
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

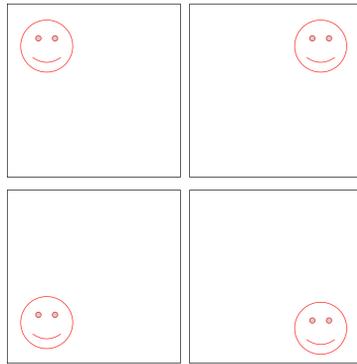
Seperti yang sudah dijelaskan pada bab 9 dan bab 12, data memiliki karakteristik (dari segi struktur) misal *sequential data*, *compositional data*, dsb. Terdapat arsitektur khusus *artificial neural network* (ANN) untuk menyelesaikan persoalan pada tipe data tertentu. Pada bab ini, kami akan memberikan 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. Penjelasan pada bab ini bersifat abstrak dan kamu harus mengerti penjelasan bab-bab sebelumnya untuk mengerti konsep pada bab ini.

13.1 Convolutional Neural Network

Subbab ini akan memaparkan **ide utama** dari *convolutional neural network* (CNN) berdasarkan *paper* asli dari LeCun dan Bengio [76] (saat buku ini ditulis sudah ada banyak variasi). CNN memiliki banyak istilah dari bidang pemrosesan gambar (karena dicetuskan dari bidang tersebut), tetapi demi

mempermudah pemahaman intuisi CNN, diikat ini akan menggunakan istilah yang lebih umum juga.

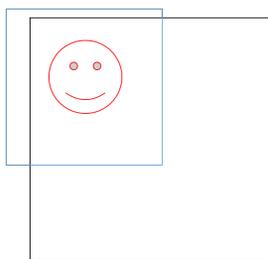
Sekarang, mari kita memasuki cerita CNN dari segi pemrosesan gambar. Objek bisa saja terlatak pada berbagai macam posisi seperti diilustrasikan oleh Gambar. 13.1. Selain tantangan variasi posisi objek, masih ada juga tantangan lain seperti rotasi objek dan perbedaan ukuran objek (*scaling*). Kita ingin 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 pola *feature vector* yang mirip walaupun posisinya digeser-geser. ANN (MLP) bisa juga mempelajari prinsip *translation invariance*, tetapi memerlukan jauh lebih banyak parameter dibanding CNN (subbab berikutnya secara lebih matematis) yang memang didesain dengan prinsip *translation invariance* (“*built-in*”).



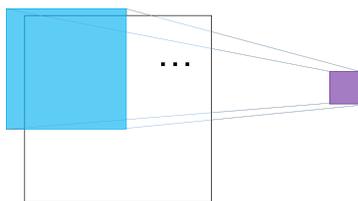
Gambar 13.1: Motivasi *convolutional neural network*.

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

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

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

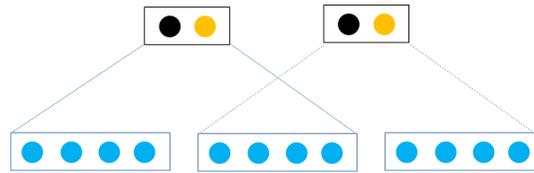
Gambar 13.3: *Sliding window*.

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).

Pada contoh sebelumnya, kita menggunakan *window* selebar 2, satu *window* mencakup 2 data lokal; i.e., $window_1 = (x_1, x_2)$, $window_2 = (x_2, x_3)$, ...; untuk suatu *input* \mathbf{x} . Kita juga dapat mempergunakan ***stride*** sebesar s ,

¹ Dikenal juga sebagai *receptive field*.

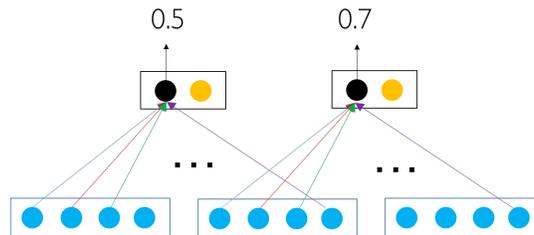
² Istilah *convolution* yang diterangkan pada konteks *machine learning* memiliki arti yang berbeda pada bidang *signal processing*.



Gambar 13.4: 1D Convolution.

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., $window_1 = (x_1, x_2)$, $window_2 = (x_3, x_4), \dots$

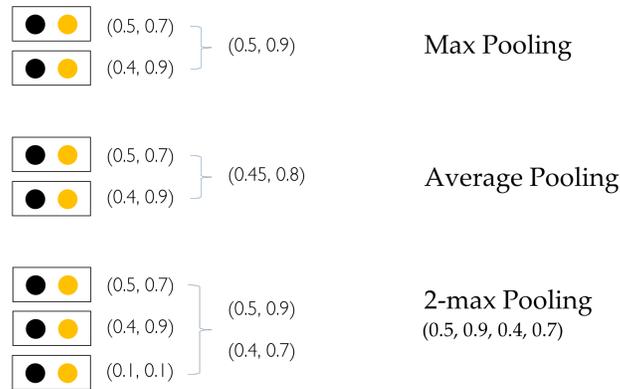
Selain *sliding window* dan *filter*, *convolutional layer* juga mengadopsi prinsip *weight sharing*. Artinya, *synapse weights* untuk suatu filter adalah sama walau *filter* tersebut dipergunakan untuk berbagai *window*. Sebagai ilustrasi, perhatikan Gambar. 13.5, warna yang sama pada *synapse weights* 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* berwarna 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* 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.



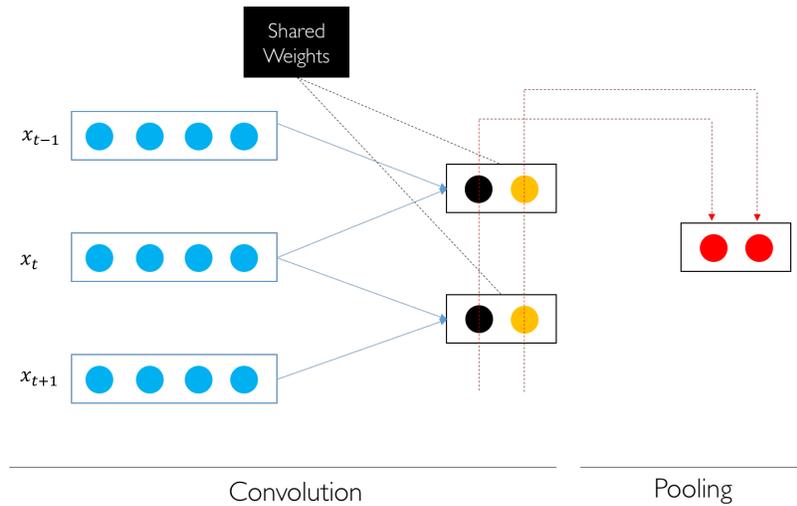
Gambar 13.6: Contoh *pooling*.

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

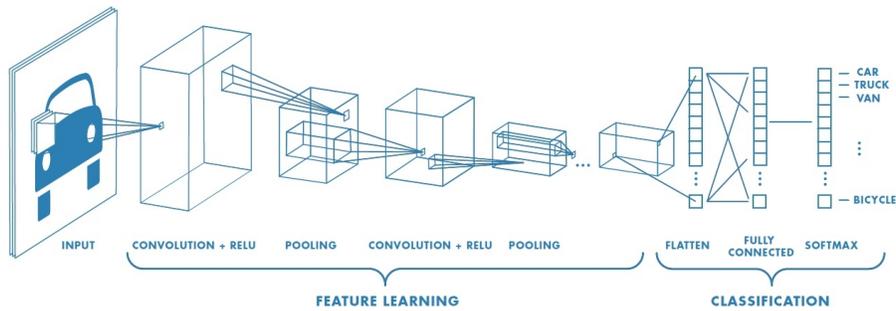
13.1.3 Rangkuman

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

³ Kami ingin pembaca mengeksplorasi sendiri *dynamic pooling*.



