# 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 traning 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 effictive 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 (w1, w2, w3, b1, w4, w5, w6, b2, w7, w8, b3).

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.

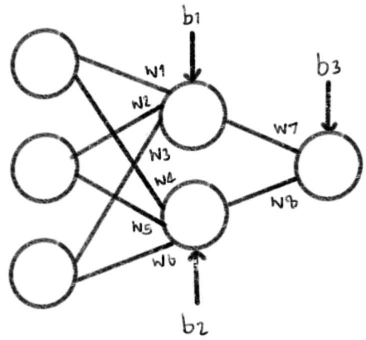



Figure 1: The 3-2-1 network.

### 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.

$$f(i) = \text{accuracy}_i + \frac{0.001}{\text{MSE}_i} \tag{1}$$

### 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, them perforrm 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 chrosome has an equal chance to be from dad or mom (implementation on 2). We will perform crossover untill 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

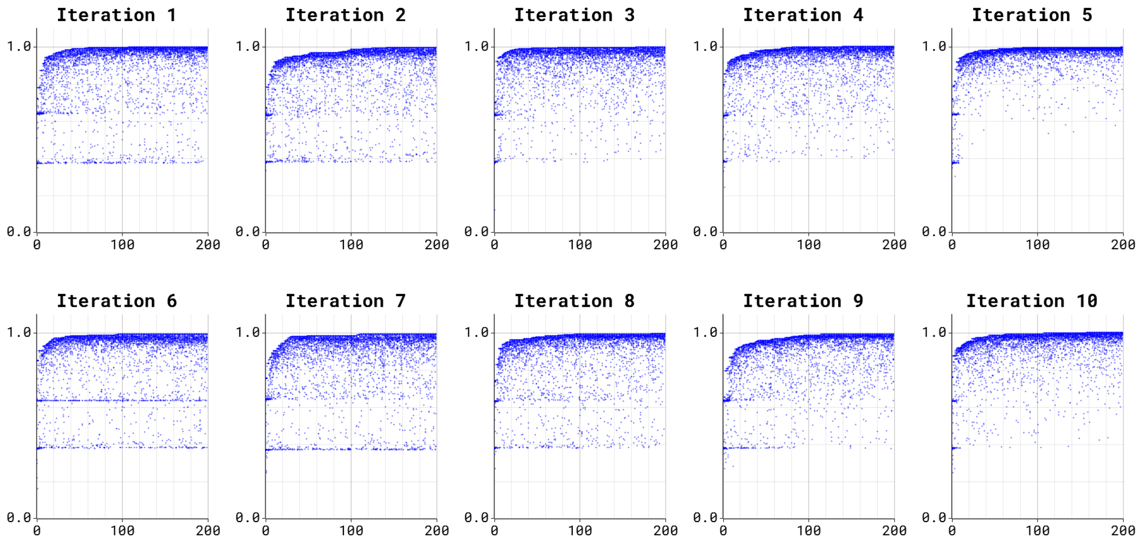
Using 10% cross-validation, and only preprocess each iteration training and validation set with min-max normalization to avoid data leakage as state on [Bro]. The min-max normalization process is done by for each feature $F$ on training set we find $max(F)$ and $min(F)$ then for each datapoint $F_x$ 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 untill 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.

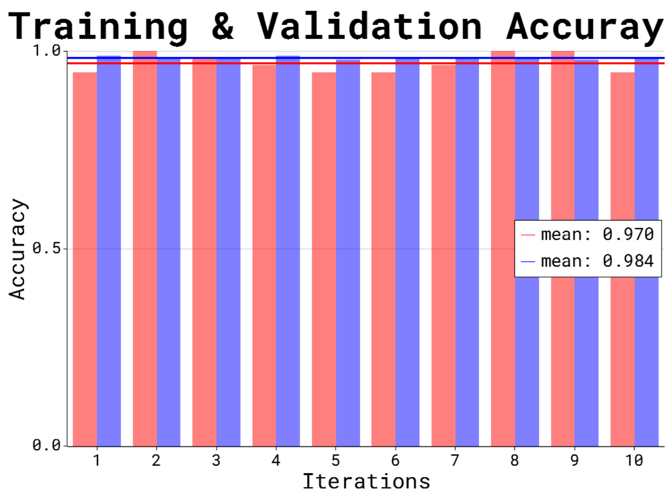
## Training Result

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).

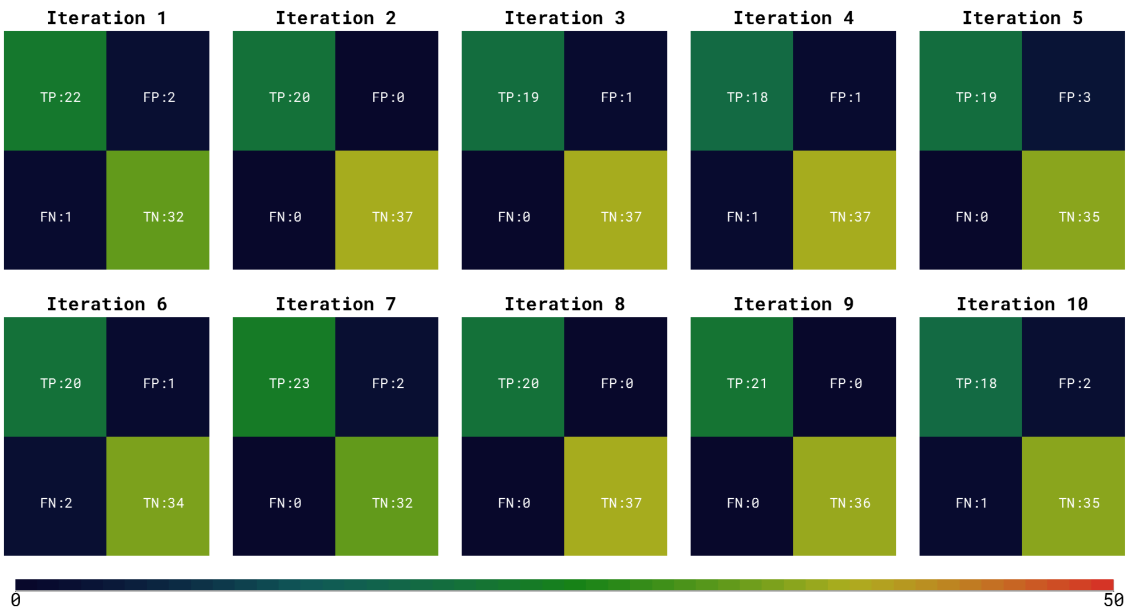
- **wdbc-30-15-1** : The base model that contains 30 input nodes, 1 hidden layer with 15 nodes, and 1 output node with all nodes using sigmoid as an activation function. We assume that this model will have accuracy > 95% with reasonable training time used. The result is shown on fig. 2.
- **wdbc-30-7-1** : 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-valiation 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.

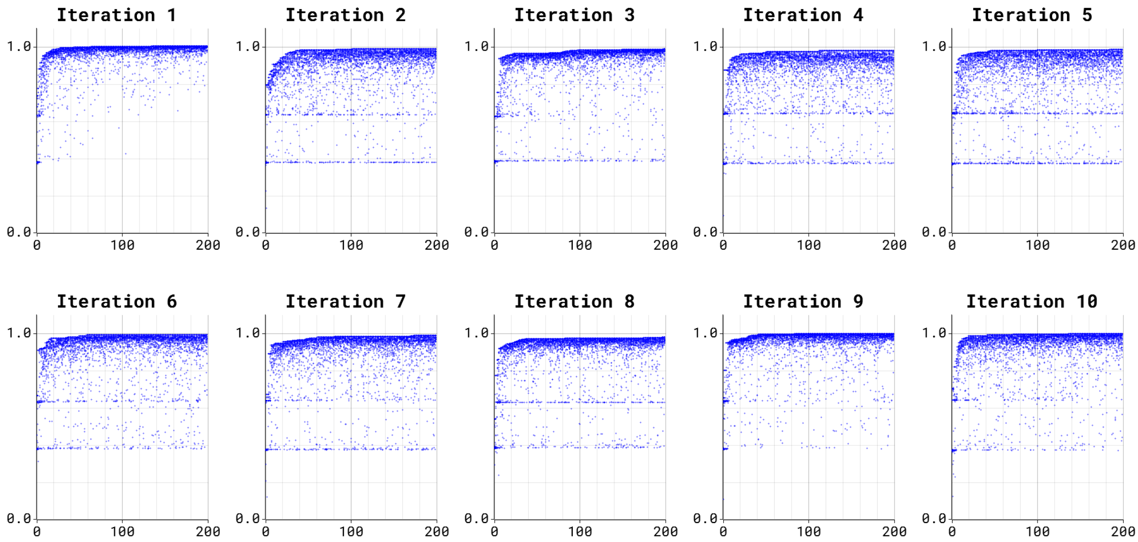



(b) The best individual from each cross-validation iteration accuracy on training set (blue) and validation set (red).

