# AutoML Agent Interface

Select the dataset type

time_series

Choose a dataset

Sunspot

Select a GROQ LLM Model

gemini-2.0-flash

Run AutoML Agent

## Instructor Response

## Configuration Plan

Preprocessing steps: 1. Scaling: Use MinMaxScaler or StandardScaler to scale the data. 2. Detrending: Remove any trend component using differencing or polynomial fitting. 3. Seasonality Decomposition: Decompose the time series into trend, seasonal, and residual components.

Feature Engineering: 1. Lagged Features: Create lagged features (e.g., X(t-1), X(t-2), ...). 2. Rolling Statistics: Calculate rolling mean, standard deviation, min, and max over a window. 3. Exponential Smoothing: Apply exponential smoothing techniques to capture trends and seasonality. 4. Fourier Transform: Use Fourier transform to extract frequency-domain features.

Common Challenges: 1. Non-stationarity: Time series data may be non-stationary, requiring transformations like differencing. 2. Autocorrelation: Account for autocorrelation in the data when building models. 3. Forecasting Horizon: The accuracy of forecasts decreases as the forecasting horizon increases.

OpenML: Dataset name: Sunspots, Tags: timeseries

## Scenario Plan

Multi-fidelity optimization is not necessary for this relatively small dataset. A BlackBoxFacade should be sufficient. Budget settings are not applicable since multi-fidelity is not used. n_workers can be set based on available CPU cores, but a value between 4 and 8 should be reasonable. Special considerations: Ensure that the data is properly scaled and detrended before training.

Scenario Configuration: { "facade_type": "BlackBoxFacade", "n_workers": 4 }

## Train Function Plan

The train function should:

1. Load the data.
2. Preprocess the data (scaling, detrending).
3. Split the data into training and validation sets.
4. Define the model architecture.
5. Train the model on the training data.
6. Validate the model on the validation data.
7. Return the validation performance.

## Generated Configuration Space Code

```
from ConfigSpace import ConfigurationSpace, UniformFloatHyperparameter, UniformInt
from ConfigSpace.conditions import EqualsCondition, OrConjunction
from ConfigSpace.hyperparameters import UnParametrizedHyperparameter

def get_configspace() -> ConfigurationSpace:
    """
    Returns a ConfigurationSpace object for hyperparameter optimization of time se

    The configuration space includes hyperparameters for model selection (LSTM, GR
    number of layers, number of units per layer, dropout rate, learning rate, batc
    and optimization parameters.  Appropriate conditions are added between depende
    """
    cs = ConfigurationSpace()

    # Model Type Selection
    model_type = CategoricalHyperparameter(
        "model_type", choices=["LSTM", "GRU", "RNN", "LinearRegression", "Naive"],
    )
    cs.add_hyperparameter(model_type)

    # LSTM/GRU/RNN Specific Hyperparameters
    num_layers = UniformIntegerHyperparameter(
        "num_layers", lower=1, upper=3, default_value=2
    )
```

```
    num_units = UniformIntegerHyperparameter(
        "num_units", lower=32, upper=256, default_value=64
    )
    dropout_rate = UniformFloatHyperparameter(
        "dropout_rate", lower=0.0, upper=0.5, default_value=0.2
    )
    cs.add_hyperparameters([num_layers, num_units, dropout_rate])

    # Linear Regression / Naive Baseline does not need layers, units, or dropout
    lstm_condition = EqualsCondition(num_layers, model_type, "LSTM")
    gru_condition = EqualsCondition(num_layers, model_type, "GRU")
    rnn_condition = EqualsCondition(num_layers, model_type, "RNN")

    condition_num_layers = OrConjunction(lstm_condition, gru_condition, rnn_condit
    cs.add_condition(condition_num_layers)

    lstm_condition = EqualsCondition(num_units, model_type, "LSTM")
    gru_condition = EqualsCondition(num_units, model_type, "GRU")
    rnn_condition = EqualsCondition(num_units, model_type, "RNN")

    condition_num_units = OrConjunction(lstm_condition, gru_condition, rnn_conditi
    cs.add_condition(condition_num_units)

    lstm_condition = EqualsCondition(dropout_rate, model_type, "LSTM")
    gru_condition = EqualsCondition(dropout_rate, model_type, "GRU")
    rnn_condition = EqualsCondition(dropout_rate, model_type, "RNN")

    condition_dropout_rate = OrConjunction(lstm_condition, gru_condition, rnn_cond
    cs.add_condition(condition_dropout_rate)

    # Shared Hyperparameters
    learning_rate = UniformFloatHyperparameter(
        "learning_rate", lower=1e-4, upper=1e-2, default_value=1e-3, log=True
    )
    batch_size = CategoricalHyperparameter(
        "batch_size", choices=[32, 64, 128, 256], default_value=64
    )
    optimizer = CategoricalHyperparameter(
        "optimizer", choices=["Adam", "SGD"], default_value="Adam"
    )
    cs.add_hyperparameters([learning_rate, batch_size, optimizer])

    # Optimizer-specific hyperparameters
    beta_1 = UniformFloatHyperparameter(
        "beta_1", lower=0.8, upper=0.99, default_value=0.9, log=False
    )
```

```
    momentum = UniformFloatHyperparameter(
        "momentum", lower=0.0, upper=0.9, default_value=0.0, log=False
    )
    cs.add_hyperparameters([beta_1, momentum])

    # Condition beta_1 on Adam
    condition_beta_1 = EqualsCondition(beta_1, optimizer, "Adam")
    cs.add_condition(condition_beta_1)

    # Condition momentum on SGD
    condition_momentum = EqualsCondition(momentum, optimizer, "SGD")
    cs.add_condition(condition_momentum)

    # Add preprocessing boolean options for scaling, detrending, seasonality
    scaler = CategoricalHyperparameter(
        "scaler", choices=["MinMaxScaler", "StandardScaler", "None"], default_valu
    )
    detrend = CategoricalHyperparameter(
        "detrend", choices=["Differencing", "Polynomial", "None"], default_value="
    )
    seasonality_decomp = CategoricalHyperparameter(
        "seasonality_decomp", choices=["STL", "ETS", "None"], default_value="None"
    )
    cs.add_hyperparameters([scaler, detrend, seasonality_decomp])


    return cs
```

