Chapter 10

Machine Learning in the Cloud

"Learning is any change in a system that produces a more or less permanent change in its capacity for adapting to its environment."
—Herbert Simon, *The Sciences of the Artificial*

Machine learning has become central to applications of cloud computing. While machine learning is considered part of the field of artificial intelligence, it has roots in statistics and mathematical optimization theory and practice. In recent years it has grown in importance as a number of critical application breakthroughs have taken place. These include human-quality speech recognition [144] and real-time automatic language translation [95], computer vision accurate and fast enough to propel self-driving cars [74], and applications of reinforcement learning that allow machines to master some of the most complex human games, such as Go [234].

What has enabled these breakthroughs has been a convergence of the availability of big data plus algorithmic advances and faster computers that have made it possible to train even deep neural networks. The same technology is now being applied to scientific problems as diverse as predicting protein structure [180], predicting the pharmacological properties of drugs [60], and identifying new materials with desired properties [264].

In this chapter we introduce some of the major machine learning tools that are available in public clouds, as well as toolkits that you can install on a private cloud. We begin with our old friend Spark and its machine learning (ML) package, and then move to Azure ML. We progress from the core "classical" ML tools, including logistic regression, clustering, and random forests, to take a brief look at deep learning and deep learning toolkits. Given our emphasis on Python, the reader may expect us to cover the excellent Python library *scikit-learn*. However, *scikit-learn* is well covered elsewhere [253], and we introduced several of its ML methods in our microservice-based science document classifier example in chapter 7. We describe the same example, but using different technology, later in this chapter.

10.1 Spark Machine Learning Library (MLlib)

Spark MLlib [198], sometimes referred to as Spark ML, provides a set of high-level APIs for creating ML pipelines. It implements four basic concepts.

- **DataFrames** are containers created from Spark RDDs to hold vectors and other structured types in a manner that permits efficient execution [45]. Spark DataFrames are similar to Pandas DataFrames and share some operations. They are distributed objects that are part of the execution graph. You can convert them to Pandas DataFrames to access them in Python.

- **Transformers** are operators that convert one DataFrame to another. Since they are nodes on the execution graph, they are not evaluated until the entire graph is executed.

- **Estimators** encapsulate ML and other algorithms. As we describe in the following, you can use the `fit(...)` method to pass a DataFrame and parameters to a learning algorithm to create a model. The model is now represented as a Transformer.

- A **Pipeline** (usually linear, but can be a directed acyclic graph) links Transformers and Estimators to specify an ML workflow. Pipelines inherit the `fit(...)` method from the contained estimator. Once the estimator is trained, the pipeline is a model and has a `transform(...)` method that can be used to push new cases through the pipeline to make predictions.

Many transformers exist, for example to turn text documents into vectors of real numbers, convert columns of a DataFrame from one form to another, or split DataFrames into subsets. There are also various kinds of estimators, ranging from those that transform vectors by projecting them onto principal component vectors, to n-gram generators that take text documents and return strings of n consecutive words. Classification models include logistic regression, decision tree classifiers, random forests, and naive Bayes. The family of clustering methods includes k-means and latent Dirichlet allocation (LDA). The MLlib online documentation provides much useful material on these and related topics [29] .

10.1.1 Logistic Regression

The example that follows employs a method called **logistic regression** [103], which we introduce here. Suppose we have a set of feature vectors $x_i \in R^n$ for i in $[0, m]$. Associated with each feature vector is a binary outcome y_i. We are interested in the conditional probability $P(y = 1|x)$, which we approximate by a function $p(x)$. Because $p(x)$ is between 0 and 1, it is not expressible as a linear function of x, and thus we cannot use regular linear regression. Instead, we look at the "odds" expression $p(x)/(1 - p(x))$ and guess that its log is linear. That is:

$$ln\left(\frac{p(x)}{1-p(x)}\right) = b_0 + b \cdot x,$$

where the offset b_0 and the vector $b = [b_1, b_2, ...b_n]$ define a hyperplane for linear regression. Solving this expression for $p(x)$ we obtain:

$$p(x) = \frac{1}{1 + e^{-(b_0 + b \cdot x)}}$$

We then predict $y = 1$ if $p(x) > 0$ and zero otherwise. Unfortunately, finding the best b_0 and b is not as easy as in the case of linear regression. However, simple Newton-like iterations converge to good solutions if we have a sample of the feature vectors and known outcomes.

(We note that the **logistic function** $\sigma(t)$ is defined as follows:

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

It is used frequently in machine learning to map a real number into a probability range $[0, 1]$; we use it for this purpose later in this chapter.)

10.1.2 Chicago Restaurant Example

To illustrate the use of Spark MLlib, we apply it to an example from the Azure HDInsight tutorial [195], namely predicting whether restaurants pass or fail health inspections based on the free text of an inspector's comments. We provide two versions of this example in notebook 18: the HDInsight version and a version that runs on any generic Spark deployment. We present the second here.

The data, from the City of Chicago Data Portal `data.cityofchicago.org`, are a set of restaurant heath inspection reports. Each inspection report contains a report number, the name of the owner of the establishment, the name of the establishment, the address, and an outcome (“Pass,” “Fail,” or some alternative such as “out of business” or “not available”). It also contains the (free-text) English comments from the inspector.

We first read the data. If we are using Azure HDInsight, we can load it from blob storage as follows. We use a simple function `csvParse` that takes each line in the CSV file and parses it using Python’s `csv.reader()` function.

```
inspections = spark.sparkContext.textFile( \
    'wasb:///HdiSamples/HdiSamples/FoodInspectionData/
     Food_Inspections1.csv').map(csvParse)
```

The version of the program in notebook 18 uses a slightly reduced dataset. We have eliminated the address fields and some other data that we do not use here.

```
inspections = spark.sparkContext.textFile(
     '/path-to-reduced-data/Food_Inspections1.csv').map(csvParse)
```

We want to create a **training set** from a set of inspection reports that contain outcomes, for use in fitting our logistic regression model. We first convert the RDD containing the data, `inspections`, to create a DataFrame, `df`, with four fields: record id, restaurant name, inspection result, and any recorded violations.

```
schema = StructType([StructField("id", IntegerType(), False),
                     StructField("name", StringType(), False),
                     StructField("results", StringType(), False),
                     StructField("violations", StringType(), True)])

df = spark.createDataFrame(inspections.map(\
             lambda l: (int(l[0]), l[2], l[3], l[4])) , schema)
df.registerTempTable('CountResults')
```

If we want to look at the first few elements, we can apply the `show()` function to return values to the Python environment.

```
df.show(5)
+-------+--------------------+---------------+--------------------+
|     id|                name|        results|          violations|
+-------+--------------------+---------------+--------------------+
|1978294|KENTUCKY FRIED CH...|           Pass|32. FOOD AND NON-...|
|1978279|          SOLO FOODS|Out of Business|                    |
|1978275|SHARKS FISH & CHI...|           Pass|34. FLOORS: CONST...|
|1978268|CARNITAS Y SUPERM...|           Pass|33. FOOD AND NON-...|
|1978261|            WINGSTOP|           Pass|                    |
+-------+--------------------+---------------+--------------------+
only showing top 5 rows
```

Fortunately for the people of Chicago, it seems that the majority of the inspections result in passing grades. We can use some DataFrame operations to count the passing and failing grades.

```
print("Passing = %d"%df[df.results == 'Pass'].count())
print("Failing = %d"%df[df.results == 'Fail'].count())

Passing = 61204
Failing = 20225
```

To train a logistic regression model, we need a DataFrame with a binary label and feature vector for each record. We do not want to use records associated with "out of business" or other special cases, so we map "Pass" and "Pass with conditions" to 1, "Fail" to 0, and all others to -1, which we filter out.

```
def labelForResults(s):
    if s == 'Fail':
        return 0.0
    elif s == 'Pass w/ Conditions' or s == 'Pass':
        return 1.0
    else:
        return -1.0

label = UserDefinedFunction(labelForResults, DoubleType())
labeledData = df.select(label(df.results).alias('label'), \
                                df.violations).where('label >= 0')
```

We now have a DataFrame with two columns, `label` and `violations` and we are ready to create and run the Spark MLlib pipeline that we will use to train our logistic regression model, which we do with the following code.

```
# 1) Define pipeline components
#   a) Tokenize 'violations' and place result in new column 'words'
tokenizer = Tokenizer(inputCol="violations", outputCol="words")
#   b) Hash 'words' to create new column of 'features'
hashingTF = HashingTF(inputCol="words" , outputCol="features")
#   c) Create instance of logistic regression
lr = LogisticRegression(maxIter=10, regParam=0.01)

# 2) Construct pipeline: tokenize, hash, logistic regression
pipeline = Pipeline(stages=[tokenizer, hashingTF, lr])

# 3) Run pipeline to create model
model = pipeline.fit(labeledData)
```

We first (1) define our three pipeline components, which (a) tokenize each `violations` entry (a text string) by reducing it to lower case and splitting it into a vector of words; (b) convert each word vector into a vector in R^n for some n, by applying a hash function to map each word token into a real number value (the new vectors have length equal to the size of the vocabulary, and are stored as sparse vectors); and (c) create an instance of logistic regression. We then (2) put everything into a pipeline and (3) fit the model with our labeled data.

