CSC321: Assignment 3

Due on March 23, 2018

Wenxin Chen - 1002157676

March 24, 2018

1 Encoder-Decoder Models and Capacity

1.1 Long Sequences

The architecture will likely perform poorly (or at least less accurately compared to shorter sequences) on long sequences. This is because the information at the beginning of the input must travel through many time steps to reach the beginning of the decoder network. The derivatives at each time step must be propagated, easily leading to an exploding or vanishing gradient which are undesirable for obtaining accurate results.

1.2 Pre-Trained Model

Table 1: Output of Pre-Trained Model

Input Word	Output Word	Ground Truth
roomba	ooorlray	oombaray
concert	oncettcay	oncertcay
hello	elleway	ellohay
table	allesay	abletay
are	areway	areway
onion	onoinway	onionway
shopping	optensstay	oppingshay
me	ecay	emay
a	away	away
international	intaraaaiionnway	internationalway
architecture	arttiririineway	architectureway
utilitarianism	usitiiiiiiiintway	utilitarianismway
sophisticated	opistisinisesay	ophistocatedsay

The pre-trained model apperas to predict words starting with vowels more accurately than words starting with consonants. Generally, with shorter words the pre-trained model is not more than one letter off the correct answer, and the incorrect letter is usually the same type as the letter that should be in its place ("type" referring to vowel or consonant). However as the words increase in length the model begins to become less and less accurate (as can be seen in the last four entries). The algorithm apperas to still be able to get the first letter and the suffix correct for these long words, but everything in between could definitely be improved. This reveals the primary weakness of the encoder-decoder model as mentioned previously.

2 Teacher-Forcing

2.1 Problems with Training

During training, teacher-forcing means that the model will only ever see ground truth inputs and will maximize the likelihood of the correct output when this is the case. It gives no consideration to the possibility of inaccurate previous predictions and assumes that no error is propagated forward (because, well, they aren't!). However, during testing it is very likely that an incorrect prediction will occur somewhere in the decoding sequence and the model will not be able to simply exchange this incorrect prediction with the ground truth as it does during training. This could result in disastrous behaviour as the algorithm has not optimized for this sort of incorrect input during training at all, likely making its outputs incoherent as a result.

2.2 Possible Solutions

A possible solution to this issue proposed by the referenced paper is to randomly select between the ground truth and the model prediction as the input to the next unit for each time step. That is, the model output during training is also used in conjunction with the ground truth to decide the input to the next unit. The paper proposes a sort of "coin-flip" to decide which of the two is used at each time step. Furthermore the paper also suggests using a "curriculum learning strategy", in which the probability of these coin flips favours the ground truth at the beginning of training (which makes sense, as the model is likely to be still extremely inaccurate at this stage) and the model outputs nearing the end of training (when the model begins to make accurate predictions). Using this method, the model will have a chance to adapt to its own incorrect outputs and minimize their propagation along the decoder sequence.

3 Gated Recurrent Unit

The gated recurrent unit (GRU) was implemented using a combination of pytorch's nn.Linear modules as well as nn.Sigmoid and nn.Tanh modules.

```
class MyGRUCell(nn.Module):
    def __init__(self, input_size, hidden_size):
        super(MyGRUCell, self).__init__()

        self.input_size = input_size
        self.hidden_size = hidden_size

        self.w_ir = nn.Linear(input_size, hidden_size)
        self.w_hr = nn.Linear(hidden_size, hidden_size)
        self.w_iz = nn.Linear(input_size, hidden_size)
        self.w_hz = nn.Linear(hidden_size, hidden_size)
        self.w_in = nn.Linear(input_size, hidden_size)
        self.w_hn = nn.Linear(hidden_size, hidden_size)

        self.sigmoid = nn.Sigmoid()
        self.tanh = nn.Tanh()

    def forward(self, x, h_prev):

        r = self.sigmoid(self.w_ir(x) + self.w_hr(h_prev))
        z = self.sigmoid(self.w_iz(x) + self.w_hz(h_prev))
        g = self.tanh(self.w_in(x) + r * (self.w_hn(h_prev)))
        h_new = (1-z) * g + z * h_prev

        return h_new

```

Listing 1 : Implementation of MyGRUCell

4 Implementing Attention

4.1 Attention Weights

The network used to learn the attention function f was implemented using nn.Sequential as shown in Listing 2.

```
    self.attention_network = nn.Sequential(
            nn.Linear(hidden_size*2, hidden_size),
            nn.ReLU(),
            nn.Linear(hidden_size, 1)
        )

