13

Arsitektur Neural Network

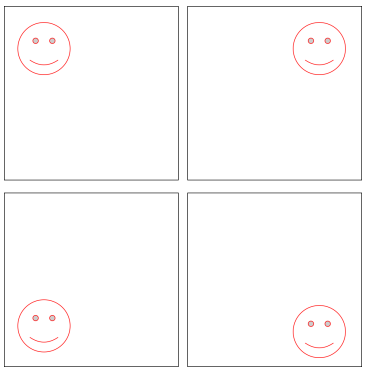
> As students cross the threshold from outside to insider, they also cross the threshold from superficial learning motivated by grades to deep learning motivated by engagement with questions. Their transformation entails an awakening–even, perhaps, a falling in love.
>
> John C. Bean

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.

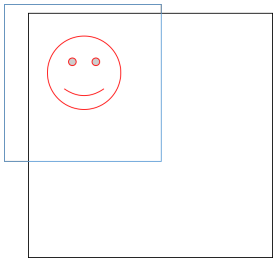
13.1 Convolutional Neural Network

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*).



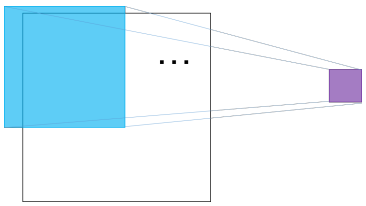
Gambar 13.1. Motivasi *convolutional neural network*



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

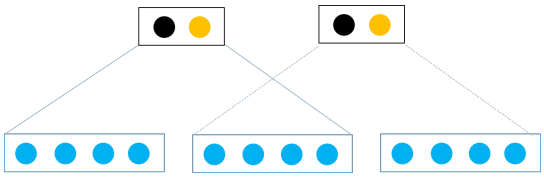
13.1.1 Convolution

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).



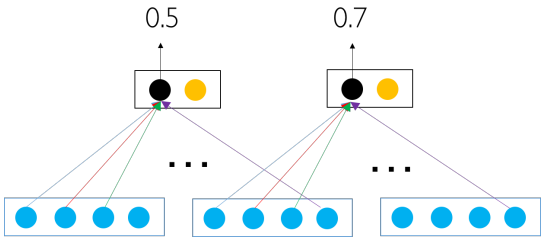
Gambar 13.4. *1D Convolution*

[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*).



