A new approach towards implementing artificial neural networks*

Grzegorz Klima†

December 15, 2013
(rev. August 26, 2015)

Abstract

The purpose of this paper is to present an innovative approach to implementing feed forward artificial neural networks (ANN). Typical implementation based on neuron, layer and network structures, and teaching algorithms explicitly traversing the network has many weaknesses which fall into three areas: computational speed, code modularity, and extensibility. The proposed approach, which addresses all these issues, is based on the insights from sparse matrix computations (Compressed Column Storage). Versions of feed forward and backpropagation algorithms taking full advantage of proposed storage schemes are presented. The described implementation strategy makes ANN code truly modular, easily extensible and, as shown in benchmarks, improves computational speed.

Keywords: artificial neural network, feed forward network, backpropagation, performance tuning, code modularization, sparse storage, C, C++.

1 Introduction

The artificial neural networks (ANNs) have been a subject of research and have been used in applied work for many years. Owing to their universal approximating capabilities and increased availability of computational power, ANNs (especially feed forward multilayer networks after rediscovery of backpropagation [1], [14]) have been applied in various areas from pattern recognition to forecasting. For introduction to the ANN see [13].

The research within the subfield of feed forward ANN has covered various topics including: improving convergence properties of the backpropagation algorithm (e.g. [7], [12]), devising pruning (e.g. [10], [8]) and construction algorithms (e.g. [6], [5]). Little effort has been put into analysis of

*This paper presents results of an independent research project. Views expressed herein are solely of the author and have not been approved by any institution the author has been/is employed by or affiliated with.

†Contact: gklima@users.sourceforge.net. I would like to thank Paweł Kowal and Michał Lipnicki for remarks on the early draft of the paper. Kaja Retkiewicz-Wijtiwiak has suggested many corrections that led to this revision of the paper. All mistakes remain mine.

computer implementation of ANNs and its impact on computational speed, although the developers of some libraries have introduced various optimizations achieving significant speed improvements over naive implementations (e.g. [11]).

This paper proposes new storage schemes for different types of feed forward ANNs. All the schemes are based on detailed analysis of network structure (connections between neurons) and insights from linear algebra computations (compressed storage of sparse matrices). Versions of feed forward and backpropagation algorithms taking full advantage of these schemes are presented. It is shown that proposed schemes allow for transparent interface of ANN classes / structures improving performance at the same time.

This paper is organized as follows. Section 2 reviews standard multilayer perceptron network and feed forward and backpropagation algorithms. Section 3 describes standard implementation strategy for multilayer perceptron networks and section 4 discusses its drawbacks. Section 5 describes the proposed implementation strategy. Section 6 reviews Fast Compressed Neural Networks library — a C++ library which applies the ideas described in this paper. Section 7 concludes.

2 Multilayer perceptron network

Consider a network with L layers $l = 1, \ldots, L$. Each neuron in layers $l = 2, \ldots, L$ is connected to neurons in the previous layer. The first layer is the input layer, while the last one is the output layer. Let N^l denote the number of neurons in layer l.

Suppose we have K records of observational data. Given $k = 1, \ldots, K$ input vectors $I^k = \left(I^k_1, \ldots, I^k_{N^1}\right)$ of length N^1, the network generates K output vectors $O^k = \left(O^k_1, \ldots, O^k_{N^L}\right)$ of length N^L.

Let $n_{l,i}$ denote the state of the ith neuron in the lth layer. The states of neurons in the first layer are set to the input values, i.e. $n_{1,i} = I^k_i$. Initial signals are transmitted over the network as follows: for each layer $l = 2, \ldots, L$ and each neuron $i = 1, \ldots, N^l$ in layer l, neuron state is computed according to the following formula:

$$n_{l,i} \leftarrow f\left(b_{l,i} + \sum_{j=1}^{N^{l-1}} w_{l,i,j} n_{l-1,j}\right), \tag{1}$$

where $w_{l,i,j}$ is the weight of connection between neuron i in layer l and neuron j in layer $l-1$, $b_{l,i}$ is the bias of neuron i in layer l and f is the so called activation function (usually a sigmoid function). The feed forward algorithm is presented below (Algorithm 1).

