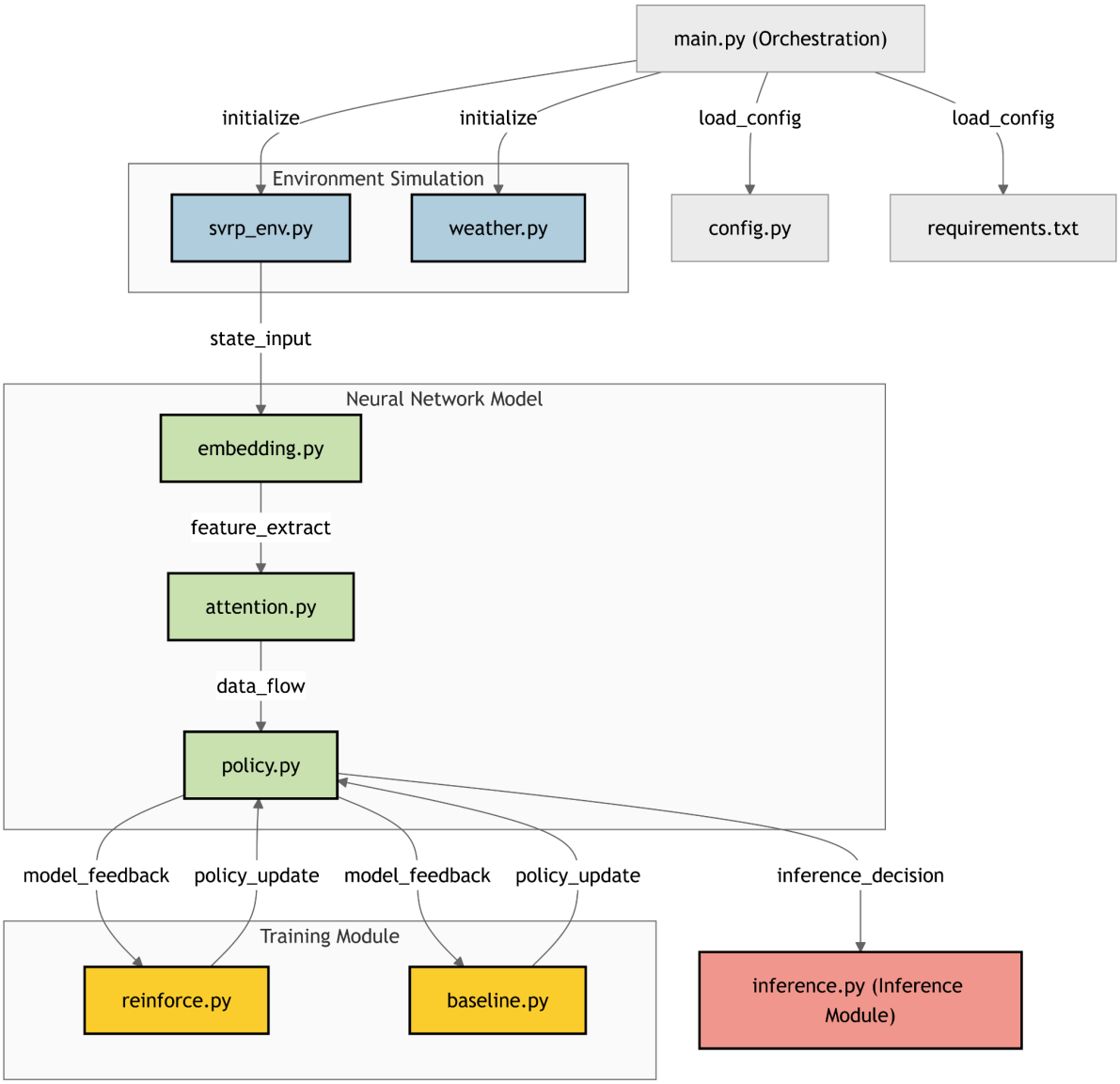
Code explanation: GitHub - Zangir/SVRP



main.py

1. Argument Parsing (`parse_args`)

- The script uses `argparse` to define and parse command-line arguments. These arguments allow users to configure various aspects of the environment, model, training, and inference. Examples include:
  - Environment settings: `--num_nodes`, `--num_vehicles`, `--capacity`, etc.
  - Model settings: `--embedding_dim`.
  - Training settings: `--epochs`, `--batch_size`, `--lr`, etc.
  - Inference settings: `--inference`, `--beam_width`, etc.
  - Other options: `--cuda` (to use GPU), `--test` (to run in test mode), and `--load_model` (to load a pre-trained model).

2. Logger Setup (`setup_logger`)

- A logger is configured to log messages to both the console and a file (`svrp_log.txt`) in the specified save directory. This helps track the progress of training and evaluation.

3. Environment Creation

- The script creates an instance of the `SVRPEnvironment` class, which simulates the stochastic vehicle routing problem. The environment handles:
  - Customer demands.
  - Vehicle capacities and positions.
  - Weather effects on demands and travel costs.

4. Model Initialization

- The `SVRPPolicy` model is instantiated. This is the neural network that learns to make routing decisions based on the environment's state.
- The `ReinforceTrainer` is created to train the policy model using the REINFORCE algorithm with a baseline to reduce variance.

5. Training (`train`)

- If the script is not in test mode (`--test` is not set), the `train` function is called to train the model:
  - The model interacts with the environment to collect data and update its parameters.
  - Metrics like rewards, policy loss, and baseline loss are logged and plotted.
  - The model is periodically saved to the checkpoints directory.

6. **Evaluation (`evaluate`)**

- The script evaluates the trained model using one of the specified inference strategies:
    - **GreedyInference**: Selects the most probable action at each step.
    - **RandomSamplingInference**: Samples multiple solutions and selects the best one.
    - **BeamSearchInference**: Maintains the top-k most probable solutions during inference.
