COMP7220/8220: Recurrent Neural Networks

Mark Dras

May 2020

Outline

1. **Basics of RNNs**
   - Recurrent Neurons
   - Training RNNs for Time Series
2. **More Advanced Cells**
3. **Natural Language**
   - Generating with Character-Level RNNs
   - Sentiment Analysis

Introduction

Idea Behind RNNs

- Our previous NNs modelled static data with fixed size inputs.
- Many phenomena are sequential and of arbitrary length, including time series.
  - Text: translation, speech-to-text (ASR), sentiment analysis.
  - Time series: stock prices.
  - Other: video data, autonomous driving systems.

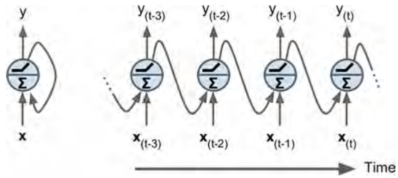
Outline

1. **Basics of RNNs**
   - Recurrent Neurons
   - Training RNNs for Time Series
2. **More Advanced Cells**
3. **Natural Language**
   - Generating with Character-Level RNNs
   - Sentiment Analysis

Individual Recurrent Neurons

Structure

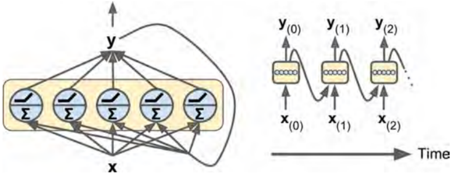
- Similar to a feedforward NN, except connections also point backwards.
- Example with one neuron.
    - Receives inputs ($\mathbf{x}_{(t)}$), produces output ($y_{(t)}$); also receives its own output from previous step ($y_{(t-1)}$).



Layers of Recurrent Neurons

Structure

- Each neuron receives inputs ($\mathbf{x}_{(t)}$), and outputs from previous step ($y_{(t-1)}$).
- Consequently, each neuron has two sets of weights, $\mathbf{w}_x$ and $\mathbf{w}_y$ resp.



Equations

Single Neuron

$$\mathbf{y}_{(t)} = \phi\left(\mathbf{x}_{(t)}^T \cdot \mathbf{w}_x + \mathbf{y}_{(t-1)}^T \cdot \mathbf{w}_y + b\right) \tag{1}$$

Layer of Neurons

$$\mathbf{Y}_{(t)} = \phi\left(\mathbf{X}_{(t)}^T \cdot \mathbf{W}_x + \mathbf{Y}_{(t-1)}^T \cdot \mathbf{W}_y + \mathbf{b}\right) \tag{2}$$

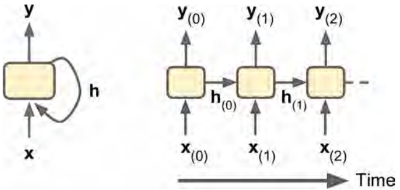
where

- $\mathbf{Y}_{(t)}$ is an $m \times n_{\text{neurons}}$ matrix containing the layer's outputs at time step t for each instance in the mini-batch (m is the number of instances in the mini-batch and n_{neurons} is the number of neurons).
- $\mathbf{X}_{(t)}$ is an $m \times n_{\text{inputs}}$ matrix containing the inputs for all instances (n_{inputs} is the number of input features).
- $\mathbf{W}_x$ is an $n_{\text{inputs}} \times n_{\text{neurons}}$ matrix containing the connection weights for the inputs of the current time step.
- $\mathbf{W}_y$ is an $n_{\text{neurons}} \times n_{\text{neurons}}$ matrix containing the connection weights for the outputs of the previous time step.
- The weight matrices $\mathbf{W}_x$ and $\mathbf{W}_y$ are often concatenated into a single weight matrix $\mathbf{W}$ of shape $(n_{\text{inputs}} + n_{\text{neurons}}) \times n_{\text{neurons}}$.
- $\mathbf{b}$ is a vector of size n_{neurons} containing each neuron's bias term.

