//

How to build a Neural Network Edition

____

maXbox Starter 57 - Fast Artificial Neural Network II

"We have art to save ourselves from the truth."
    - Friedrich Nietzsche (1844-1900)

You should choose to ask a more meaningful question. Without some context, you might as well flip a coin.

In the last tutorial we have found a library, write some code, run some tests, fiddle with features, run a test, fiddle with features, realize everything is slow, and decide to use more layers.

This tutor will go a bit further to the topic of pattern recognition which implements multilayer artificial neural networks in different languages with support for both fully connected and sparsely connected networks. With FANN Cross-platform execution in both fixed and floating point are supported.

The Fast Artificial Neural Network (FANN) library is an ANN library, which can be used from C, C++, PHP, Python, Delphi and Mathematica and is still a powerful tool for software developers. ANNs can be used in areas as diverse as creating more fascinating simulation in computer games, identifying objects or semantics in images and helping the weather forecast or predict trends of the ever-changing climate.

The process usually includes several of the following steps:

1. Extract and assemble features to be used for prediction.
2. Develop targets for the training.
3. Train a known model.
4. Assess performance on test data.

ANNs apply the principle of function approximation by example, meaning that they learn a function by looking at examples of this function. One of the simplest examples is an ANN learning the XOR function (that I show later), but it could just as easily be learning to determine a language semantic of a written text.

In the following I want to show 2 solutions, one with the fannfloat.dll and a second one with the same library from FANN (fann.sourceforge.net) precompiled since in maXbox V4.5.8.10! Small functions to build an independent micro-service.

The class <TFannNetwork> encapsulates the Fast Artificial Neural Network to prevent to much low level c-code stuff or python one.

The script can be found at:

http://www.softwareschule.ch/examples/neuralnetwork21.txt
pic: http://www.softwareschule.ch/images/wine.png
..\examples\807_FANN_XorSample2.pas

If we only had some data from the future that we could use to measure our models against, then we should be able to judge our model choice only on the resulting approximation error.

Although we cannot look into the future, we can and should simulate a similar effect by holding out a part of our data. Lets remove, for instance, a certain percentage of the data and train on the remaining one.

```
inputstr1:=
'24,24,21,18,16,14,14,12,10,09,07,06,06,06,06,06,07,07,08,08,10,12,14,14';
inputstr1:= RemoveChar(inputstr1,',');

inputstr2:=
'23,23,21,18,16,15,15,14,13,13,13,12,12,12,12,13,13,13,14,15,16,18,21,21';
inputstr2:= RemoveChar(inputstr2,',');

inputstr3:=
'24,24,21,19,18,17,17,15,14,14,13,13,13,14,15,15,18,18,20,23,26,31,36,36';
inputstr3:= RemoveChar(inputstr3,',');
```

Finally, the test dataset is a dataset used to provide an unbiased evaluation of a final model fit on the training dataset.[5]

[5] https://en.wikipedia.org/wiki/Training,_test,_and_validation_sets#Test_dataset

The DLL solution is for us the easiest one but it uncovers the dependency of the DLL and explicitly steps behind. Also you do have the flexibility to use larger values from files or databases. Our goal is to train and learn a simple XOR function. First we need some types and definitions:

```
 type
    NN: TFannNetwork;
    aoutput: TFann_Type_Array3;
    TFann_Type_Array3 = Array[0..0] of single;
    TFann_Type_Array3 = array of single; //}

 NN:= TFannNetwork.create(self)
  with NN do begin
    {Layers.Strings := (
      '2'
      '3' '1') }
    Layers.add('24') //input
    Layers.add('6') //hidden
    Layers.add('3') //output

    LearningRate:= 0.001; //0.699999988079000
    ConnectionRate:= 1.000000000000000000
    TrainingAlgorithm:= taFANN_TRAIN_RPROP
    ActivationFunctionHidden:= afFANN_SIGMOID
    ActivationFunctionOutput:= afFANN_SIGMOID
    num_epochs:= 250;
  end;
```

The FANN library supports several different training algorithms and the default algorithm (FANN_TRAIN_RPROP) might not always be the best-suited for a specific problem but in our case its best suited.

