**Report DNN untuk Tesis**

**Sumber:**

<https://visualstudiomagazine.com/articles/2014/06/01/deep-neural-networks.aspx>

<https://visualstudiomagazine.com/Articles/2015/04/01/Back-Propagation-Using-C.aspx?Page=1>

<http://yann.lecun.com/exdb/mnist/>

<http://en.wikipedia.org/wiki/Constant-weight_code>

<http://en.wikipedia.org/wiki/Fisher%E2%80%93Yates_shuffle>

<http://ssli.ee.washington.edu/~mhwang/pub/>

**Prepocessing**

Membuat jaringan syaraf tiruan dengan DNN (Deep Neural Network). DNN memiliki banyak arti, tapi yang paling mendasar adalah jaringan syaraf yang memiliki dua atau lebih. Pada program yang telah dibuat adalah contoh DNN feed-forward dengan 2 lapis hidden layer yaitu firstHidden dan secondHidden. Diambil data dari sampel data digit MNIST. Data MNIST adalah data untuk pengenalan tulisan tangan khusus untuk digit angka yang terdiri dari 60.000 data training dan 10.000 data testing.

[train-images-idx3-ubyte.gz](http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz):  training set images (9912422 bytes)   
[train-labels-idx1-ubyte.gz](http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz):  training set labels (28881 bytes)   
[t10k-images-idx3-ubyte.gz](http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz):   test set images (1648877 bytes)   
[t10k-labels-idx1-ubyte.gz](http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz):   test set labels (4542 bytes)

Data tersebut dikonversi dengan format csv. Pada dasarnya data tersebut merupakan data gambar yang diwakili dengan gambar angka grayscale yang memiliki panjang 28x28 pixel. Untuk memperhalus data dilakukan preprocessing yaitu dengan menormalisasi data sehingga memiliki range data dari 0-1 dengan membangi dengan nilai 255. Gambar grayscale memiliki range 0-255.

X = (X - min) / (max - min)

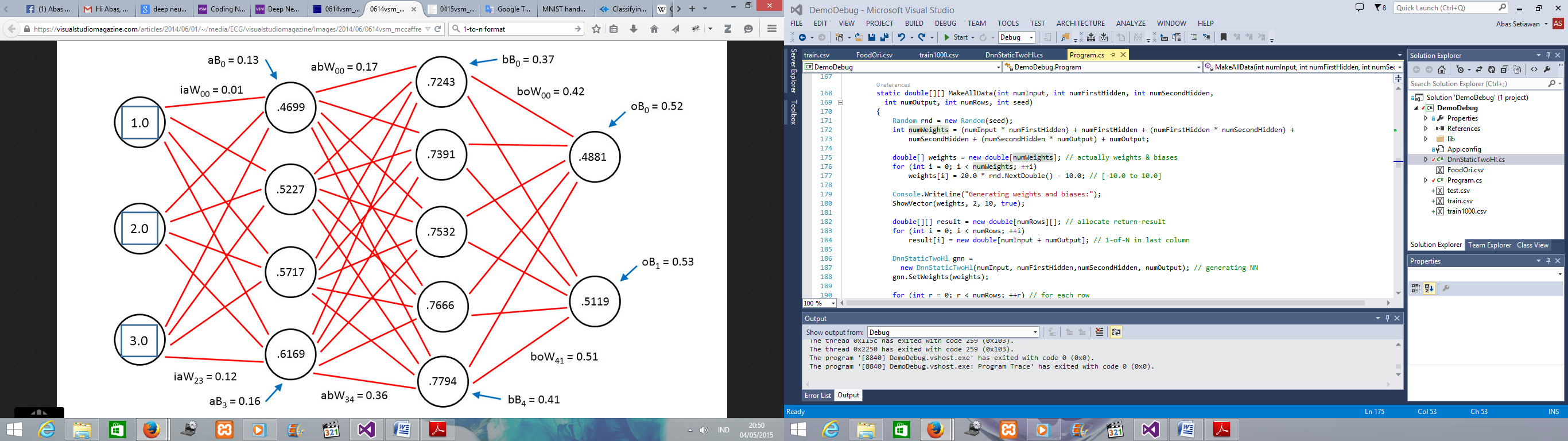
X = (X - 0) / (255 - 0)

