

Assignment 3 Report

Tanat Tangun 630610737

October 2022

This report is about the result of my implementation of Genetic Algorithm (GA) for optimizing MLP on Rust language for 261456 - INTRO COMP INTEL FOR CPE class assignment. If you are interested to know how I implement GA and use it to optimize the MLP, you can see the source code on my Github repository or in this document appendix.

Problem

We want to train multilayer perceptron (MLP) for predicting breast cancer by using Genetic Algorithm (GA). The dataset we are using is Wisconsin Diagnostic Breast Cancer (WDBC) from UCI Machine learning Repository. This dataset has 30 features that we will use for training MLP to classify if the result is benign or malignant. The class distribution are 357 benign and 212 malignant which is unbalance.

We will use only 1 output node for all models because we are training a binary classification model so we can just map malignant (M) \rightarrow 1 and benign (B) \rightarrow 0. We then have a threshold at 0.5 if output node signal is more than 0.5 then the model predict malignant (positive) else it predict benign (negative). Accuracy is then calculated by using this equation $\frac{TP+TN}{TP+TN+FN+FP}$ where TP, TN, FN, FP come from confusion matrix. The experiment to see how effective GA is in training MLP will be demonstrated on Training Result.

Our Genetic Algorithm

Initial Population

An individual is represented by a list of weights and biases of MLP. We use weights and bias of top node to bottom node of each layer to create one individual, for an example: from 3-2-1 network in fig. 1 an individual is represented by ($w_1, w_2, w_3, b_1, w_4, w_5, w_6, b_2, w_7, w_8, b_3$).

We set the numbers of individual in a population to 25 and for each individual the weights are random number in range $[-1.0, 1.0]$, and bias of each node is set to 1.0.

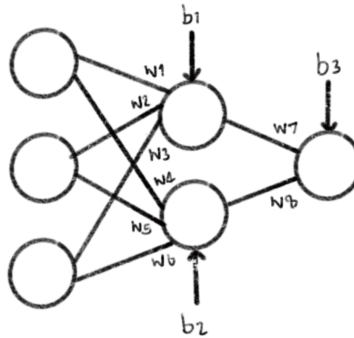


Figure 1: The 3-2-1 network.

Fitness Function

We use both accuracy and mean squared error as the fitness value following the equation eq. (1) where i is the individual and $accuracy_i, MSE_i$ are that individual accuracy and MSE from running through the full training set.

$$f(i) = accuracy_i + \frac{0.001}{MSE_i} \quad (1)$$

Selection

We use the binary deterministic tournament with reinsertion (implementation on 3) as the selection method to select and clone 25 individual to mating pool.

Crossover

We random 2 parent from mating pool to be dad and mom, then perform a crossover by doing a modified uniform crossover with $p_{at.i} = 0.5$ ([Aue13] page 113) that only produce 1 child with each position on chromosome has an equal chance to be from dad or mom (implementation on 2). We will perform crossover until we have 25 children for P^2 .

Mutation

We use strong mutation ([Aue13] page 114) with $p_m = 0.02$ on randomly selected 20 individuals from P^2 (implementation on 2).

Full Process

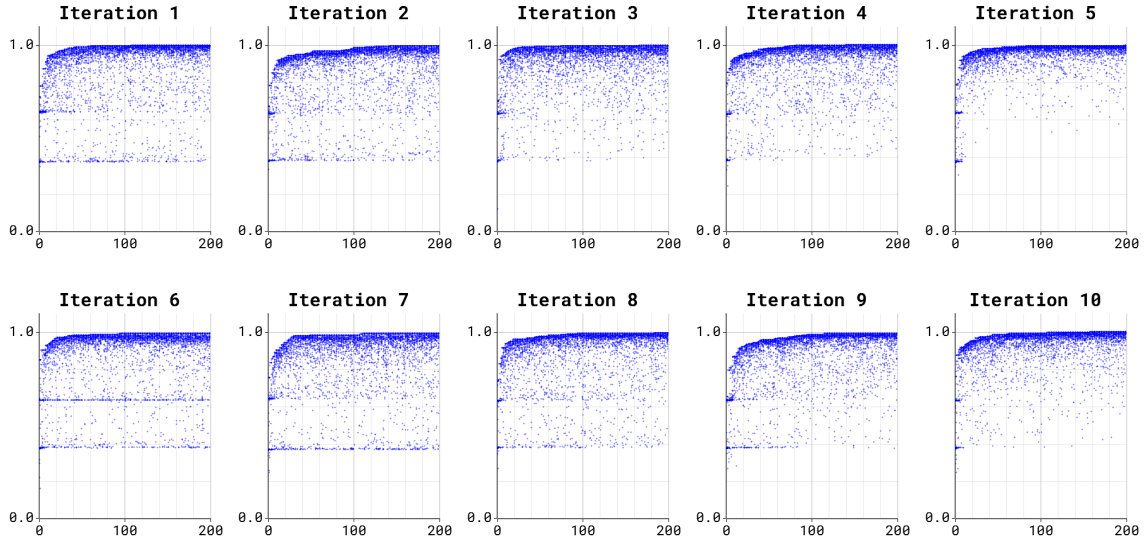
Using 10% cross-validation, and only preprocess each iteration training and validation set with min-max normalization to avoid data leakage as state on [Bro]. The min-max normalization process is done by for each feature F on training set we find $\max(F)$ and $\min(F)$ then for each datapoint F_x we compute new datapoint on both training set and validation set $F'_x = \frac{F_x - \min(F)}{\max(F) - \min(F)}$, this will guarantee that we applied the min-max normalization using \min and \max from training set on both training set and validation set. Next, for each cross-validation iteration we follow these steps (implementation on 4):

1. Initialize the population as state on Initial Population
2. For each individual on population we evaluate its fitness as state on Fitness Function and mark the individual that has the largest fitness.
3. We then process through Selection, Crossover, and Mutation to get 20 individuals.
4. For the remaining 5 individual needed, we use clones of the individual that has largest fitness from step 2 to add to the population (elitism [Aue13] on page 107).
5. Repeat step 2-4 until we fully run through 200 generations and store the individual that has the largest fitness over all generations.
6. Use that individual from step 5 to test on training and validation set.

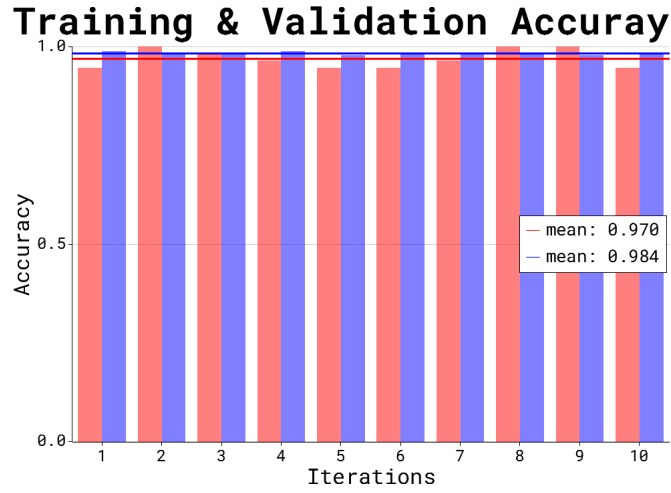
Training Result

We will experiment with 3 models which are wdbc-30-15-1, wdbc-30-7-1, and wdbc-30-15-7-1 to see if their training result will have any significant differences in training time and accuracy (implementation on 4 and we use rust compiler with release profile to build and run all trainings).

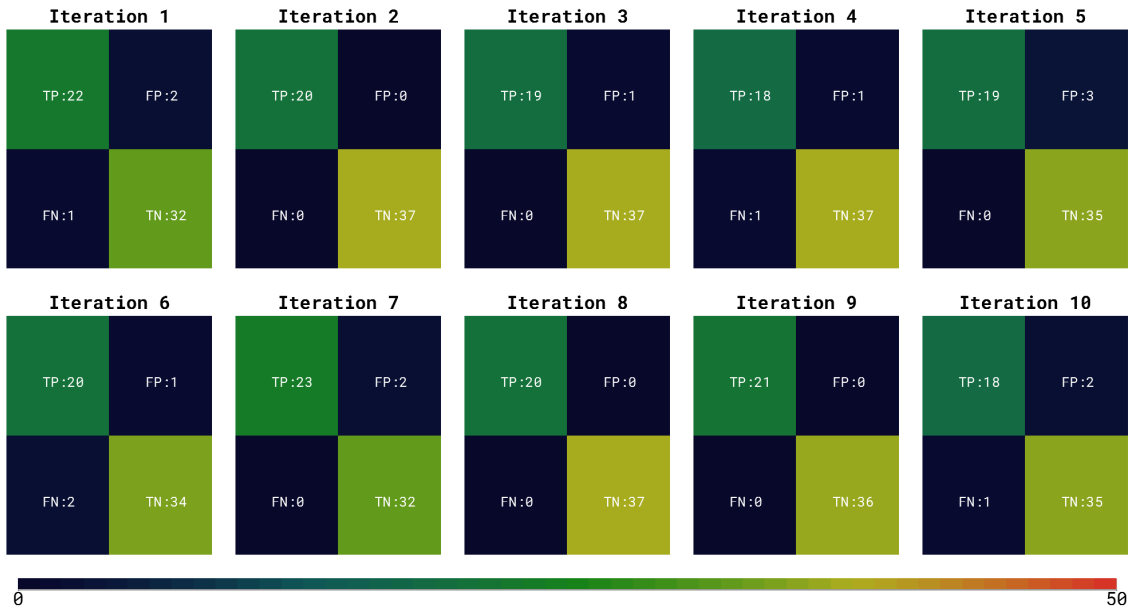
- **wdbc-30-15-1** : The base model that contains 30 input nodes, 1 hidden layer with 15 nodes, and 1 output node with all nodes using sigmoid as an activation function. We assume that this model will have accuracy $> 95\%$ with reasonable training time used. The result is shown on fig. 2.
- **wdbc-30-7-1** : A smaller model with 30 input nodes, 1 hidden layer with 7 nodes, and 1 output node. We assume that this model will have faster training time but with less accuracy than the wdbc-30-15-1. The result is shown on fig. 3
- **wdbc-30-15-7-1** : A larger model with 30 input nodes, 2 hidden layers with 15 and 7 nodes, and 1 output node. We assume that this model will have accuracy $> 98\%$ with longer training used than the wdbc-30-15-1. The result is shown on fig. 4



(a) The training process of each cross-validation iteration: x-axis is the generation, y-axis is the fitness value, and each blue dot is an individual in x generation with y fitness.

