5.Neural Networks

Gholamreza Haffari

5.Neural Networks

Gholamreza Haffari

Contents

1
Feed-Forward Neural Networks

The Basics

Neuron. To describe neural networks, we will begin by describing the simplest possible neural network, one which comprises a single *neuron*. We will use the following diagram to denote a single neuron:

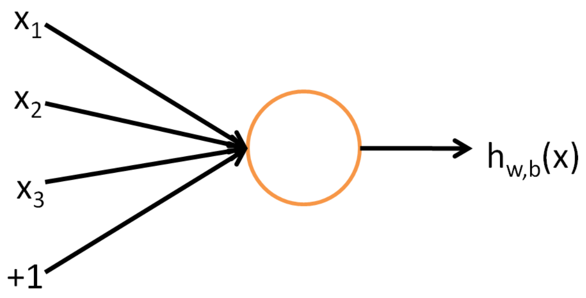


Figure 5.1.1 A single neuron

This *neuron* is a computational unit that takes as input $\mathbf{x} := (x_1, x_2, x_3)$ (and a +1 intercept term), and outputs $h_{W,b} := f(W^T x) = f(\sum_{i=1}^{3} W_i x_i + b)$ where $f : \mathbb{R} \to \mathbb{R}$ is called the *activation function*.

Activation Function. We usually choose $f(.)$ to be the *sigmoid* function:

$$\sigma(z) := \frac{1}{1+\exp(-z)}$$

The output range of the sigmoid function is [0,1]. Another common choice for $f(.)$ is the *hyperbolic tangent*, or *tanh* function:

$$tanh(z) := \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

The output range of the *tanh* function is [-1,1]. In terms of the output range, the *tanh* function is a rescaled version of the *sigmoid* function (Figure 5.1.2.).

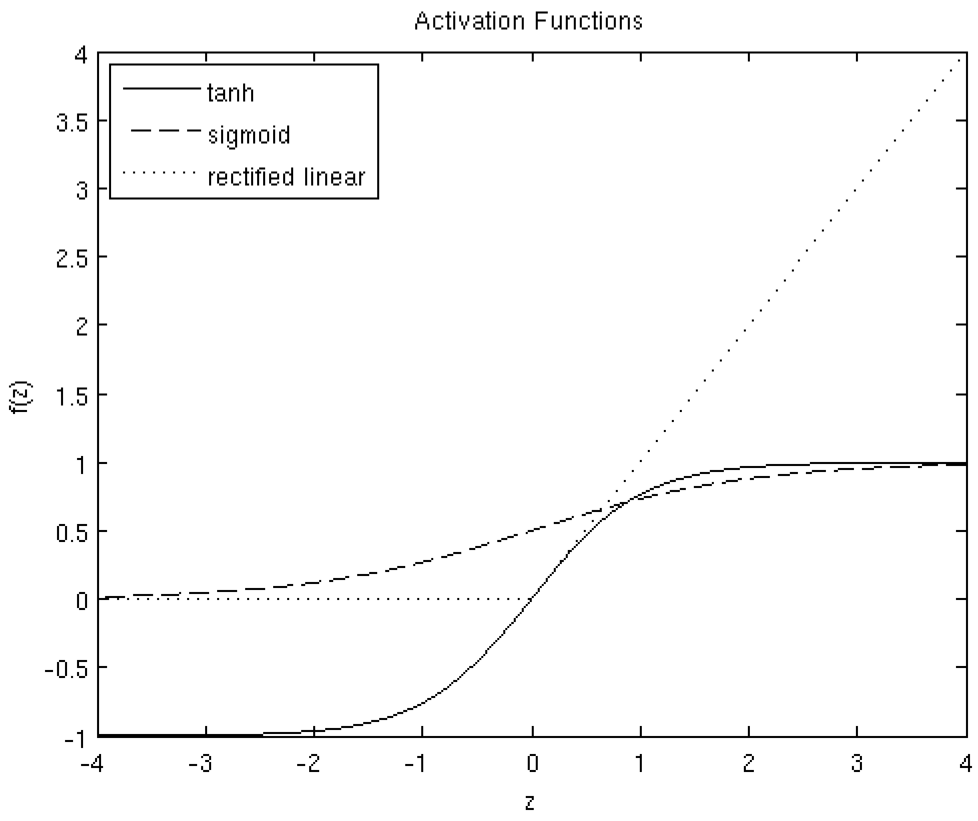


Figure 5.1.2. Three activation functions: tanh, sigmoid and rectified linear.

Let's have a look on the derivatives of these activations functions, since we will use them for the training of the model (explained in the next chapter):

$$\frac{\partial}{\partial z}\sigma(z) = \sigma(z)\big(1 - \sigma(z)\big)$$
$$\frac{\partial}{\partial z}tanh(z) = 1 - \big(tanh(z)\big)^2$$

Neural Networks

Neural Network Model. A neural network is put together by hooking together many of our simple neurons, so that the output of a neuron can be the input of another. For example, here is a small neural network:

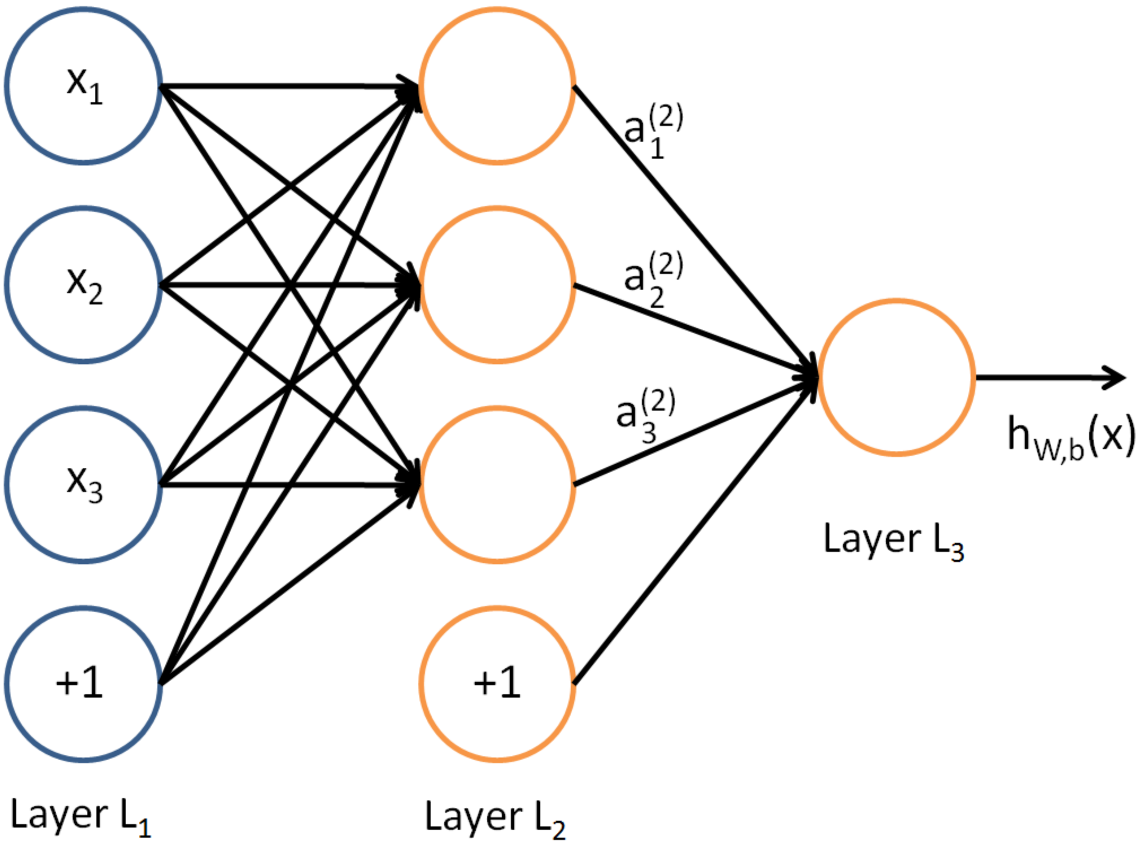


Figure 5.1.3. A 3-layer neural network

In this figure, we have used circles to also denote the inputs to the network. The circles labeled "+1" are called *bias units*, and correspond to the intercept term. The leftmost layer of the network is called the *input layer*, and the rightmost layer the *output layer* (which, in this example, has only one node). The middle layer of nodes is called the *hidden layer*, because its values are not observed in the training set. We also say that our example neural network has 3 input units (not counting the bias unit), 3 hidden units, and 1 output unit.

Network Parameters. Let's denote the number of layers in our network by n_l; thus $n_l = 3$ in our example. Let's label layer l as L_l, so layer L_1 is the input layer, and layer L_3 is the output layer. Our neural network has parameters $\boldsymbol{\theta} = (\boldsymbol{W}^{(1)}, \boldsymbol{b}^{(1)}, \boldsymbol{W}^{(2)}, \boldsymbol{b}^{(2)})$, where we write $\boldsymbol{W}_{ij}^{(l)}$ to denote the parameter (or weight) associated with the connection between unit j in layer l, and unit i in layer $l+1$ (note the order of the indices). Also, $b_i^{(l)}$ is the bias associated with unit i in layer $l+1$. Thus, in our example, we have $\boldsymbol{W}^{(1)} \in \mathbb{R}^{3\times 3}$, and $\boldsymbol{W}^{(2)} \in \mathbb{R}^{1\times 3}$. Note that bias units don't have inputs or connections going into them, since they always output the value +1. We also let s_l denote the number of nodes in layer l (not counting the bias unit).

Neuron's Activation. Let $z_i^{(l)}$ denote the total weighted sum of inputs to unit i in layer l, including the bias term (e.g., $z_i^{(2)} := \sum_{j=1}^{n} W_{ij}^{(1)} x_j + b_i^{(1)}$). Then, the output of the neutron, aka its *activation*, is denoted by $a_i^{(l)} := f(z_i^{(l)})$.