```

Listing 2 : Attention Network

The forward pass feeds the decoder hidden state and the entirety of the encoder hidden states (that is, the encoder state for every encoding time step) into the attention network and returns the softmax of the result.

```
  concat = torch.cat((expanded_hidden, annotations), 2)
        reshaped_for_attention_net = concat.view(-1, self.hidden_size*2)
        attention_net_output = self.attention_network(reshaped_for_attention_net)
        unnormalized_attention = attention_net_output.view(batch_size, seq_len, 1)

        return self.softmax(unnormalized_attention)

```

Listing 3 : Forward step for attention network

4.2 Decoder Network

The forward pass of the attention-based decoder network uses the attention weights and the encoder hidden states (annotations) to create a context vector, which contains the most useful information from the encoding states for that particular decoding unit. This is combined with the letter input to that decoder (the output of the previous unit or <SOS>) and sent to the GRU Cell to obtain the new hidden state. The actual output is then calculated from the new hidden state through a simple linear transformation. The forward code is shown below:

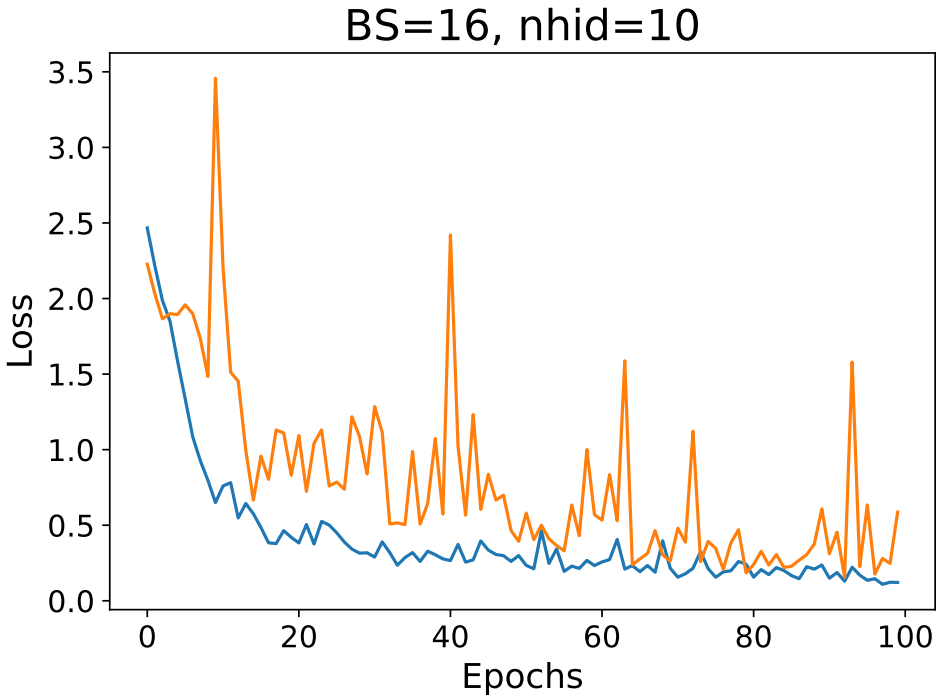
```
        embed = self.embedding(x)     # batch_size x 1 x hidden_size
        embed = embed.squeeze(1)      # batch_size x hidden_size

        attention_weights = self.attention(h_prev, annotations)
        context = torch.sum(attention_weights * annotations, 1)
        embed_and_context = torch.cat((embed, context), 1)
        h_new = self.rnn(embed_and_context, h_prev)
        output = self.out(h_new)

        return output, h_new, attention_weights

```

Listing 4 : Attention-based decoder forward pass

4.3 Training

The validation (orange) and training (blue) loss over the training epochs are shown below. Note Batch Size = 16 and hidden_size = 10.

Figure 1: Training (blue) and Validation (orange) loss over 100 epochs



5 Attention Visualizations

Some visualizations of the attention weights can be seen below, followed by a brief explanation of findings.

