UNIVERSITAS INDONESIA

METODE SELEKSI FITUR MENGGUNAKAN TEKNIK PERANKINGAN BERBASIS BOBOT SECARA MULTI STEP MENGGUNAKAN DEEP LEARNING UNTUK PENCARIAN BIOMARKER PADA DATA MICROARRAY

THESIS

MUKHLIS AMIEN
1406522102

FAKULTAS ILMU KOMPUTER
PROGRAM STUDI MAGISTER ILMU KOMPUTER
DEPOK
JUNI 2016

UNIVERSITAS INDONESIA

METODE SELEKSI FITUR MENGGUNAKAN TEKNIK PERANKINGAN BERBASIS BOBOT SECARA MULTI STEP MENGGUNAKAN DEEP LEARNING UNTUK PENCARIAN BIOMARKER PADA DATA MICROARRAY

THESIS

Diajukan sebagai salah satu syarat untuk memperoleh gelar

Master

MUKHLIS AMIEN

1406522102

FAKULTAS ILMU KOMPUTER

PROGRAM STUDI MAGISTER ILMU KOMPUTER

DEPOK

JUNI 2016

HALAMAN PERSETUJUAN

Judul	:	Metode Seleksi Fitur Menggunakan Teknik Perankingan Berbasis Bobot Secara Multi Step Menggunakan Deep Learning untuk Pencarian Biomarker pada Data Microarray
Nama	:	Mukhlis Amien
NPM	:	1406522102

Laporan Thesis ini telah diperiksa dan disetujui.

20 Juni 2016

Ir. Ito Wasito M.Sc., PhD.

Pembimbing Thesis

HALAMAN PERNYATAAN ORISINALITAS

Thesis ini adalah hasil karya saya sendiri,
dan semua sumber baik yang dikutip maupun dirujuk
telah saya nyatakan dengan benar.

Nama	:	**Mukhlis Amien**
NPM	:	**1406522102**
Tanda Tangan	:	
Tanggal	:	**20 Juni 2016**

HALAMAN PENGESAHAN

Thesis ini diajukan oleh	:	
Nama	:	Mukhlis Amien
NPM	:	1406522102
Program Studi	:	Magister Ilmu Komputer
Judul Thesis	:	Metode Seleksi Fitur Menggunakan Teknik Perankingan Berbasis Bobot Secara Multi Step Menggunakan Deep Learning untuk Pencarian Biomarker pada Data Microarray

Telah berhasil dipertahankan di hadapan Dewan Penguji dan diterima sebagai bagian persyaratan yang diperlukan untuk memperoleh gelar Master pada Program Studi Magister Ilmu Komputer, Fakultas Ilmu Komputer, Universitas Indonesia.

DEWAN PENGUJI

Pembimbing	:	Ir. Ito Wasito M.Sc., PhD.	( )
Penguji	:	Ari Saptawijaya, S.Kom., M.Sc., Ph.D.	( )
Penguji	:	dr. Iik Wilarso M.T.I.	( )
Penguji	:	Dr. Eng. M. Rahmat Widyanto S.Kom., M.Eng.	( )

Ditetapkan di : Depok
Tanggal : 22 Juli 2016

KATA PENGANTAR

Pertama-tama saya ucapkan syukur kehadirat Allah SWT dan Junjungan kita Nabi Muhammad SAW, atas segala petunjuk dan karunianyalah thesis ini bisa terselesaikan. Saya mengucapkan terima kasih yang sebesar-besarnya atas bimbingan dan petunjuk dari pembimbing saya Bpk Ito Wasito PhD. Kepada istri saya, Catur Prastiasih yang sangat mendukung saya melanjutkan sekolah. Kepada teman-teman satu bimbingan yaitu Aris dan Arida yang telah memberikan ide dan masukan serta diskusi yang mendalam.

Kepada Fakultas Ilmu Komputer Universitas Indonesia, yang telah memberikan fasilitas lab yang sangat membantu saya dalam menyelesaikan thesis ini.

Kepada orang tua saya, terima-kasih banyak atas dukungan moral dan spiritual yang diberikan. Dan saudara-saudara saya di Batu.

Depok, 20 Juni 2016

Mukhlis Amien

HALAMAN PERNYATAAN PERSETUJUAN PUBLIKASI TUGAS AKHIR UNTUK KEPENTINGAN AKADEMIS

Sebagai sivitas akademik Universitas Indonesia, saya yang bertanda tangan di bawah ini:

Nama	:	Mukhlis Amien
NPM	:	1406522102
Program Studi	:	Magister Ilmu Komputer
Fakultas	:	Ilmu Komputer
Jenis Karya	:	Thesis

demi pengembangan ilmu pengetahuan, menyetujui untuk memberikan kepada Universitas Indonesia **Hak Bebas Royalti Noneksklusif (Non-exclusive Royalty Free Right)** atas karya ilmiah saya yang berjudul:

Metode Seleksi Fitur Menggunakan Teknik Perankingan Berbasis Bobot Secara Multi Step Menggunakan Deep Learning untuk Pencarian Biomarker pada Data Microarray

beserta perangkat yang ada (jika diperlukan). Dengan Hak Bebas Royalti Noneksklusif ini Universitas Indonesia berhak menyimpan, mengalihmedia/formatkan, mengelola dalam bentuk pangkalan data (database), merawat, dan memublikasikan tugas akhir saya selama tetap mencantumkan nama saya sebagai penulis/pencipta dan sebagai pemilik Hak Cipta.

Demikian pernyatan ini saya buat dengan sebenarnya.

Dibuat di : Depok
Pada tanggal : 20 Juni 2016

Yang menyatakan

(Mukhlis Amien)

ABSTRAK

Nama	:	Mukhlis Amien
Program Studi	:	Magister Ilmu Komputer
Judul	:	Metode Seleksi Fitur Menggunakan Teknik Perankingan Berbasis Bobot Secara Multi Step Menggunakan Deep Learning untuk Pencarian Biomarker pada Data Microarray

Data ekspresi gen pada percobaan microarray memiliki ciri khas yaitu jumlah sampel yang sedikit dengan dimensi fitur yang sangat besar. Algoritma *Deep Belief Network (DBN)* adalah bagian dari algoritma *deep learning* yang menerapkan teknik *unsupervised learning* secara *greedy layer wise training*. DBN ini dapat digunakan untuk membantu menganalisa data ekspresi gen. Algoritma seleksi fitur yang berbasis pada perankingan bobot secara multi-step pada penelitian ini digunakan untuk mendapatkan fitur gen *biomarker*, yaitu profil gen yang paling informatif dengan melakukan perankingan berdasarkan bobot jaringan *deep belief network* (DBN). Algoritma ini digunakan untuk memilih fitur gen dari suatu percobaan microarray *lung adenocarcinoma* (kanker paru-paru). *Deep Belief Network* (DBN) adalah *Restricted Boltzmann Machine* (RBM) yang dirangkai menjadi jaringan yang dijajarkan untuk membentuk jaringan yang lebih dalam. Seleksi fitur gen, berdasarkan ranking bobot yang dihasilkan oleh algoritma ini terbukti dapat digunakan untuk pencarian *Biomarker*. Hal ini dibuktikan dengan melakukan evaluasi bahwa hanya dengan menggunakan *biomarker* yang didapatkan sebagai data pada teknik *machine learning* umum yaitu *multi layers perceptron (MLP)*, sudah bisa melakukan klasifikasi pasien sehat atau pasien sakit. Untuk melakukan konfirmasi bahwa gen *biomarker* tersebut adalah merupakan *biomarker* dari penyakit kanker, maka dilakukan perbandingan dengan hasil dari studi literatur.

Kata Kunci:
Microarray, ekspresi gen, Algoritma Seleksi fitur, multi-step ranking, deep belief network, restricted boltzmann machine, feature selection, deep learning, unsupervised learning, biomarker.

ABSTRACT

Name	:	Mukhlis Amien
Program	:	Magister Ilmu Komputer
Title	:	Feature Selection Method Using Multi Step Weigh Based Rank Using Deep Learning To Search Biomarker In Microarray Data

Gene expression data acquired by microarray experiments has a characteristic that the number of samples usually small but the number of features are very large. Deep Belief Network (DBN) is part of deep learning algorithms which apply unsupervised learning with greedy layer wise training technique. DBN can be used to analyse gene expression data. Feature selection algorithm used by this study is based on multi-step weight based ranking extracted from DBN model to search biomarker from gene expression profile. This algorithm is applied for lungs adenocarcinoma microarray dataset. DBNs can be viewed as a deep composition of simple, unsupervised networks of restricted Boltzmann machines (RBMs). This technique can solve the problem of searching biomarker extracted from microarray dataset. We evaluate the biomarker found by this method by using the biomarker as an input data to a supervised machine learning method using multi layers perceptron (MLP). We evaluate this MLP by analyzing the accuracy of classification problem from cancerous and healthy microarrays patients data. As a confirmation, we conduct literature study about biomarkers genes found by this methods.

Keywords:
Microarray, gene expression, Feature Selection Algoritm, deep learning, deep belief networks, restricted boltzmann machine, unsupervised learning, greedy layer-wise training, biomarker.

DAFTAR ISI

DAFTAR GAMBAR

DAFTAR TABEL

BAB 1
PENDAHULUAN

1.1 Latar Belakang

Data ekspresi gen pada percobaan *microarray* memiliki ciri khas yaitu dimensi fitur gen yang jauh lebih besar dibandingkan dengan sampel pasien. Masalah tersebut menyebabkan penerapan teknik pendeteksian penyakit genetis dengan menggunakan data ekspresi gen lebih sulit dilakukan, sebab data ekspresi gen tersebut memiliki signifikansi yang berbeda-beda. Menurut penelitian Yoon et al. (2006) dan Bandyopadhyay et al. (2014) tidak semua ekspresi gen yang didapatkan dalam percobaan *microarray* tersebut adalah gen yang informatif, bahkan jumlah ekspresi gen yang informatif untuk kasus yang diinginkan misalnya untuk pengenalan sel kanker, sangat sedikit dibandingkan dengan keseluruhan ekspresi gen yang didapatkan dalam sebuah percobaan (Bandyopadhyay et al., 2014). Data ekspresi gen yang tidak informatif tersebut dapat mengganggu dan mengurangi performa secara signifikan pada teknik pengenalan pola penyakit yang diterapkan. Akan tetapi, beberapa gen yang informatif berpengaruh secara signifikan terhadap pengenalan pola tersebut. Sebagai contoh, untuk mendiagnosa kanker paru-paru, hanya dibutuhkan sekitar 50 gen saja dari 22 ribu gen yang didapatkan dalam percobaan. Gen-gen yang paling informatif ini disebut dengan *Biomarker* (Belinsky, 2004). Sehingga hanya dengan menggunakan data *Biomarker* yang ditemukan saja, sudah dapat digunakan untuk mengenali penyakit yang diderita oleh pasien.

Pada penelitian ini, akan dibangun sebuah teknik pencarian *Biomarker* dengan metode seleksi fitur gen. Metode ini menerapkan perankingan gen secara *multi step* terhadap model yang didapatkan pada proses *training*. Arsitektur yang digunakan adalah arsitektur *Deep Belief Network (DBN)* yang merupakan bagian dari metode *deep learning*. Metode perankingan yang digunakan adalah modifikasi dari algoritma seleksi fitur untuk *logistic regression* yang dilakukan oleh Shevade and Keerthi (2003), tetapi metode ini memiliki kelemahan dan masalah dalam mengeliminasi fitur jika diterapkan secara langsung pada model DBN, dikarenakan parameter bobot (W) dan bias (b) ditempatkan di setiap fitur dan model ini hanya memiliki satu layer dibandingkan dengan DBN yang memiliki banyak layer.

DBN merupakan jaringan *Restrictive Boltzmann Machine (RBM)* yang disusun secara bertingkat. Dimulai dengan memberikan bobot random diantara dua network, yang dapat dilatih dengan cara meminimalkan perbedaan antara data asli dengan data rekonstruksinya. *Gradien* didapatkan dengan *chain rule* untuk melakukan penurunan error dengan teknik *Contrastive Divergence (CD)*. Untuk dicari bobot (W) dan bias dengan *maximum likelihood learning* secara *greedy* pada tiap layernya (Hinton and Salakhutdinov, 2006).

Pada DBN, *hidden unit* yang paling sering aktif adalah *hidden unit* yang lebih penting dibandingkan dengan *hidden unit* yang jarang aktif, oleh karena itu *hidden unit* ini memiliki parameter bobot yang lebih besar dibandingkan dengan hidden unit yang jarang aktif pada saat proses *training* dilakukan. Pemilihan fitur dilakukan dengan meranking unit-unit yang memiliki bobot tertinggi dimulai dari *layer output* menuju *layer input* untuk mendapatkan fitur gen yang paling berpengaruh. Kemudian dilakukan eliminasi bobot pada *hidden unit* per layernya secara *multi step*. Selanjutnya akan dipilih sebanyak *top-n* gen dari hasil perankingan ini untuk dievaluasi apakah *Biomarker* yang ditemukan tersebut informatif atau tidak.

Tahapan berikutnya, fitur yang telah didapatkan akan digunakan sebagai data input pada *Multi Layer Perceptron* (MLP) dengan tujuan untuk melakukan evaluasi apakah gen *Biomarker* yang ditemukan dengan perankingan tersebut dapat memperbaiki hasil klasifikasi pasien sakit atau sehat. Untuk mengetahui keakuratannya, dilakukan perbandingan hasil eksperimen ini dengan hasil pada eksperimen lain pada literatur yang juga bertujuan untuk menemukan *Biomarker*.

1.2 Rumusan Masalah

Berdasarkan pada uraian pendahuluan diatas maka dapat dibuat rumusan permasalahan sebagai berikut: Dikarenakan karakteristik sedikitnya sampel dan besarnya fitur pada data ekspresi gen serta signifikansi pencarian Biomarker pada penyakit yang disebabkan oleh genetis, maka apakah metode seleksi fitur berbasis perankingan bobot secara multi step menggunakan deep learning untuk pencarian Biomarker tersebut dapat diterapkan?

1.3 Batasan Permasalahan

- Dataset yang digunakan adalah data ekspresi gen microarray untuk penyakit kanker paru-paru yang tersedia secara bebas dengan kode GSE10072
- Data yang digunakan adalah dataset yang sudah dilakukan pengolahan awal standar.
- Komputer 1 yang digunakan adalah laptop core i7 dengan memory 8 Gb.
- Komputer 2 adalah desktop core i5, vga geForce 315 dengan memory 1 gb, dan ram 4 gb.

1.4 Tujuan Penelitian

Penelitian ini bertujuan untuk:

- Membangun metodologi pencarian *Biomarker* pada dataset ekspresi gen percobaan *microarray*.
- Membuat algoritma perankingan gen secara multi step yang diterapkan pada arsitektur DBN.
- Melakukan evaluasi apakah *Biomarker* yang ditemukan oleh metode ini untuk dilakukan verifikasi dengan literatur.

1.5 Manfaat Penelitian

Hasil dari penelitian ini memiliki manfaat :

- Framework DBN untuk pencarian *Biomarker* ini dapat diterapkan untuk mendeteksi apakah seseorang memiliki resiko genetis penyakit kanker paru-paru.
- Mendapatkan fitur gen yang paling penting dan informatif pada kasus penyakit kanker paru-paru.
- Melakukan pendeteksian kanker paru-paru secara dini dengan data yang didapatkan dari profil gen pasien pada eksperimen *microarray*.

1.6 Sistematika Penulisan

Sistematika penulisan laporan adalah sebagai berikut:

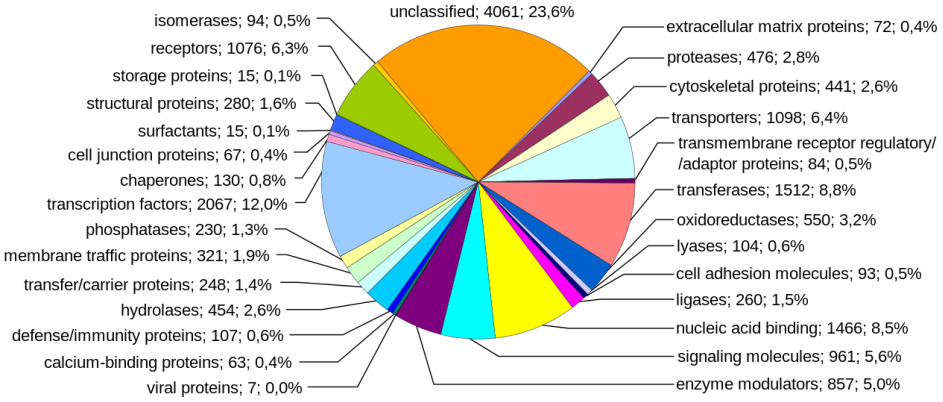
- Bab 1 PENDAHULUAN
  Berisi gambaran umum permasalahan dan metodologi apa yang akan diterapakan.
- Bab 2 TINJAUAN PUSTAKA
  Landasan teori dipakainya metodologi yang akan diterapkan dalam eksperimen ini.
- Bab 3 METODOLOGI PENELITIAN
  Penjelasan detail metodologi yang akan diterapkan dalam penelitian.
- Bab 4 PEMBAHASAN
  Pembahasan hasil dari eksperimen yang sudah dilakukan.
- Bab 5 KESIMPULAN DAN SARAN

BAB 2
TINJAUAN PUSTAKA

2.1 Ekspresi Gen

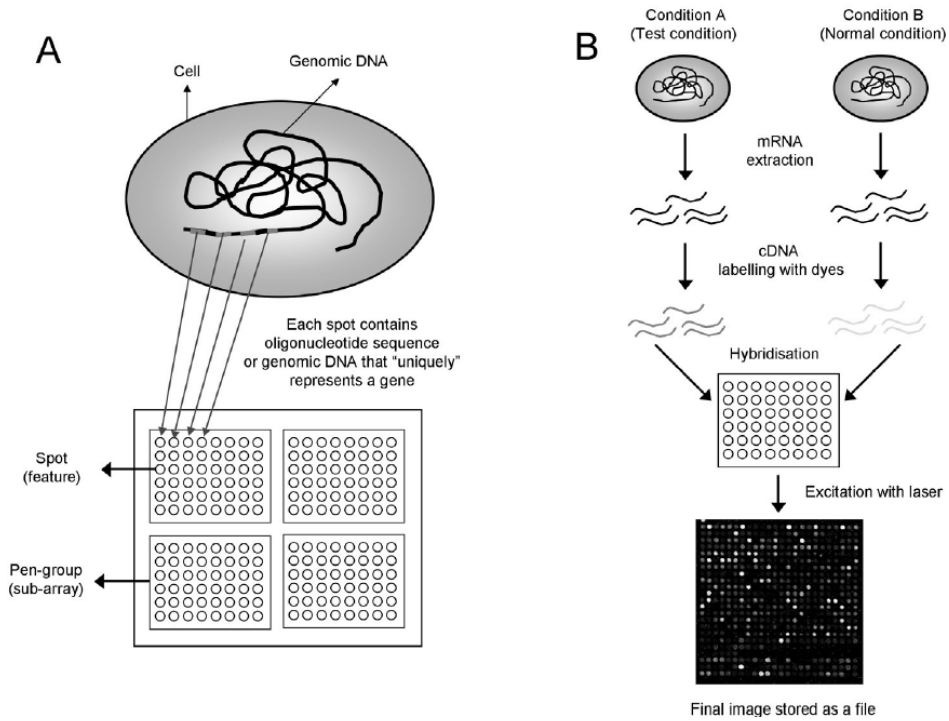
Percobaan *microarray*, mengukur tingkat aktivitas gen di dalam sebuah jaringan sel. Sehingga percobaan ini dapat memberikan informasi berdasarkan aktivitas di dalam jaringan yang bersangkutan. Data ini didapatkan dengan cara mengukur banyaknya *mRNA* yang diproduksi pada saat proses transkripsi DNA, dimana dapat diukur seberapa aktif atau seberapa berfungsinya gen tersebut dalam sebuah jaringan (Elloumi and Zomaya, 2011). Karena kanker berhubungan dengan berbagai macam aktivitas penyimpangan regulasi pada sel, maka data ekspresi gen pada kanker merefleksikan penyimpangan regulasi tersebut. Untuk menangkap keabnormalan ini, percobaan *microarray*, dimana dapat mengukur secara simultan dari level ekspresi ratusan bahkan ribuan ekspresi gen, dapat digunakan untuk mengidentifikasi kanker. Percobaan *microarray* sering dipakai untuk membandingkan profil ekspresi gen pada sel yang terkena kanker, dibandingkan dengan sel yang normal pada berbagai macam percobaan. Percobaan *microarray* digunakan untuk mengidentifikasi ekspresi yang berbeda pada dua percobaan, yang biasanya berupa data tes dan data kontrol (Elloumi and Zomaya, 2011).

Ada 23.6% fungsi gen yang belum diketahui kegunaannya sampai saat ini, hal ini merupakan tantangan pada saat dilakukan proses pengenalan penyakit yang diderita oleh pasien. Sebab ada kemungkinan gen yang sangat berpengaruh terhadap identifikasi penyakit, tetapi masih belum diketahui fungsinya. Oleh karena itu, pada proses klasifikasi penyakit dengan menggunakan machine learning, sering digunakan pengenalan secara *unsupervised learning* (Häggström, 2014).



Gambar 2.1: Ada 23,6% dari keseluruhan fungsi gen yang belum diketahui, sehingga pengetahuan tentang fungsi gen masih belum lengkap. (Häggström, 2014)

Data ekspresi gen yang masih mentah didapatkan dari percobaan di laboratorium menggunakan alat yang dinamakan dengan alat Genchip microarray. Data tersebut kemudian dilakukan pemrosesan awal untuk mendapatkan sebuah matriks ekspresi gen. Matriks ini memiliki data kolom dan baris, dimana kolom berisi data eksperimen, dan baris berisi nilai ekspresi pada tiap-tiap gen (Gambar 2.3) (Babu, 2004).



Gambar 2.2: Proses Keseluruhan Percobaan Microarray.(Yoon et al., 2006)

Pengukuran microarray direspresentasikan dengan tabel gen ekspresi, dimana bagian barisnya adalah fitur ekspresi gen, dan bagian kolom merepresentasikan

pasien.

Table 1.A: Absolute measurement

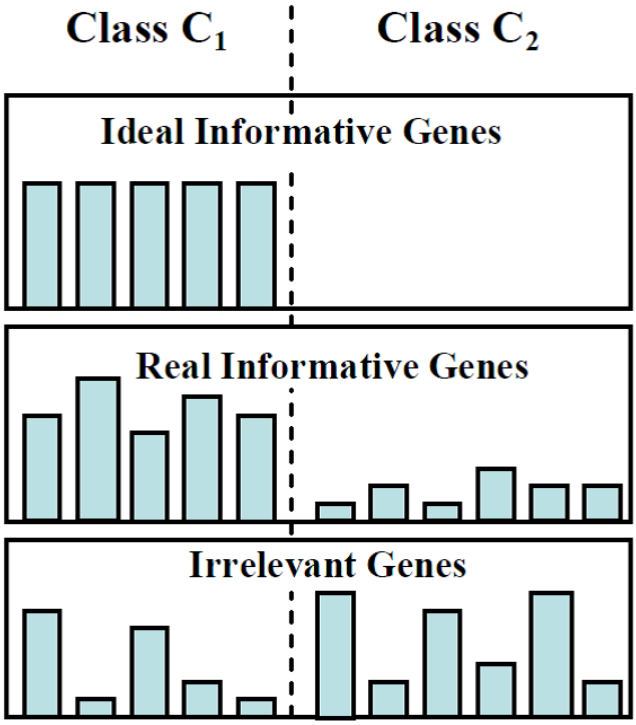
	C1	C2	C3	C4
Gene A	10	80	40	20
Gene B	100	200	400	200
Gene C	30	240	60	60
Gene D	20	160	80	80

Table 1.B: Relative measurement

	C1/C4	C2/C4	C3/C4
Gene A	0.50	4.00	2.00
Gene B	0.50	1.00	2.00
Gene C	0.50	4.00	1.00
Gene D	0.25	2.00	1.00

Gambar 2.3: Contoh data pengukuran percobaan microarray (Yoon et al., 2006)

Karena data microarray yang didapatkan dapat mencapai ribuan ekspresi dalam satu waktu secara simultan, maka data ini dapat sangat membantu dalam mengidentifikasi penyakit. Akan tetapi, hasil yang didapat dengan menganalisa beberapa data microarray yang dilakukan oleh dua percobaan yang berbeda tetapi dengan tujuan yang sama, dapat menghasilkan hasil yang sangat berbeda. Salah satu alasannya adalah terbatasnya sampel dan terlalu banyaknya profil ekspresi gen. Sehingga diperlukan metode testing statistik untuk memastikan bahwa data microarray tersebut memiliki tingkat signifikansi yang cukup, dan dipastikan bahwa perbedaan tersebut memang karena eksperimen, bukan karena kerusakan alat atau kesalahan prosedur eksperimen.



Gambar 2.4: Perbandingan Ekspresi gen yang relevan dan informatif dibandingkan dengan gen yang tidak relevan(Babu, 2004)

2.2 Pemrosesan Data Microarray

Data yang dihasilkan dari alat *microarray* ini berupa citra yang perlu diproses lebih lanjut. Sebelum data ekspresi gen dapat dianalisa lebih lanjut, perlu dilakukan pemrosesan awal yang berupa (i) perbaikan background, (ii) normalisasi data dan kemudian (iii) penyaringan data (iv) imputasi nilai yang hilang dan (v) seleksi fitur.

1. **Perbaikan Background**
   Perbiakan background ini ditujukan untuk menghilangkan titik-titik *noise* yang tidak berasal dari proses hibridisasi. Metode untuk perbaikan background ini adalah salah satu teknik yang banyak diajukan dalam penelitian(Fakoor et al., 2013).

2. **Normalisasi**
   Tujuan dari normalisasi adalah untuk mengatur bias yang dihasilkan oleh variasi proses percobaan microarray. Metode normalisasi data microarray ada banyak, dan pada penelitian ini akan digunakan normalisasi standar untuk data microarray.

3. **Penyaringan data** Tidak semua data yang didapat dari percobaan microarray bagus, kadangkala terjadi kesalahan alat dan noise yang diakibatkan oleh alat,

oleh karena itu perlu disaring, mana data yang disebabkan oleh proses biologi, dan mana yang disebabkan oleh noise alat.

4. **Imputasi Nilai yang Hilang**
   Tidak semua data ekspresi gen dapat kita dapatkan, dikarenakan rumitnya percobaan *microarray*, kadangkala data tidak kita dapatkan, oleh sebab itu diperlukan metode untuk melakukan pendekatan statistik dalam memberikan perkiraan isi data dalam titik data yang hilang tersebut.

5. **Seleksi Fitur**
   Setelah proses diatas, diperlukan teknik untuk menseleksi fitur pada data microarray. Ada banyak metode yang sudah diusulkan oleh para peneliti. Seperti pada tabel 2.1 dibawah. Dan pada titik inilah penelitian ini dijalankan.

2.3 Ekstraksi Fitur dan Seleksi Fitur Pada Penelitian Sebelumnya

Pada tabel dibawah ditunjukkan perbandingan penelitian-penelitian ekstraksi fitur dengan menggunakan berbagai macam metode. Pada penelitian-penelitian sebelumnya, kebanyakan menggunakan metode statistik dan pembelajaran mesin yang dilakukan adalah *supervised*, yaitu memiliki target. Seperti yang dilakukan oleh (Aliferis et al., 2003), Ramaswamy et al. (2001). Sedangkan percobaan *microarray* yang memiliki target, memiliki kelemahan, yaitu tidak semua target fitur gen diketahui kegunaannya. Oleh karena itu pendekatan *unsupervised* dianggap lebih cocok untuk permasalahan seleksi fitur data *microarray* (Häggström, 2014).

Tabel 2.1: Perbandingan Metode Seleksi fitur pada dataset microarray

Pengarang	Judul Paper	Metode	Dataset
C. Aliferis et al. 2003	Machine learning models for classication of lung cancer and selection of genomic markers using array gene expression data.	Reduksi fitur secara rekursif dan melakukan filter secara asosiasi univariate	Lung Cancer Microarray
Ramaswamy, S. et al. 2001	Multiclass cancer diagnosis using tumor gene expression signatures.	Pengurangan fitur secara rekursif dengan mengguanakan SVM	Various Microarray
Wang et al., 2005	Gene-expression proles to predict distant metastasis of lymph-node-negative primary breast cancer.	Mengkombinasikan seleksi fitur yang berbasis korelasi dengan pendekatan assosiasi.	Various Microarray
Sharma et. Al, 2012	Combining multiple approaches for gene microarray classification.	Mengkombinasikan banyak pendekatan ekstraksi fitur	Various Microarray

2.4 Deep Learning

Sebelum tahun 2006, melakukan training dalam arsitektur *deep learning* selalu gagal. Percobaan untuk melakukan training dengan *feedforward neural network* memiliki hasil yang lebih buruk dibandingkan dengan arsitektur yang dangkal, yaitu arsitektur dengan layer 1 atau maksimum 2 layer.
Akan tetapi tiga paper yang terbit pada 2006 secara revolusioner telah merubah hal tersebut. Sehingga setelah tahun 2006 penelitian tentang *deep learning* menjadi lebih intensif sampai sekarang dengan segala variasi arsitekturnya. Salah satu variasi arsitektur *deep learning* yang dipakai dalam thesis ini adalah *arsitektur Deep Belief Network (DBN)*. Ketiga paper tersebut adalah Hinton et al. (2006), Bengio et al. (2007) dan Poultney et al. (2006).

Learning secara *unsupervised* menggunakan *pretraining* secara tiap layer yang disebut dengan *greedy layer-wise training*, yaitu training dilakukan satu layer pada tiap satu waktu. Training ini dilakukan secara berjenjang pada layer selanjutnya. Kemudian dilakukan *supervised training* untuk melakukan *tuning parameter*, yang dimulai dari parameter hasil pretraining yang dilakukan sebelumnya.
DBN menggunakan *Restricted Boltzmann Machine (RBM)* sebagai bagian terkecil

dari layernya, yang menggunakan learning secara unsupervised yang merepresentasikan tiap layer. Sejak 2006, banyak sekali paper-paper yang mulai melakukan eksplorasi tentang deep learning ini, sehingga sejak saat itu deep learning merupakan salah satu teknik *machine learning* yang paling populer, bahkan sampai saat ini (Fakoor et al., 2013).

2.5 Energy-Based Model (EBM) Adalah Bentuk General dari Restricted Boltzman Machine (RBM)

EBM mengaitkan sebuah energi skalar pada setiap konfigurasi variable yang diinginkan. Proses learning bertujuan untuk memodifikasi fungsi energi sehingga bentuknya memiliki sifat yang diinginkan. Sebagai contoh, misalnya diinginkan sebuah bentuk konfigurasi yang memiliki energi yang rendah, maka model probabilistik dari EBM didifinisikan sebagi distribusi probabilitas melalui fungsi energi sebagi berikut: (Poultney et al., 2006)

$$p(x) = \frac{e^{-E(x)}}{Z}. \tag{2.1}$$

Z adalah faktor normalisasi yang disebut sebagai fungsi partisi untuk menganalogikan dengan sistem fisika.

$$Z = \sum_{x} e^{-E(x)} \tag{2.2}$$

EBM bisa dilatih dengan cara melakukan (stochastic) gradient descent pada negative log-likelihood (NLL)-nya secara empiris pada data training. Adapun untuk logistic regression akan didifinisikan terlebih dahulu log-likelihood $\mathcal{L}(\theta, \mathcal{D})$ dan fungsi loss-nya sebagai NLL $\ell(\theta, \mathcal{D})$ sebagai berikut:

$$\begin{aligned} \mathcal{L}(\theta, \mathcal{D}) &= \frac{1}{N} \sum_{x^{(i)} \in \mathcal{D}} \log\ p(x^{(i)}) \\ \ell(\theta, \mathcal{D}) &= -\mathcal{L}(\theta, \mathcal{D}) \end{aligned} \tag{2.3}$$

Menggunakan stochastic gradient $-\frac{\partial \log p(x^{(i)})}{\partial \theta}$, dimana θ adalah parameter dari modelnya(Poultney et al., 2006).