Forward Propagation. Given a fixed setting of the parameters $\boldsymbol{\theta}$, our neural network defines a *function* $h_{\boldsymbol{\theta}}(\boldsymbol{x})$ that outputs a real number. Specifically, the computation that our example neural network represents is given by:

$$\begin{aligned}
a_1^{(2)} &:= f\big(W_{11}^{(1)} x_1 + W_{12}^{(1)} x_2 + W_{13}^{(1)} x_3 + b_1^{(1)}\big) \\
a_2^{(2)} &:= f\big(W_{21}^{(1)} x_1 + W_{22}^{(1)} x_2 + W_{23}^{(1)} x_3 + b_2^{(1)}\big) \\
a_3^{(2)} &:= f\big(W_{31}^{(1)} x_1 + W_{32}^{(1)} x_2 + W_{33}^{(1)} x_3 + b_3^{(1)}\big) \\
h_{\boldsymbol{\theta}}(\boldsymbol{x}) &:= f\big(W_{11}^{(2)} a_1^{(2)} + W_{12}^{(2)} a_2^{(2)} + W_{13}^{(2)} a_3^{(2)} + b_1^{(2)}\big)
\end{aligned}$$

Note that this easily lends itself to a more compact notation. Specifically, if we extend the activation function $f(.)$ to apply to vectors in an *element-wise* fashion (i.e., $f([z_1, z_2, z_3]) = [f(z_1), f(z_2), f(z_3)]$), then we can write the equations above more compactly as:

$$\begin{aligned}
\boldsymbol{z}^{(2)} &:= \boldsymbol{W}^{(1)} \boldsymbol{x} + \boldsymbol{b}^{(1)} \\
\boldsymbol{a}^{(2)} &:= f(\boldsymbol{z}^{(2)}) \\
z^{(3)} &:= \boldsymbol{W}^{(2)} \boldsymbol{a}^{(2)} + b^{(2)} \\
h_{\boldsymbol{\theta}}(\boldsymbol{x}) &:= f(z^{(3)})
\end{aligned}$$

We call this step *forward propagation*. More generally, recalling that we use $a^{(1)} = \boldsymbol{x}$ to also denote the values from the input layer, then given layer l's activations $\boldsymbol{a}^{(l)}$, we can compute layer $l+1$'s activations $\boldsymbol{a}^{(l+1)}$ as:

$$\begin{aligned}
\boldsymbol{z}^{(l+1)} &:= \boldsymbol{W}^{(l)} \boldsymbol{a}^{(l)} + \boldsymbol{b}^{(l)} \\
\boldsymbol{a}^{(l+1)} &:= f(\boldsymbol{z}^{(l+1)})
\end{aligned}$$

By organizing our parameters in matrices and using matrix-vector operations (e.g. provided by GPUs), we can take advantage of fast linear algebra routines to quickly perform calculations in our network.

Feed-Forward Neural Network. We have so far focused on one example neural network, but one can also build neural networks with other architectures (meaning patterns of connectivity between neurons), including ones with multiple hidden layers. The most common choice is a n_l-layered network where layer l_1 is the input layer, layer n_l is the output layer, and each layer l is densely connected to layer $l+1$. In this setting, to compute the output of the network, we can successively compute all the activations in layer L_2, then layer L_3, and so on, up to layer L_{n_l}, using the equations above that describe the forward propagation step. This is one example of a *feedforward neural network*, since the connectivity graph does not have any directed loops or cycles.

Neural networks can also have multiple output units. For example, here is a network with two hidden layers layers L_2 and L_3 and two output units in layer L_4:

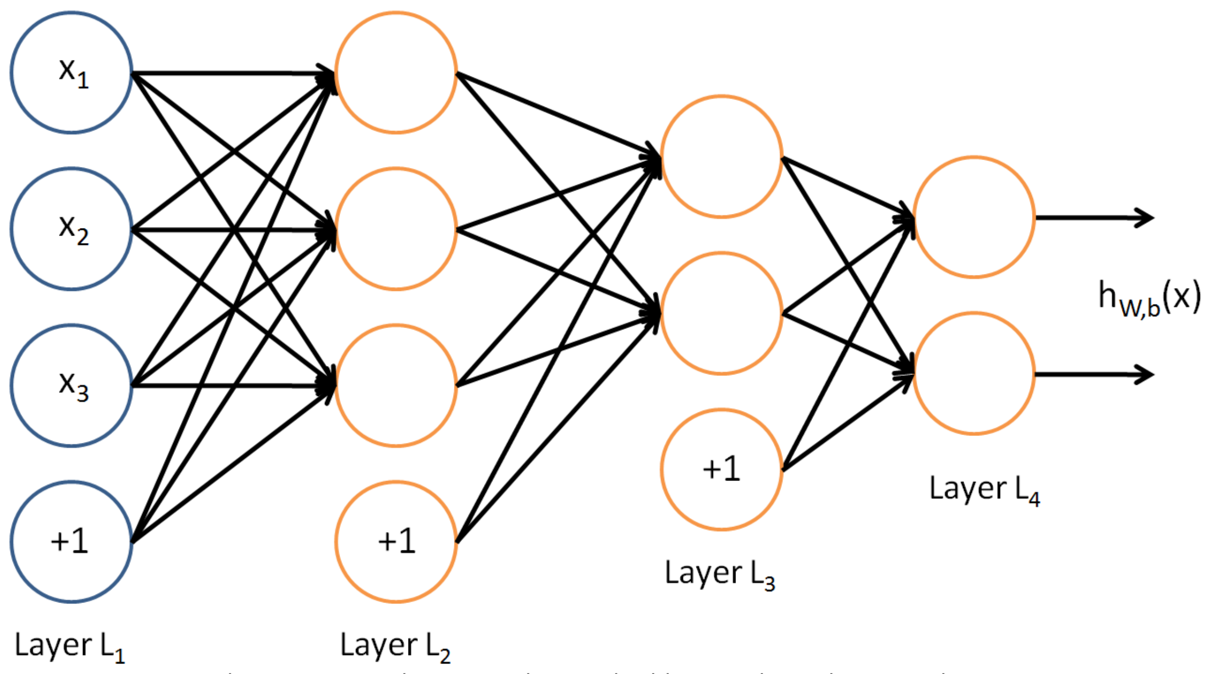


Figure 5.1.4. A 4-layer neural network with two units at the output layer.

To train this network, we would need training examples $(\boldsymbol{x}_i, \boldsymbol{y}_i)$ where $\boldsymbol{y}_i \in \mathbb{R}^2$. This sort of network is useful if there are multiple outputs that you are interested in predicting. For example, in a medical diagnosis application, the vector $\boldsymbol{x}$ might give the input features of a patient, and the different outputs $\boldsymbol{y}$'s might indicate presence or absence of different diseases.

NNs vs Perceptron. As can be seen from the above architectures, the neural network model comprises multiple stages of processing, each of which resembles the perceptron model of Module 3, and for this reason the neural network is also known as the *multilayer perceptron*, or *MLP*. A key difference compared to the perceptron, however, is that the neural network uses *continuous* nonlinearities in the hidden units (e.g. $\sigma(.)$ or $tanh$), whereas the perceptron uses step-function nonlinearities. This means that the neural network function is differentiable with respect to the network parameters, and this property will play a central role in network training (which we will see in the next Chapter).

The Power of Neural Networks

The approximation properties of feed-forward networks have been widely studied and found to be very general: the model class corresponding to neural networks can represent almost any function (given some minor conditions). Neural networks are therefore said to be *universal approximators*. For example, a two-layer network with linear outputs can uniformly approximate any continuous function on an input domain to arbitrary accuracy provided the network has a sufficiently large number of hidden units. This result holds for a wide range of hidden unit activation functions, but excluding polynomials.

In the case of classification problems, this result ensures that neural networks can approximate the target decision boundary to any required precision. Similarly, in the case of regression problems, this result ensures that the neural network model can approximate the target function to any precision. However, the price to pay in these cases is the number of neutrons in the hidden layer: as the number of neurons grows, the number of parameters grows and the network has more tendency to overfit the training data. The overfitting can be prevented by (1) using a large training data, (2) using proper regularisation methods, and (3) using *deep* architecture where the network can represent the same model class as a shallow architecture but with a much smaller number of neurons.

2
Network Training

The Training Objective

Regression. So far, we have viewed neural networks as a general class of parametric nonlinear functions from a vector $\boldsymbol{x}$ of input variables to a vector $\boldsymbol{y}$ of output variables. A simple approach to the problem of determining the network parameters is to make an analogy with the discussion of polynomial curve fitting in Module 1, and therefore to minimize a sum-of-squares error function. Given a training set comprising a set of input vectors $\{\boldsymbol{x}^{(n)}\}$, where $n = 1, \ldots, N$ together with a corresponding set of target vectors $\{\boldsymbol{y}^{(n)}\}$, we minimize the error function:

$$E(\boldsymbol{\theta}) := \frac{1}{N}\sum_{n=1}^{N} \frac{1}{2}||h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(n)}) - \boldsymbol{y}^{(n)}||_2^2$$

We can motivate this error function using a probabilistic interpretation of the network outputs. Particularly, let's assume that the target is distributed according to the following Gaussian distribution:

$$p(\boldsymbol{y}|\boldsymbol{x}, \boldsymbol{\theta}) := \mathcal{N}(\boldsymbol{y}|h_{\boldsymbol{\theta}}(\boldsymbol{x}), \boldsymbol{I})$$

where $\boldsymbol{I}$ is the *eye* (the identity) covariance matrix. The log-likelihood of the data, then, is:

$$\mathcal{L}(\boldsymbol{\theta}) := -\sum_{n=1}^{N} \ln p(\boldsymbol{y}^{(n)}|\boldsymbol{x}^{(n)}, \boldsymbol{\theta}) =$$

$$\sum_{n=1}^{N} ||h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(n)}) - \boldsymbol{y}^{(n)}||_2^2 + \text{constant}$$

So maximising the log-likelihood function $\mathcal{L}(\boldsymbol{\theta})$ is equivalent to mining the error function $E(\boldsymbol{\theta})$. The probabilistic interpretation provides us with a clearer motivation both for the choice of output unit nonlinearity and the choice of error function.