Other algos are:

```
  FANN_TRAIN_NAMES: array [0..3] of string =
        (
         'FANN_TRAIN_INCREMENTAL',
         'FANN_TRAIN_BATCH',
         'FANN_TRAIN_RPROP',
         'FANN_TRAIN_QUICKPROP'
        );
```

Artificial neurons are similar to their biological counterparts. They have input connections which are summed together to determine the strength of their output, which is the result of the sum being fed into an activation function. Though many activation functions exist, the most common is the ƒ sigmoid activation function (afFANN_SIGMOID), which outputs a number between 0 (for low input values) and 1 (for high input values).

Next we want to train our network:

For the example <neuralnetwork21.txt> see at the source, the following code refers to tutorial 56!

```
     //Train the network
        for e:=1 to 6000 do //Train ~30000 epochs
        begin
                for i:=0 to 1 do begin
                        for j:=0 to 1 do begin
                            inputs[0]:=i;
                            inputs[1]:=j;
                            outputs[0]:=i Xor j;

```

```
                            mse:= NN.Train(inputs,outputs);
                            lblMse.Caption:= Format('%.4f',[mse]);
                            Application.ProcessMessages;

                        end;
                end;
        end;
```

When an ANN or tensorflow is learning to approximate a function, it is shown examples of how the function works and the internal weights Ø in the ANN are slowly adjusted so as to produce the same output as in the examples. The hope is that when the ANN is shown a new set of input variables (testdata), it will give a correct output:

```
   for i:=0 to 1 do
     begin
        for j:=0 to 1 do
        begin
              inputs[0]:=i;
              inputs[1]:=j;
              NN.Run4(inputs,aOutput);
              MemoXor.Lines.Add(Format('%d XOR %d = %f',[i,j,aOutput[0]]));
        end;
     end;

  var i,j: integer;
      inputs: array [0..1] of single;
      aoutput: TFann_Type_Array3;
```

Having too many weights can also be a slith problem, since learning can be more difficult and there is also a chance that the ANN will learn specific features of the input variables instead of general patterns which can be extrapolated to other data sets. An output of our set is shown like this:

```
           0 XOR 0 = 0.01          Mean Square Error last: 0.0005
           0 XOR 1 = 0.98
           1 XOR 0 = 0.99
           1 XOR 1 = 0.02
```

The more you repeat press on <Train> button the closer you get the XOR values:

```
           0 XOR 0 = 0.00
           0 XOR 1 = 0.99
           1 XOR 0 = 0.99
           1 XOR 1 = 0.02
```

The training is done by continually adjusting the weights so that the output of the ANN matches the output in the training file. One cycle where the weights are adjusted to match the output in the training file is called an epoch. In this example the maximum number of epochs have been set to 6000, and a status report is printed every cycle.

So I did write the cycle result (as mean square error) out to the console, you can follow the approximation:

```
          mse:= NN.Train(inputs,outputs);
          lblMse.Caption:= Format('%.4f',[mse]);
          writeln(itoa(e) +': '+Format('%.4f',[mse]));

            1: 0.1558    .........    5997: 0.0001
            1: 0.4369    .            5997: 0.0001
            1: 0.3152    .            5997: 0.0002
            1: 0.2959    .            5997: 0.0002
            2: 0.2004    .            5998: 0.0001
            2: 0.3854    .            5998: 0.0001
            2: 0.2745    .            5998: 0.0002
            2: 0.3282    .            5998: 0.0002
            3: 0.2229    .            5999: 0.0001
```

```
            3: 0.3618    .          5999: 0.0001
            3: 0.2575    .          5999: 0.0002
            3: 0.3421    .          5999: 0.0002
            4: 0.2335    .          6000: 0.0001
            4: 0.3508    .          6000: 0.0001
            4: 0.2503    .          6000: 0.0002
            4: 0.3478 ....          6000: 0.0002
```

When measuring how close an ANN matches the desired output, the mean square error is usually used. The mean square error is the mean value of the squared difference between the actual and the desired output of the ANN, for individual training patterns. A small mean square error means a close match of the desired output.

