CSC2516: PA #3

Due on Mar 22

Roger Grosse, Jimmy Ba

Thomas Hollis

Problem 1.1

How do you think the architecture in Figure 1 will perform on long sequences, and why? Consider the amount of information the decoder gets to see about the input sequence.

Solution

I believe that the architecture in Figure 1 will perform increasingly poorly in long sequences. This is because the longer the sequence, the farther away the distance is between the encoder input and decoder output will be. Alternatively said, this will create long-term dependencies that are difficult to handle. This is turn will result in the infamous "vanishing gradient" problem whereby gradients in backpropagation are separated by so many steps that they end up vanishing to 0 (or sometimes exploding).

Problem 1.2

What are some techniques / modifications we can use to improve the performance of this architecture on long sequences? List at least two.

Solution

One simple solution to use is to keep things stable (no exploding or vanishing gradients) by using gradient clipping. Gradient clipping ensures the gradient has a norm of at most η. While the gradients will be biased, at least they won't blow up.

Alternatively we can reverse the input sequence so that there is only one time step between the first unit of the input and the first unit of the output. Usually in English, and similarly in this particular Pig-Latin application, dependencies are between units in the same or close positions within the input and output sequence.

Another approach would be to modify the architecture entirely by replacing the RNN units by LSTM units that contain memory cells which have controllers saying when to store or forget information.

Problem 1.3

What problem may arise when training with teacher forcing? Consider the differences that arise when we switch from training to testing.

Solution

Teacher forcing does not train the network to recover from a bad prediction early on in the sequence being forcasted at the decoder stage. This occurs because, in teacher forcing, the network is explicitly fed the labelled data at the decoder stage during train time, rather than left to execute based on its previous predictions. However, in test time since the labels are not available, the network feds its own predictions into the next RNN cell. Therefore, this can result in wildly incorrect ending predictions if errors arise early on and accumulate in the sequence at the decoder, as the network is incapable of recovering from the earlier mis-step.

Problem 1.4

Can you think of any way to address this issue? Read the abstract and introduction of the paper "Scheduled sampling for sequence prediction with recurrent neural networks", and answer this question in your own words.

Solution

We can address this issue by replacing the teacher forcing method by a "curriculum learning strategy". This strategy essentially changes the training stage to gradually mostly consider the generated token rather than the correct label, thereby reducing the amount of guidance that is given to the network over time. This is implemented by exposing the network to the test-time situation of prediction using its own forecasts on an increasingly large proportion of the training steps.

This encourages the model to learn, in a more robust manner, the information in the training data. Experimentally, this was shown to yield better performances on prediction tasks as the network is able to avoid some of the compounding errors previously associated with the teacher learning approach.

Problem 2.1

Find the `GRU cell` section of the notebook. Complete the `__init__` and `forward` methods of the `MyGRUCell` class, to implement the above equations. A template has been provided for the forward method.

Solution

See code attached in file `nmt.ipynb`.

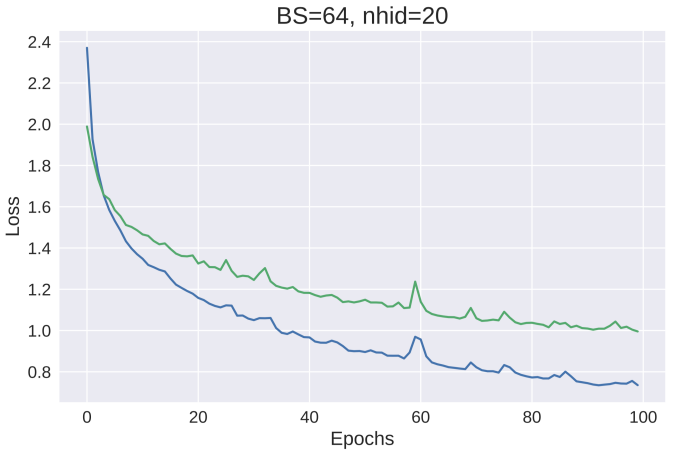
Problem 2.2

Train the GRU RNN in the "Training - RNN decoder" section. How do the results look, qualitatively? Does the model do better for certain types of words than others?

