**Recurrent Neural Networks**

Stanford University

CS224d: Deep Learning for NLP

Richard Socher

In theory, all the vectors in the class should be column vectors

**Language Models**

A language model computes a probability for a sequence of words:

• Useful for machine translation

만약 나의 machine translation 모델이 다음과 같이 프랑스 한 문장을 보고 2개의 영어 후보 문장을 제시한다면?

If we have a good probability estimation,

- Word ordering:

p(the cat is mall) > p(small the is cat)

sequence 측면에서 첫 번째 문장이 더 적합하므로, LM에 의한 해당 문장의 확률도 높다

- Word choice:

p(walking home after school) > p(walking house after school)

프랑스어의 어떤 단어가 영어의 어떤 단어로 잘 바뀌었는지 확인.

둘 다 같은 뜻이지만, 여기서는 house보다 home이 더 적합하다.

사람들은 쉽게 알맞은 단어나 단어 순서들을 알아맞힐 수 있지만, but to machine translation model just co-occurred and counted how often do I see this word next to that word versus not it might be less obvious. So, it’s useful if we have basically good probability estimates for sequences.

Similarly, when you have speech recognition, wood(나무)와 would(조동사) 인 경우, they sound very similar so, the language model might help you or tell you which of wood is more likely given the sequence of other words in front of it.

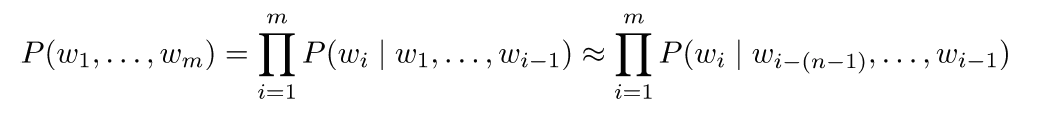
**Traditional Language Models**

Generally, it’s approximated because it’s very hard to compute the entire probability distribution given all possible sequences of the English language or other languages. So, it’s necessary to use markov assumption in traditional language model even if it’s obviously incorrect.

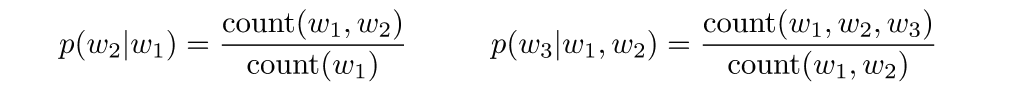
Using markov assumption, instead of predicting the next word based on the entire sequence of words before (we would have to keep the very large table), we basically just look at n words before the next word we would like to predict. So, this is an approximation but it’s obviously not a true one (Actually, the next word of a sentence will always depend on a lot of other words).

• Probability is usually conditioned on window of n previous words

• An incorrect but necessary Markov assumption! (Markov 가정: 현재 state는 과거, 미래 state로부터 독립적이다. 즉, 순서 정보를 고려하지 않는다.)



• To estimate probabilities, compute for unigrams and bigrams (conditioning on one/two previous word(s):



첫 번째는 window size가 1일 때이다.

In this case, it’s just basically looking at uni-gram probabilities. We only look at the counts of how often do we see the sequence and how often do we see by itself. We can basically go through all possible words that we have in English language and we create a very large table where we look at how often do we see the words in this sequence versus how often do we see . We should compute that very very large table. Actually, that will not be very accurate estimate (the로 예를 들면 the 다음에 수많은 단어가 나열되기 때문이다.)

두 번째는 window size가 2일 때이다.

It’s extended version using bi-gram and tri-gram. Now we collect all the counts for every possible sequence of tri-grams in the entire corpus.

In general, we want to have not just conditioning on bi-grams but ideally try 4 or 5-grams.

• Performance improves with keeping around higher n-grams counts and doing smoothing and so-called backoff (e.g., if 4-gram not found, try 3-gram, etc)

Eventually, we want to not just condition on the 4-grams because maybe there is no 4-grams that you’ve ever seen at test time. So, you might want to do so called backoff very smoothly distributions based on 4-grams, 3-grams, 2-grams, uni-grams.

• But, the problem is there are a lot of n-grams!!!

→ Gigantic RAM requirements!

• Recent state of the art: Scalable Modified Kneser-Ney Language Model Estimation by Heafield et al.: “Using one machine with 140 GB RAM for 2.8 days, we built an unpruned model on 126 billion tokens”

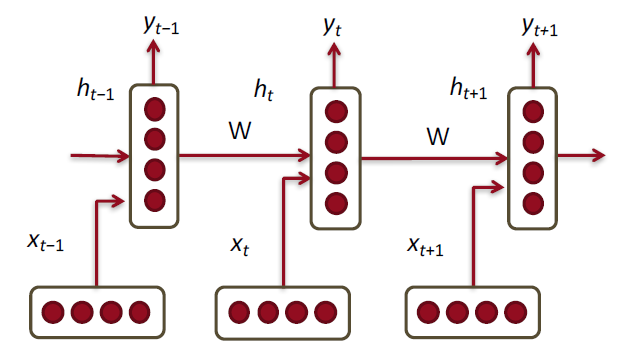
Kneser-Ney is just a way to combine the probabilities of all from the n-grams, n-1, n-2, … , to 2-grams. The best thing you could do is that after a lot of very low level optimization was to have one machine with 140 GB RAM trained for 2.8 days and built a single model on 126 billion tokens. That’s basically a huge requirement. You could never run that kind of model on your cell phone.

**Recurrent Neural Networks!**

In RNN, in theory at least, we will hope that every hidden state here captures all the information from the past. Of course in practice, that isn’t really the case of rule to find ways of making that a better approximation and more true. But, at least in theory we will hope that the hidden states that we compute every time step in the recurrent neural network will capture a lot of history that we’ve seen before.

The main reason we call the RNNs is actually to tie the weights at each time step. So, there’s no bound of how many time step we could compute these hidden states.

The nice thing here is the RAM requirement only scales the number of words (maybe 100). Instead of having to keep around the number of words to the power of 345, it’s only (if we have 100 dim word vector) (100)\*(# of words) + a couple of the weights + output (=vocabulary size).



• RNNs tie the weights at each time step

• Condition the neural network on all previous words

• RAM requirement only scales with number of words

**Recurrent Neural Network language model**

We walk through a couple of different visualizations and straight-up definition of RNNs.

We essentially assume that every time step we have a word vector that we compute.



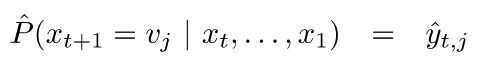
In hidden state, there is a simple linear neural network where we basically have hidden connections from the previous time step and the hidden visible connections or hidden input connections with .

