Neural Networks

Chia-Ping Chen

Professor
National Sun Yat-sen University
Department of Computer Science and Engineering

Machine Learning

Topics

- Feed-forward Networks
- Network Training
- Error Backpropagation
- Hessian Matrix
- Regularization in Neural Networks
- Mixture Density Networks
- Bayesian Neural Networks

Feed-forward Network Functions

In our discussion of linear models for regression or classification, we use output functions based on **fixed basis functions**. That is

$$y_k(\boldsymbol{x}) = f\left(\boldsymbol{w}_k^T \boldsymbol{\phi}\right)$$

where

$$\boldsymbol{\phi} = \begin{bmatrix} \phi_1(\boldsymbol{x}) \\ \vdots \\ \phi_M(\boldsymbol{x}) \end{bmatrix}$$

Instead of fixed basis functions, we can assume output functions

$$y_k(\boldsymbol{x}) = f\left(\boldsymbol{w}_k^T \boldsymbol{\phi}\right)$$

based on **parametric basis functions**

$$\boldsymbol{\phi} = \begin{bmatrix} \phi_1(\boldsymbol{x}, \boldsymbol{\theta}_1) \\ \vdots \\ \phi_M(\boldsymbol{x}, \boldsymbol{\theta}_M) \end{bmatrix}$$

The parameters $\boldsymbol{\theta}_1, \ldots, \boldsymbol{\theta}_M$ in the basis functions, as well as the parameters $\boldsymbol{w}_1, \ldots, \boldsymbol{w}_K$ in the output functions, can be learned from data.

Consider a neural network with 3 layers of units. (By the way, this is called a 2-layer network as there are 2 layers of weights.)

- Input layer

$$\boldsymbol{x} = (x_1, \ldots, x_D)$$

- Hidden layer

$$\boldsymbol{z} = (z_1, \ldots, z_M)$$

- Output layer

$$\boldsymbol{y} = (y_1, \ldots, y_K)$$

- linear combination of input units

$$a_j = \sum_{j=1}^{D} w_{ji} x_i + w_{j0}$$

- hidden-layer activation function

$$z_j = h(a_j)$$

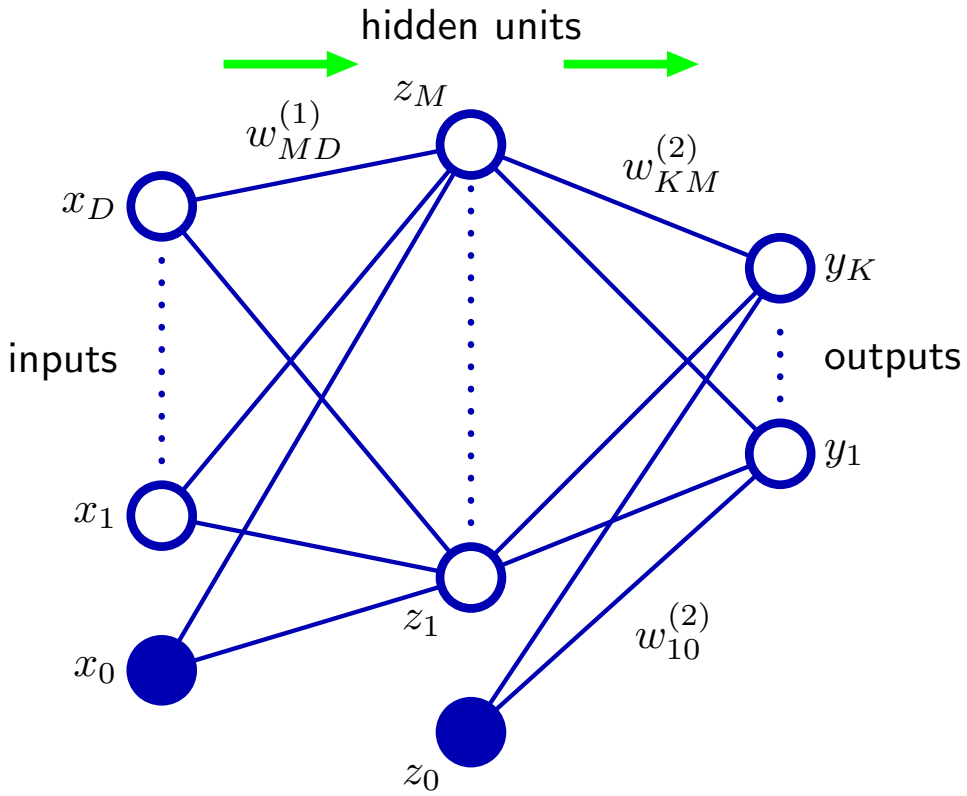
- linear combination of hidden units

$$a_k = \sum_{j=1}^{M} w_{kj} z_j + w_{k0}$$

- output-layer activation function

$$y_k = f(a_k)$$

Forward Propagation



Each hidden-layer unit corresponds to a basis function

$$\phi_j(\boldsymbol{x}) = h\left(\sum_{i=0}^{D} w_{ji}^{(1)} x_i\right)$$

Each output-layer unit corresponds to an output function acting on linear activations

$$\begin{aligned} y_k(\boldsymbol{x}) &= f\left(\sum_j w_{kj}^{(2)} \phi_j(\boldsymbol{x})\right) \\ &= f(\boldsymbol{w}_k^T \boldsymbol{\phi}) \end{aligned}$$

Output Functions

Let $\boldsymbol{a}$ be the activations of the output-layer units.

- Regression: linear output activation function

$$y_k = a_k = \sum_{j=0}^{M} w_{kj}^{(2)} h \left(\sum_{i=0}^{D} w_{ji}^{(1)} x_i \right)$$

- Binary classification: logistic sigmoid activation function

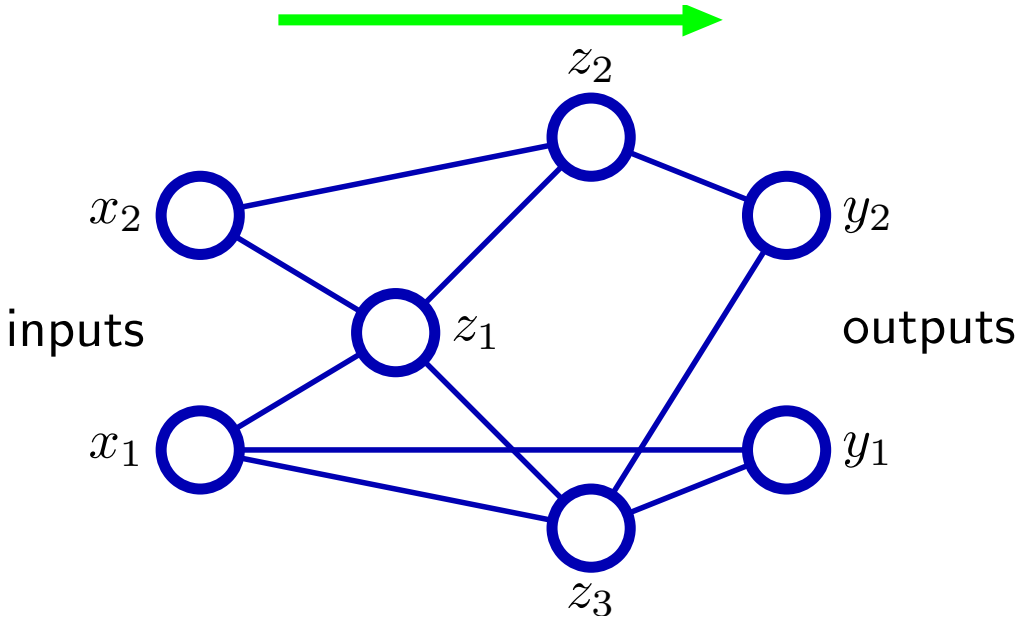
$$y_k = \sigma(a_k) = \sigma \left(\sum_{j=0}^{M} w_{kj}^{(2)} h \left(\sum_{i=0}^{D} w_{ji}^{(1)} x_i \right) \right)$$

- K-ary classification: normalized exponential

$$y_k = \frac{\exp(a_k)}{\sum_{k'=1}^{K} \exp(a'_k)}$$

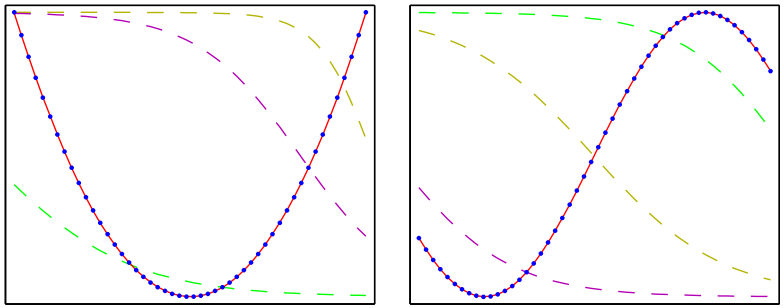
Feed-Forward Networks

In a **feed-forward net**, there are no cycles in the propagation of information from input to output.

