Hands-on Machine Learning

16. Natural Language Processing

Building a Character RNN

- We want to train an RNN to predict the next character in a sentence.
- This *char-RNN* can then be used to generate novel text, one character at a time.
- Here is a sample of the text generated by such a char-RNN model after it was trained on all of Shakespeare's works:

> PANDARUS:
> Alas, I think he shall be come approached and the day
> When little srain would be attain'd into being never fed,
> And who is but a chain and subjects of his death,
> I should not sleep.

1.
Creating the Training Dataset

Creating the Training Dataset

- Download all of Shakespeare's works:

```
import tensorflow as tf

shakespeare_url = "https://homl.info/shakespeare"  # shortcut URL
filepath = tf.keras.utils.get_file("shakespeare.txt", shakespeare_url)
with open(filepath) as f:
    shakespeare_text = f.read()
```

- Use Keras' `TextVectorization` layer to encode this text:

```
text_vec_layer = tf.keras.layers.TextVectorization(split="character",
                                                   standardize="lower")
text_vec_layer.adapt([shakespeare_text])
encoded = text_vec_layer([shakespeare_text])[0]
```

Creating the Training Dataset

- Each character is now mapped to an integer, starting at 2.
- The TextVectorization layer reserved the value 0 for padding tokens, and it reserved 1 for unknown characters.

```
encoded -= 2  # drop tokens 0 (pad) and 1 (unknown), which we will not use
n_tokens = text_vec_layer.vocabulary_size() - 2  # number of distinct chars = 39
dataset_size = len(encoded)  # total number of chars = 1,115,394
```

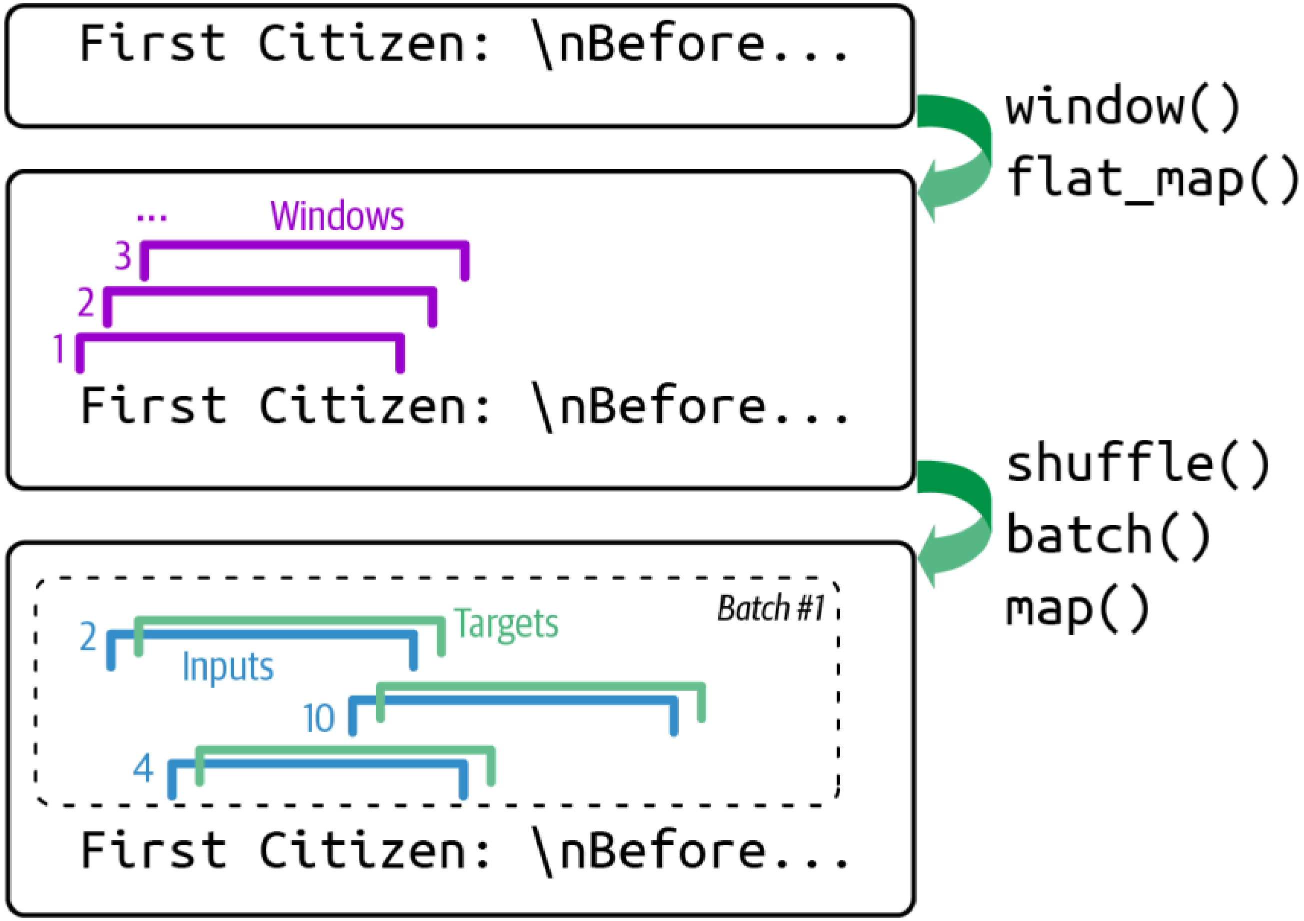
- We can turn this very long sequence into a dataset of windows that we can then use to train a sequence-to-sequence RNN.
- The targets are similar to the inputs, but shifted by one time step into the future.
- Example. Input: “to be or not to b” Target: “o be or not to be”

