How to Visualize Your Recurrent Neural Network with Attention in Keras

A technical discussion and tutorial

Zafarali Ahmed
Jun 29, 2017 · 13 min read

Neural networks are taking over every part of our lives. In particular — thanks to deep learning — Siri can fetch you a taxi using your voice; and Google can enhance and organize your photos automagically. Here at Datalogue, we use deep learning to structurally and semantically understand data, allowing us to prepare it for use automatically.

Neural networks are massively successful in the domain of computer vision. Specifically, convolutional neural networks(CNNs) take images and extract relevant features from them by using small windows that travel over the image. This understanding can be leveraged to identify objects from your camera (Google Lens) and, in the future, even drive your car (NVIDIA).

The analogous neural network for text data is the recurrent neural network (RNN). This kind of network is designed for sequential data and applies the same function to the words or characters of the text. These models are successful in translation (Google Translate), speech recognition (Cortana) and language generation.

Dr. Yoshua Bengio from Université de Montréal believes that language is going to be the next challenge for neural networks. At Datalogue, we deal with tons of text data, and we are interested in helping the community solve this challenge.

In this tutorial, we will write an RNN in Keras that can translate *human* dates (“November 5, 2016”, “5th November 2016”) into a standard format (“2016–11–05”). In particular, we want to gain some intuition into how the neural network did this. We will leverage the concept of *attention* to generate a map (like that shown in Figure 1) that shows which input characters were important in predicting the output characters.

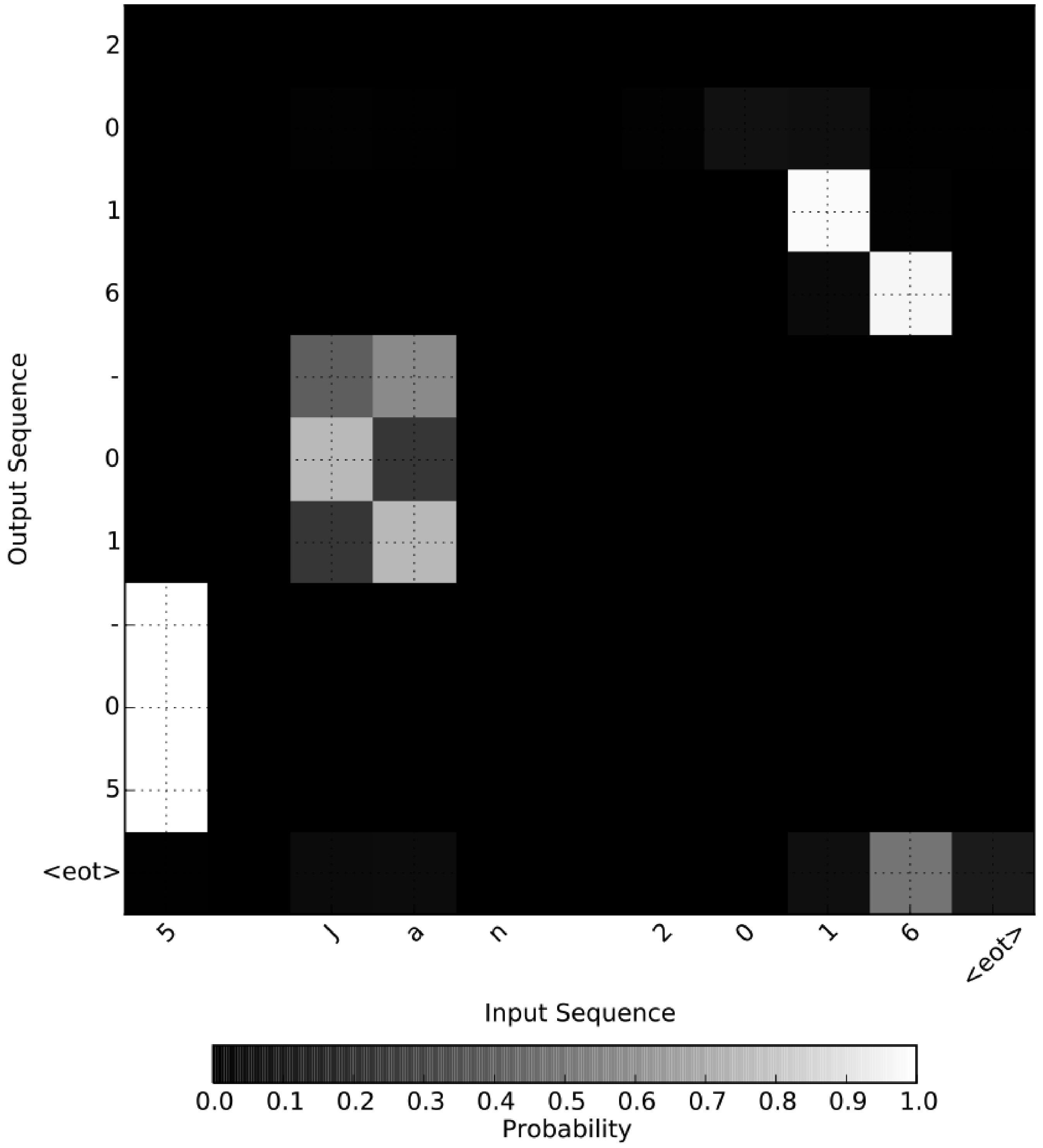


Figure 1: Attention map for the freeform date ″5 Jan 2016″. We can see that the neural network used ″16″ to decide that the year was 2016, ″Ja″ to decide that the month was 01 and the first bit of the date to decide the day of the month.

What is in this Tutorial

We start off with some technical background material to get everyone on the same page and then move on to programming this thing! Throughout the tutorial, I will provide links to more advanced content.

If you want to directly jump to the code:

datalogue/keras-attention

keras-attention - Visualizing RNNs using the attention mechanism

github.com

What you need to know

To be able to jump into the coding part of this tutorial, it would be best if you have some familiarity with Python and Keras. You should have some familiarity of linear algebra, in particular that neural networks are just a few weight matrices with some non-linearities applied to it.

The intuition of RNNs and seq2seq models will be explained below.

