TL07 Regularización

Índice

1 Regularización

Propósito: evitar modelos sobreajustados modificando el comportamiento de descenso por gradiente, objetivo y datos

Técnica básica: evaluar la bondad de cualquier modificación mediante estimación del rendimiento teórico (en validación)

1.1 Descenso por gradiente

Idea intuitiva: queremos alcanzar mínimos profundos en regiones anchas, sin caer en mínimos estrechos

Terminación temprana: técnica sencilla muy conveniente computacionalmente

Learning rate constante: si es algo grande, resultará más difícil caer en mínimos estrechos

Planificador del learning rate: quizás con uno o más ciclos de aumento-decremento para evitar mínimos estrechos

ReduceLROnPlateau: planificador estándar; caída escalonada monitorizada en validación

Dropout: técnica muy efectiva y popular que evita el sobreentrenamiento de neuronas individuales

- **Clase Dropout:** https://keras.io/api/layers/regularization_layers/dropout
- Parámetro `rate`: probabilidad de fijar cada entrada a cero; las entradas no fijadas a cero se normalizan

1.2 Objetivo

Batch size: un batch size algo pequeño añade estocasticidad extra al objetivo y dificulta el sobreajuste

Penalización de pesos: técnica estándar para penalizar pesos demasiado grandes (en capas seleccionadas)

Clase Regularizer: https://keras.io/api/layers/regularizers

Penalizaciones estándar:

- `L1(penalty)`: penalización L1
- `L2(penalty)`: penalización L2
- `L1L2(l1=penalty_l1, L2=penalty_l2)`: penalización L1 + L2 o **Elastic net**

Parámetros estándar para penalizar pesos de Dense y Conv*D:

- `kernel_regularizer`: penalización del kernel
- `bias_regularizer`: penalización del sesgo
- `activity_regularizer`: penalización de la salida

1.3 Aumento de datos

Aumento de datos: el aumento de datos dificulta el sobreajuste (de modelos grandes)

Datos sintéticos: en general se obtienen buenos resultados perturbando adecuadamente los de entrenamiento

Capas de preproceso de imágenes: https://keras.io/api/layers/preprocessing_layers

- **Clase Resizing:** https://keras.io/api/layers/preprocessing_layers/image_preprocessing/resizing
- **Clase Rescaling:** https://keras.io/api/layers/preprocessing_layers/image_preprocessing/rescaling
- **Clase CenterCrop:** https://keras.io/api/layers/preprocessing_layers/image_preprocessing/center_crop

Capas de aumento de imágenes: https://keras.io/api/layers/preprocessing_layers/image_augmentation

- **Clase RandomCrop:** https://keras.io/api/layers/preprocessing_layers/image_augmentation/random_crop
- **Clase RandomFlip:** https://keras.io/api/layers/preprocessing_layers/image_augmentation/random_flip
- **Clase RandomTranslation:** https://keras.io/api/layers/preprocessing_layers/image_augmentation/random_translation
- **Clase RandomRotation:** https://keras.io/api/layers/preprocessing_layers/image_augmentation/random_rotation
- **Clase RandomZoom:** https://keras.io/api/layers/preprocessing_layers/image_augmentation/random_zoom

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)
- Dos pares Conv2D-MaxPooling2D: $64 + 128$ filtros 3×3; 99.4% en val, 99.3% en test (del mejor modelo en val)

Inicialización: librerías, semilla, lectura de MNIST **sin normalización** 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")
x_test = x_test.astype("float32")
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 aumento de datos (rotación, translación y zoom) y dropout 0.5

In [ ]:
```
class MyHyperModel(keras_tuner.HyperModel):
    def build(self, hp):
        M = keras.Sequential()
        M.add(keras.Input(shape=(28, 28, 1)))
        factor = hp.Float("factor", min_value=0.01, max_value=0.3, step=2, sampling="log")
        M.add(keras.layers.RandomRotation(factor, fill_mode="nearest"))
        M.add(keras.layers.RandomTranslation(factor, factor, fill_mode="nearest"))
        M.add(keras.layers.RandomZoom(factor, fill_mode="nearest"))
        M.add(keras.layers.Rescaling(1./255))
        filters = 64
        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'))
        # dropout = hp.Float("dropout", min_value=0.0, max_value=0.5, step=0.1)
        dropout = 0.5
        M.add(keras.layers.Dropout(dropout))
        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 14m 00s]
val_accuracy: 0.9882000088691711

Best val_accuracy So Far: 0.9959999918937683
Total elapsed time: 03h 11m 50s
```