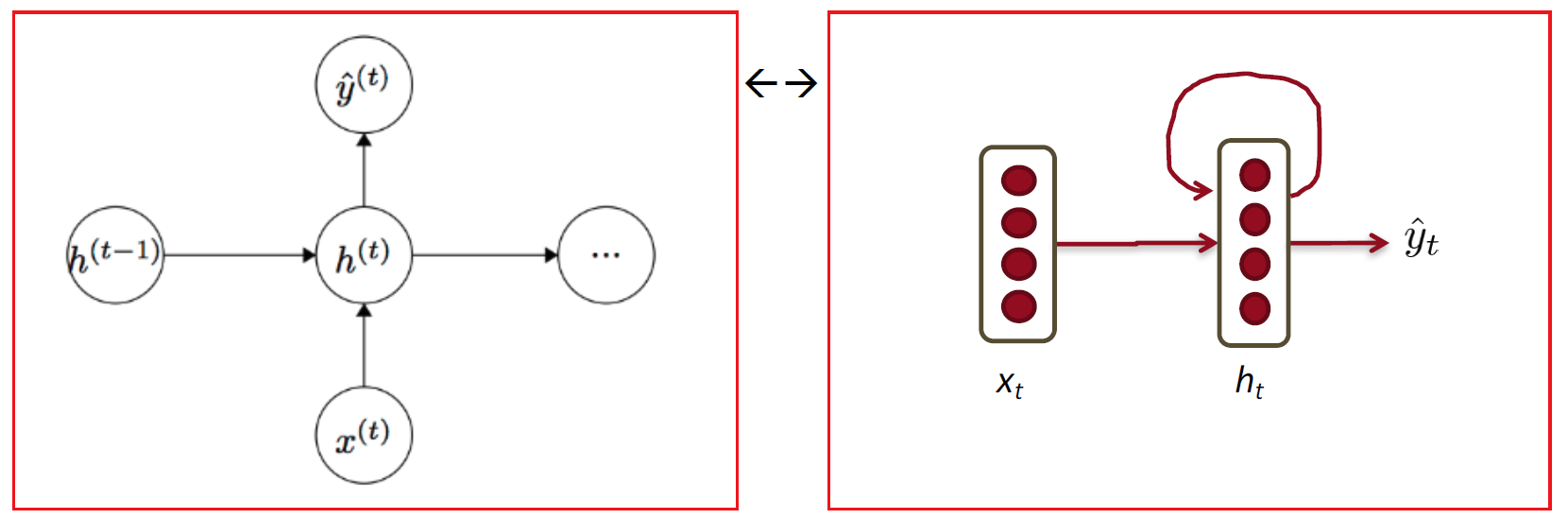


And the softmax is again just a simple classifier where we can predict any class that we want at each time step. The class could be the case of language models the next word and the class also be the sentiment of the sentence so far and the class also be the label of that word at current time step. There lots of different things you could train on this. That’s why it’s powerful and general model.



Default case for RNN language model will just be the next word that we’re trying to predict at each time step.

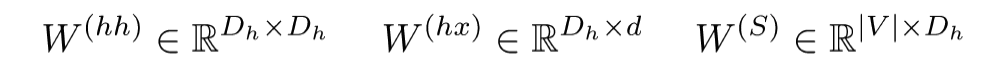




The main idea is to use the same set of W weights at all time steps. We will run into trouble (vanishing gradient problem) with this but it makes nice because that way we can essentially extend this model to arbitrary length sequences.

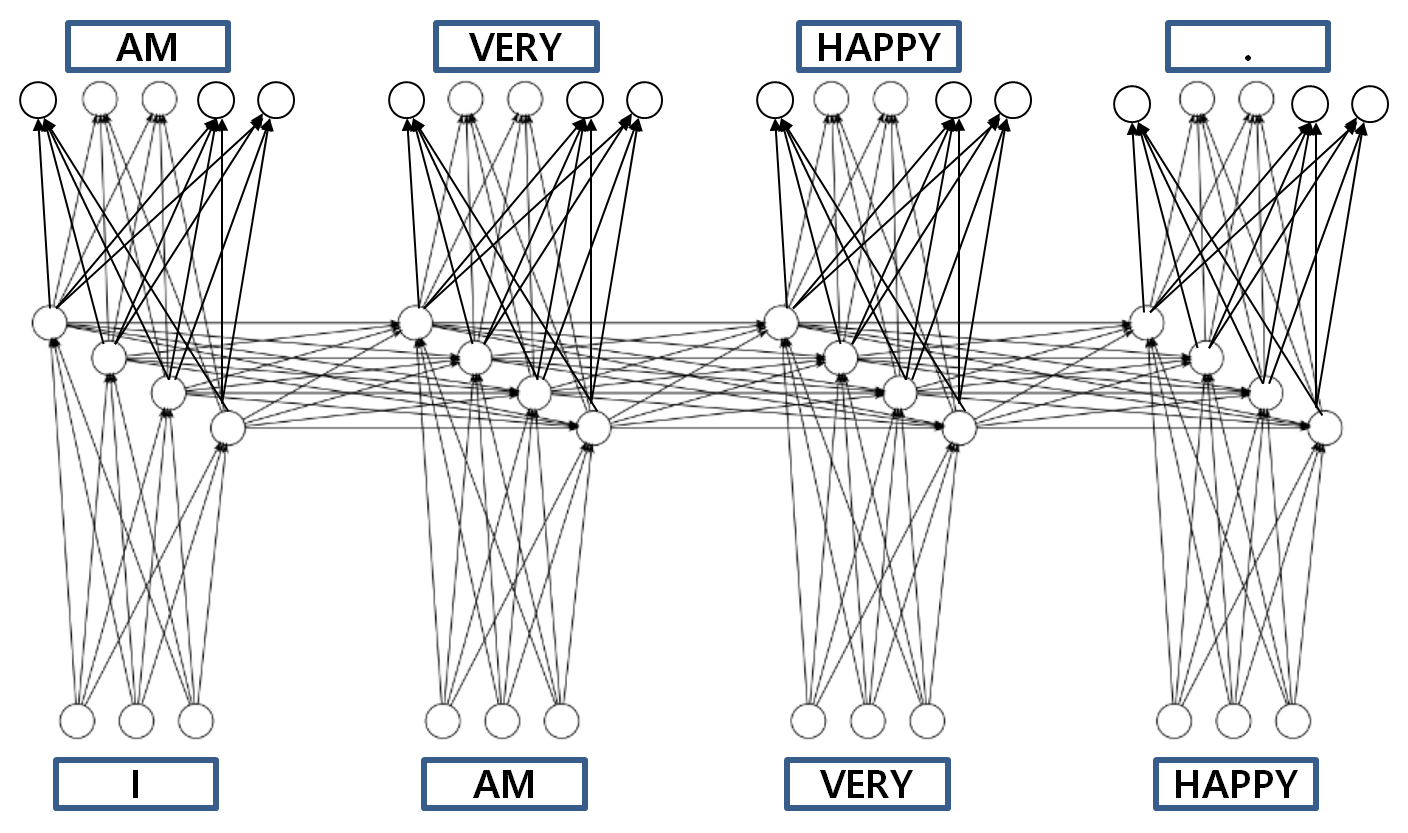
It’s very important to define the proper dimensions

is some initialization vector for the hidden layer at time step 0. is a dimensionality of hidden states.