2.5.1 EBM dengan Hidden Units

Pada banyak kasus, sampel x biasanya tidak terobservasi secara penuh, atau akan ditambahkan variabel yang tidak terobservasi secara langsung yang disebut dengan hidden unit, dimana hal ini berguna untuk meningkatkan ekspresivitas dari model. Sehingga dikenalkan bagian yang terobservasi disini dilambangkan dengan x, dan sebuah bagian yang tersembunyi dilambangkan dengan h. Sehingga bisa ditulis sebagai:

$$P(x) = \sum_h P(x,h) = \sum_h \frac{e^{-E(x,h)}}{Z}. \tag{2.4}$$

Pada kasus ini, untuk melakukan pemetaan rumus yang mirip dengan rumus 2.4 , akan dikenalkan notasi (yang merupakan inspirasi dari fisika) yaitu free energy $\mathcal{F}(x)$, yang didifinisikan sebagai berikut:

$$\mathcal{F}(x) = -\log \sum_h e^{-E(x,h)} \tag{2.5}$$

Sehingga bisa diturunkan sebagai :

$$P(x) = \frac{e^{-\mathcal{F}(x)}}{Z} \text{ dengan } Z = \sum_x e^{-\mathcal{F}(x)}.$$

Data dari gradien NLL kemudian memiliki bentuk yang menarik yaitu:

$$-\frac{\partial \log p(x)}{\partial \theta} = \frac{\partial \mathcal{F}(x)}{\partial \theta} - \sum_{\tilde{x}} p(\tilde{x}) \frac{\partial \mathcal{F}(\tilde{x})}{\partial \theta}. \tag{2.6}$$

Gradien diatas memiliki dua istilah, dimana hal tersebut mereferensikan pada fase positif dan fase negatif. Istilah positif dan negatif ini tidak merujuk pada tanda (positif/negatif) persamaan, akan tetapi merefleksikan efek pada kepadatan probabilitas yang didefinisikan oleh model. Istilah pertama, menambah probabilitas data training (dengan cara mengurangi free energy yg berhubungan), sedangkan istilah kedua mengurangi probabilitias sampel yang digenerasi oleh model (Poultney et al., 2006).

Biasanya sulit untuk menentukan gradien secara analitis, oleh karena berhubungan dengan komputasi dari $E_P[\frac{\partial \mathcal{F}(x)}{\partial \theta}]$. Dikarenakan hal ini merupakan ekspektasi semua kemungkinan konfigurasi input x (pada distribusi P yang dibentuk oleh model).
Oleh karena itu, langkah pertama agar bisa dikomputasi secara analitis maka dilakukan estimasi ekspektasi menggunakan jumlah yang pasti dari sampel pada

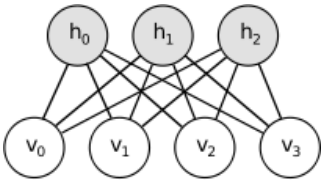
model. Sampel digunakan untuk mengestimasi gradien dari fase negatif yang direferensikan sebagai partikel negatif, dimana disimbolkan sebagai $\mathcal{N}$. Kemudian, gradien bisa ditulis sebagai (Poultney et al., 2006) :

$$-\frac{\partial \log p(x)}{\partial \theta} \approx \frac{\partial \mathcal{F}(x)}{\partial \theta} - \frac{1}{|\mathcal{N}|} \sum_{\tilde{x} \in \mathcal{N}} \frac{\partial \mathcal{F}(\tilde{x})}{\partial \theta}. \tag{2.7}$$

Dimana secara ideal, elemen seperti $\tilde{x}$ dari $\mathcal{N}$ disampel menurut P (sebagai contoh adalah menggunakan teknik sampling Monte-Carlo). Dengan rumus diatas, secara praktis hampir bisa melakukan algoritma stochastic, hanya saja partikel negatif $\mathcal{N}$ belum bisa diekstraksi. Oleh karena itu, pada literatur dengan metode Markov Chain Monte Carlo, sangat bagus digunakan pada model Restricted Boltzmann Machine (RBM) yang merupakan bentuk spesifik dari model EBM (Tutorial, 2014).

2.6 Restricted Boltzmann Machine

Boltzmann Machines (BMs) adalah bentuk khusus dari log-linear Markov Random Field (MRF), dengan kata lain, dimana fungsi energi adalah linear pada parameter bebasnya. Agar membuat BM cukup bisa merepresentasikan distribusi yang kompleks(dengan kata lain, berangkat dari setting parameter yang terbatas kepada non paramter), diasumsikan bahwa beberapa variabel tidak terobserbasi sehingga disebut hidden. Dengan memiliki variabel hidden, bisa dilakukan peningkatan kapasitas model dari BM. RBM, selanjutnya membuat BM yang terbatas pada variabel tanpa koneksi visibel-visibel dan hidden-hidden. Seperti pada gambar 2.5 (Hinton et al., 2006)



Gambar 2.5: Grafik yang Menggambarkan RBM

Fungsi energi $E(v,h)$ pada RBM didefinisikan sebagai persamaan 2.8.

$$E(v,h) = -b'v - c'h - h'Wv \tag{2.8}$$

Dimana W merepresentasikan bobot yang terkoneksi antara unit hidden dan visible dan b, c adalah bias dari visible dan hidden secara berurutan.

Hal ini bisa diterjemahkan dalam bentuk persamaan energi bebas $\mathcal{F}(v)$ seperti dibawah:

$$\mathcal{F}(v) = -b'v - \sum_i \log \sum_{h_i} e^{h_i(c_i + W_i v)}.$$

Dikarenakan struktur RBM yang spesifik, visibel dan hidden adalah independen secara bersyarat antara satu dengan lainnya. Dengan menggunakan sifat tersebut, maka dapat dituliskan :

$$p(h|v) = \prod_i p(h_i|v)$$
$$p(v|h) = \prod_j p(v_j|h).$$

2.6.1 RBMs yang Menggunakan Unit Biner

Kasus umum jika menggunakan unit biner (dimana v_j dan $h_i \in \{0,1\}$), yang didapat dari persamaan (6) dan (2), versi probabilistik dari fungsi aktivasi neuron adalah sebagai berikut(Hinton and Salakhutdinov, 2006):

$$P(h_i = 1|v) = sigm(c_i + W_i v) \tag{2.9}$$

$$P(v_j = 1|h) = sigm(b_j + W'_j h) \tag{2.10}$$

Selanjutnya, energi bebas dari RBM dengan unit biner, disederhanakan menjadi persamaan:

$$\mathcal{F}(v) = -b'v - \sum_i \log(1 + e^{(c_i + W_i v)}). \tag{2.11}$$

2.6.2 Update Persamaan dengan Unit Biner

Menghubungkan persamaan (5) dengan (9), didapatkan gradien log-likelihood untuk RBM dengan unit biner sebagai berikut:

$$\begin{aligned}
-\frac{\partial \log p(v)}{\partial W_{ij}} &= E_v[p(h_i|v) \cdot v_j] - v_j^{(i)} \cdot sigm(W_i \cdot v^{(i)} + c_i) \\
-\frac{\partial \log p(v)}{\partial c_i} &= E_v[p(h_i|v)] - sigm(W_i \cdot v^{(i)}) \\
-\frac{\partial \log p(v)}{\partial b_j} &= E_v[p(v_j|h)] - v_j^{(i)}
\end{aligned} \tag{2.12}$$

2.7 Sampling pada RBM

Sampel dari $p(x)$ bisa didapat dengan menjalankan Markov chain sampai konvergen dengan menggunakan gibbs samping sebagai operator transisi.

Gibbs sampling dari join variable random sebanyak N dari $S = (S_1, ..., S_N)$ merupakan urutan sebanyak N sampling dari sub-steps dalam bentuk $S_i \sim p(S_i|S_{-i})$ dimana S_{-i} berisi $N-1$ variabel random lain didalam S tetapi diluar S_i.

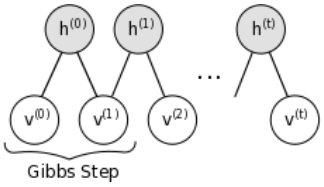
Untuk RBM, S berisi himpunan dari visible dan hidden unitnya. Akan tetapi, dikarenakan unit ini dipenden secara kondisional, maka salah satunya bisa dilakukan gibbs sampling. Pada setting disini, unit visible disampel secara simultan given nilai fix dari hidden unitnya. Demikian sebaliknya, hidden unitnya disampel secara simultan given unit visibelnya.Sehingga satu langkah Markov chain adalah sebagai berikut:

$$h^{(n+1)} \sim sigm(W'v^{(n)} + c)$$

$$v^{(n+1)} \sim sigm(Wh^{(n+1)} + b),$$

Dimana $h^{(n)}$ menunjik pada himpunan semua hidden unit pada nilai yang ke-n langkah dari Markov chain. Yang artinya adalah sebagai contoh, $h_i^{(n+1)}$ adalah secara random dipilih antara 1 (versus 0) dengan nilai probabilitas $sigm(W_i'v^{(n)} + c_i)$, demikian juga, $v_j^{(n+1)}$ adalah dipilih secara random antara 1 (versus 0) dengan probabilitas $sigm(W_{\cdot j}h^{(n+1)} + b_j)$.

Hal ini seperti digambarkan pada gambar 2.6



Gambar 2.6: Gibbs Sampling

Oleh karena $t \to \infty$, maka sampel $(v^{(t)}, h^{(t)})$ bisa dipastikan akan akurat dalam mensampel $p(v,h)$.

Secara teori, tiap parameter diupdate pada proses learning dibutuhkan satu rantai tersebut untuk konvergen. Akan tetapi hal ini sangat mahal komputasinya. Sehingga banyak diajukan algoritma untuk melatih RBM agar sampel $p(v,h)$ efisien, disaat proses learningnya.

2.8 Contrastive Divergence (CD-k)

Contrastive Divergence(CD) menggunakan trik untuk mempercepat proses sampling: Dikarenakan yang diinginkan adalah $p(v) \approx p_{train}(v)$ (distribusi data yang asli), initialisasi Markov chain dengan contoh data training (dimana, berasal dari distribusi yang mendekati p, pada distribusi final dari p). CD tidak menunggu rantai untuk konvergen. Sampel didapatkan setalah langkah ke-k dari Gibbs sampling. Pada prakteknya, $k = 1$ sudah menghasilkan hasil yang baik.

2.9 Persistent CD

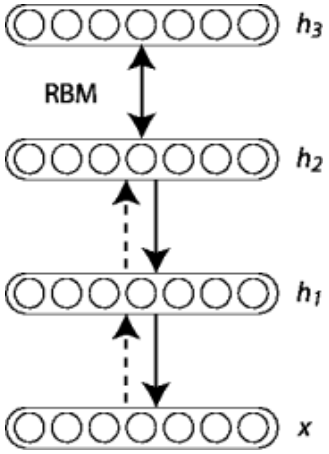
Persistent CD (P-CD) [Tieleman08] menggunakan pendekatan lain untuk mensampling $p(v,h)$. Hal ini bergantung hanya pada Markov chain tunggal, yang memiliki kondisi yang persisten (dimana, tidak melakukan restart chain pada setiap sampel yang terobservasi). Pada setiap umpdate parameter, akan di ekstraksi sampel baru dengan penjalankan chain pada langkah ke-k. Kondisi chain akan dipertahankan pada update selanjutnya.
Intuisinya adalah jika update parameternya cukup kecil dibaningkan dengan rate campuran dari Markov Chain, maka hal ini bisa mengejar perubahan modelnya.

2.10 Deep Belief Network

Hinton et al. (2006) menunjukkan bahwa RBM bisa dijajar dan dilatih secara greedy untuk membentuk sebuah jaringan yang dinamakan dengan *Deep Belief Network (DBN)*. DBN adalah model grafis dimana bisa melakukan learning untuk mengekstraksi representasi hirarki yang mendalam (deep) dari data training. Hal ini memodelkan distribusi gabungan antara vektor x sebagai observer dan ℓ layer hidden h^k sebagai berikut:

$$P(x,h^1,\ldots,h^\ell) = \left(\prod_{k=0}^{\ell-2} P(h^k|h^{k+1})\right) P(h^{\ell-1},h^\ell) \tag{2.13}$$

Dimana $x = h^0, P(h^{k-1}|h^k)$ adalah distribusi kondisional untuk unit visible dikondisikan pada unit hidden pada level k dan $P(h^{\ell-1},h^\ell)$ adalah distribusi gabungan visible-hidden pada level teratas dar RBM. Seperti diilustrasikan pada gambar 2.7.



Gambar 2.7: Arsitektur Deep Belief Network (DBN) yang merupakan gabungan dari RBM yang dibuat bertingkat

Prinsip dari *greedy layer-wise unsupervised training* bisa di aplikasikan pada DBN dengan RBM sebagai bagian pada tiap layernya [hinton] [bengio]. Pada prinsipnya prosesnya adalah sebagai berikut:

1. Latih layer pertama ssebagai RBM yang memodelkan input $x = h^{(0)}$ sebagai visible layernya.

2. Gunakan layer pertama untuk mendapatkan representasi input yang digunakan sebagai data untuk layer kedua. Ada dua solusi yang sama. Representasi ini bisa dipilih sebagai rata-rata dari aktivasi $p(h^{(1)} = 1|h^{(0)})$ atau sampel dari $p(h^{(1)}|h^{(0)})$.

3. Train layer kedua sebagai RBM dengan mengambil data transformasi (sampel atau rata-rata aktivasi) sebagai training (untuk layer visible dari RBM tersebut).

4. Iterasikan (2 dan 3) untuk semua layer yang diinginkan, setiap waktu dengan mempropagasikan keatas antara sampel atau nilai rata-ratanya.

5. Fine-tune semua parameter dari arsitektur dengan log-likelihood DBN atau dengan kriteria secara supervised setelah menambahkan layer supervised untuk memprediksikan kelas, sebagai contoh misalnya layer logistic regression.

Pada kasus ini, akan difokuskan pada fine-tuning dengan melakukan gradien descent menggunakan klassifier logistic regression dimana digunakan untuk mengklasifikasikan input x berdasar pada output dari hidden layer $h^{(l)}$ dari DBN. Fine-tune kemudian dilakukan melalui gradien descent dari NLL fungsi costnya. Dikarenakan gradien secara supervised adalah hanya non-null untuk bobot dan bias pada hidden

layer pada tiap-tiap layer, maka prosedur ini serupa dengan menerapkan initialisasi parameter dari arsitektur MLP yang deep dengan bobot dan bias dari hidden layer yang didapat pada proses training unsupervised diatas.

2.11 Cost

Cost merupakan variabel yang menggambarkan *Negative Log Likelihood*. Yang memiliki bentuk persamaan sebagai berikut:

$$\frac{1}{|\mathcal{D}|}\mathcal{L}(\theta=\{W,b\},\mathcal{D})=\frac{1}{|\mathcal{D}|}\sum_{i=0}^{|\mathcal{D}|}\log(P(Y=y^{(i)}|x^{(i)},W,b))\ell(\theta=\{W,b\},\mathcal{D}) \quad (2.14)$$

Melakukang *learning* pamameter model dengan cara meminimalisasi *Lost Function*. Sangat umum digunakan minimisasi *negative log likelihood (NLL)* ini yang ekivalen dengan memaksimalkan likelihood dari data set $\mathcal{D}$ pada model yang diparameterkan oleh θ. Definisi dari likelihood $\mathcal{L}$ dan loss ℓ maka:

$$\begin{aligned}\mathcal{L}(\theta=\{W,b\},\mathcal{D})&=\sum_{i=0}^{|\mathcal{D}|}\log(P(Y=y^{(i)}|x^{(i)},W,b))\\ \ell(\theta=\{W,b\},\mathcal{D})&=-\mathcal{L}(\theta=\{W,b\},\mathcal{D})\end{aligned} \quad (2.15)$$

Untuk meminimisasi, digunakan *stochastic gradient descen with minibatches (MSGD)* (Hinton et al., 2006).

Dalam kode python dituliskan:

cost = classifier.negative_log_likelihood(y)

Semakin kecil cost, menunjukkan semakin kecil error rekonstruksinya. Hal ini menunjukkan bahwa, data rekonstruksi mendekati bentuk data konstruksinya (diambil dari data training).

2.12 Training Secara Greedy Layer-Wise

Algoritma training deep learning secara greedy layer-wise terbukti bisa bekerja dengan baik, sebagai contoh 2 layer DBN dengan hidden layer $h^{(1)}$ dan $h^{(2)}$ dengan parameter bobot berurutan adalah $W^{(1)}$ dan $W^{(2)}$, (Hinton and Salakhutdinov, 2006) maka $\log p(x)$ bisa ditulis sebagai:

$$\begin{aligned}\log p(x)=&KL(Q(h^{(1)}|x)||p(h^{(1)}|x))+H_{Q(h^{(1)}|x)}+\\ &\sum_{h}Q(h^{(1)}|x)(\log p(h^{(1)})+\log p(x|h^{(1)})).\end{aligned} \quad (2.16)$$

$KL(Q(h^{(1)}|x)||p(h^{(1)}|x))$ merepresentasikan KL divergence antara posterior $Q(h^{(1)}|x)$ dari RBM pertama jika hal ini sendirian, dan probabilitas $p(h^{(1)}|x)$ untuk layer sayng sama tapi didifinisikan oleh keseluruhan DBN (sebagai contoh, perhitungan prior $p(h^{(1)},h^{(2)})$ didefinisikan sebagai top-level RBM). $H_{Q(h^{(1)}|x)}$ adalah entropy dari distribusi $Q(h^{(1)}|x)$.
Hal ini bisa ditunjukkan bahwa jika diinitialisasi kedua layer hidden sehingga $W^{(2)} = W^{(1)^T}, Q(h^{(1)}|x) = p(h^{(1)}|x)$ dan KL divergence nya adalah null. Maka jika di lakukan learning pada level awal RBM dan kemudian parameter $W^{(1)}$ dibuat tetap, kemudian dilakukan optimasi pada persamaan 2.15 terhadap $W^{(2)}$ bisa meningkatkan likelihood dari $p(x)$. Jika diisolasi hanya pada $W^{(2)}$ sehinggi didapatkan:

$$\sum_h Q(h^{(1)}|x)p(h^{(1)})$$

Melakukan optimasi persamaan ini dengan memperhatikan jumlah $W^{(2)}$ training pada tingkat RBM selanjutnya, menggunakan output dari $Q(h^{(1)}|x)$ sebagai distribusi training untuk RBM yang pertama.

2.13 Logistic Regression

Logistic Regression adalah salah satu klassifier yang paling dasar pembentuk dari MLP. Penjelasannya akan dimulai dari bentuk model dasarnya serta notasi matematisnya.

2.13.1 Model Logistic Regression

Logistic regression adalah klasifier yang linear dan probabilistik. Diparameterkan dengan matrik bobot W dan vektor bias b. Proses klasifikasinya adalah dengan cara memproyeksikan vektor input kedalam himpunan *hyperplane*, dimana berkorespondensi pada kelasnya. Jarak dari input ke *hyperplane* merefleksikan probabilatas dari input adalah berkorespondensi dari anggota kelasnya.
Secara matematis, probabilitas vektor input x adalah anggota dari kelas i, isi dari variabel *stochastic* Y, bisa ditulis sebagai berikut:

$$\begin{aligned} P(Y = i|x,W,b) &= softmax_i(Wx+b) \\ &= \frac{e^{W_i x+b_i}}{\sum_j e^{W_j x+b_j}} \end{aligned} \tag{2.17}$$

Prediksi dari model berupa y_{pred} adalah kelas dimana probabilitasnya maksimal,

secara spesifik ditulis sebagai:

$$y_{pred} = \text{argmax}_i P(Y = i|x, W, b) \tag{2.18}$$

2.13.2 Mendefinisikan Lost Function dari Logistic Regression

Melakukang *learning* pamameter model dengan cara meminimalisasi *Lost Function*. Pada kasus *logistic regression* yang multi-kelas, sangat umum digunakan minimisasi *negative log likelihood (NLL)* yang ekivalen dengan memaksimalkan likelihood dari data set $\mathcal{D}$ pada model yang diparameterkan oleh θ. Definisi dari likelihood $\mathcal{L}$ dan loss ℓ maka:

$$\begin{aligned} \mathcal{L}(\theta = \{W, b\}, \mathcal{D}) &= \sum_{i=0}^{|\mathcal{D}|} \log(P(Y = y^{(i)}|x^{(i)}, W, b)) \\ \ell(\theta = \{W, b\}, \mathcal{D}) &= -\mathcal{L}(\theta = \{W, b\}, \mathcal{D}) \end{aligned} \tag{2.19}$$

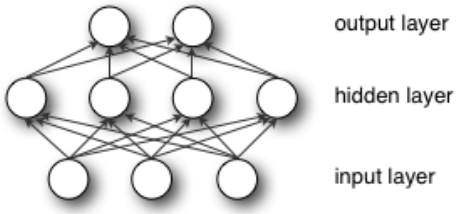
Untuk meminimisasi, digunakan *stochastic gradient descen with minibatches (MSGD)* (Hinton et al., 2006).

2.14 Multi Layer Perceptron

Arsitektur selanjutnya yang akan dibahas adalah *Multi Layer Perceptron (MLP)* Arsitektur MLP ini bisa dilihat sebagai klasifier *Logistic Regression* dimana input pada awalnya ditransformasikan menggunakan transformasi non linear Φ. Transformasi ini memproyeksikan data input kepada *space* dimana hal ini bisa terseparasi secara linear. Layer tengah ini direferensikan sebagai *hidden layer*. Satu hidden layer sebenarnya sudah cukup untuk membuat MLP sebagai aproksimator universal. Akan tetapi, ada banyak keuntungan untuk menggunakan hidden unit yang lebih dari satu layer, hal inilah yang digunakan sebagai konsep dasar dari deep learning. Algoritma untuk melakukan *training* dari MLP yang paling sering dipakai adalah algoritma *back-propagation* (Tutorial, 2014).

2.14.1 Model MLP

MLP atau sering disebut juga dengan Artificial Neural Network (ANN) adalah Perceptron yang dibentuk menjadi sebuah jaringan. MLP dengan layer tunggal bisa direpresentasikan secara grafis seperti pada Gambar 2.8 berikut.



Gambar 2.8: Arsitektur Layer Tunggal MLP

Secara formal, hidden layer tunggal dari MLP adalah sebuah fungsi $f : R^D \rightarrow R^L$, dimana D aadlah ukuran dari vektor input x dan L adalah ukuran dari output vektor $f(x)$ sehingga dengan menggunakan notasi matriks sebagai berikut :

$$f(x) = G(b^{(2)} + W^{(2)}(s(b^{(1)} + W^{(1)}x))), \tag{2.20}$$

Dengan vektor bias $b^{(1)}, b^{(2)}$; dan matrik bobot $W^{(1)}, W^{(2)}$ dan fungsi aktivasinya adalah G dan s. Sedangkan vektor $h(x) = \Phi(x) = s(b^{(1)} + W^{(1)}x)$ merupakan *hidden layer*. Setiap kolom $W_{.i}^{(1)}$ merepresentasikan bobot dari unit input yang ke-i dari *hidden unit*. Pilihan fungsi aktifasinya bisa menggunakan tanh, atau fungsi sigmoid.

$$\begin{aligned} tanh(a) &= \frac{(e^a - e^{-a})}{(e^a + e^{-a})} \\ sigm(a) &= \frac{1}{(1 + e^{-a})} \end{aligned} \tag{2.21}$$

Kedua fungsi aktivasi yaitu tanh dan sigmoid adalah fungsi skalar ke skalar akan tetapi bisa diekstensikan menjadi vektor atau tensor yang diaplikasikan secara *element wise*.

Vektor output didapatkan dengan: $o(x) = G(b^{(2)} + W^{(2)}h(x))$. Probabilitas dari keanggotaan kelas didapat dari memilih G sebagai fungsi *softmax* (untuk kasus klasifikasi multi-kelas).

Untuk melakukan *training* MLP dilakukan *learning* parameter dari model menggunakan *Stochastic Gradien Descent* dengan dibagi menjadi bagian kecil-kecil atau disebut dengan *minibatch*. Himpunan parameter pembelajarkan ditulis sebagai himpungan $\theta = \{W^{(2)}, b^{(2)}, W^{(1)}, b^{(1)}\}$. Mendapatkan gradien $\partial\ell/\partial\theta$ didapatkan dengan menerapkan algoritma *backpropagation* (Tutorial, 2014)

2.15 Metode Bonferroni untuk Evaluasi

Di dalam statistik, testing hipotesis adalah berdasar pada menolak hipotesa 0 apabila kebolehjadian data yang diobservasi dibawah hipotesa 0 adalah rendah. Jika dilakukan perbandingan berganda atau dilakukan pengetesan hipotesa, maka ke-

mungkinan untuk terjadi sebuah peristiwa yang langka menjadi meningkat, oleh karena itu, kebolehjadian untuk menolak hipotesa 0 menjadi meningkat pula (error tipe 1 meningkat). Oleh karena itu dibutuhkan untuk sebuah metode koreksi untuk menjaga agar error tipe I nya bisa dikoreksi.

Metode koreksi Bonferroni adalah berbasis pada ide dimana jika eksperimen dilakukan untuk melakukan testing pada hipotesa sebanyak m, maka untuk memelihara *familywise error rate (FWER)* adalah untuk melakukan testing hipotesis secara individu dengan level signifikansi $1/m$ dikalikan dengan level maksimum keseluruhan yang diinginkan.

Jika level signifikansi yang diinginkan semua anggota dari test adalah α, maka koreksi bonferroni akan melakukan testing secara individual dengan level signifikansinya adalah α/m. Sebagai contoh jika testing percobaan $m = 8$ dengan hipotesa yang diinginkan $\alpha = 0.05$ maka kereksi akan melakukan testing secara individual pada hipotesis pada $\alpha = 0.05/8 = 0.00625$ (Hochberg, 1988)

2.15.1 Definisi

Diberikan $H_1, ..., H_m$ adalah sebuah keluarga hipotesa dan $p_1, ..., p_m$ adalah secara berurutan merupakan *p-value*-nya. FWER adalah probabilitas untuk menolak setidaknya satu dari H_i; sehingga setidaknya ada satu error tipe I. Maka koreksi bonferroni menyatakan bahwa menolak hipotesa null untuk semua $p_i \leq \frac{\alpha}{m}$ yang mengontrol FWER. Dibuktikan dengan :

$$FWER = P\left\{\bigcup_{i=1}^{m_0}\left(p_i \leq \frac{\alpha}{m}\right)\right\} \leq \sum_{i=1}^{m_0}\left\{P\left(p_i \leq \frac{\alpha}{m}\right)\right\} \leq m_0\frac{\alpha}{m} \leq m\frac{\alpha}{m} = \alpha \quad (2.22)$$

Kontrol ini tidak memerlukan asumsi tentang ketergantungan antara p-value-nya (Hochberg, 1988).

BAB 3
METODOLOGI PENELITIAN

Penelitian ini dibagi menjadi empat tahap: (1) Mendapatkan data microarray dan pengolahan awal; (2) Perancangan algoritma; (3) Melakukan eksperimen untuk mendapatkan *hyperparameter* yang optimal. Kemudian dilanjutkan dengan testing dan evaluasi. Gambaran umum dari penelitian ini seperti pada Gambar 3.1

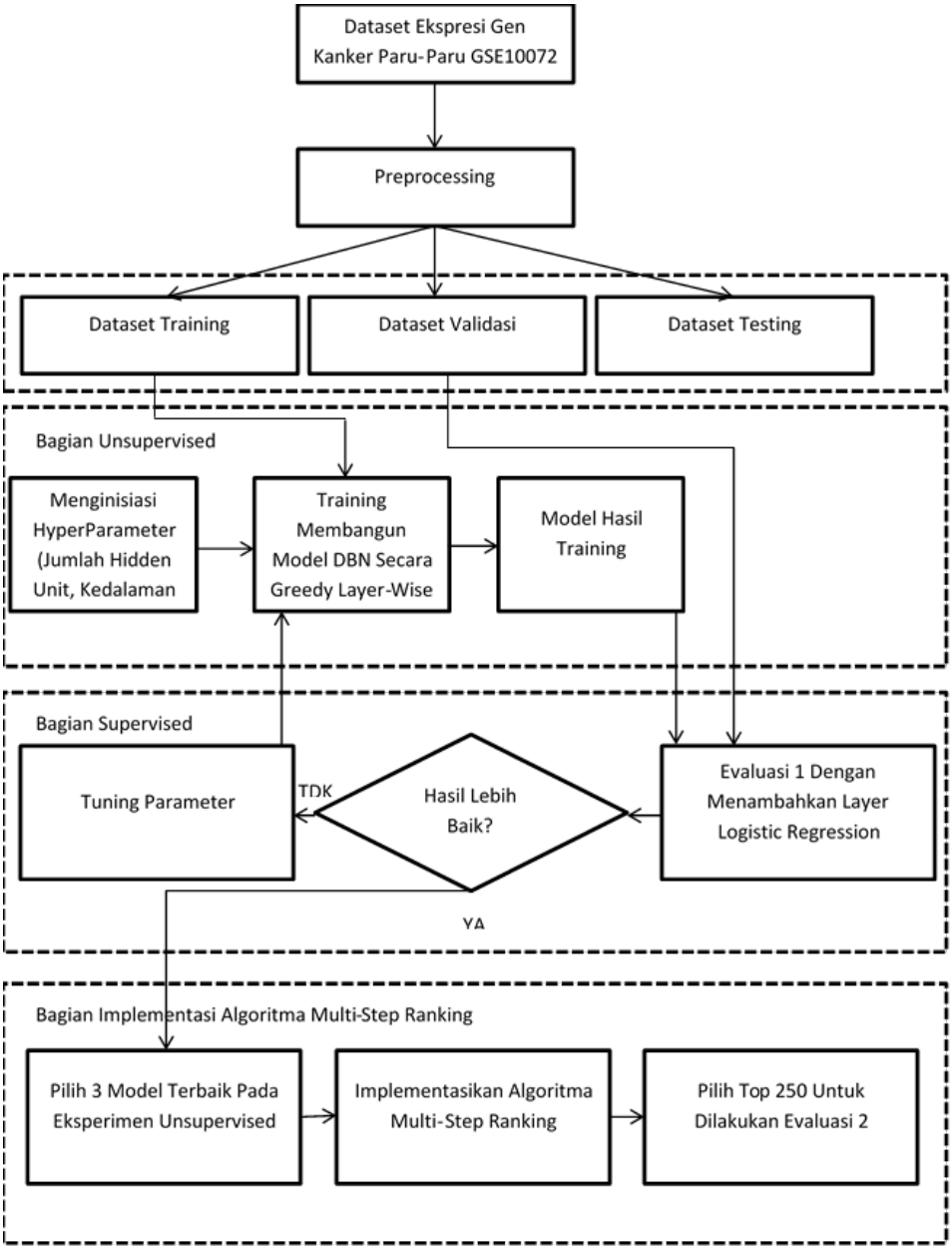
3.1 Gambaran Umum Penelitian

Secara garis besar, penelitian ini dibagi menjadi beberapa tahapan. Yang pertama adalah tahapan persiapan yaitu mendapatkan data *microarray* kemudian mengolahnya menjadi data yang siap untuk dilakukan proses seleksi fitur dan tahapan pelatihan *deep learning*. Yaitu dengan membagi 80% data untuk training, 15% data untuk validasi dan 5% data untuk testing.

Bagian kedua adalah membangun model DBN dengan teknik *unsupervised learning*. Untuk mendapatkan model terbaik secara *greedy* pada tiap-tiap layer RBM-nya. Dimana dilakuan tuning *hyperparameter* (jumlah kedalaman layer, jumlah *hidden unit* pada tiap layernya) digunakan untuk mendapatkan struktur *hyperparameter* yang cocok dengan ciri khas dari data *microarray*. Oleh karena itu diperlukan banyak percobaan untuk mendapatkan hasil yang bagus.