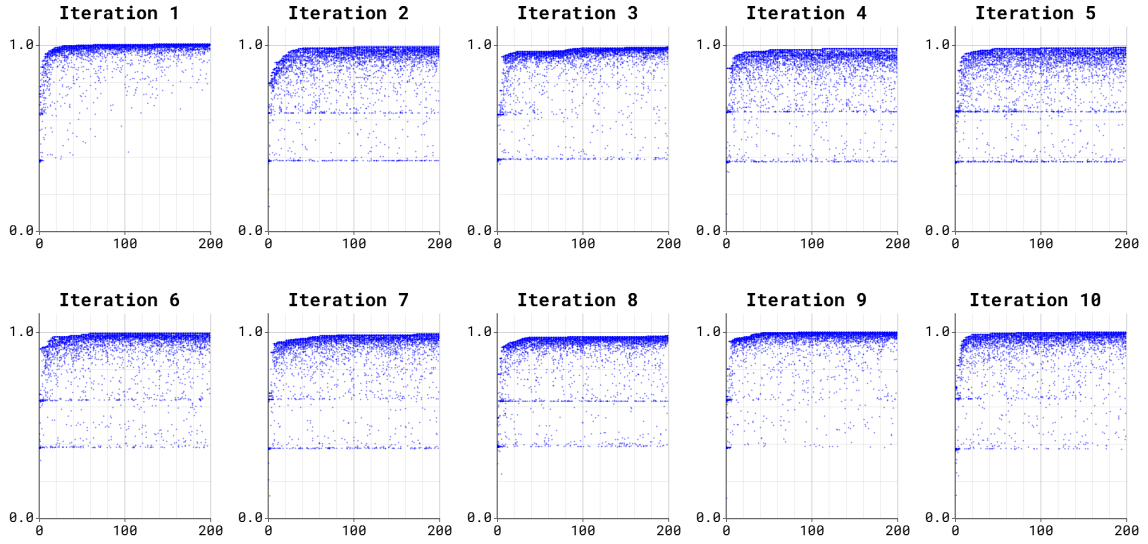


(b) The best individual from each cross-validation iteration accuracy on training set (blue) and validation set (red).

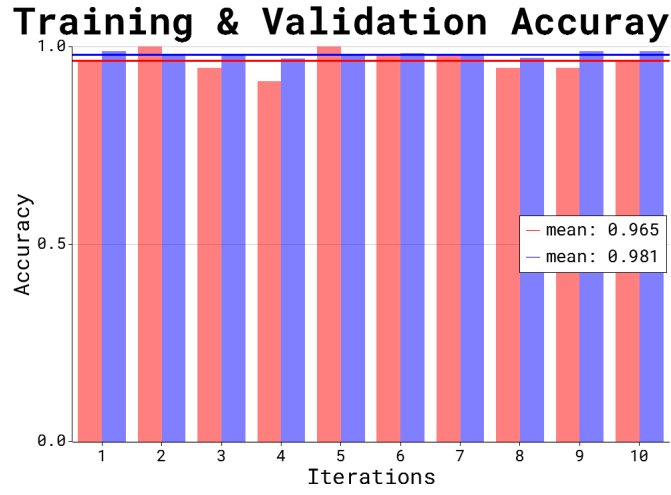


(c) The best individual from each cross-validation iteration confusion matrix on validation set.

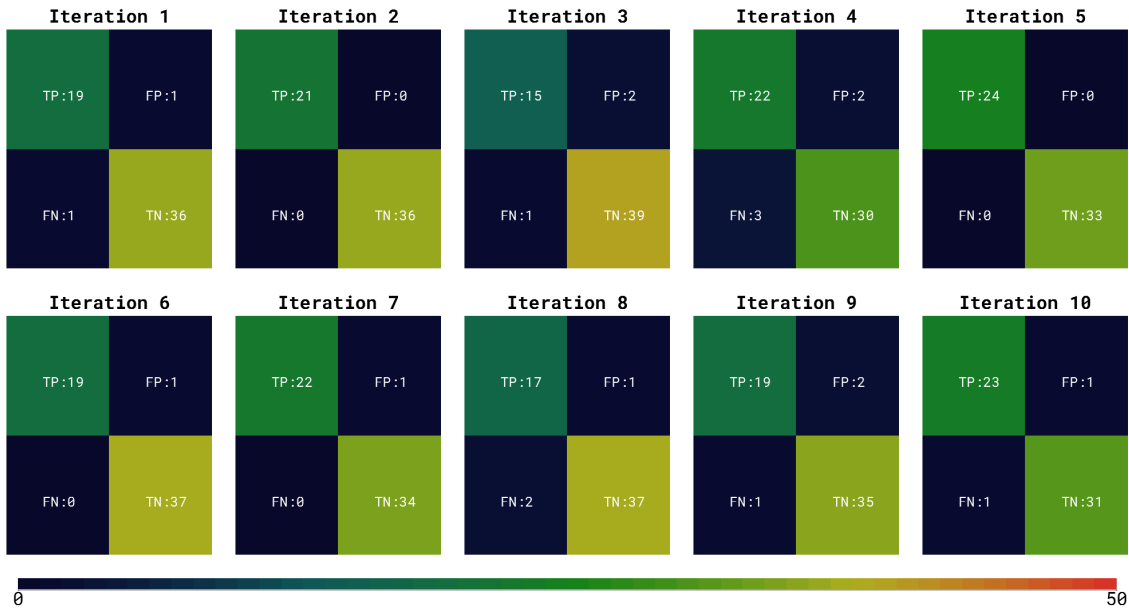
Figure 2: Training result of wdbc-30-15-1 with 20.609 seconds used for training.



(a) The training process of each cross-validation iteration: x-axis is the generation, y-axis is the fitness value, and each blue dot is an individual in x generation with y fitness.

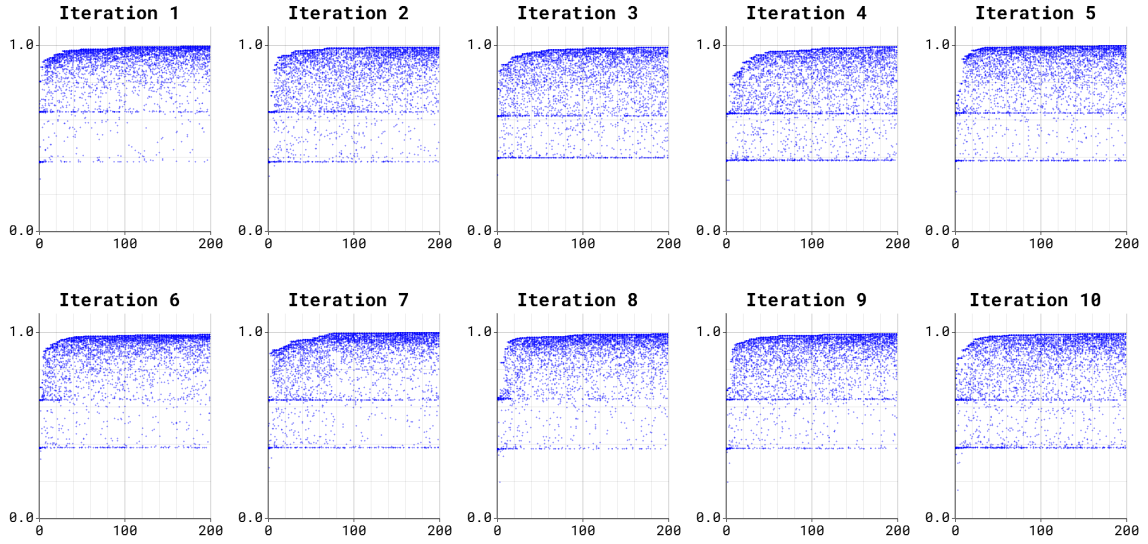


(b) The best individual from each cross-validation iteration accuracy on training set (blue) and validation set (red).

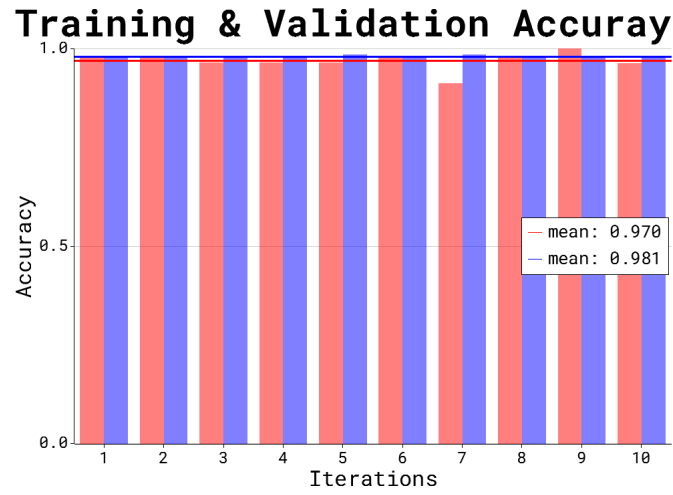


(c) The best individual from each cross-validation iteration confusion matrix on validation set.

