Deep Thoughts on Deep Learning

Mark Stamp*
Department of Computer Science
San Jose State University

December 9, 2019

1 Introduction

Deep learning is often credited with having nearly metaphysical powers to solve challenging problems. Yet, the techniques behind deep learning are often treated as mysterious black boxes. In this tutorial, we attempt to provide a solid foundation for a deeper understanding of deep learning. Our primary emphasis is on backpropagation and automatic differentiation, but we also discuss a variety of related topics, including gradient descent, and various parameters that arise. In addition, we point out some of the many connections between deep learning and other not-so-deep techniques—primarily, hidden Markov models (HMM) and support vector machines (SVM). But first, we discuss artificial neural networks, which are the basic building blocks of deep learning.

2 A Brief History of ANNs

The concept of an artificial neuron [7, 19] is not new, as the idea was proposed by McCulloch and Pitts in the 1940s [11]. However, modern computational neural networking really begins with the perceptron, which was first proposed by Rosenblatt in the late 1950s [14].

An artificial neuron with three inputs is illustrated in Figure 1. In the original McCulloch-Pitts formulation, $X_i \in \{0,1\}$, $w_i \in \{+1,-1\}$, and the

*Email: mark.stamp@sjsu.edu. This lengthy tutorial is intended to serve as a supplement to my book, *Introduction to Machine Learning with Applications in Information Security* [17], which only includes shallow thoughts on the deep topics covered here.

output $Y \in \{0,1\}$. The threshold T determines whether the output Y is 0 (inactive) or 1 (active), based on $\sum w_i X_i$. The thinking was that a neuron either fires or it does not (thus, $Y \in \{0,1\}$), and the inputs would come from other neurons (thus, $X_i \in \{0,1\}$), while the weights w_i specify whether an input is excitatory (increasing the chance of the neuron firing) or inhibitory (decreasing the chance of the neuron firing). Whenever $\sum w_i X_i > T$, the excitatory response wins, and the neuron fires; otherwise the inhibitory response wins and the neuron does not fire.

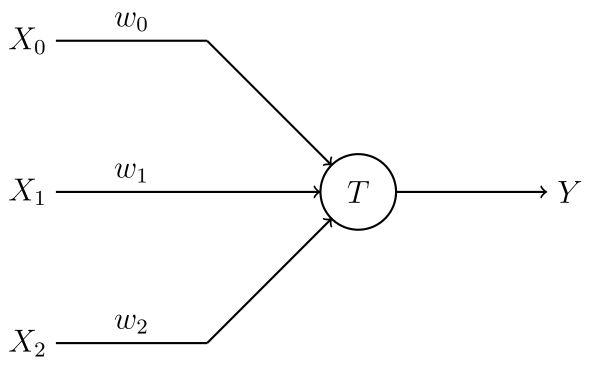


Figure 1: Artificial neuron

A *perceptron* is considerably less restrictive than a McCulloch-Pitts artificial neuron, as the X_i and w_i can be real valued. Since we want to use a perceptron as a binary classifier, the output is generally taken to be binary. McCulloch and Pitts chose such a restrictive formulation because they were trying to model logic functions. At the time, it was felt that encoding elementary logic into artificial neurons would be the key step to constructing systems with artificial intelligence. However, that point of view has certainly not panned out, while the additional generality offered by the perceptron formulation has proven extremely useful.

Given a real-valued input vector $X = (X_0, X_1, \ldots, X_{n-1})$, a perceptron can be viewed as a function of the form

$$f(X) = \sum_{i=0}^{n-1} w_i X_i + b,$$

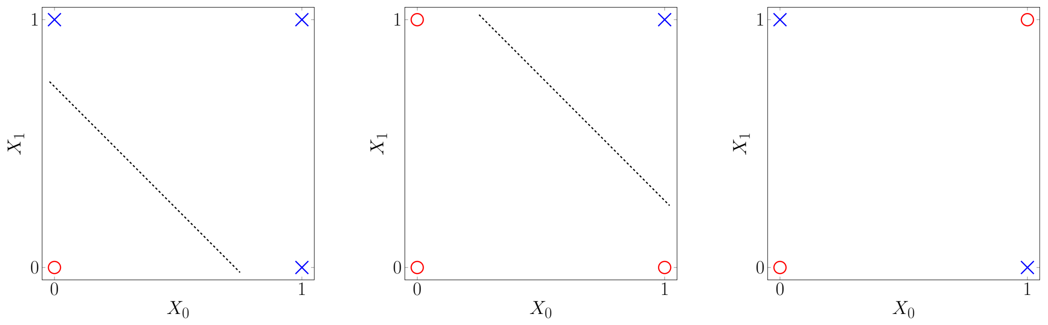
that is, a perceptron computes a weighted sum of the components. Based on a threshold, a perceptron can be used to define a binary classifier. For example, we could classify a sample X as "type 1" provided that $f(X) > T$, for some specified threshold T, and otherwise classify X as "type 0."

In the case of two dimensional input, the decision boundary of a perceptron defines a line

$$f(x,y) = w_0 x + w_1 y + b. \tag{1}$$

It follows that a perceptron cannot provide ideal separation in cases where the data itself is not linearly separable.

There was considerable research into artificial neural networks (ANN) in the 1950s and 1960s, and that era is often described as the first "golden age" of AI and neural networks. But the gold turned to lead in 1969 when an influential work by Minsky and Papert [12] emphasized the limitations of perceptrons. Specifically, they observed that the XOR function is not linearly separable, which implies that a single perceptron cannot model something as elementary as XOR. The OR, AND, and XOR functions are illustrated in Figure 2, where we see that OR and AND are linearly separable, while XOR is not.



(a) OR (b) AND (c) XOR

Figure 2: OR and AND are linearly separable but XOR is not

As the name suggests, a multilayer perceptron (MLP) is an ANN that includes multiple (hidden) layers in the form of perceptrons. An example of an MLP with two hidden layers is given in Figure 3, where each edge represent a weight that is to be determined. Unlike a single layer perceptron, MLPs are not restricted to linear decision boundaries, and hence an MLP can accurately model the XOR function. However, the perceptron training method proposed by Rosenblatt [14] cannot be used to effectively train an MLP [10]. To train a single perceptron, simple heuristics will suffice, assuming that the data is linearly separable. From a high level perspective, training a single perceptron is somewhat analogous to training a linear SVM, except that for a perceptron, we do not require that the margin (i.e., minimum separation) be maximized. However, training an MLP would appear to be challenging since we have hidden layers between the input and output, and it is not clear how changes to the weights in these hidden layers will affect each other, let alone the output.

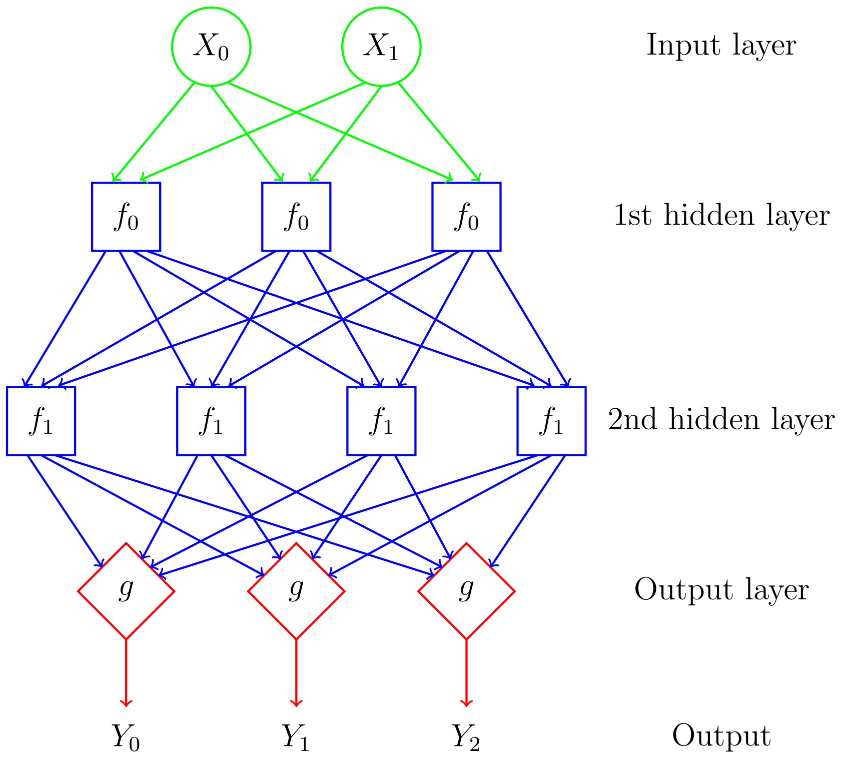


Figure 3: MLP with two hidden layers

As an aside, it is interesting to note that for SVMs, we deal with data that is not linearly separable by employing a soft margin (i.e., we allow for training errors) and by use of the so-called "kernel trick," where we map the input data to a higher dimensional feature space using a (nonlinear) kernel function. In contrast, perceptrons (in the form of MLPs) overcome the limitation of linear separability by the use of multiple layers. For an MLP, it is almost as if the nonlinear kernel function has been embedded directly into the model itself through the use of hidden layers, as opposed to a user-specified explicit kernel function, as is the case for an SVM.

One possible advantage of the MLP approach over an SVM is that for an MLP, the equivalent of the kernel function is, in effect, derived from the data and refined through the training process. In contrast, for an SVM, the kernel function is selected by a human, and once selected it does not change. In machine learning, removing those pesky humans from the learning process is a good thing. However, a possible tradeoff is that significantly more training data will likely be needed for an MLP, as compared to an SVM, due to the greater data requirement involved in learning the equivalent of a kernel function.

As another aside, we note that from a high level perspective, it is possible to view MLPs as combining some aspects of SVMs (i.e., specifically, nonlinear decision boundaries) and HMMs (i.e., hidden layers). Also, we'll see that the backpropagation algorithm that is used to train MLPs includes a forward pass and backward pass, which is eerily reminiscent of the training process that is used for HMMs.

As yet another aside, we note that an MLP is a *feedforward neural network*, which means that there are no loops—the input data and intermediate results feed directly through the network. In contrast, a recurrent neural network (RNN) can have loops, which gives an RNN a concept of memory, but can also add significant complexity.

In the book *Perceptrons: An Introduction to Computational Geometry*, published in 1969, Minsky and Papert [12] made much of the perceived shortcoming of perceptrons—in particular, the aforementioned inability to model XOR. This was widely viewed as a devastating criticism at the time, as it was believed that successful AI would need to capture basic principles of logic. Although it was known that perceptrons with multiple layers (i.e., MLPs) can model XOR, at the time, nobody knew how to efficiently train MLPs. Minsky and Papert's work was highly influential and is frequently blamed for the relative lack of interest in the field—a so-called "AI winter"—that persisted throughout the 1970s and into the early 1980s.

By 1986 there was renewed interest in ANNs, thanks in large part to the work of Rumelhart, Hinton, and Williams [15], who developed a practical means of training MLPs—the method of backpropagation. We discuss backpropagation in some detail in Section 6.

It is worth noting that there was another "AI winter" that lasted from the late 1980s through the early 1990s (at least). The proximate cause of this most recent AI winter was that the hype far outran the limited successes that had been achieved. Although deep learning has now brought ANNs back into vogue, your author (a doubting Thomas, and proud of it) is not convinced that the current artificial intelligence mania will prove any less artificial than previous AI "summers" which, on the whole, yielded mostly disappointment. Some of the ridiculous statements being made today [8] lead your eminently sensible author to believe that the hype is already hopelessly out of control.[1]

[1]In stark contrast to the nonsensical hype that envelopes far too much of the discussion of deep learning and (especially) AI, there does exist some clear-headed thinking that points to the great transformative potential of learning technology in the real world, rather than the world of science fiction. For a fine example of this latter genre, see the intriguingly-titled article, "Models will run the world" [5]. (Spoiler alert: "Models will run the world" is *not* about world domination by skinny women in swimsuits).

Next, we discuss deep learning, which builds on the foundation of ANNs. We can view the relationship between ANNs and deep learning as being somewhat akin to that of Markov chains and HMMs. That is, ANNs serve as a basic technology that can be used to build a powerful machine learning technique, analogous to the way that an HMM is built on the foundation of an elementary Markov chain. But, before we get into the details of deep learning, we consider the topic from a high-level perspective.

3 Why Deep Learning?

It is sometimes claimed that the major advantage of deep learning arises when the amount of training data is large. For example, the tutorial [9] gives a graph similar to that in Figure 4, which purports to show that deep learning will continue to achieve improved results as the size of the dataset grows, whereas other machine learning techniques will plateau at some relatively early point. That is, models generated by non-deep learning techniques will "saturate" relatively quickly, and once this saturation point is reached, more data will not yield improved models.[2] In contrast, deep learning is supposed to continue learning, essentially without limit as the volume of training data increases, or at least it will plateau at a much higher level. Of course, even if this is entirely true, there are practical computational constraints, since more data requires more computing power for training.

4 Decisions, Decisions

The essence of machine learning is that when training a model, we minimize the need for input from those fallible humans. That is, we want our machine learning models to be *data driven*, in the sense that the models learn as much as possible directly from the data itself, with minimal human intervention. However, any machine learning technique will require some human decisions—for HMMs we specify the number of hidden states, for SVMs we specify the kernel function, and so on.

For ANNs in general, and deep learning in particular, the following design decisions are relevant [6].

[2]If any learning model truly saturates, then adding more data will be counterproductive beyond some point, as the work factor for training on larger datasets increases, while there is no added benefit from the resulting trained model. It would therefore be useful to be able to predetermine a "score" of some sort that would tell us approximately how much data is optimal when training a particular learning model for a given type of data.

