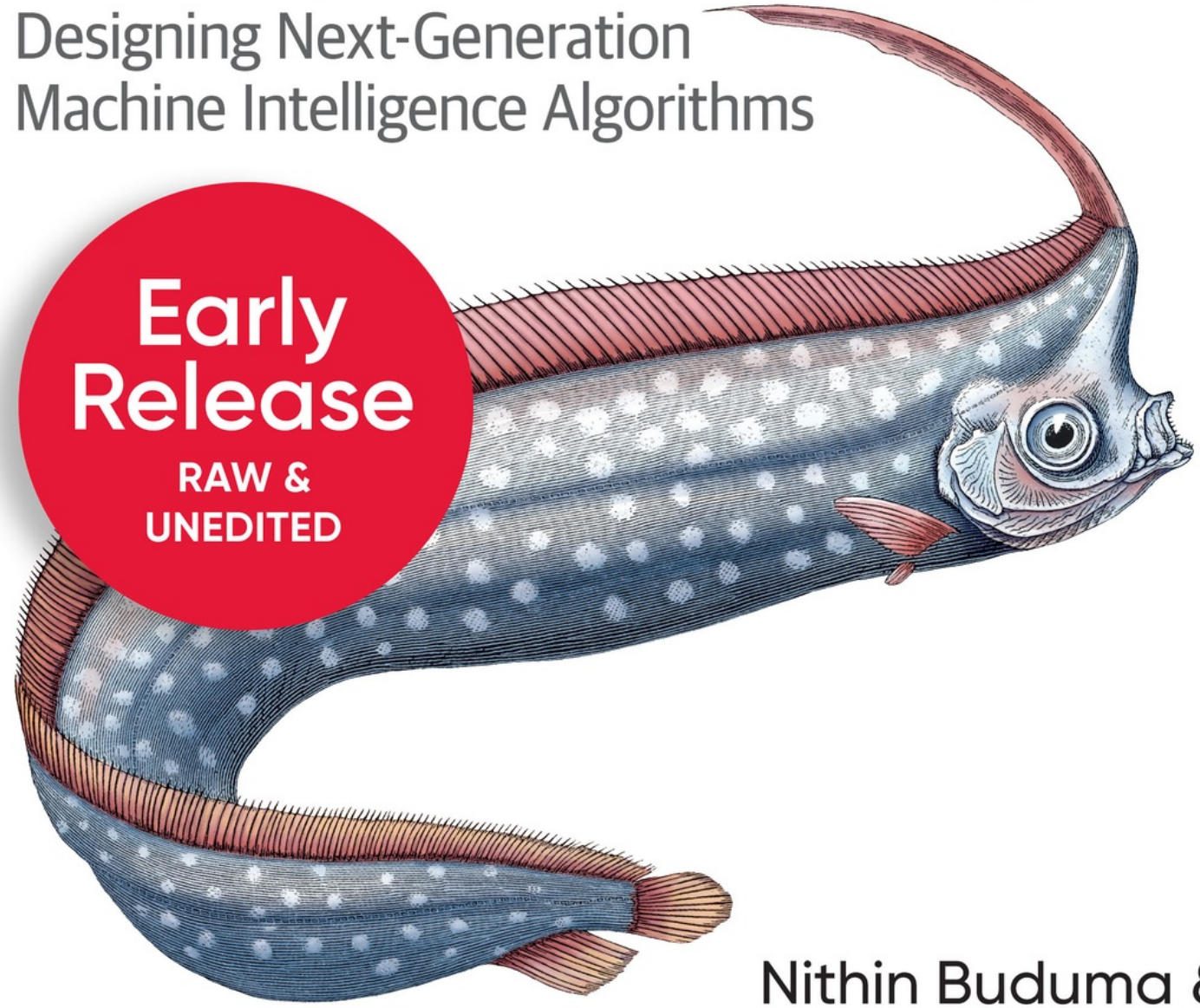
O'REILLY®

Second Edition

Fundamentals of Deep Learning

Designing Next-Generation Machine Intelligence Algorithms



Early Release

RAW & UNEDITED

Nithin Buduma &
Nikhil Buduma
with contributions by Nicholas Locascio

Fundamentals of Deep Learning

SECOND EDITION

Designing Next-Generation Machine Intelligence Algorithms

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

Nithin Buduma and Nikhil Buduma with contributions by Nicholas Locascio

Fundamentals of Deep Learning

by Nikhil Buduma, Nithin Buduma and Nicholas Lacascio

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (http://oreilly.com/safari). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editors: Mike Loukides and Shannon Cutt

Production Editor: Katherine Tozer

Copyeditor: TK

Proofreader: TK

Indexer: TK

Interior Designer: David Futato

Cover Designer: Karen Montgomery

Illustrator: Kate Dullea

March 2022: First Edition

Revision History for the Early Release

- 2021-02-03: First Release

978-1-492-08218-7

Chapter 1. Fundamentals of Linear Algebra

In this chapter, we cover important prerequisite knowledge that will motivate our discussion of deep learning techniques in the main text and optional material at the ends of selected chapters. Deep learning has recently experienced a renaissance, both in academic research and in the industry. It has pushed the limits of machine learning by leaps and bounds, revolutionizing fields such as computer vision and natural language processing. However, it is important to remember that deep learning is, at its core, a culmination of achievements in fields such as calculus, linear algebra, and probability. Although there are deeper connections to other fields of mathematics, we focus on the three listed here to help us broaden our perspective before diving into deep learning. These fields are key to unlocking both the big picture of deep learning and the intricate subtleties that make it as exciting as it is. In this first chapter on background, we cover the fundamentals of linear algebra.

Data Structures and Operations

The most important data structure in linear algebra (whenever we reference linear algebra in this text, we refer to its applied variety) is arguably the *matrix,* a 2-d array of numbers where each entry can be indexed via its row and column. Think of an Excel spreadsheet, where you have offers from Company X and Company Y as two rows, and the columns represents some characteristic of each offer such as starting salary, bonus, or position.

	Salary	Bonus	Position
Company X	\$50,000	\$5,000	Engineer
Company Y	\$45,000	\$7,500	Data Scientist

Figure 1-1. An example of tabular data

The table format is especially suited to keep track of such data, where you can index by row and column to find, for example, Company X's starting position. Matrices, similarly, are a multipurpose tool to hold all kinds of data, where the data we work in this book are of numerical form. In deep learning, matrices are often used to represent both datasets and weights in

a neural network. A dataset, for example, has many individual data points with any number of associated features. A lizard dataset might contain information on length, weight, speed, age and other important attributes. We can represent this intuitively as a matrix or table, where each row represents an individual lizard, and each column represents a lizard feature such as age. However, as opposed to the table presented above, the matrix only stores the numbers themselves and assumes the user has kept track of which rows correspond to which data points, which columns correspond to which feature, and what the units are for each feature:

	Length	Weight	Speed	Age
Gecko Grayson	0.5m	0.1kg	4kph	10 years
Komodo Ken	2.5m	50kg	15kph	20 years
Iguana Ian	0.4m	9kg	30kph	15 years

$$\longrightarrow \begin{bmatrix} 0.5 & 0.1 & 4 & 10 \\ 2.5 & 50 & 15 & 20 \\ 0.4 & 9 & 30 & 15 \end{bmatrix}$$

Figure 1-2. A comparison of tables and matrices

On the right side, we have a matrix, where it's assumed for example that the age of each lizard is in years, and Komodo Ken weighs a whopping 50 kilograms! But why even work with matrices when tables clearly give the user more information? Well, in linear algebra and even deep learning, operations such as multiplication and addition are done on the tabular data itself, but such operations can only be computed efficiently when the data is in solely numerical format.

Much of the work in linear algebra centers on the emergent properties of matrices, which are especially interesting when the matrix has certain base attributes, and operations on these data structures. *Vectors,* which can be seen as a subset type of matrices, are a 1-d array of numbers. This data structure can be used to represent an individual data point or the weights in a linear regression, for example. We cover properties of matrices and vectors as well as operations on both.

Matrix Operations

Matrices can be added, subtracted, and multiplied - there is no division of matrices, but there exists a similar concept called inversion.

When indexing a matrix, we use a tuple, where the first index represents the row number and the second index represents the column number. To add two matrices A and B, one loops through each index (i,j) of the two matrices, sums the two entries at the current index, and places that result in the same index (i,j) of a new matrix C:

$$\begin{bmatrix} 1 & 0 & 5 \\ 2 & 4 & 7 \end{bmatrix} + \begin{bmatrix} 5 & 3 & 2 \\ 6 & 1 & 4 \end{bmatrix} = \begin{bmatrix} 1+5 & 0+3 & 5+2 \\ 2+6 & 4+1 & 7+4 \end{bmatrix} = \begin{bmatrix} 6 & 3 & 7 \\ 8 & 5 & 11 \end{bmatrix}$$

Figure 1-3. Matrix addition

This algorithm implies that we can't add two matrices of different dimension, since indices that exist in one matrix wouldn't exist in the other. It also implies that the final matrix *C* is of the same dimension as *A* and *B.* In addition to adding matrices, we can multiply a matrix by a scalar. This involves simply taking the scalar and multiplying each of the entries of the matrix by it (the dimension of the resultant matrix stays constant):

$$2 * \begin{bmatrix} 1 & 0 & 5 \\ 2 & 4 & 7 \end{bmatrix} = \begin{bmatrix} 2*1 & 2*0 & 2*5 \\ 2*2 & 2*4 & 2*7 \end{bmatrix} = \begin{bmatrix} 2 & 0 & 10 \\ 4 & 8 & 14 \end{bmatrix}$$

Figure 1-4. Scalar-matrix multiplication

These two operations, addition of matrices and scalar-matrix multiplication, lead us directly to matrix subtraction, since computing *A-B* is the same as computing the matrix addition *A+(-B),* and computing *-B* is the product of a scalar -1 and the matrix *B.*

Multiplying two matrices starts to get interesting. For reasons beyond the scope of this text (motivations in a more theoretical flavor of linear algebra where matrices represent linear transformations), we define the matrix product $A \cdot B$ as:

$(A \cdot B)_{i,j} = \sum_{k'=1}^{k} A_{i,k'} B_{k',j}$

In simpler terms, this means that the value at the index *(i,j)* of $A \cdot B$ is the sum of the product of the entries in the ith row of *A* with those of the *j*th column of *B.* Below is an example of matrix multiplication:

$$\begin{bmatrix} 4 & 6 \\ 2 & 7 \end{bmatrix} \cdot \begin{bmatrix} 1 & 9 \\ 5 & 2 \end{bmatrix} = \begin{bmatrix} 4*1+6*5 & 4*9+6*2 \\ 2*1+7*5 & 2*9+7*2 \end{bmatrix} = \begin{bmatrix} 34 & 48 \\ 37 & 32 \end{bmatrix}$$

Figure 1-5. Matrix multiplication

It follows that the rows of A and the columns of B must have the same length, so two matrices can only be multiplied if the dimensions align, i.e. A is of dimension m by k and B is of dimension k by n. If this weren't the case, the formula for matrix multiplication would give us an indexing error. The dimension of the product is m by n, signifying an entry for every pair of row in A and column in B. This is the computational way of thinking about matrix multiplication, and doesn't lend itself well to theoretical interpretation. We'll call the formula for matrix multiplication presented above the *dot product interpretation of matrix multiplication,* which will make more sense after reading the Vector Operations section.

Note that matrix multiplication is not commutative, i.e. $A \cdot B \neq B \cdot A$. Of course, if we were to take a matrix A that is 2 by 3 and a matrix B that is 3 by 5 for example, by the rules of matrix multiplication $B \cdot A$ doesn't exist. However, even if the product were defined due to both matrices being square, where square means that the matrix has an equal number of rows and columns, the two products will not be the same (exercise for the reader). However, matrix multiplication is associative, i.e. $A \cdot (B + C) = A \cdot B + A \cdot C$.

Let's delve into matrix multiplication a bit further. After some algebraic manipulation, one can see that another way to formulate matrix multiplication is:

$$(A \cdot B)_{\cdot,j} = A \cdot B_{\cdot,j}$$

This states that the jth column of the product $A \cdot B$ is the matrix product of A and the jth column of B, a vector. We'll call this *column vector interpretation of matrix multiplication*:

$$\begin{aligned}
\begin{bmatrix} 4 & 6 \\ 2 & 7 \end{bmatrix} \cdot \begin{bmatrix} 1 & 9 \\ 5 & 2 \end{bmatrix} &= \begin{bmatrix} \begin{bmatrix} 4 & 6 \\ 2 & 7 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 5 \end{bmatrix} & \begin{bmatrix} 4 & 6 \\ 2 & 7 \end{bmatrix} \cdot \begin{bmatrix} 9 \\ 2 \end{bmatrix} \end{bmatrix} \\
&= \begin{bmatrix} 4*1+6*5 & 4*9+6*2 \\ 2*1+7*5 & 2*9+7*2 \end{bmatrix} \\
&= \begin{bmatrix} 34 & 48 \\ 37 & 32 \end{bmatrix}
\end{aligned}$$

Figure 1-6. Matrix multiplication: another view

In a later section, we cover matrix-vector multiplication and different ways to think about this computation, which leads to more exciting properties regarding matrices!

One of the most important matrices in linear algebra is the *identity matrix*, which is a square matrix with 1's along the main diagonal and 0's in every other entry. This matrix is usually denoted as I. When computing the product of I with any other matrix A, the result is always A - thus its name the identity matrix. Try multiplying a few matrices of your choosing with the appropriate-sized identity matrix to see why this is the case.

As noted at the beginning of the section, there is no such division operation for matrices, but there is the concept of inversion. The inverse of a matrix A is the matrix B such that $AB = BA = I$, the identity matrix (similar in idea to a number's reciprocal - when dividing by a number on both sides of an equation, we can also think of this operation as multiplying both sides by its reciprocal). If such a B exists, we denote it as A^{-1}. From this definition, we know that A must be, at the very least, a square matrix since we are able to multiply A on either side by the same matrix A^{-1}.

$$\begin{bmatrix} \frac{-5}{4} & \frac{3}{4} \\ \frac{3}{4} & \frac{-1}{4} \end{bmatrix} \begin{bmatrix} 1 & 3 \\ 3 & 5 \end{bmatrix} = \begin{bmatrix} 1 & 3 \\ 3 & 5 \end{bmatrix} \begin{bmatrix} \frac{-5}{4} & \frac{3}{4} \\ \frac{3}{4} & \frac{-1}{4} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

Figure 1-7. Matrix inversion

When trying to solve an equation such as $Ax = b$ for x, we would multiply both sides on the left by A^{-1} to get $x = A^{-1}b$ if A is invertible. There exists another necessary condition for A to be invertible which we'll discuss later.

Vector Operations

Vectors can be seen as a subset of matrices, so a lot of the operations follow from above (i.e. the properties of addition, subtraction, multiplication and inversion). However, there is some vector-specific terminology we should cover. When a vector is of dimension 1 by n, we call this vector a *row vector*, and when the vector is of dimension n by 1, we call it a *column vector*. When taking matrix product of a row vector and a column vector, one can see that the result is a single number - we call this operation the *dot product.* Below is an example of the dot product of two vectors:

$$\begin{bmatrix} 4 & 3 & 6 \end{bmatrix} \begin{bmatrix} 2 \\ 7 \\ 1 \end{bmatrix} = 4 * 2 + 3 * 7 + 6 * 1 = 35$$

Figure 1-8. Dot product

Now the reason for the name dot product interpretation of matrix multiplication might make more sense. Looking back at the formula, we see that every entry in the matrix product $(A \cdot B)_{i,j}$ is just the dot product of the corresponding row $A_{i,\cdot}$ and the corresponding column $B_{\cdot,j}$.

When the dot product of two vectors is 0, we term the two vectors to be *orthogonal*. Orthogonality is a generalization of perpendicularity to any dimension, even those far beyond the ones we can imagine! You can check in the two dimensional case, for example, that any two vectors are perpendicular if and only if (also termed *iff)* they have a dot product of 0.

When one instead takes the matrix product of a column vector and a row vector, we see that the result is quite surprisingly a matrix! This is termed the *outer product.* Below is the outer product of the same two vectors from the dot product example, except their roles as row and column vectors have been reversed:

$$\begin{bmatrix} 2 \\ 7 \\ 1 \end{bmatrix} \begin{bmatrix} 4 & 3 & 6 \end{bmatrix} = \begin{bmatrix} 2 * 4 & 2 * 3 & 2 * 6 \\ 7 * 4 & 7 * 3 & 7 * 6 \\ 1 * 4 & 1 * 3 & 1 * 6 \end{bmatrix}$$

Figure 1-9. Outer product

Matrix-vector multiplication

When multiplying a matrix *A* and a vector *v*, we can again do this via the dot product interpretation of matrix multiplication as described above. However, if we instead manipulate the expression slightly, we'll see that another way to formulate this product is:

$Av = \sum_j v_j A_{\cdot,j}$

Where each v_j is a constant to be multiplied with its corresponding column of *A.* Below is an example of this method in action:

$$\begin{bmatrix} 4 & 6 \\ 2 & 7 \\ 5 & 8 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 5 \end{bmatrix} = 1 * \begin{bmatrix} 4 \\ 2 \\ 5 \end{bmatrix} + 5 * \begin{bmatrix} 6 \\ 7 \\ 8 \end{bmatrix}$$
$$= \begin{bmatrix} 34 \\ 37 \\ 45 \end{bmatrix}$$

Figure 1-10. Matrix-vector multiplication

The Fundamental Spaces

The Column Space

Consider the set of all possible vectors v and their products Av. We term this the *column space* of *A,* or *C(A).* The term column space is used because *C(A)* represents all possible linear combinations of the columns of *A,* where a linear combination of vectors is a sum of constant scalings of each vector. The constant scaling for each column vector of *A* is determined by the choice of *v,* as we just saw in the previous section.

The column space is an example of a *vector space,* which is the space defined by a list of vectors and all possible linear combinations of this collection. Properties for formally defining a vector space pop up directly from this intuition. For example, if a set of vectors is a vector space, then the vector that arises from multiplying any vector in the space by a scalar must also be in the space. In addition, if we were to add any two vectors in the space, the result should still be in the space. In both of these operations, the vectors we start with are known to be in the vector space, and thus can be formulated as linear combinations of the original list. By performing scalar multiplication or addition on the vectors in question, we are just computing linear combinations of linear combinations, which are still linear combinations!

