Deep Learning CNN Model

```
import tensorflow as tf
from tensorflow.keras import layers, models, optimizers, callbacks, applications
import numpy as np
import os
from sklearn.utils import class_weight

# ========================
# 1. CONFIGURATION
# ========================
# Configure GPU memory growth
gpus = tf.config.experimental.list_physical_devices('GPU')
if gpus:
    try:
        for gpu in gpus:
            tf.config.experimental.set_memory_growth(gpu, True)
    except RuntimeError as e:
        print(e)

os.environ['TF_CPP_MIN_LOG_LEVEL'] = '2'
tf.keras.backend.clear_session()

# ========================
# 2. DATA PIPELINE
# ========================
def create_dataset(dataset_path, img_size=(224, 224), batch_size=32):
    """Create optimized data pipeline with RGB conversion"""
    try:
        # Training dataset
        train_ds = tf.keras.utils.image_dataset_from_directory(
            dataset_path,
            validation_split=0.2,
            subset='training',
            seed=123,
            image_size=img_size,
            batch_size=batch_size,
            label_mode='binary',
            color_mode='rgb'
        )

        # Validation dataset
        val_ds = tf.keras.utils.image_dataset_from_directory(
            dataset_path,
            validation_split=0.2,
            subset='validation',
            seed=123,
            image_size=img_size,
            batch_size=batch_size,
            label_mode='binary',
            color_mode='rgb'
        )

        # Normalization and augmentation
        def preprocess(image, label):
            image = tf.cast(image, tf.float32) / 255.0
            return image, label

        augmentation = tf.keras.Sequential([
            layers.RandomFlip("horizontal"),
            layers.RandomRotation(0.1),
        ])

        train_ds = train_ds.map(preprocess).map(
            lambda x, y: (augmentation(x), y),
            num_parallel_calls=tf.data.AUTOTUNE
        )
        val_ds = val_ds.map(preprocess)

        # Optimize pipeline
        train_ds = train_ds.prefetch(tf.data.AUTOTUNE)
        val_ds = val_ds.prefetch(tf.data.AUTOTUNE)

        # Class weights
        labels = np.concatenate([y.numpy() for x, y in train_ds], axis=0)
        class_weights = class_weight.compute_class_weight(
            'balanced',
            classes=np.unique(labels),
            y=labels.flatten()
        )
        class_weights = {i: float(weight) for i, weight in enumerate(class_weights)}

        return train_ds, val_ds, class_weights

    except Exception as e:
        print(f"Dataset creation error: {str(e)}")
        return None, None, None

# ======================
# 3. MODEL ARCHITECTURE
# ======================
def build_model(input_shape=(224, 224, 3)):
    """Build MobileNetV2-based classifier"""
    try:
        base_model = applications.MobileNetV2(
            include_top=False,
            weights='imagenet',
            input_shape=input_shape,
            pooling='avg'
        )
        base_model.trainable = False

        inputs = tf.keras.Input(shape=input_shape)
        x = base_model(inputs)
        x = layers.Dense(128, activation='relu')(x)
        x = layers.Dropout(0.3)(x)
        outputs = layers.Dense(1, activation='sigmoid')(x)

        model = tf.keras.Model(inputs, outputs)
        return model

    except Exception as e:
        print(f"Model building error: {str(e)}")
        return None

# =======================
# 4. TFLITE CONVERSION
# =======================
def convert_to_tflite(model, train_ds, output_path):
    """Convert model to TFLite with quantization"""
    try:
        def representative_dataset():
            for images, _ in train_ds.take(100):
                for img in images:
                    yield [tf.expand_dims(img, axis=0)]

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

        # Try different quantization options
        try:
            converter.target_spec.supported_ops = [tf.lite.OpsSet.TFLITE_BUILTINS_INT8]
            converter.inference_input_type = tf.uint8
            converter.inference_output_type = tf.uint8
            tflite_model = converter.convert()
            print("Created fully quantized uint8 model")
        except Exception as e:
            print(f"uint8 quantization failed, trying float32: {str(e)}")
            converter.inference_input_type = tf.float32
            converter.inference_output_type = tf.float32
            tflite_model = converter.convert()
            print("Created partially quantized float32 model")

        # Save the model
        with open(output_path, 'wb') as f:
            f.write(tflite_model)

        # Verify the converted model
        interpreter = tf.lite.Interpreter(model_content=tflite_model)
        interpreter.allocate_tensors()
        print("TFLite conversion successful!")
        return True

    except Exception as e:
        print(f"TFLite conversion failed: {str(e)}")
        return False

# =======================
# 5. TRAINING PROCESS
# =======================
def train_model(model, train_ds, val_ds, class_weights):
    """Training routine with callbacks"""
    try:
        model.compile(
            optimizer=optimizers.Adam(1e-4),
            loss='binary_crossentropy',
            metrics=['accuracy', tf.keras.metrics.AUC(name='auc')]
        )

        callbacks_list = [
            callbacks.EarlyStopping(
                monitor='val_auc',
                patience=5,
                mode='max',
                restore_best_weights=True),
            callbacks.ModelCheckpoint(
                'best_model.h5',
                monitor='val_auc',
                save_best_only=True,
                mode='max'),
            callbacks.ReduceLROnPlateau(
                monitor='val_loss',
                factor=0.5,
                patience=2,
                verbose=1)
        ]

        print("\n=== TRAINING ===")
        history = model.fit(
            train_ds,
            validation_data=val_ds,
            epochs=30,
            class_weight=class_weights,
            callbacks=callbacks_list,
            verbose=1
        )

        return model, history
    except Exception as e:
        print(f"Training error: {str(e)}")
        return None, None

# ========================
# 6. MAIN EXECUTION
# ========================
def main():
    try:
        print("Starting breast cancer classification training...")

        # Configuration
        dataset_path = r'D:\Dataset\CompleteDataset\Train'
        img_size = (224, 224)
        batch_size = 32

        # Create dataset
        train_ds, val_ds, class_weights = create_dataset(
            dataset_path,
            img_size=img_size,
            batch_size=batch_size
        )

        if train_ds is None:
            raise ValueError("Failed to create dataset pipeline")

        # Build model
        model = build_model(input_shape=(*img_size, 3))
        if model is None:
            raise ValueError("Failed to build model")

        model.summary()

        # Train model
        trained_model, history = train_model(model, train_ds, val_ds, class_weights)
        if trained_model is None:
            raise ValueError("Training failed")

        # Save Keras model
        trained_model.save('breast_cancer_model.h5')
        print("\nKeras model saved successfully!")

        # Convert to TFLite
        print("\nStarting TFLite conversion...")
        tflite_success = convert_to_tflite(
            model=trained_model,
            train_ds=train_ds,
            output_path='breast_cancer_quant.tflite'
        )
        # Evaluation
        print("\nFinal Evaluation:")
        results = trained_model.evaluate(val_ds, verbose=1)
        print(f"Validation Accuracy: {results[1]:.4f}")
        print(f"Validation AUC: {results[2]:.4f}")

        return 0 if tflite_success else 1

    except Exception as e:
        print(f"Main execution error: {str(e)}")
        return 1

if __name__ == "__main__":
    import sys
    sys.exit(main())
```