Lecture 2

Mechanics of ANN

In this lecture we consider basic mathematical techniques (mechanics of ANN).

Good news - most algorithms and methods are known for you from previous courses. Some of them are included in the topics Machine Learning, Statistics, Numerical algorithms for modelling

The first aim - we want to have a possibility to evaluate the accuracy of prediction, done by ANN.

This information is connected to the following practical question

- how to select parameters (weights) of a ANN algorithm to guarantee that the error of the prediction is minimal or small (this aim is most constructive)
- how to select optimal architecture of ANN in order to solve a given class of applied problems. (prediction, classification)

Error estimation

This topic was considered in many courses of your SP.

- an approximation error of numerical schemes for ODE and PDE
- an accuracy of statistical estimates
- an error of prediction obtained by Machine Learning algorithms

The error can be evaluated by very different techniques (some of them are simple, the other require more deep analysis).

We will remind techniques based on norms of vectors and functions.

Let us consider a vector

$$X = (x_1, x_2, \dots, x_M)$$

We also computed the projection (approximation) of it:

$$Y = (y_1, y_2, \cdots, y_M).$$

The accuracy of approximation (the error) can be evaluated by using different norms:

1. Maximum error

$$E_\infty = \max_{1 \le j \le M} |x_j - y_j|.$$

(L_∞-norm).

2. Averaged error

$$E_1 = \frac{1}{M} \sum_{j=1}^{M} |x_j - y_j|$$

(L_1-norm).

3. Quadratic norm

$$E_2^2 = \frac{1}{M} \sum_{j=1}^{M} (x_j - y_j)^2$$

$$(L_2 = E_2 \text{ - norm})$$

Also, ~~can~~ generalizations of relative versions are popular tools (adaptivity), e.g.:

$$\tilde{E}_\infty = \max_{1 \le j \le M} \frac{|x_j - y_j|}{\varepsilon + |x_j|}$$

Compare absolute and relative versions of norms, what new information is obtained (controled) by relative norms?

Next we want to study the following questions:

- How to get an ~~echo~~ information on the accuracy of prediction done by ANN algorithm?
- How to improve this accuracy if the error is too large for aims of modelling?

This part of ANN mechanism is solved during training stage

First we consider some general mathematical framework and later we will see how it can be used for ANN techniques

Let us start from 1D case

Find $a \le \bar{x} \le b$:

$$\min_{a \le x \le b} f(x) = f(\bar{x}).$$

The searching procedure is simple

We select initial approximation

$\begin{cases} a < x^0 < b \\ f_{min} = f(x^0). \end{cases}$ and compute two new predictions of $\bar{x}$: (with some step Δx)

$x^0_{dn} = x^0 - \Delta x$, (try down value)

$x^0_{up} = x^0 + \Delta x$. (try up value)

Check the new values of f:

$f(x^0_{dn})$ and $f(x^0_{up})$.

Test the obtained values of f:

$n = 1$

$x^n = x^0$

if $(f_{dn}^{n-1} < f_{min})$:

$x^n = x^0_{dn}$

$f_{min} = f_{dn}^{n-1}$

if $(f_{up}^{n-1} < f_{min})$:

$x^n = x^0_{up}$

$f_{min} = f_{up}^{n-1}$.

If $x^n \neq x^{n-1}$ then we continue iterations with $n := n+1$.

Remarks 1. In general we find a direction to minimize the value of f

2. How to define Δx to have a convergence of the algorithm?

Gradient descent method

We known from lectures on math. analysis, that the gradient ∇f ($= \frac{\partial f}{\partial x}$ for 1D functions) defines the direction where $f(x)$ increases fastest if one goes in this direction, and it decreases fastest if one goes from x in the direction of the negative gradient.

For a small enough step size η,

$$f(x^{n+1}) \leq f(x^n) \text{ if}$$

$$x^{n+1} = x^n - \eta \nabla f(x^n).$$

$$= x^n - \eta \frac{\partial f}{\partial x}(x^n).$$

For

$$f(x) = f(x_1, x_2, \dots, x_M)$$

the gradient is defined as

$$\nabla f(x) = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \dots, \frac{\partial f}{\partial x_M} \right)_{\vec{x}=x}$$

and we get the algorithm

$$x_j^{n+1} = x_j^n - \eta \left. \frac{\partial f}{\partial x_j} \right|_{x=x^n}$$

enabling (for η sufficiently small)

$$f(x^{n+1}) < f(x^n) < f(x^{n-1}) < \dots$$

Some versions of the gradient descent method serve as the most basic algorithm used for training most popular deep neural networks.

We start analysis of the training process for most simple neural network when we have

- one input data - inp number
- one output data - goal number

Then it is sufficient to define one parameter - weight w_1.

