Mathematics of Machine Learning: An introduction

Sanjeev Arora
Princeton University Computer Science
Institute for Advanced Study

Abstract

Machine learning is the subfield of computer science concerned with creating machines that can improve from experience and interaction. It relies upon mathematical optimization, statistics, and algorithm design. Rapid empirical success in this field currently outstrips mathematical understanding. This elementary article sketches the basic framework of machine learning and hints at the open mathematical problems in it.

An updated version of this article and related articles can be found on the author's webpage.

MSC: 68-02, 68Q99, 68T05.

The dictionary defines the act of learning as *gaining or acquiring knowledge or skill (in something) by study, experience, or being taught.* Machine learning, a field in computer science, seeks to design machines that learn. This may seem to fly in contradiction to the usual view of computers as fixed and logic-based devices whose behavior is completely fixed by their programmer. But this view is simplistic because it is in fact straightforward to write programs that learn new capabilities from new experiences and new data (images, pieces of text, etc.). This learnt capability can become part of its program, and of course, any newly learnt capabilities can also be trivially copied from one machine to another.

Machine learning is related to *artificial intelligence*, but somewhat distinct because it does not seek to recreate only human-like skills in a machine. Some skills —e.g., detecting patterns in millions of images from a particle accelerator, or in billions of facebook posts— may be easy for a machine, but beyond the cognitive abilities of humans. (In fact, lately machines can go beyond human capabilities in some image recognition tasks.) Conversely, many human skills such as composing good music and proving math theorems seem beyond the reach of current machine learning paradigms.

The quest to imbue machines with learning abilities rests upon an emerging body of knowledge that spans computer science, mathematical optimization, statistics, applied math, applied physics etc. It ultimately requires us to mathematically formulate nebulous concepts such as the "meaning"of a picture, or a newspaper article. This article provides a brief introduction to machine learning.

The mathematical notion closest to machine learning is *curve-fitting*, which has long been a mainstay of science and social science. For example, the supposed inverse relationship between an economy's inflation and unemployment rates, called the *Philips curve*, was discovered by fitting a curve to economic data over a few decades. Machine learning algorithms do something similar, except the settings are more complicated and with many more —sometimes, tens of millions—variables. This raises many issues, computational as well as statistical. Let's introduce them with a simple example.

1 Introduction: the linear model

Suppose a movie review consists of a paragraph or two of text, as well as a numerical score in $[0, 1]$ (0 = worst and 1 = best). The machine is trying to learn how to predict the numerical score when given only the text part of the review. As training data, it is given N movie reviews and their scores; that is, $(x^1, y^1), (x^2, y^2), \ldots, (x^N, y^N))$ where x^i is a piece of text and y^i is a score. From this dataset it has to figure out the rule for predicting the score from the text.

If the English vocabulary has V words, then each x^i can be seen as a vector in $\Re^V$, where the j'th coordinate is the number of times the j'th word appears in this piece of text. Note that V is large, say $100,000$, so this vector representation is very sparse (i.e., has very few nonzero entries) when the text review consists of a few dozen words.

The simplest approach for prediction involves a linear model. To simplify the description, assume each review has the same length, namely, has k words. The model assumes that each word has an associated *sentiment weight*, which is a scalar. The model says that the review's score can be predicted by adding up the sentiment weights of all words in the review. Note that if a word occurs k times then it contributes k times its weight.

In other words, if $\vec{\theta}$ is the vector of sentiment weights for all V dictionary words, then the machine tries to predict y^i from $\vec{\theta} \cdot x^i$. The learning algorithm consists of finding the best fit for $\vec{\theta}$ via the classic *least squares* method.

$$\min_{\theta} \quad \sum_{i=1}^{N} (\vec{\theta} \cdot x^i - y^i)^2 \tag{1}$$

After training we expect to find that the weights assigned to words are meaningful. Positive words like *terrific, enjoyable, loved* etc. get high weights and negative words like *terrible, hated, avoid* get low or negative weights.

To finish our discussion we need to address two important issues.

1.1 Computational efficiency

The first important issue is: how efficiently can we find such a vector of weights $\vec{\theta}$? Such questions about *computational complexity* are important. Luckily, here

the algorithmic task can be solved very efficiently to optimality. The reason is that the optimization problem in (1) happens to be *convex*, a notion we define below. Under fairly general conditions, convex optimization problems can be solved efficiently.

