NLP with RNNs and Attention

- We will start with character RNN which predict the next character in a sequence. They can be classified as
  - Stateless RNN – they learn on random parts of the text
  - Stateful RNN – they preserve the hidden state at end of each iteration

- Then a word RNN will be used for sentiment analysis

- RNN-based Encoder-Decoder architecture is discussed next as a tool to perform machine translation

- Finally, attention mechanisms will be introduced (next lecture)
  - First, for an Encoder-Decoder
  - Then as RNN-free architecture called Transformers

Char RNN

- To create a training set in this case we need to encode each character as an integer. We can use for that Tokenizer from Keras

```
tokenizer = keras.preprocessing.text.Tokenizer(char_level=True)
tokenizer.fit_on_texts([shakespeare_text])
```

- We use option char_level=True to encode characters (not the default choice words). Then we can easily switch between representations. For example:

```
>>> tokenizer.texts_to_sequences(["First"])
[[20, 6, 9, 8, 3]]
>>> tokenizer.sequences_to_texts([[20, 6, 9, 8, 3]])
['f i r s t']
```

Splitting a Sequential Dataset

- Avoid overlap between the training set, the validation set, and the test set. For example, we can take the first 80% of the text for the training set, then the next 10% for validation, and the last 10% for the test.

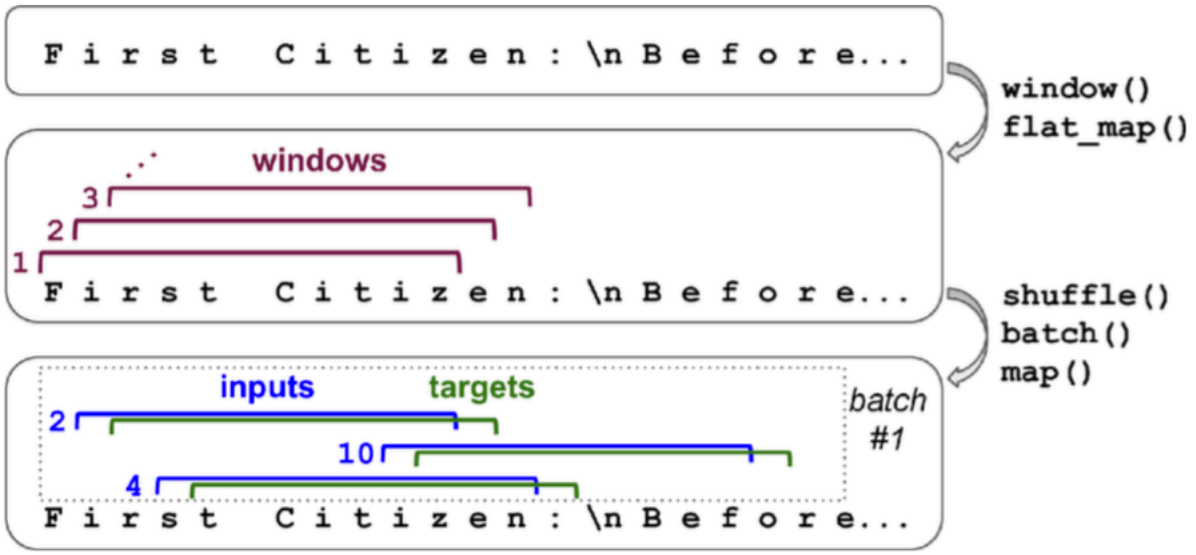
- For time series splitting across time is the general rule – e.g. 2000-2010 for training, 2011-2012 for validation, 2013-2014 for testing.

- This assumes stationarity (patterns the RNN can learn in the past will still be valid in the future). Often this is not a valid assumption (e.g. for financial series) thus caution needs to be exercised in creating the different model building sets

Chopping the Sequential Dataset into Multiple Windows

- The text which is the training set is usually a very long sequence of characters (> 1M) and can't directly be used for training an RNN (why?)
- Instead we can use sliding window to break the text. In this case, we are talking about truncated backpropagation through time
- The `window()` method creates non-overlapping windows by default. When its parameter `shift=1`, the windows overlap by 1 character and this creates the largest possible training set of equal-size chunks (windows of text).
- The result of applying `window()` is a nested dataset (like list of lists). This needs to be converted to a tensor for the modeling and prediction.
- Another transformation that needs to be done corresponds to the i.i.d. assumption (of the training set instances) underlying the Gradient Decent. We need to shuffle the created windows of text.