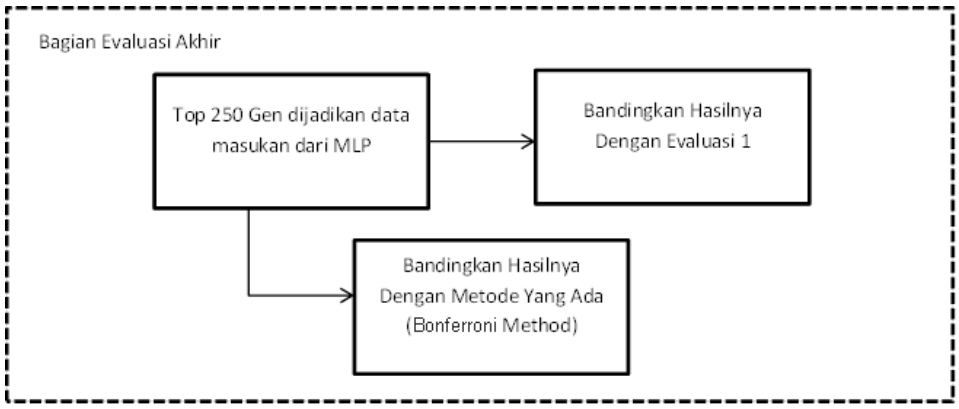
Bagian ketiga, adalah *supervised learning*, dimana merupakan evaluasi sementara dari tahap yang kedua. Dibuat layer output berupa *logistic regression*, yang digunakan untuk menguji sementara hasil dari proses *pretraining* untuk mengklasifikasikan pasien kanker dan pasien normal menggunakan dataset validasi dan dataset testing.

Bagian keempat merupakan bagian yang terpenting karena dimana ide thesis ini dibuat. Yaitu melakukan perankingan gen untuk mencari gen yang paling informatif yang didapatkan dari model pada percobaan sebelumnya. Dimana algoritma seleksi fitur untuk multi-step ranking dijalankan agar didapatkan *biomarker*.



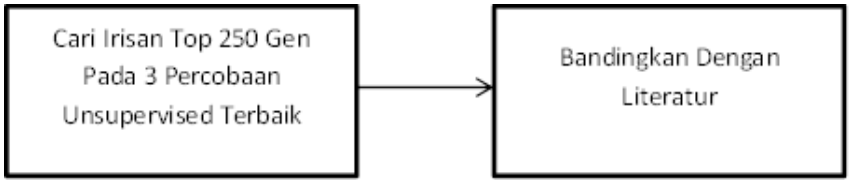
Gambar 3.1: Overview Penelitian

Tahapan terakhir adalah tahap evaluasi akhir, yaitu akan dilakukan dua kali evaluasi, yang pertama evaluasi dengan cara membandingkan evaluasi 1 (*logistic regression* sebelum dilakukan seleksi fitur) dengan evaluasi 2 (MLP setelah dilakukan seleksi fitur). Hasil dari kedua proses ini dibandingkan apakah terjadi perbaikan performa klasifikasinya.



Gambar 3.2: Overview Metode Evaluasi

Untuk evaluasi selanjutnya yaitu dilakukan konfirmasi, dimana hasil dari perankingan gen tersebut dibandingkan dengan penelitian tentang biomarker sebelumnya. Apakah gen biomarker yang ditemukan pada penelitian ini memiliki signifikansi dibandingkan dengan teknik sebelumnya.



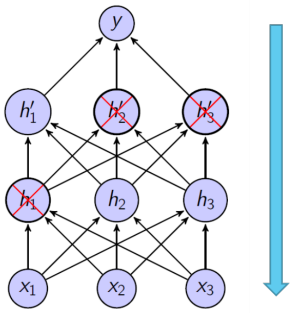
Gambar 3.3: Metode Untuk Mengkonfirmasi Biomarker

3.2 Desain Metode Perangkingan Bobot Secara Multi Step Untuk Mendapatkan Gen Biomarker

Pada penelitian ini, akan dibangun sebuah teknik pencarian *Biomarker* dengan metode seleksi fitur gen. Metode ini menerapkan perankingan gen secara *multi step* terhadap model yang didapatkan pada proses *training* yang dilakukan secara *unsupervised*. Arsitektur untuk mendapatkan modelnya adalah digunakan arsitektur *Deep Belief Network (DBN)* yang merupakan bagian dari metode *deep learning*. Metode perankingan yang digunakan adalah modifikasi dari algoritma seleksi fitur untuk *logistic regression* yang dilakukan oleh Shevade and Keerthi (2003). Akan tetapi metode ini memiliki masalah dalam mengeliminasi fitur jika diterapkan secara langsung pada model DBN, dikarenakan parameter bobot (W) dan bias (b) ditempatkan disetiap fitur dan model ini hanya memiliki satu layer dibandingkan dengan DBN yang memiliki banyak layer.

Pada DBN, *hidden unit* yang paling sering aktif adalah *hidden unit* yang lebih

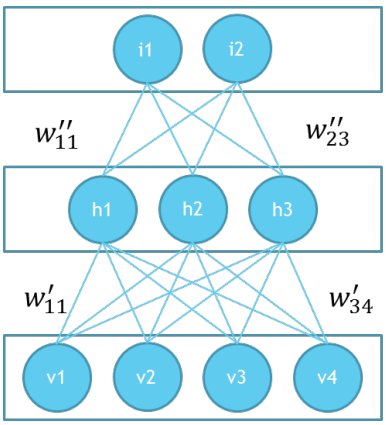
penting dibandingkan dengan unit yang jarang aktif, oleh karena itu *hidden unit* ini memiliki parameter bobot yang lebih besar dibandingkan dengan *hidden unit* yang jarang aktif pada saat proses *training* dilakukan. Pemilihan fitur dilakukan dengan meranking unit-unit yang memiliki bobot tertinggi dimulai dari *layer output* mundur secara multi-step menuju *layer input* untuk mendapatkan fitur gen yang paling berpengaruh terhadap model. Kemudian dilakukan eliminasi bobot pada *hidden unit* per layernya secara *multi step*. Selanjutnya akan dipilih sebanyak *top-n* gen dari hasil perankingan ini untuk dievaluasi apakah *Biomarker* yang ditemukan tersebut informatif atau tidak. Seperti digambarkan pada bagan Gambar 3.4



Gambar 3.4: Hidden unit yang paling sering aktif adalah neuron yang paling penting. Sedangkan yang Kurang Penting Dihapus dengan arah mundur Secara Multi-step (Duh, 2014)

3.2.1 Perhitungan Seleksi Fitur dengan Multi-Step Ranking

Contoh dibawah adalah contoh penyederhanaan dari proses multi-step ranking yang diajukan. Pada prakteknya, *visible unit* dan *hidden unit* memiliki jumlah yang besar. Sebagai contoh, pada kasus data kanker paru-paru yang diteliti ini memiliki fitur 22 ribu gen yang di ukur secara simultan dalam satu percobaan.



Step 1 : Perhitungannya mulai dari layer output (i), mundur sampai layer visible (v) :

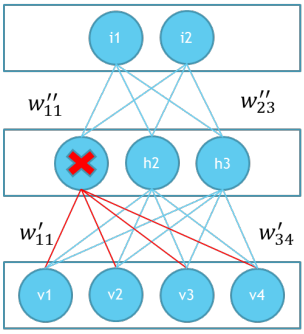
$$i_1 = w''_{11} h_1 + w''_{12} h_2 + w''_{13} h_3$$
$$i_2 = w''_{21} h_1 + w''_{22} h_2 + w''_{23} h_3$$

Ranking :

$$h_1 = w''_{11} + w''_{21}$$
$$h_2 = w''_{12} + w''_{22}$$
$$h_3 = w''_{13} + w''_{23}$$

Kemudian di ranking, misalnya didapatkan h2 dan h3 berturut-turut ranking 1 dan 2, maka h1 dieliminasi, dan bobot h1 dieliminasi untuk proses selanjutnya.

Gambar 3.5: Contoh Perhitungan tahap pertama dimulai dari top hidden unit



Eliminasi Bobot Dengan Ranking Terendah :

$$h_1 = w'_{11}v_1 + w'_{12}v_2 + w'_{13}v_3 + w'_{14}v_4$$
$$h_2 = w'_{21}v_1 + w'_{22}v_2 + w'_{23}v_3 + w'_{24}v_4$$
$$h_3 = w'_{31}v_1 + w'_{32}v_2 + w'_{33}v_3 + w'_{34}v_4$$

Lakukan sampai mencapai visible layer terakhir :

$$v_1 = w'_{11} + w'_{21} + w'_{31}$$
$$v_2 = w'_{12} + w'_{22} + w'_{32}$$
$$v_3 = w'_{13} + w'_{23} + w'_{33}$$
$$v_4 = w'_{14} + w'_{24} + w'_{34}$$

Ranking v untuk mendapatkan biomarker

Gambar 3.6: Contoh Perhitungan tahap pertama dimulai dari top hidden unit

Perhitungan diatas secara iteratif dilakukan mulai dari layer output mundur sampai layer input.

3.2.2 Desain Algoritma Multi-Step Ranking

Input : matrix yang berisi bobot dan bias
Output : Matrix yang berisi index gen dan ranking nya
Seperti pada listing 3.1

Listing 3.1: listing multi step

```
# hsl_ranking = multisteprank(model, KonfigurasiLayer):
ekstraktor = Ekstraktor()
model = InputModel
Wlayer3 = model.rbm_layers[3].W
Wlayer2 = model.rbm_layers[2].W
Wlayer1 = model.rbm_layers[1].W
Wlayer0 = model.rbm_layers[0].W

y3 = Wlayer3.get_value(True)
x3 = T.fmatrix()
x3 = y3.copy()

# ranking ujung (Layer 3)
awal3 = mtr.awal(x3)
```

```
15 jml_bobot3 = mtr.jumlah_bobot(x3, awal3)
16 ranking_jml_bobot3 = mtr.rank_hasil_jumlah(jml_bobot3)
17 top_n3 = mtr.set_top_n(ranking_jml_bobot3,70)
18
19 # ranking layer selanjutnya sampai layer = 0
20 y2 = Wlayer2.get_value(True)
21 x2 = y2.copy()
22 awal2 = mtr.extract_top_n(top_n3)
23 jml_bobot2 = mtr.jumlah_bobot(x2, awal2)
24 ranking_jml_bobot2 = mtr.rank_hasil_jumlah(jml_bobot2)
25 top_n2 = mtr.set_top_n(ranking_jml_bobot2,700)
26
27 y1 = Wlayer1.get_value(True)
28 x1 = y1.copy()
29 awal1 = mtr.extract_top_n(top_n2)
30 jml_bobot1 = mtr.jumlah_bobot(x1, awal1)
31 ranking_jml_bobot1 = mtr.rank_hasil_jumlah(jml_bobot1)
32 top_n1 = mtr.set_top_n(ranking_jml_bobot1,1500)
33
34 y0 = Wlayer0.get_value(True)
35 x0 = y0.copy()
36 awal0 = mtr.extract_top_n(top_n1)
37 jml_bobot0 = mtr.jumlah_bobot(x0, awal0)
38 ranking_jml_bobot0 = mtr.rank_hasil_jumlah(jml_bobot0)
```

3.3 Implementasi Metode Perangkingan Bobot Secara Multi Step Untuk Mendapatkan Gen Biomarker

Implementasi multi-step ranking dengan menggunakan python:

Listing 3.4 : Implementasi Multi-Step Ranking di python

```
1
2  # perkalian matrix rank weight
3  import numpy as np
4
5  def awal(w):
6      return np.ones((w.shape[1],), dtype=np.float)
7
8  def jumlah_bobot(w, top_ke_n):
9      # kalikan w dengan matrix 1
10     return w.dot(top_ke_n)
11
12 def rank_hasil_jumlah(sum_w):
13     # urutkan sum_w dan beri index
14     """
```

```
15
16      :rtype sum_w : numpy.array
17      """
18      swi = sum_w.shape[0]
19      hsl = np.arange(swi)
20      c = np.concatenate((hsl,sum_w))
21      c = c.reshape(2,swi)
22      c = c.T
23      z = c[c[:,1].argsort()[::-1]] # urutkan descending berdasarkan bobot (indeks mengikuti)
24      return z
25
26  def set_top_n(idx_sum_w, top_n = 2):
27      # set = 0 semua yang bukan top n
28      # kembalikan ke urutan semula
29      z = idx_sum_w.copy()
30      z[top_n:,1] = 0.
31      z[0:top_n,1]= 1.
32      # print 'z adalah'
33      # print z
34      d = z[z[:,0].argsort()[::]]
35      # print 'd adalah'
36      # print d
37      return d
38
39  # set_rank : melakukan setting 1 untuk top n dan
40  def extract_top_n(n):
41      return n[:,1]
42
43  def set_index_dengan_gen(bobot_akhir):
44      # index gen dengan urutan perankingannya
45      pass
46
47  def plot_diagram(a, b):
48      # plot himpunan a dan b dan anggota keduanya
49      pass
50
51
52  if __name__ == '__main__':
53      # w1 adalah bobot untuk testing
54      w1 = np.array([[0, 1, 2, 3, 4],
55                     [5, 6, 7, 8, 9],
56                     [10, 11, 12, 13, 14]])
57      a = awal(w1)
58      x = jumlah_bobot(w1,a)   # x = perhitungan bobot berdasarkan h ( 10, 35, 60)
59      y = rank_hasil_jumlah(x) # (diberi index dan diranking)
60      z = set_top_n(y,1)
61      print y
62      # print x.shape
63      # print y  # matrix penjumlahan bobot diranking sebelum diambil top N
64      # print z  # matrix penjumlahan bobot setelah diranking dan diset 0 untuk yg bukan top N
65      # print extract_top_n(z)
```

Contoh implementasi multistep rank pada model yang disimpan pada file:

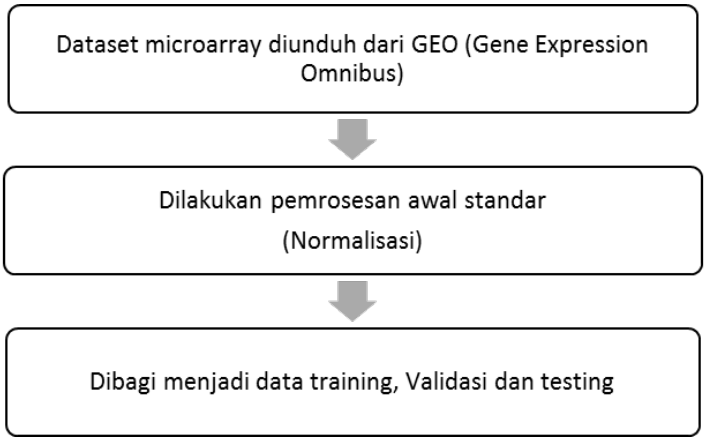
Listing 3.5: Implementasi Multistep rank Pada Model

```
1   import multistep_rank as mtr
2   import theano.tensor as T
3   import numpy as np
4   from ekstrak_csv import Ekstraktor
5
6
7   # buat function :
8   # hsl_ranking = multisteprank(model, [100,100,100]):
9
10  ekstraktor = Ekstraktor()
11
12  model =    ekstraktor.load_data("./dataset/model1000e_10k_5k_1k_500.pkl.gz")
13  print 'Jumlah layer : %i' % (model.n_layers)
14
15  Wlayer3 = model.rbm_layers[3].W
16  Wlayer2 = model.rbm_layers[2].W
17  Wlayer1 = model.rbm_layers[1].W
18  Wlayer0 = model.rbm_layers[0].W
19  # Wlayer1.shape.eval()
20
```

```
21 y3 = Wlayer3.get_value(True)
22 x3 = T.fmatrix()
23 x3 = y3.copy()
24
25 # ranking ujung
26 awal3 = mtr.awal(x3)
27 jml_bobot3 = mtr.jumlah_bobot(x3, awal3)
28 ranking_jml_bobot3 = mtr.rank_hasil_jumlah(jml_bobot3)
29 top_n3 = mtr.set_top_n(ranking_jml_bobot3,70)
30
31 # print "layer 3"
32 # print 'hasil perankingan top 50: '
33 # print ranking_jml_bobot3[:50]
34 # print 'set top n dengan 1 : '
35 # print top_n3.astype(int)
36
37 y2 = Wlayer2.get_value(True)
38 x2 = y2.copy()
39 awal2 = mtr.extract_top_n(top_n3)
40 jml_bobot2 = mtr.jumlah_bobot(x2, awal2)
41 ranking_jml_bobot2 = mtr.rank_hasil_jumlah(jml_bobot2)
42 top_n2 = mtr.set_top_n(ranking_jml_bobot2,700)
43
44 # print "layer 2"
45 # print 'hasil perankingan top 50: '
46 # print ranking_jml_bobot2[:50]
47 # print 'set top n dengan 1 : '
48 # print top_n2.astype(int)
49
50 y1 = Wlayer1.get_value(True)
51 x1 = y1.copy()
52 awal1 = mtr.extract_top_n(top_n2)
53 jml_bobot1 = mtr.jumlah_bobot(x1, awal1)
54 ranking_jml_bobot1 = mtr.rank_hasil_jumlah(jml_bobot1)
55 top_n1 = mtr.set_top_n(ranking_jml_bobot1,1500)
56
57 # print "layer 1"
58 # print 'hasil perankingan top 50: '
59 # print ranking_jml_bobot1[:50]
60 # print 'set top n dengan 1 : '
61 # print top_n1.astype(int)
62
63 y0 = Wlayer0.get_value(True)
64 x0 = y0.copy()
65 awal0 = mtr.extract_top_n(top_n1)
66 jml_bobot0 = mtr.jumlah_bobot(x0, awal0)
67 ranking_jml_bobot0 = mtr.rank_hasil_jumlah(jml_bobot0)
68 top_n0 = mtr.set_top_n(ranking_jml_bobot0,70)
69
70 print "layer visible"
71 print 'hasil perankingan top 250 layer visible 10k 5k 1k 500: '
72 print ranking_jml_bobot0[:250,0].astype(int)
```

3.4 Pengumpulan Data dan Pengolahan Awal

Data microarray tersedia secara bebas di *GEO (Gene Expression Omnibus)* [http://www.ncbi.nlm.nih.gov/geo/], dan dapat diunduh, untuk digunakan sebagai data penelitian. Kemudian dilakukan normalisasi standar yang sering di pakai pada data *microarray* dan yang sudah dibahas pada bab 2. Proses normalisasi ada banyak metode, dan akan digunakan satu metode standar untuk pengolahan awal microarray agar mendapatkan data konsisten dan dapat dibandingkan. Proses pengolahan awal dan normalisasi digunakan tools standar dan tersedia bebas yaitu R-Bioconductor.



Gambar 3.7: Proses Pengumpulan data dan Pengolahan Awal

3.5 Data Profil Gen Percobaan Microarray dan Biomarker

Definisi *Biomarker* adalah sesuatu penanda yang bisa digunakan sebagai indikator suatu penyakit dari pasien. [http://www.biomarker.co.uk/whatisabiomarkers.html] Sebagai contoh, untuk mendiagnosa kanker paru-paru, hanya dibutuhkan 26 ekspresi gen saja. Gen yang paling informatif ini disebut dengan Biomarker (Bing, 2006). Pada profil gen GSE10072 yang merupakan kanker paru-paru, menurut (Belinsky, 2004) ada 26 gen yang paling berpengaruh dari 22.283 gen yang diteliti secara bersamaan, seperti ditunjukkan pada Gambar 3.8 yang merupakan contoh dari *biomarker* kanker paru-paru.

Probe ID	Gene	Chromosomal	Current/Never† N = 30		Former/Never N = 23		Tumor/Non-Tumor N = 36	
	Symbol	Location	Fold-change	p-value	Fold-change	p-value	Fold-change	p-value
204641_at	NEK2*	1q32.2–q41	3.45	0.0001	2.84	0.0036	3.14	<0.0001
204822_at	TTK*	6q13–q21	3.27	<0.0001	2.08	0.0123	2.22	<0.0001
218009_s_at	PRC1*	15q26.1	2.99	0.0007	2.61	0.0109	2.60	<0.0001
207828_s_at	CENPF*	1q32–q41	2.88	<0.0001	2.28	0.0034	2.77	<0.0001
202095_s_at	BIRC5*	17q25	2.72	0.0002	2.10	0.0145	2.55	<0.0001
203362_s_at	MAD2L1	4q27	2.67	0.0003	1.93	0.0309	2.74	<0.0001
219918_s_at	ASPM	1q31	2.59	0.0008	2.12	0.0218	2.87	<0.0001
210559_s_at	CDC2	10q21.1	2.54	0.0009	2.02	0.0298	2.37	<0.0001
201897_s_at	CKS1B	1q21.2	2.36	0.0002	1.89	0.0152	2.47	<0.0001
204170_s_at	CKS2	9q22	2.36	0.0006	2.02	0.0148	1.69	0.0015
222077_s_at	RACGAP1*	12q13.12	2.35	0.0003	1.91	0.0178	2.13	<0.0001
203214_x_at	CDC2	10q21.1	2.29	0.0006	1.98	0.0150	2.12	<0.0001
219306_at	KIF15*	3p21.31	2.22	0.0002	2.00	0.0047	1.90	0·0001
209642_at	BUB1*	2q14	2.17	0.0009	1.68	0.0507	2.02	0.0001
210052_s_at	TPX2*	20q11.2	2.06	0.0006	1.87	0.0100	2.07	<0.0001
203418_at	CCNA2	4q25–q31	1.99	<0.0001	1.85	0.0012	1.82	<0.0001
212020_s_at	MKI67	10q25-qter	1.95	<0.0001	1.71	0.0016	1.41	0.0006
201088_at	KPNA2	17q23.1–q23.3	1.82	<0.0001	1.53	0.0079	2.34	<0.0001
211519_s_at	KIF2C*	1p34.1	1.78	0.0004	1.67	0.0062	1.51	0.0002
218252_at	CKAP2	13q14	1.75	0.0008	1.52	0.0292	1.47	0.0001
204887_s_at	PLK4	4q27–q28	1.74	0.0001	1.55	0.0066	1.48	<0.0001
211080_s_at	NEK2*	1q32.2–q41	1.57	0.0001	1.50	0.0019	1.36	0.0002
214894_x_at	MACF1	1p32–p31	0.65	0.0003	0.64	0.0016	0.52	<0.0001
208634_s_at	MACF1	1p32–p31	0.60	0.0001	0.58	0.0004	0.42	<.0.0001
202284_s_at	CDKN1A	6p21.2	0.54	0.0003	0.70	0.0668	0.65	0.0082
208893_s_at	DUSP6	12q22–q23	0.34	0.0003	0.32	0.0012	0.84	0.3102

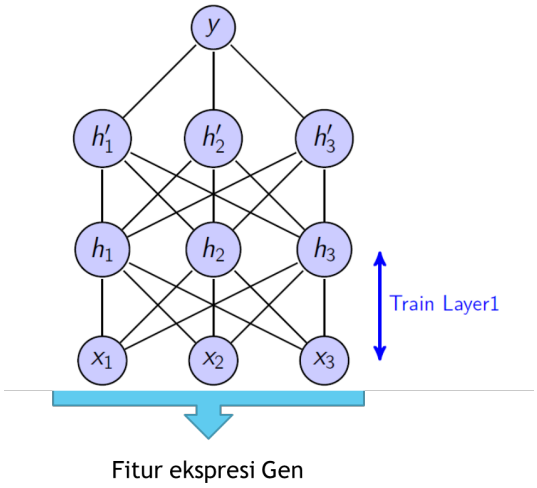
†Probe selection restricted to estimates with p<0.001 and fold-change >1.5 or <0.6667, and within the most inclusive category of genes with p≤0.001 in the GoMiner analysis (GO ID 7049, Appendix S2D).
*Genes involved in the mitotic spindle formation. The double line separates up-regulated and down-regulated probes.
doi:10.1371/journal.pone.0001651.t002

Gambar 3.8: Contoh 26 Gen Biomarker Kanker Paru-paru GSE10072 (Landi et al., 2008)

3.6 Perancangan Metodologi Penelitian

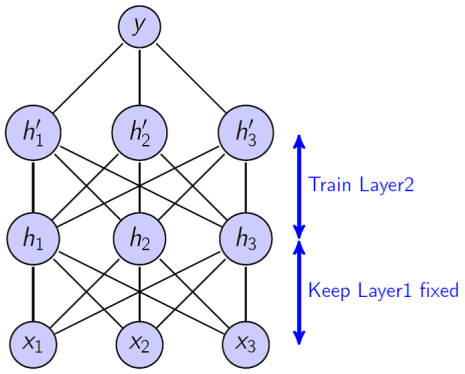
3.6.1 Tahapan *Unsupervised*

Tahap *unsupervised* adalah tahapan dimana model DBN ditraining secara *unsupervised* dengan data training pada tiap-tiap layernya secara *greedy*, artinya, proses pelatihan dilakukan secara berjenjang mulai dari layer visibel dengan hidden layer 0 dan kemudian layer ini bobotnya dibuat tetap dan digunakan sebagai input pada leyer berikutnya. Tiap layernya dihitung *cost*-nya, yang merupakan selisih dari error konstruksi dan error rekonstruksinya (Hinton et al., 2006) untuk kemudian diminimisasi errornya dengan menggunakan teknik *Contrastive Divergence (CD)*. Konsep ini disebut *greedy layer-wise training* yaitu setiap layer di traning secara independen dan satu-satu mulai dari layer input yang merupakan data ekspresi gen yang sudah disesuaikan dan dinormalisasi sampai layer output. Seperti pada Gambar 3.9



Gambar 3.9: Greedy layer-wise training pada layer visible dan hidden pertama(Duh, 2014)

Setelah layer pertama selesai di training, layer pertama dibuat *fixed* dan dipakai sebagai inputan visible dari layer selanjutnya. Demikian selanjutnya sampai layer terakhir yaitu layer output. Seperti pada Gambar 3.10



Gambar 3.10: Greedy layer-wise training pada selanjutnya, yaitu dengan membuat layer sebelumnya Fixed (Duh, 2014)

Pada tahapan training secara unsupervised ini dihitung cost function antara error konstruksi dibandingkan dengan error rekonstruksinya. Dalam RBM yaitu error konstruksi atau disebut error fase positif dibandingkan dengan error rekonstruksi atau error fase negatif.
Fungsi cost yang digunakan pada percobaan ini adalah NLL. Dimana log-likelihood $\mathcal{L}(\theta, \mathcal{D})$ dan fungsi loss-nya sebagai NLL $\ell(\theta, \mathcal{D})$ sebagai berikut:

$$\begin{aligned} \mathcal{L}(\theta, \mathcal{D}) &= \frac{1}{N} \sum_{x^{(i)} \in \mathcal{D}} \log\ p(x^{(i)}) \\ \ell(\theta, \mathcal{D}) &= -\mathcal{L}(\theta, \mathcal{D}) \end{aligned} \tag{3.1}$$

Menggunakan stochastic gradient $-\frac{\partial \log p(x^{(i)})}{\partial \theta}$, dimana θ adalah parameter dari modelnya.
Loss function yang merupakan Cost adalah negative log-likelihood dari log-likelihood model. Data dari gradien NLL kemudian memiliki bentuk yaitu:

$$-\frac{\partial \log p(x)}{\partial \theta} = \frac{\partial \mathcal{F}(x)}{\partial \theta} - \sum_{\tilde{x}} p(\tilde{x}) \frac{\partial \mathcal{F}(\tilde{x})}{\partial \theta}. \tag{3.2}$$

3.6.1.1 Cost

Cost merupakan variabel yang menggambarkan *Negative Log Likelihood*. Yang memiliki bentuk persamaan sebagai berikut:

$$\frac{1}{|\mathcal{D}|}\mathcal{L}(\theta = \{W, b\}, \mathcal{D}) = \frac{1}{|\mathcal{D}|}\sum_{i=0}^{|\mathcal{D}|} \log(P(Y = y^{(i)}|x^{(i)}, W, b))\ell(\theta = \{W, b\}, \mathcal{D}) \tag{3.3}$$

Dalam kode python dituliskan:

cost = classifier.negative_log_likelihood(y)

Semakin kecil cost, menunjukkan semakin kecil error rekonstruksinya. Hal ini menunjukkan bahwa, data rekonstruksi mendekati bentuk data konstruksinya (diambil dari data training).

3.6.2 Tahapan Supervised

Pada saat *training* secara *unsupervised* dilakukan, diukur *cost* yang menunjukkan perbedaan antara konstruksi dan rekonstruksi pada tiap layernyanya. Akan tetapi, hal ini hanya untuk mengetahui *cost* tiap-tiap layer RBM-nya, bukan seberapa baik model dalam melakukan klasifikasi. Oleh karena itu diperlukan satu layer output yang yang berupa *logistic regression* untuk mengetahui seberapa baik model dalam membedakan pasien kelas kanker dan normal.

3.6.2.1 Implementasi Logistic Regression pada Layer Output

Logistic regression adalah klasifier linear yang memiliki matriks bobot W dan vektor bias b. Klasifikasi merupakan proyeksi titik data pada sebuah himpunan *hyperplane* yang jaraknya digunakan sebagai penentu probabilitas keanggotaan kelasnya.

Secara matematis bisa dituliskan sebagai:

$$\begin{aligned} P(Y=i|x,W,b) &= softmax_i(Wx+b) \\ &= \frac{e^{W_i x+b_i}}{\sum_j e^{W_j x+b_j}} \end{aligned} \tag{3.4}$$

Output dari model akan memprediskikan dengan menghitung *argmax* dari vektor dimana elemen ke i adalah $P(Y=i|x)$.

$$y_{pred} = argmax_i P(Y=i|x,W,b) \tag{3.5}$$

Implementasinya menggunakan optimisasi stochastic gradient descent. Untuk implementasi lengkapnya ada di lampiran.

3.6.3 Tahapan Tuning Parameter

Parameter yang akan dilakukan *tuning* disini adalah: jumlah hidden units, jumlah banyaknya layer hidden dan banyaknya epoch. Tuning parameter dilakukan agar bisa didapatkan hasil yang optimum dari percobaan yang dilakukan. Tahap ini adalah tahap yang paling krusial untuk mendapatkan hasil yang diinginkan. Dikarenakan uniknya data microarray, maka dilakukan *trial and error* dari parameter-parameternya.
Proses tuning parameter ini memerlukan waktu yang lama karena setiap percobaan memiliki parameter yang diubah-ubah untuk menyesuaikan hasil yang diinginkan. Dikarenakan sifat dari microarray yang berbeda dengan citra yang sudah banyak dilakukan oleh peneliti, tuning parameter untuk data *microarray* pada arsitektur deep learning jarang dilakukan oleh peneliti, sehingga proses tuning dilakukan setiap selesai dilakukan percobaan yang memerlukan waktu antara 2 hari sampai 5 hari, tergantung dari epoch dan jumlah layer dan hidden unitnya.
Proses training pada arsitektur *deep learning* juga memerlukan kekuatan komputasi komputer yang kuat dan memory yang relatif lebih besar untuk mendapatkan model yang optimal.

3.7 Melakukan Testing Arsitektur DBN

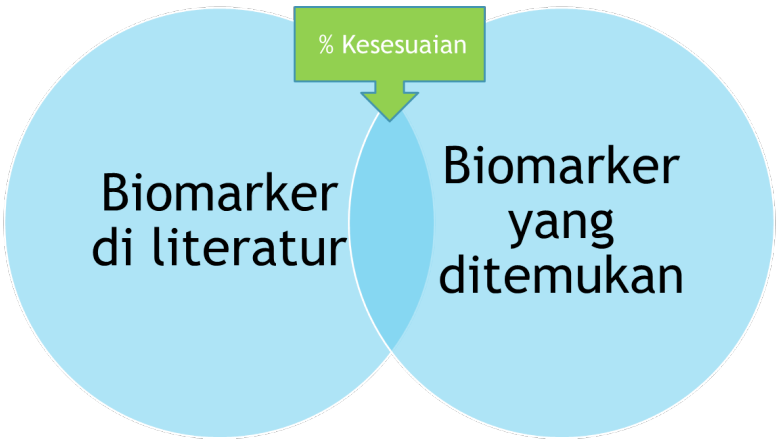
Hasil dari unsupervised learning yang dilakukan oleh DBN, akan diuji dahulu dengan dengan data validasi, apakah error rekonstruksinya lebih baik seperti pada gambar 3.1. Setelah dilakukan perankingan *biomarker*, diperlukan pengujian

apakah apakah seleksi fitur tersebut menggambarkan hasil yang diinginkan, dengan membandingkan biomarker yang dihasilkan dengan literature.