Solution

When training the GRU RNN we get the following outputs:

```
Epoch:  60 | Train loss: 0.957 | Val loss: 1.139 | Gen: ehay ailway oningionssay ishay onthincowhay
Epoch:  61 | Train loss: 0.875 | Val loss: 1.095 | Gen: ehay ailway onintondionsway issay orthindday
Epoch:  62 | Train loss: 0.846 | Val loss: 1.080 | Gen: ehay ailway onintiondsday issay ortudionway
Epoch:  63 | Train loss: 0.836 | Val loss: 1.073 | Gen: ehay airway onintiondionway issay ortudionway
Epoch:  64 | Train loss: 0.830 | Val loss: 1.068 | Gen: ehay airway onintiondionway issay ordturingway
Epoch:  65 | Train loss: 0.823 | Val loss: 1.065 | Gen: ehay airway onintiondionway issay ortudionway
Epoch:  66 | Train loss: 0.819 | Val loss: 1.065 | Gen: ehay airway onintiondionway issay ordturingway
Epoch:  67 | Train loss: 0.816 | Val loss: 1.058 | Gen: ehay airway onintiondionway issay ortudionway
Epoch:  68 | Train loss: 0.813 | Val loss: 1.066 | Gen: ehay airway onintiondionway issay ortudionway
Epoch:  69 | Train loss: 0.845 | Val loss: 1.109 | Gen: ehay airway onintononedway issay ortudionway
Epoch:  70 | Train loss: 0.822 | Val loss: 1.060 | Gen: ehay airway oningionstay issay ortudingway
Epoch:  71 | Train loss: 0.808 | Val loss: 1.047 | Gen: ehay airway onintiondionway issay ortudionway
Epoch:  72 | Train loss: 0.803 | Val loss: 1.049 | Gen: ehhay airway oningionstay issay ordurionway
Epoch:  73 | Train loss: 0.803 | Val loss: 1.053 | Gen: ehay airway onintiondionway ishay ortudingway
Epoch:  74 | Train loss: 0.796 | Val loss: 1.049 | Gen: ehhay airway oningionstay issay ortudionway
Epoch:  75 | Train loss: 0.833 | Val loss: 1.091 | Gen: ehay airway oningionsway issay ortudingway
Epoch:  76 | Train loss: 0.822 | Val loss: 1.063 | Gen: ehay airway oningionsway ishay onturedway
Epoch:  77 | Train loss: 0.797 | Val loss: 1.040 | Gen: ehay airway oningionstay issay onturedway
Epoch:  78 | Train loss: 0.785 | Val loss: 1.031 | Gen: ehay airway oningionsway issay orduringway
Epoch:  79 | Train loss: 0.778 | Val loss: 1.037 | Gen: ehay airway oningionstay ishay onturedway
Epoch:  80 | Train loss: 0.773 | Val loss: 1.038 | Gen: ehay airway oningionstay issway ordudedway
Epoch:  81 | Train loss: 0.775 | Val loss: 1.032 | Gen: ehay airway onintiondionway issay ortudingway
Epoch:  82 | Train loss: 0.768 | Val loss: 1.028 | Gen: ehay airway oningionstay issay ortudingway
Epoch:  83 | Train loss: 0.768 | Val loss: 1.016 | Gen: ehay airway oningionstay issway ordurday
Epoch:  84 | Train loss: 0.784 | Val loss: 1.044 | Gen: ehtray airway oningionstay ishay ordudedway
Epoch:  85 | Train loss: 0.775 | Val loss: 1.032 | Gen: ehay airway oningionstay issway ordurday
Epoch:  86 | Train loss: 0.801 | Val loss: 1.037 | Gen: ehay airway oningionstay ishay ordudityway
Epoch:  87 | Train loss: 0.779 | Val loss: 1.016 | Gen: ehay airway oningionteway isswray orduditay
Epoch:  88 | Train loss: 0.754 | Val loss: 1.023 | Gen: ehay airway oningionstay ishay ordudedway
Epoch:  89 | Train loss: 0.749 | Val loss: 1.012 | Gen: ehay airway onintonionway ishay ortudingway
Epoch:  90 | Train loss: 0.745 | Val loss: 1.010 | Gen: ehay airway oningionstay isswray ordurday
Epoch:  91 | Train loss: 0.738 | Val loss: 1.004 | Gen: ehay airway oningionstay ishay ordudedway
Epoch:  92 | Train loss: 0.735 | Val loss: 1.009 | Gen: ehay airway oningionstay ishay ordurday
Epoch:  93 | Train loss: 0.738 | Val loss: 1.009 | Gen: ehay airway onindiondedway ishay ortudingway
Epoch:  94 | Train loss: 0.740 | Val loss: 1.023 | Gen: ehay airway oningionstay ishay orduditay
Epoch:  95 | Train loss: 0.747 | Val loss: 1.044 | Gen: ehay airway onintonedday ishay ordurday
Epoch:  96 | Train loss: 0.743 | Val loss: 1.012 | Gen: ehray airway onindingpray ishay ordudedway
Epoch:  97 | Train loss: 0.743 | Val loss: 1.019 | Gen: ehay airway oningionsway ishay onturedway
Epoch:  98 | Train loss: 0.756 | Val loss: 1.004 | Gen: ehray airway onintingorday ishay ordurday
Epoch:  99 | Train loss: 0.735 | Val loss: 0.996 | Gen: ehray airway onindingstay isswray ordudedway
source:         the air conditioning is working
translated:     ehray airway onindingstay isswray ordudedway
```



