Arquitectura neuronal

La arquitectura neuronal que utilizamos es la CNN (Convolutional Neural Network) que permite principalmente el reconocimiento de imágenes atribuyendo automáticamente a cada imagen proporcionada en la entrada, una etiqueta correspondiente a la clase a la que pertenece. Esta arquitectura es ideal para el reconocimiento facial porque es capaz de detectar patrones espaciales complejos en el rostro, como la posición de los ojos, la forma de la boca o algún gesto para identificar emociones. Además, aprende estas características directamente de los datos sin necesidad de intervención manual, lo que la hace precisa, eficiente y robusta ante variaciones en iluminación, ángulo o expresión.

Parámetros de entrenamiento

- **Tamaño de imagen (IMG_SIZE): 224x224:** Se redimensionan todas las imágenes a este tamaño para ser compatibles con MobileNetV2.
- **Batch size (BATCH_SIZE): 16:** Tamaño de lote usado para entrenamiento y validación.
- **Épocas (EPOCHS): 20:** Número máximo de épocas para entrenar: Se usa EarlyStopping para detener el entrenamiento si no mejora.
- **Learning rate inicial (INIT_LR): 1e-4:** Tasa de aprendizaje inicial para el optimizador Adam.
- **Regularización L2 (WEIGHT_DECAY): 1e-4:** Aplicada a la capa densa para evitar overfitting.
- **Etiquetas de emociones (emotion_labels):**
  ['angry', 'disgust', 'happy', 'natural', 'sad', 'surprise']: Se espera que el modelo clasifique una imagen en una de estas seis clases. (Al final tuvimos que modificar el dataset, porque estaba mal etiquetado)

```
def augment_image(image, label):
    image = tf.image.random_flip_left_right(image)
    image = tf.image.random_brightness(image, 0.08)
    return image, label
```

Aquí aplicamos un poco de preprocesamiento al dataset, dándole cambios de luz y rotaciones a las imágenes

```python
def load_dataset(filename, batch_size=BATCH_SIZE, augment=False):
    def parse_tfrecord(example):
        feature_description = {
            "image/encoded": tf.io.FixedLenFeature([], tf.string),
            "image/object/class/label": tf.io.VarLenFeature(tf.int64),
        }
        example = tf.io.parse_single_example(example, feature_description)
        image = tf.image.decode_jpeg(example["image/encoded"], channels=3)
        image = tf.image.resize(image, [IMG_SIZE, IMG_SIZE])
        image = tf.cast(image, tf.float32) / 255.0
        image = (image - [0.485, 0.456, 0.406]) / [0.229, 0.224, 0.225]

        labels_sparse = tf.sparse.to_dense(example["image/object/class/label"])
        label = tf.cond(tf.shape(labels_sparse)[0] > 0,
                        lambda: labels_sparse[0] - 1,  # 0..5
                        lambda: tf.constant(-1, dtype=tf.int64))
        return image, label
```

El método parse_tfrecord, decodifica el archivo .tftrecord que es nuestro dataset, image/encoded es la imagen en bytes y label son las etiquetas de nuestro dataset, apartir de ahí se decodifican las imágenes, para luego redimensionar las imágenes del dataset para que sean 224x224 que es un formato más adecuado al usar MobileNetV2 (imageset).

```python
dataset = tf.data.TFRecordDataset(filename)
dataset = dataset.map(parse_tfrecord, num_parallel_calls=tf.data.AUTOTUNE)
dataset = dataset.filter(lambda _, label: label >= 0)
if augment:
    dataset = dataset.map(augment_image, num_parallel_calls=tf.data.AUTOTUNE)
return dataset.shuffle(1000).batch(batch_size).prefetch(tf.data.AUTOTUNE)
```

En este bloque se crea como tal el dataset ya con las modificaciones, y filtra entradas invalidas, además de mezclar aleatoriamente las imágenes para que no sigan un “orden” y se usa prefetch para acelerar la lectura mientras se entrena el modelo.

```python
base_model = applications.MobileNetV2(
    input_shape=(IMG_SIZE, IMG_SIZE, 3),
    include_top=False,
    weights='imagenet',
    alpha=0.5
)
base_model.trainable = True


inputs = layers.Input(shape=(IMG_SIZE, IMG_SIZE, 3))
x = base_model(inputs, training=True)
x = layers.GlobalAveragePooling2D()(x)
x = layers.Dense(192, activation='relu', kernel_regularizer=regularizers.l2(WEIGHT_DECAY))(x)
x = layers.Dropout(0.4)(x)
outputs = layers.Dense(6, activation='softmax')(x)
model = models.Model(inputs, outputs)

model.compile(
    optimizer=optimizers.Adam(INIT_LR),
    loss='sparse_categorical_crossentropy',
    metrics=['accuracy']
)
model.fit(
    train_ds,
    validation_data=val_ds,
    epochs=EPOCHS,
    callbacks=[callbacks.EarlyStopping(patience=5, restore_best_weights=True)]
)

model.save('final_model.keras')
```

En todo este bloque, es ya como tal la construcción del modelo, se pueden observar las especificaciones del modelo base, como le dimos pesos determinados de imagenet entre otras cosas. También se puede observar como le dimos una capa de 192 neuronas y un dropout de 0.4 para prevenir el overfitting o el sobreajuste. Le decimos como queremos que compile, con el optimizador Adam. Y finalmente el entrenamiento como tal con el dataset de entrenamiento, la cantidad de epochs ya mencionada, la data de validaciones y un earlystoping si el modelo no mejora conforme pasan los epochs, para finalmente se guarde en un archivo .keras.