Lab 3

Sunday, April 14, 2024 5:27 PM

21K-3210

1.) The number of training epochs determines how well the model learns the patterns in the data. Increasing the number of epochs can improve performance for a while, but then the model may begin to overfit to the training data.

2.) Changes to the dataset can also have an impact on performance. Adding more diverse or representative data can help the model generalize to previously unseen examples. However, if the augmentation is overly aggressive or introduces noise, it may degrade performance.

3.) The choice of start string can influence the outcome, particularly for generative tasks in which the model generates text. Different start strings can result in different sequences, which may affect the coherence or relevance of the generated text.

4.) Effects of Using LSTM and GRU: LSTM (Long Short-Term Memory) and GRU (Gated Recurrent Unit) are two types of recurrent neural network (RNN) architectures used to detect long-term dependencies in sequential data. In general, LSTM and GRU perform similarly on many tasks, but their performance may differ depending on the dataset and task. LSTM is widely regarded as more powerful but also more computationally expensive than GRU.

Copyright Information

```
# Copyright 2024 MIT Introduction to Deep Learning. All Rights
Reserved.
#
# Licensed under the MIT License. You may not use this file except in
compliance
# with the License. Use and/or modification of this code outside of
MIT Introduction
# to Deep Learning must reference:
#
# © MIT Introduction to Deep Learning
# http://introtodeeplearning.com
#
```

Lab 1: Intro to TensorFlow and Music Generation with RNNs

Part 2: Music Generation with RNNs

In this portion of the lab, we will explore building a Recurrent Neural Network (RNN) for music generation. We will train a model to learn the patterns in raw sheet music in ABC notation and then use this model to generate new music.

2.1 Dependencies

First, let's download the course repository, install dependencies, and import the relevant packages we'll need for this lab.

We will be using Comet ML to track our model development and training runs. First, sign up for a Comet account at this link (you can use your Google or Github account). This will generate a personal API Key, which you can find either in the first 'Get Started with Comet' page, under your account settings, or by pressing the '?' in the top right corner and then 'Quickstart Guide'. Enter this API key as the global variable `COMET_API_KEY`.

```
!pip install comet_ml > /dev/null 2>&1
import comet_ml
# TODO: ENTER YOUR API KEY HERE!! instructions above
COMET_API_KEY = "s7tyqFmWIEbLu8ax9HkD7g7jb"

# Import Tensorflow 2.0
import tensorflow as tf
```

```
# Download and import the MIT Introduction to Deep Learning package
!pip install mitdeeplearning --quiet
import mitdeeplearning as mdl

# Import all remaining packages
import numpy as np
import os
import time
import functools
from IPython import display as ipythondisplay
from tqdm import tqdm
from scipy.io.wavfile import write
!apt-get install abcmidi timidity > /dev/null 2>&1


# Check that we are using a GPU, if not switch runtimes
#   using Runtime > Change Runtime Type > GPU
assert len(tf.config.list_physical_devices('GPU')) > 0
assert COMET_API_KEY != "", "Please insert your Comet API Key"
```

```
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 2.1/2.1 MB 9.1 MB/s eta
0:00:00
etadata (setup.py) ... itdeeplearning (setup.py) ...
```

2.2 Dataset

We've gathered a dataset of thousands of Irish folk songs, represented in the ABC notation. Let's download the dataset and inspect it:

```
# Download the dataset
songs = mdl.lab1.load_training_data()

# Print one of the songs to inspect it in greater detail!
example_song = songs[0]
print("\nExample song: ")
print(example_song)
```

```
Found 817 songs in text

Example song:
X:1
T:Alexander's
Z: id:dc-hornpipe-1
M:C|
L:1/8
K:D Major
(3ABc|dAFA DFAd|fdcd FAdf|gfge fefd|(3efe (3dcB A2 (3ABc|!
dAFA DFAd|fdcd FAdf|gfge fefd|(3efe dc d2:|!
AG|FAdA FAdA|GBdB GBdB|Acec Acec|dfaf gecA|!
FAdA FAdA|GBdB GBdB|Aceg fefd|(3efe dc d2:|!

/usr/local/lib/python3.10/dist-packages/ipykernel/ipkernel.py:283:
DeprecationWarning: `should_run_async` will not call `transform_cell`
automatically in the future. Please pass the result to
`transformed_cell` argument and any exception that happen during
thetransform in `preprocessing_exc_tuple` in IPython 7.17 and above.
  and should_run_async(code)
```

We can easily convert a song in ABC notation to an audio waveform and play it back. Be patient for this conversion to run, it can take some time.

```
# Convert the ABC notation to audio file and listen to it
mdl.lab1.play_song(example_song)

<IPython.lib.display.Audio object>
```

One important thing to think about is that this notation of music does not simply contain information on the notes being played, but additionally there is meta information such as the song title, key, and tempo. How does the number of different characters that are present in the text file impact the complexity of the learning problem? This will become important soon, when we generate a numerical representation for the text data.

```
# Join our list of song strings into a single string containing all
songs
songs_joined = "\n\n".join(songs)

# Find all unique characters in the joined string
vocab = sorted(set(songs_joined))
print("There are", len(vocab), "unique characters in the dataset")

There are 83 unique characters in the dataset
```

2.3 Process the dataset for the learning task

Let's take a step back and consider our prediction task. We're trying to train a RNN model to learn patterns in ABC music, and then use this model to generate (i.e., predict) a new piece of music based on this learned information.

