

keypoint_classifier.ipynb

Thursday, August 10, 2023
3:36 PM

Import libraries and modules

```
import csv

import numpy as np
import tensorflow as tf
from sklearn.model_selection import train_test_split

RANDOM_SEED = 42
```

import csv: This module provides functionality to read and write CSV (Comma-Separated Values) files. It is used for working with tabular data in a plain text format.

import numpy as np: The numpy library is used for numerical computations in Python. It provides support for working with arrays, matrices, and various mathematical operations.

import tensorflow as tf: TensorFlow is an open-source machine learning framework developed by Google. It's widely used for building and training machine learning models, especially neural networks.

from sklearn.model_selection import train_test_split: The train_test_split function from the sklearn.model_selection module is used to split datasets into training and testing subsets.

RANDOM_SEED = 42: This line defines a constant variable RANDOM_SEED with a value of 42. This is often used to seed the random number generators in order to ensure reproducibility of random processes in machine learning experiments.

Path to files

```
dataset = 'model/keypoint_classifier/keypoint.csv'
model_save_path = 'model/keypoint_classifier/keypoint_classifier.hdf5'
```

dataset: This variable holds the path to a CSV file named 'model/keypoint_classifier/keypoint.csv'. This file is expected to contain the dataset used training the keypoint classifier model.

model_save_path: This variable holds the path where the trained keypoint classifier model be saved. The model will be saved in HDF5 format, which is a common format for saving and loading neural network models in TensorFlow.

Number of classes

```
NUM_CLASSES = 3
```

The variable NUM_CLASSES is defined and set to 3. This likely indicates that there are three classes categories in the classification problem being solved. It is related to the number of different hand that your keypoint classifier is trained to recognize.

A delimiter is a sequence of one or more characters for specifying the boundary between separate, independent regions in plain text, mathematical expressions or other data streams. An example of a delimiter is the comma character, which acts as a field delimiter in a sequence of comma-separated values.

Load txt from 1 to 41 columns

```
X_dataset = np.loadtxt(dataset, delimiter=',', dtype='float32', usecols=list(range(1, (21 * 2) + 1)))
```

- **dataset:** This variable contains the path to the CSV file from which the data is loaded.
- **delimiter=',':** This parameter specifies that the comma character ',' is used as the delimiter between values in the CSV file.
- **dtype='float32':** This parameter indicates that the data should be loaded as floating-point numbers with 32-bit precision.
- **usecols=list(range(1, (21 * 2) + 1)):** This parameter specifies the columns to be loaded from the CSV file. It uses the `list(range(1, (21 * 2) + 1))` expression to generate a list of column indices from 1 to 42 (inclusive). These indices correspond to the x and y coordinates of the keypoints for each of the 21 landmarks (each landmark has 2 coordinates).
So, the **X_dataset** will contain the x and y coordinates of the keypoints for each instance in the dataset, with each instance having 42 values (21 landmarks * 2 coordinates).

A **delimiter** is a character or sequence of characters used to separate and distinguish items within a larger sequence of data.

For example, in a CSV (Comma-Separated Values) file, commas are used as delimiters to separate individual values within each row.

Load first Column(label number) from csv file

```
y_dataset = np.loadtxt(dataset, delimiter=',', dtype='int32', usecols=(0))
```

- **dataset:** This variable contains the path to the CSV file from which the data is loaded.
- **delimiter=',':** This parameter specifies that the comma character ',' is used as the delimiter between values in the CSV file.
- **dtype='int32':** This parameter indicates that the data should be loaded as integers with 32-bit precision.
- **usecols=(0):** This parameter specifies that only the first column (column with index 0) should be loaded as labels.

As a result, the **y_dataset** will contain the keypoints label number corresponding to each instance in the dataset, where each label is an integer value loaded from the first column of CSV file.