Lets summarize the steps in the script on behalf of a click:

```
procedure TForm1btnBuildClick(Sender: TObject);
begin
       with nn do begin
         Layers.add('24')
         Layers.add('6')
         Layers.add('3')
         LearningRate:= 0.001; //0.699999988079071100
         ConnectionRate:= 1.000000000000000000
         TrainingAlgorithm:= taFANN_TRAIN_RPROP
         ActivationFunctionHidden:= afFANN_SIGMOID
         ActivationFunctionOutput:= afFANN_SIGMOID
         num_epochs:= 250;
       end; //}
       NN.Build;
       btnBuild.Enabled:=false;             //1 NN.Build();
       BtnTrain.Enabled:=true;              //2 NN.Train(inputs,outputs);
       btnRun.Enabled:=true;                //3 NN.Run4(inputs,aOutput);
       MemoXOR.Lines.add('spec def builded')
end;
```

I also realized the networks were overfitting my data, then performing poorly on the test.

Anyone who’s studied this a tad further can tell you the connections between neurons are very important and the weights associated with them are somehow used in calculating stuff.

What happens when you’re doing forward propagation (using a learned network) is simply this:

    Take the outputs from the previous layer (a vector of numbers)
    Multiply with a vector of weights (the arrows)
    Apply the cost function (this becomes the new layer)

```
  # Step 6: prepare machine learning
  # pass placeholder for x
    predict = multilayer_perceptron(x, weights, biases)
  # define cost and optimizer
    cost= tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits=predict,
labels=y))
    optimizer =tf.train.AdamOptimizer(learning_rate=learning_rate).minimize(cost)
```

https://github.com/robodhhb/BrickClassifi3r/blob/master/02_Neural%20Network%20Training%20o

Then you just repeat this for all the layers and that’s that. That is literally all that happens.

First we build the dimensions of the neuronal (or do we say neural) net with 2 input, 3 hidden and 1 output layer (neuron).

```
    Layers.add('2') //input neuron
    Layers.add('3') //hidden
    Layers.add('1') //output
```

Second we train the net, an advantage of such a training algorithm is that the weights are being altered many times during each epoch and since each training pattern alters the weights in slightly different directions.

```
   Sum(Ed_i) = wi*xi + b --> b is biases, w is weights

   print('Weights for h1')
   wh1= weights['h1'].eval(sess)
   print(wh1)
   print('\nBiases for b1')
   bb1= biases['b1'].eval(sess)
   print(bb1)
```

And third we run it, after training, the ANN could be used directly to determine which XOR function is in, but it is usually desirable to keep training and execution on testdata in two different programs or code blocks.

By the way the well known fannfloat.dll is statically linked, better performance and stability as an advantage can be seen. Put the file fannfloat.dll in your PATH.

```
{$IF Defined(FIXEDFANN)}
     const DLL_FILE = 'fannfixed.dll';
{$ELSEIF Defined(DOUBLEFANN)}
     const DLL_FILE = 'fanndouble.dll';
{$ELSE}
     const DLL_FILE = 'fannfloat.dll';
{$IFEND}

   function fann_run(ann: PFann; input: PFann_Type): Pfann_type_array; cdecl;
   function fann_run; external DLL_FILE;
```

If you want to use Fixed Fann or Double Fann as DLL_FILE please uncomment the corresponding definition in your compiler. As default fann.pas uses the <fannfloat dll>.

I did also test this on a Ubuntu 16 Mate with Wine_2.4 and IT works too!
pic: 675_virtualbox_ubuntu_sha256_advapi32dll.png
http://www.softwareschule.ch/images/virtualbox_ubuntu_advapi32dll.png

There is also no proof that every output of common hash functions in machine learning is reachable for some input, but it is expected to be true. No method better than brute force is known to check this, and brute force is entirely impractical.

If we only had some data from the future that we could use to measure our models against, then we should be able to judge our model choice only on the resulting approximation error.
Although we cannot look into the future, we can and should simulate a similar effect by holding out a part of our data. Lets remove, for instance, a certain percentage of the data and train on the remaining one could be a strategy too.