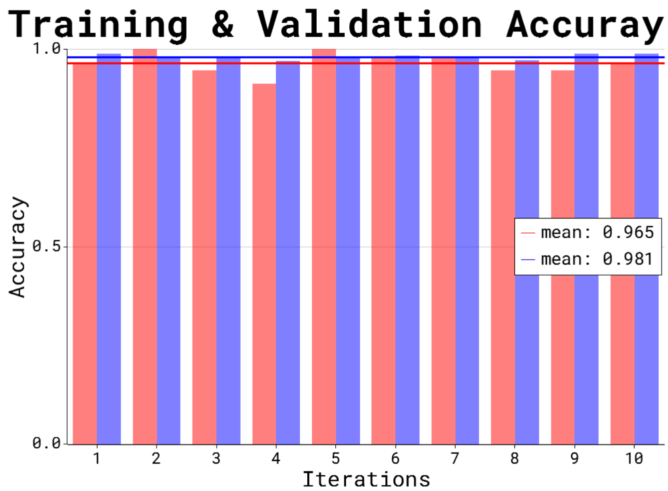



(c) The best individual from each cross-valiation iteration confusion matrix on validation set.

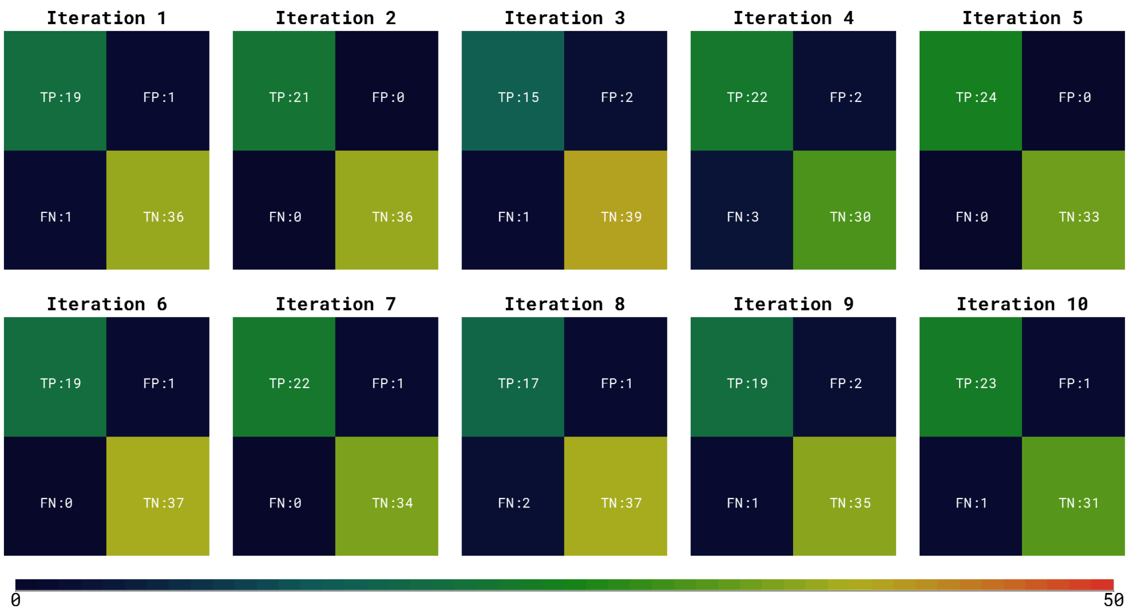
Figure 2: Training result of wdbc-30-15-1 with 20.609 seconds used for training.




(a) The training process of each cross-valiation 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.

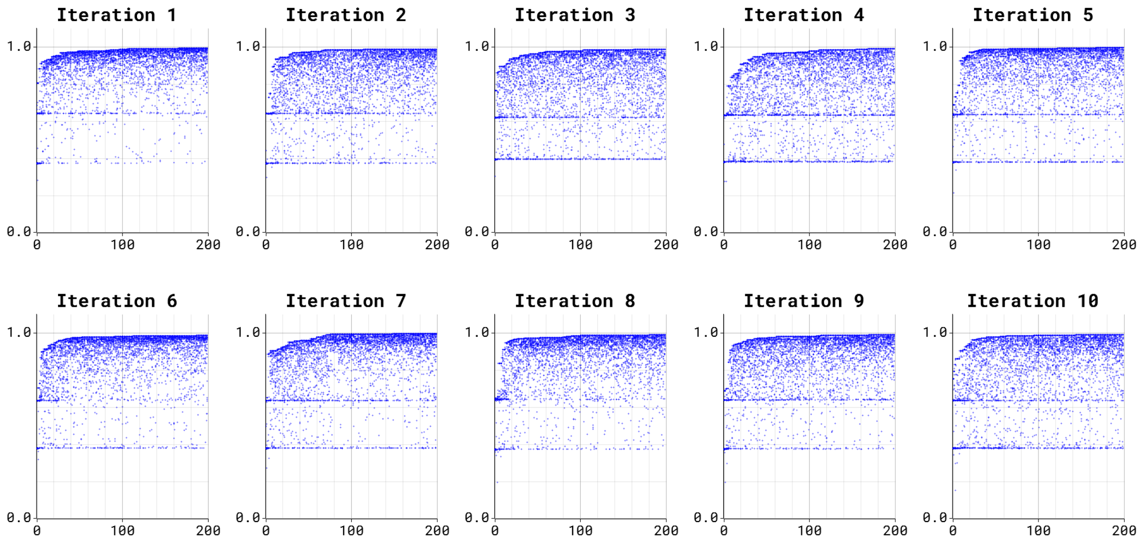



(b) The best individual from each cross-validation iteration accuracy on training set (blue) and validation set (red).

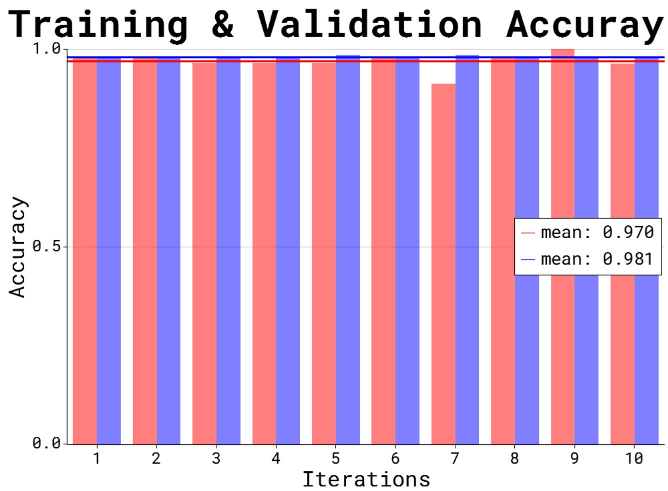



(c) The best individual from each cross-valiation iteration confusion matrix on validation set.

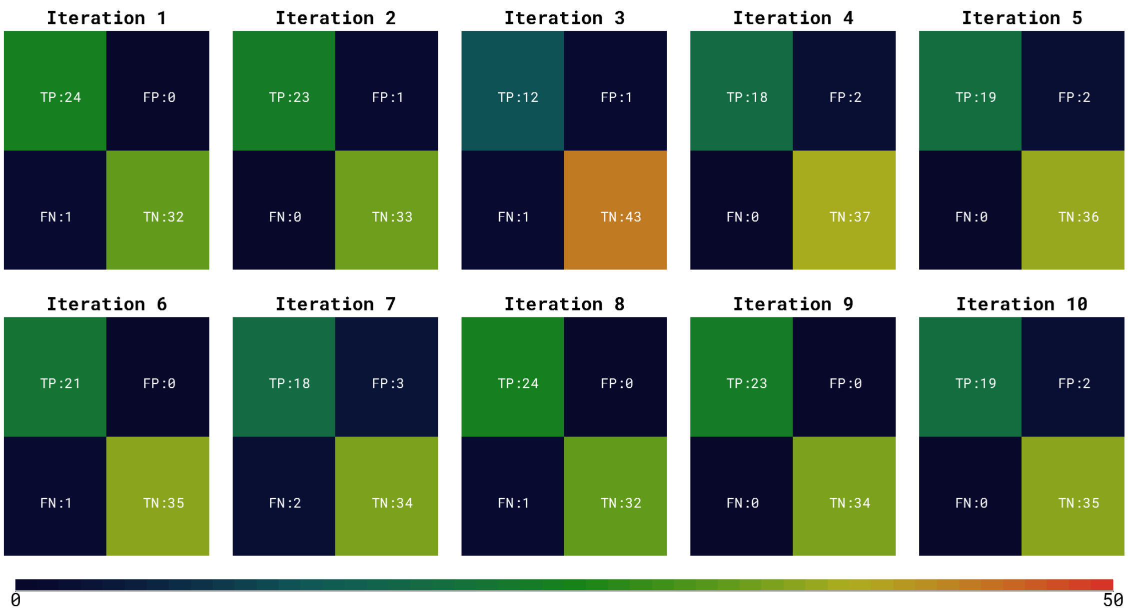
Figure 3: Training result of wdbc-30-7-1 with 14.163 seconds used for training.




