Alphabet Soup of Deep Learning Topics

Mark Stamp*
Department of Computer Science
San Jose State University

November 19, 2019

1 Introduction

In this tutorial, we discuss a variety of topics that are related to deep learning. Most topics are covered relatively briefly, and in all cases, references for further reading are provided. The material here is designed to serve as a supplement to [42], which provides an introduction to artificial neural networks, and includes a detailed discussion of backpropagation. The material in [42] is itself a supplement to the highly regarded (at least by me) book [40].

2 CNN

In this section, we provide an introduction to one of the most important and widely used learning techniques—convolutional neural networks (CNN). After a brief overview, we introduce discrete convolutions with the focus on their specific application to CNNs. We then consider a simplified example that serves to illustrate various aspects of CNNs.

2.1 Overview

Generically, artificial neural networks (ANNs) use fully connected layers. A fully connected layer can deal effectively with correlations between any points within the training vectors, regardless of whether those points are close together, far

*Email: mark.stamp@sjsu.edu. This is yet another supplement to that fine book, *Introduction to Machine Learning with Applications in Information Security*.

apart, or somewhere in between. In contrast, a CNN, is designed to deal with local structure—a convolutional layer cannot be expected to perform well when crucial information is not local. A key benefit of CNNs is that convolutional layers can be trained much more efficiently than fully connected layers.

For images, most of the important structure (edges and gradients, for example) is local. Hence, CNNs would seem to be an ideal tool for image analysis and, in fact, CNNs were developed for precisely this problem. However, CNNs have performed well in a variety of other problem domains. In general, any problem for which there exists a data representation where local structure predominates is a candidate for a CNN. In addition to images, local structure is crucial in fields such as text analysis and speech analysis, for example.

2.2 Convolution and CNNs

A discrete convolution is a sequence that is itself a composition of two sequences, and is computed as a sum of pointwise products. Let $c = x * y$ denote the convolution of sequences $x = (x_0, x_1.x_2, \ldots)$ and $y = (y_0, y_1.y_2, \ldots)$. Then the kth element of the convolution is given by

$$c_k = \sum_{k=i+j} x_i y_j = \sum_i x_i y_{k-i}$$

We can view this process as x being a "filter" (or kernel) that is applied to the sequence y over a sliding window.

For example, if $x = (x_0, x_1)$ and $y = (y_0, y_1, y_2, y_3, y_4)$, we find

$$c = x * y = \big(x_0y_1 + x_1y_0, x_0y_2 + x_1y_1, x_0y_3 + x_1y_2, x_0y_4 + x_1y_3\big)$$

If we reverse the order of the elements of x, then we have

$$c = \big(x_0y_0 + x_1y_1, x_0y_1 + x_1y_2, x_0y_2 + x_1y_3, x_0y_3 + x_1y_4\big)$$

which is, perhaps, a slightly more natural and intuitive way to view the convolution operation.

Again, we can view x as a filter that is applied to the sequence y. Henceforth, we'll define this filtering operation as convolution with the order of the elements of the filter reversed. For example, suppose that we apply the filter $x = (1, -2)$ to the sequence $y = (0, 1, 2, 3, 4)$. In this case, the convolution gives us

$$\begin{aligned} c = x * y &= \big(x_0y_0 + x_1y_1, x_0y_1 + x_1y_2, x_0y_2 + x_1y_3, x_0y_3 + x_1y_4\big) \\ &= \big(1 \cdot 0 - 2 \cdot 1, 1 \cdot 1 - 2 \cdot 2, 1 \cdot 2 - 2 \cdot 3, 1 \cdot 3 - 2 \cdot 4\big) \\ &= \big(-2, -3, -4, -5\big) \end{aligned}$$

We can define an analogous filtering (or discrete convolution) operation in two or three dimensions. For the two-dimensional case, suppose that $A = \{a_{ij}\}$ is an $N \times M$ matrix representing an image and $F = \{f_{ij}\}$ is an $n \times m$ filter. Let $C = \{c_{ij}\}$ be the convolution of F with A. As in the one-dimensional case, we denote this convolution as $C = F * A$. In this two-dimensional case, we have

$$c_{ij} = \sum_{k=0}^{n-1} \sum_{\ell=0}^{m-1} f_{k,\ell} a_{i+k,j+\ell}$$

where $i = 0, 1, \ldots, N-n$ and $j = 0, 1, \ldots, M-m$. That is, we simply apply the filter F at each offset of A to create the new—and slightly smaller—matrix that we denote as C. The three-dimensional case is completely analogous to the two-dimensional case.

We could simply define filters as we see fit, with each filter designed to correspond to a specific feature.[1] But since we are machine learning aficionados, for CNNs, we'll let the data itself determine the filters. Therefore, training a CNN can be viewed as determining filters, based on the training data. As with any respectable neural network, we can train CNNs via backpropagation.

Suppose that A represents an image and we train a CNN on the image A. Then the first convolutional layer is trained directly on the image. The filters determined at this first layer will correspond to fairly intuitive features, such as edges, basic shapes, and so on. We can then apply a second convolutional layer, that is, we apply a similar convolutional process, but the output of the first convolutional layer is the input to this second layer. At the second layer, filters are trained based on features of features. Perhaps not surprisingly, these second layer filters correspond to more abstract features of the original image A, such as the "texture." We can repeat this convolution of convolutions step again and again, at each layer obtaining filters that correspond to features representing a higher degree of abstraction, as compared to the previous layer. The final layer of a CNN is not a convolution layer, but is instead a typical fully-connected layer that can be used to classify based on complex image characteristics (e.g., "cat" versus "dog"). In addition, so-called pooling layers can be used between some of the convolutional layers. Pooling layers are simple—no training is involved—and serve primarily to reduce the dimensionality of the problem. Below, we'll give a simple example that includes a pooling layer.

In addition to having multiple convolutional layers, at each layer we can (and generally will) stack several convolutions on top of each other. These filters are all initialized randomly, so they can all learn different features. In fact, for a typical color image, the image itself can be viewed as consisting of three

[1]We'll see examples of filters applied to simple images in Section 2.3.

layers, corresponding to the R, G, and B components in the RGB color scheme. Hence, for color images, the filters for the first convolutional layer will be three dimensional, while subsequent convolutional layers can—and, typically, will—be three dimensional as well, due to the stacking of multiple convolutions/filters at each layer. For simplicity, in our example we'll only consider a black-and-white two-dimensional image, and we'll only apply one convolution at each layer.

Before considering a simple example, we note that there are advantages of CNNs that are particularly relevant in the case of image analysis. For a generic neural network, each pixel would typically be treated as a separate neuron, and for any reasonable size of image, this would result in a huge number of parameters, making training impractical. In contrast, at the first layer of a CNN, each filter is applied over the entire image, and at subsequent layers, we apply filters over the entire output of the previous layer. One effect of this approach is that it greatly reduces the number of parameters that need to be learned. Furthermore, by sliding the filter across the image as a convolution, we obtain a degree of translation invariance, i.e., we can detect image features that appears at different offsets. This can be viewed as reducing the overfitting that would otherwise likely occur.

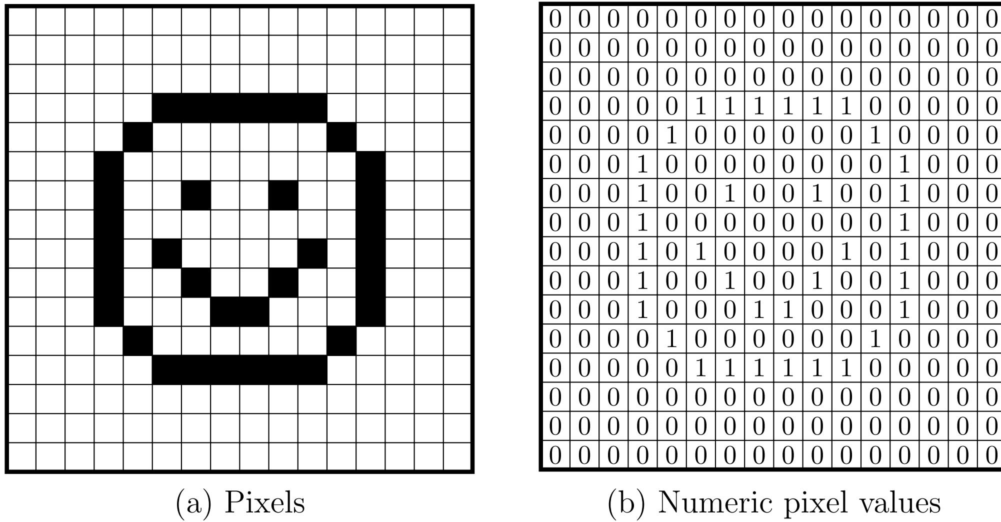
The bottom line is that CNNs represent an efficient and effective technique that was developed specifically for image analysis. However, CNNs are not restricted to image data, and can be useful in any problem domain where local structure is dominant.

2.3 Example

Now we turn our attention to a simple example that serves to illustrate some of the points discussed above. Suppose that we're dealing with black-and-white images, where each pixel is either 0 or 1, with 0 representing white and 1 representing black.[2] Further, suppose that the black-and-white images under consideration are 16×16 pixels in size. An example of such an image appears in Figure 1.

In Figure 2, we give some 3×3 filters. For example, the output of the filter in Figure 2 (a) is maximized when it aligns with a diagonal segment. Figure 3 shows the result of applying the convolution represented by the filter in Figure 2 (a) to the smiley face image in Figure 1.

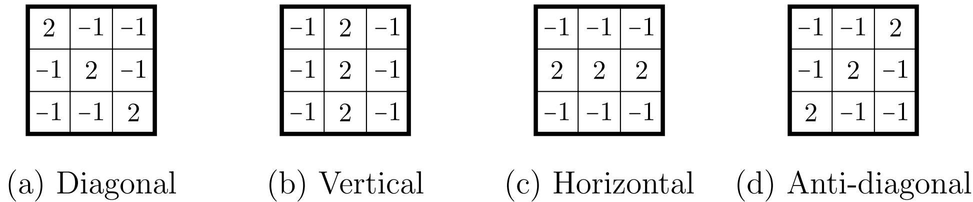
[2]Color and grayscale images are more complex. For grayscale, a nonlinear encoding (i.e., gamma encoding) is employed, so as to make better use of the range of values available. For color images, the RGB (red, green, and blue, respectively) color scheme implies that each pixel is represented by 24 bits (in an uncompressed format), in which case convolutional filters can be viewed as operating over a three-dimensional box that is three bytes deep.



0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	1	1	1	1	1	1	0	0	0	0	0
0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0
0	0	0	1	0	0	0	0	0	0	0	0	1	0	0	0
0	0	0	1	0	0	1	0	0	1	0	0	1	0	0	0
0	0	0	1	0	0	0	0	0	0	0	0	1	0	0	0
0	0	0	1	0	1	0	0	0	0	1	0	1	0	0	0
0	0	0	1	0	0	1	0	0	1	0	0	1	0	0	0
0	0	0	1	0	0	0	1	1	0	0	0	1	0	0	0
0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0
0	0	0	0	0	1	1	1	1	1	1	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

(a) Pixels (b) Numeric pixel values

Figure 1: A 16×16 black-and-white image



2	−1	−1
−1	2	−1
−1	−1	2

(a) Diagonal

−1	2	−1
−1	2	−1
−1	2	−1

(b) Vertical

−1	−1	−1
2	2	2
−1	−1	−1

(c) Horizontal

−1	−1	2
−1	2	−1
2	−1	−1

(d) Anti-diagonal

Figure 2: Examples of filters

We note that the for the convolution in Figure 3, the maximum value of 6 does indeed occur only at the three offsets where the (main) diagonal segments are all black and the off-diagonal elements are all white. These maximum values correspond to convolutions over the red boxes in Figure 4.

In a CNN, so-called pooling layers are often intermixed with convolutional layers. As with a convolutional layer, in a pooling layer, we slide a window of a fixed size over the image. But whereas the filter in a convolutional layer is learned, in a pooling layer an extremely simple filter is specified and remains unchanged throughout the training. As the name implies, in *max pooling*, we simply take the the maximum value within the filter window. An illustration of max pooling is given in Figure 5.

