Chapter 1:
Artificial Neural Networks (ANN)

March 12, 2023

Pada bab ini, kita akan menggunakan dua algoritma contoh untuk klasifikasi, yaitu ***perceptron*** dan ***adaptive linear neurons***. Kita mulai dengan mengimplementasikan *perceptron* langkah demi langkah pada `Python` dan melakukan training untuk mengklasifikasikan spesies bunga-bunga yang berbeda dari dataset Iris. Hal ini akan menolong kita memahami konsep-konsep algoritma *machine learning*(ML) untuk klasifikasi dan bagaimana mereka dapat diimpementasikan secara efisien pada `Python`

Topik-topik yang akan didiskusikan pada bagian ini diantaranya adalah:

- Membangun intuisi untuk algoritma-algoritma ML
- Menggunakan `Pandas`, `NumPy`, dan `Matplotlib` untuk membaca, memproses dan memvisualisasikan data
- Mengimplementasikan algoritma klasifikasi linier pada `Python`

1 *Artificial Neurons*

Sebetulnya ANN telah ada sejak lama, diperkenalkan tahun 1943 oleh seorang ahli fisiologi saraf (*neurophysiologist*) Warren McCulloch dan seorang matematikawan Walter Pitts. Pada paper mereka "*A Logical Calculus of Ideas Immanent in Nervous Activity*", mereka mendeskripsikan bagaimana model komputasi sederhana dari neuron (saraf) biologis dapat bekerja pada otak hewan untuk melakukan komputasi kompleks menggunakan *propositional logic*. Ini merupakan arsitektur pertama dari *neural networks* (NN). Kemudian banyak arsitektur lain ditemukan setelahnya.

Kesuksesan pertama dari ANN telah membawa kepercayaan meluas bahwa tidak lama lagi kita dapat bercakap-cakap dengan mesin cerdas. Ketika diketahui secara jelas pada tahun 1960 bahwa janji ini tidak akan terpenuhi (setidaknya untuk sementara), banyak pendanaan yang akhirnya pindah ke tempat lain sehingga ANN mengalami kemandegan panjang. Pada awal tahun 80-an, arsitektur baru ditemukan dan teknik training yang lebih baik dibangun, sehingga membangkitkan kembali ketertarikan pada *connectionism* (studi menyangkut NN). Tetapi saat itu perkembangannya sangat lambat, dan pada tahun 90-an teknik *machine learning* lain yang powerful ditemukan, seperti *Support Vector Machine*. Teknik-teknik baru tersebut terasa menawarkan hasil lebih baik dan mempunyai pondasi teoritis yang lebih kuat dibandingkan dengan ANN, sehingga studi NN banyak ditinggalkan kembali.

Sekarang kita menyaksikan gelombang ketertarikan lain pada ANN. Apakah ketertarikan ini akan sirna kembali seperti sebelumnya? Beberapa hal berikut merupakan alasan yang kuat untuk

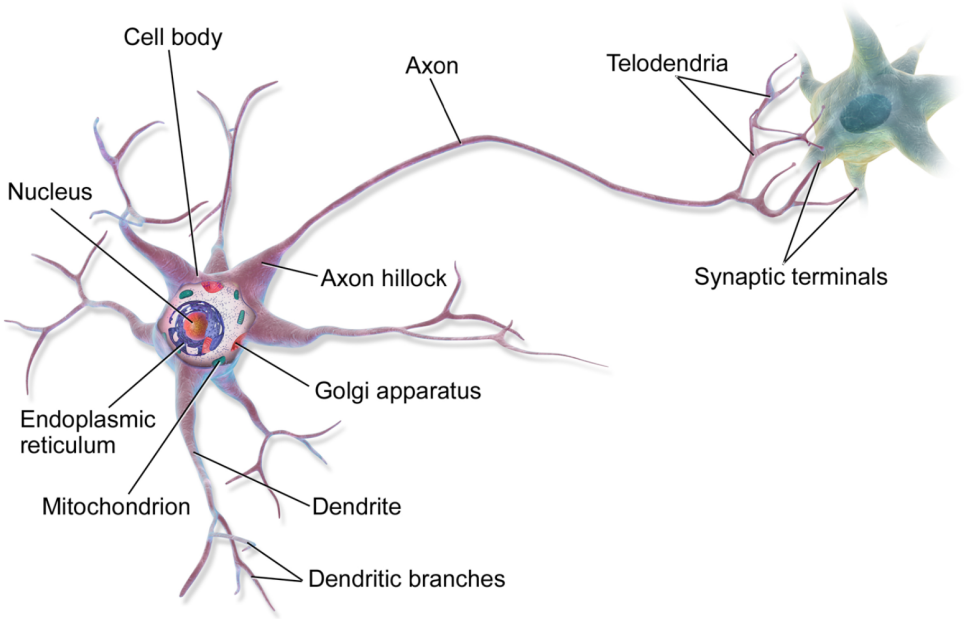
mempercayai bahwa kali ini akan berbeda dan ketertarikan kembali pada ANN akan membawa efek yang lebih besar terhadap kehidupan kita:

- Terdapat banyak data yang tersedia untuk mentraining NN, dan ANN sering mempunyai kinerja yang lebih baik dibandingkan teknik *machine learning* yang lain untuk masalah-masalah besar dan sangat kompleks.
- Peningkatan yang sangat besar pada kemampuan komputasi sejak tahun 90-an memungkinkan untuk melatih NN yang besar dengan lama waktu yang cukup masuk akal. Ini sebagian karena *Moore's law* (penambahan komponen pada IC menjadi dua kali lipat setiap dua tahun pada 50 tahun terakhir). Tetapi juga industri *gaming* yang telah merangsang produksi *Graphical Processing Unit* (GPU) secara besar-besaran. Lebih jauh, platform *Cloud* telah membuat kekuatan ini dapat diakses oleh semua orang.
- Algoritma-algoritma training telah banyak mengalami perbaikan. Sebetulnya, algoritma-algoritma ini hanya sedikit berbeda dengan yang telah digunakan pada tahun 90-an, tetapi dengan modifikasi sedikit telah membuat dampak yang sangat positif.
- Beberapa keterbatasan teoritis ANN ternyata bersifat ringan pada aspek praktis. Misalkan, orang berfikir bahwa algoritma training ANN berkinerja buruk karena kemungkinan besar berakhir pada optimum lokal. Tetapi pada kenyataannya bahwa kejadian ini jarang pada aspek praktis.
- ANN telah memasuki era perkembangan dan pendanaan yang terus menerus. Produk-produk yang menakjubkan berdasarkan ANN secara reguler telah membuat berita-berita utama, yang menarik perhatian dan pendanaan lebih banyak. Hal ini juga menyebabkan perkembangan lebih jauh untuk menghasilkan produk-produk yang lebih menakjubkan.

1.1 Neuron Biologis

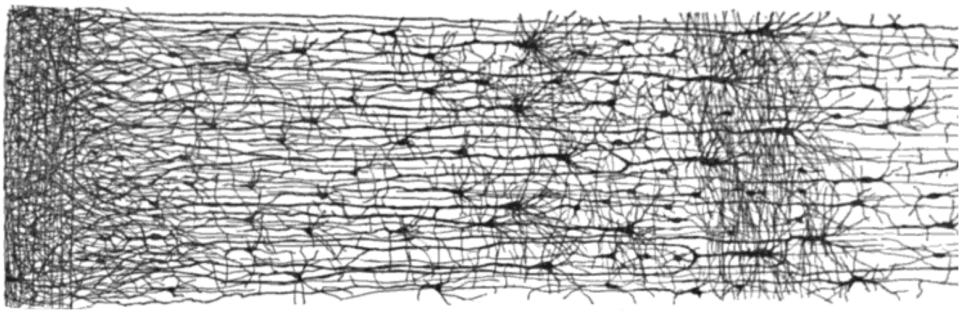
Sebelum kita mendiskusikan neuron artifisial, kita akan melihat sebentar pada neuron biologis yang direpresentasikan pada Gambar 1.1, yang merupakan sel yang terlihat tidak biasa pada otak hewan. Sel tersebut terdiri dari *cell body* yang berisi *nucleus* dan kebanyakan komponen-komponen sel yang kompleks, percabangan *dendrits*, ditambah dengan percabangan yang lebih panjang disebut *axon*. Panjang *axon* bisa jadi beberapa kali lebih panjang dibandingkan dengan *cell body*, atau bahkan sampai puluhan ribu kali lebih panjang. Dekat pada ujung, *axon* bercabang kembali yang disebut dengan *telodendria*, dan pada pucuk cabang-cabang ini terdapat struktur yang kecil disebut dengan *synaptic terminals* (atau *synapses*) yang terkoneksi lagi dengan *dendrit* atau *cell body* neuron-neuron lain. Neuron-neuron biologis ini menghasilkan impuls-impuls listrik pendek yang disebut dengan *action potentials* (AP, atau disebut sinyal) yang berjalan sepanjang *axon-axon* dan membuat *synapses* mengeluarkan sinyal-sinyal kimia disebut *neurotransmitters*. Ketika neuron menerima sejumlah *neurotransmitters* yang cukup dalam beberapa *milliseconds* (ms), neuron akan melepaskan impuls-impuls listrik sendiri (sebetulnya tergantung pada *neurotransmitters* sendiri, karena beberapa darinya akan menghalangi neuron untuk melepaskan impuls).

Neuron-neuron secara individu tampaknya berprilaku cukup sederhana, tetapi mereka diorganisir ke dalam jaringan yang sangat besar berjumlah miliaran, dengan masing-masing neuron biasanya terkoneksi dengan ribuan neuron lain. Komputasi yang sangat kompleks dapat dilakukan dengan sebuah jaring neuron-neuron yang cukup sederhana, seperti sarang semut yang dibangun dari kontribusi masing-masing semut kecil. Arsitektur dari *biological neural networks* (BNN) masih merupakan subjek dari riset yang aktif. Tetapi beberapa bagian dari otak telah dipetakan,



Gambar 1.1: Neuron Biologis

dan tampaknya neuron-neuron tersebut sering disusun pada lapisan-lapisan yang berurutan, terutama pada *cerebral cortex* (yaitu bagian luar dari otak kita), seperti yang ditunjukan Gambar 1.2.



