Programming Assignment

Language model for the Shakespeare dataset

Instructions

In this notebook, you will use the text preprocessing tools and RNN models to build a character-level language model. You will then train your model on the works of Shakespeare, and use the network to generate your own text..

Some code cells are provided you in the notebook. You should avoid editing provided code, and make sure to execute the cells in order to avoid unexpected errors. Some cells begin with the line:

`#### GRADED CELL ####`

Don't move or edit this first line - this is what the automatic grader looks for to recognise graded cells. These cells require you to write your own code to complete them, and are automatically graded when you submit the notebook. Don't edit the function name or signature provided in these cells, otherwise the automatic grader might not function properly. Inside these graded cells, you can use any functions or classes that are imported below, but make sure you don't use any variables that are outside the scope of the function.

How to submit

Complete all the tasks you are asked for in the worksheet. When you have finished and are happy with your code, press the **Submit Assignment** button at the top of this notebook.

Let's get started!

We'll start running some imports, and loading the dataset. Do not edit the existing imports in the following cell. If you would like to make further Tensorflow imports, you should add them here.

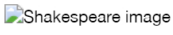
In [1]:
```
#### PACKAGE IMPORTS ####

# Run this cell first to import all required packages. Do not make any imports elsewhere in the notebook

import tensorflow as tf
import numpy as np
import json
import matplotlib.pyplot as plt
%matplotlib inline

# If you would like to make further imports from tensorflow, add them here

from tensorflow.keras.preprocessing.text import Tokenizer
```



The Shakespeare dataset

In this assignment, you will use a subset of the Shakespeare dataset. It consists of a single text file with several excerpts concatenated together. The data is in raw text form, and so far has not yet had any preprocessing.

Your goal is to construct an unsupervised character-level sequence model that can generate text according to a distribution learned from the dataset.

Load and inspect the dataset

In [2]:
```
# Load the text file into a string

with open('data/Shakespeare.txt', 'r', encoding='utf-8') as file:
    text = file.read()
```

In [3]:
```
# Create a list of chunks of text

text_chunks = text.split('.')
```

To give you a feel for what the text looks like, we will print a few chunks from the list.

In [4]:
```
# Display some randomly selected text samples

num_samples = 5
inx = np.random.choice(len(text_chunks), num_samples, replace=False)
for chunk in np.array(text_chunks)[inx]:
    print(chunk)
```

```
Shepherd:
None, sir; I have no pheasant, cock nor hen
 This jealousy
Is for a precious creature: as she's rare,
```

```
Must it be great, and as his person's mighty,
Must it be violent, and as he does conceive
He is dishonour'd by a man which ever
Profess'd to him, why, his revenges must
In that be made more bitter


FLORIZEL:
O, that must be
I' the virtue of your daughter: one being dead,
I shall have more than you can dream of yet;
Enough then for your wonder
 He says he loves my daughter:
I think so too; for never gazed the moon
Upon the water as he'll stand and read
As 'twere my daughter's eyes: and, to be plain


Third Servingman:
Pray you, poor gentleman, take up some other
station; here's no place for you; pray you, avoid: come
```

Create a character-level tokenizer

You should now write a function that returns a `Tokenizer` object. The function takes a list of strings as an argument, and should create a `Tokenizer` according to the following specification:

- The number of tokens should be unlimited (there should be as many as required by the dataset).
- Tokens should be created at the character level (not at the word level, which is the default behaviour).
- No characters should be filtered out or ignored.
- The original capitalization should be retained (do not convert the text to lower case)

The `Tokenizer` should be fit to the `list_of_strings` argument and returned by the function.

Hint: you may need to refer to the documentation for the `Tokenizer`.

In [5]:
```
#### GRADED CELL ####

# Complete the following function.
# Make sure not to change the function name or arguments.

def create_character_tokenizer(list_of_strings):
    """
    This function takes a list of strings as its argument. It should create
    and return a Tokenizer according to the above specifications.
    """

```

In [6]:
```
# Get the tokenizer

tokenizer = create_character_tokenizer(text_chunks)
print(type(tokenizer))
```

```
<class 'keras_preprocessing.text.Tokenizer'>
```

Tokenize the text

You should now write a function to use the tokenizer to map each string in `text_chunks` to its corresponding encoded sequence. The following function takes a fitted `Tokenizer` object in the first argument (as returned by `create_character_tokenizer`) and a list of strings in the second argument. The function should return a list of lists, where each sublist is a sequence of integer tokens encoding the text sequences according to the mapping stored in the tokenizer.