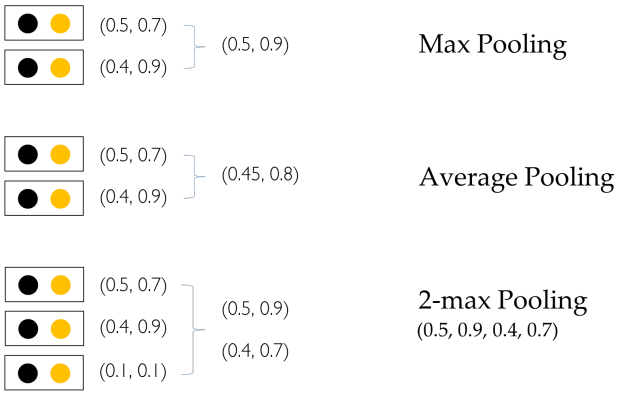
Gambar 13.5. Konsep *weight sharing*

13.1.2 Pooling

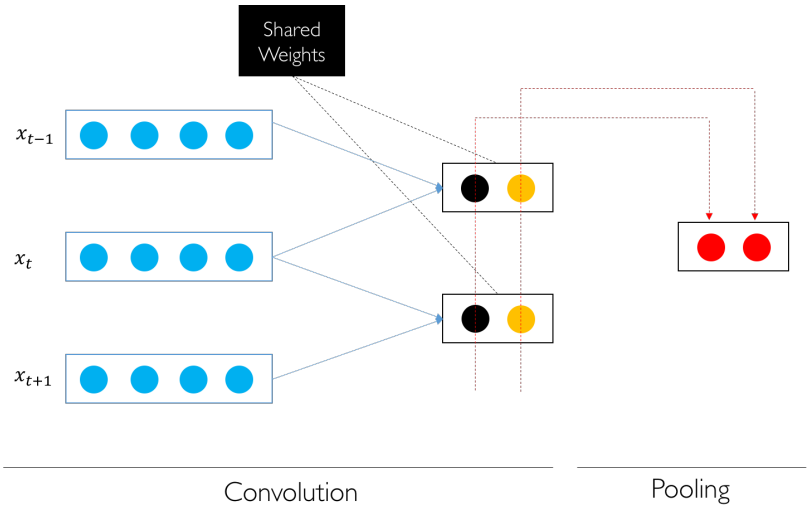
Pada tahap *convolution*, kita merubah setiap *k-sized window* menjadi satu vektor berdimensi d (yang dapat disusun menjadi matriks $\mathbf{D}$). Semua vektor yang dihasilkan pada tahap sebelumnya dikombinasikan (*pooled*) menjadi satu vektor $\mathbf{c}$. Ide utamanya adalah mengekstrak informasi paling 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.

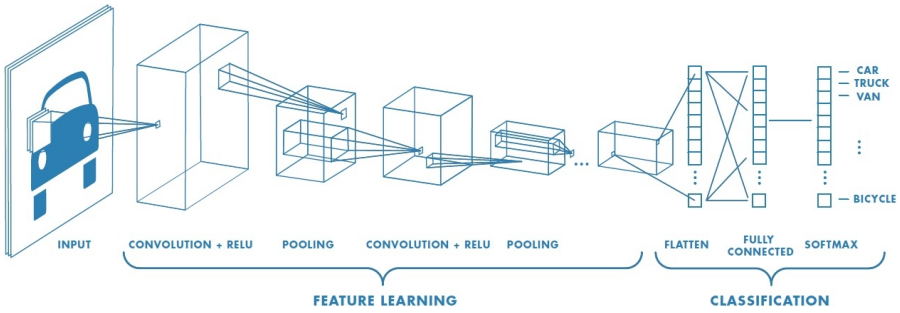


Gambar 13.6. Contoh *pooling*



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

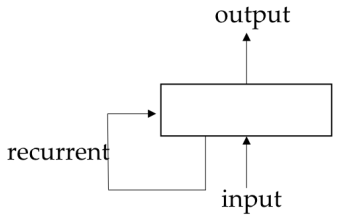
[4] `mathworks.com`

[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

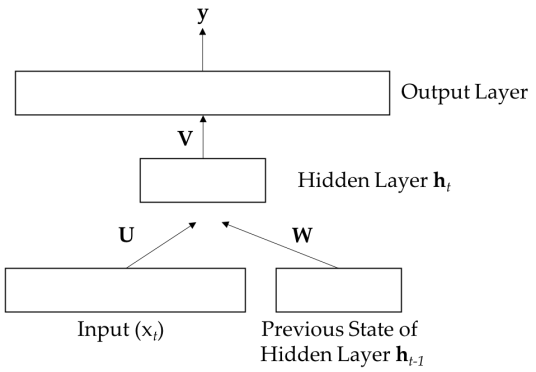
Ilustrasi Gambar. 13.9 mungkin sedikit susah dipahami karena berbentuk sangat konseptual. Bentuk lebih matematis diilustrasikan pada Gambar. 13.10 [72]. Perhitungan *hidden state* pada waktu ke-t bergantung pada *input* pada waktu ke-t (x_t) dan *hidden state* pada waktu sebelumnya (h_{t-1}).