Recall that Spark implements a graph execution model. Here, the pipeline created by the Python program is the graph; this graph is passed to the Spark execution engine by calling the `fit(...)` method on the pipeline. Notice that the `tokenizer` component adds a column `words` to our working DataFrame, and `hashingTF` adds a column `features`; thus, the working DataFrame has columns `ID, name, results, label, violations, words, features` when logistic regression is run. The names are important, as logistic regression looks for columns `label, features`, which it uses for training to build the model. The trainer is iterative; we give it 10 iterations and an algorithm-dependent value of `0.01`.

We can now test the model with a separate test collection as follows.

```
testData = spark.sparkContext.textFile(
               '/data_path/Food_Inspections2.csv')\
             .map(csvParse) \
             .map(lambda l: (int(l[0]), l[2], l[3], l[4]))
testDf = spark.createDataFrame(testData, schema).
    where("results = 'Fail' OR results = 'Pass' OR \
                 results = 'Pass w/ Conditions'")
predictionsDf = model.transform(testDf)
```

The logistic regression model has appended several new columns to the data frame, including one called `prediction`. To test our prediction success rate, we compare the `prediction` column with the `results` column.

```
numSuccesses = predictionsDf.where(\
                """(prediction = 0 AND results = 'Fail') OR \
                   (prediction = 1 AND (results = 'Pass' OR \
                   results = 'Pass w/ Conditions'))""").count()
numInspections = predictionsDf.count()
print("There were %d inspections and there were %d predictions"\
   %(numInspections,numSuccesses))
print("This is a %2.2f sucess rate"\
  %(float(numSuccesses) / float(numInspections) * 100))
```

We see the following output:

```
There were 30694 inspections and there were 27774 predictions
This is a 90.49\% success rate
```

Before getting too excited about this result, we examine other measures of success, such as **precision** and **recall**, that are widely used in ML research. When applied to our ability to predict failure, recall is the probability that we predicted as failing a randomly selected inspection from those with failing grades. As detailed in notebook 18, we find that our recall probability is only 67%. Our ability to predict failure is thus well below our ability to predict passing. The reason may be that other factors involved with failure are not reflected in the report.

10.2 Azure Machine Learning Workspace

Azure Machine Learning is a cloud portal for designing and training machine learning cloud services. It is based on a drag-and-drop component composition model, in which you build a solution to a machine learning problem by dragging parts of the solution from a pallet of tools and connecting them together into a workflow graph. You then train the solution with your data. When you are satisfied with the results, you can ask Azure to convert your graph into a running web service using the model you trained. In this sense Azure ML provides customized machine learning as an on-demand service. This is another example of serverless computation. It does not require you to deploy and manage your own VMs; the infrastructure is deployed as you need it. If your web service needs to scale up because of demand, Azure scales the underlying resources automatically.

To illustrate how Azure ML works, we return to an example that we first considered in chapter 7. Our goal is to train a system to classify scientific papers, based on their abstracts, into one of five categories: physics, math, computer science, biology, or finance. As training data we take a relatively small sample of abstracts from the arXiv online library `arxiv.org`. Each sample consists of a triple: a classification from arXiv, the paper title, and the abstract. For example, the following is the record for a 2015 paper in physics [83].

```
[ 'Physics',
'A Fast Direct Sampling Algorithm for Equilateral Closed Polygons. (arXiv:1510.02466v1 [cond-
mat.stat-mech])',
'Sampling equilateral closed polygons is of interest in the statistical study of ring polymers. Over
the past 30 years, previous authors have proposed a variety of simple Markov chain algorithms
(but have not been able to show that they converge to the correct probability distribution) and
complicated direct samplers (which require extended-precision arithmetic to evaluate numerically
unstable polynomials). We present a simple direct sampler which is fast and numerically stable.'
]
```

This example also illustrates one of the challenges of the classification problem: science has become wonderfully multidisciplinary. The topic given for this sample paper in arXiv is "condensed matter," a subject in physics. Of the four authors, however, two are in mathematics institutes and two are from physics departments, and the abstract refers to algorithms that are typically part of computer science. A human reader might reasonably consider the abstract to be describing a topic in mathematics or computer science. (In fact, multidisciplinary physics papers were so numerous in our dataset that we removed them in the experiment below.)

Let us start with a solution in Azure ML based on a multiclass version of the logistic regression algorithm. Figure 10.1 shows the graph of tasks. To understand this workflow, start at the top, which is where the data source comes into the picture. Here we take the data from Azure blob storage, where we have placed a large subset of our arXiv samples in a CSV file. Clicking the `Import Data` box opens the window that allows us to identify the URL for the input file.

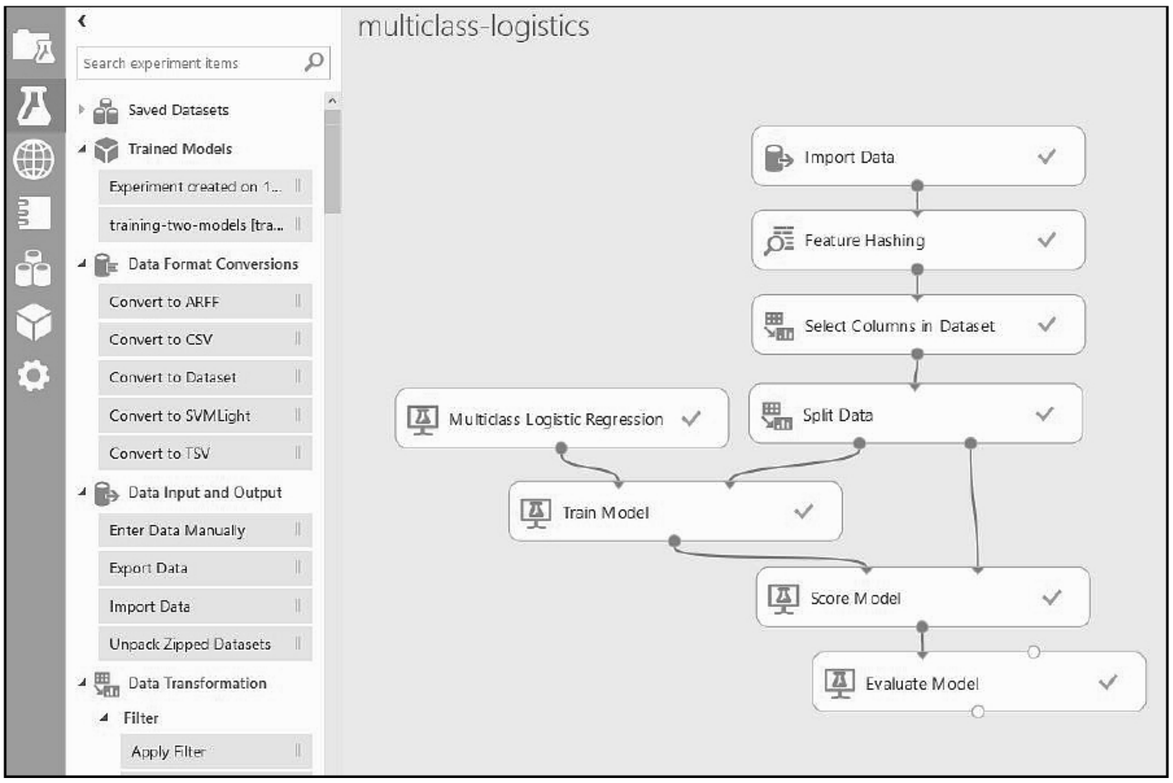


Figure 10.1: Azure ML graph used to train a multiclass logistic regression model.

The second box down, **Feature Hashing**, builds a vectorizer based on the vocabulary in the document collection. This version comes from the Vowpal Wabbit library. Its role is to convert each document into a numerical vector corresponding to the key words and phrases in the document collection. This numeric representation is essential for the actual ML phase. To create the vector, we tell the feature hasher to look only at the abstract text. What happens in the output is that the vector of numeric values for the abstract text is appended to the tuple for each document. Our tuple now has a large number of columns: class, title, abstract, and `vector[0], ..., vector[n-1]`, where n is the number of features. To configure the algorithm, we select two parameters, a hashing bin size and an n-gram length.

Before sending the example to ML training, we remove the English text of the abstract and the title, leaving only the class and the vector for each document. We accomplish this with a `Select Columns in Dataset`. Next we split the data into two subsets: a training subset and a test subset. (We specify that `Split Data` should use 75% of the data for training and the rest for testing.)

Azure ML provides a good number of the standard ML modules. Each such module has various parameters that can be selected to tune the method. For all the experiments described here, we just used the default parameter settings. The `Train Model` component accepts as one input a binding to an ML method (recall this is not a dataflow graph); the other input is the projected training data. The output of the Train Model task is not data per se but a trained model that may also be saved for later use. We can now use this trained model to classify our test data. To this end, we use the `Score Model` component, which appends another new column to our table, Scored Label, providing the classification predicted by the trained model for each row.

