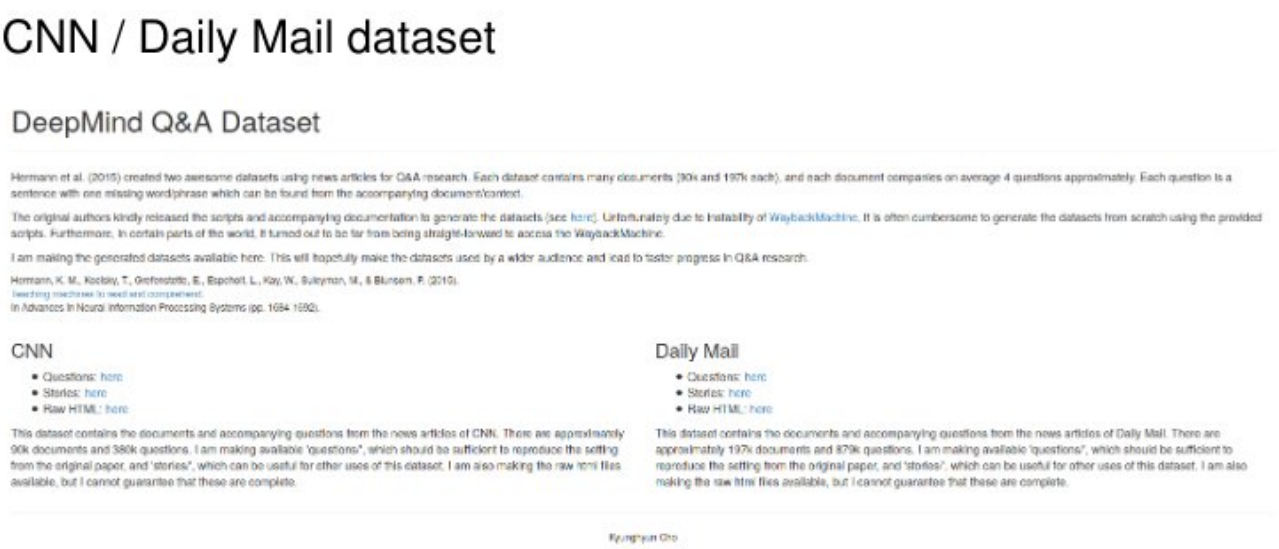
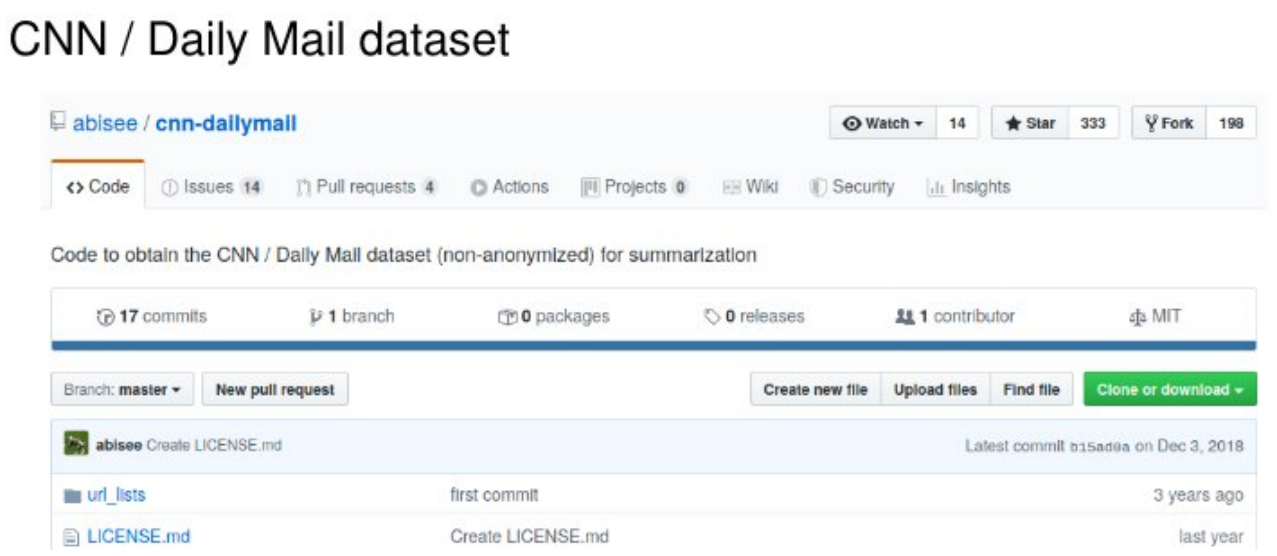
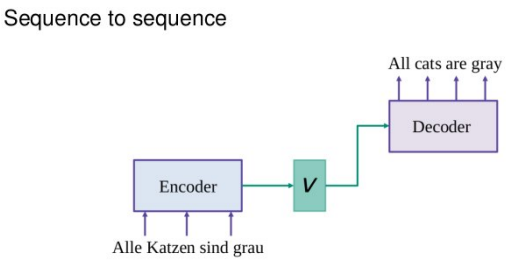