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

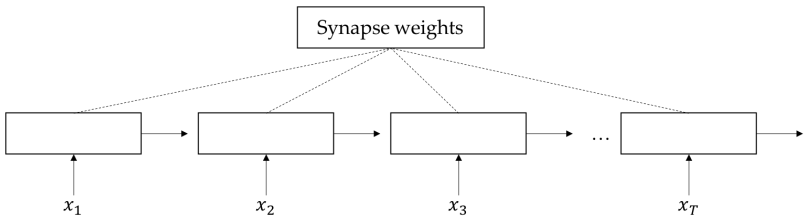
$$\mathbf{h}_t = f(x_t, \mathbf{h}_{t-1}, b) \tag{13.1}$$

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.

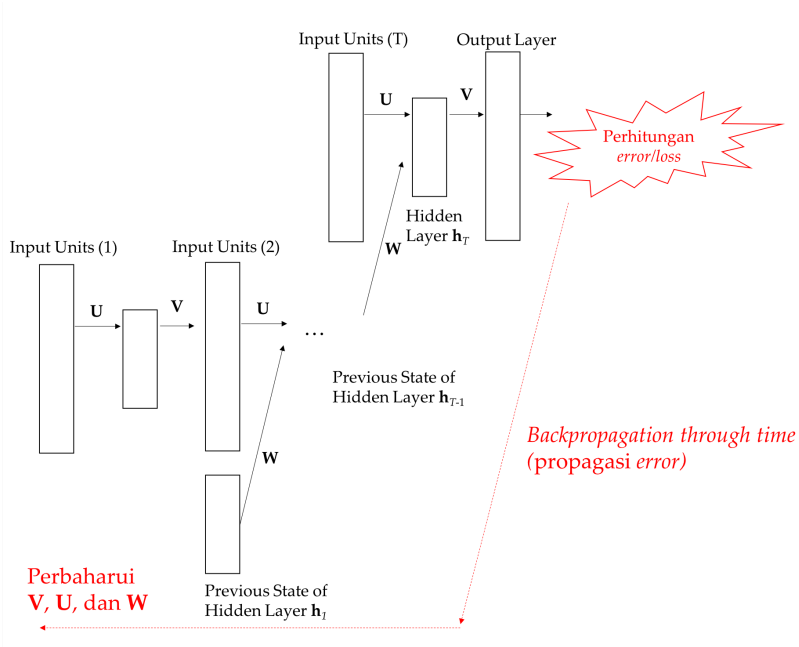
$$\begin{aligned}\mathbf{h}_t &= f(x_t, \mathbf{h}_{t-1}) \\ &= f(x_t, f(x_{t-1}, \mathbf{h}_{t-2})) \\ &= f(x_t, f(x_{t-1}, f(\{x_1, \cdots, x_{t-2}\}, \{\mathbf{h}_1, \cdots, \mathbf{h}_{t-3}\})))\end{aligned} \tag{13.2}$$

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`

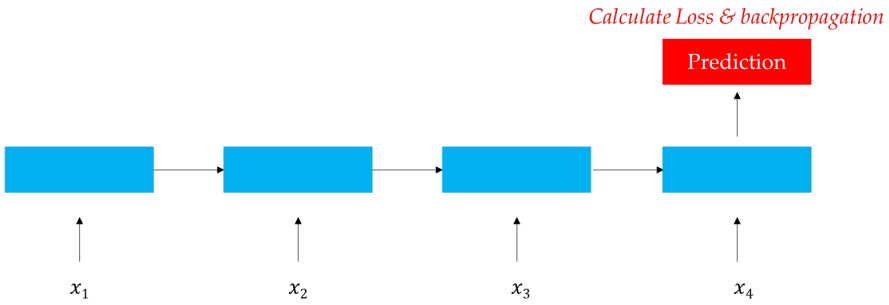
tersebut ke RNN, diilustrasikan pada Gambar. 13.11. Perlu diingat, RNN mengadopsi prinsip *parameter sharing* (serupa dengan *weight sharing* pada CNN) dimana neuron yang sama diulang-ulang saat process *feed forward*. Setelah selesai proses *feed forward*, kita memperbaharui parameter (*synapse weights*) 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`