- The evaluation results include the routes taken by vehicles, the total cost, and the mean reward across test instances.

7. **Visualization (`visualize_route`)**

- The script can visualize the routes taken by vehicles during evaluation. It plots the depot, customer locations, and the paths taken by each vehicle.

8. **Main Function (`main`)**

- The `main` function ties everything together:
    - Parses command-line arguments.
    - Sets the random seed for reproducibility.
    - Initializes the environment, model, and trainer.
    - Loads a pre-trained model if specified.
    - Runs training or evaluation based on the `--test` flag.
    - Logs the final evaluation results.

Summary

The main.py script is a comprehensive pipeline for solving the SVRP using reinforcement learning. It supports:

- Training a policy model to optimize vehicle routes under stochastic conditions.
- Evaluating the model using different inference strategies.
- Visualizing the results and saving models for future use.

Environment simulation

srvp_env.py

The svrp_env.py file defines the `SVRPEnvironment` class, which simulates the **Stochastic Vehicle Routing Problem (SVRP)**. This environment is used to model the problem, handle state transitions, compute rewards, and manage stochastic variables like weather and customer demands. It serves as the interface between the reinforcement learning agent and the problem domain.

1. Initialization (`__init__`)

- The constructor initializes the environment with the following parameters:
  - `num_nodes`: Total number of nodes (customers + depot).
  - `num_vehicles`: Number of vehicles available for routing.
  - `capacity`: Maximum capacity of each vehicle.
  - `weather_dim`: Dimensionality of weather variables affecting the problem.
  - **Ratios (`a_ratio`, `b_ratio`, `gamma_ratio`)**: Define the contribution of constant, weather, and noise components to stochastic variables.
  - `device`: Specifies whether to use CPU or GPU for computations.
- A `WeatherSimulation` instance is created to generate weather, demands, and travel costs.
- The dimensions of customer and vehicle features are calculated:
  - **Customer features**: Weather, demand, and travel costs.
  - **Vehicle features**: Position and load.

