Initialization

In [21]:
%load_ext autoreload
%autoreload 2
%matplotlib inline  

The autoreload extension is already loaded. To reload it, use:
%reload_ext autoreload

In [22]:
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

In [23]:
import tensorflow as tf
from tensorflow.keras.utils import get_file
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import GRU
from tensorflow.keras.optimizers import RMSprop
from tensorflow.keras.callbacks import Callback

Acknowledgement

- The code comes from: A. Géron: *Hands-On Machine Learning with Scikit-Learn, Keras and TensorFlow (2nd edn)*, O'Reilly, 2019

Warning

- The code takes a little time to run
- It is not important to understand this code in any case
Natural Language Processing

- In the previous lecture, we looked at a task (sentiment analysis) that requires natural language understanding. We tried various approaches including word embeddings and recurrent neural networks.
- In this lecture, we look at natural language generation — producing language.
- Specifically,
  - we build a model from training data that can predict the next character in a sentence.
  - we then use that model, with a bit of randomization, to produce new sentences in the style of the original training data.
- We do this at the level of individual characters, but it can be done at a higher-level, e.g. predicting/generating the next word.
- This might seem frivolous, but it gives insight into a number of useful systems that we will mention at the end of the lecture.

Char-RNN

- Everyone does this on Shakespeare — perhaps because if it outputs bad Shakespeare some people still think it sounds like Shakespeare!

Preprocessing the training data

- Most of the effort goes into preprocessing the dataset. Don't get bogged down in the details of this code.
- We're one-hot encoding the characters.
- We're making overlapping windows, shuffling these, and putting them into batches.

```
In [24]:
    shakespeare_url = "https://raw.githubusercontent.com/karpathy/char-rnn/master/data/tinyshakespeare/input.txt"
    filepath = get_file("shakespeare.txt", shakespeare_url)
    with open(filepath) as f:
        shakespeare_text = f.read()

In [25]:
    # Show you the first part of it
    shakespeare_text[:148]

Out[25]:
    'First Citizen:
    Before we proceed any further, hear me speak.\n    All:
    Speak, speak.
    First Citizen:
    You are all resolved rather to die than to famish?\n
In [26]:
    # Show you its distinct characters
    ".join(sorted(set(shakespeare_text.lower())))

Out[26]:
    '\n  !$&',.-3;?:abcdefghijklmnopqrstuvwxyz"'

In [27]:
    # Create and fit a character-level (rather than word-level) tokenizer
    # In effect, it lowercases and assigns ids to characters from 1 to 39 inc, e.g. ' ' is 1, 'e' is 2, etc.
    tokenizer = Tokenizer(char_level=True)
    tokenizer.fit_on_texts(shakespeare_text)
```
In [28]:
# Show you an encoding
tokenizer.texts_to_sequences(["speak"])
Out[28]:
[[8, 23, 2, 5, 25]]

In [29]:
# Show you the reverse
tokenizer.sequences_to_texts([[8, 23, 2, 5, 25]])
Out[29]:
['s p e a k']

In [30]:
max_id = len(tokenizer.word_index)  # number of distinct characters
dataset_size = tokenizer.document_count  # total number of characters

In [31]:
# Tokenize the full text (subtract 1 to use ids from 0 to 38 instead of 1 to 39, so now ' ' is 0, 'e' is 1, etc.)
encoded = np.array(tokenizer.texts_to_sequences([shakespeare_text])) - 1

In [32]:
# We'll train on the whole dataset
dataset = tf.data.Dataset.from_tensor_slices(encoded)

In [33]:
# In general, we can't train on the whole training set in one go: too long
# window() splits this into smaller windows of text
# Using shift=1 means the first window is characters 0 to 100, the second is characters 1 to 101, etc.
# Using drop_remainder=True means all windows are 101 characters long without needing us to pad the
# last ones (they are dropped)
# But window() produces a nested dataset: a dataset containing windows (each of which is a dataset)
# so we flatten it
n_steps = 100
window_length = n_steps + 1  # to include the target
dataset = dataset.window(window_length, shift=1, drop_remainder=True)
dataset = dataset.flat_map(lambda window: window.batch(window_length))

In [34]:
# Shuffle the windows and put into batches
batch_size = 32
dataset = dataset.shuffle(10000).batch(batch_size)