Ref:
    http://fann.sourceforge.net
    http://leenissen.dk/fann/wp/language-bindings/
    Neural Networks Made Simple: Steffen Nissen
    http://fann.sourceforge.net/fann_en.pdf
    http://www.softwareschule.ch/examples/neuralnetwork.txt
    https://maxbox4.wordpress.com
    https://www.tensorflow.org/

https://sourceforge.net/projects/maxbox/files/Examples/13_General/807_FANN_XorSample2.pas/download
https://sourceforge.net/projects/maxbox/files/Examples/13_General/809_FANN_XorSample_traindata.pas/download

```
 
----------------------------------------------------------------
Doc: TFannNetwork Lib Interface: @author Mauricio Pereira Maia
of unit FannNetwork;

{*---------------------------------------------------------------
  TFannNetwork Component
---------------------------------------------------------------}
  TFannNetwork = class(TComponent)
  private
    ann: PFann;
    pBuilt: boolean;
    pLayers: TStrings;
    pLearningRate: Single;
    pConnectionRate: Single;
    pLearningMomentum: Single;
    pActivationFunctionHidden: Cardinal;
    pActivationFunctionOutput: Cardinal;
    pTrainingAlgorithm: Cardinal;

    procedure SetLayers(const Value: TStrings);

    procedure SetConnectionRate(const Value: Single);
    function GetConnectionRate(): Single;
    procedure SetLearningRate(Const Value: Single);
    function GetLearningRate(): Single;

    procedure SetLearningMomentum(Const Value: Single);
    function GetLearningMomentum(): Single;
    procedure SetTrainingAlgorithm(Value: TTrainingAlgorithm);
    function GetTrainingAlgorithm(): TTrainingAlgorithm;

    procedure SetActivationFunctionHidden(Value: TActivationFunction);
    function GetActivationFunctionHidden(): TActivationFunction;
    procedure SetActivationFunctionOutput(Value: TActivationFunction);
    function GetActivationFunctionOutput(): TActivationFunction;

    function GetMSE(): Single;

    function EnumActivationFunctionToValue(Value: TActivationFunction): Cardinal;
    function ValueActivationFunctionToEnum(Value: Cardinal): TActivationFunction;
    function EnumTrainingAlgorithmToValue(Value: TTrainingAlgorithm): Cardinal;
    function ValueTrainingAlgorithmToEnum(Value: Cardinal): TTrainingAlgorithm;

  public
    constructor Create(Aowner: TComponent); override;
    destructor Destroy(); override;
    procedure Build();
    procedure UnBuild();
    function Train(Input: array of fann_type; Output: array of fann_type): single;
    procedure TrainOnFile(FileName:String;MaxEpochs:Cardinal;DesiredError:Single);
    procedure Run(Inputs: array of fann_type; var Outputs: array of fann_type);
    procedure SaveToFile(FileName: String);
    procedure LoadFromFile(Filename: string);
    // adapt to maXbox4 for strong typing
    procedure Run4(Inputs: array of fann_type; var Outputs: TFann_Type_Array3);
    {*---------------------------------------------------------
     Pointer to the Fann object.
     If you need to call the fann library directly and skip the Component.
    ----------------------------------------------------------}
    property FannObject: PFann read ann; published

    *----------------------------------------------------------
     Network Layer Structure. Each line need to have the number of neurons
        of the layer. 2 4 1
        Will make a 3 layered network with 2 input neurons, 4 hidden neurons
        and 1 output neuron.
```

```
    ------------------------------------------------------------
    property Layers: TStrings read PLayers write SetLayers;

    {*-----------------------------------------------------------
     Network Learning Rate.
    -----------------------------------------------------------}
    property LearningRate: Single read GetLearningRate write SetLearningRate;
    {*-----------------------------------------------------------
     Network Connection Rate. See the FANN docs for more info.
    ------------------------------------------------------------}
    property ConnectionRate: Single read GetConnectionRate write SetConnectionRate;
    {*------------------------------------------------------------
     Network Learning Momentum. See the FANN docs for more info.
    ------------------------------------------------------------}
    property LearningMometum: single read GetLearningMomentum write SetLearningMomentum;
    {*------------------------------------------------------------
     Fann Network Mean Square Error. See the FANN docs for more info.
    -------------------------------------------------------------}
