Chapter 13

Neural Nets and Deep Learning

In Sections 12.2 and 12.3 we discussed the design of single "neurons" (perceptrons). These take a collection of inputs and, based on weights associated with those inputs, compute a number that, compared with a threshold, determines whether to output "yes" or "no." These methods allow us to separate inputs into two classes, as long as the classes are linearly separable. However, most problems of interest and importance are not linearly separable. In this chapter, we shall consider the design of *neural nets*, which are collections of perceptrons, or *nodes*, where the outputs of one rank (or *layer* of nodes becomes the inputs to nodes at the next layer. The last layer of nodes produces the outputs of the entire neural net. The training of neural nets with many layers requires enormous numbers of training examples, but has proven to be an extremely powerful technique, referred to as *deep learning*, when it can be used.

We also consider several specialized forms of neural nets that have proved useful for special kinds of data. These forms are characterized by requiring that certain sets of nodes in the network share the same weights. Since learning all the weights on all the inputs to all the nodes of the network is in general a hard and time-consuming task, these special forms of network greatly simplify the process of training the network to recognize the desired class or classes of inputs. We shall study convolutional neural networks (CNN's), which are specially designed to recognize classes of images. We shall also study recurrent neural networks (RNN's) and long short-term memory networks (LSTM's), which are designed to recognize classes of sequences, such as sentences (sequences of words).

13.1 Introduction to Neural Nets

We begin the discussion of neural nets with an extended example. After that, we introduce the general plan of a neural net and some important terminology.

Example 13.1 : The problem we discuss is to learn the concept that "good" bit-vectors are those that have two consecutive 1's. Since we want to deal with only tiny example instances, we shall assume bit-vectors have length four. Our training examples will thus have the form $([x_1, x_2, x_3, x_4], y)$, where each of the x_i's are bits, 0 or 1. There are 16 possible training examples, and we shall assume we are given some subset of these as our training set. Notice that eight of the possible bit vectors are good – they do have consecutive 1's, and there are also eight "bad" examples. For instance, 0111 and 1100 are good; 1001 and 0100 are bad.[1]

To start, let us look at a neural net that solves this simple problem exactly. How we might design this net from training examples is the true subject for discussion, but this net will serve as an example of what we would like to achieve. The net is shown in Fig. 13.1.

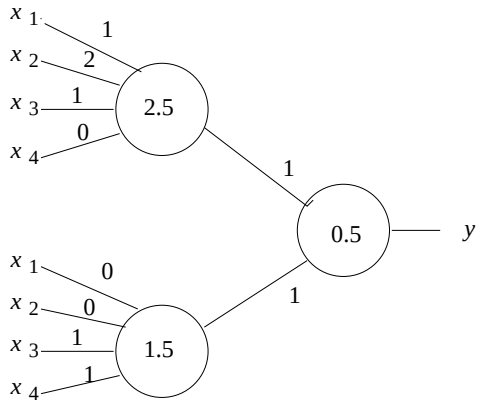


Figure 13.1: A neural net that tells whether a bit-vector has consecutive 1's

The net has two layers, the first consisting of two nodes, and the second with a single node that produces the output y. Each node is a perceptron, exactly as was described in Section 12.2. In the first layer, the first node is characterized by weight vector $[w_1, w_2, w_3, w_4] = [1, 2, 1, 0]$ and threshold 2.5. Since each input x_i is either 0 or 1, we note that the only way to reach a sum $\sum_{i=1}^{4} x_i w_i$ as high as 2.5 is if $x_2 = 1$ and at least one of x_1 and x_3 is also 1. The output of this node is 1 if and only if the input is one of 1100, 1101, 1110, 1111, 0110, or 0111. That is, it recognizes those bit-vectors that either begin with two 1's or have two 1's in the middle. The only good inputs it does not

[1]We shall show bit vectors as bit strings in what follows, so we can avoid the commas between components, each of which is 0 or 1.

recognize are those that end with 11 but do not have 11 elsewhere. these are 0011 and 1011.

Fortunately, the second node in the first layer, with weights $[0, 0, 1, 1]$ and threshold 1.5 gives output 1 whenever $x_3 = x_4 = 1$, and not otherwise. This node thus recognizes the inputs 0011 and 1011, as well as some other good inputs that are also recognized by the first node.

Now, let us turn to the second layer, with a single node; that node has weights $[1, 1]$ and threshold 0.5. It thus behaves as an "OR-gate." It gives output $y = 1$ whenever either or both of the nodes in the first layer have output 1, but gives output $y = 0$ if both of the first-layer nodes give output 0. Thus, the neural net of Fig. 13.1 gives output 1 for all the good inputs but none of the bad inputs. □

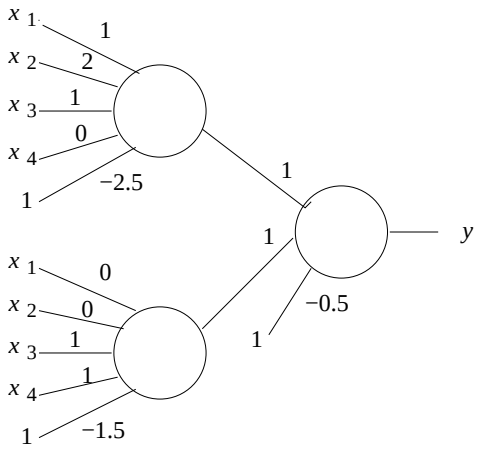


Figure 13.2: Making the threshold 0 for all nodes

It is useful in many situations to assume that nodes have a threshold of 0. Recall from Section 12.2.4 that we can always convert a perceptron with a nonzero threshold t to one with a 0 threshold if we add an additional input. That input always has value 1 and a weight equal to $-t$. For example, we can convert the net of Fig. 13.1 to that in Fig. 13.2.

13.1.1 Neural Nets, in General

Example 13.1 and its net of Fig. 13.1 is much simpler than anything that would be a useful application of neural nets. The general case is suggested by Fig. 13.3. The first, or *input layer*, is the input, which is presumed to be a vector of some length n. Each component of the vector $[x_1, x_2, \ldots, x_n]$ is an input to the net. There are one or more *hidden layers* and finally, at the end, an *output layer*, which gives the result of the net. Each of the layers can have a different number of nodes, and in fact, choosing the right number of nodes at each layer is an

important part of the design process for neural nets. Especially, note that the output layer can have many nodes. For instance, the neural net could classify inputs into many different classes, with one output node for each class.

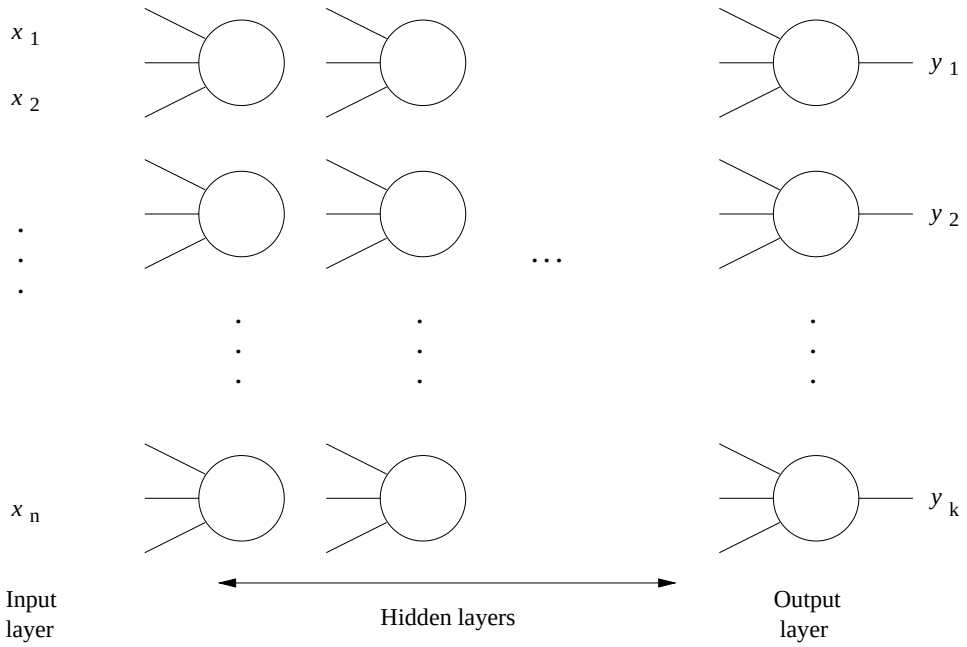


Figure 13.3: The general case of a neural network

Each layer, except for the input layer, consists of one or more nodes, which we arrange in the column that represents that layer. We can think of each node as a perceptron. The inputs to a node are outputs of some or all of the nodes in the previous layer. So that we can assume the threshold for each node is zero, we can also allow a node to have an input that is a constant, typically 1, as we suggested in Fig. 13.2. Associated with each input to each node is a *weight*. The output of the node depends on $\sum x_i w_i$, where the sum is over all the inputs x_i, and w_i is the weight of that input. Sometimes, the output is either 0 or 1; the output is 1 if that sum is positive and 0 otherwise. However, as we shall see in Section 13.2, it is often convenient, when trying to learn the weights for a neural net that solves some problem, to have outputs that are almost always close to 0 or 1, but may be slightly different. The reason, intuitively, is that it is then possible for the output of a node to be a continuous function of its inputs. We can then use gradient descent to converge to the ideal values of all the weights in the net.

13.1.2 Interconnections Among Nodes

Neural nets can differ in how the nodes at one layer are connected to nodes at the layer to its right. The most general case is when each node receives as inputs the outputs of every node of the previous layer. A layer that receives all outputs from the previous layer is said to be *fully connected*. Some other options for choosing interconnections are:

1. *Random*. For some m, we pick for each node m nodes from the previous layer and make those, and only those, be inputs to this node.

2. *Pooled*. Partition the nodes of one layer into some number of clusters. In the next layer, which is called a *pooled layer*, there is one node for each cluster, and this node has all and only the member of its cluster as inputs.

3. *Convolutional*. This approach to interconnection, which we discuss in more detail in the next section and Section 13.4, views the nodes of each layer as arranged in a grid, typically two-dimensional. In a convolutional layer, a node corresponding to coordinates (i, j) receive as inputs the nodes of the previous layer that have coordinates in some small region around (i, j). For example, the node (i, j) at one convolutional layer may have as inputs those nodes from the previous layer that correspond to coordinates (p, q), where $i \leq p \leq i + 2$ and $j \leq q \leq j + 2$ (i.e., the square of side 3 whose lower-left corner is the point (i, j).

13.1.3 Convolutional Neural Networks

A *convolutional neural network*, or *CNN*, contains one or more convolutional layers. There can also be nonconvolutional layers, such as fully connected layers and pooled layers. However, there is an important additional constraint: the weights on the inputs must be the same for all nodes of a single convolutional layer. More precisely, suppose that each node (i, j) in a convolutional layer recieves $(i + u, j + v)$ as one of its inputs, where u and v are small constants. Then there is a weight w associated with u and v (but not with i and j). For any i and j, the weight on the input to the node for (i, j) coming from the output of the node $i + u, j + v)$ from the previous layer must be w.

This restriction makes training a CNN much more efficient than training a general neural net. The reason is that there are many fewer parameters at each layer, and therefore, many fewer training examples can be used, than if each node or each layer has its own weights for the training process to discover.

CNN's have been found extremely useful for tasks such as image recognition. In fact, the CNN draws inspiration from the way the human eye processes images. The neurons of the eye are arranged in layers, similarly to the layers of a neural net. The first layer takes inputs that are essentially pixels of the image, each pixel the result of a sensor in the retina. The nodes of the first layer recognize very simple features, such as edges between light and dark. Notice that a small square of pixels, say 3-by-3, might exhibit an edge at a

particular angle, e.g., if the upper left corner is light and the other eight pixels dark. Moreover, the algorithm for recognizing an edge of a certain type is the same, regardless of where in the field of vision this little square appears. That observation justifies the CNN constraint that all the nodes of a layer use the same weights. In the eye, additional layers combine results from the previous layers to recognize more and more complex structures: long boundaries, regions of similar color, and finally faces and all the familiar objects that we see daily.

We shall have more to say about convolutional neural networks in Section 13.4. Moreover, CNN's are only one example of a kind of neural network where certain families of nodes are constrained to have the same weights. For example, in Section 13.5, we shall consider recurrent neural networks and long short-term memory networks, which are specially adapted to recognizing properties of sequences, such as sentences (sequences of words).

13.1.4 Design Issues for Neural Nets

Building a neural net to solve a given problem is partially art and partially science. Before we can begin to train a net by finding the weights on the inputs that serve our goals best, we need to make a number of design decisions. These include answering the following questions:

1. How many hidden layers should we use?
2. How many nodes will there be in each of the hidden layers?
3. In what manner will we interconnect the outputs of one layer to the inputs of the next layer?

Further, in later sections we shall see that there are other decisions that need to be made when we train the neural net. These include:

4. What cost function should we minimize to express what weights are best?
5. How should we compute the outputs of each gate as a function of the inputs? We have suggested that the normal way to compute output is to take a weighted sum of inputs and compare it to 0. But there are other computations that serve better in common circumstances.
6. What algorithm do we use to exploit the training examples in order to optimize the weights?

13.1.5 Exercises for Section 13.1

!! **Exercise 13.1.1:** Prove that the problem of Example 13.1 cannot be solved by a perceptron; i.e., the good and bad points are not linearly separable.

! **Exercise 13.1.2:** Consider the general problem of identifying bit-vectors of length n having two consecutive 1's. Assume a single hidden layer with some number of gates. What is the smallest number of gates you can have in the hidden layer if (a) $n = 5$ (b) $n = 6$?

! **Exercise 13.1.3:** Design a neural net that functions as an *exclusive-or* gate, that is, it takes two inputs and gives output 1 if exactly one of the inputs is 1 gives output 0 otherwise. *Hint*: remember that both weights and thresholds can be negative.