Memory Cells

Definition

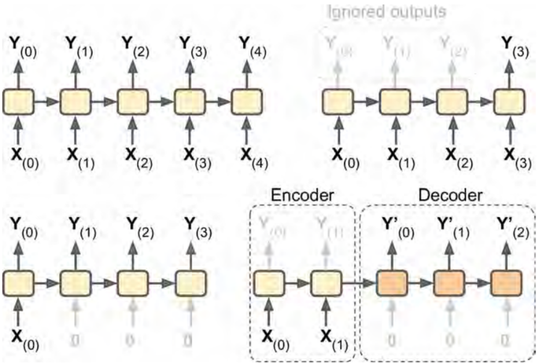
- A part of a neural network that preserves some state across time steps is called a MEMORY CELL.
- In general a cell's state at time step t, denoted $\mathbf{h}_{(t)}$, is a function of some inputs at that time step and its state at the previous time step: $\mathbf{h}_{(t)} = f(\mathbf{h}_{(t-1)}, \mathbf{x}_{(t)})$.
  - Ditto for output at time step t, denoted $\mathbf{y}_{(t)}$.



Input and Output Sequences

Four Options

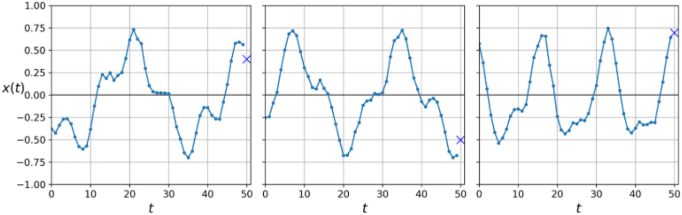
Combinations of sequence and single vector.



Outline

1. **Basics of RNNs**
   - Recurrent Neurons
   - Training RNNs for Time Series
2. **More Advanced Cells**
3. **Natural Language**
   - Generating with Character-Level RNNs
   - Sentiment Analysis

Sample Time Series



Datasets

Example

- Each training instance is a randomly selected sequence of 50 consecutive values from the time series.
  - When dealing with time series (and other types of sequences such as sentences), the input features are generally represented as 3D arrays of shape `[batch size, time steps, dimensionality]`, where dimensionality is 1 for univariate time series and more for multivariate time series.
- Targets are column vectors.

Artificial Dataset

```
n_steps = 50
series = generate_time_series(10000, n_steps + 1) #<- fn of sin
X_train, y_train = series[:7000, :n_steps], series[:7000, -1]
X_valid, y_valid = series[7000:9000, :n_steps], series[7000:9000, -1]
X_test, y_test = series[9000:, :n_steps], series[9000:, -1]
```

Baselines

Predict Most Recent Value (Naive)

```
>>> y_pred = X_valid[:, -1]
>>> np.mean(keras.losses.mean_squared_error(y_valid, y_pred))
0.020211367
```

Small Fully Connected Network

```
model = keras.models.Sequential([  # <- linear regression
    keras.layers.Flatten(input_shape=[50, 1]),
    keras.layers.Dense(1)
])
# gets MSE of 0.004 after 20 epochs
```

A Simple RNN

SimpleRNN

```
model = keras.models.Sequential([
  keras.layers.SimpleRNN(1, input_shape=[None, 1])  #<- single neuron
])
```

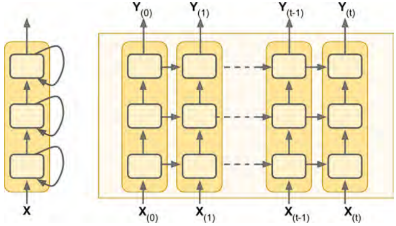
Remarks

- Don't specify length of input sequences ⇒ first dimension `None`.
- By default uses `tanh`.
- By default, recurrent layers in Keras only return the final output — for output per time step, set `return_sequences=True`.
- Compare parameters: fully connected model has one per input per time step + bias (=51), SimpleRNN has one per input and per hidden state dimension (=3).

Result

MSE = 0.014

A Deeper RNN



Three Layer

```
model = keras.models.Sequential([
    keras.layers.SimpleRNN(20, return_sequences=True, input_shape=[None, 1]),
    keras.layers.SimpleRNN(20, return_sequences=True),
    keras.layers.SimpleRNN(1)  #<- can use Dense layer and change activation
])  # MSE 0.003
```

Note

If `return_sequences` not set, RNN layer returns 2D (last time step) instead of 3D.

