Multilayer Perceptron Neural Networks
And a Software Classification Example

Paavo Nieminen

Department of Mathematical Information Technology
University of Jyväskylä

Data Mining Course (TIES447), Lecture of Nov 17, 2008
Outline

1. Basic Idea of Artificial Neural Networks (ANN)
   - Inspiration from Biological Neural Cells
   - Multilayered Perceptron (MLP)
   - Other Neural Architectures

2. Training of a Neural Network, and Use as a Classifier
   - How to Encode Data for an ANN
   - How Good or Bad Is a Neural Network
   - Backpropagation Training

3. An Example Implementation
   - How to make an ANN
A biological neuron receives electrochemical signals from many sources; when the excitation is high enough, the neuron fires, passing on the signal.

Since the 1950’s this analogy has been used as the basis of a family of mathematical models (and in marketing such models with a really cool name).
An artificial neuron combines its input signals for example by a weighted sum. The output is one numerical value, computed from a so called activation function, modeling approximately a “firing” of a natural neuron.

Mathematically, for example, \( o = f(b + \sum_{i=1}^{n} w_i(a)_i) \), where \( o \) is the output, \((a)_i\) are \( n \) inputs, \( w_i \) are summation weights, and \( f \) is the activation function; \( b \) is a bias term that determines how much activation induces firing.

→ Demo: Let us use Octave to plot activation functions (step function, logistic sigmoid, hyperbolic tangent)
Let us put many artificial neurons next to each other. Each neuron has its own weight coefficients, and an activation function is operating in each neuron. The input and the output are both numerical vectors.

Mathematically, for example, $(o_l)_j = f^l_j(b^l_j + \sum_{i=1}^{n_l-1} w^l_{j,i} (o^{l-1})_i)$, where $o^l$ is the output vector, $(o^{l-1})_i$ are $n_{l-1}$ inputs, $w^l_{j,i}$ is the weight of the $i$:th input in the $j$:th neuron, $f^l_j$ is the activation function and $b^l_j$ the bias of the $j$:th neuron. The indices $l - 1$ for the input vector and $l$ for the output vector anticipate the next step, where layers are interconnected.
By combining layers, we get an MLP

- Layers of different sizes can be chained.
- The output of each layer is computed using the same mechanism.
- The output from layer $l$ is fed forward to layer $l + 1$ (when there are no feed-back loops; things get slightly more complex if there are).
- With a bit of tweaking, the whole formulation with weights, biases, and functions, can be shown in a layer-wise matrix form; each layer has a coefficient matrix and a “function matrix”...
A compact notation for a simple MLP is

$$\mathbf{o}^0 = \mathbf{x}, \quad \mathbf{o}^l = \mathcal{F}^l(\mathbf{W}^l \hat{\mathbf{o}}^{(l-1)}) \quad \text{for} \quad l = 1, \ldots, L. \quad (1)$$

Here $\mathbf{x}$ is the input vector. We set it as the "output of the zeroth layer". The special hat notation $\hat{\mathbf{o}}^{(l-1)}$ represents an operation where a number 1 is prepended to a vector, increasing its dimension; this way the bias terms of layer $l$ can be written as the first column of matrix $\mathbf{W}^l$. The notation $\mathcal{F}^l$ means that the activation function is applied to all components of a vector (it is a “diagonal function matrix”).
Remarks About the Simple MLP

- The logistic sigmoid or the hyperbolic tangent are common choices for the activation function. These functions are differentiable.

- Usually you would use the same activation function on all layers.

- On the last layer, one can leave out the activation (i.e., “have linear activation”). Then the final layer outputs won’t be confined to the range $[0, 1]$ (logsig) or $[-1, 1]$ (tanh).

- But these, and many other variations, are to be made according to the task at hand.
Applications of Neural Networks

- MLP is a universal approximator: It can be shown that any continuous function can be approximated arbitrarily well by putting together a big enough hidden layer with suitable coefficient values. (But the proof of existence gives no construction advice for building such a net!)

