Deep Neuronal Filter

Generated by Doxygen 1.9.8

Chapter 1

Deep Neuronal Filter (DNF)

Libtorch version

https://journals.plos.org/plosone/article?id=10.1371/journal.pone.0277974

A noise reduction filter using deep networks in autoencoder configuration.

Chapter 2

Hierarchical Index

2.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Chapter 3

Class Index

3.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 4

File Index

4.1 File List

Here is a list of all documented files with brief descriptions:

Chapter 5

Class Documentation

5.1 DelayLine Class Reference

Public Member Functions

- void **init** (size_t delaySamples)
- float **process** (float input)
- float **get** (int i) const
- float **getNewest** () const

The documentation for this class was generated from the following file:

- dnf.h

5.2 DNF Class Reference

Main Deep Neuronal Network main class.

`#include <dnf.h>`

Classes

- struct Net

Public Types

- enum ActMethod { **Act_Sigmoid** = 1 , **Act_Tanh** = 2 , **Act_ReLU** = 3 , **Act_NONE** = 0 }

  Options for activation functions of all neurons in the network.

Public Member Functions

- DNF (const int nLayers, const int nTaps, const float samplingrate, const ActMethod am=Act_Tanh, const bool tryGPU=false)

  Constructor which sets up the delay lines, network layers and also calculates the number of neurons per layer so that the final layer always just has one neuron.
- void **setLearningRate** (float mu)
- float filter (const float signal, const float noise)

  Realtime sample by sample filtering operation.
- int getSignalDelaySteps () const

  Returns the length of the delay line which delays the signal polluted with noise.
- float getDelayedSignal () const

  Returns the delayed with noise polluted signal by the delay indicated by getSignalDelaySteps().
- float getRemover () const

  Returns the remover signal.
- float getOutput () const

  Returns the output of the DNF: the the noise free signal.
- ~**DNF** ()

  Frees the memory used by the DNF.
- std::vector< float > **getLayerWeightDistances** () const

  Gets the weight distances per layer.
- float **getWeightDistance** () const

  Gets the overall weight distsance.

5.2.1 Detailed Description

Main Deep Neuronal Network main class.

It's designed to be as simple as possible with only a few parameters as possible.

5.2.2 Constructor & Destructor Documentation

5.2.2.1 DNF()

```
DNF::DNF (
            const int nLayers,
            const int nTaps,
            const float samplingrate,
            const ActMethod am = Act_Tanh,
            const bool tryGPU = false )  [inline]
```

Constructor which sets up the delay lines, network layers and also calculates the number of neurons per layer so that the final layer always just has one neuron.

Parameters

nLayers	Number of layers
nTaps	Number of taps for the delay line feeding into the 1st layer
samplingrate	Sampling rate of the signals used in Hz.
am	The activation function for the neurons. Default is tanh.
tryGPU	Tries to do the learning on the GPU.

5.2.3 Member Function Documentation

5.2.3.1 filter()

```
float DNF::filter (
            const float signal,
            const float noise )
```

Realtime sample by sample filtering operation.

Parameters

signal	The signal contaminated with noise. Should be less than one.
noise	The reference noise. Should be less than one.

Returns

The filtered signal where the noise has been removed by the DNF.

5.2.3.2 getDelayedSignal()

```
float DNF::getDelayedSignal ( ) const  [inline]
```

Returns the delayed with noise polluted signal by the delay indicated by getSignalDelaySteps().

Returns

The delayed noise polluted signal sample.

5.2.3.3 getOutput()

```
float DNF::getOutput ( ) const  [inline]
```

Returns the output of the DNF: the the noise free signal.

Returns

The current output of the DNF which is idential to filter().

5.2.3.4 getRemover()

```
float DNF::getRemover ( ) const  [inline]
```

Returns the remover signal.

Returns

The current remover signal sample.

5.2.3.5 getSignalDelaySteps()

```
int DNF::getSignalDelaySteps ( ) const  [inline]
```

Returns the length of the delay line which delays the signal polluted with noise.

