11

Feedforward Neural Network

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

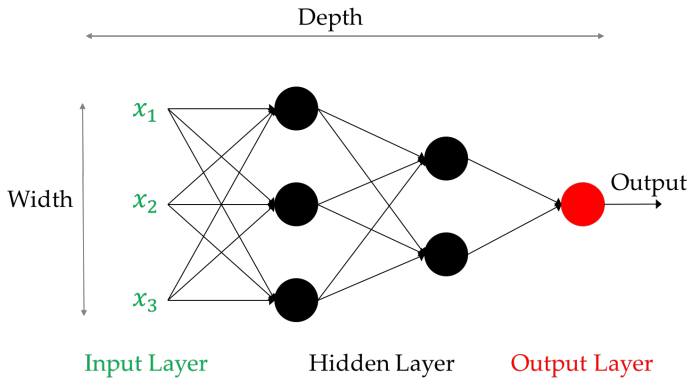
Penulis cukup yakin pembaca sudah menanti-nanti bagian ini. Bagian ketiga membahas algoritma *machine learning* yang sedang populer belakangan ini, yaitu *artificial neural network*. Buku ini lebih berfokus pada penggunaan *artifical neural network* untuk *supervised learning*. Pembahasan dimulai dari hal-hal sederhana (*single perceptron*, *multilayer perceptron*) sampai yang lebih kompleks. Sebelum membaca bab ini, ada baiknya kamu mengulang membaca bab 5 karena memberikan fondasi matematis untuk mengerti bab ini.

11.1 Definisi Artificial Neural Network

Masih ingatkah Anda materi pada bab-bab sebelumnya? *Machine learning* sebenarnya meniru bagaimana proses manusia belajar. Pada bagian ini, peneliti ingin meniru proses belajar tersebut dengan mensimulasikan jaringan saraf biologis (*neural network*) [44, 45, 46, 47]. Kami yakin banyak yang sudah tidak asing dengan istilah ini, berhubung *deep learning* sedang populer dan banyak yang membicarakannya (dan digunakan sebagai trik pemasaran). *Artificial neural network* adalah salah satu algoritma *supervised learning* yang

populer dan bisa juga digunakan untuk *semi-supervised* atau *unsupervised learning* [45, 47, 48, 49, 50]. Walaupun tujuan awalnya adalah untuk mensimulasikan jaringan saraf biologis, jaringan tiruan ini sebenenarnya simulasi yang terlalu disederhanakan, artinya simulasi yang dilakukan tidak mampu menggambarkan kompleksitas jaringan biologis manusia (menurut penulis).[1]

Artificial Neural Network (selanjutnya disingkat ANN), menghasilkan model yang sulit dibaca dan dimengerti oleh manusia karena memiliki banyak layer (kecuali *single perceptron*) dan sifat **non-linear** (merujuk pada fungsi aktivasi). Pada bidang riset ini, ANN disebut agnostik–kita percaya, tetapi sulit membuktikan kenapa konfigurasi parameter yang dihasilkan *training* bisa benar. Konsep matematis ANN itu sendiri cukup *solid*, tetapi *interpretability* model rendah menyebabkan kita tidak dapat menganalisa proses inferensi yang terjadi pada model ANN. Secara matematis, ANN ibarat sebuah graf. ANN memiliki neuron/*node* (*vertex*), dan sinapsis (*edge*). Topologi ANN akan dibahas lebih detil subbab berikutnya. Karena memiliki struktur seperti graf, operasi pada ANN mudah dijelaskan dalam notasi aljabar linear. Sebagai gambaran, ANN berbentuk seperti Gambar 11.1 (*deep neural network*, salah satu varian arsitektur). *Depth* (kedalaman) ANN mengacu pada banyaknya *layer*. Sementara *width* (lebar) ANN mengacu pada banyaknya unit pada *layer*.



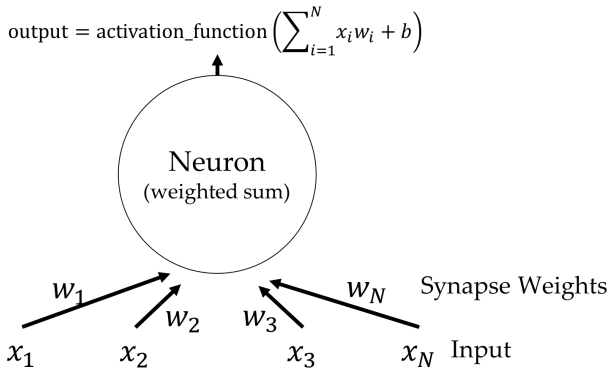
Gambar 11.1: *Deep Neural Network*.

11.2 Single Perceptron

Bentuk terkecil (minimal) sebuah ANN adalah *single perceptron* yang hanya terdiri dari sebuah neuron. Sebuah neuron diilustrasikan pada Gambar 11.2.

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

Secara matematis, terdapat *feature vector* **x** yang menjadi *input* bagi neuron tersebut. Ingat kembali, *feature vector* merepresentasikan suatu *data point*, *event* atau *instance*. Neuron akan memproses *input* **x** melalui perhitungan jumlah perkalian antara nilai *input* dan ***synapse weight***, yang dilewatkan pada **fungsi non-linear** [51, 52, 4]. Pada *training*, yang dioptimasi adalah nilai *synapse weight* (*learning parameter*). Selain itu, terdapat juga bias b sebagai kontrol tambahan (ingat materi *steepest gradient descent*). *Output* dari neuron adalah hasil fungsi aktivasi dari perhitungan jumlah perkalian antara nilai *input* dan *synapse weight*. Ada beberapa macam fungsi aktivasi, misal ***step function***, ***sign function***, ***rectifier*** dan ***sigmoid function***. Untuk selanjutnya, pada bab ini, fungsi aktivasi yang dimaksud adalah jenis ***sigmoid function***. Silahkan eksplorasi sendiri untuk fungsi aktivasi lainnya. Salah satu bentuk tipe ***sigmoid function*** diberikan pada persamaan 11.1. Bila di-*plot* menjadi grafik, fungsi ini memberikan bentuk seperti huruf "S".