To see how well we did, we use the `Evaluate Model` component, which computes a confusion matrix. Each row of the matrix tells us how the documents in that class were classified. Table 10.1 shows the confusion matrix for this experiment. Observe, for example, that a fair number of biology papers are classified as math. We attribute this to the fact that most biology papers in the archive are related to quantitative methods, and thus contain a fair amount of mathematics. To access the confusion matrix, or for that matter the output of any stage in the graph, click on the output port (the small circle) on the corresponding box to access a menu. Selecting *visualize* in that menu brings up useful information.

Table 10.1: Confusion matrix with only math, computer science, biology, and finance.

	bio	**compsci**	**finance**	**math**
bio	51.3	19.9	4.74	24.1
compsci	10.5	57.7	4.32	27.5
finance	6.45	17.2	50.4	25.8
math	6.45l	16.0	5.5	72

Now that we have trained the model, we can click the `Set Up Web Service` button (not visible, but at the bottom of the page) to turn the model into a web service. The Azure ML portal rearranges the graph by eliminating the split-train-test parts and leaves just the feature hashing, column selection, and the scoring based on the trained model. Two new nodes have been added: a web service input and a web service output. The result, with one exception, is shown in figure 10.2. The exception is that we have added a new `Select Columns` node so that we can remove the vectorized document columns from the output of the web service. We retain the original class, the predicted class, and the probabilities computed for the document being in a class.

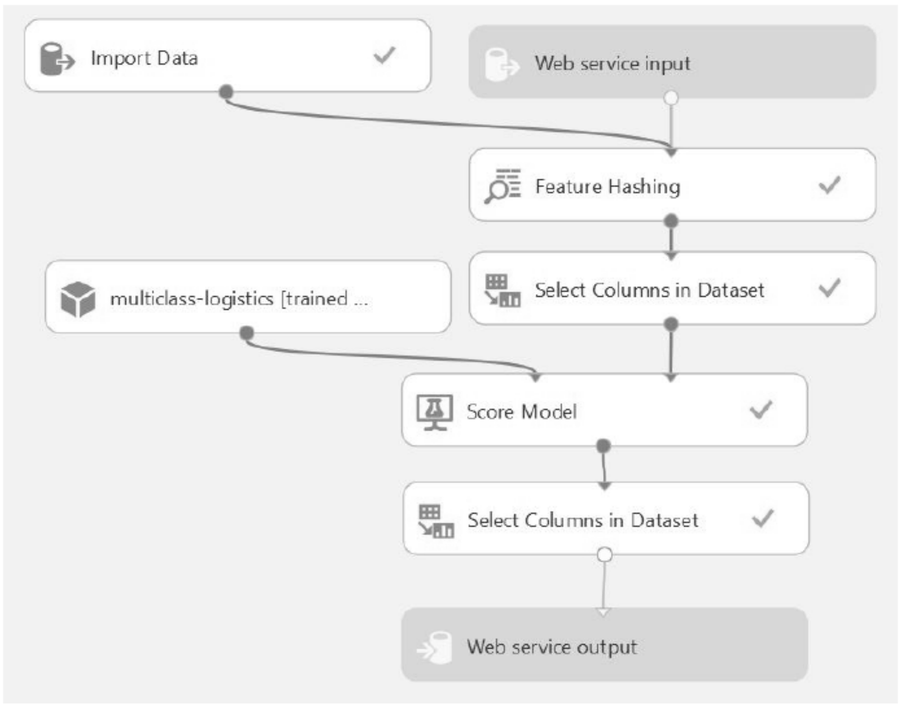

Import Data
Web service input
Feature Hashing
multiclass-logistics [trained ...
Select Columns in Dataset
Score Model
Select Columns in Dataset
Web service output

Figure 10.2: Web service graph generated by Azure ML, with an additional node to remove the vectorized document.

You can now try additional ML classifier algorithms simply by replacing the box *Multiclass Logistic Regression* with, for example, *Multiclass Neural Network* or *Random forest classifier*. Or, you can incorporate all three methods into a single web service that uses a majority vote ("consensus") method to pick the best classification for each document. As shown in figure 10.3, the construction of this consensus method is straightforward: we simply edit the web service graph for the multiclass logistic regression to add the trained models for the other two methods and then call a Python script to tie the three results together.

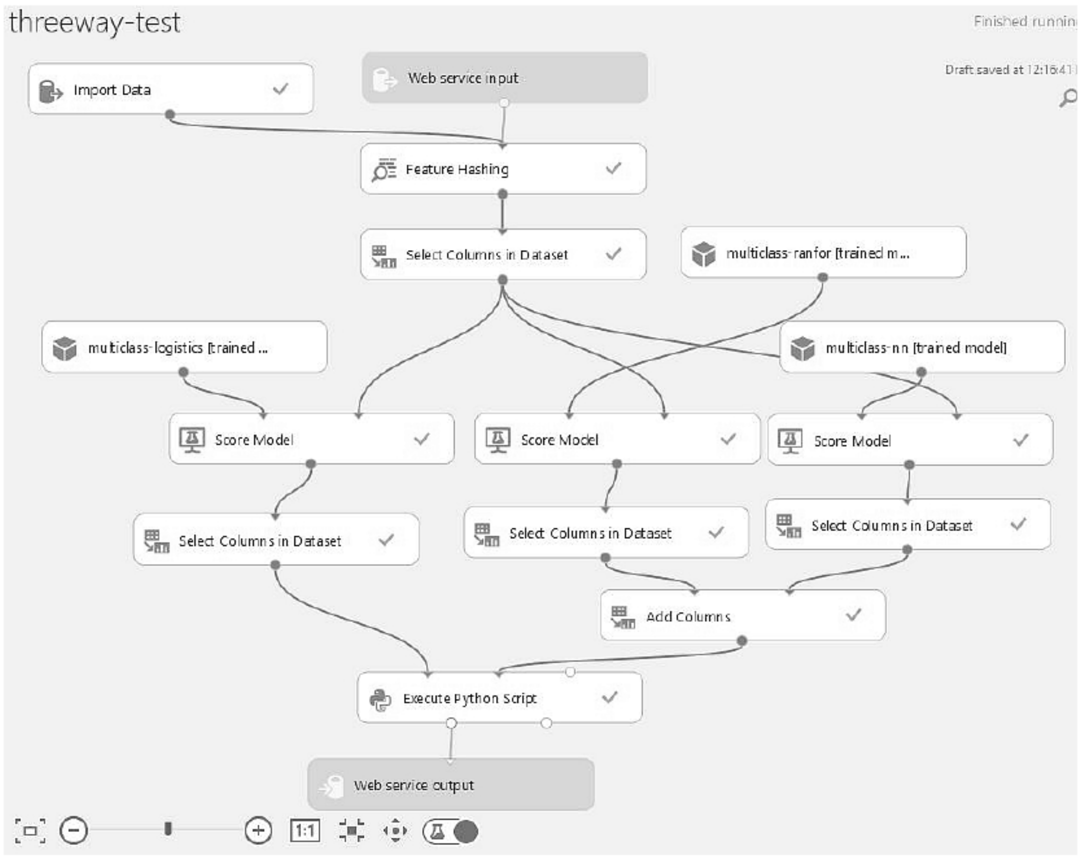

threeway-test
Finished running
Draft saved at 12:16:41
Import Data
Web service input
Feature Hashing
Select Columns in Dataset
multiclass-ranfor [trained m...
multiclass-logistics [trained ...
multiclass-nn [trained model]
Score Model
Score Model
Score Model
Select Columns in Dataset
Select Columns in Dataset
Select Columns in Dataset
Add Columns
Execute Python Script
Web service output
1:1

Figure 10.3: Modified web service graph based on a consensus model, showing three models and a Python script component, used to determine the consensus.

The Python script can simply compare the outputs from the three classifiers. If any two agree, then it selects that classification as a first choice and the classification that does not agree as a second choice. The results for the first choice, shown in table 10.2, are only modestly better than in the logistic regression case, but if we consider both the first and second choices, we reach 65% for biology, 72% for computer science, 60% for finance, and 88% for math. Notebook 19 contains this Python script as well as the code used to test and invoke the services and to compute the confusion matrices.

Table 10.2: Confusion matrix for the three-way classifier.

	bio	**compsci**	**finance**	**math**
bio	50.3	20.9	0.94	27.8
compsci	4.9	62.7	1.54	30.9
finance	5.6	9.9	47.8	36.6
math	3.91	13.5	2.39	80.3

10.3 Amazon Machine Learning Platform

The Amazon platform provides an impressive array of ML services. Each is designed to allow developers to integrate cloud ML into mobile and other applications. Three of the four are based on the remarkable progress that has been enabled by the deep learning techniques that we discuss in more detail in the next section.

Amazon Lex allows users to incorporate voice input into applications. This service is an extension of Amazon's Echo product, a small networked device with a speaker and a microphone to which you can pose questions about the weather and make requests to schedule events, play music, and report on the latest news. With Lex as a service, you can build specialized tools that allow a specific voice command to Echo to launch an Amazon lambda function to execute an application in the cloud. For example, NASA has built a replica of the NASA Mars rover that can be controlled by voice commands, and has integrated Echo into several other applications around their labs [189].