Define vector of all weights and biases as follows:

$$\mathbf{w} = \left(b_{2,1}, w_{2,1,1}, \ldots, b_{L,N^L}, \ldots, w_{L,N^L,N^{L-1}}\right).$$

The output of ANN may be thought of as a function F of inputs parametrized by weights and biases of neurons:

$$O^k = F(I^k; \mathbf{w}). \tag{2}$$

Suppose that for our K given inputs we know the observed outputs $\bar{O}^k = \left(\bar{O}^k_1, \ldots, \bar{O}^k_{N^L}\right)$. In the process of teaching ANN the weights and biases of neurons are chosen in order to minimize mean squared error:

$$\min_{\mathbf{w}} MSE = \min_{\mathbf{w}} \frac{1}{2KN^L} \sum_{k=1}^{K} \sum_{i=1}^{N^L} \left(O^k_i - \bar{O}^k_i\right)^2 = \min_{\mathbf{w}} \frac{1}{2KN^L} \sum_{k=1}^{K} \left\| F(I^k; \mathbf{w}) - \bar{O}^k \right\|^2. \tag{3}$$

♯ copy input
for $i \leftarrow 1, \ldots, N^1$ **do**
    $n_{1,i} \leftarrow I_i^k$
end for
♯ main loop
for $l \leftarrow 2, \ldots, L$ **do**
    for $i \leftarrow 1, \ldots, N^l$ **do**
        $signal_{l,i} \leftarrow b_{l,i}$ ♯ bias
        for $j \leftarrow 1, \ldots, N^{l-1}$ **do** ♯ dot product loop
            $signal_{l,i} \leftarrow signal_{l,i} + w_{l,i,j} n_{l-1,j}$
        end for
        $n_{l,i} \leftarrow f(signal_{l,i})$ ♯ activation function
    end for
end for

Algorithm 1: Feed forward algorithm for multilayer perceptron

The most popular methods of solving optimization problem in (3) are gradient methods employing the backpropagation algorithm for gradient computation (cf. [1], [14]).

Thanks to the structure of the network and the mathematical properties of (1) and (3), the gradient of MSE with respect to weights and biases can be computed in a recursive manner from the last layer to the second based on computations of the so called δs, i.e. the deviations of neuron state from its expected level (cf. [1], [14]). This explains the name — backpropagation of error. For detailed textbook exposition the reader is referred to [13]. In what follows $\delta_{l,i}$ denotes error of neuron i in layer l, $\nabla_{l,i,j}$ — the derivative of MSE w.r.t. the weight of connection between neuron i in layer l and neuron j in layer $l-1$ and $\nabla_{l,i,\text{bias}}$ — the derivative of MSE w.r.t. the bias of neuron i in layer l. Backpropagation (for one input / output pair) is presented as Algorithm 2. The gradient of total MSE is just a sum of (scaled) gradients obtained for each input / output pair.

Note that in the backpropagation algorithm we need to compute the derivative of each neuron's activation function w.r.t. its input signal. As sigmoid activation functions satisfy differential equations that allow computation of derivative based on value of a function only, there is no need to store input signals for each neuron in memory. Symmetric sigmoid activation function $f(x) = \tanh sx$, for instance, satisfies the following equation: $f'(x) = s(1 - f(x)^2)$.

3 A typical implementation of multilayer perceptron network

A standard implementation of the ANN is based on hierarchical structure in which a network consists of layers which in turn are collections of neurons. Each neuron structure/class contains information about its bias, neurons it is connected to, and the corresponding connection weights. Note that this information fully describes network structure and is sufficient for feed forward and backpropagation algorithms (assuming that the derivative of activation function can be computed based on its value only). In standard implementation, the aforementioned structures contain additional data fields, e.g. input signal, activation function pointers (or enumerative types representing them), activation function parameters, a field storing delta of a neuron etc. A simplified implementation along these lines is presented in Listing 1.

1: **call** feed_forward ♯ copy input vector I^k and perform computation
♯ initialise δs in the output layer and set to 0 in hidden layers
2: **for** $i \leftarrow N^L, \ldots, 1$ **do**
3: $\delta_{L,i} \leftarrow O_i^k - \bar{O}_i^k$
4: **end for**
5: **for** $l \leftarrow L-1, \ldots, 2$ **do**
6: **for** $i \leftarrow N^l, \ldots, 1$ **do**
7: $\delta_{l,i} \leftarrow 0$
8: **end for**
9: **end for**
♯ output layer and hidden layers except for the first
10: **for** $l \leftarrow L, \ldots, 3$ **do**
11: **for** $i \leftarrow N^l, \ldots, 1$ **do**
12: $\delta_{l,i} \leftarrow \delta_{l,i} f'(signal_{l,i})$ ♯ finalise error computation
13: **for** $j \leftarrow N^{l-1}, \ldots, 1$ **do**
14: $\delta_{l-1,j} \leftarrow \delta_{l-1,j} + w_{l,i,j}\delta_{l,i}$ ♯ backpropagate error
15: $\nabla_{l,i,j} \leftarrow \nabla_{l,i,j} + n_{l-1,j}\delta_{l,i}$ ♯ update derivative w.r.t. weight
16: **end for**
17: $\nabla_{l,i,\text{bias}} \leftarrow \nabla_{l,i,\text{bias}} + \delta_{l,i}$ ♯ update derivative w.r.t. bias
18: **end for**
19: **end for**
♯ first hidden layer
20: **for** $i \leftarrow N^2, \ldots, 1$ **do**
21: $\delta_{2,i} \leftarrow \delta_{2,i} f'(signal_{2,i})$ ♯ finalise error computation
22: **for** $j \leftarrow N^1, \ldots, 1$ **do**
23: $\nabla_{2,i,j} \leftarrow \nabla_{2,i,j} + n_{1,j}\delta_{2,i}$ ♯ update derivative w.r.t to weight
24: **end for**
25: $\nabla_{2,i,\text{bias}} \leftarrow \nabla_{2,i,\text{bias}} + \delta_{2,i}$ ♯ update derivative w.r.t. bias
26: **end for**