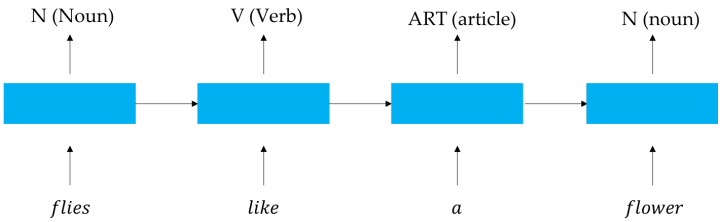
13.3 Part-of-speech Tagging Revisited

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.

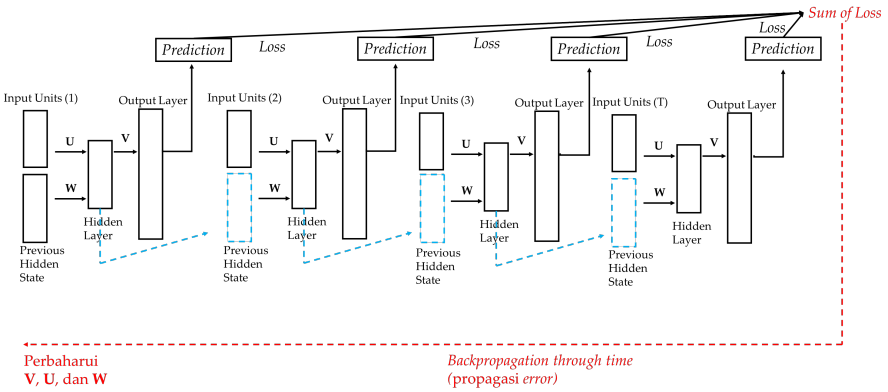
$$y_i = \underset{t_i \in C}{\arg\max}\, p(t_i | x_i) \tag{13.6}$$

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.

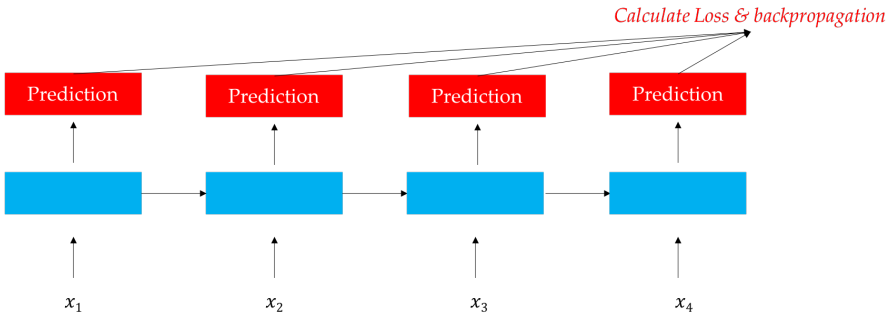


Gambar 13.15. *Sequence prediction* menggunakan RNN

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`



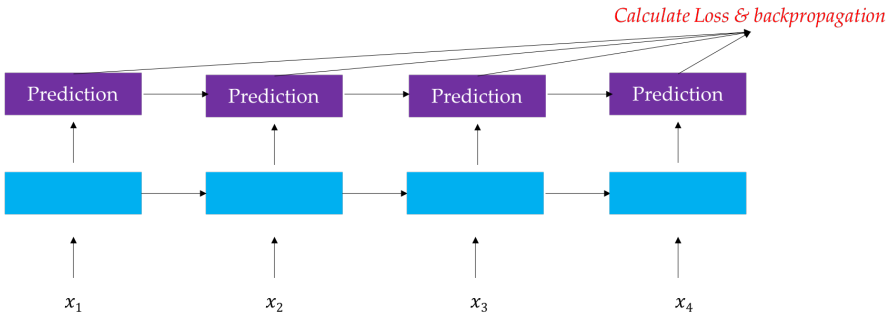
Gambar 13.16. *Sequence prediction* menggunakan RNN (disederhakan) [1]. Persegi berwarna merah umumnya melambangkan *multi-layer perceptron*

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.

