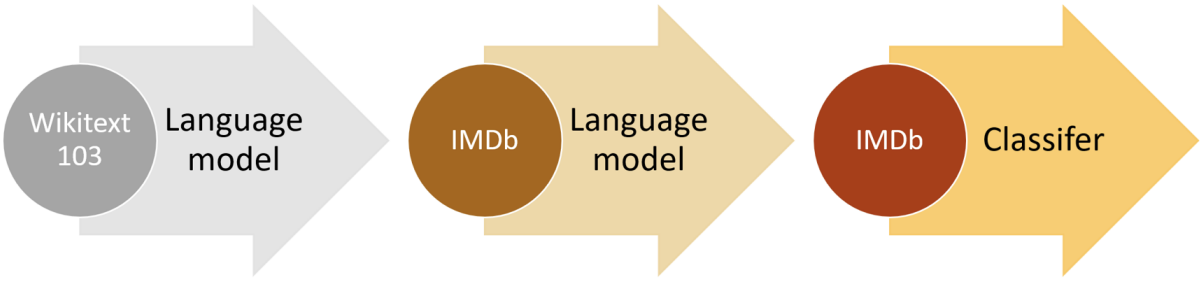
Lesson 8

Chapter 10

The pre-trained model we previously used to classify sentiment on movie reviews was a *language model*. A language model is is a model that has been trained to guess what the next word in a text is (having read the ones before). This kind of task is called self-supervised learning: we do not need to give labels to our model, just feed it lots and lots of texts. It has a process to automatically get labels from the data, and this task isn't trivial: to properly guess the next word in a sentence, the model will have to develop an understanding of the English (or other) language. The language model was trained on Wikipedia.

> Just like an imagenet model has information about what pictures look like, a language model knows about what sentences look like, how language works, and what information they contain about the world.

The ULMFit paper showed that fine-tuning the language model on extra data prior to classification through transfer learning resulted in significantly better predictions.



Text Preprocessing

The approach we took for single categorical variables was:

1. Make a list of all possible levels of that categorical variable (we'll call this list the *vocab*).
2. Replace each level with its index in the vocab.

3. Create an embedding matrix for this containing a row for each level (i.e., for each item of the vocab).
4. Use this embedding matrix as the first layer of a neural network. (A dedicated embedding matrix can take as inputs the raw vocab indexes created in step 2; this is equivalent to but faster and more efficient than a matrix that takes as input one-hot-encoded vectors representing the indexes.)

We can do nearly the same thing with text! What is new is the idea of a sequence. First we concatenate all of the documents in our dataset into one big long string and split it into words, giving us a very long list of words (or "tokens"). Our independent variable will be the sequence of words starting with the first word in our very long list and ending with the second to last, and our dependent variable will be the sequence of words starting with the second word and ending with the last word.

Our vocab will consist of a mix of common words that are already in the vocabulary of our pretrained model and new words specific to our corpus (cinematographic terms or actors names, for instance). Our embedding matrix will be built accordingly: for words that are in the vocabulary of our pretrained model, we will take the corresponding row in the embedding matrix of the pretrained model; but for new words we won't have anything, so we will just initialize the corresponding row with a random vector.

The stages are:

- Tokenization
- Numericalization (steps 1 and 2)
- Language model data loader creation
- Language model creation

Tokenization

Convert text into a list of *tokens*

- Word based approach: split on spaces
- Subword based approach: split words into substrings
- characters based approach: split a sentence into characters

---

Subword tokenization is another popular method, specially in languages that don't use spaces such as Chinese, but is fast becoming popular in English as well.

Subword tokenization takes place in two steps:

- Analyze a corpus of documents to find the most commonly occurring groups of letters. These become the vocab.
- Tokenize the corpus using this vocab of *subword units*.

The bigger the vocab, the more number of characters can be stored in each token, making more fuller words in the vocab. If the vocab is smaller, inidividual words will be split into smaller and smaller subwords.

`setup` is a special fastai class used to *train* a tokenizer on a text so that it can create the vocab for it.

Numericalization

Default size of the embedding matrix for numericalization is 60k. So any further tokens in the vocab will be replaced with `xxunk`.

Batch Size

If we have a sequence of length 90 and we define batch size as 6, then we have to split the sequence into 6 contiguous parts of length 15 each.

However, we have millions of tokens in a real dataset. For a realistic batchsize of 64, each batch would still contain millions of tokens, which won't fit in the GPU. So instead, we split it horizontally into mini-batches by defining a fixed sequence length for each minibatch.

So the original 90 length batch, broken 6×15 (bs=6)

xxbos	xxmaj	in	this	chapter	,	we	will	go	back	over	the	example	of	classifying
movie	reviews	we	studied	in	chapter	1	and	dig	deeper	under	the	surface	.	xxmaj
first	we	will	look	at	the	processing	steps	necessary	to	convert	text	into	numbers	and
how	to	customize	it	.	xxmaj	by	doing	this	,	we	'll	have	another	example
of	the	preprocessor	used	in	the	data	block	xxup	api	.	\n	xxmaj	then	we
will	study	how	we	build	a	language	model	and	train	it	for	a	while	.

gets broken into three mini-batches

xxbos	xxmaj	in	this	chapter
movie	reviews	we	studied	in
first	we	will	look	at
how	to	customize	it	.
of	the	preprocessor	used	in
will	study	how	we	build

,	we	will	go	back
chapter	1	and	dig	deeper
the	processing	steps	necessary	to
xxmaj	by	doing	this	,
the	data	block	xxup	api
a	language	model	and	train

over	the	example	of	classifying
under	the	surface	.	xxmaj
convert	text	into	numbers	and
we	'll	have	another	example
.	\n	xxmaj	then	we
it	for	a	while	.

Note how the rows are CONTINUOUS over the three minibatches!

Default batch size is 64, and default sequence length for the minicatches is 72.

The independent variable is taken by taking a sequence of length starting from token 1 to taken x. The corresponding dependent variable is the same sequence but starting from token 2 upto token x+1.

The batch size splitting done above for the language model can't be used for the final classification model. It requires the full text to be present in the independent variable.

If the batch size is 128, and the review is 3000 words long, you might end up with something that is too big to be fit in the GPU memory.

Even if we split it into subsequences (72 token long each), they will still be of different sizes becaise the total length (batch*seq*n) is different for different reiews, so n will vary between reviews.

We will expand the shortest texts to make them all the same size. To do this, we use a special padding token that will be ignored by our model. Additionally, to avoid memory issues and improve performance, we will batch together texts

that are roughly the same lengths (with some shuffling for the training set). We do this by (approximately, for the training set) sorting the documents by length prior to each epoch. The result of this is that the documents collated into a single batch will tend of be of similar lengths. We won't pad every batch to the same size, but will instead use the size of the largest document in each batch as the target size.

The sorting and padding are automatically done by the data block API for us when using a `TextBlock`, with `is_lm=False`.

Chapter 12