Chapter 2:
Multilayer Artificial Neural Networks (ANN)

March 18, 2023

Seperti yang telah diketahui, ***Deep Learning* (DL)** telah menarik banyak perhatian dan sekarang merupakan topik paling hangat pada *Machine Learning* (ML). DL dapat diartikan sebuah kumpulan algoritma yang dibangun untuk melatih ***Artificial Neural Network* (ANN)** dengan banyak layer (*multilayer*) secara efisien. Pada bab ini, kita akan mempelajari konsep dasar dari ANN sehingga lebih siap untuk mengikuti materi pada bab-bab selanjutnya, yang akan memperkenalkan DL berbasis librari-librari `Python` dan menjelaskan arsitektur-arsitektur ***Deep Neural Networks* (DNN)**.

Topik-topik yang akan dibahas pada bab ini diantaranya adalah sebagai berikut:

- Memahami konsep untuk ***Multilayer Neural Networks***
- Mengimplementasikan algoritma *backpropagation* untuk men-training NN dari awal (*from scratch*)
- Men-training *multilayer neural network* untuk klasifikasi image

1 Memodelkan Fungsi Kompleks dengan ANN

Pada Bab 1, kita telah mempelajari algoritma ML dengan *artificial neuron* (AN). AN merepresentasikan blok-blok bangunan dari *multilayer* ANN yang akan didiskusikan pada Bab 2 ini. Konsep dasar dari ANN dibangun berdasarkan hipotesis dan model bagaimana otak manusia memecahkan masalah-masalah kompleks. Meskipun ANN sudah populer saat ini, studi paling awal dari *neural network* sudah dilakukan tahun 1940 oleh Warren McCulloch dan Walter Pitt.

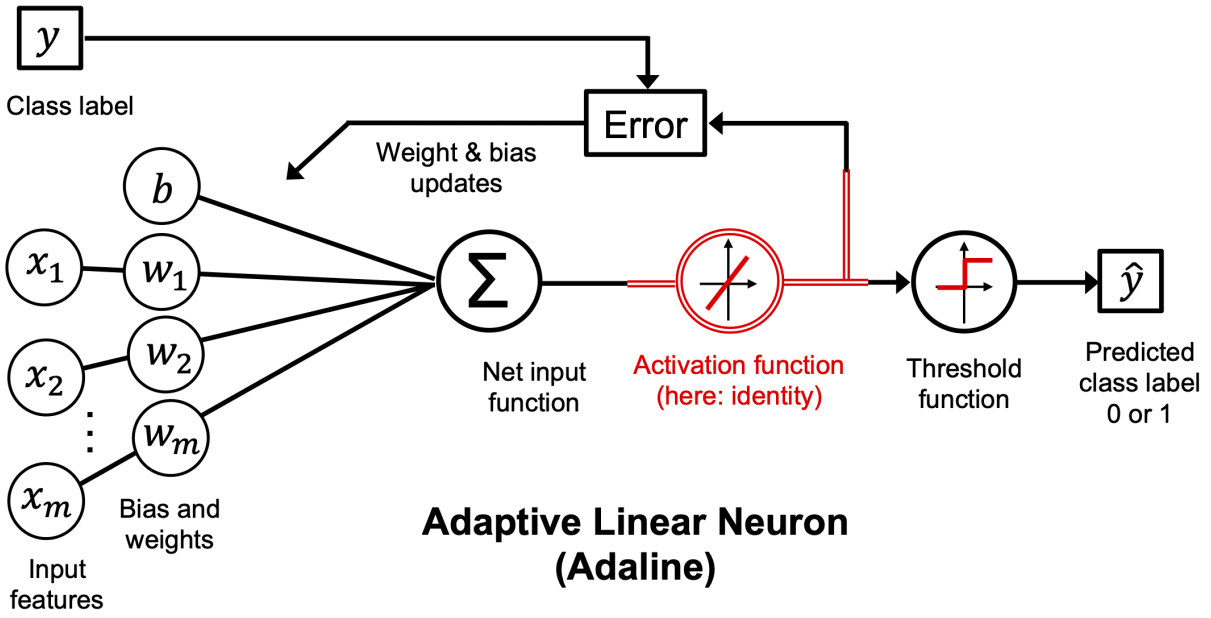
Pada dekade selanjutnya setelah model McCulloch-Pitt dan model perceptron Rosenblatt, banyak peneliti dan praktisi ML mulai kehilangan ketertarikan pada *neural networks*, karena tidak adanya solusi yang baik untuk melakukan training *neural network* dengan *multilayer*. Pada tahun 1986, ketertarikan pada *neural networks* kembali hidup, setelah D.E. Rumelhart, G.E. Hinton dan R.J. Williams menemukan dan mempopulerkan algoritma *backpropagation* untuk men-training *neural networks* secara lebih efisien, yang akan kita diskusikan pada bab ini (*Learning representations by back-propagating errors, David E. Rumelhart, Geoffrey E. Hinton, Ronald J. Williams, Nature, 323 (6088):533-536, 1986*). Pembaca yang tertarik pada sejarah dari AI, ML, dan *neural networks*, disarankan untuk membaca artikel wikipedia mengenai AI Winter, yang merupakan periode waktu dimana sebagian besar masyarakat peneliti kehilangan ketertarikan untuk mempelajari *neural network*.

Tetapi, *neural networks* di jaman sekarang sudah semakin populer. Banyak terobosan-terobosan

baru yang telah dibuat pada dekade-dekade sebelumnya yang pada akhirnya menghasilkan algoritma-algoritma dan arsitektur-arsitektur DL (*neural network* yang terdiri dari banyak layer). *Neural network* telah menjadi topik hangat bukan hanya di dunia peneliti dan akademik, tetapi juga di perusahaan-perusahaan besar seperti `Facebook`, `Microsoft`, dan `Google`, yang telah berinvestasi sangat besar pada penelitian ANN dan DL. Sekarang *neural networks* kompleks yang diperkuat oleh algoritma-algoritma DL dipandang sebagai *state of the art* ketika berhubungan dengan pemecahan masalah kompleks seperti *image* dan *voice recognition*. Contoh produk populer yang sering kita pakai di kehidupan sehari-hari yang didukung oleh algoritma-algoritma DL diantaranya adalah `Google`*'s image search* dan `Google`*'s Translate*.