Predicting Multiple Time Steps (1)

Option #1

Use existing model:

- generate one step at a time;
- feed in for next timestep.

Modifying the Existing Model

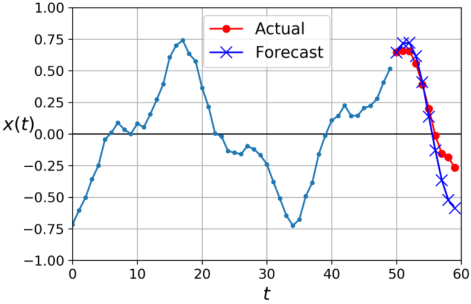
```
series = generate_time_series(1, n_steps + 10)
X_new, Y_new = series[:, :n_steps], series[:, n_steps:]
X = X_new
for step_ahead in range(10):
    y_pred_one = model.predict(X[:, step_ahead:])[:, np.newaxis, :]
    X = np.concatenate([X, y_pred_one], axis=1)

Y_pred = X[:, n_steps:]
```

Predicting Multiple Time Steps (3)

Effects

- Errors compound.
- MSE $= 0.029$.
  - Cf. 0.223 for naive, 0.0188 for linear.



Predicting Multiple Time Steps (4)

Option #2

- Predict all 10 at once (wrt target Y).
- In code: need to change targets.

Modifying the Existing Model

```
series = generate_time_series(10000, n_steps + 10)
X_train, Y_train = series[:7000, :n_steps], series[:7000, -10:, 0]
X_valid, Y_valid = series[7000:9000, :n_steps], series[7000:9000, -10:, 0]
X_test, Y_test = series[9000:, :n_steps], series[9000:, -10:, 0]

model = keras.models.Sequential([
    keras.layers.SimpleRNN(20, return_sequences=True, input_shape=[None, 1]),
    keras.layers.SimpleRNN(20),
    keras.layers.Dense(10)  # <- predict 10 outputs at last layer
])
# MSE = 0.008
```

Predicting Multiple Time Steps (5)

Option #3

Predict all 10 at each time step.

- $\Rightarrow$ sequence-to-vector RNN $\rightarrow$ sequence-to-sequence RNN.

Modifying the Existing Model

```
Y = np.empty((10000, n_steps, 10)) # each target is a sequence of 10D vectors
for step_ahead in range(1, 10 + 1):
    Y[:, :, step_ahead - 1] = series[:, step_ahead:step_ahead + n_steps, 0]
Y_train = Y[:7000]
Y_valid = Y[7000:9000]
Y_test = Y[9000:]

model = keras.models.Sequential([
    keras.layers.SimpleRNN(20, return_sequences=True, input_shape=[None, 1]),
    keras.layers.SimpleRNN(20, return_sequences=True),
    keras.layers.TimeDistributed(keras.layers.Dense(10))  #<- applied at every time step
])
```

Predicting Multiple Time Steps (6)

Calculating Final MSE

```
def last_time_step_mse(Y_true, Y_pred):
    # regular MSE is over all outputs (fine for training)
    return keras.metrics.mean_squared_error(Y_true[:, -1], Y_pred[:, -1])

optimizer = keras.optimizers.Adam(lr=0.01)
model.compile(loss="mse", optimizer=optimizer, metrics=[last_time_step_mse])
```

Final MSE

$\text{MSE} = 0.006$

Outline

1. Basics of RNNs
   - Recurrent Neurons
   - Training RNNs for Time Series
2. **More Advanced Cells**
3. Natural Language
   - Generating with Character-Level RNNs
   - Sentiment Analysis

The Difficulty of Training over Many Time Steps

Issues

- Training is slow.
  - $\Rightarrow$ Unroll the RNN only over a limited number of time steps during training (TRUNCATED BACKPROPAGATION THROUGH TIME).
- Gradients can be unstable.
- Memory of the first inputs gradually fades away.

Unstable Gradients

Issue

- Gradient Descent could increase outputs slightly at first time step, then compound.
  - Same weights used at each timestep.
- Outputs can explode with non-saturating activation.
  - Hence `tanh` default.
- Gradients similarly explode.

Normalisation

- Batch Normalisation doesn't really work.
  - Same BN would be used at each time step, with same weight.
- Layer Normalisation (across features) works better.