Recurrent Neural Networks (RNN)

An RNN is a function that applies the same transformation (known as the RNN ce*ll* or *step*) to every element of a sequence. The output of an RNN *layer* is the output of the RNN cell applied on each element of the sequence. In the case of text, these are usually successive words or characters. In addition to this, RNN cells hold an internal memory that summarize the history of the sequence it has seen so far.

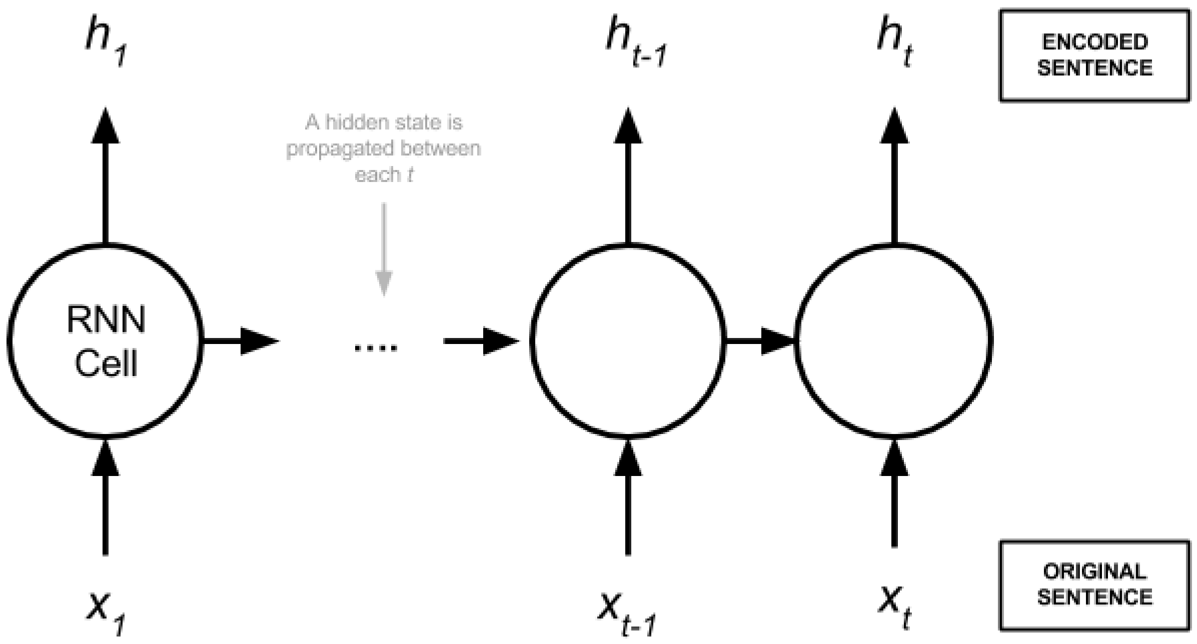


Figure 2: An example of an RNN Layer. The RNN Cell is applied to each element xi in the original sequence to get the corresponding element hi in the output sequence. The RNN cell uses a previous hidden state (propagated between the RNN cells) and the sequence element xi to do its calculations.

The output of the RNN *layer* is an encoded sequence, `h`, that can be manipulated and passed into another network. RNNs are incredibly flexible in their inputs and outputs:

1. Many-to-One: use the complete input sequence to make a single prediction `h`.
2. One-to-Many: transforming a single input to generate a sequence `h`.
3. Many-to-Many: transforming the entire input sequence into another sequence.

In theory, the sequences in our training data need not be of the same length. In practice, we pad or truncate them to be of the same length to take advantage of the static computational graph in Tensorflow.

We will focus on #3 "Many-to-Many" also known as sequence-to-sequence (**seq2seq**).

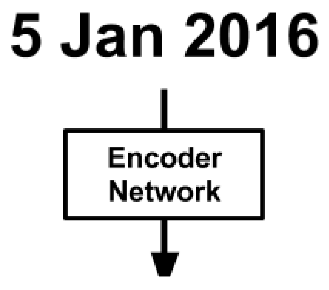
Long sequences can be difficult to learn from in RNNs due to instabilities in gradient calculations during training. To solve this, the RNN cell is replaced by a *gated cell* like the gated recurrent unit (GRU) or the long-short term memory cell (LSTM). To learn more about LSTMs and gated units, I highly recommend reading Christopher Olah's blog (it's where I first started understanding the RNN cell). From now on, whenever we talk about RNNs, we are talking about a gated cell. Since Olah's blog gives an intuitive introduction to LSTMs, we will use that.

A General Framework for seq2seq: The Encoder-Decoder Setup

Almost all neural network approaches to solving the seq2seq problem involve:

1. *Encoding* the input sentences into some abstract representation.
2. *Manipulating* this encoding.
3. *Decoding* it to our target sequence.

Our encoders and decoders can be any kind and combination of neural networks. In practice, most people use RNNs for both the encoders and decoders.



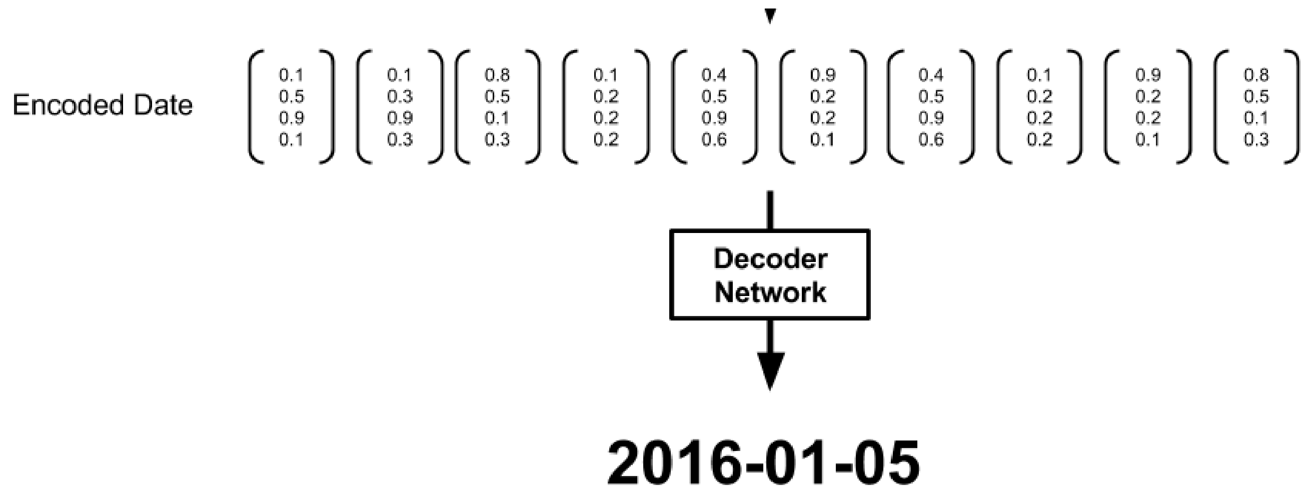


Figure 3: Set up of the encoder-decoder architecture. The encoder network processes the input sequence into an encoded sequence which is subsequently used by the decoder network to produce the output.