(a) The training process of each cross-valiation 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-valiation 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 we 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 Auephanwiriyakul. *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

```rust
pub mod activator;
pub mod ga;
pub mod loss;
pub mod mlp;
pub mod models;
pub mod utills;

use std::error::Error;
fn main() -> Result<(), Box<dyn Error>> {
    models::wdbc::wdbc_30_15_1();
    models::wdbc::wdbc_30_7_1();
    models::wdbc::wdbc_30_15_7_1();
    Ok(())
}
```

Source Code 2: ga/mod.rs

```rust
//! Genictic Algorithm Utility
pub mod selection;
use rand::{distributions::Uniform, prelude::Distribution, seq::SliceRandom, Rng};
use std::f64::consts::E;

use crate::mlp::Net;

#[derive(Clone)]
pub struct Individual {
    pub chromosome: Vec<f64>,
    pub fitness: f64,
}

impl Individual {
    pub fn new(chromosome: Vec<f64>) -> Individual {
        Individual {
            chromosome,
            fitness: 0.0,
        }
    }

    pub fn set_fitness(&mut self, v: f64) {
        self.fitness = v;
    }
}

/// return result of mating of individual in the pool
pub fn mating(pop: &Vec<Individual>) -> Vec<Individual> {
    let mut rand = rand::thread_rng();
    let new_pop: Vec<Individual> = pop
        .iter()
        .map(|_| {
            let parent: Vec<_> = pop.choose_multiple(&mut rand::thread_rng(), 2).collect();
            let new_chromosome: Vec<f64> = parent[0]
                .chromosome
                .iter()
                .zip(parent[1].chromosome.iter())
                .map(|(p0, p1)| if rand.gen_bool(0.5) { *p0 } else { *p1 })
                .collect();
            Individual::new(new_chromosome)
        })
        .collect();
    new_pop
}

/// strong mutation
pub fn mutate(pop: &Vec<Individual>, amount: usize, p_m: f64) -> Vec<Individual> {
    let mut rand = rand::thread_rng();
    let new_pop: Vec<Individual> = pop
        .choose_multiple(&mut rand::thread_rng(), amount)
        .into_iter()
        .map(|ind| {
            let mut ind_clone = ind.clone();
```

```
            for gene in ind_clone.chromosome.iter_mut() {
                let between = Uniform::from(0.0..=1.0);
                if between.sample(&mut rand) < p_m {
                    let change = 2f64 * rand::random::<f64>() - 1f64;
                    *gene += change;
                }
            }
            ind_clone
        })
        .collect();
    new_pop
}

/// non-uniform strong mutation
pub fn mutate_nonuni(
    pop: &Vec<Individual>,
    amount: usize,
    p_m: f64,
    curr_gen: usize,
) -> Vec<Individual> {
    let mut new_pop: Vec<Individual> = vec![];
    let mut rand = rand::thread_rng();
    let beta = 1.0;
    for i in 0..amount {
        let mut ind_clone = pop[i].clone();
        for j in 0..pop[i].chromosome.len() {
            let between = Uniform::from(0.0..=1.0);
            if between.sample(&mut rand) < (p_m * E.powf(-beta * curr_gen as f64)) {
                let change = 2f64 * rand::random::<f64>() - 1f64;
                ind_clone.chromosome[j] += change;
            }
        }
        new_pop.push(ind_clone);
    }
    new_pop
}

/// Create inital population of MLP from layers
///
/// return: population
pub fn init_pop(net: &Net, amount: u32) -> Vec<Individual> {
    let mut pop: Vec<Individual> = vec![];
    for _ in 0..(amount) {
        let mut chromosome: Vec<f64> = vec![];
        for l in &net.layers {
            for output in &l.w {
                for _ in output {
                    // new random weight in range [-1, 1]
                    chromosome.push(2f64 * rand::random::<f64>() - 1f64);
                }
            }
            for bias in &l.b {
                chromosome.push(*bias);
            }
        }
        pop.push(Individual::new(chromosome));
    }
    pop
}

/// assign individual weigth to net
pub fn assign_ind(net: &mut Net, individual: &Individual) {
    if net.parameters != individual.chromosome.len() as u64 {
        panic!["The neural network parameters size is not equal to individual size"];
    }
    let mut idx: usize = 0;

    for l in &mut net.layers {
        l.w.iter_mut().for_each(|w_j| {
            w_j.iter_mut().for_each(|w_ji| {
                *w_ji = individual.chromosome[idx];
                idx += 1;
            })
        });

        l.b.iter_mut().for_each(|b_i| {
```

```rust
            *b_i = individual.chromosome[idx];
            idx += 1;
        });
    }
}

#[cfg(test)]
mod tests {
    use super::*;
    use crate::{
        activator,
        mlp::{self, Layer},
    };

    #[test]
    fn test_init_pop() {
        let mut layers: Vec<mlp::Layer> = vec![];
        layers.push(Layer::new(4, 2, 1.0, activator::sigmoid()));
        layers.push(Layer::new(2, 1, 1.0, activator::sigmoid()));
        let net = Net::from_layers(layers);
        let pop = init_pop(&net, 5);

        assert_eq!(pop.len(), 5);
        assert_eq!(pop[0].chromosome.len() as u64, net.parameters);
        // check if bias is the same.
        assert_eq!(pop[0].chromosome[8], 1.0);
        assert_eq!(pop[0].chromosome[9], 1.0);
        assert_eq!(pop[0].chromosome[12], 1.0);
    }

    #[test]
    fn test_assign_ind() {
        let mut layers: Vec<mlp::Layer> = vec![];
        layers.push(Layer::new(3, 1, 1.0, activator::sigmoid()));
        layers.push(Layer::new(1, 1, 1.0, activator::sigmoid()));
        let mut net = Net::from_layers(layers);

        let individual = Individual::new(vec![2.5, 2.3, 2.1, 1.2, 1.3, 4.0]);
        assign_ind(&mut net, &individual);

        // check if network has been mutated correctly or not.
        let mut idx = 0;
        for l in net.layers {
            for output in l.w {
                for w in output {
                    assert_eq!(w, individual.chromosome[idx]);
                    idx += 1;
                }
            }
            for b in l.b {
                assert_eq!(b, individual.chromosome[idx]);
                idx += 1;
            }
        }
    }

    #[test]
    fn test_mating_and_mutate() {
        let mut pop: Vec<Individual> = vec![];
        for i in 0..4 {
            let v = i as f64 + 1.0;
            pop.push(Individual::new(vec![v, v, v, 1.0]))
        }

        let res = mating(&pop);
        let mut_res = mutate(&pop, 4, 0.5);
        assert_eq!(res.len(), pop.len());
        assert_eq!(mut_res.len(), pop.len());

        for ind in res {
            println!("{:?}", ind.chromosome);
        }
        for ind in mut_res {
            println!("{:?}", ind.chromosome);
        }
    }
```

```
}

```

Source Code 3: ga/selection.rs

```
use rand::seq::SliceRandom;
use super::Individual;

/// binary deterministic tournament with reinsertion
pub fn d_tornament(pop: &Vec<Individual>) -> Vec<Individual> {
    let mut results: Vec<Individual> = vec![];
    for _ in 0..pop.len() {
        let players: Vec<_> = pop.choose_multiple(&mut rand::thread_rng(), 2).collect();

        if players[0].fitness > players[1].fitness {
            results.push(players[0].clone());
        } else {
            results.push(players[1].clone());
        }
    }
    results
}

