Máster Universitario en Ingeniería Informática

de la Universidad de Granada

Práctica de redes neuronales

Reconocimiento óptimo de caracteres MNIST

Inteligencia Computacional (IC)

Autora

Marina Jun Carranza Sánchez

Índice

Índice de figuras

Índice de cuadros

1. Introducción

En esta sección se comentan las herramientas y el entorno de desarrollo utilizado para la práctica, así como algunos detalles acerca de la preparación de los datos.

1.1. Herramientas y entorno

Se ha optado por utilizar bibliotecas disponibles públicamente, que implementan los algoritmos necesarios para la práctica. El entorno de desarrollo elegido ha sido un Jupyter Notebook en *Google Colab*, haciendo uso de las funcionalidades de *TensorFlow* y *Keras*.

Esta elección se debe a su acceso gratuito a potentes recursos de computación como GPUs y TPUs, lo que acelera el entrenamiento de modelos complejos. Además, Colab ofrece un entorno preconfigurado y una integración fluida con Google Drive, lo que simplifica el almacenamiento de los conjuntos de datos de las imágenes.

1.2. Preparación de los datos

El objetivo es la implementación de redes neuronales para la identificación (o clasificación) correcta de dígitos en el conjunto de imágenes MNIST.

Antes de nada, se han definido cuatro funciones: las dos primeras (Figura 1) se encargan de abrir los ficheros `train-images.idx3-ubyte` y `t10k-images.idx3-ubyte`, que contiene las imágenes para el entrenamiento y las pruebas respectivamente, mientras que las dos últimas (Figura 2) sirven para cargar las etiquetas correspondientes a los dos conjuntos de imágenes (`train-labels.idx1-ubyte` y `t10k-labels.idx1-ubyte`).

```python
import numpy as np

def cargar_train_imgs():
    with open('/content/drive/MyDrive/MÁSTER/IC/samples/train-images.idx3-ubyte', 'rb') as f:
        f.read(16)  # Descartar cabecera
        images = []
        while True:
            image = f.read(784)
            if len(image) != 784:
                break
            images.append([x for x in image])
    return np.array(images)

def cargar_test_imgs():
    with open('/content/drive/MyDrive/MÁSTER/IC/samples/t10k-images.idx3-ubyte', 'rb') as f:
        f.read(16)  # Descartar cabecera
        images = []
        while True:
            image = f.read(784)
            if len(image) != 784:
                break
            images.append([x for x in image])
    return np.array(images)
```

Figura 1: Funciones para cargar las imágenes de entrenamiento y prueba

```python
def cargar_train_labels():
    with open('/content/drive/MyDrive/MÁSTER/IC/samples/train-labels.idx1-ubyte', 'rb') as f:
        f.read(8)  # Descartar cabecera
        etiquetas = [x for x in f.read()]
    return np.array(etiquetas)

def cargar_test_labels():
    with open('/content/drive/MyDrive/MÁSTER/IC/samples/t10k-labels.idx1-ubyte', 'rb') as f:
        f.read(8)  # Descartar cabecera
        etiquetas = [x for x in f.read()]
    return np.array(etiquetas)
```

Figura 2: Funciones para cargar las etiquetas de los conjuntos de imágenes

Una vez definidas las funciones anteriores, se llaman para cargar los datos. A continuación, se preprocesan las imágenes de entrenamiento y prueba normalizándolas, dividiendo sus valores de píxel por 255.0, lo que escala los valores al rango [0, 1]. Las etiquetas de ambos conjuntos se convierten a formato categórico utilizando `to_categorical` con 10 clases, lo que transforma las etiquetas en vectores one-hot adecuados para la clasificación multiclase.

Finalmente, las imágenes de entrenamiento y prueba se reestructuran con la función `reshape` para tener dimensiones de 28x28 píxeles y un único canal (blanco y negro), preparando los datos en el formato requerido para ser ingresados en las redes neuronales. Estos pasos vienen implementados en la Figura 3.

```python
# Cargar los datos
train_images = cargar_train_imgs()
train_labels = cargar_train_labels()
test_images = cargar_test_imgs()
test_labels = cargar_test_labels()

# Preprocesar los datos de entrenamiento (Normalizar y convertir etiquetas)
train_images = train_images / 255.0  # rango [0, 1]
train_labels = to_categorical(train_labels, 10)

# Preprocesar los datos de prueba (Normalizar y convertir etiquetas)
test_images = test_images / 255.0  # rango [0, 1]
test_labels = to_categorical(test_labels, 10)

train_images = train_images.reshape(-1, 28, 28, 1)  # '1': el canal (blanco y negro)
test_images = test_images.reshape(-1, 28, 28, 1)
```

Figura 3: Preparación de los conjuntos de datos

1.3. Metodología

Se va a seguir la siguiente metodología para cada red neuronal a implementar:

1. **Creación del modelo**: se crea un modelo secuencial utilizando la clase `Sequential` de *Keras*. Este modelo consiste en una pila lineal de capas que se añaden en orden.

