Andrea Stocco

Explanatory Computational Models in Cognitive Neuroscience

JANUARY 24, 2023

Contents

List of Figures

55 When presented with a pattern of neural activity, a Hopfield network will spontaneous switch to a lower-energy one until a stable pattern is found. At this point, the network has reached a stable configuration. This configuration coincides with one of the learned patterns. 102

List of Tables

Introduction

There are many ways to understand the brain, from directly manipulating the activity of neurons *in vitro* to observing the behavior of patients who have suffered a stroke. One of the most power tools that we have nowadays to understand the brain is through mathematical models and computer simulations. This approach started sometime in the '40s, but has become prominent in the past decade due to a combination of successes, including the recent rise of deep-learning neural networks in AI and the successes of reinforcement learning in robotics and automation.

Over the course of many decades, a core set of computational frameworks have become prominent in the field of computational neurosciences. These frameworks are remarkable for many reasons. Among them, they all have enjoyed enormous popularity, at different times, in computer science, Artificial Intelligence, and engineering; they all have found ways into cognitive science; and they all have given rise to their own set of specialized, consistent, and well-defined specialities.

Most importantly, all of these frameworks have become essential tools for understanding one of more aspects of the functional neuroanatomy of the brain.

Intuitively, the brain is a complex system, that has evolved to solve multiple problems at the same time. Each of these frameworks has evolved from an original simple question ("How should one learn from reward?", "How should one form memories?", "What is the best way to recognize objects?"). So, none of these frameworks really answers the question, "How does the brain work?". However, all of them provide *partial* answers; the brain *does* learn from rewards; it *does* memorize facts; and it *does| recognize objects.* Thus, in a way, these frameworks provide important insights into how certain parts of the brain work, and why they work precisely they way do. Some of these answers are partially overlapping; fading memories can be used to learn better from rewards, for example.

What is a Model?

All of these frameworks attack these problems using a *modeling* angle. There are many definitions of what a model is, but, in the simplest possible terms, a

model is just an abstract, simplified representation of a complex system. The model usually simplifies certain characteristics of the system and explicitly captures its internal workings into a set of formal equations or computational processing steps. Once these workings are captured, researchers can do a variety of things.

Explanation and Prediction

There are at least two reasons why the computational approach is important. And, although intertwined, they are also separate.

The first is *explanation*. When we understand what a circuit does, we can gather insight into why our data looks the why it does. You might have a puzzling experimental results, and the model might explain why this result occurs in the first place.

The second is *prediction*. A model that is a good approximation of a system would be able to predict what would happen in the system. An epidemiological model, for example, could be use to predict how many people would be infected by a specific disease in the upcoming days.

There is a tension between the two. In many ways, the difference between explanation and prediction is not so clear-cut as it seems. Because a model, by its very nature, produces an output every time it is run, it is always making a prediction, the difference between explanation and prediction often becomes whether a prediction is about past or future data, or between existing data or yet unseen data (even *past* unseen data).

Models as Functions

They can use the model to explain previously strange patterns of results, and, by examining the model, better understand how and why these results arise. They can also use the model to predict what would happen in circumstances that have not been experimentally tested yet. And, finally, they can compare the model to data, and examine whether the model does a good job, and to what extent.

So, a model is a theory of a particular system's *function*. In fact, mathematically, a model can be thought as a mathematical function that connects a set of conditions X to a set of observed outcomes, Y, i.e., $X \rightarrow f(X) \rightarrow Y$. The model captures how and why these initial variables X affect the outcome.

Why Should We Understand The Function?

But why would one care about the function in the first place? After all, experimental scientists do a lot of work characterizing what happens in a system when a set of variables is changed. And having a model does not get away with the need to run experiments: In most cases, nobody would trust a model's predictions blindly, and most researchers would like to see them

verified anyway. So, why would experimental sciences need to use models?

I like to summarize the difference between these two approaches with the metaphor of the difference between the rules of the movement of a chess piece and that piece's function in a game, as in Figure 1. Consider, for example, the knight. The knight's movement is the most complicated of all the chess pieces: it proceeds in every direction by two squares and then it turns and moves perpendicularly one square, making an L-shaped trajectory (Figure 1, left). But, if we were to observe how the knight is played during a game, we might not be able to make this inference at all. At the beginning of a game, for example, the knight might be used very early and place strategically to defend the center the of the board (Figure 1, center). At the end of a game, instead, the knight might be used to restrict the movement of the king in preparation for a checkmate (Figure 1, right). When we, as neuroscientists, perform experiments on how a certain brain region is being used, we are in fact just observing how the brain might be using the same chess piece under different conditions. If all we can say about the knight is that it is being used to "defend the center at the beginning" and to "attack the king in endgame", we are left with little explanatory power. Even worse, if all we can say is that "the knight is used at the beginning and the end of a game", we are left with not much knowledge than what we started with.

If, on the other hand, we can describe exactly how the knight moves, then we can make sense of all of its functions, *explain* why the player has used it that way, and *predict* how and when the knight will be used in the future.

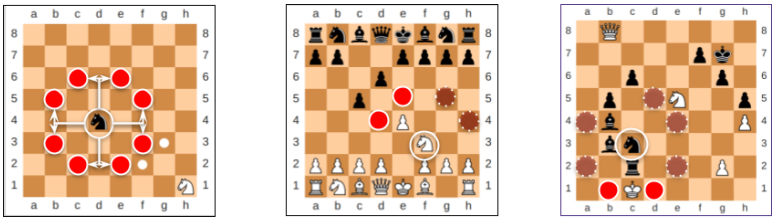


Figure 1: A comparison of the knight's "computations" (*left*) and two of its possible "functions": defending the center, at the beginning (*center*) and supporting a check, in endgame (*right*). Although the functions might be different, the computations remain the same, and in fact it is the piece's computations (the rules of movement) that explain its use in different phases of the game.

Two Traditions of Modeling

In a landmark paper, Breiman[1] identified two traditions of statistics, one he called *data modeling* and one he called *algorithmic modeling*.

In the most general term possible, a model is a function $f(X)$ that connects a set of data X to a set of outcomes Y, i.e., $X \rightarrow f(X) \rightarrow Y$. This extremely general definition holds for anything we might want to call "model": It works for detailed models of brain networks as well as for statistical models. Whether you are creating a large neural network that simulates how the visual brain sees the world or you are just fitting a linear regression model, you are still doing the same: creating a *function* that makes sense of the *data*.

The difference is how these two traditions think about this mysterious

function. The data modeling approach works a in sort of top down manner: It starts with some assumptions about the nature of the data, and proceeds to derive predictions about the outcomes from these assumptions. When a statistician states that two variables need to be "independent and normally distributed", they are doing precisely that–making assumptions about the processes that generate the data: in this case, that the data might be generated by sampling without replacement from random pool of values with a certain mean and variance. From this assumption, a statistician can derive very precise predictions about the observed outcomes Y; for example, they might derive the probability that all outcomes come from the very same pool. Because this approach starts with assumptions about the function f that generates the Y, it is called data modeling.

However, one could also be agnostic about these hypothesis, and simply use mathematical tools that approximate the underlying function from the constraints posed by the data itself. Breiman called this approach "algorithmic"; nowadays, this approach is commonly known as *machine learning*.

These two traditions reverberate throughout any modeling approach, in any field I have ever seen. In old fashioned, symbolic AI, adherents to the two traditions were called sometimes called "neats" and "scruffies" [2] and in Computational Psychiatry, for example, these two approaches are called "explanatory models" and (much more transparently) "machine learning".

[2] I personally love these terms. Full disclousure: I am a scruffy.

Confusingly, the same concepts, abstractions and techniques are sometimes be used in both approach. Take, for instance, the case of the technique called *linear regression*: it consists of using a simple model in which the effects of a series of independent variables add up to determine the value of a dependent variable. This modeling technique is commonly used in statistics be to test the existence of a relationship between two variables (an example of data modeling) and but it can also be used to approximate an unknown function, as it is used in a special technique called LASSO (an example of machine learning). Even more dramatically, neural networks can be taken as a structural model of the brain (an example of data modeling) but can also be trained, as we will see, to approximate *any* function (which is way they are ubiquitous in contemporary machine learning).

These ambiguities nonetheless, in this book I will focus on computational models of the *explanatory* tradition. All of the models described here embody a theory about a specific brain function (about learning, memory, perception) and use different abstractions to make sense of what we know.

An Example of Explanatory Model: Fitts' Law

To understand the different facets of an explanatory model, let's consider a simple one. It is a mathematical model of response times for motor movements, known as Fitts' Law [3]. Fitts' Law is an equation that predicts the time required to move a hand (or a cursor, or a pen) to a target area that has width

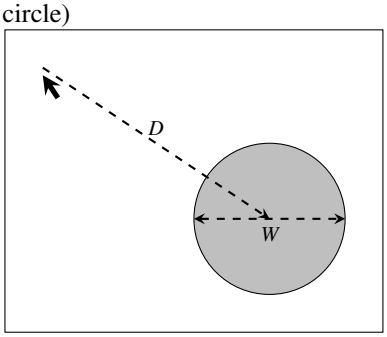
[3] Paul M Fitts. The information capacity of the human motor system in controlling the amplitude of movement. *Journal of Experimental Psychology*, 47(6):381, 1954

W and is located at a distance D from the current position of the hand. Figure 2 illustrates a typical example: calculating the time needed to move a cursor from one position to a different area. According to Fitts' Law, the time T is related to width W and distance D by Equation 1:

$$T = a + b\log_2\left(\frac{2D}{W}\right) \tag{1}$$

where a is an intercept, and can be consider the minimum amount of time required to initiate any movement, while b is a scaling factor, and can considered as a general parameter that captures an individual's speed of movement.

Figure 2: A possible application of Fitts' law: Determining the time it takes to use a mouse to move a cursor (black arrow) from its original position to a new area (the grey circle)



Inside a Model: Fit, Features, and Free Parameters

If we peek inside a model (*any* model) we can find some common elements. First, any model must have an *output*. In the case of Fitt's law, the output is the movement time T. This output might or might not reflect the data; the degree to which the model's output matches the data is called the model's *fit*.

Second, each model contains certain quantities that capture specific aspects of the outside world and environment. In Eq. 1, for example, the quantities W and D (width and distance of the target) represent everything we need to know about the world in which we need to make a movement. These variables are called *features*; in choosing the appropriate features, the designer of a model implicitly defines the levels of abstraction and the degree of simplification they want to impose on the world. Note that, once the level of abstraction is chosen, the features are, in principle, measurable properties of the outside world. (A partial exception to this rule is represented by contemporary deep-learning models, which are trained on raw data and are capable at extracting features on their own),

Finally, Eq. 1 contains two more variables, a and b. Unlike D and W, they do not represent a measurable property of the world; in fact, there is no way they can be measured independently of the the equation itself. These variables, which mediate the effect of the features (the outside world) on the output, are called *free parameters*. One of the defining characteristics of explanatory models is that, because they embody a theory, it is somewhat clear what their parameters represent. By looking at equation 1, it is clear that no response time time can ever be smaller than a; thus, a can be thought of as the smallest time it takes to initiate a movement on the given device. The paramenter b, on the other hand, mediates the additional time it takes to move a cursor to a given location. Thus, we can think of b as representing the *effort* necessary to control the movement itself. In general, a movement will be slower as D grows and faster as W grows, but some individuals will be faster overall, while others will need more time to move the cursor; these differences will be reflected in different values of b, and we can say that, when b is smaller, the amount of effort that the movement takes is also smaller.

Fitting a Model

But, now that we have identified features and free parameters, how do we know whether our model is any good?

To do so, we need to find the specific parameters the model that better fit the data. In the case of Fitts' model, that comes down to finding that data values of a and b that reduce the difference between the model's prediction Y' and the actual data Y. This difference, or any other difference we want to minimize, is called the *loss* function.

Suppose, for instance, you ran an experiment varying the distance D from a cursor to a target are of width W, and you obtain the data from Table 1.

W	D	Time (s)
100	300	3.36
150	50	1.08
220	100	1.16
110	200	3.07
40	250	4.12

Table 1: Results from a hypothetical experiment with the setup of Figure 2 , with varying values of the distances D and the target width W

We can define a loss function that quantifies how close the predictions of the Fitts model come to the times recorded in the third column in the table. For each combination of values of the parameters a and b, we can plug in the different values of W and D and compare the model's predictions Y' against the five observed values Y. A convenient way to do so is to calculate the sum of squares, much as it is done in statistics:

$$L = \sum_{y \in Y} (y - y')^2$$

Fitting a model is, therefore, the process of finding the values of parameters a and b that minimize the output of this equation.

In some cases, you might be lucky enough that there are some specific formulae that let you calculate the ideal parameters in a few simple steps. This is the case of linear regression. It also happens to be the case of Fitt's law. If you consider equation 1, you'll notice that the it is essentially a linear equation of the form $y = a + bx$, once you consider $\log_2(2D/W)$ as your independent variable x. To find the two values of a and b that create the better fit, you simply combine all of the pairs of W and D into a single variable x, concatenate all of these variables into a vector X, and apply the linear regression formula: $(X^T X)^{-1} X^T Y$.

The result, in this case, is $a = 1.4438$ and $b = 0.7560$. With these parameters value, the loss function measures only 0.093. Notice that, to calculate these values, we had to change the model's *features*: Fitts' model sees the world as made of distances and widths, but the linear regression model sees only a single value x. This is another case in which you need to transform the data to fit it into the model's worldview. But is this a good value? You can judge for yourself: Figure 3 shows the predictions of Fitts' Law, represented as line, against the experimental results.

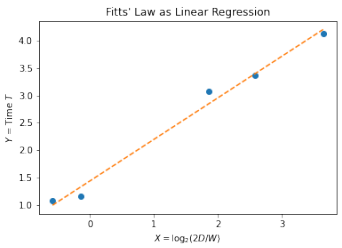


Figure 3: Predictions of Fitts' Law (red dashed line) against the experimental results of Table 1 (blue dots). The values of features W and D have been combined into a single value x, and the parameters a and b were fit with linear regression

But what if we cannot use a direct formula to calculate the best values of a model's parameters? In the most general case, it is possible to use brute force and examine multiple values of a and b until we identify the combination that minimizes our loss function. For example, one could sample all values of a and b from 0.5 to 1.5 in increments of 0.01, and compute the loss function for each combination. This approach, called *grid search*, gives

you an approximate idea of fit of Fitts' model within a slide of its parameter space, as shown in Figure 5. In the figure, colors represent the magnitude of the loss function, and darker areas represents smaller loss value and, thus, better fits. The cross sign "+" marks the position, in the parameter space, that corresponds to the solution found by linear regression.

However, this brute force, grid-search approach is rarely used in practice, as sampling all of the parameters is often unfeasible–especially as models become more complex and take longer to run. For example the plot in Figure 5 was generated by examining 10,000 combination of a and b values; such as a sample might not be feasibile. Furthermore, grid search requires setting a predefined sampling that discretaizes the possible values of a and b. For example, the grid search examined cases in which $a = 1.44$ and $a = 1.45$, but never examined the case in which $a = 1.4438$; such a value would, in fact, be invisible to the method.

For all of these reason, it is common to use special techniques called *optimization algorithms* instead of grid searches. These algorithms capitalize on the fact that, in most models, similar parameter values would produce similar results in terms of the model's loss function. In Fitt's law, for example, changing the value of a from 1.44 to 1.45 does not produce appreciable changes in the loss function, no matter what the value of b is. Furthermore, the direction of the changes in the loss function is usually consistent: If changing a from 1.44 to 1.45 increases the loss function, then a further change of a to 1.46 would likely result in an even larger loss value. In other words, that the surface of the loss function over the two parameters' values is smooth. And smooth functions can be explored fairly easily by examining finding out the direction in which the parameters can be changed to reduce the loss function. This is exactly what optimization algorithms do:they start with an initial guess for the model parameters, and modify them iteratively in the direction that reduces the loss function, until a minimum value is found[4]. Optimization algorithms explore only a small portion of the parameter space, but they quickly converge over the correct solution. Figure 5 depicts the points (in white) explored by one such method, the Nelder-Mead algorithm, to find the values of a and b that minimize the loss function of Fitts' Law, starting at $a = 1$, $b = 1$ and terminating at the same values that were identified by linear regression.

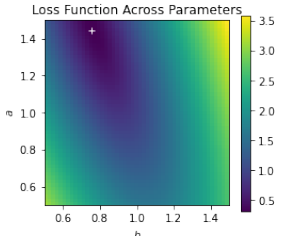


Figure 4: Loss function for the Fitts model across different values of parameters a and b, when compared against the data in Table 1.

Models as Theories and Models as Measures

Every explanatory models is an abstract, simplified representations of some system—a theory of how the system works. When fitting a model, however, researchers might be interested in two very different things: the theory itself, and the properties of the system that the theory allows to measure.

[4] For this reason, these algorithms are also called *minimization* algorithms.

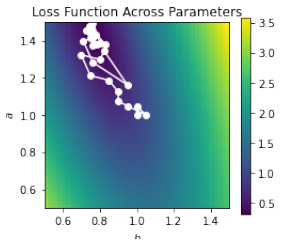


Figure 5: Points of the parameter space explored by the Nelder-Mead algorithm

Models as Theories

In the first case, the researchers might be interested in the model in itself. Every model, in a sense, is a *theory*, and the researchers might have developed the model as a new theory that explains how and why a particular set of phenomena happen. Fitting the model to empirical data is done as a way to provide a quantitative measure of how good of a theory the model is.

This is exactly how Fitts' Law was originally conceived: it was proposed as a principled way to make sense of motor movements. Like much of the '50s mathematical psychology work (such as the work of Hick [5] and Hyman [6] on response times in the presence of multiple options), it was deeply influenced by information theory[7]. In his paper, Fitts derived these equations from a purely theoretical point of view. In this sense, this model was a representation of the computational efforts that brains engaged with when performing a movement. A very *successful* representation, as evidenced by the sheer amount of citations that the original paper keeps amassing over the year. In his paper, experimental data were used corroborate the original intuition.

[5] William E Hick. On the rate of gain of information. *Quarterly Journal of Experimental Psychology*, 4(1):11–26, 1952

[6] Ray Hyman. Stimulus information as a determinant of reaction time. *Journal of experimental psychology*, 45(3):188, 1953

[7] The use of logarithms in base 2 is a dead giveaway!

Models as Measures

There is a second view of models, which is related and easily confused with the first one but remain substantially different. In this view, researchers typically *assume* that a given model is correct, and is interested in the values of the model's parameters.

Since it was originally proposed, Fitts' Law has become a *de facto* assumption in much research in Human Factors and Human Computer Interaction—that is why it is called Fitts' *Law*. In this case, the model is not used as a representation, but as a measurement method. It is assumed that the model is either true or a sufficiently accurate representation, and used to make predictions. Do you want to calculate how easy it is to move a mouse to a "submit" button on a login page? You can use Fitts' Law. Have you created a new interface that uses special laser pointers pointed to an LCD? It will still follow Fitts' Law, but you might need to collect some data and establish new values of a and b for your interface.

Because the parameters of an explanatory model are conceptually clear, it is also possible to use the model to make sense of complex data patterns. As noted above, the a and b parameters can be interpreted as the time it takes to initiate a movement and a the effort it takes to perform it. suppose you want to investigate whether it is easier to use a computer mouse or a touch device (like a smartphone's screen) to drag a window. You can collect data and extract the values of a and b for both devices. You might find, for example, that a is smaller for the touch device, as it is faster to point your finger than it is to grab a mouse and click, but b is smaller for the mouse, as it can cover longer distances much more easily than the finger. This would be

an insightful analysis, and would help you conclude, for example, that touch devices are better for small screens and mice are better for larger ones.

Parameters can be also used to investigate different between individuals. In general, some people will be consistently slower or faster at movement movements, and these differences would be reflected in different values of a and b.

An entire class of explanatory models described herein, that of accumulator models, was designed precisely as way to measure unobserved but conceptually clear processes from the distribution of response times—and they are incredibly successful at it.

Levels and Traditions of Models

As I mentioned at the beginning of this chapter, a useful model is simpler than the object it is trying to simulate. When deciding what to model, one of the fundamental choices is the level of abstraction at which you intend to capture the phenomenon of interest. Consider, for example, the case of a scientist interested in developing a model of memory. One choice would be to start from first principles: what is memory for? If memory is necessary to make relevant events of the past available, it likely mirrors the statistics of the environment, so that memories of very common events are more likely to be remembered than memories of less likely ones.

A different approach would be to abstract some feature of memory and try to capture it with a simple mechanism. For example, memories tend to fade with time; thus, we can imagine memories being discrete entities whose availability decays over time. We could borrow the metaphor of radioactive decay, and approximate forgetting with an exponential decay.

Yet another approach would be to consider where in the brain memory is implemented. We know from patient studies that memories are stored (at least initially) in a circuit known as the hippocampus. The hippocampus, and in particular an area known as CA3, has a particular structure: it is a single layer of interconnected neurons. We can start by modeling this structure, the interaction between the different neurons, and see how memories can be represented in the network of neurons.

All three of these approaches have been attempted, and they are covered in Chapters 4 and 6. David Marr, one of the pioneers of computational approaches in the study of the brain and cognition, proposed a classification of these approaches that has proven influential [8]. According to him, each phenomenon could be modeled at three levels:

- The *functional* level[9]. At this level, the experimenter is investigating the general structure of the problem, and typically asks what would be the most general and optimal solution. A memory researcher who uses a Bayesian approach to capture when memories are more likely to be

[8] David Marr. *Vision: A computational investigation into the human representation and processing of visual information*. W. H. Freeman & Company, 2010

[9] In Marr's book, this level is actually called "computational". This is often confusing, since all of the other models are also computational in the common sense of the word. So I prefer to use "functional" here

retrieved is working at this level.

- The *algorithmic* level. At this level, a researchers outlines the basic elements of the model, including how to represent the problem (the features) and how the model is working step-by-step. The memory researcher who adopts the metaphor of radioactive decay and tries to predict the effect of time on forgetting works at this level.
- The *implementation* level. At this level, the modeler seriously consider the nature of the physical processes that occur. The memory researcher who decided to study memory by modeling the interactions of neurons in the hippocampus would be working at this level.

Much has been written about these levels of abstractions, and many authors have proposed their own classifications or expanded them. There is no right or wrong levels; each and every level provides different insights into the nature of thought. Similarly, the levels are not som clear-cut. For example, is Fitts' Law a model that exists at the functional or at the implementation level? And, finally, each of these approaches stills remains an abstraction of the original phenomenon—even if you dig deep into the implementation level.

Symbolic vs. Connectionist Traditions

Historically, when the idea of understanding brain function through modeling was still in its infancy, the field of cognitive science, cognitive psychology, and artificial intelligence were very close and almost indistinguishable. But even then, modelers aligned themselves in two different traditions, which have often been name "symbolic" and "connectionist". Symbolic models tend to explictly represent abstract concepts and their relationships, much in the same way as variables are represented in a computer program. They make for often elegant theories, but they tend to overlook the nitty-gritty details of how networks of neurons carry out the computations. For example, in Section I, we will present a model of memory in which different features of a fact are represented as a list, so that "The canary is a yellow bird" becomes something like "[(Object : Canary), (Type: Bird), (Property: Yellow)]". The use of such representations has earned this tradition the name "symbolic".

In the brain, of course, these lists do not exists, and properties such like being a "bird" and being "yellow" are represented in a distributed network of neurons. The specific ways in which the brain encodes and modifies these representations can be ignored only up to a certain point. A group of researchers have argued, since the very beginning, that it is better to start right there with a better understanding of how networks of neurons represent concepts and carry out computations. This school of thought has given rise to modern neural networks (including those used in contemporary deep-learning AIs) and is has become known as "connectionist".

These two traditions have often being in sharp disagreement with each other. Over the course of decades, they have taken turns in dominating the cognitive neurosciences (and also the fields of artificial intelligence and machine learning). As usual, I maintain that there is no unique, correct answer as to which one is the best—it largely depends on what one sets out to achieve and (why not?) on individual preferences. These two traditions are reflected in the structure of this book, whose first part contains symbolic models while its second part contains connectionist models.

What This Textbook is Not

A few words of caution. This is largely a textbook I have written because I needed to create a consistent set of materials for my own classes at the University of Washington. Although I could refer my students to individual papers or tutorials, I was constantly bothered by the lack of an easy way to integrate between different aspects of my classes. So, eventually, during a my 2020-21 sabbatical, I finally got around to start this textbook.

Compared to those textbooks, the material covered here is much more focused on the systems-level neuroscience than on single neuron properties. Similarly, the focus is much more on large-scale theories (RL, memory associators) than on biophysical models.

Part I

Algorithmic Models

Reinforcement Learning

The first framework we are going to see and examine is Reinforcement Learning (RL). In essence, RL is a minimalistic theory of how an agent should adapt to an environment. Originally, the term was used to describe a class of theories put forward by psychologist to describe how animals and human would change behavior when given rewards, like food pellets. A "reward", in that literature, was technically called a "reinforcement" because it would *reinforce* (i.e. make more likely to occur) a specific behavior. This set of theories, together with their somewhat confusing lexicon, remained dominant in the animal learning literature for decades after they had basically faded from human psychology research.

In the meantime, these theories and their namesake were borrowed by early A.I. researchers who, over time, developed RL in one of the most sophisticated mathematical framework for machine learning [10]. Current RL theory is responsible for some of the most incredible advancements in A.I., including successes in the ability to achieve human-level and super human-level performance in video games [11] and classic board games, such as Go [12]–feats that were unthinkable just a few years back.

Importantly for us, the mathematical theory of RL came full circle when, in the '90s, it was discovered to provide a remarkable good fit and an elegant explanation for the dynamics of the brain circuitry of reward, learning, and motivation [13]. After all, it was found out, reinforcement learning (the mathematical theory) did provide a surprisingly good explanation for reinforcement learning (the psychological phenomenon), even if the two had run on parallel tracks for decades!

[10] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018

[11] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013

[12] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016

[13] Wolfram Schultz, Peter Dayan, and P Read Montague. A neural substrate of prediction and reward. *Science*, 275(5306):1593–1599, 1997

Overview of RL Theory

RL is a branch of machine learning, the field of computer science that studies how an artificial computational "entity" (which, depending on the sub-field, can be called a classifier, a model, or a robot) changes its behavior and learns from new data. Machine learning algorithms are typically divided into two large classes: supervised, in which the programmer provides some form of ground truth that the machine needs to learn; and unsupervised, in which the programmer provides nothing and the algorithm digs through data extracting

patterns. For instance, a classifier is a supervised machine learning system– the programmer provides an initial set of exampled (or "labels") that the algorithm uses to learn the proper classes. A clustering algorithm, by contrast, is unsupervised: once it is given data in the proper format, it will simply do its job and divide it up in the requested number of clusters.

RL does not quite fit nicely in either class. For this reason, it is called a *semi*-supervised machine learning framework. It is semi-supervised because the programmer does, in fact, provide something to the algorithm but not nearly enough information to be considered a form of ground truth. The information provided by the programmer is typically in form of rewards, and leaves the RL agent in charge of making sense of them.