Gambar 13.7: *Convolution* dan *pooling*.



Gambar 13.8: *Convolutional Neural Network*.⁴

sendiri, dalam artian CNN biasanya digunakan (dikombinasikan) pada 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* [77], misalkan data ramalan cuaca. Cuaca hari ini bergantung kurang lebih pada cuaca hari sebelumnya. Sebagai contoh apabila hari sebelumnya

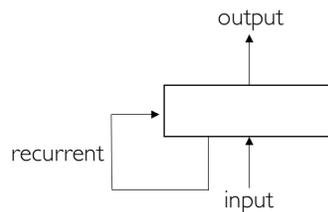
⁴ mathworks.com

mendung, ada kemungkinan hari ini hujan.⁵ Walau ada yang menganggap sifat data sekuensial dan *time series* berbeda, RNN berfokus sifat data dimana *instance* waktu sebelumnya ($t-1$) mempengaruhi *instance* pada waktu berikutnya (t). Intinya, mampu mengingat *history*.

Secara lebih umum, diberikan sebuah sekuens *input* $\mathbf{x} = (x_1, \dots, x_T)$. Data x_t (e.g., vektor, gambar, teks, suara) dipengaruhi oleh data sebelum-sebelumnya (*history*), ditulis sebagai $P(x_t | \{x_1, \dots, x_{t-1}\})$. Kami harap 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 menganggap informasi yang dimuat oleh data lama dapat direpresentasikan oleh data lebih baru, i.e., x_t juga memuat informasi x_{t-J}, \dots, x_{t-1} ; 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 memorsisasi,⁶ kita ingin mengingat **keseluruhan** sekuens (dibanding *markov assumption* yang mengingat sekuens secara terbatas), implikasinya adalah RNN yang 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.



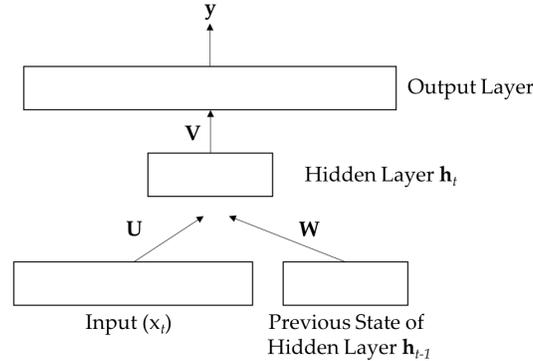
Gambar 13.9: Bentuk konseptual paling sederhana Recurrent Neural Network.

Ilustrasi Gambar. 13.9 mungkin sedikit susah dipahami karena berbentuk sangat konseptual. Bentuk lebih matematis diilustrasikan pada Gam-

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

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

bar. 13.10 [77]. 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}).



Gambar 13.10: Konsep Recurrent Neural Network.

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

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

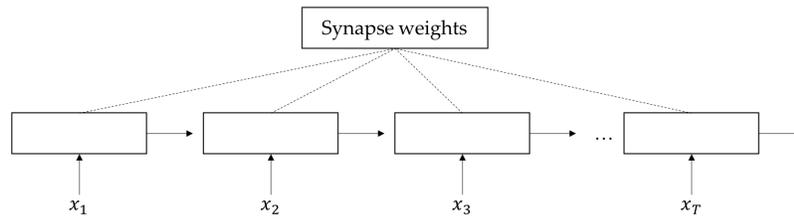
dimana f adalah fungsi aktivasi (non-linear, dapat diturunkan). Demi menyederhanakan penjelasan, penulis tidak mengikutsertakan *bias* (b) pada fungsi-fungsi berikutnya. Kami berharap pembaca selalu mengingat bahwa *bias* adalah parameter yang diikutsertakan pada fungsi *artificial neural network*. Fungsi f dapat diganti dengan variasi *neural network*,⁷ misal menggunakan *long short-term memory network* (LSTM) [78]. 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.

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

Training pada *recurrent neural network* dapat menggunakan metode *back-propagation*. 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* [79].

⁷ https://en.wikipedia.org/wiki/Recurrent_neural_network



Gambar 13.11: Konsep *feed forward* pada Recurrent Neural Network (RNN). Karena RNN menerima *input* berupa sekuens, kita memvisualisasikan proses *feed forward* dengan *unfolding* (atau *unrolling*) RNN pada keseluruhan sekuens input.

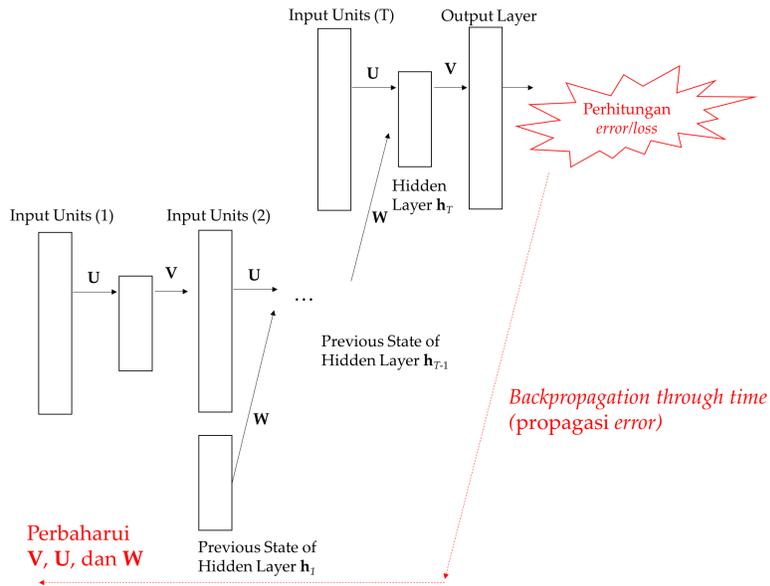
Sebagai contoh kita diberikan sebuah sekuens \mathbf{x} dengan panjang T sebagai input, dimana x_t melambangkan input ke- i (**data point**), dapat berupa vektor, gambar, teks, atau apapun. Kita melakukan *feed forward* data tersebut ke RNN, diilustrasikan pada Gambar. 13.11. Perlu diingat, RNN mengadopsi prinsip *parameter sharing* (serupa dengan *weight sharing* pada CNN) dimana neuron yang sama diulang-ulang saat proses *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.

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 program), penulis sarankan pembaca untuk melihat materi tentang ***computation graph***⁹ dan disertasi PhD oleh Mikolov [47].

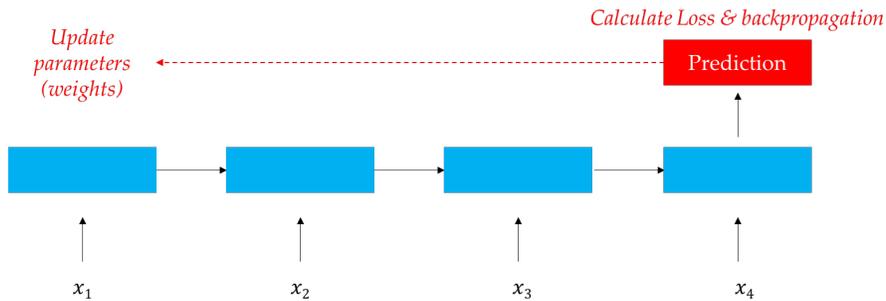
Walaupun secara konseptual RNN dapat mengingat seluruh kejadian sebelumnya, hal tersebut sulit untuk dilakukan secara praktikal untuk sekuens yang panjang. Hal ini lebih dikenal dengan *vanishing* atau *exploding gradient problem* [63, 80, 81]. Seperti yang sudah dijelaskan, ANN dan variasi arsitekturnya dilatih menggunakan teknik *stochastic gradient descent* (*gradient-based optimization*). Artinya, kita mengandalkan propagasi *error* berdasarkan tu-

⁸ Prinsip ini mirip dengan *weight sharing*.