Layer Normalisation

A Cell with Normalisation

```
class LNSimpleRNNCell(keras.layers.Layer):
    def __init__(self, units, activation="tanh", **kwargs):
        super().__init__(**kwargs)
        self.state_size = units
        self.output_size = units
        self.simple_rnn_cell = keras.layers.SimpleRNNCell(units,
                                                          activation=None)
        self.layer_norm = keras.layers.LayerNormalization()
        self.activation = keras.activations.get(activation)
    def call(self, inputs, states):
        outputs, new_states = self.simple_rnn_cell(inputs, states)
        norm_outputs = self.activation(self.layer_norm(outputs))
        return norm_outputs, [norm_outputs]
```

Model

```
model = keras.models.Sequential([
    keras.layers.RNN(LNSimpleRNNCell(20), return_sequences=True,
                     input_shape=[None, 1]),
    keras.layers.RNN(LNSimpleRNNCell(20), return_sequences=True),
    keras.layers.TimeDistributed(keras.layers.Dense(10))
])
```

Dropout

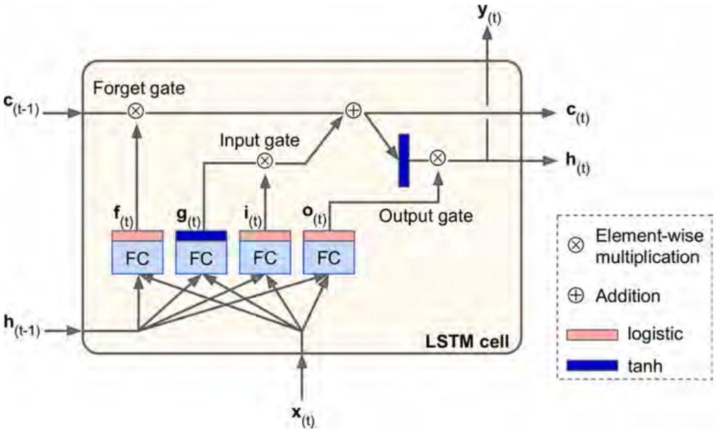
Implementing Dropout

- Could use a custom cell to apply dropout between each time step.
- However, already implemented: all recurrent layers and cells provided by Keras have dropout parameters.
  - The `dropout` hyperparameter defines the dropout rate to apply to the inputs (at each time step).
  - The `recurrent_dropout` hyperparameter defines the dropout rate for the hidden states (also at each time step).

Tackling the Short-Term Memory Problem

Long Short-Term Memory (LSTM)

Learn which parts of long-distant history to remember.



Long Short-Term Memory (LSTM) (1)

Components

$\mathbf{h}_{(t)}$, short-term state; $\mathbf{c}_{(t)}$, long-term state.

Internal Layers

Current input vector $\mathbf{x}_{(t)}$ and the previous short-term state $\mathbf{h}_{(t-1)}$ are fed to four different fully connected layers:

- The main layer outputs $\mathbf{g}_{(t)}$: analyses $\mathbf{x}_{(t)}$, $\mathbf{h}_{(t-1)}$.
- The three other layers are GATE CONTROLLERS (using logistic — 1 means open, 0 closed).
  - The FORGET GATE (controlled by $\mathbf{f}_{(t)}$) controls which parts of the long-term state should be erased.
  - The INPUT GATE (controlled by $\mathbf{i}_{(t)}$) controls which parts of $\mathbf{g}_{(t)}$ should be added to the long-term state.
  - The OUTPUT GATE (controlled by $\mathbf{o}_{(t)}$) controls which parts of the long-term state should be read and output at this time step (both to $\mathbf{h}_{(t)}$) and $\mathbf{y}_{(t)}$.