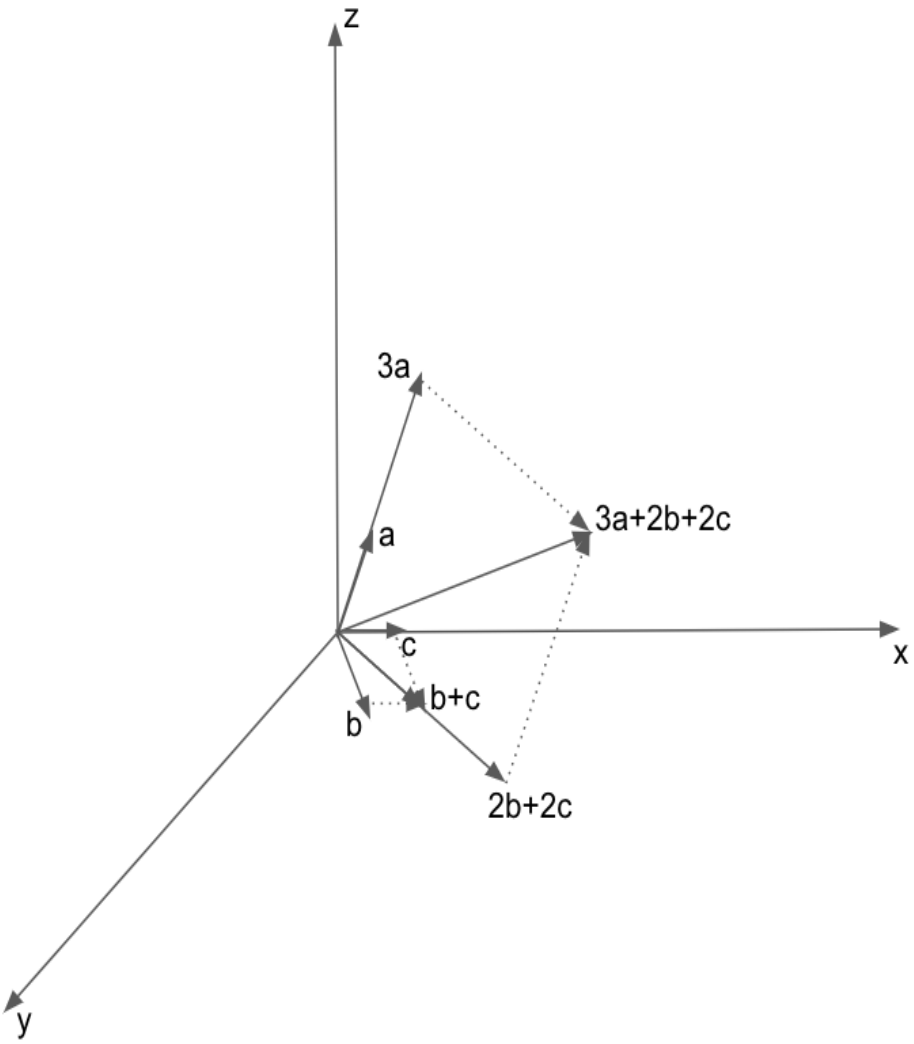


Figure 1-11. We can see that the sum, or linear combination, of the two linear combinations 3a and 2b+2c is still a linear combination of the original vectors a, b, and c.

We term these key properties of vector spaces *closed under scalar multiplication* and *closed under addition.* If a set of vectors doesn't always satisfy either of these properties, then the set clearly doesn't contain all possible linear combinations of the original list and is not a vector space.

An example of a vector space you're probably familiar with is $\mathbb{R}^3$, or the entire space defined by the x-y-z coordinate axis. The reason for the notation $\mathbb{R}^3$ is that each coordinate can take on any value in the reals, or $\mathbb{R}$, and there are three coordinates that uniquely define any such vector in this space. A collection of vectors that defines this space are the vectors (0,0,1), (0,1,0),(1,0,0), the unit vectors of each axis. Any vector (*a,b,c)* in the space can be written as *a**(1,0,0)+*b**(0,1,0)+*c**(0,0,1), a linear combination of the

collection. In the other direction, any possible linear combination of the three vectors represents some vector *(a,b,c)* that lies in $\mathbb{R}^3$.

Often times, there exist matrices *A* for which some columns are linear combinations of other columns. For example, imagine if in our lizard dataset from the opening section, we had an additional feature for each lizard's weight but instead in pounds. This is a clear redundancy in the data since this feature is completely determined by the feature for weight in kilograms. In other words, the new feature is a linear combination of the other features in the data - simply take the column for weight in kilograms, multiply it by 2.2, and sum it with all the other columns multiplied by zero to get the column for weight in pounds. Logically, if we were to remove these sorts of redundancies from *A,* then *C(A)* shouldn't change. One method to do this is to first create a list of all the original column vectors of *A,* where order is assigned arbitrarily. As one iterates through the list, check to see if the current vector is a linear combination of all the vectors that precede it. If so, remove this vector from the list and continue. It's clear that the removed vector provided no additional information beyond the ones we've already seen.

The resulting list is called the *basis* of *C(A),* and the length of the basis is the *dimension* of *C(A).* We say that the basis of any vector space *spans* the space, which means that all of the elements in the vector space can be formulated as a linear combination of basis vectors. In addition, the basis vectors are *linearly independent*, which means that none of the vectors can be written as a linear combination of the others; i.e. no redundancies. Going back to the example where we defined vector space, (0,0,1),(0,1,0),(1,0,0) would be a basis for the space $\mathbb{R}^3$, since no vector in the list is a linear combination of the others, and this list spans the entire space. And instead, the list (0,0,1),(0,1,0),(1,0,0),(2,5,1) spans the entire space, but is not linearly independent since (2,5,1) can be written as a linear combination of the first three vectors (we call such a list of vectors a *spanning list,* and of course the set of bases for a vector space is a subset of the set of spanning lists for the same space). In the other direction, we can start with a list of linearly independent vectors such as (0,0,1),(0,1,0) and grow this list to a basis by adding on vectors. For example, we can add on (1,0,0) to this list to get our standard basis (0,0,1),(0,1,0),(1,0,0). In general, we can always remove vectors from a spanning list to achieve a basis and we can always add vectors to a linearly independent list of vectors that is not already a basis to achieve a basis for a vector space.

You might be thinking, what if we had assigned a different ordering of the vectors to start? Is it possible that we achieve a different basis for *C(A)* with a potentially different length? Is it possible that these bases determine a different *C(A)* depending on the initial ordering? It is indeed the case that

we can achieve a different basis, but the length will remain the same across all possible bases, and all bases are for the same *C(A)*. In fact, it is the case that for any vector space, all bases for the space are of the same length (this length is termed the *dimension* of the space), and all spanning lists are at least as long as all bases! If you are curious about the proofs of these statements and more, please refer to a great text by Prof. Sheldon Axler called *Linear Algebra done Right*.

For an example about a different basis for the same vector space, let's revisit our favorite space, $\mathbb{R}^3$. Consider the list of vectors (0,0,1),(0,1,0), (1,0,1), all of which lie in the space. They are all also clearly linearly independent - none of the vectors in the list can be written as a linear combination of the others. And, equally importantly, the list spans the entire space: any vector (a,b,c) in $\mathbb{R}^3$ can be written as the linear combination: *(a-c)*(1,0,0)+*b**(0,1,0)+*c**(1,0,1). We have shown that (0,0,1),(0,1,0),(1,0,1) is another basis for $\mathbb{R}^3$.

The Null Space

Another key vector space is the *null space* of a matrix *A,* or *N(A).* This space consists of the vectors *v* such that *Av = 0.* We know that *v=0,* the trivial solution, will always satisfy this property. If only the trivial solution is in the null space of a matrix, we call the space trivial. However, it is possible that there exist other solutions to this equation depending on the properties of *A,* or a nontrivial null space. For a vector *v* to satisfy *Av = 0, v* must be orthogonal to each of the rows of *A.*

$$\begin{bmatrix} 4 & 6 \\ 2 & 7 \\ 5 & 8 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

$$\implies \begin{bmatrix} 4x + 6y = 0 \\ 2x + 7y = 0 \\ 5x + 8y = 0 \end{bmatrix}$$

Figure 1-12. The implication that the dot product between each row and the vector v must be equal to 0.

Let's assume *A* is of dimension 2 by 3, for example. In our case, *A's* rows cannot span $\mathbb{R}^3$ due to *A* having only two rows (remember from our recent

discussion that all bases have the same length, and all spanning lists are at least as long as all bases, so A's rows can be neither of these). At best, *A's* rows define a plane in the 3-d coordinate system. The other two options are that the rows define a line or a point. The former occurs when *A* either has two nonzero rows, where one is a multiple of the other, or has one zero row and one nonzero row. The latter occurs when *A* has two zero rows, or in other words, is the zero matrix.

In the case where *A's* row space defines a plane (or even a line for that matter), all we'd need to do to find a vector in *N(A)* is:

1. Pick any vector *v* that doesn't lie in *A's* row space.

2. Find its projection *v'* onto the row space, where the projection of *v* is defined as the vector in the space closest to *v.* Geometrically, the projection looks as if we had dropped a line down from the tip of *v* perpendicular to the space, and connected a vector from the origin to that point on the space.

3. Compute *v-v',* which is orthogonal to the row space and thus, each row vector.

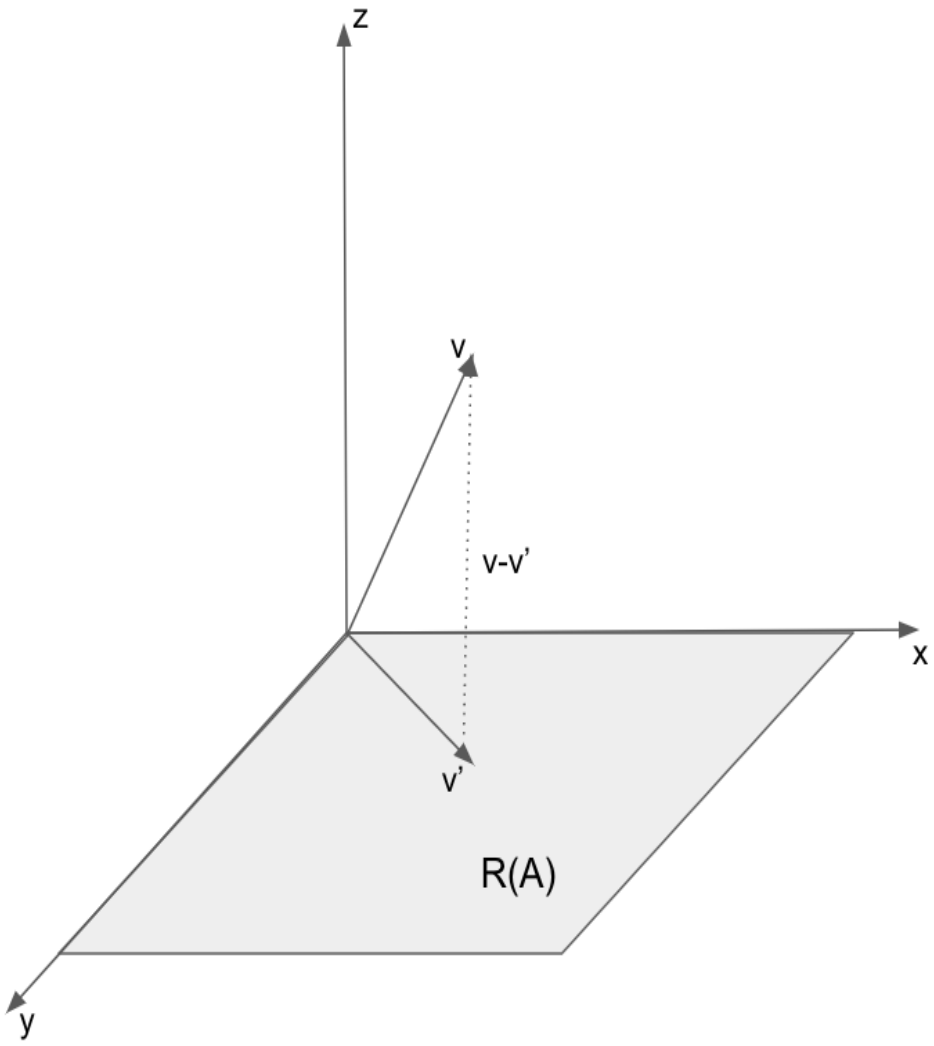


Figure 1-13. Note that v-v' is perpendicular to R(A), the row space of A, since v' was formed by dropping a perpendicular to the plane down from its tip.

An important takeaway is that nontrivial solutions to $Av = 0$ exist when the rows of A do not span $\mathbb{R}^3$; more generally, if A is of dimension m by n, nontrivial solutions to $Av = 0$ exist when the row vectors do not span $\mathbb{R}^n$. The process is similar to that shown above - pick a vector in $\mathbb{R}^n$ not in the row space, find its projection onto the row space and subtract to get a vector in null space.

But we still must show that $N(A)$ itself is a vector space. We can easily see that any linear combination of nontrivial solutions to $Av=0$ is still a solution. For example, given two nontrivial solutions v_1 and v_2 and their linear combination $c_1v_1 + c_2v_2$, where c_1 and c_2 are constants, we see that:

$$A(c_1v_1 + c_2v_2) = A(c_1v_1) + A(c_2v_2) = c_1Av_1 + c_2Av_2 = c_1 * 0 + c_2 * 0 = 0$$

Where the first equality arises from the associativity of matrix multiplication and the second from the fact that c_1 and c_2 are constants. Note that this logic can be used for any number of nontrivial solutions, not just two. Thus, the null space is defined by some collection of vectors that can be boiled down to a basis, and contains all possible linear combinations of these vectors. These characteristics make the null space a vector space.

This is all deeply connected to one of the key matrix operations presented above, the matrix inverse. We can think of a matrix's inverse as undoing the action of a matrix upon any other entity. For example, if we were to compute Av, and multiply on the left by A^{-1}, we should be left with our initial v. However, depending on the properties of A, there can exist ambiguities as to how to "undo" its action. For example, let's say v was some nonzero vector, but for some reason, $Av = 0$. If we were to multiply on the left by A^{-1}, we'd be left with $v=0$ instead of our initial v. This unfortunately goes against the properties of an inverse, and we declare that such a matrix is non-invertible, or *singular*. But why does this happen in the first place? This goes back to our observation about ambiguities. Since an inverse is supposed to undo the action of a matrix, if there are multiple initial vectors that map to the same vector via the matrix's action, trying to undo this action is impossible! Going back to our example, we know that nonzero vectors are mapped to 0 by A when A has a nontrivial null space. Thus, any matrix with a nontrivial null space is also singular.

In the other direction, if a matrix is singular, it must also have a nontrivial null space. If a matrix A is singular, then there must exist at least two different vectors $v_1$and v_2 such that $Av_1 = Av_2$. If this were not true, then A would be invertible. If one of v_1 or v_2 are zero, then A clearly has a nontrivial null space. If both are nonzero, then we can subtract Av_2 from both sides to get $Av_1 - Av_2 = A(v_1 - v_2) = 0$. Since $v_1$and v_2 are different, $v_1 - v_2$ is a nonzero vector in A's null space. We have shown that in any possible scenario where A is singular, A must also have a nontrivial null space. We have also established an iff relationship between being singular and having a nontrivial null space, meaning they are equivalent when testing for one or the other.

Eigenvectors and Eigenvalues

Matrices can act on vectors in many different ways. For most combinations of matrices and vectors, plotting the vector and its transformation doesn't provide us with any interesting patterns. However, for certain matrices and specific vectors for those matrices, the action of the matrix upon the vector gives us a very informative and surprising result: the transformation is a scalar multiple of the original! We call these vectors *eigenvectors*, and the

scalar multiple its corresponding *eigenvalue.* In this section we discuss these very special vectors, relate back to the material presented in the previous section, and begin the discussion connecting the theory of linear algebra with the practice of data science.

More formally, an eigenvector for a matrix A is a nonzero vector v such that $Av = cv$, where c is some constant (including zero, potentially).

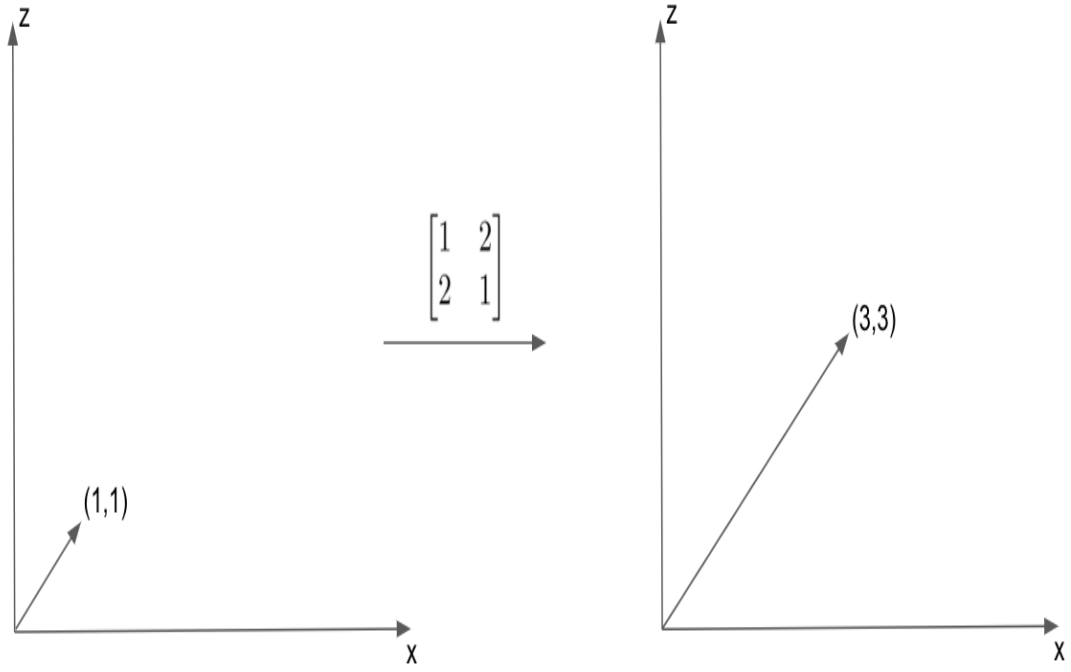


Figure 1-14. We see here that the vector (1,1) is an eigenvector of our matrix, with a corresponding eigenvalue of 3. Note that if we were to pick any random vector, such as (2,5), the transformation wouldn't look as meaningful as it does here.

Of course, if A is a rectangular matrix, it's impossible for A to have any eigenvectors. The original vector and its transformation have different sizes, and thus transformation couldn't be a scalar multiple of the original. For this reason, we limit our discussion in this section to square matrices.

The simplest example is the identity matrix. Every nonzero vector is an eigenvector of the identity matrix since $Iv = v$ for all v, with each having eigenvalue 1. Often times, however, the eigenvectors of a matrix won't be so obvious. How do we find these vectors and their corresponding eigenvalues? We know the conditions of any potential eigenvector; that is, if v is an eigenvector, it must satisfy $Av = cv$ for some scalar c:

$$Av = cv \Leftrightarrow Av - cv = 0 \Leftrightarrow (A - cI)v = 0$$

The implication here is that if $Av = cv$, then $A - cI$ must have a nontrivial null space. In the other direction, if we find a c such that $A - cI$ has a nontrivial null space, the nonzero vectors in the null space are eigenvectors of A. Of course, if A itself has a nontrivial null space, then all nonzero v in the null space satisfy the above implication when c is 0. More generally, however, we must find the c such that A - cI has a nontrivial null space. As established at the end of the last section, checking for a nontrivial null space is equivalent to testing for a matrix being singular. For reasons beyond the scope of this text, one way to test if $A - cI$ for some c is a singular matrix is to check whether its *determinant* is 0. We won't go into too much depth here, but we can think about the determinant as a function, or polynomial, that encodes properties of the matrix and results in a value of 0 iff the matrix is singular.

However, it would be inefficient, and frankly impossible, for us to test every possible c for a zero determinant! We can instead think of c as a variable in an equation and solve for it via the *characteristic polynomial*, which is the determinant of the matrix $A - cI$ set equal to 0. The roots of this polynomial give us the eigenvalues of A. To find their corresponding eigenvectors, we can plug each solution for c into $A - cI$ and then solve for the v that make(s) $(A - cI)v = 0$.

