CANN : C based Artificial Neural Network

An ANN library implemented in C

1st Manish Shetty M
01FB16ECS192
CSE dept, 5th Semester
PES University
Bangalore, India
mmshetty.98@gmail.com

2nd Megha Kalal
01FB16ECS203
CSE dept, 5th Semester
PES University
Bangalore, India
meghakalal09@gmail.com

3rd Pallavi Mishra
01FB16ECS243
CSE dept, 5th Semester

PES University
Bangalore, India
pallavim98@gmail.com

***Abstract*—CANN is an implementation of an Artificial Neural Network (ANN) library for C. The library has been designed with an emphasis on simplicity and flexibility to configuration and changes. It exposes the building blocks of an ANN and abstracts them as easy to use functions like any other C library. It provides routines to forward propagate , back propagate , choose activation functions and fast initialization of architectural parameters too. At the same time it supports efficient computation on a regular CPU allowing to train complex models on a large datasets as well. This report provides an overview of ANNs , how they work and how they have been implemented in CANN and gives technical details of each implemented component in the library.**

I. Introduction to ANN

This segment will provide a brief introduction to artificial neural networks. It is not possible (at the moment) to make an artificial brain, but it is possible to make simplified artificial neurons and artificial neural networks. These ANNs can be made in many different ways and can try to mimic the brain in many different ways. ANNs are not intelligent, but they are good for recognizing patterns and making simple rules for complex problems. They also have excellent training capabilities which is why they are often used in artificial intelligence research.

ANNs are good at generalizing from a set of training data. E.g. this means an ANN given data about a set of animals connected to a fact telling if they are mammals or not, is able to predict whether an animal outside the original set is a mammal from its data. This is a very desirable feature of ANNs, because you do not need to know the characteristics defining a mammal, the ANN will find out by itself.

A. The Artificial Neuron

A single artificial neuron can be implemented in many different ways. The general mathematical definition is as showed in equation.

$$y(x) = g\left(\sum_{i=1}^{n} w_i x_i\right)$$

x is a neuron with n input dendrites (x0 . . . xn) and one output axon y(x) and where (w0 . . . wn) are weights determining how much the inputs should be weighted.

g is an activation function that weights how powerful the output (if any) should be from the neuron, based on the sum of the input. If the artificial neuron should mimic a real neuron, the activation function g should be a simple threshold function returning 0 or 1. This is however, not the way artificial neurons are usually implemented.

For many different reasons it is smarter to have a smooth (preferably differentiable because it might be used as a gradient while learning) activation function. The output from the activation function is either between 0 and 1, or between -1 and 1, depending on which activation function is used. This is not entirely true, since e.g. the identity function, which is also sometimes used as activation function, does not have these limitations, but most other activation functions uses these limitations. The inputs and the weights are not restricted in the same way and can in principle be between and +, but they are very often small values centered around zero.

As mentioned earlier there are many different activation functions, some of the most commonly used are threshold , sigmoid.

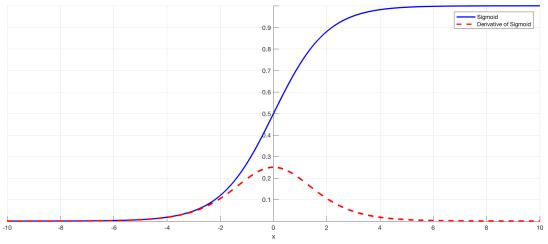

Fig. 1. sigmoid and it's derivative

B. The Artificial Neural Network

The ANN library chosen to implement is a multilayer feedforward ANN, which is the most common kind of ANN. In a multilayer feedforward ANN, the neurons are ordered in

layers, starting with an input layer and ending with an output layer. Between these two layers are a number of hidden layers. Connections in these kinds of network only go forward from one layer to the next.

Multilayer feedforward ANNs have two different phases: A training phase (sometimes also referred to as the learning phase) and an execution phase. In the training phase the ANN is trained to return a specific output when given a specific input, this is done by continuous training on a set of training data. In the execution phase the ANN returns outputs on the basis of inputs.

