Using Neural Networks for Document Classification

Grant Guenter*, Geordie Parappilly†, Weijun Qiu‡, Andy Soobrian§ and Fabian Volz¶
*57336133 t1b9 grantguenter@gmail.com
†33948134 n1u9a geordie.p@hotmail.com
‡42701136 d8x8 weijun.q@gmail.com
§51982122 o3j8 andys12358@gmail.com
¶73138159 u2p0b f.volz@live.de

Introduction

We have implemented an artificial neural network in Erlang, trained it, and then applied it to the document classification problem, that is the problem of determining its author upon input of a text passage.

The end product allows users to, given a text document, determine if the style of the document matches any of the authors included in the dataset that the algorithm has been trained on. Given sufficient examples of an author's work, it should also be possible to train the system to recognize new authors who have styles that are sufficiently distinct from those that have already been input. Since training a neural network often requires processing large data sets, we have strived for a concurrent implementation which makes use of all the available processing power.

The value of this project is manifold. It demonstrates, through the application to machine learning, which has gained much popularity recently, that concurrency is a topic of extreme relevance these days. It shows that Erlang features good built-in support for concurrent programming, making it a potentially useful language. The project also allows us to display our ability to rapidly learn an unknown programming language and exploit its benefits to write a substantial program.

This document is a detailed description of what we have accomplished and how we did it. It is self-contained and partially repeats what has already been stated in the proof-of-concept and plan. However, in order to understand the document thoroughly, it is highly recommended to be familiar with the background research report.

The first section gives an overview of the features we have been able implement as well as of those that are missing for the complete set of goals. In sections II–VI we then present and explain most parts of the code; the entire code can be found in [1]. The more generic and less interesting helper functions are defined in Section VII. Section VIII finally gives a small example of how the neural network code can be used.

I. The Final Project

We have made significant achievements in trying to complete the final project and we have been able to implement all the core functionality which is required to classify documents. More precisely, we have been able to implement the following:

- *Document acquisition:* Having downloaded a copy of Project Gutenberg's RDF catalog, the user can use the function `find_author_works` to obtain a list of integers that correspond to the books written by a specific author. This list can then be used to call `process_file_numbers` to download the documents. See Section II.
- *Document preprocessing:* After downloading the documents, the user can repeatedly call `process_file` to extract the real content, removing prefaces etc. The text of each chapter is written into a file. See Section III.
- *Text parsing:* The user can use the function `chapter_stats` on a chapter text to compute its statistics. What is returned by this function can be used as an input for an artificial neural network, that is to say it returns the X value of a training sample (X, Y). The Y value should be the j-th standard unit vector in $\mathbb{R}^r$, if this chapter was written by the j-th author and the user wants to train the network on r authors. See Section IV.
- *Neural network training:* Once the user has all training samples, he can use the function `training_session_parallelism` to get a trained neural network. He might want to customize this function to get better results. See Section V.
- *Classification:* The user can compute the statistics of an unclassified chapter and then use the previously obtained network and the `feed_fordward_` function to try to determine its author. See Section VI.

As can be seen, our project provides very powerful tools for document classification and we successfully accomplished the low-risk goals. Unfortunately, the following features of the full set of goals are still missing:

- *Node parallelism:* We have only been able to implement the training session parallelism strategy for the backpropagation algorithm. The node parallelism strategy would require to spawn an Erlang process for every node in the network. Each node would have to store a list of the process identifiers for the nodes of the previous layer and—if also the feedforward procedure should be parallelized—the subsequent layer.
  This approach makes it necessary to send and receive many messages between the processes (compared to the actual computations), thus it might not be very efficient. Still it would have been interesting to see how a more

complicated architecture is facilitated by Erlang's concurrency features.

- *All-automatic network training:* In theory, once the user has a copy of the RDF catalog, the network could be trained from a list of author names alone. This would only require to correctly put the aforementioned, provided functions together. Instead of going through the entire process and manually keeping track of the training samples, the user would only have to call two functions, one for document acquisition, preprocessing, parsing and training, and one for classifying.

