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 multilayer 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

$$\underset{\mathbf{A}_j}{\operatorname{argmin}} \left(f_M(\mathbf{A}_M, \cdots, f_2(\mathbf{A}_2, f_1(\mathbf{A}_1, \mathbf{x})) \cdots) + \lambda g(\mathbf{A}_j) \right) \tag{6.1}$$

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 kth 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.

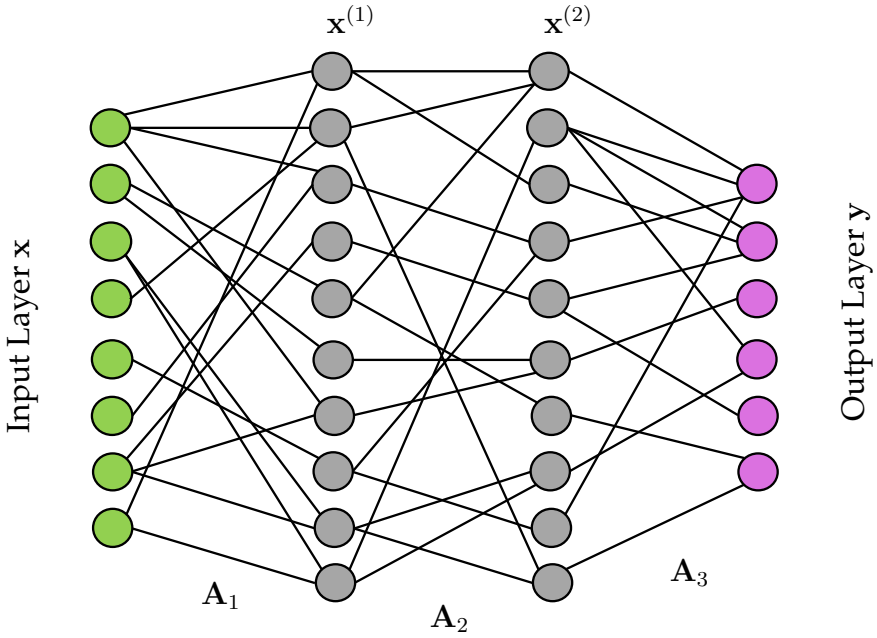

Figure 6.1: Illustration of a neural net architecture mapping an input layer $\mathbf{x}$ to an output layer $\mathbf{y}$. The middle (hidden) layers are denoted $\mathbf{x}^{(j)}$ where j determines their sequential ordering. The matrices $\mathbf{A}_j$ contain the coefficients that map each variable from one layer to the next. Although the dimensionality of the input layer $\mathbf{x} \in \mathbb{R}^n$ 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

$$\mathbf{x}^{(1)} = \mathbf{A}_1\mathbf{x} \tag{6.2a}$$
$$\mathbf{x}^{(2)} = \mathbf{A}_2\mathbf{x}^{(1)} \tag{6.2b}$$
$$\mathbf{y} = \mathbf{A}_3\mathbf{x}^{(2)}. \tag{6.2c}$$

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

$$\mathbf{y} = \mathbf{A}_3\mathbf{A}_2\mathbf{A}_1\mathbf{x}. \tag{6.3}$$

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

$$\mathbf{y} = \mathbf{A}_M\mathbf{A}_{M-1}\cdots\mathbf{A}_2\mathbf{A}_1\mathbf{x}. \tag{6.4}$$

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

$$\begin{aligned} \mathbf{x}^{(1)} &= f_1(\mathbf{A}_1, \mathbf{x}) && \text{(6.5a)} \\ \mathbf{x}^{(2)} &= f_2(\mathbf{A}_2, \mathbf{x}^{(1)}) && \text{(6.5b)} \\ \mathbf{y} &= f_3(\mathbf{A}_3, \mathbf{x}^{(2)}). && \text{(6.5c)} \end{aligned}$$

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

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

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

$$\mathbf{y} = \{\text{dog}, \text{cat}\} = \{+1, -1\} \tag{6.7}$$

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 $\mathbf{y}_j$.

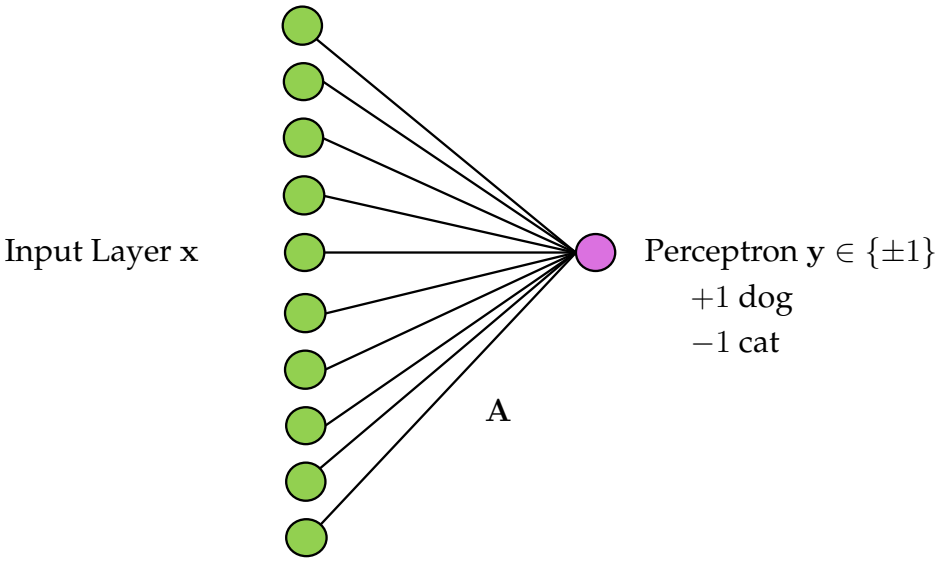

Figure 6.2: Single layer network for binary classification between dogs and cats. The output layer for this case is a perceptron with $\mathbf{y} \in \{\pm 1\}$. A linear mapping between the input image space and output output layer can be constructed for training data by solving $\mathbf{A} = \mathbf{Y}\mathbf{X}^\dagger$.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