- A summary of the above text transformations (window length = 11, batch size = 3)



- Last transformation, map(), refers to one-hot encoding (embedding) of characters:

```
dataset = dataset.map(
    lambda X_batch, Y_batch: (tf.one_hot(X_batch, depth=max_id), Y_batch))
```

- We can also add prefetching and the data preparation (the most difficult part of the modeling) is done.

[0, 0, . . , 1, 0, . . , 0] 39 unique

39

Building and Training the Char-RNN Model

- An example of RNN architecture for the Shakespeare texts described in HOML

```
model = keras.models.Sequential([
    keras.layers.GRU(128, return_sequences=True, input_shape=[None, max_id],
                     dropout=0.2, recurrent_dropout=0.2),
    keras.layers.GRU(128, return_sequences=True,
                     dropout=0.2, recurrent_dropout=0.2),
    keras.layers.TimeDistributed(keras.layers.Dense(max_id,
                                                    activation="softmax"))
])
model.compile(loss="sparse_categorical_crossentropy", optimizer="adam")
history = model.fit(dataset, epochs=20)
```

- Parameter max_id=39 because this is the number of distinct characters in the text.
- Examples of generating fake Shakespearean text and the use of temperate are given in the notebook / HOML

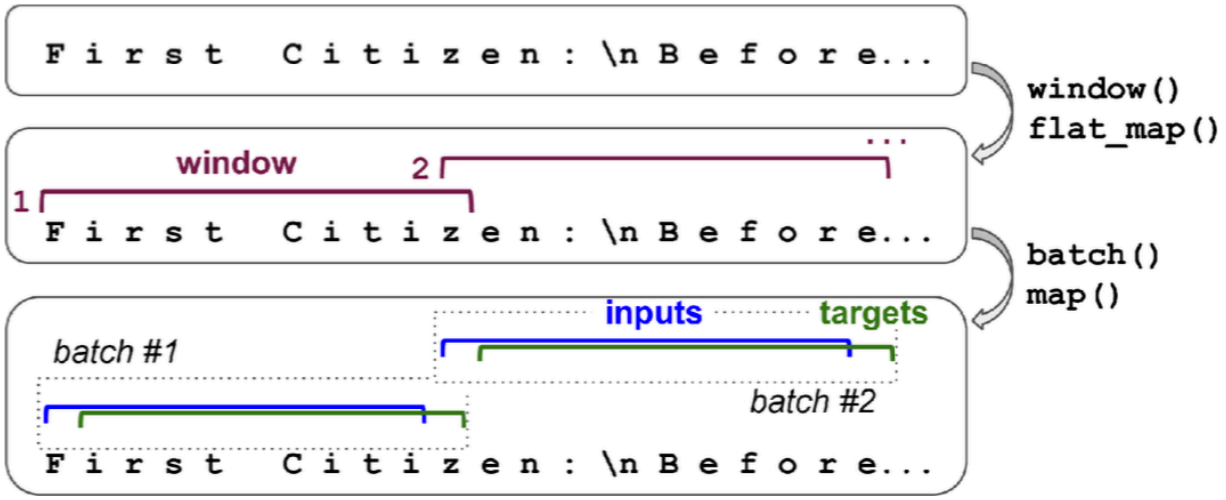
Stateful RNN

- In stateless RNNs: at each training iteration the model starts with a hidden state full of zeros, then it updates this state at each time step, and after the last time step, it throws it away.

- If we keep the final state at the end of a training batch and use it in the next batch, the model can learn long-term patterns. This is stateful RNN.

- Now, the sequences (windows) need to be nonoverlapping which is harder to accomplish than for stateless RNN. If as in the book example, batches with 32 windows are created, then the 1^{st} windows in batches 1 & 2 (window #1 and # 33, correspondingly) are not consecutive.