II. Document Acquisition

We source our documents from Project Gutenberg, an online collection of books which are in the public domain. Unfortunately, the main website of Project Gutenburg does not allow automated access. In order to search through the collection or to download files, it is necessary to obtain a local copy of the RDF catalog provided here: http://www.gutenberg.org/wiki/Gutenberg:Feeds. Because this catalog is unsorted, we use linear search to read each RDF file and determine if it matches the desired author, using the `find_author_works` function of the `rdf_processor` module.

```
-module(rdf_processor).
-export([find_author_works/1]).

% Input: String AuthorName, in format "Last Name, First
     name" ex. "Dickens, Charles"
% Output: list of Natural, each element in the list
     corresponds to a file number of one of the author's
     works.
find_author_works(AuthorName)
  -> find_author(AuthorName,1,[]).

find_author(_,50543, Works)-> Works;
find_author(AuthorName, 50283, Works)-> find_author(
     AuthorName, 50284, Works);
find_author(AuthorName, 50465, Works)-> find_author(
     AuthorName, 50466, Works);
find_author(AuthorName, 50541, Works)-> find_author(
     AuthorName, 50542, Works);
find_author(AuthorName, Num,Works)->
        DIR = "C:\\Users\\Grant\\Desktop\\Gutenburg RDF
             Files\\rdf-files.tar\\cache\\epub",    %%
             the location of the RDF files from Project
             Gutenburg
        FileAuthor = file_read_author(make_path_local(
             DIR,Num)),
        if
                FileAuthor =:= AuthorName->
                        find_author(AuthorName, Num+1,[
                             Num|Works]);
                true->
                        find_author(AuthorName, Num+1,
                             Works)
                        end.

file_read_author(FilePath)->
        case file:read_file(FilePath) of
                {ok, File}  -> {ok,File};
                {error, enoent} -> File = FilePath
        end,
        FileString = binary_to_list(File),
        case            string:str(FileString,"<pgterms:
             name>")=:= 0 of
                true->
                        "Error: No Author Found";
                false->
                        string:sub_string(FileString, (
                             string:str(FileString,"<
                             pgterms:name>")+14),(string:
                             str(FileString, "</pgterms:
                             name>")-1))
        end.

make_path_local(DIR, Num)->
        string:concat(string:concat(string:concat(string
             :concat(string:concat(DIR,"\\"),
             integer_to_list(Num)),"\\pg"),
             integer_to_list(Num)),".rdf").
```

Once it is determined which files belong to a given author, the files can be downloaded from one of Project Gutenberg's download mirror sites using the `process_file_numbers` function from the `html_processor` module.

```
-module(html_processor).
-import(httpc,[request/1]).
-export([process_file_numbers/1]).

% Input: list of Natural, the file numbers of the
     desired documents.
% Downloads, processes and writes the files to location
     DIR.
process_file_numbers([])->inets:stop(),  ok;
process_file_numbers([Head|Tail])->
        inets:start(),
        Mirror = "http://www.mirrorservice.org/sites/ftp
             .ibiblio.org/pub/docs/books/gutenberg/",
        DIR = "C:\\Users\\Grant\\Erlang Workspace\\
             DocumentProcessor\\Documents",  %% Where you
              want the ORIGINAL, downloaded documents to
             be stored
        Name = integer_to_list(Head),
        ShortPath = string:concat(string:concat(DIR,"\\"
             ),Name),
        FilePath = string:concat(ShortPath ,".txt"),
        case  string:equal("error: no txt file",
             FileContents ) of
            false-> file:write_file(FilePath,
                 FileContents),
            paragraph:remove_preface_etc(FilePath),
            paragraph:process_file(string:concat(DIR
                 ,"\\"), Name),
            inets:stop(),
            process_file_numbers(Tail);
            true-> process_file_numbers(Tail)
            end.
read_URL(URL)->
        [BinURL|_] = re:replace(URL, ".txt", ""),
        BaseURL = binary_to_list(BinURL),
        case httpc:request(get,{string:concat(BaseURL,".
             txt"), []},[], []) of
        {ok, {{_Version1, 200, _ReasonPhrase1},
             _Headers1, Body1}} ->Body1;
        _Else1 ->
        case  httpc:request(get,{string:concat(BaseURL,"
             -0.txt"), []},[], []) of
        {ok, {{_Version2, 200, _ReasonPhrase2},
             _Headers2, Body2}} -> Body2;
        _Else2 ->
        case     httpc:request(get,{string:concat(
             BaseURL,"-1.txt"), []},[], []) of
        {ok, {{_Version3, 200, _ReasonPhrase3},
             _Headers3, Body3}} -> Body3;
        _Else3 ->
        case     httpc:request(get,{string:concat(
             BaseURL,"-2.txt"), []},[], []) of
        {ok, {{_Version4, 200, _ReasonPhrase4},
             _Headers4, Body4}} -> Body4;
        _Else4 ->
        case     httpc:request(get,{string:concat(
             BaseURL,"-3.txt"), []},[], []) of
        {ok, {{_Version5, 200, _ReasonPhrase5},
             _Headers5, Body5}} -> Body5;
        _Else5 ->
        case     httpc:request(get,{string:concat(
             BaseURL,"-4.txt"), []},[], []) of
        {ok, {{_Version6, 200, _ReasonPhrase6},
             _Headers6, Body6}} -> Body6;
        _Else6 ->
```

```
        case     httpc:request(get,{string:concat(BaseURL,"-5.txt"), []},[], []) of
        ok, {{_Version7, 200, _ReasonPhrase7}, _Headers7, Body7}} -> Body7;
        _Else7 ->
        case     httpc:request(get,{string:concat(BaseURL,"-6.txt"), []},[], []) of
        {ok, {{_Version8, 200, _ReasonPhrase8}, _Headers8, Body8}} -> Body8;
        _Else8 ->
        case     httpc:request(get,{string:concat(BaseURL,"-7.txt"), []},[], []) of
        {ok, {{_Version9, 200, _ReasonPhrase9}, _Headers9, Body9}} -> Body9;
        _Else9 ->
        case     httpc:request(get,{string:concat(BaseURL,"-8.txt"), []},[], []) of
        {ok, {{_Version10, 200, _ReasonPhrase10}, _Headers10, Body10}} -> Body10;
        _Else10 ->
        case     httpc:request(get,{string:concat(BaseURL,"-9.txt"), []},[], []) of
        {ok, {{_Version11, 200, _ReasonPhrase11}, _Headers11, Body11}} -> Body11;
        _Else11 ->"error:␣no␣txt␣file" end
        end
        end
        end
        end
        end
        end
        end
        end
        end
        end.

construct_extension(FileNum,[_])-> string:concat(string:concat("/0/",FileNum),".txt");
construct_extension([_],FileNum)->string:concat(string:concat(FileNum, "/"), string:concat(FileNum,".txt"));
construct_extension([Head|Tail],FileNum)->
        string:concat(string:concat([Head],"/") , construct_extension(Tail, FileNum)).
```

