what does the train() function do?

**Summary**

* train() in neural\_network.py:90 trains a simple policy network to predict an action from a flattened game state using mock data and a legality mask.

**Key Steps**

* Initialize sizes: computes input\_size from a random flattened state and fixes action\_size=43 (neural\_network.py:90).
* Build model: creates PolicyNetwork(input\_size, action\_size) which outputs logits and masks out illegal actions in forward (neural\_network.py:23, neural\_network.py:34).
* Optimizer/loss: sets up Adam (lr=1e-3) and per-sample CrossEntropyLoss (reduction='none') (neural\_network.py:93).
* Data: generates 200 mock samples of (GameState, legal Action) and wraps them in a DataLoader via GameDataset (neural\_network.py:77, neural\_network.py:60, neural\_network.py:98–neural\_network.py:99).
* Training loop: for 10 epochs, does forward pass with the legality mask, computes loss only if the target action is legal (loss\_raw \* mask[range(len(y)), y]), backprops, and updates weights; accumulates and prints epoch loss (neural\_network.py:101–neural\_network.py:112).
* Execution: runs when the script is executed as main (neural\_network.py:114).

Notes

* It doesn’t save or return the trained model.
* action\_size=43 is a hardcoded assumption about the action space size; it must match legal\_actions\_mask and encoding functions.

The aim of this project is to train an ai to play the board game Splendor with the aim of beating human performance.

Based on the current code base, can you code a policy + value network with self play with MTCS to train the network?  
Build only the basic bricks for now, we can refine later.

Use only the parts of the code base that you need. Feel free to ignore the rest, or even suggest deletion

**Overview**

* Added a minimal AlphaZero-style framework: a shared policy+value network, MCTS guided by the network, a self-play loop generating training data, and a simple training function.

**What I Added**

* alpha\_zero.py: end-to-end bricks
  + PolicyValueNet: MLP with policy logits (43 actions) + value in [-1,1].
  + AlphaZeroMCTS: PUCT search using network priors + value.
  + self\_play\_episode(): plays a game via MCTS, returns trajectory and winner.
  + compute\_targets(): builds (X, pi, z) targets from trajectory.
  + train\_on\_batch(): joint policy cross-entropy + value MSE.
  + az\_train(): simple self-play → train loop.

**Key Integrations**

* State encoding and action space:
  + Uses nn\_input\_output.flatten\_game\_state for inputs.
  + Uses nn\_input\_output.legal\_actions\_mask to mask illegal actions.
  + Uses nn\_input\_output.index\_to\_action to map policy indices to Action.
* Game transitions:
  + Uses game\_state.GameState.apply\_action, get\_legal\_actions, and terminal/winner handling.
* Initial states:
  + Uses cards\_init.setup\_game(num\_players=2) for proper Splendor starting states.

**How It Works**

* PolicyValueNet (alpha\_zero.py:16): two-head MLP, policy head outputs 43 logits, value head outputs a scalar via tanh.
* MCTS (alpha\_zero.py:64):
  + Expansion: calls the network; masks illegal actions; stores priors in edges.
  + Selection: uses Q + c\_puct \* P \* sqrt(sumN)/(1+N) over legal edges.
  + Backup: flips sign at each step (2-player, zero-sum assumption).
  + Terminal leaf: assigns leaf value -1 from the current player’s perspective.
  + Produces pi as visit-frequency distribution; temperature can be applied.
* Self-play + Training (alpha\_zero.py:140):
  + Collects (state, pi, player\_to\_move) per step.
  + At game end, computes z = 1 if player\_to\_move == winner, else -1.
  + Trains: loss = CE(policy, pi) + MSE(value, z).

**Usage**

* Quick smoke-test: run python alpha\_zero.py (uses tiny settings).
* Programmatic training:
  + alpha\_zero.py:187: az\_train(iterations=5, games\_per\_iter=4, mcts\_simulations=64, lr=1e-3, device="cpu")
  + Returns a trained PolicyValueNet. You can then use AlphaZeroMCTS(model).run(state) to pick moves.

**Notes**

* Action space: fixed 43 actions mirrored from your nn\_input\_output.py. Some complex “return tokens” actions are skipped by the existing mask/encoding; OK for a baseline, we can extend later.
* Value handling at terminal: assumes 2-player zero-sum; draws are set to 0 if encountered (none defined currently).
* Training is intentionally minimal (single big batch per iteration, no replay buffer). We can add replay, checkpointing, and evaluation against baselines in the next step.

**Next Steps**

* Add replay buffer and multi-epoch mini-batch training per iteration.
* Save/load checkpoints; add temperature schedule (high early moves, low late-game).
* Expand action encoding to cover “return tokens” cases for strict rules coverage.
* Add evaluation: pit current net vs. random or prior net.
* Optional: GPU support toggle and batch-evaluated MCTS for speed.