DeepLearning.AI Tensorflow Developer

Natural Language Processing in TensorFlow: Week 4

Text Generation

Text generation requires training a sequence model where input is a word sequence and output is the predicted word. For generating training data, first vectorize the sequences and then for each sequence create n-gram phrases and append it to input_sequence list. Get the longest sequence length from all the sequences and apply pre-padding. From all the sequences take all words except the last one as input data and take the last word as output label. As text generation is a type of classification problem, we perform one-hot encoding of labels with vocabulary size using keras.utils.to_categorical method.

```
vectorize_layer = tf.keras.layers.TextVectorization()
vectorize_layer.adapt(corpus)
input_sequences = []
for line in corpus:
    sequence = vectorize_layer(line).numpy()
    for i in range(1, len(sequence)):
        n_gram_sequence = sequence[:i+1]
        input_sequences.append(n_gram_sequence)

max_sequence_len = max([len(x) for x in input_sequences])

input_sequences = np.array(tf.keras.utils.pad_sequences(input_sequences,
maxlen=max_sequence_len, padding='pre'))

xs, labels = input_sequences[:,:-1],input_sequences[:,-1]
ys = tf.keras.utils.to_categorical(labels, num_classes=vocab_size)
```

Perform classification model training with LSTM layer and categorical_crossentropy loss function. Train for larger epochs as it takes time to converge. Generate word sequence by concatenating generated word with previous input sequence. Bidirectional LSTM layer reduces word repetition on generated sentences. Also, can take random choice for avoiding repetition.

```
seed_text = "Laurence went to Dublin"
next_words = 100
for _ in range(next_words):
    sequence = vectorize_layer(seed_text)
    sequence = tf.keras.utils.pad_sequences([sequence], maxlen=max_sequence_len-1,
padding='pre')
    probabilities = model.predict(sequence, verbose=0)
    predicted = np.argmax(probabilities, axis=-1)[0]
    if predicted != 0:
        output_word = vocabulary[predicted]
        seed_text += " " + output_word
print(seed_text)
```

Poetry

Generate song lyrics by training sequence model with large corpus of lyrics. During training update the ADAM optimizer learning rate including embedding dimension and LSTM units in keras layers to observe the performance.

```
epochs = 100
embedding_dim = 100
lstm_units = 150
learning_rate = 0.01


model = tf.keras.models.Sequential([
    tf.keras.Input(shape=(max_sequence_len-1,)),
    tf.keras.layers.Embedding(vocab_size, embedding_dim),
    tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(lstm_units)),
    tf.keras.layers.Dense(vocab_size, activation='softmax')])


model.compile(loss='categorical_crossentropy',
   optimizer=tf.keras.optimizers.Adam(learning_rate=learning_rate),
   metrics=['accuracy'])
history = model.fit(dataset, epochs=epochs)
```

Character based text generation reduces the number of unique output labels which helps with memory requirements for one hot encoding. First create vocabulary of unique characters from given text sequences. Then vectorize each text of the sequence using StringLookup method.

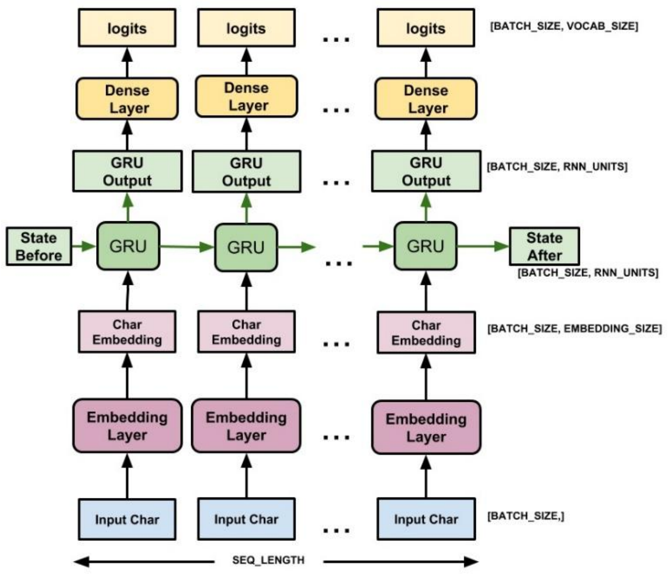
```
vocab = sorted(set(text))
example_texts = ['abcdefg', 'xyz']


chars = tf.strings.unicode_split(example_texts, input_encoding='UTF-8')
ids_from_chars = tf.keras.layers.StringLookup(vocabulary=list(vocab), mask_token=None)
ids = ids_from_chars(chars)
```