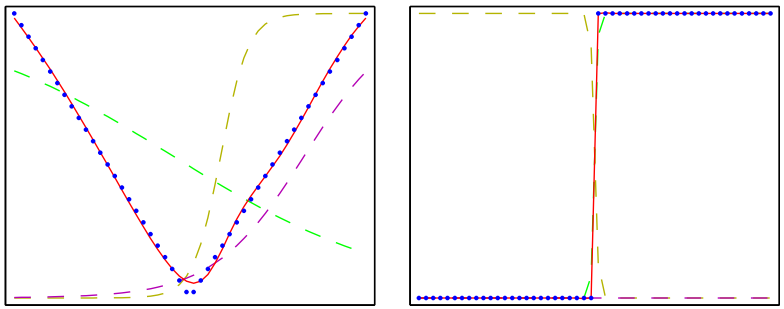


NN as Function Approximator (Regression)

4 neural networks with 1 input-layer unit, 3 hidden-layer units ($\tanh$ activation), and 1 output-layer unit (linear activation). Each network is learned with 50 data points.



x^2 $\sin x$

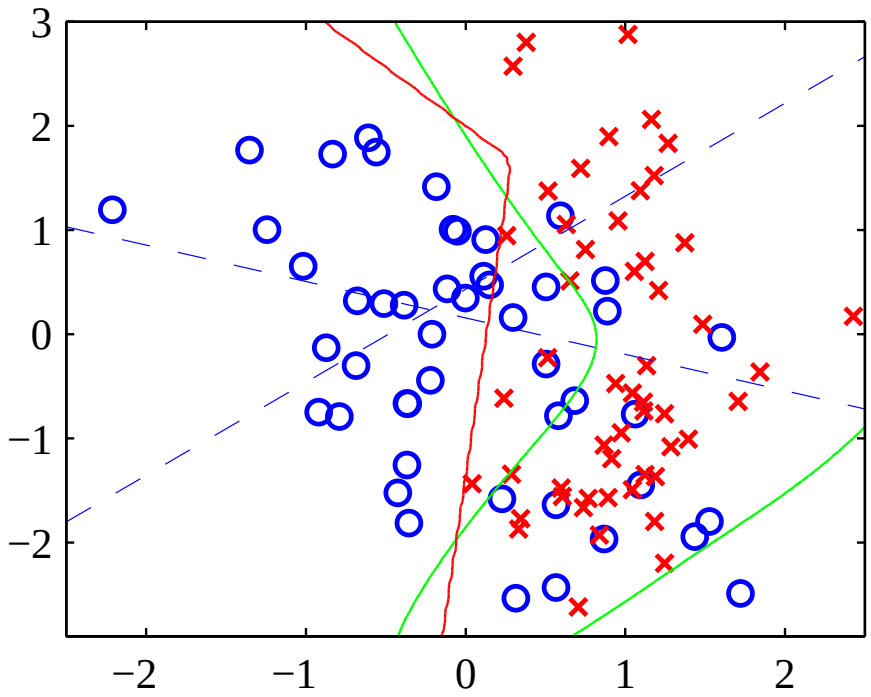


$|x|$ $H(x)$

NN in Binary Classification

Binary classification with a synthetic data set.

- green: decision boundary based on data-generation distribution
- red: decision boundary ($y = 0.5$) based on a neural net
- dashed blue: contours of $z_j = 0.5$



Network Training

- Again, we assume the target variable is corrupted by a Gaussian noise

$$t = u(\boldsymbol{x}) + \epsilon$$

- Further, we assume a parametric function for $u(\boldsymbol{x})$

$$u(\boldsymbol{x}) \approx y(\boldsymbol{x}, \boldsymbol{w})$$

- For a data point $(\boldsymbol{x}, t)$, the conditional likelihood is

$$p(t|\boldsymbol{x}) = \mathcal{N}(t|y(\boldsymbol{x}, \boldsymbol{w}), \beta^{-1})$$

- The likelihood of a data set $\{(\boldsymbol{x}_1, t_1), \ldots, (\boldsymbol{x}_N, t_N)\}$ is

$$\prod_{n=1}^{N} p(t_n|\boldsymbol{x}_n) = \prod_{n=1}^{N} \mathcal{N}(t_n|y(\boldsymbol{x}_n, \boldsymbol{w}), \beta^{-1})$$

Taking the negative logarithm of the data likelihood, we obtain

$$\frac{\beta}{2}\sum_{n=1}^{N}(y(\boldsymbol{x}_n,\boldsymbol{w})-t_n)^2-\frac{N}{2}\log\beta+\frac{N}{2}\log(2\pi)$$

Maximizing data likelihood with respect to $\boldsymbol{w}$ is equivalent to minimizing **the sum of squared errors**

$$\boldsymbol{w}_{\mathsf{ML}}=\arg\min_{\boldsymbol{w}}E(\boldsymbol{w})=\arg\min_{\boldsymbol{w}}\frac{1}{2}\sum_{n=1}^{N}(y(\boldsymbol{x}_n,\boldsymbol{w})-t_n)^2$$

Maximizing data likelihood with respect to β leads to

$$\frac{1}{\beta_{\mathsf{ML}}}=\frac{1}{N}\sum_{n=1}^{N}(y(\boldsymbol{x}_n,\boldsymbol{w}_{\mathsf{ML}})-t_n)^2$$

Binary Classification

- Again, we assume a parametric function for the posterior probability of class $\mathcal{C}_1$

$$p(\mathcal{C}_1|\boldsymbol{x}) \approx y(\boldsymbol{x}, \boldsymbol{w})$$

- By using $t = 1$ for class $\mathcal{C}_1$ and $t = 0$ for class $\mathcal{C}_2$, the likelihood of a data point $(\boldsymbol{x}, t)$ can be written as

$$p(t|\boldsymbol{x}) = y(\boldsymbol{x}, \boldsymbol{w})^t (1 - y(\boldsymbol{x}, \boldsymbol{w}))^{1-t}$$

- The likelihood of a data set $\{(\boldsymbol{x}_1, t_1), \ldots, (\boldsymbol{x}_N, t_N)\}$ is

$$\prod_{n=1}^{N} p(t_n|\boldsymbol{x}_n) = \prod_{n=1}^{N} y(\boldsymbol{x}_n, \boldsymbol{w})^t (1 - y(\boldsymbol{x}_n, \boldsymbol{w}))^{1-t}$$

- Maximizing data likelihood is equivalent to minimizing the sum of cross-entropy errors

$$E(\boldsymbol{w}) = -\sum_{n=1}^{N} t_n \log y_n + (1 - t_n) \log(1 - y_n)$$

K-ary Classification

- Again, we assume a parametric function for the class posteriors

$$p(\mathcal{C}_k|\boldsymbol{x}) \approx y_k(\boldsymbol{x}, \boldsymbol{w})$$

- By using 1-of-K target vectors, the likelihood of a data point $(\boldsymbol{x}, \boldsymbol{t})$ can be written as

$$p(\boldsymbol{t}|\boldsymbol{x}) = \prod_{k=1}^{K} y_k(\boldsymbol{x}, \boldsymbol{w})^{t_k}$$

- The likelihood of a data set $\{(\boldsymbol{x}_1, \boldsymbol{t}_1), \ldots, (\boldsymbol{x}_N, \boldsymbol{t}_N)\}$ is

$$\prod_{n=1}^{N} p(\boldsymbol{t}_n|\boldsymbol{x}_n) = \prod_{n=1}^{N} \prod_{k=1}^{K} y_k(\boldsymbol{x}_n, \boldsymbol{w})^{t_{nk}}$$

- Maximizing data likelihood is equivalent to minimizing the cross-entropy error function

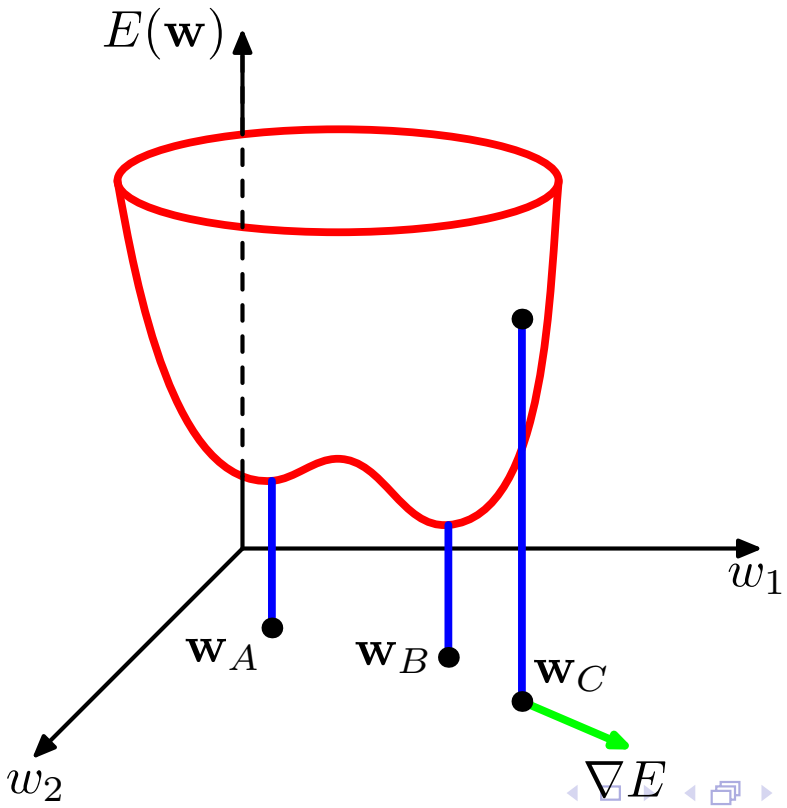
$$E(\boldsymbol{w}) = -\sum_{n=1}^{N} \sum_{k=1}^{K} t_{nk} \log y_{nk}, \;\; y_{nk} = y_k(\boldsymbol{x}_n, \boldsymbol{w})$$

The **local linear approximation** near a point $\boldsymbol{w}_0$ is

$$E(\boldsymbol{w}) \approx E(\boldsymbol{w}_0) + (\boldsymbol{w} - \boldsymbol{w}_0)^T \boldsymbol{\nabla} E(\boldsymbol{w}_0)$$

where

$$\boldsymbol{\nabla} E(\boldsymbol{w}) = \begin{bmatrix} E_{w_1} \\ E_{w_2} \end{bmatrix}$$



Gradient descent

$$\boldsymbol{w}^{(\tau+1)} = \boldsymbol{w}^{(\tau)} - \eta \boldsymbol{\nabla} E(\boldsymbol{w}^{(\tau)})$$

Stochastic gradient descent

$$\boldsymbol{w}^{(\tau+1)} = \boldsymbol{w}^{(\tau)} - \eta \boldsymbol{\nabla} E_n(\boldsymbol{w}^{(\tau)})$$

Iterative methods: update weight and re-evaluate gradient.

