MCMC in Text Generation

Shlok Mishra

Jaunuary 2024

1 Introduction

- **Language Model**: A pre-trained language model is a computational model that has been previously trained on a large corpus of text data. This training process involves the model learning various aspects of a language, such as its syntax, semantics, and context, without being explicitly programmed with the rules of the language. The goal is to enable the model to understand and generate human-like text, or to perform a variety of language-related tasks.

- **Word Embeddings**: In NLP, this is a term used for the representation of words for text analysis, typically in the form of a real-valued vector that encodes the meaning of the word such that the words that are closer in the vector space are expected to be similar in meaning.

- **Natural text Generation**: Neural text generation typically involves two stages:

  - modeling a distribution over text sequences
  - using a decoding algorithm to generate sequences with the model

  Let $\boldsymbol{y} = (y_1, \ldots, y_T)$ denote a discrete sequence where each y_t is a token from a vocabulary $\mathcal{V}$. Common neural language models factorize the probability of a sequence into the product of per-token conditionals in left-to-right order, $p_\theta = \prod_{t=1}^{T} p_\theta(y_t|\boldsymbol{y}_{<t})$, with each conditional parameterized by a shared neural network, such as transformers. Popular decoding algorithms produce text sequences $\boldsymbol{y}$ using the model p_θ, often conditioned on prompt $\boldsymbol{x}$

- **Constrained Text Generation**: We view text generation as the problem of finding a sequence that satisfies a collection of constraints. For instance, the scenario above amounts to generating a sequence $\boldsymbol{y} = (y_1, \ldots, y_T)$ subject to a soft constraint that the continuation $\boldsymbol{y}$ should be fluent and logically coherent with the prompt $\boldsymbol{x}$. Other constrained generation problems impose additional constraints, such as text infilling where coherence constraints move beyond a left-hand prefix, lexically constrained generation in which hard constraints require the output to contain given tokens, and various forms of semantically-constrained generation in which the output is softly constrained to be similar to another sequence. Since common decoding algorithms generate text monotonically, relying on $p_\theta(y_t|\boldsymbol{y}_{<t})$ for determining the next token, it is challenging to enforce these diverse constraints.

1.1 Word Embeddings

Word embeddings are a numerical representation technique for words, utilized in the domain of natural language processing. The core concept resembles multi-dimensional scaling, where words are represented as points within a high-dimensional vector space. The spatial arrangement in this space is determined through the analysis of a large text corpus, ensuring that semantic relationships and contextual similarities among words are effectively captured.

Mathematically, if we denote the vocabulary by V and the embedding space by $\mathbb{R}^d$ where d is the dimensionality of the embeddings (commonly in the hundreds), each word $w \in V$ is associated with a vector $\mathbf{v}_w \in \mathbb{R}^d$. The goal of the embedding process is to learn this mapping $f : w \rightarrow \mathbf{v}_w$ based on contextual usage patterns in the text.

The essence of word embeddings can also be viewed as a dimensionality reduction technique. It transforms the sparse, high-dimensional space of text data into a denser, continuous vector space. This transformation is not merely based on variance, as in principal component analysis (PCA), but rather optimized to preserve linguistic characteristics.

In terms of mathematical formulation, the similarity between words is quantitatively assessed using distance metrics in the embedding space, such as the cosine similarity. For two words w_1 and w_2, their similarity is given by

$$\text{similarity}(\mathbf{v}_{w_1}, \mathbf{v}_{w_2}) = \frac{\mathbf{v}_{w_1} \cdot \mathbf{v}_{w_2}}{\|\mathbf{v}_{w_1}\| \|\mathbf{v}_{w_2}\|},$$

where closer vectors indicate higher semantic similarity.

The process of learning these embeddings is akin to solving a statistical estimation problem, where the objective is to adjust the vectors such that they can predict the presence of words in a given context. This is often achieved through optimization techniques, minimizing a loss function that captures prediction errors across the corpus.

Applications of word embeddings are vast, extending to tasks such as clustering semantically similar words, solving analogies, or serving as input features for various predictive models. By transforming words into a continuous numerical form, embeddings facilitate the application of statistical analysis and machine learning algorithms to textual data, enabling a deeper understanding of language semantics.