X = X / 255

Akan tetapi karena keterbatasan yang ada maka hanya digunakan 300 data dari 70000 data total dari MNIST dikarenakan keterbatasan dalam algoritma error backpropagation. Keterbatasan tersebut akan dilanjutkan diakhir penjelasan ini. Banyaknya neuron input pada layer input menyesuaikan ukuran data per-gambar pada database MNIST yaitu 784 (28x28). Pada output layer menggunakan format 1-to-N encoding. Karena data digit adalah angka 0 sampai 9, maka jika dibinerkan akan memiliki panjang 4bit. Diambil contoh 9 akan menjadi 0100, maka dari itu akan di encoding dengan menjadi nilai 1000000000 sedangkan 0 akan menjadi 0000, maka dengan 1-to-N encoding menjadi 0000000001. Banyaknya neuron output yang digunakan adalah 10 (Sama dengan jumlah N bit pada encoding).

**DNN tanpa training**

Contoh arsitektur jaringan DNN yaitu 784-28-32-10. Nilai bobot dan bias akan di inisialisasi secara random dengan range -10 sampai 10. Misalkan terdapat arsitektur jaringan DNN 3-4-5-2 sebagai berikut:



Pada lapisan hidden pertama dan pada neuron pertama perhitungan bobot dan biasnya adalah sebagai berikut:

(1.0)(0.01) + (2.0)(0.05) + (3.0)(0.09) + 0.13 = 0.51

Kemudian pada local output di lapisan tersebut akan dimasukan dalam fungsi aktivasi tanh.

tanh(0.51) = 0.4699

Fungsi aktivasi tanh terbukti meningkatkan akurasi jika dibandingkan dengan sigmoid. Akan dijelaskan nanti pada saat training.

Dari lapisan hidden pertama dilanjutkan ke lapisan hidden kedua. Dengan contoh perhitungan pada neuron pertama pada hidden layer kedua.

(0.4699)(0.17) + (0.5227)(0.22) + (0.5717)(0.27) + (0.6169)(0.32) = 0.5466

Dan,ditambah dengan bias maka,

0.5466 + 0.37 = 0.9166

Setelah itu, dimasukkan dalam fungsi aktivasi tanh,

tanh(0.9166) = 0.7243

Pola yang sama diikuti untuk menghitung lapisan hidden kedua lainnya: 0,7391, 0,7532, 0,7666 dan 0,7794. Nilai untuk node output akhir [0] dan [1] dihitung dengan cara yang sedikit berbeda karena aktivasi Softmax digunakan untuk memaksa jumlah dari output sampai 1,0. Dengan contoh perhitungan pada neuron awal pada layer output:

0.7243)(0.43) + (0.7391)(0.45) + (0.7532)(0.47) + (0.7666)(0.49) + (0.7794)(0.51) + 0.53 = 2.3012

Aplikasikan softmax, final output [0] = exp(2.2536) / (exp(2.2536) + exp(2.3012)) = 0.4881. Dan final output [1] = exp(2.3012) / (exp(2.2536) + exp(2.3012)) = 0.5119

**DNN dengan Error Backpropagation(EBP)**

Dalam training DNN menggunakan EBP data akan di pisah 80% untuk data training dan 20% untuk testing. Beberapa parameter training dari EBP seperti maksimum iterasi dari proses training adalah 1000. Untuk mengontrol seberapa cepat proses pelatihan maka learning rate diset dengan nilai 0.05. Terakhir momentum, adalah parameter opsional untuk meningkatkan kecepatan pelatihan dan ditetapkan dengan nilai 0,01. Parameter tersebut dapat diubah-ubah sesuai keperluan training dan biasanya ditentukan oleh trial and error.

