AN ANALYSIS OF DEEP LEARNING AND ATTENTION MODELS FOR NATURAL LANGUAGE PROCESSING TASKS

Advait Pravin Savant
Sardar Patel Institute of Technology, Mumbai,India
adisav17@gmail.com

Publication History
Manuscript Reference No: IRJCS/RS/Vol.07/Issue09/SPCS10080
Received: 26, August 2020
Accepted: 05, September 2020
Published: 08, September 2020
DOI: https://doi.org/10.26562/irjcs.2020.v0709.001
Citation: Savant, A. P. (2020). An Analysis of Deep Learning and Attention models for Natural language Processing Moodels for Natural Language Processing Tasks.IRJCS:: International Research Journal of Computer Science, Volume VII, 227-232. https://doi.org/10.26562/irjcs.2020.v0709.001
Peer-review: Double-blind Peer-reviewed
Editor: Dr.A.Arul Lawrence Selvakumar, Chief Editor, IRJCS, AM Publications, India

Abstract: A Language is a means of communication among humans. It has a structure and is defined by the rules of grammar, which govern the constitution of sentences, clauses and words. Linguistics, the scientific study of language concerns itself with a wide variety of topics such as syntax and semantics. It concerns itself with how the words in a language are combined to form a sentence and how meaning and information are derived from the sentence based on the context. Natural Language processing as a computational discipline has the goal of getting computers to perform tasks, which involve language such as sentiment analysis, parts of speech tagging, machine translation and conversational agents. Classical NLP consisted of the symbolic paradigm, which was based on modelling grammar using theoretical computer science and using rules based on logic. Later on, statistical and probabilistic models became the standard with noisy channel models and Bayesian inference methods being prevalent. In the last two decades, the increase in computing power and the large amounts of data that is available over the internet has led to increased focus on Machine Learning. Machine Learning paradigms like Hidden Markov Models were successful. In the few years, there has been a surge in the use of Deep Learning for various tasks including NLP. Deep Learning is that subfield of Machine Learning, which creates representations of data using artificial neural networks based on computational graphs. In this paper, we will review how deep learning architectures can be used for the task of machine translation and building conversational agents.

Keywords: Recurrent neural networks; long short-term memory networks; Language models; encoder decoder architectures; attention mechanism;

I. RECURRENT NEURAL NETWORKS

Before we move to our NLP tasks, we need to understand Recurrent Neural Networks. It is assumed that the reader is aware of standard neural networks and their training with back-propagation using gradient descent/adam. In a standard neural network, we assume the individual data points to be independent and identically distributed. RNNs deal with sequential data. Input data consists of a sequence of feature vectors. Time here is an independent variable which iterates through the sequence. The corresponding output is also a sequence. Successive feature vectors in the input are temporally related and cannot be assumed to be independent. In a sequence, the output at one time step not only depends on the input at that time step but also on the inputs at previous time steps. We have a feedback mechanism with connections which form cycles. The hidden layer activations at a time step are given as input back to the hidden layer at the next time step. Such cycles in the computational graph enable us to define recurrence relations. This allows us to capture information from the history of the inputs. Let us define the computations. We have an input layer, a hidden layer and an output layer. Let matrix U denote the input to hidden layer weights, W denote the hidden to hidden recurrence relation weights and V denote the hidden to output weights. Let x_t denote the input feature vector at time step t, a_t denote the pre activations of the hidden state vector and s_t denote the post activations. o_t is the output layer pre-activations and y'_t is our predicted output. We share the parameters across time steps.

We start with s_0 and x_1.

$$a_t = Ux_t + Ws_{t-1} + b$$
$$s_t = tanh(a_t)$$
$$o_t = Vs_t + c$$
$$y'_t = softmax(o_t)$$

Such architecture can be used for many to many, many to one and one to many sequence mappings. We are constrained that the number of time steps in the output sequence is less than or equal to the number of time steps in the input sequence. Consider a multi way classification task where the output at each time step gives us the probability distribution for the different classes. The likelihood estimate denotes the probability of the data (sequence of input output pairs) given a set of parameters. At each time step, the probability for y_t will be the product of the predicted probabilities each of which is raised to the power of the true probabilities from the output data.

