Taller de Aprendizaje Automático

Selección de hiperparámetros en redes neuronales

Instituto de Ingeniería Eléctrica
Facultad de Ingeniería

UNIVERSIDAD
DE LA REPÚBLICA
URUGUAY

1. Selección de hiperparámetros
2. Implementación con Scikit-learn y Keras
3. Optimización Bayesiana
4. Implementación con Optuna

Selección de hiperparámetros

hiperparámetro

parámetro que controla el proceso de aprendizaje

todo parámetro que no se determina durante el aprendizaje

redes neuronales

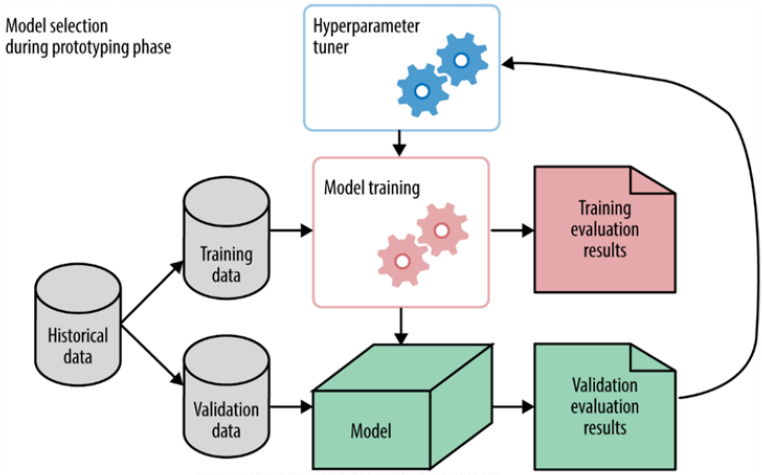
número de capas, número de neuronas por capa, tipo de función de activación, tipo de inicialización, tasa de aprendizaje, tamaño del lote, optimizador, etc.

selección de modelos

función de costo (loss)

validación cruzada, hold-out validation set

Selección de hiperparámetros



Selección de hiperparámetros

Grid search

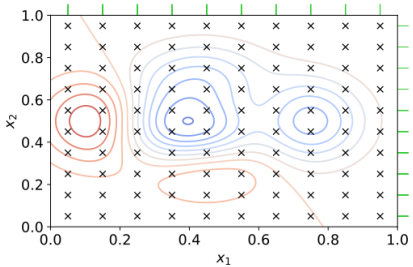
búsqueda exhaustiva en una grilla

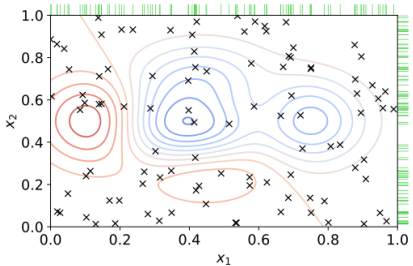
Random search

búsqueda aleatoria de combinaciones

prueba más valores de cada parámetro

- necesario establecer rangos y discretizar
- computacionalmente costo (paralelizar)
- búsqueda gruesa seguida de búsqueda fina

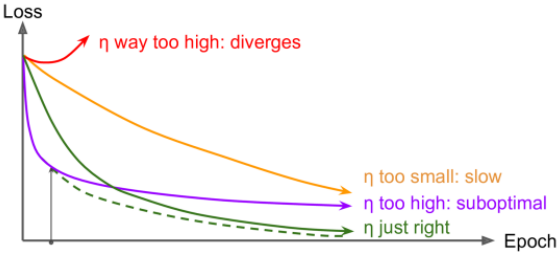




Selección de hiperparámetros

tasa de aprendizaje

η es probablemente el hiperparámetro más importante



un valor razonable η^*

1. entrenar algunos cientos de iteraciones incrementando $\eta : 10^{-6} \ldots 10$
2. identificar el valor en el que la curva comienza a dispararse
3. volver a entrenar con un valor (constante) algo menor