Hint: you may need to refer to the documentation for the `Tokenizer`.

In [7]:
```
#### GRADED CELL ####

# Complete the following function.
# Make sure not to change the function name or arguments.

def strings_to_sequences(tokenizer, list_of_strings):
    """
    This function takes a tokenizer object and a list of strings as its arguments.
    It should use the tokenizer to map the text chunks to sequences of tokens and
    then return this list of encoded sequences.
    """

```

In [8]:
```
# Encode the text chunks into tokens

seq_chunks = strings_to_sequences(tokenizer, text_chunks)
seq_chunks[0]
```

Out[8]:
```
[50,
 11,
 9,
 8,
 4,
 2,
 38,
 11,
 4,
 11,
 58,
 3,
```

```
10,
26,
12,
44,
3,
20,
5,
9,
3,
2,
19,
3,
2,
25,
9,
5,
21,
3,
3,
14,
2,
6,
10,
17,
2,
20,
15,
9,
4,
7,
3,
9,
18,
2,
7,
3,
6,
9,
2,
16,
3,
2,
8,
25,
3,
6,
29]
```

Pad the encoded sequences and store them in a numpy array

Since not all of the text chunks are the same length, you will need to pad them in order to train on batches. You should now complete the following function, which takes the list of lists of tokens, and creates a single numpy array with the token sequences in the rows, according to the following specification:

- The longest allowed sequence should be 500 tokens. Any sequence that is longer should be shortened by truncating the beginning of the sequence.
- Use zeros for padding the sequences. The zero padding should be placed before the sequences as required.

The function should then return the resulting numpy array.

Hint: you may want to refer to the documentation for the `pad_sequences` function.

In [9]:
```
#### GRADED CELL ####

# Complete the following function.
# Make sure not to change the function name or arguments.

def make_padded_dataset(sequence_chunks):
    """
    This function takes a list of lists of tokenized sequences, and transforms
    them into a 2D numpy array, padding the sequences as necessary according to
    the above specification. The function should then return the numpy array.
    """

```

In [10]:
```
# Pad the token sequence chunks and get the numpy array

padded_sequences = make_padded_dataset(seq_chunks)
```

In [11]:
```
len(padded_sequences)
```

Out[11]:
```
7886
```

Create model inputs and targets

Now you are ready to build your RNN model. The model will receive a sequence of characters and predict the next character in the sequence. At training time, the model can be passed an input sequence, with the target sequence is shifted by one.

For example, the expression `To be or not to be` appears in Shakespeare's play 'Hamlet'. Given input `To be or not to b`, the correct prediction is `o be or not to be`. Notice that the prediction is the same length as the input!

You should now write the following function to create an input and target array from the current `padded_sequences` array. The function has a single argument that is a 2D numpy array of shape `(num_examples, max_seq_len)`. It should fulfil the following specification:

- The function should return an input array and an output array, both of size `(num_examples, max_seq_len - 1)`.

- The input array should contain the first `max_seq_len - 1` tokens of each sequence.
- The output array should contain the last `max_seq_len - 1` tokens of each sequence.

The function should then return the tuple `(input_array, output_array)`. Note that it is possible to complete this function using numpy indexing alone!

In [12]:
```
#### GRADED CELL ####

# Complete the following function.
# Make sure not to change the function name or arguments.

def create_inputs_and_targets(array_of_sequences):
    """
    This function takes a 2D numpy array of token sequences, and returns a tuple of two
    elements: the first element is the input array and the second element is the output
    array, which are defined according to the above specification.
    """

```

In [13]:
```
# Create the input and output arrays

input_seq, target_seq = create_inputs_and_targets(padded_sequences)
```

In [14]:
```
input_seq.shape, target_seq.shape
```

Out[14]: ((7886, 499), (7886, 499))

Preprocess sequence array for stateful RNN

We will build our RNN language model to be stateful, so that the internal state of the RNN will be maintained across batches. For this to be effective, we need to make sure that each element of every batch follows on from the corresponding element of the preceding batch (you may want to look back at the "Stateful RNNs" reading notebook earlier in the week).

The following code processes the input and output sequence arrays so that they are ready to be split into batches for training a stateful RNN, by re-ordering the sequence examples (the rows) according to a specified batch size.