Breaking this down, what we're really asking the model is: given a character, or a sequence of characters, what is the most probable next character? We'll train the model to perform this task.

To achieve this, we will input a sequence of characters to the model, and train the model to predict the output, that is, the following character at each time step. RNNs maintain an internal state that depends on previously seen elements, so information about all characters seen up until a given moment will be taken into account in generating the prediction.

Vectorize the text

Before we begin training our RNN model, we'll need to create a numerical representation of our text-based dataset. To do this, we'll generate two lookup tables: one that maps characters to numbers, and a second that maps numbers back to characters. Recall that we just identified the unique characters present in the text.

```
### Define numerical representation of text ###

# Create a mapping from character to unique index.
# For example, to get the index of the character "d",
#   we can evaluate `char2idx["d"]`.
char2idx = {u:i for i, u in enumerate(vocab)}

# Create a mapping from indices to characters. This is
#   the inverse of char2idx and allows us to convert back
#   from unique index to the character in our vocabulary.
idx2char = np.array(vocab)
```

This gives us an integer representation for each character. Observe that the unique characters (i.e., our vocabulary) in the text are mapped as indices from 0 to `len(unique)`. Let's take a peek at this numerical representation of our dataset:

```
print('{')
for char,_ in zip(char2idx, range(20)):
    print('  {:4s}: {:3d},'.format(repr(char), char2idx[char]))
print('  ...\n}')
```

```
{
  '\n':   0,
  ' ' :   1,
  '!' :   2,
  '"' :   3,
  '#' :   4,
  "'" :   5,
```

```
  '(' :   6,
  ')' :   7,
  ',' :   8,
  '-' :   9,
  '.' :  10,
  '/' :  11,
  '0' :  12,
  '1' :  13,
  '2' :  14,
  '3' :  15,
  '4' :  16,
  '5' :  17,
  '6' :  18,
  '7' :  19,
  ...
}

### Vectorize the songs string ###

'''TODO: Write a function to convert the all songs string to a
vectorized
    (i.e., numeric) representation. Use the appropriate mapping
    above to convert from vocab characters to the corresponding
indices.

  NOTE: the output of the `vectorize_string` function
  should be a np.array with `N` elements, where `N` is
  the number of characters in the input string
'''
def vectorize_string(string):
  vectorized_output = np.array([char2idx[char] for char in string])
  return vectorized_output

# def vectorize_string(string):
  # TODO

vectorized_songs = vectorize_string(songs_joined)
```

We can also look at how the first part of the text is mapped to an integer representation:

```
print ('{} ---- characters mapped to int ---->
{}'.format(repr(songs_joined[:10]), vectorized_songs[:10]))
# check that vectorized_songs is a numpy array
assert isinstance(vectorized_songs, np.ndarray), "returned result
should be a numpy array"
```

```
'X:1\nT:Alex' ---- characters mapped to int ----> [49 22 13  0 45 22
26 67 60 79]
```

Create training examples and targets

Our next step is to actually divide the text into example sequences that we'll use during training. Each input sequence that we feed into our RNN will contain `seq_length` characters from the text. We'll also need to define a target sequence for each input sequence, which will be used in training the RNN to predict the next character. For each input, the corresponding target will contain the same length of text, except shifted one character to the right.

To do this, we'll break the text into chunks of `seq_length+1`. Suppose `seq_length` is 4 and our text is "Hello". Then, our input sequence is "Hell" and the target sequence is "ello".

The batch method will then let us convert this stream of character indices to sequences of the desired size.

```
### Batch definition to create training examples ###

def get_batch(vectorized_songs, seq_length, batch_size):
  # the length of the vectorized songs string
  n = vectorized_songs.shape[0] - 1
  # randomly choose the starting indices for the examples in the
training batch
  idx = np.random.choice(n-seq_length, batch_size)

  '''TODO: construct a list of input sequences for the training
batch'''
  input_batch = [vectorized_songs[i : i+seq_length] for i in idx]
  # input_batch = # TODO
  '''TODO: construct a list of output sequences for the training
batch'''
  output_batch = [vectorized_songs[i+1 : i+seq_length+1] for i in idx]
  # output_batch = # TODO

  # x_batch, y_batch provide the true inputs and targets for network
training
  x_batch = np.reshape(input_batch, [batch_size, seq_length])
  y_batch = np.reshape(output_batch, [batch_size, seq_length])
  return x_batch, y_batch


# Perform some simple tests to make sure your batch function is
working properly!
test_args = (vectorized_songs, 10, 2)
if not mdl.lab1.test_batch_func_types(get_batch, test_args) or \
   not mdl.lab1.test_batch_func_shapes(get_batch, test_args) or \
   not mdl.lab1.test_batch_func_next_step(get_batch, test_args):
   print("======\n[FAIL] could not pass tests")
else:
   print("======\n[PASS] passed all tests!")
```

```
[PASS] test_batch_func_types
[PASS] test_batch_func_shapes
[PASS] test_batch_func_next_step
======
[PASS] passed all tests!
```

For each of these vectors, each index is processed at a single time step. So, for the input at time step 0, the model receives the index for the first character in the sequence, and tries to predict the index of the next character. At the next timestep, it does the same thing, but the RNN considers the information from the previous step, i.e., its updated state, in addition to the current input.