Selección de hiperparámetros

número de capas ocultas

- para problemas sencillos es suficiente con unas pocas
- incrementar las capas ocultas hasta comenzar a sobreajustar
- reutilizar capas pre-entrenadas (*transfer learning*)

número de neuronas por capa

- número decreciente al aumentar profundidad (e.g. 300, 200, 100)
- la misma cantidad de neuronas por capa da igual (o mejor) desempeño
- sobredimensionar y aplicar regularización (e.g early stopping)

Implementación con Scikit-learn y Keras

```
import tensorflow as tf
from tensorflow import keras
from scikeras.wrappers import KerasRegressor

def build_model(n_hidden=1, n_neurons=30, learning_rate=3e-3, input_shape=[8]):
    model = keras.models.Sequential()
    model.add(keras.layers.InputLayer(input_shape=input_shape))
    for layer in range(n_hidden):
        model.add(keras.layers.Dense(n_neurons, activation="relu"))
    model.add(keras.layers.Dense(1))
    optimizer = keras.optimizers.SGD(lr=learning_rate)
    model.compile(loss="mse", optimizer=optimizer)
    return model

keras_reg = KerasRegressor(build_model)

keras_reg.fit(X_train, y_train, epochs=100,
              validation_data=(X_valid, y_valid),
              callbacks=[keras.callbacks.EarlyStopping(patience=10)])
```

`KerasRegressor` es un wrapper para usar el modelo desde Scikit-Learn.

Implementación con Scikit-learn y Keras

Seleccionamos hiperparámetros con `RandomizedSearchCV` de Scikit-Learn.

```
from scipy.stats import reciprocal
from sklearn.model_selection import RandomizedSearchCV

param_distribs = {
    "n_hidden": [0, 1, 2, 3],
    "n_neurons": np.arange(1, 100),
    "learning_rate": reciprocal(3e-4, 3e-2),
}

rnd_search_cv = RandomizedSearchCV(keras_reg, param_distribs, n_iter=10, cv=3, verbose=2)
rnd_search_cv.fit(X_train, y_train, epochs=100,
                  validation_data=(X_valid, y_valid),
                  callbacks=[keras.callbacks.EarlyStopping(patience=10)])
```

Callbacks en Keras

`fit` acepta un argumento `callbacks` para especificar funciones que se invocan durante el entrenamiento (al inicio, al final, para cada batch o cada época)

```
checkpoint_cb = keras.callbacks.ModelCheckpoint("my_keras_model.h5")
history = model.fit(X_train, y_train, epochs=10, callbacks=[checkpoint_cb])
```

```
checkpoint_cb = keras.callbacks.ModelCheckpoint("my_keras_model.h5", save_best_only=True)
history = model.fit(X_train, y_train, epochs=10,
                    validation_data=(X_valid, y_valid),
                    callbacks=[checkpoint_cb])
model = keras.models.load_model("my_keras_model.h5") # rollback to best model
```

```
early_stopping_cb = keras.callbacks.EarlyStopping(patience=10,
                                                  restore_best_weights=True)
history = model.fit(X_train, y_train, epochs=100,
                    validation_data=(X_valid, y_valid),
                    callbacks=[checkpoint_cb, early_stopping_cb])
```

Optimización Bayesiana[†]

$$\mathbf{x}^* = \arg\max_{\mathbf{x} \in \mathcal{X}} f(\mathbf{x})$$

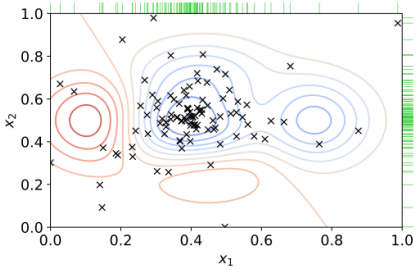
- observaciones lo más informativas posibles

$$y_i = f(\mathbf{x}_i) \quad \mathcal{D} = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_i, y_i)\}$$