2. **Compilación del modelo**: se compila el modelo, pudiendo configurar el optimizador (como `adam`), la función de pérdida (como `categorical_crossentropy`) y las métricas de evaluación (como `accuracy`, para monitorizar la precisión del modelo).

3. **Entrenamiento del modelo**: `model.fit()` entrenará el modelo con los datos de entrenamiento, pudiendo indicarle el número de épocas (`epochs`), el tamaño de lotes (`batch_size`) y el nivel de detalle de las salidas por consola (`verbose`).

4. **Cálculo del tiempo de entrenamiento**: utilizando `time.time()` para registrar los tiempos de inicio y final del entrenamiento y así poder calcular la duración total.

5. **Evaluación del modelo**: se evalúa el rendimiento del modelo tanto en el conjunto de entrenamiento como en el de prueba, teniendo en cuenta la pérdida y precisión, y pudiendo indicar con `verbose` cuántos detalles de la evaluación mostrar por la consola.

6. **Cálculo del porcentaje de error**: se calculan los errores de entrenamiento y de prueba mediante la expresión $error\,\% = 100 - accuracy * 100$. Estos errores se imprimen como porcentajes con dos decimales, proporcionando una medida clara de las tasas de fallo del modelo en ambos conjuntos de dato.

Los tres últimos pasos anteriores vienen descritos en la Figura 4.

```
# Calcular el tiempo de entrenamiento
end_time = time.time()
training_time = end_time - start_time
print(f"Tiempo de entrenamiento: {training_time:.2f} segundos")

# Evaluar el modelo en el conjunto de entrenamiento y el de prueba
train_loss, train_acc = model.evaluate(train_images, train_labels, verbose=0)
test_loss, test_acc = model.evaluate(test_images, test_labels, verbose=0)

# Calcular y mostrar los errores
train_error = 100 - train_acc * 100
test_error = 100 - test_acc * 100
print(f"Error de entrenamiento: {train_error:.2f}%")
print(f"Error de prueba: {test_error:.2f}%")
```

Figura 4: Fragmento de código para imprimir los resultados del modelo

2. Red neuronal simple

En primer lugar, se ha implementado una red neuronal simple, compuesta por una capa de entrada y una de salida de tipo softmax.

2.1. Fundamentos teóricos

La red neuronal implementada está basada en un modelo de tipo feedforward, que utiliza una arquitectura sencilla con las características descritas a continuación.

Se inicia con una capa de entrada para convertir las imágenes de entrada de forma (28, 28, 1) en vectores unidimensionales de 784 valores, como recomiendan Goodfellow (Goodfellow et al., 2016) para garantizar compatibilidad con capas densas.

La capa de salida utiliza una activación *softmax*, que convierte los logits en probabilidades interpretables para clasificación multi-clase mediante la fórmula:

$$\sigma(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

Esto asegura que las probabilidades sumen 1, facilitando su interpretación como probabilidades (Bishop, 2006). La función de pérdida empleada, `categorical_crossentropy`, mide la distancia entre las probabilidades predichas y las etiquetas reales usando la fórmula:

$$H(y, \hat{y}) = -\sum_{i=1}^{K} y_i \log(\hat{y}_i)$$

Maximizando así la probabilidad de la clase correcta (Murphy, 2012). Para la optimización, se utiliza Adam (Kingma & Ba, 2014), que combina gradientes adaptativos y momentos para acelerar la convergencia mediante actualizaciones eficientes basadas en primeros y segundos momentos de los gradientes. Finalmente, el aprendizaje supervisado se realiza mediante retropropagación (Rumelhart et al., 1986) en minibatches, ajustando los pesos para minimizar la pérdida total.

