Intro to Keras Tuner

Download and prepare the dataset

```
# Import keras
from tensorflow import keras
```

```
# Download the dataset and split into train and test sets
(img_train, label_train), (img_test, label_test) = keras.datasets.fashion_mnist.load_data()
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/train-labels-idx1-ubyte.gz
29515/29515 [==============================] - 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/train-images-idx3-ubyte.gz
26421880/26421880 [==============================] - 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/t10k-labels-idx1-ubyte.gz
5148/5148 [==============================] - 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/t10k-images-idx3-ubyte.gz
4422102/4422102 [==============================] - 0s 0us/step
```

Normalize the pixel values to make the training converge faster.

```
# Normalize pixel values between 0 and 1
img_train = img_train.astype('float32') / 255.0
img_test = img_test.astype('float32') / 255.0
```

Baseline Performance

```
# Build the baseline model using the Sequential API
b_model = keras.Sequential()
b_model.add(keras.layers.Flatten(input_shape=(28, 28)))
b_model.add(keras.layers.Dense(units=512, activation='relu', name='dense_1')) # Will tune this layer later
b_model.add(keras.layers.Dropout(0.2))
b_model.add(keras.layers.Dense(10, activation='softmax'))

# Print model summary
b_model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 flatten (Flatten)           (None, 784)               0         
                                                                 
 dense_1 (Dense)             (None, 512)               401920    
                                                                 
 dropout (Dropout)           (None, 512)               0         
                                                                 
 dense (Dense)               (None, 10)                5130      
                                                                 
=================================================================
Total params: 407,050
Trainable params: 407,050
Non-trainable params: 0
_________________________________________________________________
```

```
# Setup the training parameters
b_model.compile(
    optimizer=keras.optimizers.Adam(learning_rate=0.001), # Will tune learning rate later
    loss=keras.losses.SparseCategoricalCrossentropy(),
    metrics=['accuracy']
)
```

```
# Number of training epochs.
NUM_EPOCHS = 10

# Train the model
b_model.fit(img_train, label_train, epochs=NUM_EPOCHS, validation_split=0.2)
```

```
Epoch 1/10
1500/1500 [==============================] - 11s 3ms/step - loss: 0.5178 - accuracy: 0.8153 - val_loss: 0.3951 - val_accuracy: 0.8577
Epoch 2/10
1500/1500 [==============================] - 5s 3ms/step - loss: 0.3941 - accuracy: 0.8560 - val_loss: 0.4075 - val_accuracy: 0.8523
Epoch 3/10
1500/1500 [==============================] - 4s 3ms/step - loss: 0.3571 - accuracy: 0.8692 - val_loss: 0.3549 - val_accuracy: 0.8746
Epoch 4/10
1500/1500 [==============================] - 4s 3ms/step - loss: 0.3370 - accuracy: 0.8751 - val_loss: 0.3762 - val_accuracy: 0.8648
Epoch 5/10
1500/1500 [==============================] - 5s 3ms/step - loss: 0.3207 - accuracy: 0.8815 - val_loss: 0.3192 - val_accuracy: 0.8832
Epoch 6/10
```

```
1500/1500 [==============================] - 4s 3ms/step - loss: 0.3031 - accuracy: 0.8865 - val_loss: 0.3297 - val_accuracy: 0.8808
Epoch 7/10
1500/1500 [==============================] - 5s 3ms/step - loss: 0.2899 - accuracy: 0.8925 - val_loss: 0.3255 - val_accuracy: 0.8837
Epoch 8/10
1500/1500 [==============================] - 4s 3ms/step - loss: 0.2836 - accuracy: 0.8931 - val_loss: 0.3130 - val_accuracy: 0.8876
Epoch 9/10
1500/1500 [==============================] - 4s 3ms/step - loss: 0.2724 - accuracy: 0.8978 - val_loss: 0.3191 - val_accuracy: 0.8877
Epoch 10/10
1500/1500 [==============================] - 5s 3ms/step - loss: 0.2635 - accuracy: 0.9025 - val_loss: 0.3246 - val_accuracy: 0.8884
<keras.callbacks.History at 0x7f46c039c700>
```

```
# Evaluate model on the test set
b_eval_dict = b_model.evaluate(img_test, label_test, return_dict=True)
```

```
313/313 [==============================] - 1s 3ms/step - loss: 0.3540 - accuracy: 0.8790
```

```
# Define helper function
def print_results(model, model_name, layer_name, eval_dict):
    '''
    Prints the values of the hyparameters to tune, and the results of model evaluation

    Args:
        model (Model) - Keras model to evaluate
        model_name (string) - arbitrary string to be used in identifying the model
        layer_name (string) - name of the layer to tune
        eval_dict (dict) -  results of model.evaluate
    '''
    print(f'\n{model_name}:')

    print(f'number of units in 1st Dense layer: {model.get_layer(layer_name).units}')
    print(f'learning rate for the optimizer: {model.optimizer.lr.numpy()}')

    for key,value in eval_dict.items():
        print(f'{key}: {value}')

# Print results for baseline model
print_results(b_model, 'BASELINE MODEL', 'dense_1', b_eval_dict)
```

```
BASELINE MODEL:
number of units in 1st Dense layer: 512
learning rate for the optimizer: 0.0010000000474974513
loss: 0.3540470600128174
accuracy: 0.8790000081062317
```