- modelo probabilístico sustituto $\mathcal{M}$

$$p(y|\mathbf{x}, \mathcal{D})$$

- iterativamente:
  - evaluar configuración de hiperparámetros
  - actualizar el modelo (regla de Bayes)
  - elegir nueva configuración a evaluar



Algorithm 1 Sequential Model-Based Optimization

Input: $f, \mathcal{X}, S, \mathcal{M}$
$\mathcal{D} \leftarrow \text{INITSAMPLES}(f, \mathcal{X})$
for $i \leftarrow |\mathcal{D}|$ **to** T **do**
  $p(y \mid \mathbf{x}, \mathcal{D}) \leftarrow \text{FITMODEL}(\mathcal{M}, \mathcal{D})$
  $\mathbf{x}_i \leftarrow \arg\max_{\mathbf{x} \in \mathcal{X}} S(\mathbf{x},\ p(y \mid \mathbf{x}, \mathcal{D}))$
  $y_i \leftarrow f(\mathbf{x}_i)$ ▷ Expensive step
  $\mathcal{D} \leftarrow \mathcal{D} \cup (\mathbf{x}_i, y_i)$
end for

[†] A. Agnihotri and N. Batra, "Exploring bayesian optimization," *Distill*, 2020. https://distill.pub/2020/bayesian-optimization

Optimización Bayesiana[†]

Modelo probabilístico

Gaussian processes

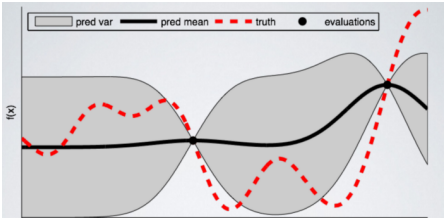
$$f \sim \mathcal{GP}(\mu, K)$$

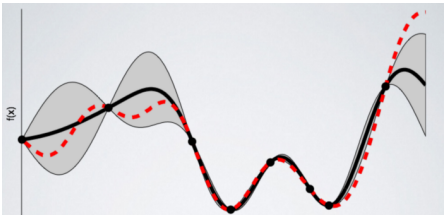
Función de adquisición S

Expected Improvement

$$EI(\mathbf{x}|\mathcal{D}) = \int_{f^*}^{\infty} (y - f^*) p(y|\mathbf{x}, \mathcal{D}) \, \mathrm{d}y$$

balance entre exploración y explotación





[†] A. Agnihotri and N. Batra, "Exploring bayesian optimization," *Distill*, 2020. https://distill.pub/2020/bayesian-optimization

Implementación con Optuna[‡]

study

optimización basada en una función objetivo

trial

una ejecución de la función objetivo

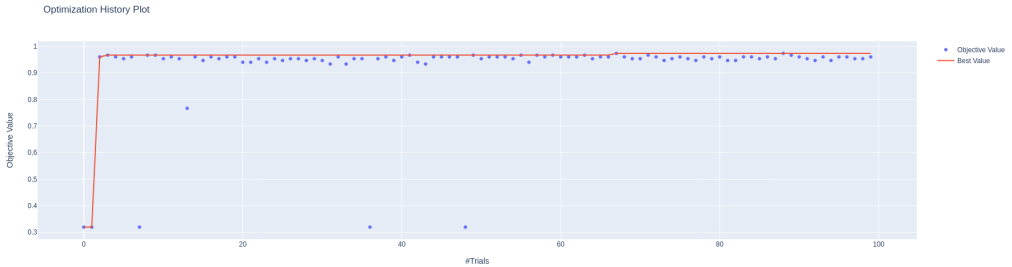
```
import optuna

def objective(trial):
    x = trial.suggest_uniform('x', -10, 10)
    return (x - 2) ** 2

study = optuna.create_study()
study.optimize(objective, n_trials=100)

study.best_params  # E.g. {'x': 2.002108042}
```