Binary Classification. Now consider the case of binary classification in which we have a single target variable y such that $y = 1$ denotes class C_1 and $y = 0$ denotes class C_2. We consider a network having a single output whose activation function is a logistic sigmoid $\sigma \in [0, 1]$. We can then interpret the network output $h_{\boldsymbol{\theta}}(\boldsymbol{x})$ as the conditional probability $p(C_1|\boldsymbol{x})$, with $p(C_2|\boldsymbol{x}) = 1 - h_{\boldsymbol{\theta}}(\boldsymbol{x})$. The conditional probability of a training input-target pair $(\boldsymbol{x}, y)$ is then

$$p(y|\boldsymbol{x}, \boldsymbol{\theta}) = [h_{\boldsymbol{\theta}}(\boldsymbol{x})]^y [1 - h_{\boldsymbol{\theta}}(\boldsymbol{x})]^{1-y}$$

If we consider a training set of independent observations, then the error function, which is given by the negative log likelihood, is then a *cross-entropy* error function of the form

$$E(\boldsymbol{\theta}) := -\sum_{n=1}^{N} y^{(n)} \ln h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(n)}) + (1-y^{(n)}) \ln(1-h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(n)}))$$

Multiclass Classification. Finally, we consider the standard multiclass classification problem in which each input is assigned to one of K mutually exclusive classes. The binary target variables $y_k \in \{0,1\}$ have a 1-of-K coding scheme indicating the class, and the network outputs are interpreted as $h_{k,\boldsymbol{\theta}}(\boldsymbol{x}) = p(y_k = 1|\boldsymbol{x})$, leading to the following error function

$$E(\boldsymbol{\theta}) := -\sum_{n=1}^{N} \sum_{k=1}^{K} y_k^{(n)} \ln h_{k,\boldsymbol{\theta}}(\boldsymbol{x}^{(n)})$$

In this case, the output unit activation function is given by the *softmax* function:

$$h_{k,\boldsymbol{\theta}}(\boldsymbol{x}) := \exp\left(z_k^{(n_l)}\right) / \sum_{j=1}^{K} \exp\left(z_j^{(n_l)}\right)$$

which satisfies $0 \leq h_k \leq 1$ and $\sum_k h_k = 1$. Recall that $z_j^{(n_l)}$ is the total weighted sum of inputs to the j-th neurone of the last layer n_l.

Summary. There is a natural choice of both output unit activation function and matching error function, according to the type of problem being solved. For regression we use linear outputs and a sum-of-squares error, for binary classifications we use the sigmoid outputs and a cross-entropy error function, and for multiclass classification we use softmax outputs with the corresponding multiclass cross-entropy error function. For classification problems involving two classes, we can use a single logistic sigmoid output, or alternatively we can use a network with two outputs having a softmax output activation function.

Parameter Optimisation

The Error Surface. We turn next to the task of finding a weight vector $\boldsymbol{\theta}$ which minimizes the chosen error function $E(\boldsymbol{\theta})$. At this point, it is useful to have a geometrical picture of the error function, which we can view as a surface sitting over weight space as shown in Figure **5.2.1**.

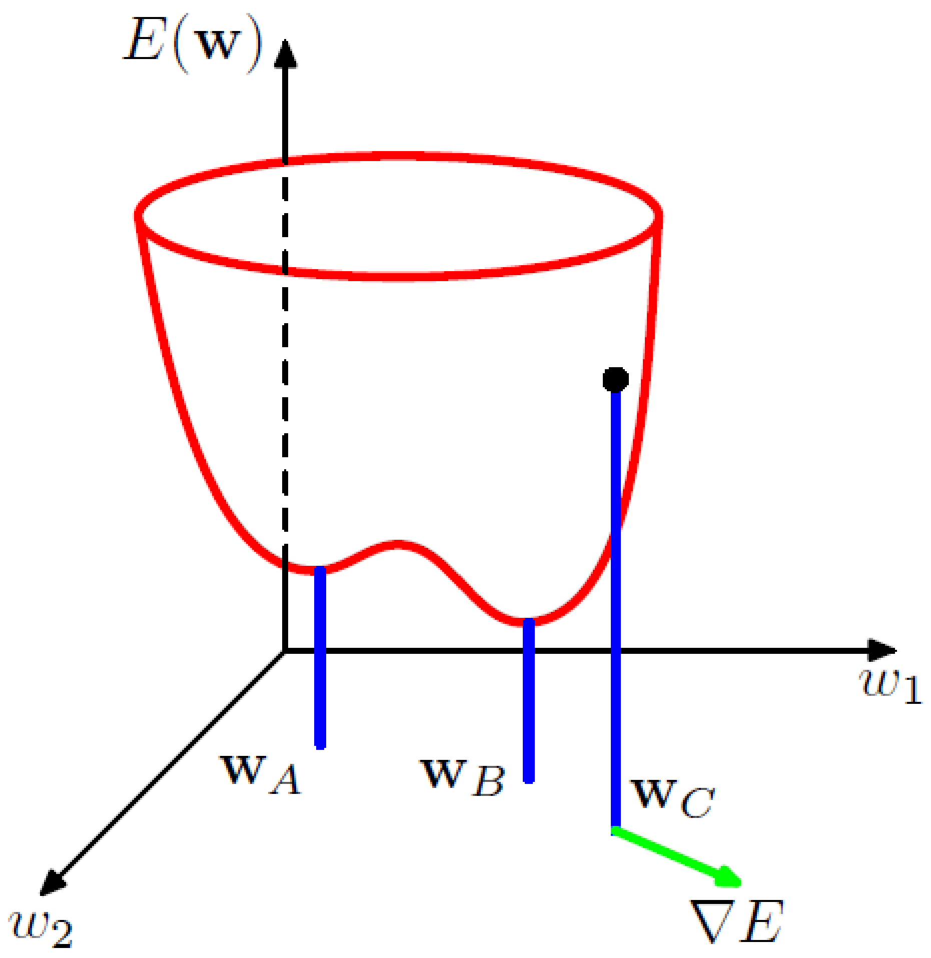


Figure 5.2.1: Geometrical view of the error function. A local and the global minimums are marked.

Note in Figure **5.2.1.** that the error function is a nonlinear and non-convex function of the weights with lots of local optima. Furthermore, as previous modules, we resort to the *(stochastic) gradient descent* algorithm, as there is no hope to find an analytical solution to the optimisation problem.

The Gradient Descent Algorithm. In the rest of this chapter, we focus on minimising the error function for the regression problem but the discussion can be easily applied to the classification problems. Let's denote the cost function to minimise for the regression problem by

$$J(\boldsymbol{\theta}) := \underbrace{\frac{1}{N}\sum_{n=1}^{N}\frac{1}{2}||\boldsymbol{h}_{\boldsymbol{\theta}}(\boldsymbol{x}^{(n)}) - \boldsymbol{y}^{(n)}||^2}_{E(\boldsymbol{\theta})} + \frac{\lambda}{2}\sum_{l=1}^{n_l-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}\left(W_{ji}^{(l)}\right)^2$$

The first term in the the average sum-of-squares error term. The second term is the ℓ_2*-regularisation* term (also called a *weight decay* term) that tends to decrease the magnitude of the weights, and helps prevent overfitting. Usually weight decay is not applied to the bias terms $b_i^{(l)}$, as reflected in our definition for $J(\boldsymbol{\theta})$. The weight decay parameter λ controls the relative importance of the two terms.

To use gradient descent algorithm to train our neural network, we need to initialise each parameter $W_{ij}^{(l)}$

and each $b_i^{(l)}$ to a small random value near zero (say according to a $\mathcal{N}(0, \epsilon^2)$) distribution for some small ϵ, say 0.01), and then apply an optimization algorithm such as (stochastic) gradient descent. Since $J(\boldsymbol{\theta})$ is a non-convex function, gradient descent is susceptible to local optima; however, in practice gradient descent usually works fairly well. Finally, note that it is important to initialize the parameters randomly, rather than to all 0's. If all the parameters start off at identical values, then all the hidden layer units will end up learning the same function of the input (more formally, $W_{ij}^{(1)}$ will be the same for all values of i, so that $a_1^{(2)} = a_2^{(2)} = a_3^{(2)} = \dots$ for any input $\boldsymbol{x}$). The random initialization serves the purpose of *symmetry breaking*.

One iteration of gradient descent updates the parameters as follows:

$$W_{ij}^{(l)} = W_{ij}^{(l)} - \eta \frac{\partial}{\partial W_{ij}^{(l)}} J(\boldsymbol{\theta})$$

$$b_i^{(l)} = b_i^{(l)} - \eta \frac{\partial}{\partial b_i^{(l)}} J(\boldsymbol{\theta})$$

where η is the learning rate. The key step is computing the partial derivatives above. We will now describe the *backpropagation* algorithm, which gives an efficient way to compute these partial derivatives.

The Backpropagation Algorithm

Gradient of the Training Objective. Let's $J(\boldsymbol{\theta}; \boldsymbol{x}, \boldsymbol{y}) := \frac{1}{2} ||h_\theta(\boldsymbol{x}) - \boldsymbol{y}||_2^2$. We first describe how backpropagation can be used to compute $\frac{\partial}{\partial W_{ij}^{(l)}} J(\boldsymbol{\theta}; \boldsymbol{x}, \boldsymbol{y})$ and $\frac{\partial}{\partial b_i^{(l)}} J(\boldsymbol{\theta}; \boldsymbol{x}, \boldsymbol{y})$; that is the partial derivatives for the contribution of one training data point to the training objective. Once we can compute these, we see that the derivative of the overall cost function $J(\boldsymbol{\theta})$ can be computed as:

$$\frac{\partial}{\partial W_{ij}^{(l)}} J(\boldsymbol{\theta}) = \frac{1}{N} \sum_{n=1}^{N} \frac{\partial}{\partial W_{ij}^{(l)}} J(\boldsymbol{\theta}; \boldsymbol{x}^{(n)}, \boldsymbol{y}^{(n)}) + \lambda W_{ij}^{(l)}$$

$$\frac{\partial}{\partial b_i^{(l)}} J(\boldsymbol{\theta}) = \frac{1}{N} \sum_{n=1}^{N} \frac{\partial}{\partial b_i^{(l)}} J(\boldsymbol{\theta}; \boldsymbol{x}^{(n)}, \boldsymbol{y}^{(n)})$$

Gradient of a Training Datum. The intuition of the backpropagation algorithm to compute the gradient is as follows. Given a training datum (x, y), we will first run a *forward pass* to compute all the activations throughout the network, including the output value of the network $h_{\boldsymbol{\theta}}(x)$. Then, for each node i in layer l, we would like to compute an *error term* $\delta_i^{(l)}$ that measures how much that node was *responsible* for any errors in the output. For an output node, we can directly measure the difference between the network's activation and the true target value, and use that to define $\delta_i^{n_l}$ (where layer n_l is the output layer). How about hidden units? For those, we will compute $\delta_i^{(l)}$ based on a weighted average of the error terms of the nodes that uses $a_i^{(l)}$ as an input.