III. Document Preprocessing

After documents are downloaded and stored, they can be partially processed, removing prefaces, introductions etc, and splitting them into chapters, each of which is stored as a separate file, in preparation for parsing. This partial processing is accomplished using the `process_file` function of the `paragraph` module.

```
-module(paragraph).
-export([process_file/2]).
-import(re,[split/3,replace/4]).
%document storage location: C:\Users\Grant\Erlang Workspace\DocumentProcessor\Documents

%FilePath is the directory that the file is in. Name is the name of the file, without extension ("practice", "testFile", etc).
process_file(FilePath,Name)->
        {ok, File} = file:read_file(string:concat(string:concat(FilePath,Name),".txt")),
        process_paragraphs(remove_rn(File),Name).

remove_preface_etc(FullFilePath)->
        {ok, File} = file:read_file(FullFilePath),
        FileString = binary_to_list(File),
        TrimmedFile = string:substr(FileString, get_start(FileString)),
        file:write_file(FullFilePath, TrimmedFile).

get_start(File)->
%%      Starters = ["\r\n\r\nCHAPTER␣1","\r\n\r\nCHAPTER␣ONE","\r\n\r\nCHAPTER␣I", "\r\n\r\nSTAVE␣ONE"],
        case string:str(File, "\r\n\r\nCHAPTER␣1") of
        0 ->
        case string:str(File,"\r\n\r\nCHAPTER␣ONE") of
        0 ->
        case string:str(File, "\r\n\r\nCHAPTER␣I") of
        0 ->
        case string:str(File, "\r\n\r\nSTAVE␣ONE") of
        0->1;
        N->N end;
N->N end;
N->N end;
N->N end.

process_paragraphs(Document,Name)->
        Paragraphs = split(Document, "CHAPTER",[{return,list}]),
        write_files(lists:map(fun remove_rn/1, Paragraphs),string:concat(Name,"_"),"0").

%DIR is the location that you want to write the individual paragraph files to.
write_files([],_,_)->ok;
write_files([ Head |Paragraphs], Name, Num)->
        DIR = "C:\\Users\\Grant\\Erlang␣Workspace\\DocumentProcessor\\Documents\\",  %%Where you want the PARSED documents to be stored
        file:write_file(string:concat(string:concat(DIR, string:concat(Name, Num)),".txt"), Head),
        {N,_} = string:to_integer(Num),
        write_files(Paragraphs, Name, integer_to_list(N+1)).

remove_rn(Document)->
         replace(Document,"\r\n","␣",[{return,list}]).
```

IV. Text Parsing

Once the document has been preprocessed, they should be turned into more structured data by extracting words and special characters. The user can call `chapter_stats` to compute the statistics for a chapter. For this step we use a helper function `isalpha`, which returns whether the input character is a letter (a-z, A-Z), a special character (comma, semicolon, exclamation mark, ...) or neither.

```
% Input: Character Ch
% Output: true if Ch is a letter; Ch if Ch is a special character; false otherwise
isalpha(Ch) -> case lists:member(Ch, ",;\"!-.") of
  true -> Ch;
 false -> case lists:member(Ch, lists:seq(97, 97+25) ++ lists:seq(65, 65+25)) of
    true -> true;
   false -> false
  end
end.
```

The `parse` function takes a string and turns it into a list of words (i.e. a continuous sequence of characters for which `parse` returns true) and special characters:

```
% Extracts words and special characters.
parse(Text) -> parse_(Text, "", false, []).

parse_("", Word, Alpha, Result) -> Result;
parse_([Ch|Rest], Word, Alpha, Result) ->
 case isalpha(Ch) of
  true -> parse_(Rest, [Ch|Word], true, Result);
 false -> if Alpha -> parse_(Rest, "", false, [{word, lists:reverse(Word)}|Result]);
              true -> parse_(Rest, "", false, Result)
          end;
      N -> if Alpha -> parse_(Rest, "", false, [{char, N}|[{word, lists:reverse(Word)}|Result]]);
              true -> parse_(Rest, "", false, [{char, N}|Result])
          end
 end.
```

Note that the output is reversed. So if we, for example, call `parse` on “Abc, def. GH-IJ kl!” and reverse the output, we get the following.

```
[{word,"Abc"},
 {char,44},
 {word,"def"},
 {char,46},
 {word,"GH"},
 {char,45},
 {word,"IJ"},
 {word,"kl"},
 {char,33}]
```

Since case sensitivity does not really make sense for our purposes, we might modify the function so that all words are lowercased.

