UNIVERSITÄT KOBLENZ · LANDAU

fnnlib: A Flexible C++ Library for Recurrent Neural Network Simulations

Studienarbeit

vorgelegt von

Dennis Faßbender

Betreuer: Dipl.-Inf. Björn Pelzer, Institut für Informatik, Fachbereich 4

Erstgutachter: Prof. Dr. Ulrich Furbach, Institut für Informatik, Fachbereich 4

Koblenz, im November 2010

THANKS

I would like to thank the following people/organizations:

Joschka Boedecker (Osaka University), for introducing me to recurrent neural networks and providing valuable input and assistance with problems during the development of fnnlib

Bjoern Pelzer (University of Koblenz), for supervising the creation of this thesis

Deutscher Akademischer Austausch Dienst (DAAD), for granting me a scholarship to finance my six-month stay at Osaka University, where I developed fnnlib

Contents

1 Introduction

1.1 Motivation

Following Frank Rosenblatt's 1958 paper on the Perceptron [5], a simple feed-forward neural network, artificial neural networks came to be considered a biologically plausible computer model of human learning. Over the decades, various types of networks have been developed, some of which will be introduced in section 2. The goal of this project was to create a flexible, easily extensible C++ library which allows its user to quickly create, train and test different network architectures, including ones which cannot be found in other libraries yet since they were only invented recently. For lack of creativity, I chose the name *fnnlib* – short for *Flexible Neural Network Library* – for the library.

I developed fnnlib during a stay at Osaka University's Graduate School of Engineering, where neural networks are used extensively in humanoid robotics research. As a result, the functionality included in the library was influenced by the special requirements of robot learning. The idea was to help researchers in general - and those at Osaka University in particular - implement and test neural networks for new research projects using a common library rather than new, potentially faulty code or recycled pieces of code.

fnnlib requires working copies of the C++ matrix library Armadillo[1] as well as LAPACK[2] and ATLAS[3].

1.2 Structure

Section 2 introduces the theory behind the different kinds of neural networks that can be created with fnnlib. Section 3 presents the library's design, i.e. the classes, their interrelationships, the motivations behind various design decisions, as well as suggestions for future additions or changes to the library. In section 4, a simple neural network will be created, trained and tested using fnnlib. Section 5 covers related work before the thesis is concluded in 6. Appendix A explains where to find fnnlib's complete documentation.

[1]http://arma.sourceforge.net/

[2]http://www.netlib.org/lapack/

[3]http://math-atlas.sourceforge.net/

2 Neural Network Theory

After a brief recapitulation of what a neural network is and how it is trained using standard backpropagation, this section will explain the Backpropagation Through Time (BPTT) algorithm, which was developed by Rumelhart et al. in 1986 [6] to train Recurrent Neural Networks. Slight variations of traditional BPTT are also used to train Recurrent Neural Networks with Parametric Bias (see section 2.3) and Continuous Time Recurrent Neural Networks (see section 2.4). Finally, section 2.5 will give an introduction to Echo State Networks, which are not trained by backpropagating errors but by solving linear systems of equations whose unknown variables are the weights of connections between neurons.

2.1 Feed-Forward Neural Networks & Backpropagation

A standard feed-forward neural network consists of an input layer, one or more hidden layers, and an output layer. Each of these layers is comprised of one or more neurons. Note that, in this thesis, the terms *neuron* and *unit* will be used interchangeably. In the most simple case, neurons in layer i are only connected to neurons in layer $i + 1$. While these connections are often referred to as *synapses*, we will stick to the term connection for the most part.
All connections are assigned weights, with the connection from neuron j to neuron k having the weight w_{kj}. The output, or activation, y of a neuron is computed by an activation function f. A simple feed-forward neural network may look like this:

The activations of the neurons in the different layers are computed successively, starting with the input layer. Its activation at time-step n is computed by applying f to the respective input written into the neurons at that time-step. The input of the following layers is computed as follows: for each incoming connection of a neuron j, the weight w_{ji} of the connection is multiplied by the activation of the neuron i at the opposite end of the connection; the activation function f is then applied to the sum $v_j(n)$ of those products, which is called the *induced local field*. Thus, the activation of neuron j at time-step n is computed according to the following formulas (the notation was adopted from Haykin [2]):

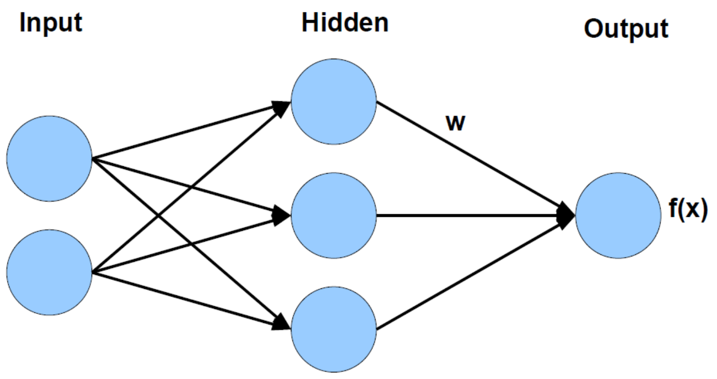

Figure 1: A simple feed-forward neural network

$$v_j(n) = \sum_i w_{ji} y_i(n) \tag{1}$$

$$y_j(n) = f(v_j(n)) \tag{2}$$

After N time-steps, the *errors* at the output neurons at the different time-steps n ($n = N, ..., 1$), i.e. the differences between their actual outputs $y_j(n)$ and their *desired outputs* $d_j(n)$ are computed:

$$e_j(n) = d_j(n) - y_j(n) \tag{3}$$

The goal is to minimize the mean squared error, which sums the squared errors of all output neurons (O denotes the set of indices of the output neurons) over all N time-steps and divides it by $2N$:

$$E = \frac{1}{2N} \sum_{n=1}^{N} \sum_{j \in O} e_j^2(n) \tag{4}$$

The goal now is to adjust the connection weights in such a way as to minimize the mean squared error. To this end, we will use the first derivative of the activation function and the error to calculate the *gradients* δ_j, starting at the output layer and iterating backwards over the layers until we reach the first hidden layer:

$$\delta_j(n) = \begin{cases} f'(v_j(n)) e_j(n); \text{ for output units} \\ f'(v_j(n)) \sum_k w_{kj} \delta_k(n); \text{ otherwise} \end{cases} \tag{5}$$

In the output layer, the error e_j is used to compute the gradient. In the hidden layers, there is no such error since there is no information about the desired output of these neurons. Instead, the sum of products of the weights of all outgoing connections and the gradients of the neurons at the opposite ends of the respective connections is multiplied by the value of the first derivative of the activation function. The errors at the output neurons are thus backpropagated through the network. Once all gradients have been computed for all time-steps n, starting at the last time-step $n = N$ and going back to the first time-step $n = 1$, the connection weights in the network are updated by adding Δw_{ji} to w_{ji}:

$$\Delta w_{ji} = \eta \sum_{n=1}^{N} \delta_j(n) y_i(n) \tag{6}$$

For each connection from a neuron i to a neuron j, Δw_{ji} is computed by iterating over all time-steps n and summing the products of $j's$ gradient and $i's$ activation at that time-step. That sum is then multiplied by the *learning rate* η, which is typically a small value chosen by the trainer.

2.2 Recurrent Neural Networks & Backpropagation Through Time

While feed-forward neural networks usually only have connections from layer n to layer $n+1$, as stated in section 2.1, neurons in recurrent neural networks (RNN) may be connected to any other neuron, including neurons in the same or a previous layer. They can be trained using a method called Backpropagation Through Time (BPTT), a slight variation of the standard backpropagation algorithm described in section 2.1. Figure 2 shows an example of a simple RNN.

This essentially means that the activation at time-step n of the hidden neurons in Figure 2 depends on their activations at time-step $n - 1$, which again was affected by the activations at time-step $n - 2$ and so on. Given the size N of a training epoch (i.e. the weights will be updated after N time-steps), an RNN can be thought of as a feed-forward network that was *unfolded in time*:

Since the epoch size was 3 in this case, there are now three copies of the old RNN. The recurrent connections have been transformed into forward connections, but they now connect the copy of neuron j at time-step n to its copy at $n+1$. As a result, the dependencies between

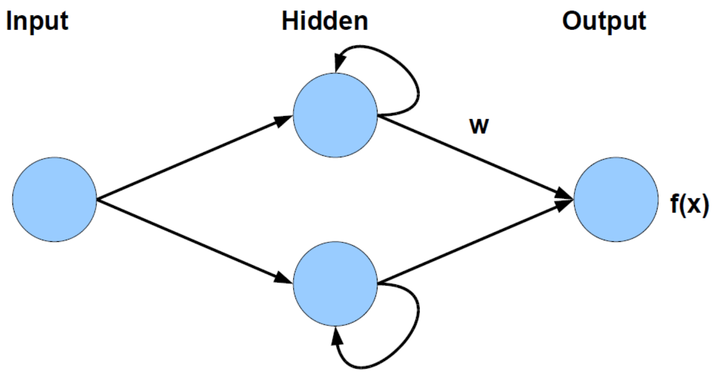

Figure 2: A simple recurrent neural network

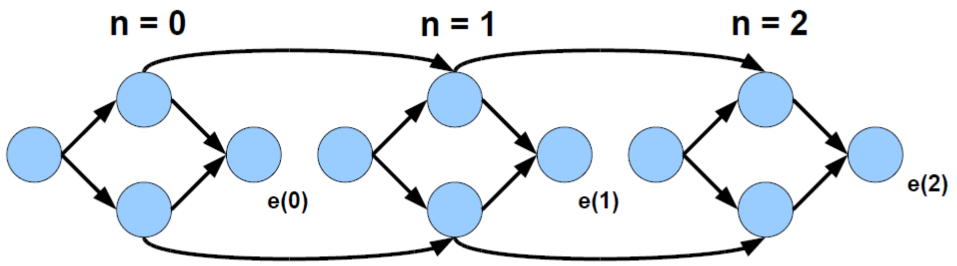

Figure 3: An RNN that was unfolded in time over three time-steps

the activations at different time-steps remain the same: $j's$ activation at time-step n still depends on its activation at the previous time-step.

In order to train the network, it is first necessary to compute the direct error at the output neuron *at all three time-steps* using the known formula

$$e_j(n) = d_j(n) - y_j(n) \tag{7}$$

Now, the gradients of the network copy at time-step $n+2$ are computed. This can be done using the standard backpropagation formula for feed-forward networks since the recurrent connections of the hidden layer have been removed. There is simply no further copy of the network to attach them to since $n+2$ is the last time-step. Afterwards, we have to go back one step in time to compute the gradients at time-step $n+1$.

