Annexe TIPE

Malaury DUTOUR - candidat 22977

2022-2023

Table des matières

1 Extraits du code embarqué

1.1 Algorithme de descente de gradient

```
#ifndef MATH_FITTING_GRADIENT_DESC_HPP
#define MATH_FITTING_GRADIENT_DESC_HPP

#include "../data_set.hpp"
#include "fitting.hpp"
#include <functional>

template <typename T, std::size_t params_count>
using GradientDescFunc = std::function<T(T, std::array<T, params_count>)>;

template <typename T, std::size_t params_count>
class GradientDescResult : public FittingResult<T> {
    std::array<T, params_count> params;
    GradientDescFunc<T, params_count> func;
    T dataset_mean;
    T dataset_std;

  public:
    GradientDescResult(GradientDescFunc<T, params_count> _func,
                       std::array<T, params_count> _params, T _dataset_mean,
                       T _dataset_std)
        : params(_params), func(_func), dataset_mean(_dataset_mean),
          dataset_std(_dataset_std) {}
    void print();
    T calculateOutput(T input);
    void serialize(std::ostream &os);
};

enum GradientDescCompletion {
    GRADIENT_DESC_CONVERGED,
    GRADIENT_DESC_MAX_ITERATIONS,
};
```

```cpp
std::string gradientDescCompletionToString(GradientDescCompletion completion);

template <typename T>
class GradientDescStats : public FittingResultStats<T> {
    GradientDescCompletion completion;
    T sse;
    T r2;
    T rmse;
    long iterations;

  public:
    GradientDescStats()
        : completion(GRADIENT_DESC_MAX_ITERATIONS), sse(T()), r2(T()),
          rmse(T()), iterations(T()) {}
    GradientDescStats(GradientDescCompletion _completion, T _sse, T _r2,
                      T _rmse, long _iterations)
        : completion(_completion), sse(_sse), r2(_r2), rmse(_rmse),
          iterations(_iterations) {}
    void print();
};

template <typename T, std::size_t params_count>
class GradientDescSettings {
    T learning_rate;
    T tolerance;
    long max_iterations;
    std::array<T, params_count> initial_params;

  public:
    GradientDescSettings(T _learning_rate, T _tolerance, long _max_iterations,
                         std::array<T, params_count> _initial_params)
        : learning_rate(_learning_rate), tolerance(_tolerance),
          max_iterations(_max_iterations), initial_params(_initial_params) {}
    T getLearningRate() { return learning_rate; }
    T getTolerance() { return tolerance; }
    long getMaxIterations() { return max_iterations; }
    std::array<T, params_count> getInitialParams() { return initial_params; }
};

template <typename T, std::size_t params_count>
class GradientDescFactory : public FittingResultFactory<T> {
    GradientDescFunc<T, params_count> func;
    GradientDescSettings<T, params_count> settings;
    GradientDescStats<T> last_stats;

    T cost_func(DataSet<T> &data, std::array<T, params_count> params);
    std::array<T, params_count>
    grad_cost_func(DataSet<T> &data, std::array<T, params_count> params);

  public:
    GradientDescFactory() = default;
    GradientDescFactory(GradientDescFunc<T, params_count> _func,
                        GradientDescSettings<T, params_count> _settings)
        : func(_func), settings(_settings){};
    std::unique_ptr<FittingResult<T>> deserialize(std::string str);
    std::unique_ptr<FittingResult<T>> getDefault();
    std::unique_ptr<FittingResult<T>> calculateFitting(DataSet<T> &data);
    std::unique_ptr<FittingResultStats<T>> getLastCalculationStats();
};

std::string gradientDescCompletionToString(GradientDescCompletion completion) {
```

```
    switch (completion) {
    case GRADIENT_DESC_CONVERGED:
        return "Converged";
    case GRADIENT_DESC_MAX_ITERATIONS:
        return "Max Iterations";
    default:
        return "Unknown";
    }
}

template <typename T>
void GradientDescStats<T>::print() {
    Serial.println("GradientDescStats: ");
    Serial.println("Completion: " +
                   String(gradientDescCompletionToString(completion).c_str()));
    Serial.println("SSE: " + String(sse, 8));
    Serial.println("R^2: " + String(r2, 8));
    Serial.println("RMSE: " + String(rmse, 8));
    Serial.println("Iterations: " + String(iterations));
}

template <typename T, std::size_t params_count>
void GradientDescResult<T, params_count>::print() {
    Serial.println("GradientDesc: ");
    for (auto param : params) {
        Serial.println(String(param, 8));
    }
    Serial.println("Dataset Mean: " + String(dataset_mean, 8));
    Serial.println("Dataset Std: " + String(dataset_std, 8));
    Serial.println();
}

template <typename T, std::size_t params_count>
T GradientDescResult<T, params_count>::calculateOutput(T input) {
