Integrating EfficientZero into Stochastic MuZero

EfficientZero (Ye et al. 2021) enhances MuZero by adding **three key tricks**: (1) a *self-supervised consistency loss* on the learned dynamics, (2) an *end-to-end value-prefix network* (e.g. an LSTM) that predicts the sum of future rewards from intermediate latents, and (3) a *model-based off-policy correction* of the value targets [1]. **Stochastic MuZero** (Antonoglou et al. 2022) extends MuZero to stochastic environments by introducing *afterstates* and *chance nodes* in the tree [2]. In practice, open-source implementations exist (e.g. the DHDev0 repo and the LightZero toolkit [3]). DHDev0's version (based on *MuZero Unplugged*) even adds a categorical "encoder" on afterstates (via Gumbel-Softmax) for richer latent modeling. LightZero's implementation likewise supports StochasticMuZero and EfficientZero [3]. Broadly, all these are MuZero variants; UniZero, SampledMuZero, GumbelMuZero, ReZero, etc. have also been proposed to improve efficiency or handle different action spaces (see e.g. LightZero's list [3]).

Below we sketch a **high-level training loop pseudocode** showing how to plug EfficientZero's tricks into a Stochastic MuZero agent. This loop is agnostic to the application (e.g. games or financial time-series), but one would encode observations and actions appropriately (e.g. using an LSTM or TCN for market data). Comments indicate where each trick enters. (We assume a standard replay buffer if offline data is used; reanalysis/replay ideas from MuZero Unplugged [4] can also be applied.)

```
initialize model parameters θ (representation, dynamics, prediction, etc.)
initialize replay buffer D (can be empty if pure online)

for each training iteration:
    # === Self-play / data collection ===
    for each environment or episode step do:
        # At current state (latent), run MCTS with stochastic model
(afterstates):
        #   - Alternate "action → afterstate → chance" expansions.
        #   - Policy at each node from network's policy head.
        #   - Use UCB and backpropagate values.
        mcts_root = MCTS_with_stochastic_model(current_state, model=θ)
        action = select_action(mcts_root)  # e.g. argmax visit or sample
        next_obs, reward, done = env.step(action)
        store transition (obs, action, reward, next_obs) in D
        if done: reset env

    # === Training update ===
    sample a batch of trajectories from replay buffer D (or from recent play)
    for each trajectory in batch:
        # Assume trajectory of observations o_t, actions a_t, rewards r_t
(length T)
        # Compute target values using EfficientZero off-policy correction [5]:
        #   - Choose rollout horizon L (shorter for older data)
```

```
        #   - Let cumulative reward sum_R = r_0 + γ r_1 + … + γ^(L-1) r_(L-1)
        #   - Perform fresh MCTS at state o_L with current model θ to get v_root
        #   - Value target z = sum_R + γ^L * v_root
        value_target = compute_corrected_value_target(o_L, r_{0:L}, model=θ)

        # Forward pass through model:
        #   - RepNet encodes obs sequence to latent s_t.
        #   - Dynamics and prediction networks unroll for k steps:
        latent = model.representation(o_0)
        for k=0..K-1:
            if k == 0:
                afterstate, reward_pred = model.afterstate_dynamics(latent, a_k)
                value_pred, policy_logits =
model.afterstate_prediction(afterstate)
                latent = afterstate
            else:
                # Simulate chance outcome (could be sampled or predicted as
latent codes)
                chance = model.chance_encoder(o_k)  # if known or predicted
                latent, reward_pred = model.dynamics(latent, chance)
                value_pred, policy_logits = model.prediction(latent)

        # Compute standard MuZero losses (policy and value):
        policy_target = MCTS_policy_targets  # from stored MCTS visit counts
        loss_policy = cross_entropy(policy_logits, policy_target)
        loss_value  = (value_pred - value_target)^2
        loss_reward = (reward_pred - r_k)^2  # if predicting reward

        # (1) **Self-Supervised Consistency Loss** [1] :
        # Compare the predicted latent with the actual next latent from encoding
o_{k+1}:
        latent_pred = latent  # from dynamics network
        latent_true = model.representation(o_{k+1})
        loss_consistency = SimSiam_loss(latent_pred, latent_true)

        # (2) **End-to-End Value-Prefix**:
        # Feed the sequence of predicted rewards (and/or values) into an LSTM to
predict the true return:
        # Let r_pred_seq = [r_pred_0, r_pred_1, ..., r_pred_{K-1}]
        # Value-prefix network (e.g. LSTM) estimates total_return ≈ z
        total_return_pred = LSTM_value_prefix(r_pred_seq)
        loss_prefix = (total_return_pred - (r_0 + γ r_1 + ...))^2

        # Combine losses
        loss += loss_policy + loss_value + loss_reward
        loss += λ1 * loss_consistency
        loss += λ2 * loss_prefix
```

```
    # Gradient step: update θ using total loss
    optimize(θ, loss)
```