The following formula is used to do this:

$$\delta_j(n) = f'(v_j(n)) \left[e_j(n) + \sum_k w_{kj} \delta_k(n') \right] \tag{8}$$

Note that $e_j(n)$ will be zero for all non-output neurons. Also, the value of n' depends on whether connection kj was a recurrent connection or not. In the former case, n' will be equal to $n + 1$ since the formerly recurrent connections now attach to the $(n+1)^{th}$ copy of the network. If the connection was not recurrent, n' will be equal to n. Similar care has to be taken when updating the weights. $\Delta(w_{ji})$ is now computed as follows:

$$\Delta w_{kj} = \eta \sum_{n=1}^{N} \delta_k(n) y_j(n') \tag{9}$$

While this formula is almost identical to the one used for feed-forward networks, n' once again depends on whether kj used to be a recurrent connection. If it did, then n' will be equal to $n - 1$, as the value transmitted over kj at time-step n was the product of the connection weight and the activation of neuron i at time-step $n - 1$. In the case of $n = 1$, $n - 1$ will refer to the non-existent time-step zero, so $y_j(n')$ will be set to zero. If kj was not recurrent, n' will be equal to n.

2.3 Recurrent Neural Networks with Parametric Bias

Thanks to the feedback loop created by recurrent connections, recurrent neural networks are well-suited for tasks such as time-series prediction. Training a network to predict a time-series works as follows: at time-step n during training, the input dataset $x(n)$ is written into the input layer, while dataset $x(n + 1)$ is given as the desired output for time-step n. If, at the end of the training process, the network has learned to generate the values $x(n+1)$ for all $x(n)$ of a pattern (e.g. a sine wave) with sufficient accuracy, it should be able to generate that pattern autonomously by writing its output back into its input layer after each step.

If a neural network is supposed to be a model of the human brain, however, it should be able to learn multiple time-series. In the case of a robot, this would be useful because one network would be able to generate several series of actuator commands, e.g. one which extends the robot's arm and one which opens its hand. Since this means that the same connection weights would have to be used to generate two or more different time-series when being presented their respective input values, the network's performance in generating each individual time-series would inevitably decrease dramatically as the number of time-series increases. As a remedy to this problem, Ito and Tani [3] came up with the idea of using bias neurons with internal values that are adjusted individually for each pattern learned by the network. (Note that the terms *time-series* and *pattern* will be used interchangeably.) An example of this type of "Recurrent Neural Network with Parametric Bias" (RNNPB) can be seen in Figure 4.

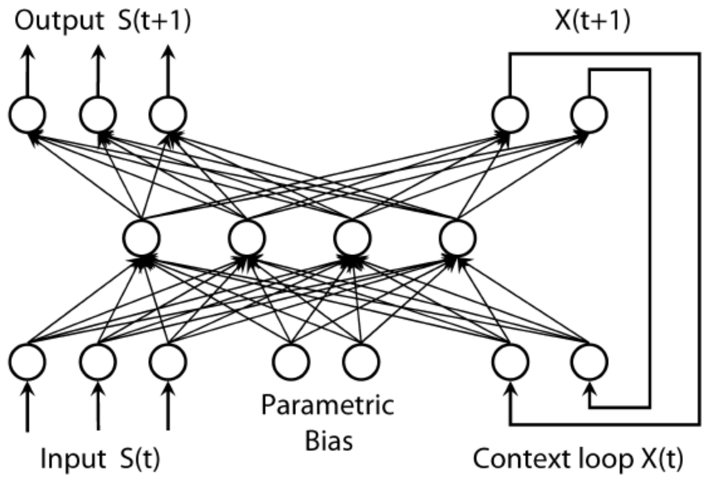

Figure 4: A RNN with a Parametric Bias layer (Source: [9])

Apart from an input layer, a hidden layer, and an output layer, this network has two small layers whose recurrent connections form a "context loop," which enables the network to keep information about past activations of the hidden layer in the network. The difference to a standard recurrent neural network, though, is the Parametric Bias (PB) layer, which consists of two neurons.

RNNPB are trained using the Backpropagation Through Time method described in section 2.2. However, the weights of the outgoing connections of the PB layer are held fixed. Instead, their internal values are adjusted. In order to compute their activations, the activation function is applied to their internal values. While there are various ways to compute the update of the internal values, the method used by fnnlib is a simple one adopted from Cuijpers et al. [1]:

$$u_j^k(e+1) = u_j^k(e) + \eta \sum_{n=1}^{N} \delta_j^k(n) \tag{10}$$

In the above equation, $u_j^k(e+1)$ denotes the internal value of PB neuron j for pattern k during the $(e+1)^{th}$ training epoch. η and $\delta_j^k(n)$ denote the learning rate and the gradient of the PB neuron, respectively. Note that the gradients are computed separately for the different time-series to be learned.
One training epoch consists of N steps, which means that the weights and the PB values are updated after N datasets of each pattern were presented to the network. The formula BPTT uses for updating the weights has to be adjusted slightly in order to account for the fact that there are now several time-series:

$$\Delta w_{ji} = \eta \frac{1}{K} \sum_{k=1}^{K} \sum_{n=1}^{N} \delta_j^k(n) y_i^k(n') \tag{11}$$

K denotes the number of time-series to be learned. Since the gradients were computed individually for each time-series using standard BPTT, the index k is needed to distinguish between the different gradients and activations of the same neuron in different time-series.

As mentioned above, the output of a PB neuron at step n is computed using

$$y_j^k(n) = f(u_j^k(n)), \tag{12}$$

where f is the activation function.

According to Ito and Tani [3], an RNNPB is better at learning multiple patterns because it is capable of "extracting relational structures" that are shared by the patterns. However, they also concede that the network may fail to do so if the relational structures are complex. Tests using fnnlib's RNNPB algorithm have suggested that the network's performance after training depends greatly on the kinds of patterns that were learned simultaneously, as well as on the number of PB neurons and their learning rate. One of the reasons I included RNNPB in fnnlib was that students at Osaka University's Graduate School of Engineering had successfully used them in basic tasks of speech synthesis.

2.4 Continuous-Time Recurrent Neural Networks

Continuous-Time Recurrent Neural Networks (CTRNN) are an extension of traditional RNN that have one more way of retaining information about past network states. While this information is kept in a standard RNN through the use of recurrent connections only, the neurons in a CTRNN have additional internal states whose values change gradually.

When a neuron in a CTRNN is supposed to fire at time-step n, its induced local field is computed. Instead of applying the activation function to the induced local field, as is done in standard RNN, a new internal state is computed based on the internal state at time-step $n-1$ and the current induced local field. The activation function is then applied to the value of the new internal state. A time constant τ is used to control how fast the internal state u changes, as can be seen in the update formula:

$$u_j(n+1) = \left(1 - \frac{1}{\tau}\right) u_j(n) + \frac{1}{\tau}\left(\sum_i w_{ji} y_i(n)\right) \tag{13}$$

$u_j(n)$ denotes the internal state of neuron j at time-step n. It is multiplied by $\left(1 - \frac{1}{\tau}\right)$, which means that this part of the sum will be zero if τ has a value of 1. For large τ, the product will be virtually equal to $u_j(n)$. In the second part of the sum, the induced local field is multiplied by $\frac{1}{\tau}$. As a result, the value of this product decreases as τ increases. If τ is 1, the new internal state $u_j(n+1)$ will depend only on the induced local field at time-step $n+1$, i.e. the neuron will behave like a standard neuron in a traditional RNN. If τ is large, on the other hand, the new state will depend almost entirely on the old internal state.

The neuron's activation at $n+1$ is simply computed by

$$y_j(n+1) = f(u_j(n+1)) \tag{14}$$

Expanding on this concept, Yamashita and Tani [8] developed Multiple-Timescale Recurrent Neural Networks (MTRNN), which are CTRNN that use different time constants for different layers. According to [8], their work was motivated by neurological research suggesting that human beings learn to perform simple movements (”motor primitives”) which are then combined to perform more complex movements. For example, the action of grasping a cup could be considered a motor primitive that is used both in drinking from a cup and in washing a cup. They tried to replicate this behavior in an MTRNN by using two fast-changing layers (time constants 1 and 5) to learn motor primitives, and a slowly changing layer (time constant 70) that was supposed to learn to combine the motor primitives of the two other layers in order to perform complex tasks. In the ideal case, it would be possible to fix the weights of the two fast-changing layers once a certain number of motor primitives have been acquired. Then, new complex movements would be learned simply by teaching the slow layer new ways of combining the motor primitives.

Figure 5 shows the network architecture used by Yamashita and Tani [8]. The input-output layer (100 neurons) and the fast context layer (60 neurons) are the aforementioned fast-changing layers with time-constants 1 and 5, respectively, while the slow context layer (20 neurons) has a time constant of 70. The input-output layer is only connected to the fast context layer. The fast context layer is connected to all layers, including itself. The slow context layer is connected to the fast context layer and itself.

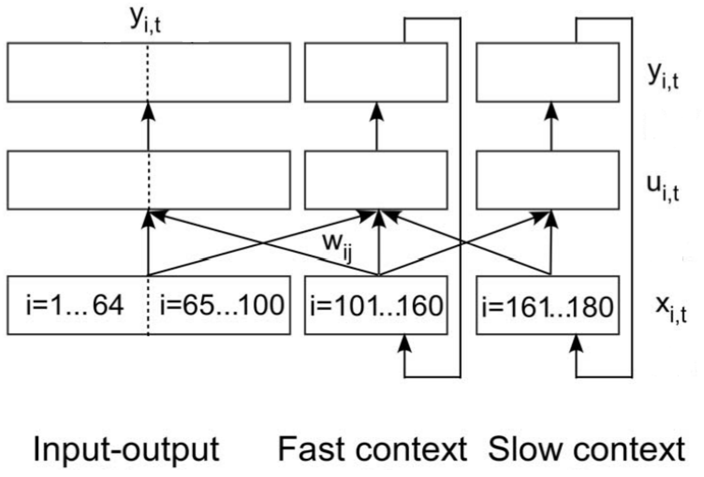

Figure 5: A Multiple-Timescale RNN (Source: [8])

Since Yamashita and Tani [8] chose the Kullback-Leibler Divergence as the error function to be minimized, the weight update formulas are different from the ones used in BPTT. (fnnlib uses the formulas in [8] to train CTRNN.) The overall error is given by

$$E = \sum_{n=1}^{N} \sum_{i \in O} d_i(n) log \left(\frac{d_i(n)}{y_i(n)} \right), \tag{15}$$

where N is the number of time-steps in the current training epoch, O is the set of indices referring to neurons in the output layer, $d_i(n)$ is the desired output of neuron i at step n, and $y_i(n)$ is its actual output. The gradients are computed using

$$\delta_i(n) = \begin{cases} y_i(n) - d_i(n) + (1 - \frac{1}{\tau_i})\delta_i(n+1); \text{ for i} \in \text{O} \\ \sum_j \delta_j(n+1) \left[\Delta_{ij}(1 - \frac{1}{\tau_i}) + \frac{1}{\tau_j} w_{ji} f'(u_i(n)) \right]; \text{ for i} \notin \text{O} \end{cases} \tag{16}$$

where Δ_{ij} is Kronecker's Delta, which is 1 if i=j and 0 otherwise. Note that Δ was chosen to denote this function simply because the commonly used δ already refers to the gradient. After the gradients have been computed, the weight updates Δw_{ij} are computed by the following formula:

$$\Delta w_{ji} = -\eta \sum_{n=1}^{N} \frac{1}{\tau_j} \delta_j(n) y_i(n-1) \tag{17}$$

While Yamashita's and Tani's [8] robot experiments involving the network in Figure 5 yielded promising results, they point out that more research needs to be conducted in order to determine whether this method of training also works with larger networks that have to perform truly complex tasks. As CTRNN may be used for robot learning at Osaka University in the future, they were included in fnnlib.