The Backpropagation Algorithm. More specifically, here is the backpropagation algorithm:

1. Perform a feedforward pass, computing the activations for layers $L_2, \ldots, L_{n_l}$
2. For each output unit i in layer n_l (the output layer), set
   $$\delta_i^{n_l} = \frac{\partial}{\partial z_i^{(n_l)}} \frac{1}{2} ||\boldsymbol{y} - h_{\boldsymbol{\theta}}(\boldsymbol{x})||_2^2 =- (y_i - a_i^{(n_l)}) f'(z_i^{(n_l)})$$
3. For $l = n_l - 1, n_l - 2, \ldots, 2$
   - For each node i in layer l
     - Set $\delta_i^{(l)} = \left(\sum_{j=1}^{s_{l+1}} W_{ji}^{(l)} \delta_j^{(l+1)} \right) f'(z_i^{(l)})$
4. Compute the desired partial derivatives as follows:
   $$\frac{\partial}{\partial W_{ij}^{(l)}} J(\boldsymbol{\theta}; \boldsymbol{x}, \boldsymbol{y}) = a_j^{(l)} \delta_i^{l+1}$$
   $$\frac{\partial}{\partial b_i^{(l)}} J(\boldsymbol{\theta}; \boldsymbol{x}, \boldsymbol{y}) = \delta_i^{l+1}$$

Finally, we can also re-write the algorithm using matrix-vectorial notation:

1. Perform a feedforward pass, computing the activations for layers $L_2, \ldots, L_{n_l}$
2. For each output unit i in layer n_l (the output layer), set
   $$\boldsymbol{\delta}^{(n_l)} =- (\boldsymbol{y} - \boldsymbol{a}^{n_l}) \odot f'(\boldsymbol{z}^{n_l})$$
3. For $l = n_l - 1, n_l - 2, \ldots, 2$
   - For each node i in layer l
     - Set $\boldsymbol{\delta}^{(l)} = \left(\left(\boldsymbol{W}^{l} \right)^T . \boldsymbol{\delta}^{(l+1)} \right) \odot f'(\boldsymbol{z}^{(l)})$
4. Compute the desired partial gradients as follows:
   $$\nabla_{\boldsymbol{W}^{(l)}} J(\boldsymbol{\theta}; \boldsymbol{x}, \boldsymbol{y}) = \boldsymbol{\delta}^{(l+1)} . \left(\boldsymbol{a}^{(l)} \right)^T$$
   $$\nabla_{\boldsymbol{b}^{(l)}} J(\boldsymbol{\theta}; \boldsymbol{x}, \boldsymbol{y}) = \boldsymbol{\delta}^{(l+1)}$$

where $\odot$ denotes the element-wise (i.e. Hadamard) multiplication between two vectors: $\boldsymbol{a} = \boldsymbol{b} \odot \boldsymbol{c}$ if and only if $a_i = b_i . c_i$.

Putting it All Together. We are now ready to describe the full gradient descent algorithm. In the pseudo-code below, $\Delta \boldsymbol{W}^{(l)}$ is a matrix (of the same dimension as $\boldsymbol{W}^{(l)}$), and $\Delta \boldsymbol{b}(l)$ is a vector (of the same dimension as $\boldsymbol{b}^{(l)}$). Note that in this notation, $\Delta \boldsymbol{W}^{(l)}$ is a matrix, and in particular it isn't "Δ times $\boldsymbol{W}^{(l)}$."

1. Initialise *randomly* the parameters of the network $\boldsymbol{W}^{(l)}$ and $\boldsymbol{b}^{(l)}$ for all l
2. While a stopping condition is not met do:
    - Set $\Delta \boldsymbol{W}^{(l)} = \boldsymbol{0}$ and $\Delta \boldsymbol{b}^{(l)} = \boldsymbol{0}$ for all l
    - For $n = 1 \ldots N$ do
        - Use back propagation to compute $\nabla_{\boldsymbol{W}^{(l)}} J(\boldsymbol{\theta}; \boldsymbol{x}^{(n)}, \boldsymbol{y}^{(n)})$ and $\nabla_{\boldsymbol{b}^{(l)}} J(\boldsymbol{\theta}; \boldsymbol{x}^{(n)}, \boldsymbol{y}^{(n)})$
        - Set $\Delta \boldsymbol{W}^{(l)} := \Delta \boldsymbol{W}^{(l)} + \nabla_{\boldsymbol{W}^{(l)}} J(\boldsymbol{\theta}; \boldsymbol{x}^{(n)}, \boldsymbol{y}^{(n)})$
        - Set $\Delta \boldsymbol{b}^{(l)} := \Delta \boldsymbol{b}^{(l)} + \nabla_{\boldsymbol{b}^{(l)}} J(\boldsymbol{\theta}; \boldsymbol{x}^{(n)}, \boldsymbol{y}^{(n)})$
    - Update the parameters: $\boldsymbol{W}^{(l)} \leftarrow \boldsymbol{W}^{(l)} - \eta \left[\left(\frac{1}{N} \Delta \boldsymbol{W}^{(l)} \right) + \lambda \boldsymbol{W}^{(l)} \right]$ and $\boldsymbol{b}^{(l)} \leftarrow \boldsymbol{b}^{(l)} - \eta \left[\frac{1}{N} \Delta \boldsymbol{b}^{(l)} \right]$

The above algorithm is the *batch* gradient descent algorithm: the parameters are updated once after the gradient of the objective function is computed over the whole training dataset. We can modify the it to the stochastic gradient descent algorithm where the parameters are updated immediately after the gradient of the objective function is computed for each individual training datum in the dataset. There are a lot of studies investigating the convergence behaviour of these two flavours of the gradient descent algorithm, leaning towards the superiority of the stochastic gradient descent (particularly for big data).

3 Regularisation

Regularisation

The number of input and outputs units in a neural network is generally determined by the dimensionality of the data set, whereas the number M of hidden units is a free parameter that can be adjusted to give the best predictive performance. Note that M controls the number of parameters (weights and biases) in the network, and so we might expect that in a maximum likelihood setting there will be an optimum value of M that gives the best generalization performance, corresponding to the optimum balance between under-fitting and over-fitting. Figure **5.3.1.** shows an example of the effect of different values of M for the sinusoidal regression problem.

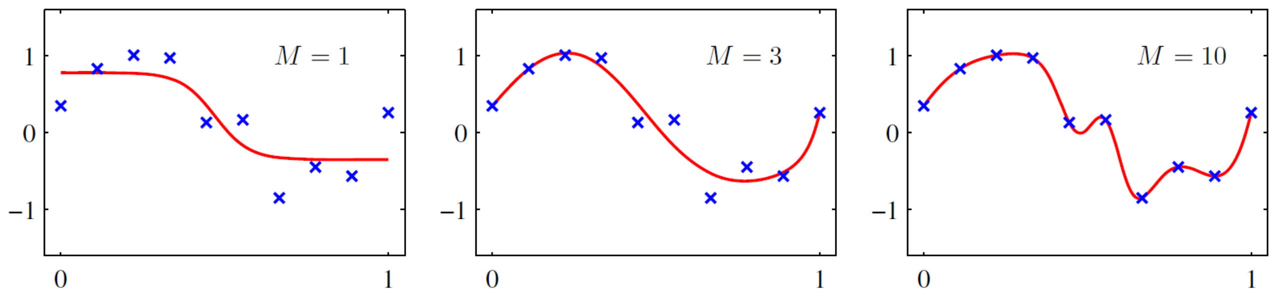


Figure 5.3.1. Examples of two-layer networks trained on 10 data points drawn from the sinusoidal data set. The graphs show the result of fitting networks having M = 1, 3 and 10 hidden units, respectively, by minimizing a sum-of-squares error function using a scaled conjugate-gradient algorithm.

The generalization error, however, is not a simple function of M due to the presence of local minima in the error function, as illustrated in Figure **5.3.2**. Here we see the effect of choosing multiple random initializations for the weight vector for a range of values of M. The overall best validation set performance in this case occurred for a particular solution having $M = 8$. In practice, one approach to choosing M is in fact to plot a graph of the kind shown in Figure **5.3.2.** and then to choose the specific solution having the smallest validation set error.

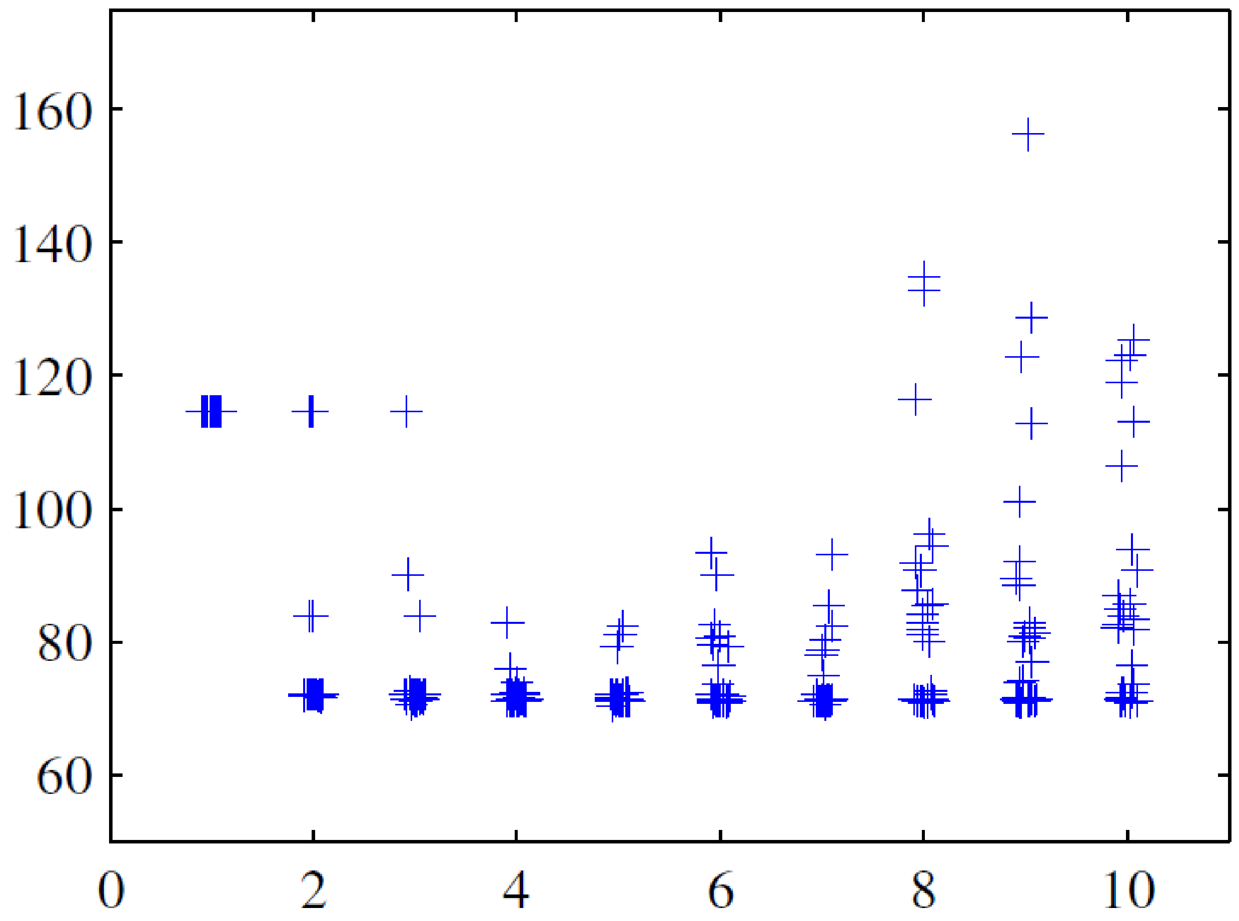


Figure 5.3.2. Plot of the sum-of-squares test-set error for the polynomial data set versus the number of hidden units in the network, with 30 random starts for each network size, showing the effect of local minima. For each new start, the weight vector was initialized by sampling from an isotropic Gaussian distribution having a mean of zero and a variance of 10.

