Chapter 9

Deep Neural Networks for Natural Language Processing

Ehsan Fathi[1] **and Babak Maleki Shoja**
East Carolina University, Greenville, NC, United States
[1]*Corresponding author: e-mail: fathie15@students.ecu.edu*

ABSTRACT
This chapter discusses the application of deep neural networks for natural language processing. First, we discuss word vector representation followed by feedforward neural networks. Next, training of deep neural network models and their optimization are discussed. Regularization for deep learning is discussed in detail. Application of deep neural approaches to language modeling is described. Lastly, convolutional neural networks and memory are described.

Keywords: Deep learning, Neural network, Convolutional neural networks, Word vectors, Regularization, Sequence modeling

1 INTRODUCTION

Dream of creating machines that can think and learn like human dates way before the first computer was built. Nowadays, *artificial intelligence* (AI) is a very active field of study and research with ubiquitous applications. Computers are very fast in solving problems that are intellectually difficult for human beings but the solution can be formulated in form of a list of clear mathematical operations. But it turns out that real challenge for AI is to solve problems and perform tasks that are very easy for human beings but hard to formulate in the form of formal list. Problems that we can solve intuitively such as recognizing objects in an image or words in speech.

After trying different solutions finally it turned out instead teaching computers to learn (rule-based methods) it works better if we let them to learn from experiences and make a hierarchy of concepts that each one is defined based on simpler concepts. Trying to define an intuitive concept based on simpler concepts and go down the hierarchy levels (layers) until reaching to the most basic concepts, we have to pass through a lot of layers. So to learn

Handbook of Statistics, Vol. 38. https://doi.org/10.1016/bs.host.2018.07.006

an intuitive concept we have to go deep in the hierarchy layers and this is the reason we call this approach *deep learning* (DL).

Different AI projects attempted to model the real world with rules and hard-coding. So computer can use this rules to inference and conclude about a statement automatically. This is known as *knowledge base* approach to AI. After years, none of the projects which were following this approach achieved a considerable success. Problems and difficulties regarding hard-coding knowledge led to another approach known as *machine learning* (ML). Main idea is to let computers to learn from raw data by extracting patterns. In this approach data are represented using some features. It turns out performance of these systems is heavily dependent to these features. The main reason that most ML methods work really well is that designers of the model (human beings) did a good job in designing the model and finding good features. For example, for the task of name entity recognition we may use features like current word, previous word, next word, current word shape (is the first letter capitalized? What about the last letter? Any other letter in the middle? All letters?), surrounding word shape sequence, current part of speech tag, and so forth to find an special entity like organization name. Then you will look at the results and may add another feature to solve a specific problem and again the problem of using features that are specific to the current dataset comes up. So if you use your model for other language or another domain then some of your features may be useless and also you have to again come up with other features to be able to perform well on the new data.

For many AI tasks or problems, right set of features can be designed. Representing data with these features to the ML algorithm solves the problem. Actually, the major part of the task (maybe 90%) is to define features for your model that can describe your data in a way that computers can understand them and then you can use your learning algorithm to optimize the weights on features. A great deal of knowledge in the specific domain you are working on is usually required. Sometimes it may take a decade for researchers to come up with the right set of features.

However, for a lot of tasks, it is extremely difficult to find right set of features. To overcome this problem, we let ML to discover not only the relation of the features (representation) to output but also to learn the features itself. This approach is known as *representation learning*. Learned representations usually lead to much better performance than hand-designed representations. But it is not easy to extract high-level abstract features from raw data. As mentioned earlier DL solves this problem by learning representations which are defined in terms of simpler representations. The beauty of DL is that it will try to automate the process of finding good features and understand different levels of representation only from raw data, e.g., images (pixels) or words (characters).

The reasons of the importance of DL are that it takes a long time to manually design features and then still they may be incomplete or over-specified

but DL learns features adaptively and quite fast. In addition, it provides a flexible and almost universal learnable framework for representing different things, from images to linguistic information. DL also can be used in both supervised and unsupervised way.

Ideas behind DL were around for a quite long time but actually starting in 2006 was when these techniques outperformed traditional ML approaches. With the exploding amount of data, nowadays it is important to mention that DL benefits more from a huge amount of data rather than traditional approaches. Also in the era of parallel computing and parallel programming, it is quite an issue to be able to use multicore CPUs and specially GPUs and DL benefits from these technologies to perform efficiently. New ideas in recent years show much better optimization and specially better accuracy using DL.

The idea of DL has a long history with different names following different philosophical view points and a lot of ups and downs. Increasing training data and computational power helped DL to improve.

DL waxed and waned through time. The last wave started with a break through on a large dataset in speech recognition in 2010 by Dahl et al. (2012).

Later in 2012 Alex Krizhevsky and Ilya Sutskever, and Geoffrey Hinton by presenting an architecture which is now called "AlexNet" in a paper titled "ImageNet Classification with Deep Convolutional Networks" using a large, deep convolutional network won ILSVRC (ImageNet Large-Scale Visual Recognition Challenge) and opened DL door to computer vision (Krizhevsky et al., 2012).

Until a few years ago DL was mostly used in the area of computer vision. We see a lot of great progress in NLP exploiting DL in recent years across NLP levels such as speech, morphology, syntax, and semantics and NLP applications such as machine translation, sentiment analysis, and question answering.

Traditionally, phonemes, morphemes, syntax, and semantics are studied as distinct problems. In contrast, DL approaches learn to predict all of them from sound features. Sound features are represented as vectors and it is easy for neural networks to combine two vectors into one, and make words or sentences. DL uses vectors to deal with different tasks. As long as you can make vectors of number from data, you can use DL approaches. Therefore, the first step is to make vectors from words.

2 WORD VECTORS REPRESENTATIONS

According to the Webster dictionary, the meaning of the "word" is the idea that represented by a word or phrase, or the idea that a person wants to express by using words, signs, etc., which it can be expressed in a work of writing, art, etc. To begin with, the history of representing meaning in a computer is briefly reviewed to demonstrate how the idea of word vectors has

emerged. The history of representing meaning in computer, started with the question of how to represent meaning in a computer. The common answer is to use a taxonomy such as ImageNet or WordNet (which was a great project in many years) that has hypernyms (is-a) relationships. It is basically a very large graph that define various and diverse relationships between words. For instance, all hypernyms of the word "panda" is orderly as follows:

- Procyonid
- Carnivore
- Placental
- Mammal
- Vertebrate
- Chordate
- Animal
- Organism
- Living thing
- Whole
- Object
- Physical entity
- Entity

There are different tree structures in WordNet and everything ends with being an "entity." More precisely, it is a data directed acyclic graph. In the above example, it starts with the entity. Any entity is either a physical entity or an abstract entity. It continues to go down a deep structure until it gets to carnivore and procyonid and finally all the way down, the word "panda." Thus, the word panda can be defined as "the panda is a procyonid/carnivore/placental/mammal/…". This is one way that can be tried to have the computer understand or represent the meaning of the word by basically connect it to other words that are structured in the graph.

In addition to nouns, this idea can be applied to adjectives as well as using synonym sets. According to the creators of synsets, they are basically a list of words that has almost same meaning with intended word. Each word (which can be a noun, adverb, adjective, etc., according to its context) has different meanings and each meaning has different synonyms.

There are a lot of issues and problems regarding aforementioned discrete representation of a word. They are a great resource for the related words with the same or at least close meaning, however, they are missing nuances. For example, the word "good" has diverse synonyms such as commodity, adept, expert, good, practiced, proficient, skillful which can be misleading. All these synonyms have almost same meaning but they are applicable in different contexts.

Another issue is that the language is evolving and numerous words are being generated and it is impossible to keep up to date. It is going to be tough for a computer to be able to capture these new words with the right meaning.

In addition, this approach is subjective and also requires a great deal of human labor to create and adapt the word graph or synsets. As an evidence, all of the presents word graphs and synsets are not vast and thorough enough and not well maintained and updated.

As the worst part, it is hard to compute accurate word similarity. For instance, synsets can only response with one dimensional (binary) answer that two words are synonyms or not. Therefore, the vast majority of rule-based and statistical NLP work regards words as atomic symbols.

In vector space terms, a word is represented by a vector with one 1 and lots of zeroes. For example, if the word "hotel" is assigned with a vector so that the computer can accurately represent the word, a vector like the following is assigned.

$$[0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 1\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0]$$

This is called a "one-on" or "one-hot" vector that describes a word in a simplest way possible for the computer. It is still very common in NLP systems and generally used in the industry. Obviously, if you have a 20K vocabulary, the dimension of the vector is 20K vectors which every one of them has only one 1 in the vector and the rest are zeroes. As the result, if a document is to be represented by one-hot vectors, there will be a bunch of one-hot vectors each representing a word appeared in the document. This is called "bag of words representation." If you have a large vocabulary (e.g., 13M words in Google 1T) those kinds of vectors will be gigantic.

The main problem is the lack of connection between words with same or close meanings. For example, the word "awesome" has no connection with the word "wonderful" or "great." Consequently, there is no way to determine which of the words are similar in this way of representing the words. If you train a model in this context, the similarity of the words "motel" and "hotel" will be zero:

$$\text{Motel}[0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 1\ 0\ 0\ 0\ 0\ 0\ 0]\text{AND}$$
$$\text{Hotel}[0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 1\ 0\ 0\ 0\ 0] = 0$$

In order to overcome above discussed deficiencies, one of ideas which underlie a lot of old and traditional representation for words as well as the newest ones is distributional similarity-based representations which represent a word by its neighbors. As an example, in order to represent the word "banking," all the words in the left and right side of the "banking" in the following sentence represent "banking."

government debt problems turning into banking *crises as happened in saying that Europe needs unified* banking *regulation to replace hodgepodge*

Consequently, banking or bank can have "debt problems." Hence, instead of having one index, a set of indices can represent a word and you already have gained something about the word. The primary question is how to make

neighbors represent words. The basic answer is collecting large text corpus with cooccurrence matrix X. There are two options to compute X. The first one is full document in which all the words in the document represent the word regardless of the any relationships between the words, for example, the words behind or after the word which is going to be represented. Document cooccurrence matrix will give general topics leading to "Latent Semantic Analysis." Thus, all sports terms will have similar entries, e.g., swimming, wet, boat and ship will be all similar to one another as they all appear in a boating topic and any notion of syntax and part of speech will be loosed when only the whole document is considered. This approach is capable of identifying the general topic of a document.

Instead of the full document option, window around each word (words in left and right) is looked up which captures both syntactic and semantic information. To start with window-based cooccurrence matrix, a simple example is presented to describe what should traditionally be done in this approach and a very simple cooccurrence matrix will be created of a very simple corpus as follows.

- I like deep learning.
- I like NLP.
- I enjoy flying.

In order to essentially build cooccurrence matrix X, symmetric window (irrelevant whether left or right context) with the length of 1 (5–10 is more common) will be utilized and the Matrix X is as follows.

Counts	I	like	enjoy	deep	learning	NLP	flying	.
I	0	2	1	0	0	0	0	0
like	2	0	0	1	0	1	0	0
enjoy	1	0	0	0	0	0	1	0
deep	0	1	0	0	1	0	0	0
learning	0	0	0	1	0	0	0	1
NLP	0	1	0	0	0	0	0	1
flying	0	0	1	0	0	0	0	1
.	0	0	0	0	1	1	1	0

According to the matrix, for example, it can be captured that enjoy and like are might be more similar to each other than enjoy and deep. This is an improvement to one-hot representation.

There are also problems with simple cooccurrence vectors such as increase in size with vocabulary (each additional word requires an extra column and row in the matrix), very high dimensional and requiring a lot of storage and

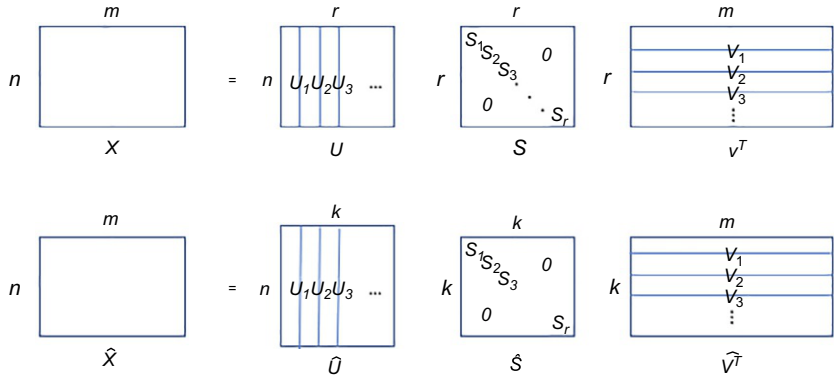

FIG. 1 Singular value decomposition for dimensionality reduction.

subsequent classification models have sparsity issues (the importance weight of each of the dimensions will be challenging), thus, models are less robust. The solution is low-dimensional vectors with the idea of a dense vector which is storing "most" of the important information in a fixed, small number of dimensions (usually between 25 and 1000 dimensions).

The first common method for Dimensionality Reduction on X is singular value decomposition (SVD) of cooccurrence matrix (Fig. 1).

$\hat{X}$ is the best rank k approximation in terms of least squares. For the previous corpus, applying SVD and plotting (two-dimensional) first two columns of U corresponding to the two biggest singular values, the result is as follows.

According to the plot, it can be interpreted that "like" and "enjoy" as well as "NLP" and "deep" are more similar and "learning" and "flying" are close to one another. Therefore, some information from our simple example of corpus is captured just using SVD. In the DL models (as well as all subsequent models), a dense vector represents a word (following example).

$$Linguistics = \begin{pmatrix} 0.286 \\ 0.792 \\ -0.177 \\ 0.107 \\ 0.109 \\ -0.542 \\ 0.349 \\ 0.271 \end{pmatrix}$$

There exist problems regarding dense vectors. There are some words that have high frequency in the corpus that do not add much in the meaning such as the, he, has, is, are, etc., which earn high values in dense vector and syntax has too much impact. As the solution, all these words can be ignored or using $min(X, t)$ for the matrix X such as $t \backsim 100$. Another problem is ramped

windows that count closer words more. Instead of using counts, correlation between words can be employed and negative values can be set to zeros. Word vectors result in valuable semantic patterns which demonstrates groups of words with close meanings (Rohde et al., 2006) (Fig. 2).

Another interesting example of the outputs of SVD is the capturing of close meaning between nouns and their corresponding words or words representing similar activities (Fig. 3).

There are some problems regarding SVD for word vectors. When new words or new documents are added, you should rerun the entire SVD. Online SVD is one of the solutions, however, the computational cost increases quadratically when the size of the matrices grows. Also, SVD has a different learning regime than other DL models. Based on single windows, the (old) idea is to directly learn low-dimensional word vectors. A recent developed model, known as word2vec, which is simpler and faster is presented by Mikolov et al. (2013b) and it is described in details in this section.

The main idea of word2vec is to predict surrounding words of every word instead of directly capturing cooccurrence counts. Each example will be probed one at a time to predict surrounding words and eventually it also captures cooccurrence statistics in an online way by updating on new words, sentences, or documents. Thus, when a new sentence or document is inserted, they can easily and faster be incorporated.

In word2vec, the cooccurrence window with the length of m (the number of words) is defined. The objective function is to maximize the log probability of any context word given the current center word.

$$J(\theta) = \frac{1}{T}\sum_{n=1}^{T} \log p(\omega_{t+j}|\omega_t)$$

in which θ is all variables that should be optimized. T is the size of corpus and we try to maximize log probability of each word at a time considering m words behind and after.

In order to represent the probability of each word conditioned by the center word, the simplest formulation for $p(\omega_{t+j}|\omega_t)$ is

$$p(o|c) = \frac{\exp(u_o^T v_c)}{\sum_{\omega=1}^{W} \exp(u_\omega^T v_c)}$$

where o is the outside word id, c is the center word id, u and v are center and outside vectors of o and c, respectively. Every word has two vectors including one vector represented as outside word and one vector which its outside words are tried to be predicted. To illustrate the model, consider a sentence with 20 words and $m = 2$. The first center word is the third word and we try to predict two words around the third word. Then, the forth word is selected as the center word and the two words around that word is predicted. This procedure

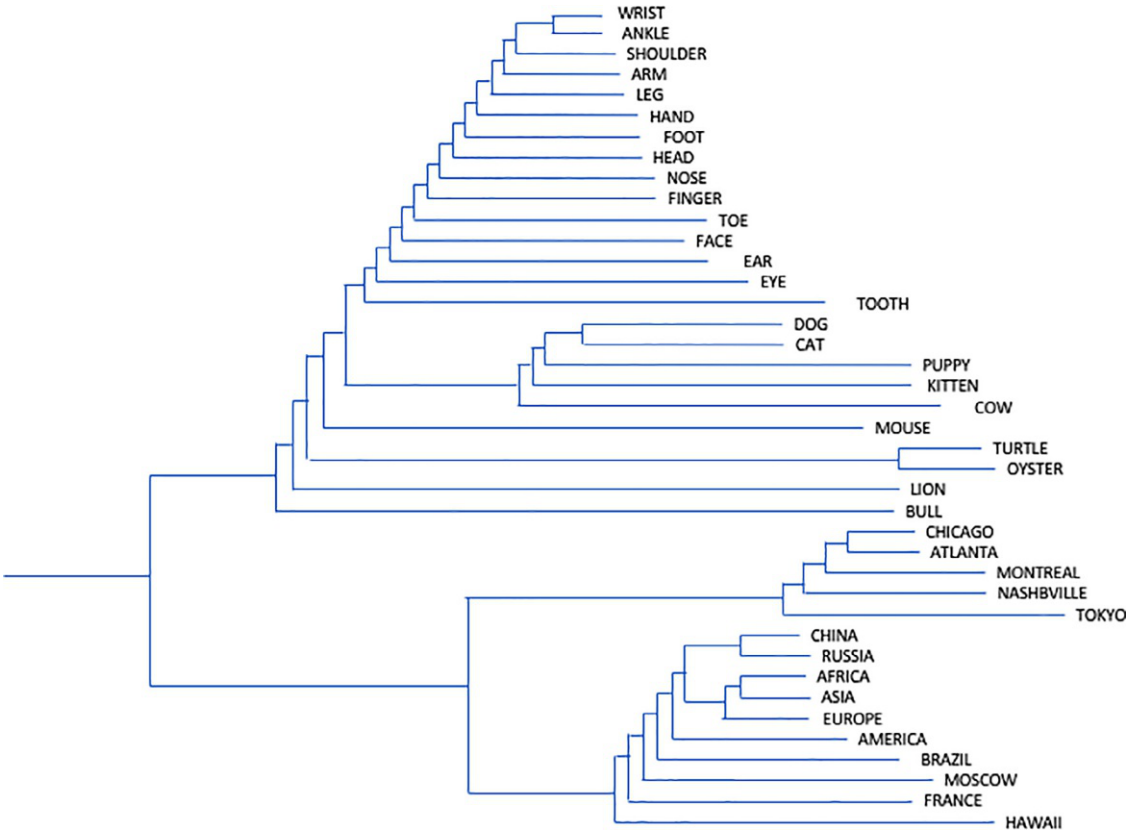

FIG. 2 Semantic pattern of groups of words.

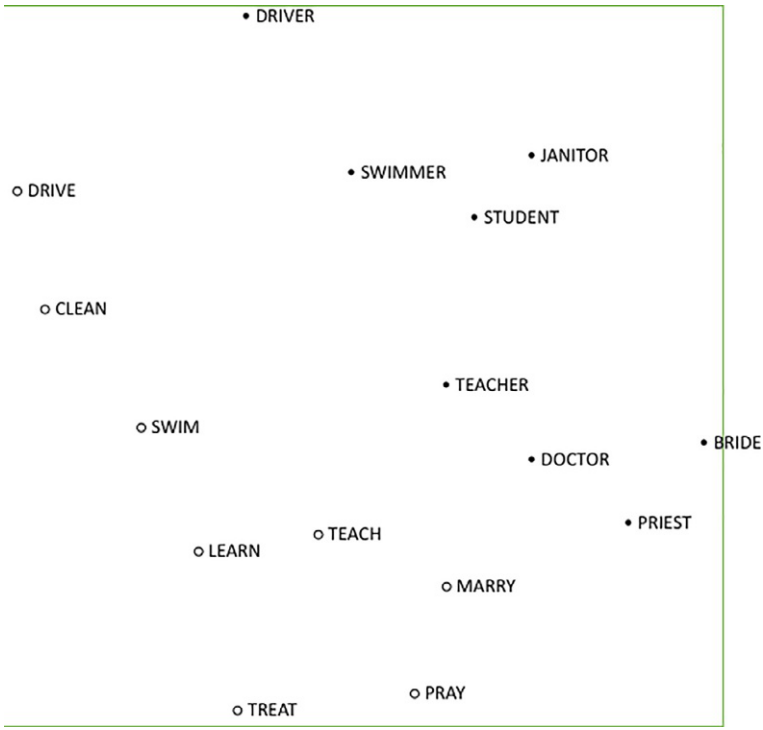

FIG. 3 An example of outputs of SVD in capturing close meanings between nouns and their corresponding words or words representing similar activities.

continues one word at a time and the last center word is 18th word. For each single window, objective function, $J(\theta)$ should be optimized. Simply, it can be minimized using gradient descent.

When the corpus is large, this objective function is not scalable and the training process is slow because the corresponding calculations are highly expensive. However, if we are able to optimize the objective function efficiently, interesting results can be captured. Analogies testing dimensions of similarity can be solved by applying vector subtraction in the embedding space. Syntactically, the relationship between single and plural words will result in the following.

$$x_{\text{apple}} - x_{\text{apples}} \approx x_{\text{house}} - x_{\text{houses}} \approx x_{\text{flower}} - x_{\text{flowers}}$$

The vector direction for the apple minus apples is similar to the flower minus flowers. This is similar for verbs and adjectives morphological forms. We will discuss it later why this method captures these similarities. Semantically, the vector direction of $x_{\text{pants}} - x_{\text{clothing}}$ is similar to $x_{\text{plate}} - x_{\text{dishes}}$ which provides some kind of taxonomy and it can be inferred that something is the subset of something.

After capturing these results, we review how the new methods relate to previous ones. Both count based and direction prediction have some advantages and disadvantages. Method like SVD are trained fast and use the statistics efficiently. However, they mainly used to capture word similarities and the aforementioned relationships are missed and disproportionate importance given to large counts (which can easily be solved by having caps on those counts). In contrast, skip-gram, NNLM, and other similar models scales with corpus size and use the statistics inefficiently, but they generate improved performance on other tasks and can capture complex patterns beyond word similarities (such as previous example patterns).

Combining the best of two worlds results in Global Vector (GloVe) model (Pennington et al., 2014). Instead of going over one window at a time, we may collect the whole corpus and basically collect all the cooccurrence statistics. Now we can run SVD only on the samples indices and optimizing following function for all ij pair of words.

$$J(\theta) = \frac{1}{2}\sum_{i,j=1}^{W} f(P_{ij})(u_i^T v_j - \log P_{ij})^2$$

P_{ij} is the probability of how often words i and j cooccurs. Because we have the sum of word pairs, some of pairs that appear so often a text, for instance the pair of "he" and "has," have large impact on the model and we need to set a maximum for f function and to prevent large counts dominating the entire objective function. Otherwise the model spends a lot of its parameter to capture those word pairs and may dominate the rest of the optimization. This model is scalable to huge corpora, trains fast and also has proper performance with small corpus and small vectors.

After we train the word vectors, we end up to U and V vectors from all u and v vectors. Since both sets of vectors capture similarities, the best solution is to sum up both of them.

$$X_{\text{final}} = U + V$$

To illustrate the results of the GloVe, following is the nearest words for the word "frog" that GloVe determined.

1. Frogs
2. Toad
3. Litoria
4. Leptodactylidae
5. Rana
6. Lizard
7. Eleutherodactylus

As shown, the rare word "eleutherodactylus" which is a kind of frog is detected.

Now the question is how to evaluate word vectors. In general language processing and in most ML fields, on some levels we have two options for evaluation including intrinsic and extrinsic tasks. Intrinsic tasks usually evaluate on a specific/intermediate subtask. For example, how often a specific relationship between two words like man/king or woman/queen or "dollar + USA −Mexico" appears. At the end, the result may not be cared but it helps to evaluate word vectors quickly as the model is in progress. In more complex natural language processing models it becomes useful like machine translation, synonym analysis, or question answering. Intrinsic tasks iterate fast and help to understand that specific model (in our case word vector model). However, when it is applied, from time to time it is needed to be sure that there is some correlation between your intrinsic tasks.

On the other hand, extrinsic tasks are basically real tasks such as question answering and semantic analysis. In general, they are preferable to intrinsic tasks but more time consuming to compute accuracy. In addition, once the whole model is being evaluated sometimes it is unclear if the system is improved because your subsystem improved or other pieces of the model are improved. In general, having both tasks is the best choice but it is hard to implement them together for most cases.

In order to evaluate intrinsic word vectors, one of the popular ones is Word Vector Analogies. For instance, if we want to find out "man" is to "woman" as "king" to something else, we hope that in the geometry of the space, this is captured. In this example, if we take the vector difference of "man" minus "woman" and do the same for the "king," ideally it appears a word vector that places "queen." We can use the following cosine distance after addition captures in which we find the argmax (the vector that maximizes the cosine distance).

$$d = argmax \frac{(x_b - x_a = x_c)^T x_i}{||x_b - x_a + x_c||}$$

It should be noted that the input words ("man", "woman," and "king") are discarded because they often comes out as the best answer. This is a popular approach, however, the information may exist but they may not be linear. It is possible that a more complex nonlinear function that capture these analogies. It is interesting that there is no extra function needed to be trained. It falls out the cooccurrence statistics and able to be captured GloVe word2vec kinds of model.

When the PCA is run over a set of pair of words, the result can be as the following example in Figs. 4 and 5. As shown, vector differences are similar and it allows us to interfere some conclusions. For example, woman−man+ madam = sir.