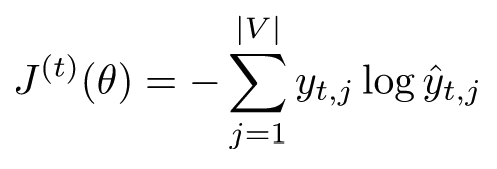
* d : dimension of word vector
* |V| : size of vocabulary (So, will be a very large matrix if we have a lot of words)

예를 들어, I am very happy. 라는 문장을 RNN모델에 학습을 시켜보자. 학습시키는 과정은 다음과 같고, 각각에 해당하는 Dimension은 아래와 같다.



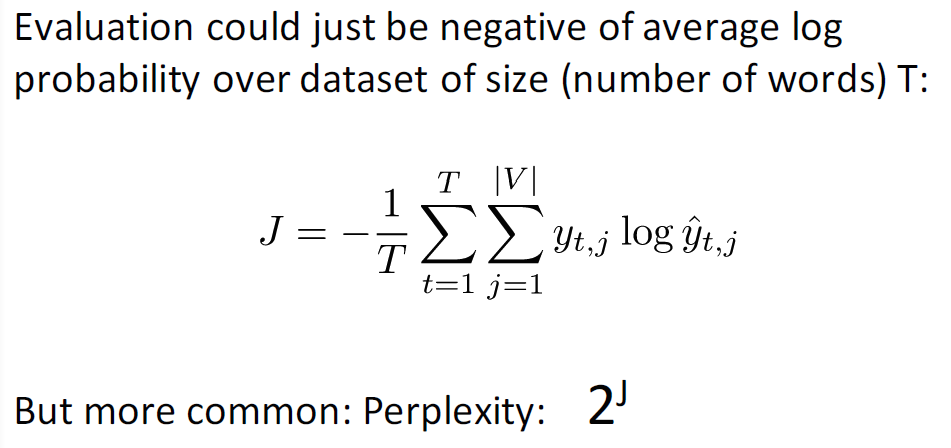
|  |  |
| --- | --- |
|  |  |

We want to predict that the probability distribution (=) over the vocabulary at each steps. And then we can compute the same cross entropy loss function to predict words instead of classes.



* Sum over all classes = sum over the vocabulary indices
* 를 ground truth라고 생각할 수 있다. 어차피 1만 남으니까.

A lot of people instead of taking this J as the final evaluation use perplexity because training is very similar to testing. We want to just push the probability to be the right distribution (only 0 and 1) So, intuitively, the less perplexed you are the better the model is.



**Training RNNs is hard**

The main problem with RNNs is that while in theory, they should capture all the information on the previous time step, the fact of matter is that this actually is really hard to be trained. The main problem comes also from its main strength of just like multiplying the same matrix at each time step during forward propagation. We’re asking a simple linear transformation plus nonlinearities to change state a lot based on any single word that’s coming in. And we’re hoping that inputs from many many times ago can still change it despite applying the same transformation at each time step.

• Multiply the same matrix at each time step during forward prop

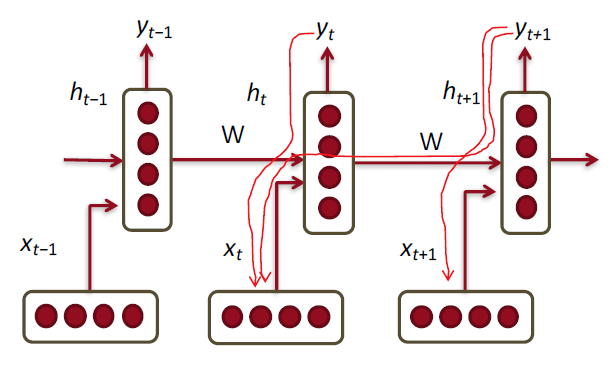
• Ideally inputs from many time steps ago can modify output y

• Take for an example RNN with 2 time steps! Insightful!

**The vanishing/exploding gradient problem**

Let’s look at the actual problems that we will get when we try to train this. In fact we have both vanishing and exploding gradient problems.

Using vanilla, optimizing it (stochastic gradient descent), there sudden some point you might get not a number. (즉, 에러값이 매우 작아진다.) finite difference gradient check하면 bug가 아닌 것을 알 수 있다. The vanishing gradient problem is a little more subtle and you might think it’s doing what you want to do but it really can’t



At training time, we get delta error signal that we add during back-propagate every time step. will propagate a delta into and will also propagate one into further time step. do that again. And it will basically keep on going into all the past time steps (until time 0 where the sequence actually start) Again it comes from chain rule.

**The vanishing gradient problem – Details**

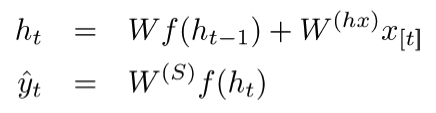
Let’s see what actually happens when we apply the chain rule to potentially very long sequence of time steps.

Let’s define RNN in slightly differently

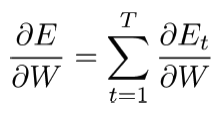
Chain rule을 직접 보여주기 위해 함수형 방식으로 표현. (softmax함수는 제외)

원래 h-t의 출력값이 sigmoid함수 출력값인데, 여기서는 h-t출력값을 f(h-t)로 표현. 함수 (어차피 함수 출력값이기 때문에)

• Similar but simpler RNN formulation:



• Total error is the sum of each error at time steps t

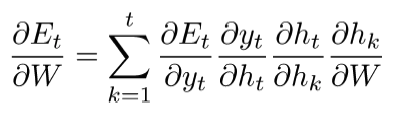


Essentially we have lots of little logistic regression problems at each time step but now they’re all underneath it to get to the hidden states that we use for as input to softmax. It happens they’re all connected so, it’s chain, a long computational graph in times for a language.

Let’s look at the element at each time step T. There will now depend on all previous time steps.

• Hardcore chain rule application

It’s long chain because you have all the hidden time steps before and the current time depends on all of the previous ones.



특히 현재 가 모든 과거의 들에 의존한다는 사실이 long chain을 만들어 준다.

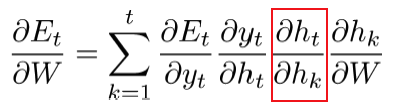
Q. Sequence길이가 매번 다를 텐데, update 기준을 어디에다 둘 것인가?

each time step에서는 W가 update되지 않는다. 모든 time이 지나고 나서야 gradient descent가 동작을 하고, W가 한 번 update된다. 즉, mini-batch size를 sentence 길이 단위 (e.g., 50 words)로 둔다.

Input Sequence 길이가 매번 다르더라도 효율적으로 한꺼번에 one block단위로 매트릭스 연산을 하기 위해 0으로 패딩 처리해서 길이를 모두 맞춘다.

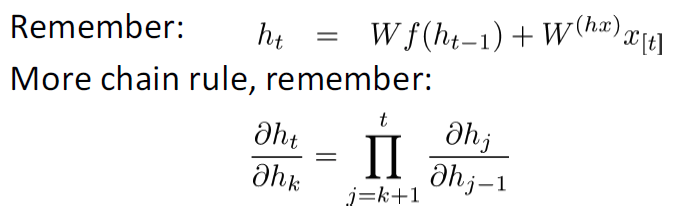
The way we will derive this and by looking at all these separate elements of the chain rule, is not as efficient as well we would do in standard back-prop. In standard backpropagation, we multiply matrices in a much more efficient way.

