Recurrent Neural Networks, Long Short-Term Memory Networks, and Recommender Systems

StartOnAI

1 Recurrent Neural Networks

1.1 Introduction and Notation

In the next two chapters, we'll cover the applications of deep learning to *natural language processing* (NLP). Human language is a system constructed for the purposes of conveying meaning to others, and is not produced by a physical manifestation of any kind. As such, it is very different from computer vision or other similar tasks. Most words are just symbols for an extra-linguistic entity—a word is *signifier* that maps to a *signified* concept or idea.

The symbols of language can be encoded in several modalities, including voice, gesture, and writing, which are continuously transmitted to the brain, which in itself seems to encode things in a continuous manner. The goal of NLP is to be capable of designing algorithms that allow computers to understand natural language for a given task.

Two examples of sequential data in NLP are audio files (music, human speech, etc) and text (written language), both of which you could process with *sequence models* (such as recurrent neural networks, which will be covered shortly)! Some examples of tasks you could perform on audio files are speech recognition, music transcription, and voice recognition. Examples of tasks you could perform on text include sentiment classification (in the context of TV show reviews, this would mean the intent of the reviewer: good or bad?), machine translation (i.e. Vietnamese to English), and named entity recognition (recognizing proper nouns, such as StartOnAI!).

As for notation, let's consider the task of named entity recognition as a motivating example. Consider the sentence "Elon Musk declared a revolution." In named entity recognition, we could represent this sentence, x, as a row vector, with 1s representing named entities:

$$\begin{bmatrix} 1 & 1 & 0 & 0 & 0 \end{bmatrix} \tag{1}$$

We can index each of the words in this sentence with $x^{<t>}$ (with the first word being $x^{<1>}$), and the length of the sentence and output vector are denoted as

T_x and T_y, respectively. To represent all of the words used in a given training set, we can parse the training set and choose the top n words to put in a *dictionary*. Then, we can represent each word as a one-hot vector of length n. If we encounter a word that's not in our dictionary, we can generate a new token UNK to represent the unknown word.

1.2 RNNs

Consider the named entity recognition (NER) task that was introduced in the previous text cell. If we were to train an ordinary deep neural network to learn the mapping from an input sentence to a row vector with the locations of named entities, we'd face the issue of working with variable length sentences and outputs. In addition, ordinary neural networks have no notion of "memory", in the sense that if they learned one relevant feature in one location in a text corpus (a set of texts), they wouldn't be able to directly apply it to other locations.

To address these issues, we can construct a *recurrent neural network* (RNN), which allows information to "persist" through time. In the context of the task at hand, we can individually make predictions for the each the words in the input sequence by using one layer per word that maps from $x^{<t>}$ to $y^{<t>}$. Then, we can use the activation from $x^{<t-1>}$ in making the prediction for $x^{<t>}$ such that the information from $x^{<1>}$ to x^{t-1} is "carried over" into the activation of $x^{<t>}$, thus allowing for information to persist across sequences. An illustration of this architecture is presented below (with slightly different notation):

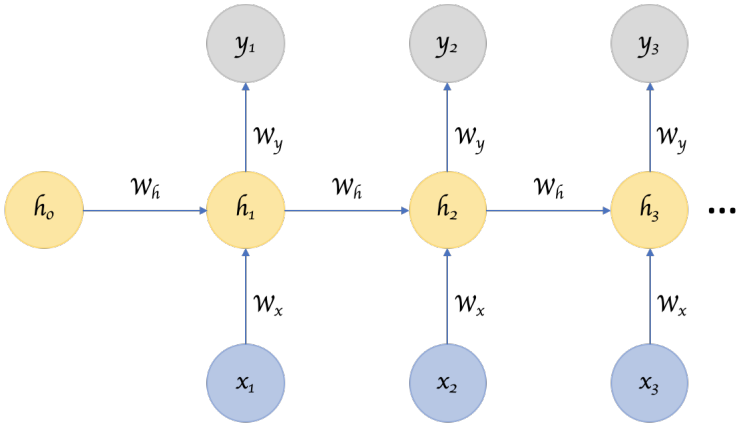

Figure 1: An example of an RNN. (Venkatachalam, M)

We'll compute $a^{<t>}$ as $g(W_{aa}a^{<t-1>} + W_{ax}x^{<t>} + b_a)$, where W_{aa} are the weights of the connection from layer $t-1$ to t, W_{ax} are the weights of the layer attempting to map $x^{<t>}$ to $y^{<t>}$, b_a is a bias, and g is an activation function (usually tanh or ReLU). Then, to compute $\hat{y}^{<t>}$, we can simply use

$g(W_{ya}a^{<t>} + b_y)$, where g is a (possibly different) activation function dependent on the output (for binary classification, it would be sigmoid), W_{ya} is another set of weights, and b_y is another bias.