! **Exercise 13.1.4:** Prove that there is no single preceptron that behaves like an exclusive-or gate.

! **Exercise 13.1.5:** Design a neural net to compute the exclusive-or of three inputs; that is, output 1 if an odd number of the three inputs is 1 and output 0 if an even number of the inputs are 1.

13.2 Dense Feedforward Networks

In the previous section, we simply exhibited a neural net that worked for the "consecutive 1's" problem. However, the true value of neural nets comes from our ability to design them given training data. To design a net, there are many choices that must be made, such as the number of layers and the number of nodes for each layer, as was discussed in Section 13.1.4. These choices can be more art than science. The computational part of training, which is more science than art, is primarily the choice of weights for the inputs to each node.

The techniques for selecting weights usually involve convergence using gradient descent. But gradient descent requires a cost function, which must be a continuous function of the weights. The nets discussed in Section 13.1.4, on the other hand, use perceptrons whose output is either 0 or 1, so the outputs are not normally continuous functions of the inputs. In this section, we shall discuss the various ways one can modify the behavior of the nodes in a net so the outputs become continuous functions of the inputs, and therefore a reasonable cost function applied to the outputs will also be continuous.

13.2.1 Linear Algebra Notation

We can succinctly describe the neural network we used for the consecutive 1's problem using linear algebra notation. The input nodes form a vector[2] $\mathbf{x} = [x_1, x_2, x_3, x_4]$, while the hidden nodes form a vector $\mathbf{h} = [h_1, h_2]$. The 4 edges connecting the input to hidden node 1 form a weight vector $\mathbf{w}_1 = [w_{11}, w_{12}, w_{13}, w_{14}]$, and similarly we have weight vector $\mathbf{w}_2$ for hidden node 2.

[2]We assume all vectors are column vectors by default. However, it is often more convenient to write row vectors, and we shall do so in the text. But in formulas, we shall use the transpose operator when we actually want to use the vector as a row rather than a column.

Why Use Linear Algebra?

Notational brevity is one reason to use linear algebra notation for neural networks. Another is performance. It turns out that graphics processing units (GPU's) have circuitry that allows for highly parallelized linear-algebra operations. Mutiplying a matrix and a vector using a single linear algebra operator is much faster than coding the same operator using nested loops. Modern deep-learning frameworks (e.g., TensorFlow, PyTorch, Caffe) harness the power of GPU's to dramatically speed up neural network computations.

The threshold inputs to the hidden layer nodes (i.e., the negatives of the thresholds) form a 2-vector $\mathbf{b} = [b_1, b_2]$, often called the *bias vector*. The perceptron applies the nonlinear *step function* to produce its output, defined as:

$$\text{step}(z) = \begin{cases} 1 & \text{when } z > 0 \\ 0 & \text{otherwise} \end{cases}$$

Each hidden node h_i can now be described using the expression:

$$h_i = \text{step}(\mathbf{w_i}^\mathsf{T}\mathbf{x} + b_i) \text{ for } i = 1, 2$$

We could organize the weight vectors $\boldsymbol{w_1}$ and $\boldsymbol{w_2}$ into a 2×4 weight matrix W, where the ith row of W is $\boldsymbol{w_i}^\mathsf{T}$. The hidden nodes can thus be described using the expression:

$$\mathbf{h} = \text{step}(W\mathbf{x} + \mathbf{b})$$

In the case of a vector input, the step function just operates element-wise on the vector to produce an output vector of the same length. We can use a similar arrangement to describe the transformation that produces the final output from the hidden layer. In this case, the final output is a scalar y, so instead of a weight matrix W we need only a weight vector $\boldsymbol{u} = [u_1, u_2]$ and a single bias c. We thus have:

$$y = \text{step}(\boldsymbol{u}^\mathsf{T}\mathbf{h} + c)$$

Linear-algebra notation works just as well when we have larger inputs and many more nodes in one hidden layer. We just need to scale the weight matrix and bias vector appropriately. That is, the matrix W has one row for each node in the layer and one column for each output from the previous layer (or for each input, if this is the first layer); the bias vector has one component for each node. It is also easy to handle the case where there is more than one node in the output layer. For example, in a multiclass classification problem, we might have an output node y_i corresponding to target class i. For a given input, the outputs specify the probability that the input belongs to the corresponding

class. This arrangement results in an output vector $\mathbf{y} = [y_1, y_2, \ldots, y_n]$, where n is number of classes. The simple network from the prior section had a boolean output, corresponding to two output classes (true and false), so we could have modeled the output equally well as a 2-vector. In the case of a vector output, we connect the hidden and output layers by a weight matrix of appropriate dimensions in place of the weight vector we used for the example.

The perceptrons in our example used a nonlinear step function. More generally, we can use any other nonlinear function, called the *activation function*, following the linear transformation. We describe commonly used activation functions starting in Section 13.2.2.

Our simple example used a single hidden layer of nodes between the input and output layers. In general, we can have many hidden layers, as was sugested in Fig. 13.3. Each hidden layer introduces an additional matrix of weights and vector of biases, as well as its own activation function. This kind of network is called a *feedforward network*, since all edges are oriented "forward," from input to output, without cycles.

Suppose there are ℓ hidden layers and an additional output layer, numbered $\ell + 1$. Let the weight matrix for the ith layer be W_i and let the bias vector for that layer be $\mathbf{b}_i$. The weights $W_1, W_2, \ldots, W_{l+1}$ and biases $\mathbf{b}_1, \mathbf{b}_2, \ldots, \mathbf{b}_{l+1}$ constitute the *parameters* of the model. Our objective is to learn the best values for these parameters to achieve the task at hand. We will soon describe how to go about learning the model parameters.

13.2.2 Activation Functions

A node (perceptron) in a neural net is designed to give a 0 or 1 (yes or no) output. Often, we want to modify that output is one of several ways, so we apply an *activation function* to the output of a node. In some cases, the activation function takes all the outputs of a layer and modifies them as a group. the reason we need an activation function is as follows. The approach we shall use to learn good parameter values for the network is gradient descent. Thus, we need activation functions that "play well" with gradient descent. In particular, we look for activation functions with the following properties:

1. The function is continuous and differentiable everywhere (or almost everywhere).
2. The derivative of the function does not *saturate* (i.e., become very small, tending towards zero) over its expected input range. Very small derivatives tend to stall out the learning process.
3. The derivative does not *explode* (i.e., become very large, tending towards infinity), since this would lead to issues of numerical instability.

The step function does not satisfy conditions (2) and (3). Its derivative explodes at 0 and is 0 everywhere else. Thus the step function does not play well with gradient descent and is not a good choice for a deep neural network.

13.2.3 The Sigmoid

Given that we cannot use the step function, we look for alternatives in the class of *sigmoid functions* – so called because of the S-shaped curve that these functions exhibit. The most commonly used sigmoid function is the *logistic sigmoid*:

$$\sigma(x) = \frac{1}{1+e^{-x}} = \frac{e^x}{1+e^x}$$

Notice that the sigmoid has the value $1/2$ at $x = 0$. For large x, the sigmoid approaches 1, and for large, negtive x, the sigmoid approaches 0.

The logistic sigmoid, like all the functions we shall discuss, are applied to vectors elementwise, so if $\mathbf{x} = [x_1, x_2, \ldots, x_n]$ then

$$\sigma(\mathbf{x}) = [\sigma(x_1), \sigma(x_2), \ldots, \sigma(x_n)]$$

The logistic sigmoid has several advantages over the step function as a way to define the output of a perceptron. The logistic sigmoid is continuous and differentiable, so it enables us to use gradient descent to discover the best weights. Since its value is in the range $[0, 1]$, it is possible to interpret the outputs of the network as a probability. However, the logistic sigmoid saturates very quickly as we move away from the "critical region" around 0. So the derivative goes towards zero and gradient-based learning can stall out. That is, weights almost stop changing, once they get away from 0.

In Section 13.3.3, when we describe the backpropagation algorithm, we shall see that we need the derivatives of activation functions and loss functions. As an exercise, you can verify that if $y = \sigma(x)$, then

$$\frac{dy}{dx} = y(1-y)$$

13.2.4 The Hyperbolic Tangent

Closely related to sigmoid is the *hyperbolic tangent* function, defined by:

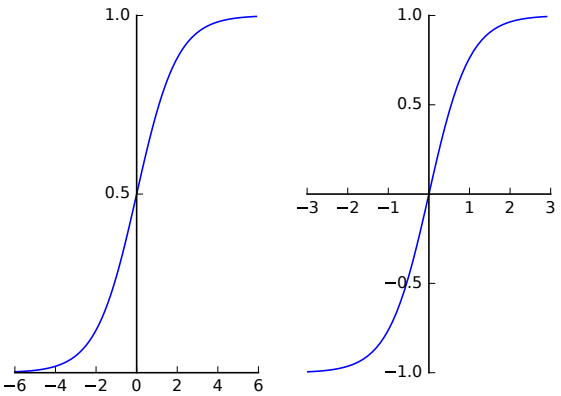
$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

Simple algebraic manipulation yields:

$$\tanh(x) = 2\sigma(2x) - 1$$

So the hyperbolic tangent is just a scaled and shifted version of the sigmoid. It has two desirable properties that make it attractive in some situations: its output is in the range $[-1, 1]$ and is symmetric around 0. It also shares the good properties and the saturation problem of the sigmoid. You may show that if $y = \tanh(x)$ then

$$\frac{dy}{dx} = 1 - y^2$$



(a) (b)

Figure 13.4: The logistic sigmoid (a) and hyperbolic tangent (b) functions

Figure 13.4 shows the logistic sigmoid and hyperbolic tangent functions. Note the difference in scale along the x-axis between the two charts. It is easy to see that the functions are identical after shifting and scaling.

13.2.5 Softmax

The *softmax* function differs from sigmoid functions in that it does not operate element-wise on a vector. Rather, the softmax function applies to an entire vector. If $\mathbf{x} = [x_1, x_2, \ldots, x_n]$, then its softmax $\mu(\mathbf{x}) = [\mu(x_1), \mu(x_2), \ldots, \mu(x_n)]$ where

$$\mu(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$$

Softmax pushes the largest component of the vector towards 1 while pushing all the other components towards zero. Also, all the outputs sum to 1, regardless of the sum of the components of the input vector. Thus, the output of the softmax function can be intepreted as a probability distribution.

A common application is to use softmax in the output layer for a classification problem. The output vector has a component corresponding to each target class, and the softmax output is interpreted as the probability of the input belonging to the corresponding class.

Softmax has the same saturation problem as the sigmoid function, since one component gets larger than all the others. There is a simple workaround to this problem, however, when softmax is used at the output layer. In this case, it is usual to pick *cross entropy* as the loss function, which undoes the exponentiation in the definition of softmax and avoids saturation. Cross entropy is explained in

Accuracy of Softmax Calculation

The denominator of the softmax function involves computing a sum of the form $\sum_j e^{x_j}$. When the x_j's take a wide range of values, their exponents e^{x_j} take on an even wider range of values – some tiny and some very large. Adding very large and very small floating point numbers leads to numerical inaccuracy issues in fixed-width floating point representations (such as 32-bit or 64-bit). Fortunately, there is a trick to avoid this problem. We observe that

$$\mu(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}} = \frac{e^{x_i - c}}{\sum_j e^{x_j - c}}$$

for any constant c. We pick $c = \max_j x_j$, so that $x_j - c \leq 0$ for all j. This ensures that $e^{x_j - c}$ is always between 0 and 1, and leads to a more accurate calculation. Most deep learning frameworks will take care to compute softmax in this manner.

Section 13.2.9. We address the problem of differentiating the softmax function in Section 13.3.3.

13.2.6 Recified Linear Unit

The *rectified linear unit*, or ReLU, is defined as:

$$f(x) = \max(0, x) = \begin{cases} x, & \text{for } x \geq 0 \\ 0, & \text{for } x < 0 \end{cases}$$

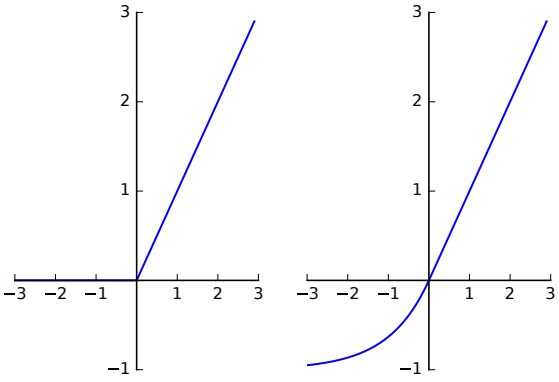
The name of this function derives from the analogy to *half-wave rectification* in electrical engineering. The function is not differentiable at 0 but is differentiable everywhere else, including at points arbitrarily close to 0. In practice, we "set" the derivative at 0 to be either 0 (the left derivative) or 1 (the right derivative).

In modern neural nets, a version of ReLU has replaced sigmoid as the default choice of activation function. The popularity of ReLU derives from two properties:

1. The gradient of ReLU remains constant and never saturates for positive x, speeding up training. It has been found in practice that networks that use ReLU offer a significat speedup in training compared to sigmoid activation.

2. Both the function and its derivative can be computed using elementary and efficient mathematical operations (no exponentiation).

ReLU does suffer from a problem related to the saturation of its derivative when $x < 0$. Once a node's input values become negative, it is possible that the



(a) (b)

Figure 13.5: The ReLU (a) and ELU (b), with $\alpha = 1$ functions

node's output get "stuck" at 0 through the rest of the training. This is called the *dying ReLU* problem.

The *Leaky ReLU* attempts to fix this problem by defining the activation function as follows:

$$f(x) = \begin{cases} x, & \text{for } x \geq 0 \\ \alpha x, & \text{for } x < 0 \end{cases}$$

where α is typically a small positive value such as 0.01. The *Parametric ReLU* (PReLU) makes α a parameter to be optimized as part of the learning process.