In [ ]:
```
tuner.results_summary(num_trials=3)
```

```
Results summary
Results in /tmp/MNIST
Showing 3 best trials
Objective(name="val_accuracy", direction="max")

Trial 00 summary
Hyperparameters:
factor: 0.04
Score: 0.9959999918937683

Trial 01 summary
Hyperparameters:
factor: 0.08
Score: 0.995199978351593

Trial 08 summary
Hyperparameters:
factor: 0.04
Score: 0.9950000047683716
```

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: {'factor': 0.04} Loss: 0.01629 Precisión: 99.52%
Model 1: Hyperparameters: {'factor': 0.08} Loss: 0.0176 Precisión: 99.43%
Model 2: Hyperparameters: {'factor': 0.04} Loss: 0.02011 Precisión: 99.45%
Model 3: Hyperparameters: {'factor': 0.02} Loss: 0.02657 Precisión: 99.45%
Model 4: Hyperparameters: {'factor': 0.02} Loss: 0.03 Precisión: 99.35%
Model 5: Hyperparameters: {'factor': 0.02} Loss: 0.02545 Precisión: 99.43%
Model 6: Hyperparameters: {'factor': 0.04} Loss: 0.01838 Precisión: 99.47%
Model 7: Hyperparameters: {'factor': 0.08} Loss: 0.02139 Precisión: 99.37%
Model 8: Hyperparameters: {'factor': 0.16} Loss: 0.04583 Precisión: 98.60%
Model 9: Hyperparameters: {'factor': 0.16} Loss: 0.04056 Precisión: 98.79%
```

Conclusión: precisión en test un poco mejor que la que teníamos

3 Fashion-MNIST

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

Coste: el mismo experimento cuesta más de 6 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)
- Dos pares Conv2D-MaxPooling2D: $64 + 128$ filtros 3×3; 92.2% en val, 91.6% en test (del mejor modelo en val)

Inicialización: librerías, semilla, lectura de Fashion-MNIST **sin normalización** 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.fashion_mnist.load_data()
x_train_val = x_train_val.astype("float32")
x_test = x_test.astype("float32")
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 aumento de datos (rotación, translación y zoom) y dropout 0.5

In [ ]:
```
class MyHyperModel(keras_tuner.HyperModel):
    def build(self, hp):
        M = keras.Sequential()
        M.add(keras.Input(shape=(28, 28, 1)))
        factor = hp.Float("factor", min_value=0.01, max_value=0.3, step=2, sampling="log")
        M.add(keras.layers.RandomRotation(factor, fill_mode="nearest"))
        M.add(keras.layers.RandomTranslation(factor, factor, fill_mode="nearest"))
        M.add(keras.layers.RandomZoom(factor, fill_mode="nearest"))
        M.add(keras.layers.Rescaling(1./255))
        filters = 64
        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'))
        # dropout = hp.Float("dropout", min_value=0.0, max_value=0.5, step=0.1)
        dropout = 0.5
        M.add(keras.layers.Dropout(dropout))
        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 48m 40s]
val_accuracy: 0.9253000020980835

Best val_accuracy So Far: 0.9262999892234802
Total elapsed time: 06h 26m 49s
```

In [ ]:
```
tuner.results_summary(num_trials=3)
```

```
Results summary
Results in /tmp/Fashion-MNIST
Showing 3 best trials
Objective(name="val_accuracy", direction="max")

Trial 05 summary
Hyperparameters:
factor: 0.01
Score: 0.9262999892234802

Trial 09 summary
Hyperparameters:
factor: 0.01
Score: 0.9253000020980835

Trial 08 summary
Hyperparameters:
factor: 0.01
Score: 0.9244999885559082
```

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: {'factor': 0.01} Loss: 0.2657 Precisión: 92.04%
Model 1: Hyperparameters: {'factor': 0.01} Loss: 0.2605 Precisión: 91.75%
Model 2: Hyperparameters: {'factor': 0.01} Loss: 0.2705 Precisión: 91.89%
Model 3: Hyperparameters: {'factor': 0.01} Loss: 0.2545 Precisión: 91.86%
Model 4: Hyperparameters: {'factor': 0.01} Loss: 0.2602 Precisión: 91.61%
Model 5: Hyperparameters: {'factor': 0.01} Loss: 0.2539 Precisión: 91.68%
Model 6: Hyperparameters: {'factor': 0.02} Loss: 0.2567 Precisión: 91.42%
Model 7: Hyperparameters: {'factor': 0.04} Loss: 0.2592 Precisión: 91.05%
Model 8: Hyperparameters: {'factor': 0.08} Loss: 0.3732 Precisión: 87.11%
Model 9: Hyperparameters: {'factor': 0.16} Loss: 0.5717 Precisión: 79.63%
```

Conclusión: precisión en test un poco mejor que la que teníamos