**Neural Networks – Attention**

Continuing on the theme of translating a sequence of words from one language to another…let’s revisit that encoder-decoder network. If we’re translating a really long sentence, then the rolled out encoder could comprise 10 or more words. The example he gives is: ‘*don’t eat that delicious smelling and tasting pizza*’. The first word is crucial to the proper understanding/translating of that sentence. However, even with LSTM’s the information carried by that first word, and ultimately stored in the context vector, can get lost amongst the contributions from all the other words, especially if there are many intervening words.

A diagram of a machine

Description automatically generated

So instead of only feeding the final context vector of the *entire* encoder into the decoder, we’ll additionally feed (a linear combination of) the individual encoder instance context vectors into the decoder. This is called Attention. And it helps mitigate the vanishing/exploding gradient problem. In reference to diagram below….

A diagram of a network

Description automatically generated

we start with the first instance of the encoder, and extract *its* context vector (i.e., its LSTM output). Actually, apparently we just extract the cell states/long term memory. So I’ll call this the cell state context vector, **v**1(e). In our illustration, this is being done by the blue bar, and the vectors are 4 dimensional vectors. And then we go to the next instance of the encoder and extract its cell state context vector **v**2(e). And we repeat until we reach the end of the encoder. Finally, we start unrolling the decoder. So after the first instance, we extract its cell state context vector **v**1(d). All of these (pink) cell state context vectors get fed into the Att function. Its job is to output a normalized linear combination of the encoded cell state context vectors. Specifically:



This is called the ***attention*** value. This vector gets fed into the NNN (pink line still), along with the output from the instance of the decoder. And these inputs are used to determine the first word of the decoder. Then as usual, this output is used as the input for the next instance of the decoder. And the process repeats. All the same encoder cell state context vectors, **v**j(e) (purple now) and the 2nd instance decoder cell state context vector, **v**2(d) (also purple) get fed into another Att, which outputs,



And this vector, along with the output of the 2nd decoder instance get fed into the NNN to output the second word. And the process repeats until the decoder outputs the EOS token.

**Training**

When training an Attention model, it is common practice to use *Teacher Forcing*. That is to say, instead of letting the decoder predict the whole sequence output, we only allow it to predict one word at a time. So regardless of what word it predicts with the first instance of its RNN, we use the *correct* output for the second instance. And along those same lines, we don’t let the output ramble on for as long as it wants. We stop the output when we should get to the ‘end of sequence’ (EOS) word/token. So for instance, if you input ‘I am a student’ into the encoder, and we expect a decoder output of ‘Je suis etudiant’, then we’d input ‘I am a student’ into the encoder, and calculate the loss from just the first output word, which should be ‘Je’. Then we’d input ‘I am a student’ into the encoder, and I think we’d input ‘Je’ into first word of the decoder (this would be the first input into the decoder after the EOS input), and grade it on the output of ‘suis’. And then we’d input ‘I am a student’ into the encoder, and ‘Je suis’ into the decoder, and grade it on the output of ‘etudiant’. Anyway, our loss function would be the usual cross entropy I guess, for all n output words, xj,



where pi(xj) is the probbility of distribution of the expected output of the jth word, and fi(xj) is the model output probability of the output of the jth word. Note pi(xj) would presumably be (0, 1, 0, 0) in the first decoder output, while fi­(xj) would be (0.07, 0.9, 0.01, 0.02).