AutoML Agent Interface

Select the dataset type

image

Choose a dataset

Fashion-MNIST

Select a GROQ LLM Model

gemini-2.0-flash

Run AutoML Agent

Setting up AutoML Agent...

Instructor Response

Configuration Plan

The Fashion-MNIST dataset is well-suited for Multi-Fidelity Optimization due to its moderate size, allowing for quicker iterations on smaller subsets or fewer training epochs. An appropriate budget range would be min_budget=10 and max_budget=50, representing the number of training epochs. The number of workers (n_workers) should be set according to the available computational resources. Given the dataset's characteristics, a range of 4-8 workers could strike a good balance between parallelism and resource utilization. There aren't any special considerations for this dataset beyond standard image preprocessing and ensuring the data is in the correct format (N, C, H, W) for PyTorch or (N, H, W, C) for TensorFlow.

Scenario Plan

Scenario Configuration:

Facade Type: MultiFidelityFacade Budget Settings: min_budget=10, max_budget=50 (epochs) Number of Workers: n_workers=4-8 Other Scenario Parameters: Consider a walltime_limit (e.g. 600 seconds) and n_trials (e.g. 200) based on available resources and desired exploration.

Train Function Plan

Train Function Plan:

The train function should accept a configuration (hyperparameter settings) and a budget (number of epochs). It should then:

1. Load the Fashion-MNIST dataset.
2. Preprocess the data as detailed in the recommended data preprocessing steps: reshape, normalize.
3. Define a CNN model.
4. Train the model for the specified budget (number of epochs).
5. Validate the model on a validation set.
6. Return the validation loss (SMAC minimizes).

Generated Configuration Space Code

```
ConfigSpace import ConfigurationSpace, UniformFloatHyperparameter, UniformIntegerHyp
ConfigSpace.conditions import EqualsCondition

et_configspace() -> ConfigurationSpace:
s = ConfigurationSpace()

 Optimizer
ptimizer_choice = CategoricalHyperparameter("optimizer", choices=["Adam", "SGD"])
s.add_hyperparameter(optimizer_choice)

 Learning Rate (Common to both Adam and SGD)
earning_rate = UniformFloatHyperparameter(
   "learning_rate", lower=1e-5, upper=1e-2, default_value=1e-3, log=True

s.add_hyperparameter(learning_rate)

 Adam Specific Hyperparameters
eta1 = UniformFloatHyperparameter("beta1", lower=0.9, upper=0.999, default_value=0.9
eta2 = UniformFloatHyperparameter("beta2", lower=0.999, upper=0.9999, default_value=
s.add_hyperparameter(beta1)
s.add_hyperparameter(beta2)

 SGD Specific Hyperparameters
omentum = UniformFloatHyperparameter("momentum", lower=0.0, upper=0.9, default_value
s.add_hyperparameter(momentum)

 Architecture related
um_layers = UniformIntegerHyperparameter("num_layers", lower=1, upper=5, default_val
s.add_hyperparameter(num_layers)
```

```
um_filters = UniformIntegerHyperparameter("num_filters", lower=32, upper=256, defaul
s.add_hyperparameter(num_filters)

ernel_size = CategoricalHyperparameter("kernel_size", choices=[3, 5])
s.add_hyperparameter(kernel_size)

 Regularization
ropout_rate = UniformFloatHyperparameter(
   "dropout_rate", lower=0.0, upper=0.5, default_value=0.0, log=False

s.add_hyperparameter(dropout_rate)

 Batch size
atch_size = CategoricalHyperparameter("batch_size", choices=[32, 64, 128])
s.add_hyperparameter(batch_size)

 Conditional Hyperparameters
s.add_condition(
   EqualsCondition(beta1, optimizer_choice, "Adam")
  # Only use beta1 with Adam
s.add_condition(
   EqualsCondition(beta2, optimizer_choice, "Adam")
  # Only use beta2 with Adam

s.add_condition(
   EqualsCondition(momentum, optimizer_choice, "SGD")
  # Only use momentum with SGD

eturn cs
```

