Documentación de CNN en Español

1. Hoja de Referencia de Funciones de Activación para CNNs

Esta hoja de referencia proporciona un catálogo de funciones de activación comúnmente utilizadas en CNNs. Cada entrada explica:

1. Cómo funciona.
2. Cómo mejora el modelo.
3. Parámetros con descripciones, rangos y consejos.
4. Código de ejemplo que puedes copiar en tu notebook y modificar.

1. ReLU (Unidad Lineal Rectificada)

Cómo funciona: Produce x si x > 0, de lo contrario 0. **Mejora:** Previene el desvanecimiento del gradiente, entrenamiento más rápido. **Parámetros:** ninguno. **Consejo:** Opción predeterminada para la mayoría de capas CNN.

```python
layers.Activation('relu')
# o directamente en la capa:
layers.Conv2D(64, (3,3), activation='relu')
```

2. LeakyReLU

Cómo funciona: Como ReLU, pero permite pequeños valores negativos (x*alpha). **Mejora:** Resuelve el problema de "ReLU murientes" (neuronas atascadas en 0). **Parámetros:**

- alpha: float [0–1], pendiente negativa (ej., 0.01). **Consejo:** Usa si sospechas neuronas muertas con ReLU.

```python
layers.LeakyReLU(alpha=0.01)
```

3. ELU (Unidad Lineal Exponencial)

Cómo funciona: Similar a ReLU pero suaviza valores negativos con curva exponencial. **Mejora:** Convergencia más rápida, evita neuronas muertas. **Parámetros:**

- alpha: float > 0, controla curva para negativos (predeterminado 1.0). **Consejo:** Útil para redes más profundas.

```python
layers.ELU(alpha=1.0)
```

4. SELU (Unidad Lineal Exponencial Escalada)

Cómo funciona: Como ELU pero con efecto auto-normalizante cuando se usa con inicialización apropiada. **Mejora:** Mantiene media y varianza estables. **Parámetros:**

- alpha, scale: fijos para auto-normalización. **Consejo:** Usa solo con inicialización 'lecun_normal' y sin BatchNorm.

```python
layers.Activation('selu')
```

5. Sigmoid

Cómo funciona: Mapea valores al rango (0,1). **Mejora:** Bueno para probabilidades, salidas binarias. **Parámetros:** ninguno. **Consejo:** Evita en capas ocultas (gradientes desvanecientes), usa para clasificación binaria.

```python
layers.Activation('sigmoid')
```

6. Tanh

Cómo funciona: Mapea valores a (-1,1). **Mejora:** Centrado en cero, mejor que sigmoid para capas ocultas. **Parámetros:** ninguno. **Consejo:** Aún sufre de gradientes desvanecientes, usa con moderación.

```python
layers.Activation('tanh')
```

7. Softmax

Cómo funciona: Convierte vector a distribución de probabilidad (suma a 1). **Mejora:** Ideal para salidas de clasificación multiclase. **Parámetros:**

- axis: int, qué eje normalizar (predeterminado -1). **Consejo:** Siempre usa en la última capa para CIFAR-10 (num_classes=10).

```python
layers.Dense(10, activation='softmax')
```

8. Softplus

Cómo funciona: Aproximación suave de ReLU (log(1+exp(x))). **Mejora:** Diferenciable en todas partes, evita corte duro en 0. **Parámetros:** ninguno. **Consejo:** Raramente usado, pero a veces entrenamiento más suave.

```python
layers.Activation('softplus')
```

9. Swish (SiLU)

Cómo funciona: No linealidad suave, x * sigmoid(x). **Mejora:** A menudo supera a ReLU en modelos profundos. **Parámetros:** ninguno. **Consejo:** Prueba en CNNs más profundas, pero más lento.

```python
tf.nn.swish(x)
# o en Keras:
layers.Activation('swish')
```

10. Mish

Cómo funciona: Activación suave, no monotónica: x * tanh(softplus(x)). **Mejora:** Afirma mejor precisión que ReLU/Swish en algunas tareas. **Parámetros:** ninguno. **Consejo:** Experimental, más lento, pero vale la pena probar.

```python
import tensorflow_addons as tfa
layers.Activation(tfa.activations.mish)
```

Consejos para CIFAR-10

- Usa ReLU como predeterminado para capas ocultas.
- LeakyReLU o ELU pueden ayudar si el entrenamiento se estanca.
- Usa Softmax en la capa final Dense(num_classes).
- Sigmoid solo si es tarea de clasificación binaria.
- Prueba Swish/Mish para experimentos, no como línea base.