2. Resetting the Environment (`reset`)

- Prepares the environment for a new episode:
  - Generates weather, demands, and travel costs using the `WeatherSimulation`.
  - Initializes:
    - **Remaining demands**: Demand for each customer.
    - **Vehicle positions**: All vehicles start at the depot (node 0).
    - **Vehicle loads**: All vehicles start with full capacity.
  - Returns the initial state:
    - **Customer features**: Weather, demand, and travel costs.
    - **Vehicle features**: Position and load.
    - **Remaining demands**: Demand for each customer.

3. Step Function (`step`)

- Executes actions (vehicle movements) and updates the environment state:
  - **Actions**: A tensor specifying the next node each vehicle will visit.

    - For each vehicle:
        - Computes travel costs based on the current and next positions.
        - Updates vehicle loads and remaining demands based on deliveries.
        - Handles refilling at the depot if a vehicle runs out of capacity.
    - Updates vehicle positions and increments the step counter.
    - Checks if the episode is complete (all demands are fulfilled).
    - Returns:
        - Updated customer and vehicle features.
        - Remaining demands.
        - Rewards (negative travel costs).
        - Done flag (whether the episode is complete).

4. **Feature Construction (`_get_features`)**

- Constructs feature tensors for the current state:
    - **Customer features**:
        - Weather variables for each node.
        - Remaining demand for each customer.
        - Travel costs to all other nodes.
    - **Vehicle features**:
        - Current position (as a one-hot encoding or index).
        - Normalized load (current load divided by capacity).

Summary

The `SVRPEnvironment` class provides a simulation of the SVRP, allowing the reinforcement learning agent to interact with it. It handles:

- **State transitions**: Updates the environment based on the agent's actions.
- **Reward computation**: Calculates negative travel costs as rewards.
- **Stochastic variables**: Incorporates weather and noise into the problem dynamics.
- **Feature extraction**: Provides the agent with structured input data (customer and vehicle features).

This environment is essential for training and evaluating the policy model in the SVRP-RL framework.

weather.py

The weather.py file defines the `WeatherSimulation` class, which is responsible for simulating weather variables and their influence on customer demands and travel costs in the **Stochastic Vehicle Routing Problem (SVRP)**. This class introduces stochasticity into the problem by modeling the effects of weather and noise on the environment.

1. Initialization (`__init__`)

- The constructor initializes the simulation with the following parameters:
  - `weather_dim`: Number of weather variables (e.g., temperature, pressure, humidity).
  - `a_ratio`: Proportion of the constant component in stochastic variables.
  - `b_ratio`: Proportion of the weather-dependent component in stochastic variables.
  - `gamma_ratio`: Proportion of the noise component in stochastic variables.
  - `seed`: Optional random seed for reproducibility.
- If a seed is provided, it sets the random seed for both NumPy and PyTorch.
- `fixed_customer_positions`: Stores fixed customer positions for deterministic scenarios.

2. Generate Weather, Demands, and Travel Costs (`generate`)

- This method generates weather variables, customer demands, and travel costs for a batch of scenarios.
- **Inputs**:
  - `batch_size`: Number of scenarios to generate.
  - `num_nodes`: Number of nodes (customers + depot).
  - `fixed_customers`: Whether to use fixed customer positions.
  - `device`: Device for tensor operations (CPU or GPU).
- **Outputs**:
  - `weather`: A tensor of shape `[batch_size, weather_dim]` representing weather variables.
  - `demands`: A tensor of shape `[batch_size, num_nodes]` representing customer demands.
  - `travel_costs`: A tensor of shape `[batch_size, num_nodes, num_nodes]` representing travel costs between nodes.
- **Steps**:
  1. **Weather Variables**:
     - Generates random weather variables for each scenario, sampled uniformly from `[-1, 1]`.
  2. **Customer Positions**:
     - If `fixed_customers` is `True` and positions are already stored, reuses them.

