

Pablo Andrés Zamora Vásquez - 21780

Diego Andrés Morales Aquino - 21762

Laboratorio No. 3

Aprendizaje profundo - CNN

1. Preparación de datos

Descargar el conjunto de datos de rótulos de tráfico que contiene las imágenes de las 43 clases mencionadas y dividir el conjunto de datos en conjuntos de entrenamiento, validación y prueba.

```
with open('datos\entrenamiento.p', 'rb') as file:
    train_data = pickle.load(file)

with open('datos\prueba.p', 'rb') as file:
    test_data = pickle.load(file)

with open('datos\\validacion.p', 'rb') as file:
    validation_data = pickle.load(file)
```

[3] ✓ 0.1s

```
print(train_data.keys())
train_data['features'].shape
```

[4] ✓ 0.0s

```
... dict_keys(['coords', 'labels', 'features', 'sizes'])
... (34799, 32, 32, 3)
```

Imprimir imagen de ejemplo

```
n = 1000 # Número de la imagen que se desea visualizar

print("Label:", train_data['labels'][n], " - ", labels[train_data['labels'][n]])

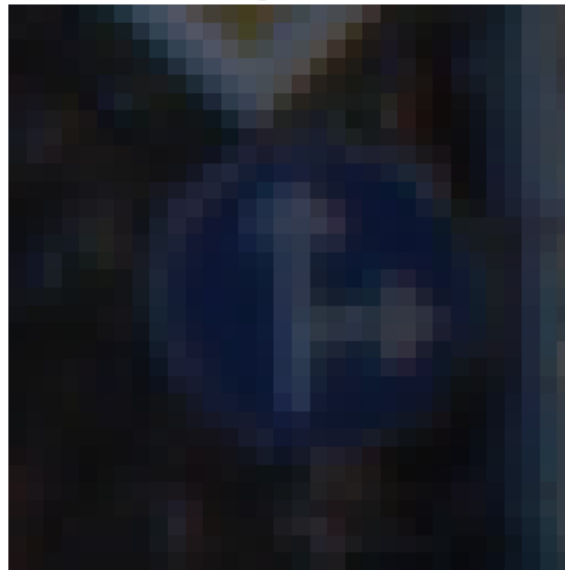
# Extraer la imagen específica
image_to_display = train_data['features'][n]

# Mostrar la imagen
plt.imshow(image_to_display)
plt.title(f'Imagen {n}')
plt.axis('off') # Opcional: para ocultar los ejes
plt.show()
```

[40] ✓ 0.1s

```
... Label: 36 - Vaya recto o a la derecha
```

Imagen 1000



Realizar preprocesamiento de las imágenes, como redimensionarlas a un tamaño estándar, normalización, etc.

Redimensionar a tamaño promedio

```
print(train_data['features'].shape)
print(test_data['features'].shape)
print(validation_data['features'].shape)
```

[6] ✓ 0.0s

```
... (34799, 32, 32, 3)
     (12630, 32, 32, 3)
     (4410, 32, 32, 3)
```

Todas las imagenes tienen las mismas dimensiones por lo que no es necesario redimensionar

✓ Normalizar imágenes

```
def normalize_images(images):  
    # Convertir los valores a float y normalizar dividiendo por 255  
    images = images.astype(np.float32) / 255.0  
    return images
```

[7] ✓ 0.0s

```
train_data['features'] = normalize_images(train_data['features'])  
test_data['features'] = normalize_images(test_data['features'])  
validation_data['features'] = normalize_images(validation_data['features'])
```

[8] ✓ 0.4s

```
# Verificar que los datos estén normalizados (rango de 0 a 1)  
print(train_data['features'].min(), train_data['features'].max())  
print(test_data['features'].min(), test_data['features'].max())  
print(validation_data['features'].min(), validation_data['features'].max())
```