Algorithm 2: Backpropagation in a multilayer perceptron — step for one data record

Given the network structure described above, teaching algorithms are implemented in C as functions (taking pointer to the network structure) or as member functions of C++ network class traversing the network and simultaneously computing gradient and updating weights.

This standard approach has many drawbacks with most important being computational speed, poorly structured code operating on the ANN and difficulties with implementing new teaching, pruning, and construction algorithms. In what follows we analyse these issues in detail.

4 Drawbacks of typical implementation

4.1 Computational speed

The speed of dot product loop in the feed forward algorithm (Algorithm 1, lines 7-9) and the speed of activation function computation are critical factors affecting the overall ANN performance. Similarly, the speed of iteration in the backpropagation algorithm or any other gradient method

```
struct neuron {
    struct neuron *neur_pl; // array of pointers to neurons in the previous layer
    int npl;                // no.  of neurons in the previous layer
    float *weights;         // weights array (of size npl + 1 for bias)
    float state;            // neuron state
    float delta;            // additional field (delta for backpropagation)
    // more additional fields follow
    // ...
};

struct layer {
    struct neuron *neurons; // array of neurons
    int no_neurons;         // no.  of neurons in layer
    // additional fields follow
    // ...
};

struct network {
    struct layer *layers;   // array of layers
    int no_layers;          // no.  of layers
    // additional fields follow
    // ...
};
```

Listing 1: Typical C structures representing feed forward ANN

significantly depends on the inner loop (Algorithm 2, lines 13-17, 22-25) in which neuron deltas and derivatives of MSE w.r.t. weights are computed.

In both cases, the memory access times are crucial for performance — the fact that each neuron is a separately allocated object dramatically reduces computational speed. Discontinuous allocation of neuron states makes it impossible to take full advantage of processor cache combined with loop unrolling or use of contemporary processors' instruction sets (SIMD instructions). There are two possible ways to address this issue: preallocating array of neuron states and keeping pointers to this array in neurons instead of actual values or storing states of neurons in each layer outside neuron structure in a continuous array. Both approaches tend to obscure the code and involve introducing new data structures.

4.2 Code modularity and extensibility

That process of updating weights in gradient algorithms is intertwined with gradient computation is not imposed by the network structure itself. It appears to be consistent decision among neural network library developers. Teaching algorithms are not separated from network classes / structures to the point that their parameters are stored in network classes / structures. Cache optimization (storing neuron states continuously) further increases structure / class size and obscures interface. We shall argue that this is bad code design.

Firstly, storing all the information about the parameters of teaching algorithms in network structure is an implementation choice and not a requirement. This approach has significant negative consequences. Every time a new algorithm is implemented, new data fields have to be introduced

in otherwise functional ANN structure. As it has been pointed out earlier, the ANN structure / class size increases and its implementation is becoming less and less transparent.

Secondly, as we have noted earlier, teaching ANN is equivalent to minimising the function of weights (3). Suppose our goal is to devise or implement a new, faster gradient or quasi-Newton algorithm. The fact that process of updating weights is intertwined with gradient computation implies that implementing this new algorithm requires modification and possibly copying parts of the existing code. This involves unnecessary development cost, stealing focus from initial goal of improving optimisation algorithm. Simultaneously, integrity and security of the existing code is put at risk.

We note that having weights (and therefore gradients) discontinuously stored in memory would introduce some overhead if one wished to retrieve weights and gradient vectors and then store the updated weights (depending on update algorithm) in network. On the other hand, this cost would still be relatively small compared to time in which the network is traversed forward and backward for every input / output pair. Those runs involve dot product, activation function computation, and updating δs and derivatives.

The lack of functions retrieving and setting weights in a typical implementation makes it impossible to easily apply other optimisation algorithms, e.g. simulated annealing, genetic algorithms, or Newton algorithm based on numerical Hessian for small networks. Any attempt at implementing such algorithms would require introduction of the aforementioned functionality.

The hierarchical structure itself also poses problems with extending functionality of the ANN implementation. Suppose that the network is implemented as a collection of C++ classes written in conformance to basic principles of hermetisation and for some reason one wished to add some neuron specific information. Apart from expanding the neuron class data members and interface one would have to expand the interfaces of layer and network classes as well.