Long Short-Term Memory (LSTM) (2)

Equations

$$\mathbf{i}_{(t)} = \sigma\left(\mathbf{W}_{xi}^{T} \cdot \mathbf{x}_{(t)} + \mathbf{W}_{hi}^{T} \cdot \mathbf{h}_{(t-1)} + \mathbf{b}_{i}\right) \tag{3}$$

$$\mathbf{f}_{(t)} = \sigma\left(\mathbf{W}_{xf}^{T} \cdot \mathbf{x}_{(t)} + \mathbf{W}_{hf}^{T} \cdot \mathbf{h}_{(t-1)} + \mathbf{b}_{f}\right) \tag{4}$$

$$\mathbf{o}_{(t)} = \sigma\left(\mathbf{W}_{oi}^{T} \cdot \mathbf{x}_{(t)} + \mathbf{W}_{oi}^{T} \cdot \mathbf{h}_{(t-1)} + \mathbf{b}_{o}\right) \tag{5}$$

$$\mathbf{g}_{(t)} = \tanh\left(\mathbf{W}_{xg}^{T} \cdot \mathbf{x}_{(t)} + \mathbf{W}_{hg}^{T} \cdot \mathbf{h}_{(t-1)} + \mathbf{b}_{g}\right) \tag{6}$$

$$\mathbf{c}_{(t)} = \mathbf{f}_{(t)} \otimes \mathbf{c}_{(t-1)} + \mathbf{i}_{(t)} \otimes \mathbf{g}_{(t)} + \tag{7}$$

$$\mathbf{y}_{(t)} = \mathbf{h}_{(t)} = \mathbf{o}_{(t)} \otimes \tanh(\mathbf{c}_{(t)}) \tag{8}$$

Using an LSTM

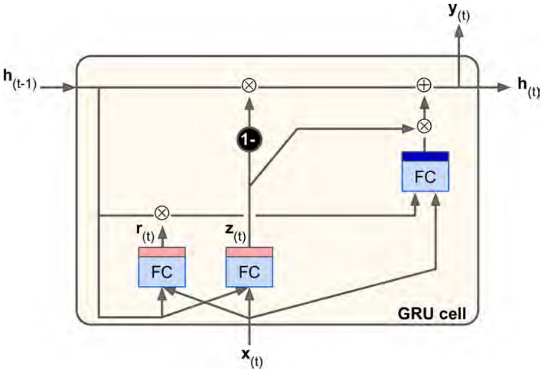
Model: Alternative #1

```
model = keras.models.Sequential([
    keras.layers.LSTM(20, return_sequences=True, input_shape=[None, 1]),
    keras.layers.LSTM(20, return_sequences=True),
    keras.layers.TimeDistributed(keras.layers.Dense(10))
])
```

Model: Alternative #2

```
model = keras.models.Sequential([
    keras.layers.RNN(keras.layers.LSTMCell(20), return_sequences=True,
                     input_shape=[None, 1]),
    keras.layers.RNN(keras.layers.LSTMCell(20), return_sequences=True),
    keras.layers.TimeDistributed(keras.layers.Dense(10))
])
```

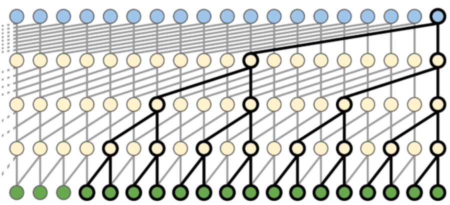
Gated Recurrent Unit (GRU)



Using a GRU

... keras.layers.GRU ...

For Audio (Originally): WaveNet



Model

```
model = keras.models.Sequential()
model.add(keras.layers.InputLayer(input_shape=[None, 1]))
for rate in (1, 2, 4, 8) * 2:
    model.add(keras.layers.Conv1D(filters=20, kernel_size=2, padding="causal",
                                  activation="relu", dilation_rate=rate))
model.add(keras.layers.Conv1D(filters=10, kernel_size=1))
model.compile(loss="mse", optimizer="adam", metrics=[last_time_step_mse])
history = model.fit(X_train, Y_train, epochs=20,
                    validation_data=(X_valid, Y_valid))
```

Outline

Outline

Turing Test

Imitation Game

- Test for characterising intelligence.
- Determined on the basis of whether a human can distinguish a human from a computer via a conversation.
- Current version: Loebner Prize.
- Requires generating text.

Using RNNs for Generating Text

- Character-level RNN: trained to predict the next character in a sentence.
- Options:
  - *Stateless*: learns on random portions of text at each iteration, without any information on the rest of the text.
  - *Stateful*: preserves the hidden state between training iterations and continues reading where it left off, allowing it to learn longer patterns.

Generating Text

Andrej Karpathy's Shakespearean Text

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.