The feedforward ANN functions are executed as follows : input is presented to the input layer, the input is propagated through all the layers until it reaches the output layer, where the output is returned. In a feedforward ANN an input can easily be propagated through the network and evaluated to an output. Whereas calculating a clear output in a network which allows connections in all directions is difficult due to formation of loops.

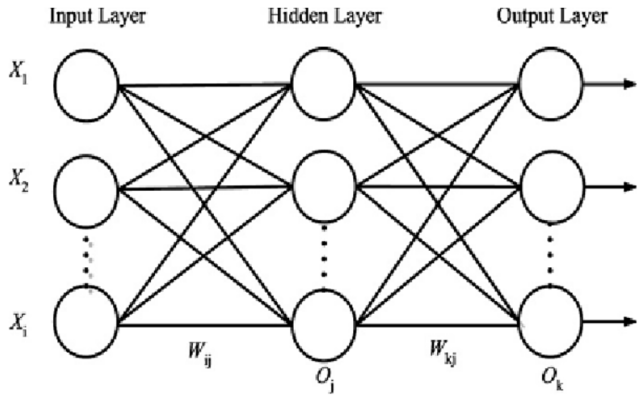

Fig. 2. A fully connected multilayer feedforward network with one hidden layer.

Two different kinds of parameters can be adjusted during the training of an ANN, the weights and the t value in the activation functions. This is impractical and it would be easier if only one of the parameters be adjusted.

II. Training an ANN

A. The Back propagation Algorithm

The back propagation algorithm works in much the same way as the name suggests: After propagating an input through the network, the error is calculated and the error is propagated back through the network while the weights are adjusted in order to make the error smaller.

Although we want to minimize the mean square error for all the training data, the most efficient way of doing this with the back propagation algorithm, is to train on data sequentially one input at a time, instead of training on the combined data. However, this means that the order the data is given in is of importance, but it also provides a very efficient way of avoiding getting stuck in a local minima. I will now explain the back propagation algorithm, in sufficient details to allow an implementation from this explanation:

First the input is propagated through the ANN to the output. After this the error e_k on a single output neuron k can be calculated as:

$$e_k = d_k - y_k$$

Where y_k is the calculated output and d_k is the desired output of neuron k. This error value is used to calculate a $delta_k$ value, which is again used for adjusting the weights. The $delta_k$ value is calculated by:

$$delta_k = e_k g'(y_k)$$

Where g' is the derivative of the activation function.

When the $delta_k$ value is calculated, we can calculate the $delta_j$ values for preceding layers. The $delta_j$ values of the previous layer is calculated from the $delta_k$ values of this layer as

$$delta_j = \eta g'(y_j) \left(\sum_{i=1}^{n} delta_k w_{jk} \right)$$

Using these delta values, the w values that the weights should be adjusted by, can be calculated by:

$$\Delta w_{jk} = delta_j y_k$$

The above value is used to adjust the weight accordingly to reduce errors in further iterations as

$$w_{jk} = w_{jk} + \Delta w_{jk}.$$

The back propagation algorithm moves on to the next input and adjusts the weights according to the output. This process goes on until a certain stop criteria is reached. The stop criteria is typically determined by measuring the mean square error of the training data while training with the data, when this mean square error reaches a certain limit, the training is stopped.

III. Design

A. Architectural

The structure of the entire the ANN can be seen from the C-structure used to represent it

The configurations mentioned above are initialized within the function

cann_init(int inputs, int hidden_layers, int hidden, int outputs)

The parameters passed to the above function are

1) Number of inputs
2) Number of hidden layers
3) Number of hidden neurons within each hidden layer
4) Number of output

Based on the values mentioned above the number of hidden weights, output weights, total number of neurons and the

```c
typedef struct cann
{

        int inputs,hidden_layers,hidden,outputs;

        //activation function for hidden layer and output
        cann_actfun activation_hidden;
        cann_actfun activation_output;


        //total no of weights
        int total_weights;

        //neurons of ip + hidden + op
        int total_neurons;

        //weight array
        double *weight;

        //output array
        double *output;

        //error delta for hidden and output neuron(total - inputs)
        double *delta;


}cann;
```