[9] ✓ 0.1s

```
... 0.0 1.0  
     0.0 1.0  
     0.0 1.0
```

2. Implementación de la arquitectura Le-Net

Presentar la arquitectura Le-Net en detalle, explicando cada capa

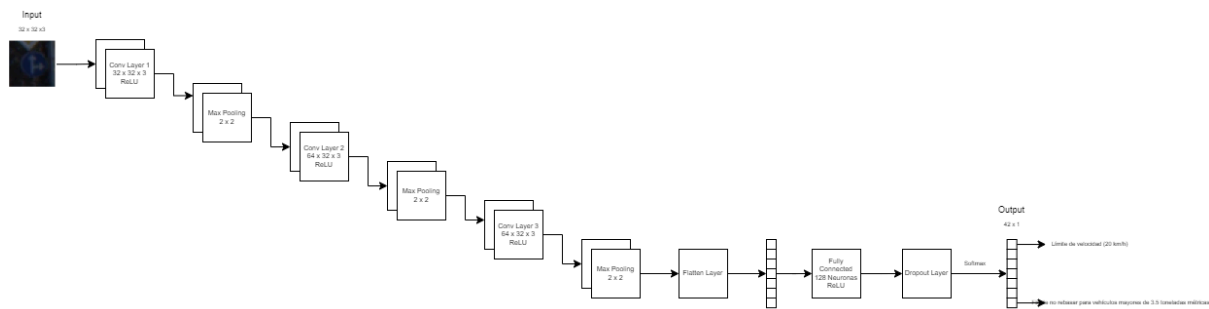
La arquitectura Le-Net fue desarrollada originalmente para la clasificación de dígitos del conjunto de datos MINST, utilizando imágenes en escala de grises (32 x 32 x 1). Modificando la arquitectura para utilizar imágenes RGB, las capas a implementar son las siguientes:


- **Capa de entrada (Input layer):** La entrada de esta arquitectura es una imagen de 32 x 32. En nuestro caso, será de 32 x 32 x 3, ya que utilizaremos imágenes en RGB.
- **Capa de primera convolución (Conv layer 1):** Esta capa aplica 32 filtros convolucionales, cada uno de tamaño 3x3, a la imagen de entrada. Se aplica convolución y la función de activación ReLU. Ya que la entrada será de 32x32x3, la salida será 30x30x32 (32 porque es el número de filtros aplicados y 30x30 porque la convolución reduce las dimensiones en 2 píxeles por cada borde debido al tamaño del kernel).

- **Primera capa de pooling (Max pooling layer 1):** Esta capa realiza un submuestreo (*max pooling*) con un tamaño de ventana de 2×2 y un *stride* de 2. Si la entrada es $30 \times 30 \times 32$, la salida será $15 \times 15 \times 32$.
- **Capa de segunda convolución (Conv layer 2):** Aplica 64 filtros convolucionales, cada uno de tamaño 3×3 , a la salida de la primera capa de pooling. Nuevamente, se realiza convolución y ReLU. Ya que la entrada es de $15 \times 15 \times 32$, la salida será de $13 \times 13 \times 64$ (64 porque es el número de filtros, 13×13 porque la convolución reduce las dimensiones en 2 píxeles por cada borde).
- **Segunda capa de pooling (Max pooling layer 2):** Realiza otro submuestreo (*max pooling*) con un tamaño de ventana de 2×2 y un *stride* de 2. Ya que la entrada es $13 \times 13 \times 64$, la salida será $6 \times 6 \times 64$.
- **Capa de tercera convolución (Conv layer 3):** Al igual que la segunda capa de convolución, aplica 64 filtros convolucionales, cada uno de tamaño 3×3 , a la salida de la segunda capa de pooling. Ya que la entrada es $6 \times 6 \times 64$, la salida será $4 \times 4 \times 64$.
- **Tercera capa de pooling (Max pooling layer 3):** Realiza el último submuestreo (*max pooling*) con un tamaño de ventana de 2×2 y un *stride* de 2. Ya que la entrada es $4 \times 4 \times 64$, la salida será de $2 \times 2 \times 64$.
- **Capa de aplanado (Flatten layer):** Aplana la salida de la tercera capa de pooling en un vector de características de 256 unidades.
- **Capa completamente conectada (Fully connected layer):** Conecta todas las 256 unidades de la capa aplanada a 128 neuronas. En esta capa se realiza multiplicación de matrices y la función de activación ReLU.
- **Capa de dropout:** Aplica Dropout para evitar el sobreajuste durante el entrenamiento. La tasa de dropout es 0.5.
- **Capa de salida (output layer):** Conecta las 128 neuronas de la capa anterior a 42 neuronas de salida, una por cada clase de imagen. Se aplica *softmax* como función de activación para obtener una distribución de probabilidades.

(Bangar, 2022).

Mostrar el diseño de la red Le-Net utilizando una herramienta de diagramación.



Ver a más detalle en:  lab3-ds.png

Explicar el proceso de convolución, función de activación y pooling.