Figure 3 shows a simple encoder-decoder setup. The encoding step usually produces a sequence of vectors, `h`, corresponding to the sequence of characters in the input date, `x`. In an RNN encoder, each vector is generated by integrating information from the past sequence of vectors.

Before `h` is passed onto the decoder, we may choose to manipulate it in preparation for the decoder. For example, we might choose to only use the last encoding, as shown in Figure 4, since in theory, it is a summary of the whole sequence.

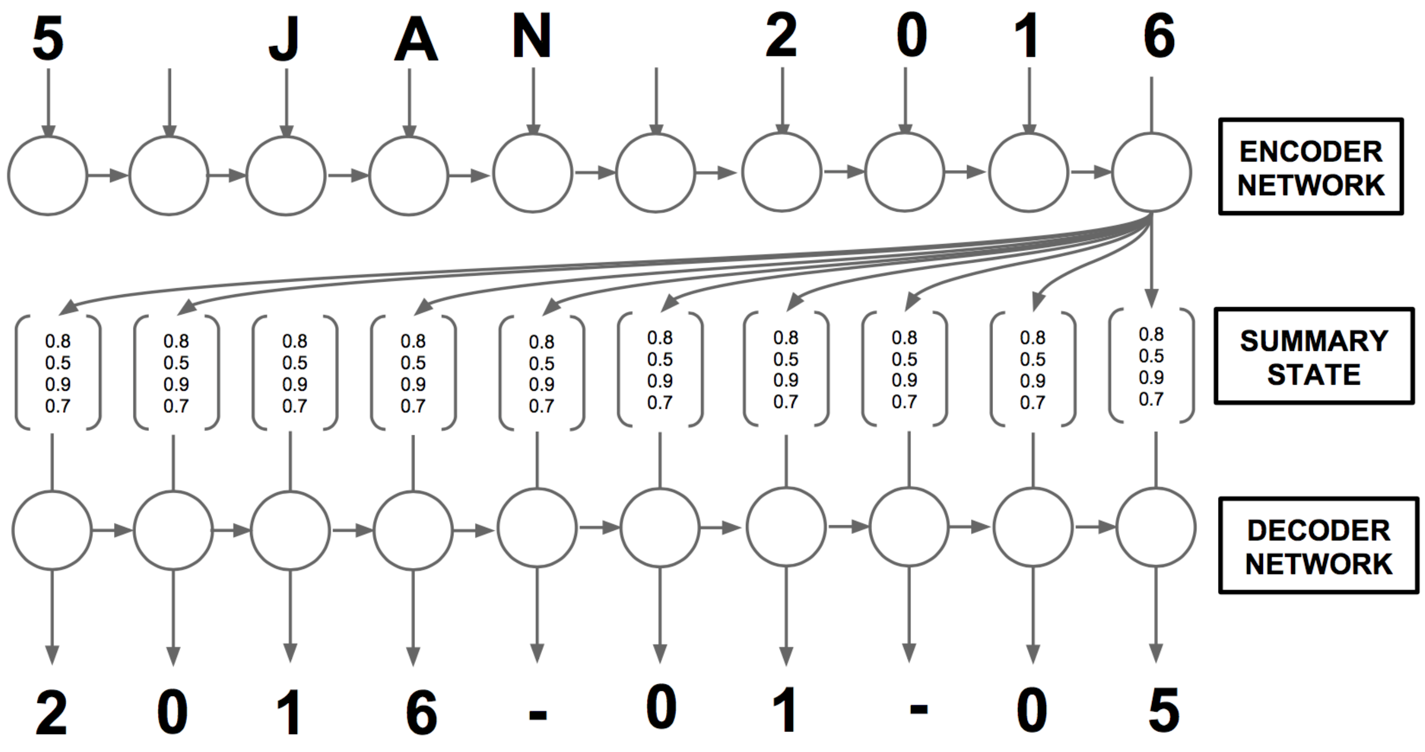


Figure 4: Use of a summary state in the encoder-decoder architecture.

Intuitively, this is similar to summarizing the whole input date into a single representation and then trying to decode that. While there may be enough information in this summary state for a classification problem like detecting sentiment (Many-to-One), it might be insufficient for an effective translation where it helps to consider the full sequence of hidden states (Figure 5).