An improvement on both the original and leaky ReLU functions is *Exponential Linear Unit*, or ELU. This function is defined as:

$$f(x) = \begin{cases} x, & \text{for } x \geq 0 \\ \alpha(e^x - 1), & \text{for } x < 0 \end{cases}$$

where $\alpha \geq 0$ is a *hyperparameter*. That is, α is held fixed during the learning process, but we can repeat the learning process with different values of α to find the best value for our problem. The node's value saturates to $-\alpha$ for large negative values of x, and a typical choice is $\alpha = 1$. ELU's drive the mean activation of nodes towards zero, which appears to speed up the learning process compared to other ReLU variants.

13.2.7 Loss Functions

A loss function quantifies the difference between a model's predictions and the output values observed in the real world (i.e., in the training set). Suppose

the observed output corresponding to input $\mathbf{x}$ is $\hat{\mathbf{y}}$ and the predicted output is $\mathbf{y}$. Then a loss fucntion $L(\mathbf{y}, \hat{\mathbf{y}})$ quantifies the prediction error for this single input. Typically, we consider the loss over a large set of observations, such as the entire training set. In that case, we usually average the losses over all training examples.

We shall consider separately two cases. In the first case, there is a single output node, and it produces a real value. In this case we study "regression loss." In the second case, there are several output nodes, each of which indicates that the input is a member of a particular class; we study this matter under "classification loss" in Section 13.2.9.

13.2.8 Regression Loss

Suppose the model has a single continuous-valued output, and $(\mathbf{x}, \hat{y})$ is a training example. For the same input $\mathbf{x}$, suppose the predicted output of the neural net is y. Then the *squared error* loss $L(y, \hat{y})$ of this prediction is:

$$L(y, \hat{y}) = (y - \hat{y})^2$$

In general, we compute the loss for a set of predictions. Suppose the observed (i.e., training set) input-output pairs are $T = \{(\mathbf{x}_1, \hat{y_1}), (\mathbf{x}_2, \hat{y_2}), \ldots, (\mathbf{x}_n, \hat{y_n})\}$, while the corresponding input-output pairs predicted by the model are $P = \{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \ldots, (\mathbf{x}_n, y_n)\}$. The *mean squared error* (MSE) for the set is:

$$L(P, T) = \frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y_i})^2$$

Note that the mean squared error is just square of the RMSE. It is convenient to omit the square root to simplify the derivative of the function, which we shall use during training. In any case, when we minimize MSE we also automatically minimize RMSE.

One problem with MSE is that it is very sensitive to outliers due the squared term. A few outliers can contribute very highly to the loss and swamp out the effect of other points, making the training process susceptible to wild swings. One way to deal with this issue is to use the *Huber Loss*. Suppose $z = y - \hat{y}$, and δ is a constant. The Huber Loss L_δ is given by:

$$L_\delta(z) = \begin{cases} z^2 & \text{if } |z| \leq \delta \\ 2\delta(|z| - \frac{1}{2}\delta) & \text{otherwise} \end{cases}$$

Figure 13.6 contrasts the squared error and Huber loss functions.

In the case where we have a vector $\mathbf{y}$ of outputs rather than a single output, we use $\|\mathbf{y} - \hat{\mathbf{y}}\|$ in place of $|y - \hat{y}|$ in the definitions of mean squared error and Huber loss.

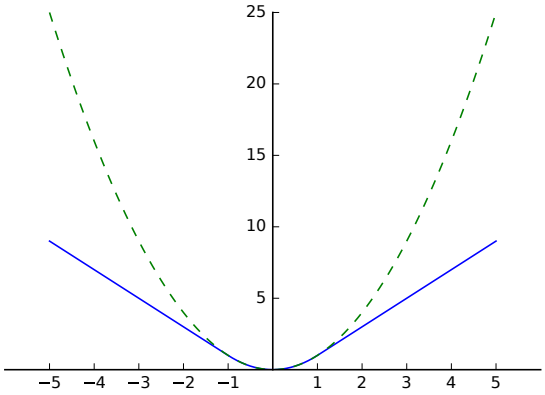


Figure 13.6: Huber Loss (solid line, $\delta = 1$) and Squared Error (dotted line) as functions of $z = y - \hat{y}$

13.2.9 Classification Loss

Consider a multiclass classification problem with target classes $C_1, C_2, \ldots, C_n$. Suppose each point in the training set is of the form $(\mathbf{x}, \mathbf{p})$ where $\mathbf{x}$ is the input and $\mathbf{p} = [p_1, p_2, \ldots, p_n]$ is the output. Here p_i gives the probability that the input $\mathbf{x}$ belongs to class C_i, with $\sum_i p_i = 1$. In many cases, we are certain that an input belongs to a particular class C_i; in this case $p_i = 1$ and $p_j = 0$ for $i \neq j$. In general, we may interpret p_i as our level of certainty that input $\mathbf{x}$ belongs to class C_i, and $\mathbf{p}$ as a *probability distribution* over the target classes.

We design our neural network to produce as output a vector

$$\mathbf{q} = [q_1, q_2, \ldots, q_n]$$

of probabilities, with $\sum_i q_i = 1$. As before, we interpret $\mathbf{q}$ as a probability distribution over the target classes, with q_i denoting the model's probability that input $\mathbf{x}$ belongs to target class C_i. In Section 13.2.5 we described a simple method to produce such a probability vector as output: use the softmax activation function in the output layer of the network.

Since both the labeled output and the model's output are probability distributions, it is natural to look for a loss function that quantifies the distance between two probability distributions. Recall the definition of entropy from Section **??**. That is, $H(\mathbf{p})$, the entropy of a discrete probability distribution $\mathbf{p}$ is:

$$H(\mathbf{p}) = -\sum_{i=1}^{n} p_i \log p_i$$

Imagine an alphabet of n symbols, and messages using these symbols. Suppose at each location in the message, the probability that symbol i appears is

p_i. Then a key result from information theory is that if we encode messages using an optimal binary code, the average number of bits per symbol needed to encode messages is $H(\mathbf{p})$.

Suppose we did not know the symbol probability distribution $\mathbf{p}$ when we design the coding scheme. Instead, we believe that symbols appear following a different probability distribution $\mathbf{q}$. We might ask what the average number of bits per symbol will be if we use this suboptimal encoding scheme. A well-known result from information theory states that the average number of bits in this case is the *cross entropy* $H(\mathbf{p}, \mathbf{q})$, defined as:

$$H(\mathbf{p}, \mathbf{q}) = -\sum_{i=1}^{n} p_i \log q_i$$

Note that $H(\mathbf{p}, \mathbf{p}) = H(\mathbf{p})$, and in general $H(\mathbf{p}, \mathbf{q}) \geq H(\mathbf{p})$. The difference between the cross entropy and the entropy is the average number of additional bits needed per symbol. It is a reasonable measure of the distance between the distributions $\mathbf{p}$ and $\mathbf{q}$, called the *Kullblack-Liebler divergence* (KL-divergence) and denoted $D(\mathbf{p}\|\mathbf{q})$:

$$D(\mathbf{p}\|\mathbf{q}) = H(\mathbf{p}, \mathbf{q}) - H(\mathbf{p}) = \sum_{i=1}^{n} p_i \log \frac{p_i}{q_i}$$

Even though KL-divergence is often regarded as a distance, it is not truly a distance measure because it is not commutative. However, it is perfectly adequate as a loss function for our purposes, since there is in fact an inherent assymmetry in the situation: $\mathbf{p}$ is the ground truth while $\mathbf{q}$ is the predicted output. Notice that minimizing the KL-divergence loss of a model is equivalent to minimizing the cross-entropy loss, since the term $H(\mathbf{p})$ depends only on the input and is independent of the model that is learned.

In practice, cross entropy is the most commonly used loss function for classification problems. Networks designed for classification often use a softmax activation function in the output layer. These choices are so common that many implementations, such as TensorFlow, offer a single function that combines softmax with cross entropy. In addition to convenience, one reason to do so is that the combined function is more stable numerically, and its derivative also takes a simple form, as we show in Section 13.3.3.

13.2.10 Exercises for Section 13.2

Exercise 13.2.1: Show that for the logistic sigmoid σ, if $y = \sigma(x)$, then

$$\frac{dy}{dx} = y(1 - y)$$

Exercise 13.2.2: Show that if $y = \tanh(x)$ then

$$\frac{dy}{dx} = 1 - y^2$$

Exercise 13.2.3: Show that $\tanh(x) = 2\sigma(2x) - 1$.

Exercise 13.2.4: Show that $\sigma(x) = 1 - \sigma(-x)$.

Exercise 13.2.5: Show that for any vector $[v_1, v_2, \ldots, v_k]$, $\sum_{i=1}^{k} \mu(v_i) = 1$.

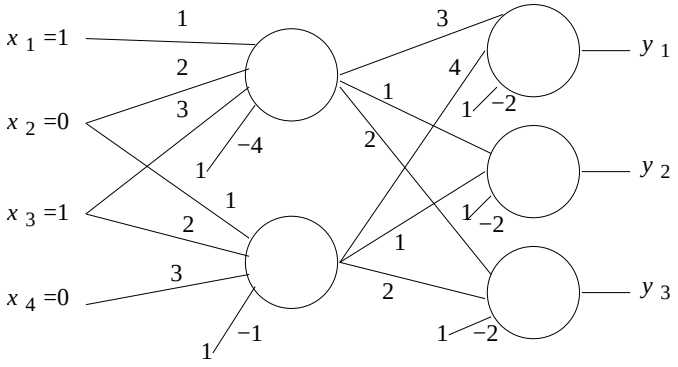


Figure 13.7: Neural net for Exercise 13.2.6

Exercise 13.2.6: In Fig. 13.7 is a neural net with paricular values shown for all the weights and inputs. Suppose that we use the sigmoid function to compute outputs of nodes at the first layer, and we use softmax to compute the outputs of the nodes in the output layer.

(a) Compute the values of the outputs for each of the five nodes.

! (b) Assuming each of the weights and each of the x_i's is a variable, express the output of the first (top) output node in terms of the weights and the x_i's,

! (c) Find the derivative of your function from part (b) with respect to the weight on the first (top) input to the first (top) node in the first layer.

13.3 Backpropagation and Gradient Descent

We now turn to the problem of training a deep network. Training a network means finding good values for the parameters (weights and thresholds) of the network. Usually, we shall have access to a training set of labeled input/output pairs. The training process tries to find parameter values that minimize the average loss on the training set. The hope is that the training set is representative of the data the model will encounter in the future, and therefore, the average loss on the training set is a good measure of the average error on all possible inputs. We must be careful, however; since deep networks have many

parameters, it is possible to find parameters that yield low training loss but nevertheless perform poorly in the real world. This phenomenon is called overfitting, a problem we have mentioned several times, starting in Section 9.4.4.

For the moment, we assume that our goal is to find parameters that minimize the expected loss on the training set. This goal is achieved by gradient descent. There is an elegant algorithm called *backpropagation* that allows us to compute these gradients efficiently. Before we describe backpropagation, we need a few preliminaries.

13.3.1 Compute Graphs

A compute graph captures the data flow of a deep network. Formally, a compute graph is a directed, acyclic graph (DAG). Each node in the compute graph has an operand and, optionally, an operator. The operand can be a scalar, a vector, or a matrix. The operator is a linear-algebra operator (such as $+$ or $\times$), an activation function (such as σ) or a loss function (such as MSE). When a node has both an operand and an operator, the operator is written above the operand.

When a node has only an operand, its output is the value associated with the operand. The output of a node with an operator is the result of applying its operator to its immediate predecessors in the graph and then assigning the result to the operand. In general, the operator can be an arbitrary expression that uses its inputs to produce an output.[3]

Example 13.2: Figure 13.8 shows the compute graph for a single-layer dense network described by $\mathbf{y} = \sigma(W\mathbf{x}+\mathbf{b})$ where $\mathbf{x}$ is the input and $\mathbf{y}$ is the output. We then compute an MSE loss against the training-set output $\hat{\mathbf{y}}$. That is, we have a single layer of n nodes. The vector $\mathbf{y}$ is of length n and represents the outputs of each of these nodes. There are k inputs, and $(\mathbf{x}, \hat{\mathbf{y}})$ represents one training example. Matrix W represents the weights on the inputs of the nodes; that is, W_{ij} is the weight for input j at the ith node. Finally, $\mathbf{b}$ represents the n biases, so its ith element is the negative of the threshold of the ith node.