2 Methodology

[1,] Pretrained language models have demonstrated impressive abilities in generating fluent texts by factorizing a string-values distribution $p_{LM}(\boldsymbol{x})(\boldsymbol{w} \in \sum^*)$ over some vocabulory $\sum$ with local normalization:

$$p_{LM}(\boldsymbol{w}) = p_{LM}(EOS|\boldsymbol{w}) \prod_{n=1}^{N} p_{LM}(w_n|\boldsymbol{w}_{<n}) \tag{1}$$

where EOS is a termination symbol, to define a random variable whose value can be a string, i.e., an element of $\sum^*$, or an infinite sequence.

$$\pi(\boldsymbol{w}) = \frac{1}{Z} exp(-U(\boldsymbol{w})) \tag{2}$$

where $U(\boldsymbol{w})$is called the energy function. The flexibility of this framework lies in the fact that one can refine an existing model by coupling its energy function with arbitrary functions that express the desired attributes of the output text.

$$U(\boldsymbol{w}) = U_{LM}(\boldsymbol{w}) + \sum_{i=1}^{I} U_i(\boldsymbol{w}) \tag{3}$$

where $U_{LM}(\boldsymbol{w}) \overset{\text{def}}{=} -logp_{LM}(\boldsymbol{w})$ and each $U_i(\boldsymbol{w})$ measures the extent to which the sequence $\boldsymbol{w}$ satisfies the $i^t h$ constraint. This energy function yields a distribution that is related to $p_{LM}(\boldsymbol{w})$ but places more probability mass on the sequences that better satisfy the constraints.

2.1 Problems in Sampling

Consider sampling a sequence of N words $\boldsymbol{w} = w_1 \cdots w_N \in \sum^N$ from the EBM defined by equations (2) and (3). The normalization constant Z is then an intractable sum of $|\sum|^N$ terms. Similarly, the locally normalized conditional probabilities needed for left-to-right autoregressive sampling—which are effectively ratios of normalization constants—are also intractable.
For MCMC, in our situation, the combinitorially large underlying state space $\sum^N$ means that naive MCMC algorithms would have near-zero acceptance rate. Gibbs sampling is also impractical since evaluating $\pi(\cdot|\boldsymbol{w}_{\backslash n})$ in a locally normalized LM would again involve a summation of $|\sum|^N$ terms.

2.2 Gradient-based Sampling

U_{LM} obtained from a pretrained neural LM is differnetiable, as well as possibly the constraint functions U_i. This allows us to use gradient-based MCMC algorithms, such as Langevin dynamics, to sample from the EBM.
However, problems arise when gradient-based sampling algorithms only directly

apply to continuous distributions. Therefore, to apply such algorithms to sample from discrete distributions, prior works that developed gradient-based sampling for energybased text generation all focus on continuous relaxations of the underlying discrete space

2.3 Faithfulness of Gradient-based Text Samplers

Consider the setting of sampling a sequence of N words $\boldsymbol{w} = w_1 \cdots w_N \in \sum^N$ from an energy-based sequence model. We denote corresponding word embeddings $\boldsymbol{x} = (\boldsymbol{x}_1, \cdots, \boldsymbol{x}_N) \in \mathcal{X} \overset{\text{def}}{=} \mathcal{V}^N \subset \mathbb{R}^{Nd}$ where $\mathcal{V} \subset \mathbb{R}^d$ is the disrecrete set of word embeddings.

3 Langevin MCMC

The core of the Langevin MCMC is the Langevin equation, which is a stochastic differential equation that describes the evolution of a particle in a potential field with some form of friction or resistance, and subject to random fluctuations. In the context of MCMC, the Langevin equation can be discretized to update the chain's state as follows:

$$x_{t+1} = x_t + \frac{\epsilon^2}{2} \nabla \log p(x_t) + \epsilon Z_t \tag{4}$$

where x_t is the state at time t, ϵ is the step size, $\nabla \log p(x_t)$ is the gradient of the log-target distribution at x_t, and Z_t is a standard Gaussian random variable. The first term in the update represents a deterministic drift towards higher probability regions, while the second term introduces randomness to explore the state space.

