Back-Propagation

COMP4211

THE DEPARTMENT OF
COMPUTER SCIENCE & ENGINEERING
計算機科學及工程學系

Back-Propagation

Nonlinear activation functions + multi-layer networks

- requires more sophisticated learning algorithms

Back-propagation (Generalized delta rule)

Idea: Gradient descent

- start with initial value for **w**
- repeat until convergence
  - compute the gradient vector of the error function for current **w**
  - move in the opposite direction

How to Compute the Gradient?

network with differentiable activation functions

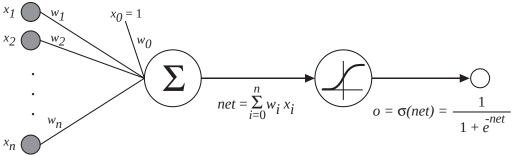
- outputs are differentiable functions of input and of the weights and biases

define an error function which is a differentiable function of the network outputs

- the error is a differentiable function of the weights
- use the chain rule to calculate the gradient

Network with One Sigmoid Unit

we first derive gradient decent rules to train one sigmoid unit

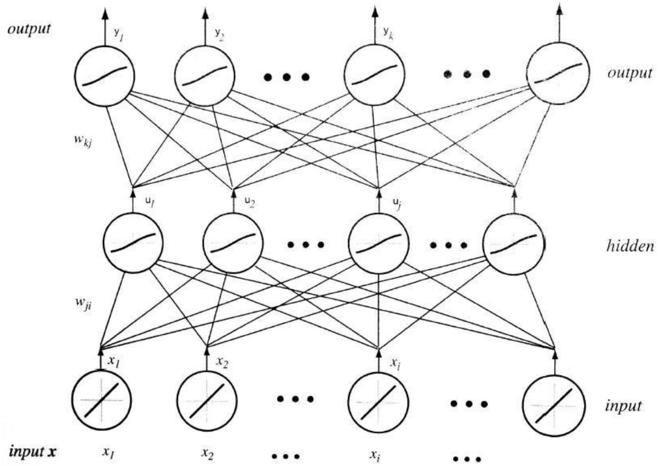


$$
\begin{aligned}
\frac{\partial E}{\partial w_i} &= \frac{\partial}{\partial w_i}\frac{1}{2}(t-o)^2 = \frac{1}{2}2(t-o)\frac{\partial}{\partial w_i}(t-o) \\
&= (t-o)\left(-\frac{\partial o}{\partial w_i}\right) = -(t-o)\,\frac{\partial o}{\partial net}\,\frac{\partial net}{\partial w_i}
\end{aligned}
$$

$$
\frac{\partial o}{\partial net} = \frac{\partial \sigma(net)}{\partial net} = o(1-o), \qquad \frac{\partial net}{\partial w_i} = \frac{\partial(\mathbf{w}'\mathbf{x})}{\partial w_i} = x_i
$$

$$
\frac{\partial E}{\partial w_i} = -(t-o)o(1-o)x_i
$$

MLP



- N_o output units
- error for one training example: $E_d = \sum_{k=1}^{N_o}(t_{d,k} - o_{d,k})^2$

Error Gradient of Weights to Output Unit k

- w_{kj}: weight for the link from unit j to (output) unit k

$$
\begin{aligned}
\frac{\partial E_d}{\partial w_{kj}} &= \frac{1}{2}\frac{\partial}{\partial w_{kj}}\sum_{m=1}^{N_o}(t_{d,m}-o_{d,m})^2 \quad \text{(dropping } d \text{ for simplicity)} \\
&= \frac{1}{2}\frac{\partial}{\partial w_{kj}}(t_k-o_k)^2 \\
&= (t_k-o_k)\frac{\partial(-o_k)}{\partial w_{kj}} \\
&= -(t_k-o_k)\frac{\partial o_k}{\partial net_k}\cdot\frac{\partial net_k}{\partial w_{kj}} \\
&= -(t_k-o_k)o_k(1-o_k)\cdot u_j \quad (u_j \text{ is the output of unit } j) \\
&= -\delta_k u_j
\end{aligned}
$$

- where $\delta_k = (t_k-o_k)o_k(1-o_k) = -\frac{\partial E_d}{\partial net_k}$

Error Gradient of Weights to Hidden Unit j

- w_{ji}: weight for the link from unit i to (hidden) unit j

$$
\begin{aligned}
\frac{\partial E_d}{\partial w_{ji}} &= \frac{1}{2}\frac{\partial}{\partial w_{ji}}\sum_{k=1}^{N_o}(t_{d,k} - o_{d,k})^2 \quad \text{(dropping } d \text{ for simplicity)} \\
&= \sum_{k=1}^{N_o}(t_k - o_k)\frac{\partial(-o_k)}{\partial w_{ji}} \\
&= -\sum_{k=1}^{N_o}(t_k - o_k)\frac{\partial o_k}{\partial net_k}\cdot\frac{\partial net_k}{\partial w_{ji}} \\
&= -\sum_{k=1}^{N_o}(t_k - o_k)\frac{\partial o_k}{\partial net_k}\cdot\frac{\partial net_k}{\partial u_j}\cdot\frac{\partial u_j}{\partial w_{ji}} \\
&= -\sum_{k=1}^{N_o}(t_k - o_k)o_k(1 - o_k)\cdot w_{kj}\cdot\frac{\partial u_j}{\partial net_j}\cdot\frac{\partial net_j}{\partial w_{ji}}
\end{aligned}
$$