As shown above, the results initially seem somewhat reasonable. The loss plot does not seem to indicate any hint of overfitting as validation loss (green), while larger than training loss (blue) as expected, does not increase toward the end at epoch 100.

However, this model produces sequences that are mostly incorrect, at least in part. Even if most letters of

most words seem to be correctly remembered there is still a significant amount of forgotten/extra characters and incorrect appending suffixes are common.

Moreover, it is worth noting that the model does indeed do better for certain types of words than others. Longer words seem to be more incorrectly remembered than shorter words. In addition, words that start with a sound composed of multiple letters like "the" are usually not well translated as the suffix appended is usually incorrect.

Problem 2.3

Use this model to translate words in the next notebook cell using `translate_sentence` function. Try a few of your own words by changing the variable `TEST_SENTENCE`. Which failure modes can you identify?

Solution

By selecting a sentence with a word with the ending "-way" and a variety of other interesting features (e.g. hyphenated words) we get the following output:

Source:	Translated:
the british born sixty-nine year	ehray itingstay ixbeshay inenay yeay
old is known as the	oldway issway ondway ashay ehray
godfather of the strain of	ofgathedsay ofway ehray aintlyway ofway
artificial intelligence called neural	artinesstay iningelleshay alledsay uerablywhay
networks	ettountay
or neural nets which involves	orray uerablywhay ettbay ickerway inatessorway
setting up computer systems to	ettingsay upray opurepray ysssssay otay
mimic the human brain allowing	impiedway ehray unateway aingray alnigssay
them to learn it is	esthay otay earybay itway issway
some experts say going to	omesay expestray away-yearay oingnay otay
radically transform our lives already	ailnatedway anteportway ourway ivelysay arlenay
is actually the way electricity	issway achulatedway ehray ayfay elentlesay
did	idday

The above output reveals the following weaknesses:
- all words (short and long) with a starting sounds composed of multiple letters (such as "th") are usually poorly translated
- longer words are usually less well translated than shorter words as the network's long range dependencies become too hard to overcome (e.g. "intelligence")
- words that contain the characters "way", even short ones, are usually poorly translated (e.g. "way" $\rightarrow$ "issway")
- when an error arises in translation this can cause a catastrophic unrecoverable translation failure (e.g. "systems" $\rightarrow$ "ysssssay")
- the dash messes up a lot of things
- the number of characters is not always correct

Problem 3.1

Implement the additive attention mechanism. Fill in the `forward` methods of the `AdditiveAttention` class. Use the `self.softmax` function in the forward pass of the `AdditiveAttention` class to normalize the weights.

Solution

See code attached in file `nmt.ipynb`.

Problem 3.2

Fill in the forward method of the `RNNAttentionDecoder` class, to implement the interface shown in Figure 4.

Solution

See code attached in file `nmt.ipynb`.

Problem 3.3

Train the Attention RNN in the "Training - RNN attention decoder" section. How do the results compare to RNN decoder without attention for certain type of words? Can you identity any failure mode? How does the training speed compare? Why?