As an another example, it works for the companies and CEOs and also superlatives as Fig. 6.

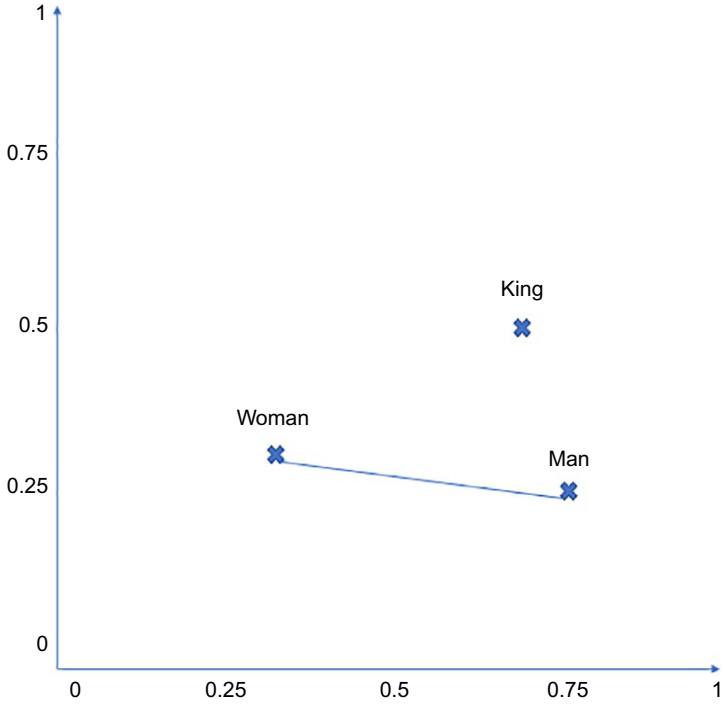

FIG. 4 Results of running PCA over a set of pair of words.

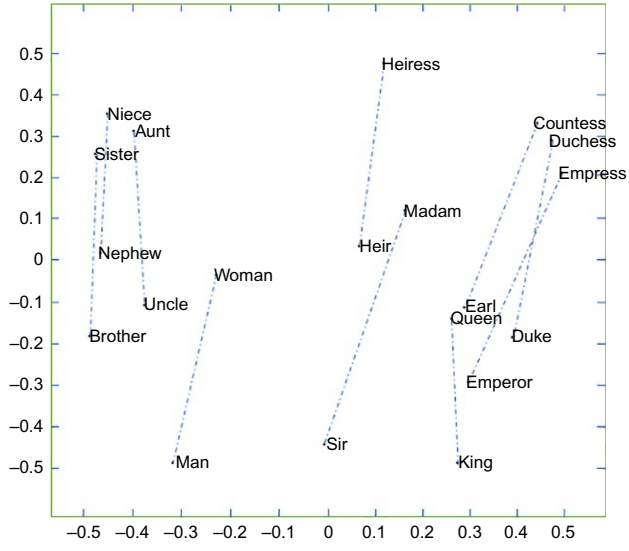

FIG. 5 Word vector analogy example.

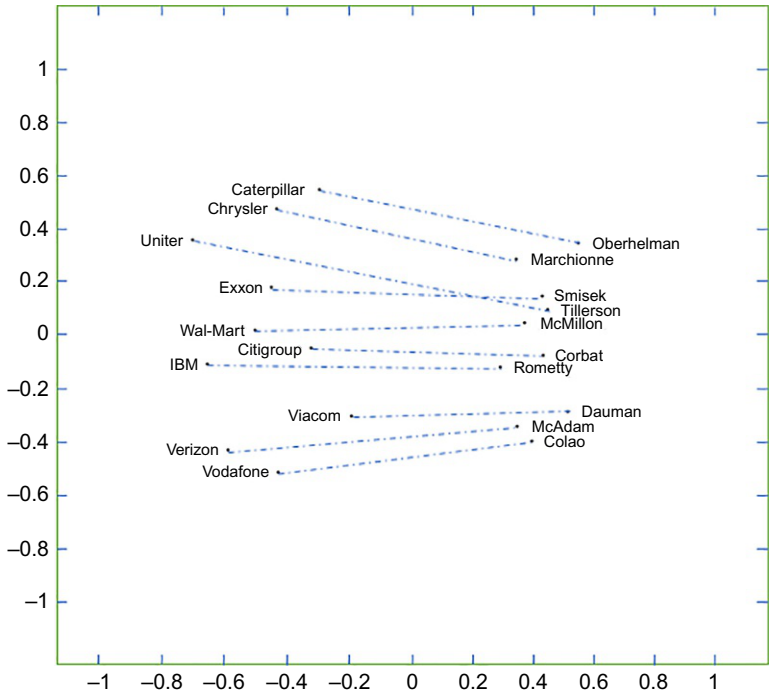

FIG. 6 Word vector analogy example—companies and their CEOs.

There are lots of similarities that can be obtained such as capital city and country, state and city, superlatives, past tense of a verb, etc.

In order to determine hyperparameters, at first we can take a look on the results of applying different models for a corpora including GloVe as shown in Fig. 7.

As demonstrated in Fig. 7, dimension and size are increased one at a time to precisely understand the effect of their changes and these are vital for the evaluation of the model. According to the results, when size and dimension are increased, the results improve and GloVe performs much better in comparison with other recent models. Underlined values shows which model has the best performance for a specific range of dimension and size and bold-underlined values show the best performance obtained among all the tested combinations of size and dimension. In every ML project, it is highly recommended to compute and visualize the performance in as many plots as possible. Both semantic and syntactic plots can be demonstrated, but we can basically care about the overall result of both (Fig. 8).

According to the plots, best dimension is about 300 (which may be slightly drop-off afterward) and the window size of 8 around each center word works well (too large window weakens the performance of the model and looses syntactic structure).

Model	Dim.	Size	Sem.	Syn.	Tot.
ivLBL	100	1.5B	55.9	50.1	53.2
HPCA	100	1.6B	4.2	16.4	10.8
GloVe	100	1.6B	67.5	54.3	60.3
SG	300	1B	61	61	61
CBOW	300	1.6B	16.1	52.6	36.1
vLBL	300	1.5B	54.2	64.8	60.0
ivLBL	300	1.5B	65.2	63.0	64.0
GloVe	300	1.6B	80.8	61.5	70.3
SVD	300	6B	6.3	8.1	7.3
SVD-S	300	6B	36.7	46.6	42.1
SVD-L	300	6B	56.6	63.0	60.1
CBOW	300	6B	63.6	67.4	65.7
SG	300	6B	73.0	66.0	69.1
GloVe	300	6B	77.4	61.5	71.7
CBOW	1000	6B	57.3	68.9	63.7
SG	1000	6B	66.1	65.1	65.6
SVD-L	300	42B	38.4	58.2	49.2
GloVe	300	42B	**81.9**	**69.3**	**75.0**

FIG. 7 Analogy evaluation and hyperparameters.

Another critical hyperparameter is the time required for training the model. There should be a compromise between efficiency and effectiveness of the model when the parameters such as size and window size (and even the model employed) is being set (Fig. 9).

Usually, having more data helps the model, however, if you need to know what data are better to choose, Wikipedia is better than news text. As illustrated in Fig. 10, using Wikipedia2014 with 1.6 billion tokens gains better overall accuracy than Gigawords with 4.3 billion tokens.

Another common intrinsic word vector evaluation is investigating word vector distances and correlation. If we ask a number of humans about the similarity between two words on a scale like 1–10 (one means the pair of words have nothing to do with each other and 10 means that the words are exactly the same) and collect the results, the effectiveness of the model can be evaluated. The potential problems are that it is somehow subjective, culture-dependent, etc. The correlation between the results of the different models and human similarity judgment enables us to compare them in the mean of their capability of capturing word similarities (Fig. 11).

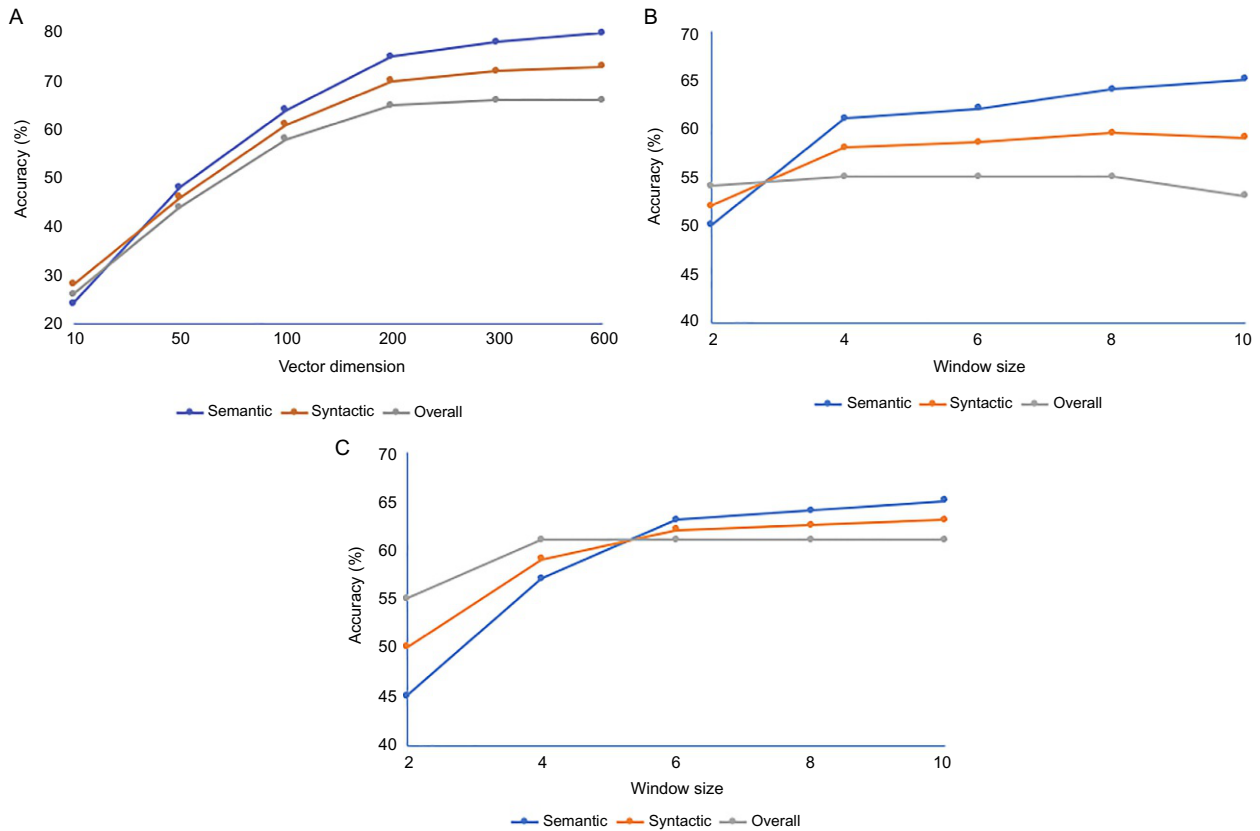

FIG. 8 Accuracy vs vector dimension and window size for symmetric and asymmetric context. (A) Symmetric context. (B) Symmetric context. (C) Asymmetric context.

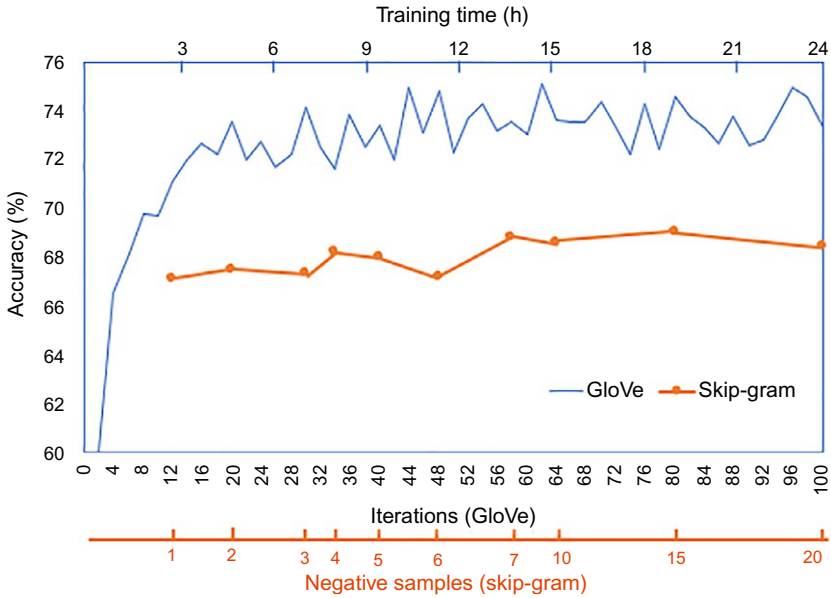

FIG. 9 Time required for training the model.

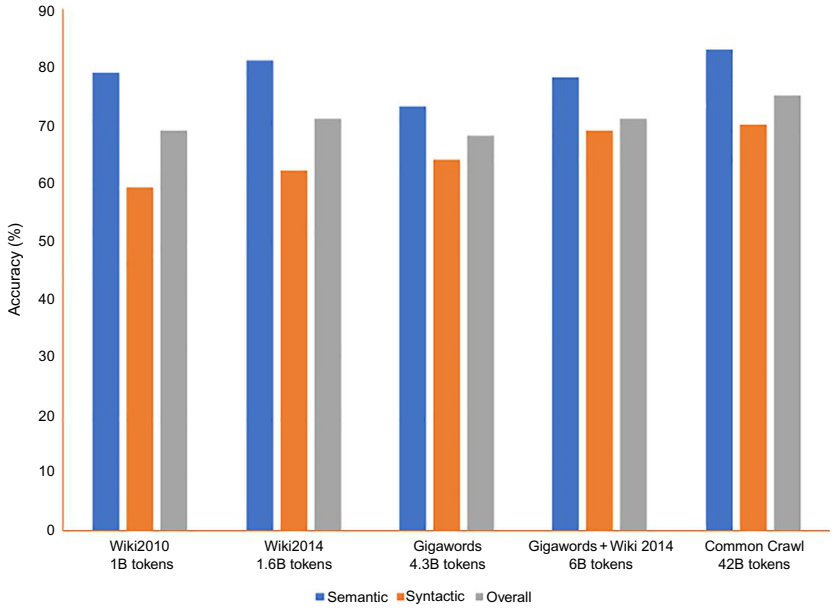

FIG. 10 Accuracy vs data.

Model	Size	WS353	MC	RG	SCWS	RW
SVD	6B	35.3	35.1	42.5	38.3	25.6
SVD-S	6B	56.5	71.5	71.0	53.6	34.7
SVD-L	6B	65.7	72.7	75.1	56.5	37.0
CBOW	6B	57.2	65.6	68.2	57.0	32.5
SG	6B	62.8	65.2	69.7	58.1	37.2
GloVe	6B	65.8	72.7	77.8	53.9	38.1
SVD-L	42B	74.0	76.4	74.1	58.3	39.9
GloVe	42B	**75.9**	**83.6**	**82.9**	**59.6**	**47.8**
CBOW*	100B	68.4	79.6	75.4	59.4	45.5

FIG. 11 Correlation evaluation.

One of the dilemmas is word ambiguity. For instance, a same word can be used in different parts of speech. The word "run" can be a verb or a noun. When such words appear in a text, if it is used, for example, as noun, then the vector is pulled in the direction which identifies the word as noun. Consequently, word vectors may be pulled in different directions for words with ambiguity. As one way to deal with this, we can cluster the word window and retain with each word assigned to multiple different clusters ($bank_1$, $bank_2$, …). Fig. 12 illustrates the results of applying this idea. As shown, $bank_2$ is close to financing, banking, and transaction (in verb cluster).

When using DL for word vectors, there are some benefits in single word classification. It has the ability to classify words accurately. In addition, word vectors can capture facts.

When we have trained word vectors, we want to train a system that provides us useful outputs such as prediction or classification. Therefore, as well as training word vectors for their own sake, we train them for a purpose. The simplest expected outcome can be classifying words in different categories. In this regard, the softmax is one of the most basic blocks for the neural networks which will be briefly described here.

Consider a word vector x which is the average of u and v in most cases and the notation usually used is as follows for the probability for class y given a random vector x.

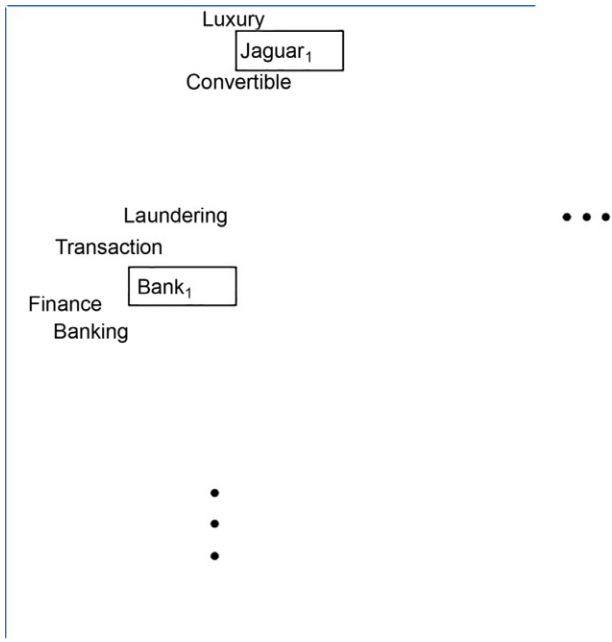

FIG. 12 Improving word representations by global context and multiple word prototypes.

$$p(y|x) = \frac{\exp(W_{y\cdot}x)}{\sum_{c=1}^{C}\exp(W_{c\cdot}x)}$$

where $W \in \mathbb{R}^{C\times d}$ and $W_{y\cdot}$ measure that we take the y row of our matrix W. We have C classes or rows and d columns. We exchangeably use logistic regression and softmax classification on word vector x to obtain probability of class y. As we train the softmax, different terminology can be used including loss function, cost function and objective function. As we compute $p(y|x)$, first we take the y's row of W and multiply that with row with x:

$$W_{y\cdot}x = \sum_{i=1}^{d} W_{y_i}x_i = f_y$$

Then we compute all f_c for $c = 1, \ldots, C$ and finally, we perform normalization to obtain probability with softmax function. In our case, the loss function tries to maximize the probability of the correct class y. Hence, we minimize the negative log probability of that class.

$$-\log p(y|x) = -\log\left(\frac{\exp(f_y)}{\sum_{c=1}^{C}\exp(f_c)}\right)$$

To provide a background regarding cross entropy error, assuming a ground truth probability distribution that is 1 at the right class and 0 elsewhere, if $p = [0, \ldots, 0, 1, 0, \ldots, 0]$ and our computed probability is q, then the cross entropy is

$$H(p,q) = -\sum_{c=1}^{C} p(c)\log q(c)$$

Because of one-hot p, p_c can only be valued zero or 1 and the only term left is the negative probability of the true class.

Cross entropy can be rewritten in terms of the entropy and Kullback–Leibler divergence between two distributions.

$$H(p,q) = H(p) + D_{\mathrm{KL}}(p\|q)$$

Since $H(p)$ in our case is zero and has no contribution to gradient, only KL divergence needs to be minimized. KL divergence is not a distance but a non-symmetric measure of the difference between two probability distributions p and q.

$$D_{\mathrm{KL}}(p\|q) = \sum_{c=1}^{C} p(c)\log\frac{p(c)}{q(c)}$$

Considering the main aforementioned objective function, in order to understand what is done at each window, consider the following corpus.

I bought a new camera yesterday.

Assuming a window size of $m = 1$ (looking up one word left and one word right for each window), we start at word "bought." Thus, we have word vector v for the "bought," v_{bought}, vector u for the outside word "I," u_{I}, and vector u for "a," u_{a}. Then, we try to predict which word index we have for the outside word vectors from v. u_{I} and v_{bought} are the first pair that should be considered. According to objective function, on the first step in this window, we have the first element $\exp(v_{\mathrm{bought}}^T u_{\mathrm{I}})$? and second element $\exp(v_{\mathrm{bought}}^T u_{\mathrm{a}})$. Then we use the numerator *sum* for over all the words in both cases.

When we are in each window, we compute the updates for all parameters being used in that window including all the inside and outside vectors. Then, inside vector is moved to the next word which used to be an outside vector and depending on the window size some of the outside vectors may be still shared.

As we compute derivatives for all the parameters in a model, we often define one vector, θ, which becomes a high dimensional one.

$$\theta = \begin{pmatrix} v_{\text{Learning}} \\ u_{\text{NLP}} \\ . \\ . \\ . \\ v_{\text{Learning}} \\ u_{\text{NLP}} \\ . \\ . \\ v_{\text{Learning}} \\ . \end{pmatrix} \in \mathbb{R}^{2dV}$$

d is usually between 25 and 500 and V can be hundreds of thousands.

In order to optimize (in most cases minimize) the objective function $J(\theta)$ over all the training data, the gradients for all windows should be computed. When we have a large corpus with millions of words, then we have many millions of windows. Considering α as step size, next update for each element of θ is as follows:

$$\theta_J^{\text{new}} = \theta_J^{\text{old}} - \alpha \frac{\partial}{\partial \theta_J^{\text{old}}} J(\theta)$$

In matrix notation for all parameters we have

$$\theta^{\text{new}} = \theta^{\text{old}} - \alpha \frac{\partial}{\partial \theta^{\text{old}}} J(\theta)$$
$$\theta^{\text{new}} = \theta^{\text{old}} - \alpha \nabla_\theta J(\theta)$$

A simple computation is Vanilla Gradient Descent Code which is simple in math and code.

while true:

theta_grad = evaluate_gradient (J, corpus, theta)
theta = theta − alpha * theta_grad

A corpus may contain 40 billion tokens and hence 40 billion windows and if we evaluate all of them for a single update, it takes a very long time to model does anything and the model cannot be evaluated until the update finishes. Therefore, this is not an efficient idea in almost all cases.

As an alternate, we consider a time step t and we will update parameters right after each window t, thus, we will make one small update at each window. This is called Stochastic Gradient Descent (SGD) as follows.

$$\theta^{\text{new}} = \theta^{\text{old}} - \alpha \nabla_\theta J_t(\theta)$$

while true:

window = sample_window (corpus)
theta_grad = evaluate_gradient (J, window, theta)
theta = theta − alpha * theta_grad

But, in each window, we only have at most 2c-1 words, thus, $\nabla_\theta J_t(\theta)$ becomes sparse. Considering the corpus "I like cats," only three vectors will be populated in this very high dimensional update vector. We are going to have only one inside vector v_{like} and two outside vector u_{I} and u_{cats}.

$$\nabla_\theta J_t(\theta) = \begin{pmatrix} 0 \\ . \\ . \\ \nabla_{v_{\text{like}}} \\ . \\ . \\ 0 \\ \nabla_{u_{\text{I}}} \\ . \\ . \\ . \\ \nabla_{u_{\text{learning}}} \\ . \\ . \end{pmatrix} \in \mathbb{R}^{2dV}$$

Then, there will be millions of numbers that are zero and it results in an inefficient code. We can only update those word vectors that actually appear at each window and not all the zeroes. We can keep around hash for word vectors as well as only update certain columns of full embedding Matrix U and V. It prevents sending a large amount of updates around.

As discussed before, the normalization factor is computationally expensive to compute all word pairs of inside and outside word vectors.

$$P(o|c) = \frac{\exp(u_o^T v_c)}{\sum_{w=1}^{W} \exp(u_w^T v_c)}$$

Instead, we can implement a model to reduce calculations. One of the popular models is skip-gram model. The main idea is that instead of having a soft matrix that lowers all the probabilities for all words that do not happen to appear, we only at a true pair of a center word and its context window and look at some other random pairs (the center word with a random word). According to Mikolov et al. (2013a), overall objective function for the skip-gram model and negative sampling is $J(\theta) = \frac{1}{T}\sum_{t=1}^{T} J_t(\theta)$ and

$$J_t(\theta) = \log\sigma(u_o^T v_c) + \sum_{i=1}^{k} \mathbb{E}_{j\sim P(w)}[\log\sigma(-u_j^T v_c)]$$

where k is the number of negative samples and we use the sigmoid function $\sigma(x) = \frac{1}{1+e^{-x}}$ and maximize the probability of two words cooccurring in first log.

A more clearer notation of the above equation is as follows:

$$J_t(\theta) = \log\sigma(u_o^T v_c) + \sum_{j\sim P(w)} [\log\sigma(-u_j^T v_c)]$$

The probability that real outside word appears is maximized and the probability that random words appear is minimized. Note that $P(w) = U(w)^{3/4}/Z$ where the unigram distribution $U(w)$ is raised to 3/4 power and the power makes less frequent words be samples more often (which may improve the outcome of the model).

Another model to reduce calculations, which we just mention here, is the continuous bag of words (CBOW) model in which center word is predicted from sum of surrounding word vectors (the average of surrounding words) instead of predicting surrounding single words from center word (which is the idea of skip-gram model).

3 FEEDFORWARD NEURAL NETWORKS

Feedforward neural networks go by different names including multilayer perceptrons (MLPs), deep feedforward networks, and so forth. These networks try to find some function $\hat{f}(x,\theta)$ which is the approximation of the true function f and then proceed to learn the optimum values for θ to make the best approximation. The word "feedforward" means that data just flow in one direction: from the input x to the output y. There are no feedback connections. Actually, if we add feedback connections to feedforward neural networks it will be recurrent neural networks which are discussed later. Feedforward networks are very important and widely used for commercial applications. Also well-known convolutional neural networks are a specialized type of feedforward networks. Feedforward networks are the most important Lego block toward recurrent networks, which by far are the most powerful type of networks in natural language processing applications.

The word "network" comes from the using graph structure to show how functions are related to each other. For example, if our model consists three functions $f^{(1)}$, $f^{(2)}$, and $f^{(3)}$ connecting to each other in the chain structure $f(x) = f^{(3)}(f^{(2)}(f^{(1)}(x)))$, $f^{(1)}$ is what we call the first layer, $f^{(2)}$ is the second layer and so forth.