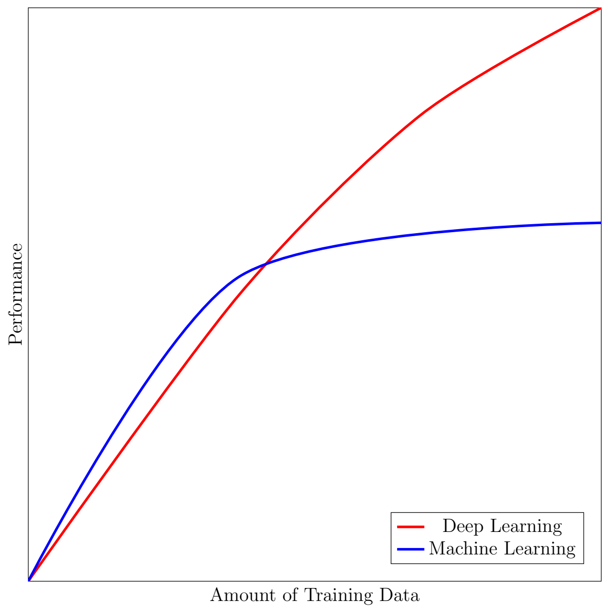


Figure 4: Model performance as a function of the amount of training data

- The *depth* of an ANN refers to the number of hidden layers. The “deep” in deep learning indicates that we employ ANNs with lots of hidden layers, where “lots” seems to generally mean as many as possible, based on available computing power.
- The *width* of an ANN is the number of neurons per layer, which need not be the same in each layer.
- In an MLP, for example, nonlinearity is necessary, and this is achieved through the *activation functions* (also known as transfer functions). Most activation functions used in deep learning are designed to mimic a step function—examples include the sigmoid (or logistic) function

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

  the hyperbolic tangent

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

  the inverse tangent (also known as arctangent)

  $$f(x) = \tan^{-1}(x),$$

and the rectified linear unit (ReLU)

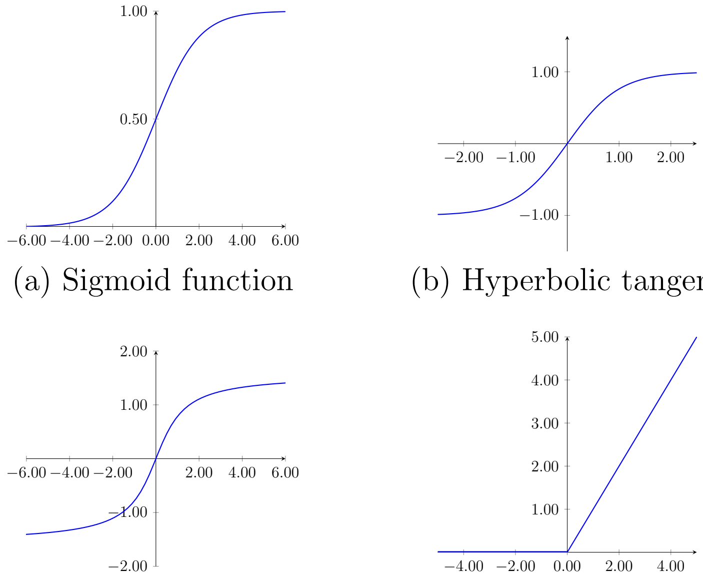
$$f(x) = \max\{0, x\} = \begin{cases} x & \text{if } x > 0 \\ 0 & \text{otherwise.} \end{cases}$$

Note that the softmax function is a generalization of the sigmoid function to multiclass problems.

The graph of each of the activation functions given above is illustrated in Figure 5. As of this writing, ReLU is the most popular activation function. Numerous variants of the ReLU function are also used, including the leaky ReLU and exponential linear unit (ELU).

- In addition to activation functions, we also specify an *objective function*. The objective function is the function that we are trying to optimize, and typically represents the training error.

- A *bias node* may be included (or not) in any hidden layer. Each bias node generates a constant value, and hence is not connected to any previous layer. When present, a bias node allows the activation function to be shifted. In the perceptron example given in (1), the bias corresponds to the y-intercept b.



(a) Sigmoid function (b) Hyperbolic tangent

(c) Arctangent (d) ReLU

Figure 5: Activation functions

For the sake of comparison with our favorite non-deep learning technique, the depth of an HMM can be viewed as the order of the underlying Markov model. Typically, for HMMs, we only consider models of order one (in which case, the current state depends only on the previous state), but it is possible to consider higher order models. The width of an HMM might be viewed as being determined by N, the number of hidden states. But, regardless of the order of the model or the choice of N, there is really only one hidden layer in any HMM. The fact that an HMM is based on linear operations implies that adding multiple hidden layers would have no effect, as the multiple layers would be equivalent to a single layer. Furthermore, the A and B matrices of an HMM can be viewed as its activation functions (with the B matrix corresponding to the output layer), and $P(\mathcal{O}\,|\,\lambda)$ corresponds to the objective function in an ANN. Note that these functions are all linear in an HMM, while at least some of the activation functions must be nonlinear in any true multilayer ANN, such as an MLP.

Neural networks are trained using the backpropagation algorithm, which is a special case of a more general technique known as reverse mode automatic differentiation. Next, we discuss automatic differentiation, and then turn our attention to the specific case of backpropagation.

5 Automatic Differentiation

First, let's recall the chain rule from calculus. Suppose that we have a composite function of the form

$$y = f(x) \ \text{ where } \ x = g(t).$$

Then, by the chain rule,

$$\frac{dy}{dt} = \frac{dy}{dx}\,\frac{dx}{dt}.$$

For a function of two or more variables, things are only slightly more complex. Consider a function of the form

$$z = f(x,y) \ \text{ where } \ x = g(t) \ \text{ and } \ y = h(t).$$

Then

$$\frac{dz}{dt} = \frac{\partial f}{\partial x}\,\frac{dx}{dt} + \frac{\partial f}{\partial y}\,\frac{dy}{dt},$$

that is, we differentiate with respect to each variable and sum the results. In the special case where $f(x,y)$ is of the form

$$z = f(x,y) \ \text{ where } \ y = g(x),$$

the chain rule still applies, and we find

$$\frac{dz}{dx} = \frac{\partial f}{\partial x}\frac{dx}{dx} + \frac{\partial f}{\partial y}\frac{dy}{dx} = \frac{\partial f}{\partial x} + \frac{\partial f}{\partial y}\frac{dy}{dx},$$

since $dx/dx = 1$. Of course, these concepts are easily extended to functions of any number of variables.

Now we consider the problem of computing derivatives efficiently. Suppose that we want to evaluate the derivative of a "reasonable" function $f(x)$ at some specific point x. One obvious way to do so comes directly from the definition of the derivative,

$$\lim_{h\to 0}\frac{f(x+h)-f(x)}{h}.$$

That is, we can simply evaluate

$$\frac{f(x+h)-f(x)}{h}$$

at some small value of h. The problem here is that if we make h very small, roundoff error will be a big problem, and if we don't make h sufficiently small, then the approximation is not likely to be good.

Instead of using the definition of the derivative, we can use symbolic computation to generate the derivative function $f'(x)$ and then evaluate this function at the desired point. This is essentially what you would typically do (assuming you are a human) if you were asked to evaluate $f'(x)$ in a calculus class.

It is possible to determine derivative functions algorithmically for reasonable functions (see, Mathematica, Maple, and Macsyma, for example). However, even if we leave aside the problems inherent in automatically generating the derivative function $f'(x)$, this approach would generally be computationally inefficient. One such efficiency issue is that components are often repeated in the derivative. For example, consider a function of the form $f(x) = g(x)/h(x)$. Then,

$$f'(x) = \frac{g'(x)h(x) - g(x)h'(x)}{h^2(x)}$$

and we see that $h(x)$ appears twice. This duplication issue only becomes worse when dealing with functions of more than one variable.

To be more concrete, let's define a "reasonable" function as one that is given in the form of a computer program. Any function that we can evaluate on a computer must be composed of fairly elementary operations, such as addition, subtraction, multiplication, division, polynomial functions, trigonometric functions, logarithms, exponentials, and so on. Since we're (hopefully) all computer geeks, the only functions we could possibly care about are those that can exist in the form of a computer program. Hence, we'll assume that any function $f(x)$ that we'll need to deal with is composed of simple operations (e.g., addition, subtraction, etc.). For any such function $f(x)$, we can evaluate the derivative $f'(x)$ at a specific point by recursively applying the chain rule to a set of simple sub-functions. This is the key insight behind automatic differentiation.

At this point, automatic differentiation might sound like symbolic differentiation, but it's not. In automatic differentiation, we are concerned with evaluating the derivative at a specified point, not obtaining an expression for the derivative function itself, which allows for automatic differentiation to be far more efficient in most cases. This should become clearer momentarily.

To see how automatic differentiation might work, let's walk through a simple example. Consider the function of two variables,

$$f(x,y) = \frac{x}{1+xy}. \tag{2}$$

For computational purposes, this function can be written in pseudo-code as in Figure 6, where x and y are initialized as desired, and z gives us $f(x,y)$ evaluated at the specified initial point (x,y).

```
1:  $v_0 = x$ // initialization
2:  $v_1 = y$ // initialization
3:  $v_2 = 1 + v_0 v_1$
4:  $v_3 = v_0 / v_2$
5:  $z = v_3$
```

Figure 6: Pseudo-code for the function $f(x,y)$ in (2)

Now, suppose that we want to evaluate the partial derivatives of $f(x,y)$, that is, we want to compute $\partial f/\partial x$ and $\partial f/\partial y$ at some specified point (x,y). By repeatedly applying the chain rule to the program version of the function $f(x,y)$, as given in Figure 6, we obtain the derivative pseudo-code in Figure 7.

For the derivative program in Figure 7, we must initialize x, y, dx, and dy, where (x,y) is the point at which we want to evaluate the partial derivative. But, how should we initialize dx and dy so as to obtain a partial derivative?

When we compute $\partial f/\partial x$, we treat x as a variable and y as a constant, which implies that for this partial derivative, we have $dx = 1$ and $dy = 0$. Thus, by initializing $(dx, dy) = (1,0)$, the value of dz in the last line of Figure 7 will give us $\partial f/\partial x$, evaluated at the specified point (x,y). For example, if we want to compute $\partial f/\partial x$ at the point $(x,y) = (3,2)$, we initialize both $(x,y) = (3,2)$

```
1: $v_0 = x$ // initialization
2: $v_1 = y$ // initialization
3: $dv_0 = dx$ // initialization
4: $dv_1 = dy$ // initialization
5: $dv_2 = v_1 dv_0 + v_0 dv_1$
6: $dv_3 = 1/v_2\, dv_0 - v_0/v_2^2\, dv_2$
7: $dz = dv_3$
```

Figure 7: Pseudo-code for partial derivatives of $f(x,y)$ in (2)

```
1: $v_0 = 3$ // initialization
2: $v_1 = 2$ // initialization
3: $dv_0 = 1$ // initialization
4: $dv_1 = 0$ // initialization
5: $dv_2 = 2$
6: $dv_3 = 1/7 - 6/7^2 = 1/49$
7: $dz = 1/49$
```

Figure 8: Evaluating $\partial f/\partial x$ at $(x,y) = (3,2)$

and $(dx, dy) = (1,0)$. Then from the derivative pseudo-code in Figure 7, we obtain $\partial f/\partial x = 1/49$, as illustrated in Figure 8.

For the function in (2), elementary calculus tells us that

$$\frac{\partial f}{\partial x} = \frac{1}{1+xy} - \frac{xy}{(1+xy)^2}.$$

It's easily verified that evaluating this function at $(x,y) = (3,2)$ agrees with the result obtained by evaluating our derivative program in Figure 8.

On the other hand, to evaluate $\partial f/\partial y$ at the point $(3,2)$, we initialize the derivative program with $(x,y) = (3,2)$ and $(dx,dy) = (0,1)$, in which case we obtain the result in Figure 9. Here, we find that the derivative program gives us $\partial f/\partial y = -9/49$. This result is also easily verified to be correct.

From the example above, we see that automatic differentiation provides a straightforward way to compute partial derivatives of any "programmable" function at a specified point. This is a very good thing indeed, but it's still far from optimal for the backpropagation problem.

For functions of more than one variable, the *gradient* plays the role of the derivative. In backpropagation, we will need to compute the gradient of a function of many variables. Our example function $f(x,y)$ in (2) has two variables,

```
1:  $v_0 = 3$ // initialization
2:  $v_1 = 2$ // initialization
3:  $dv_0 = 0$ // initialization
4:  $dv_1 = 1$ // initialization
5:  $dv_2 = 3$
6:  $dv_3 = -9/49$
7:  $dz = -9/49$
```

Figure 9: Evaluating $\partial f/\partial y$ at $(x, y) = (3, 2)$

so its gradient is of the form

$$\nabla f(x, y) = \left(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}\right).$$

As illustrated above, when using the code in Figure 7, we compute $\partial f/\partial x$ with the initialization $(dv_0, dv_1) = (1, 0)$, while $\partial f/\partial y$ requires the initialization $(dv_0, dv_1) = (0, 1)$. More generally, for a function of n variables, we have

$$\nabla f(x_0, x_1, \ldots, x_{n-1}) = \left(\frac{\partial f}{\partial x_0}, \frac{\partial f}{\partial x_1}, \ldots, \frac{\partial f}{\partial x_{n-1}}\right).$$

When using automatic differentiation (as discussed above), the analog of the derivative pseudo-code in Figure 7 is initialized with

$$(dv_0, dv_1, dv_2, \ldots, dv_{n-1}) = (1, 0, 0, \ldots, 0)$$

when evaluating $\partial f/\partial x_0$, while to compute $\partial f/\partial x_1$, we initialize

$$(dv_0, dv_1, dv_2, \ldots, dv_{n-1}) = (0, 1, 0, \ldots, 0),$$

and so on. Hence, to determine the gradient requires a total of n evaluations of the derivative code. But, we observe that the code is identical in each of these n iterations—only the initial values of the derivative variables dv_i change. This suggests that there should be a more efficient way to compute the gradient and, fortunately, this is indeed the case.

Above, we have described *forward mode automatic differentiation*. There is also *reverse mode automatic differentiation* [2] which, although somewhat less intuitive, will enable us to compute all of the partial derivatives required to evaluate the gradient of a function of any number of variables in an efficient single pass. We now outline this more efficient reverse mode, using the same example as above, namely, the function $f(x, y)$ in (2).