Prediction algorithm and error

```
pred = neural_network(inp, w_1)
```

Remind that network gives:

```
pred = inp * w_1.
```

Error (or loss function) is defined as

$$error = \frac{1}{2}(pred - goal)^2$$

python v.: $(pred - goal) ** 2$

Define a notation

$$delta = pred - goal$$

Then

$$error = \frac{1}{2} * delta ** 2$$

The gradient of error $\nabla error$

$$\left(= \frac{\partial error}{\partial w_1} \right)$$

is defined as

$$\frac{\partial error}{\partial w_1} = delta * \frac{\partial delta}{\partial w_1}$$

$$\frac{\partial delta}{\partial w_1} = \frac{\partial pred}{\partial w_1} = \frac{\partial (inp * w_1)}{\partial w_1} = inp$$

$$\Rightarrow \frac{\partial error}{\partial w_1} = inp * delta.$$

Update the weights

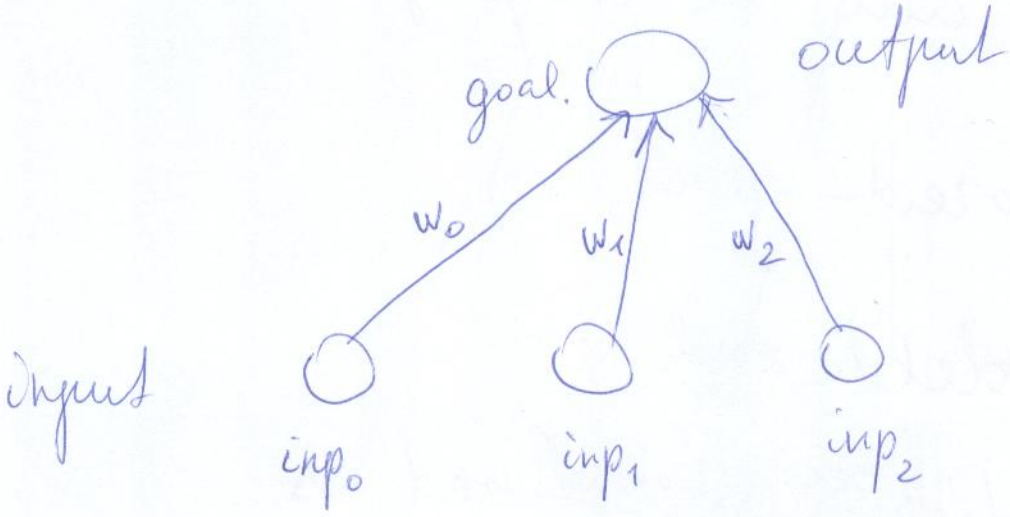
$$w_1^{n+1} = w_1^n - \gamma \; inp * delta$$

$$= w_1^n - \gamma * (inp * w_1^n - goal) * inp.$$

γ can be started with $\gamma = 1$ and we reduce γ if the process is not converging.

Gradient descent learning (training) with multiple inputs

The same technique can be used to update a network that contains multiple weights.



weights = $[w_0, w_1, w_2]$

input = $[inp_0, inp_1, inp_2]$

pred = $\sum_{j=0}^{2} inp_j w_j$

delta = pred - goal

pred = neural_network(input, weights

The loss function (error)

$$\text{error} = \frac{1}{2}(\text{pred} - \text{goal})^{**}2$$

$$= \frac{1}{2} * \text{delta} ** 2$$

where

$$\text{delta} = \text{pred} - \text{goal}$$

Now we calculate the gradient

$$\nabla \text{error} = \left(\frac{\partial \text{error}}{\partial w_0}, \frac{\partial \text{error}}{\partial w_1}, \frac{\partial \text{error}}{\partial w_2}\right)$$

$$\frac{\partial \text{error}}{\partial w_j} = \text{delta} \frac{\partial \text{delta}}{\partial w_j}$$

$$= \text{delta} \frac{\partial \text{pred}}{\partial w_j} = \text{delta} * \text{inp}_j.$$

$$W_j^{n+1} = W_j^n - \gamma \, \text{delta} * \text{inp}_j$$

$j = 0, 1, 2.$

Python code

```
def ele_mul(number, vector2)
    output = [0, 0, 0]
    for i in range (len(vector)):
        output[i] = number * vector[i]
    return output

input = [inp0, inp1, inp2]
pred = neeeral_network(input, weight)
error = 1/2 * (pred - goal) ** 2
delta = pred - goal
weight_deltas = ele_mult(delta, input)
for i in range (len(weights))
    weights[i] -= y * weight_deltas[i]
```