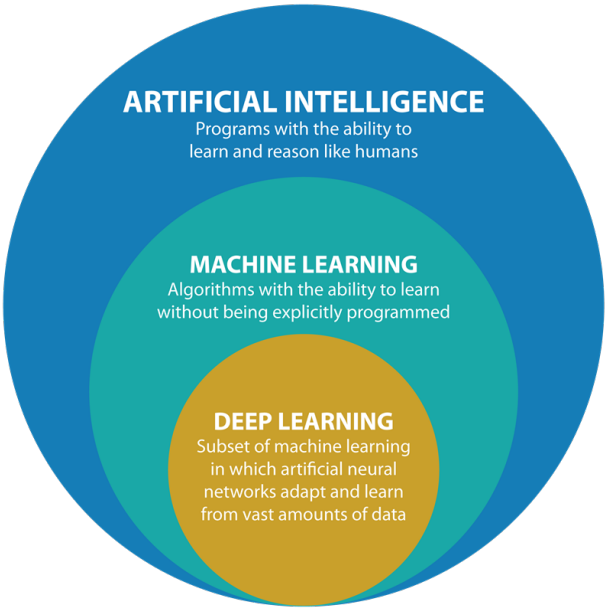
Chapter 9:
Deep Neural Networks (DNN)

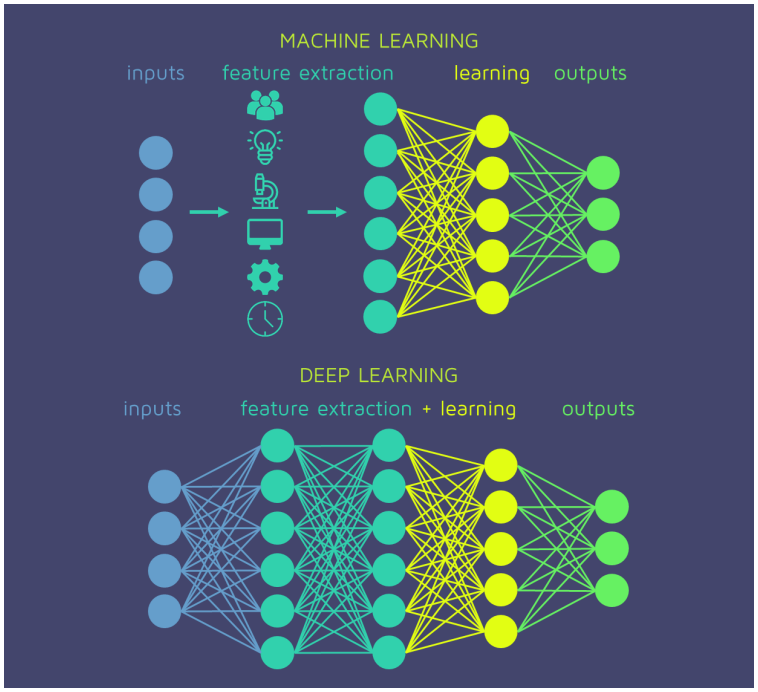
December 13, 2020

Deep Learning atau ***Deep Neural Networks*** (DNN) merupakan *sub-class* dari algoritma machine learning yang dapat memecahkan masalah-masalah dengan kompleksitas tingkat tinggi. Sehingga *deep learning* merupakan bagian dari *Machine Learning* dan keduanya tidak berseberangan dalam konsep. Semua teknik *machine learning* dikatakan *shallow*, kecuali yang menggunakan konsep *deep* (berlapis-lapis). Kedudukan *artificial intelligence*, *machine learning*, dan *deep learning* dapat diilustrasikan pada Gambar 9.1.



Gambar 9.1: Deep learning merupakan bagian dari machine learning, dan machine learning merupakan bagian dari artificial intelligence.

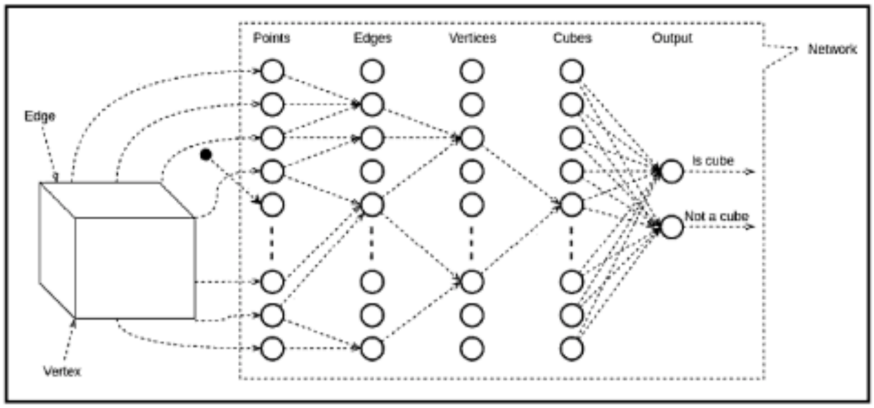
Dalam *deep learning* informasi mengalir melewati banyak layer-layer (lapisan-lapisan). Seperti manusia, informasi dipelajari secara bertahap. Layer pertama fokus untuk mempelajari (*learning*) konsep-konsep yang spesifik, sementara layer-layer dalam akan menggunakan informasi yang telah dipelajari tadi untuk menyerap konsep yang lebih abstrak. Arsitektur yang kompleks membuat DNN mempunyai kemampuan untuk melakukan ekstraksi ciri (*feature extraction*) secara otomatis. Sebaliknya, proses ekstraksi ciri pada teknik *machine learning* konvensional (*shallow learning*) dilakukan di luar tahap algoritma. Kita atau tim *data scientist* (bukan *machine*) yang mempunyai tugas untuk melakukan ekstraksi ciri dengan cara melakukan analisa data mentah dan merubahnya menjadi *feature-feature* yang berharga. Hal ini diilustrasikan pada Gambar 9.2.



Gambar 9.2: Perbedaan Machine Learning dan Deep Learning (source: https://quantdare.com/what-is-the-difference-between-deep-learning-and-machine-learning/)

1 *Feature Learning*

Untuk mengilustrasikan lebih jauh bagaimana *deep learning* (DL) bekerja, kita akan lihat cara untuk mengenali sebuah gambar geometris sederhana misalkan kubus, seperti yang terlihat di diagram pada Gambar 9.3.



Gambar 9.3: Abstraksi DNN untuk merepresentasikan kubus. Layer yang berbeda akan mengkodekan feature dengan tingkatan abstraksi yang berbeda

Kubus terdiri dari *edges* (garis) yang berpotongan membentuk *vertex* (sudut). Dimisalkan bahwa setiap titik pada ruang tiga dimensi adalah neuron (lupakan sementara bahwa hal ini membutuhkan jumlah neuron yang tak terbatas). Semua titik/neuron merupakan input pada layer pertama dari sebuah *feed-forward network* berlayer banyak (*multilayer*). Sebuah titik/neruon aktif bila titik tersebut terletak pada sebuah garis. Titik-titik/neuron-neuron yang terletak pada *edge* yang

sama mempunyai hubungan positif yang kuat pada sebuah *edge* tunggal yang sama pada layer selanjutnya. Sebaliknya, mereka mempunyai koneksi negatif dengan neuron-neuron lain pada layer selanjutnya, kecuali pada neuron di *vertices* (bentuk jamak dari *vertex*). Masing-masing neuron seperti itu terhubung dengan tiga neuron lain pada layer selanjutnya.