Solution

The translations are in general far better for the RNN with attention than for the RNN without attention. Quantitatively, this is reflected in the loss at epoch 99 which was 0.996 for the RNN without attention but is now 0.066 for the RNN with attention. Qualitatively, for certain types of words the RNN with attention correctly translates while the RNN without attention does not, notably for longer words.

Some failure modes of this RNN with attention can be identified by attempting to translate the custom text of problem 2.3, but this time with attention. This yields the following:

Source:	Translated:
the british born sixty-nine year	ethay itishbay-ornbay ixtysay inenay yearay
old is known as the	oldway isway ownknay asway ethay
godfather of the strain of	odfathergay ofway ethay ainstray ofway
artificial intelligence called neural	artificialway intelligenceway alledcay euralnay
networks	etworksnay
or neural nets which involves	orway euralnay etsnay ichwhay involvesway
setting up computer systems to	ettingsay upway omputercay ystemssay otay
mimic the human brain allowing	imicmay ethay umanhay ainbray allowingway
them to learn it is	emthay otay earnlay itway isway
some experts say going to	omesay expertsway aysay oinggay otay
radically transform our lives already	adicallyray ansformtray ourway iveslay alreadyway
is actually the way electricity	isway actuallyway ethay ayway electricityway
did	idday

From the above we see that most failure modes of the RNN are no longer present, yet the following failure modes can be still seen:
- the errors handling the "-" character in the encoder still persist (e.g. bad translation of "sixty-nine").
- errors handling the "-" character at the decoder still persist (e.g. bad translation of "british born")

The training speed is much slower for the RNN with attention (12min21s for 100 epochs) than for the RNN without attention (3min58s for 100 epochs), as expected. This is no surprise as adding attention requires the network to recompute context vectors at every time step. Computing these context vectors adds complexity to training resulting in a $t^2 \times k^2 \times d$ training complexity for RNNs with attention as opposed to $t \times k^2 \times d$ for regular RNNs, as discussed in Prof. Grosse's lecture notes.

Problem 3.4

Implement the scaled dot-product attention mechanism. Fill in the `__init__` and forward methods of the `ScaledDotAttention` class. Train the Attention RNN using scaled dot-product attention in the "Training - RNN scaled dot-product attention decoder" section. How do the results and training speed compare to the additive attention? Why is there such different?

Solution

See code attached in file `nmt.ipynb`.

Both quantitatively and qualitatively, the results from the scaled dot-product attention are quite good but not as good as the additive attention. Indeed, we can observe a loss of around 0.200 for the scaled dot-product attention while we had a loss of around 0.066 for the additive attention. This difference is as expected, and is likely due to the fact that additive attention (can be non-linear) is more powerful than scaled dot-product attention (always linear). This is reflected in the slightly less good translations that we can see from the output of the scaled dot-product attention, as compared with the additive attention output translations.

From the original paper, we expect that the scaled dot-product attention training should be faster than the additive attention. This is because, while both these forms of attention have similar theoretical complexity, dot-product attention is much faster and more space-efficient in practice as it can be implemented using highly optimized matrix multiplication code. In this assignment the scaled dot-product attention took 12min47s to train over 100 epochs compared to the 12min21s for the additive attention. These speeds are similar because we are not utilising the highly optimised matrix multiplication code tricks in our RNN that we will use in the Transformer model in the subsequent section (because of the sequential nature of RNNs).

Problem 4.1

What are the advantages and disadvantages of using additive attention vs scaled dot-product attention? List one advantage and one disadvantage for each method.

Solution

One advantage of using additive attention vs dot-product attention is that it generally outperforms dot product attention (without scaling) for larger values of d_k. However, this can be counteracted by scaling the dot product attention to avoid pushing the softmax function into regions where it has extremely small gradients. Even with this scaling though, we still find that additive attention, which can be non-linear, is more powerful than scaled dot product attention which is always linear.

One disadvantage of using additive attention vs scaled dot-product attention is that it takes longer to train than scaled dot-product attention. This is because, while both these forms of attention have similar theoretical complexity, dot-product attention is much faster and more space-efficient in practice as it can be implemented using highly optimized matrix multiplication code (as explained in the Transformer paper).

