

Chapter 5

Clustering and Classification

Machine learning is based upon optimization techniques for data. The goal is to find both a low-rank subspace for optimally embedding the data, as well as regression methods for clustering and classification of different data types. Machine learning thus provides a principled set of mathematical methods for extracting meaningful features from data, i.e. data mining, as well as binning the data into distinct and meaningful patterns that can be exploited for decision making. Specifically, it learns from and makes predictions based on data. For business applications, this is often called *predictive analytics*, and it is at the forefront of modern data-driven decision making. In an integrated system, such as is found in autonomous robotics, various machine learning components (e.g., for processing visual and tactile stimulus) can be integrated to form what we now call *artificial intelligence* (AI). To be explicit: AI is built upon integrated machine learning algorithms, which in turn are fundamentally rooted in optimization.

There are two broad categories for machine learning: *supervised machine learning* and *unsupervised machine learning*. In the former, the algorithm is presented with labelled datasets. The training data, as outlined in the cross-validation method of the last chapter, is labeled by a teacher/expert. Thus examples of the input and output of a desired model are explicitly given, and regression methods are used to find the best model for the given labeled data, via optimization. This model is then used for prediction and classification using new data. There are important variants of supervised methods, including *semi-supervised learning* in which incomplete training is given so that some of the input/output relationships are missing, i.e. for some input data, the actual output is missing. *Active learning* is another common subclass of supervised methods whereby the algorithm can only obtain training labels for a limited set of instances, based on a budget, and also has to optimize its choice of objects to acquire labels for. In an interactive framework, these can be presented to the user for labeling. Finally, in *reinforcement learning*, rewards or punishments are the training labels that help shape the regression architecture in order to build the best model. In

contrast, no labels are given for *unsupervised learning* algorithms. Thus, they must find patterns in the data in a principled way in order to determine how to cluster data and generate labels for predicting and classifying new data. In unsupervised learning, the goal itself may be to discover patterns in the data embedded in the low-rank subspaces so that *feature engineering* or *feature extraction* can be used to build an appropriate model.

In this chapter, we will consider some of the most commonly used supervised and unsupervised machine learning methods. As will be seen, our goal is to highlight how data mining can produce important data features (feature engineering) for later use in model building. We will also show that the machine learning methods can be broadly used for clustering and classification, as well as for building regression models for prediction. Critical to all of this machine learning architecture is finding low-rank feature spaces that are informative and interpretable.

5.1 Feature selection and data mining

To exploit data for diagnostics, prediction and control, dominant features of the data must be extracted. In the opening chapter of this book, SVD and PCA were introduced as methods for determining the dominant correlated structures contained within a data set. In the eigenfaces example of Sec. 1.6, for instance, the dominant features of a large number of cropped face images were shown. These eigenfaces, which are ordered by their ability to account for commonality (correlation) across the data base of faces was guaranteed to give the best set of r features for reconstructing a given face in an ℓ_2 sense with a rank- r truncation. The eigenface modes gave clear and interpretable features for identifying faces, including highlighting the eyes, nose and mouth regions as might be expected. Importantly, instead of working with the high-dimensional measurement space, the feature space allows one to consider a significantly reduced subspace where diagnostics can be performed.

The goal of data mining and machine learning is to construct and exploit the intrinsic low-rank feature space of a given data set. The feature space can be found in an unsupervised fashion by an algorithm, or it can be explicitly constructed by expert knowledge and/or correlations among the data. For eigenfaces, the features are the PCA modes generated by the SVD. Thus each PCA mode is high-dimensional, but the only quantity of importance in feature space is the weight of that particular mode in representing a given face. If one performs an r -rank truncation, then any face needs only r features to represent it in feature space. This ultimately gives a low-rank embedding of the data in an interpretable set of r features that can be leveraged for diagnostics, prediction, reconstruction and/or control.

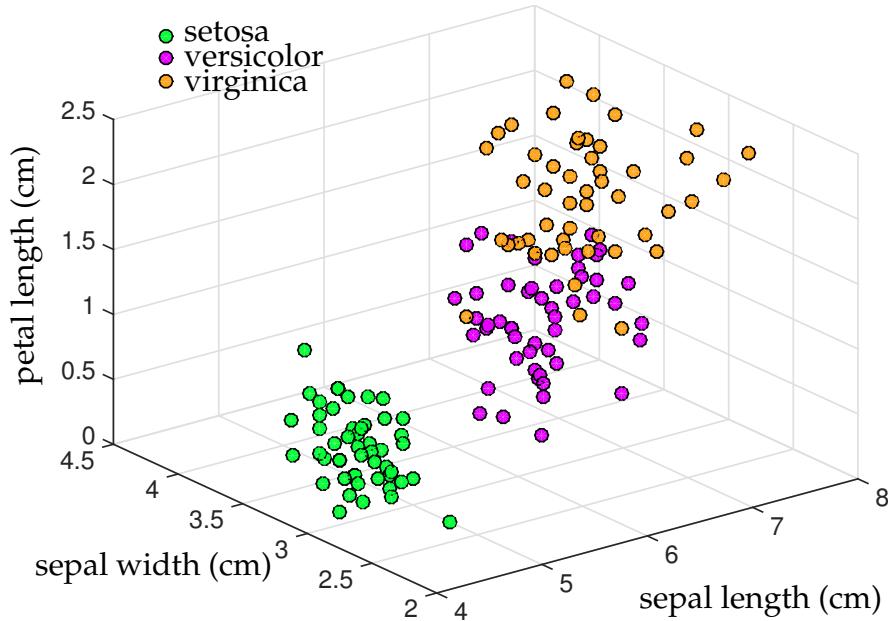


Figure 5.1: Fisher iris data set with 150 measurements over three varieties including 50 measurements each of setosa, versicolor and virginica. Each flower includes a measurement of sepal length, sepal width, petal length and petal width. The first three of these are illustrated here showing that these simple biological features are sufficient to show that the data has distinct, quantifiable differences between the species.

Several examples will be developed that illustrate how to generate a feature space, starting with a standard data set included with MATLAB. The Fisher iris data set includes measurements of 150 irises of three varieties: setosa, versicolor and virginica. The 50 samples of each flower include measurements in centimeters of the sepal length, sepal width, petal length and petal width. For this data set, the four features are already defined in terms of interpretable properties of the biology of the plants. For visualization purposes, Fig. 5.1 considers only the first three of these features. The following code accesses the Fisher iris data set:

Code 5.1: Features of the Fisher irises.

```

load fisheriris;
x1=meas(1:50,:); % setosa
x2=meas(51:100,:); % versicolor
x3=meas(101:150,:); % virginica

plot3(x1(:,1),x1(:,2),x1(:,4),'go'), hold on
plot3(x2(:,1),x2(:,2),x2(:,4),'mo')
plot3(x3(:,1),x3(:,2),x3(:,4),'ro')
```

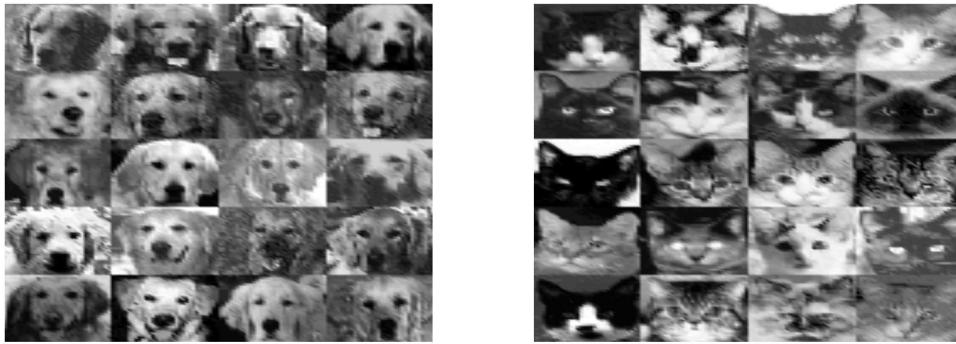


Figure 5.2: Example images of dogs (left) and cats (right). Our goal is to construct a feature space where automated classification of these images can be efficiently computed.

Figure 5.1 shows that the properties measured can be used as a good set of features for clustering and classification purposes. Specifically, the three iris varieties are well separated in this feature space. The setosa iris is most distinctive in its feature profile, while the versicolor and virginica have a small overlap among the samples taken. For this data set, machine learning is certainly not required to generate a good classification scheme. However, data generally does not so readily reduce down to simple two- and three-dimensional visual cues. Rather, decisions about clustering in feature space occur with many more variables, thus requiring the aid of computational methods to provide good classification schemes.

As a second example, we consider in Fig. 5.2 a selection from an image database of 80 dogs and 80 cats. A specific goal for this data set is to develop an automated classification method whereby the computer can distinguish between cats and dogs. In this case, the data for each cat and dog is the 64×64 pixel space of the image. Thus each image has 4096 measurements, in contrast to the 4 measurements for each example in the iris data set. Like eigenfaces, we will use the SVD to extract the dominant correlations among the images. The following code loads the data and performs a singular value decomposition on the data after the mean is subtracted. The SVD produces an ordered set of modes characterizing the correlation between all the dog and cat images. Figure 5.3 shows the first four SVD modes of the 160 images (80 dogs and 80 cats).

Code 5.2: Features of dogs and cats.

```

||| load dogData.mat
||| load catData.mat
CD=double([dog cat]);
[u,s,v]=svd(CD-mean(CD(:)), 'econ');

```

The original image space, or pixel space, is only one potential set of data to work with. The data can be transformed into a wavelet representation where

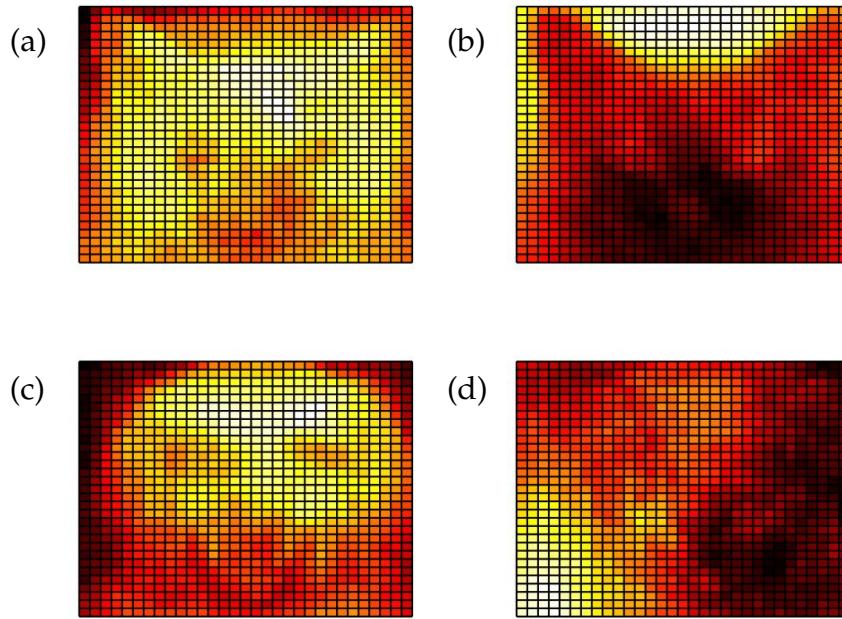


Figure 5.3: First four features (a)-(d) generated from the SVD of the 160 images of dogs and cats, i.e. these are the first four columns of the \mathbf{U} matrix of the SVD. Typical cat and dog images are shown in Fig. 5.2. Note that the first two modes (a) and (b) show that the triangular ears are important features when images are correlated. This is certainly a distinguishing feature for cats, while dogs tend to lack this feature. Thus in feature space, cats generally add these two dominant modes to promote this feature while dogs tend to subtract these features to remove the triangular ears from their representation.

edges of the images are emphasized. The following code loads in the images in their wavelet representation and computes a new low-rank embedding space.

Code 5.3: Wavelet features of dogs and cats.

```

load catData_w.mat
load dogData_w.mat
CD2=[dog_wave cat_wave];
[u2,s2,v2]=svd(CD2-mean(CD2(:)), 'econ');

```

The equivalent of Fig. 5.3 in wavelet space is shown in Fig. 5.4. Note that the wavelet representation helps emphasize many key features such as the eyes, nose, and ears, potentially making it easier to make a classification decision. Generating a feature space that enables classification is critical for constructing effective machine learning algorithms.

Whether using the image space directly or a wavelet representation, Figs. 5.3 and 5.4 respectively, the goal is to project the data onto the feature space generated by each. A good feature space helps find distinguishing features that

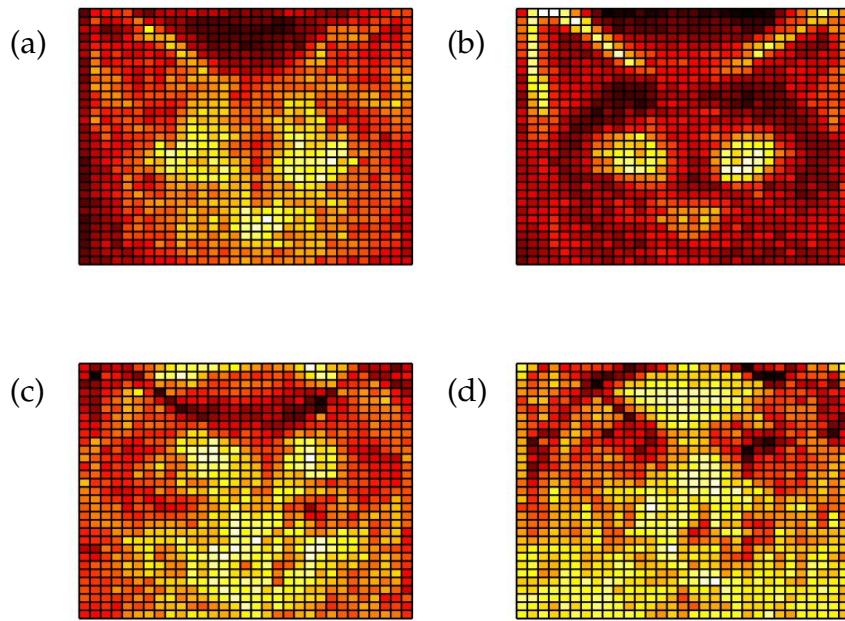


Figure 5.4: First four features (a)-(d) generated from the SVD of the 160 images of dogs and cats in the wavelet domain. As before, the first two modes (a) and (b) show that the triangular ears are important. This is an alternative representation of the dogs and cats that can help better classify dogs versus cats.

allow one to perform a variety of tasks that may include clustering, classification, and prediction. The importance of each feature to an individual image is given by the V matrix in the SVD. Specifically, each column of V determines the loading, or weighting, of each feature onto a specific image. Histograms of these loadings can then be used to visualize how distinguishable cats and dogs are from each other by each feature (See Fig. 5.5). The following code produces a histogram of the distribution of loadings for the dogs and the cats (first 80 images versus second 80 images respectively).

Code 5.4: Feature histograms of dogs and cats.

```

xbin=linspace(-0.25,0.25,20);
for j=1:4
    subplot(4,2,2*j-1)
    pdf1=hist(v(1:80,j),xbin)
    pdf2=hist(v(81:160,j),xbin)
    plot(xbin,pdf1,xbin,pdf2,'Linewidth',[2])
end

```

Figure 5.5 shows the distribution of loading scores for the first four modes for both the raw images as well as the wavelet transformed images. For both the sets of images, the distribution of loadings on the second mode clearly shows a strong separability between dogs and cats. The wavelet processed

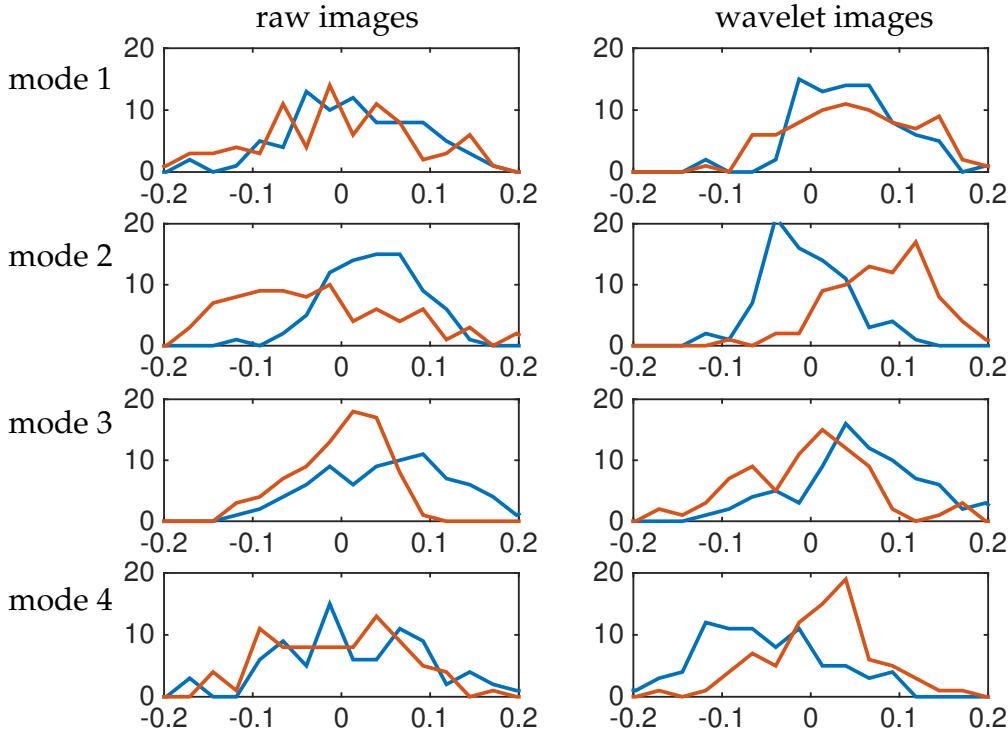


Figure 5.5: Histogram of the distribution of loadings for dogs (blue) and cats (red) on the first four dominant SVD modes. The left panel shows the distributions for the raw images (See Fig. 5.3) while the right panels show the distribution for wavelet transformed data (See Fig. 5.4. The loadings come from the columns of the V matrix of the SVD. Note the good separability between dogs and cats using the second mode.

images also show a nice separability on the fourth mode. Note that the first mode for both shows very little discrimination between the distributions and is thus not useful for classification and clustering objectives.

Features that provide strong separability between different types of data (e.g. dogs and cats) are typically exploited for machine learning tasks. This simple example shows that feature engineering is a process whereby an initial data exploration is used to help identify potential pre-processing methods. These features can then help the computer identify highly distinguishable features in a higher-dimensional space for accurate clustering, classification and prediction. As a final note, consider Fig. 5.6 which projects the dog and cat data onto the first three PCA modes (SVD modes) discovered from the raw images or their wavelet transformed counterparts. As will be seen later, the wavelet transformed images provide a higher degree of separability, and thus improved classification.

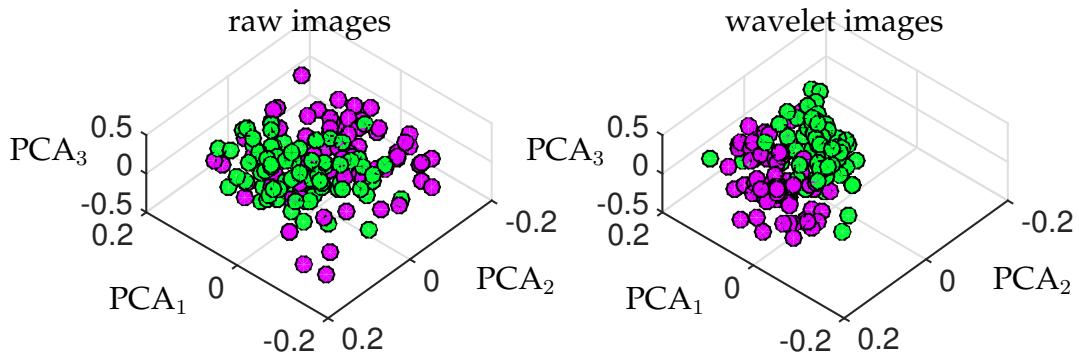


Figure 5.6: Projection of dogs (green) and cats (magenta) into feature space. Note that the raw images and their wavelet counterparts produce different embeddings of the data. Both exhibit clustering around their labeled states of dog and cat. This is exploited in the learning algorithms that follow. The wavelet images are especially good for clustering and classification as this feature space more easily separates the data.

5.2 Supervised versus unsupervised learning

As previously stated, the goal of data mining and machine learning is to construct and exploit the intrinsic low-rank feature space of a given data set. Good feature engineering and feature extraction algorithms can then be used to learn classifiers and predictors for the data. Two dominant paradigms exist for learning from data: *supervised methods* and *unsupervised methods*. Supervised data-mining algorithms are presented with labeled data sets, where the training data is labeled by a teacher/expert/supervisor. Thus examples of the input and output of a desired model are explicitly given, and regression methods are used to find the best model via optimization for the given labeled data. This model is then used for prediction and classification using new data. There are important variants of this basic architecture which include semi-supervised learning, active learning and reinforcement learning. For unsupervised learning algorithms, no training labels are given so that an algorithm must find patterns in the data in a principled way in order to determine how to cluster and classify new data. In unsupervised learning, the goal itself may be to discover patterns in the data embedded in the low-rank subspaces so that feature engineering or feature extraction can be used to build an appropriate model.

To illustrate the difference in supervised versus unsupervised learning, consider Fig. 5.7. This shows a scatter plot of two Gaussian distributions. In one case, the data is well separated so that their means are sufficiently far apart and two distinct clusters are observed. In the second case, the two distributions are brought close together so that separating the data is a challenging task. The goal of unsupervised learning is to discover clusters in the data. This is a trivial task by visual inspection, provided the two distributions are sufficiently sepa-

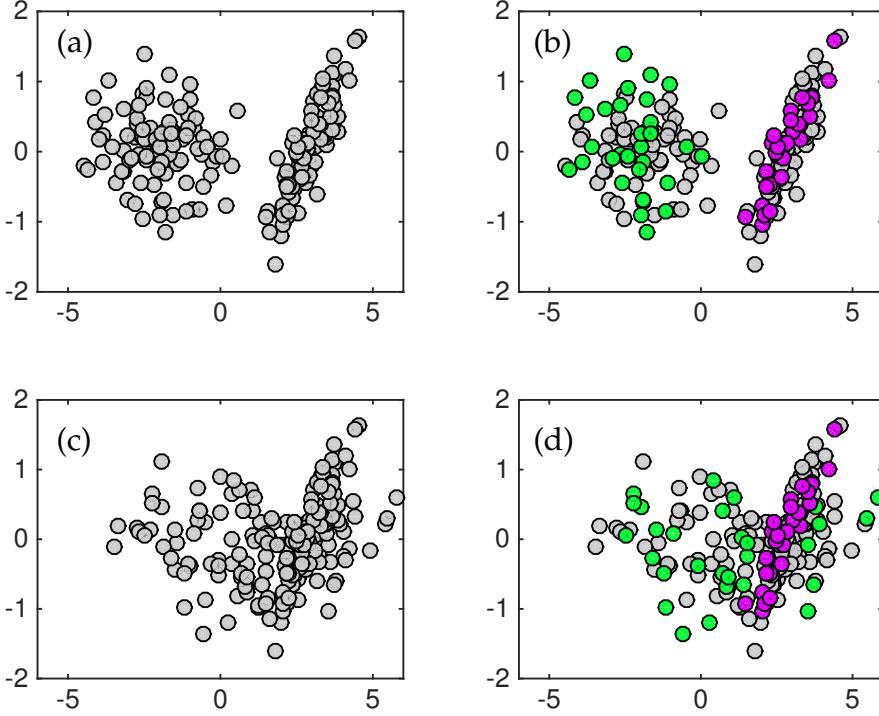


Figure 5.7: Illustration of unsupervised versus supervised learning. In the left panels (a) and (c), unsupervised learning attempts to find clusters for the data in order to classify them into two groups. For well separated data (a), the task is straightforward and labels can easily be produced. For overlapping data (c), it is a very difficult task for an unsupervised algorithm to accomplish. In the right panels (b) and (d), supervised learning provides a number of labels: green balls and magenta balls. The remaining unlabeled data is then classified as green or magenta. For well separated data (b), labeling data is easy, while overlapping data presents significant challenge.

rated. Otherwise, it becomes very difficult to distinguish clusters in the data. Supervised learning provides labels for some of the data. In this case, points are either labeled with green dots or magenta dots and the task is to classify the unlabeled data (grey dots) as either green or magenta. Much like the unsupervised architecture, if the statistical distributions that produced the data are well separated, then using the labels in combination with the data provides a simple way to classify all the unlabeled data points. Supervised algorithms also perform poorly if the data distributions have significant overlap.

Supervised and unsupervised learning can be stated mathematically. Let

$$\mathcal{D} \subset \mathbb{R}^n \quad (5.1)$$

so that \mathcal{D} is an open bounded set of dimension n . Further, let

$$\mathcal{D}' \subset \mathcal{D}. \quad (5.2)$$

The goal of classification is to build a classifier labeling all data in \mathcal{D} given data from \mathcal{D}' .

To make our problem statement more precise, consider a set of data points $\mathbf{x}_j \in \mathbb{R}^n$ and labels y_j for each point where $j = 1, 2, \dots, m$. Labels for the data can come in many forms, from numeric values, including integer labels, to text strings. For simplicity, we will label the data in a binary way as either plus or minus one so that $y_j \in \{\pm 1\}$.

For unsupervised learning, the following inputs and outputs are then associated with learning a classification task

Input

$$\text{data } \{\mathbf{x}_j \in \mathbb{R}^n, j \in Z := \{1, 2, \dots, m\}\} \quad (5.3a)$$

Output

$$\text{labels } \{y_j \in \{\pm 1\}, j \in Z\}. \quad (5.3b)$$

Thus the mathematical framing of unsupervised learning is focused on producing labels y_j for all the data. Generally, the data \mathbf{x}_j used for training the classifier is from \mathcal{D}' . The classifier is then more broadly applied, i.e. it generalizes, to the open bounded domain \mathcal{D} . If the data used to build a classifier only samples a small portion of the larger domain, then it is often the case that the classifier will not generalize well.

Supervised learning provides labels for the training stage. The inputs and outputs for this learning classification task can be stated as follows

Input

$$\text{data } \{\mathbf{x}_j \in \mathbb{R}^n, j \in Z := \{1, 2, \dots, m\}\} \quad (5.4a)$$

$$\text{labels } \{y_j \in \{\pm 1\}, j \in Z' \subset Z\} \quad (5.4b)$$

Output

$$\text{labels } \{y_j \in \{\pm 1\}, j \in Z\}. \quad (5.4c)$$

In this case, a subset of the data is labeled and the missing labels are provided for the remaining data. Technically speaking, this is a semi-supervised learning task since some of the training labels are missing. For supervised learning, all the labels are known in order to build the classifier on \mathcal{D}' . The classifier is then applied to \mathcal{D} . As with unsupervised learning, if the data used to build a classifier only samples a small portion of the larger domain, then it is often the case that the classifier will not generalize well.

For the data sets considered in our feature selection and data mining section, we can consider in more detail the key components required to build a

classification model: \mathbf{x}_j , \mathbf{y}_j , \mathcal{D} and \mathcal{D}' . The Fisher iris data of Fig. 5.1 is a classic example for which we can detail these quantities. We begin with the data collected

$$\mathbf{x}_j = \{\text{sepal length, sepal width, petal length, petal width}\}. \quad (5.5)$$

Thus each iris measurement contains four data fields, or features, for our analysis. The labels can be one of the following

$$\mathbf{y}_j = \{\text{setosa, versicolor, virginica}\}. \quad (5.6)$$

In this case the labels are text strings, and there are three of them. Note that in our formulation of supervised and unsupervised learning, there were only two outputs (binary) which were labeled either ± 1 . Generally, there can be many labels, and they are often text strings. Finally, there is the domain of the data. For this case

$$\mathcal{D}' \in \{150 \text{ iris samples: 50 setosa, 50 versicolor, and 50 virginica}\} \quad (5.7)$$

and

$$\mathcal{D} \in \{\text{the universe of setosa, versicolor and virginica irises}\}. \quad (5.8)$$

We can similarly assess the dog and cat data as follows:

$$\mathbf{x}_j = \{64 \times 64 \text{ image= 4096 pixels}\} \quad (5.9)$$

where each dog and cat is labeled as

$$\mathbf{y}_j = \{\text{dog, cat}\} = \{1, -1\}. \quad (5.10)$$

In this case the labels are text strings which can also be translated to numeric values. This is consistent with our formulation of supervised and unsupervised learning where there are only two outputs (binary) labeled either ± 1 . Finally, there is the domain of the data which is

$$\mathcal{D}' \in \{160 \text{ image samples: 80 dogs and 80 cats}\} \quad (5.11)$$

and

$$\mathcal{D} \in \{\text{the universe of dogs and cats}\}. \quad (5.12)$$

Supervised and unsupervised learning methods aim to either create algorithms for classification, clustering, or regression. The discussion above is a general strategy for classification. The previous chapter discusses regression architectures. For both tasks, the goal is to build a model from data on \mathcal{D}' that can generalize to \mathcal{D} . As already shown in the preceding chapter on regression, generalization can be very difficult and cross-validation strategies are critical.

