Neural Networks

Chapter 5

Prof. Reza Azadeh
University of Massachusetts Lowell

Previously

- We discussed models of regression and classification comprising linear combinations of fixed basis functions.

$$y(\mathbf{x}, \mathbf{w}) = f(\sum_{j=1}^{M} w_j \phi_j(\mathbf{x})) = f(\mathbf{w}^\top \boldsymbol{\phi}(\mathbf{x}))$$

Previously

- useful analytical and computational properties
- practically limited by curse of dimensionality
- can we do better?

Feed-forward Neural Networks

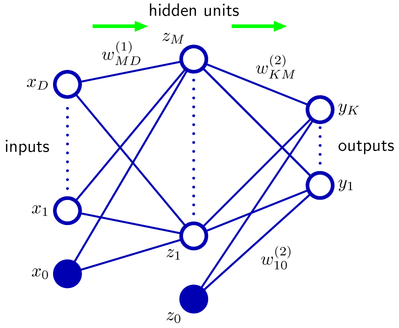
We can fix the number of basis functions but allow them to be adaptive during training.

$$y(\mathbf{x}, \mathbf{w}) = f(\sum_{j=1}^{M} w_j \phi_j(\mathbf{x})) = f(\mathbf{w}^\top \boldsymbol{\phi}(\mathbf{x}))$$

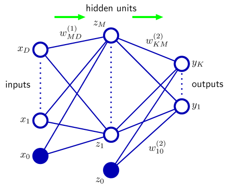
To do so, we want to make the basis functions $\phi_j(\mathbf{x})$ depend on parameters and allow the parameters to be adjusted along with coefficients $\mathbf{w}$ during the training phase.

Feed-forward Neural Networks

Among other ways, neural networks use a nonlinear function of a linear combination of the inputs



Feed-forward Neural Networks

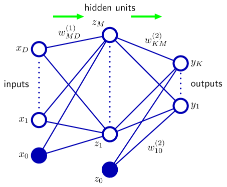


$$a_j = \sum_{i=0}^{D} w^{(1)}_{ji} x_i, \quad z_j = h(a_j)$$

$$y_k(\mathbf{x}, \mathbf{w}) = \sigma\big(\sum_{j=0}^{M} w^{(2)}_{kj} h\big(\sum_{i=0}^{D} w^{(1)}_{ji} x_i\big)\big)$$

$w^{(1)}_{ji}$ indicates parameters of the first layer of the network. Note that $w_j0^{(1)}$ are the biases. The quantities a_j are known as *activations*, each of them is then transformed using a differentiable, nonlinear *activation function* $h(.)$.

Feed-forward Neural Networks



$$a_j = \sum_{i=0}^{D} w^{(1)}_{ji} x_i, \quad z_j = h(a_j)$$

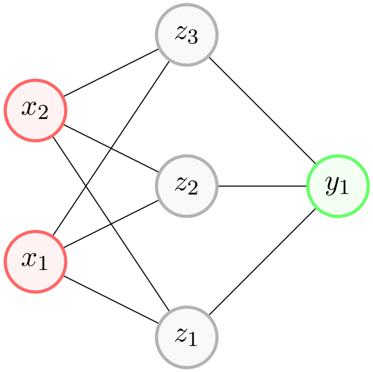
$$y_k(\mathbf{x}, \mathbf{w}) = \sigma\Big(\sum_{j=0}^{M} w^{(2)}_{kj} h\Big(\sum_{i=0}^{D} w^{(1)}_{ji} x_i\Big)\Big)$$

- For standard regression problems, we use the identity activation function so that $y_k = a_k$.
- For multiple binary classification problems, each output unit activation is transformed using a logistic sigmoid function, so that $y_k = \sigma(a_k)$, and $\sigma(a) = \frac{1}{1+\exp(-a)}$.
- For multiclass problems, a softmax activation function is used.

