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

Este se enfoca en

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

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

Red neuronal simple, con una capa de entrada y una capa de salida de tipo softmax: 7.8 % de error sobre el conjunto de prueba y 5.6 % de error sobre el conjunto de entrenamiento (tiempo de entrenamiento necesario: sobre un minuto usando una implementación en un lenguaje interpretado como Matlab).

2.1. Fundamentos teóricos

tfhtfhtr

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

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

Red neuronal multicapa, con una capa oculta de 256 unidades logísticas y una capa de salida de tipo softmax: 3.0 % de error sobre el conjunto de prueba y 0.0 % de error sobre el conjunto de entrenamiento (tiempo de entrenamiento necesario: unos cuatro minutos usando una implementación en un lenguaje interpretado como Matlab).

3.1. Fundamentos teóricos

3.2. Implementación

```python
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

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

Red neuronal convolutiva entrenada con gradiente descendente estocástico: 2.7 % de error sobre el conjunto de prueba (tiempo de entrenamiento necesario: unos 13 minutos usando una implementación en un lenguaje interpretado como Matlab).

4.1. Fundamentos teóricos

4.2. Implementación

```
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

4.3. Análisis de resultados

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 10: Modelado de la cuarta red neuronal

```python
# 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 11: 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 12: 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 13: 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