DRAAM: Deep (Recursive Auto-Associative Memory) And Applied eMbeddings

Presented to

Brandeis University

Department of Computer Science

Jordan Pollack, Advisor

by

Seth Rait

May, 2018

Contents

List of Figures

CHAPTER 1

Abstract

DRAAM is a recursive encoder-decoder network for transforming variable-length hierarchical symbolic data into fixed-width, real-valued vectors and vice-versa. It works by first transforming its input, a prosaic text corpus into lists of high-dimensional dense vectors, called word-embeddings, then feeding those into an encoder network which recursively combines vectors as nodes in a binary tree until a final encoding, the root of the tree, is reached. The decoder network likewise splits the generated encoding until it has reproduced an approximation of the input sentence embeddings. This method shows the efficacy of recursive neural networks for learning encodings for symbolic hierarchical data. While we tested only English language prose, we are hopeful that our model can be made to work on more types of symbolic data.

CHAPTER 2

Introduction

One of the difficulties in applying neural network methodologies to inherently symbolic and cognitive tasks such as Natural Language Processing, is the ability for a given model to represent its variadic input as a fixed-width output. For example, representing context-dependent lists of words (sentences) of varying lengths into single, real-valued vectors. Common techniques for solving this problem include the sliding window method, wherein fixed-width *windows* of words are considered consecutively, or else to determine a maximum input length and pad all shorter inputs to this length. These methods cannot accurately learn representations for variable-length data that are context-aware, because the representation only knows of context within its current buffer.

Jordan Pollack's 1989 paper on distributed representations of recursive data structures pointed out that the failings of machines to be able to represent recursive data structures prevented the application of connectionist models to high-level cognitive tasks like Natural Language Processing[**13**]. In that paper, he proposed a model called Recursive Auto-Associative Memory (RAAM) to solve this very problem, but its reach was limited and it could only express a very small sample space due to the technical limitations of the time and insurmountable reconstruction dilemmas.

Deep learning was then in its nascent stage, but has since progressed far enough that revisiting Pollacks model makes sense. Although techniques for connectionist modeling of Natural Language Processing (NLP) problems have proliferated, we believe that using a recursive tree-based approached as in RAAM, a method that is not commonly employed, could provide novel insights. There are now many well-known and well-used modes for the representation of symbolic data. In this paper, we take a hybrid approach to our representations.

The remainder of this paper is organized as follows. Chapter three will give a technical overview of the core infrastructure used in the implementation of DRAAM as well as short introduction to neural networks. Chapter four will give an overview of RAAM, along with its computational and theoretical limitations. Chapter five will describe the implementation of DRAAM itself. Chapter six will conclude and suggest areas for future work.

CHAPTER 3

Technical Background

3.1. Neural Networks

Artificial neural networks (commonly referred to simply as neural networks) are computing systems used to learn specific tasks through the observation of examples of similar tasks. More specifically, a neural network is a computational system made up of simple interconnected parts called *neurons*, which each apply mathematical functions to their inputs and the outputs of prior neurons in a computational graph. They do this without any prior knowledge of the task itself, which is what differentiates them from traditional computing systems. For example, an image recognition neural network may be trained to summarize chalkboard video lectures without having to understand the content of the lecture, the syntax used on the chalkboard, or the movements of the professor in front of the board[**4**]. This is done by applying a series (possibly billions) of functions to an input (in this case, a video lecture) in order to compute the output (slides summarizing the lecture).

It is not immediately clear what real-valued functions could be used to accomplish such a complicated and nuanced task and generally, that knowledge is hidden. We instead let the networks themselves learn the functions they need. Neural networks can thus be thought of as computational "black boxes". When training them, we observe their behavior, but not their internal state. In figure 3.1 below is a simple neural network which has two layers and takes two inputs, producing one output. It applies functions a_1 and a_2 to the inputs to produce the first layer's output, which becomes the input to the single node in the second layer. The values w_i and b_i are called the weights and biases respectively. The goal of a neural network is to find the specific weights and biases for each neuron in order to produce the desired output from the given input. If one were to inspect these many weights and biases, they would be seemingly random numbers that a programmer has no hope of picking themselves. We later discuss what algorithms can be used to determine these values.

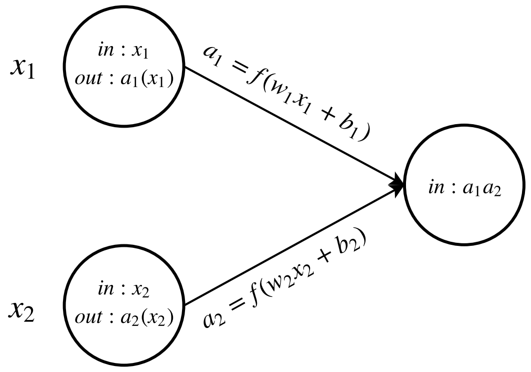


FIGURE 3.1. A Simple Neural Network

This network is called a *feedforward network* because information flows in the direction of the arrows and all arrows are pointing in the same direction. Thus, the network is a directed acyclic graph which composes functions together to produce its output[**5**]. Neural networks have been developed to accomplish many disparate and computationally difficult tasks such as image classification[**9**], speech recognition[**11**], and text translation[**16**]. Though these tasks are very different, they use the same core techniques of neural networks to accomplish their goals.

3.2. Autoencoders

Central to the understanding of both the original RAAM and our new DRAAM is the concept of an autoencoder or autoassociator network, which is a specific class of neural network. The goal of an autoencoder is to learn *representations* (also called *encodings*) of their inputs, usually for the purpose of dimensionality reduction[**6**]. That is, the goal of taking some number inputs and learning to represent those inputs in a smaller space. Other types of autoencoders, such as denoising autoencoders, are used for cleaning up data and reconstructing data from a corrupted version[**17**]. At its simplest form, an autoencoder is a fully-connected 3-layer, feed-forward neural network. Necessarily, the outer two layers (the input and output layers respectively) are the same size as each other, and the central layer is a lower dimension than the other two layers. The output of the central layer is called the encoding. Autoencoders thus have an hourglass shape. The transformation which takes place at each layer can be represented as a system of pseudo-linear equations as shown below.

$$\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \sigma \left(\begin{bmatrix} w_{1,1} & w_{1,2} & \dots & w_{1,m} \\ w_{2,1} & w_{2,2} & \dots & w_{2,m} \\ \vdots & \vdots & \ddots & \vdots \\ w_{n,1} & w_{n,2} & \dots & w_{n,m} \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} \right)$$

FIGURE 3.2. Encoder as Pseudo-linear System of Equations

In figure 3.2 above, each output x_i is calculated by multiplying each input a_i by a vector of weights w_i and adding a bias b_i. After that, some nonlinear activation function is applied. In this case, it is the sigmoid function, $\sigma(x) = \frac{1}{1+e^{-x}}$, though other common options include the rectified linear unit (ReLU) and hyperbolic tangent (tanh). The process of obtaining an output from inputs to the network amounts to a forward propagation of those inputs through each layer, where this transformation takes place on each individual x_i. Note that this is the calculation that occurs at every layer of the network, and the dimensionality of $\vec{X}$ can be several thousands or larger.