Agents and Environments

In RL, the problem is defined in terms of an *agent* and an *environment*. For simplicity, I will consider the simple case in which both the environment and the agent are *discrete*. An environment, in this sense, is a collection of possible *states* of the world $\mathscr{S} = \{s_1, s_2 \dots s_S\}$, each of which is associated with a corresponding *reward* in the set $\mathscr{R} = \{r_1, r_2 \dots r_S\}$. Each state s is a unique identifier, and it is up to the modeler to decide what a "state" is. The reward r is a scalar value, and can be positive or negative; [14]. In practice, the reward value is $r = 0$ for the greatest majority of states.

[14] RL researchers call "negative rewards" the cases in which $r < 0$. This is often confusing to other researchers, because most people assume that "rewards" must be positive; and is especially confusing to psychologists, for whom the expression "negative reward" has a specific but different meaning

The agent is defined as having a set of *actions* available $a_1, a_2 \dots a_N$ that are available and a *policy* π that defines how the agent acts, and might change over the course of learning. Technically, a policy is defined as a function that maps each state s of the environment to a probability distribution over the actions, that is, $\pi : a \rightarrow P(a)$. The agent has only one goal: to maximize the total number of rewards over time, from the present moment onwards. Assuming that time is discrete and that the present moment is indicated as t, this goal is equivalent to maximizing the quantity R_t so defined:

$$R_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \dots + \gamma^n r_\infty = \sum_{t=0}^{\infty} \gamma^t r_t \tag{2}$$

In Eq. 2, $0 < \gamma < 1$ is a *temporal discounting* parameter that ensures that R converges to a finite amount. It captures the intuitive idea that, given two identical rewards, the one that arrives sooner is better (after all, we have no guarantees about the distant future). The temporal discounting is key to avoid paradoxical situations, like Bernoulli's paradox, that otherwise plague classic decision theories.

Finally, we need to define how the agent and the environment interact with each other. In RL, the agent and the environment interact in a loop in which, at every time point t, the environment presents the agent with a state s_t and a reward r_t and the agent responds by performing an action a_t. Performing a_t results in the environment undergoing a state transition, after which it will

present the agent with the new state s_{t+1} and a new reward r_{t+1}. This loop is represented in Figure 6.

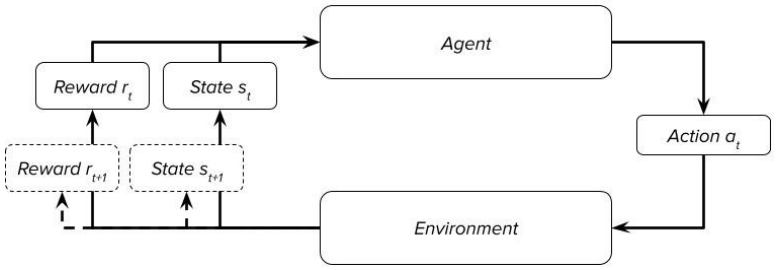


Figure 6: The RL loop

Learning the Values of States

The juice of RL is the theory behind how the agent learns. In theory, the agent could simply sample random states and actions for a long time, keeping track of all the outcomes, and estimate the average result of each action; that is the Monte Carlo or *statistical* approach. Alternatively, the agent could decide on a specific set of rules to follow (its policy) and iteratively modify it and refine it as it goes; that is the *dynamic programming* approach.

Although both approaches would work and would equally count as RL, the most common solution in RL is to use *temporal difference* (TD) methods, which were originally proposed by [15]. TD-based methods have many advantages, including the fact that the agent learns incrementally and on-line (that is, while performing the task), converge to optimal solutions, and consume very little in terms of time and memory. All of these characteristics make them also desirable for biological agents as well.

[15] Richard S Sutton. Learning to predict by the methods of temporal differences. *Machine learning*, 3(1):9–44, 1988

TD-learning methods exists that learn either the value of states s or the value of actions a for each state. We will consider the simple case of an agent that is learning about the value of the *states* first. In this case, we are assuming that the agent is either not interested in estimating the values of actions, or they might not have any control about their environment.

Learning, in this context, means that the agent need to estimate the *value* of a particular state, that is, how many rewards can be expected from now on, given that we are in this particular state. In other words, the value of a particular state $V(s_t)$ represents the agent's expectations about R_t.

Now, if the agent's expectations were, in fact, absolutely correct, then at any point in time, the agent's value for a state would perfectly match the expected future rewards, that is, $V(s_t) = R_t$, $V(s_{t+1}) = R_{t+1}$, and so on.

That means that:

$$\begin{aligned} V(s_t) &= R_t \\ &= r_t + \gamma R_{t+1} \end{aligned} \tag{3}$$

But, since in a perfect world $V(s_{t+1}) = R_{t+1}$, we can conclude that:

$$V(s_t) = r_t + \gamma V(s_{t+1}) \tag{4}$$

This simple considerations gives us a way to actually learn V. If, initially, all of our values for V are simple guesses, as time passes our estimates are going to become more precise because we gather experience with the environment and with the true rewards r that are delivered at any state. Because the more recent estimates are more precise than older ones, we can simply revise the older based on more recent. Suppose that, at time t, our previous estimate $V(s_t)$ was off by a certain error, which we indicate as δ_t. How do we estimate δ_t? Using Equation 4:

$$\begin{aligned} V(s_t) + \delta_t &= r_t + \gamma V(s_{t+1}) \\ \delta_t &= r_t + \gamma V(s_{t+1}) - V(s_t) \end{aligned} \tag{5}$$

Now, Equation 5 implies that the agent can estimate the error δ_t by simply comparing its own subsequent evaluations. Specifically, at time $t+1$, the agent can update its previous estimates of $V(s_t)$ by a fraction of the error:

$$V_{new}(s_t) \leftarrow V_{old}(s_t) + \alpha \delta_t \tag{6}$$

In this equation, the parameter α, which determines how much of the RPE is used to correct the estimates, is known as the *learning rate*.

Important Implications of δ_t

The centerpiece of RL agents is the term δ_t, which is also known as the *reward prediction error* or RPE. It is the RPE that drives learning and, under reasonable policies like Eq. 8, also drives behavioral change. The embeds several interesting properties.

- RL agents are driven by *relative*, not absolute rewards. This is also the reason why RL modelers are a bit cavalier in using positive and negative values in reward terms; a negative reward value is not necessarily a "punishments"; all that matters is how big that reward is compared to others;
- What drives change and learning is the difference between the *expected* and the *real* outcomes of actions. In other words, learning and changes in behavior are driven by *surprise*:
- Change propagates back in time, across states and actions. Because the terms in Eq. 6 are relative in time (t and $t+1$), the effects of δ_t move backwards in time over multiple passes.

- Learning and changes in behavior happen even in absence of rewards. This is perhaps the most important feature in RL theory: The δ_t terms does not reflect differences in rewards, but differences in reward *expectations*: Even when the reward term $r = 0$ in Eq. 5, it is still possible to obtain positive or negative values of δ_t that are propagated back in time. This makes it possible to apply RL to environments where true rewards (i.e., $r \neq 0$) are rare and occur only after complex sequences of actions.

The Reward Prediction Error and Dopamine

RL would have remained a nifty mathematical theory, if it weren't for the discovery that it provides ridiculously precise explanations for the firing of dopamine neurons.

Dopamine is one of the two neurotransmitters that regulate pleasure and happiness in the brain[16]. Of the two, dopamine is the one with the most dramatic effects. Dopamine release produces intense feelings of pleasure and satisfaction. The effects of dopamine are so intense that, given the opportunity of pressing a lever to activate an electrode placed in dopamine-releasing neurons, rats would happily prefer to starve than to give up the lever for food. Drugs of addiction, like cocaine and nicotine, work precisely by activating, directly or indirectly, dopamine receptors in the brain [17].

[16] The other one is serotonine. A common joke, in neuroscience, is that there only two things you enjoy in life: serotonine and dopamine.

[17] A David Redish. Addiction as a computational process gone awry. *Science*, 306(5703):1944–1947, 2004

Dopamine can be thought of as the universal coin in which all of the possible forms of primary rewards that can be gathered are translated to. This also lets the brain compare different types of primary rewards (let's say, food vs. money).

Despite the mounting evidence for the importance of dopamine, the exact function of dopamine remained elusive. Primary rewards, for example, normally elicit strong dopamine responses. However, in laboratory experiments, during which multiple trials are administered, the responses of dopamine neurons quickly disappears. More surprisingly, dopamine neurons often begin responding at apparently random moments of the experiment, but remain silent when the reward is presented.

These results remained confusing until Wolfram Schultz, Peter Dayan, and Read Montague published a landmark paper in 1997 [18]. In that paper, they summarized the main findings from a prototypical experiment, and explained them in the context to RL (Fig. 7).

[18] Wolfram Schultz, Peter Dayan, and P Read Montague. A neural substrate of prediction and reward. *Science*, 275(5306):1593–1599, 1997

In psychology and behavioral neuroscience, the prototypical experimental paradigm to study reward is called "classic conditioning" and requires the administration of a reward (typically, food) following a specific cue or stimulus. This is the exact same design used by Pavlov in his experiments with dogs. In Schultz's lab, the subjects were primates, the cue was a visual stimulus on a computer screen, and the reward was a drop of juice. Four main effects had been consistently observed:

- Dopamine neurons initially fire in response to the primary reward (the

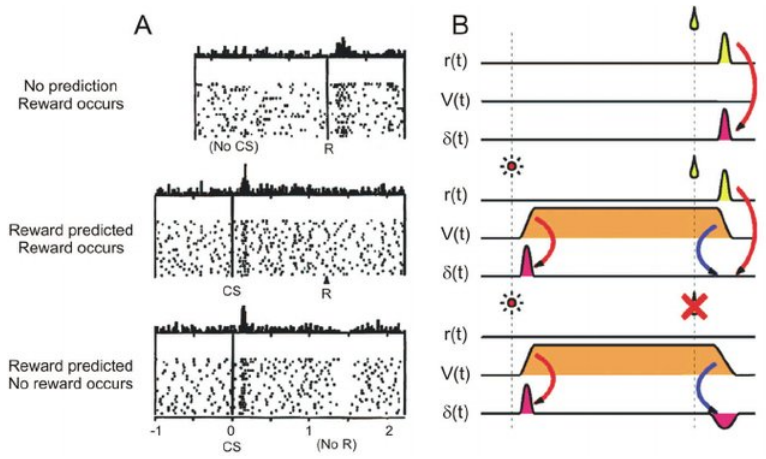


Figure 7: Data and model from On the left, trace of dopamine neuron firings ("rasterplots") recorded from a primate performing a simple cue-reward association task. On the right, model predictions for values for the reward r at time t, the state value V, and the reward prediction error δ_t.

juice; see top plot in Fig. 7A)

- Over time, the response of dopamine neurons becomes weaker and weaker, until it completely disappears (see middle plot in Fig. 7A).
- At the same time, dopamine neurons begin responding to the *cue* instead of the *primary reward*, until the cue "absorbs" all of the response of dopamine neurons.
- If, after training, the reward is unexpectedly omitted, the activity of dopamine neurons drops below baseline (see bottom plot in Fig. 7A).

Schultz, Dayan, and Montague showed that this behavior can be understood if one identifies the response of dopamine neurons as the δ_t term. In this case, the experiment has two states, corresponding to the moment at which the cue is presented (s_{cue}) and the moment at which the juice is presented (s_{juice}). The only reward r corresponds to the juice and is delivered (obviously) at state s_{juice}. The agent has no actions, and all the $Q(s)$ values are simply associated with the different states (they are indicated as $V(s)$ instead of $Q(s)$ in Fig 7). Initially, the agent has no expectations, and the $Q(s)$ values of all states is zero. The first time a reward is given, the δ_t is positive (Fig 7B, top panel).

Following Eq. 6, the value of $Q(s_{juice})$ increases over time. The value of $Q(s_{juice})$ increases until its value is exactly equal to the reward r, that is, $Q(s_{juice}) = r$. At that point $\delta_t = 0$, and dopamine neurons stop firing.

In the meantime, the Q-value of state s_{cue} keeps increasing, because it consistently leads to $Q(s_{juice})$. As shown in Eq. 6, even in absence of reward, the value of $Q(s_{cue})$ will be updated until it matches $Q(s_{juice})$. As soon as the cue appears, the δ_t will be positive (because the primate knows that it has entered a state that, eventually, will lead to a reward) and dopamine neurons will begin firing (Fig 7B, middle panel).

Finally, if the primate enters state s_{juice} but, this time, no juice is delivered, the δ_t will be negative, because the high value of the juice state (which normally includes the reward) will be followed by a $r = 0$ (Fig 7B, bottom panel).

Implementing an Agent

Although the mathematics of RL are fairly simple, it is often difficult to imagine how such an agent would be implemented in practice—what does it mean, for example, to "update the value of the state at time t"? When are the updates made? Fortunately, in a simplified environment made of discrete and finite states, it is very easy to imagine such an agent and to write code that implements it. In this paragraph, we are going to sketch out how to do so.

Implementing an Environment

RL is based on two elements, an agent and an environment, that interact with each other (Figure 6). Thus, it is essential to define an environment or task that our agent needs to be interacting with. This environment is defined by the two transition functions, the state transition function and the reward transition function.

The "environment" does not necessarily need to be a self-contained world; very commonly it can be a simple task. In the case of the primate cue-learning task described above, for example, the "environment" is the sequence of predefined states "cue", "delay", and "juice". Because the task is a Pavlovian conditioning task, there are no "actions" that the animal needs to take; the environment transitions from one state to the other by itself. Furthermore, these states always repeat in the same order, so that the probability of transitioning from "cue" to "delay", or from "delay" to "juice", is always 1.

When defining an environment, it is common to introduce an *terminal* state, which we will indicate with the word `None`, that does not represent any "real" state and simply signifies the end of a trial. This is needed because, in RL, learning takes place in between the transition between states, and the terminal state is needed to learn the value of the very last state (in this case, to learn the value of s_{juice}).

Implementing an Agent with Lookup Tables

The imaginary agent that I am going to describe here, and which will be used as a reference for most of this chapter, maintains a memory of all the states that it has encountered in a lookup table. This table, which I am going to call the *V-table*, is an approximation of the V function, and it simply lists all the possible environment states $s_1, s_2 \dots s_N$ in one column, and the agent's current estimates of their values $V(s_1), V(s_2) \dots V(s_N)$ in the other.

All of the agent's operations can be reframed in terms of this table. When the agent is evaluating a given state, it is just scanning the table until it finds the corresponding entry in the state column and returning the corresponding number on the same row in the value column. When the agent is learning, on the other hand, it is simply updating a specific number of the value column. So, the operation $V_{new}(s) \leftarrow V_{old}(s) + \delta$ translates to "add the number δ to the number in the value column that corresponds to the row where that has s on the state column".

For example, in the case of the primate experiment described by Schultz, Dayan, and Montague (Figure 7), the V-table would look, at the very beginning, like Table 2.

State s_t	**Value** $V(s_t)$
Cue	0
Delay	0
Juice	0

Table 2: An example of a V-Table for an RL agent performing the experiment of Schultz, Dayan, and Montague.

Figure 8 shows the results of running a simple agent that learns what to expect from this simple task over time. The Figure shows the changes in the V-table as the agent learn, sampled at times $t = 1, 10$ and 100.

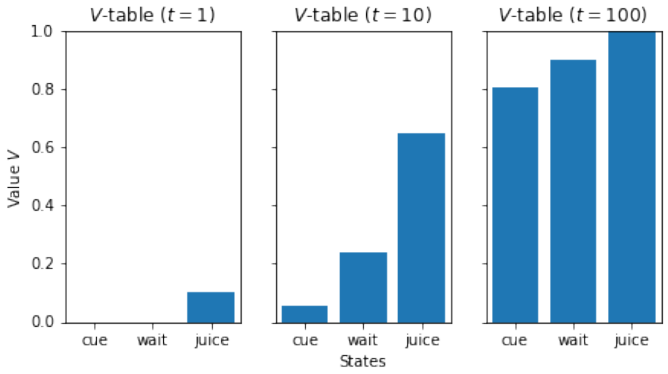


Figure 8: Changes in the V-table as an imaginary agent learns in the primate cue-learning task described in the paper by Schultz, Dayan, and Montague.

A More Complex Example

Now, let's imagine a slightly more complicated agent: one that simulates a mouse navigating a maze in search of food. The maze is made of 16 cells arranged in a 4-by-4 pattern, like the one in Figure 9.

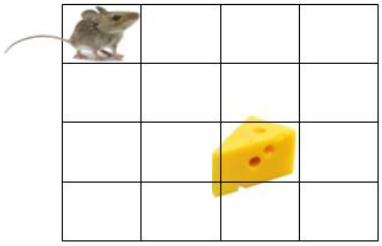


Figure 9: A simulated mouse in a 4-by-4 maze, with a reward (cheese) in position (3,3)

The maze is composed of 16 cells, which we can indicate by the pair of their (x, y) matrix coordinates; for the example, the mouse in Figure 9 is located in cell (1,1). The environment also contains one block of cheese in cell (3,3), which will function as a reward for the agent.

In this simplified world, the environment's states are the maze's cells and can also be named after their coordinates; for example, the initial state is indicated as $s_{1,1}$. In the maze, a reward is only given when the agent reaches the cell in which the cheese is located, corresponding to state $s_{3,3}$. Thus, the reward function is defined as such that $r_t = 1$ if the current state $s = s_{3,3}$, and $r_t = 0$ in all other cases (that is, when $s \neq s_{3,3}$).

In this environment, a *trial* consists of an agent's random walk through the maze until it finds the cheese reward. At that point, the trials ends–an event

that is represented, again, by the special terminal state `None`.

In Figure 8, the agent's V-table was visualized as a bargraph, with the height of each bar representing the value of the corresponding state. This representation can be used for the maze environment as well. However, since the maze is a 2D world, it is more intuitive to visualize the V-table as a matrix, with each matrix cell corresponding to the homologous position in the maze. Instead of bars of different heights, the value of each state will be visually represented by the color of the cell in the matrix. Using this representation, Figure 10 shows how the internal representation of the V-table values change as they are learned by the mouse agent after 1, 10, or 1,000 trials (that is, random walks).

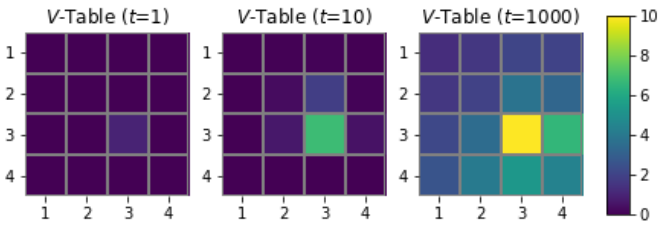


Figure 10: Representation of the Internal V-table of an agent exploring the maze of Figure 9. In each matrix, the color represents the intensity of the V value associated to each cell. The colorbar on the right illustrates the mapping between color and V value.

The progression of learning in Figure 10 is very typical. The agent learns on the first trial that state $s_{3,3}$ contains a reward and has, therefore, a higher value (shown by the slightly lighter hue). However, it will take a while for this learned value to propagate back to previous states. Because the agent moves at random, the probability of reaching the reward can be calculated exactly, and corresponds to the probability of taking the consecutive steps that must be taken to get to cell (3,3). In the four cells that are located immediately North, South, East and West of the cheese, for example, the simulated mouse has 1 out of 4 chances of reaching the reward in the next move; for this reason, the values in these states are approximately 1/4 of the value of the reward state $s_{3,3}$. The states that are immediately North-East, South-East, North-West, or South-West of the cheese have a 1/2 $\times$ 1/4 chance of reaching the cheese, because each of them has two opportunities out of four of landing on a state that has a 1/4 chance of leading to the cheese, and so on. As the mouse gathers more and more experience in the maze, the values of the states better approximates these theoretical values.

Learning the Values of Actions

The Agent's Policy

The previous sections above have described how an agent *learns*. However, they says almost nothing about how an agent *acts*. In RL, the two concepts

are kept distinct; Equation 5 could be used to correctly estimate future rewards whether an agent picks its actions at random (as in the example of Figure 10) or it always picks the best action based on current estimates. Of course, an agent that picks at random will collect less rewards over time and learn more slowly, but it will, nonetheless, still learn the best course of action. Nonetheless, *knowing* what is the best course of action does not dictate how an agent chooses its next action. Conceptually, the agent can behave poorly even if it knows better [19].

[19] Most of us do; therefore, no judgement.

The procedure by which an agent chooses what to do is called its *policy*, and, as anticipated in the first section, it is defined as a function π that is, in essence, a probability distribution function over all the possible actions $a_1, a_2, \ldots, a_n$.

In general, a *good* policy should balance exploration vs. exploitation, and select the best actions (based on current estimates) most of the time while also leaving some room to explore actions that are currently perceived as suboptimal.

There are two typical solutions that are used for agents. The first solution is the so-called ε-greedy policy. In this policy, the agents sets a small threshold ε, then draws a random number between 0 and 1. If the number is greater than ε, then the agent performs the action with the highest possible Q value. If not, the agents pick at random among all of the possible actions that are not the best. Formally, this can be expressed as such:

$$\pi : a_i \rightarrow P(a_i) = \begin{cases} 1-\varepsilon & \text{if } Q(a_i) = \max_a Q(s_t, a) \\ \varepsilon/(n-1) & \text{otherwise} \end{cases} \tag{7}$$

The second solution, somewhat more common and more elegant, is to use Boltzmann's equation to assign probabilities to each action based on their current Q-values:

$$\pi : a_i \rightarrow P(a_i) = \frac{e^{Q(s,a_i)/\tau}}{\sum_j e^{Q(s,a_j)/\tau}} \tag{8}$$

In Eq. 8, the $0 < \tau < \infty$ parameter represents the decision *temperature*, and determines the balance between exploration and exploitation. As $\tau \rightarrow 0$, the agent becomes more exploitative and greedy, deterministically picking the action with the highest Q-value; and, as $\tau \rightarrow \infty$, the agent becomes more explorative, selecting actions at random.

SARSA

The idea behind TD-learning can be extended to the case of an agent's *actions*. In this case, the agent learns by estimating a value function $Q(s,a)$ that assigns a value Q to every action a that can be taken in every specific state s. In a discrete environment, we can imagine the agent maintaining an internal Q-table, which is analogous to the V-table described above, but

contains three columns: one for all states, one for all the actions associated with each state, and one for the Q value associated with each possible state and action combination. Again, as in the case of the state values, the agent learns to approximate the value Q of an action a in a state s as the number of future rewards that can be expected from now on.

To efficiently learn Q, the agent simply compares different subsequent estimates. At any point in time, if $Q(s_t, a_t)$ at time t perfectly reflects R_t, then $Q(s_{t+1}, a_{t+1})$ also perfectly reflects R_{t+1}. But, by definition, $R_t = r_t + \gamma R_{t+1}$ (Eq. 2) so that, in a perfect world:

$$Q_{new}(s_t, a_t) \leftarrow Q_{old}(s_t, a_t) + \alpha[r_t + \gamma Q(s_{t+1}, a_{t+1}) - Q_{old}(s_t, a_t)] \tag{9}$$

This particular algorithm is known as SARSA [20], from the mnemonic of the various quantities used in the updated rule (s_t, a_t, r_t, s_{t+1}, a_{t+1}, respectively). In this learning rule, the agent is ready to update the entries of the Q-table as soon it has perceived the new state s_{t+1} and has decided the following action a_{t+1}.

[20] Gavin A Rummery and Mahesan Niranjan. *On-line Q-learning using connectionist systems*, volume 37. University of Cambridge, Department of Engineering Cambridge, UK, 1994

Implementation

An agent that learns to perform actions can be implemented in the same way as described can be implemented in the same way, with only a few minor modifications. The first is that, instead randomly choosing actions (as in the case of Figure 10), the agent will implement an algorithm such as Equations 7 and 8 . The second is that, instead of a lookup table for V values, our agent would have a Q table with three columns: one that indexes the state, one that indexes the actions for each state, and one for the Q-values associated with each action in each state.

Although the agents described in Section I already had actions that allowed it to navigate the maze, we did not provide any definition for them. Now, we can get into more detail. The simulated mouse agent possesses a set of four actions, $\mathscr{A} = \{\text{Down}, \text{Left}, \text{Up}, \text{Right}\}$, which are available at every state and have the consequence of moving the agent to a new cell. The state transition function is defined as

State s	**Action** a	**Value** $Q(s,a)$
$s_{1,1}$	Down	0
$s_{1,1}$	Left	0
...	...	...
$s_{4,4}$	Right	0

Table 3: An example of a Q-Table.

We can visualize such an example. As in Figure 10, we can visualize the Q-values as matrices, so that their spatial arrangement is consistent with the maze. Since we have four different actions, we can visualize the entire Q-table as four matrices, showing the value of the corresponding action in each position of the maze. One such result is given in Figure 11, which illustrates the Q-values for the four actions of an agent who has learned how to navigate the maze environment using SARSA over 500 different trials using an ε-greedy policy (with $\varepsilon = 0.1$).

Because the agent is using an ε-greedy policy, none of the four cells looks resembles the smooth gradients of the V-table in Figure 10. This is

because the policy, by selecting the best actions more often, prevents and even sampling of the space of possible actions, at the limit ensuing that certain cells, such as (4,4) are seldom, if ever, explored.

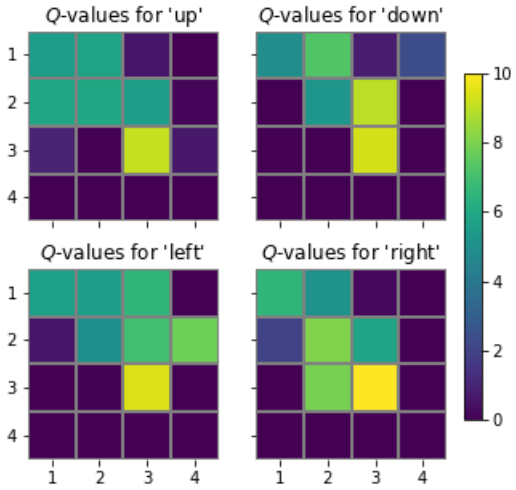


Figure 11: Q-Values in the lookup tables of an agent learning to navigate the maze of Figure 9

Q-Learning

In his original paper [21], Sutton proved the optimality of TD-learning. So, it seems only natural to assume that, if TD-learning is optimal and converges on the true values of V for every state, that SARSA would the same, and converge on the true values of Q for every state. However, one should ask, as Thor did, "is it, though?".

The answer is no, it isn't. The key is that, in TD-learning, we are estimating quantities (the V-values associated with each sate) that do not affect, *per se*, how state transitions happen. But, in SARSA, the agent is actually trying to estimate the value of actions, and actions do affect how the environment changes states. For this reason, the specific way in which the agent chooses the next action, a_{t+1}, affects the eventual values in the Q-table [22].

This problem is solved by Watkins and Dayan [23] with an algorithm called Q-learning. In Q-learning, the update rule is the following:

$$Q_{new}(s_t,a_t) \leftarrow Q_{old}(s_t,a_t) + \alpha[r_t + \gamma \max_a Q(s_{t+1},a_{t+1}) - Q_{old}(s_t,a_t)] \quad (10)$$

In Eq. 10, the term $\max_a Q(s_{t+1},a_{t+1})$ represents the Q-value of the action with the highest possible value among all of those available in state s_{t+1}. Note that, under this definition, the action with the highest possible value might not be the action effectively taken by the model.

