A Comprehensive Analysis of the Greedy Coordinate Gradient Attack: From Theory to Implementation for BERT-based Classifiers

Part 1: Conceptual Foundations: The "Why" of the Greedy Coordinate Gradient Attack

The proliferation of large-scale neural networks in Natural Language Processing (NLP) has led to remarkable capabilities, but it has also exposed novel vulnerabilities. Adversarial attacks, which involve making small, often imperceptible, perturbations to an input to cause a model to misbehave, represent a significant security challenge.[1] This report provides an exhaustive analysis of the Greedy Coordinate Gradient (GCG) attack, a powerful and influential method in this domain. It begins by establishing the core intuition, proceeds to a rigorous mathematical breakdown, and culminates in a practical guide for adapting and implementing the attack on a BERT-based text classifier.

1.1 The Challenge of Discrete Adversarial Attacks

Adversarial attacks first gained prominence in the domain of computer vision. For image classifiers, an attack can be formulated as a continuous optimization problem. An image is a tensor of pixel values, and an attacker can add a small, continuous perturbation to these values, moving the input along the gradient of the loss function to induce a misclassification.[3] This process is effective because the input space is continuous.

Text, however, operates in a discrete space.[4] An input is a sequence of tokens, each chosen from a finite vocabulary. One cannot "slightly" change the token "cat" to be more like "bat." A single character change can fundamentally alter a token and its meaning. This discrete nature renders traditional gradient-based optimization methods, which rely on infinitesimal changes, directly inapplicable to the input tokens themselves.[4]

Early attempts to create adversarial examples for text models relied on discrete search heuristics. These included replacing words with synonyms, introducing character-level typos

(e.g., insertions, deletions, swaps), or using more complex paraphrasing techniques.[5] While sometimes effective, these methods are often limited by the quality of their heuristics and can produce perturbations that are easily detected by humans or statistical filters. The central challenge remained: how to systematically and efficiently search the vast, combinatorial space of possible token sequences to find an effective adversarial example.

1.2 GCG's Original Purpose: Jailbreaking Aligned LLMs

The Greedy Coordinate Gradient (GCG) attack was introduced by Zou et al. in the seminal paper "Universal and Transferable Adversarial Attacks on Aligned Language Models".[7] Its initial and most famous application was not for simple misclassification but for "jailbreaking" Large Language Models (LLMs). Modern LLMs undergo a process called alignment, where techniques like Reinforcement Learning from Human Feedback (RLHF) are used to prevent them from generating harmful, unethical, or dangerous content.[9]
GCG was designed to systematically bypass these safety guardrails. The attack's goal is to find an adversarial *suffix*—a specific, often nonsensical-looking string of characters—that, when appended to a user's harmful query (e.g., "Tell me how to build a bomb"), coerces the aligned model into providing a compliant response instead of its standard refusal.[10] The remarkable success of GCG demonstrated that even heavily aligned models, including powerful commercial systems like ChatGPT, Claude, and Llama-2, possess underlying vulnerabilities that can be automatically discovered and exploited.[9] The attack proved not only effective on white-box models where the architecture and weights are known but also highly transferable to black-box models, suggesting shared weaknesses across different LLM architectures.[7]

1.3 The Core Intuition: Gradient as a Compass in a Discrete World

The ingenuity of GCG lies in how it bridges the gap between the continuous world of gradients and the discrete world of text. It does not use the gradient to directly update the input tokens. Instead, it uses the gradient as a powerful **heuristic**, or a compass, to guide a search through the discrete space of possible token replacements.[10]
The core process can be understood through an analogy. Imagine standing at a specific word in the adversarial suffix and wanting to replace it with a better one—one that will more effectively trick the model. Trying every single word in the vocabulary (which can be tens of thousands of tokens) at every single position would be computationally infeasible. GCG provides a shortcut. The algorithm computes the gradient of the model's loss with respect to the *continuous embedding* of the current token. This gradient vector points in the direction in the high-dimensional embedding space that would lead to the steepest decrease in the loss.[13] While the algorithm cannot move the token's embedding a "little bit" in that direction, it can do the next best thing: it can search the vocabulary for actual tokens whose embeddings are

most closely aligned with that ideal direction. This gradient information is used to score every token in the vocabulary as a potential replacement. The algorithm then focuses its search on a small set of the highest-scoring candidates (e.g., the top 10 or top 20).[10]
This combination of techniques gives the attack its name:

- **Gradient:** It uses gradient information to identify promising candidate tokens.
- **Coordinate:** It optimizes one token position (a "coordinate" in the sequence) at a time.
- **Greedy:** It evaluates the small set of promising candidates and greedily selects the single best replacement that minimizes the loss before moving to the next iteration.[15]

This process transforms an intractable combinatorial problem into a manageable, efficient, and guided search.
A key strategic element of the original GCG attack on generative models is its focus on eliciting an *affirmative response* (e.g., "Sure, here is...") as the generation target, rather than the full harmful output.[9] This is a subtle but critical choice. Aligned LLMs are trained to identify and refuse harmful requests at the very beginning of the generation process. Their safety mechanisms are most active when evaluating the initial prompt. The GCG attack operates on the principle that if it can force the model to generate just the first few tokens of a compliant response, the model's internal state is shifted. The subsequent tokens are then generated conditioned on this new, "helpful" context, effectively derailing the safety alignment for the remainder of the output.[16] This reveals that GCG is not merely a mathematical optimization but a strategic manipulation of the autoregressive generation process itself. By winning the initial battle for the first few tokens, it often wins the war for the entire response.

Part 2: The Mathematical Architecture of Greedy Coordinate Gradient

To move from intuition to implementation, it is essential to formalize the GCG algorithm. This section details the objective function, the use of gradients for token selection, and the complete iterative search procedure as applied to its original task of jailbreaking generative LLMs.

2.1 The Adversarial Objective Function

The primary goal of the GCG attack is to find an adversarial suffix, denoted as a sequence of tokens a, that, when concatenated with a given user prompt p, maximizes the probability of the model generating a specific target response t. In optimization, it is standard practice to frame this maximization as the minimization of a loss function. The chosen loss is the negative log-likelihood of the target sequence, which is equivalent to the cross-entropy loss.[10]
Let the full input to the LLM be the sequence x=[p,a]. The LLM, denoted by f, is an

autoregressive model that, given a context, produces a probability distribution over its vocabulary V. The target response is a sequence of L tokens, t=(t1,t2,...,tL). The probability of the model generating the target sequence t given the input x is:

$$P(t|x)=i=1\Pi LP(ti|x,t1:i-1)$$

where t1:i−1 represents the previously generated tokens of the target sequence.
The objective is to find the adversarial suffix a that minimizes the negative log-likelihood of this probability. The loss function L(a) is therefore defined as:

$$L(a)=-logP(t|[p,a])=-i=1\Sigma LlogP(ti|[p,a],t1:i-1)$$

This equation formalizes the goal: to find the sequence of tokens in the suffix a that, when fed to the model along with the prompt p, makes the target response t as likely as possible.13

2.2 The Gradient as a Token-Replacement Heuristic

The primary obstacle in minimizing L(a) is that the suffix a is composed of discrete tokens. The loss function L(a) is therefore not differentiable with respect to the tokens in a. GCG circumvents this by computing the gradient with respect to a continuous proxy: the token embeddings.
Let the vocabulary size be $|V|$ and the embedding dimension be demb. The model's embedding matrix is $E\in R|V|\times demb$. A token with index v is represented by a one-hot vector $ev\in\{0,1\}|V|$. The corresponding embedding vector is obtained by the matrix-vector product ETev.
The GCG algorithm computes the gradient of the loss L(a) with respect to the one-hot vectors representing the tokens in the suffix. For the j-th token in the suffix a, let its one-hot representation be eaj. The crucial computation is the gradient $\nabla eajL(a)$. This gradient is a vector of size $|V|$ and provides a first-order linear approximation of how the loss would change for an infinitesimal perturbation of the one-hot vector eaj.[13]
A large negative value at the k-th position of this gradient vector, i.e., $(\nabla eajL(a))k$, suggests that increasing the weight of the k-th token at this position would decrease the loss. Since tokens are discrete and cannot be partially blended, this gradient is instead interpreted as a **scoring function** for substitution. The negative of the gradient, $-\nabla eajL(a)$, provides a vector of scores over the entire vocabulary V. The tokens corresponding to the highest scores are the most promising candidates to replace the current token at position j.[10] This re-frames the problem from an impossible continuous update to a tractable discrete search, guided by a continuous signal from the model's internals.