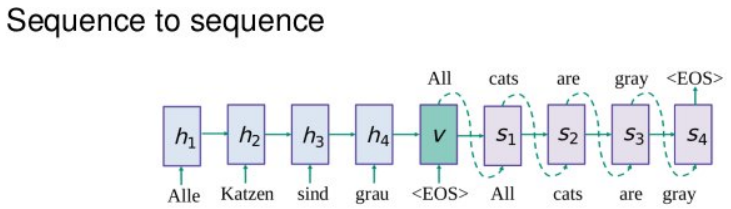


Slide 10 (seq2seq1)

We are going to speak about encoder-decoder architecture and about attention mechanism. We will cover them by the example of neural machine translation, just because they were mostly proposed for machine translation originally. But now they are applied to many, many other tasks. For example, you can think about summarization or simplification of the texts, or sequence to sequence

chatbots and many, many others. Now let us start with the general idea of the architecture. We have some sequence as the input, and we would want to get some sequence as the output. For example, this could be two sequences for different languages, right? We have our encoder and the task of the encoder is to build some hidden representation over the input sentence in some hidden way. So we get this green hidden vector that tries to encode the whole meaning of the input sentence.

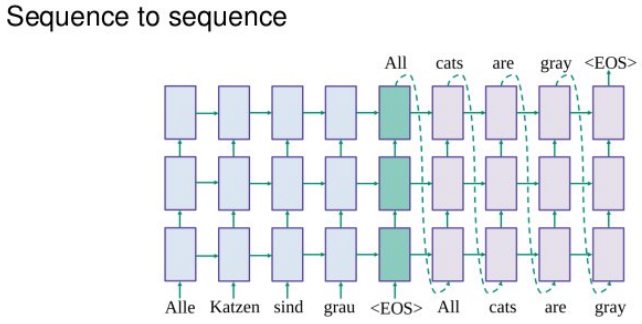
The encoder task is to decode this thought vector or context vector into some output representation. For example, the sequence of words from the other language. Now what types of encoders could we have here? Well, one most obvious type would be recurrent neural networks, but actually this is not the only option.



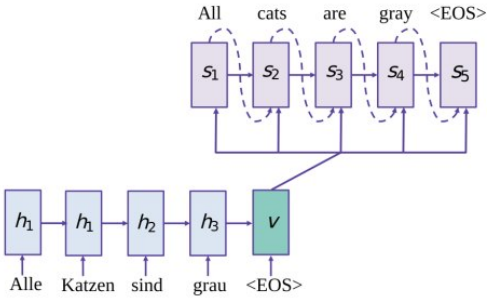
Slide 11 (seq2seq2)

Okay, now what is the first example of sequence to sequence architecture? This is the model that was proposed in 2014 and it is rather simple. So it says, we have some LSTM module or RNN module that encodes our input sentence, and then we have end of sentence token at some point. At this point, we understand that our state is our thought vector or context vector, and we need to decode starting from this moment. The decoding is conditional language modelling. So you're already familiar with language modelling with neural networks, but now it is conditioned on this context vector, the green vector. Okay, as any other language model, you usually fit the output of the previous state as the input to the next state, and generate the next words just one by one.

Slide 12(seq2seq3)



Now, let us go deeper and stack several layers of our LSTM model. You can do this straightforwardly like this. So let us move forward, and speak about a little bit different variant of the same architecture.