The (reversed) result of `parse` can then be used to calculate paragraph statistics using the `stats` function. It calculates how often each token appears, the sum of the lengths of all words, the number of words, and the lengths of the sentences.

```
% Computes basic statistics.
% SentenceLength is reversed.
stats([], CountMap, WordLengthSum, WordCount, SentenceLength)
-> {CountMap, WordLengthSum, WordCount, lists:nthtail(1, SentenceLength)};
stats([T|Rest], CountMap, WordLengthSum, WordCount, SentenceLength)
-> NewCountMap = maps:put(T, maps:get(T, CountMap, 0) + 1, CountMap),
   [CurrentLength|RestSL] = SentenceLength,
   case T of {word, Word} -> stats(Rest, NewCountMap, WordLengthSum + length(Word), WordCount + 1, [CurrentLength+1|RestSL]);
             {char, Ch} when Ch == \$. orelse Ch == \$? orelse Ch == \$! -> stats(Rest, NewCountMap, WordLengthSum, WordCount, [0|SentenceLength]);
             _Else -> stats(Rest, NewCountMap, WordLengthSum, WordCount, SentenceLength)
   end.

% Parse text and compute stats.
stats_(Text) -> stats(lists:reverse(parse(Text)), maps:new(), 0, 0, [0]).
```

The output of `stats(lists:reverse(parse(‘‘Abc, def. GH-IJ kl!’’)), maps:new(), 0, 0, [0])` is, for example:

```
{#{{char,33} => 1,
   {char,44} => 1,
   {char,45} => 1,
   {char,46} => 1,
   {word,"Abc"} => 1,
   {word,"GH"} => 1,
   {word,"IJ"} => 1,
   {word,"def"} => 1,
   {word,"kl"} => 1},
 12,5,
 [3,2]}
```

This function is called once for each paragraph, and the results are combined in a function called `chapter_stats`, which computes the statistics of a chapter. They are then used as input for the artificial neural network. We have currently decided on the following characteristics; they have already been used in [2]:

- type-token-ratio
- mean word length
- mean sentence length
- standard deviation of sentence length
- mean paragraph length
- chapter length
- number of commas, semicolons, quotation marks, exclamation marks, hyphens, “and”s, “but”s, “however”s, “if”s, “that”s, “more”s, “must”s, “might”s, “this”s, “very”s per word.

Thus, the input for the network is a 21-tuple of real numbers.

The code for `chapter_stats` is as follows.

```
% The main function for text parsing. Called by the user.
% Calculate the statistics of a chapter.
% Paragraphs must be separated by \r\n\r\n.
chapter_stats(Text)
  -> Paragraphs = re:split(Text, "\r\n\r\n",[{return, list}]),
     ParagraphStats = lists:map(fun stats_/1, Paragraphs),
     {CountMaps, WordLengthSums, WordCounts, SentenceLengths} = unzip4(ParagraphStats),
     CountMap = map_merge(CountMaps),
     WordLengthSum = lists:sum(WordLengthSums),
     WordCount = lists:sum(WordCounts),
     SentenceLength = lists:append(SentenceLengths),
     MeanSentenceLength = mean(SentenceLength),
     {unique_words(CountMap) / WordCount, % type-token-ratio
     WordLengthSum / WordCount, % mean word length
     MeanSentenceLength, % mean sentence length
     standard_deviation(SentenceLength, MeanSentenceLength), % sd of sentence length
     mean(WordCounts), % mean paragraph length
     WordCount, % chapter length
     maps:get({char, $,}, CountMap, 0) / WordCount, % comma density
     maps:get({char, $;}, CountMap, 0) / WordCount, % semicolon density
     maps:get({char, $\"}, CountMap, 0) / WordCount, % quotation mark density
     maps:get({char, $!}, CountMap, 0) / WordCount, % exclamation mark density
     maps:get({char, $-}, CountMap, 0) / WordCount, % hyphen density
     maps:get({word, "and"}, CountMap, 0) / WordCount, % "and" density
     maps:get({word, "but"}, CountMap, 0) / WordCount, % "but" density
     maps:get({word, "however"}, CountMap, 0) / WordCount, % "however" density
     maps:get({word, "if"}, CountMap, 0) / WordCount, % "if" density
     maps:get({word, "that"}, CountMap, 0) / WordCount, % "that" density
     maps:get({word, "more"}, CountMap, 0) / WordCount, % "more" density
     maps:get({word, "must"}, CountMap, 0) / WordCount,
     maps:get({word, "might"}, CountMap, 0) / WordCount,
     maps:get({word, "this"}, CountMap, 0) / WordCount,
     maps:get({word, "very"}, CountMap, 0) / WordCount}.
```