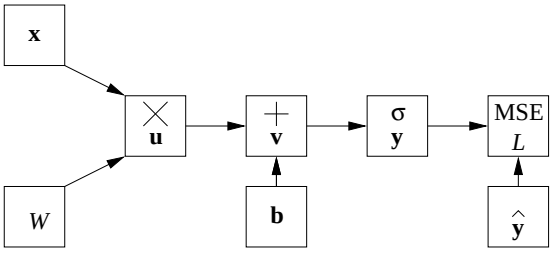


Figure 13.8: Compute graph for a single-layer dense network

For the graph in Fig. 13.8, we have:

[3]Sometimes the order of operands to the operator matters. We shall ignore that detail here and assume it is understood from the context.

$$\begin{aligned}\mathbf{u} &= W\mathbf{x}\\ \mathbf{v} &= \mathbf{u} + \mathbf{b}\\ \mathbf{y} &= \sigma(\mathbf{v})\\ L &= \text{MSE}(\mathbf{y}, \hat{\mathbf{y}})\end{aligned}$$

Each of these steps corresponds to one of the four nodes in the middle row, in order from the left. The first step corresponds to the node with operand $\mathbf{u}$ and operator $\times$. Here is an example where it must be understood that the node labeled W is the first argument. If necessary, we could label each incoming edge with a number to indicate its order, but in this case the order should be obvious, since a column vector $\mathbf{x}$ could not multiply a matrix W unless the matrix happened to have only one row. The second step corresponds to the node with operator + and operand $\mathbf{v}$. Here, the order of arguments does not matter, since + on vectors is commutative. □

13.3.2 Gradients, Jacobians, and the Chain Rule

The goal of the backpropagation algorithm is to compute the gradient of the loss function with respect to the parameters of the network. Then, we can adjust the parameters slightly in the directions that will reduce the loss, and repeat the process until we reach a selection of parameter values for which little improvement in the loss is possible. Recall the definition of the gradient: given a function $f : \mathbb{R}^N \to \mathbb{R}$ from a real-valued vector to a scalar, if $\mathbf{x} = [x_1, x_2, \ldots, x_n]$ and $y = f(\mathbf{x})$ then the gradient of y with respect to x, denoted by $\nabla_{\mathbf{x}} y$ is given by

$$\nabla_{\mathbf{x}} y = \Big[\frac{\partial y}{\partial x_1}, \frac{\partial y}{\partial x_2}, \ldots, \frac{\partial y}{\partial x_n}\Big]$$

Example 13.3: We could let function f be the loss function, e.g., the squared-error loss, which we denote L. This loss is a scalar-valued function of the output $\mathbf{y}$:

$$L(\mathbf{y}) = \|\mathbf{y} - \hat{\mathbf{y}}\|^2 = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2$$

So we can easily write down the gradient of L with respect to $\mathbf{y}$:

$$\nabla_{\mathbf{y}} L = [2(y_1 - \hat{y}_1), (y_2 - \hat{y}_2), \ldots, 2(y_n - \hat{y}_n)] = 2(\mathbf{y} - \hat{\mathbf{y}})$$

□

The generalization of the gradient to vector-valued functions is called the *Jacobian*. Suppose we have a function $f : \mathbb{R}^m \to \mathbb{R}^n$ and $\mathbf{y} = f(\mathbf{x})$. The Jacobian $J_{\mathbf{x}}(\mathbf{y})$ is given by:[4]

[4]The Jacobian is sometimes defined as the transpose of our definition. The formulations are equivalent. Recall that we are assuming all vectors are column vectors unless transposed, but we show them as row vectors so they can be written in-line.

$$J_{\mathbf{x}}(\mathbf{y}) = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_n}{\partial x_1} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_1}{\partial x_m} & \cdots & \frac{\partial y_n}{\partial x_m} \end{bmatrix}$$

We shall make use of the *chain rule* for derivatives from calculus. If $y = g(x)$ and $z = f(y) = f(g(x))$, then the chain rule says:

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}$$

Also, if $z = f(u, v)$ where $u = g(x)$ and $v = h(x)$, then

$$\frac{dz}{dx} = \frac{\partial z}{\partial u}\frac{du}{dx} + \frac{\partial z}{\partial v}\frac{dv}{dx}$$

For functions of vectors, we can restate the chain rule in terms of gradients and Jacobians. Suppose $\mathbf{y} = g(\mathbf{x})$ and $z = f(\mathbf{y}) = f(g(\mathbf{x}))$ then:

$$\nabla_{\mathbf{x}} z = J_{\mathbf{x}}(\mathbf{y})\nabla_{\mathbf{y}} z$$

If $z = f(\mathbf{u}, \mathbf{v})$ where $\mathbf{u} = g(\mathbf{x})$ and $\mathbf{v} = h(\mathbf{x})$, then

$$\nabla_{\mathbf{x}} z = J_{\mathbf{x}}(\mathbf{u})\nabla_{\mathbf{u}} z + J_{\mathbf{x}}(\mathbf{v})\nabla_{\mathbf{v}} z$$

13.3.3 The Backpropagation Algorithm

The goal of the backpropagation algorithm is to compute the gradient of the loss function with respect to the parameters of the network. Consider the compute graph from Fig. 13.8. Here the loss function L is the MSE function. We shall use the notation $g(\mathbf{z})$ to stand for $\nabla_{\mathbf{z}}(L)$, that is, the gradient of the the loss function L with respect to some vector $\mathbf{z}$. We already know the gradient of L with respect to the output $\mathbf{y}$:

$$g(\mathbf{y}) = \nabla_{\mathbf{y}}(L) = 2(\mathbf{y} - \hat{\mathbf{y}})$$

We work backwards through the compute graph, applying the chain rule at each stage. At each point we pick a node all of whose successors have already been processed. Suppose $\mathbf{a}$ is such a node, and suppose it has just one immediate successor $\mathbf{b}$ in the graph (note that in the simple compute graph of Fig. 13.8, each node has just one immediate successor). Since we have already processed node $\mathbf{b}$, we have already computed $g(\mathbf{b})$. We can now compute $g(\mathbf{a})$ using the chain rule:

$$g(\mathbf{a}) = J_{\mathbf{a}}(\mathbf{b})g(\mathbf{b})$$

In the case where node $\mathbf{a}$ has more than one successor node, we use the more general sum version of the chain rule. That is, $g(\mathbf{a})$ would be the sum of the above terms for each successor $\mathbf{b}$ of $\mathbf{a}$.

Since we shall need to compute these gradients several times, once for each iteration of gradient descent, we can avoid repeated computation by adding additional nodes to the compute graph for backpropagation: one node for each gradient computation. In general, the Jacobian $J_{\mathbf{a}}(\mathbf{b})$ is a function of both $\mathbf{a}$ and $\mathbf{b}$, and so the node for $g(\mathbf{a})$ will have arcs to it from the nodes for $\mathbf{a}$, $\mathbf{b}$ and $g(\mathbf{b})$. Popular frameworks for deep learning (e.g., TensorFlow) know how to compute the functional expression for the Jacobians and gradients of commonly used operators such as those that appear in Fig. 13.8. In that case, the developer needs only to provide the compute graph and the framework will add the new nodes for backpropagation. Figure 13.9 shows the resulting compute graph with added gradient nodes.

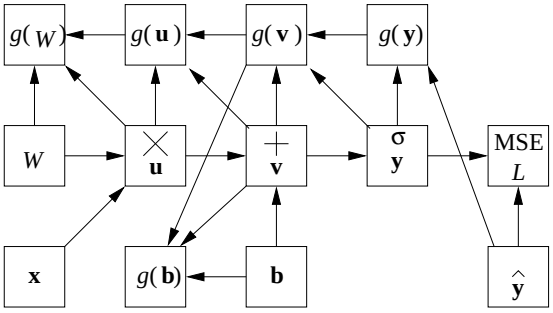


Figure 13.9: Compute graph with gradient nodes

Example 13.4: We shall work out the functional expressions for the gradients of all the nodes in Fig. 13.8. We already know $g(\mathbf{y})$. So the next node we choose to process is $\mathbf{v}$.

$$g(\mathbf{v}) = \nabla_{\mathbf{v}}(L) = J_{\mathbf{v}}(\mathbf{y})\nabla_{\mathbf{y}}(L) = J_{\mathbf{v}}(\mathbf{y})g(\mathbf{y})$$

We know that $\mathbf{y} = \sigma(\mathbf{v})$. Since σ is an element-wise operator, the Jacobian $J_{\mathbf{v}}(\mathbf{y})$ takes a particularly simple form. Using the derivative for the logistic sigmoid function from Section 13.2.2, we see that

$$\frac{\partial y_i}{\partial v_j} = \begin{cases} y_i(1-y_i) & \text{if } i = j \\ 0 & \text{otherwise} \end{cases}$$

The Jacobian is therefore a diagonal matrix:

$$J_{\mathbf{v}}(\mathbf{y}) = \begin{bmatrix} y_1(1-y_1) & 0 & \dots & 0 \\ 0 & y_2(1-y_2) & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & y_n(1-y_n) \end{bmatrix}$$

Suppose $\mathbf{s} = [s_1, s_2, \ldots, s_n]$ is a vector defined by $s_i = y_i(1 - y_i)$ (i.e., the diagonal of the Jacobian matrix). We can express $g(\mathbf{v})$ simply as

$$g(\mathbf{v}) = \mathbf{s} \circ g(\mathbf{y})$$

where $\mathbf{a} \circ \mathbf{b}$ is the vector resulting from the element-wise product of $\mathbf{a}$ and $\mathbf{b}$.[5]

Now that we have $g(\mathbf{v})$, we can compute $g(\mathbf{b})$ and $g(\mathbf{u})$. We have $g(\mathbf{b}) = J_{\mathbf{b}}(\mathbf{v})g(\mathbf{v})$ and $g(\mathbf{u}) = J_{\mathbf{u}}(\mathbf{v})g(\mathbf{v})$. Since

$$\mathbf{v} = \mathbf{u} + \mathbf{b}$$

it is straightforward to verify that

$$J_{\mathbf{b}}(\mathbf{v}) = J_{\mathbf{u}}(\mathbf{v}) = I_n$$

where I_n is the $n \times n$ identity matrix. So we have

$$g(\mathbf{b}) = g(\mathbf{u}) = g(\mathbf{v})$$

We finally come to the matrix W. Recall that $\mathbf{u} = W\mathbf{x}$. There is a potential problem here, because all the machinery we have set up works for vectors, while W is a matrix. But recall from Section 13.2.1 that we assembled the matrix W from a set of vectors $\mathbf{w}_1, \mathbf{w}_2, \ldots, \mathbf{w}_n$, where $\mathbf{w}_i^T$ is the ith row of W. The trick is to consider each of these vectors separately and compute its gradient using the usual formula.

$$g(\mathbf{w}_i) = J_{\mathbf{w}_i}(\mathbf{u})g(\mathbf{u})$$

We know that $u_i = \mathbf{w}_i^T\mathbf{x}$ and none of the other u_j's have any dependency on $\mathbf{w}_i$ for $i \neq j$. Therefore, the Jacobian $J_{\mathbf{w}_i}(\mathbf{u})$ is zero everywhere except the ith column, which is equal to $\mathbf{x}$. Thus we have

$$g(\mathbf{w}_i) = g(u_i)\mathbf{x}$$

□

Example 13.5: As we mentioned, neural networks for classification often use a softmax activation in the final layer followed by a cross-entropy loss. We will now compute the gradient of the combined operator.

Suppose the input to the combined operator is $\mathbf{y}$; let $\mathbf{q} = \mu(\mathbf{y})$, and let $l = H(\mathbf{p}, \mathbf{q})$, where $\mathbf{p}$ represents the true probability vector for the corresponding training example. We have:

$$\begin{aligned} \log(q_i) &= \log(\frac{e^{y_i}}{\sum_j e^{y_j}}) \\ &= y_i - \log(\sum_j e^{y_j}) \end{aligned}$$

[5] This operation sometimes called the *Hadamard product*, so as not to confuse it with the more usual dot product, which is the sum of the components of the Hadamard product.

Therefore, noting that $\sum_i p_i = 1$, we have:

$$\begin{aligned} l &= H(\mathbf{p}, \mathbf{q}) \\ &= -\sum_i p_i \log q_i \\ &= -\sum_i p_i (y_i - \log(\sum_j e^{y_j})) \\ &= -\sum_i p_i y_i - \log(\sum_j e^{y_j}) \sum_i p_i \\ &= -\sum_i p_i y_i - \log(\sum_j e^{y_j}) \end{aligned}$$

Differentiating, we get:

$$\begin{aligned} \frac{\partial l}{\partial y_k} &= -p_k + \frac{e^{y_k}}{\sum_j e^{y_j}} \\ &= -p_k + \mu(y_k) \\ &= q_k - p_k \end{aligned}$$

Therefore, we end up with the rather neat result:

$$\nabla_{\mathbf{y}} l = \mathbf{q} - \mathbf{p}$$

This combined gradient does not saturate or explode, and leads to good learning behavior. That observation explains why softmax and cross entropy loss work so well together in practice. □

13.3.4 Iterating Gradient Descent

Given a set of training examples, we run the compute graph in both directions for each example: forward (to compute the loss) and backwards (to compute the gradients). We average the loss and gradients across the training set to compute the average loss and the average gradient for each parameter vector.

At each iteration we update each parameter vector in the direction opposite to its gradient, so the loss will tend to decrease. Suppose $\mathbf{z}$ is a parameter vector. We set:

$$\mathbf{z} \leftarrow \mathbf{z} - \eta g(\mathbf{z})$$

Here η is a hyperparameter, the learning rate. We stop gradient descent either when the loss between successive iterations changes by only a tiny amount (i.e., we have reached a local minimum) or after a fixed number of iterations.

It is important to pick the learning rate carefully. Too small a value means gradient descent might take a very large number of iterations to converge. Too large a value might cause large oscillations in the parameter values and never

lead to convergence. Usually picking the right learning rate is a matter of trial and error. It is also possible and common to vary the learning rate. Start with an initial learning rate η_0. Then, at each iteration, multiply the learning rate by a factor β ($0 < \beta < 1$) until the learning rate reaches a sufficiently low value.

When we have a large training set, we may not want to use the entire training set for each iteration, as it might be too time-consuming. So for each iteration we randomly sample a "minibatch" of training examples. This variant is called stochastic gradient descent," as was discussed in Section 12.3.5, since we estimate the gradients using a different sample of the training set at each iteration.

We have left open the question of how the parameter values are initialized before we start gradient descent. The usual approach is to choose them at random. Popular approachaes include sampling each entry uniformly at random in $[-1, 1]$, or choosing randomly using a normal distribution. Notice that initializing all the weights to the same value would cause all nodes in a layer to behave the same way, and thus we would never reap the benefit of having different nodes in a layer recognize different features of the input.

13.3.5 Tensors

Previously, we have imagined that the inputs to a neural net are one-dimensional vectors. But there is no reason why we cannot view the input as having a higher dimension.