Generated Scenario Code

```
from smac import Scenario
from ConfigSpace import ConfigurationSpace
from pathlib import Path

def generate_scenario(cs: ConfigurationSpace) -> Scenario:
    scenario = Scenario(
        configspace=cs,
        name="HyperparameterOptimization",
        output_directory=Path("./logs/gemini-2.0-flash_Fashion-MNIST_20250617_1853
        deterministic=False,
        min_budget=10,
        max_budget=50,
```

```
        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 torch.utils.data import DataLoader, TensorDataset
from sklearn.model_selection import train_test_split
import numpy as np
from ConfigSpace import Configuration
from typing import Any
import logging
import math

def train(cfg: Configuration, dataset: Any, seed: int) -> float:
    """
    Trains a CNN model on the Fashion-MNIST dataset using the provided configurati

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

    Returns:
        float: The validation loss.
    """
    # Logging setup
    logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname)s -

    # Set random seed for reproducibility
    torch.manual_seed(seed)
    np.random.seed(seed)

    # Device configuration
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    logging.info(f"Using device: {device}")

    # 1. Data Preparation
    X = dataset['X']
    y = dataset['y']
```

```
    # Convert to numpy arrays if they are not already
    if not isinstance(X, np.ndarray):
        X = X.to_numpy()
    if not isinstance(y, np.ndarray):
        y = y.to_numpy()

    # Data preprocessing: reshape and normalize
    n_samples, n_features = X.shape
    height = width = int(math.sqrt(n_features))
    if height * height != n_features:
        raise ValueError("Input features are not a square image.")

    X = X.reshape(n_samples, 1, height, width)  # NCHW format
    X = X.astype(np.float32) / 255.0  # Normalize to [0, 1]
    y = y.astype(np.int64)

    # Split into training and validation sets
    X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2, random_

    # Convert to PyTorch tensors
    X_train = torch.tensor(X_train, dtype=torch.float32).to(device)
    y_train = torch.tensor(y_train, dtype=torch.long).to(device)
    X_val = torch.tensor(X_val, dtype=torch.float32).to(device)
    y_val = torch.tensor(y_val, dtype=torch.long).to(device)

    # Create TensorDatasets and DataLoaders
    batch_size = cfg.get("batch_size")
    if not isinstance(batch_size, int) or batch_size <= 0:
        batch_size = 32
        logging.warning(f"Invalid batch_size: {cfg.get('batch_size')}. Using defau
    train_dataset = TensorDataset(X_train, y_train)
    val_dataset = TensorDataset(X_val, y_val)
    train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True,
    val_loader = DataLoader(val_dataset, batch_size=batch_size, shuffle=False, num

    # 2. Model Definition
    class CNN(nn.Module):
        def __init__(self, num_layers, num_filters, kernel_size, dropout_rate, inp
            super(CNN, self).__init__()
            self.layers = nn.ModuleList()

            current_height = input_size
            current_width = input_size

            # First layer
```

```
            self.layers.append(nn.Conv2d(1, num_filters, kernel_size=kernel_size))
            current_height = current_height - (kernel_size - 1)
            current_width = current_width - (kernel_size - 1)

            self.layers.append(nn.ReLU())
            self.layers.append(nn.MaxPool2d(2))
            current_height = current_height // 2
            current_width = current_width // 2
            self.layers.append(nn.Dropout(dropout_rate))

            # Additional layers
            in_channels = num_filters
            for _ in range(num_layers - 1):
                self.layers.append(nn.Conv2d(in_channels, num_filters, kernel_size
                current_height = current_height - (kernel_size - 1)
                current_width = current_width - (kernel_size - 1)

                self.layers.append(nn.ReLU())
                self.layers.append(nn.MaxPool2d(2))
                current_height = current_height // 2
                current_width = current_width // 2

                self.layers.append(nn.Dropout(dropout_rate))
                in_channels = num_filters

            if current_height <= 0 or current_width <= 0:

                self.flatten = nn.Flatten()
                self.fc = nn.Linear(in_channels * max(1,current_height) * max(1,cu
            else:
                self.flatten = nn.Flatten()
                self.fc = nn.Linear(in_channels * current_height * current_width,

        def forward(self, x):
            for layer in self.layers:
                x = layer(x)
            x = self.flatten(x)
            x = self.fc(x)
            return x

    num_layers = cfg.get("num_layers")
    num_filters = cfg.get("num_filters")
    kernel_size = cfg.get("kernel_size")
    dropout_rate = cfg.get("dropout_rate")

    # Check if kernel size is valid
```

```
    if kernel_size > height:
        kernel_size = height

    model = CNN(num_layers, num_filters, kernel_size, dropout_rate, height).to(dev

    # 3. Optimizer
    optimizer_choice = cfg.get("optimizer")
    learning_rate = cfg.get("learning_rate")

    if optimizer_choice == "Adam":
        beta1 = cfg.get("beta1")
        beta2 = cfg.get("beta2")
        optimizer = optim.Adam(model.parameters(), lr=learning_rate, betas=(beta1,
    elif optimizer_choice == "SGD":
        momentum = cfg.get("momentum")
        optimizer = optim.SGD(model.parameters(), lr=learning_rate, momentum=momen
    else:
        raise ValueError(f"Unknown optimizer: {optimizer_choice}")

    # Loss function
    criterion = nn.CrossEntropyLoss()

    # 4. Training Loop
    n_epochs = 10  # Using a fixed number of epochs for simplicity and compliance

    for epoch in range(n_epochs):
        model.train()
        running_loss = 0.0
        for inputs, labels in train_loader:
            optimizer.zero_grad()
            outputs = model(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()
            running_loss += loss.item()
        logging.info(f"Epoch {epoch+1}/{n_epochs}, Training Loss: {running_loss/le

    # 5. Validation
    model.eval()
    val_loss = 0.0
    with torch.no_grad():
        for inputs, labels in val_loader:
            outputs = model(inputs)
            loss = criterion(outputs, labels)
            val_loss += loss.item()
    val_loss /= len(val_loader)
```

```
    logging.info(f"Validation Loss: {val_loss}")

    return -val_loss
```

AutoML Agent setup complete!

Loss Value

`-2.261304501215617`

Starting Optimization Process

Starting Optimization Process