The **local quadratic approximation** near a point $\boldsymbol{w}_0$ is

$$E(\boldsymbol{w}) \approx E(\boldsymbol{w}_0) + (\boldsymbol{w} - \boldsymbol{w}_0)^T \boldsymbol{\nabla} E(\boldsymbol{w}_0) + \frac{1}{2}(\boldsymbol{w} - \boldsymbol{w}_0)^T \boldsymbol{H} (\boldsymbol{w} - \boldsymbol{w}_0)$$

where $\boldsymbol{H}$ is the Hessian matrix at $\boldsymbol{w}_0$ with

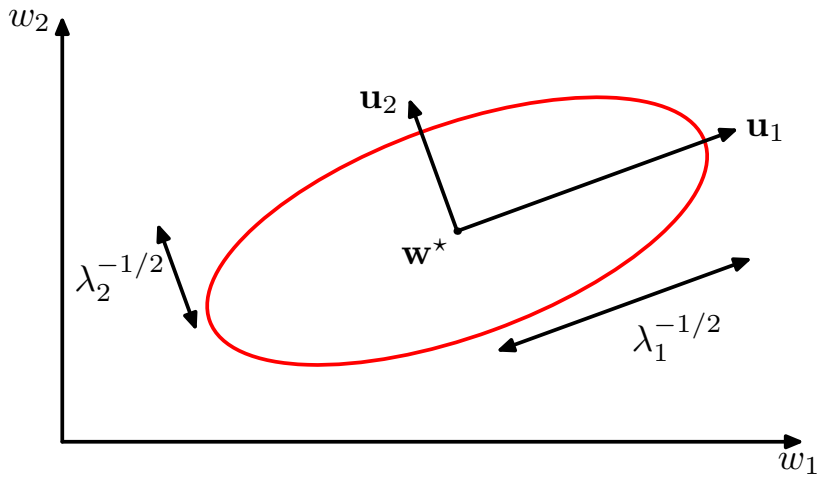
$$(\boldsymbol{H})_{ij} = E_{w_i w_j}(\boldsymbol{w}_0)$$

The critical point of the local quadratic approximation near $\boldsymbol{w}_0$ is

$$\boldsymbol{w}' = \boldsymbol{w}_0 - \boldsymbol{H}^{-1}\nabla E(\boldsymbol{w}_0)$$

This equation can be applied iteratively to update $\boldsymbol{w}$.

It may converge to a local minimum where ∇E vanishes and $\boldsymbol{H}$ is positive definite.



Error Backpropagation

In the **error backpropagation**, the gradient is computed by propagating the derivatives **backwards** through the network, starting from the **errors** between targets and outputs.

- We begin with the derivatives of the cost function with respect to the **output-layer unit activations**.
- The derivatives of the cost function with respect to the **hidden-layer unit activations** are computed by propagating information backwards in the network.
- The derivatives of the cost function with respect to the **link weights** are computed from the derivatives with respect to the unit activations.

The activation at a unit j is a weighted sum of its inputs

$$a_j = \sum_i w_{ji} z_i$$

Here w_{ji} is the weight of the link connecting unit i and unit j. Moreover, unit i is the input of this link and unit j is the output of this link. An activation function transforms the activation

$$z_j = h(a_j)$$

for propagation of information to the next level.

Let $E(\boldsymbol{w})$ be the cost function to be minimized.

- Output-layer units: for each unit k, define

$$\delta_k = \frac{\partial E}{\partial a_k}$$

- Hidden-layer units: for each unit j, define

$$\delta_j = \frac{\partial E}{\partial a_j}$$

- Link weights: for the link from unit i to unit j, we have

$$\frac{\partial E}{\partial w_{ji}} = \frac{\partial E}{\partial a_j}\frac{\partial a_j}{\partial w_{ji}} = \delta_j z_i$$

The derivatives at the hidden-layer units are related to the derivatives at the output-layer units by

$$\delta_j = h'(a_j) \sum_k w_{kj} \delta_k$$

Note

$$\frac{\partial a_k}{\partial a_j} = \frac{\partial a_k}{\partial z_j} \frac{\partial z_j}{\partial a_j} = w_{kj} h'(a_j)$$

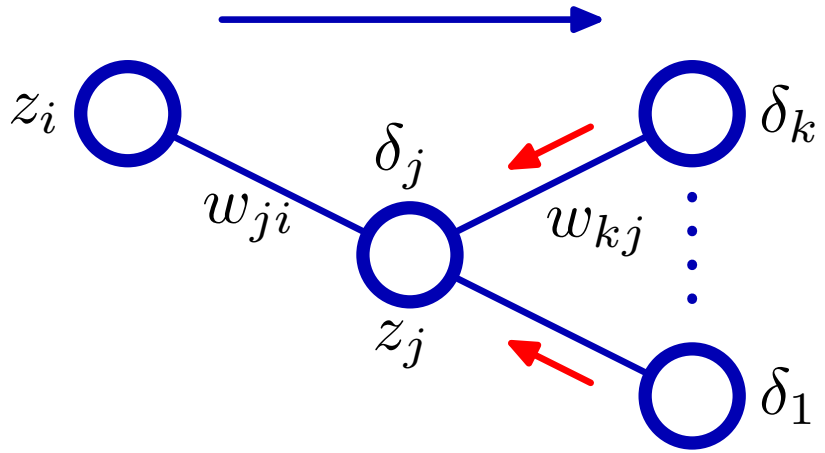
It follows that

$$\delta_j = \frac{\partial E}{\partial a_j} = \sum_k \frac{\partial E}{\partial a_k} \frac{\partial a_k}{\partial a_j} = \sum_k \delta_k w_{kj} h'(a_j) = h'(a_j) \sum_k w_{kj} \delta_k$$

The derivative at a hidden-layer unit is a linear combination of the derivatives at the output-layer units.

$$\delta_j = \overbrace{h'(a_j)}^{\text{scalar multiplication}} \overbrace{\sum_k w_{kj}\delta_k}^{\text{backward activation}}$$

δ_j can be interpreted as a backward output at unit j with the δ_k's as inputs followed by a scalar multiplication of $h'(a_j)$.



The Derivatives at the Output Layer

By design, the derivative at an output layer unit is the error between output and target, i.e.

$$\delta_k = y_k - t_k$$

- Regression: sum of squared errors cost function and linear activation function

$$\delta_k = \frac{\partial E}{\partial a_k} = \frac{\partial E}{\partial y_k}\frac{\partial y_k}{\partial a_k} = y_k - t_k$$

- Binary classification: cross-entropy cost function and logistic sigmoid activation function

$$\delta_k = \frac{\partial E}{\partial y_k}\frac{\partial y_k}{\partial a_k} = -\left(\frac{t_k}{y_k} - \frac{1-t_k}{1-y_k}\right)(y_k(1-y_k)) = y_k - t_k$$

- K-ary classification: cross-entropy cost function and normalized exponential activation function (next slide)

$$E = -\sum_k t_k \log y_k$$

$$\begin{aligned}
\delta_k &= \frac{\partial E}{\partial a_k} \\
&= \sum_{k'} \frac{\partial E}{\partial y_{k'}} \frac{\partial y_{k'}}{\partial a_k} \\
&= \sum_{k'} \left(-\frac{t_{k'}}{y_{k'}} \right) \left(y_{k'} (\delta_{k'k} - y_k) \right) \\
&= \sum_{k'} -t_{k'} (\delta_{k'k} - y_k) \\
&= \sum_{k'} y_k t_{k'} - \sum_{k'} t_{k'} \delta_{k'k} \\
&= y_k - t_k
\end{aligned}$$

Error Backpropagation

- Forward propagation for

$$a_j, z_j, a_k, y_k$$

- Derivatives of cost function at the output layer (= errors)

$$\delta_k = y_k - t_k$$

- Backpropagation of the derivatives (to the hidden layer)

$$\delta_j = h'(a_j) \sum_k \delta_k w_{kj}$$

- Derivatives with respect to the weights

$$\frac{\partial E}{\partial w_{kj}} = \delta_k z_j, \quad \frac{\partial E}{\partial w_{ji}} = \delta_j x_i$$

A Simple Example

- Output-layer activation function

$$y_k = a_k$$

- Hidden-layer activation function

$$h(a_j) = \tanh(a_j) = \frac{e^{a_j} - e^{-a_j}}{e^{a_j} + e^{-a_j}}$$

Note

$$h'(a_j) = 1 - h^2(a_j) = 1 - z_j^2$$

- Sum of squared errors cost function

$$E(\boldsymbol{w}) = \frac{1}{2}\sum_{k=1}^{K}(y_k - t_k)^2$$

- Forward propagation

$$a_j = \sum_{i=0}^{D} w_{ji} x_i, \;\; z_j = \tanh(a_j), \;\; a_k = \sum_{j=0}^{M} w_{kj} z_j, \;\; y_k = a_k$$

- Derivatives at the output units (= errors)

$$\delta_k = y_k - t_k$$

- Derivatives at the hidden units

$$\delta_j = h'(a_j) \sum_k w_{kj} \delta_k = (1 - z_j^2) \sum_{k=1}^{K} w_{kj} \delta_k$$

- The weight derivatives

$$\frac{\partial E}{\partial w_{ji}} = \delta_j x_i, \qquad \frac{\partial E}{\partial w_{kj}} = \delta_k z_j$$

The **Jacobian** consists of the derivatives of a group of dependent variables with respect to a group of input variables.

Let $\boldsymbol{x} = (x_1, \ldots, x_D)^T$ be the input variables and $\boldsymbol{y} = (y_1, \ldots, y_K)^T$ be the output variables. Note $y_i = y_i(x_1, \ldots, x_D)$. The Jacobian is

$$\boldsymbol{J} = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_1}{\partial x_D} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_K}{\partial x_1} & \cdots & \frac{\partial y_K}{\partial x_D} \end{bmatrix}$$

That is, $J_{ki} = \frac{\partial y_k}{\partial x_i}$.

The Jacobian of a feed-forward network function can be computed with **backpropagation**.

Let $\boldsymbol{x}$ be the input and $\boldsymbol{y}$ be the output. We want $J_{ki} = \frac{\partial y_k}{\partial x_i}$.