Amazon Polly is the opposite of Lex: it turns text into speech. It can speak in 27 languages with a variety of voices. Using the Speech Synthesis Markup Language, you can carefully control pronunciation and other aspects of intonation. Together with Lex, Polly makes a first step toward conversational computing. Polly and Lex do not do real-time, voice-to-voice language translation the way Skype does, but together they provide a great platform to deliver such a service.

Amazon Rekognition is at the cutting edge of deep learning applications. It takes an image as input and returns a textual description of the items that it sees in that image. For example, given an image of a scene with people, cars, bicycles, and animals, Rekognition returns a list of those items, with a measure of certainty associated with each. The service is trained with many thousands of captioned images in a manner not unlike the way natural language translation systems are trained: it considers a million images containing a cat, each with an associated caption that mentions "cat," and a model association is formed. Rekognition can also perform detailed facial analysis and comparisons.

The **Amazon Machine Learning** service, like Azure ML, can be used to create a predictive model based on training data that you provide. However, it requires much less understanding of ML concepts than does Azure ML. The Amazon Machine Learning dashboard presents the list of experiments, models, and data sources from your previous Amazon Machine Learning work. From the dashboard you can define data sources and ML models, create evaluations, and run batch predictions.

Using Amazon Machine Learning is easy. For example, we used it to build a predictive model from our collection of scientific articles in under an hour. One reason that it is so easy to use is that the options are simple. You can build only three types of models—regression, binary classification, or multiclass classification—and in each case, Amazon Machine Learning provides a single model. In the case of multiclass classification, it is multinomial logistic regression with a stochastic gradient descent optimizer. And it works well. Using the same test and training data as earlier, we obtained the results shown in table 10.3. Although the trained Amazon Machine Learning classifier failed to recognize any computational finance papers, it beat our other classifiers in the other categories. Amazon Labs has additional excellent examples [44].

Table 10.3: Confusion matrix for the science document classifier using Amazon ML.

	bio	**compsci**	**finance**	**math**
bio	62.0	19.9	0.0	18.0
compsci	3.8	78.6	0.0	17.8
finance	6.8	2.5	0.0	6.7
math	3.5	11.9	0.0	84.6

Amazon Machine Learning is also fully accessible from the Amazon REST interface. For example, you can create a ML model using Python as follows.

```
response = client.create_ml_model(
    MLModelId='string',
    MLModelName='string',
    MLModelType='REGRESSION'|'BINARY'|'MULTICLASS',
    Parameters={
        'string': 'string'
    },
    TrainingDataSourceId='string',
    Recipe='string',
    RecipeUri='string'
)
```

The parameter `ModelID` is a required, user-supplied, unique identifier; other parameters specify, for example, the maximum allowed size of the model, the maximum number of passes over the data in building the model, and a flag to tell the learners to shuffle the data. The training data source identifier is a data recipe or URI for a recipe in S3. A recipe is a JSON-like document that describes how to transform the datasets for input while building the model. (Consult the Amazon Machine Learning documents for more details.) For our science document example, we used the default recipe generated by the portal.

10.4 Deep Learning: A Shallow Introduction

The use of **artificial neural networks** for machine learning tasks has been common for at least 40 years. Mathematically, a neural network is a method of approximating a function. For example, consider the function that takes an image of a car as input and produces the name of a manufacturer as output. Or, consider the function that takes the text of a scientific abstract and outputs the most likely scientific discipline to which it belongs. In order to be a computational entity, our function and its approximation need a numerical representation. For example, suppose our function takes as input a vector of three real numbers and returns a vector of length two. Figure 10.4 is a diagram of a neural net with one hidden layer representing such a function.

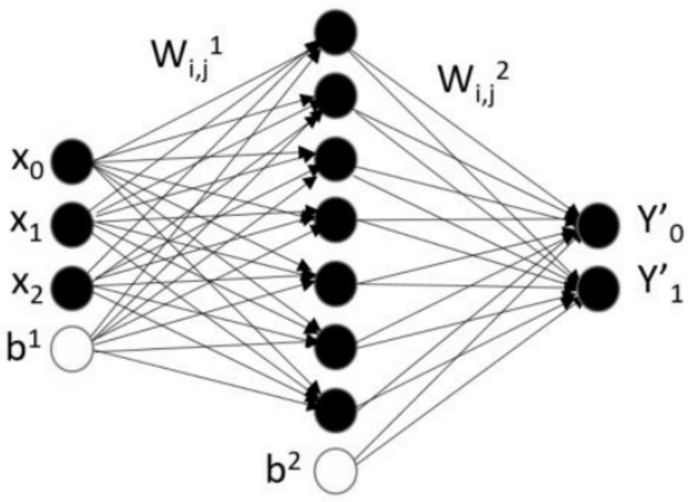


Figure 10.4: Neural network with three inputs, one hidden layer, and two outputs.

In this schematic representation, the lines represent numerical weights connecting the inputs to the n interior neurons, and the terms b are offsets. Mathematically the function is given by the following equations.

$$a_j = f(\sum_{i=0}^{2} x_i W^1_{i,j} + b_j) \qquad for \quad j = 1, n$$

$$y'_j = f'(\sum_{i=0}^{n} a_i W^2_{i,j} + b^2_j) \qquad for \quad j = 0, 1$$

The functions f and f' are called the **activation functions** for the neurons. Two commonly used activation functions are the logistic function $\sigma(t)$ that we introduced at the beginning of this chapter and the rectified linear function:

$$\text{relu}(x) = \max(0, x).$$

Another common case is the hyperbolic tangent function

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

An advantage of $\sigma(x)$ and $\tanh(x)$ is that they map values in the range $(-\infty, \infty)$ to the range $(0, 1)$, which corresponds to the idea that a neuron is either on or off (not-fired or fired). When the function represents the probability that an input corresponds to one of the outputs, we use a version of the logistic function that ensures that the probabilities all sum to one.

$$\text{softmax}(x)_j = \frac{1}{1 + \sum_{k \neq j} e^{x_k - x_j}}$$

This formulation is commonly used in multiclass classification, including in several of the examples we have studied earlier.

The trick to making the neural net truly approximate our desired function is picking the right values for the weights. There is no closed form solution for finding the best weights, but if we have a large number of labeled examples (x^i, y^i), we can try to minimize the cost function.

$$C(x^i, y^i) = \sum ||y^i - y'(x^i)||$$

The standard approach is to use a variation of gradient descent, specifically **back propagation**. We do not provide details on this algorithm here but instead refer you to two outstanding mathematical treatments of deep learning [143, 210].

10.4.1 Deep Networks

An interesting property of neural networks is that we can stack them in layers as illustrated in figure 10.5 on the next page. Furthermore, using the deep learning toolkits that we discuss in the remainder of this chapter, we can construct such networks with just a few lines of code. In this chapter, we introduce three deep learning toolkits. We first illustrate how each can be used to define some of the standard deep networks and then, in later sections, describe how to deploy and apply them in the cloud.

MXNet `github.com/dmlc/mxnet` is the first deep learning toolkit that we consider. Using MXNet, the network in figure 10.5 would look as follows.

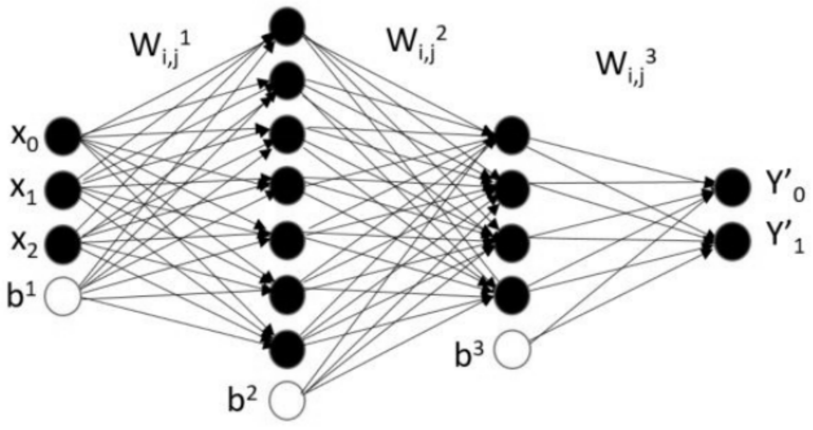


Figure 10.5: Neural network with three inputs, two hidden layers, and two outputs.

```
data = mx.symbol.Variable('x')
layr1= mx.symbol.FullyConnected(data=data,name='W1',num_hidden=7)
act1 = mx.symbol.Activation(data=layr1,name='relu1',act_type="relu")
layr2= mx.symbol.FullyConnected(data=act1,name='W2',num_hidden=4)
act2 = mx.symbol.Activation(data=layr2,name='relu2',act_type="relu")
layr3= mx.symbol.FullyConnected(data=act2, name='W3',num_hidden=2)
Y  = mx.symbol.SoftmaxOutput(data = layr3,name='softmax')
```

The code creates a stack of fully connected networks and activations that exactly describe our diagram. In the following section we return to the code needed to train and test this network.