Slide 13(seq2seq4)

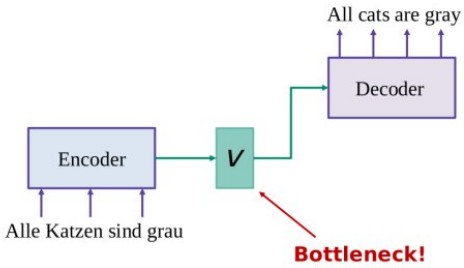
One problem with the previous architectures is that the green context vector can be forgotten. So if you only feed it as the inputs to the first state of the decoder, then you are likely to forget about it when you come to the end of your output sentence. So it would be better to feed it at every moment. And this architecture does exactly that, it says that every stage of the decoder should have three kind of errors that go to it:
- First, the error from the previous state,
- then the error from this context vector,
- and then the current input which is the output of the previous state.

$$\mathbb{P}[y_1, \dots y_J | x_1, \dots, x_I] = \prod_{j=1}^{J} \mathbb{P}[y_j | v, y_1, \dots, y_{j-1}]$$

- **Encoder:** maps the source sequence to the hidden vector
  $RNN: h_i = f(h_{i-1}, x_i)$, where $v = h_I$
- **Decoder:** performs language modeling given this vector
  $RNN: s_j = g(s_{j-1}, [y_{j-1}, v])$
- **Prediction:** $\mathbb{P}[y_j | v, y_1, \dots, y_{j-1}] = softmax(Us_j + b)$

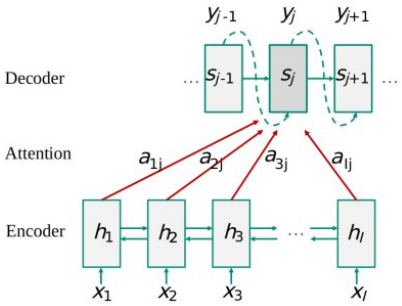
Slide 14(seq2seq5)

Okay, now let us go into more details with the formulas.
1. So you have your **sequence modeling task** conditional because you need to produce the probabilities of one sequence given another sequence, and you factorize it using the chain rule. Also importantly you see that x variables are not needed anymore because you have encoded them to the v vector.
2. **V vector** is obtained as the **last hidden state of the encoder,** and encoder is just recurrent neural network.
3. The **decoder** is also the **recurrent neural network**. However, it has more inputs, right? So you see that now I concatenate the **current input Y** with the **V vector.** And this means that I will use all kind of information, all those three errors in my transitions.
4. Now, how do we **get predictions** out of this model? Well, the easiest way is just to do softmax, right? So when you have your **decoder RNN**, you have your **hidden states of your RNN** and they are called **S_j**. You can just apply some linear layer, and then softmax, to get the probability of the current word, given everything that we have, awesome.



Slide 15(seq2seq6)

Okay, even though these representations are so nice, this is still a bottleneck. So you should think about how to avoid that. And to avoid that, we will go into attention mechanisms and this will be the topic of our next video.



Slide 17 (att1)

Attention mechanism is a super powerful technique in neural networks. So let us cover it first with some pictures and then with some formulas. We have some **encoder that has h states** and **decoder that has some s states**. Now, let us imagine that **we want to produce the next decoder state.** So we want to compute **s_j**. How can we do this? In the previous video, we just used the **v vector**, which was the information about the whole encoded input sentence. And instead of that, we could do something better. **We can look into all states of the encoder with some weights.** So this alphas denote some weights that will say us whether it is important to look there or here. How can we compute this alphas? Well, we **want them to be probabilities**, and also, we want them to capture some similarity between our current moment in the decoder and different moments in the encoder. This way, we'll look into more similar places, and they will give us the most important information to go next with our decoding.

- Encoder states are weighted to obtain the representation relevant to the decoder state:

$$v_j = \sum_{i=1}^{I} \alpha_{ij} h_i$$

- The weights are learnt and should find the most relevant encoder positions:

$$\alpha_{ij} = \frac{exp(sim(h_i, s_{j-1}))}{\sum_{i'}^{I} exp(sim(h_{i'}, s_{j-1}))}$$

Slide 18 (att2)

If we speak about the same thing with the formulas, we will say that, now, instead of just one **v** vector that we had before, we will have **v_j,** which is different for different positions of the decoder. And this **v_j** vector will be computed as an **average of encoder states.** And the weights will be computed as softmax because they need to be probabilities. And this softmax will be applied to similarities of encoder and decoder states.