Langevin MCMC is particularly useful in scenarios where the target distribution is known up to a normalization constant, which is common in Bayesian inference problems. It has been successfully applied in various fields such as machine learning, statistical physics, and computational biology, among others.

Langevin MCMC provides an efficient and effective means of sampling from complex, high-dimensional distributions by leveraging gradient information. This leads to faster convergence compared to traditional MCMC methods, making it a valuable tool in the arsenal of statistical and machine learning techniques.

4 Constrained Decoding with Langevin Dynamics (COLD)

[3, COLD] This is a decoding approach that treats text generation as sampling from an energy-based distribution, allowing for flexibly composing constraints based on the task at hand. COLD decoding generates text by sampling from an EBM defined over a sequence of "soft" tokens using Langevin dynamics, then maps the continuous sample into discrete, fluent text.

The set of constraints induces a distribution over text, written in an energy-based form as:

$$p(\boldsymbol{y}) = exp\{\sum_i \lambda_i f_i(\boldsymbol{y})\}/Z$$

where $\lambda_i \geq 0$ is the weight of the i^{th} constraint. Generating text under the constraints can then be seen as sampling from the energy-based distribution $\boldsymbol{y} \sim p(\boldsymbol{y})$.

For efficient sampling from $p(\boldsymbol{y})$ we want to use Langevin dynamics, which makes use of the gradient $\nabla_{\boldsymbol{y}} E(\boldsymbol{y})$. However, in our case $\boldsymbol{y}$ is a discrete sequence and the gradient $\nabla_{\boldsymbol{y}} E(\boldsymbol{y})$ is not well-defined. As a result, we perform Langevin dynamics with an energy defined on a sequence of continuous token vectors.

Instead of defining the energy function on discrete tokens, we define the energy function on a sequence of continuous vectors $\tilde{\boldsymbol{y}} = (\tilde{\boldsymbol{y}}_1, \ldots, \tilde{\boldsymbol{y}}_T)$, which we call a soft sequence. Each position in the soft sequence is a vector $\tilde{\boldsymbol{y}}_t \in \mathbb{R}^V$, where V is the vocabulary size, and each element $\tilde{\boldsymbol{y}}_t(v) \in \mathbb{R}$ corresponds to the logit of word v in the vocabulary.

Let's consider a specific example to understand the concept of a soft sequence in the context of natural language processing (NLP), specifically in a text generation task using a neural network model.

Imagine we are tasked with generating a text sequence one word at a time from a given vocabulary. For simplicity, let's assume our vocabulary consists of only four words: "the", "cat", "sat", and "mat". Thus, each position t in our output sequence will correspond to a soft sequence vector $\tilde{y}_t$ with a dimension of 4, where each element represents the logit for one of the words in the vocabulary.

At a certain timestep t, the model generates the following logits in $\tilde{y}_t$:

Word	**Logit**
"the"	2
"cat"	5
"sat"	1
"mat"	-1

Applying the softmax function to these logits gives us a probability distribution over our vocabulary:

Word	**Probability after Softmax**
"the"	0.12
"cat"	0.79
"sat"	0.07
"mat"	0.02

The model predicts "cat" as the most likely next word with a probability of 0.79. The vector $[0.12, 0.79, 0.07, 0.02]$ is the soft sequence at time t, representing a probability distribution over the vocabulary for the next word. This is a continuous representation, as opposed to selecting a single discrete word.

This soft sequence allows the model to express uncertainty and preferences over the next word in a nuanced way, enabling gradient-based optimization methods to work effectively by providing a differentiable way to adjust the model's parameters based on how close the soft predictions are to the desired output. In the final step of generating text, the model might sample from this distribution or pick the word with the highest probability, depending on the specific application or generation goal.

Taking the softmax of $\tilde{\boldsymbol{y}}_t$ yields a distribution over the vocabulary for position t, $\tilde{p}_t^\tau = \text{softmax}(\tilde{\boldsymbol{y}}_t/\tau)$. As $\tau \to 0$, $\tilde{p}_t^\tau$ becomes a one-hot vector, indicating a discrete token.