Instead of a max pooling scheme, sometimes *average pooling* is used. In any case, pooling can be viewed as a downsampling operation, which has the effect of reducing the dimensionality, and thus easing the computational burden of

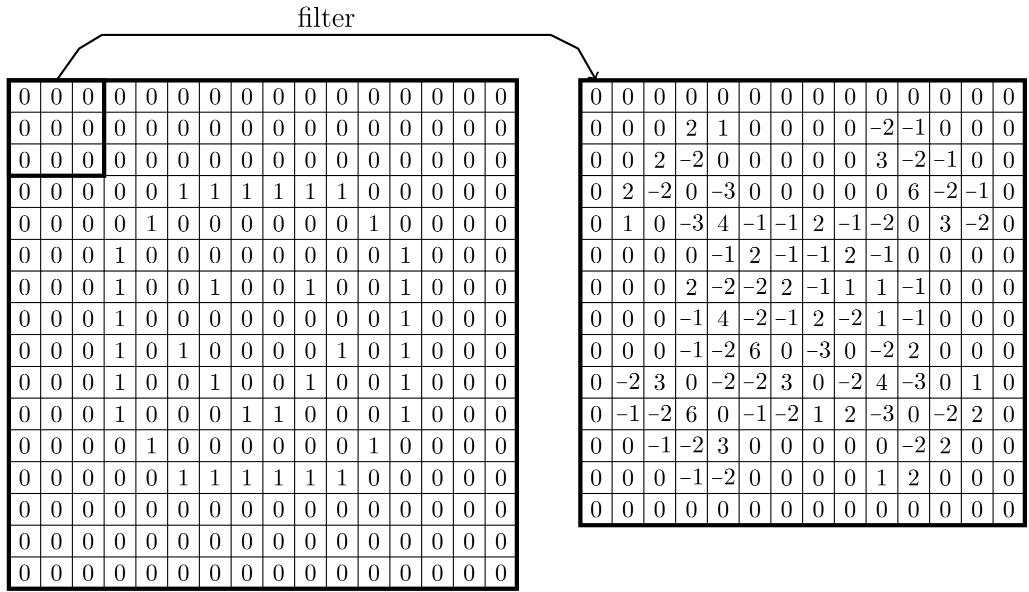


Figure 3: First convolutional layer (3×3 filter from Figure 2 (a))

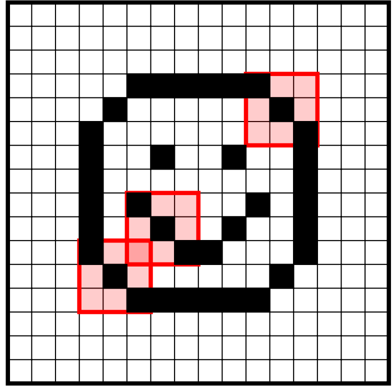


Figure 4: Maximum convolution values in Figure 3

training subsequent convolutional layers.[3] To increase the downsampling effect, pooling usually uses non-overlapping windows. Note that the dimensionality reduction of pooling could also be achieved by a convolutional layer that uses a larger stride through the data, and in [37], for example, it is claimed that such an approach results in no loss in accuracy for the resulting CNN.

[3]It is also sometimes claimed that pooling improves certain desirable characteristics of CNNs, such as translation invariance and deformation stability. However, this is disputed, and the current trend seems to clearly be in the direction of fully convolutional architectures, i.e., CNNs with no pooling layers [35].

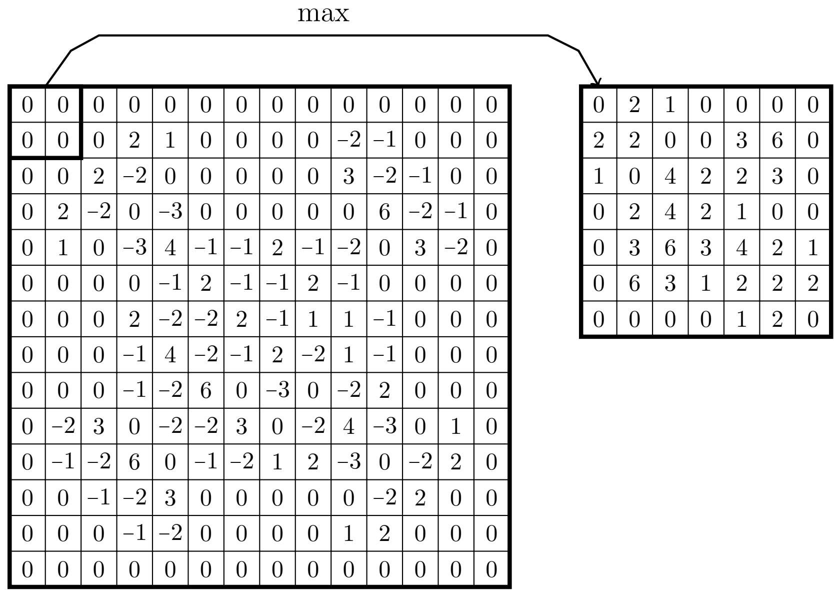


Figure 5: Max pooling layer (2×2, non-overlapping)

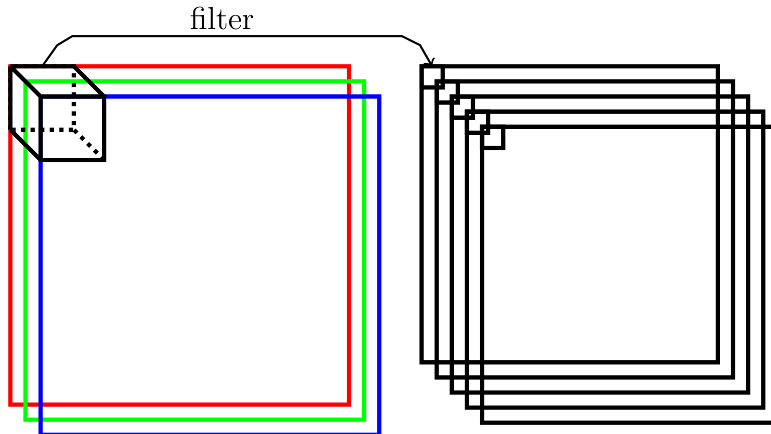


Figure 6: First convolutional layer for RGB image

An illustration of the first convolutional layer for a color image is given in Figure 6. In this case, a 3-dimensional filter is applied over the R, G, and B components in the RGB color scheme. The example in Figure 6 is meant to indicate that five different filters are being trained. Since each filter is initialized randomly, they can all learn different features. At the second convolutional layer, we can again train 3-dimensional filters, based on the output of the first convolutional layer. This process is repeated for any additional convolutional layers.

There are several possible ways to visualize the filters in convolutional layers. For example, in [47], a de-convolution technique is used to obtain the results in Figure 7. Here, each row is a randomly selected filter and the columns, from left to right, correspond to training epochs 1, 2, 5, 10, 20, 30, 40, and 64. From layer 4, we see that the training images must be faces. In general, it is apparent that the filters are learning progressively more abstract features as the layer increases.

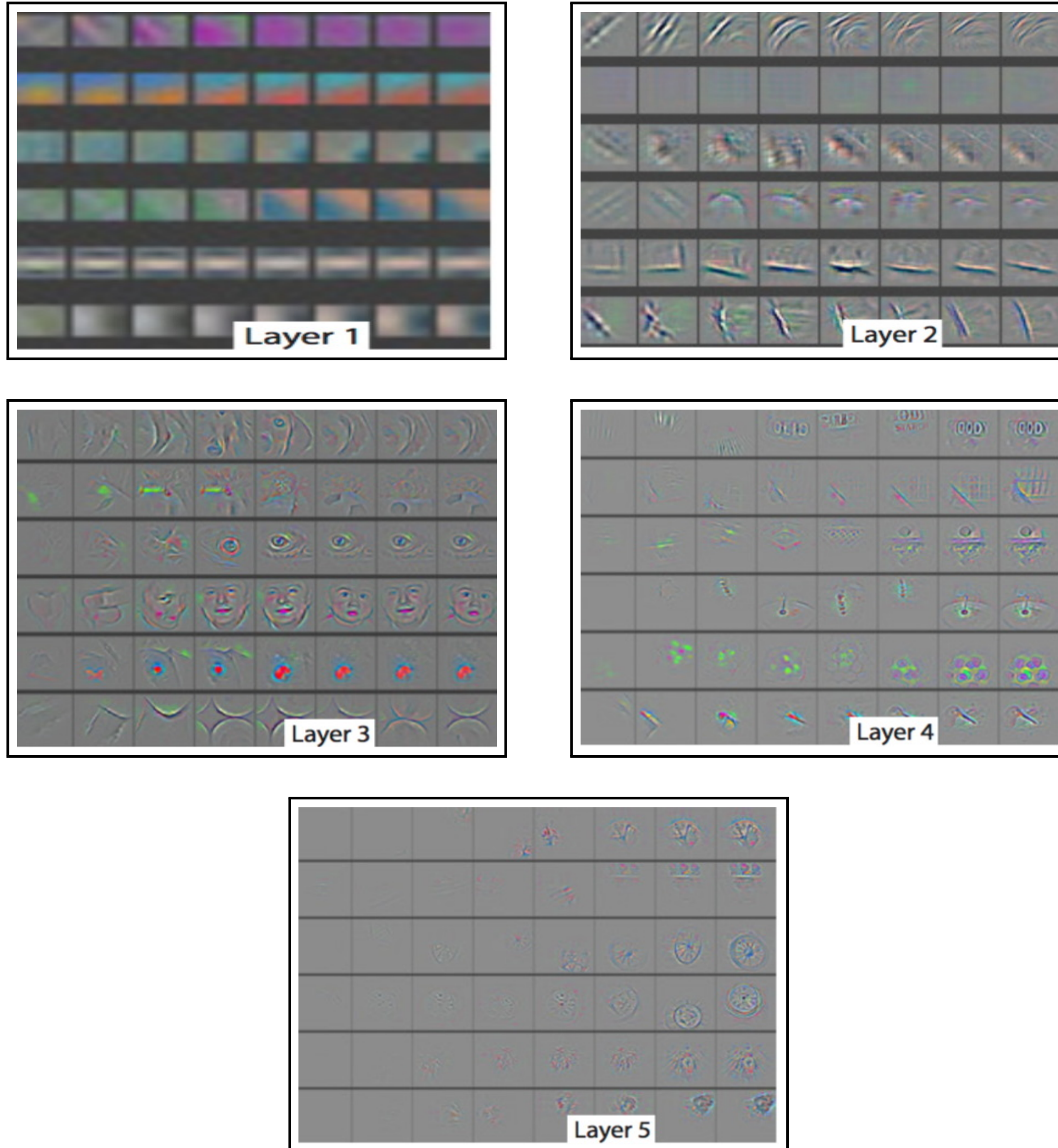


Figure 7: Visualizing convolutions [47]

A fairly detailed discussion of CNNs can be found at [17], while the paper [5] provides some interesting insights. For a more intuitive discussion, see [16], and if you want to see lots of nice pictures, take a look at [6]. More details on convolutions can be found in [32].

3 RNN

An example of a feedforward neural network with two hidden layers is give in Figure 8. This type of neural network has no "memory" in the sense that each input vector is treated independently of other input vectors. Hence, such a feedforward network is not well suited to deal with sequential data.

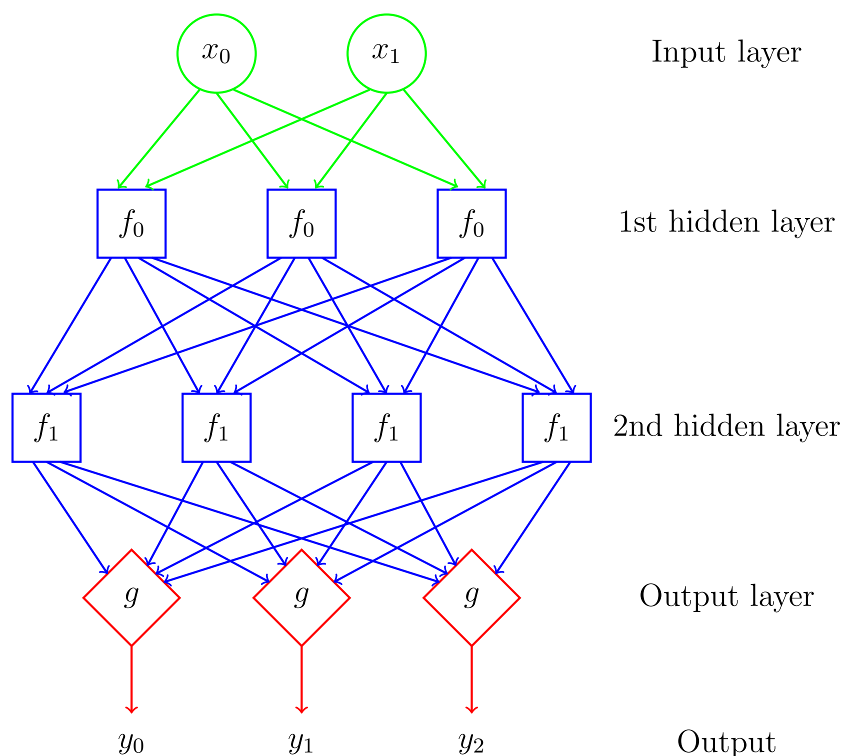


Figure 8: Feedforward neural network with two hidden layers

In some cases, it is necessary for a classifier to have memory. For example, suppose that we want to tag parts of speech in English text (i.e., noun verb, and so on). This is not feasible if we only look at words in isolation—for example, the word "all" can be an adjective, adverb, noun, or even a pronoun, and the only way to determine which is the case is to consider the context in which it

occurs. A recurrent neural network (RNN) provides a way to add memory (or context) to a feedforward neural network.

To convert a feedforward neural network into an RNN, we treat the output of the hidden states as another input. For the neural network in Figure 8, the corresponding generic RNN is illustrated in Figure 9. The structure in Figure 9 implies that there is a time-step involved, that is, we train (and score) based on a sequence of input vectors. Of course, we cannot consider infinite sequences, and even if we could, the influence of feature vectors that occurred far back in time is likely to be minimal.