Sekarang kita mempuyai dua *hidden layer* dengan tingkatan abstraksi yang berbeda, pertama untuk titik dan kedua untuk *edge*. Tetapi hal tersebut tidak cukup untuk mengkodekan kubus pada sebuah network. Jika kita perhatikan, setiap *edge*/neuron aktif dari layer kedua yang membentuk *vertex*, mempunyai hubungan positif yang signifikan dengan *vertex* tunggal yang sama pada layer ketiga. Karena setiap *edge* membentuk dua *vertices*, masing-masing *edge* neuron akan mempunyai koneksi positif dengan dua *vertices*/neuron dan koneksi negatif dengan neuron lainnya. Pada ujung, *hidden layer* terakhir berupa kubus (*cube*). Empat *vertices*/neuron membentuk kubus akan mempunyai koneksi dengan sebuah kubus/neuron yang sama pada *hidden layer* terakhir (layer kubus).

Representasi kubus diatas terlalu disederhanakan tetapi kita dapat menarik beberapa kesimpulan. Salah satunya adalah DNN baik untuk membentuk dirinya dalam mengorganisasikan data secara berjenjang (*hierarchically organized data*). Contoh, sebuah image terdiri dari piksel-piksel yang membentuk garis, *edge*, daerah (*region*) dan selanjutnya. Hal ini juga berlaku untuk *speech* dimana blok pembangun berupa *phonem-phonem*, dan juga text dimana kita punya karakter, kata dan kalimat.

Pada contoh sebelumnya, kita secara sengaja memperuntukan layer-layer pada *feature-feature* kubus. Secara praktis kita tidak akan melakukannya sendiri, tetapi DNN akan menemukan *feature-feature* tersebut secara otomatis pada proses training. *Feature-feature* ini boleh jadi tidak jelas dan secara umum tidak bisa diinterpretasikan oleh manusia. Selain itu kita juga tidak akan mengetahui tingkatan *feature* yang dikodekan pada layer-layer yang berbeda pada network tersebut. Berbeda dengan teknik *machine learning* konvensional dimana seorang pengguna harus memanfaatkan pengalamannya dalam memilih *feature* terbaik, atau lebih dikenal dengan ***feature engineering*** yang cukup memakan waktu dan tenaga untuk melakukannya. Membuat network dapat menemukan *feature* secara otomatis bukan hanya mempermudah tetapi membuat *feature-feature* lebih abstrak dan tidak rentan terhadap noise. Misalkan, penglihatan manusia dapat mengenali objek-objek dengan keragaman bentuk, ukuran, kondisi pencahayaan yang berbeda, dan bahkan ketika sebagian terhalang. Kita dapat mengenali manusia dengan potongan rambut berbeda, *feature-feature* wajah, dan bahkan ketika menggunakan topi atau masker. Hal yang serupa, *feature-feature* abstrak yang dipelajari network akan menolong mengenali wajah dengan baik meskipun pada kondisi-kondisi yang menantang.

2 Algoritma-Algoritma DNN

Kita dapat mendefinisikan *deep learning* sebagai bagian dari teknik-teknik *machine learning* dimana informasi diproses pada layer-layer yang berjenjang untuk memahami representasi-representasi dan *feature-feature* dari data dengan level kompleksitas yang semakin tinggi. Secara praktis, semua algoritma-algoritam *deep learning* adalah berbasis *neural network* dengan sifat-sifat dasar yang sama. DNN terdiri dari neuron-neuron yang saling terhubung dan disusun dalam bentuk layer-layer. Perbedaan algoritma DNN satu dengan yang lain biasanya terletak pada arsitektur network-nya dan bisa jadi juga berbeda bagaimana algoritma tersebut dilatih (*training*). Beberapa algoritma yang populer diantaranya adalah sebagai berikut:

- ***Mulitlayer Perceptrons* (MLPs)**. MLP merupakan *neural network* dengan propagasi *feed-forward*, *fully-connected* layer, dan sedikitnya terdapat satu *hidden layer*. Kita sudah mempelajari MLP di *Chapter* 8.
- ***Convolutional Neural Networks* (CNNs)**. CNN adalah *feed-forward neural network* dengan beberapa tipe-tipe pada layer-layer khusus. Sebagai contoh, layer-layer konvolusi (*convolutional layer*) akan mengimplementasikan sebuah filter pada input berupa image atau suara dengan cara menggeser filter tersebut pada keseluruhan sinyal yang datan, untuk menghasilkan peta aktivasi berdimensi n. Terdapat bukti-bukti bahwa neuron-neuron pda CNNs diatur sebagaimana sel-sel biologi diatur pada *visual cortex* dari otak. Saat sekarang ini CNNs mengalahkan kinerja semua algoritma *machine learning* pada banyak pekerjaan di bidang *computer vision* dan NLP.
- ***Recurrent Neural Networks* (RNNs)**. Tipe network seperti ini mempunyai *state* internal (*internal state*) atau memori yang berdasarkan pada semua atau sebagian dari data input yang telah diberikan pada network. Output dari RNN merupakan kombinasi dari state internal (memori dari input) dan sampel input terakhir. Pada saat yang sama, state internal berubah untuk mengakomodasi input data terbaru. Dengan sifat-sifat seperti ini, RNN merupakan kandidat yang baik untuk pekerjaan-pekerjaan yang berhubungan dengan data sekuensial, seperti text atau data *time series*.
- ***Autoencoders***. Merupakan *class* algoritma *unsupervised learning* dimana bentuk output sama dengan input, sehingga memungkinkan network untuk mempelajari representasi dasar secara lebih baik. Autoencoder merupakan salah satu contoh penggunaan *neural network* dengan cara *generative* selain ***Generative Adversarial Networks*** (GANs).

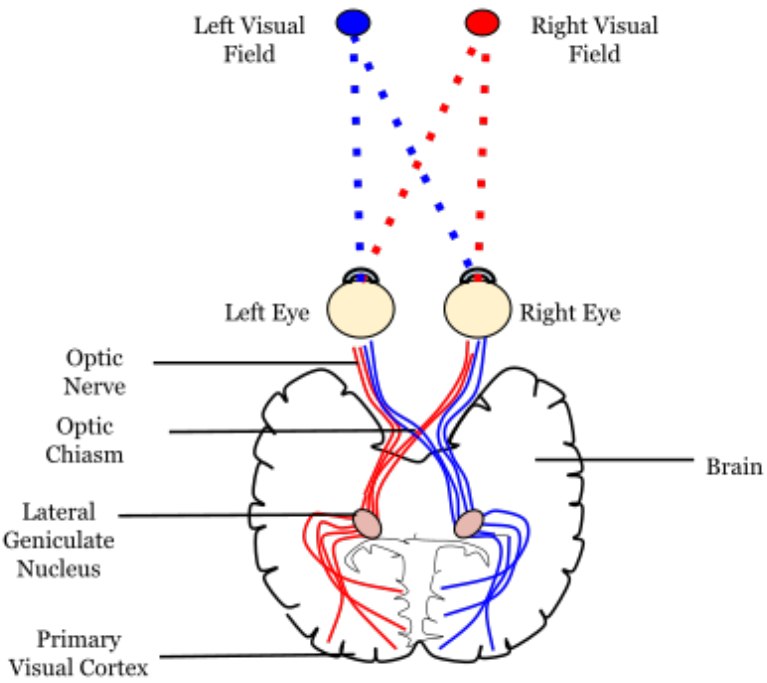
Pada bagian selanjutnya hanya akan dijelaskan CNN dan RNN secara singkat.