Returns

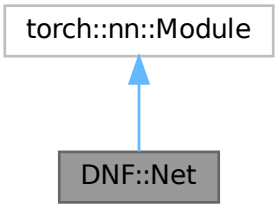
Number of delay steps in samples.

The documentation for this class was generated from the following file:

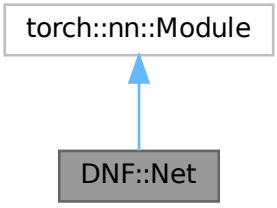
- dnf.h

5.3 DNF::Net Struct Reference

Inheritance diagram for DNF::Net:



Collaboration diagram for DNF::Net:



Public Member Functions

- **Net** (int nLayers, int nInput, bool withBias=false)
- torch::Tensor **forward** (torch::Tensor x, ActMethod am)

Public Attributes

- std::vector< torch::nn::Linear > **fc**

The documentation for this struct was generated from the following file:

- dnf.h

Chapter 6

File Documentation

6.1 dnf.h

```
00001
00007 #ifndef _DNF_H
00008 #define _DNF_H
00009
00010 #include <stdio.h>
00011 #include <stdlib.h>
00012 #include <math.h>
00013 #include <assert.h>
00014 #include <torch/torch.h>
00015 #include <thread>
00016 #include <iostream>
00017 #include <deque>
00018
00019 #ifdef NDEBUG
00020 const bool debugOutput = false;
00021 #else
00022 const bool debugOutput = true;
00023 #endif
00024
00025 class DelayLine {
00026 public:
00027     void init(size_t delaySamples)
00028     {
00029         delaySamples_ = delaySamples;
00030         buffer_ = std::deque<float>(delaySamples_, 0.0f);
00031     }
00032
00033     float process(float input) {
00034         // Output is the oldest value (front of deque)
00035         float output = buffer_.front();
00036         buffer_.pop_front();
00037
00038         // Push new input to the back
00039         buffer_.push_back(input);
00040
00041         return output;
00042     }
00043
00044     float get(int i) const {
00045     return buffer_[i];
00046     }
00047
00048     float getNewest() const {
00049     return buffer_.back();
00050     }
00051
00052 private:
00053     size_t delaySamples_ = 0;
00054     std::deque<float> buffer_;
00055 };
00056
00057
00063 class DNF {
00064 public:
00065
00069     enum ActMethod {Act_Sigmoid = 1, Act_Tanh = 2, Act_ReLU = 3, Act_NONE = 0};
00070
00071     struct Net : public torch::nn::Module {
```

```
00072     std::vector<torch::nn::Linear> fc;
00073
00074     Net(int nLayers, int nInput, bool withBias = false) {
00075         // calc an exp reduction of the numbers always reaching 1
00076         const float b = (float)exp(log(nInput)/(nLayers-1));
00077         int inputNeurons = nInput;
00078         for(int i=1;i<nLayers;i++) {
00079         int outputNeurons = (int)ceil(nInput / pow(b,i));
00080         if (i == (nLayers-1)) outputNeurons = 1;
00081         char tmp[256];
00082         sprintf(tmp,"fc%d_%d_%d",i,inputNeurons,outputNeurons);
00083         if (debugOutput)
00084             fprintf(stderr,"Creating FC layer: %s\n",tmp);
00085         torch::nn::Linear ll = register_module(
00086             tmp,
00087             torch::nn::Linear(torch::nn::LinearOptions(inputNeurons, outputNeurons).bias(withBias))
00088             );
00089         torch::nn::init::xavier_uniform_(ll->weight,xavierGain);
00090         if (withBias) torch::nn::init::constant_(ll->bias, 0.0);
00091         fc.push_back(ll);
00092         if (1 == outputNeurons) break;
00093         inputNeurons = outputNeurons;
00094         }
00095     }
00096
00097     torch::Tensor forward(torch::Tensor x, ActMethod am) {
00098         for(auto& f:fc) {
00099         switch (am) {
00100         default:
00101         case Act_Tanh:
00102             x = torch::atan(f->forward(x));
00103             break;
00104         case Act_Sigmoid:
00105             x = torch::sigmoid(f->forward(x));
00106             break;
00107         case Act_ReLU:
00108             x = torch::relu(f->forward(x));
00109             break;
00110         case Act_NONE:
00111             x = f->forward(x);
00112             break;
00113         }
00114         }
00115         return x;
00116     }
00117
00118     };
00119
00130     DNF(const int nLayers,
00131     const int nTaps,
00132     const float samplingrate,
00133     const ActMethod am = Act_Tanh,
00134     const bool tryGPU = false
00135     ) : noiseDelayLineLength(nTaps),
00136         signalDelayLineLength(noiseDelayLineLength / 2),
00137         fs(samplingrate),
00138         actMethod(am)
00139     {
00140
00141     signal_delayLine.init(signalDelayLineLength);
00142     noise_delayLine.init(noiseDelayLineLength);
00143
00144     torch::manual_seed(42);
00145
00146     torch::DeviceType device_type;
00147     if (tryGPU && torch::cuda::is_available()) {
00148         std::cout « "CUDA available. Training on GPU." « std::endl;
00149         device_type = torch::kCUDA;
00150         device = torch::Device(device_type);
00151     }
00152
00153     model = new Net(nLayers,nTaps);
00154     model->to(device);
00155     model->train();
00156
00157     optimizer = new torch::optim::SGD(model->parameters(), 0);
00158     saveInitialParameters();
00159     }
00160
00161     void setLearningRate(float mu) {
00162     for (auto& group : optimizer->param_groups()) {
00163             static_cast<torch::optim::SGDOptions&>(group.options()).lr(mu);
00164         }
00165     }
00166
00173     float filter(const float signal, const float noise);
00174
```

```
00180     inline int getSignalDelaySteps() const {
00181     return signalDelayLineLength;
00182     }
00183
00189     inline float getDelayedSignal() const {
00190     return signal_delayLine.get(0);
00191     }
00192
00197     inline float getRemover() const {
00198     return remover;
00199     }
00200
00206     inline float getOutput() const {
00207     return f_nn;
00208     }
00209
00213     ~DNF() {
00214     delete optimizer;
00215     delete model;
00216     }
00217
00221     std::vector<float> getLayerWeightDistances() const {
00222     return computeLayerDistances();
00223     }
00224
00228     float getWeightDistance() const {
00229     auto dists = computeLayerDistances();
00230     float dsum = 0;
00231     for(const auto& dlayer : dists) {
00232         dsum = dsum + dlayer;
00233     }
00234     return dsum;
00235     }
00236
00237 private:
00238
00239     void saveInitialParameters() {
00240     for (const auto& p : model->parameters()) {
00241
00242         initialParameters.push_back(p.detach().clone());
00243     }
00244     }
00245
00246     std::vector<float> computeLayerDistances() const {
00247     std::vector<float> distances;
00248     int i = 0;
00249     for (const auto& p : model->parameters()) {
00250         torch::Tensor diff = (p - initialParameters[i]).view(-1);
00251         torch::Tensor dist = torch::norm(diff, 2);
00252         distances.push_back(dist.item<float>());
00253         i++;
00254     }
00255     return distances;
00256     }
00257
00258     Net* model = nullptr;
00259     torch::optim::SGD* optimizer = nullptr;
00260     std::vector<torch::Tensor> initialParameters;
00261     const int noiseDelayLineLength;
00262     const int signalDelayLineLength;
00263     const float fs;
00264     const ActMethod actMethod;
00265     DelayLine signal_delayLine;
00266     DelayLine noise_delayLine;
00267     float remover = 0;
00268     float f_nn = 0;
00269     static constexpr double xavierGain = 0.01;
00270     torch::Device device = torch::kCPU;
00271 };
00272
00273 #endif
```

Index