Fig. 3. structure of the ANN

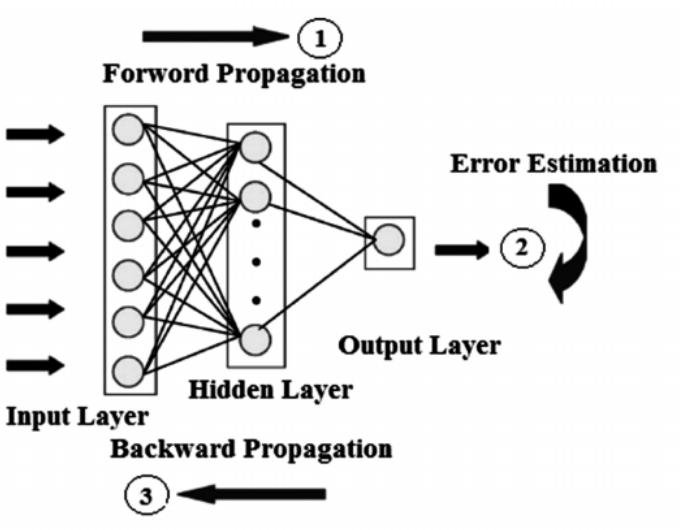

Fig. 4. structure of the ANN

total size of ann are calculated as explained in the previous section. The input, hidden and output layers are allocated with consecutive blocks of memory.

The trained model can sometimes be of a great use for further prediction. Eg: If the ann built is being used to create a GATE for an operation like xor/and then the ann which reaches a satisfactory accuracy can be saved.
Thus re-usability of the trained model is achieved by the following functions :

1) **void cann_write(cann const *ann, FILE *out_struct, FILE *out_weight)** :
   This function writes the structure and weights of the model to two separate binary files.
2) **cann*cann_read(FILE*in_struct,FILE*in_weight)**:
   This function initializes the ann structure back with the values read from the binary files to which they were stored in the previous function.

The ANN model needs to be evaluated once it's trained and tested. The evaluation is carried out by the functions

1) **float loss(double predicted[] ,double expected[] , int n)** :
   This function calculates the mean squared error(mse)
2) **float test_rmse(double predicted[] ,double expected[] , int n)** :
   This function calculates the root mean squared error(rmse)

B. Algorithmic

As explained in the mathematical section above the algorithm followed takes place in a pipelined manner.
This is achieved by 2 functions in CANN :

1) **double const *cann_run(cann const *ann, double const *inputs)**: This function is the forward propagation function that finds the sum of products of weights and inputs. It then applies the activation function chosen while initializing the network and applies it to return the outputs. It does so at every hidden as well as the output layer
2) **void cann_train(cann const *ann , double const *inputs , double const* desired_outputs, double learning_rate**): This function is the training and thus the backward propagation function that first finds the deltas(errors) and using the required formulae explained in the previous section. The process starts at errors at the output layer which is used to calculate errors of the last hidden layer and so on. These errors are then used to update the weights at ever layer.

IV. Tests and Results

A. Minor Tests

The initial tests were done on simple binary gates run by the ANN. The inputs where defined as 0,0,0,1,1,0,1,1 for which logic gates :

1) XOR
2) AND
3) OR

were implemented. The results of the ANN were close to the required outputs. Since the ANN ran on a floating point value , the final results had to be "ceiled"/"floored" to either a 1/0 so as to receive a final binary value.
eg: XOR gate run on CANN (2,1,2,1) which means CANN with 2 input nodes , 1 hidden layer , 2 hidden nodes , 1 output. Activation function at both hidden and output layer was chosen to be sigmoid.