- **Additive attention:**

$$sim(h_i, s_j) = w^T \tanh(W_h h_i + W_s s_j)$$

- **Multiplicative attention:**

$$sim(h_i, s_j) = h_i^T W s_j$$

- **Dot product also works:**

$$sim(h_i, s_j) = h_i^T s_j$$

Slide 19 (att3)

Now, do you have any ideas how to compute those similarities? So papers actually have tried lots and lots of different options, and there are just three options for you to try to memorize.

1. And maybe we just do not want to care at all with our mind how to compute it. We just want to say, "Well, neural network is something intelligent. Please do it for us." And then we just take one layer over neural network and say that it needs to predict these similarities. So you see there that you have h and s multiplied by some matrices and summed. That's why it is called **additive attention**. And then you have some non-linearity applied to this.
2. Another way is to say, maybe we need some weights there, some metrics that we need to learn, and it can help us to capture the similarity better. This thing is called **multiplicative attention.**
3. Let us just do **dot product of encoder and decoder states.** It will give us some understanding of their similarity.

$$\mathbb{P}[y_1, \dots y_J | x_1, \dots, x_I] = \prod_{j=1}^{J} \mathbb{P}[y_j | v, y_1, \dots, y_{j-1}]$$

- **Encoder:** maps the source sequence to the hidden vector
  $RNN: h_i = f(h_{i-1}, x_i)$, where $v = h_I$
- **Decoder:** performs language modeling given this vector
  $RNN: s_j = g(s_{j-1}, [y_{j-1}, v])$
- **Prediction:** $\mathbb{P}[y_j | v, y_1, \dots, y_{j-1}] = softmax(Us_j + b)$

Slide 20 (att4)

1. Now, let us put all the things together, just again to understand how does attention works. You have your conditional language modeling task. You'll try to predict **Y** sequence given s sequence. And now, you encode your **x** sequence to some **v_j** vector, which is different for every position.

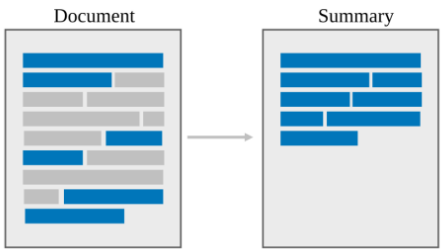
3. This **v_j** vector is used in the **decoder**. It is concatenated with the **current input** of the decoder. And this way, the decoder is aware of all the information that it needs, the previous state, the current input, and now, this specific **context vector**, computed especially for this current state.

Attention helps to focus on different parts of the sentence when you do your predictions. And for long sentences, it is really important because, otherwise, you have to encode the whole sentence into just one vector, and this is obviously not enough.

Summarization with pointer-generator networks

Slide 21 (pointer_model 1)

There are many different tasks in NLP can be solved as sequence to sequence tasks. This is a paper from Chris Manning Group, and it is nice because it tells us that on the one hand, we can use encoder-decoder architecture, and it will work somehow. On the other hand, we can think a little bit and improve a lot. So, the improvement will be based on pointer networks.

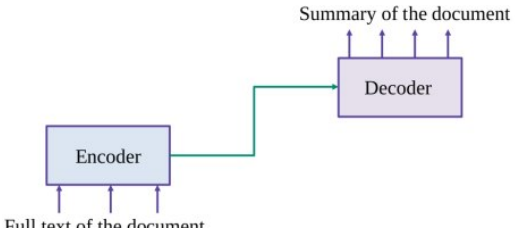


Slide 22 (pointer_model 2)

Summarization task is when you need to get the short summary of some document. Summarization can have several types.

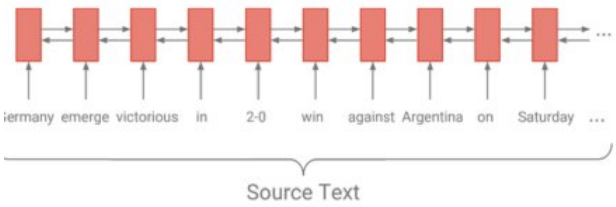
1. So we can speak about **extractive summarization**, and it means that we just extract some pieces of the original text.

2. Or we can speak about **abstractive summarization**, and it means that we want to generate some summary that is not necessarily from out text, but that nicely summarizes the whole text. So this is obviously better, but this is obviously more difficult.