2.2. Implementación

```
# Crear modelo
model = Sequential([
    Flatten(input_shape=(28, 28, 1)),  # Aplanar las imágenes
    Dense(10, activation='softmax')   # Capa de salida softmax
])

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

start_time = time.time()

# Entrenar modelo
history = model.fit(train_images, train_labels,
                    epochs=10,
                    batch_size=32,
                    verbose=2)
```

Figura 5: Modelado de la primera red neuronal simple

En la Figura 5, se define una red neuronal secuencial (`Sequential`) con una capa de entrada `Flatten`, que aplana las imágenes de tamaño (28, 28, 1) en vectores. Luego, incluye una capa de salida `Dense` con 10 neuronas y activación *softmax*, adecuada para clasificar en 10 categorías.

El modelo se compila utilizando el optimizador *adam*, la función de pérdida `categorical_crossentropy`, y la métrica de precisión (*accuracy*). El entrenamiento se realiza con el método `fit`, utilizando 10 épocas y un tamaño de batch de 32, mientras se registran métricas como precisión durante el proceso. La salida es configurada con *verbose*=2 para mostrar información detallada en la consola.

2.3. Análisis de resultados

```
Epoch 9/10
1875/1875 - 3s - 2ms/step - accuracy: 0.9299 - loss: 0.2527
Epoch 10/10
1875/1875 - 4s - 2ms/step - accuracy: 0.9310 - loss: 0.2508
Tiempo de entrenamiento: 34.14 segundos
Error de entrenamiento: 6.67%
Error de prueba: 7.12%
```

Figura 6: Resultados de la primera red neuronal simple

3. Red neuronal multicapa

En segundo lugar, se ha optado por una red neuronal con una capa oculta de 256 unidades logísticas y una capa de salida de tipo softmax.

3.1. Fundamentos teóricos

Esta red tiene una arquitectura algo más compleja que el modelo básico anterior, ya que introduce una capa oculta densa con 256 unidades logísticas y activación `ReLU` (*Rectified Linear Unit*). La función `ReLU` está definida como:

$$f(x) = \text{máx}(0, x),$$

Esta permite manejar de forma eficiente problemas de desvanecimiento del gradiente al solo activar las neuronas con valores positivos, lo que acelera la convergencia del entrenamiento (Nair & Hinton, 2010).

Esta capa oculta añade capacidad de representación al modelo, permitiendo aprender características más complejas de los datos de entrada. La capa de entrada y de salida son las mismas definidas en la anterior implementación.

3.2. Implementación

```
model = Sequential([
    Flatten(input_shape=(28, 28, 1)),
    Dense(256, activation='relu'),   # Capa oculta de 256 unidades logísticas
    Dense(10, activation='softmax')
])

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

start_time = time.time()

history = model.fit(train_images, train_labels,
                    epochs=10,
                    batch_size=32,
                    verbose=2)
```

Figura 7: Modelado de la segunda red neuronal

La implementación es prácticamente idéntica a la de la red neuronal anterior, salvo por que se le ha añadido entre la capa de entrada y la de salida una capa oculta `Dense` con 256 neuronas y activación `ReLU`, que permite aprender características no lineales.

De nuevo, el modelo se compila utilizando el optimizador *adam*, la función de pérdida `categorical_crossentropy`, y la métrica de *accuracy*.

El entrenamiento se realiza con el método `fit`, configurado para 10 épocas, un tamaño de lote de 32, y *verbose*=2 para obtener información del proceso de entrenamiento en consola.

3.3. Análisis de resultados

```
Epoch 9/10
1875/1875 - 11s - 6ms/step - accuracy: 0.9961 - loss: 0.0126
Epoch 10/10
1875/1875 - 8s - 4ms/step - accuracy: 0.9968 - loss: 0.0102
Tiempo de entrenamiento: 92.66 segundos
Error de entrenamiento: 0.25%
Error de prueba: 2.07%
```

Figura 8: Resultados de la segunda red neuronal

4. Red neuronal convolutiva

A continuación, se ha considerado una red neuronal convolutiva entrenada con gradiente descendente estocástico.

4.1. Fundamentos teóricos

La red neuronal implementada corresponde a una red convolucional (CNN), ampliamente utilizada para tareas de clasificación de imágenes debido a su capacidad para capturar patrones espaciales y características jerárquicas en los datos. Este modelo combina capas convolutivas, de *pooling* y densas, cada una diseñada para cumplir una función específica:

- **Capas convolutivas**: las capas `Conv2D` aplican filtros con tamaño (3, 3) para detectar características locales como bordes y texturas en las imágenes de entrada. Estas capas usan la activación `ReLU` vista antes, para introducir no linealidad y mejorar la eficiencia computacional (Nair & Hinton, 2010).