The term **deep neural network** generally refers to networks with many layers. Several special case networks also have proved to be of great value for certain types of input data.

10.4.2 Convolutional Neural Networks

Data with a regular spatial geometry such as images or one-dimensional streams are often analyzed with a special class of network called a **convolutional neural network** or CNN. To explain CNNs, we use our second example toolkit, **TensorFlow** `tensorflow.org`, which was open sourced by Google in 2016. We consider a classic example that appears in many tutorials and is well covered in that provided with TensorFlow `tensorflow.org/tutorials`.

Suppose you have thousands of 28×28 black and white images of handwritten digits and you want to build a system that can identify each. Images are strings of bits, but they also have a lot of local two-dimensional structure such as edges and holes. In order to find these patterns, we examine each of the many 5×5 windows in each image individually. To do so, we train the system to build a 5×5 template array $W1$ and a scalar offset b that together can be used to reduce each 5×5 window to a point in a new array `conv` by the following formula.

$$\mathtt{conv}_{p,q} = \sum_{i,k=-2}^{2} W_{i,k}\mathtt{image}_{p-i,q-k} + b$$

(The image is padded near the boundary points in the formula above so that none of the indices are out of bounds.) We next modify the `conv` array by applying the `relu` function to each x in the `conv` array so that it has no negative values. The final step, *max pooling*, simply computes the maximum value in each 2×2 block and assigns it to a smaller 14×14 array. The most interesting part of the convolutional network is that we do not use one 5×5 $W1$ template but 32 of them in parallel, producing 32 14×14 results, `pool1`, as illustrated in figure 10.6.

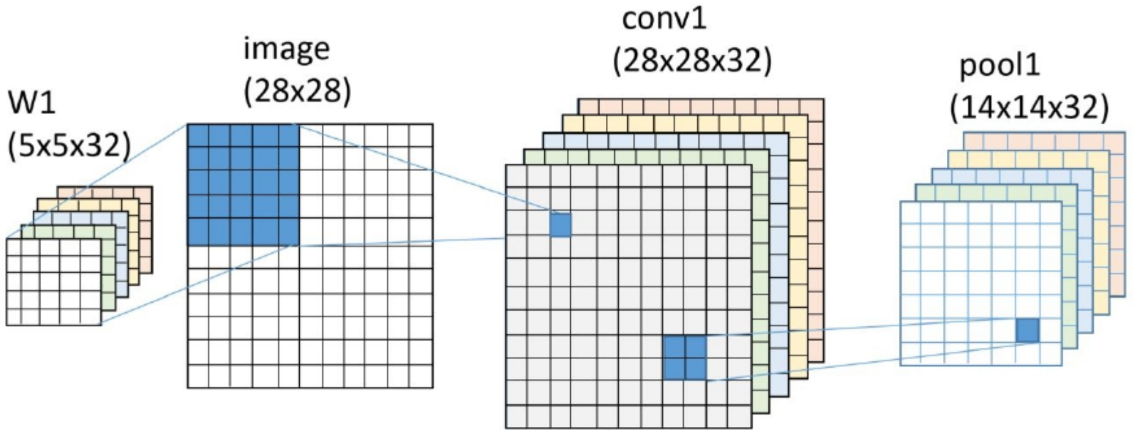


Figure 10.6: Schematic of how a convolutional neural net processes an image.

When the network is fully trained, each of the 32 5×5 templates in $W1$ is somehow different, and each selects for a different set of features in the original image. One can think of the resulting stack of 32 14×14 arrays (called `pool1`) as a type of transform of the original image, which works much like a Fourier transform to separate a signal in space and time and transform it into frequency space. This is not what is going on here; but if you are familiar with these transforms, the analogy may be helpful.

We next apply a second convolutional layer to `pool1`, but this time we apply 64 sets of 5×5 filters to each of the 32 `pool1` layers and sum the results to obtain 64 new 14×14 arrays. We then reduce these with max pooling to 64 7×7 arrays called `pool2`. From there we use a dense "all-to-all" layer and finally reduce it to 10 values, each representing the likelihood that the image corresponds to a digit 0 to 9. The TensorFlow tutorial defines two ways to build and train this network; figure 10.7 is from the community-contributed library called *layers*.

```
input_layer = tf.reshape(features, [-1, 28, 28, 1])

conv1 = tf.layers.conv2d(
      inputs=input_layer,
      filters=32,
      kernel_size=[5, 5],
      padding="same",
      activation=tf.nn.relu)

pool1 = tf.layers.max_pooling2d(inputs=conv1, \
                              pool_size=[2, 2], strides=2)

conv2 = tf.layers.conv2d(
      inputs=pool1,
      filters=64,
      kernel_size=[5, 5],
      padding="same",
      activation=tf.nn.relu)

pool2 = tf.layers.max_pooling2d(inputs=conv2, \
                             pool_size=[2, 2], strides=2)
pool2_flat = tf.reshape(pool2, [-1, 7 * 7 * 64])

dense = tf.layers.dense(inputs=pool2_flat, \
                         units=1024, activation=tf.nn.relu)

logits = tf.layers.dense(inputs=dense, units=10)
```

Figure 10.7: TensorFlow two convolutional layer digit recognition network

As you can see, these operators explicitly describe the features of our CNNs. The full program is in the TensorFlow examples tutorial layers directory in file `cnn_mnist.py`. If you would rather see a version of the same program using lower-level TensorFlow operators, you can find an excellent Jupyter notebook version in the Udacity deep learning course material [3]. CNNs have many applications in image analysis. One excellent science example is the solution to the Kaggle Galaxy Zoo Challenge, which asked participants to predict how Galaxy Zoo users would classify images of galaxies from the Sloan Digital Sky Survey. Dieleman [111] describes the solution, which uses four convolutional layers and three dense layers.

10.4.3 Recurrent Neural Networks.

Recurrent neural networks (RNNs) are widely used in language modeling problems, such as predicting the next word to be typed when texting or in automatic translation systems. RNNs can learn from sequences that have repeating patterns. For example, they can learn to "compose" text in the style of Shakespeare [168] or even music in the style of Bach [183]. They have also been used to study forest fire area coverage [94] and cycles of drought in California [178].

The input to the RNN is a word or signal, along with the state of the system based on words or signals seen so far; the output is a predicted list and a new state of the system, as shown in figure 10.8.

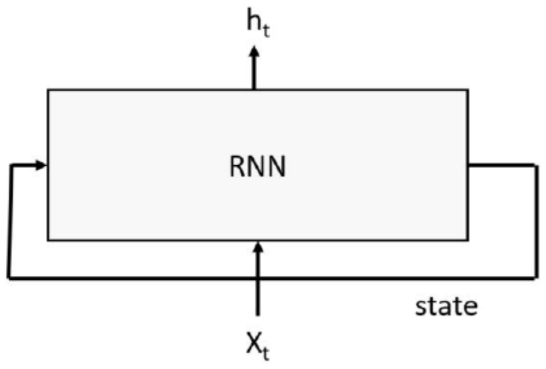


Figure 10.8: Basic RNN with input stream x and output stream h.

Many variations of the basic RNN exist. One challenge for RNNs is ensuring that the state tensors retain enough long-term memory of the sequence so that patterns are remembered. Several approaches have been used for this purpose. One popular method is the Long-Short Term Memory (LSTM) version that is defined by the following equations, where the input sequence is x, the output is h, and the state vector is the pair $[c, h]$.

$$
\begin{gathered}
i_t = \sigma(W^{(xi)}x_t + W^{(hi)}h_{t-1} + W^{(ci)}c_{t-1} + b^{(i)}) \\
f_t = \sigma(W^{(xf)}x_t + W^{(hf)}h_{t-1} + W^{(cf)}c_{t-1} + b^{(f)}) \\
c_t = f_t \cdot c_{t-1} + i_t \cdot \tanh(W^{(xc)}x_t + W^{(hc)}h_{t-1} + b^{(c)}) \\
o_t = \sigma(W^{(xo)}x_t + W^{(ho)}h_{t-1} + W^{(co)}c_t + b^{(o)}) \\
h_t = o_t \cdot \tanh(c_t)
\end{gathered}
$$

Olah provides an excellent explanation of how RNNs work [213]. We adapt one of his illustrations to show in figure 10.9 on the next page how information flows in our network. Here we use the vector concatenation notation `concat` as follows to compose the various W matrices and thus obtain a more compact representation of the equations.

$$\sigma(\text{concat}(x, h, c)) = \sigma(W[x, h, c] + b) = \sigma(W^{(x)}x + W^{(h)}h + W^{(c)}c + b).$$

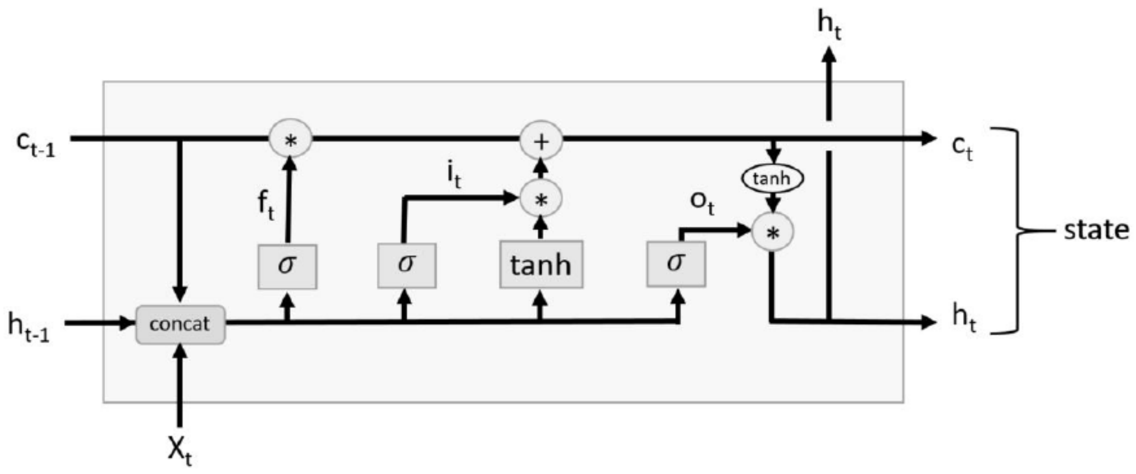


Figure 10.9: LSTM information flow, adapted from Olah [213] to fit equations in the text.

