Selected files

6 printable files

Neuron\neuron.c
Neuron\neuron.h
NN231\nn_io.c
NN231\nn.c
NN231\nn.h
NN231\examples.py

Neuron\neuron.c

```c
#include "neuron.h"

#define _USE_MATH_DEFINES
#include <math.h>

#include <stdlib.h>

static void _grad(unsigned long n, double * x,
                  double ya, double yt, double l,
                  double * grad);

static void _norm(unsigned long n, double * v);

/// <summary>
/// Calculates neuron output.
/// </summary>
/// <param name="n"> Neuron dimension: weight count / input count (include shift). </param>
/// <param name="w"> Weights array (include shift. </param>
/// <param name="x"> Input array (include shift). </param>
/// <param name="l"> Activation function smoothing coefficient. </param>
/// <returns> Neuron output. </returns>
double neuron_activate(unsigned long n, double * w, double * x, double l)
{
    // Activation function argument
    double s = 0;
    for (unsigned long i = 0; i < n; i++)
        s += w[i] * x[i];

    // Activation function
    double ya = 1.0 / (1.0 + exp(-1.0 * l * s));

    // Returning
    return ya;
}

/// <summary>
/// Adjusts neuron weights by gradient of Error from w values.
/// </summary>
/// <param name="n"> Neuron dimension: weight count / input count (include shift). </param>
/// <param name="w"> Weights array (include shift). </param>
/// <param name="x"> Input array (include shift). </param>
```

```
42  /// <param name="ya"> Neuron output. </param>
43  /// <param name="yt"> Expected output. </param>
44  /// <param name="l"> Activation function smoothing coefficient: o(s) = 1 / (1 + e ^ (-l * s))
    </param>
45  /// <param name="a"> Weight adjustment length coefficient: wn = wc - a * ngrad </param>
46  void neuron_adjust_weights(unsigned long n,
47                             double * w, double * x,
48                             double ya, double yt, double l, double a)
49  {
50      // Calculate grad
51      double * grad = (double *)malloc(n * sizeof(double));
52      _grad(n, x, ya, yt, l, grad);
53
54      // Normalize
55      _norm(n, grad);
56
57      // Adjust weight
58      for (unsigned long i = 0; i < n; i++)
59          w[i] -= a * grad[i];
60
61      // Free resources
62      free(grad);
63  }
64
65  /// <summary>
66  /// Calculates gradient of E(W).
67  /// Result is stored in grad.
68  /// </summary>
69  /// <param name="n"> Dimension. </param>
70  /// <param name="x"> Neuron input. </param>
71  /// <param name="ya"> Neuron output. </param>
72  /// <param name="yt"> Expected output. </param>
73  /// <param name="l"> Activation function smoothing coefficient. </param>
74  /// <param name="grad"> Array to store gradient coordinates in. </param>
75  static void _grad(unsigned long n, double * x,
76                    double ya, double yt, double l,
77                    double * grad)
78  {
79      for (unsigned long i = 0; i < n; i++)
80          grad[i] = (ya - yt)         // dE/dya
81                  * l * ya * (1 - ya) // dya/ds
82                  * x[i];             // ds/dwi;
83  }
84
85  /// <summary>
86  /// Normalizes given vector.
87  /// </summary>
88  /// <param name="n"> Vector dimension. </param>
89  /// <param name="v"> Vector values. </param>
90  static void _norm(unsigned long n, double * v)
91  {
92      // vector module
93      double m = 0;
94      for (unsigned long i = 0; i < n; i++)
95          m += v[i] * v[i];
96      m = sqrt(m);
```

```
97
98      // normalize
99      for (unsigned long i = 0; i < n; i++)
100         v[i] /= m;
101 }
102
```