In [35]:
# Separate the inputs (the first 100 characters) from the targets (the last, i.e. 101st, character)
dataset = dataset.map(lambda windows: (windows[:, :-1], windows[:, 1:]))

In [36]:
# At the moment, characters are replaced by ids. Now we one-hot encode them.
# (This is OK for Char-RNN. If we were doing something with words, we might use word embeddings)
dataset = dataset.map(lambda X_batch, Y_batch: (tf.one_hot(X_batch, depth=max_id), Y_batch))

In [37]:
# Prefetching means while training on one batch, the next is being prepared
dataset = dataset.prefetch(1)

In [38]:
# Show you the shape of the first batch
for X_batch, Y_batch in dataset.take(1):
    print(X_batch.shape, Y_batch.shape)
(32, 100, 39) (32, 100)
The Char-RNN Model

- The input shape is \([\text{None, max_id}]\) because of the one-hot encoding
- We'll use a couple of GRU layers with dropout on their inputs and their hidden state
- The output layer has \(\text{max_id}\) neurons, because we're predicting that number of distinct characters, i.e. we have \(\text{max_id}\) classes
- We're predicting only one character at a time, so we have \(\text{return_sequences} = \text{False}\) (default) in the output layer

```
In [39]:
def build_char_rnn():
    network = Sequential()
    network.add(GRU(128, return_sequences=True, activation="tanh",
                      dropout=0.2, recurrent_dropout=0.2, input_shape=[None, m
                      ax_id])))
    network.add(GRU(128, return_sequences=True, activation="tanh",
                      dropout=0.2, recurrent_dropout=0.2))
    network.add(Dense(max_id, activation="softmax"))
    network.compile(optimizer=RMSprop(lr=0.0003), loss="sparse_categorical_c
    rossentropy")
    return network

In [40]: char_rnn = build_char_rnn()

In [41]: # Takes a long time
   history = char_rnn.fit(dataset,
                      steps_per_epoch=dataset_size // batch_size // n_step
                      epochs=10, verbose=0)
```

Predictions using Char-RNN

- Given some text (suitably preprocessed), the model can predict the next character

```
In [42]: # A function to preprocess the text whose next character we will predict: to
   #kenize and one-hot encode
   def preprocess(text):
       X = np.array(tokenizer.texts_to_sequences(text)) - 1
       return tf.one_hot(X, max_id)

In [43]: input_text = preprocess(["How are yo"])
   prediction = char_rnn.predict_classes(input_text)
   tokenizer.sequences_to_texts(prediction + 1)[0][-1] # 1st sentence, last cha
   r

Out[43]: 'u'
```
Generating Text using Char-RNN

- To generate text, we want to make repeated predictions:
  - Feed in some initial input
  - Predict the most likely next character
  - Add the prediction to the end of the input text
  - Feed in the extended input
  - Predict the most likely next character
  - and so on
- But this leads to repetitive text
- Instead, we make it stochastic:
  - We pick the next character randomly but based on the probabilities that the network produces

```python
# The temperature parameter allows you to tune it:
# - a value close to zero favours high probability characters, but leads to
#   more repetition
# - a high value gives all characters an almost equal probability
def next_char(model, text, temperature=1):
    X = preprocess([text])
    y_proba = model.predict(X)[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 generate_text(model, text, n_chars=50, temperature=1):
    for _ in range(n_chars):
        text += next_char(model, text, temperature)
    return text
```

```python
In [44]:
```

```python
# Some examples
print("Low temperature: ", generate_text(char_rnn, "t", temperature=0.2), '
')

print("Medium temperature: ", generate_text(char_rnn, "t", temperature=1), '
')

print("High temperature: ", generate_text(char_rnn, "t", temperature=2), '
')
```

Low temperature:  t the servingman:
what i have be the ware the were

Medium temperature:  tian. metsem.
brutus:
bouan sakmmes, ene in not o

High temperature:  t notg.
secun.
bousnsrou zavtli in homvhl wekm tu
How can we make the generated text more convincing?
- Tweak everything! More layers, more neurons per layer, more epochs, …
- You could make the windows bigger by increasing \( n_{\text{steps}} \) but even LSTM and GRUs, while better than SimpleRNNs, cannot handle very long sequences
- We could change Char-RNN from being a \textbf{stateless RNN} to being a \textbf{stateful RNN}