Consider again our program to compute $f(x, y)$, as given in Figure 6. First, let's compute derivatives involving the v_i. From line 3 in Figure 6, we have

$$\frac{dv_2}{dv_0} = v_1 \text{ and } \frac{dv_2}{dv_1} = v_0,$$

while line 4 implies that

$$\frac{dv_3}{dv_0} = \frac{1}{v_2} \text{ and } \frac{dv_3}{dv_2} = -\frac{v_0}{v_2^2}$$

Next, we'll apply the chain rule and make use of the results in the previous paragraph. We begin with the trivial observation

$$\frac{dz}{dz} = 1,$$

that is, the rate of change of z with respect to itself is 1. Then from line 5 in Figure 6 we have the equally trivial result

$$\frac{dz}{dv_3} = 1,$$

which follows from the fact that $z = v_3$. Next, we find that

$$\frac{dz}{dv_2} = \frac{dz}{dv_3}\frac{dv_3}{dv_2} = -\frac{v_0}{v_2^2}\frac{dz}{dv_3},$$

and

$$\frac{dz}{dv_1} = \frac{dz}{dv_2}\frac{dv_2}{dv_1} = v_0\frac{dz}{dv_2},$$

while

$$\frac{dz}{dv_0} = \frac{dz}{dv_3}\frac{dv_3}{dv_0} + \frac{dz}{dv_2}\frac{dv_2}{dv_0} = \frac{1}{v_2}\frac{dz}{dv_3} + v_1\frac{dz}{dv_2}.$$

In Figure 10, we have summarized reverse mode automatic differentiation for our example function (2). Here, we have shorthanded dz/dz as dz and dz/dv_i as dv_i, for $i = 0, 1, 2, 3$.

Note that dv_0 in Figure 10 gives us $\partial f/\partial x$, while dv_1 is $\partial f/\partial y$, both evaluated at (x, y). Consequently, we can obtain the values of both partial derivatives, evaluated at a specified point (x, y), in a single pass through the pseudo-code for $f(x, y)$ in Figure 6 (to determine the v_i), followed by a single pass through the reverse mode differentiation pseudo-code given in Figure 10 (which determines the dv_i that represent the partial derivatives).

```
1:  $dz = 1$
2:  $dv_3 = dz$
3:  $dv_2 = -v_0/v_2^2\, dv_3$
4:  $dv_1 = v_0\, dv_2$
5:  $dv_0 = 1/v_2\, dv_3 + v_1 dv_2$
```

Figure 10: Reverse mode automatic differentiation for $f(x, y)$ in (2)

In Figure 11 we give the results of reverse mode automatic differentiation for the function $f(x, y)$ in (2), that is,

$$f(x, y) = \frac{x}{1 + xy}.$$

Here, the forward pass is computed using the pseudo-code in Figure 6 while the backward pass is computed using the code in Figure 10. From Figure 11, we see that

$$\nabla f(3, 2) = \left(\frac{\partial f}{\partial x}(3, 2), \frac{\partial f}{\partial y}(3, 2)\right) = \left(\frac{1}{49}, \frac{-9}{49}\right),$$

which agrees with the partial derivative computations in Figures 8 and 9, and is also easily verified by directly computing $\partial f/\partial x$ and $\partial f/\partial y$.

Forward pass

```
1:  $v_0 = 3$ // initialization
2:  $v_1 = 2$ // initialization
3:  $v_2 = 1 + v_0 v_1 = 7$
4:  $v_3 = v_0/v_2 = 3/7$
5:  $z = v_3 = 3/7$
```

Backward pass

```
1:  $dz = 1$
2:  $dv_3 = dz = 1$
3:  $dv_2 = -v_0/v_2^2\, dv_3$
        $= -3/49 \cdot 1 = -3/49$
4:  $dv_1 = v_0\, dv_2$
        $= 3 \cdot (-3/49) = -9/49$
5:  $dv_0 = 1/v_2\, dv_3 + v_1 dv_2$
        $= 1/7 \cdot 1 + 2 \cdot (-3/49) = 1/49$
```

Figure 11: Evaluating gradient of $f(x, y) = \dfrac{x}{1 + xy}$ at $(3, 2)$

Reverse mode automatic differentiation gives us an extremely efficient means to obtain the entire gradient, evaluated at a specified point—regardless of the number of independent variables, we simply need to execute the forward pass, followed by the backward pass. Thus, reverse mode automatic differentiation is much more efficient for computing the gradient, as compared to other methods, including the forward mode of automatic differentiation discussed above.[3]

In reverse mode automatic differentiation we have, in effect, swapped the roles of dependent and independent variables, and then applied the chain rule as usual. In the particular example considered above, we only have one dependent variable, so there is no choice in the initialization of the "partial" derivative dz/dz. However, if we have, say, two dependent variables, then we would have a similar situation as in the forward mode example above, that is, we would need to execute the reverse mode code twice to obtain the partial derivatives for both dependent variables. It follows that forward mode is more efficient when there are fewer independent variables, while reverse mode is more efficient when there are fewer dependent variables. In backpropagation, we'll have a single dependent variable (based on a cost or error function), but a large number of independent variables (representing the weights), so reverse mode is the clear winner.

6 Backpropagation

Backpropagation can be viewed as a special case of reverse mode automatic differentiation. In backpropagation, we have a forward pass, a backward pass, and we use the results of these two passes to recompute the weights of the neural network under consideration. The forward pass corresponds to simply evaluating the program that defines the objective function at a specified point. When training a neural network, the objective function that we are concerned with will typically measure the distance between the computed values (using our current estimates for the weights) and the desired (known) training values. That is, in backpropagation, we are typically dealing with an error function that is based on a distance measure. The backward pass consists of evaluating the gradient of the objective function using reverse mode automatic differentiation. Then based on the computed gradient, we'll use a steepest descent algorithm to re-estimate the weights of the ANN. This process is iterated until we reach a (local) minimum of the objective function.

[3]For more complex functions, the savings inherent in automatic differentiation are even more significant, as will become clear when you solve the homework problems.[4]

[4]You *are* solving the homework problems, aren't you?

We'll look at backpropagation in more detail based on a simpler example below. But first we need to say a little more about gradient descent. A detailed example is given in Appendix A, where we show that an HMM can be trained using gradient ascent, as opposed to the well-known Baum-Welch re-estimation hill-climb technique.

6.1 Gradient Descent

Suppose that we want to find a local minimum of the function

$$f(x) = x^6 + 2x^5 - \frac{33}{4}x^4 - 14x^3 + \frac{53}{4}x^2 + 27x + 9.$$

The graph of this function appears in Figure 12.

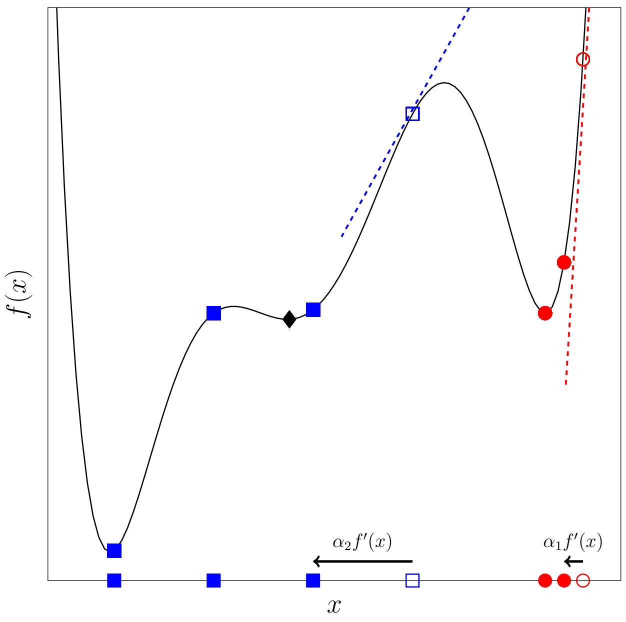


Figure 12: Two examples of gradient descent

In Figure 12, the red and blue dashed lines are the tangent lines to the curve at the red open circle and the blue open square, respectively. For any x_0, the derivative $f'(x_0)$ will give us the slope of the tangent line to f at the point x_0. Once we know the slope of the tangent line, by moving a small step in the opposite direction of this slope, we will obtain a smaller (or equal) value for the function f, and hence we can converge towards a local minimum. More precisely, in gradient descent we select an initial x_0, then compute $f'(x_0)$ and update x_0 to x_1 according to

$$x_1 = x_0 - \alpha f'(x_0),$$

where α is "small." Provided that α is sufficiently small, $f(x_1) \leq f(x_0)$, and we can iterate this process to determine a series of x_i that converge to a local minimum.

In Figure 12 we illustrate gradient descent for two different starting points—the red open circle and the blue open square—and two different step sizes, α_1 and α_2, respectively.[5] Note that in the case of the blue squares in Figure 12, we bypass a local minimum (marked with a black diamond). This shows that gradient descent is not a hill climb algorithm.

When training a neural network, the gradient descent step size is referred to as the *learning rate*, and this is a critical parameter. The significance of the learning rate can be seen in Figure 12, where a smaller value for α_2 would likely yield a different result. Note that, the learning rate need not be constant throughout the iterative training process.[6]

In a neural network application, computing the error requires that the error function be evaluated at all of the training data points. Since there may be a large number of steps in the gradient descent, and an extremely large number of training samples, this could be very costly. So, instead of evaluating the gradient at all of the training samples, we can evaluate the gradient based on a subset of the training samples at each iteration. This modification is often referred to as stochastic gradient descent (SGD),[7] but is more properly known as a *mini-batch* technique. Since we are not using all samples, there is no guarantee that we will descend at each step, but it can be shown that on average, we will move in the right direction [3]. For example, a mini-batch based gradient descent for the function in Figure 12 might follow a trajectory something like the red squiggly line in Figure 13.

Along with the learning rate, the batch size is an important parameter when training a neural network. And, as with the learning rate, the batch size need not be the same for each iteration.

We note in passing that mini-batches in backpropagation are somewhat analogous to the concept of bagging as used when training a random forest [17].

[5]The gradient descent examples in Figure 12 are oversimplified. In gradient descent, the increment $\alpha f'(x)$ varies depending on the value of the derivative, while the picture in Figure 12 shows constant increments.

[6]Intuitively, we'd probably want to set the learning rate relatively large initially, and then reduced it in later iterations as the model is closer to converging.

[7]Technically, in stochastic gradient descent, we would use only a single training sample at each iteration. For most applications of backpropagation to training a neural network, this would result in an extremely large number of iterations.

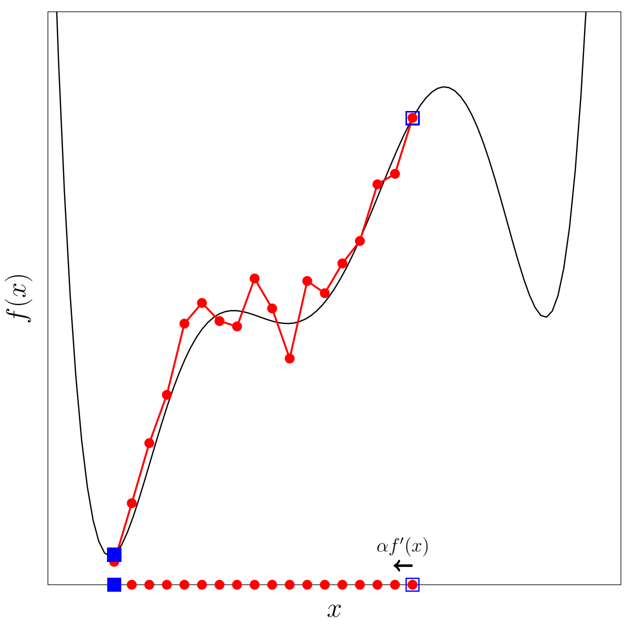


Figure 13: Mini-batch gradient descent

Bagging consists of selecting subsets of the data and features to construct the decision trees that form the random forest. Although mini-batches are used in backpropagation primarily for the sake of efficiency, mini-batches can yield a better result, since the "noisy" intermediate steps enable a model to bypass an inferior local minimum. Consequently, mini-batches can give us a more useful neural network, while bagging allows a random forest to overcome the overfitting that plagues decision trees.

The use of backpropagation to train an ANN also has some striking similarities to the sequential minimal optimization (SMO) algorithm [17]. The SMO algorithm—which efficiently solves large and sparse quadratic programming problems—is the most effectual method known for training an SVM. For the sake of efficiency, SMO relies on a gradient descent, with minimal "batches" of the Lagrangian coefficients selected at each iteration. The use of minimal batches and gradient descent in SMO are very much analogous to SGD in the context of backpropagation.

In the next section, we provide a straightforward neural network example and give some details on the use of backpropagation to train this particular network.[8] Specifically, we consider the problem of training a 2-layer MLP based on a slightly generalized version of the XOR function.

[8]For another example, see Appendix B, where we show that an HMM can be trained using Lagrange multipliers and backpropagation.

6.2 Simple MLP Example

As discussed in Section 2, the XOR function cannot be accurately modeled by a single layer perceptron. However, an MLP with two layers, such as that illustrated in Figure 14, can represent XOR. See homework Problem 2 at the end of this tutorial for more details on how to design a 2-layer MLP that is equivalent to the XOR function.

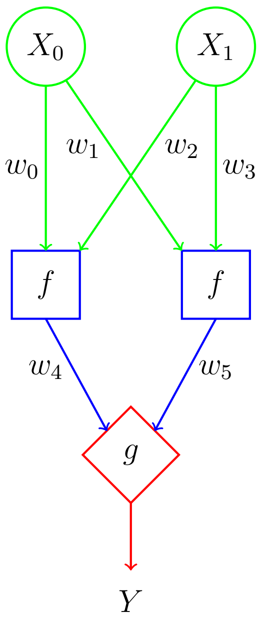


Figure 14: Simple MLP example

Our goal here is to train the MLP in Figure 14 using backpropagation. We'll choose f to be the logistic function

$$f(s,t) = \frac{1}{1+e^{-(s+t)}}$$

and, to keep it simple, we'll let $g(s,t) = s+t$. Then the function defined by this particular MLP is given by

$$\begin{aligned} Y &= w_4\, f(w_0X_0, w_2X_1) + w_5\, f(w_1X_0, w_3X_1) \\ &= \frac{w_4}{1+e^{-(w_0X_0+w_2X_1)}} + \frac{w_5}{1+e^{-(w_1X_0+w_3X_1)}}. \end{aligned} \tag{3}$$

We'll attempt to train the this MLP to model

$$F(X_0, X_1) = \text{XOR}(\lfloor X_0 + 0.5 \rfloor, \lfloor X_1 + 0.5 \rfloor), \tag{4}$$

which is a generalized form of the XOR function, defined for $0 \leq X_i < 1$. Note that in this function, we simply apply the usual rules of rounding to X_0 and X_1, and then apply the standard XOR function to these rounded values. The decision boundaries for the function in (4) are given in Figure 15.