- Begin with the output-layer derivatives

$$\frac{\partial y_k}{\partial a_{k'}} = \delta_{kk'} f'(a_{k'}) \text{ or } \frac{\partial y_k}{\partial a_{k'}} = \delta_{kk'} y_k - y_k y_{k'}$$

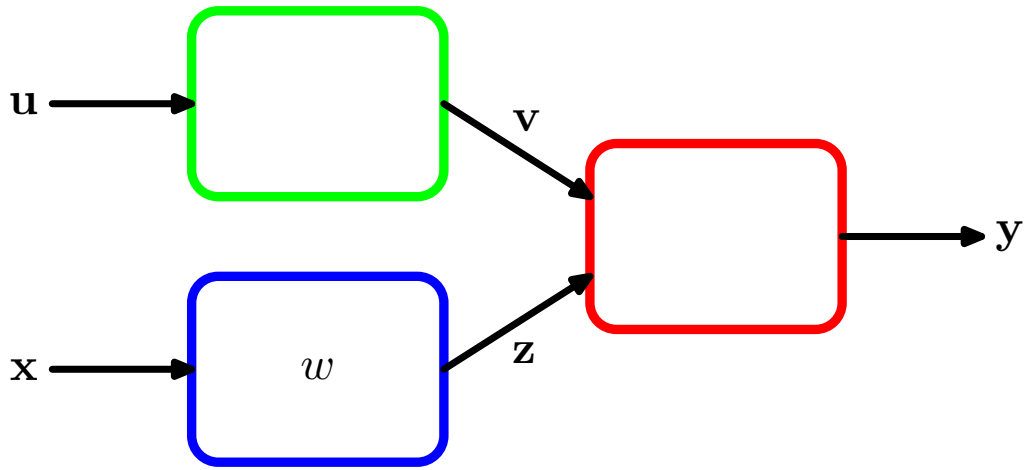
- Backpropagate the output-layer derivatives to the hidden-layer

$$\frac{\partial y_k}{\partial a_j} = \sum_{k'} \frac{\partial y_k}{\partial a_{k'}} \frac{\partial a_{k'}}{\partial a_j} = \sum_{k'} \frac{\partial y_k}{\partial a_{k'}} w_{k'j} h'(a_j)$$

- End with

$$J_{ki} = \frac{\partial y_k}{\partial x_i} = \sum_j \frac{\partial y_k}{\partial a_j} \frac{\partial a_j}{\partial x_i} = \sum_j \frac{\partial y_k}{\partial a_j} w_{ji}$$

Jacobian is convenient in a system built from distinct modules. For example, the derivative of the cost function with respect to w in



involves a Jacobian

$$\frac{\partial E}{\partial w} = \sum_{k,j} \frac{\partial E}{\partial y_k} \frac{\partial y_k}{\partial z_j} \frac{\partial z_j}{\partial w}$$

Hessian

The **Hessian** of a function consists of the second-order partial derivatives of the function with respect to the variables.

Let $\boldsymbol{w} = (w_1, \ldots, w_W)^T$ be the trainable parameters (weights and biases) of a neural network and $E(\boldsymbol{w})$ be the cost function. The Hessian is

$$\boldsymbol{H} = \begin{bmatrix} E_{w_1 w_1} & \cdots & E_{w_1 w_W} \\ \vdots & \ddots & \vdots \\ E_{w_W w_1} & \cdots & E_{w_W w_W} \end{bmatrix}$$

where

$$E_{w_m w_n} = \frac{\partial}{\partial w_n}\left(\frac{\partial E}{\partial w_m}\right) = \frac{\partial^2 E}{\partial w_n \partial w_m} = \frac{\partial^2 E}{\partial w_m \partial w_n} = E_{w_n w_m}$$

Roles Played by Hessian

- Non-linear optimization algorithm for training netwrok
- Fast procedure for re-training network
- To identify least significant weights for pruning network
- Laplace approximation for Bayesian learning of network

Consider a basic neural network with 2 layers of weights.

- Let i and i' index input-layer units, j and j' index hidden-layer units, and k and k' index output-layer units.
- The Hessian matrix has 3 different blocks

$$\frac{\partial^2 E}{\partial w_{kj} \partial w_{k'j'}}, \quad \frac{\partial^2 E}{\partial w_{ji} \partial w_{j'i'}}, \quad \frac{\partial^2 E}{\partial w_{ji} \partial w_{kj'}}$$

Start with output-layer derivatives

$$\delta_k = \frac{\partial E}{\partial a_k}, \; M_{kk'} = \frac{\partial^2 E}{\partial a_k \partial a_{k'}}$$

Backpropagate the derivatives. For the first block

$$\begin{aligned}
\frac{\partial^2 E}{\partial w_{kj} \partial w_{k'j'}} &= \frac{\partial}{\partial w_{kj}} \left(\frac{\partial E}{\partial w_{k'j'}} \right) \\
&= \frac{\partial}{\partial w_{kj}} \left(\frac{\partial E}{\partial a_{k'}} \frac{\partial a_{k'}}{\partial w_{k'j'}} \right) \\
&= z_{j'} \frac{\partial}{\partial w_{kj}} \left(\frac{\partial E}{\partial a_{k'}} \right) \\
&= z_{j'} \frac{\partial a_k}{\partial w_{kj}} \left(\frac{\partial^2 E}{\partial a_k \partial a_{k'}} \right) \\
&= z_{j'} z_j M_{kk'}
\end{aligned}$$

For the off-diagonal block

$$\begin{aligned}
\frac{\partial^2 E}{\partial w_{ji}\partial w_{kj'}} &= \frac{\partial}{\partial w_{ji}}\left(\frac{\partial E}{\partial w_{kj'}}\right) = \frac{\partial}{\partial w_{ji}}\left(\frac{\partial E}{\partial a_k}\frac{\partial a_k}{\partial w_{kj'}}\right) \\
&= \frac{\partial}{\partial w_{ji}}\left(\frac{\partial E}{\partial a_k} z_{j'}\right) \\
&= z_{j'}\frac{\partial}{\partial w_{ji}}\left(\frac{\partial E}{\partial a_k}\right) + \frac{\partial z_{j'}}{\partial w_{ji}}\left(\frac{\partial E}{\partial a_k}\right) \\
&= z_{j'}\sum_{k'}\frac{\partial a_{k'}}{\partial w_{ji}}\left(\frac{\partial^2 E}{\partial a_{k'}\partial a_k}\right) + \frac{\partial z_{j'}}{\partial w_{ji}}\left(\frac{\partial E}{\partial a_k}\right) \\
&= z_{j'}\sum_{k'}\frac{\partial a_{k'}}{\partial z_j}\frac{\partial z_j}{\partial w_{ji}}\left(\frac{\partial^2 E}{\partial a_{k'}\partial a_k}\right) + \frac{\partial z_{j'}}{\partial w_{ji}}\left(\frac{\partial E}{\partial a_k}\right) \\
&= z_{j'}\sum_{k'} w_{k'j} x_i h'(a_j) M_{k'k} + \delta_{jj'} x_i h'(a_j)\delta_k \\
&= x_i h'(a_j)\left(z_{j'}\sum_{k'} w_{k'j} M_{k'k} + \delta_{jj'}\delta_k\right)
\end{aligned}$$

Finally (an exercise)

$$\frac{\partial^2 E}{\partial w_{ji}\partial w_{j'i'}} = x_i x_{i'} h''(a_{j'})\delta_{jj'}\sum_k w_{kj'}\delta_k + x_i x_{i'} h'(a_j)h'(a_{j'})\sum_k\sum_{k'} w_{kj}w_{k'j'}M_{kk'}$$

In diagonal approximation, we replace off-diagonal elements of a Hessian by 0, and retains the diagonal elements. For the weights between the input layer and the hidden layer, the diagonal elements are

$$\frac{\partial^2 E}{\partial w_{ji}^2} = x_i^2 \left(h''(a_j) \sum_k w_{kj} \delta_k + h'(a_j)^2 \sum_k \sum_{k'} w_{kj} w_{k'j} M_{kk'} \right)$$

Further approximation may be applied by neglecting the off-diagonal elements of $\boldsymbol{M}$. We then have

$$\frac{\partial^2 E}{\partial w_{ji}^2} = x_i^2 \left(h''(a_j) \sum_k w_{kj} \delta_k + h'(a_j)^2 \sum_k w_{kj}^2 M_{kk} \right)$$

The Hessian of the sum-of-squared-errors cost function is

$$\begin{aligned}\boldsymbol{H} &= \sum_n \boldsymbol{\nabla} y_n (\boldsymbol{\nabla} y_n)^T + \sum_n (y_n - t_n) \boldsymbol{\nabla}\boldsymbol{\nabla} y_n \\ &\approx \sum_n \boldsymbol{b}_n \boldsymbol{b}_n^T\end{aligned}$$

where

$$\boldsymbol{b}_n = \boldsymbol{\nabla} y_n$$

The Hessian of the cross-entropy cost function is

$$\boldsymbol{H} \approx \sum_n y_n (1 - y_n) \boldsymbol{b}_n \boldsymbol{b}_n^T$$

In both cases, the Hessians are approximated as the sum of outer-product terms.

Inverse Hessian

The outer-product approximation of a Hessian facilitates a procedure for computing its inverse. Define

$$\boldsymbol{H}_L = \sum_{n=1}^{L} \boldsymbol{b}_n \boldsymbol{b}_n^T$$

Then

$$\boldsymbol{H}_{L+1} = \boldsymbol{H}_L + \boldsymbol{b}_{L+1} \boldsymbol{b}_{L+1}^T$$

and

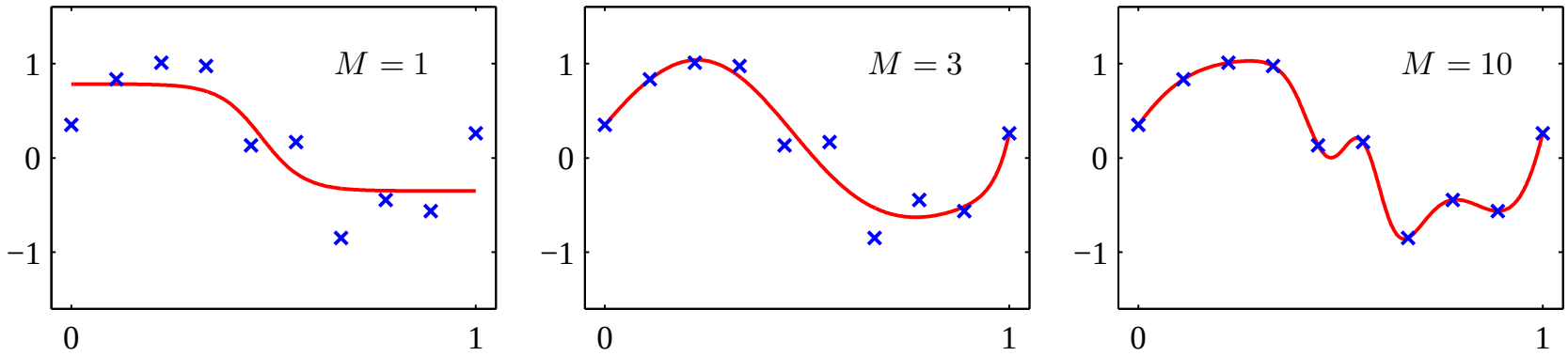
$$\boldsymbol{H}_{L+1}^{-1} = \boldsymbol{H}_L^{-1} - \frac{\boldsymbol{H}_L^{-1} \boldsymbol{b}_{L+1} \boldsymbol{b}_{L+1}^T \boldsymbol{H}_L^{-1}}{1 + \boldsymbol{b}_{L+1}^T \boldsymbol{H}_L^{-1} \boldsymbol{b}_{L+1}}$$