    return func(input, params) * dataset_std + dataset_mean;
}

template <typename T, std::size_t params_count>
void GradientDescResult<T, params_count>::serialize(std::ostream &os) {
    for (std::size_t i = 0; i < params_count; i++) {
        os << params[i] << ",";
    }
    os << dataset_mean << "," << dataset_std;
}

template <typename T, std::size_t params_count>
T GradientDescFactory<T, params_count>::cost_func(
    DataSet<T> &data, std::array<T, params_count> params) {
    T cost = data.accumulate([&](DataPoint<T> dp) {
        return std::pow(func(dp.x, params) - dp.y, 2);
    });
    return cost / (2 * data.size());
}

template <typename T, std::size_t params_count>
std::array<T, params_count> add_arr(std::array<T, params_count> a,
                                    std::array<T, params_count> b,
                                    T scale = T()) {
    std::array<T, params_count> result;
    for (std::size_t i = 0; i < params_count; i++) {
        result[i] = a[i] + b[i] * scale;
```

```
    }
    return result;
}

template <typename T, std::size_t params_count>
std::array<T, params_count>
GradientDescFactory<T, params_count>::grad_cost_func(
    DataSet<T> &data, std::array<T, params_count> params) {
    std::array<T, params_count> grad;
    std::array<T, params_count> deriv_eps;
    std::fill(deriv_eps.begin(), deriv_eps.end(), T());
    T h = std::pow(std::numeric_limits<T>::epsilon(), T(1.0 / 3.0));
    for (std::size_t i = 0; i < params_count; i++) {
        deriv_eps[i] = h;
        grad[i] = (cost_func(data, add_arr(params, deriv_eps)) -
                   cost_func(data, add_arr(params, deriv_eps, T(-1.0)))) /
                  (2 * h);

        deriv_eps[i] = T();
    }
    return grad;
}

template <typename T, std::size_t params_count>
std::unique_ptr<FittingResult<T>>
GradientDescFactory<T, params_count>::calculateFitting(
    DataSet<T> &data_nonnorm) {
    std::array<T, params_count> params;
    auto data = data_nonnorm.normalize();
    auto initial_params = settings.getInitialParams();
    std::copy(initial_params.begin(), initial_params.end(), params.begin());

    T learning_rate = settings.getLearningRate();
    long iterations = 0;
    GradientDescCompletion completion = GRADIENT_DESC_CONVERGED;
    while (true) {
        auto grad = grad_cost_func(data, params);
        for (std::size_t i = 0; i < params_count; i++) {
            params[i] -= learning_rate * grad[i];
        }

        T norm_grad = 0;
        for (std::size_t i = 0; i < params_count; i++) {
            norm_grad += std::pow(grad[i], 2);
        }

        if (norm_grad < settings.getTolerance()) {
            break;
        }
        if (iterations > settings.getMaxIterations()) {
            completion = GRADIENT_DESC_MAX_ITERATIONS;
            break;
        }
        if (iterations % 100 == 0) {
            ESP.wdtFeed();
            Serial.println("Iteration: " + String(iterations) +
                           " Norm: " + String(norm_grad, 8));
        }
        iterations++;
    }
    T cost = cost_func(data, params);
```

```
    T rmse = std::sqrt(cost / data.size());
    T sse = cost;
    T r2 = 1 - (sse / (data.size() * std::pow(data.std(), 2)));

    last_stats = GradientDescStats<T>(completion, sse, r2, rmse, iterations);

    return std::make_unique<GradientDescResult<T, params_count>>(
        func, params, data_nonnorm.mean(), data_nonnorm.std());
};

template <typename T, std::size_t params_count>
std::unique_ptr<FittingResult<T>>
GradientDescFactory<T, params_count>::deserialize(std::string str) {
    auto values = split(str, ',');
    std::array<T, params_count> params;
    for (std::size_t i = 0; i < params_count; i++) {
        params[i] = std::stod(values[i]);
    }
    T mean = std::stod(values[params_count]);
    T std = std::stod(values[params_count + 1]);
    return std::make_unique<GradientDescResult<T, params_count>>(func, params,
                                                                 mean, std);
};

template <typename T, std::size_t params_count>
std::unique_ptr<FittingResult<T>>
GradientDescFactory<T, params_count>::getDefault() {
    std::array<T, params_count> params;
    std::fill(params.begin(), params.end(), T());
    return std::make_unique<GradientDescResult<T, params_count>>(func, params,
                                                                 0, 1);
}

template <typename T, std::size_t params_count>
inline std::unique_ptr<FittingResultStats<T>>
GradientDescFactory<T, params_count>::getLastCalculationStats() {
    return std::make_unique<GradientDescStats<T>>(last_stats);
}

#endif
```