$$likelihood = \prod_{t=1}^{T} p(^{y_t}/_{\{x_1,\dots,x_t\}})$$

We want to factorize the joint probability distribution of the output values such the conditional probability factors capture the temporal dependencies between the variables. With its recurrence relations, an RNN provides us a computationally efficient parametrization of such a joint distribution. We maximize the logarithm of the likelihood estimate; this is equivalent to minimizing the negative log likelihood. As we take the logarithm, our loss function will be the sum of losses across all time steps with a cross entropy loss at each time step. The procedure for gradient computations and weight updates is backpropagation through time. We start with the last time step and calculate the derivatives of the loss with respect to the hidden layer post activations. For all previous time steps, we start by calculating the derivative of the loss at that time step with respect to the hidden layer post activations at that time step. We also calculate the derivatives of the hidden layer post activations at the next time step with respect the hidden layer post activations at that time step which enables us to map the derivative of the loss at greater time steps with respect to the hidden layer post activations at that time step. We then add the two derivatives together; the equations will make this clear. Notice that this follows from the multivariate chain rule. We just need to trace back the dependencies between the variables in our unrolled computational graph which represents the sequence of computations without cycles.
We know,

$$o_t = Vs_t + c$$

Let there be O output units and M hidden units. V has O *M dimensionality. Using the column picture of matrix multiplication [7], each of the hidden states will be multiplied to each of the corresponding columns and we will take the sum of those columns, add the bias vector to get the output vector. For a particular hidden state m, using the chain rule,

$$\frac{\partial L}{\partial s_m} = \sum_{i=1}^{O} \frac{\partial L}{\partial o_i} * \frac{\partial o_i}{\partial hs_m}$$

We can see that the second part of the product in the right hand side of the equation is the mth column of V and the summation value is the dot product of the gradients with respect to o and the mth column of V. To find the gradients with respect to other hidden states, we simply take a similar dot product with the corresponding column of V. We thus stack the columns of V as rows one after the other to get,

$$\nabla_{s_T} L = V^T \nabla_{o_T} L$$

Similarly, for a time-step t<T:

$$\nabla_{s_t} L = (\partial s_{t+1} | \partial s_t)^T \nabla_{s_{t+1}} L + (\partial o_t | \partial s_t)^T \nabla_{o_t} L$$
$$= W^T \nabla_{s_{t+1}} L * diag\big(1 - (s_{t+1})\big)^2 + V^T \nabla_{o_t} L$$

To calculate the gradients with respect to the parameters, we calculate such a gradient at each time step and then sum up across time steps to find the composite gradient. For example, For T time steps and M hidden neurons,
We know,

$$a_t = Ux_t + Ws_{t-1} + b$$

$$\nabla_W L = \sum_{t=1}^{T} \sum_{i=1}^{M} \frac{\partial L}{\partial s_{t,i}} * \frac{\partial s_{t,i}}{\partial a_{t,i}} * \frac{\partial a_{t,i}}{\partial W}$$

So when we are taking the derivative with respect to W, the ith row of the derivative matrix will be s_{t-1} and all other rows will be zeros as there is no dependence between those weights and $a_{t,i}$. The second term of the product is essentially the derivative of the activation. What we are getting to is that each element of the gradient vector for s_{t-1} is multiplied by the corresponding derivative with respect to the pre activation and that scalar is multiplied by the transpose of s_{t-1}.This vector is placed into the row of the derivative matrix based on the position of the element in s_{t-1}.This can be vectorized as follows,

$$\nabla_W L = \sum_{t=1}^{T}\sum_{i=1}^{M} \frac{\partial L}{\partial s_{t,i}} * \nabla_{W_t} s_{t,i}$$

$$\nabla_W L = \sum_{i=1}^{M} diag(1 - s_t{}^2) * \nabla_{s_t} L * s_{t-1}{}^T$$

We follow similar procedures for U and V, once we have the gradients, we use an iterative optimization algorithm based on updating the weights like adam/adagrad.