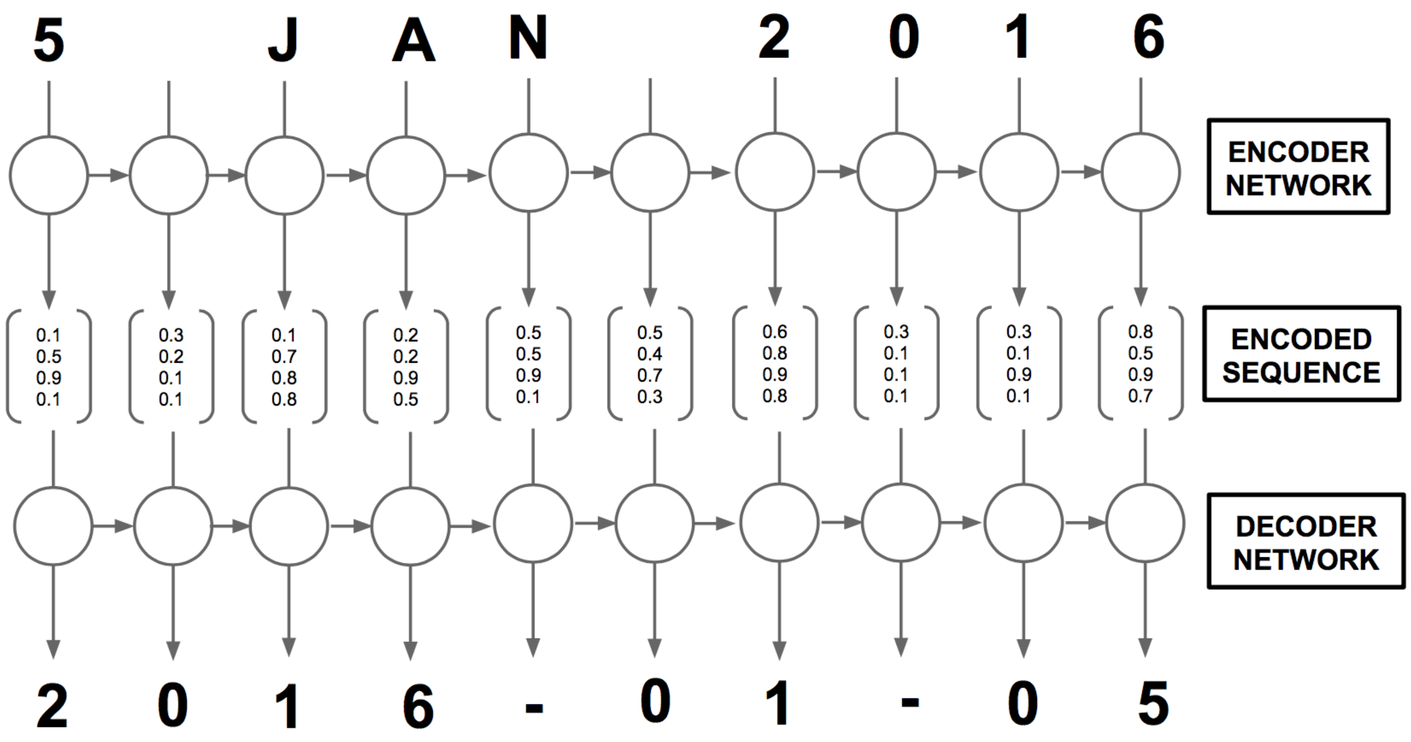


Figure 5: Use of the complete encoded sequence in the decoder network.

However, this is not how humans translate dates: We do not read the whole text and then independently write down the translation at each character. Intuitively, a person would understand that the characters "Jan" correspond to 1st month, "5" corresponds to the day and "2016" corresponds to the year. As we have already seen in Figure 1, this idea known as *attention* can be captured by RNNs and has been applied successfully in caption generation (Xu et al. 2015), speech recognition (Chan et al. 2015) and indeed machine translation (Bahdanau et al. 2014). Most importantly, they produce *interpretable models*.

A slightly more visual example of how the attention mechanism works comes from the Xu et. al, 2015 paper (Figure 6). In the most complicated example of the girl with the teddy, we can see that when generating the word "girl", the attention mechanism correctly focuses on the girl and not the teddy! Pretty brilliant. Not only does this make for pretty pictures, but it also let the authors diagnose problems in the model.

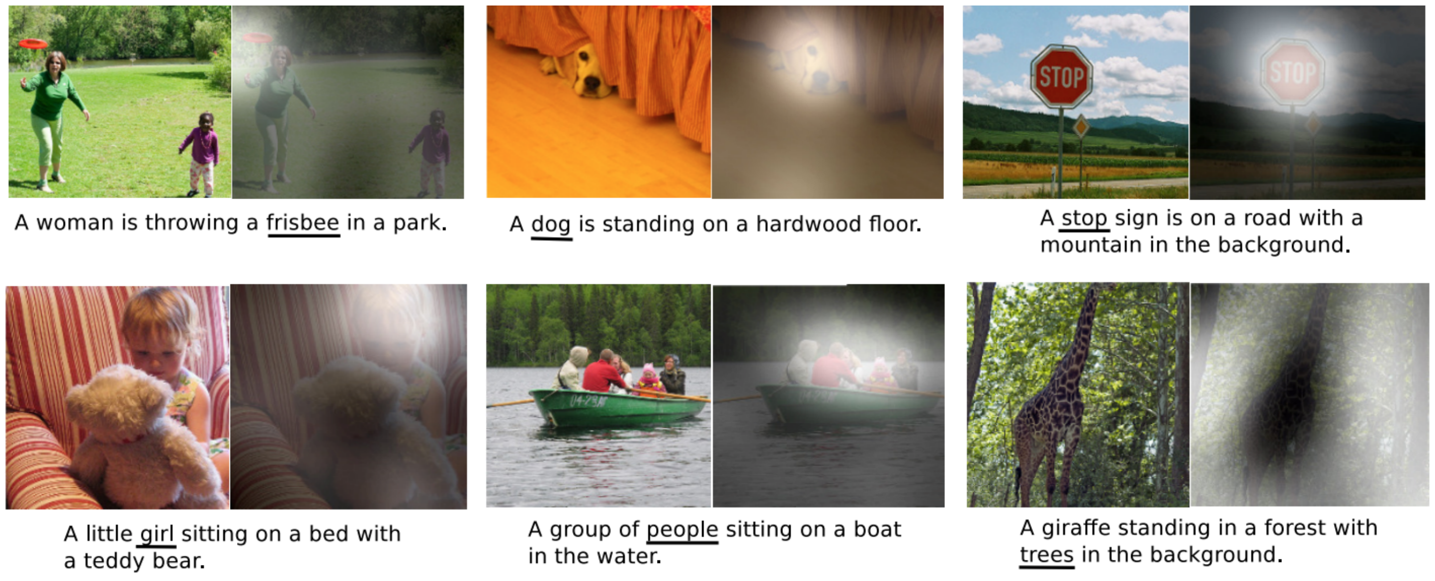

A woman is throwing a frisbee in a park.
A dog is standing on a hardwood floor.
A stop sign is on a road with a mountain in the background.
A little girl sitting on a bed with a teddy bear.
A group of people sitting on a boat in the water.
A giraffe standing in a forest with trees in the background.

Figure 6: Attending to objects in an image during caption generation. The white regions indicate where the attention mechanism focused on during the generation of the underlined word. From Xu, Kelvin, et al. "Show, attend and tell: Neural image caption generation with visual attention." International Conference on Machine Learning. 2015.