    - Otherwise, generates random positions for customers (node 0 is always the depot at `(0.5, 0.5)`).
    - Saves positions if `fixed_customers` is `True`.
3. **Demands**:
    - Computes demands for each customer based on:
        - **Constant Component**: A fixed base demand scaled by `a_ratio`.
        - **Weather Component**: Interaction terms between weather variables scaled by `b_ratio`.
        - **Noise Component**: Random noise scaled by `gamma_ratio`.
    - Ensures demands are positive (minimum value of 1.0).
4. **Travel Costs**:
    - Computes travel costs between nodes based on:
        - **Base Cost**: Euclidean distance between nodes, scaled for numerical stability.
        - **Constant Component**: Base cost scaled by `a_ratio`.
        - **Weather Component**: Interaction terms between weather variables scaled by `b_ratio`.
        - **Noise Component**: Random noise scaled by `gamma_ratio`.
    - Ensures travel costs are positive (minimum value of 0.1).

3. Generate Dataset (`generate_dataset`)

- Generates a dataset of scenarios for tasks like k-NN estimation or pretraining.
- **Inputs**:
    - **`num_scenarios`**: Number of scenarios to generate.
    - **`num_nodes`**: Number of nodes (customers + depot).
    - **`fixed_customers`**: Whether to use fixed customer positions.
    - **`device`**: Device for tensor operations.
- **Output**:
    - A dictionary containing:
        - **`weather`**: Weather variables for all scenarios.
        - **`demands`**: Customer demands for all scenarios.
        - **`travel_costs`**: Travel costs for all scenarios.

Summary

The `WeatherSimulation` class introduces stochasticity into the SVRP by simulating:

- **Weather variables**: Random factors like temperature, pressure, and humidity.
- **Customer demands**: Affected by constant, weather-dependent, and noise components.
- **Travel costs**: Influenced by distances, weather, and noise.

This class is essential for creating realistic and dynamic scenarios for training and evaluating the reinforcement learning agent in the SVRP environment.

Neural Network model

embedding.py

The embedding.py file defines two classes, `CustomerEncoder` and `VehicleEncoder`, which are responsible for encoding customer and vehicle information into embeddings. These embeddings are used by the policy model (`SVRPPolicy`) to make routing decisions in the **Stochastic Vehicle Routing Problem (SVRP)**.

1. CustomerEncoder

- **Purpose**: Encodes customer-related features (e.g., weather variables, demand, travel costs) into embeddings.
- **Architecture**:
  - Uses a 1D convolutional layer (`nn.Conv1d`) to process customer features.
  - The convolutional layer transforms the input features into embeddings of a specified dimension (`embedding_dim`).
- **Inputs**:
  - **`x`**: A tensor of shape `[batch_size, num_nodes, input_dim]`, where:
    - `batch_size`: Number of scenarios in the batch.
    - `num_nodes`: Number of nodes (customers + depot).
    - `input_dim`: Dimension of customer features (e.g., weather, demand, travel costs).
- **Outputs**:
  - A tensor of shape `[batch_size, num_nodes, embedding_dim]` containing the encoded embeddings for each customer.
- **Forward Pass**:
  1. Reshapes the input tensor for 1D convolution.
  2. Applies the convolutional layer to extract embeddings.
  3. Reshapes the output back to the original batch and node dimensions.

2. VehicleEncoder

- **Purpose**: Encodes vehicle-related features (e.g., position, load) into embeddings using an LSTM.
- **Architecture**:
  - Uses an LSTM (`nn.LSTM`) to process sequential vehicle features.
  - The LSTM outputs embeddings of a specified dimension (`embedding_dim`).
- **Inputs**:
  - **`x`**: A tensor of shape `[batch_size, num_vehicles, input_dim]`, where:
    - `batch_size`: Number of scenarios in the batch.
    - `num_vehicles`: Number of vehicles.
    - `input_dim`: Dimension of vehicle features (e.g., position, load).