We can make this concrete by taking a look at how this works over the first several characters in our text:

```
x_batch, y_batch = get_batch(vectorized_songs, seq_length=5,
batch_size=1)

for i, (input_idx, target_idx) in enumerate(zip(np.squeeze(x_batch),
np.squeeze(y_batch))):
    print("Step {:3d}".format(i))
    print("  input: {} ({:s})".format(input_idx,
repr(idx2char[input_idx])))
    print("  expected output: {} ({:s})".format(target_idx,
repr(idx2char[target_idx])))
```

```
Step   0
  input: 62 ('g')
  expected output: 61 ('f')
Step   1
  input: 61 ('f')
  expected output: 1 (' ')
Step   2
  input: 1 (' ')
  expected output: 62 ('g')
Step   3
  input: 62 ('g')
  expected output: 15 ('3')
Step   4
  input: 15 ('3')
  expected output: 82 ('|')
```

2.4 The Recurrent Neural Network (RNN) model

Now we're ready to define and train a RNN model on our ABC music dataset, and then use that trained model to generate a new song. We'll train our RNN using batches of song snippets from our dataset, which we generated in the previous section.

The model is based off the LSTM architecture, where we use a state vector to maintain information about the temporal relationships between consecutive characters. The final output of the LSTM is then fed into a fully connected `Dense` layer where we'll output a softmax over each character in the vocabulary, and then sample from this distribution to predict the next character.

As we introduced in the first portion of this lab, we'll be using the Keras API, specifically, `tf.keras.Sequential`, to define the model. Three layers are used to define the model:

- `tf.keras.layers.Embedding`: This is the input layer, consisting of a trainable lookup table that maps the numbers of each character to a vector with `embedding_dim` dimensions.
- `tf.keras.layers.LSTM`: Our LSTM network, with size `units=rnn_units`.
- `tf.keras.layers.Dense`: The output layer, with `vocab_size` outputs.

Define the RNN model

Now, we will define a function that we will use to actually build the model.

```
def LSTM(rnn_units):
  return tf.keras.layers.LSTM(
    rnn_units,
    return_sequences=True,
    recurrent_initializer='glorot_uniform',
    recurrent_activation='sigmoid',
    stateful=True,
  )
```

The time has come! Fill in the `TODOs` to define the RNN model within the `build_model` function, and then call the function you just defined to instantiate the model!

```
### Defining the RNN Model ###

'''TODO: Add LSTM and Dense layers to define the RNN model using the
Sequential API.'''
def build_model(vocab_size, embedding_dim, rnn_units, batch_size):
  model = tf.keras.Sequential([
    # Layer 1: Embedding layer to transform indices into dense vectors
    #   of a fixed embedding size
    tf.keras.layers.Embedding(vocab_size, embedding_dim,
batch_input_shape=[batch_size, None]),

    # Layer 2: LSTM with `rnn_units` number of units.
    # TODO: Call the LSTM function defined above to add this layer.
    LSTM(rnn_units),
    # LSTM('''TODO'''),

    # Layer 3: Dense (fully-connected) layer that transforms the LSTM
```

```
output
    #   into the vocabulary size.
    # TODO: Add the Dense layer.
    tf.keras.layers.Dense(vocab_size)
    # '''TODO: DENSE LAYER HERE'''
  ])

  return model

# Build a simple model with default hyperparameters. You will get the
#   chance to change these later.
model = build_model(len(vocab), embedding_dim=256, rnn_units=1024,
batch_size=32)
```

Test out the RNN model

It's always a good idea to run a few simple checks on our model to see that it behaves as expected.

First, we can use the `Model.summary` function to print out a summary of our model's internal workings. Here we can check the layers in the model, the shape of the output of each of the layers, the batch size, etc.

```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding (Embedding)       (32, None, 256)           21248

 lstm (LSTM)                 (32, None, 1024)          5246976

 dense (Dense)               (32, None, 83)            85075

=================================================================
Total params: 5353299 (20.42 MB)
Trainable params: 5353299 (20.42 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

We can also quickly check the dimensionality of our output, using a sequence length of 100. Note that the model can be run on inputs of any length.