Train test split

```
X_train, X_test, y_train, y_test = train_test_split(X_dataset, y_dataset, train_size=0.75, random_state=RANDOM_SEED)
```

- **X_dataset:** The feature data (input data) that you want to split.

- **y_dataset:** The corresponding labels for the feature data.
- **train_size=0.75:** This parameter specifies the proportion of the dataset to be used for training. In this case, 75% of the data is used for training, and the remaining 25% will be used for testing.
- **random_state=RANDOM_SEED:** This parameter sets the random seed for reproducibility. It ensures that the same random split will occur every time you run the code, which is useful for consistent testing and validation.

After calling `train_test_split`, the function returns four sets of data:

- **X_train:** The training feature data.
- **X_test:** The testing feature data.
- **y_train:** The corresponding training labels.
- **y_test:** The corresponding testing labels.

Define a neural network model using the TensorFlow Keras library

TensorFlow is an open-source **machine learning framework** developed by the **Google Brain team**. It provides a flexible and comprehensive ecosystem for building various machine learning models, especially deep learning models. TensorFlow allows you to **define, train, and deploy machine learning models** for a wide range of tasks.

"**deploy**" refers to the process of taking a trained machine learning model and making it available for use in real-world applications.

Keras is an open-source **high-level neural networks API** written in Python. It was originally developed as an independent project by François Chollet and later became **part of the TensorFlow project**. Keras offers a user-friendly interface for building and training neural networks **without requiring in-depth knowledge** of the underlying mathematical and computational details.

```
model = tf.keras.models.Sequential([
    tf.keras.layers.Input((21 * 2, )),
    tf.keras.layers.Dropout(0.2),
    tf.keras.layers.Dense(20, activation='relu'),
    tf.keras.layers.Dropout(0.4),
    tf.keras.layers.Dense(10, activation='relu'),
    tf.keras.layers.Dense(NUM_CLASSES, activation='softmax')
])
```

• Model Initialization

```
model = tf.keras.models.Sequential([...])
```

This line initializes a Sequential model. A Sequential model is a linear stack of layers, where can simply add layers one after another.

• Input Layer

```
tf.keras.layers.Input((21 * 2, ))
```

This layer defines the input shape of the model. The input has a shape of (21 * 2,), which is a flattened array of length 42 (since there are 21 keypoint coordinates, each with x and y

- **Dropout Layer**

tf.keras.layers.Dropout(0.2)

This layer applies dropout regularization to the input data. It randomly sets a fraction of units to 0 during training, which helps prevent overfitting.

In machine learning and deep learning, **overfitting** is a common problem that occurs when a learns to perform very well on the training data but **fails to generalize effectively to new, unseen data**. **Dropout regularization** is a technique used to mitigate (make less severe) overfitting by preventing complex co-adaptations of neurons during training.

The **tf.keras.layers.Dropout** layer is a part of the TensorFlow library's Keras API, which provides an easy way to create and configure various layers in a neural network. Here's how the Dropout layer works:

- **Input Data and Probability:** When you create a Dropout layer, you provide a parameter which is the dropout rate, representing the **fraction of input units (neurons) to randomly set to 0** during each training batch. In your example, you've used `tf.keras.layers.Dropout(0.2)`, where the **dropout rate is set to 0.2** or 20%.
- **Training Phase:** During the training phase, when forward and backward passes are performed to update the model's weights, the Dropout layer randomly masks a portion of the input units by setting them to 0 according to the specified dropout rate. This means that during each training iteration, **a different set of neurons are set to 0, introducing some level of randomness**.
- **Testing Phase:** During the testing or inference phase, **dropout is not applied**. The full network is used to make predictions, which is why dropout is only applied during training. This is because **dropout introduces randomness that helps the network generalize better during training** but is not needed during testing.
- **Regularization Effect:** By randomly setting a fraction of neurons to 0 during training, dropout prevents any **single neuron from becoming overly specialized or dependent** on a specific set of other neurons. This encourages the network to learn more robust and generalizable features that can lead to better performance on unseen data.
- **Effective Ensemble:** Dropout can also be thought of as a form of model ensemble. During training, the network is effectively training multiple sub-networks with different subsets of neurons active. At test time, these sub-networks are combined in a way that each neuron's contribution is scaled by the dropout probability, effectively approximating an ensemble of networks.