V. The Backpropagation Algorithm

The backpropagation algorithm is used to train the artificial neural network so that it can later be used to classify documents. The theory behind it has already been described in Section IV-A of the background research report. We give a very brief summary of what has to be computed: If $L_1, \dots, L_k$ are the layers of a neural network, if $p = |L_1|$, $r = |L_k|$, $L_k = \{B_1, \dots, B_r\}$, and if (X, Y) is a training sample with $X = (x_1, \dots, x_p)$, $Y = (y_1, \dots, y_r)$, then compute the output O_s of every node s using the feedforward algorithm (Section

VI). Compute the δ-values and the gradients for the last layer using the equations

$$\delta_{B_j} = (O_{B_j} - y_j) O_{B_j} (1 - O_{B_j}),$$
$$\frac{\partial E}{\partial w_{tB_j}} = \delta_{B_j} O_t$$

for all $j \in \{1, \dots, r\}$ and every $t \in L_{k-1}$. The term w_{ab} denotes the weight of the arc (a, b), where a and b are nodes. Then compute for all $s \in L_{j-1}$ and $t \in L_j$

$$\delta_t = O_t(1 - O_t) \sum_{u \in L_{j+1}} \delta_u w_{tu},$$
$$\frac{\partial E}{\partial w_{st}} = \delta_t O_s,$$

iteratively for $j = k-1, \dots, 2$. Finally update

$$w_{ij} := w_{ij} - \alpha \frac{\partial E}{\partial w_{ij}}$$

for all arcs (i, j), where $\alpha > 0$ is the learning rate.

We decided to first compute all δ-values and postpone the computation of the gradients to when the weights are updated.

A. Backpropagation: A Sequential Implementation

The sequential backpropagation was implemented in a function `train_network` that takes a neural network and a training sample (X, Y) and returns a new neural network with updated weights so that the regression function value at X is now closer to Y.

```
1 % Returns the network with updated weights.
2 train_network({network, Layers, Layersize, Weights}, X,
     Y)
3 -> Output = feed_forward(#network{layers=Layers,
      layersize=Layersize, weights=lists:append(Weights)},
      X),
4    Deltas = compute_deltas_init(lists:reverse(Weights),
         lists:reverse(Output), Y),
5    Alpha = 0.08
6    NewWeights = update_weights(Alpha, Output, Deltas,
         Weights),
7    #network{layers=Layers, layersize=Layersize, weights=
         NewWeights}.
```

The function first uses the feedforward algorithm (which was already shown in Section IV of the proof-of-concept) to compute the output of the network given X. This output, together with the target value Y can be used to compute the δ-values for each node—this is done in `compute_deltas_init`. Finally the function calls `update_weights` to update the weights and returns the new network. The learning rate `Alpha` can be adjusted.

Note that, unlike in the proof-of-concept, the weights of a network are now grouped by layer so that the field `weights` is a list of lists. This is why we call `lists:append` in line 3. We also have a function `feed_forward_` which does that automatically.

We now show the `compute_deltas_init` function. It computes the δ-values for the nodes of the output layer and then calls `compute_deltas` which recursively computes the δ-values of all other layers. This “distinction” between the last layer and the other layers is due to the fact that the δ-values of the output layer depend on the target value, whereas those of the other layers do not (at least not directly; cf. the equations above).

```
% Computes the delta values. The layers of the input
    network must be reversed.
compute_deltas_init(Weights, [Output|ORest], Target)
  -> Deltas = map(fun (X, Y) -> (X-Y)*X*(1-X) end,
       Output, Target),
     compute_deltas([Deltas], Weights, ORest).

% Recursively computes the delta values. The layers of
    the input network must be reversed.
% Requires the delta values of the next layer.
% Called by compute_deltas_init.
compute_deltas(Deltas, [], []) -> Deltas;
compute_deltas([Deltas|DRest], [Weights|WRest], [Output|
    ORest])
  -> ThisLayerSize = length(Weights),
     PreviousLayerSize = length(Deltas),
     AAWeights = partitionList(Weights, ThisLayerSize),
     NewDeltas = map(fun (O, NodeWeights) -> O*(1-O) *
          lists:foldl(fun ({X, Y}, Acc) -> Acc + X*Y end,
           0, myzip(Deltas, NodeWeights)) end, Output,
          AAWeights), % The delta values for this layer.
     compute_deltas([NewDeltas|[Deltas|DRest]], WRest,
          ORest).
```

Note that, since backpropagation works from the last layer to the first, we require that the list of weights and output values is reversed. This makes recursion easier.

With the output and the δ-values we can easily update the weights:

```
update_weights(Alpha, Output, Delta, Weights)
  -> map3(fun (Os, Ds, Ws) -> map3(fun (O, D, W) -> W -
       Alpha * D * O end, repeat(Os, length(Ds)),
       repeat_each(Ds, length(Os)), Ws) end, Output,
       lists:nthtail(1, Delta), Weights).
```

With these function definitions, if we want to train a neural network, we only have to repeatedly call `train_network`.