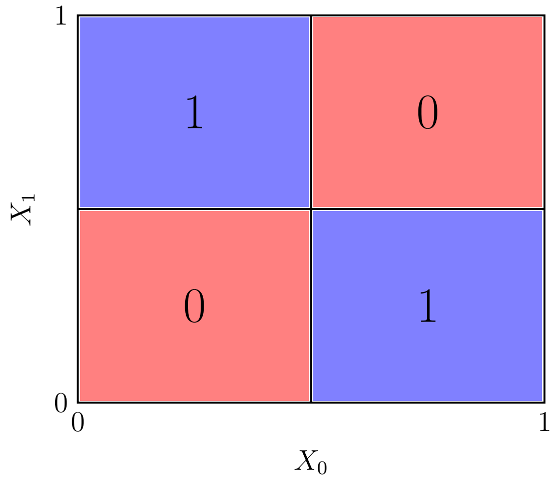


Figure 15: Decision boundaries for $F(X_0, X_1)$ in (4)

For the MLP specified by (14), the output Y is a real number. Since the output of the generalized XOR function in (4) that we are trying to model is either 0 or 1, we can map the result of the MLP to 0 or 1, depending on whether Y is closer to 0 or 1, that is

$$\text{MLP}(X_0, X_1) = \begin{cases} 0 & \text{if } Y < 1/2 \\ 1 & \text{otherwise.} \end{cases}$$

Again, we want to train this MLP to recognize the function (4). Each training sample will consist of a pair (X_0, X_1) and Z, where $0 \leq X_i < 1$ and where Z is computed using the generalized XOR function in (4).

We'll measure the error using one-half of the squared Euclidean distance. Then for any given set of weights $(w_0, w_1, \ldots, w_5)$, each training sample gives us an error term of the form

$$E(w) = \frac{1}{2}\Big(\frac{w_4}{1 + e^{-(w_0 X_0 + w_2 X_1)}} + \frac{w_5}{1 + e^{-(w_1 X_0 + w_3 X_1)}} - Z\Big)^2. \qquad (5)$$

It follows that the overall error when approximating the function in (4) by the MLP in Figure 14 is the sum over our training samples of these error terms. For simplicity, here we'll just consider the error for a single sample—it is straightforward to extend this analysis to the error for any number of training samples. Pseudo-code to evaluate the error term $E(w)$ is given in Figure 16.

```
1:  $v_0 = w_0$ // initialization
2:  $v_1 = w_1$ // initialization
3:  $v_2 = w_2$ // initialization
4:  $v_3 = w_3$ // initialization
5:  $v_4 = w_4$ // initialization
6:  $v_5 = w_5$ // initialization
7:  $v_6 = X_0 v_0 + X_1 v_2$
8:  $v_7 = X_0 v_1 + X_1 v_3$
9:  $v_8 = 1 + e^{-v_6}$
10: $v_9 = 1 + e^{-v_7}$
11: $v_{10} = v_4 / v_8$
12: $v_{11} = v_5 / v_9$
13: $v_{12} = (v_{10} + v_{11} - Z)^2 / 2$
14: $z = v_{12}$
```

Figure 16: Pseudo-code to compute the error $E(w)$ in (5)

Next, we follow the reverse mode automatic differentiation procedure outlined in Section 5. From line 7 in Figure 16 we have

$$\frac{dv_6}{dv_0} = X_0 \quad \text{and} \quad \frac{dv_6}{dv_2} = X_1,$$

while line 8 yields

$$\frac{dv_7}{dv_1} = X_0 \quad \text{and} \quad \frac{dv_7}{dv_3} = X_1,$$

and line 9 gives us

$$\frac{dv_8}{dv_6} = -e^{-v_6},$$

and so on. From these derivative expressions and the trivial observations

$$\frac{dz}{dz} = 1 \quad \text{and} \quad \frac{dz}{dv_{12}} = 1,$$

we obtain the results in Figure 17, where we use dv_i as shorthand for dz/dv_i. It follows that if we initialize

$$(v_0, v_1, \ldots, v_5) = (w_0, w_1, \ldots, w_5)$$

and compute the remaining v_i as in Figure 16, the gradient of the error function $E(w)$ is given by the final six lines in Figure 17, that is,

$$\nabla E(w) = \big(dv_0, dv_1, dv_2, dv_3, dv_4, dv_5\big).$$

```
1:  $dz = 1$
2:  $dv_{11} = v_{10} + v_{11} - Z$
3:  $dv_{10} = v_{10} + v_{11} - Z$
4:  $dv_9 = -v_5/v_9^2\,dv_{11}$
5:  $dv_8 = -v_4/v_8^2\,dv_{10}$
6:  $dv_7 = -e^{-v_7}dv_9$
7:  $dv_6 = -e^{-v_6}dv_8$
8:  $dv_5 = dv_{11}/v_9$
9:  $dv_4 = dv_{10}/v_8$
10: $dv_3 = X_1\,dv_7$
11: $dv_2 = X_1\,dv_6$
12: $dv_1 = X_0\,dv_7$
13: $dv_0 = X_0\,dv_6$
```

Figure 17: Reverse mode automatic differentiation for $E(w)$ in (5)

To apply the backpropagation algorithm, we first select initial values for the weights w_i of the MLP in Figure 14, and we use the code in Figure 16 to compute the v_i. Then we use the code in Figure 17 to compute the gradient. Based on the gradient and learning rate α, we generate updated weights $\widetilde{w}_i$ as

$$\widetilde{w}_i = w_i - \alpha\,dv_i \quad\text{for}\quad i = 0, 1. \ldots, 5.$$

This process is iterated until we reach a (local) minimum of the error function.

In the backpropagation algorithm, computing the v_i as in Figure 16 is referred to as the *forward pass*, while the code in Figure 17 is the *backward pass*. These two passes are used to compute the gradient of the error function.

We note that in terms of the v_i in Figure 16, the gradient of $E(w)$ is

$$\begin{aligned}\nabla E(w) &= \Bigl(\frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \frac{\partial E}{\partial w_2}, \frac{\partial E}{\partial w_3}, \frac{\partial E}{\partial w_4}, \frac{\partial E}{\partial w_5}\Bigr) \\ &= \bigl(X_0(e^{-v_6})(v_4/v_8^2)(v_{10} + v_{11} - Z), \\ &\qquad X_0(e^{-v_7})(v_5/v_9^2)(v_{10} + v_{11} - Z), \\ &\qquad X_1(e^{-v_6})(v_4/v_8^2)(v_{10} + v_{11} - Z), \\ &\qquad X_1(e^{-v_7})(v_5/v_9^2)(v_{10} + v_{11} - Z), \\ &\qquad (v_{10} + v_{11} - Z)/v_8, \\ &\qquad (v_{10} + v_{11} - Z)/v_9\bigr)\end{aligned}$$

However, for the backpropagation algorithm, we do not expand the partial derivatives in this way, as the backward pass provides an efficient means of computing the required partial derivatives.

Again, for the example considered here, the error function is based on only one sample. Typically, we would use a mini-batch, where the error function depends on multiple samples at each step. It is straightforward to derive the error function and gradient for the case where the error term depends on any number of training samples.

7 Conclusion

The main focus of this paper is the backpropagation algorithm. There are many additional sources of information, including the aforementioned [6] and [2]. The original source [15] is well worth reading.

In [15] it is mentioned that

8 Problems

1. For the function in (2), pseudo-code for the forward pass and backward pass of the backpropagation algorithm is given in Figures 6 and 10, respectively. Give analogous pseudo-code for the forward and backward passes for the function
$$f(x,y) = \frac{x}{1+e^{-(x+y)}}.$$

2. Consider the MLP in Figure 14, which has two inputs and one hidden layer consisting of two nodes. Suppose that we choose the function in this MLP to be
$$f(x_0,x_1) = \max\{x_0+x_1,0\} \tag{6}$$
and we let
$$g(s,t) = s+t.$$
The function $f(x_0,x_1)$ in (6) is the rectified linear unit (ReLU), which is discussed in Section 4.

    a) Give the function specified by this MLP, that is, write the output Y in terms of the inputs X_i and weights w_i.

    b) Suppose that we restrict $X_i \in \{0,1\}$ for $i=0,1$ and we also require that $Y \in \{0,1\}$ and that the weights satisfy $w_i \in \{+1,-1\}$ for $i=0,1,\ldots,5$. Under these conditions, determine weights so that the resulting MLP represents the XOR function. Construct a truth table to verify that the output Y agrees with the XOR function.

c) Since XOR is not linearly separable, your solution to part b) must not be a separating hyperplane.[9] Shade the unit square according to whether the points are classified as 0 or 1 based on your solution to part b), assuming that we classify (X_0, X_1) as 0 whenever $Y < 0.5$ and we classify (X_0, X_1) as 1 whenever $Y \geq 0.5$. Note that the unit square has corners at $(0,0)$, $(0,1)$, $(1,0)$, and $(1,1)$, and we are restricting the input (X_0, X_1) to be within this region—but we do not require X_i to be 0 or 1, as we did in part b).

3. Draw a diagram analogous to that in Figure 20, but for the case where the HMM has $N = 3$ hidden states and there are $T = 3$ observations.

4. In this problem, we show that we must have nonlinear functions in an MLP. Consider the simple MLP in Figure 14 and let $f(x_0, x_1) = ax_0 + bx_1$ and $g(s,t) = s+t$. Show that this is equivalent to a single layer perceptron, and give the function that defines the perceptron.

5. Write line 9 in Figure 17 in terms of $w_0, w_1, \ldots, w_5$ using the values of the v_i as computed Figure 16. Verify that this line gives us $\partial E / \partial w_5$.

6. In this problem, you will use the following data to train the MLP in Figure 14 to recognize the XOR function.

i	X_0	X_1	Z_i
0	0.6	0.4	1
1	0.1	0.2	0
2	0.8	0.6	0
3	0.3	0.7	1
4	0.7	0.3	1
5	0.7	0.7	0
6	0.2	0.9	1

a) Use the stochastic gradient descent (SGD) algorithm, with the pseudo-code in Figure 16 for the forward pass, and the pseudo-code in Figure 17 for the backward pass. Initialize the weights as

$$(w_0, w_1, \ldots, w_5) = (1, 2, -1, 1, -2, 1),$$

use a constant learning rate of $\alpha = 0.1$, and in each epoch[10] consider the training samples in order $i = 0, 1, \ldots, 6$. Train for 1000 epochs. Give

[9] Recall that a hyperplane in 2-dimensional space is a line.
[10] An *epoch* is one complete pass through the training data.

the final weights $(w_0, w_1, \ldots, w_5)$, compute the result of your trained MLP on each training sample, and determine the accuracy on the training set. Also, determine the accuracy of your MLP on the following test data.

i	X_0	X_1	Z_i	i	X_0	X_1	Z_i
0	0.55	0.11	1	5	0.46	0.54	1
1	0.32	0.21	0	6	0.16	0.51	1
2	0.24	0.64	1	7	0.52	0.94	0
3	0.86	0.68	0	8	0.46	0.87	1
4	0.53	0.79	0	9	0.96	0.63	0

b) Repeat part a), but use 10,000 epochs. Compare your trained MLP from part a) to your trained MLP from this part. In particular, compute the output of the function (3) on each training sample based on the trained model in a), and do the same using the model obtained in this part. Comment on these results.

7. Repeat Problem 6, but include a bias node (with bias 1) in the hidden layer. Note that this will change the objective function, and hence the forward and backward passes will need to be modified.

8. Generate pseudo-code for the backpropagation forward pass and backward pass analogous to that in Figures 16 and 17, respectively, but for the case where the error is based on a mini-batch consisting of two training samples rather than a single training sample.

9.* Baum-Welch re-estimation, which is the standard method of training an HMM, is a hill climb technique [17]. In Appendix B, we showed that HMM training can be based on a Lagrange neural network. Since we can train a neural network using backpropagation, which is not a hill climb, it follows that if we train an HMM as outlined in Appendix B, the training is not a hill climb. Thus, the neural network approach to training an HMM might be advantageous, since Baum-Welch will get stuck at the first local maximum that it encounters.

Using a Lagrange neural network, train HMMs on English text, with 27 symbols (lowercase letters a through z and word-space) with $N = 2$ hidden states and $T = 50{,}000$ observations. Use SGD (i.e., a mini-batch of size 1) and experiment with different choices for the learning rate α. In each case, train for a sufficient number of epochs so that the model converges. Compare your results to the example in Section 9.2 of [17], which also appears in Section 8 of [16].

10.* The article [1] and Appendix A discuss an alternative method for training an HMM based on gradient ascent. One advantage of this gradient based approach, as compared to Baum-Welch re-estimation, is that it can be applied in an "online" mode, that is, the model can be updated as more data is available, without retraining from scratch. In this problem, you will implement and test the algorithm in Appendix A.

In this algorithm, we'll update the the elements of the A and B matrices using a softmax technique, that is,

$$a_{ij} = \frac{e^{\tau w_{ij}}}{\sum\limits_{k=0}^{N-1} e^{\tau w_{ik}}} \quad\text{and}\quad b_i(j) = \frac{e^{\tau v_{ij}}}{\sum\limits_{k=0}^{M-1} e^{\tau v_{ik}}},$$

where τ is a "temperature" parameter. We update w_{ij} and v_{ij} in this equation as

$$\widetilde{w}_{ij} = w_{ij} + \frac{\alpha}{\mathcal{C}}\bigl(\mathcal{A}_{ij} - \mathcal{A}_i a_{ij}\bigr)$$

and

$$\widetilde{v}_{ij} = v_{ij} + \frac{\alpha}{\mathcal{C}}\bigl(\mathcal{B}_{ij} - \mathcal{B}_i b_i(j)\bigr),$$

where α is the learning rate, $\mathcal{C}$ is a function of $P(\mathcal{O} \mid \lambda)$, $\mathcal{A}_{ij}$ is the expected number of transitions from state i to state j, $\mathcal{B}_{ij}$ is the expected number of times we observe j in state i, and

$$\mathcal{A}_i = \sum_j \mathcal{A}_{ij} \quad\text{and}\quad \mathcal{B}_i = \sum_j \mathcal{B}_{ij}.$$

Each of $\mathcal{A}_{ij}$, $\mathcal{A}_i$, $\mathcal{B}_{ij}$, $\mathcal{B}_i$, and $\mathcal{C}$ depend on the observation sequence $\mathcal{O}$, and all are easily computed.[11]

a) Use the algorithm outlined in this problem to train HMMs on English text, with 27 symbols (lowercase letters a through z and word-space), using $N = 2$ hidden states and $T = 50{,}000$ observations. Experiment with various choices for the temperature parameter τ and learning rate α. Compare your results to the example in Section 9.2 of [17], which also appears in Section 8 of [16].

