Chapter 8:
Artificial Neural Networks (ANN)

December 6, 2020

Banyak hal di alam ini yang sudah menginspirasi penemuan-penemuan teknologi. Salah satunya adalah melihat arsitektur otak manusia sebagai inspirasi membuat mesin yang cerdas (*intelligent machine*). Logika inilah yang memicu *artificial neural networks* (ANN). ANN adalah model *machine learning* yang memperoleh inspirasi dari neuron biologis (*biological neuron*) yang ditemukan pada otak. Tetapi, ANN secara lambat laun menjadi cukup berbeda dengan neuron biologis tadi. Beberapa peneliti bahkan berargumen bahwa kita harus lepaskan analogi-anologi biologis secara bersamaan. Misalkan, dengan menyebut 'unit' sebagai pengganti 'neuron'. Kita dapat mengambil hal terbaik diantara dua pendapat yang ada, terbuka terhadap inspirasi biologis tanpa harus takut membuat model biologis yang tidak realistis, selama model-model tersebut dapat bekerja dengan baik.

ANN merupakan inti dari ***Deep Learning***. ANN sangat beragam, powerful, dan mudah untuk diskalakan, sehingga ANN sangat ideal untuk menghandel pekerjaan-pekerjaan *machine learning* yang besar dan sangat kompleks seperti klasifikasi miliaran gambar (mis. *Google Images*), memperkuat servis *speech recognition* (mis. Apple's Siri), merekomendasikan video terbaik untuk ditonton oleh ratusan juta orang setiap hari (mis. YouTube), atau berusaha mengalahkan juara dunia *Game of Go* (DeepMind's AlphaGo).

Pada *chapter* ini akan diperkenalkan ANN, dimulai dengan pembahasan sekilas mengenai arsitektur ANN paling awal dan dilanjutkan dengan *Multilayer Perceptrons* (MLPs), yang banyak digunakan sekarang. Kita mulai dengan bagaimana ANN lahir.

1 Dari Biologi ke *Artificial Neurons*

Sebetulnya ANN telah ada sejak lama, diperkenalkan tahun 1943 oleh seorang ahli fisiologi saraf (*neurophysiologist*) Warren McCulloh 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* (lihat *Chapter 5*). 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.

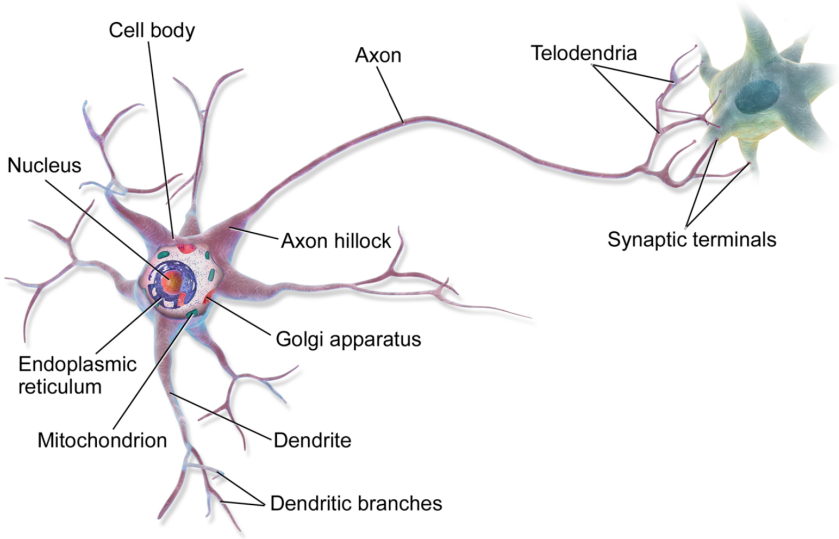
Sekarang kita menyaksikan gelombang ketertarikan lain pada ANN. Apakah ketertarikan ini akan sirna kembali seperti sebelumnya? Beberapa hal berikut merupakan alasan yang kuat untuk mempercayai bahwa kali ini akan berbeda dan ketertarikan kembali pada ANN akan membawa efek yang lebih besar terhadap kehidupan 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 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 impak 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

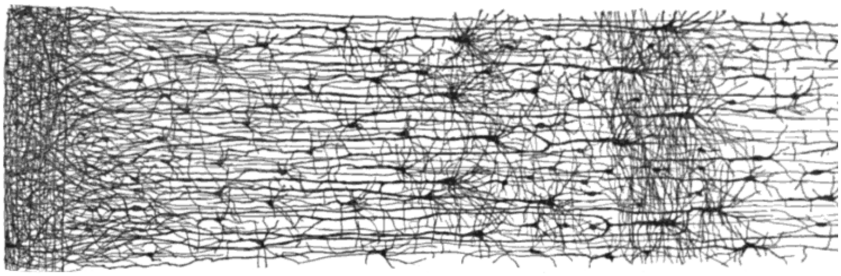
Sebelum kita mendiskusikan neuron artifisial, kita akan melihat sebentar pada neuron biologis yang direpresentasikan pada Gambar 8.1, yang merupakan sel yang terlihat tidak biasa pada otak hewan. Sel tersebut terdiri dari *cell body* yang berisi *nucleus* dan kebanyakan komponen-komponen sel yang kompleks, 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).



Gambar 8.1: Neuron biologis

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. Komutasi 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, 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 8.2.