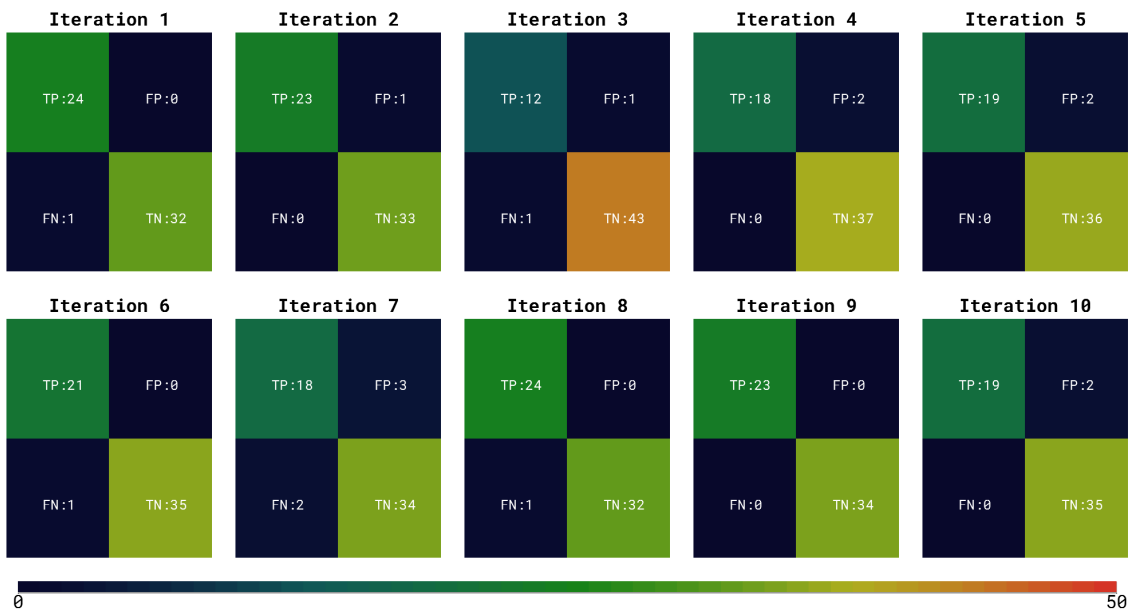
Figure 3: Training result of wdbc-30-7-1 with 14.163 seconds used for training.



(a) The training process of each cross-validation iteration: x-axis is the generation, y-axis is the fitness value, and each blue dot is an individual in x generation with y fitness.



(b) The best individual from each cross-validation iteration accuracy on training set (blue) and validation set (red).



(c) The best individual from each cross-validation iteration confusion matrix on validation set.

Figure 4: Training result of wdbc-30-15-7-1 with 24.244 seconds used for training.

Analysis

From table 1, we can see that there are no significant accuracy differences in every model which is not matching with our assumption. The reason may be that the wdbc dataset is not complex enough for the model that is larger than wdbc-30-7-1. However, the training time used for every model matches our assumption that wdbc-30-7-1 use the least time and wdbc-30-15-7-1 uses the most time. Next, we can see the convergence speed of each model on fig. 2a, fig. 3a, and fig. 4a which for all model the best individual seems to reach fitness value near 1.0 in less than 100 generations. Also, the fitness value around 1.0 seems to be the barrier for every model which the reason should be because of our fitness function that uses both accuracy and MSE to help with the overfitting problem when looking only at MSE (backpropagation method).

| Model | Training Time (seconds) | Validation Set Mean Accuracy (%) |
|----------------|-------------------------|----------------------------------|
| wdbc-30-15-1 | 20.609 | 97.0 |
| wdbc-30-7-1 | 14.163 | 96.5 |
| wdbc-30-15-7-1 | 24.244 | 97.0 |

Table 1: Training time and validation set mean accuracy (red line on fig. 2b, fig. 3b, and fig. 4b) of each model.

Summary

Genetic Algorithm (GA) is an okay algorithm to use for training MLP if we know how we should design a fitness function and how to implement GA with efficiency. GA can train MLP to create a model that is usable as we demonstrated on Training Result. Rust language is also a great tool for implementing GA because of how fast it is and how easy it is to write a memory-safe program.

References

- [Aue13] Sansanee Auephanwiriyaikul. *Introduction to Computational Intelligence for Computer Engineering*. 2013. URL: http://myweb.cmu.ac.th/sansanee.a/Intro_CI_withwatermark.pdf. (accessed: 19.10.2022).
- [Bro] Jason Brownlee. *Data Leakage in Machine Learning*. URL: <https://machinelearningmastery.com/data-leakage-machine-learning/>. (accessed: 01.09.2016).

Appendix

Source Code 1: main.rs

```
1 pub mod activator;
2 pub mod ga;
3 pub mod loss;
4 pub mod mlp;
5 pub mod models;
6 pub mod utils;
7
8 use std::error::Error;
9 fn main() -> Result<(), Box<dyn Error>> {
10     models::wdbc::wdbc_30_15_1();
11     models::wdbc::wdbc_30_7_1();
12     models::wdbc::wdbc_30_15_7_1();
13     Ok(())
14 }
```

Source Code 2: ga/mod.rs

```
1 ///! Genetic Algorithm Utility
2 pub mod selection;
3 use rand::{distributions::Uniform, prelude::Distribution, seq::SliceRandom, Rng};
4 use std::f64::consts::E;
5
6 use crate::mlp::Net;
7
8 #[derive(Clone)]
9 pub struct Individual {
10     pub chromosome: Vec<f64>,
11     pub fitness: f64,
12 }
13
14 impl Individual {
15     pub fn new(chromosome: Vec<f64>) -> Individual {
16         Individual {
17             chromosome,
18             fitness: 0.0,
19         }
20     }
21
22     pub fn set_fitness(&mut self, v: f64) {
23         self.fitness = v;
24     }
25 }
26
27 /// return result of mating of individual in the pool
28 pub fn mating(pop: &Vec<Individual>) -> Vec<Individual> {
29     let mut rand = rand::thread_rng();
30     let new_pop: Vec<Individual> = pop
31         .iter()
32         .map(|_| {
33             let parent: Vec<_> = pop.choose_multiple(&mut rand::thread_rng(), 2).collect();
34             let new_chromosome: Vec<f64> = parent[0]
35                 .chromosome
36                 .iter()
37                 .zip(parent[1].chromosome.iter())
38                 .map(|(p0, p1)| if rand.gen_bool(0.5) { *p0 } else { *p1 })
39                 .collect();
40             Individual::new(new_chromosome)
41         })
42         .collect();
43     new_pop
44 }
45
46 /// strong mutation
47 pub fn mutate(pop: &Vec<Individual>, amount: usize, p_m: f64) -> Vec<Individual> {
48     let mut rand = rand::thread_rng();
49     let new_pop: Vec<Individual> = pop
50         .choose_multiple(&mut rand::thread_rng(), amount)
51         .into_iter()
52         .map(|ind| {
53             let mut ind_clone = ind.clone();
```

```

54         for gene in ind_clone.chromosome.iter_mut() {
55             let between = Uniform::from(0.0..=1.0);
56             if between.sample(&mut rand) < p_m {
57                 let change = 2f64 * rand::random::<f64>() - 1f64;
58                 *gene += change;
59             }
60         }
61         ind_clone
62     })
63     .collect();
64     new_pop
65 }
66
67 /// non-uniform strong mutation
68 pub fn mutate_nonuni(
69     pop: &Vec<Individual>,
70     amount: usize,
71     p_m: f64,
72     curr_gen: usize,
73 ) -> Vec<Individual> {
74     let mut new_pop: Vec<Individual> = vec![];
75     let mut rand = rand::thread_rng();
76     let beta = 1.0;
77     for i in 0..amount {
78         let mut ind_clone = pop[i].clone();
79         for j in 0..pop[i].chromosome.len() {
80             let between = Uniform::from(0.0..=1.0);
81             if between.sample(&mut rand) < (p_m * E.powf(-beta * curr_gen as f64)) {
82                 let change = 2f64 * rand::random::<f64>() - 1f64;
83                 ind_clone.chromosome[j] += change;
84             }
85         }
86         new_pop.push(ind_clone);
87     }
88     new_pop
89 }
90
91 /// Create initial population of MLP from layers
92 ///
93 /// return: population
94 pub fn init_pop(net: &Net, amount: u32) -> Vec<Individual> {
95     let mut pop: Vec<Individual> = vec![];
96     for _ in 0..(amount) {
97         let mut chromosome: Vec<f64> = vec![];
98         for l in &net.layers {
99             for output in &l.w {
100                 for _ in output {
101                     // new random weight in range [-1, 1]
102                     chromosome.push(2f64 * rand::random::<f64>() - 1f64);
103                 }
104             }
105             for bias in &l.b {
106                 chromosome.push(*bias);
107             }
108         }
109         pop.push(Individual::new(chromosome));
110     }
111     pop
112 }
113
114 /// assign individual weight to net
115 pub fn assign_ind(net: &mut Net, individual: &Individual) {
116     if net.parameters != individual.chromosome.len() as u64 {
117         panic!["The neural network parameters size is not equal to individual size"];
118     }
119     let mut idx: usize = 0;
120
121     for l in &mut net.layers {
122         l.w.iter_mut().for_each(|w_j| {
123             w_j.iter_mut().for_each(|w_ji| {
124                 *w_ji = individual.chromosome[idx];
125                 idx += 1;
126             })
127         });
128         l.b.iter_mut().for_each(|b_i| {

```



```

130         *b_i = individual.chromosome[idx];
131         idx += 1;
132     });
133 }
134 }
135
136 #[cfg(test)]
137 mod tests {
138     use super::*;
139     use crate::{
140         activator,
141         mlp::{self, Layer},
142     };
143
144     #[test]
145     fn test_init_pop() {
146         let mut layers: Vec<mlp::Layer> = vec![];
147         layers.push(Layer::new(4, 2, 1.0, activator::sigmoid()));
148         layers.push(Layer::new(2, 1, 1.0, activator::sigmoid()));
149         let net = Net::from_layers(layers);
150         let pop = init_pop(&net, 5);
151
152         assert_eq!(pop.len(), 5);
153         assert_eq!(pop[0].chromosome.len() as u64, net.parameters);
154         // check if bias is the same.
155         assert_eq!(pop[0].chromosome[8], 1.0);
156         assert_eq!(pop[0].chromosome[9], 1.0);
157         assert_eq!(pop[0].chromosome[12], 1.0);
158     }
159
160     #[test]
161     fn test_assign_ind() {
162         let mut layers: Vec<mlp::Layer> = vec![];
163         layers.push(Layer::new(3, 1, 1.0, activator::sigmoid()));
164         layers.push(Layer::new(1, 1, 1.0, activator::sigmoid()));
165         let mut net = Net::from_layers(layers);
166
167         let individual = Individual::new(vec![2.5, 2.3, 2.1, 1.2, 1.3, 4.0]);
168         assign_ind(&mut net, &individual);
169
170         // check if network has been mutated correctly or not.
171         let mut idx = 0;
172         for l in net.layers {
173             for output in l.w {
174                 for w in output {
175                     assert_eq!(w, individual.chromosome[idx]);
176                     idx += 1;
177                 }
178             }
179             for b in l.b {
180                 assert_eq!(b, individual.chromosome[idx]);
181                 idx += 1;
182             }
183         }
184     }
185
186     #[test]
187     fn test_mating_and_mutate() {
188         let mut pop: Vec<Individual> = vec![];
189         for i in 0..4 {
190             let v = i as f64 + 1.0;
191             pop.push(Individual::new(vec![v, v, v, 1.0]))
192         }
193
194         let res = mating(&pop);
195         let mut_res = mutate(&pop, 4, 0.5);
196         assert_eq!(res.len(), pop.len());
197         assert_eq!(mut_res.len(), pop.len());
198
199         for ind in res {
200             println!("{}", ind.chromosome);
201         }
202         for ind in mut_res {
203             println!("{}", ind.chromosome);
204         }
205     }