[11] In terms of $\gamma_t(i,j)$, $\gamma_t(i)$ and c_t defined in [16] and Chapter 2 of [17], we have the following: $\mathcal{A}_{ij} = \sum\limits_{t=0}^{T-2} \gamma_t(i,j)$, $\mathcal{A}_i = \sum\limits_{t=0}^{T-2} \gamma_t(i)$, $\mathcal{B}_{ij} = \sum\limits_{\substack{t\in\{0,1,\ldots,T-1\}\\ \mathcal{O}_t = j}} \gamma_t(i)$, $\mathcal{B}_i = \sum\limits_{t=0}^{T-1} \gamma_t(i)$, and $\mathcal{C} = \sum\limits_{t=0}^{T-1} c_t$.

b) For the best set of parameters (τ, α) that you identified in part a), train an HMM in an online mode. Specifically, train a model based on $T_1 = 5000$ samples, to obtain weight matrices W_1 and V_1. Then train a model on the next $T_2 = 5000$ samples to obtain weight matrices W_2 and V_2. Let $W = W_1 + W + 2$ and $V = V_1 + V_2$ be the weight matrices for the combined $T = 10{,}000$ samples. Continue updating the model with blocks of 5000 samples until you reach $T = 50{,}000$ total samples. Compare the B matrix of your online model based on $V = V_1 + V_2 + \cdots + V_{10}$ to a model trained using all $T = 50{,}000$ samples at once (i.e., a non-online model). Also train this non-online model using the same parameters (τ, α) that you determined in part a).

11. From Appendix A, verify the partial derivative expressions in (12).

12. This problem deals with the weight matrix W discussed in Appendix A. Note that similar results hold for the weight matrix V.

    a) Verify that each row sum of the initial weight matrix W given in Table 1 is equal to the corresponding row sum of the final weight matrix W, as given in Table 1.

    b) Prove that each row sum of W is constant over all iterations.

References

[1] Pierre Baldi and Yves Chavin. Smooth on-line learning algorithms for hidden markov models. *Neural Computation*, 6:307–318, 1994. https://core.ac.uk/download/pdf/4881023.pdf.

[2] Atılım Güneş Baydin, Barak A. Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Automatic differentiation in machine learning: A survey. https://arxiv.org/pdf/1502.05767.pdf, 2018.

[3] Léon Bottou. On-line learning and stochastic approximations. In David Saad, editor, *On-line Learning in Neural Networks*, pages 9–42. Cambridge University Press, New York, NY, USA, 1998.

[4] The Brown corpus of standard American English. http://www.cs.toronto.edu/~gpenn/csc401/a1res.html.

[5] Steven A. Cohen and Matthew W. Granade. Models will run the world. *Wall Street Journal*. https://www.wsj.com/articles/models-will-run-the-world-1534716720, 2018.

[6] Matthew R. Gormley. Neural networks and backpropagation. https://www.cs.cmu.edu/~mgormley/courses/10601-s17/slides/lecture20-backprop.pdf, 2017.

[7] Larry Hardesty. Explained: Neural networks. http://news.mit.edu/2017/explained-neural-networks-deep-learning-0414, 2017.

[8] Alex Hern. *The Guardian*. Elon Musk says AI could lead to third world war. https://www.theguardian.com/technology/2017/sep/04/elon-musk-ai-third-world-war-vladimir-putin, 2017.

[9] Can Kaan. Deep learning tutorial for beginners. https://www.kaggle.com/kanncaa1/deep-learning-tutorial-for-beginners, 2018.

[10] Andrey Kurenkov. A 'brief' history of neural nets and deep learning. http://www.andreykurenkov.com/writing/ai/a-brief-history-of-neural-nets-and-deep-learning/, 2015.

[11] Warren S. McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 5, 1943. https://pdfs.semanticscholar.org/5272/8a99829792c3272043842455f3a110e841b1.pdf.

[12] Marvin Minsky and Seymour Papert. *Perceptrons: An Introduction to Computational Geometry*. MIT Press, 1969.

[13] Raúl Rojas. *Neural Networks — A Systematic Introduction*. Springer, 1996. https://page.mi.fu-berlin.de/rojas/neural/.

[14] Frank Rosenblatt. Principles of neurodynamics: Perceptrons and the theory of brain mechanisms. http://www.dtic.mil/dtic/tr/fulltext/u2/256582.pdf, 1961.

[15] David Rumelhart, Geoffrey Hinton, and Ronald Williams. Learning representations by back-propagating errors. *Nature*, 323(9), 1986.

[16] Mark Stamp. A revealing introduction to hidden Markov models. https://www.cs.sjsu.edu/~stamp/RUA/HMM.pdf, 2004.

[17] Mark Stamp. *Introduction to Machine Learning with Applications in Information Security*. Chapman & Hall/CRC Press, 2017. https://www.crcpress.com/Introduction-to-Machine-Learning-with-Applications-in-Information-Security/Stamp/p/book/9781138626782.

[18] Rohit Vobbilisetty, Fabio Di Troia, Richard M. Low, Corrado Aaron Visaggio, and Mark Stamp. Classic cryptanalysis using hidden Markov models. *Cryptologia*, 41(1):1–28, 2017.

[19] Charles Wallis. History of the perceptron. https://web.csulb.edu/~cwallis/artificialn/History.htm, 2017.

[20] Shengwei Zhang and A. G. Constantinides. Lagrange programming neural networks. *IEEE Transactions on Circuits and Systems—II: Analog and Digital Signal Processing*, 39(7):441–452, July 1992. https://www.researchgate.net/publication/3324333_Lagrange_Programming_neural_networks.

Appendix A: HMMs and Gradient Ascent

The article [1] provides an alternative to Baum-Welch re-estimation for training a hidden Markov model (HMM). This alternative is based on gradient ascent, whereas Baum-Welch is a hill climb. One possible advantage of the gradient ascent approach is that it can be applied in an "online" mode, that is, the model can be updated as more data is available, without retraining from scratch. Another advantage is that no probability of zero can occur. The presentation in [1] is somewhat terse, so here we flesh out a few aspects of the algorithm, as well as converting to notation that is consistent with that found in [16] and [17].

A.1: Background

So that this tutorial is reasonably self-contained, in this section, we provide a brief introduction to hidden Markov models (HMMs). For a far more thorough introduction to HMMs and the standard training technique, known as Baum-Welch re-estimation, see [16], or Chapter 2 of [17].

As the name suggests, in an HMM there is a Markov process that is not directly observable. In addition to this (hidden) Markov process, we have a series of observations that are probabilistically related to the hidden states. Figure 18 provides a generic illustration of an HMM, where the X_i represent the hidden states, the A matrix drives the Markov process, and the B matrix relates the hidden states to the observations. The dashed line can be thought of as a "curtain" that we cannot see through—although we can gain probabilistic information about the hidden states from the observations and the B matrix.

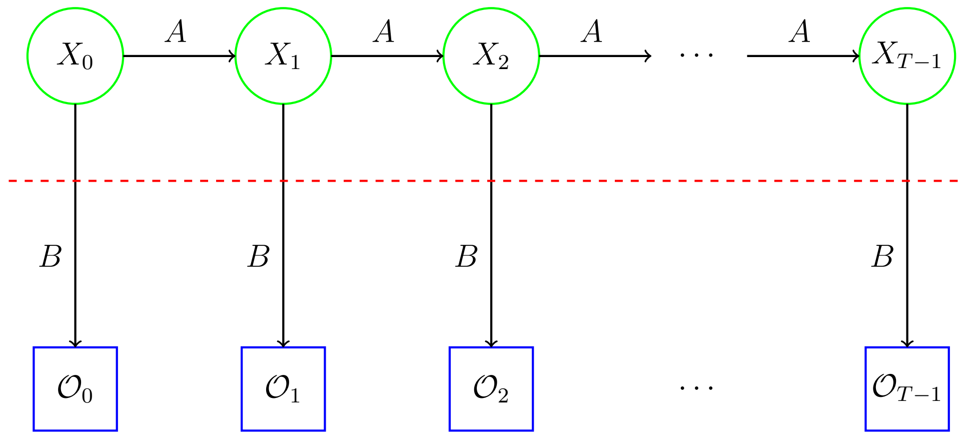


Figure 18: Hidden Markov model

The following notation will be used here, and is generally consistent with that found in [16, 17], for example.

$$
\begin{array}{rcl}
T & = & \text{length of the observation sequence} \\
N & = & \text{number of states in the model} \\
M & = & \text{number of observation symbols} \\
Q & = & \{q_0, q_1, \ldots, q_{N-1}\} = \text{distinct states of the Markov process} \\
V & = & \{0, 1, \ldots, M-1\} = \text{set of possible observations} \\
A & = & \text{state transition probabilities} \\
B & = & \text{observation probability matrix} \\
\pi & = & \text{initial state distribution} \\
\mathcal{O} & = & (\mathcal{o}_0, \mathcal{o}_1, \ldots, \mathcal{o}_{T-1}) = \text{observation sequence.}
\end{array}
$$

The matrices A, B, and π are all row stochastic, that is, each row satisfies the requirements of discrete probability distribution. Furthermore, these three matrices define the hidden Markov model, which we denote as $\lambda = (A, B, \pi)$. In the remainder of this paper, we neglect the π matrix.

Note that the observations are assumed to be drawn from $\{0, 1, \ldots, M-1\}$, which serves to simplify the notation, with no loss of generality. Thus, we have $\mathcal{o}_i \in V$ for $i = 0, 1, \ldots, T-1$. Note also that an observation sequence—as opposed to an individual observation—is denote as $\mathcal{O}$. We'll denote multiple observation sequences as $\mathcal{O}_1$, $\mathcal{O}_2$, and so on.

For $t = 0, 1, \ldots, T-1$ and $i = 0, 1, \ldots, N-1$, define

$$\alpha_t(i) = P(\mathcal{o}_0, \mathcal{o}_1, \ldots, \mathcal{o}_t, x_t = q_i \,|\, \lambda)$$

and

$$\beta_t(i) = P(\mathcal{o}_{t+1}, \mathcal{o}_{t+2}, \ldots, \mathcal{o}_{T-1} \,|\, x_t = q_i, \lambda).$$

Then $\alpha_t(i)$ is the probability of the partial observation sequence up to time t, assuming that the underlying Markov process is in state q_i at time t, and $\beta_t(i)$ is essentially the analog of $\alpha_t(i)$, but for the tail of the sequence as opposed to the head of the sequence.

Next, for $t = 0, 1, \ldots, T-1$ and $i = 0, 1, \ldots, N-1$, define

$$\gamma_t(i) = P(x_t = q_i \,|\, \mathcal{O}, \lambda)$$

and for $t = 0, 1, \ldots, T-2$ and $i, j \in \{0, 1, \ldots, N-1\}$, define the "di-gammas"

$$\gamma_t(i, j) = P(x_t = q_i, x_{t+1} = q_j \,|\, \mathcal{O}, \lambda).$$

Note that $\gamma_t(i)$ is the probability of being in a state q_i at time t while $\gamma_t(i, j)$ is the probability of being in state q_i at time t and transitioning to state q_j

at time $t+1$. The parameters $\gamma_t(i)$ and $\gamma_t(i,j)$ are easily computed in terms of $\alpha_t(i)$ and $\beta_t(i)$ as

$$\gamma_t(i) = \frac{\alpha_t(i)\beta_t(i)}{P(\mathcal{O}\,|\,\lambda)} \quad \text{and} \quad \gamma_t(i,j) = \frac{\alpha_t(i)a_{ij}b_j(\mathcal{O}_{t+1})\beta_{t+1}(j)}{P(\mathcal{O}\,|\,\lambda)}$$

and it is clear that

$$\gamma_t(i) = \sum_{j=0}^{N-1} \gamma_t(i,j).$$

Efficient recursive algorithms exist for computing the $\alpha_t(i)$ and $\beta_t(i)$; see [16, 17] for all of the gory details.

When computing the $\alpha_t(i)$ and $\beta_t(i)$, it is necessary to carefully scale so as to avoid underflow. In [16, 17], these scaling factors, which are easily computed, are denoted as c_i, for $i = 0, 1, \ldots, T-1$. It can be shown that

$$P(\mathcal{O}\,|\,\lambda) = \frac{1}{\prod_{t=0}^{T-1} c_t}. \tag{7}$$

However, in most realistic applications, $\prod c_t$ will surely result in overflow, and hence we typically quantify the performance of the model during training using

$$\log\big(P(\mathcal{O}\,|\,\lambda)\big) = -\sum_{t=0}^{T-1} \log(c_t)$$

Remarkably, computing $\gamma_t(i)$ and $\gamma_t(i,j)$ using the scaled $\alpha_t(i)$ and $\beta_t(i)$ is exact, that is, we obtain exactly the same gammas and di-gammas as we would if we were able to use the unscaled $\alpha_t(i)$ and $\beta_t(i)$.

In Baum-Welch re-estimation, the elements of the HMM matrix $A = \{a_{ij}\}$ are re-estimated based on the expected number of state transitions, which are easily computed in terms of the $\gamma_t(i)$ and $\gamma_t(i,j)$. Similarly, it is easy to update the elements of $B = \{b_i(j)\}$, once the $\gamma_t(i)$ and $\gamma_t(i,j)$ have been computed. After the matrices A and B have been updated, the $\alpha_t(i)$, $\beta_t(i)$, $\gamma_t(i,j)$, and $\gamma_t(i)$ are recomputed, and the process is repeated. It can be shown that Baum-Welch re-estimation is a hill climb, that is, $P(\mathcal{O}\,|\,\lambda)$ cannot decrease at any iteration. Since it is a hill climb, Baum-Welch re-estimation can only find a local maximum, and hence it is often beneficial to train multiple models using different initial values for the model parameters.

