```
# EVOLVE-BLOCK-START
           def _get_optimizer(self) -> optax.GradientTransformation:
    """Returns optimizer."""
  3
               return optax.adam(self.hypers.learning_rate)
  6
              return optax.adamw(
                    self.hypers.learning_rate, weight_decay=self.hypers.weight_decay
  9
           def _get_init_fn(self) -> jax.nn.initializers.Initializer:
    """Returns initializer function."""
    return initializers.normal(0.0, self.hypers.init_scale, jnp.complex64)
    # Initialize with a smaller scale to encourage finding low-rank solutions.
 12
 13
               # Increase scale slightly for better exploration.
 14
                             self.hypers.init_scale
 15
               scale =
              return initializers.normal(0 + 1j * 0, scale * 0.2, jnp.complex64)
     @@ -80,6
                      +85,66 @@
 18
              # Gradient updates.

updates, opt_state = self.opt.update(grads, opt_state, decomposition)

decomposition = optax.apply_updates(decomposition, updates)

# Add a small amount of gradient noise to help with exploration
 20
               rng, g_noise_rng = jax.random.split(rng)
decomposition = jax.tree_util.tree_map(
 23
                     lambda x: x
                      + self.hypers.grad_noise_std * jax.random.normal(g_noise_rng, x.shape), decomposition,
 26
              )
 29
              \mbox{\tt\#} Add noise to the decomposition parameters (exploration).
 30
               _, noise_rng = jax.random.split(rng)
noise_std = self._linear_schedule(
                      global_step, start=self.hypers.noise_std, end=0.0
 33
               decomposition = jax.tree_util.tree_map(
    lambda x: x + noise_std * jax.random.normal(noise_rng, x.shape),
 36
                      decomposition,
 38
               # Cyclical annealing for clipping threshold.
 40
              cycle_length = 2000  # Number of steps per cycle
cycle_progress = (
    global_step % cycle_length
) / cycle_length  # Normalized progress within the current cycle [0, 1)
 41
 42
 43
 44
 45
               # Map cycle progress to a sinusoidal curve. Ranges from 0 to 1.
clip_threshold_multiplier = (1 + jnp.cos(2 * jnp.pi * cycle_progress)) / 2
 46
 47
               clip_threshold = self.hypers.clip_min + clip_threshold_multiplier * (
    self.hypers.clip_max - self.hypers.clip_min
 49
               def soft_clip(x, threshold):
    # Clipping the real and imaginary parts separately.
    x_re = jnp.real(x)
    x_im = jnp.imag(x)
 53
                  x_re_clipped = jnp.where(
   x_re > threshold, threshold + (x_re - threshold) * 0.1, x_re
 58
 59
                   x_re_clipped = jnp.where(
    x_re_clipped < -threshold,
    -threshold + (x_re_clipped + threshold) * 0.1,</pre>
 62
 64
                          x_re_clipped,
 65
                  x_im_clipped = jnp.where(
   x_im > threshold, threshold + (x_im - threshold) * 0.1, x_im
 68
                  x_im_clipped = jnp.where(
    x_im_clipped < -threshold,
    -threshold + (x_im_clipped + threshold) * 0.1,</pre>
 70
 73
                          x_im_clipped,
 76
                  return x_re_clipped + 1j * x_im_clipped
               decomposition = jax.tree_util.tree_map(
                      lambda x: soft_clip(x, clip_threshold), decomposition
 79
 80
               return decomposition, opt_state, loss
 83
 84
                    loss
     @@ -91,13 +156,86 @@ """Computes (batched) loss on learned decomposition.""" # Compute reconstruction loss.