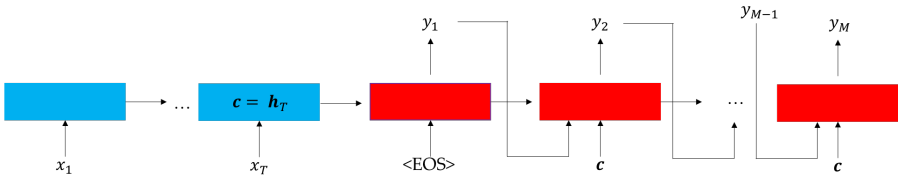
$$P(\mathbf{y} \mid \mathbf{x}, \theta) = \prod_{t=1}^{M} P(y_t \mid \{y_1, \ldots, y_{t-1}\}, \mathbf{x}, \theta), \tag{13.10}$$

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* $\mathbf{e}'_t$.

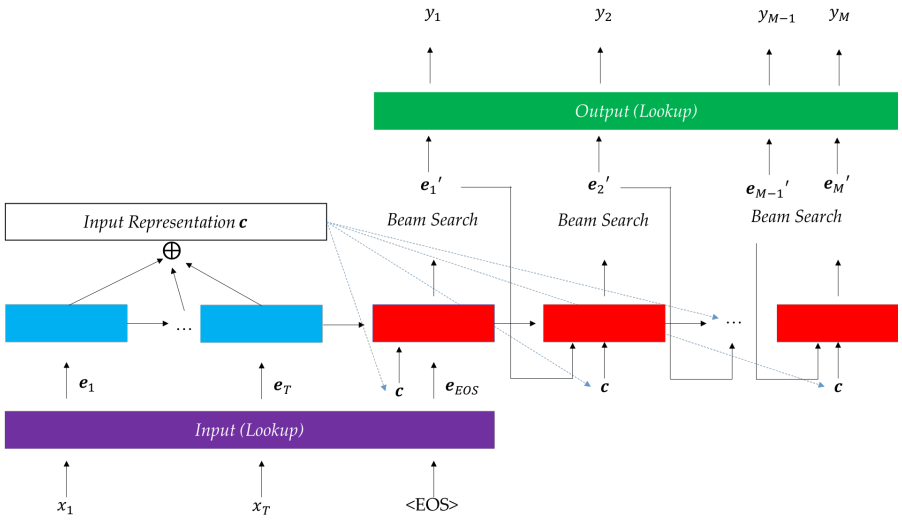
Dengan penjelasan ini, mungkin pembaca berpikir Gambar. 13.18 tidak lengkap. Kamu benar! Penulis sengaja memberikan gambar simplifikasi. Gambar lebih lengkap (dan lebih nyata) diilustrasikan pada Gambar. 13.19.

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]

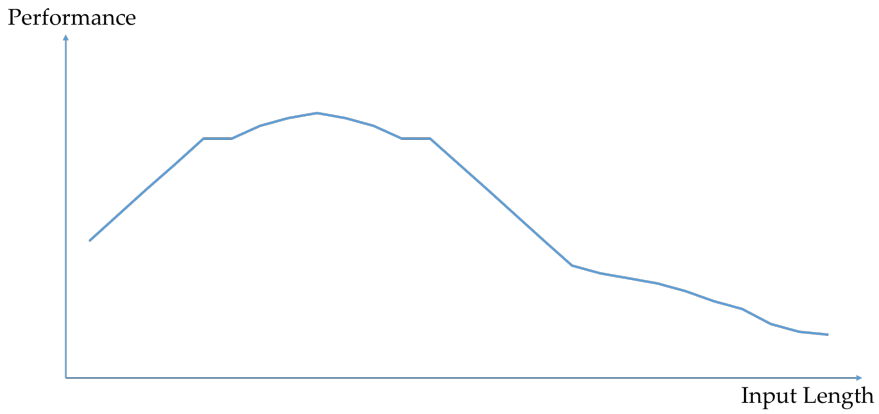
Secara sederhana, algoritma *Beam Search* mirip dengan algoritma Viterbi yang sudah dijelaskan pada bab 8, yaitu algoritma untuk mencari sekuens dengan probabilitas tertinggi. Perbedaannya terletak pada *heuristic*. Untuk menghemat memori komputer, algoritma *Beam Search* melakukan ekspansi terbatas. 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.