1.1 *Neural Network* dengan Satu Layer (*recap*)

Seperti yang telah dijelaskan pada Bab 1, salah satu contoh *neural network* dengan satu layer adalah algoritma Adaline, dengan blok implementasi terlihat pada Gambar 2.1



Gambar 2.1: Blok implementasi algoritma Adaline

Pada Bab 1, kita sudah mengimplementasikan algoritma Adaline untuk melakukan klasifikasi biner, dan telah kita gunakan algoritma optimasi *gradient descent* untuk *learning* koefisien-koefisien pembobot dari model. Pada setiap epoch, update untuk vektor pembobot $\mathbf{w}$ dilakukan dengan aturan,

$$\mathbf{w} = \mathbf{w} + \Delta w, \quad \text{dimana} \quad \Delta w = -\eta \nabla J(\mathbf{w})$$

Dengan kata lain, kita menghitung *gradient* berdasarkan keseluruhan training set dan melakukan update pembobot-pembobot dari model dengan mengambil langkah pada arah yang berlawanan dengan *gradient* $\nabla J(\mathbf{w})$. Untuk mendapatkan pembobot optimal, *Sum of Squared Errors* (SSE) $\nabla J(\mathbf{w})$ didefinisikan sebagai fungsi objektif atau *cost function* yang diminimalkan. Lebih jauh, kita kalikan *gradient* dengan sebuah faktor, yaitu *learning rate* η, yang kita pilih secara hati-hati

untuk menyeimbangkan kecepatan *learning* dengan kemungkinan terjadi *overshooting* pada saat pencarian minimum global dari *cost function*.

Pada optimisasi *gradient descent*, kita melakukan update semua pembobot secara simultan di setiap *epoch* dan kita mendefinisikan turunan parsial dari setiap pembobot w_j pada vektor pembobot $\mathbf{w}$ sebagai berikut

$$\frac{\partial J(\mathbf{w})}{\partial w_j} = -\sum_i \left(y^{(i)} - a^{(i)}\right) x_j^{(i)}$$

Di sini, $y^{(i)}$ adalah label *class* target dari sebuah sampel training tertentu $x^{(i)}$, dan $a^{(i)}$ adalah fungsi aktivasi dari neuron, yang merupakan fungsi linier (khusus untuk Adaline). Lebih jauh, kita mendefinisikan fungsi aktivasi $\phi(\cdot)$ sebagai berikut:

$$\phi(z) = z = a$$

Net input z adalah kombinasi linier dari pembobot-pembobot yang mengkoneksikan layer input terhadap layer output,

$$z = \sum_j w_j x_j = \mathbf{w}^T \mathbf{x}$$

Sementara kita menggunakan fungsi aktivasi $\phi(z)$ untuk menghitung update *gradient*, kita menerapkan sebuah fungsi *threshold* untuk memeras harga output kontinyu ke dalam label *class* biner untuk prediksi,

$$\hat{y} \begin{cases} 1 \text{ if } g(z) \geq 0 \\ -1, \text{ otherwise} \end{cases}$$

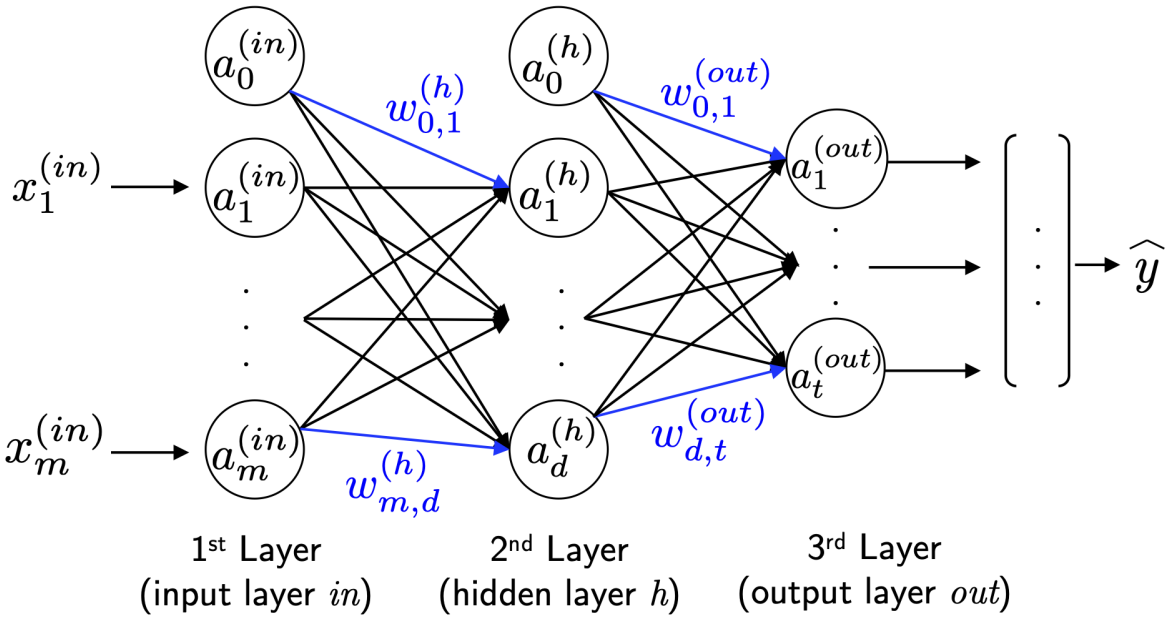
Sebagai catatan meskipun Adaline terdiri dari dua layer, satu layer input dan satu layer output, Adaline masih disebut satu layer karena keberadaan satu link antara layer input dan layer output.

Kita juga sudah mempelajari tentang trik tertentu untuk mempercepat pembelajaran model, yang disebut dengan ***stochastic gradient descent*** optimization. *Stochastic gradient descent* mengaproksimasi *cost* hanya dari satu sampel training (*online learning*) atau hanya sebagian kecil sampel training (*mini-batch learning*). Kita akan menggunakan konsep ini selanjutnya di bab ketika kita mengimplementasikan dan men-training ***multilayer perceptron*** **(MLP)**. Selain proses *learning* yang cepat (karena update yang lebih sering dibanding *gradient descent*), keberadaan noise alamiah juga bisa dianggap sebagai keuntungan ketika men-training *multilayer neural networks* dengan fungsi aktivasi non-linier, yang tidak mempunyai *cost function* yang convex. Di sini, noise dapat menolong untuk keluar dari minimum lokal, dengan detail penjelasan akan disampaikan kemudian di bab ini juga.

1.2 Pengenalan Arsitektur *Multilayer Neural Network*

Pada bagian ini, kita akan mempelajari bagimana mengkoneksikan neuron-neuron tunggal (*single layer neuron*) sehingga membentuk *multilayer feedforward neural netwoks*. Tipe khusus dari ***fully***

connected network disebut juga sebagai MLP. Gambar 2.2 mengilustrasikan konsep MLP yang terdiri dari 3 layer.



Gambar 2.2: Konsep multilayer perceptron (MLP) dengan 3 layer

MLP yang diperlihatkan pada Gambar 2.2 mempunyai satu layer input, satu layer tersembunyi (*hidden layer*) dan satu layer output. Unit-unit pada *hidden layer* terkoneksi seluruhnya (*fully connected*) ke layer input, dan layer output terkoneksi seluruhnya pada *hidden layer*. Jika network semacam ini mempunyai lebih dari satu *hidden layer*, maka kita sebut dengan ***deep artificial NN***.