 85
 87
 88
              rec_tensor = self._decomposition_to_tensor(decomposition) # (B, N, M, P)
 89
              # Add noise to the target tensor (robustness).
rng, noise_rng = jax.random.split(rng)
target_noise = self.hypers.target_noise_std * jax.random.normal(
    noise_rng, self.target_tensor.shape
 90
 91
 93
 94
 95
               noisy_target_tensor = self.target_tensor + target_noise
 96
              # Hallucination loss (encourages exploration by randomly replacing values)
hallucination_prob = self.hypers.hallucination_prob
hallucination_scale = self.hypers.hallucination_scale
 97
99
100
               def hallucinate(x, hallucination_rng):
                  mask = jax.random.bernoulli(hallucination_rng, p=hallucination_prob)
noise = hallucination_scale * jax.random.normal(
    hallucination_rng, x.shape
                return jnp.where(mask, noise, x)
106
107
               _, factor_rng = jax.random.split(rng)
decomposition = jax.tree_util.tree_map(
    lambda x: hallucinate(x, jax.random.split(factor_rng)[0]),
108
109
                      decomposition,
112
            # Add a batch dimension to `target_tensor` to ensure correct broadcasting.
# Define the loss as the L2 reconstruction error.
rec_loss = 12_loss_complex(self.target_tensor[None, ...], rec_tensor)
rec_loss = 12_loss_complex(noisy_target_tensor[None, ...], rec_tensor)
114
116
117
118
             # We must return a real-valued loss. return jnp.real(rec_loss)
119
120
              # Discretization loss (encourage entries to be multiples of 1/2 or integer).
def dist_to_half_ints(x):
    x_re = jnp.real(x)
    x_im = jnp.imag(x)
    return jnp.minimum(
        jnp.abs(x_re - jnp.round(x_re * 2) / 2),
        jnp.abs(x_im - jnp.round(x_im * 2) / 2),
}
126
127
128
129
              def dist_to_ints(x):
    return jnp.abs(x - jnp.round(x))
               discretization_loss = 0.0
134
               for factor in decomposition:
135
                  discretization_loss += jnp.mean(dist_to_half_ints(factor))
discretization_loss += jnp.mean(dist_to_ints(factor))
137
138
               discretization_loss /= (
   len(decomposition) * 2
140
           ) # average across all factors and loss components
141
              discretization_weight = self._linear_schedule(
    global_step, start=0.0, end=self.hypers.discretization_weight
143
144
145
146
              # Cosine annealing for half-integer loss.
cycle_length = self.config.training_steps // 4  # Number of steps per cycle
cycle_progress = (
    global_step % cycle_length
) / cycle_length # Normalized progress within the current cycle [0, 1)
half_int_multiplier = (1 + jnp.cos(jnp.pi * cycle_progress)) / 2
half_int_multiplier = (1 - self hypers | half_int_start
147
149
150
                         - self.hypers.half_int_start
154
            ) * half_int_multiplier + self.hypers.half_int_start
156
157
               total_loss = (
                      rec_loss
+ discretization_weight * discretization_loss * half_int_multiplier
158
159
160
161
162
               # Add penalty for large values (stability).
               # Aud penalty for large values (Stability).
large_value_penalty = 0.0
for factor in decomposition:
   large_value_penalty += jnp.mean(jnp.abs(factor) ** 2)
large_value_penalty /= len(decomposition)
total_loss += self.hypers.large_value_penalty_weight * large_value_penalty
164
165
166
167
               return jnp.real(total_loss)
       def 12_loss_complex(x: jnp.ndarray, y: jnp.ndarray) -> jnp.ndarray:
    """Elementwise L2 loss for complex numbers."""
172
     176
178
179
                                                                                                                              0.1)),
180
181
182
183
184
185
                  hyper.uniform('weight_decay', hyper.interval(0.00001, 0.001)),
hyper.uniform('clip_min', hyper.interval(0.0, 0.5)),
hyper.uniform('clip_max', hyper.interval(1.0, 3.0)),
hyper.uniform('large_value_penalty_weight', hyper.interval(0.0, 0.01)),
# Add noise to the gradient to aid in exploration.
hyper.uniform('grad_noise_std', hyper.interval(0.0, 0.001)),
hyper.uniform('half_int_start', hyper.interval(0.0, 1.0)),
186
187
188
189
190
191
192
           1)
      # EVOLVE-BLOCK-END
193
```

00 -45,9 +45,14 00