Breaking Down the HDF5 File Structure. Keras' (and Tensorflows) Native Way to Save a Trained Model.

- **Who**: J. Skyler Sampayan
- **Where**: LCP NRL-DC
- **When**: June 27, 2024

Preamble

Note: If you do not fully undersand the structure/architecture of a simple neural net **NN**, please refer to this link --> Neural Net Basics.

To start, once you train a model using **keras** you can save it. Keras saves the model into a file with the extension '**.h5**'. This preserves the architecture but also the parameters. The syntax to save the model (assuming **model** is your trained model):

```
model.save('path/to/model.h5')
```

An **HDF5** file cannot be read traditionally like a text file. It must be loaded and read through keras because of the fact that is preserves the architecture and parameters. Additionally, the main language in this tutorial is python. To load your **HDF5** file, follow the steps below.

1. Make sure to import `tensorflow` and `os`
2. Import the **HDF5** file using this command:

   ```
   from keras.models import load_model

   model = load_model('path/to/model.h5')
   ```

   or

   ```
   model = keras.models.load_model('path/to/model.h5')
   ```

3. Read the **HDF5** file and summarize it.

   ```
   model.summary()
   ```

Table Breakdown

Down below is something similar to what you should get. This example uses 6 total layers with 3 hidden layer, an output layer, and 2 dropout layers.

###

Model: "sequential_1"

Layer (type)	Output Shape	Param #

Layer (type)	Output Shape	Param #
dense_4 (Dense)	(None, 32)	128
dense_5 (Dense)	(None, 32)	1,056
dropout_1 (Dropout)	(None, 32)	0
dense_6 (Dense)	(None, 16)	528
dropout_2 (Dropout)	(None, 16)	0
dense_7 (Dense)	(None, 5)	85

Total params: 1,797 (7.02 KB)
Trainable params: 1,797 (7.02 KB)
Non-trainable params: 0 (0.00 B)

##

1. Model Name:

   The first thing we look at is the model name denoted by **Model**:. We see that if no name was specificed when saving a NN into a **HDF5** file, keras will denote it by a default name. This case **sequential_1**.

2. Layer & Type:

   Each layer has name. If no name, keras will give it a default name. The **type** tells you the type of layer which is also one of main attritubes that infer what type of architecture we are dealing with if it is not immediately clear. Down below is a condensed list of famous types typically used.

   - **Core Layers**
     - **Dense**: A fully connected layer.
     - **Activation**: Applies an activation function.
     - **Dropout**: Applies dropout to the input.
     - **Flatten**: Flattens the input.
     - **Reshape**: Reshapes the input.
   - **Convolutional Layers**
     - **Conv1D**: 1D convolution layer (e.g., for temporal data).
   - **Pooling Layers**
     - **AveragePooling1D**: 1D average pooling layer.
     - **MaxPooling1D**: 1D max pooling layer.
     - **GlobalMaxPooling1D**: Global max pooling operation for temporal data.
   - **Recurrent Layers**
     - **LSTM**: Long Short-Term Memory layer.
     - **GRU**: Gated Recurrent Unit layer.

        - **SimpleRNN**: Simple Recurrent Neural Network layer.
    - **Embedding Layers**
        - **Embedding**: Embedding layer for turning positive integers (indexes) into dense vectors of fixed size.
    - **Normalization Layers**
        - **BatchNormalization**: Applies batch normalization.
        - **LayerNormalization**: Applies layer normalization.

3. Output and Shape:

    In the second column, we see two different names and values. **Output** is the batch size which is typically **None**. This dimension is left as **None** because the batch size can vary and is specified at runtime when you train or evaluate the model. The remaining dimensions specify the **shape** of the output tensor for each layer, or how many outputs we expect from that layer. If it is a hidden layer, then we shall expect that many inputs for the next layer.

    If a **Dense** layer has **64** units, its output shape will be **(None, 64)**, indicating that each batch of input data will be transformed into a batch of outputs, each with 64 features.

    It is crucial that we denote this variable as **O**.

4. Param (#):

    Param # stands for parameter number which is denoted as **P**. This tells us the total parameter features in that layer. The equation that governs **P** for a **dense** (fully connected) layer is:

    $$ P = I \cdot O + O $$

    Where **I** is the number of input features (inputs) in that layer.

5. Summary Totals:

    The **Total Params** is just the sum of the total parameters for every layer in the entire NN. The **Trainable Params** is the subset of total parameters that will be updated during training. Both should also lists the space allocated.