3.8 Evaluasi Hasil Perangkingan Dengan Klasifikasi Secara Supervised Menggunakan MLP

Proses evaluasi dilakukan dua kali, pertama, saat menggunakan data asli tanpa seleksi fitur, yang kedua setelah dilakukan seleksi fitur. Hal ini dilakukan untuk mengetahui apakah seleksi fitur tersebut bisa memperbaiki hasil klasifikasi secara signifikan dibandingkan tanpa dilakukan seleksi fitur.
Evaluasi hasil hasil perankingan secara *supervised* diperlukan untuk mengetahui apakah hasil perankingan tersebut memperbaiki hasil klasifikasi pasien kanker dan sehat hanya dengan menggunakan gen-gen yang dipilih berdasarkan ranking yang didapatkan.

3.9 Perbandingan Hasil Perangkingan Dengan Literatur



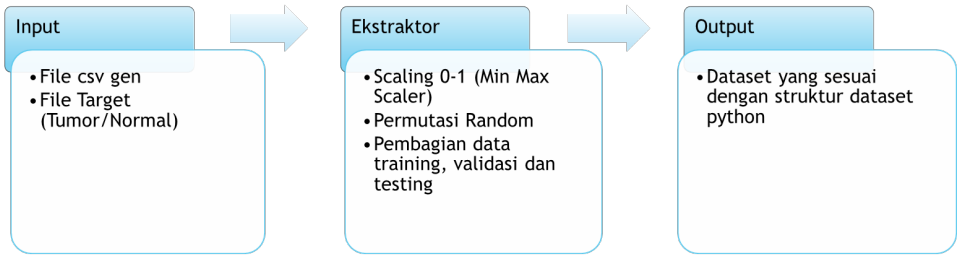
Gambar 3.11: Persen Kesesuaian Antara Biomarker yang Ditemukan dibandingkan dengan Biomarker di Literatur

Hasil perankingan pada percobaan tersebut selanjutnya diteliti apakah gen hasil perankingan tersebut adalah gen yang memiliki signifikansi terhadap penyakit yang diinginkan. Dalam kasus ini yaitu penyakit kanker paru-paru. Berikut adalah contoh 26 gen biomarker pada percobaan GSE10072 yang disitasi dari paper (Landi et al., 2008).

3.10 Modul-modul Pendukung

3.10.1 Kelas Ekstraktor

Untuk melakukan pengolahan pengolahan awal, didevelop sebuat kelas yang bernama kelas Ekstraktor. Kelas ini berfungsi untuk mengekstrak file csv dari data gen, menjadi file yang memiliki struktur data yang sesuai dengan library dbn.py di python. Hal ini dilakukan agar datanya memiliki struktur yang sesuai dengan dbn yaitu dilakukan normalisasi data profil gen yang berbentuk ekspresi gen menjadi rentang antara 0 sampai 1.



Gambar 3.12: Kelas Ekstraktor, Untuk melakukan Ekstraksi data Gen

3.10.2 Implementasi Kelas Ekstraktor di Python

Listing 4.1: Ekstraksi dataset untuk disesuaikan dengan struktur data modul dbn.py

```
1  from sklearn import preprocessing
2  from sklearn import utils
3  import numpy as np
4  import gzip, cPickle
5  from utilitas import top_n_dataset
6
7  class Salah(Exception):
8      pass
9
10 class Ekstraktor:
11     nama_file = str
12     data = np.empty
13     target_file = str
14     y = np.empty
15     jumlah_data = int
16     def norm_dataset(self,nama_file):
17         self.nama_file = nama_file + ".csv"
18         self.data = np.genfromtxt(self.nama_file, dtype=float, delimiter=",")
19         min_max_scaler = preprocessing.normalize(self.data)
20         #min_max_scaler = preprocessing.scale(self.data)
21         #min_max_scaler = preprocessing.minmax_scale(self.data)
22         np.savetxt(nama_file + "_norm.csv", min_max_scaler, delimiter=",")
```

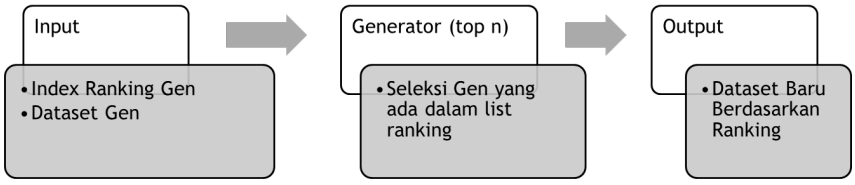
```
23
24      def generate_dataset(self, nama_file, target_file,train,valid,test, suffle = True):
25          self.nama_file = nama_file + ".csv"
26          self.target_file = target_file + ".csv"
27          self.data = np.genfromtxt(self.nama_file, dtype=float, delimiter=',')
28          self.y = np.genfromtxt(self.target_file, dtype=float, delimiter=',')
29          self.data = self.data.transpose()
30          self.jumlah_data = self.ambil_jumlah_dataset(self.data)
31          jml_train,jml_valid,jml_test =self.ambil_train_valid_test(self.jumlah_data,train,valid,test)
32          if suffle:
33              self.data, self.y = utils.shuffle(self.data,self.y,random_state = 5)
34          train_set_x = self.data[0:jml_train]
35          valid_set_x = self.data[jml_train+1:jml_train+1+jml_valid]
36          test_set_x = self.data[jml_train+1+jml_valid+1:jml_train+1+jml_valid+1+jml_test]
37          train_set_y = self.y.transpose()[2][0:jml_train]
38          valid_set_y = self.y.transpose()[2][jml_train+1:jml_train+1+jml_valid]
39          test_set_y = self.y.transpose()[2][jml_train+1+jml_valid+1:jml_train+1+jml_valid+1+jml_test]
40          train_set = train_set_x, train_set_y
41          valid_set = valid_set_x, valid_set_y
42          test_set = test_set_x, test_set_y
43          dataset = [train_set, valid_set, test_set]
44          self.simpan_data(self.nama_file + '_dataset.pkl.gz',dataset)
45          return dataset
46
47      def ambil_jumlah_dataset(self,data):
48          return data.shape[0]
49
50      def ambil_train_valid_test(self,jml_dataset,train,valid,test):
51          # ambil train valid test dalam %
52          if int(round(train+valid+test)) != 100 :
53              raise Salah("train+valid+test harus = 100%")
54          jml_train_set = int(round(float(jml_dataset)*(float(train)/100.)))
55          jml_valid_set = int(round(float(jml_dataset)*(float(valid)/100.)))
56          jml_test_set  = int(round(float(jml_dataset)*(float(test)/100.)))
57          return jml_train_set,jml_valid_set,jml_test_set
58
59      def simpan_data(self, n_file, data_simpan):
60          f = gzip.open(n_file, 'wb')
61          cPickle.dump(data_simpan, f, protocol=2)
62          f.close()
63          return data_simpan
64
65      def load_data(self, data):
66          # model_hasil = load cpickel
67          f = gzip.open(data, 'rb')
68          model_hasil = cPickle.load(f)
69          return model_hasil
70
71  class Generator:
72      ekstraktor = Ekstraktor()
73      # data_rank adalah array dari ranking data
74      def top_n_dataset(self, data_rank,dataset, namafile):
75          data_hasil = top_n_dataset(data_rank, dataset)
76          np.savetxt(namafile + ".csv", data_hasil, delimiter=",")
77          return data_hasil
78
79  if __name__ == '__main__':
80      ekstraktor = Ekstraktor()
81      generator = Generator()
82      array_rank= np.array([2, 3])
83      train = 80.5
84      valid = 14.5
85      test = 5
86      ekstraktor.norm_dataset("./dataset/iris_dataset")
87      dataset_iris = np.genfromtxt("./dataset/iris_dataset_norm.csv", dtype=float, delimiter=",")
88      generator.top_n_dataset(array_rank, dataset_iris ,"./dataset/iris_dataset_rank")
89      dataset_iris = ekstraktor.generate_dataset("./dataset/iris_dataset_rank",
90                                              "./dataset/iris_target", train, valid, test, True)
91
92      print dataset_iris
93      # ekstraktor.norm_dataset("./dataset/GSE10072_dataset")
```

Kelas ekstraktor ini melakukan adaptasi data yang tadinya memiliki struktur yang tidak kompatibel dengan library Theano yang di python, menjadi kompatibel

dan memiliki struktur data yang disesuaikan. Kemudian, dilakukan juga permutasi random agar datanya memiliki sebaran yang normal untuk kemudian dilakukan pembagian data yang terdiri dari sekian persen data training, validasi dan testing.

3.10.3 Kelas Generator

Kelas Generator ini adalah modul yang dibuat agar bisa secara otomatis memilih gen-gen yang dianggap penting pada sebuah array yang berisi index dari gen yang ada pada dataset.



Gambar 3.13: Diagram Kelas Generator yang digunakan untuk menggenerasi data gen berdasarkan rankingnya

3.10.4 Hasil Evaluasi Dengan Multi Layer Perceptron

Gambar 3.14: Diagram Proses Menggenerasi Data Untuk Dijadikan Dataset Training

Setelah didapatkan top-n gen, diperlukan proses untuk menggenerasi data ulang yang didapat dari data asli diambil gen top-n tersebut, pada penelitian ini akan diambil top 250 gen agar sesuai dengan gen yang didapat di literatur untuk kemudian dilakukan konfirmasi. Dan dievaluasi apakah terjadi peningkatan atau penurunan akurasi dibandingkan dengan tanpa adanya seleksi fitur.

BAB 4
PEMBAHASAN

Pada bab 4 ini akan dibahas tentang hasil penelitian dari metodologi yang ada pada bab tiga, dan masalah-masalah yang dihadapi pada saat implementasinya dan pembahasannya.

4.1 Overview Metodologi

Model yang dihasilkan dari *unsupervised learning* yang dilakukan oleh DBN menggunakan data training, harus diuji dahulu dengan dengan data validasi dan data testing, yaitu dengan cara memberikan satu layer output menggunakan *logistic regression* hal ini untuk mengetahui apakah klasifikasinya lebih baik atau sebaliknya. Hasil ini berpengaruh pada proses tuning parameter (jumlah layer dan jumlah hidden unitnya) untuk didapatkan *cost* yang paling optimal pada saat pre-training. Setelah dilakukan perankingan secara multi-step dari hasil percobaan yang terbaik, diperlukan pengujian apakah apakah seleksi fitur tersebut mendapatkan hasil klasifikasi yang lebih baik dengan menggunakan fitur yang telah diseleksi saja. Dengan cara membandingkan *biomarker* yang ditemukan oleh algoritma multi-step rangking dibandingkan dengan algoritma yang ada di literatur yaitu metode *bonferroni* untuk melakukan test statistik pada data gen tersebut (Hochberg, 1988).

4.2 Hasil Percobaan DBN Dengan Setting Hyperparameter yang Berbeda

Untuk mendapatkan hasil yang optimal dibutuhkan banyak percobaan dan setting parameter yang berbeda-beda, mulai dari jumlah layer, jumlah hidden unit tiap layernya, learning rate dan ukuran batch-nya. Oleh karena itu, dibawah adalah rekapitulasi percobaan dengan hasil terbaik dari sekian percobaan, dipilih lima percobaan yang paling baik hasilnya untuk kemudian dianalisa lebih jauh. Percobaan dibawah memiliki setting parameter seperti pada daftar berikut:

Tabel 4.1: Setting Parameter Awal

No.	Item	Keterangan
1	Dataset	Gene expression signature of cigarette smoking and its role in lung adenocarcinoma development and survival (Landi et al., 2008)
2	Total Data	107 Pasien
3	Kanker	58 Pasien
4	Normal	49 Pasien
5	Training	69 Pasien
6	Validasi	15 Pasien
7	Testing	23 Pasien
8	Epoch	1000 dan 2000
9	Learning Rate	0.01
10	Fitur Gen	22.283 Gen

Setelah dilakukan eksperimen secara *unsupervised* diperoleh *cost* terbaik pada Percobaan dan hasilnya ada di tabel 4.2 :

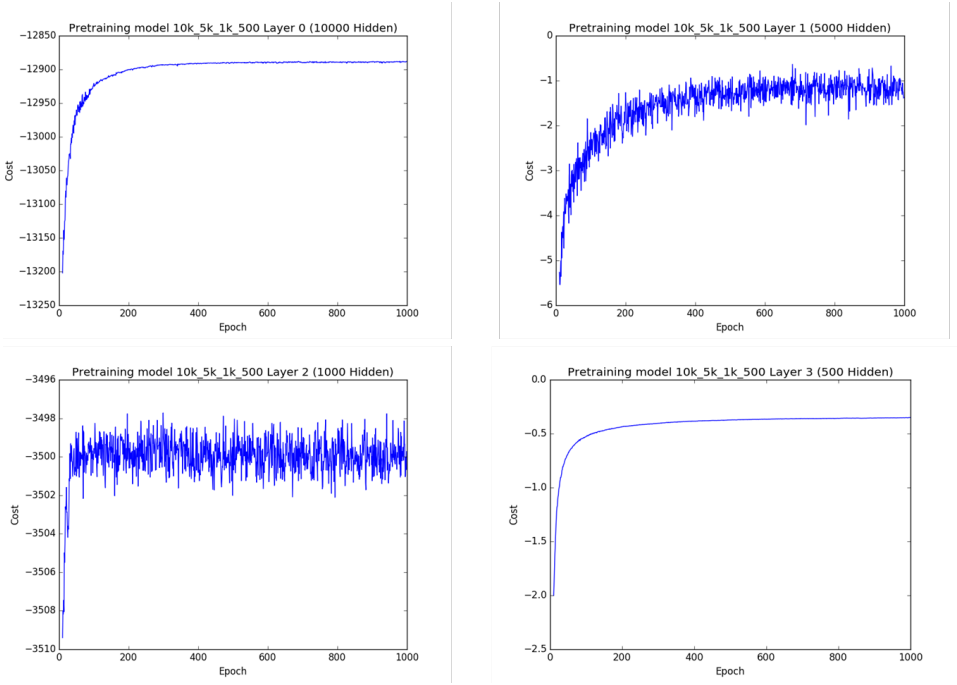
Tabel 4.2: Eksperimen DBN Unsupervised

Eks	Hidden	Epoch	Cost Lyr 0	Cost Lyr 1	Cost Lyr 2	Cost Lyr 3	Waktu (Jam)
1	[10000, 5000, 1000, 500]	1000	-12888.2	-1.37401	-3499.73	-0.351105	65
		2000	-12888.2	-0.828167	-3484.73	-0.150991	132
2	[7000, 10000, 5000, 1000]	1000	-12886.8	-1.36201	-6866.37	-0.163702	63
		2000	-12886.7	-1.57877	-6873.31	-0.0729352	138
3	[3000, 2000, 1000, 100]	1000	-12897.8	-0.862442	-1410.18	-3.244	58
		2000	-12897.0	-0.849616	-1397.09	-3.14657	123
4	[15000, 8000, 2000]	1000	-12934.5	-32.4227	-2756.41	(null)	68
5	[25000, 17000, 7000]	1000	-12888.1	-12.1715	-5446.34	(null)	72

Tabel diatas menunjukkan bahwa dengan epoch 1000 dan 2000 costnya tidak menunjukkan perbaikan secara signifikan. Bahkan untuk beberapa kasus, hasil-

nya lebih buruk. Dibawah adalah plot cost untuk percobaan yang dilakukan secara *greedy layer wise*, dari plot tersebut dapat dilihat bahwa cost pada epoch 700-an sudah tidak lagi membaik secara signifikan. Hal ini bisa dikarenakan oleh terbatasnya data training yang dipakai yaitu hanya 69 pasien dikarenakan oleh terbatasnya data yang didapatkan karena mahalnya percobaan microarray itu sendiri.

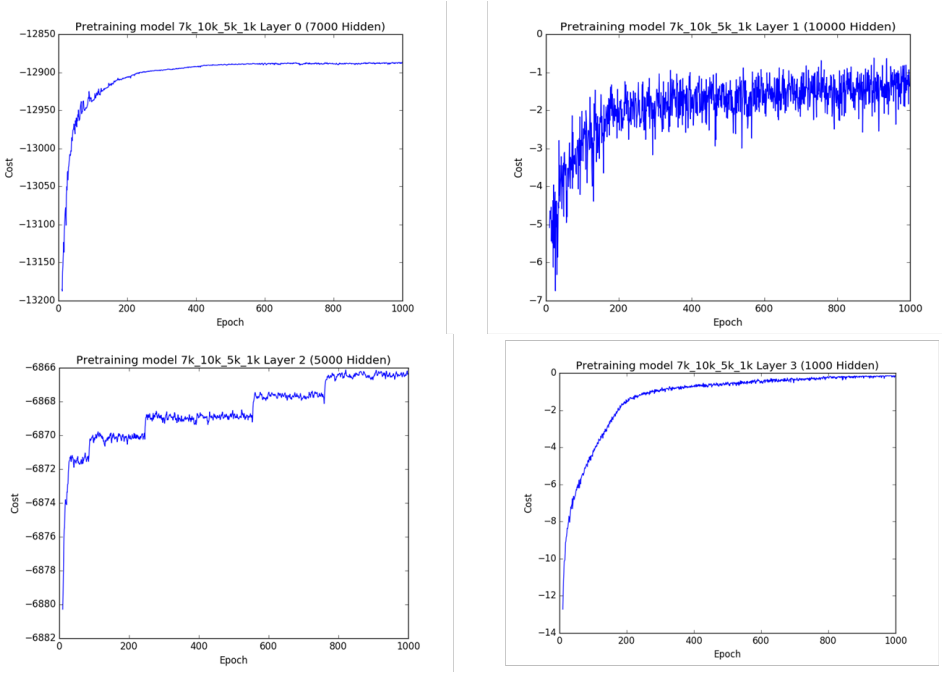
4.2.1 Plot Cost Percobaan 1 (Hidden = [10k, 5k, 1k, 500])



Gambar 4.1: Perbandingan Cost Pada Percobaan 1 Sampai 1000 Epoch Pada Tiap Layernya

Pada Gambar 4.1 merupakan perbandingan cost dari layer 0 sampai 3 (4 layer) dengan konfigurasi hidden [10000, 5000,1000,500] disitu bisa dilihat bahwa setelah epoch 500 tidak terjadi perbaikan cost yang signifikan. Juga cost pada layer 2 dan 3 memiliki ritme yang tidak stabil.

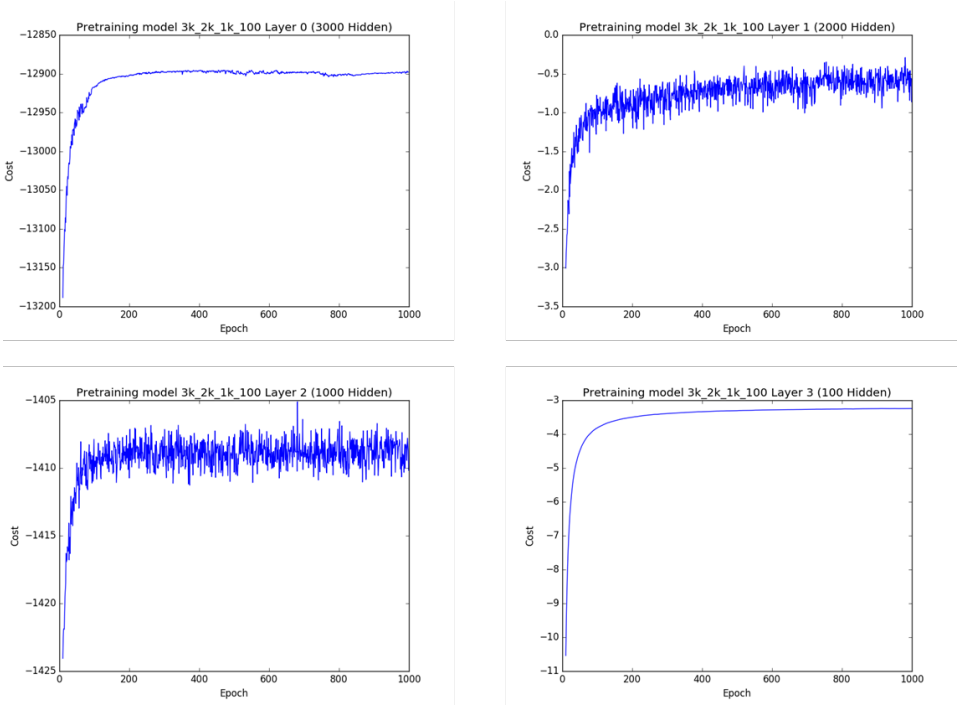
4.2.2 Plot Cost Percobaan 2 (Hidden = [7k, 10k, 5k, 1k])



Gambar 4.2: Perbandingan Cost Pada Percobaan 2 Sampai 1000 Epoch Pada Tiap Layernya

Pada Gambar 4.2 merupakan perbandingan cost dari layer 0 sampai 3 (4 layer) disitu bisa dilihat bahwa setelah epoch 500 tidak terjadi perbaikan cost yang signifikan. Juga cost pada layer 2 dan 3 memiliki ritme yang juga tidak stabil.

4.2.3 Plot Cost Percobaan 3 (Hidden = [3k, 2k, 1k, 100])



Gambar 4.3: Perbandingan Cost Pada Percobaan 3 Sampai 1000 Epoch Pada Tiap Layernya

Pada Gambar 4.3 merupakan perbandingan cost dari layer 0 sampai 3 (4 layer) disitu bisa dilihat bahwa setelah epoch 700-an tidak terjadi perbaikan cost yang signifikan. Juga *cost* pada layer 2 dan 3 memiliki ritme yang tidak stabil.

Berarti dari ketiga percobaan tersebut, secara garis besar, epoch lebih dari 700-an tidak mempengaruhi perbaikan error rekonstruksinya. Hal ini bisa disebabkan karena kurangnya data training.

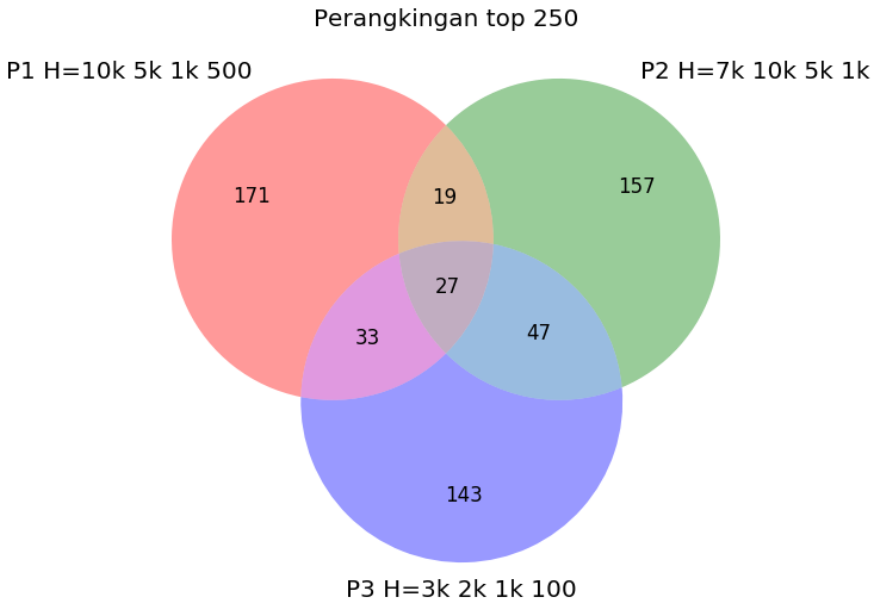
4.3 Hasil Penerapan Multi Step Ranking Bobot

Percobaan training DBN secara *unsupervised* yang dilakukan dengan setting pada tabel 4.2 diatas dipilih tiga percobaan terbaik untuk dilakukan algoritma multi-step ranking.

4.3.1 Diagram Venn Perpotongan Percobaan 1, 2 dan 3

Pada saat dilakukan multi-step ranking pada percobaan 1, 2 dan 3. Dibuat perankingan top 250 gen yang paling berpengaruh terhadap model-nya masing-masing. Kemudian, dibuat sebuah diagram untuk mendapatkan perpotongan 250 gen tersebut pada tiap-tiap percobaan. Hal ini digunakan untuk mengetahui gen-gen mana

yang selalu muncul di percobaan 1,2,3 atau muncul di dua percobaan dan hanya muncul di satu percobaan. Maka didapatkan diagram venn seperti pada Gambar 4.4



Gambar 4.4: Perbandingan Perankingan Top 250 pada tiga percobaan yang paling baik, ada 27 gen yang selalu muncul pada ketiga percobaan tersebut

Pada diagram venn diatas, ditunjukkan bahwa ada 27 gen yang selalu muncul pada percobaan 1, 2, 3. Hal ini menunjukkan bahwa gen tersebut adalah gen yang diindikasikan lebih informatif dibandingkan dengan gen yang lainnya. Ke 27 gen tersebut ada pada tabel 4.3 penemuan 27 gen yang selalu muncul pada tiga percobaan terbaik tersebut bisa diindikasikan sebagai *biomarker*. Yaitu gen yang bisa mencirikan seseorang terkena kanker paru-paru atau tidak.

Tabel 4.3: Index dan Kode Gen yang Diindikasikan sebagai *Biomarker*

Index	Kode Gen
7303	207783_x_at
1418	201891_s_at
9666	210183_x_at
15890	216520_s_at
24	200004_at
21919	38691_s_at
11298	211911_x_at
13741	214363_s_at
46	200026_at
307	200780_x_at
12727	213347_x_at
13246	213867_x_at
4418	204892_x_at
6084	206559_x_at
13765	214387_x_at
328	200801_x_at
201	200674_s_at
21860	37004_at
101	200081_s_at
232	200705_s_at
11370	211984_at
879	201352_at
11120	211720_x_at
20968	221607_x_at
115	200095_x_at
1019	201492_s_at
511	200984_s_at

Ke-27 gen pada tabel tersebut merupakan gen yang diindikasikan memiliki pengaruh yang signifikan pada percobaan. Akan tetapi hal ini perlu dilakukan konfirmasi lebih lanjut untuk memastikan bahwa gen tersebut memang berpengaruh secara signifikan terhadap penyakit kanker paru-paru. Ada dua tahapan konfirmasi yang pertama tahap konfirmasi dengan memastikan bahwa hasil klasifikasi dengan hanya menggunakan top 250 gen tersebut bisa mengklasifikasikan pasien

sehat dan pasien kanker. Tahap yang kedua adalah dengan cara konfirmasi melalui literatur tentang biomarker kanker paru-paru yang sudah ditemukan pada penelitian sebelumnya.

4.4 Bagian Supervised Learning Dengan Multi Layers Perceptron (MLP)

Pada saat dilakukan klasifikasi pasien kanker dan normal tanpa dilakukan seleksi fitur, dikarenakan banyaknya fitur gen yang merupakan noise, maka perbandingan fitur gen dan pasien menjadi sangat lebar, oleh karena itu sangat rentan dengan masalah yang sering timbul dari teknik pembelajaran mesin yaitu *overfitting*. Oleh karena itu, salah satu cara untuk menghindari overfitting adalah dengan metode seleksi fitur.
Setelah dilakukan perbandingan gen biomarker yang ditemukan pada proses perankingan diatas, top 250 gen tersebut dibuat menjadi data input untuk kasus klasifikasi. Untuk di evaluasi apakah hasil klasifikasinya lebih baik dibandingkan dengan tanpa seleksi fitur.
Tabel 4.4 merupakan perbandingan error antara logistic regression yang ditempatkan pada layer akhir DBN, tanpa dilakukan seleksi fitur. Dibandingkan dengan MLP yang memiliki 1 layer hidden dan 250 hidden unit. Untuk dilakukan training ulang dan dibandingkan dengan hasil yang diperoleh dari logistic regression.

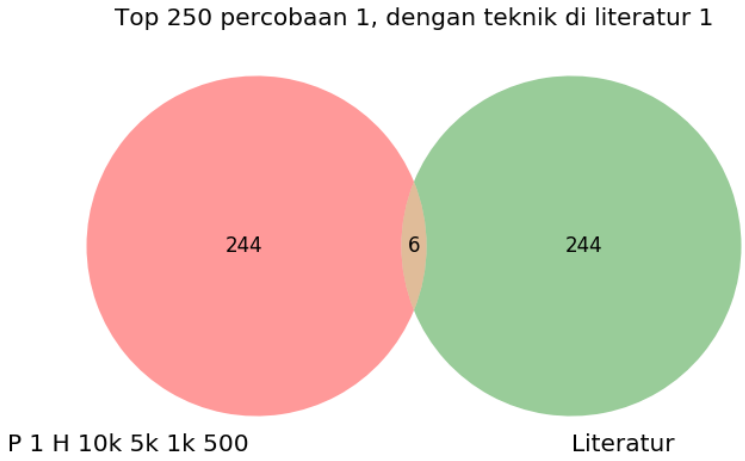
Tabel 4.4: Perbandingan Error Antara Dengan dan Tanpa Seleksi Fitur

	Tanpa Seleksi Fitur(LogReg)		Dengan Seleksi Fitur(MLP)	
Percobaan	Validation Error	Test Error	Validation Error	Test Error
1	50%	66%	5.55%	0%
2	50%	30%	0%	8.33%
3	50%	30%	0%	16%

Dari tabel 4.4 dapat disimpulkan bahwa terjadi perbaikan signifikan antara validation dan test error dibandingkan tanpa dilakukan seleksi fitur. Akan tetapi hal ini masih belum menunjukkan apakah seleksi fitur gen tersebut merupakan *biomarker*. Oleh karena itu diperlukan evaluasi lebih lanjut yaitu dengan evaluasi literatur untuk memastikan bahwa gen yang ditemukan memang informatif untuk kasus kanker paru-paru.

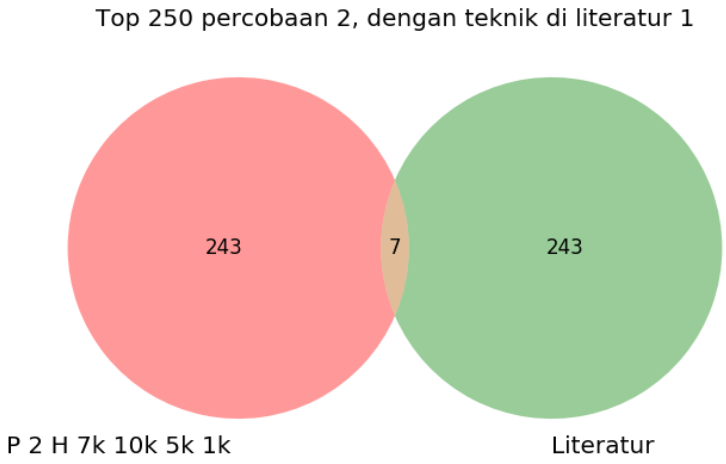
4.5 Hasil Evaluasi Dengan Literatur Pertama Bonferroni Method(Hochberg, 1988)

Metode Bonferroni adalah metode multipel testing di statistik yang paling umum digunakan untuk dataset dari percobaan *microarray*. Metode ini adalah metode yang dipakai oleh Landi et al. (2008) dalam menganalisa dataset GSE10072 yang merupakan hasil eksperimen kanker paru-paru (Landi et al., 2008) Dengan melakukan test statistik menggunakan metode bonferroni dipilih 250 gen yang paling signifikan dari hasil test statistik tersebut dibandingkan dengan gen yang dipilih dari metode multi-step ranking, didapatkan hasil sebagai berikut.



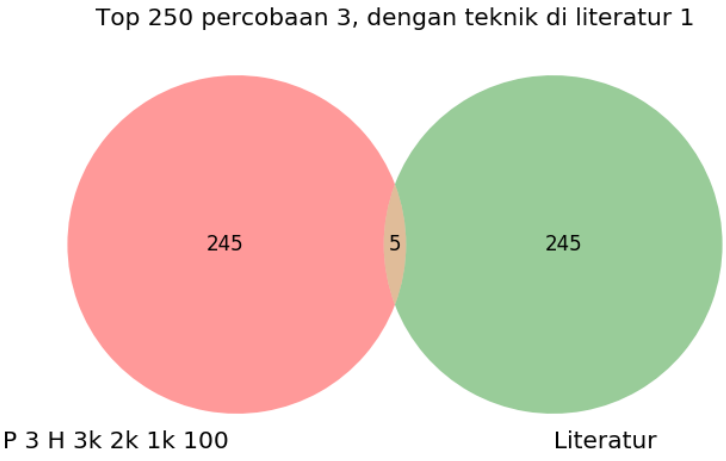
Gambar 4.5: Hasil top 250 Gen dibandingkan dengan Metode bonferroni

Pada percobaan 1, dihasilkan perpotongan 6 gen. Walaupun kelihatan kecil tetapi perpotongan 6 gen dari 22 ribu-an gen menjadi sangat signifikan untuk diteliti lebih lanjut gen-gen tersebut sebagai kandidat *Biomarker*



Gambar 4.6: Hasil top 250 Gen dibandingkan dengan Metode bonferroni

Percobaan 2 dibandingkan dengan metode bonferroni juga memiliki perpotongan yang tidak besar yaitu 7 gen saja.