The bottom line here is that we can efficiently compute $\gamma_t(i)$, $\gamma_t(i,j)$, and the scaling factors c_t. We will make use of these quantities in the gradient ascent algorithm that we next consider.

A.2: HMM Training via Gradient Ascent

The following HMM training algorithm is presented in [1] as an alternative to Baum-Welch re-estimation. In this algorithm, we update the the elements of the matrices A and B based on a softmax function and matrices of weights. The $N \times N$ weight matrix $W = \{w_{ij}\}$ is used to update A and the $N \times M$ matrix of weights $V = \{v_{ij}\}$ is used to update B. The update steps consists of

$$a_{ij} = \frac{e^{\tau w_{ij}}}{\sum\limits_{k=0}^{N-1} e^{\tau w_{ik}}} \quad \text{and} \quad b_i(j) = \frac{e^{\tau v_{ij}}}{\sum\limits_{k=0}^{M-1} e^{\tau v_{ik}}}, \tag{8}$$

where τ is a "temperature" parameter. Due to the use of the softmax function, we avoid 0 probabilities, and we are also assured that the row stochastic requirements for A and B will be satisfied, regardless of the weights in W and V.

We update W and V in (8) according to

$$\widetilde{w}_{ij} = w_{ij} + \frac{\rho}{\mathcal{C}(\mathcal{O})}\big(\mathcal{A}_{ij}(\mathcal{O}) - \mathcal{A}_i(\mathcal{O})a_{ij}\big) \tag{9}$$

and

$$\widetilde{v}_{ij} = v_{ij} + \frac{\rho}{\mathcal{C}(\mathcal{O})}\big(\mathcal{B}_{ij}(\mathcal{O}) - \mathcal{B}_i(\mathcal{O})b_i(j)\big), \tag{10}$$

where ρ is the learning rate, $\mathcal{C}(\mathcal{O}) = P(\mathcal{O}\,|\,\lambda)$, and $\mathcal{A}_{ij}(\mathcal{O})$ is the expected number of transitions from state i to state j over the observation sequence $\mathcal{O}$, and $\mathcal{B}_{ij}(\mathcal{O})$ is the expected number of times we observe j in state i for $\mathcal{O}$, while

$$\mathcal{A}_i(\mathcal{O}) = \sum_j \mathcal{A}_{ij}(\mathcal{O}) \quad \text{and} \quad \mathcal{B}_i(\mathcal{O}) = \sum_j \mathcal{B}_{ij}(\mathcal{O}).$$

Each of $\mathcal{A}_{ij}(\mathcal{O})$, $\mathcal{A}_i(\mathcal{O})$, $\mathcal{B}_{ij}(\mathcal{O})$, $\mathcal{B}_i(\mathcal{O})$, and $\mathcal{C}(\mathcal{O})$ is easily computed in terms of the $\gamma_t(i,j)$, $\gamma_t(i)$ and c_t discussed in Section A.1; see [16] or Chapter 2 of [17] for the algorithms used to compute these elements.

We have

$$\mathcal{A}_{ij}(\mathcal{O}) = \sum_{t=0}^{T-2} \gamma_t(i,j) \quad \text{and} \quad \mathcal{A}_i(\mathcal{O}) = \sum_{t=0}^{T-2} \gamma_t(i) \tag{11}$$

and

$$\mathcal{B}_{ij}(\mathcal{O}) = \sum_{\substack{t\in\{0,1,\ldots,T-1\}\\ \mathcal{O}_t=j}} \gamma_t(i) \quad \text{and} \quad \mathcal{B}_i(\mathcal{O}) = \sum_{t=0}^{T-1} \gamma_t(i)$$

and

$$\mathcal{C}(\mathcal{O}) = P(\mathcal{O}\,|\,\lambda) = \frac{1}{\prod\limits_{t=0}^{T-1} c_t}$$

However, as a practical matter, we cannot compute $\mathcal{C}(\mathcal{O})$ without having overflow. We discuss this issue in more detail below.

We now show that the update formula in (9) is a gradient ascent; a similar argument holds for (10). First, we note that (8) implies

$$\frac{\partial a_{ij}}{\partial w_{ij}} = \tau a_{ij}(1 - a_{ij}) \text{ and } \frac{\partial a_{ij}}{\partial w_{ik}} = -\tau a_{ij}a_{ik} \tag{12}$$

Next, for any given state sequence $X = (X_0, X_1, \ldots, X_{T-1})$, let

$$f(X) = \pi_{X_0} b_{X_0}(\mathcal{o}_0) a_{X_0,X_1} b_{X_1}(\mathcal{o}_1) \cdots a_{X_{T-2},X_{T-1}} b_{X_{T-1}}(\mathcal{o}_{T-1})$$

and define the likelihood function $L_{\mathcal{O}}(\lambda) = P(\mathcal{O} \,|\, \lambda)$. We have

$$L_{\mathcal{O}}(\lambda) = \sum_X f(X).$$

When training the HMM, our goal is to maximize the likelihood $L_{\mathcal{O}}(\lambda)$. It is not difficult to verify that

$$\frac{\partial f(X)}{\partial a_{ij}} = g_{ij}(X) \frac{f(X)}{a_{ij}}$$

where $g_{ij}(X)$ is simply the number of transitions from state i to state j in the state sequence X, that is, $g_{ij}(X)$ is the number of times that a_{ij} appears in $f(X)$. Therefore,

$$\frac{\partial L_{\mathcal{O}}(\lambda)}{\partial a_{ij}} = \sum_X g_{ij}(X) \frac{f(X)}{a_{ij}} = \frac{\mathcal{A}_{ij}(\mathcal{O})}{a_{ij}} \tag{13}$$

where $\mathcal{A}_{ij}(\mathcal{O}) = \sum_X g_{ij}(X) f(X)$ and, as above, $\mathcal{A}_{ij}(\mathcal{O})$ is the expected number of transitions from state i to state j. Now, by the chain rule and (12), we have

$$\begin{aligned}
\frac{\partial L_{\mathcal{O}}(\lambda)}{\partial w_{ij}} &= \sum_{k=0}^{N-1} \frac{\partial L_{\mathcal{O}}(\lambda)}{\partial a_{ik}} \frac{\partial a_{ik}}{\partial w_{ij}} \\
&= \tau \mathcal{A}_{ij}(\mathcal{O})(1 - a_{ij}) - \tau a_{ij} \sum_{k \neq j} \mathcal{A}_{ik}(\mathcal{O}) \\
&= \tau \Big(\mathcal{A}_{ij}(\mathcal{O}) - a_{ij} \sum_{k=0}^{N-1} \mathcal{A}_{ik}(\mathcal{O}) \Big) \\
&= \tau \big(\mathcal{A}_{ij}(\mathcal{O}) - a_{ij} \mathcal{A}_i(\mathcal{O}) \big)
\end{aligned}$$

It follows that

$$\frac{\partial \log L_{\mathcal{O}}(\lambda)}{\partial w_{ij}} = \frac{\tau}{L_{\mathcal{O}}(\lambda)}\big(\mathcal{A}_{ij}(\mathcal{O}) - a_{ij}\mathcal{A}_i(\mathcal{O})\big) \tag{14}$$

which gives us a gradient ascent algorithm on the negative log likelihood function. The expression in (14) directly yields the claimed update formula (9), where $\mathcal{C}(\mathcal{O})$ is a function of $P(\mathcal{O}\,|\,\lambda)$. Note that if η is the selected learning rate, then ρ that appears in (9) is $\eta\tau$. For simplicity, we'll simply refer to ρ as the learning rate.

To update the weight w_{ij}, we need to compute

$$\frac{\rho}{L_{\mathcal{O}}(\lambda)}\big(\mathcal{A}_{ij}(\mathcal{O}) - a_{ij}\mathcal{A}_i(\mathcal{O})\big). \tag{15}$$

Unfortunately, we cannot evaluate $L_{\mathcal{O}}(\lambda)$ without underflow. Fortunately, we can compute $\log L_{\mathcal{O}}(\lambda)$, and by applying the logarithm function to (15), we find

$$\begin{aligned}
z &= \log\Big(\frac{\rho}{L_{\mathcal{O}}(\lambda)}\big(\mathcal{A}_{ij}(\mathcal{O}) - a_{ij}\mathcal{A}_i(\mathcal{O})\big)\Big) \\
&= \log\rho - \log L_{\mathcal{O}}(\lambda) + \log\big(\mathcal{A}_{ij}(\mathcal{O}) - a_{ij}\mathcal{A}_i(\mathcal{O})\big) \\
&= \log\eta + \log\tau + \sum_{t=0}^{T-1}\log(c_t) + \log\big(\mathcal{A}_{ij}(\mathcal{O}) - a_{ij}\mathcal{A}_i(\mathcal{O})\big)
\end{aligned}$$

where η is the learning rate, τ is the temperature parameter, and the c_t are the scaling factors in (7). By computing z in this way, we can update the weight w_{ij} as

$$\widetilde{w}_{ij} = w_{ij} + e^z,$$

assuming that $\log(x)$ is the natural logarithm $\ln(x)$; otherwise, we simply replace e by the base of the logarithm. Of course, this only makes sense provided that the step actually ascends, in which case $\mathcal{A}_{ij}(\mathcal{O}) - a_{ij}\mathcal{A}_i(\mathcal{O}) > 0$, and even when this is the true, machine precision and numerical instability are likely to present significant difficulties. In practice, we find that

$$\mathcal{C}(\mathcal{O}) = -\log L_{\mathcal{O}}(\lambda) = \sum_{t=0}^{T-1}\log(c_t). \tag{16}$$

works reasonably well in the update formulas (9) and (10).

A.3: Online HMM Training

Suppose that we train an HMM on the observation sequence $\mathcal{O}_1$ and subsequently, more training data becomes available, in the form of another observation sequence $\mathcal{O}_2$. We would like to have a model that represents the observation super-sequence $\mathcal{O} = (\mathcal{O}_1, \mathcal{O}_2)$, where the sequence $\mathcal{O}$ consists of $\mathcal{O}_2$ appended to $\mathcal{O}_1$. We could simply start over from scratch and train a new model on $\mathcal{O}$ by using Baum-Welch, or the gradient ascent algorithm discussed in the previous section. However, our goal here is to devise an online training method, as opposed to a batch method such as Baum-Welch. That is, we want to find a method that enables us to update a trained HMM as more data becomes available, without the need to train a new model on the entire observation super-sequence. The gradient ascent algorithm discussed above is easily modified to handle the online case.

Let $\mathcal{O} = (\mathcal{O}_1, \mathcal{O}_2, \ldots, \mathcal{O}_n)$ be an observation super-sequence, consisting of n observation sequences and let $L_{\mathcal{O}}(\lambda) = P(\mathcal{O} \mid \lambda)$ be the likelihood function for the concatenated observations sequence $\mathcal{O} = (\mathcal{O}_1, \mathcal{O}_2, \ldots, \mathcal{O}_n)$. Then from (7), it follows that

$$L_{\mathcal{O}}(\lambda) = \prod_{k=1}^{n} L_{\mathcal{O}_k}(\lambda).$$

Our goal here is to compute $\partial L_{\mathcal{O}}(\lambda)/\partial w_{ij}$. But first we will determine a formula for $\partial L_{\mathcal{O}}(\lambda)/\partial a_{ij}$, and then use this result, together with the chain rule, to find the desired partial derivative.

By the product rule, and making use of (13), we find

$$\begin{aligned}\frac{\partial L_{\mathcal{O}}(\lambda)}{\partial a_{ij}} &= \frac{\partial \Big(\prod\limits_{k=1}^{n} L_{\mathcal{O}_k}(\lambda)\Big)}{\partial a_{ij}} = \frac{1}{a_{ij}} \sum_{k=1}^{n} \Big(\mathcal{A}_{ij}(\mathcal{O}_k) \prod_{\ell \neq k} L_{\mathcal{O}_\ell}(\lambda)\Big) \\ &= \frac{L_{\mathcal{O}}(\lambda)}{a_{ij}} \sum_{k=1}^{n} \frac{\mathcal{A}_{ij}(\mathcal{O}_k)}{L_{\mathcal{O}_k}(\lambda)}\end{aligned} \tag{17}$$

From the chain rule, we have

$$\frac{\partial L_{\mathcal{O}}(\lambda)}{\partial w_{ij}} = \sum_{k=0}^{N-1} \frac{\partial L_{\mathcal{O}}(\lambda)}{\partial a_{ik}} \frac{\partial a_{ik}}{\partial w_{ij}}$$

Expanding the $\partial a_{ik}/\partial w_{ij}$ terms using (12), we find

$$\frac{\partial L_{\mathcal{O}}(\lambda)}{\partial w_{ij}} = \sum_{k=0}^{N-1} \frac{\partial L_{\mathcal{O}}(\lambda)}{\partial a_{ik}} \frac{\partial a_{ik}}{\partial w_{ij}}$$

$$= \tau a_{ij} \frac{\partial L_{\mathcal{O}}(\lambda)}{\partial a_{ij}} \bigl(1 - a_{ij}\bigr) - \tau a_{ij} \sum_{k \neq j}^{N-1} a_{ik} \frac{\partial L_{\mathcal{O}}(\lambda)}{\partial a_{ik}}$$
$$= \tau a_{ij} \biggl(\frac{\partial L_{\mathcal{O}}(\lambda)}{\partial a_{ij}} - \sum_{k=0}^{N-1} a_{ik} \frac{\partial L_{\mathcal{O}}(\lambda)}{\partial a_{ik}} \biggr)$$

