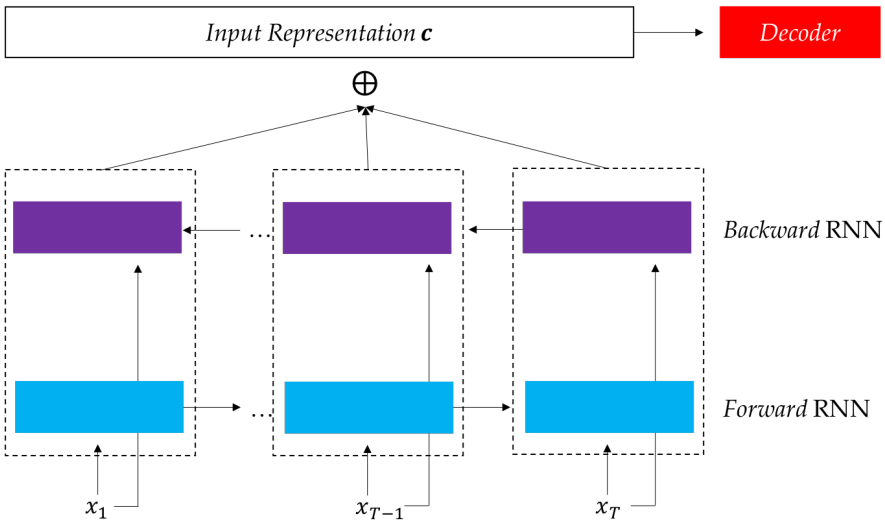
Selain RNN, kita juga dapat menggunakan *bidirectional* RNN (BiRNN) untuk mengikutsertakan pengaruh baik *hidden state* sebelum $(\mathbf{h}_1, \cdots, \mathbf{h}_{t-1})$ dan setelah $(\mathbf{h}_{t+1}, \cdots, \mathbf{h}_T)$ untuk menghitung *hidden state* sekarang $(\mathbf{h}_t)$ [80, 81, 82]. BiRNN menganggap $\mathbf{h}_t$ sebagai gabungan (*concatenation*) *forward hidden state* $\mathbf{h}_t^{\rightarrow}$ dan *backward hidden state* $\mathbf{h}_t^{\leftarrow}$, ditulis sebagai $\mathbf{h}_t = \mathbf{h}_t^{\rightarrow} + \mathbf{h}_t^{\leftarrow}$[17].

[17] Perhatikan! + disini dapat diartikan sebagai penjumlahan atau konkatenasi



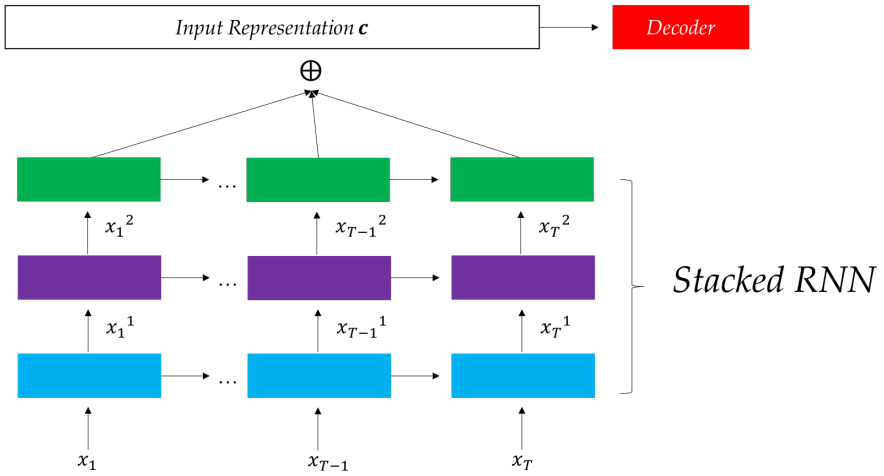
Gambar 13.24. *Attention mechanism* pada translasi bahasa [52]. Warna lebih terang merepresentasikan bobot (fokus/*attention*) lebih tinggi. Sebagai contoh, kata "menendang" berkorespondensi paling erat dengan kata "kicks"

Forward hidden state dihitung seperti RNN biasa yang sudah dijelaskan pada subbab *encoder*, yaitu $\mathbf{h}_t^{\rightarrow} = f(\mathbf{h}_{t-1}^{\rightarrow}, \mathbf{e}_t)$.. *Backward hidden state* dihitung dengan arah terbalik $\mathbf{h}_t^{\leftarrow} = f(\mathbf{h}_{t+1}^{\leftarrow}, \mathbf{e}_t)$. Ilustrasi *encoder-decoder* yang menggunakan BiRNN dapat dilihat pada Gambar. 13.25.