## Generated Scenario Code

```
from smac import Scenario
from ConfigSpace import ConfigurationSpace


def generate_scenario(cs: ConfigurationSpace) -> Scenario:
    scenario = Scenario(
        configspace=cs,
        name="HyperparameterOptimization",
        output_directory="./logs/gemini-2.0-flash_Sunspot_20250617_191023",
        deterministic=False,
        n_trials=10,
        n_workers=1
    )
    return scenario
```

# Generated Training Function Code

```
import torch
import torch.nn as nn
import torch.optim as optim
from sklearn.preprocessing import MinMaxScaler, StandardScaler
from sklearn.linear_model import LinearRegression
import pandas as pd
import numpy as np
from typing import Any
from ConfigSpace import Configuration
from statsmodels.tsa.seasonal import STL
from statsmodels.tsa.api import ExponentialSmoothing

def train(cfg: Configuration, dataset: Any, seed: int) -> float:
    """
    Trains a time series model based on the provided configuration and dataset.

    Args:
        cfg (Configuration): The configuration object containing hyperparameters.
        dataset (Any): The dataset containing 'X' (features) and 'y' (labels).
        seed (int): The random seed for reproducibility.

    Returns:
        float: The negative validation loss/error.
    """
    torch.manual_seed(seed)
    np.random.seed(seed)

    X = dataset['X']
    y = dataset['y']

    # Convert to numpy arrays if they are pandas Series
    if isinstance(X, pd.Series):
        X = X.to_numpy()
    if isinstance(y, pd.Series):
        y = y.to_numpy()

    # Handle different shapes for X
    if X.ndim == 1:
        X = X.reshape(-1, 1)

    if y.ndim == 1:
        y = y.reshape(-1, 1)
```

```
    # Infer sequence length and feature dimension
    if X.ndim == 3:
        seq_len = X.shape[1]
        num_features = X.shape[2]
    elif X.ndim == 2:
        seq_len = 1
        num_features = X.shape[1]
    else:
        raise ValueError(f"Unexpected input dimension: {X.ndim}.  Expected 2 or 3.

    model_type = cfg.get("model_type")

    # Data Preprocessing
    scaler_type = cfg.get("scaler")
    detrend_type = cfg.get("detrend")
    seasonality_decomp_type = cfg.get("seasonality_decomp")

    # Scaling
    if scaler_type == "MinMaxScaler":
        scaler = MinMaxScaler()
        X = scaler.fit_transform(X)
        y = scaler.fit_transform(y)
    elif scaler_type == "StandardScaler":
        scaler = StandardScaler()
        X = scaler.fit_transform(X)
        y = scaler.fit_transform(y)

    # Detrending
    if detrend_type == "Differencing":
        X = np.diff(X, axis=0)
        y = np.diff(y, axis=0)
    elif detrend_type == "Polynomial":
        poly = np.polyfit(np.arange(len(y)), y.flatten(), 1)  # Linear detrending
        trend = np.polyval(poly, np.arange(len(y)))
        y = y - trend.reshape(-1, 1)  # Ensure correct shape

    # Seasonality Decomposition
    if seasonality_decomp_type == "STL":
        stl = STL(y.flatten(), seasonal=13)  # Assuming seasonal period of 13
        res = stl.fit()
        y = res.trend
        X = X[13:]
        y = y[13:]
    elif seasonality_decomp_type == "ETS":
        model_ets = ExponentialSmoothing(y, seasonal_periods=12, seasonal='add').f
```

```
        y = model_ets.resid

    # Split data into training and validation sets
    train_size = int(len(X) * 0.8)
    X_train, X_val = X[:train_size], X[train_size:]
    y_train, y_val = y[:train_size], y[train_size:]

    # Convert to tensors
    X_train_tensor = torch.tensor(X_train, dtype=torch.float32)
    X_val_tensor = torch.tensor(X_val, dtype=torch.float32)
    y_train_tensor = torch.tensor(y_train, dtype=torch.float32)
    y_val_tensor = torch.tensor(y_val, dtype=torch.float32)

    # Model Definition and Training
    if model_type in ["LSTM", "GRU", "RNN"]:
        if X_train_tensor.ndim != 3 and seq_len > 1:
            X_train_tensor = X_train_tensor.reshape(X_train_tensor.shape[0], seq_l
            X_val_tensor = X_val_tensor.reshape(X_val_tensor.shape[0], seq_len, nu
        elif X_train_tensor.ndim != 3 and seq_len == 1:
            X_train_tensor = X_train_tensor.reshape(X_train_tensor.shape[0], seq_l
            X_val_tensor = X_val_tensor.reshape(X_val_tensor.shape[0], seq_len, nu

        num_layers = cfg.get("num_layers")
        num_units = cfg.get("num_units")
        dropout_rate = cfg.get("dropout_rate")
        batch_size = cfg.get("batch_size")
        learning_rate = cfg.get("learning_rate")
        optimizer_name = cfg.get("optimizer")

        class RNNModel(nn.Module):
            def __init__(self, input_size, hidden_size, num_layers, dropout, rnn_t
                super(RNNModel, self).__init__()
                self.rnn_type = rnn_type
                if rnn_type == "LSTM":
                    self.rnn = nn.LSTM(input_size, hidden_size, num_layers, batch_
                elif rnn_type == "GRU":
                    self.rnn = nn.GRU(input_size, hidden_size, num_layers, batch_f
                else:  # RNN
                    self.rnn = nn.RNN(input_size, hidden_size, num_layers, batch_f
                self.linear = nn.Linear(hidden_size, 1)
            def forward(self, x):
                out, _ = self.rnn(x)
                out = self.linear(out[:, -1, :])  # Only the last time step
                return out

        model = RNNModel(num_features, num_units, num_layers, dropout_rate, model_
```

```
        if optimizer_name == "Adam":
            beta_1 = cfg.get("beta_1")
            optimizer = optim.Adam(model.parameters(), lr=learning_rate, betas=(be
        else:  # SGD
            momentum = cfg.get("momentum")
            optimizer = optim.SGD(model.parameters(), lr=learning_rate, momentum=m

        criterion = nn.MSELoss()

        for epoch in range(10):
            model.train()
            for i in range(0, len(X_train_tensor), batch_size):
                X_batch = X_train_tensor[i:i + batch_size]
                y_batch = y_train_tensor[i:i + batch_size]

                optimizer.zero_grad()
                outputs = model(X_batch)
                loss = criterion(outputs, y_batch)
                loss.backward()
                optimizer.step()

        model.eval()
        with torch.no_grad():
            val_outputs = model(X_val_tensor)
            val_loss = criterion(val_outputs, y_val_tensor)
        return -val_loss.item()

    elif model_type == "LinearRegression":
        model = LinearRegression()
        model.fit(X_train, y_train)
        y_pred = model.predict(X_val)
        mse = np.mean((y_val - y_pred)**2)
        return -mse

    elif model_type == "Naive":
        y_pred = np.roll(y_train, 1)
        y_pred[0] = 0 # first pred is set to zero
        mse = np.mean((y_val[:len(y_pred)] - y_pred[-len(y_val):])**2) #compare th
        return -mse

    else:
        raise ValueError(f"Unknown model type: {model_type}")
```

AutoML Agent setup complete!

