# Efficient Neural Network Building: Simplifying Computations with 1D Vectors in Python and Boosting Speed with Rust

In developing my neural network (NN), I took a unique approach by utilizing a 1D weight vector for the required number of weights. This method simplifies forward NN calculations by accessing the 1D weight as a 2D weight, enhancing ease during backpropagation. I extended this simplicity to bias calculations as well. The project, implemented in both Python and Rust, addresses a common challenge of decreasing Python speed as the number of neurons increases. Leveraging Rust for performance, this dual-language implementation ensures efficient and scalable neural network development, offering a streamlined solution for both computation and speed optimization.

## Python code

```python
from random import uniform
from copy import copy
import matplotlib.pyplot as plt
import sys


class NN:
    def __init__(self, layers):
        self.layers = layers
        self.wts = []
        self.w_size = 0
        for i in range(len(layers) - 1):
            temp = layers[i : 2 + i][::-1]
            self.w_size += temp[0] * temp[1]
            self.wts.append(temp)
        self.w = [uniform(-1, 1) for _ in range(self.w_size)]
        self.b = [uniform(-1, 1) for _ in range(len(self.layers) - 1)]
        self.losses = []

    def forward(self, x):
        shift = 0
        for index, (r, c) in enumerate(self.wts):
            z = []
            for i in range(r):
                s = 0
                for j in range(c):
                    s += x[j] * self.w[shift + i * c + j]
                s += self.b[index]
                z.append(s)
            # print(z)
            shift += r * c
            x = z
        return x

    def forward_w(self, x, w, b, wi, h):
```

```
        shift = 0
        for index, (r, c) in enumerate(self.wts):
            z = []
            for i in range(r):
                s = 0
                for j in range(c):
                    if wi == shift + i * c + j:
                        s += x[j] * (w[shift + i * c + j] + h)
                    else:
                        s += x[j] * w[shift + i * c + j]
                s += b[index]
                z.append(s)
            # print(z)
            shift += r * c
            x = z
        return x

    def forward_b(self, x, w, b, bi, h):
        shift = 0
        for index, (r, c) in enumerate(self.wts):
            z = []
            for i in range(r):
                s = 0
                for j in range(c):
                    s += x[j] * w[shift + i * c + j]
                if index == bi:
                    s += b[index] + h
                else:
                    s += b[index]
                z.append(s)
            # print(z)
            shift += r * c
            x = z
        return x

    def predict(self, x):
        return [self.forward(i) for i in x]

    def loss(self, yt, y):
        l = 0
        for i, j in zip(yt, y):
            l += (i - j) ** 2
        l /= len(yt)
        return l

    def overall_loss(self, yt, y):
        l = 0
        for i in range(len(yt)):
            for j in range(len(yt[0])):
                l += (yt[i][j] - y[i][j]) ** 2
        l /= len(yt) * len(yt[0])
        return l

    def fit(self, x, y, epochs, h, lr):
        for epoch in range(epochs):
```

```
            for xi in range(len(x)):
                dw = []
                for wi in range(self.w_size):
                    wl = copy(self.w)
                    wl[wi] += h
                    wr = copy(self.w)
                    wr[wi] -= h
                    dwi = (
                        self.loss(y[xi], self.forward_w(x[xi],
                        self.w, self.b, wi, h))
                        - self.loss(
                            y[xi], self.forward_w(x[xi],
                            self.w, self.b, wi, -h)
                        )
                    ) / (2 * h)
                    dw.append(dwi)

                db = []
                for bi in range(len(self.b)):
                    bl = copy(self.b)
                    bl[bi] += h
                    br = copy(self.b)
                    br[bi] -= h
                    dbi = (
                        self.loss(y[xi], self.forward_b(x[xi],
                        self.w, self.b, bi, h))
                        - self.loss(
                            y[xi], self.forward_b(x[xi],
                        self.w, self.b, bi, -h)
                        )
                    ) / (2 * h)
                    db.append(dbi)

                for i in range(len(dw)):
                    self.w[i] -= lr * dw[i]

                for i in range(len(db)):
                    self.b[i] -= lr * db[i]
            l = self.overall_loss(y, self.predict(x))
            self.losses.append(l)
            print(f"\r{epoch+1}/{epochs}: loss: {l}    ",
                  end="", flush=True)
        print()

    def loss_plot(self, path=None):
        plt.plot(self.losses)
        if path:
            plt.savefig(path)
            plt.close()
        else:
            plt.show()


layers = list(map(int, sys.argv[1:]))
x = [[uniform(0, 1) for _ in range(layers[0])] for _ in range(100)]
```

```
true_nn = NN(layers)
y = true_nn.predict(x)

nn = NN(layers)
print(f"Initial Loss: {nn.overall_loss(y, nn.predict(x))}")
nn.fit(x, y, 100, 0.1, 0.01)
print(f"Final Loss: {nn.overall_loss(y, nn.predict(x))}")
nn.loss_plot()
```