Creating the Training Dataset

- Convert a long sequence of character IDs into a dataset of input/target window pairs:

```
def to_dataset(sequence, length, shuffle=False, seed=None, batch_size=32):
    ds = tf.data.Dataset.from_tensor_slices(sequence)
    ds = ds.window(length + 1, shift=1, drop_remainder=True)
    ds = ds.flat_map(lambda window_ds: window_ds.batch(length + 1))
    if shuffle:
        ds = ds.shuffle(100_000, seed=seed)
    ds = ds.batch(batch_size)
    return ds.map(lambda window: (window[:, :-1], window[:, 1:])).prefetch(1)
```

- If the `sequence` is [1, 2, 3, 4, 5, 6, 7, 8, 9], with `length=3`:
  Possible inputs: ([1, 2, 3], [2, 3, 4], ...)
  Corresponding targets: ([2, 3, 4], [3, 4, 5], ...)


First Citizen: \nBefore...
window()
flat_map()
...
Windows
3
2
1
First Citizen: \nBefore...
shuffle()
batch()
map()
Batch #1
2
Targets
Inputs
10
4
First Citizen: \nBefore...

Creating the Training Dataset

```python
# There's just one sample in this dataset: the input represents "to b" and the
# output represents "o be"
list(to_dataset(text_vec_layer(["To be"])[0], length=4))
```

```
[(<tf.Tensor: shape=(1, 4), dtype=int64, numpy=array([[ 4,  5,  2, 23]])>,
  <tf.Tensor: shape=(1, 4), dtype=int64, numpy=array([[ 5,  2, 23,  3]])>)]
```

- We use 90% of the text for training, 5% for validation, and 5% for testing:

```python
length = 100
tf.random.set_seed(42)
train_set = to_dataset(encoded[:1_000_000], length=length, shuffle=True, seed=42)
valid_set = to_dataset(encoded[1_000_000:1_060_000], length=length)
test_set = to_dataset(encoded[1_060_000:], length=length)
```