```

206 }
207

Source Code 3: ga/selection.rs

```
1 use rand::seq::SliceRandom;
2 use super::Individual;
3
4 /// binary deterministic tournament with reinserion
5 pub fn d_tournament(pop: &Vec<Individual>) -> Vec<Individual> {
6     let mut results: Vec<Individual> = vec![];
7     for _ in 0..pop.len() {
8         let players: Vec<_> = pop.choose_multiple(&mut rand::thread_rng(), 2).collect();
9
10        if players[0].fitness > players[1].fitness {
11            results.push(players[0].clone());
12        } else {
13            results.push(players[1].clone());
14        }
15    }
16    results
17 }
18
```

Source Code 4: models/wdbc.rs

```
1 use std::{error::Error, time::Instant};
2
3 use crate::{
4     activator,
5     ga::{self, Individual},
6     loss,
7     mlp::{self, Layer, Net},
8     utils::{
9         data::{self, confusion_count},
10        graph, io,
11    },
12 };
13
14 const IMGPATH: &str = "report/assignment_3/images";
15
16 pub fn wdbc_30_15_1() {
17     fn model() -> Net {
18         let mut layers: Vec<mlp::Layer> = vec![];
19         layers.push(Layer::new(30, 15, 1.0, activator::sigmoid()));
20         layers.push(Layer::new(15, 1, 1.0, activator::sigmoid()));
21         Net::from_layers(layers)
22     }
23     wdbc_ga(&model, "wdbc-30-15-1", IMGPATH).unwrap();
24 }
25
26 pub fn wdbc_30_7_1() {
27     fn model() -> Net {
28         let mut layers: Vec<mlp::Layer> = vec![];
29         layers.push(Layer::new(30, 7, 1.0, activator::sigmoid()));
30         layers.push(Layer::new(7, 1, 1.0, activator::sigmoid()));
31         Net::from_layers(layers)
32     }
33     wdbc_ga(&model, "wdbc-30-7-1", IMGPATH).unwrap();
34 }
35
36 pub fn wdbc_30_15_7_1() {
37     fn model() -> Net {
38         let mut layers: Vec<mlp::Layer> = vec![];
39         layers.push(Layer::new(30, 15, 1.0, activator::sigmoid()));
40         layers.push(Layer::new(15, 7, 1.0, activator::sigmoid()));
41         layers.push(Layer::new(7, 1, 1.0, activator::sigmoid()));
42         Net::from_layers(layers)
43     }
44     wdbc_ga(&model, "wdbc-30-15-7-1", IMGPATH).unwrap();
45 }
46
47 /// train mlp with genitic algorithm
48 pub fn wdbc_ga(model: &dyn Fn() -> Net, folder: &str, imgpath: &str) -> Result<(), Box<dyn Error>> {
```

```

49 let dataset = data::wdbc_dataset()?;
50 let mut valid_acc: Vec<f64> = vec![];
51 let mut train_acc: Vec<f64> = vec![];
52 let mut train_proc: Vec<Vec<(i32, f64)>> = Vec::with_capacity(10);
53 for _ in 0..10 {
54     train_proc.push(vec![]);
55 }
56
57 let mut matrix_vec: Vec<[[i32; 2]; 2]> = vec![];
58 let threshold = 0.5;
59 let max_gen = 200;
60
61 let start = Instant::now();
62 for (j, dt) in dataset.cross_valid_set(0.1).iter().enumerate() {
63     let mut net = model();
64     let (training_set, validation_set) = dt.0.minmax_norm(&dt.1);
65     let mut loss = loss::Loss::square_err();
66
67     // training with GA
68     let mut pop = ga::init_pop(&net, 25);
69     let mut best_ind = pop[0].clone();
70
71     for k in 0..max_gen {
72         let mut max_fitness = f64::MIN;
73         let mut local_best_ind = pop[0].clone();
74
75         for p in pop.iter_mut() {
76             ga::assign_ind(&mut net, &p);
77             let mut matrix = [[0, 0], [0, 0]];
78             let mut run_loss = 0.0;
79             for data in training_set.get_shuffled() {
80                 let result = net.forward(&data.inputs);
81                 run_loss += loss.criterion(&result, &data.labels);
82                 confusion_count(&mut matrix, &result, &data.labels, threshold);
83             }
84             let fitness = ((matrix[0][0] + matrix[1][1]) as f64 / training_set.len() as f64)
85                 + 0.001 / (run_loss / training_set.len() as f64);
86             p.set_fitness(fitness);
87             train_proc[j].push((k, fitness)); // track training progress
88
89             if fitness > max_fitness {
90                 max_fitness = fitness;
91                 local_best_ind = p.clone();
92             }
93             // store best individual for all generation
94             if best_ind.fitness < fitness {
95                 best_ind = p.clone();
96             }
97         }
98
99         // selection
100         let p1 = ga::selection::d_tournament(&pop);
101         let mating_result = ga::mating(&p1);
102         let mut mut_result = ga::mutate(&mating_result, 20, 0.02);
103
104         let mut new_pop: Vec<Individual> = vec![];
105         new_pop.append(&mut mut_result);
106         let pop_need = pop.len() - new_pop.len();
107
108         // elitism
109         for _ in 0..pop_need {
110             new_pop.push(local_best_ind.clone());
111         }
112
113         pop = new_pop;
114         println!("[{}, {}] max_fitness: {:.3}", j, k, max_fitness);
115     }
116
117     ga::assign_ind(&mut net, &best_ind);
118     let mut matrix = [[0, 0], [0, 0]];
119     for data in validation_set.get_datas() {
120         let result = net.forward(&data.inputs);
121         confusion_count(&mut matrix, &result, &data.labels, threshold);
122     }
123     valid_acc.push((matrix[0][0] + matrix[1][1]) as f64 / validation_set.len() as f64);
124     matrix_vec.push(matrix);

```

```

125     let mut matrix_t = [[0, 0], [0, 0]];
126     for data in training_set.get_datas() {
127         let result = net.forward(&data.inputs);
128         confusion_count(&mut matrix_t, &result, &data.labels, threshold);
129     }
130     train_acc.push((matrix_t[0][0] + matrix_t[1][1]) as f64 / training_set.len() as f64);
131     //io::save(&net.layers, format!("models/{}/{}.json", folder, j))?;
132 }
133 let duration = start.elapsed();
134 println!("Time used: {:.3} sec", duration.as_secs_f32());
135
136 graph::draw_acc_2hist(
137     [&valid_acc, &train_acc],
138     "Training & Validation Accuray",
139     ("Iterations", "Accuracy"),
140     format!("{}/{}/accuracy.png", imgpath, folder),
141 )?;
142 graph::draw_confusion(matrix_vec, format!("{}/{}/conf_mat.png", imgpath, folder))?;
143 graph::draw_ga_progress(
144     &train_proc,
145     format!("{}/{}/train_proc.png", imgpath, folder),
146 )?;
147
148 Ok(())
149 }

```

Source Code 5: mlp.rs

```

1 use crate::activator;
2
3 #[derive(Debug)]
4 pub struct Layer {
5     pub inputs: Vec<f64>,
6     pub outputs: Vec<f64>, // need to save this for backward pass
7     pub w: Vec<Vec<f64>>,
8     pub b: Vec<f64>,
9     pub grads: Vec<Vec<f64>>,
10    pub w_prev_changes: Vec<Vec<f64>>,
11    pub local_grads: Vec<f64>,
12    pub b_prev_changes: Vec<f64>,
13    pub act: activator::ActivationContainer,
14 }
15
16 impl Layer {
17     pub fn new(
18         input_features: u64,
19         output_features: u64,
20         bias: f64,
21         act: activator::ActivationContainer,
22     ) -> Layer {
23         // initialize weights matrix
24         let mut weights: Vec<Vec<f64>> = vec![];
25         let mut inputs: Vec<f64> = vec![];
26         let mut outputs: Vec<f64> = vec![];
27         let mut grads: Vec<Vec<f64>> = vec![];
28         let mut local_grads: Vec<f64> = vec![];
29         let mut w_prev_changes: Vec<Vec<f64>> = vec![];
30         let mut b_prev_changes: Vec<f64> = vec![];
31         let mut b: Vec<f64> = vec![];
32
33         for _ in 0..output_features {
34             outputs.push(0.0);
35             local_grads.push(0.0);
36             b_prev_changes.push(0.0);
37             b.push(bias);
38
39             let mut w: Vec<f64> = vec![];
40             let mut g: Vec<f64> = vec![];
41             for _ in 0..input_features {
42                 if (inputs.len() as u64) < input_features {
43                     inputs.push(0.0);
44                 }
45                 g.push(0.0);
46                 // random both positive and negative weight
47                 w.push(2f64 * rand::random::<f64>() - 1f64);