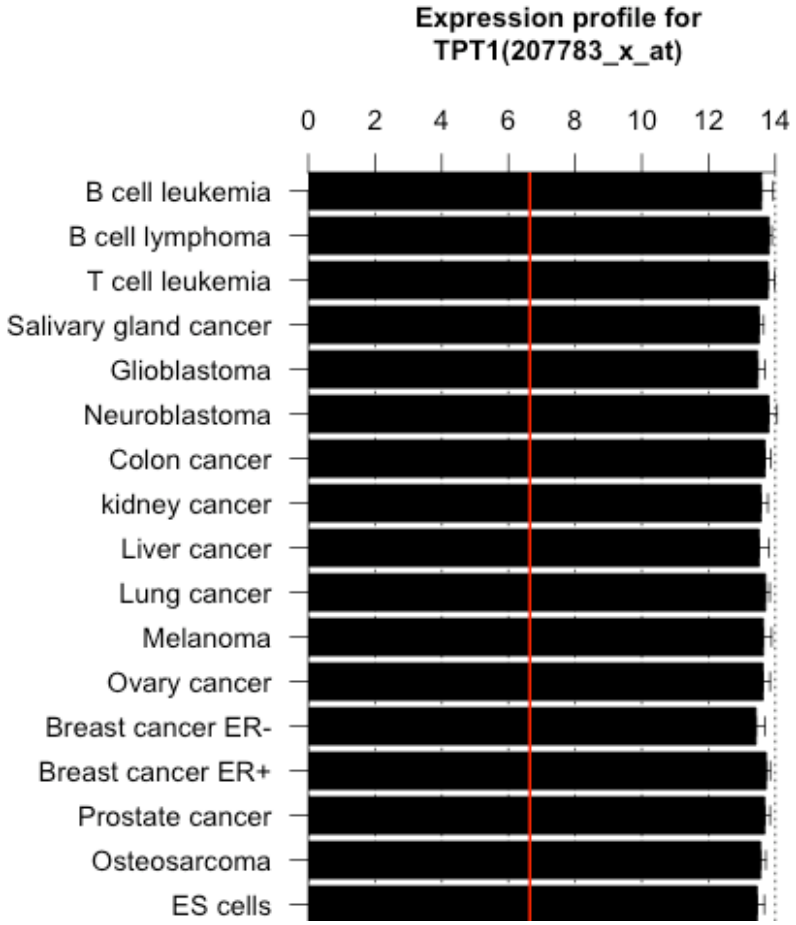
Gambar 4.7: Hasil top 250 Gen dibandingkan dengan Metode bonferroni

Percobaan 3 dibandingkan dengan metode bonferroni memiliki perpotongan ke-

sesuaaina 5 gen.

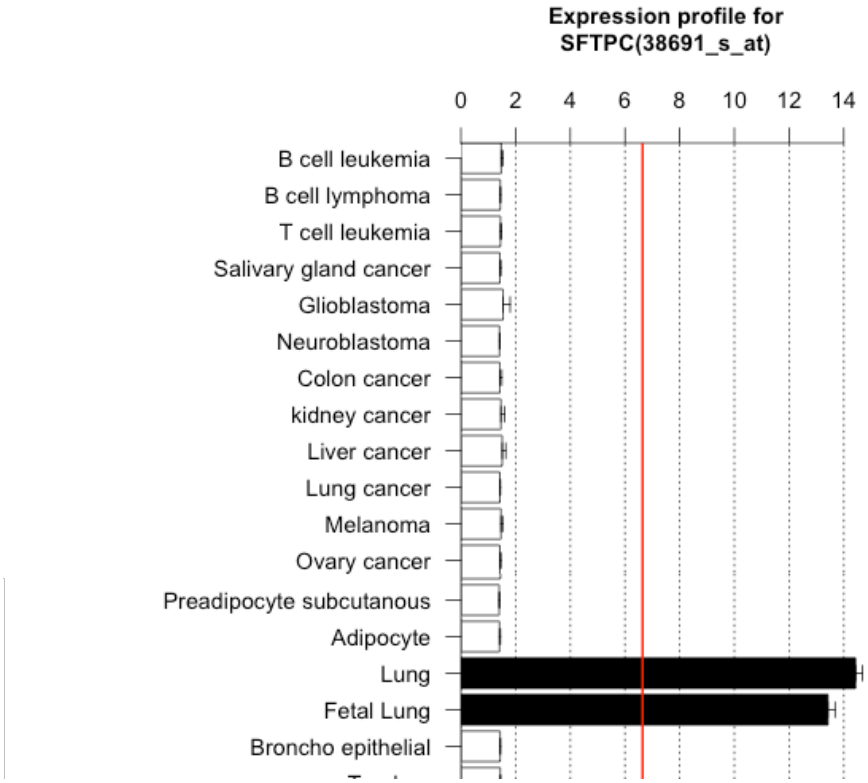
4.6 Hasil Konfirmasi Dengan Literatur Kedua Harvard Cancer Center (https://ccib.mgh.harvard.edu/xavier)

Sebanyak 27 gen yang ditemukan untuk irisan tiga percobaan terbaik, akan dilakuan review literatur lebih jauh. Menurut situs harvard cancer center, gen-gen tertentu bisa menununjukkan tingkat signifikansi gen tersebut terhadap sebuah penyakit kanker. Gen yang berada pada ranking 1 sampai 27 tersebut memiliki signifikasni yang tinggi terhadap kanker paru-paru dibandingkan dengan gen yang dipilih secara acak.



Gambar 4.8: Profil Ekspresi Gen TPT1 yang merupakan ranking pertama

Dari gambar bisa dilihat bahwa signifikansi gen TPT1 yang merupakan gen dengan ranking pertama memiliki signifikansi terhadap penyakit kanker paru-paru (lung cancer). Sumber profil gen didapat dari https://ccib.mgh.harvard.edu/xavier



Gambar 4.9: Profil Ekspresi Gen TPT1 yang merupakan ranking pertama

Pada dua contoh profil yang ditemukan yaitu gen TPT1 dan gen SFTPC bisa disimpulkan bahwa walaupun ekspresi gen tersebut ditemukan pada kanker paru-paru, tetapi tidak unik dan juga ditemukan di kanker-kanker yang lain misalnya leukemia, lymphoma dan sebagainya. Hal ini terjadi karena data yang dipakai adalah data kanker paru-paru saja. Sehingga gen yang sama bisa signifikan pada kanker-kanker lainnya dikarenakan tidak adanya data selain kanker paru-paru untuk dijadikan data trainingnya.

4.7 Kendala-Kendala yang Dialami Selama Melakukan Percobaan

Pada saat melakukan percobaan dengan menggunakan arsitektur *deep learning* kendala yang paling utama adalah lamanya waktu training dan penggunaan resource memory yang sangat besar. Dengan menggunakan komputer core i5 dengan memory vga 2 GB, dan RAM 4 GB diperlukan waktu rata-rata 3-5 hari. Seperti pada tabel 4.5. Dikarenakan oleh kendala ini maka untuk melakukan percobaan dengan arsitektur yang lebih besar, misalnya dilakukan penambahan layer (lebih dari 4 layer) dan penambahan hidden unit, menjadi terbatas. Juga masalah pada terbatas-

nya dataset untuk training yang hanya 107 sampel pasien, hal ini disebabkan oleh mahalnya percobaan *microarray* yang dilakukan sehingga sulit untuk mendapatkan data yang lebih besar lagi.

Tabel 4.5: tabel ukuran model dan waktu running

Percobaan	Konfigurasi Hidden (h0, h1, h2, h3)	Ukuran Model	Running (Jam) (1000e, 2000e)
1	10000, 5000,1000, 500	1 GB	65, 132
2	7000,10000,5000,1000	1 GB	63, 138
3	3000,2000,1000,100	275 MB	58, 123
4	15000,8000,2000	Out of Memory	-
5	25000, 17000, 7000	Out of Memory	-

Pada tabel diatas, bisa dihilhat bahwa hidden yang melebihi 15000 sudah menghabiskan RAM komputer yang hanya berukuran 4 GB. Oleh karena itu, percobaan yang seharusnya bisa memperdalam layer dan memperbesar hidden unit tidak memungkinkan untuk dilakukan.

BAB 5
KESIMPULAN DAN SARAN

5.1 Kesimpulan

Penelitian ini menerapkan seleksi fitur perankingan multi-step pada *arsitektur deep belief network (DBN)* untuk mencari *biomarker* pada data microarray penyakit kanker paru-paru. Penerapannya menggunakan library Theano pada bahasa pemrograman Python. Kesimpulan yang dapat diambil dari penelitian ini adalah sebagai berikut:

1. Metodologi pencarian *biomarker* secara *unsupervised* dengan menggunakan teknik *Deep Belief Network (DBN)* didapatkan model terbaik dengan konfigurasi hidden unit 4 layer [7000, 10000, 5000, 1000] dengan epoch 1000 dan learning rate 0.01.

2. Algoritma perankingan gen secara multi-step yang diajukan pada thesis ini, bisa dilakukan untuk network DBN yang di training secara *unsupervised* murni, dan menghasilkan hasil biomarker yang memiliki signifikansi yang tinggi.

3. Evaluasi yang dilakukan secara bertahap yaitu mulai dari dibandingkannya metode unsupervised dengan masalah klasifikasi *supervised* dengan MLP menunjukkan peningkatan hasil klasifikasi yang signifikan. Dan *biomarker* yang ditemukan, dibandingkan dengan literatur yaitu metode bonferroni menunjukkan bahwa gen yang ditemukan memiliki signifikansi yang tinggi.

5.2 Saran

Karena keterbatasan waktu penelitian dan mesin yang digunakan, maka ada banyak hal yang bisa dilakukan untuk penelitian selanjutnya yaitu:

1. Melakukan generalisasi, apakah metode ini cocok juga dilakukan untuk data *microarray* pada penyakit-penyakit lainnya selain kanker paru-paru.

2. Karena metode ini menggunakan arsitektur *deep learning* dengan jaringan DBN, apakah dengan melakukan pada network DBN yang lebih dalam (layer hidden dengan kedalaman lebih dari 4 layer) bisa meningkatkan keakuratan

pendeteksian *biomarker*. Dikarenakan terbatasnya memory komputer, maka hal ini belum memungkinkan untuk dilakukan.

3. Diterapkan arsitektur deep learning yang lainnya misalnya *stacked autoencoder, denoising autoencoder, convolutional neural-network*, dan atau arsitektur-arsitektur deep learning yang baru.

DAFTAR REFERENSI

Constantin F Aliferis, Ioannis Tsamardinos, Pierre P Massion, Alexander R Statnikov, Nafeh Fananapazir, and Douglas P Hardin. Machine learning models for classification of lung cancer and selection of genomic markers using array gene expression data. In *FLAIRS Conference*, pages 67–71, 2003.

M Mwanadan Babu. Introduction to microarray data analysis. *Computational genomics: Theory and application*, pages 225–249, 2004.

Supriyo Bandyopadhyay, Saurav Mallik, and Amit Mukhopadhyay. A survey and comparative study of statistical tests for identifying differential expression from microarray data. *Computational Biology and Bioinformatics, IEEE/ACM Transactions on*, 11(1):95–115, 2014.

Steven A Belinsky. Gene-promoter hypermethylation as a biomarker in lung cancer. *Nature Reviews Cancer*, 4(9):707–717, 2004.

Yoshua Bengio, Pascal Lamblin, Dan Popovici, Hugo Larochelle, et al. Greedy layer-wise training of deep networks. *Advances in neural information processing systems*, 19:153, 2007.

Kevin Duh. Deep learning & neural networks lecture. 2014.

Mourad Elloumi and Albert Y Zomaya. *Algorithms in computational molecular biology: techniques, approaches and applications*, volume 21. John Wiley & Sons, 2011.

Rasool Fakoor, Faisal Ladhak, Azade Nazi, and Manfred Huber. Using deep learning to enhance cancer diagnosis and classification. *roceedings of the International Conference on Machine Learning.*, 2013.

Mikael Häggström. Diagram of the pathways of human steroidogenesis. *Medicine*, 1:1, 2014.

Geoffrey E Hinton and Ruslan R Salakhutdinov. Reducing the dimensionality of data with neural networks. *Science*, 313(5786):504–507, 2006.

Geoffrey E Hinton, Simon Osindero, and Yee-Whye Teh. A fast learning algorithm for deep belief nets. *Neural computation*, 18(7):1527–1554, 2006.

Yosef Hochberg. A sharper bonferroni procedure for multiple tests of significance. *Biometrika*, 75(4):800–802, 1988.

Maria Teresa Landi, Tatiana Dracheva, Melissa Rotunno, Jonine D Figueroa, Huaitian Liu, Abhijit Dasgupta, Felecia E Mann, Junya Fukuoka, Megan Hames, Andrew W Bergen, et al. Gene expression signature of cigarette smoking and its role in lung adenocarcinoma development and survival. *PloS one*, 3(2):e1651, 2008.

Christopher Poultney, Sumit Chopra, Yann L Cun, et al. Efficient learning of sparse representations with an energy-based model. In *Advances in neural information processing systems*, pages 1137–1144, 2006.

Sridhar Ramaswamy, Pablo Tamayo, Ryan Rifkin, Sayan Mukherjee, Chen-Hsiang Yeang, Michael Angelo, Christine Ladd, Michael Reich, Eva Latulippe, Jill P Mesirov, et al. Multiclass cancer diagnosis using tumor gene expression signatures. *Proceedings of the National Academy of Sciences*, 98(26):15149–15154, 2001.

Shirish Krishnaj Shevade and S Sathiya Keerthi. A simple and efficient algorithm for gene selection using sparse logistic regression. *Bioinformatics*, 19(17):2246–2253, 2003.

Deep Learning Tutorial. Lisa lab. *University of Montreal*, 2014.

Youngmi Yoon, Jongchan Lee, and Sanghyun Park. Building a classifier for integrated microarray datasets through two-stage approach. In *BioInformatics and BioEngineering, 2006. BIBE 2006. Sixth IEEE Symposium on*, pages 94–102. IEEE, 2006.

LAMPIRAN

LAMPIRAN 1

Implementasi multi-step ranking dengan menggunakan python: Listing 1 : Implementasi Multi-Step Ranking di python

```
1
2  # perkalian matrix rank weight
3  import numpy as np
4
5  def awal(w):
6      return np.ones((w.shape[1],), dtype=np.float)
7
8  def jumlah_bobot(w, top_ke_n):
9      # kalikan w dengan matrix 1
10     return w.dot(top_ke_n)
11
12 def rank_hasil_jumlah(sum_w):
13     # urutkan sum_w dan beri index
14     """
15
16     :rtype sum_w : numpy.array
17     """
18     swi = sum_w.shape[0]
19     hsl = np.arange(swi)
20     c = np.concatenate((hsl,sum_w))
21     c = c.reshape(2,swi)
22     c = c.T
23     z = c[c[:,1].argsort()[::-1]] # urutkan descending berdasarkan bobot (indeks mengikuti)
24     return z
25
26 def set_top_n(idx_sum_w, top_n = 2):
27     # set = 0 semua yang bukan top n
28     # kembalikan ke urutan semula
29     z = idx_sum_w.copy()
30     z[top_n:,1] = 0.
31     z[0:top_n,1]= 1.
32     # print 'z adalah'
33     # print z
34     d = z[z[:,0].argsort()[::]]
35     # print 'd adalah'
36     # print d
37     return d
38
39 # set_rank : melakukan setting 1 untuk top n dan
40 def extract_top_n(n):
41     return n[:,1]
42
43 def set_index_dengan_gen(bobot_akhir):
44     # index gen dengan urutan perankingannya
45     pass
46
47 def plot_diagram(a, b):
48     # plot himpunan a dan b dan anggota keduanya
49     pass
50
51
52 if __name__ == '__main__':
53     # w1 adalah bobot untuk testing
54     w1 = np.array([[0, 1, 2, 3, 4],
55                    [5, 6, 7, 8, 9],
56                    [10, 11, 12, 13, 14]])
57     a = awal(w1)
58     x = jumlah_bobot(w1,a)   # x = perhitungan bobot berdasarkan h ( 10, 35, 60)
59     y = rank_hasil_jumlah(x) # (diberi index dan diranking)
60     z = set_top_n(y,1)
61     print y
62     # print x.shape
63     # print y  # matrix penjumlahan bobot diranking sebelum diambil top N
64     # print z  # matrix penjumlahan bobot setelah diranking dan diset 0 untuk yg bukan top N
```

```
65      # print extract_top_n(z)
```

Contoh implementasi multistep rank pada model yang disimpan pada file: Listing 2 : Implementasi Multistep rank Pada Model

```
1   import multistep_rank as mtr
2   import theano.tensor as T
3   import numpy as np
4   from ekstrak_csv import Ekstraktor
5
6
7   # buat function :
8   # hsl_ranking = multisteprank(model, [100,100,100]):
9
10  ekstraktor = Ekstraktor()
11
12  model =    ekstraktor.load_data("./dataset/model1000e_10k_5k_1k_500.pkl.gz")
13  print 'Jumlah layer : %i' % (model.n_layers)
14
15  Wlayer3 = model.rbm_layers[3].W
16  Wlayer2 = model.rbm_layers[2].W
17  Wlayer1 = model.rbm_layers[1].W
18  Wlayer0 = model.rbm_layers[0].W
19  # Wlayer1.shape.eval()
20
21  y3 = Wlayer3.get_value(True)
22  x3 = T.fmatrix()
23  x3 = y3.copy()
24
25  # ranking ujung
26  awal3 = mtr.awal(x3)
27  jml_bobot3 = mtr.jumlah_bobot(x3, awal3)
28  ranking_jml_bobot3 = mtr.rank_hasil_jumlah(jml_bobot3)
29  top_n3 = mtr.set_top_n(ranking_jml_bobot3,70)
30
31  # print "layer 3"
32  # print 'hasil perankingan top 50: '
33  # print ranking_jml_bobot3[:50]
34  # print 'set top n dengan 1 : '
35  # print top_n3.astype(int)
36
37  y2 = Wlayer2.get_value(True)
38  x2 = y2.copy()
39  awal2 = mtr.extract_top_n(top_n3)
40  jml_bobot2 = mtr.jumlah_bobot(x2, awal2)
41  ranking_jml_bobot2 = mtr.rank_hasil_jumlah(jml_bobot2)
42  top_n2 = mtr.set_top_n(ranking_jml_bobot2,700)
43
44  # print "layer 2"
45  # print 'hasil perankingan top 50: '
46  # print ranking_jml_bobot2[:50]
47  # print 'set top n dengan 1 : '
48  # print top_n2.astype(int)
49
50  y1 = Wlayer1.get_value(True)
51  x1 = y1.copy()
52  awal1 = mtr.extract_top_n(top_n2)
53  jml_bobot1 = mtr.jumlah_bobot(x1, awal1)
54  ranking_jml_bobot1 = mtr.rank_hasil_jumlah(jml_bobot1)
55  top_n1 = mtr.set_top_n(ranking_jml_bobot1,1500)
56
57  # print "layer 1"
58  # print 'hasil perankingan top 50: '
59  # print ranking_jml_bobot1[:50]
60  # print 'set top n dengan 1 : '
61  # print top_n1.astype(int)
62
63  y0 = Wlayer0.get_value(True)
64  x0 = y0.copy()
65  awal0 = mtr.extract_top_n(top_n1)
66  jml_bobot0 = mtr.jumlah_bobot(x0, awal0)
67  ranking_jml_bobot0 = mtr.rank_hasil_jumlah(jml_bobot0)
68  top_n0 = mtr.set_top_n(ranking_jml_bobot0,70)
69
70  print "layer visible"
```

```
print 'hasil perankingan top 250 layer visible 10k 5k 1k 500: '
print ranking_jml_bobot0[:250,0].astype(int)
```

Listing 3 : Implementasi melakukan plotting diambil dari log

```
import numpy as np
import matplotlib.pyplot as plt
import thesis.ekstrak_csv as eks

kamus = {"Pre-training layer": "", "epoch": "", "cost": "", "\n": ""}


def replace_all(text, dic):
    for i, j in dic.iteritems():
        text = text.replace(i, j)
    return text


def load_file_text(nama_file):
    text_file = open(nama_file, "r")
    lst = text_file.readlines()
    a = np.array([replace_all(lst[0], kamus).split(",")], float)
    for i in range(1,len(lst)):
        b = np.array([replace_all(lst[i], kamus).split(",")], float)
        a = np.r_[a,b]
    return a

def load_epch_layer(mat, jml_epoch, layer):
    return mat[(layer*jml_epoch):((layer+1)*jml_epoch),1:3]

def load_file_ekstrak_layer_epoch_cost(nama_file_log_test):
    c = load_file_text(nama_file_log_test)
    return c

if __name__ == '__main__':
    # contoh pemakaian load matrix
    # edit file log sampai hanya ada layer epoch dan cost saja
    # simpan dengan kode jml epoch layer
    # load file log dengan :

    # f = load_file_ekstrak_layer_epoch_cost("../thesis_test/dataset_test/log_test.log")
    # g = load_epch_layer(f, 2, 1)  # g = matrix dengan isi epoch dan cost pada layer 2


    f = load_file_ekstrak_layer_epoch_cost("../thesis/dataset/log1000e_3k_2k_1k_100.txt")
    ekstraktor = eks.Ekstraktor()

    # ekstraktor.simpan_data("../thesis/dataset/1000e_3k_2k_1k_100_lyr1",g)
    # plot layer 1
    plt.ylabel("Cost")
    plt.xlabel("Epoch")
    plt.title("Pretraining model 3k_2k_1k_100 Layer 0 (3000 Hidden)")
    g_0 = load_epch_layer(f,1000,0)
    plt.plot(g_0[10:,0],g_0[10:,1])
    plt.show()

    plt.ylabel("Cost")
    plt.xlabel("Epoch")
    plt.title("Pretraining model 3k_2k_1k_100 Layer 1 (2000 Hidden)")
    g_1 = load_epch_layer(f, 1000, 1)
    plt.plot(g_1[10:, 0], g_1[10:, 1])
    plt.show()

    plt.ylabel("Cost")
    plt.xlabel("Epoch")
    plt.title("Pretraining model 3k_2k_1k_100 Layer 2 (1000 Hidden)")
    g_2 = load_epch_layer(f, 1000, 2)
    plt.plot(g_2[10:, 0], g_2[10:, 1])
    plt.show()

    plt.ylabel("Cost")
    plt.xlabel("Epoch")
    plt.title("Pretraining model 3k_2k_1k_100 Layer 3 (100 Hidden)")
    g_3 = load_epch_layer(f, 1000, 3)
    plt.plot(g_3[10:, 0], g_3[10:, 1])
    plt.show()
```

```
72
73      f = load_file_ekstrak_layer_epoch_cost("../thesis/dataset/log1000e_7k_10k_5k_1k.txt")
74      ekstraktor = eks.Ekstraktor()
75      #
76      # # ekstraktor.simpan_data("../thesis/dataset/1000e_3k_2k_1k_100_lyr1",g)
77      # # plot layer 1
78      # ekstraktor.simpan_data("../thesis/dataset/1000e_3k_2k_1k_100_lyr1",g)
79      # plot layer 1
80      plt.ylabel("Cost")
81      plt.xlabel("Epoch")
82      plt.title("Pretraining model 7k 10k 5k 1k Layer 0 (7000 Hidden)")
83      g_0 = load_epch_layer(f, 1000, 0)
84      plt.plot(g_0[10:, 0], g_0[10:, 1])
85      plt.show()
86
87      plt.ylabel("Cost")
88      plt.xlabel("Epoch")
89      plt.title("Pretraining model 7k 10k 5k 1k Layer 1 (10000 Hidden)")
90      g_1 = load_epch_layer(f, 1000, 1)
91      plt.plot(g_1[10:, 0], g_1[10:, 1])
92      plt.show()
93
94      plt.ylabel("Cost")
95      plt.xlabel("Epoch")
96      plt.title("Pretraining model 7k 10k 5k 1k Layer 2 (5000 Hidden)")
97      g_2 = load_epch_layer(f, 1000, 2)
98      plt.plot(g_2[10:, 0], g_2[10:, 1])
99      plt.show()
100
101     plt.ylabel("Cost")
102     plt.xlabel("Epoch")
103     plt.title("Pretraining model 7k 10k 5k 1k Layer 3 (1000 Hidden)")
104     g_3 = load_epch_layer(f, 1000, 3)
105     plt.plot(g_3[10:, 0], g_3[10:, 1])
106     plt.show()
107
108     f = load_file_ekstrak_layer_epoch_cost("../thesis/dataset/log1000e_10k_5k_1k_500.txt")
109     ekstraktor = eks.Ekstraktor()
110     #
111     # # ekstraktor.simpan_data("../thesis/dataset/1000e_3k_2k_1k_100_lyr1",g)
112     # # plot layer 1
113     # ekstraktor.simpan_data("../thesis/dataset/1000e_3k_2k_1k_100_lyr1",g)
114     # plot layer 1
115     plt.ylabel("Cost")
116     plt.xlabel("Epoch")
117     plt.title("Pretraining model 10k 5k 1k 500 Layer 0 (10000 Hidden)")
118     g_0 = load_epch_layer(f, 1000, 0)
119     plt.plot(g_0[10:, 0], g_0[10:, 1])
120     plt.show()
121
122     plt.ylabel("Cost")
123     plt.xlabel("Epoch")
124     plt.title("Pretraining model 10k 5k 1k 500 Layer 1 (5000 Hidden)")
125     g_1 = load_epch_layer(f, 1000, 1)
126     plt.plot(g_1[10:, 0], g_1[10:, 1])
127     plt.show()
128
129     plt.ylabel("Cost")
130     plt.xlabel("Epoch")
131     plt.title("Pretraining model 10k 5k 1k 500 Layer 2 (1000 Hidden)")
132     g_2 = load_epch_layer(f, 1000, 2)
133     plt.plot(g_2[10:, 0], g_2[10:, 1])
134     plt.show()
135
136     plt.ylabel("Cost")
137     plt.xlabel("Epoch")
138     plt.title("Pretraining model 10k 5k 1k 500 Layer 3 (500 Hidden)")
139     g_3 = load_epch_layer(f, 1000, 3)
140     plt.plot(g_3[10:, 0], g_3[10:, 1])
141     plt.show()
```

Listing 3 : Implementasi melakukan plotting untuk epoch 2000

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3  import thesis.ekstrak_csv as eks
```

```
4
5  kamus = {"Pre-training_layer": "", "epoch": "", "cost": "", "\n": ""}
6
7
8  def replace_all(text, dic):
9      for i, j in dic.iteritems():
10         text = text.replace(i, j)
11     return text
12
13
14 def load_file_text(nama_file):
15     text_file = open(nama_file, "r")
16     lst = text_file.readlines()
17     a = np.array([replace_all(lst[0], kamus).split(",")], float)
18     for i in range(1,len(lst)):
19         b = np.array([replace_all(lst[i], kamus).split(",")], float)
20         a = np.r_[a,b]
21     return a
22
23 def load_epch_layer(mat, jml_epoch, layer):
24     return mat[(layer*jml_epoch):((layer+1)*jml_epoch),1:3]
25
26 def load_file_ekstrak_layer_epoch_cost(nama_file_log_test):
27     c = load_file_text(nama_file_log_test)
28     return c
29
30 if __name__ == '__main__':
31     # contoh pemakaian load matrix
32     # edit file log sampai hanya ada layer epoch dan cost saja
33     # simpan dengan kode jml epoch layer
34     # load file log dengan :
35
36     # f = load_file_ekstrak_layer_epoch_cost("../thesis_test/dataset_test/log_test.log")
37     # g = load_epch_layer(f, 2, 1)  # g = matrix dengan isi epoch dan cost pada layer 2
38
39
40     f = load_file_ekstrak_layer_epoch_cost("../thesis/dataset/logout2000e_3k_2k_1k_100.txt")
41     ekstraktor = eks.Ekstraktor()
42
43     # ekstraktor.simpan_data("../thesis/dataset/1000e_3k_2k_1k_100_lyr1",g)
44     # plot layer 1
45     plt.ylabel("Cost")
46     plt.xlabel("Epoch")
47     plt.title("Pretraining_model_3k_2k_1k_100_Layer_0_(3000_Hidden)")
48     g_0 = load_epch_layer(f,2000,0)
49     plt.plot(g_0[10:,0],g_0[10:,1])
50     plt.show()
51
52     plt.ylabel("Cost")
53     plt.xlabel("Epoch")
54     plt.title("Pretraining_model_3k_2k_1k_100_Layer_1_(2000_Hidden)")
55     g_1 = load_epch_layer(f, 2000, 1)
56     plt.plot(g_1[10:, 0], g_1[10:, 1])
57     plt.show()
58
59     plt.ylabel("Cost")
60     plt.xlabel("Epoch")
61     plt.title("Pretraining_model_3k_2k_1k_100_Layer_2_(2000_Hidden)")
62     g_2 = load_epch_layer(f, 2000, 2)
63     plt.plot(g_2[10:, 0], g_2[10:, 1])
64     plt.show()
65
66     plt.ylabel("Cost")
67     plt.xlabel("Epoch")
68     plt.title("Pretraining_model_3k_2k_1k_100_Layer_3_(100_Hidden)")
69     g_3 = load_epch_layer(f, 2000, 3)
70     plt.plot(g_3[10:, 0], g_3[10:, 1])
71     plt.show()
72
73     f = load_file_ekstrak_layer_epoch_cost("../thesis/dataset/logout2000e_7k_10k_5k_1k.txt")
74     ekstraktor = eks.Ekstraktor()
75     #
76     # # ekstraktor.simpan_data("../thesis/dataset/1000e_3k_2k_1k_100_lyr1",g)
77     # # plot layer 1
78     # ekstraktor.simpan_data("../thesis/dataset/1000e_3k_2k_1k_100_lyr1",g)
79     # plot layer 1
80     plt.ylabel("Cost")
```

```
    plt.xlabel("Epoch")
    plt.title("Pretraining model 7k 10k 5k 1k Layer 0 (7000 Hidden)")
    g_0 = load_epch_layer(f, 2000, 0)
    plt.plot(g_0[10:, 0], g_0[10:, 1])
    plt.show()

    plt.ylabel("Cost")
    plt.xlabel("Epoch")
    plt.title("Pretraining model 7k 10k 5k 1k Layer 1 (10000 Hidden)")
    g_1 = load_epch_layer(f, 2000, 1)
    plt.plot(g_1[10:, 0], g_1[10:, 1])
    plt.show()

    plt.ylabel("Cost")
    plt.xlabel("Epoch")
    plt.title("Pretraining model 7k 10k 5k 1k Layer 2 (5000 Hidden)")
    g_2 = load_epch_layer(f, 2000, 2)
    plt.plot(g_2[10:, 0], g_2[10:, 1])
    plt.show()

    plt.ylabel("Cost")
    plt.xlabel("Epoch")
    plt.title("Pretraining model 7k 10k 5k 1k Layer 3 (1000 Hidden)")
    g_3 = load_epch_layer(f, 2000, 3)
    plt.plot(g_3[10:, 0], g_3[10:, 1])
    plt.show()

    f = load_file_ekstrak_layer_epoch_cost("../thesis/dataset/logout2000e_10k_5k_1k_500.txt")
    ekstraktor = eks.Ekstraktor()
    #
    # # ekstraktor.simpan_data("../thesis/dataset/1000e_3k_2k_1k_100_lyr1",g)
    # # plot layer 1
    # ekstraktor.simpan_data("../thesis/dataset/1000e_3k_2k_1k_100_lyr1",g)
    # plot layer 1
    plt.ylabel("Cost")
    plt.xlabel("Epoch")
    plt.title("Pretraining model 10k 5k 1k 500 Layer 0 (10000 Hidden)")
    g_0 = load_epch_layer(f, 2000, 0)
    plt.plot(g_0[10:, 0], g_0[10:, 1])
    plt.show()

    plt.ylabel("Cost")
    plt.xlabel("Epoch")
    plt.title("Pretraining model 10k 5k 1k 500 Layer 1 (5000 Hidden)")
    g_1 = load_epch_layer(f, 2000, 1)
    plt.plot(g_1[10:, 0], g_1[10:, 1])
    plt.show()

    plt.ylabel("Cost")
    plt.xlabel("Epoch")
    plt.title("Pretraining model 10k 5k 1k 500 Layer 2 (1000 Hidden)")
    g_2 = load_epch_layer(f, 2000, 2)
    plt.plot(g_2[10:, 0], g_2[10:, 1])
    plt.show()

    plt.ylabel("Cost")
    plt.xlabel("Epoch")
    plt.title("Pretraining model 10k 5k 1k 500 Layer 3 (500 Hidden)")
    g_3 = load_epch_layer(f, 2000, 3)
    plt.plot(g_3[10:, 0], g_3[10:, 1])
    plt.show()
```

Listing 4 : Implementasi melakukan perankingan pada model percobaan 1 :