1.2 Statistical efficiency

The second question is statistical: how do we quantitatively measure the success of learning after training with N datapoints? The end result of training is the learnt weight vector $\vec{\theta}$, and it is useful only if the machine is able to use it to predict the ratings for reviews that it *hasn't seen during training*. This is called *generalization* and it is a nontrivial issue. For instance, suppose $V = 100,000$ and we are only given $10,000$ reviews. Then by simple linear algebra of underdetermined systems, there *always* exists a weight vector such that $\vec{\theta} \cdot x^i = y^i$ for all i. Surely such a generic solution doesn't do well on unseen reviews? (This is analogous to interpolating a degree 20 curves to only 10 datapoints, and expecting it to fit unseen datapoints.) Surprisingly, in real-life it can, provided we change the above objective to the following, where λ is a scalar that is discovered by experimenting with the data, as explained below.

$$\min_{\theta} \quad \sum_i (\vec{\theta} \cdot x^i - y^i)^2 + \lambda \|\vec{\theta}\|_2^2 \tag{2}$$

To obtain guarantees on generalization, we make a key assumption: reviews used for training are *independent* samples from a *fixed distribution* on all possible reviews. This raises inconvenient philosophical questions about whether there even *exists* such an invariant distribution across all reviews —e.g., surely last year's movie reviews come from a different distribution than this year's? We brush away such questions, while noting in passing that it is an active area of research to formulate learning in more realistic settings —such as when the learner and teacher are allowed to interact, or when teacher is allowed to tailor the examples to speed up learning.

Having made that assumption, we are trying to prove that

$$\| \frac{1}{N} (\sum_i \theta^* \cdot x^i) - E_x[\theta^* \cdot x] \| \leq \epsilon, \tag{3}$$

where the expectation in the second term is over the entire distribution of reviews. At first glance this appears to be a trivial matter of bounding the difference between the population average and the sample average, in other words, to use measure concentration bounds. But actually there is a complication: the solution θ^* was computed using the sample, and thus depends intimately upon it. We handle this complication by taking a union bound over all possible θ^*.

First, we can discretize θ^* by rounding off entries in θ^* to the nearest integer multiple of ϵ, since this can affect the predicted score by at most $\epsilon/2$. Now all entries in θ^* are at least ϵ, which means there are at most $m = \|\theta^*\|^2/\epsilon^2$ of them. The number of possible choices for such vectors is at most $T = \binom{V}{m}(1/\epsilon)^m$ where

recall that V denotes the number of words in the dictionary. Now (3) follows from standard concentration bounds provided the number of training samples exceed $c_0 \log T/\epsilon^2$ for some suitable (and explicit) constant c_0. This number grows roughly as $\|\theta^*\|^2 \log V/\epsilon^2$, which is usually much smaller than V.

By now it should be clearer what role the tunable λ multiplier plays in (2). For best generalization we wish to find a solution θ^* that minimizes the ℓ_2 norm. Increasing λ penalizes solutions θ with higher ℓ_2 norm, so it serves to balance the ℓ_2 norm against the total ℓ_2 error on training data. So the algorithm can start with a high value of λ (which rules out all θ except those with very low norm) and then perform binary search to home in on a value that balances the error in (3) and the ℓ_2 norm just exactly so that the we end up with the minimum norm solution.

The above simple argument can be strengthened in various ways and ultimately connects with broader questions in statistics [HTF09] as well as beautiful parts of discrete mathematics such as VC dimension and Rademacher complexity [BDSS14].

2 Supervised learning

The above simple example illustrates a more general paradigm: *supervised learning*, which concerns learning to classify data-points after seeing many labeled examples. This is the most well-known and successful paradigm of machine learning. To illustrate it we use a famous and empirically successful example, *image recognition*. Imagine we have divided everyday objects into k classes: *chair, building, dog, drink* etc. and want to train the machine to assign the correct label when given an image. Here each image is in pixel format, so assume it is a point in $\Re^d$ The training data contains N images of each class, where N is some modest number (such as 1000). Let the labels be $\{1, 2, \ldots, k\}$. In formalizing the learning problem, it helps to think of the label y^i of x^i as a vector in $\Re^k$: it has an entry 1 in coordinate y^i and zero in other coordinates. Ideally, the learning algorithm would learn to produce labels with only one nonzero coordinate as well, which we encourage by appropriately setting up the optimization problem.