```

```

48         }
49         weights.push(w);
50         grads.push(g.clone());
51         w_prev_changes.push(g);
52     }
53     Layer {
54         inputs,
55         outputs,
56         w: weights,
57         b,
58         grads,
59         w_prev_changes,
60         local_grads,
61         b_prev_changes,
62         act,
63     }
64 }
65
66 pub fn forward(&mut self, inputs: &Vec<f64>) -> Vec<f64> {
67     if inputs.len() != self.inputs.len() {
68         panic!("forward: input size is wrong");
69     }
70
71     let result: Vec<f64> = self
72         .w
73         .iter()
74         .zip(self.b.iter())
75         .zip(self.outputs.iter_mut())
76         .map(|((w_j, b_j), o_j)| {
77             let sum = inputs
78                 .iter()
79                 .zip(w_j.iter())
80                 .fold(0.0, |s, (v, w_ji)| s + w_ji * v)
81                 + b_j;
82             *o_j = sum;
83             (self.act.func)(sum)
84         })
85         .collect();
86
87     self.inputs = inputs.clone();
88     result
89 }
90
91 pub fn update(&mut self, lr: f64, momentum: f64) {
92     for j in 0..self.w.len() {
93         let delta_b = lr * self.local_grads[j] + momentum * self.b_prev_changes[j];
94         self.b[j] -= delta_b; // update each neuron bias
95         self.b_prev_changes[j] = delta_b;
96         for i in 0..self.w[j].len() {
97             // update each weights
98             let delta_w = lr * self.grads[j][i] + momentum * self.w_prev_changes[j][i];
99             self.w[j][i] -= delta_w;
100             self.w_prev_changes[j][i] = delta_w;
101         }
102     }
103 }
104
105 pub fn zero_grad(&mut self) {
106     for j in 0..self.outputs.len() {
107         self.local_grads[j] = 0.0;
108         for i in 0..self.grads[j].len() {
109             self.grads[j][i] = 0.0;
110         }
111     }
112 }
113 }
114
115 #[derive(Debug)]
116 pub struct Net {
117     pub layers: Vec<Layer>,
118     pub parameters: u64,
119 }
120
121 impl Net {
122     pub fn from_layers(layers: Vec<Layer>) -> Net {
123         let mut parameters: u64 = 0;

```

```

124     for l in &layers {
125         parameters += (l.w.len() * l.w[0].len()) as u64;
126         parameters += l.b.len() as u64;
127     }
128
129     Net { layers, parameters }
130 }
131
132 pub fn new(architecture: Vec<u64>) -> Net {
133     let mut layers: Vec<Layer> = vec![];
134     for i in 1..architecture.len() {
135         layers.push(Layer::new(
136             architecture[i - 1],
137             architecture[i],
138             1f64,
139             activator::sigmoid(),
140         ))
141     }
142     Net::from_layers(layers)
143 }
144
145 pub fn zero_grad(&mut self) {
146     for l in 0..self.layers.len() {
147         self.layers[l].zero_grad();
148     }
149 }
150
151 pub fn forward(&mut self, input: &Vec<f64>) -> Vec<f64> {
152     let mut result = self.layers[0].forward(input);
153     for l in 1..self.layers.len() {
154         result = self.layers[l].forward(&result);
155     }
156     result
157 }
158
159 pub fn update(&mut self, lr: f64, momentum: f64) {
160     for l in 0..self.layers.len() {
161         self.layers[l].update(lr, momentum);
162     }
163 }
164 }
165
166 #[cfg(test)]
167 mod tests {
168     use super::*;
169
170     #[test]
171     fn test_linear_new() {
172         let linear = Layer::new(2, 3, 1.0, activator::linear());
173         assert_eq!(linear.outputs.len(), 3);
174         assert_eq!(linear.inputs.len(), 2);
175
176         assert_eq!(linear.w.len(), 3);
177         assert_eq!(linear.w[0].len(), 2);
178         assert_eq!(linear.b.len(), 3);
179
180         assert_eq!(linear.grads.len(), 3);
181         assert_eq!(linear.w_prev_changes.len(), 3);
182         assert_eq!(linear.grads[0].len(), 2);
183         assert_eq!(linear.w_prev_changes[0].len(), 2);
184         assert_eq!(linear.local_grads.len(), 3);
185         assert_eq!(linear.b_prev_changes.len(), 3);
186     }
187
188     #[test]
189     fn test_linear_forward1() {
190         let mut linear = Layer::new(2, 1, 1.0, activator::sigmoid());
191
192         for j in 0..linear.w.len() {
193             for i in 0..linear.w[j].len() {
194                 linear.w[j][i] = 1.0;
195             }
196         }
197
198         assert_eq!(linear.forward(&vec![1.0, 1.0])[0], 0.9525741268224334);
199         assert_eq!(linear.outputs[0], 3.0);

```

```

200     }
201
202     #[test]
203     fn test_linear_forward2() {
204         let mut linear = Layer::new(2, 2, 1.0, activator::sigmoid());
205
206         for j in 0..linear.w.len() {
207             for i in 0..linear.w[j].len() {
208                 linear.w[j][i] = (j as f64) + 1.0;
209             }
210         }
211         let result = linear.forward(&vec![0.0, 1.0]);
212         assert_eq!(linear.outputs[0], 2.0);
213         assert_eq!(linear.outputs[1], 3.0);
214         assert_eq!(result[0], 0.8807970779778823);
215         assert_eq!(result[1], 0.9525741268224334);
216     }
217 }

```

Source Code 6: activator.rs

```

1  #[derive(Debug)]
2  pub struct ActivationContainer {
3      pub func: fn(f64) -> f64,
4      pub der: fn(f64) -> f64,
5      pub name: String,
6  }
7
8  pub fn sigmoid() -> ActivationContainer {
9      fn func(input: f64) -> f64 {
10         1.0 / (1.0 + (-input).exp())
11     }
12     fn der(input: f64) -> f64 {
13         func(input) * (1.0 - func(input))
14     }
15     ActivationContainer {
16         func,
17         der,
18         name: "sigmoid".to_string(),
19     }
20 }
21
22 pub fn relu() -> ActivationContainer {
23     fn func(input: f64) -> f64 {
24         return f64::max(0.0, input);
25     }
26     fn der(input: f64) -> f64 {
27         if input > 0.0 {
28             return 1.0;
29         } else {
30             return 0.0;
31         }
32     }
33     ActivationContainer {
34         func,
35         der,
36         name: "relu".to_string(),
37     }
38 }
39
40 pub fn linear() -> ActivationContainer {
41     fn func(input: f64) -> f64 {
42         input
43     }
44     fn der(_input: f64) -> f64 {
45         1.0
46     }
47     ActivationContainer {
48         func,
49         der,
50         name: "linear".to_string(),
51     }
52 }
53
54 #[cfg(test)]

```

```

55 mod tests {
56     use super::*;
57
58     #[test]
59     fn test_sigmoid() {
60         let act = sigmoid();
61
62         assert_eq!((act.func)(1.0), 0.7310585786300048792512);
63         assert_eq!((act.func)(-1.0), 0.2689414213699951207488);
64         assert_eq!((act.func)(0.0), 0.5);
65         assert_eq!((act.der)(1.0), 0.1966119332414818525374);
66         assert_eq!((act.der)(-1.0), 0.1966119332414818525374);
67         assert_eq!((act.der)(0.0), 0.25);
68     }
69
70     #[test]
71     fn test_relu() {
72         let act = relu();
73
74         assert_eq!((act.func)(-1.0), 0.0);
75         assert_eq!((act.func)(20.0), 20.0);
76         assert_eq!((act.der)(-1.0), 0.0);
77         assert_eq!((act.der)(20.0), 1.0);
78     }
79 }
80

```

Source Code 7: loss.rs

```

1  use crate::mlp;
2
3  pub struct Loss {
4      outputs: Vec<f64>,
5      desired: Vec<f64>,
6      pub func: fn(f64, f64) -> f64,
7      pub der: fn(f64, f64) -> f64,
8  }
9
10 impl Loss {
11     /// Squared Error
12     pub fn square_err() -> Loss {
13         fn func(output: f64, desired: f64) -> f64 {
14             0.5 * (output - desired).powi(2)
15         }
16         fn der(output: f64, desired: f64) -> f64 {
17             output - desired
18         }
19
20         Loss {
21             outputs: vec![],
22             desired: vec![],
23             func,
24             der,
25         }
26     }
27
28     /// Binary Cross Entropy
29     pub fn bce() -> Loss {
30         fn func(output: f64, desired: f64) -> f64 {
31             -desired * output.ln() + (1.0 - desired) * (1.0 - output).ln()
32         }
33         fn der(output: f64, desired: f64) -> f64 {
34             -(desired / output - (1.0 - desired) / (1.0 - output))
35         }
36
37         Loss {
38             outputs: vec![],
39             desired: vec![],
40             func,
41             der,
42         }
43     }
44
45     pub fn criterion(&mut self, outputs: &Vec<f64>, desired: &Vec<f64>) -> f64 {
46         if outputs.len() != desired.len() {

```



```

47         panic!("outputs size is not equal to desired size");
48     }
49     let loss = outputs
50         .iter()
51         .zip(desired.iter())
52         .fold(0.0, |ls, (o, d)| ls + (self.func)(*o, *d));
53     self.outputs = outputs.clone();
54     self.desired = desired.clone();
55     loss
56 }
57
58 pub fn backward(&self, layers: &mut Vec<mlp::Layer>) {
59     for l in (0..layers.len()).rev() {
60         // output layer
61         if l == layers.len() - 1 {
62             for j in 0..layers[l].outputs.len() {
63                 // compute grads
64                 let local_grad = (self.der)(self.outputs[j], self.desired[j])
65                     * (layers[l].act.der)(layers[l].outputs[j]);
66
67                 layers[l].local_grads[j] = local_grad;
68
69                 // set grads for each weight
70                 for k in 0..(layers[l - 1].outputs.len()) {
71                     layers[l].grads[j][k] =
72                         (layers[l - 1].act.func)(layers[l - 1].outputs[k]) * local_grad;
73                 }
74             }
75             continue;
76         }
77         // hidden layer
78         for j in 0..layers[l].outputs.len() {
79             // calculate local_grad based on previous local_grad
80             let mut local_grad = 0f64;
81             for i in 0..layers[l + 1].w.len() {
82                 for k in 0..layers[l + 1].w[i].len() {
83                     local_grad += layers[l + 1].w[i][k] * layers[l + 1].local_grads[i];
84                 }
85             }
86             local_grad = (layers[l].act.der)(layers[l].outputs[j]) * local_grad;
87             layers[l].local_grads[j] = local_grad;
88
89             // set grads for each weight
90             if l == 0 {
91                 for k in 0..layers[l].inputs.len() {
92                     layers[l].grads[j][k] = layers[l].inputs[k] * local_grad;
93                 }
94             } else {
95                 for k in 0..layers[l - 1].outputs.len() {
96                     layers[l].grads[j][k] =
97                         (layers[l - 1].act.func)(layers[l - 1].outputs[k]) * local_grad;
98                 }
99             }
100         }
101     }
102 }
103
104 #[cfg(test)]
105 mod tests {
106     use super::*;
107
108     #[test]
109     fn test_mse_func() {
110         assert_eq!((Loss::square_err().func)(2.0, 1.0), 0.5);
111         assert_eq!((Loss::square_err().func)(5.0, 0.0), 12.5);
112     }
113
114     #[test]
115     fn test_mse_der() {
116         assert_eq!((Loss::square_err().der)(2.0, 1.0), 1.0);
117         assert_eq!((Loss::square_err().der)(5.0, 0.0), 5.0);
118     }
119
120     #[test]
121     fn test_mse() {