For training we consider char sequence of fixed seq_length where the input and output are char stream of seq_length. So, we divide the text into the chunk of length seq_length + 1 and shift the sequence to one char right for creating output label.

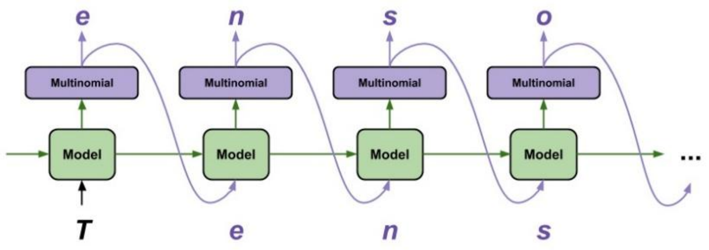
```
all_ids = ids_from_chars(tf.strings.unicode_split(text, 'UTF-8'))
ids_dataset = tf.data.Dataset.from_tensor_slices(all_ids)
sequences = ids_dataset.batch(seq_length+1, drop_remainder=True)


def split_input_target(sequence):
   input_text = sequence[:-1]
   target_text = sequence[1:]
   return input_text, target_text


dataset_split = sequences.map(split_input_target)
```

Then we apply batching, shuffling and caching on char sequence dataset. For sequence model we stack embedding, GRU and Dense layers. For each character the model looks up the embedding, runs the GRU one timestep with the embedding as input, and applies the dense layer to generate logits predicting the log-likelihood of the next character. We perform training with SparseCategoricalCrossentropy loss function with logits set to TRUE and ADAM optimizer.

```
BATCH_SIZE = 64
BUFFER_SIZE = 10000
dataset = (
     dataset_split
    .shuffle(BUFFER_SIZE)
    .batch(BATCH_SIZE, drop_remainder=True)
    .cache()
    .prefetch(tf.data.AUTOTUNE))

model = tf.keras.Sequential([
    tf.keras.layers.Embedding(vocab_size, embedding_dim),
    tf.keras.layers.GRU(rnn_units, return_sequences=True),
    tf.keras.layers.Dense(vocab_size)])

loss = tf.losses.SparseCategoricalCrossentropy(from_logits=True)
model.compile(optimizer='adam', loss=loss, metrics=['sparse_categorical_accuracy'])
history = model.fit(dataset, epochs=EPOCHS)
```



Generating text with the char sequence model is to run it in a loop, and keep track of the model's internal state in each step. Each time the model is called, previous predicted char along with previous state is passed as parameter.



```
def generate_one_step(self, inputs, states=None):
    # Convert strings to token IDs.
    input_chars = tf.strings.unicode_split(inputs, 'UTF-8')
    input_ids = self.ids_from_chars(input_chars).to_tensor()
    # Embedding layer
    x = self.model.layers[0](input_ids)
    # GRU layer
    x = self.model.layers[1](x, initial_state=states)
    # Get the hidden state of the last timestep
    states = x[:, -1, :]
    # Dense layer
    predicted_logits = self.model.layers[2](x)
    # Only use the last prediction.
    predicted_logits = predicted_logits[:, -1, :]
    predicted_logits = predicted_logits/self.temperature
    # Apply the prediction mask: prevent "[UNK]" from being generated.
    predicted_logits = predicted_logits + self.prediction_mask
    # # Sample the output logits to generate token IDs.
    predicted_ids = tf.random.categorical(predicted_logits, num_samples=1)
    predicted_ids = tf.squeeze(predicted_ids, axis=-1)
    # # Convert from token ids to characters
    predicted_chars = self.chars_from_ids(predicted_ids)
    # Return the characters and model state.
    return predicted_chars, states
```