Side note: Calculating the determinant for any matrix of reasonable size is quite prohibitive in terms of computational cost. Although we won't delve further into this, algorithms today use a version of the QR algorithm (named after the QR matrix decomposition) to calculate the eigenvalues of a matrix. If you'd like to learn more about these and similar such algorithms, we highly recommend lecture notes or books on numerical linear algebra!

How does our study of eigenvalues and eigenvectors connect to that of data science? Principal component analysis, or PCA, is one of the most famous algorithms in data science, and it uses the eigenvectors and eigenvalues of a special matrix called the correlation matrix to perform dimensionality reduction on the original data matrix. We will discuss correlation and related concepts in the next chapter on probability, and learn more about PCA in Chapter 6.

Summary

In this chapter, we investigated some of the basics of applied linear algebra. We learned about the key data structures and operations that rule both applied linear algebra and deep learning, and different ways to view these fundamental operations. For example, we learned that the dot product view of matrix multiplication was important from a computational lens, while the column vector approach led us into our discussion on the fundamental spaces quite naturally. We also got a peek at some of the surprising hidden

properties of matrices, such as eigenvalues and eigenvectors, and how these properties are widely utilized in data science even to this day.

Chapter 2. Fundamentals of Probability

Probability is a field of mathematics that quantifies our uncertainty regarding events. For example, when rolling a dice or flipping a coin, barring any irregularities in the dice or coin themselves we are uncertain about the result to come. However, we can quantify our belief in each of the potential outcomes via probabilities. We say, for example, that on every coin toss the probability of the coin showing up heads is $\frac{1}{2}$. And on every dice roll, we say the probability of the dice facing up with a five is $\frac{1}{6}$. These are the sorts of probabilities we talk about with ease in our daily lives, but how can we define and utilize them effectively? In this chapter we'll discuss the fundamentals of probability and how they connect to key concepts in deep learning.

Events and Probability

As discussed in the Preface, when running a trial such as rolling a dice or tossing a coin, we intuitively assign some belief to the trial's possible outcomes. In this section, we aim to formalize some of these concepts. In particular, we will begin by working in this *discrete* space, where discrete signifies a finite or countably infinite number of possibilities. Both rolling a dice and tossing a coin are in the discrete space - when rolling a fair dice there are six possible outcomes and when tossing a fair coin there are two. We term the entire set of possibilities for an experiment the *sample space.* For example, the numbers one through six would make up the sample space for rolling a fair dice. We can define *events* as subsets of the sample space. The event of rolling at least a three corresponds with the dice facing up any number in the subset of three, four, five and six in the sample space defined previously. A set of probabilities that sum to 1 over all outcomes in the sample space is termed a

probability distribution over that sample space, and these distributions will be the main focus of our discussion.

In general, we won't worry too much on where exactly these probabilities come from, as that requires a much more rigorous and thorough examination beyond the scope of this text. However, we will give some intuition about the different interpretations. At a high level, the *frequentist* view sees the probability of an outcome as arising from its frequency over a long-run experiment. In the case of fair dice, this view claims we can say the probability of any side of the dice showing up on a given roll is $\frac{1}{6}$ since performing a large number of rolls and counting up the occurrences of each side will give us an estimate that is roughly this fraction. As the number of rolls in the experiment grows, we see that this estimate gets closer and closer to the limit $\frac{1}{6}$, the outcome's probability. On the other hand, the *B*ayesian view of probability is based more on quantifying our prior belief in hypotheses and how we update our beliefs in light of new data. For a fair dice, the Bayesian view would claim there is no prior information, both from the dice's structure and the rolling process, that would suggest any side of the dice as being more likely to turn up than any other side. Thus, we would say each outcome has probability $\frac{1}{6}$, our prior belief. The set of probabilities, in this case all being $\frac{1}{6}$, associated with each outcome is termed our *prior*. As we see new data, the Bayesian view gives us a methodology to update our prior accordingly, where we term this new belief our *posterior*. This Bayesian view is sometimes directly applied to neural network training, where we first assume that each weight in the network has some prior associated with it. As we train the network, we update the prior associated with each weight accordingly to better fit the data we see. At the end of training, we are left with a posterior distribution associated with each weight.

We will assume throughout this chapter that the probabilities associated with any outcome have been determined via reasonable methods, and rather focus on how we can manipulate these probabilities for use in our analyses. We start with the tenets of probability, specifically in the discrete space:

1. The sum of probabilities for all possible outcomes in a sample space must be equal to one. In other words, the probability distribution over the sample space must sum to one. This should make sense intuitively, since the set of all outcomes in the sample space must represent the entire set of possibilities. The probability distribution not summing to one would imply the existence of possibilities not accounted for, which is contradictory. Mathematically, we say that for any valid probability distribution, $\sum_o P(o) = 1$, where o represents an outcome.

2. Let E_1be an event, and recall that we define an event as a subset of possible outcomes. We call E_1^c the *complement* of E_1 , or all possible outcomes in the sample space that are not in E_1. The second tenet of probability is that $P(E_1) = 1 - P(E_1^c)$. This is just an application of the first tenet - if this were not true, it would clearly contradict the first tenet. In Figure 1, we see an example of this, where S represents the entire space of outcomes and the event and its complement together form the entirety of S:

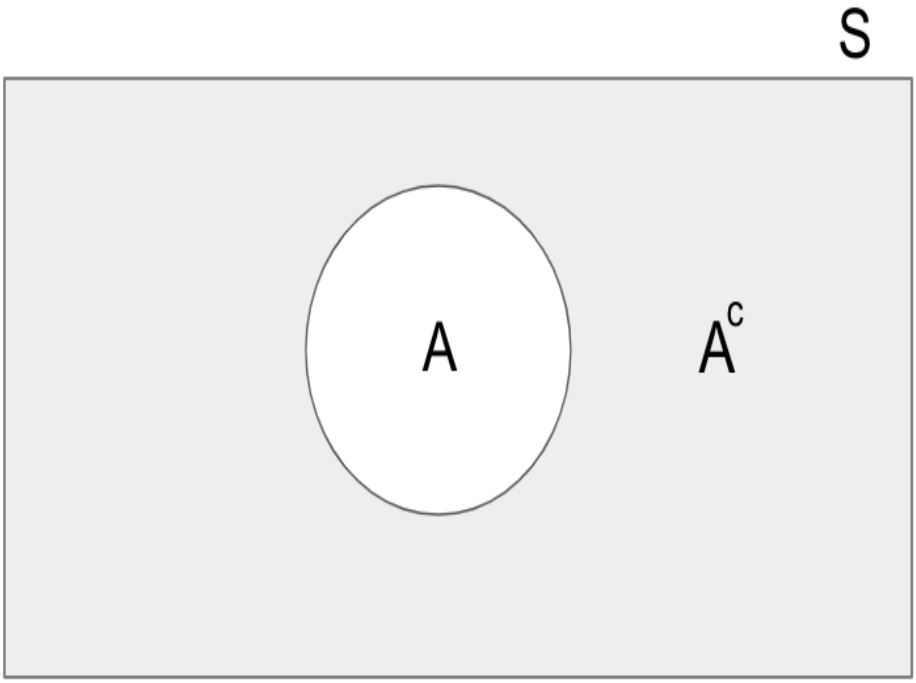


Figure 2-1. Figure 1: We see here how the event A and its complement interact to form the entire set of possibilities, S. The complement simply defines all the possibilities not originally in A.

3. Let E_1 and E_2 be two events, where E_1 is a subset (not necessarily strict) of E_2. The third tenet is that $P(E_1) \leq P(E_2)$. This, again, shouldn't be too surprising - the second event has at least as many outcomes as the first event and all the outcomes the first event has since the second is a superset of the first. If this tenet were not true, that would imply the existence of outcomes with negative probability, which is impossible from our definitions.

4. The fourth and last tenet of probability is the principle of inclusion and exclusion, which states that $P(A \cup B) = P(A) + P(B) - P(A \cap B)$. For those not familiar with this terminology, the $\cup$ denotes the *union* of the two events, a set operation that takes the two events and returns an event that contains all elements from the two original sets. The $\cap$, or *intersection,* is a set operation that returns an event which contains all elements belonging to

both of the two original sets. The idea behind the equality presented is that by just naively summing the probabilities of *A* and *B,* we double count the elements that belong to both sets. Thus, to accurately obtain the probability of the union, we must subtract the probability of the intersection. In Figure 2 we show two events and what their intersection would look like physically, while the union is all the outcomes in the combined area of the events:

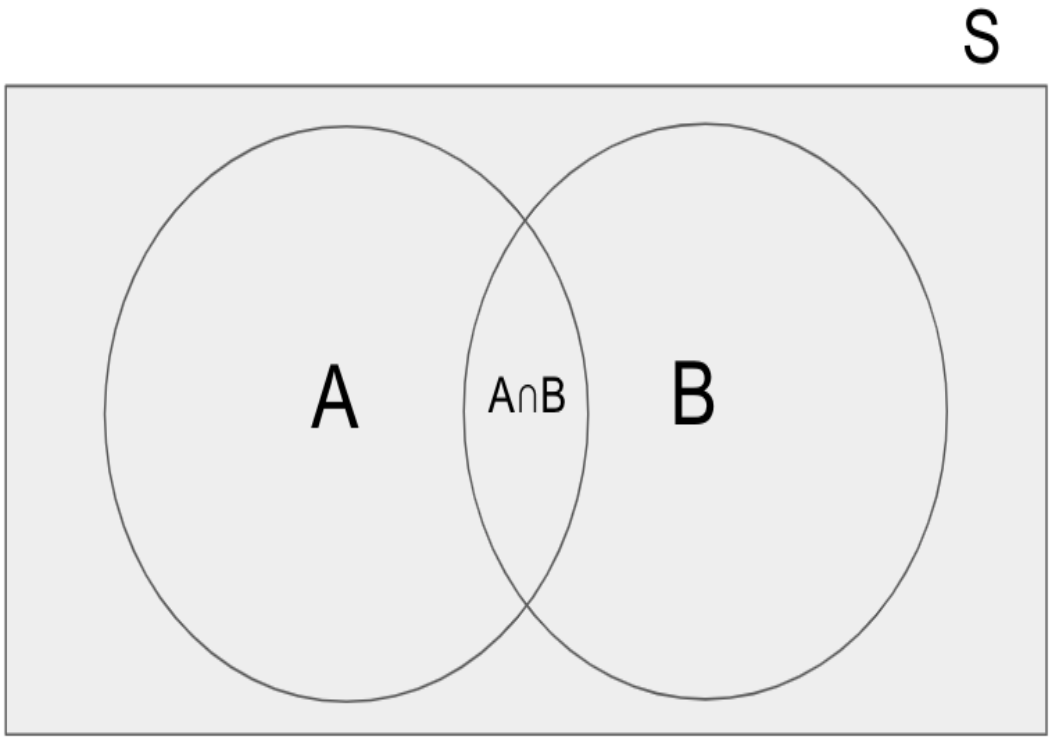


Figure 2-2. The middle sliver labeled as the intersection is the overlap between the two sets and contains all the outcomes that are in both sets. The union can be seen as all the events in the combined area of the two circles - thus if we were to just add their probabilities naively, we would double count all the outcomes in the middle sliver.

These tenets of probability find their way in everything that has to do with the field. For example, in deep learning, most of our problems fall into one of two categories: *regression* and *classification*. In the latter, we train a neural model that can predict the likelihood that the input belongs to one of a discrete number of classes. The famous MNIST digits dataset, for example, provides us with pictures of digits and associated numerical labels in the range of 0 through 9. Our objective is to build a *classifier* that can take in this picture and

return the most likely label as its guess. This is naturally formulated as a problem in probability - the classifier produces a probability distribution over the sample space, 0 through 9, for any given input and its best guess is the digit that has the highest probability! How does this relate to our tenets? Since the classifier is producing a probability distribution, it must follow the tenets. For example, the probabilities associated with each digit must sum to one - a quick back of the envelope check to ensure the model isn't buggy. In the next section, we cover probabilities where we are initially given relevant information that affects our beliefs and how to use that information.

Conditional Probability

Knowing information often changes our beliefs, and by consequence, our probabilities. Going back to our classic dice example, we may roll the dice thinking that it's fair while in reality there's a hidden weight at the dice's core, making it more likely to land up a number greater than three. As we roll the dice we of course start to notice this pattern, and our belief regarding the dice's fairness starts to shift. This is at the core of conditional probability itself. Instead of thinking simply about $P(biased)$ or $P(fair)$, we have to think about probabilities like $P(biased|information)$ instead. This quantity, which we term a *conditional probability,* is spoken as "the probability the dice is biased *given* the information we've seen". How do we think about such probabilities intuitively? For starters, we must imagine that we are now in a different universe than the one we started in. The new universe is one that incorporates the information we've seen since the start of the experiment, e.g. our past dice rolls. Going back to our MNIST example, the probability distribution the trained neural net produces is actually a conditional probability distribution! The probability the input image is zero, for example, can be seen as *P(0|input).* In plain English, we want to find the probability of a zero given all of the pixels that make up the specific input image we fed into our neural net. Our new universe is the universe in which the input pixels have taken on this specific configuration of values. This is distinct from simply looking at *P(0),* the probability of returning a zero, which we can think about in terms of prior belief. Without any knowledge of the input pixel

configuration, we'd have no reason to believe that the possibility of returning a zero is any more or less likely than that of any other digit!

Sometimes, seeing certain information does not change our probabilities - we call this property *independence.* For example, Tom Brady may have thrown a touchdown pass after the third roll of our experiment, but incorporating that information into our new universe should (hopefully!) have no impact on the likelihood of the dice being biased. We state this independence property as *P(biased|Tom Brady throws a touchdown pass) = P(biased).* Note that any two events E_1 and E_2 that satisfy this property are independent. Perhaps slightly more counterintuitively, if it happens to be the case that all of our dice rolls so far don't numerically change our prior belief regarding the dice's fairness (maybe the dice rolls so far have shown up evenly across one through six and our initial prior belief was that the dice was fair), we'd still say that these events are independent. Finally, note that independence is symmetric - if $P(E_1|E_2) = P(E_1)$, then it is also the case that $P(E_2|E_1) = P(E_2)$.

In the last section, we introduced intersection and union notation. It turns out we can break down the intersection operation into a product of probabilities! We have the following equality: $P(E_1 \cap E_2) = P(E_1|E_2) * P(E_2)$. Let's break down the intuition here. On the left hand side, we have the probability that both events E_1 and E_2 have occurred. On the right hand side, we have the same idea, but expressed slightly differently. In the universe where both events have occurred, one way to arrive in this universe is to first have E_2occur, followed by E_1. Porting this intuition into mathematical terms, we must first find the probability that E_2 has occurred followed by the probability that E_1 has occurred in the universe where E_2 has already occurred. How do we combine these two probabilities? Intuitively, it makes sense that we multiply them - we must have both events occur, the first unconditionally and the second in the universe where the first has already occurred. Note that the order of these events doesn't really matter, as both paths get us to the same universe. So, more completely,
$P(E_1 \cap E_2) = P(E_1|E_2) * P(E_2) = P(E_2|E_1) * P(E_1)$

However, some of these paths make much more physical sense than others. For example, if we think of E_1 as the event where someone

contracts a disease, and E_2 as the event that the patient shows symptoms of the disease, the path in which the patient contracts the disease and then shows symptoms makes much more physical sense than the reverse.

In the case where the two events are independent, we have that $P(E_1 \cap E_2) = P(E_1|E_2) * P(E_2) = P(E_1) * P(E_2)$. Hopefully this makes some intuitive sense by now! In the independence scenario, the fact that E_2 has occurred doesn't affect the chances of E_1 occurring; i.e. incorporating this information into the new universe doesn't affect the probability of the next event. In the next section, we cover random variables, which are relevant summaries of events and also have their own probability distributions!

Random Variables

Once again, let's consider the coin flipping experiment. If we flip a coin some finite number of times, natural questions start to arise. How many heads did we encounter during our experiment? How many tails? How many tails until the first head? Every outcome in such an experiment has an answer to each of the listed questions. If we flip a coin say, five times, and we receive the sequence TTHHT, we have seen two heads, three tails, and two tails until the first head. We can think of a *random variable* as a map, or a function, from the sample space to another space, such as the integers in this example. Such a function would take as input the sequence TTHHT and output one of the three answers listed depending on the question we ask. The value that the random variable takes on would be the output associated with result of the experiment. Although random variables are deterministic in that they map a given input to a single output, there are not deterministic in that they also have a distribution associated with their output space! This is due to the inherent randomness in the experiment - depending on the probability of the input outcome, its corresponding output may be more or less likely than other outputs.

One easy way to begin is to just think of this map as an identity function - whatever we flip or roll, its map in the output space is exactly the same as the input. Encoding a heads as a one and a tails as a zero, we can define a random variable representing the coin flip

as whether the coin came up heads, i.e. $C(1) = 1$, where C is our random variable. In the dice scenario, the mapped output is the same as whatever we rolled, i.e. $D(5) = 5$, where D is our random variable. Why should we care about random variables and their distributions? It turns out they play a vital role in deep learning and machine learning as a whole! For example, in a later chapter, we will cover the concept of dropout, a technique for mitigating overfitting in neural networks. The idea of a dropout layer is that, during training, it independently and at random masks every neuron in the previous layer with some probability. This prevents the network from becoming overly dependent on specific connections or subnetworks. We can think of every neuron in the previous layer as representing a coin flip-type experiment. The only difference is that we set the probability of this experiment, rather than a fair coin having by default probability $\frac{1}{2}$ of showing up either side. Each neuron has a random variable X associated with it, with input one if the dropout layer decides to mask it and zero otherwise. X is an identity function from the input space to the output space, i.e. $X(1) = 1$ and $X(0) = 0$.

Random variables, in general, need not be the identity map! Most functions you can think of are valid methods of mapping the input space to an output space where the random variable is defined. For example, if the input space were every possible length n sequence of coin flips, the function could be to count the number of heads in the sequence and square it. Some random variables can even be expressed as functions of other random variables, or a function of a function, as we will cover in a later section. If we again consider the input space of every possible length n sequence of coin flips, the random variable counting the number of heads in the input sequence is the same as counting whether each individual coin flip turned up heads and taking a sum all of those values! In mathematical terms, we say $X = \sum_{i=1}^{n} C_i$, where X is the random variable representing the total number of heads and C_i is the binary random variable associated with the ith coin flip. Back to the dropout example, we can think of the random variable representing the total number of masked out neurons as the sum of binary random variables representing each neuron.

In the future, when we want to refer to the event where the random variable takes on a specific value c (the domain being the output

space we've been referring to), we will concisely write this as $X = c$. We denote the probability that the random variable takes on a specific value as $P(X = c)$, for example. The probability that the random variable takes on any given value in the output space is just the sum of the probabilities of the inputs that map to it! This should make some intuitive sense, as this is basically the fourth tenet of probability where the intersection between any two events is the empty set since all the events we start from are individual, distinct inputs. Note that $P(X)$ itself is also a probability distribution that follows all the basic tenets of probability described in the first section. In the next section, we consider statistics regarding random variables.