Feed-forward Neural Networks

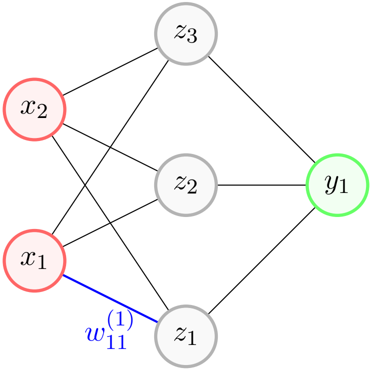
- The approximation properties of the feed-forward neural networks have been widely studied.
- They are said to be **universal approximators**.
- A two-layer network with linear outputs can uniformly approximate any continuous function on a compact input domain to arbitrary accuracy provided the network has a sufficiently large number of hidden layers.

A detailed example (1)



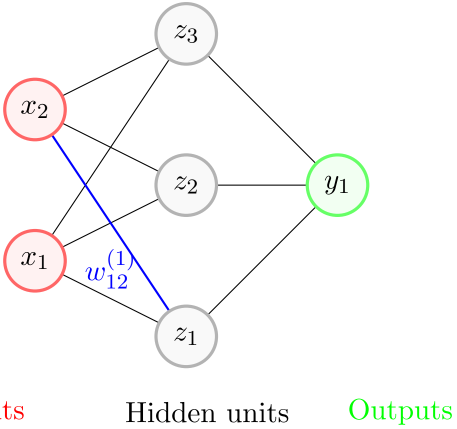
Inputs Hidden units Outputs

A detailed example (2)

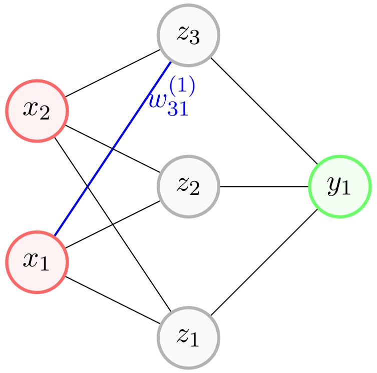


Inputs Hidden units Outputs

A detailed example (3)

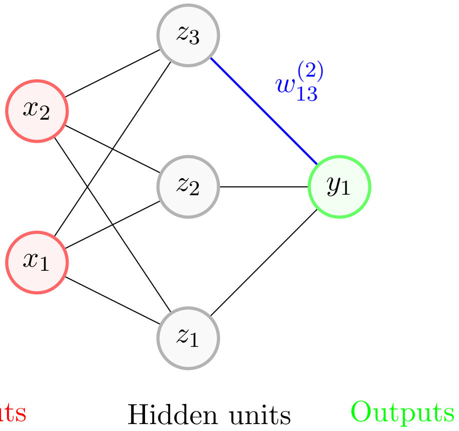


A detailed example (4)

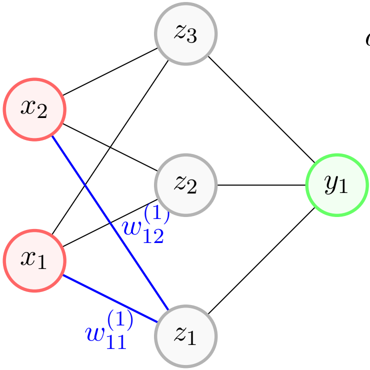


Inputs Hidden units Outputs

A detailed example (5)

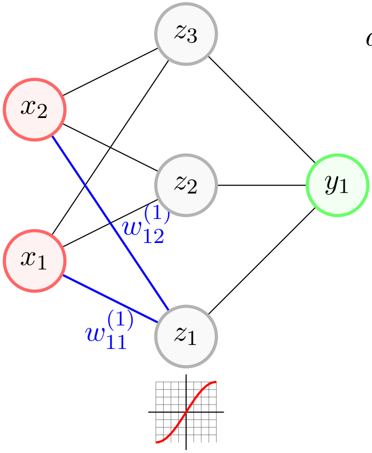


A detailed example (6)