By specifying an energy $E(\tilde{\boldsymbol{y}})$ on the soft sequence $\tilde{\boldsymbol{y}}$, we can use Langevin dynamics to obtain a sample. Specifically, the sampling is done by forming a Markov chain:

$$\tilde{\boldsymbol{y}}^{(n+1)} \leftarrow \tilde{\boldsymbol{y}}^{(n)} - \eta \nabla_{\tilde{\boldsymbol{y}}} E(\tilde{\boldsymbol{y}}^{(n)}) + \epsilon^{(n)}, \tag{5}$$

where $\eta > 0$ is the step size, and $\epsilon^{(n)} \in \mathcal{N}(0, \sigma)$ is the noise at iteration n. By adding the right amount of noise and annealing the step size, the procedure will converge to samples from the true distribution. That is, if we let $p^{(n)}$ be the distribution such that $\tilde{y}^{(n)} \sim p^{(n)}$, then as $n \to \infty$ and $\sigma \to 0$, we have $p^{(n)} \to p(\tilde{y}) := \exp\{-E(\tilde{y})\}/Z$. That is, the procedure ends up generating samples from the distribution induced by the energy function.

4.1 Constraints

4.1.1 Soft Constraints

The soft fluency constraint is designed to promote fluency in generated text sequences by leveraging a pre-trained language model. The fluency of a sequence is evaluated based on how closely the generated soft sequence matches the distribution predicted by the language model. The constraint is mathematically represented as:

$$f_{\overrightarrow{\text{LM}}}(\tilde{\mathbf{y}}) = \sum_{t=1}^{T} \sum_{v \in V} p_{\overrightarrow{\text{LM}}}(v|\tilde{\mathbf{y}}_{<t}) \log \text{softmax}(\tilde{\mathbf{y}}_t(v)), \tag{6}$$

where:

- $f_{\overrightarrow{\text{LM}}}(\tilde{\mathbf{y}})$ is the fluency constraint for the soft sequence $\tilde{\mathbf{y}}$,
- T is the length of the soft sequence,
- V is the vocabulary set,
- $p_{\overrightarrow{\text{LM}}}(v|\tilde{\mathbf{y}}_{<t})$ is the probability of the word v at position t predicted by the language model given the preceding soft tokens $\tilde{\mathbf{y}}_{<t}$,
- $\tilde{\mathbf{y}}_t(v)$ is the logit for word v at position t in the soft sequence,
- The softmax function is applied to $\tilde{\mathbf{y}}_t(v)$ to convert logits into a probability distribution over the vocabulary for each position t.

The inner summation $\sum_{v \in V}$ computes the negative cross-entropy between the distribution predicted by the language model and the distribution represented by the soft sequence at each position t. The outer summation $\sum_{t=1}^{T}$ aggregates this measure across the entire sequence, ensuring that the generated sequence is fluent and coherent according to the language model's predictions.

This constraint effectively encourages the generated sequence to be not only fluent but also contextually coherent, making it a powerful tool for generating high-quality, natural-sounding text.

In the context of generating a sequence, T represents the length of the sequence being generated or evaluated. The knowledge of T beforehand depends on the specific generation task:

- **Fixed-Length Generation:** For tasks where the desired output length is predetermined, T is known at the outset. This allows the generation process to be tailored to produce sequences that exactly meet the predefined length.
- **Dynamic Generation:** In scenarios where the sequence ends based on specific criteria (like an end-of-sequence token), T is not fixed beforehand. Instead, it becomes known only as the generation process unfolds, with the sequence length adapting dynamically based on the content being generated.

4.1.2 Future-token Prediction Constraint

In applications such as text infilling, the task involves predicting missing tokens within a given sequence, where certain future input tokens remain fixed and are known beforehand. These fixed future tokens should influence the updating of past positions to ensure coherence and contextual relevance throughout the sequence.

For instance, consider the task of updating the missing word in the sentence "The __ has eight legs." It is essential that the predicted word for the blank space not only fits grammatically but also semantically with the future tokens, in this case, "has eight legs."