Gambar 11.2: *Single Perceptron.*

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

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

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

dimana o adalah *output* dan f adalah fungsi **non-linear yang dapat diturunkan secara matematis** (*differentiable non-linear function*–selanjutnya disebut "fungsi non-linear" saja.). Bentuk ini tidak lain dan tidak bukan adalah persamaan model linear yang ditransformasi dengan fungsi non-linear.

Secara filosofis, ANN bekerja mirip dengan model linear, yaitu mencari *decision boudary*. Apabila beberapa model non-linear ini digabungkan, maka kemampuannya akan menjadi lebih hebat (subbab berikutnya). Yang menjadikan ANN "spesial" adalah penggunaan fungsi non-linear.

Untuk melakukan pembelajaran *single perceptron*, *training* dilakukan menggunakan ***perceptron training rule***. Prosesnya sebagai berikut [4, 51, 52]:

1. Inisiasi nilai *synapse weights*, bisa *random* ataupun dengan aturan tertentu. Untuk heuristik aturan inisiasi, ada baiknya membaca buku referensi [1, 11].
2. Lewatkan input pada neuron, kemudian kita akan mendapatkan nilai *output*. Kegiatan ini disebut ***feedforward***.
3. Nilai *output* (*actual output*) tersebut dibandingkan dengan *desired output*.
4. Apabila nilai *output* sesuai dengan *desired output*, tidak perlu mengubah apa-apa.
5. Apabila nilai *output* tidak sesuai dengan *desired output*, hitung nilai *error* (*loss*) kemudian lakukan perubahan terhadap *learning parameter* (*synapse weight*).
6. Ulangi langkah-langkah ini sampai tidak ada perubahan nilai *error*, nilai *error* kurang dari sama dengan suatu *threshold* (biasanya mendekati 0), atau sudah mengulangi proses latihan sebanyak T kali (*threshold*).

Salah satu cara melatih *neural network* adalah dengan mengoptimalkan *error function*, diberikan pada persamaan 11.3[2] (dapat diganti dengan *absolute value*). Perubahan nilai *synapse weight* saat proses latihan (apabila $E \neq 0$ diberikan pada persamaan 11.4, dimana y melambangkan *desired output*,[3] $o = f(\mathbf{x} \cdot \mathbf{w} + \mathbf{b})$ melambangkan *actual output* untuk $\mathbf{x}$ sebagai input, η adalah *learning rate*.

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

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

Hasil akhir pembelajaran adalah konfigurasi *synapse weight* yang mengoptimalkan nilai *error*. Saat klasifikasi, kita melewatkan *input* baru pada jaringan yang telah dibangun, kemudian tinggal mengambil hasilnya. Pada contoh kali ini, seolah-olah *single perceptron* hanya dapat digunakan untuk melakukan *binary classification* (hanya ada dua kelas, nilai 0 dan 1). Untuk *multi-class classification*, kita dapat menerapkan berbagai strategi, misal *thresholding*, i.e., nilai *output* $0 - 0.2$ mengacu pada kelas pertama, $0.2 - 0.4$ untuk kelas kedua, dst.

[2] Pada umumnya, kita tidak menggunakan satu data, tetapi *batch-sized*.
[3] Pada contoh ini, kebetulan banyaknya *output* neuron hanyalah satu.

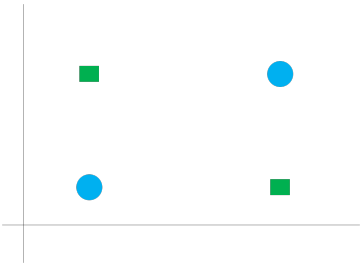
11.3 Permasalahan XOR

Sedikit sejarah, *perceptron* sebenarnya cukup populer sekitar tahun 1950-1960. Entah karena suatu sebab, *perceptron* menjadi tidak populer dan digantikan oleh model linear. Saat itu, belum ada algoritma yang bekerja dengan relatif bagus untuk melatih *perceptron* yang digabungkan (*multilayer perceptron*). Model linear mendapat popularitas hingga kira-kira sekitar tahun 1990'an atau awal 2000'an. Berkat penemuan *backpropagation* sekitar awal 1980,[4] *multilayer perceptron* menjadi semakin populer. Perlu dicatat, komunitas riset bisa jadi seperti cerita ini. Suatu teknik yang baru belum tentu bisa segera diimplementasikan karena beberapa kendala (misal kendala kemampuan komputasi).

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

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

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

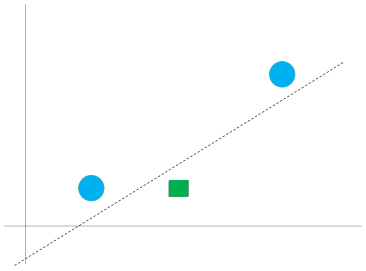


Gambar 11.3: Permasalahan XOR.