1.2 Manipulation des données

```
#ifndef MATH_DATA_SET_HPP
#define MATH_DATA_SET_HPP

#include <vector>

template <typename T>
class DataPoint {
  public:
    DataPoint(T x, T y) : x(x), y(y) {}
    T x;
    T y;
};

template <typename T>
class DataSet {
    std::vector<DataPoint<T>> data;
```

```
  public:
    DataSet() = default;
    std::size_t size();
    void appendDataPoint(DataPoint<T> dataPoint);
    void extend(DataSet<T> dataSet);
    void clear();
    DataPoint<T> at(std::size_t index);
    T accumulate(std::function<T(DataPoint<T> &)> func);
    DataSet<T> map(std::function<DataPoint<T>(DataPoint<T> &)> func);
    DataSet<T> normalize();
    T std();
    T mean();
    void print();
};

template <typename T>
std::size_t DataSet<T>::size() {
    return data.size();
}

template <typename T>
T DataSet<T>::accumulate(std::function<T(DataPoint<T> &)> func) {
    T result = 0;
    for (auto dataPoint : data) {
        result += func(dataPoint);
    }
    return result;
}

template <typename T>
DataSet<T> DataSet<T>::map(std::function<DataPoint<T>(DataPoint<T> &)> func) {
    DataSet<T> result;
    for (auto dataPoint : data) {
        result.appendDataPoint(func(dataPoint));
    }
    return result;
}

template <typename T>
DataSet<T> DataSet<T>::normalize() {
    DataSet<T> result;
    T mean = this->mean();
    T std = this->std();
    for (auto dataPoint : data) {
        result.appendDataPoint(
            DataPoint<T>(dataPoint.x, (dataPoint.y - mean) / std));
    }
    return result;
}

template <typename T>
T DataSet<T>::std() {
    T mean = this->mean();
    T sum = accumulate([&mean](DataPoint<T> dataPoint) {
        return std::pow(dataPoint.y - mean, 2);
    });
    return std::sqrt(sum / size());
}

template <typename T>
T DataSet<T>::mean() {
```

```
    return accumulate([](DataPoint<T> dataPoint) { return dataPoint.y; }) /
            size();
}

template <typename T>
void DataSet<T>::print() {
    for (auto dataPoint : data) {
        Serial.print(dataPoint.x);
        Serial.print(" ");
        Serial.println(dataPoint.y);
    }
}

template <typename T>
void DataSet<T>::appendDataPoint(DataPoint<T> dataPoint) {
    data.push_back(dataPoint);
}

template <typename T>
void DataSet<T>::extend(DataSet<T> dataSet) {
    data.insert(data.end(), dataSet.data.begin(), dataSet.data.end());
}

template <typename T>
void DataSet<T>::clear() {
    data.clear();
}

template <typename T>
DataPoint<T> DataSet<T>::at(std::size_t index) {
    return data[index];
}

#endif
```