Logic Breakdown

If we look at the first two layers of the NN, also shown below,

###

Model: "sequential_1"

Layer (type)	Output Shape	Param #
dense_4 (Dense)	(None, 32)	128

Layer (type)	Output Shape	Param #
dense_5 (Dense)	(None, 32)	1,056
dropout_1 (Dropout)	(None, 32)	0
dense_6 (Dense)	(None, 16)	528
dropout_2 (Dropout)	(None, 16)	0
dense_7 (Dense)	(None, 5)	85

Total params: 1,797 (7.02 KB)
Trainable params: 1,797 (7.02 KB)
Non-trainable params: 0 (0.00 B)

##

we are interested in how the **Param #** affects the shape and how that gives us insight about the structure. In **dense_4** we see we have 32 nodes in that layer and 128 parameters that dictate that layer. Going back to our governing equation of **P** we can see how that affects the next layer and so on.

$$ P = I \cdot O + O $$

$$128 = (I \cdot 32) + 32$$

$$I = 3$$

Solving for **I** we get 3, which means that this layers takes 3 inputs. We can do the same for **dense_5**, where we expect the input to be 32 because of the amount of outputs or nodes from the previous layer.

$$ 1056 = (I \cdot 32) + 32 $$

$$ I = 32$$

One things we notice about the governing equation for **P** is that it is just the preactivation function for that layer. Where the preactivation function is different for different architectures. The preactivation function for a **dense** (fully connected) layer is:

$$ Z = W \cdot x + B $$

Where **Z** is the preactivation output, **W** is weights, **B** is bias, and **x** is the input, and where each term is a scalar quantity.

Now, however, if we are not dealing with a dense simple NN and instead are dealing with a convolutional NN, we would have more than just one value in the **shape** column and thus our preactivation function also changes. We could get something like this:

##

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 62, 62, 32)	896

##

Where everything before the last value represents the dimension of the filter. Thus the preactivation function would be:

$$ Z = Conv(W,x) + B$$

where

$$Conv(W,x) = (w \times h \times N) * x$$

Where **w** is the width of the filter, **h** is the height of the filter, and **N** is the total number of filters being used. If that is the case where we end up having a different NN architecture, the equations are different but the tracing and logic are the same. Thus we can just adapt the implementation to match the equation necessary.

Necessary Add-Ons

What we do not see in the `model.summary()` method call is the **postactivation** functions used per layer. For that we need the following:

> **Here is the syntax to print out layers activation function:**
>
> ```
> for layer in model.layers:
>    activation = layer.activation.__name__ if hasattr(layer, 'activation')
> else 'None'
>    print(f"Layer: {layer.name}, Type: {layer.__class__.__name__},
> Activation: {activation}")
> ```

where the array size of the ouput is the same as the number of layers. From that we shall expect to see **ReLu, Leaky ReLu, Sigmoid, Tanh, Softmax**, etc.

> **An example output of the same NN is shown below:**
>
> ```
> Layer: dense_4, Activation Function: sigmoid
> Layer: dense_5, Activation Function: sigmoid
> Layer: dropout_1, Activation Function: None
> Layer: dense_6, Activation Function: {'module': 'keras.layers',
> 'class_name': 'LeakyReLU', 'config': {'name': 'leaky_re_lu_1', 'trainable':
> True, 'dtype': 'float32', 'negative_slope': 0.05000000074505806},
> >'registered_name': None}
> Layer: dropout_2, Activation Function: None
> Layer: dense_7, Activation Function: elu
> ```

Extra Commands of Interest:

- Configures the model for training by specifying the optimizer, loss function, and metrics to monitor.

  `model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])`

- Trains the model for a fixed number of epochs (iterations over a dataset).

  ```
  history = model.fit(X_train, y_train, epochs=10, batch_size=32,
  validation_split=0.2)
  ```

- Evaluates the model on a given dataset and returns the loss value and metrics values.

  ```
  loss, accuracy = model.evaluate(X_test, y_test)
  ```

- Generates output predictions for the input samples.

  ```
  predictions = model.predict(X_new)
  ```

- Returns a list of all weight tensors in the model.

  ```
  weights = model.get_weights()
  ```

- Returns a list of all layers in the model.

  ```
  layers = model.layers
  for layer in layers:
    print(layer.name, layer.output_shape)
  ```