Hyperparameter Tuning with SageMaker

Module 18: Introduction to Hyperparameter Tuning

What are Hyperparameters?

In machine learning, models learn parameters from the data itself (like weights and biases in a neural network). **Hyperparameters**, on the other hand, are external configuration settings that are not learned from the data. They are set before the training process begins and control the very nature of the training algorithm.

Common Examples of Hyperparameters:

- **For Neural Networks:** Learning rate, number of hidden layers, number of neurons per layer, batch size, dropout rate.
- **For Tree-based Models (XGBoost, Random Forest):** Number of trees, maximum depth of a tree, minimum samples per leaf, learning rate (boosting).
- **For SVM (Support Vector Machines):** Regularization parameter (C), kernel type (linear, RBF), gamma.

Why Does Hyperparameter Tuning Matter?

Choosing the right hyperparameters is crucial because they directly impact model performance (e.g., accuracy, precision, recall). The default values provided by libraries are often not optimal for your specific dataset and problem.

- **The "No Free Lunch" Theorem:** There is no single set of hyperparameters that works best for all problems. The optimal set must be discovered empirically.
- **Impact on Performance:** A poorly chosen learning rate can cause a model to converge too slowly, diverge, or get stuck in a local minimum. The wrong model complexity (e.g., tree depth) can lead to severe underfitting or overfitting.
- **Manual Tuning is Inefficient:** Manually trying different combinations is a time-consuming, tedious, and often suboptimal process. It doesn't scale and is difficult to reproduce.

The Core Problem: Searching the Hyperparameter Space

The goal of hyperparameter tuning is to find the combination of hyperparameters that results in the best model performance, as measured by a predefined metric (e.g., validation accuracy). This combination is a point in a multi-dimensional **hyperparameter space**.

- **Search Space:** The defined range of possible values for each hyperparameter (e.g., `learning_rate`: from `0.01` to `0.2`).
- **Objective Metric:** The metric you want to optimize (e.g., `Validation-Accuracy` to maximize, or `Validation-Loss` to minimize).

The challenge is that evaluating a single point in this space requires training an entire model, which can be computationally expensive and slow. Therefore, we need efficient strategies to explore this space.

Module 19: Hands-on with Grid and Random Search

Strategy 1: Grid Search

Concept: Grid Search is a traditional and exhaustive method. It works by defining a "grid" of hyperparameters where each axis represents a hyperparameter, and the points on the grid are the specific values to try.

- **How it works:** It performs an exhaustive search over *every single combination* of the predefined hyperparameter values.
- **Example:**
  - Hyperparameter 1: `learning_rate` = [0.01, 0.1, 0.2]
  - Hyperparameter 2: `max_depth` = [3, 5, 7]
  - The grid search would train **3 * 3 = 9** separate models.

Pros:

- Simple to understand and implement.
- Embarrassingly parallel (each training job is independent).
- Guaranteed to find the best point *within the predefined grid*.

Cons:

- **Suffers from the "Curse of Dimensionality":** The number of required training jobs grows exponentially with the number of hyperparameters. For 5 hyperparameters with 5 values each, you need 5^5 = 3,125 training jobs. This becomes computationally prohibitive.

Strategy 2: Random Search

Concept: Random Search addresses a key weakness of Grid Search. Instead of searching exhaustively, it randomly samples a fixed number of hyperparameter combinations from the defined search space.

- **How it works:** You specify the number of total jobs (`max_jobs`), and SageMaker will randomly pick a combination of hyperparameters for each job.
- **Bergstra & Bengio's Insight (2012):** For most real-world problems, only a few hyperparameters truly matter. Random search explores the space more effectively because it doesn't waste resources on "unimportant" dimensions, unlike grid search which searches all dimensions uniformly.

Pros:

- Much more efficient than grid search for high-dimensional spaces.
- Often finds a good combination with far fewer training jobs.
- Easy to implement and parallelize.