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

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

[4] http://people.idsia.ch/~juergen/who-invented-backpropagation.html



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

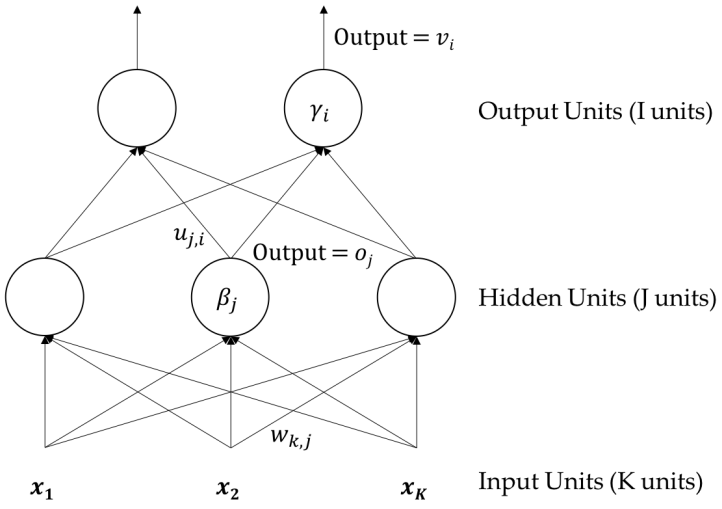
Sudah dijelaskan pada bab model linear, permasalahan dunia nyata tidak sesederhana ini (kebetulan ditransformasikan menjadi data dengan dimensi yang sama). Pada permasalahan praktis, kita harus mentransformasi data menjadi dimensi lebih tinggi (dari 2D menjadi 3D). Berbeda dengan ide utama linear model/*kernel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

Perlu dicatat, pemodelan non-linear inilah yang membuat ANN menjadi hebat. ANN mungkin secara luas didefinisikan mencakup *single perceptron* tetapi secara praktis, **ANN sebenarnya mengacu pada *multilayer perceptron* dan arsitektur lebih kompleks** (dijelaskan pada subbab berikutnya). Pada masa ini, (hampir) tidak ada lagi yang menggunakan *single perceptron*. Untuk bab-bab kedepan, ketika kami menyebut ANN maka yang diacu adalah *multilayer perceptron* dan arsitektur lebih kompleks (*single perceptron* di-*exclude*). Hal ini disebabkan oleh *single perceptron* tidak dapat mempelajari XOR *function* secara independen tanpa *feature engineering*, sementara *multilayer perceptron* bisa [53].

11.4 Multilayer Perceptron

Kamu sudah belajar bagaimana proses *training* untuk *single perceptron*. Selanjutnya kita akan mempelajari *multilayer perceptron* (MLP) yang juga dikenal sebagai ***feedforward neural network***. Kami tekankan sekali lagi, **istilah "ANN" selanjutnya mengacu pada MLP dan arsitektur lebih kompleks**.

Perhatikan ilustrasi pada Gambar 11.5, *multilayer perceptron* secara literal memiliki beberapa *layers*. Pada buku ini, secara umum ada tiga *layers*: *input*, *hidden*, dan *output layer*. *Input layer* menerima *input* (tanpa melakukan operasi apapun), kemudian nilai *input* (tanpa dilewatkan ke fungsi aktivasi) diberikan ke *hidden units* (persamaan 11.6). Pada *hidden units*, *input* diproses dan dilakukan perhitungan hasil fungsi aktivasi untuk tiap-tiap neuron, lalu hasilnya diberikan ke *layer* berikutnya (persamaan 11.7, σ adalah fungsi aktivasi). *Output* dari *input layer* akan diterima sebagai input bagi *hidden layer*. Begitupula seterusnya *hidden layer* akan mengirimkan hasilnya untuk *output layer*. Kegiatan ini dinamakan ***feed forward*** [45, 4]. Hal serupa berlaku untuk *artificial neural network* dengan lebih dari tiga *layers*. Parameter neuron dapat dioptimisasi menggunakan metode *gradient-based optimization* (dibahas pada subabb berikutnya, ingat kembali bab 5). Perlu diperhatikan, MLP adalah gabungan dari banyak fungsi non-linear. Seperti yang disampaikan pada subbab sebelumnya, gabungan banyak fungsi non-linear ini lebih hebat dibanding *single perceptron*. Seperti yang kamu lihat pada Gambar 11.5, masing-masing neuron terkoneksi dengan semua neuron pada *layer* berikutnya. Konfigurasi ini disebut sebagai ***fully connected***. MLP pada umumnya menggunakan konfigurasi *fully connected*.



Gambar 11.5: *Multilayer Perceptron* 2.

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

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

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

$$\mathbf{v} = \sigma(\mathbf{o}\mathbf{U} + \gamma) = \sigma((\sigma(\mathbf{x}\mathbf{W} + \beta))\mathbf{U} + \gamma) \tag{11.8}$$

Untuk melatih MLP, algoritma yang umumnya digunakan adalah ***backpropagation*** [54]. Arti kata *backpropagation* sulit untuk diterjemahkan ke dalam bahasa Indonesia. Kita memperbaharui parameter (*synapse weights*) secara bertahap (dari *output* ke *input* layer, karena itu disebut *backpropagation*) berdasarkan *error*/*loss* (*output* dibandingkan dengan *desired output*). Intinya adalah mengkoreksi *synapse weight* dari *output layer* ke *hidden layer*, kemudian *error* tersebut dipropagasi ke layer sebelum-sebelumnya. Artinya, perubahan *synapse weight* pada suatu layer dipengaruhi oleh perubahan *synapse weight* pada layer setelahnya.[5] *Backpropagation* tidak lain dan tidak bukan adalah metode *gradient-based optimization* yang diterapkan pada ANN.