$$a_1 = w_{11}^{(1)} x_1 + w_{12}^{(1)} x_2$$

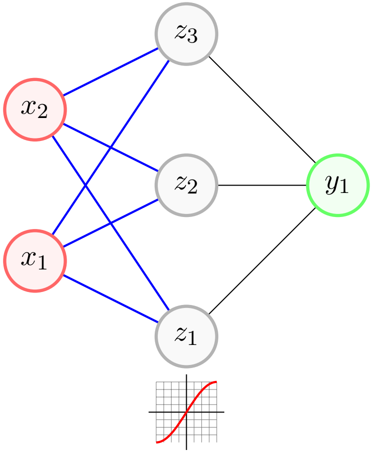
A detailed example (7)



$$a_1 = w_{11}^{(1)} x_1 + w_{12}^{(1)} x_2$$

$$z_1 = h(a_1)$$

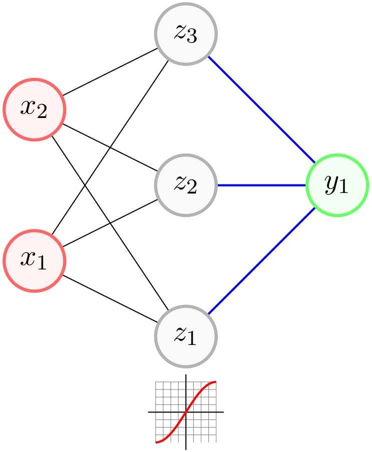
A detailed example (8)



$$\begin{bmatrix} z_1 \\ z_2 \\ z_3 \end{bmatrix} = h\left(\begin{bmatrix} w_{11}^{(1)} & w_{12}^{(1)} \\ w_{21}^{(1)} & w_{22}^{(1)} \\ w_{31}^{(1)} & w_{32}^{(1)} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \right)$$

$$\mathbf{z} = h(\mathbf{W}^{(1)}\mathbf{x})$$

A detailed example (9)



$$y_1 = \sigma\left(\begin{bmatrix} w_{11}^{(2)} & w_{12}^{(2)} & w_{13}^{(2)} \end{bmatrix} \begin{bmatrix} z_1 \\ z_2 \\ z_3 \end{bmatrix} \right)$$

$$y = \sigma(\mathbf{W}^{(2)}\mathbf{z})$$

A detailed example (10)

For a multi-layer neural network, we can write

$$\mathbf{y} = \sigma\Big(\mathbf{W}^{(n)} \dots h\Big(\mathbf{W}^{(2)}\big(h(\mathbf{W}^{(1)}\mathbf{x})\big)\Big)\Big)$$

Network Training

Given a training set of input vectors $\{\mathbf{x}_n\}$, where $n = 1, \ldots, N$, together with a corresponding set of target vectors $\{\mathbf{t}_n\}$, we minimize the error function

$$E(\mathbf{w}) = \frac{1}{2} \sum_{n=1}^{N} \|\mathbf{y}(\mathbf{x}_n, \mathbf{w}) - \mathbf{t}_n\|^2$$

Choice of error function

We assume (we later relax this assumption) that t has a Gaussian distribution with an $\mathbf{x}$-dependent mean, which is given by the output of the neural network, so that

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

where β^{-1} is the precision of the Gaussian noise.
For a data set of N independent, identically distributed observations $\mathbf{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$, along with corresponding target values $\mathbf{t} = \{t_1, \ldots, t_N\}$, we can construct the corresponding likelihood function

$$p(\mathbf{t}|\mathbf{X}, \mathbf{w}, \beta) = \prod_{n=1}^{N} p(t_n|\mathbf{x}_n, \mathbf{w}, \beta)$$

Choice of error function

The negative log likelihood is

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

which can be used to learn the parameters $\mathbf{w}$ and β. For determining $\mathbf{w}$, we maximize the likelihood function which is equivalent to minimizing the sum-of-squares error function

$$E(\mathbf{w})=\frac{1}{2}\sum_{n=1}^{N}\{y(\mathbf{x}_n,\mathbf{w})-t_n\}^2$$

when we find $\mathbf{w}_{\mathrm{ML}}$, we then can find

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

Important Property

For regression, the network has an output activation function that is the identity, so that $y_k = a_k$.
The corresponding sum-of-squares error function has the property

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

which will be used in the error backpropagation calculation.