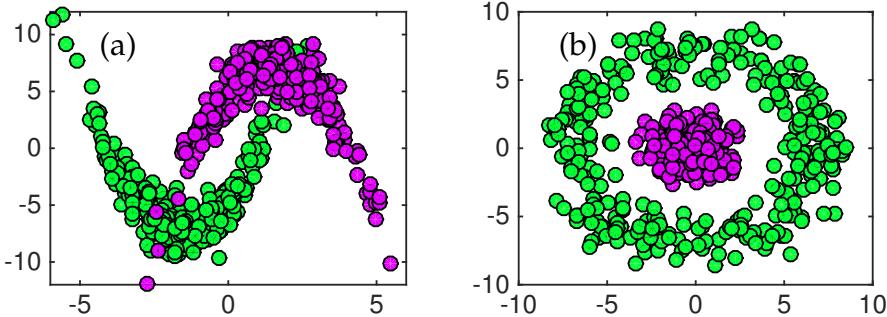


Figure 5.8: Classification and regression models for data can be difficult when the data have nonlinear functions which separate them. In this case, the function separating the green and magenta balls can be difficult to extract. Moreover, if only a small sample of the data \mathcal{D}' is available, then a generalizable model may be impossible to construct for \mathcal{D} . The left data set (a) represents two half-moon shapes that are just superimposed while the concentric rings in (b) requires a circle as a separation boundary between the data. Both are challenging to produce.

Deep neural networks, which are state-of-the-art machine learning algorithms for regression and classification, often have difficulty generalizing. Creating strong generalization schemes is at the forefront of machine learning research.

Some of the difficulties in generalization can be illustrated in Fig. 5.8. These data sets, although easily classified and clustered through visual inspection can be difficult for many regression and classification schemes. Essentially, the boundary between the data forms a nonlinear manifold that is often difficult to characterize. Moreover, if the sampling data \mathcal{D}' only captures a portion of the manifold, then a classification or regression model will almost surely fail in characterizing \mathcal{D} . These are also only two-dimensional depictions of a classification problem. It is not difficult to imagine how complicated such data embeddings can be in higher dimensional space. Visualization in such cases is essentially impossible and one must rely on algorithms to extract the meaningful boundaries separating data. What follows in this chapter and the next are methods for classification and regression given data on \mathcal{D}' that may or may not be labelled. There is quite a diversity of mathematical methods available for performing such tasks.

5.3 Unsupervised learning: k -means clustering

A variety of supervised and unsupervised algorithms will be highlighted in this chapter. We will start with one of the most prominent unsupervised algorithms in use today: k -means clustering. The k -means algorithm assumes one is given a set of vector valued data with the goal of partitioning m observations

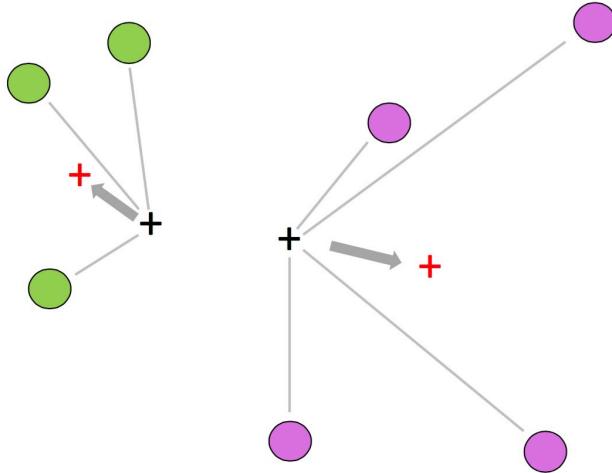


Figure 5.9: Illustration of the k -means algorithm for $k = 2$. Two initial starting values of the mean are given (black +). Each point is labeled as belonging to one of the two means. The green balls are thus labeled as part of the cluster with the left + and the magenta balls are labeled as part of the right +. Once labeled, the mean of the two clusters is recomputed (red +). The process is repeated until the means converge.

into k clusters. Each observation is labeled as belonging to a cluster with the nearest mean, which serves as a proxy (prototype) for that cluster. This results in a partitioning of the data space into Voronoi cells.

Although the number of observations and dimension of the system are known, the number of partitions k is generally unknown and must also be determined. Alternatively, the user simply chooses a number of clusters to extract from the data. The k -means algorithm is iterative, first assuming initial values for the mean of each cluster and then updating the means until the algorithm has converged. Figure 5.9 depicts the update rule of the k -means algorithm. The algorithm proceeds as follows: (i) given initial values for k distinct means, compute the distance of each observation \mathbf{x}_j to each of the k means. (ii) Label each observation as belonging to the nearest mean. (iii) Once labeling is completed, find the *center-of-mass* (mean) for each group of labeled points. These new means are then used to start back at step (i) in the algorithm. This is a heuristic algorithm that was first proposed by Stuart Lloyd in 1957 [339], although it was not published until 1982.

The k -means objective can be stated formally in terms of an optimization problem. Specifically, the following minimization describes this process

$$\operatorname{argmin}_{\boldsymbol{\mu}_j} \sum_{j=1}^k \sum_{\mathbf{x}_j \in \mathcal{D}'_j} \|\mathbf{x}_j - \boldsymbol{\mu}_j\|^2 \quad (5.13)$$

where the $\boldsymbol{\mu}_j$ denote the mean of the j th cluster and \mathcal{D}'_j denotes the subdomain of data associated with that cluster. This minimizes the within-cluster sum of

squares. In general, solving the optimization problem as stated is NP -hard, making it computationally intractable. However, there are a number of heuristic algorithms that provide good performance despite not having a guarantee that they will converge to the globally optimal solution.

Cross-validation of the k -means algorithm, as well as any machine learning algorithm, is critical for determining its effectiveness. Without labels the cross validation procedure is more nuanced as there is no ground truth to compare with. The cross-validation methods of the last section, however, can still be used to test the robustness of the classifier to different sub-selections of the data through k -fold cross-validation. The following portions of code generate Lloyd's algorithm for k -means clustering. We first consider making two clusters of data and partitioning the data into a training and test set.

Code 5.5: k -means data generation.

```
% training & testing set sizes
n1=100; % training set size
n2=50; % test set size

% random ellipse 1 centered at (0,0)
x=randn(n1+n2,1); y=0.5*randn(n1+n2,1);

% random ellipse 2 centered at (1,-2) and rotated by theta
x2=randn(n1+n2,1)+1; y2=0.2*randn(n1+n2,1)-2; theta=pi/4;
A=[cos(theta) -sin(theta); sin(theta) cos(theta)];
x3=A(1,1)*x2+A(1,2)*y2; y3=A(2,1)*x2+A(2,2)*y2;
subplot(2,2,1)
plot(x(1:n1),y(1:n1),'ro'), hold on
plot(x3(1:n1),y3(1:n1),'bo')

% training set: first 200 of 240 points
X1=[x3(1:n1) y3(1:n1)];
X2=[x(1:n1) y(1:n1)];

Y=[X1; X2]; Z=[ones(n1,1); 2*ones(n1,1)];

% test set: remaining 40 points
x1test=[x3(n1+1:end) y3(n1+1:end)];
x2test=[x(n1+1:end) y(n1+1:end)];
```

Figure 5.11 shows the data generated from two distinct Gaussian distributions. In this case, we have ground truth data to check the k -means clustering against. In general, this is not the case. The Lloyd algorithm guesses the number of clusters and the initial cluster means and then proceeds to update them in an iterative fashion. k -means is sensitive to the initial guess and many modern versions of the algorithm also provide principled strategies for initialization.

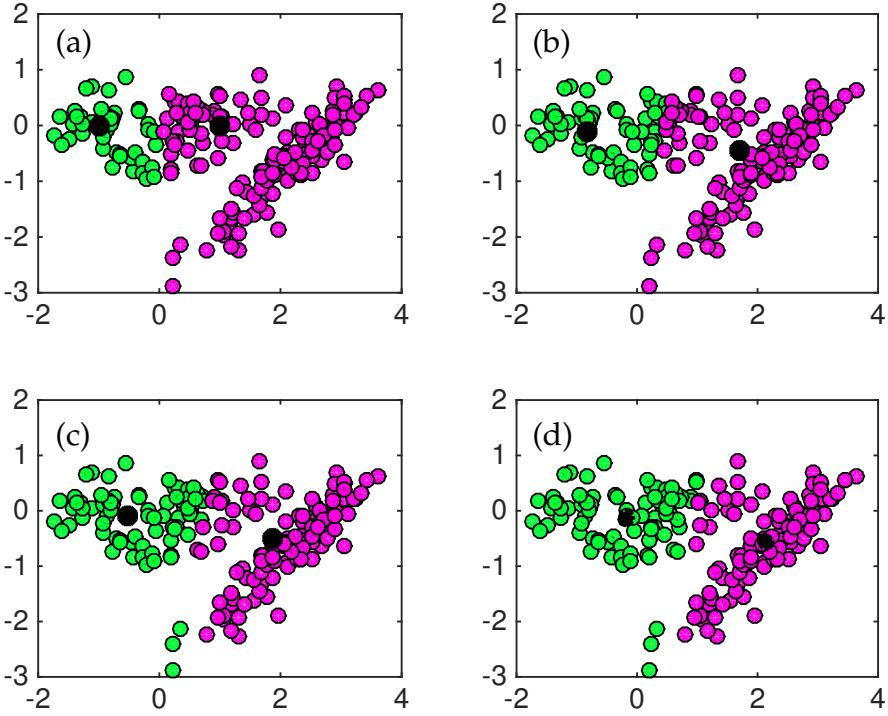


Figure 5.10: Illustration of the k -means iteration procedure based upon Lloyd's algorithm [339]. Two clusters are sought so that $k = 2$. The initial guesses (black circles in panel (a)) are used to initially label all the data according to their distance from each initial guess for the mean. The means are then updated by computing the means of the newly labeled data. This two-stage heuristic converges after approximately four iterations.

Code 5.6: Lloyd algorithm for k -means.

```

g1=[-1 0]; g2=[1 0]; % Initial guess
for j=1:4
    class1=[]; class2=[];
    for jj=1:length(Y)
        d1=norm(g1-Y(jj,:));
        d2=norm(g2-Y(jj,:));
        if d1<d2
            class1=[class1; [Y(jj,1) Y(jj,2) ]];
        else
            class2=[class2; [Y(jj,1) Y(jj,2) ]];
        end
    end
    g1=[mean(class1(1:end,1)) mean(class1(1:end,2))];
    g2=[mean(class2(1:end,1)) mean(class2(1:end,2))];
end

```

Figure 5.10 shows the iterative procedure of the k -means clustering. The

two initial guesses are used to initially label all the data points (Fig. 5.10(a)). New means are computed and the data relabeled. After only four iterations, the clusters converge. This algorithm was explicitly developed here to show how the iteration procedure rapidly provides an unsupervised labeling of all of the data. MATLAB has a built in k -means algorithm that only requires a data matrix and the number of clusters desired. It is simple to use and provides a valuable diagnostic tool for data. The following code uses the MATLAB command `mean` and also extracts the *decision line* generated from the algorithm separating the two clusters.

Code 5.7: k -means using MATLAB.

```
% kmeans code
[ind,c]=kmeans(Y,2);
plot(c(1,1),c(1,2),'k*','Linewidth',[2])
plot(c(2,1),c(2,2),'k*','Linewidth',[2])

midx=(c(1,1)+c(2,1))/2; midy=(c(1,2)+c(2,2))/2;
slope=(c(2,2)-c(1,2))/(c(2,1)-c(1,1)); % rise/run
b=midy+(1/slope)*midx;
xsep=-1:0.1:2; ysep=-(1/slope)*xsep+b;

figure(1), subplot(2,2,1), hold on
plot(xsep,ysep,'k','Linewidth',[2]), axis([-2 4 -3 2])

% error on test data
figure(1), subplot(2,2,2)
plot(x(n1+1:end),y(n1+1:end),'ro'), hold on
plot(x3(n1+1:end),y3(n1+1:end),'bo')
plot(xsep,ysep,'k','Linewidth',[2]), axis([-2 4 -3 2])
```

Figure 5.11 shows the results of the k -means algorithm and depicts the decision line separating the data into two clusters. The green and magenta balls denote the true labels of the data, showing that the k -means line does not correctly extract the labels. Indeed, a supervised algorithm is more proficient in extracting the ground truth results, as will be shown later in this chapter. Regardless, the algorithm does get a majority of the data labeled correctly.

The success of k -means is based on two factors: (i) no supervision is required, and (ii) it is a fast heuristic algorithm. The example here shows that the method is not very accurate, but this is often the case in unsupervised methods as the algorithm has limited knowledge of the data. Cross-validation efforts, such as k -fold cross-validation, can help improve the model and make the unsupervised learning more accurate, but it will generally be less accurate than a supervised algorithm that has labeled data.

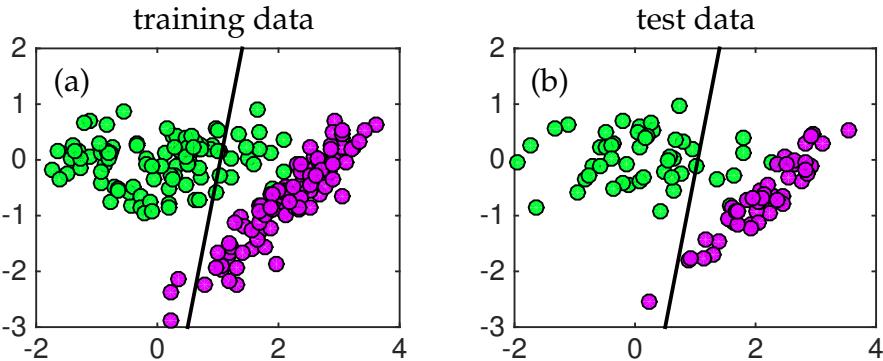


Figure 5.11: k -means clustering of the data using MATLAB's `means` command. Only the data and number of clusters need be specified. (a) The training data is used to produce a decision line (black line) separating the clusters. Note that the line is clearly not optimal. The classification line can then be used on withheld data to test the accuracy of the algorithm. For the test data, one (of 50) magenta ball would be mislabeled while six (of 50) green balls are mislabeled.

5.4 Unsupervised hierarchical clustering: Dendrogram

Another commonly used unsupervised algorithm for clustering data is a *dendrogram*. Like k -means clustering, dendrograms are created from a simple hierarchical algorithm, allowing one to efficiently visualize if data is clustered without any labeling or supervision. This hierarchical approach will be applied to the data illustrated in Fig. 5.12 where a ground truth is known. Hierarchical clustering methods are generated either from a top-down or a bottom-up approach. Specifically, they are one of two types:

Agglomerative: Each data point x_j is its own cluster initially. The data is merged in pairs as one creates a hierarchy of clusters. The merging of data eventually stops once all the data has been merged into a single über cluster. This is the bottom-up approach in hierarchical clustering.

Divisive: In this case, all the observations x_j are initially part of a single giant cluster. The data is then recursively split into smaller and smaller clusters. The splitting continues until the algorithm stops according to a user specified objective. The divisive method can split the data until each data point is its own node.

In general, the merging and splitting of data is accomplished with a heuristic, greedy algorithm which is easy to execute computationally. The results of hierarchical clustering are usually presented in a dendrogram.

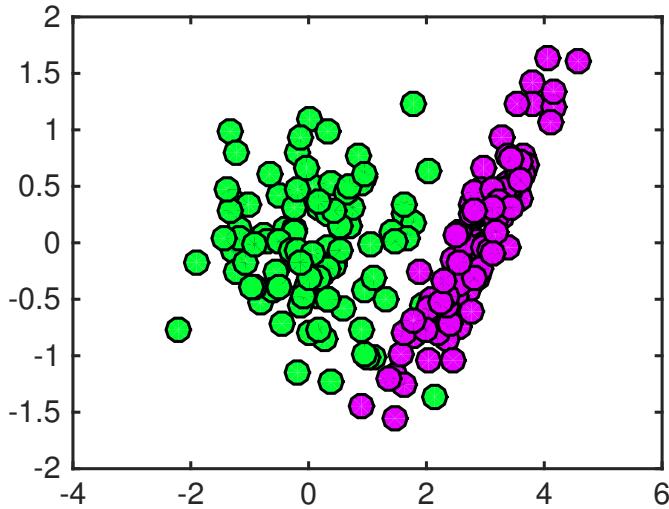


Figure 5.12: Example data used for construction of a dendrogram. The data is constructed from two Gaussian distributions (50 points each) that are easy to discern through a visual inspection. The dendrogram will produce a hierarchy that ideally would separate green balls from magenta balls.

In this section, we will focus on agglomerative hierarchical clustering and the dendrogram command from MATLAB. Like the Lloyd algorithm for k -means clustering, building the dendrogram proceeds from a simple algorithmic structure based on computing the distance between data points. Although we typically use a Euclidean distance, there are a number of important distance metrics one might consider for different types of data. Some typical distances are given as follows:

$$\text{Euclidean distance } \|\mathbf{x}_j - \mathbf{x}_k\|_2 \quad (5.14a)$$

$$\text{Squared Euclidean distance } \|\mathbf{x}_j - \mathbf{x}_k\|_2^2 \quad (5.14b)$$

$$\text{Manhattan distance } \|\mathbf{x}_j - \mathbf{x}_k\|_1 \quad (5.14c)$$

$$\text{Maximum distance } \|\mathbf{x}_j - \mathbf{x}_k\|_\infty \quad (5.14d)$$

$$\text{Mahalanobis distance } \sqrt{(\mathbf{x}_j - \mathbf{x}_k)^T \mathbf{C}^{-1} (\mathbf{x}_j - \mathbf{x}_k)} \quad (5.14e)$$

where \mathbf{C}^{-1} is the covariance matrix. As already illustrated in the previous chapter, the choice of norm can make a tremendous difference for exposing patterns in the data that can be exploited for clustering and classification.

The dendrogram algorithm is shown in Fig. 5.13. The algorithm is as follows: (i) the distance between all m data points \mathbf{x}_j is computed (the figure illustrates the use of a Euclidian distance), (ii) the closest two data points are merged into a single new data point midway between their original locations, and (iii) repeat the calculation with the new $m - 1$ points. The algorithm continues until the data has been hierarchically merged into a single data point.

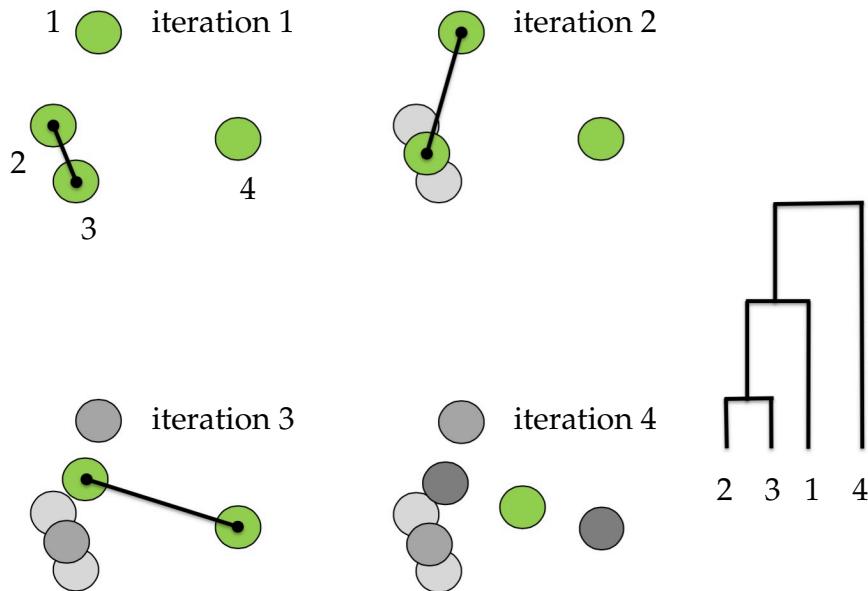


Figure 5.13: Illustration of the agglomerative hierarchical clustering scheme applied to four data points. In the algorithm, the distance between the four data points is computed. Initially the Euclidian distance between points 2 and 3 is closest. Points 2 and 3 are now merged into a point mid-way between them and the distances are once again computed. The dendrogram on the right shows how the process generates a summary (dendrogram) of the hierarchical clustering. Note that the length of the branches of the dendrogram tree are directly related to the distance between the merged points.

The following code performs a hierarchical clustering using the `dendrogram` command from MATLAB. The example we use is the same as that considered for k -means clustering. Figure 5.12 shows the data under consideration. Visual inspection shows two clear clusters that are easily discernible. As with k -means, our goal is to see how well a dendrogram can extract the two clusters.

Code 5.8: Dendrogram for unsupervised clustering.

```

||| Y3=[X1 (1:50,:); X2 (1:50,:)];
||| Y2 = pdist(Y3, 'euclidean');
||| Z = linkage(Y2, 'average');
||| thresh=0.85*max(Z (:,3));
||| [H,T,O]=dendrogram(Z,100, 'ColorThreshold', thresh);

```

Figure 5.14 shows the dendrogram associated with the data in Fig. 5.12. The structure of the algorithm shows which points are merged as well as the distance between points. The threshold command is important in labeling where each point belongs in the hierarchical scheme. By setting the threshold at different levels, there can be more or fewer clusters in the dendrogram. The following code uses the output of the dendrogram to show how the data was labeled. Recall that the first 50 data points are from the green cluster and the second 50

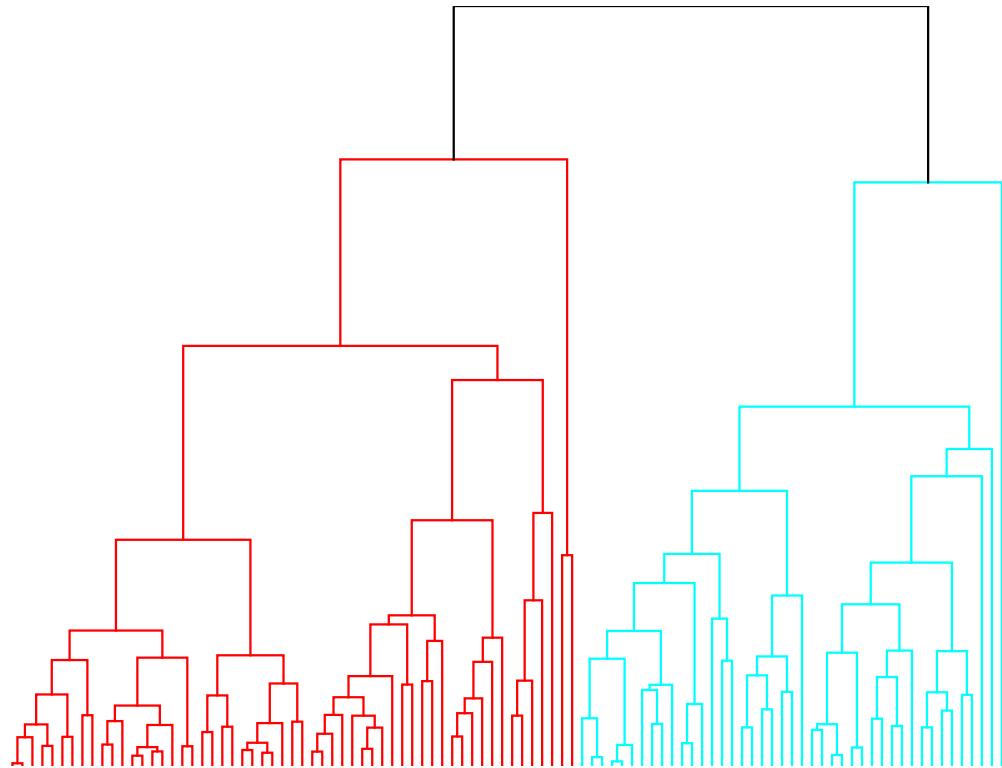


Figure 5.14: Dendrogram structure produced from the data in Fig. 5.12. The dendrogram shows which points are merged as well as the distance between points. Two clusters are generated for this level of threshold.

data points are from the magenta cluster.

Code 5.9: Dendrogram labels for cats and dogs.

```
|| bar(0), hold on
|| plot([0 100],[50 50],'r:','Linewidth',2)
|| plot([50.5 50.5],[0 100],'r:','Linewidth',2)
```

Figure 5.15 shows how the data was clustered in the dendrogram. If perfect clustering had been achieved, then the first 50 points would have been below the horizontal dotted red line while the second 50 points would have been above the horizontal dotted red line. The vertical dotted red line is the line separating the green dots on the left from the magenta dots on the right.

The following code shows how a greater number of clusters are generated by adjusting the threshold in the `dendrogram` command. This is equivalent to setting the number of clusters in k -means to something greater than two. Recall that one rarely has a ground truth to compare with when doing unsupervised clustering, so tuning the threshold becomes important.

```
|| thresh=0.25*max(Z(:,3));
|| [H,T,O]=dendrogram(Z,100,'ColorThreshold',thresh);
```

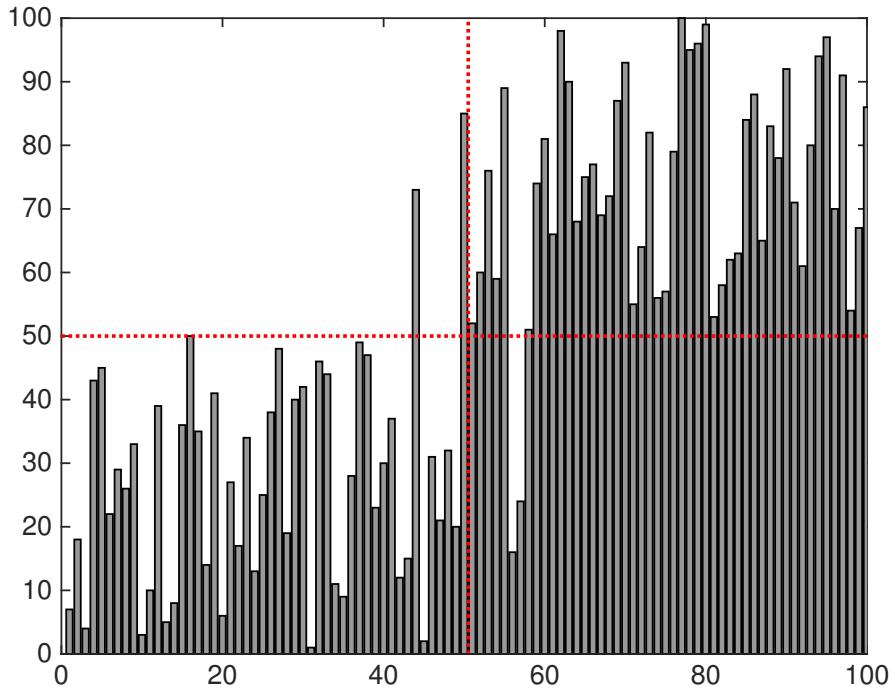


Figure 5.15: Clustering outcome from dendrogram routine. This is a summary of Fig. 5.14, showing how each of the points was clustered through the distance metric. The horizontal red dotted line shows where the ideal separation should occur. The first 50 points (green dots of Fig. 5.12) should be grouped so that they are below the red horizontal line in the lower left quadrant. The second 50 points (magenta dots of Fig. 5.12) should be grouped above the red horizontal line in the upper right quadrant. In summary, the dendrogram only misclassified two green points and two magenta points.

Figure 5.16 shows a new dendrogram with a different threshold. Note that in this case, the hierarchical clustering produces more than a dozen clusters. The tuning parameter can be seen to be critical for unsupervised clustering, much like choosing the number of clusters in k -means. In summary, both k -means and hierarchical clustering provide a method whereby data can be parsed automatically into clusters. This provides a starting point for interpretations and analysis in data mining.