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

Ingat kembali *chain rule* pada perkuliahan diferensial

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

Error, untuk MLP diberikan oleh persamaan 11.10 (untuk satu data *point*), dimana I adalah banyaknya *output unit* dan θ adalah kumpulan *weight matrices* (semua parameter pada MLP). Kami inggatkan kembali perhitungan *error* bisa juga menggunakan nilai absolut.[6]

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

[6] Kami menggunakan tanda kurung agar lebih mudah dibaca penurunan rumusnya.

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

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

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

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

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

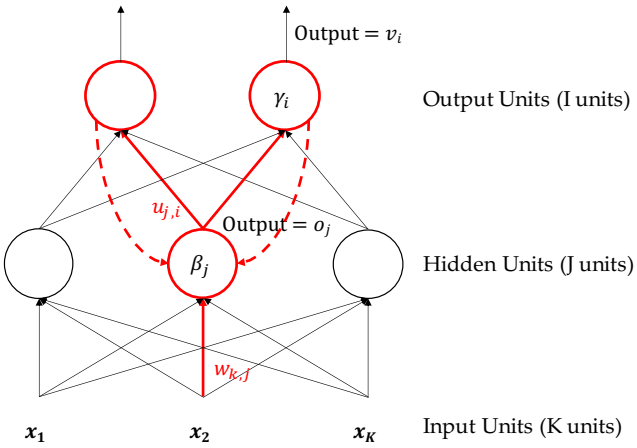
$$\begin{aligned}\frac{\delta E(\theta)}{\delta w_{k,j}} &= \sum_{i=1}^{I}(y_i - v_i)\frac{\delta v_i}{\delta w_{k,j}} \\ &= \sum_{i=1}^{I}(y_i - v_i)\frac{\delta v_i}{\delta o_j}\frac{\delta o_j}{\delta w_{k,j}} \\ &= \sum_{i=1}^{I}(y_i - v_i)(v_i(1 - v_i)u_{j,i})(o_j(1 - o_j)x_k)\end{aligned}$$

Perhatikan, diferensial $w_{k,j}$ memiliki $\sum$ sementara $u_{j,i}$ tidak ada. Hal ini disebabkan karena $u_{j,i}$ hanya berkorespondensi dengan satu *output* neuron. Sementara $w_{k,j}$ berkorespondensi dengan banyak *output* neuron. Dengan kata lain, nilai $w_{k,j}$ mempengaruhi hasil operasi yang terjadi pada banyak *output* neuron, sehingga banyak neuron mempropagasi *error* kembali ke $w_{k,j}$. Ilustrasi diberikan pada Gambar 11.6.

Metode penurunan serupa dapat juga digunakan untuk menentukan perubahan β dan γ. Jadi proses *backpropagation* untuk kasus Gambar 11.5 dapat diberikan seperti pada Gambar 11.7 dimana η adalah *learning rate*. Untuk *artificial neural network dengan* lebih dari 3 *layers*, kita pun bisa menurunkan persamaannya. Secara umum, proses melatih ANN (apapun variasi arsitekturnya) mengikuti *framework perceptron training rule* (subbab 11.2). Cerita pada buku ini menggunakan *error* sebagai *utility function* karena mudah diilustrasikan, serta contoh ini sering digunakan pada referensi lainnya. Pada permasalahan praktis, *cross entropy* adalah *utility function* yang sering digunakan untuk melatih ANN (untuk permasalahan klasifikasi).

11.5 Interpretability

Interpretability ada dua macam yaitu *model interpretability* (i.e., apakah struktur model pembelajaran mesin dapat dipahami) dan *prediction interpretability* (i.e., bagaimana memahami dan memverifikasi cara *input* dipetakan



Gambar 11.6: Justifikasi penggunaan $\sum$ pada penurunan dari *hidden* ke *input layer*.

(2) Hidden to Output

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

(3) Output to Hidden

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

(1) Input to Hidden Layer

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

(4) Hidden to Input

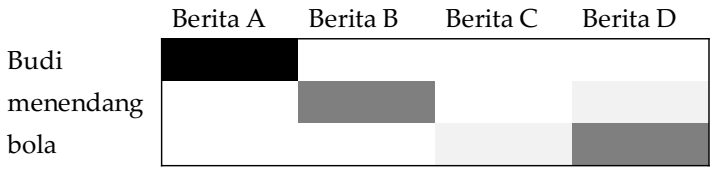
$$\varphi_j = \sum_{i=1}^{I} \delta_i u_{j,i} o_j(1 - o_j)$$
$$\Delta w_{k,j} = -\eta(t)\varphi_j x_k$$
$$\Delta\beta_j = -\eta(t)\varphi_j$$

Gambar 11.7: Proses latihan MLP menggunakan *backpropagation*.

menjadi *output*) [55]. Contoh teknik pembelajaran mesin yang mudah diinterpretasikan baik secara struktur dan prediksinya adalah *decision tree* (bab 6). Struktur *decision tree* berupa pohon keputusan mudah dimengerti oleh manusia dan prediksi (keputusan) dapat dilacak (*trace*). Seperti yang sudah dijelaskan pada bagian pengantar, ANN (MLP) biasanya dianggap sebagai metode *black box* atau susah untuk dinterpretasikan (terutama *model*

interpretability-nya). Hal ini disebabkan oleh kedalaman (*depth*) yaitu memiliki beberapa *layer* dan *non-linearities*. Suatu unit pada *output layer* dipengaruhi oleh kombinasi (*arbitrary combination*) banyak parameter pada *layers* sebelumnya yang dilewatkan pada fungsi non-linear. Sulit untuk mengetahui bagaimana pengaruh bobot suatu unit pada suatu *layer* berpengaruh pada *output layer*, beserta bagaimana pengaruh kombinasi bobot. Intinya, fitur dan *output* tidak memiliki korespondensi satu-satu. Berbeda dengan model linear, kita tahu parameter (dan bobotnya) untuk setiap *input*. Salah satu arah riset adalah mencari cara agar keputusan yang dihasilkan oleh ANN dapat dijelaskan [56],[7] salah satu contoh nyata adalah *attention mechanism* [57, 58] (subbab 13.4.4) untuk *prediction interpretability*. Survey tentang *interpretability* dapat dibaca pada *paper* oleh Doshi-Velez dan Kim [59].