    - `hidden`: Optional previous hidden state for sequential processing.
- **Outputs**:
    - A tensor of shape `[batch_size, num_vehicles, embedding_dim]` containing the encoded embeddings for each vehicle.
    - The updated hidden state of the LSTM.
- **Forward Pass**:
    1. Reshapes the input tensor for LSTM processing.
    2. Initializes the hidden state if not provided.
    3. Applies the LSTM to extract embeddings.
    4. Reshapes the output back to the original batch and vehicle dimensions.

Summary

The embedding.py file provides the building blocks for encoding customer and vehicle information into embeddings:

- **CustomerEncoder**: Encodes customer features using a 1D convolutional layer.
- **VehicleEncoder**: Encodes vehicle features using an LSTM.

These embeddings are critical for the `SVRPPolicy` model, as they represent the state of the environment in a compact and meaningful way, enabling the policy to make informed routing decisions.

attention.py

The attention.py file defines two key components, `AttentionLayer` and `MaskingLayer`, which are used in the **Stochastic Vehicle Routing Problem (SVRP)** to compute probabilities for selecting the next node for each vehicle. These components are critical for the policy model (`SVRPPolicy`) to make routing decisions based on customer and vehicle embeddings.

1. AttentionLayer

- **Purpose**: Implements an attention mechanism to compute probabilities for selecting the next node for each vehicle based on customer and vehicle embeddings.
- **Architecture**:
  - **Query (`q`)**: Derived from vehicle embeddings (memory embeddings).
  - **Key (`k`)** and **Value (`v`)**: Derived from customer embeddings (state embeddings).
  - **Dot-Product Attention**: Computes attention scores between queries and keys, scaled by the square root of the embedding dimension.
  - **Masking**: Optionally applies a mask to exclude invalid nodes (e.g., nodes with zero demand).
  - **Softmax**: Converts attention scores into probabilities.
- **Inputs**:
  - **`state_embeddings`**: Tensor of shape `[batch_size, num_nodes, embedding_dim]`, representing customer information.
  - **`memory_embeddings`**: Tensor of shape `[batch_size, num_vehicles, embedding_dim]`, representing vehicle information.
  - **`mask`**: Optional boolean tensor of shape `[batch_size, num_nodes]`, where `True` indicates nodes that should be excluded.
- **Outputs**:
  - **`probs`**: Tensor of shape `[batch_size, num_vehicles, num_nodes]`, containing probabilities for each node being selected as the next position for each vehicle.
- **Forward Pass**:
  1. **Compute Query, Key, and Value**:
     - `q` is computed from `memory_embeddings` (vehicles).
     - `k` and `v` are computed from `state_embeddings` (customers).
  2. **Attention Scores**:
     - Computes scaled dot-product attention scores between `q` and `k`.
  3. **Masking**:
     - Applies the mask to exclude invalid nodes by setting their scores to a very large negative value (`-1e9`).
  4. **Softmax**:
     - Converts the masked scores into probabilities using the softmax function.

5. **Output**:
    - Returns the probabilities for each node being selected.

2. **MaskingLayer**

- **Purpose**: Creates a mask to prevent the attention layer from selecting nodes that have already been visited or whose demand has been fulfilled.
- **Inputs**:
    - **`demands`**: Tensor of shape `[batch_size, num_nodes]`, representing the remaining demand for each node.
- **Outputs**:
    - **`mask`**: Boolean tensor of shape `[batch_size, num_nodes]`, where `True` indicates nodes that should be masked (e.g., nodes with zero demand).
- **Forward Pass**:
    1. **Mask Nodes with Zero Demand**:
        - Creates a mask where nodes with zero or negative demand are marked as `True`.
    2. **Exclude Depot**:
        - Ensures that the depot (node 0) is never masked, even if its demand is zero.
    3. **Output**:
        - Returns the mask.

Summary

The attention.py file provides the attention mechanism and masking logic for the policy model:

- **`AttentionLayer`**: Computes probabilities for selecting the next node for each vehicle using an attention mechanism based on customer and vehicle embeddings.
- **`MaskingLayer`**: Ensures that invalid nodes (e.g., nodes with zero demand) are excluded from the attention mechanism.