```

Source Code 4: models/wdbc.rs

```
use std::{error::Error, time::Instant};

use crate::{
    activator,
    ga::{self, Individual},
    loss,
    mlp::{self, Layer, Net},
    utills::{
        data::{self, confusion_count},
        graph, io,
    },
};

const IMGPATH: &str = "report/assignment_3/images";

pub fn wdbc_30_15_1() {
    fn model() -> Net {
        let mut layers: Vec<mlp::Layer> = vec![];
        layers.push(Layer::new(30, 15, 1.0, activator::sigmoid()));
        layers.push(Layer::new(15, 1, 1.0, activator::sigmoid()));
        Net::from_layers(layers)
    }
    wdbc_ga(&model, "wdbc-30-15-1", IMGPATH).unwrap();
}

pub fn wdbc_30_7_1() {
    fn model() -> Net {
        let mut layers: Vec<mlp::Layer> = vec![];
        layers.push(Layer::new(30, 7, 1.0, activator::sigmoid()));
        layers.push(Layer::new(7, 1, 1.0, activator::sigmoid()));
        Net::from_layers(layers)
    }
    wdbc_ga(&model, "wdbc-30-7-1", IMGPATH).unwrap();
}

pub fn wdbc_30_15_7_1() {
    fn model() -> Net {
        let mut layers: Vec<mlp::Layer> = vec![];
        layers.push(Layer::new(30, 15, 1.0, activator::sigmoid()));
        layers.push(Layer::new(15, 7, 1.0, activator::sigmoid()));
        layers.push(Layer::new(7, 1, 1.0, activator::sigmoid()));
        Net::from_layers(layers)
    }
    wdbc_ga(&model, "wdbc-30-15-7-1", IMGPATH).unwrap();
}

/// train mlp with genitic algorithm
pub fn wdbc_ga(model: &dyn Fn() -> Net, folder: &str, imgpath: &str) -> Result<(), Box<dyn Error>> {
```

```
    let dataset = data::wdbc_dataset()?;
    let mut valid_acc: Vec<f64> = vec![];
    let mut train_acc: Vec<f64> = vec![];
    let mut train_proc: Vec<Vec<(i32, f64)>> = Vec::with_capacity(10);
    for _ in 0..10 {
        train_proc.push(vec![]);
    }

    let mut matrix_vec: Vec<[[i32; 2]; 2]> = vec![];
    let threshold = 0.5;
    let max_gen = 200;

    let start = Instant::now();
    for (j, dt) in dataset.cross_valid_set(0.1).iter().enumerate() {
        let mut net = model();
        let (training_set, validation_set) = dt.0.minmax_norm(&dt.1);
        let mut loss = loss::Loss::square_err();

        // training with GA
        let mut pop = ga::init_pop(&net, 25);
        let mut best_ind = pop[0].clone();

        for k in 0..max_gen {
            let mut max_fitness = f64::MIN;
            let mut local_best_ind = pop[0].clone();

            for p in pop.iter_mut() {
                ga::assign_ind(&mut net, &p);
                let mut matrix = [[0, 0], [0, 0]];
                let mut run_loss = 0.0;
                for data in training_set.get_shuffled() {
                    let result = net.forward(&data.inputs);
                    run_loss += loss.criterion(&result, &data.labels);
                    confusion_count(&mut matrix, &result, &data.labels, threshold);
                }
                let fitness = ((matrix[0][0] + matrix[1][1]) as f64 / training_set.len() as f64)
                    + 0.001 / (run_loss / training_set.len() as f64);
                p.set_fitness(fitness);
                train_proc[j].push((k, fitness)); // track training progress

                if fitness > max_fitness {
                    max_fitness = fitness;
                    local_best_ind = p.clone();
                }
                // store best individual for all generation
                if best_ind.fitness < fitness {
                    best_ind = p.clone();
                }
            }

            // selection
            let p1 = ga::selection::d_tornament(&pop);
            let mating_result = ga::mating(&p1);
            let mut mut_result = ga::mutate(&mating_result, 20, 0.02);

            let mut new_pop: Vec<Individual> = vec![];
            new_pop.append(&mut mut_result);
            let pop_need = pop.len() - new_pop.len();

            // elitsm
            for _ in 0..pop_need {
                new_pop.push(local_best_ind.clone());
            }

            pop = new_pop;
            println!("[{}, {}] max_fitness: {:.3}", j, k, max_fitness);
        }

        ga::assign_ind(&mut net, &best_ind);
        let mut matrix = [[0, 0], [0, 0]];
        for data in validation_set.get_datas() {
            let result = net.forward(&data.inputs);
            confusion_count(&mut matrix, &result, &data.labels, threshold);
        }
        valid_acc.push((matrix[0][0] + matrix[1][1]) as f64 / validation_set.len() as f64);
        matrix_vec.push(matrix);
```

```rust
        let mut matrix_t = [[0, 0], [0, 0]];
        for data in training_set.get_datas() {
            let result = net.forward(&data.inputs);
            confusion_count(&mut matrix_t, &result, &data.labels, threshold);
        }
        train_acc.push((matrix_t[0][0] + matrix_t[1][1]) as f64 / training_set.len() as f64);
        //io::save(&net.layers, format!("models/{}/{}.json", folder, j))?;
    }
    let duration = start.elapsed();
    println!("Time used: {:.3} sec", duration.as_secs_f32());

    graph::draw_acc_2hist(
        [&valid_acc, &train_acc],
        "Training & Validation Accuray",
        ("Iterations", "Accuracy"),
        format!("{}/{}/accuracy.png", imgpath, folder),
    )?;
    graph::draw_confustion(matrix_vec, format!("{}/{}/conf_mat.png", imgpath, folder))?;
    graph::draw_ga_progress(
        &train_proc,
        format!("{}/{}/train_proc.png", imgpath, folder),
    )?;

    Ok(())
}
```

Source Code 5: mlp.rs

```rust
use crate::activator;

#[derive(Debug)]
pub struct Layer {
    pub inputs: Vec<f64>,
    pub outputs: Vec<f64>, // need to save this for backward pass
    pub w: Vec<Vec<f64>>,
    pub b: Vec<f64>,
    pub grads: Vec<Vec<f64>>,
    pub w_prev_changes: Vec<Vec<f64>>,
    pub local_grads: Vec<f64>,
    pub b_prev_changes: Vec<f64>,
    pub act: activator::ActivationContainer,
}

impl Layer {
    pub fn new(
        input_features: u64,
        output_features: u64,
        bias: f64,
        act: activator::ActivationContainer,
    ) -> Layer {
        // initialize weights matrix
        let mut weights: Vec<Vec<f64>> = vec![];
        let mut inputs: Vec<f64> = vec![];
        let mut outputs: Vec<f64> = vec![];
        let mut grads: Vec<Vec<f64>> = vec![];
        let mut local_grads: Vec<f64> = vec![];
        let mut w_prev_changes: Vec<Vec<f64>> = vec![];
        let mut b_prev_changes: Vec<f64> = vec![];
        let mut b: Vec<f64> = vec![];

        for _ in 0..output_features {
            outputs.push(0.0);
            local_grads.push(0.0);
            b_prev_changes.push(0.0);
            b.push(bias);

            let mut w: Vec<f64> = vec![];
            let mut g: Vec<f64> = vec![];
            for _ in 0..input_features {
                if (inputs.len() as u64) < input_features {
                    inputs.push(0.0);
                }
                g.push(0.0);
                // random both positive and negative weight
                w.push(2f64 * rand::random::<f64>() - 1f64);
```

```
            }
            weights.push(w);
            grads.push(g.clone());
            w_prev_changes.push(g);
        }
        Layer {
            inputs,
            outputs,
            w: weights,
            b,
            grads,
            w_prev_changes,
            local_grads,
            b_prev_changes,
            act,
        }
    }

    pub fn forward(&mut self, inputs: &Vec<f64>) -> Vec<f64> {
        if inputs.len() != self.inputs.len() {
            panic!("forward: input size is wrong");
        }

        let result: Vec<f64> = self
            .w
            .iter()
            .zip(self.b.iter())
            .zip(self.outputs.iter_mut())
            .map(|((w_j, b_j), o_j)| {
                let sum = inputs
                    .iter()
                    .zip(w_j.iter())
                    .fold(0.0, |s, (v, w_ji)| s + w_ji * v)
                    + b_j;
                *o_j = sum;
                (self.act.func)(sum)
            })
            .collect();

        self.inputs = inputs.clone();
        result
    }

    pub fn update(&mut self, lr: f64, momentum: f64) {
        for j in 0..self.w.len() {
            let delta_b = lr * self.local_grads[j] + momentum * self.b_prev_changes[j];
            self.b[j] -= delta_b; // update each neuron bias
            self.b_prev_changes[j] = delta_b;
            for i in 0..self.w[j].len() {
                // update each weights
                let delta_w = lr * self.grads[j][i] + momentum * self.w_prev_changes[j][i];
                self.w[j][i] -= delta_w;
                self.w_prev_changes[j][i] = delta_w;
            }
        }
    }

    pub fn zero_grad(&mut self) {
        for j in 0..self.outputs.len() {
            self.local_grads[j] = 0.0;
            for i in 0..self.grads[j].len() {
                self.grads[j][i] = 0.0;
            }
        }
    }
}

#[derive(Debug)]
pub struct Net {
    pub layers: Vec<Layer>,
    pub parameters: u64,
}

impl Net {
    pub fn from_layers(layers: Vec<Layer>) -> Net {
        let mut parameters: u64 = 0;
```

```
        for l in &layers {
            parameters += (l.w.len() * l.w[0].len()) as u64;
            parameters += l.b.len() as u64;
        }

        Net { layers, parameters }
    }

    pub fn new(architecture: Vec<u64>) -> Net {
        let mut layers: Vec<Layer> = vec![];
        for i in 1..architecture.len() {
            layers.push(Layer::new(
                architecture[i - 1],
                architecture[i],
                1f64,
                activator::sigmoid(),
            ))
        }
        Net::from_layers(layers)
    }

    pub fn zero_grad(&mut self) {
        for l in 0..self.layers.len() {
            self.layers[l].zero_grad();
        }
    }

    pub fn forward(&mut self, input: &Vec<f64>) -> Vec<f64> {
        let mut result = self.layers[0].forward(input);
        for l in 1..self.layers.len() {
            result = self.layers[l].forward(&result);
        }
        result
    }

    pub fn update(&mut self, lr: f64, momentum: f64) {
        for l in 0..self.layers.len() {
            self.layers[l].update(lr, momentum);
        }
    }
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn test_linear_new() {
        let linear = Layer::new(2, 3, 1.0, activator::linear());
        assert_eq!(linear.outputs.len(), 3);
        assert_eq!(linear.inputs.len(), 2);

        assert_eq!(linear.w.len(), 3);
        assert_eq!(linear.w[0].len(), 2);
        assert_eq!(linear.b.len(), 3);

        assert_eq!(linear.grads.len(), 3);
        assert_eq!(linear.w_prev_changes.len(), 3);
        assert_eq!(linear.grads[0].len(), 2);
        assert_eq!(linear.w_prev_changes[0].len(), 2);
        assert_eq!(linear.local_grads.len(), 3);
        assert_eq!(linear.b_prev_changes.len(), 3);
    }

    #[test]
    fn test_linear_forward1() {
        let mut linear = Layer::new(2, 1, 1.0, activator::sigmoid());

        for j in 0..linear.w.len() {
            for i in 0..linear.w[j].len() {
                linear.w[j][i] = 1.0;
            }
        }

        assert_eq!(linear.forward(&vec![1.0, 1.0])[0], 0.9525741268224334);
        assert_eq!(linear.outputs[0], 3.0);
```

```
    }

    #[test]
    fn test_linear_forward2() {
        let mut linear = Layer::new(2, 2, 1.0, activator::sigmoid());

        for j in 0..linear.w.len() {
            for i in 0..linear.w[j].len() {
                linear.w[j][i] = (j as f64) + 1.0;
            }
        }
        let result = linear.forward(&vec![0.0, 1.0]);
        assert_eq!(linear.outputs[0], 2.0);
        assert_eq!(linear.outputs[1], 3.0);
        assert_eq!(result[0], 0.8807970779778823);
        assert_eq!(result[1], 0.9525741268224334);
    }
}
```