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

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 $\mathbf{X}$

$$\mathbf{A} = \mathbf{Y}\mathbf{X}^\dagger. \tag{6.9}$$

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)];
```

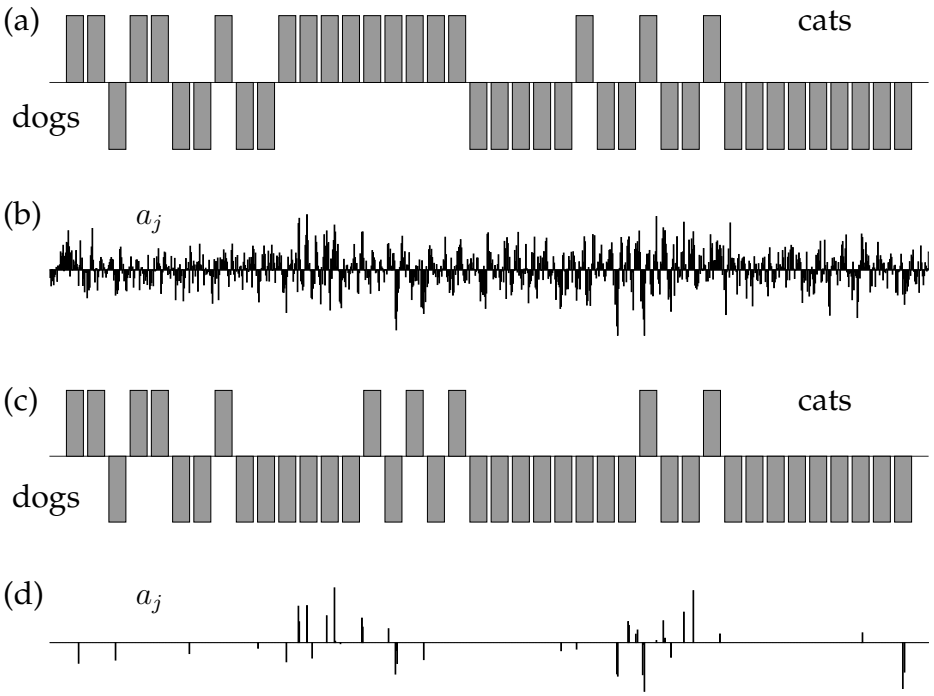

Figure 6.3: Classification of withheld data tested on a trained, single-layer network with linear mapping between inputs (pixel space) and a single output. (a) and (c) are the bar graph of the output layer score $\mathbf{y} \in \{\pm 1\}$ achieved for the withheld data using a pseudo-inverse for training 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 $\mathbf{A}$ 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)
```

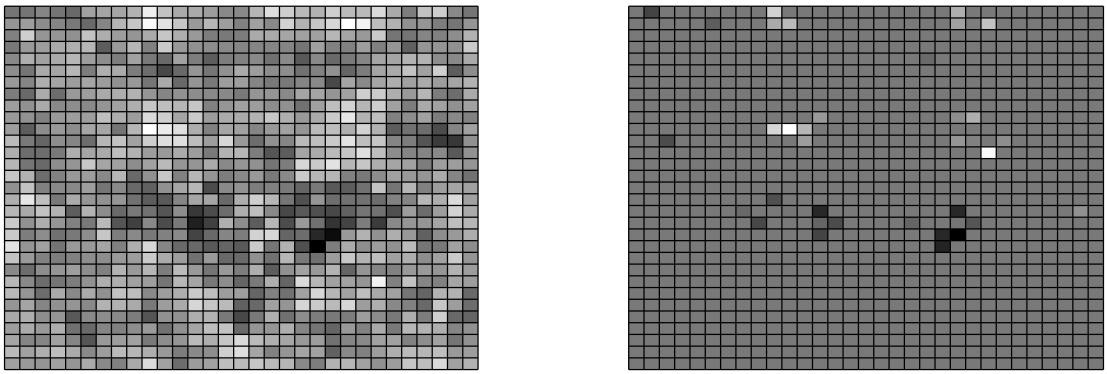

Figure 6.4: Weightings of the matrix $\mathbf{A}$ reshaped into 32×32 arrays. The left matrix shows the matrix $\mathbf{A}$ computed by least-square regression (the pseudo-inverse) while the right matrix shows the matrix $\mathbf{A}$ computed by LASSO. Both matrices provide similar classification scores 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 $\mathbf{A}$. 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 $\mathbf{A}$ for the two solution strategies reshaped into 32×32 images. Note that for the pseudo-inverse, the weightings of the matrix elements $\mathbf{A}$ 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 $\mathbf{A}$.

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: $\mathbf{Ax} = \mathbf{y}$ in (6.8). We highlight here common nonlinear

transformations from input-to-output space represented by

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

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

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

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

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

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

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

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 $\mathbf{A}_1$, $\mathbf{A}_2, \cdots \mathbf{A}_M$, 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