The creators of SpaCy have an in-depth overview of the encoder-attention-decoder paradigm. For other ways of modifying RNNs you can head over to this Distill article.

This tutorial will feature a single *bidirectional* LSTM as the encoder and the *attention* decoder (more on this later). More concretely, we will implement a simpler version of the model presented in *"Neural machine translation by jointly learning to align and translate." (Bahdanau, Dzmitry, Kyunghyun Cho, and Yoshua Bengio.2014)*. I'll walk through some of the math, but I invite you to jump into the appendices of the paper to get your hands dirty!

Now that we have learned about RNNs and the intuition behind the attention mechanism, let us learn how to implement it and subsequently obtain some nice visualizations. All code for subsequent sections is provided at `datalogue/keras-attention`. The complete implementation of the model below is in `/models/NMT.py`

The Encoder

Since we are trying to learn about AttentionRNNs, we will skip implementing our own vanilla RNN (LSTM) and use the one that ships with Keras. We can invoke it using:

```
BLSTM = Bidirectional(LSTM(encoder_units, return_sequences=True))
```

The parameter `encoder_units` is the size of the weight matrix. We use `return_sequences=True` here because we'd like to access the complete encoded sequence rather than the final summary state as described above.

Our BLSTM will consume the *characters* in the input sentence `x=(x1,...,xT)` and output an encoded sequence `h=(h1,...,hT)` where `T` is the number of characters in the date. Note that this is slightly different to the Bahdanau et al. paper where the sentence is a collection of words rather than characters. We also do not refer to the encoded sequence as the *annotations* like in the original paper.

The Decoder

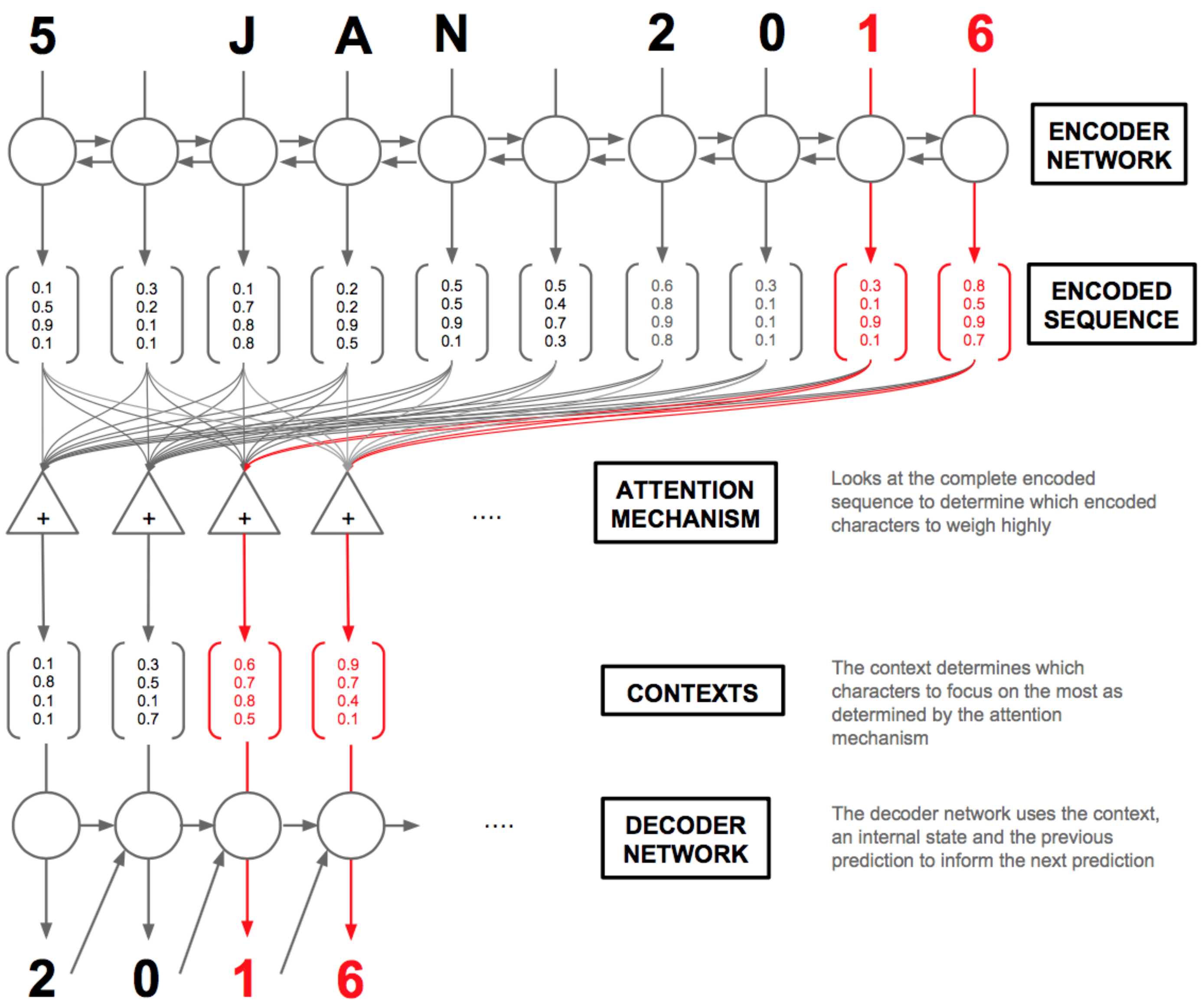


Figure 7: Overview of the Attention mechanism in an Encoder-Decoder setup. The attention mechanism creates context vectors. The decoder network uses these context vectors as well as the previous prediction to make the next one. The red arrows highlight which characters the attention mechanism will weigh highly in producing the output characters "1" and "6".

Now for the interesting part: the decoder. For any given character at position `t` in the sequence, our decoder accepts the encoded sequence `h=(h1,...,hT)` as well as the previous hidden state *st-1* (shared within the decoder cell) and character *yt-1* . Our decoder layer will output `y=(y1,...,yT)` (the characters in the standardized date). Our overall architecture is summarized in Figure 7.