5 Proposed approach

5.1 Requirements for the interface

We specify the following conditions for the proper ANN implementation:

1. Object Oriented design — the ANN is a class.
2. Hermetisation — internal representation of the ANN data is completely hidden from the user.
3. In particular, the ANN class provides methods for retrieving weights and gradients (computed by backpropagation) and a method for updating weights.
4. Teaching algorithms are separated from the ANN class — they operate on weights and gradient vectors.
5. Pruning and construction algorithms are separated from the ANN class — the class interface provides member functions for turning connections on/off, removing and adding neurons, freezing weights (in cascade networks).
6. Internal representation of the ANN structure minimises memory access times in performance critical loops (feed forward and backpropagation algorithms).

As it turns out, the stipulated goals can be easily achieved using our proposed ANN structure implementation.

```
struct vv {
    int N;                  // no.  of vectors
    int *dims;              // individual vector lengths
    float **data;           // array of pointers to arrays
};
```

Listing 2: C structure representing vectors of different lengths

```
typedef std::vector<std::vector<float> > vv;
```

Listing 3: C++ class representing vectors of different lengths using STL containers

5.2 Storage — continuous representation of a vector of vectors

Suppose we have N vectors of different lengths $d^i : i = 0, 1, \ldots, N-1$. We here make use of 0-based indexing. A C data structure representing this type of data is presented in listing 2. C++ STL vector container would simplify this construction as shown in listing 3.

In either of two implementations referring to element j in vector i requires retrieving pointer to vector i and then the value using double indexing `vv_object[i][j]`. Data is not stored continuously in memory and the creation of an object of such class requires $N + 1$ memory allocations.

Suppose now we create one vector of length $\sum_{n=0}^{N-1} d^n$ stacking all the N vectors continuously in memory. The data in the first vector (index 0) starts at index 0, the data in the second vector (index 1) starts at index d^0, the data in the ith vector ($i > 0$) starts at index $\sum_{k=1}^{i} d^{k-1}$. Suppose we create a vector of integers p of length $N + 1$, satisfying the following conditions:

$$p^i = \begin{cases} 0 & : \quad i = 0, \\ p^{i-1} + d^{i-1} = \sum_{k=1}^{i} d^{k-1} & : \quad i = 1, \ldots, N \end{cases} . \tag{4}$$

Then the index of the jth value in vector i can be computed using our vector of 'pointers' p as $p^i + j$. A simple C++ class implementing this idea is presented in listing 4. The value p^N is the total length of our continuous vector (total data size). Note that indices $k = p^i, p^i + 1, \ldots, p^{i+1} - 1$ represent indices of all values in vector i, which is useful in indexing in loops.

The described approach has been successfully applied in the field of sparse matrix computations. A sparse matrix can be thought of as a vector of vectors of row indices and a vector of vectors of corresponding values, where each vector of row indices and vector of values represent one column. Storing the vector of all row indices and the vector of all values continuously and making use of

```
class vv {
  public:
    vv(const std::vector<int> &dims);                    // constructor
    float& get(int i, int j) { return d[p[i] + j];}      // element access
  private:
    std::vector<int> p;                                  // 'pointers'
    std::vector<float> d;                                // actual data
};
```

Listing 4: Vectors of different lengths using continuous storage and 'pointers' in C++

the vector of 'pointers' for indexing, we obtain what is called the Compressed Column Storage representation of sparse matrix. For detailed exposition and algorithms employing this representation see [2].

5.3 Application to multilayer perceptron ANN

As it has been pointed out earlier the ANN structure needs to hold the following data only: number of layers, numbers of neurons in layers, weights and biases, neuron states, and information about activation functions.

Representing neuron states is straightforward within our framework: all we need is just one vector of all neuron states and a vector of neuron pointers np constructed from a vector of numbers of neurons in layers. We shall be using our notation from section 2 but now with 0-based indices. The layers will now be numbered $l = 0, \ldots, L-1$ and neuron in each layer $i = 0, \ldots, N^l - 1$. We construct vector of neuron 'pointers' as follows:

$$np^l = \begin{cases} 0 & : \quad l = 0, \\ np^{l-1} + N^{l-1} = \sum_{k=1}^{l} N^{k-1} & : \quad l = 1, \ldots, L \end{cases} . \tag{5}$$

Using this vector of 'pointers' and one continuous vector of data, any other neuron specific characteristics may be stored (e.g. activation functions and their parameters if one wished for some reason to differentiate activation functions at neuron level). Index of neuron i in layer l (0-based) is given by $np^l + i$.