The overall process of the feedforward stage of a neural network can be summarized as a composition of semi-linear functions applied to a given input. This is true of all neural networks. For autoencoders in particular, there is the added constraint that if the function composition of the input layer to some non-output layer is called E, and the composition of E through the output layer is called D, then when with input x, $D(E(a)) = x$. In the simple three-layer autoencoder shown below, $E(a)$ is produced by applying these transformations once again to the input, and $D(E(a))$ is produced by applying these transformations once to $E(a)$ using potentially different weights and biases.

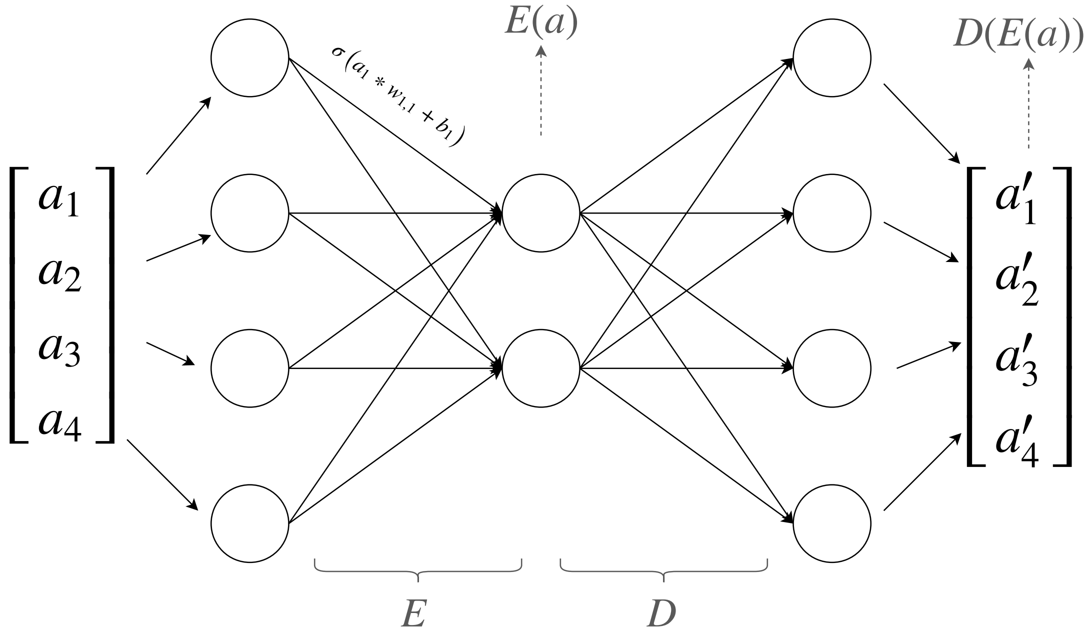


Figure 3.3. 4-2-4 Autoencoder

3.3. Backprop and Gradient Descent

In order for the weights and biases at each layer to produce the desired encoding and decoding to meet the requirement: $D(E(a)) = a$, we need to have the correct values for all the weights and biases (the parameters). There are two pieces of information we need to know in order to arrive at the correct parameters for our model. We need to know:

(1) How bad our original parameters are
(2) How to change them so that they get better over time.

Solving (1) is fairly straightforward. We can compare the outputs of our model $\vec{A}\prime$ to their respective inputs $\vec{A}$ and see how far from each other they are. Since there are multiple inputs and multiple outputs to compare, we do not really care about how each individual input and output performs, just how the model performs on average. Also irrelevant is the sign of the difference between individual a_i and a'_i, since the magnitude of the difference is all that is needed in order to know how bad the results are. We want to penalize larger differences more than smaller ones, since larger differences represent worse results, so we square the values both to normalize them (remove their sign) and give greater precedence to larger errors. We substantially calculate our loss function using the Mean Squared Error of the inputs with respect to the outputs. The loss for input vector $\vec{A} = [a_1...a_n]$ with output vector $\vec{A}\prime = [a'_1...a'_n]$ is given as:

$$\ell = \frac{1}{n}\sum_{i=1}^{n}(a_i - a'_i)^2$$

FIGURE 3.4. Mean Squared Error

Solving (2) is a little more difficult. Now that we know how close we are to our goal, we need a way of getting closer to it. To do this, we use a technique called backwards propagation of errors by gradient descent, wherein we apply the chain rule for computing partial first derivatives to the cost function (in terms of weights and biases) in order to find the gradient. The *gradient* of a function of several variables $f(x_1, x_2, ...x_n)$ denoted as ∇f is a vector giving the direction and magnitude of steepest increase of a function, and is computed as $\nabla f(x_1, x_2...x_n) = (\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, ...\frac{\partial f}{\partial x_n})$. So, it is a necessary condition that the function f have a derivative. Since f must have a derivative, so must the activation function being used. Next, we take the negative of the gradient (hence the name, gradient *descent*) to move towards the minimum of the function. This is exactly what it means for a neural network to *learn.* To learn is just to figure out which weights and biases to use in order to minimize our loss.

This process is similar to rolling a ball down a hill to find valleys. The distance the ball rolls on each iteration of training is called the *learning rate.* We repeat this step many times, given by an *epoch* (the number of times we input the data into the network while training it) until we are very close to a minimum. Each of the learning rate and epoch are called the hyper-parameters of a network and choosing the right values for them is done experimentally for each network. Generally, smaller learning rates pair with more epochs, and smaller learning rates are usually more appropriate as they help to find minima without "rolling too far".