Equations

As shown in Figure 6, the decoder is quite complicated. So let's break it down into the steps executed by the decoder cell when trying to predict character `t.` In the following equations, the capital letter variables represent trainable parameters (Note that I have dropped the bias terms for brevity.)

$$e_{j,t} = V_a \cdot \tanh(W_a s_{t-1} + U_a h_j)$$

$$\alpha_{j,t} = \frac{\exp(e_j)}{\sum_{k=1}^{T} \exp(e_k)}$$

Equation 1: A feed-forward neural network that calculates the unnormalized importance of character j in predicting character t. **Equation 2:** The softmax operation that normalizes the probability.

1. Calculate the attention probabilities `α=(α1,…,αT)` based on the encoded sequence and the internal hidden state of the decoder cell, *st-1* . These are shown in Equation 1 and Equation 2.

$$c_t = \sum_{k=1}^{T} \alpha_{k,t} h_k$$

Figure 3: Calculation of the context vector for the t-th character.

2. Calculate the *context* vector which is the weighted sum of the encoded sequence with the attention probabilities. Intuitively, this vector summarizes the importance of the different encoded characters in predicting the *t* -th character.

$$r_t = \sigma(W_r y_{t-1} + U_r s_{t-1} + C_r c_t)$$
$$z_t = \sigma(W_z y_{t-1} + U_z s_{t-1} + C_z c_t)$$
$$\hat{s}_t = \tanh(W_p y_{t-1} + U_p[r_t \circ s_{t-1}] + C_p c_t)$$
$$s_t = (1 - z_t) \circ s_{t-1} + z_t \circ \hat{s}_t$$

Equation 4: Reset gate. **Equation 5:** Update gate. **Equation 6:** Proposal hidden state. **Equation 7:** New hidden state.

3. We then update our hidden state. If you are familiar with the equations of an LSTM cell, these might be ring a bell as the reset gate, `r` , update gate, `z` , and the proposal state. We use the reset gate to control how much information from the previous hidden state `st-1` is used to create a proposal hidden state. The update gate controls how we much of the proposal we use in the new hidden state `st` . (Confused? See step by step walk through of LSTM equations)

$$y_t = \sigma(W_o y_{t-1} + U_o s_t + C_o c_t)$$

Equation 8: A simple neural network to predict the next character.

4. Calculate the `t` -th character using a simple one layer neural network using the context, hidden state, and previous character. This is a modification from the paper which uses a *maxout* layer. Since we are trying to keep things as simple as possible this works fine!

Equations 1–8 are applied for every character in the encoded sequence to produce a decoded sequence `y` which represents the probability of a translated character at each position.

Code

Our custom layer is implemented in `models/custom_recurrent.py` . This part is somewhat complicated in particular because of the manipulations we need to make for vectorized operations acting on the complete encoded sequence. It will make more sense when you think about it. I promise it will become easier the more you look at the equations and the code simultaneously.

A minimal custom Keras layer has to implement a few methods: `__init__`, `compute_ouput_shape`, `build` and `call`. For completeness, we also implement `get_config` which allows you to load the model back into memory easily. In addition to these, a Keras recurrent layer implements a `step` method that will hold all the computations of our cell.

First let us break down boiler-plate layer code:

- `__init__` is what is called when the Layer is first instantiated. It sets functions that will eventually initialize the weights, regularizers, and constraints. Since the output of our layer is a sequence, we hard code `self.return_sequences=True`.
- `build` is called when we run `Model.compile(…)`. Since our model is quite complicated, you can see there are a ton of weights to initialize here. The call `self.add_weight` automatically handles initializing the weights and setting them as trainable within the model. Weights with the subscript `a` are used to calculate the context vector (step 1 and 2). Weights with the subscript `r`, `z`, `p` will calculate the new hidden states from step 3. Finally, weights with the subscript `o` will calculate the output of the layer.
- Some convenience functions are implemented as well: (a) `compute_output_shape` will calculate output shapes for any given input; (b) `get_config` let's us load the model using just a saved file (once we are done training)

Now for the cell logic:

- By default, each execution of the cell only has information from the previous time step. Since we need to access the entire encoded sequence within the cell, we need to save it somewhere. Therefore, we make a simple modification in `call`. The only way I could find to do this was to set the sequence being fed into the cell as `self.x` so that we can access it later:

```
def call(self, x):
    # store the whole sequence so we can "attend" to it at each timestep
    self.x_seq = x

    # apply the a dense layer over the time dimension of the sequence
    # do it here because it doesn't depend on any previous steps
    # and thefore we can save computation time:

    self._uxpb = _time_distributed_dense(self.x_seq, self.U_a, b=self.b_a,
                                         input_dim=self.input_dim,
                                         timesteps=self.timesteps,
                                         output_dim=self.units)

    return super(AttentionDecoder, self).call(x)
```

attention-decoder-call.py hosted with ❤ by GitHub view raw

Now we need to think vectorized: The `_time_distributed_dense` function calculates the last term of Equation 1 for all the elements of the encoded sequence.

- We now walk through the most important part of the code, that is in `step` which executes the cell logic. Recall that `step` is applied to every element of the input sequence.

```
def step(self, x, states):

    # obtain elements of the previous time step.
    ytm, stm = states

    ##    ##    ##    ##    ##    ##    ##    ##    ##

    # equation 1

    # > repeat the hidden state to the length of the sequence
    _stm = K.repeat(stm, self.timesteps)

    # > now multiplty the weight matrix with the
    #   repeated hidden state
    _Wxstm = K.dot(_stm, self.W_a)

    # > calculate the unnormalized probabilities
    et = K.dot(activations.tanh(_Wxstm + self._uxpb),
               K.expand_dims(self.V_a))

    ##    ##    ##    ##    ##    ##    ##    ##    ##

    # equation 2
    at = K.exp(et)
    at_sum = K.sum(at, axis=1)
    at_sum_repeated = K.repeat(at_sum, self.timesteps)
    # vector of size (batchsize, timesteps, 1)
    at /= at_sum_repeated

    ##    ##    ##    ##    ##    ##    ##    ##    ##
```

```

        # equation 3
        context = K.squeeze(
                    K.batch_dot(at,
                                self.x_seq,
                                axes=1),
                    axis=1)

        # ~~~> calculate new hidden state

        # equation 4  (reset gate)
        rt = activations.sigmoid(
            K.dot(ytm, self.W_r)
            + K.dot(stm, self.U_r)
            + K.dot(context, self.C_r)
            + self.b_r)

        # equation 5 (update gate)
        zt = activations.sigmoid(
            K.dot(ytm, self.W_z)
            + K.dot(stm, self.U_z)
            + K.dot(context, self.C_z)
            + self.b_z)

        # equation 6 (proposal state)
        s_tp = activations.tanh(
            K.dot(ytm, self.W_p)
            + K.dot((rt * stm), self.U_p)
            + K.dot(context, self.C_p)
            + self.b_p)

        # equation 7 (new hidden states)
        st = (1-zt)*stm + zt * s_tp

        # equation 8
        # the probability of having each character.
        yt = activations.softmax(
            K.dot(ytm, self.W_o)
            + K.dot(st, self.U_o)
            + K.dot(context, self.C_o)
            + self.b_o)

        # a switch so that we can return the
        # attention for visualizations
        if self.return_probabilities:
            return at, [yt, st]
        else:
            return yt, [yt, st]
```