Weight Decay. There are, however, other ways to control the complexity of a neural network model in order to avoid over-fitting. From our discussion of polynomial curve fitting in Module 1, we see that an alternative approach is to choose a relatively large value for M and then to control complexity by the addition of a regularization term to the error function. The simplest regularizer is the quadratic (aka weight decay), giving a regularised error function as we saw in the previous Chapter. The effective model complexity is then determined by the choice of the regularization coefficient λ.

Early Stopping. An alternative to regularisation as a way of controlling the effective complexity of a network is the procedure of *early stopping*. The training of nonlinear network models corresponds to an iterative reduction of the error function defined with respect to a set of training data. For many of the optimization algorithms used for network training, such as batch gradient descent, the error is a nonincreasing function of the iteration index. However, the error measured with respect to independent data, generally called a validation set, often shows a decrease at first, followed by an increase as the network starts to over-fit. Training can therefore be stopped at the point of smallest error with respect to the validation data set, as indicated in Figure **5.3.3**, in order to obtain a network having good generalization performance.

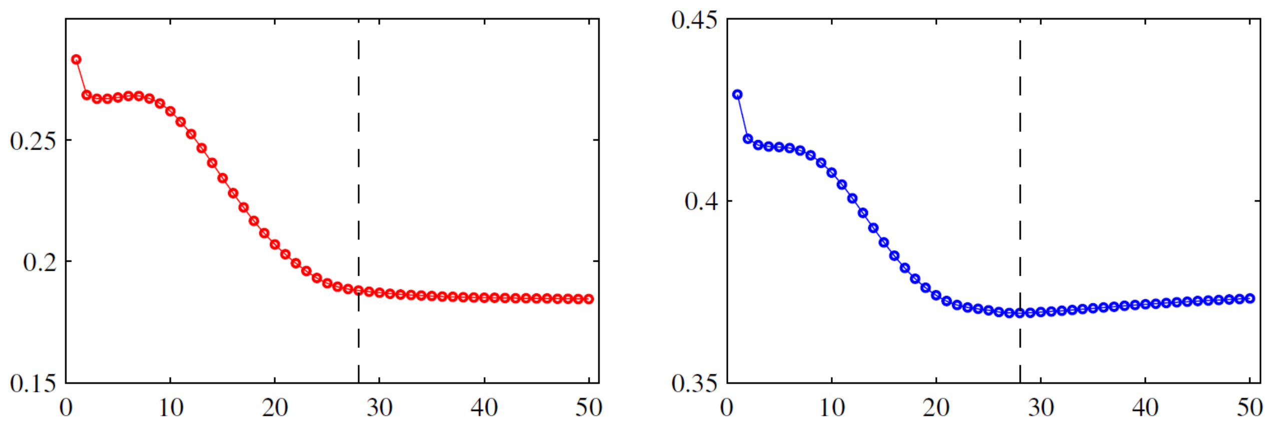


Figure 5.3.3.: An illustration of the behaviour of training set error (left) and validation set error (right) during a typical training session, as a function of the iteration step, for the sinusoidal data set. The goal of achieving the best generalization performance suggests that training should be stopped at the point shown by the vertical dashed lines, corresponding to the minimum of the validation set error.

The behaviour of the network in this case is sometimes explained qualitatively in terms of the effective number of degrees of freedom in the network, in which this number starts out small and then to grows during the training process, corresponding to a steady increase in the effective complexity of the model. Halting training before a minimum of the training error has been reached then represents a way of limiting the effective network complexity.

In the case of a quadratic error function (i.e. the regression problem), we can verify this insight, and show that early stopping should exhibit similar behaviour to regularization using a simple weight-decay term. This can be understood from Figure **5.3.4.** If, in the absence of weight decay, the weight vector starts at the origin and proceeds during training along a path that follows the local negative gradient vector, then the weight vector will move initially parallel to the w_2 axis through a point corresponding roughly to $\tilde{\boldsymbol{w}}$ and then move towards the minimum of the error function $\boldsymbol{w}_{ML}$. Stopping at a point near $\tilde{\boldsymbol{w}}$ is therefore similar to weight decay. The relationship between early stopping and weight decay can be made quantitative, thereby showing that the quantity $\tau\eta$ (where τ is the iteration index, and η is the learning rate parameter) plays the role of the reciprocal of the regularization parameter λ. The effective number of parameters in the network therefore grows during the course of training.

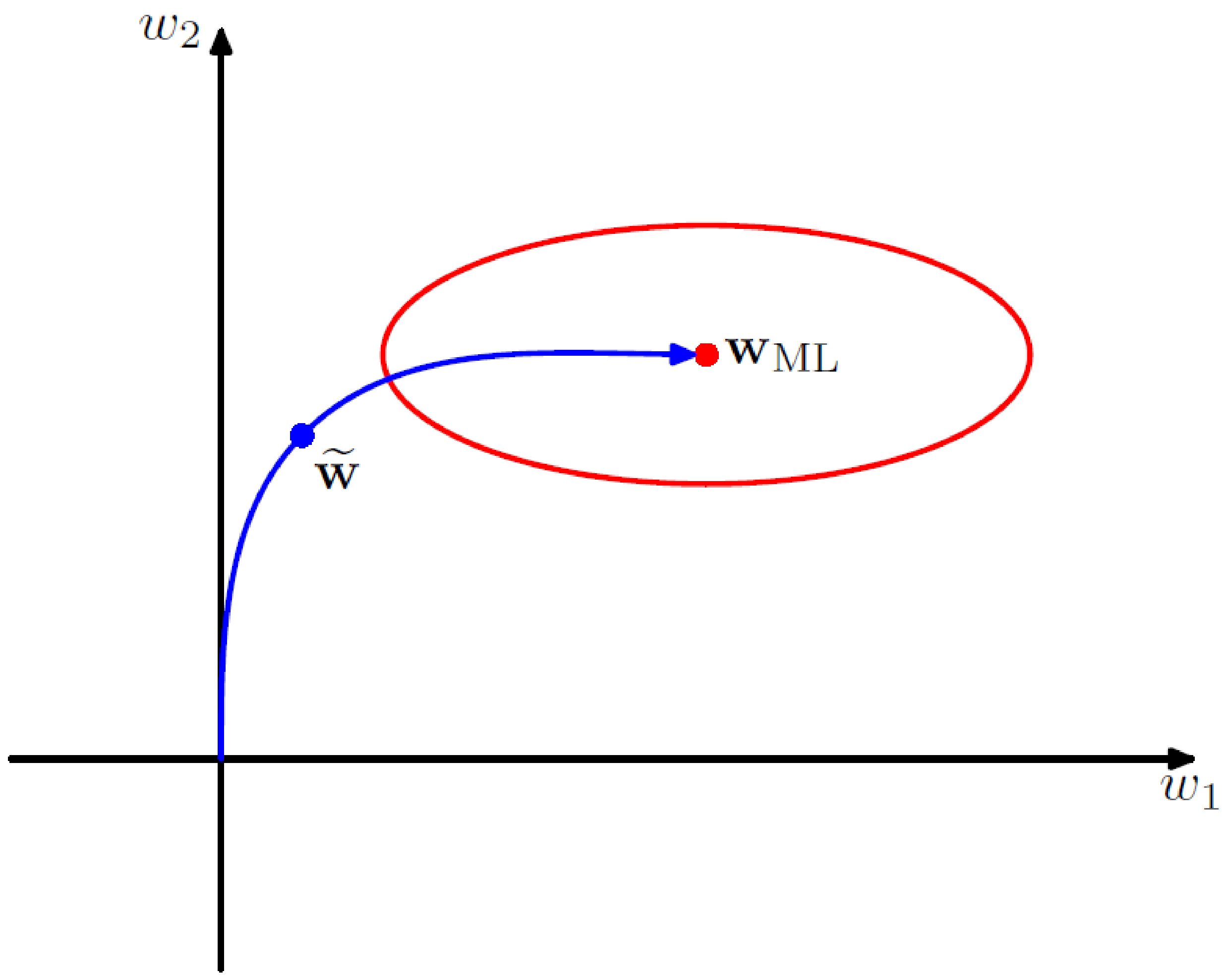


Figure 5.3.4: A schematic illustration of why early stopping can give similar results to weight decay in the case of a quadratic error function.

4
Activity 5.1: Neural Network

In this activity we implement a basic 3-layer feed-forward neural network to practice back-propagation algorithm.This activity helps you to complete Assignment 2.

Please download this (https://www.alexandriarepository.org/wp-content/uploads/20160701174801/Activity5.zip) zip file (right click and then choose "save link as") that contains the Jupyter notebook (.ipynb).

For detailed discussion about setting your programming environment up, please refer to Appendix B: Setting up Your Programming Environme (https://www.alexandriarepository.org/?post_type=module&p=97654)

5 Unsupervised and Self-Taught Learning

Unsupervised Learning with NNs

Autoencoders. In the previous chapters, we considered neural networks in the context of *supervised learning*, where the role of the network is to predict the output variables given values for the input variables. However, neural networks have also been applied to *unsupervised learning* where they have been used for dimensionality reduction. This is achieved by using a network having the same number of outputs as inputs, and optimizing the weights so as to minimize some measure of the *reconstruction error* between inputs and outputs with respect to a set of training data.