To summarize, we minimize the mean squared error between the inputs and the outputs by propagating backwards through the network. During this propagation, we take partial derivatives to find gradient vectors, which help us to locate minima of the function by taking small steps in the direction opposite the gradient. We repeat this process *epoch* times, each time coming closer and closer to a minimum value for the loss function, thus making our network continually better at reconstructing its input. When updating the weights and biases through back propagation and gradient descent ceases to decrease the error, we know we have arrived at a minimum of the function[1].

3.4. Word Embeddings

DRAAM relies heavily on the use of word embeddings, which are real-valued vector representations of subwords, words, and phrases. Word embeddings were invented as a means for performing computations on natural language data. Computers are woefully inept at handling symbolic data, so a way to represent words as real-valued vectors is needed.

A common model for constructing these vectors is Word2vec[**12**]. Word2vec uses a shallow two-layer neural network to produce embeddings of several hundred dimensions from large vocabularies. Word2vec can be trained in two different modes of operation. In skip-gram mode, the model uses the current word to predict a surrounding context. It weighs closer words heavier than more distant words. In the second mode, called continuous bag of words (CBOW), the current word is predicted from a context window surrounding that word, with the order of the words in the window being irrelevant. CBOW mode is faster at computing embeddings, but skip-gram is better for rare words.

[1]We say *a* minimum instead of *the* minimum, because the function may very well have many minima, and standard gradient descent finds *local* minima, note *global* minima. Finding the global minimum of a function is a much more difficult task than finding local minima.

Word2vec is also trained using two different methods. The hierarchical softmax algorithm which uses a Huffman tree[2] to approximate the conditional log-likelihood of a context is used to train infrequent words. For words which are more common, negative sampling is used. Negative sampling minimizes log-likelihood of negative instances.

The biggest benefit of models like Word2vec is that words which have semantically similar meanings produce similar embeddings. Because of this fact, one can perform semantically meaningful, arithmetic operations on the embeddings. For example, the operation: vec(King) - vec(Man) + vec(Woman) yields a vector very close to vec(Queen)[**12**].[3] This makes word embeddings generated by these models highly adaptable to other tasks such as further processing by neural networks.

[2]A Huffman Tree[**7**] is a binary tree encoding the relative frequency of symbols and is used primarily to construct Huffman codes, a common type of prefix code used for lossless compression. Here, it is used for conditional log log-likelihood of specific letters in the sample.

[3][**14**] suggests some issues with this vector arithmetic and questions the accuracy of this claim, which was initially made by Mikolov in Word2vec.

CHAPTER 4

Previous Work

4.1. RAAM

Pollack's Recursive Auto-Associative Memory, or RAAM[**13**] , is a system used to recursively encode symbolic variable length tree structures into fixed-width real-valued vectors. RAAM networks are nearly identical to the 2k - k - 2k three-layer autoencoders described in the previous chapter, except the outputs from the center layer (the encoder) are applied to the first layer as inputs, therefore making the network recursive.

In RAAM, an input of n symbols is encoded pairwise wherein the n inputs are each of the leaves of a binary tree[1], and the output from the middle layer (before being applied recursively) is the vertex representing the parent node of the two inputs. This output is then fed back into the RAAM with another output from two more of the original inputs. The process continues until the root of the tree is discovered, at which point we have obtained the final encoding of the input. For an input of n symbols, there are $n/2$ first-step encodings, all of which are fed back in as inputs. Thus, the inputs are actually a moving target with an initial size of n, but which generate $2n-1$ pairwise encodings. The decoder recursively decodes and splits until (hopefully) the original input is generated again.

Since intermediate encodings are fed back into the network as inputs, hence creating a "moving target" for the inputs, RAAM is very sensitive to its hyper parameters.

[1]Pollack also explores a ternary-RAAM, which is nearly identical to this explanation, but has a branching factor of three instead of two, ie. a trinary tree.

The learning rate (for gradient descent) and momentum (for integrating previous steps) need to be set low enough in order not to change representations so much as to invalidate the decreasing error, but set high enough that learning occurs. If they are set too high, intermediate representations will change so drastically, that the previous measure of error is a poor measure for how well the model is doing, since the input has changed so much. If the hyper-parameters are too low, the model will take too long to converge.

Below is a sample RAAM encoding for the input a_1, a_2, a_3, a_4. First R_1 is generated by feeding a_1 and a_2 into the input neurons in the network. then R_2 is generated likewise from a_3 and a_4. Finally, R_3 is generated by feeding in R_1 and R_2 in the same way as the original inputs. After this process finishes, we can then decode R_3 back into R_1 and R_2, which in turn can be decoded back into approximations of the original input: a'_1, a'_2, a'_3, a'_4 which we hope are very close to a_1, a_2, a_3, a_4 respectively.