3 *Convolutional Neural Networks* (CNNs)

CNN berkembang dari studi *visual cortex* otak (lihat Gambar 9.4) dan telah digunakan untuk *image recognition* sejak tahun 80-an. Beberapa tahun terakhir, CNN dapat mencapai kemampuan manusia super untuk pekerjaan-pekerjaan yang berkaitan dengan visual yang kompleks, karena didukung oleh meningkatnya kemampuan komputasi, jumlah data yang semakin banyak dan metode untuk training yang semakin baik. CNNs telah meningkatkan kemampuan untuk aplikasi-aplikasi servis pencarian image-image, *self-driving car*, sistem klasifikasi video otomatis, dan banyak lagi. Lebih jauh, CNN saat ini tidak hanya terbatas pada aplikasi persepsi visual, tapi juga berhasil pada pekerjaan-pekerjaan lain, seperti *video recognition* dan *natural language processing*.

3.1 Arsitektur *Visual Cortext*

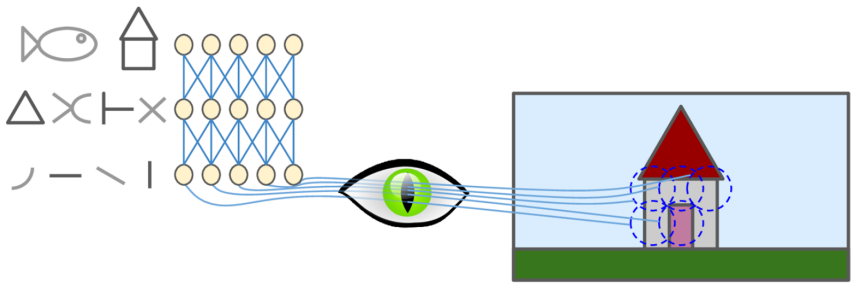
David H. Hubel dan Torsten Wiesel yang mendapatkan hadiah Nobel pada tahun 1981 dalam bidang *Physiology/Medicine*, dapat menunjukan secara khusus bahwa banyak neuron-neuron pada *visual cortex* mempunyai bidang reseptif lokal (*local receptive field*) yang kecil. Artinya neuron-neuron pada *visual cortex* hanya bereaksi pada rangsangan visual yang berlokasi pada daerah terbatas dari bidang visual (lihat Gambar 9.4), dimana bidang reseptif lokal dari lima neuron-nueron yang tergambar direpresentasikan dengan lingkaran putus-putus. Bidang reseptif dari



Gambar 9.4: Visual cortex terletak pada bagian belakang otak yang berfungsi untuk mengolah informasi visual.

neuron-neuron yang berbeda bisa jadi tumpang tindih, dan bersama-sama membentuk kesuluruhan bidan visual.

Lebih jauh, David dan Torsten menunjukan bahwa beberapa neuron bereaksi hanya pada image-image dari garis-garis horizontal, sementara yang lain bereaksi hanya pada garis-garis dengan orientasi yang berbeda (Dua neuron boleh jadi mempunyai bidang reseptif yang sama tetapi bereaksi pada orientasi garis yang berbeda). Keduanya juga memperhatikan bahwa beberapa neuron mempunyai bidang reseptif yang lebih besar, dan bereaksi pada pola-pola yang lebih kompleks yang merupakan kombinasi dari pola-pola tingkat rendah. Observasi ini membawa ide bahwa neuron pada tingkatan lebih tinggi adalah berbasis ouput-output neuron-neuron tetangga yang mempunyai tingkatan rendah. Pada Gambar 9.3 diperlihatkan bahwa setiap neuron dikoneksikan hanya dengan beberapa neuron dari layer sebelumnya. Arsitektur yang *powerful* ini mampu mendeteksi banyak macam pola-pola kompleks di area sembarang pada bidang visual.



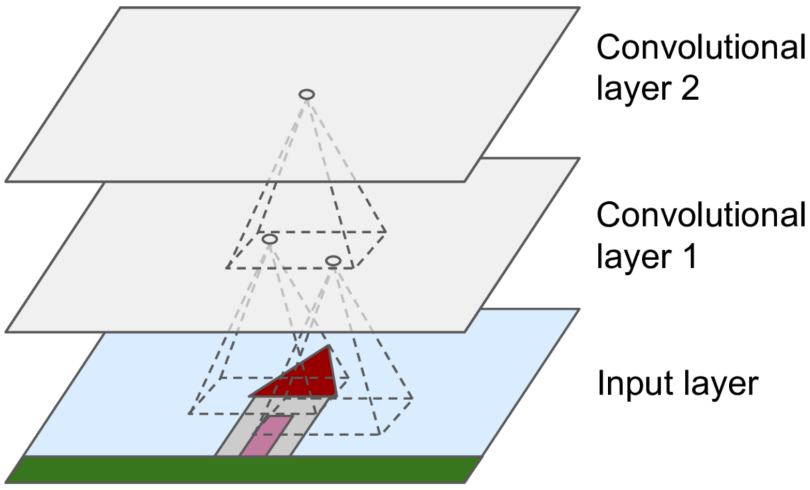
Gambar 9.5: Neuron-neuron biologis pada visual cortex hanya merespon pada pola-pola khusus pada bagian kecil di bidang visual yang disebut bidang reseptif (receptive field)

Studi tentang *visual cortex* ini menginspirasi ***neocognitron*** yang diperkenalkan pada tahun 1980 dan kemudian berevolusi secara gradual yang sekarang dikenal dengan ***convolutional neural networks*** (CNN). Kejadian penting lainnya adalah paper tahun 1998 dari Yann LeCun, yang memperkenalkan arsitektur terkenal *Le_Net*-5 yang banyak digunakan oleh Bank-Bank untuk nomor cek yang ditulis tangan. Arsitektur ini mempunyai blok-blok pembangun yang sudah kita ketahui, seperti *fully-connected layer* dan fungsi aktivasi sigmoid. Tetapi juga memperkenalkan dua blok baru yaitu *convolutional layer* dan *pooling layer*.

Catatan. Kenapa kita tidak menggunakan *deep neural network* dengan *fully connected layers* untuk pekerjaan *image recognition*? Hal itu karena, meskipun penggunaan *fully connected layers* dapat bekerja dengan baik untuk image-image berukuran kecil (mis. MNIST), tetapi gagal ketika harus menggunakan image-image dengan ukuran lebih besar karena membutuhkan parameter-parameter yang sangat besar. Misalkan, untuk image dengan ukuran 100×100 mempunyai total piksel berjumlah 10000 piksel. Dan jika layer pertama berisi hanya 1000 neuron (dimana sudah betul-betul membatasi jumlah informasi yang dapat ditransmisikan pada layer selanjutnya), artinya total 100 juta koneksi. Dan itu hanya layer pertama saja. CNN mengatasi masalah ini dengan menggunakan koneksi layer-layer secara parsial dan men-*sharing* bobot (*weight sharing*).