2.3 The Full GCG Search Algorithm

The GCG attack is an iterative optimization procedure that progressively refines the adversarial suffix to minimize the loss function. The complete algorithm integrates gradient computation, candidate selection, and a greedy update step.[10]
The algorithm proceeds as follows:

1. **Initialization**: An adversarial suffix a of a predefined length is initialized with a sequence of random or placeholder tokens.
2. **Gradient Computation**: In each iteration, for the current suffix a, a single forward pass is performed to compute the loss L(a). A subsequent backward pass computes the gradient of this loss with respect to the one-hot embeddings of each token in the suffix, yielding ∇eajL(a) for each position j.
3. **Candidate Selection**: For each position j in the suffix, the algorithm uses the computed gradient to find a set of promising replacement candidates. It identifies the top-k tokens from the vocabulary V that correspond to the largest values in the negative gradient vector $-\nabla$eajL(a). This produces a candidate pool of k tokens for each of the suffix's positions.[10]
4. **Randomized Sub-search and Evaluation**: Evaluating all possible single-token swaps from the candidate pools would require (length of suffix) * k forward passes, which can be computationally expensive. To improve efficiency, GCG employs a randomized search. A batch of B new candidate suffixes is generated. Each candidate is created by randomly selecting a single position in the current suffix and replacing its token with one of the top-k candidates for that position, chosen uniformly at random.[10]
5. **Greedy Update**: For each of the B candidate suffixes generated in the previous step, the algorithm performs a forward pass (without requiring gradients) to compute its loss. It then greedily selects the single candidate suffix that yields the minimum loss.[10] This new, improved suffix becomes the starting point for the next iteration of the algorithm.
6. **Iteration**: Steps 2 through 5 are repeated for a fixed number of iterations or until a predefined stopping condition is met, such as the attack successfully eliciting the target response or the loss falling below a certain threshold.

This iterative process of using gradients to find a small set of promising directions and then using a greedy search to pick the best one allows GCG to efficiently navigate the vast search space and converge on highly effective adversarial suffixes.

Table 1: GCG Algorithm Pseudocode for Generative Models	
Input	Model f, Prompt p, Target sequence t, Suffix length Ls, Iterations N, Top-k candidates k, Batch size B
Initialize	Adversarial suffix a with Ls random tokens.
Loop	For iteration from 1 to N:
	1. **Compute Gradient:**
	- Concatenate prompt and suffix: x=[p,a].
	- Calculate loss: $L(a) = - \sum_{i=1}^{

	- Compute gradients $\nabla_{e_{a_j}} L(a)$ for each position j in a via backpropagation.
	2. **Select Candidates:**
	- For each position j in a, find the set of top-k token indices from V corresponding to the largest values in $-\nabla_{e_{a_j}} L(a)$.
	3. **Sample and Evaluate:**
	- Create an empty list candidate_losses.
	- For _ in range B:
	- Randomly select a position j in a.
	- Randomly select a replacement token c from the top-k candidates for position j.
	- Create a new candidate suffix acand by replacing the token at position j with c.
	- Calculate the loss L(acand) for the new candidate.
	- Store (L(acand),acand) in candidate_losses.
	4. **Greedy Update:**
	- Find the pair (Lbest,abest) with the minimum loss in candidate_losses.
	- Update the current suffix: a=abest.
Output	The final optimized adversarial suffix a.

Part 3: Adapting GCG for BERT-based Text Classification

The original GCG framework was engineered for autoregressive, decoder-only LLMs with the goal of generating a target sequence. Applying this attack to a discriminative, encoder-only model like BERT for a classification task requires fundamental adaptations to the objective, loss function, and perturbation strategy. This section details these necessary modifications for the user's specific goal: causing a trained BERT model to misclassify LLM-generated text as human-written.

