Explanation of SimpleDNN and Algorithm Classes

English Explanation

SimpleDNN Class

The `SimpleDNN` class is a simple implementation of a two-layer neural network from scratch without using any deep learning frameworks like TensorFlow or PyTorch. Here's a detailed breakdown of its components and functionality:

Initialization (`__init__` method)

- **Weights and Biases Initialization**:
  - `self.weights1` and `self.biases1` are the weights and biases for the first layer (input to hidden).
  - `self.weights2` and `self.biases2` are the weights and biases for the second layer (hidden to output).
  - The weights are initialized randomly with small values (multiplied by 0.01) to break symmetry.
  - The biases are initialized to zero.

Activation Functions

- **ReLU Activation** (`relu` method): Applies the Rectified Linear Unit (ReLU) activation function, which outputs the input directly if it is positive; otherwise, it outputs zero.
- **ReLU Derivative** (`relu_derivative` method): Computes the derivative of the ReLU function, which is used during backpropagation.

Forward Pass (`forward` method)

- Computes the output of the network for a given input `x`.
- **Layer 1**: `z1 = x * weights1 + biases1`, then `a1 = ReLU(z1)`.
- **Layer 2**: `z2 = a1 * weights2 + biases2`.

Backward Pass (`backward` method)

- Performs backpropagation to update the weights and biases based on the error.
- **Output Layer Error**: `dz2 = output - y`, where `output` is the predicted value and `y` is the actual value.
- **Gradient Calculation**:
  - `dw2` and `db2` are gradients for the second layer weights and biases.
  - `dz1` is the error propagated back to the first layer.
  - `dw1` and `db1` are gradients for the first layer weights and biases.
- **Weights and Biases Update**: Updates the weights and biases using the calculated gradients and the learning rate.

Training (`train` method)

- Trains the network using the given input data `x` and labels `y` for a specified number of epochs.
- For each epoch, it performs a forward pass, computes the loss, and performs a backward pass to update the weights and biases.

Prediction (`predict` method)

- Computes the output of the network for a given input `x`.

Algorithm Class

The `Algorithm` class is designed to represent the AI player using a neural network to predict the opponent's card based on the bet. Here's a detailed breakdown of its components and functionality:

Initialization (`__init__` method)

- Initializes the deck of cards and sets the initial points.
- Trains the DNN model by calling the `train_dnn` method.

DNN Training (`train_dnn` method)

- Generates training data (bet amounts and corresponding card values) using the `generate_training_data` method.
- Normalizes the training data.

- Initializes and trains a `SimpleDNN` model using the generated data.
- Returns the trained model.

Training Data Generation (`generate_training_data` method)

- Generates a specified number of training samples.
- Each sample consists of a random card value (1-10) and a corresponding bet amount, which is a product of the card value and a random multiplier (1-10).

Card Picking (`pick` method)

- Randomly picks a card from the deck.

Decision to Give Up (`giveUp` method)

- Uses the DNN model to predict the opponent's card based on the current bet.
- If the predicted card is 2 or lower, the AI decides to give up; otherwise, it continues.

Raise Bet (`raiseBet` method)

- Raises the current bet by a fixed amount (10), ensuring it doesn't exceed the AI's points.

Initial Bet (`bet` method)

- Places an initial bet of a fixed amount (10), ensuring it doesn't exceed the AI's points.

Card Prediction (`predict_card` method)

- Normalizes the bet amount.
- Uses the DNN model to predict the card corresponding to the normalized bet.
- Denormalizes the predicted card value.

Korean Explanation

SimpleDNN 클래스

`SimpleDNN` 클래스는 TensorFlow나 PyTorch와 같은 딥러닝 프레임워크를 사용하지 않고, 기본적인 2층 신경망을 구현한 것입니다. 각 구성 요소와 기능을 자세히 설명하면 다음과 같습니다:

초기화 (`__init__` 메서드)

- **가중치 및 바이어스 초기화**:
  - `self.weights1` 및 `self.biases1`는 첫 번째 층(입력에서 은닉층)의 가중치와 바이어스입니다.
  - `self.weights2` 및 `self.biases2`는 두 번째 층(은닉층에서 출력층)의 가중치와 바이어스입니다.
  - 가중치는 대칭을 깨기 위해 작은 값(0.01로 곱함)으로 무작위 초기화됩니다.
  - 바이어스는 0으로 초기화됩니다.