- The simplest solution is to use batches that contain only 1 window and will look as the schema on next page



- To build a stateful model, we need to set a few parameters such as stateful=True and reset states at the end of each epoch
- After this model is trained, it will only be possible to use it to make predictions for batches of the same size as were used during training. To avoid this restriction, create an identical stateless model, and copy the stateful model's weights to this model.
- The proper batching can be done as in the notebook

Word-level models - Sentiment Analysis

- The IMDb reviews dataset is the “hello world” of NLP: it consists of 50,000 movie reviews in English (25,000 for training, 25,000 for testing) extracted from the famous Internet Movie Database, along with a simple binary target for each review indicating whether it is negative (0) or positive (1).

- Keras provides a simple function to load it:

```
>>> (X_train, y_train), (X_test, y_test) = keras.datasets.imdb.load_data()
>>> X_train[0][:10]
[1, 14, 22, 16, 43, 530, 973, 1622, 1385, 65]
```

- Each integer represents a word. All punctuation was removed, and then words were converted to lowercase, split by spaces, and finally indexed by frequency (so low integers correspond to frequent words).
- The integers 0, 1, and 2 are special: they represent the padding token, the start-of-sequence (SOS) token, and unknown words, respectively. If you want to visualize a review, you can decode it like this:

```
word_index = keras.datasets.imdb.get_word_index()
id_to_word = {id_ + 3: word for word, id_ in word_index.items()}
for id_, token in enumerate(("<pad>", "<sos>", "<unk>")):
    id_to_word[id_] = token

" ".join([id_to_word[id_] for id_ in X_train[0][:10]])
'<sos> this film was just brilliant casting location scenery story'
```

- In a real project, you will have to preprocess the text yourself – see previous lecture. Or you can do that using the same Tokenizer class we used earlier, but this time setting `char_level=False` (which is the default). How to treat spaces?

- A pre-processing function is defined in HOML – it starts by truncating the reviews to the first 300 characters. After some manipulations, a rugged tensor is created. It is consequently turned in to a dense tensor by padding.

‘

- Then a vocabulary is created (n=10K) & “embedding” (look-up) is added

- A model is then fit – an architecture is shown below

```
embed_size = 128
model = keras.models.Sequential([
    keras.layers.Embedding(vocab_size + num_oov_buckets, embed_size,
                           input_shape=[None]),
    keras.layers.GRU(128, return_sequences=True),
    keras.layers.GRU(128),
    keras.layers.Dense(1, activation="sigmoid")
])
model.compile(loss="binary_crossentropy", optimizer="adam",
              metrics=["accuracy"])
history = model.fit(train_set, epochs=5)
```

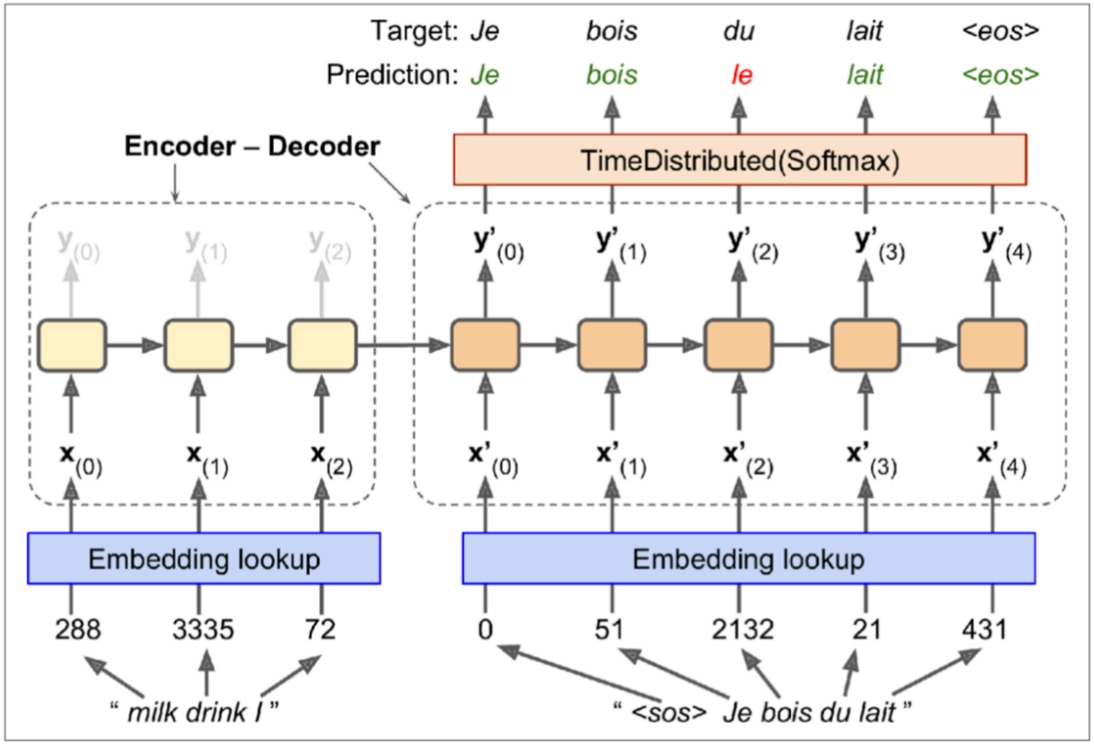
- First layer embeds the word IDs into a 128-dim space (hyperparameter)
- The rest of the model is fairly straightforward:
  - 2 GRU layers; 2nd one returning only the output of the last time step
  - The output layer is just a single neuron using the sigmoid activation function to output the estimated probability that the review expresses a positive sentiment regarding the movie.

