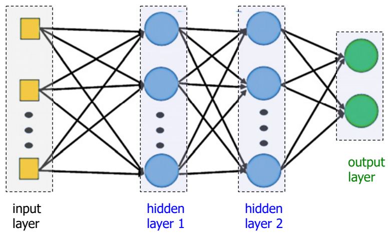
MLP and related topics

Akito Sakurai

Multilayer perceptron (MLP)

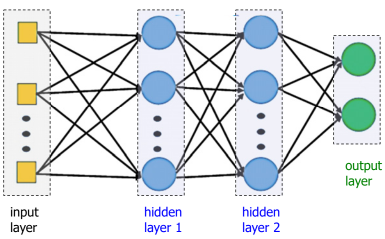
- A multilayer perceptron (MLP) is a network of nodes consisting of at least three layers of nodes (an input, a hidden and an output layer).



- The number of hidden layers and neurons in each hidden layer are hyperparameters to be set.

Purpose of MLP learning

Input x → **Non-linear function with parameters π** $y = \varphi(x; \pi)$ → **Output y**

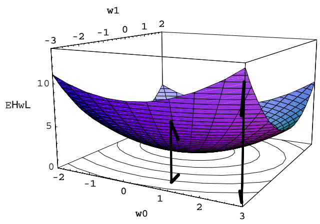


- Given a training dataset $\{(x_i, t_i)\}$, find π that minimizes an error or a loss, e.g., $\sum_i (y_i - t_i)^2$ where $y_i = \varphi(x_i; \pi)$

In fact, this is not correct. If this one is "try to minimize", it is almost correct.

How to minimize the error

- Many methods were proposed and tried.
- The simplest one among them is "steepest descent"
  - Or steepest ascent if you intend to find the maximum.



The direction of the "steepest descent is the opposite of the gradient of the objective function which is normal to its contour.
You will go along the steepest descent direction iteratively.

Steepest descent

- An iterative method: you descent the objective function value surface $E(w)=loss(w)$ iteratively.
- Gradient is the steepest ascent direction
- Therefore the it repeats

$$\Delta W = \alpha\left(-\frac{\partial E}{\partial W}\right) \qquad \text{where } E(w) = \sum_i (\varphi(x_i; w) - t_i)^2$$

$$W \leftarrow W + \Delta W$$

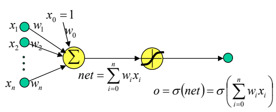
where α is called a learning rate which should be appropriately defined.

When you were undergraduate …

$$E = (t-y)^2$$

$$\frac{\partial E}{\partial W} = 2(t-y)\underbrace{\frac{\partial (t-y)}{\partial W}}$$

$$= -2(t-y)x$$



When the activation function is the identity function, i.e.

$$y = \sum_{i=1}^{n} w_i x_i$$

$$\frac{\partial (t-y)}{\partial W} = -\frac{\partial y}{\partial W} \quad \text{(t is a target, and is independent of W}$$

$$= -\frac{\partial \left(\sum_{i=1}^{n} w_i x_i\right)}{\partial W}$$

$$= -x \quad \text{(x is a vector)}$$

A bit more

$$W \leftarrow W + \alpha\left(-\frac{\partial E}{\partial W}\right) \quad \text{and} \quad \frac{\partial E}{\partial W} = -2(t-y)x$$

Therefore, if we rewrite 2α as α,

$$W \leftarrow W + \Delta W \quad \text{where} \quad \Delta W = \alpha(t-y)x$$

$$\text{where} \quad y = \sum_{i=1}^{n} w_i x_i \quad \text{is assumed}$$

Comparison with Perceptron learning algorithm

- If you look at the resulted rule, it looks like the perceptron learning rule:

$$W \leftarrow W + \Delta W \quad \text{where} \quad \Delta W = \begin{cases} x & \text{if } t \neq y \text{ and } t = 1 \\ -x & \text{if } t \neq y \text{ and } t = -1 \\ 0 & \text{if } t = y \end{cases}$$

which obtained by setting α=1, $t = \pm 1$, $y = \pm 1$ in the steepest descent:

$$W \leftarrow W + \Delta W \quad \text{where} \quad \Delta W = \alpha(t-y)x$$

- In other word, it looks like proving the validity of perceptron learning algorithm
- But not, because the threshold function is not the identity function and is not differentiable