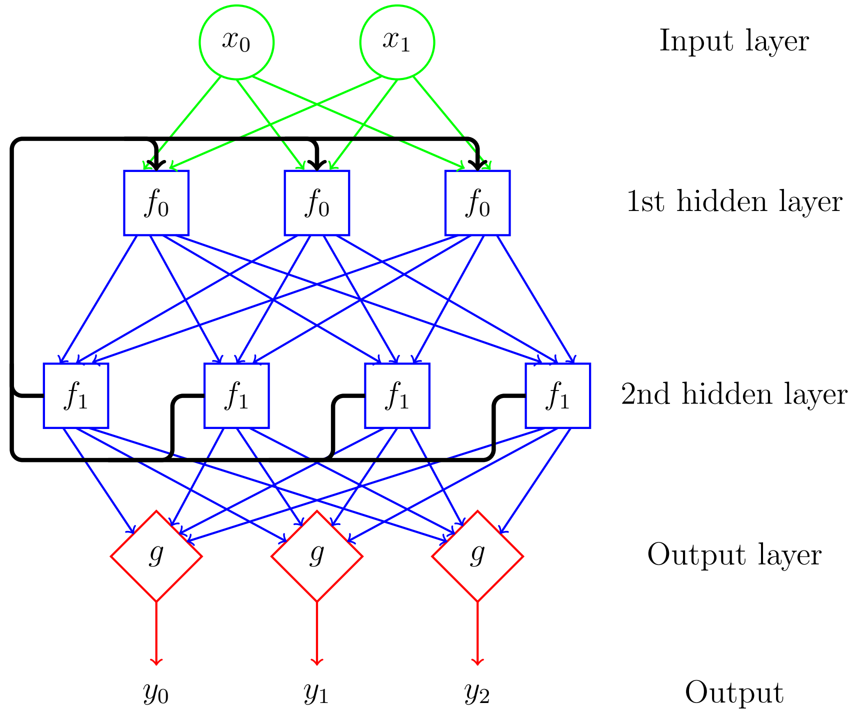


Figure 9: Network in Figure 8 as an RNN

The RNN in Figure 9 can be "unrolled," as illustrated in Figure 10. Note that in this case, we use f to represent the hidden layer or layers, while the notation X_t is used to represent (x_0, x_1) at time step t from un-unrolled RNN in Figure 9 and, similarly, Y_t corresponds to (y_0, y_1, y_2) at time t. From the unrolled form, it is clear that any RNN can be treated as a special case of a feedforward neural network, where the intermediate hidden layers (f in our notation) all have identical structure and weights. We can take advantage of this special structure to efficiently train an RNN using a (slight) variant of backpropagation, known as backpropagation through time (BPTT).

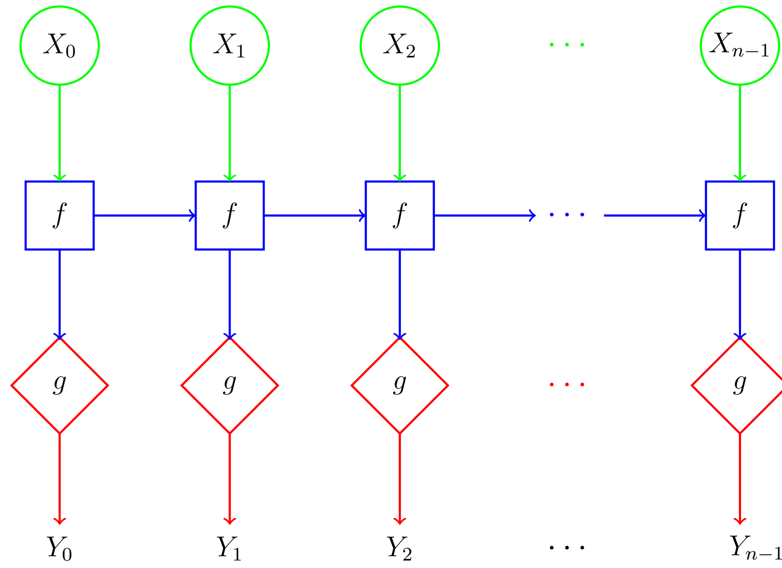
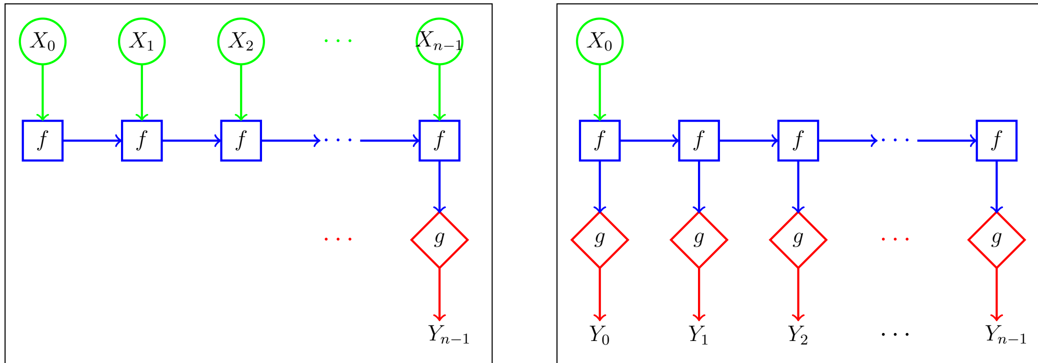


Figure 10: Unrolled RNN (sequence-to-sequence model)

Before briefly turning our attention to BPTT, we illustrate some variants of a generic RNN. An RNN such as that illustrated in Figure 10 is known as a sequence-to-sequence model, since each input sequence $(X_0, X_1, \ldots, X_{n-1})$ corresponds to an output sequence $(Y_0, Y_1, \ldots, Y_{n-1})$. In Figure 11 (a), we have illustrated a many-to-one example of an RNN, that is, the case where an input sequence of the form $(X_0, X_1, \ldots, X_{n-1})$ corresponds to the single output Y_{n-1}. At the other extreme, Figure 11 (b) illustrates a one-to-many RNN, where the single input X_0 corresponds to the output sequence $(Y_0, Y_1, \ldots, Y_{n-1})$.



(a) Many-to-one RNN (b) One-to-many RNN

Figure 11: Variants of the generic RNN in Figure 10

A many-to-one model might be appropriate for part-of-speech tagging, for example, while a one-to-many RNN could be used for music generation. An example of an application where a sequence-to-sequence (or many-to-many) RNN would be appropriate is machine translation. There are numerous possible variants of the sequence-to-sequence RNN. Also, note that a feedforward neural network, such as that in Figure 8, can be viewed as a one-to-one RNN.

Multi-layer RNNs can also be considered. This can be viewed as training multiple RNNs simultaneously, with the first RNN trained on the input data, the second RNN trained on the hidden states of the first RNN, and so on. A two-layer (sequence-to-sequence) RNN is illustrated in Figure 12. Of course, more layers are possible, but the training complexity will increase, and hence only "shallow" RNN architectures (in terms of the number of layers) are generally considered.

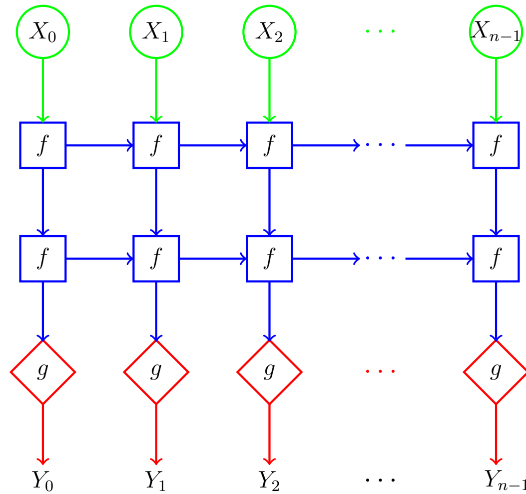


Figure 12: Two layer RNN

3.1 BPTT

RNNs can be viewed as neural networks that are designed specifically for time series or other sequential data. With an RNN, the number of parameters is reduced so as to ease the training burden. This situation is somewhat analogous to CNNs, which are designed to efficiently deal with local structure (e.g., in images). That is, both CNNs and RNNs serve to make training more efficient—as compared to generic feedforward neural networks—for specific classes of problems. Backpropagation through time (BPTT) is simply an ever-so-slight variation on backpropagation that is optimized for training RNNs.

In Figure 13 we give a detailed view of a many-to-one (actually, two-to-one) RNN. In this case, we see that the 10 weights, $(w_0, w_1, \ldots, w_9)$ must be determined via training.

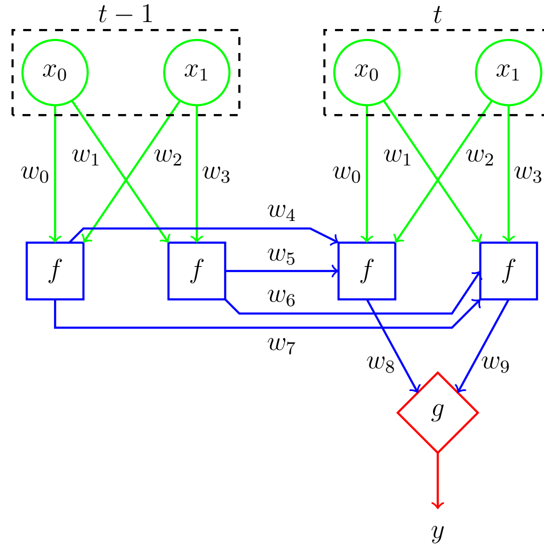


Figure 13: Simple RNN example

In Figure 14 we give a neural network that is essentially the fully connected version of the RNN in Figure 13. Note that in this fully-connected version, there are 20 parameters to be determined. In an RNN, we assume that the data represents sequential input and hence the reduction in the number of weights is justified, since we are simply eliminating from consideration cases where the past is influenced by the future.[4]

Next, we consider gradient issues that plague generic RNNs. Then we discuss modified RNN architectures that help to reduce the effect of these gradient-related problems.

3.2 Vanishing and Exploding Gradients

In Figure 15 (a), we have illustrated an RNN with a single neuron at each layer, where f is the activation function, X_t is the input, and E_t is the error at step t. Figure 15 (b) is the unrolled version of this same RNN, where Z_t is the composition of functions that occurs at step t.

[4]Obviously, the inventors of RNNs were not familiar with *Back to the Future* or *Star Trek*, both of which conclusively demonstrate that the future can have a large influence on the past.

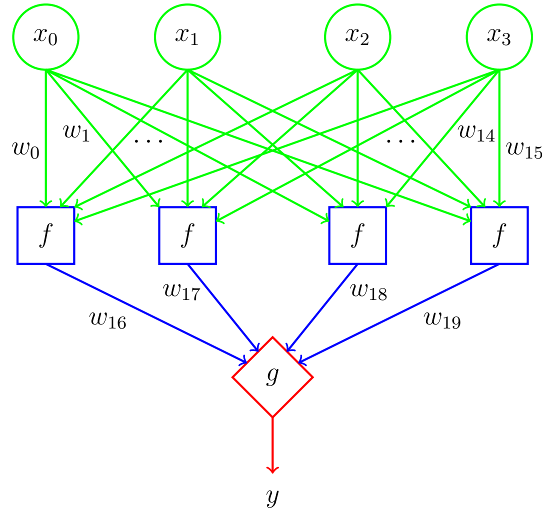
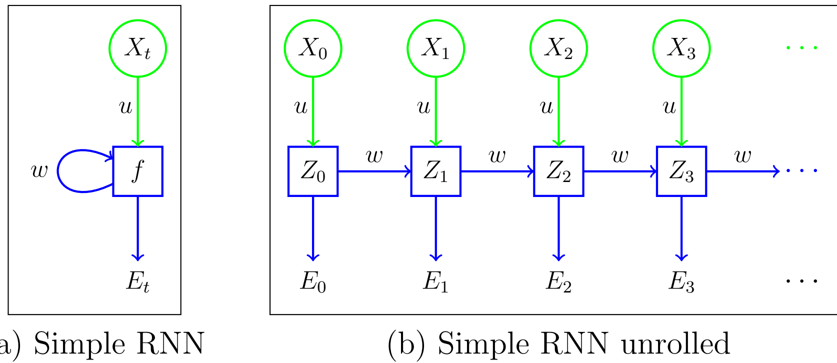


Figure 14: Fully connected analog of Figure 13



(a) Simple RNN (b) Simple RNN unrolled

Figure 15: Simple RNN and its unrolled version

Typically, we would concatenate the input to the internal state of the RNN, but for simplicity, we'll just sum the states. Then we have

$$Z_t = f(wZ_{t-1} + uX_t) \tag{1}$$

where we define $Z_{-1} = 0$. To further simplify, we'll also assume that the weights u and w are scalars.

Note that we can unroll each Z_t further to obtain

$$\begin{aligned}
Z_0 &= f(uX_0) \\
Z_1 &= f(wZ_0 + uX_1) \\
&= f(wf(uX_0) + uX_1)
\end{aligned}$$

$$
\begin{aligned}
Z_2 &= f(wZ_1 + uX_2) \\
&= f(wf(wZ_0 + uX_1) + uX_2) \\
&= f(wf(wf(uX_0) + uX_1) + uX_2) \\
Z_3 &= f(wZ_2 + uX_3) \\
&= f(wf(wZ_1 + uX_2) + uX_3) \\
&= f(wf(wf(wZ_0 + uX_1) + uX_2) + uX_3) \\
&= f(wf(wf(wf(uX_0) + uX_1) + uX_2) + uX_3)
\end{aligned}
$$

However, below we'll use the expressions in equation (1).