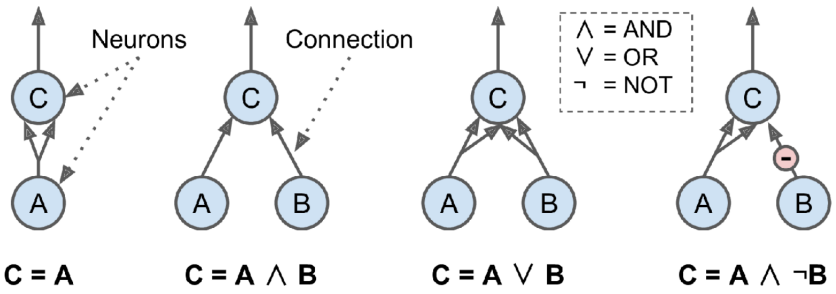


Gambar 8.2: Beberapa lapisan-lapisan pada BNN (cortex manusia)

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 8.3), dengan asumsi sebuah neuron teraktivasi ketika sedikitnya dua inputnya aktif.



Gambar 8.3: ANN mengoperasikan komputasi logika sederhana

Cara kerja jaringan adalah sebagai berikut:

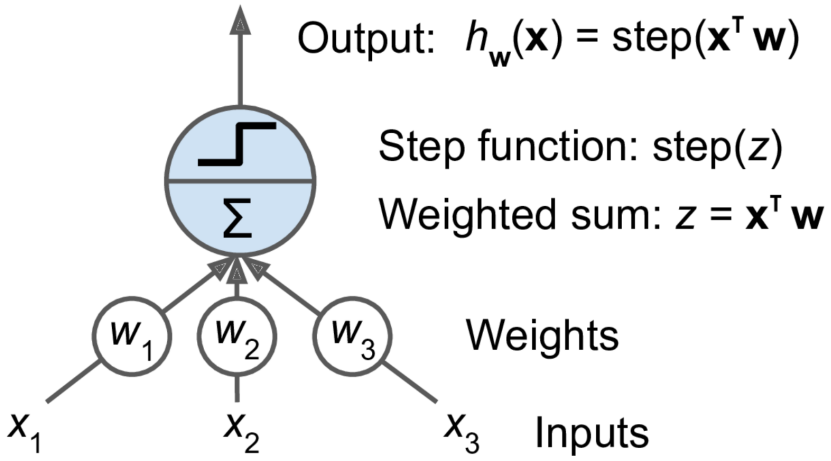
- Jaringan pertama sebelah kiri adalah fungsi identitas: Jika neuron A diaktivasi, kemudian neuron C teraktivasi juga (karena menerima dua sinyal input dari neuron A). Tetapi jika neuron A *off* maka neuron C *off* juga.
- Jaringan kedua menunjukan logika AND: Neuron C diaktivasi ketika kedua neuron 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.

Perceptron merupakan salah satu arsitektur ANN yang paling sederhana, yang ditemukan tahun 1957 oleh Frank Rosenblatt. Arsitektur ini berdasarkan AN yang sedikit berbeda yang disebut dengan ***threshold logic unit*** (TLU), atau kadang disebut dengan ***linear threshold unit*** (TLU). Input dan outputnya adalah bilangan (bukan hanya harga *on/off*), dan setiap koneksi input diasosiasikan dengan sebuah pembobot (*weight*). TLU menghitung penjumlahan input yang diboboti ($z = w_1x_1 + w_2x_2 + \cdots + w_nx_n = x^Tw$), kemudian mengaplikasikan fungsi step pada penjumlahan tersebut untu menghasilkan: $h_w(x) = \text{step}(z)$ dimana $z = x^Tw$.

Fungsi step yang paling banyak digunakan pada Perceptrons adalah ***Heaviside step function***, pada Persamaan (8.1). Dan kadang kadang digunakan fungsi ***sign function***.

> ***Persamaan (8.1).*** Fungsi step yang umumnya digunakan pada Perceptron (asumsi threshold = 0)
>
> $$\text{heaviside}(z) = \begin{cases} 0 \text{ jika } z < 0 \\ 1 \text{ jika } z \geq 0 \end{cases}$$
>
> $$\text{sgn}(z) = \begin{cases} -1 \text{ jika } z < 0 \\ 0 \text{ jika } z = 0 \\ 1 \text{ jika } z \geq 0 \end{cases}$$

1.3 *Perceptron*



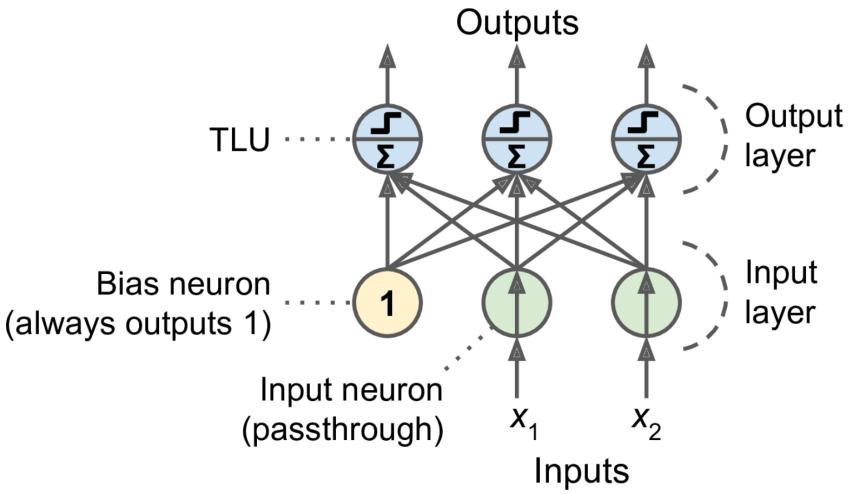
Gambar 8.4: Threshold Logic Unit (TLU)

Satu TLU dapat digunakan untuk klasifikasi biner linier yang sederhana, yaitu menghitung kombinasi linier dari input-input. Jika hasilnya melebihi *threshold* maka TLU akan mengeluarkan *positive class*, dan sebaliknya akan mengeluarkan *negative class* (seperti regresi logistik dan SVM linier). Sebagai contoh, kita dapat menggunakan sebuah TLU untuk mengklasifikasikan bunga Iris berdasarkan panjang petal dan lebar petal, dan juga menambahkan *feature* bias $x_0 = 1$, seperti yang telah kita pelajari sebelumnya. Mentraining TLU pada kasus ini mempunyai arti menemukan harga-harga yang tepat untuk w_0, w_1 dan w_2.