Regularization

Motivation of Regularization

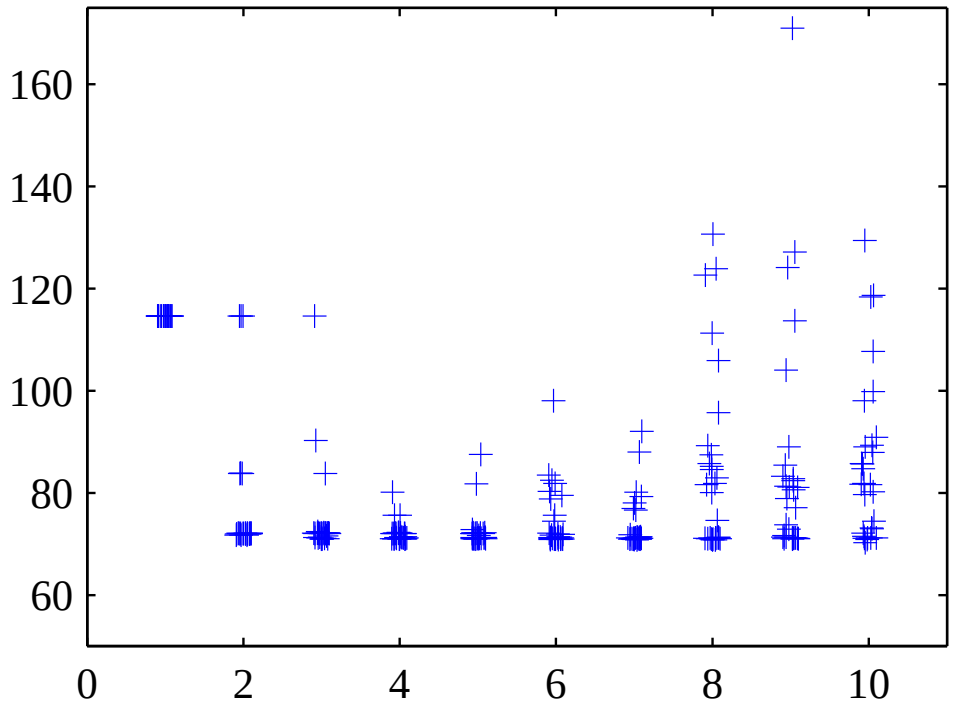
- Avoid over-fitting
- Include terms in the cost function to reduce generalization error
- Reduce the effective number of parameters
- Put model complexity under control

Over-fitting with Neural Networks



Two-layer networks trained on 10 data points with a sum-of-squared-errors cost function. The number of hidden-layer units M is shown.

Effect of Local Minimum



The sum-of-squared errors of polynomial data test set. For each hidden-layer size, 30 random initializations of the parameters, with isotropic zero-mean Gaussian distribution, are used.

In **weight decay**, we use a regularized cost function

$$\tilde{E}(\boldsymbol{w}) = E(\boldsymbol{w}) + \frac{\lambda}{2}\boldsymbol{w}^T\boldsymbol{w}$$

The regularization term (a.k.a. **regularizer**) $\frac{\lambda}{2}\boldsymbol{w}^T\boldsymbol{w}$ can be interpreted as the negative logarithm of a zero-mean Gaussian prior distribution for $\boldsymbol{w}$.

For a basic neural network with 2 layers of weights, a regularizer that is consistent with the invariance to linear transformation of the input/output variables is

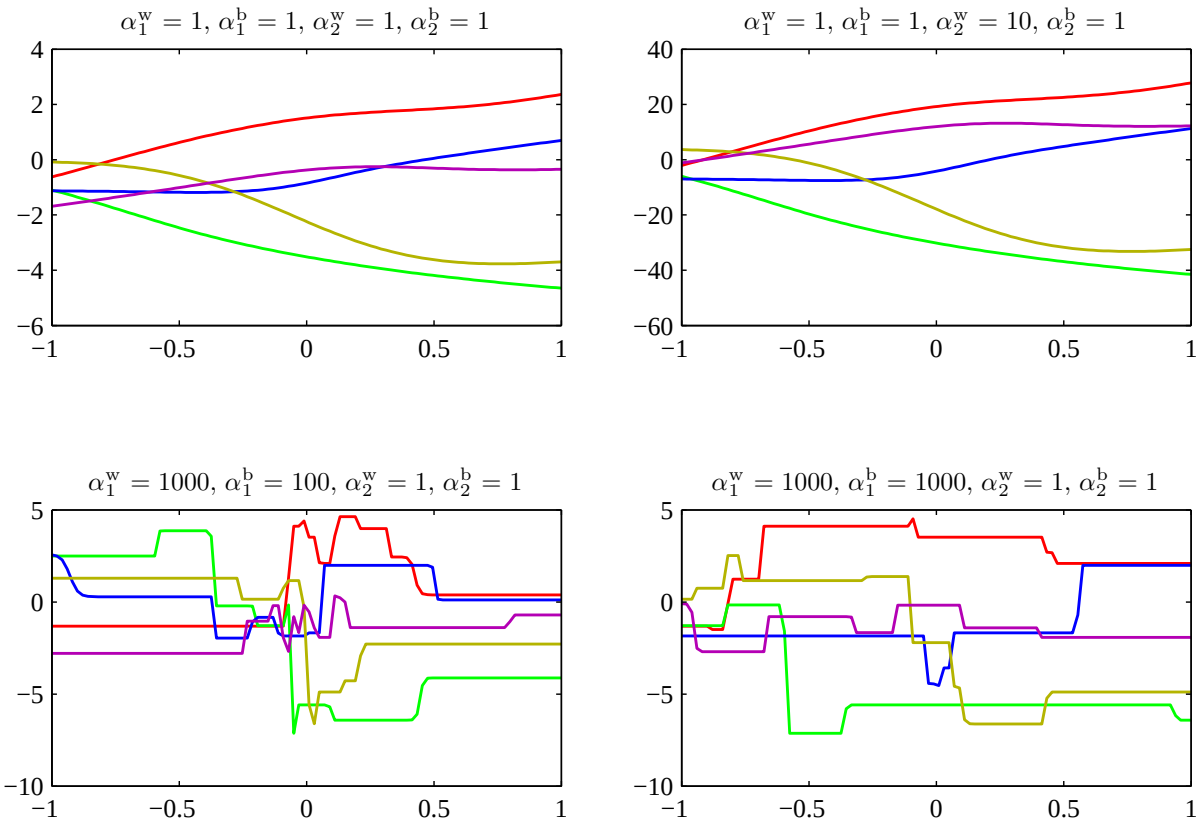
$$\frac{\lambda_1}{2}\sum_{w\in\mathcal{W}_1} w^2 + \frac{\lambda_2}{2}\sum_{w\in\mathcal{W}_2} w^2$$

More generally, we can consider priors

$$p(\boldsymbol{w}|\boldsymbol{\alpha}) \propto \exp\left(-\frac{1}{2}\sum_k \alpha_k \|\boldsymbol{w}\|_k^2\right)$$

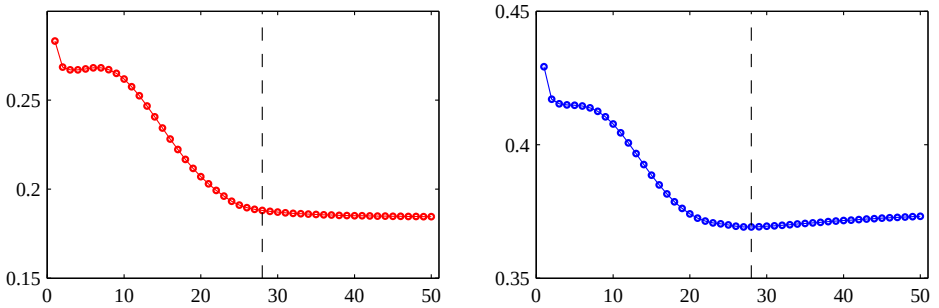
where

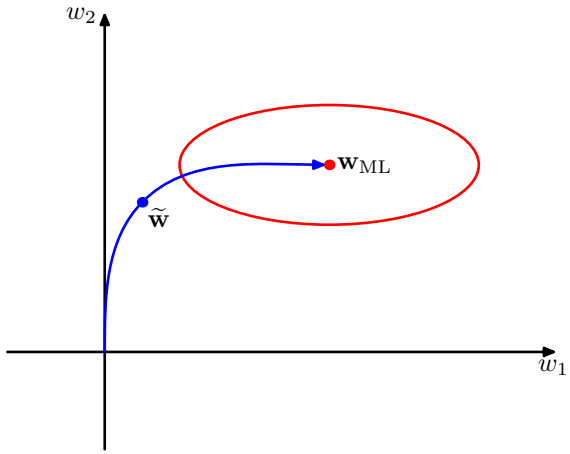
$$\|\boldsymbol{w}\|_k^2 = \sum_{w\in\mathcal{W}_k} w^2$$



Samples of network functions using 4 hyperparameters for the prior distribution $p(\boldsymbol{w}|\boldsymbol{\alpha})$ over the weights and biases in a 2-layer network with 12 hidden units with $\tanh$ activation function.