Example 13.6 : A grey-scale photo might be represented by a two-dimensional array of real numbers, corresponding to the intensity of each pixel. Each pixel of a color image typically requires 3 dimensions. That is, each pixel itself is a vector with three components, say for the intensity of the pixel in the red, green, and blue colors. One useful way to view a color image as input to a neural net is to think of each training example as a two-dimensional array of pixels, where the value of each pixel is not a real number, as we have heretofore imagined, but a vector with three dimensions, one for each of the three colors. □

Similarly, we have viewed each layer of a neural net as a column of nodes. But there is no reason we cannot imagine the nodes in a layer to be organized as a two-dimensional array or even an array of dimension greater than two. Finally, we have viewed the input values as real numbers and similarly viewed the values produced by each node as a real number. But we could also think of the values attached to each input or output of a node as a vector or a higher-dimensional structure. The natural generalization of vectors and matrices is the *tensor*, which is an n-dimensional array of scalars.

Unfortunately, the backpropagation algorithm we described works for vectors, not higher-dimensional tensors. In such cases, we resort to the same trick we used in Section 13.3.3, where we unrolled the matrix W into collection of

vectors. Just as we regard an $m \times n$ matrix as a set of m n-vectors, we can regard a 3-dimensional tensor of dimensionality $l \times m \times n$ as a set of lm n-vectors, and similarly for tensors of higher dimension.

Example 13.7: This example is based on the *MNIST dataset*.[6] This dataset consists of 28×28 monochrome images, each represented by a two-dimensional square bit array whose sides are of length 28. Our goal is to build a neural net that determines whether an image corresponds to a handwritten digit (0-9) and if so which one. Consider a single image X, which is a 28×28 matrix. Suppose the first layer of our network is a dense layer[7] consisting of 49 hidden nodes, which we shall imagine is arranged in a 7×7 array. We model the hidden layer as a 7×7 matrix H, where the output of the node in row i and column j is h_{ij}.

We can model the weights for each of the 28×28 inputs and each of the 7×7 nodes as a *weight tensor* W with dimensions $7 \times 7 \times 28 \times 28$. That is, W_{ijkl} represents the weight for the input pixel in row i and column j of the image to the node whose position in the array of nodes is row k and column l. Then:

$$h_{ij} = \sum_{k=1}^{28} \sum_{l=1}^{28} w_{ijkl} x_{kl} \text{ for } 1 \leq i, j \leq 7$$

where we omit the bias term for simplicity (i.e., we assume all thresholds of all nodes are 0.

An equivalent way to think about this structure to *flatten* the input X into a vector $\mathbf{x}$ of length 784 (since $28 \times 28 = 784$) and the hidden layer H into a vector $\mathbf{h}$ of length 49. We flatten the weight tensor W as follows: the last two dimensions are flattened into a single dimension to match $\mathbf{x}$, and its first two dimensions are flattened into a single dimension to match the hidden vector $\mathbf{h}$, resulting in a 49×784 weight matrix. Suppose as in Section 13.2.1, we have $\mathbf{w_i^T}$ denote the ith row of this new weight matrix. We can now write:

$$h_i = \mathbf{w_i^T x} \text{ for } 1 \leq i \leq 49$$

It's straightforward to see that there is a 1-to-1 mapping between the hidden nodes in the old and new arrangements. Moreover, the output of each hidden node is determined by a dot product, just as in Section 13.2.1. Thus the tensor notation is just a convenient way to group vectors. □

Thus the tensors used in neural networks have little in common with the tensors used in Physics and other mathematical sciences. A tensor in our context is just a nested collection of vectors. The only tensor operation we shall need is the *flattening* of a tensor by merging dimensions as in Example 13.7. We can use the backpropagation algorithm described in Section 13.3.3 for tensors once we have flattened them approriately.

[6] See `yann.lecun.com/exdb/mnist/`.

[7] In reality, the first network layer for this problem is likely to be a convolutional layer. See Section 13.4

13.3.6 Exercises for Section 13.3

Exercise 13.3.1: This exercise uses the neural net from Fig. 13.7. However, assume that the weights on all inputs are variables rather than the constants shown. Note, however, that some inputs do not feed one of the nodes in the first layer, so these weights are fixed at 0. Assume that the input vector is $\mathbf{x}$, the output vector is $\mathbf{y}$, and the output of the two nodes in the hidden layer is the vector $\mathbf{z}$. Also, let the matrix and bias vector connecting $\mathbf{x}$ to $\mathbf{z}$ be W_1 and $\mathbf{b}_1$, while the matrix and bias vector connecting $\mathbf{z}$ to $\mathbf{y}$ are W_2 and $\mathbf{b}_2$. Assume that the activation function at the hidden layer is the hyperbolic tangent, and the activation function at the output is the identity function (i.e., no change to the outputs is made). Finally, assume the loss function is mean-squared error, where $\hat{\mathbf{y}}$ is the true output vector for a given input vector $\mathbf{x}$. Draw the compute graph that shows how $\mathbf{y}$ is computed from $\mathbf{x}$.

Exercise 13.3.2: For the network described in Exercise 13.3.1:

(a) What is $J_{\mathbf{y}}(\mathbf{z})$?

(b) What is $J_{\mathbf{z}}(\mathbf{x})$?

(c) Express $g(\mathbf{x})$ in terms of $g(\mathbf{tanh}\,(\mathbf{z}))$.

(d) Express $g(\mathbf{x})$ in terms of the loss function.

(e) Draw the compute graph with gradient computation for the entire network.

13.4 Convolutional Neural Networks

Consider a fully-connected network layer for processing 224×224 images, with each pixel encoded using 3 color values (often called *channels* – the values of intensity for red, green, and blue)).[8] The number of weight parameters in the connection layer for each output node is $224 \times 224 \times 3 = 150,528$. Suppose we had 224 nodes in the output layer. We would end up with a total of over 33 million parameters! Given that our training set of images is usually of the order of tens or hundreds of thousands of images, the huge number of parameters would quickly lead to overfitting even using just a single layer.

A Convolutional Neural Network (CNN) greatly reduces the number of parameters by taking advantage of the properties of image data. CNN's introduce two new varieties of network layers: convolutional layers and pooling layers.

[8] This is the size of most of the images on ImageNet, for example.

13.4.1 Convolutional Layers

Convolutional layers make use of the fact that image features often are described by small contiguous areas in the image. For example, at the first convolutional layer, we might recognize small sections of edges in the image. At later layers, more complex structures, such as features that look like flower petals or eyes might be recognized. The idea that simplifies the calculation is the fact that the recognition of features such as edges does not depend on where in the image the edge is. Thus, we can train a single node to recognize a small section of edge, say an edge through a 5-by-5 region of pixels. This idea benefits us in two ways.

1. The node in question needs only inputs from 25 pixels corresponding to any 5-by-5 square, not inputs from all 224-by-224 pixels. That saves us a lot in the representation of the trained CNN.

2. The number of weights that we need to learn in the training process is greatly reduced. For each node in the layer, we require only one weight for each input to that node – say 75 weights if a pixel is represented by RGB values, not the 150,528 weights that we suggested above would be needed for an ordinary, fully connected layer.

We shall think of the nodes in a convolutional layer as *filters* for learning features. A filter examines a small spatially contiguous area of the image – traditionally, a small square area such as a 5×5 square of pixels. Moreover, since many features of interest may occur anywhere in the input image (and possibly in more than one location), we apply the same filter at many locations on the input.

To keep things simple, suppose the input consists of monochromatic images, that is, grey-scale images whose pixels are each a single value. Each image is thus encoded by a 2-dimensional pixel array of size 224×224. A 5×5 filter F is encoded by a 5×5 weight matrix W and single bias parameter b. When the filter is applied on a similarly-sized square region of input image X with the top left corner of the filter aligned with the image pixel x_{ij}, the *response* of the filter at this position, denoted r_{ij}, is given by:

$$r_{ij} = \sum_{k=0}^{4} \sum_{l=0}^{4} x_{i+k,j+l} w_{kl} + b \tag{13.1}$$

We now slide the filter along the length and width of the input image, applying the filter at each position, so that we capture every possible 5×5 square region of pixels in the image. Notice that we can apply the filter at input locations $1 \leq i \leq 220$ and $1 \leq j \leq 220$, although it does not "fit" at positions with a higher i or j. The resulting set of responses r_{ij} are then passed through an activation function to form the *activation map* R of the filter. In most cases, the activation function is ReLU or one of its variants. When trained, i.e., the

weights w_{ij} of the filter are determined, the filter will recognize some feature of the image, and the activation map tells whether (or to what degree) this feaure is present at each position of the image.

Example 13.8: In Fig. 13.10(b), we see a 2×2 filter, which is to be applied to the 4×4 image in Fig. 13.10(b). To do so, we lay the filter over all nine of the 2×2 squares of the image. In the figure, we suggest the filter being placed over the 2×2 square in the upper-right corner. After overlaying the filter, we multiply each of the filter elements by the corresponding image element and then take the sum of the products. In principle, we then need to add in a bias term, but in this example, we shall assume the bias is 0.

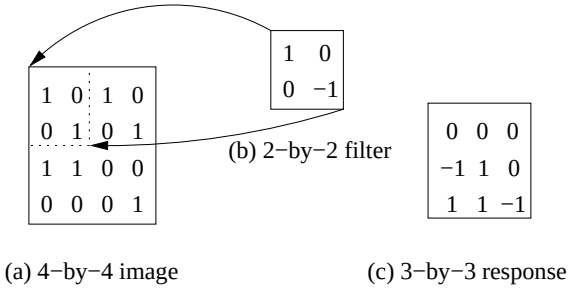


Figure 13.10: Applying a filter to an image

Another way to look at this process is that we turn the filter into a vector by concatenating its rows, in order, and we do the same to the square of the image. Then, we take the dot product of the vectors. For instance, the filter can be thought of as the vector $[1, 0, 0, -1]$, and the square in the upper-left corner of the image can be thought of as the vector $[1, 0, 0, 1]$. The dot product of these vectors is $1 \times 1 + 0 \times 0 + 0 \times 0 + (-1) \times 1 = 0$. Thus, the result, shown in Fig. 13.10(c), has a 0 for its upper-left entry.

For another example, if we slide the filter down one row, the dot product of the filter as a vector and the vector formed from the first two elements of the second and third rows of the image is $1 \times 0 + 0 \times 1 + 0 \times 1 + (-1) \times 1 = -1$. Thus, the first element of the second row of the convolution is -1. □

When we deal with color images, the input has three channels. That is, each pixel is represented by three values, one for each color. Suppose we have a color image of size $224 \times 224 \times 3$. The filter's output will also have three channels, and so the filter is now encoded by a $5 \times 5 \times 3$ weight matrix W and single bias parameter b. The activation map R still remains 5×5, with each response given by:

$$r_{ij} = \sum_{k=0}^{4} \sum_{l=0}^{4} \sum_{d=1}^{3} x_{i+k,j+l,d} w_{kld} + b \tag{13.2}$$

In our example, we imagined a filter of size 5. In general, the size of the filter is a hyperparameter of the convolutional layer. Filters of size 3, 4, or 5 are most

commonly used. Note that the filter size specifies only the width and height of the filter; the number of channels of the filter always matches the number of channels of the input.

The activation map in our example is slightly smaller than the input. In many cases, it is convenient to have the activation map be of the same size as the input. We can expand the repsonse by using *zero padding*: adding additional rows and columns of zeros to pad out the input. A zero padding of p corresponds to adding p rows of zeros each to the top and bottom, and p columns to the left and right, increasing the dimensionality of the input by $2p$ along both width and height. A zero padding of 2 in our example would augment the input size to 228×228 and result in an activation map of size 224×224, the same size as the original input image.

The third hyperparameter of interest is *stride*. In our example, we assumed that we apply the filter at every possible point in the input image. We could think instead of sliding the filter one pixel at a time along the width and height of the input, corresponding to a stride $s = 1$. Instead, we could slide the filter along the width and the height of image two or three pixels at a time, corresponding to a stride s of 2 or 3. The larger the stride, the smaller the activation map compared to the input.

Suppose the input is an $m \times m$ square of pixels, the output an $n \times n$ square, filter size is f, stride is s, and zero padding is p. It is easily seen that:

$$n = (m - f + 2p)/s + 1 \tag{13.3}$$

In particular, we must be careful to pick hyperparameters such that s evenly divides $m - f + 2p$; else we would have an invalid arrangement for the convolutional layer, and most software implementations would throw an exception.

We can intuitively think of a filter as looking for an image feature, such as a splotch of color or an edge. Classifying an image usually requires identifying many features. We therefore use many filters, ideally one for each useful feature. During the training of the CNN, we hope that each filter will learn to identify one of these features. Suppose we use k filters; to keep things simple, we constrain all filters to use same size, stride, and zero padding. Then the output contains k activation maps. The dimensionality of the output layer is therefore $n \times n \times k$, where n is given by Equation 13.3.

The set of k filters together constitute a *convolutional layer*. Given input with d channels, a filter of size f requires $df^2 + 1$ parameters (df^2 weight parameters and 1 bias parameter). Therefore, a convolutional layer of k such filters uses $k(df^2 + 1)$ parameters.

Example 13.9 : Continuing the ImageNet example, suppose the input consists of $224 \times 224 \times 3$ images, and we use a convolutional layer of 64 filters, each of size 5, stride 1, and zero padding 2. The size of the output layer is $224 \times 224 \times 64$. Each filter needs $3 \times 5 \times 5 + 1 = 76$ parameters (including one for the threshold) and the convolutional layer contains $64 \times 76 = 4864$ parameters – orders of magnitude smaller than the number of parameters for a fully connected layer with the same input and output sizes. □

13.4.2 Convolution and Cross-Correlation

This subsection is a short detour to explain why Convolutional Neural Networks are so named. It is not a pre-requisite for any of the other material in this chapter.