Perceptron hanya terdiri dari satu layer TLU, dimana setiap TLU dikoneksikan pada semua input. Ketika semua neuron pada sebuah layer dikoneksikan pada setiap neuron di layer sebelumnya (yaitu neuron-neuron input), maka layer tersebut disebut dengan layer yang terhubung sempurna (***fully connected layer***), atau layer padat (***dense layer***). Input-input pada perceptron di masukan ke dalam neuron-neuron khusus pelintas (*passthrough neurons*) yang disebut dengan neuron-neuron input (***input neuron***). Neuron input tersebut akan mengeluarkan apapun yang dimasukan. Semua neuron-neuron input membentuk lapisan input (***input layer***). Lebih jauh, *feature* bias ekstra biasanya ditambahkan ($x_0 = 1$), yang biasanya direpresentasikan menggunakan neuron tipe spesial yang disebut neuron bias (***bias neuron***) dengan output selalu sama dengan 1 sepanjang waktu. Sebuah perceptron dengan 2 input dan 3 output direpresentasikan pada Gambar 8.5. Perceptron ini dapat mengklasifikasikan *instance-instance* secara bersamaan ke dalam 3 *class* biner, sehingga dikategorikan sebagai *multioutput classifier*.

Dengan menggunakan Persamaan (8.2), yang merupakan persamaan aljabar sederhana, maka dimungkinkan untuk menghitung output dari sebuah layer AN untuk setiap *instance* secara sekaligus.

> ***Persamaan (8.2).*** Menghitung output-output dari sebuah *fully connected layer*
>
> $h_{\mathbf{w},\mathbf{b}(\mathbf{X})} = \phi\left(\mathbf{XW} + \mathbf{b}\right)$
>
> Berikut keterangan variabel dari Persamaan (8.2):
>
> - **X** merepresentasikan matriks input *features*. Mempunyai satu baris per *instance* dan satu kolom per *feature*



Gambar 8.5: Arsitektur Perceptron dengan 2 neuron input, satu neuron bias, dan 3 neuron output

- **W** adalah matriks pembobot yang berisi semua bobot-bobot koneksi kecuali yang berasal dari *bias neuron*.
- **b** merupakan *bias vector* yang berisi semua bobot-bobot koneksi antara *bias neuron* dan *artificial neuron*, dimana terdapat satu bagian *bias* untuk setiap *artificial neuron*.
- ϕ adalah fungsi aktivasi. Ketika semua *artificial neuron* merupakan TLU, maka fungsi aktivasi adalah fungsi step (catatan: terdapat fungsi aktivasi lain)

Pertanyaan selanjutnya adalah bagaimana melatih Perceptron? ALgoritma training Perceptron diajukan oleh Rosenblatt yang mendapatkan inspirasi dari *Hebb's rule*. Pada bukunya tahun 1949, *The Organiation of Behaviour*, Donald Hebb menyarankan bahwa ketika neuron biologis sering memicu neuron lain, koneksi antar dua neuron ini tumbuh menjadi lebih kuat. Bobot koneksi antara dua neuron cenderung bertambah apabila keduanya memicu secara berbarengan. Aturan ini kemudian dikenal dengan ***Heb's rule*** atau (***Hebbian Learning***). Perceptron dilatih untuk menggunakan varian dari aturan ini dengan mempertimbangkan error yang dibuat oleh *network* ketika membuat prediksi. Aturan *learning* dari Perceptron memperkuat koneksi-koneksi yang membantu mengurangi error. Lebih spesifik lagi, Perceptron diberikan satu *training instance* pada setiap satuan waktu, dan untuk setiap *instance* Perceptron akan membuat prediksi. Untuk setiap luaran neuron yang membuat prediksi salah, akan memperkuat bobot-bobot koneksi dari input-input yang berkontribusi terhadap prediksi yang benar. Aturan ini ditulis pada Persamaan (8.3).

Persamaan (8.3). *Learning rule* dari Perceptron

$$w_{i,j}^{(\text{next step})} = w_{i,j} + \eta \left(y_j - \hat{y}_j\right) x_i$$

dengan definisi variabel sebagai berikut

- $w_i j$ adalah bobot koneksi antara neuron input ke-i dan neuron output ke-j.
- x_i adalah harga input ke-i dari *training instance* sekarang
- $\hat{y}_j$ adalah output dari output neuron ke-j untuk *training instance* sekarang
- y_j adalah output target dari output neuron ke-j untuk *training instance* sekarang

- η adalah *learning rate*

`Scikit Learn` menyediakan *class* `Perceptron` yang mengimplementasikan sebuah jaringan dengan satu TLU. Kita dapat menggunakannya seperti yang pernah kita lakukan sebelum-sebelumnya, contohnya menggunakan dataset Iris.