Cons:

- There is no guarantee of finding the absolute best combination.
- It can still be inefficient as it does not use information from past trials to inform the next one.

Using SageMaker Automatic Model Tuning

SageMaker's **Automatic Model Tuning** service handles the orchestration of both Grid and Random Search (as well as the more advanced Bayesian optimization).

Key Components in a SageMaker Tuning Job:

1. **HyperparameterRanges:** A JSON object defining the name, type (`Integer`, `Continuous`, `Categorical`), and range/min/max for each hyperparameter.
2. **ObjectiveMetricName:** The name of the metric your training script logs that the tuner will optimize (e.g., `validation:accuracy`).

3. **MetricDefinitions:** How the tuner should parse the metric from your training job's CloudWatch logs.
4. **Strategy:** Set to `"Grid"` or `"Random"`.
5. **ResourceLimits:** `MaxNumberOfTrainingJobs` and `MaxParallelTrainingJobs` (for controlling parallelism and cost).

Example Snippet (Random Search with Boto3):

```
from sagemaker.tuner import HyperparameterTuner, RandomSearch
from sagemaker.hyperparameters import HyperparameterRanges, ContinuousParameter, IntegerParameter

# Define the search ranges
hyperparameter_ranges = {
    'learning-rate': ContinuousParameter(0.01, 0.2),
    'max-depth': IntegerParameter(3, 10)
}

# Instantiate the Tuner
tuner = HyperparameterTuner(
    estimator=my_sagemaker_estimator,
    objective_metric_name='validation:accuracy',
    hyperparameter_ranges=hyperparameter_ranges,
    metric_definitions=[{'Name': 'validation:accuracy', 'Regex': 'Validation-accuracy: ([0-9\\.]+)'}],
    strategy='Random',
    max_jobs=20,
    max_parallel_jobs=2
)

# Start the tuning job
tuner.fit({'train': s3_train_data, 'validation': s3_validation_data})
```

Module 20: Bayesian Optimization with SageMaker

The Limitation of Grid and Random Search

Both Grid and Random Search are **uninformed** methods. They do not learn from the results of previous training jobs. If a region of the hyperparameter space is performing poorly, they will continue to sample from it, wasting computational resources.

Basics of Bayesian Optimization

Bayesian Optimization is a sequential, **informed** strategy for finding the optimum of a black-box function (in our case, the model's performance as a function of its hyperparameters). It is particularly well-suited for functions that are expensive to evaluate, like training a machine learning model.

It works in a cycle:

1. **Build a Surrogate Model:** It uses a probabilistic model, typically a **Gaussian Process (GP)**, to approximate the objective function. The GP models your uncertainty about the function's shape.

2. **Select the Next Point using an Acquisition Function:** An acquisition function uses the surrogate model to decide which hyperparameter combination to try next. It balances two competing goals:
    - **Exploration:** Sampling in areas where the surrogate model is uncertain.
    - **Exploitation:** Sampling in areas where the surrogate model predicts a high objective value.
3. **Evaluate the Objective Function:** Train a model with the selected hyperparameters and record the result (e.g., validation accuracy).
4. **Update the Surrogate Model:** Incorporate the new data point (hyperparameters -> result) into the Gaussian Process, making it a more accurate surrogate of the true objective function.
5. **Repeat:** Go back to step 2 until a stopping condition is met (e.g., max number of jobs).

This feedback loop allows Bayesian optimization to intelligently focus its search on the most promising regions of the hyperparameter space.

Setting up a SageMaker Tuning Job with Bayesian Optimization

In SageMaker, this is the default strategy (`"Bayesian"`). The setup is almost identical to Random Search, but you change the strategy.

Key Considerations:

- **Number of Parallel Jobs:** Bayesian optimization is inherently sequential. To maintain its effectiveness, you should limit `MaxParallelTrainingJobs`. A common practice is to set it to `1` for purely sequential learning, or a small number (e.g., `2` or `3`) to speed things up slightly without sacrificing too much efficiency.
- **Number of Initial Jobs:** SageMaker often runs a few random jobs first to "seed" the Gaussian Process model before starting the Bayesian loop.