Cara paling umum untuk menjelaskan keputusan pada ANN adalah menggunakan *heat map*. Sederhananya, kita lewatkan suatu data $\mathbf{x}$ pada ANN, kemudian kita lakukan *feed-forward* sekali (misal dari *input* ke *hidden layer* dengan parameter $\mathbf{W}$). Kemudian, kita visualisasikan $\mathbf{x} \cdot \mathbf{W}$ (ilustrasi pada Gambar 11.8). Dengan ini, kita kurang lebih dapat mengetahui bagian *input* mana yang berpengaruh terhadap keputusan di *layer* berikutnya.



Gambar 11.8: Contoh *heat map* pada persoalan klasifikasi teks. *Input* berupa kata-kata yang dimuat pada suatu berita. Output adalah kategori berita untuk *input*. Warna lebih gelap menandakan bobot lebih tinggi. Sebagai contoh, kata "menendang" berkorespondensi paling erat dengan kelas berita B.

11.6 Binary Classification

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

[7] Karena struktur lebih susah, setidaknya beranjak dari keputusan terlebih dahulu
[8] https://en.wikipedia.org/wiki/Sign_function
[9] https://en.wikipedia.org/wiki/Hyperbolic_function

gunakan fungsi seperti sigmoid,[10] dimana *output* pada masing-masing neuron berada pada *range* nilai $[0, 1]$.

11.7 Multi-class Classification

Multilayer perceptron dapat memiliki lebih dari satu *output unit*. Seumpama kita mempunyai empat kelas, dengan demikian kita dapat merepresentasikan keempat kelas tersebut sebagai empat *output units*. Kelas pertama direpresentasikan dengan unit pertama, kelas kedua dengan unit kedua, dst. Untuk C kelas, kita dapat merepresentasikannya dengan C *output units*. Kita dapat merepresentasikan data harus dimasukkan ke kelas mana menggunakan *sparse vector*, yaitu bernilai 0 atau 1. Elemen ke-i bernilai 1 apabila data masuk ke kelas c_i, sementara nilai elemen lainnya adalah 0 (ilustrasi pada Gambar 11.9).*Output* ANN dilewatkan pada suatu fungsi softmax yang melambangkan probabilitas *class-assignment*; i.e., kita ingin *output* agar semirip mungkin dengan *sparse vector* (*desired output*). Pada kasus ini, *output* ANN adalah sebuah distribusi yang melambangkan *input* di-*assign* ke kelas tertentu. Ingat kembali materi bab 5, *cross entropy* cocok digunakan sebagai *utility function* ketika *output* berbentuk distribusi.

$$\begin{matrix} c_1 & c_2 & c_3 & c_4 \\ \end{matrix}$$
$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$
Label = c_1
Label = c_2
Label = c_3
Label = c_4

Gambar 11.9: Ilustrasi representasi *desired output* pada *multi-class classification* menggunakan *sparse vector* (sering disebut sebagai "*one-hot vector*" di komunitas ANN).

11.8 Multi-label Classification

Seperti halnya *multi-class classification*, kita dapat menggunakan sebanyak C neuron untuk merepresentasikan C kelas pada *multi-label classification*. Seperti yang sudah dijelaskan pada bab 5, perbedaan *multi-class* dan *multi-label* terletak pada cara interpretasi *output* dan evaluasi *output*. Pada umumnya, *layer* terakhir diaktivasi dengan fungsi sigmoid, dimana tiap neuron n_i merepresentasikan probabilitas suatu dapat diklasifikasikan sebagai kelas c_i

[10] https://en.wikipedia.org/wiki/Sigmoid_function

$$\begin{array}{cccc} c_1 & c_2 & c_3 & c_4 \\ \left[\begin{array}{cccc} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 \end{array}\right. & & & \left.\begin{array}{l} \text{Label} = c_1, c_3 \\ \text{Label} = c_2 \\ \text{Label} = c_1, c_4 \\ \text{Label} = c_2, c_3, c_4 \end{array}\right] \end{array}$$

Gambar 11.10: Ilustrasi representasi *desired output* pada *multi-label classification*.

atau tidak (Gambar 11.10). Pada umumnya, *binary cross entropy* digunakan sebagai *loss* (*utility function*) pada *multi-label classification*, yaitu perhitungan *cross entropy* untuk tiap-tiap *output* unit (bukan sekaligus semua *output unit* seperti pada *multi-class classification*).