```
x, y = get_batch(vectorized_songs, seq_length=100, batch_size=32)
pred = model(x)
print("Input shape:      ", x.shape, " # (batch_size,
sequence_length)")
```

```
print("Prediction shape: ", pred.shape, "# (batch_size,
sequence_length, vocab_size)")
```

```
Input shape:       (32, 100)  # (batch_size, sequence_length)
Prediction shape:  (32, 100, 83) # (batch_size, sequence_length,
vocab_size)
```

Predictions from the untrained model

Let's take a look at what our untrained model is predicting.

To get actual predictions from the model, we sample from the output distribution, which is defined by a `softmax` over our character vocabulary. This will give us actual character indices. This means we are using a categorical distribution to sample over the example prediction. This gives a prediction of the next character (specifically its index) at each timestep.

Note here that we sample from this probability distribution, as opposed to simply taking the `argmax`, which can cause the model to get stuck in a loop.

Let's try this sampling out for the first example in the batch.

```
sampled_indices = tf.random.categorical(pred[0], num_samples=1)
sampled_indices = tf.squeeze(sampled_indices,axis=-1).numpy()
sampled_indices
```

```
array([44, 39, 56, 33, 57,  4, 31, 53, 54, 37,  3, 28, 13, 20, 32, 73,
14,
       38, 33, 81, 14, 69, 17, 35, 71, 57, 21, 80,  8, 11, 50, 51, 23,
75,
       69, 46, 38, 31, 72, 73, 40, 79, 74, 60,  0, 53, 63, 21, 82, 58,
35,
       47, 60, 29, 42, 80, 80, 24, 18, 77, 78, 49, 11, 21,  9, 13, 68,
73,
       78, 43, 42, 33, 14, 67, 66,  4, 30, 20, 54, 44, 78, 70, 29, 18,
71,
       20, 29, 76, 56, 47, 65,  8,  4, 19, 81, 28,  8, 75, 68, 39])
```

We can now decode these to see the text predicted by the untrained model:

```
print("Input: \n", repr("".join(idx2char[x[0]])))
print()
print("Next Char Predictions: \n",
repr("".join(idx2char[sampled_indices])))
```

```
Input:
 'cd^f|g^fdf gabg|fdbg fdcd|BGA^F G=FDF|!\nG2GA BABG|FcFG AGFD|GABG
dBGA|B2G2 G3:|!\nf|gfdf gabg|gfdf dB'

Next Char Predictions:
```

```
 'SNaHb#F]^L"C18Gr2MHz2n5Jpb9y,/YZ<tnUMFqrOxse\n]h9|cJVeDQyy=6vwX/9-
1mrwRQH2lk#E8^SwoD6p8DuaVj,#7zC,tmN'
```

As you can see, the text predicted by the untrained model is pretty nonsensical! How can we do better? We can train the network!

2.5 Training the model: loss and training operations

Now it's time to train the model!

At this point, we can think of our next character prediction problem as a standard classification problem. Given the previous state of the RNN, as well as the input at a given time step, we want to predict the class of the next character -- that is, to actually predict the next character.

To train our model on this classification task, we can use a form of the `crossentropy` loss (negative log likelihood loss). Specifically, we will use the `sparse_categorical_crossentropy` loss, as it utilizes integer targets for categorical classification tasks. We will want to compute the loss using the true targets -- the `labels` -- and the predicted targets -- the `logits`.

Let's first compute the loss using our example predictions from the untrained model:

```
### Defining the loss function ###

'''TODO: define the loss function to compute and return the loss
between
    the true labels and predictions (logits). Set the argument
from_logits=True.'''
def compute_loss(labels, logits):
  loss = tf.keras.losses.sparse_categorical_crossentropy(labels,
logits, from_logits=True)
  # loss = tf.keras.losses.sparse_categorical_crossentropy('''TODO''',
'''TODO''', from_logits=True) # TODO
  return loss

'''TODO: compute the loss using the true next characters from the
example batch
    and the predictions from the untrained model several cells
above'''
example_batch_loss = compute_loss(y, pred)
# example_batch_loss = compute_loss('''TODO''', '''TODO''') # TODO

print("Prediction shape: ", pred.shape, " # (batch_size,
sequence_length, vocab_size)")
print("scalar_loss:      ", example_batch_loss.numpy().mean())
```

```
Prediction shape:  (32, 100, 83)  # (batch_size, sequence_length,
vocab_size)
scalar_loss:       4.4161644
```

Let's start by defining some hyperparameters for training the model. To start, we have provided some reasonable values for some of the parameters. It is up to you to use what we've learned in class to help optimize the parameter selection here!

```
### Hyperparameter setting and optimization ###

vocab_size = len(vocab)

# Model parameters:
params = dict(
  num_training_iterations = 3000,  # Increase this to train longer
  batch_size = 8,  # Experiment between 1 and 64
  seq_length = 100,  # Experiment between 50 and 500
  learning_rate = 5e-3,  # Experiment between 1e-5 and 1e-1
  embedding_dim = 256,
  rnn_units = 1024,  # Experiment between 1 and 2048
)

# Checkpoint location:
checkpoint_dir = './training_checkpoints'
checkpoint_prefix = os.path.join(checkpoint_dir, "my_ckpt")
```

Having defined our hyperparameters we can set up for experiment tracking with Comet. `Experiment` are the core objects in Comet and will allow us to track training and model development. Here we have written a short function to create a new comet experiment. Note that in this setup, when hyperparameters change, you can run the `create_experiment()` function to initiate a new experiment. All experiments defined with the same `project_name` will live under that project in your Comet interface.