The representation of weights is a bit more complicated. First, note that each neuron in layer $l = 1, \ldots, L-1$ (0-based indexing) has the same number of connections (N^{l-1}) and the same number of associated weights ($N^{l-1} + 1$, including bias). This means that the total number of weights of neurons in layer l is equal to $\left(N^{l-1} + 1\right) N^l$. Consider the following vector of 'pointers' of length $L + 1$:

$$wp^l = \begin{cases} 0 & : \quad l = 0, 1, \\ wp^{l-1} + N^{l-1}\left(N^{l-2} + 1\right) = \sum_{k=2}^{l} N^{k-1}\left(N^{k-2} + 1\right) & : \quad l = 2, \ldots, L \end{cases} . \tag{6}$$

It is up to the decision of the developer whether to store biases before or after connection weights. Suppose we store all weights in one continuous vector with each neuron's bias stored before connection weights. The index of bias of neuron i in layer $l > 0$ is equal to $wp^l + (N^{l-1}+1)i$ and the weight of its connection with neuron j in the previous layer is stored at index $wp^l + (N^{l-1}+1)i + j + 1$. This expression is a bit complicated and would definitely introduce overhead if it were to be evaluated in a loop. However, as shown in Algorithm 3 and Algorithm 4, both feed forward and backpropagation algorithms can be written without evaluating any neuron or weight indices explicitly.

In what follows, we assume that neuron states are stored in vector n, δs in vector δ of the same length, weights in vector w, and derivatives w.r.t. those weights in vector ∇. Neuron and weight indices are ni and wi respectively. Feed forward and backpropagation algorithms adjusted to our representation are presented in Algorithm 3 and Algorithm 4. The idea behind these algorithms is to loop over neurons and weights continuously while maintaining the information about current layer and neuron.

Note that the design of both algorithms is crucial for taking full advantage of the proposed network representation. In both proposed algorithms weight indices are continuously incremented (decremented) and the inner loops allow for employing natural optimisation techniques (e.g. loop unrolling). Inner loops in both algorithms can be replaced by calls to optimised BLAS level-1 routines (cf. [9], [4], [3]).

```
    ♯ copy input
 1: for ni ← 0, ..., np^1 − 1 do
 2:     n[ni] ← I^k_ni
 3: end for
    ♯ main loop
 4: ni ← np^1, wi ← 0    ♯ initialise neuron and weight indices
 5: for l ← 1, ..., L − 1 do
 6:     for i ← 0, ..., N^l − 1 do
 7:         signal ← w[wi]    ♯ bias
 8:         wi ← wi + 1
 9:         npi ← np^{l−1}    ♯ index of neuron in the previous layer
10:         for j ← 0, ..., N^{l−1} − 1 do    ♯ dot product loop
11:             signal ← signal + w[wi] · n[npi]
12:             wi ← wi + 1, npi ← npi + 1
13:         end for
14:         n[ni] ← f(signal)    ♯ activation function
15:         ni ← ni + 1
16:     end for
17: end for
```

Algorithm 3: Feed forward algorithm for compressed representation of multilayer perceptron ANN with 0-based indexing

5.4 Not fully connected, relatively dense multilayer perceptron networks

Our approach turns out to be easily applicable to the networks that are not fully connected. To store information about status of a connection (on/off) we just need a vector of Boolean values of the same length indexed in the same fashion as vector of weights. Suppose we want to close connection (l, i, j) (or turn off bias of neuron (l, i)). One needs to set Boolean value to `false` and set weight to zero only.

When retrieving weights vector, one needs to know the number of active connections and when copying weights' values in a loop it is necessary to check whether the connection (or bias) is turned on. The feed forward algorithm (Algorithm 3) will work without modification since the values of turned off connections are set to 0. Adjustment of the backpropagation algorithm (Algorithm 4) is also trivial: since weights of inactive connections are set to 0, δs of neurons in the previous layer will not be influenced by them. With gradient computation one can either skip the inactive connections in the inner loop or perform the computation for all connections and later ignore the entries corresponding to weights that have been turned off. The latter option is faster as branching (conditional statement) in a performance critical loop significantly reduces execution speed.

Let us define the connection rate as a ratio of active connections to their total number given by $\sum_{k=2}^{L} N^{k-1}\left(N^{k-2}+1\right)$. One might think that this way of representing not fully connected networks involves memory and speed overhead measured by the reciprocal of connection rate. We argue that the overheads are actually overestimated for densely but not fully connected networks. The described situation is analogous to storing a relatively dense matrix using sparse representation — actually, more memory is used and speed is compromised.