- Definisikan *class* Perceptron

[4]:
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris
from sklearn.linear_model import Perceptron


iris = load_iris()
X = iris.data[:, (2, 3)]  # petal length, petal width
y = (iris.target == 0).astype(np.int)

per_clf = Perceptron(max_iter=1000, tol=1e-3, random_state=42)
per_clf.fit(X, y)

y_pred = per_clf.predict([[2, 0.5]])
```

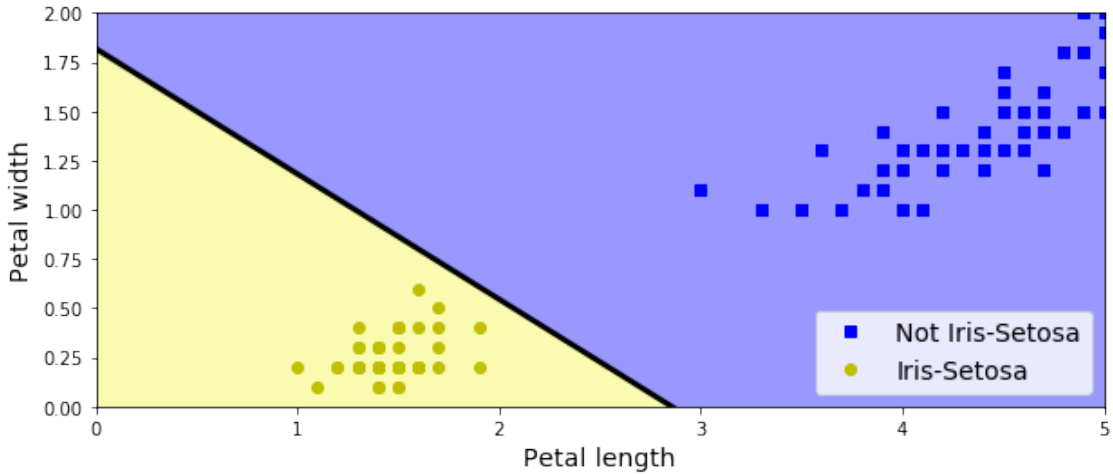
[5]:
```
y_pred
```

[5]:
```
array([1])
```

- Memplotting *decision boundary*

[7]:
```
a = -per_clf.coef_[0][0] / per_clf.coef_[0][1]
b = -per_clf.intercept_ / per_clf.coef_[0][1]

axes = [0, 5, 0, 2]

x0, x1 = np.meshgrid(
        np.linspace(axes[0], axes[1], 500).reshape(-1, 1),
        np.linspace(axes[2], axes[3], 200).reshape(-1, 1),
    )
X_new = np.c_[x0.ravel(), x1.ravel()]
y_predict = per_clf.predict(X_new)
zz = y_predict.reshape(x0.shape)

plt.figure(figsize=(10, 4))
plt.plot(X[y==0, 0], X[y==0, 1], "bs", label="Not Iris-Setosa")
plt.plot(X[y==1, 0], X[y==1, 1], "yo", label="Iris-Setosa")

plt.plot([axes[0], axes[1]], [a * axes[0] + b, a * axes[1] + b], "k-",␣
 ↪linewidth=3)
from matplotlib.colors import ListedColormap
```

```
custom_cmap = ListedColormap(['#9898ff', '#fafab0'])