5.1 Visualization Figures

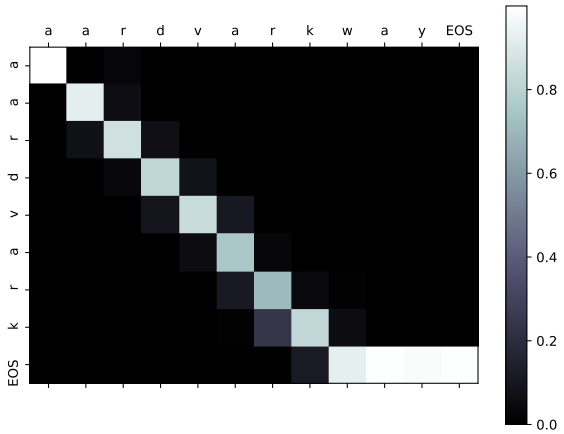


Figure 2: Attention Weights For "Aardvark"

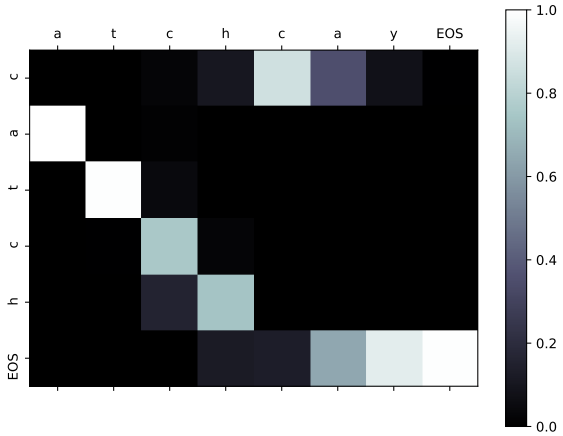


Figure 3: Attention Weights For "Catch"

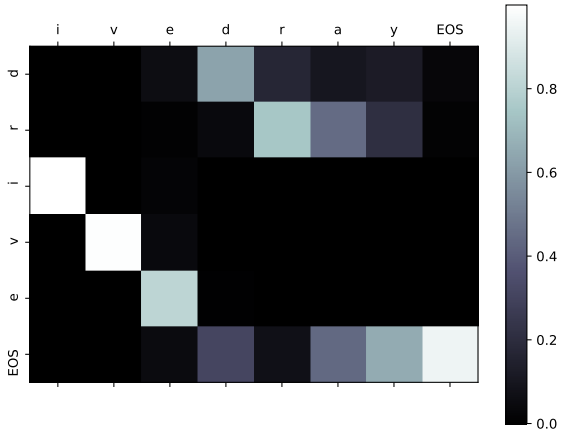


Figure 4: Attention Weights For "Drive"

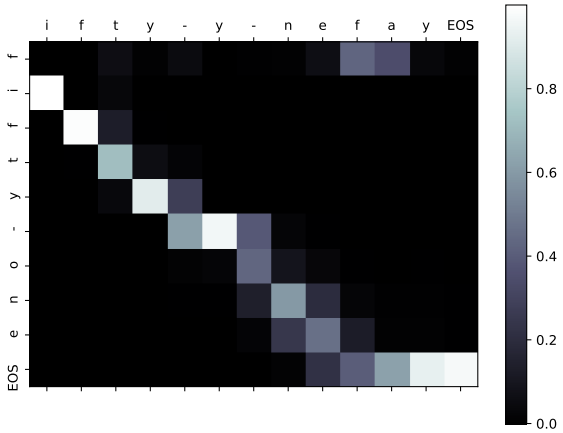


Figure 5: Attention Weights For "Fifty-One"

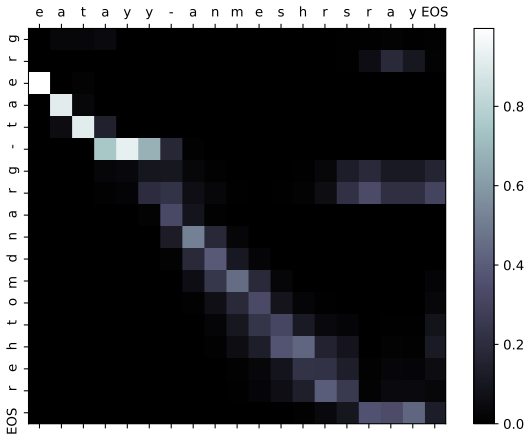


Figure 6: Attention Weights For "Great-Grandmother"