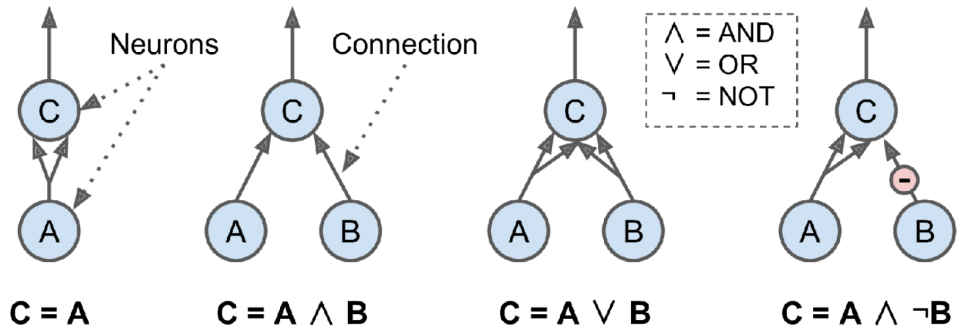
Gambar 1.2: Beberapa lapisan-lapisan pada BNN (cortex manusia)

1.2 Komputasi Logika dengan Neurons

McCulloch dan Pitss mengajukan model yang sangat sederhana dari neuron biologis yang kemudian dikenal dengan *artificial neuron* (AN). AN mempunyai satu atau lebih input biner (*on/off*) dan satu output biner. AN mengaktivasi outputnya ketika jumlah input aktif lebih besar jumlah tertentu. Pada papernya, mereka menunjukan bahwa bahkan dengan model sederhana seperti itu, memungkinkan untuk membuat jaringan AN yang menghitung operasi logika yang kita inginkan. Untuk melihat bagaimana jaringan bekerja, kita misalkan membuat sejumlah ANN yang mengoperasikan hitungan logika terentu (lihat Gambar 1.3, dengan asumsi **sebuah neuron teraktivasi ketika sedikitnya dua inputnya aktif**.

Cara kerja jaringan adalah sebagai berikut:

- Jaringan pertama sebelah kiri adalah fungsi identitas: Jika neuron A diaktivasi, kemudian



Gambar 1.3: ANN mengoperasikan komputasi logika sederhana

neuron C teraktivasi juga (karena menerima dua sinyal input dari neuron A). Tetapi jika neuron A *off* maka neuron C *off* juga.

- Jaringan kedua menunjukan logika AND: Neuron C diaktivasi ketika kedua neuron A dan B teraktivasi (hanya satu sinya input tidak cukup untuk mengaktivasi neuron C).
- Jaringan ketiga menunjukan logika OR: Neuron C teraktivasi jika neuron A atau B teraktivasi (atau keduanya).
- Akhirnya, jika kita misalkan sebuah koneksi input dapat menghalangi aktivitas neuron (yang memang terjadi pada neuron-neuron biologis), maka jaringan keempat menghitung operasi logika yang sedikit berbeda dan lebih kompleks: Neuron C teraktivasi hanya jika neuron A aktif dan neuron B *off*. Jika neuron A aktif sepanjang waktu, maka kita dapatkan operasi logika NOT, yaitu neuron C aktif ketika neuron B *off*, demikian sebaliknya.

1.3 Definisi Formal untuk *Artificial Neuron*

Secara lebih formal, kita bisa menempatkan ide *artificial neuron* pada konteks persoalan klasifikasi biner (*binary classification*) dimana terdapat dua *class* yaitu 1 (*class* positif) dan -1 (*class* negatif) sebagai penyederhanaan. Kemudian bisa didefinisikan sebuah fungsi keputusan (*decision function*) $\phi(z)$ yang merupakan kombinasi linier dari harga-harga input $\mathbf{x}$ dan pasangan vektor pembobot $\mathbf{w}$, dimana z disebut sebagai *net input* $z = w_1x_1 + \ldots + w_mx_m$:

$$\mathbf{w} = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_m \end{bmatrix}, \quad \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{bmatrix}.$$

Kemudian, jika *net input* dari sampel tertentu $\mathbf{x}^{(i)}$ lebih besar dari ambang batas θ yang telah didefinisikan sebelumnya, maka akan diprediksi sebagai *class* 1, atau jika sebaliknya maka akan diprediksi sebagai *class* -1.

Pada algoritma perceptron, fungsi keputusan $\phi(\cdot)$ merupakan varian dari sebuah **fungsi unit step (*unit step function*)**,

$$\phi(z) \begin{cases} 1 \text{ if } z \geq \theta \\ -1, \text{ otherwise} \end{cases}$$

Untuk membuat lebih sederhana, kita bisa membawa *threshold* θ ke bagian kiri dari persamaan dan mendefinisikan pembobot $w_0 = -\theta$ dan $x_0 = 1$, sehingga kita bisa menuliskan kembali z sebagai

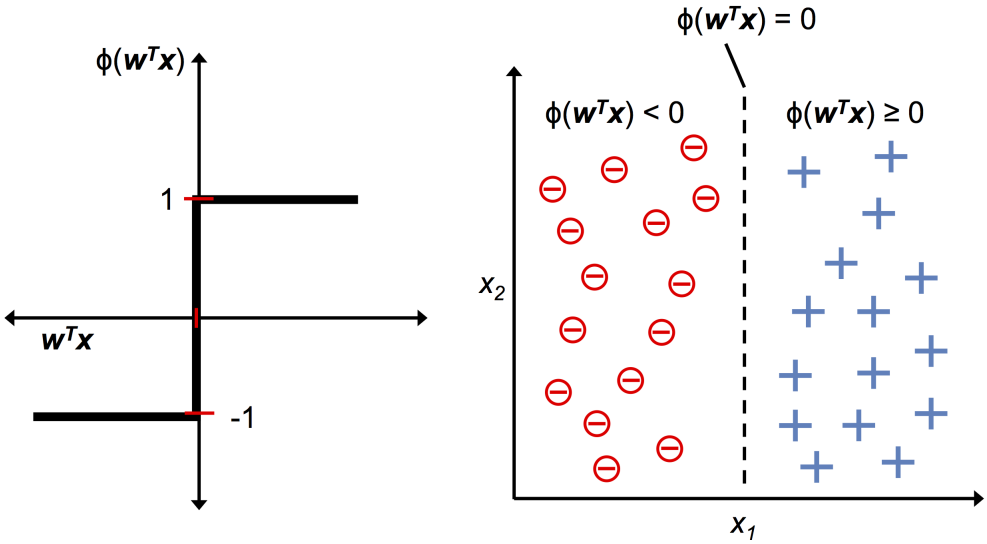
$$z = w_0 x_0 + w_1 x_1 + \ldots + w_m x_m = \mathbf{w}^T \mathbf{x}$$

dimana $(\cdot)^T$ menyatakan operasi transpose. Sehingga fungsi unit step dapat dituliskan kembali seperti pada Persamaan (1.1).