Binary classification

When we have a single target variable, such that $t = 1$ denotes class $\mathcal{C}_1$, and $t = 0$ denotes class $\mathcal{C}_2$, we consider a network having a single output whose activation function is a logistic sigmoid

$$y = \sigma(a) = \frac{1}{1 + \exp(-a)}$$

The conditional distribution of targets can be represented using a Bernoulli distribution.

Binary classification - Choice of error function

If we consider a training set of independent observations, the error function is the *cross-entropy* error function

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

where y_n denotes $y(\mathbf{x}_n, \mathbf{w})$.
Note: using the cross-entropy error function instead of the sum-of-squares for a classification problem leads to faster training as we as improved generalization.

K separate binary classifications

We can use a network with K outputs each of which has a logistic sigmoid activation function.
Taking the negative logarithm of the corresponding likelihood function then gives the error function

$$E(\mathbf{w}) = -\sum_{n=1}^{N}\sum_{k=1}^{K}\{t_{nk}\ln y_{nk} + (1-t_{nk})\ln(1-y_{nk})\}$$

where y_{nk} denotes $y_k(\mathbf{x}_n, \mathbf{w})$.

Multiclass Classification

Each input is assigned to one of the K mutually exclusive classes. The binary target variables $t_k \in \{0, 1\}$ have a 1-of-K coding scheme indicating the class and the network outputs are interpreted as $y_k(\mathbf{x}_n, \mathbf{w}) = p(t_k = 1|\mathbf{x})$, leading to the following error function

$$E(\mathbf{w}) = -\sum_{n=1}^{N} \sum_{k=1}^{K} t_{nk} \ln y_k(\mathbf{x}_n, \mathbf{w}).$$

and the output unit activation function is softmax

$$y_k(\mathbf{x}, \mathbf{w}) = \frac{\exp(a_k(\mathbf{x}, \mathbf{w}))}{\sum_j \exp(a_j(\mathbf{x}, \mathbf{w}))}.$$

Parameter Optimization

Find a weight vector $\mathbf{w}$ that minimizes the error function $E(\mathbf{w})$.

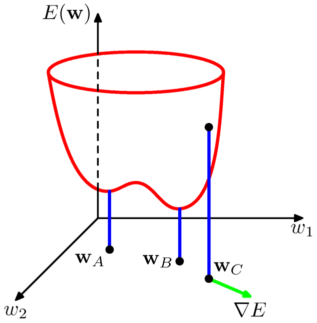


Figure: Surface representing $E(\mathbf{w})$ sitting over weight space. Point $\mathbf{w}_A$ is a local minimum and $\mathbf{w}_B$ is a global minimum. At any point $\mathbf{w}_C$, the local gradient of the error surface is given by vector ∇E.

Optimization Challenges

- The error function typically has a highly nonlinear dependence on the weights and bias parameters, and there will be many points in weight space where the gradient vanishes.
- Typically there will be multiple inequivalent stationary points and in particular multiple inequivalent minima (i.e., local and global minima).
- For a successful application of neural networks, it may not be necessary to find the global minimum but it may be necessary to compare several local minima to find a sufficiently good solution.

Numerical Optimization

- With no analytical solution to $\nabla E(\mathbf{w}) = 0$, we resort to iterative numerical optimization.
- Many iterative methods start from an initial solution $\mathbf{w}_0$ and take steps $\Delta\mathbf{w}$ toward the minimum at each step τ:

$$\mathbf{w}^{\tau+1} = \mathbf{w}^{\tau} + \Delta\mathbf{w}^{\tau}$$

- One of the most way of calculating the step is using the gradient: $\Delta\mathbf{w} = \nabla E(\mathbf{w})$.
- How should we calculate the gradient?