Let’s look at this particular output here (Useful for analysis we’ll look at)



• It’s similar to backprop but less efficient formulation

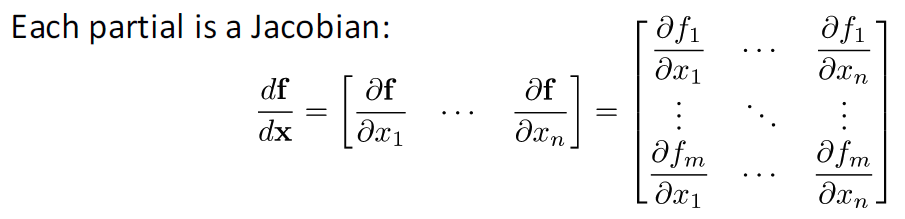
In order to compute the derivative of h-t at the current time step where the error came in with respect to all the potential (h-k)s that came before in this time series, we also have to apply the chain rule. 즉, chain rule 적용을 총 2번한다.



As you compute above, this is essentially going to be a derivative of a vector with respect to another vector. (외적하는 느낌) That means we have to use a Jacobean.

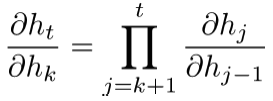
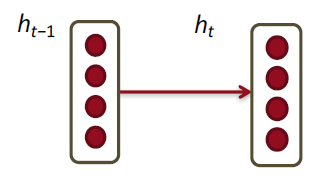
In general, the definition of Jacobian is when you have a vector-valued output function which is actually very rare (all cross entropy error usually end up being a single number. So, the most cases our function have a single output) But, when calculating the partial of , we will use the Jacobian.

General definition if you have a vector-valued output function, is basically that you have a vector for each of these derivatives for f with respect to the vector x. Since f itself is a vector, you basically get overall matrix with all the partial. It’s like a cross product. Each element of this output has a derivative with respect to each of this element of the output.

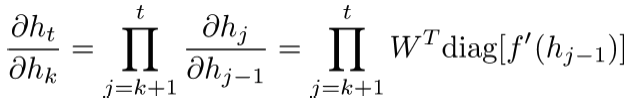


Actually, we don’t really have to use Jacobeans so, if you’re accidentally running into a Jacobean you probably could have simplified things when calculating derivatives.

Now, let’s look at this exact derivative here ,

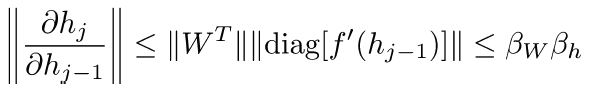
 

To compute this Jacobian , we look at all the elements of the Jacobian matrix. And it will become relatively straightforward. 즉, Jacobian matrix를 다음과 같이 표현할 수 있다. One way to describe that very efficiently, is to actually combine and diagonal.



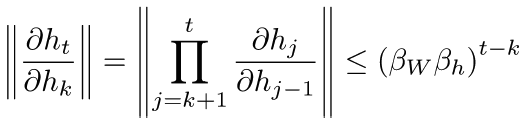
Now, let’s look at the norms of the Jacobian.

• Analyzing the norms of the Jacobians, yields:



We can define the upper bound of these norms. 가 에 mapping되고, 는 에 mapping된다. 따라서 는 upper bound로 정의할 수 있다.

Now, the gradient total is the product of all these Jacobian marices and each of these associated with a step in the full computation until the current time step t. So, once we multiply all of the upper bound of the norms (all of these Jacobian) with each other, we can basically get to this final output, which is .



Now, the problem is that these betas could be either very small in which case we multiply all Jacobians. Or they can be larger than 1. 즉, 가 1보다 작을 수도 있고, 1보다 클 수도 있다. 1보다 작으면 결과값이 매우 작게 되고, 1보다크면 결과값이 매우 크게 나온다.

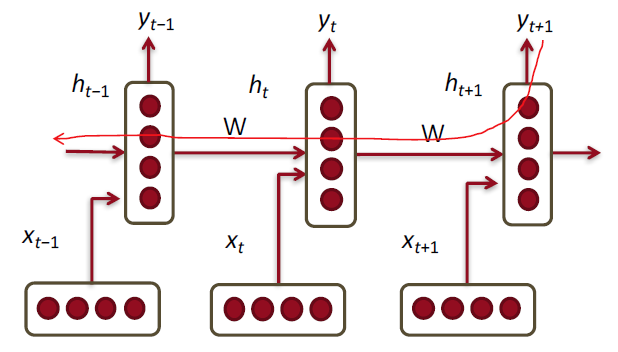
t can be a thousand or 5 thousand so, even if one of the beta is only 2, 2의 1000승 또는 5000승이 되어버리면 매우 큰 수가 되어버린다. (책 한 권을 한 문장이라고 생각(t=1000)하고 character단위로 학습해버릴 수도 있다.

• This can become very small or very large quickly [Bengio et al 1994], and the locality assumption of gradient descent breaks down → Vanishing or exploding gradient

So, we applied the chain rule on a high level and we looked at some simple upper bounds in terms of the norms of some of the elements of those chains in the chain rule. And we compute the derivation on each element. This is some intuition where the vanishing exploding gradient problems that come from.

**Why is the vanishing gradient a problem?**

Why the vanishing gradient problem actually is the real problem? The error at the time step, will ideally actually tell the next previous time step from many many time steps away, to change during backpropagation.



Let me give you some intuition. When we try to predict this word, y-(t+1), we will be able to modify some signal to change based on what the next word is.

**The vanishing gradient problem for language models**

• In the case of language modeling or question answering words from time steps far away are not taken into consideration when training to predict the next word

• Example,

* Jane walked into the room. John walked in too. It was late in the day. Jane said hi to \_\_\_

Given the example, what is your the internal language saying the next word?

You’re able to take all the previous time steps. (which is arbitrary harder)

Question answering captures all possible NLP problems. What’s the translation of the sentence? What POS sequences of the sentence? and so on.

In some sense, language modeling if we really were able to predict everything perfectly would solve almost all the NLP problems. In practice, it doesn’t really.

Again, in the example, here we see the problem of why we might want Jane and John to modify over 1,2,3, ~ 6, … and so on time steps, to try to help us predict the next word. The problem is when we actually do back-propagate that the next here is John and now we multiply the Jaconian that smaller than one at some point, it will in many cases empirically you can get to the most 7 time steps back. And after 7 time steps, the error does not modify anything in our parameter space anymore. The gradients almost have vanished until then. Since they’re so tiny, they don’t change the internal states anymore during backpropagation.

That means the model cannot tell itself that really what mattered back here you should give some change to your internal state to better predict the next word.