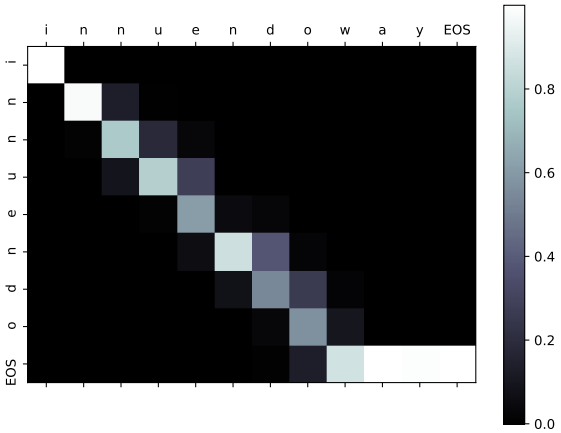


Figure 7: Attention Weights For "Innuendo"

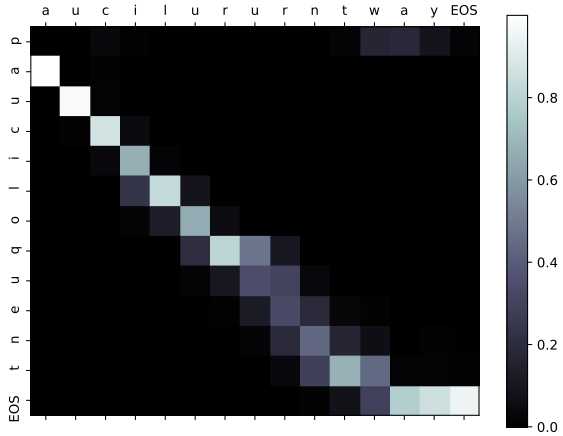


Figure 8: Attention Weights For "Pauciloquent"

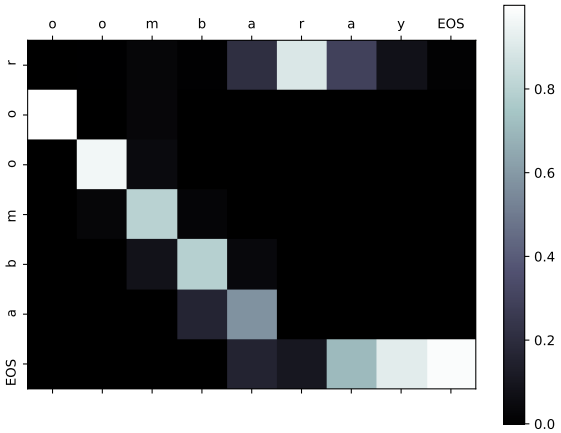


Figure 9: Attention Weights For "Roomba"

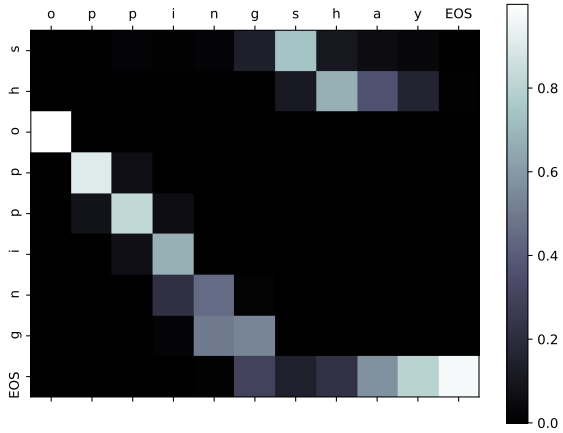


Figure 10: Attention Weights For "Shopping"

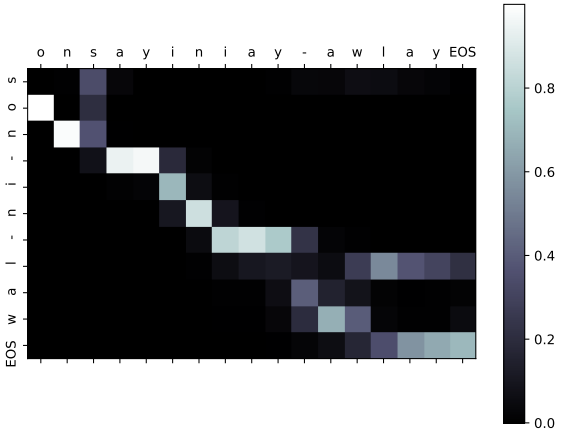


Figure 11: Attention Weights For "Son-In-Law"