The convolutional layer is named because of the resemblance it bears to the convolution operation from functional analysis, which is often used in signal processing and probability theory. Given functions f and g, usually defined over the time domain, their convolution $(f * g)(t)$ is defined as:

$$(f * g)(t) = \int_{-\infty}^{\infty} f(\tau)g(t-\tau)d\tau = \int_{-\infty}^{\infty} f(t-\tau)g(\tau)d\tau$$

Here we are interested in the discrete version of convolution, where f and g are defined over the integers:

$$(f * g)(i) = \sum_{k=-\infty}^{\infty} f(k)g(i-k) = \sum_{k=-\infty}^{\infty} f(i-k)g(k)$$

Often convolution is viewed as using function g to transform function f. In this context, the function g is sometimes called the *kernel*. When the kernel is finite, so $g(k)$ is only defined for $k = 0, 1, \ldots, m-1$, the definition simplifies to:

$$(f * g)(i) = \sum_{k=0}^{m-1} f(i-k)g(k)$$

We can extend the definition to 2-dimensional functions:

$$(f * g)(i,j) = \sum_{k=0}^{m-1}\sum_{l=0}^{m-1} f(i-k, j-l)g(k,l)$$

Let us define h to be the kernel obtained by *flipping* g, i.e., $h(i,j) = g(-i,-j)$ for $i, j \in \{0, \ldots, m-1\}$. It can verified that the convolution $f * h$ of f with the flipped kernel h is given by:

$$(f * h)(i,j) = \sum_{k=0}^{m-1}\sum_{l=0}^{m-1} f(i+k, j+l)g(k,l) \tag{13.4}$$

Note the similarity of Equation 13.4 to Equation 13.1, ignoring the bias term b. The operation of the convolutional layer can be thought of as the convolution of the input with a flipped kernel. This similarity is the reason why convolutional layers are so named, and filters are sometimes called kernels.

The *cross-correlation* $f \star g$ is defined by $(f \star g)(x,y) = (f * h)(x,y)$ where h is the flipped version of g. Thus the operation of the convolutional layer can also be seen as the cross-correlation of the input with the filter.

13.4.3 Pooling Layers

A pooling layer takes as input the output of a convolutional layer and produces an output with smaller spatial extent. The size reduction is accomplished by using a *pooling function* to compute aggregates over small contiguous regions of the input. For example we might use *max pooling* over nonoverlapping 2×2 regions of the input; in this case there is an output node corresponding to every nonoverlapping 2×2 region of the input, with the output value being the maximum value among the 4 inputs in the region. The aggregation operates independently on each channel of the input. The resulting output layer is 25% the size of the input layer. There are three elements in defining a pooling layer:

1. The *pooling function*, which is most commonly the max function but could in theory be any aggregate function, such as average.

2. The *size* f of each pool, which specifies that each pool uses an $f \times f$ square of inputs.

3. The stride s between pools, analogous to the stride used in the convolutional layer.

The most common use cases in practice are $f = 2$ and $s = 2$, which specifies nonoverlapping 2×2 regions, and $f = 3$, $s = 2$, which specifies 3×3 regions with some overlap. Higher values of f lead to too much loss of information in practice. Note that the pooling operation shrinks the height and width of the input layer, but preserves the number of channels. It operates independently on each channel of its input. Note that unlike the convolution layer, it is not common practice to use zero padding for the max pooling layer.

Pooling is appropriate if we we believe that features are approximately invariant to small translations. For example, we might care about the relative locations of features such as legs or wings and not their exact locations. In such cases pooling can greatly reduce the size of the hidden layer that forms the input to the subsequent layers of the network.

Example 13.10: Suppose we apply max pooling with size = 2 and stride = 2 to the $224 \times 224 \times 64$ output of the convolutional layer from Example 13.9. The resulting output is of size $112 \times 112 \times 64$. □

13.4.4 CNN Architecture

Now that we have seen the building blocks of Convolutional Neural Networks, we can put them together to build deep networks. A typical CNN alternates convolutional and pooling layers, ending with one or more fully-connected layers that produce the final output.

Example 13.11: For instance, Fig. 13.11 is a simple network architecture for classifying ImageNet images into one of 1000 image classes, loosely inspired by

VGGnet.[9] This simple network strictly alternates convolutional and pooling layers. In practice, high-performing networks are finely tuned to their task, and may stack convolutional layers directly on top of one another with the occasional pooling layer in between. Moreover, there is often more than one fully-connected layer before the final output. The input to the first layer is the 224-by-224, 3-channel image. The input to each subsequent layer is the output of the previous layer.

Layer Type	Size	Stride	Pad	Filter Count	Output Size
Convolution	3	1	1	64	$224 \times 224 \times 64$
Max Pool	2	2	0	64	$112 \times 112 \times 64$
Convolution	3	1	1	128	$112 \times 112 \times 128$
Max Pool	2	2	0	128	$56 \times 56 \times 128$
Convolution	3	1	1	256	$56 \times 56 \times 256$
Max Pool	2	2	0	256	$28 \times 28 \times 256$
Convolution	3	1	1	512	$14 \times 14 \times 512$
Max Pool	2	2	0	512	$14 \times 14 \times 512$
Convolution	3	1	1	1024	$14 \times 14 \times 1024$
Max Pool	2	2	0	1024	$7 \times 7 \times 1024$
Fully Connected					$1 \times 1 \times 1000$

Figure 13.11: Layers of a Convolutional Neural Network

The first layer is a convolutional layer, consisting of 64 filters, each with three channels, as would be the case for any color-image processor. The filters are 3-by-3, and the stride is 1, so every 3-by-3 square of the image is an input to the filter. There is one unit of zero-padding, so the number of outputs of this layer equals the number of inputs. Further, notice that we can view the output as another 224-by-224 array. Each element of this array consists of 64 filters, each of which is a 3-channel pixel.

The output of the first layer is fed to a max-pool layer, in which we divide the 224-by-224 array into 2-by-2 squares (because both the size and stride are 2). Thus, the 224-by-224 array has become a 112-by-112 array, and there are still the same 64 filters.

At the third layer, which is again convolutional, we take the 112-by-112 array of pixels from the second layer as input. This layer has more filters than the first layer – 128 to be exact. The intuitive reason for the increase is that the first layer recognizes very simple structures, like edges, and there are not too many different simple structures. However, the third layer should be recognizing somewhat more complex features, and these features can involve a 6-by-6 square of the input image, because of the pooling done at the second layer. Similarly, each subsequent convolutional layer takes inputs from the

[9] K. Simonyan and A. Zussman,"Very Deep Convolutional Networks for Large-Scale Image Recognition," arXiv:1409–1556v6.

How Many Nodes in a Convolutional Layer?

We have referred to a node of a convolutional layer as a "filter." That filter may be a single node, or as in Example 13.11, a set of several nodes, one for each channel. When we train the CNN, we determine weights for each filter, so there are relatively few nodes. For instance, in the first layer in Example 13.11, there are 192 nodes, three for each of the 64 filters. However, when we apply the trained CNN to an input, we apply each filter to every pixel in the input. Thus, in Example 13.11, each of the 64 filters of the first layer is applied to $224 \times 224 = 50,176$ pixels. The key point to remember is that although a CNN has a succinct representation, the application of the represented neural net to data requires a significant amount of computation. The same, by the way, can be said for the other specialized forms of neural net that we discuss in Section 13.5.

previous pooling layer and its filters can represent structures of progressively larger sizes and complexities. Thus, the number of filters has been chosen to double at each convolution layer.

Finally, the eleventh, and last, layer is a fully connected layer. It has 1000 nodes, corresponding to the 1000 images classes we are trying to learn how to recognize. Being fully connected, each of these 1000 nodes takes all $7 \times 7 \times 3 = 147$ outputs from the 10th layer; the factor 3 is from the fact that all filters of the previous layers have three channels. □

Designing CNN's and other deep network architectures is still more art than science. Over the past few years, however, some rules of thumb have emerged that are worth keeping in mind:

1. Deep networks that use many convolutional layers, each using many small filters, are better than shallow networks that use large filters.

2. A simple pattern is to use convolutional layers that preserve the spatial extent of their input by zero padding, and have size reduction done exclusively by pooling layers.

3. Smaller strides work better in practice than larger strides.

4. It's very useful to have the input size evenly divisible by 2 many times.

13.4.5 Implementation and Training

An examination of Equation 13.1 (generalized so the filter is $f \times f$ rather than 5-by-5) suggests that we can express each entry in the output of the convolution as the dot product of vectors, followed by a scalar sum. To do so, we must flatten

the filter $\boldsymbol{F}$ and the corresponding region in the input into vectors. Consider the convolution of an $m \times m \times 1$ tensor $\boldsymbol{X}$ (i.e., $\boldsymbol{X}$ is actually an $m \times m$ matrix) with an $f \times f$ filter $\boldsymbol{F}$ and bias b, to produce as output the $n \times n$ matrix $\boldsymbol{Z}$. We now explain how to implement the convolution operation using a single vector-matrix multiplication.

We first flatten the filter $\boldsymbol{F}$ into a $f^2 \times 1$ vector $\boldsymbol{g}$. We then create matrix $\boldsymbol{Y}$ from $\boldsymbol{X}$ as follows: each square $f \times f$ region of $\boldsymbol{X}$ is flattened into a $f^2 \times 1$ vector, and all these vectors are lined up as columns to form a single $f^2 \times n^2$ matrix $\boldsymbol{Y}$. Construct the $n^2 \times 1$ vector $\boldsymbol{b}$ so that all its entries are equal to the bias b. Then

$$\boldsymbol{z} = \boldsymbol{Y}^\top \boldsymbol{g} + \boldsymbol{b}$$

yields a $n^2 \times 1$ vector $\boldsymbol{z}$. Moreover, each element of $\boldsymbol{z}$ is a single element in the convolution. Therefore, we can rearrange the entries in $\boldsymbol{z}$ into an $n \times n$ matrix $\boldsymbol{Z}$ that gives the output of the convolution.

Notice that the matrix $\boldsymbol{Y}$ is larger than the input $\boldsymbol{X}$ (approximately by a factor of f^2), because each entry of $\boldsymbol{X}$ is repeated many times in $\boldsymbol{Y}$. Thus, this implementation uses a lot of memory. However, multiplying a matrix and a vector is extremely fast on modern hardware such as Graphics Processing Units (GPU's), and so it is the method used in practice.

This approach to computing convolutions can easily be extended to the case of inputs with more than one channel. Moreover, we can also handle the case where we have k filters rather than just 1. We then need to replace the vector $\boldsymbol{g}$ with a $df^2 \times k$ matrix $\boldsymbol{G}$ and use a larger matrix $\boldsymbol{Y}$ ($df^2 \times n^2$). We also need to use an $n^2 \times k$ bias matrix $\boldsymbol{B}$, where each column repeats the bias term of the corresponding filter. Finally, the output of the convolution is expressed by an $n^2 \times k$ matrix $\boldsymbol{C}$, with a column for the output of each filter, where:

$$\boldsymbol{C} = \boldsymbol{Y}^\top \boldsymbol{F} + \boldsymbol{B}$$

We have explained how to perform the forward pass through the convolutional layer. During training, we shall need to backpropagate through the layer. Since each entry in the output of convolution is a dot product of vectors followed by a sum, we can use the techniques from Section 13.3.3 to compute derivatives. It turns out that the derivative of a convolution can also be expressed as a convolution, but we shall not go into the details here.

13.4.6 Exercises for Section 13.4

Exercise 13.4.1: Suppose images are 512-by-512, and we use a filter that is 3-by-3.

(a) How many responses will be computed for this layer of a CNN?

(b) How much zero padding is necessary to produce an output of size equal to the input?

(c) Suppose we do not do any zero padding. If the output of one layer is input to the next layer, after how many layers will there be no output at all?

Exercise 13.4.2: Repeat Exercise 13.4.1(a) and (c) for the case when there is a stride of three.

Exercise 13.4.3: Suppose we have the output of an m-by-m convolutional layer with k filters, each having d channels. These outputs are fed to a pooling layer with size f and stride s. How many output values does the pooling layer produce?

Exercise 13.4.4: For this exercise, assume that inputs are single bits 0 (white) and 1 (black). Consider a 3-by-3 filter, whose weights are w_{ij}, for $0 \leq i \leq 2$ and $0 \leq j \leq 2$, and whose bias is b. Suggest wieghts and bias so that the output of this filter would detect the following simple features:

(a) A vertical boundary, where the left column is 0, and the other two columns are 1.

(b) A diagonal boundary, where only the triangle of three pixels in the upper right corner are 1.

(c) a corner, in which the 2-by-2 square in the lower right is 0 and the other pixels are 1.

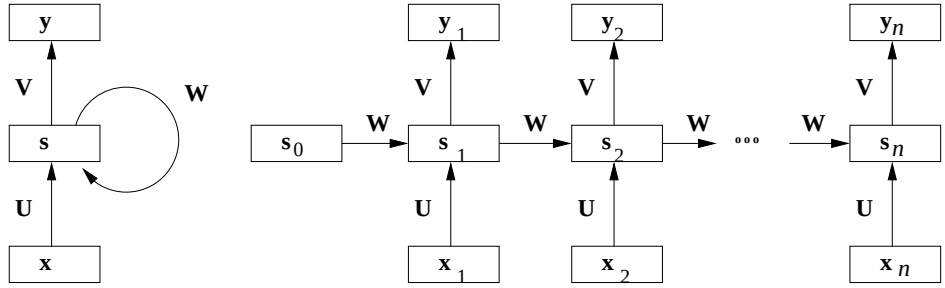
13.5 Recurrent Neural Networks

Just as CNN's are a specialized family of neural networks for processing 2-dimensional image data, recurrent neural networks (RNN's) are networks specially designed for processing sequential data. Sequential data naturally arises in many settings: a sentence is a sequence of words; a video is a sequence of images; a stock market ticker is a sequence of prices.

Consider a simple example from the field of language processing. Each input is a sentence, modeled as a sequence of words. After processing each prefix of the sequence, we would like to predict the next word in the sentence; the output at each step is a probability vector across words. Our example suggests two key design considerations:

1. The output at each point depends on the entire prefix of the sentence until that point, and not just the last word. The network needs to retain some "memory" of the past.

2. The underlying language model does not change across positions in the sequence, so we should use the same parameters (weights for each of the nodes) at each position.