# Loss Value

`-0.0883231737506197`

# Starting Optimization Process

Starting Optimization Process

# Prompts Used

{

"config" :

~~~~
"**Generate a production-grade Python configuration space for machine learning hyperparameter optimization with the following STRICT requirements:**

---

### **Function signature** must be:

```python
from ConfigSpace import ConfigurationSpace, UniformFloatHyperparameter, UniformIntegerHyperparameter, CategoricalHyperparameter
def get_configspace() -> ConfigurationSpace:
```

---

### **Configuration Space Requirements:**

* The configuration space **must** be appropriate for the dataset type and characteristics:
  * Dataset Description: `This is a time series dataset.
Number of samples: 289
Time index type: <class 'pandas.core.indexes.range.RangeIndex'>
Time range: 0 to 288
Features:
- 0

### Time Series Handling Requirements
- Assume `dataset['X']` is a 3D array or tensor with shape `(num_samples, sequence_length, num_features)`.
- If `dataset['X']` is 2D, raise a `ValueError` if the model is RNN-based (`LSTM`, `GRU`, `RNN`).
- Do **not** flatten the input when using RNN-based models.
- Use `batch_first=True` in all recurrent models to maintain `(batch, seq_len, features)` format.
- Dynamically infer sequence length as `X.shape[1]` and feature dimension as `X.shape[2]`.
- If `X.ndim != 3` and a sequential model is selected, raise a clear error with shape info.
- Example input validation check:
  ```python
  if model_type in ['LSTM', 'GRU', 'RNN'] and X_tensor.ndim != 3:
      raise ValueError(f"Expected 3D input (batch, seq_len, features) for {model_type}, got {X_tensor.shape}")
  ```
- Time index or datetime values can be logged but should not be used in the model unless specified.
`
~~~~

```
* Recommended Configuration based on the planner:
  * `Preprocessing steps: 1. Scaling: Use MinMaxScaler or StandardScaler to
scale the data.
2. Detrending: Remove any trend component using differencing or polynomial
fitting.
3. Seasonality Decomposition: Decompose the time series into trend, seasonal,
and residual components.

Feature Engineering: 1. Lagged Features: Create lagged features (e.g., X(t-1),
X(t-2), ...).
2. Rolling Statistics: Calculate rolling mean, standard deviation, min, and max
over a window.
3. Exponential Smoothing: Apply exponential smoothing techniques to capture
trends and seasonality.
4. Fourier Transform: Use Fourier transform to extract frequency-domain
features.

Common Challenges: 1. Non-stationarity: Time series data may be non-stationary,
requiring transformations like differencing.
2. Autocorrelation: Account for autocorrelation in the data when building
models.
3. Forecasting Horizon: The accuracy of forecasts decreases as the forecasting
horizon increases.

OpenML: Dataset name: Sunspots, Tags: timeseries`

* The configuration space **must** include:
  * Appropriate hyperparameter ranges based on the dataset characteristics
  * Reasonable default values
  * Proper hyperparameter types (continuous, discrete, categorical)
  * Conditional hyperparameters if needed
  * Proper bounds and constraints

* **Best Practices:**
  * Use meaningful hyperparameter names
  * Include proper documentation for each hyperparameter
  * Consider dataset size and complexity when setting ranges
  * Ensure ranges are not too narrow or too wide
  * Add proper conditions between dependent hyperparameters

* **Common Hyperparameters to Consider:**
  * Learning rate (if applicable)
  * Model-specific hyperparameters
  * Regularization parameters
  * Architecture parameters
  * Optimization parameters
```

````
---

### **Output Format:**

* Return **only** the `get_configspace()` function
* Include necessary imports
* No example usage or additional code
* The function must be self-contained and executable

---

### **Error Prevention:**

* Ensure all hyperparameter names are valid Python identifiers
* Verify that all ranges and bounds are valid
* Check that conditional hyperparameters are properly defined
* Validate that default values are within the specified ranges

---

### **Example Structure:**

```python
def get_configspace() -> ConfigurationSpace:
    cs = ConfigurationSpace()

    # Add hyperparameters
    learning_rate = UniformFloatHyperparameter(
        "learning_rate", lower=1e-4, upper=1e-1, default_value=1e-2, log=True
    )
    cs.add_hyperparameter(learning_rate)

    # Add more hyperparameters...

    return cs
```

---

**Reminder:** The output must be limited to:
* Valid `import` statements
* A single `get_configspace()` function that returns a properly configured
`ConfigurationSpace` object
* No additional code or explanations"
````