Suppose that only weights of active connections are stored in memory. This means that weight j

1: **call** feed_forward ♯ copy input vector I^k and perform computation
♯ initialise δs in the output layer
2: **for** $ni \leftarrow np^L - 1, \ldots, np^{L-1}$, $i \leftarrow N^{L-1} - 1, \ldots, 0$ **do**
3: $\delta[ni] \leftarrow n[ni] - \bar{O}_i^k$
4: **end for**
♯ set δs in hidden layers to 0
5: **for** $ni \leftarrow np^{L-1} - 1, \ldots, np^1$ **do**
6: $\delta[ni] \leftarrow 0$
7: **end for**
8: $ni \leftarrow np^L - 1, wi \leftarrow wp^L - 1$ ♯ initialise neuron and weight indices
♯ output layer and hidden layers except for the first
9: **for** $l \leftarrow L - 1, \ldots, 2$ **do**
10: **for** $i \leftarrow N^l - 1, \ldots, 0$ **do**
11: $\delta[ni] \leftarrow \delta[ni] f'(signal[ni])$ ♯ finalise error computation
12: $npi \leftarrow np^l - 1$ ♯ index of neuron in the previous layer
13: **for** $j \leftarrow N^{l-1} - 1, \ldots, 0$ **do**
14: $\delta[npi] \leftarrow \delta[npi] + w[wi] \cdot \delta[ni]$ ♯ backpropagate error
15: $\nabla[wi] \leftarrow \nabla[wi] + n[npi] \cdot \delta[ni]$ ♯ update derivative w.r.t to weight
16: $wi \leftarrow wi - 1$, $npi \leftarrow npi - 1$
17: **end for**
18: $\nabla[wi] \leftarrow \nabla[wi] + \delta[ni]$ ♯ update derivative w.r.t. bias
19: $wi \leftarrow wi - 1$, $ni \leftarrow ni - 1$
20: **end for**
21: **end for**
♯ first hidden layer
22: **for** $i \leftarrow N^1, \ldots, 0$ **do**
23: $\delta[ni] \leftarrow \delta[ni] f'(signal[ni])$ ♯ finalise error computation
24: $npi \leftarrow np^l - 1$ ♯ index of neuron in the previous layer
25: **for** $j \leftarrow N^0 - 1, \ldots, 0$ **do**
26: $\nabla[wi] \leftarrow \nabla[wi] + n[npi] \cdot \delta[ni]$ ♯ update derivative w.r.t to weight
27: $wi \leftarrow wi - 1$, $npi \leftarrow npi - 1$
28: **end for**
29: $\nabla[wi] \leftarrow \nabla[wi] + \delta[ni]$ ♯ update derivative w.r.t. bias
30: $wi \leftarrow wi - 1$, $ni \leftarrow ni - 1$
31: **end for**

Algorithm 4: Backpropagation algorithm for compressed representation of multilayer perceptron ANN with 0-based indexing — step for one data record

is not necessarily the weight of connection with neuron j in the previous layer. Therefore, one needs a map (represented as a vector) from connection indices to indices of neurons in the previous layer. This additional array increases memory requirements. First, the dot product algorithm needs to check the index of neuron associated with a given weight and then jump to the neuron state vector at the retrieved index. An implementation along these lines is presented in Listing 5.

Because of the indirect indexing this dot product (even with some loop unrolling) will be slower than the dot product of two continuous vectors. The speed overhead caused by looping over inactive

```
float
dot_prod(float *a,   // dense vector
         float *b,   // nonzero values in sparse vector
         int *ind,   // indices of nonzero entries in sparse vector
         int n)      // length of vectors
{
    float s = 0.;
    for (int i = 0; i < n; ++i)
        s += a[ind[i]] * b[i];
    return s;
}
```

Listing 5: Dot product of dense and sparse vectors

connections (and adding zeros) will only be observable in truly sparse networks. The representation applicable in such situations is described in the next section.

5.5 General feed forward ANN

Our approach can be applied to general feed forward networks, i.e. networks without distinguished layers or sparsely connected multilayer perceptron networks. Consider a network consisting of N^I input, N^H hidden, and N^O output neurons. Input neurons do not take any other neurons' states as inputs and output neurons are not connected to each other. Each neuron $i \geq N^I$ (0-based indexing) may only be connected to neurons with indices $< \min\{i, N^I + N^H\}$. Set $N = N^I + N^H + N^O$. This type of network can be represented as an upper-triangular $N \times N$ sparse matrix A with $A_{j,i}$ being connection weight between neuron i and j and $A_{i,i}$ being bias of neuron i.

Let nc_i denote the number of (incoming) connections of the ith neuron, with $nc_i = 0$ for $i = 0, \ldots, N^I - 1$. Our vector of 'pointers' np_i is of length $N + 1$ and satisfies:

$$np_l = \begin{cases} 0 & : \quad l = 0 \\ np_{l-1} + nc_{l-1} = \sum_{k=1}^{l} nc_{k-1} & : \quad l = 1, \ldots, N \end{cases} . \tag{7}$$

Weights are represented as vector w of length np_N. Vector m of the same length stores indices of neurons corresponding to weights (a mapping from weights to neurons). This is effectively the Compressed Column Storage of a sparse incidence matrix representing network connections.