Consider first a feed-forward network of the form shown in Figure **5.5.1.**, having D inputs, D output units and M hidden units, with $M < D$. The targets used to train the network are simply the input vectors themselves, so that the network is attempting to map each input vector onto itself. Such a network is said to form an *autoencoder*. Since the number of hidden units is smaller than the number of inputs, a perfect reconstruction of all input vectors is not in general possible. We therefore determine the network parameters $\boldsymbol{\theta}$ by minimizing an error function which captures the degree of mismatch between the input vectors and their reconstructions. In particular, we shall choose a sum-of-squares error of the form:

$$E(\boldsymbol{\theta}) := \tfrac{1}{2} \sum_{n=1}^{N} ||h_{\boldsymbol{\theta}}(\boldsymbol{x}_n) - \boldsymbol{x}_n||_2^2$$

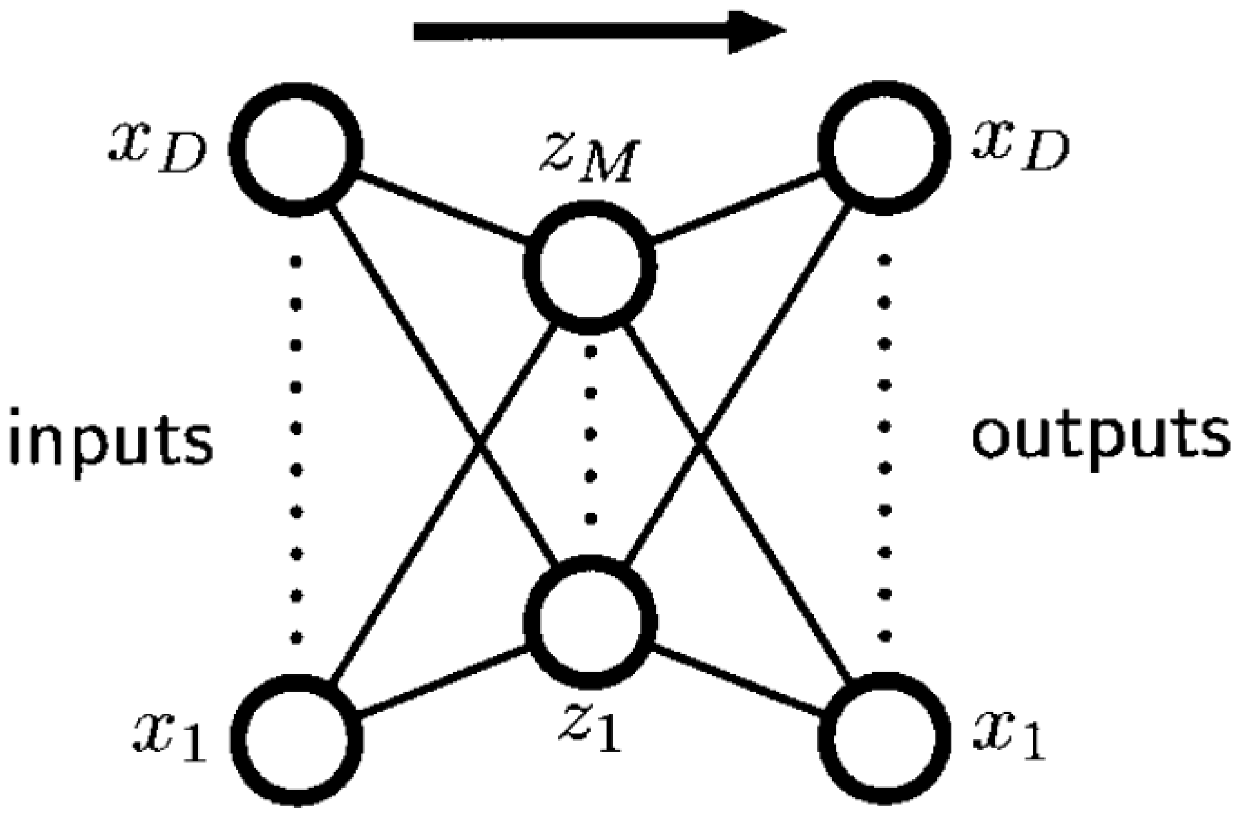


Figure 5.5.1: An autoassociative mUltilayer perceptron having two layers of weights. Such a network is trained to map input vectors onto themselves by minimization ot a sum-ot-squares error. Even with nonlinear units in the hidden layer, such a network is equivalent to linear principal component analysis. Links representing bias parameters have been omitted for clarity.

Linear Dimensionality Reduction and Relation to PCA. If the hidden units have linear activations functions, then it can be shown that the error function has a unique global minimum, and that at this minimum the network performs a projection onto the M-dimensional subspace which is spanned by the first M principal components of the data. Thus, the vectors of weights which lead into the hidden units in Figure **5.5.1.** form a basis set which spans the principal subspace, which is exactly the subspace obtained by the PCA method. As opposed to PCA, though, the vectors obtained by the autoencoder need not be orthogonal or normalized. The connection between this autoencoder and PCA is unsurprising, since both principal component analysis and the neural network are using linear dimensionality reduction and are minimizing the same sum-of-squares error function.

It might be thought that the limitations of a linear dimensionality reduction could be overcome by using nonlinear (sigmoidal) activation functions for the hidden units in the network in Figure **5.5.1.** However, even with nonlinear hidden units, the minimum error solution is again given by the projection onto the principal component subspace. There is therefore no advantage in using two-layer neural networks to perform dimensionality reduction. Standard techniques for principal component analysis (based on linear algebra matrix decomposition techniques) are guaranteed to give the correct solution in finite time.

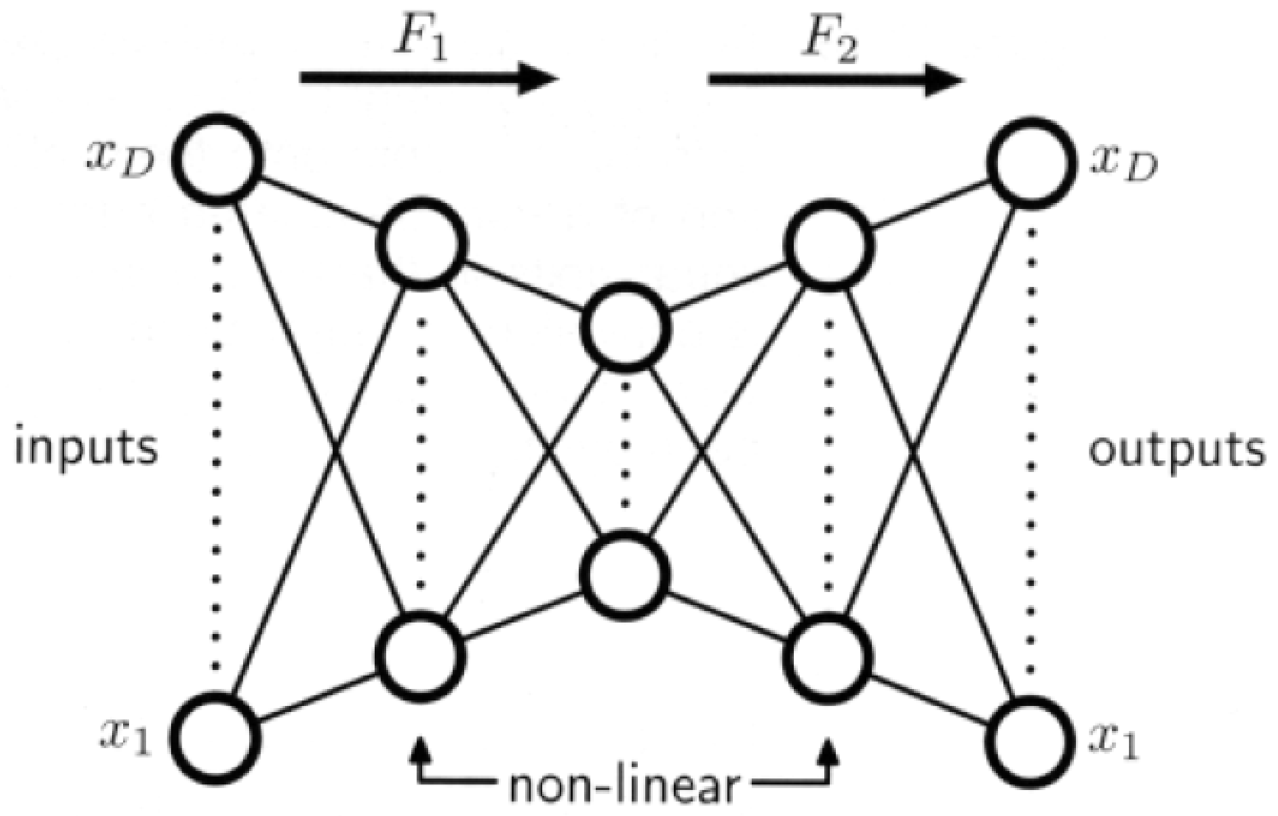


Figure 5.5.2. Addition of extra hidden layers of nonlinear units gives an auloassocialive network which can perform a nonlinear dimensionality reduction.

Non-Linear Dimensionality Reduction with Autoencoders. The situation is different, however, if additional hidden layers are permitted in the network. Consider the four-layer autoencoder shown in Figure **5.5.2**. Again the output units are linear, and the M units in the second hidden layer can also be linear, however, the first and third hidden layers have sigmoidal nonlinear activation functions. The network is again trained by minimization of the error function. We can view this network as two successive functional mappings $\boldsymbol{F}_1$ and $\boldsymbol{F}_2$, as indicated in Figure **5.5.2**. The first mapping $\boldsymbol{F}_1$ projects the original D-dimensional data onto an M-dimensional subspace $\mathcal{S}$ defined by the activations of the units in the second hidden layer. Because of the presence of the first hidden layer of nonlinear units, this mapping is very general, and in particular is not restricted to being linear. Similarly, the second half of the network defines an arbitrary functional mapping from the M-dimensional space back into the original D-dimensional input space. This has a simple geometrical interpretation, as indicated for the case $D = 3$ and $M = 2$ in Figure **5.5.3**.

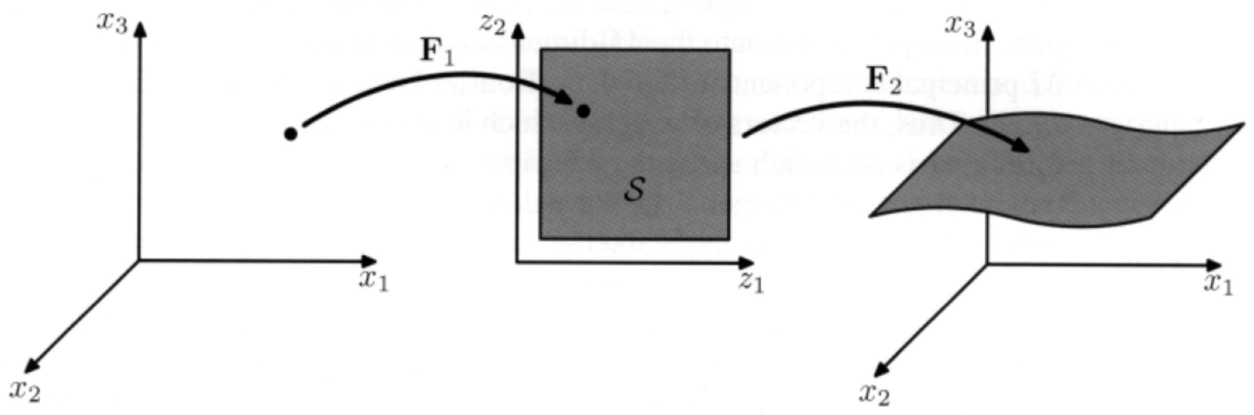


Figure 5.5.3: Geometrical interpretation of the mappings performed by the network in Figure 5.5.2 for the case of 0 = 3 inputs and AI = 2 units in the middle hidden layer.