1.3 Backpropagation Through Time

Instead of using the usual backpropagation algorithm for computing the gradient of the cost with respect to the parameters, we'll use *backpropagation through time*. The loss function for the t-th word is defined as

$$L^{<t>}(\hat{y}^{<t>}, y^{<t>}) = -y^{<t>} log\, \hat{y}^{<t>} - (1 - y^{<t>})\, log(1 - \hat{y}^{<t>}) \tag{2}$$

(the standard cross-entropy loss), and the loss function for the entire sequence is defined as

$$L(\hat{y}, y) = \sum_{t=1}^{T_y} L^{<t>}(\hat{y}^{<t>}.y^{<t>}) \tag{3}$$

(the sum of the losses for every word). Then, we can perform ordinary backpropagation through each of the layers and through of the connections between the layers, which then allows us to update the parameters with an optimization algorithm.

So far, we've only seen an example where $T_X = T_y$ (that is, the input sequence is of same length as the output sequence), which is called a *many-to-many* architecture. A *many-to-one* architecture (which is used in sentiment classification) is one where the input sentence is fully "read" by the network before outputting a single prediction in the last layer. In contrast, a *one-to-many* architecture (used for music transcription) is one where the input is used only in the first layer.

1.4 RNNs as Language Models

Simply put, language models compute the probability of occurrence of a number of words in a given sequence. For instance, which sentence is more likely: "He ate a pair." or "He ate a pear."? Without any context, most people would say that the second sentence is more likely. Thus, language models are a fundamental component of both speech recognition and machine translation, since both require such models to figure out if a transcription/translation is accurate.

To build a language model with an RNN, we'll need a training set comprised of a large corpus of text. Then, for a given input sentence, we'll *tokenize* it into one-hot vectors according to a dictionary generated from the training set (and add an end-of-sentence/EOS token at the end). Then, we can construct the RNN as follows:

1. The first "layer" will take in $x^{<1>}$ and $a^{<0>}$ (both of which are just 0) and use them to generate an activation $a^{<1>}$ and a prediction $\hat{y}^{<1>}$ with softmax.

2. The second "layer" will take in $x^{<2>}$ (the actual first word) and $a^{<1>}$ to generate an activation $a^{<2>}$ and a prediction $\hat{y}^{<1>}$ based on the previous $t-1$ predictions.

We can then define the loss function as

$$L = \sum_t L^{<t>}(\hat{y}^{<t>}, y^{<t>}) \tag{4}$$

where

$$L^{<t>}(\hat{y}^{<t>}, y^{<t>}) = -\sum_i y_i^{<t>} log\ \hat{y}_i^{<t>} \tag{5}$$

(which is just the softmax loss; don't worry about this!). This RNN-based language model can therefore learn to predict the next word in a sequence given the previous words.

1.5 Vanishing Gradients

An RNN-based language given a sentence with a long-term dependency (such as verb tense consistency) would actually not be able to capture this dependency due to the vanishing gradient problem, which prevents a given RNN from memorizing whether a noun was singular or plural and predicting the appropriate verb later in the sentence. Thankfully, we can address this issue with *gated recurrent units* and *long-short term memory networks*

2 Long Short-Term Memory Networks

2.1 Long-Term Dependencies

Consider the sentence "I grew up in *France* (...) I speak fluent *French*". If we were to use an ordinary RNN as a language model to predict the next word (which would eventually be French), we would need to remember the context of France, from further back. Unfortunately, as the gap between words with long-term dependencies grow, ordinary RNNs become unable to learn to connect the information. In theory, RNNs are absolutely capable of handling such "long-term dependencies." A human could carefully pick parameters for them to solve toy problems of this form. Sadly, in practice, RNNs don't seem to be able to learn them.