```

```

123     let mut loss = Loss::square_err();
124
125     let l = loss.criterion(&vec![2.0, 1.0, 0.0], &vec![0.0, 1.0, 2.0]);
126     assert_eq!(l, 4.0);
127
128     loss.criterion(
129         &vec![34.0, 37.0, 44.0, 47.0, 48.0],
130         &vec![37.0, 40.0, 46.0, 44.0, 46.0],
131     );
132     assert_eq!(l, 4.0);
133 }
134
135 #[test]
136 fn test_bce_func() {
137     println!("{}", (Loss::bce().func)(0.9, 0.0));
138     println!("{}", (Loss::bce().func)(0.9, 1.0));
139 }
140 }
141

```

Source Code 8: utils/data.rs

```

1  use super::io::read_lines;
2  use rand::prelude::SliceRandom;
3  use serde::Deserialize;
4  use std::error::Error;
5
6  pub fn max(vec: &Vec<f64>) -> f64 {
7      vec.iter().fold(f64::NAN, |max, &v| v.max(max))
8  }
9
10 pub fn min(vec: &Vec<f64>) -> f64 {
11     vec.iter().fold(f64::NAN, |min, &v| v.min(min))
12 }
13
14 pub fn std(vec: &Vec<f64>, mean: f64) -> f64 {
15     let n = vec.len() as f64;
16     vec.iter()
17         .fold(0.0f64, |sum, &val| sum + (val - mean).powi(2) / n)
18         .sqrt()
19 }
20
21 pub fn mean(vec: &Vec<f64>) -> f64 {
22     let n = vec.len() as f64;
23     vec.iter().fold(0.0f64, |mean, &val| mean + val / n)
24 }
25
26 pub fn standardization(data: &Vec<f64>, mean: f64, std: f64) -> Vec<f64> {
27     data.iter().map(|x| (x - mean) / std).collect()
28 }
29
30 pub fn minmax_norm(data: &Vec<f64>, min: f64, max: f64) -> Vec<f64> {
31     data.iter().map(|x| (x - min) / (max - min)).collect()
32 }
33
34 #[derive(Debug, Clone)]
35 pub struct Data {
36     pub inputs: Vec<f64>,
37     pub labels: Vec<f64>,
38 }
39
40 #[derive(Clone)]
41 pub struct DataSet {
42     datas: Vec<Data>,
43 }
44
45 impl DataSet {
46     pub fn new(datas: Vec<Data>) -> DataSet {
47         DataSet { datas }
48     }
49
50     pub fn cross_valid_set(&self, percent: f64) -> Vec<(DataSet, DataSet)> {
51         if percent < 0.0 && percent > 1.0 {
52             panic!("argument percent must be in range [0, 1]")
53         }
54         let k = (percent * (self.datas.len() as f64)).ceil() as usize; // fold size

```

```

54     let n = (self.datas.len() as f64 / k as f64).ceil() as usize; // number of folds
55     let datas = self.get_shuffled().clone(); // shuffled data before slicing it
56     let mut set: Vec<DataSet, DataSet> = vec![];
57
58     let mut curr: usize = 0;
59     for _ in 0..n {
60         let r_pt: usize = if curr + k > datas.len() {
61             datas.len()
62         } else {
63             curr + k
64         };
65
66         let validation_set: Vec<Data> = datas[curr..r_pt].to_vec();
67         let training_set: Vec<Data> = if curr > 0 {
68             let mut temp = datas[0..curr].to_vec();
69             temp.append(&mut datas[r_pt..datas.len()].to_vec());
70             temp
71         } else {
72             datas[r_pt..datas.len()].to_vec()
73         };
74
75         set.push((DataSet::new(training_set), DataSet::new(validation_set)));
76         curr += k
77     }
78     set
79 }
80
81 pub fn data_points(&self) -> Vec<f64> {
82     let mut data_points: Vec<f64> = vec![];
83     for mut dt in self.datas.clone() {
84         data_points.append(&mut dt.inputs);
85         data_points.append(&mut dt.labels);
86     }
87     data_points
88 }
89
90 pub fn max(&self) -> f64 {
91     max(&self.data_points())
92 }
93
94 pub fn min(&self) -> f64 {
95     min(&self.data_points())
96 }
97
98 pub fn std(&self) -> f64 {
99     std(&self.data_points(), self.mean())
100 }
101
102 pub fn mean(&self) -> f64 {
103     mean(&self.data_points())
104 }
105
106 pub fn len(&self) -> usize {
107     self.datas.len()
108 }
109
110 pub fn standardization(&self) -> DataSet {
111     // this kind of wrong
112     let mean = self.mean();
113     let std = self.std();
114     let datas: Vec<Data> = self
115         .get_datas()
116         .into_iter()
117         .map(|dt| {
118             let inputs: Vec<f64> = standardization(&dt.inputs, mean, std);
119             let labels: Vec<f64> = standardization(&dt.labels, mean, std);
120             Data { inputs, labels }
121         })
122         .collect();
123     DataSet::new(datas)
124 }
125
126 /// this could be implement to be cleaner but I'm lazy
127 pub fn minmax_norm(&self, valid_set: &DataSet) -> (DataSet, DataSet) {
128     // this is very not efficient
129     let size = self.datas[0].inputs.len();

```

```

130     let mut features: Vec<Vec<f64>> = Vec::with_capacity(size);
131     let mut v_features: Vec<Vec<f64>> = Vec::with_capacity(size);
132
133     for _ in 0..size {
134         features.push(vec![]);
135         v_features.push(vec![]);
136     }
137     for dt in self.datas.iter() {
138         for (f, x) in features.iter_mut().zip(dt.inputs.iter()) {
139             f.push(*x);
140         }
141     }
142     for v_dt in valid_set.datas.iter() {
143         for (vf, vx) in v_features.iter_mut().zip(v_dt.inputs.iter()) {
144             vf.push(*vx);
145         }
146     }
147     for (f, vf) in features.iter_mut().zip(v_features.iter_mut()) {
148         let (min, max) = (min(f), max(f));
149         *f = minmax_norm(f, min, max);
150         *vf = minmax_norm(vf, min, max);
151     }
152
153     let datas: Vec<Data> = self
154         .datas
155         .iter()
156         .enumerate()
157         .map(|(i, dt)| {
158             let inputs: Vec<f64> = features.iter().map(|x| x[i]).collect();
159             Data {
160                 labels: dt.labels.clone(),
161                 inputs,
162             }
163         })
164         .collect();
165
166     let v_datas: Vec<Data> = valid_set
167         .datas
168         .iter()
169         .enumerate()
170         .map(|(i, dt)| {
171             let inputs: Vec<f64> = v_features.iter().map(|x| x[i]).collect();
172             Data {
173                 labels: dt.labels.clone(),
174                 inputs,
175             }
176         })
177         .collect();
178
179     (DataSet::new(datas), DataSet::new(v_datas))
180 }
181
182 pub fn get_datas(&self) -> Vec<Data> {
183     self.datas.clone()
184 }
185
186 pub fn get_shuffled(&self) -> Vec<Data> {
187     let mut shuffled_datas = self.datas.clone();
188     shuffled_datas.shuffle(&mut rand::thread_rng());
189     shuffled_datas
190 }
191 }
192
193 pub fn confusion_count(
194     matrix: &mut [[i32; 2]; 2],
195     result: &Vec<f64>,
196     label: &Vec<f64>,
197     threshold: f64,
198 ) {
199     if result[0] > threshold {
200         // true positive
201         if label[0] == 1.0 {
202             matrix[0][0] += 1
203         } else {
204             // false negative
205             matrix[1][0] += 1

```

```

206     }
207 } else if result[0] <= threshold {
208     // true negative
209     if label[0] == 0.0 {
210         matrix[1][1] += 1
211     }
212     // false positive
213     else {
214         matrix[0][1] += 1
215     }
216 }
217 }
218
219 pub fn un_standardization(value: f64, mean: f64, std: f64) -> f64 {
220     value * std + mean
221 }
222
223 pub fn xor_dataset() -> DataSet {
224     let inputs = vec![[0.0, 0.0], [0.0, 1.0], [1.0, 0.0], [1.0, 1.0]];
225     let labels = vec![[0.0], [1.0], [1.0], [0.0]];
226     let mut datas: Vec<Data> = vec![];
227     for i in 0..4 {
228         datas.push(Data {
229             inputs: inputs[i].to_vec(),
230             labels: labels[i].to_vec(),
231         });
232     }
233
234     DataSet::new(datas)
235 }
236
237 pub fn flood_dataset() -> Result<DataSet, Box<dyn Error>> {
238     #[derive(Deserialize)]
239     struct Record {
240         s1_t3: f64,
241         s1_t2: f64,
242         s1_t1: f64,
243         s1_t0: f64,
244         s2_t3: f64,
245         s2_t2: f64,
246         s2_t1: f64,
247         s2_t0: f64,
248         t7: f64,
249     }
250
251     let mut datas: Vec<Data> = vec![];
252     let mut reader = csv::Reader::from_path("data/flood_dataset.csv")?;
253     for record in reader.deserialize() {
254         let record: Record = record?;
255         let mut inputs: Vec<f64> = vec![];
256         // station 1
257         inputs.push(record.s1_t3);
258         inputs.push(record.s1_t2);
259         inputs.push(record.s1_t1);
260         inputs.push(record.s1_t0);
261         // station 2
262         inputs.push(record.s2_t3);
263         inputs.push(record.s2_t2);
264         inputs.push(record.s2_t1);
265         inputs.push(record.s2_t0);
266
267         let labels: Vec<f64> = vec![f64::from(record.t7)];
268         datas.push(Data { inputs, labels });
269     }
270     Ok(DataSet::new(datas))
271 }
272
273 pub fn cross_dataset() -> Result<DataSet, Box<dyn Error>> {
274     let mut datas: Vec<Data> = vec![];
275     let mut lines = read_lines("data/cross.pat")?;
276     while let (Some(_), Some(Ok(l1)), Some(Ok(l2))) = (lines.next(), lines.next(), lines.next()) {
277         let mut inputs: Vec<f64> = vec![];
278         let mut labels: Vec<f64> = vec![];
279         for w in l1.split(" ") {
280             let v: f64 = w.parse().unwrap();
281             inputs.push(v);

