Logs dictionary

July 30, 2021

1 Using the logs dictionary

In this reading, we will learn how to take advantage of the `logs` dictionary in Keras to define our own callbacks and check the progress of a model.

```
In [1]: import tensorflow as tf
        print(tf.__version__)
```

```
2.0.0
```

The `logs` dictionary stores the loss value, along with all of the metrics we are using at the end of a batch or epoch.

We can incorporate information from the `logs` dictionary into our own custom callbacks.

Let's see this in action in the context of a model we will construct and fit to the `sklearn` diabetes dataset that we have been using in this module.

Let's first import the dataset, and split it into the training and test sets.

```
In [2]: # Load the diabetes dataset

        from sklearn.datasets import load_diabetes

        diabetes_dataset = load_diabetes()
```

```
In [3]: # Save the input and target variables

        from sklearn.model_selection import train_test_split

        data = diabetes_dataset['data']
        targets = diabetes_dataset['target']
```

```
In [4]: # Split the data set into training and test sets

        train_data, test_data, train_targets, test_targets = train_test_split(data, targets, t
```

Now we construct our model.

```
In [5]: # Build the model

        from tensorflow.keras.models import Sequential
        from tensorflow.keras.layers import Dense

        model = tf.keras.Sequential([
            Dense(128, activation='relu', input_shape=(train_data.shape[1],)),
            Dense(64,activation='relu'),
            tf.keras.layers.BatchNormalization(),
            Dense(64, activation='relu'),
            Dense(64, activation='relu'),
            Dense(1)
        ])
```

We now compile the model, with * Mean squared error as the loss function, * the Adam optimizer, and * Mean absolute error (`mae`) as a metric.

```
In [6]: # Compile the model

        model.compile(loss='mse', optimizer="adam", metrics=['mae'])
```

1.0.1 Defining a custom callback

Now we define our custom callback using the `logs` dictionary to access the loss and metric values.

```
In [7]: # Create the custom callback

        class LossAndMetricCallback(tf.keras.callbacks.Callback):

            # Print the loss after every second batch in the training set
            def on_train_batch_end(self, batch, logs=None):
                if batch %2 ==0:
                    print('\n After batch {}, the loss is {:7.2f}.'.format(batch, logs['loss']

            # Print the loss after each batch in the test set
            def on_test_batch_end(self, batch, logs=None):
                print('\n After batch {}, the loss is {:7.2f}.'.format(batch, logs['loss']))

            # Print the loss and mean absolute error after each epoch
            def on_epoch_end(self, epoch, logs=None):
                print('Epoch {}: Average loss is {:7.2f}, mean absolute error is {:7.2f}.'.for

            # Notify the user when prediction has finished on each batch
            def on_predict_batch_end(self,batch, logs=None):
                print("Finished prediction on batch {}!".format(batch))
```

We now fit the model to the data, and specify that we would like to use our custom callback `LossAndMetricCallback()`.

```
In [8]: # Train the model

        history = model.fit(train_data, train_targets, epochs=20,
                            batch_size=100, callbacks=[LossAndMetricCallback()], verbose=False
```

```

 After batch 0, the loss is 30323.72.

 After batch 2, the loss is 26164.28.
Epoch 0: Average loss is 28525.89, mean absolute error is  151.17.

 After batch 0, the loss is 27180.83.

 After batch 2, the loss is 27484.88.
Epoch 1: Average loss is 28404.80, mean absolute error is  150.83.

 After batch 0, the loss is 27664.53.

 After batch 2, the loss is 29737.17.
Epoch 2: Average loss is 28241.58, mean absolute error is  150.34.

 After batch 0, the loss is 25934.97.

 After batch 2, the loss is 27198.30.
Epoch 3: Average loss is 27995.85, mean absolute error is  149.61.

 After batch 0, the loss is 30869.27.

 After batch 2, the loss is 27201.69.
Epoch 4: Average loss is 27650.10, mean absolute error is  148.57.

 After batch 0, the loss is 24326.89.

 After batch 2, the loss is 28813.03.
Epoch 5: Average loss is 27150.59, mean absolute error is  147.09.

 After batch 0, the loss is 23458.87.

 After batch 2, the loss is 32664.55.
Epoch 6: Average loss is 26504.15, mean absolute error is  145.14.

 After batch 0, the loss is 26053.36.

 After batch 2, the loss is 25112.98.
Epoch 7: Average loss is 25683.33, mean absolute error is  142.61.

 After batch 0, the loss is 26941.66.
 After batch 2, the loss is 22569.52.
Epoch 8: Average loss is 24664.00, mean absolute error is  139.40.

 After batch 0, the loss is 26090.36.

 After batch 2, the loss is 24594.71.
Epoch 9: Average loss is 23429.86, mean absolute error is  135.41.

 After batch 0, the loss is 20444.63.

 After batch 2, the loss is 24768.77.
Epoch 10: Average loss is 21916.99, mean absolute error is  130.48.

 After batch 0, the loss is 21967.53.

 After batch 2, the loss is 19009.12.
Epoch 11: Average loss is 20124.52, mean absolute error is  124.41.

 After batch 0, the loss is 16625.52.

 After batch 2, the loss is 17659.37.
Epoch 12: Average loss is 18073.49, mean absolute error is  117.00.

 After batch 0, the loss is 15536.90.

 After batch 2, the loss is 13876.30.
Epoch 13: Average loss is 16005.16, mean absolute error is  108.55.

 After batch 0, the loss is 15191.41.

 After batch 2, the loss is 10114.77.
Epoch 14: Average loss is 13769.83, mean absolute error is   98.50.

 After batch 0, the loss is 13722.21.

 After batch 2, the loss is 10132.80.
Epoch 15: Average loss is 11628.14, mean absolute error is   88.49.

 After batch 0, the loss is 8307.27.

 After batch 2, the loss is 8979.81.
Epoch 16: Average loss is 9457.71, mean absolute error is   77.50.

 After batch 0, the loss is 7161.81.

 After batch 2, the loss is 7735.57.
Epoch 17: Average loss is 7790.20, mean absolute error is   68.72.
```

```
 After batch 0, the loss is 5642.38.

 After batch 2, the loss is 6502.99.
Epoch 18: Average loss is 6664.10, mean absolute error is   62.93.

 After batch 0, the loss is 5555.41.

 After batch 2, the loss is 6342.96.
Epoch 19: Average loss is 5810.87, mean absolute error is   58.25.
```