2.5 Echo State Networks

Developed by H. Jaeger [4], Echo State Networks (ESN) are a special type of recurrent neural network which, unlike the RNN presented so far, is not trained using gradient-based methods. In the most simple case, an ESN consists of an input layer, a large "reservoir" layer, and an output layer. The input layer is fully connected to the reservoir (i.e. each input neuron has outgoing connections to all reservoir neurons), with the reservoir being fully connected to the output layer. Furthermore, there are random recurrent connections between the reservoir neurons. An illustration of an ESN can be seen in Figure 6, which is followed by further explanatory remarks. While the reservoir typically contains at least 100 neurons, this example only has four reservoir neurons to keep the explanations simple.

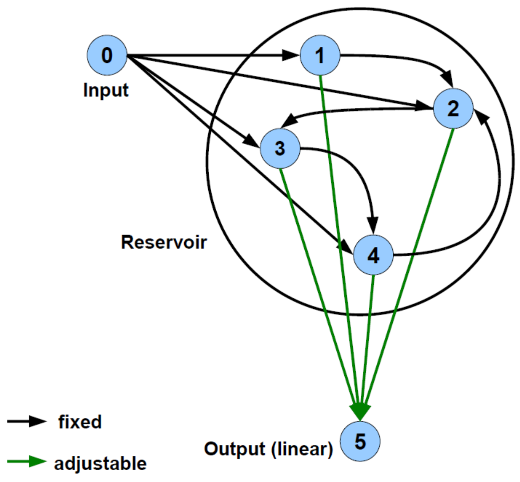

Figure 6: A simple Echo State Network

As can be seen in Figure 6, both the input-to-reservoir and the reservoir-to-reservoir connections will be held fixed during training. Only the reservoir-to-output weights will be adjusted. Also, the output unit does not have a non-linear activation function; instead, the output is identical to the unit's induced local field, i.e. the sum of the products of connection inputs and connection weights. The input and reservoir units may have non-linear activation functions, though.

In order to train an ESN, the network is first run for a certain number of time-steps in order to wash out the initial reservoir state. This is done since, at time-step 1, the value transmitted by the recurrent reservoir connections is chosen to be either zero or a small random value. Note that during the entire training phase, including this initial washout phase, only the activations of the input and reservoir layers need to be computed, whereas the potential activations of the output layer can be ignored.

After n sets of input data were presented to the network, the next m activations of the reservoir units (1 through 4 in this example) are collected into a $m \times 4$ state matrix S, where the number of columns is equal to the number of reservoir units:

$$S = \begin{pmatrix} y_1(n+1) & y_2(n+1) & y_3(n+1) & y_4(n+1) \\ y_1(n+2) & y_2(n+2) & y_3(n+2) & y_4(n+2) \\ ... & ... & ... & ... \\ y_1(n+m) & y_2(n+m) & y_3(n+m) & y_4(n+m) \end{pmatrix} \tag{18}$$

At the same time, the desired output is collected into a $m \times 1$ matrix D, whose number of columns corresponds to the number of output units.

$$D = \begin{pmatrix} d_5(n+1) \\ d_5(n+2) \\ ... \\ d_5(n+m) \end{pmatrix} \tag{19}$$

Since both the input-to-reservoir and the reservoir-to-reservoir weights remain fixed, we can be sure that the reservoir's activations at time-steps $n+1$ through $n+m$ will be the same when the network is reset and presented with the same input data again. In order to make sure that the network will produce the correct output, we first set up a system of linear equations, with the connection weights $w_{j,i}$ being the unknown variables. Note that i denotes a reservoir unit (1 through 4 in this case), while j (which is always equal to 5, since our example only has one output unit) denotes an output unit to which i is connected.

$$\begin{pmatrix} y_1(n+1)w_{5,1} + ... + y_4(n+1)w_{5,4} = d_5(n+1) \\ y_1(n+2)w_{5,1} + ... + y_4(n+2)w_{5,4} = d_5(n+2) \\ ... \\ y_1(n+m)w_{5,1} + ... + y_4(n+m)w_{5,4} = d_5(n+m) \end{pmatrix} \tag{20}$$

The 4×1 weight matrix (a vector, in this case) containing the values of $w_{5,1}$ through $w_{5,4}$ that represent a solution to this linear system can then be computed using the Wiener-Hopf method:

$$w_{out} = ((S'S)^{-1}S'D)', \tag{21}$$

where $(\cdot)'$ is the transpose operator and $(\cdot)^{-1}$ is the matrix inverse operator. Alternatively, we can use the pseudoinverse method, which is slower but may still find solutions when Wiener-Hopf fails.

$$w_{out} = (S^{\dagger}D)' \tag{22}$$

Here, $S^\dagger$ denotes the pseudoinverse of S.

As evidenced by tests conducted with fnnlib, training an ESN usually is much faster than training a standard RNN using BPTT, which may need many iterations to converge or may not converge at all in some cases. The performance of the latter algorithm also depends strongly on the size of the training epoch, the learning rate, and the number of training epochs chosen by the trainer. Another issue which does not occur with ESN is the "vanishing gradients problem" ([2], p. 819): sometimes, the output of an RNN at a certain time-step may depend on an input in the distant past. In that case, adjusting the training parameters in such a way as to enable the network to detect changes in that input can be very hard. With ESN, it tends to be easier to determine appropriate values for the few parameters that can be controlled (e.g. duration of the washout phase, number m of rows in state matrix).

As was the case with RNNPB and CTRNN, the inclusion of ESN in fnnlib was also partly motivated by the fact that research on this type of neural network is being done at Osaka University's Graduate School of Engineering. While many members of the school - and, possibly, other researchers around the globe - had never actually used ESN, I believe that they may be more inclined to give them a try in their research projects with a library like fnnlib at their disposal.

3 Design of the Library

This section will present fnnlib's design, i.e. its classes and their interrelationships. Moreover, it will give reasons as to why certain design decisions were made and present ideas for future changes or additions. Note that detailed information about how to use the classes can be found in fnnlib's documentation (see Appendix A); the purpose of this section is to give an abstract overview of the classes rather than going through all methods and explaining exactly what each parameter does.

First, we will have a look at the class *DataSource* and its child classes, which are responsible for providing a neural network with input and desired-output data (see 3.1). In section 3.2, the activation function class *ActivationFunction* and its child classes will be discussed. Section 3.3 covers *Initialization* and its children, i.e. classes that initialize connections between layers, or between a neuron and a layer. Section 3.4 will introduce the *Adaptation* class, which can be used to adapt ESN reservoir layers prior to training. Section 3.5 will cover the training algorithms for ESN, starting with the base class *ESNTrainingAlgorithm*. Unlike the gradient-based methods, these algotihms need to be covered before the layer classes *Layer* and its child class *ESNLayer* are introduced in section 3.6. Section 3.7 provides a brief summary and explains the relationships between the classes covered up to that point. Next, section 3.8 will introduce the class *NeuralNetwork* and its child classes. Only after that section will the base class for gradient-based training algorithms (*RNNTrainingAlgorithm*) and its children be covered in section 3.9. Finally, section 3.10 will explain how the classes *ErrorData* and *NetworkErrorData* fit into the overall design and allow the trainer to collect error data during the testing phase.

Throughout this section, there will be code listings showing the declarations of various methods. While the actual declarations in fnnlib's header files contain no parameter names, the declarations in the code listings do. The names are the same as those that appear in the definitions and were added for clarity. Also, if several methods of one class have the same visibility but are presented in separate code listings, their visibility modifier (*public*, in most cases) will be inserted into all code listings even though it only appears once in the actual source code.

3.1 Classes for Input/Desired-Output Data

In fnnlib, input/desired-output data is represented by the class *DataSource*, an abstract base class from which *StaticDataSource* is derived. As can be seen in Figure 7, *StaticDataSource* also has a child class, namely *FileDataSource*. So what are the differences between these classes?

3.1.1 DataSource

DataSource is an abstract base class for both static and dynamic data sources. While static data sources can be created using *StaticDataSource* or a class derived from it, dynamic data sources can be derived directly from *DataSource*. "Dynamic" data in this case means data generated on the fly. Even though a data source may generate the same data every time it used, it is also possible to generate data that depends on the system clock, for example. In order to understand this better, we will first look at an excerpt from the declaration of *DataSource*.

```
public:
  DataSource (int nNumberOfSets, int nSetSize);
  inline int GetSetSize ();
  inline int GetNumberOfSets ();
  virtual void GetSetAt (int nIndex, vec* y) = 0;

protected:
  int num_sets;
  int set_size;
```

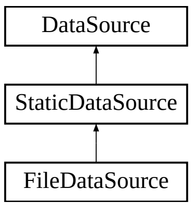

Figure 7: DataSource and its child classes

The constructor only needs to know the number of sets as well as the size of each set. These values are stored in *num_sets* and *set_size*, which can be accessed using the two *Get* methods. Implementations of *GetSetAt* must write the data set at time-step *nIndex* into the vector y. Now suppose the user has created a neural network and, as a first test, he or she want to see how well it learns to generate a sine wave represented by 1000 data points. There would be no need to generate those data points and store them in files; instead, it would be possible to create two child classes of *DataSource* (say *SineIn* and *SineOut*). In each of these classes, only *GetSetAt* would have to be defined. Since *SineIn* represents the input, its *GetSetAt* method would simply write $sin((nIndex \bmod 1000) \times 2\pi/1000)$ into y. *SineOut* represents the desired output at the same time-step, so its *GetSetAt* method would write the value to be generated by the network into y, i.e. $sin(((nIndex + 1) \bmod 1000) \times 2\pi/1000)$.

Of course, *GetSetAt* could be arbitrarily complex, performing tasks such as transforming live image data generated by a camera into a vector and writing it into y. In order to allow this kind of flexibility, the base class *DataSource* was left as general as it is.

3.1.2 StaticDataSource

In addition to the attributes of its parent class *DataSource*, *StaticDataSource* contains an array of vectors that holds the data associated with the respective object. The array's elements can be accessed using the *GetSetAt* method, which is no longer pure virtual, meaning that it is possible to create objects of type *StaticDataSource*.

```
public:
  void GetSetAt (int nIndex, vec* y);
protected:
  vec* data;
```

The memory for *data* is allocated by the constructor, with the array having *num_sets* fields and each vector having dimension *set_size*. Note that all vectors will be zero vectors. If a static data source containing meaningful data is needed, it can be derived from this class, but it will not be necessary to define a *GetSetAt* method anymore. One important thing to be aware of is that *GetSetAt* as defined in *StaticDataSource* does not perform any bounds checking. Instead, *nIndex* will be modulo divided by the total number of sets. If its value is negative, the result of the modulo division will be added to the total number of sets. This way, any value of *nIndex* will be transformed into a valid index. The decision to do this was made because it would save a lot of ugly *if* statements later on, thus making the code more readable.