B. Backpropagation: Training Session Parallelism

Recall the idea behind training session parallelism (cf. Section IV-B of the background research report): Start with an artificial neural network and a training data set $X_1, \dots, X_n \in \mathbb{R}^p$, $Y_1, \dots, Y_n \in \mathbb{R}^r$. For each $i = 1, \dots, n$, train the network with the sample (X_i, Y_i) (using `train_network`). If the training error is sufficiently small or after a predefined number of iterations, stop. Otherwise, repeat the previous step. This entire procedure is known as a training session. With training session parallelism, several such training sessions are performed in parallel, each with different initial (random) neural networks. Once all sessions have terminated, we can choose the network with the smallest regression error on the training data.

Since we already have the `train_network` function and given Erlang’s concurrency features, training session parallelism can be implemented easily.

```
% Train N networks with training data Xs, Ys (lists) and
     return the best network.
training_session_parallelism(Xs, Ys, N)
  -> training_session_parallelism_(Xs, Ys, N, N).

% Train N networks with training data Xs, Ys (lists) and
     return the best network.
% N-K sessions have already been started.
```

```
7 training_session_parallelism_(Xs, Ys, N, 0)
8   -> receive_results(Xs, Ys, N); % all sessions have been started; now collect the results
9 training_session_parallelism_(Xs, Ys, N, K)
10   -> InputLayerSize = length(lists:nth(1, Xs)), % the size of the first and last layer are determined by the training data
11      OutputLayerSize = length(lists:nth(1, Ys)),
12      HiddenLayerSize = round(1.5*InputLayerSize), % can be adjusted
13      Network = #network{layers=3, layersize=[InputLayerSize,HiddenLayerSize, OutputLayerSize], weights=[random_list(InputLayerSize*HiddenLayerSize), random_list(HiddenLayerSize*OutputLayerSize)]}, % can be adjusted
14      NumberOfIterations = 500000, % can be adjusted
15      spawn(ann, train_concurrently, [self(), Network, myzip(Xs, Ys), NumberOfIterations]), % start training session in new thread
16      training_session_parallelism_(Xs, Ys, N, K-1). % start the remaining K-1 sessions recursively
```

In this case we use a neural network with three layers. The size of the first layer is the dimension of the input data; the size of the last layer is the dimension of the target data—you have no choice. However, you could decide to have more than one hidden layer or to change the size of the hidden layer. The initial weights are chosen uniformly at random from the interval $(0,1) \subseteq \mathbb{R}$ using `random_list`, which we will define later. Additionally you could decide to change the number of iterations for each training session.

After the (partly) random neural network is created, `training_session_parallelism_` creates a new Erlang process (line 15; cf. Section II of the background research report; note that `train_concurrently` is in the `ann` module) which does one training session with that network and the training data. The new process is also given the ID of the current process (`self()`) so that it will be able to send back the results. Then the remaining $K-1$ sessions are started recursively. Once all sessions are started, it calls `receive_results` to *receive* the N networks and choose the best:

```
% Collect N results. Xs, Ys: training data.
receive_results(Xs, Ys, N) -> receive_results_(Xs, Ys, N, 0, 9999999999).

receive_results_(Xs, Ys, 0, BestNetwork, SmallestError)
  -> {BestNetwork, SmallestError}; % when all networks have been received, return the one with the smallest training error
receive_results_(Xs, Ys, N, BestNetwork, SmallestError)
  -> receive
       NewNetwork
         -> NewError = compute_training_error(NewNetwork, Xs, Ys),
            if NewError < SmallestError ->
                 receive_results_(Xs, Ys, N-1, NewNetwork, NewError);
               true -> receive_results_(Xs, Ys, N-1, BestNetwork, SmallestError)
            end
     end.
```

In order to complete the code, all that is left to do is define `train_concurrently`. This function repeatedly calls `train_network` from V-A and after the final iteration it sends the network back to the "main process" using the `!` operator.

```
train_concurrently(Process, Network, TrainingSamples, 0)
  -> Process ! Network;
train_concurrently(Process, Network, TrainingSamples, N)
  -> train_concurrently(Process, lists:foldl(fun ({X, Y}, Acc) -> train_network(Acc, X, Y) end, Network, TrainingSamples), TrainingSamples, N-1).
```

The call to `lists:foldl` does an entire round of training with all the samples being used once. Note that `TrainingSamples` is a tuple, with the first element being a list of the X values of the training samples, and the second element being a list of the Y values. It comes from the call to `myzip` in line 15 of the code for `training_session_parallelism_`.

Just like `myzip` and other helper functions, the code for `compute_training_error` is shown in VII.

VI. The Feedforward Algorithm

In this section we present our implementation of the feedforward algorithm for multilayer perceptrons as it was implicitly given in section III-A of our background research report.[1] It is not only needed to classify the unknown documents, but also for the backpropagation algorithm, which is why our implementation returns the output of every node instead of just the output of the nodes in the output layer.