- **Capas de *Pooling***: las capas `MaxPooling2D` con tamaño de ventana (2, 2) reducen la dimensionalidad espacial de las características extraídas, manteniendo la información más relevante y reduciendo la carga computacional. El *max-pooling* selecciona el valor máximo en cada ventana, lo que ayuda a captar las características más prominentes.

- **Capas densas**: la salida de las capas convolutivas es aplanada con `Flatten` y pasada a una capa completamente conectada (`Dense`) con 128 neuronas y activación `ReLU`, para integrar las características aprendidas. Finalmente, una capa de salida con activación *softmax* convierte las predicciones en probabilidades para las 10 clases del problema.

El modelo utiliza el optimizador SGD (*Stochastic Gradient Descent*) con tasa de aprendizaje de 0.01 y *momentum* de 0.9. El *momentum* acelera la convergencia al suavizar las actualizaciones de gradiente, evitando oscilaciones en direcciones ortogonales al gradiente principal (Sutskever et al., 2013). La función de pérdida utilizada es `categorical_crossentropy`, adecuada para clasificación multiclase, y se valida el rendimiento durante el entrenamiento utilizando un 20 % de los datos de entrenamiento como conjunto de validación interna.

Las CNN, como este modelo, son efectivas para la extracción automática de características espaciales y han demostrado gran éxito en tareas de visión computacional, como describen LeCun (LeCun et al., 1998) y Krizhevsky (Krizhevsky et al., 2012).

4.2. Implementación

```python
model = Sequential([
    Conv2D(32, kernel_size=(3, 3), activation='relu', input_shape=(28, 28, 1)),  # Primera capa convolutiva
    MaxPooling2D(pool_size=(2, 2)),                                              # Capa de max-pooling
    Conv2D(64, kernel_size=(3, 3), activation='relu'),                           # Segunda capa convolutiva
    MaxPooling2D(pool_size=(2, 2)),                                              # Capa de max-pooling
    Flatten(),                                                                   # Aplanar la salida
    Dense(128, activation='relu'),                                               # Capa densa con 128 unidades
    Dense(10, activation='softmax')                                              # Capa de salida softmax
])

# Compilar el modelo usando SGD
model.compile(optimizer=SGD(learning_rate=0.01, momentum=0.9),  # SGD con momentum
              loss='categorical_crossentropy',
              metrics=['accuracy'])

start_time = time.time()

history = model.fit(train_images, train_labels,
                    epochs=15,            # Ajustar para garantizar buena convergencia
                    batch_size=32,
                    validation_split=0.2, # Validación interna
                    verbose=2)
```

Figura 9: Modelado de la tercera red neuronal

El modelo secuencial implementado consta de las siguientes capas:

- Dos capas convolutivas (`Conv2D`) con 32 y 64 filtros respectivamente, tamaño de kernel (3, 3) y activación `ReLU`, para extraer características espaciales locales.
- Dos capas de *max-pooling* (`MaxPooling2D`) con tamaño de ventana (2, 2), para reducir la dimensionalidad espacial.

- Una capa de aplanamiento (`Flatten`) para transformar las características espaciales en un vector.
- Una capa densa (`Dense`) con 128 unidades y activación `ReLU`, para integrar las características aprendidas.
- Una capa de salida (`Dense`) con 10 unidades y activación *softmax*, para clasificación en 10 clases.

El modelo se compila utilizando el optimizador SGD con una tasa de aprendizaje de 0.01 y *momentum* de 0.9. El entrenamiento se realiza con el método `fit`, configurado para 15 épocas, tamaño de lote de 32 y un *split* del 20 % de los datos de entrenamiento como conjunto de validación. La salida se configura con `verbose=2` para mostrar el progreso en consola.

4.3. Análisis de resultados

```
Epoch 14/15
1500/1500 - 81s - 54ms/step - accuracy: 0.9985 - loss: 0.0046 - val_accuracy: 0.9895 - val_loss: 0.0446
Epoch 15/15
1500/1500 - 82s - 55ms/step - accuracy: 0.9996 - loss: 0.0016 - val_accuracy: 0.9907 - val_loss: 0.0465
Tiempo de entrenamiento: 1026.72 segundos
Error de entrenamiento: 0.24%
Error de prueba: 0.93%
```

Figura 10: Resultados de la tercera red neuronal

5. Deep learning

Deep learning usando pre-entrenamiento de autoencoders para extraer características de las imágenes usando técnicas no supervisadas y una red neuronal simple con una capa de tipo softmax: del 1.8 % al 2.2 % de error sobre el conjunto de prueba (tiempo de entrenamiento necesario: unos veinte minutos usando una implementación en un lenguaje interpretado como Matlab).