3.1.3 FileDataSource

A child class of *StaticDataSource*, *FileDataSource* was designed as the data source to be used in the majority of cases. The only difference between this class and its parent class is a new constructor that expects a path to a file as its third argument.

```
public:
  FileDataSource (int nNumberOfSets, int nSetSize,
                  const char* strFileName);
```

The constructor will open the file, store its contents in the *data* attribute inherited from *StaticDataSource*, and close the file afterwards. Currently, *FileDataSource* can only process very simple text files that consist of *nNumberOfSets* × *nSetSize* floating point numbers separated by whitespace characters only. If the file contains fewer numbers, the constructor will read them all and interpret non-existent numbers as zeros. If the file contains more numbers than indicated by the first two arguments to the constructor, they will be ignored.

Obviously, this is one place where fnnlib can and should be extended in the future in order to enable it to process e.g. CSV files or files whose sets of numbers are interrupted by comments. Support for those types of files has not been included yet because it is reasonably easy to convert them into files compatible with *FileDataSource*.

3.2 Classes Representing Activation Functions

During the planning phase for fnnlib, it was decided that the user should be able to assign activation functions to neurons on a layer-by-layer basis. Having to choose one activation function for an entire network would have been too restrictive, especially in the case of an ESN, whose output layer cannot have a non-linear activation function. This would have meant that the entire network would have had to use this non-linear activation function in all layers. While it would also have been possible to let the user assign activation functions individually to each neuron, this would have made the library considerably more complex while offering little benefit in practical applications. After all, it is always possible to put neurons that are supposed to have different activation functions into separate layers and assign the appropriate activation function to the respective layer.

3.2.1 ActivationFunction

All activation functions in fnnlib are represented by classes derived from the abstract base class *ActivationFunction*, which contains the two pure virtual methods *ComputeActivations* and *ComputeDerivatives*.

```
public:
  virtual void ComputeActivations (const vec* x, vec* y) = 0;
  virtual void ComputeDerivatives (const vec* x, vec* y) = 0;
```

In non-abstract child classes, *ComputeActivations* will be called to compute the activations of a layer based on the induced local fields in x and write the results of the computation into y. *ComputeDerivatives* will be used to compute the values of the derivative of the activation function at the points indicated by x. In order to add a new activation function to fnnlib, all that needs to be done is to derive a class from *ActivationFunction* that implements the two pure virtual methods. fnnlib already contains several built-in activation functions, though; the UML diagram in Figure 8 shows the classes representing them.

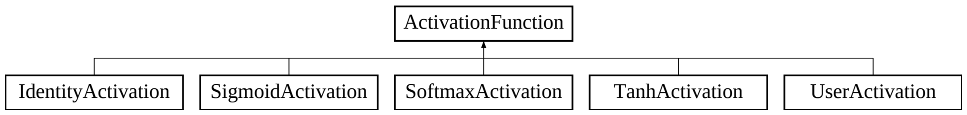

Figure 8: ActivationFunction and its Child Classes

Before introducing the various child classes, it should be pointed out that the user will usually only have to create objects of type *UserActivation*. *ActivationFunction* already has *public static* attributes of type *IdentityActivation**, *SigmoidActivation**, *SoftmaxActivation**, and *TanhActivation** that can be passed to any function needing an object of one of those classes.

```
public:
  static SigmoidActivation* SIGMOID;
  static TanhActivation* TANH;
  static SoftmaxActivation* SOFTMAX;
  static IdentityActivation* IDENTITY;
```

The following subsections will briefly explain the five child classes of *ActivationFunction*.

3.2.2 IdentityActivation

This is the most simple activation function contained in fnnlib. It just copies the layer's induced local fields into its vector of activations, i.e. it is the identity function. The value of the derivative will always be 1.

3.2.3 SigmoidActivation

Objects of this class compute activations and values of the derivative according to the formula of the sigmoid function, whose argument u will be the induced local field of the respective neuron:

$$y_i = f(u_i) = \frac{1}{1+e^{-u_i}} \tag{23}$$

$$y_i' = f'(u_i) = (1 - f(u_i)) \times f(u_i) \tag{24}$$

What is special about the sigmoid activation function is that it can be adapted using Intrinsic Plasticity (IP) adaptation [7], in which case two parameters are added:

$$y_i = \frac{1}{1 + e^{-a_i u_i - b_i}} \tag{25}$$

The a_i and b_i parameters are set for each neuron individually by the *Adaptation* class. Apart from the inherited *ComputeActivation* method that expected two vectors as arguments, *SigmoidActivation* has another *ComputeActivation* method that receives a pointer to an *Activation* object containing the parameters.

```
public:
  void ComputeActivations (const vec* x, Adaptation* adapt,
                           vec* y);
```

Note that there is no special formula for the derivative of the adapted sigmoid function; this is because adaptation can only be performed on ESN layers, but ESN do not require computation of the derivatives.

More information on the *Adaptation* class, IP adaptation in general and why it is useful can be found in section 3.4.

3.2.4 SoftmaxActivation

This activation function was adopted from Yamashita and Tani [8], who used it for the output layer of their CTRNN. Given the induced local fields u of a layer, the activations y of the layer's neurons will be computed by

$$y_i = \frac{e^{u_i}}{\sum_j e^{u_j}} \tag{26}$$

where $\sum_j$ iterates over all neurons in the layer. The value of the partial derivative w.r.t. u_i is computed in two steps.

$$p_i = \sum_j e^{u_j} - e^{u_i} \tag{27}$$

$$y'_i = \frac{p_i \times e^{u_i}}{(e^{u_i} + p_i)^2} \tag{28}$$

3.2.5 TanhActivation

This class represents the hyperbolic tangent activation function and uses the C++ standard library's *tanh* function to compute a layer's activations. The values of the derivative are computed using

$$y_i' = tanh'(u_i) = 1 - tanh(u_i)^2 \tag{29}$$

3.2.6 UserActivation

The *UserActivation* provides a way of adding a custom activation function to fnnlib without having to derive a new class from *ActivationFunction*. Its constructor takes as arguments two pointers to functions that map a *double* value to a *double* value; the functions represent an activation function and its first derivative, respectively.

```
public:
  UserActivation (double (*UserDefActFunc) (double),
                  double (*DerUserDefActFunc) (double));
```

UserDefActFunc is short for "user-defined activation function," while *DerUserDefActFunc* stands for "detivative of user-defiend activation function." These function pointers are stored inside the object. When *ComputeActivations* or *ComputeDerivatives* are called, they will iterate over the elements of x (the vector of induced local fields) and pass the values to *UserDefActFunc* or *DerUserDefActFunc*, storing the results in the vector y.

This alternative way of adding an activation function was included because it might be faster than creating a new class, especially if the functions to be passed to the constructor have already been defined. It also saves the user the trouble of looking up how to use Armadillo's *vec* type. On the other hand, this method only works with activation functions that compute a neuron's activation independently of the induced local fields of the other neurons in the same layer. If the activation of neuron j at time-step n does depend on the induced local fields of its neighbors, access to the vector containing those values is needed. In that case, creating a new child class of *ActivationFunction* and implementing the two *Compute** methods is the way to go since all of a layer's induced local fields will be accessible inside those methods.

3.3 Classes for Weight Initialization

As long as a neural network is very small or only sparsely connected, the user may want to set each connection weight manually. However, there are cases where the number of connections calls for a different way of initializing the weights. This is especially true for ESN with their large reservoir layers. As a result, a class representing initialization functions was included in fnnlib, namely *Initialization*.

3.3.1 Initialization

An abstract base class, *Initialization*'s most important methods are the virtual method *Initialize* and the static method *GenerateRandomWeight*.

```
public:
  virtual void Initialize (mat* w) = 0;
  static inline double GenerateRandomWeight (double min,
                                             double max);
```

Non-abstract child classes of *Initialization* will have to implement an *Initialize* method that fills the weight matrix w. The entry w_{ij} in the i^{th} row and the j^{th} column represents the weight of the connection from neuron i to neuron j. The dimensions of w depend on what kind of connections are being initialized. If they connect a neuron to a layer L, w will be a $1 \times size(L)$ matrix; if they connect a layer L_1 to a layer L_2 (note that L_1 and L_2 may be the same layer), w's dimensions will be $size(L_1) \times size(L_2)$.

The method *GenerateRandomWeight* was included so as to provide the user with a fast way of generating random numbers x within a certain range (min $\leq$ x $\leq$ max). It might come in handy when new child classes of *Initialization* that work with random weights are added.

fnnlib already comes with several built-in initialization methods. Figure 9 illustrates the relationships between the classes representing them and the base class *Initialization*.

The following sections will briefly explain the built-in activation methods and give reasons as to why they were included. For information on how

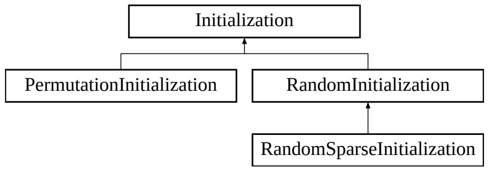

Figure 9: Initialization and its child classes

to use them, please consult the documentation in Appendix A. Note that this section only introduces the classes that can be used to initialize connection weights, but it does not explain where the weight matrix they work on even comes from. This will become clear in sections 3.6 and 3.8, though.

3.3.2 RandomInitialization

Experience has shown that a good way of initializing the connections between a neuron and a layer or between two layers in a standard RNN is to assign small random weights to them. Hence why the class *RandomInitialization*, whose constructor expects two *double* values *fMin* and *fMax* as arguments, was added. When the class's *Initialize* method is called, it simply writes a random value between *fMin* and *fMax* into each entry of the weight matrix.

3.3.3 RandomSparseInitialization

This class is derived from *RandomInitialization*, and it works in a similar way. In addition to *fMin* and *fMax*, its constructor expects a *double* argument *fSparseness*. It must have a value between 0 and 1, specifying what percentage of connections should be assigned a weight other than zero. If *fSparseness* is set to 0.75, for example, this means that roughly 75% of the entries in the weight matrix will have a random value between *fMin* and *fMax* assigned to them, whereas the rest will be set to zero. This method is especially useful for initializing the internal connections of an ESN's reservoir layer.

3.3.4 PermutationInitialization

Like *RandomSparseInitialization*, this class may prove useful when working with standard RNN, but is primarily intended for use with ESN and the internal connections of their reservoirs. Its *Initialize* method first transforms the weight matrix into an identity matrix. (This is one of the reasons why it probably only makes sense to use this class to initialize the internal connections of a layer: In that case, the weight matrix will be square, resulting in a proper identity matrix; if the connections between two layers of different sizes were to be initialized, there would be some zero rows (or colums) left after filling the diagonal with ones.) The columns of the weight matrix are then shuffled. Finally, the 1s are replaced by either a constant value or a random value, depending on which constructor was used (again, please see the documentation for details). Provided that the weight matrix was square, this will result in a matrix where each row i has only one non-zero value. This in turn means that neuron i will only have one outgoing connection with a non-zero weight. While this method of initialization may seem unintuitive, it tends to yield good results when used for ESN reservoirs, as evidenced by the practical example in section 4.