Finally, we expand the $\partial L_{\mathcal{O}}(\lambda)/\partial a_{ik}$ terms using (17) to obtain

$$\begin{aligned}
\frac{\partial L_{\mathcal{O}}(\lambda)}{\partial w_{ij}} &= \tau a_{ij} \biggl(\frac{\partial L_{\mathcal{O}}(\lambda)}{\partial a_{ij}} - \sum_{k=0}^{N-1} a_{ik} \frac{\partial L_{\mathcal{O}}(\lambda)}{\partial a_{ik}} \biggr) \\
&= \tau a_{ij} \biggl(\frac{L_{\mathcal{O}}(\lambda)}{a_{ij}} \sum_{\ell=1}^{n} \frac{\mathcal{A}_{ij}(\mathcal{O}_\ell)}{L_{\mathcal{O}_\ell}(\lambda)} - L_{\mathcal{O}}(\lambda) \sum_{k=0}^{N-1} \sum_{\ell=1}^{n} \frac{\mathcal{A}_{ik}(\mathcal{O}_\ell)}{L_{\mathcal{O}_\ell}(\lambda)} \biggr) \\
&= \tau L_{\mathcal{O}}(\lambda) \biggl(\sum_{\ell=1}^{n} \frac{\mathcal{A}_{ij}(\mathcal{O}_\ell)}{L_{\mathcal{O}_\ell}(\lambda)} - a_{ij} \sum_{\ell=1}^{n} \sum_{k=0}^{N-1} \frac{\mathcal{A}_{ik}(\mathcal{O}_\ell)}{L_{\mathcal{O}_\ell}(\lambda)} \biggr) \\
&= \tau L_{\mathcal{O}}(\lambda) \biggl(\sum_{\ell=1}^{n} \frac{\mathcal{A}_{ij}(\mathcal{O}_\ell)}{L_{\mathcal{O}_\ell}(\lambda)} - a_{ij} \sum_{\ell=1}^{n} \frac{\mathcal{A}_{i}(\mathcal{O}_\ell)}{L_{\mathcal{O}_\ell}(\lambda)} \biggr) \\
&= \tau L_{\mathcal{O}}(\lambda) \sum_{\ell=1}^{n} \frac{1}{L_{\mathcal{O}_\ell}(\lambda)} \Bigl(\mathcal{A}_{ij}(\mathcal{O}_\ell) - a_{ij} \mathcal{A}_{i}(\mathcal{O}_\ell) \Bigr)
\end{aligned}$$

It follows immediately that for the log likelihood function $\log L_{\mathcal{O}}(\lambda)$, we have

$$\begin{aligned}
\frac{\partial \log L_{\mathcal{O}}(\lambda)}{\partial w_{ij}} &= \frac{1}{L_{\mathcal{O}}(\lambda)} \frac{\partial L_{\mathcal{O}}(\lambda)}{\partial w_{ij}} \\
&= \tau \sum_{k=1}^{n} \frac{1}{L_{\mathcal{O}_k}(\lambda)} \Bigl(\mathcal{A}_{ij}(\mathcal{O}_k) - a_{ij} \mathcal{A}_{i}(\mathcal{O}_k) \Bigr)
\end{aligned}$$

Analogous to (16), we use

$$\mathcal{C}(\mathcal{O}_k) = \sum_{t=0}^{T-1} \log(c_t)$$

in place of the normalizing factor $L_{\mathcal{O}_k}(\lambda)$. We also replace τ with the learning rate ρ.

The upshot here is that we can update the weights w_{ij} for the observation super-sequence $\mathcal{O} = (\mathcal{O}_1, \mathcal{O}_2, \ldots, \mathcal{O}_n)$ based on the weights obtained for each sequence $\mathcal{O}_i$. Consequently, as more training data becomes available, we only need to compute weights for these new observations, as opposed to retraining in batch mode over the entire super-sequence. Thus we have an online HMM training algorithm based on the log likelihood function $\log L_{\mathcal{O}}(\lambda)$. Analogous statements hold for the weights v_{ij} that are used to update the B matrix.

A.4: HMM Gradient Ascent Example

Here, we duplicate the example found in Section 8 of [16] and Section 9.2 of [17], but using the gradient ascent algorithm for HMM training, rather than Baum-Welch re-estimation. For this example, we train an HMM on English text, where all characters other than the letters and word space have been removed, and all uppercase letters have been converted to lowercase. This leaves us with $M = 27$ distinct observation symbols. We'll choose $N = 2$ hidden states, and train based on a sequence of $T = 50{,}000$ observations extracted from the Brown Corpus [4]. Also, we'll train for 500 iterations of the gradient ascent defined by equations (8), (9), and (10). That is, we initialize W and V, then compute initial values for A and B based on these initial values for W and V using the softmax function in (8). We then proceed to update W and V according to (9) and (10), respectively, and recompute A and B via (8) for 500 iterations. We select $\tau = 2.5$ and $\rho = 12.0$ for the temperature and learning rate parameters.

The initial and final weights W are given in Table 1, while the initial and final weights V appear in Table 2. Similarly, the initial and final A matrix is in Table 3, while the initial and final B matrix is given in Table 4. From the final B matrix, it is clear that the hidden states in the converged model correspond to the consonants and vowels (where"y" acts as a consonant and space acts as a vowel). These results are entirely consistent with results obtained using Baum-Welch re-estimation, as given in Table 3 in [16] and Table 9.1 in [17]

Table 1: Initial and final weights W

	Initial		Final	
State 0	2.00000	2.00000	1.79125	2.20875
State 1	1.00000	2.00000	1.67991	1.32009

In this section, we used $T = 50{,}000$ observations when training the HMM via gradient ascent. This number was chosen for consistency with the English text examples in [16] and [17], which were trained using Baum-Welch re-estimation. In data-limited problems, such as the classic cryptanalysis problems considered in [18], vast numbers of models are trained with random initial conditions, and the best model is selected. It would be interesting to compare the performance of Baum-Welch and gradient ascent training on such data-limited problems. Intuitively, it seem likely that for optimal choices of τ and ρ, the gradient ascent would, on average, converge to an equivalent or better solution than Baum-Welch. Of course, it may be difficult to determine optimal values for the parameters in general, but it might be possible to find near-optimal values in some interesting special cases.

Table 2: Initial and final weights in V (transposed)

	Initial		Final	
	State 0	State 1	State 0	State 1
a	2.00000	2.00000	3.22750	0.95100
b	1.00000	1.00000	0.67290	1.68933
c	1.00000	2.00000	0.88350	2.04444
d	1.00000	2.00000	0.71186	2.12596
e	2.00000	2.00000	3.40646	0.75535
f	1.00000	2.00000	0.57337	1.86240
g	1.00000	2.00000	0.90313	1.75979
h	2.00000	1.00000	1.92843	2.12216
i	1.00000	1.00000	3.18567	0.32958
j	1.00000	1.00000	0.47211	0.90600
k	2.00000	1.00000	1.30889	1.24483
l	1.00000	1.00000	1.40761	2.14610
m	1.00000	1.00000	0.74382	1.89906
n	1.00000	2.00000	0.84075	2.32939
o	2.00000	2.00000	3.21378	0.57570
p	2.00000	1.00000	1.13639	1.87137
q	2.00000	2.00000	0.60648	0.55095
r	1.00000	1.00000	0.90378	2.28106
s	1.00000	2.00000	0.80650	2.31661
t	1.00000	1.00000	1.44821	2.44446
u	2.00000	2.00000	2.77354	0.42467
v	1.00000	1.00000	0.60466	1.54489
w	1.00000	1.00000	0.55860	1.68829
x	2.00000	1.00000	0.78836	1.01365
y	1.00000	2.00000	0.75138	1.72812
z	2.00000	2.00000	0.54381	0.43223
space	2.00000	1.00000	3.59852	0.96260

Table 3: Initial and final A

	Initial		Final	
State 0	0.50000	0.50000	0.26043	0.73957
State 1	0.07586	0.92414	0.71086	0.28914

Table 4: Initial and final B (transposed)

	Initial		Final	
	State 0	State 1	State 0	State 1
a	0.08121	0.07068	0.13537	0.00364
b	0.00667	0.00580	0.00023	0.02307
c	0.00667	0.07068	0.00039	0.05605
d	0.00667	0.07068	0.00025	0.06873
e	0.08121	0.07068	0.21176	0.00223
f	0.00667	0.07068	0.00018	0.03556
g	0.00667	0.07068	0.00041	0.02751
h	0.08121	0.00580	0.00526	0.06808
i	0.00667	0.00580	0.12193	0.00077
j	0.00667	0.00580	0.00014	0.00326
k	0.08121	0.00580	0.00112	0.00759
l	0.00667	0.00580	0.00143	0.07227
m	0.00667	0.00580	0.00027	0.03897
n	0.00667	0.07068	0.00035	0.11429
o	0.08121	0.07068	0.13081	0.00143
p	0.08121	0.00580	0.00073	0.03637
q	0.08121	0.07068	0.00019	0.00134
r	0.00667	0.00580	0.00041	0.10128
s	0.00667	0.07068	0.00032	0.11069
t	0.00667	0.00580	0.00158	0.15238
u	0.08121	0.07068	0.04352	0.00098
v	0.00667	0.00580	0.00019	0.01608
w	0.00667	0.00580	0.00017	0.02301
x	0.08121	0.00580	0.00030	0.00426
y	0.00667	0.07068	0.00028	0.02542
z	0.08121	0.07068	0.00017	0.00100
space	0.08121	0.00580	0.34226	0.00375

Appendix B: Lagrangian for HMM Training

As a first example of backpropagation, we turn to a seemingly unlikely source, namely, our old friend, the hidden Markov model (HMM).[12] The presentation here was stimulated by the stimulating discussion in Section 7.4.2 of Rojas' book [13]. Before we get to backpropagation, we'll provide a very brief overview of HMMs. The reader who is not intimately familiar with HMMs is encouraged to review your humble author's fine tutorial [16], or the equally stellar presentation in Chapter 2 of your modest author's most excellent book [17].

As the name suggests, a hidden Markov model includes a Markov process that is "hidden," in the sense that it is not directly observable. An HMM also includes a series of observations that are probabilistically related to the hidden states. A generic view of an HMM is given in Figure 19, where the X_i are the hidden states, the $\mathcal{O}_i$ are observations, A is the matrix that drives the (hidden) Markov process, B contains the probabilities that relate the hidden states to the observations, and T is the number of observations. The dashed line can be thought of as a "curtain" that we cannot see through—we can observe the observations,[13] but we cannot (directly) view the hidden states.

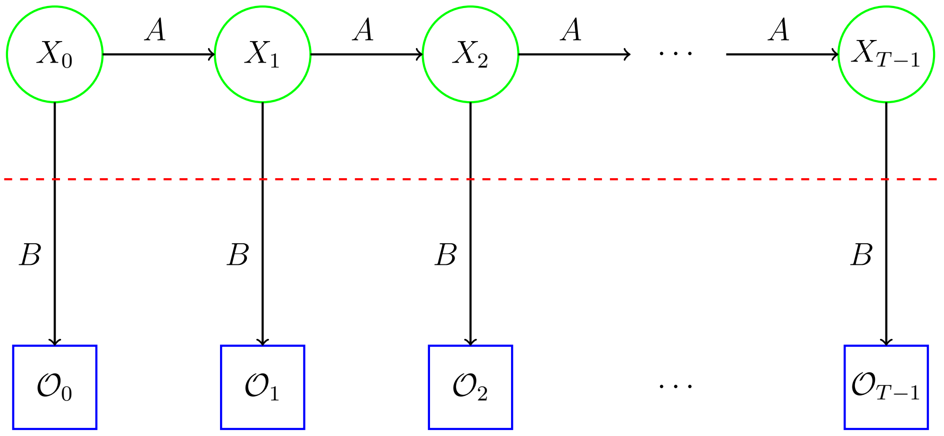


Figure 19: Generic view of a hidden Markov model

In an HMM, the A matrix is $N \times N$, where N is the number of hidden states, while the B matrix is $N \times M$, where M is the number of distinct observation symbols. There is also a $1 \times N$ initial state distribution matrix denoted as π. All three of the matrices A, B, and π are row stochastic, which is a four-dollar

[12]If you don't view HMMs as your friend—old or otherwise—then you obviously have not read [16] or [17].

[13]This sentence is brought to you courtesy of the Department of Redundancy Department.

word[14] meaning that each row satisfies the requirements of a discrete probability distribution. These three matrices define an HMM, which we denote as the triple $\lambda = (A, B, \pi)$.

The practical utility and widespread usage of HMMs derives largely from the fact that there are efficient algorithms to solve the following three problems.

1. Given a trained HMM, $\lambda = (A, B, \pi)$, and an observation sequence $\mathcal{O}_i$, for $i = 0, 1, \ldots, T-1$, we can score the observation sequence against the model.

2. Given a trained HMM, $\lambda = (A, B, \pi)$, and an observation sequence $\mathcal{O}_i$, for $i = 0, 1, \ldots, T-1$, we can determine the "best" hidden state sequence X_i, for $i = 0, 1, \ldots, T-1$. Here, "best" is defined in terms of expectation maximization (EM), that is, we maximize the expected number of correct states. Note that this is in contrast to the definition of "best" used in a dynamic program, where we determine the highest scoring overall path.

3. Given an observation sequence $\mathcal{O}_i$, for $i = 0, 1, \ldots, T-1$, we can train a model $\lambda = (A, B, \pi)$ to fit the observations. In this training process, we must specify N, the number of hidden states, but otherwise the process is entirely data driven. This is the sense in which HMMs are a machine learning technique.

Often, we'll train a model using the algorithm alluded to in problem 3, above, then use the resulting model to score sequences via the algorithm corresponding to problem 1. For example, we might train an HMM on opcode sequences extracted from a collection of known malware samples, all of which belong to a specific malware family. Then, given an unknown sample that we want to classify, we can extract its opcode sequence and score this sequence of observations against the HMM. If the sequence scores high, then the sample closely matches the training samples, and we would classify it as belonging to the same malware family that was used for training. On the other hand, if the sample does not score well against the model, we would conclude that it does not belong to the same malware family that was used to train the HMM.[15] Using an HMM in this way, the model acts essentially as a higher-level "signature," in the sense that the model can be used to classify an entire malware family, rather than an individual malware sample.

[14]Yes, I do realize that "row stochastic" is a phrase, not a word. But, "four-dollar phrase" just doesn't sound right.

[15]That's a long-winded way of saying that an HMM yields a binary classifier.

The usual way to train an HMM is via the well-known Baum-Welch re-estimation algorithm [17], which is an iterative hill climb procedure. In this process, we initialize the A, B, and π matrices, then compute new elements for these matrices—in a fairly straightforward and intuitive way—based on the observation sequence. By simply repeating this re-estimation process, we climb to a local maximum in the parameter space, where the "parameters" are the elements of the matrices that define the HMM.

Here, we want to show that the HMM training process can also be viewed as an artificial neural network (ANN), and hence an HMM can be trained using backpropagation. Lagrange multipliers—another old friend of ours—also play an important role in this story.