Problem 4.2

Fill in the `forward` method in the `CausalScaledDotAttention`.
It will be mostly the same as the `ScaledDotAttention` class. The additional computation is to mask out the attention to the future time steps. You will need to add `self.neg_inf` to some of the entries in the unnormalized attention weights.

Solution

See code attached in file `nmt.ipynb`.

Problem 4.3

Fill in the `forward` method of the `TransformerDecoder` class, to implement the interface shown in Figure 5. Train the transformer in the "Training - Transformer decoder" section. How do the translation results compare to the previous decoders? How does the training speed compare?

Solution

See code attached in file `nmt.ipynb`.

Both quantitatively and qualitatively, the results from the Transformer are the best so far as they outperform the RNN with additive attention and the RNN with scaled dot attention. Indeed, we can observe a loss of around 0.031 for the Transformer while we had a loss of around 0.066 for the RNN with additive attention.

As for the training time, we expect from the original Transformer paper, that the scaled dot-product Transformer should be faster than the RNNs with attention. This is because we can now leverage the highly optimized matrix multiplication code to better vectorise the operations, as discussed in the paper. In this assignment the Transformer took 7min06s to train over 100 epochs compared to the 12min21s for the additive attention. This confirms expectations as the Transformer was faster to train.

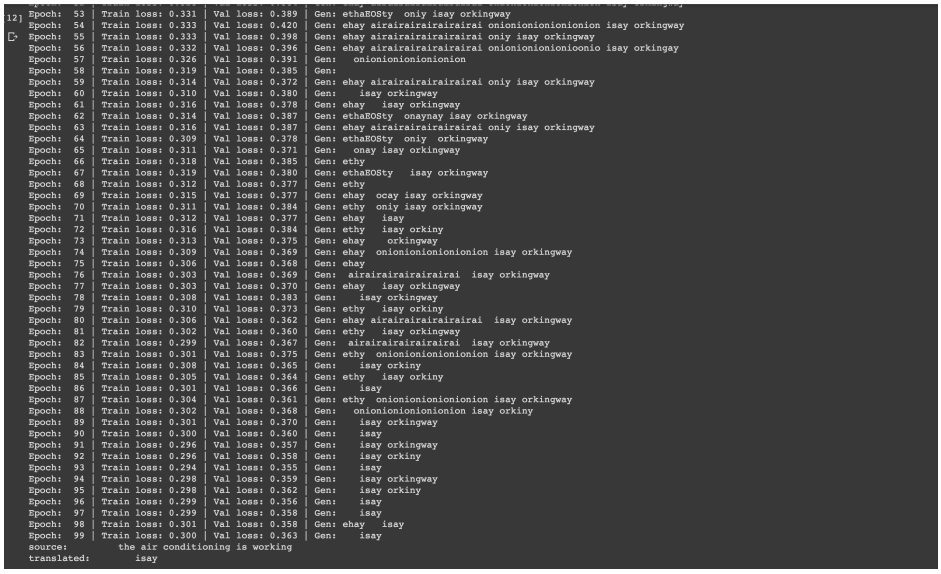
Problem 4.4

Modify the transformer decoder `__init__` to use non-causal attention for both self attention and encoder

attention. What do you observe when training this modified transformer? How do the results compare with the causal model? Why?

Solution

When training this modified non-causal transformer the results are significantly worse than those for the causal one. The following output shows what the trained non-causal transformer does during translation:



Indeed, from the output above we can tell that the performance is significantly impacted. This is because attending in a non-causal way to the input (i.e. not controlling the masking properly) destroys the auto-regressive nature of the Transformer. This comes, as expected, at a significant cost both in terms of the quantitative loss and in terms of the logic of the qualitative output translations.

Problem 4.5

In the lecture, we mentioned the transformer encoder is not able to learn the ordering of its inputs without the explicit positional encoding. Why does our simple transformer decoder work without the positional encoding?

Solution

Positional encoding is an approach to machine learning whereby a positional encoding matrix has values defined by equations that are variable on the position of the word in the sentence. This is usually implemented using a sinusoidal function or something of the like to encode positional information. When added to the embedding matrix, each character embedding is altered in a way specific to its position.