3.4 The Adaptation Class

The *Adaptation* class offers the ability to perform Intrinsic Plasticity adaptation [7] on ESN reservoirs that use the sigmoid activation function (class *SigmoidActivation*). IP adaptation is performed prior to training, but during the adaptation phase, the network is presented with the same input as during training. In short, this method adjusts the a and b parameters of the sigmoid activation function (see section 3.2) until the neurons' output distributions form exponential distributions. This has been found to be a way of increasing the information transmission inside the reservoir [7]. For more information on the theory behind this method, please see Steil's paper [7] on the topic, as an in-depth introduction would go beyond the scope of this thesis.

In order to understand how this class works, it is necessary to take a brief look at some of its methods.

```
public:
  Adaptation (double fMu, double fLearningRate);
  inline double GetA (int nUnit);
  inline double GetB (int nUnit);
  void Adapt (vec* x, vec* y);
```

The constructor expects two arguments that control the adaptation process. *GetA* and *GetB* return the adjusted a and b parameters of the neuron that has index *nUnit* inside the layer to which the *Adaptation* object is assigned. These two methods will be called by *SigmoidActivation*'s extended *ComputeActivations* method (see section 3.2). Finally, *Adapt* performs the actual adaptation of the a and b parameters based on a vector x of induced local fields and a vector y containing the layer's activations. How and when this method is invoked will become clear when the classes *ESNLayer* and *ESN* are covered in sections 3.6 and 3.8, respectively.

3.5 Classes Representing ESN Training Methods

ESNTrainingAlgorithm and its child classes are the last in a series of mostly unrelated classes that need to be covered before the classes that combine them all to form an actual neural network. As the name suggests, *ESNTrainingAlgorithm* is an abstract base class for ESN training methods, and ESN training methods only. The decision to separate them completely from the gradient-based methods with base class *RNNTrainingAlgorithm* was made because ESN training methods can be assigned on a layer-by-layer basis whereas a single gradient-based algorithm has to be chosen for the entire network when working with other RNN. This is also why *RNNTrainingAlgorithm* will be covered *after* the introduction of the layer classes in section 3.6. Before the various classes in this section are presented, Figure 10 shows how they are related.

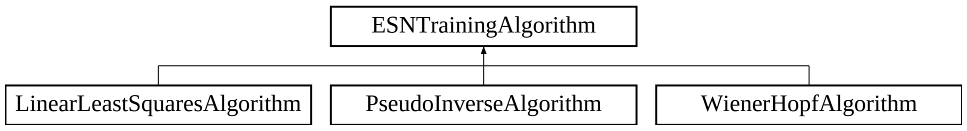

Figure 10: ESNTrainingAlgorithm and its Child Classes

We will first have a look at *ESNTrainingAlgorithm*.

3.5.1 ESNTrainingAlgorithm

The most important parts of this class's declaration can be found in the code listing below.

```
public:
  virtual mat ComputeWeightMatrix (mat* matStates,
                                   mat* matDesiredOutput) = 0;

  static PseudoInverseAlgorithm* PSEUDOINV;
  static WienerHopfAlgorithm* WIENERHOPF;
```

```
    static LinearLeastSquaresAlgorithm* SOLVE;
```

All that is required of its subclasses is that they implement the pure virtual method *ComputeWeightMatrix*. The first argument to this method is the matrix of activations that were collected over a certain number of time-steps. To be more specific, it will only contain the activations of those neurons that are connected to the layer to which the *ESNTrainingAlgorithm* object was assigned. The second argument is the matrix containing the desired output values for each neuron in the layer at each time-step. Note that the matrices *matStates* and *matDesiredOutput* were denoted by S and D respectively in section 2.5. Based on these two matrices, implementations of *ComputeWeightMatrix* must return a weight matrix containing new values for the weights of *incoming* connections to the layer the object is assigned to.

As can be seen in the above code listing, static pointers to existing objects of the three child classes were included in *ESNTrainingAlgorithm*. Consequently, there is usually no need for the user to create objects of those classes manually, since one object can be assigned to more than one layer if needed. This reuse of objects is possible because the classes do not have any attributes with object-specific values on which the result of *ComputeWeightMatrix* might depend. The returned weight matrix will only depend on the method's two arguments.

3.5.2 PseudoInverseAlgorithm

The child class *PseudoInverseAlgorithm* uses the pseudoinverse method to compute the weight matrix (see section 2.5). Apart from actually implementing the *ComputeWeightMatrix* method, no further functionality is added compared to the parent class. The same holds true for the other two child classes, which is why they will be covered very briefly.

3.5.3 WienerHopfAlgorithm

WienerHopfAlgorithm uses the Wiener-Hopf method to compute the weight matrix (see section 2.5).

3.5.4 LinearLeastSquaresAlgorithm

LinearLeastSquaresAlgorithm simply invokes Armadillo's *solve* method to compute a solution to the system of equations formed by the matrices. While no statistical analysis of the performance of these methods in fnnlib has been done, *LinearLeastSquaresAlgorithm* seems to offer the best combination of speed and numerical stability.

3.6 The Layer Classes

During the planning phase for fnnlib, one of the questions that came up was how a neural network should be represented by classes, and which of its components should be represented by classes of their own. One solution would have been to create a neuron class, a layer class, and a neural network class, with layer objects being comprised of neuron objects and neural network objects holding an array of layer objects. This would have provided an easy way of adjusting the settings of each neuron individually, but with large networks, handling hundreds of neuron objects would have become infeasible.

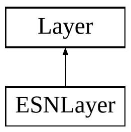

Figure 11: Layer and ESNLayer

Another possibility would have been to not have separate classes for layers and neurons but to just create one class that represents a neural network and all of its components. However, since the user should be able to assign things such as activation functions and initialization methods on a layer-by-layer basis, this one neural network class would have been overloaded with *Set* methods that would change the attributes of the various layers. While the user would have had to deal with one object only, that object would have been very complex.

In the end, a compromise between the above-mentioned approaches was made. Neurons would not be represented by a separate class, but there would be a layer class whose objects would be passed to a neural network class. This section deals with the layer class *Layer* and its child class *ESNLayer* (also shown in Figure 11), whereas the base class for actual networks (*NeuralNetwork*) will be covered in section 3.8.

3.6.1 Layer

The base class *Layer* represents layers of the gradient-based types of RNN (standard RNN, RNNPB, and CTRNN). The code listing below shows the declaration of those methods that are important for the purpose of understanding fnnlib's design. As always, the complete documentation can be found in Appendix A.

```
public:
  Layer (int nUnits, ActivationFunction* actFunc = NULL,
         bool bIsPbLayer = false, bool bIsConstPb = false,
         double fTau = 1.0, vec* fDefStates = NULL);

  virtual inline void ComputeActivations (vec* x, vec* y);
  inline void ComputeDerivatives (vec* x, vec* y);
```

In the case of the constructor, default values are included for all arguments except one in order to save the user the trouble of assigning all those values manually each time a *Layer* object is created. Even though more information about the *Layer* class, including its complete code, can be found in the documentation, we should have a brief look at the arguments of the constructor in order to understand what kind of information is stored in the objects. *nUnits* specifies the number of neurons in the layer. *actFunc* is the activation function to be used, with *IdentityActivation* being chosen if the default value NULL is used. If *bIsPbLayer* is *true*, the layer will be treated as a PB layer, with *bIsConstPb* specifying whether the internal PB values should be constant or trainable. *fTau* is the time constant, which should only be set to a value other than 1.0 if a continuous-time learning algorithm is used (see section 2.4). *fDefStates* is a vector containing the default internal values of the neurons, with NULL being interpreted as the zero vector.

Apart from the constructor, we also see the methods *ComputeActivations* and *ComputeDerivatives* in the above code listing. They simply invoke the identically named methods of the *ActivationFunction* object *actFunc* points to, passing x and y to them. The two methods were included in this class in order to provide access to a layer's activation function without having to retrieve a pointer to the actual *ActivationFunction* object.

3.6.2 ESNLayer

As can be seen in Figure 11, *ESNLayer* is derived from *Layer*, which means that the layers of an ESN offer all of the functionality of standard RNN layers plus some additional features. Once again, we will first examine the class's most important methods.

```
public:
  ESNLayer (int nUnits, ActivationFunction* actFunc = NULL,
            ESNTrainingAlgorithm* trainingAlgorithm = NULL,
            Initialization* initFunc = NULL,
            mat* weights = NULL,
            Adaptation* adapt = NULL,
            bool bIsPbESNLayer = false,
            double fTau = 1.0, vec* fDefStates = NULL);

  inline void Adapt (vec* x, vec* y);
  inline mat ComputeWeightMatrix (mat* s, mat* d);
  inline void ComputeActivations (vec* x, vec* y);
```

Unlike *Layer*'s constructor, this constructor accepts pointers to objects of type *ESNTrainingAlgorithm*, *Initialization*, and *Adaptation*. They will be used to train the layer's incoming connections, initialize its internal connections and adapt its activation function, respectively. The weights of the internal connections can also be set explicitly by passing a weight matrix *weights* to the constructor. However, if *initFunc* is a non-NULL pointer, the values in *weights* will be ignored. The other parameters serve the same purposes as they did in the *Layer* class. Note that the *Layer* constructor's Boolean argument *bIsConstPb* is missing; this is because fnnlib only allows constant PB neurons in ESN, so *bIsConstPb* will always be set to *true* when invoking the parent class's constructor. Remember that in other RNN, PB values are adjusted based on the gradient, which is not even computed in ESN. However, instead of disabling support for PB neurons altogether, the decision to at least include constant PB neurons was made. It is up to the user to find out whether they can help increase an ESN's performance by injecting pattern-specific values into the network in tasks such as predicting multiple time-series. This is in line with fnnlib's underlying philosophy of offering the maximum amount of flexibility even when it is not clear yet whether the features will actually offer any significant benefit.

The other three methods in the above code listing all act as interfaces to objects whose pointers were passed to the constructor. The *Adapt* method simply checks if the object actually has an adaptable activation function assigned to it and, if that is the case, passes the arguments to the *Adapt* method of the *Adaptation* object pointed to by the constructor argument *adapt*. Similarly, *ComputeWeightMa-*

trix and *ComputeActivations* delegate their tasks to the identically named methods of *ESNTrainingAlgorithm* and *ActivationFunction*, respectively. Note that pointers to objects of these two classes can be passed to the constructor. If they have NULL values, pointers to default objects will be used (see documentation). The *ComputeActivations* method of the parent class was overwritten because it only provides access to the unadapted sigmoid activation function. The new method checks whether a valid pointer to an *Adaptation* object was passed to the constructor and whether the *ActivationFunction* object is actually of type *SigmoidActivation.* If that is the case, IP adaptation is possible. The method will then pass a pointer to its *Adaptation* object to *SigmoidActivation*'s extended *ComputeActivations* method, which can access the adapted a and b parameters.