Early Stopping





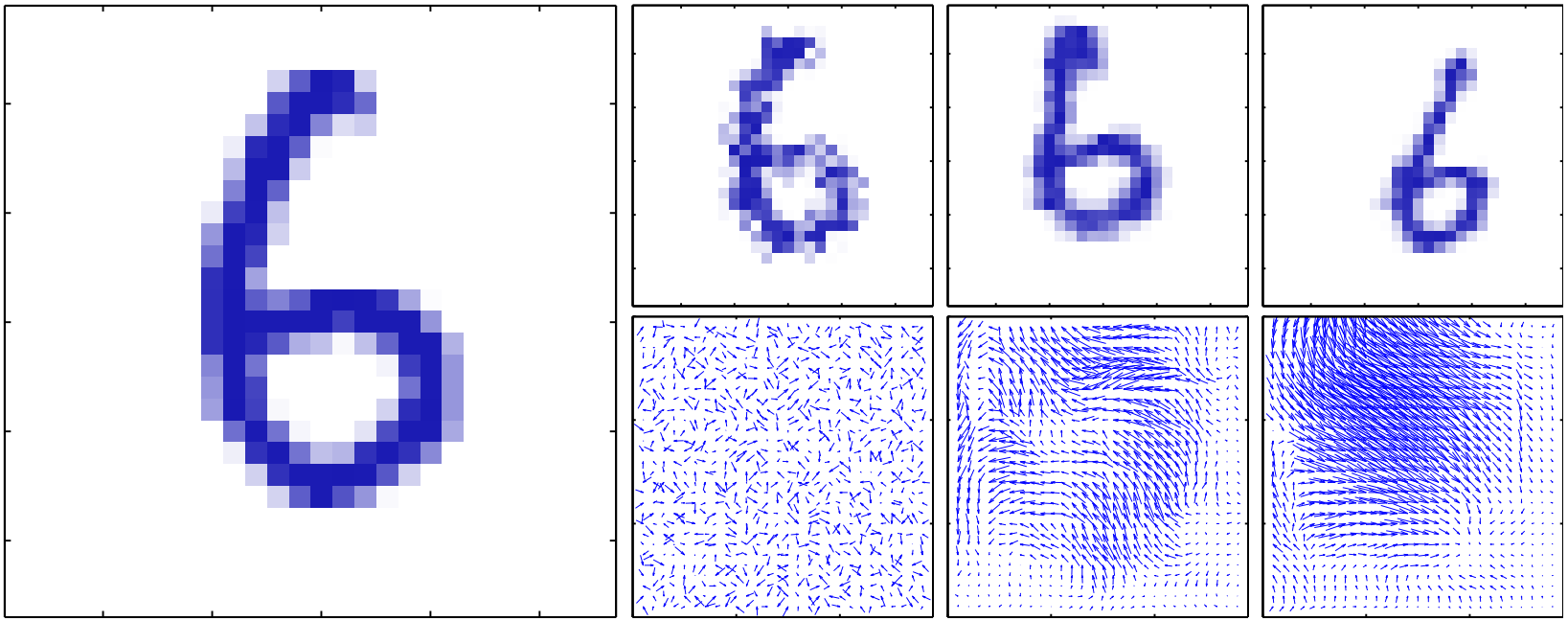
Training set error and validation set error as a function of the iteration step. Early stopping has a similar effect as weight decay.

Invariance

In many classification problems, the prediction should be unchanged under certain transformation of the input.

- collect sufficiently many examples (very expensive)
- use replicas of training data subject to transformation
- add a term to the cost function to penalize output changes
- extract features that are invariant under certain transformation
- build the invariance properties into the model

Data Augmentation



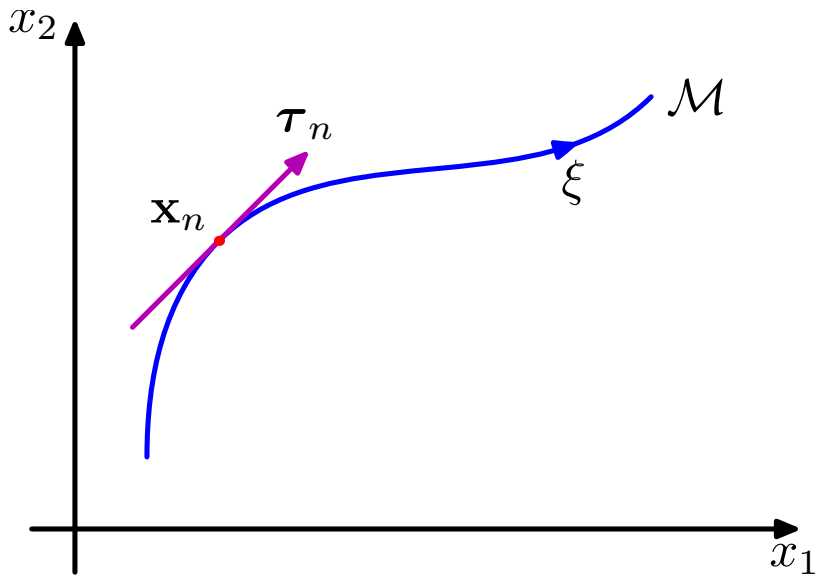
Data augmentation with a hand-written digit.

Tangent Propagation

Consider a continuous transformation starting from point $\boldsymbol{x}$.

- Suppose the transformation is governed by a parameter ξ
- A trajectory $\mathcal{M} : \xi \mapsto \boldsymbol{s}(\boldsymbol{x}_n, \xi)$ with $\boldsymbol{s}(\boldsymbol{x}_n, 0) = \boldsymbol{x}_n$ is traced
- The tangent vector of $\boldsymbol{s}(\boldsymbol{x}_n, \xi)$ at $\boldsymbol{x}_n$ is

$$\boldsymbol{\tau}_n = \left. \frac{\partial \boldsymbol{s}}{\partial \xi} \right|_{\xi=0}$$



Along trajectory $\mathcal{M}$ we have

$$\boldsymbol{x} = \boldsymbol{s}(\boldsymbol{x}_n, \xi)$$

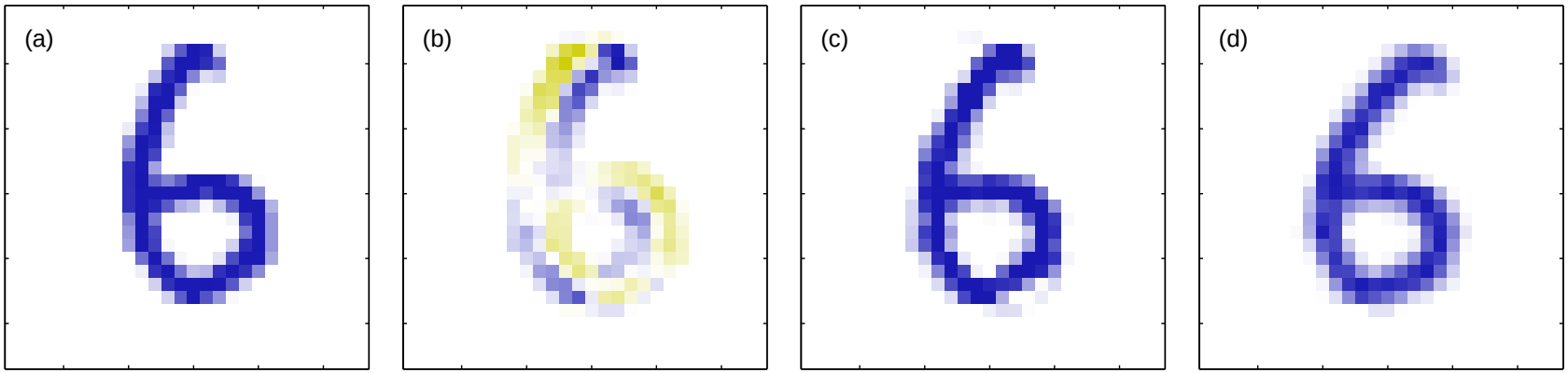
Consider output function $y_k(\boldsymbol{x})$ along trajectory $\mathcal{M}$. At $\boldsymbol{x}_n$

$$\left.\frac{\partial y_k}{\partial \xi}\right|_{\boldsymbol{x}=\boldsymbol{x}_n, \xi=0} = \left.\sum_{i=1}^{D} \frac{\partial y_k}{\partial x_i} \frac{\partial x_i}{\partial \xi}\right|_{\boldsymbol{x}=\boldsymbol{x}_n, \xi=0} = \sum_{i=1}^{D} J_{nki} \tau_{ni}$$

If we want the output to remain invariant with respect to tangent direction of the transformation, we add a term to the cost function

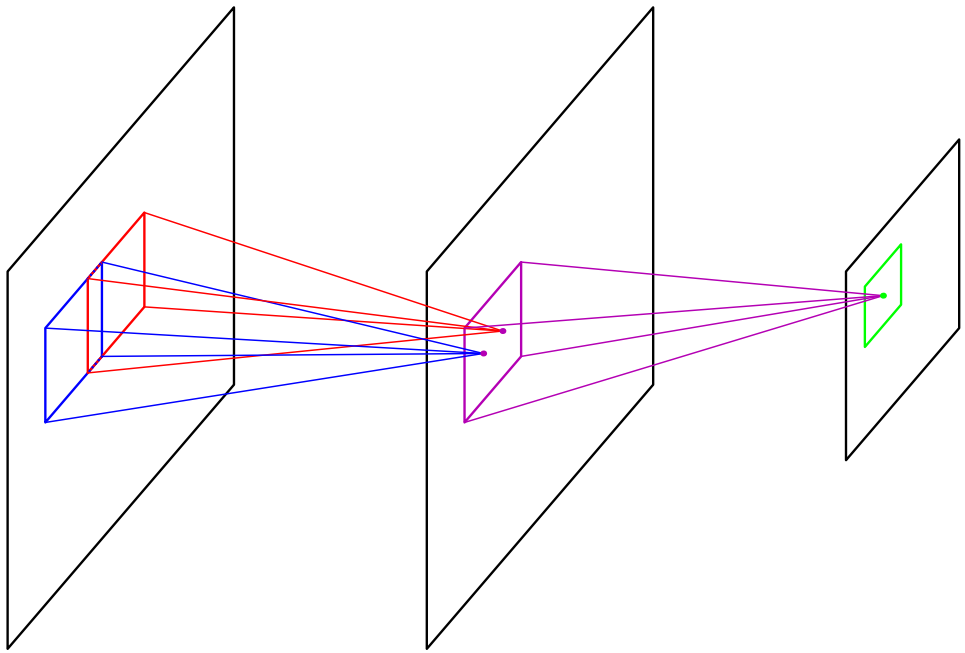
$$\tilde{E} = E + \lambda \sum_n \sum_k \left(\sum_i J_{nki} \tau_{ni} \right)^2$$

The tangent vector of a transformation can be approximated by finite differences.