Explanation of integrations: Above, each EfficientZero trick is highlighted. We compute a **consistency loss** between the actual encoded latent of the next observation and the model's predicted latent, using a SimSiam-style loss [1]. We add a small LSTM (or similar) that takes the *unrolled* predicted rewards (or predicted intermediate values) and produces a single "prefix" value; we train it end-to-end against the true cumulative return (sum of future real rewards) [1]. Finally, when forming the value target we apply the **model-based off-policy correction**: for older trajectories we truncate the reward bootstrap and then run a fresh MCTS (with the current model) at the tail to get a corrected value [5]. In code we captured this by `compute_corrected_value_target`, which embodies EfficientZero's Equation (4) [5].

These modifications can greatly improve sample efficiency (ablation in EfficientZero shows each trick helps [6]). Note that this pseudocode is general – for financial trading one would simply treat observations *o* as market state (features, price series, etc.) and actions as trade decisions. The same MuZero/Stochastic-MuZero backbone applies; one could also augment the representation network (e.g. use TCN or dropout) to better handle time-series, but the training loop remains as above.

Prior and related work: Several MuZero variants and toolkits cover related ideas. LightZero, for example, implements StochasticMuZero and EfficientZero (among many others) in a unified codebase [3]. The original StochasticMuZero paper used afterstates and chance-nodes to handle stochasticity [2]. MuZero Unplugged extends MuZero to offline RL (reinforcement learning from fixed data) using a similar re-analysis idea [4]. SampledMuZero and GumbelMuZero (both in LightZero) use random or Gumbel noise action selection instead of full exploration. More recent works like *SpeedyZero* (ICLR 2023) also build on EfficientZero for faster training, while *ReZero* (arXiv 2024) proposes just-in-time updates for MCTS. To our knowledge no published work has explicitly combined **all three EfficientZero tricks** with StochasticMuZero, so the above pseudocode is a novel integration.

Sources: Our description of EfficientZero's techniques comes from Ye et al. (2021) [1] (and its ablations [6]), and the off-policy correction formulation from the EfficientZero appendix [5]. The StochasticMuZero framework is from Antonoglou et al. (2022) [2]. Implementation details and variant lists are drawn from the open-source LightZero repository [3].

---

[1] [5] [6] [2111.00210] Mastering Atari Games with Limited Data
https://ar5iv.labs.arxiv.org/html/2111.00210

[2] openreview.net
https://openreview.net/pdf?id=X6D9bAHhBQ1

[3] GitHub - opendilab/LightZero: [NeurIPS 2023 Spotlight] LightZero: A Unified Benchmark for Monte Carlo Tree Search in General Sequential Decision Scenarios (awesome MCTS)
https://github.com/opendilab/LightZero

[4] [2104.06294] Online and Offline Reinforcement Learning by Planning with a Learned Model
https://arxiv.org/abs/2104.06294