3.7 A First Summary

Before the remaining classes are covered, we should first recapitulate what has been said about the interrelationships between the various classes. To this end, let us first look at an UML diagram illustrating those aspects of fnnlib's design presented so far. The diagram will be followed by brief comments that summarize what was described in more detail in the preceding sections. For the sake of clarity, some child classes are simply subsumed by one symbolic class named "...", which is supposed to point out that there are further child classes that are not vital to understanding the overall design.

First, there is the *Layer* class and its child class *ESNLayer.* Each *Layer* has one *ActivationFunction* assigned to it, but one *ActivationFunction* can be assigned to multiple *Layers.* An *ESNLayer* always has an *ESNTrainingAlgorithm* assigned to it. (Note that the presence of child classes of *ESNTrainingAlgorithm* is indicated by the aforementioned "..." class.) It *may* have an *Initialization* and an *Adaptation* object assigned to it. If it does have an *Adaptation* object assigned to it, and if the *ActivationFunction* is of type *SigmoidActivation*, then the *Adaptation* object will serve to adapt the sigmoid activation function. An *Adaptation* object can only be assigned to one *ESNLayer* object, whereas an *Initialization* object may be assigned to multiple *ESNLayer* objects since it holds no layer-specific data.

After this brief recapitulation, we are finally ready to examine the classes representing actual neural networks, starting with the abstract

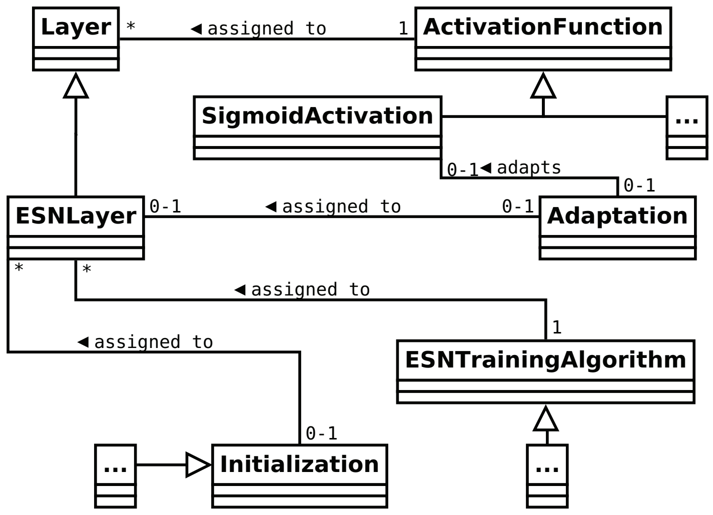

Figure 12: Interrelationships between the classes covered so far

base class *NeuralNetwork*.

3.8 Classes Representing Neural Networks

NeuralNetwork is an abstract base class for two types of networks, namely Recurrent Neural Networks with gradient-based learning algorithms (RNN[PB] & CTRNN) and Echo State Networks. The former are objects of the class *RNN*, while the latter are represented by the *ESN* class. Figure 13 illustrates the relationships between the three classes.

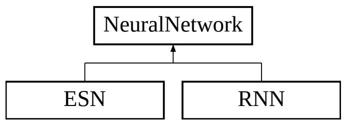

Figure 13: NeuralNetwork and its child classes

We will first look at the *NeuralNetwork* class before moving on to *RNN* and *ESN*.

3.8.1 NeuralNetwork

Even though this class is abstract, it already implements a lot of functionality required by its child classes and allocates memory for data structures that will be used when training or testing a network. Instead of cramming all of its important methods into one code listing, we will cover them one by one this time. The first method to examine is the constructor.

```
public:
  NeuralNetwork (int nLayers, Layer** layerArray,
                 int nSequences = 1);
```

Its first argument is the number of layers the network will be comprised of, followed by an array of pointers to the actual *Layer* objects. The third argument tells the network how many patterns it will have to learn simultaneously, with the default value being 1. The downside to this approach is that the user will have to create the *Layer** array manually, adjusting both its size and the *nLayers* argument accordingly when layers are added or removed. In the future, a new class (e.g. *LayerSet*) may be added to facilitate this process. It could, for example, provide a method called *AddLayer*, which would create new *Layer* objects, storing pointers to them and keeping track of the total number of objects created. That way, the user could simply pass a pointer to a *LayerSet* object to the constructor of a neural network class, with that class being able to retrieve the number of layers and the pointers to their objects using appropriate *Get* methods.

Next, we will see how the class *DataSource* is related to *NeuralNetwork* by looking at two methods that provide a network with input and desired-output data.

```
public:
  void SetInput (int nSeq, int nLayer,
                 DataSource* dataSource,
                 int nSrcLayer = -1,
                 int nStartCopyingAfter = 0);

  void SetOutput (int nSeq, int nLayer,
                  DataSource* dataSource,
                  bool bUseTeacherForcing);
```

Both methods assign a *DataSource* object to the layer with index *nLayer*. *nSeq* is the index of the sequence (= pattern) whose input/desired-output data is represented by *dataSource*. Since it was assumed that users would mostly create networks that are supposed to learn one sequence, there are identically named methods which lack the first parameter and simply call the corresponding extended method with *nSeq* set to zero. For more information on the other parameters, please consult the documentation in Appendix A. The purpose of this short introduction was just to point out the relationship between *DataSource* and *NeuralNetwork*.

Another aspect of fnnlib's design that should be explained is the *SetDefStates* method of *NeuralNetwork*.

```
public:
  void SetDefStates (int nSeq, int nLayer, vec* fValues);
```

It sets the default internal values for a specific layer in a specific data sequence to be learned. This may seem strange because section 3.6 demonstrated how default internal values can be passed to a layer's constructor. However, if the default values always had to be passed directly to *Layer* objects, working with those objects would have become inconvenient. If a network was supposed to learn multiple patterns, an entire array of vectors of default values would have to be passed to the *Layer* constructor. Hence, the constructor would also need to know the number of patterns to be learned by the network it will be part of; otherwise, it would not be able to determine the size of the array. Besides, the user may want to use different input/desired-output data during the training and testing phases. If *SetDefStates* did not exist, he or she would have to retrieve a pointer to the layer object and assign a new *DataSource* object to it. With *SetDefStates*, it is much easier to assign a new data source to a layer after training, for example. Nevertheless, the current approach is a little "dirty," and an alternative way of setting default internal states may be added in the future.

NeuralNetworks also offers several methods that connect the network's layers to each other. In this section, however, we will only examine the one which illustrats how fnnlib's classes interact. Its declaration looks as follows:

```
public:
  void ConnectLayerToLayer (int nSrcLayer, int nDstLayer,
                            Initialization* initFunc,
                            bool bTrainable);
```

ConnectLayerToLayer connects layer *nSrcLayer* to *nDstLayer* (both are zero-based indices), with *bTrainable* specifying whether the weights should be trained or not. The *Initialization* object pointed to by *initFunc* will be used to initialize the matrix representing the weights of the connections between the two layers. (Actually, the two indices may even refer to the same layer, which is how internal connections in RNN layers can be set up. While an ESN layer's internal connections can be initialized by an *Initialization* object passed to its constructor, this method provides another way of accomplishing the same task.)

There are two more methods that need to be mentioned before the child classes of *NeuralNetwork* are covered. The first of them is *Run*, which runs the network for a certain number of time-steps, collecting data that can be used for training or testing. This method is the same for RNN and ESN, i.e. it is not overwritten by either child class. The last important method is *CollectErrorData*, whose declaration can be found in the code listing below.

```
protected:
  void CollectErrorData (int nSeq, NetworkErrorData* ed);
```

This method will later be invoked by other methods of the classes *RNNTrainingAlgorithm* (see 3.9.1) and *ESN*. It stores error data collected during testing in an object of type *NetworkErrorData*, a class which will be covered in section 3.10.2. The latter section will also explain how *CollectErrorData* is invoked. Error data can be retrieved for each pattern individually if the network has learned multiple pattterns. In that case, *nSeq* will be the index of the pattern for which error data should be stored in *ed*.

3.8.2 RNN

RNN is a child class of *NeuralNetwork* that represents neural networks which use gradient-based learning algorithms. Compared to its parent class, *RNN*'s constructor has one additional parameter:

```
public:
  RNN (int nLayers, Layer** layerArray,
         RNNTrainingAlgorithm* rnnAlg,
         int nSequences = 1);
```

rnnAlg specifies the training algorithm to be used for this network. This object will be used to adjust the network's weights and minimize the error function when *RNN*'s *Train* method is invoked. *RNNTrainingAlgorithm* and its child classes will be covered in section 3.9.

Besides a *Train* method, *RNN* also offers a *Test* method which runs the network and collects error data using *NeuralNetwork*'s *CollectEr-*

rorData method. *Test* returns a pointer to a *NetworkErrorData* object (see section 3.10.2 for more information on this class).

Figure 14 is a UML class diagram illustrating how *RNN* is related to fnnlib's other classes. The symbolic classes called "..." once again indicate that there are child classes which are not displayed in the diagram.

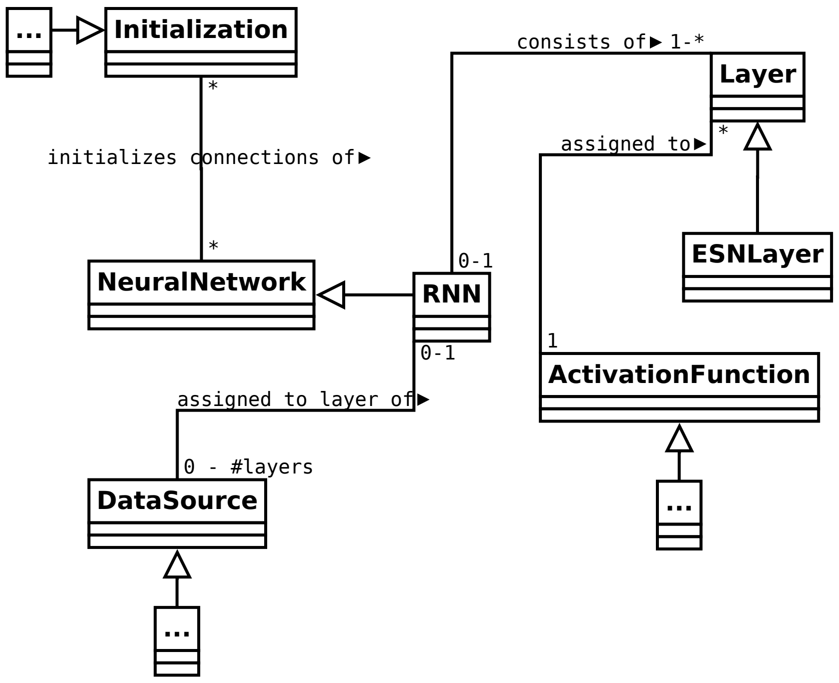

Figure 14: RNN and its relationships with other classes

3.8.3 ESN

NeuralNetwork's child class *ESN* represents Echo State Networks. Its constructor is almost identical to that of its parent class, except that it only accepts an array of pointers to *ESNLayer* objects:

```
public:
  ESN (int nLayers, ESNLayer** layerArray, int nSequences = 1);
```

Like *RNN*, this class also offers methods called *Train* and *Test* which serve the same purpose as they do in *RNN*. It is important to examine the *Train* method more closely in order understand how fnnlib's classes interact when the method is invoked. While the following description is not comprehensive, it should be sufficient for our purposes.