These components are essential for enabling the policy model to focus on valid routing options and make informed decisions in the SVRP.

policy.py

The policy.py file defines the `SVRPPolicy` class, which is the neural network model used to make routing decisions in the **Stochastic Vehicle Routing Problem (SVRP)**. This policy model combines customer and vehicle information, processes it through encoders, and uses an attention mechanism to compute probabilities for selecting the next node for each vehicle.

1. Overview

- The `SVRPPolicy` class is a PyTorch `nn.Module` that serves as the policy network for reinforcement learning.
- It integrates:
  - **Customer Encoder**: Encodes customer-related features (e.g., weather, demand, travel costs).
  - **Vehicle Encoder**: Encodes vehicle-related features (e.g., position, load).
  - **Attention Layer**: Computes attention scores to determine the next node for each vehicle.
  - **Masking Layer**: Masks nodes that are no longer valid (e.g., fulfilled demands).

2. Initialization (`__init__`)

- The constructor initializes the following components:
  - **Customer Encoder**: Encodes customer features into embeddings of size `embedding_dim`.
  - **Vehicle Encoder**: Encodes vehicle features into embeddings of size `embedding_dim`.
  - **Attention Layer**: Computes attention scores between customer and vehicle embeddings with AttentionLayer.
  - **Masking Layer**: Creates masks to exclude invalid nodes (e.g., nodes with zero demand) with MaskinLayer.
- The `embedding_dim` parameter determines the size of the embeddings used throughout the model.

3. Forward Pass (`forward`)

- The `forward` method processes the input features and computes log probabilities for selecting the next node for each vehicle.
- **Inputs**:
  - **`customer_features`**: Tensor of shape `[batch_size, num_nodes, customer_input_dim]` containing customer-related features (e.g., weather, demand, travel costs).

    - **`vehicle_features`**: Tensor of shape `[batch_size, num_vehicles, vehicle_input_dim]` containing vehicle-related features (e.g., position, load).
    - **`demands`**: Tensor of shape `[batch_size, num_nodes]` representing the remaining demand for each node.
    - **`hidden`**: Optional hidden state for sequential processing in the vehicle encoder.
- **Steps**:
    1. **Customer Encoding**: Encodes customer features into embeddings using the `CustomerEncoder`.
    2. **Vehicle Encoding**: Encodes vehicle features into embeddings using the `VehicleEncoder`. Optionally updates the hidden state.
    3. **Masking**: Creates a mask to exclude nodes with zero demand or invalid nodes using the MaskingLayer.
    4. **Attention**: Computes attention scores between customer and vehicle embeddings, applying the mask to exclude invalid nodes using the AttentionLayer.
    5. **Log Probabilities**: Converts attention scores into log probabilities for numerical stability.
- **Outputs**:
    - **`log_probs`**: Tensor of shape `[batch_size, num_vehicles, num_nodes]` containing log probabilities for selecting each node.
    - **`hidden`**: Updated hidden state for sequential processing.

4. Action Sampling (`sample_action`)

- The `sample_action` method selects the next node for each vehicle based on the computed log probabilities.
- **Inputs**:
    - **`log_probs`**: Tensor of shape `[batch_size, num_vehicles, num_nodes]` containing log probabilities.
    - **`greedy`**: If `True`, selects the most probable action (greedy approach). Otherwise, samples actions probabilistically.
- **Outputs**:
    - **`actions`**: Tensor of shape `[batch_size, num_vehicles]` containing the selected node indices for each vehicle.
- **Steps**:
    1. If `greedy` is `True`, selects the node with the highest probability for each vehicle.
    2. Otherwise, samples actions from the probability distribution derived from the log probabilities.

Summary

The `SVRPPolicy` class is the core decision-making model for the SVRP. It:
- Encodes customer and vehicle features into embeddings.

- Uses an attention mechanism to compute probabilities for selecting the next node for each vehicle.
- Supports both greedy and probabilistic action selection.