Catatan Tambahan 2.1. Menambah *hidden layer*

Kita dapat menambah lebih banyak *hidden layer* pada MLP untuk menghasilkan arsitektur network yang lebih dalam. Secara praktis, kita dapat memikirkan sejumlah layer-layer dan unit-unit pada NN sebagai *hyperparameter* tambahan yang ingin kita optimasi untuk permasalahan yang diberikan. Tetapi, *gradient-gradient* error yang akan kita hitung selanjutnya menggunakan algoritma *backpropagation* akan semakin mengecil seiring dengan penambahan jumlah layer pada network. Permasalahan *vanishing gradient* membuat *learning* model menjadi lebih menantang. Oleh karena itu, algoritma-algoritma yang khusus dibangun untuk menolong training seperti struktur DNN, disebut dengan **Pembelajaran Dalam (*Deep Learning*)**

Seperti yang ditunjukan pada Gambar 2.2, untuk unit ke-i pada layer ke-l akan disimbolkan dengan variabel $a_i^{(l)}$. Untuk membuat permasalahan matematikanya dan implementasi kode-nya lebih sedikit intuitif, kita tidak akan menggunakan indeks numerik dalam menyatakan layer. Kita akan gunakan *superscript in* untuk menyatakan layer input, *superscript h* untuk *hidden layer* dan *superscript out* untuk output layer. Misalkan, $a_i^{(in)}$ menyatakan harga ke-i dari layer input, $a_i^{(h)}$ harga ke-i dari *hidden layer* dan $a_i^{(out)}$ menyatakan harga ke-i dari layer output. Unit aktivasi $a_0^{(in)}$ dan $a_0^{(h)}$ adalah unit-unit bias yang diset sama dengan 1. Aktivasi dari unit-unit pada layer input

hanya input-nya itu sendiri ditambah dengan unit bias,

$$\mathbf{a}^{(in)} = \begin{bmatrix} a_0^{(in)} \\ a_1^{(in)} \\ \vdots \\ a_m^{(in)} \end{bmatrix} = \begin{bmatrix} 1 \\ x_1^{(in)} \\ \vdots \\ x_m^{(in)} \end{bmatrix}$$

Catatan Tambahan 2.2. Konvensi Notasi untuk Unit-Unit Bias

Kita akan mengimplementasikan MLP pada bab ini di bagian selanjutnya dengan menggunakan vektor-vektor bias yang terpisah, untuk membuat implementasi kode lebih efisien dan mudah untuk dibaca. Konsep ini juga digunakan pada `Tensor Flow` (*deep learning library*) yang akan dibahas pada Bab 3. Tetapi, persamaan matematis akan terlihat lebih kompleks jika seandainya kita harus bekerja dengan variabel tambahan untuk bias. Komputasi dengan memberikan nilai 1 pada vektor input (seperti terlihat sebelumnya) dan menggunakan variabel pembobot sebagai bias adalah sama dengan operasi menggunakan vektro bias terpisah. Hal ini hanya konvensi semata.

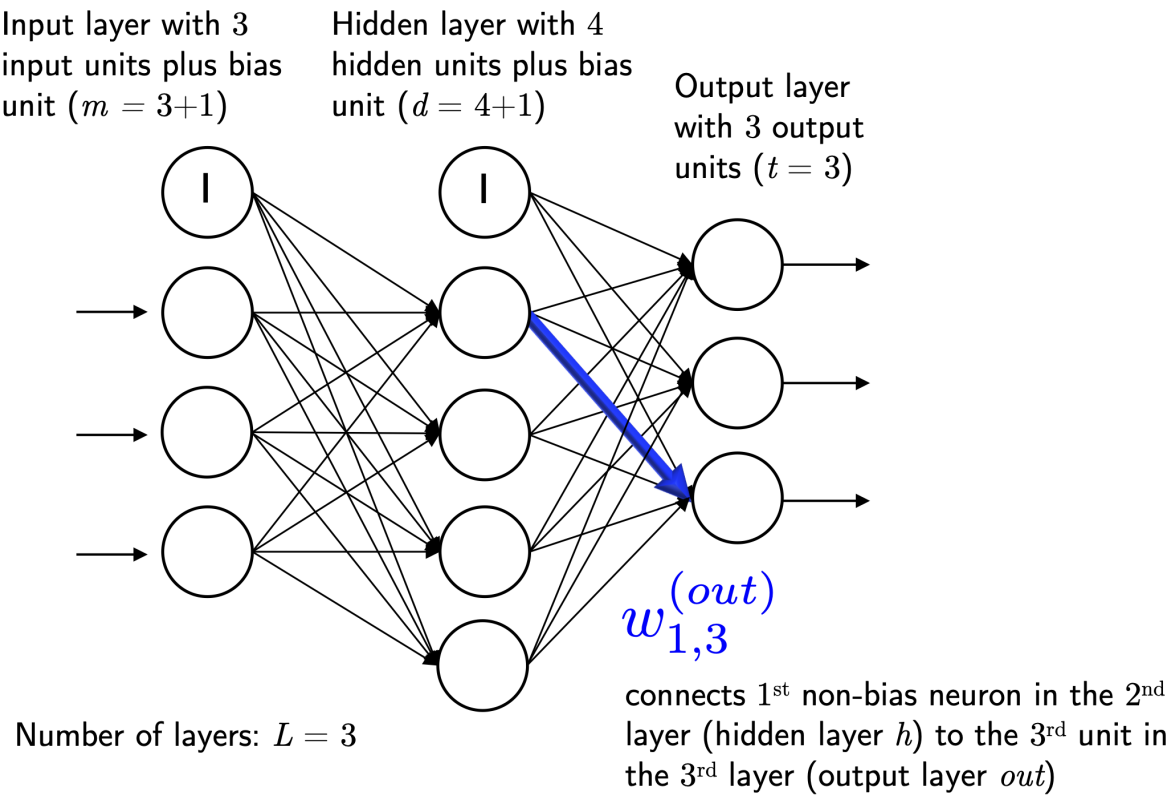
Setiap unit pada layer ke-l terkoneksi dengan semua unit-unit pada layer ke-$l+1$ melalui koefisien-koefisien pembobot. Sebagai contoh, koneksi antara unit ke-k pada layer l dan unit ke-j pada layer ke-$l+1$ dinyatakan dengan $w_{k,j}^{(l)}$. Dari Gambar 2.2, kita merepresentasikan matriks pembobot yang mengkoneksikan layer input ke *hidden layer* sebagai $\mathbf{W}^{(h)}$, dan matriks pembobot yang mengkoneksikan *hidden layer* ke layer output sebagai $\mathbf{W}^{(out)}$

Kendatipun satu unit pada layer output akan cukup untuk klasifikasi biner, tetapi kita dapat melakukan klasifikasi *multiclass* melalui generalisasi menggunakan teknik ***one-versus-all*** **(OvA)** (diajarkan pada MK Pembelajaran Mesin dan Aplikasi). Untuk memberikan pemahaman lebih jauh, kita bisa gunakan representasi ***one-hot*** untuk variabel kategorial. Misalkan, label 3 *class* pada dataset Iris (0 = Sentosa, 1 = Versicolor, 2 = Virginica) direpresentasikan dengan

$$0 = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \quad 1 = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, \quad 2 = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}.$$

Dengan *one-hot vector* ini, kita bisa menyelesaikan persoalan klasifikasi dengan jumlah label *class* sembarang yang ada pada dataset training.