plt.contourf(x0, x1, zz, cmap=custom_cmap)
plt.xlabel("Petal length", fontsize=14)
plt.ylabel("Petal width", fontsize=14)
plt.legend(loc="lower right", fontsize=14)
plt.axis(axes)
plt.show()
```

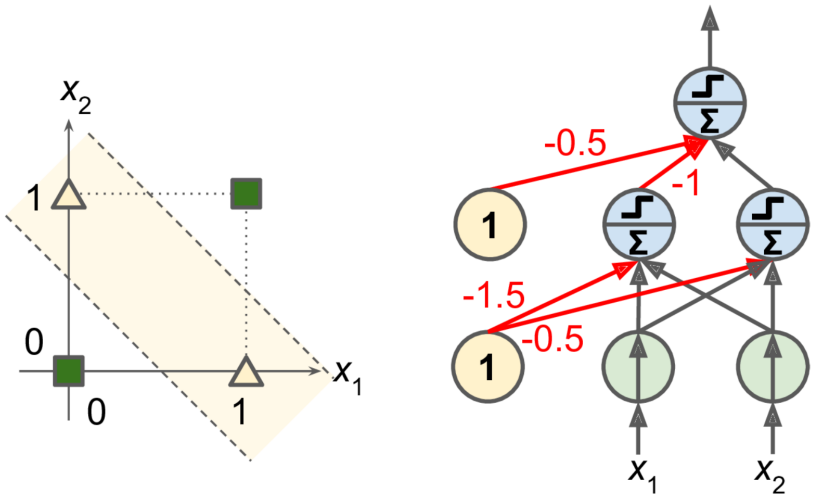


Decision boundary dari setiap neuron output adalah linier, sehingga Perceptron tidak mampu untuk mempelajari pola yang kompleks (seperti *classifier* regresi logistik). Tetapi jika *training instance linearly separable*, Rosenblatt telah mendemonstrasikan bahwa algoritma ini akan konvergen pada sebuah solusi, yang kemudian disebut dengan ***Perceptron convergence theorem***.

Kita dapat lihat bahwa algoritma *learning* dari Perceptron mirip dengan *Stochastic Gradient Descent*. Kenyataannya *class* `Perceptron` dari `Scikit Learn` ekivalen dengan sebuah `SGDClassifier` dengan *hyperparameter* sebagi berikut: `loss="Perceptron"`, `learning_rate="constant"`, `eta0=1` (*learning rate*), dan `penalty = None` (tidak ada regulasinya). Harap diperhatikan, tidak seperti *classifier* regresi logistik, Perceptron tidak mempunyai luaran probabilitas dari *class* . Tetapi Perceptron mebuat prediksi berdasarkan *hard threshold*. Inilah alasan mengapa regresi logistik lebih memilih regresi logistik dibanding Perceptron.

Pada tahun 1969 monograph *Perceptrons*, Marvin Minsky dan Seymour Papert menggarisbawahi beberapa kelemahan utama dari Percepetron. Khususnya kenyataan Perceptron tidak bisa memberikan solusi untuk permasalahan-permasalahan sederhana, misalkan masalah klasifikasi *Exclusive OR* (XOR), seperti yang ditunjukan pada Gambar 8.6 sebelah kiri. Permasalahan ini akan terjadi untuk semua model klasifikasi linier, termasuk regresi logistik. Tetapi para peneliti telah berharap terlalu banyak terhadap Perceptron, dan beberapa merasa kecewa kemudian meninggalkan *neural networks* secara bersamaan, dan memilih masalah-masalah dengan level lebih tinggi (*higher level problem*) seperti *logic*, *problem solving*, dan *searching*.

Kelemahan Perceptron di atas kemudian diketahui dapat dipecahkan dengan menumpukan



Gambar 8.6: Masalah klasifikasi XOR dan solusi MLP

(*stacking*) beberapa Perceptrons. Hasil ANN ini disebut dengan ***Multilayer Perceptron*** (MLP). Sebuah MLP dapat memecahkan masalah XOR, seperti yang dapat kita verifikasi dengan menghitung luaran dari MLP pada Gambar 8.6 sebelah kanan. Dengan input $(0, 0)$ atau $(1, 1)$ maka keluaran MLP adalah 0, sedangkan ketika input (0,1) atau (1,0) maka output adalah 1. Semua koneksi mempunyai bobot yang sama yaitu 1, kecuali empat koneksi dimana bobot ditunjukan. Silahkan dicoba sendiri MLP dapat mengatasi masalah XOR.

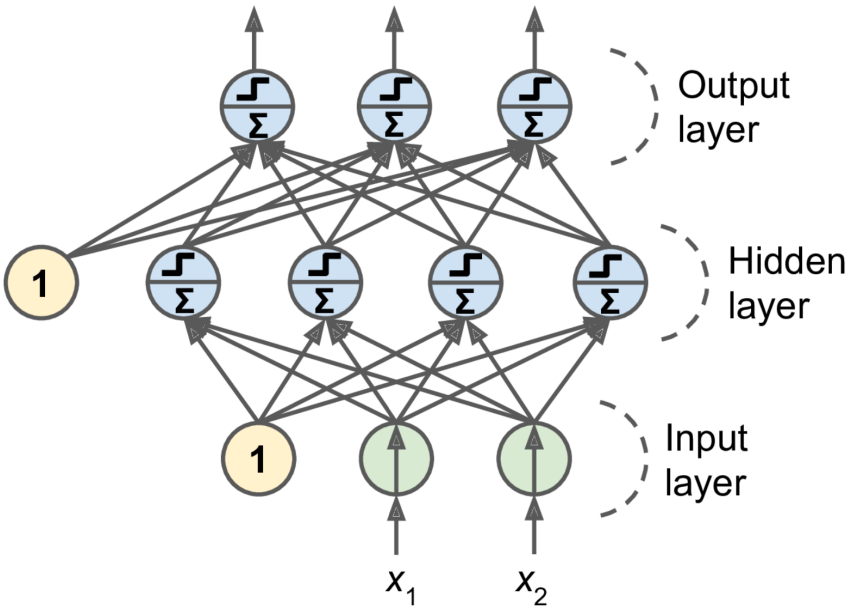
2 *Multilayer Perceptron* (MLP) dan *Backpropagation*

MLP terdiri dari satu layer input pelintas (*passthrough*), satu atau lebih TLU yang disebut dengan ***hidden layer***, dan satu layer terakhir yang disebut dengan layer output (lihat Gambar 8.7). Layer-layer yang dekat dengan layer input disebut dengan *lower layers* dan layer-layer yang dekat dengan layer output disebut dengan *upper layers*. Setiap layer kecuali layer output akan memasukan neuron bias dan dihubungkan seluruhnya (*fully connected*) pada layer selanjutnya.

Catatan. Aliran sinyal hanya pada satu arah yaitu dari input ke output, sehingga arsitektur ini merupakan contoh dari ***feedforward neural network*** (FNN).

Ketika ANN berisi tumpukan yang dalam (*deep*) dari *hidden layer*, maka disebut dengan ***Deep Neural Network*** (DNN). Bidang *Deep Learning* mempelajari DNNs, dan lebih umum mempelajari model-model yang berisi tumpukan komputasi yang dalam/banyak (*deep stacks of computations*). Kendatipun demikian, banyak orang yang mengatakan *Deep Learning* meskipun sedang membicarakan *neural networks* secara umum termasuk *shallow neural networks*.