In [15]:
```
# Fix the batch size for training

batch_size = 32
```

In [16]:
```
# Prepare input and output arrays for training the stateful RNN

num_examples = input_seq.shape[0]

num_processed_examples = num_examples - (num_examples % batch_size)

input_seq = input_seq[:num_processed_examples]
target_seq = target_seq[:num_processed_examples]

steps = int(num_processed_examples / 32)  # steps per epoch

inx = np.empty((0,), dtype=np.int32)
for i in range(steps):
    inx = np.concatenate((inx, i + np.arange(0, num_processed_examples, steps)))

input_seq_stateful = input_seq[inx]
target_seq_stateful = target_seq[inx]
```

Split the data into training and validation sets

We will set aside approximately 20% of the data for validation.

In [17]:
```
# Create the training and validation splits

num_train_examples = int(batch_size * ((0.8 * num_processed_examples) // batch_size))

input_train = input_seq_stateful[:num_train_examples]
target_train = target_seq_stateful[:num_train_examples]

input_valid = input_seq_stateful[num_train_examples:]
target_valid = target_seq_stateful[num_train_examples:]
```

Create training and validation Dataset objects

You should now write a function to take the training and validation input and target arrays, and create training and validation `tf.data.Dataset` objects. The function takes an input array and target array in the first two arguments, and the batch size in the third argument. Your function should do the following:

- Create a `Dataset` using the `from_tensor_slices` static method, passing in a tuple of the input and output numpy arrays.
- Batch the `Dataset` using the `batch_size` argument, setting `drop_remainder` to `True`.

The function should then return the `Dataset` object.

In [18]:
```
#### GRADED CELL ####

# Complete the following function.
# Make sure not to change the function name or arguments.

def make_Dataset(input_array, target_array, batch_size):
    """
    This function takes two 2D numpy arrays in the first two arguments, and an integer
    batch_size in the third argument. It should create and return a Dataset object
    using the two numpy arrays and batch size according to the above specification.
    """
```

```
11
12
```

In [19]:
```
# Create the training and validation Datasets

train_data = make_Dataset(input_train, target_train, batch_size)
valid_data = make_Dataset(input_valid, target_valid, batch_size)
```

In [20]:
```
valid_data
```

Out[20]:
```
<BatchDataset shapes: ((32, 499), (32, 499)), types: (tf.int32, tf.int32)>
```

Build the recurrent neural network model

You are now ready to build your RNN character-level language model. You should write the following function to build the model; the function takes arguments for the batch size and vocabulary size (number of tokens). Using the Sequential API, your function should build your model according to the following specifications:

- The first layer should be an Embedding layer with an embedding dimension of 256 and set the vocabulary size to `vocab_size` from the function argument.
- The Embedding layer should also mask the zero padding in the input sequences.
- The Embedding layer should also set the `batch_input_shape` to `(batch_size, None)` (a fixed batch size is required for stateful RNNs).
- The next layer should be a (uni-directional) GRU layer with 1024 units, set to be a stateful RNN layer.
- The GRU layer should return the full sequence, instead of just the output state at the final time step.
- The final layer should be a Dense layer with `vocab_size` units and no activation function.

In total, the network should have 3 layers.

In [96]:
```
#### GRADED CELL ####

# Complete the following function.
# Make sure not to change the function name or arguments.

def get_model(vocab_size, batch_size):
    """
    This function takes a vocabulary size and batch size, and builds and returns a
    Sequential model according to the above specification.
    """

```

In [97]:
```
# Build the model and print the model summary

model = get_model(len(tokenizer.word_index) + 1, batch_size)
model.summary()
```

```
Model: "sequential_7"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_7 (Embedding)      (32, None, 256)           16896
_________________________________________________________________
gru_7 (GRU)                  (32, None, 1024)          3938304
_________________________________________________________________
dense_7 (Dense)              (32, None, 66)            67650
=================================================================
Total params: 4,022,850
Trainable params: 4,022,850
Non-trainable params: 0
_________________________________________________________________
```

Compile and train the model

You are now ready to compile and train the model. For this model and dataset, the training time is very long. Therefore for this assignment it is not a requirement to train the model. We have pre-trained a model for you (using the code below) and saved the model weights, which can be loaded to get the model predictions.

It is recommended to use accelerator hardware (e.g. using Colab) when training this model. It would also be beneficial to increase the size of the model, e.g. by stacking extra recurrent layers.