"scenario" :

````
"**Generate a production-grade Python scenario configuration for SMAC
hyperparameter optimization with the following STRICT requirements:**

---

### **Function signature** must be:

```python
from smac import Scenario
from ConfigSpace import ConfigurationSpace
def generate_scenario(cs: ConfigurationSpace) -> Scenario:
```


---

### **Scenario Configuration Requirements:**

* The scenario **must** be optimized for the dataset characteristics:
  * Dataset Description: `This is a time series dataset.
Number of samples: 289
Time index type: <class 'pandas.core.indexes.range.RangeIndex'>
Time range: 0 to 288
Features:
- 0

### Time Series Handling Requirements
- Assume `dataset['X']` is a 3D array or tensor with shape `(num_samples,
sequence_length, num_features)`.
- If `dataset['X']` is 2D, raise a `ValueError` if the model is RNN-based
(`LSTM`, `GRU`, `RNN`).
- Do **not** flatten the input when using RNN-based models.
- Use `batch_first=True` in all recurrent models to maintain `(batch, seq_len,
features)` format.
- Dynamically infer sequence length as `X.shape[1]` and feature dimension as
`X.shape[2]`.
- If `X.ndim != 3` and a sequential model is selected, raise a clear error with
shape info.
- Example input validation check:
  ```python
  if model_type in ['LSTM', 'GRU', 'RNN'] and X_tensor.ndim != 3:
      raise ValueError(f"Expected 3D input (batch, seq_len, features) for
{model_type}, got {X_tensor.shape}")
  ```
- Time index or datetime values can be logged but should not be used in the
model unless specified.
`
````

```
* The scenario **must** include:
  * Appropriate budget settings (min_budget, max_budget)
  * Optimal number of workers for parallelization
  * Reasonable walltime and CPU time limits
  * Proper trial resource constraints
  * Appropriate number of trials

* **Best Practices:**
  * Set deterministic=False for better generalization
  * Use multi-fidelity optimization when appropriate
  * Configure proper output directory structure
  * Set appropriate trial resource limits
  * Enable parallel optimization when possible

* **Resource Management:**
  * Set appropriate memory limits for trials
  * Configure proper walltime limits
  * Enable parallel processing when beneficial
  * Consider dataset size for budget settings

---


### **Available Parameters:**
    configspace : ConfigurationSpace
        The configuration space from which to sample the configurations.
    name : str | None, defaults to None
        The name of the run. If no name is passed, SMAC generates a hash from
the meta data.
        Specify this argument to identify your run easily.
    output_directory : Path, defaults to Path("smac3_output")
        The directory in which to save the output. The files are saved in
`./output_directory/name/seed`.
    deterministic : bool, defaults to False
        If deterministic is set to true, only one seed is passed to the target
function.
        Otherwise, multiple seeds (if n_seeds of the intensifier is greater
than 1) are passed
        to the target function to ensure generalization.
    objectives : str | list[str] | None, defaults to "cost"
        The objective(s) to optimize. This argument is required for multi-
objective optimization.
    crash_cost : float | list[float], defaults to np.inf
        Defines the cost for a failed trial. In case of multi-objective, each
objective can be associated with
        a different cost.
    termination_cost_threshold : float | list[float], defaults to np.inf
        Defines a cost threshold when the optimization should stop. In case of
```

```
multi-objective, each objective *must* be
            associated with a cost. The optimization stops when all objectives
crossed the threshold.
        walltime_limit : float, defaults to np.inf
            The maximum time in seconds that SMAC is allowed to run.
        cputime_limit : float, defaults to np.inf
            The maximum CPU time in seconds that SMAC is allowed to run.
        trial_walltime_limit : float | None, defaults to None
            The maximum time in seconds that a trial is allowed to run. If not
specified,
            no constraints are enforced. Otherwise, the process will be spawned by
pynisher.
        trial_memory_limit : int | None, defaults to None
            The maximum memory in MB that a trial is allowed to use. If not
specified,
            no constraints are enforced. Otherwise, the process will be spawned by
pynisher.
        n_trials : int, defaults to 100
            The maximum number of trials (combination of configuration, seed,
budget, and instance, depending on the task)
            to run.
        use_default_config: bool, defaults to False.
            If True, the configspace's default configuration is evaluated in the
initial design.
            For historic benchmark reasons, this is False by default.
            Notice, that this will result in n_configs + 1 for the initial design.
Respecting n_trials,
            this will result in one fewer evaluated configuration in the
optimization.
        instances : list[str] | None, defaults to None
            Names of the instances to use. If None, no instances are used.
            Instances could be dataset names, seeds, subsets, etc.
        instance_features : dict[str, list[float]] | None, defaults to None
            Instances can be associated with features. For example, meta data of
the dataset (mean, var, ...) can be
            incorporated which are then further used to expand the training data of
the surrogate model.
        min_budget : float | int | None, defaults to None
            The minimum budget (epochs, subset size, number of instances, ...) that
is used for the optimization.
            Use this argument if you use multi-fidelity or instance optimization.
        max_budget : float | int | None, defaults to None
            The maximum budget (epochs, subset size, number of instances, ...) that
is used for the optimization.
            Use this argument if you use multi-fidelity or instance optimization.
        seed : int, defaults to 0
            The seed is used to make results reproducible. If seed is -1, SMAC will
```

````
        The seed is used to make results reproducible. If seed is -1, SMAC will
generate a random seed.
    n_workers : int, defaults to 1
        The number of workers to use for parallelization. If `n_workers` is
greather than 1, SMAC will use
        Dask to parallelize the optimization.

---

### **Output Format:**

* Return **only** the `generate_scenario(cs)` function
* Include necessary imports
* No example usage or additional code
* The function must be self-contained and executable

---

### **Error Prevention:**

* Ensure all parameters are within valid ranges
* Verify that resource limits are reasonable
* Check that budget settings are appropriate
* Validate that parallelization settings are correct
* Ensure the training function can be pickled for parallel processing

---

### **Example Structure:**