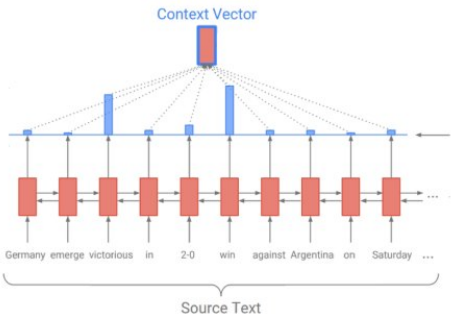
Slide 23 (pointer_model 3)

Now, let us try to do **extractive summary with sequence to sequence model**. So you get your full text as the input, and you get your summary as the output, and you have your encoder and decoder as usual.



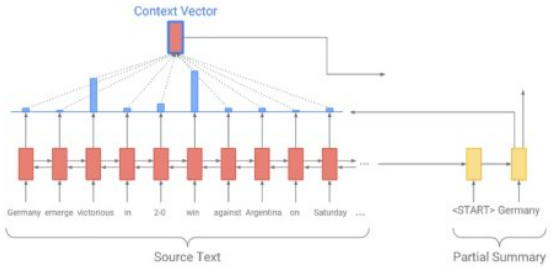
Slide 24 (pointer_model 4)

We have usually some encoder, for example bidirectional LSTM and



Slide 25 (pointer_model 5)

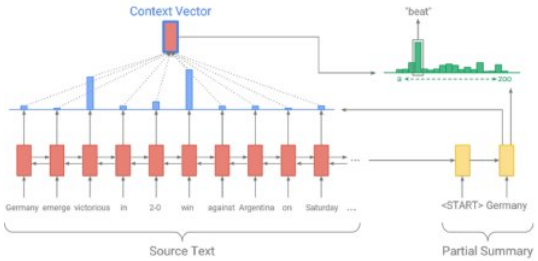
then we have some attention mechanism, which means that we produce some probabilities that tells us what are the most important moments in our input sentence. Now, you see there is **some arrow on the right of the slide.**



Slide 26 (pointer_model 6)

The attention mechanism is about the important moments of the **encoder based on the current moment of the decoder.** So, now we definitely have the **yellow part** which is **decoder**, and then the **current state** of this decoder tells us how to compute attention.

Slide 27 (pointer_model 7)



Just to have the complete scheme, we can say that we use this attention mechanism to generate our distribution or vocabulary. Awesome. So, this is just a recap of encoder-decoder attention architecture.

One problem is that the model is abstractive, so the model generates a lot, but it doesn't know that sometimes, it will be better just to copy something from the input. So, the next architecture will tell us how to do it.

1. Attention distribution (over source positions):

$$e_i^j = w^T \tanh(W_h h_i + W_s s_j + b_{attn})$$

$$p^j = softmax(e^j)$$

2. Vocabulary distribution (generative model):

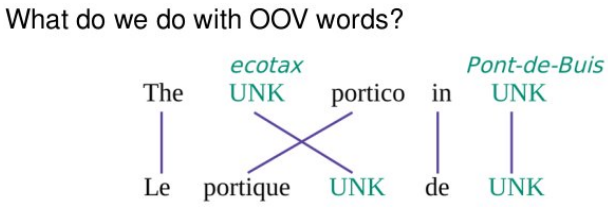
$$v_j = \sum_i p_i^j h_i$$

$$p_{vocab} = softmax(V'(V[s_j, v_j] + b) + b')$$

Slide 28 (pointer_model 8)

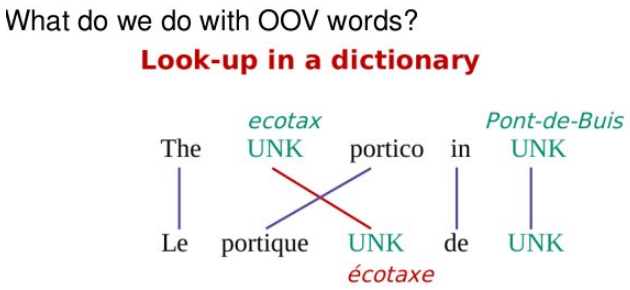
Let us have a closer look into the formulas and then see how we can improve the model. So, first, attention distribution. **H** is the **encoder** states and **S** is the **decoder** states. So, we use both of them to compute the **attention weights,** and we apply **softmax** to get probabilities. Then, we use these probabilities to weigh **encoder states** and get **v_j. v_j** is the **context vector** specific for the position **j** over the **decoder.**

Then how do we use it? We have seen in some other videos that we can use it to compute the next state of the decoder. In this model, we will go in a little bit more simple way. Our **decoder** will be just normal RNN model, but we will take the state of this RNN model **s_j** and concatenate with **v_j** and use it to produce the probabilities of the outcomes. So, we just concatenate them, apply some transformations, and do softmax to get the probabilities of the words in our vocabulary.