```
import multistep_rank as mtr
import theano.tensor as T
import numpy as np
from ekstrak_csv import Ekstraktor


# buat function :
# hsl_ranking = multisteprank(model, [100,100,100]):

ekstraktor = Ekstraktor()

model =    ekstraktor.load_data("./dataset/model1000e_3k_2k_1k_100.pkl.gz")
```

```
print 'Jumlah layer : %i' % (model.n_layers)

Wlayer3 = model.rbm_layers[3].W
Wlayer2 = model.rbm_layers[2].W
Wlayer1 = model.rbm_layers[1].W
Wlayer0 = model.rbm_layers[0].W
# Wlayer1.shape.eval()

y3 = Wlayer3.get_value(True)
x3 = T.fmatrix()
x3 = y3.copy()

# ranking ujung
awal3 = mtr.awal(x3)
jml_bobot3 = mtr.jumlah_bobot(x3, awal3)
ranking_jml_bobot3 = mtr.rank_hasil_jumlah(jml_bobot3)
top_n3 = mtr.set_top_n(ranking_jml_bobot3,70)

# print "layer 3"
# print 'hasil perankingan top 50: '
# print ranking_jml_bobot3[:50]
# print 'set top n dengan 1 : '
# print top_n3.astype(int)

y2 = Wlayer2.get_value(True)
x2 = y2.copy()
awal2 = mtr.extract_top_n(top_n3)
jml_bobot2 = mtr.jumlah_bobot(x2, awal2)
ranking_jml_bobot2 = mtr.rank_hasil_jumlah(jml_bobot2)
top_n2 = mtr.set_top_n(ranking_jml_bobot2,700)

# print "layer 2"
# print 'hasil perankingan top 50: '
# print ranking_jml_bobot2[:50]
# print 'set top n dengan 1 : '
# print top_n2.astype(int)

y1 = Wlayer1.get_value(True)
x1 = y1.copy()
awal1 = mtr.extract_top_n(top_n2)
jml_bobot1 = mtr.jumlah_bobot(x1, awal1)
ranking_jml_bobot1 = mtr.rank_hasil_jumlah(jml_bobot1)
top_n1 = mtr.set_top_n(ranking_jml_bobot1,1500)

# print "layer 1"
# print 'hasil perankingan top 50: '
# print ranking_jml_bobot1[:50]
# print 'set top n dengan 1 : '
# print top_n1.astype(int)

y0 = Wlayer0.get_value(True)
x0 = y0.copy()
awal0 = mtr.extract_top_n(top_n1)
jml_bobot0 = mtr.jumlah_bobot(x0, awal0)
ranking_jml_bobot0 = mtr.rank_hasil_jumlah(jml_bobot0)
# top_n0 = mtr.set_top_n(ranking_jml_bobot0,70)

print "layer visible"
print 'hasil perankingan top 250 layer visible 3k 2k 1k 100: '
print ranking_jml_bobot0[:250,0].astype(int)
```

Listing 5 : Implementasi melakukan perankingan pada model percobaan 2 :

```
import multistep_rank as mtr
import theano.tensor as T
import numpy as np
from ekstrak_csv import Ekstraktor


# buat function :
# hsl_ranking = multisteprank(model, [100,100,100]):

ekstraktor = Ekstraktor()

model =    ekstraktor.load_data("./dataset/model1000e_7k_10k_5k_1k.pkl.gz")
print 'Jumlah layer : %i' % (model.n_layers)
```

```
14
15 Wlayer3 = model.rbm_layers[3].W
16 Wlayer2 = model.rbm_layers[2].W
17 Wlayer1 = model.rbm_layers[1].W
18 Wlayer0 = model.rbm_layers[0].W
19 # Wlayer1.shape.eval()
20
21 y3 = Wlayer3.get_value(True)
22 x3 = T.fmatrix()
23 x3 = y3.copy()
24
25 # ranking ujung
26 awal3 = mtr.awal(x3)
27 jml_bobot3 = mtr.jumlah_bobot(x3, awal3)
28 ranking_jml_bobot3 = mtr.rank_hasil_jumlah(jml_bobot3)
29 top_n3 = mtr.set_top_n(ranking_jml_bobot3,500)
30
31 # print "layer 3"
32 # print 'hasil perankingan top 50: '
33 # print ranking_jml_bobot3[:50]
34 # print 'set top n dengan 1 : '
35 # print top_n3.astype(int)
36
37 y2 = Wlayer2.get_value(True)
38 x2 = y2.copy()
39 awal2 = mtr.extract_top_n(top_n3)
40 jml_bobot2 = mtr.jumlah_bobot(x2, awal2)
41 ranking_jml_bobot2 = mtr.rank_hasil_jumlah(jml_bobot2)
42 top_n2 = mtr.set_top_n(ranking_jml_bobot2,2500)
43
44 # print "layer 2"
45 # print 'hasil perankingan top 50: '
46 # print ranking_jml_bobot2[:50]
47 # print 'set top n dengan 1 : '
48 # print top_n2.astype(int)
49
50 y1 = Wlayer1.get_value(True)
51 x1 = y1.copy()
52 awal1 = mtr.extract_top_n(top_n2)
53 jml_bobot1 = mtr.jumlah_bobot(x1, awal1)
54 ranking_jml_bobot1 = mtr.rank_hasil_jumlah(jml_bobot1)
55 top_n1 = mtr.set_top_n(ranking_jml_bobot1,1500)
56
57 # print "layer 1"
58 # print 'hasil perankingan top 50: '
59 # print ranking_jml_bobot1[:50]
60 # print 'set top n dengan 1 : '
61 # print top_n1.astype(int)
62
63 y0 = Wlayer0.get_value(True)
64 x0 = y0.copy()
65 awal0 = mtr.extract_top_n(top_n1)
66 jml_bobot0 = mtr.jumlah_bobot(x0, awal0)
67 ranking_jml_bobot0 = mtr.rank_hasil_jumlah(jml_bobot0)
68 top_n0 = mtr.set_top_n(ranking_jml_bobot0,7000)
69
70 print "layer visible"
71 print 'hasil perankingan top 250 visible 7k 10k 5k 1k: '
72 print ranking_jml_bobot0[:250,0].astype(int)
```

Listing 6 : Implementasi melakukan perankingan pada model percobaan 3 :

```
1  import multistep_rank as mtr
2  import theano.tensor as T
3  import numpy as np
4  from ekstrak_csv import Ekstraktor
5
6
7  # buat function :
8  # hsl_ranking = multisteprank(model, [100,100,100]):
9
10 ekstraktor = Ekstraktor()
11
12 model =   ekstraktor.load_data("./dataset/model1000e_10k_5k_1k_500.pkl.gz")
13 print 'Jumlah layer : %i' % (model.n_layers)
14
```

```
Wlayer3 = model.rbm_layers[3].W
Wlayer2 = model.rbm_layers[2].W
Wlayer1 = model.rbm_layers[1].W
Wlayer0 = model.rbm_layers[0].W
# Wlayer1.shape.eval()

y3 = Wlayer3.get_value(True)
x3 = T.fmatrix()
x3 = y3.copy()

# ranking ujung
awal3 = mtr.awal(x3)
jml_bobot3 = mtr.jumlah_bobot(x3, awal3)
ranking_jml_bobot3 = mtr.rank_hasil_jumlah(jml_bobot3)
top_n3 = mtr.set_top_n(ranking_jml_bobot3,70)

# print "layer 3"
# print 'hasil perankingan top 50: '
# print ranking_jml_bobot3[:50]
# print 'set top n dengan 1 : '
# print top_n3.astype(int)

y2 = Wlayer2.get_value(True)
x2 = y2.copy()
awal2 = mtr.extract_top_n(top_n3)
jml_bobot2 = mtr.jumlah_bobot(x2, awal2)
ranking_jml_bobot2 = mtr.rank_hasil_jumlah(jml_bobot2)
top_n2 = mtr.set_top_n(ranking_jml_bobot2,700)

# print "layer 2"
# print 'hasil perankingan top 50: '
# print ranking_jml_bobot2[:50]
# print 'set top n dengan 1 : '
# print top_n2.astype(int)

y1 = Wlayer1.get_value(True)
x1 = y1.copy()
awal1 = mtr.extract_top_n(top_n2)
jml_bobot1 = mtr.jumlah_bobot(x1, awal1)
ranking_jml_bobot1 = mtr.rank_hasil_jumlah(jml_bobot1)
top_n1 = mtr.set_top_n(ranking_jml_bobot1,1500)

# print "layer 1"
# print 'hasil perankingan top 50: '
# print ranking_jml_bobot1[:50]
# print 'set top n dengan 1 : '
# print top_n1.astype(int)

y0 = Wlayer0.get_value(True)
x0 = y0.copy()
awal0 = mtr.extract_top_n(top_n1)
jml_bobot0 = mtr.jumlah_bobot(x0, awal0)
ranking_jml_bobot0 = mtr.rank_hasil_jumlah(jml_bobot0)
top_n0 = mtr.set_top_n(ranking_jml_bobot0,70)

print "layer visible"
print 'hasil perankingan top 250 layer visible 10k 5k 1k 500: '
print ranking_jml_bobot0[:250,0].astype(int)
```

Listing 6 : Implementasi diagram venn untuk percobaan 1, 2 dan 3 :

```
from __future__ import print_function
from matplotlib import pyplot as plt
import numpy as np
from matplotlib_venn import venn3, venn3_circles, venn2, venn2_circles
from ekstrak_csv import Generator, Ekstraktor
# plt.figure(figsize=(4,4))
# v = venn3(subsets=(1, 1, 1, 1, 1, 1, 1), set_labels = ('A', 'B', 'C'))
# v.get_patch_by_id('100').set_alpha(1.0)
# v.get_patch_by_id('100').set_color('white')
# v.get_label_by_id('100').set_text('Unknown')
# v.get_label_by_id('A').set_text('Set "A"')
# c = venn3_circles(subsets=(1, 1, 1, 1, 1, 1, 1), linestyle='dashed')
# c[0].set_lw(1.0)
# c[0].set_ls('dotted')
# plt.title("Sample Venn diagram")
```

```
# plt.annotate('Unknown set', xy=v.get_label_by_id('100').get_position() - np.array([0, 0.05]), xytext=(-70,-70),
#              ha='center', textcoords='offset points', bbox=dict(boxstyle='round,pad=0.5', fc='gray', alpha=0.1),
#              arrowprops=dict(arrowstyle='->', connectionstyle='arc3,rad=0.5',color='gray'))
# plt.show()

# # hasil perankingan top 50 visible 7k 10k 5k 1k:
# set1 = set([1019,21919,12172,6084,460,328,201,2635,11120,13246,11298,20968,
#             8350,1418,8262,344,46,11323,141,21860,10428,243,8137,88,8218,598,17096,
#             501,160,22276,13034,307,887,371,781,24,11570,15602,11110,112,11606,1556,956,21107,
#             7809,18198,2071,959,14530,8366])
#
# # hasil perankingan top 50 layer visible 10k 5k 1k 500:
# set2 = set([12253,2540,13765,328,21234,15890,4196,13246,49,14566,398,22275,11329,11370,641,
#             3274,377,10793,21919,664,2176,2549,5375,12332,8473,14362,4418,137,8181,32,1631,
#             464,16598,9965,27,11314,61,861,39,10112,1019,101,12727,11298,50,4064,8135,54,511,1521])
#
# # hasil perankingan top 50 layer visible 3k 2k 1k 100:
# set3 = set([328,12172,12253,2540,7303,8189,13246,11120,344,22230,201,11356,15602,820,10718,
#             1019,8633,115,9295,15712,14686,1077,4501,10934,11032,18198,11570,244,24,7809,4418,
#             10110,39,8262,21011,275,17199,14210,576,460,6084,9108,8627,8282,
#             268,1418,11327,6682,470,145])
#
# set4 = set([22205,9046,3922,5734,3797,4203,9564,21773,6227,19082,
#             4726,4457,18576,16415,9360,3799,8475,12485,8962,8474,18960,
#             1639,1067,7067,3506,9103,5461,4883,6006,6267,4245,9778,6226,
#             13883,5808,5305,19390,5594,4257,6013,4008,5478,12831,14826,15704,
#             5733,5593,405,18028,2593])
#
set_top_250 = set([22205,9046,3922,5734,3797,4203,9564,21773,6227,19082,4726,4457,18576,16415,9360,3799,8475,12485,
                   8962,8474,18960,1639,1067,7067,3506,9103,5461,4883,6006,6267,4245,9778,6226,13883,5808,5305,19390,
                   5594,4257,6013,4008,5478,12831,14826,15704,5733,5593,405,18028,2593,21985,3391,9326,18427,12617,
                   2051,4168,13352,9777,438,8961,8663,12696,8660,8106,2404,9382,4420,1066,5618,5692,21106,20040,4105,
                   20357,2405,3338,17261,5556,11482,19533,18454,12628,18815,3994,17145,2286,20128,18803,18799,2591,
                   22055,5392,4910,13513,21968,11309,18086,9379,18183,21962,959,12550,21991,1104,20495,9389,3088,
                   15290,8476,8776,4773,11496,12697,12206,4501,18313,3097,16801,875,21107,2274,9325,18422,19229,6183,
                   13278,3679,227,13469,9101,17710,11456,8985,12097,4852,20494,2849,17235,18734,22155,4829,18593,7281,
                   5033,3339,12114,8130,5158,9035,19184,12921,5252,22158,3361,6478,18678,12451,177,21876,3075,9102,
                   18530,4096,13094,5372,1336,2616,9250,4908,13223,3891,1532,18168,11125,12020,2273,12357,12797,
                   12440,18528,3413,6024,540,19366,9021,4931,3270,4481,21576,9197,17135,21992,12761,3680,2436,4529,
                   9031,3775,5504,18074,19019,3074,10222,8973,4430,9487,5134,19045,22231,16889,3598,11737,10121,21200,
                   10673,18080,5080,4997,4959,819,18019,12836,19607,5695,16543,758,289,12999,1995,18456,5508,5409,
                   2534,22147,9579,12064,11738,322,17173,11965,13421,6176,3414,19140,13765,5024,18138,4238,11372])
#
# venn3([set1, set2, set3], ('H 7k 10k 5k 1k ', 'H 10k 5k 1k 500', 'H 3k 2k 1k 100'))
# plt.show()
#
# venn2([set2, set_top_250], ('H 10k 5k 1k 500k ', 'Literatur'))
# plt.show()
#
# venn2([set3, set_top_250], ('H 3k 2k 1k 100k ', 'Literatur'))
# plt.show()

# hasil perankingan top 250 layer visible 10k 5k 1k 500:
set1_250 = set([12253, 2540,13765, 328,21234,15890, 4196,13246,  49,14566, 398,22275, 11329, 11370,
641, 3274,
                377,10793,21919,  664, 2176, 2549, 5375,12332,  8473,14362, 4418,  137, 8181,  32, 1631,
464,16598,
                9965,  27,11314,  61,  861,  39,10112, 1019,  101,12727,11298,  50, 4064, 8135,  54,
511, 1521,
                102, 6279,  318,  405,10484, 3964,12118,15612,11120,13695, 18313,  454, 1716, 1192,22233,
959,21239,
                241, 8249, 6084,11484,  953, 14210,  663,  201,  24,10224, 1556,11068,17116,  232,10513,
46, 8423,
                1833, 1518, 2746, 2586,  254, 8310, 1516,11937,  115,13732,11330,11577, 13625,21860,11756,17539,
                300,1077 ,2660,13706, 1250,18094,  887,10516, 10234,  114,17452,20968,
331, 3470,13741, 8980,13034,
                11969,17209,  397,  9666, 8564,  330,11537,21182,13408,12736,  428,12410,  307,13006,21115,
863,
                243, 1615, 9904, 8582, 8416,11669, 7485, 8125,21513, 3068,21588, 17612,11976, 2323,11208,11149,16820,
                971,12757,11538,  268,12698, 9384,  8364, 3483, 2014, 1548,14551,11435,22142,18988,
568, 1123,21090,
                11810, 21927, 4313, 6016,  645, 6380,12159,20577,13117,
703,22227, 1481,17136, 12967,13256,21691,
                17156,11433,12370,18783, 6115,  47,  159, 1418,  833, 22246, 2635, 2476, 2871,15808, 8185, 2004,
                338,10428, 3800,13602,11796, 11695,20846,  223, 5734,10934, 1901,
879, 2082, 8494, 6843,11060, 1896,
                22108,10303,14797, 2163, 9108, 5150,17405, 7905,  141,12360, 8438, 1306,
```

```
1795, 1428,12697,12109,
                7303,12149,10961,16620,13551,11420, 1033,17786, 13729,  261,12184, 1478, 1445,12953,18035,
                633,10690,11462])

# hasil perankingan top 250 visible 7k 10k 5k 1k:
set2_250 = set ([ 1019,21919,12172, 6084,  460,  328,  201, 2635,11120,13246, 11298,20968,  8350, 1418, 8262,
                  344,   46,11323,  141,21860,10428,  243,  8137,   88,  8218,  598,17096,  501,
160,22276,13034,
                  307,  887,  371,   781,   24, 11570,15602,11110,  112,11606, 1556,  956,21107, 7809,18198,
2071,
                  959, 14530, 8366, 3557, 1087,13765,  925,  322, 7361,  184,13025,  202,11497,
8319, 4270,15812,
                  13741,  245, 6279, 3677,11770, 5150, 3745,  308, 7622,  1176,
799,12871,11332, 8078, 6688,11248,
                  8647,  101,10110,18086,12857,  7303,11775,  954, 8442,22230, 1192,
321,22268,17127,12466,21784,
                  15890, 12595,  275, 1366, 4418,  366,14044,  507, 9754,17967, 1029, 8328, 8631, 21031, 8220,
                  50,11695, 1433, 6880,17327, 3171,17465,17343,11370,  849, 15963,11796,17137,
204, 8197,17590,
                  797,12418,  232, 8352,  588, 4562,  1604, 8306, 5243,11147,12673, 8627,
186, 1741,17143, 2906,
                  17216,17473,  319,11612,  511,   77,13817,10706,12727,  100,17339,  290,
265, 8594, 20592,12483,
                  1388,  474,10688,13268, 3546,16762,13207,  274,   45,  114,  6855,
453,13260,18624,12794,12419,
                  728,  577,13078, 1313,  454,18583,  2186, 8268,  879,21976,17446, 8192, 8830, 1766, 8723,11032,
                  1548,11149, 21011,   78,  744,11489,17104,  845,  187, 4196,17457,  145, 4518, 1637,
105,12912,
                  1010,11372,11703,12045,  846,11402, 5324, 1908,  536, 9697, 17134,
115, 9459, 8258, 2533, 8994,
                  11001, 8299, 9666,  393, 2237, 8633, 14772, 3499, 9148,11832,
445, 9911, 1955, 8139,13617,  548,
                  15799, 5333,  8625, 2032, 3891,14752,20743, 7603, 8381,10353,12535,22229])

# hasil perankingan top 250 layer visible 3k 2k 1k 100:
set3_250 = set ([  328,12172,12253, 2540, 7303, 8189,13246,11120,  344,22230,  201,11356, 15602,
820,10718, 1019,
                   8633,  115, 9295,15712,14686, 1077, 4501,10934, 11032,18198,11570,  244,
24, 7809, 4418,10110,
                   39, 8262,21011,  275, 17199,14210,  576,  460, 6084, 9108, 8627, 8282,
268, 1418,11327, 6682,
                   470,  145,   61,21031,11144,22014,14312, 9904,12332,13765,21860,   38, 10112, 8450,
742, 1105,
                   12911,11119,  107,  396, 1521, 9965,12466,17212,  1127,  697, 3700, 3283,
231, 1155,11354, 2636,
                   8611,11376,  971,10501, 13741, 8712,  405,11817,  334,17096,13706,
623,21992, 4332,16762, 1332,
                   22233,21970,11775, 8306,11329, 9666,10893,11417,17327, 6790,13732,17743, 13191,12316,
245,14449,
                   990,12174,   51, 5180,  346,11110,12779,11639,  398, 1414, 9292, 5508, 8568,
879, 8136,  81,
                   232,11669,11704,13239, 11580,  36,  202,  940,16203, 8220, 3513, 8388,14549,15890,11710, 8491,
                   16459,12568, 8339,  445,10490, 1863,17967,10303, 8416,  101,17539,
46, 21098, 8319,14484, 7816,
                   17219, 2804,18988,18103, 2071, 8381,  494, 2171,  184,15334,  989,11065,17684,
281,17369, 4518,
                   308, 7105, 8280, 1021, 11500,17405, 3325,11370,  204,17348, 5243,22222,12794,20968, 8268,
220,
                   21919,11433,  532, 8995,11810,20323, 8137, 9692,12952, 5073, 5986,11147, 11436, 8249,17747, 2897,
                   9937,11576,12727,11367, 1298,22276,  307,  953,   57,  728, 1636, 9986,11314,11298,
621, 8124,
                   8564,  127, 3278,14375, 22023,10601, 1960,  305,17862,
371,17577,12045, 3544,10236,10395, 3758,
                   511,  538,11497,11658,21905, 8196, 9905,  720, 7603,21819])
def plot_venn():
    venn2([set1_250, set_top_250], ('P 1 H 10k 5k 1k 500 ', 'Literatur'))
    plt.title("Top 250 percobaan 1, dengan teknik di literatur 1")
    plt.show()

    venn2([set2_250, set_top_250], ('P 2 H 7k 10k 5k 1k ', 'Literatur'))
    plt.title("Top 250 percobaan 2, dengan teknik di literatur 1")
    plt.show()


    venn2([set3_250, set_top_250], ('P 3 H 3k 2k 1k 100 ', 'Literatur'))
    plt.title("Top 250 percobaan 3, dengan teknik di literatur 1")
    plt.show()
```

```
134
135     # venn3([set1_250, set2_250, set3_250], ('P1 H=10k 5k 1k 500', 'P2 H=7k 10k 5k 1k', 'P3 H=3k 2k 1k 100'))
136     # plt.title("Perangkingan top 250")
137     # plt.show()
138
139 if __name__ == '__main__':
140     set_all = set1_250 & set2_250 & set3_250
141     ekstraktor = Ekstraktor()
142     generator = Generator()
143     array_rank = np.array(list(set3_250))
144     # train = 70.5
145     # valid = 15.5
146     # test = 14
147     # ekstraktor.norm_dataset("./dataset/GSE10072_dataset") # dataset asli untuk dilakukan normalisasi
148     # dataset_gse = np.genfromtxt("./dataset/GSE10072_dataset_norm.csv", dtype=float, delimiter=",")
149     #  hasil dimasukan ke var dataset_gse
150     # generator.top_n_dataset(array_rank, dataset_gse, "./dataset/GSE10072_dataset_rank_set3")
151     #  generate dataset dari rankingnya
152     # dataset_gse = ekstraktor.generate_dataset("./dataset/GSE10072_dataset_rank_set3",
153     #                                           "./dataset/GSE10072_TARGET", train, valid, test, True)
154     #
155     # plot_venn()
156     print(set_all)
```

Listing 7 : Implementasi Ekstraktor :

```
1  from sklearn import preprocessing
2  from sklearn import utils
3  import numpy as np
4  import gzip, cPickle
5  from utilitas import top_n_dataset
6
7  class Salah(Exception):
8      pass
9
10 class Ekstraktor:
11     nama_file = str
12     data = np.empty
13     target_file = str
14     y = np.empty
15     jumlah_data = int
16     def norm_dataset(self,nama_file):
17         self.nama_file = nama_file + ".csv"
18         self.data = np.genfromtxt(self.nama_file, dtype=float, delimiter=",")
19         min_max_scaler = preprocessing.normalize(self.data)
20         #min_max_scaler = preprocessing.scale(self.data)
21         #min_max_scaler = preprocessing.minmax_scale(self.data)
22         np.savetxt(nama_file + "_norm.csv", min_max_scaler, delimiter=",")
23
24     def generate_dataset(self, nama_file, target_file,train,valid,test, suffle = True):
25         self.nama_file = nama_file + ".csv"
26         self.target_file = target_file + ".csv"
27         self.data = np.genfromtxt(self.nama_file, dtype=float, delimiter=',')
28         self.y = np.genfromtxt(self.target_file, dtype=float, delimiter=',')
29         self.data = self.data.transpose()
30         self.jumlah_data = self.ambil_jumlah_dataset(self.data)
31         jml_train,jml_valid,jml_test =self.ambil_train_valid_test(self.jumlah_data,train,valid,test)
32         if suffle:
33             self.data, self.y = utils.shuffle(self.data,self.y,random_state = 5)
34         train_set_x = self.data[0:jml_train]
35         valid_set_x = self.data[jml_train+1:jml_train+1+jml_valid]
36         test_set_x = self.data[jml_train+1+jml_valid+1:jml_train+1+jml_valid+1+jml_test]
37         train_set_y = self.y.transpose()[2][0:jml_train]
38         valid_set_y = self.y.transpose()[2][jml_train+1:jml_train+1+jml_valid]
39         test_set_y = self.y.transpose()[2][jml_train+1+jml_valid+1:jml_train+1+jml_valid+1+jml_test]
40         train_set = train_set_x, train_set_y
41         valid_set = valid_set_x, valid_set_y
42         test_set = test_set_x, test_set_y
43         dataset = [train_set, valid_set, test_set]
44         self.simpan_data(self.nama_file + '_dataset.pkl.gz',dataset)
45         return dataset
46
47     def ambil_jumlah_dataset(self,data):
48         return data.shape[0]
49
50     def ambil_train_valid_test(self,jml_dataset,train,valid,test):
```

```
        # ambil train valid test dalam %
        if int(round(train+valid+test)) != 100 :
            raise Salah("train+valid+test harus = 100%")
        jml_train_set = int(round(float(jml_dataset)*(float(train)/100.)))
        jml_valid_set = int(round(float(jml_dataset)*(float(valid)/100.)))
        jml_test_set  = int(round(float(jml_dataset)*(float(test)/100.)))
        return jml_train_set,jml_valid_set,jml_test_set

    def simpan_data(self, n_file, data_simpan):
        f = gzip.open(n_file, 'wb')
        cPickle.dump(data_simpan, f, protocol=2)
        f.close()
        return data_simpan

    def load_data(self, data):
        # model_hasil = load cpickel
        f = gzip.open(data, 'rb')
        model_hasil = cPickle.load(f)
        return model_hasil

class Generator:
    ekstraktor = Ekstraktor()
    # data_rank adalah array dari ranking data
    def top_n_dataset(self, data_rank,dataset, namafile):
        data_hasil = top_n_dataset(data_rank, dataset)
        np.savetxt(namafile + ".csv", data_hasil, delimiter=",")
        return data_hasil

if __name__ == '__main__':
    ekstraktor = Ekstraktor()
    generator = Generator()
    array_rank= np.array([2, 3])
    train = 80.5
    valid = 14.5
    test = 5
    ekstraktor.norm_dataset("./dataset/iris_dataset")
    dataset_iris = np.genfromtxt("./dataset/iris_dataset_norm.csv", dtype=float, delimiter=",")
    generator.top_n_dataset(array_rank, dataset_iris,"./dataset/iris_dataset_rank")
    dataset_iris = ekstraktor.generate_dataset("./dataset/iris_dataset_rank",
                                        "./dataset/iris_target", train, valid, test, True)

    print dataset_iris
    # ekstraktor.norm_dataset("./dataset/GSE10072_dataset")
```

Listing 8 : Implementasi Melakukan training model :

```
import gzip, cPickle
import numpy as np
import six.moves.cPickle as pickle

import gc
import sys
from logger import Logger
from ekstrak_csv import Ekstraktor
from DBN import test_DBN


ekstraktor = Ekstraktor()

def percobaan1_4l_2000e():
    finetune_lr=0.1
    pretraining_epochs=2000
    pretrain_lr=0.01
    k=1
    training_epochs=100
    dataset='./dataset/gse10072.pkl.gz'
    batch_size= 5
    n_v=22283
    n_output=2

    # percobaan 1 dengan layer 10k 5k 1k 500
    sys.stdout = Logger("./dataset/logout2000e_10k_5k_1k_500.txt")
    hidden_sizes=[10000, 5000, 1000, 500]
    model_hasil = test_DBN(finetune_lr, pretraining_epochs,
                  pretrain_lr, k, training_epochs,
                  dataset, batch_size,hidden_sizes, n_v,n_output)
```

```
31
32      ekstraktor.simpan_data("./dataset/model2000e_10k_5k_1k_500.pkl.gz", model_hasil)
33      del model_hasil
34      gc.collect()
35
36      # percobaan 2
37      sys.stdout = Logger("./dataset/logout2000e_7k_10k_5k_1k.txt")
38
39      hidden_sizes=[7000, 10000, 5000, 1000]
40      model_hasil = test_DBN(finetune_lr, pretraining_epochs,
41                  pretrain_lr, k, training_epochs,
42                  dataset, batch_size,hidden_sizes, n_v,n_output)
43
44      ekstraktor.simpan_data("./dataset/model2000e_7k_10k_5k_1k.pkl.gz", model_hasil)
45      del model_hasil
46      gc.collect()
47
48      # percobaan 3
49      sys.stdout = Logger("./dataset/logout2000e_3k_2k_1k_100.txt")
50      hidden_sizes=[3000, 2000, 1000, 100]
51      model_hasil = test_DBN(finetune_lr, pretraining_epochs,
52                  pretrain_lr, k, training_epochs,
53                  dataset, batch_size,hidden_sizes, n_v,n_output)
54
55      ekstraktor.simpan_data("./dataset/model2000e_3k_2k_1k_100.pkl.gz", model_hasil)
56      del model_hasil
57      gc.collect()
58
59  def percobaan2_3l_1000e():
60      finetune_lr=0.1
61      pretraining_epochs=500
62      pretrain_lr=0.001
63      k=1
64      training_epochs=100
65      dataset='./dataset/gse10072.pkl.gz'
66      batch_size= 5
67      n_v=22283
68      n_output=2
69
70      # percobaan 1 dengan layer 10k 5k 1k 500
71      sys.stdout = Logger("./dataset/logout1000e_15k_8k_2k.txt")
72      hidden_sizes=[19000, 4000, 2000]
73      model_hasil = test_DBN(finetune_lr, pretraining_epochs,
74                  pretrain_lr, k, training_epochs,
75                  dataset, batch_size,hidden_sizes, n_v,n_output)
76
77      ekstraktor.simpan_data("./dataset/model1000e_18k_10k_2k_500.pkl.gz", model_hasil)
78      del model_hasil
79      gc.collect()
80      #
81      # # percobaan 2
82      # sys.stdout = Logger("./dataset/logout2000e_7k_10k_5k_1k.txt")
83      #
84      # hidden_sizes=[7000, 10000, 5000, 1000]
85      # model_hasil = test_DBN(finetune_lr, pretraining_epochs,
86      #             pretrain_lr, k, training_epochs,
87      #             dataset, batch_size,hidden_sizes, n_v,n_output)
88      #
89      # ekstraktor.simpan_data("./dataset/model2000e_7k_10k_5k_1k.pkl.gz", model_hasil)
90      # del model_hasil
91      # gc.collect()
92      #
93      # # percobaan 3
94      # sys.stdout = Logger("./dataset/logout2000e_3k_2k_1k_100.txt")
95      # hidden_sizes=[3000, 2000, 1000, 100]
96      # model_hasil = test_DBN(finetune_lr, pretraining_epochs,
97      #             pretrain_lr, k, training_epochs,
98      #             dataset, batch_size,hidden_sizes, n_v,n_output)
99      #
100     # ekstraktor.simpan_data("model2000e_3k_2k_1k_100.pkl.gz", model_hasil)
101     # del model_hasil
102     # gc.collect()
103
104
105
106 if __name__ == '__main__':
107     percobaan1_4l_2000e()
```