Ensemble methods in machine learning involve combining multiple individual models to create a stronger, more accurate predictive model.

An **overfitted model** performs extremely well on the training data but fails to generalize effectively to new, unseen data.

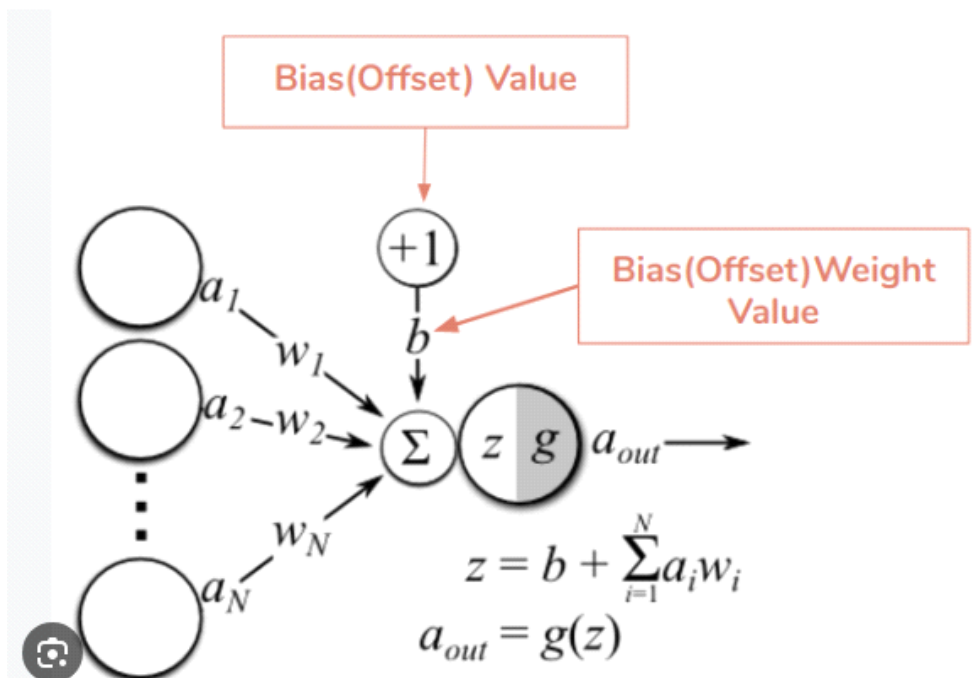
To summarize, the **tf.keras.layers.Dropout** layer helps reduce overfitting by introducing noise and randomness during training, which prevents the network from relying too heavily on specific neurons and encourages more generalized learning.

- **Dense Layer**

```
tf.keras.layers.Dense(20, activation='relu')
```

This is a fully connected (dense) layer with 20 units and ReLU (Rectified Linear Unit) function. The layer takes the output from the previous layer, applies matrix multiplication weights, and then applies the ReLU activation function element-wise.