```
### Create a Comet experiment to track our training run ###

def create_experiment():
  # end any prior experiments
  if 'experiment' in locals():
    experiment.end()

  # initiate the comet experiment for tracking
  experiment = comet_ml.Experiment(
                  api_key=COMET_API_KEY,
                  project_name="6S191_Lab1_Part2")
  # log our hyperparameters, defined above, to the experiment
  for param, value in params.items():
    experiment.log_parameter(param, value)
  experiment.flush()

  return experiment
```

Now, we are ready to define our training operation -- the optimizer and duration of training -- and use this function to train the model. You will experiment with the choice of optimizer and

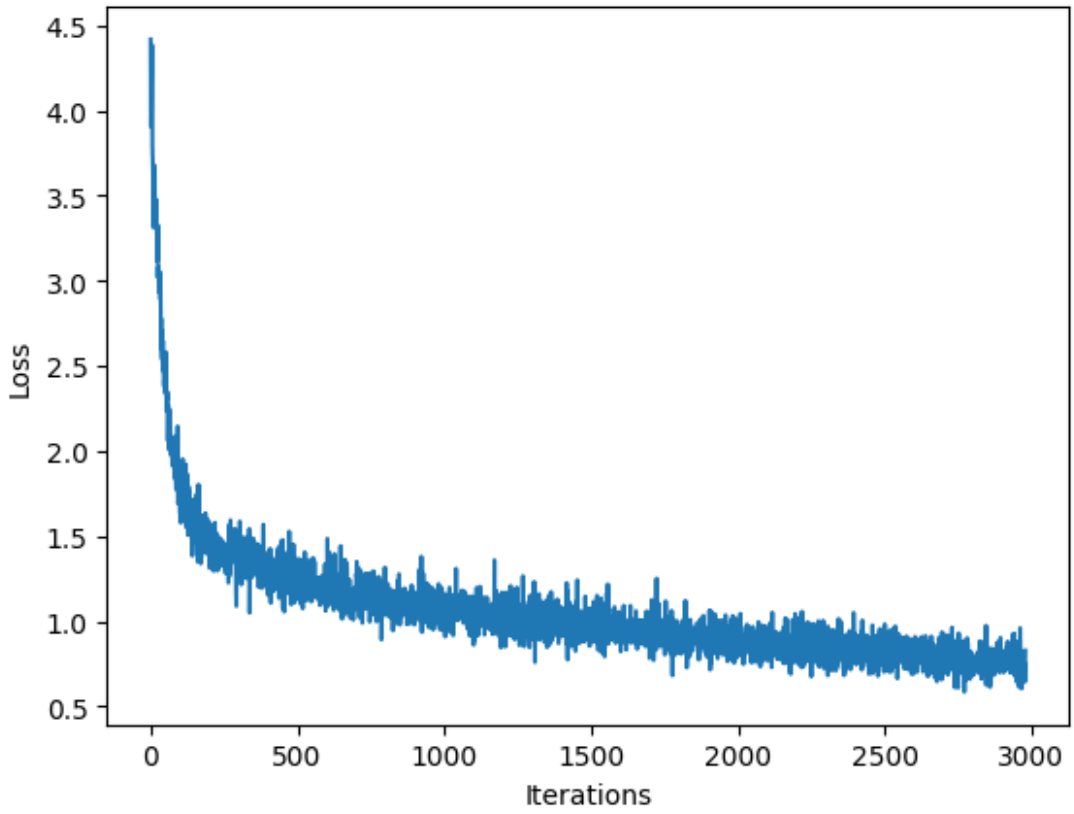
the duration for which you train your models, and see how these changes affect the network's output. Some optimizers you may like to try are `Adam` and `Adagrad`.

First, we will instantiate a new model and an optimizer. Then, we will use the `tf.GradientTape` method to perform the backpropagation operations.

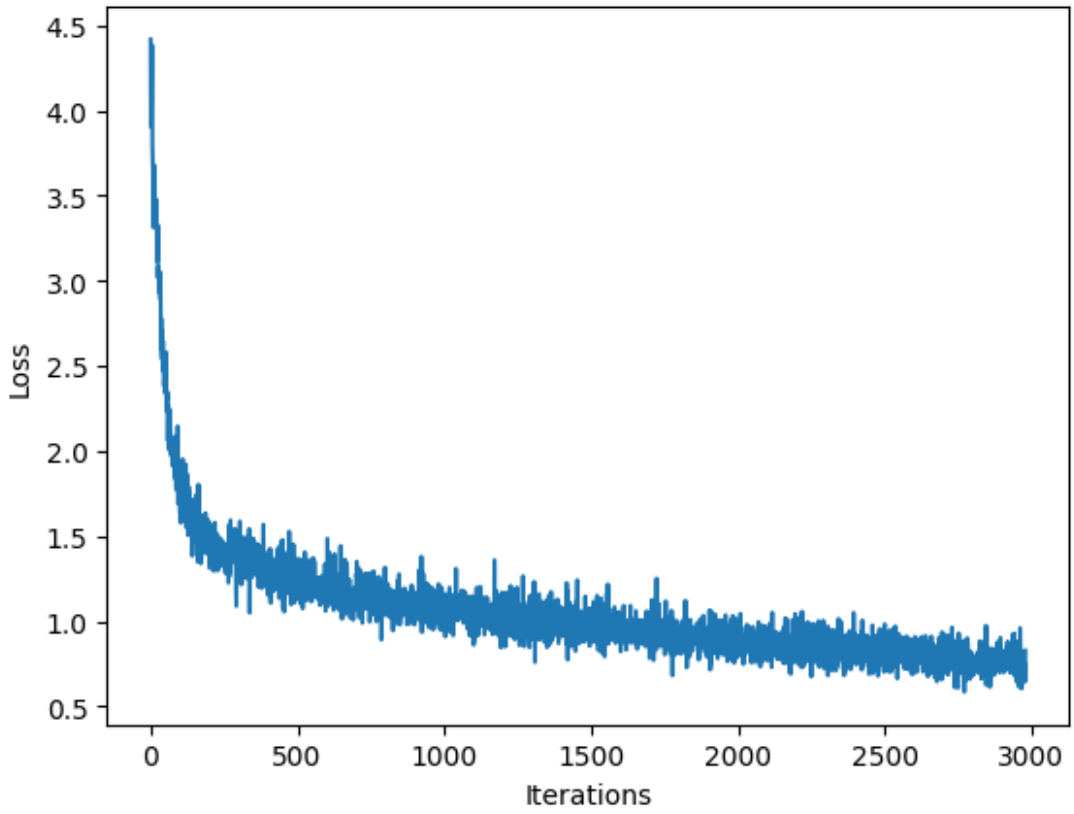
We will also generate a print-out of the model's progress through training, which will help us easily visualize whether or not we are minimizing the loss.

