TÜBİTAK Internship Report

First of all, before going into internship report, let me introduce myself and tell a bit about how the internship was and the thoughts about the community there.

My name is Akif Faruk NANE studying at Bosporus University as a sophomore student. I think I have a really good background of computer science comparing to my peers. I always want to do crazy things and have been doing so whole my life. Began the adventure with mathematics and shifted to the competitive programming, -you know- one always should find the best optimal solution of a problem. Beside those, I also studied C# in high school, starting from 7th grade that now it's been 7-8 years. Subsequently I got interested in Artificial Intelligence specifically deep learning. My internship here was also on deep learning. Lately, I have been working on writing heavily optimized numeric library with only C#, since .Net Core supports Simd instructions and caught C++ in terms of performance. I hope I can find someone who knows about these topics in Tübitak as well.

Let's get into the internship. I really liked the atmosphere and the environment in Bilgem. I've met many people and luckily and especially the one whose name is Kadir working at Hisar lab who guided and some kind of mentored me. I had a few good conversations with him talking for hours how we can achieve better language models and about humans and the business life etc. However, -beside the pros- the cons were the waking time which was really hard for me, I got accustomed after a week or so and I'm not sure now if working at Bilgem makes a great deal of benefits for me. I'm kind of a person who learns alone and whose interests could differ in time easily. Also, I'd like to mention that I enjoy developing technologies that others can utilize and create products. So, I could quite say I don't like working on the projects that are closer to the customer side, which requires nonscientific effort a lot.

Okay, now let's get into the real stuff.

Akif Faruk NANE

Language models

Language models are used to predict the next upcoming word based on what's written before and also score the compatibility of the words, in other words the likelihood probability of the sentence being created. The simplest model is N-gram models. N-gram is a sequence of N words creating a phrase. To illustrate, 2-gram is a sequence of 2 words like 'Good morning', or for 3-gram, it might be 'Studying computer science'. We use N-grams to create a language model by assigning probabilities to each word.

N-gram

To determine the probability of a sentence created, first the probability of the words in the sentence should be determined. If the probability of words is known, it is easy to find the probability of a sentence by simply multiplying all probabilities of given words. Let's denote the probability of a word in a context of H as $P(W \mid H)$. It is simply calculated by using the count of phrases in the training data. The count of the phrase H combined with the word W divided by the count of the phrase H.

$$P(W \mid H) = C(W,H) \,/\, C(H)$$

$C(X)$ specifies the count of the phrase X in the training data. For a sentence consisting of words $(w_1, w_2, \ldots., w_n)$, the probability of word w_n is $C(w_1, w_2, \ldots, w_n) \,/\, \mathrm{C}(w_1, w_2, \ldots, w_{n-1})$. So, it might be said the probability of 'Have a good day' is,

$$C(\text{'}Have\ a\ good\ day\text{'}) \,/\, C(\text{'}Have\ a\ good\text{'})$$

For a training set ($\text{'}Have\ a\ good\ day\text{'}, \text{'}Have\ a\ good\ time\text{'}, \text{'}Have\ a\ good\ breakfast\text{'}$) its probability is $1/3$. Now, let's prove that the probability of a sentence equals the multiplication of the probability of words.

$$P(w_1, w_2, \ldots., w_n) = P(w_1)\, P(w_2 \mid w_1)\, P(w_3 \mid w_1, w_2) \ldots.. P(w_n \mid w_1, \ldots., w_{n-1})$$

- ⇨ $P(w_1) = C(w_1) \,/\, C(\,)$
- ⇨ $P(w_2 \mid w_1) = C(w_1, w_2) \,/\, C(w_1)$
- ⇨ $P(w_3 \mid w_1, w_2) = C(w_1, w_2, w_3) \,/\, C(w_1, w_2)$
- ⇨ .
- ⇨ $P(w_n \mid w_1, \ldots., w_{n-1}) = C(\,w_1, \ldots., w_n) \,/\, C(\,w_1, \ldots., w_{n-1})$

Multiplying all the expressions, we get $C(\,w_1, \ldots., w_n) \,/\, C(\,)$ meaning the count of W divided by the count of all sentences in training set.

For N-grams we generally calculate the likelihood probability. Probability of a word w_i is $P(w_i \mid H)$ where H is considered to corresponding last $n-1$ words. Thus, the probability of a word turns into $P(w_i \mid w_{i-n+1}, w_{i-n+2}, \ldots, w_{i-1})$.

