Text Generation via Various Sampling Methods

Avdhoot Golekar (23B0060), Geet Sethi (23B2258), Panav Shah (23B3323)

October 22, 2025

1 Task 0: Basic Sampling Methods

We implemented three basic sampling methods: greedy decoding, temperature sampling, and top-k sampling. Each of them is pretty straightforward to implement.

1.1 Implementation

The core logic is give below where we replace the `selection-algo` with the actual selection algorithm i.e. argmax for greedy, modified softmax for temperature, and a top-k selection for top-k.

Algorithm 1 Basic Algorithm

```
Input: tokenizer, model, prefix, max_new, eos_id
input_ids ← tokenizer.encode(prefix)
input_ids_size ← len(input_ids[0])
for i = 1 to max_new do
    logits ← model(input_ids).logits[0, -1, :]
    next_token_id ← selection-algo(logits)
    if next_token_id == eos_id then
        break
    end if
    input_ids ← concat(input_ids, [next_token_id])
end for
return tokenizer.decode(input_ids[0][input_ids_size:])
```

2 Task 1: Sequential Importance Sampling

2.1 Implementation

We use the same basic algorithm as in Task 0 with the top-k algorithm initially. Then we compute the reward for each sequence and weight them by $\exp(\beta \cdot \text{reward})$.

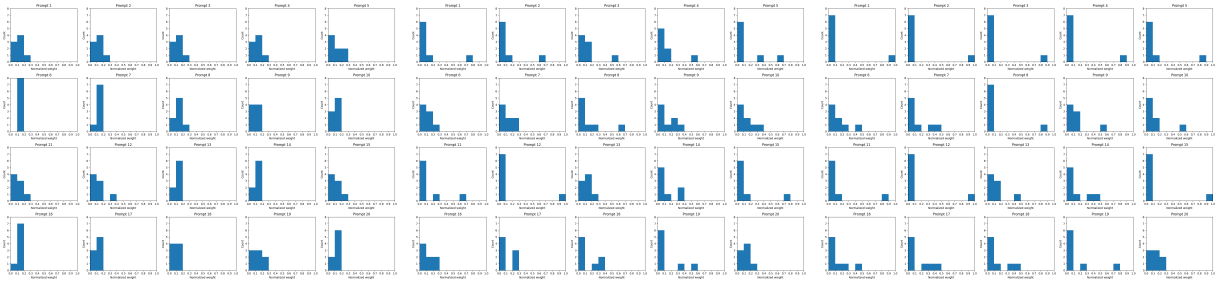
Algorithm 2 Sequential Importance Sampling

Input: tokenizer, model, reward_calc, prefix, K, max_new_tokens, eos_id, beta, k
gen_ids ← batched_topk_decode_ids(tokenizer, model, prefix, max_new_tokens, k, K, eos_id)
samples ← []
weights ← []
for $i = 1$ to K **do**
    reward ← get_total_reward(reward_calc, tokenizer, gen_ids[i])
    weight ← exp(beta × reward)
    samples.append({"text": decode(gen_ids[i]), "weight": weight})
    weights.append(weight)
end for
total_weights ← sum(weights)
normalized_weights ← [w / total_weights for w in weights]
return {"samples": samples, "normalized_weights": normalized_weights}

Note that we use a batched approach to generate the sequences in parallel to improve performance since GPUs are great at parallel processing. This `batched_topk_decode_ids()` function is pretty much the same as the basic algorithm with the top-k selection algorithm slightly modified.

To calculate the reward, we use the `get_total_reward()` function which computes the cumulative reward for a sequence using trigram-based scoring from the `FastRewardCalculator` class which uses the trigram probabilities given to us.

2.2 Weight Distribution Plots



(a) IS ($\beta = 1.0$) (b) IS ($\beta = 5.0$) (c) IS ($\beta = 10.0$)

Figure 1: Weight distribution histograms for different β values

We can see that the distribution is concentrated around 0 since most of the sequences have very small weights and only 1-2 have significant weights.

3 Task 2: Sequential Monte Carlo

The implementation of SMC is also pretty similar to that of the IS with the only difference being that we compute the incremental reward at each step and resample the sequences based on their weights to prevent weight degeneracy.

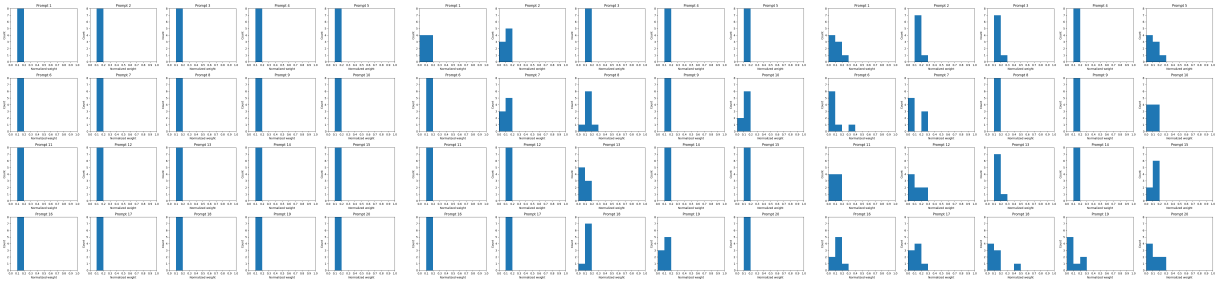
3.1 Implementation

Note that this is implemented in a batched manner just like the IS algorithm which makes the code slightly more complicated but also more efficient!