If you had general simpler non-recurrent models, where you just have an input and you have a lot of deep model or deep sequence, you can also get the vanishing gradient problem. It’s not as common but it still happened. It’s very problematic if you have some input and have a very deep network and you try to predict some output. 즉, deep network에서 Error signal이 input까지 도달하지 않는다면, 그 모델은 학습되지 않고 여전히 랜덤한 상태로 있을 것이다.

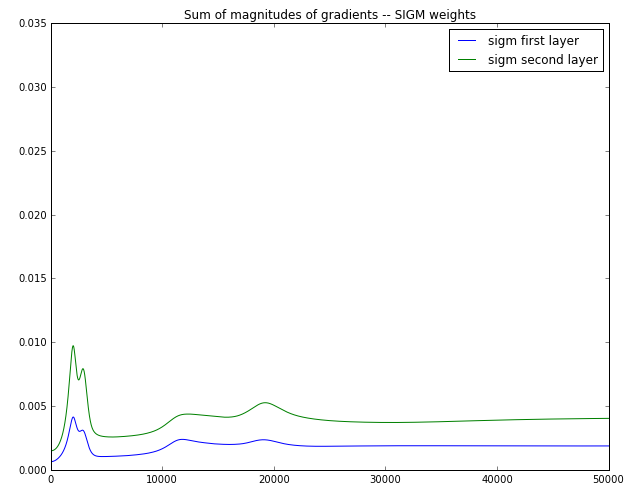
**IPython Notebook with vanishing gradient example**

• Example of simple and clean NNet implementation

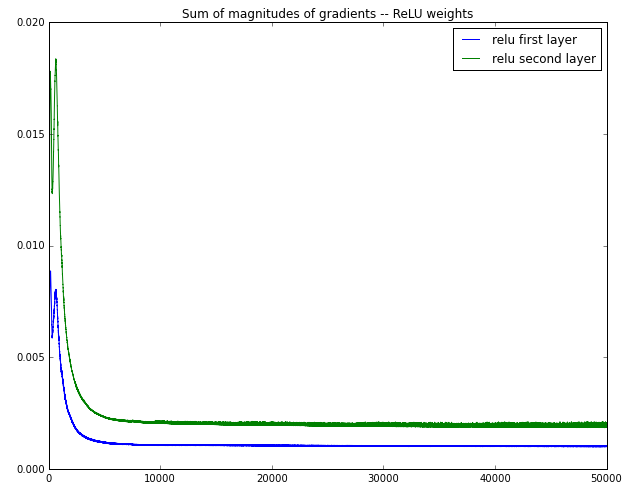
• Comparison of sigmoid and ReLu units

• A little bit of vanishing gradient

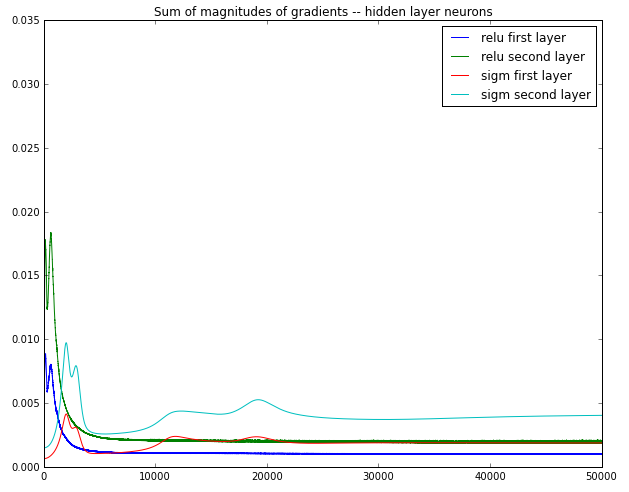
We basically look at the sum of the magnitude of those gradients over time.



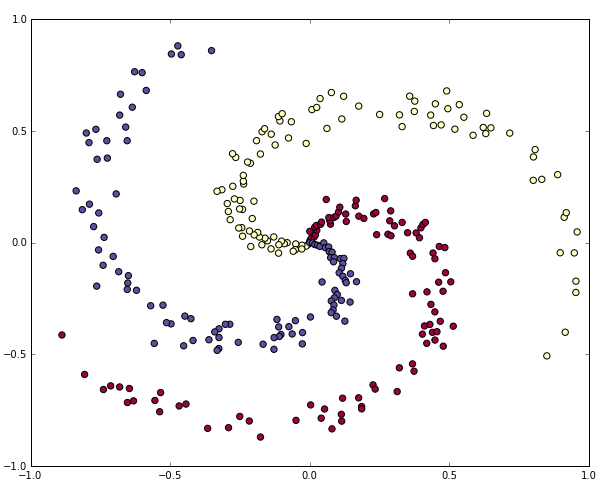
It’s not a Recurrent Neural Network, but a just simple 2 Neural Net. We already see that they’re almost half the size and already relatively small. Imagine they get half to each time and when you have 5000 times, they’re getting close to 0 very quickly.



It’s not as bad for rectified linear (it’s just linear once you’re not at 0 stage). So, you won’t have this problem as much. We won’t go down as much. But, let’s think about it. If you take the same matrix W, and you have some parts of which a linear and you keep multiplying the same W with linear thing, then you get to exploding not even exploding gradient, but exploding hidden states. Because during back-propagation, we multiply the same thing and so all the things that are positive you blow it up. So, relu is not the simple straightforward solution.



이런 비선형 데이터가 주어졌을 때, classification을 어떻게 하나. Sigmoid와 relu를 비교하면서 visualization해보자.



|  |  |
| --- | --- |
| This is the output with the sigmoid. Actually, it wasn’t able to perfectly fit | This is the output with relu. You can see some of the lines a little straighter because it’s linear elements for the hidden states. And it very nicely fit the function. |

**Trick for exploding gradient: clipping trick**

In order to use RNN, how do we deal with these problems?

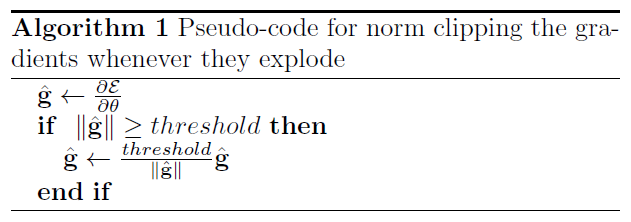
Vanishing gradient descent problems are much harder to fix and actually, we can’t use the vanilla RNN, instead, we should use long-short-term memory RNNs that are much more complex, but are interesting combinations of those Lego blocks.

If you think that all you need to know to make a certain predictions are the last seven time steps, then the vanishing gradient problem isn’t problematic. It’s only the problem if you want to push the accuracy higher and you think that your problem requires you to understand the entire sequence of the words that came before.

For exploding gradient problems, that we really have to solve, you can’t train them at all.

Trick is whenever gradients get too large, you just set them to maximum amount.

• The solution first introduced by Mikolov is to clip gradients to a maximum value. Actually, it’s not correct value of gradient. It just happens to working pretty well in practice.