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

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

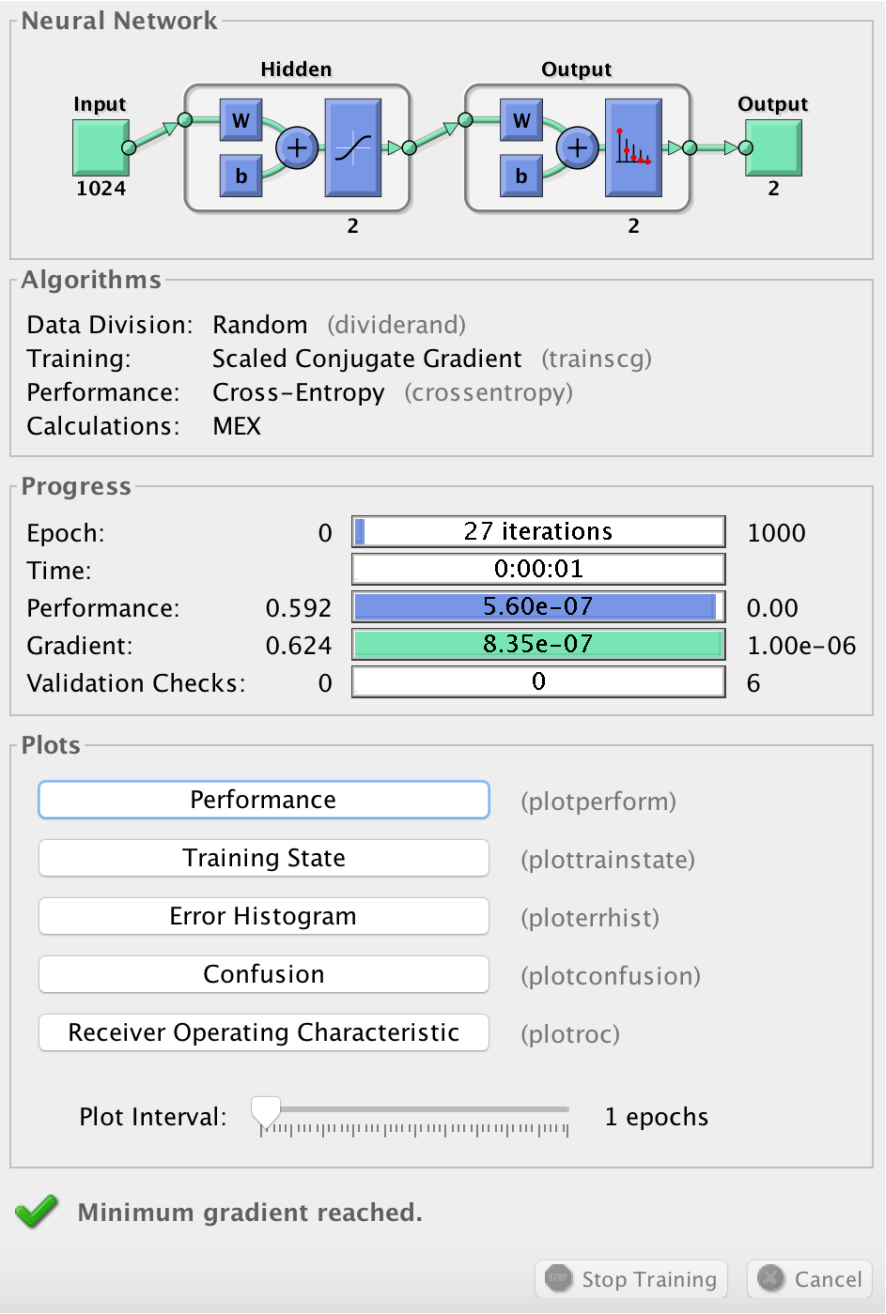

Figure 6.5: MATLAB neural network visualization tool. The number of iterations along with the performance can all be accessed from the interactive graphical tool. The performance, error histogram and confusion buttons produce Figs. 6.7-6.9 respectively.

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

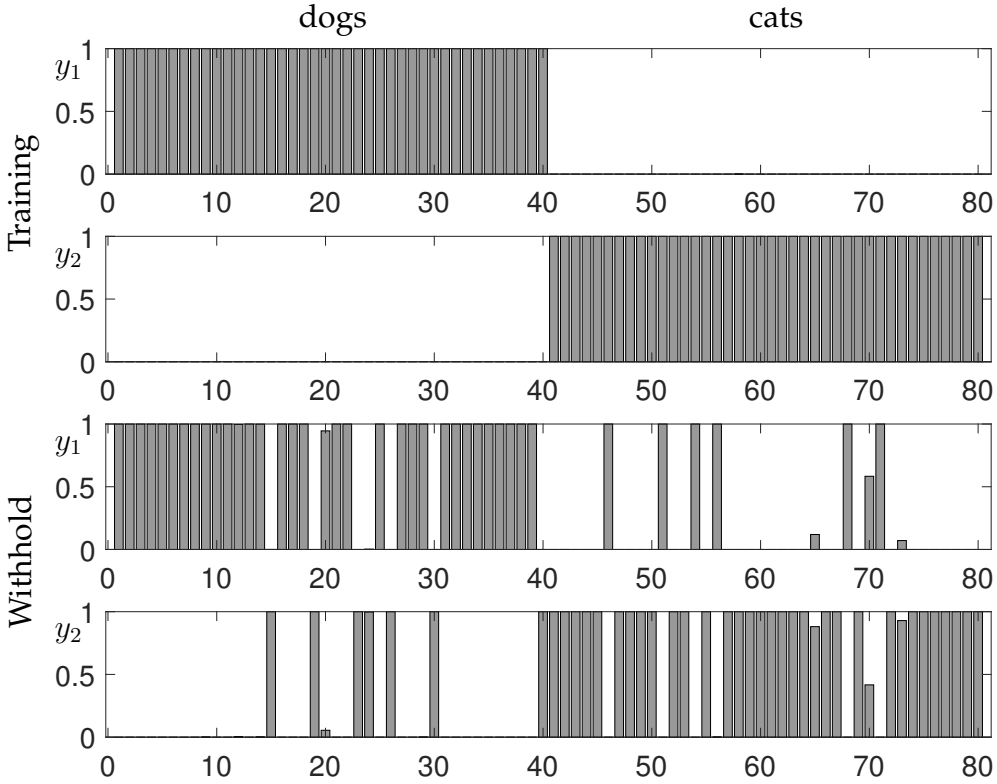

Figure 6.6: Comparison of the output vectors $\mathbf{y} = [y_1\ y_2]^T$ which are ideally (6.12) for the dogs and cats considered here. The NN training stage produces a cross-validated classifier that achieves 100% accuracy in classifying the training data (top two panels for 40 dogs and 40 cats). When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