5.5 Mixture models and the expectation-maximization algorithm

The third unsupervised method we consider is known as *finite mixture models*. Often the models are assumed to be Gaussian distributions in which case this method is known as *Gaussian mixture models* (GMM). The basic assumption in

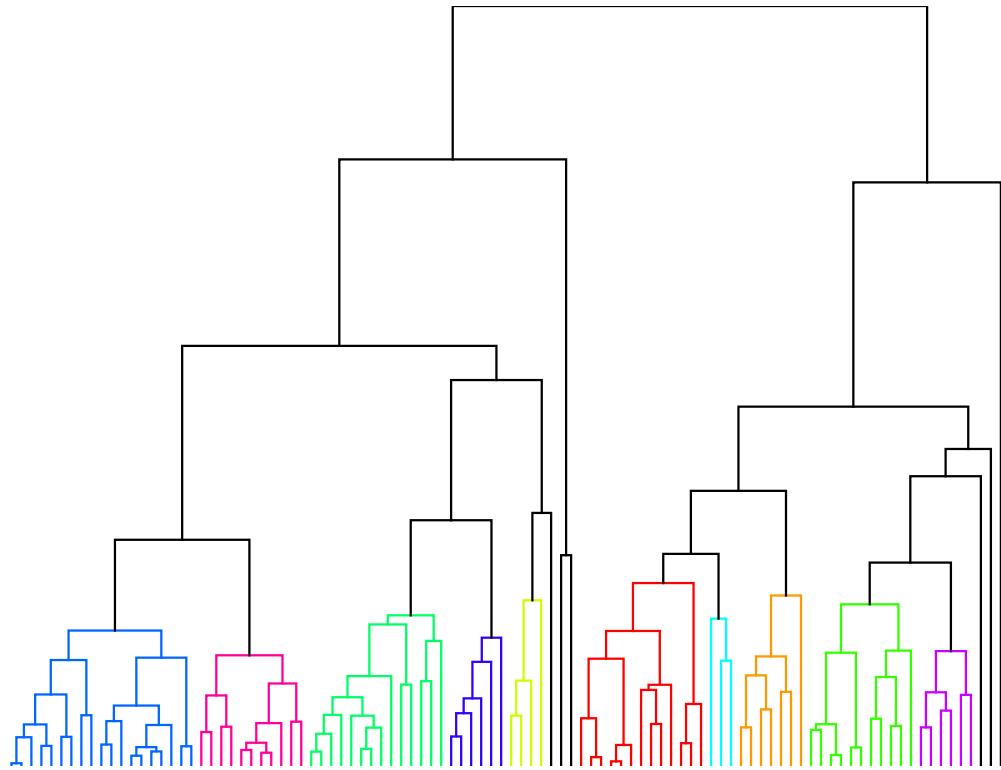


Figure 5.16: Dendrogram structure produced from the data in Fig. 5.12 with a different threshold used than in Fig. 5.14. The dendrogram shows which points are merged as well as the distance between points. In this case, more than a dozen clusters are generated.

this method is that data observations \mathbf{x}_j are a mixture of a set of k processes that combine to form the measurement. Like k -means and hierarchical clustering, the GMM model we fit to the data requires that we specify the number of mixtures k and the individual statistical properties of each mixture that best fit the data. GMMs are especially useful since the assumption that each mixture model has a Gaussian distribution implies that it can be completely characterized by two parameters: the mean and the variance.

The algorithm that enables the GMM computes the maximum-likelihood using the famous *Expectation-Maximization* (EM) algorithm of Dempster, Laird and Rubin [148]. The EM algorithm is designed to find maximum likelihood parameters of statistical models. Generally, the iterative structure of the algorithm finds a local maximum-likelihood, which estimates the true parameters that cannot be directly solved for. As with most data, the observed data involves many latent or unmeasured variables and unknown parameters. Regardless, the alternating and iterative construction of the algorithm recursively estimates the best parameters possible from an initial guess. The EM algo-

rithm proceeds like the k -means algorithm in that initial guesses for the mean and variance are given for the assumed k -distributions. The algorithm then recursively updates the weights of the mixtures versus the parameters of each mixture. One alternates between these two until convergence is achieved.

In any such iteration scheme, it is not obvious that the solution will converge, or that the solution is good, since it typically falls into a local value of the maximum-likelihood. But it can be proven that in this context it does converge, and that the derivative of the likelihood is arbitrarily close to zero at that point, which in turn means that the point is either a maximum or a saddle point [561]. In general, multiple maxima may occur, with no guarantee that the global maximum will be found. Some likelihoods also have singularities, i.e., nonsensical maxima. For example, one of the solutions that may be found by EM in a mixture model involves setting one of the components to have zero variance and the mean equal to one of the data points. Cross-validation can often alleviate some of the common pitfalls that can occur by initializing the algorithm with some bad initial guesses.

The fundamental assumption of the mixture model is that the probability density function (PDF) for observations of data \mathbf{x}_j is a weighted linear sum of a set of unknown distributions

$$f(\mathbf{x}_j, \Theta) = \sum_{p=1}^k \alpha_p f_p(\mathbf{x}_j, \Theta_p) \quad (5.15)$$

where $f(\cdot)$ is the measured PDF, $f_p(\cdot)$ is the PDF of the mixture j , and k is the total number of mixtures. Each of the PDFs $f_j(\cdot)$ is weighted by α_p ($\alpha_1 + \alpha_2 + \dots + \alpha_k = 1$) and parametrized by an unknown vector of parameters Θ_p . To state the objective of mixture models more precisely then: *Given the observed PDF $f(\mathbf{x}_j, \Theta)$, estimate the mixture weights α_p and the parameters of the distribution Θ_p .* Note that Θ is a vector containing all the parameters Θ_p . Making this task somewhat easier is the fact that we assume the form of the PDF distribution $f_p(\cdot)$.

For GMM, the parameters in the vector Θ_p are known to include only two variables: the mean μ_p and variance σ_p . Moreover, the distribution $f_p(\cdot)$ is normally distributed so that (5.15) becomes

$$f(\mathbf{x}_j, \Theta) = \sum_{p=1}^k \alpha_p \mathcal{N}_p(\mathbf{x}_j, \mu_p, \sigma_p). \quad (5.16)$$

This gives a much more tractable framework since there are now a limited set of parameters. Thus once one assumes a number of mixtures k , then the task is to determine α_p along with μ_p and σ_p for each mixture. It should be noted that there are many other distributions besides Gaussian that can be imposed, but

GMM are common since without prior knowledge, an assumption of Gaussian distribution is typically assumed.

An estimate of the parameter vector Θ can be computed using the *maximum likelihood estimate* (MLE) of Fisher. The MLE computes the value of Θ from the roots of

$$\frac{\partial L(\Theta)}{\partial \Theta} = 0 \quad (5.17)$$

where the log-likelihood function L is

$$L(\Theta) = \sum_{j=1}^n \log f(\mathbf{x}_j | \Theta) \quad (5.18)$$

and the sum is over all the n data vectors \mathbf{x}_j . The solution to this optimization problem, i.e. when the derivative is zero, produces a local maximizer. This maximizer can be computed using the EM algorithm since derivatives cannot be explicitly computed without an analytic form.

The EM algorithm starts by assuming an initial estimate (guess) of the parameter vector Θ . This estimate can be used to estimate

$$\tau_p(\mathbf{x}_j, \Theta) = \frac{\alpha_p f_p(\mathbf{x}_j, \Theta_p)}{f(\mathbf{x}_j, \Theta)} \quad (5.19)$$

which is the posterior probability of component membership of \mathbf{x}_j in the p th distribution. In other words, does \mathbf{x}_j belong to the p th mixture? The E-step of the EM algorithm uses this posterior to compute memberships. For GMM, the algorithm proceeds as follows: Given an initial parametrization of Θ and α_p , compute

$$\tau_p^{(k)}(\mathbf{x}_j) = \frac{\alpha_p^{(k)} \mathcal{N}_p(\mathbf{x}_j, \mu_p^{(k)}, \sigma_p^{(k)})}{\mathcal{N}(\mathbf{x}_j, \Theta^{(k)})}. \quad (5.20)$$

With an estimated posterior probability, the M-step of the algorithm then updates the parameters and mixture weights

$$\alpha_p^{(k+1)} = \frac{1}{n} \sum_{j=1}^n \tau_p^{(k)}(\mathbf{x}_j) \quad (5.21a)$$

$$\mu_p^{(k+1)} = \frac{\sum_{j=1}^n \mathbf{x}_j \tau_p^{(k)}(\mathbf{x}_j)}{\sum_{j=1}^n \tau_p^{(k)}(\mathbf{x}_j)} \quad (5.21b)$$

$$\Sigma_p^{(k+1)} = \frac{\sum_{j=1}^n \tau_p^{(k)}(\mathbf{x}_j) (\mathbf{x}_j - \mu_p^{(k+1)}) (\mathbf{x}_j - \mu_p^{(k+1)})^T}{\sum_{j=1}^n \tau_p^{(k)}(\mathbf{x}_j)} \quad (5.21c)$$

where the matrix $\Sigma_p^{(k+1)}$ is the covariance matrix containing the variance parameters. The E- and M-steps are alternated until convergence within a specified

tolerance. Recall that to initialize the algorithm, the number of mixture models k must be specified and initial parametrization (guesses) of the distributions given. This is similar to the k -means algorithm where the number of clusters k is prescribed and an initial guess for the cluster centers is specified.

The GMM is popular since it simply fits k Gaussian distributions to data, which is reasonable for unsupervised learning. The GMM algorithm also has a stronger theoretical base than most unsupervised methods as both k -means and hierarchical clustering are simply defined as algorithms. The primary assumption in GMM is the number of clusters and the form of the distribution $f(\cdot)$.

The following code executes a GMM model on the second and fourth principal components of the dog and cat wavelet image data introduced previously in Figs. 5.4-5.6. Thus the features are the second and fourth columns of the right singular vector of the SVD. The `fitgmdist` command is used to extract the mixture model.

Code 5.10: Gaussian mixture model for cats versus dogs.

```
dogcat=v(:,2:2:4);
GMModel=fitgmdist(dogcat,2)
AIC= GMModel.AIC

subplot(2,2,1)
h=ezcontour(@(x1,x2)pdf(GMModel,[x1 x2]));
subplot(2,2,2)
h=ezmesh(@(x1,x2)pdf(GMModel,[x1 x2]));
```

The results of the algorithm can be plotted for visual inspection, and the parameters associated with each Gaussian are given. Specifically, the mixing proportion of each model along with the mean in each of the two dimensions of the feature space. The following is displayed to the screen.

```
Component 1:
Mixing proportion: 0.355535
Mean: -0.0290 -0.0753

Component 2:
Mixing proportion: 0.644465
Mean: 0.0758 0.0076

AIC =
-792.8105
```

The code can also produce an AIC score for how well the mixture of Gaussians explain the data. This gives a principled method for cross-validating in order to determine the number of mixtures required to describe the data.

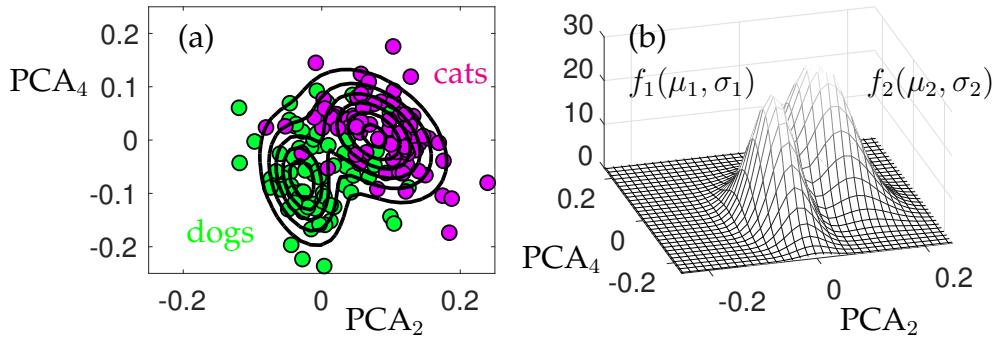


Figure 5.17: GMM fit of the second and fourth principal components of the dog and cat wavelet image data. The two Gaussians are well placed over the distinct dog and cat features as shown in (a). The PDF of the Gaussian models extracted are highlighted in (b) in arbitrary units.

Figure 5.17 shows the results of the GMM fitting procedure along with the original data of cats and dogs. The Gaussians produced from the fitting procedure are also illustrated. The `fitgmdist` command can also be used with `cluster` to label new data from the feature separation discovered by GMM.

5.6 Supervised learning and linear discriminants

We now turn our attention to supervised learning methods. One of the earliest supervised methods for classification of data was developed by Fisher in 1936 in the context of taxonomy [183]. His *linear discriminant analysis* (LDA) is still one of the standard techniques for classification. It was generalized by C. R. Rao for multi-class data in 1948 [446]. The goal of these algorithms is to find a linear combination of features that characterizes or separates two or more classes of objects or events in the data. Importantly, for this supervised technique we have labeled data which guides the classification algorithm. Figure 5.18 illustrates the concept of finding an optimal low-dimensional embedding of the data for classification. The LDA algorithm aims to solve an optimization problem to find a subspace whereby the different labeled data have clear separation between their distribution of points. This then makes classification easier because an optimal feature space has been selected.

The supervised learning architecture includes a training and withhold set of data. The withhold set is never used to train the classifier. However, the training data can be partitioned into k -folds, for instance, to help build a better classification model. The last chapter details how cross-validation should be appropriately used. The goal here is to train an algorithm that uses feature space to make a decision about how to classify data. Figure 5.18 gives a cartoon of the key idea involved in LDA. In our example, two data sets are considered

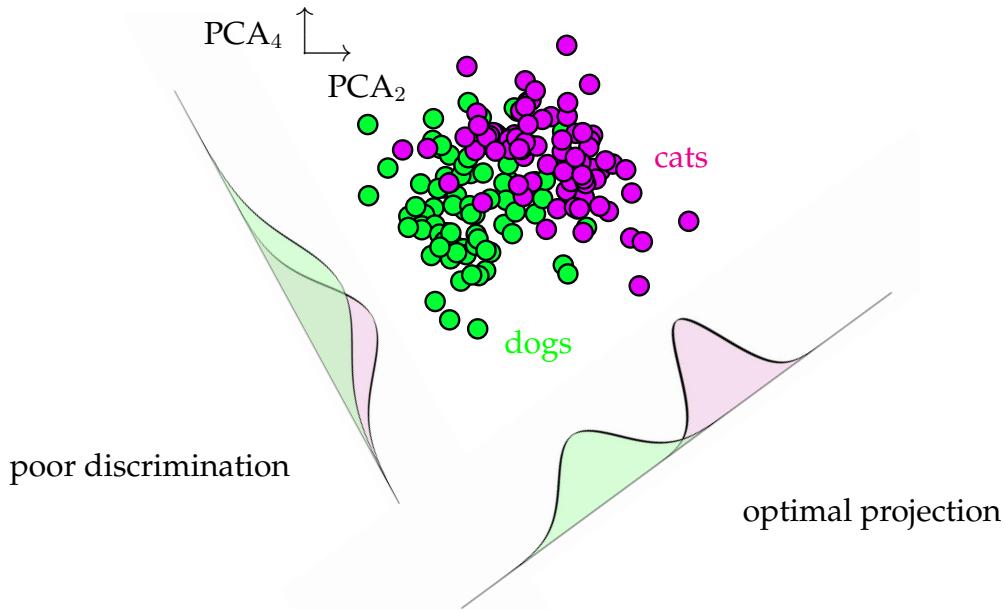


Figure 5.18: Illustration of linear discriminant analysis (LDA). The LDA optimization method produces an optimal dimensionality reduction to a decision line for classification. The figure illustrates the projection of data onto the second and fourth principal component modes of the dog and cat wavelet data considered in Fig. 5.4. Without optimization, a general projection can lead to very poor discrimination between the data. However, the LDA separates the probability distribution functions in an optimal way.

and projected onto new bases. In the left figure, the projection shows that the data is completely mixed, making it difficult to separate the data. In the right figure, which is the ideal caricature for LDA, the data are well separated with the means μ_1 and μ_2 being well apart when projected onto the chosen subspace. Thus the goal of LDA is two-fold: *find a suitable projection that maximizes the distance between the inter-class data while minimizing the intra-class data.*

For a two-class LDA, this results in the following mathematical formulation. Construct a projection w such that

$$w = \arg \max_w \frac{w^T S_B w}{w^T S_W w} \quad (5.22)$$

where the scatter matrices for between-class S_B and within-class S_W data are given by

$$S_B = (\mu_2 - \mu_1)(\mu_2 - \mu_1)^T \quad (5.23)$$

$$S_W = \sum_{j=1}^2 \sum_{\mathbf{x}} (\mathbf{x} - \mu_j)(\mathbf{x} - \mu_j)^T. \quad (5.24)$$

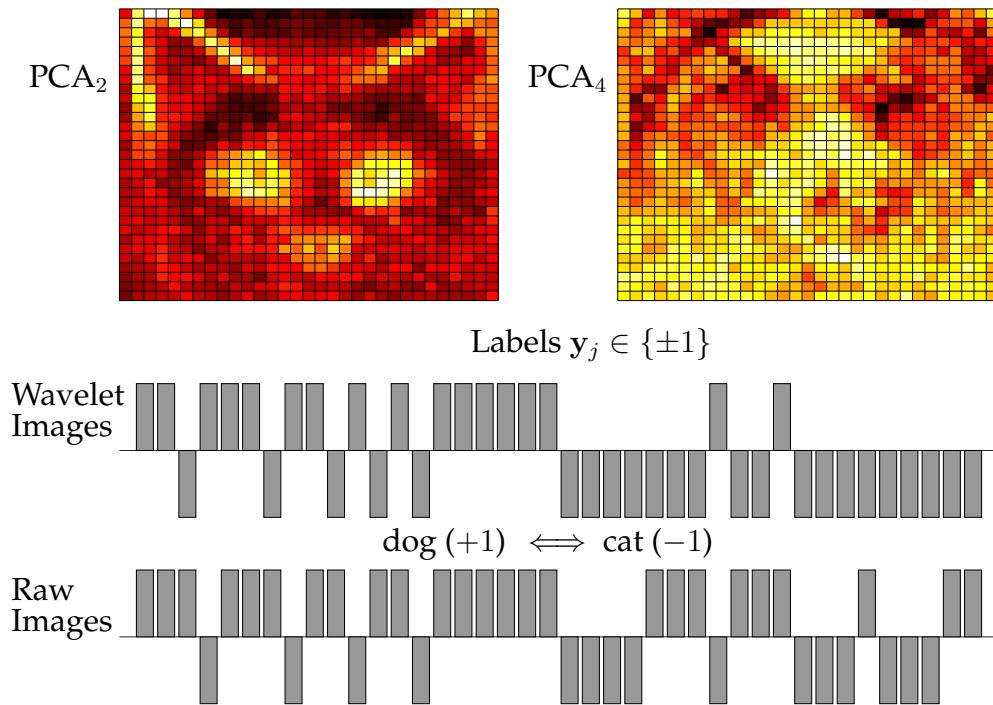


Figure 5.19: Depiction of the performance achieved for classification using the second and fourth principal component modes. The top two panels are PCA modes (features) used to build a classifier. The labels returned are either $y_j \in \{\pm 1\}$. The ground truth answer in this case should produce a vector of 20 ones followed by 20 negative ones.

These quantities essentially measure the variance of the data sets as well as the variance of the difference in the means. The criterion in (5.22) is commonly known as the generalized Rayleigh quotient whose solution can be found via the generalized eigenvalue problem

$$\mathbf{S}_B \mathbf{w} = \lambda \mathbf{S}_W \mathbf{w} \quad (5.25)$$

where the maximum eigenvalue λ and its associated eigenvector gives the quantity of interest and the projection basis. Thus, once the scatter matrices are constructed, the generalized eigenvectors can be constructed with MATLAB.

Performing an LDA analysis in MATLAB is simple. One needs only to organize the data into a training set with labels, which can then be applied to a test data set. Given a set of data \mathbf{x}_j for $j = 1, 2, \dots, m$ with corresponding labels y_j , the algorithm will find an optimal classification space as shown in Fig. 5.18. New data \mathbf{x}_k with $k = m+1, m+2, \dots, m+n$ can then be evaluated and labeled. We illustrate the classification of data using the dog and cat data set introduced in the feature section of this chapter. Specifically, we consider the dog and cat images in the wavelet domain and label them so that $y_j \in \{\pm 1\}$ ($y_j = 1$ is a dog

and $y_j = -1$ is a cat). The following code trains on the first 60 images of dogs and cats, and then tests the classifier on the remaining 20 dog and cat images. For simplicity, we train on the second and fourth principal components as these show good discrimination between dogs and cats (See Fig. 5.5).

Code 5.11: LDA analysis of dogs versus cats.

```

load catData_w.mat
load dogData_w.mat
CD=[dog_wave cat_wave];
[u,s,v]=svd(CD-mean(CD(:))) ;

xtrain=[v(1:60,2:2:4); v(81:140,2:2:4)];
label=[ones(60,1); -1*ones(60,1)];
test=[v(61:80,2:2:4); v(141:160,2:2:4)];

class=classify(test,xtrain,label);
truth=[ones(20,1); -1*ones(20,1)];
E=100-sum(0.5*abs(class-truth))/40*100

```

Note that the `classify` command in MATLAB takes in the three matrices of interest: the training data, the test data, and the labels for the training data. What is produced are the labels for the test set. One can also extract from this command the decision line for online use. Figure 5.19 shows the results of the classification on the 40 test data samples. Recall that this classification is performed using only the second and fourth PCA modes which cluster as shown in Fig. 5.18. The returned labels are either ± 1 depending on whether a cat or dog is labeled. The ground truth labels for the test data should return a $+1$ (dogs) for the first 20 test sets and a -1 (cats) for the second test set. The accuracy of classification for this realization is 82.5% (2/20 cats are mislabeled while 5/20 dogs are mislabeled). Comparing the wavelet images to the raw images we see that the feature selection in the raw images is not as good. In particular, for the same two principal components, 9/20 cats are mislabeled and 4/20 dogs are mislabeled.

Of course, the data is fairly limited and cross-validation should always be performed to evaluate the classifier. The following code runs 100 trials of the `classify` command where 60 dog and cat images are randomly selected and tested against the remaining 20 images.

Code 5.12: Cross-validation of the LDA analysis.

```

for jj=1:100;
    r1=randperm(80); r2=randperm(80);
    ind1=r1(1:60); ind2=r2(1:60)+60;
    ind1t=r1(61:80); ind2t=r2(61:80)+60;

```

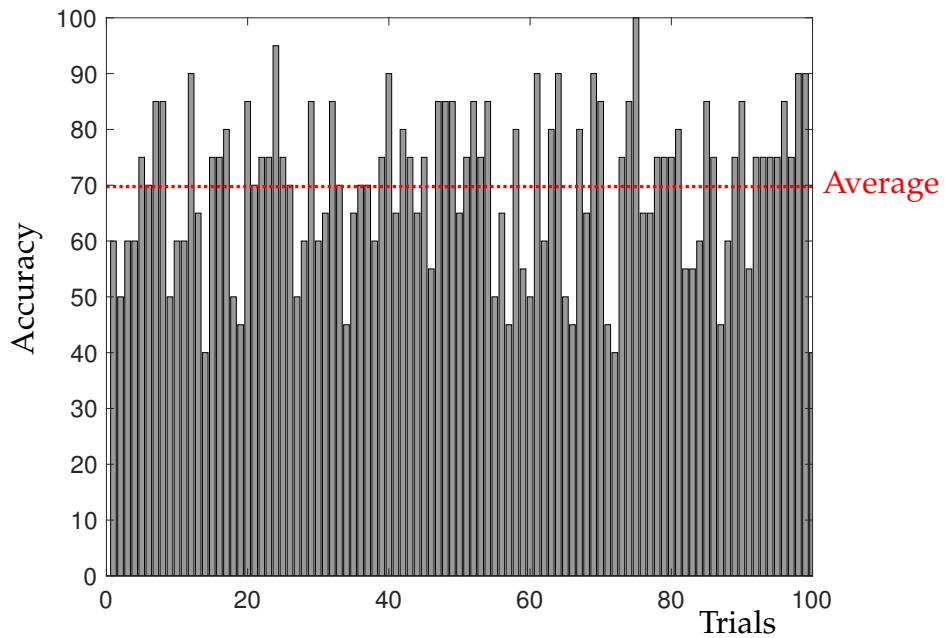


Figure 5.20: Performance of the LDA over 100 trials. Note the variability that can occur in the classifier depending on which data is selected for training and testing. This highlights the importance of cross-validation for building a robust classifier.

```

xtrain=[v(ind1,2:2:4); v(ind2,2:2:4)];
test=[v(ind1t,2:2:4); v(ind2t,2:2:4)];

label=[ones(60,1); -1*ones(60,1)];
truth=[ones(20,1); -1*ones(20,1)];
class=classify(test,xtrain,label);
E(jj)=sum(abs(class-truth))/40*100;
end

```

Figure 5.20 shows the results of the cross-validation over 100 trials. Note the variability that can occur from trial to trial. Specifically, the performance can achieve 100%, but can also be as low as 40%, which is worse than a coin flip. The average classification score (red dotted line) is around 70%. Cross-validation, as already highlighted in the regression chapter, is critical for testing and robustifying the model. Recall that the methods for producing a classifier are based on optimization and regression, so that all the cross-validation methods can be ported to the clustering and classification problem.

In addition to a linear discriminant line, a quadratic discriminant line can be found to separate the data. Indeed, the `classify` command in MATLAB allows one to not only produce the classifier, but also extract the line of separation between the data. The following commands are used to produce labels for new data as well as the discrimination line between the dogs and cats.

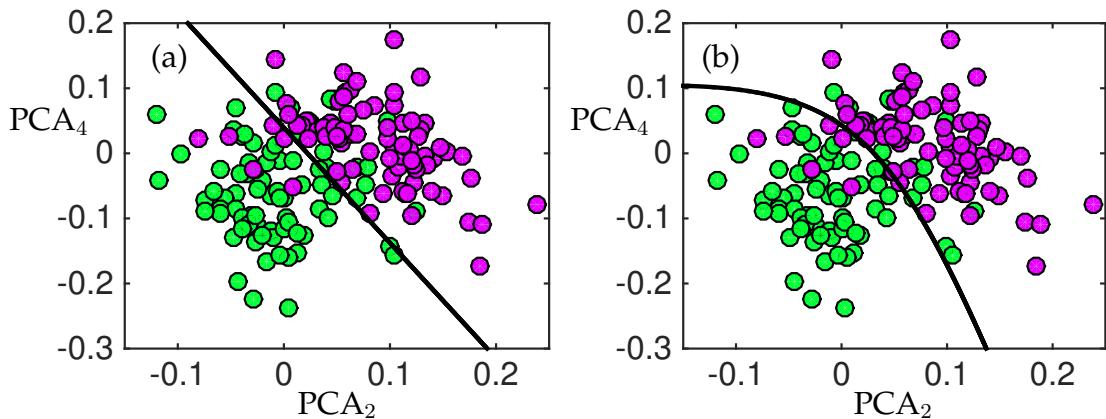


Figure 5.21: Classification line for (a) linear discriminant (LDA) and (b) quadratic discriminant (QDA) for dog (green dots) versus cat (magenta dots) data projected onto the second and fourth principal components. This two dimensional feature space allows for a good discrimination in the data. The two lines represent the best line and parabola for separating the data for a given training sample.

Code 5.13: Plotting the linear and quadratic discrimination lines.

```

subplot(2,2,1)
[class,~,~,~,coeff]=classify(test,xtrain,label);
K = coeff(1,2).const;
L = coeff(1,2).linear;
f = @(x,y) K + [x y]*L;
h2 = ezplot(f,[-.15 0.25 -.3 0.2]);
subplot(2,2,2)
[class,~,~,~,coeff]=classify(test,xtrain,label,'quadratic');
K = coeff(1,2).const;
L = coeff(1,2).linear;
Q = coeff(1,2).quadratic;
f = @(x,y) K + [x y]*L + sum(([x y]*Q) .* [x y], 2);
h2 = ezplot(f,[-.15 0.25 -.3 0.2]);

```

Figure 5.21 shows the dog and cat data along with the linear and quadratic lines separating them. This linear or quadratic fit is found in the structured variable **coeff** which is returned with **classify**. The quadratic line of separation can often offer a little more flexibility when trying to fit boundaries separating data. A major advantage of LDA based methods: they are easily interpretable and easy to compute. Thus, they are widely used across many branches of the sciences for classification of data.