[21] Richard S Sutton. Learning to predict by the methods of temporal differences. *Machine learning*, 3(1):9–44, 1988

[22] SARSA does, however, other desirable characteristics; notably, as we will see, it greatly simplifies the use of eligibility traces.

[23] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992

Thus, the key difference between SARSA and Q-learning is the action that, at the new state s_{t+1}, is chosen as the yardstick against which the previous action's Q-value is compared, and which will determine the size and direction of the RPE. In SARSA, the action is simply the next action that agent has already decided to take. In Q-learning, however, the action is the best possible action that is available at state s_{t+1}, *whether the agent takes it or not*. In RL lingo, SARSA is said to be a *on-policy* learning rule, because its updates depend on the actions that were effectively taken. Q-learning, on the other hand, is said to an *off-policy* learning rule because the agent learns from the best estimates so far, whether they reflect its behavior or not.

The difference between the results of the two types of learning can be striking. Let's consider, as an example, a variation of the original maze example in Figure 9. In this case, the maze is being modified by adding a "pit" running in through cells (1,2) and (1,3) in the upper row. To simulate the nefarious effects of falling in the pit, the states $s_{1,2}$ and $s_{1,3}$ have a reward value of $r = -10$. Finally, the cheese block is placed right behind the pit, in cell (1,4).

The difference is illustrated in Figure 12. Each plot in the Figure shows the probability that each cells is visited during a single trial, after learning has occurred with either Q-learning (left) or SARSA (right). In this particular example, both agents were trained for 1,000 trials and both used the same ε-greedy policy with $\varepsilon = 0.1$.

Because there is a non-zero chance that the model would accidentally take the "Up" action while running in the horizontal segment of the maze towards its rewards, SARSA eventually learns to take a longer path, which minimizes the risks. Q-learning, on the other hand, is insensitive to this risk, and learns what is effectively the optimal (that is, shortest) path to the reward, *even if* the agent might sometimes incur in an accidental cost.

Figure 12: Preferred navigation paths in the 4 by 4 maze after 1,000 learning trials with Q-learning (left) or SARSA (right).

SARSA and Q-learning in the Brain

The results of the previous paragraph seem to suggest that a biological agent should rely on SARSA, rather than Q-learning: although it cannot be demonstrated to converge on optimal values, SARSA does take the agent's

actual behavior into account and avoids overexposing to potentially choosing bad actions. But, common sense considerations aside, is this also true of the brain?

Learning in Non-Markov Environments

A known limitation of the simple version of RL described herein is that it cannot be applied to non-Markov environments. In non-Markov environments, the back-propagation of the error term would not work properly, and actions would not get proper credit for later rewards.

Consider, for example, this simple variation of the environment used by Redish [24] in his study of addiction. In this environment, rewards at the end, delivered in the two states marked as "Win" and "Loss", are dependent on the two possible states, "A" and "B", that branch out of the initial, "Start" state.

[24] A David Redish. Addiction as a computational process gone awry. *Science*, 306(5703):1944–1947, 2004

Multiple-step Backups: TD(n) and Q(n)

Perhaps the simplest solution to this problem is to extend the scope of the update window of the temporal difference algorithms. All of the algorithms we have seen so far involve a backup term, which updates the values of a state or an action at time t based on the new values at times $t+1$. This backup term creates a sort of chain rule, which propagates the RPE term δ_t back in time.

There is nothing that prevents us from using a larger window, and updating the values at time t based on the experiences at time $t+2$, for example.

Eligibility Traces

The typical solution to this problem is to augment the RL agent with a form of internal "memory" of its previous states. The memory of a state s is represented as an *eligibility trace* $e(s)$. Initially, each trace is set to zero. Then, every time the agent enters a new state s', *each* of the eligibility traces in memory are updated according to the following rule

$$e(s) = \begin{cases} \lambda e(s) & \text{if } s' \neq s \\ \lambda e(s) + 1 & \text{otherwise} \end{cases} \tag{11}$$

in which $0 < \lambda < 1$ is a decay parameter. Thus, eligibility traces decay over time but increase with re-use, roughly capturing the role of recency and frequency in memory. The eligibility traces can be used to bridge the temporal gap that separates an action from its temporally delayed consequences in a non-Markov environments.

TD-learning with Eligibility Traces: TD(λ)

The eligibility traces can be easily integrated into TD-learning, creating a variant known as TD(λ). In TD(λ), the original TD-learning algorithm is

modified as follows. After all of the eligibility traces in memory have been updated, *all* of the $V(s)$ values for all states are also updated in proportion to the value of of the corresponding eligibility trace.

$$V_{new}(s) \leftarrow V_{old}(s) + \alpha \delta_t e(s) \tag{12}$$

The key aspect of Equation 12 is that, once a δ_t value is calculated by comparing predicted and observed values of a state, the prediction error is propagated back in time to *all* states in proportion to their distance in time, which is embodied by the $e(s)$ term. In other words, TD(λ) can be seen as a version of TD(n) with "fading" memories, where the size of the backup terms decays with the temporal distance. In, fact, we can think of TD(λ) as a generalization of TD(n): When $\lambda = 0$, TD(λ) reduces to TD-learning, or TD($n = 1$). When $\lambda = 1$, on the other hand, TD(λ) approximates TD($n \rightarrow \infty$).

Implementation

To implement a TD(λ) agent, we just need to modify the existing TD-agents in three ways. First, in addition to the existing lookup table for V values, we need to add an E-table that associates each state state with the current value of its eligibility trace. At the beginning, all of the entries of the E-table are zero. Second, the learning rule needs to be modified so that, when a new state s' is perceived, all of the entries in the E table are updated according to Equation 11. Third, once E is updated, all of the values of the V table are updated according to Equation 12.

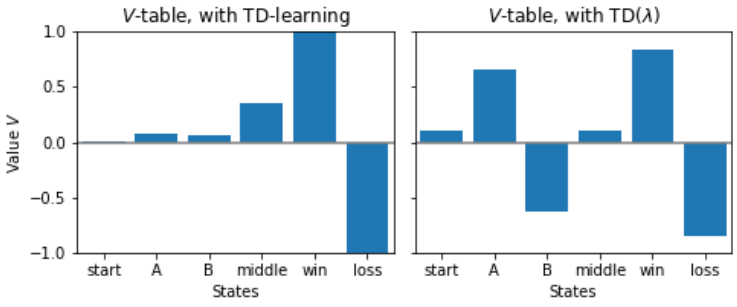


Figure 13: V-tables of an agent learning a non-Markov environment using the standard TD-learning algorithm (left) or TD(λ) (right). The latter can successfully recovery the long-distance dependency between the rewards in the "win" and "loss" states and the earlier "A" and "B" states.

We can also examine how eligibility traces are applied in the case of the TD-learning agent navigating the grid maze and described in Section I. In this case, the agent performs a random walk at every trial, and the states represent the cells visited in its path. Let us imagine an agent that, starting in the usual location in cell $(1,1)$, performs the following series of moves:

Eligibility Traces for Actions

What happens in the case of methods that estimate the values of actions, such a Q-learning and SARSA? In the case of SARSA, translating TD(λ) is trivial.

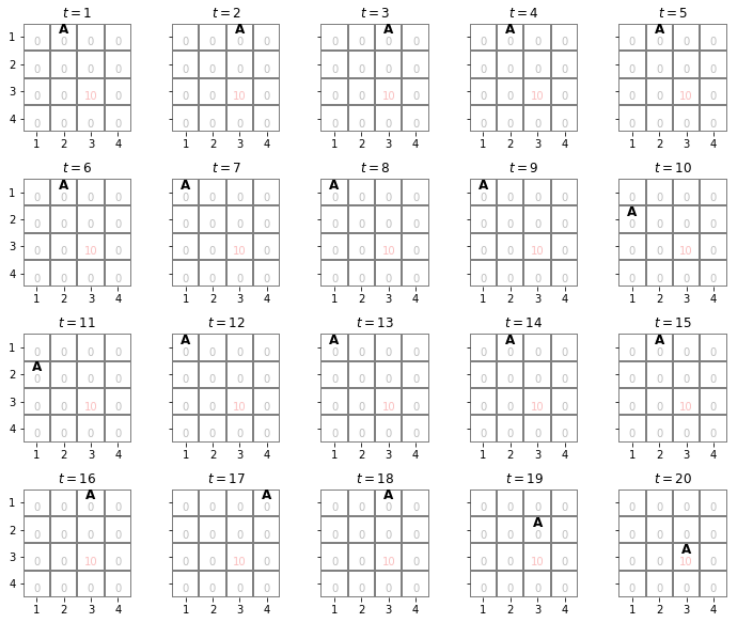


Figure 14: A 20-step random walk of a agent moving in the maze environment until it finds the reward

To do so, we first modify the definition of eligibility trace, from a decaying memory of visiting a given state, $e(s)$, to a decaying memory of performing a specific action in a given state, $e(s,a)$. Then, we modify Equation 9 along the lines of Equation 12, obtaining what is known as SARSA(λ):

$$Q_{new}(s,a) \leftarrow Q_{old}(s,a) + \alpha \delta_t e(s,a) \tag{13}$$

But what about Q-learning? As it turns out, such a straightforward transformation is not possible. This is because in Q-learning (Equation 10) the backup term is based on the best action available $\max_a Q(s_{t+1}, a_{t+1})$, which might or *might not* be executed at time $t+1$. However, if the best action is not chosen, how should its eligibility trace be updated? Should it be increased by one, because it was still considered? Or should the eligibility trace decrease, because it was not actually executed and experienced?

These problems were actually noticed as soon as Q-learning was presented [25]. To solve this problem, Watkins and Dayan proposed a mixed strategy, which they called $Q(\lambda)$ and in which the use of the eligibility traces depends on the policy π. In this strategy, the model follows SARSA(λ)'s rule in Equation 13 so long as the action picked by model is also the best action in Q-learning's algorithm, that is, as long as $Q(s_{t+1}, a_{t+1}) = \max_a Q(s_{t+1}, a)$. However, when the policy selects an action that is *not* the optimal action, Q(λ) resets all the eligibility traces to zero. Because the current state s_t's

[25] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992

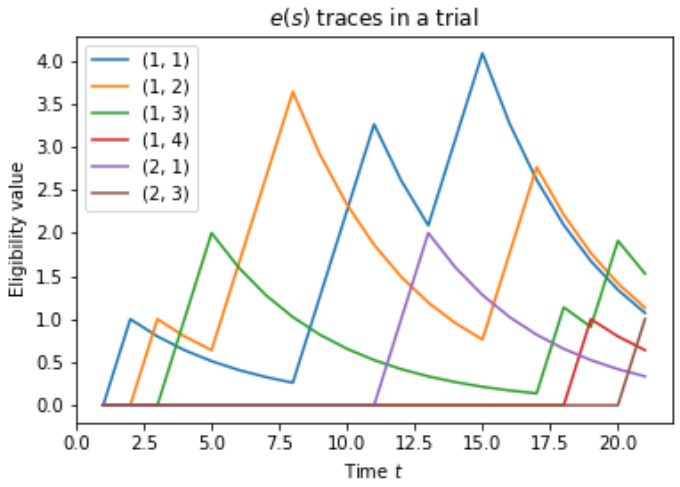


Figure 15: Timecourse of the eligibility traces as the agent walks in the path of Figure 14

.

eligibility trace still gets incremented by 1 (as in Equation 11) and all the other traces are now zero, the learning algorithm reduces to Q-learning for this trial. In practice, this algorithm uses SARSA(λ) as long as there is an uninterrupted sequence of optimal actions, and resets itself, starting a new set of traces, whenever the policy deviates from optimal.

Learning the Policy With Actor-Critic Methods

So far, our agents have maintained three separate components: A V-table for states, a Q-table for actions, and a policy. The distinction between the policy and the Q-table, however, is somewhat blurred; as we have seen, certain algorithm, like SARSA, will find different solutions based on the policy that is actually being used (see Figure 12). Furthermore, most reasonable policies would depend on the values of actions, as is the case for the policies described in Equations 7 and 8.

Intuitively, it does not make much sense to keep the policy entirely separated from what an agent learns about the value of its own actions, although it might be conceptually cleaner from a computational standpoint. If we consider biological agents, the distinction is even less sensible, as it is clearly the agent's objective to quickly learn how to *act* in a given environment and how to avoid making mistakes. or this reason, scientists have researched ways in which RL agents can be simplified and their policy could be learned together with the value of their actions.

The result of this line of research has been a series of algorithms known as *Actor-Critic Methods*. In these algorithms, the RL agent is divided into components: an *Actor* that chooses the actions to perform in a given state and a *Critic* that uses RL methods to learn the values of different actions in an environment. Thus, Actor implements the policy function, and the Critic

provides feedback to the Actor's choices. This way, AC agents use what they have learned to directly update the policy.

The Advantage Actor-Critic (A2C)

.

Although the Q Critic unifies the Q-table and the policy, it remains an agent that learns the values of states and actions separately. That is certainly convenient from an engineering point of view; depending on the specifics of the environment we are facing, we can choose whether to compute V-functions and tables (when, for instance, we are interested in states and have little power over the environment) or Q-functions and tables (when, for instance, we have much more control over what we do).

But living creatures have no such luxury. They do not get to choose the environment they live in; if they do, they do not get to neatly pick the best way to design their own brain to select the most convenient algorithm; and, even if they were to do so, they would not have the comfort of a well designed and characterized environment, one for which there is a good solution that would remain constant across their lifespan.

As a consequence, researchers have been investigating whether there is a way to unify the ideas behind state- and action-based temporal difference methods, so that agents would not need to learn separate V and Q tables and perform redundant learning steps after perceiving any new state.

The Advantage Actor-Critic (A2C) architecture is perhaps the most intuitive way to implement an Actor-Critic. Instead of a Q-function, the A2C agent learns a so-called Advantage function, which is defined as such:

$$A(s,a) = V(s) - Q(s,a) \tag{14}$$

In practice, the advantage of an action a in state s is how much it can improve over the standard expectations for that state, $V(s)$. The expectations for $V(s)$ depend on the policy.

The key advantage of the A2C architecture is that we can now update $A(s,a)$ after observing the next state and its value, $V(s_{t+1})$. Therefore, we can use the TD-learning equation 6 to calculate δ_t, and use δ_t to update $A(s,a)$:

$$A(s_t,a_t)_{new} = A(s_t,a_t)_{old} + \beta\delta_t \tag{15}$$

Notice the massive simplification carried out in the A2C agent. Instead of keeping track of multiple tables and policies, the A2C agent can be implemented with minimal use of memory resources. All that this agent needs is a V-table to keep track of the states, and an implementation of TD-learning to update expectations. The policy can be implemented directly by keeping track of a series of state-action associations that are directly updated whenever δ_t is calculated. Once implemented the agent can be easily deployed in

both Pavolvian and instrumental conditioning situations without any chance to the architecture.

The Actor-Critic Framework in the Brain

At this point, it should come at not surprise that Actor-Critic agents, and A2C in particular, bear a strong resemblance to the architecture of the basal ganglia, and in particular of the basal ganglia's central nucleus, the striatum [26]. The striatum is made of two different components, the *ventral* striatum, also known as *nucleus accumbens*, and the *dorsal* striatum. Both components receive inputs from large portions of the cortex; however, the ventral part receives dopamine inputs from the VTA, and the dorsal part projects back to the motor cortex.

In fact, the architecture of the basal ganglia circuit bears a remarkable similarity to the design of the A2C critic [27].

[26] Andrew G Barto. Adaptive critics and the basal ganglia. *Models of information processing in the basal ganglia*, 215, 1995

[27] Yuji Takahashi, Geoffrey Schoenbaum, and Yael Niv. Silencing the critics: understanding the effects of cocaine sensitization on dorsolateral and ventral striatum in the context of an actor/critic model. *Frontiers in neuroscience*, 2:14, 2008

Model-Based RL

All of the methods that have been described so far belong to a specific form of RL called *model-free*. It is called as such because the agent does not have to maintain (i.e., it is "free" from) an internal model of the environment to act properly in the world. Unsurprisingly, these models require remarkably little in terms of computations and memory, and they provide a nice framework to explain the function of those brain circuits that, through repeated rewards, form habits and stimulus-response associations, such the basal ganglia [28].

Also unsurprisingly, model-free RL is fairly limited and, being habitual by nature, it is also inflexible. Suppose, for instance, that our imaginary mouse is placed again in the maze that has been used in the previous examples–only, this time, the mouse is *not* hungry. Following RL, the mouse would find itself drawn to the cheese location, whether it has any desire to eat cheese or not. And, if the mouse is thirsty, its previously learned Q-values would be essentially useless, as they would keep directing the animal towards the cheese but provide no clue as to where to find water.

To overcome these difficulties, a *smart* agent needs to able to understand and predict the how the environment would change. This, in turn, require forming an internal model. In formal terms, this internal model is a representation of the environment's state transition function $P(s_t, a_t, s_{t+1})$.

In our simplified and discrete domain, we will assume that the agent uses yet another table, which we will call the S-table. The S-table contains four columns. Like the Q-table, the first two columns represent the current state s_t and action a_t, and will be used to identify the correct row. The last two columns store the new state s_{t+1} and the associated reward r_{t+1}:

[28] Henry H Yin and Barbara J Knowlton. The role of the basal ganglia in habit formation. *Nature Reviews Neuroscience*, 7(6):464–476, 2006

State s_t	**Action** a_t	**Reward** r_{t+1}	**State** s_{t+1}
$s_{1,1}$	Right	0	$s_{1,2}$
$s_{1,1}$	Down	0	$s_{2,1}$
...	...	...	...
$s_{2,3}$	Down	1	$s_{3,3}$
...	...	...	...

Table 4: An example of S-Table: Combinations of states and actions are associated with the consecutive state and associated reward.

One-step Q-planning

Perhaps the simplest model-based RL algorithm is the so-called one-step Q-planning. In this algorithm, it is assumed that agent is equipped with a policy *pi* on how to choose actions based on their Q values, but has not learned the values of any action yet (that is, the Q tables are empty). The agent, however, does have a model of the environment (an S-table) to start with. Such a situation could arise, for example, when an agent is not acting in a particular environment but is allowed to witness another agent's actions, thus acquiring knowledge about state transitions without ever experiencing the consequences of their actions.

In this situation, the agent needs to transfer its knowledge from the S-table to the Q-table before being able to apply its policy. To do so, the algorithm performs a sort of mental simulation: it will vicariously experience the effects of the actions by repeatedly selecting a random state s and a random action a associated with it from the list of possible states it has learned, and using its S-table to simulate the consequences of a. After looking up the following s_{t+1} and reward r_{t+1}, the agent can then apply Q-learning and update its own Q-tables. Because this process only requires looking up entries in the agent's memory, it can be repeated for a very large number of times, always with a new random state and action, resulting in Q-learning to converge. In fact, because the states are chosen at random, the agent can sample all the possible combinations of states and actions during the planning phase, generating a better coverage of the values of actions in every part of the maze. As an example, Figure 16 shows the Q-table of a one-step Q-planning agent after the first planning step. Note how different the Q-table looks from the one in Figure 11. When the planning step is completed, the agent can finally use its policy π to act in an environment that it has never experienced directly.

Note that "planning", in this sense, is not necessarily the formulation of a chain of steps that the agent will follow. Rather, it is the repeated simulation of events that allow the knowledge encapsulated in the S-table to transfer to the Q-table. Similarly, the "one-step" part of the algorithm might require many thousand cycles of simulation from this transfer to take place; it can be considered as a single step only because it does not involve any physical action with the environment, and can be entirely contained between the moment in which a new state is perceived and the moment at which a new action is selected.

The flexibility of model-based RL stems from the fact that, if the environment were to suddenly change, or if the agent were to change its goals and decide that, for example, cheese is not as much a reward as water, the agent would only need to change the entries in the S-table and, in a single iteration of one-step Q-planning, would be able to relearn an entirely different set of Q-tables that would be better suited for the new goals or the new environment.

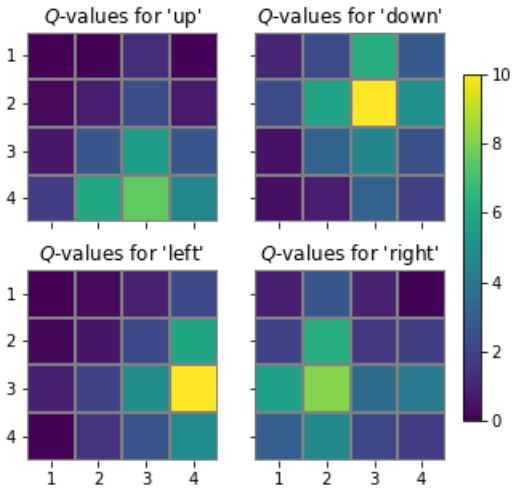


Figure 16: Q-table of a model-based agent implementing a one-step Q-planning algorithm with $n = 1,000$ simulation steps. Before the first move, the agent has precise knowledge of the exact value of each action in each state

.

Dyna-Q

The situation described above is highly unusual, as the agent has access to complete model-based knowledge before being required to perform any action in the environment. A much more likely case is one in which the agent is tossed into a new environment, and the agent is left with learning both model-based and model-free representations—that is, it starts with uninitialized or empty S-tables and Q-tables. In this case, the most common solution is an algorithm known as Dyna-Q [29].

In Dyna-Q, the agent starts by performing the planning step through one-step Q-planning. That is, the agent first updates its own Q-tables based on the existent model of the environment, if any. If the model is empty, the agent imply picks at random. Either way, this first step results in an action a_t being chosen and executed. The execution of a_t results in the environment undergoing a state transition, upon which the agent perceives a new state s_{t+1} and a new reward r_{t+1}. The new state and reward to used to perform two updates: First, it is used to update the agent's Q-tables; second, it is used to update the agent's model of the environment and its S-table. Thus, after every action, the agent updates both the model-based and the model-free representations.

By leveraging both model-free (Q-tables) and model-based (S-tables) at the same time, Dyna-Q is usually much faster and more efficient than any model-free algorithm. The image below the path taken by an agent learning with Dyna-Q (top row) and by Q-learning (bottom row) at times $t = 1, 10, 20$. Both models have the same model-free parameters ($\alpha = 0.1$, $\gamma = 0.9$). It is apparent that Dyna-Q learns much faster.

[29] Richard S Sutton. Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. In *Machine learning proceedings 1990*, pages 216–224. Elsevier, 1990

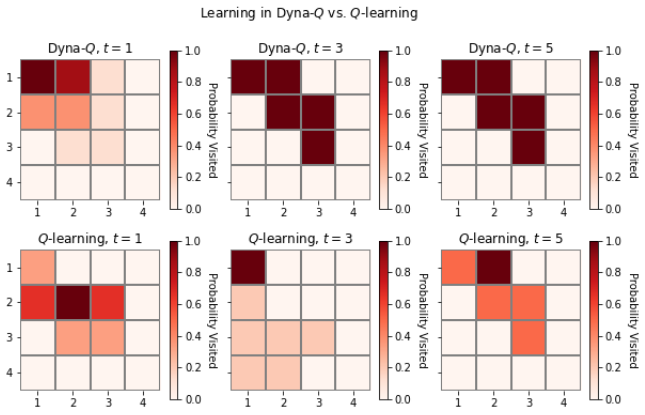


Figure 17: A comparison of Dyna-Q and Q-learning as an agent learns to navigate the maze of Figure 9

.

Model-Based RL as Declarative Memory

From the neuroscientific point of view, it is interesting to consider Model-Based RL as another way in which RL can be complemented by declarative memory, akin to the case of eligibility traces in Section I. In fact, a specific connection can be again made to the old animal learning literature. A lengthy debate that spanned centuries was whether animals who were trained to perform a particular task in the lab were actually learning "stimulus-response" (S-R) or "stimulus-stimulus" (S-S) associations. From the point of view of RL, S-R associations can be easily identified with the Q-function and the Q-table (which maps a state-action pair onto its estimated cumulative reward), while an S-S association corresponds to the state transition function and the S-table of model-based RL. Thus, it is apparent that animals can do both, and can learn either representation.

This was famously shown by Edward Tolman in a series of experiments in the '30s. Perhaps the most famous of these experiments was one in which Tolman and colleagues trained three groups of rats to perform a maze [30]. Each animal was put at one end of the maze, and was taken out when they reached the opposite end of the maze. For animals in the first group ("R" for "Reward"), a food pellet was placed at the end of the maze. Animals in the second group, instead, were not given any reward ("NR", for "No Reward", in Figure 18).

Unsurprisingly, Tolman found that animals in the first group learned faster than animals in the second group. As a learning measure, they recorded the number of "errors" that animals made during a trial, i.e. the number of turns they made that were not along the shortest path to the food reward. As shown in Figure 18, the first group shows a much sharper drop of errors over time than the second[31].

A third group of rats, however, began receiving the reward only at the 11th day of training (NR-R in Figure 18, for "No Reward - Reward"). Surprisingly,

[30] Edward C Tolman. Cognitive maps in rats and men. *Psychological review*, 55(4):189, 1948

[31] In case you are wondering why the NR rats also show learning... Keep in mind that getting out of a lab experiment is rewarding as well, especially if there is no food at the end!

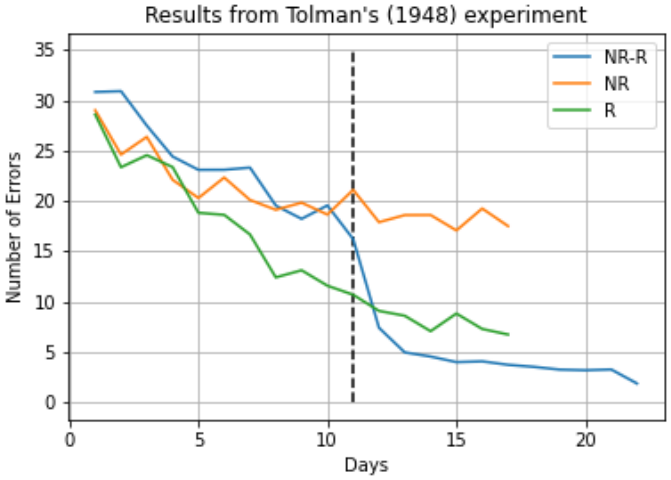


Figure 18: Results from Tolman's original experiment demonstrating the existence of "cognitive maps"

Tolman reported that, from that moment on, these rats performed exactly as well as the rats who had been receiving the rewards since Day 1; in fact, they even slightly outperformed that group. Tolman concluded that, whatever learning was going on in this third group, must have occurred in the first 10 days and at the same rate as the first group, even in absence of rewards and overt behavior. He specifically put forward an hypothesis that was hugely controversial at that time—that rats were actually learning "cognitive maps".

Of course, that was *exactly* what the rats were learning.

Model-Based RL as Planning

Although classic studies like Tolman's [32] showed that animals can learn S-S associations that are, in essence, analogous to the entries of our agent's *S*-table, they say nothing about how these representations are ultimately used by an agent. In the process of deriving new *Q*-tables, the agent engages in a repeated simulation of outcomes that are encoded in declarative memory.