```python
def generate_scenario(cs: ConfigurationSpace) -> Scenario:
    scenario = Scenario(
        configspace=cs,
        name="HyperparameterOptimization",
        output_directory="./logs/gemini-2.0-flash_Sunspot_20250617_191023"
//this is important and should not be changed
        deterministic=True,
        //other parameters based on the information
    )
    return scenario
```

---

### **Suggested Scenario Plan:**

Multi-fidelity optimization is not necessary for this relatively small dataset
````

```
Multi-fidelity optimization is not necessary for this relatively small dataset.
A BlackBoxFacade should be sufficient.
Budget settings are not applicable since multi-fidelity is not used.
n_workers can be set based on available CPU cores, but a value between 4 and 8
should be reasonable.
Special considerations: Ensure that the data is properly scaled and detrended
before training.

Scenario Configuration:
{
  "facade_type": "BlackBoxFacade",
  "n_workers": 4
}

---

**Reminder:** The output must be limited to:
* Valid `import` statements
* A single `generate_scenario(cs)` function that returns a properly configured
`Scenario` object
* No additional code or explanations
* The output_directory should be "./logs/gemini-2.0-
flash_Sunspot_20250617_191023"
* Set the number of trials to 10 for sufficient exploration
* set the number of workers to 1
* do not set these parameters: walltime_limit, cputime_limit,
trial_walltime_limit ,trial_memory_limit="
```

"train_function" :

````
"**Generate a production-grade Python training function for machine learning with the following STRICT requirements:**

---

### **Function signature** must be:

```python
from ConfigSpace import Configuration
from typing import Any
def train(cfg: Configuration, dataset: Any, seed: int) -> float:
```

---

### **Function Behavior Requirements:**

* The function **must** handle the dataset properly:
  * Dataset Description: `This is a time series dataset.
Number of samples: 289
Time index type: <class 'pandas.core.indexes.range.RangeIndex'>
Time range: 0 to 288
Features:
- 0

### Time Series Handling Requirements
- Assume `dataset['X']` is a 3D array or tensor with shape `(num_samples, sequence_length, num_features)`.
- If `dataset['X']` is 2D, raise a `ValueError` if the model is RNN-based (`LSTM`, `GRU`, `RNN`).
- Do **not** flatten the input when using RNN-based models.
- Use `batch_first=True` in all recurrent models to maintain `(batch, seq_len, features)` format.
- Dynamically infer sequence length as `X.shape[1]` and feature dimension as `X.shape[2]`.
- If `X.ndim != 3` and a sequential model is selected, raise a clear error with shape info.
- Example input validation check:
  ```python
  if model_type in ['LSTM', 'GRU', 'RNN'] and X_tensor.ndim != 3:
      raise ValueError(f"Expected 3D input (batch, seq_len, features) for {model_type}, got {X_tensor.shape}")
  ```
- Time index or datetime values can be logged but should not be used in the model unless specified.
`
  * ConfigSpace Definition: `from ConfigSpace import ConfigurationSpace,
````

```
UniformFloatHyperparameter, UniformIntegerHyperparameter,
CategoricalHyperparameter
from ConfigSpace.conditions import EqualsCondition, OrConjunction
from ConfigSpace.hyperparameters import UnParametrizedHyperparameter

def get_configspace() -> ConfigurationSpace:
    """
    Returns a ConfigurationSpace object for hyperparameter optimization of time
series models.

    The configuration space includes hyperparameters for model selection (LSTM,
GRU, RNN, Linear Regression, or Naive),
    number of layers, number of units per layer, dropout rate, learning rate,
batch size,
    and optimization parameters.  Appropriate conditions are added between
dependent hyperparameters.
    """
    cs = ConfigurationSpace()

    # Model Type Selection
    model_type = CategoricalHyperparameter(
        "model_type", choices=["LSTM", "GRU", "RNN", "LinearRegression",
"Naive"], default_value="LSTM"
    )
    cs.add_hyperparameter(model_type)

    # LSTM/GRU/RNN Specific Hyperparameters
    num_layers = UniformIntegerHyperparameter(
        "num_layers", lower=1, upper=3, default_value=2
    )
    num_units = UniformIntegerHyperparameter(
        "num_units", lower=32, upper=256, default_value=64
    )
    dropout_rate = UniformFloatHyperparameter(
        "dropout_rate", lower=0.0, upper=0.5, default_value=0.2
    )
    cs.add_hyperparameters([num_layers, num_units, dropout_rate])

    # Linear Regression / Naive Baseline does not need layers, units, or
dropout
    lstm_condition = EqualsCondition(num_layers, model_type, "LSTM")
    gru_condition = EqualsCondition(num_layers, model_type, "GRU")
    rnn_condition = EqualsCondition(num_layers, model_type, "RNN")

    condition_num_layers = OrConjunction(lstm_condition, gru_condition,
rnn_condition)
    cs.add_condition(condition_num_layers)
```

```
    lstm_condition = EqualsCondition(num_units, model_type, "LSTM")
    gru_condition = EqualsCondition(num_units, model_type, "GRU")
    rnn_condition = EqualsCondition(num_units, model_type, "RNN")

    condition_num_units = OrConjunction(lstm_condition, gru_condition,
rnn_condition)
    cs.add_condition(condition_num_units)

    lstm_condition = EqualsCondition(dropout_rate, model_type, "LSTM")
    gru_condition = EqualsCondition(dropout_rate, model_type, "GRU")
    rnn_condition = EqualsCondition(dropout_rate, model_type, "RNN")

    condition_dropout_rate = OrConjunction(lstm_condition, gru_condition,
rnn_condition)
    cs.add_condition(condition_dropout_rate)

    # Shared Hyperparameters
    learning_rate = UniformFloatHyperparameter(
        "learning_rate", lower=1e-4, upper=1e-2, default_value=1e-3, log=True
    )
    batch_size = CategoricalHyperparameter(
        "batch_size", choices=[32, 64, 128, 256], default_value=64
    )
    optimizer = CategoricalHyperparameter(
        "optimizer", choices=["Adam", "SGD"], default_value="Adam"
    )
    cs.add_hyperparameters([learning_rate, batch_size, optimizer])

    # Optimizer-specific hyperparameters
    beta_1 = UniformFloatHyperparameter(