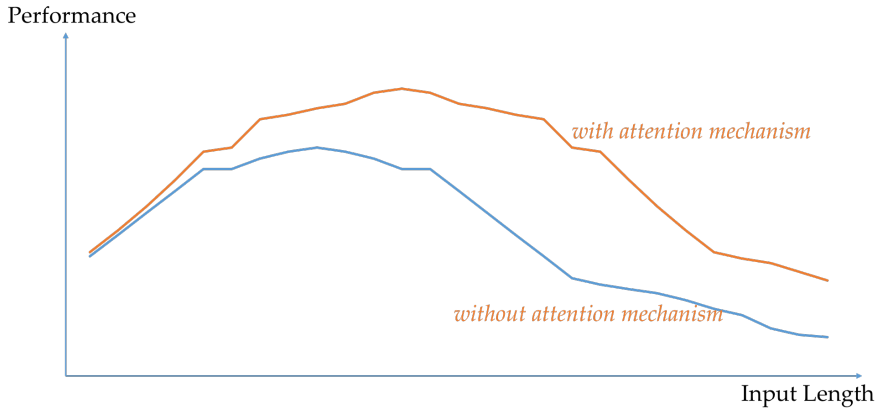


Gambar 13.21. Permasalahan *input* yang panjang

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

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

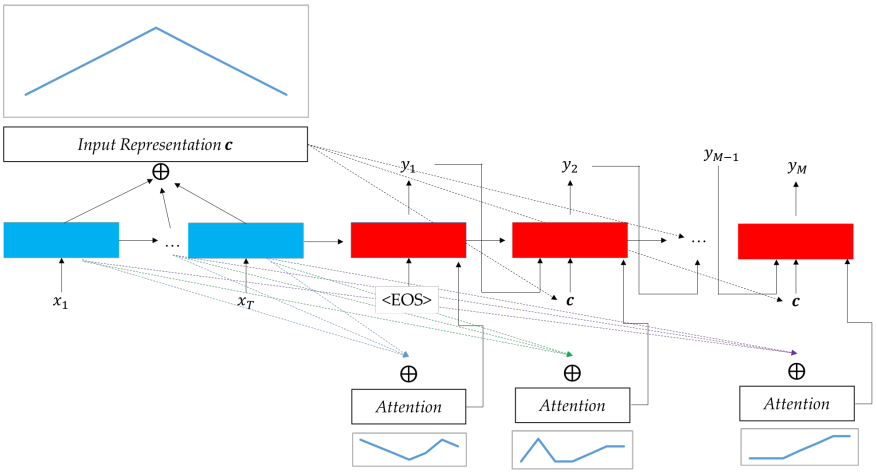
dimana $\mathbf{k}_t$ merepresentasikan seberapa (*how much*) *decoder* harus memfokuskan diri ke *hidden state* tertentu pada *encoder* untuk menghasilkan *output* saat ke-t. $\mathbf{k}_t$ dapat dihitung pada persamaan 13.17

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

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.