The final layer is called the output layer. Using training data we try to push our $\hat{f}$ toward the true function to make a more accurate approximation.

Training data show what should be the result of the output layer but it is the training algorithm which using training data decide how to changes parameters of the model for the intermediate layers to produce the desired output. These layers are called hidden layers. Each layer consists of some units working in parallel. The dimensionality of these hidden layers or the number of units in each layer determines the width of the model. We call this unites neurons and so the name neural networks. The term "neural" is used because the idea behind these networks is inspired by neuroscience and the mechanism in which neurons work in brain. So each neuron receives some input from other neurons and compute and activation value.

To set up our notation we start with a review of classification using linear models and then from there we follow our way to find how we can overcome their limitation using neural networks.

Linear models like logistic regression have their own place in literature and because of some properties like fitting efficiently and reliably are interesting and attractive to use. In general we have a training dataset which includes one to N training samples which we show as $\{x_i, y_i\}_{i=1}^{N}$ in which x_i is input which can be a word, a window of multiple words, a whole sentence, or even entire documents. Based on the context of the problem, y_i is the output of the corresponding input which we try to predict. y_i in classification problems can be a label like named entities, sentiment, and it can be multiple words like a sequence of words (a sentence) which we confront in machine translation task.

In general ML problems, we use manually designed features and we want to find weights W for each of them. In classic classification problem we try to train a logistic regression to find a boundary which separate two classes. After finding weights, to find the probability of each input belonging to the class y, first we calculate scores for each class y by multiplying the y-th row of W with x.

$$W_y.x = \sum_{i=1}^{d} W_{y_i} x_i = f_y$$

where d is the dimension of the input. In the same way, we can calculate scores (f_c) for all the classes $c = 1, \ldots, C$. Then to compare these scores with each other to find the winner we need to normalize them. For this purpose traditionally *softmax* function is used. Using softmax, scores will be normalized between 0 and 1 which we can consider as the probability of the input x belonging to the class y.

$$p(y|x) = \frac{e^{f_y}}{\sum_{c=1}^{C} e^{f_c}}$$

Obviously the objective is to maximize the probability of the correct class y whose our input x belongs to. If we consider the ideal vector $P(c)$ as a one-hot vector for the right class c, we can measure how well we are doing using cross entropy loss function.

$$H(p,P) = -\sum_{c=1}^{C} P(c)\log p(c|x)$$

because of P being a on-hot vector, the only term left is the negative probability of the right class. Notice that maximizing the probability of the right class is the same objective as minimizing the negative logarithm of the probability.

$$Max(p(y|x)) = Max(\log(p(y|x))) = Min(-\log p(y|x))$$

Hence, in general for the whole dataset $\{x_i, y_i\}_{i=1}^{N}$ cross entropy loss function is

$$J(\theta) = \frac{1}{N}\sum_{i=1}^{N} -\log\left(\frac{e^{f_{y_i}}}{\sum_{c=1}^{C} e^{f_c}}\right)$$

where J is the overall objective function and θ is all the parameter and notice for the sake of simplicity f_y is not expanded in the objective function.

$$f_y = f_y(x) = W_{y\cdot}x = \sum_{j=1}^{d} W_{y_j}x_j$$

Notice that $W_{y\cdot}x$ is a dot product. The whole process of calculating scores and normalizing them through softmax function and then cross entropy error $H(\text{softmax}(WX + b), P)$ is called *multinomial logistic classification*.

Now we continue our way to use classification in the context of language. In the context of language single words are rarely classified because based on the context, words can have different and sometimes completely opposite meanings. These words are called autoantonyms. For example, "to sanction" can mean "to permit" or "to punish" and word "overlook" can have the meaning of "to supervise," or "to neglect" or "to seed" can mean "to remove seeds" or "to place seeds." Also based on the context, words can have different named entities. For example, "python" can be a programming language or a snake. Also, "Paris" can be a city or a person, Paris Hilton.

So the idea is to classify a word in the context of a "window" of its neighbor words. Considering name entity recognition task we need to classify each word as a location, person, organization, and so on. Window size can be different. For example, a window of size five includes the word we are trying to

classify in the center plus two previous words and next two words. There are many different ways of dealing with neighbor words. Maybe the first thing that comes to your mind is to average all the words in the window but then the problem is that this approach does not consider word orders and loses position information. Another idea is to concatenate all the word vectors in the window then use the softmax classifier on the whole window and use the cross entropy loss function as before.

3.1 Neural Networks

Problem with softmax or logistic regression and in general linear models is that they are linear! They just give us linear decision boundaries in the space and they miss understanding the interaction between input variables. Also, usually our boundaries are not linear and specially in high dimensional space we need a very flexible approach for classification to improve accuracy. There are ways of extending linear models like fitting the model on a generic nonlinear transformed input $\phi(x)$ but still they cannot memorize prior information to solve advanced tasks. There are even simple functions like XOR (exclusive or) that linear models cannot learn.

Here is where neural networks come in. Neural networks are very flexible in finding complex decision boundaries. As we discussed earlier, DL uses training data to learn θ and the representation $\phi(x)$ and maps from representation to desired output by learning w. So our model is $y = f(x, \theta, w) = w^T\phi(x, \theta)$. Training process is performed using an optimization algorithm.

We need to simulate a single neuron in mathematical notation. A single neuron consists of some inputs, a bias unit (like the intercept term), an activation function and output (Fig. 13). To shed some light on the secret of neural networks let us say if you understand the multinomial logistic classification you already understand how a basic neuron works.

$$h_{w,b}(x) = f(w^T x + b)$$

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

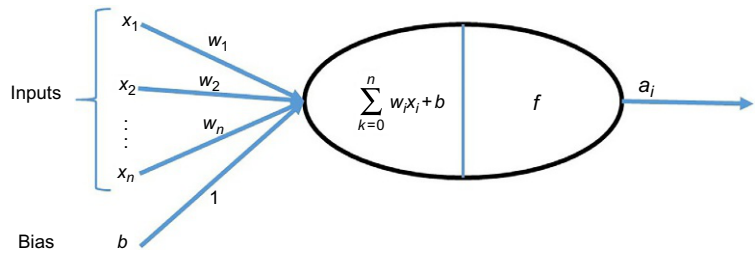

FIG. 13 A single neuron with n inputs, a bias, and one output.

The reason for adding function f, in this case sigmoid function, to our calculations is that without nonlinearity, adding more layers will not useful because each two linear layers (ignoring the bias term) can be compiled into a single linear layer (Fig. 14).

$$W_1 W_2 x = Wx$$

Obviously, we need to use a nonlinear function. One of the most common ways in neural networks to do it is by an affine transformation followed by a fixed, nonlinear function. We call this function activation function.

However, in the above notation, the activation function we used f is called a *sigmoid* function. There is some advantageous of using a sigmoid function. First of all, it has nice derivatives and also it compress the output value between 0 and 1. $h_{w,\,b}(x)$ is the output of the neuron. Now we feed a vector of inputs to some of these neurons and produce the output. In multinomial logistic classification we have a softmax at the end (or on top) that classifies these outputs but in neural networks we can feed these outputs to another layer of neurons as input. Our last layer is the loss function which shows how effective is our network. One of the most famous loss functions is cross entropy. Loss function will guide the intermediate neuron variables in a way that their output minimizes the loss function. These intermediate layers are called hidden layers (Fig. 15).

Now the questions are what if we add a bunch of more hidden layers? Yes! It will be a deep neural network. So as you see, deep neural networks are just concatenation of binary logistic regression units. Do not worry if you have questions. We will go through the details later.

To make it more clear let us define our terms in matrix notation.

$$z = Wx + b$$

$$a = f(z)$$

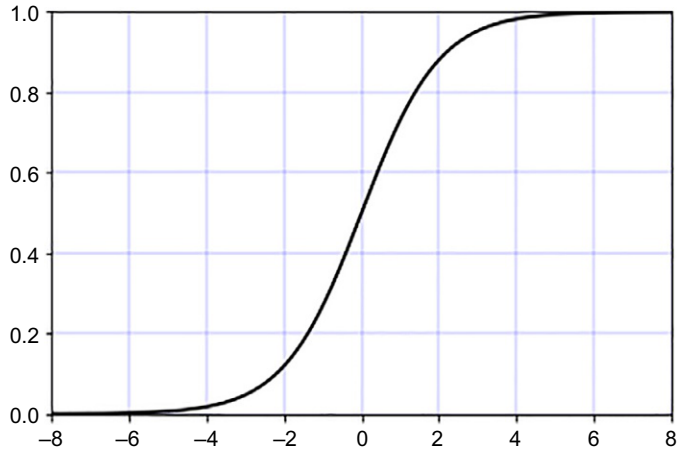

FIG. 14 Sigmoid function.

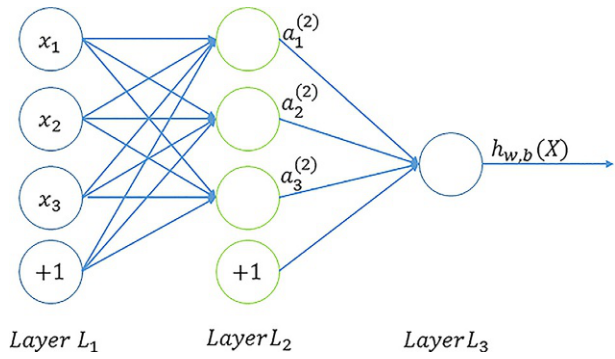

FIG. 15 A simple neural network.

This is exactly the same as multinomial regression classification just different names. Notice input to each neuron is sum of the inputs from previous layer and so the output will be

$$a_i = f(W_{i1}x_1) + W_{i2}x_2 + W_{i3}x_3 + b_i$$

where f is applied element wise:

$$f([z_1, z_2, z_3]) = [f(z_1), f(z_2), f(z_3)]$$

Coming back to our word window classification, assume we want to classify the center word to check if it is a location or not using a three-layer neural network.

Considering each word vector is a 4×1 vector and we have eight units in our hidden layer:

$$p(y|x) = softmax(W_2 a)$$
$$a = f(z)$$
$$z = W_1 x + b$$
$$x \in \mathbb{R}^{20\times 1}, W_1 \in \mathbb{R}^{8\times 20}, W_2 \in \mathbb{R}^{8\times 1}$$

This is the feedforward process. In the next section we will talk about how to train neural networks.

4 TRAINING DEEP MODELS AND OPTIMIZATION

Training a neural network is very similar to training any other ML model with gradient descent. Without going through the gory details, what we need to know is that training algorithm for neural networks is almost always an iterative, gradient-based optimizer which tries to decrease (or descend) the loss function rather than a linear equation solver or convex optimization solver which we use for logistic regression or SVMs. To use gradient-based algorithm we need to choose our loss function and output representation type.

Usually loss function needs a regularization term as a company. We will talk about regularization strategies later. The common practice in most of recent neural networks is to use maximum likelihood to train the model and by maximum likelihood we mean the negative log-likelihood or in another words the cross entropy between training data and model's results. One of the major reasons to use cross entropy loss function is that in many cases hidden layers use exp function to make the output and exp function becomes saturate (flat) if the argument is very negative and the log part in cross entropy cancels the exp function and so we can have a better gradient. Because zero gradient is problematic.

Output unit can have different types of output. Every type of output unit can also be used as a hidden unit. There are different kinds of output units. The most simple one is linear unit that make an affine transformation without nonlinearity: $\hat{y} = W^T + b$. One of the problems with linear functions is that the range of the function is not limited so interpretation of the result is difficult. For example, in binary classification we just need to know the answer is one or zero or more specifically we need to calculate probability of the $y = 1$ given x. $p(y = 1|x)$ One of the most common ways of handling this problem is to use Sigmoid function. Because in addition to compress the answer to lie in the range of zero and one which can be used as a probability, it also has strong and smooth gradient.

$$\hat{y} = \sigma(w^T h + b)$$

As we saw earlier it has two parts. First part is a linear part $z = w^T h + b$ and σ which is a logistic sigmoid function $\sigma(z) = \frac{1}{1+e^{-z}}$ which we used as the activation function to convert z to a probability. Roughly speaking and without going through the hairy math, to train sigmoid output units almost always cross entropy loss function is recommended.

What if we have more than two classes? What if there is n classes? When we want to compute the probability distribution over a discrete variable with n possible values we can use softmax function. We can look at softmax as a generalization of Sigmoid function. Softmax functions are usually used as a classifier over n different classes. To make it a valid probability distribution, we need each $\hat{y}_i = p(y = i|x)$ to be between 0 and 1 and all of them sum up to 1: $\sum_i^n p(y = i|x) = 1$. As we saw earlier, the softmax function is

$$softmax(z)_i = \frac{\exp(z_i)}{\sum_j \exp(z_j)}$$

4.1 Hidden Units

There are several types of hidden units. Actually there is even more than we can find in articles. Because just those that had the best performance on the

task introduced in each paper but design process is involved with a lot of trial and error to find training network on which type on hidden units works the best for the model and it is usually not possible to predict in advance. However, choosing rectified linear units as the default choice is usually recommended. Designing hidden units is a very active area of research and there is more to explore than have been discovered by now. Since almost all of the hidden units apply an activation function to an affine linear transformation of the original input, usually different hidden units are distinguished by their activation function. So here we talk about some famous and important type of activation functions and so hidden units.

4.1.1 Recitified Linear Unit

Nowadays, after experimenting a lot of different kinds of functions and years of research on activation functions, using Rectified Linear Unit or ReLU is the default recommendation. It is a very simple activation function: $f(z) = max(0, z)$. This activation function is recommended for most feedforward neural networks (Fig. 16).

An ReLU consists of two linear pieces: first part is zero (both in value and gradient) and second part is simply the input itself. It may sounds weird the first time you think about it but applying it to the output of linear transformation yields a nonlinear transformation. Consequence of the fact that ReLUs are close to linear functions is that they share a lot of properties with them, properties that make linear models easy to optimize with gradient-based methods and also generalize well. Gradient is large and consistent whenever unit is active. Derivative of the function is zero when it is inactive and 1 when it is active. Second derivative is zero except one point so it means it is much more effective for learning rather than activation functions that

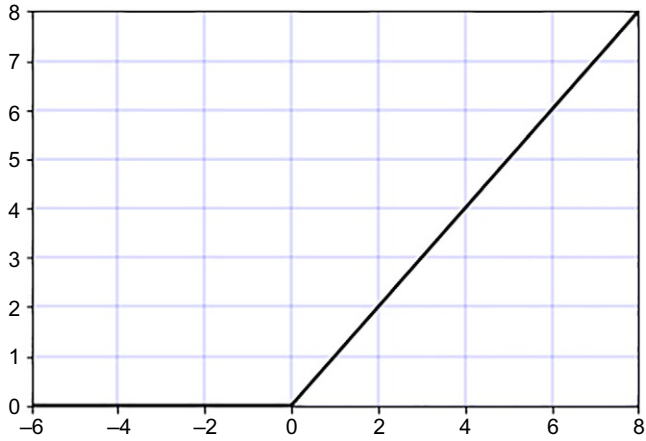

FIG. 16 ReLU activation function.

have second-order derivative effects. The only problem with ReLUs is that they cannot perform learning when the activation is zero. However, there is several approaches to generalize ReLUs to make sure gradient is available everywhere.

4.1.2 Logistic Sigmoid and Hyperbolic Tangent

Before ReLUs come around the most common activation function for hidden units was the logistic sigmoid activation function $f(z) = \sigma(z) = \frac{1}{1+e^{-z}}$ or hyperbolic tangent function $f(z) = tanh(z) = 2\sigma(2z) - 1$. As we talked earlier, sigmoid function can be used as an output unit as a binary classifier to compute the probability of $p(y = 1|x)$. A drawback on the sigmoidal units is that they get saturate (flat) when the value of z is very negative or very positive and they are very sensitive if z is around zero (Fig. 17). These activation functions are more common in recurrent neural networks, autoencoders which have some requirements that make them more suitable than ReLUs and feedforward networks despite the saturation problems they have.

One of the main questions in designing any network is how many layers and how may units in each layer we need. Usually, they are referred as depth and width of the model. Next question is how they should connect to each other. We talked about layers earlier. Usually layers arrangement is in the chain structure. So each layer's output is the next layer's input and for each layer we have

$$h^{(1)} = f^{(1)}(W^{(1)T}x + b^{(1)})$$
$$h^{(2)} = f^{(2)}(W^{(2)T}h^{(1)} + b^{(2)})$$

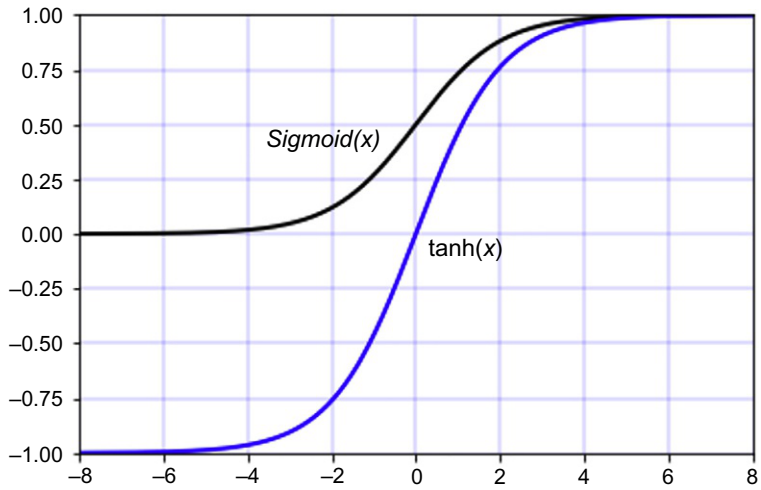

FIG. 17 tanh function vs sigmoid function.

in which $h^{(1)}$ is the first layer and $h^{(2)}$ is the second layer and so on. Deeper networks usually have fewer parameters rather than wider networks but they are harder to optimize. There is no such a rule for the number of layers and the number of units for each layer and based on the task, the ideal architecture should be found according to the experimentation.

Learning linear functions is easy because usually optimizing loss functions ends up in convex optimization when they are applied to linear functions. However, we are often interested in learning nonlinear functions. So the question is how we can come up with a family of functions that are suited for the kind of nonlinearity we are trying to learn. Beauty of DL is that it provides a framework that with enough number of hidden layers; it can approximate any type of nonlinear function. More specifically, the universal approximation theorem states that a feedforward network with a linear output layer and at least one hidden layer with any squashing activation function (like Sigmoid or tanh) can approximate any Borel measurable function from one finite-dimensional space to another with any desired nonzero amount of error, provided that the network is given enough hidden units. For the purpose of our discussion, it is enough to know any continuous function on a closed and bounded subset of $\mathbb{R}^n$ is Borel measurable and therefore may be approximated by a neural network. It is also true for any finite-dimensional discrete space and later have been proved that works for a bigger family of activation functions like ReLUs. Although, this theorem means that regardless of the function we are trying to learn, if we have a large enough network we will be able to learn it but it is not guaranteed that the training algorithm we use to be able to learn it. There are two reasons. First, the optimization algorithm we use for training our network may not be able to find the optimum value of parameters which approximate the true function. Second reason is that training algorithm may overfit to the training data and so fail to approximate the true function.

Based on the universal theorem, in a feedforward network, only one hidden layer is enough to approximate any function but in practice that layer may get infeasibly large. As stated before, deeper models reduce the total number of required units and approximation error and lead to better generalization.

4.2 Backpropagation

For all gradient-based learning algorithms, e.g., gradient descent, as name implies, we need to calculate the gradient and then use it to learn or in other words train our network. There are different ways to compute gradient. Backward Propagation, backpropagation, or even in shorter form BackProp is an approach to accomplish this task. Remember backprop is not the whole learning algorithm and it is a method to compute gradient that another algorithm like gradient descend can use to perform learning process. Advantage of backprop comparing to other methods is that it is much faster because of using an inexpensive procedure.

As we discussed, each network includes an input layer, an output layer, and some intermediate hidden layers. Each layer is connected to another one with weights. The whole training process aim to optimize these weights (and biases), hence, by feeding an input we get the right output. Backprop is a supervised method. It means we have some labeled data and each time we use our network to predict the right output we can check if we produced the right output and if not we can measure our error and modify our network based on that error and in other words learn from our mistakes.

Let us start with a very simple example to sketch a big picture and give you some intuitions. Consider the case of shooting basketball ball, obviously our goal is to put the ball in the basket, the right output. What are the features that define characteristics of a shoot? or what are our inputs? To keep it simple let us say there are three components including direction (d), angle (a), and force (f). Imagine you first shoot the ball toward basket with some initial values (d_1, a_1, f_1) and you miss (output was wrong). Now you can measure your error. First you may check the direction. For example, if the ball landed left side of the basket you may change your direction somewhat to the right. Then you see ball did not reach to the basket. Then you may increase the force or in case you pass the board you may decrease it! How about angle? You may need to increase the angle if, for example, in case you hit bellow basket so it follows a nicer curved path toward basket.

Now we adjust our inputs based on our observation of the error of first shoot and shoot the ball again with (d_1, a_1, f_1) and we repeat this process until ball goes through the basket (we get the right output). This is what we do naturally where our brain computes the errors and learn from them. By far we described a very simplified version of gradient descent. Now we need to figure out how to change the angle, or force. Basically your fingers and your arm determine these but how we should change the way we use our fingers and our arm to throw the ball? As we said before, training example provides a mean to measure the error of the output layer but not the hidden layers and now you see that you can imagine fingers and arms as units of hidden layers that you need to train (like you train your arm and fingers during exercise). Now, new questions are how much of the final error is made because of your fingers? or your arms? What if you change these? How much effect each one has on the final output? In mathematical language these are derivatives or gradients of the final cost function with respect to each component and parameters of the network. Backpropagation is a method that provides a fast way to compute these gradients.

As we saw, we have the error for the final output to find the error with respect to each layer, we start to propagate this error in backward direction from the output layer to the previous layer and then we continue to cover the whole network. This is where the name "backpropagation" comes from. Intuitively you can think about it like a feedforward process that the input is cost function and it flows backward through the network.

Afterward we adjust each parameter based on the error we propagated backward through network. We repeat this procedure until the output error is less than a predetermined threshold and then we say our network is trained and it means that now we can use it for inputs out of the training data.

Remember from calculus if you have $y = f(x)$ when you want to know how y changes with respect to changes in x, you calculate first derivative with respect to x which you may have seen it in any of these form: $\nabla_x y = \frac{\partial y}{\partial x} = y' = f'(x)$.

Last tool we need is the chain rule of calculus. Remember the chain structure we discussed earlier. Consider $z = f(g(x))$ and we need to know how z changes with respect to changes in x. Chain rule states that $\frac{\partial z}{\partial x} = \frac{\partial f}{\partial x} = \frac{\partial f}{\partial g}\frac{\partial g}{\partial x}$. Considering f and g are different layers of our network, you see how we can compute derivatives of the cost function $J(\theta)$ with respect to different parameters like weights and biases in different layers using chain rule.

Our final goal is to train the network and as we discussed, by training we mean learning weights $w_{ij}^{(l)}$ and biases $b_i^{(l)}$ for each neuron in every layer l and to learn them we need partial derivatives of cost function C with respect to weights $\frac{\partial C}{\partial w_{ij}^{(l)}}$ and with respect to biases $\frac{\partial C}{\partial b_i^{(l)}}$. To compute these, first we introduce δ_i^l which is local error signal in the layer l and neuron i:

$$\delta_i^l = \frac{\partial C}{\partial z_i^{(l)}}$$

Backprop gives us a way to find δ^l in each layer and then extract what we really need $\frac{\partial C}{\partial w_{ij}^{(l)}}$ and $\frac{\partial C}{\partial b_i^{(l)}}$ out of it. So following the chain rule for the last layer L we have

$$\delta_i^L = \frac{\partial C}{\partial z_i^{(L)}} = \frac{\partial C}{\partial a_i^{(L)}}\frac{\partial a_i^{(L)}}{\partial z_i^L} = \frac{\partial C}{\partial a_i^{(L)}} f'(z_i^L)$$

According to the fact that cost function C and activation function f are known, we can easily compute it. Rewriting it in matrix notation we have

$$\delta^L = \nabla_a C \odot f'(z^L)$$

in which $\odot$ is elementwise product which is called Hadamard product or Schur product, too. Now we need to "propagate" this error to the previous layer. It means we need to find a way to compute each local error signal δ_i^l in terms of the next layer error signal δ_i^{l+1}:

$$\delta_i^l = \frac{\partial C}{\partial z_i^{(l)}} = \sum_j \frac{\partial C}{\partial z_j^{(l+1)}}\frac{\partial z_j^{(l+1)}}{\partial z_i^{(l)}} = \sum_j \delta_j^{l+1}\frac{\partial z_j^{(l+1)}}{\partial z_i^{(l)}}$$

δ_i^{l+1} is perfect now let us elaborate on the right side:

$$z_j^{(l+1)} = \sum_i w_{ij}^{l+1} a_j^l + b_j^{l+1} = \sum_i w_{ij}^{l+1} f(z_j^l) + b_j^{l+1}$$

and differentiating it we can write

$$\frac{\partial z_j^{(l+1)}}{\partial z_i^{(l)}} = w_{ij}^{l+1} f'(z_j^l)$$

substituting it in the main equation we obtain

$$\delta_i^l = \sum_j w_{ij}^{l+1} \delta_i^{l+1} f'(z_i^l)$$

rewriting it in matrix notation we have

$$\delta^l = ((W^l)^T \delta^{l+1}) \odot f'(z^l)$$

where $(W^l)^T$ is the transpose of the weight matrix W^l. Using this equation we can compute local error signal for each layer. Now let us take $\frac{\partial C}{\partial b_i^l}$ out of it.

$$\frac{\partial C}{\partial b_i^l} = \frac{\partial C}{\partial z_i^l} \frac{\partial z_i^l}{\partial b_i^l} = \delta_i^l \frac{\partial \sum_j w_{ij}^l a_j^{l-1} + b_i^l}{\partial b_i^l} = \delta_i^l$$

It means rate of the change $\frac{\partial C}{\partial b_i^l}$ is exactly equal to the local error signal δ_i^l. Rewriting it in matrix notation we have

$$\frac{\partial C}{\partial b} = \delta$$

Final step is to compute $\frac{\partial C}{\partial w_{ij}^l}$:

$$\frac{\partial C}{\partial w_{ij}^l} = \frac{\partial C}{\partial z_i^l} \frac{\partial z_i^l}{\partial w_{ij}^l} = \delta_i^l \frac{\partial \sum_j w_{ij}^l a_j^{l-1} + b_i^l}{\partial w_{ij}^l} = \delta_i^l a_j^{l-1}$$

And done! Now we know how to compute error signal δ^L in the last layer L, how to back propagate this error through network to find local error signal δ^l for each layer l, and how to compute $\frac{\partial C}{\partial w_{ij}^l}$ and $\frac{\partial C}{\partial b_i^l}$. Hence we have everything we need to compute gradients we need to use a gradient-based learning algorithm to learn optimal weights and biases, or in other words, train our model.