We can also use our callback in the `evaluate` function...

```
In [9]: # Evaluate the model

        model_eval = model.evaluate(test_data, test_targets, batch_size=10,
                                    callbacks=[LossAndMetricCallback()], verbose=False)
```

```
 After batch 0, the loss is 16673.69.

 After batch 1, the loss is 17845.54.

 After batch 2, the loss is 23011.87.

 After batch 3, the loss is 17906.10.

 After batch 4, the loss is 22836.11.
```

...And also the `predict` function.

```
In [10]: # Get predictions from the model

         model_pred = model.predict(test_data, batch_size=10,
                                    callbacks=[LossAndMetricCallback()], verbose=False)
```

```
Finished prediction on batch 0!
Finished prediction on batch 1!
Finished prediction on batch 2!
Finished prediction on batch 3!
Finished prediction on batch 4!
```

1.0.2 Application - learning rate scheduler

Let's now look at a more sophisticated custom callback.

We are going to define a callback to change the learning rate of the optimiser of a model during training. We will do this by specifying the epochs and new learning rates where we would like it to be changed.

First we define the auxillary function that returns the learning rate for each epoch based on our schedule.

```
In [11]: # Define the learning rate schedule. The tuples below are (start_epoch, new_learning_

         lr_schedule = [
             (4, 0.03), (7, 0.02), (11, 0.005), (15, 0.007)
         ]

         def get_new_epoch_lr(epoch, lr):
             # Checks to see if the input epoch is listed in the learning rate schedule
             # and if so, returns index in lr_schedule
             epoch_in_sched = [i for i in range(len(lr_schedule)) if lr_schedule[i][0]==int(epo
             if len(epoch_in_sched)>0:
                 # If it is, return the learning rate corresponding to the epoch
                 return lr_schedule[epoch_in_sched[0]][1]
             else:
                 # Otherwise, return the existing learning rate
                 return lr
```

Let's now define the callback itself.

```
In [12]: # Define the custom callback

         class LRScheduler(tf.keras.callbacks.Callback):

             def __init__(self, new_lr):
                 super(LRScheduler, self).__init__()
                 # Add the new learning rate function to our callback
                 self.new_lr = new_lr

             def on_epoch_begin(self, epoch, logs=None):
                 # Make sure that the optimizer we have chosen has a learning rate, and raise
                 if not hasattr(self.model.optimizer, 'lr'):
                       raise ValueError('Error: Optimizer does not have a learning rate.')

                 # Get the current learning rate
                 curr_rate = float(tf.keras.backend.get_value(self.model.optimizer.lr))

                 # Call the auxillary function to get the scheduled learning rate for the curr

                 scheduled_rate = self.new_lr(epoch, curr_rate)

                 # Set the learning rate to the scheduled learning rate
                 tf.keras.backend.set_value(self.model.optimizer.lr, scheduled_rate)
                 print('Learning rate for epoch {} is {:7.3f}'.format(epoch, scheduled_rate))
```

Let's now train the model again with our new callback.

```
In [13]: # Build the same model as before
```

```
        new_model = tf.keras.Sequential([
            Dense(128, activation='relu', input_shape=(train_data.shape[1],)),
            Dense(64,activation='relu'),
            tf.keras.layers.BatchNormalization(),
            Dense(64, activation='relu'),
            Dense(64, activation='relu'),
            Dense(1)
        ])
```

```
In [14]: # Compile the model

         new_model.compile(loss='mse',
                         optimizer="adam",
                         metrics=['mae', 'mse'])
```

```
In [15]: # Fit the model with our learning rate scheduler callback

         new_history = new_model.fit(train_data, train_targets, epochs=20,
                                     batch_size=100, callbacks=[LRScheduler(get_new_epoch_lr)]
```

```
Learning rate for epoch 0 is   0.001
Learning rate for epoch 1 is   0.001
Learning rate for epoch 2 is   0.001
Learning rate for epoch 3 is   0.001
Learning rate for epoch 4 is   0.030
Learning rate for epoch 5 is   0.030
Learning rate for epoch 6 is   0.030
Learning rate for epoch 7 is   0.020
Learning rate for epoch 8 is   0.020
Learning rate for epoch 9 is   0.020
Learning rate for epoch 10 is   0.020
Learning rate for epoch 11 is   0.005
Learning rate for epoch 12 is   0.005
Learning rate for epoch 13 is   0.005
Learning rate for epoch 14 is   0.005
Learning rate for epoch 15 is   0.007
Learning rate for epoch 16 is   0.007
Learning rate for epoch 17 is   0.007
Learning rate for epoch 18 is   0.007
Learning rate for epoch 19 is   0.007
```

1.0.3 Further reading and resources

- https://www.tensorflow.org/guide/keras/custom_callback
- https://www.tensorflow.org/api_docs/python/tf/keras/callbacks/Callback