A Character-Level RNN Generator (1)

Data

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

Tokenising Text

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

# maps to token ID, starting at 0
#>>> tokenizer.texts_to_sequences(["First"])
#[[20, 6, 9, 8, 3]]
#>>> tokenizer.sequences_to_texts([[20, 6, 9, 8, 3]])
#['f i r s t']
#>>> max_id = len(tokenizer.word_index) # number of distinct characters
#>>> dataset_size = tokenizer.document_count # total number of characters
```

A Character-Level RNN Generator (2)

Splitting a Sequential Dataset

- Data overlaps: training could overlap with test.
- Matters a lot for time series.
  - Typically split chronologically (e.g. train: years 2000–2012; dev: 2013–2015; test: 2016–2018).
- For us here, straightforward.

Creating Training Data

```
train_size = dataset_size * 90 // 100
dataset = tf.data.Dataset.from_tensor_slices(encoded[:train_size])
# a single sequence of >1m chars
```

A Character-Level RNN Generator (3)

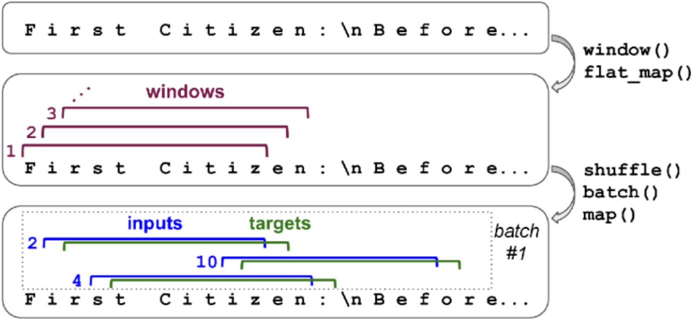
Splitting Training Data

```
n_steps = 100
window_length = n_steps + 1 # target = input shifted 1 character ahead
dataset = dataset.window(window_length, shift=1, drop_remainder=True)
  #<- truncated backprop
dataset = dataset.flat_map(lambda window: window.batch(window_length))
  # change dataset from nested ("list of lists") to flat (of tensors)
```

Other Preparation

```
batch_size = 32
dataset = dataset.shuffle(10000).batch(batch_size)
dataset = dataset.map(lambda windows: (windows[:, :-1], windows[:,1:]))
  # separate input from output
dataset = dataset.map(
    lambda X_batch, Y_batch: (tf.one_hot(X_batch, depth=max_id), Y_batch))
  # encode chars as one-hot vectors
dataset = dataset.prefetch(1)
  # see https://keras.io/getting_started/intro_to_keras_for_engineers/
```

A Character-Level RNN Generator (4)



A Character-Level RNN Generator (5)

Building and Training the Model

```
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)
```

Using the Model

```
def preprocess(texts):
    # transforming to the same representation as for training
    X = np.array(tokenizer.texts_to_sequences(texts)) - 1
    return tf.one_hot(X, max_id)

>>> X_new = preprocess(["How are yo"])
>>> Y_pred = model.predict_classes(X_new)
>>> tokenizer.sequences_to_texts(Y_pred + 1)[0][-1] # 1st sentence, last char
'u'
```

A Character-Level RNN Generator (6)

Generating Shakespearean Text

- Could give chunk of text, predict next letter.
  - Tends to repeat words.
- Instead, generate next letter probabilistically.
  - In proportion to class log probabilities.
  - Modify by TEMPERATURE: close to 0 favours high-probability characters, high value more uniform.

Functions for Producing Texts

```
def next_char(text, temperature=1):
    X_new = preprocess([text])
    y_proba = model.predict(X_new)[0, -1:, :]
    rescaled_logits = tf.math.log(y_proba) / temperature
    char_id = tf.random.categorical(rescaled_logits, num_samples=1) + 1
    return tokenizer.sequences_to_texts(char_id.numpy())[0]

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

A Character-Level RNN Generator (7)

Examples

```
>>> print(complete_text("t", temperature=0.2))
the belly the great and who shall be the belly the
>>> print(complete_text("w", temperature=1))
thing? or why you gremio.
who make which the first
>>> print(complete_text("w", temperature=2))
th no cce:
yeolg-hormer firi. a play asks.
fol rusb
```

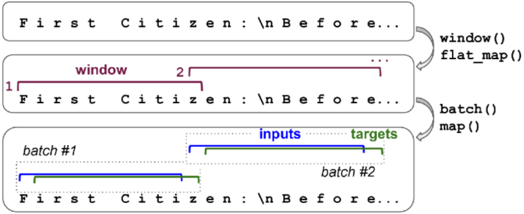
Stateful RNNs

Motivation

- In stateless RNNs, hidden states start at all zeroes, get trained across timesteps, then thrown away.
- Instead, want to keep final states across training batches.