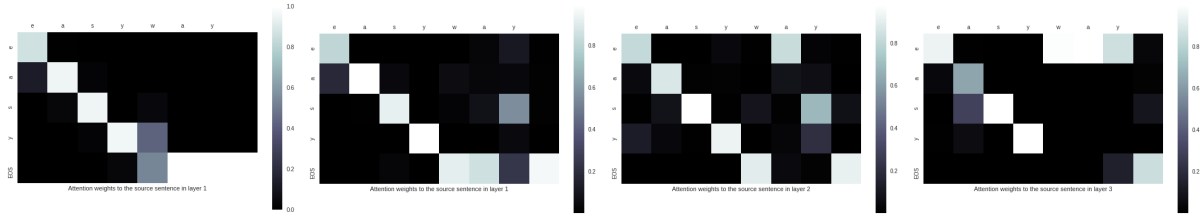
Here, we make no such use of explicit position embedding yet the Transformer model learns to pay attention to the right word. Indeed, this is possible because we have learned the attention weights such that at each step the attention is focused on the most beneficial character or characters. Because of the sequential nature of the task, the final attention of the Transformer here works because we have masked a triangular matrix such that we cannot get information from future states in a non-causal way. This results in a working transformer without explicitly working using positional encoding.

Problem 5.1

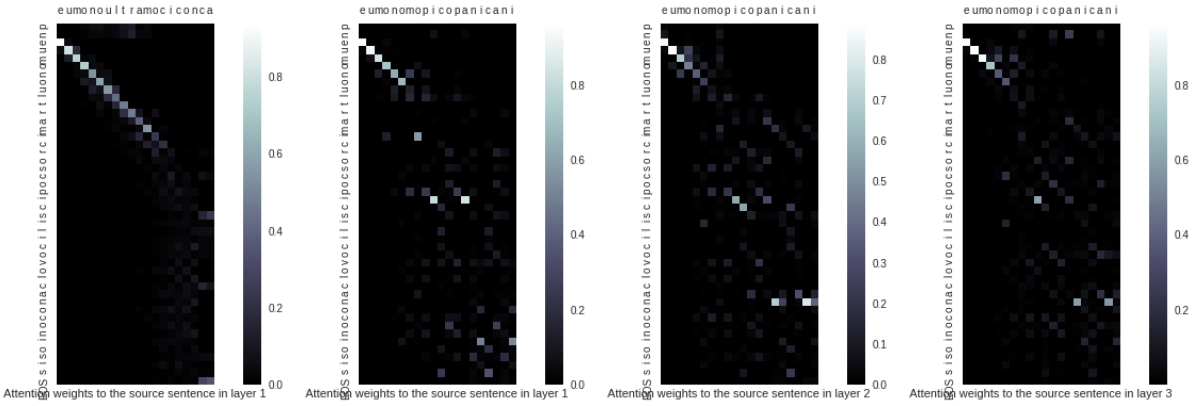
Visualize different attention models using your own word by modifying `TEST_WORD_ATTN`. Since the model operates at the character-level, the input doesn't even have to be a real word in the dictionary. You can be creative! You should examine the generated attention maps. Try to find failure cases, and hypothesize about why they occur. Include attention maps for both success and failure cases in your writeup, along with your hypothesis about why the models succeeds or fails.

Solution

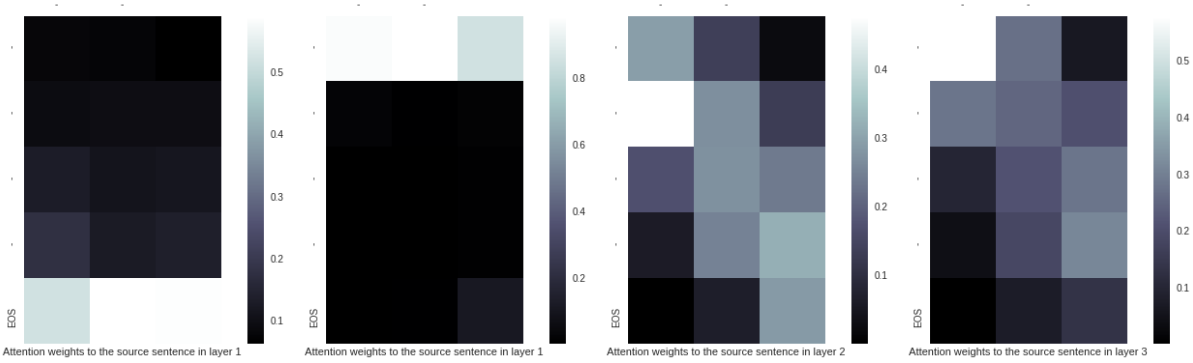
To visualise different attention models, multiple words were used as inputs. The most notable and interesting ones are as follows (1 baseline, 4 failure cases):
- "easy": a simple and short word for which the model successfully translates to Pig-Latin (to use as a baseline)