Source Code 6: activator.rs

```
#[derive(Debug)]
pub struct ActivationContainer {
    pub func: fn(f64) -> f64,
    pub der: fn(f64) -> f64,
    pub name: String,
}

pub fn sigmoid() -> ActivationContainer {
    fn func(input: f64) -> f64 {
        1.0 / (1.0 + (-input).exp())
    }
    fn der(input: f64) -> f64 {
        func(input) * (1.0 - func(input))
    }
    ActivationContainer {
        func,
        der,
        name: "sigmoid".to_string(),
    }
}

pub fn relu() -> ActivationContainer {
    fn func(input: f64) -> f64 {
        return f64::max(0.0, input);
    }
    fn der(input: f64) -> f64 {
        if input > 0.0 {
            return 1.0;
        } else {
            return 0.0;
        }
    }
    ActivationContainer {
        func,
        der,
        name: "relu".to_string(),
    }
}

pub fn linear() -> ActivationContainer {
    fn func(input: f64) -> f64 {
        input
    }
    fn der(_input: f64) -> f64 {
        1.0
    }
    ActivationContainer {
        func,
        der,
        name: "linear".to_string(),
    }
}

#[cfg(test)]
```

```rust
mod tests {
    use super::*;

    #[test]
    fn test_sigmoid() {
        let act = sigmoid();

        assert_eq!((act.func)(1.0), 0.7310585786300048792512);
        assert_eq!((act.func)(-1.0), 0.2689414213699951207488);
        assert_eq!((act.func)(0.0), 0.5);
        assert_eq!((act.der)(1.0), 0.1966119332414818525374);
        assert_eq!((act.der)(-1.0), 0.1966119332414818525374);
        assert_eq!((act.der)(0.0), 0.25);
    }

    #[test]
    fn test_relu() {
        let act = relu();

        assert_eq!((act.func)(-1.0), 0.0);
        assert_eq!((act.func)(20.0), 20.0);
        assert_eq!((act.der)(-1.0), 0.0);
        assert_eq!((act.der)(20.0), 1.0);
    }
}

```

Source Code 7: loss.rs

```rust
use crate::mlp;

pub struct Loss {
    outputs: Vec<f64>,
    desired: Vec<f64>,
    pub func: fn(f64, f64) -> f64,
    pub der: fn(f64, f64) -> f64,
}

impl Loss {
    /// Squared Error
    pub fn square_err() -> Loss {
        fn func(output: f64, desired: f64) -> f64 {
            0.5 * (output - desired).powi(2)
        }
        fn der(output: f64, desired: f64) -> f64 {
            output - desired
        }

        Loss {
            outputs: vec![],
            desired: vec![],
            func,
            der,
        }
    }

    /// Binary Cross Entropy
    pub fn bce() -> Loss {
        fn func(output: f64, desired: f64) -> f64 {
            -desired * output.ln() + (1.0 - desired) * (1.0 - output).ln()
        }
        fn der(output: f64, desired: f64) -> f64 {
            -(desired / output - (1.0 - desired) / (1.0 - output))
        }

        Loss {
            outputs: vec![],
            desired: vec![],
            func,
            der,
        }
    }

    pub fn criterion(&mut self, outputs: &Vec<f64>, desired: &Vec<f64>) -> f64 {
        if outputs.len() != desired.len() {
```

```
            panic!("outputs size is not equal to desired size");
        }
        let loss = outputs
            .iter()
            .zip(desired.iter())
            .fold(0.0, |ls, (o, d)| ls + (self.func)(*o, *d));
        self.outputs = outputs.clone();
        self.desired = desired.clone();
        loss
    }

    pub fn backward(&self, layers: &mut Vec<mlp::Layer>) {
        for l in (0..layers.len()).rev() {
            // output layer
            if l == layers.len() - 1 {
                for j in 0..layers[l].outputs.len() {
                    // compute grads
                    let local_grad = (self.der)(self.outputs[j], self.desired[j])
                        * (layers[l].act.der)(layers[l].outputs[j]);

                    layers[l].local_grads[j] = local_grad;

                    // set grads for each weight
                    for k in 0..(layers[l - 1].outputs.len()) {
                        layers[l].grads[j][k] =
                            (layers[l - 1].act.func)(layers[l - 1].outputs[k]) * local_grad;
                    }
                }
                continue;
            }
            // hidden layer
            for j in 0..layers[l].outputs.len() {
                // calculate local_grad based on previous local_grad
                let mut local_grad = 0f64;
                for i in 0..layers[l + 1].w.len() {
                    for k in 0..layers[l + 1].w[i].len() {
                        local_grad += layers[l + 1].w[i][k] * layers[l + 1].local_grads[i];
                    }
                }
                local_grad = (layers[l].act.der)(layers[l].outputs[j]) * local_grad;
                layers[l].local_grads[j] = local_grad;

                // set grads for each weight
                if l == 0 {
                    for k in 0..layers[l].inputs.len() {
                        layers[l].grads[j][k] = layers[l].inputs[k] * local_grad;
                    }
                } else {
                    for k in 0..layers[l - 1].outputs.len() {
                        layers[l].grads[j][k] =
                            (layers[l - 1].act.func)(layers[l - 1].outputs[k]) * local_grad;
                    }
                }
            }
        }
    }
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn test_mse_func() {
        assert_eq!((Loss::square_err().func)(2.0, 1.0), 0.5);
        assert_eq!((Loss::square_err().func)(5.0, 0.0), 12.5);
    }

    #[test]
    fn test_mse_der() {
        assert_eq!((Loss::square_err().der)(2.0, 1.0), 1.0);
        assert_eq!((Loss::square_err().der)(5.0, 0.0), 5.0);
    }

    #[test]
    fn test_mse() {
```

```rust
        let mut loss = Loss::square_err();

        let l = loss.criterion(&vec![2.0, 1.0, 0.0], &vec![0.0, 1.0, 2.0]);
        assert_eq!(l, 4.0);

        loss.criterion(
            &vec![34.0, 37.0, 44.0, 47.0, 48.0],
            &vec![37.0, 40.0, 46.0, 44.0, 46.0],
        );
        assert_eq!(l, 4.0);
    }

    #[test]
    fn test_bce_func() {
        println!("{}", (Loss::bce().func)(0.9, 0.0));
        println!("{}", (Loss::bce().func)(0.9, 1.0));
    }
}