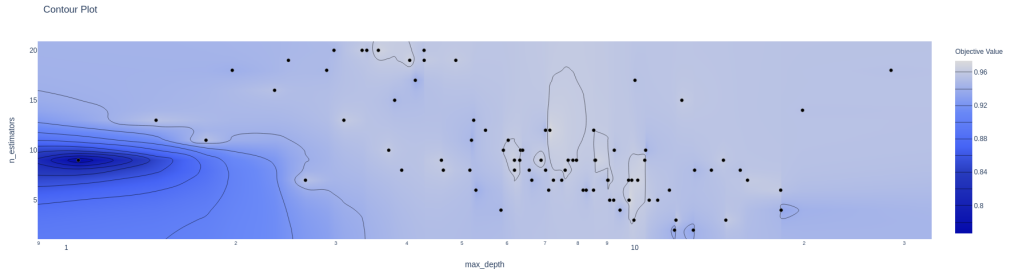
[‡] T. Akiba, S. Sano, T. Yanase, T. Ohta, and M. Koyama, "Optuna: A next-generation hyperparameter optimization framework," in *Proceedings of the 25rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2019

Implementación con Optuna y Scikit-learn

```
import sklearn
import optuna

# 1. Define an objective function to be maximized.
def objective(trial):

    # 2. Suggest values for the hyperparameters using a trial object.
    classifier_name = trial.suggest_categorical('classifier', ['SVC', 'RandomForest'])
    if classifier_name == 'SVC':
         svc_c = trial.suggest_loguniform('svc_c', 1e-10, 1e10)
         classifier_obj = sklearn.svm.SVC(C=svc_c, gamma='auto')
    else:
        rf_max_depth = int(trial.suggest_loguniform('rf_max_depth', 2, 32))
        classifier_obj = sklearn.ensemble.RandomForestClassifier(max_depth=rf_max_depth,
                                                                 n_estimators=10)
    ...
    return accuracy

# 3. Create a study object and optimize the objective function.
study = optuna.create_study(direction='maximize')
study.optimize(objective, n_trials=100)
```

Implementación con Optuna y Scikit-learn





Implementación con Optuna y Keras

```
import keras
import optuna

# 1. Define an objective function to be maximized.
def objective(trial):
    model = Sequential()
    # 2. Suggest values of the hyperparameters using a trial object.
    model.add(
        Conv2D(filters=trial.suggest_categorical('filters', [32, 64]),
               kernel_size=trial.suggest_categorical('kernel_size', [3, 5]),
               strides=trial.suggest_categorical('strides', [1, 2]),
               activation=trial.suggest_categorical('activation', ['relu', 'linear']),
               input_shape=input_shape))
    model.add(Flatten())
    model.add(Dense(CLASSES, activation='softmax'))
    # We compile our model with a sampled learning rate.
    lr = trial.suggest_loguniform('lr', 1e-5, 1e-1)
    model.compile(loss='sparse_categorical_crossentropy', optimizer=RMSprop(lr=lr), metrics=['accuracy'])
    ...
    return accuracy
# 3. Create a study object and optimize the objective function.
study = optuna.create_study(direction='maximize')
study.optimize(objective, n_trials=100)
```

Ejercicio

`TensorBoard` es una herramienta interactiva para inspeccionar el entrenamiento y los modelos. `Keras` proporciona una funcion callback `TensorBoard()` para guardar información durante el entrenamiento. Estudie la forma de usarla y acceder a la información desde `TensorBoard`.

Referencias


A. Agnihotri and N. Batra, "Exploring bayesian optimization," *Distill*, 2020. https://distill.pub/2020/bayesian-optimization.


T. Akiba, S. Sano, T. Yanase, T. Ohta, and M. Koyama, "Optuna: A next-generation hyperparameter optimization framework," in *Proceedings of the 25rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2019.


A. Géron, *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow, 3rd Edition*. O'Reilly Media, Inc., 2022.


B. Shahriari, K. Swersky, Z. Wang, R. P. Adams, and N. de Freitas, "Taking the human out of the loop: A review of bayesian optimization," *Proceedings of the IEEE*, vol. 104, no. 1, pp. 148–175, 2016.