We denote the elements of the matrices that define an HMM as $A = \{a_{ij}\}$, $B = \{b_j(k)\}$, and $\pi = \{\pi_i\}$. Training an HMM on $\mathcal{O}_0, \mathcal{O}_1, \ldots, \mathcal{O}_{T-1}$ implies that we attempt to determine the matrices of $\lambda = (A, B, \pi)$ so that

$$P(\mathcal{O} \mid \lambda) = \sum_X \pi_{X_0} b_{X_0}(\mathcal{O}_0) a_{X_0,X_1} b_{X_1}(\mathcal{O}_1) \cdots a_{X_{T-2},X_{T-1}} b_{X_{T-1}}(\mathcal{O}_{T-1}) \tag{18}$$

is maximized.[16] In (18), the sum is computed over all possible state sequences of the form $X = (X_0, X_1, \ldots, X_{T-1})$. Directly computing $P(\mathcal{O} \mid \lambda)$ by this formula would require about $2TN^T$ operations, and hence is infeasible in almost any realistic scenario. The HMM *forward algorithm*, which is also known as the α-pass, allows us to compute this probability efficiently, with only about N^2T multiplications. The forward algorithm is the key to making Baum-Welch re-estimation practical, and to efficient scoring using a trained HMM.

For example, consider the case where we have $N = 2$ and $T = 3$. Then the expression (18) can be computed using the graph in Figure 20. To train the corresponding HMM, we want to determine π_i, a_{ij}, and $b_j(k)$ that maximize the probability produced by the graph in Figure 20. All of the functions in this graph are "nice" (we only have multiplication and addition), so it might be feasible to train this network using the backpropagation algorithm. However, since A and B and π are all row stochastic, there are constraints on the parameters (i.e., the edge weights) in the graph. Specifically, each π_i, a_{ij}, and $b_j(k)$ must be between 0 and 1, and each row of A, B and π must sum to 1. Lagrange multipliers provide a way to deal with constrained optimization problems—see Chapter 5 in [17], for example, where Lagrange multipliers are introduced and discussed in the context of SVMs.

[16]We have slightly abused the notation. While the observations are assumed to be numeric, with $\mathcal{O}_i \in \{0, 1, 2, \ldots, M-1\}$, the states X_i are not necessarily numeric values. Thus, when we write X_i as an index, it should be taken to mean that the states have been ordered and X_i is the position in that ordering of the hidden state that occurs at step i.

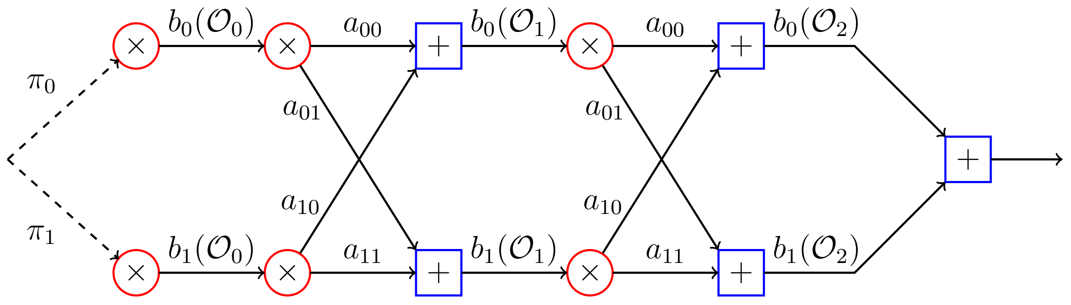


Figure 20: Graph to compute $P(\mathcal{O}\,|\,\lambda)$ for $N = 2$ and $T = 3$

For the example in Figure 20, let

$$\begin{aligned} x &= (x_0, x_1, \ldots, x_{11}) \\ &= \big(\pi_0, \pi_1, a_{00}, a_{01}, a_{10}, a_{11}, \\ &\qquad b_0(\mathcal{O}_1), b_0(\mathcal{O}_0), b_0(\mathcal{O}_2), b_1(\mathcal{O}_1), b_1(\mathcal{O}_0), b_1(\mathcal{O}_2)\big). \end{aligned} \tag{19}$$

Then the equality constraints

$$\begin{aligned} g_0(x) &= x_0 + x_1 = 1 \\ g_1(x) &= x_2 + x_3 = 1 \\ g_2(x) &= x_4 + x_5 = 1 \\ g_3(x) &= x_6 + x_7 + x_8 = 1 \\ g_4(x) &= x_9 + x_{10} + x_{11} = 1 \end{aligned} \tag{20}$$

must be satisfied. In addition there are inequality constraints of the form

$$0 \le x_i \le 1 \ \text{ for } \ i = 0, 1, \ldots, 11.$$

Ignoring the inequality constraints, the Lagrangian for this problem is

$$L(x, u) = f(x) + \sum_{i=0}^{4} u_i\big(g_i(x) - 1\big), \tag{21}$$

where $f(x)$ is the function computed in Figure 20 and the u_i are the Lagrange multipliers.[17] For convenience, in Figure 21, we've rewritten the graph given in Figure 20 in terms of the x_i in (19).

Unfortunately, Figures 20 and 21 are not in a form that is particularly friendly for applying the backpropagation algorithm. But, graphs such as these

[17]Typically, the coefficients u_i are denoted as λ_i, but here we'll use u_i so as to avoid confusion with the HMM, for which we denote the model as λ.

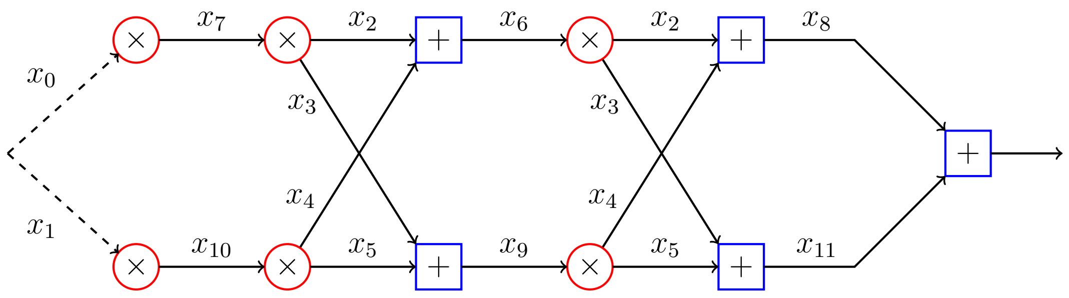


Figure 21: Graph in Figure 20 with x from (19)

do enable us to fairly easily derive the HMM forward algorithm, which is the key step in the standard HMM training technique of Baum-Welch re-estimation. For the sake of brevity, we omit the details of the derivation of the HMM forward algorithm; see [16] or Chapter 2 of [17] for all of the glorious details.

The HMM forward algorithm is given here in Figure 22. Note that the observation sequence is of length T and the model has N hidden states.

```
1: for $i = 0, 1, \ldots, N-1$
2:     $\alpha_0(i) = \pi_i b_i(\mathcal{O}_0)$
3: for $t = 1, 2, \ldots, T-1$
4:     for $i = 0, 1, \ldots, N-1$
5:         $\alpha_t(i) = \left(\sum_{j=0}^{N-1} \alpha_{t-1}(j) a_{ji}\right) b_i(\mathcal{O}_t)$
```

Figure 22: HMM forward algorithm (without scaling)

The probability of the observation sequence with respect to the model λ is easily computed in terms of the $\alpha_t(i)$ as

$$P(\mathcal{O} \mid \lambda) = \sum_{i=0}^{N-1} \alpha_{T-1}(i).$$

For the specific example in Figure 20, the forward algorithm computation is given in Figure 23.

Rewriting the code in Figure 23 in terms of the x_i in (19), and putting the equations in the form of a program similar to those in Section 5, we obtain the pseudo-code in Figure 24.

The code in Figure 24 enables us to efficiently compute $f(x) = P(\mathcal{O} \mid \lambda)$ for any specified values of the parameters, i.e., the elements of the matrices of the

```
1:  $\alpha_0(0) = \pi_0 b_0(\mathcal{O}_0)$
2:  $\alpha_0(1) = \pi_1 b_1(\mathcal{O}_0)$
3:  $\alpha_1(0) = \big(\alpha_0(0)a_{00} + \alpha_0(1)a_{10}\big)b_0(\mathcal{O}_1)$
4:  $\alpha_1(1) = \big(\alpha_0(0)a_{01} + \alpha_0(1)a_{11}\big)b_1(\mathcal{O}_1)$
5:  $\alpha_2(0) = \big(\alpha_1(0)a_{00} + \alpha_1(1)a_{10}\big)b_0(\mathcal{O}_2)$
6:  $\alpha_2(1) = \big(\alpha_1(0)a_{01} + \alpha_1(1)a_{11}\big)b_1(\mathcal{O}_2)$
7:  $P(\mathcal{O}\,|\,\lambda) = \alpha_2(0) + \alpha_2(1)$
```

Figure 23: HMM forward algorithm applied to the graph in Figure 21

```
1:  $v_i = x_i$ for $i = 0, 1, \ldots, 11$ // initialization
2:  $v_{12} = v_0 v_7$
3:  $v_{13} = v_1 v_{10}$
4:  $v_{14} = (v_{12}v_2 + v_{13}v_4)v_6$
5:  $v_{15} = (v_{12}v_3 + v_{13}v_5)v_9$
6:  $v_{16} = (v_{14}v_2 + v_{15}v_4)v_8$
7:  $v_{17} = (v_{14}v_3 + v_{15}v_5)v_{11}$
8:  $v_{18} = v_{16} + v_{17}$
9:  $z = v_{18}$
```

Figure 24: Pseudocode for $P(\mathcal{O}\,|\,\lambda)$ using the notation in (19)

model $\lambda = (A, B, \pi)$. Note that we could also easily write code to implement each of the equality constraints $g_0(x)$ through $g_4(x)$ in (20).

Recall that our goal here is to train the model, that is, we want to find optimal values for the model parameters, where "optimal" means that we maximize $P(\mathcal{O}\,|\,\lambda)$. Believe it or not, we have made progress towards this goal, as the code in Figure 24 gives us a simple (and, more importantly, programmable) version of the function $f(x)$ that appears in the Lagrangian in (21).

To solve the constrained optimization problem defined by the Lagrangian function $L(x, u)$ in (21), we must find vectors x and u that satisfy the system of partial derivative equations

$$\frac{\partial L}{\partial x_i} = 0 \quad\text{and}\quad \frac{\partial L}{\partial u_i} = 0. \tag{22}$$

In fact, it is not difficult to show that any solution to (22) must occur at a saddle point of L with respect to the x and u variables [20].

When deriving the expressions that define a support vector machine (SVM), we can set up the problem as a Lagrangian. This approach is particularly nice in the context of SVMs, since it makes the "kernel trick" a lot less tricky; see Chapter 5 of [17] for all of the gory details.

Next, we want to put the Lagrangian corresponding to the HMM training problem into the form of a neural network. Then we'll be in position to apply backpropagation to train an HMM. Note that this implies that other Lagrange multiplier problems can be solved via backpropagation. For example, SVM training can be based on Lagrange multipliers, so we should also be able to train an SVM using backpropagation.

The Lagrangian in (21) can be computed for any specified values of x and u using the graph in Figure 25, where we have defined $h_i(x) = g_i(x) - 1$. Here, the h_i neurons serve to enforce the constraints, keeping any solution within the feasible region, while the purpose of the f neuron is to enable us to maximize the objective function, subject to the constraints.

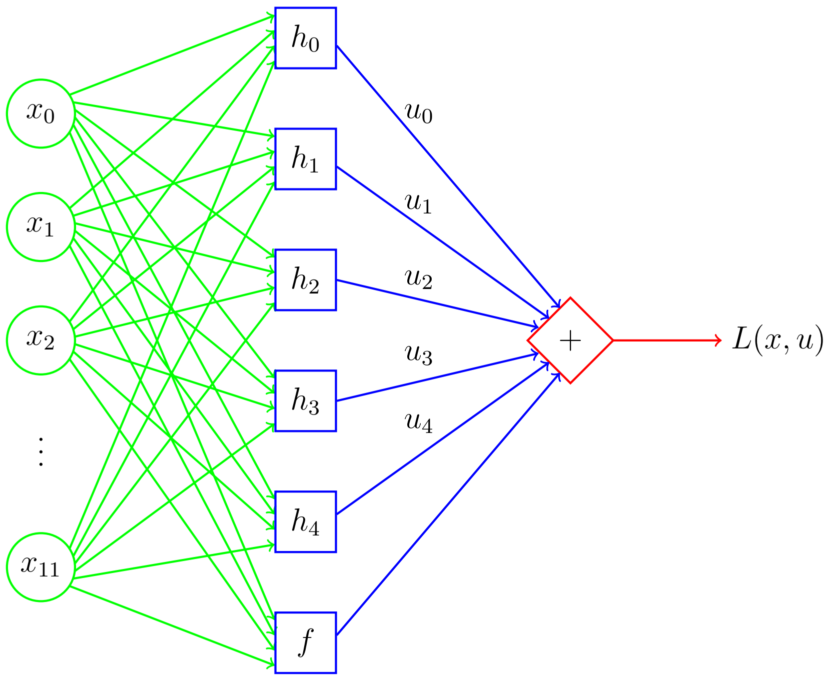


Figure 25: Neural network for $L(x, u)$ in (21) where $h_i(x) = g_i(x) - 1$

Again, to solve the problem in Figure 25, we need to find x and u for which the partial derivative equations given in (22) are satisfied. The way we'll (numerically) solve this problem is fairly straightforward. First, we generate a program to compute the gradient of L at any specified point (x, u). For this step, we use reverse mode automatic differentiation, as discussed in Section 5. Then we'll initialize (x, u) and compute the gradient. In the unlikely event that the partials derivatives are all 0, we're done; otherwise, we need to update the point (x, u) and compute the gradient at this new point. A key idea in backpropagation is that we make smart modifications when we update (x, u) so as to speed convergence. Gradient descent enables us to update the point (x, u) by moving in the direction that will provide the most improvement.[18] That is, we adjust the x and u values in the steepest direction, according to the gradient.

[18]Gradient descent for the Lagrange multiplier problem is slightly complicated by the fact that a solution occurs at a saddle point.