- Such an approximator could obviously be a good tool for approximating an unknown function.

- Other applications include classification, denoising, control of machines . . .

- See literature using the keyword ”Neural network” or pick for starters something like http://en.wikipedia.org/wiki/Artificial_neural_network
What has been shown here is a simple artificial feed-forward neural network with fully connected weights. There are many variations (better for some tasks).

Connections can be omitted, they can go further down than the next layer, weight coefficients can be shared among connections, . . .

Feed-back loops, with delay, are possible. These recurrent ANNs are objects with an internal state (memory), well-suited for time-series forecasting and signal processing.

Radial basis function networks (RBF), Support Vector Machines (SVM)

Self-Organizing Maps (SOM)

Again, there is quite a lot of information available by searching with the above keywords.
Do we become slaves of the machines on 2015?

- One day people will program a machine that is more intelligent than humanity (see Matrix the Motion Picture)?
- It will be interesting to see. What we have today with ANNs is in the order of $10^5$ neurons, and the models are probably still off from real biological examples of intelligence. Regardless of the model, there would be something like $10^{12}$ neurons in the human brain.
- Apparently we humans are still many innovations away from being enslaved by our machines!
- We are quite safe during 2015.
- And if it tries to enslave you, pour some coffee on it, and it’ll die. I tested this last week; it worked fine, and the laptop broke before it had me enslaved.

(Let’s have a fifteen minute break here.)
Outline

1. Basic Idea of Artificial Neural Networks (ANN)
   - Inspiration from Biological Neural Cells
   - Multilayered Perceptron (MLP)
   - Other Neural Architectures

2. Training of a Neural Network, and Use as a Classifier
   - How to Encode Data for an ANN
   - How Good or Bad Is a Neural Network
   - Backpropagation Training

3. An Example Implementation
   - How to make an ANN
We would like a classifier that works as well as possible for unforeseen data, i.e., generalizes well.

What we have available is called the training set (of labeled data items).

We might want to use a validation set, which we use to make decisions while we train the net.

Then we might have a test set, that tells whether the training methodology using train and validation sets really works. (Or we do comparisons between methods.)

The train/validation/test sets are split from the available data set; preferably it is big enough!

How well the MLP works, obviously depends on how well the data set used in training represents reality (and how well the cost function gets minimized).
Goal setting: What kind of a network do we need?

For example, we need a classifier:

- The MLP should approximate a function that will tell you the category of an item, for example if a software module like a Java class belongs to the category “probably buggy” or to “probably not buggy”.

This time we consider only the above. But to give a different example, consider a predictor:

- The MLP should approximate a function that will give out a continuous measure, based on others, for example the man-months probably needed to create a software product, based on some design parameters.
Everything Needs to Be in Vectors

- An MLP operates on numerical vectors, so everything needs to be coded as ordered n-tuples of floating point numbers.
- For example, use software metrics (LOC, . . .).
- The numbers are called “features”.
- Multiple features can be concatenated into a feature vector (input to the ANN).
Coding of Nominal Features

- When a feature is essentially the name of a category, it is usually coded as an integer number (as in: 1=GUI class, 2=IO class, 3=Database class).

- BUT in the case of an MLP classifier, we should convert these to binary vectors, for example: [1 -1 -1] = GUI class, [-1 1 -1] = IO class, [-1 -1 1] = Database class.

- The same should be done with the outputs of the classification task (i.e., names of the classes).

- Reason: Nominal entries lack a natural order, so the distance of their encoding should be the same between each two classes.
Let us evaluate how bad a certain MLP is!

If we can minimize the value of badness, we get a good MLP.

Suitable cost function is for example the mean squared error $J(\{\mathbf{W}\}) = \frac{1}{2N} \sum_{i=1}^{N} \| \mathcal{N}(\mathbf{x}_i) - \mathbf{y}_i \|^2$ where $\mathcal{N}(\mathbf{x}_i)$ is the output for a training vector $\mathbf{x}_i$ and $\mathbf{y}_i$ is the known truth about where the ideal classifier would place $\mathbf{x}_i$.