Persamaan (1.1). *Fungsi unit step*

$$\phi(z) \begin{cases} 1 \text{ if } z \geq 0 \\ -1, \text{ otherwise} \end{cases}.$$

Pada literatur ML, *threshold* negatif atau pembobot $w_0 = -\theta$ biasanya disebut dengan **bias unit**. Gambar 1.4 menunjukan bagaimana *net input* $z = \mathbf{w}^T \mathbf{x}$ diperas menjadi output biner (-1 atau 1) dengan menggunakan fungsi keputusan dari perceptron (gambar kiri), dan bagaimana digunakan untuk mendiskriminasi antara dua *class* yang terpisah secara linier, atau *linearly separable classes* (gambar kanan)



Gambar 1.4: Net input menjadi output biner (gambar kiri) dan fungsi keputusan dari perceptron (gambar kanan)

1.4 *Learning Rule* dari Perceptron

Ide dari neuron McCulloch-Pitts (MCP) dan model perceptron dengan *threshold* dari Rosenblatt adalah menggunakan pendekatan reduksionis (seseorang yang menganalisa dan menjabarkan fenomena kompleks dengan unsur-unsur fundamentalnya yang lebih sederhana) untuk menyerupai bagaimana sebuah neuron pada otak bekerja. Aturan perceptron awal dari Rosenblatt cukup sederhana dan dapat dituliskan sebagai berikut:

1. Insialisasi bobot-bobot (*weights*) $= 0$ atau bilangan acak kecil
2. Untuk tiap sampel training $\mathbf{x}^{(i)}$

a. hitung harga output $\hat{y}$

b. update bobot-bobot

Dalam langkah di atas, harga output adalah label *class* yang diprediksi oleh fungsi unit step, dan update masing-masing bobot w_j pada vektor pembobot **w** dapat dituliskan secara formal dengan

$$w_j := w_j + \Delta w_j$$

Harga Δw_j yang digunakan untuk update bobot w_j, dihitung berdasarkan ***Perceptron Learning Rule***.

Persamaan (1.2). *Perceptron Learning Rule*

$$\Delta w_j = \eta \left(y^{(i)} - \hat{y}^{(i)} \right) x_j^{(i)}$$

Dimana η adalah ***learning rate*** (biasanya sebuah konstan antara 0.0 dan 1.0), $y^{(i)}$ adalah label *class* yang nyata (***true class label***) dari sampel training ke-i, dan $\hat{y}^{(i)}$ adalah label *class* hasil prediksi (***predicted class label***). Hal yang perlu dicatat adalah semua bobot w_j pada vektor pembobot **w** di-update secara bersamaan, artinya kita tidak menghitung kembali label hasil prediksi $\hat{y}^{(i)}$ sebelum semua pembobot diupdate melalui masing-masing harga update yang bersesuaian Δw_j. Sebagai contoh, untuk dataset dua dimensi, update dapat ditulis sebagai berikut:

$$\begin{aligned}
\Delta w_0 &= \eta \left(y^{(i)} - \hat{y}^{(i)} \right) \\
\Delta w_1 &= \eta \left(y^{(i)} - \hat{y}^{(i)} \right) x_1^{(i)} \\
\Delta w_2 &= \eta \left(y^{(i)} - \hat{y}^{(i)} \right) x_2^{(i)}.
\end{aligned}$$

Sebelum kita mengimplementasikan *perceptron rule* pada `Python`, terlebih dahulu akan kita lihat contoh kecil untuk mengilustrasikan bagaimana sederhananya *learning rule* ini.

Pada dua skenario dimana perceptron memprediksi label *class* dengan betul, bobot-bobot tetap tidak berubah karena harga update sama dengan 0:

- $y^{(i)} = -1, \quad \hat{y}^{(i)} = -1, \quad \Delta w_j = \eta \left(-1 - (-1)\right) x_j^{(i)} = 0$
- $y^{(i)} = 1, \quad \hat{y}^{(i)} = 1, \quad \Delta w_j = \eta \left(1 - 1\right) x_j^{(i)} = 0.$

Tetapi, ketika terjadi prediksi label *class* yang salah, bobot-bobot akan didorong menuju target *class* positif atau negatif:

- $y^{(i)} = 1, \quad \hat{y}^{(i)} = -1, \quad \Delta w_j = \eta \left(1 - (-1)\right) x_j^{(i)} = \eta(2) x_j^{(i)}$
- $y^{(i)} = -1, \quad \hat{y}^{(i)} = 1, \quad \Delta w_j = \eta \left(-1 - 1\right) x_j^{(i)} = \eta(-2) x_j^{(i)}.$