attention-decoder-step.py hosted with ❤ by GitHub view raw

In this cell we want to access the previous character `ytm` and hidden state `stm` which is obtained from `states` in line 4.

Think vectorized: we manipulate `stm` to repeat it for the number of characters we have in our input sequence.

On lines 11–18 we implement a version of equation 1 that does the calculations on all the characters in the sequence at once.

In lines 24–28 we have implemented Equation 2 in the vectorized form for the whole sequence. We use `repeat` to allow us to divide every time step by the respective sums.

To calculate the context vector, we need to keep in mind that `self.x_seq` and `at` have a "batch dimension" and therefore we need to use `batch_dot` to avoid doing the multiplication over that dimension. The squeeze operation just removes left-over dimensions. This is done in lines 33–37.

The next few lines of code are a more straightforward implementation of equations 4 – 8.

Now we think a bit ahead: We would like to calculate those fancy attention maps in Figure 1. To do this, we need a “toggle” that returns the attention probabilities `at` .

Training

Data

Any good learning problem should have training data. In this case, it’s easy enough thanks to the Faker library which can generate fake dates with ease. I also use the Babel library to generate dates in different languages and formats as inspired by rasmusbergpalm/normalization. The script `data/generate.py` will generate some fake data and I won’t bore you with the details but invite you to poke around or to make it better.

The script also generates a vocabulary that will convert characters into integers so that the neural network can understand it. Also included is a script in `data/reader.py` to read and prepare data for the neural network to consume.

Model

This simple model with a bidirectional LSTM and the decoder we wrote above is implemented in `models/NMT.py` . You can run it using `python run.py` where I have set some default arguments (Readme has more information). I’d recommend training on a machine with a GPU as it can be prohibitively slow on a CPU-only machine.

If you want to skip the training part, I have provided some weights in `weights/`

Visualization

Now the easy part. For the visualizer implemented in `visualizer.py` , we need to load the weights in twice: Once with the predictive model, and the other to obtain the probabilities. Since we implemented the model architecture in `models/NMT.py` we can simply call the function twice:

```
from models.NMT import simpleNMT
```

```
predictive_model = simpleNMT(...)
predictive_model.load_weights(..., return_probabilities=False)

probability_model = simpleNMT(..., return_probabilities=True)
probability_model.load_weights(...)
```

To simply use the implemented visualizer, you can type:

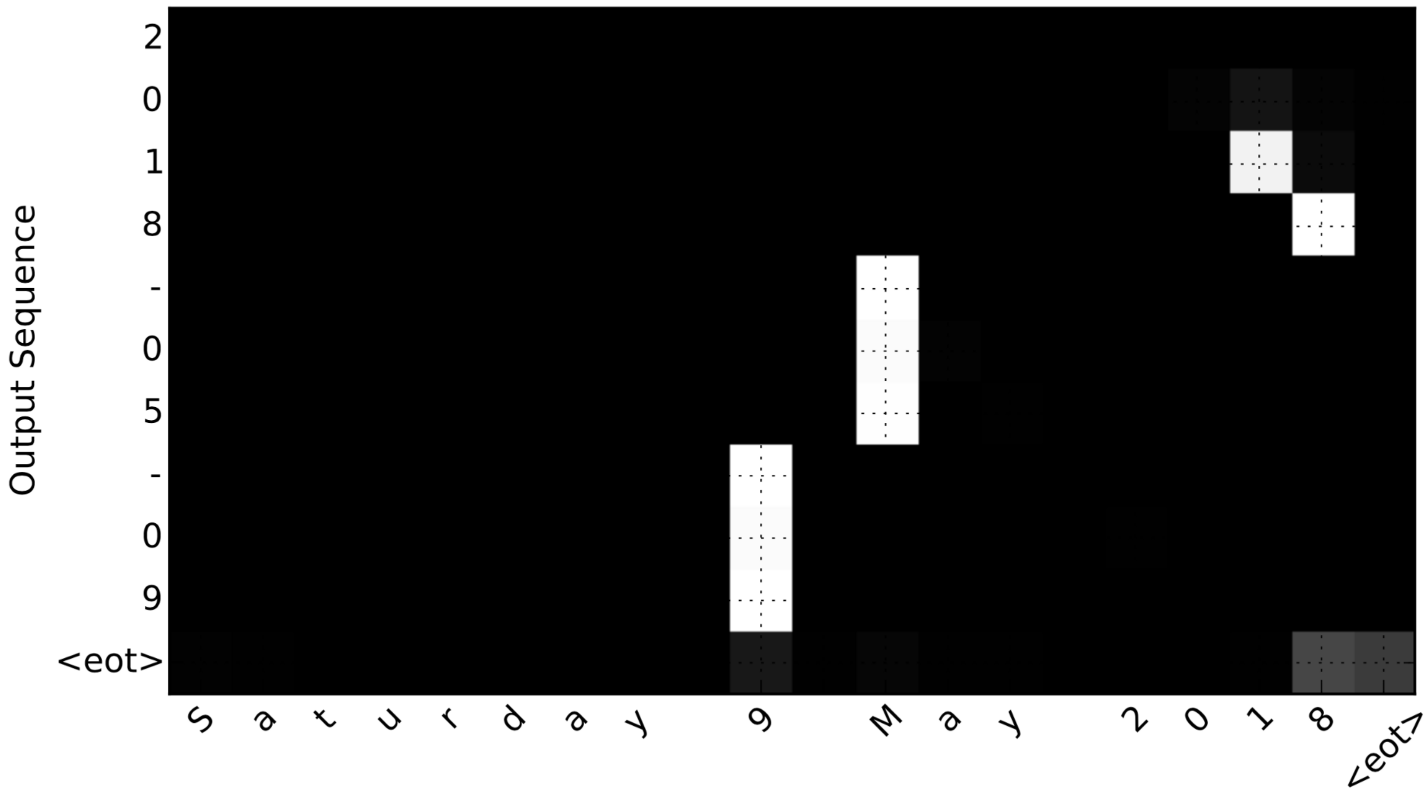
```
python visualizer.py -h
```

to see available command line arguments.