Local quadratic approximation (1)

Consider the Taylor expansion of $E(\mathbf{w})$ around some point $\hat{\mathbf{w}}$ in weight space

$$E(\mathbf{w}) \approx E(\hat{\mathbf{w}}) + (\mathbf{w} - \hat{\mathbf{w}})^\top \mathbf{b} + \frac{1}{2}(\mathbf{w} - \hat{\mathbf{w}})^\top \mathbf{H}(\mathbf{w} - \hat{\mathbf{w}})$$

where

$$\begin{aligned} \mathbf{b} &\equiv \nabla E|_{(\mathbf{w}=\hat{\mathbf{w}})} \\ \mathbf{H} &\equiv \nabla\nabla E|_{(\mathbf{w}=\hat{\mathbf{w}})} = \frac{\partial E}{\partial w_i \partial w_j}|_{(\mathbf{w}=\hat{\mathbf{w}})} \end{aligned}$$

Local quadratic approximation (2)

Our approximated gradient is

$$\nabla E \approx \mathbf{b} + \mathbf{H}(\mathbf{w} - \hat{\mathbf{w}})$$

For point $\mathbf{w}$ that is sufficiently close to $\hat{\mathbf{w}}$, this approximation will give reasonable approximation for the error and its gradient.

Gradient Descent optimization (1)

The simplest approach to using gradient information is to choose the weight update to comprise a small step in the direction of the negative gradient:

$$\mathbf{w}^{\tau+1} = \mathbf{w}^{\tau} + \Delta\mathbf{w}^{\tau}$$

$$\downarrow$$

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

where $\eta > 0$ is the *learning rate*.

Gradient Descent optimization (2)

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

- The gradient must be re-evaluated after each update.
- The error function is defined with respect to the training set, and so each step requires that the entire training set be processed in order to evaluate ∇E.
- Techniques that use the entire data set at once are called *batch* methods.
- This optimization method is called *Gradient Descent* or *Steepest Descent*.

Stochastic Gradient Descent optimization

- For training neural networks on large data sets, there is an online version of gradient descent known as *Sequential Gradient Descent* or *Stochastic Gradient Descent*.
- This method makes an update on the weight vector based on one data point at a time, so that

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

- It allows the algorithm to escape from local minima, since a stationary point w.r.t the error function for the whole data set will generally not be a stationary point for each data point individually.

Error Backpropagation

- Goal: finding an efficient technique for evaluating the gradient of an error function $E(\mathbf{w})$ for a feed-forward neural network.
- We shall see that this can be achieved using a local message passing scheme in which information is sent alternatively forwards and backwards through the network and is known as *error backpropagation*, or sometime as *backprop*.

Error Backpropagation

Our derivation should work for a general network having arbitrary feed-forward topology, arbitrary differentiable nonlinear activation functions, and a broad class of error functions.

Evaluation of error-function derivatives (1)

Consider a linear model and an error function

$$y_k = \sum_i w_{ki} x_i$$

$$E_n = \frac{1}{2} \sum_k (y_{nk} - t_{nk})^2$$

where $y_{nk} = y_k(\mathbf{x}_n, \mathbf{w})$. The gradient of the error function w.r.t the weight w_{ji} is

$$\frac{\partial E_n}{\partial w_{ji}} = (y_{nj} - t_{nj}) x_{ni}$$

Evaluation of error-function derivatives (2)

In a general feed-forward network, each unit computes a weighted sum of its inputs of the form

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

where z_i is the activation of a unit, or input, that sends a connection to unit j, and w_{ji} is the weight associated with the connection.

Evaluation of error-function derivatives (3)

For each input set to the network, we calculate the activations of all of the hidden and output units by successive application of

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

$$z_j = h(a_j)$$

This process is called *forward propagation*.