1.3 Sauvergarde etréstitution des résultats de l'ajustemnt de courbe

```
#ifndef CONFIG_CONFIG_HPP
#define CONFIG_CONFIG_HPP

#include "../io/adc_mux.hpp"
#include "../math/fitting/fitting.hpp"
#include "LittleFS.h"
#include <array>
#include <iomanip>
#include <ostream>
#include <sstream>
#include <string>

template <typename T, std::size_t size>
class Config {
    std::array<std::unique_ptr<FittingResult<T>>, size> fittingResult;
    std::array<DataSet<T>, size> dataSet;
    std::unique_ptr<FittingResultFactory<T>> fittingResultFactory;

  public:
    Config() = delete;
    Config(std::unique_ptr<FittingResultFactory<T>> fittingResultFactory);
    void serialize(std::ostream &os);
    void deserialize(std::istream &is);
```

```
    void print();
    void setToDefault();

    std::unique_ptr<FittingResult<T>> getFittingResultAt(std::size_t index);
    void calculateFittingResultAt(std::size_t index);
    std::unique_ptr<FittingResultStats<T>>
    getFittingResultStatsAt(std::size_t index);
    void printFittingResultAt(std::size_t index);
    void extendDatasetAt(DataSet<T> dataset, std::size_t index);
    void setDatasetAt(DataSet<T> dataset, std::size_t index);
    DataSet<T> getDatasetAt(std::size_t index);
    void convertToWeight(std::array<T, size> readings,
                         std::array<T, size> &weight);
};

template <typename T, std::size_t size>
Config<T, size>::Config(
    std::unique_ptr<FittingResultFactory<T>> fittingResultFactory)
    : fittingResultFactory(std::move(fittingResultFactory)) {
    setToDefault();
}

template <typename T, std::size_t size>
void Config<T, size>::serialize(std::ostream &os) {
    os << std::setprecision(std::numeric_limits<T>::digits10 / 2);
    os << std::to_string(size) << '\n';
    for (size_t i = 0; i < size; i++) {
        fittingResult[i]->serialize(os);
        os << '\n';
    }
    for (size_t i = 0; i < size; i++) {
        os << std::to_string(dataSet[i].size()) << '\n';
        for (size_t j = 0; j < dataSet[i].size(); j++) {
            auto point = dataSet[i].at(j);
            os << std::to_string(point.x) << ',' << std::to_string(point.y)
               << '\n';
        }
    }
}

template <typename T, std::size_t size>
std::unique_ptr<FittingResult<T>>
Config<T, size>::getFittingResultAt(size_t index) {
    return this->fittingResult[index];
}

std::vector<std::string> split(std::string str, char delimiter) {
    std::vector<std::string> result;
    std::string token;
    std::stringstream ss(str);
    // NOLINTNEXTLINE(bugprone-infinite-loop)
    while (std::getline(ss, token, delimiter)) {
        result.push_back(token);
    }
    return result;
}

template <typename T, std::size_t size>
void Config<T, size>::deserialize(std::istream &is) {
    std::string line;
    std::getline(is, line);
```

```
    for (size_t i = 0; i < size; i++) {
        std::getline(is, line);
        fittingResult[i].reset(
            fittingResultFactory->deserialize(line).release());
    }
    for (size_t i = 0; i < size; i++) {
        std::getline(is, line);
        auto size_d = std::stoi(line);
        for (size_t j = 0; j < size_d; j++) {
            std::getline(is, line);
            auto values = split(line, ',');
            auto point =
                DataPoint<T>{std::stof(values[0]), std::stof(values[1])};
            dataSet[i].appendDataPoint(point);
        }
    }
}

template <typename T, std::size_t size>
void Config<T, size>::print() {
    std::stringstream ss;
    serialize(ss);
    Serial.println(ss.str().c_str());
}

template <typename T, std::size_t size>
void Config<T, size>::setToDefault() {
    for (size_t i = 0; i < size; i++) {
        fittingResult[i].reset(fittingResultFactory->getDefault().release());
        dataSet[i].clear();
    }
}

template <typename T, std::size_t size>
void Config<T, size>::calculateFittingResultAt(std::size_t index) {
    fittingResult[index].reset(
        fittingResultFactory->calculateFitting(dataSet[index]).release());
}

template <typename T, std::size_t size>
std::unique_ptr<FittingResultStats<T>>
Config<T, size>::getFittingResultStatsAt(std::size_t index) {
    return this->fittingResultFactory->getLastCalculationStats();
}

template <typename T, std::size_t size>
void Config<T, size>::printFittingResultAt(std::size_t index) {
    this->fittingResult[index]->print();
}

template <typename T, std::size_t size>
void Config<T, size>::setDatasetAt(DataSet<T> dataset, std::size_t index) {
    this->dataSet[index] = dataset;
}

template <typename T, std::size_t size>
DataSet<T> Config<T, size>::getDatasetAt(std::size_t index) {
    return this->dataSet[index];
}

template <typename T, std::size_t size>
```

```
void Config<T, size>::extendDatasetAt(DataSet<T> dataset, std::size_t index) {
    this->dataSet[index].extend(dataset);
}

template <typename T, std::size_t size>
void Config<T, size>::convertToWeight(std::array<T, size> readings,
                                       std::array<T, size> &weight) {
    for (std::size_t i = 0; i < size; i++) {
        weight[i] = fittingResult[i]->calculateOutput(readings[i]);
    }
}

#endif
```