However, another way in which the agent can decide how to act is by carefully examining all of the possible outcomes of a choice in its own head before making a move. This process is called planning, and requires a considerable amount of resources in terms of both episodic memory (to build the *S*-table and remember the states) and in terms of working memory (to build and maintain a tree of possible options).

[32] Edward C Tolman. Cognitive maps in rats and men. *Psychological review*, 55(4):189, 1948

Accumulator Models of Decision-Making

Reinforcement learning models describe how agents learn from their environments and make decisions. However, while the learning aspects of Reinforcement Learning have been examined in details and mapped to specific aspects of brain function, the neural underpinnings of their decision part has been overlooked. In essence, RL agents delegate the decision to “policy” that works like an oracle: given a state, the policy would return the action to perform. But *how* is such a decision exactly made? Unlike the detailed mechanisms by which V and Q tables are updated, RL models are silent about the mechanics of decision making.

And yet, decision-making is such an important aspect of cognition that it cannot be ignored. In addition to choosing between alternative actions, organisms face decisions across of a variety of other circumstances, such as deciding which of two pieces of food is bigger, or even, simply, whether a particular object is predator or not.

Fortunately, a specific family of models exists that have been designed specifically to capture and account for the dynamics of decision-making. These models, which exist in many variations, are collectively referred to as *accumulator models*; they are also known, in other fields, as diffusion models or sequential sampling models.

Accumulator models are based on the idea that the decisions process unfolds over a period of time. In that period, evidence in favor of one or more options is accumulated. A decision is made as soon as the evidence for one option exceeds a predefined amount. There are two main families of accumulator models: diffusion models and race models.

The Drift Diffusion Model

The first category of accumulator models we are going to see is called Diffusion models. These models were designed the handle what is perhaps the most prototypical case of decision-making–the case in which there are only two options, and the decided must choose one of them. This situation is technically known as a "two-alternative forced choice task", or 2AFC.

The simplest paradigm in this sense was popularized by Ratcliff [33], and has become known as the “Drift-Diffusion Model” or DDM.

[33] Roger Ratcliff. A theory of memory retrieval. *Psychological review*, 85(2):59, 1978

As in all accumulator models, in DDMs a decision is made by accumulating evidence over time towards one of two. Specifically, a DDM has at least three parameters:

- The *drift rate v*. This is crucial parameter in a DDM. The drift parameter captures the degree to which evidence moves, on average, towards one option or the other.
- The *decision boundary A*. By convention one of the two options is associated with the value of zero while the other is associated with the boundary $A > 0$. The models begins its meandering drift at the value in between the two options, that is, $A/2$.
- The *non-decision time Ter*. Although the model is designed to model the time it takes to make a decision, in real experiments the response time include other factors that have nothing to do with the decision process *per se*, such as the time necessary to move the eyes and fixate a stimulus or the time to initiate and button press. All of these factors are assumed to be constant across a single experiment, and compounded into a single factor that accounts for the "non-decision time". This parameter is called *Ter* (from its original name, "Time for Encoding and Response").

In Ratcliff's original formulation [34], the drift-diffusion model is continuous and the drift rate can be understood as a force vector constantly applied to a moving particle. This approach leads to very elegant but complicated math. In most applications, the model is simplified as discrete, with accumulation of evidence occurring at small, discrete steps over time and in fixed increments towards one of the two options. In the discrete case, the drift rate v is interpreted as the probability that, in a given time increment, the evidence will accumulate towards the boundary A. Thus, the drift value is constrained to be $0 <= v <= 1$, the value of $v = 0.5$ represents indifference between the two options, and any value $v > 0.5$ indicates a preferences towards the option associated with the boundary A.

[34] Roger Ratcliff. A theory of memory retrieval. *Psychological review*, 85(2):59, 1978

Note that this model makes a lot of predictions. If simulated for a sufficient number of times, the model predicts not only the probabilities that each option will be chosen, but also the *distribution* of the response times associated with them. This abundance of behavioral predictions has made DDMs a favorite in the field of cognitive psychology, where they have become one of the dominant frameworks. Figure 19 illustrates an example such model, with parameters $A = 1$, $v = 0.54$, and $Ter = 1$ (one second).

What Are The Two "Options"?

So far, I have been intentionally cagey about what we intend as the model's options.

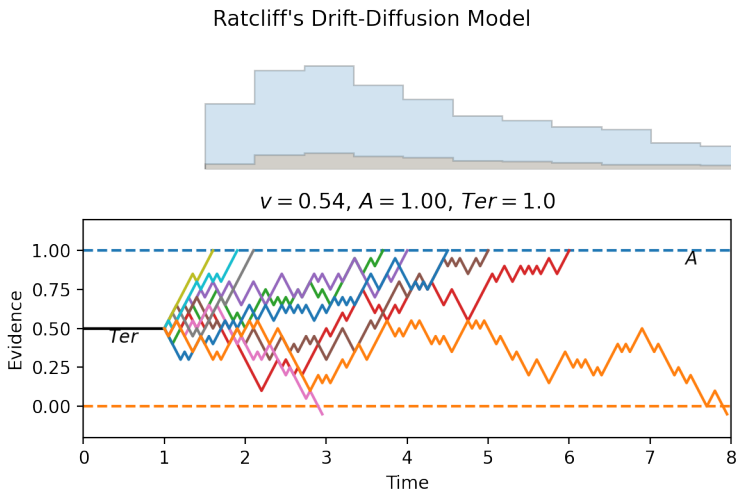


Figure 19: Ratcliff's Drift Diffusion Model

Consider, for example, what has been perhaps the most experimental paradigm to which this model has been applied, the motion coherence detection. In this type of paradigm, participants are presented with an array of moving dots. All of dots are moving and at the same speed, but the degree to which they are moving *in the same direction* changes across conditions. Figure 20 illustrates three hypothetical cases in which 0%, 50%, or 100% of the dots are moving to the right. The direction of the coherent motion is not given to participants beforehand; on some trials, the coherent motion might be towards the left and, in other trials, towards the right. Participants are typically instructed to indicate the direction of coherent motion with the left or the right hand. In cases in which this paradigm was carried out in non-human primates, the animals typically respond by performing a saccade, i.e., by moving the eyes in the intended location.

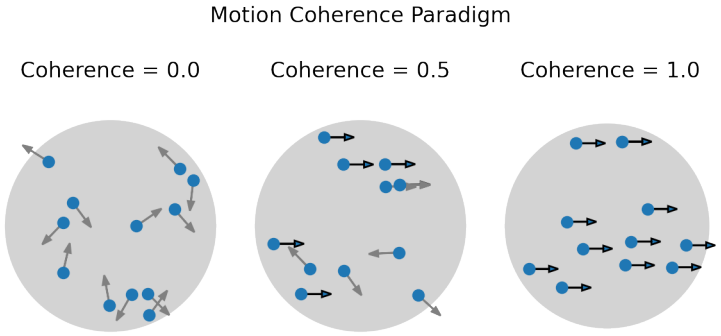


Figure 20: Example stimuli from a motion coherence paradigm

In this paradigm, the degree of motion coherence clearly maps to the drift rate v. But what are the two responses? Intuitively, the answer seems straightforward: the two responses are left and right. But there are other ways to

think about this. If an experimenter is interested in studying hand dominance, for example, they might decide to code the response that corresponds to the dominant hand of each participant as A, and the response that corresponds to the non-dominant hand as 0. So, A could be either left or right, depending on the self-attested preferences of the individual. In what is perhaps the most common case with DDMs, an experimenter might be interested in the difference between correct and incorrect answers, and would code all the correct answers as A, and all of the incorrect answers as 0, independently of whether they were right or wrong.

Note that the interpretation of the other parameters also changes as an effect of this decision. If the responses were coded as left vs. right, a response bias parameter z could be easily interpreted as an effect of hand dominance, and a drift rate v could be interpreted as a greater ease to distinguish movement in the dominant hemifield. But, if the two responses were coded as correct vs. incorrect, the interpretation would be different, and v would likely be considered as a measure of the subjective ease of the decision and z as some facilitatory effect—for example, the effect of a visual cue that warns of the direction of motion [35].

[35] Martijn J Mulder, Eric-Jan Wagenmakers, Roger Ratcliff, Wouter Boekel, and Birte U Forstmann. Bias in the brain: a diffusion model analysis of prior probability and potential payoff. *Journal of Neuroscience*, 32(7):2335–2343, 2012

The Speed-Accuracy Trade-Off

The three parameters of the model, v, A, and Ter, reflect different characteristics. The drift rate v is supposed to reflect the characteristics of the stimuli that we are trying to capture.

One of the noteworthy characteristics of this model is that it provides an elegant, built-in explanation for the *speed-accuracy trade-off*. This is a well-know effect in psychology whereby an individual's accuracy in a decision-making task[36] is inversely proportional to their speed. In other words, one could either be fast and imprecise, or slow and precise.

[36] And, probably, in every task

We have seen an example of this tradeoff in the first notebook: it is explicitely embedded in Fitts' model of response times. In that model, however, the speed-accuracy trade-off was directly encoded in the equations. In Ratcliff's DDM model, instead, the speed-accuracy tradeoff comes into being as a side effect of adjusting the decision boundary A. Given a certain drift rate v (which is a property of a stimulus), a decision maker can adjust the decision boundary and either choose to accumulate a lot of evidence before making a decision (being more precise but slower) or choose to lower the threshold so that a decision can be made earlier, but based on less evidence. In other words, the speed-accuracy tradeoff exists because evidence accumulation is noisy and outside the control of the decision-maker, and the only way to be more precise is to accumulate more evidence, which necessarily takes time.

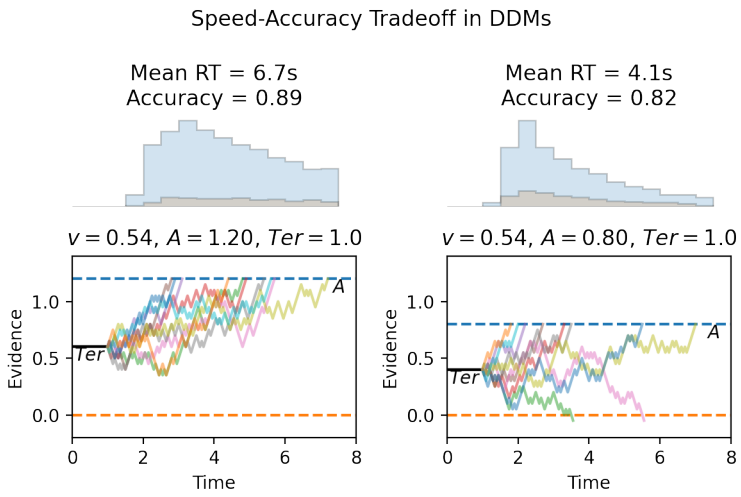


Figure 21: Speed accuracy tradeoff in DDMs. *Left:* A model with high decision threshold will be most accurate but take a longer time to decide. *Right:* By lowering the decision threshold, responses can be made more quickly, but the number of errors is going to increase

The Response Bias z

One of the assumptions of the DDM model, as described here, is that the two options are equivalent except for the relative evidence in favor of one of them—a factor captured by the parameter v. If there is no evidence in favor of either option, the model predicts that both options will be chosen equally frequently and with similar distributions of response times.

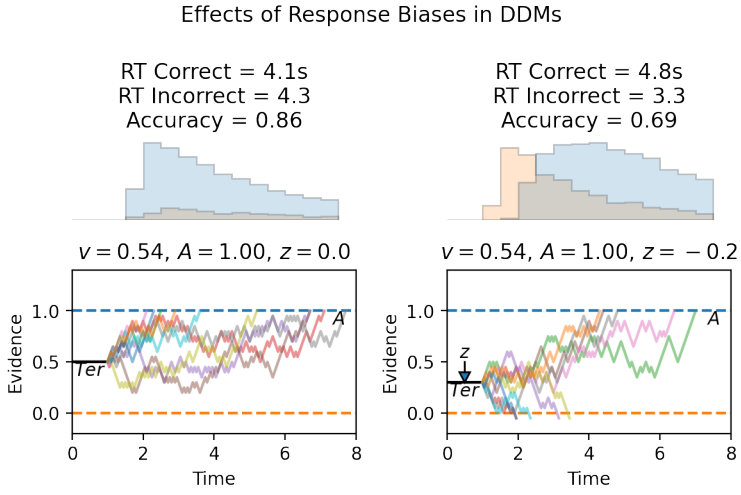


Figure 22: Compared to a canonical DDM with neural starting point (*Left*), a DDM with an initial response bias z is more likely and faster at reaching the boundary that is closer to the starting point *Right*

In a study [37], it was found that response biases were affected by prior knowledge of the task.

[37] Martijn J Mulder, Eric-Jan Wagenmakers, Roger Ratcliff, Wouter Boekel, and Birte U Forstmann. Bias in the brain: a diffusion model analysis of prior probability and potential payoff. *Journal of Neuroscience*, 32(7):2335–2343, 2012

Modeling Errors

We have seen above that Ratcliff's model is commonly applied by coding the two options as the correct and incorrect response to a given stimulus,

independently of the response. In this case, the drift parameter can be easily understood as a measure of decision difficulty: the easier it is to make a decision, the quicker the response times and the more likely the correct response is to be selected.

One noteworthy limitation of this approach, however, is that errors, in human experiments (and likely in primate experiments as well) exhibit different distributions of response times. The nature of this difference depends a bit on the task. When the experiment stresses quick response times.

These apparently complicated pattern can be captured by adding another source of inter-trial variability to the model—specifically, by making the drift rate vary randomly across trials. In this case, the drift rate is not constant across trials, but is drawn from a normal distribution with mean μ_v and standard deviation σ_v: $v \sim \mathcal{N}(\mu_v, \sigma_v)$

Relationship between Accumulator Models and RL

It can be shown mathematically that, in Ratcliff's model, and that the probability $P(A)$ of crossing the boundary A can be computed as:

$$P(A) = \frac{1}{1 + e^{-2v \times A}} \tag{16}$$

This provides us with an interesting connection to RL. As noticed above, one of the most common policies in RL is the Boltzmann policy, whereby a particular action a is chosen over all the other competitors using the following equation:

$$P(a_i) = \frac{e^{Q(s,a_i)/\tau}}{\sum_j e^{Q(s,a_j)/\tau}}$$

where τ is the decision temperature parameter. In the case of two options, a_1 and a_2, this expression reduces to:

$$\begin{aligned} P(a_1) &= \frac{e^{Q(a_1)/\tau}}{e^{Q(a_1)/\tau} + e^{Q(a_2)/\tau}} \\ &= \frac{e^{Q(a_1)/\tau}}{e^{Q(a_1)/\tau} \times \left(1 + \left[e^{Q(a_2)/\tau} / e^{Q(a_1)/\tau}\right]\right)} \\ &= \frac{1}{1 + e^{Q(a_2)/\tau} / e^{Q(a_1)/\tau}} \\ &= \frac{1}{1 + e^{Q(a_2)/\tau - Q(a_1)/\tau}} \\ &= \frac{1}{1 + e^{-\Delta Q/\tau}} \end{aligned} \tag{17}$$

where $\Delta Q = Q(a_1) - Q(a_2)$ is the difference in the Q values of the two actions.

The two equations 16 and 17 are clearly similar, and become the same if we assume that $v = \Delta Q/2$ and $A = 1/\tau$. These identifications are reasonable. In DDM, the drift rate v supposedly reflects the ease of the discrimination between the two options; in the case of two actions, larger differences in perceived values should also facilitate the decision in favor of the better options. Similarly, and as noted above, as the boundary A increase, the decision process becomes increasingly more likely to converge on the correct response, exactly as the Boltzmann policy becomes more likely to converge on the best option when as the temperature τ drops.

In fact, models like DDM are often used as a back-end to generate realistic response times for RL agents or other types of models that compute values for decision: These values are then passed to an accumulator model that generates the predicted number of choices and the corresponding response time distributions (e.g., [38]).

[38] Sebastian Musslick, Amitai Shenhav, Matthew M Botvinick, and Jonathan D Cohen. A computational model of control allocation based on the expected value of control. In *The 2nd multidisciplinary conference on reinforcement learning and decision making*, 2015

Race Models

So far, we have seen models in which a binary choice needs to be made and evidence accumulates in favor or against that choice. But what happens if we have more than two choices? As noticed above, classic accumulator models do not work beyond 2AFC tasks.

One simple solution would be to divide the options across multiple accumulator models. This can be done easily through the use of *race models*. Race models are a category of accumulator models that have only one possible option. In the case of multiple alternatives, a different model is created for each possible options. Unlike DDMs, in which evidence for one alternative also counts against the other alternative, in race models evidence is accumulated independently for each option. A response as soon as the evidence cross the decision threshold in one accumulator. This is the reason these models are called "race" models: All accumulators are competing with each other by independently gathering evidence, and the first one to cross the threshold wins the race.

The Linear Ballistic Accumulator Model

The best-known example of race model is the Linear Ballistic Accumulator model [39], or LBA. The functioning of this model is illustrated in Figure 23. Unlike DDM, in LBA the evidence accrus at a fixed rate throughout the trials, hence the name "ballistic". Thus, once the trial starts, the evidence grows linearly moves along the the same path. This is why these type of modes are sometimes referred to as a "random ray" model [40], as opposed to the "random walk" trial dynamics of DDM (Figure 19). Because of its nature, it is possible to calculate exactly the time t at which evidence will cross the decision boundary for each trial using trigonometry: $t = Ter + A/\tan(v)$

[39] Scott D Brown and Andrew Heathcote. The simplest complete model of choice response time: Linear ballistic accumulation. *Cognitive psychology*, 57(3):153–178, 2008

[40] Adam Reeves, Nayantara Santhi, and Stefano DeCaro. A random-ray model for visual search and object recognition. *Spatial Vision*, 18:73–83, 2005

The only source of variability in the LBA is the variability in the drift parameter v. To create inter-trial variability, the drift rate is sampled from a Gaussian distribution, as in DDM: $v \sim \mathcal{N}(\mu_v, \sigma_v)$. Notice that, while in DDM this is an additional mechanism that is added to the natural variability of trials, in LBA this is a necessary source of variability because each trial is ballistic; without variability in v, all trials will be the same.

Another source of variability in LBA the starting point k, which is sampled from a uniform distribution between 0 and a maximum value $m \leq A$, i.e. $k \sim \mathcal{U}(0, m)$.

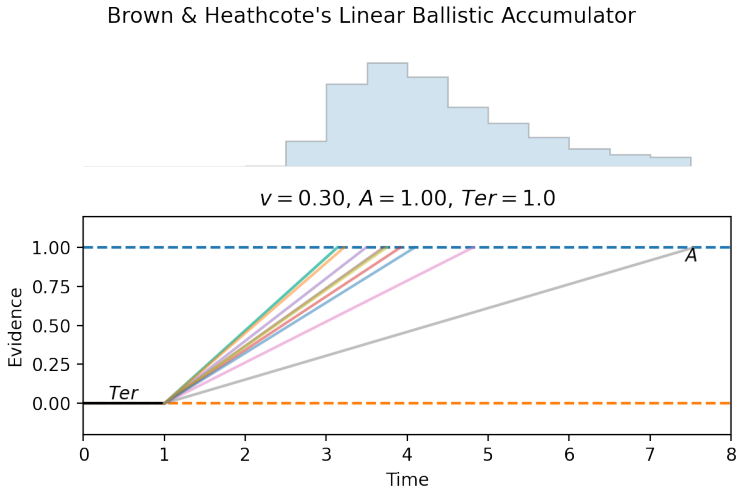


Figure 23: Brown and Heathcote's Linear Ballistic Accumulator model

To simulate a decision between options,

Models of Long-Term Memory

A Bayesian approach to memory

Long-term memory is the ability to retain information over long-periods of time. Neuroscientists distinguish between *procedural* memory, or memory for habits, and memory, or memory for facts and events [41]. Model-free reinforcement learning, for example, is an excellent framework to understand procedural memory. In this chapter, instead, we will examine explanatory models of declarative memory.

[41] Larry R Squire. Memory systems of the brain: a brief history and current perspective. *Neurobiology of learning and memory*, 82(3):171–177, 2004

Much like Reinforcement Learning can be seen as an adaptive mechanisms to optimally solve the problem of maximizing future rewards, so many declarative memory frameworks see memory as an adaptive mechanism to solve to problem of optimally retrieving information, given the statistics of the environment. In fact, this problem has been likened to other information-retrieval problems, such as the problem of predicting book rentals in a public library [42]. This is, in turn, a statistical inference problem, and can be formalized using a Bayesian approach.

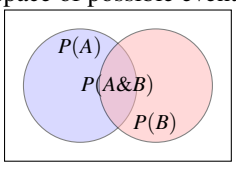
[42] John R Anderson and Robert Milson. Human memory: An adaptive perspective. *Psychological Review*, 96(4):703, 1989

Bayes' Theorem

Bayesian approaches are based on Bayes' theorem, a mathematical tool to calculate probabilities of events given the context in which they appear.

The bases of Bayes' theorem are easy to explain. Consider two events, A and B. As it is usual in probability theory, they can be represented as areas in a universe of possibile events. In Figure 24, the red area represents all the cases in which A happen, while the blue area represents all the cases in which B happens. The purple area in-between is the probability that both A and B occur at the same time.

Figure 24: Visual illustration of the probabilities of two events, A and B, in the space of possible events



The *probabilities* of A and B, denoted as $P(A)$ and $P(B)$, are the proportion of the white rectangle that is covered by the red and blue areas respectively. The purple overlap area is the area in which both A and B occur is the probability of their joint occurrence, that is, $P(A\&B)$.

The key to understanding Bayes theorem is that all of these values can be expressed in relation to each other. To this end, Bayes introduced a special notation that denotes *conditional* probabilities. For example, $P(A|B)$, that is,

the probability of of A given B, is the proportion of the purple area (where both A and B occur) that is part of the blue area (where B occurs, with or without A). The same area can also be expressed as $P(B|A)$, that is, the area in which B within the area in which A occurs. Thus, we can write:

$$\begin{aligned} P(A\&B) &= P(A|B) \times P(B) \\ P(A\&B) &= P(B|A) \times P(A) \end{aligned}$$

It follows that $P(A|B) \times P(B) = P(B|A) \times P(A)$, and therefore:

$$P(A|B) = \frac{P(A) \times P(B|A)}{P(B} \tag{18}$$

These quantities are given special names in reference to event A. The term $P(A|B)$ is called the *posterior* probability of A; the expression "posterior" indicates that this is what we know about the probability of A *after* we have learned that B is true. Similarly, the quantity $P(A)$ is called the *prior* probability of A; this name indicates that this is the what we know about A *before* having any other information. The quantity $P(B|A)$ is called the likelihood of A. Finally, the term $P(B)$ is called the *marginal* probability; it is the probability of the other events that are not A.

What Bayes theorem states is that, once we know B, we can get an a better and more accurate idea of how likely A is to happen by combining our prior knowledge of A with our knowledge of B and our knowledge of how A and B are related.

The Rational Analysis Framework

Bayes theorem become the bases for the so-called *rational analysis* approach to memory [25]. In Bayesian approaches to cognition, an agent is considered as an "ideal observer" with limitless access to information and no cost in retrieving. This is, obviously, a grand simplification, but it offers some advantages. Under these circumstances, it is becomes possible to formalize memory as a problem about the statistics of the environment, rather than a problem of the mechanics of the agents.

But what is the "problem" that memory has evolved to solve? It is assumed that long-term has evolved to make information available at a later time if it were needed. For instance, a squirrel would need to remember the place where it buried a nut as well as the location where it was almost killed by a hawk. These two pieces of information might be needed in different context: it is important to remember where the nut is buried when the squirrel is hungry, and it is important to remember where the hawk hunts when looking for a place to hide a new nut. The availability of a memory, thefore, should adaptively reflect its probability of being needed in a certain context. Thus, if we indicate the specific memory as m and the current context as Q (composed of different elemental cues $q_1, q_2, \ldots q_n$) a memory retention function

reflects the logarithm of the posterior odds $P(m|Q)/P(\neg m|Q)$, which can be expressed, per Bayes' theorem, as the product of prior odds $P(m)/P(\neg m)$ and likelihood $P(Q|m)/P(Q|\neg m)$. Assuming, for simplicity, that each cue q is independent from each other, we can simplify this expression as

$$\begin{aligned}\frac{P(m|Q)}{P(\neg m|Q)} &= \frac{P(m)}{P(\neg m)}\frac{P(Q|m)}{P(Q|\neg m)} \\ &= \frac{P(m)}{P(\neg m)}) \times \prod_q \frac{P(q|m)}{P(q|\neg m)} \\ &= \frac{P(m)}{P(\neg m)}) \times \prod_q \frac{P(q|m)}{P(q)} \end{aligned} \tag{19}$$

The last step in Eq. 19 is an approximation derived from the consideration that, for large numbers of memories, $P(q|\neg m) \approx P(q)$.

The terms in Eq. 19 have a straightforward explanation in terms of the cognitive psychology of memory [26, 25, 27, 28]. Specifically, the log prior odds capturing the effects of the previous history of usage of m and the log-likelihood corresponding to the effects of contextual cues in memory retrieval.

A Formal Model of Prior Probabilities

Many models have tried to capture the mechanism by which prior probabilities change over time.

What is the shape of forgetting?

In general, it is intuitive that time is a part of the prior probability of retrieving a memory. The more time passes without retrieving a particular fact, the more we can assume that it is not gonna be needed in the future. As an example, consider the results from perhaps the first systematic test of human memory: Ebbinghaus' own test data, first described in 1885 [43] (Figure 25).

Ebbinghaus conducted a series of experiments creating random lists of non-sensical syllables. Instead of measuring probabilities of recall, he calculated how many times it took him to memorize the list well enough to repeat it perfectly. He then restudied the list after letting different intervals of time pass by, and recorded how many times he had to restudy it to recite it perfectly again. The effect of forgetting was measured by calculating the the percent difference between the first and the second number of attempts, which he termed *percent savings*. Perfect recall, in which a person does not need to restudy the list, corresponds to 100% savings, while having to restudy the list the same amount of times (or more) corresponds to 0% savings.

Figure 25 plots Ebbinghaus' percent savings across different intervals, from 15 minutes to one full month. The effect of time on memory is visible as a decay effect.

[43] Hermann Ebbinghaus. *Über das gedächtnis: untersuchungen zur experimentellen psychologie*. Duncker & Humblot, 1885

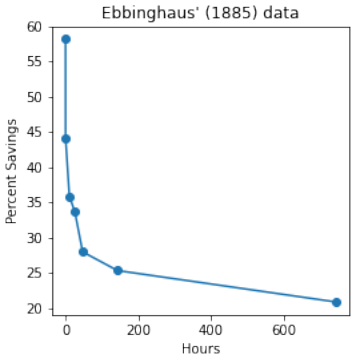


Figure 25: Ebbinghaus' classic results

But how exactly does time affect memory? One possibility is that the effect of time is exponential. Exponential function have an expression of the form $P(m) = \alpha^{\beta t}$ with time t being the exponent (hence the name). Another possibility is that the effect of time is that of a power function. Power functions would have the form $P(m) = \alpha \times t^{\beta}$, where time t is a the base and the exponent is a fixed parameter β.