11.9 Deep Neural Network

Deep Neural Network (DNN) adalah *artificial neural network* yang memiliki banyak *layer*. Pada umumnya, *deep neural network* memiliki lebih dari 3 *layers* (*input layer*, $N \geq 2$ *hidden layers*, *output layer*), dengan kata lain adalah MLP dengan lebih banyak *layer*. Karena ada relatif banyak layer, disebutlah *deep*. Proses pembelajaran pada DNN disebut sebagai *deep learning* (tidak ada bedanya dengan proses latihan pada jaringan yang lebih dangkal, ini hanyalah istilah keren saja) [9]. Jaringan *neural network* pada DNN disebut *deep neural network*.[11]

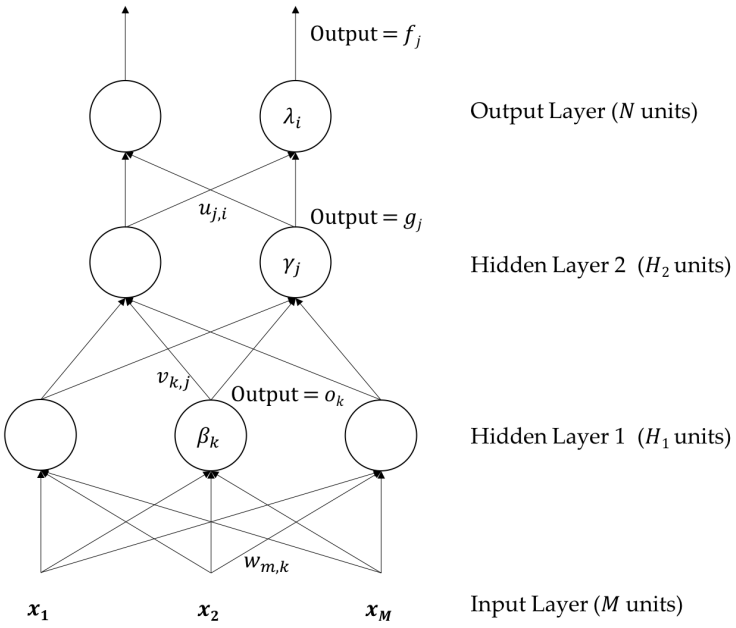
Perhatikan Gambar 11.11 yang memiliki 4 *layers*. Cara menghitung *final output* sama seperti MLP, diberikan pada persamaan 11.11 dimana β, γ, λ adalah *noise* atau *bias*, σ adalah fungsi aktivasi.

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

Cara melatih *deep neural network*, salah satunya dapat menggunakan *back-propagation*. Seperti pada bagian sebelumnya, kita hanya perlu menurunkan rumusnya saja. **Penurunan diserahkan pada pembaca sebagai latihan**. Hasil proses penurunan dapat dilihat pada Gambar 11.12.

Deep network terdiri dari banyak *layer* dan *synapse weight*, karenanya estimasi parameter susah dilakukan. Arti filosofisnya adalah susah/lama untuk menentukan relasi antara *input* dan *output*. Walaupun *deep learning* sepertinya kompleks, tetapi entah kenapa dapat bekerja dengan baik untuk permasalahan praktis [9]. *Deep learning* dapat menemukan relasi "tersembunyi"

[11] Terkadang disingkat menjadi *deep network* saja.



Gambar 11.11: *Deep Neural Network*.

(3) Hidden 2 to Output

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

(4) Output to Hidden 2

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

(2) Hidden 1 to Hidden 2

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

(5) Hidden 2 to Hidden 1

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

(1) Input to Hidden Layer

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

(6) Hidden 1 to Input

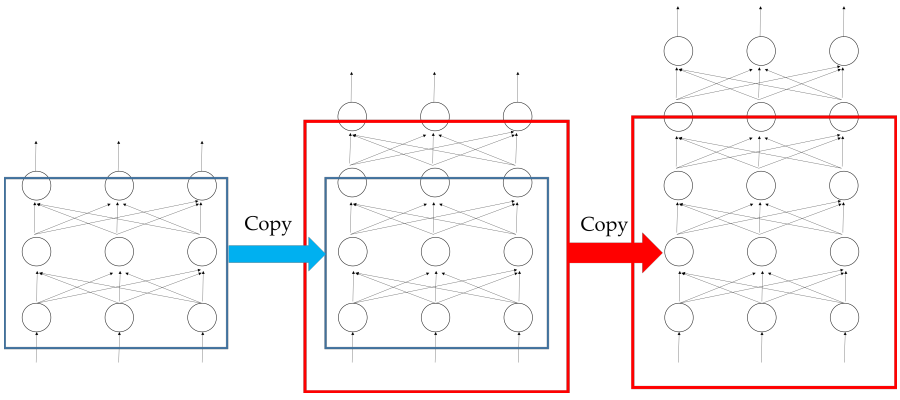
$$\mu_k = \sum_{j=1}^{H_2} \varphi_j v_{k,j} o_k (1 - o_k)$$
$$\Delta w_{m,k} = -\eta(t)\mu_k x_m$$
$$\Delta\beta_k = -\eta(t)\beta_k$$

Gambar 11.12: Proses latihan DNN menggunakan *backpropagation*.

antara *input* dan *output*, yang tidak dapat diselesaikan menggunakan *multi-layer perceptron* (3 layers). Perhatikan, kamu harus ingat bahwa satu langkah *feedforward* memiliki analogi dengan transformasi. Jadi, *input* ditransformasikan secara non-linear sampai akhirnya pada *output*, berbentuk distribusi *class-assignment*.

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

Karena memiliki banyak parameter, proses latihan ANN pada umumnya lambat. Ada beberapa strategi untuk mempercepat pembelajaran menggunakan deep learning, misalnya: regularisasi, ***successive learning***, dan penggunaan ***autoencoder*** [9]. Sebagai contoh, arti *successive learning* adalah jaringan yang dibangun secara bertahap. Misal kita latih ANN dengan 3 *layers*, kemudian kita lanjutkan 3 *layers* tersebut menjadi 4 *layers*, lalu kita latih lagi menjadi 5 *layers*, dst. Hal ini sesuai dengan [60], yaitu mulai dari hal kecil. Ilustrasinya dapat dilihat pada Gambar 11.13. Menggunakan *deep learning* harus hati-hati karena pembelajaran cenderung *divergen* (artinya, *minimum squared error* belum tentu semakin rendah seiring berjalannya waktu–*swing* relatif sering).