To address this requirement, we introduce a *future-token prediction constraint.* This constraint is designed to adjust the soft tokens in the sequence to maximize the likelihood of the known future input tokens, denoted as x_r. The constraint is mathematically represented as follows:

$$f_{pred}(\tilde{\mathbf{y}}; \mathbf{x_r}) = \sum_{k=1}^{K} \log p_{\overrightarrow{LM}}(x_{r,k} | \tilde{\mathbf{y}}, \mathbf{x}_{r,<k}), \tag{7}$$

where:

- **K** is the length of the sequence $\mathbf{x_r}$,

- $p_{\overrightarrow{LM}}(x_{r,k}|\tilde{\mathbf{y}}, \mathbf{x}_{r,<k})$ represents the probability, predicted by the underlying language model (**LM**), of the future token $x_{r,k}$ given the soft sequence $\tilde{\mathbf{y}}$ and all preceding future tokens $\mathbf{x}_{r,<k}$.

In essence, this constraint adjusts the soft sequence $\tilde{y}$ such that the underlying language model is more likely to predict the future tokens x_r correctly after being presented with $\tilde{y}$. This ensures that the generated or updated sequence is not only grammatically correct but also contextually coherent with the fixed future tokens.

4.1.3 N-gram Similarity Constraint

In many constrained generation scenarios, requirements are placed on the wording and expression of the generated text sequences. For example, lexically constrained generation tasks require the presence of certain keywords within the text samples, while tasks related to counterfactual reasoning or text editing necessitate that the generated text retains the essence of a reference sequence.

To address these requirements, we introduce an *n-gram similarity constraint*. This constraint is designed to favor sequences that exhibit overlap with a reference sequence y^* at the n-gram level. The constraint is mathematically expressed as follows:

$$f_{\text{sim}}(\tilde{\mathbf{y}}; \mathbf{y}^*) = \text{ngram-match}(\tilde{\mathbf{y}}, \mathbf{y}^*), \tag{8}$$

where ngram-match$(\cdot, \cdot)$ denotes a recent differentiable n-gram matching function, which serves as a differentiable approximation to the BLEU-n metric. This function is capable of quantifying the similarity between the generated sequence $\tilde{y}$ and the reference sequence y^* based on their shared n-grams.

When $n = 1$ and y^* represents a sequence of keywords, this constraint effectively enforces that $\tilde{y}$ assigns higher values to the specified keywords (1-grams), ensuring their presence in the generated text. Conversely, when n is larger and y^* is a more extensive reference sequence, the constraint encourages $\tilde{y}$ to resemble y^* more closely by assigning high values to tokens that constitute n-grams found within y^*.

This n-gram similarity constraint plays a critical role in ensuring that generated text sequences adhere to specified lexical requirements or retain significant elements of reference sequences, thereby enhancing the relevance and coherence of the generated content in accordance with the task-specific constraints.

4.2 From Soft to Discrete and Fluent Text

Upon obtaining a soft sequence sample $\tilde{\mathbf{y}}$ from running Langevin dynamics, our goal is to map this soft sequence to a discrete text sequence. This discrete sequence is considered as the output of COLD decoding. A straightforward approach for this mapping would involve selecting the most-likely token at each position t, defined as $y_t = \arg\max_v \tilde{\mathbf{y}}_t(v)$. However, this direct method might lead to text that suffers from fluency issues, even when a soft fluency constraint is employed. This is due to the presence of competing constraints that may compromise fluency for satisfying other criteria.

To address these challenges and ensure the fluency of the resulting text, we incorporate the underlying language model (e.g., $\mathbf{GPT2-XL}$) as a "guardian" in the process of obtaining the discrete sequence. Specifically, for each position t, the language model is used to produce the top-k most-likely candidate tokens based on its generation distribution, conditional on the preceding tokens. We denote this set of candidates as V_{k_t}.

From these top-k candidates, we then select the most likely token according to the soft sample $\tilde{\mathbf{y}}$:

$$y_t = \arg\max_{v \in V_{k_t}} \tilde{\mathbf{y}}_t(v). \tag{9}$$

This approach is referred to as "top-k filtering." Text generated through this method tends to exhibit higher fluency, as each token is among the top-k most probable choices according to the language model. Additionally, to facilitate the satisfaction of specific constraints (e.g., n-gram similarity), we may expand the candidate set V_{k_t} to include tokens that are critical for meeting those constraints, such as in tasks involving abductive reasoning and lexically constrained decoding.