Why consider only last $n-1$ words to calculate the probability of n_{th} word? As everyone knows well that language is a creative tool that humans use. So, what if someone creates a sentence that never been created but likely to be created in some context H very meaningfully? Do we say, the sentence does not show up in the training set, so the probability must be zero? It would be not correct. Instead of finding the whole phrase in the training set, we look for smaller meaningful phrases that might more likely to be in the training set. By combining the probabilities of words, we calculate the probability of the whole sentence. This is why n-gram models are used. It is not logical to search one big phrase in data. No

matter how big data is, there will always be unseen sentences. Maybe right now I am writing one of those, who knows…

Let's name it as **zero situation** if we don't have the phrase that we search for in our data, which means $C(W) = 0$. There are smoothing techniques to overcome zero situations. However, we will not give point to those.

If you are to analyze the number $\boldsymbol{n}$ in N gram, what should $\boldsymbol{n}$ be, what if $\boldsymbol{n}$ is a big number or a small one, what would you say?

The less n becomes:

- ⇨ The less data is required. Because there are less combinations of words comparing to greater $\boldsymbol{n}$s.
- ⇨ The less zero situation there might be.
- ⇨ The less meaningful, but the more creative sentences could be predicted or estimated. The semantic meanings and the combinations are correct and likely trustable for those in training sets created by human beings. The shorter the phrases becomes in the training set, the smaller meanings they hold. Forming new sentences with those combinations might not be meaningful sometimes. This shows that they are more creative when n is smaller.

Perplexity

Perplexity is a metric likely considered to be the average error score of each word. So, this expression basically calculates a geometric average in order to enable us to comparing probabilities of sentences.

$$PP(w) = \sqrt[L]{\prod_{i=0}^{L} \frac{1}{p(w_i|w_1 \cdot \cdots w_{i-1})}}$$

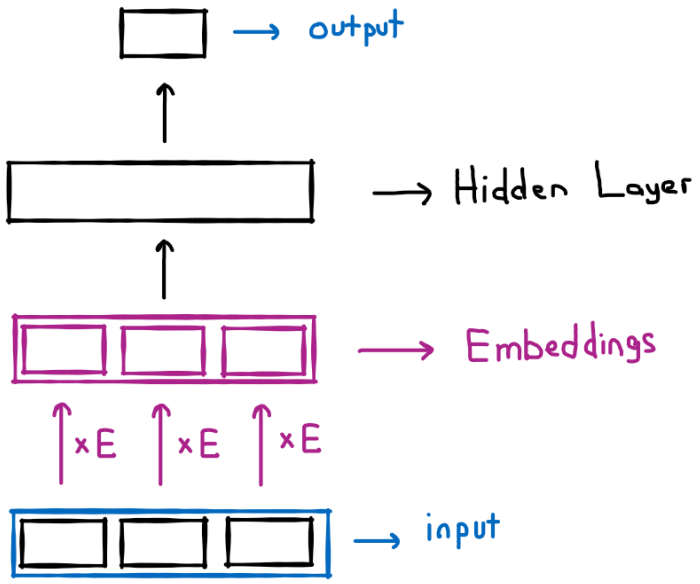
The n-gram version would be:

$$PP(w) = \sqrt[L]{\prod_{i=0}^{L} \frac{1}{p(w_i|w_{i-n+1} \cdot \cdots w_{i-1})}}$$

Feed Forward Neural Language Model

The simplest n-gram model with neural networks would be a feed forward neural LM. The model takes $n-1$ words as input and gives the probabilistic distribution of what the next word should be as output.

The scheme of an example model is shown below.



The words at the input are passed to the network as **one-hot vectors**. The one-hot vector of w_i is an array whose element i is one and the rest are zeros. Assume the **dictionary size is T**. The input consists of 3 words in the example which corresponds to $3T$ numbers in one-hot coded state.

Each one-hot vector is passed to up-forward layer being multiplied by matrix **E** which is the abbreviation of **Embedding Matrix**. It can be said that embedding matrices are word to vector converters. It simply takes a one-hot vector and converts it into a meaningful another vector. By doing so, it allows us to feed the meaning of the words into the network.

The output has T numbers every of which are between zero and one and each one corresponds the probability of being the next word predicted.