For any activation function

$$E = (t-y)^2 \qquad y = f(y_{in}),\ y_{in} = \sum_{i=1}^{n} w_i x_i$$

$$\frac{\partial E}{\partial W} = 2(t-y)\underbrace{\frac{\partial (t-y)}{\partial W}}$$

$$= -2(t-y) f'(y_{in})x$$

$$\begin{aligned} \frac{\partial (t-y)}{\partial W} &= -\frac{\partial y}{\partial W} \\ &= -\frac{\partial f(y_{in})}{\partial W} \\ &= -\frac{\partial f(y_{in})}{\partial y_{in}}\frac{\partial y_{in}}{\partial W} \\ &= -f'(y_{in})\frac{\partial\left(\sum_{i=1}^{n} w_i x_i\right)}{\partial W} \\ &= -f'(y_{in})x \end{aligned}$$

(x is a vector)

In short

$$W \leftarrow W + \alpha\left(-\frac{\partial E}{\partial W}\right) \quad \text{where} \quad \frac{\partial E}{\partial W} = -2(t-y) f'(y_{in})x$$

Rewriting 2α as α,

$$W \leftarrow W + \Delta W \quad \text{where} \quad \Delta W = \alpha(t-y) f'(y_{in})x$$

By the way

- We have considered a learning rule when a sample is given.
- But this is not appropriate
- Because we have to consider the error or loss which is not just of a sample but of a set of all the samples.
- Because if we decrease an error caused by a single sample x_1 it may increase an error caused by another sample x_2 and may increase the total sum of errors.
- Therefore we have to consider

$$E = \sum_s (t_s - y_s)^2 \quad \text{but not} \quad E = (t-y)^2$$

and

$$W \leftarrow W + \Delta W \quad \text{where} \quad \Delta W = \alpha \sum_s (t_s - y_s) f'(y_{in,s}) x_s$$

(this will be called "batch mode" (vs. online mode))

Is this correct?

- Yes, it is. If the learning rate α is decreasing to 0 with appropriate speed, by iteratively applying the rule:

$$W \leftarrow W + \Delta W \quad \text{where} \quad \Delta W = \alpha \sum_s (t_s - y_s) f'(y_{in,s}) x_s$$

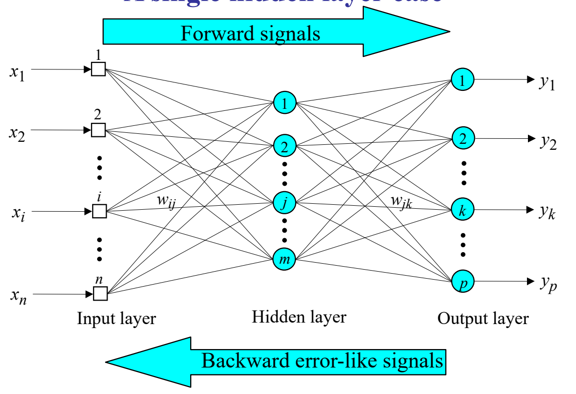
we can get at a (local) minimum of $E = \sum_s (t_s - y_s)^2$

- Then the batch mode is good enough?
- The story is not so simple.
- In reality, it is known empirically that the online mode minimization will give us smaller E.
  - Why?
- It is also known that mini-batch is better when there are many samples

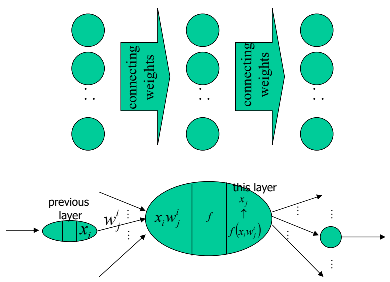
Multi-layer (or MLP) case

- No problem. The steepest descent is applicable.
- Well, in fact there is a problem.
- In case of MLP, the function to be minimized is very complicated. It is a compound function of many functions, although each component function is simple.
- There is a solution which was already found in 1980's and named error backpropagation algorithm.

