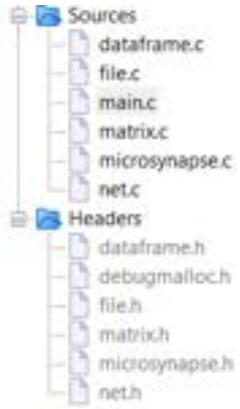
Microsynapse

felhasználói dokumentáció

Tartalom

A könyvtár használata

A microsynapse.h könyvtárat legegyszerűbben úgy használhatjuk fel saját programunkban, hogy a projektünk mappájába másoljuk a könyvtár fájljait, majd hozzáadjuk ezeket a projekthez. Ezután már csak az `#include "microsynapse.h"` sort kell beillesztenünk a programunk megfelelő részébe, és elérhetővé válnak a könyvtár függvényei. A projektnek a következő elemeket kell tartalmaznia:



Dataframe-ek, dataframe műveletek, adatbeolvasás

A könyvtár a *dataframe* származtatott típussal valósítja meg a feladatok nagy részét, ez a matrix 2-dimenziós, valós számokat tároló tömbből, és a *df_dim* dimenziót tároló struktúrából áll, melynek *rows* és *cols* paraméterei vannak, ezek adják meg a mátrix sorait és oszlopait.

Egy .csv (vesszővel elválasztott értékeket tároló) fájlból a read_csv() függvénnyel kaphatjuk meg az értékeket, melynek hívásakor a program beolvassa az adott fájl értékeit, és egy dataframe típusú változóban tárolja el azt. Az első paraméter az abszolút elérési útvonal, a második pedig az elválasztó karakter.

Egy fájl oszlopainak és sorainak meghatározásában segítségünkre lehet továbbá a get_dim() függvény is, amely a fentihez hasonló paramétereket kap meg, és dim formátumban adja vissza a fájl dimenzióit.

Mátrixműveletek

A könyvtárban találhatók olyan függvények, amik jelentősen megkönnyítik a mátrixokkal való munkát.

A *transpose()* függvény transzponálja a megadott 2D-s tömböt.

A *find_max()* függvény megtalálja egy 2D-s tömb abszolút legnagyobb értékét. Ha a tömbben negatív számok szerepelnek, akkor az *is_unsigned*-et 0-ra, egyébként 1-re kell állítanunk.

A *s_scale()* függvény egy adott 2D-s tömb elemeit(többnyire dataframe-et) 0 és 1 közötti értékekre méretezi (minden értéket eloszt a maximum értékkel).

Train-test split

Egy neurális háló életciklusának legmeghatározóbb része a tanulási-tesztelési fázis, tulajdonképpen ekkor dől el, hogy hasznos-e az, amit alkottunk. A beolvasott adathalmazból ki kell választanunk a megfelelő koordinátájú értékeket, és fel kell osztanunk azokat úgy, hogy tanulásra és tesztelésre is az általunk kívánt mennyiségű adat álljon rendelkezésünkre. Ezt a train_test_split() nevű függvénnyel tudjuk megvalósítani, amihez a split származtatott típus áll rendelkezésünkre. A függvény argumentumai a következők: egy dataframe, amiben a kiindulási adatok vannak, a paraméterek száma, amelyek alapján tanítani és tesztelni akarunk, a felosztás aránya, és az oszlopok koordinátái. Használata a következő:

```
int x_indexes[] = {0, 1};
int y_indexes[] = {2, 3};

split x_split;
split y_split;

double ratio = 0.7;
int train_no = (int) df.df_dim.rows * ratio;
int test_no = df.df_dim.rows - train_no;

x_split = train_test_split(df, 2, ratio, x_indexes);
y_split = train_test_split(df, 2, ratio, y_indexes);
```

Hálók létrehozása

A könyvtár használatával létrehozhatunk neurális modelleket, amelyeknek kezeléséhez további funkciókkal rendelkezünk. A háló három típusból épül fel: model, layer, és neuron.

Egy modellt a *create_model()* függvénnyel hozhatunk létre, mely egy model típusba menti a létrehozott hálót. Paraméterként az inputok, az outputok, a rejtett rétegek számát, valamint a kimenet aktivációs függvényét (SIGMOID, RELU, TANH).

A rejtett rétegeket az *add()* függvénnyel adhatjuk ténylegesen a modellhez, amelynek bemenetei maga a modell, a rétegben lévő neuronok száma, és az aktivációs függvény. A modell által igényelt memóriát és a súlyokat az *init()* kezeli annak inicializálásakor, amely csak a modellt kapja meg paraméterként.

Végső soron, a hálót tanítanunk és tesztelnünk is kell. Ehhez rendelkezésre áll a *fwd_pass()* függvény, amely a modell és az inputok megadása után végigfuttatja a hálón a betáplált értékeket, és megadja az outputok értékeit. Fontos: az betáplált paraméterek dimenziójának meg kell egyeznie az inputok számával.