In backpropagation, we compute the gradient of the error terms E_t. It is clear from Figure 15 (b) that the gradient of E_t will include $\partial Z_t/\partial w$. Computing these partial derivatives, we find

$$
\begin{aligned}
\frac{\partial Z_1}{\partial w} &= f'(wZ_0 + uX_1)\frac{\partial(wZ_0 + uX_1)}{\partial w} \\
&= Z_0 f'(wZ_0 + uX_1) \\
\frac{\partial Z_2}{\partial w} &= f'(wZ_1 + uX_2)\frac{\partial(wZ_1 + uX_2)}{\partial w} \\
&= f'(wZ_1 + uX_2)\Big(Z_1 + w\frac{\partial Z_1}{\partial w}\Big) \\
&= Z_1 f'(wZ_1 + uX_2) + wf'(wZ_1 + uX_2)\frac{\partial Z_1}{\partial w} \\
&= Z_1 f'(wZ_1 + uX_2) + wZ_0 f'(wZ_0 + uX_1)f'(wZ_1 + uX_2) \\
\frac{\partial Z_3}{\partial w} &= f'(wZ_2 + uX_3)\frac{\partial(wZ_2 + uX_3)}{\partial w} \\
&= f'(wZ_2 + uX_3)\Big(Z_2 + w\frac{\partial Z_2}{\partial w}\Big) \\
&= Z_2 f'(wZ_2 + uX_3) + wf'(wZ_2 + uX_3)\frac{\partial Z_2}{\partial w} \\
&= Z_2 f'(wZ_2 + uX_3) \\
&\quad + wZ_1 f'(wZ_1 + uX_2)f'(wZ_2 + uX_3) \\
&\quad + w^2 Z_0 f'(wZ_0 + uX_1)f'(wZ_1 + uX_2)f'(wZ_2 + uX_3)
\end{aligned}
$$

In general,

$$
\frac{\partial Z_t}{\partial w} = \sum_{k=0}^{t-1} w^{t-1-k} Z_k \prod_{j=k}^{t-1} f'(wZ_j + uX_{j+1}) \tag{2}
$$

holds for all $t \geq 1$.

Now we consider the effect of the activation function f that appears in equation (2). A popular choice for f is the sigmoid function

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

It is easily verified that the first derivative of $\sigma(x)$ can be written as

$$\sigma'(x) = \sigma(x)\bigl(1 - \sigma(x)\bigr).$$

The graphs of $\sigma(x)$ and $\sigma'(x)$ appear in Figure 16. From these graphs, we see that $\sigma(x) < 1.0$ and $\sigma'(x) < 0.25$ hold for all x.

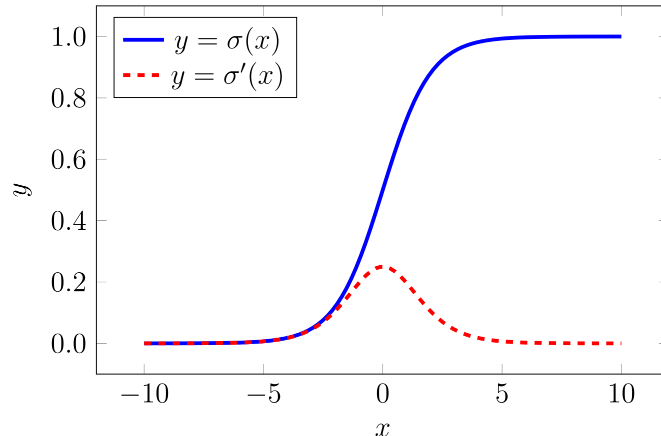


Figure 16: Graph of sigmoid function $\sigma(x)$ and its derivative $\sigma'(x)$

The terms on the right-hand side in equation (2) include products of the derivative of the activation function. Observe that the number of f' terms increases the further back in time that we go.

Assuming that the weights are initialized so that $w \leq 1$, and assuming that we choose the sigmoid as our activation function f, then during backpropagation, the partial derivative terms in (2) "vanish," in the sense of tending to 0 exponentially the further back in time that we go. Consequently, even though we might design our RNN to look far back in time, in reality the distant layers will have no effect. This problem can occur in deep neural networks as well, that is, the deeper layers might have no effect, due to a *vanishing gradient*.[5]

[5]The vanishing gradient problem is somewhat reminiscent of a problem that occurs when training a hidden Markov model (HMM) using Baum-Welch re-estimation. In the Baum-Welch algorithm, products of probabilities are computed, which tend to 0 exponentially. To avoid underflow in Baum-Welch, log probabilities and scaling can be used [39].

We could try to prevent the gradient from vanishing by initializing the weights to large values. While this can eliminate the vanishing gradient problem, unless we are very lucky in our choice of initial values, we will now likely have an *exploding gradient*, that is, the gradient will grow exponentially larger the further back in time that we go. This growth in the gradient is due to the exponentiated w term that appears in equation (2).

We can also have cases where the gradient oscillates wildly. Avoiding vanishing, exploding and, more generally, unstable gradients is challenging. In fact, it has recently been shown that many of the techniques proposed to reduce the problems caused by these types of gradient issues may be considerably less effective than previously believed [33].

One simple way to reduce the effect of a such gradient issues is to limit how far back in time we backpropagate. For example, in Figure 17 we illustrate a case where we only allow the gradient to back propagate two time-steps. Such a truncated BPTT (TBPTT) is somewhat analogous to using a minibatch during backpropagation in a feedforward neural network. By this logic, limiting TBPTT to a single time-step is the analog of stochastic gradient descent.

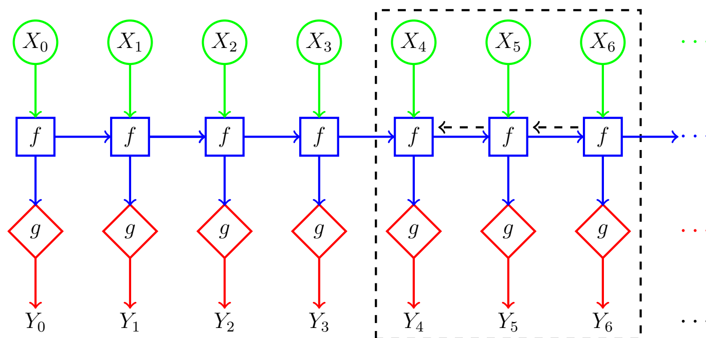


Figure 17: Backpropagation limited to two time steps

As an aside, we note that a hidden Markov model (HMM) of order one is somewhat analogous to the case where we limit BPTT to a single time step. Typically, we train HMMs using a hill climb technique (i.e., Baum-Welch re-estimation), but HMMs can be trained using gradient ascent [1, 42] which, perhaps, makes the connection to TBPTT even stronger.

Next, we discuss two RNN architectures that are designed to deal with vanishing gradients. First, we consider long short-term memory (LSTM) networks in some detail and we then briefly discuss a variant of LSTM. In fact, a vast number of variants of the LSTM architecture have been developed. However, according to an extensive empirical study [9], "none of the variants can improve upon the standard LSTM architecture significantly."

3.3 LSTM

In addition to being a tongue twister, long short-term memory (LSTM) networks are a class of RNN architectures that are designed to deal with long-range dependencies. That is, LSTM can deal with "gaps" between the appearance of a feature and the point at which it is needed by the model [9]. The claim to fame of LSTM is that it can reduce the effect of a vanishing gradient, which is what enables such models to account for longer-range dependencies [12].

Before outlining the main ideas behind LSTM, we note that the LSTM architecture has been one of the most commercially successful learning techniques ever developed. Among many other applications, LSTMs have been used in Google Allo [19], Google Translate [45], Apple's Siri [21], and Amazon Alexa [10]. However, recently the dominance of LSTM may have begun to wane. ResNet has been shown to have theoretical advantages over LSTM, and it outperforms LSTM in a wide range of applications [33].

Figure 18 illustrates an LSTM. The obvious difference from a generic vanilla RNN is that an LSTM has two lines entering and exiting each state. As in a standard RNN, one of these lines represents the hidden state, while the second line is designed to serve as a gradient "highway" during backpropagation. In this way, the gradient can "flow" much further back with less chance that it will vanish along the way.

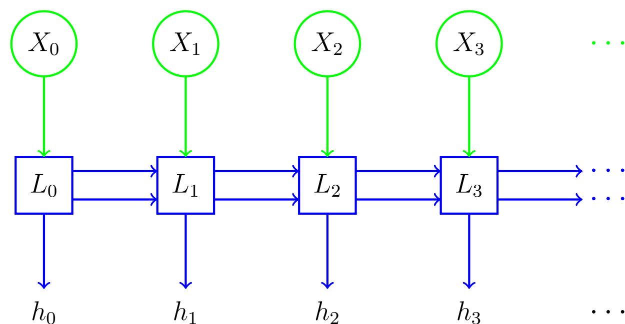


Figure 18: LSTM

In Figure 19 we expand one of the LSTM cells L_t that appear in Figure 18. Here, σ is the sigmoid function, τ is the hyperbolic tangent (i.e., tanh) function, the operators "$\times$" and "$+$" are pointwise multiplication and addition, respectively, while "$\|$" indicates concatenation of vectors. The vector i_t is the "input" gate, f_t is the "forget" gate, and o_t is the "output" gate. The vector g_t is an intermediate gate and does not have a cool name, but is sometimes referred to as the "gate" gate [22], which, come to think of it, is especially cool. We have much more to say about these gates below.

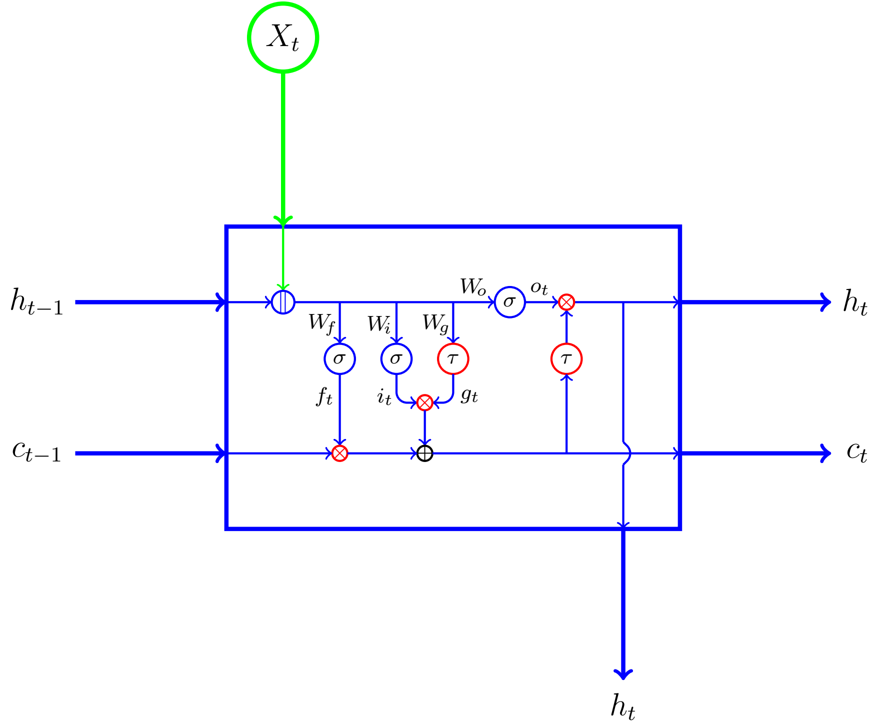


Figure 19: One timestep of an LSTM

The gate vectors that appear in Figure 19 are computed as

$$f_t = \sigma\left(W_f\begin{pmatrix} h_{t-1} \\ X_t \end{pmatrix} + b_f\right) \qquad g_t = \tau\left(W_g\begin{pmatrix} h_{t-1} \\ X_t \end{pmatrix} + b_g\right)$$

$$i_t = \sigma\left(W_i\begin{pmatrix} h_{t-1} \\ X_t \end{pmatrix} + b_i\right) \qquad o_t = \sigma\left(W_o\begin{pmatrix} h_{t-1} \\ X_t \end{pmatrix} + b_o\right)$$

while the outputs are

$$\begin{aligned} c_t &= f_t \otimes c_{t-1} \oplus i_t \otimes g_t \\ h_t &= o_t \otimes \tau(c_t) \end{aligned}$$

where "$\otimes$" is pointwise multiplication and "$\oplus$" is the usual pointwise addition. Note that each of the weight matrices is $n \times 2n$.

In matrix form, ignoring the bias terms b, we have

$$\begin{pmatrix} i_t \\ f_t \\ o_t \\ g_t \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tau \end{pmatrix} W \begin{pmatrix} h_{t-1} \\ X_t \end{pmatrix}$$

where X_t and h_{t-1} are column vectors of length n, and W is the $4n \times 2n$ weight matrix

$$W = \begin{pmatrix} W_i \\ W_f \\ W_o \\ W_g \end{pmatrix}$$

Further, each of the gates i_t, f_t, o_t, and g_t is a column vectors of length n. Recall that the sigmoid σ squashes its input to be within the range of 0 to 1, whereas the tanh function τ gives output within the range of -1 to $+1$.