- **Number of Neurons (Units):** The first parameter, 20, specifies the number of neurons (or units) in the dense layer. Each neuron in this layer is responsible for learning and representing different patterns in the input data. The number of units is a design choice and can be adjusted based on the complexity of the problem and the network architecture. **Each neuron in the Dense layer takes inputs from all the neurons in the previous layer (or the raw input data if it's one of the initial layers), performs a weighted sum of these inputs along with a bias term, and then applies an activation function to produce an output.**
- **Activation Function:** The activation parameter is set to 'relu', which stands for **Rectified Linear Activation**. **Activation functions** introduce **non-linearity** into the neural network, allowing it to model complex relationships between inputs and outputs. The Rectified Linear Unit (ReLU) activation function is one of the most commonly used activation functions in deep learning. It's defined as $f(x) = \max(0, x)$, which means that it outputs the input value if it's positive, and 0 otherwise. ReLU helps the network learn quickly and avoids the vanishing gradient problem.
The choice of activation function can impact the learning dynamics and the network's ability to capture certain types of patterns in the data.
- **Input to the Layer:** The input to a Dense layer comes from the previous layer in the neural network. If this Dense layer is one of the initial layers in the network, then the input would be the raw input data. If it's not the first layer, the input comes from the previous layer's output.
- **Weights and Bias:** Each connection between neurons has **an associated weight**, and **each neuron has a bias term**. These weights and biases are learned **during the training process**. The weighted sum of inputs and biases is passed through the activation function to produce the output of the neuron.
- **Output:** The output of the Dense layer is a set of activations produced by the activation function for each neuron. These activations can be interpreted as the representation or features learned by the network from the input data.



Dropout Layer

`tf.keras.layers.Dropout(0.4)`

Another dropout layer is added after the first dense layer for additional regularization.

Dense Layer

`tf.keras.layers.Dense(10, activation='relu')`

This is another fully connected layer with 10 units and ReLU activation.

Dense Layer (Output Layer)

`tf.keras.layers.Dense(NUM_CLASSES, activation='softmax')`

- **Number of Units (Neurons):** The NUM_CLASSES variable represents the number of classes in your classification problem. Each class corresponds to a unique category or label that you want the model to predict. The number of neurons in the output layer is set to **NUM_CLASSES**. This means that there will be one neuron for each class, and the network will produce scores or probabilities for each class.
- **Activation Function - Softmax:** The activation function used for the output layer is softmax. The softmax function is often used in **multi-class classification problems**. It takes a set of raw scores (logits) produced by the previous layers and converts them into a probability distribution over the classes. **Each output neuron's value after the softmax activation represents the probability of the input belonging to the corresponding class.**

Summary of the architecture of your neural network model

```
model.summary() # tf.keras.utils.plot_model(model, show_shapes=True)
```

Model: "sequential"

Layer (type)	Output Shape	Param #
dropout (Dropout)	(None, 42)	0
dense (Dense)	(None, 20)	860
dropout_1 (Dropout)	(None, 20)	0
dense_1 (Dense)	(None, 10)	210
dense_2 (Dense)	(None, 3)	33
Total params: 1,103		
Trainable params: 1,103		
Non-trainable params: 0		

Model Name: "sequential_2" is the name of the model. It's a sequential model, meaning layers are added one after the other.

Layer Summary: This section lists the layers in your model, along with some details about each layer.

- **Layer (type):** This column shows the type of each layer in the model.
- **Output Shape:** This column indicates the shape of the output produced by each layer. example, (None, 42) means the layer produces an output with a batch size of "None" (which is determined during runtime) and a feature dimension of 42.
- **Param #:** This column indicates the number of trainable parameters in each layer. Trainable parameters are the weights and biases that are learned during training to optimize the model's performance.

Total params: This line shows the total number of trainable parameters in the entire model. In this case, there are 1180 trainable parameters.

Trainable params: This line indicates the number of trainable parameters that can be updated during training. The value is the same as the total number of parameters, 1180.

Non-trainable params: This line indicates the number of parameters that are not trainable. In this case, there are no non-trainable parameters.

The architecture seems to consist of:

- **A Dropout layer with 42 output units.**
- **A fully connected (Dense) layer with 20 output units and 860 parameters.**
- **Another Dropout layer.**
- **Another fully connected (Dense) layer with 10 output units and 210 parameters.**
- **A final fully connected (Dense) layer with 10 output units and 110 parameters.**


The number of parameters in a dense (fully connected) layer is determined by the number of input units, the number of output units, and an additional bias term for each output unit.

```
params = (input_units + 1) * output_units
```

``input_units``: The number of units in the previous layer (or the input layer if this is the first layer).