```

```

282     }
283     for w in 12.split(" ") {
284         let v: f64 = w.parse().unwrap();
285         // class 1 0 -> 1
286         // class 0 1 -> 0
287         labels.push(v);
288         break;
289     }
290     datas.push(Data { inputs, labels });
291 }
292 Ok(DataSet::new(datas))
293 }
294
295 pub fn wdbc_dataset() -> Result<DataSet, Box<dyn Error>> {
296     let mut datas: Vec<Data> = vec![];
297     let mut lines = read_lines("data/wdbc.txt"?);
298     while let Some(Ok(line)) = lines.next() {
299         let mut inputs: Vec<f64> = vec![];
300         let mut labels: Vec<f64> = vec![]; // M (malignant) = 1.0, B (benign) = 0.0
301         let arr: Vec<&str> = line.split(",").collect();
302         if arr[1] == "M" {
303             labels.push(1.0);
304         } else if arr[1] == "B" {
305             labels.push(0.0);
306         }
307         for w in &arr[2..] {
308             let v: f64 = w.parse()?;
309             inputs.push(v);
310         }
311         datas.push(Data { inputs, labels });
312     }
313     Ok(DataSet::new(datas))
314 }
315
316 #[cfg(test)]
317 mod tests {
318     use super::*;
319
320     #[test]
321     fn temp_test() -> Result<(), Box<dyn Error>> {
322         let dt = wdbc_dataset()?;
323         println!("{}", dt.get_datas()[0].inputs.len());
324
325         /*
326         let dt = flood_dataset()?.cross_valid_set(0.1);
327         let training_set = &dt[0].0;
328         let validation_set = &dt[0].1;
329
330         println!("mean: {}, std: {}", validation_set.mean(), validation_set.std());
331         println!("{}", validation_set.get_datas());
332         println!("{}", standardization(validation_set).get_datas());
333         */
334
335         /*
336         if let Ok(dt) = cross_dataset() {
337             println!("{}", dt.get_datas());
338         }
339         */
340         Ok(())
341     }
342
343     #[test]
344     fn test_min_max() -> Result<(), Box<dyn Error>> {
345         let dt = flood_dataset()?;
346         assert_eq!(dt.max(), 628.0);
347         assert_eq!(dt.min(), 95.0);
348         Ok(())
349     }
350 }
351

```

Source Code 9: utils/graph.rs

```

1  use plotters::coord::Shift;
2  use plotters::prelude::*;
3  use std::error::Error;
4
5  const FONT: &str = "Roboto Mono";
6  const CAPTION: i32 = 70;
7  const SERIE_LABEL: i32 = 32;
8  const AXIS_LABEL: i32 = 40;
9
10 pub struct LossGraph {
11     loss: Vec<Vec<f64>>,
12     valid_loss: Vec<Vec<f64>>,
13 }
14
15 impl LossGraph {
16     pub fn new() -> LossGraph {
17         let loss: Vec<Vec<f64>> = vec![];
18         let valid_loss: Vec<Vec<f64>> = vec![];
19         LossGraph { loss, valid_loss }
20     }
21
22     pub fn add_loss(&mut self, training: Vec<f64>, validation: Vec<f64>) {
23         self.loss.push(training);
24         self.valid_loss.push(validation);
25     }
26     /// Draw training loss and validation loss at each epoch (x_vec)
27     pub fn draw_loss(
28         &self,
29         idx: u32,
30         root: &DrawingArea<BitMapBackend, Shift>,
31         loss_vec: &Vec<f64>,
32         valid_loss_vec: &Vec<f64>,
33         max_loss: f64,
34     ) -> Result<(), Box<dyn Error>> {
35         let min_loss1 = loss_vec.iter().fold(f64::NAN, |min, &val| val.min(min));
36         let min_loss2 = valid_loss_vec
37             .iter()
38             .fold(f64::NAN, |min, &val| val.min(min));
39         let min_loss = if min_loss1.min(min_loss2) > 0.0 {
40             0.0
41         } else {
42             min_loss1.min(min_loss2)
43         };
44
45         let mut chart = ChartBuilder::on(&root)
46             .caption(
47                 format!("Loss {} ", idx),
48                 ("Hack", 44, FontStyle::Bold).into_font(),
49             )
50             .margin(20)
51             .x_label_area_size(50)
52             .y_label_area_size(60)
53             .build_cartesian_2d(0..loss_vec.len(), min_loss..max_loss)?;
54
55         chart
56             .configure_mesh()
57             .y_desc("Loss")
58             .x_desc("Epochs")
59             .axis_desc_style(("Hack", 20))
60             .draw()?;
61
62         chart.draw_series(LineSeries::new(
63             loss_vec.iter().enumerate().map(|(i, x)| (i + 1, *x)),
64             &RED,
65         ))?;
66
67         chart.draw_series(LineSeries::new(
68             valid_loss_vec.iter().enumerate().map(|(i, x)| (i + 1, *x)),
69             &BLUE,
70         ))?;
71
72         root.present()?;
73         Ok(())
74     }
75
76     pub fn max_loss(&self) -> f64 {

```

```

77         f64::max(
78             self.loss.iter().fold(f64::NAN, |max, vec| {
79                 let max_loss = vec.iter().fold(f64::NAN, |max, &val| val.max(max));
80                 f64::max(max_loss, max)
81             }),
82             self.valid_loss.iter().fold(f64::NAN, |max, vec| {
83                 let max_loss = vec.iter().fold(f64::NAN, |max, &val| val.max(max));
84                 f64::max(max_loss, max)
85             }),
86         )
87     }
88
89     pub fn draw(&self, path: String) -> Result<(), Box<dyn Error>> {
90         let root = BitMapBackend::new(&path, (2000, 1000)).into_drawing_area();
91         root.fill(&WHITE)?;
92         // hardcoded for 10 iterations
93         let drawing_areas = root.split_evenly((2, 5));
94
95         let mut loss_iter = self.loss.iter();
96         let mut valid_loss_iter = self.valid_loss.iter();
97         let max_loss = self.max_loss();
98         for (drawing_area, idx) in drawing_areas.iter().zip(1..) {
99             if let (Some(loss_vec), Some(valid_loss_vec)) =
100                 (loss_iter.next(), valid_loss_iter.next())
101             {
102                 self.draw_loss(idx, drawing_area, loss_vec, valid_loss_vec, max_loss)?;
103             }
104         }
105         Ok(())
106     }
107 }
108
109 /// Draw histogram of given datas
110 /// axes_desc - (for x, for y)
111 pub fn draw_acc_hist(
112     datas: &Vec<f64>,
113     title: &str,
114     axes_desc: (&str, &str),
115     path: String,
116 ) -> Result<(), Box<dyn Error>> {
117     let n = datas.len();
118     let mean = datas
119         .iter()
120         .fold(0.0f64, |mean, &val| mean + val / n as f64);
121
122     let root = BitMapBackend::new(&path, (1024, 768)).into_drawing_area();
123     root.fill(&WHITE)?;
124
125     let mut chart = ChartBuilder::on(&root)
126         .caption(title, ("Hack", 44, FontStyle::Bold).into_font())
127         .margin(20)
128         .x_label_area_size(50)
129         .y_label_area_size(60)
130         .build_cartesian_2d((1..n).into_segmented(), 0.0..1.0)?
131         .set_secondary_coord(1..n, 0.0..1.0);
132
133     chart
134         .configure_mesh()
135         .disable_x_mesh()
136         .y_max_light_lines(0)
137         .y_desc(axes_desc.1)
138         .x_desc(axes_desc.0)
139         .axis_desc_style(("Hack", 20))
140         .y_labels(3)
141         .draw()?;
142
143     let hist = Histogram::vertical(&chart)
144         .style(RED.mix(0.5).filled())
145         .margin(10)
146         .data(datas.iter().enumerate().map(|(i, x)| (i + 1, *x)));
147
148     chart.draw_series(hist)?;
149
150     chart
151         .draw_secondary_series(LineSeries::new(
152             datas.iter().enumerate().map(|(i, _)| (i + 1, mean)),

```



```

153         BLUE.filled().stroke_width(2),
154     ))?
155     .label(format!("mean: {:.3}", mean))
156     .legend(|(x, y)| PathElement::new(vec![(x, y), (x + 20, y)], &BLUE));
157
158     chart
159         .configure_series_labels()
160         .label_font(("Hack", 14).into_font())
161         .background_style(&WHITE)
162         .border_style(&BLACK)
163         .draw()?;
164
165     root.present()?;
166     Ok(())
167 }
168
169 pub fn draw_acc_2hist(
170     datas: [&Vec<f64>; 2],
171     title: &str,
172     axes_desc: (&str, &str),
173     path: String,
174 ) -> Result<(), Box<dyn Error>> {
175     let n = datas.iter().fold(0f64, |max, l| max.max(l.len() as f64));
176     let mean: Vec<f64> = datas
177         .iter()
178         .map(|l| {
179             l.iter()
180                 .fold(0f64, |mean, &val| mean + val / l.len() as f64)
181         })
182         .collect();
183
184     let root = BitMapBackend::new(&path, (1024, 768)).into_drawing_area();
185     root.fill(&WHITE)?;
186
187     let mut chart = ChartBuilder::on(&root)
188         .caption(title, (FONT, CAPTION, FontStyle::Bold).into_font())
189         .margin(20)
190         .x_label_area_size(70)
191         .y_label_area_size(90)
192         .build_cartesian_2d((1..n as usize).into_segmented(), 0.0..1.0)?
193         .set_secondary_coord(0.0..n, 0.0..1.0);
194
195     chart
196         .configure_mesh()
197         .disable_x_mesh()
198         .y_max_light_lines(0)
199         .y_desc(axes_desc.1)
200         .x_desc(axes_desc.0)
201         .axis_desc_style((FONT, AXIS_LABEL))
202         .y_labels(3)
203         .label_style((FONT, AXIS_LABEL - 10))
204         .draw()?;
205
206     let a = datas[0].iter().zip(0..).map(|(y, x)| {
207         Rectangle::new(
208             [(x as f64 + 0.1, *y), (x as f64 + 0.5, 0f64)],
209             Into::<ShapeStyle>::into(&RED.mix(0.5)).filled(),
210         )
211     });
212
213     let b = datas[1].iter().zip(0..).map(|(y, x)| {
214         Rectangle::new(
215             [(x as f64 + 0.5, *y), (x as f64 + 0.9, 0f64)],
216             Into::<ShapeStyle>::into(&BLUE.mix(0.5)).filled(),
217         )
218     });
219
220     chart.draw_secondary_series(a)?;
221     chart.draw_secondary_series(b)?;
222
223     let v: Vec<usize> = (0..(n + 1.0) as usize).collect();
224     chart
225         .draw_secondary_series(Lineseries::new(
226             v.iter().map(|i| (*i as f64, mean[0])),
227             RED.filled().stroke_width(2),
228         ))?