- Additional model tuning & improvements
  - Masking
    - e.g. tells the model directly to ignore padding tokens (with ID=0) if `mask_zero=True` is passed in the Embedding layer
    - When a recurrent layer encounters a masked time step, it simply copies the output from the previous time step
    - But using masking can slow down the fitting – see ”scorpion note” in the text
  - Reusing pretrained embeddings
- So far we applied RNNs to
  - Time Series
  - Char-RNN – for text generation
  - Word-level RNN – for sentiment analysis

Next …

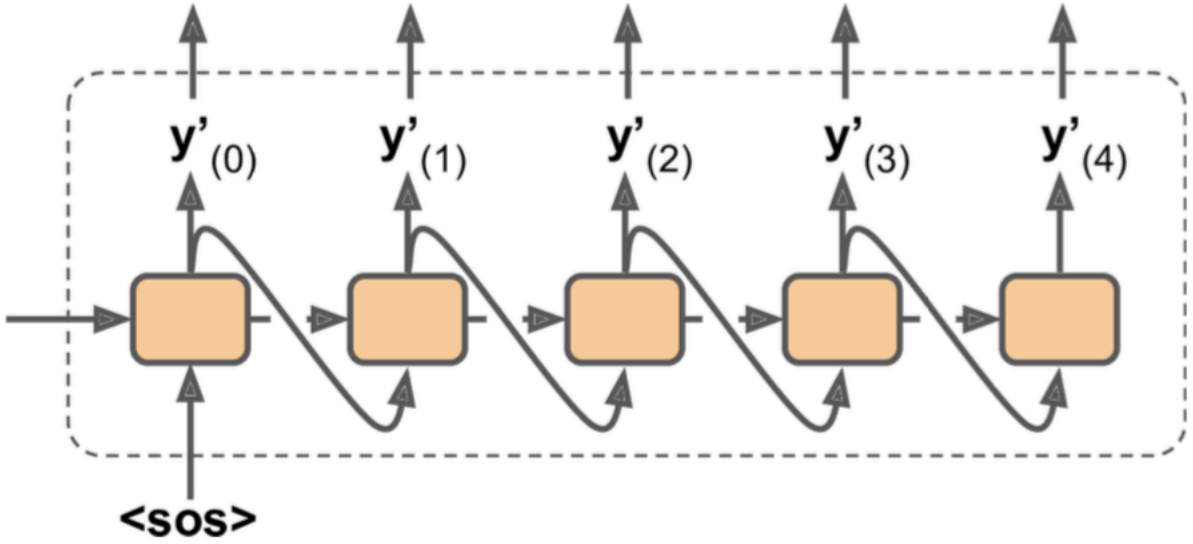
An Encoder–Decoder Network for Neural Machine Translation (NMT)

- We are looking here at a simple NMT model which translates English to French
- French translations are also used as inputs to the decoder, but shifted back by one step. In other words, the decoder is given as input the word that it should have output at the previous step (regardless of what it actually output).
- For the very first word, it is given the start-of-sequence (SOS) token. The decoder is expected to end the sentence with an end-of-sequence (EOS) token.