To highlight the intuition behind LSTM, we follow a similar approach as that given in the excellent presentation [22]. Specifically, we focus on the extreme cases, that is, we assume that the output of each sigmoid σ is either 0 or 1, and each hyperbolic tangent τ is either -1 or $+1$. Then the forget gate f_t is a vector of 0s and 1s, where the 0s tell us the elements of c_{t-1} that we'll forget and the 1s indicate the elements to remember. In the middle section of the diagram, the input gate i_t and gate gate g_t together determine which elements of c_{t-1} to increment or decrement. Specifically, when element j of i_t is 1 and element j of g_t is $+1$, we increment element j of c_{t-1}. And if element j of i_t is 1 and element j of g_t is -1, then we decrement element j of c_{t-1}. This serves to emphasize or de-emphasize particular elements in the new-and-improved cell state c_t. Finally, the output gate o_t determines which elements of the cell state will become part of the hidden state h_t. Note that the hidden states h_t is fed into the output layer of the LSTM. Also note that before the cell states are operated on by the output gate, the values are first squeezed down to be within the range of -1 to $+1$ by the τ function.

Of course, in general, the LSTM gates are not simply counters that increment or decrement by 1. But, the intuition is the same, that is, the gates keep track of incremental changes, thus allowing relevant information to flow over long distances via the cell state. In this way, LSTM negates some of the limitations caused by vanishing gradients.

3.4 GRU

As mentioned above, there are large number of variants of the basic LSTM architecture. Most such variants are slight variants, with only minor changes from a standard LSTM. A gated recurrent unit (GRU), on the other hand, is a fairly radical departure from an LSTM. Although the internal state of a GRU is somewhat complex and, perhaps, less intuitive than that of an LSTM, there are fewer parameters in a GRU, and hence it is easier to train a GRU, and less training data is required. The wiring diagram for a GRU is given in Figure 20.

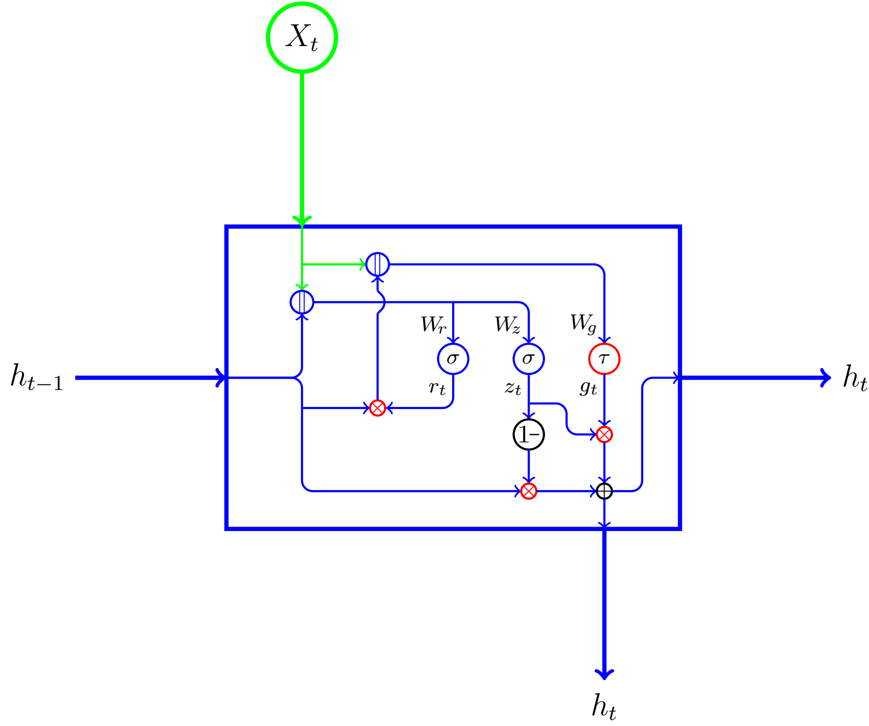


Figure 20: One timestep of a GRU

The gate vectors that appear in Figure 19 are computed as

$$z_t = \sigma\left(W_z\begin{pmatrix} h_{t-1} \\ X_t \end{pmatrix} + b_z\right)$$

$$r_t = \sigma\left(W_r\begin{pmatrix} h_{t-1} \\ X_t \end{pmatrix} + b_r\right)$$

$$g_t = \tau\left(W_g\begin{pmatrix} r_t \otimes h_{t-1} \\ X_t \end{pmatrix} + b_g\right)$$

while the output is

$$h_t = (1 - z_t) \otimes h_{t-1} \oplus z_t \otimes g_t$$

where "$\otimes$" is pointwise multiplication and "$\oplus$" is the usual pointwise addition. Note that each of the weight matrices is $n \times 2n$.

In matrix form, ignoring the bias terms b, we have

$$\begin{pmatrix} z_t \\ r_t \\ g_t \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ 0 \end{pmatrix} W\begin{pmatrix} h_{t-1} \\ X_t \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ \tau \end{pmatrix} W\begin{pmatrix} r_t \otimes h_{t-1} \\ X_t \end{pmatrix}$$

where X_t and h_{t-1} are column vectors of length n, and W is the $3n \times 2n$ weight matrix

$$W = \begin{pmatrix} W_z \\ W_r \\ W_g \end{pmatrix}$$

Each of the gates z_t, r_t, and g_t is a column vectors of length n.

The intuition behind a GRU is that it replaces the input, forget, and output gates of an LSTM with just two gates—an "update" gate z_t and a "reset" gate r_t. The GRU update gate serves a similar purpose as the combined output and forget gates of an LSTM. Specifically, the update serves to determine what to output (or write) and what to forget. The function $1 - z_t$ in the GRU implies that anything that is not output must be forgotten. Thus, the GRU is less flexible as compared to an LSTM, since an LSTM allows us to independently select elements for output and elements that are forgotten. The GRU reset gate and the LSTM input gate each serve to combine new input with previous memory.

The gating in a GRU is more complex and somewhat less intuitive as compared to that found in an LSTM. In any case, the most radical departure of the GRU from the LSTM architecture is that there is no cell state in a GRU. This implies that any memory must be stored in the hidden state h_t. This simplification (as compared to an LSTM) relies on the fact that in a GRU, the write and forget operations have been combined.

3.5 Recursive Neural Network

We mention in passing that recursive neural networks can be viewed as generalizing recurrent neural networks.[6] In a recursive neural network, we can recurse over any hierarchical structure, with trees being the archetypal example. Then training can be accomplished via backpropagation through structure (BPTS), often using stochastic gradient descent for simplicity. In contrast, a recurrent neural network is restricted to one particular structure—that of a linear chain.

3.6 Last Word on RNNs

RNNs are useful in cases where the input data is sequential. Generic RNN architectures are subject to vanishing and exploding gradients, which limit the length of the history (or gaps) that can effectively be incorporated into

[6]Unfortunately, "recursive neural network" is typically also abbreviated as RNN. Here, we'll reserve RNN for recurrent neural networks and not use any abbreviation when referring to recursive neural networks.

such models. Relatively complex RNN-based architectures—such as LSTM and its variants—have been developed that can better handle such gradient issues. These architectures have proven to be commercially successful across a wide range of products.

A good general discussion of RNNs can be found in [30], and an overview of various RNN-specific topics—with links to many relevant articles—is available at [28]. A more detailed (mathematical) description can be found in Chapter 10 of [7]. The slides at [22] provide a good general introduction to RNNs, with nice examples and a brief, but excellent, discussion of LSTM.

4 ResNet

At the time of this writing, residual network (ResNet) is considered the state of the art in deep learning for many image analysis problems. A residual network is one in which instead of approximating a function $F(x)$, we approximate the "residual," which is defined as $H(x) = F(x) - x$. Then the desired solution is given by $F(x) = H(x) + x$.

The original motivation for considering residuals was based on the observation that deeper networks sometimes produce worse results, even when vanishing gradients are not the cause [11]. This is somewhat counter-intuitive, as the network should simply learn identity mappings when a model is deeper than necessary. To overcome this "degradation" problem, the authors of [11] experiment with residual mappings and provide extensive empirical evidence that the resulting ResNet architecture yields improved results as compared to standard feedforward networks for a variety of problems. The authors of [11] conjecture that the success of ResNet follows from the fact that the identity map corresponds to a residual of zero, and "if an identity mapping were optimal, it would be easier to push the residual to zero than to fit an identity mapping by a stack of nonlinear layers."

Whereas LSTM uses a complex gating structure to ease gradient flow, ResNet defines additional connections that correspond to identity layers. This enables ResNet to deal with vanishing gradients, as well as the aforementioned degradation problem. These identity layers allow a ResNet model to skip over layers during training, which serves to effectively reduce the minimum depth when training. Intuitively, ResNet is able to train deeper networks by, in effect, training over a considerably shallower network in the initial stages, with later stages of training serving to flesh out the intermediate connections. This approach was inspired by pyramidal cells in the brain, which have a similar characteristic in the sense that they bridge "layers" of neurons [38].

A very high-level illustrative example of a ResNet architecture is given in Figure 21, where each curved edge represents an identity transformation. Note that in this case, the identity transformations enable the model to skip over two layers. In principle, ResNet would seem to be applicable to any flavor of deep neural network, but in practice it seems to applied to CNNs.

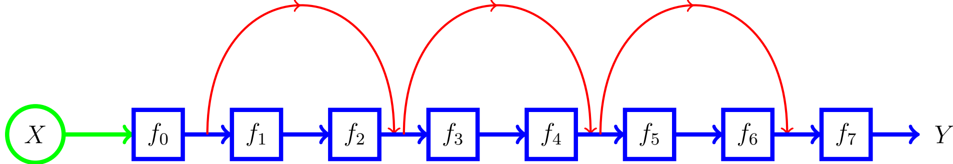


Figure 21: Example of a ResNet architecture

If a ResNet has N identity paths, then the network contains 2^N distinct feedforward networks. For example, the ResNet in Figure 21 can be expanded into the graph in Figure 22. Note that most of the paths in a ResNet are relatively short.

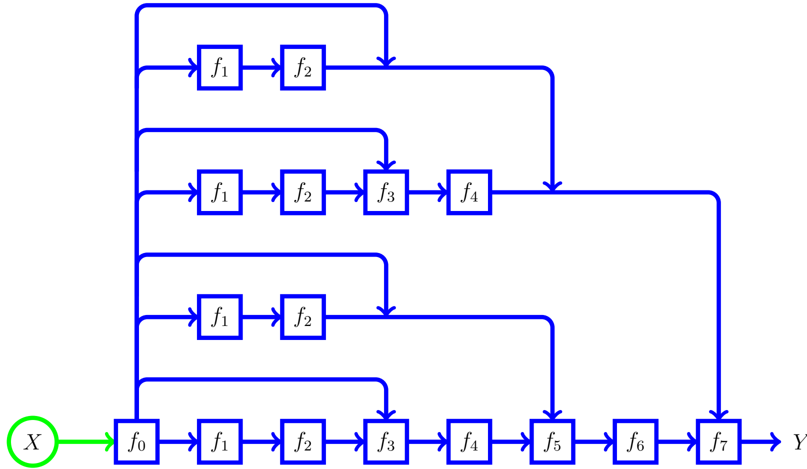


Figure 22: Another view of the ResNet architecture in Figure 21

Surprisingly, the paper [44] provides evidence that in spite of being trained simultaneously, the multiple paths in a ResNet "show ensemble-like behavior in the sense that they do not strongly depend on each other." And perhaps an even more surprising result in [44] shows that "only the short paths are needed during training, as longer paths do not contribute any gradient." In other words, a deep ResNet architecture is more properly viewed as a collection of multiple, relatively shallow networks.

5 GAN

Let $\{X_i\}$ be a collection of samples and $\{Y_i\}$ a corresponding set of class labels. In statistics, a *discriminative* model is one that models the conditional probability distribution $P(Y \mid X)$. Such a discriminative model can be used to classify samples—given an input X of the same type as the training samples $\{X_i\}$, the model enables us to easily determine the most likely class of X by simply computing $P(Y \mid X)$ for each class label Y.

In contrast, a model is said to be *generative* if it models the joint probability distribution of X and Y, which we denote as $P(X, Y)$. Such a model is called "generative" because by sampling from this distribution, we can generate new pairs (X_i, Y_i) that fit the probability distribution. Note that we can produce a discriminative model from a generative model, since

$$P(Y \mid X) = \frac{P(X, Y)}{P(X)}$$

Therefore, in some sense, a generative model is inherently more general than a discriminative model.

Consider, for example, hidden Markov models (HMM) [39], which are a popular class of classic machine learning techniques. An HMM is defined by the three matrices in $\lambda = (A, B, \pi)$, where π is the initial state distribution, A contains the transition probability distributions for the hidden states, and B consists of the observation probability distributions corresponding to the hidden states. If we train an HMM on a given dataset, then we can easily generate samples that match the probability distributions of the HMM. To generate such samples, we first randomly select an initial state based on the probabilities in π. Then we repeat the following steps until the desired observation sequence length is reached: Randomly select an observation based on the current state, using the probabilities in B, and randomly select the next state, based on the probabilities in A. The resulting observation sequence will be indistinguishable (in the HMM sense) from the data that was used to train the HMM.

From the discussion in the previous paragraph, it is clear that a trained HMM is a generative model. However, it is more typical to use an HMM as a discriminative model. In discriminative mode, we determine a threshold, then we classify a given observation sequence as matching the model if its HMM score is above the specified threshold. This example shows that in practice, it is easy to use a generative model as a discriminative model.

On the other hand, while a trained SVM serves to classify samples, we could not use such a model to generate samples that match the training set. Thus, an SVM is an example of a discriminative model.