```
### Define optimizer and training operation ###

'''TODO: instantiate a new model for training using the `build_model`
  function and the hyperparameters created above.'''
model = build_model(vocab_size, params["embedding_dim"],
params["rnn_units"], params["batch_size"])
# model = build_model('''TODO: arguments''')

'''TODO: instantiate an optimizer with its learning rate.
  Checkout the tensorflow website for a list of supported optimizers.
  https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/
  Try using the Adam optimizer to start.'''
optimizer = tf.keras.optimizers.Adam(params["learning_rate"])
# optimizer = # TODO

@tf.function
def train_step(x, y):
  # Use tf.GradientTape()
  with tf.GradientTape() as tape:

    '''TODO: feed the current input into the model and generate
predictions'''
    y_hat = model(x) # TODO
    # y_hat = model('''TODO''')

    '''TODO: compute the loss!'''
    loss = compute_loss(y, y_hat) # TODO
    # loss = compute_loss('''TODO''', '''TODO''')

  # Now, compute the gradients
  '''TODO: complete the function call for gradient computation.
      Remember that we want the gradient of the loss with respect all
      of the model parameters.
      HINT: use `model.trainable_variables` to get a list of all model
      parameters.'''
  grads = tape.gradient(loss, model.trainable_variables) # TODO
  # grads = tape.gradient('''TODO''', '''TODO''')

  # Apply the gradients to the optimizer so it can update the model
accordingly
  optimizer.apply_gradients(zip(grads, model.trainable_variables))
  return loss
```

```
##################
# Begin training!#
##################

history = []
plotter = mdl.util.PeriodicPlotter(sec=2, xlabel='Iterations',
ylabel='Loss')
experiment = create_experiment()

if hasattr(tqdm, '_instances'): tqdm._instances.clear() # clear if it
exists
for iter in tqdm(range(params["num_training_iterations"])):

  # Grab a batch and propagate it through the network
  x_batch, y_batch = get_batch(vectorized_songs, params["seq_length"],
params["batch_size"])
  loss = train_step(x_batch, y_batch)

  # log the loss to the Comet interface! we will be able to track it
there.
  experiment.log_metric("loss", loss.numpy().mean(), step=iter)
  # Update the progress bar and also visualize within notebook
  history.append(loss.numpy().mean())
  plotter.plot(history)