1.4 Exemple de configuration stockant les résultats de l'ajustement de courbe

```
4
0.7074954,1.11653,-1.744186,2500.091,1580.995
0.8625062,1.28479,-1.486115,2277.889,1600.382
0.3374475,2.082868,-1.383819,2500.143,1792.644
1.486843,1.095044,-1.955273,2350.1,1581.781
11
3.300750,1.000000
1.606625,500.000000
0.899375,1000.000000
0.627625,1500.000000
0.524500,2000.000000
0.393625,2500.000000
0.319125,3000.000000
0.330000,3500.000000
0.315625,4000.000000
0.352000,4500.000000
0.249500,5000.000000
9
3.300750,1.000000
2.120500,500.000000
1.319875,1000.000000
0.957625,1500.000000
0.713000,2000.000000
0.532250,3000.000000
0.436375,3500.000000
0.424375,4000.000000
0.381250,5000.000000
7
3.306000,1.000000
1.265000,500.000000
0.656000,1500.000000
0.484250,2500.000000
0.440500,3500.000000
0.388000,4500.000000
0.360500,5000.000000
10
3.306875,1.000000
1.738625,500.000000
1.215000,1000.000000
0.993000,1500.000000
0.776750,2000.000000
0.737625,2500.000000
0.655250,3000.000000
0.699625,3500.000000
0.465250,4500.000000
```

```
0.445000,5000.000000
```

2 Extraction des données de la carte SD

2.1 Code réalisant l'extraction

```c
#include "stdio.h"
#include <stdlib.h>
#include "stdint.h"
#include "string.h"

int main(int argc, char *argv[])
{
    FILE *in, *out;
    uint16_t time;
    int16_t ch1, ch2, ch3, ch4;

    if (argc != 2 && argc != 3)
    {
        printf("Incorrect number of arguments, provided: %d, expected: 1 \n", argc - 1);
        return -1;
    }

    in = fopen(argv[1], "rb");

    char *outName = malloc(strlen(argv[1]) + 5);
    strcpy(outName, argv[1]);
    outName[strlen(argv[1]) - 4] = '\0';
    strcat(outName, ".csv");
    out = fopen(outName, "w");

    if (in == NULL)
    {
        printf("Could not open file %s \n", argv[1]);
        return -1;
    }

    int offset = 0;
    if (argc == 3)
    {
        offset = atoi(argv[2]);
    }
    else
    {
        printf("No offset provided, trying to find offset in file \n");
        char c;
        while (fread(&c, sizeof(char), 1, in) == 1)
        {
            if (c == '#')
            {
                offset = ftell(in);
                int i;
                for (i = 0; i < 9; i++)
                {
                    fread(&c, sizeof(char), 1, in);
                    if (c != '#')
                    {
                        break;
                    }
                }
                if (i == 9)
```

```
                {
                    offset += 9;
                    printf("Found offset at %d \n", offset);
                    break;
                }
            }
        }
    }

    printf("Converting %s to out.csv, offset %d \n", argv[1], offset);
    fprintf(out, "Time,Channel 1,Channel 2,Channel 3,Channel 4\n");
    fseek(in, offset, SEEK_SET);
    while (fread(&time, sizeof(uint16_t), 1, in) == 1)
    {
        fread(&ch1, sizeof(int16_t), 1, in);
        fread(&ch2, sizeof(int16_t), 1, in);
        fread(&ch3, sizeof(int16_t), 1, in);
        fread(&ch4, sizeof(int16_t), 1, in);
        fprintf(out, "%d,%d,%d,%d,%d\n", time, ch1, ch2, ch3, ch4);
    }
}
```

2.2 Extrait de la sortie du programme

```
Time,Channel 1,Channel 2,Channel 3,Channel 4
18690,-17663,19978,13829,10755
19970,-19711,17162,24581,21763
20482,-14335,17162,29701,19459
20226,-14847,18954,29189,23299
19714,-25599,17674,29445,31747
20738,-18431,16906,30725,-32253
19714,-20991,18698,-32251,-31997
19714,-26623,16650,-30971,-27645
21506,32001,17674,-27643,-22013
```