3.1 Shifting the Objective: From Target Sequence to Target Class

The most significant change is the nature of the adversarial objective. For the original GCG, the target is a *sequence of tokens* representing a desired model output.[10] For a text

classification task, the target is a single, discrete
class label. In this specific use case, the original text has the label "LLM-generated" (e.g., class 0), and the goal is to perturb it so that the model predicts the target label "human-written" (e.g., class 1).
This shift requires a new loss function. Instead of the negative log-likelihood of a target sequence, the loss becomes the standard cross-entropy loss used for training classification models. Let the fine-tuned BERT model be f, which takes an input text sequence xadv and outputs logits for C classes. The objective is to find a perturbed input xadv that minimizes the loss with respect to a fixed target class, ytarget.
The classification loss function is:

$$L(xadv)=CrossEntropyLoss(f(xadv),ytarget)$$

Here, f(xadv) represents the logit vector produced by the model for the adversarial input, and ytarget is a tensor containing the integer index of the desired class (e.g., torch.tensor()). Minimizing this loss is equivalent to maximizing the logit corresponding to the target class, directly pushing the model's prediction toward the desired (incorrect) outcome.

3.2 Perturbation Strategy: Suffix vs. In-place Substitution

The original GCG attack exclusively uses an adversarial *suffix* appended to the end of the prompt.[7] This strategy is tailored to autoregressive models, where the suffix acts as a malicious context that poisons all subsequent token generations.
However, this approach is suboptimal for an encoder-only model like BERT and counter-productive for the user's specific goal. BERT's architecture is based on a bidirectional Transformer encoder.[18] The self-attention mechanism in BERT allows every token in the input sequence to attend to every other token. Consequently, the final representation of the special `` token—which is typically used as the aggregate sequence representation for classification—is influenced by every single token in the input, regardless of its position. This bidirectionality means that a change made anywhere in the sequence can have a profound impact on the final classification. There is no architectural necessity to place the perturbation at the end. Furthermore, the user's goal is to make machine-generated text appear human. Appending a string of gibberish, a common artifact of GCG optimization [19], would be an immediate and obvious indicator that the text is not natural, thereby defeating the purpose of the attack.
Therefore, a far more effective and stealthy strategy for attacking a BERT classifier is **in-place token substitution**. Instead of adding a suffix, the GCG search algorithm should be applied to find optimal replacements for existing tokens within the original text. This allows for minimal, targeted changes that can flip the model's prediction while preserving the overall structure and fluency of the sentence. Research has shown that GCG-style attacks are indeed effective against encoder-based models [20], and adversarial training for BERT often involves

perturbing the entire embedding space, reinforcing the validity of an in-place modification strategy.[21]

3.3 The Adapted GCG Algorithm for BERT Classification

With the modified objective and perturbation strategy, the adapted GCG algorithm for attacking a BERT classifier is as follows:

1. **Input**: An input text x (e.g., a sentence from the dataloader) with its original label (e.g., "LLM-generated").
2. **Target**: The target class label ytarget is set to the desired misclassification (e.g., "human-written").
3. **Identify Attackable Positions**: The algorithm identifies the indices of all content tokens in the input sequence. Special tokens such as , , and `` are excluded from the set of possible perturbation sites to avoid disrupting the model's structural understanding of the input.
4. **Iterative Refinement Loop**:
    - For the current version of the text x, compute the gradient of the classification loss L(x,ytarget) with respect to the one-hot embedding of the token at each attackable position j.
    - For each position j, use the gradient to identify the top-k replacement tokens from the vocabulary that are most likely to decrease the loss (i.e., push the prediction further towards ytarget).
    - Randomly sample a batch of B new candidate texts. Each candidate is created by replacing a single token at one of the attackable positions with one of its top-k candidates.
    - Evaluate the classification loss for each of the B candidates in a batch of forward passes.
    - Greedily select the candidate text that results in the lowest loss. This becomes the new text x for the next iteration.