### Output

```
$ python3 src/main.py 10 5 1
Initial Loss: 7.999480407188753
100/100: loss: 8.070065512389859e-08
Final Loss: 8.070065512389859e-08
```

## Rust code

### nn.rs

```
use rand::Rng;
use std::io::Write;

pub struct NN {
    layers: Vec<usize>,
    wts: Vec<(usize, usize)>,
    w_size: usize,
    w: Vec<f64>,
    b: Vec<f64>,
    losses: Vec<f64>,
}

impl NN {
    pub fn new(layers: Vec<usize>) -> Self {
        let mut rng = rand::thread_rng();
        let mut wts = Vec::with_capacity(layers.len() - 1);
        let mut w_size = 0;
        for i in 0..layers.len() - 1 {
            let temp = layers[i..2 + i].to_vec();
            w_size += temp[0] * temp[1];
            wts.push((temp[1], temp[0]));
        }
        let w: Vec<f64> = vec![0.0; w_size]
            .iter()
            .map(|_| rng.gen_range(-1.0..=1.0))
            .collect();
        let b: Vec<f64> = vec![0.0; layers.len() - 1]
            .iter()
            .map(|_| rng.gen_range(-1.0..=1.0))
```

```
            .collect();

        Self {
            layers,
            wts,
            w_size,
            w,
            b,
            losses: vec![],
        }
    }

    pub fn forward(&self, x: &Vec<f64>) -> Vec<f64> {
        let mut temp_x = x.clone();
        let mut shift = 0;
        for (index, (r, c)) in self.wts.iter().enumerate() {
            let mut z = Vec::with_capacity(*r);
            for i in 0..*r {
                let mut s = 0.0;
                for j in 0..*c {
                    s += temp_x[j] * self.w[shift + i * c + j];
                }
                s += self.b[index];
                z.push(s);
            }
            // print(z)
            shift += r * c;
            // std::mem::swap(&mut temp_x, &mut z);
            temp_x = z;
        }
        temp_x
    }

    fn forward_w(&self, x: &Vec<f64>, w: &Vec<f64>, b: &Vec<f64>,
                 wi: usize, h: f64) -> Vec<f64> {
        let mut temp_x = x.clone();
        let mut shift = 0;
        for (index, (r, c)) in self.wts.iter().enumerate() {
            let mut z = Vec::with_capacity(*r);
            for i in 0..*r {
                let mut s = 0.0;
                for j in 0..*c {
                    if wi == shift + i * c + j {
                        s += temp_x[j] * (w[shift + i * c + j] + h);
                    } else {
                        s += temp_x[j] * w[shift + i * c + j];
                    }
                }
                s += b[index];
                z.push(s);
            }
            // print(z)
            shift += r * c;
            // std::mem::swap(&mut temp_x, &mut z);
            temp_x = z;
```

```
        }
        temp_x
    }

    fn forward_b(&self, x: &Vec<f64>, w: &Vec<f64>, b: &Vec<f64>,
                 bi: usize, h: f64) -> Vec<f64> {
        let mut temp_x = x.clone();
        let mut shift = 0;
        for (index, (r, c)) in self.wts.iter().enumerate() {
            let mut z = Vec::with_capacity(*r);
            for i in 0..*r {
                let mut s = 0.0;
                for j in 0..*c {
                    s += temp_x[j] * w[shift + i * c + j];
                }
                if index == bi {
                    s += b[index] + h;
                } else {
                    s += b[index];
                }
                z.push(s);
            }
            // print(z)
            shift += r * c;
            // std::mem::swap(&mut temp_x, &mut z);
            temp_x = z;
        }
        temp_x
    }

    pub fn predict(&self, x: &Vec<Vec<f64>>) -> Vec<Vec<f64>> {