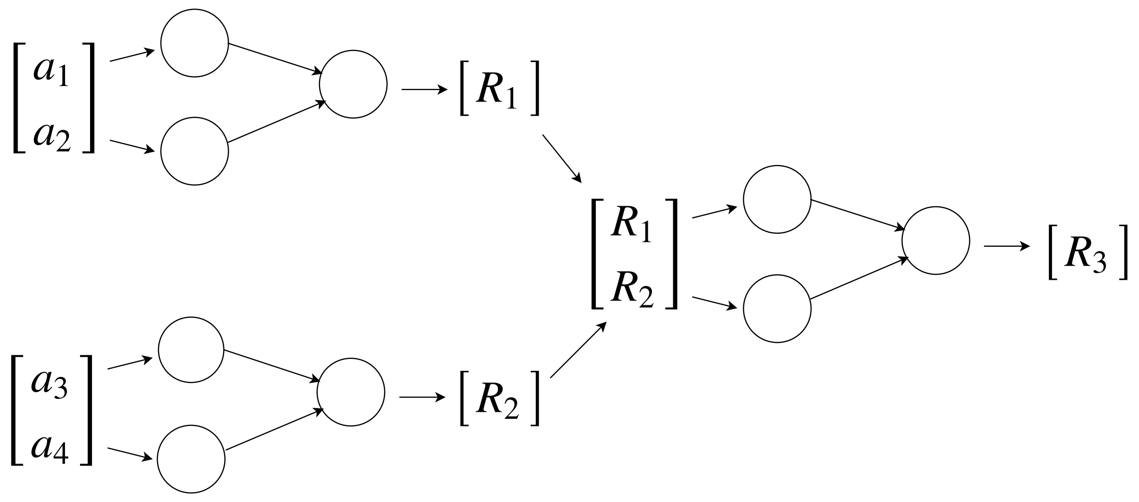


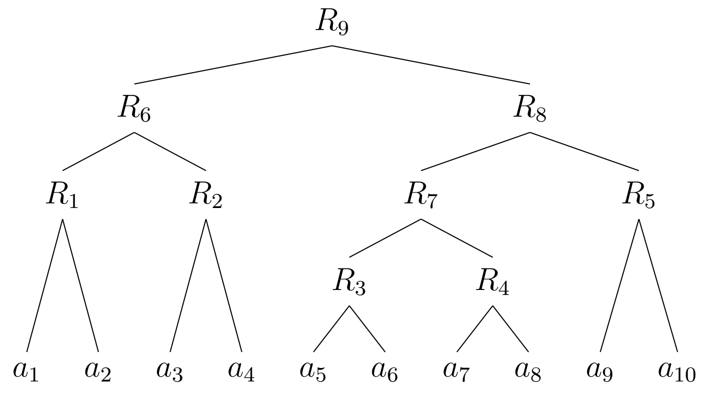
FIGURE 4.1. RAAM Encoder

As shown above in figure 4.1, Encodings are generated on a per-pair basis, then added to the input to be re-encoded, which yields two interesting facts:

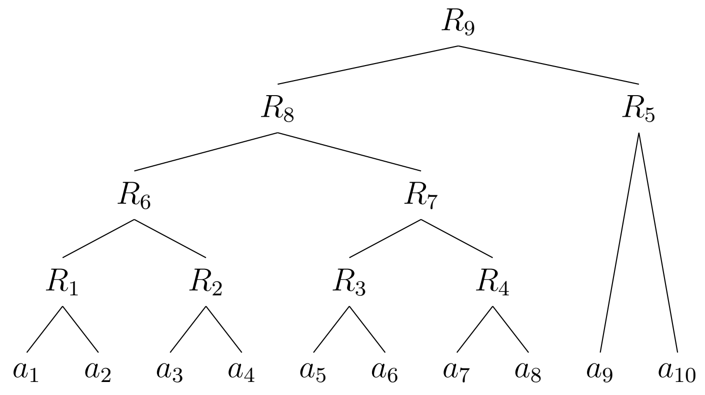
(1) Representations are developed for all subtrees which can be generated from a given input. This is a computationally expensive task which has both advantages and drawbacks when compared to the possibility of only computing the single root encoding. An advantage is that pairs of inputs which occur frequently (such as the bigram "good morning") only need to be encoded one time, since their encoding will be stored. However, storing all intermediate encodings gives rise to very slow training and high memory costs.

(2) When the input size is a power of two, it can be represented as a complete binary tree. However, when the input size is not a power of two, there are several ways one could represent it as a binary tree, as shown in figure 4.2. One method might be to encode the input as a left-branching tree (B), wherein for odd inputs (which are generated from even inputs that are not a power of two), the last symbol in the original input is left as the last symbol in the next round of encoding, and is only ever encoded with the penultimate representation. A better solution, however, is to recycle the last symbol in an odd-length input as the first symbol in the next round, to preserve the trees shape (A).

The RAAM decoder works in the same fashion as the encoder, though needs to overcome one additional complexity. When encoding, it is very easy to decide upon a termination condition, that is, it is easy to tell the encoder when to stop encoding. This happens when the input has been condensed into one item. However, it is not immediately clear how to know how many times to decode and split an encoding to produce its original input. Given some encoding R_x, how do we determine how many times to pass R_x and its decodings through the decoder? Pollack solves this dilemma using a terminal test which transforms the output from binary to analog.


R_9
R_6
R_8
R_1
R_2
R_7
R_5
R_3
R_4
a_1 a_2 a_3 a_4 a_5 a_6 a_7 a_8 a_9 a_{10}

(A) An even tree


R_9
R_8
R_5
R_6
R_7
R_1
R_2
R_3
R_4
a_1 a_2 a_3 a_4 a_5 a_6 a_7 a_8 a_9 a_{10}

(B) A left branching tree

FIGURE 4.2. Alternate encodings of the sentence $[a_1...a_{10}]$

Before explaining the terminal test, it is important to understand the structure of the inputs to RAAM. Pollack uses an n-hot similarity-based classifying scheme for his inputs. In this scheme, where inputs to the RAAM are sentences, words in the sentences are divided into five classes, THING, HUMAN, PREPOSITION, ADJECTIVE, and VERB, and each word in the group is represented as a unique (to that group) bit-string vector. Sentences are then the concatenation of the bit-strings which make up their constituent words, with empty vectors representing sentences without one of the five classes.

After encoding a sentence, we are left with one real-valued vector representing it. To recover the original sentence, we apply the vector to the decoder recursively, each time checking whether all values in the resulting vectors are within some threshold τ (which Pollack sets to $\tau = 0.2$) for terminals and $\upsilon = 0.05$ for non-terminals. Double thresholding is needed because, while $\tau = 0.2$ is a fine threshold for converting real-valued vectors to analog identifiers, it is much too permissive to be able to successfully reconstruct terminals from their their non-terminal representations. That is, to reconstruct terminals a and b from their representation R_1, when obtaining R_1 from R_2, R_1' must be very close to R_1.

4.2. Limitations and Capacity

The original RAAM had some notable drawbacks which have prevented it from being able to effectively scale to industry-sized data sets commonly used in NLP. Most of these issues pertained to the terminal test. This resulted in four main issues, three of which are discussed by [**10**]. First, if representations never pass the terminal test, the deconstructor gets caught in an infinite loop. This could be potentially easy to fix by breaking the test after a certain number of deconstructions (assuming one has an upper bound of how big inputs are) and then doing a nearest neighbor search to find the most likely input that generated the vector.