(a) The basic unit of an RNN. (b) The unrolled RNN of length *n*.

Figure 13.12: RNN architecture

These considerations lead naturally to a *recurrent* network model, where we perform the same operation at each step, with the input to each step being dependent upon the output from prior steps. Figure 13.12 shows the structure of a typical recurrent neural network. The input is a sequence $\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_n$, and the output is also a sequence $\boldsymbol{y}_1, \boldsymbol{y}_2, \ldots, \boldsymbol{y}_n$. In our example, each input $\boldsymbol{x}_i$ represents a word, and each output $\boldsymbol{y}_i$ is a probability vector for the next word in the seentence. The input $\boldsymbol{x}_i$ is typically encoded as a *1-hot vector* – a vector of length equal to the number of possible words,[10] with a 1 in the position corresponding to the input word and a 0 in all other positions. There are two important difference between a general neural network and the RNN:

1. The RNN has inputs at all (or almost all) layers, and not just at the first layer.

2. The weights at each of the first n layers are constrained to be the same; these weights are the matrices U and W in Equation 13.5 below. Thus, each of the first n layers has the same set of nodes, and corresponding nodes from each of the layers share weights (and are thus really the same node), just as nodes of a CNN representing different locations share weights and are thus really the same node.

At each step t, we have a hidden state vector $\boldsymbol{s}_t$ that functions as the memory in which the network encodes information about the prefix of the sequence it has seen. The hidden state at time t is a function of the input at time t and the hidden state at time $t-1$:

$$\boldsymbol{s}_t = f(U\boldsymbol{x}_t + W\boldsymbol{s}_{t-1} + \boldsymbol{b}) \tag{13.5}$$

[10]Since there could in principle be an infinite number of words, we might in practice devote components of the vector only to the most common words or the words that are most important in the application at hand. Other words would all be represented by a single additional component of the vector.

Here f is a nonlinear activation function such as tanh or sigmoid. U and W are matrices of weights, and $\boldsymbol{b}$ is a vector of biases. We define $\boldsymbol{s}_0$ to be a vector of all zeros. The output at time t is a function of the hidden state at time t, after being transformed by a parameter matrix V and an activation function g:

$$\boldsymbol{y}_t = g(V\boldsymbol{s}_t + \boldsymbol{c})$$

In our example, g might be the softmax function to ensure that the output is a valid probability vector.

The RNN in Figure 13.12 has an output at each time step. In some applications, such as machine translation, we need just a single output at the end of each sentence. In such cases, the RNN's single output is further processed by one or more fully-connected layers to generate the final output.

It is simplest to assume that our RNN's inputs are fixed-length sequences of length n. In this case, we simply unroll the RNN to contain n time steps. In practice, many applications have to deal with variable-length sequences e.g., sentences of varying lengths. There are two approaches to deal with this situation:

1. *Zero-padding.* Fix n to be the longest sequence we process, and pad out shorter sequences to be length n.
2. *Bucketing.* Group sequences according to length, and build a separate RNN for each length.

A combination of these two approaches is used. We can create a bucket for a small number of different lengths, and assign a sequence to the bucket of the shortest length that is at least as long as the sequence. Then, within a bucket we use padding for sequences that are shorter than the maximum length for that bucket.

13.5.1 Training RNN's

We use backpropagation to train an RNN, just as we would any neural network. Let us work through an example. Suppose our input consists of sequences of length n. Our network uses the activation function tanh for the state update, softmax for the output, and the loss function is cross-entropy. Since the network has several outputs, one at each time-step, we seek to minimize the total error e, defined as the sum of the losses e_i at each time step i:

$$e = \sum_{i=1}^{n} e_i$$

To simplify notation, we use the following conventions. Suppose $\boldsymbol{x}$ and $\boldsymbol{y}$ are vectors, and z is a scalar. We define:

$$\begin{aligned} \frac{dz}{d\boldsymbol{x}} &= \nabla_{\boldsymbol{x}} z \\ \frac{d\boldsymbol{y}}{d\boldsymbol{x}} &= J_{\boldsymbol{x}}(\boldsymbol{y}) \end{aligned}$$

Moreover, suppose W is a matrix, and $\boldsymbol{w}$ is the vector obtained by concatenating the rows of W. Then:

$$\begin{aligned} \frac{dz}{dW} &= \frac{dz}{d\boldsymbol{w}} \\ \frac{d\boldsymbol{y}}{dW} &= \frac{d\boldsymbol{y}}{d\boldsymbol{w}} \end{aligned}$$

These conventions also extend naturally to partial derivatives.

We use backpropagation to compute the gradients of the error with respect to the network parameters. We focus on $\frac{de}{dW}$; the gradients for U and V are similar, and left as exercises for the reader. It is clear that:

$$\frac{de}{dW} = \sum_{t=1}^{n} \frac{de_t}{dW}$$

Focusing on step t, we have:

$$\frac{de_t}{dW} = \frac{ds_t}{dW}\frac{de_t}{ds_t}$$

We leave it as an exercise to verify that:

$$\frac{de_t}{ds_t} = V^{\mathsf{T}}(\boldsymbol{y}_t - \hat{\boldsymbol{y}}_t) \tag{13.6}$$

Setting $R_t = \frac{ds_t}{dW}$, we note that $\boldsymbol{s}_t = \tanh(\boldsymbol{z}_t)$, where $\boldsymbol{z}_t = W\boldsymbol{s}_{t-1} + U\boldsymbol{x}_t + b$, we have:

$$R_t = \frac{d\boldsymbol{s}_t}{d\boldsymbol{z}_t}\frac{d\boldsymbol{z}_t}{dW}$$

It is straightforward to verify that $\frac{d\boldsymbol{s}_t}{d\boldsymbol{z}_t}$ is the diagonal matrix A defined by:

$$a_{ij} = \begin{cases} 1 - s_{t_i}^2 & \text{when } i = j \\ 0 & \text{otherwise} \end{cases}$$

We should to be careful to note that $\boldsymbol{z}_t$ is a function of W both directly and indirectly, since $\boldsymbol{s}_{t-1}$ also depends on W. So we must express $\frac{d\boldsymbol{z}_t}{dW}$ as a sum of two terms:

$$\frac{d\boldsymbol{z}_t}{dW} = \frac{\partial \boldsymbol{z}_t}{\partial W} + \frac{\partial \boldsymbol{z}_t}{\partial \boldsymbol{s}_{t-1}}\frac{d\boldsymbol{s}_{t-1}}{dW}$$

It is easily verified that:

$$\frac{\partial \boldsymbol{z}_t}{\partial \boldsymbol{s}_{t-1}} = W^{\mathsf{T}}$$

The form of $\frac{\partial \boldsymbol{z}_t}{\partial W}$ is a little trickier. It is a matrix B with mostly zero entries, the nonzero elements being entries from $\boldsymbol{s}_{t-1}$. We leave the computation of B

as an exercise for the reader. Now, noting that $\frac{d\boldsymbol{s}_{t-1}}{dW}$ is just R_{t-1}, we have the recurrence:

$$R_t = A(B + W^{\mathsf{T}} R_{t-1})$$

Setting $P_t = AB$ and $Q_t = AW^{\mathsf{T}}$, we end up with:

$$R_t = P_t + Q_t R_{t-1} \tag{13.7}$$

We can use this recurrence to set up an iterative evaluation of R_t, and thence $\frac{de}{dW}$. We initialize the iteration by setting R_0 to the matrix with all zeros. This iterative method for computing the gradients of an RNN is called *Backpropagation Through Time* (BPTT), since it reveals the effects of earlier time-steps on later time-steps.

13.5.2 Vanishing and Exploding Gradients

RNN's are a simple and appealing model for learning from sequences, and the BPTT algorithm is straightforward to implement. Unfortunately, RNN's have a fatal flaw that limits their use in many practical applications. They are effective only at learning short-term connections between nearby elements in the sequence and ineffective at learning long-distance connections. Long-distance connections are crucial in many applications; for example, there can be an arbitrary number of words or clauses separating a verb or a pronoun from the subject with which it is associated.

In order to understand the cause of this limitation, let us unroll Equation 13.7:

$$\begin{aligned} R_t &= P_t + Q_t R_{t-1} \\ &= P_t + Q_t(P_{t-1} + Q_{t-1} R_{t-2}) \\ &= P_t + Q_t P_{t-1} + Q_t Q_{t-1} R_{t-2} \\ &\dots \end{aligned}$$

Ultimately yielding:

$$R_t = P_t + \sum_{j=0}^{t-1} P_j \prod_{k=j+1}^{t} Q_k \tag{13.8}$$

From Equation 13.8, it is clear that the contribution of step i to R_t is given by:

$$R_t^i = P_i \prod_{k=i+1}^{t} Q_k \tag{13.9}$$

Equation 13.9 includes the product of several matrices that look like diagonal matrix A. Each entry in A is strictly less than 1. Just as the product of many numbers, each strictly less than 1, approaches zero as we add more multiplicands, the term $\prod_{k=i+1}^{t} Q_k$ approaches zero for $i \ll t$. In other words, the gradient at step t is determined entirely by the preceding few time steps, with

very little contribution from much earlier time steps. This phenomenon is called the problem of *vanishing gradients*.

Equation 13.9 results in vanishing gradients because we used the tanh activation function for state update. If instead we use other activation functions such as ReLU, we end up with the product of many matrices with large entries, resulting in the problem of *exploding gradients*. Exploding gradients are easier to handle than vanishing gradients, because we can *clip* the gradient at each step to lie within a fixed range. However, the resulting RNN's still have trouble learning long-distance associations.

13.5.3 Long Short-Term Memory (LSTM)

The LSTM model is a refinement of the basic RNN model to address the problem of learning long-distance associations. In the past few years, LSTM has become popular as the de-facto sequence-learning model, and has been used with success in many applications. Let us understand the intuition behind the LSTM model before we describe it formally. The main elements of the LSTM model are:

1. The ability to *forget* information by purging it from memory. For example, when analyzing a text we might want to discard information about a sentence when it ends. Or when analyzing a the sequence of frames in a movie, we might want to forget about the location of a scene when the next scene begins.

2. The ability to *save* selected information into memory. For example, when we process product reviews, we might want to save only words expressing opinions (e.g., excellent, terrible) and ignore other words.

3. The ability to *focus* only on the aspects of memory that are immediately relevant. For example, focus only on information about the characters of the current movie scene, or only on the subject of the sentence currently being analyzed. We can implement this focus by using a 2-tier architecture: a long-term memory that retains information about the entire processed prefix of the sequence, and a working memory that is restricted to the items of immediate relevance.

The RNN model has a single hidden state vector $\boldsymbol{s}_t$ at time t. The LSTM model adds an additional state vector $\boldsymbol{c}_t$, called the *cell state*, for each time t. Intuitively, the hidden state corresponds to working memory and the cell state corresponds to long-term memory. Both state vectors are of the same length, and both have entries in the range $[-1, 1]$. We may imagine the working memory having most of its entries near zero with only the relevant entries turned "on."

The architectural ingredient that enables the ability to forget, save, and focus is the *gate*. A gate $\boldsymbol{g}$ is just a vector of the same length as a state vector

$\boldsymbol{s}$; each of the gate's entries is between 0 and 1. The Hadamard product[11] $\boldsymbol{s} \circ \boldsymbol{g}$ allows us to selectively pass through certain parts of the state while filtering out others. Usually, a gate vector is created by a linear combination of the hidden state and the current input. We then apply a sigmoid function to "squash" its entries to lie between 0 and 1. In general, an LSTM may use several different kinds of gate vectors, each for a different purpose. At time t, we can create a gate $\boldsymbol{g}$ as follows:

$$\boldsymbol{g} = \sigma(W\boldsymbol{s}_{t-1} + U\boldsymbol{x}_t + \boldsymbol{b})$$

Here W and U are weight matrices and $\boldsymbol{b}$ is a bias vector.

At time t, we first compute a candidate state update vector $\boldsymbol{h}_t$ based on the previous hidden state and the current input:

$$\boldsymbol{h}_t = \tanh(W_h\boldsymbol{s}_{t-1} + U_h\boldsymbol{x}_t + \boldsymbol{b}_h) \tag{13.10}$$

Note that W, U, and $\boldsymbol{b}$ with subscript h are two weight matrices and a bias vector that we learn and use for just the purpose of computing $\boldsymbol{h}_t$ for each t.

We also compute two gates, the *forget gate* $\boldsymbol{f}_t$ and the *input gate* $\boldsymbol{i}_t$. The forget gate determines which aspects of the long-term memory we retain. The input gate determines which parts of the candidate state update to save into the long-term memory. These gates are computed using different weight matrices and bias vectors, which also must be learned. We indicate these matrices and vector with subscripts f and i, respectively.

$$\boldsymbol{f}_t = \sigma(W_f\boldsymbol{s}_{t-1} + U_f\boldsymbol{x}_t + \boldsymbol{b}_f) \tag{13.11}$$

$$\boldsymbol{i}_t = \sigma(W_i\boldsymbol{s}_{t-1} + U_i\boldsymbol{x}_t + \boldsymbol{b}_i) \tag{13.12}$$

We update the long-term memory using the gates and the candidate update vector as follows:[12]

$$\boldsymbol{c}_t = \boldsymbol{c}_{t-1} \circ \boldsymbol{f}_t + \boldsymbol{h}_t \circ \boldsymbol{i}_t \tag{13.13}$$

Now that we have updated the long-term memory, we need to update the working memory. We do this in two steps. The first step is to create an *output gate* $\boldsymbol{o}_t$. The second step is to apply this gate to the long-term memory, followed by a tanh activation:[13]

$$\boldsymbol{o}_t = \sigma(W_o\boldsymbol{s}_{t-1} + U_o\boldsymbol{x}_t + \boldsymbol{b}_o) \tag{13.14}$$

$$\boldsymbol{s}_t = \tanh(\boldsymbol{c}_t \circ \boldsymbol{o}_t) \tag{13.15}$$

[11] The *Hadamard product* of vectors $[x_1, x_2, \ldots, x_n]$ and $[y_q, y_2, \ldots, y_n]$ is the vector whose components are the products of the corresponding components of the two argument vectors, that is, $[x_1y_1, x_2y_2, \ldots, x_ny_n]$. The same operation may be applied to any matrices that have the same dimensions.