활성화 함수

- **ReLU 활성화 함수** (`relu` 메서드): ReLU(Rectified Linear Unit) 활성화 함수를 적용하여 입력이 양수이면 입력을 그대로 출력하고, 그렇지 않으면 0을 출력합니다.
- **ReLU 도함수** (`relu_derivative` 메서드): 역전파 중 사용되는 ReLU 함수의 도함수를 계산합니다.

순방향 전파 (`forward` 메서드)

- 주어진 입력 `x`에 대해 네트워크의 출력을 계산합니다.
- **1층**: `z1 = x * weights1 + biases1`, 그런 다음 `a1 = ReLU(z1)`.
- **2층**: `z2 = a1 * weights2 + biases2`.

역방향 전파 (`backward` 메서드)

- 오류를 기반으로 가중치와 바이어스를 업데이트하기 위해 역전파를 수행합니다.
- **출력층 오류**: `dz2 = output - y`, 여기서 `output`은 예측값이고 `y`는 실제 값입니다.
- **기울기 계산**:
  - `dw2` 및 `db2`는 두 번째 층의 가중치와 바이어스에 대한 기울기입니다.

  - dz1은 첫 번째 층으로 전파된 오류입니다.
  - dw1 및 db1은 첫 번째 층의 가중치와 바이어스에 대한 기울기입니다.

- **가중치 및 바이어스 업데이트**: 계산된 기울기와 학습률을 사용하여 가중치와 바이어스를 업데이트합니다.

훈련 (train 메서드)

- 지정된 에포크 수 동안 주어진 입력 데이터 x와 레이블 y를 사용하여 네트워크를 훈련시킵니다.
- 각 에포크마다 순방향 전파를 수행하고, 손실을 계산하며, 역방향 전파를 수행하여 가중치와 바이어스를 업데이트합니다.

예측 (predict 메서드)

- 주어진 입력 x에 대해 네트워크의 출력을 계산합니다.

Algorithm 클래스

Algorithm 클래스는 베팅에 기반하여 상대방의 카드를 예측하기 위해 신경망을 사용하는 AI 플레이어를 나타냅니다. 각 구성 요소와 기능을 자세히 설명하면 다음과 같습니다:

초기화 (__init__ 메서드)

- 카드 덱을 초기화하고 초기 포인트를 설정합니다.
- train_dnn 메서드를 호출하여 DNN 모델을 훈련시킵니다.

DNN 훈련 (train_dnn 메서드)

- generate_training_data 메서드를 사용하여 훈련 데이터를 생성합니다(베팅 금액 및 해당 카드 값).
- 훈련 데이터를 정규화합니다.
- 생성된 데이터를 사용하여 SimpleDNN 모델을 초기화하고 훈련시킵니다.
- 훈련된 모델을 반환합니다.

훈련 데이터 생성 (generate_training_data 메서드)

- 지정된 수의 훈련 샘플을 생성합니다.
- 각 샘플은 무작위 카드 값(1-10)과 해당 베팅 금액으로 구성됩니다. 베팅 금액은 카드 값과 무작위 곱셈 값(1-10)의 곱입니다.

카드 선택 (pick 메서드)

- 덱에서 무작위로 카드를 선택합니다.

포기 결정 (giveUp 메서드)

- 현재 베팅에 따라 DNN 모델을 사용하여 상대방의 카드를 예측합니다.
- 예측된 카드가 2 이하이면 AI는 포기하고, 그렇지 않으면 계속합니다.

베팅 증가 (raiseBet 메서드)

- 현재 베팅을 일정 금액(10)만큼 증가시키며, AI의 포인트를 초과하지 않도록 합니다.

초기 베팅 (bet 메서드)

- 일정 금액(10)으로 초기 베팅을 하며, AI의 포인트를 초과하지 않도록 합니다.

카드 예측 (predict_card 메서드)

- 베팅 금액을 정규화합니다.
- 정규화된 베팅 금액을 기반으로 DNN 모델을 사용하여 카드를 예측합니다.
- 예측된 카드 값을 역정규화합니다.