- **Convolución:** Es una operación matemática que combina dos conjuntos de información. Es utilizada para extraer características locales de las imágenes, como bordes, texturas y otros patrones importantes. Un filtro, llamado kernel es una matriz pequeña de pesos que se desplaza sobre la imagen de entrada de acuerdo con un tamaño de paso, llamado *stride*. En cada posición, los valores del filtro se multiplican por cada elemento con los valores correspondientes de la imagen. Los productos de la multiplicación se suman para obtener un solo valor que constituye un píxel en la característica de salida. Este proceso se repite para cada posición del filtro en la imagen, generando un mapa de características, llamado *feature map*.
- **Función de activación:** En las capas de convolucionales a implementar en el modelo, se utilizará la función de activación **ReLU** (*Rectified Linear Unit*) con el propósito de introducir no linealidad en el modelo, permitiendo que la red aprenda características más complejas. ReLU se define mediante la fórmula:

$$ReLU(x) = \max(0, x)$$

- **Pooling:** El *pooling* o submuestreo permite reducir la dimensionalidad de los mapas de características, conservando la información más importante. Para este modelo, se utilizará *max pooling*, el cual selecciona el valor máximo en una ventana (en este caso, de 2 x 2) y lo utiliza como valor de salida para esa región.

3. Implementación de la arquitectura Le-Net

Utilizar la biblioteca de aprendizaje profundo TensorFlow para construir la arquitectura LeNet y definir la estructura de capas convolucionales, capas de pooling y capas fully connected.

Construcción del modelo

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Activation, Dropout, Flatten, Dense, Conv2D, MaxPooling2D
from tensorflow.keras.callbacks import TensorBoard
```

[23] ✓ 0.0s

```
modelo = Sequential()

forma_imagen = train_data['features'][0].shape

modelo.add(Conv2D(filters=32, kernel_size=(3, 3), input_shape=forma_imagen, activation='relu'))
modelo.add(MaxPooling2D(pool_size=(2, 2)))

modelo.add(Conv2D(filters=64, kernel_size=(3, 3), activation='relu'))
modelo.add(MaxPooling2D(pool_size=(2, 2)))

modelo.add(Conv2D(filters=64, kernel_size=(3, 3), activation='relu'))
modelo.add(MaxPooling2D(pool_size=(2, 2)))

modelo.add(Flatten())

modelo.add(Dense(128, activation='relu'))

modelo.add(Dropout(0.5))

# Ajuste para multiclase con softmax
num_clases = len(labels)
modelo.add(Dense(num_clases))
modelo.add(Activation('softmax'))

modelo.compile(loss='categorical_crossentropy',
               optimizer='adam',
               metrics=['accuracy'])
```

✓ 0.1s

Layer (type)	Output Shape	Param #
conv2d_3 (Conv2D)	(None, 30, 30, 32)	896
max_pooling2d_3 (MaxPooling2D)	(None, 15, 15, 32)	0
conv2d_4 (Conv2D)	(None, 13, 13, 64)	18,496
max_pooling2d_4 (MaxPooling2D)	(None, 6, 6, 64)	0
conv2d_5 (Conv2D)	(None, 4, 4, 64)	36,928
max_pooling2d_5 (MaxPooling2D)	(None, 2, 2, 64)	0
flatten_1 (Flatten)	(None, 256)	0
dense_2 (Dense)	(None, 128)	32,896
dropout_1 (Dropout)	(None, 128)	0
dense_3 (Dense)	(None, 43)	5,547
activation_1 (Activation)	(None, 43)	0

... Total params: 94,763 (370.17 KB)

... Trainable params: 94,763 (370.17 KB)

... Non-trainable params: 0 (0.00 B)

Explicar la importancia de la función de pérdida y el optimizador.

La importancia de la **función de pérdida** radica en que mide qué tan bien o mal está desempeñándose el modelo. Es decir, es una medida cuantitativa de la diferencia entre las predicciones del modelo y las etiquetas reales. El objetivo del entrenamiento es minimizar esta función de pérdida, ya que una pérdida más alta indica que el modelo está lejos de la solución correcta, mientras que una pérdida más baja indica que el modelo se está ajustando mejor a los datos.

La función de pérdida implementada fue *categorical cross entropy*, ya que suele utilizarse para problemas de clasificación multiclase, tal como el de las señales de tránsito.

(Shah, 2023).

Por otro lado, pero íntimamente ligado a la función de pérdida, la importancia del **optimizador** es que es el algoritmo que ajusta los pesos del modelo para minimizar la función de pérdida. Utiliza el gradiente de la función de pérdida con respecto a los parámetros del modelo para actualizar los pesos en la dirección que reduce la pérdida.

Para esta implementación, se utilizó *Adam*, el cual combina las ventajas de dos otros algoritmos de optimización: *AdaGrad* y *RMSProp*.

4. Implementación de la arquitectura Le-Net

Explicar el proceso de entrenamiento de la red neuronal.

El proceso de entrenamiento de una red neuronal implica ajustar los pesos del modelo para minimizar la función de pérdida en los datos de entrenamiento. Como se mencionó anteriormente, el modelo se compilará utilizando una función de pérdida *categorical_crossentropy*, un optimizador *adam* y métricas para evaluar el rendimiento *accuracy*. También se implementarán *callbacks*, específicamente *early stopping*, el cual detiene el entrenamiento si la pérdida de validación no mejora después de un número determinado de épocas, y *tensorboard*, que permite monitorear el entrenamiento en la librería del mismo nombre.

El entrenamiento se realiza en múltiples épocas. Una época se define como una pasada completa por todos los datos de entrenamiento. Así mismo, los datos de entrenamiento se dividen en lotes o *batches* más pequeños. Cada lote se pasa a través del modelo y se actualizan los pesos en función del error calculado para ese lote. Para cada lote de datos, se realiza un *forward pass*, donde los datos de entrada pasan a través de la red, capa por capa, hasta obtener las predicciones. La función de pérdida (*categorical_crossentropy*) compara las predicciones del modelo con las etiquetas verdaderas y calcula el error. Luego, se calculan los gradientes de la función de pérdida con respecto a los pesos del modelo y el optimizador (*Adam*) utiliza estos gradientes para actualizar los pesos en la dirección que minimiza la pérdida. Por último, después de cada época, el modelo se evalúa en el conjunto de validación para monitorear su rendimiento y detectar posibles problemas de sobreajuste.

Mostrar cómo cargar los datos de entrenamiento y validación en lotes. Definir hiperparámetros como tasa de aprendizaje, número de épocas, tamaño de lote, etc.

Los datos de entrenamiento se almacenan en memoria como *train_data* y *validation_data*. Además, se utilizan *labels* para codificar las clases:

```
# Convertir las etiquetas a one-hot encoding
num_clases = len(labels)
train_labels = to_categorical(train_data['labels'], num_clases)
validation_labels = to_categorical(validation_data['labels'], num_clases)
test_labels = to_categorical(test_data['labels'], num_clases)
```

✓ 0.0s

Se definen 30 épocas para el entrenamiento, así como un tamaño de lote de 32. No se definió la tasa de aprendizaje, ya que el optimizador (*Adam*) tiene un valor por defecto para esta (0.001).

```
history = modelo.fit(
    train_data['features'],
    train_labels,
    validation_data=(validation_data['features'], validation_labels),
    epochs=30,
    batch_size=32,
    callbacks = [early_stop, tablero])
```

[33] ✓ 1m 12.9s

```
... Epoch 1/30
1088/1088 ————— 14s 13ms/step - accuracy: 0.9766 - loss: 0.0783 - val_accuracy: 0.9544 - val_loss: 0.2248
Epoch 2/30
1088/1088 ————— 16s 15ms/step - accuracy: 0.9792 - loss: 0.0638 - val_accuracy: 0.9601 - val_loss: 0.1634
Epoch 3/30
1088/1088 ————— 14s 13ms/step - accuracy: 0.9831 - loss: 0.0543 - val_accuracy: 0.9676 - val_loss: 0.1886
Epoch 4/30
1088/1088 ————— 14s 13ms/step - accuracy: 0.9866 - loss: 0.0452 - val_accuracy: 0.9608 - val_loss: 0.1981
Epoch 5/30
1088/1088 ————— 14s 13ms/step - accuracy: 0.9849 - loss: 0.0562 - val_accuracy: 0.9592 - val_loss: 0.1947
```

5. Evaluación del modelo

Evaluar el modelo entrenado utilizando el conjunto de prueba.

Evaluar el modelo

```
test_loss, test_acc = modelo.evaluate(test_data['features'], test_labels)
print(f'Precisión en el conjunto de prueba: {test_acc}')
```

[34] ✓ 1.6s

