WIA1006 Machine Learning Lab 2

Group: Undergraduate Descent

Name	Matric No.	Occurrence
SAM WEI HONG	U2102776	6
Yap Yoong Siew	S2122194	5
DAMIA FATIHAH BINTI IMRAN	U2001010	6
LEE KEAT EN	U2102836	6
ARINA NATASHA BINTI HOURI	U2000655	8

Part 1: TensorFlow

1. Explain the difference between a 1-D and a 2-D tensor.

A 1-D tensor has only one dimension, which represents an array or a vector. On the other hand, a 2-D tensor has two dimensions, which represents a matrix. In terms of rank, a 1-D tensor has a rank of 1, while a 2-D tensor has a rank of 2.

2. Provide the syntax to define a 4-D tensor with specific dimensions.

Given that the dimensions of a 4-D tensor are (a × b × c × d), we will define the tensor of zeros using the syntax:

tensor = tf.zeros([a, b, c, d])

3. Provide examples of defining tensor computations.

To add two tensors, we use the tf.add() function. To perform subtraction between two tensors, we use the tf.subtract() function. For multiplication, we use the tf.multiply() function.

Example Code:

tf.add(1, 4) will have output of 5 since 1 plus 4 is equal to 5.

tf.subtract(8, 4) will have output of 4 since 8 minus 4 is equal to 4.

tf.multiply(2, 3) will have output of 6 since 2 times 3 is equal to 6.

4. What is Dense and what is Sequential in TensorFlow. Explain with the necessary examples.

Dense is a fully-connected neural network layer where each neuron in the layer is connected to all neurons of its previous layer. Sequential is a linear stack of layers where each layer has exactly one input tensor and one output tensor. These layers aid in implementing common neural networks operations, such as update weights, compute losses, and define inter-layer connectivity.

Using the Sequential API, we can create neural networks by stacking layers together. For instance, we can create a Sequential model and add some Dense layers to it. This model will take input arrays of shape (None, 16) and output arrays of shape (None, 32).

```
model = tf.keras.Sequential()

model.add(tf.keras.Input(shape=(16, )))

model.add(tf.keras.layers.Dense(32, activation='sigmoid'))

model.add(tf.keras.layers.Dense(32))

model_output = model(input).numpy()
```

5. What is a SubClass in TensorFlow? How do you specify custom behavior using a SubClass?

A subclass of Model class groups layers together to enable model training and inference, which allows us to define the forward pass through the network using the call function. Subclass consists of two functions, which are _init_ and call. In the _init_ method, all the tf.keras layers or custom implemented layers are defined, and those layers based on the network design will be called inside the call method to perform a forward pass. Essentially, subclassing allows the flexibility to define custom layers, custom training loops, custom activation functions, and custom models. It is fully-customizable and enables us to implement our own custom forward pass of the model.

To specify custom behaviours, such as during training and inference, we can use boolean arguments in the call method. For instance, we can pass an isIdentity argument to the call method so that the method will only output the input if this argument is set to true. Hence, we can say this isIdentity argument controls the network behaviours.

6. Explain how automatic differentiation works in TensorFlow.

TensorFlow needs to remember what operations happen in what order during the forward pass. Then, during the backward pass, TensorFlow traverses this list of operations in reverse order to compute gradients.

TensorFlow provides the tf.GradientTape API for automatic differentiation; that is, computing the gradient of computation with respect to some inputs, usually tf.Variables. TensorFlow "records" relevant operations executed inside the context of a tf.GradientTape onto a "tape". TensorFlow then uses that tape to compute the gradients of a "recorded" computation using reverse mode differentiation. The tape is then discarded by default in which subsequent calls to it will cause a runtime error.

Part 2: Music Generation with RNNs

Provide a summary of how music generation is achieved with RNNs using TensorFlow.

Explain every step in detail.

To generate a piece of music, we use RNNs which are suited for processing sequential data like audio. A RNN maintains an internal state that depends on previously seen elements and uses the information about all elements seen until a given moment in generating the prediction. By providing it with a character or a sequence of characters to begin, the model will predict the next highest probability character. The model will then accept the predicted character as the new input and generate the subsequent characters. This process continues until a new piece of music is produced.

We will first train our RNN model to learn patterns in ABC music and use this learned information for the character prediction. To do this, we gather a dataset of Irish folk songs represented in ABC notation. Then, we create an integer representation of the text-based dataset to be used by the model. After computing the number of unique characters in this dataset, we can create two lookup tables, where one maps each character in the vocabulary to a different integer, and another maps the integer back to the original character. Using them, we can convert a song into a vectorised (numeric) string to feed the model and then convert the model's output back to characters for the music generation.

Following this, we divide the vectorised string into batches of example sequences, each consisting of a fixed number of consecutive characters from the string. Every input sequence has a target sequence which shifts one character to the right and is expected to be outputted by the RNN model. To obtain such sequences, we split the string into chunks. Each input sequence excludes the last character of a chunk, while each target sequence excludes the first character of the chunk.

With the batches of song snippets prepared, we can now define and train our RNN model using the Keras API (ts.keras.Sequential). The model is based on the Long Short-Term Memory (LSTM) architecture that uses a state vector to maintain information about the temporal relationships between consecutive characters. The embedding layer acts as the input layer, which maps the numbers of each character to a dense vector of a fixed embedding size using a lookup table. For the hidden layers, we use LSTM networks with sizes of rnn_units. Its output is fed into a fully connected Dense layer and applied a softmax function over each character in the vocabulary to predict the next character.

Next, we define the loss function of the network using the sparse categorical cross entropy loss (negative log likelihood loss) since our task is considered a classification problem. Moreover, we define the hyperparameters and optimisation parameters for the model training. Then, we can compute the loss function after we feed the input into the model and generate the predictions. The gradient is then computed using tf.GradientTape and applied to the optimiser to

perform the backpropagation. We can also generate a graph of loss against iterations to ensure that the loss function is decreasing.

After the model completes its training, we can finally use it to generate some music. We first initialise a “seed” start string and the RNN state and set the number of characters that we want to generate. With the start string and the RNN state, we can obtain the probability distribution over the next predicted character. We then use a multinomial distribution to sample and calculate the index of the predicted character. At each time step, the updated RNN state is fed back into the model to have more context in making the subsequent predictions. Since the model gets more information from the previous predictions, it is able to learn the sequential dependencies in the music. Finally, we convert the output of the model, which is in ABC format, into an audio file so that we can play our generated music.