3.2 Layer-Layer Konvolusi (*Convolutional Layers*)

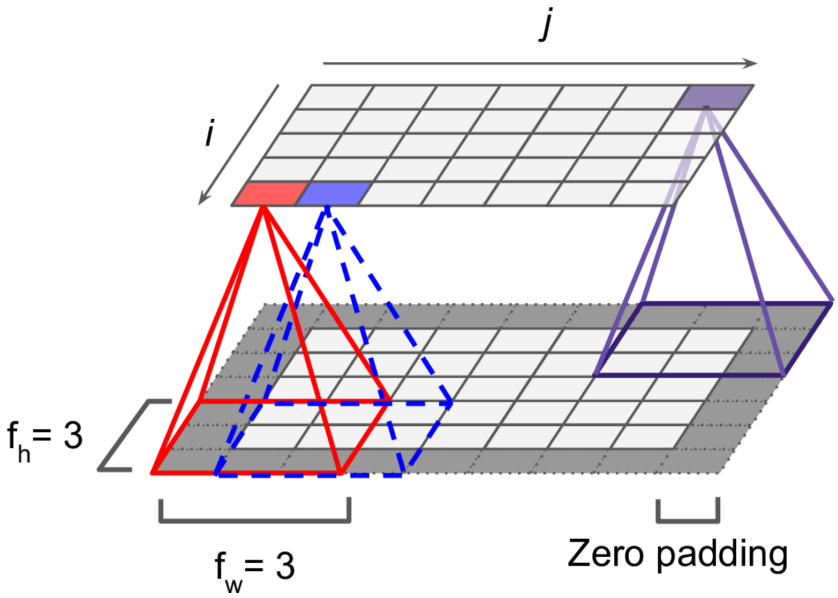
Blok bangunan (*building block*) yang paling penting pada CNN adalah ***convolutional layer*** (CL). Neuron-neuron pada CL pertama tidak dikoneksikan pada setiap piksel-piksel dari image input, tetapi hanya piksel-piksel pada bidang repseptif mereka (lihat Gambar 9.6). Kemudian, setiap neuron pada CL kedua dihubungkan dengan neuron-neuron yang berlokasi di dalam sebuah *rectangle* pada layer pertama. Arsitektur ini membuat network terkonsentrasi pada *feature-feature* level rendah yang kecil di *hidden layer* pertama, kemudian mereka terhimpun ke dalam *feature-feature* level tinggi pada *hidden layer* selanjutnya, dan begitu seterusnya. Arsitektur yang berjenjang seperti ini sebetulnya biasa pada image-image dunia nyata, yang merupakan alasan juga mengapa CNN dapat bekerja dengan baik untuk *image recognition*.



Gambar 9.6: Layer-layer CNN dengan bidang reseptif lokal berbentuk *rectangular*.

Catatan. Pada CNN setiap layer direpresentasikan dalam 2D, sehingga membuat lebih mudah mencocokan neuron-neuron dengan input-input yang berhubungan.

Neuron yang terletak pada baris ke-i dan kolom ke-j pada layer tertentu dihubungkan dengan output-output dari layer sebelumnya yang berlokasi pada baris ke-i s/d $i + f_h - 1$, kolom ke-j s/d $j + f_w - 1$, dimana f_h dan f_w adalah berturut-turut tinggi dan lebar bidang reseptif (lihat Gambar 9.7). Untuk membuat sebuah layer memiliki tinggi dan lebar yang sama dengan layer sebelumnya, maka biasanya ditambahkan nol-nol sekitaran input yang disebut dengan *zero padding*, seperti yang ditujukan pada Gambar 9.7.

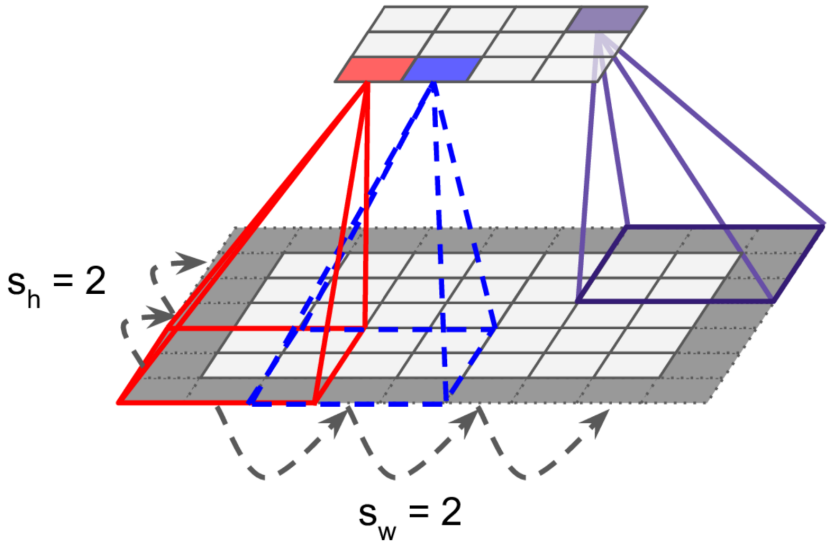


Gambar 9.7: Koneksi antara layer-layer dan zero padding

Adalah memungkinkan juga mengkoneksikan layer input yang besar ke layer yang jauh lebih kecil dengan memberikan ruang yang lebih besar pada bidang reseptif, seperti yang ditunjukan pada Gambar 9.8. Hal ini akan mengurangi kompleksitas komputasi secara dramatis. Pergeseran dari satu bidang reseptif ke selanjutnya disebut degan *stride*. Pada gambar terlihat bahwa layer input 5×7 (plus *zero padding*) dikoneksikan dengan layer 3×4 menggunakan bidang reseptif 3×3 dan *stride* dua (pada contoh ini *stride* sama pada kedua arah, tetapi tidak harus seperti itu). Sebuah neuron yang berlokasi pada baris i dan kolom j pada layer atas dikoneksikan dengan output dari layer-layer sebelumnya yang berlokasi pada baris ke-$(i \times s_h)$ s/d ke-$(i \times s_h + f_h - 1)$, kolom ke-$(j \times s_w)$ s/d ke-$(j \times s_w + f_w - 1)$, dimana s_h dan s_w adalah berturut-turut *stride* vertikal dan horisontal.

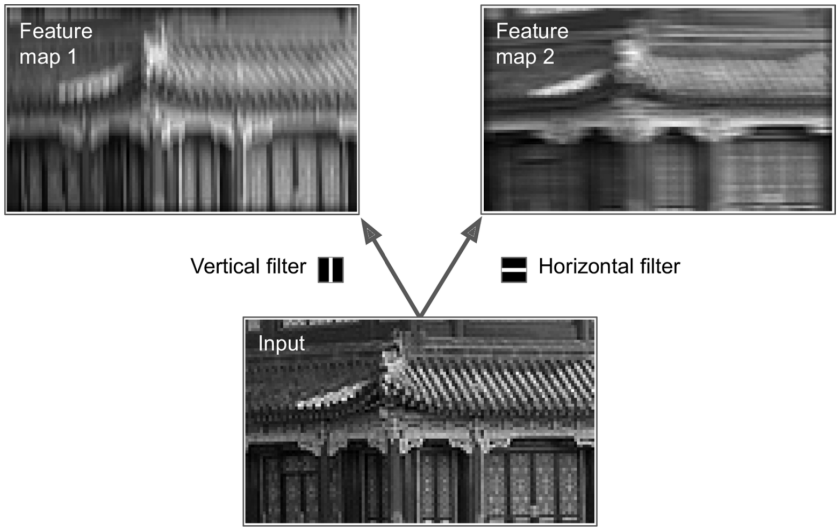
3.2.1 *Filter*

Bobot-bobot sebuah neuron dapat direpresentasikan sebagai sebuah image kecil seukuran dengan bidang reseptif. Sebagai contoh, Gambar 9.9 menampilkan dua set bobot yang mungkin, yang disebut dengan ***filter*** (atau *convolutional kernel*). Filter yang pertama direpresentasikan sebagai persegi hitam dengan garis putih vertikal di tengah (matriks 7×7 semuanya bernilai nol kecuali pada kolom tengah, yang semuanya berharga 1). Neuron dengan menggunakan pembobot-pembobot ini akan mengabaikan semua yang berada pada bidang reseptif kecuali pada garis vertikal ditengah (hal ini terjadi karena semua input dikalikan dengan nol, kecuali yang berlokasi pada garis vertikal ditengah).Filter kedua adalah persegi hitam dengan garis horisontal putih di tengah. Sekali lagi, neuron-neuron yang menggunakan bobot-bobot ini akan mengabaikan



Gambar 9.8: Mengurangi dimensi menggunakan stride 2.

apapun pada bidang reseptif kecuali untuk garis putih di tengah.