Expectation and Variance

As we discussed in the last section, a random variable is a map from input space to output space, where inputs are generated according to some probability distribution. The random variable can be thought of as a relevant summary of the input, and can take on many forms depending on the question we ask. Sometimes, it's useful to understand statistics regarding the random variable. For example, if we flip a coin eight times, how many heads do we expect to see on average? And, of course, we don't see the average number of heads all the time - how much does the number of heads we see tend to vary? The first quantity is what we call the random variable's *expectation*, and the second is the random variable's *variance*.

For a random variable X, we denote its expectation as $\mathbb{E}[X]$. We can think of this as the average value that X takes on, weighted by the probability of each of those outcomes. Mathematically, this is written as $\mathbb{E}[X] = \sum_o o * P(X = o)$. Note that if all outcomes o are equally likely, we get a simple average of all the outcomes. It makes sense to use the probability of the outcome as a weighting, since some outcomes are more likely than others and the average value we observe will be skewed towards such outcomes. For a single fair coin flip, the expected number of heads would be $\sum_{o \in \{0,1\}} o * P(o) = 0 * 0.5 + 1 * 0.5 = 0.5$. In other words, we'd expect to see half of a head for any given fair coin flip! Of course, this makes no physical sense in that we could never possibly flip half

of a head, but this gives you an idea of proportions we'd expect to see over a long run experiment. Going back to our example of length *n* sequences of coin flips, let's try to find the expected number of heads in such a sequence. We have *n+1* possible number of heads, and according to our formula, we'd need to find the probability of attaining each possible number to use as our weights. Mathematically, we'd need to compute $\sum_{x \in \{0,\ldots,n\}} x * P(X = x)$, where X is the random variable representing the total number of heads. However, as n gets larger and larger, performing this calculation starts to become more and more complicated.

Instead, let's denote X_i as the binary random variable for the *i*th coin flip and use the observation we made in the last section of being able to break up the total number of heads into a sum over heads/tails for all the individual coin flips. Since we know $X = X_1 + X_2 + \ldots + X_n$, we can also say that $\mathbb{E}[X] = \mathbb{E}[X_1 + X_2 + \ldots + X_n]$. How does making this substitution make our problem easier? We now introduce the concept of *linearity of expectation,* which states we can break up the right hand side into the sum $\mathbb{E}[X_1] + \mathbb{E}[X_2] + \ldots + \mathbb{E}[X_n]$. We know that the expected number of heads for each flip is 0.5, so the expected number of heads in a sequence of *n* flips is just 0.5**n!* This is much simpler than going down the previous route, as this approach's difficulty does not scale with the number of flips. Let's go over the simplification we made in a bit more detail. Mathematically, if we have any two independent random variables A and B:

$$\mathbb{E}[A + B] = \sum_{a,b} (a + b) * P(A = a, B = b)$$
$$= \sum_{a,b} (a + b) * P(A = a) * P(B = b)$$
$$= \sum_{a,b} a * P(A = a) * P(B = b) + b * P(A = a) * P(B = b)$$
$$= \sum_{a,b} a * P(A = a) * P(B = b) + \sum_{a,b} b * P(A = a) * P(B = b)$$
$$= \sum_{a} a * P(A = a) \sum_{b} P(B = b) + \sum_{b} b * P(B = b) \sum_{a} P(A = a)$$
$$= \sum_{a} a * P(A = a) + \sum_{b} b * P(B = b)$$
$$= \mathbb{E}[A] + \mathbb{E}[B]$$

Note that we made the independence assumption we talked about earlier in the chapter here when we broke up the probability of the

event $A = a$ and the event $B = b$ into a product of the two individual probabilities. The rest of the derivation doesn't require additional assumptions, so we recommend working through the algebra on your own. Although we won't show this for the dependent case, linearity of expectation also holds for dependent random variables!

Going back to the dropout example, the expectation of the total number of masked neurons can be broken up into a sum of expectations over each neuron. The expected number of masked neurons, similarly to the expected number of heads in a sequence of coin flips, is *p*n,* where *p* is the probability of being masked (and the expectation of each individual binary random variable representing a neuron) and *n* is the number of neurons.

As we mentioned at the beginning of this section, we don't always see the expected number of occurrences of an event in every repetition of an experiment. In some cases, such as the expected number of heads in a single, fair coin flip from earlier, we never see it! In the remainder of this section, we will quantify the average deviation, or variance, from the expected value we see in repetitions of an experiment.

We define the variance, or Var(X), as $\mathbb{E}\Big[(X-\mu)^2\Big]$, where we let $\mu = \mathbb{E}[X]$. In plain English, this measure represents the average squared difference between the value X takes on and its expectation. Note that $(X-\mu)^2$ itself is also a random variable since it is a function of a function (X), which is still a function! Although we won't get into too much detail about why we use this formula in particular, we encourage the reader to think about why we don't use a formula such as $\mathbb{E}[X-\mu]$instead. To obtain a slightly simpler form for the variance, we can perform the following simplification:

$$\mathbb{E}\Big[(X-\mu)^2\Big] = \mathbb{E}\big[X^2 - 2\mu X + \mu^2\big]$$

$$= \mathbb{E}\big[X^2\big] - \mathbb{E}[2\mu X] + \mathbb{E}\big[\mu^2\big]$$

$$= \mathbb{E}\big[X^2\big] - 2\mu\mathbb{E}[X] + \mu^2$$

$$= \mathbb{E}\big[X^2\big] - 2\mathbb{E}[X]^2 + \mathbb{E}[X]^2$$

$$= \mathbb{E}\big[X^2\big] - \mathbb{E}[X]^2$$

Let's take a moment to go through each of these steps. In the first step, we fully express the random variable as all of its component terms via classic binomial expansion. In the second step, we perform linearity of expectation to break out the component terms into their own, individual expectations. In the third step, we note that μ, or $\mathbb{E}[X]$, and its square are both constants and thus can be pulled out of the surrounding expectation. They are constants since they are not a function of the value X takes on and are instead evaluated using the entire domain (the set of values X can take on). Constants can be seen as random variables that can only take on one value, which is the constant itself. Thus, their expectations, or the average value the random variable takes on, is the constant itself since we always see the constant. The final steps are algebraic manipulations that bring us to the simplified result. Let's use this formula to find the variance of the binary random variable representing a single neuron under dropout, and p is the probability of the neuron being masked out:

$$\mathbb{E}\left[X^2\right] - \mathbb{E}[X]^2 = \sum_{x \in 0,1} x^2 * P(X = x) - \left(\sum_{x \in 0,1} x * P(X = x)\right)^2$$

$$= \sum_{x \in 0,1} x^2 * P(X = x) - p^2$$

$$= p - p^2$$

$$= p(1 - p)$$

These simplifications should make some intuitive sense by now. We know from our earlier subsection on expectations that the expectation of the binary random variable representing a neuron is just p, and the rest is algebraic simplifications. We highly encourage the reader to work through these derivations on their own! As we start to think about the random variable representing the number of masked neurons in the entire layer, we naturally ask the question of whether there exists a similar linearity property for variance as there does for expectation. Unfortunately, the property does not hold in general:

$$Var(A + B) = \mathbb{E}\Big[(A + B)^2\Big] - \mathbb{E}[A + B]^2$$

$$= \mathbb{E}\left[A^2 + 2 * A * B + B^2\right] - (\mathbb{E}[A] + \mathbb{E}[B])^2$$

$$= \mathbb{E}\left[A^2\right] + 2\mathbb{E}[A * B] + \mathbb{E}\left[B^2\right] - \mathbb{E}[A]^2 - 2\mathbb{E}[A]\mathbb{E}[B] - \mathbb{E}[B]^2$$

$$= \mathbb{E}\left[A^2\right] - \mathbb{E}[A]^2 + \mathbb{E}\left[B^2\right] - \mathbb{E}[B]^2 + 2\mathbb{E}[A * B] - 2\mathbb{E}[A]\mathbb{E}[B]$$
$$= Var(A) + Var(B) + 2(\mathbb{E}[A * B] - \mathbb{E}[A]\mathbb{E}[B])$$
$$= Var(A) + Var(B) + 2Cov(A, B)$$

As we can see from the last line, the final term in the expression, which we call the *covariance* between the two random variables, ruins our hope for linearity. However, covariance is another key concept in probability - the intuition for covariance is that it measures the dependence between two random variables. As one random variable more completely determines the value of another random variable (think of A as the number of heads in a sequence of coin flips and B as the number of tails in the same sequence of coin flips), the magnitude of the covariance increases. Thus, it stands to reason that if A and B are independent random variables, the covariance between them should be zero, and linearity should hold in this special case! We highly encourage the reader to work through the math and show this on their own.

Back to the dropout example, the variance of the total number of masked neurons can be broken up into a sum of variances over each neuron, since each neuron is masked independently. The variance of the number of masked neurons is *p(1-p)*n,* where *p(1-p)* is the variance for any given neuron and *n* is the number of neurons. Expectation and variance in dropout allow us to understand more deeply what we expect to see when applying such a layer in a deep neural network.

Bayes' Theorem

Going back to our discussion on conditional probability, we noted that the probability of intersection between two events could be written as a product of a conditional distribution and a distribution over a single event. Let's translate this into the language of random variables, now that we have introduced this new terminology. We denote A to be one random variable, and B to denote a second. Let a be a value that A can take on, and b be a value that B can take on. The analogy to the intersection operation for random variables is the *joint probability distribution P(A=a,B=b),* which denotes the event where $A=a$ and $B=b$. We can think of $A=a$ and $B=b$ as individual

events, and when we write *P(A=a,B=b)*, we are considering the probability that both events have occurred, i.e. their intersection $P(A = a \cap B = b)$. Note that we generally write the joint probability distribution as *P(A,B),* since this encompasses all possible joint settings of the r.v.'s *A* and B.

We mentioned earlier that intersection operations could be written as the product of a conditional distribution and a distribution over a single event. Re-writing this in the format for r.v.'s, we have *P(A=a,B=b) = P(A=a|B=b)P(B=b)*. And more generally, considering all possible joint settings of the two r.v.'s, we have *P(A,B) = P(A|B)P(B).* We also discussed how there also always exists a second way of writing this joint distribution as a product: *P(A=a,B=b) = P(B=b|A=a)P(A=a),* and more generally*, P(A,B) = P(B|A)P(A).* We noted that, sometimes, one of these paths makes more sense than the others. For example, in the case where symptoms are represented by *A,* and disease is represented by *B,* the path in which *B* takes on a value *b* and then *A* takes on a value *a* in the universe where *B=b* has occurred makes much more sense than the reverse since, biologically, people contract a disease first and only then show symptoms for that disease!

However, this doesn't mean that the reverse isn't useful. It is almost universally the case that people show up to the hospital with symptoms, and doctors must try to infer the most likely disease from these symptoms. *Bayes' Theorem* gives us a way of calculating the probability of a disease given the observed symptoms. Since the same joint probability distribution can be written in the two manners mentioned in the above paragraph, we have the following equality:

$$P(B|A) = \frac{P(A|B)P(B)}{P(A)}$$

If *B* represents disease, while *A* represents symptoms, this gives us a method for computing the likelihood of any disease given the observed symptoms. Let's analyze the right hand side to see if the equality also makes intuitive sense. The likelihood of symptoms given the disease times the likelihood of the disease is just the joint distribution, which makes sense as the numerator here. The denominator is the likelihood of seeing those symptoms, which can also be expressed as a sum of the numerator over all possible diseases:

$P(A) = \sum_b P(A, B = b)$

In more concise terms, we have:

$P(B = b_{query}|A) = \frac{P(B=b_{query},A)}{\sum_b P(B=b,A)}$

Bayes' Theorem is a very valuable application of probability in the real-world, especially in the case of disease prediction. Additionally, if we replace the r.v. for symptoms with an r.v. representing the result of a test for a specific disease and the r.v. over all diseases with an r.v. over the specific disease, we can infer the likelihood of actually having a specific disease given a positive test for it using Bayes' Theorem. This is a very common problem in most hospitals, and is especially relevant to epidemiology given the recent outbreak of Covid-19.

Entropy, Cross-Entropy, and KL Divergence

Section content goes here

Summary

In this chapter we covered the fundamentals of probability, first building the intuition behind the basics of probability distributions and then moving to relevant applications of probability such as conditional probability, random variables, expectation, and variance. We saw the applications of probability in deep learning, such as how a neural net parametrizes a probability distribution during classification tasks and how we can quantify the mathematical properties of dropout, a regularization technique in neural nets. We hope that this introduction puts the rest of the book in perspective and allows you to more rigorously understand future concepts.

Chapter 3. Deep Reinforcement Learning

Nicholas Locascio[1]

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 14th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at mpotter@oreilly.com.

In this chapter, we'll discuss reinforcement learning, which is a branch of machine learning that deals with learning via interaction and feedback. Reinforcement learning is essential to building agents that can not only perceive and interpret the world, but also take action and interact with it. We will discuss how to incorporate deep neural networks into the framework of reinforcement learning and discuss recent advances and improvements in this field.

Deep Reinforcement Learning Masters Atari Games

The application of deep neural networks to reinforcement learning had a major breakthrough in 2014, when the London startup DeepMind astonished the machine learning community by unveiling a deep neural network that could learn to play Atari games with superhuman skill. This network, termed a *Deep Q-Network* (DQN) was the first large-scale successful application of reinforcement learning with deep neural networks. DQN was so remarkable because the same architecture, without any changes, was capable of learning 49 different Atari games, despite each game having different rules, goals, and gameplay structure. To accomplish this feat, DeepMind brought together many traditional

ideas in reinforcement learning while also developing a few novel techniques that proved key to DQN's success. Later in this chapter we will implement DQN, as it is described in the *Nature* paper "Human-level control through deep reinforcement learning."[2] But first, let's take a dive into reinforcement learning (Figure 3-1).

Figure 3-1. A deep reinforcement learning agent playing Breakout. This image is from the OpenAI Gym[3] DQN agent that we build in this chapter.

What Is Reinforcement Learning?

Reinforcement learning, at its essentials, is learning by interacting with an environment. This learning process involves an *actor*, an *environment*, and a *reward signal*. The actor chooses to take an action in the environment, for which the actor is rewarded accordingly. The way in which an actor chooses actions is called a *policy*. The actor wants to increase the reward it receives, and so must learn an optimal policy for interacting with the environment (Figure 3-2).

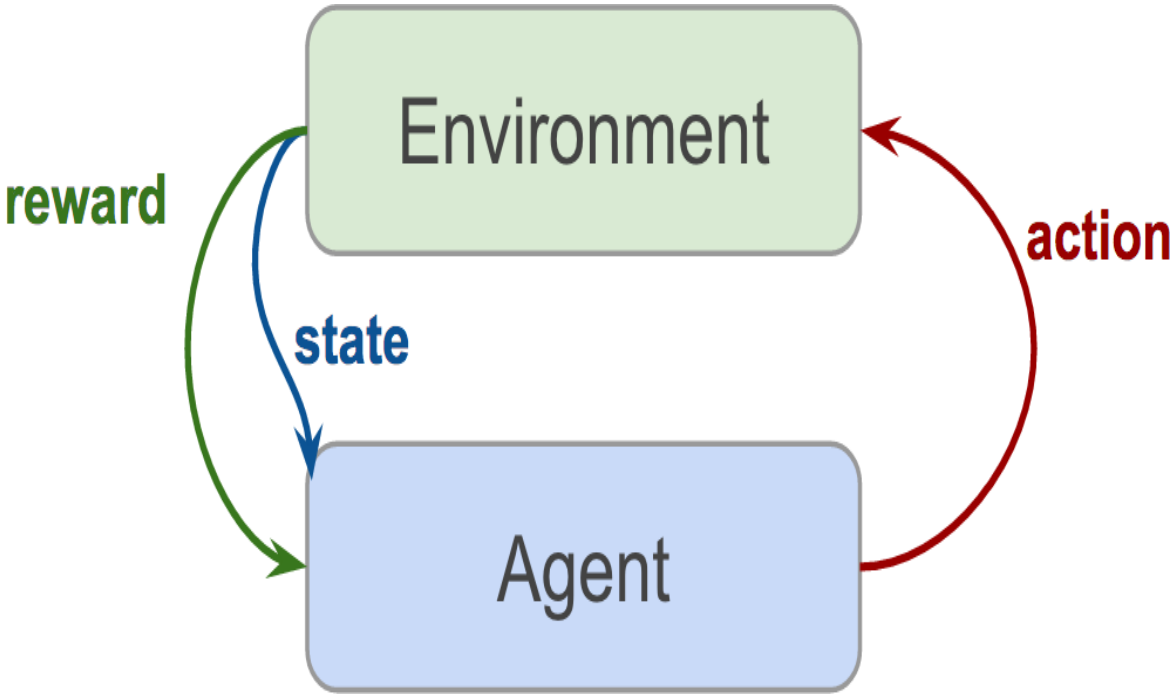


Figure 3-2. Reinforcement learning setup

Reinforcement learning is different from the other types of learning that we have covered thus far. In traditional supervised learning, we are given data and labels, and are tasked with predicting labels given data. In unsupervised learning, we are given just data and are tasked with discovering underlying structure in this data. In reinforcement learning, we are given neither data nor labels. Our learning signal is derived from the rewards given to the agent by the environment.

Reinforcement learning is exciting to many in the artificial intelligence community because it is a general-purpose framework for creating intelligent agents. Given an environment and some rewards, the agent learns to interact with that environment to maximize its total reward. This type of learning is more in line with how humans develop. Yes, we can build a pretty good model to classify dogs from cats with extremely high accuracy by training on thousands of images. But you won’t find this

approach used in any elementary schools. Humans interact with their environment to learn representations of the world which they can use to make decisions.

Furthermore, reinforcement learning applications are at the forefront of many cutting-edge technologies including self-driving cars, robotic motor control, game playing, air-conditioning control, ad-placement optimization, and stock market trading strategies.

As an illustrative exercise, we'll be tackling a simple reinforcement learning and control problem called pole-balancing. In this problem, there is a cart with a pole that is connected by a hinge, so the pole can swing around the cart. There is an agent that can control the cart, moving it left or right. There is an environment, which rewards the agent when the pole is pointed upward, and penalizes the agent when the pole falls over (Figure 3-3).

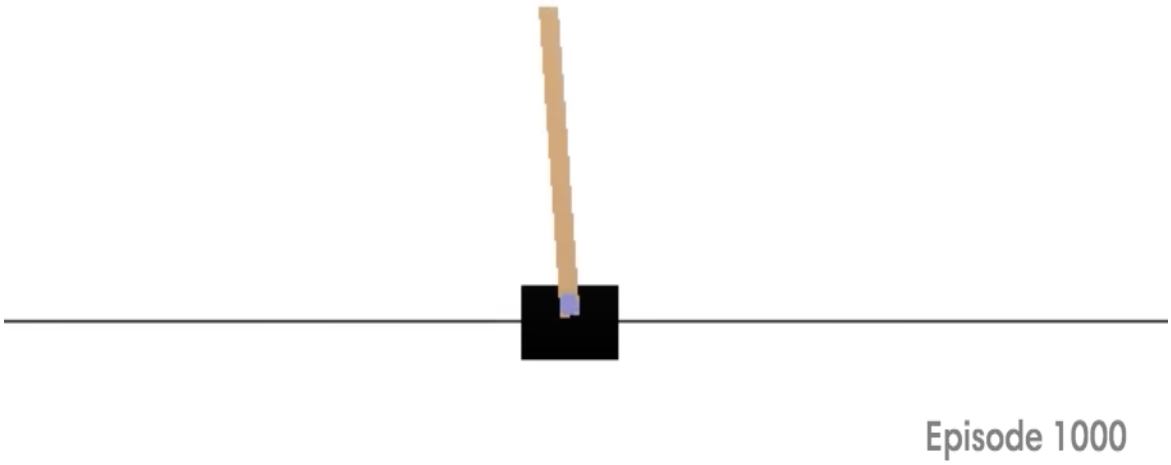


Figure 3-3. A simple reinforcement learning agent balancing a pole. This image is from our OpenAI Gym Policy Gradient agent that we build in this chapter.