Example Snippet (Bayesian Optimization):

```
from sagemaker.tuner import HyperparameterTuner

# Using the same hyperparameter_ranges as before

tuner = HyperparameterTuner(
    estimator=my_sagemaker_estimator,
    objective_metric_name='validation:accuracy',
    hyperparameter_ranges=hyperparameter_ranges,
    metric_definitions=[{'Name': 'validation:accuracy', 'Regex': 'Validation-accuracy: ([0-9\\.]+)'}],
    strategy='Bayesian',  # This is the only key change
    max_jobs=20,
    max_parallel_jobs=2  # Keep this low for Bayesian
)

tuner.fit({'train': s3_train_data, 'validation': s3_validation_data})
```

Module 21: Q&A and Recap

Key Takeaways Recap

1. **Hyperparameters vs. Parameters:** Hyperparameters are set before training and control the learning process; parameters are learned during training.
2. **Why Tune?** Manual tuning is inefficient. Automated tuning is essential for achieving peak model performance and reproducibility.
3. **Grid Search:** Exhaustive and simple but scales poorly. Best for small, discrete search spaces.

4. **Random Search:** More efficient than grid search for most problems, especially with many hyperparameters. It is an excellent baseline method.
5. **Bayesian Optimization:** An intelligent, sequential method that uses past results to inform future trials. It is typically the most sample-efficient strategy, meaning it finds a good solution with fewer training jobs.
6. **SageMaker Automatic Model Tuning:** A managed service that simplifies running large-scale hyperparameter tuning jobs, supporting all three strategies (Grid, Random, Bayesian).

Best Practices for SageMaker Tuning

- **Start with a Broad Search:** Use Random Search with a reasonable budget (e.g., 20-50 jobs) to narrow down the ranges for your most important hyperparameters.
- **Refine with Bayesian:** Use the results of the initial Random Search to define a more focused search space, and then run a Bayesian optimization job to fine-tune.
- **Log Your Metrics Correctly:** Ensure your training script prints the objective metric to `stdout` in a format that the `metric_definitions` Regex can parse.
- **Use Warm Start:** SageMaker allows you to "warm start" a new tuning job using the results from a previous one. This is incredibly useful for continuing the search without starting from scratch.
- **Monitor Costs:** Keep an eye on `MaxNumberOfTrainingJobs` and `MaxParallelTrainingJobs`, as they directly impact your AWS bill.

Frequently Asked Questions (Q&A)

Q: When should I use Grid Search over Bayesian Optimization? A: Almost never for complex models. Grid Search is only practical when you have a very small number of hyperparameters (1-3) and a limited number of values to try for each. For all other cases, Random or Bayesian is preferred.

Q: How many hyperparameters can I tune at once? A: While SageMaker supports many, the "curse of dimensionality" still applies. It's best to focus on the 3-5 hyperparameters that are known to have the most impact on your model type. Tuning too many at once can make the search space too sparse for any method to be effective.

Q: How do I choose between `ContinuousParameter`, `IntegerParameter`, and `CategoricalParameter`? A: Use the type that matches your hyperparameter.

- `ContinuousParameter`: For real-valued numbers (e.g., learning rate, dropout rate).
- `IntegerParameter`: For whole numbers (e.g., number of layers, batch size).
- `CategoricalParameter`: For discrete, non-numeric choices (e.g., optimizer `['adam', 'sgd']`, kernel type).

Q: My tuning job is taking a long time. How can I speed it up? A:

1. Increase `MaxParallelTrainingJobs` (though be cautious with Bayesian).
2. Ensure your training script and infrastructure (e.g., using a GPU instance) are optimized for speed.
3. Use a smaller dataset or fewer epochs for the tuning phase to get a rough idea of good hyperparameters, then do a final training with the best set on the full dataset.
4. Narrow your search space based on prior knowledge or a preliminary Random Search.