        "beta_1", lower=0.8, upper=0.99, default_value=0.9, log=False
    )
    momentum = UniformFloatHyperparameter(
        "momentum", lower=0.0, upper=0.9, default_value=0.0, log=False
    )
    cs.add_hyperparameters([beta_1, momentum])

    # Condition beta_1 on Adam
    condition_beta_1 = EqualsCondition(beta_1, optimizer, "Adam")
    cs.add_condition(condition_beta_1)

    # Condition momentum on SGD
    condition_momentum = EqualsCondition(momentum, optimizer, "SGD")
    cs.add_condition(condition_momentum)

    # Add preprocessing boolean options for scaling, detrending, seasonality
```

```
    # Add preprocessing boolean options for scaling, detrending, seasonality
    scaler = CategoricalHyperparameter(
        "scaler", choices=["MinMaxScaler", "StandardScaler", "None"],
default_value="StandardScaler"
    )
    detrend = CategoricalHyperparameter(
        "detrend", choices=["Differencing", "Polynomial", "None"],
default_value="None"
    )
    seasonality_decomp = CategoricalHyperparameter(
        "seasonality_decomp", choices=["STL", "ETS", "None"],
default_value="None"
    )
    cs.add_hyperparameters([scaler, detrend, seasonality_decomp])


    return cs
`

  * SMAC Scenario: `from smac import Scenario
from ConfigSpace import ConfigurationSpace


def generate_scenario(cs: ConfigurationSpace) -> Scenario:
    scenario = Scenario(
        configspace=cs,
        name="HyperparameterOptimization",
        output_directory="./logs/gemini-2.0-flash_Sunspot_20250617_191023",
        deterministic=False,
        n_trials=10,
        n_workers=1
    )
    return scenario
`


* The function **must** accept a `dataset` dictionary with:
  * `dataset['X']`: feature matrix or input tensor
  * `dataset['y']`: label vector or label tensor

* The function **must** handle the configuration properly:
  * Access primitive values using `cfg.get('key')`
  * Handle all hyperparameters defined in the configuration space
  * Apply proper type conversion and validation
  * Handle conditional hyperparameters correctly

* **Model Requirements:**
  * Infer input and output dimensions dynamically
  * Follow data format requirements
  * Handle necessary data transformations
```

```
    * Handle necessary data transformations
    * Implement proper model initialization
    * Use appropriate loss functions
    * Apply proper regularization
    * Handle model-specific requirements

  * **Training Requirements:**
    * Implement proper training loop
    * Handle batch processing
    * Apply proper optimization
    * Implement early stopping if needed
    * Handle validation if required
    * Return appropriate loss value

  * **Performance Optimization Requirements:**
    * Minimize memory usage and allocations
    * Use vectorized operations where possible
    * Avoid unnecessary data copying
    * Optimize data loading and preprocessing
    * Use efficient data structures
    * Minimize CPU/GPU synchronization
    * Implement efficient batch processing
    * Use appropriate device placement (CPU/GPU)
    * Optimize model forward/backward passes
    * Minimize Python overhead

  * **Code Optimization Requirements:**
    * Keep code minimal and focused
    * Avoid redundant computations
    * Use efficient algorithms
    * Minimize function calls
    * Optimize loops and iterations
    * Use appropriate data types
    * Avoid unnecessary object creation
    * Implement efficient error handling
    * Use appropriate caching strategies
    * The train function should be computational efficient

  * **Best Practices:**
    * Use proper error handling
    * Implement proper logging
    * Handle edge cases
    * Ensure reproducibility
    * Optimize performance
    * Follow framework best practices

  ---
```

````
### **Frameworks:**

Choose **one** of the following frameworks based on the dataset and
requirements:
* **PyTorch**: For deep learning tasks
* **TensorFlow**: For deep learning tasks
* **scikit-learn**: For traditional ML tasks

---

### **Output Format:**

* Return **only** the `train()` function
* Include necessary imports
* No example usage or additional code
* The function must be self-contained and executable
* Code must be minimal and optimized for performance

---

### **Error Prevention:**

* Validate all inputs
* Handle missing or invalid hyperparameters
* Check data types and shapes
* Handle edge cases
* Implement proper error messages

---

### **Example Structure:**