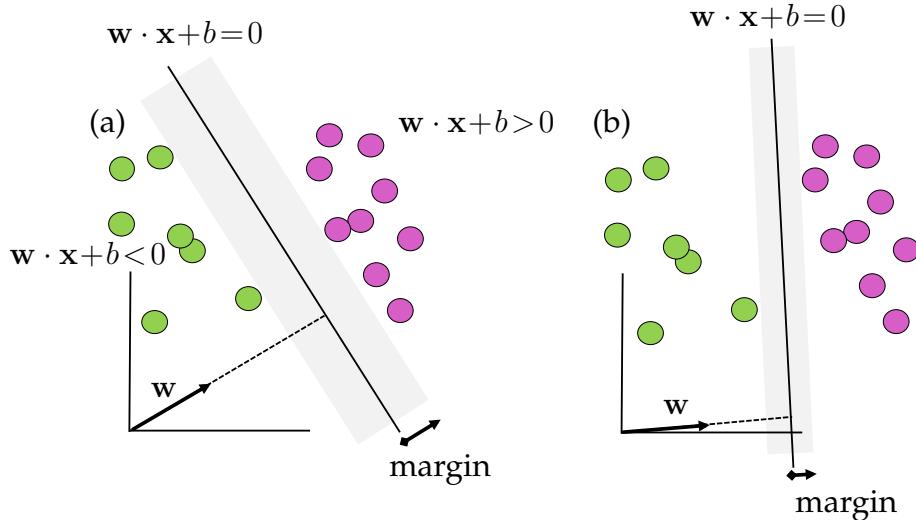


Figure 5.22: The SVM classification scheme constructs a hyperplane $w \cdot x + b = 0$ that optimally separates the labeled data. The area of the margin separating the labeled data is maximal in (a) and much less in (b). Determining the vector w and parameter b is the goal of the SVM optimization. Note that for data to the right of the hyperplane $w \cdot x + b > 0$, while for data to the left $w \cdot x + b < 0$. Thus the classification labels $y_j \in \{\pm 1\}$ for the data to the left or right of the hyperplane is given by $y_j(w \cdot x_j + b) = \text{sign}(w \cdot x_j + b)$. So only the sign of $w \cdot x + b$ needs to be determined in order to label the data. The vectors touching the edge of the gray regions of are termed the *support vectors*.

5.7 Support vector machines (SVM)

One of the most successful data mining methods developed to date is the *support vector machine* (SVM). It is a core machine learning tool that is used widely in industry and science, often providing results that are better than competing methods. Along with the *random forest* algorithm, they have been pillars of machine learning in the last few decades. With enough training data, the SVM can now be replaced with deep neural nets. But otherwise, SVM and random forest are frequently used algorithms for applications where the best classification scores are required.

The original SVM algorithm by Vapnik and Chervonenkis evolved out of the statistical learning literature in 1963, where hyperplanes are optimized to split the data into distinct clusters. Nearly three decades later, Boser, Guyon and Vapnik created nonlinear classifiers by applying the kernel trick to maximum-margin hyperplanes [70]. The current standard incarnation (soft margin) was proposed by Cortes and Vapnik in the mid-1990s [138].

Linear SVM

The key idea of the linear SVM method is to construct a hyperplane

$$\mathbf{w} \cdot \mathbf{x} + b = 0 \quad (5.26)$$

where the vector \mathbf{w} and constant b parametrize the hyperplane. Figure 5.22 shows two potential hyperplanes splitting a set of data. Each has a different value of \mathbf{w} and constant b . The optimization problem associated with SVM is to not only optimize a decision line which makes the fewest labeling errors for the data, but also optimizes the largest margin between the data, shown in the gray region of Fig. 5.22. The vectors that determine the boundaries of the margin, i.e. the vectors touching the edge of the gray regions, are termed the *support vectors*. Given the hyperplane (5.26), a new data point \mathbf{x}_j can be classified by simply computing the sign of $(\mathbf{w} \cdot \mathbf{x}_j + b)$. Specifically, for classification labels $y_j \in \{\pm 1\}$, the data to the left or right of the hyperplane is given by

$$y_j(\mathbf{w} \cdot \mathbf{x}_j + b) = \text{sign}(\mathbf{w} \cdot \mathbf{x}_j + b) = \begin{cases} +1 & \text{magenta ball} \\ -1 & \text{green ball.} \end{cases} \quad (5.27)$$

Thus the classifier y_j is explicitly dependent on the position of \mathbf{x}_j .

Critical to the success of the SVM is determining \mathbf{w} and b in a principled way. As with all machine learning methods, an appropriate optimization must be formulated. The optimization is aimed at both minimizing the number of misclassified data points as well as creating the largest margin possible. To construct the optimization objective function, we define a loss function

$$\ell(y_j, \bar{y}_j) = \ell(y_j, \text{sign}(\mathbf{w} \cdot \mathbf{x}_j + b)) = \begin{cases} 0 & \text{if } y_j = \text{sign}(\mathbf{w} \cdot \mathbf{x}_j + b) \\ +1 & \text{if } y_j \neq \text{sign}(\mathbf{w} \cdot \mathbf{x}_j + b) \end{cases}. \quad (5.28)$$

Stated more simply

$$\ell(y_j, \bar{y}_j) = \begin{cases} 0 & \text{if data is correctly labeled} \\ +1 & \text{if data is incorrectly labeled} \end{cases}. \quad (5.29)$$

Thus each mislabeled point produces a loss of unity. The training error over m data points is the sum of the loss functions $\ell(y_j, \bar{y}_j)$.

In addition to minimizing the loss function, the goal is also to make the margin as large as possible. We can then frame the linear SVM optimization problem as

$$\underset{\mathbf{w}, b}{\operatorname{argmin}} \sum_{j=1}^m \ell(y_j, \bar{y}_j) + \frac{1}{2} \|\mathbf{w}\|^2 \quad \text{subject to} \quad \min_j |\mathbf{x}_j \cdot \mathbf{w}| = 1. \quad (5.30)$$

Although this is a concise statement of the optimization problem, the fact that the loss function is discrete and constructed from ones and zeros makes it very

difficult to actually optimize. Most optimization algorithms are based on some form of gradient descent which requires smooth objective functions in order to compute derivatives or gradients to update the solution. A more common formulation then is given by

$$\operatorname{argmin}_{\mathbf{w}, b} \sum_{j=1}^m H(\mathbf{y}_j, \bar{\mathbf{y}}_j) + \frac{1}{2} \|\mathbf{w}\|^2 \quad \text{subject to} \quad \min_j |\mathbf{x}_j \cdot \mathbf{w}| = 1 \quad (5.31)$$

where α is the weighting of the loss function and $H(z) = \max(0, 1-z)$ is called a Hinge loss function. This is a smooth function that counts the number of errors in a linear way and that allows for piecewise differentiation so that standard optimization routines can be employed.

Nonlinear SVM

Although easily interpretable, linear classifiers are of limited value. They are simply too restrictive for data embedded in a high-dimensional space and which may have the structured separation as illustrated in Fig. 5.8. To build more sophisticated classification curves, the feature space for SVM must be enriched. SVM does this by included nonlinear features and then building hyperplanes in this new space. To do this, one simply maps the data into a nonlinear, higher-dimensional space

$$\mathbf{x} \mapsto \Phi(\mathbf{x}). \quad (5.32)$$

We can call the $\Phi(\mathbf{x})$ new *observables* of the data. The SVM algorithm now learns the hyperplanes that optimally split the data into distinct clusters in a new space. Thus one now considers the hyperplane function

$$f(\mathbf{x}) = \mathbf{w} \cdot \Phi(\mathbf{x}) + b \quad (5.33)$$

with corresponding labels $y_j \in \{\pm 1\}$ for each point $f(\mathbf{x}_j)$.

This simple idea, of enriching feature space by defining new functions of the data \mathbf{x} , is exceptionally powerful for clustering and classification. As a simple example, consider two dimensional data $\mathbf{x} = (x_1, x_2)$. One can easily enrich the space by considering polynomials of the data.

$$(x_1, x_2) \mapsto (z_1, z_2, z_3) := (x_1, x_2, x_1^2 + x_2^2). \quad (5.34)$$

This gives a new set of polynomial coordinates in x_1 and x_2 that can be used to embed the data. This philosophy is simple: by embedding the data in a higher dimensional space, it is much more likely to be separable by hyperplanes. As a simple example, consider the data illustrated in Fig. 5.8(b). A linear classifier (or hyperplane) in the x_1 - x_2 plane will clearly not be able to separate the data.

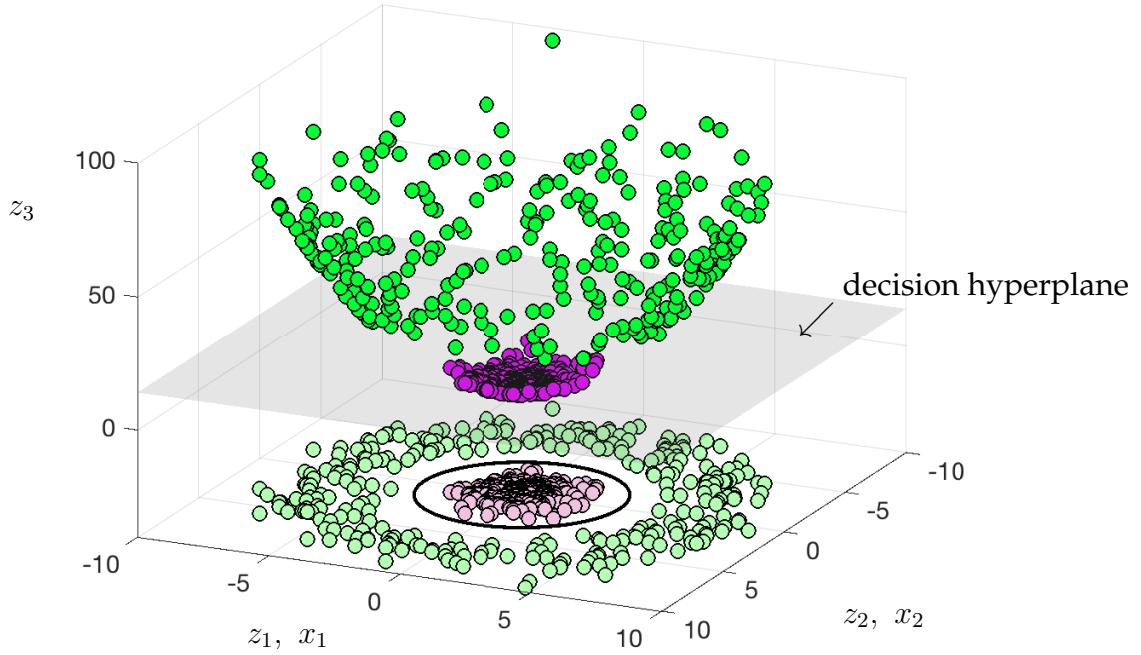


Figure 5.23: The nonlinear embedding of Fig. 5.8(b) using the variables $(x_1, x_2) \mapsto (z_1, z_2, z_3) := (x_1, x_2, x_1^2 + x_2^2)$ in (5.34). A hyperplane can now easily separate the green from magenta balls, showing that linear classification can be accomplished simply by enriching the measurement space of the data. Visual inspection alone suggests that nearly optimal separation can be achieved with the plane $z_3 \approx 14$ (shaded gray plane). In the original coordinate system this gives a circular classification line (black line on the plane x_1 versus x_2) with radius $r = \sqrt{z_3} = \sqrt{x_1^2 + x_2^2} \approx \sqrt{14}$. This example makes it obvious how a hyperplane in higher-dimensions can produce curved classification lines in the original data space.

However, the embedding (5.34) projects into a three dimensional space which can be easily separated by a hyperplane as illustrated in Fig. 5.23.

The ability of SVM to embed in higher-dimensional nonlinear spaces makes it one of the most successful machine learning algorithms developed. The underlying optimization algorithm (5.31) remains unchanged, except that the previous labeling function $\bar{y}_j = \text{sign}(\mathbf{w} \cdot \mathbf{x}_j + b)$ is now

$$\bar{y}_j = \text{sign}(\mathbf{w} \cdot \Phi(\mathbf{x}_j) + b). \quad (5.35)$$

The function $\Phi(\mathbf{x})$ specifies the enriched space of observables. As a general rule, more features are better for classification.

Kernel methods for SVM

Despite its promise, the SVM method of building nonlinear classifiers by enriching in higher-dimensions leads to a computationally intractable optimization. Specifically, the large number of additional features leads to the *curse of dimensionality*. Thus computing the vectors \mathbf{w} is prohibitively expensive and may not even be represented explicitly in memory. The *kernel trick* solves this problem. In this scenario, the \mathbf{w} vector is represented as follows

$$\mathbf{w} = \sum_{j=1}^m \alpha_j \Phi(\mathbf{x}_j) \quad (5.36)$$

where α_j are parameters that weight the different nonlinear observable functions $\Phi(\mathbf{x}_j)$. Thus the vector \mathbf{w} is expanded in the observable set of functions. We can then generalize (5.33) to the following

$$f(\mathbf{x}) = \sum_{j=1}^m \alpha_j \Phi(\mathbf{x}_j) \cdot \Phi(\mathbf{x}) + b. \quad (5.37)$$

The *kernel function* [479] is then defined as

$$K(\mathbf{x}_j, \mathbf{x}) = \Phi(\mathbf{x}_j) \cdot \Phi(\mathbf{x}). \quad (5.38)$$

With this new definition of \mathbf{w} , the optimization problem (5.31) becomes

$$\underset{\alpha, b}{\operatorname{argmin}} \sum_{j=1}^m H(\mathbf{y}_j, \bar{\mathbf{y}}_j) + \frac{1}{2} \left\| \sum_{j=1}^m \alpha_j \Phi(\mathbf{x}_j) \right\|^2 \text{ subject to } \min_j |\mathbf{x}_j \cdot \mathbf{w}| = 1 \quad (5.39)$$

where α is the vector of α_j coefficients that must be determined in the minimization process. There are different conventions for representing the minimization. However, in this formulation, the minimization is now over α instead of \mathbf{w} .

In this formulation, the kernel function $K(\mathbf{x}_j, \mathbf{x})$ essentially allows us to represent Taylor series expansions of a large (infinite) number of observables in a compact way [479]. The kernel function enables one to operate in a high-dimensional, implicit feature space without ever computing the coordinates of the data in that space, but rather by simply computing the inner products between all pairs of data in the feature space. For instance, two of the most commonly used kernel functions are

$$\text{Radial basis functions (RBF): } K(\mathbf{x}_j, \mathbf{x}) = \exp(-\gamma \|\mathbf{x}_j - \mathbf{x}\|^2) \quad (5.40a)$$

$$\text{Polynomial kernel: } K(\mathbf{x}_j, \mathbf{x}) = (\mathbf{x}_j \cdot \mathbf{x} + 1)^N \quad (5.40b)$$

where N is the degree of polynomials to be considered, which is exceptionally large to evaluate without using the kernel trick, and γ is the width of the Gaussian kernel measuring the distance between individual data points x_j and the classification line. These functions can be differentiated in order to optimize (5.39).

This represents the major theoretical underpinning of the SVM method. It allows us to construct higher-dimensional spaces using observables generated by kernel functions. Moreover, it results in a computationally tractable optimization. The following code shows the basic workings of the kernel method on the example of dog and cat classification data. In the first example, a standard linear SVM is used, while in the second, the RBF is executed as an option.

Code 5.14: SVM classification.

```

load catData_w.mat
load dogData_w.mat
CD=[dog_wave cat_wave];
[u,s,v]=svd(CD-mean(CD(:)));
features=1:20;
xtrain=[v(1:60,features); v(81:140,features)];
label=[ones(60,1); -1*ones(60,1)];
test=[v(61:80,features); v(141:160,features)];
truth=[ones(20,1); -1*ones(20,1)];

Mdl = fitcsvm(xtrain,label);
test_labels = predict(Mdl,test);

Mdl = fitcsvm(xtrain,label,'KernelFunction','RBF');
test_labels = predict(Mdl,test);
CMdl = crossval(Mdl);           % cross-validate the model
classLoss = kfoldLoss(CMdl)    % compute class loss

```

Note that in this code we have demonstrated some of the diagnostic features of the SVM method in MATLAB, including the cross-validation and class loss scores that are associated with training. This is a superficial treatment of the SVM. Overall, SVM is one of the most sophisticated machine learning tools in MATLAB and there are many options that can be executed in order to tune performance and extract accuracy/cross-validation metrics.

5.8 Classification trees and random forest

Decision trees are common in business. They establish an algorithmic flow chart for making decisions based on criteria that are deemed important and related to a desired outcome. Often the decision trees are constructed by experts

with knowledge of the workflow involved in the decision making process. *Decision tree learning* provides a principled method based on data for creating a predictive model for classification and/or regression. Along with SVM, classification and regression trees are core machine learning and data mining algorithms used in industry given their demonstrated success. The work of Leo Breiman and co-workers [79] established many of the theoretical foundations exploited today for data mining.

The decision tree is a hierarchical construct that looks for optimal ways to split the data in order to provide a robust classification and regression. It is the opposite of the unsupervised dendrogram hierarchical clustering previously demonstrated. In this case, our goal is not to move from bottom up in the clustering process, but from top down in order to create the best splits possible for classification. The fact that it is a supervised algorithm, which uses labeled data, allows us to split the data accordingly.

There are significant advantages in developing decision trees for classification and regression: (i) they often produce interpretable results that can be graphically displayed, making them easy to interpret even for non-experts, (ii) they can handle numerical or categorical data equally well, (iii) they can be statistically validated so that the reliability of the model can be assessed, (iv) they perform well with large data sets at scale, and (v) the algorithms mirror human decision making, again making them more interpretable and useful.

As one might expect, the success of decision tree learning has produced a large number of innovations and algorithms for how to best split the data. The coverage here will be limited, but we will highlight the basic architecture for data splitting and tree construction. Recall that we have the following:

$$\text{data } \{\mathbf{x}_j \in \mathbb{R}^n, j \in Z := \{1, 2, \dots, m\}\} \quad (5.41a)$$

$$\text{labels } \{\mathbf{y}_j \in \{\pm 1\}, j \in Z' \subset Z\}. \quad (5.41b)$$

The basic decision tree algorithm is fairly simple: (i) scan through each component (feature) x_k ($k = 1, 2, \dots, n$) of the vector \mathbf{x}_j to identify the value of x_j that gives the best labeling prediction for y_j . (ii) Compare the prediction accuracy for each split on the feature x_j . The feature giving the best segmentation of the data is selected as the split for the tree. (iii) With the two new branches of the tree created, this process is repeated on each branch. The algorithm terminates once the each individual data point is a unique cluster, known as a *leaf*, on a new branch of the tree. This is essentially the inverse of the dendrogram.

As a specific example, consider the Fisher iris data set from Fig. 5.1. For this data, each flower had four features (petal width and length, sepal width and length), and three labels (setosa, versicolor and virginica). There were fifty flowers of each variety for a total of 150 data points. Thus for this data the

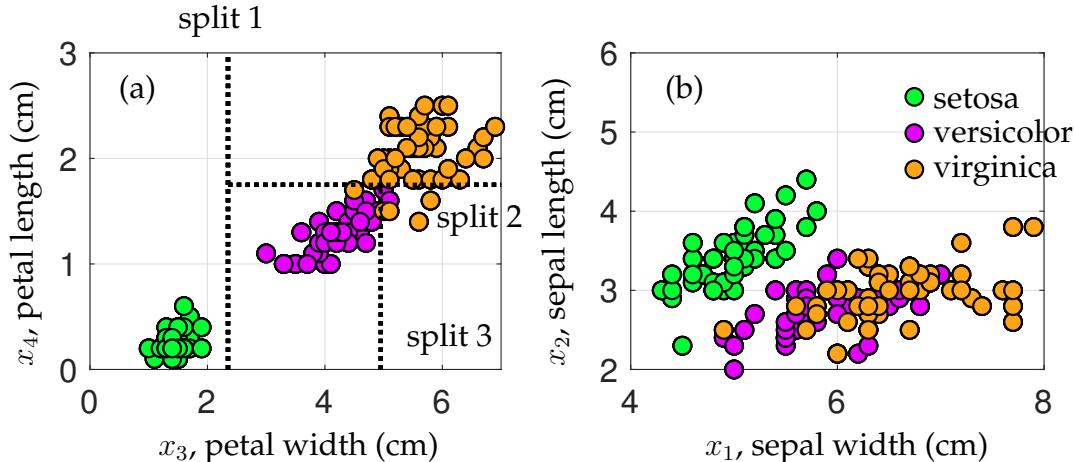


Figure 5.24: Illustration of the splitting procedure for decision tree learning performed on the Fisher iris data set. Each variable x_1 through x_4 is scanned over to determine the best split of data which retains the best correct classification of the labeled data in the split. The variable $x_3 = 2.35$ provides the first split in the data for building a classification tree. This is followed by a second split at $x_4 = 1.75$ and a third split at $x_3 = 4.95$. Only three splits are shown. The classification tree after three splits is shown in Fig. 5.25. Note that although the setosa data in the x_1 and x_2 direction seems to be well separated along a diagonal line, the decision tree can only split along horizontal and vertical lines.

vector \mathbf{x}_j has the four components

$$x_1 = \text{sepal width} \quad (5.42\text{a})$$

$$x_2 = \text{sepal length} \quad (5.42\text{b})$$

$$x_3 = \text{petal width} \quad (5.42\text{c})$$

$$x_4 = \text{petal length}. \quad (5.42\text{d})$$

The decision tree algorithm scans over these four features in order to decide how to best split the data. Figure 5.24 shows the splitting process in the space of the four variables x_1 through x_4 . Illustrated are two data planes containing x_1 versus x_2 (panel (b)) and x_3 versus x_4 (panel (a)). By visual inspection, one can see that the x_3 (petal length) variable maximally separates the data. In fact, the decision tree performs the first split of the data at $x_3 = 2.35$. No further splitting is required to predict setosa, as this first split is sufficient. The variable x_4 then provides the next most promising split at $x_4 = 1.75$. Finally, a third split is performed at $x_3 = 4.95$. Only three splits are shown. This process shows that the splitting procedure is has an intuitive appeal as the data splits optimally separating the data are clear visible. Moreover, the splitting does not occur on the x_1 and x_2 (width and length) variables as they do not provide a clear separation of the data. Figure 5.25 shows the tree used for Fig. 5.24.

The following code fits a tree to the Fisher iris data. Note that the `fitctree`

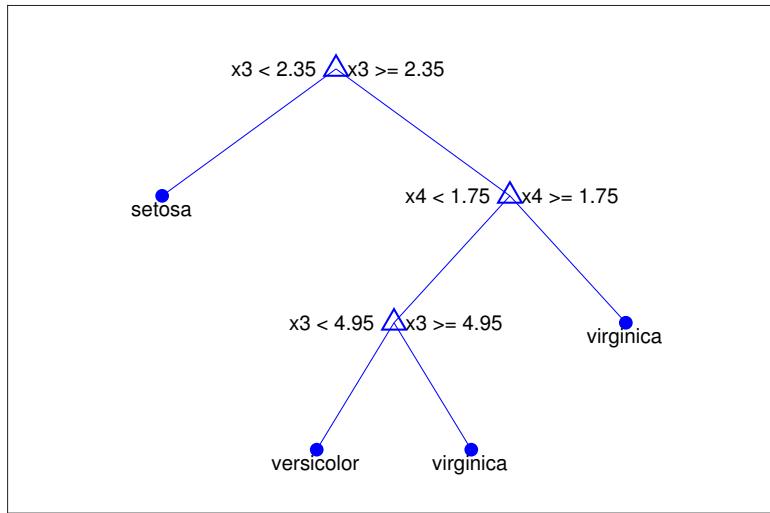


Figure 5.25: Tree structure generated by the MATLAB `fitctree` command. Note that only three splits are conducted, creating a classification tree that produces a class error of 4.67%

command allows for many options, including a cross-validation procedure (used in the code) and parameter tuning (not used in the code).

Code 5.15: Decision tree classification of Fisher iris data.

```

load fisheriris;
tree=fitctree(meas,species,'MaxNumSplits',3,'CrossVal','on')
view(tree.Trained{1}, 'Mode', 'graph');
classError = kfoldLoss(tree)

x1=meas(1:50,:); % setosa
x2=meas(51:100,:); % versicolor
x3=meas(101:150,:); % virginica
  
```

The results of the splitting procedure are demonstrated in Fig. 5.25. The `view` command generates an interactive window showing the tree structure. The tree can be pruned and other diagnostics are shown in this interactive graphic format. The class error achieved for the Fisher iris data is 4.67%.

As a second example, we construct a decision tree to classify dogs versus cats using our previously considered wavelet images. The following code loads and splits the data.

Code 5.16: Decision tree classification of dogs versus cats.

```

load catData_w.mat
load dogData_w.mat
  
```

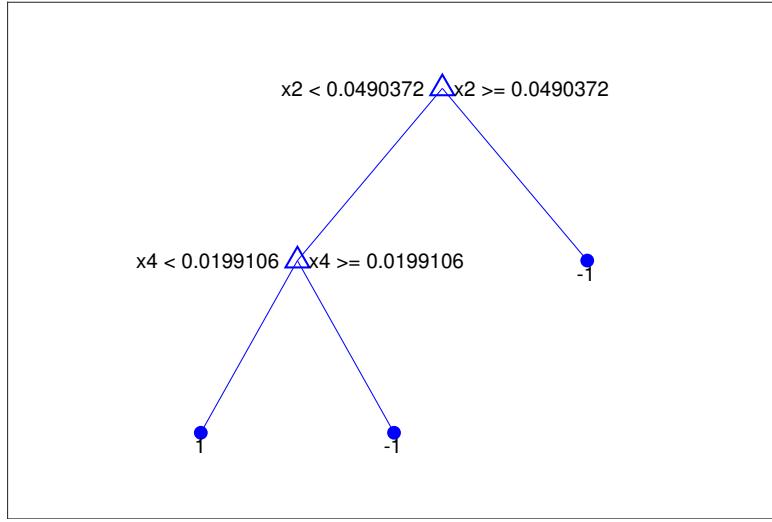


Figure 5.26: Tree structure generated by the MATLAB `fitctree` command for dog versus cat data. Note that only two splits are conducted, creating a classification tree that produces a class error of approximately 16%

```

CD=[dog_wave cat_wave];
[u,s,v]=svd(CD-mean(CD(:)));
features=1:20;
xtrain=[v(1:60,features); v(81:140,features)];
label=[ones(60,1); -1*ones(60,1)];
test=[v(61:80,features); v(141:160,features)];
truth=[ones(20,1); -1*ones(20,1)];

Mdl = fitctree(xtrain,label,'MaxNumSplits',2,'CrossVal','on');
classError = kfoldLoss(Mdl)
view(Mdl.Trained{1}, 'Mode', 'graph');
classError = kfoldLoss(Mdl)
  
```

Figure 5.26 shows the resulting classification tree. Note that the decision tree learning algorithm identifies the first two splits as occurring along the x_2 and x_4 variables respectively. These two variables have been considered previously since their histograms show them to be more distinguishable than the other PCA components (See Fig. 5.5). For this splitting, which has been cross-validated, the class error achieved is approximately 16%, which can be compared with the 30% error of LDA.

As a final example, we consider census data that is included in MATLAB.