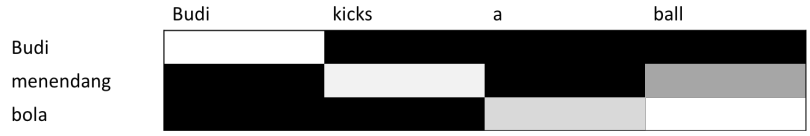
Gambar 13.23. *Encoder-decoder with attention*

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

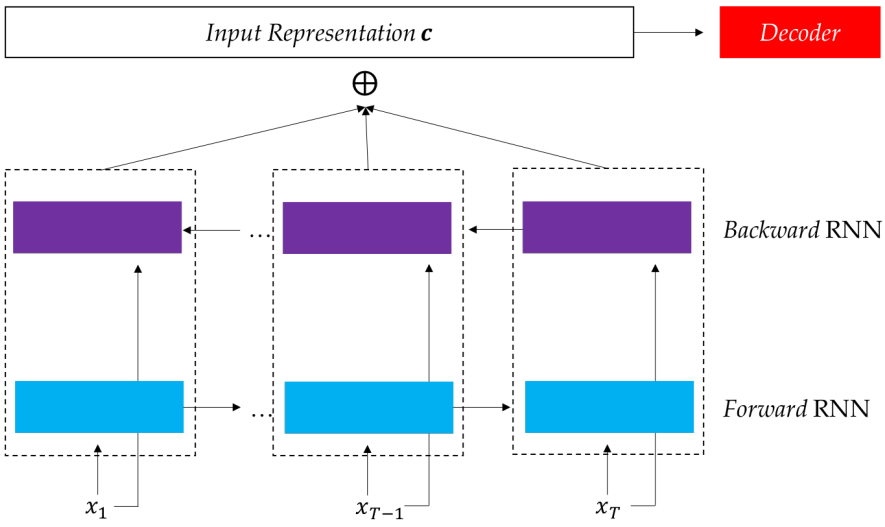
Selain RNN, kita juga dapat menggunakan *bidirectional* RNN (BiRNN) untuk mengikutsertakan pengaruh baik *hidden state* sebelum $(\mathbf{h}_1, \cdots, \mathbf{h}_{t-1})$ dan setelah $(\mathbf{h}_{t+1}, \cdots, \mathbf{h}_T)$ untuk menghitung *hidden state* sekarang $(\mathbf{h}_t)$ [80, 81, 82]. BiRNN 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



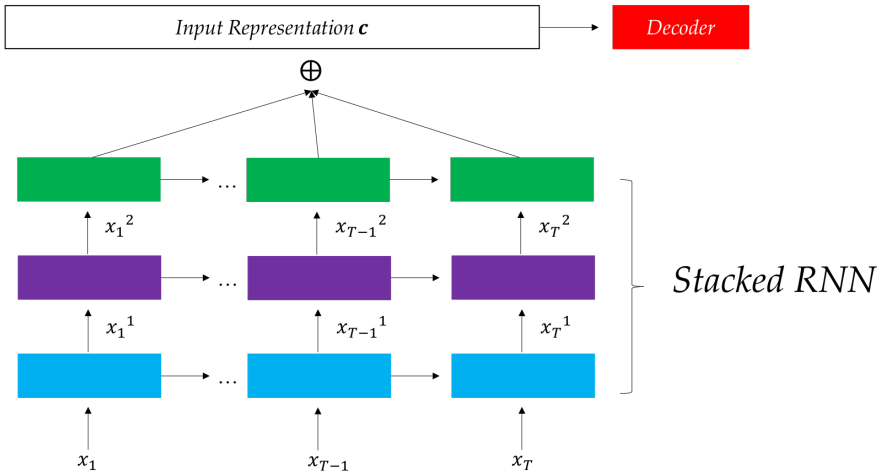
Gambar 13.24. *Attention mechanism* pada translasi bahasa [52]. Warna lebih terang merepresentasikan bobot (fokus/*attention*) lebih tinggi. Sebagai contoh, kata "menendang" berkorespondensi paling erat dengan kata "kicks"

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`