This question might seem idle, but is not. Exponential and power curves look almost identical to the naked eye, but they have (very) different mathematical properties. And these mathematical properties are important if we want to build a model. For example, an exponential function has a fixed *half-time*, i.e. a fixed interval of time during which the quantity that we measure reduces by half. This is a familiar property of radioactive materials: if a material has a half time of (let's say), one minute, it means that, after minute, 1 Kg of the material will have become ½Kg, and, one minute later, the ½Kg will have turned into ¼Kg. The pace of decay is fixed, and the half-time interval is meaningful measure. But power functions do not have these property; they have other interesting properties (for example, they are scale-invariant) but their pace is ever changing.

The simplest way to determine whether a particular function is an exponential or a power function is to plot them on modified axes. If both the x and y axes are in log-scale (a so-called *log-log plot*), the power function would look like a straight line, but the exponential function would look like a curve[44]. For example, we can take the original data from Ebbinghaus and then plot them on a log-log plot. The result, shown in Figure 26 , is an almost perfect straight line.

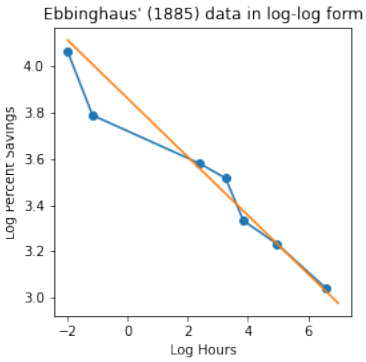


Figure 26: Ebbginghaus's results from his own memory experiments

[44] The exponential would look like a straight line if only the x-axis in log-scale.

Rationality of Memory

One of the key ideas of the Bayesian approach is that memory is adaptive. If it is, its laws should reflect the dynamics of the environment we live in. This assumption was empirically tested by Anderson and Schooler [45]. The authors tested three different sources of information: words occurring in the headlines of the New York Times; Words pronounced by parents to toddlers ("child-directed speech", available through the CHILDES database); and, finally, email senders in Anderson's own email inbox. All of these three sources of information were monitored for 100 consecutive days, and the statistics for each word collected. In each case, the the relationship between time passed and the probability of the information showing up again followed a power function. Not only all three sources of information follow the same law: Their very own parameters are similar to each other!

[45] John R Anderson and Lael J Schooler. Reflections of the environment in memory. *Psychological science*, 2(6):396–408, 1991

The ACT-R Model

The rational analysis framework, together with the power law of forgetting, forms the bases of what is, perhaps, the most influential explanatory model of human memory, ACT-R [46]. Many other important models (e.g., REM: [47] or MINERVA [48]) share almost identical assumptions, so we can focus on ACT-R without losing generality. Note that, although ACT-R has grown to be a much more complete theory that spans more than memory, here we will focus only on its long-term memory component.

Initially, we will not make any assumption about how memories are internally represented. Instead, we will assume that our brain registers large amounts of information into "snapshots" and that these snapshots, which we will refer to as a *memories*, can be accessed later on.

Each time a snapshot is taken, a new *trace* is made. Each trace decays with time. So, if we indicate as t_i the time at which the the i-th trace was created, then we indicate that the odds of retrieving it decay with a power function:

$$\frac{P(i)}{P(\neg i)} = (t - t_i)^{-d} \tag{20}$$

[46] John R Anderson and Gordon H Bower. *Human Associative Memory*. Psychology Press, 2014; John R Anderson. Retrieval of information from long-term memory. *Science*, 220(4592):25–30, 1983; John R Anderson, Daniel Bothell, Michael D Byrne, Scott Douglass, Christian Lebiere, and Yulin Qin. An integrated theory of the mind. *Psychological review*, 111(4):1036, 2004; and John R Anderson. *How can the human mind occur in the physical universe?* Oxford University Press, 2009

[47] Richard M Shiffrin and Mark Steyvers. A model for recognition memory: Rem—retrieving effectively from memory. *Psychonomic bulletin & review*, 4(2):145–166, 1997

[48] Douglas L Hintzman. Minerva 2: A simulation model of human memory. *Behavior Research Methods, Instruments, & Computers*, 16(2):96–101, 1984

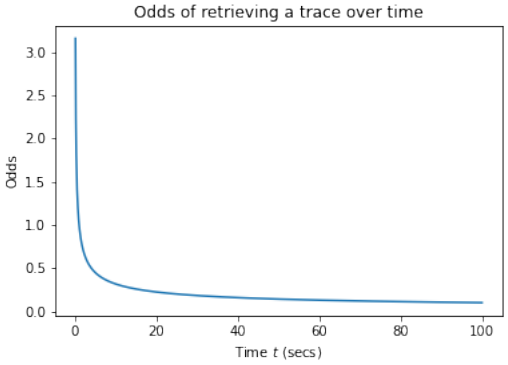


Figure 27: The odds of retrieving a particular memory trace, created at time $t_i = 0$, decay over time according to a power function

where d is the *decay rate* of memory, i.e., the speed at which memories are forgotten. Figure 27 illustrates the retention curve of a trace created at time $t_i = 0$ with a decay rate of $d = 0.5$.

In the ACT-R model, it is assumed that the contribution of all traces is a summed, that is, each trace contributes linearly to the odds of retrieving the memory it belongs to. Thus, the odds of retrieving a memory m is the sum of the odds of retrieving any of its associated traces.

$$\begin{aligned} \frac{P(m)}{P(\neg m)} &= \sum_i \frac{P(i)}{P(\neg i)} \\ &= \sum_i (t - t_i)^{-d} \end{aligned}$$

Finally, it is common to express memory in terms of *activation*, a quantity that is defined as the *log odds* of retrieving a memory. Formally, the activation $A(m)$ of a memory m is given by Equation 21.

$$A(m) = \log \frac{P(m)}{P(\neg m)} \tag{21}$$

One might wonder why it would be necessary to add yet another measure of memory, after probabilities and odds. The main reason is that activation is a *convenient* measure. Unlike probabilities or odds, activation values span the entire domain of real numbers, from $-\infty$ to $+\infty$. The mid-point at which a memory is equally likely to be retrieved or forgotten, which corresponds to a probability value of $P(m) = 0.5$ and an odds value of $P(m)/P(\neg m) = 1$, becomes an activation value of $A(m) = 0$. In this sense, activation values can be thought of as forming a scale with a meaningful zero value. A memory with a positive activation value is more likely than not to be remembered, and a memory with a negative activation value is more likely than not to be forgotten. For this reason, it is convenient to refer to the value of $A(m) = 0$ as to the *forgetting threshold*.

The relationship between the odds of retrieving individual traces, the activation of a memory, and the forgetting threshold is shown in Figure 28

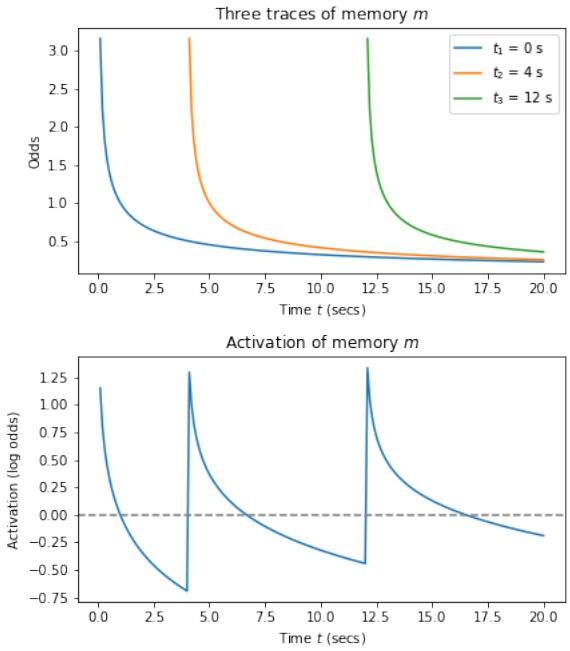


Figure 28: (Top) The declining retrieval odds of three traces associated with the same memory m and created at different times; (Bottom) The activation of m of reflects the summed effects of their traces and their decline over time.

Combining Equations 20-21, we can formally define activation of a memory at time t as:

$$A(m,t) = \log \sum_i (t - t_i)^{-d} \tag{22}$$

Frequency, Recency, and the Spacing Effect

Any good memory model should be able to correctly explain the fundamental effects reported in the literature. In addition to the power law of forgetting [49], other important memory effects include the recency, frequency, and spacing effect.

Equation 22 captures the basic effects of recency and frequency. Recency arises as a consequence of the power law of forgetting, which makes the activation of a memory decline as a power function of its age (Figure 29A). Frequency, on the other hand, depends on the summed effect of the accumulation of traces, by which a makes a memory with more associated traces retains greater activation than a memory with the same age but fewer associated traces (Figure 29B).

[49] Allen Newell and P Rosenbloom. Mechanisms of skill acquisition. In John Robert Anderson, editor, *Cognitive skills and their acquisition*, chapter 1, pages 1–56. Lawrence Erlbaum Associates, 1981

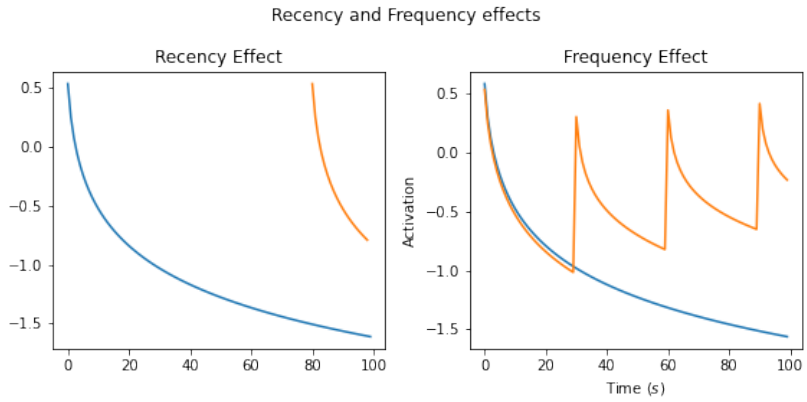


Figure 29: Effects of recency and frequency

In addition to recency and frequency, another fundamental law of memory is the *spacing* effect, that is, the phenomenon by which the probability of retrieving a memory is higher when the interval between the encodings of its traces (the "spacing") is larger. The spacing effect is typically studied in experiments in which a particular item is presented twice, with different intervals between the two presentations; each presentation is assumed to result in an independent trace. The time between the second presentation and the final test is maintained constant, and the interval between the two traces is varied.

It is easy to see that Equation 22 cannot account for the spacing effect, as the combined effects of each trace are simply additive. If anything, a larger gap implies that the first trace was created *earlier* than in the case of a shorter gap. Thus, in the case of a larger gap, the activity of the first trace would have decayed more, resulting in lesser activation for the memory—exactly the opposite of what is experimentally found.

To account for the spacing effect, Pavlik and Anderson [50] introduced a modification to the decay term d. Specifically, they relaxed the constrain

[50] Philip I Pavlik Jr and John R Anderson. Practice and forgetting effects on vocabulary memory: An activation-based model of the spacing effect. *Cognitive science*, 29(4):559–586, 2005

forcing d to be constant across all traces. Instead, they allowed every trace to have its own specific decay term d_i:

$$A(m,t) = \sum_i (t - t_j)^{d_i} s \tag{23}$$

The trace-specific term d_i depends on the current value of the activation $A(m,t = t_i)$ at the moment in which the trace was created. Thus, when the i-th trace is created, it is given a decay rate d_i calculated as follows:

$$d_i = c\ \mathrm{e}^{A(m,t=t_i)} + \alpha \tag{24}$$

where $A(m,t = t_i)$ represents the activation of m at time t_i. The spacing effect is made possible by including the term $c\ \mathrm{e}^{A(m,t=t_i)}$ in the computation of the decay rate. When two traces are temporally close together, the corresponding memory's activation at the moment the second trace is encoded is higher, resulting in a larger value of $c\ \mathrm{e}^{B(m,t=t_i)}$ and, therefore, a larger decay rate for the second trace.

Note that, even when allowing for different traces to decay at different rates, decay is still determined by a single parameter, α.

The final, complete model is noteworthy for its reliability, having been used to successfully model a variety of memory results [37, 36] and having been used to successfully derive optimal schedules for learning practice [38]. The rate of forgetting α in Eq. 24 has been also used as an idiographic (i.e., person-specific) parameter [51], with α remaining a stable and reliable trait within the same individual across sessions and materials, and to assess individual differences in real-life outcomes, such as a student's success at answering test questions after studying [39].

[51] Florian Sense, Friederike Behrens, Rob R Meijer, and Hedderik van Rijn. An individual's rate of forgetting is stable over time but differs across materials. *Topics in cognitive science*, 8(1):305–321, 2016

Posterior Probabilities and the Role of Context

So, far, the model has included only equations that capture the prior history of a memory, that is, the Bayesian prior odds $P(m)/P(\neg m)$. But what about the likelihood odds $P(Q|m)/P(Q|\neg m)$ in Equation 19?

Because we have expressed the availability of memory in terms of activation, we first need to start by putting Equation 19 in log form:

$$\begin{aligned}
\frac{P(m|Q)}{P(\neg m|Q)} &= \frac{P(m)}{P(\neg m)}) \times \prod_q \frac{P(q|m)}{P(q)} \\
\log \frac{P(m|Q)}{P(\neg m|Q)} &= \log \left(\frac{P(m)}{P(\neg m)}) \times \prod_q \frac{P(q|m)}{P(q)} \right) \\
&= \log \frac{P(m)}{P(\neg m)}) + \log \prod_q \frac{P(q|m)}{P(q)} \\
&= \log \frac{P(m)}{P(\neg m)}) + \sum_q \log \frac{P(q|m)}{P(q)}
\end{aligned} \tag{25}$$

Equation 25 makes it clear how the concept of *activation* relates to Bayesian equation: Equation 22 is really an analytic expression of the first part of the right-hand side of this equation, i.e. $\log[(P(m)/P(\neg m)]$. This quantity is technically referred to as *base level* activation, as it refers to the changes in a memory's activation that are only dependt on its own history and the passing of time.

The effects of *context* on a memory's availability, on ther other hand, are captured by the second term, $\sum_q \log \frac{P(q|m)}{P(q)}$. According to this formula, The contextual effects are the sum of the contributions of each cue q in the environment that is associated with m—that is i.e., has a non-zero value of $P(m|q)$. This is both intuitive and appealing: Exactly like each additional trace of m adds to m's activation, so each additional environmental cue that is predictive of m (that is $P(m|q) > 0$) adds to its activation.

But how does this summation occur? And how can we keep track of $P(m|q)$? To these questions, it is necessary to explain how ACT-R internally represents memories.

Memory Representation

In ACT-R, memories are internally represented as records of features. Historically, these records are referred to as "chunks" and their features as "slots", although this chapter will use the more transparent and less technical terms "memories" and "features". Features represent the atomic elements of a memory, such fundamental sensory information (e.g., the color yellow) and basic concepts (e.g., the magnitude of a number). Each feature is identified by a name and a value. For example, the property of having the color "yellow" is represented as the pair (Color: Yellow), with the first element being the feature name and the second the feature value. Note that the name of a feature exists only to make ACT-R programs easy to read and write, and has no implication for how memories are implemented in the brain. With this in mind, the semantic fact that "A canary is a yellow bird" can be represented as a record of features such as [(Object: Canary), (Type : Bird), (Color: Yellow)]. A single memory can be made up of an arbitrary number of features, with the only constraint that two features cannot share the same name. Long-term memory is a finite collection of such memories.

Although this representational format might seem too unconstrained and symbolic, it is equivalent to the vector representations used in other formal models [52] or in neural network models [53] of long-term memory. In these models, a single memory is represented by a vector of fixed size; feature names are represented by a subset of element positions in a vector; and feature values by specific numeric values of the corresponding elements. For example, in Rogers' model of semantic memory, the property (Color: Yellow) is represented in the 64 "perceptual" artificial neurons located in positions 41–104 of the network's input layer. By contrast, the property (Type : Bird) is

[52] Richard M Shiffrin and Mark Steyvers. A model for recognition memory: Rem—retrieving effectively from memory. *Psychonomic bulletin & review*, 4(2):145–166, 1997; and Douglas L Hintzman. Minerva 2: A simulation model of human memory. *Behavior Research Methods, Instruments, & Computers*, 16(2):96–101, 1984

[53] Timothy T Rogers, Matthew A Lambon Ralph, Peter Garrard, Sasha Bozeat, James L McClelland, John R Hodges, and Karalyn Patterson. Structure and deterioration of semantic memory: a neuropsychological and computational investigation. *Psychological review*, 111(1):205, 2004

represented by values of the 16 neurons in position 137–152. Thus, any ACT-R memory can be transformed into a corresponding vector representation if the list of possible features is predefined and the features that are not present in a memory are set to a default value of zero. This translation scheme, in fact, was used to create a functional neural network implementation of ACT-R [54].

[54] Christian Lebiere and John R Anderson. A connectionist implementation of the act-r production system. In *Proceedings of the fifteenth annual conference of the Cognitive Science Society*, pages 635–640, 1993

Contextual Effects and Spreading Activation

The contextual activation component of Equation 21 can best be understood by considering a classic representation format for memories, namely, semantic networks [55]. In semantic networks, each memory represents a node, and associated memories are connected by directional links. The strength of the link between q and m reflects the statistics of co-occurrence between the two events. The terminal leafs of this network represent basic, atomic representations, such as the sensory information corresponding to the "Yellow" or the abstract concept of "Two". Figure 30 provides a visual representation of how the concept of "A canary is a yellow bird" is represented in such a network and how its representation partially overlaps with the concept of "A canary is a bird that sings", as well as with the concepts of "The Sun is yellow star" and "Taylor Swift is an artist who sings".

[55] John R Anderson. *The architecture of cognition*. Lawrence Erlbaum Associates, 1983; and Allan M Collins and Elizabeth F Loftus. A spreading-activation theory of semantic processing. *Psychological review*, 82(6):407, 1975

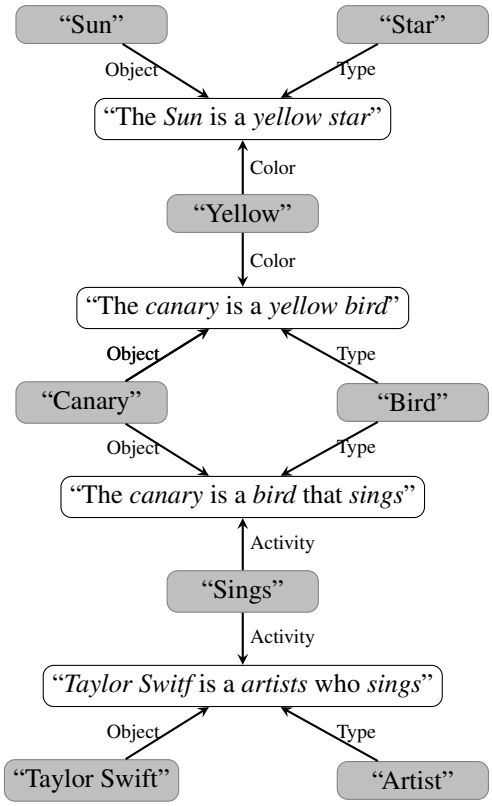


Figure 30: Semantic network representation of the four ACT-R memories "The canary is a yellow bird", "The canary is a bird that sings", "Taylor Swift is an artists who sings", and "The Sun is a yellow star". Grey boxes represent basic concepts (that is, terminal nodes), and white boxes represent the facts built upon them

Contextual effects can be understood as an additional amount of activation that *flows* from the memory nodes that are part of the context $Q = q_1, q_2 \ldots q_N$. This activation boost is known as *spreading* activation [56]. Greater co-occurrence of q when m is present (i.e., $P(q|m)$) corresponds to stronger links and thus results in greater activation.

[56] Allan M Collins and Elizabeth F Loftus. A spreading-activation theory of semantic processing. *Psychological review*, 82(6):407, 1975

But where does spreading activation come from? If base-level activation reflects the creation of new traces and their fading due to passive forgetting, spreading activation can be interpreted, in psychological terms, as the amount of *attention* paid to the current context during retrieval. For example, the fact that the canary sings would become more active when someone is paying attention to the fact that Taylor Swift also sings (because of the spreading activation from "singing") and will become even more active when someone is paying attention to the fact that the canary is yellow (because of the spreading activation from "Canary" *and* "Bird").

This form of attentional control can also be interpreted in terms of *working memory*, that is, an individual capacity to maintain, process, and update short-term information [57]. Specifically, controlled activation of long-term memory elements through attention can explain the relationship between performance in complex span tasks and the ability to control interference [58]. In fact, Daily and colleagues [59] were able to show that individual variations in W values capture idiographic differences in working memory performances and that individual differences in W values, when estimated independently through a working memory task, successfully predicted performance on other tasks that demand cognitive control.

[57] Alan Baddeley. Working memory. *Science*, 255(5044):556–559, 1992; Alan D Baddeley and Robert H Logie. Working memory: The multiple-component model. 1999; and Alan Baddeley. Working memory. *Current biology*, 20(4):R136–R140, 2010

[58] Michael J Kane, M Kathryn Bleckley, Andrew RA Conway, and Randall W Engle. A controlled-attention view of working-memory capacity. *Journal of experimental psychology: General*, 130(2):169, 2001; and Gregory C Burgess, Jeremy R Gray, Andrew RA Conway, and Todd S Braver. Neural mechanisms of interference control underlie the relationship between fluid intelligence and working memory span. *Journal of experimental psychology: general*, 140(4):674, 2011

[59] Larry Z Daily, Marsha C Lovett, and Lynne M Reder. Modeling individual differences in working memory performance: A source activation account. *Cognitive Science*, 25(3):315–353, 2001

Formally, if there is a direct link from q to memory m, then m receives an activation boost that is proportional to the product between the strength of the link connecting q to m (indicated as $S_{q \to m}$) and the attentional weight given that cue. The attentional weight is usually simplified as a single scalar quantity, W, that is assigned to all of features that are present in the context Q.

The total amount of spreading activation $S(m)$ that m receives is the sum of all of the partial effects of each element q:

$$S(m) = W \times \sum_{q \in Q} S_{q \to m} \tag{26}$$

Equation 26 equation can be related to the contextual term in in Equation 21 by assuming that each association $s_q \to m$ approximates the quantity $\log P(q|m)/P(q)$. Under these conditions, the quantity $P(q)$ can be measured by simply counting the number n of memories that contain q as a feature. Imagine, for simplicity, we have only one contextual cue we are paying attention to. In this case, the amount of spreading activation from q becomes.

$$\begin{aligned} S(m) &= W \times S_{q \to m} \\ &= W \times \left(\log \frac{P(q|m)}{P(q)} \right) \\ &= W \times (\log P(q|m) - \log P(q)) \\ &= W \log P(q|m) - W \log n \end{aligned} \tag{27}$$

The key insight from Equation 27 is that, as n grows, the spread of contextual activation *diminishes*: the more common is a cue, the less it can contribute to retrieve a memory. This assumption plays an important role in explaining some paradoxical phenomena of memory retrieval, most notably the *fan effect* [60].

[60] John Robert Anderson. Retrieval of propositional information from long-term memory. *Cognitive psychology*, 6(4):451–474, 1974; and John R Anderson and Lynne M Reder. The fan effect: New results and new theories. *Journal of Experimental Psychology: General*, 128(2):186, 1999

The Fan Effect and Retrieval Times

In the fan effect, memories that share the same attribute are harder to retrieve, and take longer time, than memories that have unique attributes when the attribute itself is used as a cue; this is explained by the denominator n being larger for common attributes.

To see how that happens, let us consider the set up of prototypical fan experiment. In this paradigm, participants are asked to memorize a series of pairs of associated nouns. Typically, the pairs are a proper nouns that indicate a person and a location, and the cover story is to memorize where each person is located—for example, "The Hippie is in the Park" [61]. Persons and locations are distributed so that some persons are in one location only, while other persons would visit two or three locations. Similarly, certain locations will only contain one person, while othe location would be visited by two or three persons. Figure 31 shows an example of stimuli used in a experiment. The number of persons associated with each location, and the number of locations associated with a given person, represent the *fan* of the corresponding noun. In the figure, blue boxes indicate nouns with a fan of one, such as "Lawyer" and "Store"; these boxes have only one incoming arrow. Green boxes indicate nouns with a fan of two, such as "Hippie" and "Park": they have two incoming arrows. Finally, red boxes indicate nouns with a fan of three, and they receive three arrows from three other names.

[61] This is a real stimulus from the original experiment. It was, after all, in the '70s.

Equation 27 can be simplified to make the fan effect predictions a bit clearer. Specifically, $WlogP(q|m)$ can fixed to a constant value k. The amount of spreading activation is then $k - W\log(n)$, with n being the fan number associated with a given noun. In fact, for large n, the quantity $W\log(n)$ can become larger than k, making the spreading activation component negative. It follows that the additional spreading activation for any noun is diminished in proportion to the number of associated fans. In other words, knowing more facts about a fact might make it harder to retrieve that fact.

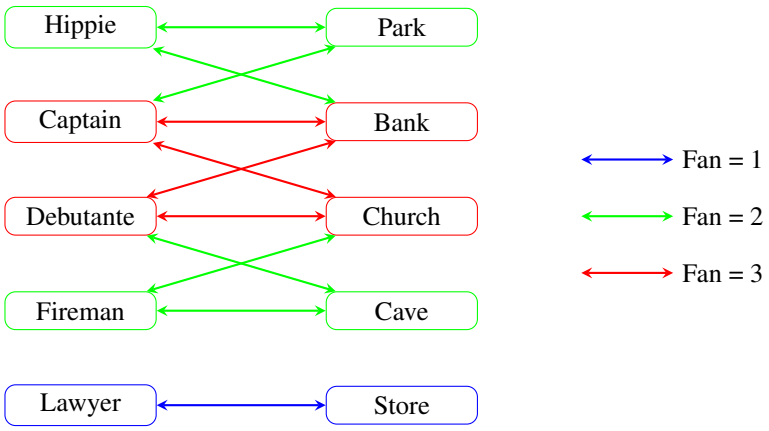


Figure 31: Typical design of a Fan Effect experiment

In a fan experiment, p[articipants are asked to judge whether statements connecting persons and locations are true or not, e.g. "Is the Hippie in the Park?" (Yes, it is not). The nouns in Figure 31 are set up so that, no matter whether a participant attempts to retrieve all the locations associated with "Hippie" or all teh persons in the "Park", the fan is always the same (Fan $= 3$, in this case). Thus, it is possible to predict which particular pair would have lower spreading activation, and thus would be more difficult to answer.