This policy model is trained using reinforcement learning to optimize vehicle routing decisions under stochastic conditions.

Training module

reinforce.py

The reinforce.py file implements the **REINFORCE algorithm** for training the policy network in the **Stochastic Vehicle Routing Problem (SVRP)**. The `ReinforceTrainer` class is responsible for optimizing the policy model using policy gradients, with a baseline to reduce variance, and includes entropy regularization to encourage exploration.

1. Initialization (`__init__`)

- **Inputs**:
  - `policy_model`: The policy network (e.g., `SVRPPolicy`) to be trained.
  - `embedding_dim`: Dimension of the embeddings used in the policy and baseline models.
  - `lr`: Learning rate for the policy network.
  - `baseline_lr`: Learning rate for the baseline model.
  - `entropy_weight`: Weight for entropy regularization (encourages exploration).
  - `device`: Device for tensor operations (CPU or GPU).
- **Components**:
  - **Policy Optimizer**: An Adam optimizer for the policy network.
  - **Baseline Model**: A separate model (`BaselineModel`) to estimate the expected return, used to reduce variance in policy gradients.
  - **Baseline Optimizer**: An Adam optimizer for the baseline model.
  - **Loss Functions**:
    - **MSE Loss**: For training the baseline model.
  - **Tracking Metrics**: Lists to track rewards, policy losses, and baseline losses during training.

2. Training an Episode (`train_episode`)

- Trains the policy network for one episode by interacting with the environment.
- **Inputs**:
  - env: The SVRP environment.
  - `batch_size`: Number of parallel environments.
  - `max_steps`: Maximum number of steps per episode.
- **Steps**:
  1. **Reset Environment**: Initializes the environment and retrieves initial features (customer features, vehicle features, demands).
  2. **Trajectory Storage**: Initializes lists to store log probabilities, rewards, entropies, and baseline values for the episode.
  3. **Episode Loop**:
     - For each step:

        - **Forward Pass**: Passes the current state through the policy network to compute log probabilities for actions.
        - **Action Sampling**: Samples actions for each vehicle based on the log probabilities.
        - **Entropy Calculation**: Computes entropy of the action distribution for regularization.
        - **Baseline Prediction**: Uses the baseline model to estimate the expected return.
        - **Environment Step**: Executes the sampled actions in the environment and retrieves the next state, rewards, and done flags.
        - **Store Trajectory Data**: Saves log probabilities, rewards, entropies, and baseline values for later use.
        - **Check Completion**: Ends the loop if all environments are done.
    4. **Process Trajectories**:
        - Computes returns (discounted sum of future rewards).
        - Calculates advantages (returns - baseline values).
        - Computes policy loss and baseline loss.
    5. **Update Models**:
        - Backpropagates and updates the policy network using the policy loss.
        - Backpropagates and updates the baseline model using the baseline loss.
    6. **Metrics**:
        - Tracks the mean reward, policy loss, and baseline loss for the episode.
- **Outputs**:
    - `mean_reward`: Average reward for the episode.
    - `policy_loss`: Policy loss for the episode.
    - `baseline_loss`: Baseline loss for the episode.

3. Returns Calculation (`_compute_returns`)

- Computes the discounted sum of future rewards (returns) for each step in the episode.
- **Inputs**:
    - `rewards`: Tensor of shape `[episode_length, batch_size]` containing rewards at each step.
    - `mask`: Tensor of shape `[episode_length, batch_size]` indicating active environments.
    - `gamma`: Discount factor for future rewards.
- **Outputs**:
    - `returns`: Tensor of shape `[episode_length, batch_size]` containing the computed returns.

4. **Policy Loss Calculation (`_compute_policy_loss`)**

- Computes the policy loss using the REINFORCE algorithm with a baseline.
- **Inputs**:
  - `log_probs`: Log probabilities of the selected actions.
  - `entropies`: Entropy of the action distributions.
  - `advantages`: Advantages (returns - baseline values).
  - `mask`: Mask to handle variable-length episodes.
- **Outputs**:
  - `policy_loss`: Scalar tensor representing the total policy loss.
- **Details**:
  - The policy gradient is computed as `-log_probs * advantages`.
  - Entropy regularization is added to encourage exploration.

