AutoML Agent Interface

Select the dataset type

image

Choose a dataset

MNIST

Select a GROQ LLM Model

gemini-2.0-flash

Run AutoML Agent

Instructor Response

Configuration Plan

To effectively train models on the MNIST dataset, consider these configurations:

1. Data Preprocessing:
    - Input Format: For CNNs, reshape input to (N, 1, 28, 28) for PyTorch or (N, 28, 28, 1) for TensorFlow. For dense layers, flatten the input to (N, 784).
    - Normalization: Scale pixel values to the range [0, 1] by dividing by 255.0.
2. Model Architecture:
    - CNN: A simple CNN with convolutional layers, ReLU activations, max pooling, and fully connected layers is a good starting point. The provided VGGNet configuration can serve as inspiration.
    - Dense Network: A multi-layer perceptron with ReLU activations can also be used, especially for baseline performance.
3. Training Parameters:
    - Batch Size: 128 is a commonly used batch size.
    - Epochs: Train for at least 10 epochs, but monitor validation loss to prevent overfitting.
    - Optimizer: Adam or SGD with momentum are suitable optimizers.
    - Learning Rate: Start with 0.001 and adjust as needed.

4. Regularization:
    - Dropout: Add dropout layers (e.g., rate of 0.5) to prevent overfitting.
    - Batch Normalization: Use batch normalization after convolutional layers.
5. SMAC Configuration:
    - Facade Type: MultiFidelityFacade is appropriate if considering varying training epochs or subsets of the data.
    - Budget Settings: Set min_budget to a small number of epochs (e.g., 5) and max_budget to the desired maximum number of epochs (e.g., 50).
    - Number of Workers: Set n_workers based on available CPU/GPU resources. Start with 4 and increase if possible.
6. SMAC Scenario Parameters:
    - walltime_limit: Set a reasonable time limit (e.g., 3600 seconds).
    - n_trials: Set the number of trials based on the budget and time constraints (e.g., 200).

Best parameters to set in the Configuration Space:

- Learning Rate: UniformFloat(lower=1e-4, upper=1e-1, log=True, default_value=1e-3)
- Batch Size: Categorical([32, 64, 128, 256], default_value=128)
- Optimizer: Categorical(["Adam", "SGD"], default_value="Adam")
- Number of Convolutional Layers: Integer(lower=2, upper=5, default_value=3)
- Dropout Rate: UniformFloat(lower=0.0, upper=0.5, default_value=0.2)
- Number of Epochs: Integer(lower=10, upper=50, default_value=20)

Scenario Plan

1. Data Preprocessing:
    - Reshape input data to the appropriate format (N, 1, 28, 28) for CNNs or flatten to (N, 784) for dense layers.
    - Normalize pixel values by dividing by 255.0.
2. Model Definition:
    - Define a CNN or dense network using a configuration space. Allow SMAC to tune hyperparameters like the number of layers, filter sizes, and dropout rates.
3. SMAC Scenario Configuration:
    - Use MultiFidelityFacade for multi-fidelity optimization to potentially reduce optimization time.
    - Set min_budget and max_budget for the number of epochs.

- Set n_workers based on available resources.
- Set walltime_limit and n_trials to control the optimization process.

4. Special Considerations:
   - MNIST is a relatively simple dataset, so more complex models may not be necessary.
   - Ensure proper data loading and batching to avoid memory issues.
   - Monitor training and validation loss to prevent overfitting.

Train Function Plan

1. Define the Model:
   - The train function should take a configuration (hyperparameter settings) as input.
   - Based on the configuration, create the neural network model (CNN or dense network).
2. Data Loading and Preprocessing:
   - Load the MNIST dataset.
   - Apply the necessary preprocessing steps (reshaping, normalization).
   - Create data loaders for training and validation.
3. Training Loop:
   - Set up the optimizer (e.g., Adam, SGD) with the learning rate from the configuration.
   - Iterate over the training data for a specified number of epochs (potentially controlled by the budget in multi-fidelity optimization).
   - Calculate the loss and update the model parameters.
4. Validation:
   - After each epoch (or at the end of training), evaluate the model on the validation set.
   - Calculate the validation accuracy or loss.
5. Return Value:
   - Return the validation loss (or 1 - validation accuracy) as the performance metric for SMAC to minimize.

