Some tools for

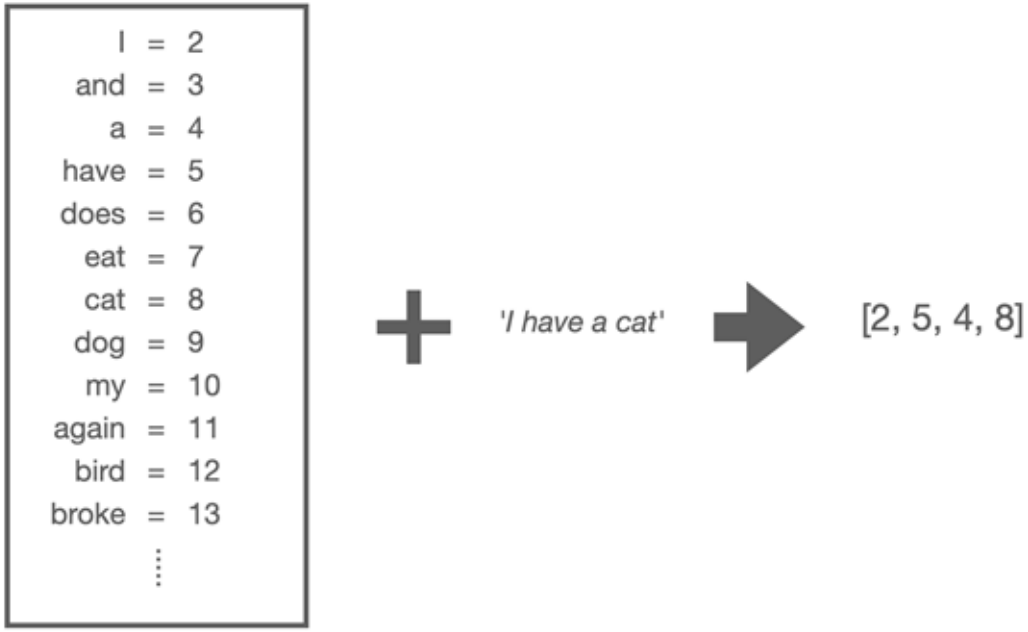
Natural Language Processing

Recitation 5

1. Tokenization

Tokenization associates each word with an integer.

- The size of your tokenizer dictionary is your vocabulary size. Your vocabulary size could be based on your dataset, or an artificial threshold that you set.
- At training and test time, your model will ignore any word that isn't in your vocabulary
- We store the most frequent words at lowest integers. In many language models, you may even cut the tail of the distribution
- The tokenizer dictionary also needs to include PAD and SOS, placeholder tokens that indicate padding or the start of a sentence.



1. Tokenization

Count the words

```
# Create a dictionary of words
self.word_count = {}
for review in self.df["review"]:
    for word in review.split():
        if word not in self.word_count:
            self.word_count[word] = 1
        else:
            self.word_count[word] += 1
```

Sort the words by frequency

```
print(f"There are {len(self.word_count)} distict words")
# insert the padding token PAD to index 0
self.word_count = dict([('PAD', 1), ('SOS', 1)] + sorted(self.word_count.items(), key=lambda x: x[1], reverse=True)[:vocabulary_size-2])
```

Word-to-int dictionary

```
# Create dictionary to convert words to integers
self.word_dict = {word: i for i, word in enumerate(self.word_count)}
print(f"Vocabulary size: {len(self.word_dict)}")
```

Tokenize the dataset

```
for i in range(self.length):
    review_words = self.df["review"][i].split()
    indices = [1] # start with the 'SOS' token in each review
    for word in review_words:
        if word in self.word_dict:
            indices.append(self.word_dict[word])

    label = self.df["sentiment"][i]
    self.texts.append(torch.tensor(indices[:max_len], dtype=int))
    self.labels.append(label)
```

2. Truncation & Padding

A language model could analyze anything from a 5-word sentence to a full novel. We need to:

- Set a limit to how a long a sentence can be (truncation)
- Bring all the shorter sentences to this length to standardize the inputs (padding)

Truncation

SOS	I	have	a	cat	and	it	keeps	chasing	after	a	bird
1	2	5	4	8	3	14	58	322	17	4	12

Padding

SOS	I	have	a	cat	PAD	PAD	PAD	PAD	PAD		
1	2	5	4	8	0	0	0	0	0		

2. Truncation & Padding

```
def imdb_collate(batch):
    texts, labels = zip(*batch)
    text_pad = pad_sequence(texts, batch_first=True)
    lengths_text = torch.tensor([len(seq) for seq in texts])
    labels = torch.tensor(labels, dtype=torch.float32)
    return text_pad, lengths_text, labels
```

highly complex piece of code

Truncation and padding happen after tokenization and before embedding

3. Embedding

Embedding converts a token into a vectorized representation that can be looked on at any time.