The second problem is the converse of the first, wherein a non-terminal passes the terminal test. Third, there is a tradeoff between precision and capacity in RAAM. Smaller tolerances (such as $\tau = 0.1$ and $\upsilon = 0.02$) will slow down convergence during the training stage, but larger tolerances will allow fewer representations, thus decreasing capacity. Lastly, using bit-patterns, the Hamming distance [2] between some

[2]The Hamming distance, or edit distance, between two strings a and b is the number of places where a needs to change in order to be equal to b. Hamming distance is symmetric, so $Hamming(a, b) == Hamming(b, a)$.

terminals in the same class is equal to 1, so similar terminals could be decoded interchangeably as each other. For example, the preposition "John loves Mary", could easily be decoded as "Frank loves Mary". This still leaves the grammatical structure of the input intact, which is desirable, but does change the overall meaning of the preposition.

4.3. IRAAM

The limited practical capacity of RAAM is problematic, but inspection of its theoretical representational capacity yields some insights into how a RAAM-type system with a higher capacity could be constructed. As mentioned above, using a terminal test is problematic and is the cause of many of RAAM's representational woes. Levy[**10**] presents a different way of describing RAAMs which leads to a more dynamic way of identifying terminals, thus elucidating the theoretical capacity of RAAMs and increasing their practical performance. Recall equation 3.2 where we represent a layer of an autoencoder as a system of semi-linear equations. Levy presents a similar view of a complete RAAM decoder as follows (vector size of 4):

$$t_k \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \sigma \left(\begin{bmatrix} w_{k,1,1} & w_{k,1,2} & w_{k,1,3} & w_{k,1,4} \\ w_{k,2,1} & w_{k,2,2} & w_{k,2,3} & w_{k,2,4} \\ w_{k,3,1} & w_{k,3,2} & w_{k,3,3} & w_{k,3,4} \\ w_{k,4,1} & w_{k,4,2} & w_{k,3,4} & w_{k,4,4} \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ a_4 \end{bmatrix} + \begin{bmatrix} b_{k,1} \\ b_{k,2} \\ b_{k,3} \\ b_{k,4} \end{bmatrix} \right)$$

FIGURE 4.3. Decoder as IFS

Each $t_k \in T$ is in the set of transforms which decode each of the left and right subtrees[3], and the matrix W and vector b are the standard weights and biases as in figure 3.2. Levy points out that this way of presenting a RAAM decoder is very

[3]When Levy presents this in his work, he uses a ternary-RAAM, and so T is the set of transforms which decode each of the left, right, and center subtrees.

similar to an Iterated Function System (IFS). IFSs are a method for constructing fractals by applying outputs of a set of linear transforms as the inputs to that same set of transforms, much in the same way that RAAM applies the output of its encoder into the input again to achieve its recursive representations.

In an IFS, as the number of iterative applications approaches infinity, the limit is called the attractor of the IFS. The attractor is the fixed point of the transforms. If all transformations $t_k \in T$ are contractive, that is, the distance between two points decreases after application of the transform, then the IFS has an attractor and that attractor is a fractal. Fractals are notable in that, as the name implies, they are fractional dimensional [4] and, more importantly to RAAM, their enclosed surface area is infinite.[5]

As Levy points out, because of the use of the non-linear sigmoid function, RAAM cannot technically be an IFS, but the properties of the sigmoid activation function are still useful in this viewpoint, as that function is pseudo-contractive, since the range of σ is (0, 1). Though it will not always bring points closer on successive iterations, it will shrink successive areas of the images of the unit square.

This realization allows for a new kind of terminal test for RAAM decoders: *is the point on the attractor?* If so, the point is a terminal, if not, we continue decoding until we reach a point on the attractor. Helpfully, this new terminal test has none of the issues of the previous one, with some added benefits. The theoretical representational capacity of a RAAM implementing this terminal test is infinite (so we call it IRAAM, or Infinite RAAM), since every point on the attractor is a terminal, and the surface

[4]The dimension of a given fractal is called its *Hausdorff dimension* and can be computed as the ratio of the logarithm of the number of transforms to the logrithm of their scaling factor.[**15**]

[5]More accurately, for all 2-dimensional fractals, the surface area is 0, but the surface area *enclosed by* the fractal depends on the shape of the specific fractal in question. For fractals of higher dimension, their surface area is infinite.

area enclosed by the fractal is infinite. This also gives us a way of generating terminals from random encodings. One must only pick a random point not on the attractor and decode it until landing on the attractor to find its terminals. This then yields a way to find the space of all possible representations of a given IRAAM: just apply the decoder to every point not already on the attractor.

Practically, the matter is somewhat messier. As stated by Barnsley[**1**], each point on an attractor can be associated with an address which is the sequence of the indices of the transforms used to arrive at that point from some other points, which is how we obtain IRAAM's infinite capacity. However, an infinitely long address is of little use to someone creating an IRAAM on a physical computer, so we must limit IRAAMs to a certain *resolution*. That is, we cannot rely on the fact that in the limit, an IRAAM has infinite capacity, as we need to halt our computations at some point. Therefore, there is a direct relationship between the resolution of the IRAAM and its representational capacity.

CHAPTER 5

DRAAM

We designed several new models based off of RAAM. First, we implement a RAAM using one-hot vectors to encode English sentences. Next, we phased-out the one-hot vectors in favor of word embeddings, as described in section 3.4. Lastly, we create a RAAM using a deep recursive neural network, which we call Deep RAAM, or DRAAM.

These models were all trained using an optimizer based on a variant of gradient descent called Adam[**8**]. Adam is similar to stochastic gradient descent(SGD), but combines methods from other algorithms as well. From the Adaptive Gradient Algorithm (AdaGrad), it adopts a per-parameter/weight learning rate (as opposed to the single learning rate α used by SGD) which improves performance on problems which have sparse gradients[**3**]. From the Root Mean Square Propagation Algorithm (RMSProp), it borrows the idea of adaptive learning rates. That is, the parameter learning rates are updated based on the mean of the magnitudes of the gradients for the weights, or how quickly the weights are changing. This is particularly well suited for our problem because of the moving target inputs.

5.1. One-hot Limited Gradient Flow

Our first experiment was a recursive one-hot autoencoder, similar to that in [**13**] for encoding prepositions with a branching factor of 3, but somewhat simplified. Pollack's approach in this section was to transform each terminal - a grammatical preposition - into a similarity-based 16-bit binary representation by division of the words into the semantic classes THING, HUMAN, PREP, ADJ, and VERB. Each word in each group was then given a bit pattern and patterns were concatenated, inserting groups of 0 for empty spots. These 16-bit patterns were then encoded and decoded as described in section 4.1.