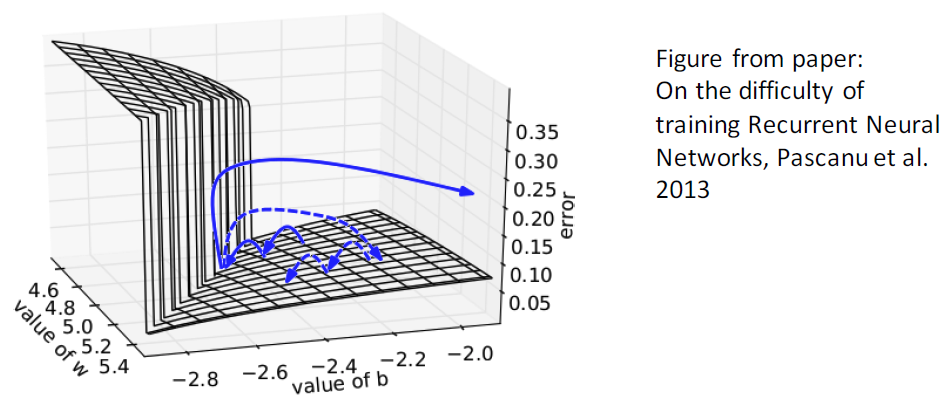
• Super simple but, it makes a big difference in RNNs

**Gradient clipping intuition**

Begio and his group try to understand what’s going on there and gain some intuition.

Again RNNs are super old models. People want to be understanding these models a lot better, they come up with lots of tricks to train them. That’s one of the many reasons for having this resurgence and deep learning. It’s not just sort of the old connectionist networks. There are lots of interesting new idea and this is one of them.

Let’s look at the super simplest possible recurrent neural network which has only a single hidden state. So, instead of H a vector, it’s just a single number and we still have the value of b as our bias term. And we basically try to have a very simple prediction problem and now we can visualize how big the error is, depending on where the weight is for the simple prediction problem.



• Error surface of a single hidden unit RNN

• High curvature walls

• Solid lines: standard gradient descent trajectories

• Dashed lines gradients rescaled to fixed size

This visualization shows us they’re very high curvature walls when you optimize the objective function. When you take convex optimization, we always assume you have a very nice bowl and you try to get to the global optimum.

bowl모양과는 다르게, It’s a very not well behaved objective function or optimization landscape that we try to optimize.

When you here (this relatively smooth curvature), you basically just take small steps for each gradient descent step that you do and you will notice here is once you hit this point (=high curvature wall) here, the standard gradient descent will push away back very far from this.

The idea of clipping the gradient, is to rescaling it (say don’t go very far away)

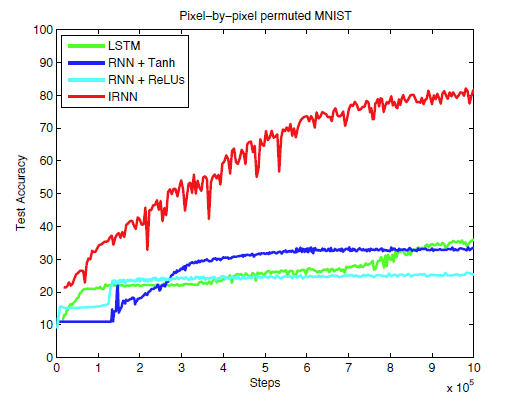
**For vanishing gradients: Initialization + ReLus!**

There’s another trick for dealing with vanishing gradient problems and it works for some cases but it’s not the final answer that everybody will use. But, there are two important tricks. The really important one is we can actually initialize W with simple identity matrix.

Intuitively, whenever you bring in a word vector, as you start your optimization, you can average them all in. and then you let the model start doing something smarter averaging. (원래 보통 W matrix를 랜덤값으로 초기화한다.)

Again, the idea is saying as well let’s not make a completely randomized, but make default to average to word vector as we start to compute what a good hidden state is.

Also, he actually uses to rectified linear units for sequences that aren’t super crazy long. 여기서는 MNIST 데이터를 사용하였다. CNN을 사용하지 않고, every pixel마다 time step을 달아서 RNN으로 모델링 하였다. You have relatively 32 by 32 many hidden states.



• Initialize to identity matrix I and f(z)=rect(z)=max(z,0)

• → Huge difference!

• Initialization idea first introduced in Parsing with Compositional Vector Grammars, Socher et al. 2013

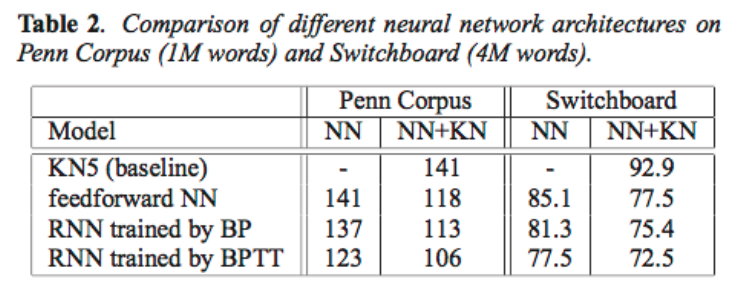
• New experiments with recurrent neural nets in A Simple Way to Initialize Recurrent Networks of Rectified Linear Units, Le et al. 2015

**Perplexity Results**

In a more realistic problem of RNNs, how they started to gain a lot of popularity is language modeling.

Before NN came out, the standard language modeling is a clever way to combine the counts and probabilities from 5-grams, 4-grams,…,etc.

→ KN5 = Count-based language model with Kneser-Ney smoothing & 5-grams



* Penn corpus : less perplexity to RNN than KN5
* 현재는 성능이 매우 향상되었지만, 이 테이블 수치는 실험 초기수치로서 사람들이 처음 NN에 대해 좋게 평가하게 된 계기가 되었다. 그리고 RAM requirements of the NN are already a lot better than that of these gigantic count base model.
* It’s a little confusing, BP and BPTT should be the same thing. There is basically a simple trick (one last implementation trick)

**One last implementation trick**

• You only need to pass backwards through your sequence once and accumulate all the deltas from each

As you do forward propagation, you only have to do one back-propagation to update all your weights. 즉, 과거 시간으로 가는 것은 한 번만 한다.

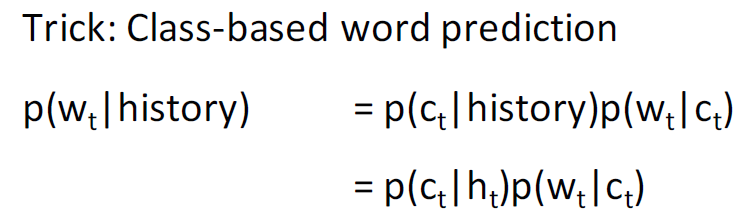
본래적으로는 매 시간마다 과거 시간여행을 떠나야 한다. 예를들어, time step 10, you go and you back-propagate your errors all the way to time step 0. 그리고 time step 11도… 그리고 time step 12도….time step 0까지 시간여행을 떠나야 한다.