Menentukan kapan harus berhenti pelatihan jaringan saraf adalah masalah yang sulit. Di sini, menggunakan 1.000 iterasi adalah nilai perkiraan saja. Kontrol Tingkat belajar berapa banyak setiap bobot dan bias nilai dapat berubah dalam setiap langkah pembaruan. Nilai-nilai yang lebih besar meningkatkan kecepatan pelatihan di risiko overshooting nilai bobot yang optimal. Tingkat momentum membantu mencegah pelatihan terjebak pada local minimum, nilai-nilai bobot non-optimal dan juga mencegah osilasi di mana pelatihan pernah menyatu dengan nilai-nilai yang stabil.

Ketepatan model pada data uji memberikan perkiraan yang sangat kasar dari seberapa akurat model akan ketika disajikan dengan data baru yang memiliki nilai output yang tidak diketahui. Ketepatan model pada data pelatihan ini berguna untuk menentukan apakah model over-fitting telah terjadi. Jika akurasi prediksi model pada data pelatihan secara signifikan lebih besar daripada akurasi pada data uji, maka ada kemungkinan kuat bahwa over-fitting telah terjadi dan re-training dengan nilai parameter baru diperlukan.

**Implementasi Error Back-Propagation Training**

Back-propagation didasarkan pada calculus partial derivative. Setiap bobot dan bias nilai memiliki turunan parsial yang saling terkait. Dapat dipikirkan bahwa turunan parsial sebagai nilai yang berisi informasi tentang berapa banyak, dan ke arah, nilai bobot disesuaikan untuk mengurangi error kesalahan. Pengumpulan semua derivatif parsial disebut gradien. Namun, untuk kesederhanaan, setiap derivatif parsial umumnya hanya disebut gradien.

Tujuan dari pelatihan pada EBP adalah perubahan nilai bobot. Terdiri dari menghitung nilai delta menggunakan learning rate dan gradien. Pada arsitektur standar NN yang mempunyai 1 hidden layer dan 1 output layer. Maka akan terbentuk dua error gradien sinyal yang berhubungan yaitu output sinyal dan hidden sinyal. Pada arsitektur DNN terdapat 2 hidden layer dan 1 output layer maka terbentuk 3 error gradien sinyal.

double[] oSignals = new double[numOutput];

double[] fhSignals = new double[numFirstHidden];

double[] shSignals = new double[numSecondHidden];

Istilah momentum adalah opsional, tetapi hampir selalu digunakan dengan Back-propagation. Momentum membutuhkan nilai-nilai bobot dan bias delta (∆) dari iterasi pelatihan sebelumnya.

double[][] ifhPrevWeightsDelta = MakeMatrix(numInput, numFirstHidden, 0.0);

double[] fhPrevBiasesDelta = new double[numFirstHidden];

double[][] fhshPrevWeightsDelta = MakeMatrix(numFirstHidden, numSecondHidden, 0.0);

double[] shPrevBiasesDelta = new double[numSecondHidden];

double[][] shoPrevWeightsDelta = MakeMatrix(numSecondHidden, numOutput, 0.0);

double[] oPrevBiasesDelta = new double[numOutput];

int epoch = 0;

double[] xValues = new double[numInput]; // inputs

double[] tValues = new double[numOutput]; // target values

double derivative = 0.0;

double errorSignal = 0.0;

Selain itu juga terdapat variabel epoch sebagai nilai pencacahan untuk setiap iterasi. Array xValues memegang nilai input dari data pelatihan. Sedangkan Array tValues memegang nilai target dari data pelatihan. Variable derivative berhubungan dengan simbol dan . Variabel errorSignal adalah nilai pengurangan antara nilai pada output layer dikurangi dengan nilai target. Variabel errInterval mengontrol seberapa sering untuk menghitung dan menampilkan error saat selama pelatihan. Sebelum loop pelatihan dimulai, array yang memegang indeks dari data pelatihan harus dibuat terlebih dahulu.

int[] sequence = new int[trainData.Length];

for (int i = 0; i < sequence.Length; ++i)

sequence[i] = i;