We use a third toolkit, the **Microsoft Cognitive Toolkit** (formerly known as the Computational Network Toolkit **CNTK**), to illustrate the application of RNNs. Specfically, we consider a *sequence-to-sequence* LSTM from the Microsoft Cognitive Toolkit distribution that is trained with input from financial news items such as “shorter maturities are considered a sign of rising rates because portfolio managers can capture higher rates sooner” and “j. p. <unk> vice chairman of grace and co. which holds an interest in this company was elected a director.” Following training, this network can be used to generate other sentences with a similar structure.

To illustrate what this network can do, we saved the trained W and b arrays and two other arrays that together define its structure. We then loaded these arrays into the Python version of the RNN shown on the next page, which we created by transcribing the equations above.

```python
def rnn(word, old_h, old_c):
      Xvec = getvec(word, E)
      i = Sigmoid(np.matmul(WXI, Xvec) +
                  np.matmul(WHI, old_h) + WCI*old_c + bI)
      f = Sigmoid(np.matmul(WXF, Xvec) +
                  np.matmul(WHF, old_h) + WCF*old_c + bF)
      c = f*old_c + i*(np.tanh(np.matmul(WXC, Xvec) +
                               np.matmul(WHC, old_h) + bC))
      o = Sigmoid(np.matmul(WXO, Xvec)+
                  np.matmul(WHO, old_h)+ (WCO * c)+ bO)
      h = o * np.tanh(c)
      # Extract ordered list of five best possible next words
      q = h.copy()
      q.shape = (1, 200)
      output = np.matmul(q, W2)
      outlist = getwordsfromoutput(output)
      return h, c, outlist
```

As you can see, this code is almost a literal translation of the equations. The only difference is that the code has as input a text string for the input word, while the equations take a vector encoding of the word as input. The RNN training generated the encoding matrix E, which has the nice property that the ith column of the matrix corresponds to the word in the ith position in the vocabulary list. The function `getvec(word, E)` takes the embedding tensor E, looks up the position of the word in the vocabulary list, and returns the column vector of E that corresponds to that word. The output of one pass through the LSTM cell is the vector h. This is a compact representation of the words likely to follow the input text to this point. To convert this back into "vocabulary" space, we multiply it by another trained vector $W2$. The size of our vocabulary is 10,000, and the vector output is that length. The ith element of the output represents the relative likelihood that the ith word is the next word to follow the input so far. Our addition, `Getwordsfromoutput`, simply returns the top five candidate words, in order of likelihood.

To see whether this LSTM is truly a recurrent network, we provide the network with a starting word, let it suggest the next word, and repeat this process to construct a "sentence." In the code on the next page, we randomly pick one of the top three suggested by the network as the next word.

```
c = np.zeros(shape = (200, 1))
h = np.zeros(shape = (200, 1))
output = np.zeros(shape = (10000, 1))
word = 'my'
sentence= word
for _ in range(40):
    h, c, outlist = rnn(word, h, c)
    word = outlist[randint(0,3)]
    sentence = sentence + " " +word
print(sentence+".")
```

Testing this code with the start word "my" produced the following output.

```
my new rules which would create an interest position here unless
there should prove signs of such things too quickly although the
market could be done better toward paying further volatility where
it would pay cash around again if everybody can.
```

Using "the" as our start word produced the following.

```
the company reported third-quarter results reflecting a number
compared between N barrels including pretax operating loss
from a month following fiscal month ending july earlier
compared slightly higher while six-month cds increased
sharply tuesday after an after-tax loss reflecting a strong.
```

This RNN is hallucinating financial news. The sentences are obviously nonsense, but they are excellent examples of mimicry of the patterns that the network was trained with. The sentences end rather abruptly because of the 40-word limit in the code. If you let it go, it runs until the state vector for the sentence seems to break down. Try this yourself. To make it easy to play with this example, we have put the code in notebook 20 along with the 50 MB model data.

10.5 Amazon MXNet Virtual Machine Image

MXNet [92] `github.com/dmlc/mxnet` is an open source library for distributed parallel machine learning. It was originally developed at Carnegie Mellon, the University of Washington, and Stanford. MXNet can be programmed with Python, Julia, R, Go, Matlab, or C++ and runs on many different platforms, including clusters and GPUs. It is also now the deep learning framework of choice for Amazon [256]. Amazon has also released the **Amazon Deep Learning AMI** [13], which includes not only MXNet but also CNTK and TensorFlow, plus other good toolkits that we have not discussed here, including Caffe, Theano, and Torch. Jupyter and the Anaconda tools are there, too.

Configuring the Amazon AMI to use Jupyter is easy. Go to the Amazon Marketplace on the EC2 portal and search for "deep learning"; you will find the Deep Learning AMI. Then select the server type. This AMI is tuned for the `p2.16xlarge` instances (64 virtual cores plus 16 NVIDIA K80 GPUs). This is an expensive option. If you simply want to experiment, it works well with a no-GPU eight-core option such as `m4.2xlarge`. When the VM comes up, log in with `ssh`, and configure Jupyter for remote access as follows.

```
>cd .jupyter
>openssl req -x509 -nodes -days 365 -newkey rsa:1024 \
  -keyout mykey.key -out mycert.pem
>ipython
[1]: from notebook.auth import passwd
[2]: passwd()
Enter password:
Verify password:
Out[2]: 'sha1:---- long string -----------'
```

Remember your password, and copy the long `sha1` string. Next create the file `.jupyter/jupyter_notebook_config.py`, and add the following lines.

```
c = get_config()
c.NotebookApp.password = u'sha1:----long string -----------'
c.NotebookApp.ip = '*'
c.NotebookApp.port = 8888
c.NotebookApp.open_browser = False
```

Now invoke Jupyter as follows.

```
jupyter notebook --certfile=.jupyter/mycert.pem \
                 --keyfile=.jupyter/mykey.key
```

Then, go to `https://`*`ipaddress`*`:8888` in your browser, where *ipaddress* is the external IP address of your virtual machine. Once you have accessed Jupyter within your browser, visit `src/mxnet/example/notebooks` to run MXNet.

Many excellent MXNet tutorials are available in addition to those in the AMI notebooks file, for example on the MXNet community site. To illustrate MXNet's use, we examine a particularly deep neural network trained on a dataset with 10 million images. Resnet-152 [152] is a network with 152 convolutional layers based on a concept called deep residual learning, which solved an important problem with training deep networks called the vanishing gradient problem. Put simply, it states that training by gradient descent methods fails for deep networks because as the network grows in depth, the computable gradients become numerically so small that there is no stable descent direction. Deep residual training approaches the problem differently by adding an identity mapping from one layer to the next so that one solves a residual problem rather than the original. It turns out that the residual is much easier for stochastic gradient descent to solve.

Resnets of various sizes have been built with each of the toolkits mentioned here. Here we describe one that is part of the MXNet tutorials [41]. (Notebook 21 provides a Jupyter version.) The network has 150 convolutional layers with a softmax output on a fully connected layer, with 11,221 nodes representing the 11,221 image labels that it is trained to recognize. The input is a 3×224×224 RGB format image that is loaded into a batch normalization function and then sent to the first convolutional layer. The example first fetches the archived data for the model. There are three main files.