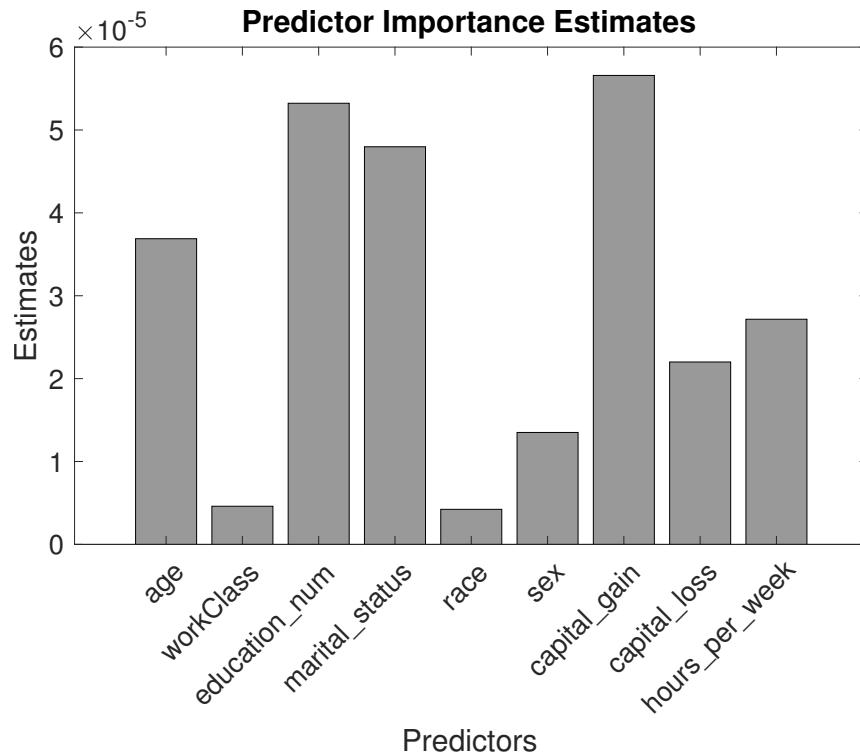


Figure 5.27: Importance of variables for prediction of salary data for the US census of 1994. The classification tree architecture allows for sophisticated treatment of data, including understanding how each variable contributes statistically to predicting a classification outcome.

The following code shows some important uses of the classification and regression tree architecture. In particular, the variables included can be used to make associations between relationships. In this case, the various data is used to predict the salary data. Thus, salary is the outcome of the classification. Moreover, the importance of each variable and its relation to salary can be computed, as shown in Fig. 5.27. The following code highlights some of the functionality of the tree architecture.

Code 5.17: Decision tree classification of census data.

```

load census1994
X = adultdata(:, {'age', 'workClass', 'education_num',
    'marital_status', 'race', 'sex', 'capital_gain', ...
    'capital_loss', 'hours_per_week', 'salary'});
Mdl = fitctree(X, 'salary', 'PredictorSelection', 'curvature',
    'Surrogate', 'on');
imp = predictorImportance(Mdl);

```

```

||| bar(imp,'FaceColor',[.6 .6 .6],'EdgeColor','k');
||| title('Predictor Importance Estimates');
||| ylabel('Estimates'); xlabel('Predictors'); h = gca;
||| h.XTickLabel = Mdl.PredictorNames;
||| h.XTickLabelRotation = 45;

```

As with the SVM algorithm, there exists a wide variety of tuning parameters for classification trees, and this is a superficial treatment. Overall, such trees are one of the most sophisticated machine learning tools in MATLAB and there are many options that can be executed to tune performance and extract accuracy/cross-validation metrics.

Random forest algorithms

Before closing this section, it is important to mention Breiman's *random forest* [77] innovations for decision learning trees. Random forests, or random decision forests, are an ensemble learning method for classification and regression. This is an important innovation since the decision trees created by splitting are generally not robust to different samples of the data. Thus one can generate two significantly different classification trees with two subsamples of the data. This presents significant challenges for cross-validation. In ensemble learning, a multitude of decision trees are constructed in the training process. The random decision forests correct for a decision trees' habit of overfitting to their training set, thus providing a more robust framework for classification.

There are many variants of the random forest architecture, including variants with *boosting* and *bagging*. These will not be considered here except to mention that the MATLAB `figctree` exploits many of these techniques through its options. One way to think about ensemble learning is that it allows for robust classification trees. It often does this by focusing its training efforts on hard-to-classify data instead of easy-to-classify data. Random forests, bagging and boosting are all extensive subjects in their own right, but have already been incorporated into leading software which build decision learning trees.

5.9 Top 10 algorithms in data mining 2008

This chapter has illustrated the tremendous diversity of supervised and unsupervised methods available for the analysis of data. Although the algorithms are now easily accessible through many commercial and open-source software packages, the difficulty is now evaluating which method(s) should be used on a given problem. In December 2006, various machine learning experts attending the IEEE International Conference on Data Mining (ICDM) identified the top 10 algorithms for data mining [562]. The identified algorithms were the following: C4.5, *k*-Means, SVM, Apriori, EM, PageRank, AdaBoost, *k*NN, Naive

Bayes, and CART. These top 10 algorithms were identified at the time as being among the most influential data mining algorithms in the research community. In the summary article, each algorithm was briefly described along with its impact and potential future directions of research. The 10 algorithms covered classification, clustering, statistical learning, association analysis, and link mining, which are all among the most important topics in data mining research and development. Interestingly, deep learning and neural networks, which are the topic of the next chapter, are not mentioned in the article. The landscape of data science would change significantly in 2012 with the ImageNET data set, and deep convolutional neural networks began to dominate almost any meaningful metric for classification and regression accuracy.

In this section, we highlight their identified top 10 algorithms and the basic mathematical structure of each. Many of them have already been covered in this chapter. This list is not exhaustive, nor does it rank them beyond their inclusion in the top 10 list. Our objective is simply to highlight what was considered by the community as the state-of-the-art data mining tools in 2008. We begin with those algorithms already considered previously in this chapter.

***k*-means**

This is one of the workhorse unsupervised algorithms. As already demonstrated, the goal of *k*-means is simply to cluster by proximity to a set of *k* points. By updating the locations of the *k* points according to the mean of the points closest to them, the algorithm iterates to the *k*-means. The structure of the MATLAB command is as follows

```
|| [labels,centers]=kmeans (X, k)
```

The **means** command takes in data **X** and the number of prescribed clusters *k*. It returns labels for each point **labels** along with their location **centers**.

EM (mixture models)

Mixture models are the second workhorse algorithm for unsupervised learning. The assumption underlying the mixture models is that the observed data is produced by a mixture of different probability distribution functions whose weightings are unknown. Moreover, the parameters must be estimated, thus requiring the Expectation-Maximization (EM) algorithm. The structure of the MATLAB command is as follows

```
|| Model=fitgmdist (X, k)
```

where the **fitgmdist** by default fits Gaussian mixtures to the data **X** in *k* clusters. The **Model** output is a structured variable containing information on the probability distributions (mean, variance, etc.) along with the goodness-of-fit.

Support vector machine (SVM)

One of the most powerful and flexible supervised learning algorithms used for most of the 90s and 2000s, the SVM is an exceptional off-the-shelf method for classification and regression. The main idea: project the data into higher dimensions and split the data with hyperplanes. Critical to making this work in practice was the kernel trick for efficiently evaluating inner products of functions in higher-dimensional space. The structure of the MATLAB command is as follows

```
|| Model = fitcsvm(xtrain,label);
|| test_labels = predict(Model,test);
```

where the `fitcsvm` command takes in labeled training data denoted by `train` and `label`, and it produces a structured output `Model`. The structured output can be used along with the `predict` command to take test data `test` and produce labels (`test_labels`). There exist many options and tuning parameters for `fitcsvm`, making it one of the best off-the-shelf methods.

CART (classification and regression tree)

This was the subject of the last section and was demonstrated to provide another powerful technique of supervised learning. The underlying idea was to split the data in a principled and informed way so as to produce an interpretable clustering of the data. The data splitting occurs along a single variable at a time to produce branches of the tree structure. The structure of the MATLAB command is as follows

```
|| tree = fitctree(xtrain,label);
```

where the `fitctree` command takes in labeled training data denoted by `train` and `label`, and it produces a structured output `tree`. There are many options and tuning parameters for `fitctree`, making it one of the best off-the-shelf methods.

k-nearest neighbors (kNN)

This is perhaps the simplest supervised algorithm to understand. It is highly interpretable and easy to execute. Given a new data point x_k which does not have a label, simply find the k nearest neighbors x_j with labels y_j . The label of the new point x_k is determined by a majority vote of the kNN. Given a model for the data, the MATLAB command to execute the kNN search is the following

```
|| label = knnsearch(Mdl,test)
```

where the `knnsearch` uses the `Mdl` to label the test data `test`.

Naive Bayes

The Naive Bayes algorithm provides an intuitive framework for supervised learning. It is simple to construct and does not require any complicated parameter estimation, similar to SVM and/or classification trees. It further gives highly interpretable results that are remarkably good in practice. The method is based upon Bayes's theorem and the computation of conditional probabilities. Thus one can estimate the label of a new data point based on the prior probability distributions of the labeled data. The MATLAB command structure for constructing a Naive Bayes model is the following

```
|| Model = fitNaiveBayes(xtrain,label)
```

where the `fitcNativeBayes` command takes in labeled training data denoted by `xtrain` and `label`, and it produces a structured output `Model`. The structured output can be used with the `predict` command to label test data `test`.

AdaBoost (ensemble learning and boosting)

AdaBoost is an example of an *ensemble learning* algorithm [188]. Broadly speaking, AdaBoost is a form of random forest [77] which takes into account an ensemble of decision tree models. The way all boosting algorithms work is to first consider an equal weighting for all training data x_j . Boosting re-weights the importance of the data according to how difficult they are to classify. Thus the algorithm focuses on harder to classify data. Thus a family of weak learners can be trained to yield a strong learner by boosting the importance of hard to classify data [470]. This concept and its usefulness are based upon a seminal theoretical contribution by Kearns and Valiant [283]. The structure of the MATLAB command is as follows

```
|| ada = fitcensemble(xtrain,label,'Method','AdaBoostM1')
```

where the `fitcensemble` command is a general ensemble learner that can do many more things than AdaBoost, including robust boosting and gradient boosting. Gradient boosting is one of the most powerful techniques [189].

C4.5 (ensemble learning of decision trees)

This algorithm is another variant of decision tree learning developed by J. R. Quinlan [443, 444]. At its core, the algorithm splits the data according to an information entropy score. In its latest versions, it supports boosting as well as many other well known functionalities to improve performance. Broadly, we can think of this as a strong performing version of CART. The `fitcensemble` algorithm highlighted with AdaBoost gives a generic ensemble learning architecture that can incorporate decision trees, allowing for a C4.5-like algorithm.

Apriori algorithm

The last two methods highlighted here tend to focus on different aspects of data mining. In the Apriori algorithm, the goal is to find frequent itemsets from data. Although this may sound trivial, it is not since data sets tend to be very large and can easily produce NP-hard computations because of the combinatorial nature of the algorithms. The Apriori algorithm provides an efficient algorithm for finding frequent itemsets using a candidate generation architecture [4]. This algorithm can then be used for fast learning of associate rules in the data.

PageRank

The founding of Google by Sergey Brin and Larry Page revolved around the PageRank algorithm [82]. PageRank produces a static ranking of variables, such as web pages, by computing an off-line value for each variable that does not depend on search queries. The PageRank is associated with graph theory as it originally interpreted a hyperlink from one page to another as a vote. From this, and various modifications of the original algorithm, one can then compute an importance score for each variable and provide an ordered rank list. The number of enhancements for this algorithm is quite large. Producing accurate orderings of variables (web pages) and their importance remains an active topic of research.

Suggested reading

Texts

- (1) **Machine learning: a probabilistic perspective**, by K. P. Murphy, 2012 [396].
- (2) **Pattern recognition and machine learning**, by C. M. Bishop, 2006 [64].
- (3) **Pattern classification**, by R. O. Duda, P. E. Hart, and D. G. Stork, 2000 [161].
- (4) **An introduction to statistical learning**, by G. James, D. Witten, T. Hastie and R. Tibshirani, 2013 [264].
- (5) **Learning with kernels: support vector machines, regularization, optimization, and beyond**, by B. Schölkopf and A. J. Smola, 2002 [479].
- (6) **Classification and regression trees**, by L. Breiman, J. Friedman, C. J. Stone and R. A. Olshen, 1984 [79].
- (7) **Random forests**, by L. Breiman, 2001 [77].

Papers and reviews

- (1) **Top 10 algorithms in data mining**, by X. Wu et al., *Knowledge and information systems*, 2008 [562].
- (2) **The strength of weak learnability**, by R. E. Schapire, *Machine Learning*, 1990 [470].
- (3) **Greedy function approximation: a gradient boosting machine**, by J. H. Friedman, *Annals of Statistics*, 2001 [189].

Chapter 6

Neural Networks and Deep Learning

Neural networks (NNs) were inspired by the Nobel prize winning work of Hubel and Wiesel on the primary visual cortex of cats [259]. Their seminal experiments showed that neuronal networks were organized in hierarchical layers of cells for processing visual stimulus. The first mathematical model of the NN, termed the Neocognitron in 1980 [193], had many of the characteristic features of today's deep convolutional NNs (or DCNNs), including a multi-layer structure, convolution, max pooling and nonlinear dynamical nodes. The recent success of DCNNs in computer vision has been enabled by two critical components: (i) the continued growth of computational power, and (ii) exceptionally large labeled data sets which take advantage of the power of a *deep* multi-layer architecture. Indeed, although the theoretical inception of NNs has an almost four-decade history, the analysis of the ImageNet data set in 2012 [310] provided a watershed moment for NNs and deep learning [324]. Prior to this data set, there were a number of data sets available with approximately tens of thousands of labeled images. ImageNet provided over 15 million labeled, high-resolution images with over 22,000 categories. DCNNs, which are only one potential category of NNs, have since transformed the field of computer vision by dominating the performance metrics in almost every meaningful computer vision task intended for classification and identification.

Although ImageNet has been critically enabling for the field, NNs were textbook material in the early 1990s with a focus typically on a small number of layers. Critical machine learning tasks such as principal component analysis (PCA) were shown to be intimately connected with networks which included back propagation. Importantly, there were a number of critical innovations which established multilayer feedforward networks as a class of universal approximators [255]. The past five years have seen tremendous advances in NN architectures, many designed and tailored for specific application areas. Innovations have come from algorithmic modifications that have led to significant performance gains in a variety of fields. These innovations include pretraining, dropout, inception modules, data augmentation with virtual examples, batch

normalization, and/or residual learning (See Ref. [216] for a detailed exposition of NNs). This is only a partial list of potential algorithmic innovations, thus highlighting the continuing and rapid pace of progress in the field. Remarkably, NNs were not even listed as one of the top 10 algorithms of data mining in 2008 [562]. But a decade later, its undeniable and growing list of successes on challenge data sets make it perhaps the most important data mining tool for our emerging generation of scientists and engineers.

As already shown in the last two chapters, all of machine learning revolves fundamentally around optimization. NNs specifically optimize over a compositional function

$$\operatorname{argmin}_{\mathbf{A}_j} (f_M(\mathbf{A}_M, \dots, f_2(\mathbf{A}_2, f_1(\mathbf{A}_1, \mathbf{x})) \dots) + \lambda g(\mathbf{A}_j)) \quad (6.1)$$

which is often solved using stochastic gradient descent and back propagation algorithms. Each matrix \mathbf{A}_k denotes the weights connecting the neural network from the k th to $(k+1)$ th layer. It is a massively underdetermined system which is regularized by $g(\mathbf{A}_j)$. Composition and regularization are critical for generating expressive representations of the data and preventing overfitting, respectively. This general optimization framework is at the center of deep learning algorithms, and its solution will be considered in this chapter. Importantly, NNs have significant potential for overfitting of data so that cross-validation must be carefully considered. Recall that *if you don't cross-validate, you is dumb*.

6.1 Neural networks: 1-Layer networks

The generic architecture of a multi-layer NN is shown in Fig. 6.1. For classification tasks, the goal of the NN is to map a set of input data to a classification. Specifically, we train the NN to accurately map the data \mathbf{x}_j to their correct label \mathbf{y}_j . As shown in Fig. 6.1, the input space has the dimension of the raw data $\mathbf{x}_j \in \mathbb{R}^n$. The output layer has the dimension of the designed classification space. Constructing the output layer will be discussed further in the following.

Immediately, one can see that there are a great number of design questions regarding NNs. How many layers should be used? What should be the dimension of the layers? How should the output layer be designed? Should one use all-to-all or sparsified connections between layers? How should the mapping between layers be performed: a *linear mapping* or a *nonlinear mapping*? Much like the tuning options on SVM and classification trees, NNs have a significant number of design options that can be tuned to improve performance.

Initially, we consider the mapping between layers of Fig. 6.1. We denote the various layers between input and output as $\mathbf{x}^{(k)}$ where k is the layer number.

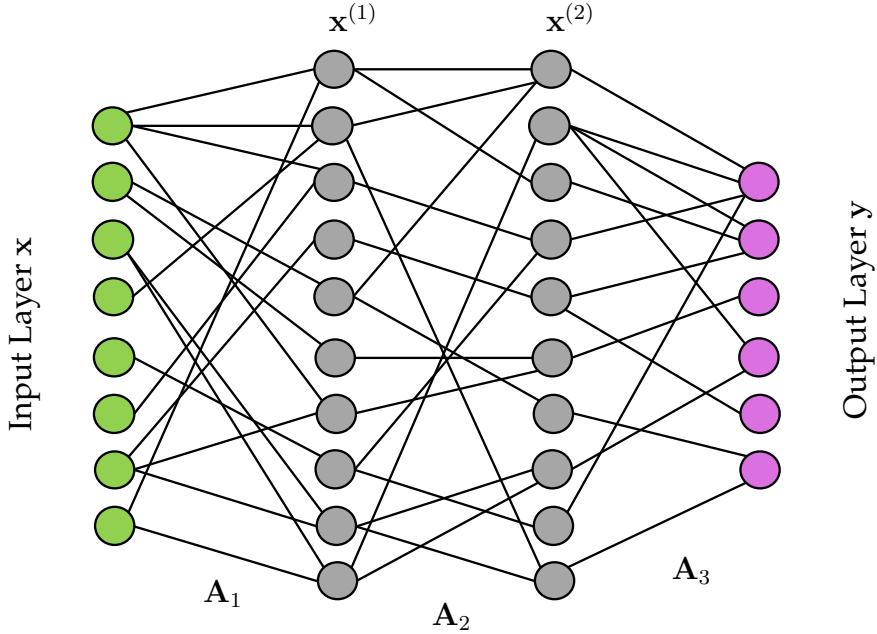


Figure 6.1: Illustration of a neural net architecture mapping an input layer x to an output layer y . The middle (hidden) layers are denoted $x^{(j)}$ where j determines their sequential ordering. The matrices A_j contain the coefficients that map each variable from one layer to the next. Although the dimensionality of the input layer $x \in \mathbb{R}^n$ is known, there is great flexibility in choosing the dimension of the inner layers as well as how to structure the output layer. The number of layers and how to map between layers is also selected by the user. This flexible architecture gives great freedom in building a good classifier.

For a linear mapping between layers, the following relations hold

$$x^{(1)} = A_1 x \quad (6.2a)$$

$$x^{(2)} = A_2 x^{(1)} \quad (6.2b)$$

$$y = A_3 x^{(2)}. \quad (6.2c)$$

This forms a compositional structure so that the mapping between input and output can be represented as

$$y = A_3 A_2 A_1 x. \quad (6.3)$$

This basic architecture can scale to M layers so that a general representation between input data and the output layer for a linear NN is given by

$$y = A_M A_{M-1} \cdots A_2 A_1 x. \quad (6.4)$$

This is generally a highly underdetermined system that requires some constraints on the solution in order to select a unique solution. One constraint

is immediately obvious: The mapping must generate M distinct matrices that give the best mapping. It should be noted that linear mappings, even with a compositional structure, can only produce a limited range of functional responses due to the limitations of the linearity.

Nonlinear mappings are also possible, and generally used, in constructing the NN. Indeed, nonlinear activation functions allow for a richer set of functional responses than their linear counterparts. In this case, the connections between layers are given by

$$\mathbf{x}^{(1)} = f_1(\mathbf{A}_1, \mathbf{x}) \quad (6.5a)$$

$$\mathbf{x}^{(2)} = f_2(\mathbf{A}_2, \mathbf{x}^{(1)}) \quad (6.5b)$$

$$\mathbf{y} = f_3(\mathbf{A}_3, \mathbf{x}^{(2)}). \quad (6.5c)$$

Note that we have used different nonlinear functions $f_j(\cdot)$ between layers. Often a single function is used; however, there is no constraint that this is necessary. In terms of mapping the data between input and output over M layers, the following is derived

$$\mathbf{y} = f_M(\mathbf{A}_M, \dots, f_2(\mathbf{A}_2, f_1(\mathbf{A}_1, \mathbf{x})) \dots) \quad (6.6)$$

which can be compared with (6.1) for the general optimization which constructs the NN. As a highly underdetermined system, constraints should be imposed in order to extract a desired solution type, as in (6.1). For big data applications such as ImageNET and computer vision tasks, the optimization associated with this compositional framework is expensive given the number of variables that must be determined. However, for moderate sized networks, it can be performed on workstation and laptop computers. Modern stochastic gradient descent and back propagation algorithms enable this optimization, and both are covered in later sections.

A one-layer network

To gain insight into how an NN might be constructed, we will consider a single layer network that is optimized to build a classifier between dogs and cats. The dog and cat example was considered extensively in the previous chapter. Recall that we were given images of dogs and cats, or a wavelet version of dogs and cats. Figure 6.2 shows our construction. To make this as simple as possible, we consider the simple NN output

$$\mathbf{y} = \{\text{dog, cat}\} = \{+1, -1\} \quad (6.7)$$

which labels each data vector with an output $\mathbf{y} \in \{\pm 1\}$. In this case the output layer is a single node. As in previous supervised learning algorithms the goal is to determine a mapping so that each data vector \mathbf{x}_j is labeled correctly by y_j .

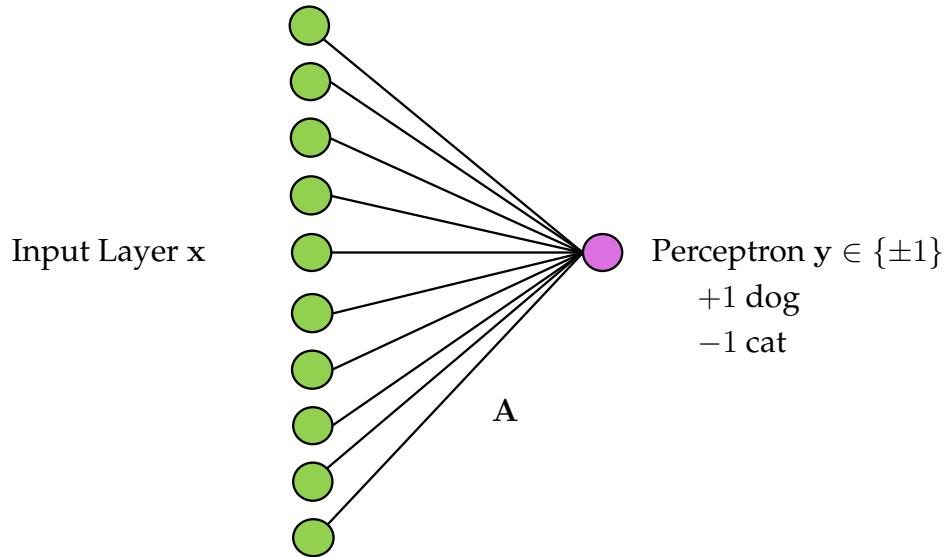


Figure 6.2: Single layer network for binary classification between dogs and cats. The output layer for this case is a perceptron with $y \in \{\pm 1\}$. A linear mapping between the input image space and output output layer can be constructed for training data by solving $\mathbf{A} = \mathbf{Y}\mathbf{X}^\dagger$. This gives a least square regression for the matrix \mathbf{A} mapping the images to label space.

The easiest mapping is a linear mapping between the input images $\mathbf{x}_j \in \mathbb{R}^n$ and the output layer. This gives a linear system $\mathbf{AX} = \mathbf{Y}$ of the form

$$\mathbf{AX} = \mathbf{Y} \rightarrow [a_1 \ a_2 \ \cdots \ a_n] \begin{bmatrix} | & | & & | \\ \mathbf{x}_1 & \mathbf{x}_2 & \cdots & \mathbf{x}_p \\ | & | & & | \end{bmatrix} = [+1 \ +1 \ \cdots \ -1 \ -1] \quad (6.8)$$

where each column of the matrix \mathbf{X} is a dog or cat image and the columns of \mathbf{Y} are its corresponding labels. Since the output layer is a single node, both \mathbf{A} and \mathbf{Y} reduce to vectors. In this case, our goal is to determine the matrix (vector) \mathbf{A} with components a_j . The simplest solution is to take the pseudo-inverse of the data matrix \mathbf{X}

$$\mathbf{A} = \mathbf{Y}\mathbf{X}^\dagger. \quad (6.9)$$

Thus a single output layer allows us to build a NN using least-square fitting. Of course, we could also solve this linear system in a variety of other ways, including with sparsity-promoting methods. The following code solves this problem through both least-square fitting (`pinv`) and the LASSO.

Code 6.1: 1-layer, linear neural network.

```

load catData_w.mat; load dogData_w.mat; CD=[dog_wave
    cat_wave];
train=[dog_wave(:,1:60) cat_wave(:,1:60)];
test=[dog_wave(:,61:80) cat_wave(:,61:80)];
```

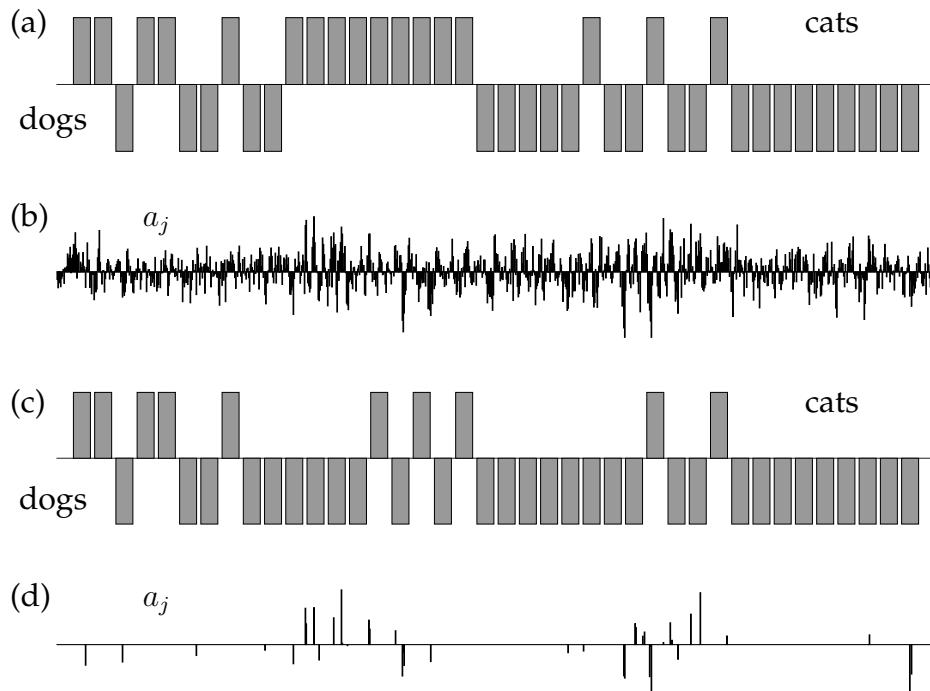


Figure 6.3: Classification of withheld data tested on a trained, single-layer network with linear mapping between inputs (pixel space) and a single output. (a) and (c) are the bar graph of the output layer score $y \in \{\pm 1\}$ achieved for the withheld data using a pseudo-inverse for training and the LASSO for training respectively. The results show in both cases that dogs are more often misclassified than cats are misclassified. (b) and (d) show the coefficients of the matrix A for the pseudo-inverse and LASSO respectively. Note that the LASSO has only a small number of nonzero elements, thus suggesting the NN is highly sparse.

```

label=[ones(60,1); -1*ones(60,1)].';

A=label*pinv(train); test_labels=sign(A*test);
subplot(4,1,1), bar(test_labels)
subplot(4,1,2), bar(A)
figure(2), subplot(2,2,1)
A2=flipud(reshape(A,32,32)); pcolor(A2), colormap(gray)

figure(1), subplot(4,1,3)
A=lasso(train.',label.', 'Lambda', 0.1).';
test_labels=sign(A*test);
bar(test_labels)
subplot(4,1,4)
bar(A)
figure(2), subplot(2,2,2)
A2=flipud(reshape(A,32,32)); pcolor(A2), colormap(gray)

```

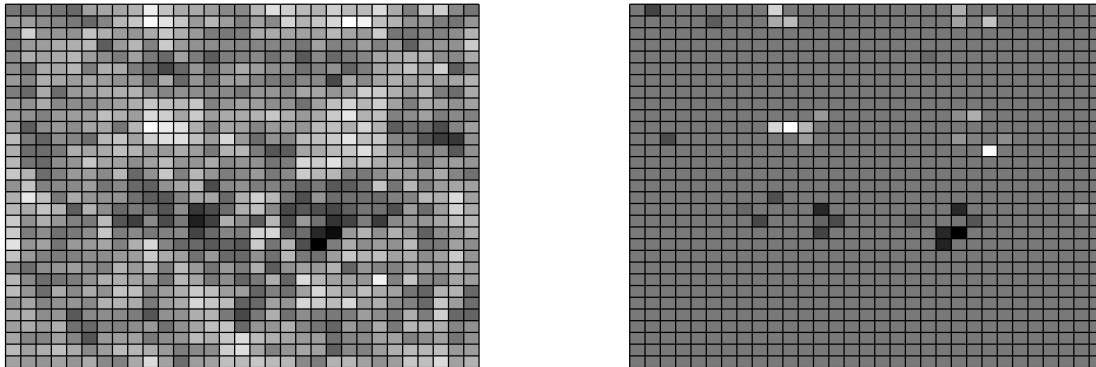


Figure 6.4: Weightings of the matrix A reshaped into 32×32 arrays. The left matrix shows the matrix A computed by least-square regression (the pseudo-inverse) while the right matrix shows the matrix A computed by LASSO. Both matrices provide similar classification scores on withheld data. They further provide interpretability in the sense that the results from the pseudo-inverse show many of the features of dogs and cats while the LASSO shows that measuring near the eyes and ears alone can give the features required for distinguishing between dogs and cats.