2.
Building and Training the Char-RNN Model

The Char-RNN Model

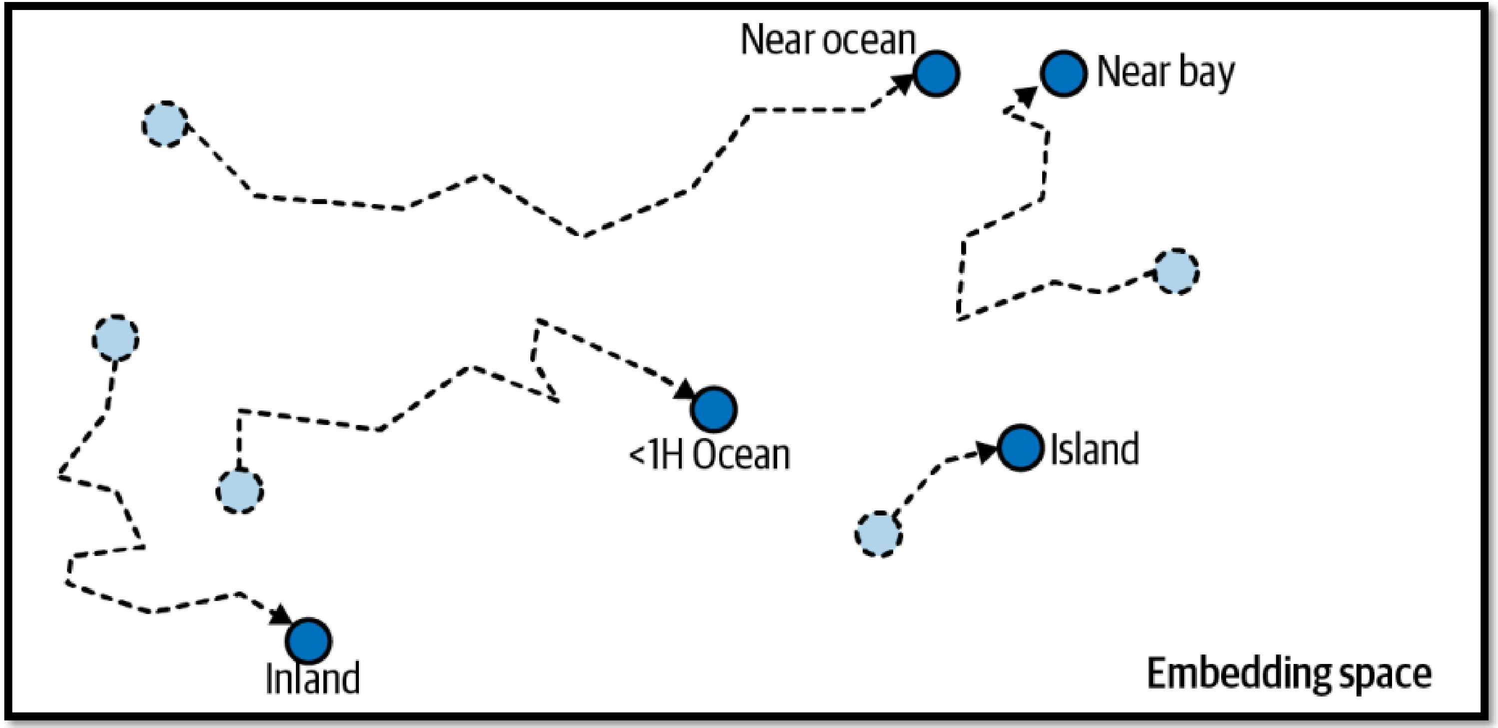
- We build and train a model with one GRU layer composed of 128 units.

```
model = tf.keras.Sequential([
    tf.keras.layers.Embedding(input_dim=n_tokens, output_dim=16),
    tf.keras.layers.GRU(128, return_sequences=True),
    tf.keras.layers.Dense(n_tokens, activation="softmax")
])
model.compile(loss="sparse_categorical_crossentropy", optimizer="nadam",
              metrics=["accuracy"])
model_ckpt = tf.keras.callbacks.ModelCheckpoint(
    "my_shakespeare_model", monitor="val_accuracy", save_best_only=True)
history = model.fit(train_set, validation_data=valid_set, epochs=10,
                    callbacks=[model_ckpt])
```