In the realm of deep learning, a discriminative network is designed to classify samples, while a generative network is designed to generate samples that "fit" the training data. From the discussion above, it is clear that we can always obtain a discriminative model from a generative model. Intuitively, it would seem that training a (more general) generative model in order to obtain a (more specific) discriminative model would be undesirable, since we do not need the full generality of the model. However, reality appears to be somewhat more subtle. In [31] it is shown that for one generative-discriminative pair (naïve Bayes and logistic regression) the discriminative models do indeed have a lower asymptotic error; however the generative models consistently converge faster. This suggests that with limited training data, a generative model might produce a superior discriminative model, as compared to directly training the corresponding discriminative model. In any case, in the realm of deep learning, discriminative models dominate, with an example of a typical application being image classification. In contrast, generative models have only recently come into vogue, with an example application being the creation of fake images.

Now, suppose that when training a discriminative neural network, in addition to the real training data, we generate "fake" training samples that follow a similar probability distribution as the real samples. Further, suppose that these fake training samples are designed to trick the discriminative network into making classification mistakes. Such samples would tend to improve the training of the network, thus making it stronger and more effective than if we had restricted the training to only the real data.

Although intuitively appealing, several problem arises when trying to implement a training technique based on fake samples. For one thing, we generally don't know the distribution of the training set, which often lives in an extremely high dimensional space of great complexity. Another issue is that during training, the discriminative network is constantly evolving, so determining samples that are likely to trick the network is a moving target. Another concern is that if the fake training samples are too difficult—or too easy—to distinguish at any point in the training process, we are unlikely to see any improvement over simply using the real training data

Several techniques have been proposed to try to take advantage of fake data so as to improve the training process. In the case of a generative adversarial network (GAN), we use a neural network to generate the fake data—a generative network is trained to defeat a discriminative network. Furthermore, the discriminative and generative networks are trained simultaneously in a minimax game. This approach sidesteps the complications involved in trying to model the probability distribution of the training samples. In fact, the generative network in a GAN simply uses random noise as its underlying probability distribution.

To summarize, a GAN consists of two competing neural networks—a generative network and a discriminative network—with the generative network creating fake data that is designed to defeat the discriminative network. The two networks are trained simultaneously following a game-theoretic approach. In this way, both networks improve, with the ultimate objective being a discriminative model (and/or a generative model) that is stronger than it would have been if it was trained only on the real training data.

We define two neural networks, namely, a discriminator $D(x;\theta_d)$, and a generator $G(z;\theta_g)$, where θ_d consists of the parameters of the discriminator network, and θ_g consists of the parameters of the generator network. Here, we'll describe the training process in terms of images, but other types of data could be used. Also, to simplify the notation, we'll suppress the dependence on θ_d and θ_g in the remainder of this discussion, except where it is essential for understanding and may not be clear from context.

The generator $G(z)$ produce a fake image (based on the random seed value z) with the goal of tricking the discriminator into believing it is a real training image. In contrast, the discriminator $D(x)$ returns a value in the range of 0 to 1 that can be viewed as its estimate of the probability that the image x is real. For example, $D(x) = 1$ means that the discriminator is completely certain that the image is real, while $D(x) = 0$ tells us that the discriminator is sure that the image is fake, and $D(x) = 1/2$ implies that the discriminator is clueless. Note that the discriminator must deal with both real and fake images, while the generator is only concerned with generating fake images that trick the discriminator.

The generator G wins if D thinks its fake images are real. Thus, we can train G by making $1 - D(G(z))$ as close to zero as possible or, equivalently, by minimizing $\log(1 - D(G(z)))$. On the other hand, D wins if it can distinguish the fake images from real images so, ideally, when training D we want $D(x) = 1$, when x is a real image, and $D(G(z)) = 0$ for fake images $G(z)$. Therefore, we can train D by maximizing $D(x)(1 - D(G(z))$ or, equivalently, by maximizing $\log(D(x)) + \log(1 - D(G(z)))$. We want the D and G models to be in competition, so they can strengthen each other. This can be accomplished by formulating the training in terms of the minimax game

$$\min_G \max_D \Big(E\big(\log(D(x))\big) + E\big(\log(1 - D(G(z)))\big) \Big) \tag{3}$$

where E is the expected value, relative to the implied probability distribution. Specifically, for the max over D, the expectation is with respect to the real sample distribution which has parameters θ_d, while for the min over G, the expectation is with respect to the fake sample distribution, which is specified by the parameters θ_g.

In the case of stochastic gradient descent (or ascent), at each iteration we consider one real sample x and one fake sample $G(z)$. Then, due to the max in equation (3), we first perform gradient ascent to update the discriminator network D. This is followed by gradient descent to update generator network G. Of course, both of these steps rely on backpropagation.

Note that for the discriminator network D, the backpropagation error term involves

$$\log\bigl(D(x)\bigr) + \log\bigl(1 - D(G(z))\bigr)$$

while for the generator network G, the error term involves only

$$\log\bigl(1 - D(G(z))\bigr) \tag{4}$$

Of course, in practice, we would typically use a minibatch of, say, m real samples and m fake samples at each update of D and G, rather than a strict stochastic gradient descent/ascent.

There is one technical issue that arises when attempting to train the generator network G as outlined above. As illustrated in Figure 23, the gradient of the expression in (4) is nearly flat for values of $D(G(z))$ near zero. This implies that early in training, when the generator network is sure to be extremely weak—and hence the discriminator can easily identify most $G(z)$ images as fake—it will be difficult for the G network to learn. From, Figure 23 we also see that

$$\log\bigl(D(G(z))\bigr) \tag{5}$$

is relatively steep near zero. Hence, instead training G based on a gradient ascent involving equation (4), we'll perform gradient *descent* based on (5). Note that we've simply replaced the problem of maximizing $1 - D(G(z))$ with the equivalent problem of minimizing the probability $D(G(z))$.

The algorithm for training a GAN is summarized in Figure 24. In some applications, letting $\texttt{iters} = 1$ works best, while in others, $\texttt{iters} > 1$ yields better results. In the latter case, we update the discriminator network D multiple times for each update of the generator network G. This implies that in such cases, the generator might otherwise overwhelm the discriminator, that is, the generator is in some sense easier to train. Finally, while a GAN certainly is an advanced architecture, it is important to realize that training reduces to a fairly straightforward application of gradient ascent.

As with LSTM, there are a vast number of variations on the basic GAN approach outlined here; see [23] for a list of nearly 50 such variants. Additional sources of information on GANs include the original paper on the subject [8] and the excellent slides at [23].

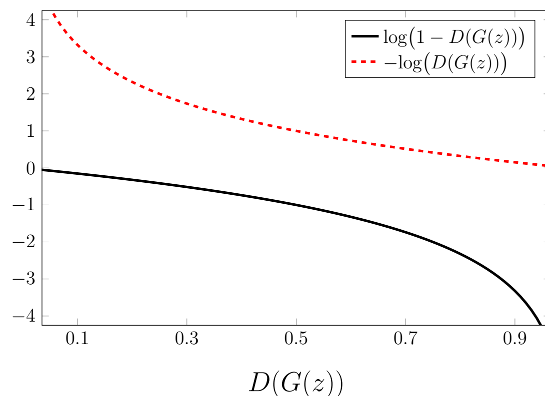


Figure 23: Gradient of generator network G

0:	initialize parameters θ_d and θ_g and `iters` ≥ 1
1:	repeat
2:	for $k = 1$ to `iters`
3:	randomly select n noise samples $Z = (z_0, z_1, \ldots, z_{n-1})$
4:	randomly select n real samples $X = (x_0, x_1, \ldots, x_{n-1})$
5:	update θ_d by gradient ascent on $\displaystyle\sum_{i=0}^{n-1}\Big(\log\big(D(x_i)\big) + \log\big(1 - D(G(z_i))\big)\Big)$
6:	next k
7:	randomly select n noise samples $Z = (z_0, z_1, \ldots, z_{n-1})$
8:	update θ_g by gradient ascent on $\displaystyle\sum_{i=0}^{n-1}\log\big(D(G(z_i))\big)$
9:	until stopping criteria is met
10:	return(θ_d, θ_g)

Figure 24: GAN training algorithm

6 Word2Vec

Word2Vec is a technique for embedding terms in a high-dimensional space, where the term embeddings are obtained by training a shallow neural network. After the training process, words that are more similar in context will tend to be closer together in the Word2Vec space.

Surprisingly, some algebraic properties also hold for Word2Vec embeddings. For example, according to [26], if we let

$$w_0 = \text{“king”}, w_1 = \text{“man”}, w_2 = \text{“woman”}, w_3 = \text{“queen”}$$

and $V(w_i)$ is the Word2Vec embedding of word w_i, then $V(w_3)$ is the vector that is closest—in terms of cosine similarity—to

$$V(w_0) - V(w_1) + V(w_2)$$

Results such as this indicate that Word2Vec embeddings capture significant aspects of the semantics of the language.

Before discussing the basic ideas behind Word2Vec, let's consider a somewhat analogous approach to generating vector representations based on hidden Markov models. And, to begin with let's consider individual letters, as opposed to words—we'll call this Letter2Vec.

Recall that an HMM is defined by the three matrices A, B, and π, and is denoted as $\lambda = (A, B, \pi)$. The π matrix contains the initial state probabilities, A contains the hidden state transition probabilities, and B consists of the observation probability distributions corresponding to the hidden states. Each of these matrices is row stochastic, that is, each row satisfies the requirements of a discrete probability distribution. Notation-wise, we'll let N be the number of hidden states, M is the number of distinct observation symbols, and T is the length of the observation (i.e., training) sequence. Note that M and T are determined by the training data, while N is a user-defined parameter. For more details in HMMs, see [39] or Rabiner's fine paper [34].

Suppose that we train an HMM on a sequence of letters extracted from English text, where we convert all upper-case letters to lower-case and discard any character that is not an alphabetic letter or word-space. Then $M = 27$, and we select $N = 2$ hidden states, and we'll use $T = 50{,}000$ observations for training. Note that each observation is one of the $M = 27$ symbols (letters plus word-space). For the example discussed below, the sequence of $T = 50{,}000$ observations was obtained from the Brown corpus of English [3]. Of course, any source of English text could be used.

For one specific case, an HMM trained with the parameters listed in the previous paragraph yields the B matrix in Table 1. Observe that this B matrix gives us two probability distributions over the observation symbols—one for each of the hidden states. We observe that one hidden state essentially corresponds to vowels, while the other corresponds to consonants. This simple example nicely illustrates the concept of machine learning, as no a priori assumption was made concerning consonants and vowels, and the only parameter we selected was the number of hidden states N. Through the training process, the model learned a crucial aspect of English directly from the data. This illustrative example is discussed in more detail in [39] and originally appeared in Cave and Neuwirth's classic paper [4].

Table 1: Final B^T for HMM

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

Suppose that for a given letter ℓ, we define its Letter2Vec representation $V(\ell)$ to be the corresponding row of the matrix B^T in Table 1. Then, for example,

$$\begin{array}{ll} V(\text{a}) = \begin{pmatrix} 0.13537 & 0.00364 \end{pmatrix} & V(\text{e}) = \begin{pmatrix} 0.21176 & 0.00223 \end{pmatrix} \\ V(\text{s}) = \begin{pmatrix} 0.00032 & 0.11069 \end{pmatrix} & V(\text{t}) = \begin{pmatrix} 0.00158 & 0.15238 \end{pmatrix} \end{array} \tag{6}$$

Next, we consider the distance between these Letter2Vec representations. Instead of using Euclidean distance, we'll measure the cosine similarity.[7]

The cosine similarity of vectors X and Y is the cosine of the angle between the two vectors. Let $S(X,Y)$ denote the cosine similarity between vectors X and Y. Then for $X = (X_0, X_1, \ldots, X_{n-1})$ and $Y = (Y_0, Y_1, \ldots, Y_{n-1})$,

$$S(X,Y) = \frac{\displaystyle\sum_{i=0}^{n-1} X_i Y_i}{\sqrt{\displaystyle\sum_{i=0}^{n-1} X_i^2}\sqrt{\displaystyle\sum_{i=0}^{n-1} Y_i^2}}$$

In general, we have $-1 \le S(X,Y) \le 1$, but since our Letter2Vec encoding vectors consist of probabilities—and hence are non-negative values—we'll always have $0 \le S(X,Y) \le 1$.

[7] Cosine similarity is not a true metric, since it does not, in general, satisfy the triangle inequality.

When considering cosine similarity, the length of the vectors is irrelevant, as we are only considering the angle between vectors. Consequently, we might want to consider vectors of length one, $\widetilde{X} = X/\|X\|$ and $\widetilde{Y} = Y/\|Y\|$, in which case the cosine similarity simplifies to the dot product

$$S(\widetilde{X}, \widetilde{Y}) = \sum_{i=0}^{n-1} \widetilde{X}_i \widetilde{Y}_i$$

Henceforth, we'll use the notation $\widetilde{X}$ to indicate a vector X that has been normalized to be of length one.