The dot product in the feed forward algorithm employs the idea from Listing 5. At the beginning of feed forward algorithm states of neurons past input layer are set to 1. This allows to compute input signal in a dot product loop only, without taking special considerations for bias. The feed forward algorithm for this structure is presented as Algorithm 5. Adjusting the backpropagation algorithm to this sparse representation in a similar fashion is straightforward and therefore, we omit its presentation here.

5.6 Cascade networks

Cascade networks are a special case of feed forward networks in which each neuron in the hidden layers is connected to all previous neurons. In this case a vector of 'pointers' is redundant as given

♯ copy input
1: **for** $ni \leftarrow 0, \ldots, N^I - 1$ **do**
2: $\quad n[ni] \leftarrow I^k_{ni}$
3: **end for**
♯ set remaining neuron states to 1
4: **for** $ni \leftarrow N^I, \ldots, N - 1$ **do**
5: $\quad n[ni] \leftarrow 1$
6: **end for**
♯ main loop
7: **for** $ni \leftarrow 0, \ldots, N - 1$ **do**
8: $\quad signal \leftarrow 0$
9: $\quad$ **for** $wi \leftarrow np_i, \ldots, np_{i+1} - 1$ **do** ♯ dot product loop
10: $\quad\quad signal \leftarrow signal + w[wi] \cdot n[m[wi]]$
11: $\quad$ **end for**
12: $\quad n[ni] \leftarrow f(signal)$ ♯ activation function
13: **end for**

Algorithm 5: Feed forward algorithm for compressed representation of sparse feed forward ANN with 0-based indexing

the index i, the 'pointer' to the beginning of ith neuron's weights can be computed as follows:

$$np_i = \begin{cases} 0 & : \quad i = 0, \ldots, N^I \\ \frac{(i-N^I)(i+N^I+1)}{2} & : \quad i = N^I + 1, \ldots, N^I + N^H - 1 \\ \frac{N^H(N^H+2N^I+1)}{2} + (N^I + N^H)(i - N^I - N^H) & : \quad i = N^I + N^H, \ldots, N \end{cases} . \quad (8)$$

Note that the representation of cascade network is essentially a slight modification of the packed storage of upper triangular matrix. This special structure can be easily utilised in the feed forward and backpropagation algorithms. In fact, the algorithms for this case are similar to Algorithm 3 and Algorithm 4 and, as before, allow for optimisation of inner loops. We omit their presentation for this case.

6 FCNN — an implementation of the idea

6.1 Overview

The Fast Compressed Neural Networks library is a C++ implementation of the approach described in this paper.[1] At current development stage it provides users with the multilayer perceptron network class template which implements the network structure described in section 5.4. This class admits the following activation functions (with the possibility of differentiating them between hidden and output layers):

- symmetric sigmoid,
- sigmoid,

[1] FCNN library can be downloaded from http://fcnn.sourceforge.net/. R interface to FCNN is available on CRAN at https://cran.r-project.org/web/packages/FCNN4R/.

- piecewise linear approximation of symmetric sigmoid,
- piecewise linear approximation of sigmoid,
- symmetric step,
- step,
- linear.

The following teaching (batch) algorithms are implemented:

- backpropagation (steepest descent),
- Rprop [12].

The FCNN library also implements the following pruning algorithms:

- minimum magnitude,
- Optimal Brain Surgeon [8].

The library provides users with `Matrix` class for basic operations and `Dataset` class for working with datasets stored in text files.

All classes are templated to allow for single and double precision computations.

6.2 Optimisations

FCNN uses the object structure and algorithms described in section 5.4. The only optimisations pertain to inner loops in the feed forward and backpropagation algorithms. The former is rewritten as separate inlined function with loops unrolled using template metaprogramming techniques. The inner loop in Algorithm 4 is rewritten using two calls to BLAS-like `axpy` function. This function is also inlined with loops unrolled using template metaprogramming techniques. Of course, some performance improvements could be achieved by using tuned/vendor BLAS library calls.[2]

6.3 Benchmarks

For benchmarks the Fast Artificial Neural Network (FANN) library [11] has been chosen. FANN is written in C with ports to many other languages and seems to be the most popular open source neural network library. It boasts to be faster up to 2 orders of magnitude than its competitors. These speed-ups relative to other libraries have been accomplished through cache optimisation and loop unrolling.

In what follows we use three benchmark network structures (fully connected) and problems from the FANN technical report [11]. We test both the feed forward average run time as well as the average iteration time for the backpropagation algorithm using one thread only.