2.2 Gated Recurrent Units

Consider the equation for computing the activation at time t in an RNN: $a^{<t>} = g(W_a[a^{<t-1>}, x^{<t>}] + b_a)$ (this is a simplified version of the equation from the last notebook!). A *gated recurrent unit* (GRU), which was introduced by two groups of authors in 2014, contains a *memory cell* c.

At time t, a memory cell will have the value $c^{<t>}$, and at every timestep,

we'll consider overwriting $c^{<t>}$ with a candidate value $\tilde{c}^{<t>} = tanh(W_c[\Gamma_r \cdot c^{<t-1>}, x^{<t>}]+b_c)$, where Γ_r is a "relevance gate" computed with $\sigma(W_r[c^{<t-1>}, x^{<t>}]+b_r)$, W_c are weights, $c^{<t-1>}$ is the previous memory value, and b_c is a bias value (again, this is just simplified notation!).

Then, to decide whether or not to overwrite $c^{<t>}$, we'll use an update gate $\Gamma_u = \sigma(W_u[c^{<t-1>}, x^{<t>}] + b_u)$, which is either very close to 0 or very close to 1. Thus, $c^{<t>} = \Gamma_u \cdot \tilde{c}^{<t>} + (1 - \Gamma_u) \cdot c^{<t-1>}$, where $\cdot$ denotes elementwise multiplication. (Note: Γ_u can, of course, be n-dimensional in correspondance to the dimension of the memory cell; Thus, the gate actually updates individual bits)

To summarize, these are the four main equations that govern the behavior of GRUs:

- Relevance gate: $\Gamma_r = \sigma(W_r[c^{<t-1>}, x^{<t>}] + b_r)$
- Candidate memory cell value at time t: $\tilde{c}^{<t>} = tanh(W_c[\Gamma_r \cdot c^{<t-1>}, x^{<t>}]+b_c)$
- Update gate: $\Gamma_u = \sigma(W_u[c^{<t-1>}, x^{<t>}] + b_u)$
- Memory cell value at time t: $c^{<t>} = \Gamma_u \cdot \tilde{c}^{<t>} + (1 - \Gamma_u) \cdot c^{<t-1>}$

For instance, consider the sentence "The W-Trucks were very fresh." At $t = 2$, the memory cell would update to remember that "W-Trucks" is plural. Then, at $t = 3$, the memory would use the memory cell to use the correct noun. After then, the memory cell can be updated again to remember new long-term dependencies.

2.3 LSTMs

Long-Short Term Memory Networks (LSTMs), like GRUs, are a type of RNN (a *gated* RNN) capable of learning long-term dependencies, and were first introduced in 1997. LSTMs are generally considered to be more powerful and general than GRUs due to using multiple gates in one LSTM cell. Instead of having just an update gate, LSTMs have an update gate, a forget gate (so it can optionally hold on to a memory value and add to it), and an output gate. Here are the equations you'll need:

- Memory Cell Candidate Value: $\tilde{c}^{<t>} = tanh(w_c[a^{<t-1>}, x^{<t>}] + b_c)$ (notice that we're using different notation for $c^{<t-1>}$)
- Update Gate: $\Gamma_u = \sigma(W_u[a^{<t-1>}, x^{<t>}] + b_u)$
- Forget Gate: $\Gamma_f = \sigma(W_f[a^{<t-1>}, x^{<t>}] + b_f)$
- Output Gate: $\Gamma_o = \sigma(W_o[a^{<t-1>}, x^{<t>}] + b_o)$

- New Memory Cell Value: $c^{<t>} = \Gamma_u \star \tilde{c}^{<t>} + \Gamma_f \star c^{<t-1>}$, where $\star$ denotes elementwise multiplication; This gives the LSTM the option of either keeping the previous memory value and adding $\Gamma_u \star \tilde{c}^{<t>}$ to it or forgetting it and replacing it with the candidate value scaled by the update gate
- Activation: $a^{<t>} = \Gamma_o \star tanh(c^{<t>})$

Here's a diagram of an LSTM cell:

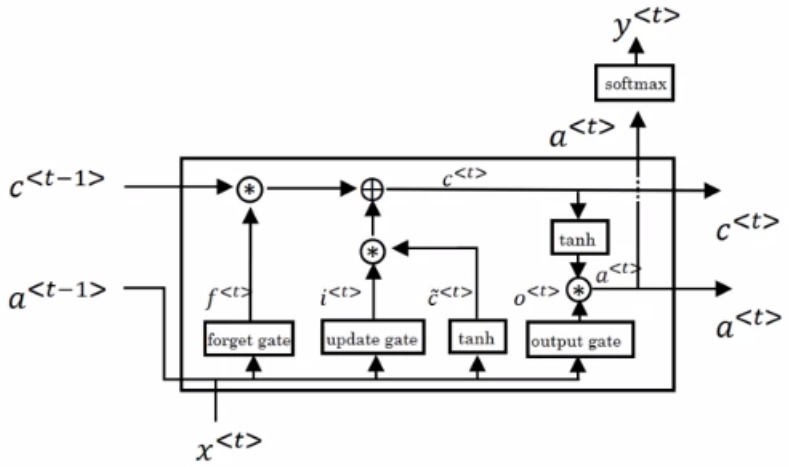

Figure 2: An LSTM cell. (Olah, C. (2015). *Understanding LSTM Networks*)

It's actually fairly easy for an LSTM to remember information for long periods of time as long as the update and forget gates are set properly (think of this as an LSTM's "default behavior"!)

3 Recommender Systems

3.1 Introduction and Terminology

Recommender systems are systems that aim to predict the "rating" that a user would give to an item, and are often used in commercial applications, such as recommending videos on a streaming service or recommending news articles. There are two commonly used types of recommendations: home page recommendations and related item recommendations. Home page recommendations are personalized to a user based on their known interests, and related item recommendations are recommendations similar to a particular item, usually the item currently being viewed (i.e. a video or app), and will usually be displayed alongside that item. In platforms with millions (or even billions) of items, recommender systems can help users find similar items based on their known interests.

Terminology-wise, *queries* (or context) are the information a system uses to make recommendations, and can be combinations of user information (i.e. ID) and additional context (i.e. time of day), and *embeddings* are a mapping from a set (the set of queries/items to recommend) to a vector space called the *embedding space*.

A common architecture for recommender systems consists of *candidate generation* (where the system reduces a large corpus into a much smaller set of candidates), *scoring* (where the systems scores and ranks the candidates), and *reranking* (where the system takes into account additional constraints for the final ranking).

3.2 Applications

Some applications of recommender systems include:

- Product Recommendations: Amazon, Ebay, and other online retailers use recommendations systems to present suggest products to users based on purchases made by similar users
- Streaming Service Recommendations: Youtube, Netflix, Hulu, and other streaming services recommend videos/movies to users based on ratings made by similar users and their general popularity
- News Recommendations: News services have also used recommender systems to recommend articles to readers based on the articles that they have previously read and the similarity of the content from other articles

3.3 Candidate Generation

The two most common candidate generation approaches are *content-based filtering*, where generation is based on similarity between items to recommend items similar to what the user may like, and *collaborative filtering*, where generation is based on similarities between queries and items simultaneously to provide recommendations.

3.3.1 Content-based Filtering

With content-based filtering, items have features, and the user is in the same feature space. Some of the user's features can be explicitly provided, or implicitly provided based on items they have previously interacted with. The model can then use a similarity metric (i.e. dot product) to score each candidate item. While this method can capture a user's specific interests, it is only as good as the hand-engineered features, and has limited ability to expand on the user's interests.

3.3.2 Collaborative Filtering

With collaborative filtering, the limitations of content-based filtering can be addressed, as it uses similarities between users and items *simultaneously*. The embeddings used can be learned automatically without the need for hand-engineering features. If we construct a feedback matrix in which the rows represent users and the column represent movies, we observe that the feedback is either explicit (users specify their rating) and implicit (the system infers that the user is interested). Using multiple embeddings (i.e. type of item and audience), we can place each user and item in the same embedding space such that their interest are explained.