The machine has to learn a function $f_\theta \colon \Re^d \to \Re^k$ that classifies the images correctly, where θ are the parameters in the description of f_θ. The training objective –variously called *loss function* and *empirical risk*—is

$$\min_\theta \sum_{i=1}^{N} (f_\theta(x^i) - y^i)^2. \qquad \textit{(ℓ_2 loss).} \tag{4}$$

Variations of this formulation are used as well, for example the following where y_j denotes jth coordinate of y:

$$\min_\theta \sum_{i=1}^{N} \sum_{j=1}^{k} y_j^i \log(f_\theta(x^i)_j) \qquad \textit{(cross entropy loss).} \tag{5}$$

This framework for supervised learning goes by the name *Empirical Risk Minimization (ERM)* [Vap98]. The learning *generalizes* if the expected loss of the optimum solution θ^* on the entire distribution is close to that on the samples. The flip side of this issue is statistical efficiency —determining the minimum number of samples that lead to good generalization—as was discussed earlier.

Regularization. Often the performance of gradient descent on an objective $g(\theta)$—both with regards to optimization speed and generalization—is greatly aided by adding a *regularization* term h to the objective, turning it into $g(\theta) + \lambda h(\theta)$. This $h(\theta)$ term shapes the optimization landscape, and its effect can be tuned by varying the multiplier λ. The term $\lambda\|\theta\|_2^2$ in (2) is in fact a form of regularization, and aids generalization as we saw.

2.1 Mathematical optimization in machine learning

The problems in (2) (4) (5) are instances of the following general problem where $g\colon \Re^n \to \Re$ and K is a compact subset of $\Re^n$.

$$\begin{aligned} \min \quad & g(\theta) \\ & \theta \in K \end{aligned}$$

The minimum exists, but can we find it efficiently? One could imagine using a variety of algorithms to solve such an optimization problem —optimization theory is quite well-developed! Usually design of such algorithms needs to assume that the objects in question are efficiently computable. Specifically, given a θ we need to be able to (a) efficiently *compute* $f(\theta)$ and (b) check if $\theta \in K$. Both assumptions are easily true in machine learning setting.

In practice, machine learning algorithms often use some variant of gradient descent, which seems to give the best balance between performance and scalability. Basically the same algorithm that is covered in freshman calculus, this algorithm iteratively improves the solution, starting at initial point θ^0 and then finding $\theta^1, \theta^2, \ldots$, such that at step t

$$s^{t+1} \leftarrow \theta^t - \eta \nabla g(\theta^t)) \tag{6}$$

$$\theta^{t+1} \leftarrow \text{Proj}(s^{t+1}, K) \tag{7}$$

where $\eta > 0$ is called *learning rate* and $\text{Proj}(s^{t+1}, K)$ is the point in K closest to s^{t+1}, also called *projection* of s^{t+1} on K. Pythagoras theorem implies monotonicity: $g(\theta^{t+1}) \leq g(\theta^t)$. In general, gradient descent started with arbitrary θ^0 is not guaranteed to reach the minimum, as is clear from the figure. It converges to a stationary point where $\nabla(f) = 0$, and at best we can hope this is a local optimum.

A well-behaved special case is when g is a *convex* function and K is a convex body, as is the case in (2). Then gradient descent does reach the global optimum

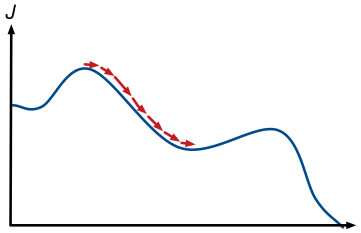

Figure 1: Gradient descent on a nonconvex function is not guaranteed to reach the global minimum.

if run long enough. Under modest conditions —e.g., a bound on the Lipschitz constant—it approaches the global optimum quite quickly. A comprehensive survey of such *convex optimization* procedures appears in [BV08].

But in general, problems (4), (5) are not convex and gradient descent can converge, at best, to a local optimum. A nonconvex problem may have multiple local optima, with some having lower objective values than others. So it is unclear which ones gradient descent ends up at. Nevertheless, in practice gradient descent works quite well: the solutions found are generally of good quality. Explaining why this happens is an important open problem. It is known that regularization can help, and a cottage industry of tricks has sprung up for regularizing the problem. Another important trick that helps is *stochastic gradient descent*, whereby one estimates the gradient of ERM objective 4 via a small sample of training samples: this improves the running time, and also seems to act as a regularizer.