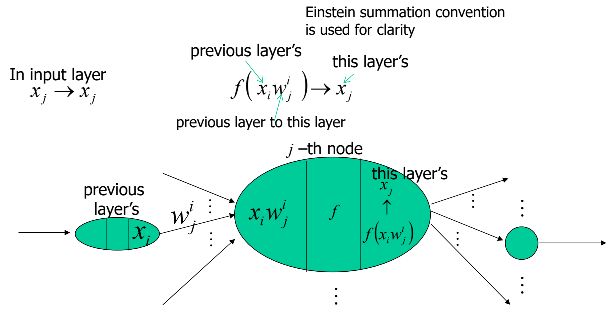
A single hidden layer case



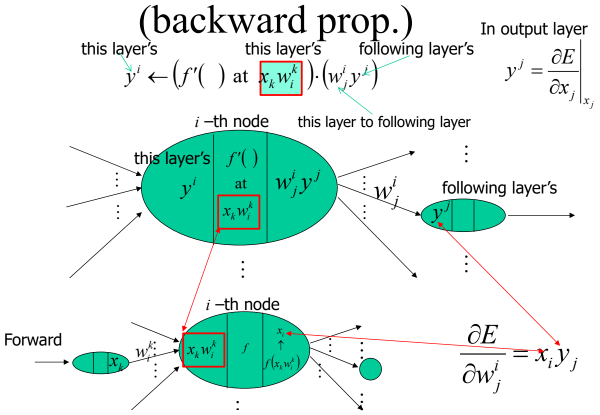
Error back-propagation or BP



Calculation of outputs (forward prop.)



Calculation of gradients (backward prop.)



Summary 1: Supervised learn.

- How to obtain parameter values:
  - Suppose $\{(x_t, y_t) | 1 \le t \le N\}$ is a training dataset,
  - network i/o relation is $y = F(W, x)$, and
  - an error function e.g.,

$$E(W) = \frac{1}{N}\sum_{t=1}^{N}\left(F(W, x_t) - y_t\right)^2$$

  - Find out W that minimizes this.

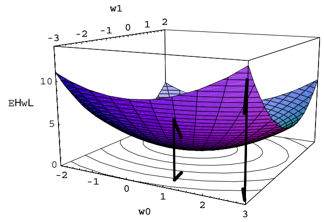
In fact, this is not correct. If this one is "try to minimize", it is almost correct.

Summary 2: method

- It is sufficient to solve a system of equations obtained by differentiating E and equating them to be 0. Is that correct?
  - E should be differentiable.
  - For the perceptron, E is not.
- The system is non-linear and complicated. There is no closed form solution.
- A practical solution is an iterative method that will find a series $W_1, W_2, W_3 \cdots$ for which $E(W_1) > E(W_2) > E(W_3) > \cdots$

Summary 3: iterative method

- Many methods have been proposed.
- The simplest one among them is "steepest descent"



The steepest descent direction is normal to contour lines (planes).

Summary 4: calculation

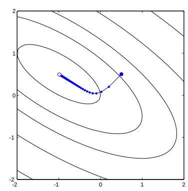
- How is the steepest descent calculated?
- The gradient is the steepest ascent.
- Therefore

$$\Delta w_i^j = -\eta \cdot \frac{\partial E}{\partial w_i^j}(W)$$

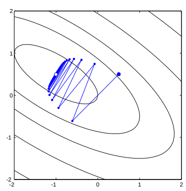
$$w_i^{j,new} = w_i^j + \Delta w_i^j$$

is the method where η is a learning rate which should be determined carefully.