Our model encodes letters as one-hot vectors in the following way. For a language with k arbitrarily-ordered symbols, symbol number n is a vector whose nth element is 1, and all other elements are 0. Sentences are then generated from the vectors by creating lists of these vectors. Thus, a sentence is a list of m vectors of size k, where m is the number of words in the sentence and k is the dimensionality of each vector (ie. the number of symbols in the language). Within each list, symbols are encoded as pairwise concatenations of their vectors as in a 2k-k-2k RAAM, with the outputs of the center layer being fed back into the input layer until all symbols in a sentence are consumed. The encodings take place in rounds starting with the terminal symbols as the roots of a binary tree. Once all inputs of a round are consumed, they are fed back into the input layer and the next round begins. For sentences of odd length, this results in a left-branching binary tree, wherein the penultimate encoding is concatenated with the last input symbol from the original input to produce the final encoding.

This RAAM uses a rectified linear unit activation for each neuron. A rectified linear unit (ReLU) is the function $f(x) = max(0, x)$[1]. As such, the output is 0 when the input is negative, and the output is equal to the input otherwise. Our loss is the mean squared error as before, and uses a limited gradient flow, ie. the loss is calculated on a per-layer basis. For example, for a sentence with terminals a, b, c, d an encoder function E and decoder function D. The loss is calculated as

[1]In section 3.3, we said the activation function needs to be differentiable in order for gradient descent to work. ReLU, however, is not differentiable at $x = 0$, so to solve this we use a subderivative. When $x = 0$ we manually set the derivative to 0. So $\frac{d}{dx}f(x)|_{x=0} = 0$

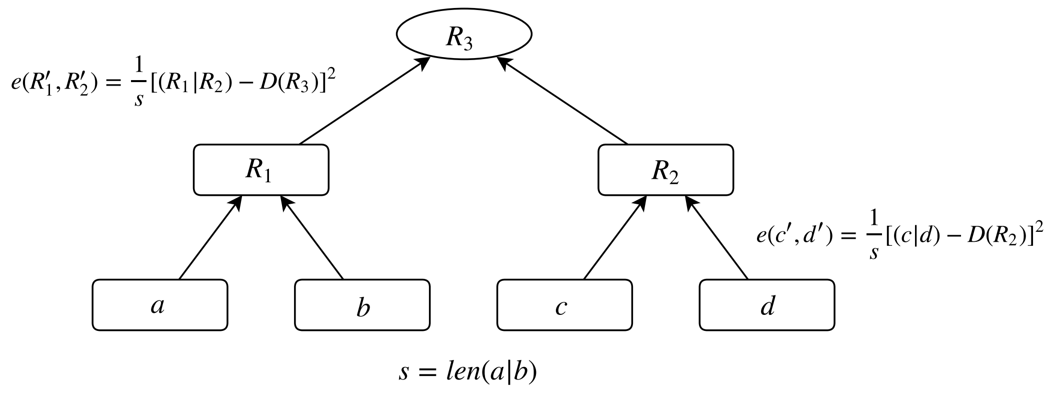

R_3
$e(R'_1,R'_2) = \frac{1}{s}[(R_1|R_2) - D(R_3)]^2$
R_1
R_2
$e(c',d') = \frac{1}{s}[(c|d) - D(R_2)]^2$
a
b
c
d
$s = len(a|b)$

FIGURE 5.1. Limited Gradient Flow RAAM Encoder

$MSE((a|b), D(E(a|b)))$ where E is not applied recursively. So, if $E(a|b) = R_1$, and $D(R_1) = (a'|b')$, we calculate the loss at this level before continuing the recursive encoding. Because of this fact, we call this model a limited gradient flow network. We later explore an alternate method for calculating loss.

This model was able to successfully encode and decode its training set, but did not generalize well to novel data. This was expected, as one-hot vectors are information-sparse and contain no information about the underlying data, so there is little information and no patterns for models to learn.

5.2. Dense Limited Gradient Flow

The next system we implemented was nearly identical to the first, but instead of using one-hot vectors to represent symbols in the language, we used 300-dimensional real-valued dense vectors obtained from the fasttext network[**2**]. Fasttext is a model used to generate word embeddings similar to Word2vec. Embeddings created by fasttext contain subword information by training on character n-grams. This is a desirable property for embeddings to have, as it increases the similarity of tenses of the same words, as well as words with similar prefixes and suffixes. Our training and testing sets were formed from “Pride and Prejudice”, “Emma”, and “Sense and Sensibility”. Sentences were parsed from the text and converted into lists of 300 dimensional vectors using fasttext. They were then fed into a network in a similar way to the previous experiment, but with a few key differences.

This network was deeper, with an input layer of 600 neurons (2x300), a first hidden layer of 450 units, and a central layer, representing the encoded sentences, as 300 units. The decoder was symmetric to the encoder. Instead of using a sigmoidal activation, a hyperbolic tangent ($tanh(x) = \frac{e^{2x}-1}{e^{2x}+1}$) was used, as it matched the range of the input vectors: $(-1, 1)$. For our learning rate, we used the value $1 \cdot 10^{-5}$, which was chosen experimentally. This model showed a greater ability to generalize, as it was able to produce similar losses after training for sentences in the validation set as sentences in the training set.

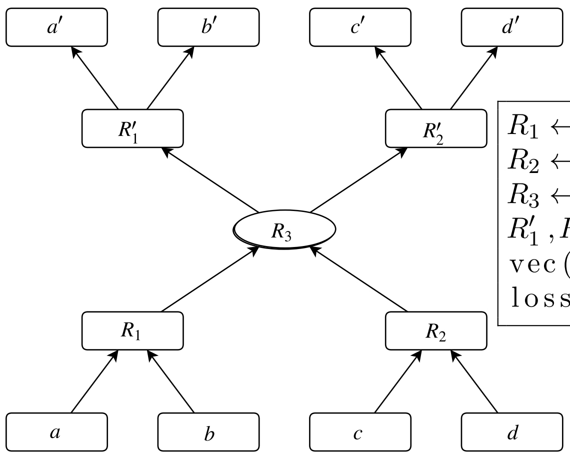
5.3. Deep RAAM

Our last model is similar to the previous one, in that it uses word2vec embeddings, but does away with the limited gradient flow in favor of end-to-end loss calculation. In this model, the loss for terminals is calculated as the MSE of the input terminals and the recursively decoded output terminals. Thus, for sentences of length n, the depth of the network that encodes them is $2log_2(n) - 1$. We therefor call this model Deep Raam And Applied eMbeddings, or DRAAM.