In [23]:
```
# Choose whether to train a new model or load the pre-trained model

skip_training = 1
```

In [24]:
```
# Compile and train the model, or load pre-trained weights

if not skip_training:
    checkpoint_callback=tf.keras.callbacks.ModelCheckpoint(filepath='./models/ckpt',
                                                           save_weights_only=True,
                                                           save_best_only=True)
    model.compile(optimizer='adam', loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
                  metrics=['sparse_categorical_accuracy'])
    history = model.fit(train_data, epochs=15, validation_data=valid_data,
                        validation_steps=50, callbacks=[checkpoint_callback])
```

In [25]:
```
# Save model history as a json file, or load it if using pre-trained weights

if not skip_training:
    history_dict = dict()
    for k, v in history.history.items():
        history_dict[k] = [float(val) for val in history.history[k]]
    with open('models/history.json', 'w+') as json_file:
        json.dump(history_dict, json_file, sort_keys=True, indent=4)
```

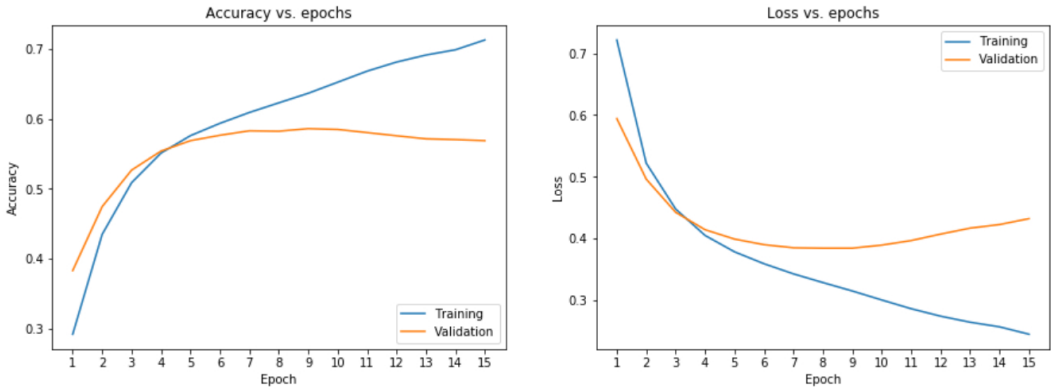
```
else:
    with open('models/history.json', 'r') as json_file:
        history_dict = json.load(json_file)
```

Plot the learning curves

In [26]:
```
# Run this cell to plot accuracy vs epoch and loss vs epoch

plt.figure(figsize=(15,5))
plt.subplot(121)
plt.plot(history_dict['sparse_categorical_accuracy'])
plt.plot(history_dict['val_sparse_categorical_accuracy'])
plt.title('Accuracy vs. epochs')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.xticks(np.arange(len(history_dict['sparse_categorical_accuracy'])))
ax = plt.gca()
ax.set_xticklabels(1 + np.arange(len(history_dict['sparse_categorical_accuracy'])))
plt.legend(['Training', 'Validation'], loc='lower right')

plt.subplot(122)
plt.plot(history_dict['loss'])
plt.plot(history_dict['val_loss'])
plt.title('Loss vs. epochs')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.xticks(np.arange(len(history_dict['sparse_categorical_accuracy'])))
ax = plt.gca()
ax.set_xticklabels(1 + np.arange(len(history_dict['sparse_categorical_accuracy'])))
plt.legend(['Training', 'Validation'], loc='upper right')
plt.show()
```



Write a text generation algorithm

You can now use the model to generate text! In order to generate a single text sequence, the model needs to be rebuilt with a batch size of 1.

In [98]:
```
# Re-build the model and load the saved weights

model = get_model(len(tokenizer.word_index), batch_size=1)
model.load_weights(tf.train.latest_checkpoint('./models/'))
```

Out[98]:
```
<tensorflow.python.training.tracking.util.CheckpointLoadStatus at 0x7f4cd4348278>
```

In [99]:
```
model.summary()
```

```
Model: "sequential_8"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_8 (Embedding)      (1, None, 256)            16640
_________________________________________________________________
gru_8 (GRU)                  (1, None, 1024)           3938304
_________________________________________________________________
dense_8 (Dense)              (1, None, 65)             66625
=================================================================
Total params: 4,021,569
Trainable params: 4,021,569
Non-trainable params: 0
_________________________________________________________________
```

An algorithm to generate text is as follows:

1. Specify a seed string (e.g. `'ROMEO:'`) to get the network started, and a define number of characters for the model to generate, `num_generation_steps`.
2. Tokenize this sentence to obtain a list containing one list of the integer tokens.
3. Reset the initial state of the network.
4. Convert the token list into a Tensor (or numpy array) and pass it to your model as a batch of size one.
5. Get the model prediction (logits) for the last time step and extract the state of the recurrent layer.
6. Use the logits to construct a categorical distribution and sample a token from it.
7. Repeat the following for `num_generation_steps - 1` steps:
   A. Use the saved state of the recurrent layer and the last sampled token to get new logit predictions
   B. Use the logits to construct a new categorical distribution and sample a token from it.
   C. Save the updated state of the recurrent layer.
8. Take the final list of tokens and convert to text using the Tokenizer