```

Source Code 8: utills/data.rs

```rust
use super::io::read_lines;
use rand::prelude::SliceRandom;
use serde::Deserialize;
use std::error::Error;

pub fn max(vec: &Vec<f64>) -> f64 {
    vec.iter().fold(f64::NAN, |max, &v| v.max(max))
}

pub fn min(vec: &Vec<f64>) -> f64 {
    vec.iter().fold(f64::NAN, |min, &v| v.min(min))
}

pub fn std(vec: &Vec<f64>, mean: f64) -> f64 {
    let n = vec.len() as f64;
    vec.iter()
        .fold(0.0f64, |sum, &val| sum + (val - mean).powi(2) / n)
        .sqrt()
}

pub fn mean(vec: &Vec<f64>) -> f64 {
    let n = vec.len() as f64;
    vec.iter().fold(0.0f64, |mean, &val| mean + val / n)
}

pub fn standardization(data: &Vec<f64>, mean: f64, std: f64) -> Vec<f64> {
    data.iter().map(|x| (x - mean) / std).collect()
}

pub fn minmax_norm(data: &Vec<f64>, min: f64, max: f64) -> Vec<f64> {
    data.iter().map(|x| (x - min) / (max - min)).collect()
}

#[derive(Debug, Clone)]
pub struct Data {
    pub inputs: Vec<f64>,
    pub labels: Vec<f64>,
}
#[derive(Clone)]
pub struct DataSet {
    datas: Vec<Data>,
}

impl DataSet {
    pub fn new(datas: Vec<Data>) -> DataSet {
        DataSet { datas }
    }

    pub fn cross_valid_set(&self, percent: f64) -> Vec<(DataSet, DataSet)> {
        if percent < 0.0 && percent > 1.0 {
            panic!("argument percent must be in range [0, 1]")
        }
        let k = (percent * (self.datas.len() as f64)).ceil() as usize; // fold size
```

```rust
        let n = (self.datas.len() as f64 / k as f64).ceil() as usize; // number of folds
        let datas = self.get_shuffled().clone(); // shuffled data before slicing it
        let mut set: Vec<(DataSet, DataSet)> = vec![];

        let mut curr: usize = 0;
        for _ in 0..n {
            let r_pt: usize = if curr + k > datas.len() {
                datas.len()
            } else {
                curr + k
            };

            let validation_set: Vec<Data> = datas[curr..r_pt].to_vec();
            let training_set: Vec<Data> = if curr > 0 {
                let mut temp = datas[0..curr].to_vec();
                temp.append(&mut datas[r_pt..datas.len()].to_vec());
                temp
            } else {
                datas[r_pt..datas.len()].to_vec()
            };

            set.push((DataSet::new(training_set), DataSet::new(validation_set)));
            curr += k
        }
        set
    }

    pub fn data_points(&self) -> Vec<f64> {
        let mut data_points: Vec<f64> = vec![];
        for mut dt in self.datas.clone() {
            data_points.append(&mut dt.inputs);
            data_points.append(&mut dt.labels);
        }
        data_points
    }

    pub fn max(&self) -> f64 {
        max(&self.data_points())
    }

    pub fn min(&self) -> f64 {
        min(&self.data_points())
    }

    pub fn std(&self) -> f64 {
        std(&self.data_points(), self.mean())
    }

    pub fn mean(&self) -> f64 {
        mean(&self.data_points())
    }

    pub fn len(&self) -> usize {
        self.datas.len()
    }

    pub fn standardization(&self) -> DataSet {
        // this kind of wrong
        let mean = self.mean();
        let std = self.std();
        let datas: Vec<Data> = self
            .get_datas()
            .into_iter()
            .map(|dt| {
                let inputs: Vec<f64> = standardization(&dt.inputs, mean, std);
                let labels: Vec<f64> = standardization(&dt.labels, mean, std);
                Data { inputs, labels }
            })
            .collect();
        DataSet::new(datas)
    }

    /// this could be implement to be cleaner but I'm lazy
    pub fn minmax_norm(&self, valid_set: &DataSet) -> (DataSet, DataSet) {
        // this is very not efficient
        let size = self.datas[0].inputs.len();
```

```
        let mut features: Vec<Vec<f64>> = Vec::with_capacity(size);
        let mut v_features: Vec<Vec<f64>> = Vec::with_capacity(size);

        for _ in 0..size {
            features.push(vec![]);
            v_features.push(vec![]);
        }
        for dt in self.datas.iter() {
            for (f, x) in features.iter_mut().zip(dt.inputs.iter()) {
                f.push(*x);
            }
        }
        for v_dt in valid_set.datas.iter() {
            for (vf, vx) in v_features.iter_mut().zip(v_dt.inputs.iter()) {
                vf.push(*vx);
            }
        }
        for (f, vf) in features.iter_mut().zip(v_features.iter_mut()) {
            let (min, max) = (min(f), max(f));
            *f = minmax_norm(f, min, max);
            *vf = minmax_norm(vf, min, max);
        }

        let datas: Vec<Data> = self
            .datas
            .iter()
            .enumerate()
            .map(|(i, dt)| {
                let inputs: Vec<f64> = features.iter().map(|x| x[i]).collect();
                Data {
                    labels: dt.labels.clone(),
                    inputs,
                }
            })
            .collect();

        let v_datas: Vec<Data> = valid_set
            .datas
            .iter()
            .enumerate()
            .map(|(i, dt)| {
                let inputs: Vec<f64> = v_features.iter().map(|x| x[i]).collect();
                Data {
                    labels: dt.labels.clone(),
                    inputs,
                }
            })
            .collect();

        (DataSet::new(datas), DataSet::new(v_datas))
    }

    pub fn get_datas(&self) -> Vec<Data> {
        self.datas.clone()
    }

    pub fn get_shuffled(&self) -> Vec<Data> {
        let mut shuffled_datas = self.datas.clone();
        shuffled_datas.shuffle(&mut rand::thread_rng());
        shuffled_datas
    }
}

pub fn confusion_count(
    matrix: &mut [[i32; 2]; 2],
    result: &Vec<f64>,
    label: &Vec<f64>,
    threshold: f64,
) {
    if result[0] > threshold {
        // true positive
        if label[0] == 1.0 {
            matrix[0][0] += 1
        } else {
            // false negative
            matrix[1][0] += 1
```

```
        }
    } else if result[0] <= threshold {
        // true negative
        if label[0] == 0.0 {
            matrix[1][1] += 1
        }
        // false positive
        else {
            matrix[0][1] += 1
        }
    }
}

pub fn un_standardization(value: f64, mean: f64, std: f64) -> f64 {
    value * std + mean
}

pub fn xor_dataset() -> DataSet {
    let inputs = vec![[0.0, 0.0], [0.0, 1.0], [1.0, 0.0], [1.0, 1.0]];
    let labels = vec![[0.0], [1.0], [1.0], [0.0]];
    let mut datas: Vec<Data> = vec![];
    for i in 0..4 {
        datas.push(Data {
            inputs: inputs[i].to_vec(),
            labels: labels[i].to_vec(),
        });
    }

    DataSet::new(datas)
}

pub fn flood_dataset() -> Result<DataSet, Box<dyn Error>> {
    #[derive(Deserialize)]
    struct Record {
        s1_t3: f64,
        s1_t2: f64,
        s1_t1: f64,
        s1_t0: f64,
        s2_t3: f64,
        s2_t2: f64,
        s2_t1: f64,
        s2_t0: f64,
        t7: f64,
    }

    let mut datas: Vec<Data> = vec![];
    let mut reader = csv::Reader::from_path("data/flood_dataset.csv")?;
    for record in reader.deserialize() {
        let record: Record = record?;
        let mut inputs: Vec<f64> = vec![];
        // station 1
        inputs.push(record.s1_t3);
        inputs.push(record.s1_t2);
        inputs.push(record.s1_t1);
        inputs.push(record.s1_t0);
        // station 2
        inputs.push(record.s2_t3);
        inputs.push(record.s2_t2);
        inputs.push(record.s2_t1);
        inputs.push(record.s2_t0);

        let labels: Vec<f64> = vec![f64::from(record.t7)];
        datas.push(Data { inputs, labels });
    }
    Ok(DataSet::new(datas))
}

pub fn cross_dataset() -> Result<DataSet, Box<dyn Error>> {
    let mut datas: Vec<Data> = vec![];
    let mut lines = read_lines("data/cross.pat")?;
    while let (Some(_), Some(Ok(l1)), Some(Ok(l2))) = (lines.next(), lines.next(), lines.next()) {
        let mut inputs: Vec<f64> = vec![];
        let mut labels: Vec<f64> = vec![];
        for w in l1.split(" ") {
            let v: f64 = w.parse().unwrap();
            inputs.push(v);
```

```
        }
        for w in l2.split(" ") {
            let v: f64 = w.parse().unwrap();
            // class 1 0 -> 1
            // class 0 1 -> 0
            labels.push(v);
            break;
        }
        datas.push(Data { inputs, labels });
    }
    Ok(DataSet::new(datas))
}

pub fn wdbc_dataset() -> Result<DataSet, Box<dyn Error>> {
    let mut datas: Vec<Data> = vec![];
    let mut lines = read_lines("data/wdbc.txt")?;
    while let Some(Ok(line)) = lines.next() {
        let mut inputs: Vec<f64> = vec![];
        let mut labels: Vec<f64> = vec![]; // M (malignant) = 1.0, B (benign) = 0.0
        let arr: Vec<&str> = line.split(",").collect();
        if arr[1] == "M" {
            labels.push(1.0);
        } else if arr[1] == "B" {
            labels.push(0.0);
        }
        for w in &arr[2..] {
            let v: f64 = w.parse()?;
            inputs.push(v);
        }
        datas.push(Data { inputs, labels });
    }
    Ok(DataSet::new(datas))
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn temp_test() -> Result<(), Box<dyn Error>> {
        let dt = wdbc_dataset()?;
        println!("{:?}", dt.get_datas()[0].inputs.len());

        /*
        let dt = flood_dataset()?.cross_valid_set(0.1);
        let training_set = &dt[0].0;
        let validation_set = &dt[0].1;

        println!("mean: {}, std: {}", validation_set.mean(), validation_set.std());
        println!("\n{:?}", validation_set.get_datas());
        println!("\n\n{:?}", standardization(validation_set).get_datas());
         */

        /*
        if let Ok(dt) = cross_dataset() {
            println!("{:?}", dt.get_datas());
        }
        */
        Ok(())
    }

    #[test]
    fn test_min_max() -> Result<(), Box<dyn Error>> {
        let dt = flood_dataset()?;
        assert_eq!(dt.max(), 628.0);
        assert_eq!(dt.min(), 95.0);
        Ok(())
    }
}