Gambar 11.13: Contoh *successive learning*.

11.10 Tips

Pada contoh yang diberikan, *error* atau *loss* dihitung per tiap data point. Artinya begitu ada melewatkan suatu *input*, parameter langsung dioptimisasi sesuai dengan *loss*. Pada umumnya, hal ini tidak baik untuk dilakukan karena ANN menjadi tidak stabil. Metode yang lebih baik digunakan adalah teknik *minibatches*. Yaitu mengoptimisasi parameter untuk beberapa buah *inputs*. Jadi, update parameter dilakukan per *batch*. Perhitungan *error* juga berubah, diberikan pada persamaan 11.12 dimana B melambangkan ***batch size*** (banyaknya *instance* per *batch*), $\mathbf{y}$ adalah *desired output* dan $\mathbf{o}$ adalah *actual output*. Perhitungan *error* saat menggunakan *minibatches* secara sederhana adalah rata-rata (bisa diganti dengan penjumlahan saja) individual *error* untuk semua *instance* yang ada pada *batch* bersangkutan. Setelah menghitung *error* per *batch*, barulah *backpropagation* dilakukan.

$$E(\text{minibatch}) = \frac{1}{B}\sum_{i=1}^{B} \|\mathbf{y} - \mathbf{o}\|^2 \tag{11.12}$$

Data mana saja yang dimasukkan ke suatu *batch* dalam dipilih secara acak. Seperti yang mungkin kamu sadari secara intuitif, urutan data yang disajikan saat *training* mempengaruhi kinerja ANN. Pengacakan ini menjadi penting agar ANN mampu mengeneralisasi dengan baik. Kita dapat mengatur laju pembelajaran dengan menggunakan *learning rate*. Selain menggunakan *learning rate*, kita juga dapat menggunakan *momentum* (subbab 5.6).

Pada library/API *deep learning*, *learning rate* pada umumnya berubah-ubah sesuai dengan waktu. Selain itu, tidak ada nilai khusus (*rule-of-thumb*) untuk *learning rate* terbaik. Pada umumnya, kita inisiasi *learning rate* dengan nilai $\{0.001, 0.01, 0.1\}$ [1]. Biasanya, kita menginisiasi proses latihan dengan nilai *learning rate* cukup besar, kemudian mengecil seiring berjalannya waktu.[12] Kemudian, kita mencari konfigurasi parameter terbaik dengan metode ***grid-search***,[13] yaitu dengan mencoba-coba parameter secara *exhaustive* (*brute-force*) kemudian memilih parameter yang memberikan kinerja terbaik.

ANN sensitif terhadap inisialisasi parameter, dengan demikian banyak metode inisialisasi parameter misalkan, nilai *synapse weights* diambil dari distribusi binomial (silahkan eksplorasi lebih lanjut). Dengan hal ini, kinerja ANN dengan arsitektur yang sama dapat berbeda ketika dilatih ulang dari awal. Karena sensitif terhadap inisialisasi parameter, kamu bisa saja mendapatkan performa yang berbeda saat melatih suatu arsitektur ANN dengan data yang sama (untuk permasalahan yang sama). Untuk menghindari *bias* inisialisasi parameter, biasanya ANN dilatih beberapa kali (umumnya 5, 10, atau 20 kali). Kinerja ANN yang dilaporkan adalah nilai kinerja rata-rata

[12] Analogi: ngebut saat baru berangkat, kemudian memelan saat sudah dekat dengan tujuan agar tidak terlewat.

[13] https://en.wikipedia.org/wiki/Hyperparameter_optimization

dan varians (*variance*). Hal ini untuk menghindari *overclaiming*. Pada konteks pembahasan saat ini, artinya seseorang bisa saja mendapatkan model dengan performa sangat baik, padahal performa arsitektur yang sama bisa saja berkurang saat dicoba oleh orang lain (isu *replicability*).

Kamu mungkin sudah menyadari bahwa melatih ANN harus telaten, terutama dibanding model linear. Untuk model linear, ia akan memberikan konfigurasi parameter yang sama untuk *training data* yang sama (kinerja pun sama). Tetapi, ANN dapat konfigurasi parameter yang berbeda untuk *training data* yang sama (kinerja pun berbeda). Pada model linear, kemungkinan besar variasi terjadi saat mengganti data. Pada ANN, variasi kinerja ada pada seluruh proses! Untuk membandingkan dua arsitektur ANN pada suatu dataset, kita dapat menggunakan *statistical hypothesis testing* (arsitektur X lebih baik dari arsitektur Y secara signifikan dengan nilai $p < threshold$). Penulis merekomendasikan untuk membaca [14, 15] perihal *hypothesis testing*.

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

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

Curriculum learning juga adalah tips yang layak disebutkan (*mention*) [61]. Penulis kurang mengerti detilnya, sehingga pembaca diharapkan membaca sendiri. Intinya adalah memutuskan apa yang harus ANN pelajari terlebih dahulu (mulai dari mempelajari hal mudah sebelum mempelajari hal yang susah).