5 REGULARIZATION FOR DEEP LEARNING

In this section, a brief description of regularization in the context of ML and DL is presented. One of the crucial problems in ML is to construct an

algorithm that performs well on both training data and new input data. Regularization includes strategies that designed to reduce the test error at the expense of increased training error. Developing more effective and efficient regularization strategies is one of the hot areas of research and there exists different forms of regularization available for DL practitioners. In the following, regularization is described in details focusing on those strategies for DL models. We provide extension of standard concepts of ML to the particular case of neural networks.

Regularization is defined as any modification that reduces the learning algorithm generalization error while training error is not affected. There are several regularization approaches such as putting constraints on ML model, adding restrictions on the parameter values, inserting extra terms in the objective function (a soft constraint), etc. These constraints and penalties either designed to encode specific kinds of given knowledge or to express a generic preference for a simpler model to promote generalization. Another forms of regularization called ensemble methods, combine multiple hypotheses that explain the training data.

Most of the regularization strategies employed in DL are based on regularizing estimators. Regularization of an estimator trades increased bias for reduced variance. A regularizer becomes effective when it reduces variance significantly while the bias is almost unchanged. In the context of generalization and overfitting, the goal of regularization is to take a model in which variance dominates the estimation error (rather than bias) into the model being trained.

Most applications of DL algorithms such as images, audio sequences, and text (where the true generating process involves simulating the entire universe) are in domains that true data generating process is almost outside the model family. However, a complex model family may not include true data generating process. This means that controlling the complexity of the model is not simply finding the size and the number of parameters for the model. Almost always, a large model that is regularized properly is the best fitting model for DL scenarios.

In the rest of this section, we review several strategies for creating a large, deep and regularized model.

5.1 Parameter Norm Penalties

Linear models such as linear regression and logistic regression are straightforward and effective regularization strategies which have been used prior to the advent of DL.

As used in many regularization approaches, by adding a parameter norm penalty to the objective function, the capacity of the model becomes limited. In these approaches, training algorithm minimizes both original objective function on the training data and some measures of the size of a single

parameter or a subset of the parameters. Here, we briefly discuss the effect of different norms on the model parameters as they are used as penalties. It should be noted that for the neural networks, typically we use a parameter norm penalty that penalizes only the weights of the affine transformation at each layer. Since the biases basically require less data to fit accurately in comparison with the weights, we leave the biases unregularized. The reason is that weights indicate the interaction between two variables whereas the biases control only a single variable. Moreover, regularizing the biases can result in a significant amount of underfitting.

Considering different penalties for the layers of a neural network may be useful, however, it makes the computations more expensive and using same α coefficients for all layers is still reasonable.

5.1.1 L^2 Parameter Regularization

L^2 parameter regularization (also known as ridge regression or Tikhonov regularization) is a simple and common regularization strategy. It adds a regularization term to objective function in order to derive the weights closer to the origin. Considering no bias parameter, the behavior of this type of regularization can be studied through gradient of the regularized objective function. Gradient step for updating the weights can be simply demonstrated as follows:

$$w \leftarrow (1-\epsilon\alpha)w - \epsilon\nabla_w J(w; X, y)$$

Consequently, the addition of the regularization term modifies the learning rule and decreases the weight factor on each step prior to the primary gradient update. The analysis will be then simplified by quadratic approximation of the objective function in the neighborhood of the weights with minimum unregularized training cost. Considering w^* as the minimum, the approximation of $\hat{J}$ is $\hat{J} = J(w^*) + \frac{1}{2}(w-w^*)^T H(w-w^*)$. When $\nabla_w \hat{J}(w) = H(w-w^*) = 0$, $\hat{J}$ is minimum. In order to apply weight decay gradient approach, the location of the minimum (regularized solution), $\tilde{w}$, is employed and we have $\tilde{w} = (H+\alpha I)^{-1} H w^*$. $\tilde{w}$ approaches w^* when α approaches 0. Weight decay rescales w^* along the axes that are defined by eigenvector of H. It preserves directions along which the parameters significantly reduce the objective functions. In other words, small eigenvalues of H indicates that moving along that direction is not much effective in minimizing the objective function, hence, corresponding weight vectors will be decayed as the regularization is utilized during training of the model. As illustrated in Fig. 18, the effect of regularization diminishes as λ_i increases whereas the magnitude of the components decreases as λ_i decreases.

In the context of ML, L^2 regularization helps the algorithm by distinguishing those inputs with higher variance. Accordingly, when the covariance of a feature with the target is insignificant in comparison with the added variance, its weight will be shrunk during the training process.

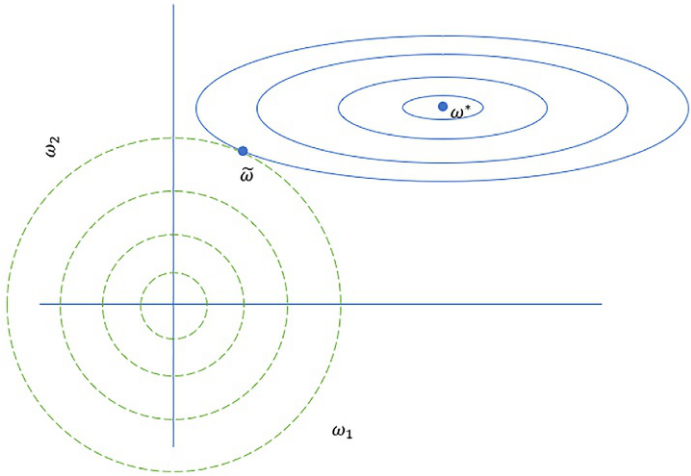

FIG. 18 The effect of L_2 regularization on the optimal value of w.

5.1.2 L^1 Regularization

L^1 regularization is defined as the sum of absolute values of the parameters. Here, we discuss the effect of this regularization and compare it with L^2 regularization. To do so, simple linear regression model with no bias is considered.

The strength of the regularization in L^1 weight decay is controlled by scaling the penalty via a hyperparameter, α. In contrast to L^2 regularization, the gradient do not scales linearly and consequently a clean algebraic solutions of the quadratic approximation of J will not be resulted. Since the model has a quadratic cost function, a truncated Taylor series can approximate the cost function. Moreover, another simplification is made by assuming that the Hessian matrix is diagonal. However, this assumption is true when all correlations between the input features are removed in preprocessing of the model.

The analytical solution of the approximate cost function leads to two possible outcomes. First, L^1 regularization pushes all w_i values becomes equal to zero when $w_i^* \leq \frac{\alpha}{H_{i,i}}$. When it gains positive values, the regularization shifts w_i in that direction equal to $\frac{\alpha}{H_{i,i}}$.

L^1 regularization provides a more sparse solution as the result of the optimal value of zero for a few parameters. In comparison to L^2 regularization, L^1 regularization causes the parameters to become sparse when α is large enough. This simplifies ML serving as a selection mechanism. L^1 regularization discards a subset of the weights by making them equal to zero.

5.1.3 Norm Penalties as Constrained Optimization

We can consider the cost function regularized by parameter norm penalty of $\alpha\Omega(\theta)$. If $\Omega(\theta)$ needs to be constrained by a constant, k, a Lagrange function can be constructed and the solution is

$$\theta^* = \underset{\theta}{argmin} \max_{\alpha,\alpha\geq 0} \mathcal{L}(\theta,\alpha).$$

In order to solve this problem, α and θ need to be modified using a procedure in which α increases when $\Omega(\theta) > k$ and decreases when $\Omega(\theta) < k$. A desirable α^* value tries to shrink $\Omega(\theta)$ while it remains greater than k.

In order to investigate the constraint impact, the problem is a function of θ when we fix α^* as follows:

$$\theta^* = \underset{\theta}{argmin}\, \mathcal{L}(\theta,\alpha^*) = \underset{\theta}{argmin} J(\theta; X, y) + \alpha^*\omega(\theta)$$

Since it is same as the regularized training problem, a parameter norm penalty can be considered as imposing a constraint on weights. Based on ω, the region that the weights are constrained can be obtained. However, we cannot determine the value of k based on the value of α^*. Instead, we can control the exact size of the constraint region by increasing/decreasing α to shrink/grow the constraint region.

One of the issues of constraints with penalties is that penalties can lead to stuck in local minima as the result of nonconvex optimization problem. In contrast, since explicit constraints and reprojection encourage the weights to leave the constraint region when they become large, they work much better than training with a penalty on the norm of the weights. In addition, explicit constraints with reprojection avoid positive feedback loop in high learning rates (which cause a rapid increase in the value of θ until a numerical overflow occurs), hence, impose some stability on the optimization procedure.

5.1.4 Dataset Augmentation and Noise Robustness

Obviously, ML models generalize better where there are more data available to train them; however, the amount of data is limited in practice. Creating fake data is one solution to increase the amount of input data and classification is easiest method. This approach is somehow straightforward for some ML tasks such as object recognition and it is not easy to apply for some other, e.g., a density estimation task. Data augmentation can be used effectively for speech recognition.

Neural networks need improvement for robustness as they are not very robust to noise. Noise injection, as a form of data augmentation, can be a part of unsupervised learning algorithms at multiple levels of abstraction (when noise is applied to hidden units). It should be noted that dataset augmentation

can dramatically reduce the generalization error. Consequently, when we are comparing two ML algorithms, we should consider the impact of the dataset augmentation.

As mentioned above, noise injection to inputs is a dataset augmentation strategy. Dropout algorithm which will be discussed later is the major development of applying noise to hidden layers which is more powerful than simply shrinking the parameters.

In the context of regularization models, noise is utilized by adding it to weights and mostly used in recurrent neural networks as a stochastic implementation of Bayesian inference over weights. Model weights can be considered uncertain and adding noise is the way to reflect it. This tries to make the parameters to move to regions where output is influenced relatively small by small perturbation of the weights (regions where the model is not significantly sensitive to small changes in weights).

5.1.5 Semisupervised and Multitask Learning

In order to predict y from x, a model can be constructed that uses both unsupervised and supervised components. There should be a trade-off between supervised criterion and unsupervised (generative) criterion. To do so, we should control how much of the unsupervised criterion is included in the overall criterion and find a trade-off better than purely generative or discriminative training criterion (see Chapelle et al., 2006 for more details).

As a way to improve generalization, multitask learning put more pressure on the parameters toward values yielding better generalization. Fig. 19 shows

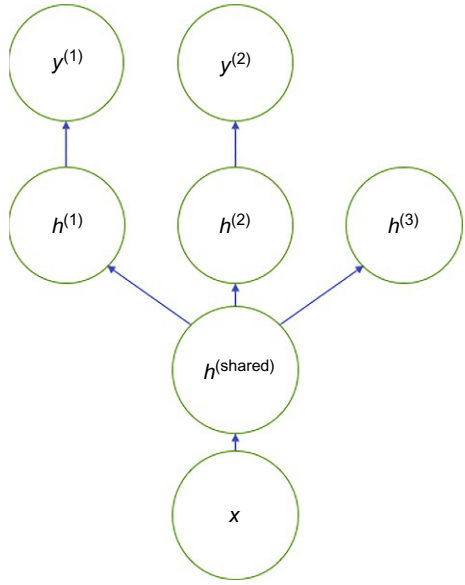

FIG. 19 A common form of multitask learning.

an illustration of a common form of multitask learning in DL frameworks where the tasks share a common input while involving different target random variables. In the figure, the model can be divided to two parts and associated parameters including task-specific parameters (upper layers) and generic parameters (lower layers). The lower layers are shared across tasks and upper layers are learned on top of those to yield a shared representation $h^{(shared)}$. The basic assumption is that there is a pool of factors explaining the variations in the input x and each task is associated to a subset of the shared factors.

Because of increased number of examples for shared generic parameters, generalization can be improved (obviously when it is valid to assume something is shared across some of the tasks).

5.1.6 Early Stopping

It is often observed that for the large models with sufficient representational capacity to overfit the task, training error decreases steadily over time while validation error starts to increase after a certain point. To deal with this issue, a popular strategy of regularization in DL can be used which is called early stopping. In this strategy, the best parameter setting yielding lowest validation error is reserved and when training is terminated (with lowest training error), the model associated to the parameter setting with minimum validation error is selected (not the latest one). Early stopping can be viewed as a hyperparameter selection algorithm where the number of training steps is another hyperparameter. One of the costs of hyperparameter automatically is periodically running the validation set evaluation throughout training. If a separate machine, CPU or GPU is available, this can be done in parallel; otherwise, we can use a validation set (small in comparison with the training set) or evaluating the validation set error less frequently. Another cost is the cost of maintaining of the best parameters setting which is negligible using a slower and large form of memory. On the other hand, early stopping has the advantage that it requires no change in the underlying training procedure, the objective function or the set of allowable parameter values while in weight decay, too much weight decay cause the network be trapped in a local minima. For more details regarding the mechanism of early stopping, see Goodfellow et al. (2016).

5.1.7 Parameter Tying and Parameter Sharing

In the previous regularization methods, adding constraints or penalties are done with respect to a fixed region or point. However, when we have knowledge about what values a parameter should take such as dependency between some parameters (knowing that their values should be close to each other), we need other ways of dictating this knowledge to the model. For instance, we can define a parameter norm penalty for the distance between two parameters when it is required of the parameters to have close values. The more popular way is *parameter sharing* utilizing constraints that force sets of parameters to be equal. The advantage of parameter sharing over parameter norm penalty

is that we only need to store a unique set of parameters in memory which leads to significant saving of the memory footprint of the model in certain models. In particular, parameter sharing is the most popular regularization for convolutional neural networks applied to computer vision. It allows these networks to incorporate domain knowledge into the network architecture and reduce the number of unique model parameters. Moreover, without an increase in training data, it makes it possible to drastically increase network sizes.

5.2 Sparse Representation

As opposed to weight decay that places penalty directly on the model parameters, a complicated penalty strategy can be applied on the activation of the units encouraging their activation to be sparse. In this strategy, many of the elements of the representation are zero. Same sorts of mechanisms used for parameter regularization are applicable to representational regularization. A norm penalty on the representation is added to J loss function denoted as $\omega(h)$ and the regularized loss function is as follows:

$$J(\theta; X, y) = J(\theta; X, y) + \alpha\omega(h)$$

where α is a coefficient indicating the relative contribution of the norm penalty term (larger values lead to more regularization).

In addition to norm penalty on the representation, a hard constraint on the activation values can be used to obtain representational sparsity. Sparsity regularization can be used in a variety of contexts and any model with hidden units can be made sparse.

5.2.1 Bagging and Other Ensemble Methods

The idea of bootstrap aggregating or bagging is rooted in a general strategy in ML called model averaging (ensemble methods). In this strategy, several different models are trained separately and each model votes on the output for test examples. Usually, different models do not make all the same errors on the test set. That is why ensemble methods are useful. There are different ensemble methods. It is possible to train models which are completely different in algorithm and objective function or a same kind of model can be reused several times (bagging). In bagging, different datasets are constructed by sampling from the original dataset, thus, each dataset is missing some examples from the original dataset and also each dataset has duplicate examples. These differences in datasets lead to different trained models.

In addition, due to random initialization, random selection of minibatches, etc., neural networks reach to different solution points even if all models are trained on the same dataset. Consequently, they can benefit from model averaging.

The cost of model averaging is increased computation and memory. Therefore, it is discouraged when benchmarking algorithms for scientific papers are used.

5.2.2 Dropout

Dropout is a powerful yet computationally inexpensive regularization method. It does not significantly limit the type of the model or training procedure and works with almost any type of model that uses a distributed representation such as feedforward neural networks, probabilistic models, and recurrent neural networks. When a neural network is large, bagging becomes impractical due to the high cost of training and evaluating these networks in terms of runtime and memory. In these circumstances, dropout provides a bagged ensemble of exponentially many neural networks with an inexpensive approximation to training and evaluation. As shown in Fig. 20, subnetworks are constructed by removing nonoutput units from the base network and dropout trains the ensemble. In order to remove a unit from the network, the output value of the unit is multiplied by zero. However, this needs some modifications for

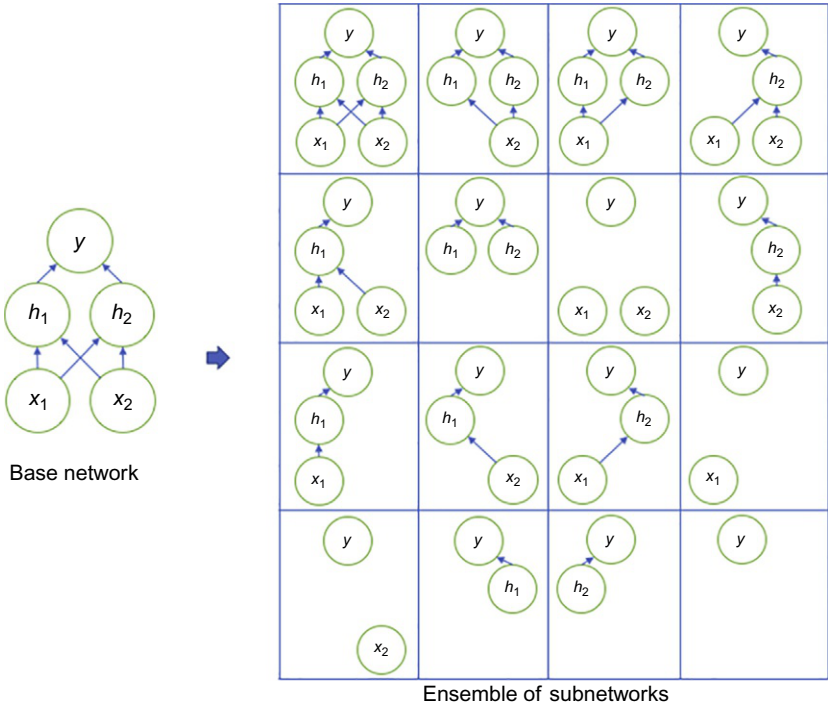

FIG. 20 An ensemble consisting of subnetworks from base network that dropout trains.

some models. While the simplest way is multiplication by zero, there are other operations for removing a unit from the network.

As described before, k different models are defined to train model on k constructed different datasets from training set. The aim of dropout is approximating this process for exponentially large number of neural networks.

To train with dropout minibatch-based learning algorithm is employed. In this algorithm, a different binary mask is sampled independently randomly to be applied to all of the input and hidden units in the network. Prior to training the model, the probability of sampling a mask value of one is fixed (as a hyperparameter). The probability of inclusion of an input unit and a hidden unit is typically 0.8 and 0.5, respectively. Then, as shown in Fig. 21, we run forward propagation followed by backpropagation and the learning update as usual.

The difference between dropout and bagging is that in bagging, the models are all independent while in dropout models share parameters which makes it possible to represent and exponential number of models with a tractable amount of memory. Another difference is that in bagging each model is trained to convergence. In contrast, in the case of dropout, a small portion of the possible subnetwork is trained for a single step and the remaining subnetworks arrive at good setting of the parameters as the result of parameter sharing. These are the only differences of the dropout and bagging.

In order to improve dropout, fast dropout (Wang and Manning, 2013) and dropout boosting (Warde-Farley et al., 2013) are proposed in the literature. It also has ispired other stochastic approaches such as DropConnect (Wan et al., 2013).

Dropout goes further than being described as a means of performing approximate bagging. It also trains an ensemble of models that share hidden units where each hidden unit must perform well regardless of which other hidden units are in the model and must be able to be swapped and interchanged between models. Consequently, dropout regularizes each hidden unit to be a feature that works well in different contexts. As important aspects of

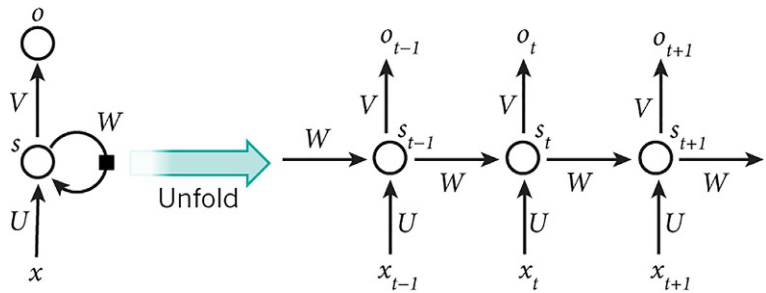

FIG. 21 Unfolding the self-loop in recurrent neural networks.

dropout, it should be noted that masking noise is applied to the hidden units where the power of dropout arises from as well as the fact that noise is multiplicative which does not allow a pathological solution to the noise robustness problem.

In this chapter, most of the regularization approaches and strategies (as a central theme of ML) for neural networks has presented and explained.

6 SEQUENCE MODELING (LANGUAGE MODELING)

Computers languages or programming languages are designed to be efficient, unambiguous, and easy to parse. Natural languages, in comparison with programming languages, are ambiguous and defy formal description. In several tasks in NLP, like machine translation, computer needs to read a sentence in one human natural language like German as the input and output translation of it as an equal sentence in another one like French. Language models define a probability for a sequence of words, character, or bytes. In machine translation this probability is used for word ordering and word choice problems. For example, in different languages ordering of words can be different. In some languages adjectives come before the noun where in some others they come after the noun. So comparing the probability of two sequences with the same words but different ordering shows which one is more likely to happen.

$$P(I\ wish\ I\ could\ fly) > P(I\ fly\ could\ wish\ I)$$

In every language we have different words for one concept. Thus, while translating from one language to another, for each concept we need to choose between different words. Again, we can compare probability of the sequence with different words and choose the one with the highest probability.

$$P(taking\ a\ picture) > P(taking\ a\ photograph) > P(taking\ an\ image)$$

Many approaches consider natural language as a sequence of words instead of individual characters or bytes. These models are called word-based language models. Since a number of words are so large language, models should work in a very high dimensional sparse discrete space. Here we introduce strategies to deal with such a space efficient.

6.1 Count-Based Models or n-Grams

An n-gram is a sequence of n tokens (in word-based models tokens are words). This approach compute a conditional probability of the n-th token given the previous $n - 1$ tokens. This model defines these probability products of the conditional probability of all the smaller sequence of words. Computing the perfect probability for each word based on all the words before

it in the sequence is difficult because there is a lot of them. To solve this problem we just consider a window of n words before each word which is mentioned in Section 2.

$$P(w_1,\ldots,W_m)=\prod_{i=1}^{m}P(w_i|w_1,\ldots,w_{i-1})$$

To compute this probabilities these models simply count the number of unigrams (1-gram), bigrams (2-grams), and so on conditioning on the previous words.

$$p(w_2|w_1)=\frac{count(w_1,w_2)}{count(w_1)}\quad p(w_3|w_1,w_2)=\frac{count(w_1,w_2,w_3)}{count(w_1,w_2)}$$

There are some problems for computing this probability. For example, if there is a probability which is equal to zero, then result of the all product is zero. Or if it is in the denominator, then it is undefined. For example, in test time if the model confronts an input that was not in the training set, then the probability will be equal to zero. To solve this problem n-gram models in general use *smoothing* methods. Smoothing techniques shift probability mass of the observed sequence to the similar unobserved ones. Chen and Goodman (1996) compare different techniques from an empirical viewpoint. Performance improves as we count higher order n-grams but as we go to upper-orders, number of n-grams quickly decreases. For example, the number of specific 5-grams occurring in the training dataset is not as much as a 3-grams or 4-grams that is extracted from it. An approach to deal with this problem is called *Back-off methods*. These methods search for lower-order n-grams if the frequency of the context is too small.

One of the disadvantages of n-gram models is that there is a lot of n-grams! Actually if we have $|V|$ words in our vocabulary, there will be $|V|^n$ n-grams, so-called curse of dimensionality problem. Even with a huge training dataset and low values of n, most of n-grams will not occur. From computational view as we go to higher order n-grams gigantic amount of RAM is required. Heafield et al. (2013) used 140 GB RAM for 2.8 days to build an unpruned model on 126 billion tokens.

There is another fundamental problem in count-based models and it is the distance between words. In these models any pair of words have the same distance from each other as from any other word in one-hot vector space. So it is not possible to share knowledge between one word and other semantically similar words. To overcome this problem and improve the efficiency of these models, class-based language models introduce word categories and improve the statistical relation between the words that are in the same category. This approach uses clustering to put words in different groups or classes based on their cooccurrence frequencies with other words. Then, the model can use classes IDs instead of individual words.