Untuk seseorang yang baru dengan representasi NN, notasi indeks (*superscript* dan *subscript*) awal mulanya kadang membingungkan, tetapi akan lebih masuk akal pada bagian selanjutnya ketika kita sudah merepresentasikan NN dengan vektor. Seperti yang telah dikenalkan sebelumnya, kita meringkaskan bobot-bobot yang menghubungkan layer input dan *hidden* layer sebagai matriks $\mathbf{W}^{(h)} \in \mathbb{R}^{m \times d}$, dimana d adalah jumlah unit-unit *hidden* dan m adalah jumlah unit input termasuk unit bias. Karena notasi-notasi sangat penting, Gambar 2.3 meringkas apa-apa yang telah kita pelajari dalam bentuk ilustrasi deskriptif dari MLP $3-4-3$.



Gambar 2.3: Ilustrasi MLP $3 - 4 - 3$

1.3 Aktivasi *Neural Network* dengan *Forward Propagation*

Bagian ini akan mendeskripsikan proses dari ***forward propagation*** untuk menghitung output dari model *Multilayer Perceptron* (MLP). Untuk memahami bagaimana *forward propagation* berada pada konteks *learning* sebuah model MLP, prosedur *learning* dalam MLP dapat disingkat sebagai berikut:

1. Mulai pada layer input, pola data training diteruskan melalui network untuk menghasilkan output
2. Berdasarkan output network, error yang ingin diminimalkan menggunakan *cost function* kemudian dihitung. Hal ini akan dijelaskan kemudian
3. Error dipropagasi balik (*backpropagete*) dimana turunannya terhadap masing-masing bobot pada network dihitung dan kemudian model diupdate

Setelah kita melakukan pengulangan tiga langkah di atas untuk banyak *epoch* dan mempelajari bobot-bobot dari MLP, kemudian kita menggunakan *forward propagation* untuk menghitung output network dan mengimplementasikan fungsi *threshold* untuk menghasilkan prediksi label *class* dengan representasi *one-hot* yang telah dijelaskan pada bagian sebelumnya.

Sekarang, kita akan mempelajari masing-masing langkah dari *forward propagation* untuk menghasilkan output dari pola-pola data training. Karena setiap unit pada *hidden layer* dihubungkan dengan semua unit-unit pada layer input, pertama kita menghitung unit aktivasi pada *hidden layer*

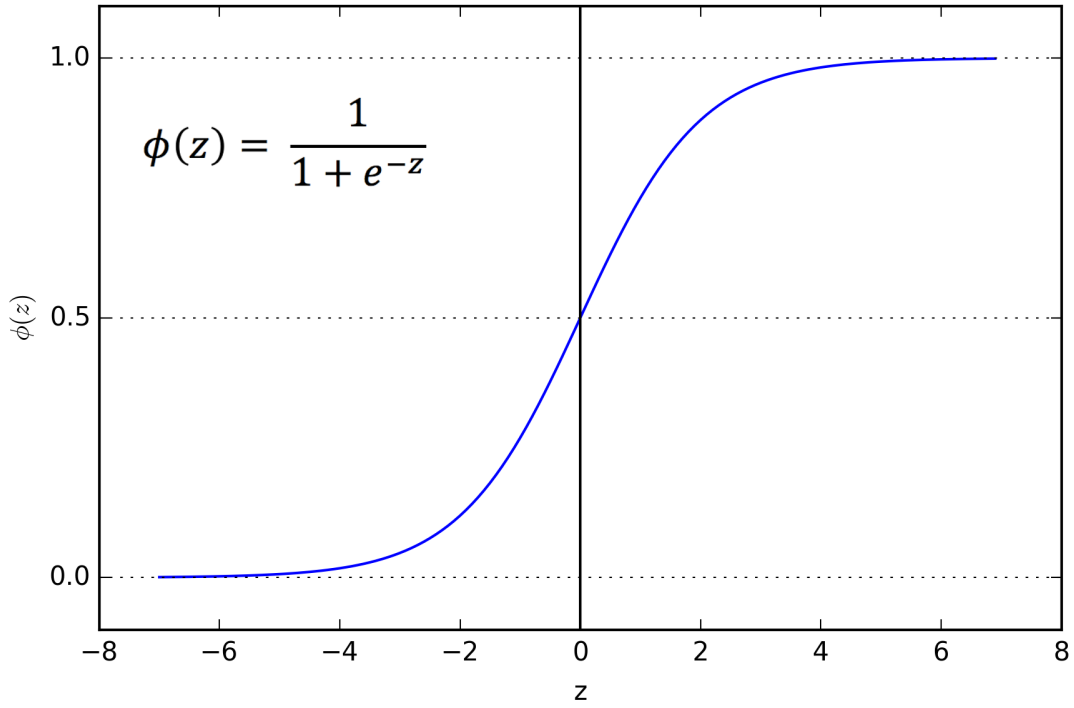
$a_1^{(h)}$ sebagai berikut

$$z_1^{(h)} = a_0^{(in)} w_{0,1}^{(h)} + a_1^{(in)} w_{1,1}^{(h)} + \cdots + a_m^{(in)} w_{m,1}^{(h)}$$
$$a_1^{(h)} = \phi\left(z_1^{(h)}\right)$$

Dimana, $z_1^{(h)}$ adalah net input dan $\phi(\cdot)$ adalah fungsi aktivasi yang dapat diturunkan (*differentiable*) untuk mempelajari bobot-bobot yang menghubungkan neuron-neuron menggunakan pendekatan *gradient-based*. Untuk memecahkan masalah kompleks seperti klasifikasi image, maka dibutuhkan fungsi-fungsi aktivasi yag non-linier pada model MLP seperti fungsi aktivasi sigmoid (logistik) seperti yang telah dijelaskan pada bagian regresi logistik (mata kuliah Pembelajaran Mesin Lanjut dan Aplikasi),

$$\phi(z) = \frac{1}{1 + e^{-z}}$$

Fungsi sigmoid merupakan kurva berbentuk $\mathcal{S}$ yang memetakan net input z pada distribusi logistik dengan nilai pada range 0 ke 1 dan memotong sumbu y pada $z = 0$, seperti yang ditunjukan pada Gambar 2.4.