11.11 Regularization and Dropout

Seperti yang sudah dijelaskan pada model linear. Kita ingin model mengeneralisasi dengan baik (kinerja baik pada *training data* dan *unseen examples*). Kita dapat menambahkan fungsi regularisasi untuk mengontrol kompleksitas ANN. Regularisasi pada ANN cukup *straightforward* seperti regularisasi pada model linear (subbab 5.9). Kami yakin pembaca bisa mengeksplorasi sendiri.

Selain itu, agar ANN tidak "bergantung" pada satu atau beberapa *synapse weights* saja, kita dapat menggunakan *dropout*. *Dropout* berarti me-*nol*-kan

nilai *synapse weights* dengan nilai *rate* tertentu. Misalkan kita *nol*-kan nilai 30% *synapse weights* (*dropout rate*= 0.3) secara random. Hal ini dapat dicapai dengan teknik *masking*, yaitu mengalikan *synapse weights* dengan suatu *mask*.

Ingat kembali ANN secara umum, persamaan 11.13 dimana $\mathbf{W}$ adalah *synapse weights*, $\mathbf{x}$ adalah *input* (dalam pembahasan saat ini, dapat merepresentasikan *hidden state* pada suatu layer), b adalah *bias* dan f adalah fungsi aktivasi (non-linear). Kita buat suatu *mask* untuk *synapse weights* seperti pada persamaan 11.14, dimana $\mathbf{p}$ adalah vektor dan $p_i = [0, 1]$ merepresentasikan *synapse weight* diikutsertakan atau tidak. $r\%$ (*dropout rate*) elemen vektor $\mathbf{p}$ bernilai 0. Biasanya $\mathbf{p}$ diambil dari *bernoulli distribution* [1]. Kemudian, saat *feed forward*, kita ganti *synapse weights* menggunakan *mask* seperti pada persamaan 11.15. Saat menghitung *backpropagation*, turunan fungsi juga mengikutsertakan *mask* (gradient di-*mask*). Kami sarankan untuk membaca *paper* oleh Srivastava et al. [62] tentang *dropout* pada ANN. Contoh implementasi *dropout* dapat dilihat pada pranala berikut.[14] Teknik *regularization* dan *dropout* sudah menjadi metode yang cukup "standar" dan diaplikasikan pada berbagai macam arsitektur.

$$o = f(\mathbf{x} \cdot \mathbf{W} + b) \tag{11.13}$$

$$\mathbf{W}' = \mathbf{p} \cdot \mathbf{W} \tag{11.14}$$

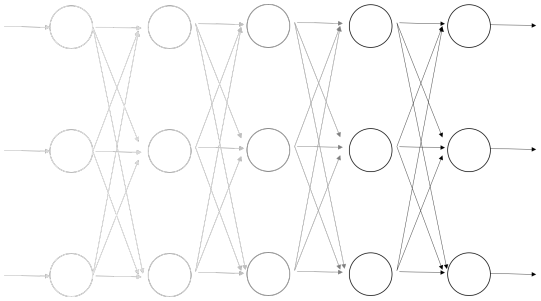
$$o = f(\mathbf{x} \cdot \mathbf{W}' + b) \tag{11.15}$$

11.12 Vanishing and Exploding Gradients

Pada beberapa kasus, nilai gradien ($\Delta\mathbf{W}$ - perubahan parameter) sangat kecil (mendekati nol - *vanishing*) atau sangat besar (*explode*). *Vanishing gradient problem* umum terjadi untuk ANN yang sangat dalam (*deep*), yaitu memiliki banyak *layer*. Hal ini juga terjadi pada arsitektur khusus, seperti *recurrent neural network* saat diberikan input yang panjang [63]. Turunan suatu fungsi bernilai lebih kecil dari fungsi tersebut. Artinya nilai gradient pada *input layer* bernilai lebih kecil dari *output layer*. Apabila kita memiliki banyak *layer*, nilai gradient saat *backpropagation* mendekati nol ketika diturunkan kembali dalam banyak proses. Ilustrasi *vanishing gradient* diberikan pada Gambar 11.14 (analogikan dengan *heat map*). Saat melakukan *backpropagation*, nilai gradien menjadi mendekati nol (warna semakin putih, delta nilai semakin menghilang). Penanganan permasalahan ini masih merupakan topik riset tersendiri. Sebagai contoh, pada arsitektur *recurrent neural network*, biasanya digunakan fungsi aktivasi *long short term memory* (LSTM) atau *gated recurrent unit* (GRU) untuk menangani *vanishing gradient problem*. Selain

[14] https://gist.github.com/yusugomori/cf7bce19b8e16d57488a

nilai gradien, nilai *synapse weights* juga bisa sangat kecil atau sangat besar. Hal ini juga tidak baik!



Gambar 11.14: Ilustrasi *vanishing gradient problem*.

11.13 Rangkuman

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

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

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

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

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

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

Neural network adalah salah satu *learning machine* yang dapat menemukan *hidden structure* atau pola data "implisit". Secara umum, *learning machine*

tipe ini sering menjadi *overfitting/overtraining*, yaitu model memiliki kinerja sangat baik pada *training data*, tapi buruk pada *testing data/unseen example*. Oleh sebab itu, menggunakan *neural network* harus hati-hati.

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

Soal Latihan

11.1. Turunan

(a) Turunkanlah perubahan *noise/bias* untuk *training* pada MLP.
(b) Turunkanlah proses *training deep neural network* pada Gambar 11.12 termasuk perubahan *noise/bias*.

11.2. Neural Network Training

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

11.3. Regularization Technique

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

11.4. Softmax Function

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

11.5. Transformasi atribut

Secara alamiah *neural network* membutuhkan data dengan atribut numerik untuk klasifikasi. Jelaskan konversi/strategi penanganan atribut nominal pada *neural network*!