```

Source Code 9: utills/graph.rs

```rust
use plotters::coord::Shift;
use plotters::prelude::*;
use std::error::Error;

const FONT: &str = "Roboto Mono";
const CAPTION: i32 = 70;
const SERIE_LABEL: i32 = 32;
const AXIS_LABEL: i32 = 40;

pub struct LossGraph {
    loss: Vec<Vec<f64>>,
    valid_loss: Vec<Vec<f64>>,
}

impl LossGraph {
    pub fn new() -> LossGraph {
        let loss: Vec<Vec<f64>> = vec![];
        let valid_loss: Vec<Vec<f64>> = vec![];
        LossGraph { loss, valid_loss }
    }

    pub fn add_loss(&mut self, training: Vec<f64>, validation: Vec<f64>) {
        self.loss.push(training);
        self.valid_loss.push(validation);
    }
    /// Draw training loss and validation loss at each epoch (x_vec)
    pub fn draw_loss(
        &self,
        idx: u32,
        root: &DrawingArea<BitMapBackend, Shift>,
        loss_vec: &Vec<f64>,
        valid_loss_vec: &Vec<f64>,
        max_loss: f64,
    ) -> Result<(), Box<dyn Error>> {
        let min_loss1 = loss_vec.iter().fold(f64::NAN, |min, &val| val.min(min));
        let min_loss2 = valid_loss_vec
            .iter()
            .fold(f64::NAN, |min, &val| val.min(min));
        let min_loss = if min_loss1.min(min_loss2) > 0.0 {
            0.0
        } else {
            min_loss1.min(min_loss2)
        };

        let mut chart = ChartBuilder::on(&root)
            .caption(
                format!("Loss {}", idx),
                ("Hack", 44, FontStyle::Bold).into_font(),
            )
            .margin(20)
            .x_label_area_size(50)
            .y_label_area_size(60)
            .build_cartesian_2d(0..loss_vec.len(), min_loss..max_loss)?;

        chart
            .configure_mesh()
            .y_desc("Loss")
            .x_desc("Epochs")
            .axis_desc_style(("Hack", 20))
            .draw()?;

        chart.draw_series(LineSeries::new(
            loss_vec.iter().enumerate().map(|(i, x)| (i + 1, *x)),
            &RED,
        ))?;

        chart.draw_series(LineSeries::new(
            valid_loss_vec.iter().enumerate().map(|(i, x)| (i + 1, *x)),
            &BLUE,
        ))?;

        root.present()?;
        Ok(())
    }

    pub fn max_loss(&self) -> f64 {
```

```
        f64::max(
            self.loss.iter().fold(f64::NAN, |max, vec| {
                let max_loss = vec.iter().fold(f64::NAN, |max, &val| val.max(max));
                f64::max(max_loss, max)
            }),
            self.valid_loss.iter().fold(f64::NAN, |max, vec| {
                let max_loss = vec.iter().fold(f64::NAN, |max, &val| val.max(max));
                f64::max(max_loss, max)
            }),
        )
    }

    pub fn draw(&self, path: String) -> Result<(), Box<dyn Error>> {
        let root = BitMapBackend::new(&path, (2000, 1000)).into_drawing_area();
        root.fill(&WHITE)?;
        // hardcode for 10 iteraions
        let drawing_areas = root.split_evenly((2, 5));

        let mut loss_iter = self.loss.iter();
        let mut valid_loss_iter = self.valid_loss.iter();
        let max_loss = self.max_loss();
        for (drawing_area, idx) in drawing_areas.iter().zip(1..) {
            if let (Some(loss_vec), Some(valid_loss_vec)) =
                (loss_iter.next(), valid_loss_iter.next())
            {
                self.draw_loss(idx, drawing_area, loss_vec, valid_loss_vec, max_loss)?;
            }
        }
        Ok(())
    }
}

/// Draw histogram of given datas
/// axes_desc - (for x, for y)
pub fn draw_acc_hist(
    datas: &Vec<f64>,
    title: &str,
    axes_desc: (&str, &str),
    path: String,
) -> Result<(), Box<dyn Error>> {
    let n = datas.len();
    let mean = datas
        .iter()
        .fold(0.0f64, |mean, &val| mean + val / n as f64);

    let root = BitMapBackend::new(&path, (1024, 768)).into_drawing_area();
    root.fill(&WHITE)?;

    let mut chart = ChartBuilder::on(&root)
        .caption(title, ("Hack", 44, FontStyle::Bold).into_font())
        .margin(20)
        .x_label_area_size(50)
        .y_label_area_size(60)
        .build_cartesian_2d((1..n).into_segmented(), 0.0..1.0)?
        .set_secondary_coord(1..n, 0.0..1.0);

    chart
        .configure_mesh()
        .disable_x_mesh()
        .y_max_light_lines(0)
        .y_desc(axes_desc.1)
        .x_desc(axes_desc.0)
        .axis_desc_style(("Hack", 20))
        .y_labels(3)
        .draw()?;

    let hist = Histogram::vertical(&chart)
        .style(RED.mix(0.5).filled())
        .margin(10)
        .data(datas.iter().enumerate().map(|(i, x)| (i + 1, *x)));

    chart.draw_series(hist)?;

    chart
        .draw_secondary_series(LineSeries::new(
            datas.iter().enumerate().map(|(i, _)| (i + 1, mean)),
```

```
            BLUE.filled().stroke_width(2),
        ))?
        .label(format!("mean: {:.3}", mean))
        .legend(|(x, y)| PathElement::new(vec![(x, y), (x + 20, y)], &BLUE));

    chart
        .configure_series_labels()
        .label_font(("Hack", 14).into_font())
        .background_style(&WHITE)
        .border_style(&BLACK)
        .draw()?;

    root.present()?;
    Ok(())
}

pub fn draw_acc_2hist(
    datas: [&Vec<f64>; 2],
    title: &str,
    axes_desc: (&str, &str),
    path: String,
) -> Result<(), Box<dyn Error>> {
    let n = datas.iter().fold(0f64, |max, l| max.max(l.len() as f64));
    let mean: Vec<f64> = datas
        .iter()
        .map(|l| {
            l.iter()
                .fold(0f64, |mean, &val| mean + val / l.len() as f64)
        })
        .collect();

    let root = BitMapBackend::new(&path, (1024, 768)).into_drawing_area();
    root.fill(&WHITE)?;

    let mut chart = ChartBuilder::on(&root)
        .caption(title, (FONT, CAPTION, FontStyle::Bold).into_font())
        .margin(20)
        .x_label_area_size(70)
        .y_label_area_size(90)
        .build_cartesian_2d((1..n as u32).into_segmented(), 0.0..1.0)?
        .set_secondary_coord(0.0..n, 0.0..1.0);

    chart
        .configure_mesh()
        .disable_x_mesh()
        .y_max_light_lines(0)
        .y_desc(axes_desc.1)
        .x_desc(axes_desc.0)
        .axis_desc_style((FONT, AXIS_LABEL))
        .y_labels(3)
        .label_style((FONT, AXIS_LABEL - 10))
        .draw()?;

    let a = datas[0].iter().zip(0..).map(|(y, x)| {
        Rectangle::new(
            [(x as f64 + 0.1, *y), (x as f64 + 0.5, 0f64)],
            Into::<ShapeStyle>::into(&RED.mix(0.5)).filled(),
        )
    });

    let b = datas[1].iter().zip(0..).map(|(y, x)| {
        Rectangle::new(
            [(x as f64 + 0.5, *y), (x as f64 + 0.9, 0f64)],
            Into::<ShapeStyle>::into(&BLUE.mix(0.5)).filled(),
        )
    });

    chart.draw_secondary_series(a)?;
    chart.draw_secondary_series(b)?;

    let v: Vec<usize> = (0..(n + 1.0) as usize).collect();
    chart
        .draw_secondary_series(LineSeries::new(
            v.iter().map(|i| (*i as f64, mean[0])),
            RED.filled().stroke_width(2),
        ))?
```

```
        .label(format!("mean: {:.3}", mean[0]))
        .legend(|(x, y)| PathElement::new(vec![(x, y), (x + 20, y)], &RED));

    chart
        .draw_secondary_series(LineSeries::new(
            v.iter().map(|i| (*i as f64, mean[1])),
            BLUE.filled().stroke_width(2),
        ))?
        .label(format!("mean: {:.3}", mean[1]))
        .legend(|(x, y)| PathElement::new(vec![(x, y), (x + 20, y)], &BLUE));

    chart
        .configure_series_labels()
        .label_font((FONT, SERIE_LABEL).into_font())
        .background_style(&WHITE)
        .border_style(&BLACK)
        .draw()?;

    root.present()?;
    Ok(())
}

/// Draw confusion matrix
pub fn draw_confustion(matrix_vec: Vec<[[i32; 2]; 2]>, path: String) -> Result<(), Box<dyn Error>> {
    let root = BitMapBackend::new(&path, (2000, 1100)).into_drawing_area();
    root.fill(&WHITE)?;

    let (top, down) = root.split_vertically(1000);

    let mut chart = ChartBuilder::on(&down)
        .margin(20)
        .margin_left(40)
        .margin_right(40)
        .x_label_area_size(40)
        .build_cartesian_2d(0i32..50i32, 0i32..1i32)?;
    chart
        .configure_mesh()
        .disable_y_axis()
        .disable_y_mesh()
        .x_labels(3)
        .label_style((FONT, 40))
        .draw()?;

    chart.draw_series((0..50).map(|x| {
        Rectangle::new(
            [(x, 0), (x + 1, 1)],
            HSLColor(
                240.0 / 360.0 - 240.0 / 360.0 * (x as f64 / 50.0),
                0.7,
                0.1 + 0.4 * x as f64 / 50.0,
            )
            .filled(),
        )
    }))?;
    // hardcode for 10 iteraions
    let drawing_areas = top.split_evenly((2, 5));
    let mut matrix_iter = matrix_vec.iter();
    for (drawing_area, idx) in drawing_areas.iter().zip(1..) {
        if let Some(matrix) = matrix_iter.next() {
            let mut chart = ChartBuilder::on(&drawing_area)
                .caption(
                    format!("Iteration {}", idx),
                    (FONT, 40, FontStyle::Bold).into_font(),
                )
                .margin(20)
                .build_cartesian_2d(0i32..2i32, 2i32..0i32)?
                .set_secondary_coord(0f64..2f64, 2f64..0f64);

            chart
                .configure_mesh()
                .disable_axes()
                .max_light_lines(4)
                .disable_x_mesh()
                .disable_y_mesh()
                .label_style(("Hack", 20))
                .draw()?;
```

```
            chart.draw_series(
                matrix
                    .iter()
                    .zip(0..)
                    .map(|(l, y)| l.iter().zip(0..).map(move |(v, x)| (x, y, v)))
                    .flatten()
                    .map(|(x, y, v)| {
                        Rectangle::new(
                            [(x, y), (x + 1, y + 1)],
                            HSLColor(
                                240.0 / 360.0 - 240.0 / 360.0 * (*v as f64 / 50.0),
                                0.7,
                                0.1 + 0.4 * *v as f64 / 50.0,
                            )
                            .filled(),
                        )
                    }),
            )?;

            chart.draw_secondary_series(
                matrix
                    .iter()
                    .zip(0..)
                    .map(|(l, y)| l.iter().zip(0..).map(move |(v, x)| (x, y, v)))
                    .flatten()
                    .map(|(x, y, v)| {
                        let text: String = if x == 0 && y == 0 {
                            format!["TP:{}", v]
                        } else if x == 1 && y == 0 {
                            format!["FP:{}", v]
                        } else if x == 0 && y == 1 {
                            format!["FN:{}", v]
                        } else {
                            format!["TN:{}", v]
                        };

                        Text::new(
                            text,
                            ((2.0 * x as f64 + 0.7) / 2.0, (2.0 * y as f64 + 1.0) / 2.0),
                            FONT.into_font().resize(30.0).color(&WHITE),
                        )
                    }),
            )?;
        }
    }
    root.present()?;
    Ok(())
}

/// Receive each cross-validation vector of each individual fitness value.
pub fn draw_ga_progress(
    cv_fitness: &Vec<Vec<(i32, f64)>>,
    path: String,
) -> Result<(), Box<dyn Error>> {
    let root = BitMapBackend::new(&path, (2000, 1000)).into_drawing_area();
    root.fill(&WHITE)?;

    // This is mostly hardcoded
    let drawing_areas = root.split_evenly((2, 5));
    for ((drawing_area, idx), fitness) in drawing_areas.iter().zip(1..).zip(cv_fitness.iter()) {
        let mut chart = ChartBuilder::on(&drawing_area)
            .caption(
                format!("Iteration {}", idx),
                (FONT, 40, FontStyle::Bold).into_font(),
            )
            .margin(40)
            .x_label_area_size(20)
            .y_label_area_size(20)
            .build_cartesian_2d(0i32..200i32, 0.0..1.1)?;

        chart
            .configure_mesh()
            .x_labels(3)
            .y_labels(2)
            .label_style((FONT, 30))
```

```
            .max_light_lines(4)
            .draw()?;

        chart.draw_series(
            fitness
                .iter()
                .map(|x| Circle::new((x.0, x.1), 1, BLUE.mix(0.5).filled())),
        )?;
    }
    root.present()?;
    Ok(())
}