[12] Technically, an entry of 1 in the forget gate results in retaining the corresponding memory entry; so the forget gate should really be called the *remember gate*. Similarly, the input gate might be better named the *save gate*. Here we follow the naming convention commonly used in the literature.

[13] Once again, the output gate might be better named the *focus gate* since it focuses the working memory on certain aspects of the long-term memory.

Here, we use subscript o to indicate another pairs of weight matrices and a bias vector that must be learned.

Finally, the output at time t is computed in exactly the same manner as the RNN output:

$$\boldsymbol{y}_t = g(V\boldsymbol{s}_t + \boldsymbol{d}) \tag{13.16}$$

where g is an activation function, V is a weight matrix and $\boldsymbol{d}$ is a bias vector.

Equations 13.10 through 13.16 describe the state update operations at a single time step t of an LSTM. We can think of plain RNN as a special case of LSTM. When we set the forget gate to all 0's (so we throw away all prior long-term memory) and the input gate to all 1's (save the entire candidate state update), and the output gate to all 1's (working memory is same as long-term memory), we get something that looks very close to an RNN, the only difference being an extra tanh factor.

The ability to selectively forget allows LSTM's to avoid the vanishing-gradient problem at the expense of introducing many more parameters than vanilla RNN's. While we will not provide a rigorous proof here, we note that the key to avoiding vanishing gradients is the long-term memory update in Equation 13.13. Several variations of the basic LSTM model have been proposed; the most common variant is the *gated recurrent Unit* (GRU) model, which uses a single state vector instead of two state vectors (long-term and short-term). A GRU has fewer parameters than an LSTM and might be suitable in some situations with smaller data sets.

13.5.4 Exercises for Section 13.5

Exercise 13.5.1: In this exercise, you are asked to design the input weights for one or more nodes of the hidden state of an RNN. The input is a sequence of bits, 0 or 1 only.[14] Note that you can use other nodes to help with the node requested. Also note that you can apply a transformation to the output of the node so a "yes" answer has one value and a "no" answer has another.

(a) A node to signal when the input is 1 and the previous input is 0.

! (b) A node to signal when the last three inputs have all been 1.

!! (c) A node to signal when the input is the same as the previous input.

! Exercise 13.5.2: Verify Equation 13.6.

! Exercise 13.5.3: Give the formulas for the gradients $\frac{de}{dU}$ and $\frac{de}{dV}$ for the general RNN of Fig.13.12.

[14] We should understand that RNN's, like any neural network, is to be learned from data, not designed as we are suggesting you do here.

13.6 Regularization

Thus far, we have presented our goal as one of minimizing loss (i.e., prediction error) on the training set. Gradient descent and stochastic gradient descent help us achieve this objective. In practice, the real objective of training is to minimize the loss on new and hitherto unseen inputs. Our hope is that our training set is representative of unknown future inputs, so a low loss on the training set translates into good performance on new inputs. Unfortunately, the trained model sometimes learns idiosyncrasies of the training data that allow it have low training loss, but not generalize well to new inputs – the familiar problem of overfitting.

How can we tell if a model has overfit? In general, we split the available data into a training set and a test set. We train the model using only the training-set data, withholding the test set. We then evaluate the performance of the model on the test set. If the model performs much worse on the test set than on the training set, we know the model has overfit. Assuming data points are independent of one another, we can pick a fraction of the available data points at random to form the test set. A common ratio for the training:test split is 80:20. i.e, 80% of the data for training and 20% for test. We have to be careful, however: in sequence-learning problems (e.g., modeling time series), the state of the sequence at any point in time encodes information about the past. In such cases the final piece of the sequence is a better test set.

Overfitting is a general problem that affects all machine-learning models. However, deep neural networks are particularly susceptible to overfitting, because they use many more parameters (weights and biases) than other kinds of models. Several techniques have been developed to reduce overfitting in deep networks, usually by trading higher training error for better generalization. The process is referred to as *model regularization*. In this section we describe some of the most important regularization methods for deep learning.

13.6.1 Norm Penalties

Gradient descent is not guaranteed to learn parameters (weights and biases) that reduce the training loss to an absolute minimum. In practice, the procedure learns parameters that correspond to a *local minimum* in the training loss. There are usually many local minima, and some might lead to better generalization than others. In practice, it has been observed that solutions where the learned weights have low absolute values tend to generalize better than solutions with large weights.

We can force gradient descent to favor solutions with low weight values by adding a term to the loss function. Suppose $\boldsymbol{w}$ is the vector of all the weight values in the model, and L_0 is loss function used by our model. We define a new loss function L as follows:

$$L = L_0 + \alpha \left\| \boldsymbol{w} \right\|^2 \tag{13.17}$$

The loss function L penalizes large weight values. Here α is a hyperparameter that trades off between minimizing the original loss function L_0 and the penalty associated with the L_2-norm of the weights. Instead of the L_2-norm, we could penalize the L_1-norm of the weights:

$$L = L_0 + \alpha \sum_i |w_i| \tag{13.18}$$

In practice, it is observed that the L_2-norm penalty works best for most applications. The L_1-norm penalty is useful in some situations calling for model compression, because it tends to produce models where many of the weights are zero.

13.6.2 Dropout

Dropout is a technique that reduces overfitting by making random changes to the underlying deep neural network. Recall that when we train using stochastic gradient descent, at each step we sample at random a minibatch of inputs to process. When using dropout, we also select at random a certain fraction (say half) of all the hidden nodes from the network and delete them, along with any edges connected to them. We then perform forward propagation and back-propagation for the minibatch using this modified network, and update the weights and biases. After processing the minibatch, we restore all the deleted nodes and edges. When we sample the next minibatch, we delete a different random subset of nodes and repeat the training process.

The fraction of hidden nodes deleted each time is a hyperparameter called the *dropout rate*. When training is complete, and we actually use the full network, we need to take into account that the full network contains a larger number of hidden nodes than the networks used for training. We therefore need to scale the weight on each outgoing edge from a hidden node by the dropout rate.

Why does dropout reduce overfitting? Several hypotheses have been put forward, but perhaps the most convincing argument is that dropout allows a single neural network to behave effectively as a collection of neural networks. Imagine that we have a collection of neural networks, each with a different network topology. Suppose we trained each network independently using the training data, and used some kind of voting or averaging scheme to create a higher-level model. Such a scheme would perform better than any of the individual networks. The dropout technique simulates this setup without explicitly creating a collection of neural networks.

13.6.3 Early Stopping

In Section 13.6.1 we suggested iterating through training examples (or minibatches) until we reach a local minimum in the loss function. In practice, this approach leads to overfitting. It has been observed that while the loss on the

training set (the *training loss*) decreases through the training process, the loss on the test set (the *test loss*) often behaves differently. The test loss falls during the initial part of the training, and then many hit a minimum and actually increase after a large number of training iterations, even as the training loss keeps falling.

Intuitively, the point at when the test loss starts to increase is the point at which the training process has started learning idiosyncrasies of the training data rather than a generalizable model. A simple approach to avoiding this problem is to stop the training when the test loss stops falling. There is, however, a subtle problem with this approach: we might inadvertently overfit to the test data (rather than to the training data) by stopping training at the point of minimum test loss. Therefore, the test error no longer is a reliable measure of the true performance of the model on hitherto unseen inputs.

The usual solution is to use a third subset of inputs, the validation set, to determine the point at which we stop training. We split the data not just into training and test sets, but into three groups: training, validation, and test. Both the validation and test sets are withheld from the training process. When the loss on the validation set stops decreasing, we stop the training process. Since the test set has played no role at all in the training process, the test error remains a reliable indicator of the true performance of the model.

13.6.4 Dataset Augmentation

The accuracy of most machine-learning models increases when we provide additional training data. Usually, larger training sets also lead to less overfitting. When the actual training data available is limited, we can often create additional synthetic training examples by applying transformations or adding noise.

For example, consider the digit-classification problem we encountered in Example 13.7. It is clear that if we rotate an image corresponding to a digit by a few degrees, it still remains an image of the same digit. We can augment the training data by systematically applying transformations of this kind. One way to think of this process is as a way to encode additional domain knowledge (e.g., a slightly distorted image of a cat is still an image of a cat).

13.7 Summary of Chapter 13

- ✦ *Neural Nets*: A neural net is a collection of perceptrons (nodes), usually organized in layers, where the outputs from one layer provide inputs to the next layers. The first (inout) layer takes external inputs, and the last (output) layer indicates the class of the input. Other layers in the middle are called hidden layers and generally are trained to recognize intermediate concepts needed to determine the output.

- ✦ *Types of Layers*: Many layers are fully connected, meaning that each node in the layer has all the nodes of the previous layer as inputs. Other layers

are pooled, meaning that the nodes of the previous layer are partitioned, and each node of this layer takes as input only the nodes of one block of the partition. Convolutional layers are also used, especially in image processing applications.

- *Convolutional Layers*: Convolutional layers can be viewed as if their nodes were organized into a two-dimensional array of pixels, with each pixel represented by the same collection of nodes. The weights on corresponding nodes from different pixels must be the same, so they are in effect the same node, and we need learn only one set of weights for each family of nodes, one from each pixel.

- *Activation Functions*: The output of a node in a neural net is determined by first taking the weighted sum of its inputs, using the weights that are learned during the process of training the net. An activation function is then applied to this sum. Common activation functions include the sigmoid function, the hyperbolic tangent, softmax, and various forms of linear recified unit functions.

- *Loss Functions*: These measure the difference between the output of the net and the correct output according to the training set. Commonly used loss functions include squared-error loss, Huber loss, classification loss, and cross-entropy loss.

- *Training a Neural Net*: We train a neural net by repeatedly computing the output of the net on training examples and computing the average loss on the training examples. Weights on the nodes are then adjusted by propagating the loss backward through the net, using the backpropagation algorithm.

- *Backpropagation*: By choosing our activation functions and loss functions to be differentiable, we can compute a deriative of the loss function with respect to every weight in the network. Thus, we can determine the direction in which to adjust each weight to reduce the loss. Using the chain rule, these directions can be computed layer-by-layer, from the output to the input.

- *Convolutional Neural Networks*: These typically consist of a large number of convolutional layers, along with pooled layers and fully connected layers. They are well suited to processing images, where the first convolutional layers recognize simple features, such as boundries, and later layers recognize progressively more complex features.

- *Recurrent Neural Networks*: These are designed to recognize sequences, such as sentences (sequences of words). There is one layer for each position in the sequence, and the nodes are divided into families, which each have one node at each layer. The nodes of a family are constrained to have

the same weights, so the training process therefore needs to deal with a relatively small number of weights.

- ✦ *Long Short-Term Memory Networks*: These improve on RNN's by adding a second state vector – the cell state – to enable some information about the sequence to be retained, while most information is forgotten after a while. In addition, we learn gate vectors that control what information is retained from the input, the state, and the output.

- ✦ *Avoiding Overfitting*: There are a number of specialized techniques designed to avoid overfitting a deep network. These include penalizing large weights, randomly dropping some nodes each time we apply a step of gradient descent, and use of a validation set to enable us to stop training when the loss on the validation set bottoms out.

13.8 References for Chapter 13

For information on TensorFlow, see [12]. You can learn about Pytorch at [10] and about Caffe at [1]. The MNIST database is described in [9].

Backpropagation as the way to train deep neural nets is from [11].

The idea of convolutional neural networks begins with [2], which defined convolutional layers and pooling layers. However, it was [13] that introduced the idea of requiring nodes in one convolutional layer to share weights. The application of these networks to character recognition and other important tasks is from [8] and [7]. Also see [6] for the application to ImageNet of CNN's. ImageNet is available from [5].

Recurrent neural networks first appeared in [4]. Long short-term memory is from [3].

1. `http://caffe2.ai`

2. Fukushima, K., "Neocognitron, a self-organizing neural network model for a mechanism of pattern recognition unaffected by shift of position," *Biological Cybernetics* **36**:1 (1980), pp. 193–202.

3. Hochreiter, S. and J. Schmidhuber, "Long Short-Term Memory," *Neural Computation* **9**:8 (1997), pp. 1735-1780.

4. J. J. Hopfield, "Neural networks and physical systems with emergent collective computational abilities," *Proceedings of the National Academy of Sciences* **79**:8 (1982), pp. 2554-2558.

5. `http://www.image-net.org`.

6. Krizhevsky, A, I. Sutskever, and G.E. Hinton, "Image classification with deep convolutional neural networks," *Advances in Neural Information Processing Systems*, pp. 1097–1105, 2012.

7. LeCun, Y. and Y. Bengio, "Convolutional networks for images, speech, and time series," *The Handbook of Brain Theory and Neural Networks* (M. Arbib, ed.) **3361**:10 (1995).

8. LeCun, Y., B. Boser, J.S. Denker, D. Henderson, R.E. Howard, and W. Hubbard, "Backpropagation applied to handwritten zip code recognition," *Neural Computation* **1**:4 (1989) pp. 541–551.

9. LeCun, Y., C. Cortes, and C.J.C. Burges, "The MNIST database of handwritten digits," `http://http://yann.lecun.com/exdb/mnist/`

10. `http://pytorch.org`

11. Rumelhart, D.E., G.E. Hinton, and R.J. Williams, "Learning representations by back-propagating errors, *Nature* **323** (1986), pp. 533–536.

12. `http://www.tensorflow.org`

13. Waibel, A., T. Hanazawa, G.E. Hinton, K. Shikano, and K.J. Lang, "Phoneme recognition using time-delay neural networks," *IEEE Transactions on Acoustics, Speech, and Signal Processing* **37**:3 (1989), pp. 328–339.