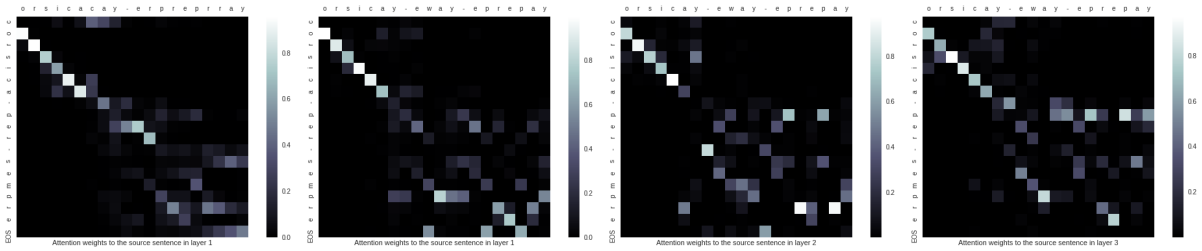
- "pneumonoultramicroscopicsilicovolcanoconiosis": the longest word in the english language (to test memory)



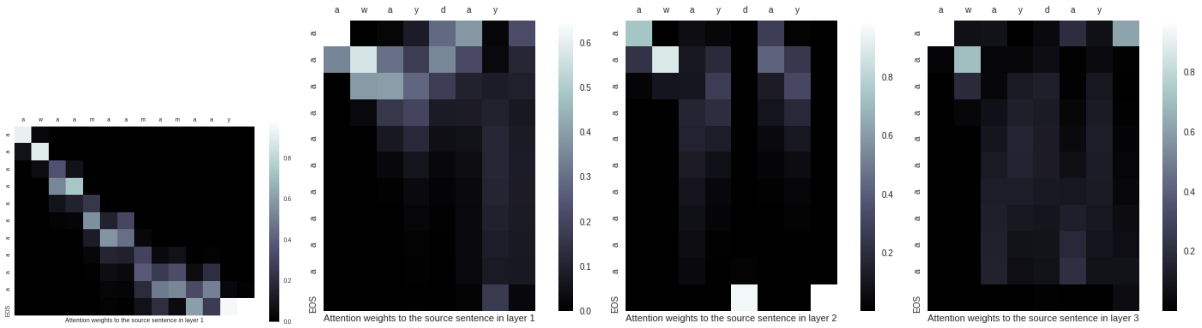
- "—-": a short sequence of four dash characters (to test foreign structures without requiring much memorisation)



- "corsica-per-sempre": three short words from another language (Corsican) joined together with dashes (to test other languages and foreign structures)



- "aaaaaaaaaa": a relatively long and repetitive sequence (to see which character is being attended to for long repetitions)



From the above visualisations, we can notice the following interesting dynamics and hypotheses:
- simple words succeed as the attention is payed diagonally for the input area to be remembered (i.e. before the appending suffix), as seen in the "easy" baseline maps
- very long words start off with a nice diagonal attention as desired but eventually the attention spreads all over the input space leading to poor translations, as seen in the "pneumonoultramicroscopicsilicovolcanoconiosis" maps
- words with dashes start to fail as soon as the first dash is introduced (breaks the diagonality of the attention), as seen in the "—-" and "corsica-per-sempre" maps
- repetitive words are fairly well handled by the RNN which pays attention diagonally, whereas the Transformer just keeps its attention on only a few of the input characters (as they are all the same), as seen in the "aaaaaaaaaa" maps