6.2 Recurrent Neural Networks Language Models

When you watch a movie, read a book, or even now reading this very sentence, you do not start to understand each word without its relation to other words and its position in the sentence or even in the higher level in the story line. It means words are not completely independent of each other to create the meaning of the sentence as J.R. Firth says "You shall know a word by the company it keeps." It means that you figure out the meaning of each word based on your understanding of the precedent words. Each word can have different meaning based on the context or story line and other words are what make this story line. You start to read each sentence and each word with having all the previous information you have in your mind, not starting from the scratch, and you will edit that information based on the new data you receive. You see there is a persistency in your understanding as you continue to read. Traditional neural networks are unable to keep this persistency, they cannot persistently use reasoning based on the previous information to, for example, predict what may happen next. In traditional neural networks, we assume all inputs are independent and this is a bad assumption when we need relationships! For instance, when you want to predict the next word in the sentence you need to know previous words and their relation to the word you want to predict. Recurrent neural networks (RNN) are introduced to solve this issue. The difference between traditional neural networks and recurrent neural networks is that in traditional networks, loops, or feed backs are not allowed. In other words, cycles are not allowed in our computational graph in traditional networks, but they are allowed and actually they are the main feature of recurrent neural networks. These feedback or loops provide the persistency we need. RNN architecture is specialized to process sequential data like a sentence, or in general $x_1, x_2, \ldots, x_t$. The term "recurrent" comes from the idea that these network perform the same process on every element of the sequence but in each step is also dependent to the previous step.

In other words, value of the hidden layer in time t which is $h^{(t)}$ (also called status of the network) is a function of the $x^{(t)}$ and $h^{(t-1)}$.

$$h^{(t)} = f(x^{(t)}, h^{(t-1)})$$

The intuition behind it is that hidden layer plays the role of the memory of the network which captures information that network has seen so far. In theory, this memory should remember what happened long time ago but in practice it is limited and actually it just can look back just a few steps. We will go through details of this limitation later. Roughly speaking, the memory capacity is fixed and limited, because networks tries to map an arbitrary length sequence of inputs $x^{(1)}, x^{(2)}, \ldots, x^{(t)}$ to a fixed length vector $h^{(t)}$, it cannot capture everything. As it goes forward, to be able to remember new information, it loses some of the past information and that is why it is referred as lossy memory. Notice that new information does not completely replace old

information. In other words, this memory is not like bits of the RAM memory, it is a high dimensional space, and network can selectively keep some part of the past information more precisely. The output layer reads the information from the hidden layer and interprets it as the output or the prediction of the network at the step t which is $o(t)$.

By unfolding or unrolling, we just simply expand the loop for each time step. After unfolding the network we end up with a feedforward network whose number of layers is equal to the inputs. Now we talk about components of the network.

- x_t is the input at time step t. It can be, for example, each word of the sentence.
- h_t is the state of the hidden layer at time step t. This is the lossy memory of the network. As you see in Fig. 21 there is no particular activation function used to apply the nonlinearity in hidden layers. Here we choose hyperbolic tangent activation function. Notice that in order to compute h_1, the first hidden layer state, we need h_0. Usually we initialize it to all zeroes.
- o_t is the output of the network at time step t. It is just from the memory of the network at time step t. Form of the output and loss function, like the activation function, is designer's choice. In the task of predicting the next word in the sentence, it can be the unnormalized log probabilities across our vocabulary. Then as a natural way, we can apply *softmax* function to get $\hat{y}_t$.
- b and c are the bias vectors and U, V, and W are weight matrices to map data from input to hidden unit, hidden to output, and hidden to hidden units, respectively, which are parameters of the model.

Now we can formulate our network as follows:

$$z_t = Wh_{t-1} + Ux_t + b$$
$$h_t = tanh(z_t)$$
$$o_t = Vh_t + C$$
$$\hat{y}_t = softmax(o_t)$$

As you may noticed an RNN uses the same parameters, like weight matrices, in every step. It is basically because of the fact that with unfolding the network we actually expanded a "loop" and that is the nature of the loop to do the same thing in each time step. Sharing parameters across every time step vastly reduces the number of model parameters in total and it means we have less parameters to learn. But everything comes in a price and it means if we need to just learn a few parameters instead of a lot of parameters you can guess that it will not be easy. Actually difficulty of learning the shared parameters is the price RNNs pay to reduce the number of parameters. To give you an intuition, if you consider the unrolled graph, a typical feedforward

graph to map output of each layer to the next layer we need a weight matrix (I am ignoring the bias term for the sake of simplicity). If you imagine each weight matrix as a key that opens the gate to the next layer to go through whole network, the number of keys we need is equal to number of layers we have in network. Learning a shared W is like making a super key which opens every gate and, as you may guess, it is not easy or cheap. Fig. 21 shows a typical RNN which maps an input sequence to an output sequence of the same length but in general depending to the task we may need just one output, like when we are doing the sentiment analysis that the only thing we need is the final output, or we may need a different length output, like when we translate a sentence to another language. We will talk more about these types of networks.

To compute the loss function for mapping an input sequence x paired with a set of corresponding y values, we sum all of the loss values in each time step. Training RNNs is like training traditional neural networks with some differences. To compute gradients of the network we need to do a forward propagation followed by a backward propagation, which is an expensive operation. Since every step is dependent to the previous one, it makes the whole process sequential. Hence, we cannot use parallelization techniques to reduce the runtime. To train RNNs, we apply backprop to the unfolded graph and this is called backpropagation through time or BPTT. For the sake of generalization, we do not define any specific loss function or activation function.

$$L(y,\hat{y}) = \sum_t L_t(y_t,\hat{y}_t)$$

$$z_t = Wh_{t-1} + Ux_t + b$$

$$h_t = \sigma(z_t)$$

$$\hat{y}_t = Vh_t + C$$

To stay consistent with the literature and to keep it simple we removed the *softmax* operation which is a postprocess for output and now we use $\hat{y}_t$ instead of o_t as output of the model in time step t. Please notice that σ is not sigmoid activation function and can be any activation function. Based on the above equation we have

$$\frac{\partial L}{\partial L_t} = 1 \tag{1}$$

Now we want to compute derivative of the loss function with respect to the output in each time step t

$$\frac{\partial L}{\partial \hat{y}_t} = \frac{\partial L}{\partial L_t}\frac{\partial L_t}{\partial \hat{y}_t} = \frac{\partial L_t}{\partial \hat{y}_t}$$

since from Eq. (1), we know $\frac{\partial L}{\partial L_t}$ is equal to one. Loss function derivative with respect to h_T, which is so-called local error signal for the last hidden layer or δ_T is given by:

$$\delta_T = \frac{\partial L}{\partial h_T} = \frac{\partial L}{\partial L_T}\frac{\partial L_T}{\partial \hat{y}_T}\frac{\partial \hat{y}_T}{\partial h_T} = \frac{\partial L_T}{\partial \hat{y}_T} \odot V \tag{2}$$

the term $\frac{\partial L_T}{\partial \hat{y}_T}$ depends on the choice of loss function. Now that we have the derivative for the last hidden unit as we did for usual backprop we need to calculate the local error signal for each previous layer δ_t:

$$\delta_t = \frac{\partial L}{\partial h_t} = \frac{\partial L}{\partial \hat{y}_t}\frac{\partial \hat{y}_t}{\partial h_t} + \frac{\partial L}{\partial h_{t+1}}\frac{\partial h_{t+1}}{\partial h_t} = \frac{\partial L}{\partial \hat{y}_t} \odot V + \delta_{t+1} \odot \sigma'(z_t)W \tag{3}$$

Note that because h_t has two descendents $\hat{y}_t$ and h_{t+1} we have to sum up the effect from both directions. Now we can calculate gradients for W, U, V. We sum up all the gradients over all time steps:

$$\frac{\partial L}{\partial W} = \sum_t \frac{\partial L_t}{\partial W} = \sum_t \frac{\partial L_t}{\partial h_t}\frac{\partial h_t}{\partial W} = \delta_t \odot \sigma'(z_t)h_{t-1} \tag{4}$$

We know $h_t = \sigma(Wh_{t-1} + Ux_t + b)$ depends on h_{t-1} which in turn depends on h_{t-2} itself and so forth. It means that in order to find out the gradient in time step t, we need to backpropagate through the networks to $t = 1$. In the same way for U we have

$$\frac{\partial L}{\partial U} = \sum_t \frac{\partial L}{\partial h_t}\frac{\partial h_t}{\partial U} = \delta_t \odot \sigma'(z_t)x_t \tag{5}$$

Calculations for V are even simpler:

$$\frac{\partial L}{\partial V} = \sum_t \frac{\partial L_t}{\partial \hat{y}_t}\frac{\partial \hat{y}_t}{\partial V} = \sum_t \frac{\partial L_t}{\partial \hat{y}_t} \odot h_t \tag{6}$$

The point is that unlike W and U, to calculate $\frac{\partial L_t}{\partial V}$ all we need is values at the current time step and it is not dependent to the previous time steps. For biases we have

$$\frac{\partial L}{\partial c} = \sum_t \frac{\partial L_t}{\partial \hat{y}_t}\frac{\partial \hat{y}_t}{\partial c} = \sum_t \frac{\partial L_t}{\partial \hat{y}_t} \tag{7}$$

$$\frac{\partial L}{\partial b} = \sum_t \frac{\partial L}{\partial h_t}\frac{\partial h_t}{\partial b} = \sum_t \frac{\partial L}{\partial h_t} \tag{8}$$

Notice here, for the sake of simplicity, we defined $\delta_t = \frac{\partial L}{\partial h_t}$ for local error signal instead of $\delta_t = \frac{\partial L}{\partial z_t}$ similar to what we introduced in backprop. However, if you want to go with the same δ, you can use the chain rule like: $\frac{\partial L}{\partial z_t} = \frac{\partial L}{\partial h_t}\frac{\partial h_t}{\partial z_t}$

where $\frac{\partial h_t}{\partial z_t} = \sigma'(z_t)$. As you see, BPTT is like more of a fancy name for the standard backprop which is applied on the unfolded recurrent network.

In NLP tasks in addition to words we need to have a way to define start and end of the sentence, or in another words length of the sequence. There are different ways to handle this issue. Here, we briefly introduce two of them. One of the ways is to add a symbol for the start and another for end of the sentence to the vocabulary and also add them to start and end of our training samples. Whenever model is generating a sentence and the end symbol comes up, model stops the generating process. Another solution is to let the model itself decide to continue or stop generation. For example, trains a sigmoid function in output unit to decide to continue or end in each time step.

In the graphical illustration of the network, each edge shows a direct relationship between two variables and so it means there is a parameter to learn. Starting with a fully connected network (each time step is connected to every time step in the past), different models tried to remove some edges that are not carrying a strong interaction and as a consequence, reduce the number of parameters to learn and hence making model easier to train and make the computationally more efficient. As we said, in theory, RNNs are able to have access to information from long time ago but in practice it is able to catch information from a few steps ago. Based on these two reasons it sounds reasonable to make the Markov assumption and assume in each time step we need information from k steps ago rather than entire past time steps. Actually, in a lot of task we do not need the entire history. However, there are tasks in which we need the entire past since we believe each past time steps has its own effect on the next output.

Imagine solving a "fill the blank" question which the blank part is in the middle of the sentence. Of course you do not read part of the sentence which comes before the blank and leave the part after the blank. What if the sentence starts with the blank? This is an example of tasks in which we need to have access to entire sequence, it means past, present, and future inputs, to generate the output in each time step. Same issue comes up in speech recognition, hand writing recognition, and many other sequence-to-sequence tasks. By now we just discussed typical or so-called vanilla RNN which just have access the past and present input. This issue is addressed in next sections.

6.3 Bidirectional Neural Networks

The idea behind bidirectional networks is simple and as name implies, to predict the missing part and have access to the future inputs in addition to the past inputs, we run an additional RNN on top of the previous one which moves backward through time starting from the end of the sequence toward beginning. Output will be computed based on the hidden state of both RNNs (Fig. 22).

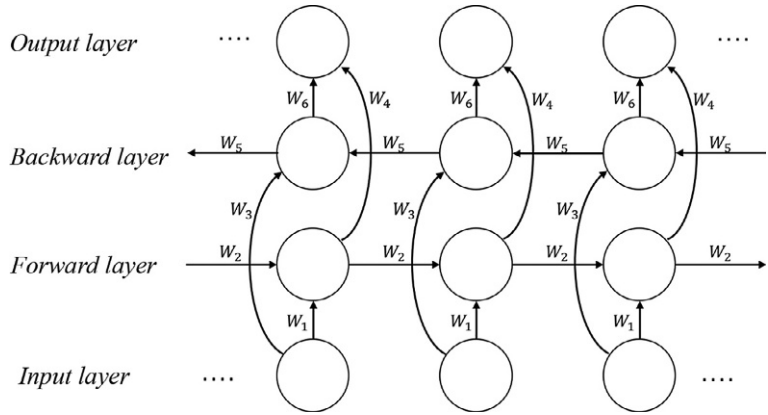

FIG. 22 Bidirectional RNN.

Looking at the two layers of hidden states on top of each other, or in another words, two RNNs stacked on top of each other, maybe the first thing that comes to your mind is that what if we add more layers. After all, that is the theme is DL. Actually we can extend it to two-dimensional space with adding two more layers of hidden states. So for each direction including up, down, left, and right, we have one RNN. As a two-dimensional space imagine an image where in each pixel we can compute the output based on the pixels in four directions around that pixel which mostly captures local information as well as it can have access to far away inputs.

6.4 Vanishing and Exploding Gradient Problem

The vanishing gradient problem was originally discovered by Hochreiter in 1991 (Hochreiter and Schmidhuber, 1997) and since then attracted a lot of attention. It turns out as we go deeper and deeper in layers, one of the biggest issues any training algorithm needs to deal with is to be able to remember dependencies from previous layers. Consider the most simple example where in a recurrent neural network which repeatedly multiply the same number w to the new input, it is clear that after t steps we have multiplied w, t times, or in other words w^t. Now when you want to train your network, your gradient value propagated over t steps is scaled with w^t. Obviously if the absolute value of w is greater than 1 your gradient value will explode which makes learning process unstable and if it is less than one it will vanish which makes it difficult to move toward the right direction to improve the performance of the network. Now in matrix notation, to prevent this problem we need the absolute value of the eigenvalue of the matrix W to be close to 1. This issue arise especially in the case of recurrent neural networks which use the same matrix W at each time step. In the case of feedforward neural networks this is not a big issue and can be avoided even in very deep networks (Sussillo, 2014).

Usually exploding gradients problem is not as difficult to deal with as vanishing gradients problem. We can simply limit the gradient value to avoid exploding gradient problem. But even if we assume our network parameters are set to avoid exploding gradients and so can store memories, we still have to deal with the problem of long-term dependencies which vanishes through multiplication of a lot of Jacobians.

Maybe the first thought comes to mind is that we can make the parameters to stay in the safe area and prevent vanishing or exploding problem but to be able to store long-term memories in a stable way, we have to enter to the area which makes parameters vanish. Performing experiment (Bengio et al., 1994) explored the trade-off between efficient learning by gradient descent and latching on information for long periods and specifically as the time span of dependencies increases the probability of successful training of a recurrent neural network using stochastic gradient descent learning algorithm approaches zero even when the time span length is 10 or 20. They showed gradient contribution as you go far from the current state shrink exponentially and goes toward zero very fast. In other words, error signal approaches zero and consequently the error signals propagated for further steps become zero. It means we lose long-term dependencies. In practice in a lot of cases backpropagation is truncated to a few steps. To explore the mathematical side of the vanishing and exploding gradients problem in more detail, you can read Bengio et al. (1994) and Pascanu et al. (2013).

There are different strategies to avoid these problems. One way is to design two parts in the model. To give you some intuition, you can think about clock hands which work in multiple time scales. One part of the model takes care of short-term dependencies (fine-grained, like the minute hand of the clock) and other one handles long-term dependencies (coarse-grained, like hour hand). In this way, model operates at multiple time scale. First part deals with details when second part takes care of transferring information from long time ago. To implement this strategy, adding skip connections across time, leaky units, and removal of some of the connections have been used. Adding skip connection was first introduced by Lin et al. (1996) which added some direct connections from d time step ago to mitigate the vanishing problem. Adding skip connections gradients shrinks exponentially as a function of τd rather than τ.

Leaky units are hidden units with linear self-connections. Using this strategy to compute hidden unit state at each time step we have

$$h_t = \alpha h_{t-1} + (1-\alpha)\tanh(Vx_t)$$

depending on the value of α information about the past can remembered for a long time (when α is close to one) and it can be discarded very fast (when α is close to zero). Using leaky unites is a more flexible and smooth way of accessing to the information from the past rather than skip connections because of the flexibility of choosing α rather than an integer d

(El Hihi and Bengio, 1996; Mozer, 1992). Also we can either manually fix α or learn them as a model parameter (Pascanu et al., 2013).

"Removing connections" idea to handle the long-term dependencies is to remove the length-one connections and replace them with longer connections to force unites to operate on a long time scale. Using skip connection we are adding connections to the network but there is no guarantee that units receiving these connections will use information from the skip connection and they may choose short-term connections over long-term connections but using this method they are forced to do it. There are two major approaches to do it. One is to make different groups of leaky unites to work on different time scales (Mozer, 1992; Pascanu et al., 2013), and the other one is to have discrete updates at different frequency (El Hihi and Bengio, 1996; Koutnik et al., 2014).

Furthermore, proper initialization and using regularization help vanishing problem. Also, ReLU activation function because of their derivation is less vulnerable to vanishing problem comparing to sigmoid or tanh.

6.5 The Long Short-Term Memory and Gated RNNs

After a few years Hochreiter introduced the problem of long-term dependencies in recurrent neural networks in 1991. In another paper in 1997 (Hochreiter and Schmidhuber, 1997) he introduced a new method called long short-term memory (LSTM) to solve this problem. LSTMs are specifically designed to be able to remember long-term dependencies. As of today LSTMs are the most effective strategy in sequence modeling. To keep long-term dependencies we need to carry some information along but we may not need to carry all of the information. Using leaky unites, we choose a constant α to determine how much of the past information we want to pass to the next state. However, we may need to be more flexible about α. We may want to pass different amount of information at each state based on the situation. In other words, we can generalize what leaky units do in a customized way for each step and that is what LSTMs or in general gated recurrent neural networks do. Gated recurrent neural networks use a mechanism to be able to decide to keep the memory or clear it at each step. The core idea behind LSTM is to make a path through the self-loop where gradient can flow for a long duration. LSTM has been used for different tasks like handwriting recognition and generation, speech recognition, machine translation, image captioning, parsing, etc.

LSTM like a usual recurrent neural network has a self-loop, the difference between LSTM and usual networks is the inside structure. In ordinary recurrent networks, we apply a nonlinearity function on the affine transformation of the input but LSTM has LSTM cells which has a internal self-loop in addition to the outer self-loop. LSTM has a mechanism consisting gating units to control how to manage the flow of information.

The most important component of the LSTM is the cell state $s^{(t)}$ which conveys the information along the entire chain with some minor linear interactions. Cell state is similar to leaky units. In leaky units, we have a parameter α which controls how much of the past memory is going to be carried along. But in LSTM we have a gate to determine it which is called forget gate defined as follow:

$$f_i^{(t)} = \sigma(b_i^f + \sum_j U_{i,j}^f x_j^{(t)} + W_{i,j}^f h_j^{(t-1)})$$

where U is input weight and W is the recurrent weight for the forget gate. This gate looks at the previous hidden layer vector $h^{(t-1)}$, and the current input $x^{(t)}$ and give back numbers between 0 and 1 for each number in the cell state. Zero means nothing and should go through all, i.e., "completely get rid of it," and 1 means everything should go through, i.e., "completely keep this." For example, in the case of next word predicting when network encounter a new subject based on the language, next words and verbs may change based on the gender and plurality of the subject so forget gate may decide to get rid of the information about the previous subject.

Now we need to decide which information needs to be added. LSTM do this using two components. First, a sigmoid layer called input gate choose candidates that needs to be added. Then the second part is using a tanh function that creates a vector of candidate values which can be added to the cell state. Combining these two, we are ready to add new information to cell state.

$$I^{(t)} = \sigma(b_i + \sum_j U_{i,j} x_j^{(t)} + W_{i,j} h_j^{(t-1)})$$

$$C^{(t)} = \tanh(b_i^C + \sum_j U_{i,j}^C x_j^{(t)} + W_{i,j}^C h_j^{(t-1)})$$

So cell state will update as follows:

$$s^{(t)} = f^{(t)} s^{(t-1)} + I^{(t)} C^{(t)}$$

The only part left is output of the cell. LSTM cell output is based on the cell state but we need to decide which parts we want to output. So we use a sigmoid layer to decide about parts we want to output, which is called output gate, and then we use a tanh to squeeze values then we multiply these two so we have the parts that we want to output with their values.

$$o^{(t)} = \sigma(b_i^o + \sum_j U_{i,j}^o x_j^{(t)} + W_{i,j}^o h_j^{(t-1)})$$

$$h^{(t)} = o^{(t)} \tanh(s^{(t)})$$

There are a lot of different variants of LSTM. One of them is adding peephole connections which mean letting gates to be able to have access to the cell

state. In other words, gates can look at the cell before they decide about forgetting, adding, or outputting information. Some versions just have peepholes for some of the gates and not others.

Another example of gated networks is gated recurrent unit (GRU) (Cho et al., 2014). GRU combines forget and input gated. It also Merges hidden and cell state. Roughly speaking GRU is a simplified version of LSTM but because training them is not as expensive as LSTM, GRU has been growing increasingly popular. There are a lot of other variants. Greff et al. (2017) and Jozefowicz et al. (2015) have done a great comparison between popular variants finding that there is not a clear winner which beats all LSTM and GRU.

6.6 Encoder-Decoder Sequence-to-Sequence Architectures

Architectures we discussed so far have equal size of inputs and output sequences but what if the desired output is longer or shorter than input? For example, when you want to translate a sentence, the translated version does not have necessarily the same length as the original version, or in question answering task most of the times there is not a relationship between length of the input and output. The answer of a very long question can be only one word. We have the same situation in speech recognition although in this case length of the input and output would be related but they are not equal.

Introducing architectures for mapping a variable length input to a variable length output started with Cho et al. (2014) work and followed by Sutskever et al. (2014) who came up with the architecture independently and managed to make state-of-the-art translation using this approach. They follow different approaches but the idea behind the whole architecture is simple. First a recurrent network reads the input and output a representation of the input as a vector. This part of the system is called reader, or encoder, or input RNN. Then second RNN, which is called decoder, or output RNN, or writer take output of the encoder as input and generates the output sequence. The most important part to notice in this architecture is that the length of the input can be different from length of the output. To train this model both RNNs are trained jointly to maximize the average of $\log\left(y^{(1)},y^{(2)},\ldots,y^{(n_y)}|x^{(1)},x^{(2)},\ldots,x^{(n_x)}\right)$ over all pairs of x and y sequence in the training set. The major limitation of this model reveals when the input is long and since the encoder's output size is fixed it may be too small to be able to represent a long sequence properly. Bahdanau et al. (2014) introduced this problem and proposed a variable length sequence instead of a fixed size vector for the encoder output. Moreover, they introduced an attention mechanism that learns how to associate the elements in the encoder's output to the decoder's output.

6.7 Recursive Neural Networks

We talked about word vector space models in which similar words are clustered together which means the vector of Italy is close to the vector of France. But words come with the company of other words. Since we can describe a concept in different ways using different words, now the question is what about phrases or sentences which have the same meaning despite having different words? How can we represent the meaning of longer phrases? The short basic answer is to map them in the same word vector space. In the previous sections, different approaches like word2vec or Glove were introduced to map single words. The problem with those approaches is that they cannot capture representation of longer phrases. On the other side of the spectrum, there are other approaches for documents vector which can map phrases and they are good for information retrieval and document exploration but the problem with these approaches is that they ignore the word order and as a consequence we cannot get to understand the details of the representation or our understanding. So now the question is how we can capture the meaning and semantics and also syntactic structure of a sentence without ignoring word ordering.

One of the major approaches is using principles of compositionality. To do so, we use the meaning of the words and rules to combine them. In recurrent networks, we were going from left to right to read sentences but to use the rules to combine words, we need to parse sentences first which means to determine role of each word and the way each word as a noun, verb, adjective, etc., combines to make bigger components of a sentence and at last the way these chunks combine to make the sentence itself. If we draw a diagram of parsing, a sentence is like an upside down tree in which leaves are words and as we go up based on the parsing rules each joint shows the role of each combination of the words until we reach to the root on top which is the sentence.

If you remember the diagrams from recursive algorithms, there is quite a bit of similarity and so is the name. Recursive neural networks (RNN) unfortunately have the same abbreviation as Recurrent Neural Networks so be careful about that. But in general, we can say recursive networks are the generalization of recurrent networks because recurrent networks make chain which is a special type of tree that just have connections to the right joint but in recursive networks we do not have that obligation. The idea behind recursive networks is to construct a model which is able to jointly learn parse trees and also compositional vector representation. It is exactly what we are looking for, i.e., capturing syntactic and semantic information of a sentence.

There is a big history of papers behind the nature of the languages as the cognitive facts about understanding them which is beyond the scope of this book but for our purposes, we can mention advantages of recursive language modeling. First, it is very helpful in disambiguation. There are different

meanings for each words and actually sometimes based on the meaning, they may change the way a sentence should parse. Consider the sentences "I eat pizza with a fork" and "I eat pizza with ketchup." In the first one "a fork" is part of a noun phrase which combines with "with" to make a phrase preposition which is part of the verb phrase including three components of "eat," "pizza," and "with a fork." But in the second sentence "pizza with ketchup" is a noun phrase which makes a verb phrase combining with "eat." Recursive networks are also very helpful to the task of reference. To clarify what we mean by task of reference, consider the sentence "I am going to my brother's house, he paid for the ticket to there… " Who is "he"? Where is "there"? Recursive models are useful in labeling, too. Another advantage of RNN is that for a sequence of length t, depth of the of the network can be reduced from t to $O(\log(t))$ which because of the reducing the depth can help with long-term dependencies. RNN have been also successful in computer vision (Socher et al., 2011).