II. LONG SHORT TERM MEMORY NETWORKS

There is a lot of recursive computation in Recurrent Neural Networks. In the forward pass and in the flow of gradients backward, we are multiplying terms with the same matrices repeatedly. This leads to the problem of vanishing and exploding gradients. Vanishing gradients are more frequent. What happens is that the gradient values die down to very small magnitudes and we are unable to represent long-term dependencies across various time steps. We if simplify the model and study a rough analogue, Consider the recurrence relation,

$$h_t = W h_{t-1}$$

We can do an eigen-decomposition of W [7] and get,

$$W = Q \wedge Q^{-1}$$

The columns of Q are the eigen-vectors of W and $\wedge$ is a diagonal matrix where the ith position on the diagonal corresponds to the ith eigen-vector in Q. The order of the eigen-vectors in Q can be arbitrary. This factorization enables us to show,

$$W^k = Q \wedge^k Q^{-1}$$
$$h_k = Q \wedge^k Q^{-1} h_0$$
$$let\ Q^{-1} h_0 = c_0$$

x denotes column vectors which are the eigen-vectors of W.

$$h_k = [x_1 \quad \dots \quad x_n] \begin{bmatrix} \lambda_1{}^k & & \\ & .. & \\ & & \lambda_n{}^k \end{bmatrix} \begin{bmatrix} c_1 \\ : \\ c_n \end{bmatrix}$$

We use the column picture of matrix multiplication[7] to get,

$$h_k = c_1 \lambda_1{}^k x_1 + \ \dots + \ c_n \lambda_n{}^k x_n$$

We see that the eigen-values when raised to a large power will tend to decay to zero if their magnitude is less than 1 and explode if greater than 1. A similar thing happens in RNN gradient computations. The h term here will be dominated by the eigen-vector corresponding to the highest eigen-value. We want to create paths through time where the derivative neither vanishes nor explodes. We can do that will linear self-connections. If s'_t is the candidate for the state at time step t based on our computations, we define the true state vector s_t as follows,

$$s_t = \alpha * s_{t-1} + (1-\alpha) * s'_t$$

When we take the derivative of s_t with respect to s_{t-1}, we get the α parameter added with other terms. An α value near one ensures that the information of the past states can be remembered. In this way using self-connections, we can create paths through time, which allow the gradients to flow for long durations. What we can also do is to make these self-connections conditioned on the context by making dependant on another hidden unit with its own set of parameters, which can be learned via backpropagation. Using this approach, the timescale of integration of past information can be adjusted dynamically. We introduce a modelling choice that enables us to selectively read from the output of the state at the previous time-step, selectively forget information from the previous state and selectively write the output from the current state to the next state. As we shall see, this is done via gates. There is a unit (a vector of neurons) for the input gate (selective reading), one for the forget gate (selective forgetting) and one for the output gate (selective writing). The hidden state and these units together form a cell. These cells are connected recurrently to one another across time steps.

Let h_t denote the output from the cell at time step t to the next cell, let s'_t denote the candidate for the state of the cell, let s_t denote the true final state of the cell, let i denote the input gate vector, o denote the output gate vector and f denote the forget gate vector. We take the sigmoidal activation function for the gates. Here the * symbol denotes element wise multiplication.

We have,

$$i_t = \sigma(W_i h_{t-1} + U_i x_t + b_i)$$
$$f_t = \sigma(W_f h_{t-1} + U_f x_t + b_f)$$
$$o_t = \sigma(W_o h_{t-1} + U_o x_t + b_o)$$
$$s'_t = tanh(W h_{t-1} + U x_t + b)$$
$$s_t = f_t * s_{t-1} + i_t * s'_t$$
$$h_t = \tanh(s_t) * o_t$$

We extend the backpropagation algorithm for gradient computations and subsequent training [9]. LSTMs enable us to model long term dependencies. For example, the memory of the first input persists across the cells when the forget gate is open (sigmoid values close to 1) and the input gate is closed (sigmoid values close to zero.)

III. WORD EMBEDDING

In order to feed the words into an RNN, we need to create a vectorial representation of words. The naïve way of doing this is one hot encoding. We need a more compact representation of words which can capture the semantics and the similarities between the words. We can train a neural network which takes one hot vectors of size V(size of the vocabulary) as input, has a hidden layer of size D where D is much less than V, has an output layer of size V. The output consists of one hot vectors. We group together pairs of words where one word appears in the context of the other using our corpus. Given the context word, the neural network tries to predict the target word. As the input and output are one hot vectors, the matrix multiplication is simplified. The output at the target word position is essentially the dot product of a column in the input to hidden layer matrix corresponding to the input word and a row corresponding to the output word in the hidden to output layer matrix. Other positions in the output have zero as their target label. We take a softmax over the output and use the cross entropy loss. Since the softmax over the vocabulary is computationally expensive, what we do in practice is take the sigmoid of the dot product and train it for true context word, target word pairs and false context word, target word pairs. What happens after training is that in each of the two matrices, for all the words in the vocabulary of size V, we obtain a size D(D<<V) representation of the word which captures its semantics in the language[2].