Gambar 2.4: Fungsi aktivasi sigmoid

MLP merupakan contoh yang tipikal dari *feedforward* ANN. Istilah *feedforward* mengacu pada kenyataan bahwa setiap layer berlaku sebagai input untuk layer selanjutnya tanpa *loop* (berkebalikan dengan *recurrent NN*, yang akan dijelaskan secara detail pada bab selanjutnya). Istilah

Multilayer Perceptron mungkin agak sedikit membingungkan karena *artificial neurons* pada arsitektur network ini merupakan unit-unit sigmoid, bukan perceptron. Kita dapat melihat neuron di MLP sebagai unit-unit regresi logistik yang menghasilkan harga-harga kontinyu pada range 0 dan 1.

Dengan tujuan efisiensi kode dan kemudahan untuk dibaca, kita akan menuliskan aktivasi dalam bentuk yang lebih padu menggunakan konsep Aljabar Linier. Hal ini akan memudahkan kita untuk implementasi kode dalam bentuk vektor menggunakan `NumPy` daripada harus menuliskan dalam bentuk iterasi menggunakan loop `for`,

$$\mathbf{z}^{(h)} = \mathbf{a}^{(in)}\mathbf{W}^{(h)}$$
$$\mathbf{a}^{(h)} = \phi\left(\mathbf{z}^{(h)}\right)$$

Di sini, $\mathbf{a}^{(in)}$ merupakan vektor fitur berdimensi $1 \times m$ dari sampel $\mathbf{x}^{(in)}$ plus sebuah unit bias. $\mathbf{W}^{(h)}$ adalah matriks pembobot berdimensi $m \times d$, dimana d adalah jumlah unit-unit pada *hidden layer*. Setelah perkalian matriks, kita akan memperoleh vektor net input $\mathbf{z}^{(h)}$ untuk menghitung fungsi aktivasi $\mathbf{a}^{(h)}$ (dimana $\mathbf{a}^{(h)} \in \mathbb{R}^{1\times d}$).

Lebih jauh, kita dapat menggeneralisasi perhitungan ini untuk semua n sample-sample pada dataset training,

$$\mathbf{Z}^{(h)} = \mathbf{A}^{(in)}\mathbf{W}^{(h)},$$

Dimana $\mathbf{A}^{(in)}$ sekarang adalah matriks $n \times m$, dan hasil perkalian matriks adalah matriks net input $\mathbf{Z}^{(h)}$ berdimensi $n \times d$. Kemudian, kita mengaplikasikan fungsi aktivasi $\phi(\cdot)$ pada setiap harga dari matriks net input untuk mendapatkan matriks aktivasi berdimensi $n \times d$

$$\mathbf{A}^{(h)} = \phi\left(\mathbf{z}^{(h)}\right).$$

Selanjutnya, kita dapat menuliskan aktivasi layer output dalam bentuk vektor untuk banyak sampel sebagai berikut

$$\mathbf{Z}^{(out)} = \mathbf{A}^{(h)}\mathbf{W}^{(out)},$$

di sini kita mengalikan matriks $\mathbf{W}^{(out)}$ yang berdimensi $d \times t$ (t adalah jumlah unit output) dengan matriks $\mathbf{A}^{(h)}$ berdimensi $n \times d$ untuk mendapatkan matriks $\mathbf{Z}^{(out)}$ yang berdimensi $n \times t$ (kolom-kolom pada matriks ini adalah output-output untuk masing-masing sampel).

Pada bagian akhir, kita mengimplementasikan fungsi aktivasi sigmoid untuk memperoleh output berharga kontinyu dari network kita,

$$\mathbf{A}^{(out)} = \phi\left(\mathbf{z}^{(out)}\right), \quad \mathbf{A}^{(out)} \in \mathbb{R}^{n\times t}.$$

2 Training sebuah *Artificial Neural Network* (ANN)

Pada bagian ini kita akan mempelajari beberapa konsep secara lebih dalam, seperti *cost function* logistik dan algoritma *backpropagation* yang kita implementasikan untuk mempelajari bobot-bobot.

2.1 Menghitung *Cost Function* Logistik

Cost function logistik yang kita implementasikan sebetulnya cukup sederhana untuk dimengerti karena merupakan *cost function* yang sama dengan yang telah kita pelajari di regresi logistik. *Cost function* tersebut dapat dituliskan sebagai persamaan berikut,

$$J(\mathbf{w}) = -\sum_{i=1}^{n} y^{[i]} \log\left(a^{[i]}\right) + \left(1 - y^{[i]}\right) \log\left(1 - a^{[i]}\right)$$

Dimana $a^{[i]}$ adalah aktivasi sigmoid dari sampel ke-i di dataset, yang kita hitung pada langkah *forward propagation*

$$a^{[i]} = \phi\left(z^{[i]}\right)$$

Sebagai catatan superscript $[i]$ di sini menyatakan indeks dari sampel training dan bukan layer.

Untuk mengurangi *overfitting* kita akan menambahkan regularisasi $L2$ yang didefinisikan sebagai berikut

$$L2 = \lambda \|\mathbf{w}\|_2^2 = \lambda \sum_{j=1}^{m} w_j^2$$

Dengan menambahkan regularisasi $L2$ pada *cost function* logistik, kita akan peroleh persamaan berikut

$$J(\mathbf{w}) = -\sum_{i=1}^{n} y^{[i]} \log\left(a^{[i]}\right) + \left(1 - y^{[i]}\right) \log\left(1 - a^{[i]}\right) + \frac{\lambda}{2} \|\mathbf{w}\|_2^2$$

Sebelumnya kita mengimplementasikan MLP untuk klasifikasi multiclass yang menghasilkan vektor luaran dengan sejumlah t elemen yang kita butuhkan untuk dibandingkan dengan vektor target berdimensi $t \times 1$ dalam bentuk representasi *one-hot encoding*. Jika kita memprediksi label *class* dari input sebuah image dengan label *class* 2 menggunakan MLP, aktivasi dari layer ketiga dan target akan terlihat serupa dengan

$$a^{(out)} = \begin{bmatrix} 0,1 \\ 0,9 \\ \vdots \\ 0,3 \end{bmatrix}, \quad y = \begin{bmatrix} 0 \\ 1 \\ \vdots \\ 0 \end{bmatrix}$$

Dengan demikian, kita perlu melakukan generalisasi *cost function* logistik keseluruh unit aktivasi yang berjumlah t pada network.

Cost function tanpa Regularisasi dapat dituliskan kembali menjadi

$$J(\mathbf{W}) = -\sum_{i=1}^{n}\sum_{j=1}^{t} y_j^{[i]} \log\left(a_j^{[i]}\right) + \left(1 - y_j^{[i]}\right) \log\left(1 - a_j^{[i]}\right)$$

Di sini, superscript $[i]$ adalah indeks dari sampel tertentu dari dataset training yang dimiliki.