Markov Decision Processes (MDP)

Our pole-balancing example has a few important elements, which we formalize as a *Markov Decision Process* (MDP). These elements are:

State

The cart has a range of possible places on the x-plane where it can be. Similarly, the pole has a range of possible angles.

Action

The agent can take action by moving the cart either left or right.

State Transition

When the agent acts, the environment changes—the cart moves and the pole changes angle and velocity.

Reward

If an agent balances the pole well, it receives a positive reward. If the pole falls, the agent receives a negative reward.

An MDP is defined as the following:

- S, a finite set of possible states
- A, a finite set of actions
- $P\left(r, s^{'} \middle| s, a\right)$, a state transition function
- R, reward function

MDPs offer a mathematical framework for modeling decision-making in a given environment (Figure 3-4).

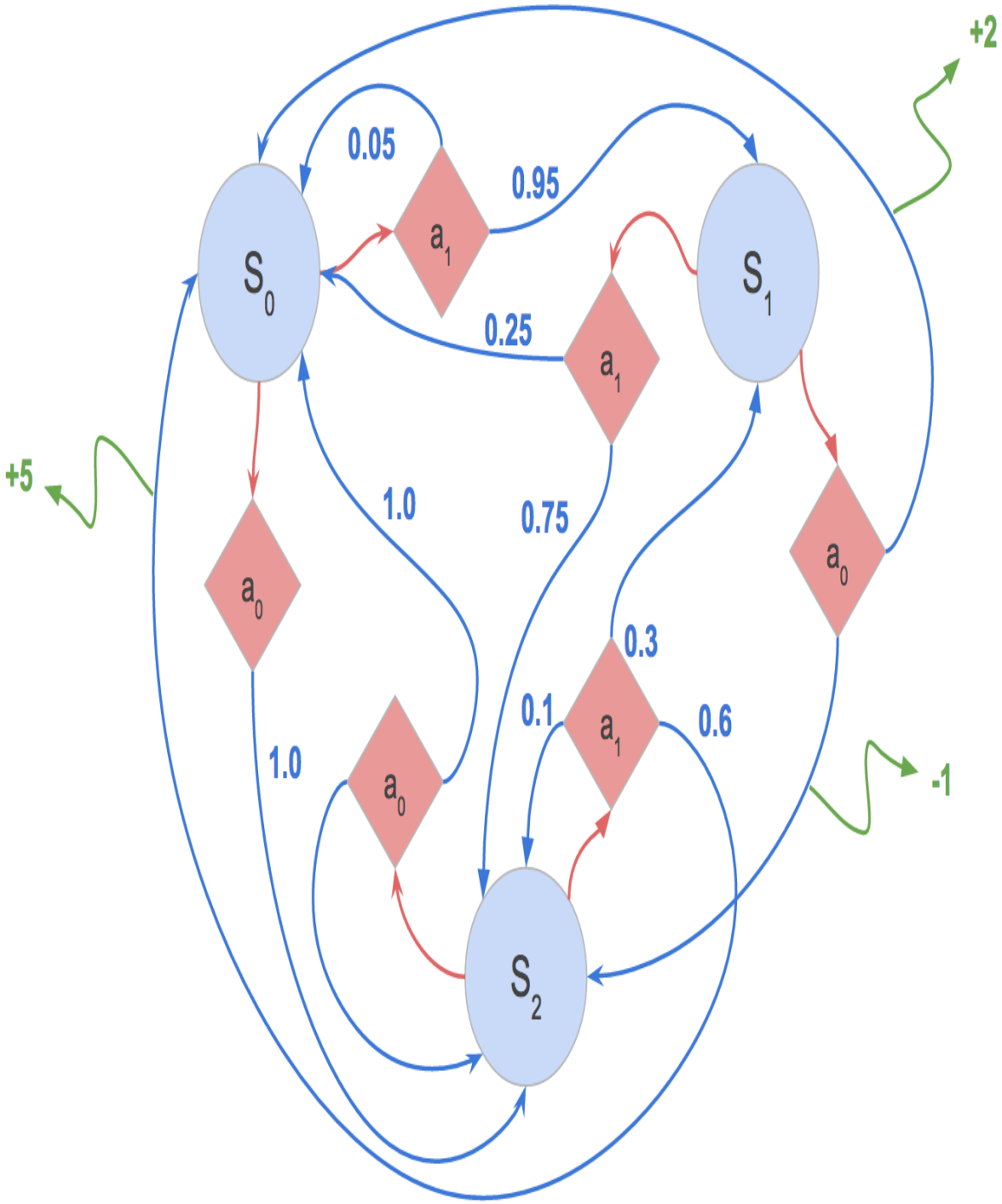


Figure 3-4. An example of an MDP. Blue circles represent the states of the environment. Red diamonds represent actions that can be taken. The edges from diamonds to circles represent the transition from one state to the next. The numbers along these edges represent the probability of taking a certain action. The numbers at the end of the green arrows represent the reward given to the agent for making the given transition.

As an agent takes action in an MDP framework, it forms an *episode*. An episode consists of series of tuples of states, actions, and rewards. Episodes run until the environment reaches a terminal state, like the

“Game Over” screen in Atari games, or when the pole hits the ground in our pole-cart example. The following equation shows the variables in an episode:

$(s_0, a_0, r_0), (s_1, a_1, r_1), \ldots (s_n, a_n, r_n)$

In pole-cart, our environment state can be a tuple of the position of the cart and the angle of the pole, like so: (x_{cart}, θ_{pole}).

Policy

MDP’s aim is to find an optimal policy for our agent. *Policies* are the way in which our agent acts based on its current state. Formally, policies can be represented as a function π that chooses the action a that the agent will take in state s.

The objective of our MDP is to find a policy to maximize the expected future return:

$\max_\pi \quad E[R_0 + R_1 + \ldots R_t | \pi]$

In this objective, R represents the *future return* of each episode. Let’s define exactly what future return means.

Future Return

Future return is how we consider the rewards of the future. Choosing the best action requires consideration of not only the immediate effects of that action, but also the long-term consequences. Sometimes the best action actually has a negative immediate effect, but a better long-term result. For example, a mountain-climbing agent that is rewarded by its altitude may actually have to climb downhill to reach a better path to the mountain’s peak.

Therefore, we want our agents to optimize for *future return*. In order to do that, the agent must consider the future consequences of its actions. For example, in a game of Pong, the agent receives a reward when the ball passes into the opponent’s goal. However, the actions responsible for this reward (the inputs that position the racquet to strike scoring hit) happen many time steps before the reward is received. The reward for each of those actions is delayed.

We can incorporate delayed rewards into our overall reward signal by constructing a *return* for each time step that takes into account future rewards as well as immediate rewards. A naive approach for calculating *future return* for a time step may be a simple sum like so:

$R_t = \sum_{k=0}^{T} r_{t+k}$

We can calculate all returns, R, where $R = \{R_0, R_1, \ldots R_i, \ldots R_n\}$ with the following code:

```
def calculate_naive_returns(rewards):
""" Calculates a list of naive returns given a
    list of rewards."""
    total_returns = np.zeros(len(rewards))
    total_return = 0.0
    for t in range(len(rewards), 0):
        total_return = total_return + reward
        total_returns[t] = total_return
    return total_returns
```

This naive approach successfully incorporates future rewards so the agent can learn an optimal global policy. This approach values future rewards equally to immediate rewards. However, this equal consideration of all rewards is problematic. With infinite time steps, this expression can diverge to infinity, so we must to find a way to bound it. Furthermore, with equal consideration at each time step, the agent can optimize for a very future reward, and we would learn a policy that lacks any sense of urgency or time sensitivity in pursuing its rewards.

Instead, we should value future rewards slightly less in order to force our agents to learn to get rewards quickly. We accomplish this with a strategy called *discounted future return.*

Discounted Future Return

To implement discounted future return, we scale the reward of a current state by the discount factor, γ, to the power of the current time step. In this way, we penalize agents that take many actions before receiving positive reward. Discounted rewards bias our agent to prefer receiving reward in immediate future, which is advantageous to learning a good policy. We can express the reward as follows:

$R_t = \sum_{k=0}^{T} \gamma^t r_{t+k+1}$

The discount factor, γ, represents the level of discounting we want to achieve and can be between 0 and 1. High γ means little discounting, low γ provides much discounting. A typical γ hyperparameter setting is between 0.99 and 0.97.

We can implement discounted return like so:

```
def discount_rewards(rewards, gamma=0.98):
    discounted_returns = [0 for _ in rewards]
```

```
discounted_returns[-1] = rewards[-1]
for t in range(len(rewards)-2, -1, -1): # iterate backwards
    discounted_returns[t] = rewards[t] +
      discounted_returns[t+1]*gamma
return discounted_returns
```

Explore Versus Exploit

Reinforcement learning is fundamentally a trial-and-error process. In such a framework, an agent afraid to make mistakes can prove to be highly problematic. Consider the following scenario. A mouse is placed in the maze shown in Figure 3-5. Our agent must control the mouse to maximize reward. If the mouse gets the water, it receives a reward of +1; if the mouse reaches a poison container (red), it receives a reward of -10; if the mouse gets the cheese, it receives a reward of +100. Upon receiving reward, the episode is over. The optimal policy involves the mouse successfully navigating to the cheese and eating it.

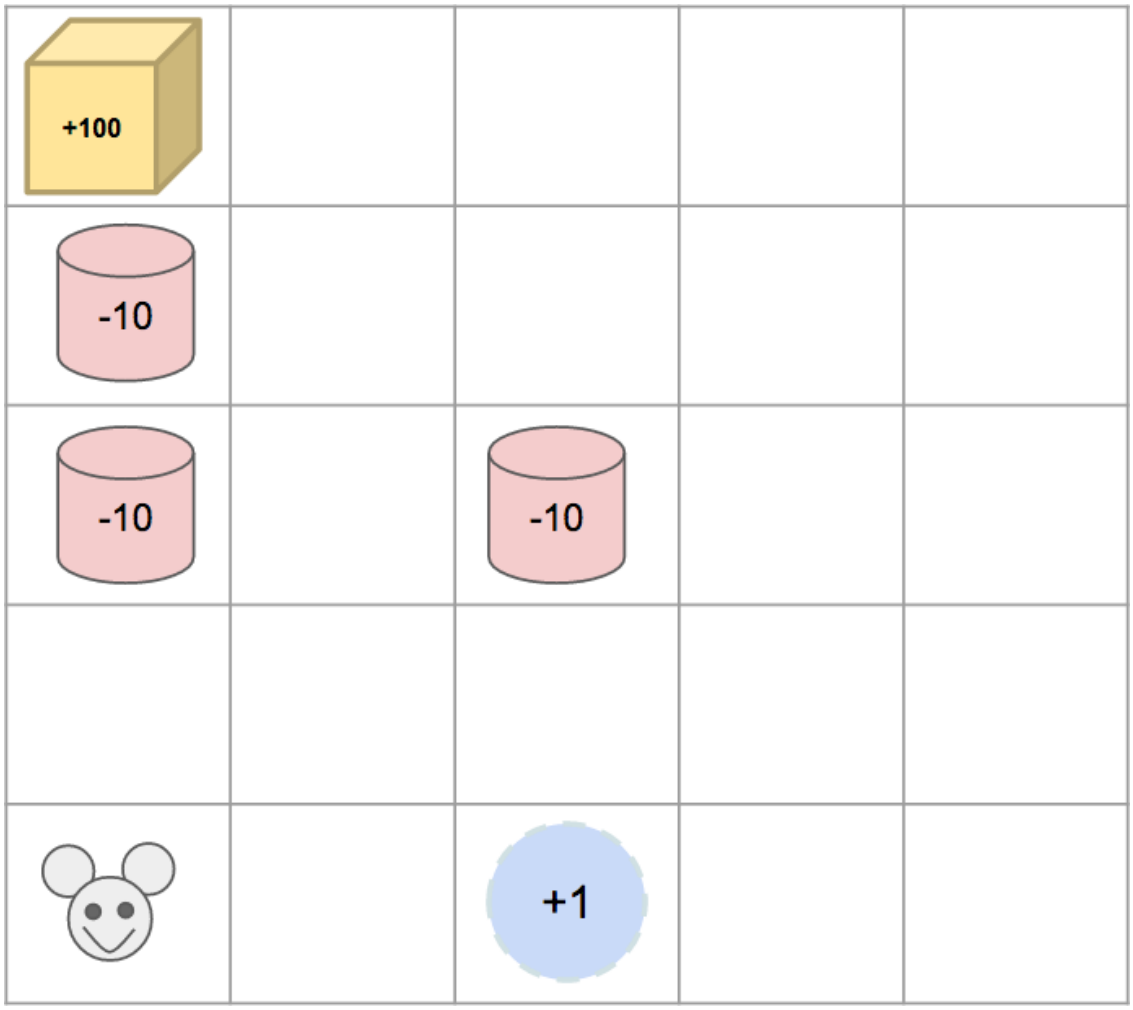


Figure 3-5. A predicament that many mice find themselves in

In the first episode, the mouse takes the left route, steps on a trap, and receives a -10 reward. In the second episode, the mouse avoids the left path, since it resulted in such a negative reward, and drinks the water immediately to its right for a +1 reward. After two episodes, it would seem that the mouse has found a good policy. It continues to follow its learned policy on subsequent episodes and achieves the moderate +1 reward reliably. Since our agent utilizes a greedy strategy—always choosing the model's best action—it is stuck in a policy that is a *local maximum*.

To prevent such a situation, it may be useful for the agent to deviate from the model's recommendation and take a suboptimal action in order to *explore* more of the environment. So instead of taking the immediate right turn to *exploit* the environment to get water and the reliable +1 reward, our agent may choose to take a left turn and venture into more treacherous areas in search of a more optimal policy. Too much exploration, and our agent fails to optimize any reward. Not enough exploration can result in our agent getting stuck in local minimum. This balance of *explore versus exploit* is crucial to learning a successful policy.

ϵ-Greedy

One strategy for balancing the explore-exploit dilemma is called *e-Greedy*. e-Greedy is a simple strategy that involves making a choice at each step to either take the agent's top recommended action or take a random action. The probability that the agent takes a random action is the value known as ϵ:

We can implement ϵ-Greedy like so:

```
def epsilon_greedy_action(action_distribution, epsilon=1e-1):
    if random.random() < epsilon:
        return np.argmax(np.random.random(
           action_distribution.shape))
    else:
        return np.argmax(action_distribution)
```

Annealed ϵ-Greedy

When training a reinforcement learning model, oftentimes we want to do more exploring in the beginning since our model knows little of the world. Later, once our model has seen much of the environment and learned a good policy, we want our agent to trust itself more to further optimize its policy. To accomplish this, we cast aside the idea of a fixed ϵ , and instead anneal it over time, having it start low and increase by a factor after each training episode. Typical settings for annealed e-

Greedy scenarios include annealing from 0.99 to 0.1 over 10,000 scenarios. We can implement annealing like so:

```
def epsilon_greedy_action_annealed(action_distribution,
                                   percentage,
                                   epsilon_start=1.0,
                                   epsilon_end=1e-2):
    annealed_epsilon = epsilon_start*(1.0-percentage) +
      epsilon_end*percentage
    if random.random() < annealed_epsilon:
        return np.argmax(np.random.random(
          action_distribution.shape))
    else:
        return np.argmax(action_distribution)
```

Policy Versus Value Learning

So far we've defined the setup of reinforcement learning, discussed discounted future return, and looked at the trade-offs of explore versus exploit. What we haven't talked about is how we're actually going to teach an agent to maximize its reward. Approaches to this fall into two broad categories: *policy learning* and *value learning*. In policy learning, we are directly learning a policy that maximizes reward. In value learning, we are learning the value of every state + action pair. If you were trying to learn to ride a bike, a policy learning approach would be to think about how pushing on the right pedal while you were falling to the left would course-correct you. If you were trying to learn to ride a bike with a value learning approach, you would assign a score to different bike orientations and actions you can take in those positions. We'll be covering both in this chapter, so let's start with policy learning.

Policy Learning via Policy Gradients

In typical supervised learning, we can use stochastic gradient descent to update our parameters to minimize the loss computed from our network's output and the true label. We are optimizing the expression:

$$\text{argmin}_\theta \sum_i \log p(y_i \mid x_i; \theta)$$

In reinforcement learning, we don't have a true label, only reward signals. However, we can still use SGD to optimize our weights using something called *policy gradients*.[4] We can use the actions the agent takes, and the returns associated with those actions, to encourage our model weights to take good actions that lead to high reward, and to avoid bad ones that lead to low reward. The expression we optimize for is:

$$\text{argmin}_\theta -\sum_i R_i \log p(y_i \mid x_i; \theta)$$

where y_i is the action taken by the agent at time step t and where R_i is our discounted future return**.** A In this way, we scale our loss by the value of our return, so if the model chose an action that led to negative return, this would lead to greater loss. Furthermore, if the model is very confident in that bad decision, it would get penalized even more, since we are taking into account the log probability of the model choosing that action. With our loss function defined, we can apply SGD to minimize our loss and learn a good policy.

Pole-Cart with Policy Gradients

We're going to implement a policy-gradient agent to solve pole-cart, a classic reinforcement learning problem. We will be using an environment from the OpenAi Gym created just for this task.

OpenAI Gym

The OpenAI Gym is a Python toolkit for developing reinforcement agents. OpenAI Gym provides an easy-to-use interface for interacting with a variety of environments. It contains over 100 open-source implementations of common reinforcement learning environments. OpenAI Gym speeds up development of reinforcement learning agents by handling everything on the environment simulation side, allowing researchers to focus on their agent and learning algorithms. Another benefit of OpenAI Gym is that researchers can fairly compare and evaluate their results with others because they can all use the same standardized environment for a task. We'll be using the pole-cart environment from OpenAI Gym to create an agent that can easily interact with this environment.