5.1. Fundamentos teóricos

5.2. Implementación

```
# Crear el autoencoder
input_img = Input(shape=(28, 28, 1))

# Codificador
x = Conv2D(32, (3, 3), activation='relu', padding='same')(input_img)
x = MaxPooling2D((2, 2), padding='same')(x)
x = Conv2D(64, (3, 3), activation='relu', padding='same')(x)
encoded = MaxPooling2D((2, 2), padding='same')(x)

# Decodificador
x = Conv2D(64, (3, 3), activation='relu', padding='same')(encoded)
x = UpSampling2D((2, 2))(x)
x = Conv2D(32, (3, 3), activation='relu', padding='same')(x)
x = UpSampling2D((2, 2))(x)
decoded = Conv2D(1, (3, 3), activation='sigmoid', padding='same')(x)

autoencoder = Model(input_img, decoded)
autoencoder.compile(optimizer='adam', loss='binary_crossentropy')
```

Figura 11: Modelado de la cuarta red neuronal

```
# Entrenar el autoencoder
start_time = time.time()
autoencoder.fit(train_images, train_images,
                epochs=10,
                batch_size=256,
                validation_split=0.2,
                verbose=2)
end_time = time.time()

print(f"Tiempo de entrenamiento del autoencoder: {end_time - start_time:.2f} segundos")

# Extraer características del autoencoder
encoder = Model(input_img, encoded)
encoded_train = encoder.predict(train_images)
encoded_test = encoder.predict(test_images)

# Aplanar las características extraídas
X_train_flat = encoded_train.reshape(encoded_train.shape[0], -1)
X_test_flat = encoded_test.reshape(encoded_test.shape[0], -1)
```

Figura 12: Modelado de la cuarta red neuronal

```python
# Crear y entrenar la red neuronal simple
model = Sequential([
    Dense(128, activation='relu', input_shape=(X_train_flat.shape[1],)),
    Dense(10, activation='softmax')
])

# Compilar el modelo usando SGD
model.compile(optimizer=SGD(learning_rate=0.01, momentum=0.9),
              loss='categorical_crossentropy',
              metrics=['accuracy'])

# Entrenar el modelo supervisado
start_time = time.time()
history = model.fit(X_train_flat, train_labels,
                    epochs=20,
                    batch_size=32,
                    validation_split=0.2,
                    verbose=2)
end_time = time.time()
```

Figura 13: Modelado de la cuarta red neuronal

5.3. Análisis de resultados

```
Epoch 19/20
1500/1500 - 4s - 3ms/step - accuracy: 0.9851 - loss: 0.0460 - val_accuracy: 0.9763 - val_loss: 0.0912
Epoch 20/20
1500/1500 - 3s - 2ms/step - accuracy: 0.9852 - loss: 0.0439 - val_accuracy: 0.9761 - val_loss: 0.0828
Tiempo de entrenamiento total (autoencoder + red neuronal): 173.28 segundos
Error de entrenamiento: 1.45%
Error de prueba: 2.10%
```

Figura 14: Resultados de la cuarta red neuronal

6. Red neuronal combinada mejorada

Llegar a 99.7 % mínimo 98

6.1. Fundamentos teóricos

6.2. Implementación

1. Dividir train set en validation set
2. Data augmentation
3. Convolutiva
4. EarlyStopping
5. ReduceLROnPlateau

6.3. Análisis de resultados

7. Conclusiones y trabajos futuros

En conclusión, este

8. Bibliografía

Bishop, C. M. (2006). *Pattern Recognition and Machine Learning*. Springer.

Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep Learning*. MIT Press.

Kingma, D. P., & Ba, J. (2014). Adam: A Method for Stochastic Optimization. *arXiv preprint arXiv:1412.6980*.

Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). ImageNet Classification with Deep Convolutional Neural Networks. *Advances in Neural Information Processing Systems*, *25*, 1097-1105.

LeCun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). Gradient-Based Learning Applied to Document Recognition. *Proceedings of the IEEE*, *86*(11), 2278-2324.

Murphy, K. P. (2012). *Machine Learning: A Probabilistic Perspective*. MIT Press.

Nair, V., & Hinton, G. E. (2010). Rectified Linear Units Improve Restricted Boltzmann Machines. *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, 807-814.

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature*, *323*(6088), 533-536.

Sutskever, I., Martens, J., Dahl, G., & Hinton, G. (2013). On the Importance of Initialization and Momentum in Deep Learning. *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, 1139-1147.