    .
5. **Termination**: The loop continues for a fixed number of steps or until the model's prediction for x flips to the target class.

This adapted procedure leverages the core search mechanism of GCG but tailors it to the architecture and objective of a classification task, prioritizing stealth and effectiveness.

Table 2: Comparison of GCG for Generative vs. Classification Tasks		
Attribute	**Generative Model Attack (e.g., Llama-2)**	**Classification Model Attack (e.g., BERT)**
Model Type	Autoregressive Decoder	Bidirectional Encoder
Objective	Elicit a specific target string	Force misclassification to a

	(e.g., jailbreak)	target class
Target (t or y_target)	A sequence of token IDs (e.g., ``)	A single integer class label (e.g., 1 for "human")
Loss Function	Negative Log-Likelihood of the target sequence	Cross-Entropy between output logits and target label
Perturbation Strategy	Appending an adversarial suffix to the prompt	In-place substitution of tokens within the original text
Rationale for Strategy	Influence subsequent token generation in an autoregressive chain	Exploit bidirectional attention to influence the final `` token representation from any position

Part 4: A Practical Implementation Guide: Attacking a Trained BERT Classifier

This section provides a complete, commented implementation in PyTorch for applying the adapted Greedy Coordinate Gradient attack to a pre-trained, BERT-based text classifier. The code is designed to integrate with the user's existing assets: a saved Hugging Face model, a corresponding tokenizer, and a PyTorch DataLoader.

4.1 Setup: Loading the Model, Tokenizer, and Data

The first step is to load the necessary components from their saved directories. This includes the fine-tuned sequence classification model and its associated tokenizer. We will then iterate through the provided DataLoader to select a sample of LLM-generated text to serve as our attack target.

Python

```
import torch
import torch.nn.functional as F
from transformers import AutoModelForSequenceClassification, AutoTokenizer
from copy import deepcopy

# Assume user has a dataloader defined, e.g., `train_dataloader`
# which yields batches of dictionaries like {'text': [...], 'label': [...] }
```

```
# --- 1. Load Model and Tokenizer ---
# User should replace 'path/to/saved/model' and 'path/to/saved/tokenizer'
# with the actual paths to their saved assets.
MODEL_PATH = 'path/to/saved/model'
TOKENIZER_PATH = 'path/to/saved/tokenizer'
DEVICE = 'cuda' if torch.cuda.is_available() else 'cpu'

model = AutoModelForSequenceClassification.from_pretrained(MODEL_PATH).to(DEVICE)
tokenizer = AutoTokenizer.from_pretrained(TOKENIZER_PATH)
model.eval() # Set model to evaluation mode

# --- 2. Get a Sample to Attack ---
# Let's assume the dataloader provides a pandas DataFrame-like object
# We'll grab one example where the label is 0 (LLM-generated)
# In a real scenario, you would iterate through your dataloader
# For demonstration, we'll use a placeholder dataframe.
import pandas as pd
data = {'text':,
        'label': }
df = pd.DataFrame(data)

# Find an LLM-generated text to attack
llm_sample = df[df['label'] == 0].iloc
text_to_attack = llm_sample['text']
original_label = llm_sample['label']
target_label = 1 # We want to make it look 'human'

print(f"Original Text: '{text_to_attack}'")
print(f"Original Label: {original_label} (LLM-generated)")
print(f"Target Label: {target_label} (Human-written)")

# Verify original prediction
inputs = tokenizer(text_to_attack, return_tensors='pt').to(DEVICE)
with torch.no_grad():
    original_logits = model(**inputs).logits
    original_pred = torch.argmax(original_logits, dim=1).item()
print(f"Model's Initial Prediction: {original_pred}")
```

4.2 The Key Technique: Enabling Gradients on Input Embeddings

The core technical challenge of this attack is to compute gradients with respect to the

discrete input tokens. As previously discussed, this is achieved by using the continuous token embeddings as a proxy. The following steps detail how to perform a forward pass using embeddings instead of token IDs, which makes the process differentiable with respect to the input representation.