Bagian regularisasi yang lebih umum berikut tampak sangat kompleks untuk pertama kali, tapi di sini kita hanya menjumlahkan semua pembobot dari sebuah layer ke-l (tanpa bagian bias) yang kita tambahkan pada kolom pertama:

$$J(\mathbf{W}) = -\sum_{i=1}^{n}\sum_{j=1}^{t} y_j^{[i]} \log\left(a_j^{[i]}\right) + \left(1 - y_j^{[i]}\right) \log\left(1 - a_j^{[i]}\right) + \frac{\lambda}{2}\sum_{l=1}^{L-1}\sum_{i=1}^{u_l}\sum_{j=1}^{u_{l+1}}\left(w_{j,l}^{(l)}\right)^2$$

Di sini u_l menyatakan jumlah unit-unit dalam sebuah layer l, dan ekspresi berikut menyatakan bagian penalti (regularisasi)

$$\frac{\lambda}{2}\sum_{l=1}^{L-1}\sum_{i=1}^{u_l}\sum_{j=1}^{u_{l+1}}\left(w_{j,l}^{(l)}\right)^2$$

Tujuan kita adalah untuk meminimumkan *cost function* $J(\mathbf{W})$ dengan cara menurunkan parameter $\mathbf{W}$ terhadap masing-masing bobot untuk setiap layer pada network,

$$\frac{\partial}{\partial w_{j,i}^{(l)}} J\left(\mathbf{w}\right)$$

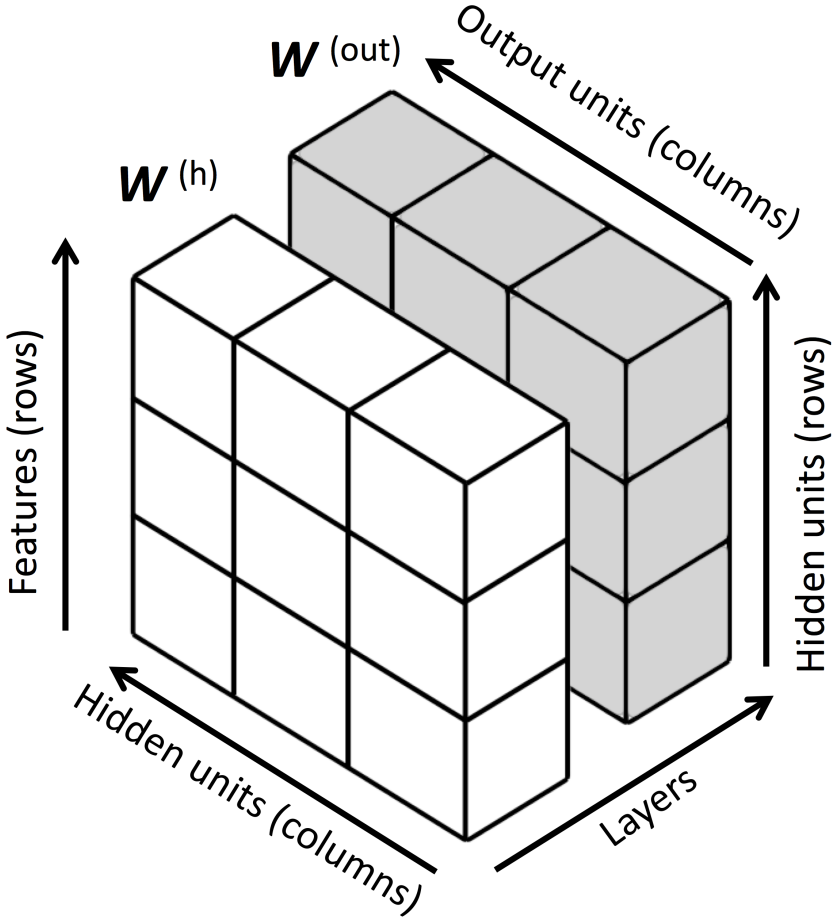
Pada bagian selanjutnya, kita akan membicarakan algoritma ***backpropagation*** yang bisa gunakan untuk menghitung turunan parsial dalam meminimumkan *cost function*.

Sebagai catatan, $\mathbf{W}$ terdiri dari banyak matriks-matriks. Pada MLP dengan satu *hidden layer*, maka $\mathbf{W}$ terdiri dari matriks pembobot $\mathbf{W}^{(h)}$ yang menghubungkan layer input ke *hidden layer* dan matriks pembobot $\mathbf{W}^{(out)}$ yang menghubungkan *hidden layer* terhadap layer output. Visualisasi dari tensor tiga dimensi $\mathbf{W}$ dapat dilihat pada Gambar 2.5.

Pada gambar yang disederhanakan ini, $\mathbf{W}^{(h)}$ dan $\mathbf{W}^{(out)}$ terlihat seperti mempunyai jumlah baris dan kolom yang sama, dimana hal ini jarang terjadi jika kita tidak menginisialisasi MLP dengan jumlah unit *hidden*, output dan fitur input dengan jumlah yang sama. Bagian selanjutnya akan membicarakan dimensi dari $\mathbf{W}^{(h)}$ dan $\mathbf{W}^{(out)}$ lebih detail pada konteks algoritma *backpropagation*.

2.2 Intuisi untuk Algoritma *Backpropagation*

Meskipun algoritma *backpropagation* dipopulerkan lebih dari 30 tahun yang lalu (*Learning representations by back-propagating errors, D.E. Rumelhart, G.E. Hinton, and R.J. Williams, Nature*, 323:6088, pages 533-536, 1986), tetapi algoritma ini merupakan satu algoritma yang masih banyak digunakan untuk men-training ANN secara efisien. Bagian ini akan menjelaskan secara singkat dan gambaran besar bagaiaman algoritma ini bekerja, sebelum membahas konsep matematis secara lebih detail.



Gambar 2.5: Visualisasi tensor tiga dimensi **W**

Pada prinsipnya *backpropagation* merupakan pendekatan komputasi yang sangat efisien untuk menghitung turunan parsial dari *cost function* yang kompleks pada multilayer ANN. Tujuannya adalah menggunakan turunan tersebut untuk mempelajari (*learn*) koefisien-koefisien pembobot sebagai parameter pada multilayer ANN tersebut. Tantangan pada multilayer ANN adalah keharusan berhubungan dengan jumlah koefisien-koefisien pembobot yang sangat besar pada dimensi ruang fitur yang tinggi. Berkebalikan dengan *cost function* pada layer tunggal seperti Adaline atau regresi logistik, error surface pada *cost function* multilayer ANN ini tidak convex atau *smooth* sebagai fungsi parameter. Terdapat banyak minimum lokal yang harus ditanggulangi untuk menemukan minimum global dari *cost function*.