2.2 Nonconvex models and deep learning

Clearly, the linear model studied above is simplistic. It associates a sentiment score with each word, and sums up the sentiment scores of the words in a review to get an idea of the numerical score. Thus the score only depends upon the multi-set of words in it and completely ignores linguistic structure: "Good, is it not?"gets the same score as "It is not good."Clearly, a fuller understanding of the text must involve more nuanced consideration of larger units such as phrases and sentences. One could try to hand-design features that the machine should pay attention to, e.g., those involving antonyms, synonyms etc. While these can help to some extent, empirically the best results are obtained by just letting the machine automatically figure out the features that it finds most useful. The most powerful current technique for doing this is to train a deep net. A thorough treatment of deep learning appears in the text [GBC16].

Deep net is a modern name for *neural net*, a notion from the 1940s. It is loosely inspired by the neurons of human brain, specifically the way they are interconnected via wiring that transmits electric signals and their mode of producing an output depending upon the sum of the incoming signals. A *deep net with d hidden layers* consists of d matrices $A_1, A_2, \ldots, A_d$, and a specific

function $\sigma\colon \Re \to \Re$ called the *nonlinearity*. The most popular nonlinearity σ these days is the *rectilinear linear* function $\text{RELU}_b(x) = \max\{0, x - b\}$. Here b is called the *bias*, and it is also a parameter of the network together with the A_i's. Defining $y^0 = x^0$ this net computes $y^1, y^2, \ldots, y^d$ where $y^{i+1} = \sigma(A_i y^i)$. Here $\sigma(z)$ denotes the vector obtained by applying σ to each coordinate of z. Also we are assuming that the dimensions of y^i's and A_i's match so that the matrix-vector products are well-defined. Each coordinate of a computed vector y^i is refered to as a *node* of the net, and each entry of one of the A_i's is refered to as an *edge*. The *output* of the net is y^d. The *size* of the net is the number of nodes in it. The number of parameters is the number of edges plus the number of nodes.

A deep net thus defines an input-output behavior, mapping the input vector x^0 to the output vector $y^d = f_{A_1, A_2, \ldots, A_d, \vec{b}}(x^0)$ where A_i's are the layer matrices and $\vec{b}$ is the vector of all bias values at the nodes. Thus this model can be used to do supervised learning, where the trainable parameters are the matrices and the biases. (An important subcase of a deep net is a *convolutional deep net* where the matrices A_i's have a specific compact representation whereby the same weight is reused in a fixed pattern across the input. These are easier to train in practice especially on data such as images which have patterns that are well-represented by such nets. We will ignore convolution in this survey.)

How does depth help in deep nets? While a net with a single hidden layer (i.e., depth 2) can in principle express any function computed by a net with more layers, doing so may come at a cost of requiring vastly more nodes [ES16, Tel16]. Training such a vast net would be computationally infeasible. Thus increasing depth allows a more succinct net to do interesting classification tasks.

To train d-layer deep nets for supervised learning using the above-mentioned *Empirical Risk Minimization* paradigm, we need to solve an optimization problem that solves for the matrices $A_1, A_2, \ldots, A_d$ and the bias vector $\vec{b}$. Writing out the expression for Empirical Risk we find it to be nonconvex in the variables. Nevertheless, we can plough ahead and try to solve it using some variant of gradient descent.

Backpropagation: To do gradient descent millions of times we need a quick way to compute the gradient of the objective. Since the final output is obtained by applying a composition of single layers, computing the gradient is a simple matter of applying the chain rule. Anybody who's taken freshman calculus can write this gradient. The tricky issue is to do so efficiently, meaning given the matrix entries and the bias values, to compute the gradient using as few basic operations —additions and multiplications— as possible. (An elementary operation like addition and multiplication is, simplistically speaking, a unit of effort for the computer's CPU.) Applying chain rule naively would require a number of operations that grows *quadratically* in the number of parameters. Since modern deep nets are often trained with tens of millions of parameters, quadratic in that number would be rather large even for today's computers. A clever algorithm called *backpropagation* can compute the gradient with number

of operations that is *linear* in the number of parameters. This is a crucial saving that enables deep learning to get off the ground, so to speak. An elementary exposition of backpropagation and its variants appears at [AM16].