```
108     # percobaan2_3l_1000e()
```

Source code setelah ini diambil dari library Theano di www.deeplearning.net

Listing 9 : Implementasi Logistic Regression :

```
1   """
2   This tutorial introduces logistic regression using Theano and stochastic
3   gradient descent.
4
5   Logistic regression is a probabilistic, linear classifier. It is parametrized
6   by a weight matrix :math:`W` and a bias vector :math:`b`. Classification is
7   done by projecting data points onto a set of hyperplanes, the distance to
8   which is used to determine a class membership probability.
9
10  Mathematically, this can be written as:
11
12  .. math::
13    P(Y=i|x, W,b) &= softmax_i(W x + b) \\
14                  &= \frac {e^{W_i x + b_i}} {\sum_j e^{W_j x + b_j}}
15
16
17  The output of the model or prediction is then done by taking the argmax of
18  the vector whose i'th element is P(Y=i|x).
19
20  .. math::
21
22    y_{pred} = argmax_i P(Y=i|x,W,b)
23
24
25  This tutorial presents a stochastic gradient descent optimization method
26  suitable for large datasets.
27
28
29  References:
30
31      - textbooks: "Pattern Recognition and Machine Learning" -
32                   Christopher M. Bishop, section 4.3.2
33
34  """
35
36  from __future__ import print_function
37
38  __docformat__ = 'restructedtext en'
39
40  import six.moves.cPickle as pickle
41  import gzip
42  import os
43  import sys
44  import timeit
45
46  import numpy
47
48  import theano
49  import theano.tensor as T
50
51
52  class LogisticRegression(object):
53      """Multi-class Logistic Regression Class
54
55      The logistic regression is fully described by a weight matrix :math:`W`
56      and bias vector :math:`b`. Classification is done by projecting data
57      points onto a set of hyperplanes, the distance to which is used to
58      determine a class membership probability.
59      """
60
61      def __init__(self, input, n_in, n_out):
62          """ Initialize the parameters of the logistic regression
63
64          :type input: theano.tensor.TensorType
65          :param input: symbolic variable that describes the input of the
66                        architecture (one minibatch)
67
68          :type n_in: int
```

```
69          :param n_in: number of input units, the dimension of the space in
70                       which the datapoints lie
71
72          :type n_out: int
73          :param n_out: number of output units, the dimension of the space in
74                        which the labels lie
75
76          """
77          # start-snippet-1
78          # initialize with 0 the weights W as a matrix of shape (n_in, n_out)
79          self.W = theano.shared(
80              value=numpy.zeros(
81                  (n_in, n_out),
82                  dtype=theano.config.floatX
83              ),
84              name='W',
85              borrow=True
86          )
87          # initialize the biases b as a vector of n_out 0s
88          self.b = theano.shared(
89              value=numpy.zeros(
90                  (n_out,),
91                  dtype=theano.config.floatX
92              ),
93              name='b',
94              borrow=True
95          )
96
97          # symbolic expression for computing the matrix of class-membership
98          # probabilities
99          # Where:
100         # W is a matrix where column-k represent the separation hyperplane for
101         # class-k
102         # x is a matrix where row-j  represents input training sample-j
103         # b is a vector where element-k represent the free parameter of
104         # hyperplane-k
105         self.p_y_given_x = T.nnet.softmax(T.dot(input, self.W) + self.b)
106
107
108         # symbolic description of how to compute prediction as class whose
109         # probability is maximal
110         self.y_pred = T.argmax(self.p_y_given_x, axis=1)
111         # end-snippet-1
112
113         # parameters of the model
114         self.params = [self.W, self.b]
115
116         # keep track of model input
117         self.input = input
118
119     def negative_log_likelihood(self, y):
120         """Return the mean of the negative log-likelihood of the prediction
121         of this model under a given target distribution.
122
123         .. math::
124
125             \frac{1}{|\mathcal{D}|} \mathcal{L} (\theta=\{W,b\}, \mathcal{D}) =
126             \frac{1}{|\mathcal{D}|} \sum_{i=0}^{|\mathcal{D}|}
127                 \log(P(Y=y^{(i)}|x^{(i)}, W,b)) \\
128             \ell (\theta=\{W,b\}, \mathcal{D})
129
130         :type y: theano.tensor.TensorType
131         :param y: corresponds to a vector that gives for each example the
132                   correct label
133
134         Note: we use the mean instead of the sum so that
135               the learning rate is less dependent on the batch size
136         """
137         # start-snippet-2
138         # y.shape[0] is (symbolically) the number of rows in y, i.e.,
139         # number of examples (call it n) in the minibatch
140         # T.arange(y.shape[0]) is a symbolic vector which will contain
141         # [0,1,2,... n-1] T.log(self.p_y_given_x) is a matrix of
142         # Log-Probabilities (call it LP) with one row per example and
143         # one column per class LP[T.arange(y.shape[0]),y] is a vector
144         # v containing [LP[0,y[0]], LP[1,y[1]], LP[2,y[2]], ...,
145         # LP[n-1,y[n-1]]] and T.mean(LP[T.arange(y.shape[0]),y]) is
```

```
146         # the mean (across minibatch examples) of the elements in v,
147         # i.e., the mean log-likelihood across the minibatch.
148         return -T.mean(T.log(self.p_y_given_x)[T.arange(y.shape[0]), y])
149         # end-snippet-2
150
151     def errors(self, y):
152         """Return a float representing the number of errors in the minibatch
153         over the total number of examples of the minibatch ; zero one
154         loss over the size of the minibatch
155
156         :type y: theano.tensor.TensorType
157         :param y: corresponds to a vector that gives for each example the
158                   correct label
159         """
160
161         # check if y has same dimension of y_pred
162         if y.ndim != self.y_pred.ndim:
163             raise TypeError(
164                 'y_should_have_the_same_shape_as_self.y_pred',
165                 ('y', y.type, 'y_pred', self.y_pred.type)
166             )
167         # check if y is of the correct datatype
168         if y.dtype.startswith('int'):
169             # the T.neq operator returns a vector of 0s and 1s, where 1
170             # represents a mistake in prediction
171             return T.mean(T.neq(self.y_pred, y))
172         else:
173             raise NotImplementedError()
174
175
176 def load_data(dataset):
177     ''' Loads the dataset
178
179     :type dataset: string
180     :param dataset: the path to the dataset (here MNIST)
181     '''
182
183     #############
184     # LOAD DATA #
185     #############
186
187     # Download the MNIST dataset if it is not present
188     data_dir, data_file = os.path.split(dataset)
189     if data_dir == "" and not os.path.isfile(dataset):
190         # Check if dataset is in the data directory.
191         new_path = os.path.join(
192             os.path.split(__file__)[0],
193             "..",
194             "data",
195             dataset
196         )
197         if os.path.isfile(new_path) or data_file == 'mnist.pkl.gz':
198             dataset = new_path
199
200     if (not os.path.isfile(dataset)) and data_file == 'mnist.pkl.gz':
201         from six.moves import urllib
202         origin = (
203             'http://www.iro.umontreal.ca/~lisa/deep/data/mnist/mnist.pkl.gz'
204         )
205         print('Downloading_data_from_%s' % origin)
206         urllib.request.urlretrieve(origin, dataset)
207
208     print('..._loading_data')
209
210     # Load the dataset
211     with gzip.open(dataset, 'rb') as f:
212         try:
213             train_set, valid_set, test_set = pickle.load(f, encoding='latin1')
214         except:
215             train_set, valid_set, test_set = pickle.load(f)
216     # train_set, valid_set, test_set format: tuple(input, target)
217     # input is a numpy.ndarray of 2 dimensions (a matrix)
218     # where each row corresponds to an example. target is a
219     # numpy.ndarray of 1 dimension (vector) that has the same length as
220     # the number of rows in the input. It should give the target
221     # to the example with the same index in the input.
222
```

```
223     def shared_dataset(data_xy, borrow=True):
224         """ Function that loads the dataset into shared variables
225 
226         The reason we store our dataset in shared variables is to allow
227         Theano to copy it into the GPU memory (when code is run on GPU).
228         Since copying data into the GPU is slow, copying a minibatch everytime
229         is needed (the default behaviour if the data is not in a shared
230         variable) would lead to a large decrease in performance.
231         """
232         data_x, data_y = data_xy
233         shared_x = theano.shared(numpy.asarray(data_x,
234                                                dtype=theano.config.floatX),
235                                  borrow=borrow)
236         shared_y = theano.shared(numpy.asarray(data_y,
237                                                dtype=theano.config.floatX),
238                                  borrow=borrow)
239         # When storing data on the GPU it has to be stored as floats
240         # therefore we will store the labels as ``floatX'' as well
241         # (``shared_y'' does exactly that). But during our computations
242         # we need them as ints (we use labels as index, and if they are
243         # floats it doesn't make sense) therefore instead of returning
244         # ``shared_y'' we will have to cast it to int. This little hack
245         # lets ous get around this issue
246         return shared_x, T.cast(shared_y, 'int32')
247 
248     test_set_x, test_set_y = shared_dataset(test_set)
249     valid_set_x, valid_set_y = shared_dataset(valid_set)
250     train_set_x, train_set_y = shared_dataset(train_set)
251 
252     rval = [(train_set_x, train_set_y), (valid_set_x, valid_set_y),
253             (test_set_x, test_set_y)]
254     return rval
255 
256 
257 def sgd_optimization_mnist(learning_rate=0.13, n_epochs=1000,
258                            dataset='mnist.pkl.gz',
259                            batch_size=600):
260     """
261     Demonstrate stochastic gradient descent optimization of a log-linear
262     model
263 
264     This is demonstrated on MNIST.
265 
266     :type learning_rate: float
267     :param learning_rate: learning rate used (factor for the stochastic
268                           gradient)
269 
270     :type n_epochs: int
271     :param n_epochs: maximal number of epochs to run the optimizer
272 
273     :type dataset: string
274     :param dataset: the path of the MNIST dataset file from
275                  http://www.iro.umontreal.ca/~lisa/deep/data/mnist/mnist.pkl.gz
276 
277     """
278     datasets = load_data(dataset)
279 
280     train_set_x, train_set_y = datasets[0]
281     valid_set_x, valid_set_y = datasets[1]
282     test_set_x, test_set_y = datasets[2]
283 
284     # compute number of minibatches for training, validation and testing
285     n_train_batches = train_set_x.get_value(borrow=True).shape[0] // batch_size
286     n_valid_batches = valid_set_x.get_value(borrow=True).shape[0] // batch_size
287     n_test_batches = test_set_x.get_value(borrow=True).shape[0] // batch_size
288 
289     ######################
290     # BUILD ACTUAL MODEL #
291     ######################
292     print('... building the model')
293 
294     # allocate symbolic variables for the data
295     index = T.lscalar()  # index to a [mini]batch
296 
297     # generate symbolic variables for input (x and y represent a
298     # minibatch)
299     x = T.matrix('x')  # data, presented as rasterized images
```

```
    y = T.ivector('y')  # labels, presented as 1D vector of [int] labels

    # construct the logistic regression class
    # Each MNIST image has size 28*28
    classifier = LogisticRegression(input=x, n_in=28 * 28, n_out=10)

    # the cost we minimize during training is the negative log likelihood of
    # the model in symbolic format
    cost = classifier.negative_log_likelihood(y)

    # compiling a Theano function that computes the mistakes that are made by
    # the model on a minibatch
    test_model = theano.function(
        inputs=[index],
        outputs=classifier.errors(y),
        givens={
            x: test_set_x[index * batch_size: (index + 1) * batch_size],
            y: test_set_y[index * batch_size: (index + 1) * batch_size]
        }
    )

    validate_model = theano.function(
        inputs=[index],
        outputs=classifier.errors(y),
        givens={
            x: valid_set_x[index * batch_size: (index + 1) * batch_size],
            y: valid_set_y[index * batch_size: (index + 1) * batch_size]
        }
    )

    # compute the gradient of cost with respect to theta = (W,b)
    g_W = T.grad(cost=cost, wrt=classifier.W)
    g_b = T.grad(cost=cost, wrt=classifier.b)

    # start-snippet-3
    # specify how to update the parameters of the model as a list of
    # (variable, update expression) pairs.
    updates = [(classifier.W, classifier.W - learning_rate * g_W),
               (classifier.b, classifier.b - learning_rate * g_b)]

    # compiling a Theano function `train_model` that returns the cost, but in
    # the same time updates the parameter of the model based on the rules
    # defined in `updates`
    train_model = theano.function(
        inputs=[index],
        outputs=cost,
        updates=updates,
        givens={
            x: train_set_x[index * batch_size: (index + 1) * batch_size],
            y: train_set_y[index * batch_size: (index + 1) * batch_size]
        }
    )
    # end-snippet-3

    ###############
    # TRAIN MODEL #
    ###############
    print('... training the model')
    # early-stopping parameters
    patience = 5000  # look as this many examples regardless
    patience_increase = 2  # wait this much longer when a new best is
                           # found
    improvement_threshold = 0.995  # a relative improvement of this much is
                                   # considered significant
    validation_frequency = min(n_train_batches, patience // 2)
                                   # go through this many
                                   # minibatche before checking the network
                                   # on the validation set; in this case we
                                   # check every epoch

    best_validation_loss = numpy.inf
    test_score = 0.
    start_time = timeit.default_timer()

    done_looping = False
    epoch = 0
    while (epoch < n_epochs) and (not done_looping):
```

```
        epoch = epoch + 1
        for minibatch_index in range(n_train_batches):

            minibatch_avg_cost = train_model(minibatch_index)
            # iteration number
            iter = (epoch - 1) * n_train_batches + minibatch_index

            if (iter + 1) % validation_frequency == 0:
                # compute zero-one loss on validation set
                validation_losses = [validate_model(i)
                                     for i in range(n_valid_batches)]
                this_validation_loss = numpy.mean(validation_losses)

                print(
                    'epoch %i, minibatch %i/%i, validation error %f %%' %
                    (
                        epoch,
                        minibatch_index + 1,
                        n_train_batches,
                        this_validation_loss * 100.
                    )
                )

                # if we got the best validation score until now
                if this_validation_loss < best_validation_loss:
                    #improve patience if loss improvement is good enough
                    if this_validation_loss < best_validation_loss *  \
                       improvement_threshold:
                        patience = max(patience, iter * patience_increase)

                    best_validation_loss = this_validation_loss
                    # test it on the test set

                    test_losses = [test_model(i)
                                   for i in range(n_test_batches)]
                    test_score = numpy.mean(test_losses)

                    print(
                        (
                            '     epoch %i, minibatch %i/%i, test error of'
                            ' best model %f %%'
                        ) %
                        (
                            epoch,
                            minibatch_index + 1,
                            n_train_batches,
                            test_score * 100.
                        )
                    )

                    # save the best model
                    with open('./dataset/best_model.pkl', 'wb') as f:
                        pickle.dump(classifier, f)

            if patience <= iter:
                done_looping = True
                break

    end_time = timeit.default_timer()
    print(
        (
            'Optimization complete with best validation score of %f %%,'
            'with test performance %f %%'
        )
        % (best_validation_loss * 100., test_score * 100.)
    )
    print('The code run for %d epochs, with %f epochs/sec' % (
        epoch, 1. * epoch / (end_time - start_time)))
    print(('The code for file ' +
           os.path.split(__file__)[1] +
           ' ran for %.1fs' % ((end_time - start_time))), file=sys.stderr)


def predict():
    """
    An example of how to load a trained model and use it
    to predict labels.
```

```
    """

    # load the saved model
    classifier = pickle.load(open('./dataset/best_model.pkl'))

    # compile a predictor function
    predict_model = theano.function(
        inputs=[classifier.input],
        outputs=classifier.y_pred)

    # We can test it on some examples from test test
    dataset='mnist.pkl.gz'
    datasets = load_data(dataset)
    test_set_x, test_set_y = datasets[2]
    test_set_x = test_set_x.get_value()

    predicted_values = predict_model(test_set_x[:10])
    print("Predicted values for the first 10 examples in test set:")
    print(predicted_values)


if __name__ == '__main__':
    sgd_optimization_mnist()
```

Listing 10 : Implementasi Restricted Boltzmann Machine :

```
"""This tutorial introduces restricted boltzmann machines (RBM) using Theano.

Boltzmann Machines (BMs) are a particular form of energy-based model which
contain hidden variables. Restricted Boltzmann Machines further restrict BMs
to those without visible-visible and hidden-hidden connections.
"""

from __future__ import print_function

import timeit

try:
    import PIL.Image as Image
except ImportError:
    import Image

import numpy

import theano
import theano.tensor as T
import os

from theano.tensor.shared_randomstreams import RandomStreams

from utils import tile_raster_images
from logistic_sgd import load_data


# start-snippet-1
class RBM(object):
    """Restricted Boltzmann Machine (RBM)  """
    def __init__(
        self,
        input=None,
        n_visible=784,
        n_hidden=500,
        W=None,
        hbias=None,
        vbias=None,
        numpy_rng=None,
        theano_rng=None
    ):
        """
        RBM constructor. Defines the parameters of the model along with
        basic operations for inferring hidden from visible (and vice-versa),
        as well as for performing CD updates.

        :param input: None for standalone RBMs or symbolic variable if RBM is
        part of a larger graph.

```

```
51          :param n_visible: number of visible units
52
53          :param n_hidden: number of hidden units
54
55          :param W: None for standalone RBMs or symbolic variable pointing to a
56          shared weight matrix in case RBM is part of a DBN network; in a DBN,
57          the weights are shared between RBMs and layers of a MLP
58
59          :param hbias: None for standalone RBMs or symbolic variable pointing
60          to a shared hidden units bias vector in case RBM is part of a
61          different network
62
63          :param vbias: None for standalone RBMs or a symbolic variable
64          pointing to a shared visible units bias
65          """
66
67          self.n_visible = n_visible
68          self.n_hidden = n_hidden
69
70          if numpy_rng is None:
71              # create a number generator
72              numpy_rng = numpy.random.RandomState(1234)
73
74          if theano_rng is None:
75              theano_rng = RandomStreams(numpy_rng.randint(2 ** 30))
76
77          if W is None:
78              # W is initialized with `initial_W` which is uniformely
79              # sampled from -4*sqrt(6./(n_visible+n_hidden)) and
80              # 4*sqrt(6./(n_hidden+n_visible)) the output of uniform if
81              # converted using asarray to dtype theano.config.floatX so
82              # that the code is runable on GPU
83              initial_W = numpy.asarray(
84                  numpy_rng.uniform(
85                      low=-4 * numpy.sqrt(6. / (n_hidden + n_visible)),
86                      high=4 * numpy.sqrt(6. / (n_hidden + n_visible)),
87                      size=(n_visible, n_hidden)
88                  ),
89                  dtype=theano.config.floatX
90              )
91              # theano shared variables for weights and biases
92              W = theano.shared(value=initial_W, name='W', borrow=True)
93
94          if hbias is None:
95              # create shared variable for hidden units bias
96              hbias = theano.shared(
97                  value=numpy.zeros(
98                      n_hidden,
99                      dtype=theano.config.floatX
100                 ),
101                 name='hbias',
102                 borrow=True
103             )
104
105         if vbias is None:
106             # create shared variable for visible units bias
107             vbias = theano.shared(
108                 value=numpy.zeros(
109                     n_visible,
110                     dtype=theano.config.floatX
111                 ),
112                 name='vbias',
113                 borrow=True
114             )
115
116         # initialize input layer for standalone RBM or layer0 of DBN
117         self.input = input
118         if not input:
119             self.input = T.matrix('input')
120
121         self.W = W
122         self.hbias = hbias
123         self.vbias = vbias
124         self.theano_rng = theano_rng
125         # **** WARNING: It is not a good idea to put things in this list
126         # other than shared variables created in this function.
127         self.params = [self.W, self.hbias, self.vbias]
```

```
128         # end-snippet-1
129
130     def free_energy(self, v_sample):
131         ''' Function to compute the free energy '''
132         wx_b = T.dot(v_sample, self.W) + self.hbias
133         vbias_term = T.dot(v_sample, self.vbias)
134         hidden_term = T.sum(T.log(1 + T.exp(wx_b)), axis=1)
135         return -hidden_term - vbias_term
136
137     def propup(self, vis):
138         '''This function propagates the visible units activation upwards to
139         the hidden units
140
141         Note that we return also the pre-sigmoid activation of the
142         layer. As it will turn out later, due to how Theano deals with
143         optimizations, this symbolic variable will be needed to write
144         down a more stable computational graph (see details in the
145         reconstruction cost function)
146
147         '''
148         pre_sigmoid_activation = T.dot(vis, self.W) + self.hbias
149         return [pre_sigmoid_activation, T.nnet.sigmoid(pre_sigmoid_activation)]
150
151     def sample_h_given_v(self, v0_sample):
152         ''' This function infers state of hidden units given visible units '''
153         # compute the activation of the hidden units given a sample of
154         # the visibles
155         pre_sigmoid_h1, h1_mean = self.propup(v0_sample)
156         # get a sample of the hiddens given their activation
157         # Note that theano_rng.binomial returns a symbolic sample of dtype
158         # int64 by default. If we want to keep our computations in floatX
159         # for the GPU we need to specify to return the dtype floatX
160         h1_sample = self.theano_rng.binomial(size=h1_mean.shape,
161                                              n=1, p=h1_mean,
162                                              dtype=theano.config.floatX)
163         return [pre_sigmoid_h1, h1_mean, h1_sample]
164
165     def propdown(self, hid):
166         '''This function propagates the hidden units activation downwards to
167         the visible units
168
169         Note that we return also the pre_sigmoid_activation of the
170         layer. As it will turn out later, due to how Theano deals with
171         optimizations, this symbolic variable will be needed to write
172         down a more stable computational graph (see details in the
173         reconstruction cost function)
174
175         '''
176         pre_sigmoid_activation = T.dot(hid, self.W.T) + self.vbias
177         return [pre_sigmoid_activation, T.nnet.sigmoid(pre_sigmoid_activation)]
178
179     def sample_v_given_h(self, h0_sample):
180         ''' This function infers state of visible units given hidden units '''
181         # compute the activation of the visible given the hidden sample
182         pre_sigmoid_v1, v1_mean = self.propdown(h0_sample)
183         # get a sample of the visible given their activation
184         # Note that theano_rng.binomial returns a symbolic sample of dtype
185         # int64 by default. If we want to keep our computations in floatX
186         # for the GPU we need to specify to return the dtype floatX
187         v1_sample = self.theano_rng.binomial(size=v1_mean.shape,
188                                              n=1, p=v1_mean,
189                                              dtype=theano.config.floatX)
190         return [pre_sigmoid_v1, v1_mean, v1_sample]
191
192     def gibbs_hvh(self, h0_sample):
193         ''' This function implements one step of Gibbs sampling,
194             starting from the hidden state'''
195         pre_sigmoid_v1, v1_mean, v1_sample = self.sample_v_given_h(h0_sample)
196         pre_sigmoid_h1, h1_mean, h1_sample = self.sample_h_given_v(v1_sample)
197         return [pre_sigmoid_v1, v1_mean, v1_sample,
198                 pre_sigmoid_h1, h1_mean, h1_sample]
199
200     def gibbs_vhv(self, v0_sample):
201         ''' This function implements one step of Gibbs sampling,
202             starting from the visible state'''
203         pre_sigmoid_h1, h1_mean, h1_sample = self.sample_h_given_v(v0_sample)
204         pre_sigmoid_v1, v1_mean, v1_sample = self.sample_v_given_h(h1_sample)
```

```
205         return [pre_sigmoid_h1, h1_mean, h1_sample,
206                 pre_sigmoid_v1, v1_mean, v1_sample]
207
208     # start-snippet-2
209     def get_cost_updates(self, lr=0.1, persistent=None, k=1):
210         """This functions implements one step of CD-k or PCD-k
211
212         :param lr: learning rate used to train the RBM
213
214         :param persistent: None for CD. For PCD, shared variable
215             containing old state of Gibbs chain. This must be a shared
216             variable of size (batch size, number of hidden units).
217
218         :param k: number of Gibbs steps to do in CD-k/PCD-k
219
220         Returns a proxy for the cost and the updates dictionary. The
221         dictionary contains the update rules for weights and biases but
222         also an update of the shared variable used to store the persistent
223         chain, if one is used.
224
225         """
226
227         # compute positive phase
228         pre_sigmoid_ph, ph_mean, ph_sample = self.sample_h_given_v(self.input)
229
230         # decide how to initialize persistent chain:
231         # for CD, we use the newly generate hidden sample
232         # for PCD, we initialize from the old state of the chain
233         if persistent is None:
234             chain_start = ph_sample
235         else:
236             chain_start = persistent
237         # end-snippet-2
238         # perform actual negative phase
239         # in order to implement CD-k/PCD-k we need to scan over the
240         # function that implements one gibbs step k times.
241         # Read Theano tutorial on scan for more information :
242         # http://deeplearning.net/software/theano/library/scan.html
243         # the scan will return the entire Gibbs chain
244         (
245             [
246                 pre_sigmoid_nvs,
247                 nv_means,
248                 nv_samples,
249                 pre_sigmoid_nhs,
250                 nh_means,
251                 nh_samples
252             ],
253             updates
254         ) = theano.scan(
255             self.gibbs_hvh,
256             # the None are place holders, saying that
257             # chain_start is the initial state corresponding to the
258             # 6th output
259             outputs_info=[None, None, None, None, None, chain_start],
260             n_steps=k
261         )
262         # start-snippet-3
263         # determine gradients on RBM parameters
264         # note that we only need the sample at the end of the chain
265         chain_end = nv_samples[-1]
266
267         cost = T.mean(self.free_energy(self.input)) - T.mean(
268             self.free_energy(chain_end))
269         # We must not compute the gradient through the gibbs sampling
270         gparams = T.grad(cost, self.params, consider_constant=[chain_end])
271         # end-snippet-3 start-snippet-4
272         # constructs the update dictionary
273         for gparam, param in zip(gparams, self.params):
274             # make sure that the learning rate is of the right dtype
275             updates[param] = param - gparam * T.cast(
276                 lr,
277                 dtype=theano.config.floatX
278             )
279         if persistent:
280             # Note that this works only if persistent is a shared variable
281             updates[persistent] = nh_samples[-1]
```

```
282             # pseudo-likelihood is a better proxy for PCD
283             monitoring_cost = self.get_pseudo_likelihood_cost(updates)
284         else:
285             # reconstruction cross-entropy is a better proxy for CD
286             monitoring_cost = self.get_reconstruction_cost(updates,
287                                                            pre_sigmoid_nvs[-1])
288 
289         return monitoring_cost, updates
290         # end-snippet-4
291 
292     def get_pseudo_likelihood_cost(self, updates):
293         """Stochastic approximation to the pseudo-likelihood"""
294 
295         # index of bit i in expression p(x_i | x_{\i})
296         bit_i_idx = theano.shared(value=0, name='bit_i_idx')
297 
298         # binarize the input image by rounding to nearest integer
299         xi = T.round(self.input)
300 
301         # calculate free energy for the given bit configuration
302         fe_xi = self.free_energy(xi)
303 
304         # flip bit x_i of matrix xi and preserve all other bits x_{\i}
305         # Equivalent to xi[:, bit_i_idx] = 1-xi[:, bit_i_idx], but assigns
306         # the result to xi_flip, instead of working in place on xi.
307         xi_flip = T.set_subtensor(xi[:, bit_i_idx], 1 - xi[:, bit_i_idx])
308 
309         # calculate free energy with bit flipped
310         fe_xi_flip = self.free_energy(xi_flip)
311 
312         # equivalent to e^(-FE(x_i)) / (e^(-FE(x_i)) + e^(-FE(x_{\i})))
313         cost = T.mean(self.n_visible * T.log(T.nnet.sigmoid(fe_xi_flip -
314                                                             fe_xi)))
315 
316         # increment bit_i_idx % number as part of updates
317         updates[bit_i_idx] = (bit_i_idx + 1) % self.n_visible
318 
319         return cost
320 
321     def get_reconstruction_cost(self, updates, pre_sigmoid_nv):
322         """Approximation to the reconstruction error
323 
324         Note that this function requires the pre-sigmoid activation as
325         input.  To understand why this is so you need to understand a
326         bit about how Theano works. Whenever you compile a Theano
327         function, the computational graph that you pass as input gets
328         optimized for speed and stability.  This is done by changing
329         several parts of the subgraphs with others.  One such
330         optimization expresses terms of the form log(sigmoid(x)) in
331         terms of softplus.  We need this optimization for the
332         cross-entropy since sigmoid of numbers larger than 30. (or
333         even less then that) turn to 1. and numbers smaller than
334         -30. turn to 0 which in terms will force theano to compute
335         log(0) and therefore we will get either -inf or NaN as
336         cost. If the value is expressed in terms of softplus we do not
337         get this undesirable behaviour. This optimization usually
338         works fine, but here we have a special case. The sigmoid is
339         applied inside the scan op, while the log is
340         outside. Therefore Theano will only see log(scan(..)) instead
341         of log(sigmoid(..)) and will not apply the wanted
342         optimization. We can not go and replace the sigmoid in scan
343         with something else also, because this only needs to be done
344         on the last step. Therefore the easiest and more efficient way
345         is to get also the pre-sigmoid activation as an output of
346         scan, and apply both the log and sigmoid outside scan such
347         that Theano can catch and optimize the expression.
348 
349         """
350 
351         cross_entropy = T.mean(
352             T.sum(
353                 self.input * T.log(T.nnet.sigmoid(pre_sigmoid_nv)) +
354                 (1 - self.input) * T.log(1 - T.nnet.sigmoid(pre_sigmoid_nv)),
355                 axis=1
356             )
357         )
358 
```

```
359         return cross_entropy
360
361
362 def test_rbm(learning_rate=0.1, training_epochs=15,
363               dataset='mnist.pkl.gz', batch_size=20,
364               n_chains=20, n_samples=10, output_folder='rbm_plots',
365               n_hidden=500):
366     """
367     Demonstrate how to train and afterwards sample from it using Theano.
368
369     This is demonstrated on MNIST.
370
371     :param learning_rate: learning rate used for training the RBM
372
373     :param training_epochs: number of epochs used for training
374
375     :param dataset: path the the pickled dataset
376
377     :param batch_size: size of a batch used to train the RBM
378
379     :param n_chains: number of parallel Gibbs chains to be used for sampling
380
381     :param n_samples: number of samples to plot for each chain
382
383     """
384     datasets = load_data(dataset)
385
386     train_set_x, train_set_y = datasets[0]
387     test_set_x, test_set_y = datasets[2]
388
389     # compute number of minibatches for training, validation and testing
390     n_train_batches = train_set_x.get_value(borrow=True).shape[0] // batch_size
391
392     # allocate symbolic variables for the data
393     index = T.lscalar()    # index to a [mini]batch
394     x = T.matrix('x')  # the data is presented as rasterized images
395
396     rng = numpy.random.RandomState(123)
397     theano_rng = RandomStreams(rng.randint(2 ** 30))
398
399     # initialize storage for the persistent chain (state = hidden
400     # layer of chain)
401     persistent_chain = theano.shared(numpy.zeros((batch_size, n_hidden),
402                                                  dtype=theano.config.floatX),
403                                      borrow=True)
404
405     # construct the RBM class
406     rbm = RBM(input=x, n_visible=28 * 28,
407               n_hidden=n_hidden, numpy_rng=rng, theano_rng=theano_rng)
408
409     # get the cost and the gradient corresponding to one step of CD-15
410     cost, updates = rbm.get_cost_updates(lr=learning_rate,
411                                          persistent=persistent_chain, k=15)
412
413     #################################
414     #     Training the RBM          #
415     #################################
416     if not os.path.isdir(output_folder):
417         os.makedirs(output_folder)
418     os.chdir(output_folder)
419
420     # start-snippet-5
421     # it is ok for a theano function to have no output
422     # the purpose of train_rbm is solely to update the RBM parameters
423     train_rbm = theano.function(
424         [index],
425         cost,
426         updates=updates,
427         givens={
428             x: train_set_x[index * batch_size: (index + 1) * batch_size]
429         },
430         name='train_rbm'
431     )
432
433     plotting_time = 0.
434     start_time = timeit.default_timer()
435
```