Neuron\neuron.h

```
1  #pragma once
2
3  /// <summary>
4  /// Calculates neuron output.
5  /// </summary>
6  /// <param name="n"> Neuron dimension: weight count / input count (include shift). </param>
7  /// <param name="w"> Weights array (include shift. </param>
8  /// <param name="x"> Input array (include shift). </param>
9  /// <param name="l"> Activation function smoothing coefficient. </param>
10 /// <returns> Neuron output. </returns>
11 double neuron_activate(unsigned long n, double * w, double * x, double l);
12
13 /// <summary>
14 /// Adjusts neuron weights by gradient of Error from w values.
15 /// </summary>
16 /// <param name="n"> Neuron dimension: weight count / input count (include shift). </param>
17 /// <param name="w"> Weights array (include shift). </param>
18 /// <param name="x"> Input array (include shift). </param>
19 /// <param name="ya"> Neuron output. </param>
20 /// <param name="yt"> Expected output. </param>
21 /// <param name="l"> Activation function smoothing coefficient: o(s) = 1 / (1 + e ^ (-l * s))
   </param>
22 /// <param name="a"> Weight adjustment length coefficient: wn = wc - a * ngrad </param>
23 void neuron_adjust_weights(unsigned long n,
24                            double * w, double * x,
25                            double ya, double yt, double l, double a);
26
```

NN231\nn_io.c

```
1  #include "nn.h"
2
3  /// <summary>
4  /// Writes neural network to stream in binary format.
5  /// </summary>
6  /// <param name="l"> NN lambda parameter. </param>
7  /// <param name="a"> NN alpha parameter. </param>
8  /// <param name="w12"> Weights of layer1 -> layer2 as matrix. </param>
9  /// <param name="w23"> Weights of layer2 -> layer3 as vector. </param>
10 /// <param name="f"> Output stream. </param>
11 void nn_fwrite(double l, double a,
12                double w12[L1COUNT + 1][L2COUNT], double w23[L2COUNT + 1],
13                FILE * f)
14 {
15     fwrite(&l, sizeof(double), 1, f);
```

```
    fwrite(&a, sizeof(double), 1, f);
    for (int i = 0; i < L1COUNT + 1; i++)
        fwrite(w12[i], sizeof(double), L2COUNT, f);
    fwrite(w23, sizeof(double), L2COUNT + 1, f);
}

/// <summary>
/// Reads neural network from stream.
/// </summary>
/// <param name="l"> Pointer to read NN lambda parameter in. </param>
/// <param name="a"> Pointer to read NN alpha parameter in. </param>
/// <param name="w12"> Matrix to read layer1->layer2 weights in. </param>
/// <param name="w23"> Vector to read layer2->layer3 weights in. </param>
/// <param name="f"> Output stream. </param>
void nn_fread(double * l, double * a,
              double w12[L1COUNT + 1][L2COUNT], double w23[L2COUNT + 1],
              FILE * f)
{
    fread(l, sizeof(double), 1, f);
    fread(a, sizeof(double), 1, f);
    for (int i = 0; i < L1COUNT + 1; i++)
        fread(w12[i], sizeof(double), L2COUNT, f);
    fread(w23, sizeof(double), L2COUNT + 1, f);
}

/// <summary>
/// Writes neural network to stream in text format.
/// </summary>
/// <param name="l"> NN lambda parameter. </param>
/// <param name="a"> NN alpha parameter. </param>
/// <param name="w12"> Weights of layer1 -> layer2 as matrix. </param>
/// <param name="w23"> Weights of layer2 -> layer3 as vector. </param>
/// <param name="f"> Output stream. </param>
void nn_fprint(double l, double a,
               double w12[L1COUNT + 1][L2COUNT], double w23[L2COUNT + 1],
               FILE * stream)
{
    fprintf(stream, "l = %.4lf; a = %.4lf;\n", l, a);

    fprintf(stream, "w12:");
    for (int i = 0; i < L1COUNT + 1; i++)
    {
        fputs("\n|", stream);
        for (int j = 0; j < L2COUNT; j++)
        {
            fprintf(stream, " %.4lf |", w12[i][j]);
        }
    }

    fprintf(stream, "\nw23:\n|");
    for (int i = 0; i < L2COUNT + 1; i++)
    {
        fprintf(stream, " %.4lf |", w23[i]);
    }
    fputc('\n', stream);
}
```