We first define the data structure for a neural network. It consists of the number of layers (an integer), the size of each layer (a list of integers), and the weights (a list of integers).

The weights must be carefully ordered: If layer l has n nodes $(l,1),\ldots,(l,n)$ and layer $l+1$ has m nodes $(l+1,1),\ldots,(l+1,m)$, we use the following ordering of the arcs in $L_l \times L_{l+1}$ $(1 \leq l < k)$

$$((l,1),(l+1,1)),\ldots,((l,n),(l+1,1)),((l,1),(l+1,2)),\ldots ((l,n),(l+1,m))$$

to list the weights of the arcs in $L_1 \times L_2, \ldots, L_{k-1} \times L_k$ (in this order).

The implementation of the feedforward algorithm works by recursion over the layers: We want to compute the output of a k-layer network when given some input. Once we have computed the output of the first layer (by applying the sigmoid function), we combine them according to the weights of the arcs between the first and the second layer. This gives us the input to the second layer, and we have reduced the task to computing the output of a $(k-1)$-layer network.

```
-record(network, {layers, layersize, weights}).

testann() -> #network{layers=3, layersize=[2,1,2], weights=[[1,1],[1,1]]}.

% Weights is a list of lists
feed_forward_({network, Layers, Layersize, Weights}, Input) -> feed_forward(#network{layers=Layers, layersize=Layersize, weights=lists:append(Weights)}, Input).

% Weights is a list
feed_forward({network, Layers, Layersize, Weights}, Input)
-> if Layers == 1 -> [lists:map(fun sigmoid/1, Input)];
            true -> InputSize = length(Input),
```

[1]We use the same notation.

```
                        Output = lists:map(fun sigmoid/1,
                             Input),
                        NextLayerSize = lists:nth(2,
                             Layersize),
                        NextInput = compute_next_input(
                             Output, NextLayerSize, Weights)
                             ,
                        NewWeights = lists:nthtail(length(
                             Output) * NextLayerSize,
                             Weights),
                        [Output|feed_forward(#network{
                             layers = Layers-1, layersize =
                             lists:nthtail(1, Layersize),
                             weights = NewWeights},
                             NextInput)]
    end.

% Combines the output of the current layer to compute
     the input to the next layer.
compute_next_input(Output, 0, Weights) -> [];
compute_next_input(Output, NextLayerSize, Weights)
-> NextInput = lists:foldl(fun ({Out, Wei}, Sum) -> Sum
     + Out * Wei end, 0, myzip(Output, Weights)),
   [NextInput|compute_next_input(Output, NextLayerSize
        -1, lists:nthtail(length(Output), Weights))].

% Like the regular zip, but the lists can have different
      lengths.
myzip([], Bs) -> [];
myzip(As, []) -> [];
myzip([A|As], [B|Bs]) -> [{A, B}|myzip(As,Bs)].

sigmoid(X) -> 1 / (1 + math:exp(-X)).
```