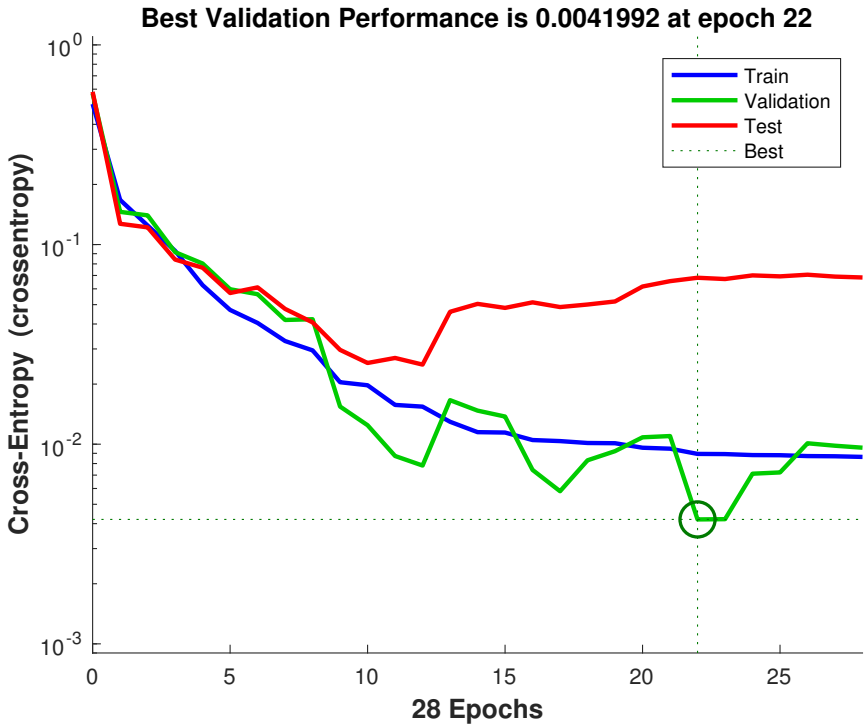

Figure 6.7: Summary of training of the NN over a number of epochs. The NN architecture automatically separates the data into training, validation and test sets. The training continues (with a maximum of 1000 epochs) until the validation error curve hits a minimum. The training then stops and the trained 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,

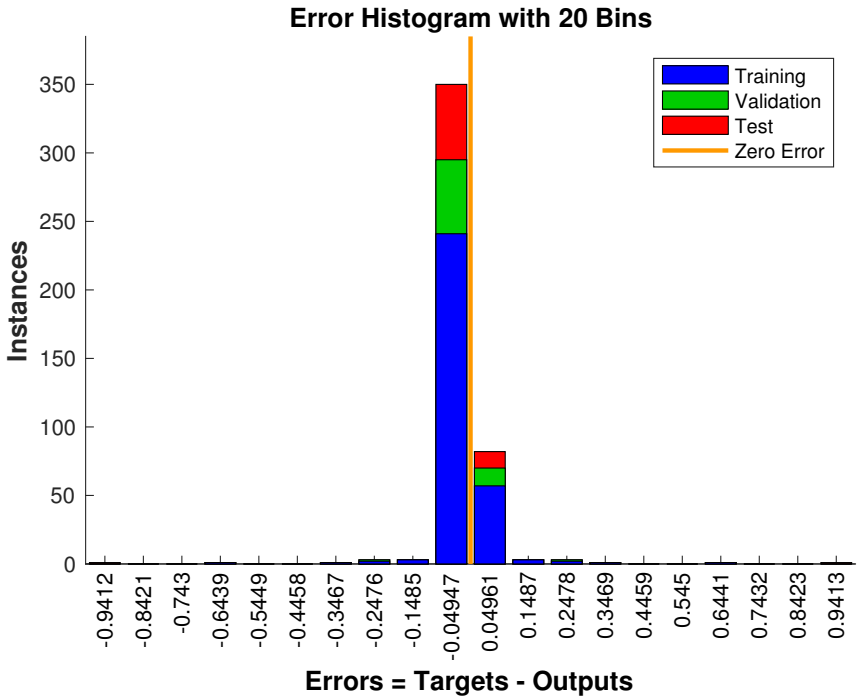

Figure 6.8: Summary of the error performance of the NN architecture for training, validation and test sets. This figure is accessed with the **errorhistogram** button on the NN interactive tool of Fig. 6.6.

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

$$y = g(z, b) = g(f(x, a), b). \tag{6.13}$$

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

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

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

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

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

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

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

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 $\mathbf{y}$, whose ideal ground-truth output is $\mathbf{y}_0$. The derivatives with respect to each network weight is then computed using backprop 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

$$f(\xi, \alpha) = g(\xi, \alpha) = \alpha\xi. \tag{6.17}$$

In this case we have in Fig. 6.10

$$z = ax \tag{6.18a}$$

$$y = bz. \tag{6.18b}$$

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

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

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

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).