IV. LANGUAGE MODELS

A language model is able to give us a joint probability of a sequence of words in a sentence. Such a computation requires it to capture the structure of a language. When we use an RNN to train a language model, we start with a token which denotes a start of the sentence. There is also a token, which denotes the end of a sentence. These have vector representations. We iterate over the sentences in our corpus. After initializing the RNN, we feed the start token and try to predict the first word as our output. The RNN predicts a probability distribution over words using softmax at the output. We then feed the true first word into the input of the RNN and try to predict the second word. Then we feed the true second word as input into the RNN and try to predict the third word. This feeding of the output back into the RNN is called teacher forcing. At the last step, we try to predict the end of sentence token. We train the model for all our sentences using the cross entropy loss and after this process obtain an RNN with parameters tuned for language modelling. This makes sense because the joint probability of the sentence is factor in our RNN as follows,

$$p(y_1, \dots, y_T) = p(y_1) * \prod_{t=2}^{T} p({}^{y_t}/_{y_{t-1}}, \dots, y_1)$$

This takes into consideration the temporal nature of the sequence of words. In order to sample a sentence from our trained language model RNN, we give in the start of sequence token at the first time step and predict the probability distribution for the first word. We then choose the first word based on that distribution (we can choose the word with the highest probability or make a choice based on that distribution). We then use the vector representation of that word as input to the RNN in the second time step in order to predict the probability distribution over the second word. We choose the second word and feed its vector representation into the RNN as input for the next time step. We repeat this procedure until the end of sequence token is produced.

V. ENCODER DECODER ARCHITECTURES

Representations are an integral part of deep learning. They have an important meaning even in neuroscience. In a standard RNN, we are constrained that the output sequence size is less than or equal to the input sequence, they are of the same length or it as a many to one mapping.

In tasks such as machine translation, the output sequence and input sequence can be of any length. In the encoder decoder architecture, we have two RNNs, one encoder RNN that reads the input and produces a representation of that input and one decoder RNN which produces the output based on that representation.

In machine translation, the encoder RNN reads the input sentence, we do not have the output, we move through the hidden states across time steps as we process the sentence and the final hidden state is used as a context/representation of the sentence. The decoder RNN is conditioned on the context to produce the output sequence. What we mean is that the context is set as the first hidden state for the decoder RNN and then it is used as a language model to produce the translation just as we saw before [1]. We had used teacher forcing, we can even feed in the context at each stage of the decoder RNN. We have the target sentence and its vector representations in that language as output. The labels are one hot encoded based on the vocabulary of that language. We will have two vocabularies, one for the input language and one for the output language. We train the composite system using backpropagation just as we had done earlier. The gradient computations are scalable now that we understand the forward pass.

VI. BI DIRECTIONAL RNNS

In the RNNs which we have discussed, the causal structure is supposed to be that the current output at a particular time step depends on the output at the current and previous time steps. In certain language modelling applications, the current output at a time step may also depend upon the input at future time steps. Our RNNs are not capturing that dependency. What we do in bidirectional RNNs is that we have two separate hidden state vectors. One of them is a forward RNN and one in a backward RNN. Each of these will have separate weight matrices with the input and separate weight matrices with the output. In the forward RNN, the computations are similar to what we have had earlier. We start with the first time step and move forward in the forward pass. Computations in the backward RNN start from the last time step. In the forward pass for the backward RNN, we start from the last time step and computations move in the reverse direction. After the forward pass, the output is given by combining these two separate hidden state vectors by adding the product of the weight corresponding matrices and the hidden state vectors. In the backward pass, for the forward RNN, we perform backpropagation through time just as we had earlier. For the second RNN, the backpropagation through time starts at the first time step and then the gradients flow towards the last time step. We use adam/adagrad for updating the weights. If it is a many to one mapping, we take the concatenation last hidden state of the forward RNN and the first hidden state of the backward RNN in order to generate the output.