Generated Configuration Space Code

```
from ConfigSpace import ConfigurationSpace, UniformFloatHyperparameter, UniformInt

def get_configspace() -> ConfigurationSpace:
    """
    Defines the configuration space for hyperparameter optimization of a CNN model
```

```
    Returns:
        ConfigurationSpace: The configuration space object.
    """
    cs = ConfigurationSpace()

    # Learning Rate
    learning_rate = UniformFloatHyperparameter(
        "learning_rate",
        lower=1e-4,
        upper=1e-1,
        default_value=1e-3,
        log=True
    )
    cs.add_hyperparameter(learning_rate)

    # Batch Size
    batch_size = CategoricalHyperparameter(
        "batch_size",
        choices=[32, 64, 128, 256],
        default_value=128
    )
    cs.add_hyperparameter(batch_size)

    # Optimizer
    optimizer = CategoricalHyperparameter(
        "optimizer",
        choices=["Adam", "SGD"],
        default_value="Adam"
    )
    cs.add_hyperparameter(optimizer)

    # Number of Convolutional Layers
    num_conv_layers = UniformIntegerHyperparameter(
        "num_conv_layers",
        lower=2,
        upper=5,
        default_value=3
    )
    cs.add_hyperparameter(num_conv_layers)

    # Dropout Rate
    dropout_rate = UniformFloatHyperparameter(
        "dropout_rate",
        lower=0.0,
        upper=0.5,
```

```
        default_value=0.2
    )
    cs.add_hyperparameter(dropout_rate)

    # Number of Epochs
    num_epochs = UniformIntegerHyperparameter(
        "num_epochs",
        lower=10,
        upper=50,
        default_value=20
    )
    cs.add_hyperparameter(num_epochs)

    # Number of Filters for the first Conv Layer
    num_filters_first_layer = CategoricalHyperparameter(
        "num_filters_first_layer",
        choices=[16, 32, 64],
        default_value=32
    )
    cs.add_hyperparameter(num_filters_first_layer)

    use_batch_norm = CategoricalHyperparameter(
        "use_batch_norm",
        choices=[True, False],
        default_value=True
    )
    cs.add_hyperparameter(use_batch_norm)

    return cs
```

Generated Scenario Code

```
from smac import Scenario
from ConfigSpace import ConfigurationSpace

def generate_scenario(cs: ConfigurationSpace) -> Scenario:
    """
    Generates a SMAC scenario configuration for hyperparameter optimization of an

    Args:
        cs (ConfigurationSpace): The configuration space from which to sample conf

    Returns:
        Scenario: A SMAC Scenario object configured for the image dataset.
```

```
    """

    scenario = Scenario(
        configspace=cs,
        name="HyperparameterOptimization",
        output_directory="./logs/gemini-2.0-flash_MNIST_20250617_182223",
        deterministic=False,
        n_trials=10,
        min_budget=1,
        max_budget=10,
        n_workers=1
    )
    return scenario
```

Generated Training Function Code

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from ConfigSpace import Configuration
from typing import Any
import numpy as np
from math import sqrt
import pandas as pd

def train(cfg: Configuration, dataset: Any, seed: int) -> float:
    """
    Trains a CNN model on an image dataset using PyTorch.

    Args:
        cfg (Configuration): Hyperparameter configuration.
        dataset (Any): Dictionary containing 'X' (features) and 'y' (labels).
        seed (int): Random seed for reproducibility.

