**Chapter 5: Autoregressive Models**

1. A **family** of models that simplify the generative modeling problem by treating it as a sequential process. They condition prediction on previous values in the sequence, rather than on a latent random variable. Therefore, they try to explicitly model the data generating distribution rather than an approximation of it (like in VAE). Here we will see two autoregressive models: LSTM and PixelCNN.
2. Working with text data: There are several key difference between text and image data so many that works well in images might not work well for text. In particular:
   1. Text data is composed of discrete chunks (either characters or words), whereas pixels in an image are points in a continuous color spectrum. We can easily make a green pixel bluer, but it is not obvious how we should go about making the word *cat* more like the word *dog*. So we can easily apply BP, to calculate the gradient of loss function w.r.t individual pixels to establish the direction in which pixel colors should be changed to minimize the loss, we can’t obviously apply BP in the same way, so we need a work around.
   2. Text data has time dimension but no spatial dimension, whereas images as two spatial dimension but lacks the time dimension. The order of words is highly important in text data and words wouldn’t make sense in reverse, whereas images can usually be flipped without affecting the content. Furthermore, there are long-term dependencies between words that need to be captured by the model: for example, the answer to a question or carry forward the content of a pronoun. With image data, all pixels can be processed simultaneously.
   3. Text data is highly sensitive to small changes in the individual units (words or characters). Image data is generally less sensitive to changes in individual pixel units – a picture of a house would still be recognizable as a house even is some pixels were altered – but with text data, changing even a few words can drastically alter the meaning of the passage, or make it nonsensical. This makes it very difficult to train a model to generate coherent text, as every word is vital to the overall meaning of the passage.
   4. Text data has rules-based grammatical structure, whereas image data doesn’t follow set rules about how the pixels values should be assigned.
3. **Tokenization:** The first step in text processing is cleaning up and tokenizing the text. It’s a process of splitting the text up into individual units, such as words or characters. How you tokenize, depends on the task that you are trying to do. There are both pros and cons of words and character tokenization, and your choice will affect how you need to clean the text prior to modelling and the output from your model.

**If you use word tokens:**

1. All text can be converted to lowercase, to ensure capitalized words at the start of sentences are tokenized the same way as the same words appearing in the middle of sentence. However, it might not always be desirable; for example some proper nouns, such as names or places, may benefit from remaining capitalized so that they are tokenized independently.
2. The text vocab (set of distinct words in the training set) may be very large, with some appearing very sparsely or perhaps only once. We may replace sparse words with *unknown token*, rather than including them as separate tokens, to reduce the number of weights the neural network to learn. For character tokenization, the vocab will be small.
3. Words can be stemmed, meaning that they are reduced to their base form. Going, went becomes go.
4. You need to either tokenize the punctuation marks or remove them altogether,
5. Using word tokenization means the model will never be able to predict words outside the training vocab.

**Note:** The sort of tokenization and cleaning up depends on the task to perform. **Stemming** might lead to loss of context and meaning: "going," "went," and "go" have different contextual meanings. If they are all reduced to "go," the model loses the ability to distinguish between past, present, and future tense. **Punctuation,** if removed the model won’t know when to put a full stop or a comma in a sentence. So, as far as **LLMs and NLP models focused on generation are concerned,** stemming or punctuation might hurt the performance, whereas for text classification, we might get away with it without hurting the performance.

Code:

def pad\_sentences(sentence):  
 sentence = re.sub(f"([{string.punctuation}])", r" \1 ", sentence)  
 sentence = re.sub(" +", " ", sentence)  
 return sentence.strip()

# Note: re.sub, searches for the pattern in sentence (i.e. first parameter), and replace with the second parameter.

1. The replacement r" \1 " adds spaces **before and after** each punctuation mark.
2. **re.sub(" +", " ", s)**

* This replaces **multiple spaces** with a **single space**, ensuring a clean format.
* Example:  
  **Before:** "Hello , how are you ? " (extra spaces)  
  **After:** "Hello , how are you ?" (cleaned up)

We do this because we also want to add punctuation mark in our vocabulary.

dataset = tf.data.Dataset.from\_tensor\_slices(filtered\_data)  
dataset = dataset.batch(BATCH\_SIZE).shuffle(1000).prefetch(tf.data.AUTOTUNE)  
  
# Step 3: Create a Vectorization layer  
# A preprocessing layer which maps text features to integer sequences.  
vectorize\_layer = tf.keras.layers.TextVectorization(  
 standardize='lower',  
 output\_mode='int',  
 max\_tokens=VOCAB\_SIZE,  
 output\_sequence\_length=MAX\_LEN + 1  
)  
vectorize\_layer.adapt(dataset) # Sort of fits to the dataset  
vocabulary = vectorize\_layer.get\_vocabulary() # Get the vocab

Full docs: [hier](https://www.tensorflow.org/api_docs/python/tf/keras/layers/TextVectorization). Some of the parameters are explained below:

1. A preprocessing layer which maps text features to integer sequences. It transforms a batch of strings (one example = one string) into either a list of token indices (one example = 1D tensor of integer token indices) or a dense representation (one example = 1D tensor of float values representing data about the example's tokens). This layer is meant to handle natural language inputs.
2. The processing of each example contains the following steps:
   1. Standardize each example (usually lowercasing + punctuation stripping)
   2. Split each example into substrings (usually words)
   3. Recombine substrings into tokens (usually ngrams)
   4. Index tokens (associate a unique int value with each token)
   5. Transform each example using this index, either into a vector of ints or a dense float vector.
3. Parameters:
   1. max\_tokens = Maximum size of the vocabulary for this layer. This should only be specified when adapting a vocabulary or when setting pad\_to\_max\_tokens=True. Note that this vocabulary contains 1 OOV token, so the effective number of tokens is (max\_tokens - 1 - (1 if output\_mode == "int" else 0)).
   2. Standardize: To apply to input text: **None, lower\_and\_strip\_punctuation**, **lower** and **strip\_punctuation**, or a **callable** function, where input will be passed and should be standardize and returned. Rest is self-explanatory.
   3. **output\_mode**:
      1. "int": Outputs integer indices, one integer index per split string token. When output\_mode == "int", 0 is reserved for masked locations; this reduces the vocab size to max\_tokens - 2 instead of max\_tokens - 1.
      2. "multi\_hot": Outputs a single int array per batch, of either vocab\_size or max\_tokens size, containing 1s in all elements where the token mapped to that index exists at least once in the batch item.
      3. "count": Like "multi\_hot", but the int array contains a count of the number of times the token at that index appeared in the batch item.
   4. output\_sequence\_length: Only valid in INT mode. If set, the output will have its time dimension padded or truncated to exactly output\_sequence\_length values, resulting in a tensor shape (batch\_size,output\_sequence\_length)regardless of how many tokens resulted from the splitting step.
4. Embedding Layer: It’s essentially a lookup table that converts each integer token into a vector of length *embedding\_size*. The lookup vectors are learned by the model as weights. Therefore, the number of weights learned by this layer is equal to *# vocab\_size \* embedding\_size*. We embed each integer token into a continuous vector because it enables the model to learn a representation for each word that is able to be updated through BP. We could also do one-hot encoding for each input token, but using an embedding layer is preferred because it makes the embedding itself trainable, this giving model more flexibility in deciding how to embed each token to improve its performance.

Therefore, the Input Layer passes a tensor of integer sequences of shape [batch\_size, seq\_length] into Embedding layer, which outputs a tensor of size [batch\_size, seq\_length, embedding\_size], which is then passed to the model.