So in each step, we have two children nodes of the tree in which each one is a representation vector as input and need the output to be a same dimensional representation which captures the semantic representation and a score which shows how plausible the parent node would be. Obviously we want to increase our score. So here is the equation to combine two children:

$$p = \tanh\left(W\begin{pmatrix} c_1 \\ c_2 \end{pmatrix} + b\right)$$
$$s = U^T p$$

Question is how to choose which words or actually which nodes we are going to combine in each step. The answer is that we follow a simple greedy algorithm. We try all the adjacent nodes and we choose the one with the highest score. Now we do the same thing between other nodes and the parent node. We will do it again and again until we exhaust all the nodes and we end up having just one node which is the representation of the sentence. Score of the tree y is sum of the parsing decision scores at each node.

$$s(x,y) = \sum_{n \in nodes(y)} s_n$$

Notice that y is the tree we made during parsing using greedy algorithm. For objective function we use max-margin parsing (Taskar et al., 2004) which is a supervised max-margin objective.

$$J = \sum_i s(x_i, y_i) - \max_{y \in A(x_i)} \left(s(x_i, y) + \Delta(y, y_i)\right)$$

Notice that y_i are the correct trees we have in our dataset and the set $A(x_i)$ is the set of every possible trees. We used a simple greedy algorithm to search through this huge space but there are better and more effective search

algorithms you can use. Also, notice that the term $\Delta(y, y_i)$ is the difference between the correct tree and the tree in hand so it penalizes incorrect decisions.

Next issue may come to mind is that we are using the same neural network and the same weight matrix W for combining all type of combinations, but what if we relax that constraint? What if we let the model to have different weights. First issue it makes is that doing this increases our weight matrices to learn but the performance boost we gain covers up for the cost we pay. This type of model which learns different weights for different type of syntactical combination is called syntactically united recursive neural networks (SU-RNN) (Socher et al., 2013a).

For the next step, we can think about the effect each word can have on the other one. Like adverbs emphasis next word comes after them but how can we do that? How can we make the model to combine an adverb and the next word in a way that the meaning of the second word is scaled? After all, all we have is a word vector which represent the meaning if the word. To be able to embed the effect of each word on the others we need to have a word matrix instead of a word vector. This is called matrix vector recursive neural networks (MV-RNN).

But results of these models still show three types of errors. First is called negated positives. If we can change the whole sentiment of a sentence from positive to negative by changing one word, MV-RNN cannot weight that one word strong enough to flip the whole sentiment of the sentence. Second type of errors is called negated negative. For example, when we say something is "not bad" we lessened the negativeness of the sentence to the neutral. MV-RNN has difficulties to recognize it. The last error type is "X BUT Y conjunction." It means if the first part of the sentence "X" is negative but the last part "Y" is positive the whole sentence is positive. MR-RNN has problems to deal with this.

But if you think about the structure of the network in each step we concatenate two vectors and put them in a affine transformation, then we apply a nonlinearity on it. But if we need to express the effect of words on each other in a nonlinear way we need to combine them in a way that allows it. In order to do that, first we concatenate two word vectors which form a vector $x \in \mathbb{R}^{2d}$ and we use a quadratic instead of a linear transformation:

$$p = \tanh\left(x^T V x + W x\right)$$

Note that V is a third-order tensor with $\mathbb{R}^{2d \times 2d \times d}$ dimension. We compute $x^T V\,[i] x \forall i \in [1, 2, \ldots, d]$ slices of the tensor outputting a vector $\in \mathbb{R}^d$. Then we can add Wx and then apply nonlinearity. The quadratic shows that we can indeed allow for the multiplicative type of interaction between the word vectors without needing to maintain and learn word matrices. This model is called recursive neural tensor network (Socher et al., 2013b).

7 CONVOLUTIONAL NEURAL NETWORKS

In the context of natural language processing, Recurrent Neural Networks (RNNs) are strong and effective. However, these nets have some issues. They cannot capture phrases in isolation and require prefix (left side) context to capture a phrase. As following Fig. 23, if we want to have a representation of "my birth" in the whole sentence, phrase vector $\begin{bmatrix}2.5\\3.8\end{bmatrix}$ not only captures "my birthday," but also it captures "the country of." In other words, you always go from left to right, thus, you usually see your classifier only at the end once it reads the whole sentence in a left to right flow. In this situation, it is hard to keep the complex relationship alive throughout this flow over numerous time steps. This is one of the issues of RNNs that CNNs tries to resolve.

The main idea of CNN is that instead of computing a single representation of vector at every time step that captures the left side context, it computes a phrase vector for every single phrase in the sentence. In the abovementioned example, at the first step, we can compute vectors for two words "the country" in isolation. For the whole sentence, we can compute vectors for all bigrams or trigrams including "the country," "country of," "of my," "my birthday," "the country of," "country of my," "of my birth" and then the fourgrams, i.e., "the country of my" and "country of my birth" and finally "the country of my birth." If this was a sentiment classification, for instance, and for one them, the vector is captured "not very good," eventually we will try to handle and push that vector to softmax by some other forms which will be described soon. This is the idea of the first layer of a convolutional network for NLP. This will compute vectors regardless of whether the phrase is grammatical. We know from parsing that certain phrases like "the country of" is not a proper noun phrase, rather, it is an odd ungrammatical chunk. However, this model does not care about linguistic or cognitive possibility in any kind of way for language. Since these representations (bigrams, trigrams, etc.) can be computed in parallel, this is going to be a big advantage. After computing all these vectors, we group them.

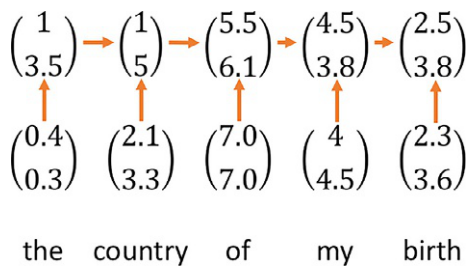

FIG. 23 An example of having a representation of a phrase in a sentence.

7.1 What is Convolution?

Following, is 1d discrete convolution which is the simplest definition for any convolution operator of a filter over another function.

$$(f \times g)[n] = \sum_{m=-M}^{M} f[n-m][g[m]]$$

where n is a specific point in time, and M, in the context of NLP, is the window size. Basically, you multiply a filter at different locations of the input. Convolution is very strong and effective to extract features from images. Next figure, shows a 2D example where yellow square is filter weights and the green one is input (Fig. 24).

Single layer CNN is a simple variant using one convolutional layer and pooling. This is based on the work of Collobert and Weston (Collobert et al., 2011) and Kim (2014) on convolutional neural networks for sentence classification. First we define the notations clearly.

- Word vectors: $x_i \in \mathbb{R}^k$
- Sentence: $x_{1:n} = x_1 \oplus x_2 \oplus \ldots \oplus x_n$ (vectors concatenated)
- Concatenation of words in range: $x_{i:i+j}$
- Convolutional filter: $w \in \mathbb{R}^{hk}$ (goes over window of h words)
- Windows size can be 2 or higher, e.g., 3

According to notations, we start with word vectors in a k dimensional vector. Then, we represent a sentence through concatenation. For concatenating all n word vectors, circled plus operator is used and they are concatenated lengthwise assuming they are all concatenated as a long row. We may want to extract a specific words in a range, from time step i to time step $i + j$. Therefore, our convolutional filter (defined in terms of the window size of h the vector size k) will be a vector w of parameter that are going to be learned with standard stochastic gradient decent-type optimization methods. The size of the filtering affects the learning significantly. Longer the filter leads to more computation to handle. Also, longer filters are able to capture more phrases but you are more likely to overfit your model. The size of the filter should be a hyperparameter. There are some tricks, where you can have multiple filters with multiple lengths which allow you to prevent overfitting. We will discuss it more in this section.

Assume that it is needed to have a convolutional filter that at each time step looks at three different word vectors and tries to combine them into some kind of feature representation such as a single number. Then, we have three times number of dimensions of each word vector filter. Fig. 25 shows a very simple example of a convolutional filter for a two-dimensional word vector with a window size of 3, hence, we basically have a six dimensional w here. You should note that w is a single vector, not a matrix, just as out word vectors that are concatenated into a single vector.

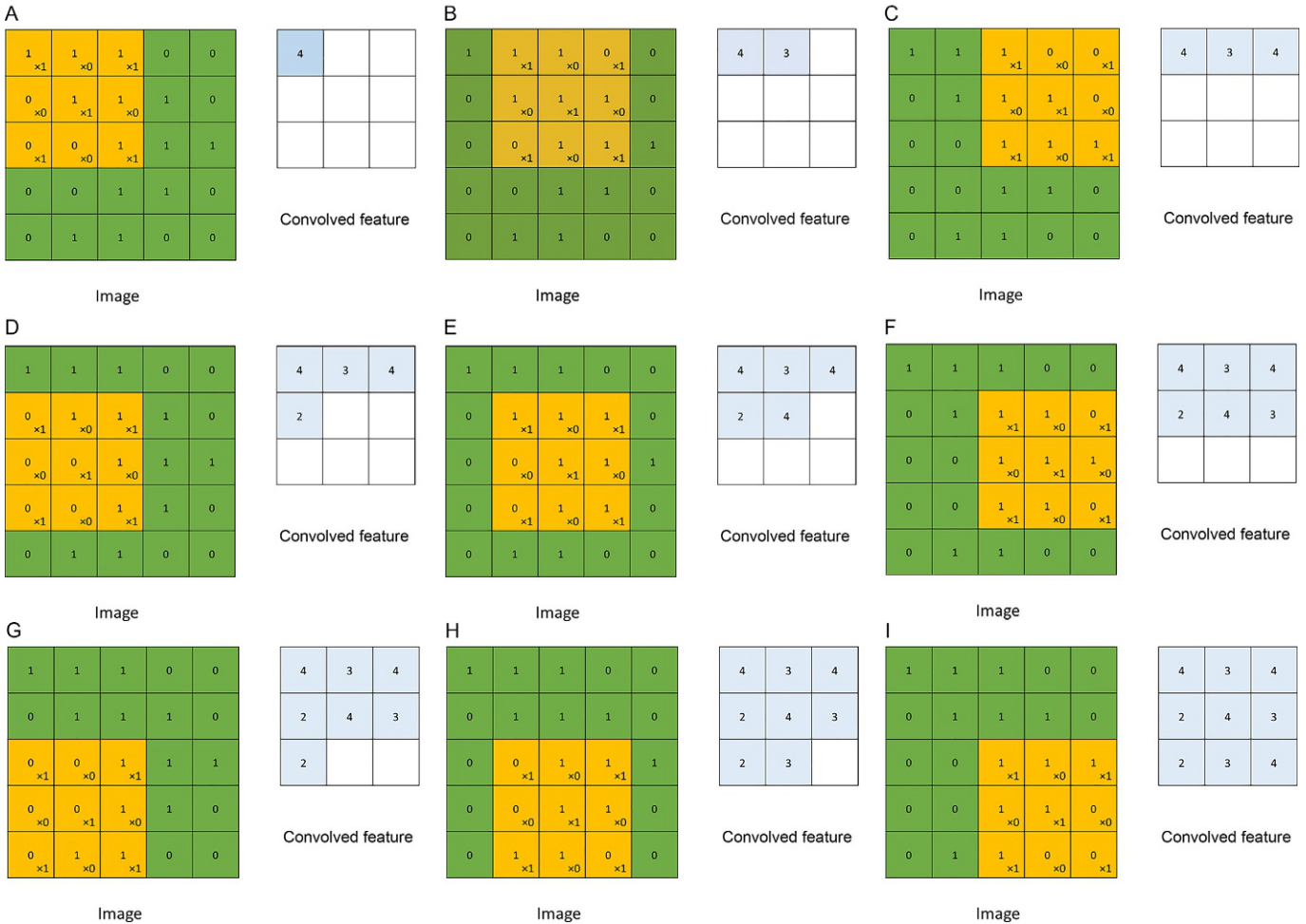

FIG. 24 2D convolution example: *yellow* shows the filter weights and *green* shows input from Stanfod UFLDL wiki. (A) Step 1. (B) Step 2. (C) Step 3. (D) Step 4. (E) Step 5. (F) Step 6. (G) Step 7. (H) Step 8. (I) Step 9.

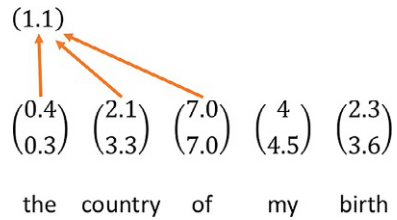

FIG. 25 Example of convolutional filter for a two-dimensional word vector with a window size of 3.

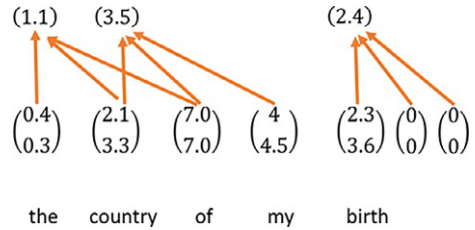

FIG. 26 How to apply filter to last words of the sentence.

Now we discuss why it is a neural network and describe the computations. In order to compute a feature in a time step, for the previous example, we have an inner product of w vector of parameters multiplied by the i-th time step plus window size, h.

$$c_i = f(W^T X_{i:i+h-1} + b)$$

For example, to calculate c_1, we have W times X_{13} or simply we have the concatenation of those word vectors in our product. b is the bias term and we add a nonlinearity at the end.

Having the sentence, $x_{1:n} = x_1 \oplus x_2 \oplus \ldots \oplus x_n$, all possible windows of length h are $x_{1:h}$, $x_{2:h+1}$, …, $x_{n-h+1:n}$. Since we have the computation of c_i at every time step, it means that we have a feature map defined as $C = [c_1, c_2, \ldots, c_{n-h+1}] \in \mathbb{R}^{n-h+1}$. Each c value takes the same w and has inter-products with different windows in each time step.

If continue concatenating the words, when we reach the last word vector, we require two more words to apply the filter. As illustrated in Fig. 26, zero vectors (in this example two vectors) are to apply last word of the sentence. This may also be done on the left side of the sentence.

C vector is going to be a long $n - h + 1$ dimensional vector and it is going to be of different length for sentences with different number of words. However, if we want to plug it into a softmax classifier, it needs a fixed dimensional vector. Due to this variable length vector at this point, we want to eventually have a fixed dimensional feature vector representing the whole sentence. To do so, a new type of building block called a pooling operator

or pooling layer will be introduced. In particular, we use a max-over-time pooling layer (or max-pooling layer). The idea is to capture the most important activation. As there are different elements computed for every window, we hope that the inner product is large enough for that filter if it sees a certain kind of phrase. Assume that the word vectors are relatively normalized. Then it is desirable to have a large cosine similarity between the filter and certain pattern, e.g., positive words or phrases and only one filter will only be good at picking up that pattern. This will be capture by the largest c_i which has a very large activation for that particular filter w. Consequently, we get this as $\hat{c} = maxC$ and it can ignore all the rest sentence. It is going to be able to pick out one particular bigram very accurately. The problem here is that $\hat{c}$ is just a single number of all the element in C vector. However, in addition to just one particular type of bigram or trigram, we need to extract more features, hence, we are going to have multiple filters w through convolving multiple of them. As we train this model, we hope that some of the filters will be very active and have very large interproducts with particular types of bigrams or trigrams.

We can have multiple different window sizes and in each time step we will max pool to get a single number for that filter for that sentence. It should be noted that since we have a random initialization, when we use different filters of different of same length they learn different feature. In other words, as we apply SGD, different filters will move and start to pick up different patters in order to maximize overall objective function as the result of random initialization.

It is worth to mention that there are several researches that explore different pooling schemes and there is no proper mathematical reason of which one works better, however, in the max pool we can intuitively see that we try to fire when a specific type of N-gram is observed and this signal is passed to the next higher layer. Thus, using a single value (as the result of max pool) is better than other approaches like averaging all values in C (averaging may also wash out the strong signal that we get from one particular unigram, bigram, or trigram).

Now we are going to discuss another idea that combines the concept of word vectors with some extensions and instead of representing the sentence only as a single concatenation of all of the word vectors, we start with two copies of the sentence. Then, we are going to backpropagate into only one set and keep other "static." In order to explain this, remember that word vectors can be trained on a very large unsupervised scope so they capture semantic similarities. Now, if you start backpropagating your specific task into the word vectors, they will start to move around when you see that word vector in your supervised classification problem in that dataset. This means that as you push certain vectors that you see in your training datasets somewhere else, the vectors that you do not see stay where they are and might misclassified if they only appear in the test set. Consequently, by having these two

channels, we try to have some of the goodness of the first copy of the word vectors to be really good on that task while the second set of word vectors to stay where they are, having proper general semantic similarities in vector space goodness that we have from supervise word vectors.

Both of these channels are going to be added to each of c_is before applying max-pool, hence we will pool over both those channels.

The final model which is the simplest one, is just concatenating all the $\hat{c}_i$s to obtain final feature vector, $z = [\hat{c_1}, \ldots, \hat{c_m}]$ where m is the number of filters. Then we will plug z directly into softmax, and train $y = softmax(W^{(S)}z + b)$ with standard logistic regression cross entropy error. Note that by using two copies of word vectors, we certainly doubling the memory requirement of the model. However, it is only the second copies of word vectors that we are going to backpropagate into for the task.

Following is a graphical description the model (Kim, 2014) (Fig. 27). Here, we have n words and each word has k dimensions. This particular model shows us two applications of a bigram filter (shown with red lines) and one of a trigram filter (shown with yellow lines), then, they are max-pooled to a single number. For each of the filters, we obtain one long sent of features and then we get a single number after max-pooling over all the activations.

7.2 Tricks to Improve the Performance

There are some tricks that were employed by Kim in 2014 (Kim, 2014) to improve the performance of the simple model described above. Here, we present a couple of these tricks. The first one is dropout which is described in Section 5.2.2. However, it is one of the effective tricks that can be applied in different contexts and it is differently applied for the convolutional

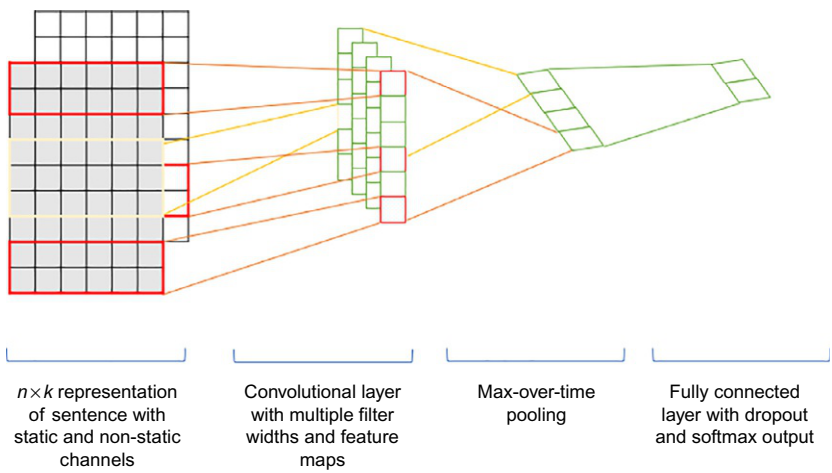

FIG. 27 Graphical description of CNN for n words with k dimensions.

networks and recurrent neural networks. Therefore, it is good to investigate another application of dropout for this particular convolutional neural network. The idea is to randomly mask or dropout or set to zero some of the feature weights that are in final feature vector. In our case, it is z. In other words, we are going to create a mast vector r of Bernoulli random variables with probability of p (as a hyperparameter) set to 1 and with probability $1 - p$ set to zero. This ends up deleting certain features during training. Consequently, as we go through all the filters using a bigram and another bigram, it might accidentally or randomly one of the two bigrams. This prevents coadaptation (overfitting to see specific feature constellations).

$$y = softmax(W^{(S)}(roz) + b)$$

In order to understand what happens at training time, assume $p = 0.5$. Then, half of the features are randomly eliminated during training and the mode is going to get used to see a much smaller in norm feature vector z. In other words, gradients are backpropagated through those elements of z for which $r_i = 1$. During test time where there is no dropout (because we do not want to delete any feature), feature vectors z are larger, hence, the final vector is scaled by means of Bernoulli probability p. Then, we will end up in the same order of magnitude as we did at training time.

$$\hat{W}^{(S)} = pW^{(S)}$$

It is reported that dropout leads to 2–4 percent improvement in accuracy and it makes it possible to use very large networks without overfitting.

Another regularization trick is using L_2 norms of weight vectors of each class, c, (row in softmax weight $W^{(S)}$ to fixed number s (a hyperparameter)). If $||W_{c}^{(S)}|| > s$, then it will rescale it so that $||W_{c}^{(S)}|| = s$. Intuitively, they will force the model to never be too certain and have very large weights for any particular class. There is a minor improvement using L_2 regularization in this way, however, it is not very common to apply it.

If we review the model described, we have several adjustments and hyperparameters, hence, one should understand each hyperparameter, know what they really are, and know which ones really matters to final performance of the research or project being conducted. The particular convolutional neural network model described in this section amazingly have the same set of hyperparameters for a lot of different experiments (Kim, 2014) including sentiment analysis and subjectivity classification. Since there are different possible values for each hyperparameter and it is not practical to evaluate all of the possible combinations, one needs to limit the boundaries for each hyperparameter and select a set of randomly selected hyperparameters values.

There are some other choices that you need to make when you are building a CNN. We will review these choices in the following and hopefully they help you better understand CNNs.

7.3 Narrow vs Wide Convolution

In Fig. 24, we described convolution through applying a 3 × 3 filter. As we can see, the filter works for the center of the matrix. The question is how we can apply the filter in the edges where there is no neighbor for the elements, for example, for the top-left element elements on top and left side are missing to be able to apply the filter. As shown in Fig. 26, one solution is zero-padding, i.e., considering value of zero for all missing elements and use them for further calculations. This is called *wide convolution*. If this is not utilized, it will be called *narrow convolution*. Fig. 28 is the graphical representation of narrow and wide convolution for an input size of 7 and filter size 5 (Kim, 2014). As demonstrated in this figure, wide convolution is helpful and sometimes necessary when the filter is relatively large to the input size. In out simple example, narrow convolution yields to an output size of 3 while wide convolution leads to an output size of 11. General formulation for the output size is as follows:

$$n_{\text{out}} = (n_{\text{in}} + 2 \times n_{\text{padding}} - n_{\text{filter}}) + 1$$

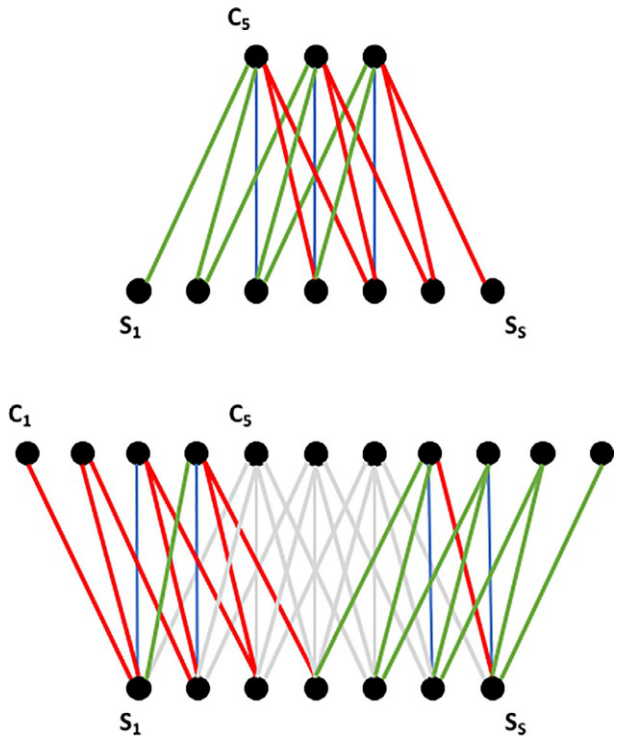

FIG. 28 Narrow convolution vs wide convolution with input size of 7 and filter size of 5.

7.4 Stride Size

Stride size is the shift of filter at each step which is a hyperparameter for the convolutions. Following is an example of stride sizes of 1 and 2. As shown in the figure, larger stride size leads to smaller output size. Typically, the stride size of 1 is used in the literature, however, larger stride size makes the model behaving like an RNN (Fig. 29).

7.5 Application of CNN as Input for RNN

One of the most exciting applications was to take such CNN, have various pooling operations and take that as input to all time steps of a recurrent neural network for machine translation. In fact, it uses CNN for encoding instead of an LSTM (and benefiting from the possibility of parallelization) and RNN for decoding (Kalchbrenner and Blunsom, 2013; Kalchbrenner et al., 2014).

8 MEMORY

When we are reasoning or incorporating information, we tend to remember salient information and forget useless parts of the information. Memory is an important aspect of natural language understanding and processing. To start with, we review RNNs and discuss the bottleneck that limits its performance. Recurrent neural networks are ways of designing new architectures that allow us to process sequences of variable length and can be think of

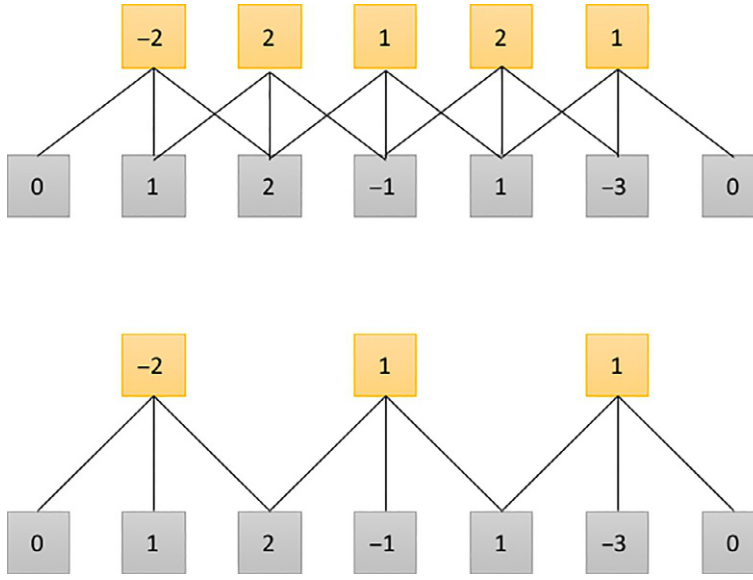

FIG. 29 Stride size of 1 vs stride size of 2.

processing units that take inputs and then produce outputs so that, for example, the distribution over what the next word is gained given the history and they track history by having a loop recurrence through updating their states and sharing parameters across the transformations. Conceptually, the activation of hidden layers in an RNN is modeling the history of a set of sequences through connecting back layer to their own layer. One of the reasons for the popularity of RNNs in NLP is that they fit variable width problems well. They are also capable of capturing long range dependencies. Some obvious applications of RNNs are language modeling, sequence labeling, and sentence classification. However, for each of these tasks, there are simpler and better models. One of the interesting things about RNNs is that they can model probabilities in a continuous basis. We can condition the decoding mechanism on data, then, the conditional probability as the sequence of texts can be modeled given vector representation (β in Fig. 30) and decoding can be influenced by β (fed as extra input) as the initiating state of decoding process by concatenating with the output.