4.3 Implementation of COLD Decoding

Sample-and-Select

COLD decoding facilitates the generation of multiple text samples from the distribution induced by the energy function $E(\tilde{\mathbf{y}})$. Based on the specific requirements of the task, these samples can either be presented as a set of outputs or a single sequence can be selected from the set according to certain criteria (e.g., different energy terms) for tasks as demonstrated in our experiments. This "sample-and-select" approach distinguishes itself from deterministic constrained decoding methods, which focus on optimizing a single sequence, and finds broad application across various generation settings.

Initialization

The soft sequence $\tilde{\mathbf{y}}$ is initialized using greedy decoding with the language model p_{LM} to generate initial output logits. Preliminary experiments indicate that the choice of initialization strategy has a limited impact on the generation results.

Noise Schedule

During each iteration of Langevin dynamics, noise $\boldsymbol{\epsilon}^{(n)} \sim \mathcal{N}(0, \sigma^{(n)})$ is added to the gradient. We employ a gradually decreasing schedule for $\sigma^{(n)}$ across iterations, transitioning the decoding process from exploration to optimization. Typically, the noise schedule sets or reduces σ to $\{1, 0.5, 0.1, 0.05, 0.01\}$ at iterations $\{0, 50, 500, 1000, 1500\}$, respectively.

Long Sequences

COLD decoding inherently produces a fixed-length sequence $\mathbf{y} = (y_1, \ldots, y_T)$. To generate longer sequences, particularly in cases where y_T does not signify the conclusion of a sentence, we utilize p_{LM} for a continuation of $\mathbf{y}$ using greedy decoding.

4.4 Implementation Details

The implementation involves initializing parameters and the noise term, setting up the optimizer and scheduler, and executing the optimization loop as follows:

```
    # Initialization
y_logits = init_logits
epsilon = torch.nn.Parameter(torch.zeros_like(y_logits))

# Optimizer and Scheduler Setup
if args.prefix_length > 0:
    optim = torch.optim.Adam([epsilon, prefix_logits], lr=args.stepsize)
else:
    optim = torch.optim.Adam([epsilon], lr=args.stepsize)
scheduler = torch.optim.lr_scheduler.StepLR(optimizer=optim, step_size=args.stepsize_iters,
                                             gamma=args.stepsize_ratio)

# Optimization Loop
optim.zero_grad()
y_logits_ = y_logits + epsilon
# Compute loss in a function (not shown)
if iter < args.num_iters - 1:
    loss.backward()  # Compute gradients
    optim.step()  # Update parameters
    scheduler.step()  # Update learning rate
    last_lr = scheduler.get_last_lr()[0]
```

This implementation iteratively updates the parameters to minimize the loss function, incorporating noise to aid in exploring the parameter space. The use of an adaptive learning rate scheduler and the Adam optimizer facilitates efficient and effective optimization.

4.5 Problems in their Implementation

The use of the Adam optimizer for the Langevin MCMC algorithm is based on the equation:

$$\tilde{\boldsymbol{y}}^{(n+1)} \leftarrow \tilde{\boldsymbol{y}}^{(n)} - \eta \nabla_{\tilde{\boldsymbol{y}}} E(\tilde{\boldsymbol{y}}^{(n)}) + \epsilon^{(n)}, \tag{10}$$

However, this approach diverges from the traditional Langevin MCMC methodology. Adam, being an adaptive optimizer, modifies learning rates based on the first and second moments of the gradients, which could interfere with the stochastic nature required by the Langevin dynamics.

In contrast, using `optim.SGD` with a fixed learning rate and incorporating noise with a specific standard deviation directly into the parameter updates can more faithfully replicate Langevin MCMC. This is because Langevin MCMC requires adding Gaussian noise to the gradient descent step to explore the parameter space more effectively. The standard deviation of this noise is crucial as it determines the "temperature" of the sampling process, influencing how the algorithm explores local minima. Therefore, while Adam may expedite convergence in certain optimization tasks, `optim.SGD` with carefully calibrated noise and learning rate parameters can better fulfill the requirements of Langevin MCMC, maintaining the balance between exploration and exploitation inherent to the algorithm.