        x.iter().map(|i| self.forward(i)).collect()
    }

    fn loss(&self, yt: &Vec<f64>, y: &Vec<f64>) -> f64 {
        let mut l = 0.0;
        for i in 0..yt.len() {
            l += (yt[i] - y[i]).powi(2);
        }
        l /= yt.len() as f64;
        l
    }

    pub fn overall_loss(&self, yt: &Vec<Vec<f64>>,
                        y: &Vec<Vec<f64>>) -> f64 {
        let mut l = 0.0;
        for i in 0..yt.len() {
            for j in 0..yt[0].len() {
                l += (yt[i][j] - y[i][j]).powi(2);
            }
        }
        l /= (yt.len() * yt[0].len()) as f64;
        l
    }
```

```
    pub fn fit(&mut self, x: &Vec<Vec<f64>>, y: &Vec<Vec<f64>>,
                epochs: usize, h: f64, lr: f64) {
        for epoch in 0..epochs {
            // println!("epochs:{}%", (epoch + 1) * 100 / epochs);
            // std::io::stdout().flush().unwrap();
            for xi in 0..x.len() {
                let mut dw = Vec::with_capacity(self.w_size);
                for wi in 0..self.w_size {
                    // print!("\rwi:{}%", (wi + 1) * 100 /self.w_size);
                    // std::io::stdout().flush().unwrap();
                    let dwi = (self.loss(&y[xi],
                        &self.forward_w(&x[xi], &self.w, &self.b, wi, h))
                        - self.loss(&y[xi], &self.forward_w(&x[xi],
                        &self.w, &self.b, wi, -h))) / (2.0 * h);
                    dw.push(dwi);
                }
                // println!();

                let mut db = Vec::with_capacity(self.b.len());
                for bi in 0..self.b.len() {
                    let dbi = (self.loss(&y[xi],
                        &self.forward_b(&x[xi], &self.w, &self.b, bi, h))
                        - self.loss(&y[xi], &self.forward_b(&x[xi],
                        &self.w, &self.b, bi, -h))) / (2.0 * h);
                    db.push(dbi);
                }

                for i in 0..dw.len() {
                    self.w[i] -= lr * dw[i];
                }

                for i in 0..db.len() {
                    self.b[i] -= lr * db[i];
                }
            }
            let l = self.overall_loss(&y, &self.predict(x));
            self.losses.push(l);
            print!("\r{}/{}: loss: {}", epoch + 1, epochs, l);
            std::io::stdout().flush().unwrap();
        }
        println!();
    }
}

pub fn rand_matrix(i: usize, j: usize) -> Vec<Vec<f64>> {
    let mut rng = rand::thread_rng();
    let mut v = vec![vec![0.0; j]; i];
    for a in 0..i {
        for b in 0..j {
            v[a][b] = rng.gen_range(-1.0..=1.0);
        }
    }
    v
}
```

## main.rs

```rust
mod nn;
use nn::{rand_matrix, NN};

fn input() -> Vec<usize> {
    let i: Vec<String> = std::env::args().collect();
    i[1..i.len()]
        .iter()
        .map(|i| i.parse::<usize>().unwrap())
        .collect()
}

fn main() {
    let layers = input();
    let x = rand_matrix(100, layers[0]);

    let true_nn = NN::new(layers.clone());
    let y = true_nn.predict(&x);

    let mut nn = NN::new(layers);
    println!("Initial Loss: {}",
             nn.overall_loss(&y, &nn.predict(&x)));
    nn.fit(&x, &y, 100, 0.00001, 0.001);
    println!("Final Loss: {}",
             nn.overall_loss(&y, &nn.predict(&x)));
}
```

### Output

```
$ cargo run --release 10 5 1
    Finished release [optimized] target(s) in 9.09s
     Running `target/release/Mnist_DL_from_scratch 10 5 1`
Initial Loss: 5.542724251467661
100/100: loss: 0.00000003194128331923568
Final Loss: 0.00000003194128331923568
```