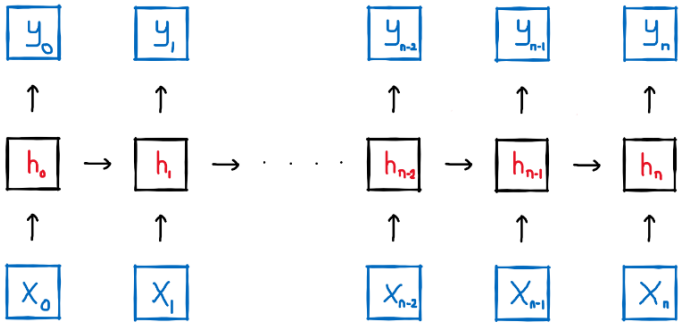
There are many ways to create Embedding Matrices. Pretrained embeddings can be used in neural networks. It can also be trained during the training process as well. Needless to say, it's up to you whether the Embedding Matrix should be fixed, trainable or semi-trainable. You can mix the previous meanings in the Embedding Matrix with the new meanings required for the problem that you are dealing with.

Recurrent Neural Networks

Recurrent neural networks are created to process a sequential data. By sequential, it means that the input we give at each time step has relation with past and future values. So, the data is dependent on previous data, or upcoming data and the model is expected to be able to give correct output by inferring the relation between the inputs.

In designing neural networks, if you can find the correct answer with certain data access limitations by yourself, there is a chance that model also can give the answer correctly. However, don't forget that if you can't do so by yourself with data access limitations meaning for example if it's not RNN, you are probably not able to look backward data, the model also will not be able to give the correct answer as well. All in all, consider what is to be accessible by model and what to expect from model to do.

The simple RNN model is shown below (x - input, y - output).



The data flows between hidden states by being multiplied every time by the matrix W_h which does not change over (sequential) time.

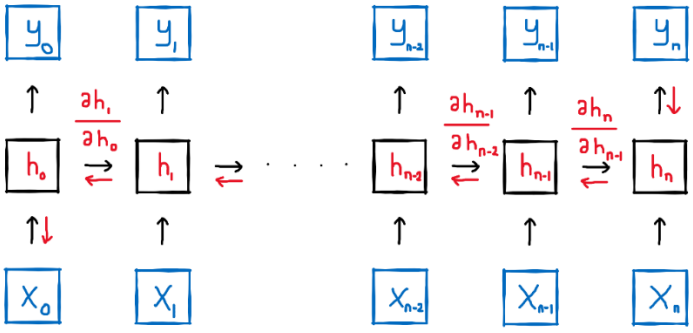
$$h_t = W_h \times h_{t-1} + S(W_x \times x_t + b_x)$$

(S: activation function)

So, this is a simple RNN that takes and processes an input, mingles it with previous hidden state. Thus, it creates a new hidden state that has information from both previous inputs and the current input.

Vanishing - Exploding Gradients

Backpropagation through time uses chain multiplications to calculate derivative of every output y_t with respect to each trainable parameter in the model. In training parameters according to y_n, as you see below, we many times calculate $\frac{\partial h_t}{\partial h_{t-1}} \cong W_h$ roughly. I am not going into matrix derivatives, let's just say W_h instead. While finding the gradients, as we go backward through time, the gradients are multiplied by W_h.



$$\frac{\partial h_n}{\partial h_0} \cong (W_h)^n$$

According to the formula above, the gradients are too sensitive to W_h matrix. In case $W_h > 1$, the gradients explode, otherwise $W_h < 1$ vanish.

Overcoming Vanishing - Exploding Gradients

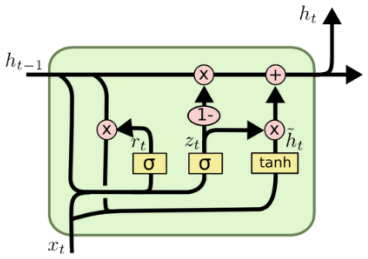
Before how to overcome Vanishing – Exploding Gradient problem, I'd like to press a point related with the topic.