NN231\nn.c

```c
#include "nn.h"

#define _USE_MATH_DEFINES
#include <math.h>

#define SIGMA(x, l) 1.0 / (1.0 + exp(-1.0 * l * x))

/// <summary>
/// NN activation implementation.
/// </summary>
/// <param name="x"> Input signal vector. </param>
/// <param name="w12"> layer1->layer2 weights as matrix. </param>
/// <param name="w23"> layer2->layer3 weights as vector. </param>
/// <param name="l"> NN lambda parameter. </param>
/// <param name="o1"> Vector to write layer1 output. </param>
/// <param name="o2"> Vector to write layer2 output. </param>
/// <param name="oa"> Pointer to write layer3 output. </param>
void nn_activate(double x[L1COUNT],
                 double w12[L1COUNT + 1][L2COUNT], double w23[L2COUNT + 1], double l,
                 double o1[L1COUNT + 1], double o2[L2COUNT + 1], double * oa)
{
    // 1st layer output + shift
    for (int i = 0; i < L1COUNT; i++)
        o1[i] = x[i];
    o1[L1COUNT] = 1.0;

    // Sum for 2nd layer
    double s2[L2COUNT] = { 0.0, 0.0, 0.0 };
    for (int i = 0; i < L2COUNT; i++)
        for (int j = 0; j < L1COUNT + 1; j++)
            s2[i] += o1[j] * w12[j][i];

    // 2nd layer output + shift
    for (int i = 0; i < L2COUNT; i++)
        o2[i] = SIGMA(s2[i], l);
    o2[L2COUNT] = 1.0;

    // Sum for 3rd layer
    double s3 = 0;
    for (int i = 0; i < L2COUNT + 1; i++)
        s3 += o2[i] * w23[i];

    // Final output
    *oa = SIGMA(s3, l);
}

/// <summary>
/// Does 1 nn weights adjustment based on layer1, layer2, layer3 output
/// and expected NN output.
/// </summary>
/// <param name="oa"> Layer3 output. </param>
```

```
/// <param name="ot"> Expected NN output. </param>
/// <param name="w12"> layer1->layer2 weights as matrix. </param>
/// <param name="w23"> layer2->layer3 weights as vector. </param>
/// <param name="l"> NN lambda parameter. </param>
/// <param name="a"> NN alpha parameter. </param>
/// <param name="o1"> Layer1 output. </param>
/// <param name="o2"> Layer2 output. </param>
void nn_adjust(double oa, double ot,
               double w12[L1COUNT + 1][L2COUNT], double w23[L2COUNT + 1], double l, double a,
               double o1[L1COUNT + 1], double o2[L2COUNT + 1])
{
    // Common part
    double cdelta = l * oa * (1.0 - oa) * (oa - ot);

    // 2l -> 3l deltas
    double deltas23[L2COUNT + 1];
    for (int i = 0; i < L2COUNT + 1; i++)
        deltas23[i] = o2[i] * cdelta;

    // 1l -> 2l deltas
    double deltas12[L1COUNT + 1][L2COUNT];
    for (int i = 0; i < L1COUNT + 1; i++)
        for (int j = 0; j < L2COUNT; j++)
            deltas12[i][j] = o1[i] * l * o2[j] * (1 - o2[j]) * w23[j] * cdelta;

    // 2l -> 3l adjustment
    for (int i = 0; i < L2COUNT + 1; i++)
        w23[i] -= (a * deltas23[i]);

    // 1l -> 2l adjustment
    for (int i = 0; i < L1COUNT + 1; i++)
        for (int j = 0; j < L2COUNT; j++)
            w12[i][j] -= (a * deltas12[i][j]);
}