Selama pelatihan, penting untuk menyajikan data pelatihan dalam nilai dengan urutan acak yang berbeda setiap kali melalui loop pelatihan. Urutan array akan diacak dan digunakan untuk menentukan urutan item pelatihan akan diproses. Berikutnya, metode Train dimulai dari main loop pelatihan.

int errInterval = maxEpochs / 10; // interval to check error

while (epoch < maxEpochs)

{

++epoch;

if (epoch % errInterval == 0 && epoch < maxEpochs)

{

//get error from MSE

double trainErr = Error(trainData);

Console.WriteLine("epoch = " + epoch + " MSE error = " +

trainErr.ToString("F4"));

}

Shuffle(sequence);

........

Menghitung Mean Square Error adalah operasi yang mahal karena semuanya pelatihan harus digunakan dalam perhitungan. Namun, ada banyak yang bisa salah selama pelatihan sehingga sangat dianjurkan untuk memantau kesalahan. Metode pembantu Shuffle mengacak indeks disimpan dalam array urutan menggunakan algoritma Fisher-Yates. Berikutnya, setiap item pelatihan diproses:

for (int ii = 0; ii < trainData.Length; ++ii)

{

int idx = sequence[ii];

Array.Copy(trainData[idx], xValues, numInput);

Array.Copy(trainData[idx], numInput, tValues, 0, numOutput);

ComputeOutputs(xValues);

Nilai input dan target yang ditarik dari item pelatihan saat ini. Nilai input dimasukkan ke metode ComputeOutputs, yang tidak hanya itu, menyimpan nilai-nilai output internal. Perhatikan bahwa eksplisit nilai return array dari ComputeOutputs diabaikan. Berikutnya, sinyal output node dihitung.

, dimana adalah error sinyal pada node j dan adalah nilai turunana dari fungsi output dari node j.

, untuk output layer yang menggunakan softmax

adalah nilai output pada layer pada node j.

, dimana adalah nilai target.

Jadi persamaan diatas akan menjadi seperti berikut.

for (int k = 0; k < numOutput; ++k)

{

errorSignal = tValues[k] - finalOutputs[k]; // Wikipedia uses (o-t)

derivative = (1 - finalOutputs[k]) \* finalOutputs[k]; // for softmax

oSignals[k] = errorSignal \* derivative;

}

Di sini, derivatif dihitung dengan asumsi bahwa lapisan output node menggunakan aktivasi Softmax, yang merupakan bentuk fungsi sigmoid logistik. Dalam situasi yang tidak memungkinkan bahwa untuk menggunakan fungsi aktivasi yang berbeda maka bagian kode ini harus dirubah.

Karena pada DNN dibelakang layer output adalah hidden layer ke dua (arsitektur dengan dua hidden layer), jadi akan dihitung bobot dan bias antara hidden-kedua dengan output.

// 2. compute second-hidden-to-output weight gradients using output signals

for (int j2 = 0; j2 < numSecondHidden; ++j2)

for (int k = 0; k < numOutput; ++k)

shoGrads[j2][k] = oSignals[k] \* shOutputs[j2];

// 2b. compute output bias gradients using output signals

for (int k = 0; k < numOutput; ++k)

shobGrads[k] = oSignals[k] \* 1.0; // dummy assoc. input value

Perhatikan bahwa nilai 1.0 adalah masukan yang terkait dengan output gradien bias. Ini digunakan untuk menggaambarkan kemiripan antara bobot pada layer tersembunyi dengan output, dimana nilai input yang terkait datang dari nilai local output hidden layer node (dalam hal ini hidden layer kedua), dan output node bias, yang memiliki sebuah nilai konstan dummy 1.0. Berikutnya adalah menghiddung hidden node sinyal.

for (int j2 = 0; j2 < numSecondHidden; ++j2)

{

derivative = (1 + shOutputs[j2]) \* (1 - shOutputs[j2]); // for tanh

double sum = 0.0; // need sums of output signals times second-hidden-to-output //weights

for (int k = 0; k < numOutput; ++k)