Q: How do I know if my tuning job was successful? A: The SageMaker console provides detailed analytics. You can view all training jobs, a plot of the objective metric over time, and the best hyperparameters found. The best model can be deployed directly from the tuner object: `tuner.deploy(...)`.

Exercise 1: Basic Python Data Types and Operations

```
# 1. Create variables of different types
learning_rate = 0.01
num_layers = 5
model_name = "xgboost"
is_trained = False

# TODO: Print the type of each variable using type()
print("Type of learning_rate:", type(learning_rate))
print("Type of num_layers:", type(num_layers))
print("Type of model_name:", type(model_name))
print("Type of is_trained:", type(is_trained))

# 2. Basic arithmetic operations
batch_size = 32
num_epochs = 10
total_samples = 1000

# TODO: Calculate batches per epoch and total batches
batches_per_epoch = total_samples / batch_size
total_batches = num_epochs * batches_per_epoch

print(f"Batches per epoch: {batches_per_epoch}")
print(f"Total batches: {total_batches}")
```

Exercise 2: Lists and Dictionaries for Hyperparameters

```
# 1. Create lists of possible hyperparameter values
learning_rates = [0.001, 0.01, 0.1, 0.2]
batch_sizes = [32, 64, 128, 256]
optimizers = ["adam", "sgd", "rmsprop"]

# TODO: Access elements from lists
print("First learning rate:", learning_rates[0])
print("Last batch size:", batch_sizes[-1])
print("Available optimizers:", len(optimizers))

# 2. Create a dictionary representing hyperparameters
hyperparams = {
    "learning_rate": 0.01,
    "batch_size": 32,
    "num_epochs": 50,
    "optimizer": "adam"
}

# TODO: Access and modify dictionary values
print("Current learning rate:", hyperparams["learning_rate"])
hyperparams["learning_rate"] = 0.02
hyperparams["dropout_rate"] = 0.5  # Add new hyperparameter

print("Updated hyperparameters:", hyperparams)
```

Exercise 3: Loops and Conditional Logic

```
# 1. Loop through hyperparameter combinations (simplified grid search)
learning_rates = [0.01, 0.1]
batch_sizes = [32, 64]

# TODO: Use nested loops to iterate through all combinations
print("Grid Search Combinations:")
for lr in learning_rates:
    for bs in batch_sizes:
        print(f"Learning rate: {lr}, Batch size: {bs}")

# 2. Conditional logic for model configuration
def create_model_config(learning_rate, dataset_size):
    """
    TODO: Based on learning rate and dataset size,
    recommend batch size and epochs
    """
    if learning_rate > 0.1:
        batch_size = 32
        epochs = 50
    elif learning_rate > 0.01:
        batch_size = 64
        epochs = 100
    else:
        batch_size = 128
        epochs = 200

    if dataset_size > 10000:
        batch_size = min(batch_size * 2, 512)

    return batch_size, epochs

# Test the function
print(create_model_config(0.05, 5000))  # Should return (64, 100)
print(create_model_config(0.001, 20000)) # Should return (256, 200)
```

Exercise 4: Functions for ML Operations

```
# 1. Create a function to calculate evaluation metrics
def calculate_metrics(predictions, targets):
    """
    TODO: Calculate accuracy and loss
    predictions: list of predicted values
    targets: list of actual values
    """
    correct = 0
    total_loss = 0

    for pred, target in zip(predictions, targets):
        if pred == target:
            correct += 1
        total_loss += (pred - target) ** 2

    accuracy = correct / len(predictions)
    loss = total_loss / len(predictions)

    return accuracy, loss

# Test the function
predictions = [1, 0, 1, 1, 0]
targets = [1, 0, 0, 1, 1]
accuracy, loss = calculate_metrics(predictions, targets)
print(f"Accuracy: {accuracy:.2f}, Loss: {loss:.2f}")

# 2. Function to log metrics (simulating SageMaker behavior)
def log_metrics(epoch, accuracy, loss):
    """
    TODO: Print metrics in a format that SageMaker can parse
    """
    print(f"Epoch {epoch}: Training-accuracy: {accuracy:.4f}")
    print(f"Epoch {epoch}: Validation-loss: {loss:.4f}")

# Test logging
log_metrics(1, 0.85, 0.45)
log_metrics(2, 0.88, 0.38)
```

