TL06 Capas convolucionales y de agrupación

Índice

1 Capas convolucionales y de agrupación

Capas convolucionales:

- **API:** https://keras.io/api/layers/convolution_layers
- **Clase Conv2D:** https://keras.io/api/layers/convolution_layers/convolution2d
- Algunos parámetros relevantes de la clase Conv2D:
  - `filters`: número de filtros (canales de salida) a aplicar (en cada canal de entrada)
  - `kernel_size`: tamaño de los filtros; entero o par de enteros
  - `strides=(1, 1)`: saltos; entero o par de enteros
  - `padding="valid"`: "valid" indica sin relleno; "same" indica salida del mismo tamaño que la entrada (sin saltos)
  - `data_format=None`: "channels_last" (si no se ha configurado otra cosa) o "channels_first"
  - `activation=None`: función de activación

Capas de agrupación:

- **API:** https://keras.io/api/layers/pooling_layers
- **Clase MaxPooling2D:** https://keras.io/api/layers/pooling_layers/max_pooling2d
- Algunos parámetros relevantes de la clase MaxPooling2D:
  - `pool_size=(2, 2)`: tamaño de la ventana; entero o par de enteros
  - `strides=None`: saltos; entero o par de enteros
  - `padding="valid"`: "valid" indica sin relleno; "same" indica salida del mismo tamaño que la entrada (sin saltos)
  - `data_format=None`: "channels_last" (si no se ha configurado otra cosa) o "channels_first"

Ejemplo de red convolucional básica: dos pares Conv2D-MaxPooling2D previos a un MLP

2 MNIST

MNIST: resumen de resultados

- MLP inicial: MLP con una capa oculta de 800 RELUs, batch size 16, 10 épocas; 98.1% en test
- Mejor arquitectura: una capa oculta de 800 RELUs, 98.2% en val, 98.2% en test (98.2% modelo val)
- Learning rate y batch size: ajustados a 0.00168 y 256; 98.5% en val, 98.5% en test (98.5% modelo val)
- ReduceLROnPlateau: factor 0.3787 y paciencia 10; 98.5% en val, 98.4% en test (98.4% modelo val)

Inicialización: librerías, semilla, lectura de MNIST **sin aplanar imágenes** y partición train-val-test

In [ ]:
```
import numpy as np; import matplotlib.pyplot as plt
import os; os.environ['TF_CPP_MIN_LOG_LEVEL'] = '2'
import keras; import keras_tuner
keras.utils.set_random_seed(23); input_dim = (28, 28, 1); num_classes = 10
(x_train_val, y_train_val), (x_test, y_test) = keras.datasets.mnist.load_data()
x_train_val = x_train_val.astype("float32") / 255.0
x_test = x_test.astype("float32") / 255.0
x_train_val = np.expand_dims(x_train_val, -1)
x_test = np.expand_dims(x_test, -1)
print(x_train_val.shape, y_train_val.shape, x_test.shape, y_test.shape)
y_train_val = keras.utils.to_categorical(y_train_val, num_classes)
y_test = keras.utils.to_categorical(y_test, num_classes)
x_train = x_train_val[:-10000]; x_val = x_train_val[-10000:]
y_train = y_train_val[:-10000]; y_val = y_train_val[-10000:]
```

```
(60000, 28, 28, 1) (60000,) (10000, 28, 28, 1) (10000,)
```

MyHyperModel: exploramos número de filtros de la primera capa; doblamos en la segunda

In [ ]:
```python
class MyHyperModel(keras_tuner.HyperModel):
    def build(self, hp):
        M = keras.Sequential()
        M.add(keras.Input(shape=(28, 28, 1)))
        filters = hp.Int("filters", min_value=8, max_value=64, step=2, sampling="log")
        M.add(keras.layers.Conv2D(filters, kernel_size=(3, 3), activation="relu"))
        M.add(keras.layers.MaxPooling2D(pool_size=(2, 2)))
        M.add(keras.layers.Conv2D(2*filters, kernel_size=(3, 3), activation="relu"))
        M.add(keras.layers.MaxPooling2D(pool_size=(2, 2)))
        M.add(keras.layers.Flatten())
        M.add(keras.layers.Dense(units=800, activation='relu'))
        M.add(keras.layers.Dense(10, activation='softmax'))
        opt = keras.optimizers.Adam(learning_rate=0.00168)
        M.compile(loss="categorical_crossentropy", optimizer=opt, metrics=["accuracy"])
        return M
    def fit(self, hp, M, x, y, xy_val, **kwargs):
        factor = 0.3787; patience = 5
        reduce_cb = keras.callbacks.ReduceLROnPlateau(
            monitor='val_accuracy', factor=factor, patience=patience, min_delta=1e-4, min_lr=1e-5)
        early_cb = keras.callbacks.EarlyStopping(monitor='val_accuracy', patience=2*patience, min_delta=1e-5)
        kwargs['callbacks'].extend([reduce_cb, early_cb])
        return M.fit(x, y, batch_size=256, epochs=100, validation_data=xy_val, **kwargs)
```