Example visualizations

Let us now examine the attentions we generated from `probability_model`. The `predictive_model` above returns the translated date that you see on the y-axis. On the x-axis is our input date. The map shows us which input characters (on the x-axis) were used in the prediction of the output character on the y-axis. The brighter the white, the more weight that character had. Here are some I thought were quite interesting.

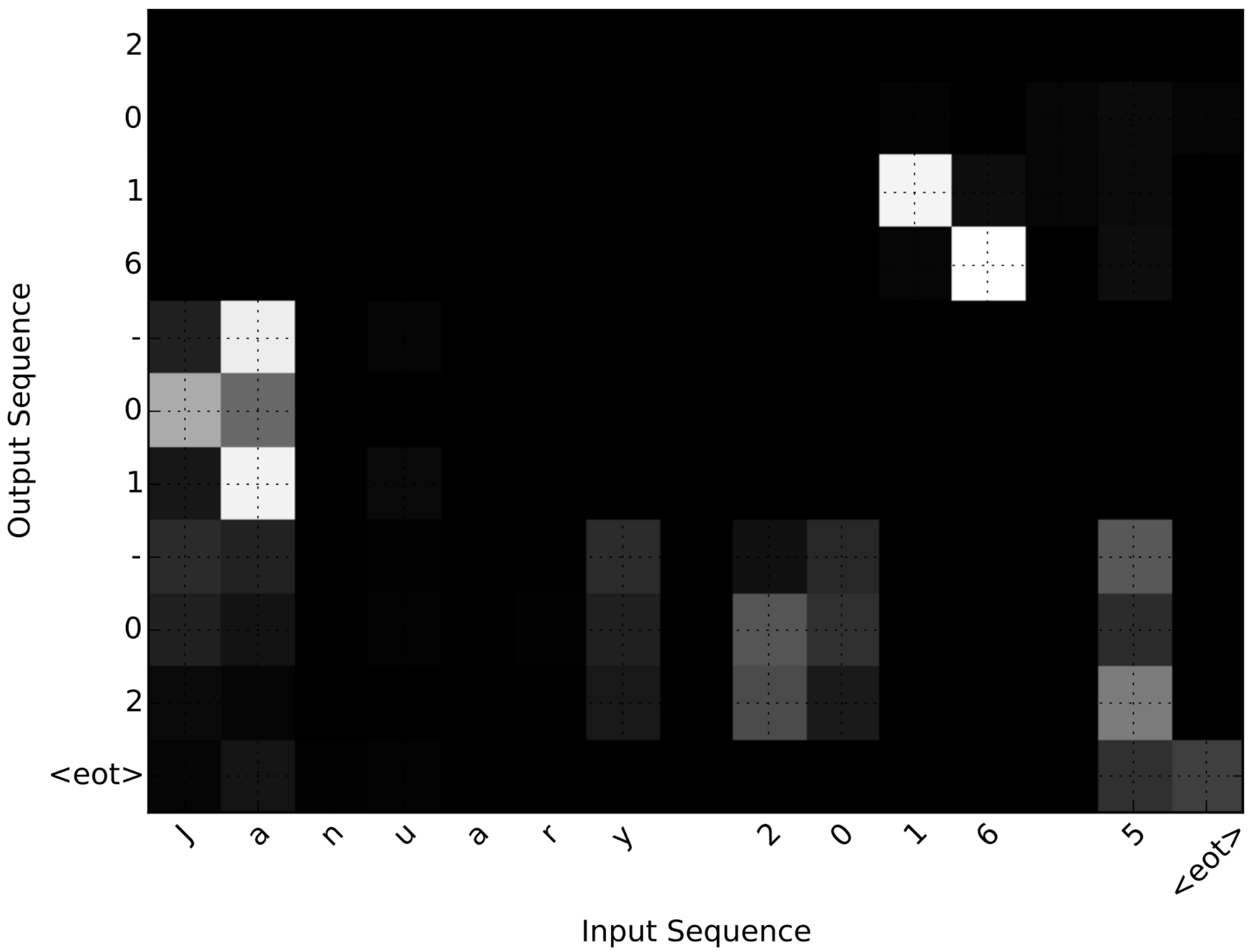
A correct example which doesn't pay attention to unnecessary information like days of the week:



Example 1: The model has learned to ignore "Saturday" during translation. We can observe that "9" is used in the prediction of "-09" (the day). The letter "M" is used to predict "-05" (the month). The last two digits of 2018 are used to predict the year.

And here is an example of an incorrect translation because we submitted our example text in a novel order: “January 2016 05” was translated to “2016–01–02” rather than “2016–01–05”.

We can see that the model has mistakenly interpreted the characters “20” from 2016 as the day of the month. The activations are weak (some even correctly on the actual date of “5”) which can provide us with some insight into how to better train this model.



Example 2: We can see the weirdly formatted date "January 2016 5" is incorrectly translated as 2016–01–02 where the "02" comes from the "20" in 2016

Conclusion

I hope this tutorial has shown you how to solve a machine learning problem from start to finish. Furthermore, I hope it helps you in trying to visualize seq2seq problems with recurrent neural networks. If there is something I missed or you think I could do better, please feel free to reach out via twitter or make an issue on our github, I’d love to chat!

Acknowledgements

Zaf is an intern at Datalogue partially supported by an NSERC Experience Award. The Datalogue Team contributed for code review and reading the proofs of this post.

Thanks to Johanan Ottensooser, Nicolas Joseph, and Sonia Sen.

Machine Learning Recurrent Neural Network Language Translation Deep Learning Keras