VII. Helper Functions

Here we show the code for some helper functions. They are minor extensions to existing Erlang functions or easy functions which are not inherently related to neural networks, or both. The provided comments should be enough to understand the code or at least their purpose.

```
% Merge all maps, adding the values of common keys.
map_merge([]) -> maps:new();
map_merge([X|[]]) -> X;
map_merge([X|[Y|Rest]])
  -> map_merge([maps:fold(fun (K, V, AccIn) -> maps:put(
       K, maps:get(K, AccIn, 0) + V, AccIn) end, X, Y)|
       Rest]).
```

```
% Like the regular unzip, but List is a list of 4-tuples
unzip4(List) -> lists:foldr(fun ({A, B, C, D}, {As, Bs,
     Cs, Ds}) -> {[A|As], [B|Bs], [C|Cs], [D|Ds]} end,
     {[],[],[],[]}, List).
```

```
% Counts the number of unique words (entries with key {
     word, _}).
unique_words(CountMap)
  -> lists:foldl(fun (X, Acc)
                 -> case X of {word, _} -> Acc + 1;
                                   _Else -> Acc end end,
                 0, maps:keys(CountMap)).
```

```
% Compute the mean of the elements in Xs.
mean(Xs) -> lists:foldr(fun (X, Acc) -> X + Acc end, 0,
     Xs) / length(Xs).
```

```
% Compute the standard deviation of the elements in Xs.
% XMean = mean(Xs)
standard_deviation(Xs, XMean) -> math:sqrt(lists:foldr(
     fun (X, Acc) -> (X-XMean)*(X-XMean) + Acc end, 0, Xs
     ) / length(Xs)).
```

```
% Compute the regression error for the training data set
compute_training_error(Network, [], []) -> 0;
compute_training_error(Network, [X|Xs], [Y|Ys])
  -> Output = lists:last(feed_forward_(Network, X)),
     lists:sum(map(fun (T, O) -> (T-O)*(T-O) end, Y,
          Output)) + compute_training_error(Network, Xs,
          Ys).
```

```
% N-fold concatenation of Xs
repeat(Xs, 0) -> [];
repeat(Xs, N) -> Xs ++ repeat(Xs, N-1).
```

```
% [x1,...,xK] -> [x1,...,x1,x2,...,x2,...,xK,...,xK]
repeat_each([], N) -> [];
repeat_each([X|Xs], N) -> repeat([X], N) ++ repeat_each(
     Xs, N).
```

```
% Partition the list into N smaller lists by assigning
     the elements cyclically.
% partitionList([1,2,3,4,5,6,7,8,9], 3) = [[1,4,7],
     [2,5,8], [3,6,9]]
partitionList(List, N) -> partitionList_(List, N, N).
partitionList_(List, N, 0) -> [];
partitionList_(List, N, K) -> [partListHelp(List, N, 0)|
     partitionList_(lists:nthtail(1, List), N, K-1)].
partListHelp([], N, M) -> [];
partListHelp([X|Xs], N, 0) -> [X|partListHelp(Xs, N, N
     -1)];
partListHelp([X|Xs], N, M) -> partListHelp(Xs, N, M-1).
```

```
% Like the regular map, but with two lists.
map(Function, [], Ys) -> [];
map(Function, Xs, []) -> [];
map(Function, [X|Xs], [Y|Ys])
  -> [Function(X, Y)|map(Function,Xs,Ys)].
```

```
% Like the regular map, but with three lists.
map3(Function, [], Ys, Zs) -> [];
map3(Function, Xs, [], Zs) -> [];
map3(Function, Xs, Ys, []) -> [];
map3(Function, [X|Xs], [Y|Ys], [Z|Zs])
  -> [Function(X, Y, Z)|map3(Function,Xs,Ys, Zs)].
```

```
% Like the regular zip, but the lists can have different
      lengths.
myzip([], Bs) -> [];
myzip(As, []) -> [];
myzip([A|As], [B|Bs]) -> [{A, B}|myzip(As,Bs)].
```

```
% Return a list of length N, each element being a random
      number with uniform ditribution on the interval
     (0,1).
random_list(0) -> [];
random_list(N) -> [random:uniform()|random_list(N-1)].
```

```
% The sigmoid function.
sigmoid(X) -> 1 / (1 + math:exp(-X)).
```

VIII. Neural Network Demonstration

We give a small example on how the code from Sections V and VI can be used to train a neural network.

Suppose we want to approximate a function $f : \mathbb{R}^2 \to \mathbb{R}$ and we only know the values

$$f(0,0) = 0,$$
$$f(1,2) = 0.5.$$

The corresponding training data set is `Xs = [[0,0],[1,2]]` and `Ys = [[0],[0.5]]`. Suppose we want to train two networks simultaneously and choose the one with the smaller training error. We can do this as follows:

```
1 Eshell V7.1
2 (ann@User)1> {Network, TrainingError} = ann:
      training_session_parallelism([[0,0],[1,2]],
      [[0],[0.5]], 2).
3 {{network,3,
4            [2,3,1],
5            [[-0.19834785545597572, 2.159548206120467,
                  3.1102682198495764,
6              -4.345304952255812, 3.9994175264983896,
                   -5.033623392917592],
7             [4.467630279109158, -9.770363242212438,
                  -11.638732867738266]]},
8  1.158818489081351e-4}
```

We see that we get a small network with three layers, and the training error as computed by `compute_training_error` is about $1.16 \cdot 10^{-4}$. We can check the goodness of the approximation on the training data using `feed_forward_`:

```
1 (ann@User)2> ann:feed_forward_(Network, [0,0]).
2 [[0.5, 0.5],
3  [0.7272272880121131, 0.3503460704822918,
       0.3735299139496292],
4  [0.010756390215779843]]
```

As already said, this computes the output of all nodes ordered by layer. In this case, the regression function value at $(0,0)$ is about 0.01 (line 4), which is close to 0.

```
1 (ann@User)3> ann:feed_forward_(Network, [1,2]).
2 [[0.7310585786300049, 0.8807970779778823],
3  [0.8528485744767521, 0.17457504547237046,
       0.1809695787141338],
4  [0.49957348102737303]]
```

The regression function value at $(1,2)$ is about 0.4996 (line 4), which is very close to 0.5.

This should be evidence enough to show that the neural network code works and is very easy to use.

Conclusion

We have developed our project into a useful tool for document classification, which even features generally usable, concurrent code for neural networks. With a little more time and effort we could have implemented node parallelism and significantly increased the usability of the provided code by requiring less user interaction. Additionally the training efficiency and the goodness of approximation could certainly be increased by tweaking the network structure and the choice of the learning rate. Still we think that we have created a valuable basis for anyone who is interested in the intersection between neural networks and concurrent programming with Erlang.

References

[1] Our GitHub Repository. https://github.com/guenterg/cpsc311project, December 7, 2015.

[2] R. C. He and K. Rasheed. Using Machine Learning Techniques for Stylometry. http://www2.tcs.ifi.lmu.de/~ramyaa/publications/stylometry.pdf, November 25, 2015.