In order to build this network in Tensorflow, which does not natively support dynamic networks, which would be requeired to encode sentences of varying length, we truncated/padded sentences with $\vec{0}$ to all contain the same number of “words”. Further, each word vector was padded by some fraction of its overall length p (which we originally set to $p = \frac{1}{2}$), resulting in each word vector having a dimension of 450.

We do this in order for the terminals' representations to have "room" to differentiate themselves. Since the vectors produced by fasttext are dense, a model which needs to learn compressed representations of these vectors will have little chance to do so without the added padding.

As an example for how the loss calculation of this model differs from the previous one, let the vector encoding of a word a be $vec(a)$, the encoding of a vector be $E(vec)$ and the decoding of a vector be $D(vec)$. For a sentence "Hello, how are you?"[2], the loss for vec(Hello) relies on the computation of the rest of the tree.



```
$R_1 \leftarrow$ E(vec(Hello)|vec(how))
$R_2 \leftarrow$ E(vec(are)|vec(you))
$R_3 \leftarrow$ E($R_1$|$R_2$)
$R'_1$, $R'_2$ =split(D($R_3$))
vec(hello)', vec(how)'=split(D($R'_1$))
loss = MSE(vec(hello)',vec(hello))
```

FIGURE 5.2. Calculating loss in a DRAAM

As shown in figure 5.2 above, calculating loss for inputs now requires an end-to-end computation, unlike in previous RAAMs. That is, the loss for input a can not be computed until a' is generated, which requires first computing representations of b, c, and d. In the previous models, one could calculate the loss for a and a' after only computing R_1. This model is preferable to the limited gradient flow models because the intermediate representations become "free variables", giving the network more

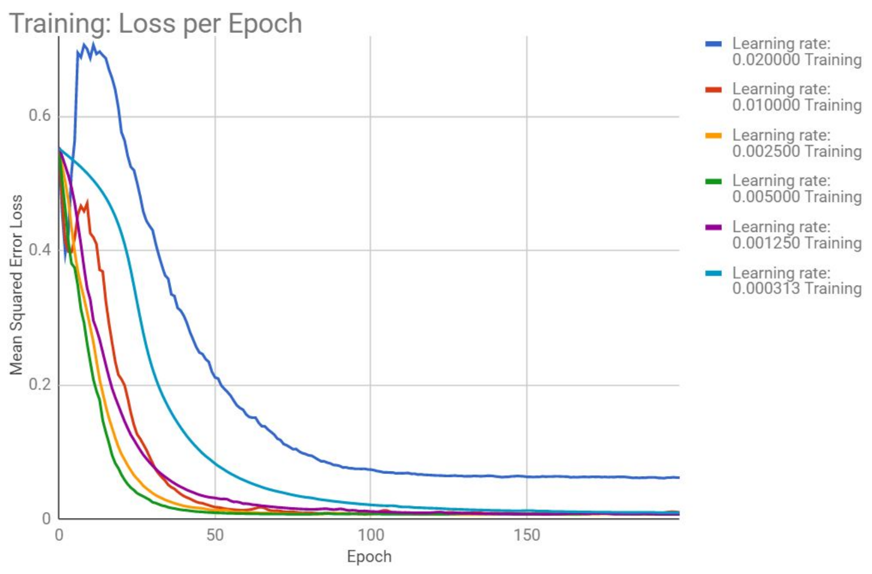
[2]For the purpose of an simple example, we are representing this sentence as four distinct symbols, however in our model it would be parsed as six symbols. Punctuation marks are each represented in the same manner as words.

freedom for learning latent features in the data. However, since DRAAM, no longer stores intermediate representations (R_1 and R_2 in figure 5.2), we cannot explicitly store representations of common two-word phrases such as "*it is*".

This model solves the double terminal test needed in training the original RAAM in two ways. First, since intermediate representations are ephemeral and sentences are truncated/padded to all be the same size, we know how many decodings we need to do. Second, even if we were to implement a dynamic network which allows for varying sentence lengths, one terminal test would likely suffice, as we would only need to check whether the last $p \cdot len(vec)$ digits in the output are very close to 0, denoting a terminal.

Setting the parameters and hyper-parameters of DRAAM was carried out experimentally. We chose a learning rate of $5 \cdot 10^{-3}$ and again used hyperbolic tangent activation functions at each layer. Loss values over time for various learning rates are given in figure 5.3 below both for training data and testing data. Interestingly, DRAAM did not notice any significant degradation in reconstruction quality (as measured by loss and cosine similarity[3]) when shrinking the padding parameter p. We were able to successfully decrease p from $p = \frac{1}{2}$ to $p = \frac{1}{30}$ (resulting in paddings between 150 and 10 for a vector size of 300) without reducing the representation capacity of DRAAM. This suggests that any increase in representational capacity afforded to the network given by the padding is counteracted by the increased difficulty in reconstructing the p zeros. This was a welcome discovery, as it reduces the size of sentence representations by a considerable amount.

[3]The cosine similarity of two vectors is the magnitude of the angle between them (measured by their cosine) and is bounded on the interval $[-1, 1]$, where 1 represents linearly dependent vectors. Thus, if two vectors have a cosine similarity close to 1, they are very similar to each other. Thus, if two vectors are *the same*, they will have a cosine similarity of 1. Though, since all linearly dependent vectors have a cosine similarity of 1, this metric cannot be used alone, thus we also collect information on the MSE loss.