Algorithm 3 Sequential Monte Carlo

Input: tokenizer, model, reward_calc, prefix, N, max_new_tokens, eos_id, beta, k
input_ids ← tokenizer.encode(prefix).repeat(N, 1)
completed_ids ← zeros(N)
for $t = 1$ to max_new_tokens **do**
  logits ← model(input_ids).logits[:, -1, :]
  top_k_logits, top_k_indices ← topk(logits, k, dim=-1)
  probs ← softmax(top_k_logits, dim=-1)
  next_token_ids ← []
  weights ← []
  for $i = 1$ to N **do**
    if not completed_ids[i] **then**
      sampled_idx ← multinomial(probs[i], 1)
      next_token_id ← top_k_indices[i, sampled_idx]
      ΔR_t ← get_total_reward(input_ids[i]) - get_total_reward(input_ids[i, :-1])
      weight ← exp(beta × ΔR_t)
    else
      next_token_id ← eos_id
      weight ← 1.0
    end if
    next_token_ids.append(next_token_id)
    weights.append(weight)
  end for
  input_ids ← concat(input_ids, next_token_ids.unsqueeze(-1), dim=-1)
  normalized_weights ← weights / sum(weights)
  resampled_indices ← multinomial(normalized_weights, N, replacement=True)
  input_ids ← input_ids[resampled_indices]
  completed_ids ← completed_ids.masked_fill(next_token_ids == eos_id, 1)
  if completed_ids.all() **then**
    break
  end if
end for
return {"samples": samples, "normalized_weights": normalized_weights}

3.2 Weight Distribution Plots



(a) SMC ($\beta = 1.0$) (b) SMC ($\beta = 5.0$) (c) SMC ($\beta = 10.0$)

Figure 2: Weight distribution histograms for different β values

The distribution starts to become slightly more diffused compared to the IS distribution as we increase the β value spreading out but still most of the weights are small. This tells us that our SMC algorithm is working and is able to generate sequences with higher rewards.

4 Task 3: Twisted Sequential Monte Carlo

The TSMC algorithm's implementation is also the same as that of SMC with the only difference being that we compute the twisted reward at each step and use it to calculate the weight instead of a simple incremental reward.

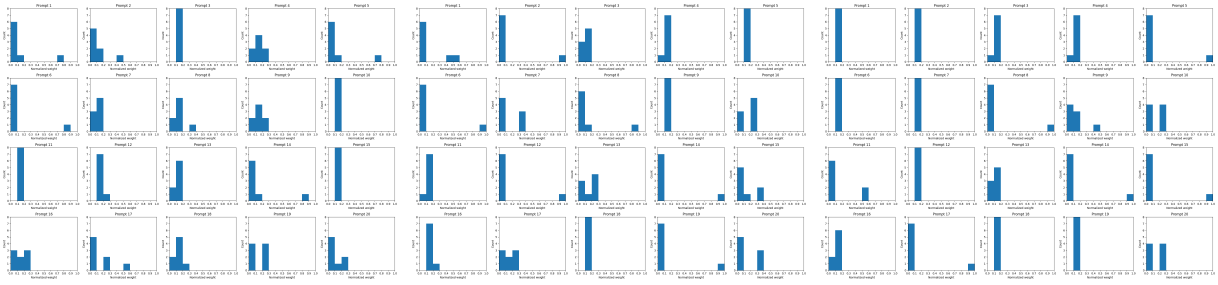
4.1 Implementation

Note that the twisted reward uses bigram probabilities to estimate the expected future reward when only a single token is generated but since these were not directly provided to us, we generate them using the unigram and bigram counts given to us. We also generate the expected rewards for the next token using the 1st token and the 1st two tokens before hand to avoid recomputing them at each step. The code for this can be found in `utils.py`. (These files are generated once and then cached for future use.)