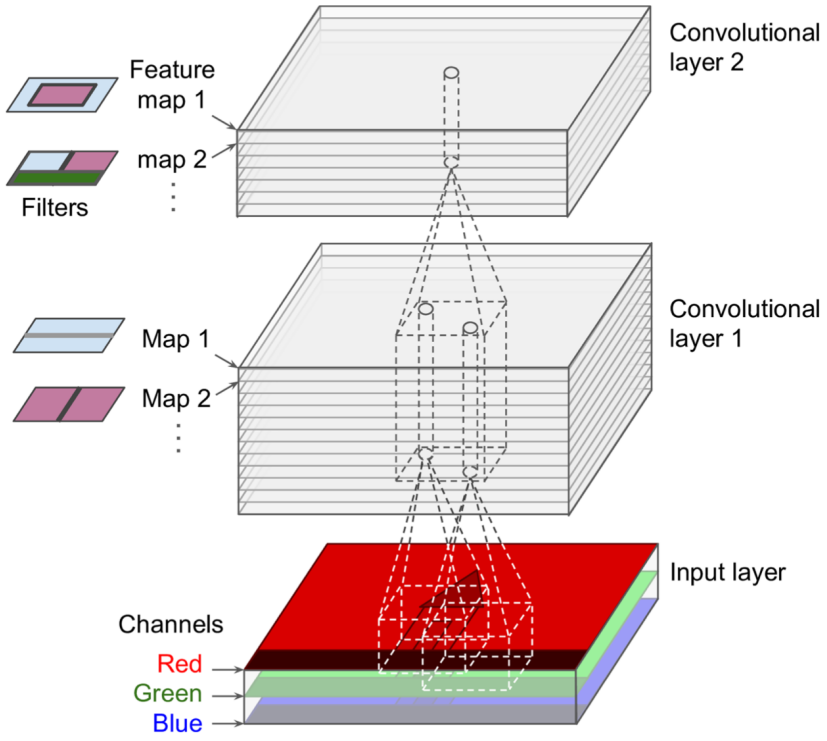
Gambar 9.9: Menggunakan dua filter yang berbeda untuk mendapatkan dua feature map

Sekarang jika semua neuron-neuron pada sebuah layer menggunakan filter garis vertikal yang sama (dan bagian bias yang sama), kemudian network diberikan input image yang ditunjukan pada Gambar 9.9 (gambar bawah), layer akan mengeluarkan image sebelah kiri atas. Terlihat bahwa garis putih vertikal mendapatkan peningkatan sedangkan komponen lain menjadi kabur. Hal yang serupa terjadi pada image sebelah kanan atas dimana semua neuron menggunakan filter garis horisontal, sehingga garis putih horisontal akan mendapatkan peningkatan sedangkan selebihnya menjadi kabur. Oleh karena itu, sebuah layer yang penuh neuron menggunakan filter yang sama akan mengeluarkan sebuah *feature map* yang menekankan area-area image yang paling mengaktivasi filter. Tentu saja, kita tidak perlu mendefinisikan filter-filter secara manual, sebaliknya selama training *convolutional layer* akan secara otomatis mempelajari filter-filter yang paling berguna untuk tugas tertentu, dan layer-layer di atasnya akan belajar mengkombinasikan-

nya membentuk pola yang lebih kompleks.

3.2.2 Menumpukan Beberapa *Feature Maps*

Sampai bagian ini, sebagai penyederhanaan kita sudah merepresentasikan output dari masing-masing *convolutional layer* sebagai layer 2D. Tetapi kenyataannya *convolutional layer* mempunyai banyak filter dan mengeluarkan satu *feature map* per filter, sehingga akan lebih akurat direpresentasikan dalam 3D (lihat Gambar 9.10). Terdapat satu neuron per piksel pada setiap *feature map*, dan semua neuron-neuron di dalam *feature map* berbagi parameter yang sama (yaitu, bobot dan bagian bias yang sama). Neuron pada *feature map* yang berbeda menggunakan parameter yang berbeda. Bidang reseptif sebuah neuron adalah sama seperti yang telah dideskripsikan sebelumnya, tetapi diperluas keseluruh *feature map-feature map* dari semua layer sebelumnya. Singkatnya, CL secara simultan mengaplikasikan filter-filter yang dapat dilatih terhadap inputnya, membuatnya mampu untuk mendeteksi beberapa *feature* dimanapun pada inputnya.



Gambar 9.10: Convolutional layer dengan multiple feature maps, dan image dengan 3 kanal warna

Catatan. Kenyataan bahwa semua neuron-neuron pada sebuah *feature map* saling berbagi parameter-parameter yang sama telah mengurangi jumlah parameter dalam model secara drastis. Sekali CNN telah belajar untuk mengenali pola pada suatu lokasi, maka dapat mengenalinya pada lokasi lain sembarang. Sebaliknya, jika DNN biasa telah belajar untuk mengenali sebuah pola pada satu lokasi, maka hanya akan mengenalinya pada lokasi khusus tersebut.

Image-image input juga terdiri dari beberapa sublayer-sublayer, satu untuk setiap kanal warna. Biasanya terdapat tiga kanal warna yaitu, *red*, *green*, *blue* (RGB). Image-image *grayscale* hanya mempunyai satu kanal warna saja, tetapi image-image bisa jadi memiliki jumlah kanal warna

lebih banyak lagi, seperti image-image satelit yang menangkap frekuensi-frekuensi cahaya ekstra (seperti *infrared*).

Khususnya, sebuah neuron berlokasi di baris ke-i dan kolom ke-j dari sebuah *feature map* ke-k pada CL ke-l dihubungkan dengan output-output neuron pada layer sebelumnya $l-1$, yang berlokasi pada baris-baris ke-$(i \times s_h)$ s/d ke-$(i \times s_h + f_h - 1$ dan kolom-kolom ke $(j \times s_w + f_w - 1)$ melalui keseluruhan *feature map* (pada layer $l-1$). Sebagai catatan semua neuron-neuron yang berada pada baris ke-i dan kolom ke-j tetapi berada pada *feature map-feature map* yang berbeda dikoneksikan pada output-output neuron-neuron yang sama dari layer sebelumnya.

Persamaan (9.1) meringkas penjelasan sebelumnya dengan menggunakan sebuah persamaan matematika. Persamaan ini menunjukan bagaimana menghitung output dari neuron tertentu pada CL. Persamaan terlihat cukup kompleks karena menggunakan banyak indeks, tetapi sebetulnya hanya menghitung penjumlahan input-input yang diboboti, ditambah dengan bagian bias.

Persamaan (9.1). Menghitung ouput dari sebuah neuron pada *convolutional layer*

$$z_{i,j,k} = b_k + \sum_{u=0}^{f_h-1} \sum_{v=0}^{f_w-1} \sum_{k'=0}^{f_{n'}-1} x_{i',j',k'} . w_{u,v,k',k}$$

dimana $i^{'} = i \times s_h + u$ dan $j^{'} = j \times s_w + v$.