VII. ATTENTION MECHANISM

Consider machine translation. The accuracy for machine translation is measured by the BLEU score [5]. What happens with Encoder Decoder Architectures for long sentences is that the accuracy decreases with increasing sentence length. It is noticed that the single context is a bit small to properly summarize the whole sequence. As far as machine translation is concerned, we humans do not practically read the whole sentence, memorize it and then generate the translation from scratch. We focus on certain important parts of the translation in each time step. We generate a part of the translation and then move to the next part. With the attention based model, we have an encoder bi directional RNN whose hidden state vectors, the concatenation of the states of forward and backward RNNs, are used as feature vectors for producing the context for the decoder RNN. As the decoder RNN generates one word at a time, at each time step there are attention weights which tell it that when it is generating the t^{th} output word, how much attention should be payed to the t'^{th} input word. At each time step for the decoder, we are taking a weighted sum of the feature vectors from the encoder RNN (the weights are the attention weights) and using this sum as the context at that time step. In training for a sentence to sentence mapping, there will be Tx*Ty attention weights where Tx is the number of input time steps and Ty is the number of output time steps. For each decoder time step, the sum of the attention weights for all input RNN time steps is taking to be one as we model it probabilistically. The idea is to learn these attention weights from the data. Based on our constraint for the attention weights(α), we model them as the result of a softmax operation based on a quantity g(t,t') defined for each input time step t' and output time step t.

$$\alpha(t,t') = \frac{\exp(g(t,t'))}{\sum_{t'=1}^{T_x} \exp(g(t,t'))}$$

We must now decide how to find the attention weights. It is reasonable to assume that the attention payed at each time step depends not only on the encoder feature vector at that time step but also on where we are in the output sequence. What we do to find the attention weights at output time step t is we concatenate the hidden state at output time step t-1 along with the feature vector at input time step t and use this in a single layer neural network to find g(t,t'). Once we find the g values for all input time steps at that particular output time step, we can take the softmax to find the attention weights. The important point is that this neural network to find the g values is trainable via backpropagation.

The context goes as input to the state of the decoder RNN and then we predict the next output word at that time step. One thing is that as we feed the context as input for the decoder RNN, we can use teacher forcing while training and concatenate the true output at the previous time step with the context at that time step as input to the decoder RNN at that time step.

When we use these attention models and trust backpropagation, we obtain high accuracy machine to machine translation results [4][3]. We can bi directional LSTMs instead of RNNs as we have described earlier. We make use of what the LSTM cell gives as output at each time step and ignore the cell state. For conversational agents, we follow the same approach, we train the input RNN based on the sentences from one speaker and the output RNN with the corresponding next sentence by the second speaker. After training across a large number of sentences, for a new input sentence, we can use the decoder generatively as a language model. After training, while we are doing a translation, it is observed that for a word at a particular output time step, the input words which are important in generating that word show higher attention weight values. This justifies our modelling choice.

VIII. CONCLUSION

The aim of this paper was to conceptually analyse and review the use of attention models in Natural Language Processing tasks. We discussed RNNs, LSTMs, Word Embedding, Bi Directional RNNs, language models, encoder decoder architectures and finally the attention model. Notice that we did not discuss linguistics. As a matter of computational linguistics, machine translation is a complicated task [8]. The beauty of deep learning is that with enough data, the model can learn the various abstractions, concepts and structures involved in language by created useful parameterized representations. It is indeed remarkable how backpropagation can be scaled to various architectures and the same architecture can be used for various tasks. Connectionist frameworks of learning and intelligence are working. The field of machine learning is fast evolving and the future of deep learning is promising.

REFERENCES

1. Cho, K., Van Merriënboer, B., Bahdanau, D. and Bengio, Y., 2014. On the properties of neural machine translation: Encoder-decoder approaches. arXiv preprint arXiv:1409.1259.
2. Distributed Representations of Words and Phrases and their Compositionality. Tomas Mikolov et al, NIPS, 2013.
3. Luong, M.T., Pham, H. and Manning, C.D., 2015. Effective approaches to attention-based neural machine translation. arXiv preprint arXiv:1508.04025.
4. D. Bahdanau, K. Cho, and Y. Bengio. 2015. Neural machine translation by jointly learning to align and translate. In ICLR.
5. BLEU: a Method for Automatic Evaluation of Machine Translation Kishore Papineni et al, IBM,TJ Watson research centre ,2002.
6. Goodfellow, Bengio and Courville, Deep Learning, MIT press,2016.
7. Gilbert Strang, Linear Algebra and its applications, MIT press,2006.
8. Daniel Jurafsky and James H.Martin ,Speech and Language Processing, Pearson Education, 2009.
9. Alex Graves, Supervised Sequence Labelling with Recurrent Neural Networks, Springer, 2012.