Because participants make very few errors in laboratory memory experiments, the fan effect is typically measured not in terms of probabilities of retrieving the correct pair, but in terms of response times. A fundamental law of memory is that memories that are more difficult to remember (and therefore more likely to be forgotten) also take longer to retrieve. This is modeled by connecting the time T it takes to retrieve a specific memory to its activation, as shown in Equation 28:

$$T = T_{er} + F\ \mathrm{e}^{-f\ A(m)} \tag{28}$$

It is possible to use Equation 28 to derive predictions for what we would expect. First, we split the exponent $A(m)$ into the baseline and the spreading components:

$$\begin{aligned} T &= T_{er} + F\mathrm{e}^{-f\ [B(m)+k-log(n)]} \\ &= T_{er} + F\left[\mathrm{e}^{-f\ [B(m)+k]}\ e^{-f\times -log(n)]}\right. \\ &= T_{er} + F\left[\mathrm{e}^{-f\ [B(m)+k]} \times e^{fn}\right] \\ &= T_{er} + F\left[\mathrm{e}^{-f[B(m)+k]} \times f \times n\right] \end{aligned}$$

Notice that the second term is multiplied by n: this implies every new association between nouns contributes *linearly* to the time it takes to retrieve the correct sentence. The data from the original experiment [62] confirms this prediction:

[62] John Robert Anderson. Retrieval of propositional information from long-term memory. *Cognitive psychology*, 6(4):451–474, 1974

Connection to Accumulator Models

The model outlined above is a theory of a how a memory trace can be retrieved with time. This process can be seen as a form of race model, in which the decision to be made is the retrieval of a memory and the process involves the accumulation of evidence until a certain threshold is reached. In fact, accumulator models were originally proposed precisely to explain the mechanisms of retrieval from long-term memory [63].

A deeper connection between accumulator models and the ACT-R memory theory was drawn by Maarten van der Velder and colleagues [64]. To understand the relationship between the two process, let's recall the equations that control the time it takes for memory retrieval:

$$T = Fe^{-fA(m,t)}$$

and the equation that controls the retrieval time in the Linear Ballistic Accumulator models:

$$T = (A-k)/v$$

The first equation can be rewritten as:

$$\begin{aligned} T &= Fe^{-fA(m,t)} \\ &= F/e^{fA(m,t)} \end{aligned}$$

The two equations now have the same structure, and become the same if we assume that

$$\begin{aligned} F &= A-k \\ v &= e^{A(m,t)} \end{aligned}$$

Note that, since the quantity $A(m,t)$ is supposed to reflect the *log* odds of retrieving m, the drift rate v simply reflects the odds of retrieving m.

$$v = \frac{P(m)}{P(\neg m)}$$

Also, it follows that, if we have modeled a memory retrieval process with LBA, we can translate our findings in ACT-R by calculating directing the activation level of the retrieved memory:

$$A(m,t) = \log v$$

[63] Roger Ratcliff. A theory of memory retrieval. *Psychological review*, 85(2):59, 1978

[64] Maarten van der Velde, Florian Sense, Jelmer P Borst, Leendert van Maanen, and Hedderik van Rijn. Capturing dynamic performance in a cognitive model: Estimating act-r memory parameters with the linear ballistic accumulator. *Topics in Cognitive Science*, 14(4):889–903, 2022

Part II

Neural Networks

Perceptrons and Feedforward Networks

In the previous chapters we have seen theoretical frameworks that describe the computations of certain brain circuits. However, we have rarely discussed *how* brain circuits can carry out these computations, or any computation at all.

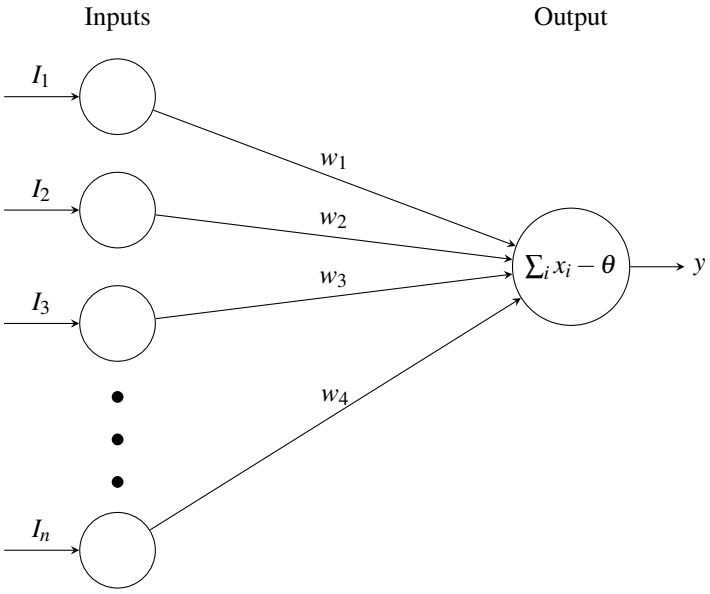
An alternative to these approaches is to model the basic computations of the neurons first, and see how different behaviors can be generated by it. This approach is often known as *connectionism* or *parallel distributed processing*.

The McCullogh-Pitts neuron

The simplest model of the neuron was proposed by McCullogh and Pitts in 1943 [65]. At the time, computers were in their infancy but the theory behind them was already established. Among other things, it was established that universal computing machines could work with binary representations that are operated upon by a set of logical gates.

McCullogh, a neuroscientist, and Pitts, a mathematician, noted that biological neurons could be thought of as logical gates. A neuron, they reasoned, can be in one of two states: It is either firing an action potential, or not. These two states can be mapped to the binary codes of 1 and 0 in a digital computer. Neurons are interconnected, thus each neuron receives inputs from a subset of other neurons. These connections can be thought of as a set of logical predicates (each of them also being 0 or 1) entering a logical proposition, and the neuron's response as the output of such proposition. Abstracting away from all of the biological details, they proposed a highly stylized model of the neuron's computations.

[65] Warren S McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, 1943



Although extremely simple, this model proved phenomenally useful. In fact, pretty much everything else in the history of artificial neural networks has been built on small variations of this model.

In this model, the neuron's soma sums up all of the inputs from all of its input neurons $x_1, x_2 \ldots, x_n$. This sum is compared to an internal threshold θ, which represents the neuron's membrane potential. If the sum of all the inputs of all neurons exceeds the threshold, the neuron fires. If not, the neuron remains silent. Formally, the output of the neuron, y, is defined as:

$$y = \begin{cases} 1, & \text{if } \sum_i x_i - \theta > 0 \\ 0, & \text{if } \sum_i x_i - \theta \leq 0 \end{cases} \tag{29}$$

Neurons as Logic Gates

The original goal of McCullogh and Pitts was to show that assemblies of their simplified neurons (and, by extension, the human brain) could work to compute Boolean functions. In computer science, Boolean functions, such as AND and OR, are known as *logical gates* and, because they operate on and return only the binary values 0 and 1, they are the basic components of the bit-wise operations of any modern digital computer.

As a start, let us consider the two most common Boolean functions, the AND and OR gates. These functions can be expressed as simple table that express the desired output, 0 or 1, for each combination of their two inputs. The AND function will return 1 only if both of its arguments are 1, while the OR gate will return 1 if at least one of its arguments is 1 (Tables 5 and 7).

To implement these gates, we can create a simple network with two input neurons connected to a single output neuron. The architecture of such

x_1	x_2	y
0	0	0
0	1	0
1	0	0
1	1	1

Table 5: The AND logical gate

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	1

Table 6: The OR logical gate

network is shown in Figure 32:

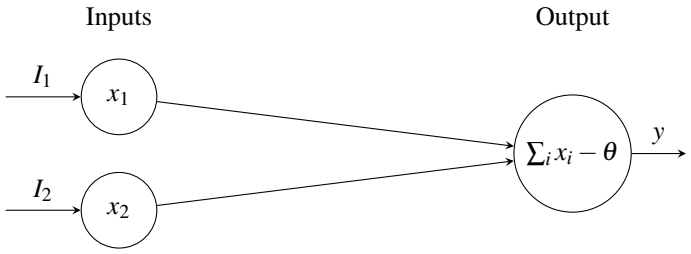


Figure 32: A network of McCullogh-Pitts neurons designed to simulated a two-argument logical gate, such as AND or OR. The input units represent the truth values of its arguments, while the output value represents the truth value of the corresponding function.

It is very simple to set up this network to implement either an AND or a OR gate by changing the value of the threshold θ of the output neuron. This is easy to visualize if we represent the values of the two input neurons as axes of a bi-dimensional plane (Figure **??**. The summed inputs can be visualized as a line passing the plane, and the threshold θ as the horizontal offset of that line. By adjusting θ, the line can be moved to effectively isolate the responses we want from the output neuron. When $\theta = 0$, an output of 1 from a single input neuron is sufficient to elicit a response in the output neuron. When $\theta = 1$, however, both neurons need to active to elicit a response.

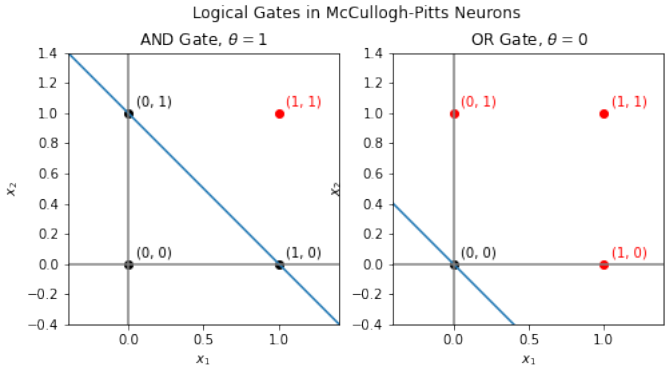


Figure 33: A single McCullogh and Pitts neuron can work as an AND or as an OR logic gate by setting its threshold θ at different values. In both panels, the blue line represents the sum of the two inputs (x_1 and x_2) minus the threshold; points represent the four possible input configurations; red points represent input configuration that trigger a response in the output neuron.

As it turns out, however, AND and OR gates are *not* sufficient to create a digital computer: other logical functions exist that cannot be created by simply combining ANDs and ORs. One of the fundamental gates that is needed to create a working computer is the NOT gate, which corresponds to the logical operation of negation.

The NOT gate takes a single argument x and returns 1 if the $x = 0$, and 0 if $x = 1$. It is easy to see that, no matter which threshold value is picked, there is not way to make the output neuron respond in the desired way.

x	y
0	1
1	0

Table 7: The NOT logical gate

To make it work, we need to introduce the next important approximation to the artificial neuron, the synaptic weight w. Much like real synapses, synaptic weights in neural networks modulate the effect of the input neuron on the output neuron. Specifically, the output neuron now fires based on thresholded, weighted sum of its inputs, i.e. $\sum_i w_i x_i - \theta$.

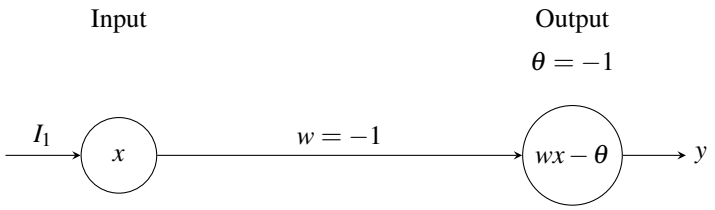


Figure 34: A network implementing the NOT gate with McCullogh-Pitts neurons

The NOT gate can be implemented with two McCullogh-Pitts neurons, one for the input and one for the output, as shown in Figure 34. The correct behavior can be achieved by manipulating both threshold and synaptic weight, and, specifically, by setting $w = -1$ and $\theta = -1$.

Complex Logical Operations

With the addition of synaptic weights, we can now combine multiple McCullogh-Pitts neurons to create complex logical circuits. It can be demonstrated, in fact, that arrangements of McCullogh-Pitts neurons can be used to create digital computers—a CPU is, in the very end, a collection of logical gates. The possibility of creating *any* complex logical operation out of basic logical operations is called *functional completeness* and is an important property of any computing device. If a set of logical gates is functionally complete, then any arbitrary logical function can be built out of them and, because all of the operations on a computer are logical operations (as they work on bnary values), then any function can be computed with them. It can be mathematically shown, for example, that AND and NOT are functionally complete; thus, because these these two functions can be implemented with McCullogh-Pitts neurons, McCullogh and Pitts neurons are functionally complete.

A canonical test case for functional completeness is the "Exclusive OR" gate, or XOR. This is a logic function that take two binary values and returns 1 if *only one* of its values is 1, and return zero if they are both 1 or both 0 (Table 8)

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0

Table 8: The XOR logical gate

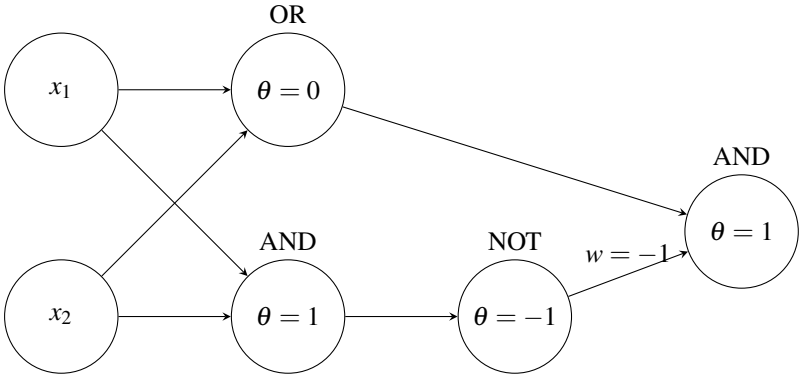


Figure 35: A network implementing the XOR gate with McCullogh-Pitts neurons

Perceptrons

The McCullogh-Pitts neuron is strictly binary to reflect the property, that, at any point in time, neurons either fire or do not. However, while apparently faithful to the biological properties of neurons, this simplification misses the point that neurons also transmit information over time. In this sense, it becomes important to think not only of whether a neuron fires, but also how frequently. This fact is well-known to neurophysiologist, who indicate the frequency of action potentials as the neuron's *firing rate*.

One could have MP neurons firing at realistic time levels. This approach, known as *spike*-coding, pushes for realistic simulations.

A simpler approach is to use continuous activation functions and take their output value as a proxy of how much a neuron is firing. This approach is called *rate*-coding, and is the dominant approach in artificial neuron networks. All of contemporary AI is based on rate-coded neural networks.

When one moves from the binary units of McCullogh and Pitts to rate-coded neurons with continuous activation functions, the first problem to solve is, which activation function should we use? We will start this chapter with the simplest possible function, that is, a *linear* function. The output of a linear unit y_j is simply the summed of the outputs of its input neurons, weighted by the respective synaptic weights, minus a threshold:

$$y_j = \sum_i w_{i,j}\, x_i - \theta \tag{30}$$

These type of linear neurons are typically called *perceptrons* [66] and were the bases of an early type of brain-inspired digital computer capable of learning, the Mark I. They key to perceptrons is that, unlike McCulloch and Pitts neurons, they can learn how to perform new tasks.

[66] Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958

Learning in Perceptrons

The main idea behind perceptrons (and, in fact, artificial neural networks in general) is that they can learn in a way that is similar to the way the human brain learns. Looking at Equation 30, it is clear that there are only a few terms that can be operated upon:

1. The threshold θ. A neuron can become more or less resistant to change from its inputs. This is the bases of the so-called *BCM learning rule* [67].
2. The inputs x_j. In particular, new units can be added or removed from the network. This corresponds to the biological processes of cell death and neurogenesis, and is the basis of the learning algorithm known as *Cascade Correlation* [68].
3. The synaptic weight $w_{j,i}$. This corresponds to the biological process of long-term potentiation (LTP) and long-term depression (LTD), and is

[67] Elie L Bienenstock, Leon N Cooper, and Paul W Munro. Theory for the development of neuron selectivity: orientation specificity and binocular interaction in visual cortex. *Journal of Neuroscience*, 2(1):32–48, 1982

[68] Scott E Fahlman and Christian Lebiere. The cascade-correlation learning architecture. Technical report, Carnegie-Mellon University, School of Computer Science, Pittsburgh, PA, 1990

perhaps the most ubiquitous form of learning in the nervous system.

As it happens, virtually all learning algorithms for neural networks are expressed in terms of $w_{j,i}$. In fact, changes in a neuron's threshold or adding/removing units can also be expressed in terms of conveniently modifying synaptic weights, so this has become the most fundamental way to frame learning in artificial neural networks.

Learning as Gradient Descent

Although learning algorithms for neural networks take different forms that are highly dependent on the structure of the network, most of them share a the same general structure: they are all form of *gradient descent* over the network's error function.

The idea of gradient descent can be easily understood imagining the simplest possible network, a network that contains two input neurons and one output neuron. Let's say that we want the neuron to perform the logical AND function: it needs to return 1 when both of its neurons are 1.

For such network, we can imagine an error function that is the sum of the squared response error over all the possible patterns: (0, 0), (0, 1),(1, 0), (1, 1). For all the values of w_1 and w_2, we can calculate the output neuron's response (it's just $(w_1x_1 + w_2x_2)^2$ across the four possible combinations of x_1 and x_2). The result is a 3D surface, shown in Figure 36

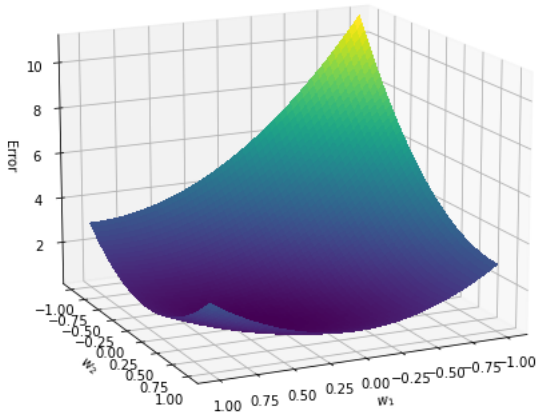


Figure 36: Error function for a simple perceptron computing the AND logical function

Note that the surface never actually reaches zero, which means that this perceptron will never learn to give the *exact* solution. However, for different values of w_1 and w_2, the network makes smaller or larger error. More importantly, there is a point in which the network's error is the smallest

possible. This location, called the *global minimum*, is where we want to train our network to be.

Now, imagine to place a ball anywhere on that surface; it would roll downhill towards the minimum. In doing so, the ball will naturally move towards states that have the lowest potential energy, which is, in turn, a function of the height of the surface. When rolling down along the surface doing so, the ball will naturally follow the direction of the steepest decline. The idea of gradient descent is to simply borrow this physical analogy and transfer it to the error surface. Like a ball moves to minimize its elevation, so we want to move to reduce the height of the error curve. In fact, the idea of gradient descent is so general that virtually every learning algorithm can be expressed as a form of gradient descent[69].

[69] For instance, all of the Reinforcement Learning algorithms in the previous chapter can be formalized as gradient descent over an error surface.

In a neural network, gradient descent can be formalized as follow. First, let's define an error function that captures the degree to which the behavior of the network departs from the intended or target behavior. Such an error function is commonly know as a *loss function* an indicated with the letter L. In general the error or loss function is defined over a set of patterns P, each of which is a combination of input vectors $\mathbf{x}_p$ and a desired target response value t_p. For each pattern, we can record the squared difference between the target response t_p and the actual neuron response y_p. As defined above, the loss function is the sum these squared differences over all possible patterns P. Formally:

$$L = \frac{1}{2}\sum_p (t_p - y_p)^2 \tag{31}$$

What we want to do is to change the synaptic weights in such a way that would reduce the value of L. To do this, we calculate the first derivative of the loss function:

$$\Delta w_{i,j} = -\frac{\partial L}{w_{i,j}} \tag{32}$$

To calculate this quantity, we will use the *chain rule*, splitting the partial derivative as the product of two derivatives of related quantities:

$$\frac{\partial L}{w_{i,j}} = \frac{\partial L}{y_j} \times \frac{\partial y}{w_{i,j}}$$

These two quantities represent (1) the change in error due to a neuron's activity and (2) the change in a neuron's activity that is due to a synapse. They can be calculated separately.

The first quantity is easy to calculate, as it is the very definition of error in Eq. 31, and is just the squared difference between the target and the actual value of the neuron. In turn, this is a an easy derivative to calculate:

$$\begin{aligned}\frac{\partial L}{y_j} &= \frac{\partial \frac{1}{2}\sum_p (t_j - y_j)^2}{y_j} \\ &= \frac{\frac{1}{2}\partial \sum_j (t_j - y_j)^2}{y_j} \\ &= \frac{1}{2} \times 2 \times (t_j - y_j) \\ &= (t_j - y_j)\end{aligned}$$

The second quantity $\partial y_j / \partial w_{i,j}$ depends on the activation function. In our simplified world of perceptions, the activation function is linear, so $y_j = \sum_i wi, jx_i$:

$$\begin{aligned}\frac{\partial y_i}{w_{i,j}} &= \frac{\partial \sum_i w_{i,j} x_i}{w_{i,j}} \\ &= \frac{w_{i,j} x_i}{w_{i,j}} \\ &= x_i\end{aligned}$$

Putting both results together, we have a definition of the gradient descent learning rule for perceptrons:

$$\frac{\partial L}{w_{i,j}} = (t_j - y_j) \times x_i$$

This quantity gives us the slope of the error curve at a particular point in time. To learn the correct values for each $w_{i,j}$, we are just going to follow the gradient downwards, and adjust the synaptic weight accordingly:

$$w_{i,j}^{new} \leftarrow w_{i,j}^{old} - \eta (t_j - y_j) \times x_i \tag{33}$$

In Eq. 33, the parameter η is the *learning rate*, and is equivalent to the parameter α in reinforcement learning. It has a negative sign because the learning happens in the opposite direction (i.e., downwards) of the slope of the curve [70]. In the neural networks literature, it is common to express these learning rules as a function of $\Delta w_{i,j}$, that is, the change in synaptic weight, and write them as:

[70] Now you are ready to answer this question: Why was α positive in RL? Which function were we climbing *upwards?*

$$\Delta w_{i,j} = -\eta (t_j - y_j) \times x_i$$

To implement gradient descent, a perceptron is first initialized with random synaptic weights, and then *trained* by applying Eq. 33 over a series of consecutive *epochs*, progressively changing the synaptic weights until we reach the lowest value of the error function.

The plots in Fig. 37 illustrate this progress. The plots represent data from the perceptron representing the AND gate and whose error surface in depicted in Fig. 36. Specifically, the plots represent how the two synaptic

weights w_1 and w_2 (left two panels) and the error function (third panel) change over 50 epochs of training with a learning rate of $\eta = 0.05$. In this case, we initialized the weights to the values $w_1 = w_2 = -1$, which give the greatest error value in Fig. 36. With every training epoch, the weights are corrected upwards and the error declines. Eventually, after 50 epochs, the error function reaches a plateau and the synaptic weights reach an asymptotic value of 0.33. At about this value, the network error is minimal. As we noted before, the network never reaches a "perfect" response, but it does reach a reasonable approximation. The fourth panel in Fig. 37 shows the perceptron's responses (as blue bars) to the four possible inputs of a logic gate, together with the desired responses (as red dots). The perceptron over-responds to the two cases, (0, 1) and (1, 0) in which one of its inputs is turned on, and under-responds when they are both on. It would still be possible, however, to create a digital circuit out of this neuron, for example by applying a threshold at $\theta = 0.5$ (the grey dashed line).

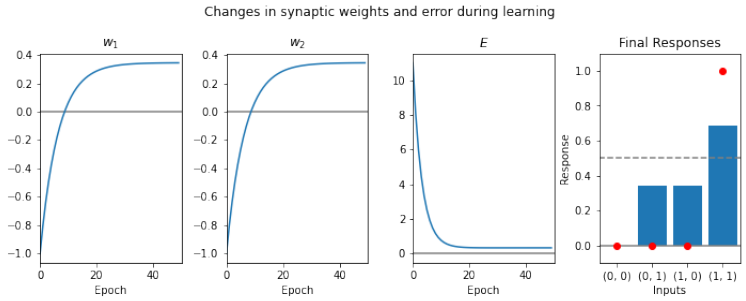


Figure 37: A perceptron learning the AND logical gate.

As the third panel shows, the gradient descent algorithm has identified a place in which the error is minimal: any change in synaptic weights would either increase the error to the responses to (0, 1) and (1, 0) or increase the error for (1, 1). The error plot in Figure 37, in fact, is a 2D visualization of the trajectory followed by the network as it descends the error surface in Figure 36. Figure 38 visualizes the consecutive steps of this path over the error surface, showing how it follows the steepest descent down to the basin of the curve.

!h

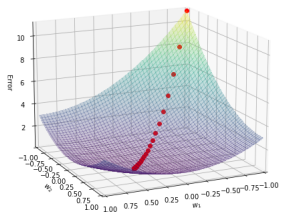


Figure 38: The learning path of Figure 37 overlaid over the error surface of Figure 36

.

Thresholds and Bias Units

It is possible to apply the principle of gradient descent to find solutions not only for each synapses $w_{i,j}$ but also for each threshold θ of each neuron. In fact, the two approaches can be applied together, further improving the performance of a perceptron.

Although learning algorithms exists that specifically work on the neuronal thresholds [71], this is, however, rarely done in practice. The reason is that it is possible and easy to recast the role of a neuronal threshold in terms of synapses. This can be done by expanding the network adding one more

[71] Elie L Bienenstock, Leon N Cooper, and Paul W Munro. Theory for the development of neuron selectivity: orientation specificity and binocular interaction in visual cortex. *Journal of Neuroscience*, 2(1):32–48, 1982

"dummy" input neuron whose activation is always $x = -1$. This dummy neuron is called the *bias unit* and indicated as b. The product $x_b w_b$ of the bias unit's output and the corresponding synaptic weights plays the same functional role as the threshold θ. By learning the optimal value of w_b together, a perceptron is effectively learning the optimal value of the threshold θ.

Image Recognition with Perceptrons

For all of their theoretical interest (and we will return to them), logical gates are not the most relevant task that neurons have evolved to solve. Instead, we will apply our learning rule to a simple *visual* task, the recognition of character. After all, there is a reason these models are called "perceptrons"!. In fact, character recognition is one of the best-known benchmark tasks in machine learning and it has a an associated database, MNIST [72], of thousands of hand-written digits that is routinely used to measure of the performance of different algorithms.

To keep things simple, we will start with a *much* simplified version of MNIST stimuli, one in which all digits are stylized, LED-like numbers in a 7×4 grid (Figure 39).

To start, we will train a simple perceptron to recognize just one number, the number 7. To do so, we will create a network designed as in Figure.

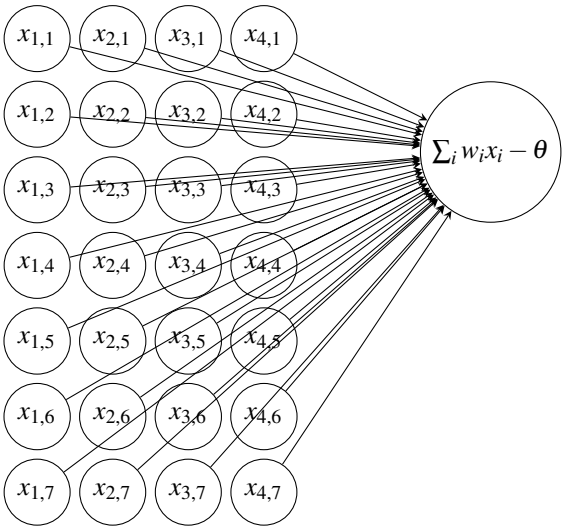


Figure 40: Architecture of a perceptron for recognizing the digits of Figure 39

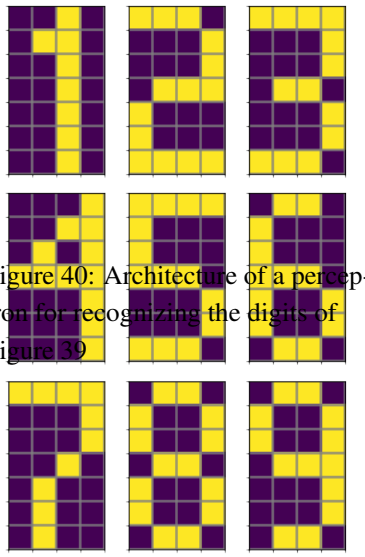


Figure 39: Simple single-digit stimuli for a perceptron.