B. Classification Problem

The classification test was done on the **Iris data** set where the first four attributes(**sepal length, sepal width, petal length, petal width**) were given as the inputs to the ANN. The class labels to be predicted were :

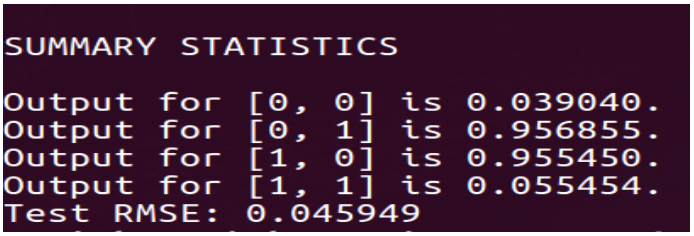

Fig. 5. xor gate built on CANN

1) "Iris-setosa",
2) "Iris-versicolor",
3) "Iris-virginica"

The labels were encoded using **one hot encoding** in order to obtain numeric outputs that made the training easier. Since this was a classification problem **Accuracy** was measured to represent the results of the test and the activation function at both **hidden and output layer was chosen to be sigmoid.**

IRIS run on CANN(4, 1, 4, 3) :

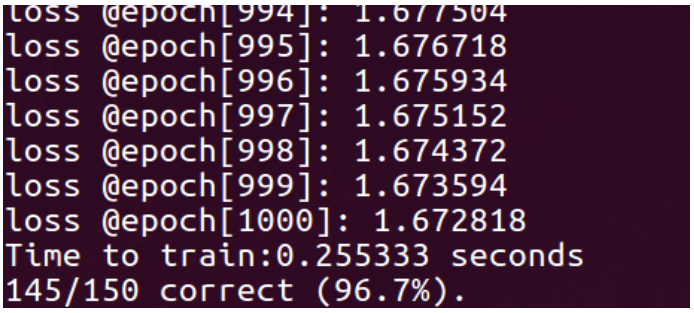

Fig. 6. IRIS classification on CANN

C. Regression Problem

Even though people prefer to not do regression problems with ANNs , we tried to apply it on **Auto-Mpg** dataset which is a dataset that can be used to predict mileage of a car based on it's features. The input attributes were **cylinders, displacement , horsepower ,weight and acceleration** and the value to be predicted was **mpg**.

After some rigorous research it was assumed that a regression problem on an ANN is best solved without any hidden layer , especially if it's a linear(can be multiple linear regression) regression as it gives a better outlook on the form of the equation **Y= mx + c**.

The Activation at the **output layer was ReLU**. Since it is a regression problem the test results were measured using **RMSE**.

Auto-Mpg run on CANN(5, 0, 0, 1) :

D. Comparison with python ANN libraries

The CANN model was compared against the ANN models built using python's inbuilt Keras and Tensorflow libraries on the same datasets. The following results were obtained :

1) Classification on Iris Dataset

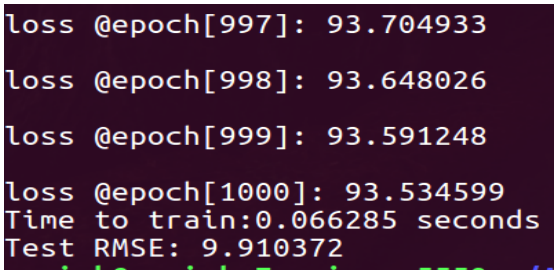

Fig. 7. IRIS classification on CANN

	CANN	Python
Accuracy	96.7	90.0
Time	0.2553	36.1941

We observed that the CANN model proved to be better in terms of Accuracy and Time taken to build the model than the python model. The CANN model wascheckedd for any kind of overfitting , and none was found. Also the overfitting can be removed if necessary by controlling the learning rate.

2) Regression on Auto-Mpg

	CANN	Python
RMSE	9.910	5.882
Time	0.0662	15.4276

We observed that the CANN is still better in terms of time taken to build the model but a little behind in terms of RMSE values as compared to the python model.

V. Conclusion

On the whole CANN has been built with at most flexibility in it's design and is easily compatible with any optimisations and applications. The Nets built can easily be stored in a file and read from them for further use. CANN library on github : https://github.com/ManishShettyM/CANN