```

```

229     .label(format!("mean: {:.3}", mean[0]))
230     .legend(|(x, y)| PathElement::new(vec![(x, y), (x + 20, y)], &RED));
231
232 chart
233     .draw_secondary_series(LineSeries::new(
234         v.iter().map(|i| (*i as f64, mean[1])),
235         BLUE.filled().stroke_width(2),
236     ))?
237     .label(format!("mean: {:.3}", mean[1]))
238     .legend(|(x, y)| PathElement::new(vec![(x, y), (x + 20, y)], &BLUE));
239
240 chart
241     .configure_series_labels()
242     .label_font((FONT, SERIE_LABEL).into_font())
243     .background_style(&WHITE)
244     .border_style(&BLACK)
245     .draw()?;
246
247 root.present()?;
248 Ok(())
249 }
250
251 /// Draw confusion matrix
252 pub fn draw_confustion(matrix_vec: Vec<[[i32; 2]; 2]>, path: String) -> Result<(), Box<dyn Error>> {
253     let root = BitMapBackend::new(&path, (2000, 1100)).into_drawing_area();
254     root.fill(&WHITE)?;
255
256     let (top, down) = root.split_vertically(1000);
257
258     let mut chart = ChartBuilder::on(&down)
259         .margin(20)
260         .margin_left(40)
261         .margin_right(40)
262         .x_label_area_size(40)
263         .build_cartesian_2d(0i32..50i32, 0i32..1i32)?;
264     chart
265         .configure_mesh()
266         .disable_y_axis()
267         .disable_y_mesh()
268         .x_labels(3)
269         .label_style((FONT, 40))
270         .draw()?;
271
272     chart.draw_series((0..50).map(|x| {
273         Rectangle::new(
274             [(x, 0), (x + 1, 1)],
275             HSLColor(
276                 240.0 / 360.0 - 240.0 / 360.0 * (x as f64 / 50.0),
277                 0.7,
278                 0.1 + 0.4 * x as f64 / 50.0,
279             )
280             .filled(),
281         )
282     }))?;
283     // hardcode for 10 iterations
284     let drawing_areas = top.split_evenly((2, 5));
285     let mut matrix_iter = matrix_vec.iter();
286     for (drawing_area, idx) in drawing_areas.iter().zip(1..) {
287         if let Some(matrix) = matrix_iter.next() {
288             let mut chart = ChartBuilder::on(&drawing_area)
289                 .caption(
290                     format!("Iteration {}", idx),
291                     (FONT, 40, FontStyle::Bold).into_font(),
292                 )
293                 .margin(20)
294                 .build_cartesian_2d(0i32..2i32, 2i32..0i32)?
295                 .set_secondary_coord(0f64..2f64, 2f64..0f64);
296
297             chart
298                 .configure_mesh()
299                 .disable_axes()
300                 .max_light_lines(4)
301                 .disable_x_mesh()
302                 .disable_y_mesh()
303                 .label_style(("Hack", 20))
304                 .draw()?;

```

```

305
306     chart.draw_series(
307         matrix
308             .iter()
309             .zip(0..)
310             .map(|(l, y)| 1.iter().zip(0..).map(move |(v, x)| (x, y, v)))
311             .flatten()
312             .map(|(x, y, v)| {
313                 Rectangle::new(
314                     [(x, y), (x + 1, y + 1)],
315                     HSLColor(
316                         240.0 / 360.0 - 240.0 / 360.0 * (*v as f64 / 50.0),
317                         0.7,
318                         0.1 + 0.4 * *v as f64 / 50.0,
319                     )
320                     .filled(),
321                 )
322             }),
323     )?;
324
325     chart.draw_secondary_series(
326         matrix
327             .iter()
328             .zip(0..)
329             .map(|(l, y)| 1.iter().zip(0..).map(move |(v, x)| (x, y, v)))
330             .flatten()
331             .map(|(x, y, v)| {
332                 let text: String = if x == 0 && y == 0 {
333                     format!("TP:{}", v)
334                 } else if x == 1 && y == 0 {
335                     format!("FP:{}", v)
336                 } else if x == 0 && y == 1 {
337                     format!("FN:{}", v)
338                 } else {
339                     format!("TN:{}", v)
340                 };
341
342                 Text::new(
343                     text,
344                     ((2.0 * x as f64 + 0.7) / 2.0, (2.0 * y as f64 + 1.0) / 2.0),
345                     FONT.into_font().resize(30.0).color(&WHITE),
346                 )
347             }),
348     )?;
349 }
350
351 root.present()?;
352 Ok(())
353 }
354
355 /// Receive each cross-validation vector of each individual fitness value.
356 pub fn draw_ga_progress(
357     cv_fitness: &Vec<Vec<i32, f64>>,
358     path: String,
359 ) -> Result<(), Box<dyn Error>> {
360     let root = BitMapBackend::new(&path, (2000, 1000)).into_drawing_area();
361     root.fill(&WHITE)?;
362
363     // This is mostly hardcoded
364     let drawing_areas = root.split_evenly((2, 5));
365     for ((drawing_area, idx), fitness) in drawing_areas.iter().zip(1..).zip(cv_fitness.iter()) {
366         let mut chart = ChartBuilder::on(&drawing_area)
367             .caption(
368                 format!("Iteration {}", idx),
369                 (FONT, 40, FontStyle::Bold).into_font(),
370             )
371             .margin(40)
372             .x_label_area_size(20)
373             .y_label_area_size(20)
374             .build_cartesian_2d(0i32..200i32, 0.0..1.1)?;
375
376         chart
377             .configure_mesh()
378             .x_labels(3)
379             .y_labels(2)
380             .label_style((FONT, 30))

```

```

381         .max_light_lines(4)
382         .draw()?;
383
384     chart.draw_series(
385         fitness
386         .iter()
387         .map(|x| Circle::new((x.0, x.1), 1, BLUE.mix(0.5).filled())),
388     )?;
389 }
390 root.present()?;
391 Ok(())
392 }
393

```

Source Code 10: utils/io.rs

```

1  use crate::activator;
2  use crate::mlp;
3  use serde_json::{json, to_writer_pretty, Value};
4  use std::error::Error;
5  use std::fs::create_dir;
6  use std::fs::File;
7  use std::io::Read;
8  use std::io::{self, BufRead};
9  use std::path::Path;
10
11 pub fn save(layers: &Vec<mlp::Layer>, path: String) -> Result<(), Box<dyn Error>> {
12     let mut json: Vec<Value> = vec![];
13
14     for l in layers {
15         json.push(json!({
16             "inputs": l.inputs.len(),
17             "outputs": l.outputs.len(),
18             "w": l.w,
19             "b": l.b,
20             "act": l.act.name
21         }));
22     }
23     let result = json!(json);
24     let file = File::create(path)?;
25     to_writer_pretty(&file, &result)?;
26     Ok(())
27 }
28
29 pub fn read_lines<P>(filename: P) -> io::Result<io::Lines<io::BufReader<File>>>
30 where
31     P: AsRef<Path>,
32 {
33     let file = File::open(filename)?;
34     Ok(io::BufReader::new(file).lines())
35 }
36
37 pub fn read_file<P>(filename: P) -> Result<String, Box<dyn Error>>
38 where
39     P: AsRef<Path>,
40 {
41     let mut file = File::open(filename)?;
42     let mut contents = String::new();
43     file.read_to_string(&mut contents)?;
44     Ok(contents)
45 }
46
47 pub fn load<P>(filename: P) -> Result<mlp::Net, Box<dyn Error>>
48 where
49     P: AsRef<Path>,
50 {
51     let contents = read_file(filename)?;
52
53     let json: Value = serde_json::from_str(&contents)?;
54     let mut layers: Vec<mlp::Layer> = vec![];
55
56     for l in json.as_array().unwrap() {
57         // default layer activation is simple linear f(x) = x
58         let mut layer = mlp::Layer::new(
59             l["inputs"].as_u64().unwrap(),

```

```

60         l["outputs"].as_u64().unwrap(),
61         0.0,
62         activator::linear(),
63     );
64     // setting activation function
65     if l["act"] == "sigmoid" {
66         layer.act = activator::sigmoid();
67     }
68
69     // setting weights and bias
70     let w = l["w"].as_array().unwrap();
71     let b = l["b"].as_array().unwrap();
72     for j in 0..w.len() {
73         layer.b[j] = b[j].as_f64().unwrap();
74         let w_j = w[j].as_array().unwrap();
75         for i in 0..w_j.len() {
76             layer.w[j][i] = w_j[i].as_f64().unwrap();
77         }
78     }
79
80     layers.push(layer);
81 }
82
83 Ok(mlp::Net::from_layers(layers))
84 }
85
86 /// Check if specify folder exists in models and img folder, if not create it
87 ///
88 /// Return models path and img path
89 pub fn check_dir(folder: &str) -> Result<(String, String), Box<dyn Error>> {
90     let models_path = format!("models/{}", folder);
91     if !Path::new(&models_path).exists() {
92         create_dir(&models_path)?;
93     }
94     let img_path = format!("report/images/{}", folder);
95     if !Path::new(&img_path).exists() {
96         create_dir(&img_path)?;
97     }
98     Ok((models_path, img_path))
99 }
100

```