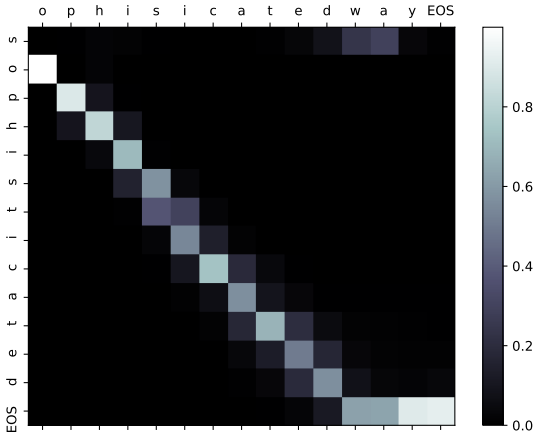


Figure 12: Attention Weights For "Sophisticated"

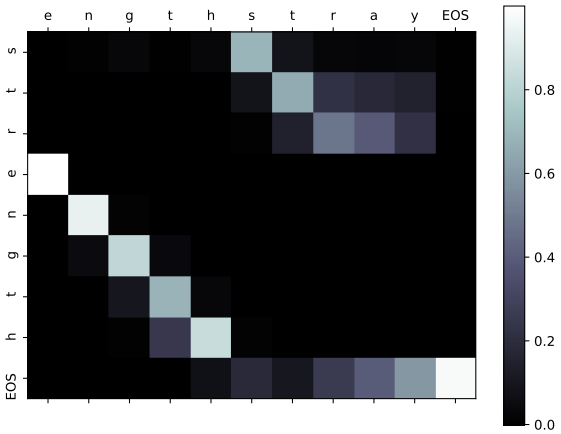


Figure 13: Attention Weights For "Strength"

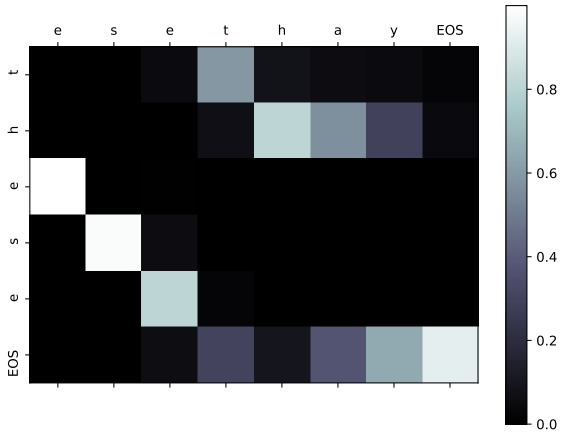


Figure 14: Attention Weights For "These"

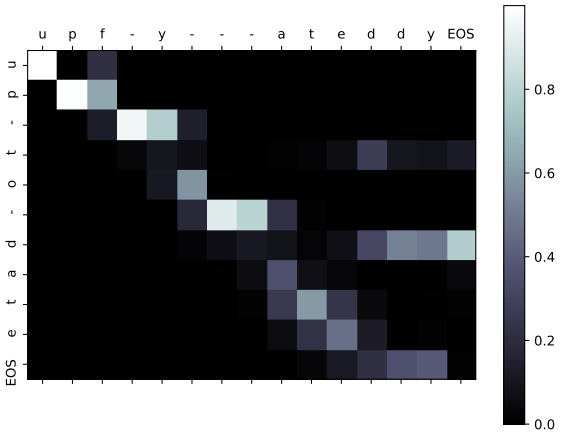


Figure 15: Attention Weights For "Up-To-Date"