Embedding Layer

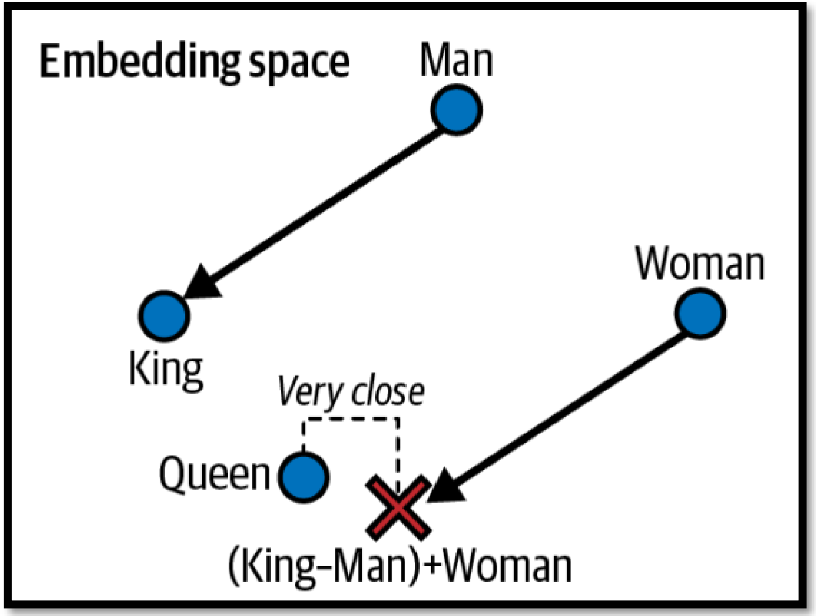
- An embedding is a dense representation of some higher-dimensional data, such as a category, or a word in a vocabulary.
  - If there are 50,000 possible categories, then one-hot encoding would produce a 50,000-dimensional sparse vector but an embedding would be a comparatively small dense vector; e.g. with just 100 dimensions.
- In deep learning, embeddings are initialized randomly, and they are trained by gradient descent, along with the other model parameters.
  - These embeddings are trainable, and will improve during training;
- Representation Learning: The better representation, makes it easier for the neural network to make accurate predictions, so training tends to make embeddings useful representations of the categories.

Embedding Layer



Word Embeddings

- If you compute *King – Man + Woman* (adding and subtracting the embedding vectors of these words), then the result will be very close to the embedding of the word *Queen*.
- Similarly, *Madrid–Spain+France*, is close to *Paris*.



The Char-RNN Model

- We build and train a model with one GRU layer composed of 128 units.

```
model = tf.keras.Sequential([
    tf.keras.layers.Embedding(input_dim=n_tokens, output_dim=16),
    tf.keras.layers.GRU(128, return_sequences=True),
    tf.keras.layers.Dense(n_tokens, activation="softmax")
])
model.compile(loss="sparse_categorical_crossentropy", optimizer="nadam",
              metrics=["accuracy"])
model_ckpt = tf.keras.callbacks.ModelCheckpoint(
    "my_shakespeare_model", monitor="val_accuracy", save_best_only=True)
history = model.fit(train_set, validation_data=valid_set, epochs=10,
                    callbacks=[model_ckpt])
```

Add Text Preprocessing

- Wrap the model in a final model that contains text preprocessing:

```
shakespeare_model = tf.keras.Sequential([
    text_vec_layer,
    tf.keras.layers.Lambda(lambda X: X - 2),  # no <PAD> or <UNK> tokens
    model
])
```

- Use it to predict the next character in a sentence:

```
y_proba = shakespeare_model.predict(["To be or not to b"])[0, -1]
y_pred = tf.argmax(y_proba)  # choose the most probable character ID
text_vec_layer.get_vocabulary()[y_pred + 2]
```

```
'e'
```

3.
Generating Fake Shakespearean Text

Softmax Distribution

- The standard softmax function $\sigma: R^K \to (0,1)^K$, where $K > 1$, takes a vector $\mathbf{z} = (z_1, \dots, z_K) \in R^K$ and computes each component of vector $\sigma(\mathbf{z}) \in (0,1)^K$:

$$\sigma(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

- Example: $\sigma(1,2,8) \approx (0.001, 0.002, 0.997)$.
- Softmax with temperature T:

$$\sigma(\mathbf{z})_i = \frac{e^{z_i/T}}{\sum_{j=1}^{K} e^{z_j/T}}$$

- A higher temperature results in a more uniform output distribution (i.e. "more random"), while a lower temperature results in a sharper output distribution, with maximum value dominating.