- `resent-152-symbol.json`, a complete description of the network as a large json file
- `resnet-152-0000.params`, a binary file containing all parameters for the trained model
- `synset.txt`, a text file containing the 1,121 image labels, one per line

You can then load the pretrained model data, build a model from the data, and apply the model to a JPEG image. (The image must be converted to 3×244×244 RGB format: see notebook 21.)

```
import mxnet as mx
# 1) Load the pretrained model data
with open('full-synset.txt','r') as f:
    synsets = [l.rstrip() for l in f]
sym,arg_params,aux_params=
    mx.model.load_checkpoint('full-resnet-152',0)
# 2) Build a model from the data
mod = mx.mod.Module(symbol=sym, context=mx.gpu())
mod.bind(for_training=False, data_shapes=[('data', (1,3,224,224))])
mod.set_params(arg_params, aux_params)
# 3) Send JPEG image to network for prediction
mod.forward(Batch([mx.nd.array(img)]))
prob = mod.get_outputs()[0].asnumpy()
prob = np.squeeze(prob)
a = np.argsort(prob)[::-1]
for i in a[0:5]:
    print('probability=%f, class=%s' %(prob[i], synsets[i]))
```

You will find that the accuracy of the network in recognizing images is excellent. Below we selected four images in figure 10.10 from the Bing image pages with a focus on biology. You can see from the results in table 10.4 that the top choice of the network was correct for each example, although the confidence was less high for yeast and seahorse. These results clearly illustrate the potential for automatic image recognition in aiding scientific tasks.

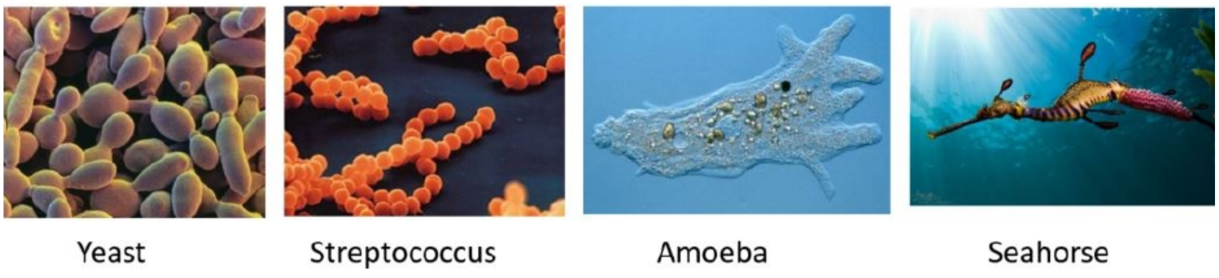


Figure 10.10: Three sample images that we have fed to the MXNet Resnet-152 network.

Table 10.4: Identification of images in figure 10.10 along with estimated probabilities.

Yeast	**Streptococcus**	**Amoeba**	**Seahorse**
p=0.26, yeast	p=0.75, streptococcus, streptococci, strep	p=0.70, ameba, amoeba	p=0.33, seahorse
p=0.21, microorganism	p=0.08, staphylococcus, staph	p=0.15, microorganism	p=0.12, marine animal, marine creature, sea animal
p=0.21, cell	p=0.06, yeast	p=0.05, ciliate, ciliated protozoan, ciliophoran	p=0.12, benthos
p=0.06, streptococcus, strep	p=0.04, microorganism, micro-organism	p=0.04, paramecium, paramecia	p=0.05, invertebrate
p=0.05, eukaryote, eucaryote	p=0.01, cytomegalovirus, CMV	p=0.03, photomicrograph	p=0.04, pipefish, needlefish

10.6 Google TensorFlow in the Cloud

Google's **TensorFlow** is a frequently discussed and used deep learning toolkit. If you have installed the Amazon Deep Learning AMI, then you already have TensorFlow installed, and you can begin experimenting right away. While we have already introduced TensorFlow when discussing convolutional neural networks, we need to look at some core concepts before we dive more deeply.

Let us start with tensors, which are generalizations of arrays to dimensions beyond 1 and 2. In TensorFlow, tensors are created and stored in container objects

Table 10.5: (Fake) graduate school admission data.

GRE	GPA	Rank	Decision
800	4.0	4	0
339	2.0	1	1
750	3.9	1	1
800	4.0	2	0

that are one of three types: variables, placeholders, and constants. To illustrate the use of TensorFlow, we build a logistic regression model of some (fake) graduate school admissions decisions. Our data, shown in table 10.5, consist of a GRE exam score in the (pre-2012) range of 0 to 800; a grade point average (GPA) in the range 0.0 to 4.0; and the rank of the student's undergraduate institution from 4 to 1 (top). The admission decision is binary.

To build the model, we first initialize TensorFlow for an interactive session and define two variables and two placeholders, as follows.

```
import tensorflow as tf
import numpy as np
import csv
sess = tf.InteractiveSession()

x = tf.placeholder(tf.float32, shape=(None,3))
y = tf.placeholder(tf.float32, shape =(None,1))

# Set model weights
W = tf.Variable(tf.zeros([3, 1]))
b = tf.Variable(tf.zeros([1]))
```

The placeholder tensor x represents the triple $[GRE, GPA, Rank]$ from our data and the placeholder y holds the corresponding Admissions Decision. W and b are the learned variables that minimize the cost function.

$$cost = \sum_{i=0}^{1}(y - \sigma(W \cdot x + b))^2)$$

In this equation, $W \cdot x$ is the dot product, but the placeholders are of shape `(None, 3)` and `(None,1)`, respectively. In TensorFlow, this means that they can hold an array of size N x 3 and N x 1, respectively, for any value of N. The minimization step in TensorFlow now takes the following form, defining a graph with inputs x and y feeding into a cost function, which is then passed to the optimizer to select the W and b that minimize the cost.

```
pred = tf.sigmoid(tf.matmul(x, W) + b)
cost = tf.sqrt(tf.reduce_sum((y - pred)**2/batch_size))
opt = tf.train.AdamOptimizer()
optimizer = opt.minimize(cost)
```

The standard way to train a system in TensorFlow (and indeed in the other packages that we discuss here) is to run the optimizer with successive batches of training data. To do this, we need to initialize the TensorFlow variables with the current interactive session. We use a Python function `get_batch()` that pulls a batch of values from `train_data` and stores them in `train_label` arrays.

```
training_epochs = 100000
batch_size = 100
display_step = 1000
init = tf.initialize_all_variables()

sess.run(init)
# Training cycle
for epoch in range(training_epochs):
   avg_cost = 0.
   total_batch = int(len(train_data)/batch_size)
   # Loop over all batches
   for i in range(total_batch):
       batch_xs,batch_ys=get_batch(batch_size,train_data,train_label)
       # Fit training using batch data
       _,c=sess.run([optimizer,cost],
                    feed_dict={x:batch_xs, y:batch_ys})
       # Compute average loss
       avg_cost += c / total_batch
   # Display logs per epoch step
   if (epoch+1) % display_step == 0:
       print("Epoch:", '%04d' % (epoch+1), "cost=", str(avg_cost))
```

Figure 10.11: TensorFlow code for training the simple logistic regression function.

The code segment in figure 10.11 illustrates how data are passed to the computation graph for evaluation with the `sess.run()` function via a Python dictionary, which binds the data to the specific TensorFlow placeholders. Notebook 23 provides additional details, including analysis of the results. You will see that training on the fake admissions dataset led to a model in which the decision to admit is based solely on the student graduating from the top school. In this case the training rapidly converged, since this rule is easy to “learn.” The score was 99.9% accurate. If we base the admission decision on the equally inappropriate policy of granting admission only to those students who either scored an 800 on the GRE or came from the top school, the learning does not converge as fast and the best we could achieve was 83% accuracy.

A good exercise for you to try would be to convert this model to a neural network with one or more hidden layers. See whether you can improve the result!

10.7 Microsoft Cognitive Toolkit

We introduced the **Microsoft Cognitive Toolkit** in section 10.4.3 on page 209, when discussing recurrent neural networks. The CNTK team has made this software available for download in a variety of formats so that deep learning examples can be run on Azure as clusters of Docker containers, in the following configurations:

- CNTK-CPU-InfiniBand-IntelMPI for execution across multiple InfiniBand RDMA VMs
- CNTK-CPU-OpenMPI for multi-instance VMs
- CNTK-GPU-OpenMPI for multiple GPU-equipped servers such as the NC class, which have 24 cores and 4 K80 NVIDIA GPUs

These deployments each use the Azure Batch Shipyard Docker model, part of Azure Batch [6]. (Shipyard also provides scripts to provision Dockerized clusters for MXNet and TensorFlow with similar configurations.)

You also can deploy CNTK on your Windows 10 PC or in a VM running in any cloud. We provide detailed deployment instructions in notebook 22, along with an example that we describe below. The style of computing is similar to Spark, TensorFlow, and others that we have looked at. We use Python to build a flow graph of computations that we invoke with data using an `eval` operation. To illustrate the style, we create three tensors to hold the input values to a graph and then tie those tensors to the matrix-multiply operator and vector addition.

```
import numpy as np
import cntk
X = cntk.input_variable((1,2))
M = cntk.input_variable((2,3))
B = cntk.input_variable((1,3))
Y = cntk.times(X,M)+B
```

`X` is a 1×2-dimensional tensor, that is, a vector of length 2; `M` is a 2×3 matrix; and `B` is a vector of length 3. The expression `Y=X*M+B` yields a vector of length 3.