For the vector encodings in (6), we find that for the vowels "a" and "e", the cosine similarity is $S(V(\text{a}), V(\text{e})) = 0.9999$. In contrast, the cosine similarity of the vowel "a" and the consonant "t" is $S(V(\text{a}), V(\text{t})) = 0.0372$. The normalized vectors $V(\text{a})$ and $V(\text{t})$ are illustrated in Figure 25. Using the notation in this figure, cosine similarity is $S(V(\text{a}), V(\text{t})) = \cos(\theta)$

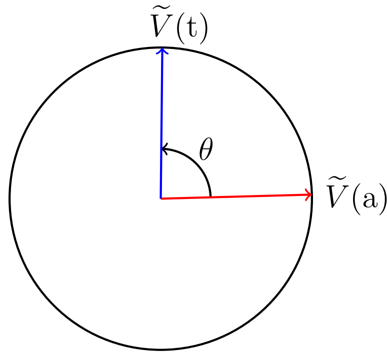


Figure 25: Normalized vectors $\widetilde{V}(\text{a})$ and $\widetilde{V}(\text{t})$

These results indicate that these Letter2Vec encodings—which are derived from a trained HMM—provide useful information on the similarity (or not) of pairs of letters. Note that we could obtain a vector encoding of any dimension by simply training an HMM with the number of hidden states N equal to the desired dimension.

Our HMM-based approach to Letter2Vec encoding is interesting, but we want to encode words, not letters. Analogous to the Letter2Vec embeddings discussed above, we could train an HMM on words and then use the columns of the resulting B matrix (equivalently, the rows of B^T) to define word vectors. The state of the art for Word2Vec uses a dataset corresponding to $M = 10{,}000$, $N = 300$ and $T = 10^9$. Training an HMM with similar parameters would be decidedly non-trivial, as the work factor is on the order of N^2T.

While the word embedding technique discussed in the previous paragraph—let's call it HMM2Vec—is plausible, it has some potential limitations. Perhaps the biggest issue with HMM2Vec is that we typically train an HMM based on a Markov model of order one. This means that the current state only depends on the immediately-preceding state. By basing our word embeddings on such a model, the resulting vectors would likely provide only a very limited sense of context. While we can train HMMs using models of higher order, the work factor would be prohibitive.

Word2Vec uses a similar approach as the HMM2Vec concept outlined above. But, instead of using an HMM, Word2Vec is based on a shallow (one hidden layer) neural network. Analogous to HMM2Vec, in Word2Vec, we are not interested in the resulting model itself, but instead we make use the learning that is represented by the trained model to define word embeddings. Next, we consider the basic ideas behind Word2Vec. Our presentation is fairly similar to that found in the excellent tutorial [25].

Suppose that we have a vocabulary of size M. We'll encode each word as a "one-hot" vector of length M. For example, suppose that our vocabulary consists of the set of $M = 8$ words

$$\begin{aligned} W &= (w_0, w_1, w_2, w_3, w_4, w_5, w_6, w_7) \\ &= (\text{“for”}, \text{“giant”}, \text{“leap”}, \text{“man”}, \text{“mankind”}, \text{“one”}, \text{“small”}, \text{“step”}) \end{aligned}$$

Then we encode "for" and "man" as

$$E(w_0) = E(\text{“for”}) = 10000000 \;\text{ and }\; E(w_3) = E(\text{“man”}) = 00010000$$

respectively.

Now, suppose that our training data consists of the phrase

$$\text{“one small step for man one giant leap for mankind”} \tag{7}$$

To obtain training samples, we specify a window size, and for each offset we use all pairs of words within the specified window. For example, if we select a window size of two, then from (7), we obtain the training pairs in Table 2.

Consider the pair "(for,man)" from the fourth row in Table 2. As one-hot vectors, this training pair corresponds to input 10000000 and output 00010000.

A neural network similar to that in Figure 26 is used to generate Word2Vec embeddings. The input is a one-hot vector of length M representing the first element of a training pair, such as those in Table 2, and the network is trained to output the second element of the ordered pair. The hidden layer consists of N linear neurons and the output layer uses a softmax function to generate M

Table 2: Training data

Offset	Training pairs
"[one] small step ..."	(one,small), (one,step)
"one [small] step for ..."	(small,one), (small,step), (small,for)
"one small [step] for man ..."	(step,one), (step,small), (step,for), (step,man)
"... small step [for] man one ..."	(for,small), (for,step), (for,man), (for,one)
"... step for [man] one giant ..."	(man,step), (man,for), (man,one), (man,giant)
"... for man [one] giant leap ..."	(one,for), (one,man), (one,giant), (one,leap)
"... man one [giant] leap for ..."	(giant,man), (giant,one), (giant,leap), (giant,for)
"... one giant [leap] for mankind"	(leap,one), (leap,giant), (leap,for), (leap,mankind)
"... giant leap [for] mankind"	(for,giant), (for,leap), (for,mankind)
"... leap for [mankind]"	(mankind,leap), (mankind,for)

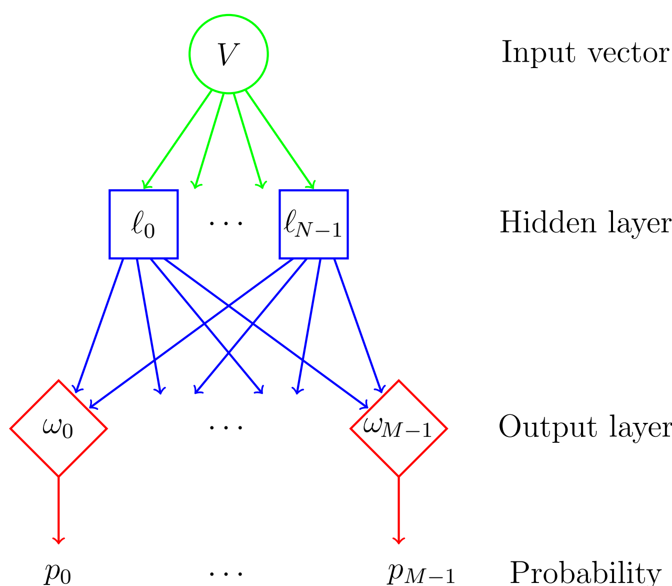


Figure 26: Neural network for Word2Vec embeddings

probabilities, where p_i is the probability of the output vector corresponding to w_i for the given input.

Observe that the Word2Vec network in Figure 26 has NM weights that are to be determined, as represented by the blue lines from the hidden layer to the output layer. For each output node ω_i, there are N edges (i.e., weights) from the hidden layer. The N weights that connect to output node ω_i form the Word2Vec embedding $V(w_i)$ of the word w_i.

As mentioned above, the state of the art in Word2Vec for English text is based on a vocabulary of $M = 10{,}000$ words, and embedding vectors of length $N = 300$. These embeddings are obtained by training on a set of about 10^9 samples. Clearly, training a model of this magnitude is an extremely challenging computational task, as there are 3×10^6 weights to be determined, not to mention a huge number of training samples to deal with. Most of the complexity of Word2Vec comes from tricks that are used to make it feasible to train such a large network with a massive amount of data.

One trick that is used to speed training in Word2Vec is subsampling of frequent words. Common words such as "a" and "the" contribute little to the model, so these words can appear in training pairs at a much lower rate than they are present in the training text.

The most significant work-saving trick that is used in Word2Vec is so-called "negative sampling." When training a neural network, each training sample potentially affects all of the weights of the model. Instead of adjusting all of the weights, in Word2Vec, only a small number of "negative" samples have their weights modified per training sample. For example, suppose that the output vector of a training pair corresponds to word w_0. Then the "positive" weights are those of output node ω_0, and all of the corresponding weights are modified. In addition, a small subset of the $M - 1$ "negative" words (i.e., every word in the dataset except w_0) are selected and only the weights of the corresponding output nodes are modfied. The distribution used to select the negative subset is biased towards more frequent words.

A high level discussion of Word2Vec can be found in [2], while a very nice and intuitive—yet reasonably detailed—introduction is given in [25]. The original paper describing Word2Vec is [26] and an immediate followup paper discusses a variety of improvements that mostly serve to make training practical for large datasets [27].

7 Transfer Learning

Suppose that we have thoroughly trained a model and it performs well for a given task. Then, if we have a closely related task, we might be able to reuse most of the previously-trained model to solve this new problem. Specifically, we can attempt to take advantage of the learning embedded in the pre-trained model, and simple retrain the output layer, keeping all other layers fixed. This approach, which is known as transfer learning, has the advantages of being much faster than training a new model completely from scratch, and it also requires far less data.

For example, suppose that we have a deep neural network that was trained on a massive dataset of generic images. Now suppose that we want to distinguish between road signs for an autonomous vehicle application. Then we could simply retrain the output layer on road signs, thus taking advantage of all of the general image learning embedded in the deeper layers of the existing model. As a somewhat less intuitive example, we can treat executable files as images, in which case transfer learning has been applied to distinguish malware from benign, and to classify malware samples into their respective families [46].

8 Ensemble Techniques

In many situations, improved results can be obtained by ensembles, that is, by employing various combinations of deep learning and/or machine learning techniques. Many—if not all—recent machine learning/deep learning competitions have been won by ensembles. For example, the winning strategy for the famous Netflix Prize included a restricted Boltzmann machine (RBM) and gradient boosted decision trees, along with several other techniques [20].

Broadly speaking, we can classify ensembles as bagging, boosting, or stacking, or some combination thereof [43]. Next, we briefly describe each of these three general approaches to constructing ensembles.

Bagging, which is short for "boostrap aggregation," consists of using different subsets of the data or features (or both) to generate distinct scoring functions or classifiers. The results are then combined in some way, which could be as simple as a sum of the scores or a majority vote of the corresponding classifications. In the case of bagging, we assume that the same scoring (or classification) method is used for all scores in the ensemble. For example, bagging is used when generating a random forest, where each individual scoring function is based on a simple decision tree. One benefit of bagging is that it can reduce overfitting, which is a particular problem for decision trees.

Boosting is a process whereby a collection of distinct classifiers are combined to produce a stronger classifier. Often, boosting deals with weak classifiers that are combined in an adaptive or iterative manner that yields a stronger overall classifier. We restrict our definition of boosting to cases where the classifiers are closely related, in the sense that they differ only in terms of parameters. From this perspective, boosting can be viewed as "bagging" based on classifiers, rather than data or features. That is, the scoring (or classification) functions are simply re-parametrized versions of one technique. Popular boosting techniques, such as AdaBoost, can produce arbitrarily strong classifiers, given a sufficient number of weak classifiers [41].

Stacking is a generic ensemble method that combines disparate techniques using a meta-classifier [36]. The scoring functions or classifiers used in stacking can be significantly different from each other. By this definition, both bagging and boosting can be considered to be special cases of stacking.

Because stacking generalizes both bagging and boosting, it is not surprising that stacking-based ensembles can often outperform ensembles that are restricted to bagging or boosting. This is clear from recent competitions, including the KDD Cup [18] and Kaggle competitions [15], as well as the aforementioned Netflix prize [29]. However, Frankenstein-like ensembles are not the end of the story, as efficiency and practicality are often ignored in machine learning competitions. In contrast, when building classifiers in practice, it is virtually always necessary to consider efficiency. A good example of the difference between a competition and a real-world application is provided by the Netflix Prize. The approach that was used to win the Netflix Prize was never fully implemented by Netflix because the improvements over existing techniques "did not seem to justify the engineering effort needed to bring them into a production environment" [14]. Of course, the appropriate tradeoff between efficiency and practicality will depend on the specifics of the problem at hand.

9 Combination Architectures

It is possible—and often highly desirable—to combine various neural network-based architectures. While this could certainly be done in an ensemble approach, sometimes architectures are combined to form a single, non-ensemble technique. For example, suppose that we want to train a network to caption images. Then we might train a CNN on the images and use some flavor of an RNN (e.g., LSTM) to learn captions [13]. The result of the CNN and RNN could be combined via a final fully connected layer to generate captions for given images. This specific type of combination is referred to as an encoder-decoder architecture, since the image is first "encoded" via the CNN, as is the text using an RNN, with the final result "decoded" to generate the captions.

As another example of a combination architecture, we could combine a deep neural network (DNN) with a classic machine learning technique, such as a hidden Markov model (HMM) to create a DNN-HMM architecture [24]. In this case, we might train a DNN, then use the resulting model to generate sequential output by applying the DNN to a sequence of inputs. We would then train an HMM on the output sequence generated by the DNN. In this particular architecture, the HMM would, in effect, be acting as an RNN-like output layer for the DNN. One potential advantage of such an approach is that

HMMs are far more analyzable than neural networks, and hence we might be able to improve the overall model by carefully consider the cases where it makes mistakes.

10 TF-IDF

Term frequency-inverse document frequency (TF-IDF) was developed to serve as a simple method for weighting terms in a document according to their significance, with the goal of automatically indexing documents. Actually, there are several different ways to compute TF-IDF. We describe one of the more popular approaches here.

The general idea behind TF-IDF is fairly simple. First, we compute the relative frequency of a term within a particular document (TF), and then multiply the TF by the inverse of the fraction of documents—within the set of documents under consideration—that include the term (IDF). Below, we give the precise form of both the TF and IDF calculations, and we also discuss the motivation for (and intuition behind) the resulting formulae.

Let $D = \{d_0, d_1, \ldots, d_{n-1}\}$ be a collection of documents, and let N_i be the number of terms in document d_i. Let $T = \{t_0, t_1, \ldots, t_{M-1}\}$ be the distinct terms in D, and let $N_{i,j}$ denote the number of times that term t_j appears in document d_i. In addition, we define the indicator function

$$I(N_{i,j}) = \begin{cases} 1 & \text{if } N_{i,j} > 0 \\ 0 & \text{otherwise} \end{cases}$$

That is, $I(N_{i,j})$ simply indicates whether term t_j is present in document d_i.