Figures 6.3 and 6.4 show the results of this linear single-layer NN with single node output layer. Specifically, the four rows of Fig. 6.3 show the output layer on the withheld test data for both the pseudo-inverse and LASSO methods along with a bar graph of the 32×32 (1024 pixels) weightings of the matrix A . Note that all matrix elements are nonzero in the pseudo-inverse solution, while the LASSO highlights a small number of pixels that can classify the pictures as well as using all pixels. Figure 6.4 shows the matrix A for the two solution strategies reshaped into 32×32 images. Note that for the pseudo-inverse, the weightings of the matrix elements A show many features of the cat and dog face. For the LASSO method, only a few pixels are required that are clustered near the eyes and ears. Thus for this single layer network, interpretable results are achieved by looking at the weights generated in the matrix A .

6.2 Multi-layer networks and activation functions

The previous section constructed what is perhaps the simplest NN possible. It was linear, had a single layer, and a single output layer neuron. The potential generalizations are endless, but we will focus on two simple extensions of the NN in this section. The first extension concerns the assumption of linearity in which we assumed that there is a linear transform from the image space to the output layer: $Ax = y$ in (6.8). We highlight here common nonlinear

transformations from input-to-output space represented by

$$\mathbf{y} = f(\mathbf{A}, \mathbf{x}) \quad (6.10)$$

where $f(\cdot)$ is a specified *activation function* (transfer function) for our mapping.

The linear mapping used previously, although simple, does not offer the flexibility and performance that other mappings offer. Some standard activation functions are given by

$$f(x) = x \quad - \text{ linear} \quad (6.11a)$$

$$f(x) = \begin{cases} 0 & x \leq 0 \\ 1 & x > 0 \end{cases} \quad - \text{ binary step} \quad (6.11b)$$

$$f(x) = \frac{1}{1 + \exp(-x)} \quad - \text{ logistic (soft step)} \quad (6.11c)$$

$$f(x) = \tanh(x) \quad - \text{ TanH} \quad (6.11d)$$

$$f(x) = \begin{cases} 0 & x \leq 0 \\ x & x > 0 \end{cases} \quad - \text{ rectified linear unit (ReLU).} \quad (6.11e)$$

There are other possibilities, but these are perhaps the most commonly considered in practice and they will serve for our purposes. Importantly, the chosen function $f(x)$ will be differentiated in order to be used in gradient descent algorithms for optimization. Each of the functions above is either differentiable or piecewise differentiable. Perhaps the most commonly used activation function is currently the ReLU, which we denote $f(x) = \text{ReLU}(x)$.

With a nonlinear activation function $f(x)$, or if there are more than one layer, then standard linear optimization routines such as the pseudo-inverse and LASSO can no longer be used. Although this may not seem immediately significant, recall that we are optimizing in a high-dimensional space where each entry of the matrix \mathbf{A} needs to be found through optimization. Even moderate to small problems can be computationally expensive to solve without using specialty optimization methods. Fortunately, the two dominant optimization components for training NNs, stochastic gradient descent and backpropagation, are included with the neural network function calls in MATLAB. As these methods are critically enabling, both of them are considered in detail in the next two sections of this chapter.

Multiple layers can also be considered as shown in (6.4) and (6.5). In this case, the optimization must simultaneously identify multiple connectivity matrices $\mathbf{A}_1, \mathbf{A}_2, \dots, \mathbf{A}_M$, in contrast to the linear case where only a single matrix is determined $\bar{\mathbf{A}} = \mathbf{A}_M \cdots \mathbf{A}_2 \mathbf{A}_M$. The multiple layer structure significantly increases the size of the optimization problem as each matrix element of the M matrices must be determined. Even for a one layer structure, an optimization routine such as `fminsearch` will be severely challenged when considering a

nonlinear transfer function and one needs to move to a gradient descent-based algorithm.

MATLAB's neural network toolbox, much like TensorFlow in python, has a wide range of features which makes it exceptionally powerful and convenient for building NNs. In the following code, we will train a NN to classify between dogs and cats as in the previous example. However, in this case, we allow the single layer to have a nonlinear transfer function that maps the input to the output layer. The output layer for this example will be modified to the following

$$\mathbf{y} = \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \{\text{dog}\} \text{ and } \mathbf{y} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \{\text{cat}\}. \quad (6.12)$$

Half of the data is extracted for training, while the other half is used for testing the results. The following code builds a network using the `train` command to classify between our images.

Code 6.2: Neural network with nonlinear transfer functions.

```

load catData_w.mat; load dogData_w.mat;
CD=[dog_wave cat_wave];

x=[dog_wave (:,1:40) cat_wave (:,1:40)];
x2=[dog_wave (:,41:80) cat_wave (:,41:80)];
label=[ones(40,1) zeros(40,1);
       zeros(40,1) ones(40,1)].';

net = patternnet(2,'trainscg');
net.layers{1}.transferFcn = 'tansig';

net = train(net,x,label);
view(net)
y = net(x);
y2= net(x2);
perf = perform(net,label,y);
classes2 = vec2ind(y);
classes3 = vec2ind(y2);

```

In the code above, the `patternnet` command builds a classification network with two outputs (6.12). It also optimizes with the option `trainscg` which is a *scaled conjugate gradient backpropagation*. The `net.layers` also allows us to specify the transfer function, in this case hyperbolic tangent functions (6.11d). The `view(net)` command produces a diagnostic tool shown in Fig. 6.5 that summarizes the optimization and NN.

The results of the classification for a cross-validated training set as well as a withhold set are shown in Fig. 6.6. Specifically, the desired outputs are given by the vectors (6.12). For both the training and withhold sets, the two components

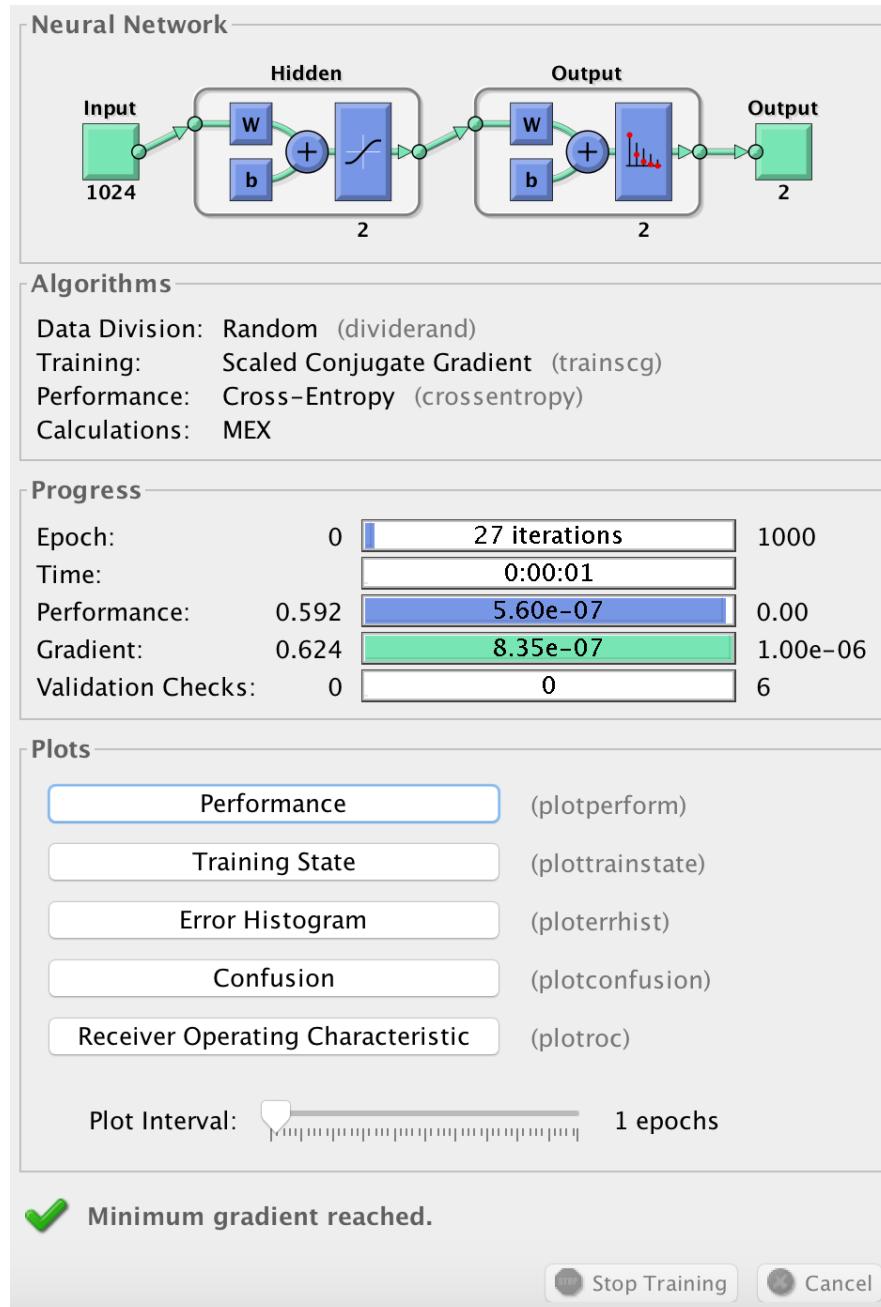


Figure 6.5: MATLAB neural network visualization tool. The number of iterations along with the performance can all be accessed from the interactive graphical tool. The performance, error histogram and confusion buttons produce Figs. 6.7-6.9 respectively.

of the vector are shown for the 80 training images (40 cats and 40 dogs) and the 80 withheld images (40 cats and 40 dogs). The training set produces a perfect classifier using a one layer network with a hyperbolic tangent transfer function

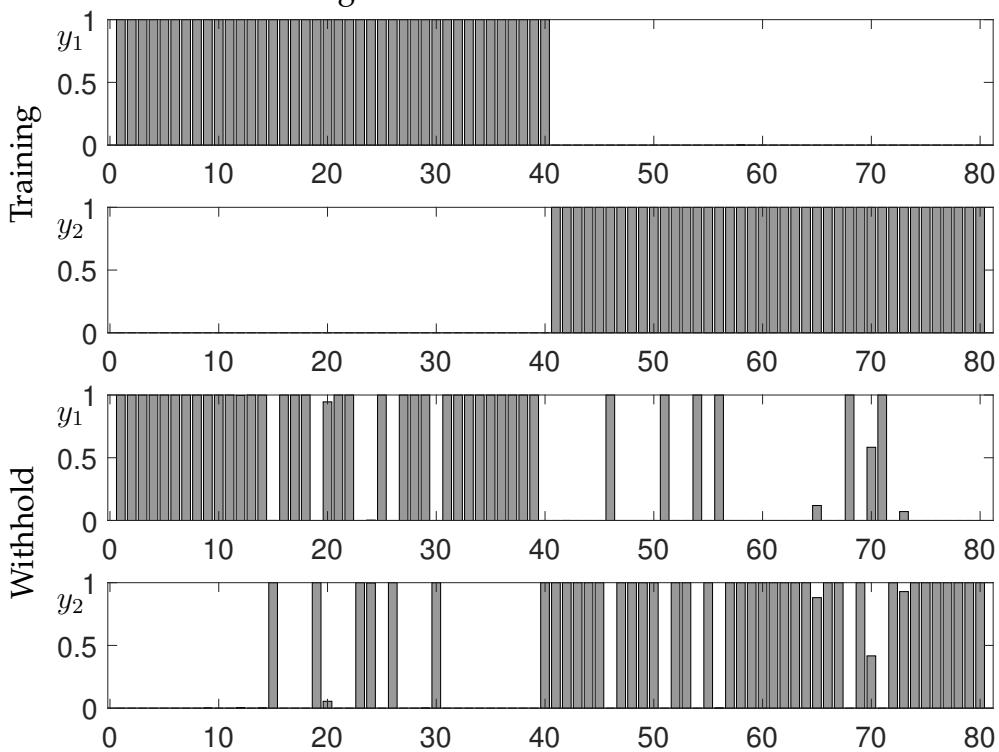


Figure 6.6: Comparison of the output vectors $\mathbf{y} = [y_1 \ y_2]^T$ which are ideally (6.12) for the dogs and cats considered here. The NN training stage produces a cross-validated classifier that achieves 100% accuracy in classifying the training data (top two panels for 40 dogs and 40 cats). When applied to a withheld set, 85% accuracy is achieved (bottom two panels for 40 dogs and 40 cats).

(6.11d). On the withheld data, it incorrectly identifies 6 of 40 dogs and cats, yielding an accuracy of $\approx 85\%$ on new data.

The diagnostic tool shown in Fig. 6.5 allows access to a number of features critical for evaluating the NN. Figure 6.7 is a summary of the performance achieved by the NN training tool. In this figure, the training algorithm automatically breaks the data into a training, validation and test set. The back-propagation enabled, stochastic gradient descent optimization algorithm then iterates through a number of training epochs until the cross-validated error achieves a minimum. In this case, twenty-two epochs is sufficient to achieve a minimum. The error on the test set is significantly higher than what is achieved for cross-validation. For this case, only a limited amount of data is used for training (40 dogs and 40 cats), thus making it difficult to achieve great performance. Regardless, as already shown, once the algorithm has been trained it can be used to evaluate new data as shown in Fig. 6.6.

There are two other features easily available with the NN diagnostic tool of Fig. 6.5. Figure 6.8 shows an error histogram associated with the trained net-

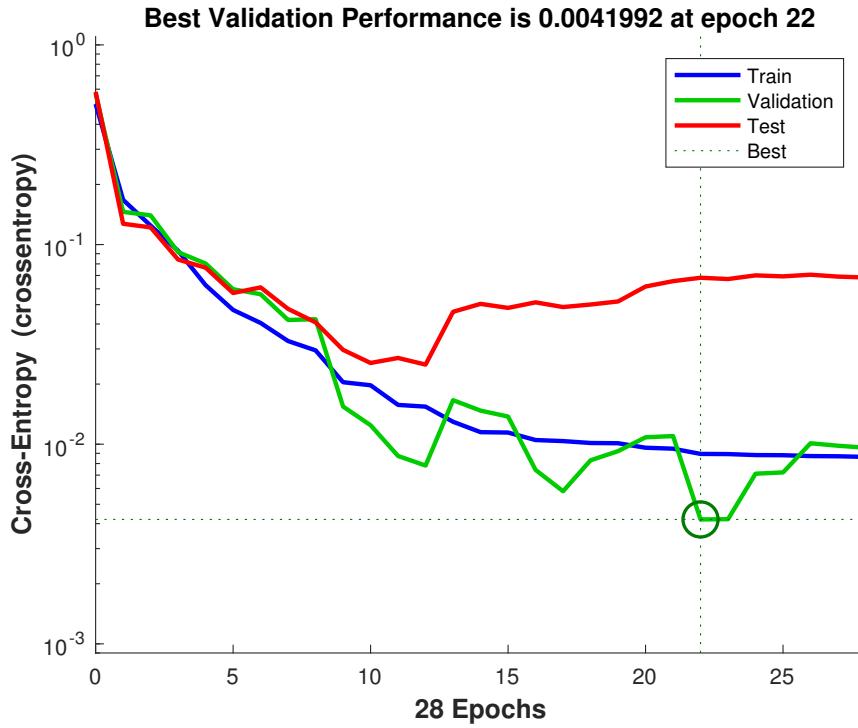


Figure 6.7: Summary of training of the NN over a number of epochs. The NN architecture automatically separates the data into training, validation and test sets. The training continues (with a maximum of 1000 epochs) until the validation error curve hits a minimum. The training then stops and the trained algorithm is then used on the test set to evaluate performance. The NN trained here has only a limited amount of data (40 dogs and 40 cats), thus limiting the performance. This figure is accessed with the performance button on the NN interactive tool of Fig. 6.6.

work. As with Fig. 6.7, the data is divided into training, validation, and test sets. This provides an overall assessment of the classification quality that can be achieved by the NN training algorithm. Another view of the performance can be seen in the confusion matrices for the training, validation, and test data. This is shown in Fig. 6.9. Overall, between Figs. 6.7 to 6.9, high-quality diagnostic tools are available to evaluate how well the NN is able to achieve its classification task. The performance limits are easily seen in these figures.

6.3 The backpropagation algorithm

As was shown for the NNs of the last two sections, training data is required to determine the weights of the network. Specifically, the network weights are determined so as to best classify dog versus cat images. In the 1-layer network,

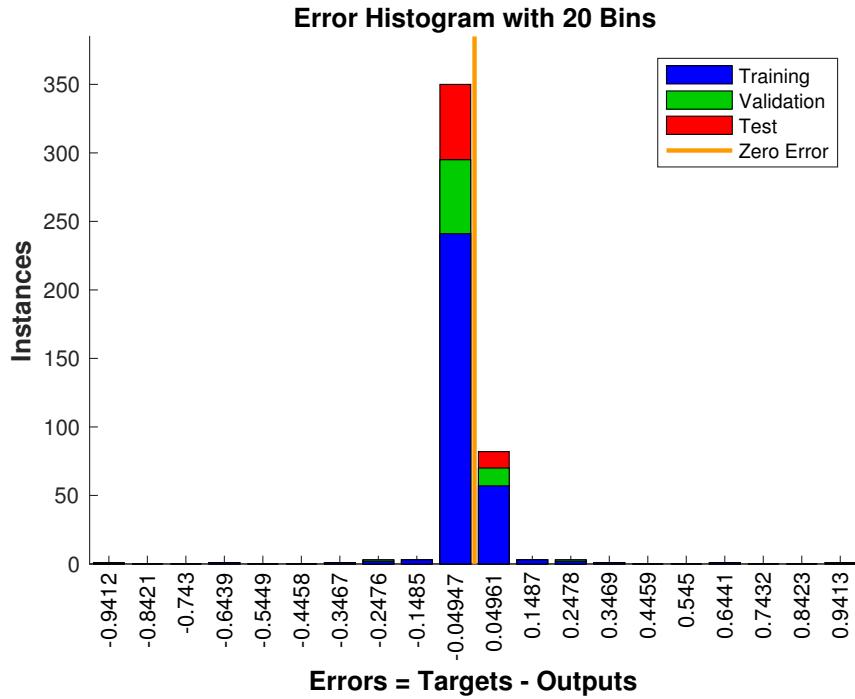


Figure 6.8: Summary of the error performance of the NN architecture for training, validation and test sets. This figure is accessed with the `errorhistogram` button on the NN interactive tool of Fig. 6.6.

this was done using both least-square regression and LASSO. This shows that at its core, an optimization routine and objective function is required to determine the weights. The objective function should minimize a measure of the misclassified images. The optimization, however, can be modified by imposing a regularizer or constraints, such as the ℓ_1 penalization in LASSO.

In practice, the objective function chosen for optimization is not the true objective function desired, but rather a proxy for it. Proxies are chosen largely due to the ability to differentiate the objective function in a computationally tractable manner. There are also many different objective functions for different tasks. Instead, one often considers a suitably chosen loss function so as to approximate the true objective. Ultimately, computational tractability is critical for training NNs.

The backpropagation algorithm (backprop) exploits the compositional nature of NNs in order to frame an optimization problem for determining the weights of the network. Specifically, it produces a formulation amenable to standard gradient descent optimization (See Sec. 4.2). Backprop relies on a simple mathematical principle: the chain rule for differentiation. Moreover, it can be proven that the computational time required to evaluate the gradient is within a factor of five of the time required for computing the actual function itself [44]. This is known as the Baur-Strassen theorem. Figure 6.10 gives

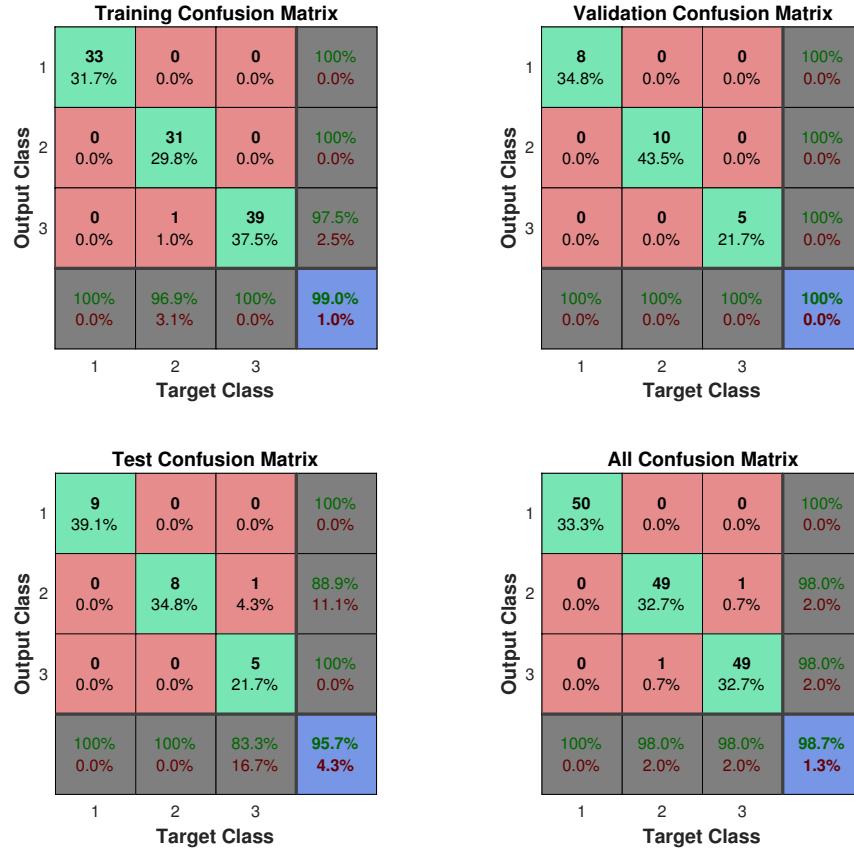


Figure 6.9: Summary of the error performance through confusion matrices of the NN architecture for training, validation and test sets. This figure is accessed with the confusion button on the NN interactive tool of Fig. 6.6.

the simplest example of backprop and how the gradient descent is to be performed. The input-to-output relationship for this single node, one hidden layer network, is given by

$$y = g(z, b) = g(f(x, a), b). \quad (6.13)$$

Thus given a function $f(\cdot)$ and $g(\cdot)$ with weighting constants a and b , the output error produced by the network can be computed against the ground truth as

$$E = \frac{1}{2}(y_0 - y)^2 \quad (6.14)$$

where y_0 is the correct output and y is the NN approximation to the output. The goal is to find a and b to minimize the error. The minimization requires

$$\frac{\partial E}{\partial a} = -(y_0 - y) \frac{dy}{dz} \frac{dz}{da} = 0. \quad (6.15)$$

A critical observation is that the compositional nature of the network along with the chain rule forces the optimization to backpropagate error through the

network. In particular, the terms dy/dz dz/da show how this backprop occurs. Given functions $f(\cdot)$ and $g(\cdot)$, the chain rule can be explicitly computed.

Backprop results in an iterative, gradient descent update rule

$$a_{k+1} = a_k + \delta \frac{\partial E}{\partial a_k} \quad (6.16a)$$

$$b_{k+1} = b_k + \delta \frac{\partial E}{\partial b_k} \quad (6.16b)$$

where δ is the so-called learning rate and $\partial E/\partial a$ along with $\partial E/\partial b$ can be explicitly computed using (6.15). The iteration algorithm is executed to convergence. As with all iterative optimization, a good initial guess is critical to achieve a good solution in a reasonable amount of computational time.

Backprop proceeds as follows: (i) A NN is specified along with a labeled training set. (ii) The initial weights of the network are set to random values. Importantly, one must not initialize the weights to zero, similar to what may be done in other machine learning algorithms. If weights are initialized to zero, after each update, the outgoing weights of each neuron will be identical, because the gradients will be identical. Moreover, NNs often get stuck at local optima where the gradient is zero but that are not global minima, so random weight initialization allows one to have a chance of circumventing this by starting at many different random values. (iii) The training data is run through the network to produce an output y , whose ideal ground-truth output is y_0 . The derivatives with respect to each network weight is then computed using back-prop formulas (6.15). (iv) For a given learning rate δ , the network weights are updated as in (6.16). (v) We return to step (iii) and continue iterating until a maximum number of iterations is reached or convergence is achieved.

As a simple example, consider the linear activation function

$$f(\xi, \alpha) = g(\xi, \alpha) = \alpha\xi. \quad (6.17)$$

In this case we have in Fig. 6.10

$$z = ax \quad (6.18a)$$

$$y = bz. \quad (6.18b)$$

We can now explicitly compute the gradients such as (6.15). This gives

$$\frac{\partial E}{\partial a} = -(y_0 - y) \frac{dy}{dz} \frac{dz}{da} = -(y_0 - y) \cdot b \cdot x \quad (6.19a)$$

$$\frac{\partial E}{\partial b} = -(y_0 - y) \frac{dy}{db} = -(y_0 - y)z = -(y_0 - y) \cdot a \cdot x. \quad (6.19b)$$

Thus with the current values of a and b , along with the input-output pair x and y and target truth y_0 , each derivative can be evaluated. This provides the required information to perform the update (6.16).

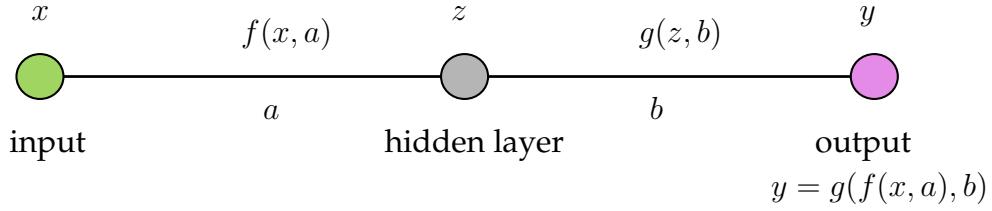


Figure 6.10: Illustration of the backpropagation algorithm on a one-node, one hidden layer network. The compositional nature of the network gives the input-output relationship $y = g(z, b) = g(f(x, a), b)$. By minimizing the error between the output y and its desired output y_0 , the composition along with the chain rule produces an explicit formula (6.15) for updating the values of the weights. Note that the chain rule backpropagates the error all the way through the network. Thus by minimizing the output, the chain rule acts on the compositional function to produce a product of derivative terms that advance backward through the network.

The backprop for a deeper net follows in a similar fashion. Consider a network with M hidden layers labeled z_1 to z_m with the first connection weight a between x and z_1 . The generalization of Fig. 6.10 and (6.15) is given by

$$\frac{\partial E}{\partial a} = -(y_0 - y) \frac{dy}{dz_m} \frac{dz_m}{dz_{m-1}} \dots \frac{dz_2}{dz_1} \frac{dz_1}{da}. \quad (6.20)$$

The cascade of derivates induced by the composition and chain rule highlights the backpropagation of errors that occurs when minimizing the classification error.