The idea of conditional language modeling has led to transduction problems which can cast several NLP tasks such as parsing and translation. For example, when you are observing a French sentence, a representation β for that sentence is produced to be decoded in English.

The approach of transduction is usually referred as sequence-to-sequence mapping in the literature. The goal is to transform a source sequence, $s = s_1, s_2, \ldots, s_m$ (for example, the French sentence), into a target sequence, $t = t_1, t_2, \ldots, t_n$ (for example, the English translation). The basic form of doing so is to produce a representation for the source sequence, s, then the conditional probability of the next symbol of the target sequence will be modeled. Fig. 31 is the graphical description of deep LSTM for translation where you read in the French sentence word by word into RNN and update the hidden cells. When you indicate switching modes by hitting the symbol of finishing the sentence at the end of the French sentence, RNN starts to output English words and feeding back the generated sentence.

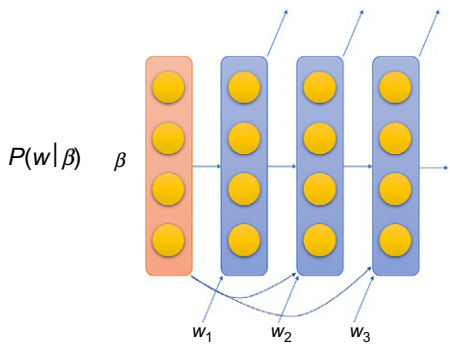

FIG. 30 Transduction with conditional models.

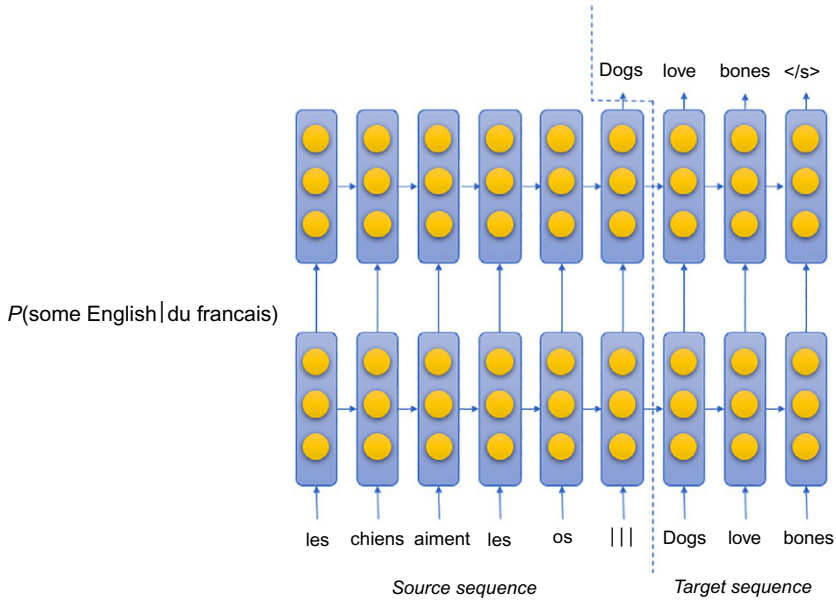

FIG. 31 Deep LSTM for translation (Sutskever et al., 2014).

Recently, there are researches who use the same architecture for machine translation for computation. In the paper of Zaremba and Sutskever (2014), instead of sentences as the inputs, they used Python scripts character by character and the output was numerical result of the computation character by character (instead of translated sentence).

Now we discuss the bottleneck for simple RNNs. This very generic sequence-to-sequence model is capable of dealing with lots of tasks. However, there is a bottleneck between the source and the target when the transduction is being performed. As shown in Fig. 32, given the French sentence, all the information required to decode the English sentence has to be present in the activations of the network at this particular time step. The previous state only matters during training, however, all the information is needed during decoding. This figure is an example of nonadaptive capacity. It can be imagined that the source sentence grows so that if you try to translate longer bits of French, more information needs to be compressed to the fixed size representation. Therefore, since there is no ideal infinite compressor, we get bigger losses as decoding continues unless we make the network bigger. This is an undesirable property that the size of the hidden layers needs to map on the data and it is needed to break this bond.

In addition, when we train these models, the target sequence modeling should be learned first. For instance, when we are translating from English to French, first, the model learns what an average sort of French sentence

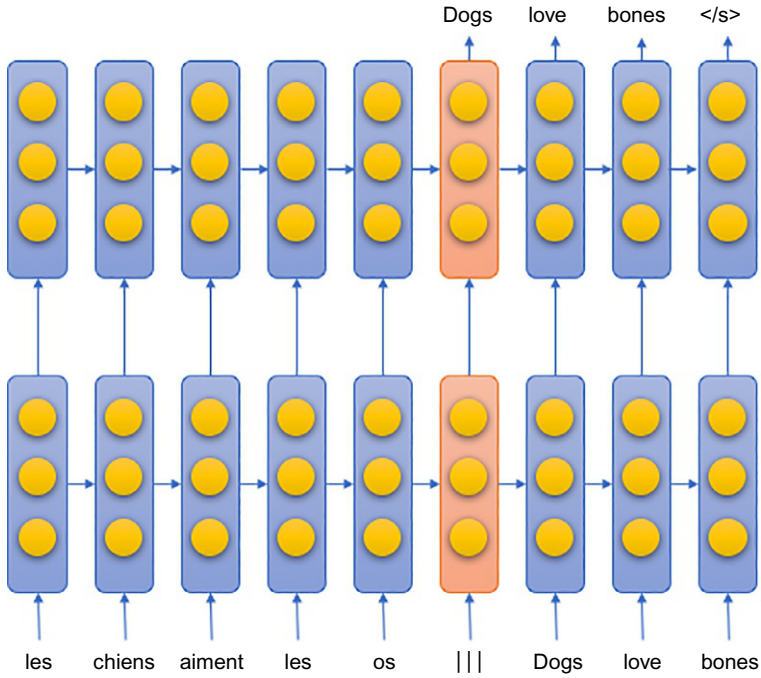

FIG. 32 Bottleneck for simple RNNs.

looks like before propagating dependencies back into the source and properly conditioning on the source. Hence, a lot of time will be spent on learning French language and a small part of the time is spent on learning dependencies toward the end of learning. Another way of looking at this in terms of learning is that we get a rich gradient close to the classification decisions we make regarding what the distribution of a word is and for the gradient to flow back to the activation, thus, the weights can capture dependencies through the nonlinearities of the network. That is why gradient starvation on the encoder side exists which is time consuming to learn.

From a computational perspective, there are some limitations to RNNs. There are different levels in computational hierarchy including finite state machines (FSMs), pushdown automata (PDA), and Turing machines. FSMs (which are placed at the bottom of the computational hierarchy) are transitions between a finite number of states that models the class of regular languages. If you add a stack to an FSM you obtain a PDA (places in the middle of the hierarchy) which corresponds to the class of context-free languages that you might generate from context-free grammar. Then, we get to the Turing machines (at the top of the hierarchy) where the corresponding language class is recursively enumerable languages or computable functions. RNNs are as

expressive as a Turning machine. In the literature, it is stated that any Turing machine can be encoded as RNN if you set the proper weights.

As we discussed before, feedforward networks are universal function approximators which means if you have a large enough feedforward network, you get an arbitrary bound with the actual function. However, expressivity is not equal to learnability. In other words, the ability of RNNs to express a Turing machine does not mean that is able to learn it. RNNs can express or approximate a set of Turing machines, but, it does not mean that they can learn from data a very precise set of weights.

Simple neural networks such as GRU and LSTM cannot learn Turing machines in normal sequence pace. There are several reasons for this. First of all, RNNs do not control a notion of external tape. Inputs are seen as they are fed by the training mechanism and it is not possible to go back and revisit previous inputs and you cannot update the internal state they operate in lock-step with the sequence being run over. In other words, sequence is exposed in forced order. Another weaker argument is that maximizing the likelihood of the target, given the sequence modeling, produces model close to training data distribution. In addition, we can regularize RNNs through putting penalties and you can try and induce sparsity in the weights with L_1 regularization described in Section 5. However, it is unreasonable to expect yielding computational model that express a Turing machine by a simple regularization mechanism like L_1.

We can think of simple RRNs as approximator for FSMs. For instance, when you are learning a language model, you are learning order-N Markov chains effectively while you do not need to specify N (in contrast to N-gram language models where it is needed to specify order-N Markov assumption in advance). Consequently, LSTMs, RNNs, and GRUs can be considered as memoryless in theory. However, they can simulate memory in terms of dependencies which is very limited and bounded form of memory. There is no intensive to learn beyond the range length structure and complexity of dependencies observed in the time of training.

There are some problems with this type of paradigm. Since RNN needs to capture dependencies in its activations, RNN state has to act not only as memory but also as controller as it is required to keep tracking of the dependencies to release the information and allocate new information in memory. Moreover, having longer dependencies to be captured as well as tracking more dependencies leads to more memory requirements while fixed size RNN hidden layer is limited in terms of what information can be stored without destructing information as the result of compression. Also, memory requirement increases for more complex or structured dependencies. When an interdependency with high complexity in the source needs to be captured and used to produce translation more accurately, required RNN size becomes larger. Ultimately, if there is a high performance machine to handle all these problems, FSMs are very basic in terms of their capabilities. Arguably,

natural languages such as English are at least context-free or weakly context-sensitive and they should be modeled via a PDA. It is argued that these languages do not need to use higher order grammars to understand natural language. Even if we do not consider that natural language is context-free, due to the rule of parsimony, it is also useful to model more complex computational models with RNNs. For example, assume a very simple context-free language that is not recognized by an FSM, $a^n b^n$. Then, there is an unbounded number of a's and same number of b's.

Regular language ($N + 1$ *rules*)

$$\varepsilon|(ab)|(aabb)|(aaabbb)|\ldots$$

In practice, the actual number of a and b is bounded. If the number n is bounded, this becomes regular again. The regular language that used to capture this bounded version has $N + 1$ rules. Thus, empty strings (ab), ($aabb$), and so on should be matched.

CFG (two rules)

$$S \rightarrow aSb$$
$$S \rightarrow \varepsilon$$

Context-free grammar has two rules (if counting is included, it will have three rules) including match the recursion and then match the empty string occasionally to terminate the recursion. Considering the computational hierarchy explained before, in the sense of LSTM, deep LSTM, GRUs, etc., we are at the FSM level and we need to move up the hierarchy.

Here, we briefly review stepping back and making into an API. As shown in Fig. 33, abstractly, an RNN takes inputs and produces outputs and maintains state. In Fig. 34, a more complex RNN is shown to demonstrate graphically that a lot more is happening in the cell which fits API. All the operations inside the cell are employed to wire arrows shown in the figure.

We are modeling a function, $\text{RNN}: X \times P \rightarrow Y \times N$, in which the input is $x_t \in X$, previous recurrent state is $p_t \in P$, the output is $y_t \in Y$, and $n_t \in N$ is the updated recurrent state. Abstractly, it is modeling a sequence from inputs and previous states toward the sequence of outputs and next states. Usually, previous and next state's domain are the same and the domains of inputs and outputs are related. In summary, we keep things differentiable and states are tracked using P/N (states track overall state of our machine) and we are free to manipulate inside the cell, hence, we can add elements of memory in the cell. In order to enhance RNNs with memory, we explore different models that satisfy the general blueprint where there is a controller within RNN cell to deal with input and output and there is a memory which is interacting with controller (Fig. 35). The controller can be like LSTM and both controller and memory can have their own state preserved through time (mutable or immutable).

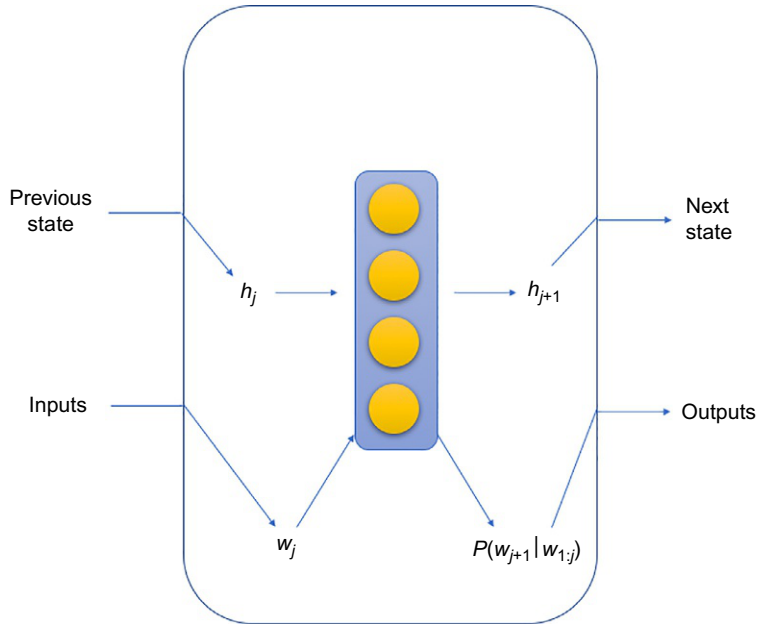

FIG. 33 An RNN with inputs and outputs.

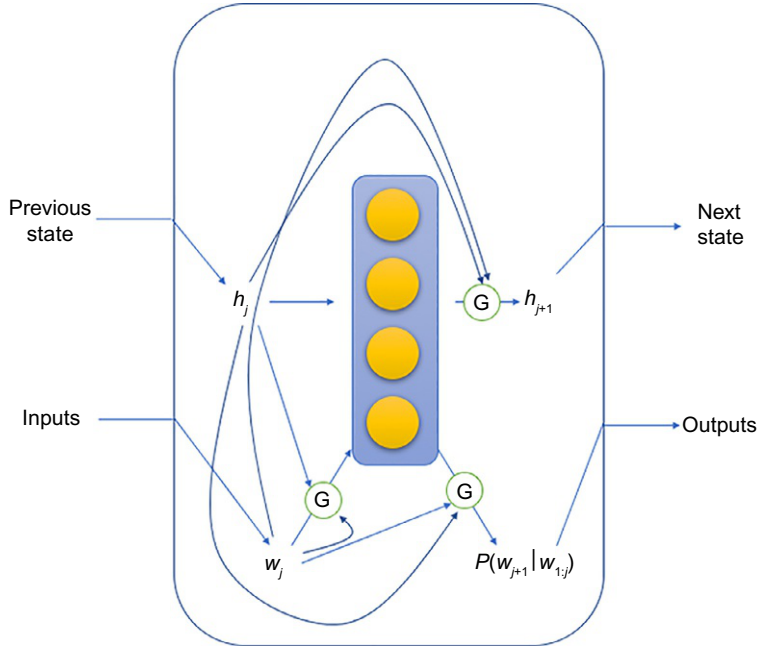

FIG. 34 A recurrent cell.

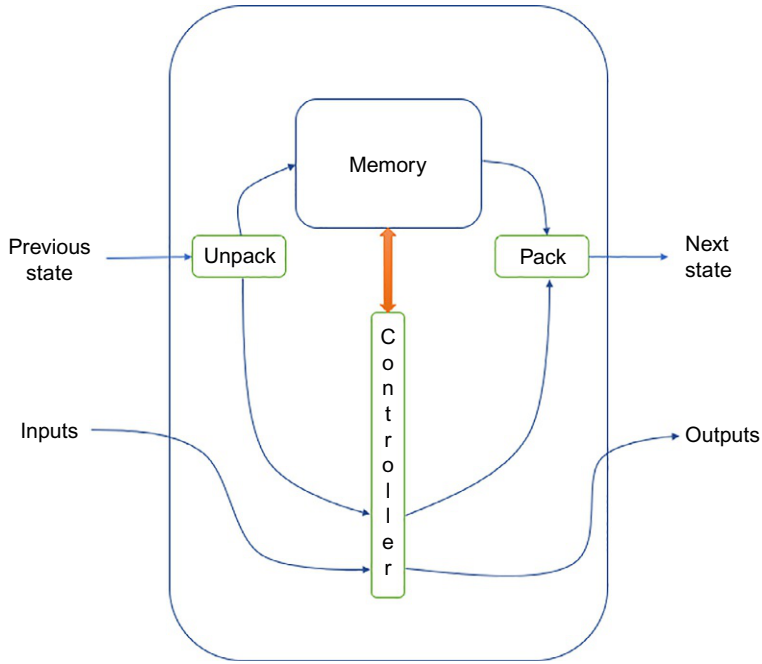

FIG. 35 The controller–memory split.

8.1 Attention (ROM)

We start with the notion of attention (ROM). Basically in attention, we have an array of vectors representing some data produced by some mechanism. The controller, which is the LSTM, deals with I/O logic and provides some ways to extract information from attention matrix. At each time step, you need to read the data array and accumulate gradients in the memory on the backward path. There are different forms of attention. First is often called early fusion (see Fig. 36) which is based on the previous state of the controller. A vector will be produced from the memory and concatenated with the inputs of the controller to produce output. The memory is read-only, thus, it does not change and memory and controller are repacked to produce the next state to allow retrieving the next read. In early fusion, controller looks at the state of the memory according to the query produced by its previous state before updating its own internal state. Therefore, a non-Markov assumption is made here about tracking the history of the attention. Based on the previous state of the controller, we take some sort of inner product paraphrasing or other similarity of kernel with each vector in the memory to product a logic. By soft maxing, we have a distribution over the positions in the memory which can be seen as memorizing. Attention will take a mean field estimate of the state

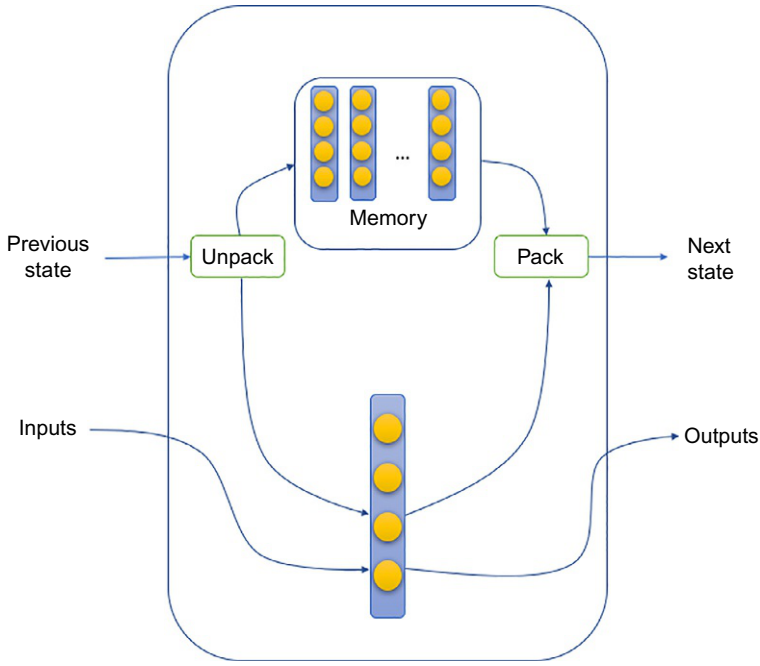

FIG. 36 The controller–memory split (early fusion).

memory based on the stochastic distribution. Therefore, according to the probability of each slot, they will take the weighted average of all values in memory which will be a vector and the vector will be concatenated with the RNN inputs and update internal state which in turn serves to retrieve another vector in the next time step that is responsible for the producing the outputs.

Another type of attention is late fusion (Fig. 37). In late fusion, there is a similar process to early fusion except that current state of the RNN is going to be used to retrieve information from the memory and then it is concatenated with the output of the RNN to predict the next word. In contrast to early fusion, we are making a Markov assumption while we are keeping track of the history of attention within the RNN. We concatenate the output of the attentive process with the output of RNN and making a prediction over the outputs, jointly.

Attention is abstractly a kind of read-only memory. The encoder in a sequence model will produce an array of representation, e.g., for a sort of vector for token, as the interpretation of this vector, it is both the content to be retrieved by the attention mechanism and a semantic index for that position. In encoder-decoder models with ROM, representations are packed into an attention matrix and decoder is a controller which affects the operations. This has some good properties which will be discussed here and results in

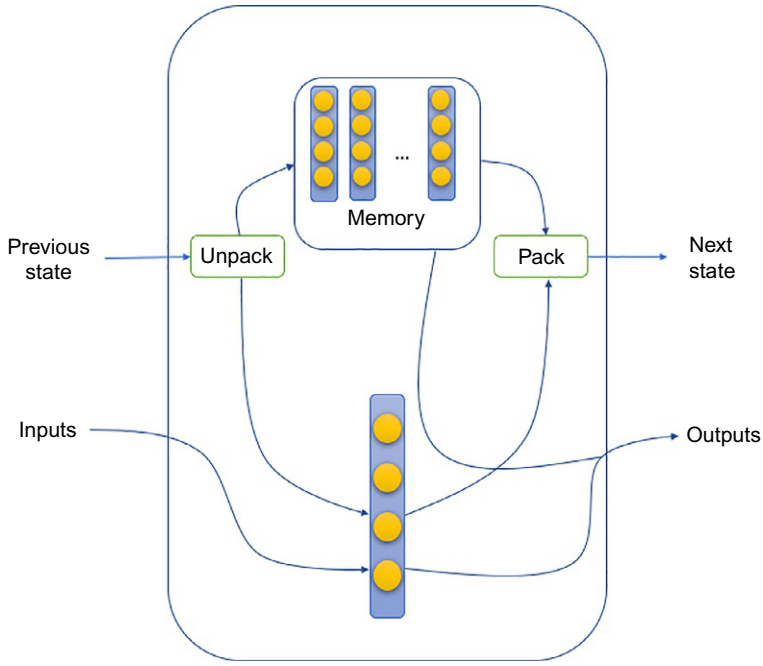

FIG. 37 The controller–memory split (late fusion).

bypassing the transduction bottleneck we discussed at the beginning of the section (where the encoder is gradient starved). Attention can also be seen as computing soft alignment between sequences.

In a standard sequence-to-sequence model, all the information required to decode into a natural language, e.g., English, should be presented in the red layer shown in Fig. 32. Attention can be used to enhance it. When the encoder was running forward in the same sequence-to-sequence model and except during training, we did not care about what these activations are. During the forward path, it is required to preserve these as attention matrix to be used. Thus, during the decoding, we look back upon the past to check the best match among the words in target language for translation, e.g., French to decide what to translate based on the hidden layer activations at any time step. Since this operation is not differentiable, gradient is flowing from the decisions made about what to generate through the activations of the RNN as well as attentive decisions (Fig. 38). This makes gradient to come directly to the output states of RNN during encoding phase instead of going through the nonlinearities.

Some of the applications of this are recognizing textual entailment (RTE) and missing translation. RTE is the task of comparing a pair of sentences (one called hypothesis and one called premise). The relationship between these two sentences is whether the premise is contradictional to the hypothesis, or neutral or entailment. For example,

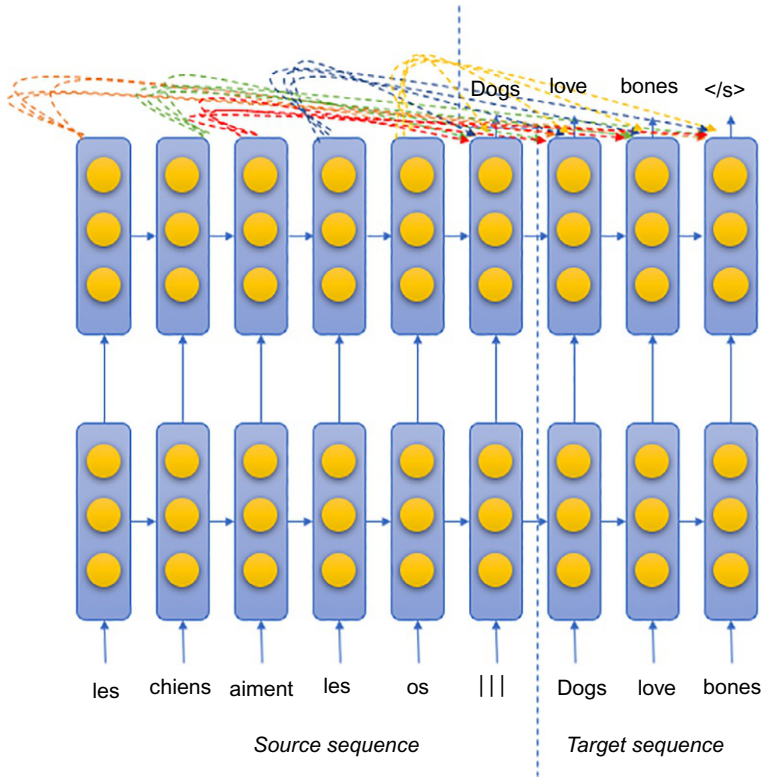

FIG. 38 RNN with attention.