At this point, we could simply define the term frequency of term t_j in document d_i as $N_{i,j}$. However, since documents can be of different lengths, we will calculate the term frequency as a relative frequency, that is,

$$\text{tf}(i,j) = \frac{N_{i,j}}{N_i}$$

Obviously, $\text{tf}(i,j)$ tells us which terms occur most frequently in a given document. But, frequent terms are not always the most informative. For example, if the word "the" occurs frequently in a document, then its term frequency will be high, but this does not provide much useful information as to the content of the document. So, we'll also consider how common a term is over the entire set of documents.

There are several possible ways to determine whether a term is rare or common in D. We'll use the inverse document frequency formula

$$\text{idf}(i,j) = \log_2\left(\frac{n}{\sum_i I(N_{i,j})}\right) = \log_2\bigl(n\bigr) - \log_2\Bigl(\sum_i I(N_{i,j})\Bigr)$$

for this purpose. Recall that n is the number of documents in D and observe that the summation term simply counts the number of these n document that include the term t_j. The logarithm serves to dampen the effect of the $\text{idf}(i,j)$, which would otherwise tend to dominate the calculation. Then TF-IDF is computed as the product

$$\text{TF-IDF}(i,j) = \text{tf}(i,j) \cdot \text{idf}(i,j).$$

When $\text{TF-IDF}(i,j)$ is "large," the term t_j occurs relatively often in document d_i, while t_j is not too common (on a per-document basis) in the set D. The intuition is that such a term should be important for indexing the document d_i as it should have special significance within d_i.

TF-IDF has many potential uses in learning, beyond finding index terms in documents. It can be used, for example, in feature engineering as a means of determining the relative importance of various features.

11 Problems

1. Consider the CNN example in Section 2.
   a) Repeat the convolution in Figure 3, using the filter in Figure 2 (d) instead of the filter in Figure 2 (a).
   b) Apply a max pooling layer to the right-hand side of Figure 3, using a pool of size 4×4.
2. In this problem, we determine the number of parameters in the CNN example outlined in Section 2.3.
   a) TBD
   b) TBD
   c) TBD
3. Consider a three-to-one recurrent neural network (RNN).
   a) Draw the equivalent diagrams as in Figures 13 and 14 for the case of a three-to-one RNN.

b) How many weights must be estimated in a three-to-one RNN, and how many weights are there in the fully connected analog to a three-to-one RNN?

c) In general, how many weights are there in an n-to-one RNN, and how many weight are there in the fully connected analog of such an RNN?

4. Rewrite the partial derivative expression in equation (2) without any Z_i terms appearing on the right-hand side. Hint: Rewrite Z_k and Z_j that appear on the right-hand side in terms of f, w, u, and inputs X_i, then simplify the result.

5. LSTM problem.

6. Consider a ResNet architecture where each identity path skips n nodes, and there are m identity paths, so that the total depth of the network is $N = mn + 2$. Note that this includes an initial node f_0 and a final node f_{mn+1}. For example, for the ResNet in Figure 21, we have $n = 2$ and $m = 3$.

   a) Prove that a ResNet with the specified parameters consists of exactly 2^m feedforward networks.

   b) Prove that for this ResNet, $\binom{m}{k}$ of its feedforward networks are of length $2 + kn$, for $k = 0, 1, \ldots, m$.

   c) Determine the average length of the feedforward paths that comprise this ResNet.

7. Another ResNet problem.

8. GAN problem.

9. Another GAN problem.

10. In contrast to the shallow neural network used in Word2Vec, in Section 6 we mentioned that an HMM could be used to generate letter encodings (which we called Letter2Vec) or word encodings (which we called HMM2Vec). We observed that for Letter2Vec, consonants and vowels formed two distinct groups, based on cosine similarity.

    a) Implement Letter2Vec for $N = 3$ and $M = 27$, which results in embedding vectors of length three. Group the letters based on cosine similarity. Discuss the letter relationships within and between your letter groupings.

b) Repeat part a), but using $N = 4$.

c)* Implement HMM2Vec and train on the Brown Corpus with $N = 100$. Let $V(w)$ be the HMM2Vec embedding of word w and let

$$w_0 = \text{“king”}, w_1 = \text{“man”}, w_2 = \text{“woman”}, w_3 = \text{“queen”}$$

Measure the distance, in terms of cosine similarity, from $V(w_3)$ to the vector

$$V(w_0) - V(w_1) + V(w_2) \tag{8}$$

List all words w in your training set, for which $V(w)$ is closer in cosine similarity to (8) than $V(w_3)$.

11. In Section 8 we categorize ensembles as bagging, boosting, or stacking. Give a plausible example—different from those found in the book—for each of the following.

    a) An application where bagging is used.
    b) An application where boosting is used.
    c) An application where stacking is used.
    d) An application where both bagging and stacking are used.
    e) An application where both boosting and stacking are used.
    f) An application where bagging, boosting, and stacking are all used.

12. TF-IDF problems.

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] Suvro Banerjee. Word2vec — A baby step in deep learning but a giant leap towards natural language processing. https://medium.com/explore-artificial-intelligence/word2vec-a-baby-step-in-deep-learning-but-a-giant-leap-towards-natural-language-processing-40fe4e8602ba, 2018.

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

[4] Robert L. Cave and Lee P. Neuwirth. Hidden Markov models for English. In *Hidden Markov Models for Speech*, pages 16–56, IDA-CRD, Princeton, New Jersey, 1980. https://www.cs.sjsu.edu/~stamp/RUA/CaveNeuwirth/index.html.

[5] Daphne Cornelisse. An intuitive guide to convolutional neural networks. https://medium.freecodecamp.org/an-intuitive-guide-to-convolutional-neural-networks-260c2de0a050, 2018.

[6] Adit Deshpande. A beginner's guide to understanding convolutional neural networks. https://adeshpande3.github.io/A-Beginner%27s-Guide-To-Understanding-Convolutional-Neural-Networks/, 2018.

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

[8] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Proceedings of the 27th International Conference on Neural Information Processing Systems*, volume 2 of *NIPS'14*, pages 2672–2680, 2014.

[9] Klaus Greff, Rupesh Kumar Srivastava, Jan Koutník, Bas R. Steunebrink, and Jürgen Schmidhuber. LSTM: A search space odyssey. *IEEE Transactions on Neural Networks and Learning Systems*, 28(10):2222–2232, 2017. https://arxiv.org/pdf/1503.04069.pdf.

[10] Arpit Gupta. Alexa blogs: How Alexa is learning to converse more naturally. https://developer.amazon.com/blogs/alexa/post/15bf7d2a-5e5c-4d43-90ae-c2596c9cc3a6/how-alexa-is-learning-to-converse-more-naturally, 2018.

[11] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. https://arxiv.org/pdf/1512.03385.pdf.

[12] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997. http://www.bioinf.jku.at/publications/older/2604.pdf.

[13] MD. Zakir Hossain, Ferdous Sohel, Mohd Fairuz Shiratuddin, and Hamid Laga. A comprehensive survey of deep learning for image captioning. *ACM Computing Surveys*, 51(6):118:1–118:36, 2019. https://arxiv.org/pdf/1810.04020.pdf.

[14] Casey Johnston. Netflix never used its $1 million algorithm due to engineering costs. *Wired*, 2012. https://www.wired.com/2012/04/netflix-prize-costs/.

[15] Welcome to Kaggle competitions. https://www.kaggle.com/competitions, 2018.

[16] Ioannis Kalfas. Modeling visual neurons with convolutional neural networks. https://towardsdatascience.com/modeling-visual-neurons-with-convolutional-neural-networks-e9c01ddfdfa7, 2018.

[17] Andrej Karpathy. Convolutional neural networks for visual recognition. http://cs231n.github.io/convolutional-networks/, 2018.

[18] KDD Cup of fresh air. https://biendata.com/competition/kdd_2018/, 2018.

[19] Pranav Khaitan. Google AI blog: Chat smarter with Allo. https://ai.googleblog.com/2016/05/chat-smarter-with-allo.html, 2016.

[20] Yehuda Koren. The BellKor solution to the Netflix Prize. https://www.netflixprize.com/assets/GrandPrize2009_BPC_BellKor.pdf, 2009.

[21] Steven Levy. The iBrain is here—and it's already inside your phone. *Wired*. https://www.wired.com/2016/08/an-exclusive-look-at-how-ai-and-machine-learning-work-at-apple/, 2016.

[22] Fei-Fei Li, Justin Johnson, and Serena Yeung. Lecture 10: Recurrent Neural Networks. http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture10.pdf, 2017.

[23] Fei-Fei Li, Justin Johnson, and Serena Yeung. Lecture 13: Generative Models. http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture13.pdf, 2017.

[24] Longfei Li, Yong Zhao, Dongmei Jiang, Yanning Zhang, Fengna Wang, Isabel Gonzalez, Enescu Valentin, and Hichem Sahli. Hybrid deep neural network-hidden Markov model (DNN-HMM) based speech emotion recognition. In *Proceedings of the 2013 Humaine Association Conference on Affective Computing and Intelligent Interaction*, ACII '13, pages 312–317, 2013.

[25] Chris McCormick. Word2vec tutorial — The skip-gram model. http://mccormickml.com/2016/04/19/word2vec-tutorial-the-skip-gram-model/, 2016.

[26] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. https://arxiv.org/abs/1301.3781, 2013.

[27] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. Distributed representations of words and phrases and their compositionality. https://papers.nips.cc/paper/5021-distributed-representations-of-words-and-phrases-and-their-compositionality.pdf, 2013.

[28] Abhishek Narwekar and Anusri Pampari. Recurrent neural network architectures. http://slazebni.cs.illinois.edu/spring17/lec20_rnn.pdf, 2016.

[29] Netflix Prize. https://www.netflixprize.com, 2009.

[30] Graham Neubig. NLP programming tutorial 8 — Recurrent neural nets. http://www.phontron.com/slides/nlp-programming-en-08-rnn.pdf, 2018.

[31] Andrew Y. Ng and Michael I. Jordan. On discriminative vs. generative classifiers: A comparison of logistic regression and naïve Bayes. In *Proceedings of the 14th International Conference on Neural Information Processing Systems: Natural and Synthetic*, NIPS'01, pages 841–848, 2001.

[32] Christopher Olah. Understanding convolutions. http://colah.github.io/posts/2014-07-Understanding-Convolutions/, 2014.

[33] George Philipp, Dawn Song, and Jaime G. Carbonell. The exploding gradient problem demystified — Definition, prevalence, impact, origin, tradeoffs, and solutions. https://arxiv.org/pdf/1712.05577.pdf, 2018.

[34] Lawrence R. Rabiner. A tutorial on hidden Markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–286, 1989. https://www.cs.sjsu.edu/~stamp/RUA/Rabiner.pdf.

[35] Avraham Ruderman, Neil C. Rabinowitz, Ari S. Morcos, and Daniel Zoran. Pooling is neither necessary nor sufficient for appropriate deformation stability in CNNs. https://arxiv.org/abs/1804.04438, 2018.

[36] Vadim Smolyakov. Ensemble learning to improve machine learning results. https://blog.statsbot.co/ensemble-learning-d1dcd548e936, 2017.

[37] Jost Tobias Springenberg, Alexey Dosovitskiy, Thomas Brox, and Martin Riedmiller. Striving for simplicity: The all convolutional net. https://arxiv.org/abs/1412.6806, 2014.

[38] Nelson Spruston. Pyramidal neurons: Dendritic structure and synaptic integration. *Nature Reviews Neuroscience*, 9:206–221, 2019. https://www.nature.com/articles/nrn2286.

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

[40] 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.

[41] Mark Stamp. Boost your knowledge of AdaBoost. https://www.cs.sjsu.edu/~stamp/RUA/ada.pdf, 2018.

[42] Mark Stamp. Deep thoughts on deep learning. https://www.cs.sjsu.edu/~stamp/RUA/ann.pdf, 2018.

[43] Mark Stamp and Fabio Di Troia. On ensemble learning. https://www.cs.sjsu.edu/~stamp/RUA/ensemble.pdf, 2019.

[44] Andreas Veit, Michael Wilber, and Serge Belongie. Residual networks behave like ensembles of relatively shallow networks. https://arxiv.org/pdf/1605.06431.pdf.

[45] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, Melvin Johnson, Xiaobing Liu, Łukasz Kaiser, Stephan Gouws, Yoshikiyo Kato, Taku Kudo, Hideto Kazawa, Keith Stevens, George Kurian, Nishant Patil, Wei Wang, Cliff Young, Jason Smith, Jason Riesa, Alex Rudnick, Oriol Vinyals, Greg Corrado, Macduff Hughes, and Jeffrey Dean. Google's neural machine translation system: Bridging the gap between human and machine translation. https://arxiv.org/abs/1609.08144, 2016.

[46] Sravani Yajamanam, Vikash Raja Samuel Selvin, Fabio Di Troia, and Mark Stamp. Deep learning versus gist descriptors for image-based malware classification. In *Proceedings of the 4th International Conference on Information Systems Security and Privacy*, ICISSP 2018, pages 553–561, 2018.

[47] Matthew D. Zeiler and Rob Fergus. Visualizing and understanding convolutional networks. https://cs.nyu.edu/~fergus/papers/zeilerECCV2014.pdf, 2014.