Creating an Agent

To create an agent that can interact with an OpenAI environment, we'll define a class PGAgent, which will contain our model architecture, model weights, and hyperparameters:

```
class PGAgent(object):

    def __init__(self, session, state_size, num_actions,
        hidden_size, learning_rate=1e-3,
        explore_exploit_setting=
          'epsilon_greedy_annealed_1.0->0.001'):
        self.session = session
        self.state_size = state_size
        self.num_actions = num_actions
```

```
        self.hidden_size = hidden_size
        self.learning_rate = learning_rate
        self.explore_exploit_setting = explore_exploit_setting

        self.build_model()
        self.build_training()

    def build_model(self):
        with tf.variable_scope('pg-model'):
            self.state = tf.placeholder(
                shape=[None, self.state_size],
                dtype=tf.float32)
            self.h0 = slim.fully_connected(self.state,
            self.hidden_size)
            self.h1 = slim.fully_connected(self.h0,
            self.hidden_size)
            self.output = slim.fully_connected(
                self.h1, self.num_actions,
                activation_fn=tf.nn.softmax)
            # self.output = slim.fully_connected(self.h1,
              self.num_actions)

    def build_training(self):
        self.action_input = tf.placeholder(tf.int32,
          shape=[None])
        self.reward_input = tf.placeholder(tf.float32,
          shape=[None])

        # Select the logits related to the action taken
        self.output_index_for_actions = (tf.range(
            0, tf.shape(self.output)[0]) *
              tf.shape(self.output)[1]) +
               self.action_input
        self.logits_for_actions = tf.gather(
            tf.reshape(self.output, [-1]),
            self.output_index_for_actions)

        self.loss = - \
            tf.reduce_mean(tf.log(self.logits_for_actions) *
              self.reward_input)

        self.optimizer = tf.train.AdamOptimizer(
            learning_rate=self.learning_rate)
        self.train_step = self.optimizer.minimize(self.loss)

    def sample_action_from_distribution(
      self, action_distribution,
      epsilon_percentage):
        # Choose an action based on the action probability
        # distribution and an explore vs exploit
        if self.explore_exploit_setting == 'greedy':
            action = greedy_action(action_distribution)
        elif self.explore_exploit_setting ==
          'epsilon_greedy_0.05':
            action = epsilon_greedy_action(action_distribution,
              0.05)
        elif self.explore_exploit_setting ==
          'epsilon_greedy_0.25':
            action = epsilon_greedy_action(action_distribution,
```

```
            0.25)
        elif self.explore_exploit_setting ==
          'epsilon_greedy_0.50':
            action = epsilon_greedy_action(action_distribution,
              0.50)
        elif self.explore_exploit_setting ==
          'epsilon_greedy_0.90':
            action = epsilon_greedy_action(action_distribution,
              0.90)
        elif self.explore_exploit_setting ==
          'epsilon_greedy_annealed_1.0->0.001':
            action = epsilon_greedy_action_annealed(
                action_distribution, epsilon_percentage, 1.0,
                  0.001)
        elif self.explore_exploit_setting ==
          'epsilon_greedy_annealed_0.5->0.001':
            action = epsilon_greedy_action_annealed(
                action_distribution, epsilon_percentage, 0.5,
                  0.001)
        elif self.explore_exploit_setting ==
          'epsilon_greedy_annealed_0.25->0.001':
            action = epsilon_greedy_action_annealed(
                action_distribution, epsilon_percentage, 0.25,
                  0.001)

        return action

    def predict_action(self, state, epsilon_percentage):
        action_distribution = self.session.run(
            self.output, feed_dict={self.state: [state]})[0]
        action = self.sample_action_from_distribution(
            action_distribution, epsilon_percentage)
        return action
```

Building the Model and Optimizer

Lets break down some important functions. In `build_model()` , we define our model architecture as a three-layer neural network. The model returns a layer of three nodes, each representing the model's action probability distribution. In `build_training()`, we implement our policy gradient optimizer. We express our objective loss as we talked about, scaling the model's prediction probability for an action with the return received for taking that action, and summing these all up to form a minibatch. With our objective defined, we can use `tf.AdamOptimizer`, which will adjust our weights according to the gradient to minimize our loss.

Sampling Actions

We define the `predict_action` function, which samples an action based on the model's action probability distribution output. We support the various sampling strategies that we talked about to balance explore

versus exploit, including greedy, epsilon greedy, and epsilon greedy annealing.

Keeping Track of History

We'll be aggregating our gradients from multiple episode runs, so it will be useful to keep track of state, action, and reward tuples. To this end, we implement an episode history and memory:

```
class EpisodeHistory(object):

    def __init__(self):
        self.states = []
        self.actions = []
        self.rewards = []
        self.state_primes = []
        self.discounted_returns = []

    def add_to_history(self, state, action, reward,
      state_prime):
        self.states.append(state)
        self.actions.append(action)
        self.rewards.append(reward)
        self.state_primes.append(state_prime)


class Memory(object):

    def __init__(self):
        self.states = []
        self.actions = []
        self.rewards = []
        self.state_primes = []
        self.discounted_returns = []

    def reset_memory(self):
        self.states = []
        self.actions = []
        self.rewards = []
        self.state_primes = []
        self.discounted_returns = []

    def add_episode(self, episode):
        self.states += episode.states
        self.actions += episode.actions
        self.rewards += episode.rewards
        self.discounted_returns += episode.discounted_returns
```

Policy Gradient Main Function

Lets put this all together in our main function, which will create an OpenAI Gym environment for CartPole, make an instance of our agent, and have our agent interact with and train on the CartPole environment:

```
def main(argv):
    # Configure Settings
    total_episodes = 5000
    total_steps_max = 10000
    epsilon_stop = 3000
    train_frequency = 8
    max_episode_length = 500
    render_start = -1
    should_render = False

    explore_exploit_setting =
      'epsilon_greedy_annealed_1.0->0.001'

    env = gym.make('CartPole-v0')
    state_size = env.observation_space.shape[0]  # 4 for
                                                 # CartPole-v0
    num_actions = env.action_space.n  # 2 for CartPole-v0

    solved = False
    with tf.Session() as session:
        agent = PGAgent(session=session, state_size=state_size,
          num_actions=num_actions,
                        hidden_size=16,
                          explore_exploit_setting=
                            explore_exploit_setting)
        session.run(tf.global_variables_initializer())

        episode_rewards = []
        batch_losses = []

        global_memory = Memory()
        steps = 0
        for i in tqdm.tqdm(range(total_episodes)):
            state = env.reset()
            episode_reward = 0.0
            episode_history = EpisodeHistory()
            epsilon_percentage = float(min(i/float(
              epsilon_stop), 1.0))
            for j in range(max_episode_length):
                action = agent.predict_action(state,
                  epsilon_percentage)

                state_prime, reward, terminal, _ =
                  env.step(action)
                if (render_start > 0 and i >
                  render_start and should_render) \
                    or (solved and should_render):
                    env.render()
                episode_history.add_to_history(
                    state, action, reward, state_prime)
                state = state_prime
                episode_reward += reward
                steps += 1
                if terminal:
                    episode_history.discounted_returns =
                      discount_rewards(
                        episode_history.rewards)
                    global_memory.add_episode(
                      episode_history)
```

```
            if np.mod(i, train_frequency) == 0:
                feed_dict = {
                agent.reward_input: np.array(
                  global_memory.discounted_returns),
                agent.action_input: np.array(
                  global_memory.actions),
                agent.state: np.array(
                  global_memory.states)}
                _, batch_loss = session.run(
                    [agent.train_step, agent.loss],
                      feed_dict=feed_dict)
                batch_losses.append(batch_loss)
                global_memory.reset_memory()

            episode_rewards.append(episode_reward)
            break

    if i % 10:
        if np.mean(episode_rewards[:-100]) >
          100.0:
            solved = True
        else:
            solved = False
```

This code will train a CartPole agent to successfully and consistently balance the pole.

PGAgent Performance on Pole-Cart

Figure 3-6 is a chart of the average reward of our agent at each step of training. We try out 8 different sampling methods, and achieve best results with epsilon greedy annealing from 1.0 to 0.001.

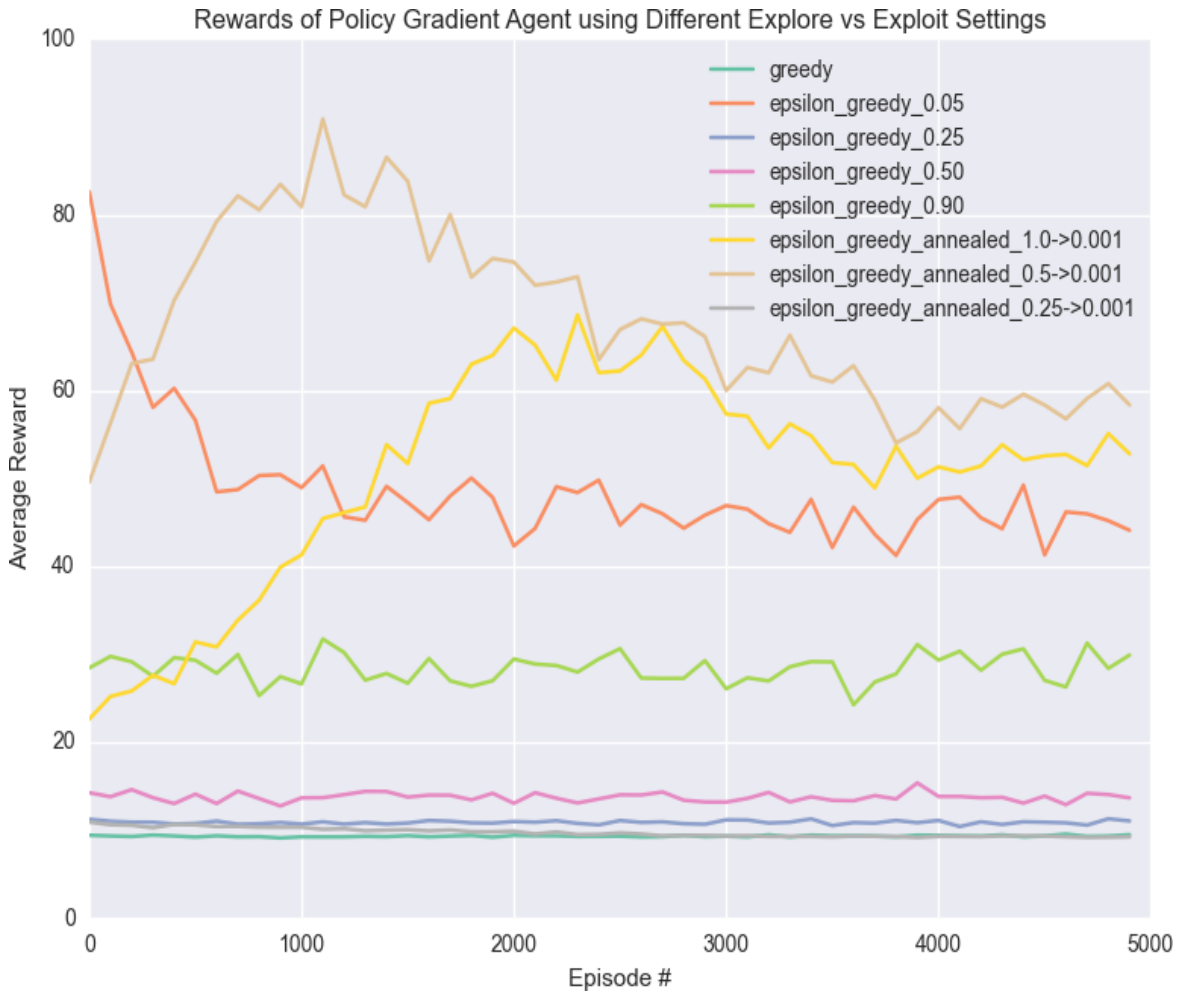


Figure 3-6. Explore-exploit configurations affect how fast and how well learning occurs

Notice how, across the board, standard epsilon greedy does very poorly. Lets talk about why this might be. With a high epsilon set to 0.9, we are taking a random action 90% of the time. Even if the model learns to execute the perfect actions, we'll still only be using these 10% of the time. On the other end, with a low epsilon of 0.05, we are taking what our model believes to be optimal actions the vast majority of the time. This performance is a bit better, but gets stuck in a local reward minimum because it lacks the ability to explore other strategies. So neither epsilon greedy of 0.05 nor 0.9 gives us great results. The former places too much emphasis on exploration, and the latter, too little. This is why epsilon annealing is such a powerful sampling strategy. It allows the model to explore early and exploit late, which is crucial to learning good policies.

Trust-region Policy Optimization

Trust-region policy optimization, or *TRPO* for short, is a framework that ensures policy improvement while preventing the policy from shifting too much during each training step. TRPO has been empirically shown to outperform many of its fellow policy gradient and policy iteration methods, allowing researchers to effectively learn complex, nonlinear policies (often parametrized by large neural networks) that weren't previously possible through gradient-based methods. In this section, we will motivate TRPO and describe its objective in more detail.

The idea of preventing the policy from shifting too much during each training step is not a new one - most regularized optimization procedures do this indirectly by penalizing the norm of the parameters for example, globally ensuring the norm of the parameters doesn't get too high. Of course, in cases where regularized optimization can also be formulated as constrained optimization (where there are explicit bounds on the norm of the parameter vector), such as l2-regularized linear regression, we have a direct equivalence to the idea of preventing the policy from shifting too much during each training step. The per-step change in the norm of the parameters is bounded by the range of the constraint, since all possible parameter values must fall in this range. For those interested, I would recommend looking further into the equivalence between Tikhonov and Ivanov regularization in linear regression.

Preventing the policy from shifting too much during each training step has the standard effect of regularized optimization - it promotes stability in training, which is ideal in preventing overfitting to new data. How do we define a shift in the policy? Policies are simply discrete probability distributions over the action space given a state, $\pi_\theta(a|s)$, for which we can use notions of dissimilarity introduced in the Fundamentals of Probability. The original TRPO paper introduces a bound on the average KL divergence (over all possible states) between the current policy and the new policy.

Now that we've introduced the constraint portion of TRPO's constrained optimization, we will motivate and define the objective function. First, we will recap and introduce some terminology:

$\eta(\pi) = \mathbb{E}_{s_0,a_0,s_1,a_1,\dots}\left[\sum_{t=0}^{\infty} \gamma^t r(s_t)\right]$

$Q_\pi(s_t, a_t) = \mathbb{E}_{s_{t+1},a_{t+1},\dots}\left[\sum_{l=0}^{\infty} \gamma^l r(s_{t+l})\right]$

$V_\pi(s_t) = \mathbb{E}_{a_t,s_{t+1},a_{t+1}\dots}\left[\sum_{l=0}^{\infty} \gamma^l r(s_{t+l})\right]$

$A_\pi(s,a) = Q_\pi(s,a) - V_\pi(s)$

$\rho_\pi(s) = \sum_{i=0}^{\infty} \gamma^i P(s_t = s)$

The first term is $\eta(\pi)$, which represents the *expected discounted reward*. We saw the finite-time horizon version of the term inside the expectation earlier when we discussed future discounted reward. Instead of looking at a single trajectory here, we take the expectation over all possible trajectories as defined by our policy π. Note that, as usual, we can estimate this expectation via an empirical average by sampling trajectories using π. The second term, which will be discussed in more detail during the section on Q-Learning, is the *Q-function* $Q_\pi(s_t, a_t)$, which looks very similar to the previous term but is instead defined as the expected discounted return from time t given we are in some state s_t and perform a defined action a_t in that state. We again calculate the expectation using our policy π. Note that the time t doesn't actually matter all too much since we only consider an infinite time horizon and the expected discounted return from t rather than from the beginning of the trajectory.

The third term is $V_\pi(s_t)$, or the *value function* at a particular state at time *t.* The value function can actually be more concisely written as $V_\pi(s_t) = \mathbb{E}_{a_t}[Q_\pi(s_t, a_t)]$, or the expectation of the Q-function w.r.t $\pi(a_t|s_t)$. In essence, the Q-function supposes that we take a defined action a_t in state s_t, while the value function leaves a_t as a variable. Thus, to get the value function, all we need to do is take the expectation of the Q-function w.r.t the distribution over a_t knowing the current state s_t. The result is the weighted average of the Q-function, where the weights are $\pi(a_t|s_t)$. In essence, this term captures the average future discounted return we'd expect to see starting in some state s_t.

The fourth term is $A_\pi(s, a)$, or the *advantage function.* Note that we have dropped the time t by now for the reasons mentioned earlier. The intuition for the advantage function is that it, under a fixed policy π, quantifies the benefit of letting trajectories play out after taking a particular action a in the current state s over simply letting trajectories play out from the current state s completely unconstrained. Even more concisely, it defines how much better, or worse, in the long run it is to initially take action a in state s compared to the average.

The final term, or the *unnormalized discounted visitation frequency,* re-introduces the time term *t.* This term is a function of the probability of being in state s at each time t from the start to infinity. This term will be important in our definition of the objective function. The original TRPO paper chose to optimize the model parameters by maximizing this objective function:

$L_{\theta_{old}}(\theta) = \sum_s \rho_{\theta_{old}}(s) \sum_a \pi_\theta(a|s) A_{\theta_{old}}(s, a)$

$\theta_{new} = \text{argmax}_\theta L_{\theta_{old}}(\theta)$

Although we won't fully show the derivation behind this objective as it is quite mathematically involved and beyond the scope of this text, we provide some intuition. Let's first examine this term: $\sum_a \pi_\theta(a|s) A_{\theta_{old}}(s,a)$, assuming a fixed state *s*. For the sake of argument, let's replace θ with θ_{old} as our proposed policy's parameters, which also represents our current policy's parameters:

$$\sum_a \pi_{\theta_{old}}(a|s) A_{\pi_{\theta_{old}}}(s,a) = \mathbb{E}_{a\sim\pi_{\theta_{old}}(a|s)}\left[A_{\pi_{\theta_{old}}}(s,a)\right]$$

$$= \mathbb{E}_{a\sim\pi_{\theta_{old}}(a|s)}\left[Q_{\pi_{\theta_{old}}}(s,a)\right] - \mathbb{E}_{a\sim\pi_{\theta_{old}}(a|s)}\left[V_{\pi_{\theta_{old}}}(s)\right]$$

$$= \mathbb{E}_{a\sim\pi_{\theta_{old}}(a|s)}\left[Q_{\pi_{\theta_{old}}}(s,a)\right] - V_{\pi_{\theta_{old}}}(s)$$

$$= V_{\pi_{\theta_{old}}}(s) - V_{\pi_{\theta_{old}}}(s)$$

$$= 0$$

What have we shown here? Earlier, we talked about how $A_{\pi_{\theta_{old}}}(s,a)$ defines how much better or worse it is, under the current policy, to take action *a* in state *s* compared to what we expect to see starting from state *s* unconstrained. Here, we showed that if we average each of these advantages weighted by the current policy's distribution, we are left with zero average advantage over the current policy - this makes a lot of intuitive sense, since the proposed policy and the current policy are the exact same. We don't expect to see any performance gain by replacing the current policy with itself!

Now, if we replace θ with a different proposed policy's parameters θ_{alt}, the above derivation leads us to: $\mathbb{E}_{a\sim\pi_{\theta_{alt}}(a|s)}\left[Q_{\pi_{\theta_{old}}}(s,a)\right] - V_{\pi_{\theta_{old}}}(s)$. This is as far as simplification will take us, since the actions in the first term are no longer distributed as the current policy and we can't make the simplification that led us to the penultimate step. If we evaluate this expression and we receive a positive result, we can interpret the result as representing a positive average advantage from following the proposed policy compared to following the current policy, directly translating to a performance gain for this specific state *s* by replacing the current policy with the proposed policy.

Note that we have only been considering a specific state *s.* But even if we see a performance gain for some state, it might be the case that that state only rarely shows up. This leads us to the inclusion of the term

$\sum_s \rho_{\theta_{old}}(s)$, which quantifies how often we see a given state. We can actually re-write this as an expectation even though this is an unnormalized distribution - all we'd need to do is factor out the normalizing constant, which is also a constant from the perspective of θ since the normalizing constant is solely a function of θ_{old}.

