Selected files

7 printable files

pnnlib\pch.c
pnnlib\pch.h
pnnlib\pnn_alloc_check.h
pnnlib\pnn_core.c
pnnlib\pnn_io.c
pnnlib\pnn_memory.c
pnnlib\pnn.h

pnnlib\pch.c

```c
// pch.c: source file corresponding to the pre-compiled header

#include "pch.h"

// When you are using pre-compiled headers, this source file is necessary for compilation to succeed.

```

pnnlib\pch.h

```c
// pch.h: This is a precompiled header file.
// Files listed below are compiled only once, improving build performance for future builds.
// This also affects IntelliSense performance, including code completion and many code browsing features.
// However, files listed here are ALL re-compiled if any one of them is updated between builds.
// Do not add files here that you will be updating frequently as this negates the performance advantage.

#ifndef PCH_H
#define PCH_H

#include "pnn.h"
#include "pnn_alloc_check.h"

#endif //PCH_H

```

pnnlib\pnn_alloc_check.h

```c
#pragma once

#include <stdio.h>
#include <stdlib.h>

/// <summary>
/// Macros to check for memory allocation fail.
/// </summary>
#define ALLOC_ERR_MSG "Error: memory allocation failed.\n"
```

```
#define pnn_fail_alloc_check(p)\
{\
    if (p == NULL)\
    {\
        fprintf(stderr, ALLOC_ERR_MSG);\
        abort();\
    }\
}


```

pnnlib\pnn_core.c

```
#include "pch.h"

#include "pnn.h"

#include "pnn_alloc_check.h"

#include <stdlib.h>
#include <math.h>
#include <omp.h>
#include <string.h>

#define MAX_SINGLE_THREAD 250

static double act(int property_count, pnn_reference * reference, double * input, double sigma);
static int index_of_largest_prediction(int n, pnn_prediction * prediction_arr);
static int are_class_names_equal(char * class_name1, char * class_name2);

pnn_prediction * pnn_predict(pnn_data * net, double * input)
{
    pnn_prediction * prediction_arr = (pnn_prediction *)malloc(net->class_count * sizeof(pnn_prediction));
    pnn_fail_alloc_check(prediction_arr);

    for (int i = 0; i < net->class_count; i++)
    {
        pnn_class * current_class = net->pnn_class_arr[i];

        prediction_arr[i].class_name = current_class->class_name;
        prediction_arr[i].prediction = 0;

        for (int j = 0; j < current_class->reference_count; j++)
        {
            prediction_arr[i].prediction += act(net->property_count,
                                                current_class->reference_arr[j], input, net->sigma);
        }
    }

    return prediction_arr;
}

```

```
40  pnn_evaluation * pnn_evaluate(pnn_data * net, pnn_data * data)
41  {
42      pnn_evaluation * evaluation_arr = (pnn_evaluation *)malloc(data->class_count *
    sizeof(pnn_evaluation));
43      pnn_fail_alloc_check(evaluation_arr);
44
45      for (int i = 0; i < data->class_count; i++)
46      {
47          pnn_class * current_class = data->pnn_class_arr[i];
48          pnn_evaluation * current_evaluation = &(evaluation_arr[i]);
49
50          current_evaluation->class_name = current_class->class_name;
51          current_evaluation->accuracy = 0;
52
53          if (current_class->reference_count > MAX_SINGLE_THREAD)
54          {
55              double acc = 0;
56              int j;
57              #pragma omp parallel for reduction(+:acc)
58              for (j = 0; j < current_class->reference_count; j++)
59              {
60                  pnn_reference * current_reference = current_class->reference_arr[j];
61
62                  pnn_prediction * prediction_arr = pnn_predict(net, current_reference->
    reference);
63                  int k = index_of_largest_prediction(net->class_count, prediction_arr);
64
65                  if (are_class_names_equal(current_class->class_name, prediction_arr[k]
    .class_name) == 0)
66                      acc += 1.0;
67
68                  free(prediction_arr);
69              }
70
71              current_evaluation->accuracy = acc / (double)(current_class->reference_count);
72          }
73          else
74          {
75              for (int j = 0; j < current_class->reference_count; j++)
76              {
77                  pnn_reference * current_reference = current_class->reference_arr[j];
78
79                  pnn_prediction * prediction_arr = pnn_predict(net, current_reference->
    reference);
80                  int k = index_of_largest_prediction(net->class_count, prediction_arr);
81
82                  if (are_class_names_equal(current_class->class_name, prediction_arr[k]
    .class_name) == 0)
83                      current_evaluation->accuracy += 1.0;
84
85                  free(prediction_arr);
86              }
87
88              current_evaluation->accuracy /= (double)(current_class->reference_count);
89          }
90      }
91
```

```
92      return evaluation_arr;
93  }
94
95  static double act(int property_count, pnn_reference * reference, double * input, double
    sigma)
96  {
97      double acc = 0, d;
98      for (int i = 0; i < property_count; i++)
99      {
100         d = reference->reference[i] - input[i];
101         acc += d * d;
102     }
103     acc /= -(sigma * sigma);
104
105     return exp(acc);
106 }
107
108 static int index_of_largest_prediction(int n, pnn_prediction * prediction_arr)
109 {
110     int i = 0;
111     for (int j = 0; j < n; j++)
112     {
113         if (prediction_arr[i].prediction < prediction_arr[j].prediction)
114             i = j;
115     }
116     return i;
117 }
118
119 static int are_class_names_equal(char * class_name1, char * class_name2)
120 {
121     char * dash1 = strchr(class_name1, '-');
122     char * dash2 = strchr(class_name2, '-');
123
124     size_t n = dash1 - class_name1;
125     if (n != dash2 - class_name2)
126         return 1;
127
128     return strncmp(class_name1, class_name2, n);
129 }
130
```