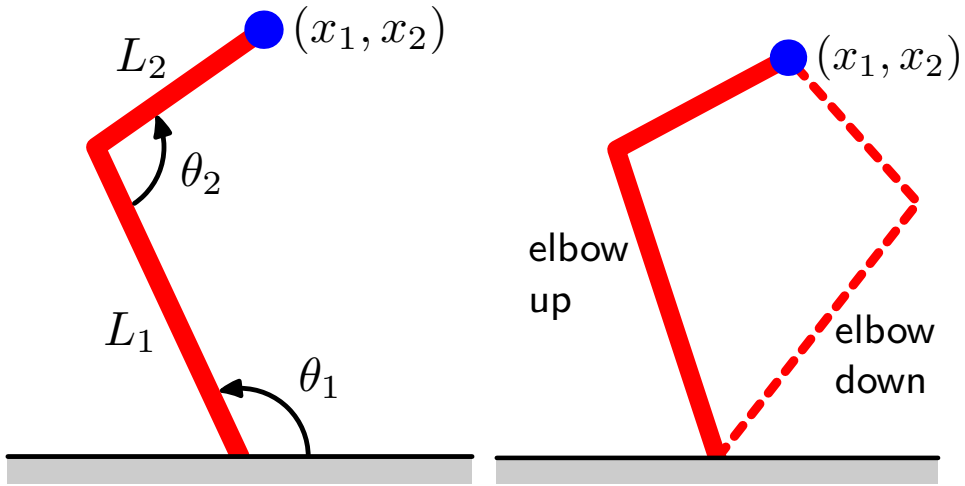
(a) original image
(b) tangent vector for clockwise rotation
(c) synthetic data using tangent vector
(d) true image rotated

- local receptive field
- weight sharing
- sub-sampling



Mixture Density Networks

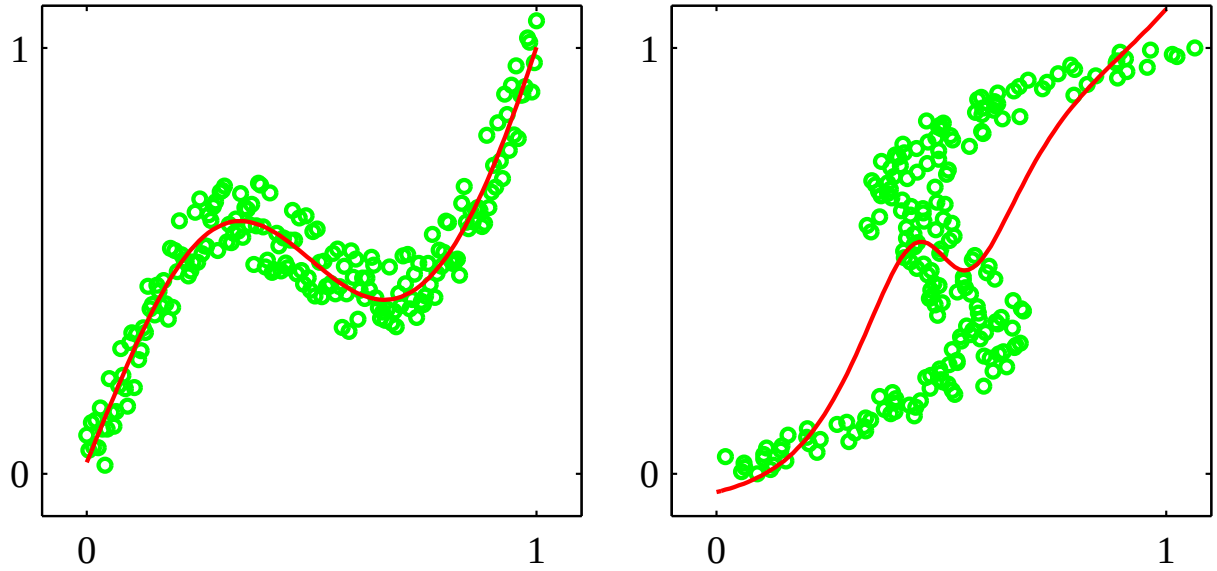
Forward Problem and Inverse Problem



In a **forward problem**, the causes are given and the effects are to be decided. In an **inverse problem**, the effects are given and the causes are to be decided. In an inverse problem, it is common to have multiple causes for given effects.

Example

Fitting a data set with a 2-layer network with 6 hidden-layer units.



The data set is generated according to

$$t_n = f(x_n) + \epsilon_n, \;\; f(x) = x + 0.3\sin(2\pi x)$$

In the forward problem, we fit a data set to $t \approx y(x, \boldsymbol{w})$. In the inverse problem, we fit a data set to $x \approx y'(t, \boldsymbol{w}')$.

- If there are multiple choices for a given input, such as in an inverse problem, we need a **mixture model** for the conditional distribution.
- A general framework is

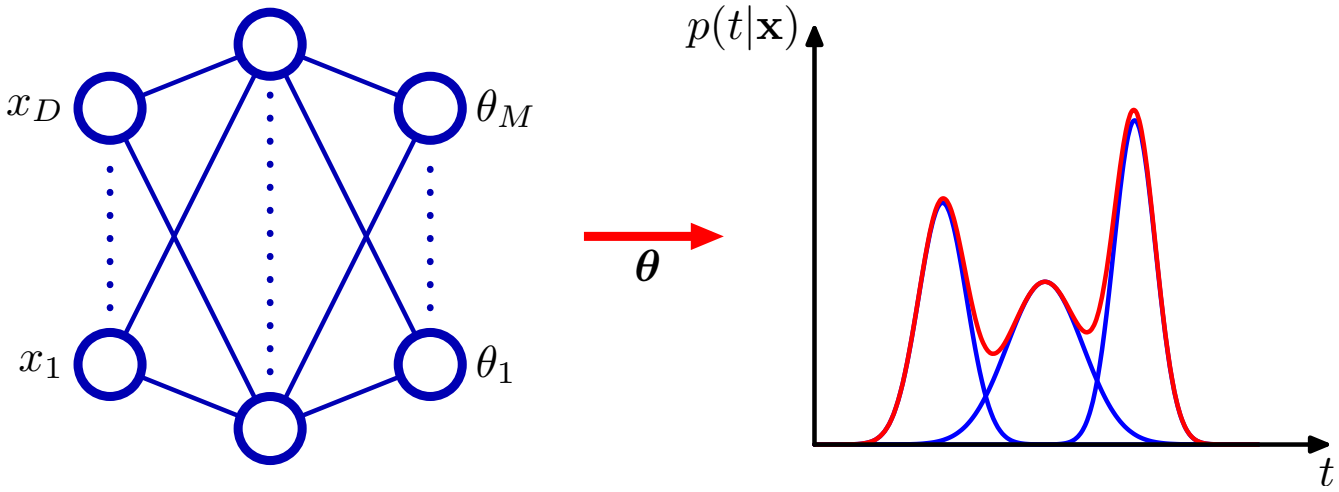
$$p(\boldsymbol{t}|\boldsymbol{x}) \approx \sum_k c_k(\boldsymbol{x}) p_k(\boldsymbol{t}|\boldsymbol{w}(\boldsymbol{x}))$$

  where the model parameters, including the mixing coefficients and the parameters in the component densities, depend on $\boldsymbol{x}$.
- Using Gaussian component densities, we have

$$p(\boldsymbol{t}|\boldsymbol{x}) \approx \sum_k \pi_k(\boldsymbol{x}) \mathcal{N}(\boldsymbol{t}|\boldsymbol{\mu}_k(\boldsymbol{x}), \boldsymbol{\Sigma}_k(\boldsymbol{x}))$$

  It approximates arbitrary conditional distributions.

In **mixture density network**, we use a neural network to model the functions relating the parameters in the mixture density to the input variables.



Output Activation Functions

Consider a mixture model with Gaussian component densities, each with a diagonal covariance matrix $\boldsymbol{\Sigma}_k(\boldsymbol{x}) = \sigma_k^2(\boldsymbol{x})\boldsymbol{I}$.

- Normalized mixing coefficients can be achieved by softmax
- Positive variances can be achieved by exponential activation
- Gaussian means can be approximated by simple linear activation

The error function is

$$E(\boldsymbol{w}) = -\sum_n \log p(\boldsymbol{t}_n|\boldsymbol{x}_n)$$

$$= -\sum_n \log \left(\sum_k \pi_k(\boldsymbol{x}_n, \boldsymbol{w}) \mathcal{N}(\boldsymbol{t}_n|\boldsymbol{\mu}_k(\boldsymbol{x}_n, \boldsymbol{w}), \sigma_k^2(\boldsymbol{x}_n, \boldsymbol{w})\boldsymbol{I}) \right)$$

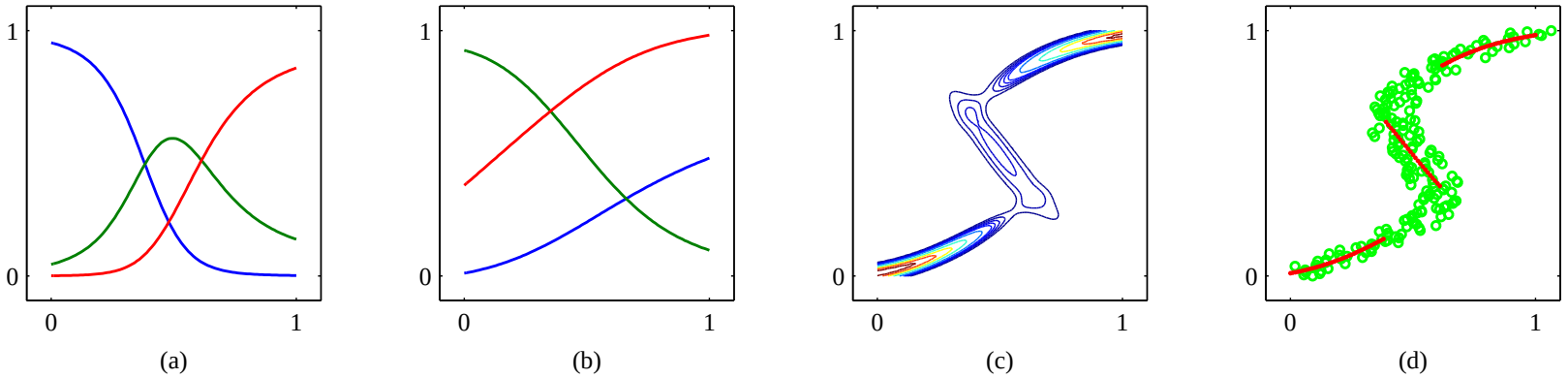
The derivatives of E with respect to the network output activations are

$$\frac{\partial E_n}{\partial a_k^\pi} = \pi_k - \gamma_{nk} \quad \text{where } \gamma_{nk} = \frac{\pi_k \mathcal{N}_{nk}}{\sum_l \pi_l \mathcal{N}_{nl}}$$

and

$$\frac{\partial E_n}{\partial a_{kl}^\mu} = \gamma_{nk} \left(\frac{\mu_{kl} - t_{nl}}{\sigma_k^2} \right), \quad \frac{\partial E_n}{\partial a_k^\sigma} = \gamma_{nk} \left(L - \frac{\|\boldsymbol{t}_n - \boldsymbol{\mu}_k\|^2}{\sigma_k^2} \right)$$