⁹ <https://www.coursera.org/learn/neural-networks-deep-learning/lecture/4Wd0Y/computation-graph>



Gambar 13.12: Konsep *backpropagation through time* [47].



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

runan. 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 tersebut yang dipropagasikan pada langkah berikutnya menjadi semakin kecil. 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 y . Vektor y dapat digunakan un-

tuk 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.

$$\mathbf{y} = \text{RNN}(x_1, \dots, x_N) \quad (13.3)$$

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

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

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*) [82, 83, 84]. Kami harap kamu masih ingat materi bab 8 yang membahas apa itu persoalan POS *tagging*.

Diberikan sebuah sekuens kata $\mathbf{x} = \{x_1, \dots, x_T\}$, kita ingin mencari sekuens *output* $\mathbf{y} = \{y_1, \dots, y_T\}$ (*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 .

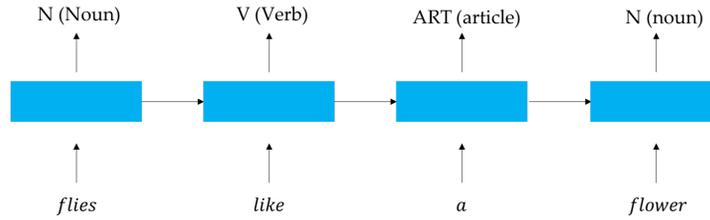
Ilustrasi diberikan oleh Gambar. 13.14.

$$y_1, \dots, y_T = \arg \max_{t_1, \dots, t_T; t_i \in C} P(t_1, \dots, t_T | x_1, \dots, x_T) \quad (13.5)$$

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.

$$y_i = \arg \max_{t_i \in C} P(t_i | x_i) \quad (13.6)$$

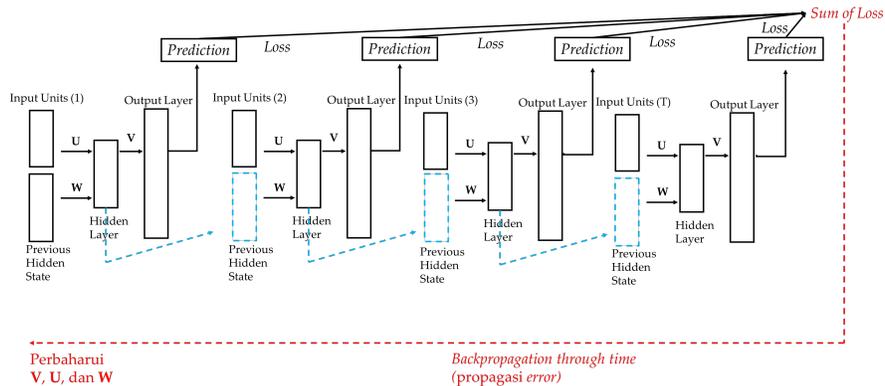
Akan tetapi, seperti yang sudah dibahas sebelum sebelumnya, *markov assumption* memiliki kelemahan. Kelemahan utama adalah tidak menggunakan



Gambar 13.14: POS *tagging* menggunakan Recurrent Neural Network.

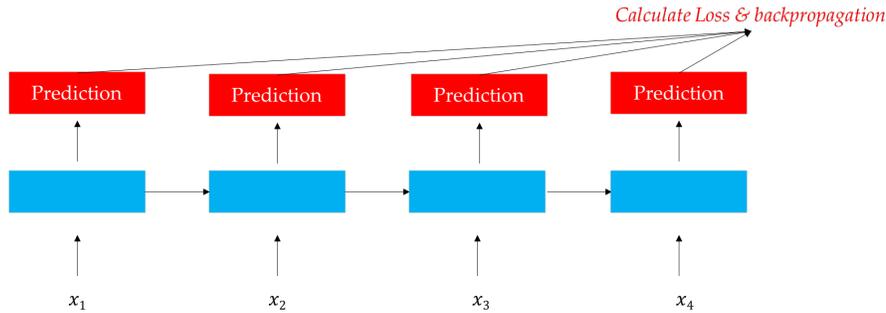
keseluruhan *history*. Persoalan ini cocok untuk diselesaikan oleh RNN karena kemampuannya untuk mengingat seluruh sekuens (berbeda dengan *hidden markov model* (HMM) yang menggunakan *markov assumption*). Secara teoritis (dan juga praktis—sejauh yang penulis ketahui.) Pada banyak persoalan, RNN menghasilkan performa yang lebih baik dibanding HMM. Tetapi hal ini bergantung juga pada variasi arsitektur. Dengan ini, persoalan POS *tagging* (*full history*) diilustrasikan oleh persamaan 13.7.

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



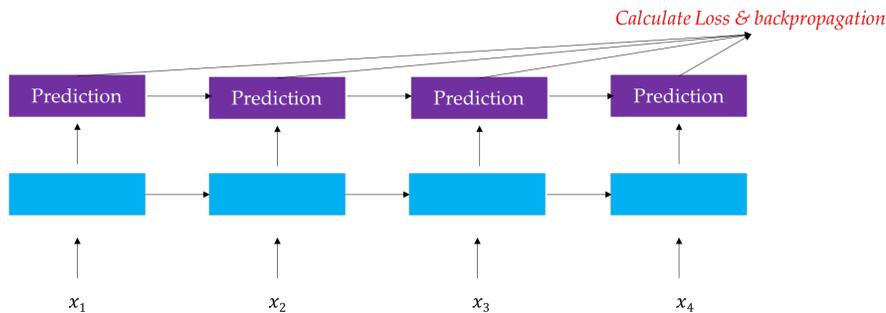
Gambar 13.15: *Sequence prediction* menggunakan Recurrent Neural Network.

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 melewati setiap *hidden layer* pada RNN pada suatu jaringan (anggap sebuah MLP+softmax).



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

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*.¹⁰ 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 13.17: *Sequence prediction* menggunakan Recurrent Neural Network (disederhakan), dimana prediksi saat waktu ke- t dipengaruhi oleh hasil prediksi pada waktu $t - 1$.

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

¹⁰ https://en.wikipedia.org/wiki/Named-entity_recognition

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

$$y_i = \arg \max_{t_i \in C} P(t_i | y_1, \dots, y_{i-1}, x_1, \dots, x_i) \quad (13.8)$$

Salah satu strategi untuk menangani hal tersebut adalah dengan melewatkan POS *tag* pada sebuah RNN juga, seperti pada 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).

$$P(t_i | y_1, \dots, y_{i-1}, x_1, \dots, x_i) = \text{softmax}(\text{MLP}([\text{RNN}^x(x_1, \dots, x_i); \text{RNN}^{\text{tag}}(t_1, \dots, t_{i-1})])) \quad (13.9)$$

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, \dots, y_{i-1}) dan sebuah *conditioning context* \mathbf{c} (berupa vektor).

Arsitektur yang dibahas pada subbab ini adalah variasi RNN untuk permasalahan *sequence generation*.¹¹ Diberikan sekuens *input* $\mathbf{x} = (x_1, \dots, x_T)$. Kita ingin mencari sekuens *output* $\mathbf{y} = (y_1, \dots, y_M)$. Pada subbab sebelumnya, x_i berkorespondensi langsung dengan y_i , e.g., 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} ; i.e., *conditioning context* dan *history* $\{y_1, \dots, y_{i-1}\}$. Panjang sekuens *output* M tidak mesti sama dengan panjang sekuens *input* T . Permasalahan ini masuk ke dalam kerangka *conditioned generation* dimana keseluruhan *input* \mathbf{x} dapat direpresentasikan menjadi sebuah vektor \mathbf{c} (*coding*). Vektor \mathbf{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-*paraphrase*, dsb. Artinya ada sebuah *input* dan kita ingin menghasilkan (*generate/produce*) sebuah *output* yang cocok untuk *input* tersebut. Hal ini dapat dicapai dengan memodelkan pasangan *input-output* $p(\mathbf{y} | \mathbf{x})$. Umumnya, kita mengasumsikan ada kumpulan parameter θ yang mengontrol *conditional probability*, sehingga kita transformasi *conditional probability* menjadi $p(\mathbf{y} | \mathbf{x}, \theta)$. *Conditional probability* $p(\mathbf{y} | \mathbf{x}, \theta)$ dapat difaktorkan sebagai persamaan 13.10. Kami harap kamu mampu membedakan persamaan 13.10 dan persamaan 13.5 (dan 13.8) dengan jeli. Sedikit