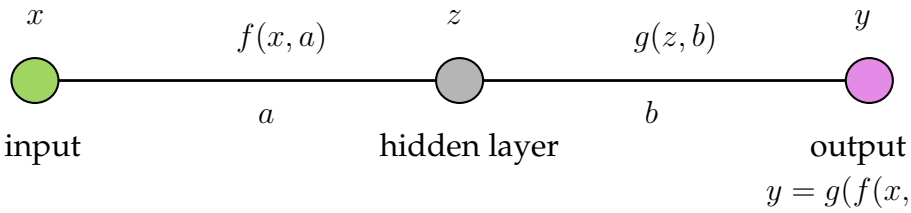

Figure 6.10: Illustration of the backpropagation algorithm on a one-node, one hidden layer network. The compositional nature of the network gives the input-output relationship $y = g(z, b) = g(f(x, a), b)$. By minimizing the error between the output y and its desired output y_0, the composition along 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

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

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

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

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

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

where this equation holds for the jth component of the vector $\mathbf{w}$. 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

$$f(x) = f(x, \boldsymbol{\beta}) \tag{6.23}$$

where $\boldsymbol{\beta}$ are fitting coefficients used to minimize the error. In NNs, the parameters $\boldsymbol{\beta}$ are the network weights, thus we can rewrite this in the form

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

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

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

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

ith row and jth column of the kth matrix ($k = 1, 2, \cdots M$). 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

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

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

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

and

$$E_k(\mathbf{A}_1, \mathbf{A}_2, \cdots, \mathbf{A}_M) = \left(f_k(\mathbf{x}_k, \mathbf{A}_1, \mathbf{A}_2, \cdots, \mathbf{A}_M) - \mathbf{y}_k\right)^2 \tag{6.28}$$

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

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

where $\mathbf{w}_j$ is the vector of all the network weights from $\mathbf{A}_j$ ($j = 1, 2, \cdots, M$) at the jth iteration, and the gradient is computed using only the kth 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

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

where $K \in [k_1, k_2, \cdots k_p]$ 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

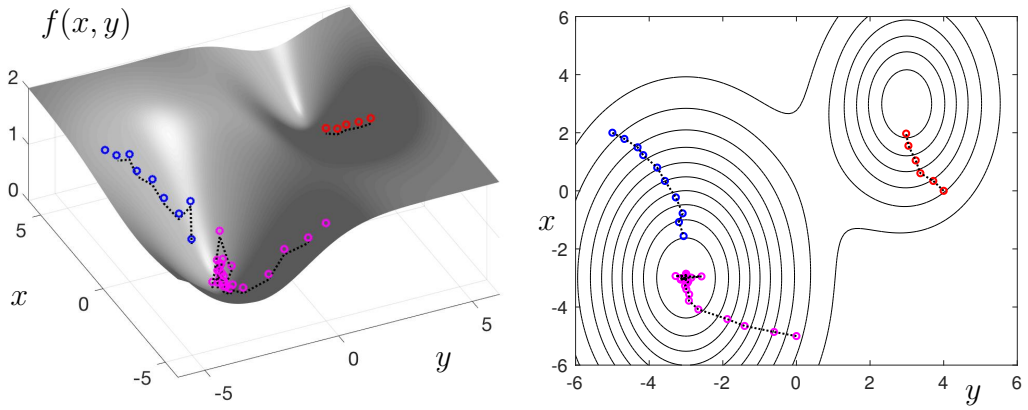

Figure 6.11: Stochastic gradient descent applied to the function featured in Fig. 4.3(b). The convergence can be compared to a full gradient descent algorithm as shown in Fig. 4.6. Each step of the stochastic (batch) gradient descent selects 100 data points for approximating the gradient, instead of 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

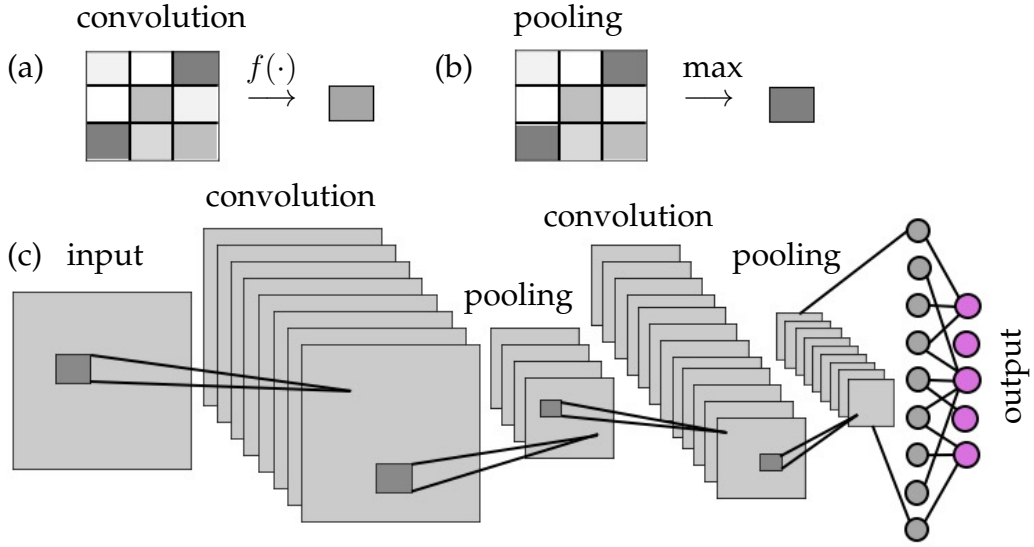

Figure 6.12: Prototypical DCNN architecture which includes commonly used convolutional and pooling layers. The dark gray boxes show the convolutional sampling from layer to layer. Note that for each layer, many functional transformations can be used to produce a variety of feature spaces. The network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

learning by Goodfellow et al. [216] provides a detailed an 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 signifiant 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 2x2 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