  # Update the model with the changed weights!
  if iter % 100 == 0:
    model.save_weights(checkpoint_prefix)

# Save the trained model and the weights
model.save_weights(checkpoint_prefix)
experiment.flush()
```




```
100%|██████████| 3000/3000 [01:46<00:00, 28.12it/s]
COMET INFO: Uploading 47 metrics, params and output messages
```



2.6 Generate music using the RNN model

Now, we can use our trained RNN model to generate some music! When generating music, we'll have to feed the model some sort of seed to get it started (because it can't predict anything without something to start with!).

Once we have a generated seed, we can then iteratively predict each successive character (remember, we are using the ABC representation for our music) using our trained RNN. More specifically, recall that our RNN outputs a `softmax` over possible successive characters. For inference, we iteratively sample from these distributions, and then use our samples to encode a generated song in the ABC format.

Then, all we have to do is write it to a file and listen!

Restore the latest checkpoint

To keep this inference step simple, we will use a batch size of 1. Because of how the RNN state is passed from timestep to timestep, the model will only be able to accept a fixed batch size once it is built.

To run the model with a different `batch_size`, we'll need to rebuild the model and restore the weights from the latest checkpoint, i.e., the weights after the last checkpoint during training:

```
'''TODO: Rebuild the model using a batch_size=1'''
model = build_model(vocab_size, params["embedding_dim"],
params["rnn_units"], batch_size=1) # TODO
# model = build_model('''TODO''', '''TODO''', '''TODO''',
batch_size=1)

# Restore the model weights for the last checkpoint after training
model.load_weights(tf.train.latest_checkpoint(checkpoint_dir))
model.build(tf.TensorShape([1, None]))

model.summary()
```

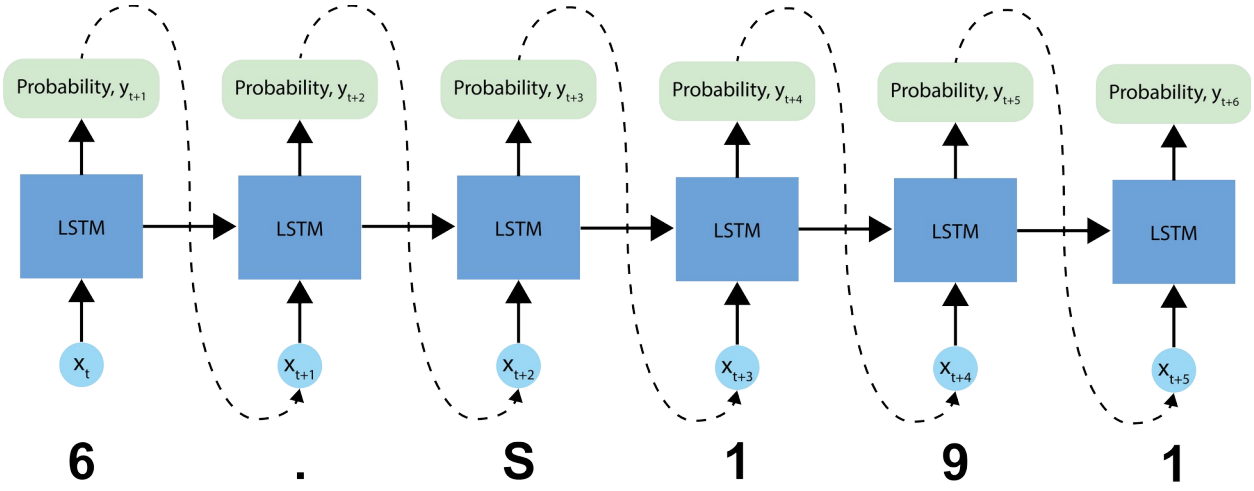
```
Model: "sequential_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding_2 (Embedding)     (1, None, 256)            21248

 lstm_2 (LSTM)               (1, None, 1024)           5246976

 dense_2 (Dense)             (1, None, 83)             85075

=================================================================
Total params: 5353299 (20.42 MB)
Trainable params: 5353299 (20.42 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

Notice that we have fed in a fixed `batch_size` of 1 for inference.

The prediction procedure

Now, we're ready to write the code to generate text in the ABC music format:

- Initialize a "seed" start string and the RNN state, and set the number of characters we want to generate.
- Use the start string and the RNN state to obtain the probability distribution over the next predicted character.
- Sample from multinomial distribution to calculate the index of the predicted character. This predicted character is then used as the next input to the model.
- At each time step, the updated RNN state is fed back into the model, so that it now has more context in making the next prediction. After predicting the next character, the updated RNN states are again fed back into the model, which is how it learns sequence dependencies in the data, as it gets more information from the previous predictions.