``output_units``: The number of units in the current dense layer.

csharp

 Copy code

```
params = (42 + 1) * 20 = 860
```

Save Checkpoints and stop training

```
# モデルチェックポイントのコールバック  
cp_callback = tf.keras.callbacks.ModelCheckpoint(  
    model_save_path, verbose=1, save_weights_only=False)  
# 早期打ち切り用コールバック  
es_callback = tf.keras.callbacks.EarlyStopping(patience=20, verbose=1)
```

Model Checkpoint Callback: The ModelCheckpoint callback is used to save the model's weights and architecture during training. It's a common practice to save checkpoints periodically to ensure that even if the training process is interrupted, you can still resume from the most recent checkpoint instead of starting from scratch. The callback takes a few parameters:

- **model_save_path:** This is the path where the model checkpoints will be saved.
- **verbose=1:** This parameter controls the verbosity of the progress messages displayed during checkpoint saving.
- **save_weights_only=False:** Setting this parameter to False indicates that both the model's architecture and its weights will be saved.

Early Stopping Callback: The EarlyStopping callback is used to stop training when a certain condition is met. This can help prevent overfitting and save training time if the model's performance on a validation set stops improving. The callback takes a few parameters:

- **patience=20:** This parameter determines how many epochs to wait before stopping the training once the model's performance has stopped improving. In this case, training will be stopped if there's no improvement for 20 consecutive epochs.
- **verbose=1:** Similar to the ModelCheckpoint callback, this parameter controls the verbosity of the progress messages displayed during early stopping.

The **verbosity parameter** is commonly used in various machine learning libraries, including TensorFlow's Keras API. It controls the level of detail of the output messages that are displayed during the training process.

- **0:** Silent mode. No output messages will be displayed during training.
- **1:** Default mode. Progress messages, such as the number of completed epochs and the training loss, will be displayed.
- **2:** Verbose mode. More detailed information, including the progress of individual batches,

will be displayed.

The term "**epochs**" refers to the number of times a machine learning algorithm, such as a neural network, iterates over the entire training dataset during the training process. In each epoch, the algorithm goes through all the training data, updating its internal parameters (weights and biases) in an attempt to minimize the loss function.

Configuring the compilation of a Keras model before training

```
# モデルコンパイル
model.compile(
    optimizer='adam',
    loss='sparse_categorical_crossentropy',
    metrics=['accuracy']
)
```

model.compile(): This method is used to configure the training process of a Keras model. It prepares the model for training by specifying the optimizer, loss function, and metrics to be used during the training process.

optimizer='adam': This line specifies the optimizer to be used for updating the model's weights during training. In this case, the "Adam" optimizer is used. Adam is a popular optimization algorithm that adapts the learning rate over time.

loss='sparse_categorical_crossentropy': This line sets the loss function that the model will optimize during training. The loss function is a measure of how well the model's predictions match the actual target values. The "sparse categorical cross-entropy" loss is often used for multi-class classification problems where the target values are integers representing class labels.

metrics=['accuracy']: This line specifies the evaluation metric(s) that will be monitored during training and validation. The "accuracy" metric is a common choice for classification tasks, as it measures the ratio of correct predictions to the total number of predictions.

Train a Keras model using the fit method

```
model.fit(
    X_train,
    y_train,
    epochs=1000,
    batch_size=128,
    validation_data=(X_test, y_test),
    callbacks=[cp_callback, es_callback]
)
```

X_train: This is your training data, typically a NumPy array or a TensorFlow tensor, containing the input features for training examples.

y_train: This is the corresponding target labels for the training data.

epochs=1000: This specifies the number of training epochs. In this case, the model will be trained for a maximum of 1000 epochs.

batch_size=128: This sets the batch size for each iteration of training. The training data is divided into batches, and the model's weights are updated after processing each batch. A batch size of 128 means that 128 training examples will be processed before updating the model's weights.

validation_data=(X_test, y_test): This specifies the validation data that will be used to

evaluate the model's performance during training. `X_test` contains the input features for the validation set, and `y_test` contains the corresponding target labels.