Such a network effectively performs a *nonlinear PCA*. It has the advantage of not being limited to linear transformations, although it contains standard PCA as a special case. However, training the network now is challenging, since the error function is highly nonlinear and non-convex with lots of local optima. We provide an effective method to train such deep autoencoders in the next section.

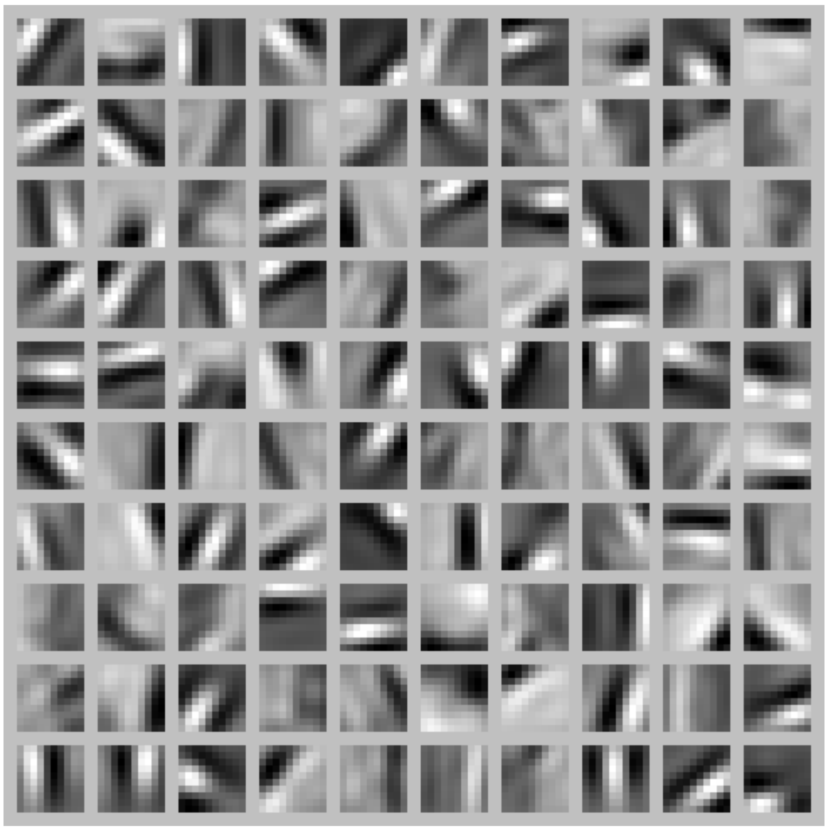
Visualizing a Trained Autoencoder. Having trained an autoencoder, we would now like to visualize the function learned by the model, to try to understand what it has learned. Consider the case of training an autoencoder on 10-by-10 images, so that $D = 100$. Each hidden unit i computes a function of the input:

$$a_i^{(2)} = f\big(\sum_{j=1}^{100} W_{ij}^{(1)} x_j + b_i^{(1)}\big)$$

We will visualize the function computed by hidden unit i, which depends on the parameters $W_{ij}^{(1)}$ (ignoring the bias term for now), using a 2D image. In particular, we think of $a_i^{(2)}$ as some non-linear feature of the input image $\boldsymbol{x}$. We ask: What input image $\boldsymbol{x}$ would cause $a_i^{(2)}$ to be maximally activated? (Less formally, what is the feature that hidden unit i is looking for?) For this question to have a non-trivial answer, we must impose some constraints on $\boldsymbol{x}$. If we suppose that the input is norm constrained by $||\boldsymbol{x}|| \leq 1$, then one can show (try doing it yourself) that the input which maximally activates hidden unit i is given by setting pixel x_j (for all 100 pixels, $j = 1, \ldots, 100$) to

$$x_j = \frac{W_{ij}^{(1)}}{\sqrt{\sum_{j=1}^{100} (W_{ij}^{(1)})^2}}$$

By displaying the image formed by these pixel intensity values, we can begin to understand what feature hidden unit i is looking for. This is an example of features learned by an auto encoder on 10-by-10 images:



Each square in the figure above shows the (norm bounded) input image $\boldsymbol{x}$ that maximally actives one of hidden units. We see that the different hidden units have learned to detect edges at different positions and orientations in the image.

Self-Taught Learning with NNs

The Motivation and Idea. Assuming that we have a sufficiently powerful learning algorithm. One of the most reliable ways to get better performance is to give the algorithm more data. This has led to the that aphorism that in machine learning, "sometimes it's not who has the best algorithm that wins; it's who has the most data."

One can always try to get more labeled data, but this can be expensive. The promise of self-taught learning is that if we can get our algorithms to learn from *unlabelled* data, then we can easily obtain massive amounts of it to significantly reduce the efforts in creating large labeled training datasets.

In Self-taught learning, we will give our algorithms a large amount of unlabeled data with which to learn a good feature representation of the input. If we are trying to solve a specific classification task, then we take this learned feature representation and whatever (perhaps small amount of) labeled data we have for that classification task, and apply supervised learning on that labeled data to solve the classification task. In summary, self-taught learning is an approach to learn from labeled and labeled data, a problem scenario which is called *semi-supervised learning*.

Learning Representations. We have already seen how autoencoders can be used to learn features from unlabeled data. Concretely, suppose we have an unlabeled training set $\{\boldsymbol{x}_u^{(1)}, \ldots, \boldsymbol{x}_u^{(N_u)}\}$ with N_u unlabeled examples (the subscript u stands for *unlabeled*). We can then train an autoencoder on this data:

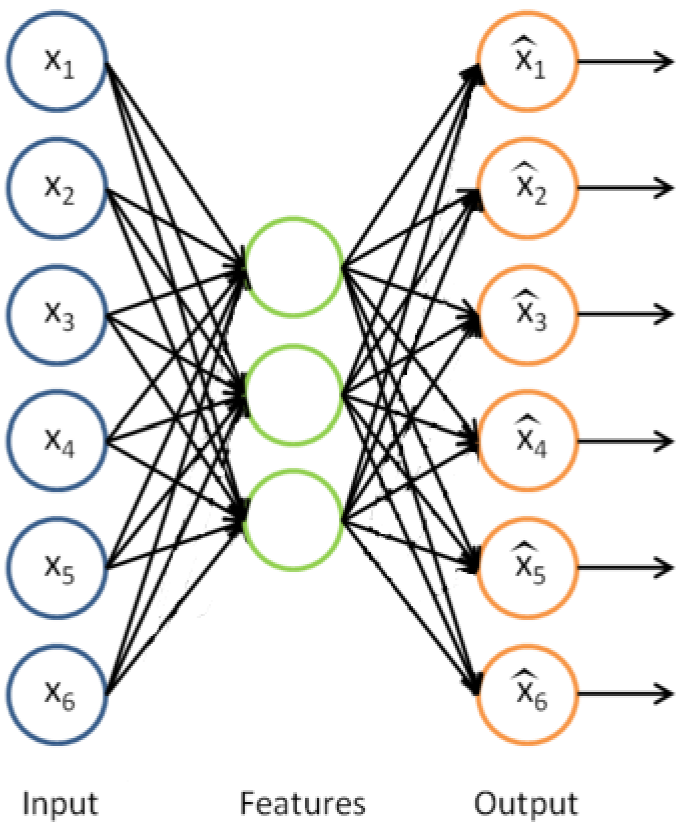


Figure 5.5.4.: Illustration of a fully connected autoencoder with 3 units at the hidden layer.

Having trained the parameters $\boldsymbol{W}^{(1)}$ of this model, given any new input $\boldsymbol{x}$, we can now compute the corresponding vector of activations $\boldsymbol{a}$ of the hidden units. As we saw previously, this often gives a better representation of the input than the original raw input $\boldsymbol{x}$. We can also visualize the algorithm for computing the features/activations $\boldsymbol{a}$ as the following neural network:

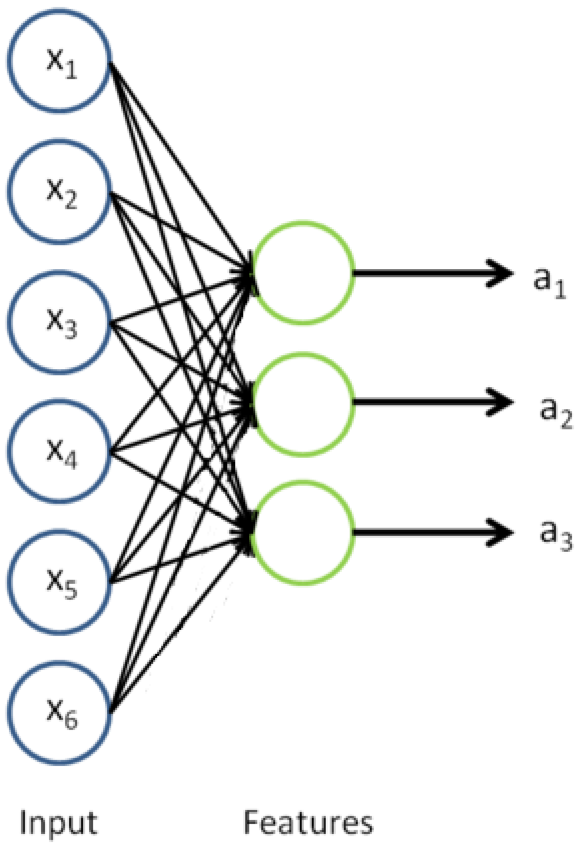


Figure 5.5.5: The same as Figure 5.5.4. only the last layer is removed.

This is just the autoencoder that we previously had, with with the final layer removed.