pnnlib\pnn_io.c

```
1   #include "pch.h"
2
3   #include "pnn.h"
4
5   #include "pnn_alloc_check.h"
6
7   #include <stdio.h>
8   #include <stdlib.h>
9
10  pnn_data * pnn_data_load(FILE * f)
11  {
12      int property_count;
```

```
    int total_reference_count;
    int class_count;

    fread(&property_count, sizeof(int), 1, f);
    fread(&total_reference_count, sizeof(int), 1, f);
    fread(&class_count, sizeof(int), 1, f);

    pnn_class ** pnn_class_arr = (pnn_class **)malloc(class_count * sizeof(pnn_class *));
    pnn_fail_alloc_check(pnn_class_arr);
    for (struct
         {
             int i;
             size_t class_name_len; char * class_name;
             int reference_count;
         }
         state_class = { .i = 0, .class_name_len = 0, .class_name = NULL, .reference_count = 0 };
         state_class.i < class_count;
         state_class.i++)
    {
        fread(&(state_class.class_name_len), sizeof(size_t), 1, f);

        state_class.class_name = (char *)malloc((state_class.class_name_len + 1) * sizeof(char));
        pnn_fail_alloc_check(state_class.class_name);
        fread(state_class.class_name, sizeof(char), state_class.class_name_len, f);
        state_class.class_name[state_class.class_name_len] = '\0';

        fread(&(state_class.reference_count), sizeof(int), 1, f);

        pnn_reference ** pnn_reference_arr =
            (pnn_reference **)malloc(state_class.reference_count * sizeof(pnn_reference *));
        pnn_fail_alloc_check(pnn_reference_arr);
        for (struct
             {
                 int j;
                 int reference_id; int property_count;
                 double * reference;
             }
             state_reference = { .j = 0, .reference_id = 0, .property_count = 0, .reference = NULL};
             state_reference.j < state_class.reference_count;
             state_reference.j++)
        {
            fread(&(state_reference.reference_id), sizeof(int), 1, f);
            fread(&(state_reference.property_count), sizeof(int), 1, f);

            state_reference.reference = (double *)malloc(state_reference.property_count * sizeof(double));
            pnn_fail_alloc_check(state_reference.reference);
            fread(state_reference.reference, sizeof(double), state_reference.property_count, f);

            pnn_reference_arr[state_reference.j] = pnn_reference_create(state_reference.reference_id, state_reference.property_count,
state_reference.reference);
        }
```

```
64
65          pnn_class_arr[state_class.i] = pnn_class_create(state_class.class_name,
    state_class.reference_count, pnn_reference_arr);
66      }
67
68      pnn_data * data = pnn_data_create(DEFAULT_SIGMA, property_count, total_reference_count,
    class_count,
69                                        pnn_class_arr);
70
71      return data;
72  }
73
74  void pnn_data_fprint(pnn_data * data, FILE * f)
75  {
76      fprintf(f, "sigma = %lf; property_count = %d; total_reference_count = %d; class_count =
    %d\n",
77              data->sigma, data->property_count, data->total_reference_count, data->
    class_count);
78
79      for (struct
80           {
81               int i;
82               pnn_class * current_class;
83           }
84           state_class = { .i = 0, .current_class = NULL };
85           state_class.i < data->class_count; state_class.i++)
86      {
87          state_class.current_class = data->pnn_class_arr[state_class.i];
88
89          fprintf(f, "    class_name = %s; reference_count = %d\n",
90                  state_class.current_class->class_name, state_class.current_class->
    reference_count);
91
92          for (struct
93               {
94                  int j;
95                  pnn_reference * current_reference;
96               }
97               state_reference = { .j = 0, .current_reference = NULL };
98               state_reference.j < state_class.current_class->reference_count;
    state_reference.j++)
99          {
100             state_reference.current_reference = state_class.current_class->
    reference_arr[state_reference.j];
101
102             fprintf(f, "        id = %d; property_count = %d; ",
103                     state_reference.current_reference->id, state_reference.current_reference-
    >property_count);
104
105             for (int k = 0; k < state_reference.current_reference->property_count - 1; k++)
106             {
107                 fprintf(f, "%.2lf,", state_reference.current_reference->reference[k]);
108             }
109             fprintf(f, "%.2lf\n", state_reference.current_reference->
    reference[state_reference.current_reference->property_count - 1]);
110         }
111     }
112 }
```