```

NN231\nn.h

```
#pragma once

#include <stdio.h>

#define L1COUNT 2
#define L2COUNT 3

/// <summary>
/// NN activation implementation.
/// </summary>
/// <param name="x"> Input signal vector. </param>
/// <param name="w12"> layer1->layer2 weights as matrix. </param>
/// <param name="w23"> layer2->layer3 weights as vector. </param>
/// <param name="l"> NN lambda parameter. </param>
/// <param name="o1"> Vector to write layer1 output. </param>
/// <param name="o2"> Vector to write layer2 output. </param>
/// <param name="oa"> Pointer to write layer3 output. </param>
```

```
18 void nn_activate(double x[L1COUNT],
19                  double w12[L1COUNT + 1][L2COUNT], double w23[L2COUNT + 1], double l,
20                  double o1[L1COUNT + 1], double o2[L2COUNT + 1], double * oa);
21
22 /// <summary>
23 /// Does 1 nn weights adjustment based on layer1, layer2, layer3 output
24 /// and expected NN output.
25 /// </summary>
26 /// <param name="oa"> Layer3 output. </param>
27 /// <param name="ot"> Expected NN output. </param>
28 /// <param name="w12"> layer1->layer2 weights as matrix. </param>
29 /// <param name="w23"> layer2->layer3 weights as vector. </param>
30 /// <param name="l"> NN lambda parameter. </param>
31 /// <param name="a"> NN alpha parameter. </param>
32 /// <param name="o1"> Layer1 output. </param>
33 /// <param name="o2"> Layer2 output. </param>
34 void nn_adjust(double oa, double ot,
35                double w12[L1COUNT + 1][L2COUNT], double w23[L2COUNT + 1], double l, double a,
36                double o1[L1COUNT + 1], double o2[L2COUNT + 1]);
37
38 /// <summary>
39 /// Writes neural network to stream in binary format.
40 /// </summary>
41 /// <param name="l"> NN lambda parameter. </param>
42 /// <param name="a"> NN alpha parameter. </param>
43 /// <param name="w12"> Weights of layer1 -> layer2 as matrix. </param>
44 /// <param name="w23"> Weights of layer2 -> layer3 as vector. </param>
45 /// <param name="f"> Output stream. </param>
46 void nn_fwrite(double l, double a,
47                double w12[L1COUNT + 1][L2COUNT], double w23[L2COUNT + 1],
48                FILE * f);
49
50 /// <summary>
51 /// Reads neural network from stream.
52 /// </summary>
53 /// <param name="l"> Pointer to read NN lambda parameter in. </param>
54 /// <param name="a"> Pointer to read NN alpha parameter in. </param>
55 /// <param name="w12"> Matrix to read layer1->layer2 weights in. </param>
56 /// <param name="w23"> Vector to read layer2->layer3 weights in. </param>
57 /// <param name="f"> Output stream. </param>
58 void nn_fread(double * l, double * a,
59               double w12[L1COUNT + 1][L2COUNT], double w23[L2COUNT + 1],
60               FILE * f);
61
62 /// <summary>
63 /// Writes neural network to stream in text format.
64 /// </summary>
65 /// <param name="l"> NN lambda parameter. </param>
66 /// <param name="a"> NN alpha parameter. </param>
67 /// <param name="w12"> Weights of layer1 -> layer2 as matrix. </param>
68 /// <param name="w23"> Weights of layer2 -> layer3 as vector. </param>
69 /// <param name="f"> Output stream. </param>
70 void nn_fprint(double l, double a,
71                double w12[L1COUNT + 1][L2COUNT], double w23[L2COUNT + 1],
72                FILE * stream);
73
```

NN231\examples.py

```python
import random

OUTPUT = "examples.txt"
EXAMPLE_COUNT = 30

examples = []
while (len(examples) != EXAMPLE_COUNT):
    a = round(random.random(), 4)
    b = round(random.random(), 4)
    if a + b <= 1.0:
        examples.append((a, b, round(a + b, 6)))

with open(OUTPUT, 'w') as file:
    file.write(f"{EXAMPLE_COUNT}\n")
    for item in examples:
        file.write(f"{item[0]} {item[1]} {item[2]}\n")

```