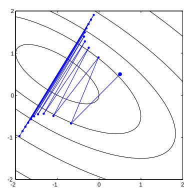
Behavior of progress



when the learning rate is small

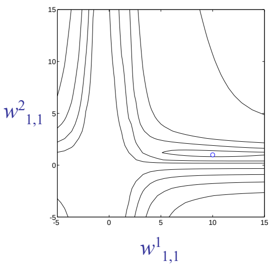


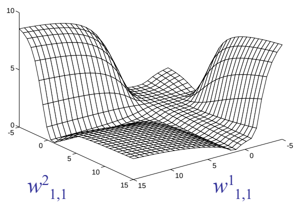
when the learning rate is relatively large



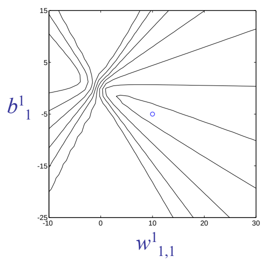
when the learning rate is large

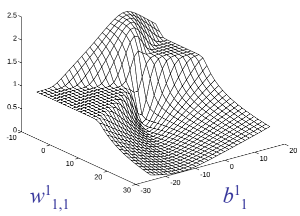
An example of E surface



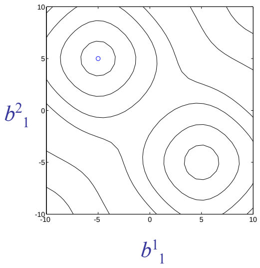


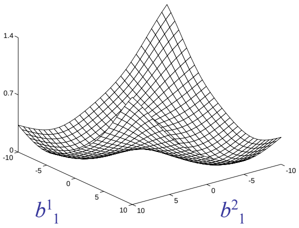
Another example



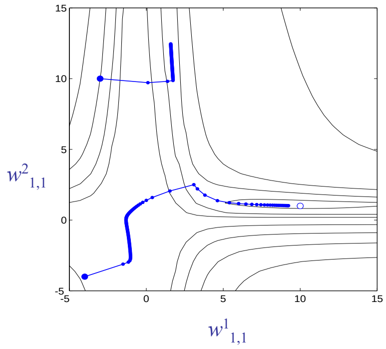


Another example

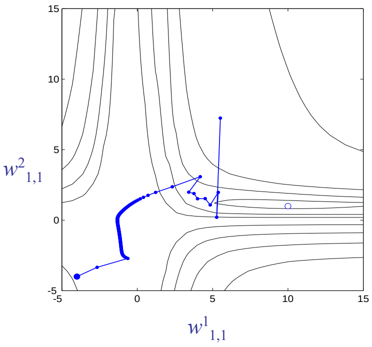




An example of convergence



An example of divergence

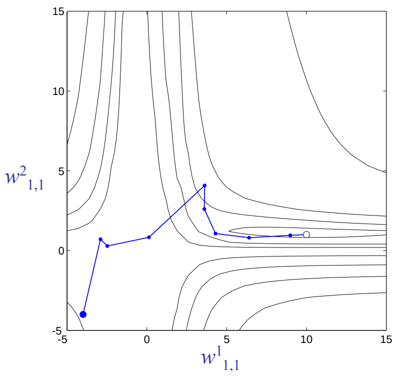


Is BP satisfactory?

- Of course not!
- There are many good optimization method better than steepest descent are known.
- In fact many methods were applied to NN.
- It worked relatively nicely but not remarkably well
  - One of the problems is computation time
    - Fast methods require matrix inversion of the size |W|∗|W| (|W| is the number of weights)
    - Methods that successively approximate the inversion of a matrix are often used.
  - But high rate of success and avoidance of local optima that compensate the high cost computation is not expected.
  - Neural Networks seem to be simple in a sense but in fact are very peculiar. There are many singular points in its search space.
- Levenberg-Marquardt might be a good method.

  Timothy Masters, Advanced Algorithms for Neural Networks: A C++ Sourcebook, John Wiley & Sons (1995).