```

Source Code 10: utills/io.rs

```
use crate::activator;
use crate::mlp;
use serde_json::{json, to_writer_pretty, Value};
use std::error::Error;
use std::fs::create_dir;
use std::fs::File;
use std::io::Read;
use std::io::{self, BufRead};
use std::path::Path;

pub fn save(layers: &Vec<mlp::Layer>, path: String) -> Result<(), Box<dyn Error>> {
    let mut json: Vec<Value> = vec![];

    for l in layers {
        json.push(json!({
            "inputs": l.inputs.len(),
            "outputs": l.outputs.len(),
            "w": l.w,
            "b": l.b,
            "act": l.act.name
        }));
    }
    let result = json!(json);
    let file = File::create(path)?;
    to_writer_pretty(&file, &result)?;
    Ok(())
}

pub fn read_lines<P>(filename: P) -> io::Result<io::Lines<io::BufReader<File>>>
where
    P: AsRef<Path>,
{
    let file = File::open(filename)?;
    Ok(io::BufReader::new(file).lines())
}

pub fn read_file<P>(filename: P) -> Result<String, Box<dyn Error>>
where
    P: AsRef<Path>,
{
    let mut file = File::open(filename)?;
    let mut contents = String::new();
    file.read_to_string(&mut contents)?;
    Ok(contents)
}

pub fn load<P>(filename: P) -> Result<mlp::Net, Box<dyn Error>>
where
    P: AsRef<Path>,
{
    let contents = read_file(filename)?;

    let json: Value = serde_json::from_str(&contents)?;
    let mut layers: Vec<mlp::Layer> = vec![];

    for l in json.as_array().unwrap() {
        // default layer activation is simeple linear f(x) = x
        let mut layer = mlp::Layer::new(
            l["inputs"].as_u64().unwrap(),
```

```rust
            l["outputs"].as_u64().unwrap(),
            0.0,
            activator::linear(),
        );
        // setting activation function
        if l["act"] == "sigmoid" {
            layer.act = activator::sigmoid();
        }

        // setting weights and bias
        let w = l["w"].as_array().unwrap();
        let b = l["b"].as_array().unwrap();
        for j in 0..w.len() {
            layer.b[j] = b[j].as_f64().unwrap();
            let w_j = w[j].as_array().unwrap();
            for i in 0..w_j.len() {
                layer.w[j][i] = w_j[i].as_f64().unwrap();
            }
        }

        layers.push(layer);
    }

    Ok(mlp::Net::from_layers(layers))
}

/// Check if specify folder exists in models and img folder, if not create it
///
/// Return models path and img path
pub fn check_dir(folder: &str) -> Result<(String, String), Box<dyn Error>> {
    let models_path = format!("models/{}", folder);
    if !Path::new(&models_path).exists() {
        create_dir(&models_path)?;
    }
    let img_path = format!("report/images/{}", folder);
    if !Path::new(&img_path).exists() {
        create_dir(&img_path)?;
    }
    Ok((models_path, img_path))
}

```