Keep in mind that $\sum_s \rho_{\theta_{old}}(s)$ is evaluated using the current policy rather than the proposed policy - this is because, as noted in the paper, the complex dependency this introduces on θ when optimizing the alternative objective (which uses $\sum_s \rho_{\theta}(s)$) w.r.t θ makes the optimization process difficult. Additionally, the paper proves that the first-order gradient matches that of the alternative objective anyways, allowing us to make this substitution without introducing a biased gradient estimate. We won't show this here however, as it is beyond the scope of the text.

Putting everything together, we have the following constrained optimization objective:

$$\theta^* = \text{argmax}_\theta \sum_s \rho_{\theta_{old}}(s) \sum_a \pi_\theta(a|s) A_{\theta_{old}}(s,a)$$

$$s.t. \text{ Avg. } \text{KL}(\theta_{old}, \theta) \leq \delta$$

Where the average KL divergence denotes the expected KL divergence between policies over all states. This is what we call the *trust region,* and it represents the parameter settings that lie close enough to the current parameter setting, mitigate training instability, and mitigate overfitting. How do we go about optimizing this objective? The inner summation looks like an expectation w.r.t $\pi_\theta(a, s)$, but all we have is our current setting of parameter values θ_{old}. In the standard setting, or *on-policy* setting, we are sampling from the same policy we are optimizing, so we can use classic policy gradient optimization for this. However, TRPO can be modified to work in the *off-policy* setting as well, where the policy we are sampling from is different from the policy we are optimizing. Generally, the reason for this distinction is that we may have a behavior policy, the policy we are sampling from that may be more exploratory in nature, while we learn the target policy, which is to be optimized. In the off-policy setting, since we are sampling actions from a different distribution *q(a|s)* (the behavior policy) from $\pi_\theta(a|s)$ (the target policy) we instead use the following constrained optimization objective:

$$\theta^* = \text{argmax}_\theta \sum_s \rho_{\theta_{old}}(s) \sum_a \frac{\pi_\theta(a|s)}{q(a|s)} A_{\theta_{old}}(s,a)$$

$$s.t. \text{ Avg. } \text{KL}(\theta_{old}, \theta) \leq \delta$$

The addition of q(a|s) accounts for the fact that we are sampling from a separate behavior policy. We can think about this more concretely in terms of expectations:

$$\sum_a \pi_\theta(a|s) A_{\theta_{old}}(s,a) = \sum_a \frac{q(a|s)}{q(a|s)} \pi_\theta(a|s) A_{\theta_{old}}(s,a)$$

$$= \sum_a q(a|s) \frac{\pi_\theta(a|s)}{q(a|s)} A_{\theta_{old}}(s,a)$$

$$= \mathbb{E}_{q(a|s)} \left[\frac{\pi_\theta(a|s)}{q(a|s)} A_{\theta_{old}}(s,a) \right]$$

Note that the left hand side of the first equality can be written as an expectation of the advantage w.r.t the target policy. In a few algebraic manipulations, we were able to convert our original objective into an equivalent objective, but with an expectation that is taken w.r.t the behavior policy! This is ideal because we are sampling from the behavior policy and can thus use standard minibatch gradient descent techniques to optimize this objective (note that adding in the constraint on the KL divergence makes this a bit more complicated than just standard gradient descent). And finally, we have already seen methods for sampling from the unnormalized probability distribution $\rho_{old}(s)$ for the outer expectation, amongst others that exist in academic literature.

Proximal Policy Optimization

One issue with TRPO is that its optimization is relatively complicated due to the inclusion of the average KL divergence term and involves second-order optimization to perform. *Proximal policy optimization*, or *PPO* for short, is an algorithm that tries to retain the benefits of TRPO without the complicated optimization. PPO proposes the following objective instead:

$$J(\theta) = \mathbb{E}\left[\min\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_{old}}(a|s)} A_{\theta_{old}}(s,a), \text{clip}\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_{old}}(a|s)}, 1-\epsilon, 1+\epsilon \right) A_{\theta_{old}}(s,a) \right) \right]$$

$$\theta^* = \text{argmax}_\theta J(\theta)$$

Note that we no longer have a complex constraint, but rather an extra term built into the optimization objective. The clip function represents an upper limit and a lower limit on the ratio between the target policy and the behavior policy, where any ratio above or below these limits is set equal to the corresponding limit. Note the inclusion of the minimum between the original and the clipped, which prevents us from making extreme updates and keeps us from overfitting.

As stated in the paper introducing PPO, it's important to notice that the objective for TRPO and PPO have the same gradient at $\theta = \theta_{old}$. This is the case in at least the on-policy setting, where we have a single policy from which we are sampling and optimizing (i.e. no distinction between the behavior and target policy). Let's take a closer look at why this is the case. To do this, we first need to re-formulate TRPO's constrained optimization objective as an equivalent regularized optimization objective (recall from early in the previous section), which we can do according to the theory. The objective looks like:

$$J^{TRPO}(\theta) = \mathbb{E}[\frac{\pi_\theta(a|s)}{\pi_{\theta_{old}}(a|s)} A(s,a) - \beta * \mathrm{KL}(\pi_{\theta_{old}}(a|s)||\pi_\theta(a|s))]$$

Note that we can separate the expression within the expectation into a difference of expectations due to the linearity of expectation. If we first consider the second expectation, or the KL term, we'll notice that this term is minimized at $\theta = \theta_{old}$ since the reference distribution is parametrized using θ_{old}. Thus, the gradient at this setting is zero, since we have already reached the global minimum! We are left with only the gradient of the first expectation, $\nabla_\theta \mathbb{E}\left[\frac{\pi_\theta(a|s)}{\pi_{\theta_{old}}(a|s)} A(s,a)\right]$.

Looking to the objective for PPO, we notice that at $\theta = \theta_{old}$, the ratio between the two policies is one, eliminating the need for the clip term. Thus, we are left with a minimum over two equivalent terms, which simplifies to the expectation over a single term. The gradient comes out to exactly what we just saw for the TRPO objective: $\nabla_\theta \mathbb{E}\left[\frac{\pi_\theta(a|s)}{\pi_{\theta_{old}}(a|s)} A(s,a)\right]$.

We have shown that PPO has the same gradient as TRPO in the select on-policy setting, and is additionally much easier to optimize in practice. PPO has also shown strong empirical results on a variety of tasks, and has become widely used in the field of deep RL.

Q-Learning and Deep Q-Networks

Q-learning is in the category of reinforcement learning called value-learning. Instead of directly learning a policy, we will be learning the value of states and actions. Q-learning involves learning a function, a *Q-function*, which represents the quality of a state, action pair. The Q-function, defined *Q(s, a)*, is a function that calculates the maximum discounted future return when action a is performed in state s.

The Q-value represents our expected long-term rewards, given we are at a state, and take an action, and then take every subsequent action

perfectly (to maximize expected future reward). This can be expressed formally as:

$$Q^*(s_t, a_t) = max_\pi E\left[\sum_{i=t}^{T} \gamma^i r^i\right]$$

A question you may be asking is, how can we know Q-values? It is difficult, even for humans, to know how good an action is, because you need to know how you are going to act in the future. Our expected future returns depend on what our long-term strategy is going to be. This seems to be a bit of a chicken-and-egg problem. In order to value a state, action pair you need to know all the perfect subsequent actions. And in order to know the best actions, you need to have accurate values for a state and action.

The Bellman Equation

We solve this dilemma by defining our Q-values as a function of future Q-values. This relation is called the *Bellman equation*, and it states that the maximum future reward for taking action is the current reward plus the next step's max future reward from taking the next action a':

$$Q^*(s_t, a_t) = E\left[r_t + \gamma \max_{a'} Q^*\left(s_{t+1}, a^{'}\right)\right]$$

This recursive definition allows us to relate between Q-values.

And since we can now relate between Q-values past and future, this equation conveniently defines an update rule. Namely, we can update past Q-values to be based on future Q-values. This is powerful because there exists a Q-value we know is correct: the Q-value of the very last action before the episode is over. For this last state, we know exactly that the next action led to the next reward, so we can perfectly set the Q-values for that state. We can use the update rule, then, to propagate that Q-value to the previous time step:

$$\widehat{Q_j} \rightarrow \widehat{Q_{j+1}} \rightarrow \widehat{Q_{j+2}} \rightarrow \ldots \rightarrow Q^*$$

This updating of the Q-value is known as *value iteration*.

Our first Q-value starts out completely wrong, but this is perfectly acceptable. With each iteration, we can update our Q-value via the correct one from the future. After one iteration, the last Q-value is accurate, since it is just the reward from the last state and action before episode termination. Then we perform our Q-value update, which sets the second-to-last Q-value. In our next iteration, we can guarantee that the last two Q-values are correct, and so on and so forth. Through value

iteration, we will be guaranteed convergence on the ultimate optimal Q-value.

Issues with Value Iteration

Value iteration produces a mapping between state and action pairs with corresponding Q-values, and we are constructing a table of these mappings, or a *Q-table*. Lets briefly talk about the size of this Q-table. Value iteration is an exhaustive process that requires a full traversal of the entire space of state, action pairs. In a game like Breakout, with 100 bricks that can be either present or not, with 50 positions for the paddle to be in, and 250 positions for the ball to be in, and 3 actions, we have already constructed a space that is far, far larger than the sum of all computational capacity of humanity. Furthermore, in stochastic environments, the space of our Q-table would be even larger, and possibly infinite. With such a large space, it will be intractable for us to find all of the Q-values for every state, action pair. Clearly this approach is not going to work. How else are we going to do Q-learning?

Approximating the Q-Function

The size of our Q-table makes the naive approach intractable for any nontoy problem. However, what if we relax our requirement for an optimal Q-function? If instead, we learn approximations of the Q-function, we can use a model to estimate our Q-function. Instead of having to experience every state, action pair to update our Q-table, we can learn a function that approximates this table, and even generalizes outside of its own experience. This means we won't have to perform an exhaustive search through all possible Q-values to learn a Q-function.

Deep Q-Network (DQN)

This was the main motivation behind DeepMind's work on Deep Q-Network (DQN). DQN uses a deep neural network that takes an image (the state) in to estimate the Q-value for all possible actions.

Training DQN

We would like to train our network to approximate the Q-function. We express this Q-function approximation as a function of our model's parameters, like this:

$$\widehat{Q_\theta}(s, a \mid \theta) \sim Q^*(s, a)$$

Remember, Q-learning is a value-learning algorithm. We are not learning a policy directly, but rather we are learning the values of each state, action pair, regardless if they are good or not. We have expressed our model's Q-function approximation as Qtheta, and we would like this to be close to the future expected reward. Using the Bellman Equation from earlier, we can express this future expected reward as:

$$R_t^* = \left(r_t + \gamma \max_{a'} \widehat{Q}\left(s_{t+1}, a' \middle| \theta\right)\right)$$

Our objective is to minimize the difference between our Q's approximation, and the next Q value:

$$\min_\theta \sum_{e \in E} \sum_{t=0}^{T} \widehat{Q}(s_t, a_t|\theta) - R_t^*$$

Expanding this expression gives us our full objective:

$$\min_\theta \sum_{e \in E} \sum_{t=0}^{T} \widehat{Q}(s_t, a_t|\theta) - \left(r_t + \gamma \max_{a'} \widehat{Q}\left(s_{t+1}, a' \middle| \theta\right)\right)$$

This objective is fully differentiable as a function of our model parameters, and we can find gradients to use in stochastic gradient descent to minimize this loss.

Learning Stability

One issue you may have noticed is that we are defining our loss function based on the difference of our model's predicted Q-value of this step and the predicted Q-value of the next step. In this way our loss is doubly dependent on our model parameters. With each parameter update, the Q-values are constantly shifting, and we are using shifting Q-values to do further updates. This high correlation of updates can lead to feedback loops and instability in our learning where our parameters may oscillate and make the loss diverge.

We can employ a couple of simple engineering hacks to remedy this correlation problem; namely, target Q-network and experience replay.

Target Q-Network

Instead of updating a single network frequently with respect to itself, we can reduce this codependence by introducing a second network, called the *target network*. Our loss function features to instances of the Q-function, $\widehat{Q}(s_t, a_t|\theta)$ and $\widehat{Q}\left(s_{t+1}, a' \middle| \theta\right)$. We are going to have the first Q be represented as our prediction network, and our second Q will be produced by the target Q-network. The target Q-network is a copy of our

prediction network that lags in its parameter updates. We only update the target Q-network to equal the prediction network every few batches. This provides much needed stability to our Q-values, and we can now properly learn a good Q-function.

Experience Replay

There is yet another source of irksome instability to our learning: the high correlations of recent experiences. If we train our DQN with batches drawn from recent experience, these action, state pairs are all going to be related to one another. This is harmful because we want our batch gradients to be representative of the entire gradient, and if our data is not representative of the data distribution, our batch gradient will not be an accurate estimate of the true gradient.

So we have to break up this correlation of data in our batches. We can do this using something called *experience replay*. In experience replay, we store all of the agent's experiences as a table, and to construct a batch, we randomly sample from these experience. We store these experiences in a table as (s_i, a_i, r_i, s_{i+1}) tuples. From these four values, we can compute our loss function, and thus our gradient to optimize our network.

This experience replay table is more of a queue than a table. The experiences an agent sees early in training may not be representative of the experiences a trained agent finds itself in later, so it is useful to remove very old experiences from our table.

From Q-Function to Policy

Q-learning is a value learning paradigm, not a policy learning algorithm. This means we are not directly learning a policy for acting in our environment. But can't we construct a policy from what our Q-function tells us? If we have learned a good Q-function approximation, this means we know the value of every action for every state. We could then trivially construct an optimal policy in the following way: look at our Q-function for all actions in our current state, choose the action with the max Q-value, enter a new state, and repeat. If our Q-function is optimal, our policy derived from it will be optimal. With this in mind, we can express the optimal policy as follows:

$$\pi(s;\theta) = \text{argmax}_{a'} \widehat{Q^*}\left(s, a';\theta\right)$$

We can also use the sampling techniques we discussed earlier to make a stochastic policy that sometime deviates from the Q-function recommendations to vary the amount of exploration our agent does.

DQN and the Markov Assumption

DQN is still a Markov decision process that relies on the *Markov assumption*, which assumes that the next state s_i+1 depends only on the current state s_i and action a_i, and not on any previous states or actions. This assumption doesn't hold true for many environments where the game's state cannot be summed up in a single frame. For example, in Pong, the ball's velocity (an important factor in successful gameplay) is not captured in any single game frame. The Markov assumption makes modeling decision processes much simpler and reliable, but often at a loss of modeling power.

DQN's Solution to the Markov Assumption

DQN solves this problem by utilizing *state history*. Instead of processing one game frame as the game's state, DQN considers the past four game frames as the game's current state. This allows DQN to utilize time-dependent information. This is a bit of an engineering hack, and we will discuss better ways of dealing with sequences of states at the end of this chapter.

Playing Breakout wth DQN

Lets pull all of what we learned together and actually go about implementing DQN to play Breakout. First we start out by defining our DQNAgent:

```
# DQNAgent

class DQNAgent(object):

    def __init__(self, session, num_actions,
                 learning_rate=1e-3, history_length=4,
                 screen_height=84, screen_width=84,
                 gamma=0.98):
        self.session = session
        self.num_actions = num_actions
        self.learning_rate = learning_rate
        self.history_length = history_length
        self.screen_height = screen_height
        self.screen_width = screen_width
        self.gamma = gamma

        self.build_prediction_network()
```

```
        self.build_target_network()
        self.build_training()

    def build_prediction_network(self):
        with tf.variable_scope('pred_network'):
            self.s_t = tf.placeholder('float32', shape=[
                                      None,
                                      self.history_length,
                                      self.screen_height,
                                      self.screen_width],
                                      name='state')
            self.conv_0 = slim.conv2d(self.s_t, 32, 8, 4,
              scope='conv_0')
            self.conv_1 = slim.conv2d(self.conv_0, 64, 4, 2,
              scope='conv_1')
            self.conv_2 = slim.conv2d(self.conv_1, 64, 3, 1,
              scope='conv_2')

            shape = self.conv_2.get_shape().as_list()

            self.flattened = tf.reshape(
                self.conv_2, [-1, shape[1]*shape[2]*shape[3]])
            self.fc_0 = slim.fully_connected(self.flattened,
               512, scope='fc_0')
            self.q_t = slim.fully_connected(
              self.fc_0, self.num_actions, activation_fn=None,
               scope='q_values')

            self.q_action = tf.argmax(self.q_t, dimension=1)

    def build_target_network(self):
        with tf.variable_scope('target_network'):
            self.target_s_t = tf.placeholder('float32',
              shape=[None, self.history_length,
                self.screen_height, self.screen_width],
                  name='state')
            self.target_conv_0 = slim.conv2d(
                self.target_s_t, 32, 8, 4, scope='conv_0')
            self.target_conv_1 = slim.conv2d(
                self.target_conv_0, 64, 4, 2, scope='conv_1')
            self.target_conv_2 = slim.conv2d(
                self.target_conv_1, 64, 3, 1, scope='conv_2')

            shape = self.conv_2.get_shape().as_list()

            self.target_flattened = tf.reshape(
                self.target_conv_2, [-1,
                  shape[1]*shape[2]*shape[3]])
            self.target_fc_0 = slim.fully_connected(
                self.target_flattened, 512, scope='fc_0')
            self.target_q = slim.fully_connected(
                self.target_fc_0, self.num_actions,
                  activation_fn=None, scope='q_values')

    def update_target_q_weights(self):
        pred_vars = tf.get_collection(
            tf.GraphKeys.GLOBAL_VARIABLES, scope=
              'pred_network')
        target_vars = tf.get_collection(
```

```
        tf.GraphKeys.GLOBAL_VARIABLES, scope=
          'target_network')
    for target_var, pred_var in zip(target_vars, pred_vars):
        weight_input = tf.placeholder('float32',
          name='weight')
        target_var.assign(weight_input).eval(
            {weight_input: pred_var.eval()})

def build_training(self):
    self.target_q_t = tf.placeholder('float32', [None],
      name='target_q_t')
    self.action = tf.placeholder('int64', [None],
      name='action')

    action_one_hot = tf.one_hot(
        self.action, self.num_actions, 1.0, 0.0,
          name='action_one_hot')
    q_of_action = tf.reduce_sum(
        self.q_t * action_one_hot, reduction_indices=1,
          name='q_of_action')

    self.delta = tf.square((self.target_q_t - q_of_action))
    self.loss = tf.reduce_mean(self.delta, name='loss')

    self.optimizer = tf.train.AdamOptimizer(
        learning_rate=self.learning_rate)
    self.train_step = self.optimizer.minimize(self.loss)

def sample_action_from_distribution(self,
  action_distribution, epsilon_percentage):
    # Choose an action based on the action probability
    # distribution
    action = epsilon_greedy_action_annealed(
        action_distribution, epsilon_percentage)
    return action

def predict_action(self, state, epsilon_percentage):
    action_distribution = self.session.run(
        self.q_t, feed_dict={self.s_t: [state]})[0]
    action = self.sample_action_from_distribution(
        action_distribution, epsilon_percentage)
    return action

def process_state_into_stacked_frames(self, frame,
  past_frames, past_state=None):
    full_state = np.zeros(
        (self.history_length, self.screen_width,
          self.screen_height))

    if past_state is not None:
        for i in range(len(past_state)-1):
            full_state[i, :, :] = past_state[i+1,
              :, :]
        full_state[-1, :, :] = imresize(to_grayscale(frame),
                                        (self.screen_width,
                                        self.screen_height))
                                          /255.0
    else:
        all_frames = past_frames + [frame]
```

```
            for i, frame_f in enumerate(all_frames):
                full_state[i, :, :] = imresize(
                    to_grayscale(frame_f), (self.screen_width,
                      self.screen_height))/255.0
            full_state = full_state.astype('float32')
        return full_state
```

There is a lot going on in this class, so let's break it down.