Experimento: exploración y resumen de resultados

In [ ]:
```
tuner = keras_tuner.BayesianOptimization(
    MyHyperModel(), objective="val_accuracy", max_trials=10, executions_per_trial=1,
    overwrite=True, directory="/tmp", project_name="MNIST")
```

In [ ]:
```
tuner.search(x_train, y_train, (x_val, y_val))
```

```
Trial 10 Complete [00h 12m 55s]
val_accuracy: 0.9940000176429749

Best val_accuracy So Far: 0.9940000176429749
Total elapsed time: 01h 38m 03s
```

In [ ]:
```
tuner.results_summary(num_trials=4)
```

```
Results summary
Results in /tmp/MNIST
Showing 4 best trials
Objective(name="val_accuracy", direction="max")

Trial 02 summary
Hyperparameters:
filters: 64
Score: 0.9940000176429749

Trial 09 summary
Hyperparameters:
filters: 64
Score: 0.9940000176429749

Trial 00 summary
Hyperparameters:
filters: 32
Score: 0.9932000041007996

Trial 03 summary
Hyperparameters:
filters: 64
Score: 0.9929999709129333
```

Experimento (cont.): evaluación en test de los mejores modelos en validación

In [ ]:
```python
num_models = 10
best_hyperparameters = tuner.get_best_hyperparameters(num_trials=num_models)
best_models = tuner.get_best_models(num_models=num_models)
for m in range(num_models):
    values = best_hyperparameters[m].values
    score = best_models[m].evaluate(x_test, y_test, verbose=0)
    print(f'Model {m}: Hyperparameters: {values!s} Loss: {score[0]:.4} Precisión: {score[1]:.2%}')
```

```
Model 0: Hyperparameters: {'filters': 64} Loss: 0.03234 Precisión: 99.31%
Model 1: Hyperparameters: {'filters': 64} Loss: 0.03318 Precisión: 99.38%
Model 2: Hyperparameters: {'filters': 32} Loss: 0.03167 Precisión: 99.33%
Model 3: Hyperparameters: {'filters': 64} Loss: 0.03387 Precisión: 99.47%
Model 4: Hyperparameters: {'filters': 64} Loss: 0.03082 Precisión: 99.38%
Model 5: Hyperparameters: {'filters': 64} Loss: 0.0273 Precisión: 99.43%
Model 6: Hyperparameters: {'filters': 64} Loss: 0.03103 Precisión: 99.37%
Model 7: Hyperparameters: {'filters': 8} Loss: 0.03838 Precisión: 99.23%
Model 8: Hyperparameters: {'filters': 16} Loss: 0.03671 Precisión: 99.25%
Model 9: Hyperparameters: {'filters': 16} Loss: 0.03302 Precisión: 99.23%
```

Conclusión: precisión en test claramente mejor que la que teníamos

3 Fashion-MNIST

Ejercicio: realiza un experimento similar al de MNIST con Fashion-MNIST

Coste: el mismo experimento cuesta unas 4.5 horas, por lo que conviene reducir el coste de alguna forma

Fashion-MNIST: resumen de resultados

- MLP inicial: MLP con una capa oculta de 800 RELUs, batch size 16, 20 épocas; 88.0% en test
- Mejor arquitectura: una capa oculta de 800 RELUs, 89.0% en val, 88.3% en test (88.0% modelo val)
- Learning rate y batch size: ajustados a 0.00015 y 256; 89.6% en val, 89.8% en test (89.1% modelo val)
- ReduceLROnPlateau: factor 0.32 y paciencia 5; 90.0% en val, 89.6% en test (89.5% modelo val)

Inicialización: librerías, semilla, lectura de Fashion-MNIST **sin aplanar imágenes** y partición train-val-test

In [ ]:
```python
import numpy as np; import matplotlib.pyplot as plt
import os; os.environ['TF_CPP_MIN_LOG_LEVEL'] = '2'
import keras; import keras_tuner
keras.utils.set_random_seed(23); input_dim = (28, 28, 1); num_classes = 10
(x_train_val, y_train_val), (x_test, y_test) = keras.datasets.fashion_mnist.load_data()
x_train_val = x_train_val.astype("float32") / 255.0
x_test = x_test.astype("float32") / 255.0
x_train_val = np.expand_dims(x_train_val, -1)
x_test = np.expand_dims(x_test, -1)
print(x_train_val.shape, y_train_val.shape, x_test.shape, y_test.shape)
y_train_val = keras.utils.to_categorical(y_train_val, num_classes)
y_test = keras.utils.to_categorical(y_test, num_classes)
x_train = x_train_val[:-10000]; x_val = x_train_val[-10000:]
y_train = y_train_val[:-10000]; y_val = y_train_val[-10000:]
```

```
(60000, 28, 28, 1) (60000,) (10000, 28, 28, 1) (10000,)
```