Computational and statistical complexity. It can be shown that finding the optimum deep net is in general computationally intractable. However, this refers to computational complexity for unnatural, worst-case instances. Real-life instances are better behaved, and clearly good training is possible. Furthermore, there is evidence that *overparametrizing* the network with many more parameters than necessary can simplify the training. Consequently, today's deep nets are often trained with many more parameters than the number of training examples. *A priori* this raises fears that overparametrization would lead to lack of generalization but in practice generalization does not appear to suffer. Explaining why generalization happens is an open problem, unlike in the linear case described earlier.

What fueled deep learning's rise? While the basic ingredients of deep learning were known for several decades, a confluence of factors around 2011 led to its rapid progress and adoption. The first was availability of large labeled datasets. Datasets for training image recognition software used to be created in academia, and it was just not feasible for a small academic team to hand-label a very large number of images. Starting a decade ago, researchers could use crowdsourcing to create datasets containing millions of humanly-labeled images, such as ImageNet [DDS+09]. The second factor was availability of extremely fast *Graphical Processing Units* (GPUs) that brought the power of supercomputers to grad student desktops and fed a wave of experimentation that led to deep learning's resurgence. The third factor is developments in the theory of optimization for machine learning. The new generation of researchers understand notions such as regularization and acceleration and were able to employ them effectively —as well as design new ideas such as batch normalization, dropout, AdaGrad, Adam, etc.—to improve optimization —specifically, what things to try when training a large net fails initially.

Finally. enormous corporate interest in uses of deep learning leads to enormous research effort in industry as well.

3 Unsupervised learning

The techniques discussed thus far can train machines to do *classification tasks* where the output is a scalar (or small number of scalars) and there is plentiful training data that has been labeled by humans. But this captures only a small part of what we humans consider as learning. One suspects that a big part of our learning is *unsupervised*, whereby we passively observe the world around us and notice patterns in it. When we see a new animal or bird while visiting a new continent, we do not need to be told its name to already be able to describe

it, and relate it to animals we've seen in the past. Efforts to endow machines with such capabilities have not been as successful.

Viewed from a distance, all methods for unsupervised learning try to formalize a notion of "high level"descriptor of data. If the training datapoints are $x^1, x^2, \ldots,$, one assumes that each has an implicit (i.e., unknown) high level descriptor $h^1, h^2, \ldots$. To give an (advanced) example, x^i could be a pixel-level description of a photo of an unknown bird, and h^i could say in some form "white bird with long legs and long beak." Clearly, each h^i corresponds to multiple (even infinitely many) images and conversely even an image can have multiple high level descriptions. Methods for unsupervised learning allow for this possibility. They define some (possibly loose) way to go from x^i to h^i and vice versa. The following is a non-exhaustive list of ideas that have been tried for many years.

3.1 Dimension reduction of some sort

Dimension reduction amounts to finding low-dimensional vectors $y^1, y^2, \ldots,$ that capture the "essential properties"of $x^1, x^2, \ldots$. The simplest example is to try to approximate the distance: for all i, j the distance between y^i and y^j is approximately the same as between x^i and x^j.

Specific formulations include *Principal Component Analysis* (project to top k eigen-directions of $\sum_i x^i \otimes x^i$), *Manifold Learning* (assume there is an unknown low-dimensional manifold $\mathcal{M}$ such that each $x^i = h^i + noise$ where h^i is a point on the manifold), tSNE, etc.

3.2 Fitting a bayesian model to the data

This method assumes that there is a distribution $p_\theta(x, h)$ from which the sample x^i's were generated. Here θ is a vector of parameters that describe the distribution, and p_{theta} comes from a specific family of distributions. To give a simple example, a multivariate gaussian distribution is given by the density function $p_\Sigma(x, h) = \exp(\frac{(x-h)^T \Sigma (x-h)}{2})$ where h is the mean and Σ^{-1} is the covariance matrix. Hence we can think of x as h with some added noise.

Examples of bayesian models in unsupervised learning include *topic models*, *hidden markov models*, *mixed membership models*, *indian buffet process*, *hierarchichal topic models*, *Restricted Boltzmann Machines* etc.

There are two important problems associated with this approach to unsupervised learning. We assume that the machine is given independent samples $x^1, x^2, \ldots, x^N$ from the distribution $p_\theta(x) = \int p_\theta(x, h) dh$. (In words, "pick a sample (x, h) from $p_\theta(x, h)$, and discard the h.") It is customary to assume $p_\theta(x, h)$ factors as $p_\theta(x|h)p_\theta(h)$ where $p_\theta(h)$ has some simple functional form that is known. (Note that such a $p_\theta(h)$ always exists by Bayes' rule, but in general may not have a simple functional form.)