A full generalization of backprop involves multiple layers as well multiple nodes per layer. The general situation is illustrated in Fig. 6.1. The objective is to determine the matrix elements of each matrix \mathbf{A}_j . Thus a significant number of network parameters need to be updated in gradient descent. Indeed, training a network can often be computationally infeasible even though the update rules for individual weights is not difficult. NNs can thus suffer from the curse of dimensionality as each matrix from one layer to another requires updating n^2 coefficients for an n -dimensional input, assuming the two connected layers are both n -dimensional.

Denoting all the weights to be updated by the vector \mathbf{w} , where \mathbf{w} contains all the elements of the matrices \mathbf{A}_j illustrated in Fig. 6.1, then

$$\mathbf{w}_{k+1} = \mathbf{w}_k + \delta \nabla E \quad (6.21)$$

where the gradient of the error ∇E , through the composition and chain rule, produces the backpropagation algorithm for updating the weights and reduc-

ing the error. Expressed in a component-by-component way

$$w_{k+1}^j = w_k^j + \delta \frac{\partial E}{\partial w_k^j} \quad (6.22)$$

where this equation holds for the j th component of the vector w . The term $\partial E / \partial w^j$ produces the backpropagation through the chain rule, i.e. it produces the sequential set of functions to evaluate as in (6.20). Methods for solving this optimization more quickly, or even simply enabling the computation to be tractable, remain of active research interest. Perhaps the most important method is stochastic gradient descent which is considered in the next section.

6.4 The stochastic gradient descent algorithm

Training neural networks is computationally expensive due to the size of the NNs being trained. Even NNs of modest size can become prohibitively expensive if the optimization routines used for training are not well informed. Two algorithms have been especially critical for enabling the training of NNs: *stochastic gradient descent* (SGD) and backprop. Backprop allows for an efficient computation of the objective function's gradient while SGD provides a more rapid evaluation of the optimal network weights. Although alternative optimization methods for training NNs continue to provide computational improvements, backprop and SGD are both considered here in detail so as to give the reader an idea of the core architecture for building NNs.

Gradient descent was considered in Sec. 4.2. Recall that this algorithm was developed for nonlinear regression where the data fit takes the general form

$$f(x) = f(x, \beta) \quad (6.23)$$

where β are fitting coefficients used to minimize the error. In NNs, the parameters β are the network weights, thus we can rewrite this in the form

$$f(\mathbf{x}) = f(\mathbf{x}, \mathbf{A}_1, \mathbf{A}_2, \dots, \mathbf{A}_M) \quad (6.24)$$

where the \mathbf{A}_j are the connectivity matrices from one layer to the next in the NN. Thus \mathbf{A}_1 connects the first and second layers, and there are M hidden layers.

The goal of training the NN is to minimize the error between the network and the data. The standard root-mean square error for this case is defined as

$$\underset{\mathbf{A}_j}{\operatorname{argmin}} E(\mathbf{A}_1, \mathbf{A}_2, \dots, \mathbf{A}_M) = \underset{\mathbf{A}_j}{\operatorname{argmin}} \sum_{k=1}^n (f(\mathbf{x}_k, \mathbf{A}_1, \mathbf{A}_2, \dots, \mathbf{A}_M) - \mathbf{y}_k)^2 \quad (6.25)$$

which can be minimized by setting the partial derivative with respect to each matrix component to zero, i.e. we require $\partial E / \partial (a_{ij})_k = 0$ where $(a_{ij})_k$ is the

i th row and j th column of the k th matrix ($k = 1, 2, \dots, M$). Recall that the zero derivate is a minimum since there is no maximum error. This gives the gradient $\nabla f(\mathbf{x})$ of the function with respect to the NN parameters. Note further that $f(\cdot)$ is the function evaluated at each of the n data points.

As was shown in Sec. 4.2, this leads to a Newton-Raphson iteration scheme for finding the minima

$$\mathbf{x}_{j+1}(\delta) = \mathbf{x}_j - \delta \nabla f(\mathbf{x}_j) \quad (6.26)$$

where δ is a parameter determining how far a step should be taken along the gradient direction. In NNs, this parameter is called the *learning rate*. Unlike standard gradient descent, it can be computationally prohibitive to compute an optimal learning rate.

Although the optimization formulation is easily constructed, evaluating (6.25) is often computationally intractable for NNs. This due to two reasons: (i) the number of matrix weighting parameters for each \mathbf{A}_j is quite large, and (ii) the number of data points n is generally also large.

To render the computation (6.25) potentially tractable, SGD does not estimate the gradient in (6.26) using all n data points. Rather, a single, randomly chosen data point, or a subset for *batch gradient descent*, is used to approximate the gradient at each step of the iteration. In this case, we can reformulate the least-square fitting of (6.25) so that

$$E(\mathbf{A}_1, \mathbf{A}_2, \dots, \mathbf{A}_M) = \sum_{k=1}^n E_k(\mathbf{A}_1, \mathbf{A}_2, \dots, \mathbf{A}_M) \quad (6.27)$$

and

$$E_k(\mathbf{A}_1, \mathbf{A}_2, \dots, \mathbf{A}_M) = (f_k(\mathbf{x}_k, \mathbf{A}_1, \mathbf{A}_2, \dots, \mathbf{A}_M) - \mathbf{y}_k)^2 \quad (6.28)$$

where $f_k(\cdot)$ is now the fitting function for each data point, and the entries of the matrices \mathbf{A}_j are determined from the optimization process.

The gradient descent iteration algorithm (6.26) is now updated as follows

$$\mathbf{w}_{j+1}(\delta) = \mathbf{w}_j - \delta \nabla f_k(\mathbf{w}_j) \quad (6.29)$$

where \mathbf{w}_j is the vector of all the network weights from \mathbf{A}_j ($j = 1, 2, \dots, M$) at the j th iteration, and the gradient is computed using only the k th data point and $f_k(\cdot)$. Thus instead of computing the gradient with all n points, only a single data point is randomly selected and used. At the next iteration, another randomly selected point is used to compute the gradient and update the solution. The algorithm may require multiple passes through all the data to converge, but each step is now easy to evaluate versus the expensive computation of the Jacobian which is required for the gradient. If instead of a single point, a subset of points is used, then we have the following batch gradient descent algorithm

$$\mathbf{w}_{j+1}(\delta) = \mathbf{w}_j - \delta \nabla f_K(\mathbf{w}_j) \quad (6.30)$$

where $K \in [k_1, k_2, \dots, k_p]$ denotes the p randomly selected data points k_j used to approximate the gradient.

The following code is a modification of the code shown in Sec. 4.2 for gradient descent. The modification here involves taking a significant subsampling of the data to approximate the gradient. Specifically, a batch gradient descent is illustrated with a fixed learning rate of $\delta = 2$. Ten points are used to approximate the gradient of the function at each step.

Code 6.3: Stochastic gradient descent algorithm.

```

h=0.1; x=-6:h:6; y=-6:h:6; n=length(x);
[X,Y]=meshgrid(x,y); clear x, clear y

F1=1.5-1.6*exp(-0.05*(3*(X+3).^2+(Y+3).^2));
F=F1 + (0.5-exp(-0.1*(3*(X-3).^2+(Y-3).^2)));
[dFx,dFy]=gradient(F,h,h);

x0=[4 0 -5]; y0=[0 -5 2]; col=['ro','bo','mo'];
for jj=1:3
    q=randperm(n); i1=sort(q(1:10));
    q2=randperm(n); i2=sort(q2(1:10));
    x(1)=x0(jj); y(1)=y0(jj);
    f(1)=interp2(X(i1,i2),Y(i1,i2),F(i1,i2),x(1),y(1));
    dfx=interp2(X(i1,i2),Y(i1,i2),dFx(i1,i2),x(1),y(1));
    dfy=interp2(X(i1,i2),Y(i1,i2),dFy(i1,i2),x(1),y(1));

    tau=2;
    for j=1:50
        x(j+1)=x(j)-tau*dfx; % update x, y, and f
        y(j+1)=y(j)-tau*dfy;
        q=randperm(n); ind1=sort(q(1:10));
        q2=randperm(n); ind2=sort(q2(1:10));
        f(j+1)=interp2(X(i1,i2),Y(i1,i2),F(i1,i2),x(j+1),y(j+1))
        dfx=interp2(X(i1,i2),Y(i1,i2),dFx(i1,i2),x(j+1),y(j+1));
        dfy=interp2(X(i1,i2),Y(i1,i2),dFy(i1,i2),x(j+1),y(j+1));
        if abs(f(j+1)-f(j))<10^(-6) % check convergence
            break
        end
    end
    if jj==1; x1=x; y1=y; f1=f; end
    if jj==2; x2=x; y2=y; f2=f; end
    if jj==3; x3=x; y3=y; f3=f; end
    clear x, clear y, clear f
end

```

Figure 6.11 shows the convergence of SGD for three initial conditions. As

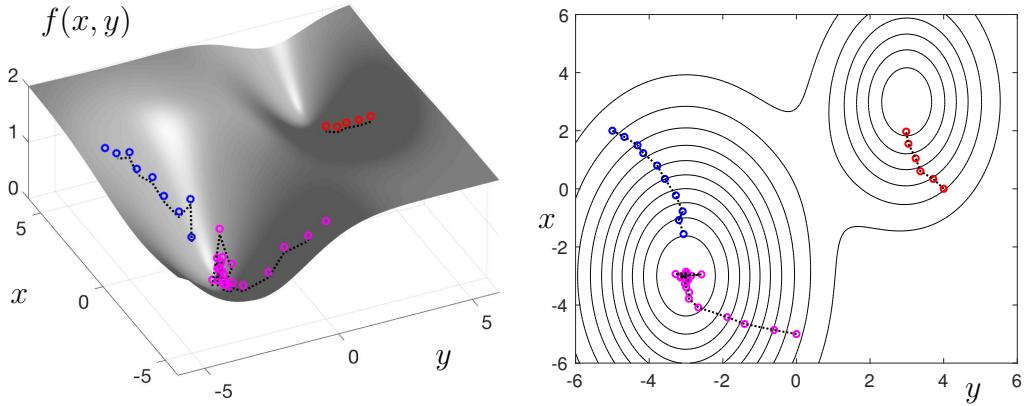


Figure 6.11: Stochastic gradient descent applied to the function featured in Fig. 4.3(b). The convergence can be compared to a full gradient descent algorithm as shown in Fig. 4.6. Each step of the stochastic (batch) gradient descent selects 100 data points for approximating the gradient, instead of the 10^4 data points of the data. Three initial conditions are shown: $(x_0, y_0) = \{(4, 0), (0, -5), (-5, 2)\}$. The first of these (red circles) gets stuck in a local minima while the other two initial conditions (blue and magenta) find the global minima. Interpolation of the gradient functions of Fig. 4.5 are used to update the solutions.

with gradient descent, the algorithm can get stuck in local minima. However, the SGD now approximates the gradient with only 100 points instead of the full 10^4 points, thus allowing for a computation which is three orders of magnitude smaller. Importantly, the SGD is a scalable algorithm, allowing for significant computational savings even as the data grows to be high-dimensional. For this reason, SGD has become a critically enabling part of NN training. Note that the learning rate, batch size, and data sampling play an important role in the convergence of the method.

6.5 Deep convolutional neural networks

With the basics of the NN architecture in hand, along with an understanding of how to formulate an optimization framework (backprop) and actually compute the gradient descent efficiently (SGD), we are ready to construct *deep convolution neural nets* (DCNN) which are the fundamental building blocks of *deep learning* methods. Indeed, today when practitioners generally talk about NNs for practical use, they are typically talking about DCNNs. But as much as we would like to have a principled approach to building DCNNs, there remains a great deal of artistry and expert intuition for producing the highest performing networks. Moreover, DCNNs are especially prone to overtraining, thus requiring special care to cross-validate the results. The recent textbook on deep

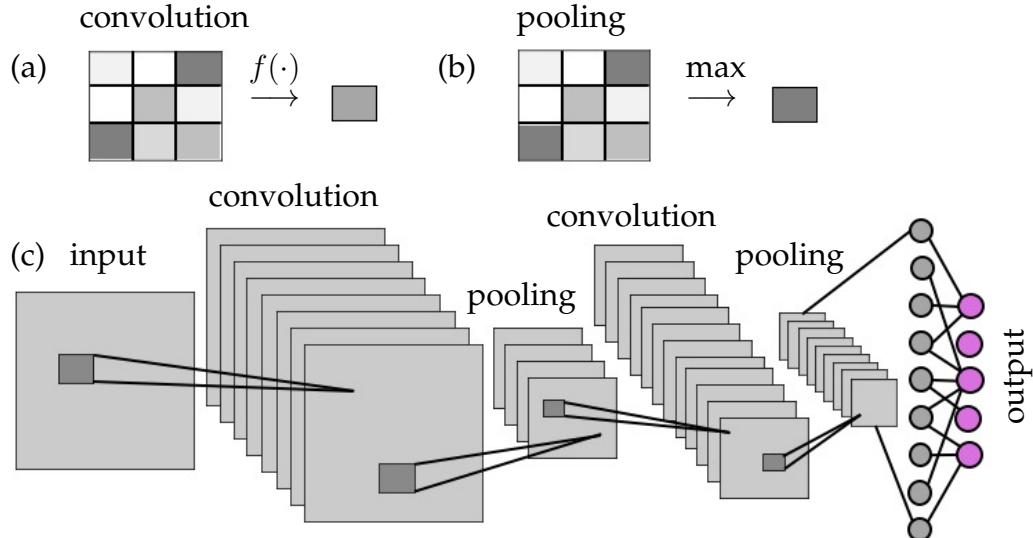


Figure 6.12: Prototypical DCNN architecture which includes commonly used convolutional and pooling layers. The dark gray boxes show the convolutional sampling from layer to layer. Note that for each layer, many functional transformations can be used to produce a variety of feature spaces. The network ultimately integrates all this information into the output layer.

learning by Goodfellow et al. [216] provides a detailed and extensive account of the state-of-the-art in DCNNs. It is especially useful for highlighting many rules-of-thumb and tricks for training effective DCNNs.

Like SVM and random forest algorithms, the MATLAB package for building NNs has a tremendous number of features and tuning parameters. This flexibility is both advantageous and overwhelming at the same time. As was pointed out at the beginning of this chapter, it is immediately evident that there are a great number of design questions regarding NNs. How many layers should be used? What should be the dimension of the layers? How should the output layer be designed? Should one use all-to-all or sparsified connections between layers? How should the mapping between layers be performed: a *linear mapping* or a *nonlinear mapping*?

The prototypical structure of a DCNN is illustrated in Fig. 6.12. Included in the visualization is a number of commonly used convolutional and pooling layers. Also illustrated is the fact that each layer can be used to build multiple downstream layers, or *feature spaces*, that can be engineered by the choice of activation functions and/or network parametrizations. All of these layers are ultimately combined into the output layer. The number of connections that require updating through backprop and SGD can be extraordinarily high, thus even modest networks and training data may require significant computational resources. A typical DCNN is constructed of a number of layers, with DCNNs typically having between 7-10 layers. More recent efforts have considered the advantages of a truly deep network with approximately 100 layers, but the

merits of such architectures are still not fully known. The following paragraphs highlight some of the more prominent elements that comprise DCNNs, including convolutional layers, pooling layers, fully-connected layers and dropout.

Convolutional layers

Convolutional layers are similar to windowed (Gabor) Fourier transforms or wavelets from Chapter 2, in that a small selection of the full high-dimensional input space is extracted and used for feature engineering. Figure 6.12 shows the convolutional windows (dark gray boxes) that are slid across the entire layer (light gray boxes). Each convolution window transforms the data into a new node through a given activation function, as shown in Fig. 6.12(a). The feature spaces are thus built from the smaller patches of the data. Convolutional layers are especially useful for images as they can extract important features such as edges. Wavelets are also known to efficiently extract such features and there are deep mathematical connections between wavelets and DCNNs as shown by Mallat and co-workers [358, 12]. Note that in Fig. 6.12, the input layer can be used to construct many layers by simply manipulating the activation function $f(\cdot)$ to the next layer as well the size of the convolutional window.

Pooling layers

It is common to periodically insert a Pooling layer between successive convolutional layers in a DCNN architecture. Its function is to progressively reduce the spatial size of the representation in order to reduce the number of parameters and computation in the network. This is an effective strategy to (i) help control overfitting and (ii) fit the computation in memory. Pooling layers operate independently on every depth slice of the input and resize them spatially. Using the max operation, i.e. the maximum value for all the nodes in its convolutional window, is called *max pooling*. In image processing, the most common form of max pooling is a pooling layer with filters of size 2×2 applied with a stride of 2 downsamples every depth slice in the input by 2 along both width and height, discarding 75% of the activations. Every max pooling operation would in this case be taking a max over 4 numbers (a 2×2 region in some depth slice). The depth dimension remains unchanged. An example max pooling operation is shown in Fig. 6.12(b), where a 3×3 convolutional cell is transformed to a single number which is the maximum of the 9 numbers.

Fully-connected layers

Occasionally, fully-connected layers are inserted into the DCNN so that different regions can be connected. The pooling and convolutional layers are *local*

connections only, while the fully-connected layer restores *global* connectivity. This is another commonly used layer in the DCNN architecture, providing a potentially important feature space to improve performance.

Dropout

Overfitting is a serious problem in DCNNs. Indeed, overfitting is at the core of why DCNNs often fail to demonstrate good generalizability properties (See Chapter 4 on regression). Large DCNNs are also slow to use, making it difficult to deal with overfitting by combining the predictions of many different large neural nets for online implementation. Dropout is a technique which helps address this problem. The key idea is to randomly drop nodes in the network (along with their connections) from the DCNN during training, i.e. during SGD/backprop updates of the network weights. This prevents units from co-adapting too much. During training, dropout samples form an exponential number of different “thinned” networks. This idea is similar to the ensemble methods for building random forests. At test time, it is easy to approximate the effect of averaging the predictions of all these thinned networks by simply using a single unthinned network that has smaller weights. This significantly reduces overfitting and has shown to give major improvements over other regularization methods [499].

There are many other techniques that have been devised for training DCNNs, but the above methods highlight some of the most commonly used. The most successful applications of these techniques tend to be in computer vision tasks where DCNNs offer unparalleled performance in comparison to other machine learning methods. Importantly, the ImageNET data set is what allowed these DCNN layers to be maximally leveraged for human level recognition performance.

To illustrate how to train and execute a DCNN, we use data from MATLAB. Specifically, we use a data set that has a training and test set with the alphabet characters A, B, and C. The following code loads the data set and plots a representative sample of the characters in Fig. 6.13.

Code 6.4: Loading alphabet images.

```
load lettersTrainSet
perm = randperm(1500, 20);
for j = 1:20
    subplot(4,5,j);
    imshow(XTrain(:,:,:,:perm(j)));
end
```

This code loads the training data, XTrain, that contains 1500 28×28 grayscale images of the letters A, B, and C in a 4-D array. There are equal numbers of

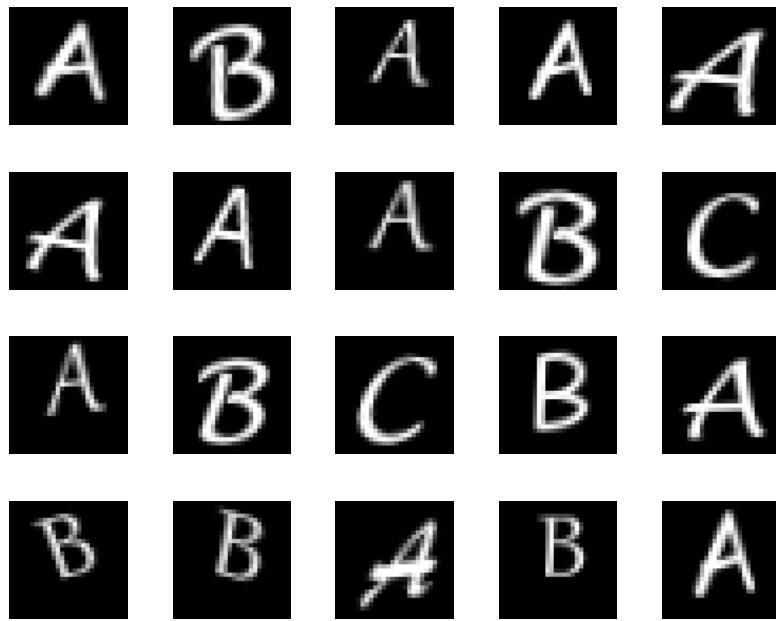


Figure 6.13: Representative images of the alphabet characters A, B, and C. There are a total of 1500 28×28 grayscale images (XTrain) of the letters that are labeled (TTrain).

each letter in the data set. The variable TTrain contains the categorical array of the letter labels, i.e. the truth labels. The following code constructs and trains a DCNN.

Code 6.5: Train a DCNN.

```

layers = [imageInputLayer([28 28 1]);
          convolution2dLayer(5,16);
          reluLayer();
          maxPooling2dLayer(2,'Stride',2);
          fullyConnectedLayer(3);
          softmaxLayer();
          classificationLayer()];
options = trainingOptions('sgdm');
rng('default') % For reproducibility
net = trainNetwork(XTrain,TTrain,layers,options);

```

Note the simplicity in how diverse network layers are easily put together. In addition, a ReLu activation layer is specified along with the training method of stochastic gradient descent (sgdm). The `trainNetwork` command integrates the options and layer specifications to build the best classifier possible. The resulting trained network can now be used on a test data set.

Code 6.6: Test the DCNN performance.

```
||| load lettersTestSet;
||| YTest = classify(net, XTest);
||| accuracy = sum(YTest == TTest) / numel(TTest)
```

The resulting classification performance is approximately 93%. One can see by this code structure that modifying the network architecture and specifications is trivial. Indeed, one can probably easily engineer a network to outperform the illustrated DCNN. As already mentioned, artistry and expert intuition are critical for producing the highest performing networks.

6.6 Neural networks for dynamical systems

Neural networks offer an amazingly flexible architecture for performing a diverse set of mathematical tasks. To return to S. Mallat: *Supervised learning is a high-dimensional interpolation problem* [358]. Thus if sufficiently rich data can be acquired, NNs offer the ability to interrogate the data for a variety of tasks centered on classification and prediction. To this point, the tasks demonstrated have primarily been concerned with computer vision. However, NNs can also be used for future state predictions of dynamical systems (See Chapter 7).

To demonstrate the usefulness of NNs for applications in dynamical systems, we will consider the Lorenz system of differential equations [345]

$$\dot{x} = \sigma(y - x) \quad (6.31a)$$

$$\dot{y} = x(\rho - z) - y \quad (6.31b)$$

$$\dot{z} = xy - \beta z, \quad (6.31c)$$

where the state of the system is given by $\mathbf{x} = [x \ y \ z]^T$ with the parameters $\sigma = 10$, $\rho = 28$, and $\beta = 8/3$. This system will be considered in further detail in the next chapter. For the present, we will simulate this nonlinear system and use it as a demonstration of how NNs can be trained to characterize dynamical systems. Specifically, the goal of this section is to demonstrate that we can train a NN to learn an update rule which advances the state space from \mathbf{x}_k to \mathbf{x}_{k+1} , where k denotes the state of the system at time t_k . Accurately advancing the solution in time requires a nonlinear transfer function since Lorenz itself is nonlinear.

The training data required for the NN is constructed from high-accuracy simulations of the Lorenz system. The following code generates a diverse set of initial conditions. One hundred initial conditions are considered in order to generate one hundred trajectories. The sampling time is fixed at $\Delta t = 0.01$. Note that the sampling time is not the same as the time-steps taken by the 4th-order Runge-Kutta method [316]. The time-steps are adaptively chosen to meet the stringent tolerances of accuracy chosen for this example.

Code 6.7: Create training data of Lorenz trajectories.

```
% Simulate Lorenz system
dt=0.01; T=8; t=0:dt:T;
b=8/3; sig=10; r=28;

Lorenz = @(t,x)([ sig * (x(2) - x(1)) ; ...
                  r * x(1)-x(1) * x(3) - x(2) ; ...
                  x(1) * x(2) - b*x(3) ]);
ode_options = odeset('RelTol',1e-10, 'AbsTol',1e-11);

input=[]; output=[];
for j=1:100 % training trajectories
    x0=30*(rand(3,1)-0.5);
    [t,y] = ode45(Lorenz,t,x0);
    input=[input; y(1:end-1,:)];
    output=[output; y(2:end,:)];
    plot3(y(:,1),y(:,2),y(:,3)), hold on
    plot3(x0(1),x0(2),x0(3),'ro')
end
```

The simulation of the Lorenz system produces two key matrices: `input` and `output`. The former is a matrix of the system at \mathbf{x}_k , while the latter is the corresponding state of the system \mathbf{x}_{k+1} advanced $\Delta t = 0.01$.

The NN must learn the nonlinear mapping from \mathbf{x}_k to \mathbf{x}_{k+1} . Figure 6.14 shows the various trajectories used to train the NN. Note the diversity of initial conditions and the underlying attractor of the Lorenz system.

We now build a NN trained on trajectories of Fig. 6.14 to advance the solution $\Delta t = 0.01$ into the future for an arbitrary initial condition. Here, a three-layer network is constructed with ten nodes in each layer and a different activation unit for each layer. The choice of activation types, nodes in the layer and number of layers are arbitrary. It is trivial to make the network deeper and wider and enforce different activation units. The performance of the NN for the arbitrary choices made is quite remarkable and does not require additional tuning. The NN is built with the following few lines of code.

Code 6.8: Build a neural network for Lorenz system.

```
net = feedforwardnet([10 10 10]);
net.layers{1}.transferFcn = 'logsig';
net.layers{2}.transferFcn = 'radbas';
net.layers{3}.transferFcn = 'purelin';
net = train(net,input.',output.');
```

The code produces a function `net` which can be used with a new set of data to produce predictions of the future. Specifically, the function `net` gives the nonlinear mapping from \mathbf{x}_k to \mathbf{x}_{k+1} . Figure 6.15 shows the structure of the net-

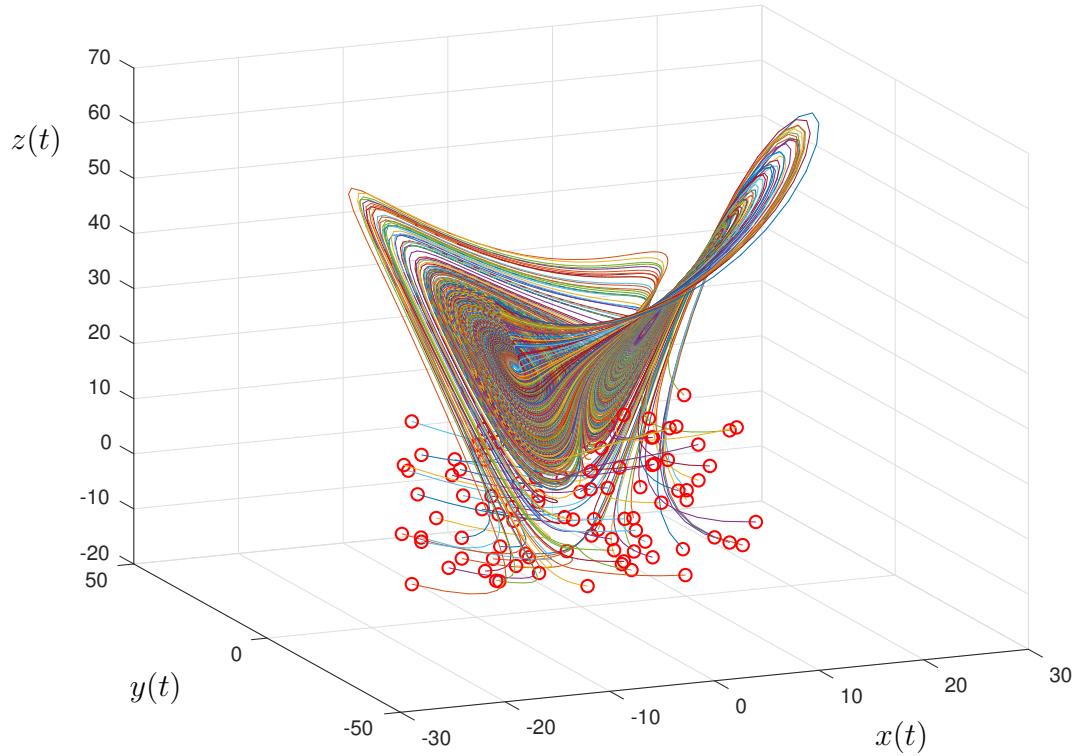


Figure 6.14: Evolution of the Lorenz dynamical equations for one hundred randomly chosen initial conditions (red circles). For the parameters $\sigma = 10$, $\rho = 28$, and $\beta = 8/3$, all trajectories collapse to an attractor. These trajectories, generated from a diverse set of initial data, are used to train a neural network to learn the nonlinear mapping from \mathbf{x}_k to \mathbf{x}_{k+1} .

work along with the performance of the training over 1000 epochs of training. The results of the cross-validation are also demonstrated. The NN converges steadily to a network that produces accuracies on the order of 10^{-5} .