Pada persamaan di atas:

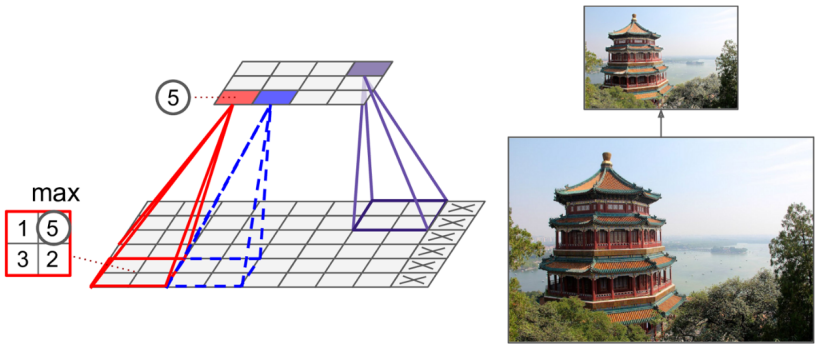
- $z_{i,j,k}$ adalah output dari neuron yang pada baris ke-i dan kolom ke-j di *feature map* ke-k dari CL ke-l.
- s_h dan s_w adalah *stride* horizontal dan vertikal, sedangkan f_h dan f_w adalah tinggi dan lebar dari bidang resiptif, kemudian $f_{n'}$ menyatakan jumlah *feature map* pada layer sebelumnya (layer $l-1$).
- $x_{i',j',k'}$ adalah output dari neuron yang berada di layer ke $l-1$, baris ke-$i^{'}$, kolom ke-$j^{'}$, *feature map* ke-$k^{'}$ (atau kanal ke-$k^{'}$ jika layer sebelumnya adalah layer input)
- b_k adalah bagian bias (*bias term*) untuk *feature map* ke-k (pada layer ke-l). Kita dapat memikirkannya sebagai tombol yang dapat mengatur tingkat kecerahan dari *feature map* ke-k.
- $w_{u,v,k',k}$ adalah bobot koneksi antaran sembaran neuron pada *feature map* ke-k pada layer ke-l dan inputnya berada pada lokasi baris ke-u, kolom ke-v (relatif terhadap bidang reseptif dari neuron), dan *feature map* ke-$k^{'}$.

3.3 Layer-Layer *Pooling* (*Pooling Layers*)

Setelah kita memahami bagaimana CL bekerja, *pooling layer* (PL) cukup mudah untuk dikuasai. Tujuan utamanya adalah melakukan *subsampling* atau menyusutkan image input untuk mengurangi beban komputasi, penggunaan memori, dan jumlah parameter (menghindari resiko *overfitting*).

Seperti pada CL, setiap neuron pada PL dikoneksikan dengan output-output sejumlah neuron-neuron dari layer sebelumnya, berlokasi dalam sebuah bidang resiptif *rectangular* yang kecil. Kita harus mendefinisikan ukurannya, *stride*, dan tipe *padding*, seperti sebelumnya. Tetapi, PL tidak mempunyai bobot, apa yang dilakukan hanya mengagregasi input-input menggunakan fungsi

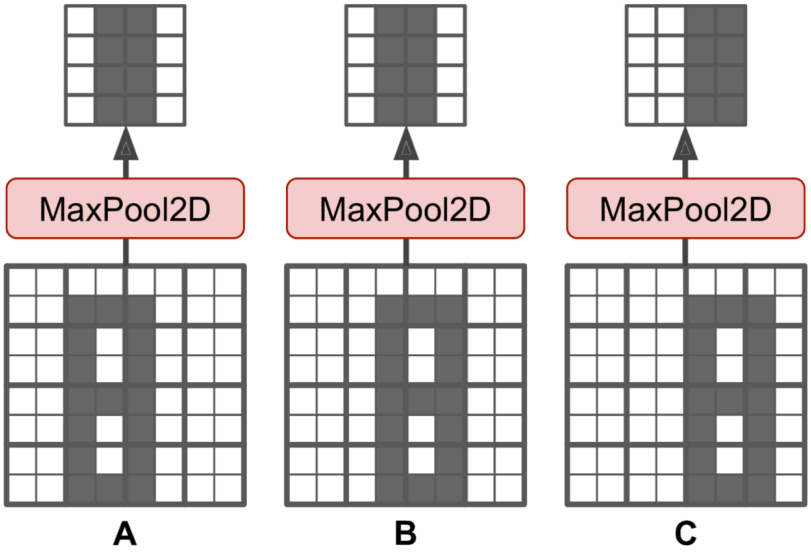
agregasi seperti `max` ataum `min`. Gambar 9.11 menunjukan sebuah *max pooling layer* yang merupakan tipe paling umum dari PL. Pada contoh ini, kita menggunakan 2×2 *pooling kernel* dengan *stride* 2 dan tanpa *padding*. Hanya harga *max* input pada setiap bidang resiptif yang akan sampai pada layer selanjutnya, sementara input-input lain ditanggalkan. Contoh, di bagian bawah bidang resiptif pada gambar 9.11, harga input adalah $1, 5, 3, 2$, maka hanya harga *max* 5 yang diteruskan ke layer selanjutnya. Karena mempunyai *stride* 2, image output mempunyai lebar dan tinggi setengah dari image input (dibulatkan karena tidak menggunakan *padding*).



Gambar 9.11: Max pooling layer (2 x 2 pooling kernel, stride 2 dan tanpa padding)

Selain untuk mengurangi komputasi, penggunaan memori dan jumlah parameter, *max* PL juga memasukan sejumlah tingkatan invariansi pada translasi-translasi kecil, seperti yang ditunjukan pada Gambar 9.12. Di sini kita mengasumsikan bahwa piksel yang cerah mempunyai harga yang lebih rendah dibandingkan dengan piksel yang gelap, dan kita mempertimbangkan tiga image (A,B,C) yang melewati *max* PL dengan kernel 2×2 dan *stride* 2. Image B dan C adalah sama dengan image A, tetapi digeser sebesar satu atau dua piksel ke kanan. Seperti yang dapat kita lihat, output-output dari *max* PL untuk image A dan B adalah identik. Inilah maksud dari invariansi translasi (*translation invariance*). Untuk image C, output berbeda yaitu tergeser satu piksel ke sebelah kanan (tetapi masih terdapat invariansi 75%). Dengan menyisipkan sebuah *max* PL pada setiap beberapa layer di CNN, maka dimungkinkan untuk mendapatkan level invariansi translasi pada skala yang lebih besar. Lebih jauh, *max* PL menawarkan sejumlah kecil invariansi rotasi dan invariansi skala kecil. Invariansi seperti ini (meskipun terbatas) menjadi berguna pada kasus-kasus dimana prediksi tidak seharusnya tergantung pada detail-detail seperti ini, seperti pada Klasifikasi.

Max PL mempunyai dua kelemahan juga. Pertama, terlihat jelas bahwa *max* PL bersifat merusak. Meskipun pada kernel kecil 2×2 dan *stride* 2, output akan dua kali lebih kecil pada kedua arah (areanya akan mengecil menjadi seperempatnya), atau mengabaikan 75% harga-harga input. Dan pada beberapa aplikasi, invariansi tidak diinginkan. Misalkan pada segmentasi semantik (melakukan klasifikasi setiap piksel pada image berdasarkan kepeimilikan piksel-piksel pada sebuah objek). Jika image input ditranslasikan satu piksel ke kanan, maka output seharusnya juga ditranslasikan satu piksel ke sebelah kanan. Tujuan pada kasus ini adalah *equivariance*, bukan *invariance*. Artinya perubahan kecil pada input harus membawa perubahan kecil yang berhubungan pada output.