A man is celebrating a touchdown at the stadium.

- The man is at a concert → Contradiction (indicated in red)
- The man is drunk → Neutral (indicated in blue)
- The man is at a football game → Entailment (indicated in green)

Due to high ambiguity and its reliance to world knowledge, this is a very difficult task. Fig. 39 illustrates word-by-word attention in a simple sequence-to-sequence task (a simple end to end approach as a sequence classification task). The sum of the first end to end runs LSTM over premise and hypothesis to classify premise as neutral, contradiction, or entailment. While reading the hypothesis, the system is allowed to check the premise to find triggers for contradiction or entailment.

Using the attention, it is possible to visualize what the system is thinking which leads to its decision. It is visualized by a heat map which is illustrated in Fig. 40. As shown in the figure, there should be no relation between hypothesis and premise, however, it shows that boy and girl are aligned to kids and the system uses this to make a decision.

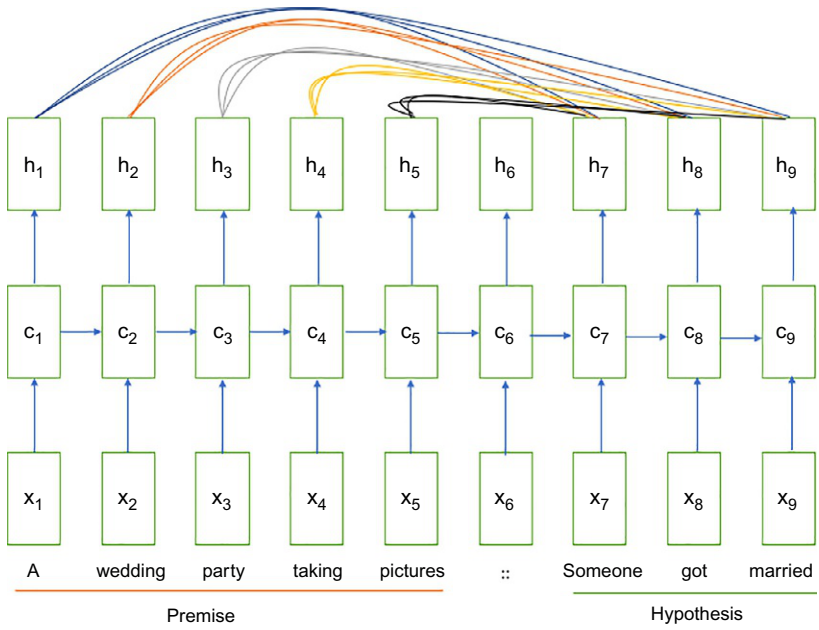

FIG. 39 Word-by-word attention.

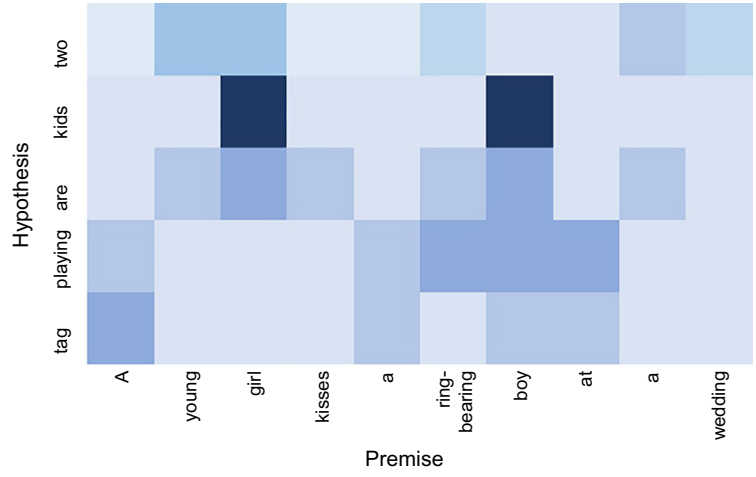

FIG. 40 Attention heat map.

For large-scale comprehension, it is needed to read a long story of thousands of words, for instance from a news article, to answer a question which is presented before or after reading the story. It can be a closed form query to fill the blank or gap and the model needs to return the element that

completes the sentence properly. This is one the areas that ML with attention can be utilized. To do so, if a sequence-to-sequence approach is employed, a very long range dependency needs to be captured which is difficult to do. When attention is used, this long range dependency capturing is pushed out of the RNN state and it will be possible to just find the most similar element in the story. In other words, an embedding is created for each token in the story or document and the query is to read word by word. At each step, we need to attend over the document through query and combine the attention distribution, iteratively. Then, we can predict the answer more precisely. This can also be visualized using a heat map that indicates which words or phrases or part of the document is used to make decision.

Attention is a successful approach to mitigate the limitations of the original sequence-to-sequence models and it is adaptable to different problems. It makes it possible to skip transduction bottleneck. There are a variety of other attention and coattention networks in different models that can be tailored for specific processes. As the limitation of attention, it forms a read-only memory which puts a strong onus on controller to track what has been read and makes the encoder to do a lot of legwork. It is required to capture proper representations on the encoder in order to make it possible for controller to differentiate different entities.

8.2 Register Machines (RAM)

We discussed attention as a read-only mechanism in order to illustrate the stepping back process from RNNs to the idea of API within which we can have controller–memory split and their interaction. Here, we return to the computational hierarchy and discuss the controller–memory split which is utilized to generate differentiable register-based machine or random-access memory (RAM). In the computational hierarchy, now we jump from FSMs to Truing machines where we have a randomly addressable tape and the ability to read and write arbitrary locations. Here we discuss how to do so and explain this kind of API.

Attention is considered as ROM. In the early fusion or late fusion, read-only memory and the controller interact with each other. Now let us assume that the controller can affect the memory state and update it through time. The general idea behind neural random-access memory (neural RAM) and register machines is that the controller, which deals with inputs and outputs, is capable of generating differently parameterized distributions over memory register in order to read or write. It is up to the controller to what and where to write or delete from the memory. In contrast to attention, the state of the controller and memory are updated and after producing outputs by recurrent cell, it goes to next state (Fig. 41).

Here, the previous state is unpacked to the previous hidden layer of the controller and memory as well as the last read vector. The last read from

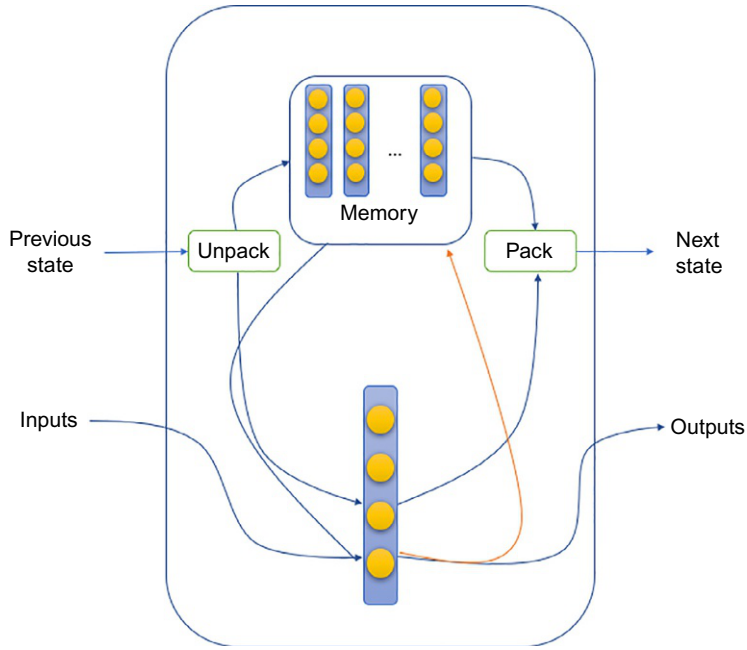

FIG. 41 Neural random-access memory.

memory will construct the inputs to the controller and the current input to the overall module. It is a standard LSTM style of updating where we are feeding these inputs and the previous controller state to the controller and the next state of the controller will be produced and the overall output of the network will be generated. Also, a set of read and right keys will be generated (one or more keys for reading and one or more keys for writing) as well as one or more vector for what you are going to write over proper write head. Through an attention style mechanism over the memory, read keys are used to produce a read vector to be fed to the controller in the next time step (similar to the attention). In contrast to attention, at the write stage, the memory state will be updated by the controller based on the value of the output vector instead of having an immutable memory. This is a simple type of RAM style mechanism for RNNs which implements a content-based addressing. The distribution over what and where you are going to write can be improved based on the key values and the contents of the memory.

There are some extensions in the literature. In location-based addressing, for example, instead of having an attention style mechanism over the memory (in addition to that), the controller role is to produce a shift instead of producing keys and to update read head. Another extension is using a mix of location and content-based addressing. Also, hard addressing is utilized instead of soft addressing. In this extension it is not possible to

back propagate through discrete decisions made about what and where to write, but you can use reinforcement learning-based methods. Furthermore, there are extensions including memory prefixes, memory key/content factorization, and heuristic addressing mechanisms.

The idea of having the controller to be able to read from and write to attention matrix was initially sold as Neural Turing Machine. Briefly, we describe the relation between these models and Turing machines. In contrast to RNNs that cannot move over the tape, it can be seen that within the memory that is preserved and updated through time, we are internalizing part of the tape that described as external in classical Turing machines. Moreover, through the content-based or location-based addressing, controller is able to control the tape motion. RNN can be assumed to act as a controller and can learn the state transition in an actual Turing machine. However, it does not provide the generality that an intermission has. Because the number of available operations in this paradigm is operating in lockstep input and output. Consequently, whenever a symbol is read or outputted, it is required to read the memory. This will cause a problem for some problems like sorting (an $n\log(n)$ algorithm). Because under this paradigm, you need to have n steps to output the target sequence and the $n\log(n)$ steps in between in order to rearrange everything in the memory and perform pivots and permutations if you are internalizing the quicksort algorithm. As the computational steps tie with the data, you can only learn linear algorithms and nothing beyond that with this.

Memory networks and advanced memory mechanisms have not been widely applied to real natural language processing tasks. The reason is that while there is a promising intuition about doing RAM and it is possible to store arbitrary bits of information in the registers, there is a lot of moving parts and hyperparameters in these architectures and the complexity of these architectures makes them hard to train and tune. Also, there is not always an immediate mapping from the performing capabilities of these architectures to problems we want to solve.

8.3 Neural Pushdown Automata

As discussed above, currently there are difficulties to reach to the top of the computational hierarchy, i.e., Turing machines. However, between the FSMs and Turing machines, there is a class of simpler computational models which is called PDA which adds a stack to an FSM and is capable of modeling context-free languages.

As shown in Fig. 42, this still fits the same sort of controller–memory split similar to Turing machine and the controller not only reads from memory (as an attention), but also it is able to update the memory. However, when the memory acts like a stack, it constrains the update.

In the literature, there are researches that introduced neural differentiable stack and we describe one of them. The basic idea is that there will be a controller that deals with input and output and preserves its state. In addition,

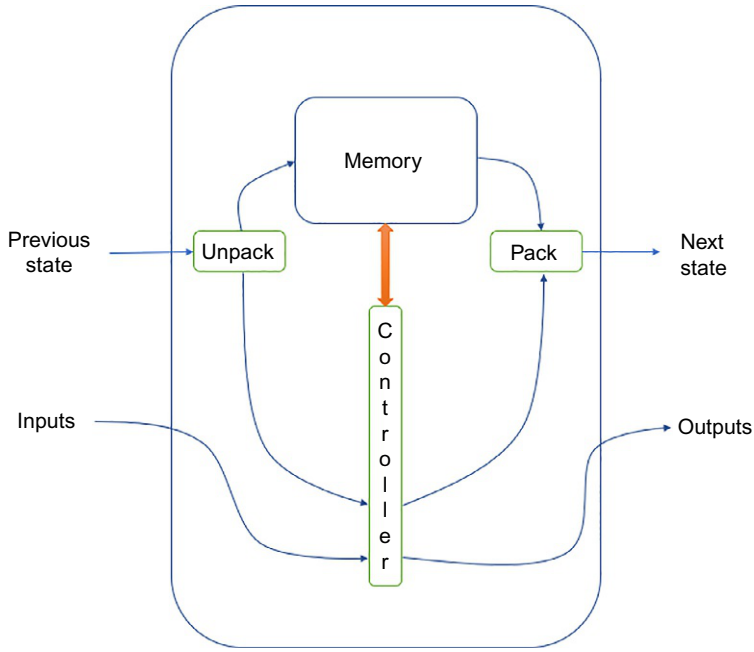

FIG. 42 The controller–memory split in PDA.

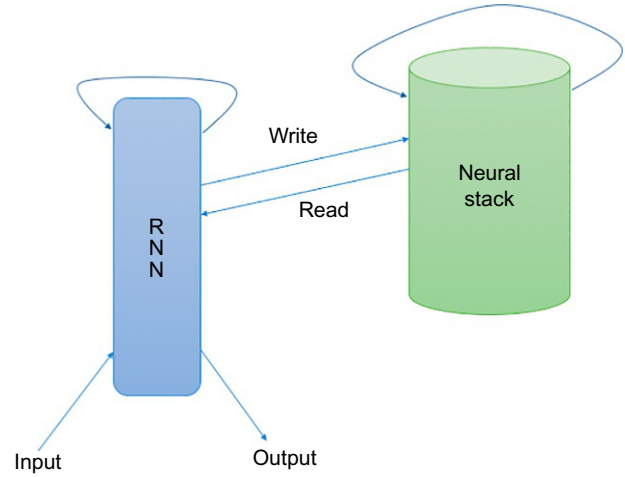

FIG. 43 Representation of a neural stack and its interaction with RNN.

there is a differentiable approximation of the stack and the controller can update memory and write by pushing a new vector to stack or by popping off a vector. It does not need to read the entire values and there is a restricted access for reading and writing (Fig. 43). As shown in Fig. 44, neural stack is

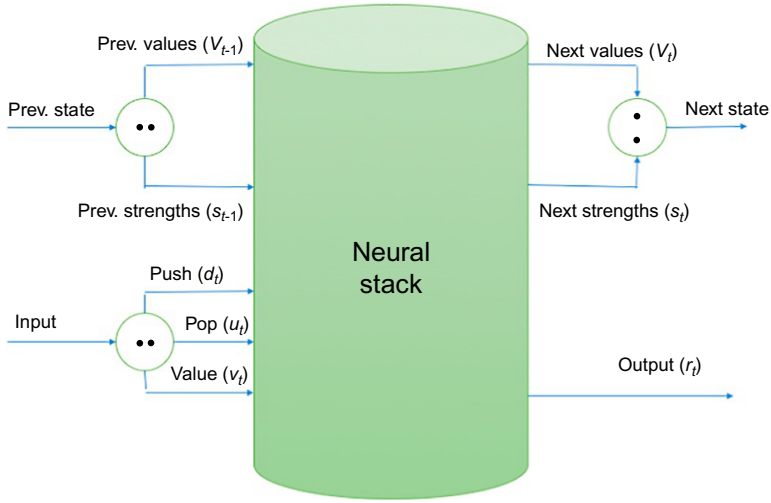

FIG. 44 Neural stack modeling.

difficult to model due to its complexity and typically they have a very discrete data structure. In order to make things differentiable, it is required to introduce continuous relaxation for those operations as well as the notion of a stack state. To do so, a differentiable stack can be assumed as an RNN since it has inputs, outputs and it tracks its own state. Inputs are a continuous push or pop signal and also a vector value. Regarding the state, if we decompose the stack into two elements, one is the values that historically pushed to the slack and at a particular time step, we check how long it has been that these values are in the slack. The previous state will be updated to have new values and strength based on the push pop and value inputs.

The controller task is to read inputs from the environment. In order to connect the neural stack to the controller, the controller read symbols that it may push to the stack or it may condition the output and it should generate a specific output. It also has its previous state. There are several elements in the controller's output and the controller generates the overall output of RNN cell as well as the signal for the neural stack (Fig. 45).

Having a stack in RNN, there are few experiments that we can do. It is possible to evaluate stack and queue enhanced RNNs on synthetic transduction tasks such as copying and reversing sequences. Therefore, data without any local linguistic regularity can be generated. More linguistically motivated tasks examples are subject-verb-object to subject-object-verb reordering and genderless to gendered grammar generated by a context-free grammars.

Different forms of differential data structures such as stack and queues where it is possible to push and pop on both ends can be compared in terms of their performance in dealing with different tasks. Fig. 46 shows the performance of each one in dealing with the experiments. In this evaluation, the

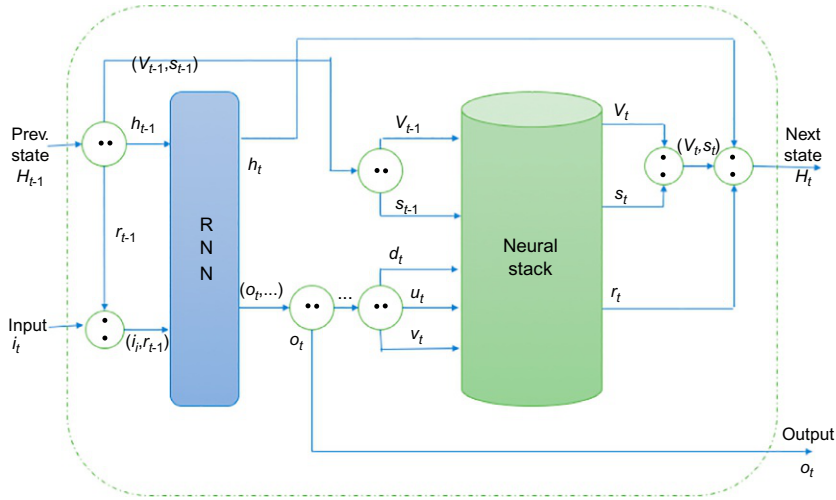

FIG. 45 RNN and neural stack modeling.

Experiment	Stack	Queue	DeQue	Deep LSTM
Copy	Poor	Solved	Solved	Poor
Reversal	Solved	Poor	Solved	Poor
SVO-SOV	Solved	Solved	Solved	Converges
Conjugation	Converges	Solved	Solved	Converges

FIG. 46 Performance evaluation.

solution for longer sequences are preserved for each Neural Stack, Queue, or DeQue that solves the problem and then they are tested against a 2× length of training sequences.

In order to summarize the Neural PDA, initial experiments demonstrate that these differential stacks allow us to obtain models that are decent approximations of classical PDA when they are trained long enough. They are capable of solving problems that PDA should be able to solve. These differential data structures are supposed to be effective and effective for syntactically rich natural language where tree or nested structures exists, thus, they should

be useful for parsing and compositionality. However, these are difficult to implement and require some work to apply. Finally, they are tied to the number of computational steps, a similar problem in a normal Turing machine where we are not able to take arbitrary spans of computation prior to producing output. Otherwise, reversing and copying are possible with two stacks. Similar to Neural Turing Machines, it is an attempt to climb the computational hierarchy; this is a good step from FSM approximator to PDA approximator. However, it cannot be considered as a complete step.

Looking at these stacks, Neural Turing Machines and attention mechanisms, they are easy to design a complex model with their strong intuitions behind what form of computation they are doing (Kumar et al., 2016). However, it is not clear how well these models can solve real complex problems and they are not always worth it to develop the model. You should always think of the complexity of the problem that needs to be solved in addition to thinking about the model.

9 SUMMARY

In this chapter, we presented application of deep neural network approaches to natural language processing tasks. We discussed feedforward neural networks, training and optimization of DL architectures, regularization for deep learning, language modeling, convolutional neural networks, and memory.

REFERENCES

Bahdanau, D., Cho, K., Bengio, Y., 2014. Neural machine translation by jointly learning to align and translate. arXiv:1409.0473 (arXiv preprint).

Bengio, Y., Simard, P., Frasconi, P., 1994. Learning long-term dependencies with gradient descent is difficult. IEEE Trans. Neural Netw. 5 (2), 157–166.

Chapelle, O., Scholkopf, B., Zien, A., 2006. Semi-Supervised Learning. MIT Press.

Chen, S.F., Goodman, J., 1996. An empirical study of smoothing techniques for language modeling. In: Proceedings of the 34th Annual Meeting on Association for Computational Linguistics, pp. 310–318.

Cho, K., Van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., Bengio, Y., 2014. Learning phrase representations using RNN encoder-decoder for statistical machine translation. arXiv:1406.1078 (arXiv preprint).

Collobert, R., Weston, J., Bottou, L., Karlen, M., Kavukcuoglu, K., Kuksa, P., 2011. Natural language processing (almost) from scratch. J. Mach. Learn. Res. 12, 2493–2537.

Dahl, G.E., Yu, D., Deng, L., Acero, A., 2012. Context-dependent pre-trained deep neural networks for large-vocabulary speech recognition. IEEE Trans. Audio Speech Lang. Process. 20 (1), 30–42.

El Hihi, S., Bengio, Y., 1996. Hierarchical recurrent neural networks for long-term dependencies. In: Advances in Neural Information Processing Systems, pp. 493–499.

Goodfellow, I., Bengio, Y., Courville, A., 2016. Deep Learning. MIT Press.

Greff, K., Srivastava, R.K., Koutník, J., Steunebrink, B.R., Schmidhuber, J., 2017. LSTM: a search space odyssey. IEEE Trans. Neural Netw. Learn. Syst. 28 (10), 2222–2232.

Heafield, K., Pouzyrevsky, I., Clark, J.H., Koehn, P., 2013. Scalable modified Kneser-Ney language model estimation. In: ACL (2), pp. 690–696.

Hochreiter, S., Schmidhuber, J., 1997. Long short-term memory. Neural Comput. 9 (8), 1735–1780.

Jozefowicz, R., Zaremba, W., Sutskever, I., 2015. An empirical exploration of recurrent network architectures. In: Proceedings of the 32nd International Conference on Machine Learning (ICML-15), pp. 2342–2350.

Kalchbrenner, N., Blunsom, P., 2013. Recurrent continuous translation models. In: EMNLP, vol. 3, p. 413.

Kalchbrenner, N., Grefenstette, E., Blunsom, P., 2014. A convolutional neural network for modelling sentences. arXiv:1404.2188 (arXiv preprint).

Kim, Y., 2014. Convolutional neural networks for sentence classification. arXiv:1408.5882 (arXiv preprint).

Koutnik, J., Greff, K., Gomez, F., Schmidhuber, J., 2014. A clockwork RNN. International Conference on Machine Learning, pp. 1863–1871.

Krizhevsky, A., Sutskever, I., Hinton, G.E., 2012. Imagenet classification with deep convolutional neural networks. In: Advances in Neural Information Processing Systems, pp. 1097–1105.

Kumar, A., Irsoy, O., Ondruska, P., Iyyer, M., Bradbury, J., Gulrajani, I., Zhong, V., Paulus, R., Socher, R., 2016. Ask me anything: dynamic memory networks for natural language processing. In: International Conference on Machine Learning, pp. 1378–1387.

Lin, T., Horne, B.G., Tino, P., Giles, C.L., 1996. Learning long-term dependencies in NARX recurrent neural networks. IEEE Trans. Neural Netw. 7 (6), 1329–1338.

Mikolov, T., Sutskever, I., Chen, K., Corrado, G.S., Dean, J., 2013. Distributed representations of words and phrases and their compositionality. In: Advances in Neural Information Processing Systems, pp. 3111–3119.

Mikolov, T., Yih, W.-T., Zweig, G., 2013. Linguistic regularities in continuous space word representations. In: NAACL-HLT, vol. 13, pp. 746–751.

Mozer, M.C., 1992. Induction of multiscale temporal structure. In: Advances in Neural Information Processing Systems, pp. 275–282.

Pascanu, R., Mikolov, T., Bengio, Y., 2013. On the difficulty of training recurrent neural networks. In: International Conference on Machine Learning, pp. 1310–1318.

Pennington, J., Socher, R., Manning, C.D., 2014. Glove: global vectors for word representation. EMNLP, vol. 14, pp. 1532–1543.

Rohde, D.L.T., Gonnerman, L.M., Plaut, D.C., 2006. An improved model of semantic similarity based on lexical co-occurrence. Commun. ACM 8, 627–633.

Socher, R., Lin, C.C., Manning, C., Ng, A.Y., 2011. Parsing natural scenes and natural language with recursive neural networks. In: Proceedings of the 28th International Conference on Machine Learning (ICML-11), pp. 129–136.

Socher, R., Bauer, J., Manning, C.D., Ng, A.Y., 2013. Parsing with compositional vector grammars. In: ACL (1), pp. 455–465.

Socher, R., Perelygin, A., Wu, J.Y., Chuang, J., Manning, C.D., Ng, A.Y., Potts, C., et al., 2013. Recursive deep models for semantic compositionality over a sentiment treebank. In: Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP), vol. 1631, p. 1642.

Sussillo, D., 2014. Random walks: training very deep nonlinear feed-forward networks with smart initialization. arXiv:1412.6558 (arXiv preprint).

Sutskever, I., Vinyals, O., Le, Q.V., 2014. Sequence to sequence learning with neural networks. In: Advances in Neural Information Processing Systems, pp. 3104–3112.

Taskar, B., Klein, D., Collins, M., Koller, D., Manning, C.D., 2004. Max-margin parsing. In: EMNLP, vol. 1, p. 3.

Wan, L., Zeiler, M., Zhang, S., Le Cun, Y., Fergus, R., 2013. Regularization of neural networks using dropconnect. In: International Conference on Machine Learning, pp. 1058–1066.

Wang, S., Manning, C., 2013. Fast dropout training. In: Proceedings of the 30th International Conference on Machine Learning (ICML-13), pp. 118–126.

Warde-Farley, D., Goodfellow, I.J., Courville, A., Bengio, Y., 2013. An empirical analysis of dropout in piecewise linear networks. arXiv:1312.6197 (arXiv preprint).

Zaremba, W., Sutskever, I., 2014. Learning to execute. arXiv:1410.4615 (arXiv preprint).