---

2. Lista de Verificación Mini de Entrenamiento CNN (Keras + CIFAR-10)

✅ Preparación de Datos

- Normalizar imágenes: x / 255.0
- Codificar etiquetas en one-hot
- Dividir en train / val / test
- Añadir aumento (voltear, rotar, desplazar)

✅ Diseño del Modelo

- Conv2D + ReLU + Pool → repetir
- Aumentar filtros gradualmente (32 → 64 → 128)
- Añadir Dropout (0.25–0.5) para regularización
- Usar BatchNormalization para estabilidad
- GlobalAveragePooling2D antes de Dense

✅ Entrenamiento

- Optimizador: Adam (lr=0.001 predeterminado)
- Tamaño de lote: 32 o 64
- Épocas: 20–50 (vigilar sobreajuste)
- Usar callbacks:
  - EarlyStopping
  - ModelCheckpoint
  - ReduceLROnPlateau

✅ Activaciones

- Capas ocultas: ReLU
- Salida: Softmax (num_classes=10)

✅ Monitoreo

- Graficar precisión/pérdida de entrenamiento vs validación
- Si pérdida val ↑ mientras pérdida train ↓ → sobreajuste
- Ajustar dropout o usar más aumento de datos

✅ Evaluación

- Usar model.evaluate() en conjunto de prueba
- Revisar muestras mal clasificadas para aprender debilidades

✅ Guardar y Cargar

- Guardar: model.save("cnn_model.h5")
- Cargar: keras.models.load_model("cnn_model.h5")

🌟 Reglas de Oro para Principiantes

- Empezar simple, añadir complejidad después
- Cambiar UN parámetro a la vez
- Documentar lo que funciona (filtros, lr, dropout, etc.)
- No perseguir 100% de precisión — enfocarse en el proceso de aprendizaje

3. Hoja de Referencia de Capas CNN para CIFAR-10

Esta hoja de referencia proporciona un catálogo de capas comúnmente utilizadas en CNNs. Cada entrada explica:

1. Cómo funciona la capa.
2. Cómo mejora el modelo.
3. Parámetros con descripciones, rangos y consejos.
4. Código de ejemplo que puedes copiar en tu notebook y modificar.

1. Conv2D

Cómo funciona: Aplica filtros de convolución para extraer características locales (bordes, texturas, formas). **Mejora:** Aprende jerarquías espaciales de características; capas más profundas capturan patrones complejos. **Parámetros:**

- filters: int, número de filtros (ej., 32, 64, 128). Más filtros capturan características más ricas pero aumentan el cómputo.
- kernel_size: tupla de 2 enteros (ej., (3,3), (5,5)). Pequeño = detalle, grande = contexto.
- strides: tupla de 2 enteros, tamaño de paso (predeterminado (1,1)). Más grande = más rápido, menos detalle.
- padding: 'valid' (sin relleno) o 'same' (relleno cero para mantener tamaño).
- activation: usualmente 'relu'. **Consejo:** Empezar con kernels (3,3), padding 'same'.

```python
layers.Conv2D(64, (3,3), activation='relu', padding='same')
```

2. MaxPooling2D

Cómo funciona: Submuestrea mapas de características tomando el valor máximo en cada región. **Mejora:** Reduce cómputo y refuerza invarianza espacial. **Parámetros:**

- pool_size: tupla (ej., (2,2), (3,3)).
- strides: paso del pooling (predeterminado = pool_size). **Consejo:** (2,2) es estándar; evita pooling demasiado agresivo temprano.

```python
layers.MaxPooling2D((2,2))
```

3. AveragePooling2D

Cómo funciona: Similar a MaxPooling, pero toma promedio en lugar de máximo. **Mejora:** Mapas de características más suaves, mantiene más información general. **Parámetros:**

- pool_size: tupla (ej., (2,2), (3,3)).
- strides: tamaño de paso. **Consejo:** Usa cuando quieras señales más suaves en lugar de nítidas.

```python
layers.AveragePooling2D((2,2))
```

4. GlobalAveragePooling2D

Cómo funciona: Reduce cada mapa de características a un solo valor (promedio). **Mejora:** Reduce enormemente parámetros, evita sobreajuste, a menudo reemplaza Flatten. **Parámetros:** ninguno. **Consejo:** Usa antes de capas Dense para compacidad.

```python
layers.GlobalAveragePooling2D()
```

5. Flatten

Cómo funciona: Aplana mapas de características 2D en un vector 1D. **Mejora:** Prepara para capas Dense. **Parámetros:** ninguno. **Consejo:** Más parámetros que GlobalAveragePooling2D, riesgo de sobreajuste.

```python
layers.Flatten()
```

6. Dense

Cómo funciona: Capa completamente conectada, combina todas las entradas. **Mejora:** Aprende combinaciones complejas y específicas de la tarea de características. **Parámetros:**