Complete and experiment with this code block (as well as some of the aspects of network definition and training!), and see how the model performs. How do songs generated after training with a small number of epochs compare to those generated after a longer duration of training?

```
### Prediction of a generated song ###

def generate_text(model, start_string, generation_length=1000):
  # Evaluation step (generating ABC text using the learned RNN model)

  '''TODO: convert the start string to numbers (vectorize)'''
  input_eval = [char2idx[s] for s in start_string] # TODO
  # input_eval = ['''TODO''']
  input_eval = tf.expand_dims(input_eval, 0)

  # Empty string to store our results
  text_generated = []

  # Here batch size == 1
  model.reset_states()
  tqdm._instances.clear()

  for i in tqdm(range(generation_length)):
      '''TODO: evaluate the inputs and generate the next character
predictions'''
      predictions = model(input_eval)
      # predictions = model('''TODO''')

      # Remove the batch dimension
      predictions = tf.squeeze(predictions, 0)

      '''TODO: use a multinomial distribution to sample'''
      predicted_id = tf.random.categorical(predictions, num_samples=1)
[-1,0].numpy()
      # predicted_id = tf.random.categorical('''TODO''',
```

```
num_samples=1)[-1,0].numpy()

      # Pass the prediction along with the previous hidden state
      #   as the next inputs to the model
      input_eval = tf.expand_dims([predicted_id], 0)

      '''TODO: add the predicted character to the generated text!'''
      # Hint: consider what format the prediction is in vs. the output
      text_generated.append(idx2char[predicted_id]) # TODO
      # text_generated.append('''TODO''')

  return (start_string + ''.join(text_generated))

'''TODO: Use the model and the function defined above to generate ABC
format text of length 1000!
    As you may notice, ABC files start with "X" - this may be a good
start string.'''
generated_text = generate_text(model, start_string="X",
generation_length=1000) # TODO
# generated_text = generate_text('''TODO''', start_string="X",
generation_length=1000)
```

```
100%|██████████| 1000/1000 [00:09<00:00, 108.52it/s]
```

Play back the generated music!

We can now call a function to convert the ABC format text to an audio file, and then play that back to check out our generated music! Try training longer if the resulting song is not long enough, or re-generating the song!

We will save the song to Comet -- you will be able to find your songs under the `Audio` and `Assets & Artificats` pages in your Comet interface for the project. Note the `log_asset()` documentation, where you will see how to specify file names and other parameters for saving your assets.

```
### Play back generated songs ###

generated_songs = mdl.lab1.extract_song_snippet(generated_text)

for i, song in enumerate(generated_songs):
  # Synthesize the waveform from a song
  waveform = mdl.lab1.play_song(song)

  # If its a valid song (correct syntax), lets play it!
  if waveform:
    print("Generated song", i)
    ipythondisplay.display(waveform)

    numeric_data = np.frombuffer(waveform.data, dtype=np.int16)
    wav_file_path = f"output_{i}.wav"
```

```
    write(wav_file_path, 88200, numeric_data)

    # save your song to the Comet interface -- you can access it there
    experiment.log_asset(wav_file_path)
```

```
Found 2 songs in text
Generated song 0

<IPython.lib.display.Audio object>

Generated song 1

<IPython.lib.display.Audio object>
```

```
# when done, end the comet experiment
experiment.end()
```

```
COMET WARNING: Couldn't retrieve Google Colab notebook content
COMET INFO:
---------------------------------------------------------------------------
-----------------
COMET INFO: Comet.ml Experiment Summary
COMET INFO:
---------------------------------------------------------------------------
-----------------
COMET INFO:   Data:
COMET INFO:     display_summary_level : 1
COMET INFO:     name                  : olympic_egret_7239
COMET INFO:     url                   :
https://www.comet.com/pkcoder420/6s191-lab1-
part2/2246f4f7e1e646988c753d1e7c909a3d
COMET INFO:   Metrics [count] (min, max):
COMET INFO:     loss [3000] : (0.5845882296562195, 4.418431758880615)
COMET INFO:   Parameters:
COMET INFO:     batch_size              : 8
COMET INFO:     embedding_dim           : 256
COMET INFO:     learning_rate           : 0.005
COMET INFO:     num_training_iterations : 3000
COMET INFO:     rnn_units               : 1024
COMET INFO:     seq_length              : 100
COMET INFO:   Uploads:
COMET INFO:     asset               : 2 (19.39 MB)
COMET INFO:     environment details : 1
COMET INFO:     filename            : 1
COMET INFO:     installed packages  : 1
COMET INFO:     notebook            : 1
COMET INFO:     os packages         : 1
COMET INFO:     source_code         : 1
COMET INFO:
```

2.7 Experiment and **get awarded for the best songs**!

Congrats on making your first sequence model in TensorFlow! It's a pretty big accomplishment, and hopefully you have some sweet tunes to show for it.

Consider how you may improve your model and what seems to be most important in terms of performance. Here are some ideas to get you started:

- How does the number of training epochs affect the performance?
- What if you alter or augment the dataset?
- Does the choice of start string significantly affect the result?

Try to optimize your model and submit your best song! **Participants will be eligible for prizes during the January 2024 offering. To enter the competition, you must upload the following to this submission link:**

- a recording of your song;
- iPython notebook with the code you used to generate the song;
- a description and/or diagram of the architecture and hyperparameters you used -- if there are any additional or interesting modifications you made to the template code, please include these in your description.

Name your file in the following format: `[FirstName]_[LastName]_RNNMusic`, followed by the file format (.zip, .mp4, .ipynb, .pdf, etc). ZIP files of all three components are preferred over individual files. If you submit individual files, you must name the individual files according to the above nomenclature.

You can also tweet us at @MITDeepLearning a copy of the song (but this will not enter you into the competition)! See this example song generated by a previous student (credit Ana Heart): song from May 20, 2020.

Have fun and happy listening!