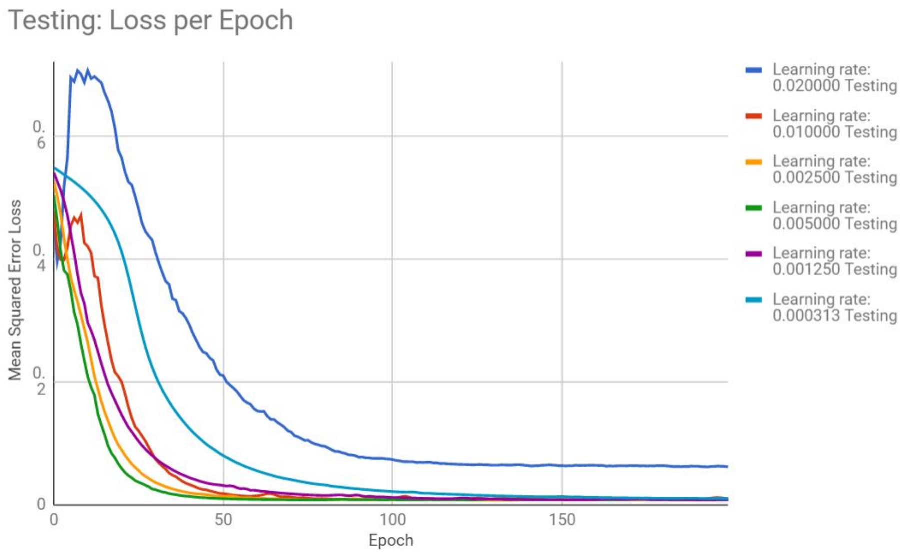


Figure 5.3. Loss During Training

CHAPTER 6

Conclusion and Further Work

In this paper, we presented new approaches to the problem of representing variable-width symbolic data in fixed-width representations. Our three models gave us new insights into the practice of building recursive neural networks and applying them to common NLP tasks. The last model. We built off of previous work on recursive representations[**13, 10**] and word embeddings[**2, 12**], resulting in a scheme which combines models for both to accomplish its goal. Our final model, DRAAM, is a recursive autoencoder which uses word embeddins to transform variable length sentences into a fixed-width numeric vector. It provides a novel update to RAAMs and we believe it will benefit from future work, extending its usefulness to a wider array of situations.

Any further work on DRAAM should start by implementing a dynamic network to allow for sentences of varying length without needing to alter them to be the same length. This could be achieved by implementing DRAAM in a framework other than Tensorflow that natively supports dynamic networks, or otherwise using a library such as Tensorflow Fold in order to accomplish the same task within Tensorflow. As mentioned in the previous section, were this to be implemented, a terminal test would need to be used in order to determine when a representation has been fully decoded into terminals. This could merely be a check for when the last $p \cdot log_2(n) - 1$ digits are very near 0, or could be a thresholded cosine similarity to terminals. Further, one could implement an IDRAAM, combining DRAAM's architecture with that of Levy's IRAAM[**10**], further expanding upon DRAAM's representational capacity.

Lastly, future work would explore different methods of concatenating pairs of words in sentences for encoding. In our experiments, we merely concatenated adjacent words. Instead, one could concatenate words which share the same parent in a parse tree of the sentence the words make up. This would further exploit the context-aware nature of the embeddings and could help mitigate reconstruction errors, so that if a sentence is not reconstructed correctly, it still may be reconstructed with words that have a similar meaning and the same part of speech.

CHAPTER 7

Acknowledgments

I would like to thank Professor Jordan Pollack for introducing me to this interesting topic, and for giving me the opportunity to conduct research and learn about subjects I had previously no experience in. I would also like to thank Nick Moran and Aaditya Prakash for answering my endless questions, pointing me towards interesting problems, providing helpful solutions, and elucidating exotic StackOverflow threads. Finally, I would like to thank my adviser for the past four years, Professor Antonella Di Lillo, without whom I would have surely gone mad, and who has given me helpful advice and pushed me to pursue this thesis.

Bibliography

1. Michael F Barnsley, *Fractals everywhere*, Academic press, 2014.
2. Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov, *Enriching word vectors with subword information*, CoRR **abs/1607.04606** (2016).
3. John Duchi, Elad Hazan, and Yoram Singer, *Adaptive subgradient methods for online learning and stochastic optimization*, Journal of Machine Learning Research **12** (2011), no. Jul, 2121–2159.
4. Solomon E Garber, Aaditya Prakash, Nick Moran, Richard Alterman, Maria Altebarmakian, Antonella Di Lillo, and James A Storer, *A two tier approach to chalkboard video lecture summary*, Frontiers in Education Conference (FIE), IEEE, 2017, pp. 1–9.
5. Ian Goodfellow, Yoshua Bengio, and Aaron Courville, *Deep learning*, MIT Press, 2016, url-http://www.deeplearningbook.org.
6. Geoffrey E Hinton and Ruslan R Salakhutdinov, *Reducing the dimensionality of data with neural networks*, science **313** (2006), no. 5786, 504–507.
7. David A Huffman, *A method for the construction of minimum-redundancy codes*, Proceedings of the IRE **40** (1952), no. 9, 1098–1101.
8. Diederik P. Kingma and Jimmy Ba, *Adam: A method for stochastic optimization*, CoRR **abs/1412.6980** (2014).
9. Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton, *Imagenet classification with deep convolutional neural networks*, Advances in neural information processing systems, 2012, pp. 1097–1105.
10. Simon D Levy et al., *Infinite raam: Initial explorations into a fractal basis for cognition*, Ph.D. thesis, Brandeis University, 2002.
11. Tomáš Mikolov, Martin Karafiát, Lukáš Burget, Jan Černockỳ, and Sanjeev Khudanpur, *Recurrent neural network based language model*, Eleventh Annual Conference of the International Speech Communication Association, 2010.
12. Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean, *Distributed representations of words and phrases and their compositionality*, CoRR **abs/1310.4546** (2013).
13. J. B. Pollack, *Recursive distributed representations*, Artif. Intell. **46** (1990), no. 1-2, 77–105.
14. Anna Rogers, Aleksandr Drozd, and Bofang Li, *The (too many) problems of analogical reasoning with word vectors*, Proceedings of the 6th Joint Conference on Lexical and Computational Semantics (* SEM 2017), 2017, pp. 135–148.
15. Manfred Schroeder, *Fractals, chaos, power laws: Minutes from an infinite paradise*, Courier Corporation, 2009.
16. Ilya Sutskever, Oriol Vinyals, and Quoc V Le, *Sequence to sequence learning with neural networks*, Advances in neural information processing systems, 2014, pp. 3104–3112.
17. Pascal Vincent, Hugo Larochelle, Yoshua Bengio, and Pierre-Antoine Manzagol, *Extracting and composing robust features with denoising autoencoders*, Proceedings of the 25th international conference on Machine learning, ACM, 2008, pp. 1096–1103.