- A form of linear layer that will be optimized along with the rest of NN during training.
- Weights start at random. Along training, embeddings will shift to optimize model performance and learn to capture the meaning/context of a word

- Embeddings are a way to bring “language” into a latent dimension.
- The higher the embedding length, the more subtle the meaning

3. Embedding

RNN starter code

```
train_dataloader, test_dataloader, vocabulary = get_dataloader(vocabulary_size, max_review_length, batch_size)
word_embeddings = nn.Embedding(vocabulary_size, input_size)
```

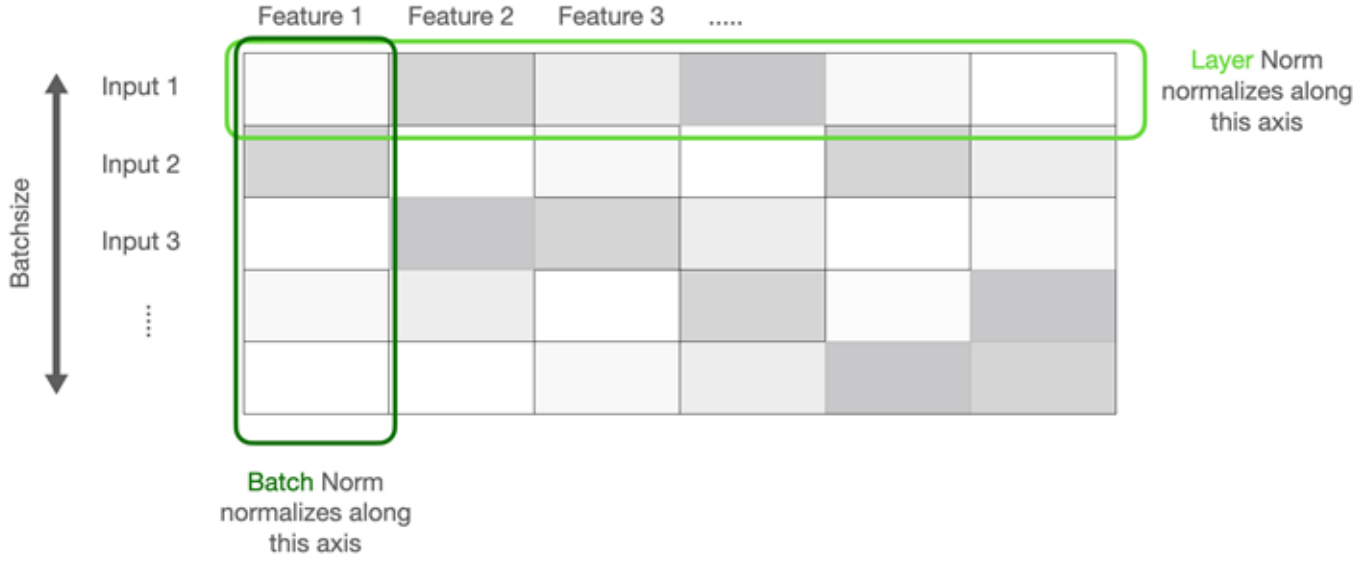
Transformer starter code

```
class TransformerEmbedding(nn.Module):
    def __init__(self, vocab_size, d_model, max_len=500, drop_prob=0.1):
        super(TransformerEmbedding, self).__init__()
        self.tok_emb = nn.Embedding(vocab_size, d_model, padding_idx=0)
        self.pos_emb = PositionalEncoding(d_model, max_len)
        self.drop_out = nn.Dropout(p=drop_prob)
```

4. Layer Norm

Layer normalization is a traditional alternative to batch norm in RNN and Transformers.

- We normalize each input *independently* across all of its features



4. Layer Norm

An extremely famous image that is always useful to just keep in mind