Untuk memahami lebih jauh dari faktor pengali $x_j^{(i)}$, kita lihat contoh sederhana lain dimana kita mempunyai

$$\hat{y}^{(i)} = -1, \quad y^{(i)} = +1, \quad \eta = 1$$

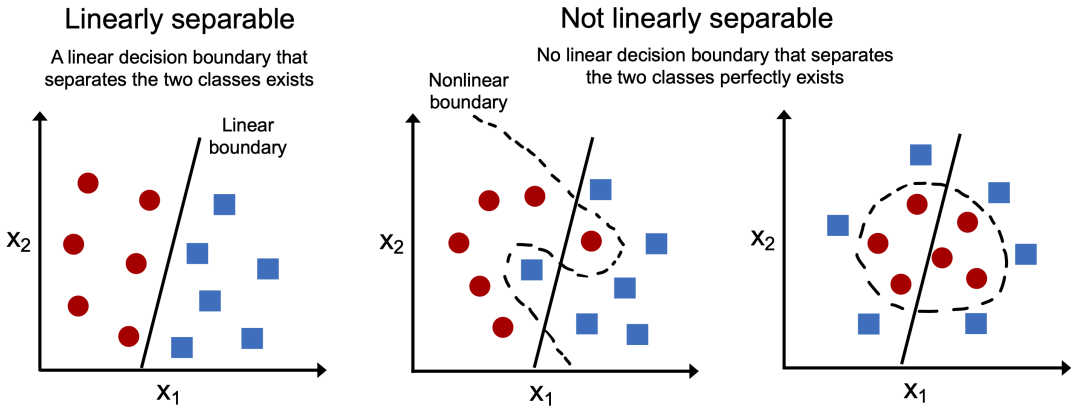
Kemudian diasumsikan bahwa $x_j^{(i)} = 0.5$, dan terjadi kesalahan klasifikasi (*misclassify*) sampel ini sebagai -1. Pada kasus ini, kita akan menambah bobotnya menjadi 1 sehingga *net input*, $x_j^{(i)} \times w_j$, menjadi lebih positif pada saat selanjutnya kita menemui sampel ini. Dengan demikian, lebih memunginkan memperoleh harga di atas *threshold* dari fungsi step sehingga mengklasifikasikan sampel ini sebagai $+1$:

$$\Delta w_j = (1 - (-1))0.5 = (2)0.5 = 1$$

Dengan demikian update pembobot sebanding dengan harga $x_j^{(i)}$. Sebagai contoh, jika kita mempunyai sampel lain $x_j^{(i)} = 2$ dan diklasifikasikan salah sebagai -1, kita akan mendorong batas keputusan (*decision boundary*) dengan harga yang lebih besar lagi untuk mengklasifikasikan sampel ini secara benar di waktu selanjutnya:

$$\Delta w_j = (1^{(i)} - (-1)^{(i)})2^{(i)} = (2)2^{(i)} = 4$$

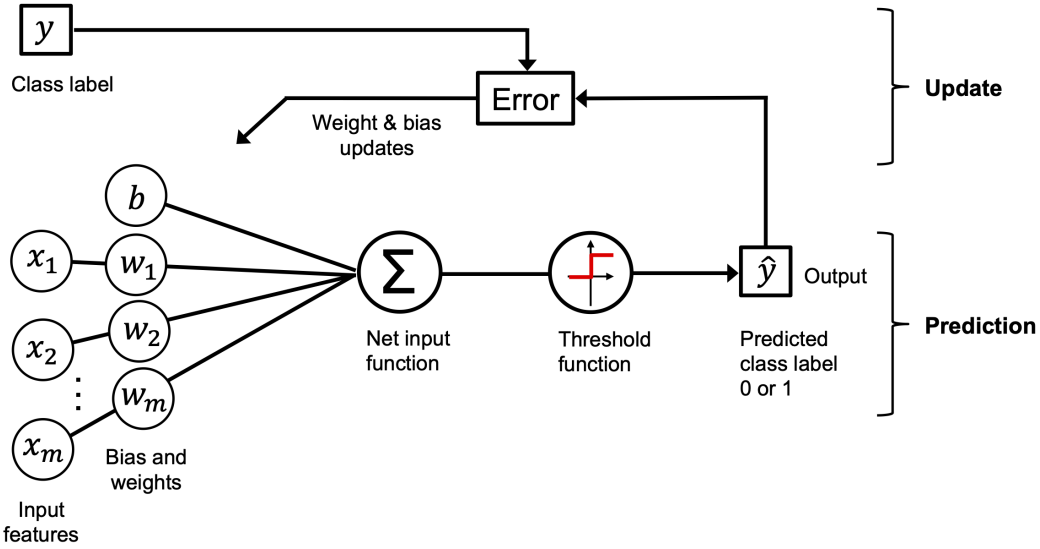
Sebagai catatan penting, konvergensi perceptron hanya dijamin jika kedua *class* adalah ***linearly separable*** dan *learning rate* cukup kecil. Jika kedua *class* tidak bisa dipisahkan dengan batas keputusan yang linier (*linear decision boundary*), kita dapat tentukan jumlah maksimum iterasi melalui dataset (***epochs***) dan atau menentukan sebuah *threshold* untuk jumlah mis-klasifikasi yang masih ditoleransi. Jika tidak maka perceptron tidak akan berhenti untuk melakukan update pembobot-pembobot. Ilustrasi kondisi *linearly separable* dan yang bukan dapat dilihat pada Gambar 1.5.