Now, suppose we have a labeled training set $\{(\boldsymbol{x}_l^{(1)}, \boldsymbol{y}^{(1)}), \ldots, (\boldsymbol{x}_l^{(N_l)}, \boldsymbol{y}^{(N_l)})\}$ of N_l labeled examples (the subscript l stands for *labeled*). We can now find a better representation for the inputs. In particular, rather than representing the first training example as $\boldsymbol{x}_l^{(1)}$, we can feed $\boldsymbol{x}_l^{(1)}$ as the input to our autoencoder, and obtain the corresponding vector of activations $\boldsymbol{a}_l^{(1)}$. To represent this example, we can either just replace the original feature vector with $\boldsymbol{a}_l^{(1)}$. Alternatively, we can concatenate the two feature vectors together, getting a representation $(\boldsymbol{x}_l^{(1)}, \boldsymbol{a}_l^{(1)})$.

Thus, our training set now becomes $\{(\boldsymbol{x}_l^{(1)}, \boldsymbol{a}_l^{(1)}, \boldsymbol{y}^{(1)}), \ldots, (\boldsymbol{x}_l^{(N_l)}, \boldsymbol{x}_l^{(N_l)}, \boldsymbol{y}^{(N_l)})\}$ if we use the replacement representation which concats the original input with its autoencoder representation, or $\{(\boldsymbol{a}_l^{(1)}, \boldsymbol{y}^{(1)}), \ldots, (\boldsymbol{a}_l^{(N_l)}, \boldsymbol{y}^{(N_l)})\}$ if we use the replacement representation from the autoencoder.

Finally, we can train a supervised learning algorithm such as the feed-forward NNs to obtain a function that makes predictions on the $\boldsymbol{y}$ values. Given a test example $\boldsymbol{x}_{\text{test}}$, we would then follow the same procedure: we feed it to the autoencoder to get $\boldsymbol{a}_{\text{test}}$, then the representation with autoencoder is fed into the feedforward classifier to get a prediction.

6
Activity 5.2: Autoencoder

In this activity we use H2O deep learning library to perform some experiments on autoencoder networks . This activity helps you to complete Assignment 2. For a detailed instruction for installing H2O library, please refer to Appendix **5.A.**

Please download this (https://www.alexandriarepository.org/wp-content/uploads/20160701174801/Activity5.zip) zip file (right click and then choose "save link as") that contains the R script (.r), Jupyter notebook (.ipynb) and output file (.html). The training data file (csv) can be downloaded from here: train_0_3_small (https://www.alexandriarepository.org/wp-content/uploads/20161005113206/train_0_3_small.csv). In case you want to create it yourself, here is the code to recreate it:

```
set.seed(12345)
dat <- read.csv('./train.csv')
dat <- dat[dat$label<4,]
dat <- dat[sample(nrow(dat), 1000),]
write.csv(x = dat, file= './train_0_3_small.csv', row.names = FALSE)
```

If you have access to a Jupyter instance, the notebook (and dataset) is enough to run the experiments. Otherwise, you may read the instructions and discussions from the HTML file and execute the R script. For detailed discussion about setting your programming environment up, please refer to Appendix B: Setting up Your Programming Environme (https://www.alexandriarepository.org/?post_type=module&p=97654)

7 OPTIONAL: Convolutional Neural Networks

Convolution

Fully Connected Networks. In the feedforward neural networks, one design choice that we had made was to *fully connect* all the hidden units to all the input units. On the relatively small images (e.g. 28-by-28 images in MNIST), it was computationally feasible to learn features on the entire image. However, with larger images (eg 96-by-96 images) learning fully connected networks that span the entire image is very computationally expensive. You would have about 10^4 input units, and assuming you have 100 neurone in the first hidden layer, you would have on the order of 10^6 parameters to learn. The feedforward and backpropagation computations would also be about 10^2 times slower, compared to 28-by-28 images.

Locally Connected Networks. One simple solution to this problem is to restrict the connections between the hidden units and the input units, allowing each hidden unit to connect to only a small subset of the input units. Specifically, each hidden unit will connect to only a small contiguous region of pixels in the input image. For input modalities different than images, there is often also a natural way to select *contiguous groups* of input units to connect to a single hidden unit as well; for example, for audio, a hidden unit might be connected to only the input units corresponding to a certain time span of the input audio clip.

This idea of having locally connected networks also draws inspiration from how the early visual system is wired up in biology. Specifically, neurons in the visual cortex have localized receptive fields (i.e., they respond only to stimuli in a certain location).

Convolutions. Each neurone of the first hidden layer learns a *feature of the input image. Natural images have the property of being stationary*, meaning that the statistics of one part of the image are the same as any other part. This suggests that the features that we learn at one part of the image can also be applied to other parts of the image, and we can use the same features at all locations.

More precisely, having learned features over small (say 8-by-8) patches sampled randomly from the larger image, we can then apply this learned 8-by-8 *feature detector* anywhere in the image. Specifically, we can take the learned 8-by-8 features and *convolve* them with the larger image, thus obtaining a different feature activation value at each location in the image.

To give a concrete example, suppose you have learned features on 8-by-8 patches sampled from a 96-by-96 image. Suppose further that the number of these learned feature detectors is 100. To get the convolved features, for every 8-by-8 region of the 96-by-96 image, that is, the 8×8 regions starting at $(1,1),(1,2),\ldots,(89,89)$ you would extract the 8-by-8 patch, and run it through your 100 feature detectors. This would result in 100 sets of 89^2 convolved features.

FIGURE http://ufldl.stanford.edu/tutorial/supervised/FeatureExtractionUsingConvolution/

Often, given some large r-by-c images x_{large}, we first train an *auto encoder* (they are NNs for *unsupervised* learning of features, covered in the next chapter) on small a-by-b patches $\boldsymbol{x}_{\text{small}}$ sampled from these images. This results in learning learn k features $f = \sigma(\boldsymbol{W}^{(1)}\boldsymbol{x}_{\text{small}} + \boldsymbol{b}^{(1)})$ (where σ is the sigmoid function), given by the weights $\boldsymbol{W}^{(1)}$ and biases $\boldsymbol{b}^{(1)}$ from the visible units to the hidden units. For every a-by-b patch $\boldsymbol{x}_s$ in the large image, we compute $f_s = \sigma(\boldsymbol{W}^{(1)}.\boldsymbol{x}_s + \boldsymbol{b}^{(1)})$, giving us $\boldsymbol{f}_{\text{convolved}}$, which is a $k \times (r-a+1) \times (c-b+1)$ vector of convolved features.

In the next section, we further describe how to *pool* these features together to get even better features for classification.

Pooling

Overview. After obtaining features using convolution, we would next like to use them for classification. In theory, one could use all the extracted features with a classifier such as a softmax classifier, but this can be computationally challenging. Consider for instance images of size 96-by-96 pixels, and suppose we have learned 400 features over 8-by-8 inputs. Each convolution results in an output of size [](96-8+1)*(96-8+1)=7921[], and since we have 400 features, this results in a vector of []89^2∗400=3,168,400[] features per example. Learning a classifier with inputs having 3+ million features can be unwieldy, and can also be prone to over-fitting.

To address this, first recall that we decided to obtain convolved features because images have the "stationarity" property, which implies that features that are useful in one region are also likely to be useful for other regions. Thus, to describe a large image, one natural approach is to *aggregate* statistics of these features at various locations. For example, one could compute the mean (or max) value of a particular feature over a region of the image. These summary statistics are much lower in dimension (compared to using all of the extracted features) and can also improve results (less over-fitting). This aggregation features is called *pooling*. As an example, we can have *mean pooling* or *max pooling*, depending on the pooling operation applied.

The following image shows how pooling is done over 4 non-overlapping regions of the image:

FIGURE http://ufldl.stanford.edu/tutorial/supervised/Pooling/

Formally, after obtaining our convolved features as described earlier, we decide the size of the region, say m-by-n to pool our convolved features over. Then, we divide our convolved features into disjoint m-by-n regions, and take the mean (or maximum) feature activation over these regions to obtain the pooled convolved features. These pooled features can then be used for classification.

In the next section, we further describe how to "pool" these features together to get even better features for classification.

Deep Convolutional Neural Networks

The Model. A convolutional neural network (CNN) consists of a number of convolutional and pooling layers optionally followed by fully connected layers. The convolutional layer will have k feature detectors, also called filters or kernels, where each feature detector is applied over a patch of the input. Each filter is then convolved with the input to produce a *feature map*, and then the feature map is aggregated by a pooling operation such as max-pooling. The convolution-pooling layers can be repeated and put afar each other multiple times to build deep CNN architectures.

FIGURE http://ufldl.stanford.edu/tutorial/supervised/ConvolutionalNeuralNetwork/

CNN Training. CNN can be trained by the backpropagation algorithm via (stochastic) gradient descent. pre-training?

8
Appendix: H2O Library

H2O Library

Introduction

H2O is an open source library that provides lots of machine learning algorithms which perform effectively fast. One can use this library using H2O graphical interface (called H2O Flow) or connect its R console to an H2O cluster running either locally or in the cloud. Here, we install an H2O cluster on our machine and connect our R notebook (Jupyter) use REST APIs.

Installation

Before going further, download the latest version (http://www.h2o.ai/download/h2o/choose) of H2O and unzip it in your prefered local folder. Then, go to that folder and run `java -jar h2o.jar` in your command prompt or terminal. Now you should be able to see H2O Flow at your browser http://127.0.0.1:54321/flow/index.html. Finally, follow these steps (http://www.h2o.ai/download/h2o/r) to install H2O library into your R Stodio/Terminal/Notebook.

Note 1: make sure your H2O cluster and R library have exactly the same version number.

Note 2: if you set some http proxy, save your proxy setting as `proxy.old <- Sys.getenv('http_proxy')` and unset the proxy via `Sys.setenv('http_proxy'='')` before executing the following statements. You can revert the proxy settings when you done with the experiments.

Initialization

To load H2O library, simply run `library(h2o)`. Then, cal `h2o.init` to initiate your local H2O cluster.

```
library(h2o)
#proxy.old <- Sys.getenv('http_proxy');Sys.setenv('http_proxy'='');
localH2O = h2o.init(nthreads = -1, port = 54321, startH2O = FALSE)
```

Importing Data

Unlike the regular libraries, we need to import files/datasets to H2O before executing any ML algorithm. The following code is an example of submitting a CSV file and converting it into objects that are readable by H2O functions.

```
train.file = './train.csv';
train.frame = h2o.importFile(path = train.file,sep=',')
train.label = as.character(as.vector(train.frame[,1]))
```

If everything goes right, you should be able to see the imported data when perform listing on H2O:

```
# Shows the data objects on the H2O platform
h2o.ls()
```