Generating New Text

- *Greedy decoding:* to generate new text, feed the char-RNN model some text, make it predict *the most likely* next letter, add it to the end of the text, then give the extended text to the model to guess the next letter.
  - In practice this often leads to the same words being repeated!
- We can sample the next character randomly, with a probability equal to the estimated probability, using `tf.random.categorical`() function.
  - This will generate more diverse and interesting text.
  - The `categorical`() function samples random class indices, given the class log probabilities (logits):

```
log_probas = tf.math.log([[0.5, 0.4, 0.1]])  # probas = 50%, 40%, and 10%
tf.random.categorical(log_probas, num_samples=8)  # draw 8 samples
```

```
<tf.Tensor: shape=(1, 8), dtype=int64, numpy=array([[0, 1, 0, 2, 1, 0, 0, 1]])>
```

Sequence-to-Vector Network

- To have more control over the diversity of the generated text, we can divide the logits by a number called the *temperature*.
  - A temperature close to zero favors high-probability characters, while a high temperature gives all characters an equal probability.

```
def next_char(text, temperature=1):
    y_proba = shakespeare_model.predict([text])[0, -1:]
    rescaled_logits = tf.math.log(y_proba) / temperature
    char_id = tf.random.categorical(rescaled_logits, num_samples=1)[0, 0]
    return text_vec_layer.get_vocabulary()[char_id + 2]
```

- We can write a helper function that call `next_char`() repeatedly:

```
def extend_text(text, n_chars=50, temperature=1):
    for _ in range(n_chars):
        text += next_char(text, temperature)
    return text
```

Generated Text

```
print(extend_text("To be or not to be", temperature=0.01))
```

```
To be or not to be the duke
as it is a proper strange death,
and the
```

```
print(extend_text("To be or not to be", temperature=1))
```

```
To be or not to behold?

second push:
gremio, lord all, a sistermen,
```

```
print(extend_text("To be or not to be", temperature=100))
```

```
To be or not to bef ,mt'&o3fpadm!$
wh!nse?bws3est--vgerdjw?c-y-ewznq
```

Training RNNs

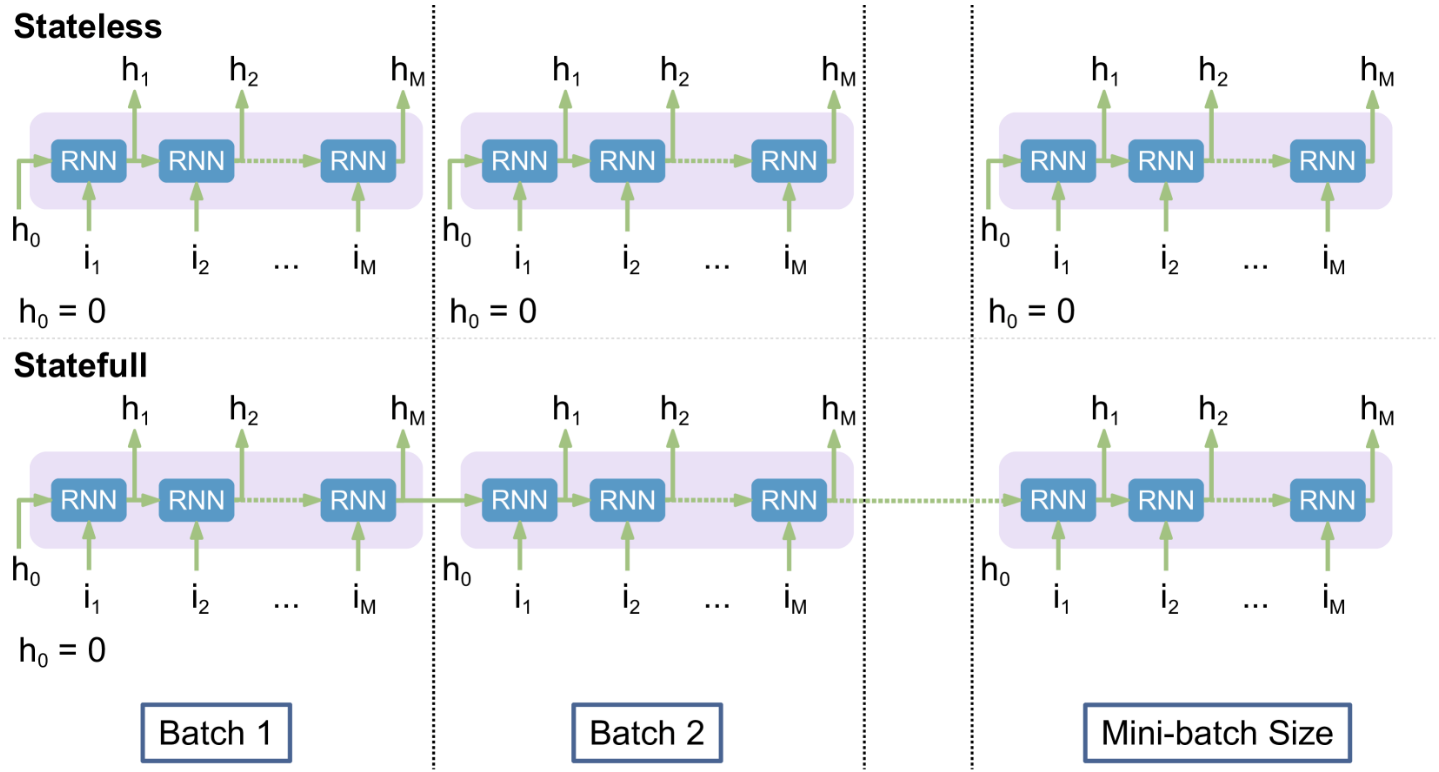
- To generate more convincing text, a common technique is to sample only from the top k characters, or only from the smallest set of top characters whose total probability exceeds some threshold (this is called *nucleus sampling*).
- Alternatively, you could try using *beam search*, or using more GRU layers and more neurons per layer, training for longer, and adding some regularization if needed.
- Also note that the model is currently incapable of learning patterns longer than `length`, which is just 100 characters.
  - You could try making this window larger, but it will also make training harder, and even LSTM and GRU cells cannot handle very long sequences.
  - An alternative approach is to use a stateful RNN.