\textbf{Stateless RNNs and Stateful RNNs}

- \textbf{Stateless RNN}: In a training iteration,
  - will be trained on a batch of random chunks of the text
  - hidden state starts at all zeros
  - processes the input, step by step
  - after the last timestep, throws away the hidden state
- \textbf{Stateful RNN}:
  - preserve the hidden state at the end of the last timestep
  - use it as the initial hidden state for the next batch
  This way, we can learn longer patterns despite only back-propagating through short sequences
- However, we now must arrange our batches quite carefully
  - Each input sequence in a batch starts where the corresponding sequence in the previous batch finished
  - In other words, we must remove the overlapping and the shuffling that we used in the stateless RNN
- Keras comes with a parameter for its recurrent layers, \texttt{stateful=True}

\textbf{Applications}

- \textit{Sunspring} is a sci-fi movie whose script was generated by an LSTM trained on existing movie scripts: http://www.thereforefilms.com/sunspring.html
- We can generate music in this way too, e.g. https://folkrnn.org/
- There are researchers who are trying to generate explanations using these techniques
- But let’s look briefly at image captioning, machine translation and question-answering
Image Captioning

- Google have learned a model for automatically captioning images
- It is trained on a dataset of images plus human captions
- The neural network combines a convolutional network to find features in the image (using the Inception V3 model) but also a recurrent neural network that predicts words, similar to what we have just studied

Google's image captioning system

- There are many details, which we will ignore
  - Handling training sentences of various lengths (we can pad but we can't crop; common instead is for batches to contain sentences of the same length)
  - After the network predicts the end-of-sentence token, any further outputs should be ignored and not contribute to the loss
  - This is multi-class classification for an enormous number of classes (one per English word); even computing softmax will be slow; we can speed this up during training (not afterwards) with sampled softmax, so that the loss is only approximated
- Take a look at this Twitter thread about sheep ("https://twitter.com/JanelleCShane/status/969239712190746624?s=19")
Machine Translation

- Datasets of parallel text are often available, e.g. proceedings of the Canadian parliament are published in both English and French. These give our training set
  - The words may be assigned ids and these may be one-hot encoded
  - Or we might include a word embedding layer (not shown below)

During training, at each timestep, we feed a word of the English sentence into the encoder and the previous word of the French sentence into the decoder. The decoder is predicting the next word. If its prediction is incorrect, weights will get changed

After training, at each timestep, we feed a word of the English sentence that is to be translated into the encoder and the previous predicted French word into the decoder

There are many details, many of them are the same as the ones we described for image captioning. But new details too, e.g.

- We feed in the English sentence in reverse

There are also many new ideas, which we do not have time to cover. Look them up if interested, e.g.:
  - Bidirectional RNNs;
  - Using beam search for a less greedy solution
  - Attention mechanisms
    - These have had a huge effect
    - Instead of feeding just the last encoder output into the decoder, we feed in all outputs
    - The attention mechanism is, e.g., a weighted sum of these encoder outputs
    - These weights determine which encoder output (and hence, in some sense, which word of the English input) it will focus on at this step
  - How are these weights learned? The attention mechanism has a small neural network called an attention layer (not yet available in Keras), which learns them

Attention mechanisms have been added to image captioning systems and there are even neural networks (such as the Transformer) which use attention layers in place of recurrent layers and convolutional layers.

Google's Machine Translation system

Question-Answering

- AI has long had the goal of producing a question-answering system, especially one that can hold extended conversations — it’s central to the Turing Test, for example
- Traditional chatbots can just about function in very narrow domains but they fail as soon as the conversation becomes more general
- We are now seeing neural networks for doing this — quite similar to the ones for Machine Translation: an encoder and a decoder, trained on, e.g., social media conversations

Conclusions

- There has been huge progress in NLP in the last couple of years
- In particular, using a variety of architectures, including Transformers, there has been a lot of work on producing pretrained models that you can use as layers in your own architecture
- One famous example is Google's BERT: https://github.com/google-research/bert (https://github.com/google-research/bert)
  - Google claim that this has much improved their search engine's ability to answer questions (as opposed to traditional keyword search): https://www.blog.google/products/search/search-language-understanding-bert/ (https://www.blog.google/products/search/search-language-understanding-bert/)
- Of course, the big question, which we'll return to in the last lecture of the module is: do these neural networks actually understand anything?