Hálók tanítása, tesztelése

A könyvtár szerfelett egyszerűvé teszi a létrehozott hálók tanítását és tesztelését. Ennek módját az XOR feladat megtanításával szemléltetjük. Az XOR függvénynek két bemenete van, melyek 0, vagy 1 értéket vehetnek fel. Ha az egyik bemenet aktív, a másik nem, akkor a függvény 1 értéket vesz fel, ha a bemenetek megegyeznek, akkor pedig nullát. Első lépésként olvassuk be az adatokat az xor.csv fájlból, majd jelöljük ki a kimeneti és bemeneti (X, y) koordinátákat. Mivel kevés adatunk van, az adathalmaz egészét felhasználjuk a tanuláshoz, majd a teszteléshez is.

```
dataframe df = read_csv("C:\\infoc\\xor.csv", ",");
int x_coords[] = {0, 1};
int y_coords[] = {2};

split x_split = train_test_split(df, 2, 1.0, x_coords);
split y_split = train_test_split(df, 1, 1.0, y_coords);
```

Ezek után inicializáljuk a modellünket. A háló 2 bemenettel, egy kimenettel és egy rejtett réteggel rendelkezik. A rejtett rétegek paramétereit a háló létrehozása után az add() függvénnyel tudjuk meghatározni. A kimenet aktivációs függvénye legyen sigmoid.

```
//modell létrehozása és inicializálása
model* m;
m = create_model(2, 1, 1, SIGMOID);
add(m, 2, RELU);
init(m);
```

Végül inicializáljuk a hálót. Nem maradt más hátra, minthogy valóban megtanítsuk valamire a létrehozott modellt. Ezt a train() függvénnyel tehetjük meg, ennek visszatérési értéke az utolsó 100 iteráció alatt felgyülemlett hibák összege. Adjuk meg a modellt, az X és y dataframe-eket, a tanulási rátát (általában 0.15, vagy 0.2), és az iterációk számát. Mivel nem használunk komolyabb optimalizáló algoritmust, és a hibafüggvényünk is kezdetleges, fennáll a lehetősége annak, hogy a tanulás alatt a háló egy olyan lokális minimumot talál meg, ami számunkra nem megfelelő. Ennek kiküszöbölésére az egész folyamatot belefoglalhatjuk egy while-ciklusba, amely akkor ér véget, ha a globális hiba az általunk megadott tűréshatár alá esik. A cikluban mindig újrainicializáljuk a hálót, hisz különben a modell ugyanazt a lokális minimumot találná meg.

```
//háló tanítása
double GL_ERR = train(m, x_split.train, y_split.train, 0.2, 50000);
while(GL_ERR > 2) {
    fwrand(m);
    GL_ERR = train(m, x_split.train, y_split.train, 0.2, 50000);
    printf("[TRAINING] global error: %f\n", GL_ERR);
}
```

Ha továbbra is hitetlenkedünk, letesztelhetjük a modell teljesítményét.

```
test(m, x_split.train, y_split.train);
```

Végül az ízlésünk szerint tökéletesített modellt tetszőleges inputokkal is szólásra bírhatjuk, ekkor a megjósolt értékeket a pred() függvény az általunk kijelölt output tömbbe menti.

```
//háló kipróbálása tetszőleges inputtal
double proba[] = {1.0, 0.0};
double ertekek[1];
pred(m, proba, ertekek);
for(int i = 0; i < sizeof(ertekek)/sizeof(double); i++) {
    printf("predicted: %f\n", ertekek[i]);
}
```

Hálók mentése

Ha úgy érezzük, hogy a hálót eleget tanítottuk, elmenthetjük annak belső értékeit, majd más alkalommal be is tölthetjük őket. Ezekre a *save_net()* és a *load_net()* függvényeket használhatjuk. Mindkettőnek meg kell adnunk az elmenteni kívánt modellt, a súlyok és az eltolássúlyok tárolására szánt fájlok nevét. A save_net() létrehozza ezeket a fájlokat, és feltölti a megfelelő adatokkal, a load_net() viszont már meglévő fájlokkal dolgozik.

```
save_net(m, "weights_path.dat", "biases_path.dat");
load_net(m, "weights_path.dat", "biases_path.dat");
```

Útmutató a függvények és típusok helyes használatához

```
typedef struct dim {
    int rows;
    int cols;
} dim;
```

```
typedef struct dataframe{
    double** matrix; // c
    dim df_dim;
} dataframe;
```

```
typedef struct split {
    dataframe train;
    dataframe test;
} split;
```

```
typedef struct model {
    int no_inputs;
    int no_outputs;
    int no_hidden_layers;
    int added_layers;
    struct layer* layers;

    struct neuron* outputs;
} model;
```

```
typedef struct layer {
    int no_neurons;
    activation activation;

    struct neuron* neurons;
}layer;
```

```
typedef struct neuron {
    double output;
    double output_delta;
    double z;
    double z_delta;

    double bias;
    double bias_delta; //deri
    double* weights;
    double* weight_deltas; //
} neuron;
```

matrix.h
double find_max(double** matrix, dim d, int is_unsigned);
void s_scale(double** matrix, dim d, double max);
double dot(const double* a_vec, const double* b_vec, int height);
double** transpose(double** matrix, dim d);
void free_matrix(double** matrix, dim d);
void free_transpose(double** matrix, dim d);

dataframe.h
dim get_dim(FILE* file, const char* sep);
void read_err(FILE* fp, char* path);
void alloc_err(double** matrix);
dataframe read_csv(char* path, char *sep);
void free_df(dataframe df);
void head(dataframe d);
net.h
struct split train_test_split(dataframe d, int no_params, double ratio, int* coords);
double scaled_rand();
double sigmoid(double no);
double relu(double no);
double actv(bool is_prime, const int type, double no);
struct neuron* alloc_neurons(const int no_neurons, const int outputs);
struct model* create_model(const int nips, const int nops, const int no_hidden_layers, const int output_activation);
void init(model* m);
void add(model* model, const int no_neurons, char* activation);
void feed_input(model* m, const double* inputs);
void fwd_pass(model* m, double* inputs);
void out_prop(model* m, double* outs);
void bwd_pass(model* m, double* outs);
void update_weights(model* m, double alpha);
int shuffle(long instances);
double train(model* m, dataframe xt, dataframe y, double alpha, int epochs);
void test(model* m, dataframe xt, dataframe y);
void free_model(model* m);
file.h
void save_net(model* m, char* weights_file, char* biases_file);
void load_net(model* m, char* weights_file, char* biases_file);