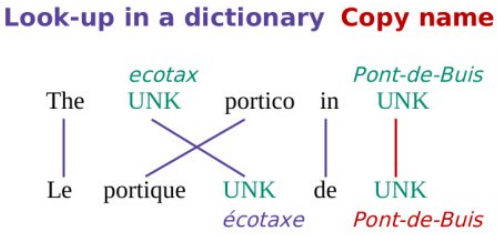
Slide 29 (pointer_model 9)

And copy mechanism is something to help with out-of-vocabulary words. Imagine you have some sentence, and some words are **UNK** tokens. So you do not have them in the vocabulary, and you do not know how to translate them, you get **UNK** tokens as the result. But what if you know the **word alignments**?



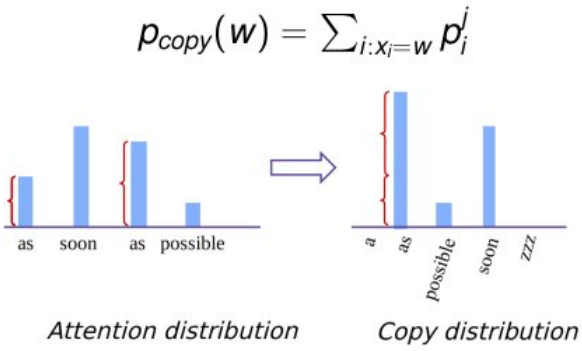
Slide 30 (pointer_model 10)

If you know how the words are aligned, you can use that. So you can say, okay, this **UNK** corresponds to that **UNK**, which corresponds to this source word. Let us just **look up in the dictionary.**



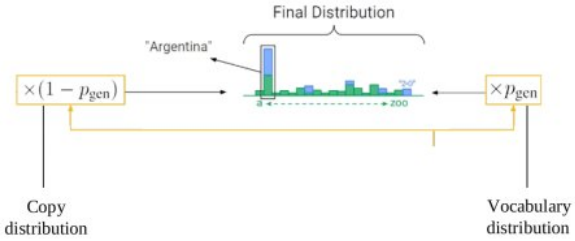
Slide 31 (pointer_model 11)

or let us just **copy**, because this is the name of some place or some other name. Let us just copy this as is. This is why it is called copy mechanism, and the algorithm is super simple.



Slide 32 (pointer_model 12)

Now, how can we improve our model? We would want to have some copy distribution. So, this distribution should tell us that sometimes it is nice just to copy something from the input. How can we do this? Well, we have attention distribution that already have the probabilities of different moments in the input. What if we just sum them by the words? So, for example, we have seen **as two times** in our input sequence. Let us say the **probability of as should be equal to the sum of those two**. And in this way, we'll get some distribution over words that occurred in our input.



Slide 33 (pointer_model 13)

Now, the final thing to do will be just to have a mixture of those two distributions. So, one is this **copy distribution** that tells that some words from the input are good, and **another distribution is our generative model**.

4. Final distribution:

$$p_{final} = p_{gen} p_{vocab} + (1 - p_{gen}) p_{copy}$$

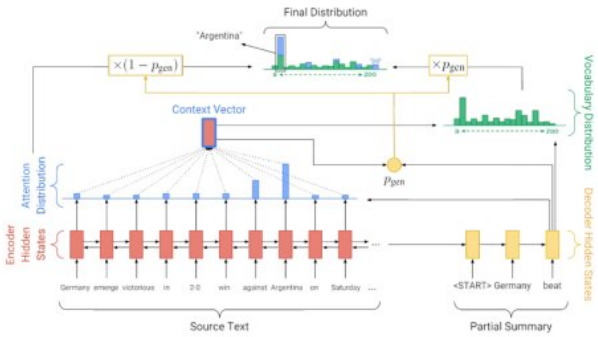
$$p_{gen} = sigmoid(w_v^T v_j + w_s^T s_j + w_x^T y_{j-1} + b_{gen})$$