Sudah bertahun-tahun para peneliti berjuang mencari cara untuk mentraining MLP tanpa kesuksesan. Tetapi, tahun 1986, David Rumelhart, Geoffrey Hinton dan Ronald Williams mempublikasikan paper terobosan yang memperkenalkan algoritma training ***Backpropagation*** (paper "*Learning Internal Representation by Error Propagation*", *Defense Technical Information Center technical report*, September 1985). Secara singkat, algoritma ini seperti *Gradient Descent* yang menggunakan teknis yang efisien untuk menghitung gradien secara otomatis. Dengan hanya melewatkan dua kali ke dalam network, satu *forward* dan satu *backward*, algoritma *backpropagation* mampu menghitung gradien dari error network terhadap setiap parameter model. Dengan kata lain, algoritma



Gambar 8.7: Arsitektur MLP dengan dua input, satu hidden layer dengan empat neuron, dan tiga neuron output (neuron bias ditunjukan di sini meskipun biasanya implisit)

mencari bagaimana bobot koneksi dan setiap bagian bias diatur sehingga mengurangi error. Setelah didapatkan gradiennya, maka algoritma akan menjalankan langkah-langkah *Gradient Descent*, dan keseluruhan proses diulang sampai network konvergen pada solusi.

Catatan. Perhitungan gradien secara otomatis disebut dengan ***automatic differentiation*** atau ***autodiff***. Banyak teknik *autodiff* dengan pro dan kontra yang berbeda-beda. Metode yang dipakai pada *backpropagation* adalah ***reverse-mode autodiff*** yang cepat dan presisi. Metode ini sangat cocok ketika fungsi yang diturunkan mempunyai banyak variabel (mis. bobot koneksi) dan sedikit output (mis. satu *loss*).

Detail dari algoritma adalah sebagai berikut:

- Algoritma menghandel satu *mini-batch* pada satu satuan waktu (misalkan, masing-masing berisi 32 *instance*), dan menelusuri *full training set* beberapa kali. Setiap sekali iterasi disebut dengan *epoch*.
- Setiap *mini-batch* dilewatkan ke dalam layer input dari network, yang kemudian mengirimkannya ke *hidden layer* pertama. Algoritma kemudian menghitung output dari semua neuron-neuron pada layer ini (untuk setiap *instance* pada *mini-batch*). Hasilnya dilewatkan pada layer selanjutnya, outputnya dihitung dan dilewatkan pada layer selanjutnya, demikian seterusnya sampai ada output pada layer terakhir yaitu layer output. Mekanisme ini disebut *forward pass* yang sangat mirip dengan melakukan prediksi, kecuali semua *intermeddiate results* dipelihara karena diperlukan pada saat *backward pass*.
- Kemudian, algoritma mengukur error output network (mis. menggunakan sebuah *loss function* yang membandingkan ouput yang diinginkan dengan output aktual dari network, dan mengembalikan beberapa ukuran kerja dari eror.
- Kemudian algoritma menghitung seberapa besar masing-masing koneksi output berkontribusi terhadap error. Hal ini dilakukan secara analitis dengan menerapkan *chain rule*

(mungkin aturan yang paling fundamental di kalkulus), yang membuat langkah ini cepat dan presisi.

- Algoritma ini selanjutnya mengukur seberapa besar kontribusi error datang dari setiap koneksi pada layer dibawahnya dengan menggunakan *chain rule* kembali, tetapi pada langkah ini *chain rule* bekerja terbalik (*backward*) berurutan sampai algoritma mencapai layer input. Seperti yang telah dijelaskan semuanya, *reverse pass* mengukur gradien error melalui semua bobot koneksi pada network dengan mempropagasikan error gradien *backward* melalui network, sehingga namanya disebut algoritma *backpropagation*).
- Terakhir, algoritma melakukan langkah *Gradient Descent* untuk memodifikasi semua bobot-bobot koneksi pada network menggunakan error gradien yang telah dihitung sebelumnya.

Algoritma ini cukup penting sehingga kita akan meringkasnya kembali: Untuk setiap training *instance*, algoritma *backpropagation* pertama kali akan membuat prediksi (*forward pass*) dan mengukur error, yang kemudian masuk ke dalam setiap layer secara berlawanan umtuk mengukur kontribusi error dari masing-masing koneksi (*reverse pass*), dan akhirnya akan memodifikasi bobot koneksi untuk mengurangi error (langkah *gradient descent*).

Catatan. Penting untuk melakukan inisialisasi bobot koneksi pada *hidden layer* secara random, atau proses training akan gagal. Misalkan, jika kita menginisialisasi semua bobot dan bias sama dengan nol, maka semua neuron pada layer tertentu menjadi seperti sama secara sempurna. Sehingga *backpropagation* akan mempengaruhinya dengan cara yang betul-betul sama dan mereka akan tetap identik. Dengan kata lain, meskipun terdapat ratusan neuron per layer, model kita akan bertindak seolah hanya satu neuron per layer dan tidak akan menghasilkan model yang terlalu pintar. Tetapi jika kita menginisialisasi bobot secara random. kita akan memecah kesimetrian dan membuat algoritma *backpropagation* mentraining neuron-neuron yang tidak seragam.