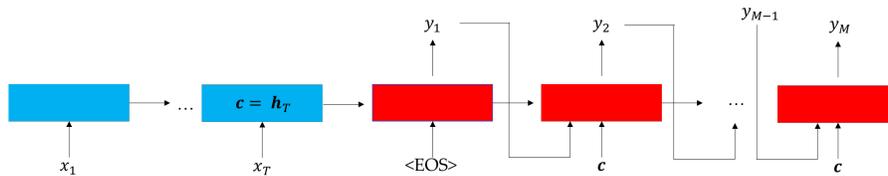
¹¹ Umumnya untuk bidang pemrosesan bahasa alami.

perbedaan pada formula menyebabkan makna yang berbeda. Objektif *training* adalah untuk meminimalkan *loss function*, sebagai contoh berbentuk *log likelihood function* diberikan pada persamaan 13.11, dimana \mathbf{D} melambangkan *training data*.¹²

$$p(\mathbf{y} | \mathbf{x}, \theta) = \prod_{t=1}^M p(y_t | \{y_1, \dots, y_{t-1}\}, \mathbf{x}, \theta), \quad (13.10)$$

$$L(\theta) = - \sum_{\{\mathbf{x}, \mathbf{y}\} \in \mathbf{D}} \log p(\mathbf{y} | \mathbf{x}, \theta) \quad (13.11)$$

Persamaan 13.10 dapat dimodelkan dengan *encoder-decoder* model yang terdiri dari dua buah RNN dimana satu RNN sebagai *encoder*, satu lagi sebagai *decoder*. *Neural Network*, pada kasus ini, bertindak sebagai *controlling parameter* θ . 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*.



Gambar 13.18: Konsep *encoder-decoder* [81]. “<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)* bertindak sebagai kumpulan parameter θ yang mengatur *conditional probability*. *Encoder-decoder* juga dilatih menggunakan prinsip *gradient-based optimization* untuk *tuning* parameter yang mengkondisikan *conditional probability* [81]. Dengan ini, persamaan 13.10 sudah didefinisikan sebagai *neural network* sebagai persamaan 13.12. “enc” dan “dec” adalah fungsi *encoder* dan *decoder*, yaitu sekumpulan transformasi non-linear.

¹² Ingat kembali materi *cross entropy*!

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

$$y_t = \text{dec}(\{y_1, \dots, y_{t-1}\}, \text{enc}(\mathbf{x}), \theta) \quad (13.12)$$

Begitu model dilatih, *encoder-decoder* akan mencari *output* $\hat{\mathbf{y}}$ terbaik untuk suatu input \mathbf{x} , dillustrasikan pada persamaan 13.13. 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.

$$\hat{\mathbf{y}} = \arg \max_{\mathbf{y}} p(\mathbf{y} | \mathbf{x}, \theta) \quad (13.13)$$

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

$$\begin{aligned} \mathbf{h}_t &= f(\mathbf{h}_{t-1}, \mathbf{e}_t) \\ &= f(\mathbf{h}_{t-1}\mathbf{U} + \mathbf{e}_t\mathbf{W}) \end{aligned} \quad (13.14)$$

Representasi *input* \mathbf{c} dihitung dengan persamaan 13.15, yaitu sebagai *weighted sum* dari *hidden states* [57], dimana q adalah fungsi aktivasi non-linear. Secara lebih sederhana, kita boleh langsung menggunakan \mathbf{h}_T sebagai konteks \mathbf{c} [81] karena kita mengasumsikan \mathbf{h}_T mengandung seluruh informasi yang ada di *input*.

$$\mathbf{c} = q(\{\mathbf{h}_1, \dots, \mathbf{h}_T\}) \quad (13.15)$$

Walaupun disebut sebagai representasi keseluruhan sekuens *input*, informasi awal pada *input* yang panjang dapat hilang. Artinya \mathbf{c} bisa saja memuat lebih banyak informasi *input* ujung-ujung akhir. Salah satu strategi yang dapat digunakan adalah dengan membalik (*reversing*) sekuens *input*. Sebagai contoh, *input* $\mathbf{x} = (x_1, \dots, x_T)$ dibalik menjadi (x_T, \dots, x_1) agar bagian awal (\dots, x_2, x_1) lebih dekat dengan *decoder* [81]. Informasi yang berada dekat dengan *decoder* cenderung lebih diingat. Kami ingin pembaca mengingat bahwa teknik ini pun tidaklah sempurna.

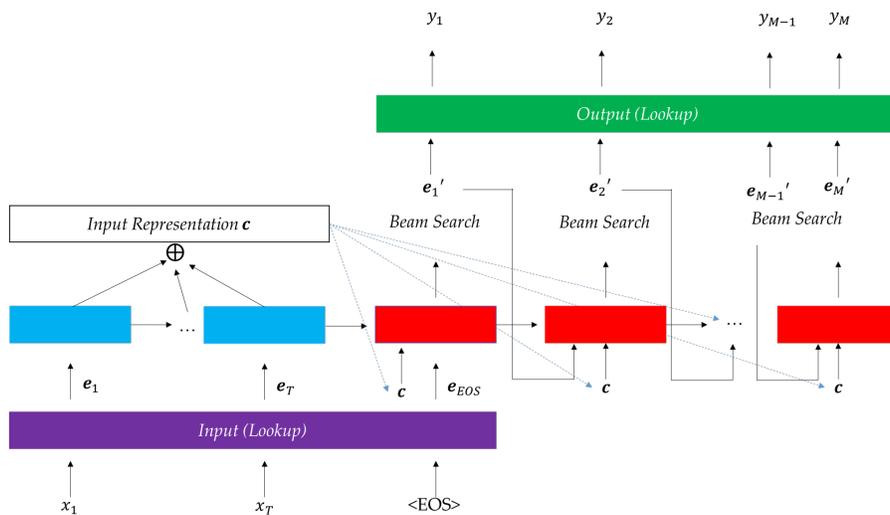
13.4.2 Decoder

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, \dots, y_M) , disebut sebagai proses **decoding**. Mirip dengan *encoder*, kita menggunakan RNN untuk menghasilkan *output* seperti diilustrasikan pada persamaan 13.16, 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 .

$$\begin{aligned} \mathbf{h}'_t &= f(\mathbf{h}'_{t-1}, \mathbf{e}'_{t-1}, \mathbf{c}) \\ &= f(\mathbf{h}'_{t-1}\mathbf{H} + \mathbf{e}'_{t-1}\mathbf{E} + \mathbf{cC}) \end{aligned} \tag{13.16}$$

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.



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

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 (e.g., *word embedding* menjadi kata). Komponen “*Beam Search*” dijelaskan pada subbab berikutnya.