Matrix factorization is a simple embedding model—given a feedback matrix $A \in R^{m \times n}$, where m is the number of users/queries and n is the number of items, the model will learn a user embedding matrix $U \in R^{m \times d}$ (where row i is the embedding for user i) and an item embedding matrix $V \in R^{n \times d}$ (where row j is the embedding for item j). These are learned such that UV^T is a decent approximation of the feedback matrix A. In regards to choosing an objective function, *weighted matrix factorization* is likely the best option, as it decomposes the objective into a sum over observed entities and a sum over unobservations entities, which are treated as zeroes:

$$min_{U \in R^{m \times d}, V \in R^{n \times d}} \sum_{(i,j) \in obs} (A_{ij} - \langle U_i, V_j \rangle)^2 + w_0 \sum_{(i,j) \notin obs} (\langle U_i, V_j \rangle)^2 \quad (6)$$

where w_0 is a hyperparameter that weights the two terms so that the objective is not dominated by one or the other.

With this, we can now minimize this objective function with either stochastic gradient descent or *weighted alternating least squares* (WALS), which is specialized to this particular objective. WALS works by initializing the embeddings randomly, then alternating between fixing U and solving for V and vice versa.

Some limitiations of matrix factorization include the difficulty of using any features beyond the query ID/item ID and the relevance of recommendations (that is, if something is popular, it will be recommended to everyone). With DNNs (in particular, a softmax model), we can address these by treating the problem as a multiclass prediction problem where the input is the user query and the output is a probability vector with a size equivalent to the number of items in the corpus, representing the probability to interact with each item. The loss function of a such a softmax model would compare $\hat{p}$, the output of the softmax layer, and p, the ground truth representing the items the user has interacted with. The cross-entropy loss satisfies this, as it compares two probability distributions.

The probability of item j is given by $\hat{p}_j = \frac{exp(\langle \psi(x), V_j \rangle)}{Z}$, where Z is a normalization constant not dependant on j. $\psi(x) \in R^D$ is the output of the last

hidden layer, and is called the embedding of the query x, and $V_j \in R^d$ is the vector of weights connecting the final hidden layer to the output j, and is called the embedding of item j.

What we called the item embedding matrix V in matrix factorization is now the matrix of weights of the softmax layer. Instead of learning one embedding U_i per query i, the system learns learns a mapping from the query feature x to an embedding $\psi(x) = R^d$. Thus, the softmax model can be thought of as a generalization of matrix factorization.

3.4 Retrieval

At serve time, given a query, the system either looks up the query from the user embedding matrix or computes the query at serve time by running a softmax model on feature x. Once the query embedding q has been retreived, the system can then search for item embeddings V_j that are close to q in the embedding space, which is a nearest neighbors problem.

3.5 Scoring

The recommender system can then use a model to score the pool of candidates. The reason for not allowing the candidate generator score and rank the candidates with their similarity scores is that some systems rely on multiple candidate generators, and may need to use more features to accurately rank the candidates. The choice of objective function can dramatically affect the quality of recommendations, so it is important to choose wisely.

3.6 Reranking

Three factors that should be of consideration when designing a reranking model are freshness, diversity, and fairness. Keeping the model fresh can aid in making good recommendations, which can be done by rerunning training on new training data and creating an "average" user to represent new users in matrix factorization, among other solutions. To promote diversity in recommendations, a system could train multiple candidate generators using different sources, train multiple rankers using different objective functions, and rerank items based on metadata. To treat all users fairly, a system could include diverse perspectives, train models on comprehensive datasets, and make separate models for different groups, among other solutions.

4 References

Venkatachalam, M. (2019, June 22). Recurrent Neural Networks. Retrieved July 10, 2020, from https://towardsdatascience.com/recurrent-neural-networks-d4642c9bc7ce

Olah, C. (2015) Understanding LSTM Networks. Retrieved from https://colah.github.io/posts/2015-08-Understanding-LSTMs/

Recommendation Systems Overview nbsp;—nbsp; Google Developers. (n.d.). Retrieved July 15, 2020, from https://developers.google.com/machine-learning/recommendation/overview/type

Jure Leskovec; Anand Rajaraman; Jeffrey David Ullman (9 January 2020). Mining of Massive Datasets. Cambridge University Press. ISBN 978-1-108-75131-5.