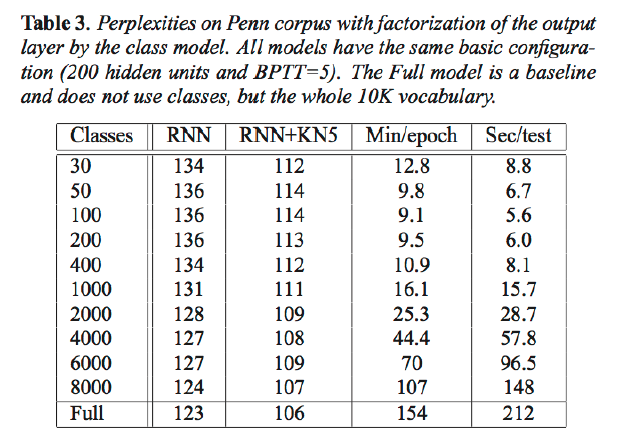
**Problem: Softmax is huge and slow**

One trick that was evaluated and analyzed is to also have so-called Class-based or hierarchical word prediction. The main idea is to have classes and each class has a subset of the words. So, training time is the time we only need to update the class probabilities.

It’s very a clever idea. Because softmax has a hundred thousand words, if you predict just the next word, you have to have very costly matrix multiplication (A hundred thousand times the size of hidden states)

Here you just have the size of hidden states times the number of classes and times the number of word in that class only. So, it’s much more efficient.





* Minutes per epoch are much smaller if you have fewer classes. And as you have more classes, it gets slower
* The more classes, the better perplexity but also worse speed

**Sequence modeling for other tasks**

Language modeling은 simple task이고, 좀 더 high level task에는 sequence modeling이 있다.

• Classify each word into:

- NER

- Entity level sentiment in context

- opinionated expressions

• Example application and slides from paper Opinion Mining with Deep Recurrent Nets by Irsoy and Cardie 2014

**Opinion Mining with Deep Recurrent Nets**

Goal: Classify each word as

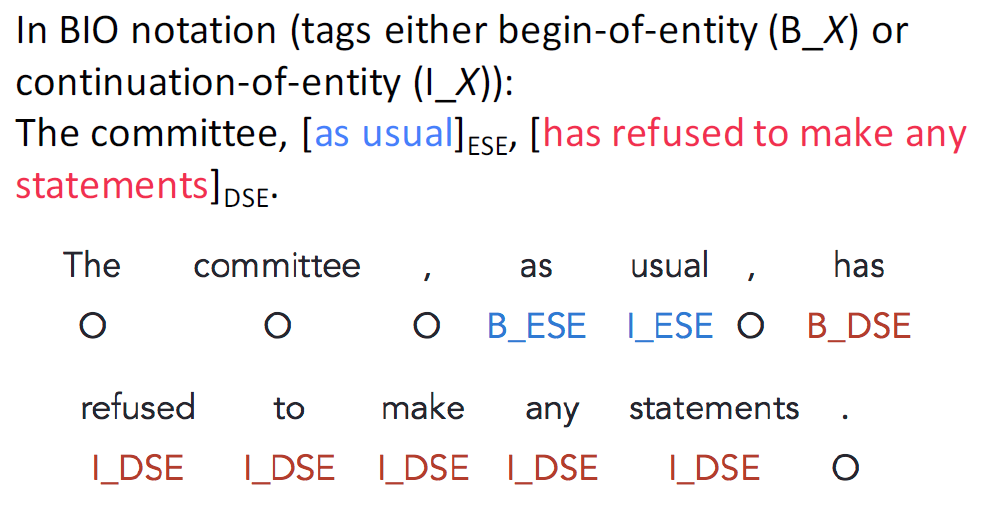
① direct subjective expressions (DSEs)

② expressive subjective expressions (ESEs)

- DSE: Explicit mentions of private states or speech events expressing private states

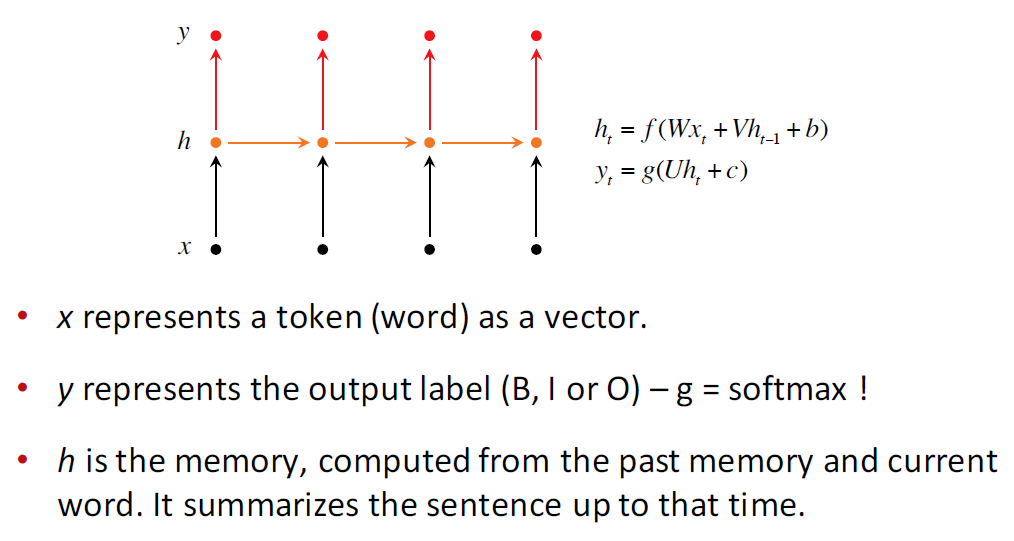
- ESE: Expressions that indicate sentiment, emotion, etc. without explicitly conveying them

For example….



총 5가지 종류의 태그가 있다. O, B\_ESE, I\_ESE, B\_ESE, I\_DSE

**Approach: Recurrent Neural Network**

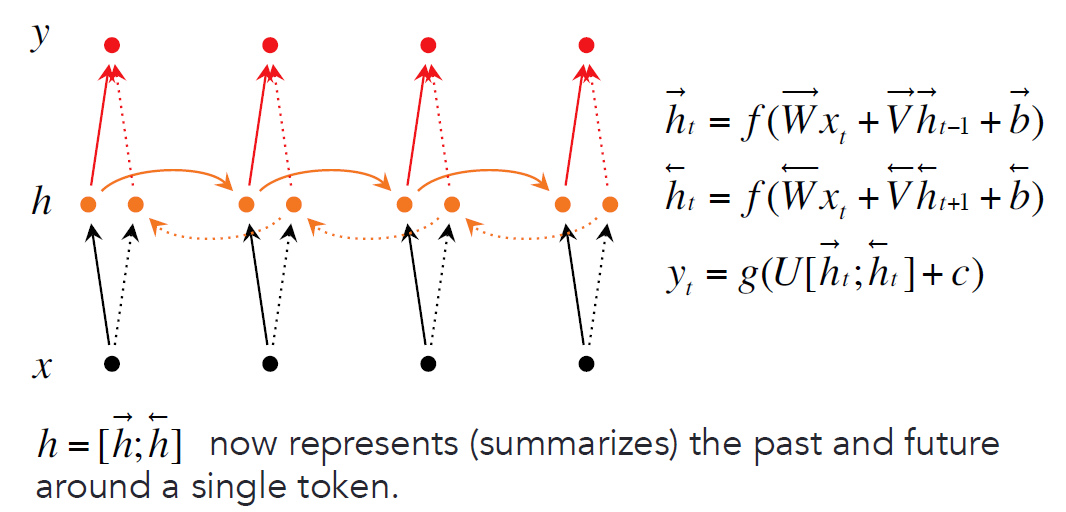


**Bidirectional RNNs**

이전 예제에서 us usual에서 usual을 태깅할 때, 왼쪽 시퀀스만 보는게 아니고, 오른쪽 시퀀스도 같이 보는게 더 좋을 수도 있다. 여기서 바로 오른쪽에 콤마가 있는데 이는 태깅을 하는데 도움을 줄 수도 있다. Again, to classify a single word in the context, it’s very helpful to look at not just left context but also the right context.