    Returns:
        float: Negative validation accuracy.
    """
    torch.manual_seed(seed)

    # Extract hyperparameters
    lr = cfg.get('learning_rate')
    batch_size = cfg.get('batch_size')
    optimizer_name = cfg.get('optimizer')
```

```
    num_conv_layers = cfg.get('num_conv_layers')
    dropout_rate = cfg.get('dropout_rate')
    num_epochs = cfg.get('num_epochs')
    num_filters_first_layer = cfg.get('num_filters_first_layer')
    use_batch_norm = cfg.get('use_batch_norm')

    # Device configuration
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

    # Data preparation
    X, y = dataset['X'], dataset['y']

    # Reshape and normalize data
    if isinstance(X, pd.DataFrame):
        X = X.values
    if isinstance(y, pd.Series):
        y = y.values

    batch_size = int(batch_size)
    if not isinstance(batch_size, int) or batch_size <= 0:
        raise ValueError(f"batch_size should be a positive integer value, but got

    if len(X.shape) == 2:
        n_features = X.shape[1]
        height = width = int(sqrt(n_features))
        if height * height != n_features:
            raise ValueError("Input is not a square image")
        X = X.reshape(-1, 1, height, width)
    elif len(X.shape) == 3:
        X = X.reshape(-1, 1, X.shape[1], X.shape[2])
    elif len(X.shape) == 4:
        pass  # Assuming (N, C, H, W) or (N, H, W, C)
    else:
        raise ValueError("Invalid input shape")

    X = X.astype(np.float32) / 255.0
    y = y.astype(np.int64)

    X = torch.tensor(X, device=device)
    y = torch.tensor(y, device=device)

    # Split into training and validation (80/20 split)
    train_size = int(0.8 * len(X))
    X_train, X_val = X[:train_size], X[train_size:]
    y_train, y_val = y[:train_size], y[train_size:]
```

```
    train_dataset = torch.utils.data.TensorDataset(X_train, y_train)
    val_dataset = torch.utils.data.TensorDataset(X_val, y_val)

    train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=batch_siz
    val_loader = torch.utils.data.DataLoader(val_dataset, batch_size=batch_size, s


    # Model definition
    class CNN(nn.Module):
        def __init__(self, num_classes, num_conv_layers, dropout_rate, num_filters
            super(CNN, self).__init__()
            self.conv_layers = nn.ModuleList()
            self.batch_norm_layers = nn.ModuleList()  # Batch norm layers list

            in_channels = 1  # Assuming grayscale images
            num_filters = num_filters_first_layer

            for i in range(num_conv_layers):
                self.conv_layers.append(nn.Conv2d(in_channels, num_filters, kernel
                if use_batch_norm:
                    self.batch_norm_layers.append(nn.BatchNorm2d(num_filters))  #
                in_channels = num_filters
                num_filters = min(num_filters * 2, 128)  # Increase filters with e

            self.pool = nn.MaxPool2d(2, 2)
            self.dropout = nn.Dropout(dropout_rate)

            # Calculate the size of the flattened layer after convolutions
            self.flattened_size = self._calculate_flattened_size(1, X.shape[2], X.

            self.fc1 = nn.Linear(self.flattened_size, 128)  # Reduced FC layer siz
            self.fc2 = nn.Linear(128, num_classes)

            self.use_batch_norm = use_batch_norm

        def _calculate_flattened_size(self, in_channels, height, width, use_batch_
            # Simulate the convolutional layers to determine the output size
            x = torch.randn(1, in_channels, height, width)
            with torch.no_grad():
                for i, conv in enumerate(self.conv_layers):
                    x = F.relu(conv(x))
                    if use_batch_norm and i < len(self.batch_norm_layers):
                        x = self.batch_norm_layers[i](x)  # Apply batch norm if en
                    x = self.pool(x)
            return x.view(1, -1).size(1)
```

```
        def forward(self, x):
            for i, conv in enumerate(self.conv_layers):
                x = F.relu(conv(x))
                if self.use_batch_norm and i < len(self.batch_norm_layers):
                    x = self.batch_norm_layers[i](x)  # Apply batch norm if enable
                x = self.pool(x)
            x = torch.flatten(x, 1)
            x = self.dropout(x)
            x = F.relu(self.fc1(x))
            x = self.fc2(x)
            return x

    model = CNN(len(torch.unique(y)), num_conv_layers, dropout_rate, num_filters_f


    # Optimizer
    if optimizer_name == "Adam":
        optimizer = optim.Adam(model.parameters(), lr=lr)
    elif optimizer_name == "SGD":
        optimizer = optim.SGD(model.parameters(), lr=lr, momentum=0.9)
    else:
        raise ValueError(f"Unknown optimizer: {optimizer_name}")