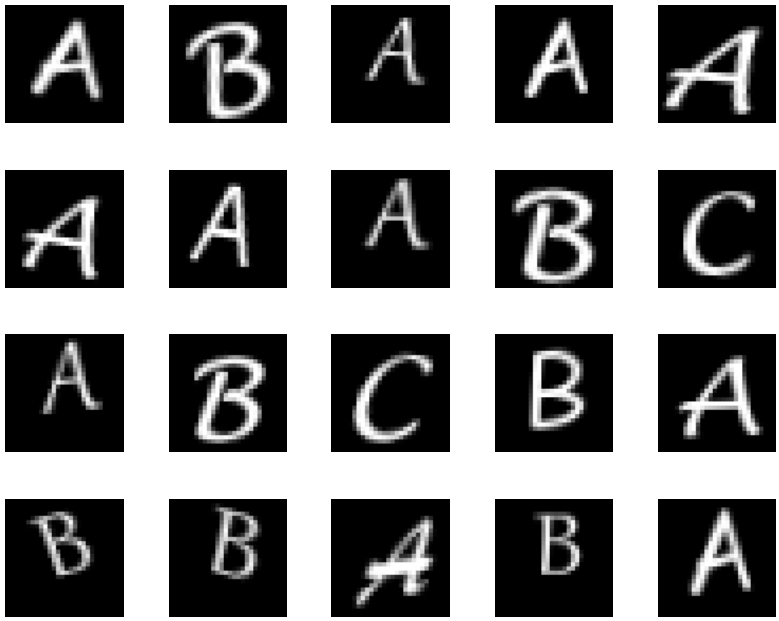

Figure 6.13: Representative images of the alphabet characters A, B, and C. There are a total of 1500 28×28 grayscale images (XTrain) of the letters that are labeled (TTrain).

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]

$$\dot{x} = \sigma(y - x) \tag{6.31a}$$
$$\dot{y} = x(\rho - z) - y \tag{6.31b}$$
$$\dot{z} = xy - \beta z, \tag{6.31c}$$

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 to 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 build 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-

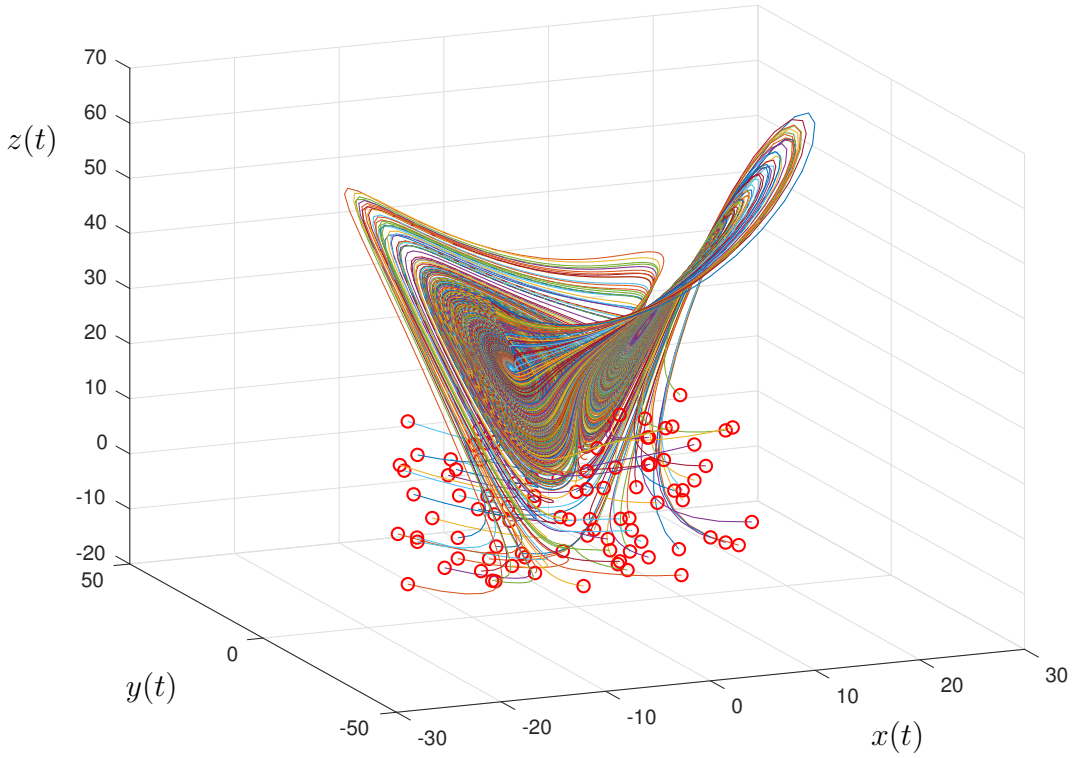

Figure 6.14: Evolution of the Lorenz dynamical equations for one hundred randomly chosen initial conditions (red circles). For the parameters $\sigma = 10, \rho = 28,$ and $\beta = 8/3$, all trajectories collapse to an attractor. These trajectories, generated from a diverse set of initial data, are used 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 for 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.