Gambar 13.25. *Encoder-decoder* dengan BiRNN

Selain variasi RNN menjadi BiRNN kita dapat menggunakan *stacked RNN* seperti pada Gambar. 13.26 dimana *output* pada RNN pertama bertindak sebagai *input* pada RNN kedua. *Hidden states* yang digunakan untuk menghasilkan representasi *encoding* adalah RNN pada tumpukan paling atas. Kita juga dapat menggunakan variasi *attention mechanism* seperti *neural checklist model* [83] atau *graph-based attention* [84]. Selain yang disebutkan, masih banyak variasi lain yang ada, silahkan eksplorasi lebih lanjut sendiri.



Gambar 13.26. *Encoder-decoder* dengan *stacked* RNN

13.4.6 Rangkuman

Sequence to sequence adalah salah satu bentuk *conditioned generation*. Artinya, menggunakan RNN untuk menghasilkan (*generate*) suatu sekuens *output* yang dikondisikan oleh variabel tertentu. Diktat ini memberikan contoh bagaimana menghasilkan suatu sekuens *output* berdasarkan sekuens *input* (*conditioned on a sequence of input*). Selain *input* berupa sekuens, konsep ini juga dapat diaplikasikan pada bentuk lainnya. Misalnya, menghasilkan *caption* saat input yang diberikan adalah sebuah gambar [85]. Kita ubah *encoder* menjadi sebuah CNN (ingat kembali subbab 13.1) dan *decoder* berupa RNN [85]. Gabungan CNN-RNN tersebut dilatih bersama menggunakan metode *backpropagation*.

Perhatikan, walaupun memiliki kemiripan dengan *hidden markov model*, *sequence to sequence* bukanlah *generative model*. Pada *generative model*, kita ingin memodelkan *joint probability* $p(x, y) = p(y \mid x)p(x)$ (walaupun secara tidak langsung, misal menggunakan teori Bayes). *Sequence to sequence* adalah *discriminative model* walaupun *output*-nya berupa sekuens, ia tidak memodelkan $p(x)$ (berbeda dengan *(*hidden markov model). Kita ingin memodelkan *conditional probability* $p(y \mid x)$ secara langsung, seperti *classifier* lainnya (e.g., *logistic regression*). Jadi yang dimodelkan antara *generative* dan *discriminative model* adalah dua hal yang berbeda.

13.5 Arsitektur Lainnya

Selain arsitektur yang sudah dipaparkan, masih banyak arsitektur lain baik bersifat generik (dapat digunakan untuk berbagai karakteristik data) maupun spesifik (cocok untuk data dengan karakteristik tertentu atau permasalahan

tertentu) sebagai contoh, *Restricted Boltzman Machine*[18] dan *Generative Adversarial Network* (GAN)[19]. Saat buku ini ditulis, GAN dan *adversarial training* sedang populer.

13.6 Architecture Ablation

Pada bab 9, kamu telah mempelajari *feature ablation*, yaitu memilih-milih elemen pada *input* (untuk dibuang), sehingga model memiliki kinerja optimal. Pada *neural network*, proses *feature engineering* mungkin tidak sepenting pada model-model yang sudah kamu pelajari sebelumnya (e.g., model linear) karena ia dapat memodelkan interaksi yang kompleks dari seluruh elemen *input*. Pada *neural network*, masalah yang muncul adalah memilih arsitektur yang tepat, seperti menentukan jumlah *hidden layers* (dan berapa unit). Contoh lain adalah memilih fungsi aktivasi yang cocok. Walaupun *nueral network* memberikan kita kemudahan dari segi pemilihan fitur, kita memiliki kesulitan dalam menentukan arsitektur. Terlebih lagi, alasan pemilih suatu jumlah *units* pada suatu *layer* (e.g., 512 dibanding 256 *units*) mungkin tidak dapat dijustifikasi dengan sangat akurat. Pada *feature ablation*, kita dapat menjustifikasi alasan untuk menghilangkan suatu fitur. Pada *neural network*, kita susah menjelaskan alasan pemilihan karena *search space*-nya jauh lebih besar.

Soal Latihan

13.1. POS *tagging* Pada subbab 13.3, disebutkan bahwa *bidirectional recurrent neural network* lebih cocok untuk persoalan POS *tagging*. Jelaskan mengapa! (hint pada bab 8)

13.2. Eksplorasi Jelaskanlah pada teman-temanmu apa dan bagaimana prinsip kerja:

(a) *Restricted Boltzman Machine*
(b) *Generative Adversarial Network*

[18] `https://deeplearning4j.org/restrictedboltzmannmachine`
[19] `https://deeplearning4j.org/generative-adversarial-network`