Complications

- Need longer windows to guarantee no overlap of training input.
- Batching consequently harder.



Outline

IDMB Dataset (1)

The Dataset

[positive] A wonderful little production. The filming technique is very unassuming- very old-time-BBC fashion and gives a comforting, and sometimes discomforting, sense of realism to the entire piece. ...
[negative] Encouraged by the positive comments about this film on here I was looking forward to watching this film. Bad mistake. I've seen 950+ films and this is truly one of the worst of them - it's awful in almost every way: ...

imdb.load_data()

```
>>> (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]

>>> 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'
```

IDMB Dataset (2)

Alternative: TensorFlow Datasets

```
import tensorflow_datasets as tfds

datasets, info = tfds.load("imdb_reviews", as_supervised=True, with_info=True)
train_size = info.splits["train"].num_examples
```

Preprocessing

```
def preprocess(X_batch, y_batch):   # <- using only TF functions
    X_batch = tf.strings.substr(X_batch, 0, 300)
    X_batch = tf.strings.regex_replace(X_batch, b"<br\\s*/?>", b" ")
    X_batch = tf.strings.regex_replace(X_batch, b"[^a-zA-Z']", b" ")
    X_batch = tf.strings.split(X_batch)
    return X_batch.to_tensor(default_value=b"<pad>"), y_batch
```

IDMB Dataset (3)

Constructing the Vocabulary

```
from collections import Counter
vocabulary = Counter()
for X_batch, y_batch in datasets["train"].batch(32).map(preprocess):
    for review in X_batch:
        vocabulary.update(list(review.numpy()))

#>>> vocabulary.most_common()[:3]
# [(b'<pad>', 215797), (b'the', 61137), (b'a', 38564)]
```

Truncating the Vocab and Constructing Lookup Table

```
vocab_size = 10000
truncated_vocabulary = [
    word for word, count in vocabulary.most_common()[:vocab_size]]

words = tf.constant(truncated_vocabulary)
word_ids = tf.range(len(truncated_vocabulary), dtype=tf.int64)
vocab_init = tf.lookup.KeyValueTensorInitializer(words, word_ids)
num_oov_buckets = 1000
table = tf.lookup.StaticVocabularyTable(vocab_init, num_oov_buckets)

#>>> table.lookup(tf.constant([b"This movie was faaaaaantastic".split()]))
#<tf.Tensor: [...], dtype=int64, numpy=array([[   22,    12,    11, 10054]])>
```

A Sentiment Prediction Model

Constructing the Training Set

```
def encode_words(X_batch, y_batch):
    return table.lookup(X_batch), y_batch

train_set = datasets["train"].batch(32).map(preprocess)
train_set = train_set.map(encode_words).prefetch(1)
```

Building the Model

```
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)
```

Note

Can use MASKING to ignore padding tokens.

Pretrained Embeddings

Using TensorFlow Hub

```
import tensorflow_hub as hub

model = keras.Sequential([
    hub.KerasLayer("https://tfhub.dev/google/tf2-preview/nnlm-en-dim50/1",
                   dtype=tf.string, input_shape=[], output_shape=[50]),
    keras.layers.Dense(128, activation="relu"),
    keras.layers.Dense(1, activation="sigmoid")
])
model.compile(loss="binary_crossentropy", optimizer="adam",
              metrics=["accuracy"])
```

Training the Model

```
datasets, info = tfds.load("imdb_reviews", as_supervised=True, with_info=True)
train_size = info.splits["train"].num_examples
batch_size = 32
train_set = datasets["train"].batch(batch_size).prefetch(1)
history = model.fit(train_set, epochs=5)
```