`callbacks=[cp_callback, es_callback]`: This parameter allows you to pass a list of callback instances to monitor and control the training process. In this case, both the `ModelCheckpoint` and `EarlyStopping` callbacks are included.

Batch processing is a technique used to divide these large datasets into smaller subsets called batches, which are processed sequentially by the model instead of using the entire dataset at once.

SUMMARY

- It will process batches of training data (`batch_size` examples at a time) for the **specified of epochs (epochs)**.
- After each epoch, it will **evaluate its performance on the validation data** provided using the `validation_data` parameter.
- The **`ModelCheckpoint` callback** will save the model's weights and architecture at each epoch, and the **`EarlyStopping` callback** will monitor the validation performance and potentially stop training if no improvement is observed for a certain number of epochs (`patience` parameter).

Evaluating model

```
val_loss, val_acc = model.evaluate(X_test, y_test, batch_size=128)
```

`val_loss` will hold the loss value, and **`val_acc`** will hold the accuracy value that the model achieved on the provided test dataset.

Loading a pre-trained Keras model

```
# 保存したモデルのロード
model = tf.keras.models.load_model(model_save_path)
```

Make prediction on single test sample

```
# 推論テスト
predict_result = model.predict(np.array([X_test[0]]))
print(np.squeeze(predict_result))
print(np.argmax(np.squeeze(predict_result)))
```

```
arduino
1/1 [=====] - 0s 128ms/step
```

This line indicates that a single batch (1/1) has been processed. The batch size is 1, and it took approximately 128 milliseconds to process.

csharp

Copy code

```
[2.5900962e-10 5.1857654e-12 1.0544278e-21 1.6318206e-15 9.9999988e-01
1.5537900e-14 1.5628451e-27 2.7757443e-09 9.5385921e-08 1.3346240e-24]
```

This line shows the prediction result for the input test sample. **Each value in the array corresponds to the model's predicted probability for a specific class.** For example, the predicted probability for class 0 is 2.5900962e-10, the predicted probability for class 1 is 5.1857654e-12, and so on. The predicted probabilities are very small for most classes, except for class 4 (9.9999988e-01), which has a very high predicted probability. This indicates that the model strongly believes the input sample belongs to class 4.

Confusion Matrix

		Actual	
		Dog	Not Dog
Predicted	Dog	True Positive (TP)	False Positive (FP)
	Not Dog	False Negative (FN)	True Negative (TN)

True Positive (TP): It is the total counts having both predicted and actual values are Dog.

True Negative (TN): It is the total counts having both predicted and actual values are Not Dog.

False Positive (FP): It is the total counts having prediction is Dog while actually Not Dog.

False Negative (FN): It is the total counts having prediction is Not Dog while actually, it is Dog.

Save a TensorFlow model to a file

```
# 推論専用のモデルとして保存
model.save(model_save_path, include_optimizer=False)
```

This line of code calls the save method on a TensorFlow model (model). The **model_save_path** should contain the path where you want to save the model. The **include_optimizer** parameter is set to False, indicating that you don't want to include optimizer information in the saved model file. Optimizer information is used during training, and if you're only interested in deploying the model for inference, you can save space by excluding it.

TensorFlow Lite (TFLite) is a lightweight version of the TensorFlow framework designed for mobile embedded devices. TFLite allows you to deploy machine learning models on these devices with efficiency and performance in mind.