Once the NN is trained on the trajectory data, the nonlinear model mapping \mathbf{x}_k to \mathbf{x}_{k+1} can be used to predict the future state of the system from an initial condition. In the following code, the trained function `net` is used to take an initial condition and advance the solution Δt . The output can be re-inserted into the `net` function to estimate the solution $2\Delta t$ into the future. This iterative mapping can produce a prediction for the future state as far into the future as desired. In what follows, the mapping is used to predict the Lorenz solutions eight time units into the future from a given initial condition. This can then be compared against the ground truth simulation of the evolution using a 4th-order Runge-Kutta method. The following iteration scheme gives the NN approximation to the dynamics.

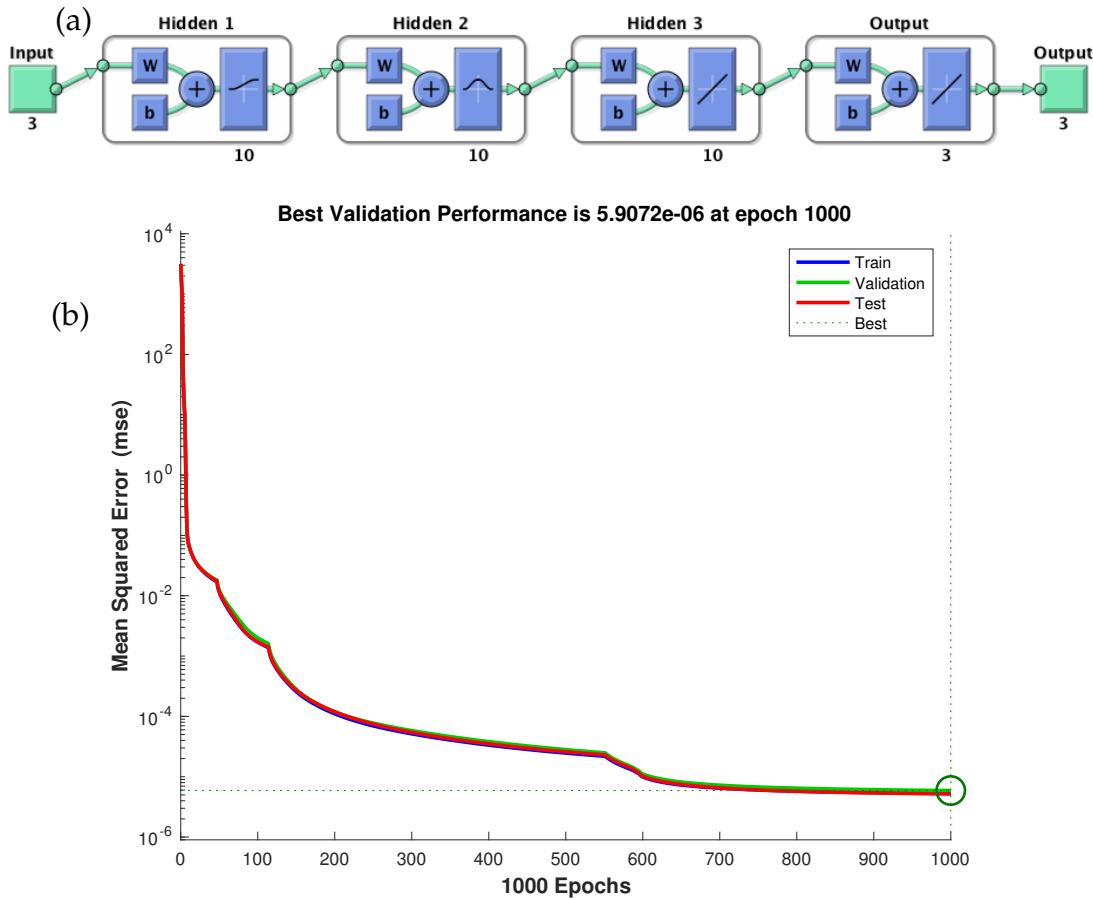


Figure 6.15: (a) Network architecture used to train the NN on the trajectory data of Fig. 6.14. A three-layer network is constructed with ten nodes in each layer and a different activation unit for each layer. (b) Performance summary of the NN optimization algorithm. Over 1000 epochs of training, accuracies on the order of 10^{-5} are produced. The NN is also cross-validated in the process.

Code 6.9: Neural network for prediction.

```

ynn(1,:)=x0;
for jj=2:length(t)
    y0=net(x0);
    ynn(jj,:)=y0.'; x0=y0;
end
plot3(ynn(:,1),ynn(:,2),ynn(:,3),':','Linewidth',[2])

```

Figure 6.16 shows the evolution of two randomly drawn trajectories (solid lines) compared against the NN prediction of the trajectories (dotted lines). The NN prediction is remarkably accurate in producing an approximation to the high-accuracy simulations. This shows that the data used for training is capable of producing a high-quality nonlinear model mapping x_k to x_{k+1} . The quality

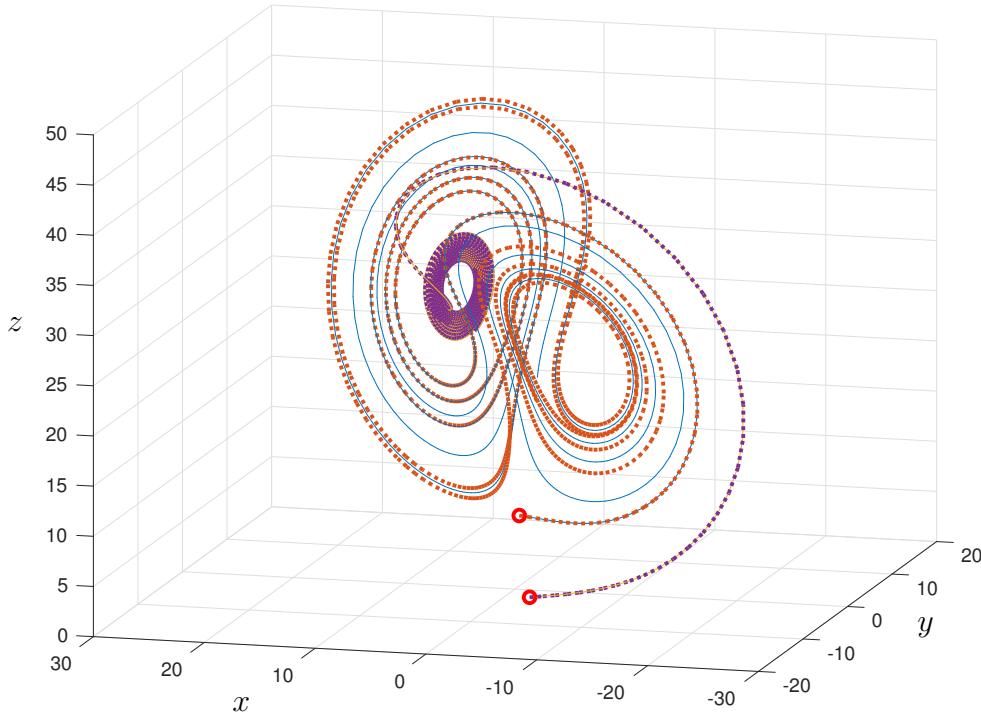


Figure 6.16: Comparison of the time evolution of the Lorenz system (solid line) with the NN prediction (dotted line) for two randomly chosen initial conditions (red dots). The NN prediction stays close to the dynamical trajectory of the Lorenz model. A more detailed comparison is given in Fig. 6.17.

of the approximation is more clearly seen in Fig. 6.17 where the time evolution of the individual components of x are shown against the NN predictions. See Sec. 7.5 for further details.

In conclusion, the NN can be trained to learn dynamics. More precisely, the NN seems to learn an algorithm which is approximately equivalent to a 4th-order Runge-Kutta scheme for advancing the solution a time-step Δt . Indeed, NNs have been used to model dynamical systems [215] and other physical processes [381] for decades. However, great strides have been made recently in using DNNs to learn Koopman embeddings, resulting in several excellent papers [550, 368, 513, 564, 412, 332]. For example, the VAMPnet architecture [550, 368] uses a time-lagged auto-encoder and a custom variational score to identify Koopman coordinates on an impressive protein folding example. In an alternative formulation, variational auto-encoders can build low-rank models that are efficient and compact representations of the Koopman operator from data [349]. By construction, the resulting network is both parsimonious and interpretable, retaining the flexibility of neural networks and the physical interpretation of Koopman theory. In all of these recent studies, DNN representations have been shown to be more flexible and exhibit higher accuracy than other leading methods on challenging problems.

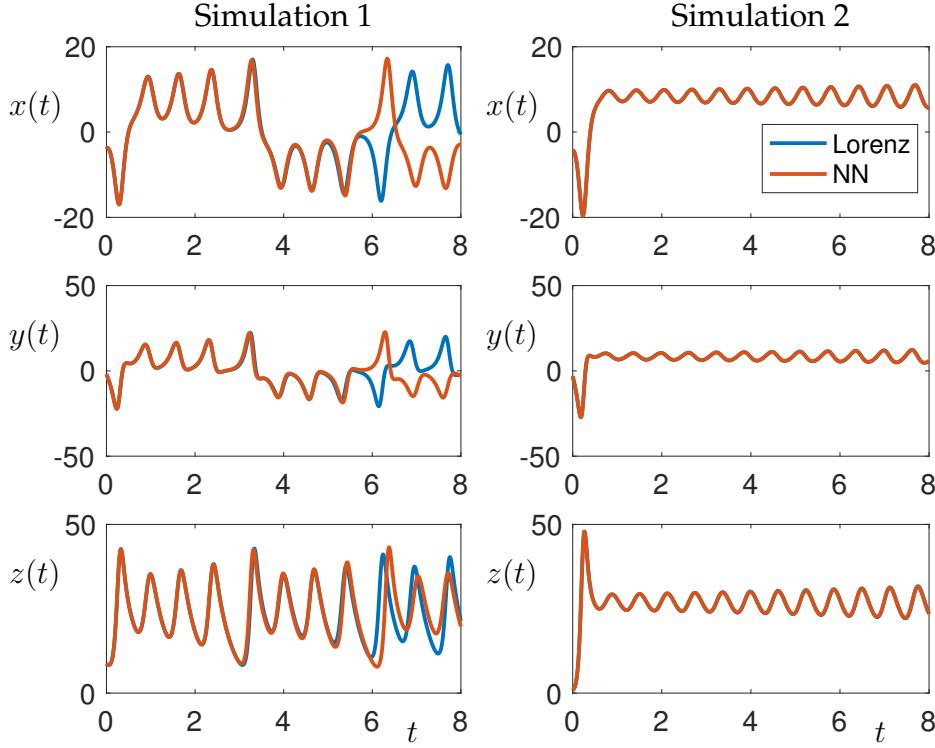


Figure 6.17: Comparison of the time evolution of the Lorenz system for two randomly chosen initial conditions (Also shown in Fig. 6.16). The left column shows that the evolution of the Lorenz differential equations and the NN mapping gives identical results until $t \approx 5.5$, at which point they diverge. In contrast, the NN prediction stays on the trajectory of the second initial condition for the entire time window.

6.7 The diversity of neural networks

There are a wide variety of NN architectures, with only a few of the most dominant architectures considered thus far. This chapter and book does not attempt to give a comprehensive assessment of the state-of-the-art in neural networks. Rather, our focus is on illustrating some of the key concepts and enabling mathematical architectures that have led NNs to a dominant position in modern data science. For a more in-depth review, please see [216]. However, to conclude this chapter, we would like to highlight some of the NN architectures that are used in practice for various data science tasks. This overview is inspired by the *neural network zoo* as highlighted by Fjodor Van Veen of the Asimov Institute (<http://www.asimovinstitute.org>).

The neural network zoo highlights some of the different architectural structures around NNs. Some of the networks highlighted are commonly used across industry, while others serve niche roles for specific applications. Regardless, it demonstrates that tremendous variability and research effort focused on

NNs as a core data science tool. Figure 6.18 highlights the prototype structures to be discussed in what follows. Note that the bottom panel has a key to the different type of nodes in the network, including input cells, output cells, and hidden cells. Additionally, the hidden layer NN cells can have memory effects, kernel structures and/or convolution/pooling. For each NN architecture, a brief description is given along with the original paper proposing the technique.

Perceptron

The first mathematical model of NNs by Fukushima was termed the Neocognitron in 1980 [193]. His model had a single layer with a single output cell called the perceptron, which made a categorial decision based on the sign of the output. Figure 6.2 shows this architecture to classify between dogs and cats. The perceptron is an algorithm for supervised learning of binary classifiers.

Feed forward (FF)

Feed forward networks connect the input layer to output layer by forming connections between the units so that they do not form a cycle. Figure 6.1 has already shown a version of this architecture where the information simply propagates from left to right in the network. It is often the workhorse of supervised learning where the weights are trained so as to best classify a given set of data. A feedforward network was used in Figs. 6.5 and 6.15 for training a classifier for dogs versus cats and predicting time-steps of the Lorenz attractor respectively. An important subclass of feed forward networks is *deep feed forward* (DFF) NNs. DFFs simply put together a larger number of hidden layers, typically 7-10 layers, to form the NN. A second important class of FF is the *radial basis network*, which uses radial basis functions as the activation units [88]. Like any FF network, radial basis function networks have many uses, including function approximation, time series prediction, classification, and control.

Recurrent neural network (RNN)

Illustrated in Fig. 6.18(a), RNNs are characterized by connections between units that form a directed graph along a sequence. This allows it to exhibit dynamic temporal behavior for a time sequence [172]. Unlike feedforward neural networks, RNNs can use their internal state (memory) to process sequences of inputs. The prototypical architecture in Fig. 6.18(a) shows that each cell feeds back on itself. This self-interaction, which is not part of the FF architecture, allows for a variety of innovations. Specifically, it allows for time delays and/or feedback loops. Such controlled states are referred to as gated state or gated

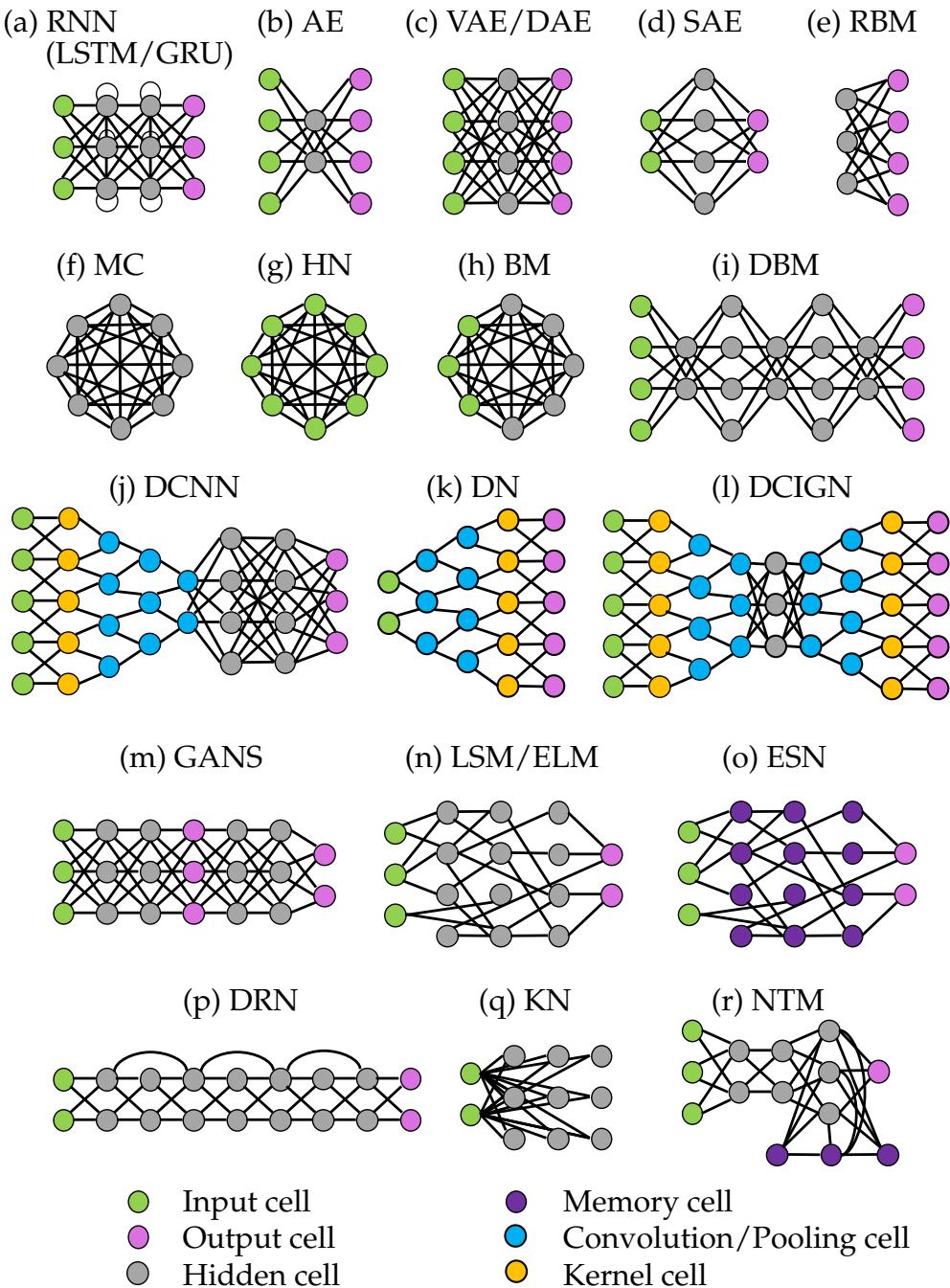


Figure 6.18: Neural network architectures commonly considered in the literature. The NNs are comprised of input nodes, output nodes, and hidden nodes. Additionally, the nodes can have memory, perform convolution and/or pooling, and perform a kernel transformation. Each network, and their acronym is explained in the text.

memory, and are part of two key innovations: *long-short term memory* (LSTM) networks [248] and *gated recurrent units* (GRU) [132]. LSTM is of particular importance as it revolutionized speech recognition, setting a variety of performance records and outperforming traditional models in a variety of speech applications. GRUs are a variation of LSTMs which have been demonstrated to exhibit better performance on smaller datasets.

Auto encoder (AE)

The aim of an auto encoder, represented in Fig. 6.18(b), is to learn a representation (encoding) for a set of data, typically for the purpose of dimensionality reduction. For AEs, the input and output cells are matched so that the AE is essentially constructed to be a nonlinear transform into and out of a new representation, acting as an approximate identity map on the data. Thus AEs can be thought of as a generalization of linear dimensionality reduction techniques such as PCA. AEs can potentially produce nonlinear PCA representations of the data, or nonlinear manifolds on which the data should be embedded [71]. Since most data lives in nonlinear subspaces, AEs are an important class of NN for data science, with many innovations and modifications. Three important modifications of the standard AE are commonly used. The *variational auto encoder* (VAE) [290] (shown in Fig. 6.18(c)) is a popular approach to unsupervised learning of complicated distributions. By making strong assumptions concerning the distribution of latent variables, it can be trained using standard gradient descent algorithms to provide a good assessments of data in an unsupervised fashion. The *denoising auto encoder* (DAE) [541] (shown in Fig. 6.18(c)) takes a partially corrupted input during training to recover the original undistorted input. Thus noise is intentionally added to the input in order to learn the nonlinear embedding. Finally, the *sparse auto encoder* (SAE) [432] (shown in Fig. 6.18(d)) imposes sparsity on the hidden units during training, while having a larger number of hidden units than inputs, so that an autoencoder can learn useful structures in the input data. Sparsity is typically imposed by thresholding all but the few strongest hidden unit activations.

Markov chain (MC)

A Markov chain is a stochastic model describing a sequence of possible events in which the probability of each event depends only on the state attained in the previous event. So although not formally a NN, it shares many common features with RNNs. Markov chains are standard even in undergraduate probability and statistics courses. Figure 6.18(f) shows the basic architecture where each cell is connected to the other cells by a probability model for a transition.

Hopfield network (HN)

A Hopfield network is a form of a RNN which was popularized by John Hopfield in 1982 for understanding human memory [254]. Figure 6.18(g) shows the basic architecture of an all-to-all connected network where each node can act as an input cell. The network serves as a trainable content-addressable *associative* memory system with binary threshold nodes. Given an input, it is iterated on the network with a guarantee to converge to a local minimum. Sometimes it converge to a false pattern, or memory (wrong local minimum), rather than the stored pattern (expected local minimum).

Boltzmann machine (BM)

The Boltzmann machine, sometimes called a stochastic Hopfield network with hidden units, is a stochastic, generative counterpart of the Hopfield network. They were one of the first neural networks capable of learning internal representations, and are able to represent and (given sufficient time) solve difficult combinatoric problems [246]. Figure 6.18(h) shows the structure of the BM. Note that unlike Markov chains (which have no input units) or Hopfield networks (where all cells are inputs), the BM is a hybrid which has a mixture of input cells and hidden units. Boltzmann machines are intuitively appealing due to their resemblance to the dynamics of simple physical processes. They are named after the Boltzmann distribution in statistical mechanics, which is used in their sampling function.

Restricted Boltzmann machine (RBM)

Introduced under the name *Harmonium* by Paul Smolensky in 1986 [493], RBMs have been proposed for dimensionality reduction, classification, collaborative filtering, feature learning, and topic modeling. They can be trained for either supervised or unsupervised tasks. G. Hinton helped bring them to prominence by developing fast algorithms for evaluating them [397]. RBMs are a subset of BMs where restrictions are imposed on the NN such that nodes in the NN must form a bipartite graph (See Fig. 6.18(e)). Thus a pair of nodes from each of the two groups of units (commonly referred to as the “visible” and “hidden” units, respectively) may have a symmetric connection between them; there are no connections between nodes within a group. RBMs can be used in deep learning networks and deep belief networks by stacking RBMs and optionally fine-tuning the resulting deep network with gradient descent and backpropagation.

Deep belief network (DBN)

DBNs are a generative graphical model that are composed of multiple layers of latent hidden variables, with connections between the layers but not between units within each layer [52]. Figure 6.18(i) shows the architecture of the DBN. The training of the DBNs can be done stack by stack from AE or RBM layers. Thus each of these layers only has to learn to encode the previous network, which is effectively a greedy training algorithm for finding locally optimal solutions. Thus DBNs can be viewed as a composition of simple, unsupervised networks such as RBMs and AEs where each sub-network's hidden layer serves as the visible layer for the next.

Deep convolutional neural network (DCNN)

DCNNs are the workhorse of computer vision and have already been considered in this chapter. They are abstractly represented in Fig. 6.18(j), and in a more specific fashion in Fig. 6.12. Their impact and influence on computer vision cannot be overestimated. They were originally developed for document recognition [325].

Deconvolutional network (DN)

Deconvolutional Networks, shown in Fig. 6.18(k), are essentially a reverse of DCNNs [567]. The mathematical structure of DNs permit the unsupervised construction of hierarchical image representations. These representations can be used for both low-level tasks such as denoising, as well as providing features for object recognition. Each level of the hierarchy groups information from the level beneath to form more complex features that exist over a larger scale in the image. As with DCNNs, it is well suited for computer vision tasks.

Deep convolutional inverse graphics network (DCIGN)

The DCIGN is a form of a VAE that uses DCNNs for the encoding and decoding [313]. As with the AE/VAE/SAE structures, the output layer shown in Fig. 6.18(l) is constrained to match the input layer. DCIGN combine the power of DCNNs with VAEs, which provides a formative mathematical architecture for computer visions and image processing.

Generative adversarial network (GAN)

In an innovative modification of NNs, the GAN architecture of Fig. 6.18(m) trains two networks simultaneously [217]. The networks, often which are a combination of DCNNs and/or FFs, train by one of the networks generating

content which the other attempts to judge. Specifically, one network generates candidates and the other evaluates them. Typically, the generative network learns to map from a latent space to a particular data distribution of interest, while the discriminative network discriminates between instances from the true data distribution and candidates produced by the generator. The generative network's training objective is to increase the error rate of the discriminative network (i.e., "fool" the discriminator network by producing novel synthesized instances that appear to have come from the true data distribution). The GAN architecture has produced interesting results in computer vision for producing synthetic data, such as images and movies.

Liquid state machine (LSM)

The LSM shown in Fig. 6.18(n) is a particular kind of spiking neural network [352]. An LSM consists of a large collection of nodes, each of which receives time varying input from external sources (the inputs) as well as from other nodes. Nodes are randomly connected to each other. The recurrent nature of the connections turns the time varying input into a spatio-temporal pattern of activations in the network nodes. The spatio-temporal patterns of activation are read out by linear discriminant units. This architecture is motivated by spiking neurons in the brain, thus helping understand how information processing and discrimination might happen using spiking neurons.

Extreme learning machine (ELM)

With the same underlying architecture of an LSM shown in Fig. 6.18(n), the ELM is a FF network for classification, regression, clustering, sparse approximation, compression and feature learning with a single layer or multiple layers of hidden nodes, where the parameters of hidden nodes (not just the weights connecting inputs to hidden nodes) need not be tuned. These hidden nodes can be randomly assigned and never updated, or can be inherited from their ancestors without being changed. In most cases, the output weights of hidden nodes are usually learned in a single step, which essentially amounts to learning a linear model [108].

Echo state network (ESN)

ESNs are RNNs with a sparsely connected hidden layer (with typically 1% connectivity). The connectivity and weights of hidden neurons have memory and are fixed and randomly assigned (See Fig. 6.18(o)). Thus like LSMs and ELMs they are not fixed into a well-ordered layered structure. The weights of output neurons can be learned so that the network can generate specific temporal

patterns [263].

Deep residual network (DRN)

DRNs took the deep learning world by storm when Microsoft Research released Deep Residual Learning for Image Recognition [237]. These networks led to 1st-place winning entries in all five main tracks of the ImageNet and COCO 2015 competitions, which covered image classification, object detection, and semantic segmentation. The robustness of ResNets has since been proven by various visual recognition tasks and by non-visual tasks involving speech and language. DRNs are very deep FF networks where there are extra connections that pass from one layer to a layer two to five layers downstream. This then carries input from an earlier stage to a future stage. These networks can be 150 layers deep, which is only abstractly represented in Fig. 6.18(p).

Kohonen network (KN)

Kohonen networks are also known as self-organizing feature maps [298]. KNs use competitive learning to classify data without supervision. Input is presented to the KN as in Fig. 6.18(q), after which the network assesses which of the neurons closely match that input. These self-organizing maps differ from other NNs as they apply competitive learning as opposed to error-correction learning (such as backpropagation with gradient descent), and in the sense that they use a neighborhood function to preserve the topological properties of the input space. This makes KNs useful for low-dimensional visualization of high-dimensional data.

Neural Turing machine (NTM)

An NTM implements a NN controller coupled to an external memory resource (See Fig. 6.18(r)), which it interacts with through attentional mechanisms [219]. The memory interactions are differentiable end-to-end, making it possible to optimize them using gradient descent. An NTM with a LSTM controller can infer simple algorithms such as copying, sorting, and associative recall from input and output examples.

Suggested reading

Texts

- (1) **Deep learning**, by I. Goodfellow, Y. Bengio and A. Courville, 2016 [216].
- (2) **Neural networks for pattern recognition**, by C. M. Bishop, 1995 [63].

Papers and reviews

- (1) **Deep learning**, by Y. LeCun, Y. Bengio and G. Hinton, *Nature*, 2015 [324].
- (2) **Understanding deep convolutional networks**, by S. Mallat, *Phil. Trans. R. Soc. A*, 2016 [358].
- (3) **Deep learning: mathematics and neuroscience**, by T. Poggio, *Views & Reviews, McGovern Center for Brains, Minds and Machines*, 2016 [430].
- (4) **Imagenet classification with deep convolutional neural**, by A. Krizhevsky, I. Sutskever and G. Hinton, *Advances in neural information processing systems*, 2012 [310].