5. **Baseline Loss Calculation (`_compute_baseline_loss`)**

- Computes the baseline loss using Mean Squared Error (MSE) between the baseline predictions and the returns.
- **Inputs**:
  - `baseline_values`: Predicted baseline values.
  - `returns`: Computed returns.
  - `mask`: Mask to handle variable-length episodes.
- **Outputs**:
  - `baseline_loss`: Scalar tensor representing the total baseline loss.

6. **Model Saving and Loading**

- **`save_models`**: Saves the policy and baseline models to disk.
- **`load_models`**: Loads the policy and baseline models from disk.

Summary

The reinforce.py file implements the **REINFORCE algorithm** with a baseline for training the policy network in the SVRP. It:

- Interacts with the environment to collect trajectories.
- Computes policy gradients using the REINFORCE formula.
- Uses a baseline model to reduce variance in the policy gradients.
- Includes entropy regularization to encourage exploration.
- Tracks and updates both the policy and baseline models during training.

This trainer is a key component of the SVRP-RL framework, enabling the policy network to learn optimal routing strategies through reinforcement learning.

baseline.py

The baseline.py file defines the `BaselineModel` class, which is used to estimate the **expected return** of a state in the **REINFORCE algorithm**. This baseline helps reduce the variance of the policy gradient updates by providing a reference value (the expected return) to compare against the actual return.

1. Purpose

- The baseline model predicts the expected return for a given state, which is used in the REINFORCE algorithm to compute the **advantage**: [\text{Advantage} = \text{Return} - \text{Baseline Prediction}]
- By subtracting the baseline prediction, the variance of the policy gradient is reduced, leading to more stable training.

2. Initialization (`__init__`)

- **Inputs**:
  - `embedding_dim`: The dimension of the intermediate embeddings used in the model.
- **Architecture**:
  - A fully connected feedforward neural network with three layers:
    1. **`fc1`**: Linear layer mapping the input features to `embedding_dim`.
    2. **`fc2`**: Linear layer mapping `embedding_dim` to `embedding_dim // 2`.
    3. **`fc3`**: Linear layer mapping `embedding_dim // 2` to a single scalar output (the predicted return).
  - **Activation Function**: ReLU is used after the first two layers for non-linearity.

3. Forward Pass (`forward`)

- **Inputs**:
  - `customer_features`: Tensor of shape `[batch_size, num_nodes, feature_dim]`, representing features for all customers in the batch.
  - `vehicle_features`: Tensor of shape `[batch_size, num_vehicles, feature_dim]`, representing features for all vehicles in the batch.
- **Steps**:
  1. **Average Features**:
     - Computes the mean of the customer features across all nodes (`avg_customer`).
     - Computes the mean of the vehicle features across all vehicles (`avg_vehicle`).
  2. **Concatenate Features**:
     - Concatenates the averaged customer and vehicle features into a single tensor of shape `[batch_size, combined_feature_dim]`.

3. **Pass Through Network**:
    - Processes the concatenated features through the fully connected layers (`fc1`, `fc2`, `fc3`) with ReLU activations after the first two layers.
4. **Output**:
    - Produces a tensor of shape `[batch_size, 1]`, representing the predicted expected return for each state in the batch.

Summary

The `BaselineModel` is a simple feedforward neural network that predicts the expected return for a given state. It:

- Takes customer and vehicle features as input.
- Averages the features to create a compact representation of the state.
- Processes the state representation through fully connected layers to predict the expected return.

This model is trained alongside the policy network in the REINFORCE algorithm to minimize the **mean squared error (MSE)** between the predicted return and the actual return. By providing a baseline for the policy gradient, it helps stabilize and improve the training process.