    # Loss function
    criterion = nn.CrossEntropyLoss()

    # Training loop
    for epoch in range(num_epochs):
        model.train()
        for batch_idx, (data, target) in enumerate(train_loader):
            optimizer.zero_grad()
            output = model(data)
            loss = criterion(output, target)
            loss.backward()
            optimizer.step()

    # Validation
    model.eval()
    correct = 0
    total = 0
    with torch.no_grad():
        for data, target in val_loader:
            output = model(data)
            _, predicted = torch.max(output.data, 1)
            total += target.size(0)
            correct += (predicted == target).sum().item()
```

```
    accuracy = correct / total
    print(f"Validation Accuracy: {accuracy:.4f}")

    return -accuracy
```

AutoML Agent setup complete!

Loss Value

`-0.9873333333333333`

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 an image dataset.

Number of classes: 10
Class distribution:
1   6742
7   6265
3   6131
2   5958
9   5949
0   5923
6   5918
8   5851
4   5842
5   5421
Name: count, dtype: int64

Image Data Handling Requirements:
1. Input Format Requirements:
   - For CNN models: Input must be in (batch, channels, height, width) format
   - For dense/linear layers: Input should be flattened

2. Data Processing Steps:
   a) For flattened input (2D):
      - Calculate dimensions: height = width = int(sqrt(n_features))
      - Verify square dimensions: height * height == n_features
      - Reshape to (N, 1, H, W) for CNNs
   b) For 3D input (N, H, W):
      - Add channel dimension: reshape to (N, 1, H, W)

```
   c) For 4D input:
      - Verify channel order matches framework requirements

3. Framework-Specific Format:
   - PyTorch: (N, C, H, W)
   - TensorFlow: (N, H, W, C)
   - Convert between formats if necessary

4. Normalization:
   - Scale pixel values to [0, 1] by dividing by 255.0
   - Or standardize to mean=0, std=1
`
```

* Recommended Configuration based on the planner:
  * `To effectively train models on the MNIST dataset, consider these configurations:

1. Data Preprocessing:
    * Input Format: For CNNs, reshape input to (N, 1, 28, 28) for PyTorch or (N, 28, 28, 1) for TensorFlow. For dense layers, flatten the input to (N, 784).
    * Normalization: Scale pixel values to the range [0, 1] by dividing by 255.0.

2. Model Architecture:
    * CNN: A simple CNN with convolutional layers, ReLU activations, max pooling, and fully connected layers is a good starting point. The provided VGGNet configuration can serve as inspiration.
    * Dense Network: A multi-layer perceptron with ReLU activations can also be used, especially for baseline performance.

3. Training Parameters:
    * Batch Size: 128 is a commonly used batch size.
    * Epochs: Train for at least 10 epochs, but monitor validation loss to prevent overfitting.
    * Optimizer: Adam or SGD with momentum are suitable optimizers.
    * Learning Rate: Start with 0.001 and adjust as needed.

4. Regularization:
    * Dropout: Add dropout layers (e.g., rate of 0.5) to prevent overfitting.
    * Batch Normalization: Use batch normalization after convolutional layers.

5. SMAC Configuration:
    * Facade Type: MultiFidelityFacade is appropriate if considering varying training epochs or subsets of the data.
    * Budget Settings: Set min_budget to a small number of epochs (e.g., 5) and max_budget to the desired maximum number of epochs (e.g., 50).

```
    *   Number of Workers: Set n_workers based on available CPU/GPU resources.
Start with 4 and increase if possible.

6.  SMAC Scenario Parameters:
    *   walltime_limit: Set a reasonable time limit (e.g., 3600 seconds).
    *   n_trials: Set the number of trials based on the budget and time
constraints (e.g., 200).