Gambar 1.5: Ilustrasi data-data training yang linearly separable dan yang bukan

Sebelum kita masuk ke bagian selanjutnya, apa yang telah dipelajari dapat disimpulkan ke dalam diagram sederhana pada Gambar 1.6 berikut, yang mengilustrasikan konsep perceptron secara umum.

Diagram pada Gambar 1.6 mengilustrasikan bagaimana perceptron menerima input dari sebuah sampel data $\mathbf{x}$, dan mengkombinasikannya dengan vektor pembobot $\mathbf{w}$ untuk menghitung *net input*. Kemudian *net input* tersebut dilewatkan ke dalam sebuah fungsi *threshold* yang akan membangkitkan output biner -1 atau 1 sebagai label *class* hasil prediksi dari sampel tersebut. Selama proses *learning*, output ini digunakan untuk menghitung error dari prediksi dan melakukan update pembobot-pembobot.



Gambar 1.6: Diagram konsep perceptron secara umum yang bukan

2 *Adaptive Linear Neurons* (Adaline) dan Konvergensi dari *Learning*

Pada bagian ini kita akan melihat tipe lain dari neural network satu layer, yaitu ***ADAptive LInear NEuron* (Adaline)**. Adaline dipublikasikan oleh Bernard Widrow dan mahasiswa doktoralnya Tedd Hoff, hanya beberapa tahun setelah algoritma perceptron dari Rosenblatt. Adaline bisa dianggap sebagai perbaikan dari percetron Rosenblatt. Referensi Adaline dapat dilihat di *An Adaptive "Adaline" Neuron Using Chemical "Memistors", Technical Report Number 1553-2, B. Widrow and others, Stanford Electron Labs, Stanford, CA, October 1960*.

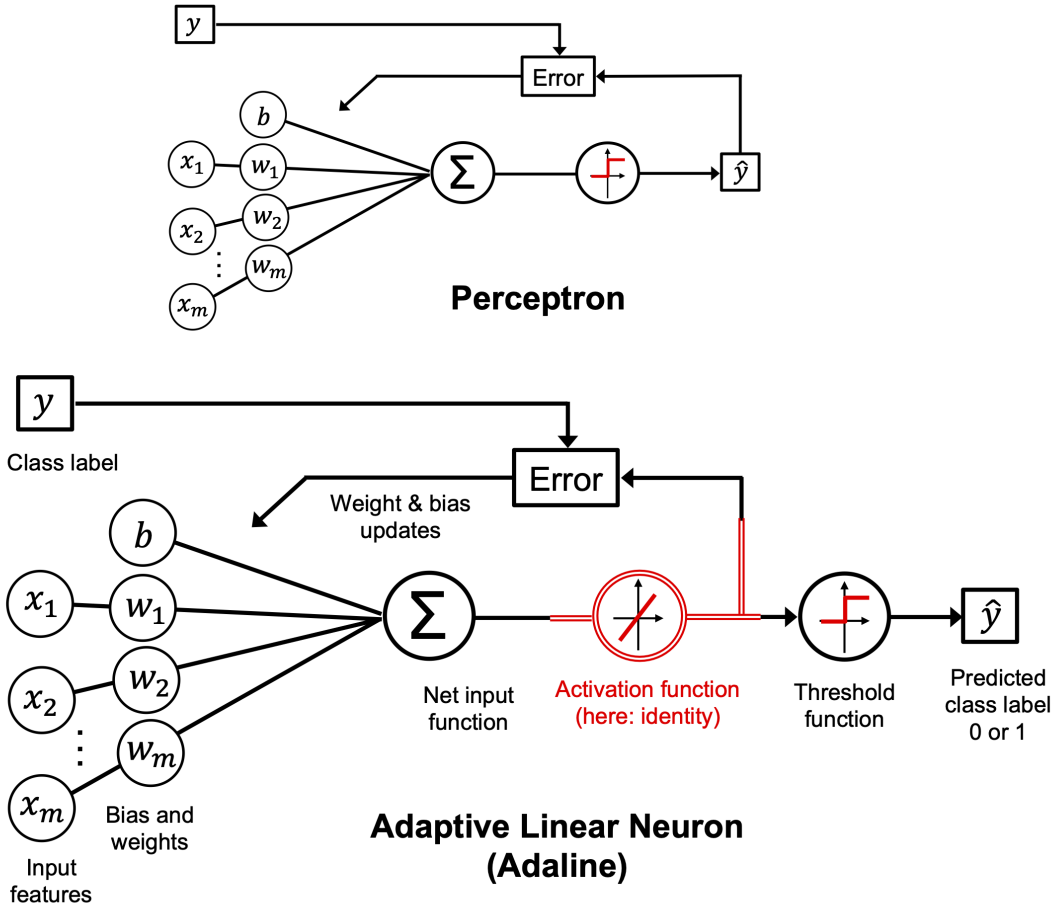
Algoritma Adaline cukup nemarik karena mengilustrasikan konsep kunci dalam mendefinisikan dan meminimalkan fungsi *cost* kontinyu. Hal ini meletakan dasar-dasar untuk memahami algoritma-algoritma lanjut dari ML baik klasifikasi (regresi logistik, support vector machine (SVM)) dan model regresi yang telah dibahas pada mata kuliah Pembelajaran Mesin dan Aplikasi.