Evaluation of error-function derivatives (4)

To calculate the derivative of E_n, we can apply the chain rule for partial derivatives

$$\frac{\partial E_n}{\partial w_{ji}} = \frac{\partial E_n}{\partial a_j}\frac{\partial a_j}{\partial w_{ji}}$$

We introduce a useful notation $\delta_j \equiv \frac{\partial E_n}{\partial a_j}$ where δ is called *error*.
We also can see that $\frac{\partial a_j}{\partial w_{ji}} = z_i$. So, we get

$$\frac{\partial E_n}{\partial w_{ji}} = \delta_j z_i.$$

Evaluation of error-function derivatives (5)

- For the output units, we have $\delta_k = y_k - t_k$.
- For the hidden units, we use the chain rule to calculate the error

$$\delta_j \equiv \frac{\partial E_n}{\partial a_j} = \sum_k \frac{\partial E_n}{\partial a_k} \frac{\partial a_k}{\partial a_j}$$

where the sum runs over all units k to which unit j sends connections.

Evaluation of error-function derivatives (6)

$$\delta_j \equiv \frac{\partial E_n}{\partial a_j} = \sum_k \frac{\partial E_n}{\partial a_k} \frac{\partial a_k}{\partial a_j}$$

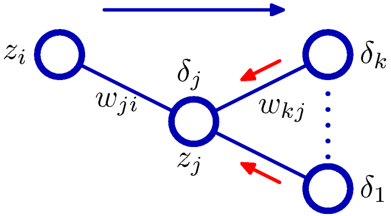


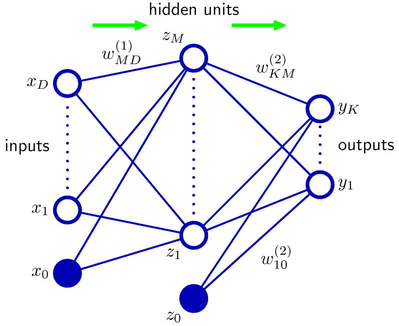
Figure: calculation of δ_j for hidden unit j by backpropagation of δ's from those units k to which unit j sends connections. The blue arrow denotes the direction of information flow during forward propagation, and the red arrow indicates the backward propagation of error information.

Error Backpropagation process

1. Apply an input vector $\mathbf{x}_n$ to the network and forward propagate through the network using $a_j = \sum_i w_{ji} z_i$ and $z_j = h(a_j)$ to find the activation of all the hidden and output units.
2. Evaluate δ_k for all the output units using $\delta_k = y_k - t_k$.
3. Backpropagate the δ's using $\delta_j = h'(a_j) \sum_k w_{kj} \delta_k$ to obtain δ_j for each hidden unit in the network.
4. Use $\frac{\partial E_n}{\partial w_{ji}} = \delta_j z_i$ to evaluate the required derivatives.

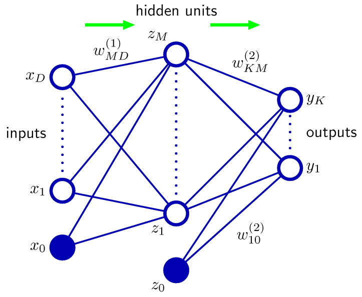
A Simple Example (1)

Consider a two-layer neural network with a sum-of-squares error, in which the output units have linear activation functions so that $y_k = a_k$, while the hidden units have the sigmoid activation function given by $h(a) = \tanh(a) = \frac{e^a - e^{-a}}{e^a + e^{-a}}$.



A Simple Example (2)

We consider a standard sum-of-squares error $E_n = \frac{1}{2}\sum_{k=1}^{K}(y_k - t_k)^2$ where y_k is the activation of output unit k, and t_k is the corresponding target, for input $\mathbf{x}_n$.



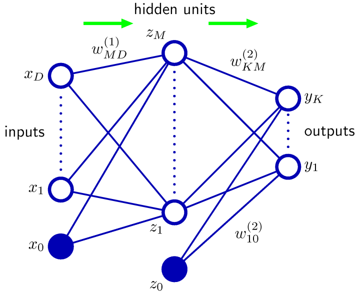
A Simple Example (3)