Untuk membuat algoritma ini bekerja dengan baik, beberapa perubahan penting yang dilakukan pada arsitektur MLP adalah menggantikan fungsi step dengan fungsi sigmoid, $\sigma = \dfrac{1}{1+\exp(-z)}$ (lihat juga regresi logistik). Hal ini cukup esensial karena fungsi step hanya mempunyai segmen-segmen datar, sehingga tidak mempunyai gradien untuk mengimplementasikan *gradient descent* (*gradient descent* tidak dapat bergerak pada fungsi yang rata), sementara fungsi logistik mempunyai turunan non-zero di semua tempat, yang membuat *gradient descent* bekerja pada setiap langkahnya. Sebetulnya, *backpropagation* bekerja sangat baik juga dengan banyak fungsi aktivasi, bukan hanya fungsi logistik. Berikut contoh dua fungsi pilihan lain:

- Fungsi Tangen Hiperbolik (*Hyperbolic Tangent Function*): $\tanh(z) = 2\sigma(2z) - 1$

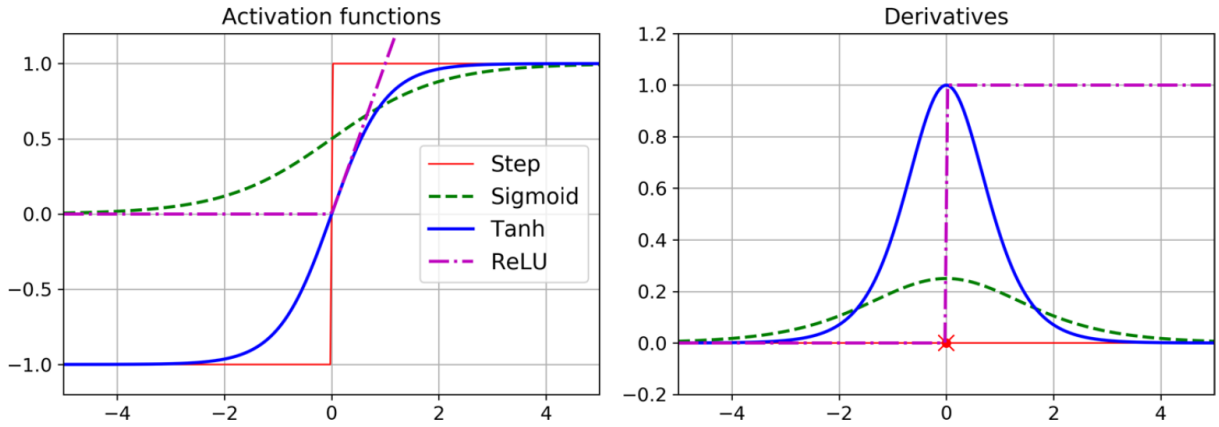
  Seperti fungsi logistik, fungsi aktivasi ini berbentuk S (*S-shaped*), kontinyu, dan dapat diturunkan, tetapi outputnya mampunyai harga antara -1 s/d 1 (bukan 0 s/d 1 seperti pada fungsi logistik). Range ini membuat output cenderung berada di sekitar 0 pada saat training dimulai, yang sering menolong untuk mempercepat konvergensi.

- Fungsi *Rectified Linear Unit* (ReLU): $\text{ReLU}(z) = \max(0, z)$

  Fungsi ReLU adalah kontinyu tetapi sayangnya tidak dapat diturunkan (*not differentiable*) pada $z = 0$ (kemiringan berubah secara tiba tiba, yang dapat membuat *gradient descent* loncat-loncat (*bounce around*)), dan turunan adalah 0 untuk $z < 0$. Saat praktis, hal ini dapat bekerja dengan baik dan mempunyai keuntungan kecepatan saat dihitung, sehingga manjadi *default*. Yang paling penting, kenyataan bahwa ReLU tidak mempunyai nilai maksimum

output dapat menolong untuk mengurangi beberapa isu saat *gradient descent*.

Mengapa kita membutuhkan fungsi aktivasi? Karena jika kita membuat rantai transformasi linier, yang kita peroleh hanya transformasi linier juga. Misalkan, $f(x) = 2x + 3$ dan $g(x) = 5x - 1$, maka hasil transformasi berantai dari kedua fungsi ini adalah fungsi linier lain, $f(g(x)) = 2(5x - 1) + 3 = 10x + 1$. Sehingga jika kita tidak mempunyai nonlinieritas antar layer, maka meskipun kita mempunyai tumpukan yang dalam dari layer-layer, akan sama seperti hanya mempunyai satu layer saja. Dengan kondisi tersebut kita tidak akan dapat memecahkan permasalahan yang kompleks. Sebaliknya, jika dipunyai DNN yang besar dengan fungsi aktivasi nonlinier secara teori kita dapat mengaproksimasi fungsi kontinyu sembarang. Fungsi-fungsi aktivasi yang populer beserta turunannya dapat dilihat pada Gambar 8.8.



Gambar 8.8: Fungsi-fungsi aktivasi beserta turunannya

Kita sudah mempelajari asal mula *neural network*, arsitekturnya, dan bagaimana menghitung output. Selain itu kita telah mempelajari algoritma *Backpropagation*. Dua *subsection* selanjutnya akan menjelaskan secara singkat kegunaannya, yaitu Regresi MLP dan Klasifikasi MLP.

2.1 Regresi MLP

MLP dapat digunakan untuk regresi. Jika kita ingin memprediksi nilai tunggal (misalkan harga rumah untuk *feature-feature* tertentu), maka kita hanya membutuhkan sebuah neuron dengan satu output yang merupakan nilai yang diprediksinya. Untuk regresi multivariat (memprediksi beberapa nilai sekaligus), maka kita membutuhkan satu neuron output untuk setiap dimensi outputnya. Sebagai contoh, untuk mencari lokasi titik tengah sebuah objek pada image, kita perlu memprediksi koordinat 2D, maka dibutuhkan dua neuron output. Jika diinginkan juga menempatkan *bounding box* sekitar objek, maka kita memerlukan dua variabel lagi yaitu lebar dan tinggi dari objek. Sehingga, kita memerlukan empat neuron output.