Parameter learning consists of estimating the best θ that explains the data. The method used is classical *maximum likelihood*: select the θ that assigns the maximum probability to the data. Since the data $x^1, x^2, \ldots, x^N$ were indepen-

dent samples from the distribution, this amounts to

$$\text{argmax}_\theta \prod_i p_\theta(x^i). \tag{8}$$

It is customary to take logarithms and re-express as

$$\text{argmax}_\theta \sum_i \log p_\theta(x^i), \tag{9}$$

which is the so-called *cross-entropy* loss.

Inference involves constructing h given x, where θ is assumed to be known. This involves sampling from the conditional distribution $p_\theta(h|x)$, which is given by Bayes rule.

While the problems are clear enough, the calculations are not easy. For fairly simple models, inference and parameter learning can be computationally intractable. It is customary to use heuristic approaches such as *Expectation Maximization* and *variational inference*. Recently there has been success in designing provably efficient algorithms for parameter learning via *tensor decomposition* methods; see [AGH$^+$14] for a comprehensive introduction.

3.3 Learning to generate portion of a datapoint from the rest

As mentioned, a full bayesian treatment of unsupervised learning runs into difficult computational problems that have not been easy to solve for large-scale problems. A more successful approach is to treat unsupervised learning more analogously to supervised learning, by observing that there is *implicit supervision* in the data itself.

Concretely, in many settings the datapoint x is much larger (i.e, has many more coordinates) than the latent h, which after all is a meant to be a high-level description. Thus h in principle could be inferred from (say) the first 3/4th of coordinates of x. And given h we could predict (at least in a probabilistic sense) the last 1/4th of the coordinates of x. This train of thought suggests that the last 1/4th coordinates of x can be predicted from its first 3/4th coordinates. Thus if we try to set ourselves the task of predicting the last 1/4th coordinates of x from its first 3/4th coordinates, implicitly we must need to learn the underlying structure, in other words, some version of h.

Concretely, if the input x is written as x_1x_2 where x_1 contains the first 3/4th of coordinates and x_2 the last 1/4th then such a learning approach assumes there is a mapping f_θ such that $f_\theta(x_1) \approx x_2$ where $\approx$ is formalized using some measure of closeness, e.g., ℓ_p norm. Here θ is a vector of parameters. For example, θ could describe a multilayer deep net that maps x_1 to x_2, and the deep net could be found via something like

$$\text{argmin}_\theta \sum_i |x_2^i - f_\theta(x_1^i)|_2^2. \tag{10}$$

This is very analogous to the Empirical Risk Minimization paradigm mentioned above.

Application: Word embeddings. How can we mathematically capture the meaning of an English word? From a mathematical viewpoint one is tempted to reach for mathematical notions such as model theory, which codifies semantics for formal logic. However, the meaning of a word is much more elusive. For one, the word may have multiple meanings (*bank* can refer to a financial institution or the side of a river), and each meaning may have many shades of meaning (is *paint* used in the same sense in *he painted the wall* and *he painted a mural on the wall*?)

In machine learning it has been more useful to represent the meaning of the word with a vector. This started with work in information retrieval ([TP10]) but recent techniques resort to the general idea sketched above. Specifically, it assumes that every word w is represented by a vector $v_w \in \Re^d$ for some d which is not too large or too small. (Depending upon the application, d is chosen to be a few hundred to a few thousand. There is no good theory explaining the choice.) Thus the model parameters θ consists of these vectors, one for every word in the English dictionary. The model is trained by assuming that if we black out a word in a text corpus, then we can typically figure out the missing word by looking at say 5 words to the left and to the right. For example in the famous **word2vec** method [MSC+13], the precise functional form assumed is

$$\Pr[w \mid w_1, w_2, \ldots, w_5] \propto \exp(v_w \cdot (\frac{1}{5}\sum_i v_{w_i})). \tag{11}$$

Training such a model requires some tricks, which we won't cover here. Note that the trained embeddings have fascinating properties. One of them is the ability to solve *word analogy* tasks. To solve the analogy problem *man : woman :: king : ??*, one tries to find the word w such that $v_w - v_{king}$ is most similar to $v_{woman} - v_{man}$, that is to say, minimizes $\|v_w - v_{king} + v_{woman} - v_{man}\|_2^2$. Among all $100,000$ words in the English dictionary, the minimizer word happens to be *queen*. This simple idea can solve many simple word analogies, though success rate is far from perfect. This and related discoveries have made word embeddings a useful tool in natural language processing. A theoretical explanation for the above method for analogy solving appears in [ALL+16].