Konsep *chain rule* dari kelas kalkulus dasar akan digunakan. *Chain rule* merupakan pendekatan perhitungan turunan fungsi kompleks yang bersarang (*nested function*), seperti $f\left(g(x)\right)$ sebagai berikut

$$\frac{d}{dx}\left[f(g(x))\right] = \frac{df}{dg} \cdot \frac{dg}{dx}$$

Dengan cara serupa, kita dapat menggunakan *chain rule* untuk komposisi fungsi yang panjang dan sembarang. Sebagai contoh, jika kita mempunyai lima fungsi yang berbeda,

$f(x)$, $g(x)$, $h(x)$, $u(x)$, dan $v(x)$, kemudian F menyatakan fungsi komposisi: $F(x) = f(g(h(u(v(x)))))$. Dengan mengaplikasikan *chain rule*, kita dapat menghitung turunan fungsi ini sebagai berikut

$$\frac{dF}{dx} = \frac{d}{dx}F(x) = \frac{d}{dx}f(g(h(u(v(x)))) = \frac{df}{dg} \cdot \frac{dg}{dh} \cdot \frac{dh}{du} \cdot \frac{du}{dv} \cdot \frac{dv}{dx}$$

Pada konteks aljabar komputasi, teknik-teknik telah dikembangkan untuk memecahkan permasalahan turunan di atas, yang dikenal dengan nama ***automatic differentiation***. *Automatic differentiation* mempunyai dua mode, mode *forward* dan *reverse*, dan *backpropagation* merupakan kasus khusus dari mode *reverse*. Pengaplikasian *chain rule* dengan mode *forward* bisa sangat mahal karena harus mengalikan banyak matriks untuk setiap layer (Jacobians) kemudian terakhir mengalikan dengan sebuah vektor untuk mendapatkan output. Berbeda dengan mode *reverse*, kita memulai dari kanan ke kiri, yaitu mengalikan matriks dengan sebuah vektor akan menghasilkan vektor, yang kemudian dikalikan kembali dengan sebuah matriks, dan demikian seterusnya. Perkalian matriks dan vektor seperti pada *backpropagation* secara komputasi lebih murah dibandingkan dengan perkalian matriks dan matriks. Hal inilah yang menyebabkan *backpropagation* menjadi salah satu algoritma paling populer pada training neural networks.

2.3 Training Neural Networks via *Backpropagation*

Pada bagian ini, matematika untuk *backpropagation* dijelaskan untuk dapat memahami bagaimana bobot dipelajari pada neural networks secara efisien. Tergantung pada pemahaman matematika sebelumnya, representasi matematika berikut boleh jadi terasa sulit pada saat pertama kali ditelusuri.

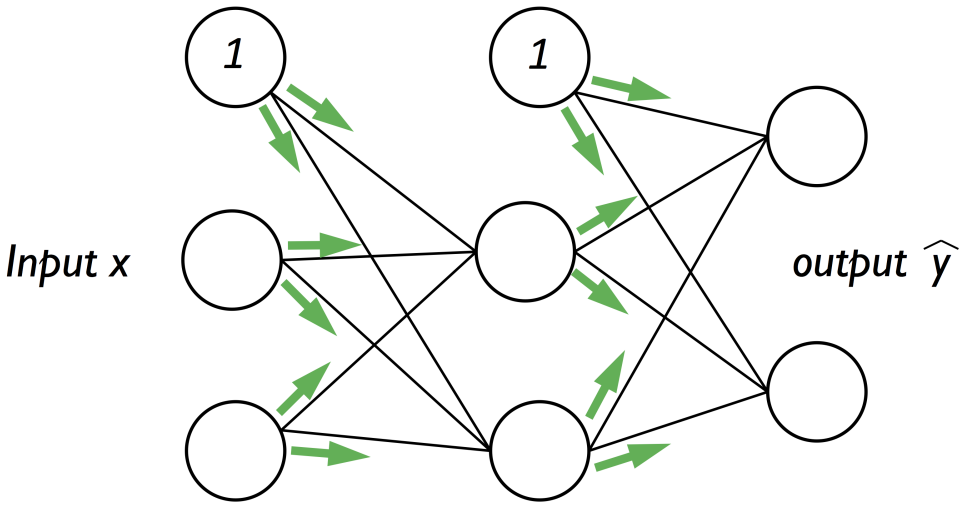
Pada sub-bab sebelumnya sudah diperlihatkan bagaimana menghitung *cost* sebagai perbedaan antara aktivasi pada layer terakhir dan label *target class*. Sekarang akan diperlihatkan bagaimana algoritma *backpropagation* bekerja untuk update pembobot pada model MLP dari perspektif matematika. Seperti yang telah disampaikan sebelumnya, pertama kali kita harus implementasi *forward propagation* untuk menghasilkan aktivasi pada layer output yang dapat diformulasikan sebagai berikut:

$$\begin{aligned}
\mathbf{Z}^{(h)} &= \mathbf{A}^{(in)}\mathbf{W}^{(h)} && \text{(net input dari hidden layer)} \\
\mathbf{A}^{(h)} &= \phi\left(\mathbf{Z}^{(h)}\right) && \text{(aktivasi dari hidden layer)} \\
\mathbf{Z}^{(out)} &= \mathbf{A}^{(h)}\mathbf{W}^{(out)} && \text{(net input dari layer output)} \\
\mathbf{A}^{(out)} &= \phi\left(\mathbf{Z}^{(out)}\right) && \text{(aktivasi dari layer output)}
\end{aligned}$$

Secara ringkas, kita hanya melakukan *forward-propagate* fitur-fitur input melalui koneksi pada network, seperti yang diilustrasikan pada Gambar 2.6.

Pada *backpropagation*, error berpropagasi dari kanan ke kiri, dimulai dengan menghitung vektor error dari layer output

$$\boldsymbol{\delta}^{(out)} = \mathbf{a}^{(out)} - \mathbf{y}.$$



Gambar 2.6: Ilustrasi forward propagate

Dimana **y** adalah vektor dari label *true class*. Kemudian kita menghitung error pada *hidden layer*

$$\delta^{(h)} = \delta^{(h)} \left(\mathbf{W}^{(out)}\right)^T \odot \frac{\partial \phi\left(z^{(h)}\right)}{\partial z^{(h)}}$$

Pada persamaan di atas $\frac{\partial \phi\left(z^{(h)}\right)}{\partial z^{(h)}}$ menyatakan turunan parsial dari fungsi aktivasi sigmoid

$$\frac{\partial \phi\left(z^{(h)}\right)}{\partial z^{(h)}} = \left(a^{(h)} \odot \left(1 - a^{(h)}\right)\right)$$

Sedangkan simbol $\odot$ pada konteks ini menyatakan perkalian antar elemen (*element-wise*).