13.4.3 Beam Search

Kita ingin mencari sekuens *output* yang memaksimalkan nilai probabilitas pada persamaan 13.13. Artinya, kita ingin mencari *output* terbaik. Pada su-

atu 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

```

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 $P(y_t | y_{t-1})$ 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 buatan.¹⁶ Pseudo-code *Beam Search* diberikan pada Gambar. 13.20 (*direct quotation*).

13.4.4 Attention-based Mechanism

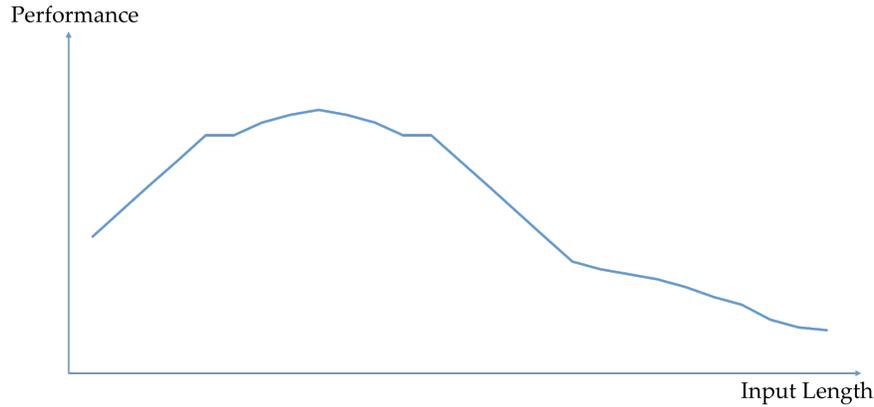
Seperti yang sudah dijelaskan sebelumnya, model *encoder-decoder* memiliki masalah saat diberikan sekuens yang panjang (*vanishing* atau *exploding gradient problem*). Kinerja model dibandingkan dengan panjang *input* kurang lebih dapat diilustrasikan pada Gambar. 13.21. Secara sederhana, kinerja model menurun seiring sekuens input bertambah panjang. Selain itu, representasi \mathbf{c} yang dihasilkan *encoder* harus memuat informasi keseluruhan *input* walaupun sulit dilakukan. Ditambah lagi, *decoder* menggunakan representasinya \mathbf{c} saja tanpa boleh melihat bagian-bagian khusus *input* saat *decoding*. Hal ini tidak sesuai dengan cara kerja manusia, misalnya pada kasus

¹⁴ https://en.wikipedia.org/wiki/Beam_search

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

¹⁶ <https://www.youtube.com/watch?v=j1H3jAAG1EA&t=2131s>

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.

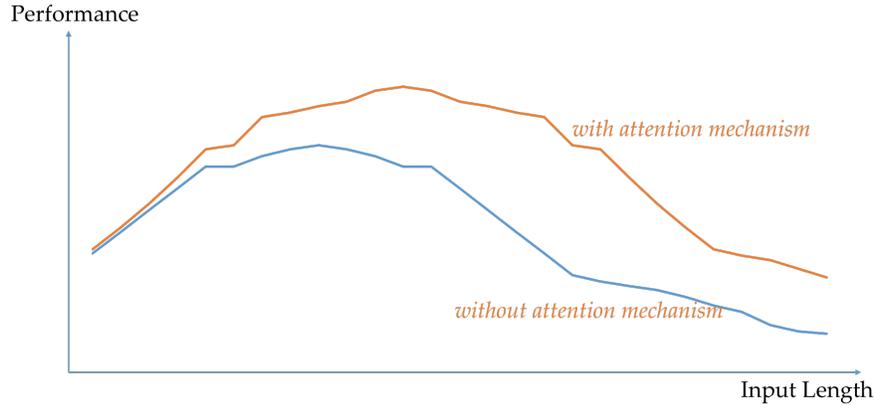


Gambar 13.21: Permasalahan *input* yang panjang.

Sudah dijelaskan sebelumnya bahwa representasi sekuens *input* \mathbf{c} adalah sebuah *weighted sum*. \mathbf{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** [57, 58]. Ide ini sangat 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.

Dengan menggunakan *attention mechanism*, kita dapat mentransformasi persamaan 13.16 pada *decoder* menjadi persamaan 13.17, 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 13.18, 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 $t - 1$.

$$\mathbf{h}'_t = f'(\mathbf{h}'_{t-1}, \mathbf{e}'_{t-1}, \mathbf{c}, \mathbf{k}_t) \quad (13.17)$$

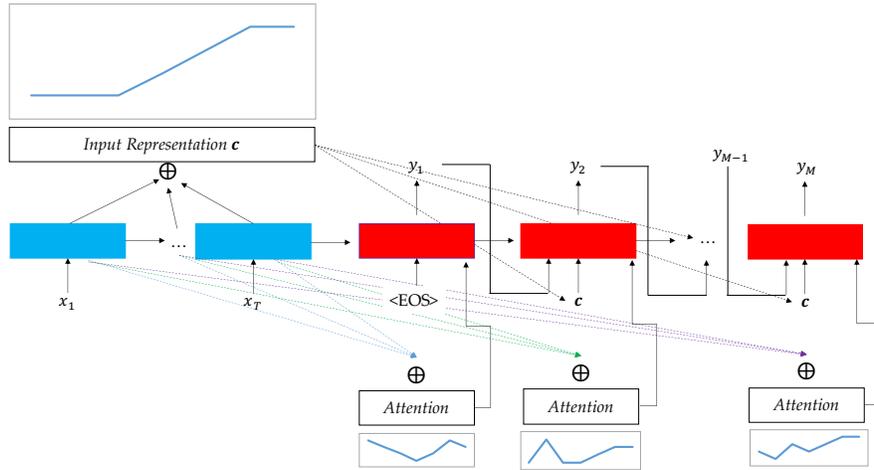
Gambar 13.22: Menggunakan vs tidak menggunakan *attention*.

$$\mathbf{k}_t = \sum_{i=1}^T \alpha_{t,i} \mathbf{h}_i \quad (13.18)$$

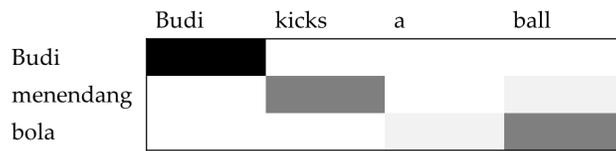
$$\alpha_{t,i} = \frac{\exp(\mathbf{h}_i \cdot \mathbf{h}'_{t-1})}{\sum_{z=1}^T \exp(\mathbf{h}_z \cdot \mathbf{h}'_{t-1})}$$

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* \mathbf{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.

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 [57]. *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 [56].



Gambar 13.23: *Encoder-decoder with attention.*



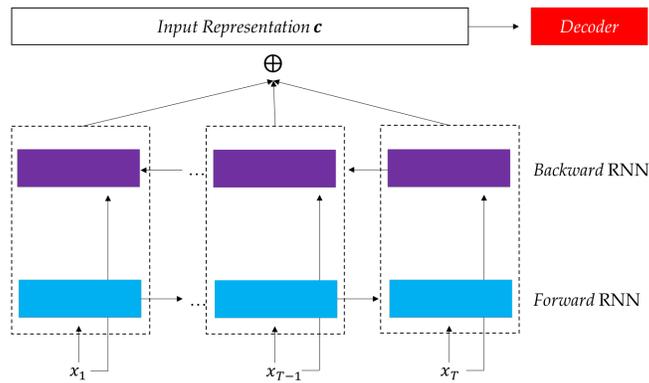
Gambar 13.24: *Attention mechanism* pada translasi bahasa [57]. Warna lebih gelap merepresentasikan bobot (fokus/*attention*) lebih tinggi. Sebagai contoh, kata “menendang” berkorespondensi paling erat dengan kata “kicks”.

13.4.5 Variasi Arsitektur Sequence to Sequence

Selain RNN, kita juga dapat menggunakan *bidirectional* RNN (BiRNN) untuk mengikutsertakan pengaruh baik *hidden state* sebelum ($\mathbf{h}_1, \dots, \mathbf{h}_{t-1}$) dan setelah ($\mathbf{h}_{t+1}, \dots, \mathbf{h}_T$) untuk menghitung *hidden state* sekarang (\mathbf{h}_t) [85, 86, 87]. 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 *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.

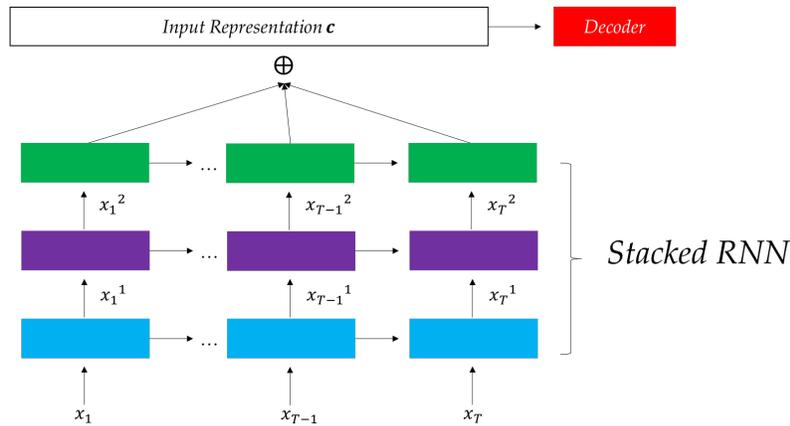
Selain variasi RNN menjadi BiRNN kita dapat menggunakan *stacked RNN* seperti pada Gambar. 13.26 dimana *output* pada RNN pertama bertindak sebagai *input* pada RNN kedua. *Hidden states* yang digunakan untuk menghasilkan representasi *encoding* adalah RNN pada tumpukan paling atas. Kita

¹⁷ Perhatikan! + disini dapat diartikan sebagai penjumlahan atau konkatensasi



Gambar 13.25: *Encoder-decoder* dengan Bidirectional Recurrent Neural Network.

juga dapat menggunakan variasi *attention mechanism* seperti *neural checklist model* [88] atau *graph-based attention* [89]. Selain yang disebutkan, masih banyak variasi lain yang ada, silahkan eksplorasi lebih lanjut sendiri.



Gambar 13.26: *Encoder-decoder* dengan *stacked* Recurrent Neural Network.

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 [90]. Kita ubah *encoder* menjadi sebuah CNN (ingat kembali subbab 13.1) dan *decoder* berupa RNN [90]. 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* $p(x, y) = p(y | x)p(x)$ (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* $p(y | x)$ secara langsung, seperti *classifier* lainnya (e.g., *logistic regression*). Jadi yang dimodelkan antara *generative* dan *discriminative model* adalah dua hal yang berbeda.