Results



Network architecture: 5 $\tanh$ hidden-layer units, 9 output-layer units

(a) mixing coefficients $\pi_k(x),\ k = 1, 2, 3$

(b) mean $\mu_k(x)$

(c) contours of conditional density

(d) conditional mode (red)

Bayesian Neural Networks

Step by Step

- Make the Gaussian noise assumption and parametric function approximation

$$p(t|\boldsymbol{x}, \boldsymbol{w}, \beta) = \mathcal{N}(t|y(\boldsymbol{x}, \boldsymbol{w}), \beta^{-1})$$

- Assume a Gaussian prior for $\boldsymbol{w}$

$$p(\boldsymbol{w}|\alpha) = \mathcal{N}(\boldsymbol{w}|\mathbf{0}, \alpha^{-1}\boldsymbol{I})$$

- The likelihood of data set $\mathcal{D} = \{(\boldsymbol{x}_n, t_n)\}$is

$$p(\mathcal{D}|\boldsymbol{w}, \beta) = \prod_n \mathcal{N}(t_n|y(\boldsymbol{x}_n, \boldsymbol{w}), \beta^{-1})$$

- The posterior distribution of $\boldsymbol{w}$ is non-Gaussian, with

$$\log p(\boldsymbol{w}|\mathcal{D}) = -\frac{\alpha}{2}\boldsymbol{w}^T\boldsymbol{w} - \frac{\beta}{2}\sum_n (y(\boldsymbol{x}_n, \boldsymbol{w}) - t_n)^2 + \text{const}$$

- Apply the Laplace approximation for the posterior of $\boldsymbol{w}$

$$p(\boldsymbol{w}|\mathcal{D}) \approx q(\boldsymbol{w}|\mathcal{D}) = \mathcal{N}(\boldsymbol{w}|\boldsymbol{w}_{\text{MAP}}, \boldsymbol{A}^{-1})$$

where

$$\boldsymbol{A} = -\boldsymbol{\nabla}\boldsymbol{\nabla} \log p(\boldsymbol{w}|\mathcal{D}) = \alpha \boldsymbol{I} + \beta \boldsymbol{H}(\boldsymbol{w}_{\text{MAP}})$$

- The predictive distribution of t is

$$p(t|\boldsymbol{x}, \mathcal{D}) = \int p(t|\boldsymbol{x}, \boldsymbol{w}) p(\boldsymbol{w}|\mathcal{D}) d\boldsymbol{w} \approx \int p(t|\boldsymbol{x}, \boldsymbol{w}) q(\boldsymbol{w}|\mathcal{D}) d\boldsymbol{w}$$

- Approximate $y(\boldsymbol{x}, \boldsymbol{w}) \approx y(\boldsymbol{x}, \boldsymbol{w}_{\text{MAP}}) + \boldsymbol{g}^T(\boldsymbol{w} - \boldsymbol{w}_{\text{MAP}})$ where $\boldsymbol{g} = \boldsymbol{\nabla}_{\boldsymbol{w}} y(\boldsymbol{x}, \boldsymbol{w}_{\text{MAP}})$. Then

$$p(t|\boldsymbol{x}, \mathcal{D}) \approx \mathcal{N}(t|y(\boldsymbol{x}, \boldsymbol{w}_{\text{MAP}}), \sigma^2(\boldsymbol{x}))$$

where

$$\sigma^2(\boldsymbol{x}) = \beta^{-1} \boldsymbol{g}^T \boldsymbol{A}^{-1} \boldsymbol{g}$$

When the hyperparameters α and β are not fixed, we iteratively update α and β and the posterior distribution.

- Given α and β, we update $\boldsymbol{w}_{\text{MAP}}$ by maximizing the posterior distribution

$$p(\boldsymbol{w}|\mathcal{D}, \alpha, \beta) \propto p(\boldsymbol{w}|\alpha)p(\mathcal{D}|\boldsymbol{w}, \beta)$$

- Given $\boldsymbol{w}_{\text{MAP}}$, we update α and β by maximizing the marginal likelihood

$$p(\mathcal{D}|\alpha, \beta) = \int p(\boldsymbol{w}|\alpha)p(\mathcal{D}|\boldsymbol{w}, \beta)d\boldsymbol{w}$$

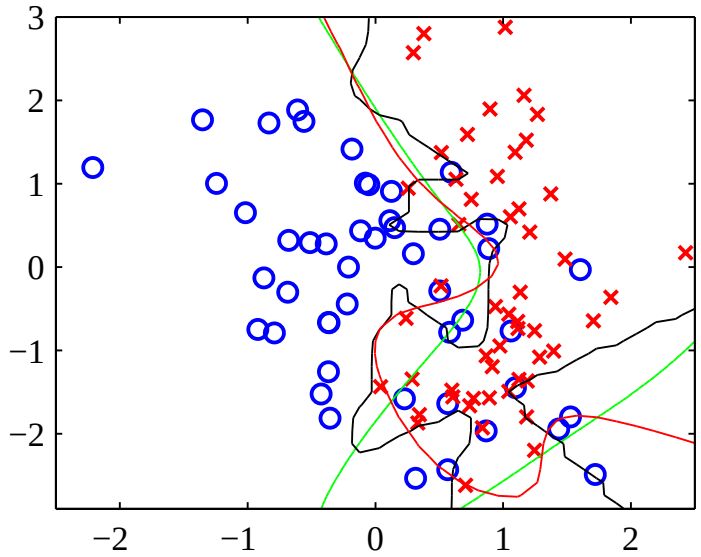
In Bayesian setting, the main difference between learning regression and learning binary classification is the likelihood function. In binary classification, the data likelihood is

$$\log p(\mathcal{D}|\boldsymbol{w}) = \sum_n t_n \log y_n + (1 - t_n) \log(1 - y_n)$$

Given α, the parameters that maximizes the posterior probability is

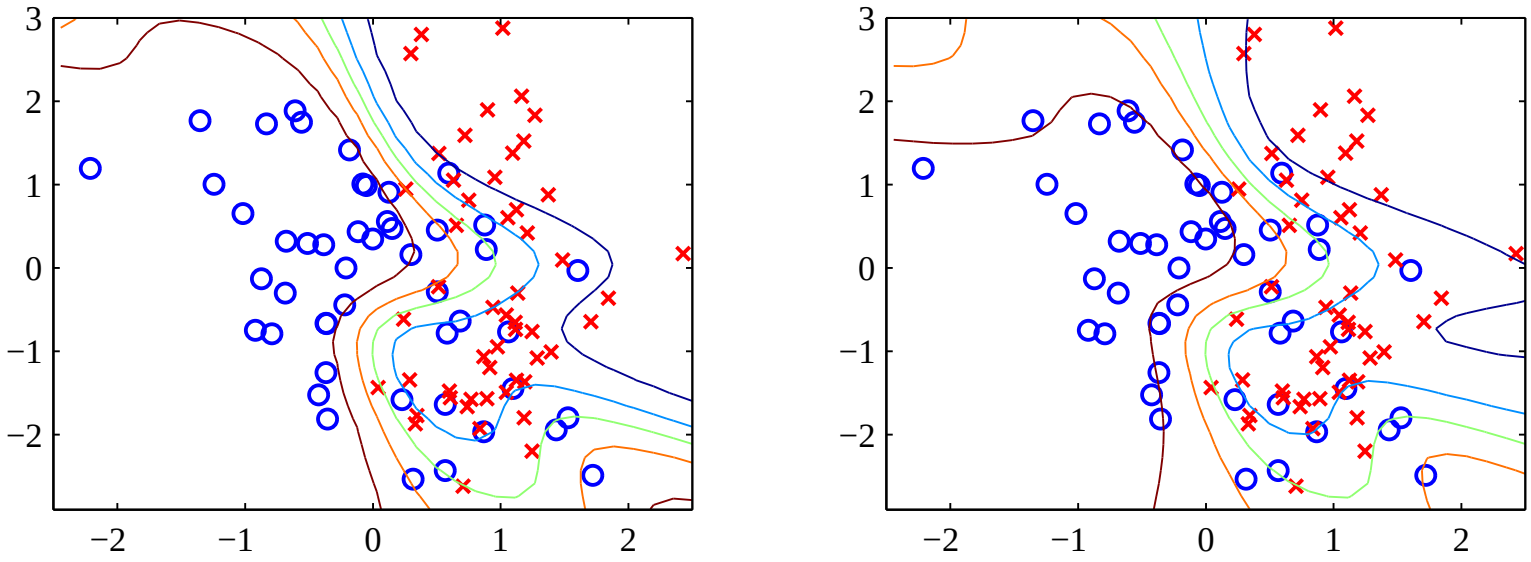
$$\boldsymbol{w}_{\text{MAP}} = \arg\min_{\boldsymbol{w}} - \log p(\mathcal{D}|\boldsymbol{w}) + \frac{\alpha}{2}\boldsymbol{w}^T\boldsymbol{w}$$

Again we alternate between updating the hyperparameter α and updating the posterior distribution.



The optimal decision boundary (green), the decision boundary learned by maximum likelihood (black), and the decision boundary learned by a regularizer whose hyperparameter α is optimized using the evidence procedure (red). A 2-layer network with 8 hidden-layer units is used to fit data.

Predictive Distribution



A 2-layer network with 8 hidden-layer $\tanh$ units is used to fit data.
Left: point estimate

$$y(\boldsymbol{x}) = p(t|\boldsymbol{x}, \mathcal{D}) \approx p(t|\boldsymbol{x}, \boldsymbol{w}_{\text{MAP}})$$

Right: Bayesian

$$y(\boldsymbol{x}) = p(t|\boldsymbol{x}, \mathcal{D}) = \int p(t|\boldsymbol{x}, \boldsymbol{w}) q(\boldsymbol{w}|\mathcal{D}) d\boldsymbol{w}$$

Going Deep

- DeepSpeech (automatic speech recognition)
- WaveNet (speech synthesis)
- ImageNet (image classification)
- Translator (machine translation)
- AlphaGo (honorary 10 dan)

LeCun, Bengio, and Hinton, "Deep Learning", Nature