Catatan Tambahan 2.1. Penurunan parsial dari fungsi aktivasi

Untuk orang yang mempunyai rasa ingin tahu yang tinggi berikut adalah penurunan parsial dari fungsi aktivasi.

$$
\begin{aligned}
\frac{\partial \phi(z)}{\partial z} &= \frac{\partial}{\partial z}\left(\frac{1}{1+e^{-z}}\right) \\
&= \frac{e^{-z}}{(1+e^{-z})^2} \\
&= \frac{1+e^{-z}}{(1+e^{-z})^2} - \left(\frac{1}{1+e^{-z}}\right)^2 \\
&= \frac{1}{(1+e^{-z})} - \left(\frac{1}{1+e^{-z}}\right)^2 \\
&= \phi(z) - (\phi(z))^2 \\
&= \phi(z)\,(1-\phi(z)) \\
&= a(1-a)
\end{aligned}
$$

Selanjutnya, kita hitung $\delta^{(h)}$ yang merupakan error matriks *hidden layer* dengan persamaan berikut

$$
\delta^{(h)} = \delta^{(out)}\left(\mathbf{W}^{(out)}\right)^T \odot \left(a^{(h)} \odot \left(1-a^{(h)}\right)\right)
$$

Penjelasan lebih detail untuk lebih mengerti bagaimana menghitung $\delta^{(h)}$ adalah sebagai berikut. Pada persamaan di atas kita menggunakan transpose $\left(\mathbf{W}^{(out)}\right)^T$ dari matriks $\left(\mathbf{W}^{(out)}\right)$ berdimensi $n \times t$, dimana t adalah jumlah label *class* output dan h adalah jumlah *hidden units*. Perkalian antara matriks $\delta^{(out)}$ berdimensi $n \times t$ dan matriks $\left(\mathbf{W}^{(out)}\right)^T$ berdimensi $t \times h$ menghasilkan sebuah matriks berdimensi $n \times h$ yang kemudian dikalikan per-elemen (*element-wise*) dengan turunan sigmoid dengan ukuran yang sama untuk menghasilkan matriks $\delta^{(h)}$ berdimensi $n \times h$.

Akhirnya setelah mendapatkan bagian δ, turunan dari *cost function* dapat dituliskan sebagai berikut

$$
\begin{aligned}
\frac{\partial}{\partial w_{i,j}^{(out)}} J(\mathbf{W}) &= a_j^{(h)} \delta_i^{(out)} \\
\frac{\partial}{\partial w_{i,j}^{(h)}} J(\mathbf{W}) &= a_j^{(in)} \delta_i^{(h)}
\end{aligned}
$$

Selanjutnya, kita perlu mengakumulasikan turunan parsial dari setiap node pada masing-masing layer dan error dari node pada layer selanjutnya. Tetapi ingat bahwa kita perlu menghitung $\Delta_{i,j}^{(i)}$ untuk setiap sampel pada dataset training. Oleh sebab itu, lebih mudah untuk mengimplementasikan dalam bentuk vektor sebagai berikut

$$
\begin{aligned}
\Delta^{(h)} &= \left(\mathbf{A}^{(in)}\right)^T \delta^{(h)} \\
\Delta^{(out)} &= \left(\mathbf{A}^{(h)}\right)^T \delta^{(out)}
\end{aligned}
$$

Dan setelah mengakumulasikan turunan parsial, bagian regularisasi untuk mengurangi *overfitting* dapat ditambahkan sebagai berikut

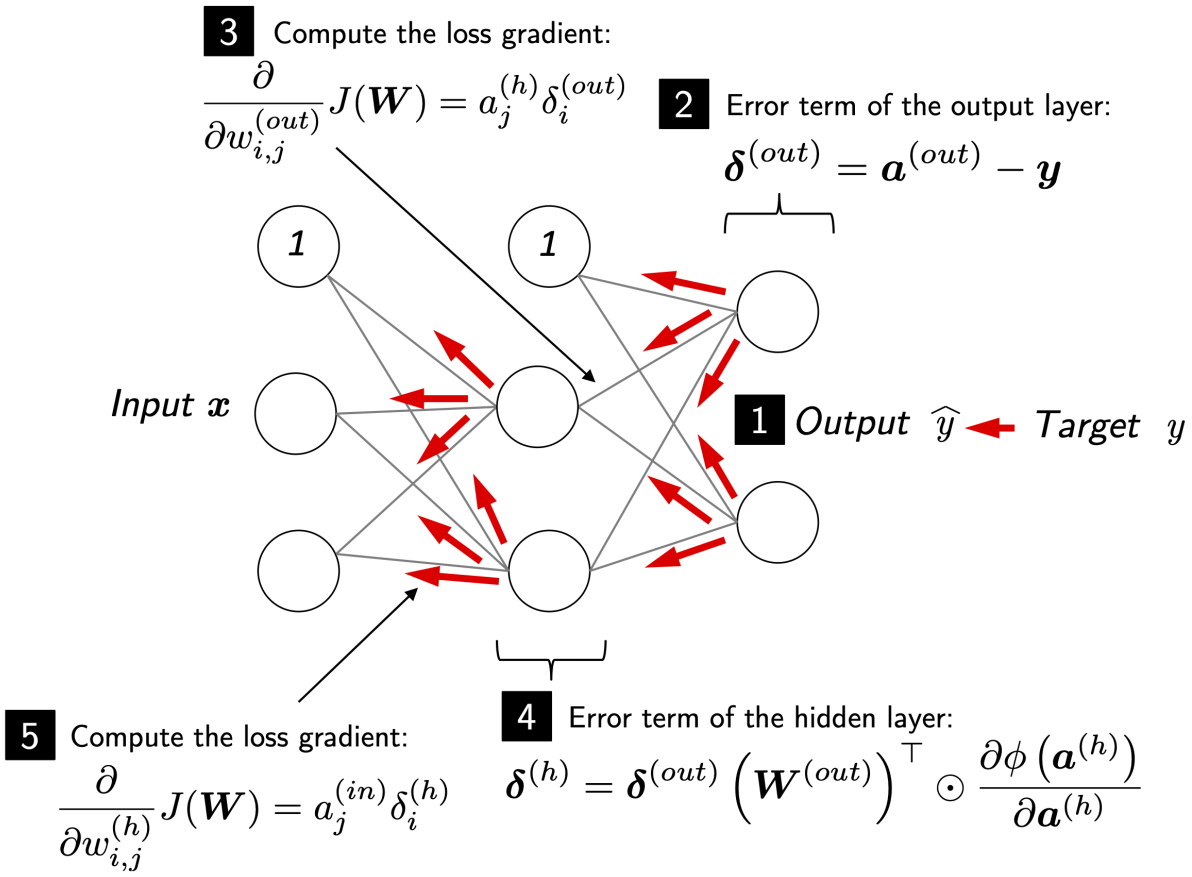
$$\Delta^{(l)} := \Delta^{(l)} + \lambda^{(l)} \mathbf{W}^{(l)}$$

Sebagai catatan unit-unit dari bias biasanya tidak dimasukan ke dalam regularisasi.

Terakhir, setelah gradien-gradien dihitung, kita dapat melakukan update dari pembobot-pembobot dengan mengambil langkah berlawanan arah dengan gradien untuk setiap layer l

$$\mathbf{W}^{(l)} := \mathbf{W}^{(l)} - \eta \Delta^{(l)}$$

Gambar 2.7 berikut mengilustrasikan bagaimana algoritma *backpropagation* bekerja.



Gambar 2.7: Ilustrasi algoritma backpropagation