Keras Tuner

To perform hypertuning with Keras Tuner, need to:

- Define the model
- Select which hyperparameters to tune
- Define the search space
- Define the search strategy

Install and import packages

```
# Install Keras Tuner
!pip install -q -U keras-tuner
```

```
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 167.3/167.3 KB 5.3 MB/s eta 0:00:00
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 1.6/1.6 MB 34.2 MB/s eta 0:00:00
```

```
# Import required packages
import tensorflow as tf
import keras_tuner as kt
```

Define the model

The model for hypertuning is called a *hypermodel*. Need to define the hyperparameter search space in addition to the model architecture.

Two approaches to define a hypermodel:

- By using a model builder function
- By subclassing the HyperModel class of the Keras Tuner API

In below we use the first approach: Use a model builder function to define the image classification model. This function returns a compiled model and uses hyperparameters defined inline to hypertune the model.

Two hyperparameters that are setup for tuning:

- the number of hidden units of the first Dense layer
- the learning rate of the Adam optimizer

HyperParameters object configures the hyperparameter:

- use `Int()` to define the search space for the Dense units
- use `Choice()` for the learning rate

```
def model_builder(hp):
    '''
    Builds the model and sets up the hyperparameters to tune.

    Args:
        hp - Keras tuner object

    Returns:
        model with hyperparameters to tune
    '''

    # Initialize the Sequential API and start stacking the layers
    model = keras.Sequential()
    model.add(keras.layers.Flatten(input_shape=(28, 28)))

    # Tune the number of units in the first Dense layer
    # Choose an optimal value between 32-512
    hp_units = hp.Int('units', min_value=32, max_value=512, step=32)
    model.add(keras.layers.Dense(units=hp_units, activation='relu', name='tuned_dense_1'))

    # Add next layers
    model.add(keras.layers.Dropout(0.2))
    model.add(keras.layers.Dense(10, activation='softmax'))

    # Tune the learning rate for the optimizer
    # Choose an optimal value from 0.01, 0.001, or 0.0001
    hp_learning_rate = hp.Choice('learning_rate', values=[1e-2, 1e-3, 1e-4])

    model.compile(
        optimizer=keras.optimizers.Adam(learning_rate=hp_learning_rate),
        loss=keras.losses.SparseCategoricalCrossentropy(),
        metrics=['accuracy']
    )

    return model
```

Instantiate the Tuner and perform hypertuning

Keras Tuner has four tuners available with built-in strategies - `RandomSearch`, `Hyperband`, `BayesianOptimization`, and `Sklearn`.

Here we use the Hyperband tuner. Similar to sport championship, the algorithm trains a large number of models for a few epochs and carries forward only the top-performing half of models to the next round.

Hyperband determines the number of models to train in a bracket by computing 1 + $\log_{factor}$(`max_epochs`) and rounding it up to the nearest integer.

The `directory` save logs and checkpoints for every trial (model configuration) run during the hyperparameter search. If re-run the hyperparameter search, the Keras Tuner uses the existing state from these logs to resume the search. To disable this behavior, pass an additional `overwrite=True` argument while instantiating the tuner.

```
# Instantiate the tuner
tuner = kt.Hyperband(
    model_builder, # the hypermodel
    objective='val_accuracy',
    max_epochs=10,
    factor=3,
    directory='kt_dir',
    project_name='kt_hyperband'
)
```

```
# Display hypertuning settings
tuner.search_space_summary()
```

```
Search space summary
Default search space size: 2
units (Int)
{'default': None, 'conditions': [], 'min_value': 32, 'max_value': 512, 'step': 32, 'sampling': 'linear'}
learning_rate (Choice)
{'default': 0.01, 'conditions': [], 'values': [0.01, 0.001, 0.0001], 'ordered': True}
```

```
# Pass in an EarlyStopping callback to stop training early when a metric is not improving
stop_early = tf.keras.callbacks.EarlyStopping(monitor='val_loss', patience=5)
```

```
# Perform hypertuning
tuner.search(img_train, label_train, epochs=NUM_EPOCHS, validation_split=0.2, callbacks=[stop_early])
```

```
Trial 30 Complete [00h 00m 52s]
val_accuracy: 0.8844166398048401

Best val_accuracy So Far: 0.8860833048820496
Total elapsed time: 00h 13m 47s
```

```
# Get the optimal hyperparameters from the results
best_hps=tuner.get_best_hyperparameters()[0]

print(f"""
The hyperparameter search is complete. The optimal number of units in the first densely-connected
layer is {best_hps.get('units')} and the optimal learning rate for the optimizer
is {best_hps.get('learning_rate')}.
""")
```

```
The hyperparameter search is complete. The optimal number of units in the first densely-connected
layer is 128 and the optimal learning rate for the optimizer
is 0.001.
```