So, this is basically a least square sum problem . . .
How is the cost function affected by each weight coefficient

Let us write the error induced by the \( i \):th training-target vector pair as \( e_i = \mathcal{N}(\{W^l\})(x_i) - y_i \). It turns out that the gradient of the mean squared cost function can be computed iteratively in a layer-wise matrix form by using the backpropagation of error:

\[
\nabla_{W^l} J(\{W^l\}) = \frac{1}{N} \sum_{i=1}^{N} \xi_i^l [\hat{o}_i^{(l-1)}]^T,
\]

where (assuming no activation on layer \( L \))

\[
\xi_i^L = e_i, \tag{2}
\]

\[
\xi_i^l = \text{Diag}\{(F^l)'(W^l \hat{o}_i^{(l-1)})\}(W_1^{(l+1)})^T \xi_i^{(l+1)}. \tag{3}
\]

In the equation \( W_1^{(l+1)} \) denotes a matrix that you get by removing the first column (containing bias terms) of \( W^{(l+1)} \).
Let us Go Downhill

Now we have a cost function that tells how bad some set of weights are for the training data. We are also able to compute the gradient of this function. The simplest way to minimize badness, thereby making the MLP work better is the steepest descent method:

- On every iteration, we look at the gradient and take a small step in the opposite direction. If the step size is small enough, a local minimum is bound to be found. We recognize such when a bottom (small gradient) is reached.

If the step size is too big, we find nothing useful; if it is too small, it takes a long time to get anywhere. Nevertheless, steepest descent works, and it is very commonly used.
On these slides we have a differentiable cost function and the simplest possible training method that uses gradient information.

The field of mathematics known as “Optimization” continuingly develops better methods for minimizing a function.

Some methods applicable for MLP training are conjugate gradient methods, Broyden-Fletcher-Goldfarb-Shanno (BFGS), Levenberg-Marquardt (LM) . . .

Also genetic algorithms have been used with success.

The cost can be formulated differently, including non-smooth functions (that are at least in some points non-differentiable). We have methods for those, too!

So... learn more by searching with the keyword “optimization”
Outline

1. Basic Idea of Artificial Neural Networks (ANN)
   - Inspiration from Biological Neural Cells
   - Multilayered Perceptron (MLP)
   - Other Neural Architectures

2. Training of a Neural Network, and Use as a Classifier
   - How to Encode Data for an ANN
   - How Good or Bad Is a Neural Network
   - Backpropagation Training

3. An Example Implementation
   - How to make an ANN
Let us look at some code. (Made with Matlab; can be run also in Octave)

Even though we have Matlab available for use at the university, I tend to prefer Octave because Matlab requires pretty stupid stupid silly connections with the license server, and that has brought just problems, problems, most of the time. And the last time I tried, my home Linux refused to co-operate with Matlab, invoking security policy issues; very inconvenient indeed.

Don’t let this personal opinion affect your decisions, though. Matlab is a beautiful tool when it doesn’t boil your head.
Summary

This was a 95-minute mini-introduction to the idea, implementation, and application (as a classifier) of an artificial neural network (ANN) of the kind that is commonly called a multilayer perceptron (MLP). Key points:

- Inspired by nature, we have matrix computations and sigmoidal functions.
- A few dozen lines of Matlab code is enough to create a working MLP. (When using C, Java, or such, you’ll need a library for matrix computations, but the MLP code itself remains small.)
- Gradient-based optimization is suitable for training.
- Need train, validation, and test sets of known input-output pairs.
- First attempts for the demonstration case yielded 70% accuracy of bug detection. Admittedly, this can be “tuulitulos”.

Paavo Nieminen
Multilayer Perceptron Neural Networks