```
... 395/395 ————— 1s 4ms/step - accuracy: 0.9508 - loss: 0.2927
Precisión en el conjunto de prueba: 0.9482185244560242
```


Se obtuvo una precisión del **94.82%**

Mostrar cómo calcular métricas de evaluación, como Precisión, Recall y F1-Score para cada clase.

Utilizando el módulo *metrics* de la librería *sklearn*. Se obtuvo la precisión, el *recall*, el *f1-score* y el soporte de cada clase del conjunto de datos:

395/395	2s 4ms/step	precision	recall	f1-score	support
	Limite velocidad (20km/h)	0.97	0.93	0.95	60
	Limite velocidad (30km/h)	0.92	0.99	0.95	720
	Limite velocidad (50km/h)	0.92	0.97	0.94	750
	Limite velocidad (60km/h)	0.98	0.93	0.95	450
	Limite velocidad (70km/h)	0.93	0.98	0.96	660
	Limite velocidad (80km/h)	0.91	0.93	0.92	630
	Fin de limite velocidad (80km/h)	1.00	0.79	0.88	150
	Limite velocidad (100km/h)	0.96	0.92	0.94	450
	Limite velocidad (120km/h)	0.97	0.86	0.91	450
	No rebasar	0.97	0.99	0.98	480
	No rebasar para vehículos mayores de 3.5 toneladas métricas	0.97	0.99	0.98	660
	Derecho-de-vía en la siguiente intersección	0.95	0.96	0.96	420
	Camino prioritario	0.93	0.99	0.96	690
	Ceda el paso	0.95	0.99	0.97	720
	Alto	1.00	0.99	0.99	270
	No vehículos	0.99	0.86	0.92	210
	Prohibido vehículos mayores de 3.5 toneladas métricas	0.99	0.99	0.99	150
	No hay entrada	1.00	0.95	0.97	360
	Precaución general	0.95	0.89	0.92	390
	Curva peligrosa a la izquierda	0.81	0.98	0.89	60
	Curva peligrosa a la derecha	0.84	0.90	0.87	90
	Doble curva	0.94	0.52	0.67	90
	Camino disparejo	1.00	0.93	0.96	120
	Camino resbaloso	0.91	0.98	0.94	150
	Camino se reduce a la derecha	0.92	0.73	0.81	90
	Trabajos adelante	0.98	0.90	0.94	480
	Señales de Tráfico -semáforos-	0.88	0.98	0.93	180
	Cruce de peatones	0.42	0.48	0.45	60
	Cruce de Niños	0.90	0.95	0.92	150
	Cruce de bicicletas	0.81	0.99	0.89	90
	Cuidado hielo/nieve	0.94	0.73	0.82	150
	Cruce de animales silvestres	0.98	0.98	0.98	270
	Fin de todos los limites de velocidad y rebase	0.89	0.93	0.91	60
	Gire a la derecha adelante	1.00	1.00	1.00	210
	Gire a la izquierda adelante	1.00	0.99	1.00	120
	Recto solo	0.99	0.99	0.99	390
	Vaya recto o a la derecha	0.99	0.93	0.96	120
	Vaya recto o a la izquierda	1.00	0.97	0.98	60
	Manténgase a la derecha	0.98	0.99	0.98	690
	Manténgase a la izquierda	0.99	0.96	0.97	90
	Vuelta en U obligada	0.96	0.98	0.97	90
	Fin de no rebasar	0.98	0.78	0.87	60
	Fin de no rebasar para vehículos mayores de 3.5 toneladas métricas	0.86	0.99	0.92	90
	accuracy			0.95	12630
	macro avg	0.94	0.92	0.92	12630
	weighted avg	0.95	0.95	0.95	12630

6. Mejoras y experimentación (opcional)

Discutir posibles mejoras en el rendimiento del modelo, como ajuste de hiperparámetros, aumento de datos, regularización, etc.

Para mejorar el modelo implementado mediante el ajuste de hiperparámetros, podría utilizarse **grid search**, el cual explora un espacio de hiperparámetros definido y evalúa todas las combinaciones posibles.

Por otro lado, si se decide mejorar el rendimiento del modelo mediante el aumento de datos, podría ser un poco más complicado, ya que el conjunto de datos consta de imágenes reales de señales de tránsito, por lo que podría intentarse modificar las imágenes con rotaciones, traslaciones o espejados, pero dado que la orientación de las señales constituye un aspecto importante de estas, podría no ser una buena idea.

Finalmente, ya se aplica regularización al modelo mediante **dropout**, aunque podría intentarse aplicar L_2 en su lugar para determinar si se obtiene un mejor desempeño.

Referencias

Bangar, S. (2022). *LeNet 5 Architecture Explained*. Obtenido de <https://medium.com/@siddheshb008/lenet-5-architecture-explained-3b559cb2d52b>

Shah, D. (2023). *Cross Entropy Loss: Intro, Applications, Code*. Obtenido de <https://www.v7labs.com/blog/cross-entropy-loss-guide#:~:text=Categorical%20cross%2Dentropy%20is%20used,a%20machine%20learning%20loss%20function>.