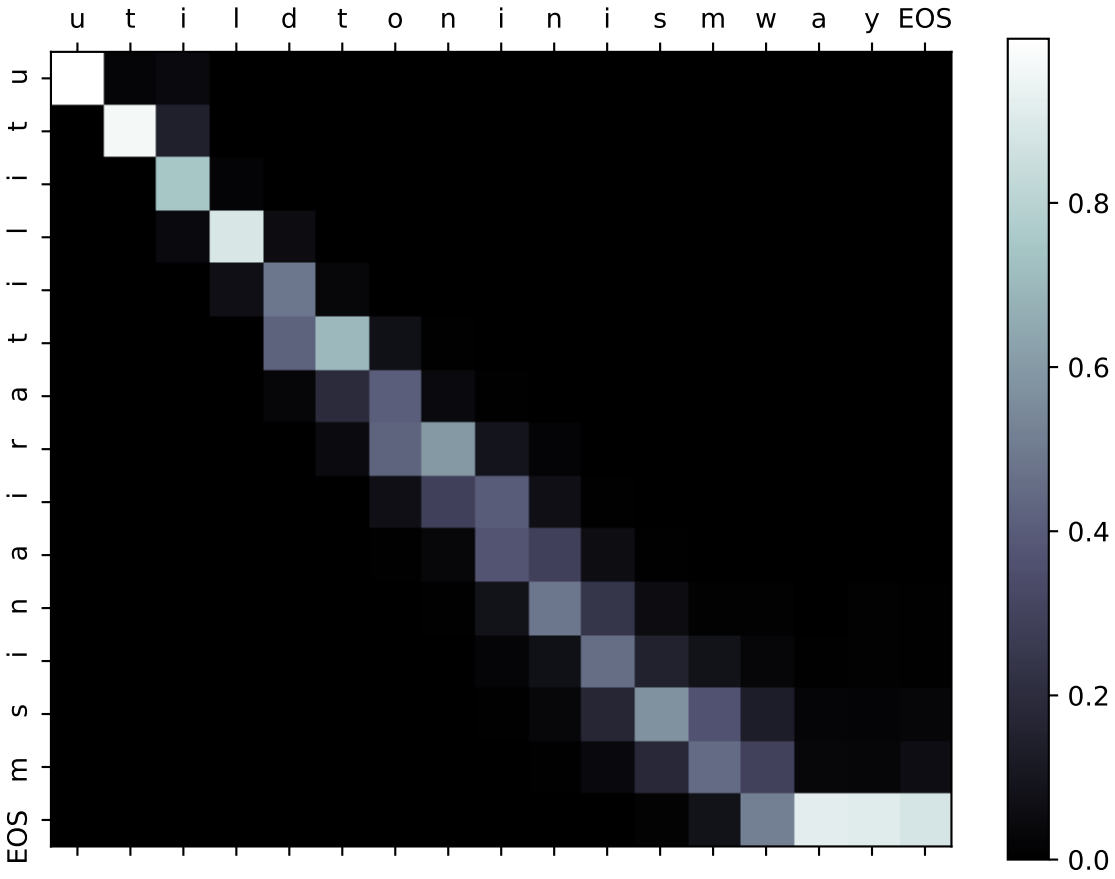


Figure 16: Attention Weights For "Utilitarianism"

5.2 Explanation of Findings

The algorithm appears to perform well on short to medium length words regardless of the nature of the word. For example, the model correctly translates "Aardvark", "Catch", "Drive", "Innuendo", "Roomba", "Shopping", and "These", among others. The attention networks follows a predictable pattern in these words, always starting at the first vowel in the word and moving to the next encoded letter one by one until the end is reached, at which point it starts again from the beginning (unless the word begins in a vowel). The attention network consistently assigns a large weight to <EOS> when all letters have been exhausted. In words beginning with vowels, the attention network appears to practically assign the entirety of the weights to <EOS>, leading to "way" being output as the end of the word (see Fig 2, 7, 16). In words beginning with a consonant the attention network assigns a portion of the weights to the beginning letters as well as <EOS>, signaling the decoder network to use a different ending other than "way" (see Fig 3, 4, 9, and others).

With short to medium length words the model is quite consistent, even translating "Strength" correctly, a word that begins with three consonants.

With longer words, the algorithm does worse. This appears to be the result of the attention algorithm being less and less "sure" of itself the further down the decoder network it goes. For example, in the words "Pauciloquent", "Sophisticated", and "Utilitarianism", the algorithm is a few letters off the correct translation. The attention network can be seen getting "fuzzier" down the decoder network and assigns probabilities to a wider range of encoding units (when there really should be only one). In shorter words such as "Roomba" the attentions can be seen to be less spread out in general. Perhaps adding more longer words to the training set would help improve the model's performance on longer words.

Finally, the model performs even worse on words containing dashes. This may actually be a weakness within the decoding network rather than the attention network. Observing the attention weights of the words containing dashes, it actually does a reasonable job of assigning the weights to the relevant units. For example, in "Fifty-One" it clearly assigns a high weight to the dash for three consecutive decoder units, which makes sense- the dash signals the end of a word and so the pig latin ending should be output by the decoder. This behaviour is seen in most of the words containing dashes to some degree, although words containing longer dashes and longer words end up confusing the attention network somewhat (see Fig 6, 11, 15). Perhaps with more dashed words in the training set, the attention network would be able to handle these cases better. However, even in the cases that it did handle them quite well ("Fifty-One") the decoder network still failed to output a result that was even remotely close, so there may be an issue with the decoder's ability to handle dashes as well.