4.
Stateful RNN

Stateless vs. Stateful RNN

- *Stateless RNN*: 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 as it is not needed anymore.
- *Stateful RNN*: if we instruct the RNN to preserve this final state after processing a training batch and use it as the initial state for the next training batch.
  - This way the model could learn long-term patterns despite only backpropagating through short sequences.

Stateless vs. Stateful RNN



When to use stateful RNN?

- A stateful RNN only makes sense if each input sequence in a batch starts exactly where the corresponding sequence in the previous batch left off.
  - Use sequential and non-overlapping input sequences (rather than the shuffled and overlapping sequences we used to train stateless RNNs).
- When creating the `tf.data.Dataset`, use `shift=length` (instead of `shift=1`) when calling the `window`() method.
- We must *not* call the `shuffle`() method.

Batching for Stateful RNN

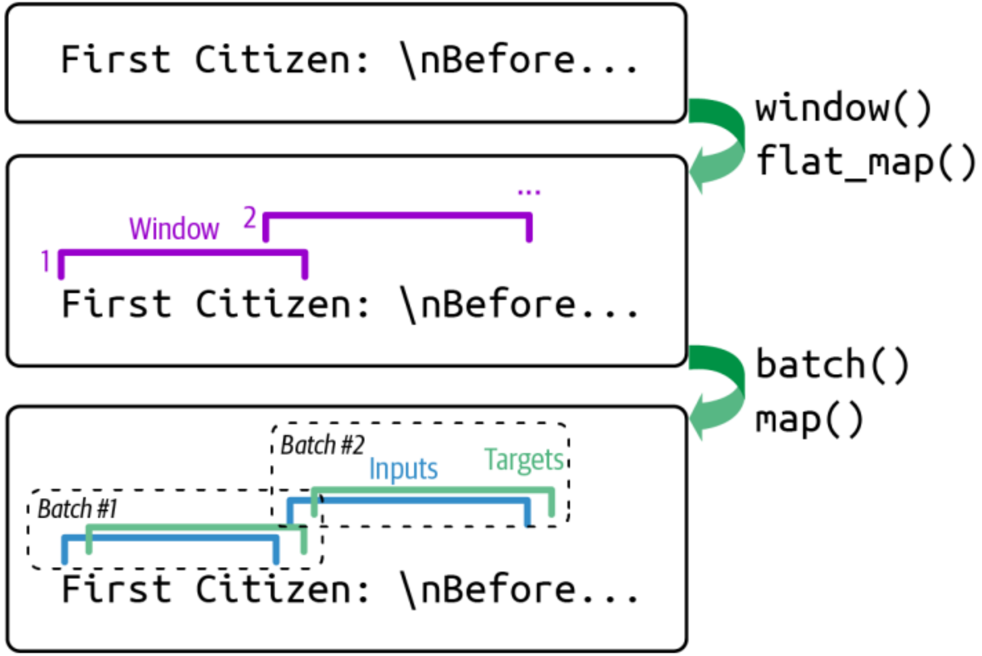
- If we call `batch`(32), then 32 consecutive windows would be put in the same batch, and the following batch would not continue each of these windows where it left off.
  - The simplest solution to this problem is to just use a batch size of 1.

```
def to_dataset_for_stateful_rnn(sequence, length):
    ds = tf.data.Dataset.from_tensor_slices(sequence)
    ds = ds.window(length + 1, shift=length, drop_remainder=True)
    ds = ds.flat_map(lambda window: window.batch(length + 1)).batch(1)
    return ds.map(lambda window: (window[:, :-1], window[:, 1:])).prefetch(1)

stateful_train_set = to_dataset_for_stateful_rnn(encoded[:1_000_000], length)
stateful_valid_set = to_dataset_for_stateful_rnn(encoded[1_000_000:1_060_000], length)
stateful_test_set = to_dataset_for_stateful_rnn(encoded[1_060_000:], length)
```