Secara umum, ketika membangun MLP untuk regresi, kita tidak ingin menggunakan fungsi aktivasi sembarang untuk neuron-neuron output. Jika kita ingin memastikan bahwa output selalu positif, maka kita dapat gunakan fungsi aktivasi ReLU pada layer output. Atau sebagai alternatif kita dapat gunakan fungsi aktivasi *softplus* yang merupakan varian halus dari ReLU: $\text{softplus}(z) = \log(1 + \exp(z))$. Terlihat nilai akan dekat dengan 0 jika z negatif dan mendekati z jika z positif. Terakhir, jika ingin memastikan bahwa prediksi akan jatuh dalam range tertentu, maka kita dapat gunakan fungsi logistik atau fungsi tangen hiperbolik, kemudian lakukan penskalaan pada label-label sehingga berada pada range 0 s/d 1 untuk fungsi logistik, dan -1 s/d 1

untuk tangen hiperbolik. Tabel 8.9. merupakan ringkasan arsitektur pada regresi MLP.

Hyperparameter	Typical value
# input neurons	One per input feature (e.g., 28 x 28 = 784 for MNIST)
# hidden layers	Depends on the problem, but typically 1 to 5
# neurons per hidden layer	Depends on the problem, but typically 10 to 100
# output neurons	1 per prediction dimension
Hidden activation	ReLU (or SELU, see Chapter 11)
Output activation	None, or ReLU/softplus (if positive outputs) or logistic/tanh (if bounded outputs)
Loss function	MSE or MAE/Huber (if outliers)

Tabel 8.1: Tipikal arsitektur regresi MLP

Loss function yang digunakan saat training biasanya *mean-squared error*. Tetapi jika banyak pencilan data (*outlier*) pada training set, maka disarankan untuk menggunakan *mean absolute error*. Sebagai alternatif, kita bisa gunakan *Huber Loss* yang merupakan kombinasi keduanya.

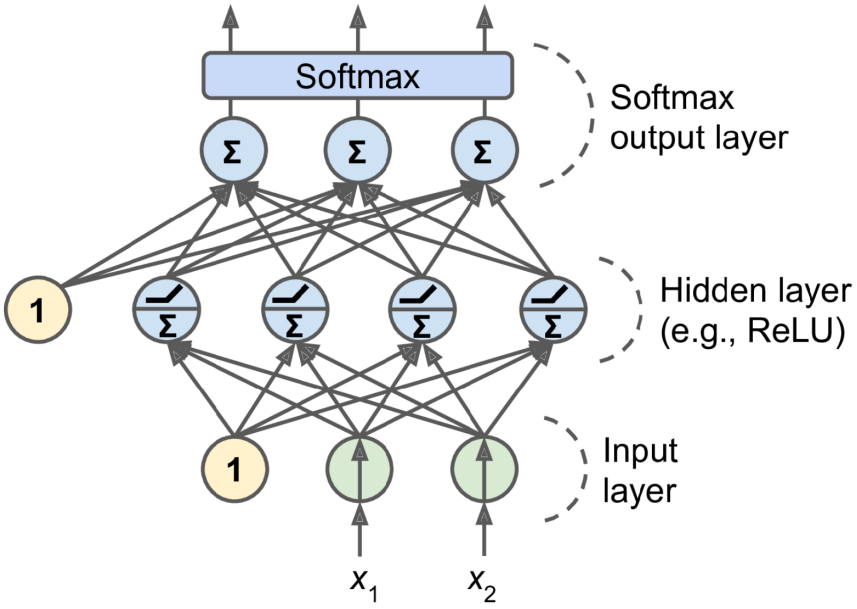
Catatan. *Huber Loss* adalah kuadratik ketika error lebih kecil dari threshold δ (biasanya 1), tetapi linier ketika error lebih besar dari δ. Bagian linier membuatnya kurang sensitif diandingkan dengan *mean-squared error*, dan bagian kuadratiknya membuat lebih cepat konvergen dan lebih presisi daripada *mean-squared error*.

2.2 Klasifikasi MLP

MLP dapat juga digunakan untuk klasifikasi. Untuk masalah klasifikasi biner, kita hanya membutuhkan satu neuron output menggunakan fugnsi aktivasi logistik. Outputnya berupa bilangan antara 0 dan 1, dimana dapat diinterpretasikan sebagai estimasi probabilitas dari *positive class*. Estimasi probabilitas untuk *negative class* akan sama dengan 1 dikurangi nilai estimasi probabilitas *positive class*.

MLP juga dapat dengan mudah menghandel masalah klasifikasi multilabel. Misalkan, kita mempunyai sistem klasifikasi email yang memprediksi apakah masing-masing email yang datang adalah email bukan spam (ham) atau email spam, dan sekaligus memprediksi apakah email tersebut *urgent* atau *nonurgent*. Pada kasus ini kita membutuhkan dua neuron output, keduanya menggunakan fungsi aktivasi logistik. Yang pertama akan mengeluarkan probabilitas bahwa email spam, dan yang kedua akan mengeluarkan probabilitas bahwa email *urgent*. Lebih umum, kita dapat mendedikasikan satu neuron output untuk setiap *positive class*. Sebagai catatan, probabilitas output tidak perlu berjumlah 1. Hal ini akan membuat model mengeluarkan kombinasi label, misalkan kita akan mendapatkan email yang *nonurgent* ham, *urgent* ham, *nonurgent* spam, dan mungkin bahkan *urgent spam* (meskipun bisa jadi ini merupakan error).

Jika setiap *instance* hanya dapat dipunyai oleh satu *class* tertentu dari tiga atau lebih *class* lain yang mungkin, maka kita membutuhkan satu neuron output untuk tiap *class*. Dalam kasus ini, maka kita gunakan fungsi aktivasi *softmax* untuk keseluruhan layer (lihat Gambar 8.10). Fungsi *softmax* (lihat *Chapter* 2), akan menjamin bahwa semua estimasi probabilitas akan berada diantara 0 dan 1 dan dijumlahkan sama dengan 1. Ini disebut dengan klasifikasi multiclass.



Gambar 8.9: MLP modern (termasuk ReLU dan Softmax) untuk klasifikasi

Menyangkut *loss function*, karena kita memprediksi distribusi probabilitas, *cross entropy loss* (atau *log loss*) merupakan pilihan yang baik. Tabel 8.11 menunjukan tipikal arsitektur klasifikasi MLP.

Hyperparameter	Binary classification	Multilabel binary classification	Multiclass classification
Input and hidden layers	Same as regression	Same as regression	Same as regression
# output neurons	1	1 per label	1 per class
Output layer activation	Logistic	Logistic	Softmax
Loss function	Cross entropy	Cross entropy	Cross entropy

Tabel 8.2: Tipikal arsitektur klasifikasi MLP