First, *Train* will run the ESN for a given number of time-steps using *NeuralNetwork::Run*. That method will collect the activations of the reservoir layer(s) into a matrix. At the same time, the desired-output matrix will be constructed from data read from the *DataSource* object(s) associated with the output layer(s). *Train* will then pass those matrices to the *ComputeWeightMatrix* method(s) of the *ESNTrainingAlgorithm* object(s) associated with the output layer(s). Finally, the old values in the weight matrices stored inside *ESN* will be replaced by the values in the newly computed weight matrices.

ESN::Test will be explained in more detail when the *NetworkErrorData* class is introduced in section 3.10.2. In addition to *Train* and *Test*, *ESN* also has an *Adapt* method that should be called even before *Train*. It will then adapt the activation functions of all layers that have an *Adaptation* object assigned to them and use the sigmoid activation function.

Figure 15 illustrates the relationships between *ESN* and fnnlib's other classes.

3.9 RNN Training Algorithms

In fnnlib, RNN training algorithms are represented by child classes of the abstract base class *RNNTrainingAlgorithm*. There are two built-in algorithms, *BPTT* and *CBPTT*, which implement Backpropagation Through Time and Continuous Backpropagation Through Time, respectively. Figure 16 shows a class diagram of *RNNTrainingAlgorithm* and its children.

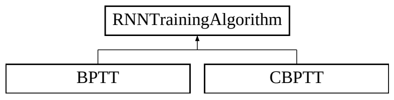

Figure 16: RNNTrainingAlgorithm and its child classes

The following subsections will introduce the base class and its two child classes.

3.9.1 RNNTrainingAlgorithm

RNNTrainingAlgorithm is a friend class of *RNN* because it needs access to many of *RNN*'s attributes to train an RNN. We will first

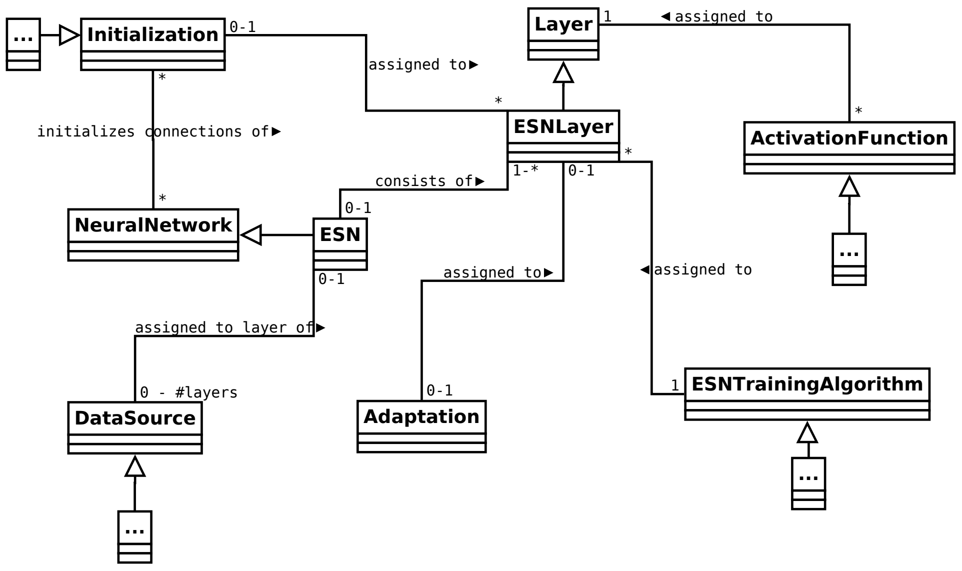

Figure 15: ESN and its relationships with other classes

have a look at its constructor, the method *Train*, and its three pure virtual methods that any non-abstract child class must implement.

```
public:
  RNNTrainingAlgorithm (int nPasses, int nEpochSize,
                        double fLearningRate,
                        double fPbLearningRate);

protected:
  void Train (RNN* net);

  virtual void ComputeGradients (RNN* net,
                                 int nSequence,
                                 int nEpochSize) = 0;

  virtual void UpdatePbs (RNN* net, int nSeq,
                          int nEpochSize,
                          double fPbLearningRate) = 0;

```

```
    virtual void UpdateWeights (RNN* net) = 0;
```

The constructor expects four parameters that will control the training process. The weights will be updated after each epoch of *nEpochSize* steps, with *nPasses* denoting the number of training epochs. The last two parameters are the learning rate for weights and PB neurons respectively.

As shown in section 3.8.2, the constructor of *RNN* receives a pointer *rnnAlg* to an *RNNTrainingAlgorithm* object. In practice, the object will actually be of type *BPTT* or *CBPTT* since *RNNTrainingAlgorithm* is abstract. Since the operations that follow are the same for both of these classes, let us assume that *rnnAlg* points to a *BPTT* object. Whenever *BPTT* is mentioned in the following sentences, the statement would also be true if *BPTT* was replaced by *CBPTT*.

When the user calls *RNN*'s *Train* method, *RNN* in turn calls *BPTT::Train* and passes a pointer to itself (*this*) to the method. *BPTT::Train* then computes the gradients and updates the PB values and the weights of the RNN by invoking *BPTT*'s implementations of the methods *ComputeGradients*, *UpdatePbs*, and *UpdateWeights*. Note that neither *BPTT* nor *CBPTT* overwrite the *Train* method inherited from *RNNTrainingAlgorithm*. In order to add another RNN training algorithm, all that needs to be done is to derive a class from *RNNTrainingAlgorithm* and implement the three pure virtual methods. The inherited *Train* method will then train the network by invoking those methods.

3.9.2 BPTT

BPTT is a child class of *RNNTrainingAlgorithm* that implements its abstract parent's *ComputeGradients*, *UpdatePbs*, and *UpdateWeights* methods. The preceding subsection explained how these methods are invoked during training. *BPTT* trains an RNN using the standard Backpropagation Through Time method introduced in section 2.2.

3.9.3 CBPTT

The only difference between *CBPTT* and *BPTT* (see above) is that it uses a continuous variant of the Backpropagation Through Time method (see section 2.4) to train an RNN.

3.10 Classes for Performance Analysis

This section will introduce two classes - *LayerErrorData* and *Network-ErrorData* - that collect error data based upon which a neural network's performance can be analyzed. While neither of the two classes is derived from the other, they are still closely related.

3.10.1 LayerErrorData

This class represents error data collected for one particular output layer in a neural network. Its objects store the layer's actual output as well as its desired output, which allows the user to compute the layer's mean squared error and its Kullback-Leibler divergence by invoking the appropriate methods. Please consult the documentation in Apppendix A for more information on the methods, as this is irrelevant as far as fnnlib's design is concerend. What is important to note is that the user will never have to create *LayerErrorData* objects manually. Instead, they will be generated automatically and then returned on request by the class *NetworkErrorData*, which will be explained next.

3.10.2 NetworkErrorData

To put it simply, *NetworkErrorData* objects store error data for a whole network, with the error data for the various layers being held in an array of *LayerErrorData* objects. The class provides access to those objects via a method called *GetLayerData*. Pointers to objects of type *NetworkErrorData* are returned to the user by the *Test* methods of both *RNN* and *ESN*. These processes - the two neural network classes handle them slightly differently - will be examined more closely in the following paragraphs.

In order to find out how well an RNN has learned to generate a particular pattern, the user needs to invoke *RNN::Test* using appropriate parameters. *RNN::Test* will then invoke *RNNTrainingAlgorithm::Test*, i.e. the *Test* method of the *RNNTrainingAlgorithm* object assigned to the RNN. In doing so, it will pass a pointer to its *RNN* object (*this*) to the method. Since *RNNTrainingAlgorithm* is a friend class of *RNN*, *RNNTrainingAlgorithm::Test* can use this pointer to run the RNN and collect error data by calling *RNN::CollectErrorData*, a method *RNN* inherited from its parent class *NeuralNetwork*. This method will receive a pointer to a newly created *NetworkErrorData* object, which it will then fill with data. Afterwards, *RNNTrainingAlgorithm::Test* returns the filled object to *RNN::Test*, which in turn returns it to the user.

With ESN, this process is a lot simpler. When the user calls *ESN::Test*, that method will run the network and create a *NetworkErrorData* object. A pointer to the object will be passed to the inherited method
ESN::CollectErrorData to have it filled with data. *ESN::Test* then returns the object to the user.

Why then does the same task require a much more complicated interaction of objects in the case of RNN? The simple reason is that the *RNNTrainingAlgorithm* class is involved because, with RNN, it is possible to train PB neurons even during the testing phase. In contrast, ESN can simply be run without any of their components being adapted during testing.

4 A Small Practical Example

In this section, we will see how to use fnnlib to create, train, and test an Echo State Network. The purpose of this section is to give the reader a feel of what actually using fnnlib is like. More information on this topic can be found in the documentation.

The training data in this example will consist of 1000 datasets generated by the Mackey-Glass equation[4], with the input data stored in a textfile "mg-in.txt" and the desired-output data stored in "mg-out.txt". The network will be trained to perform a next-step prediction, i.e. given the value at time-step *n* in a time-series, it should predict the value at time-step *n+1*. The following code listing shows how to create the network, with details below.

```
PermutationInitialization res_init (-0.1, 0.1);

ESNLayer** layers = new ESNLayer*[3];
layers[0] = new ESNLayer (1);
layers[1] = new ESNLayer (100,
                          ActivationFunction::TANH,
                          NULL,
                          &res_init);
```

[4]http://www.scholarpedia.org/article/Mackey-Glass_equation

```
 9 layers[2] = new ESNLayer (1);
10 ESN* net = new ESN (3, layers);
```

The ESN *net* has three layers: an input layer of size 1, a reservoir of size 100, and an output layer of size 1. The input and output layers will use the identity activation function, whereas the reservoir is assigned the hyperbolic tangent activation function. The reservoir's internal connections will be initialized by the *PermutationInitialization* object *res_init*. If the reservoir's incoming connections needed to be trained, the *NULL* argument could be replaced by a pointer to an *ESNTrainingAlgorithm* object. By default, fnnlib would use the *LinearLeastSquaresAlgorithm* class for training. (Note that we would also have to assign desired-output data to the reservoir in order to train its incoming connections, so nothing will be done in this example anyway.) Later on it will actually use that class to train the incoming connections of the output layer, since we did not assign any training algorithm to it (see line 9).