- units: int, número de neuronas (ej., 128, 512).
- activation: 'relu', 'softmax', etc. **Consejo:** Usa 'relu' en capas ocultas, 'softmax' para salida.

```python
layers.Dense(512, activation='relu')
```

7. Dropout

Cómo funciona: Desactiva aleatoriamente neuronas durante el entrenamiento. **Mejora:** Previene sobreajuste, mejora generalización. **Parámetros:**

- rate: float [0.0–1.0], fracción de neuronas eliminadas (ej., 0.25, 0.5). **Consejo:** Usa tasas más altas (0.5) antes de capas Dense finales.

```python
layers.Dropout(0.5)
```

8. BatchNormalization

Cómo funciona: Normaliza activaciones de la capa anterior. **Mejora:** Acelera entrenamiento, estabiliza aprendizaje. **Parámetros:**

- momentum: float [0–1], para promedio móvil (predeterminado 0.99).
- epsilon: float pequeño para evitar división por cero (predeterminado 0.001). **Consejo:** Coloca después de Conv/Dense, antes de activación.

```python
layers.BatchNormalization()
```

9. Conv2DTranspose (Deconvolución)

Cómo funciona: Realiza lo contrario de convolución, usado para sobremuestreo. **Mejora:** Útil para generación de imágenes o cuando necesitas mapas de características más grandes. **Parámetros:** igual que Conv2D. **Consejo:** Para tareas de clasificación, usualmente no necesario; para autoencoders, sí.

```python
layers.Conv2DTranspose(64, (3,3), strides=(2,2), padding='same')
```

10. SeparableConv2D

Cómo funciona: Factoriza convolución en depthwise + pointwise. **Mejora:** Reduce cómputo manteniendo rendimiento. **Parámetros:** igual que Conv2D. **Consejo:** Usa para modelos más ligeros.

```python
layers.SeparableConv2D(64, (3,3), activation='relu', padding='same')
```

11. DepthwiseConv2D

Cómo funciona: Aplica una sola convolución por canal de entrada. **Mejora:** Muy eficiente, usado en arquitecturas tipo MobileNet. **Parámetros:** igual que Conv2D. **Consejo:** Combina con convolución pointwise para eficiencia.

```python
layers.DepthwiseConv2D((3,3), padding='same')
```

12. Activation

Cómo funciona: Aplica transformación no lineal (ej., relu, sigmoid, tanh). **Mejora:** Permite que la red aprenda mapeos complejos. **Parámetros:**

- activation: string o función ('relu', 'sigmoid', 'softmax'). **Consejo:** 'relu' es estándar para ocultas, 'softmax' para salida.

```python
layers.Activation('relu')
```

13. SpatialDropout2D

Cómo funciona: Elimina mapas de características completos en lugar de neuronas aleatorias. **Mejora:** Regularización más efectiva para CNNs. **Parámetros:**

- rate: float [0–1]. **Consejo:** Usa en lugar de Dropout en capas Conv.

```python
layers.SpatialDropout2D(0.3)
```

14. LSTM / GRU (opcional para datos secuenciales)

Cómo funciona: Capas recurrentes que manejan dependencias temporales. **Mejora:** Útil para video o series temporales, menos común para CIFAR. **Parámetros:** units, return_sequences, activation. **Consejo:** Omitir para CNNs básicas.

```python
layers.LSTM(128)
```

Consejos para CIFAR-10

- Empezar simple: Conv2D + MaxPooling + Dense + Dropout.
- Añadir BatchNormalization para estabilizar.
- Usar GlobalAveragePooling2D en lugar de Flatten para reducir parámetros.
- Ajustar filtros gradualmente (32 → 64 → 128).
- Evitar modelos demasiado profundos al principio; las imágenes CIFAR-10 son pequeñas.

---

4. Consejos y Consideraciones para Entrenar CNNs con Keras

Esta guía es para principiantes construyendo y entrenando CNNs (como para CIFAR-10).
Proporciona consejos simples pero técnicos para ayudarte a evitar errores comunes.

1. Preparación de Datos

- Siempre normalizar datos de imagen: dividir valores de píxeles por 255.0 para escalar a [0,1].
- Mezclar datos de entrenamiento para evitar sesgo de orden de aprendizaje.
- Usar codificación one-hot para etiquetas (ej., [0,0,1,0,...] para clase 2).
- Aumento de Datos: aumentar artificialmente variedad del conjunto de datos con rotaciones, volteos, desplazamientos, zooms.

```python
keras.preprocessing.image.ImageDataGenerator(
  rotation_range=15,
  width_shift_range=0.1,
  height_shift_range=0.1,
  horizontal_flip=True
)
```

2. Diseño del Modelo

- Empezar simple: Conv → Pool → Dense → Softmax.
- Aumentar filtros gradualmente (32 → 64 → 128).
- Usar Dropout para evitar sobreajuste.
- BatchNormalization a menudo mejora estabilidad y velocidad.
- Reemplazar Flatten con GlobalAveragePooling2D para reducir parámetros.