However, no computation has taken place at this point: we have only constructed a graph of the computation. To execute the graph, we input values for `X`, `B`, and `M`, and then apply the `eval` operator on `Y`, as follows. We use Numpy arrays to initialize the tensors and, in a manner identical to TensorFlow, supply a dictionary of bindings to the eval operator as follows.

```
x = [[ np.asarray([[40,50]]) ]]
m = [[ np.asarray([[1, 2, 3], [4, 5, 6]]) ]]
b = [[ np.asarray([1., 1., 1.])]]

print(Y.eval({X:x, M: m, B: b}))

----- output -------------

array([[[[ 241.,  331.,  421.]]]], dtype=float32)
```

CNTK also supports several other tensor container types, such as `Constant`, for a scalar, vector, or other multidimensional tensor with values that do not change, and `ParameterTensor`, for a tensor variable whose value is to be modified during network training.

Many more tensor operators exist, and we cannot discuss them all here. However, one important class is the set of operators that can be used to build multilevel neural networks. Called the *layers library*, they form a critical part of CNTK. One of the most basic is the `Dense(dim)` layer, which creates a fully connected layer of output dimension `dim`. Many other standard layer types exist, including Convolutional, MaxPooling, AveragePooling, and LSTM. Layers can also be stacked with a simple operator called `sequential`. We show two examples taken directly from the CNTK documentation [27]. The first is a standard five-level image recognition network based on convolutional layers.

```
with default_options(activation=relu):
  conv_net = Sequential ([
     # 3 layers of convolution and dimension reduction by pooling
     Convolution((5,5),32,pad=True),MaxPooling((3,3),strides=(2,2)),
     Convolution((5,5),32,pad=True),MaxPooling((3,3),strides=(2,2)),
     Convolution((5,5),64,pad=True),MaxPooling((3,3),strides=(2,2)),
     # 2 dense layers for classification
     Dense(64),
     Dense(10, activation=None)
  ])
```

The second example, on the next page, is a recurrent LSTM network that takes words **embedded** in a vector of size 150, passes them to the LSTM, and produces output through a dense network of dimension `labelDim`.

```
model = Sequential ([
    Embedding(150),          # Embed into a 150-dimensional vector
    Recurrence(LSTM(300)),   # Forward LSTM
    Dense(labelDim)          # Word-wise classification
])
```

You use word embeddings when your inputs are sparse vectors of size equal to the word vocabulary (i.e., if item i in the vector is 1, then the word is the ith element of the vocabulary), in which case the embedding matrix has size vocabulary-size by number of inputs. For example, if there are 10,000 words in the vocabulary and you have 150 inputs, then the matrix is 10,000 rows of length 150, and the ith word in the vocabulary corresponds to the ith row. The embedding matrix may be passed as a parameter or learned as part of training. We illustrate its use with a detailed example later in this chapter.

The `Sequential` operator used in the same code can be thought of as a concatenation of the layers in the given sequence. The `Recurrence` operator is used to wrap the correct LSTM output back to the input for the next input to the network. For details, we refer you to the tutorials provided by CNTK. One example of particular interest concerns **reinforced learning**, a technique that allows networks to use feedback from dynamical systems, such as games, in order to learn how to control them. We reference a more detailed discussion online [134].

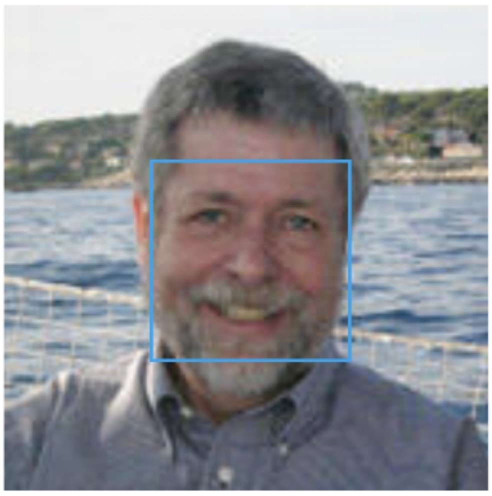
Azure also provides a large collection of pretrained machine learning services similar to those provided by the Amazon Machine Learning platform: the **Cortana cognitive services**. Specifically, these include web service APIs for speech and language understanding; text analysis; language translation; face recognition and attitude analysis; and search over Microsoft's academic research database and graph. Figure 10.12 shows an example of their use.

10.8 Summary

We have introduced a variety of cloud and open source machine learning tools. We began with a simple logistic regression demonstration that used the machine learning tools in Spark running in an Azure HDInsight cluster. We next turned to the Azure Machine Learning workspace Azure ML, a portal-based tool that provides a drop-and-drag way to compose, train, and test a machine learning model and then convert it automatically into a web service. Amazon also provides a portal-based tool, Amazon Machine Learning, that allows you to build and train a predictive model and deploy it as a service. In addition, both Azure and Amazon provide pre-trained models for image and text analysis, in the Cortana services and the Amazon ML platform, respectively.

We devoted the remainder of this chapter to looking at deep learning and the TensorFlow, CNTK, and MXNet toolkits. The capabilities of these tools can sometimes seem almost miraculous, but as Oren Etzioni [118] observes, "Deep learning isn't a dangerous magic genie. It's just math." We presented a modest introduction to the topic and described two of the most commonly used networks: convolutional and recurrent. We described the use of the Amazon virtual machine image (AMI) for machine learning, which includes MXNet, Amazon's preferred deep learning toolkit, as well as deployments of all the other deep learning frameworks. We illustrated MXNet with the Resnet-152 image recognition network first designed by Microsoft Research. Resnet-152 consists of 152 layers, and we demonstrated how it can be used to help classify biological samples. This type of image recognition has been used successfully in scientific studies ranging from protein structure to galaxy classification [180, 60, 264, 111].

We also used the Amazon ML AMI to demonstrate TensorFlow, Google's open



```
[
  {
    "faceRectangle": {
      "left": 45,
      "top": 48,
      "width": 62,
      "height": 62
    },
    "scores": {
      "anger": 0.0000115756638,
      "contempt": 0.00005204394,
      "disgust": 0.0000272641719,
      "fear": 9.037577e-8,
      "happiness": 0.998033762,
      "neutral": 0.00184232311,
      "sadness": 0.0000301841555,
      "surprise": 0.00000277762956
    }
  }
]
```

Figure 10.12: Cortana face recognition and attitude analysis web service. When applied to an image of a person on a sailboat, it returns the JSON document on the right. Cortana determines that there is one extremely (99.8%!) happy face in the picture.

source contribution to the deep learning world. We illustrated how one defines a convolution neural network in TensorFlow as part of our discussion of that topic, and we provided a complete example of using TensorFlow for logistic regression. Microsoft's cognitive tool kit (CNTK) was the third toolkit that we presented. We illustrated some of its basic features, including its use for deep learning. CNTK also provides an excellent environment for Jupyter, as well as many good tutorials.

We have provided in this chapter only a small introduction to the subject of machine learning. In addition to the deep learning toolkits mentioned here, Theano [47] and Caffe [161] are widely used. Keras `keras.io` is another interesting Python library that runs on top of Theano and TensorFlow. We also have not discussed the work done by IBM with their impressive Watson services—or systems such as Torch `torch.ch`.

Deep learning has had a profound impact on the technical directions of each of the major cloud vendors. The role of deep neural networks in science is still small, but we expect it to grow.

Another topic that we have not addressed in this chapter is the performance of ML toolkits for various tasks. In chapter 7 we discussed the various ways by which a computation can be scaled to solve bigger problems. One approach is the SPMD model of communicating sequential processes by using the Message Passing Standard (MPI) model (see section 7.2 on page 97). Another is the graph execution dataflow model (see chapter 9), used in Spark, Flink, and the deep learning toolkits described here.

Clearly we can write ML algorithms using either MPI or Spark. We should therefore be concerned about understanding the relative performance and programmability of the two approaches. Kamburugamuve et al. [166] address this topic and demonstrate that MPI implementations of two standard ML algorithms perform much better than the versions in Spark and Flink. Often the differences were factors of orders of magnitude in execution time. They also acknowledge that the MPI versions were harder to program than the Spark versions. The same team has released a library of MPI tools called SPIDAL, designed to perform data analytics on HPC clusters [116].

10.9 Resources

The classic *Data Mining: Concepts and Techniques* [148], recently updated, provides a strong introduction to data mining and knowledge discovery. *Deep Learning* [143] is an exceptional treatment of that technology.

For those interested in learning more of the basics of machine learning with Python and Jupyter, two good books are *Python Machine Learning* [224] and *Introduction to Machine Learning with Python: A Guide for Data Scientists* [207]. All the examples in this chapter, with the exception of k-means, involve supervised learning. These books treat the subject of unsupervised learning in more depth.

On the topic of deep learning, each of the three toolkits covered in this chapter—CNTK, TensorFlow, and MXNet—provides extensive tutorials in their standard distributions, when downloaded and installed.

We also mention the six notebooks introduced in this chapter.

- Notebook 18 demonstrates the use of Spark machine learning for logistic regression.
- Notebook 19 can be used to send data to an AzureML web service.
- Notebook 20 demonstrates how to load and use the RNN model originally built with CNTK.
- Notebook 21 shows how to load and use the MXNet Resnet-152 model to classify images.
- Notebook 22 discusses the installation and use of CNTK.
- Notebook 23 illustrates simple logistic regression using TensorFlow.