In this figure, the cells of the digit matrix represent the inputs to the neuron. We can think of them as neurons whose activation value is clamped and fixed to the value of the corresponding cell. All of these 7×4 neurons connect to a single output neuron, whose intended target responses is 1 if the number is 7 and 0 otherwise.

To do

[72] Yann LeCun. The mnist database of handwritten digits. *http://yann. lecun. com/exdb/mnist/*, 1998

Examining the Weights

In neural networks, virtually all of the computations occur through the synapses that connect different layers of neurons. For this reason, examining the synapses is the easiest way to understand what the network has learned during training. Figure 41 visualizes the synapses connecting the input neurons to the output unit after the perceptron has been trained to recognize each of the digits of Figure 39.

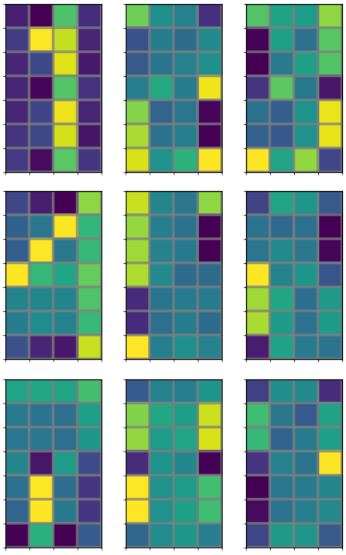


Figure 41: Weights of the perceptron after training it to recognize each of the digits in figure 39

Limits of Perceptrons

Although perceptrons do seem amazing, they are actually pretty limited, as was pointed out as early as the '60s [73]. Their main weakness is that they rely on linear activation functions—a fact that inherently limits their ability to approximate functions.

As an example, let's try to train a perceptron to perform the XOR function we have seen in section II. The perceptron for the XOR gate has the same structure as the perceptron for the AND gate; the only thing that changes is the specific patterns of inputs and targets it is trained on. Unfortunately, gradient descent cannot find a solution for such problem, as shown in Figure 42.

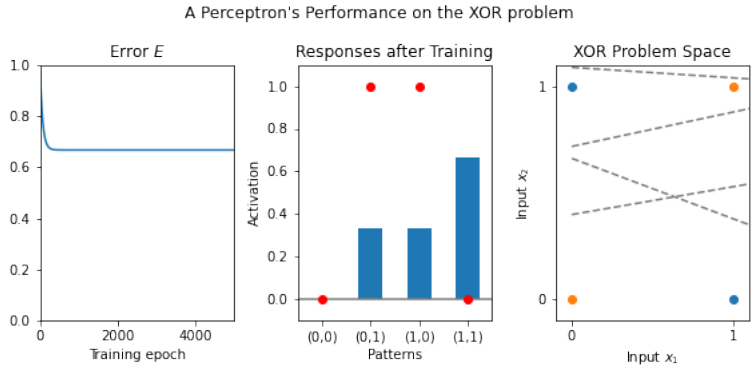


Figure 42: A perceptron cannot solve the XOR problem

In the case of McCulloch and Pitts neurons, it was possible to create a XOR network by wiring together multiple neurons in layers. Would it be possible to do so with perceptrons? The answer, sadly, is no. The key difference is that the McCullogh and Pitt neurons use a step activation function (Fig. 29). No matter how many layers are used, a network of perceptrons will never be able to learn to even approximate a XOR function. The reason, again, lies in the use of a linear activation function: the sum of multiple linear functions is still a linear function. Thus, any two neurons wired up in sequence with synaptic weights w_1 and w_2 can always be reduced to a single neuron with a synaptic weight equal to $w_1 \times_{w} 2$. No matter how many layers we add, a network of perceptrons with a linear activation function remains a *linear* classifier.

[73] Marvin Minsky and Seymour A Papert. *Perceptrons: An introduction to computational geometry*. MIT press, 2017

In fact, perceptrons are mathematically equivalent to linear regression models in statistics: while they can extract the ideal combination of weights to predict a particular outcome from data, they can only do so if the intended responses are *linearly separable*.

Problems like the XOR gate, by contrast, are not linearly separable: if we represent the problem on a 2D space, with the two axes being the possible values of the input neurons x_1 and x_2, it is not possible to draw a straight line that separates the correct solutions (blue dots in the left panel of Figure 42) from the incorrect ones (orange dots).

Feedforward Networks and Backpropagation

To overcome the limitations of perceptrons, it is necessary to have networks with multiple layers of neurons and non-linear outputs. The general mathematical expression of non-linear neurons is:

$$y = f(\sum_i w_i x_i - \theta)$$

where f is the neuron's *activation function*. The two most typical non-linear activation functions are the logistic function $y = 1/(1+e^x)$ and the hyperbolic tangent function $y = \tanh(x)$. Both functions are non-linear and have a bounded range (between 0 and 1 and between -1 and 1, respectively), which is helpful in preventing a neuron's output to grow unbounded. Finally, and unlike the step function, both functions are continuous and derivable—an important characteristic for making them trainable through backpropagation. The two functions are illustrated in Figure 43.

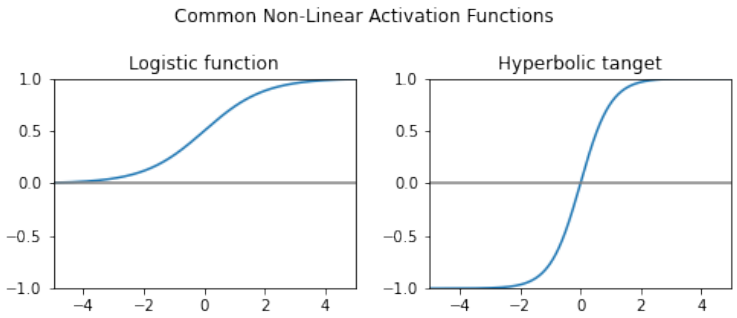


Figure 43: Two common non-linear activation functions, the logistic function (left) and the hyperbolic tangent function (right)

Backpropagation

Backpropagation is a generalization to the principle of gradient descent to the case of non-linear neurons. It was discovered at least twice, first by Paul Werbos in his 1974 dissertation and then again in 1986 by Rumelhart, Hinton, an Williams who published the paper that popularized it [74] and that kickstarted the neural networks revolution of the '80s.

[74] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986

The fact that units are non-linear complicates the situation quite a bit, but the math can still be worked out. As in gradient descent, we can use the chain rule to decompose the derivation of the error function. Only, this time we will apply the chain rule twice, dividing the derivative of the error function into three components: (1) the change in error E due o the change in the output of a neuron y_j; (2) the change in the output of a neuron due to a change in its input in_j, and (3) the change in the input of a neuron due to a change in the synapses $w_{i,j}$ projecting to neuron j.

Because this decomposition would lead us to consider the output of neurons in different layers, we will add an extra index to our notation, using the expression y_j^l to indicate the output of the j-th neuron in the l-th layer. (Note that, in this notation, l does not represent an exponent, but yet another index). Furthermore, layers will be numbered consecutively, with the first layer ($l = 1$) corresponding to the input layer and the last layer corresponding to the output layer of a network.

With this notation in place, we can now define the new gradient descent rule as:

$$\frac{\partial L}{\partial w_{i,j}} = \frac{\partial E}{\partial y_j^l} \frac{\partial y_j^l}{\partial \text{in}_j^l} \frac{\partial \text{in}_j^l}{\partial w_{i,j}} \tag{34}$$

Now, each of these terms can be examined separately. The easiest term is the last one, which can be simplified as follows:

$$\begin{aligned} \frac{\partial \text{in}_j^l}{\partial w_{i,j}} &= \frac{\partial \sum_i y_i^l w_{i,j}}{\partial w_{i,j}} \\ &= y_i^l \end{aligned}$$

Conceptually, this result has a straightforward meaning—the change in the net input of a neuron in layer due to a change in the synapses is simply the amount of activation that flows through that synapses, that is, the output of the projecting neuron from the previous layer.

The second term in Eq. 34 is also fairly simple to calculate:

$$\frac{\partial y_j^l}{\partial \text{in}_j^l} = \frac{\partial f(\text{in}_j^l)}{\text{in}_j^l)} = f'(\text{in}_j^l)$$

Again, the interpretation is straightforward: the change in the output of a neuron due to a change in its input is just the derivative of its activation function (this is literally the definition of a derivative!).

The remaining term $\partial L / \partial w_{i,j}$, however, is a bit harder to calculate. If the neuron y_j^l is in the output layer, then the change is simply the derivative of the error function, which we have already calculated in the case of perceptrons

$$\frac{\partial E}{\partial y_j^l} = \frac{\partial \frac{1}{2}(t_j(p) - y_j^l)^2}{\partial y_j^l} = t_j(p) - y_j^l$$

But what if neuron j is not an output neuron and belongs to a middle layer instead? In general, it is impossible to solve this case exactly without knowing the precise geometry of the network and working out all the details for how errors propagate through different layers. We can, however (and this is the genius intuition!) simply express the effects of a change in the neuron output on the global error as a function of the effects that change in neuron output has on the error *of the next layer* $l+1$. Because we can calculate exactly the change in error for the last (output) layer, the error of all the previous layers can be calculated recursively.

Specifically, the quantity $\partial E / \partial y_j^l$ can be rewritten as:

$$\frac{\partial L}{\partial y_j^l} = \sum_k \frac{\partial E}{\partial y_k^{l+1}} \frac{\partial y_k^{l+1}}{\partial \text{in}_k^{l+1}} w_{j,k} \tag{35}$$

This follows from the consideration that, whatever the change in error E due to the change in output y_j, it will necessarily propagate to the next layer through the effects that j has on the inputs of all of the neurons j projects to—which are, in turn, propagated through their specific synapses $w_{j,k}$. Notice that the first two terms of Eq. 35 are just the second and third term of backpropagation for layer $l+1$. This implies that backpropagation can be applied recursively, by first calculating the error for the output layer, and then calculating the error of all the other layers by multiplying the error in the layer above by the synaptic matrix between the two layers.

The power of backpropagation (and, for that matter, its very name) lays precisely in its recursive formulation. Once a stable estimate for errors is computed at the output layer, the adjustments in synaptic weights become a simple chain of operations, in which the results of the previous step are used as an argument for the current calculations. It is also extremely general, as it allows to train complex networks with an arbitrary number of layers. To see these types of computations in action, we will now consider the following implementation.

Implementation

At the core, all the operations of a neural network can be expressed in the form of linear algebra. This is the internal representation used by all modern software packages, and it is key to the great speed-up in performance for neural models that was made possible by the availability of GPUs (which operate on numeric value matrices rather than logical instructions).

In this formulation, a layer of neurons in position i is represented as a vector $\mathbf{y}_i$, and the synapses between the i-th and the jth layers of neurons are represented as a matrix $\mathbf{W}_{i,j}$ (notice that now i and j indicate layers, not individual neurons!). The inputs to layer j are the product between the synaptic matrix and the transpose of the input layer vector: $\mathbf{W}_{i,j}\,\mathbf{y}_i^T$. The activation function f now becomes a vector function $f(\mathbf{x})$, which is applied

to all the elements of vector $\mathbf{x}$.

To propagate an specific pattern through a network, we perform the following matrix operations recursively. Starting with the *second* layer $i = 2$:

1. Calculate the *input* to layer i: $\mathbf{in}_i = \mathbf{W}_{i-1,i}\, \mathbf{y}_{i-1}^T$
2. Calculate the *output* of layer i: $\mathbf{y}_i = f(\mathbf{in}_i)$
3. Repeat until there are no more layers.

During training, instead, the network is examined backwards, updating each synaptic matrix recursively. Specifically, starting with the synaptic matrix $\mathbf{W}_{i,j}$ that is closest to the output layer, we perform the following operations:

1. Calculate the *derivative* of the following layer j, $\mathbf{y}'_j = f(\mathbf{y}_j)$
2. Record the *error* $\mathbf{e}_j$ of the following layer j. If layer j is the output layer, its error can be calculated directly: $\mathbf{e}_j = t(\mathbf{p}) - \mathbf{y}_j$. If j is not the output layer, its error would have been calculated in the previous pass.
3. Record the output of the previous layer, $\mathbf{y}_i$
4. Update the synaptic matrix $\mathbf{W}_{i,j}$: $\Delta\mathbf{W}_{i,j} = \eta\mathbf{y}_i[\mathbf{y}'_j \odot \mathbf{e}_j]$
5. Finally, calculate the error in the previous layer, $\mathbf{e}_i$, by propagating back the error of the next layer through the synaptic matrix: $\mathbf{e}_i = \mathbf{e}_j W_{i,j}^T$
6. Repeat the steps above for the next synaptic matrix down the network's hierarchy, until there are no more synapses to update.

How Many Layers? The Computational Power of Feedforward Networks

Backpropagation allows us to relax the two limitations of perceptrons, namely, linear activation functions and single layers. But how powerful are our neural networks now? Can we achieve the same power of a digital computer, like the original McCulloch-Pitts neurons? And, if so, how many layers would be needed?

The surprising answer is given by Cybenko's theorem [75], and is that three non-linear layers are all that is needed emulate any digital computer. More precisely, Cybenko's theorem states that any computable function (that is, any function that can be executed on a digital computer) can be approximated, to an arbitrary degree of precision, by a three-layer feed-forward neural network with sigmoid activation neurons. Note that Cybenko's theorem is highly theoretical: Why McCulloch and Pitts showed exactly how their types of neurons could be combined to create arbitrary logical functions, Cybenko's theorem only guarantees that there exists one feedforward network whose synaptic matrices are set up in such a way to *approximate* the function to a desired level. More importantly, it does not say anything about how many

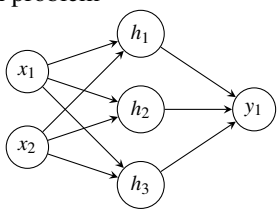
[75] George Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, 2(4):303–314, 1989

neurons would be needed, nor how such network would need to be trained. Still, the result is impressive and certainly contributed to popularizing this type of networks. I think most neuroscientists would agree that sigmoid feedforward neural networks were the most common type of network one would find in modeling papers of the 90s and 00s.

But let's test Cybenko's theorem with a practical example: Training a feedforward neural network to perform the XOR logical function. In this case, the network can be trained over the full list of possible arguments of the function, i.e., the pairs (0,0), (0,1), (1,0) and (1,1).

In this specific example, we will create a network with six neurons: two in the input layer (to represent the initial logical values), one in the output layer (to represent the logical gate's output), and three sandwiched in the middle layer, which is commonly known as the *hidden* layer of the three-layer network. The network will be initialized by giving each synapse a small random value, and will be trained with backpropagation over all of the four known examples in the training set. The learning rate will be set to $\eta = 0.01$.

Figure 44: Architecture of a feed-forward neural network to solve the XOR problem



All of the neurons will use the logistic function (Figure 43, left) as their activation function. The logistic function has the attractive property of having a convenient derivative, which has traditionally made it a darling of neuron netwok models. Specifically:

$$
\begin{aligned}
y &= \frac{1}{1+\mathrm{e}^{-x}} \\
y' &= \frac{1}{1+\mathrm{e}^{-x}} \times \left(1 - \frac{1}{1+\mathrm{e}^{-x}}\right) \\
y' &= y \times (1-y)
\end{aligned}
$$

That is, the derivatived of the activation function y' can be calculated directly by multiplying the neuron's output y by its complement $(1-y)$, further simplifying the calculations.

The left panel in Figure 45 shows how the error of the network changes over the different epochs of training. Unlike the smooth gradient descent of perceptrons, the profile of the error function alternates between plateaus and sharp drops in the error. This is one of the consequences of the non-linearity of the network. The right panel shows the network response to the XOR patterns, using the same conventions of Figure 37: red dots represent the desired response and blue bars represent the network's output to each pair. Unlike perceptrons, the network can learn an extremely good approximation to the XOR function—in fact, as Cybenko's theorem states, it is possible learn an *arbitrarily* good approximation by varying the size of the hidden layer and the duration of training.

In the case of a feedforward neural network, we can examine what has been learned by inspecting the activity elicited by the different training inputs on the hidden layer. In our case, the hidden layer is made of three neurons, h_1,

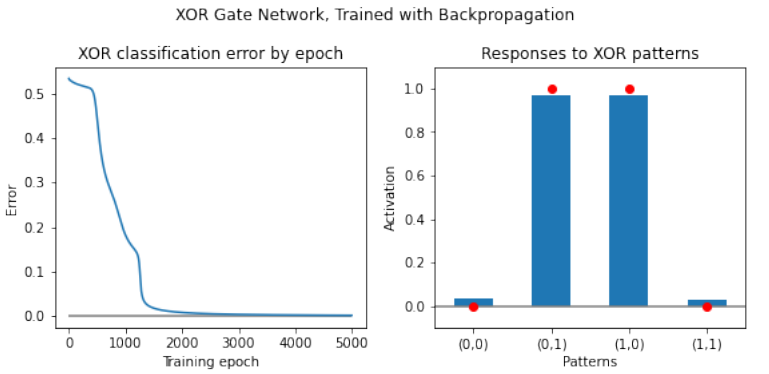


Figure 45: (*Left*) Decline in the network error over 5,000 training epochs; (*Right*) Responses of the XOR network trained with backpropagation

h_2, and h_3, and four inputs were used during training. Figure **??** illustrates the different responses of the three hidden units to the four possible inputs:

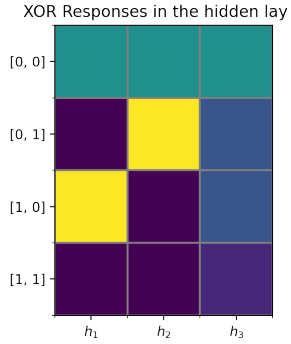


Figure 46: Responses of the three hidden neurons of the XOR network to the four possible logical inputs

Convolutional Neural Networks

The simple character recognition system we have seen so far would not cut it for long. Although it is possible to train a simple feedforward network to perform extremely well on a database such as MNIST, some substantial obstacles remain to its realistic use. This is because, despite its variety, the MNIST stimuli are still very carefully chosen. All the numbers are isolated, centered, and occupy roughly the same size. If any of these assumptions is violated, however, the network would fail.

Consider, for example, *scale* invariance. All of the stimuli in MNIST database are carefully designed to take roughly the same amount of space on the image. But the size of handwriting also varies across individuals, with some writing in very large, and some others in very small, characters. Once a network has learned how to respond to a large "7", it will likely not generalize to a small "7". Another problem is *translational* invariance. The MNIST digits are all nicely centered in the middle of their matrices. Finally, in the most complicated case of object recognition, we have the problem of *viewpoint* invariance: We can recognize a object or a friend's face even if seen from unusual points of view.

These limitations do not violate Cybenko's theorem. As we noted before, Cybenko's theorem does not say how a network could be trained to approximate any function. In the limit, approximating the function might require training the network for an inconceivably long amount of time over all of the possible examples of this function.

Instead, what we really want is a network that can generalize what it has learned and apply it to different stimuli.

Convolutional neural networks differ from the simple networks we have seen so far in three fundamental aspects.

The first is that they use multiple hidden layers. It is the use of multiple layers that has given this research the name of "deep learning".

The Convolutional Layers

The first is the use of convolutional layers and units with limited receptive field. To

To understand how convolutional layers work, let's consider a hypothetical CNN built from scratch. The input layer of our network will be a 16×16 matrix. Instead of being connected to each unit in the input layer, units in the hidden layer are connected only to a subset of units. These connections are topologically organized.

In addition to having a limited and partially overlapping receptive field, units in a convolutional layer have another characteristic: All of the input

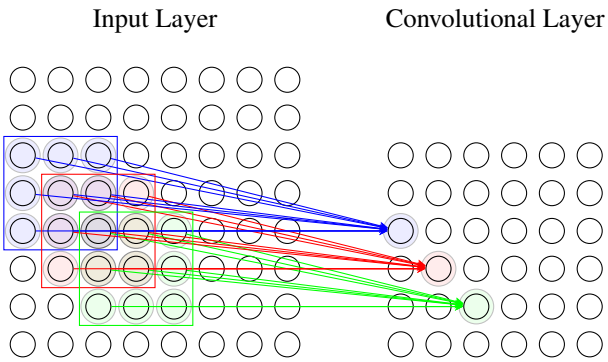


Figure 47: Architecture of a convolutional layer

synapses have exactly the same weights.

To understand how that works.

The Subsampling Layers

In addition to convolutional layers, CNNs typically contain subsampling layers, also known as *maxpooling* layers. In fact, CNNs typically alternate convolutional and subsampling layers.

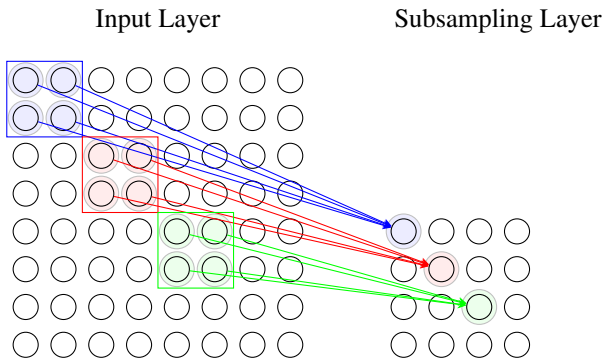


Figure 48: Architecture of a subsampling (or maxpooling) layer

ReLUs

The third and final ingredient in CNNs (and, in fact, in most modern neural networks) is the use of a particular form of activation function known as the Rectifier Linear Unit, or ReLU . A ReLU unit responds only if its summed input is greater than zero. If that is the case, the unit simply returns its summed input, otherwise, it returns zero:

$$y = \begin{cases} \sum_i x_i, & \text{if } \sum_i x_i > 0 \\ 0, & \text{if } \sum_i x_i \leq 0 \end{cases}$$

As usual, the threshold θ is ignored and we will assume that a bias unit is automatically added to the unit's inputs.

ReLUs offer many advantages. First, although the ReLU function is non-linear, its positive part is a simple linear function, which greatly simplifies all computations. Second, because they are silent (i.e., $y = 0$) for all values that do not exceed the threshold. Imagine a network that is initially set up with small random synaptic values: Statistically, it is expected that about half of the neurons would received small negative inputs. As a result, about half of the units would be silent. This makes it easier to sparsify the representations. Among the disadvantages of ReLUs is the fact that their activation function is discontinuous and, therefore, non-differentiable. Because of this, some researchers prefer to use the softplus function $y = \log(1/(1+e^x))$. Most modelers, however, simply split the ReLU into two differentiable parts, the linear part when its inputs are $x > 0$ (and whose derivative is $dy/dx = 1$) and the constant part where the inputs are $x < 0$ (and whose derivative is $dy/dx = 0$).

AlexNet

The ideas outlined before were originally presented by Yann LeCun in a 1998 paper that achieved unprecedented success in recognizing the handwritten digits of the MNIST database [76]. However, it was a 2012 paper that really made them mainstream. That paper introduced a system called AlexNet [77], which learned to correctly classify (with up to > 80% accuracy) the 1.3 million images contained in the ImageNet database. It is impossible to overestimate the impact that AlexNet has had on both AI and computational neurosciences; at the moment of writing, that paper has amassed over 100,000 citations, making it one of the most cited scientific paper of all time, across all fields.

[76] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998

[77] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25, 2012

The Visual System

One of the remarkable thing about CNNs is the degree to which they resemble the architecture of the visual system.

As an example, compare the 96 kernels learned by AlexNet's first convolutional layer with the actual recordings of V1 neurons in the macaque.

ere explain artiphysiology and how AlexNet's cells are remarkably similar to real V4 cells

Image Computable Models

Image computable vs non-image computable models

To give an example of how remarkable AlexNet is, we can compare it to two other models.

The first is Poggio's model.

The second is Pasupathy's model.

Deep Learning

The series of ideas outlined in the previous sections, based on stacking multiple convolutional layers, are the foundation of what is now called “deep learning”, which has become, since the 2010s the dominant field of AI.

Hebbian Learning and Autoassociators

Since its first proposal, backpropagation has been both celebrated for its power (it can be generalized to train any network) and caused many eyebrows to rise for its apparent lack of biological plausibility. After all, for backpropagation to mimic the type of learning that happens in biological neurons, a number of things would need to occur:

- Networks would need to have purely feedforward projections;
- Somehow, they would need to be able to calculate the an error function that is dependent on the response of the neurons they are projecting to;
- Somehow, these error values would need to find their way back without feedback projections;
- The error values should be modulated by the derivative of the neurons' activation function.

Critics typically point out that, exactly the artificial neuron, the backprop algorithm is a convenient approximation of the underlying biological process. For example, feedback projections could exists and be used specifically for conveying errors during learning; derivatives can be calculated through spike trains [78]. Nonetheless, the gravity and sheer amount of these concerns have pushed some researchers to look for alternative learning rules that do not violate as many assumptions.

[78] Brian N Lundstrom, Matthew H Higgs, William J Spain, and Adrienne L Fairhall. Fractional differentiation by neocortical pyramidal neurons. *Nature neuroscience*, 11(11):1335–1342, 2008

Classic Hebbian Learning

Classic Hebbian learning is perhaps the first neural learning rule ever outlined. It was popularized by psychologist Donald Hebb in a 1949 book [79] and is often described as the principle that "neurons that fire together, wire together". The idea behind this principle is that two neurons that often fire at the same time end up forming more synapses between them. Eventually, when the synapses between the two of them are sufficiently strong, the firing of a single neuron is sufficient to trigger an action potential in the second neuron. Such a mechanisms is generally compatible with the process of long-term potentiation (LTP) as currently understood.

[79] Donald Olding Hebb. *The organization of behavior: A neuropsychological theory*. Psychology Press, 2005

In this simple version, Hebbian learning can be described by Equation 36:

$$\Delta w_{i,j} = \eta x_i x_j \tag{36}$$

where η is the usual learning rate, $w_{i,j}$ is the weight of the synapses between the i-th and the j-th neuron, and x_i and x_j are their outputs. It is easy to say that, if both neurons fire, their activation x will be $x > 0$ and therefore the synaptic weight w will increase. If, on the other hand, at least one neuron is not firing, then their output is $x = 0$ and the synaptic weight will not increase.

Equation 36 provides an intuitive way to capture simple forms of "associative" learning, that is, conditioning or the acquisition of stimulus-response associations.

Thus, classic Hebbian learning is unstable, as its growth is not compensated by any mechanisms. Although helpful even in its current form, for most pratical applications the basic Hebbian principles of Equation 36 needs to be augmented with extensuions that dampen its unchecked growth, giving rise to a variety of modified Hebbian rules.