5. Training:

$$Loss = -\frac{1}{J} \sum_{j=1}^{J} \log p_{final}(y_j)$$

Slide 34 (pointer_model 14)

So just a little bit more formulas. How do we weigh these two distributions? We weigh them with some probability **p_generation** here, which is also sum function. So every thing which is in **green** on this slide is some **parameters**. So, you just learn these **parameters** and you learn to produce this probability to weigh two kinds of distributions. And this **weighting coefficient** depends on everything that you have, on the **context vector v_j,** on the **decoder state s_j**, on the **current inputs to the decoder.** So you just apply transformations to everything that you have and then **sigmoid** to get probability. **The training objective for our model would be, as usual, cross-entropy loss** with this final distribution. So, we will try to predict those words that we need to predict. This is similar to **likelihood maximization**, and we will need to optimize the subjective.



Slide 35 (pointer_model 15)

Now, this is just the whole architecture, just once again. We have encoder with attention, we have yellow decoder, and then we have two kinds of distributions that we weigh together and get the final distribution on top.

This is called pointer-generation model because it has two pieces, generative model and pointer network. So this part about copying some phrases from the input would be called pointer network here.

Additional:
Coverage mechanism. *Remember you have attention probabilities. You know how much attention you give to every distinct piece of the input. Now, let us just accumulate it. So at every step, we are going to sum all those attention distributions to some coverage vector, and this coverage vector will know that certain pieces have been attended already many times. How do you compute the attention then? Well, to compute attention, you would also need to take into account the coverage vector. So the only difference here is that you have one more term there, the coverage vector multiplied by some parameters, green as usual, and this is not enough. So you also need to put it to the loss. Apart from the loss that you had before, you will have one more term for the loss. It will be called coverage loss and the idea is to minimize the minimum of the attention probabilities and the coverage vector. So imagine you want to attend some moment that has been already attended a lot, then this minimum will be high and you will want to minimize it. And that's why you will have to have small attention probability at this moment. On the opposite, if you have some moment with low coverage value, then you are safe to try to have high attention weight here because the minimum will be still the low coverage value, so the loss will not be high. So this loss motivates you to attend those places that haven't been attended a lot yet.*

	ROUGE score		
	1	2	L
abstractive model (Nallapati et al., 2016)	35.46	13.30	32.65
extractive model (Nallapati et al., 2017)	39.6	16.2	35.3
lead-3 baseline	40.34	17.70	36.57
seq2seq + attention	31.33	11.81	28.83
pointer-generator	36.44	15.66	33.42
pointer-generator + coverage	**39.53**	**17.28**	**36.38**

Slide 36 (pointer_model 16)

ROUGE score is an automatic measure for summarization. Now, you can see that pointer-generator networks perform better than vanilla seq2seq plus attention, and coverage mechanism improves the system even more. However, all those models are not that good if we compare them to some baselines. One very competitive baseline would be just to take first three sentences over the text. But it is very simple and extractive baseline, so there is no idea how to improve it. I mean, this is just something that you get out of this very straightforward approach.

Paper results

- Union of the best models: WGAN(IWGAN) is an improved version of GAN which solves problems with training and stability; Pointer-network is one of the best models for a summarization task.
- GAN setup gives a possibility to train the model in an unsupervised way. Therefore it is possible to use big datasets in order to increase quality.
- Pointer-network did not gives guarantee that:
  - summary will be human readable and that it will be key information
  - we need to configure parameters in order to get short summaries
- Discriminator should solve the problem being human readable
- Reconstructor solves the problem making summaries short (If the information will be not the key one than we will not be able to reconstruct it correctly, like in AutoEncoder)

Key learnings from the paper

- Embed, encode, attend, predict: The powerful deep learning formula for state-of-the-art NLP models.
- Keras very flexible framework which is underestimated in community.
- Keras and tensorflow is very unstable from version to version.
- Migration from python2 to python3 is nightmare.
- Human readable code using code patterns speed up development and understanding.

Critics of authors

- Paper is very bad detailed, there are missed parts.
- Code provided by authors is hardly reproducible and not documented.
- The final results evaluation is made with many assumptions which does not give a possibility to compare it with baseline results.
- contribution from the paper is not significant enough to the field.

Future work

- Make an architecture more simple which will speed up training and future modification for the model:
  - Check whether model performs better with another type of network for Generator and Reconstructor.
  - Measure performance with Reconstructor and without.