Perbedaan utama antara *Adaline Rule* (atau disebut juga ***Widrow-Hoff*** rule) dan perceptron Rosenblatt adalah Adaline mengupdate bobot berdasarkan fungsi aktivasi linear, sedangkan perceptron meggunakan unit fungsi step. Pada Adaline, fungsi aktivasi ini $\phi(z)$ adalah fungsi identitas dari *net input*, sehingga

$$\phi\left(\mathbf{w}^T\mathbf{x}\right) = \mathbf{w}^T\mathbf{x}.$$

Dalam hal ini fungsi aktivasi linier digunakan untuk *learning* dari bobot, dan masih digunakan fungsi *threshold* untuk membuat prediksi akhir, serupa dengan fungsi unit step yang kita lihat sebelumnya. Perbedaan utama antara perceptron dan Adaline terlihat pada Gambar 1.7

Gambar 1.7 menunjukan bahwa algoritma Adaline membandingkan label *class* nyata dengan output kontinyu dari fungsi aktivasi linier untuk menghitung error model dan mengupdate bobot-bobot. Sedangkan, perceptron membandingkan label-label *class* nyata dengn label-label *class* hasil prediksi.



Gambar 1.7: Perbedaan utama perceptron dan Adaline

2.1 Meminimumkan *Cost Function* dengan *Gradient Descent*

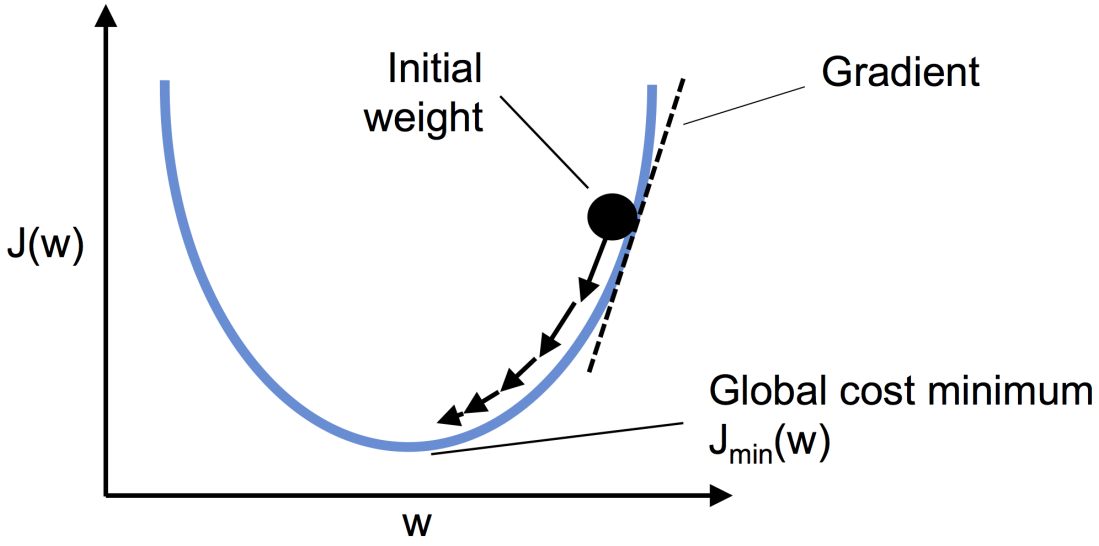
Salah satu unsur utama dari algoritma *supervised* ML adalah mendefinisikan fungsi objektif (***objective function***) yang akan dioptimasi selama proses *learning*. Fungsi objektif ini merupakan *cost function* yang ingin kita minimalkan. Pada Adaline, kita dapat mendefinisikan *cost function* J untuk *learning* pembobot-pembobot sebagai fungsi dari ***Sum of Squared Errors* (SSE)** antara luaran yang dihitung (*calculated outcome*) dan label *class* nyata:

$$J(\mathbf{w}) = \frac{1}{2}\sum_i \left(y^{(i)} - \phi\left(z^{(i)}\right)\right)^2$$

Unsur $\frac{1}{2}$ ditambahkan untuk memudahkan saat menderivasi gradien yang akan kita lihat di bagian selanjutnya. Kelebihan utama dari fungsi aktivasi linier kontinyu adalah *cost function* dapat diturunkan, sedangkan pada perceptron tidak. Hal lain yang menguntungkan adalah *cost function* bersifat ***Convex***. Sehingga, kita bisa menggunakan algoritma optimasi yang cukup powerful, disebut dengan ***gradient descent***, dalam menemukan pembobot-pembobot yang dapat meminimumkan *cost function* sehingga mampu mengklasifikasikan sampel-sampel pada sebuah dataset.

Ilustrasi dari *gradient descent* dapat dilihat pada Gambar 1.8, yang menggambarkan ide utama

dari *gradient descent* adalah seperti menuruni bukit sampai minimum *local* atau *global cost* dicapai. Pada setiap iterasi, kita mengambil langkah yang berlawanan dengan arah dari *gradient* dimana ukuran langkah ditentukan oleh *learning rate* dan juga kemiringan *gradient*.