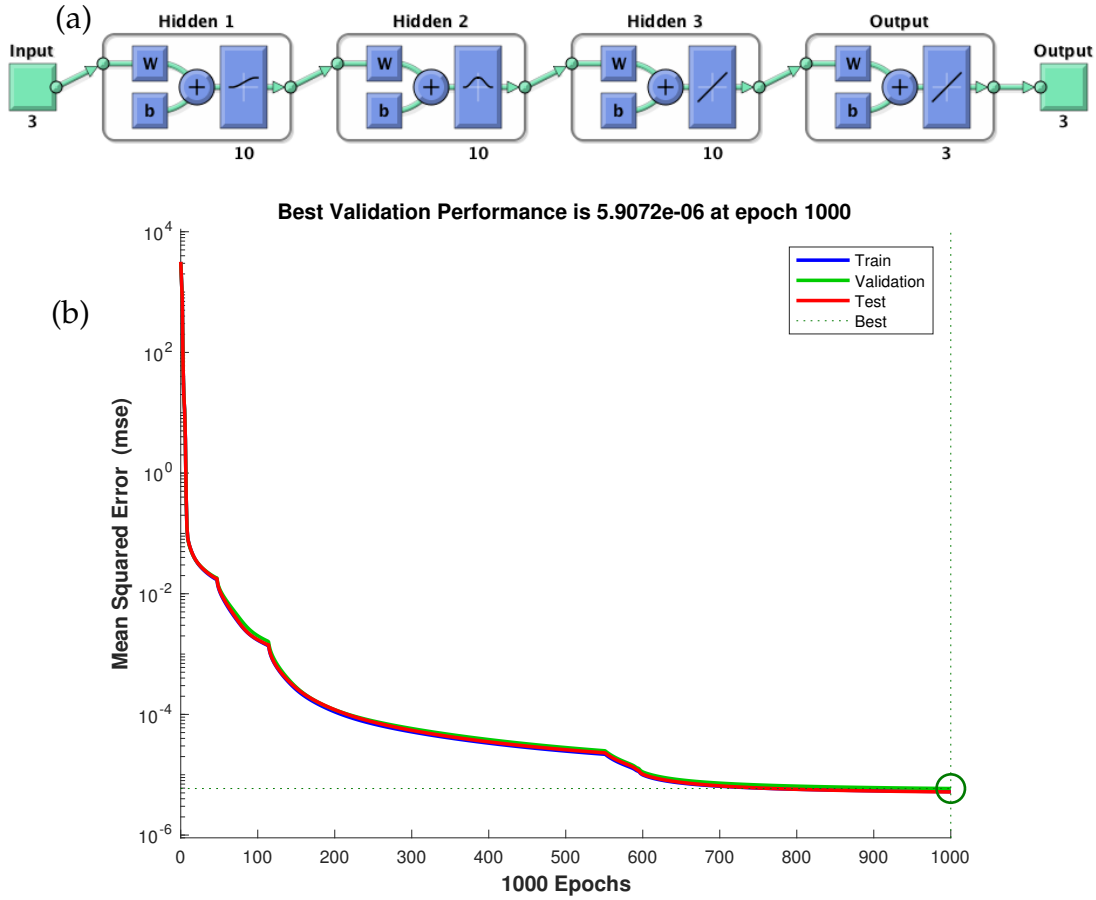

Figure 6.15: (a) Network architecture used to train the NN on the trajectory data of Fig. 6.14. A three-layer network is constructed with ten nodes in each layer and a different activation unit for each layer. (b) Performance summary of the NN optimization algorithm. Over 1000 epochs of training, accuracies 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 $\mathbf{x}_k$ to $\mathbf{x}_{k+1}$. The quality

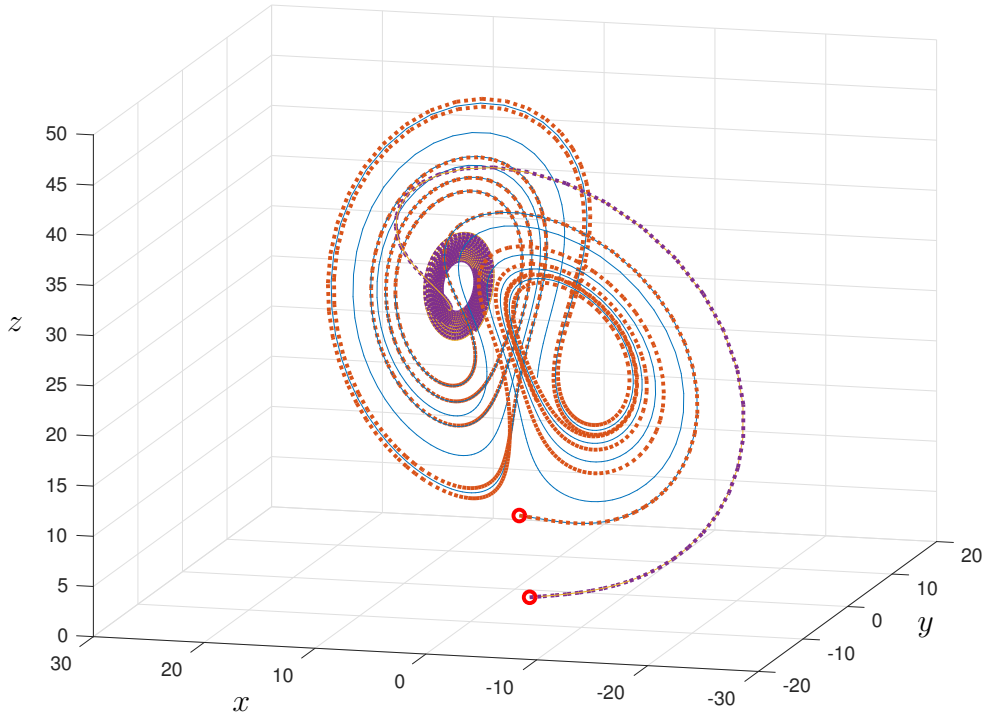

Figure 6.16: Comparison of the time evolution of the Lorenz system (solid line) with the NN prediction (dotted line) for two randomly chosen initial conditions (red dots). The NN prediction stays close to the dynamical trajectory of the Lorenz model. A more detailed comparison is given in Fig. 6.17.

of the approximation is more clearly seen in Fig. 6.17 where the time evolution of the individual components of $\mathbf{x}$ 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.