3.4 Deep Generative Models

Deep nets, which were mentioned above, have also been used for unsupervised learning although the successes here are not as spectacular so far. A deep generative model G consists of a deep net that is defined completely analogously as before, which maps $\Re^d$ to $\Re^n$ for some d, n. It maps a random seed s, usually assumed to be a sample from the standard Gaussian distribution in $\Re^d$, to a vector x in $\Re^n$ that is supposed to be a random sample from the target

distribution that we are trying to learn. This model is trained using a set of samples from the target distribution $\mathcal{D}$ (for example, real-life images).

Thus the deep net implicitly defines a probability distribution $\mathcal{U}$, which we are trying to make close to $\mathcal{D}$. This technically is a subcase of the setting in Section 3.2, and the main idea in training is to do some form of gradient descent on the objective (9). Some notable notions in this line of work include Restricted Boltzman Machines [HS06], Variational Autoencoders [KW14], and Generative Adversarial Nets [GPAM$^+$15].

4 Reinforcement learning

Reinforcement learning concerns design of autonomous agents that take a sequence (potentially of unbounded length) of actions. For example, a self-driving car that has to take a dozens of actions every second, and maintain a safe course on the road. Such an agent may be trained a long time in various ways, but once trained has to be autonomous. Another setting where similar issues arise is in playing a complicated game like Chess or Go, where machines now outplay humans.

To formulate the goals of such learning, let's identify key aspects of such a system. (a) It needs to maintain some *state* at every time step, to allow it to store relevant information from previous steps (e.g., current speed, direction, separation from nearby vehicles) that will be needed in future steps. We denote the set of all possible states by S. (b) There is uncertainty in every measurement and action, which will be modeled via *probabilities*. (c) In each state the agent has the choice of some actions. Let A denote the set of possible actions. When the agent takes action $a \in A$ in a state, it makes a probabilistic transition to another state. (d) The agent moves from state to state as follows. Upon reaching a state, it takes an action, which causes it to transition probabilistically to another state, and in the process get some internal *reward*. This reward is its "internal motivation," so to speak. For example, reward function for a self-driving car may be a simple function of distances from the nearest vehicles in all four directions. The agent is trying to maximise this reward, as formalized later.

Similar frameworks have been well-studied in the past century in fields such as *control theory*, *finance*, *economic theory*, *operations research*, etc. In machine learning the above framework is called a *Markov Decision Process (MDP)*. As sketched above, it consists of the following components: a finite set of *states* S; a set of actions A (each action can be taken in each state); a *probabilistic transition function* that gives for each pair of states (s, s') and action a a probability $p(s, a, s')$ of transitioning to s' when action a is taken in state s (for all s, a it satisfies $\sum_{s'} p(s, a, s') = 1$); and a *reward function* that gives for each pair of states (s, s') and action a a reward $r(s, a, s')$ which is obtained when an action a is taken in state s followed by a transition to state s'.

The goal of the learner is to identify a policy π, which maps states to actions. Once an agent decides a policy $\pi\colon S \to A$, the MDP turns effectively into a

markov chain, where $p(s, \pi(s), s')$ is the probability of transitioning to s' at the next step if the agent is currently at state s. Thus if it is started in a state s_0, the agent's trajectory is a random sample from the distribution of random walks starting from s_0. It is customary to assume for convenience that this markov chain is ergodic. Thus if $s_0, s_1, s_2, \ldots,$ are random variables listing am infinite sequence of states that are visited during a random walk starting from s_0 then the expected reward is

$$E[\sum_{i=0}^{\infty} R(s_i, \pi(s_i), s_{i+1})].$$

In general this can be infinite, so it is customary to use a *discounting* whereby rewards obtained t steps into the future are treated as if they were multiplied by a factor γ^t where $\gamma < 1$ is the *discount factor*. Then total expected reward

$$E[\sum_{i=0}^{\infty} \gamma^i R(s_i, \pi(s_i), s_{i+1})]$$

stays finite. (The discounting idea is borrowed from economics, where this is a formalization of the familiar human instinct to treat *a bird in hand as better than two in the bush*.) The policy is optimum if this discount reward is optimum for every choice of s_0. The optimum policy can be computed using *dynamic programming* or *linear programming* in time that is a fixed polynomial of the number of states.

