TL05 LearningRateSchedule

Índice

1 LearningRateSchedule

ReduceLROnPlateau: planificador estándar implementado como callback

- **API callbacks:** https://keras.io/api/callbacks
- Modifica el learning rate cuando no mejora la métrica monitorizada en validación
- El planificador propiamente dicho se fija en compilación

Clase LearningRateSchedule: planificadores alternativos al learning rate "constante"

- **API Learning rate schedules:** https://keras.io/api/optimizers/learning_rate_schedules
- ExponentialDecay, PiecewiseConstantDecay, PolynomialDecay, InverseTimeDecay, CosineDecay, CosineDecayRestarts
- Se quiere hallar un "buen" mínimo y aproximarlo bien
- Algunos heurísticos establecen uno o más ciclos de aumento-decremento como forma de regularización
- El efecto del planificador depende en gran medida del optimzador escogido (SGD, Adam, AdamW)
- Por su relativa simiplicidad, SGD es un optimizador adecuado para comparar planificadores

Ejemplo: la sección 2 incluye un ejemplo de uso para MNIST

Examen: la sección 3 describe el ejercicio a realizar, similar al ejemplo, pero con Fashion-MNIST

2 MNIST con PolynomialDecay

MNIST: resumen de resultados (con Adam)

- 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)

MNIST con SGD: 98.16% en test con `learning_rate=0.3168`, `momentum=0.1134` y `nesterov=False`

Inicialización: librerías, semilla, lectura de MNIST 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 = 784; num_classes = 10
(x_train_val, y_train_val), (x_test, y_test) = keras.datasets.mnist.load_data()
x_train_val = x_train_val.reshape(-1, input_dim).astype("float32") / 255.0
x_test = x_test.reshape(-1, input_dim).astype("float32") / 255.0
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:]
```

MyHyperModel: exploramos learning rate inicial y final, con 10 decay_steps y power por omisión

In [ ]:
```python
class MyHyperModel(keras_tuner.HyperModel):
    def build(self, hp):
        M = keras.Sequential()
        M.add(keras.Input(shape=(784,)))
        M.add(keras.layers.Dense(units=800, activation='relu'))
        M.add(keras.layers.Dense(10, activation='softmax'))
        initial_learning_rate = hp.Float("initial_learning rate", min_value=0.3168, max_value=0.3400)
        end_learning_rate = hp.Float("end_learning rate", min_value=0.3000, max_value=0.3168)
        decay_steps = 10
        lr_schedule = keras.optimizers.schedules.PolynomialDecay(
            initial_learning_rate, decay_steps, end_learning_rate)
        opt = keras.optimizers.SGD(learning_rate=lr_schedule)
        M.compile(loss="categorical_crossentropy", optimizer=opt, metrics=["accuracy"])
        return M
    def fit(self, hp, M, x, y, xy_val, **kwargs):
        patience = 10
        early_cb = keras.callbacks.EarlyStopping(monitor='val_accuracy', patience=2*patience, min_delta=0.0)
        kwargs['callbacks'].append(early_cb)
        return M.fit(x, y, batch_size=256, epochs=100, validation_data=xy_val, **kwargs)
```

Experimento: exploración y evaluación en test del mejor modelo en validación

```
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 01m 04s]
val_accuracy: 0.9821000099182129

Best val_accuracy So Far: 0.9832000136375427
Total elapsed time: 00h 11m 24s
```

```
In [ ]: tuner.results_summary(num_trials=1)
```

```
Results summary
Results in /tmp/MNIST
Showing 1 best trials
Objective(name="val_accuracy", direction="max")

Trial 03 summary
Hyperparameters:
initial_learning rate: 0.334588319033129
end_learning rate: 0.3100941646365156
Score: 0.9832000136375427
```

```
In [ ]: best = tuner.get_best_models(num_models=1)[0]
        score = best.evaluate(x_test, y_test, verbose=0)
        print(f'Loss: {score[0]:.4}\nPrecisión: {score[1]:.2%}')
```

```
Loss: 0.06383
Precisión: 98.14%
```

Conclusión: precisión en test similar a la que la que teníamos con learning rate constante (y ReduceLROnPlateau)

3 Fashion-MNIST con PolynomialDecay

Fashion-MNIST:

- 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)

Fashion-MNIST con SGD: 89.5% en test con `learning_rate=0.2983`, `momentum=0.1104` y `nesterov=True`

Inicialización: librerías, semilla, lectura de MNIST 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 = 784; 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.reshape(-1, input_dim).astype("float32") / 255.0
x_test = x_test.reshape(-1, input_dim).astype("float32") / 255.0
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:]
```

MyHyperModel: exploramos learning rate inicial y final, con 10 decay_steps y power por omisión

In [ ]:
```python
class MyHyperModel(keras_tuner.HyperModel):
    def build(self, hp):
        M = keras.Sequential()
        M.add(keras.Input(shape=(784,)))
        M.add(keras.layers.Dense(units=800, activation='relu'))
        M.add(keras.layers.Dense(10, activation='softmax'))
        initial_learning_rate = hp.Float("initial_learning rate", min_value=0.2983, max_value=0.3200)
        end_learning_rate = hp.Float("end_learning rate", min_value=0.2800, max_value=0.2983)
        decay_steps = 10
        lr_schedule = keras.optimizers.schedules.PolynomialDecay(
            initial_learning_rate, decay_steps, end_learning_rate)
        opt = keras.optimizers.SGD(learning_rate=lr_schedule)
        M.compile(loss="categorical_crossentropy", optimizer=opt, metrics=["accuracy"])
        return M
    def fit(self, hp, M, x, y, xy_val, **kwargs):
        patience = 10
        early_cb = keras.callbacks.EarlyStopping(monitor='val_accuracy', patience=2*patience, min_delta=0.0)
        kwargs['callbacks'].append(early_cb)
        return M.fit(x, y, batch_size=256, epochs=100, validation_data=xy_val, **kwargs)
```

Experimento: exploración y evaluación en test del mejor modelo en validación

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 01m 14s]
val_accuracy: 0.8989999890327454

Best val_accuracy So Far: 0.8992000222206116
Total elapsed time: 00h 11m 28s
```

In [ ]:
```
tuner.results_summary(num_trials=1)
```

```
Results summary
Results in /tmp/Fashion-MNIST
Showing 1 best trials
Objective(name="val_accuracy", direction="max")

Trial 06 summary
Hyperparameters:
initial_learning rate: 0.30349711132185786
end_learning rate: 0.28079857268647473
Score: 0.8992000222206116
```

In [ ]:
```
best = tuner.get_best_models(num_models=1)[0]
score = best.evaluate(x_test, y_test, verbose=0)
print(f'Loss: {score[0]:.4}\nPrecisión: {score[1]:.2%}')
```

```
Loss: 0.3802
Precisión: 88.83%
```

Conclusión: precisión en test menor que la que teníamos con learning rate constante (y ReduceLROnPlateau)