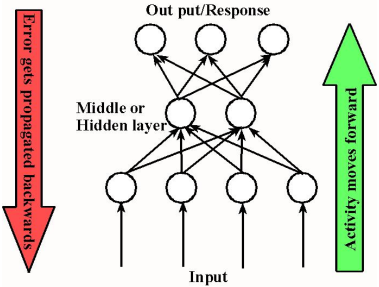
Error Gradient of Weights to Hidden Unit j...

$$\begin{aligned}
\frac{\partial E_d}{\partial w_{ji}} &= -\sum_{k=1}^{N_o}(t_k - o_k)o_k(1-o_k)\cdot w_{kj}\cdot\frac{\partial u_j}{\partial net_j}\cdot\frac{\partial net_j}{\partial w_{ji}} \\
&= -\sum_{k=1}^{N_o}(t_k - o_k)o_k(1-o_k)\cdot w_{kj}\cdot u_j(1-u_j)\cdot u_i \\
&= -\left[\sum_{k=1}^{N_o}\delta_k w_{kj}\right]\cdot u_j(1-u_j)\cdot u_i \\
&= -\delta_j u_i
\end{aligned}$$

- where $\delta_j = u_j(1-u_j)\left[\sum_{k=1}^{N_o}\delta_k w_{kj}\right] = -\frac{\partial E_d}{\partial net_j}$
- note that i may be an input unit. In that case, u_i is just x_i

Weight Update Rule

- output weight: $\Delta w_{kj} = \eta \delta_k u_j$
  $$\delta_k = o_k(1 - o_k)(t_k - o_k)$$
- hidden weight: $\Delta w_{ji} = \eta \delta_j u_i$
  $$\delta_j = u_j(1 - u_j)\left[\sum_{k=1}^{N_o} \delta_k w_{kj}\right]$$



- we need to "propagate error back" when computing the gradient ector

Backpropagation Algorithm (Stochastic Version)

begin
  initialize all weights to small random numbers;
  repeat
    for *each training example* **do**
      /* `propagate input forward` */
      input the example and compute the network outputs;
      /* `propagate errors backward` */
      for *each output unit k* **do** $\delta_k \leftarrow o_k(1 - o_k)(t_k - o_k)$;
      for *each hidden unit j* **do** $\delta_j \leftarrow o_j(1 - o_j) \sum_{k=1}^{N_o} w_{kj} \delta_k$;
      /* `update weights` */
      for *each network weight* w_{ji} *(weight from i to j)* **do**
        $\Delta w_{ji} = \eta \delta_j u_i$;
        $w_{ji} \leftarrow w_{ji} + \Delta w_{ji}$;
      end
    end
  until *convergence* ;
end

Backpropagation Algorithm (Batch Version)

```
begin
    initialize all weights to small random numbers;
    repeat
        for each (i, j) do  initialize each Δw_ji to zero;
        for each training example do
            /* propagate input forward                                  */
            input the example and compute the network outputs;
            /* propagate errors backward                                */
            for each output unit k do  δ_k ← o_k(1 − o_k)(t_k − o_k);
            for each hidden unit j do  δ_j ← o_j(1 − o_j) Σ_{k=1}^{N_o} w_kj δ_k ;
            for each (i, j) do  Δw_ji ← Δw_ji + η δ_j u_i;
        end
        /* update weights                                               */
        for each network weight w_ji (weight from i to j) do
            w_ji ← w_ji + Δw_ji;
        end
    until convergence ;
end
```

begin
- initialize all weights to small random numbers;
- **repeat**
  - **for** *each* (i,j) **do** initialize each Δw_{ji} to zero;
  - **for** *each training example* **do**
    - /* propagate input forward */
    - input the example and compute the network outputs;
    - /* propagate errors backward */
    - **for** *each output unit* k **do** $\delta_k \leftarrow o_k(1-o_k)(t_k-o_k)$;
    - **for** *each hidden unit* j **do** $\boxed{\delta_j} \leftarrow o_j(1-o_j)\sum_{k=1}^{N_o} w_{kj} \boxed{\delta_k}$;
    - **for** *each* (i,j) **do** $\Delta w_{ji} \leftarrow \Delta w_{ji} + \eta \delta_j u_i$;
  - **end**
  - /* update weights */
  - **for** *each network weight* w_{ji} *(weight from* i *to* j*)* **do**
    - $w_{ji} \leftarrow w_{ji} + \Delta w_{ji}$;
  - **end**
- **until** *convergence* ;

end

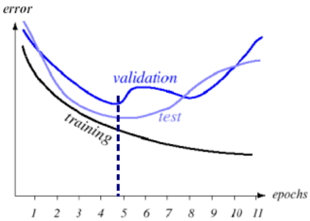
Practical Details

how to initialize the weight values?