3. Elegir Activaciones

- Capas ocultas: 'relu' es estándar.
- Capa de salida: 'softmax' para clasificación multiclase (CIFAR-10).
- Probar LeakyReLU/ELU si el entrenamiento está atascado.

4. Elegir Optimizadores

- Adam: buena opción predeterminada, adapta tasas de aprendizaje automáticamente.
- SGD + momentum: más control, a veces mayor precisión con ajuste.
- RMSprop: funciona bien para CNNs también.
- Siempre monitorear tasa de aprendizaje; muy alta = inestable, muy baja = lento.

5. Proceso de Entrenamiento

- Épocas: 20–50 para CIFAR-10, pero usar EarlyStopping para evitar sobreajuste.
- Tamaño de lote: valores comunes = 32, 64, 128. Lotes más grandes entrenan más rápido pero necesitan más memoria.
- Tasa de aprendizaje: empezar con 0.001 (predeterminado Adam). Usar LearningRateScheduler o ReduceLROnPlateau.
- Siempre dividir datos en conjuntos de entrenamiento y validación.

6. Monitorear Entrenamiento

- Observar curvas de precisión/pérdida de entrenamiento vs validación.
- Si pérdida de validación sube mientras pérdida de entrenamiento baja → sobreajuste.
- Usar callbacks:
  - EarlyStopping: parar cuando validación deje de mejorar.
  - ModelCheckpoint: guardar mejor modelo.
  - ReduceLROnPlateau: bajar tasa de aprendizaje si se atasca.

7. Errores Comunes a Evitar

- Olvidar normalizar datos → modelo no convergerá.
- Usar modelo demasiado profundo muy temprano → sobreajuste, entrenamiento lento.
- No mezclar datos → generalización pobre.
- Entrenar muy pocas épocas → subajuste.

8. Evaluación y Pruebas

- Usar model.evaluate() en conjunto de prueba para resultados imparciales.
- Siempre comparar precisión en entrenamiento, validación y prueba.
- Visualizar imágenes mal clasificadas para entender debilidades del modelo.

9. Guardar y Cargar Modelos

- Guardar modelo después del entrenamiento:

```
python

model.save("cnn_model.h5")
```

- Cargar después:

```
python

keras.models.load_model("cnn_model.h5")
```

10. Consejos Prácticos para Principiantes

- Empezar pequeño y verificar que el modelo funciona antes de hacerlo complejo.
- Cambiar una cosa a la vez (filtros, dropout, optimizador).
- Siempre hacer seguimiento de precisión/pérdida a través de épocas.
- No perseguir 100% de precisión; enfocarse en proceso de aprendizaje.
- Documentar tus experimentos (qué funcionó, qué no).

Lista de Verificación Rápida Antes del Entrenamiento

- [ ] Normalizar datos (x/255.0)
- [ ] Codificar etiquetas en one-hot
- [ ] Dividir en train/val/test
- [ ] Añadir dropout y batch norm donde sea necesario
- [ ] Usar 'relu' + 'softmax'
- [ ] Empezar con optimizador Adam, lr=0.001
- [ ] Monitorear precisión y pérdida de validación
- [ ] Guardar mejor modelo con checkpoints

Ejemplo: CNN Pequeña con Dos Bloques Convolucionales

python

```python
from tensorflow.keras import layers, models

def create_small_cnn(input_shape=(32,32,3), num_classes=10):
    model = models.Sequential()

    # Primer Bloque Conv
    model.add(layers.Conv2D(32, (3,3), activation='relu', padding='same', input_shape=input_shape))
    model.add(layers.BatchNormalization())
    model.add(layers.Conv2D(32, (3,3), activation='relu', padding='same'))
    model.add(layers.MaxPooling2D((2,2)))
    model.add(layers.Dropout(0.25))

    # Segundo Bloque Conv
    model.add(layers.Conv2D(64, (3,3), activation='relu', padding='same'))
    model.add(layers.BatchNormalization())
    model.add(layers.Conv2D(64, (3,3), activation='relu', padding='same'))
    model.add(layers.MaxPooling2D((2,2)))
    model.add(layers.Dropout(0.25))

    # Capas de Salida
    model.add(layers.GlobalAveragePooling2D())
    model.add(layers.Dense(128, activation='relu'))
    model.add(layers.Dropout(0.5))
    model.add(layers.Dense(num_classes, activation='softmax'))

    return model

# Ejemplo de uso
cnn_model = create_small_cnn()
cnn_model.summary()
```