{

sum += oSignals[k] \* shToOutWeight[j2][k]; // represents error signal

}

shSignals[j2] = derivative \* sum;

}

Perhitungan ini menerapkan persamaan,

, untuk j pada hidden layer dimana,

, untuk hidden layer yang menggunakan fungsi tanh.

adalah nilai turunan dari . adalah lokal output dari node j yang merupakan hidden layer. Jika dijabarkan maka persamaan diatas akan menjadi,

dan jantung algoritma back-propagation. Ini terlihat sangat tidak jelas. Pada Wikipedia sudah dijelaskan dengan indah dengan matematika yang rumit. Pada Wikipedia mengasumsikan fungsi sigmoid logistik digunakan untuk lapisan simpul aktivasi tersembunyi. Dalam tulisan ini digunakan fungsi hiperbolik tangen (tanh) dan nilai variabel derivatif sesuai dengan tanh. Perhatikan perhitungan sinyal simpul tersembunyi memerlukan nilai-nilai sinyal output node, pada dasarnya bekerja mundur. Inilah sebabnya mengapa algoritma ini diberi nama back-propagation.

Perhitungan yang telah disebutkan tadi diatas akan diulang sampai pada koneksi antara hidden layer pertama dengan input layer. Di sini, pernyataan menghitung nilai delta variabel sesuai dengan (∆). Bahwa nilai delta disimpan untuk digunakan dalam iterasi berikutnya untuk menerapkan mekanisme momentum. Perhatikan bahwa semua bobot dan nilai bias diperbarui setelah gradien telah dihitung untuk item data pelatihan tunggal. Pada prinsipnya, gradien harus dihitung dengan mengumpulkan sinyal kesalahan atas semua item pelatihan. Tapi memperbarui bobot dan bias setelah setiap item data pelatihan pada dasarnya adalah memperkirakan gradien secara keseluruhan dan lebih efisien. Pendekatan ini biasanya disebut pelatihan "online" atau "stochastic". Alternatif ini biasanya disebut pelatihan "batch".

, adalah laju pelatihan atau learning rate dan adalah gradien.

Akan dimasukkan dalam,

, dimana ini adalah persamaan untuk pembaharuan semua bobot dan bias.

Jika terdapat momentum maka persamaan tersebut derubah menjadi,

, adalah bobot sebelumnya.

Persamaan tersebut dapatlebih dijabarkan sebagai berikut.

+

// update input-to-first-hidden weights

for (int i = 0; i < numInput; ++i)

{

for (int j1 = 0; j1 < numFirstHidden; ++j1)

{

double delta = ifhGrads[i][j1] \* learningRate;

inputToFhWeight[i][j1] += delta; // would be -= if (o-t)

inputToFhWeight[i][j1] += ifhPrevWeightsDelta[i][j1] \* momentum;

ifhPrevWeightsDelta[i][j1] = delta; // save for next time

}

}

// update first-hidden biases

for (int j1 = 0; j1 < numFirstHidden; ++j1)

{

double delta = fhbGrads[j1] \* learningRate;

fhBias[j1] += delta;

fhBias[j1] += fhPrevBiasesDelta[j1] \* momentum;

fhPrevBiasesDelta[j1] = delta;

}

Lakukan update bobot dan bias pada hidden layer pertama ke hidden layer kedua dan hidden layer kedua ke output layer dengan souce seperti diatas. Pada tahapan ini, bobot terbaik dan nilai bias disimpan dalam objek jaringan syaraf. Fungsi Train disimpulkan dengan mengambil beberapa nilai dan dikembalikan lagi pada jaringan syaraf.

......

} // each training item

}// while

double[] bestWts = GetWeights();

return bestWts;

Di sini, salinan serial dari bobot internal dan nilai-nilai bias dibuat menggunakan metode GetWeights. Alternatif adalah untuk refactor metode Train untuk mengembalikan void, dan kemudian memanggil GetWeights yang diperlukan setelah pelatihan.