- initialize to some small random values

when to stop training?

1. after a fixed number of iterations through the loop
2. once the training error falls below some threshold
3. stop at a minimum of the error on the validation set



Efficiency of Backprop

- let W be the total number of weights (and biases)
- a single evaluation of the error function takes about $O(W)$ operations
  - number of weights typically much greater than the number of units
  - most of the time is on evaluating the sums, with the evaluation of the transfer functions a small overhead
  - each term in the sum reqires one multiplication and one addition

compare with the naive method of numerical differentiation (finite difference)

$$\frac{\partial E}{\partial w_i} = \frac{E(w_i + \epsilon) - E(w_i)}{\epsilon}$$

- takes $O(W^2)$ operations

Speed

- training can be very slow in networks with multiple hidden layers
- testing is fast

How to Speed Up BP Training?

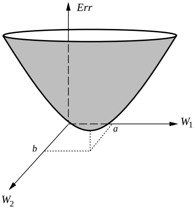
1. use of momentum term
   - give each weight some inertia or momentum

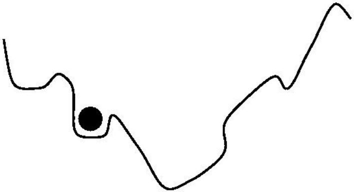
$$\Delta w_{ji}(t+1) = -\eta \frac{\partial E}{\partial w_{ji}} + \alpha \Delta w_{ji}(t)$$

   - $0 < \alpha < 1$: momentum parameter (e.g., $\alpha = 0.9$)
2. dynamic adapt η
3. higher-order information of error surface
4. more sophisticated optimization algorithms

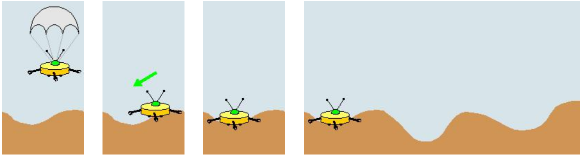
Local Minima

The error surface can have multiple local minima



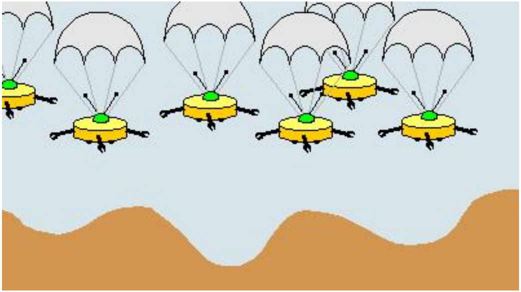


Gradient descent is only guaranteed to converge toward some local minimum, and not necessarily to the global minimum



Local Minima...

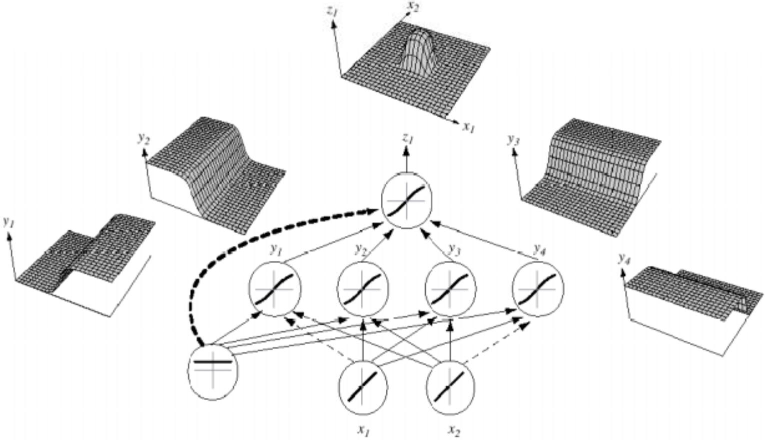
how to escape from locally optimal solutions?



- train multiple networks using the same data, but initialize each network with different random weights

Universal Approximation

only one layer of sigmoid hidden units suffices to approximate any well-behaved function to arbitrary precision



Universal Approximation...

network with > 2 layers also have universal approximation property

why need networks with > 2 hidden layers?

- by using extra layers we might find a network with fewer weights in total while still achieving the same level of accuracy