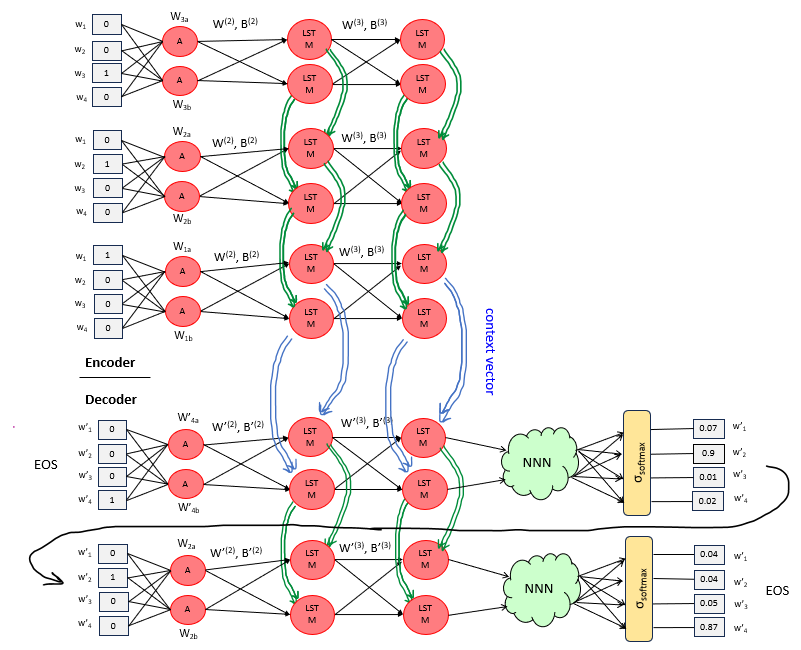
**Neural Networks – Encoder/Decoder**

Now want to discuss sequence to sequence modeling. The idea here is that we have a sequence as input (say words in a sentence in one language), and we want to output another sequence (say words in a sentence in another language). An example model to accomplish this task is shown below. So first we have a word embedder serving as input into an ‘encoder’ RNN. The word embedder takes a word, or punctuation (like period, question mark, etc.), which I’m going to call a word as well, and converts it to a number via the word-embedding algorithm discussed in previous file. So for instance, (w3a, w3b) would be the input into the top two activation functions respectively, as indicated. Note there are still to-be-trained weights and biases along the lines going from the A’s to the LSTM. These weights and biases would be the same for each layer of the network.



This word vector will be passed as input into the RNN. These encoder-decoder networks are typically composed of RNN’s with multiple hidden layers, and hidden layers multiple nodes deep. And with nodes consisting of LSTM’s modules. And this RNN is unrolled as many times as we have words in the sequence. When we’re done, the encoder LSTM outputs (called the **context vector** with its own weights/biases) are input into the decoder network. The decoder network is also an RNN with its own distinct (4 in this case) LSTM modules/nodes. The input into the first instance of the RNN is an ‘end of sequence/beginning of new sequence’ word, given by (0001) in this illustration. The RNN instance uses this and the context vector to output information into a normal neural network (NNN) to output to a softmax layer to conjure up the first word in the output sequence. Or at least the most probable word. It then feeds this most probable word as input into the next instance of the decoder RNN. And this instance would output another word, and feed it into the next instance of the decoder RNN, etc. The process would stop when a decoder RNN instance outputs an ‘end of sequence’ token, denoted again as 0001 in this illustration.

Should point out that the length of the vocabulary (4) of the encoder words and the length of the vocabulary (4) of the decoder words was the same in this illustration. They need not be. Also, both vocabulary’s words have dimensionality d = 2. But they don’t have to be the same.

**Training**

When training a seq2seq 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).