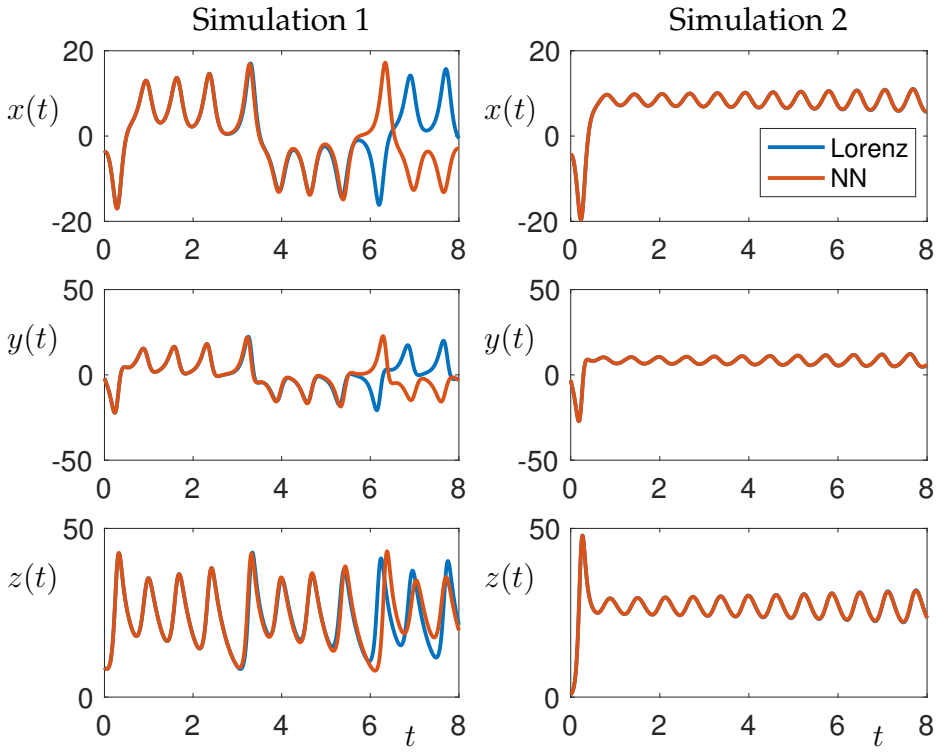

Figure 6.17: Comparison of the time evolution of the Lorenz system for two randomly chosen initial conditions (Also shown in Fig. 6.16). The left column shows that the evolution of the Lorenz differential equations and the NN mapping gives identical results until $t \approx 5.5$, at which point they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

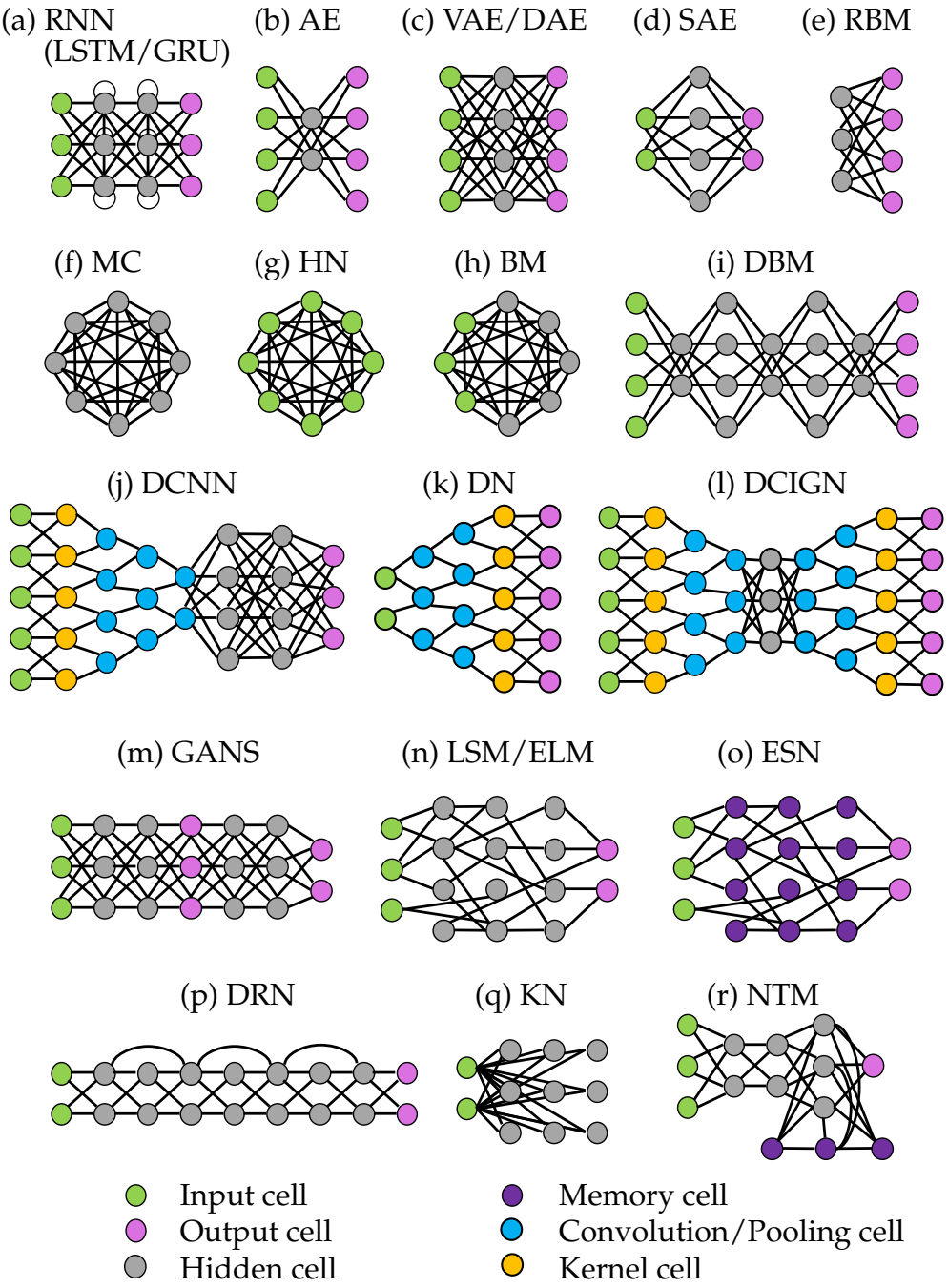

Figure 6.18: Neural network architectures commonly considered in the literature. The NNs are comprised of input nodes, output nodes, and hidden nodes. Additionally, the nodes can have memory, perform convolution and/or pooling, and perform a kernel transformation. Each network, and their acronym is explained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