Best parameters to set in the Configuration Space:

*   Learning Rate: UniformFloat(lower=1e-4, upper=1e-1, log=True,
default_value=1e-3)
*   Batch Size: Categorical([32, 64, 128, 256], default_value=128)
*   Optimizer: Categorical(["Adam", "SGD"], default_value="Adam")
*   Number of Convolutional Layers: Integer(lower=2, upper=5, default_value=3)
*   Dropout Rate: UniformFloat(lower=0.0, upper=0.5, default_value=0.2)
*   Number of Epochs: Integer(lower=10, upper=50, default_value=20)`

* 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
```

```
- 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 an image dataset.

Number of classes: 10
Class distribution:
1   6742
7   6265
3   6131
2   5958
9   5949
0   5923
6   5918
8   5851
4   5842
5   5421
Name: count, dtype: int64

Image Data Handling Requirements:
1. Input Format Requirements:
   - For CNN models: Input must be in (batch, channels, height, width) format
   - For dense/linear layers: Input should be flattened

2. Data Processing Steps:
   a) For flattened input (2D):
      - Calculate dimensions: height = width = int(sqrt(n_features))
      - Verify square dimensions: height * height == n_features
      - Reshape to (N, 1, H, W) for CNNs
   b) For 3D input (N, H, W):
      - Add channel dimension: reshape to (N, 1, H, W)
   c) For 4D input:

```
      - Verify channel order matches framework requirements

3. Framework-Specific Format:
   - PyTorch: (N, C, H, W)
   - TensorFlow: (N, H, W, C)
   - Convert between formats if necessary

4. Normalization:
   - Scale pixel values to [0, 1] by dividing by 255.0
   - Or standardize to mean=0, std=1
`

* 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
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_MNIST_20250617_182223" //this
```
```

```
        output_directory="./logs/gemini-2.0-flash_MNIST_20250617_182223",  # this is important and should not be changed
        deterministic=True,
        # other parameters based on the information
    )
    return scenario
```

Suggested Scenario Plan:

1. Data Preprocessing:
    * Reshape input data to the appropriate format (N, 1, 28, 28) for CNNs or flatten to (N, 784) for dense layers.
    * Normalize pixel values by dividing by 255.0.

2. Model Definition:
    * Define a CNN or dense network using a configuration space. Allow SMAC to tune hyperparameters like the number of layers, filter sizes, and dropout rates.

3. SMAC Scenario Configuration:
    * Use MultiFidelityFacade for multi-fidelity optimization to potentially reduce optimization time.
    * Set min_budget and max_budget for the number of epochs.
    * Set n_workers based on available resources.
    * Set walltime_limit and n_trials to control the optimization process.

4. Special Considerations:
    * MNIST is a relatively simple dataset, so more complex models may not be necessary.
    * Ensure proper data loading and batching to avoid memory issues.
    * Monitor training and validation loss to prevent overfitting.

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_MNIST_20250617_182223"
* 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="

```
    tr_lat_wattime_thrft',tr_lat_memory_thrft=
```

```
"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 an image dataset.

Number of classes: 10
Class distribution:
1   6742
7   6265
3   6131
2   5958
9   5949
0   5923
6   5918
8   5851
4   5842
5   5421
Name: count, dtype: int64

Image Data Handling Requirements:
1. Input Format Requirements:
   - For CNN models: Input must be in (batch, channels, height, width) format
   - For dense/linear layers: Input should be flattened