There are many activation functions that we use when building a model. Some of them decrease gradients. Let's compare the derivatives of x and $S(x)$. The derivatives of x: $\frac{\partial x}{\partial a}$ and of $S(x)$: $\frac{\partial S(x)}{\partial x}\ \frac{\partial x}{\partial a}$. Divide both terms by $\frac{\partial x}{\partial a}$, we get 1 and $\frac{\partial S(x)}{\partial x}$. Which one is greater? For sigmoid, tanh, SoftMax etc. $\frac{\partial S(x)}{\partial x} < 1$. On the other hand, for ReLU, $\frac{\partial S(x)}{\partial x} = 1$. Thus, for ReLU function, gradients don't vanish and remain the same.

So, ReLU might be a good choice in order to overcome vanishing gradients problem. The second solution is determining W_h in a way that gradients remain the same. Initializing W_h as identity matrix could work, because we want it to be ineffective element in matrix multiplication. So, the term being multiplied by W_h will remain the same: $A \times W_h = A$. The third option is using LSTM - GRU cells.

GRU

The basic idea of why GRU and LSTM cells work is that they allow model to determine how much they need to forget at each time step. So, vanishing gradient might still happen, however it is controlled by model itself. Instead of updating hidden states by multiplying fixed W_h for each time step, it is multiplied by the term (z_t in GRU) which is not fixed and changes from time to time. $\widetilde{h_t}$ new information extracted from fusion of x_t and h_{t-1} is multiplied by z_t. z_t decides how much to take from and forget $\widetilde{h_t}$.

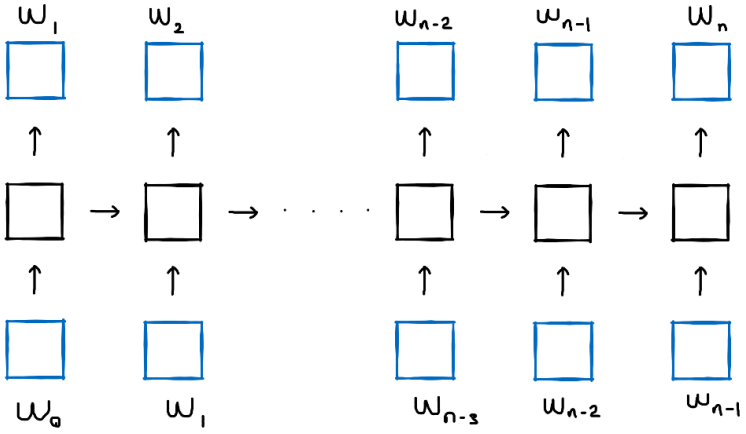


$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$
$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$
$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$
$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

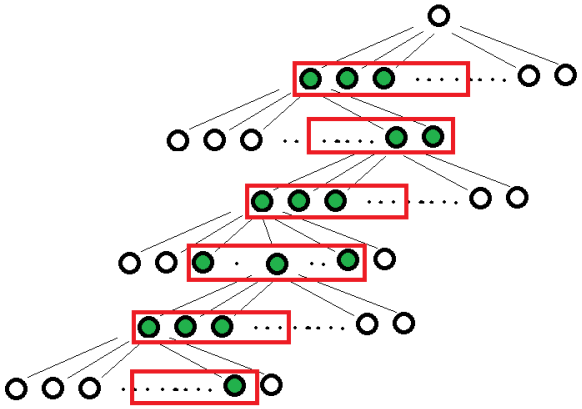
(https://medium.com/mlrecipies/deep-learning-basics-gated-recurrent-unit-gru-1d8e9fae7280)

RNN Language Model

RNN Language model is an RNN model where we give a word and get a word at each time step. It takes first word, give us what the second word should be. Then, we get that output, give it as input at next time step like in the picture below. We do this process until we encounter the stop-word. Remember that words are passed to network as one-hot vectors.



RNN model can be used to determine the probability of a sentence and create a sentence from a beginning word. While creating a sentence, how do we decide which word is going to be the next word? At each time step, we get probabilities of each word. Choosing the highest probability as the next word might be a solution. However, sometimes it might not be a good solution. There could be situations such as you choose the highest probability, but you don't find a stop-word or there would be a better sentence, if you chose the second highest probability. As you might guess, a spontaneous probability tree emerges which needs to be traveled. Our aim is to travel all possibilities that end with stop-word. However, it's not practical to travel all nodes and it's impossible because it goes forever. So, we prefer to travel the highest k nodes in BFS order, assuming that those who are not belong to k nodes are probably not good solutions.



Akif Faruk NANE