MyHyperModel: exploramos número de filtros de la primera capa; doblamos en la segunda

In [ ]:
```
class MyHyperModel(keras_tuner.HyperModel):
    def build(self, hp):
        M = keras.Sequential()
        M.add(keras.Input(shape=(28, 28, 1)))
        filters = hp.Int("filters", min_value=8, max_value=64, step=2, sampling="log")
        M.add(keras.layers.Conv2D(filters, kernel_size=(3, 3), activation="relu"))
        M.add(keras.layers.MaxPooling2D(pool_size=(2, 2)))
        M.add(keras.layers.Conv2D(2*filters, kernel_size=(3, 3), activation="relu"))
        M.add(keras.layers.MaxPooling2D(pool_size=(2, 2)))
        M.add(keras.layers.Flatten())
        M.add(keras.layers.Dense(units=800, activation='relu'))
        M.add(keras.layers.Dense(10, activation='softmax'))
        opt = keras.optimizers.Adam(learning_rate=0.00015)
        M.compile(loss="categorical_crossentropy", optimizer=opt, metrics=["accuracy"])
        return M
    def fit(self, hp, M, x, y, xy_val, **kwargs):
        factor = 0.32; patience = 5
        reduce_cb = keras.callbacks.ReduceLROnPlateau(
            monitor='val_accuracy', factor=factor, patience=patience, min_delta=1e-4, min_lr=1e-5)
        early_cb = keras.callbacks.EarlyStopping(monitor='val_accuracy', patience=2*patience, min_delta=1e-5)
        kwargs['callbacks'].extend([reduce_cb, early_cb])
        return M.fit(x, y, batch_size=256, epochs=100, validation_data=xy_val, **kwargs)
```

Experimento: exploración y resumen de resultados

In [ ]:
```
tuner = keras_tuner.BayesianOptimization(
    MyHyperModel(), objective="val_accuracy", max_trials=10, executions_per_trial=1,
    overwrite=True, directory="/tmp", project_name="Fashion-MNIST")
```

In [ ]:
```
tuner.search(x_train, y_train, (x_val, y_val))
```

```
Trial 10 Complete [00h 38m 00s]
val_accuracy: 0.9203000068664551

Best val_accuracy So Far: 0.9222000241279602
Total elapsed time: 04h 32m 45s
```

In [ ]:
```
tuner.results_summary(num_trials=4)
```

```
Results summary
Results in /tmp/Fashion-MNIST
Showing 4 best trials
Objective(name="val_accuracy", direction="max")

Trial 03 summary
Hyperparameters:
filters: 64
Score: 0.9222000241279602

Trial 06 summary
Hyperparameters:
filters: 64
Score: 0.9218000173568726

Trial 02 summary
Hyperparameters:
filters: 64
Score: 0.9205999970436096

Trial 09 summary
Hyperparameters:
filters: 64
Score: 0.9203000068664551
```

Experimento (cont.): evaluación en test de los mejores modelos en validación

In [ ]:
```
num_models = 10
best_hyperparameters = tuner.get_best_hyperparameters(num_trials=num_models)
best_models = tuner.get_best_models(num_models=num_models)
for m in range(num_models):
    values = best_hyperparameters[m].values
    score = best_models[m].evaluate(x_test, y_test, verbose=0)
    print(f'Model {m}: Hyperparameters: {values!s} Loss: {score[0]:.4} Precisión: {score[1]:.2%}')
```

```
Model 0: Hyperparameters: {'filters': 64} Loss: 0.2649 Precisión: 91.62%
Model 1: Hyperparameters: {'filters': 64} Loss: 0.2455 Precisión: 91.64%
Model 2: Hyperparameters: {'filters': 64} Loss: 0.2511 Precisión: 91.19%
Model 3: Hyperparameters: {'filters': 64} Loss: 0.2476 Precisión: 91.47%
Model 4: Hyperparameters: {'filters': 64} Loss: 0.2451 Precisión: 91.66%
Model 5: Hyperparameters: {'filters': 64} Loss: 0.2532 Precisión: 91.36%
Model 6: Hyperparameters: {'filters': 64} Loss: 0.2508 Precisión: 91.33%
Model 7: Hyperparameters: {'filters': 32} Loss: 0.2521 Precisión: 91.23%
Model 8: Hyperparameters: {'filters': 16} Loss: 0.272 Precisión: 90.80%
Model 9: Hyperparameters: {'filters': 8} Loss: 0.2789 Precisión: 90.16%
```

Conclusión: precisión en test claramente mejor que la que teníamos