Batching for Stateful RNN

- Preparing a dataset of consecutive sequence fragments for a stateful RNN:



Create the Stateful RNN

```
model = tf.keras.Sequential([
    tf.keras.layers.Embedding(input_dim=n_tokens, output_dim=16, batch_input_shape=[1, None]),
    tf.keras.layers.GRU(128, return_sequences=True, stateful=True),
    tf.keras.layers.Dense(n_tokens, activation="softmax")
])
```

- At the end of each epoch, we need to reset the states before we go back to the beginning of the text. For this, we can use a small custom Keras callback:

```
class ResetStatesCallback(tf.keras.callbacks.Callback):
    def on_epoch_begin(self, epoch, logs):
        self.model.reset_states()
```

Create the Stateful RNN

- Use a different directory to save the checkpoints:

```
model_ckpt = tf.keras.callbacks.ModelCheckpoint(
    "my_stateful_shakespeare_model",
    monitor="val_accuracy",
    save_best_only=True)
```

- Compile the model and train it using our callback:

```
model.compile(loss="sparse_categorical_crossentropy", optimizer="nadam",
              metrics=["accuracy"])
history = model.fit(stateful_train_set, validation_data=stateful_valid_set,
                    epochs=10, callbacks=[ResetStatesCallback(), model_ckpt])
```

- It will only be possible to use the trained stateful RNN to make predictions for batches of the same size as were used during training.

Converting Stateful RNN to Stateless

- To avoid same size batches restriction, create an identical *stateless* model, and copy the stateful model's weights to this model:

```
stateless_model = tf.keras.Sequential([
    tf.keras.layers.Embedding(input_dim=n_tokens, output_dim=16),
    tf.keras.layers.GRU(128, return_sequences=True),
    tf.keras.layers.Dense(n_tokens, activation="softmax")
])
```

```
stateless_model.build(tf.TensorShape([None, None]))
```

```
stateless_model.set_weights(model.get_weights())
```

```
shakespeare_model = tf.keras.Sequential([
    text_vec_layer,
    tf.keras.layers.Lambda(lambda X: X - 2),  # no <PAD> or <UNK> tokens
    stateless_model
])
```

Unsupervised Pretraining in NLP

- Although a char-RNN model is just trained to predict the next character, this simple task actually requires it to learn higher-level tasks as well.
  - For example, to find the next character after "Great movie, I really", it's helpful to understand that the sentence is positive, so what follows is more likely to be the letter "l" (for "loved") rather than "h" (for "hated").

- A paper by OpenAI researchers describes how the authors trained a big char-RNN-like model on a large dataset, and found that one of the neurons acted as an excellent sentiment analysis classifier: although the model was trained without labels, the *sentiment neuron* reached state-of-the-art performance on sentiment analysis benchmarks.
  - This motivated unsupervised pretraining in NLP.