113

pnnlib\pnn_memory.c

```c
#include "pch.h"

#include "pnn.h"

#include "pnn_alloc_check.h"

#include <stdlib.h>

pnn_reference * pnn_reference_create(int id, int property_count, double * reference)
{
    pnn_reference * obj = (pnn_reference *)malloc(sizeof(pnn_reference));
    pnn_fail_alloc_check(obj);

    obj->id = id;
    obj->property_count = property_count;
    obj->reference = reference;

    return obj;
}

pnn_reference * pnn_reference_free(pnn_reference * obj)
{
    free(obj->reference);
    free(obj);

    return NULL;
}

pnn_class * pnn_class_create(char * class_name, int reference_count, pnn_reference **
reference_arr)
{
    pnn_class * obj = (pnn_class *)malloc(sizeof(pnn_class));
    pnn_fail_alloc_check(obj);

    obj->class_name = class_name;
    obj->reference_count = reference_count;
    obj->reference_arr = reference_arr;

    return obj;
}

pnn_class * pnn_class_free(pnn_class * obj)
{
    free(obj->class_name);
    for (int i = 0; i < obj->reference_count; i++)
        pnn_reference_free(obj->reference_arr[i]);
    free(obj);

    return NULL;
}

```

```
pnn_data * pnn_data_create(double sigma, int property_count, int total_reference_count, int class_count,
                           pnn_class ** pnn_class_arr)
{
    pnn_data * obj = (pnn_data *)malloc(sizeof(pnn_data));
    pnn_fail_alloc_check(obj);

    obj->sigma = sigma;
    obj->property_count = property_count;
    obj->total_reference_count = total_reference_count;
    obj->class_count = class_count;
    obj->pnn_class_arr = pnn_class_arr;

    return obj;
}

pnn_data * pnn_data_free(pnn_data * obj)
{
    for (int i = 0; i < obj->class_count; i++)
        pnn_class_free(obj->pnn_class_arr[i]);
    free(obj);

    return NULL;
}

```

pnnlib\pnn.h

```
#pragma once

#include <stdio.h>

#define DEFAULT_SIGMA 0.01
#define EPSILON 0.001

typedef struct pnn_reference
{
    int id;
    int property_count;
    double * reference;
} pnn_reference;

typedef struct pnn_class
{
    char * class_name;
    int reference_count;
    struct pnn_reference ** reference_arr;
} pnn_class;

typedef struct pnn_data
{
    double sigma;
    int property_count;
    int total_reference_count;
    int class_count;
```

```
28      struct pnn_class ** pnn_class_arr;
29  }pnn_data;
30
31  pnn_reference * pnn_reference_create(int id, int property_count, double * reference);
32  pnn_reference * pnn_reference_free(pnn_reference * obj);
33
34  pnn_class * pnn_class_create(char * class_name, int reference_count, pnn_reference **
    reference_arr);
35  pnn_class * pnn_class_free(pnn_class * obj);
36
37  pnn_data * pnn_data_create(double sigma, int property_count, int total_reference_count, int
    class_count,
38                             pnn_class ** pnn_class_arr);
39  pnn_data * pnn_data_free(pnn_data * obj);
40
41  pnn_data * pnn_data_load(FILE * f);
42  void pnn_data_fprint(pnn_data * data, FILE * f);
43
44  typedef struct
45  {
46      char * class_name;
47      double prediction;
48  }
49  pnn_prediction;
50
51  typedef struct
52  {
53      char * class_name;
54      double accuracy;
55  }
56  pnn_evaluation;
57
58  pnn_prediction * pnn_predict(pnn_data * data, double * input);
59  pnn_evaluation * pnn_evaluate(pnn_data * net, pnn_data * data);
60
```