```python
def train(cfg: Configuration, dataset: Any, seed: int) -> float:
    # Set random seed for reproducibility
    torch.manual_seed(seed)

    # Extract hyperparameters efficiently
    lr, bs = cfg.get('learning_rate'), cfg.get('batch_size')

    # Prepare data efficiently
    X, y = dataset['X'], dataset['y']

    # Initialize model with optimized parameters
    model = Model(X.shape[1], **cfg).to(device)
````
````

```
    # Optimized training loop
    for epoch in range(10):
        loss = train_epoch(model, X, y, lr, bs)

    return loss
```
```

---

**Reminder:** The output must be limited to:
* Valid `import` statements
* A single `train()` function that returns a float loss value
* No additional code or explanations
* Code must be optimized for performance and minimal in size
* Return negative loss/error since SMAC minimizes the objective
* For accuracy metrics, return negative accuracy (e.g. -accuracy)
* For error metrics, return the raw error value (e.g. mse, rmse)
* Ensure consistent sign convention across all metrics
* For tracking the progress add prints
* Do not cheat in order to escape an Error and do not use Try ExceptThe train
function should:
1. Load the data.
2. Preprocess the data (scaling, detrending).
3. Split the data into training and validation sets.
4. Define the model architecture.
5. Train the model on the training data.
6. Validate the model on the validation data.
7. Return the validation performance."
}
```

Download Generated Code and Prompts