Take a sequence. Reverse it. And then run another RNN with a different set of weights.

For classification you want to incorporate information from words both preceding and following.

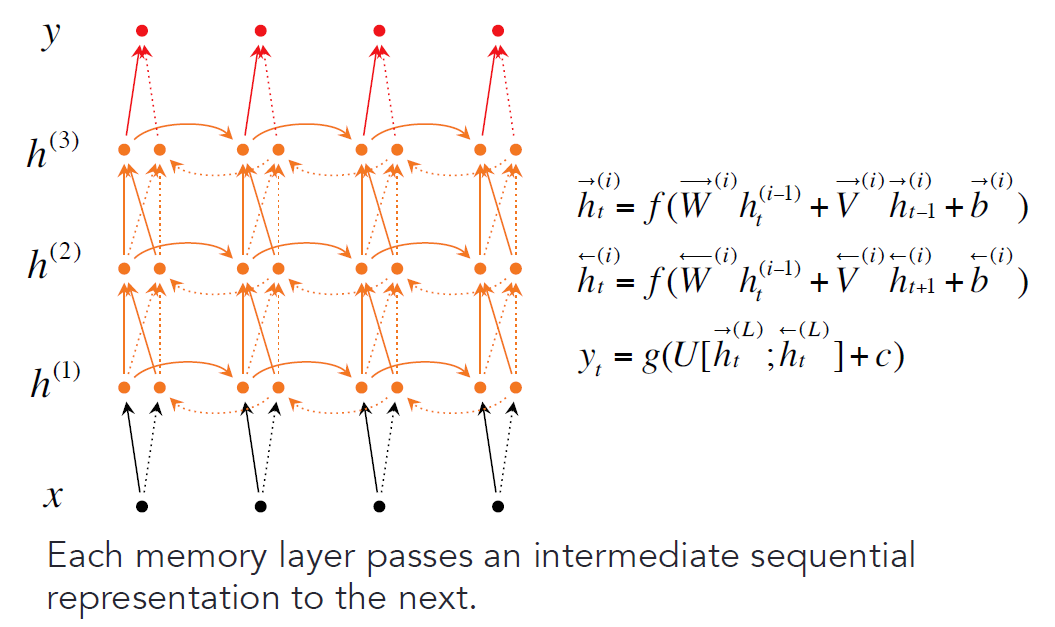


* Concatenate the two vectors and gives that as an input to the softmax
* 두 벡터를 averaging하는 방법도 일리가 있다.
* 왼쪽으로만 봐도 충분한 정보를 얻을 수 있지만, 어떤 경우에 간혹 오른쪽에서 critical한 정보를 얻을 수도 있다. 예를 들어, 어떤 단어가 형용사, 부사가 다 되는데, 오른쪽에 있는 단어가 명사면 단번에 형용사로 알아볼 수 있다.

The question that why to use bidirectional RNNs is because we can really keep track of everything. Even if we could keep very good track of everything to the left, it will help in many cases to predict a label for the current word but also taking into consideration everything to the right.

**Deep Bidirectional RNNs**

Now, what if we want to make it even more powerful? And we actually want to have a deep transformation that every word vector goes through and every hidden state goes through.



**Data**

How to evaluate this? Just look at how often you get it right.

• MPQA 1.2 corpus (Wiebe et al. 2005)

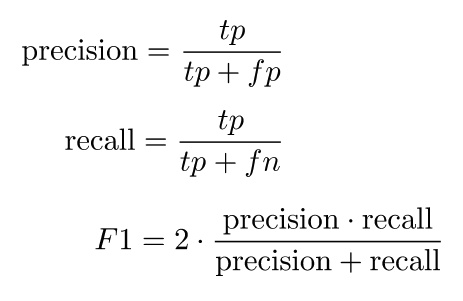
• consists of 535 news articles (11,111 sentences)

• manually labeled with DSE and ESEs at the phrase level

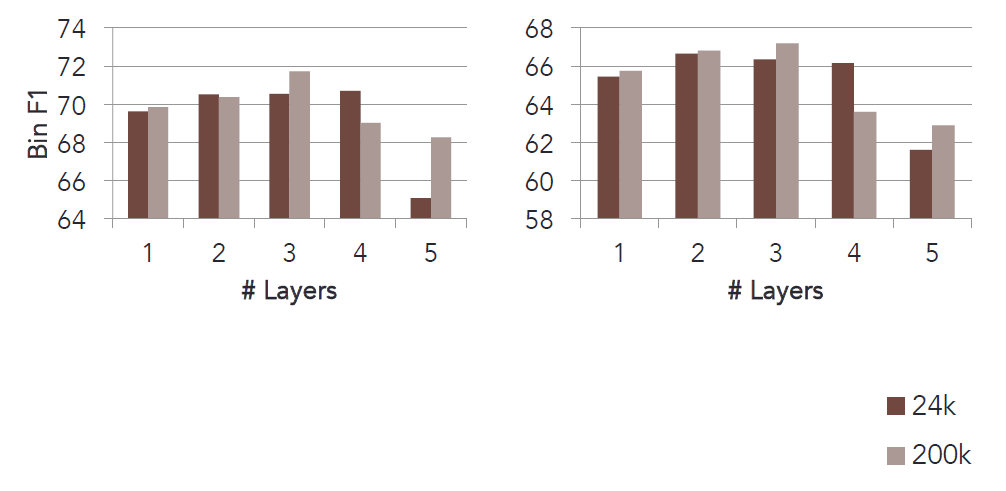
The problem is the most words don’t have any of the labels.

이렇게 데이터가 skewed distribution일 경우 precision 이외에 전체를 볼 수 있는 recall도 함께 고려해줘야 한다.

• Evaluation: F1



**Evaluation**



파라미터 개수 (24k or 200k) 그리고 layer 개수에 따라 모델 성능이 어떻게 달라지는지 확인.

3 layer 초과되면 성능이 더 떨어진다.

**Recap**

• Recurrent Neural Network is one of the best deepNLP model families

• Training them is hard because of vanishing and exploding gradient problems

• They can be extended in many ways and their training improved with many tricks

• Next week: Most important and powerful RNN extensions with LSTMs and GRUs