Levenberg-Marquardt



Stochastic Gradient Descent

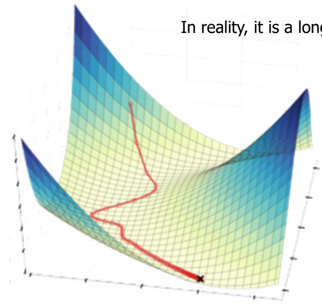
- On-line version of BP
  - One or more (but far fewer than all) samples are used for calculation of updates at one time.
    - The latter is called mini-batch in deep learning.
  - Samples to be used are selected randomly.
    - Usually all the samples are ordered randomly and are sampled in order from the first. When all the samples are used, they are re-ordered randomly.
  - In NN case, sometimes the convergence becomes very slow
    - Could you guess why?

$$\vec{w} \leftarrow \vec{w} - \alpha g_{\vec{w}} \qquad g_{\vec{w}} = \frac{\partial E}{\partial \vec{w}}$$

SGD for NN

- In processes of error-decreasing, there often exist of periods of very slow convergence.
  - It is called plateau
  - Why?
- There exists degeneracy in search space of weights where singular points of error function exist
  - Many occurrences of the same shape in the error function causes the singular points.
  - Current weight set passes close to the points.
- Many speeding up methods have been proposed.

Going through plateau



In reality, it is a long valley of gentle slope.

http://librimind.com/2016/03/optimizations-of-gradient-descent/

Utilization of momentum

- Basic idea:
  - On a plateau, steps are short but direction is similar.
  - To accelerate the process, adding previous steps seems to be promising.
    - It might be better to forget very old steps
  - Exponential discount of old steps is a good idea.

$$\vec{v} \leftarrow \mu\vec{v} - \alpha g_{\vec{w}}$$
$$\vec{w} \leftarrow \vec{w} + \vec{v}$$

⟵

$$\vec{w} \leftarrow \vec{w} - \alpha g_{\vec{w}}$$

Examples of acceleration

- AdaGrad

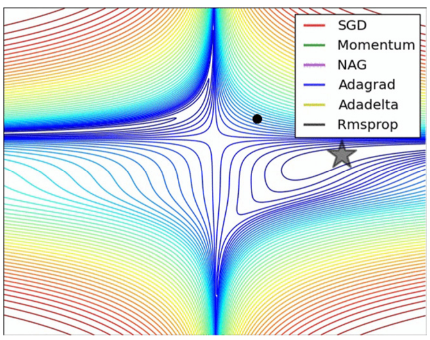
$$\vec{r} \leftarrow \vec{r} + g_{\vec{w}}^2$$
$$\vec{w} \leftarrow \vec{w} - \frac{\alpha}{\sqrt{\vec{r}} + \varepsilon} g_{\vec{w}}$$

- AdaDelta

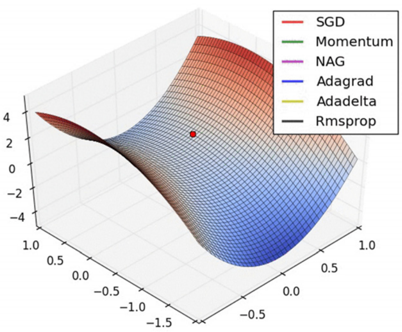
$$\vec{r} \leftarrow \beta\vec{r} + (1-\beta) g_{\vec{w}}^2$$
$$\vec{v} \leftarrow \frac{\sqrt{\vec{s}} + \varepsilon}{\sqrt{\vec{r}} + \varepsilon} g_{\vec{w}}$$
$$\vec{s} \leftarrow \beta\vec{s} + (1-\beta)\vec{v}^2$$
$$\vec{w} \leftarrow \vec{w} - \vec{v}$$

- Adam

$$\vec{v} \leftarrow \beta\vec{v} + (1-\beta) g_{\vec{w}}$$
$$\vec{r} \leftarrow \gamma\vec{r} + (1-\gamma) g_{\vec{w}}^2$$
$$\vec{w} \leftarrow \vec{w} - \frac{\alpha}{\sqrt{\frac{\vec{r}}{1-\gamma^t} + \varepsilon}} \frac{\vec{v}}{1-\beta^t} g_{\vec{w}}$$