Pada subbab ini, penulis memberikan contoh *attention mechanism* yang beroperasi antara *encoder* dan *decoder*. Masih banyak variasi lainnya seperti *self-attention*, *multi-head attention* dan *hierarchical-attention* [91, 92]. Walaupun motivasi dan penggunaan variasi *attention mechanism* berbeda-beda, konsep dasarnya sama yaitu mengekstrak (atau mengambil) informasi dari bagian *network* lainnya.

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. Untuk menyederhanakan pencarian arsitektur, pada umumnya kita dapat menganggap sebuah *neural network* tersusun atas beberapa

¹⁸ <https://deeplearning4j.org/restrictedboltzmannmachine>

¹⁹ <https://deeplearning4j.org/generative-adversarial-network>

“modul”. Pembagian *neural network* menjadi modul adalah hal yang relatif. Untuk mencari tahu konfigurasi arsitektur yang memberikan performa maksimal, kita dapat melakukan *architecture ablation*. Idenya mirip dengan *feature ablation*, dimana kita mencoba mengganti-ganti bagian (modul) neural network. Sebagai contoh, ingat kembali arsitektur *sequence to sequence* dimana kita memiliki *encoder* dan *decoder*. Kita dapat menggunakan RNN, Bidirectional RNN, ataupun Stacked RNN sebagai *encoder*. Hal ini adalah salah satu contoh *architecture ablation*. Akan tetapi, bisa jadi kita mengasumsikan modul yang lebih kecil. Sebagai contoh, menggunakan RNN *encoder* pada *sequence to sequence* dan kita coba mengganti-ganti fungsi aktivasi.

Architecture ablation ini bisa menjadi semakin rumit tergantung persepsi kita tentang definisi modul pada *neural network*, seperti sampai menentukan jumlah *hidden layers* dan berapa jumlah unit pada masing-masing *layer*. Contoh lain adalah memilih fungsi aktivasi yang cocok untuk setiap *hidden layer*. Pada kasus ini “modul” kita adalah sebuah *layer*. Walaupun *neural network* memberikan kita kemudahan dari segi pemilihan fitur, kita memiliki kesulitan dalam menentukan arsitektur. Terlebih lagi, alasan pemilihan banyaknya *units* pada suatu *layer* (e.g., 512 dibanding 256 *units*) mungkin tidak dapat dijustifikasi dengan akurat. Pada *feature ablation*, kita dapat menjustifikasi alasan untuk menghilangkan suatu fitur. Pada *neural network*, kita susah menjelaskan alasan pemilihan arsitektur (dan konfigurasi parameter) karena *search space*-nya jauh lebih besar.

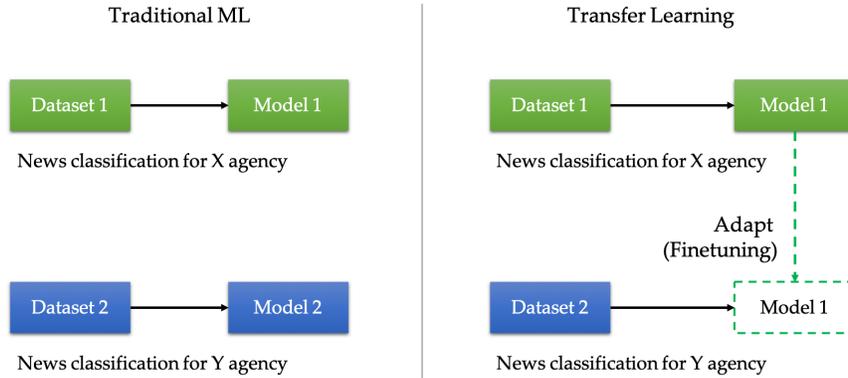
13.7 Transfer Learning

Walau konsep *transfer learning* (TL) tidak terbatas pada *neural network*, subbab ini membahas pemanfaatan TL paling umum pada *neural network*. Pembaca dipersilahkan mengeksplorasi lebih lanjut.

Bayangkan kondisi berikut. Ada dua orang, Haryanto dan Wira. Saat masih kecil, Wira pernah belajar cara memainkan Ukulele, sedangkan Haryanto tidak. Ketika kedua orang tersebut belajar memainkan gitar, menurutmu siapa yang bisa menguasai gitar lebih cepat?

Pada TL, kita ingin menggunakan suatu pengetahuan (*knowledge*) pada suatu *task* T_1 , untuk menyelesaikan permasalahan *task* T_2 [93, 94]. Kita memiliki asumsi bahwa T_1 memiliki kaitan dengan T_2 , sedemikian sehingga fasih pada T_1 akan menyebabkan kita fasih pada T_2 (atau lebih fasih dibandingkan tidak menguasai T_1 sama sekali). Perhatikan Gambar 13.27 yang mengilustrasikan perbedaan pembelajaran mesin biasa dan penggunaan TL. Pada pembelajaran mesin biasa, kita melatih model untuk masing-masing *task*. Pada TL, kita menggunakan model yang sudah ada, disebut *pretrained model*, untuk *task* baru. Selain dimotivasi oleh kemiripan kedua *tasks*, TL juga dimotivasi oleh ketersediaan data. Misal dataset untuk *task* T_1 banyak, sedangkan untuk *task* T_2 sedikit. Berhubung T_1 dan T_2 memiliki kemiripan, model untuk

T_1 yang diadaptasi untuk T_2 akan konvergen lebih cepat dibanding melatih model dari awal untuk T_2 .



Gambar 13.27: Pembelajaran mesin tradisional vs. menggunakan *transfer learning*.



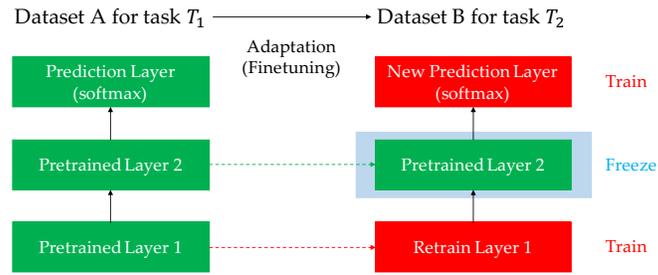
Gambar 13.28: Proses *transfer learning*.

Proses mengadaptasi suatu *pretrained model* disebut **finetuning** (Gambar 13.28). Pertama-tama kita ganti *layer* terakhir (*prediction layer*) pada *pretrained model* menggunakan *layer* baru yang diinisialisasi secara *random*.²⁰ Kemudian, kita latih kembali model yang sudah ada menggunakan data untuk T_2 .

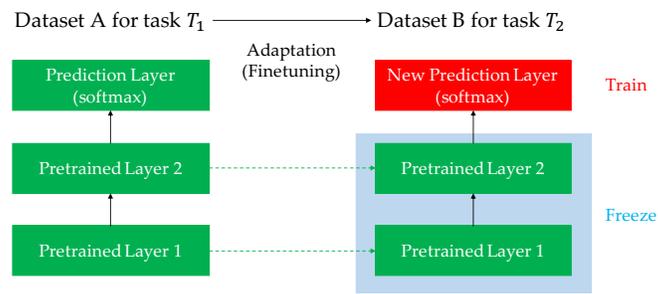
Secara umum, ada tiga cara untuk melakukan *finetuning*.

1. **Freeze some layers.** Kita *freeze* beberapa *layer* (parameternya tidak diperbaharui saat *finetuning*), kemudian latih *layer* lainnya. Ilustrasi diberikan pada Gambar 13.29.
2. **Train only new last layer.** Kita *freeze* semua *layer*, kecuali *layer* terakhir untuk task T_2 . Ilustrasi diberikan pada Gambar 13.30
