Module 11: Regularization and Hyperparameter Tuning

Overview

All machine learning models have hyperparameters—settings chosen before training that control model complexity and behavior. This module covers regularization techniques that prevent overfitting and systematic methods for tuning hyperparameters. You will learn L1/L2 regularization, dropout, early stopping, cross-validation, grid search, and how to build robust ML pipelines.

---

1. The Overfitting Problem

Symptoms

- Training accuracy much higher than validation accuracy
- Validation loss increases while training loss decreases
- Model memorizes training data instead of learning patterns

Solutions Overview

Technique	How it helps
L1/L2 Regularization	Penalizes large weights
Dropout	Prevents co-adaptation
Early Stopping	Stops before overfitting
Data Augmentation	More effective training data
Simpler Architecture	Fewer parameters

---

2. L2 Regularization (Ridge)

The Idea

Add penalty for large weights to the loss function:

$$\text{Loss\_regularized} = \text{Loss\_original} + \quad \times \Sigma(w^2)$$

Effect

- Shrinks all weights toward zero
- Large weights are penalized heavily
- Prevents any single feature from dominating
- λ (lambda) controls regularization strength

Implementation

```
# scikit-learn
from sklearn.linear_model import Ridge
```

```
model = Ridge(alpha=1.0)  # alpha is

# Keras
layers.Dense(64, kernel_regularizer='l2')
```

---

3. L1 Regularization (Lasso)

The Idea

Penalize the absolute value of weights:

```
Loss_regularized = Loss_original +   × Σ|w|
```

Effect

- Encourages sparse solutions
- Some weights become exactly zero
- Automatic feature selection
- Useful when many features are irrelevant

L1 vs L2 Comparison

L1 (Lasso)	L2 (Ridge)
Sparse weights	All weights non-zero
Feature selection	Feature shrinking
Corners in optimization	Smooth optimization

Elastic Net

Combines L1 and L2:

```
Loss +  × ×Σ|w| + (1− )× ×Σw²
```

---

4. Dropout

The Idea

During training, randomly set activations to zero with probability p.

Effect

- Prevents neurons from co-adapting
- Ensemble effect (training many sub-networks)
- Strong regularization for deep networks

Implementation

```
# Keras
model = keras.Sequential([
    layers.Dense(256, activation='relu'),
    layers.Dropout(0.5),  # 50% dropout
    layers.Dense(128, activation='relu'),
    layers.Dropout(0.3),
    layers.Dense(10, activation='softmax')
])
```

Best Practices

- Typical rates: 0.1-0.5
- Higher dropout for larger layers
- Disable during inference (automatic in Keras)

---

5. Early Stopping

The Idea

Monitor validation loss during training and stop when it stops improving.

Implementation

```
early_stop = keras.callbacks.EarlyStopping(
    monitor='val_loss',
    patience=10,  # Wait 10 epochs for improvement
    restore_best_weights=True
)

model.fit(X_train, y_train,
          validation_split=0.2,
          callbacks=[early_stop],
          epochs=1000)  # Will stop early
```

Benefits

- Automatic regularization
- No need to guess number of epochs
- Saves best model automatically

---

6. Cross-Validation

The Problem

Single train/validation split may be unrepresentative.

K-Fold Cross-Validation

1. Split data into k equal folds
2. For each fold:
   - Use that fold as validation
   - Train on remaining k-1 folds
   - Record performance
3. Average performance across all folds

Benefits

- More robust performance estimate
- Uses all data for both training and validation
- Variance in estimate indicates stability

Implementation

```
from sklearn.model_selection import cross_val_score

scores = cross_val_score(model, X, y, cv=5)
print(f"Mean: {scores.mean():.3f}, Std: {scores.std():.3f}")
```

---

7. Hyperparameter Tuning

Parameters vs Hyperparameters

Parameters	Hyperparameters
Learned during training	Set before training
Weights, biases	Learning rate, layers
Optimized by gradient descent	Tuned by search

Common Hyperparameters

- Learning rate
- Regularization strength (λ)
- Number of layers/units
- Dropout rate
- Batch size
- Number of trees (ensembles)

---

8. Grid Search

The Idea

Try all combinations of hyperparameter values.

Example

```
from sklearn.model_selection import GridSearchCV

param_grid = {
    'C': [0.1, 1, 10],
    'kernel': ['linear', 'rbf'],
    'gamma': [0.01, 0.1]
}

grid = GridSearchCV(SVC(), param_grid, cv=5)
grid.fit(X_train, y_train)

print(f"Best params: {grid.best_params_}")
print(f"Best score: {grid.best_score_:.3f}")
```

Tradeoffs

- Exhaustive: tries everything
- Expensive: combinations grow exponentially
- Good for small parameter spaces

---

9. Random Search

The Idea

Sample random combinations instead of trying all.

Benefits

- More efficient for large spaces
- Often finds good solutions faster
- Works with continuous distributions

Implementation

```
from sklearn.model_selection import RandomizedSearchCV
from scipy.stats import uniform, loguniform

param_dist = {
    'C': loguniform(0.01, 100),
```

```
    'gamma': loguniform(0.001, 1)
}

random_search = RandomizedSearchCV(
    SVC(), param_dist, n_iter=20, cv=5
)
random_search.fit(X_train, y_train)
```

---

10. Scikit-learn Pipelines

Why Pipelines?

- Chain preprocessing and modeling
- Ensure test data processed correctly
- Simplify cross-validation
- Prevent data leakage

Example

```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler

pipeline = Pipeline([
    ('scaler', StandardScaler()),
    ('classifier', LogisticRegression())
])

# Cross-validation now includes scaling
scores = cross_val_score(pipeline, X, y, cv=5)
```

With Hyperparameter Tuning

```
param_grid = {
    'classifier__C': [0.1, 1, 10],
    'classifier__penalty': ['l1', 'l2']
}

grid = GridSearchCV(pipeline, param_grid, cv=5)
```

Key Takeaways

1. **Overfitting** happens when models memorize training data
2. **L2 regularization** shrinks all weights toward zero
3. **L1 regularization** creates sparse solutions (feature selection)
4. **Dropout** randomly zeros activations during training

5. **Early stopping** monitors validation loss and stops training
6. **Cross-validation** provides robust performance estimates
7. **Grid search** is exhaustive; **random search** is efficient
8. **Pipelines** ensure correct preprocessing during CV

Connections to Other Modules

- **Module 4**: Regularization for logistic regression
- **Module 6-10**: Regularization for neural networks
- **Module 12**: Validation for fairness evaluation

Further Reading

- [Links to be added]
- scikit-learn documentation: Model Selection
- Keras callbacks documentation