Gambar 9.12: Invariansi pada translasi kecil

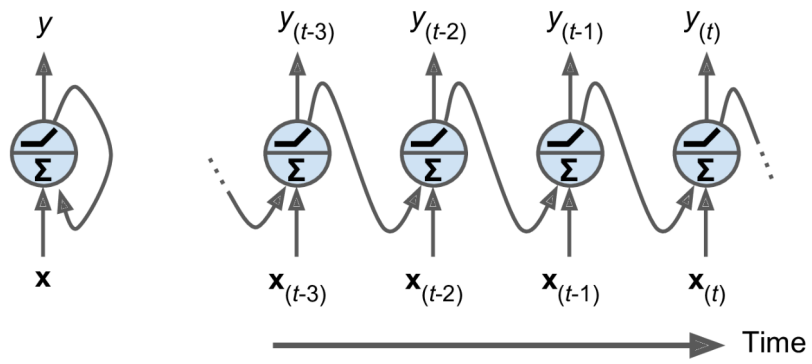
4 *Recurrent Neural Networks* (RNNs)

Recurrent neural networks (RNN) merupakan sebuah *class* networks yang dapat digunakan untuk memprediksi yang akan terjadi sampai pada titik tertentu. RNNs dapat menganalisa data *time series* seperti *stock prices* dan dapat memberikan rekomendasi kapan harus membeli dan kapan menjual. Pada *autonomous driving systems*, RNN dapat mengantisipasi trajektori dari mobil dan menghindari kecelakaan. Secara umum, RNN dapat bekerja pada urutan data dengan panjang sembarang, bukan input-input dengan ukuran tetap seperti pada network-network yang sudah kita bahas sebelumnya. Sebagai contoh, RNN dapat menjadikan kalimat, dokumen, atau sampel audio sebagai input, sehingga sangat berguna untuk aplikasi *natural language processing* (NLP) seperti penterjemah otomatis atau *speech-to-text*.

RNN bukan satu-satunya tipe *neural network* yang mampu menghandel data-data sekuensial. Untuk sekuen yang kecil, *dense network* reguler bisa melakukannya, dan untuk sekuen yang panjang seperti sampel audio atau text, CNN dapat digunakan dan bekerja cukup baik.

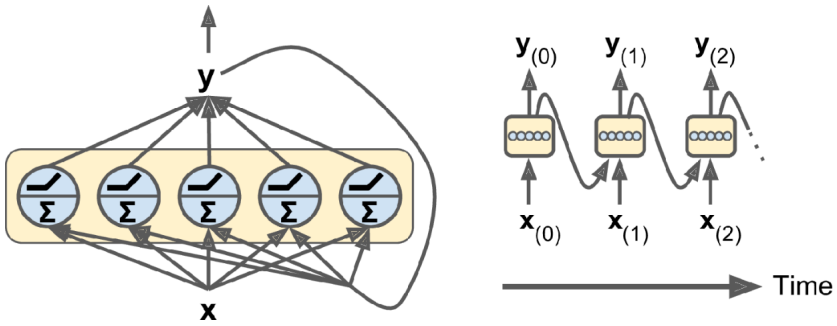
4.1 *Recurrent Neurons* dan Layer-Layer

Sampai tahap ini kita telah befokus pada *feedforward neural networks*, dimana aktivasi hanya bergerak satu arah, dari layer input ke layer output. RNN mirip dengan *feedforward neural networks*, kecuali RNN mempunyai koneksi yang mengarah ke belakang. Kita akan lihat RNN sederhana yang paling mungkin, terdiri dari satu neuron input yang menghasilkan output, dan melakukan umpan balik dari output ke input kembali, seperti yang ditunjukan pada Gambar 9.13 paling kiri. Pada tiap langkah waktu t (yang disebut juga *frame*), *recurrent neuron* menerima input $x_{(t)}$ dan umpan balik dari output sendiri dari selang waktu sebelumnya $y_{(t-1)}$. Karena tidak ada output sebelumnya saat inisial, maka biasanya dianggap 0. Kita dapat merepresentasikan network yang kecil ini terhadap sumbu waktu, seperti yang ditunjukan pada Gambar 9.13 sebelah kanan. Proses ini disebut dengan *unrolling the network through time*, dimana *recurrent neuron* yang sama direpresentasikan sekali untuk tiap langkah waktu.



Gambar 9.13: Sebuah recurrent neuron (kiri) dan evolusinya pada tiap langkah (kanan)

Kita juga bisa dengan mudah membuat sebuah layer yang terdiri dari banyak *recurrent neuron*. Pada setiap langkah waktu t, setiap neuron menerima dua masukan, yaitu vektor input $x_{(t)}$ dan vektor output dari langkah waktu sebelumnya $y_{(t-1)}$ seperti yang ditunjukan pada Gambar 9.14. Harap diperhatikan bahwa sekarang input dan output berupa vektor karena terdiri dari banyak neuron. Jika hanya terdapat satu neuron, maka output adalah skalar.



Gambar 9.14: Sebuah layer dari recurrent neuron (kiri) dan evolusinya pada langkah waktu

Setiap *recurrent neuron* mempunyai dua set pembobot, satu untuk input $x_{(t)}$ dan satu untuk output pada langkah waktu sebelumnya $y_{(t-1)}$, dengan simbol pembobot secara berurutan adalah w_x dan w_y. Jika kita mempertimbangkan keseluruhan *recurrent layer* bukan hanya satu *recurrent neuron* saja, maka kita bisa menempatkan semua vektor bobot pada dua matriks W_x dan W_y. Vektor output dari keseluruhan *recurrent layer* dapat dihitung seperti yang telah anda bayangkan, terlihat pada Persamaan (9.2).

Persamaan (9.2). Perhitungan output dari sebuah *recurrent layer* untuk satu *instance*

$$y_{(t)} = \phi\left(W_x^T x_{(t)} + W_y^T y_{(t-1)} + b\right)$$

Seperti pada *feedforward neural networks*, kita dapat menghitung output dari *recurrent layer* dengan satu kali perhitungan untuk sebuah *mini-batch* dengan menempatkan semua input-inputnya pada langkah waktu ke-t pada matriks input $X_{(t)}$, seperti yang terlihat pada Persamaan (9.3).

Persamaan (9.3). Output dari sebuah *recurrent neurons* untuk semua *instances* pada sebuah *mini-batch*

$$Y_{(t)} = \phi\left(X_{(t)}W_x + Y_{(t-1)}W_y + b\right) \quad (1)$$

$$= \phi\left(\begin{bmatrix} X_{(t)} & Y_{(t-1)} \end{bmatrix} W + b\right) \quad (2)$$

dengan $W = \begin{bmatrix} W_x \\ W_y \end{bmatrix}$

dimana pada persamaan ini:

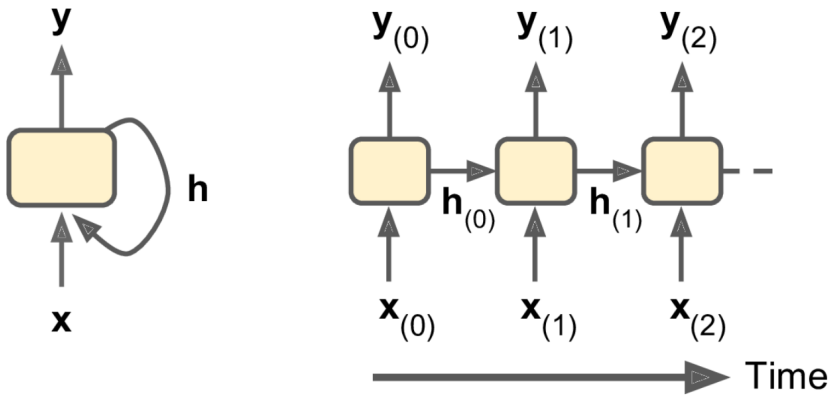
- $Y_{(t)}$ adalah adalah matriks $m \times n_{neurons}$ yang berisi output-ouput layer pada langkah waktu ke-t untuk setiap *instance* pada *mini-batch* (m adalah jumlah *instance* pada *mini-batch* dan $n_{neurons}$ adalah jumlah neuron.
- $X_{(t)}$ adalah matriks $m \times n_{input}$ yang berisi bobot-bobot koneksi untuk semua *instance*, dan n_{inputs} adalah jumlah *feature* input.
- W_x adalah adalah matriks $n_{neurons} \times n_{neurons}$ yang berisi bobot-bobot koneksi untuk input pada langkah waktu saat sekarang.
- b adalah vektor dengan ukuran $n_{neurons}$ yang berisi masing-masing *bias term*.
- Matriks pembobot W_x dan W_y sering digabung secara vertikal kedalam sebuah matriks tunggal W dengan ukuran $(n_{inputs} + n_{(neurons)}) \times n_{neurons}$ (lihat baris kedua pada Persamaan (9.3)).
- Notasi $\begin{bmatrix} X_{(t)} & Y_{(t-1)} \end{bmatrix}$ merepresentasikan penggabungan secara horisontal dari matriks $X_{(t)}$ dan $Y_{(t-1)}$.

Harap diperhatikan bahwa $Y_{(t)}$ adalah fungsi dari $X_{(t)}$ dan $Y_{(t-1)}$, yang merupakan fungsi dari $X_{(t-1)}$ dan $Y_{(t-2)}$, yang merupakan fungsi dari $X_{(t-2)}$ dan $Y_{(t-3)}$, dan seterusnya. Hal ini membuat $Y_{(t)}$ merupakan fungsi dari semua input semenjak $t = 0$ (yaitu $X_{(0)}, X_{(1)}, \cdots, X_{(t)}$. Pada langkah pertama, $t = 0$, tidak terdapat output sebelumnya, sehingga semuanya diasumsikan bernilai 0.

4.2 *Memory Cells*

Karena output dari *recurrent neuron* pada langkah waktu ke-t merupakan fungsi dri semua output-ouput langkah sebelumnya, kita bisa menyimpulkan adanya bentuk *memory*. Bagian dari *neural networks* yang memelihara beberapa *state* melalui langkah waktu disebut dengan ***memory cells*** (atau sederhananya *cell*). Sebuah *recurrent neuron*, atau layer dari *recurrent neuron* merupakan sebuah *cell* dasar, mampu untuk mempelajari hanya pola pendek (biasanya sekitar 10 langkah, tetapi bervariasi tergantung pada masalah yang akan dikerjakan).

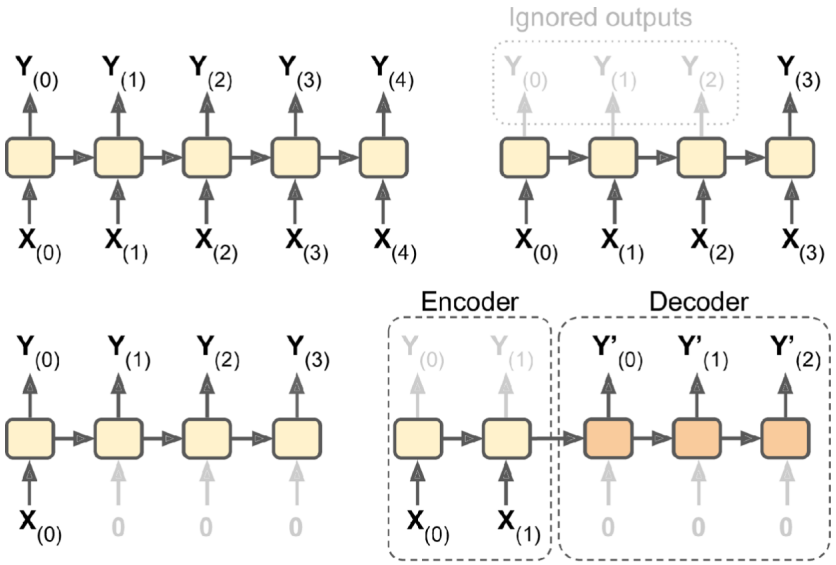
Secara umum state *cell* pada langkah waktu ke-t, dituliskan sebagai $h_{(t)}$ ('h' untuk *hidden*), merupakan fungsi dari beberapa input pada langkah waktu tersebut dan state pada langkah waktu sebelumnya $h_{(t)} = f\left(h_{(t-1)}, x_{(t)}\right)$. Outputnya pada langkah waktu ke-t, atau $y_{(t)}$ juga merupakan fungsi dari state sebelumnya dan input sekarang. Pada kasus dari *cell* dasar yang telah kita diskusikan sebelumnya, output sama dengan state, tetapi pada *cell* yang lebih kompleks maka tidak selalu seperti itu (lihat Gambar 9.15).



Gambar 9.15: Sebuah hidden state dari sebuah cell dan outputnya bisa jadi berbeda

4.3 Sekuens Input dan Output

RNN dapat secara bersamaan mengambil sekuens input dan menghasilkan sekuens dari output (lihat Gambar 9.16 di kiri atas). Tipe *sequence-to-sequence network* seperti ini berguna untuk memprediksi *time series* seperti untuk *stock prices*. Misalkan kita jadikan input *stock prices* dari N hari terakhir, dan mengeluarkan prediksi dengan menggeser satu hari ke depan sebagai prediksi (dari $N-1$ ke besok).



Gambar 9.16: Network dari Seq-to-seq (kiri atas), seq-to-vector (kanan atas), vector-to-seq (kiri bawah), dan Encoder-Decoder (kanan bawah)

Sebagai alternatif, kita bisa memberikan sebuah sekuens input terhadap network dan mengabaikan keseluruhan output kecuali hanya satu (lihat Gambar 9.16 kanan atas). Tipe ini disebut dengan *sequence-to-vector-network*. Misalkan, kita dapat memberikan sekuens kata-kata menyangkut *movie review* sebagai input dan network akan mengeluarkan skor sentimen (mis. pada skala -1 [benci] s/d $+1$ [cinta]). Sebaliknya, kita bisa juga memberikan input vektor berulang-ulang pada setiap langkah waktu dan menghasilkan output berupa sekuens (lihat Gambar 9.16 kiri bawah), yang disebut dengan *vector-to-sequence network*.

Terakhir, kita bisa juga mempunyai *sequence-to-vector network* yang disebut dengan *encoder* diikuti dengan *vector-to-sequence network* yang disebut dengan *decoder* (lihat Gambar 9.16 kanan bawah). Misalkan, skema ini bisa digunakan untuk menterjemahkan kalimat dari satu bahasa ke bahasa lain. Network akan diberikan input sebuah kalimat dari satu bahasa, *encoder* akan merubah kalimat menjadi sebuar representasi vektor, dan kemudian *decoder* akan mendekodekan vektor ini ke dalam kalimat dari bahasa yang berbeda. Model dengan dua langkah ini yang disebut dengan *Encoder-Decoder* bekerja jauh lebih baik dibandingkan dengan mencoba mentranslasikan *on the fly* dengan RNN *sequence-to-sequence* tunggal (seperti Gambar 9.16 kiri atas). Kata-kata terakhir dari sebuah kalimat bisa mempengaruhi kata-kata pertama dari translasi, sehingga kita perlu menunggu sampai melihat kalimat secara keseluruhan sebelum menterjemahkannya.