```
436     # go through training epochs
437     for epoch in range(training_epochs):
438
439         # go through the training set
440         mean_cost = []
441         for batch_index in range(n_train_batches):
442             mean_cost += [train_rbm(batch_index)]
443
444         print('Training epoch %d, cost is ' % epoch, numpy.mean(mean_cost))
445
446         # Plot filters after each training epoch
447         plotting_start = timeit.default_timer()
448         # Construct image from the weight matrix
449         image = Image.fromarray(
450             tile_raster_images(
451                 X=rbm.W.get_value(borrow=True).T,
452                 img_shape=(28, 28),
453                 tile_shape=(10, 10),
454                 tile_spacing=(1, 1)
455             )
456         )
457         image.save('filters_at_epoch_%i.png' % epoch)
458         plotting_stop = timeit.default_timer()
459         plotting_time += (plotting_stop - plotting_start)
460
461     end_time = timeit.default_timer()
462
463     pretraining_time = (end_time - start_time) - plotting_time
464
465     print ('Training took %f minutes' % (pretraining_time / 60.))
466     # end-snippet-5 start-snippet-6
467     ###################################
468     #      Sampling from the RBM      #
469     ###################################
470     # find out the number of test samples
471     number_of_test_samples = test_set_x.get_value(borrow=True).shape[0]
472
473     # pick random test examples, with which to initialize the persistent chain
474     test_idx = rng.randint(number_of_test_samples - n_chains)
475     persistent_vis_chain = theano.shared(
476         numpy.asarray(
477             test_set_x.get_value(borrow=True)[test_idx:test_idx + n_chains],
478             dtype=theano.config.floatX
479         )
480     )
481     # end-snippet-6 start-snippet-7
482     plot_every = 1000
483     # define one step of Gibbs sampling (mf = mean-field) define a
484     # function that does `plot_every` steps before returning the
485     # sample for plotting
486     (
487         [
488             presig_hids,
489             hid_mfs,
490             hid_samples,
491             presig_vis,
492             vis_mfs,
493             vis_samples
494         ],
495         updates
496     ) = theano.scan(
497         rbm.gibbs_vhv,
498         outputs_info=[None, None, None, None, None, persistent_vis_chain],
499         n_steps=plot_every
500     )
501
502     # add to updates the shared variable that takes care of our persistent
503     # chain :.
504     updates.update({persistent_vis_chain: vis_samples[-1]})
505     # construct the function that implements our persistent chain.
506     # we generate the "mean field" activations for plotting and the actual
507     # samples for reinitializing the state of our persistent chain
508     sample_fn = theano.function(
509         [],
510         [
511             vis_mfs[-1],
512             vis_samples[-1]
```

```
513             ],
514             updates=updates,
515             name='sample_fn'
516         )
517
518         # create a space to store the image for plotting ( we need to leave
519         # room for the tile_spacing as well)
520         image_data = numpy.zeros(
521             (29 * n_samples + 1, 29 * n_chains - 1),
522             dtype='uint8'
523         )
524         for idx in range(n_samples):
525             # generate `plot_every` intermediate samples that we discard,
526             # because successive samples in the chain are too correlated
527             vis_mf, vis_sample = sample_fn()
528             print(' ... plotting sample %d' % idx)
529             image_data[29 * idx:29 * idx + 28, :] = tile_raster_images(
530                 X=vis_mf,
531                 img_shape=(28, 28),
532                 tile_shape=(1, n_chains),
533                 tile_spacing=(1, 1)
534             )
535
536         # construct image
537         image = Image.fromarray(image_data)
538         image.save('samples.png')
539         # end-snippet-7
540         os.chdir('../')
541
542 if __name__ == '__main__':
543     test_rbm()
```

Listing 11 : Implementasi Restricted Boltzmann Machine :

```
1  """
2  """
3  import os
4  import sys
5  import timeit
6
7  import numpy
8
9  import theano
10 import theano.tensor as T
11 from theano.sandbox.rng_mrg import MRG_RandomStreams
12
13 from logistic_sgd import LogisticRegression, load_data
14 from mlp import HiddenLayer
15 from rbm import RBM
16
17
18 # start-snippet-1
19 class DBN(object):
20     """Deep Belief Network
21
22     A deep belief network is obtained by stacking several RBMs on top of each
23     other. The hidden layer of the RBM at layer `i` becomes the input of the
24     RBM at layer `i+1`. The first layer RBM gets as input the input of the
25     network, and the hidden layer of the last RBM represents the output. When
26     used for classification, the DBN is treated as a MLP, by adding a logistic
27     regression layer on top.
28     """
29
30     def __init__(self, numpy_rng, theano_rng=None, n_ins=784,
31                  hidden_layers_sizes=[500, 500], n_outs=10):
32         """This class is made to support a variable number of layers.
33
34         :type numpy_rng: numpy.random.RandomState
35         :param numpy_rng: numpy random number generator used to draw initial
36                           weights
37
38         :type theano_rng: theano.tensor.shared_randomstreams.RandomStreams
39         :param theano_rng: Theano random generator; if None is given one is
40                            generated based on a seed drawn from `rng`
41
42         :type n_ins: int
```

```
43          :param n_ins: dimension of the input to the DBN
44
45          :type hidden_layers_sizes: list of ints
46          :param hidden_layers_sizes: intermediate layers size, must contain
47                                      at least one value
48
49          :type n_outs: int
50          :param n_outs: dimension of the output of the network
51          """
52
53          self.sigmoid_layers = []
54          self.rbm_layers = []
55          self.params = []
56          self.n_layers = len(hidden_layers_sizes)
57
58          assert self.n_layers > 0
59
60          if not theano_rng:
61              theano_rng = MRG_RandomStreams(numpy_rng.randint(2 ** 30))
62
63          # allocate symbolic variables for the data
64          self.x = T.matrix('x')  # the data is presented as rasterized images
65          self.y = T.ivector('y')  # the labels are presented as 1D vector
66                                   # of [int] labels
67          # end-snippet-1
68          # The DBN is an MLP, for which all weights of intermediate
69          # layers are shared with a different RBM.  We will first
70          # construct the DBN as a deep multilayer perceptron, and when
71          # constructing each sigmoidal layer we also construct an RBM
72          # that shares weights with that layer. During pretraining we
73          # will train these RBMs (which will lead to chainging the
74          # weights of the MLP as well) During finetuning we will finish
75          # training the DBN by doing stochastic gradient descent on the
76          # MLP.
77
78          for i in range(self.n_layers):
79              # construct the sigmoidal layer
80
81              # the size of the input is either the number of hidden
82              # units of the layer below or the input size if we are on
83              # the first layer
84              if i == 0:
85                  input_size = n_ins
86              else:
87                  input_size = hidden_layers_sizes[i - 1]
88
89              # the input to this layer is either the activation of the
90              # hidden layer below or the input of the DBN if you are on
91              # the first layer
92              if i == 0:
93                  layer_input = self.x
94              else:
95                  layer_input = self.sigmoid_layers[-1].output
96
97              sigmoid_layer = HiddenLayer(rng=numpy_rng,
98                                          input=layer_input,
99                                          n_in=input_size,
100                                         n_out=hidden_layers_sizes[i],
101                                         activation=T.nnet.sigmoid)
102
103             # add the layer to our list of layers
104             self.sigmoid_layers.append(sigmoid_layer)
105
106             # its arguably a philosophical question...  but we are
107             # going to only declare that the parameters of the
108             # sigmoid_layers are parameters of the DBN. The visible
109             # biases in the RBM are parameters of those RBMs, but not
110             # of the DBN.
111             self.params.extend(sigmoid_layer.params)
112
113             # Construct an RBM that shared weights with this layer
114             rbm_layer = RBM(numpy_rng=numpy_rng,
115                             theano_rng=theano_rng,
116                             input=layer_input,
117                             n_visible=input_size,
118                             n_hidden=hidden_layers_sizes[i],
119                             W=sigmoid_layer.W,
```

```
120                             hbias=sigmoid_layer.b)
121             self.rbm_layers.append(rbm_layer)
122
123         # We now need to add a logistic layer on top of the MLP
124         self.logLayer = LogisticRegression(
125             input=self.sigmoid_layers[-1].output,
126             n_in=hidden_layers_sizes[-1],
127             n_out=n_outs)
128         self.params.extend(self.logLayer.params)
129
130         # compute the cost for second phase of training, defined as the
131         # negative log likelihood of the logistic regression (output) layer
132         self.finetune_cost = self.logLayer.negative_log_likelihood(self.y)
133
134         # compute the gradients with respect to the model parameters
135         # symbolic variable that points to the number of errors made on the
136         # minibatch given by self.x and self.y
137         self.errors = self.logLayer.errors(self.y)
138
139     def pretraining_functions(self, train_set_x, batch_size, k):
140         '''Generates a list of functions, for performing one step of
141         gradient descent at a given layer. The function will require
142         as input the minibatch index, and to train an RBM you just
143         need to iterate, calling the corresponding function on all
144         minibatch indexes.
145
146         :type train_set_x: theano.tensor.TensorType
147         :param train_set_x: Shared var. that contains all datapoints used
148                             for training the RBM
149         :type batch_size: int
150         :param batch_size: size of a [mini]batch
151         :param k: number of Gibbs steps to do in CD-k / PCD-k
152
153         '''
154
155         # index to a [mini]batch
156         index = T.lscalar('index')  # index to a minibatch
157         learning_rate = T.scalar('lr')  # learning rate to use
158
159         # number of batches
160         n_batches = train_set_x.get_value(borrow=True).shape[0] / batch_size
161         # begining of a batch, given `index`
162         batch_begin = index * batch_size
163         # ending of a batch given `index`
164         batch_end = batch_begin + batch_size
165
166         pretrain_fns = []
167         for rbm in self.rbm_layers:
168
169             # get the cost and the updates list
170             # using CD-k here (persisent=None) for training each RBM.
171             # TODO: change cost function to reconstruction error
172             cost, updates = rbm.get_cost_updates(learning_rate,
173                                                  persistent=None, k=k)
174
175             # compile the theano function
176             fn = theano.function(
177                 inputs=[index, theano.In(learning_rate, value=0.1)],
178                 outputs=cost,
179                 updates=updates,
180                 givens={
181                     self.x: train_set_x[batch_begin:batch_end]
182                 }
183             )
184             # append `fn` to the list of functions
185             pretrain_fns.append(fn)
186
187         return pretrain_fns
188
189     def build_finetune_functions(self, datasets, batch_size, learning_rate):
190         '''Generates a function `train` that implements one step of
191         finetuning, a function `validate` that computes the error on a
192         batch from the validation set, and a function `test` that
193         computes the error on a batch from the testing set
194
195         :type datasets: list of pairs of theano.tensor.TensorType
196         :param datasets: It is a list that contain all the datasets;
```

```
197                     the has to contain three pairs, `train`,
198                     `valid`, `test` in this order, where each pair
199                     is formed of two Theano variables, one for the
200                     datapoints, the other for the labels
201         :type batch_size: int
202         :param batch_size: size of a minibatch
203         :type learning_rate: float
204         :param learning_rate: learning rate used during finetune stage
205
206         '''
207
208         (train_set_x, train_set_y) = datasets[0]
209         (valid_set_x, valid_set_y) = datasets[1]
210         (test_set_x, test_set_y) = datasets[2]
211
212         # compute number of minibatches for training, validation and testing
213         n_valid_batches = valid_set_x.get_value(borrow=True).shape[0]
214         n_valid_batches /= batch_size
215         n_test_batches = test_set_x.get_value(borrow=True).shape[0]
216         n_test_batches /= batch_size
217
218         index = T.lscalar('index')  # index to a [mini]batch
219
220         # compute the gradients with respect to the model parameters
221         gparams = T.grad(self.finetune_cost, self.params)
222
223         # compute list of fine-tuning updates
224         updates = []
225         for param, gparam in zip(self.params, gparams):
226             updates.append((param, param - gparam * learning_rate))
227
228         train_fn = theano.function(
229             inputs=[index],
230             outputs=self.finetune_cost,
231             updates=updates,
232             givens={
233                 self.x: train_set_x[
234                     index * batch_size: (index + 1) * batch_size
235                 ],
236                 self.y: train_set_y[
237                     index * batch_size: (index + 1) * batch_size
238                 ]
239             }
240         )
241
242         test_score_i = theano.function(
243             [index],
244             self.errors,
245             givens={
246                 self.x: test_set_x[
247                     index * batch_size: (index + 1) * batch_size
248                 ],
249                 self.y: test_set_y[
250                     index * batch_size: (index + 1) * batch_size
251                 ]
252             }
253         )
254
255         valid_score_i = theano.function(
256             [index],
257             self.errors,
258             givens={
259                 self.x: valid_set_x[
260                     index * batch_size: (index + 1) * batch_size
261                 ],
262                 self.y: valid_set_y[
263                     index * batch_size: (index + 1) * batch_size
264                 ]
265             }
266         )
267
268         # Create a function that scans the entire validation set
269         def valid_score():
270             return [valid_score_i(i) for i in range(n_valid_batches)]
271
272         # Create a function that scans the entire test set
273         def test_score():
```

```
274             return [test_score_i(i) for i in range(n_test_batches)]
275
276         return train_fn, valid_score, test_score
277
278
279 def test_DBN(finetune_lr=0.1, pretraining_epochs=100,
280              pretrain_lr=0.01, k=1, training_epochs=1000,
281              dataset='mnist.pkl.gz', batch_size=10, hidden_sizes=[1000, 1000, 1000], n_v=28 * 28,n_output=10):
282     """
283     Demonstrates how to train and test a Deep Belief Network.
284
285     This is demonstrated on MNIST.
286
287     :type finetune_lr: float
288     :param finetune_lr: learning rate used in the finetune stage
289     :type pretraining_epochs: int
290     :param pretraining_epochs: number of epoch to do pretraining
291     :type pretrain_lr: float
292     :param pretrain_lr: learning rate to be used during pre-training
293     :type k: int
294     :param k: number of Gibbs steps in CD/PCD
295     :type training_epochs: int
296     :param training_epochs: maximal number of iterations ot run the optimizer
297     :type dataset: string
298     :param dataset: path the the pickled dataset
299     :type batch_size: int
300     :param batch_size: the size of a minibatch
301     """
302
303     datasets = load_data(dataset)
304
305     train_set_x, train_set_y = datasets[0]
306     valid_set_x, valid_set_y = datasets[1]
307     test_set_x, test_set_y = datasets[2]
308
309     # compute number of minibatches for training, validation and testing
310     n_train_batches = train_set_x.get_value(borrow=True).shape[0] / batch_size
311
312     # numpy random generator
313     numpy_rng = numpy.random.RandomState(123)
314     print '... building the model'
315     # construct the Deep Belief Network
316     dbn = DBN(numpy_rng=numpy_rng, n_ins=n_v,
317               hidden_layers_sizes=hidden_sizes,
318               n_outs=n_output)
319
320     # start-snippet-2
321     #########################
322     # PRETRAINING THE MODEL #
323     #########################
324     log = []
325     print '... getting the pretraining functions'
326     pretraining_fns = dbn.pretraining_functions(train_set_x=train_set_x,
327                                                 batch_size=batch_size,
328                                                 k=k)
329
330     print '... pre-training the model'
331     start_time = timeit.default_timer()
332     ## Pre-train layer-wise
333     for i in range(dbn.n_layers):
334         # go through pretraining epochs
335         for epoch in range(pretraining_epochs):
336             # go through the training set
337             c = []
338             for batch_index in range(n_train_batches):
339                 c.append(pretraining_fns[i](index=batch_index,
340                                             lr=pretrain_lr))
341             print 'Pre-training layer %i, epoch %d, cost ' % (i, epoch),
342             print numpy.mean(c)
343
344
345     end_time = timeit.default_timer()
346     # end-snippet-2
347     print >> sys.stderr, ('The pretraining code for file ' +
348                           os.path.split(__file__)[1] +
349                           ' ran for %.2fm' % ((end_time - start_time) / 60.))
350     ########################
```

```
    # FINETUNING THE MODEL #
    ########################

    # get the training, validation and testing function for the model
    print '... getting the finetuning functions'
    train_fn, validate_model, test_model = dbn.build_finetune_functions(
        datasets=datasets,
        batch_size=batch_size,
        learning_rate=finetune_lr
    )

    print '... finetuning the model'
    # early-stopping parameters
    patience = 4 * n_train_batches  # look as this many examples regardless
    patience_increase = 2.    # wait this much longer when a new best is
                              # found
    improvement_threshold = 0.995  # a relative improvement of this much is
                                   # considered significant
    validation_frequency = min(n_train_batches, patience / 2)
                                  # go through this many
                                  # minibatches before checking the network
                                  # on the validation set; in this case we
                                  # check every epoch

    best_validation_loss = numpy.inf
    test_score = 0.
    start_time = timeit.default_timer()

    done_looping = False
    epoch = 0

    while (epoch < training_epochs) and (not done_looping):
        epoch = epoch + 1
        for minibatch_index in range(n_train_batches):

            minibatch_avg_cost = train_fn(minibatch_index)
            iter = (epoch - 1) * n_train_batches + minibatch_index

            if (iter + 1) % validation_frequency == 0:

                validation_losses = validate_model()
                this_validation_loss = numpy.mean(validation_losses)
                print(
                    'epoch %i, minibatch %i/%i, validation error %f %%'
                    % (
                        epoch,
                        minibatch_index + 1,
                        n_train_batches,
                        this_validation_loss * 100.
                    )
                )

                # if we got the best validation score until now
                if this_validation_loss < best_validation_loss:

                    #improve patience if loss improvement is good enough
                    if (
                        this_validation_loss < best_validation_loss *
                        improvement_threshold
                    ):
                        patience = max(patience, iter * patience_increase)

                    # save best validation score and iteration number
                    best_validation_loss = this_validation_loss
                    best_iter = iter

                    # test it on the test set
                    test_losses = test_model()
                    test_score = numpy.mean(test_losses)
                    print(('     epoch %i, minibatch %i/%i, test error of '
                           'best model %f %%') %
                          (epoch, minibatch_index + 1, n_train_batches,
                           test_score * 100.))

            if patience <= iter:
                done_looping = True
                break
```

```
428
429     end_time = timeit.default_timer()
430     print(
431         (
432             'Optimization complete with best validation score of %f %%, '
433             'obtained at iteration %i, '
434             'with test performance %f %%'
435         ) % (best_validation_loss * 100., best_iter + 1, test_score * 100.)
436     )
437     print >> sys.stderr, ('The fine tuning code for file ' +
438                           os.path.split(__file__)[1] +
439                           ' ran for %.2fm' % ((end_time - start_time)
440                                               / 60.))
441     return dbn
442
443
444 if __name__ == '__main__':
445     test_DBN()
```

Listing 12 : Implementasi Multi Layers Perceptron :

```
1   """
2   This tutorial introduces the multilayer perceptron using Theano.
3
4    A multilayer perceptron is a logistic regressor where
5   instead of feeding the input to the logistic regression you insert a
6   intermediate layer, called the hidden layer, that has a nonlinear
7   activation function (usually tanh or sigmoid) . One can use many such
8   hidden layers making the architecture deep. The tutorial will also tackle
9   the problem of MNIST digit classification.
10
11  .. math::
12
13      f(x) = G( b^{(2)} + W^{(2)}( s( b^{(1)} + W^{(1)} x))),
14
15  References:
16
17      - textbooks: "Pattern Recognition and Machine Learning" -
18                   Christopher M. Bishop, section 5
19
20  """
21
22  from __future__ import print_function
23
24  __docformat__ = 'restructedtext en'
25
26
27  import os
28  import sys
29  import timeit
30
31  import numpy
32
33  import theano
34  import theano.tensor as T
35
36
37  from logistic_sgd import LogisticRegression, load_data
38
39
40  # start-snippet-1
41  class HiddenLayer(object):
42      def __init__(self, rng, input, n_in, n_out, W=None, b=None,
43                   activation=T.tanh):
44          """
45          Typical hidden layer of a MLP: units are fully-connected and have
46          sigmoidal activation function. Weight matrix W is of shape (n_in,n_out)
47          and the bias vector b is of shape (n_out,).
48
49          NOTE : The nonlinearity used here is tanh
50
51          Hidden unit activation is given by: tanh(dot(input,W) + b)
52
53          :type rng: numpy.random.RandomState
54          :param rng: a random number generator used to initialize weights
55
```

```
56          :type input: theano.tensor.dmatrix
57          :param input: a symbolic tensor of shape (n_examples, n_in)
58
59          :type n_in: int
60          :param n_in: dimensionality of input
61
62          :type n_out: int
63          :param n_out: number of hidden units
64
65          :type activation: theano.Op or function
66          :param activation: Non linearity to be applied in the hidden
67                             layer
68          """
69          self.input = input
70          # end-snippet-1
71
72          # `W` is initialized with `W_values` which is uniformely sampled
73          # from sqrt(-6./(n_in+n_hidden)) and sqrt(6./(n_in+n_hidden))
74          # for tanh activation function
75          # the output of uniform if converted using asarray to dtype
76          # theano.config.floatX so that the code is runable on GPU
77          # Note : optimal initialization of weights is dependent on the
78          #         activation function used (among other things).
79          #         For example, results presented in [Xavier10] suggest that you
80          #         should use 4 times larger initial weights for sigmoid
81          #         compared to tanh
82          #         We have no info for other function, so we use the same as
83          #         tanh.
84          if W is None:
85              W_values = numpy.asarray(
86                  rng.uniform(
87                      low=-numpy.sqrt(6. / (n_in + n_out)),
88                      high=numpy.sqrt(6. / (n_in + n_out)),
89                      size=(n_in, n_out)
90                  ),
91                  dtype=theano.config.floatX
92              )
93              if activation == theano.tensor.nnet.sigmoid:
94                  W_values *= 4
95
96              W = theano.shared(value=W_values, name='W', borrow=True)
97
98          if b is None:
99              b_values = numpy.zeros((n_out,), dtype=theano.config.floatX)
100             b = theano.shared(value=b_values, name='b', borrow=True)
101
102         self.W = W
103         self.b = b
104
105         lin_output = T.dot(input, self.W) + self.b
106         self.output = (
107             lin_output if activation is None
108             else activation(lin_output)
109         )
110         # parameters of the model
111         self.params = [self.W, self.b]
112
113
114 # start-snippet-2
115 class MLP(object):
116     """Multi-Layer Perceptron Class
117
118     A multilayer perceptron is a feedforward artificial neural network model
119     that has one layer or more of hidden units and nonlinear activations.
120     Intermediate layers usually have as activation function tanh or the
121     sigmoid function (defined here by a ``HiddenLayer`` class)  while the
122     top layer is a softmax layer (defined here by a ``LogisticRegression``
123     class).
124     """
125
126     def __init__(self, rng, input, n_in, n_hidden, n_out):
127         """Initialize the parameters for the multilayer perceptron
128
129         :type rng: numpy.random.RandomState
130         :param rng: a random number generator used to initialize weights
131
132         :type input: theano.tensor.TensorType
```

```
133         :param input: symbolic variable that describes the input of the
134         architecture (one minibatch)
135
136         :type n_in: int
137         :param n_in: number of input units, the dimension of the space in
138         which the datapoints lie
139
140         :type n_hidden: int
141         :param n_hidden: number of hidden units
142
143         :type n_out: int
144         :param n_out: number of output units, the dimension of the space in
145         which the labels lie
146
147         """
148
149         # Since we are dealing with a one hidden layer MLP, this will translate
150         # into a HiddenLayer with a tanh activation function connected to the
151         # LogisticRegression layer; the activation function can be replaced by
152         # sigmoid or any other nonlinear function
153         self.hiddenLayer = HiddenLayer(
154             rng=rng,
155             input=input,
156             n_in=n_in,
157             n_out=n_hidden,
158             activation=T.tanh
159         )
160
161         # The logistic regression layer gets as input the hidden units
162         # of the hidden layer
163         self.logRegressionLayer = LogisticRegression(
164             input=self.hiddenLayer.output,
165             n_in=n_hidden,
166             n_out=n_out
167         )
168         # end-snippet-2 start-snippet-3
169         # L1 norm ; one regularization option is to enforce L1 norm to
170         # be small
171         self.L1 = (
172             abs(self.hiddenLayer.W).sum()
173             + abs(self.logRegressionLayer.W).sum()
174         )
175
176         # square of L2 norm ; one regularization option is to enforce
177         # square of L2 norm to be small
178         self.L2_sqr = (
179             (self.hiddenLayer.W ** 2).sum()
180             + (self.logRegressionLayer.W ** 2).sum()
181         )
182
183         # negative log likelihood of the MLP is given by the negative
184         # log likelihood of the output of the model, computed in the
185         # logistic regression layer
186         self.negative_log_likelihood = (
187             self.logRegressionLayer.negative_log_likelihood
188         )
189         # same holds for the function computing the number of errors
190         self.errors = self.logRegressionLayer.errors
191
192         # the parameters of the model are the parameters of the two layer it is
193         # made out of
194         self.params = self.hiddenLayer.params + self.logRegressionLayer.params
195         # end-snippet-3
196
197         # keep track of model input
198         self.input = input
199
200
201 def test_mlp(learning_rate=0.01, L1_reg=0.00, L2_reg=0.0001, n_epochs=1000,
202              dataset='mnist.pkl.gz', batch_size=20, n_hidden=500):
203     """
204     Demonstrate stochastic gradient descent optimization for a multilayer
205     perceptron
206
207     This is demonstrated on MNIST.
208
209     :type learning_rate: float
```

```
210     :param learning_rate: learning rate used (factor for the stochastic
211     gradient
212
213     :type L1_reg: float
214     :param L1_reg: L1-norm's weight when added to the cost (see
215     regularization)
216
217     :type L2_reg: float
218     :param L2_reg: L2-norm's weight when added to the cost (see
219     regularization)
220
221     :type n_epochs: int
222     :param n_epochs: maximal number of epochs to run the optimizer
223
224     :type dataset: string
225     :param dataset: the path of the MNIST dataset file from
226                  http://www.iro.umontreal.ca/~lisa/deep/data/mnist/mnist.pkl.gz
227
228
229     """
230     datasets = load_data(dataset)
231
232     train_set_x, train_set_y = datasets[0]
233     valid_set_x, valid_set_y = datasets[1]
234     test_set_x, test_set_y = datasets[2]
235
236     # compute number of minibatches for training, validation and testing
237     n_train_batches = train_set_x.get_value(borrow=True).shape[0] // batch_size
238     n_valid_batches = valid_set_x.get_value(borrow=True).shape[0] // batch_size
239     n_test_batches = test_set_x.get_value(borrow=True).shape[0] // batch_size
240
241     ######################
242     # BUILD ACTUAL MODEL #
243     ######################
244     print('... building the model')
245
246     # allocate symbolic variables for the data
247     index = T.lscalar()  # index to a [mini]batch
248     x = T.matrix('x')  # the data is presented as rasterized images
249     y = T.ivector('y')  # the labels are presented as 1D vector of
250                         # [int] labels
251
252     rng = numpy.random.RandomState(1234)
253
254     # construct the MLP class
255     classifier = MLP(
256         rng=rng,
257         input=x,
258         n_in=28 * 28,
259         n_hidden=n_hidden,
260         n_out=10
261     )
262
263     # start-snippet-4
264     # the cost we minimize during training is the negative log likelihood of
265     # the model plus the regularization terms (L1 and L2); cost is expressed
266     # here symbolically
267     cost = (
268         classifier.negative_log_likelihood(y)
269         + L1_reg * classifier.L1
270         + L2_reg * classifier.L2_sqr
271     )
272     # end-snippet-4
273
274     # compiling a Theano function that computes the mistakes that are made
275     # by the model on a minibatch
276     test_model = theano.function(
277         inputs=[index],
278         outputs=classifier.errors(y),
279         givens={
280             x: test_set_x[index * batch_size:(index + 1) * batch_size],
281             y: test_set_y[index * batch_size:(index + 1) * batch_size]
282         }
283     )
284
285     validate_model = theano.function(
286         inputs=[index],
```

```
287         outputs=classifier.errors(y),
288         givens={
289             x: valid_set_x[index * batch_size:(index + 1) * batch_size],
290             y: valid_set_y[index * batch_size:(index + 1) * batch_size]
291         }
292     )
293 
294     # start-snippet-5
295     # compute the gradient of cost with respect to theta (sotred in params)
296     # the resulting gradients will be stored in a list gparams
297     gparams = [T.grad(cost, param) for param in classifier.params]
298 
299     # specify how to update the parameters of the model as a list of
300     # (variable, update expression) pairs
301 
302     # given two lists of the same length, A = [a1, a2, a3, a4] and
303     # B = [b1, b2, b3, b4], zip generates a list C of same size, where each
304     # element is a pair formed from the two lists :
305     #    C = [(a1, b1), (a2, b2), (a3, b3), (a4, b4)]
306     updates = [
307         (param, param - learning_rate * gparam)
308         for param, gparam in zip(classifier.params, gparams)
309     ]
310 
311     # compiling a Theano function `train_model` that returns the cost, but
312     # in the same time updates the parameter of the model based on the rules
313     # defined in `updates`
314     train_model = theano.function(
315         inputs=[index],
316         outputs=cost,
317         updates=updates,
318         givens={
319             x: train_set_x[index * batch_size: (index + 1) * batch_size],
320             y: train_set_y[index * batch_size: (index + 1) * batch_size]
321         }
322     )
323     # end-snippet-5
324 
325     ###############
326     # TRAIN MODEL #
327     ###############
328     print('... training')
329 
330     # early-stopping parameters
331     patience = 10000  # look as this many examples regardless
332     patience_increase = 2  # wait this much longer when a new best is
333                            # found
334     improvement_threshold = 0.995  # a relative improvement of this much is
335                                    # considered significant
336     validation_frequency = min(n_train_batches, patience // 2)
337                                 # go through this many
338                                 # minibatche before checking the network
339                                 # on the validation set; in this case we
340                                 # check every epoch
341 
342     best_validation_loss = numpy.inf
343     best_iter = 0
344     test_score = 0.
345     start_time = timeit.default_timer()
346 
347     epoch = 0
348     done_looping = False
349 
350     while (epoch < n_epochs) and (not done_looping):
351         epoch = epoch + 1
352         for minibatch_index in range(n_train_batches):
353 
354             minibatch_avg_cost = train_model(minibatch_index)
355             # iteration number
356             iter = (epoch - 1) * n_train_batches + minibatch_index
357 
358             if (iter + 1) % validation_frequency == 0:
359                 # compute zero-one loss on validation set
360                 validation_losses = [validate_model(i) for i
361                                      in range(n_valid_batches)]
362                 this_validation_loss = numpy.mean(validation_losses)
363 
```

```
                print(
                    'epoch %i, minibatch %i/%i, validation error %f %%' %
                    (
                        epoch,
                        minibatch_index + 1,
                        n_train_batches,
                        this_validation_loss * 100.
                    )
                )

                # if we got the best validation score until now
                if this_validation_loss < best_validation_loss:
                    #improve patience if loss improvement is good enough
                    if (
                        this_validation_loss < best_validation_loss *
                        improvement_threshold
                    ):
                        patience = max(patience, iter * patience_increase)

                    best_validation_loss = this_validation_loss
                    best_iter = iter

                    # test it on the test set
                    test_losses = [test_model(i) for i
                                   in range(n_test_batches)]
                    test_score = numpy.mean(test_losses)

                    print(('     epoch %i, minibatch %i/%i, test error of '
                           'best model %f %%') %
                          (epoch, minibatch_index + 1, n_train_batches,
                           test_score * 100.))

            if patience <= iter:
                done_looping = True
                break

    end_time = timeit.default_timer()
    print(('Optimization complete. Best validation score of %f %% '
           'obtained at iteration %i, with test performance %f %%') %
          (best_validation_loss * 100., best_iter + 1, test_score * 100.))
    print(('The code for file ' +
           os.path.split(__file__)[1] +
           ' ran for %.2fm' % ((end_time - start_time) / 60.)), file=sys.stderr)


if __name__ == '__main__':
    test_mlp()
```