Convert a TensorFlow Keras model to TensorFlow Lite format and save it as a .tflite file

```
tflite_save_path = 'model/keypoint_classifier/keypoint_classifier.tflite'

converter = tf.lite.TFLiteConverter.from_keras_model(model)
converter.optimizations = [tf.lite.Optimize.DEFAULT]
tflite_quantized_model = converter.convert()

open(tflite_save_path, 'wb').write(tflite_quantized_model)
```

tflite_save_path = 'model/keypoint_classifier/keypoint_classifier.tflite': This line defines the file path where the converted TensorFlow Lite model will be saved. The path includes the desired directory ('model/keypoint_classifier/') and file name ('keypoint_classifier.tflite').

converter = tf.lite.TFLiteConverter.from_keras_model(model): Here, you create a TFLiteConverter object using the from_keras_model method. This converter is used to convert a TensorFlow Keras model (model) into TensorFlow Lite format.

converter.optimizations = [tf.lite.Optimize.DEFAULT]: This line sets optimization options for the converter. The specified optimization, tf.lite.Optimize.DEFAULT, indicates that default optimization techniques should be applied during conversion. These optimizations aim to reduce the size and improve the performance of the resulting TensorFlow Lite model.

tflite_quantized_model = converter.convert(): This line performs the actual conversion of the Keras model to TensorFlow Lite format. The result is a quantized TensorFlow Lite model stored in the variable tflite_quantized_model.

open(tflite_save_path, 'wb').write(tflite_quantized_model): Here, you open the specified file path in binary write mode ('wb') and write the content of the quantized TensorFlow Lite model (tflite_quantized_model) to the file. This effectively saves the model as a .tflite file at the specified location.

Create a TensorFlow Lite interpreter and allocate memory

```
interpreter = tf.lite.Interpreter(model_path=tflite_save_path)
interpreter.allocate_tensors()
```

interpreter = tf.lite.Interpreter(model_path=tflite_save_path): Here, you create a Lite interpreter by initializing an instance of the tf.lite.Interpreter class. You provide the

`model_path` argument to specify the path to the saved TensorFlow Lite model (.tflite file) you want to load into the interpreter. The interpreter will be used to run inference on the model.
interpreter.allocate_tensors(): After creating the interpreter, you need to allocate memory it to prepare it for inference. This line of code triggers the allocation of memory resources required for input and output tensors of the TensorFlow Lite model.

Output indicates the size of TFLITE Model in bytes

Input and Output Details

```
input_details = interpreter.get_input_details()
output_details = interpreter.get_output_details()
```

The `input_details` and `output_details` will provide you with information such as the data type, shape, and name of the input and output tensors.

Setting the input tensor of the TensorFlow Lite interpreter with the input data for inference

```
interpreter.set_tensor(input_details[0]['index'], np.array([X_test[0]]))
```

`input_details[0]['index']`: This retrieves the index of the first input tensor in the interpreter's input details. Different models may have multiple input tensors, so you need to specify which one you're setting.

`np.array([X_test[0]])`: This creates a NumPy array containing the input data for inference. `X_test[0]` likely represents a single input data point from your test dataset.

`interpreter.set_tensor(input_details[0]['index'], np.array([X_test[0]]))`: This line sets the input tensor of the interpreter with the input data you provided. It's effectively passing the input data to the model for inference.

Retrieves the inference results from the output tensor of the TensorFlow Lite interpreter

```
%%time
# 推論実施
interpreter.invoke()
tflite_results = interpreter.get_tensor(output_details[0]['index'])
```

1. **%%time**: This is a Jupyter Notebook magic command that measures the execution time of the code block following it.
2. **interpreter.invoke()**: This line invokes the TensorFlow Lite interpreter to perform inference using the input data that you previously set using `interpreter.set_tensor()`.
3. **tflite_results = interpreter.get_tensor(output_details[0]['index'])**: This line retrieves the inference results from the output tensor of the TensorFlow Lite interpreter. It uses the `get_tensor()` method and specifies the index of the output tensor using `output_details[0]['index']`.

Inference results obtained from the TensorFlow Lite interpreter.

```
print(np.squeeze(tflite_results))  
print(np.argmax(np.squeeze(tflite_results)))
```