Note that the internal state of the recurrent layer can be accessed using the `states` property. For the GRU layer, it is a list of one variable:

In [86]:
```
# Inspect the model's current recurrent state

model.layers[1].states
```

Out[86]:
```
[<tf.Variable 'gru_4/Variable:0' shape=(1, 1024) dtype=float32, numpy=array([[0., 0., 0., ..., 0., 0., 0.]], dtype=float32)>]
```

We will break the algorithm down into two steps. First, you should now complete the following function that takes a sequence of tokens of any length and returns the model's prediction (the logits) for the last time step. The specification is as follows:

- The token sequence will be a python list, containing one list of integer tokens, e.g. `[[1, 2, 3, 4]]`
- The function should convert the list into a 2D Tensor or numpy array
- If the function argument `initial_state` is `None`, then the function should reset the state of the recurrent layer to zeros.
- Otherwise, if the function argument `initial_state` is a 2D Tensor or numpy array, assign the value of the internal state of the GRU layer to this argument.
- Get the model's prediction (logits) for the last time step only.

The function should then return the logits as a 2D numpy array, where the first dimension is equal to 1 (batch size).

Hint: the internal state of the recurrent can be reset to zeros using the `reset_states` method.

In [102]:
```
#### GRADED CELL ####

# Complete the following function.
# Make sure not to change the function name or arguments.

def get_logits(model, token_sequence, initial_state=None):
    """
    This function takes a model object, a token sequence and an optional initial
    state for the recurrent layer. The function should return the logits prediction
    for the final time step as a 2D numpy array.
    """

```

In [105]:
```
# Test the get_logits function by passing a dummy token sequence

dummy_initial_state = tf.random.normal(model.layers[1].states[0].shape)
get_logits(model, [[1, 2, 3, 4]], initial_state=dummy_initial_state)
```

Out[105]:
```
array([[-7.8635387 ,  2.5842235 ,  1.7061311 ,  1.1696024 ,  3.5771918 ,
        -0.29500112, -5.6794972 ,  2.1364007 ,  6.402848  ,  4.5042667 ,
         3.1246886 , -1.60197   ,  4.657864  ,  1.6309471 ,  6.3137264 ,
         6.8166666 , -1.9862776 ,  0.48749858,  2.9697273 , -0.6942751 ,
         2.919373  , -0.67325234, -9.16465   , -1.2702218 ,  3.4456475 ,
        -2.16475   , -6.4975514 , -0.46681565, -1.0272744 , -5.086094  ,
         2.11488   , -6.4433165 , -6.5558376 , -3.9154153 , -2.506845  ,
        -5.183504  , -7.3166847 , -6.5578966 , -2.9369526 , -4.613743  ,
        -7.218315  , -7.8147097 , -5.643158  , -3.2375395 , -5.3275547 ,
        -6.711257  , -2.1535444 , -6.589007  , -0.83005106, -7.8602414 ,
        -6.2945538 , -4.8834615 , -7.313702  , -8.558353  , -5.8940043 ,
        -1.8490337 , -1.1474149 , -1.5565965 , -7.1139627 , -6.993729  ,
        -3.8342485 , -3.4291258 , -7.113218  , -6.8122406 , -5.1638665 ]],
      dtype=float32)
```

You should now write a function that takes a logits prediction similar to the above, uses it to create a categorical distribution, and samples a token from this distribution. The following function takes a 2D numpy array `logits` as an argument, and should return a single integer prediction that is sampled from the categorical distribution.

Hint: you might find the `tf.random.categorical` function useful for this; see the documentation here.