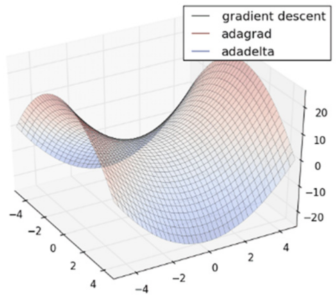


http://cs231n.github.io/neural-networks-3/



NAG: Nesterov Accelerated Gradient Descent

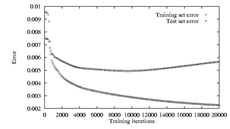
http://cs231n.github.io/neural-networks-3/



http://cpmarkchang.logdown.com/posts/467674-optimization-method-adadelta

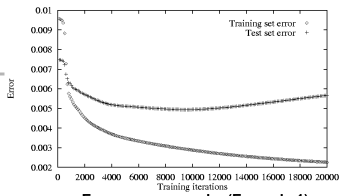
Overtraining in iterative methods

- Over-trained: when a model fits well to the training dataset but predicts badly on unseen samples.
  - A model behaves better on D_{train}, but worse on D_{test}
- For iterative methods:
  - (when properly initialized) they start under-trained (because under-trained models are omnipresent)
  - When learning process proceeds, the model fits better to the training dataset. Generally speaking, the models learn general regularity that exist in any choice of training datasets, i.e., the models are not over-trained.
  - When learning process goes further, the models start learning of specific regularity found in the training dataset that the models face, i.e., the models become over-trained.

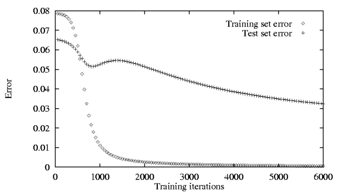


NN case

- Over-training process can be found by tracing prediction error on a validation dataset.
- But it is very difficult to declare a start of over-training.



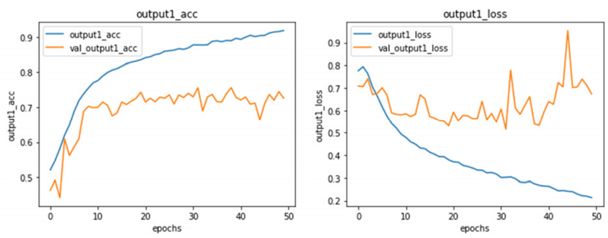
Error versus epochs (Example 1)



Error versus epochs (Example 2)

Tom Michel “Machine Learning”

Headaches in an NN case



Regularization

- To ease the damage of over-training, a kind of penalty function is added to the term to be minimized. The penalty function penalizes conducts of deviating far away from the starting point.

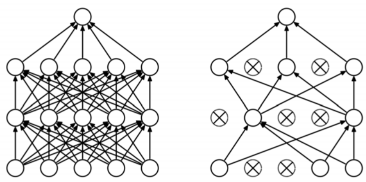
$$E(\vec{w}) \equiv \sum_{d\in D}\sum_{k\in outputs}\left(t_{k,d}-o_{k,d}\right)^2+\gamma\sum_{i,j} w_{j,i}{}^2$$

L1 relatively encourages sparsity $||\theta||_1 = \sum_i |\theta_i|$

L2 relatively discourages large weights $||\theta||_2 = \sqrt{\sum_i \theta_i^2}$

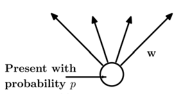
$$E(\vec{w}) \equiv \sum_{d\in D}\sum_{k\in outputs}\left[\left(t_{k,d}-o_{k,d}\right)^2+\mu\sum_{j\in inputs}\left(\frac{\partial t_{k,d}}{\partial x_d^j}-\frac{\partial o_{k,d}}{\partial x_d^j}\right)^2\right]$$

Dropout

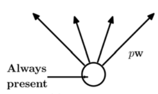


(a) Standard Neural Net (b) After applying dropout.

Not permanent but just randomly



(a) At training time



(b) At test time

Srivastava et al. 2014. “Dropout: a simple way to prevent neural networks from overfitting”