3. **Train all layers.** Setelah menggantu *layer terakhir*, kita latih semua *layer* untuk task T_2 . Ilustrasi diberikan pada Gambar 13.31.

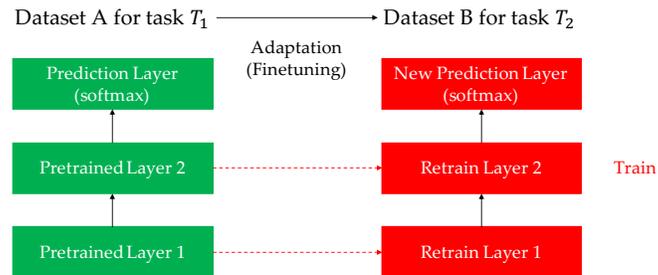
²⁰ Penulis rasa, hal ini hampir wajib hukumnya



Gambar 13.29: *Freeze some layers.*



Gambar 13.30: *Train only new last layer.*



Gambar 13.31: *Train all layers.*

Selain alasan yang sudah disebutkan, TL juga digunakan untuk mempercepat *training*. Konon *pre-trained* model pada umumnya membutuhkan waktu *training* yang lebih cepat (lebih sedikit iterasi) dibanding melatih model baru. Dengan demikian, kita dapat menghemat listrik dan mengurangi polusi CO₂. TL juga berkaitan erat dengan *successive learning* (bab 11), dimana

kita melatih arsitektur lebih kecil kemudian menggunakannya pada arsitektur yang lebih besar.

Demikian konsep paling dasar TL. Selebihkan, penulis menyarankan untuk membaca *paper* atau tutorial terkait, seperti dibawah berikut.

- <https://www.cs.uic.edu/liub/Lifelong-Learning-tutorial-slides.pdf>
- <https://www.aclweb.org/anthology/attachments/N19-5004.Presentation.pdf>

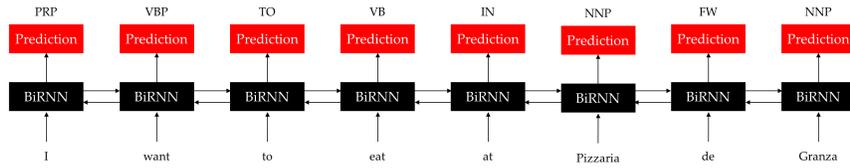
13.8 Multi-task Learning

Subbab ini akan menjelaskan *framework* melatih model pembelajaran mesin menggunakan *multi-task learning* (MTL). Walaupun konsep MTL tidak terbatas pada *neural network*, bab ini membahas konsep tersebut menggunakan arsitektur *neural network* sebagai contoh (karena itu dimasukkan ke dalam bab ini). Kami hanya memberikan penjelasan paling inti MTL menggunakan contoh yang sederhana.

Pada MTL, kita melatih model untuk mengerjakan beberapa hal yang mirip atau berkaitan, secara bersamaan. Misalnya, melatih model POS *tagging* dan *named-entity recognition* [95], mesin penerjemah untuk beberapa pasangan bahasa [96], klasifikasi teks [97] dan *discourse parsing* [98]. Karena model dilatih untuk beberapa permasalahan yang mirip (sejenis), kita berharap agar model mampu mendapatkan “intuisi” dasar yang dapat digunakan untuk menyelesaikan semua permasalahan. Perbedaan TL (dalam konteks pembahasan sebelumnya) dan MTL terletak pada *timing* pelatihan. Apabila pada TL, model untuk *task* T_1 dan T_2 dilatih pada waktu yang berbeda, sedangkan untuk MTL, dilatih bersamaan.

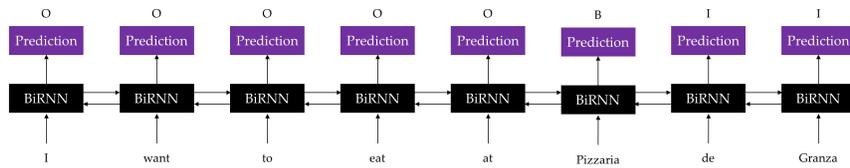
Perhatikan Gambar 13.32 yang merupakan ilustrasi permasalahan POS *tagging*. Diberikan *input* sekuens kata \mathbf{x} , kita ingin mencari sekuens *tag* \mathbf{y} terbaik untuk melambangkan kelas tiap kata. Kami harap kamu masih ingat definisi permasalahan tersebut karena sudah dibahas pada bab-bab sebelumnya. Kita ingin memodelkan *conditional probability* $p(\mathbf{y} | \mathbf{x}, \theta)$. POS *tagging* adalah salah satu *sequence tagging task*, dimana setiap elemen *input* berkorespondensi dengan elemen *output*. Kita dapat melatih model BiRNN ditambah dengan MLP untuk melakukan prediksi kelas kata. Sebelumnya, telah dijelaskan bahwa BiRNN mungkin lebih intuitif untuk POS *tagging* dibanding RNN biasa. Hal ini karena kita dapat memodelkan “konteks” kata (*surrounding words*) dengan lebih baik, yaitu informasi dari kata sebelum dan sesudah (BiRNN), dibanding hanya mendapat informasi dari kata sebelum (RNN).

Sekarang kamu perhatikan Gambar 13.33 yang mengilustrasikan *named entity recognition task* (NER). *Named entity* secara sederhana adalah objek yang bernama, misal lokasi geografis, nama perusahaan, dan nama orang. Pada NER, kita ingin mengekstraksi *named entity* yang ada pada *input*. *Task* ini biasanya direpresentasikan dengan BIO *coding scheme*. Artinya, *output*



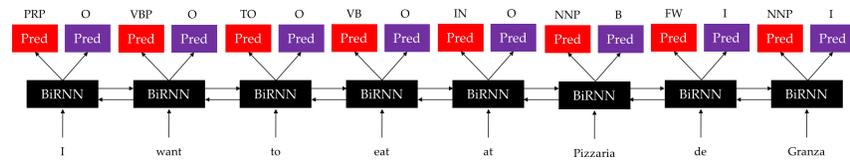
Gambar 13.32: POS *tagger*.

untuk NER adalah pilihan B (*begin*), I (*inside*) dan O (*outside*). Apabila suatu kata adalah kata pertama dari suatu *named entity*, kita mengasosiasikannya dengan *output* B. Apabila suatu kata adalah bagian dari *named entity*, tetapi bukan kata pertama, maka diasosiasikan dengan *output* I. Selain keduanya, diasosiasikan dengan *output* O. Seperti POS *tagging*, NER juga merupakan *sequence tagging* karena kita ingin memodelkan $p(\mathbf{y} \mid \mathbf{x}, \theta)$ untuk \mathbf{x} adalah *input* dan \mathbf{y} adalah *output* (BIO).



Gambar 13.33: *Named Entity Recognition*.

POS *tagging* dan NER dianggap sebagai *task* yang “mirip” karena keduanya memiliki cara penyelesaian masalah yang mirip. Selain dapat diselesaikan dengan cara yang mirip, kedua *task* tersebut memiliki *nature* yang sama. Dengan alasan ini, kita dapat melatih model untuk POS *tagging* dan NER dengan kerangka *multi-task learning*. Akan tetapi, menentukan apakah dua *task* memiliki *nature* yang mirip ibarat sebuah seni (butuh *sense*) dibanding *hard science* [1].