Contrastive Hebbian Learning

One such rule is Contrastive Hebbian Learning (CHL [80]). CHL takes place in the same type of multi-layered network that is used by Backprop; only, instead of being strictly feedforward, the network needs to have both feedforward and feedback projections between consecutive layers. This change in the network's architecture introduces interesting dynamics–and poses some problems. Because the network has both feedback and feedforward projections, a single pass will not be sufficient to determine the network's output. Instead, in this case, the input neurons will need to be kept at their input values while all the other neuron's outputs are recalculated multiple times. In analogy to what happens in electrophysiological experiments, the input neurons are said to be *clamped* to their input values. The process of updating the In CHL, learning between two neurons i and j takes place following the simplest Hebbian rule:

[80] Geoffrey E Hinton. Deterministic boltzmann learning performs steepest descent in weight-space. *Neural computation*, 1(1):143–150, 1989

$$\Delta w_{i,j} = \alpha x_i x_j$$

As noted above, this updating rule is unstable. To account for this, CHL assumes that learning take places in two different phases, a *learning* phase and an *unlearning* phase. During the learning phase, the network is forced to reproduce a target output response t, while during the unlearning phase the network is free to converge to its actual response a. To force the network to converge to the desired response t, the network's output neurons are clamped to their desired states, and the network is let to go through multiple update phases until it settles while both the input and the output neurons are

clamped. The total update occurring on every synapse w is the difference of the Hebbian updates between the two phases, i.e.:

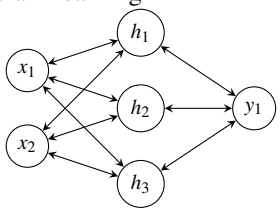
$$\Delta w_{i,j} = \alpha(x_i^t x_j^t - x_i^a x_j^a) \tag{37}$$

where x^t represents the output of a neuron when the the network is being forced to reproduce the desired target t, and x^a is the output of a cell in when the network is let free to converge towards its own actual response.

A network trained with CHL can learn the same patterns of a network trained with backpropagation. As an example, we can use CHL to train a small, three-layered neural network to solve the XOR problem. The network has the same topology of the feedforward XOR network in Figure 44, and consists of six neurons arranged in three layers 49. The only differences is that, in this case, connections between layers are bidirectional, so that, for every synapse connection neuron i to neuron j, a different synapse also exists that connections j to i.

Figure 50 illustrates the results of training the network of Figure 49. The network was trained with

Figure 49: Architecture of a recurrent neural network trained to solve the XOR problem with Contrastive Hebbian Learning



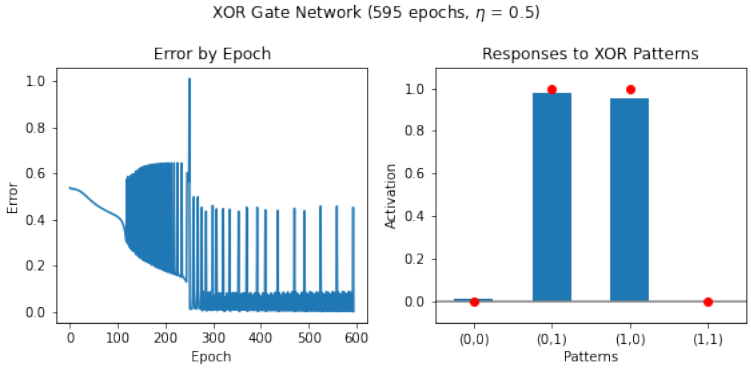


Figure 50: Training a recurrent network to solve the XOR problem with Contrastive Hebbian Learning. *Left:* Changes in the error value as learning progresses; *Right:* Final performance on the XOR problem.

Convergence and Energy Function

To understand how CHL works, it is necessary to first appreciate what drives the dynamics of a network that includes both feedforward and feedback projections. Such a network will cycle through multiple states, and converge to a state in which the activity of all neurons will remains stable. It can be shown that such a state can be defined as a local minimum for the network's *energy* E, a quantity defined as such:

$$E = -\frac{1}{2}\sum_i^N \sum_j^N w_{i,j} x_i x_j \tag{38}$$

When given appropriate inputs, the network will converge towards the closest state with minimum energy. To train a recurrent network, therefore, we need to reduce the energy associated with the target response until it is

as low as the level of energy of the actual response. This is equivalent to minimizing an error function that corresponds to the difference between the energy of the actual state and the energy of the actual state:

$$\begin{aligned} E &= E(t) - E(a) \\ &= -\frac{1}{2}\sum_{i}^{N}\sum_{j}^{N} w_{i,j}x_i^t x_j^t - -\frac{1}{2}\sum_{i}^{N}\sum_{j}^{N} w_{i,j}x_i^a x_j^a \end{aligned}$$

We can therefore express learning as a form of gradient descent over this error function. As in the case of perceptrons and backpropagation, we can derive a learning algorithm by performing gradient descent over this error function, and taking its negative derivative. The CHL equation (Eq. 37) ends up being the result of this derivation process:

$$\begin{aligned} \Delta w_{i,j} &= -\frac{\partial E}{\partial w_{i,j}} \\ &= -\frac{\partial\left(-\frac{1}{2}\sum_i^N\sum_j^N w_{i,j}x_i^t x_j^t - -\frac{1}{2}\sum_i^N\sum_j^N w_{i,j}x_i^a x_j^a\right)}{\partial w_{i,j}} \\ &= -\frac{1}{2}(x_i^t x_j^t - x_i^a x_j^a) \\ &\approx x_i^a x_j^a - x_i^t x_j^t \end{aligned}$$

Thus, both backpropagation and CHL are learning rules that can be derived from gradient descent; they simply perform gradient descent on different error functions. In CHL, the error function is the difference in the energy of the desired target state and the energy of the actual state of the network. The CHL equation works by reducing the energy of the state corresponding to the target out t until it becomes lower that the energy of the actual output a, at which point, the network would naturally converge towards the target state. Because backpropagation is designed to work on feedforward networks, it cannot use the difference in energy states as its error function—feedforward networks have no meaningful definition of energy since they are updated in a single pass and do cycle through different states. Instead, the error function in backpropagation is defined as the distance between the desired target output t and the actual one a. Figure 51 provides a visual illustration of these differences.

In fact, it can be mathematically shown that CHL and Backpropagation are equivalent, and problems solved with backpropagation can be also solved with CHL [81].

[81] Xiaohui Xie and H Sebastian Seung. Equivalence of backpropagation and contrastive hebbian learning in a layered network. *Neural computation*, 15(2):441–454, 2003

Oja's Rule

Despite its advantages, CHL is still a form of supervised learning, and, as such, needs a set of known examples and a supervising trainer to tell the

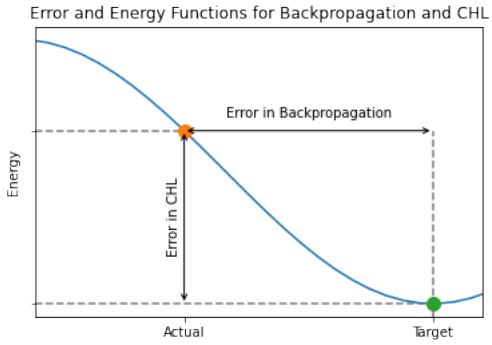


Figure 51: A comparison of back-propagation and CHL. The figure illustrates the energy values (blue line) associated with different possible states of the network, including a target (green) and the actual response (orange). CHL is the derivative of the difference in the the energy states, while back-propagation is the derivative of the difference between target and actual responses

network when a specific response should be learned or unlearned. The brain must have, instead, some basic Hebbian mechanism that are both stable and unsupervised.

One way to make Hebbian learning stable is to constrain the unbounded growth of synapses. To contrain this growth, many algorithms have been proposed that normalize synaptic weights. One of such methods is Oja's rule [82], in which synaptic weights are updated according to the equation:

$$\Delta w_{i,j} = y_j(y_i - w_{i,j}y_j) \tag{39}$$

[82] Erkki Oja. Simplified neuron model as a principal component analyzer. *Journal of mathematical biology*, 15(3):267–273, 1982

Like in Hebbian learning, in this rule, synaptic weights grow in proportion to both y_i and y_j. The term $w_{i,j}y_j$, however, penalizes the growth of the synaptic weight beyond 1. It also favors the growth of synaptic weights in proportion to the frequency at which neurons i and j fire together in the training sample. These characteristics make Oja's rule especially apt for extracting stable patterns in the data, that is, combinations of neurons that are commonly active in the data. In fact, it can be shown that a neuron trained with Oja's rule learns to extract the first principal component of its data.

To see an application of Oja's rule, let us consider again the digit recognition problem. Unlike the clear-cut digits of Figure 39, real hand-written digits have considerable variations from each other. As an example, Figure 52 provides fifteen examples from the MNIST database [83].

[83] Yann LeCun. The mnist database of handwritten digits. *http://yann. lecun. com/exdb/mnist/*, 1998

Despite their obvious relative differences, the fifteen digits bear some clear similarity between them: in all cases the "3" is drawn following the same basic trajectory, which makes some pixels in an arc in the top, an arc in the bottom,and in the very center to be more likely to show up in the digit. A network trained with Oja's rule would be able to understand this pattern and extract the commonalities between all of them.

Figure 53 shows the learning progression of a network trained with Oja's rule on the fifteen digits of Figure 52. The network is made of $28x28$ input neurons connected to a single output neurons y with a linear activation function—that is, a perceptron. In each training epoch, all of the fifteen

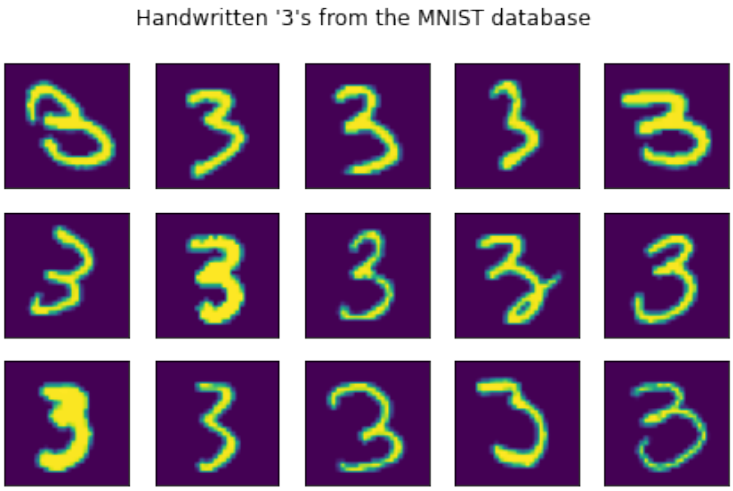


Figure 52: Examples of handwritten “3”s from the MNIST database

exemplars are presented once. When one pattern is presented, the activation of the input neurons is set to the values of the corresponding pixels, the activation of the output unit is calculated, and the synapses between input and output units are calculated according to Equation 39. As Figure 53 shows, the network is quickly learning the shape of a prototypical “3”, extracted from the training examples. As a result, the output unit would be maximally sensitive to any digit that most resembles a “3”.

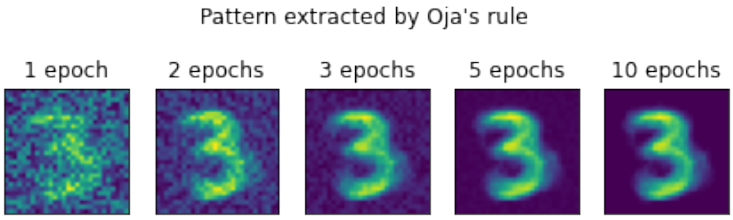


Figure 53: A network trained with Oja’s rule over a variety of hand-written digits (Figure 52 has learned their common features

Autoassociators and Memory

All of the types of networks we have seen so far have the characteristic of being *heteroassociative*: they learn an association between a specified input and its desired output.

This type of networks suffer from problems. To see an example, let’s consider what would happen if we train the same network of Figure X on only half of the patters

Hopfield Networks

Hopfield networks are a particular type of autoassociator networks introduced by physicist John Hopfield in 1982 [84].

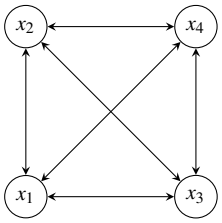
The Hopfield network is one of the simplest and oldest types of neural networks. It remains, however, one of the most influential and among those with the deepest implications for the cognitive neurosciences. A Hopfield network is made of N independent neurons, each of which is connected to all other neurons, as in Figure 54. Thus, a Hopfield network has no "layers" and no internal subdivisions, and all neurons are on equal footing.

There are no self-connections, that is, neurons do not project back to themselves. In addition, the connections are symmetric, so that $w_{i,j} = w_{j,i}$. Finally, each neuron is a binary unit, whose output is either $+1$ or -1 and whose activation function is:

$$y_i = \begin{cases} -1 & \text{if } \sum_j w_{i,j}\, x_j \leq 0 \\ +1 & \text{if } \sum_j w_{i,j}\, x_j > 0 \end{cases} \tag{40}$$

[84] John J Hopfield. Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the national academy of sciences*, 79(8):2554–2558, 1982

Figure 54: An example of a Hopfield network, with $N = 4$ neurons fully interconnected with each other.



Learning in a Hopfield Network

Learning in a Hopfield network happens in a single pass. Specifically, when the network is in a specific state that needs to be learned, its synapses are updated using the Hebbian learning rule:

$$w_{i,j} = w_{j,i} = x_i\, x_j$$

What makes this types of networks unique is that, once a specific pattern has been memoried, the network can memorize an additional one by simply re-applying the Hebbian learning rule above. Unlike feedforward networks or recurrent networks trained with CHL, Hopfield networks can learn new things without catastrophic interference.

State Transitions in a Hopfield Network

Now, to understand what happens, we need to refer back to. At any point in time, the state of each neuron might change, flipping from -1 to $+1$ as the states of all other neurons in the network change. These changes do not (usually) continue indefinitely; the network continues to update its state until a stable pattern of neuronal activity is found.

These changes do not happen randomly. In his seminal paper, Hopfield showed that the network attempts to minimize its energy function. The energy function was defined in Eq. 38. Because, in a Hopfield network, the synapses are symmetric, we can redefine the energy function as such:

$$E = \frac{1}{2} \sum_{j>i}^{N} w_{i,j} x_i x_j \tag{41}$$

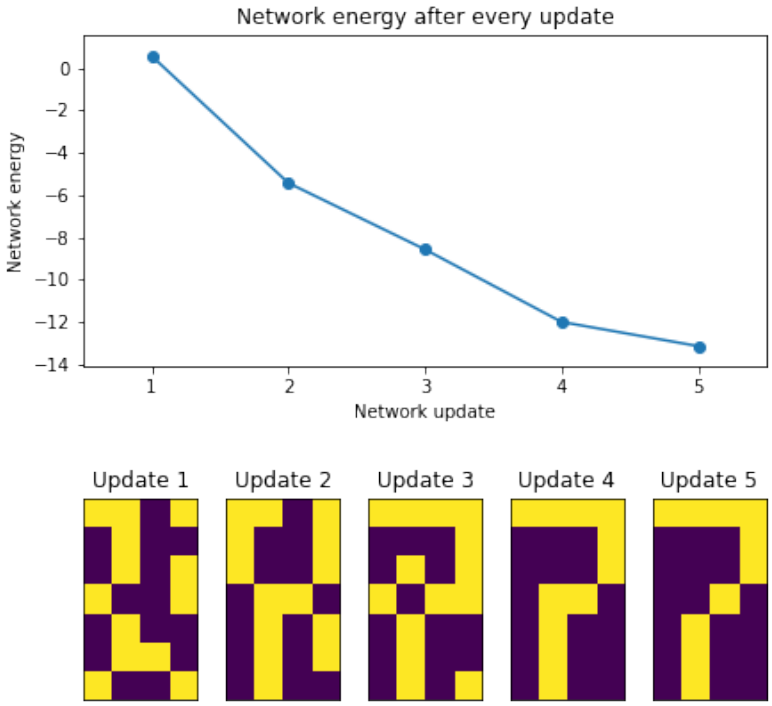


Figure 55: When presented with a pattern of neural activity, a Hopfield network will spontaneous switch to a lower-energy one until a stable pattern is found. At this point, the network has reached a stable configuration. This configuration coincides with one of the learned patterns.

Hopfield Networks and the Hippocampus

As it turns out, Hopfield networks are a simple but really good model of how the human brain can form episodic memories. To see how, let's revisit one of our favorite circuits, the hippocampus.

Recurrent Neural Networks

Recurrent Networks

1. Simple Recurrent Network, or Elman network. As the hidden layer chains two consecutive epochs (and therefore two consecutive stimuli), it provides a way to chain two consecutive events in time. Like the backup term in RL, this allows for propagating back prediction errors from epoch t to t 1. Unlike RL, however, the hidden layer does not suffer from temporal difference methods' inability to overcome non-Markov environment and long-distance dependencies between stimuli. This is because the hidden layer, contains a fading memory of all previous epoch presentations and, although stimuli presented much earlier are destined to be much more faintly represented, they are still marginally present in the hidden layer, and their learning is still reflected in its synaptic layers. To draw a parallel with RL, the hidden layer reflects elements of both the one-step backup term in the RPE (the copying of the layer from one epoch to the next) and of eligibility traces (the presence of fading traces of previous epochs).
2. Jordan network
3. Long-Term Short-Term Memory networks. Recurrent networks provide a powerful way to learn temporal dependencies across stimuli. Yet, they suffer from some severe disadvantages. The main one is the problem of the so-called vanishing gradient: It is true that the network can learn remote temporal dependencies, but the more distant the dependency between two stimuli, the more the original stimulus would have faded from the hidden layer. This problem is, in fact, serious enough that even relatively short distance dependencies might require many million epochs of training to be learned.
4. Natural Language Processing
5. Application: Prefrontal Cortex, meta-learning.

Bibliography

[1] Paul M Fitts. The information capacity of the human motor system in controlling the amplitude of movement. *Journal of Experimental Psychology*, 47(6):381, 1954.

[2] William E Hick. On the rate of gain of information. *Quarterly Journal of Experimental Psychology*, 4(1):11–26, 1952.

[3] Ray Hyman. Stimulus information as a determinant of reaction time. *Journal of experimental psychology*, 45(3):188, 1953.

[4] David Marr. *Vision: A computational investigation into the human representation and processing of visual information*. W. H. Freeman & Company, 2010.

[5] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[6] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[7] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016.

[8] Wolfram Schultz, Peter Dayan, and P Read Montague. A neural substrate of prediction and reward. *Science*, 275(5306):1593–1599, 1997.

[9] Richard S Sutton. Learning to predict by the methods of temporal differences. *Machine learning*, 3(1):9–44, 1988.

[10] A David Redish. Addiction as a computational process gone awry. *Science*, 306(5703):1944–1947, 2004.

[11] Gavin A Rummery and Mahesan Niranjan. *On-line Q-learning using connectionist systems*, volume 37. University of Cambridge, Department of Engineering Cambridge, UK, 1994.

[12] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.

[13] Andrew G Barto. Adaptive critics and the basal ganglia. *Models of information processing in the basal ganglia*, 215, 1995.

[14] Yuji Takahashi, Geoffrey Schoenbaum, and Yael Niv. Silencing the critics: understanding the effects of cocaine sensitization on dorsolateral and ventral striatum in the context of an actor/critic model. *Frontiers in neuroscience*, 2:14, 2008.

[15] Henry H Yin and Barbara J Knowlton. The role of the basal ganglia in habit formation. *Nature Reviews Neuroscience*, 7(6):464–476, 2006.

[16] Richard S Sutton. Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. In *Machine learning proceedings 1990*, pages 216–224. Elsevier, 1990.

[17] Edward C Tolman. Cognitive maps in rats and men. *Psychological review*, 55(4):189, 1948.

[18] Roger Ratcliff. A theory of memory retrieval. *Psychological review*, 85(2):59, 1978.

[19] Martijn J Mulder, Eric-Jan Wagenmakers, Roger Ratcliff, Wouter Boekel, and Birte U Forstmann. Bias in the brain: a diffusion model analysis of prior probability and potential payoff. *Journal of Neuroscience*, 32(7):2335–2343, 2012.

[20] Sebastian Musslick, Amitai Shenhav, Matthew M Botvinick, and Jonathan D Cohen. A computational model of control allocation based on the expected value of control. In *The 2nd multidisciplinary conference on reinforcement learning and decision making*, 2015.

[21] Scott D Brown and Andrew Heathcote. The simplest complete model of choice response time: Linear ballistic accumulation. *Cognitive psychology*, 57(3):153–178, 2008.

[22] Adam Reeves, Nayantara Santhi, and Stefano DeCaro. A random-ray model for visual search and object recognition. *Spatial Vision*, 18:73–83, 2005.

[23] Larry R Squire. Memory systems of the brain: a brief history and current perspective. *Neurobiology of learning and memory*, 82(3):171–177, 2004.

[24] John R Anderson and Robert Milson. Human memory: An adaptive perspective. *Psychological Review*, 96(4):703, 1989.

[25] John R Anderson. *The adaptive character of thought*. Lawrence Erlbaum Associates, 1990.

[26] John R Anderson. Retrieval of information from long-term memory. *Science*, 220(4592):25–30, 1983.

[27] John R Anderson, Lynne M Reder, and Christian Lebiere. Working memory: Activation limitations on retrieval. *Cognitive psychology*, 30(3):221–256, 1996.

[28] John R Anderson, Daniel Bothell, Michael D Byrne, Scott Douglass, Christian Lebiere, and Yulin Qin. An integrated theory of the mind. *Psychological review*, 111(4):1036, 2004.

[29] Hermann Ebbinghaus. *Über das gedächtnis: untersuchungen zur experimentellen psychologie*. Duncker & Humblot, 1885.

[30] John R Anderson and Lael J Schooler. Reflections of the environment in memory. *Psychological science*, 2(6):396–408, 1991.

[31] John R Anderson and Gordon H Bower. *Human Associative Memory*. Psychology Press, 2014.

[32] John R Anderson. *How can the human mind occur in the physical universe?* Oxford University Press, 2009.

[33] Richard M Shiffrin and Mark Steyvers. A model for recognition memory: Rem—retrieving effectively from memory. *Psychonomic bulletin & review*, 4(2):145–166, 1997.

[34] Douglas L Hintzman. Minerva 2: A simulation model of human memory. *Behavior Research Methods, Instruments, & Computers*, 16(2):96–101, 1984.

[35] Allen Newell and P Rosenbloom. Mechanisms of skill acquisition. In John Robert Anderson, editor, *Cognitive skills and their acquisition*, chapter 1, pages 1–56. Lawrence Erlbaum Associates, 1981.

[36] Philip I Pavlik Jr and John R Anderson. Practice and forgetting effects on vocabulary memory: An activation-based model of the spacing effect. *Cognitive science*, 29(4):559–586, 2005.

[37] John R Anderson, Jon M Fincham, and Scott Douglass. Practice and retention: a unifying analysis. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, 25(5):1120, 1999.

[38] Philip I Pavlik and John R Anderson. Using a model to compute the optimal schedule of practice. *Journal of Experimental Psychology: Applied*, 14(2):101, 2008.

[39] Florian Sense, Friederike Behrens, Rob R Meijer, and Hedderik van Rijn. An individual's rate of forgetting is stable over time but differs across materials. *Topics in cognitive science*, 8(1):305–321, 2016.

[40] Timothy T Rogers, Matthew A Lambon Ralph, Peter Garrard, Sasha Bozeat, James L McClelland, John R Hodges, and Karalyn Patterson. Structure and deterioration of semantic memory: a neuropsychological and computational investigation. *Psychological review*, 111(1):205, 2004.

[41] Christian Lebiere and John R Anderson. A connectionist implementation of the act-r production system. In *Proceedings of the fifteenth annual conference of the Cognitive Science Society*, pages 635–640, 1993.

[42] John R Anderson. *The architecture of cognition*. Lawrence Erlbaum Associates, 1983.

[43] Allan M Collins and Elizabeth F Loftus. A spreading-activation theory of semantic processing. *Psychological review*, 82(6):407, 1975.

[44] Alan Baddeley. Working memory. *Science*, 255(5044):556–559, 1992.

[45] Alan D Baddeley and Robert H Logie. Working memory: The multiple-component model. 1999.

[46] Alan Baddeley. Working memory. *Current biology*, 20(4):R136–R140, 2010.

[47] Michael J Kane, M Kathryn Bleckley, Andrew RA Conway, and Randall W Engle. A controlled-attention view of working-memory capacity. *Journal of experimental psychology: General*, 130(2):169, 2001.

[48] Gregory C Burgess, Jeremy R Gray, Andrew RA Conway, and Todd S Braver. Neural mechanisms of interference control underlie the relationship between fluid intelligence and working memory span. *Journal of experimental psychology: general*, 140(4):674, 2011.

[49] Larry Z Daily, Marsha C Lovett, and Lynne M Reder. Modeling individual differences in working memory performance: A source activation account. *Cognitive Science*, 25(3):315–353, 2001.

[50] John Robert Anderson. Retrieval of propositional information from long-term memory. *Cognitive psychology*, 6(4):451–474, 1974.

[51] John R Anderson and Lynne M Reder. The fan effect: New results and new theories. *Journal of Experimental Psychology: General*, 128(2):186, 1999.

[52] Maarten van der Velde, Florian Sense, Jelmer P Borst, Leendert van Maanen, and Hedderik van Rijn. Capturing dynamic performance in a cognitive model: Estimating act-r memory parameters with the linear ballistic accumulator. *Topics in Cognitive Science*, 14(4):889–903, 2022.

[53] Warren S McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, 1943.

[54] Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

[55] Elie L Bienenstock, Leon N Cooper, and Paul W Munro. Theory for the development of neuron selectivity: orientation specificity and binocular interaction in visual cortex. *Journal of Neuroscience*, 2(1):32–48, 1982.

[56] Scott E Fahlman and Christian Lebiere. The cascade-correlation learning architecture. Technical report, Carnegie-Mellon University, School of Computer Science, Pittsburgh, PA, 1990.

[57] Yann LeCun. The mnist database of handwritten digits. *http://yann. lecun. com/exdb/mnist/*, 1998.

[58] Marvin Minsky and Seymour A Papert. *Perceptrons: An introduction to computational geometry*. MIT press, 2017.

[59] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986.

[60] George Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, 2(4):303–314, 1989.

[61] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[62] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25, 2012.

[63] Brian N Lundstrom, Matthew H Higgs, William J Spain, and Adrienne L Fairhall. Fractional differentiation by neocortical pyramidal neurons. *Nature neuroscience*, 11(11):1335–1342, 2008.

[64] Donald Olding Hebb. *The organization of behavior: A neuropsychological theory*. Psychology Press, 2005.

[65] Geoffrey E Hinton. Deterministic boltzmann learning performs steepest descent in weight-space. *Neural computation*, 1(1):143–150, 1989.

[66] Xiaohui Xie and H Sebastian Seung. Equivalence of backpropagation and contrastive hebbian learning in a layered network. *Neural computation*, 15(2):441–454, 2003.

[67] Erkki Oja. Simplified neuron model as a principal component analyzer. *Journal of mathematical biology*, 15(3):267–273, 1982.

[68] John J Hopfield. Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the national academy of sciences*, 79(8):2554–2558, 1982.