Gambar 1.8: Ilustrasi gradient descent

Dengan menggunakan *gradient descent*, kita dapat mengupdate bobot-bobot dengan mengambil sebuah langkah dengan arah berlawanan terhadap *gradient* $\nabla J(\mathbf{w})$

$$\mathbf{w} := \mathbf{w} + \Delta w$$

Dimana perubahan pembobot Δw didefinisikan sebagai *gradient* negatif dikalikan dengan *learning rate* η,

$$\Delta w = -\eta \nabla J(\mathbf{w})$$

Untuk menghitung *gradient* dari *cost function*, kita perlu menghitung turunan parsial dari *cost function* terhadap masing-masing bobot w_j (lihat **Catatan Tambahan 1.1**)

$$\frac{\partial J}{\partial w_j} = -\sum_i \left(y^{(i)} - \phi\left(z^{(i)}\right) \right) x_j^{(i)}$$

Sehingga update dari pembobot w_j dapat dituliskan sebagai

$$\Delta w_j = -\eta \frac{\partial J}{\partial w_j} = \eta \sum_i \left(y^{(i)} - \phi\left(z^{(i)}\right) \right) x_j^{(i)}$$

Karena update dilakukan secara bersamaan untuk setiap pembobot, maka ***Adaline learning rule*** menjadi

$$\mathbf{w} = \mathbf{w} + \Delta w$$

Persamaan (1.3). *Adaline Learning Rule*

$\Delta w_j := \eta \sum_i \left(y^{(i)} - \phi\left(z^{(i)}\right)\right) x_j^{(i)}$.

Meskipun *learning rule* dari Adaline serupa dengan perceptron, tetapi harus diperhatikan bahwa $\phi\left(z^{(i)}\right)$ dengan $z^{(i)} = \mathbf{w}^T\mathbf{x}^{(i)}$ adalah bilangan real dan bukan label *class* bernilai integer. Lebih jauh, update dari pembobot dihitung berdasarkan semua sampel-sampel pada training set (bukan update pembobot dengan sampel satu persatu). Oleh sebab itu pendekatan pada Adaline ini disebut juga dengan ***batch gradient descent***.

Catatan Tambahan 1.1. Penurunan parsial dari *cost function* SSE

Berdasarkan Kalkulus, turunan parsial dari *cost function* SSE terhadap pembobot ke-j (w_j) dapat diperoleh dengan cara berikut

$$\frac{\partial J}{\partial w_j} = \frac{\partial}{\partial w_j}\frac{1}{2}\sum_i \left(y^{(i)} - \phi\left(z^{(i)}\right)\right)^2 \tag{1}$$

$$= \frac{1}{2}\frac{\partial}{\partial w_j}\sum_i \left(y^{(i)} - \phi\left(z^{(i)}\right)\right)^2 \tag{2}$$

$$= \frac{1}{2}\sum_i 2\left(y^{(i)} - \phi\left(z^{(i)}\right)\right)\frac{\partial}{\partial w_j}\left(y^{(i)} - \phi\left(z^{(i)}\right)\right) \tag{3}$$

$$= \sum_i \left(y^{(i)} - \phi\left(z^{(i)}\right)\right)\frac{\partial}{\partial w_j}\left(y^{(i)} - \sum_i \left(w_j^{(i)} x_j^{(i)}\right)\right) \tag{4}$$

$$= \sum_i \left(y^{(i)} - \phi\left(z^{(i)}\right)\right)\left(-x_j^{(i)}\right) \tag{5}$$

$$= -\sum_i \left(y^{(i)} - \phi\left(z^{(i)}\right)\right) x_j^{(i)} \tag{6}$$

2.2 *Stochastic Gradient Descent*

Pada sub-bab sebelumnya, kita telah mengetahui bagaimana meminimumkan *cost function* dengan mengambil langkah dengan arah yang berlawanan terhadap gradient yang dihitung berdasarkan keseluruhan data training atau disebut juga *batch gradient descent* (BGD). Dalam machine learning, sering kali kita harus bekerja dengan dataset yang sangat besar, dengan ukuran data bisa jutaan. Jika digunakan pada kondisi data yang sangat besar, maka BGD akan menghasilkan harga komputasi yang sangat besar, karena algoritma akan melakukan iterasi berkali-kali menggunakan keseluruhan dataset di setiap langkah untuk menuju minimum global.

Metode alternatif dari BGD yang cukup populer adalah *stochastic gradient descent* (SGD), yang sering juga disebut *iterative* atau *online gradient descent*. Jika pada BGD update pembobot dilakukan berdasarkan penjumlahan error akumulasi dari selurung data training, $\mathbf{x}^{(i)}$,

$$\Delta\mathbf{w} = \eta\sum_i \left(y^{(i)} - \phi\left(z^{(i)}\right)\right)\mathbf{x}^{(i)},$$

maka pada SGD, upadate pembobot dilakukan satu persatu untuk setiap sampel training, sehingga

$$\Delta \mathbf{w} = \eta \left(y^{(i)} - \phi \left(z^{(i)} \right) \right) \mathbf{x}^{(i)}.$$

Meskipun SGD bisa dilihat sebagai metode aproksimasi dari gradient descent, SGD konvergen lebih cepat untuk mencapai titip optimum (minimum). Selain itu, karena gradient dihitung berdasarkan satu sampel training maka permukaan error (*error surface*) mempunyai noise lebih besar dibandingkan BGD, SGD mempunyai kemungkinan lebih besar untuk keluar dari minimum lokal menuju minimum global. Untuk mendapatkan hasil SGD yang memuaskan, maka disarankan untuk memberikan data training secara random.

Kompromi antara BGD dan SGD dapat dilakukan, dimana kita bisa mengimplementasikan BGD dengan menggunakan subset data training lebih kecil (mini), misalkan 32 sampel training dalam satu waktu. Metode ini disebut dengan *mini-batch gradient descent* (MGD). Keuntungan MGD dibanding BDG adalah MGD mencapai konvergensi yang lebih cepat dibanding BGD, karena update pembobot yang lebih sering. Selain itu MGD bisa menggunakan operasi vektor sebagai pengganti `for` loop yang digunakan oleh SGD pada setiap data training, sehingga akan meningkatkan efisiensi komputasi dari algoritma learning.