In [111]:
```
#### GRADED CELL ####

# Complete the following function.
# Make sure not to change the function name or arguments.

def sample_token(logits):
    """
    This function takes a 2D numpy array as an input, and constructs a
    categorical distribution using it. It should then sample from this
    distribution and return the sample as a single integer.
    """

```

In [112]:
```
# Test the sample_token function by passing dummy logits

dummy_initial_state = tf.random.normal(model.layers[1].states[0].shape)
dummy_logits = get_logits(model, [[1, 2, 3, 4]], initial_state=dummy_initial_state)
sample_token(dummy_logits)
```

Out[112]:
```
8
```

In [79]:
```
logits_size = dummy_logits.shape[1]
dummy_logits = -np.inf*np.ones((1, logits_size))
dummy_logits[0, 20] = 0
sample_token(dummy_logits)
random_inx = np.random.choice(logits_size, 2, replace=False)
random_inx1, random_inx2 = random_inx[0], random_inx[1]
print(random_inx1, random_inx2)
dummy_logits = -np.inf*np.ones((1, logits_size))
dummy_logits[0, random_inx1] = 0
dummy_logits[0, random_inx2] = 0
sampled_token = []
```

```
for _ in range(100):
    sampled_token.append(sample_token(dummy_logits))

l_tokens, l_counts = np.unique(np.array(sampled_token), return_counts=True)
len(l_tokens) == 2
```

```
43 30
```

Out[79]:
```
True
```

Generate text from the model

You are now ready to generate text from the model!

In [113]:
```
# Create a seed string and number of generation steps

init_string = 'ROMEO:'
num_generation_steps = 1000
```

In [114]:
```
# Use the model to generate a token sequence

token_sequence = tokenizer.texts_to_sequences([init_string])
initial_state = None
input_sequence = token_sequence

for _ in range(num_generation_steps):
    logits = get_logits(model, input_sequence, initial_state=initial_state)
    sampled_token = sample_token(logits)
    token_sequence[0].append(sampled_token)
    input_sequence = [[sampled_token]]
    initial_state = model.layers[1].states[0].numpy()

print(tokenizer.sequences_to_texts(token_sequence)[0][::2])
```

```
ROMEO:pi;aoe<K>hom<K>mtdy<K>uossmB<K>g<K>wos y<K>ortea s<K>hodfayiRnssoay<K>hnsy<K>Nnfaosl<K>rte<K>o<K>wttv r<K>faoh BiHm<K>fds
hd <K>,n

<K>lt<K>fdshy<K>a s <K>nr<K>tds<K>Istea s<K>Fosvhanb<K>gwwnfnA iEw<K>ea <K>l ,nw <K>ea <K> osre<K>et<K>l Atea sikt<K>eov <K>orl
<K>uol <K>ou<K>wsdnewd
<K>lodcae sTh<K>a oseyi;anfa<K>a nr<K>orl<K>ea <K>r fvDIoI s<K>eaoe<K>ea <K>b srnfntdhyiMm<K>s A rc <K>u <K>htu <K>kanh<K>?odre
l<K>zd
nnri! etuol <K>;aoe<K>u orh<K> rtdcaTl<K>mtdy<K>twe<K>ea  y<K> qfdh liWbtr<K>eanh<K>,ts
l<K>ea <K>Vdhenf y<K>orl<K>Im<K>mtdyiHm<K>woea sTh<K>
nA m<K>hdbb s<K>wns hC<K>orl<K>uov <K>or<K>ossorf i:rtrG<K>aoA <K>mtd<K>eov <K>um<K>hesorc <K>kanh<K>tds<K>e,n
eiiL:k'RMFEpi:<K>ut, l<K>et<K>mtdyDDeaoe<K>h
 ,<K>u <K>
 oA <K>eaoe<K>,oh<K>um<K>utea sTh<K>atdh yi:rl<K>oconrhe<K> qf hhntrC<K>gT

<K>
 oA <K>mtd<K>u or<K>et<K>wnrlioca <K>nr<K>ctr y<K>eaoe<K>adhIorl<K>ea m<K>hao

<K>I <K>rt<K>henrc<K>ne<K>udheifor<K>uol <K>um<K>dr,om<K>,oh<K>e,nhenem<K>ea <K>ldv <K>,a s <K>neiI
 hh l<K>
ttvh<K>obb os
 y<K>tw<K>mtd<K>eadh<K>uorTh<K>a osTh<K>stmo
<K>osuyi:conrhe<K>ea <K>h sAnr <K>uor<K>Im<K>aorlTh<K>nrf rentriReorh<K>,anfa<K>cnA h<K>
nv <K>uorm<K>btnhtrpigw<K>bm h <K>ea <K>obb rdroentrikov <K>um<K>bo
of p<K>eatdca<K>
 rl<K>VtnrTl<K>ea <K>heorhyi;nha<K>fos wd
<K>jd hentrhB<K>g<K>
nA <K>u <K>et<K>h  yi
```

Congratulations for completing this programming assignment! In the next week of the course we will see how to build customised models and layers, and make custom training loops.