2. Data Processing Steps:
   a) For flattened input (2D):
      - Calculate dimensions: height = width = int(sqrt(n_features))
      - Verify square dimensions: height * height == n_features
      - Reshape to (N, 1, H, W) for CNNs
   b) For 3D input (N, H, W):
      - Add channel dimension: reshape to (N, 1, H, W)
   c) For 4D input:

```
      - Verify channel order matches framework requirements

3. Framework-Specific Format:
   - PyTorch: (N, C, H, W)
   - TensorFlow: (N, H, W, C)
   - Convert between formats if necessary

4. Normalization:
   - Scale pixel values to [0, 1] by dividing by 255.0
   - Or standardize to mean=0, std=1
`
  * ConfigSpace Definition: `from ConfigSpace import ConfigurationSpace,
UniformFloatHyperparameter, UniformIntegerHyperparameter,
CategoricalHyperparameter, InCondition

def get_configspace() -> ConfigurationSpace:
    """
    Defines the configuration space for hyperparameter optimization of a CNN
model.

    Returns:
        ConfigurationSpace: The configuration space object.
    """
    cs = ConfigurationSpace()

    # Learning Rate
    learning_rate = UniformFloatHyperparameter(
        "learning_rate",
        lower=1e-4,
        upper=1e-1,
        default_value=1e-3,
        log=True
    )
    cs.add_hyperparameter(learning_rate)

    # Batch Size
    batch_size = CategoricalHyperparameter(
        "batch_size",
        choices=[32, 64, 128, 256],
        default_value=128
    )
    cs.add_hyperparameter(batch_size)

    # Optimizer
    optimizer = CategoricalHyperparameter(
        "optimizer",
        choices=["Adam", "SGD"],
```

```
        default_value="Adam"
    )
    cs.add_hyperparameter(optimizer)

    # Number of Convolutional Layers
    num_conv_layers = UniformIntegerHyperparameter(
        "num_conv_layers",
        lower=2,
        upper=5,
        default_value=3
    )
    cs.add_hyperparameter(num_conv_layers)

    # Dropout Rate
    dropout_rate = UniformFloatHyperparameter(
        "dropout_rate",
        lower=0.0,
        upper=0.5,
        default_value=0.2
    )
    cs.add_hyperparameter(dropout_rate)

    # Number of Epochs
    num_epochs = UniformIntegerHyperparameter(
        "num_epochs",
        lower=10,
        upper=50,
        default_value=20
    )
    cs.add_hyperparameter(num_epochs)

    # Number of Filters for the first Conv Layer
    num_filters_first_layer = CategoricalHyperparameter(
        "num_filters_first_layer",
        choices=[16, 32, 64],
        default_value=32
    )
    cs.add_hyperparameter(num_filters_first_layer)

    use_batch_norm = CategoricalHyperparameter(
        "use_batch_norm",
        choices=[True, False],
        default_value=True
    )
    cs.add_hyperparameter(use_batch_norm)

    return cs
```

```
    return cs
```

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

def generate_scenario(cs: ConfigurationSpace) -> Scenario:
    """
    Generates a SMAC scenario configuration for hyperparameter optimization of an image dataset.

    Args:
        cs (ConfigurationSpace): The configuration space from which to sample configurations.

    Returns:
        Scenario: A SMAC Scenario object configured for the image dataset.
    """

    scenario = Scenario(
        configspace=cs,
        name="HyperparameterOptimization",
        output_directory="./logs/gemini-2.0-flash_MNIST_20250617_182223",
        deterministic=False,
        n_trials=10,
        min_budget=1,
        max_budget=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
  * Implement proper model initialization
  * Use appropriate loss functions
  * Apply proper regularization

```
  * 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
```

```
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
  1. Define the Model:
    * The train function should take a configuration (hyperparameter settings) as input.
    * Based on the configuration, create the neural network model (CNN or dense network).

2. Data Loading and Preprocessing:
    * Load the MNIST dataset.
    * Apply the necessary preprocessing steps (reshaping, normalization).
    * Create data loaders for training and validation.

3. Training Loop:
    * Set up the optimizer (e.g., Adam, SGD) with the learning rate from the configuration.
    * Iterate over the training data for a specified number of epochs (potentially controlled by the budget in multi-fidelity optimization).
    * Calculate the loss and update the model parameters.

4. Validation:
    * After each epoch (or at the end of training), evaluate the model on the validation set.
    * Calculate the validation accuracy or loss.

5. Return Value:
    * Return the validation loss (or 1 - validation accuracy) as the performance metric for SMAC to minimize.
"
}

Download Generated Code and Prompts