However, in practice today the set of states is often very large, or even infinite. For example, perhaps a state is a vector in $\Re^d$ and an action is a vector in $\Re^k$, which makes a policy a function from $\Re^d$ to $\Re^k$. Now there is no known efficient algorithm for finding an optimum policy, and in fact the task is known to be NP-hard. In practice, various heuristics are known such as *policy iteration* and *value interation*, where the policy being computed is represented implicitly via a suitable representation, often a deep net. Usually the machine does not know the underlying MDP and has to learn it while coming up with the policy. For a detailed introduction see [SB98]. Providing theoretical support for this heuristic work is an important open problem, since obvious ways to formalize it run into NP-hard problems. A start would be to formalize what it means for training to generalize here, since the above algorithms such as policy iteration do an *exploration* to progressively improve the policy, which takes us far afield from the independent sample framework utilized in our treatment of supervised learning in Section 1.

We note that the above framework can be changed in various ways to provide other well-studied frameworks that we will not describe here, such as *online computation*, *bandit optimization*, etc.. These capture less general types of sequential decision-making, which retain aspects of classical optimization by restricting attention to convex functions. For an introduction see [Haz].

Acknowledgements

Sanjeev Arora's work is supported by the NSF, ONR, Simons Foundation, Schmidt Foundation, SRC, Yahoo Research and Mozilla Foundation. Thanks to Mark Goresky, Avi Wigderson and Yi Zhang for useful feedback on the manuscript.

References

[AGH+14] Animashree Anandkumar, Rong Ge, Daniel Hsu, Sham M. Kakade, and Matus Telgarsky. Tensor decompositions for learning latent variable models. *Journal of Machine Learning Research*, 15:2773–2832, 2014. http://jmlr.org/papers/v15/anandkumar14b.html.

[ALL+16] Sanjeev Arora, Yuanzhi Li, Yingyu Liang, Tengyu Ma, and Andrej Risteski. A latent variable model approach to pmi-based word embeddings. *Trans. Assoc. Comp. Linguistics*, pages 385–399, 2016.

[AM16] Sanjeev Arora and Tengyu Ma. Backpropagation: An introduction, 2016. http://www.offconvex.org/2016/12/20/backprop/.

[BDSS14] Shai Ben-David and Shai Shalev-Schwartz. *Understanding Machine Learning: From Theory to Algorithms*. Cambridge University Press, 2014. http://www.cs.huji.ac.il/ shais/UnderstandingMachineLearning/.

[BV08] Stephen Boyd and Lieven Vandenberghe. *Convex Optimization*. Cambridge University Press, 2008. https://web.stanford.edu/ boyd/cvxbook/.

[DDS+09] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *Proc. IEEE CVPR*, 2009.

[ES16] Ronen Eldan and Ohad Shamir. Power of depth for feedforward neural networks. In *Proc. Conference on Learning Theory*, 2016.

[GBC16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[GPAM+15] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial networks. In *Proc. Neural Information Processing Systems*, 2015.

[Haz] Elad Hazan. Online convex optimization. http://ocobook.cs.princeton.edu.

[HS06] Geoff Hinton and Ruslan Salakhutdinov. Reducing the dimensionality of data with neural networks. *Science*, pages 504–507, 2006.

[HTF09] Trevor Hastie, Robert Tibshirani, and Robert Friedman. *The Elements of Statistical Learning*. Springer Verlag, 2009.

[KW14] Diederik Kingma and Max Welling. Auto-encoding variational bayes. In *Proc. International Conference on Learning Representations*, 2014.

[MSC+13] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In *Proc. Neural Information Processing Systems*, 2013.

[SB98] Richard Sutton and Arthur Barto. *Reinforcement Learning: An Introduction*. MIT Press, 1998.

[Tel16] Matus Telgarsky. Benefits of depth in neural networks. In *Proc. Conference on Learning Theory*, 2016.

[TP10] Peter Turney and Patrick Pantel. From frequency to meaning: Vector space models of semantics. *Journal of Artificial Intelligence Research*, 37:141–188, 2010.

[Vap98] Vladimir N. Vapnik. *Statistical Learning Theory*. Wiley, 1998.