Performing a forward propagation

$$a_j = \sum_{i=0}^{D} w_{ji}^{(1)} x_i$$

$$z_j = \tanh(a_j)$$

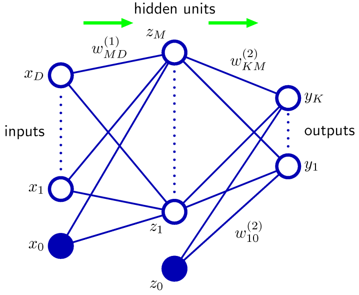
$$y_k = \sum_{j=0}^{M} w_{kj}^{(2)} z_j$$



A Simple Example (4)

Next we compute δ's for each output using $\delta_k = y_k - t_k$. Then we backpropagate these to obtain δ's for the hidden unit using

$$\delta_j = (1 - z_j^2) \sum_{k=1}^{K} w_{kj} \delta_k.$$

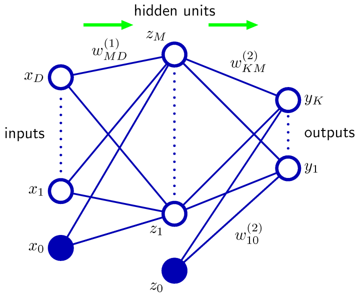


A Simple Example (5)

Finally the derivative w.r.t the first layer and second layer weights are given by

$$\frac{\partial E_n}{\partial w_{ji}^{(1)}} = \delta_j x_i$$

$$\frac{\partial E_n}{\partial w_{kj}^{(2)}} = \delta_k z_j$$



The Hessian Matrix

- The technique of backpropagation can be used to obtain the first derivatives of an error function w.r.t the weight in the network.
- Backpropagation can also be used to evaluate the second derivatives of the error, known as the Hessian matrix.
    1. Hessian is used in many optimization algorithms.
    2. Hessian forms the basis of a fast procedure of re-training a feed-forward network following a small change in the training data.
- Several Hessian approximation methods have been developed.

Network properties

- The number of input and output units is determined by the dimensionality of the data set.
- The number M of hidden units is a free parameter that can be adjusted to give the best performance.

Regularization in Neural Networks (1)

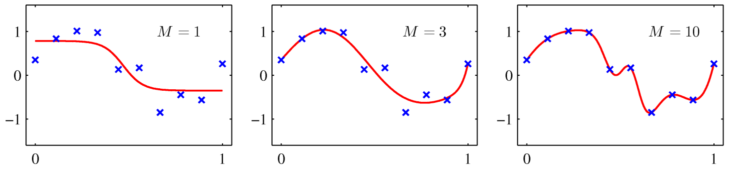


Figure: A two-layer network trained on 10 data points drawn from the sinusoidal data set. The graphs show the result of fitting networks having $M = 1, 2$ and 10 hidden units, respectively, by minimizing a sum-of-squares error function using a scaled conjugate-gradient algorithm.

Regularization in Neural Networks (2)

In practice, one approach to choosing M is to plot a graph like the one shown here and then choose the specific solution having the smallest validation set error.

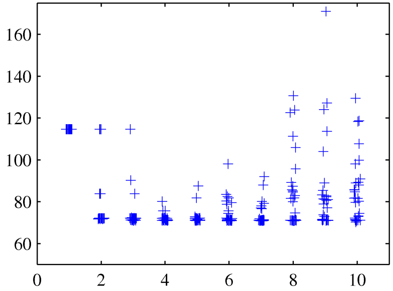


Figure: sum-of-squares test-set error for the polynomial data set versus the number of hidden units in the network, with 30 random starts for each network size.

Early Stopping

An alternative to regularization as a way of controlling the effective complexity of a network is *early stopping*. This process considers the fact that the error start increasing when the netwrok starts to overfit.

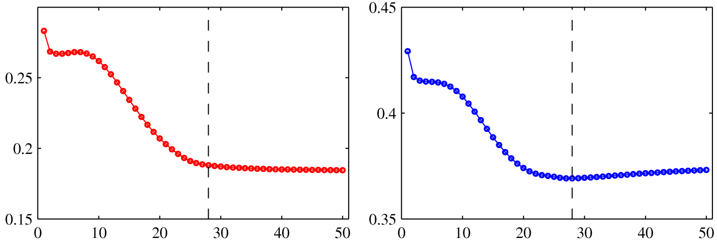


Figure: (left) training set error (right) validation set error. The goal of achieving the best generalization performance suggests that training should be stopped at the point shown by the vertical dashed lines, corresponding to the minimum of validation set error.