Next, the input-to-reservoir and reservoir-to-output connections need to be initialized.

```
1 RandomInitialization rand_init (-0.1, 0.1);
2 net->ConnectLayerToLayer (0, 1, &rand_init, false);
3 net->ConnectLayerToLayer (1, 2, &rand_init, true);
```

The two method calls tell *net* to let *rand_init* initialize the connections between layer 0 (input) and 1 (reservoir), and layer 1 and 2 (output). The former connections will be fixed (as indicated by argument 4, *false*), while the latter connections will be trainable. The next step is to provide the network with input and desired-output data.

```
1 FileDataSource ds_in (1000, 1, (char*) "mg-in.txt");
2 FileDataSource ds_out (1000, 1, (char*) "mg-out.txt");
3 net->SetInput (0, &ds_in);
4 net->SetOutput (2, &ds_out, false);
```

As can be seen in the first two lines, *ds_in* and *ds_out* will read 1000 datasets of size 1 (i.e. 1000 double values) from the two files. *net*'s first layer (index 0) is finally turned into a true input layer by having a data source assigned to it in line 3. The source of desired-output data *ds_out* is assigned to layer 3 (index 2) in line 4. Argument 3 to *SetOutput* specifies whether *teacher forcing* should be used. If it is set to *true*, the network will write the desired-output data into the layer during training, thus pretending that the neurons always produce the correct output. This is only necessary if the layer has outgoing connections, though, so we set the argument to *false*.

Finally, we will train the network, test it, and print the mean squared error that was computed for its output layer. During the initial washout phase (see section 2.5), the network will be run for 1000 time-steps without any data being collected. The reservoir activations and desired-output values of the next 1000 time-steps will be collected and used to compute new reservoir-to-output weights (remember that only these weights were chosen to be trainable). This all happens in line 1 of the following code listing, with argument 1 specifying the length of the washout phase and argument 2 giving the length of the actual training phase.

```
1 net->Train (1000, 1000);
2 NetworkErrorData* ned = net->Test (1000, 1000);
3 cout << "MSE: " << ned->GetLayerData (2)->GetMse () << endl;
```

After training, the network is tested. Once again, there is an initial washout phase of length 1000 before error data is collected during the following 1000 time-steps (as indicated by the arguments to *Test* in line 2). The *Test* method will then return a pointer to a *NetworkErrorData* object that could potentially hold error data for several output layers. Since layer 3 was the only output layer, we retrieve a pointer to its *LayerErrorData* object (zero-based index 2) and have it compute the mean squared error (MSE) in line 3.

Note that this example is very basic, with the same input data being used during training and testing. The mean squared error in a test run was 0.000000451050. Figure 17 shows a plot of both the original data (green) and the ESN-generated data (blue) on a small interval, but since the error was very small, it is virtually impossible to tell the curves apart. While this renders the figure rather useless, it does show that the results achieved by the ESN are very good. Figure 18 shows a magnified part of the curves to prove that the result was not perfect.

Another example code for creating an ESN can be found in the documentation. It also shows how to create an RNN that is trained using the BPTT algorithm.

5 Related Work

This section will briefly introduce several software libraries that provide functionality similar to what fnnlib offers. Naturally, the selection of projects pesented here is by no means exhaustive. Some projects that seemed worth mentioning at first glance were excluded for the simple reason that development was stopped four or more years ago.

The first of the libraries is FANN[5], which is short for Fast Artificial Neural Network Library. It is an Open Source library written in ANSI C and claims to be up to 150 faster than other neural network libraries. Apart from a comprehensive reference manual, it offers bindings for many other languages such as C++, Java, Perl, and PHP. As

[5]http://leenissen.dk/fann/

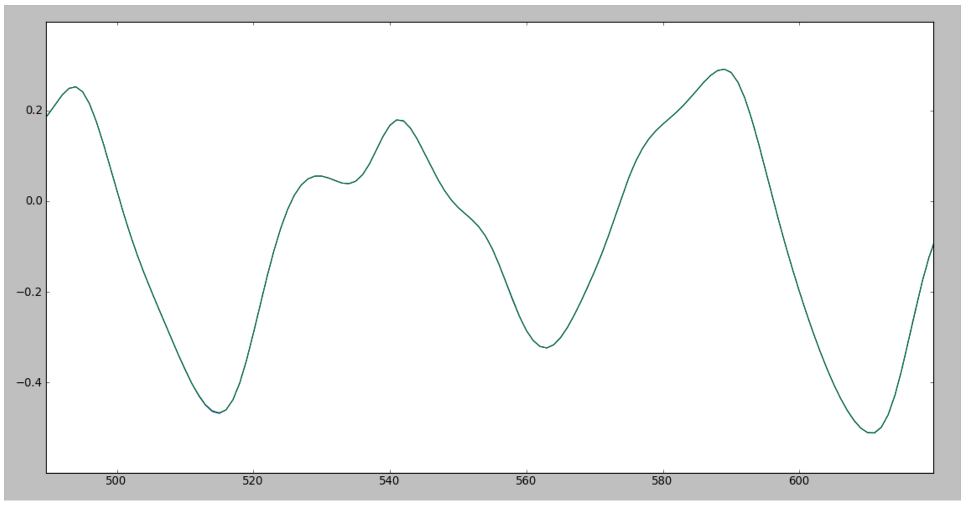

Figure 17: Original Mackey-Glass data (green) plotted against data predicted by ESN (blue); see Figure 18 for a magnified picture.

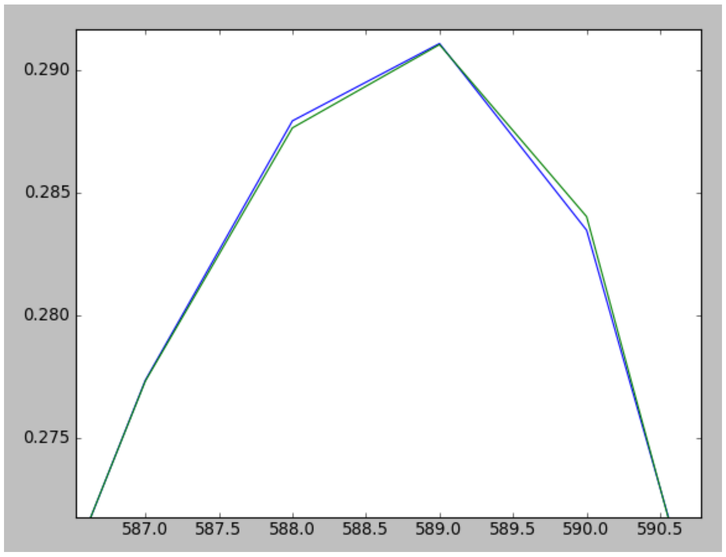

Figure 18: A part of Figure 17 magnified; original Mackey-Glass data (green) plotted against data predicted by ESN (blue)

far as functionality is concerned, FANN offers several backpropagation methods that can be used to train recurrent neural networks. Unlike fnnlib, it does not support Continuous-Time RNN, trainable PB layers, or Echo State Networks. However, one feature that fnnlib may adopt from libraries such as FANN is the possibility to save entire neural networks to files and load them again later on.

Next up is Flood[6], an Open Source C++ library developed at the International Center for Numerical Methods in Engineering[7]. Its website contains a comprehensive user's guide which also explains the theory behind the algorithms used in the library. Flood offers several features and, more specifically, training methods that are not currently included in fnnlib, such as Newton's method or evolutionary weight adaptation, but it does not offer support for Continuous-Time RNN or Echo State Networks. It also imposes more restrictions on the architecture of a neural network.

One library that does include support for Echo State Networks is aureservoir[8], another Open Source C++ library. Unlike fnnlib, it includes features such as leaky integrator neurons or the ridge regression training algorithm. However, this library specializes in Echo State Networks and does not support general RNN that are trained using gradient-based methods. Also, it does not allow the kind of architectural flexibility offered by fnnlib, such as creating ESN with multiple reservoirs. aureservoir's documentation was generated over 2.5 years ago, so it is not clear whether development of the library will continue in the future.

Another piece of software that supports Echo State Networks is the MATLAB Toolbox for Echo State Networks[9], which was written by Herbert Jaeger and members of his research group. As indicated by its name, it is not a software library for use in practical applications but was intended "mainly for didactic purposes and quick experiments."[10] Like aureservoir, the toolbox offers some functionality not included in fnnlib, such as leaky integrator neurons, but it also supports basic ESN architectures only. In order to run the toolbox, MathWorks' commer-

[6]http://www.cimne.com/flood/default.asp
[7]http://www.cimne.com/
[8]http://aureservoir.sourceforge.net/
[9]http://www.reservoir-computing.org/node/129
[10]http://www.reservoir-computing.org/node/129

cially available MATLAB software[11] is required.

6 Conclusion

While there are many neural network libraries available on the Internet, fnnlib offers a unique selection of features. Its support of state-of-the-art network architectures as well as the great amount of flexibility it allows make it especially suitable for use in research. Even though fnnlib was influenced by the particular requirements of Osaka University's Graduate School of Engineering, other researchers - especially in the field of robotics - should find it useful, too.

As mentioned in sections 3 and 5, there are several ideas for future modifications to the library that will enhance its functionality, even though it is currently impossible to tell how soon new features will be added. Potential users or developers should visit fnnlib's SourceForge project page[12] to track its development.

References

[1] R. H. Cuijpers, F. Stuijt, and I. G. Sprinkhuizen-Kuyper. Generalisation of action sequences in rnnpb networks with mirror properties. *Proceedings of the European Symposiumon Neural Networks (ESANN)*, 2009.

[2] S. S. Haykin. *Neural Networks and Learning Machines, 3rd Edition*. Pearson Education, Inc., 2009.

[3] M. Ito and J. Tani. On-line imitative interaction with a humanoid robot using a dynamic neural network model of a mirror system. *Adaptive Behavior*, 12(2), 2004.

[4] H. Jaeger. Echo state network. *Scholarpedia*, 2(9), 2007.

[5] F. Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6), 1958.

[11] http://www.mathworks.com/

[12] http://sourceforge.net/projects/fnnlib/

[6] D.E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning internal representations by backpropagating errors. *Nature*, 323, 1986.

[7] J. J. Steil. Online reservoir adaptation by intrinsic plasticity for backpropagationdecorrelation and echo state learning. *Neural Networks*, 20(4), 2007.

[8] Y. Yamashita and J. Tani. Emergence of functional hierarchy in a multiple timescale neural network model: A humanoid robot experiment. *PLoS Comput Biol*, 4(11), 2008.

[9] R. Yokoya, T. Ogata, J. Tani, K. Komatani, and H. G. Okuno. Experience-based imitation using rnnpb. *Advanced Robotics*, 21(12), 2007.

A Documentation

As can be seen in the source code files, fnnlib uses Doxygen-style[13] comments. fnnlib's SourceForge page[14] contains the complete documentation[15] generated by Doxygen.

[13] http://www.stack.nl/˜dimitri/doxygen/
[14] http://sourceforge.net/projects/fnnlib/
[15] http://sunet.dl.sourceforge.net/project/fnnlib/doxygen-doc.pdf