To implement this using `optim.SGD`, the noise $\epsilon^{(n)}$ should be sampled from a Gaussian distribution where the variance is proportional to twice the learning rate. The appropriate equation that reflects this requirement is:

$$\epsilon^{(n)} \sim \mathcal{N}(0, 2\eta) \tag{11}$$

This equation ensures that the injected noise is consistent with the temperature of the Langevin dynamics, thereby facilitating an exploration that mimics the theoretical underpinnings of Langevin MCMC more closely than what is achieved using Adam.

4.6 Loss Computation Overview

The loss computation for models operating in counterfactual, abductive, and lexical modes involves multiple components, including left-to-right (L2R) loss, right-to-left (R2L) loss, and constraint losses. The total loss is a combination of these components, weighted appropriately according to the model configuration.

4.6.1 Loss Components

L2R Loss: The L2R loss is computed using a soft negative log-likelihood function, applied to the model's forward pass logits, after applying a top-k filter and adjusting by an output logits temperature.

R2L Loss: The R2L loss computation varies based on the mode:

- In counterfactual mode, it involves reversing the model's logits, processing them through the backward model, and computing the soft NLL against the adjusted and reversed logits.
- In abductive and lexical modes, the logits are flipped, processed through a backward model, adjusted for repetition, and the soft NLL is computed against the filtered and temperature-adjusted logits.

Constraint Loss: Depending on the mode:

- Counterfactual mode utilizes a BLEU-like loss computed against a set of n-grams.
- Abductive and lexical modes combine a cross-entropy loss and a BLEU-like loss, with the latter weighted by a specific factor.

4.6.2 Total Loss Computation

The total loss is calculated as a weighted sum of the L2R and R2L losses, alongside the constraint loss, each adjusted by specific weights reflecting the importance of each component and the overall model configuration. The formula for the total loss is:

$$\begin{aligned}\text{loss} = {} & (1.0 - \text{args.constraint\_weight}) \times \text{args.lr\_nll\_portion} \times \text{lr\_nll\_loss} \\ & + \text{args.constraint\_weight} \times (1 - \text{args.lr\_nll\_portion}) \times \text{rl\_nll\_loss} \\ & + \text{args.constraint\_weight} \times \text{c\_loss}\end{aligned}$$

This loss is then averaged across the batch for optimization.

5 Multiple Constraints from LMs using Langevin Dynamics (MuCoLa)

[2,] This work defines a Markov Chain using gradients of the energy function with respect to token embeddings of the output sequence. Stochasticity is introduced into the process to generate diverse samples, achieved by modifying the gradients with additive noise, a process known as Langevin Dynamics. The energy function is operationalized by setting each constraint to be smaller than a threshold, and expressing it as a Lagrangian—with language model likelihood as the primary objective. This approach allows for the combination of any number of constraints of varying scales without the need for tuning their weights. It is demonstrated that low-energy solutions under this definition are true samples from the LM distribution. This algorithm is referred to as **MuCoLa**, an acronym for sampling with multiple constraints from LMs using Langevin Dynamics.

Let $P(\boldsymbol{y}|\boldsymbol{x};\theta)$ model the conditional probability distribution of an output token sequence $\boldsymbol{y} = (y_1, \dots, y_N)$, given an optional input token sequence $\boldsymbol{x} = (x_1, \dots, x_M)$ where $x_m, y_n \in V$. We are interested in constrained sampling from P—finding output sequences $\boldsymbol{y}$ that have a high probability under P while minimizing a given set of constraint functions: $\{f_1, \dots, f_C\}$. We assume that each $f_i : ([\boldsymbol{x}], \boldsymbol{y}) \rightarrow \mathbb{R}$ is defined such that a lower value of f_i implies that the output better satisfies the constraint.

Instead of representing each target token y_n as a soft representation over the vocabulary $\tilde{y}_n \in \mathbb{R}^{|V|}$, we represent it as $\tilde{e}_n \in \boldsymbol{E}$, where $\boldsymbol{E}$ denotes the embedding table of the underlying language model containing $|V|$ vectors of size d, with $d \ll |V|$. We denote this sequence of embeddings as $\tilde{\boldsymbol{e}} = \{\tilde{e}_1, \ldots, \tilde{e}_N\}$.