Algorithm 4 Twisted Sequential Monte Carlo

Input: tokenizer, model, reward_calc, prefix, N, max_new_tokens, eos_id, beta, k
input_ids ← tokenizer.encode(prefix).repeat(N, 1)
completed_ids ← zeros(N)
total_probabilities ← ones(N)
for $t = 1$ to max_new_tokens **do**
  logits ← model(input_ids).logits[:, -1, :]
  top_k_logits, top_k_indices ← topk(logits, k, dim=-1)
  probs ← softmax(top_k_logits, dim=-1)
  next_token_ids ← []
  weights ← []
  for $i = 1$ to N **do**
    if not completed_ids[i] **then**
      sampled_idx ← multinomial(probs[i], 1)
      next_token_id ← top_k_indices[i, sampled_idx]
      next_token_prob ← top_k_logits[i, sampled_idx]
      total_probabilities[i] ← total_probabilities[i] × next_token_prob
      updated_input_ids ← concat(input_ids[i], [next_token_id])
      if $t <$ max_new_tokens - 1 **then**
        $\pi_t \leftarrow$ exp(beta × get_twisted_reward(updated_input_ids))
      else
        $\pi_t \leftarrow$ total_probabilities[i] × exp(beta × get_total_reward(updated_input_ids))
      end if
      $\pi_{t-1} \leftarrow$ exp(beta × get_twisted_reward(input_ids[i]))
      weight ← π_t / (π_{t-1}× next_token_prob)
    else
      next_token_id ← eos_id
      weight ← 1.0
    end if
    next_token_ids.append(next_token_id)
    weights.append(weight)
  end for
  input_ids ← concat(input_ids, next_token_ids.unsqueeze(-1), dim=-1)
  normalized_weights ← weights / sum(weights)
  resampled_indices ← multinomial(normalized_weights, N, replacement=True)
  input_ids ← input_ids[resampled_indices]
  total_probabilities ← total_probabilities[resampled_indices]
  completed_ids ← completed_ids.masked_fill(next_token_ids == eos_id, 1)
  if completed_ids.all() **then**
    break
  end if
end for
return {"samples": samples, "normalized_weights": normalized_weights}

4.2 Weight Distribution Plots



(a) TSMC ($\beta = 1.0$) (b) TSMC ($\beta = 5.0$) (c) TSMC ($\beta = 10.0$)

Figure 3: Weight distribution histograms for different β values

The distribution becomes more diffused than the SMC distribution with some weights also getting quite large though most weights are still small. This tells us that our twisted reward is working but it is not perfect else we would have seen a pretty uniform distribution.

5 Task 4: Bonus Task

We take inspiration from the paper by A. Karan et al. [1] where they propose a method to improve the reasoning capabilities of the model by sampling from the power distribution of the model's output instead of training a separate reasoning model. To implement this, they use Metropolis-Hastings sampling to sample from the power distribution since sampling directly from the power distribution is not easy.

We use a similar algorithm as our base but incorporate our reward function to guide the sampling process in our favour. The code for this can be found in `generate_task4_power.py`.

However, since the power distribution makes the output more peaked (similar to temperature sampling), it turns out to not be very effective for our task and the results are not very good.

6 Experimental Results

Table 1: Experimental Results Summary

Method	Expected Reward	Perplexity	Entropy	Avg Length
Task 0: Basic Sampling Methods				
Greedy	2.793	3.365	5.107	49.0
Temp-0.5	2.735	4.103	5.666	49.0
Temp-0.9	2.592	6.672	5.901	49.0
TopK-5	2.769	5.306	5.774	49.0
TopK-10	2.734	6.280	5.851	49.0
Task 1: Sequential Importance Sampling				
IS[$\beta = 1.0$]	2.899	6.439	5.848	49.0
IS[$\beta = 5.0$]	3.462	7.296	5.848	49.0
IS[$\beta = 10.0$]	3.714	7.633	5.848	49.0
Task 2: Sequential Monte Carlo				
SMC[$\beta = 1.0$]	2.745	6.000	5.509	49.0
SMC[$\beta = 5.0$]	3.170	6.652	5.253	49.0
SMC[$\beta = 10.0$]	3.585	6.428	5.286	49.0
Task 3: Twisted Sequential Monte Carlo				
TSMC[$\beta = 1.0$]	3.148	5.495	5.149	49.0
TSMC[$\beta = 5.0$]	3.846	6.964	4.819	49.0
TSMC[$\beta = 10.0$]	3.077	9.248	4.855	49.0
Task 4: Bonus Task				
PowerMCMC[$\alpha = 4.0; steps = 10; k = 10$]	2.340	3.513	5.774	49.0

6.1 Key Findings

- **Expected Reward**: As expected, the expected reward increases as we improve our sampling methods from basic ones to importance sampling to SMC to TSMC.
- **Perplexity**: The perplexity also shows a similar trend as the expected reward though there is a slight decrease in the perplexity values for the TSMC & SMC methods compared to the IS method.
- **Entropy**: The entropy, as expected, is lower for the TSMC & SMC methods compared to the IS method since we resample the sequences based on their weights leading to similar sequences.
- **Average Length**: The average length is constant for all the methods since we don't let the model generate more than 50 tokens and the models don't stop generating new tokens either since we are asking it a pretty open ended question i.e. continuing a story.

6.2 Conclusions

1. **Reward Optimization**: Advanced methods (IS, SMC, TSMC) significantly outperform basic sampling methods in expected reward, with TSMC achieving the highest peak perfor-

mance.

2. **Coherence Trade-off**: Higher reward optimization generally comes at the cost of grammatical coherence, as evidenced by increasing perplexity which is expected since we are using a corpus which has a lot of complicated grammar.

7 References

References

[1] Karan, A. et al. (2025). Reasoning with Sampling: Your Base Model is Smarter Than You Think. arXiv:2510.14901v1.