All benchmarks have been run on Intel(R) Core(TM) i3-2367M CPU (1.40GHz) system with 10GB of RAM running Linux version 3.8.0-33-generic. All code has been compiled with `gcc` version 4.7.3.

In all of the networks activation function in the hidden layer was set to piecewise linear approximation of symmetric sigmoid function, while the output layer activation function was set to piecewise linear approximation of sigmoid function.

[2] This would be the case for large networks. For smaller ones the overhead of calling BLAS routine would outweigh the performance improvements from using tuned BLAS.

Network	No. of records	No. of runs	FCNN	FANN	Speed-up
125(in)-32-3(out)	4062	100	$5.44 \cdot 10^{-6}$	$2.26 \cdot 10^{-5}$	4.1
82(in)-16-8-19(out)	341	500	$3.26 \cdot 10^{-6}$	$1.16 \cdot 10^{-5}$	3.6
21(in)-16-8-3(out)	3600	200	$1.39 \cdot 10^{-6}$	$4.08 \cdot 10^{-6}$	2.9

Table 1: Feed forward average run times — FCNN vs FANN (double precision)

Network	No. of data	No. of runs	FCNN	FANN	Speed-up
125(in)-32-3(out)	4062	100	$1.27 \cdot 10^{-5}$	$5.85 \cdot 10^{-5}$	4.6
82(in)-16-8-19(out)	342	500	$7.39 \cdot 10^{-6}$	$3.29 \cdot 10^{-5}$	4.5
21(in)-16-8-3(out)	3600	200	$3.03 \cdot 10^{-6}$	$1.13 \cdot 10^{-5}$	3.7

Table 2: Backpropagation (gradient) average run times — FCNN vs FANN (double precision)

The benchmarks show that FCNN is about 4 times faster than FANN. The speed-up increases with the size of the network. It is worth mentioning that without any inner loop unrolling FCNN is still faster. This is mainly thanks to the continuous representation in memory.

7 Conclusions

We have shown that by employing insights from numerical linear algebra in the ANN design, one can obtain more efficient implementations of different types of feed forward ANNs. This should not be surprising — an ANN is a directed graph which can be represented as an (usually sparse) incidence matrix. We have also shown that this approach allows a modularised and easily extensible implementation with simultaneous speed improvements.

A properly designed ANN library with stable interface and efficient computational engine could allow researchers in the field to concentrate more on the algorithmic aspects of ANN computations and less on the gory details of the ANN implementation. It is hoped that this paper and the FCNN library will be a step towards this goal.

References

[1] A.E. Bryson and Y.C. Ho. *Applied optimal control: optimization, estimation, and control.* Blaisdell book in the pure and applied sciences. Blaisdell Pub. Co., 1969.

[2] Timothy A. Davis. *Direct Methods for Sparse Linear Systems (Fundamentals of Algorithms).* Society for Industrial and Applied Mathematics, September 2006.

[3] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Richard J. Hanson. Algorithm 656: an extended set of basic linear algebra subprograms: model implementation and test programs. *ACM Trans. Math. Softw.*, 14(1):18–32, 1988.

[4] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Richard J. Hanson. An extended set of FORTRAN basic linear algebra subprograms. *ACM Trans. Math. Softw.*, 14(1):1–17, 1988.

[5] S. E. Fahlman, L. D. Baker, and J. A. Boyan. The Cascade 2 Learning Architecture. Technical Report CMU-CS-TR-96-184, Carnegie Mellon University, 1996.

[6] S. E. Fahlman and C. Lebiere. The Cascade-Correlation Learning Architecture. In D. S. Touretzky, editor, *Advances in Neural Information Processing Systems*, volume 2, pages 524–532, Denver 1989, 1990. Morgan Kaufmann, San Mateo.

[7] Scott E. Fahlman. An empirical study of learning speed in back-propagation networks. Technical Report Computer Science Technical Report, 1988.

[8] B. Hassibi, D. G. Stork, and G. J. Wolff. Optimal Brain Surgeon and General Network Pruning. Technical Report CRC-TR-9235, RICOH California Research Centre, 1992.

[9] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for FORTRAN usage. *ACM Trans. Math. Softw.*, 5(3):308–323, 1979.

[10] Yann Le Cun, John S. Denker, and Sara A. Solla. Optimal Brain Damage. In *Advances in Neural Information Processing Systems*, volume 2, pages 598–605, 1990.

[11] Steffen Nissen. Implementation of a fast artificial neural network library (fann). Technical report, Department of Computer Science University of Copenhagen (DIKU), October 2003.

[12] M. Riedmiller. *Rprop - Description and Implementation Details: Technical Report.* Inst. f. Logik, Komplexität u. Deduktionssysteme, 1994.

[13] Raúl Rojas. *Neural Networks - A Systematic Introduction.* Springer-Verlag, Berlin, 1996.

[14] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, October 1986.