At an update step t, instead of feeding each $\tilde{\boldsymbol{y}}$ to the model(s) (which are then transformed into an embedding to be fed to the first layer), we directly feed $\tilde{\boldsymbol{e}}$ to the first layer to compute the energy function, now defined as a function of embeddings instead of tokens. In the case of deterministic minimization, these vectors are updated as:

$$\tilde{\boldsymbol{e}}_t = \mathrm{Proj}_{\boldsymbol{E}}\left(\tilde{\boldsymbol{e}}^{t-1} - \eta \nabla_{\tilde{\boldsymbol{e}}} \mathcal{E}(\tilde{\boldsymbol{e}}^{t-1})\right)$$

where $\mathrm{Proj}_{\boldsymbol{E}}(\hat{e}) = \arg\min_{e \in \boldsymbol{E}} \|e - \hat{e}\|^2$ denotes a projection operation on the embedding table $\boldsymbol{E}$. In other words, after every gradient step, we project each updated vector back to a quantized space, that is the embedding table, using Euclidean distance as the metric. This projection is done to prevent adversarial solutions.

5.1 Toy Example: Energy-Based Language Model

Consider a toy energy-based language model (LM) over a sequence of N tokens, with a binary vocabulary and a one-dimensional embedding. Let the vocabulary be $\mathcal{V} = \Sigma = \{-1, +1\}$.

We define an energy function in the form:

$$U(\boldsymbol{x}) = -\beta \left(\frac{1}{2} \boldsymbol{x}^T A \boldsymbol{x} + \boldsymbol{b}^T \boldsymbol{x} \right)$$

where $\boldsymbol{x} \in \mathcal{V}^N$. The probability distribution of the model is given by:

$$\pi_{toy}(\boldsymbol{x}) \propto \exp(-U(\boldsymbol{x}))$$

Here, A is the adjacency matrix of an N-cycle. Specifically, A is a symmetric $N \times N$ matrix with $A_{i,i+1} = A_{i+1,i} = 1$ for $i = 1, \ldots, N-1$, and $A_{1,N} = A_{N,1} = 1$, with all other elements being zero. This structure models the interaction between adjacent tokens in the sequence. The vector $\boldsymbol{b}$ is set to zero ($\boldsymbol{b} = \boldsymbol{0}$).

The final form of the energy function, considering the specific form of A and $\boldsymbol{b} = \boldsymbol{0}$, is:

$$U(\boldsymbol{x}) = -\frac{\beta}{2} \sum_{i=1}^{N} (x_i x_{i+1} + x_N x_1)$$

where x_i is the i-th component of $\boldsymbol{x}$ and indices are taken modulo N.

This setup corresponds to a linear-chain Ising model with zero magnetic field. The Ising model is a statistical mechanical model used to describe ferromagnetism in statistical physics. In our context, the model describes a sequence of tokens (spins) where each token interacts only with its immediate neighbors. The absence of an external magnetic field (represented by $\boldsymbol{b} = \boldsymbol{0}$) simplifies the model, focusing on the interactions between adjacent tokens.

The energy function $U(\boldsymbol{x})$ is differentiable with respect to β, as it consists of linear and quadratic terms in $\boldsymbol{x}$. This differentiability is crucial for applying optimization algorithms and gradient-based learning methods.

The reasons for choosing this model are:

1. Differentiability of the energy function enables the application of gradient-based algorithms.
2. The model allows for exact computation of the distribution for small N.
3. The binary vocabulary simplifies the computation of the transition matrix of MuCoLa exactly and the stationary distribution as well.

References

[1] Li Du, Afra Amini, Lucas Torroba Hennigen, Xinyan Velocity Yu, Jason Eisner, Holden Lee, and Ryan Cotterell. Principled gradient-based markov chain monte carlo for text generation, 2023.

[2] Sachin Kumar, Biswajit Paria, and Yulia Tsvetkov. Gradient-based constrained sampling from language models, 2022.

[3] Lianhui Qin, Sean Welleck, Daniel Khashabi, and Yejin Choi. Cold decoding: Energy-based constrained text generation with langevin dynamics, 2022.