# Documentation for Kivy Application with TensorFlow Lite Model Integration

## Contents



# 1 Introduction

This document describes the design and functionality of a Kivy-based application integrated with a TensorFlow Lite model for image classification tasks. The application is divided into two main components:

- **UI Implementation:** Defined in the `main.py` file.
- **TensorFlow Model Wrapper:** Defined in the `model.py` file.

# 2 File Descriptions

## 2.1 main.py

This file contains the implementation of a Kivy application with a user interface that allows the user to capture images, display information, and interact with a TensorFlow Lite model.

### 2.1.1 Main Components

- **Kivy Widgets:** Utilizes several Kivy widgets such as `Camera`, `Label`, `Button`, and layout containers like `BoxLayout` and `GridLayout`.
- **Image Capture and Display:** Enables the user to capture an image from the camera, save it locally, and display it using the `Image` widget.
- **Model Prediction:** Processes the captured image and uses the TensorFlow Lite model to predict the class of the image.

#### 2.1.2 Key Functions

- `build(self)`: Initializes the layout and widgets for the application.
- `on_press__(self, instance)`: Captures the current frame from the camera, saves it, and either displays the saved image or switches back to the camera feed.
- `on_button_click(self, instance)`: Toggles the camera's play state.
- `__predict(self)`: Processes the saved image and uses the TensorFlow Lite model to predict its class.
- `add_scroll_box(self, instance)`: Displays detailed information about different skin conditions in a scrollable format.
- `return_to_main(self, instance)`: Removes the scrollable information display and returns to the main layout.

#### 2.1.3 Label Mapping

Maps the integer labels predicted by the model to descriptive string labels representing skin conditions.

```
label_mapping = {
    0: 'nv',
    1: 'mel',
    2: 'bkl',
    3: 'bcc',
    4: 'akiec',
    5: 'vasc',
    6: 'df'
}
```

#### 2.1.4 Skin Condition Definitions

Provides descriptions for each label, aiding users in understanding the classification results.

### 2.2 model.py

This file implements a wrapper class for loading and running inferences using a TensorFlow Lite model in a Python environment.

#### 2.2.1 Class: `TensorFlowModel`

- `load(self, model_filename, num_threads=None)`: Loads the TensorFlow Lite model from the specified file.
- `allocate_tensors(self)`: Allocates the necessary tensors for the model, retrieving input and output shapes and types.
- `get_input_shape(self)`: Returns the shape of the model's input tensor.
- `resize_input(self, shape)`: Resizes the model's input tensor to the specified shape and reallocates tensors if necessary.
- `pred(self, x)`: Accepts a NumPy array as input, processes it into a format suitable for the TensorFlow Lite model, runs inference, and returns the model's predictions.

#### 2.2.2 Dependencies

- Java Native Interface (JNI): Uses the `jnius` library to access TensorFlow Lite Java classes.
- NumPy: Processes input data and outputs.

## 3 Execution Flow

1. The user interacts with the application through the UI implemented in `main.py`.
2. The application captures an image using the Kivy `Camera` widget.
3. The image is saved locally and preprocessed for prediction.
4. The preprocessed image is passed to the TensorFlow Lite model for classification.
5. The classification result is displayed on the UI.

## 4 Installation and Setup

1. Install the required Python libraries:

```
pip install kivy jnius numpy
```

2. Ensure that the TensorFlow Lite model file (`model.tflite`) is in the working directory.
3. Run the application:

```
python main.py
```

## 5 Future Enhancements

- Add support for multiple input and output tensors.
- Enhance error handling and logging.
- Implement advanced preprocessing techniques for input images.
- Improve UI/UX with more intuitive interactions.

## 6 Conclusion

This application demonstrates the integration of a TensorFlow Lite model with a Kivy-based user interface for skin condition classification. The modular design ensures that it can be extended and enhanced for more complex tasks in the future.