Build and train the model

Now that you have the best set of hyperparameters, you can rebuild the hypermodel with these values and retrain it.

```
# Build the model with the optimal hyperparameters
h_model = tuner.hypermodel.build(best_hps)
h_model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 flatten_1 (Flatten)         (None, 784)               0

 tuned_dense_1 (Dense)       (None, 128)               100480

 dropout_1 (Dropout)         (None, 128)               0

 dense_1 (Dense)             (None, 10)                1290

=================================================================
Total params: 101,770
Trainable params: 101,770
Non-trainable params: 0
_________________________________________________________________
```

```
# Train the hypertuned model
h_model.fit(img_train, label_train, epochs=NUM_EPOCHS, validation_split=0.2)
```

```
Epoch 1/10
1500/1500 [==============================] - 8s 5ms/step - loss: 0.5503 - accuracy: 0.8057 - val_loss: 0.4311 - val_accuracy: 0.8456
Epoch 2/10
1500/1500 [==============================] - 6s 4ms/step - loss: 0.4137 - accuracy: 0.8504 - val_loss: 0.3772 - val_accuracy: 0.8616
Epoch 3/10
1500/1500 [==============================] - 5s 3ms/step - loss: 0.3746 - accuracy: 0.8639 - val_loss: 0.3658 - val_accuracy: 0.8662
Epoch 4/10
1500/1500 [==============================] - 4s 3ms/step - loss: 0.3550 - accuracy: 0.8690 - val_loss: 0.3441 - val_accuracy: 0.8754
Epoch 5/10
1500/1500 [==============================] - 5s 4ms/step - loss: 0.3378 - accuracy: 0.8756 - val_loss: 0.3414 - val_accuracy: 0.8748
Epoch 6/10
1500/1500 [==============================] - 4s 3ms/step - loss: 0.3240 - accuracy: 0.8788 - val_loss: 0.3239 - val_accuracy: 0.8822
Epoch 7/10
1500/1500 [==============================] - 4s 3ms/step - loss: 0.3112 - accuracy: 0.8850 - val_loss: 0.3214 - val_accuracy: 0.8828
Epoch 8/10
1500/1500 [==============================] - 5s 3ms/step - loss: 0.2985 - accuracy: 0.8889 - val_loss: 0.3265 - val_accuracy: 0.8829
Epoch 9/10
1500/1500 [==============================] - 4s 3ms/step - loss: 0.2935 - accuracy: 0.8904 - val_loss: 0.3443 - val_accuracy: 0.8759
Epoch 10/10
1500/1500 [==============================] - 5s 3ms/step - loss: 0.2850 - accuracy: 0.8931 - val_loss: 0.3262 - val_accuracy: 0.8828
<keras.callbacks.History at 0x7f462a517190>
```

```
# Evaluate the hypertuned model against the test set
h_eval_dict = h_model.evaluate(img_test, label_test, return_dict=True)
```

```
313/313 [==============================] - 1s 2ms/step - loss: 0.3527 - accuracy: 0.8781
```

```
# Print results of the baseline and hypertuned model
print_results(b_model, 'BASELINE MODEL', 'dense_1', b_eval_dict)
print_results(h_model, 'HYPERTUNED MODEL', 'tuned_dense_1', h_eval_dict)
```

```
BASELINE MODEL:
number of units in 1st Dense layer: 512
learning rate for the optimizer: 0.0010000000474974513
loss: 0.3540470600128174
accuracy: 0.8790000081062317
```

```
HYPERTUNED MODEL:
number of units in 1st Dense layer: 128
learning rate for the optimizer: 0.0010000000474974513
loss: 0.3527015745639801
accuracy: 0.8780999779701233
```

0 秒 完成时间：16:22