Invariances

- The prediction should remain unchanged or *invariant* if the input is transformed.
- For example, in classification of objects in 2D images, a particular object should be assinged the same class irrespective of its position or orientation.
- If sufficiently large number of training patterns are available, then an adaptive model such as neural network can learn invariance, at least approximately.
- For example, include examples of objects at many different positions in the training set $\rightarrow$ might be impractical.

Invariances - Practical solutions

1. The training set is *augmented* using replicas of the training patterns, transformed according to the desired invariances. For instance, in the digit recognition example, make multiple copies of each image in which the digit is shifted to a different position. $\rightarrow$ relatively easy to implement.
2. A regularization term is added to the error function to penalize changes in the model output when the input is transformed. This leads to the technique of *tangent propagation*.

Invariances - Practical solutions

3. Invariance is built into the pre-processing by extracting features that are invariant under the required transformations. The advantage is that it can correctly extrapolate well beyond the range of transformations included in the training set.
4. Build the invariance properties into the structure of a neural network. One way to achieve this is through the use of local receptive fields and shared weights. $\rightarrow$ Convolutional Neural Networks.

Convolutional Networks (1)

- Developed in 1989 and later in 1998 and wildly applied to image data.
- Assume a set of images from our digits data set. Each image is a set of pixel intensity values. The output is a posterior distribution over the ten digit classes.
- The network should be invariant to transformations such as elastic deformations.

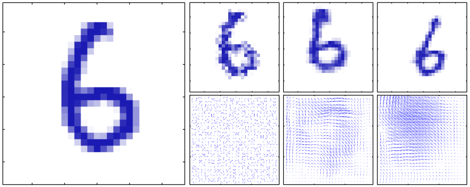
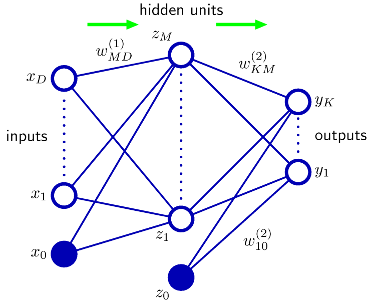


Figure: (left) original, (left, top row) three examples of warped digits with the corresponding displacement fields show on the bottom row.

Convolutional Networks (2)

- One solution is to treat each image as the input to a fully connected network.
- Given a sufficiently large training set, this network could yield a good solution and would learn appropriate invariances by example.

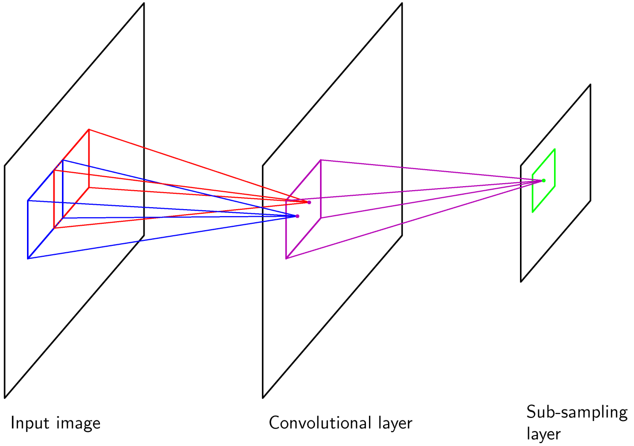


Convolutional Networks (3)

- However, this approach ignores ignores a key property of images, which is that nearby pixels are more strongly correlated than more distant pixels.
- CNNs use three mechanisms:
  1. local receptive fields
  2. weight sharing, and
  3. subsampling

Convolutional Networks (4)

Structure of a CNN is shown here


Input image
Convolutional layer
Sub-sampling layer