Exercise 5: Working with Classes and Objects

```
# 1. Create a simple model configuration class
class ModelConfig:
    def __init__(self, learning_rate, batch_size, num_epochs):
        self.learning_rate = learning_rate
        self.batch_size = batch_size
        self.num_epochs = num_epochs

    def display_config(self):
        """TODO: Print the configuration in a readable format"""
        print(f"Model Configuration:")
        print(f"  Learning Rate: {self.learning_rate}")
        print(f"  Batch Size: {self.batch_size}")
        print(f"  Number of Epochs: {self.num_epochs}")

    def update_learning_rate(self, new_lr):
        """TODO: Update learning rate with validation"""
        if 0 < new_lr < 1:
            self.learning_rate = new_lr
        else:
            print("Error: Learning rate must be between 0 and 1")

# Test the class
config1 = ModelConfig(0.01, 64, 100)
config1.display_config()
config1.update_learning_rate(0.05)
config1.display_config()
```

Exercise 6: File I/O and Data Handling

```
# 1. Read and write configuration files
def save_hyperparameters(filename, hyperparams):
    """
    TODO: Save hyperparameters to a file
    """
    with open(filename, 'w') as f:
        for key, value in hyperparams.items():
            f.write(f"{key}: {value}\n")
    print(f"Hyperparameters saved to {filename}")

def load_hyperparameters(filename):
    """
    TODO: Load hyperparameters from a file
    """
    hyperparams = {}
    with open(filename, 'r') as f:
        for line in f:
            key, value = line.strip().split(": ")
            # Convert numeric values
            try:
                value = float(value) if '.' in value else int(value)
            except ValueError:
                pass  # Keep as string if conversion fails
            hyperparams[key] = value
    return hyperparams

# Test file operations
test_params = {"learning_rate": 0.01, "batch_size": 64, "model": "xgboost"}
save_hyperparameters("model_config.txt", test_params)
loaded_params = load_hyperparameters("model_config.txt")
print("Loaded parameters:", loaded_params)
```

Exercise 7: Basic Error Handling

```
# 1. Add error handling to hyperparameter validation
def validate_hyperparameters(hyperparams):
    """
    TODO: Validate hyperparameters with proper error handling
    """
    try:
        if hyperparams["learning_rate"] <= 0:
            raise ValueError("Learning rate must be positive")

        if hyperparams["batch_size"] <= 0:
            raise ValueError("Batch size must be positive")

        if hyperparams["batch_size"] > 1024:
            print("Warning: Large batch size may cause memory issues")

        print("All hyperparameters are valid!")
        return True

    except KeyError as e:
        print(f"Missing hyperparameter: {e}")
        return False
    except ValueError as e:
        print(f"Invalid value: {e}")
        return False

# Test validation
good_params = {"learning_rate": 0.01, "batch_size": 64}
bad_params = {"learning_rate": -0.1, "batch_size": 64}
missing_params = {"batch_size": 64}

validate_hyperparameters(good_params)
validate_hyperparameters(bad_params)
validate_hyperparameters(missing_params)
```

Solutions

▶ Click to view solutions

These exercises cover the fundamental Python concepts needed to work with SageMaker's Hyperparameter Tuning:

- Data types and operations
- Lists and dictionaries for parameter storage
- Loops for iterating through combinations
- Functions for reusable logic
- Classes for organizing configuration
- File I/O for saving/loading parameters
- Error handling for robust code

Complete these before starting the SageMaker Hyperparameter Tuning course for the best learning experience!