    property MSE: Single read GetMSE;
    {*-------------------------------------------------------------
     Training Algorithm used by the network. See the FANN docs for more info.
    --------------------------------------------------------------}
    property TrainingAlgorithm: TTrainingAlgorithm read GetTrainingAlgorithm write SetTrainingAlgorithm;

    {*-------------------------------------------------------------
     Activation Function used by the hidden layers. See FANN docs for more info.
    --------------------------------------------------------------}
    property ActivationFunctionHidden: TActivationFunction read GetActivationFunctionHidden write SetActivationFunctionHidden;
    {*-------------------------------------------------------------
     Activation Function used by the output layers. See the FANN docs for more info.
    --------------------------------------------------------------}
    property ActivationFunctionOutput: TActivationFunction read GetActivationFunctionOutput write SetActivationFunctionOutput;
end;

 Performance Abstract:

 While training the ANN is often the big time consumer, execution can often be more time consuming, especially in systems where the ANN needs to be executed hundreds of times per second or if the ANN is very large. For this reason, several measures can be applied to make the FANN library execute even faster than it already does.
 One method is to change the activation function to use a stepwise linear activation function, which is faster to execute, but which is also a bit less precise. It is also a good idea to reduce the number of hidden neurons if possible, since this will reduce the execution time. from <fann_en.pdf>

    1. Extract and assemble features to be used for prediction.
    2. Develop targets for the training and data.
    3. Train a known model.
    4. Assess performance on test data.

Steps with Python 3.6 and TensorFlow

#####################################################
import tensorflow as tf
from numpy import loadtxt, savetxt, reshape
import datetime as dt

# Step 1: Import Training Data (xTrain and yTrain)
print('Lese xTrain- und yTrain-Daten')
xTrain= loadtxt('xTrain_TwoCubesCylinder375-24.csv')
yTrain= loadtxt('yTrain_TwoCubesCylinder375-3.csv')

# Step 2: Import Test Data (xTest and yTest)
```

```
print('Lese xTest und yTest-Daten')
xTest= loadtxt('xTest_TwoCubesCylinder300-24.csv')
yTest= loadtxt('yTest_TwoCubesCylinder300-3.csv')

# Step 3: Define learnparameters
learning_rate = 0.001
num_epochs = 250
num_examples= xTrain.shape[0]
print('number of traindata: '+repr(num_examples))
print('number of testdata: ' +repr(xTest.shape[0]))

.......

# Step 5: Initialise modell with randomnumbers
weights = {
    'h1': tf.Variable(tf.random_normal([n_input, n_hidden_1])),
    'out': tf.Variable(tf.random_normal([n_hidden_1,n_classes]))
}
biases = {
    'b1': tf.Variable(tf.random_normal([n_hidden_1])),
    'out': tf.Variable(tf.random_normal([n_classes]))
}

#Step 7: Training
print('\n-------------Trainingstime----------------')
with tf.Session() as sess:
    sess.run(init)
    for i in range(num_epochs):
        for j in range(num_examples):  //xTrain.shape[0]
            _, c = sess.run([optimizer, cost],
                            feed_dict={x: [xTrain[j]],
                                       y: [yTrain[j]]})
        if i % 25 == 0:
            print('epoch {0}: cost = {1}'.format(i, c))
    print('epoch {0}: cost = {1}'.format(i, c))
    print('Training finished.')
    duration = (dt.datetime.now() - start)
    print("traintime: " + str(duration))

#Step 8: compute match based on testdata
    correct_prediction = tf.equal(tf.argmax(predict, 1), tf.argmax(y, 1))
    accuracy = tf.reduce_mean(tf.cast(correct_prediction, "float"))
    print('Testergebnis:', accuracy.eval({x: xTest, y: yTest}))

>>> Lese xTrain- und yTrain-Daten
Lese xTest und yTest-Daten
Anzahl der Trainingdaten: 375 Anzahl der Testdaten: 300
NN-Architektur: 24 - 6 - 3

-------------Trainingsphase-----------------
epoch 0: cost = 22.401424407958984
epoch 25: cost = 0.0005733515135943389
epoch 50: cost = 0.0009584600338712335
epoch 75: cost = 0.0005021026590839028
epoch 100: cost = 0.0003413571394048631
epoch 125: cost = 0.00024423000286333263
epoch 150: cost = 0.00017510310863144696
epoch 175: cost = 0.00012706902634818107
epoch 200: cost = 9.417090768693015e-05
epoch 225: cost = 7.116541382856667e-05
epoch 249: cost = 5.4834770708112046e-05
Training beendet.
Dauer: 0:00:49.869540
Testergebnis: 0.863333
```