<https://jalammar.github.io/illustrated-transformer/>

<http://nlp.seas.harvard.edu/2018/04/03/attention.html>

<http://nlp.seas.harvard.edu/2018/04/03/attention.html>

Implementation

<https://github.com/omarsar/transformers>

**Helpful Concepts**

**Bag of Words**

* A model used to create a vector output of a sentence
* It works by building a vocabulary (the data is pre-processed to remove stop words, it can also be lemmatized or stemmed – used to get the root of the word
* It creates vectors by counting the number of times a word appears in the sentence and assigning it the corresponding number in the vector
* **Problem:** 
  + Vocabulary size can get too large causing the dimensionality of the vector output to increase significantly
  + Prone to overfitting

**Covariate Shift**

* If you learned some x -> y mapping and the distribution of x changes – your function will not perform well and may have to retrain your algorithm
* Example, if you trained a network to predict cat or no cat but only used black cats
* Model will not generalize well to pictures of colored cats and will likely misclassify them
* **Application**
  + In neural nets or even shallow networks – hidden layer values are constantly shifting
  + For the 3rd hidden layer – the values of the second hidden layer for example are constantly shifting (likewise with the second hidden layer and the first etc.)
  + This means that some of the hidden layers will likely suffer from covariate shift
  + In order to limit the amount by which updated values in earlier layers will affect values deeper in the network will see, **can use batch norm**
  + Ensures that even if the actual values change, the mean and variance of the distribution stay the same (can be mean 0 and variance 1 or smt else you force it to be)
    - Allows more stable ground for later networks to stand on
  + **Batch norm** also has a kind of regularization effect (similar to dropout)
  + **Ultimately, batch norm helps speed up learning and normalize later hidden layers in the network**

**The Transformer**

* Takes in an input of a sentence and outputs a sentence
* Composed of an encoder and a decoder
  + The encoders and decoders are arranged in stacks
  + Encoding component is stacked with 6 encoders
  + Decoding component is stacked with the same number
* Encoder
  + The encoder is made up of a self-attention layer and a feed forward neural networkA screenshot of a cell phone

    Description automatically generated
  + The self-attention layer is used to look at other words in the input sentence
* As with most NLP algorithms, start by converting each word into a vector using an embedding algorithm
* The first encoding layer receives the word embeddings as input (other encoders receive the output of the previous encoding layer)
  + \*Note – size of the list/word embedding is a hyperparameter that we tune (usually = longest sentence in the training data)
* **Key Property** 
  + The word embeddings flow in parallel
  + Meaning each word in each position of the sentence takes a different path in the encoder
* Each sub-layer also has an add & normalize

A screenshot of a cell phone

Description automatically generated

* **Self-Attention**
  + Method used by the transformer to learn high-level dependencies between words in a sentence
    - E.g. “The animal didn’t cross the street because it was tired.”
    - Would be very hard for a model to learn what “it” was referring to
    - Self-attention allows the transformer to understand how relevant words are associated with each other – that “it’ refers to animal
  + How does it do this?
    - It starts by creating 3 vectors from each word embedding: a Query vector, Key vector, and Value vector
      * These vectors are generated my multiplying them with pre-assigned matrices whose weights are learned in the training stage
      * \*NOTE these 3 vectors can be smaller or the same size as the original vectors
      * At a high level, these 3 vectors are useful for learning these dependencies
    - Step 2: calculate a score for each of the other input words given a word
      * For example, in the word “Think Different” The word Think would have a score for “Think” and a score for “Different”
      * These scores help the model learn to focus on which parts of the sentence are the most relevant
      * This is done by multiplying the query vector of the word with the respective key vector of the other words in the input sentence
        + E.g. so q1 (Think) \* k1 (Think) and q1 \* k2 (Different)
    - Step 3: divide these scores by the square root of the dimension (so /8) and this step is helpful for creating more stable gradients
    - Step 4: softmax the scores so that they are all positive and add up to 1
    - Step 5: multiply each of the value vectors (of each of the corresponding words) by the their respective softmax(score)
      * This allows the model to focus on more relevant/important parts and down out irrelevant parts of the sentence
    - Step 6: sum all of the scores vectors
    - This becomes the output for the self-attention layer at this position (i.e. for the first word, second etc.)
* **Multi-Head Attention**
  + Instead of performing a single attention function (across all of the inputs), the multi-head attention performed attention several times
  + And concatenated the output to learn “multiple representation subspaces”
  + This also allowed model to expand attention to focus on other parts of the sentence for each word – stops these vectors being dominated by the same word e.g. “Think” in “Think Different” associating most strongly with “Think”
  + The final vector (which is fed into the neural network) is computed by concatenating all of these Z matrices and multiplying them by a weight matrix to produce the output layer
* **Positional Encoding** 
  + To code information about the order of a sentence and the relative distances/positioning between words:
  + Positional encodings are generated which are added to the word embeddings before being fed into the self-attention layer
  + This is done because there is no recurrence or convolution – so some meaning must be injected into the model about the order of a sentence
* **The Decoder** 
  + Has same structure but sandwiched in between the self-attention layer and the feed forward neural network is an encoder-decoder attention layer
  + Encoder starts by processing the input and producing a set of attention vectors K and V and these are used by the “encoding-decoding attention” layer
    - This layer helps output to focus on the relevant/appropriate parts of the input sentence
  + At each timestep the previous outputs are fed back into the decoder layer before outputting the next word in the sentence
    - Just like was done with the encoding layer, embed these outputs and add positional encodings
  + This is done until a special symbol indicating the end of the sentence has been reached
  + **Self-attention layer**
    - Operates a bit different to the self-attention layer in the encoding layer
    - Only allowed to attend to earlier positions in the output – masks later words (by setting them to -infinity before the softmax step – makes them 0)
  + **Encoder-decoder attention layer**
    - Works in the same way as the multi-headed attention but creates the queries matrix from the previous decoding layer
    - It gets the keys and values from the output of the encoding layers
* **The Final Layer**
  + Consists of a linear and softmax layer
  + The linear layer is a simple feedforward neural network that converts the output layer into a giant logits vector
    - Logits vector is a vector with the vocabulary size
  + This vector is then softmaxed and the word corresponding to the highest probability cell is selected
* **Loss Function**
  + Obtain the error by subtracting our model untrained output with the label (one-hot encoded vector with the correct word at each time step)
  + More on this:
    - <https://colah.github.io/posts/2015-09-Visual-Information/>
    - <https://www.countbayesie.com/blog/2017/5/9/kullback-leibler-divergence-explained>
  + **Greedy Decoding**
    - Select the word with the highest probability and discard the rest
  + **Beam Search**
    - Hold onto the top n words and in the next output position
    - Then run the output with each of the n words
    - Whichever word produced the least error would be selected
  + Both of these are hyperparameters we can control