Building Our Architecture

We build our two Q-networks: the prediction network and the target Q-network. Notice how they have the same architecture definition, since they are the same network, with the target Q just having delayed parameter updates. Since we are learning to play Breakout from pure pixel input, our game state is an array of pixels. We pass this image through three convolution layers, and then two fully connected layers to produce our Q-values for each of our potential actions.

Stacking Frames

You may notice that our state input is actually of size `[None, self.history_length, self.screen_height, self.screen_width]`. Remember, in order to model and capture time-dependent state variables like speed, DQN uses not just one image, but a group of consecutive images, also known as a *history*. Each of these consecutive images is treated as a separate channel. We construct these stacked frames with the helper function `process_state_into_stacked_frames(self, frame, past_frames, past_state=None)` .

Setting Up Training Operations

Our loss function is derived from our objective expression from earlier in this chapter:

$$\min_\theta \sum_{e \in E} \sum_{t=0}^{T} \widehat{Q}(s_t, a_t|\theta) - \left(r_t + \gamma \max_{a'} \widehat{Q}\left(s_{t+1}, a'\middle|\theta\right)\right)$$

We want our prediction network to equal our target network, plus the return at the current time step. We can express this in pure TensorFlow code as the difference between the output of our prediction network and the output of our target network. We use this gradient to update and train our prediction network, using `AdamOptimizer`.

Updating Our Target Q-Network

To ensure a stable learning environment, we only update our target Q-network once every four batches. Our update rule for the target Q-network is pretty simple: we just set its weights equal to the prediction network. We do this in the function `update_target_q_network(self)`. We can use `tf.get_collection()` to grab the variables of the prediction and target network scopes. We can loop through these variables and run the `tf.assign()` operation to set the target Q-network's weights equal to those of the prediction network.

Implementing Experience Replay

We've discussed how experience replay can help de-correlate our gradient batch updates to improve our the quality of our Q-learning and subsequent derived policy. Let's walk though a simple implementation of experience replay. We expose a method `add_episode(self, episode)` which takes an entire episode (an `EpisodeHistory` object) and adds it to the ExperienceReplayTable. It then checks if the table is full and removes the oldest experiences from the table.

When it comes time to sample from this table, we can call `sample_batch(self, batch_size)` to randomly construct a batch from our table of experiences:

```
class ExperienceReplayTable(object):

    def __init__(self, table_size=5000):
        self.states = []
        self.actions = []
        self.rewards = []
        self.state_primes = []
        self.discounted_returns = []

        self.table_size = table_size

    def add_episode(self, episode):
        self.states += episode.states
        self.actions += episode.actions
        self.rewards += episode.rewards
        self.discounted_returns += episode.discounted_returns
        self.state_primes += episode.state_primes

        self.purge_old_experiences()

    def purge_old_experiences(self):
        if len(self.states) > self.table_size:
            self.states = self.states[-self.table_size:]
            self.actions = self.actions[-self.table_size:]
            self.rewards = self.rewards[-self.table_size:]
            self.discounted_returns = self.discounted_returns[
                -self.table_size:]
            self.state_primes = self.state_primes[
```

```
            -self.table_size:]

    def sample_batch(self, batch_size):
        s_t, action, reward, s_t_plus_1, terminal = [], [],
          [], [], []
        rands = np.arange(len(self.states))
        np.random.shuffle(rands)
        rands = rands[:batch_size]
        for r_i in rands:
            s_t.append(self.states[r_i])
            action.append(self.actions[r_i])
            reward.append(self.rewards[r_i])
            s_t_plus_1.append(self.state_primes[r_i])
            terminal.append(self.discounted_returns[r_i])
        return np.array(s_t), np.array(action),
            np.array(reward), np.array(s_t_plus_1),
            np.array(terminal)
```

DQN Main Loop

Let's put this all together in our main function, which will create an OpenAI Gym environment for Breakout, make an instance of our DQNAgent, and have our agent interact with and train to play Breakout successfully:

```
def main(argv):
    # Configure Settings
    run_index = 0
    learn_start = 100
    scale = 10
    total_episodes = 500*scale
    epsilon_stop = 250*scale
    train_frequency = 4
    target_frequency = 16
    batch_size = 32
    max_episode_length = 1000
    render_start = total_episodes - 10
    should_render = True

    env = gym.make('Breakout-v0')
    num_actions = env.action_space.n

    solved = False
    with tf.Session() as session:
        agent = DQNAgent(session=session,
          num_actions=num_actions)
        session.run(tf.global_variables_initializer())

        episode_rewards = []
        batch_losses = []

        replay_table = ExperienceReplayTable()
        global_step_counter = 0
        for i in tqdm.tqdm(range(total_episodes)):
            frame = env.reset()
            past_frames = [frame] * (agent.history_length-1)
```

```
state = agent.process_state_into_stacked_frames(
    frame, past_frames, past_state=None)
episode_reward = 0.0
episode_history = EpisodeHistory()
epsilon_percentage = float(min(i/float(
  epsilon_stop), 1.0))
for j in range(max_episode_length):
    action = agent.predict_action(state,
      epsilon_percentage)
    if global_step_counter < learn_start:
        action = random_action(agent.num_actions)

    # print(action)
    frame_prime, reward, terminal, _ = env.step(
      action)
    state_prime =
      agent.process_state_into_stacked_frames(
        frame_prime, past_frames,
          past_state=state)

    past_frames.append(frame_prime)
    past_frames = past_frames[-4:]

    if (render_start > 0 and (i >
      render_start)
            and should_render) or (solved and
              should_render):
        env.render()
    episode_history.add_to_history(
        state, action, reward, state_prime)
    state = state_prime
    episode_reward += reward
    global_step_counter += 1
    if j == (max_episode_length - 1):
        terminal = True

    if terminal:
        episode_history.discounted_returns =
          discount_rewards(
            episode_history.rewards)
        replay_table.add_episode(episode_history)

        if global_step_counter > learn_start:
            if global_step_counter %
              train_frequency == 0:
                s_t, action, reward, s_t_plus_1,
                  terminal = \
                    replay_table.sample_batch(
                      batch_size)
                q_t_plus_1 = agent.target_q.eval(
                    {agent.target_s_t:
                      s_t_plus_1})

                terminal = np.array(terminal) + 0.
                max_q_t_plus_1 = np.max(q_t_plus_1,
                  axis=1)
                target_q_t = (1. - terminal) * \
                    agent.gamma * max_q_t_plus_1 +
                      reward
```

```
                    _, q_t, loss = agent.session.run(
                        [agent.train_step, agent.q_t,
                          agent.loss], {
                        agent.target_q_t: target_q_t,
                        agent.action: action,
                        agent.s_t: s_t
                    })

                if global_step_counter %
                  target_frequency == 0:
                    agent.update_target_q_weights()

            episode_rewards.append(episode_reward)
            break

    if i % 50 == 0:
        ave_reward = np.mean(episode_rewards[-100:])
        print(ave_reward)
        if ave_reward > 50.0:
            solved = False
        else:
            solved = False
```

DQNAgent Results on Breakout

We train our DQNAgent for 1,000 episodes to see the learning curve. To obtain superhuman results on Atari, typical training time runs up to several days. However, we can see a general upward trend in reward pretty quickly, as shown in Figure 3-7.

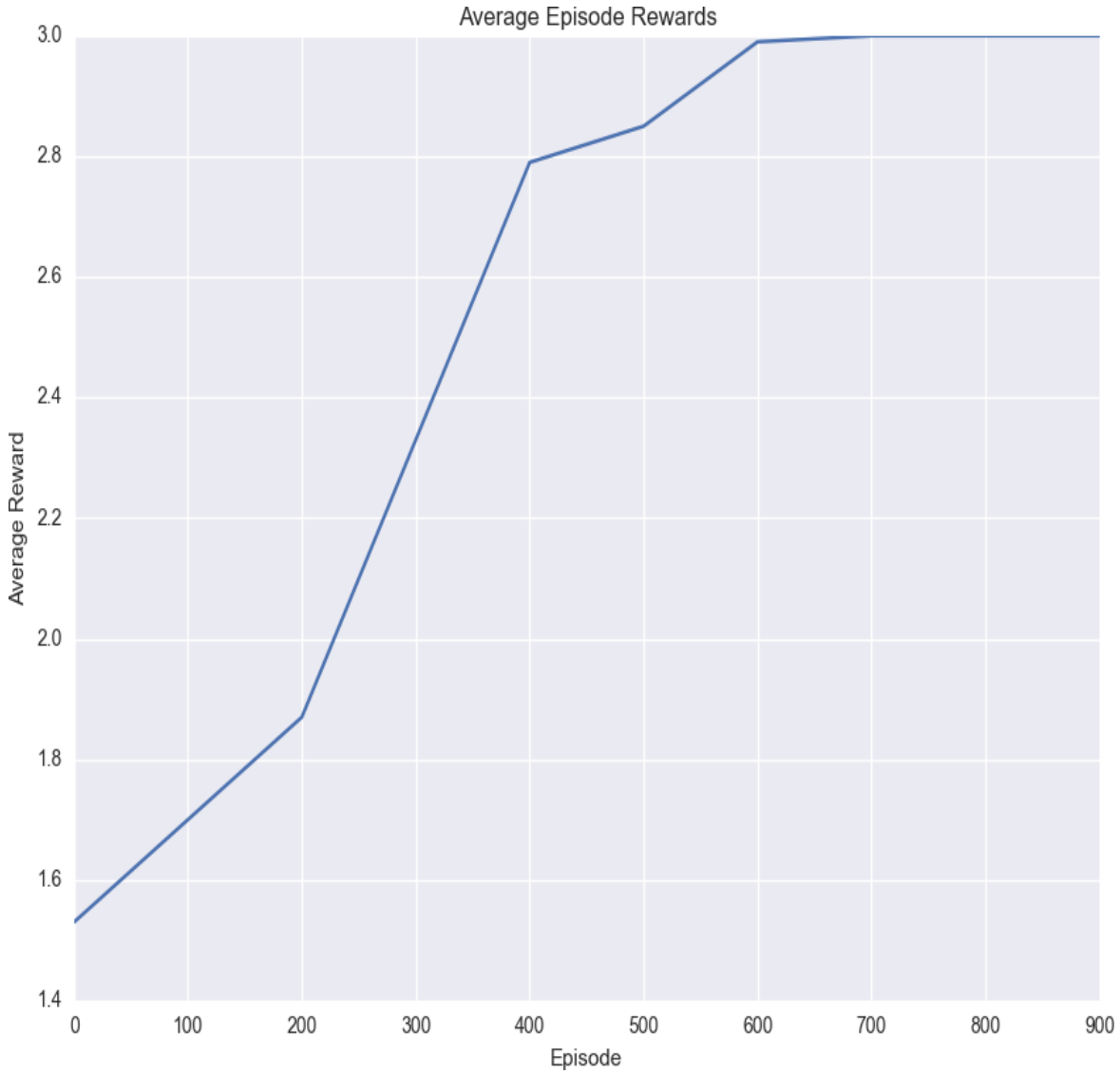


Figure 3-7. Our DQN agent gets increasingly better at Breakout during training as it learns a good value function and also acts less stochastically due to epsilon-greedy annealing

Improving and Moving Beyond DQN

DQN did a pretty good job back in 2013 in solving Atari tasks, but had some serious shortcomings. DQN's many weaknesses include that it takes very long to train, doesn't work well on certain types of games, and requires retraining for every new game. Much of the deep reinforcement learning research of the past few years has been in addressing these various weaknesses.

Deep Recurrent Q-Networks (DRQN)

Remember the Markov assumption? The one that states that the next state relies only on the previous state and the action taken by the agent? DQN's solution to the Markov assumption problem, stacking four consecutive frames as separate channels, sidesteps this issue and is a bit of an ad hoc engineering hack. Why four frames and not 10? This imposed frames history hyperparameter limits the model's generality. How do we deal with arbitrary sequences of related data? That's right: we can use what we learned back in [Link to Come] on recurrent neural networks to model sequences with *deep recurrent Q-networks* (DRQN).

DRQN uses a recurrent layer to transfer a latent knowledge of state from one time step to the next. In this way, the model itself can learn how many frames are informative to include in its state and can even learn to throw away noninformative ones or remember things from long ago.

DRQN has even been extended to include neural attention mechanism, as shown in Sorokin et al.'s 2015 paper "Deep Attention Recurrent Q-Network" (DAQRN).[5] Since DRQN is dealing with sequences of data, it can attend to certain parts of the sequence. This ability to attend to certain parts of the image both improves performance and provides model interpretability by producing a rationale for the action taken.

DRQN has shown to be better than DQN at playing first-person shooter (FPS) games like DOOM,[6] as well as improving performance on certain Atari games with long time-dependencies, like Seaquest.[7]

Asynchronous Advantage Actor-Critic Agent (A3C)

Asynchronous advantage actor-critic (A3C) is a new approach to deep reinforcement learning introduced in the 2016 DeepMind paper "Asynchronous Methods for Deep Reinforcement Learning."[8] Let's discuss what it is and why it improves upon DQN.

A3C is *asynchronous*, which means we can parallelize our agent across many threads, which means orders of magnitude faster training by speeding up our environment simulation. A3C runs many environments at once to gather experiences. Beyond the speed increase, this approach presents another significant advantage in that it further decorrelates the experiences in our batches, because the batch is being filled with the experiences of numerous agents in different scenarios simultaneously.

A3C uses an *actor-critic*[9] method. Actor-critic methods involve learning both a value function $V(s_t)$ (the critic) and also a policy $\pi(s_t)$, (the actor). Early in this chapter, we delineated two different approaches to reinforcement learning: value learning and policy learning. A3C combines

the strengths of each, using the critic's value function to improve the actor's policy.

A3C uses an *advantage* function instead of a pure discounted future return. When doing policy learning, we want to penalize the agent when it chooses an action that leads to a bad reward. A3C aims to achieve this same goal, but uses advantage instead of reward as its criterion. Advantage represents the difference between the model's prediction of the quality of the action taken versus the actual quality of the action taken. We can express advantage as:

$$A_t = Q^*(s_t, a_t) - V(s_t).$$

A3C has a value function, V(t), but it does not express a Q-function. Instead, A3C estimates the advantage by using the discounted future reward as an approximation for the Q-function:

$$A_t = R_t - V(s_t)$$

These three techniques proved key to A3C's takeover of most deep reinforcement learning benchmarks. A3C agents can learn to play Atari Breakout in less than 12 hours, whereas DQN agents may take 3 to 4 days.

UNsupervised REinforcement and Auxiliary Learning (UNREAL)

UNREAL is an improvement on A3C introduced in "Reinforcement learning with unsupervised auxiliary tasks" [10] by Jaderberg et al., who, you guessed it, are from DeepMind.

UNREAL addresses the problem of reward sparsity. Reinforcement learning is so difficult because our agent just receives rewards, and it is hard to determine exactly why rewards increase or decrease, which makes learning difficult. Additionally, in reinforcement learning, we must learn a good representation of the world as well as a good policy to achieve reward. Doing all of this with a weak learning signal like sparse rewards is quite a tall order.

UNREAL asks the question, what can we learn from the world without rewards, and aims to learn a useful world representation in an unsupervised matter. Specifically, UNREAL adds some additional unsupervised auxiliary tasks to its overall objective.

The first task involves the UNREAL agent learning about how its actions affect the environment. The agent is tasked with controlling pixel values on the screen by taking actions. To produce a set of pixel values in the

next frame, the agent must take a specific action in this frame. In this way, the agent learns how its actions affect the world around it, enabling it to learn a representation of the world that takes into account its own actions.

The second task involves the UNREAL agent learning *reward prediction.* Given a sequence of states, the agent is tasked with predicting the value of the next reward received. The intuition behind this is that if an agent can predict the next reward, it probably has a pretty good model of the future state of the environment, which will be useful when constructing a policy.

As a result of these unsupervised auxiliary tasks, UNREAL is able to learn around 10 times faster than A3C on the Labyrynth game environment. UNREAL highlights the importance of learning good world representations and how unsupervised learning can aid in weak learning signal or low-resource learning problems like reinforcement learning.

Summary

In this chapter, we covered the fundamentals of reinforcement learning, including MDP's, maximum discounted future rewards, and explore versus exploit. We also covered various approaches to deep reinforcement learning, including policy gradients and Deep Q-Networks, and touched on some recent improvements on DQN and new developments in deep reinforcement learning.

Reinforcement learning is essential to building agents that can not only perceive and interpret the world, but also take action and interact with it. Deep reinforcement learning has made major advancements toward this goal, successfully producing agents capable of mastering Atari games, safely driving automobiles, trading stocks profitably, controlling robots, and more.

---

1 *http://nicklocascio.com/*

2 Mnih, Volodymyr, et al. "Human-level control through deep reinforcement learning." *Nature* 518.7540 (2015): 529-533.

3 Brockman, Greg, et al. "OpenAI Gym." *arXiv preprint arXiv*:1606.01540 (2016). *https://gym.openai.com/*

4 Sutton, Richard S., et al. "Policy Gradient Methods for Reinforcement Learning with Function Approximation." NIPS. Vol. 99. 1999.

5 Sorokin, Ivan, et al. "Deep Attention Recurrent Q-Network." *arXiv preprint arXiv*:1512.01693 (2015).

6 *https://en.wikipedia.org/wiki/Doom_(1993_video_game)*

7 *https://en.wikipedia.org/wiki/Seaquest_(video_game)*

8 Mnih, Volodymyr, et al. “Asynchronous methods for deep reinforcement learning.” *International Conference on Machine Learning*. 2016.

9 Konda, Vijay R., and John N. Tsitsiklis. “Actor-Critic Algorithms.” *NIPS*. Vol. 13. 1999.

10 Jaderberg, Max, et al. “Reinforcement Learning with Unsupervised Auxiliary Tasks.” *arXiv preprint arXiv*:1611.05397 (2016).

About the Author

Nikhil Buduma is the cofounder and chief scientist of Remedy, a San Francisco-based company that is building a new system for data-driven primary healthcare. At the age of 16, he managed a drug discovery laboratory at San Jose State University and developed novel low-cost screening methodologies for resource-constrained communities. By the age of 19, he was a two-time gold medalist at the International Biology Olympiad. He later attended MIT, where he focused on developing large-scale data systems to impact healthcare delivery, mental health, and medical research. At MIT, he cofounded Lean On Me, a national nonprofit organization that provides an anonymous text hotline to enable effective peer support on college campus and leverages data to effect positive mental health and wellness outcomes. Today, Nikhil spends his free time investing in hard technology and data companies through his venture fund, Q Venture Partners, and managing a data analytics team for the Milwaukee Brewers baseball team.

Nithin Buduma is one of the first machine learning engineers at XY.ai, a start-up based out of Harvard and Stanford working to help healthcare companies leverage their massive datasets.

k. Summary