- At each step, the decoder outputs a score for each word in the output vocabulary (i.e., French), and then the softmax layer turns these scores into probabilities. E.g., at the first step the word “Je” may have a probability of 20%, “Tu” may have a probability of 1%, and so on. The word with the highest probability is output.

- At inference time (after training), there is no target sentence to feed to the decoder. Instead, simply feed the decoder the word that it output at the previous step, as shown below (this will require an embedding lookup that is not indicated).



- We assumed that all inputs have a constant length. When sentences have very different lengths, we group them in buckets with similar lengths: 1-6, 7-12, 13-18 etc., and use padding for the shorter ones, e.g.

  "I drink milk" -> "<pad><pad><pad> milk I drink"

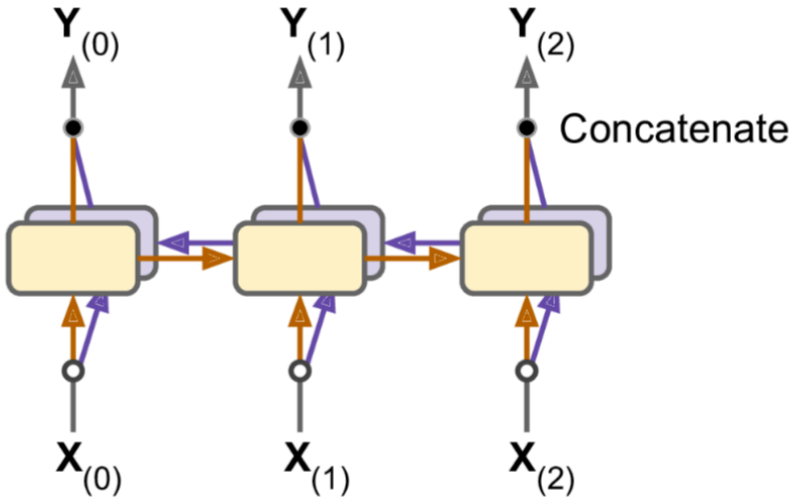
- Sampled softmax – with large vocabularies, only logits of the correct word and a random sample of incorrect words is output in the loss function to reduce computation time. Model (without data?) in .ipynb

```
import tensorflow_addons as tfa

encoder_inputs = keras.layers.Input(shape=[None], dtype=np.int32)
decoder_inputs = keras.layers.Input(shape=[None], dtype=np.int32)
sequence_lengths = keras.layers.Input(shape=[], dtype=np.int32)

embeddings = keras.layers.Embedding(vocab_size, embed_size)
encoder_embeddings = embeddings(encoder_inputs)
decoder_embeddings = embeddings(decoder_inputs)

encoder = keras.layers.LSTM(512, return_state=True)
encoder_outputs, state_h, state_c = encoder(encoder_embeddings)
encoder_state = [state_h, state_c]

sampler = tfa.seq2seq.sampler.TrainingSampler()

decoder_cell = keras.layers.LSTMCell(512)
output_layer = keras.layers.Dense(vocab_size)
decoder = tfa.seq2seq.basic_decoder.BasicDecoder(decoder_cell, sampler,
                                                 output_layer=output_layer)
final_outputs, final_state, final_sequence_lengths = decoder(
    decoder_embeddings, initial_state=encoder_state,
    sequence_length=sequence_lengths)
Y_proba = tf.nn.softmax(final_outputs.rnn_output)

model = keras.Model(inputs=[encoder_inputs, decoder_inputs, sequence_lengths],
                    outputs=[Y_proba])
```

Bidirectional RNNs

- For many NLP tasks, such as NMT, it is often preferable to look ahead at the next words before encoding a given word.

  Consider the phrases “the Queen of the United Kingdom,” “the queen of hearts,” and “the queen bee”: to properly encode the word “queen,” you need to look ahead. One solution is presented below:



Beam Search (read)

- How can we give the model a chance to go back and fix mistakes it made earlier? One of the most common solutions is *beam search*: it keeps track of a short list of the *k* most promising sentences (say, the top three), and at each decoder step it tries to extend them by one word, keeping only the *k* most likely sentences. The parameter *k* is called the *beam width*.

- Unfortunately, this model will be really bad at translating long sentences. Once again, the problem comes from the limited short- term memory of RNNs. *Attention mechanisms* are the game-changing innovation that addressed this problem.

Next Lecture

Attention Mechanisms

Recent Innovations in Language Models