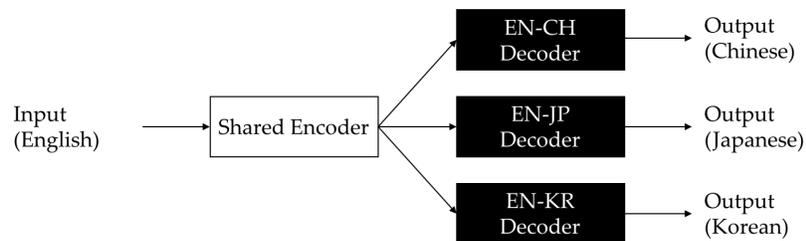


Gambar 13.34: *Multi-task Learning* untuk POS *tagging* dan *Named Entity Recognition*.

Ide utama MTL adalah melatih *shared representation*. Sebagai ilustrasi, perhatikan Gambar 13.34. Sebelumnya, kita melatih dua model dengan BiRNN yang dilewatkan pada MLP. Pada saat ini, kita melatih BiRNN yang dianggap sebagai *shared representation*. BiRNN diharapkan memiliki “intuisi” untuk menyelesaikan kedua permasalahan, berhubung keduanya memiliki *nature* yang sama. Setiap *hidden layer* pada BiRNN dilewatkan pada MLP untuk melakukan prediksi pada masing-masing *task*. Tujuan utama MTL adalah untuk meningkatkan kinerja. Kita melatih model untuk *task X* dengan meminjam “intuisi” penyelesaian dari *task Y* dengan harapan “intuisi” yang dibawa dari *task Y* dapat memberikan informasi tambahan untuk penyelesaian *task X*.



(a) Sequence to Sequence Machine Translation



(b) Machine Translation in Multi-task Setting

Gambar 13.35: *Multi-task Learning* pada mesin translasi.

Perhatikan contoh berikutnya tentang MTL pada mesin translasi (Gambar 13.35). Pada permasalahan mesin translasi, kita melatih model menggunakan data paralel kombinasi pasangan bahasa X - Y .

Penggunaan MTL pada mesin mesin translasi pada umumnya dimotivasi oleh dua alasan.

- Pada kombinasi pasangan bahasa tertentu, tersedia *dataset* dengan jumlah yang banyak. Tetapi, bisa jadi kita hanya memiliki *dataset* berukuran kecil untuk bahasa tertentu. Sebagai contoh, data mesin translasi untuk pasangan English-France lebih besar dibanding English-Indonesia. Karena kedua kombinasi pasangan bahasa memiliki *nature* yang cukup sama, kita dapat menggunakan MTL sebagai kompensasi data English-Indonesia

yang sedikit agar model pembelajaran bisa konvergen. Dalam artian, *encoder* yang dilatih menggunakan sedikit data kemungkinan memiliki performa yang kurang baik. Dengan ini, kita latih suatu *encoder* menggunakan data English-France dan English-Indonesia agar model bisa konvergen. Pada kasus ini, transfer learning juga dapat digunakan. Kita melatih model English-France, kemudian memasang *encoder* yang sudah dilatih dengan *decoder* baru untuk bahasa Indonesia.

- Seperti yang sudah dijelaskan sebelumnya, kita ingin menggunakan “intuisi” penyelesaian suatu permasalahan untuk permasalahan lainnya, berhubung solusinya keduanya mirip. Dengan hal ini, kita harap kita mampu meningkatkan kinerja model. Sebagai contoh, kombinasi pasangan bahasa English-Japanese dan English-Korean, berhubung kedua bahasa target memiliki struktur yang mirip.

Pada kerangka MTL, *utility function* atau objektif *training* adalah meminimalkan *joint loss* semua *tasks* (hal ini juga membedakan TL dan MTL), diberikan pada persamaan 13.19. Kita dapat mendefinisikan *loss* pada kerangka MTL sebagai penjumlahan *loss* pada masing-masing *task*, seperti pada persamaan 13.20. Apabila kita menganggap suatu *task* lebih penting dari *task* lainnya, kita dapat menggunakan *weighted sum*, seperti pada persamaan 13.21. Kita juga dapat menggunakan *dynamic weighting* untuk memperhitungkan *uncertainty* pada tiap *task*, seperti pada persamaan 13.22 [99], dimana σ melambangkan varians *task-specific loss*.

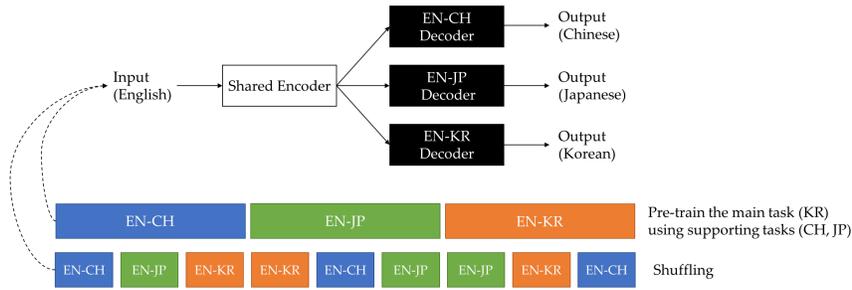
$$\mathcal{L}_{\text{MTL}} = q(\mathcal{L}_{T_1}, \dots, \mathcal{L}_{T_D}) \quad (13.19)$$

$$q(\mathcal{L}_{T_1}, \dots, \mathcal{L}_{T_D}) = \sum_i^D \mathcal{L}_{T_i} \quad (13.20)$$

$$q(\mathcal{L}_{T_1}, \dots, \mathcal{L}_{T_D}) = \sum_i^D \alpha_i \mathcal{L}_{T_i} \quad (13.21)$$

$$q(\mathcal{L}_{T_1}, \dots, \mathcal{L}_{T_D}) = \sum_i^D \frac{1}{2\sigma_i^2} \mathcal{L}_{T_i} + \ln(\sigma_i) \quad (13.22)$$

Saat melatih MTL, tujuan *training* dapat mempengaruhi proses penyajian data. Seumpama saat melatih mesin translasi untuk English-{Chinese, Japanese, Korean}, kita ingin menganggap English-Korean sebagai *main task* sementara sisanya sebagai *supporting task*, kita dapat melakukan *pre-training* menggunakan data English-Chinese dan English-Japanese terlebih dahulu, diikuti oleh English-Korean (tetapi *loss* tetap *joint loss*). Pada kasus ini, penggunaan *joint weighted loss* dapat dijustifikasi. Di lain pihak, apabila kita menganggap semua *tasks* penting, kita dapat melakukan data *shuffling* sehingga urutan *training data* tidak bias pada *task* tertentu. Pada kasus ini,



Gambar 13.36: *Multi-task Learning setup.*

penggunaan *joint loss-sum* dapat dijustifikasi. Ilustrasi diberikan pada Gambar 13.36.

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) *Boltzman Machine*
- (b) *Restricted Boltzman Machine*
- (c) *Generative Adversarial Network*