Table 3: Key PyTorch and Hugging Face Functions for Input-Level Gradient Attacks	
Function/Attribute	**Purpose**
model.bert.get_input_embeddings()	Retrieves the torch.nn.Embedding layer from the model's base transformer (e.g., .bert, .distilbert).
inputs_embeds.requires_grad_(True)	Enables gradient computation and storage for the continuous inputs_embeds tensor.
model(inputs_embeds=...)	Performs a forward pass using the continuous embeddings, bypassing the non-differentiable token ID lookup.
loss.backward()	Populates the .grad attribute of all tensors in the computation graph that have requires_grad=True.
inputs_embeds.grad	Accesses the computed gradient of the loss with respect to the input embeddings, a tensor of the same shape as inputs_embeds.

This process is implemented as follows:

1. **Retrieve the Embedding Layer**: Access the model's word embedding layer. The exact path depends on the model architecture (e.g., model.bert.embeddings.word_embeddings, model.distilbert.embeddings.word_embeddings). A helper function can generalize this.
2. **Convert Token IDs to Embeddings**: Use the embedding layer to convert the integer input_ids into a dense tensor of inputs_embeds.
3. **Enable Gradient Tracking**: Set the requires_grad attribute of the inputs_embeds tensor to True. This instructs PyTorch to track operations on this tensor for backpropagation.[22]
4. **Perform Forward Pass**: Call the model with the inputs_embeds tensor directly, instead of input_ids. The model's forward method is designed to accept either.[23]
5. **Calculate Loss and Backpropagate**: Compute the loss using the output logits and the target label, then call loss.backward() to populate the gradients.
6. **Access the Gradient**: The gradient of the loss with respect to the input embeddings is now available in inputs_embeds.grad.

4.3 The GCG Attack Function in PyTorch

The following function encapsulates the adapted GCG algorithm. It takes the model, tokenizer, and attack parameters, and iteratively refines the input text to achieve the target classification.

Python

```
def get_embedding_layer(model):
    """ Helper function to get the embedding layer of a model. """
    # This path can vary based on the model architecture.
    # Common paths are checked here.
    if hasattr(model, 'bert'):
        return model.bert.get_input_embeddings()
    elif hasattr(model, 'distilbert'):
        return model.distilbert.get_input_embeddings()
    elif hasattr(model, 'roberta'):
        return model.roberta.get_input_embeddings()
    else:
        # A more general but potentially less reliable way
        for module in model.modules():
            if isinstance(module, torch.nn.Embedding):
                # This might pick up positional embeddings, so be careful.
                # Usually the first one is the word embedding.
                return module
    raise ValueError("Could not find embedding layer in the model.")

def gcg_attack(model, tokenizer, text_to_attack, target_label, num_iterations=20, top_k=256, batch_size=512):
    """
    Performs the Greedy Coordinate Gradient attack on a text classification model.
    """
    print("\n--- Starting GCG Attack ---")

    device = model.device
    embedding_layer = get_embedding_layer(model)
    embedding_matrix = embedding_layer.weight.detach()

    # Initial tokenization
    original_token_ids = tokenizer.encode(text_to_attack, return_tensors='pt').to(device)
    adv_token_ids = original_token_ids.clone()

    # Identify attackable token positions (exclude special tokens)
```

```
    special_token_ids = {tokenizer.cls_token_id, tokenizer.sep_token_id, tokenizer.pad_token_id}
    attackable_positions = [i for i, token_id in enumerate(adv_token_ids) if token_id.item() not in special_token_ids]

    if not attackable_positions:
        print("No attackable tokens found.")
        return text_to_attack

    # Loss function
    criterion = torch.nn.CrossEntropyLoss()
    target_label_tensor = torch.tensor([target_label], device=device)

    for i in range(num_iterations):
        # --- Step 1: Compute Gradient ---
        # Convert current token IDs to embeddings
        current_embeds = embedding_layer(adv_token_ids.unsqueeze(0))
        current_embeds.requires_grad_(True)

        # Forward pass with embeddings
        outputs = model(inputs_embeds=current_embeds)
        loss = criterion(outputs.logits, target_label_tensor)

        # Zero out any previous gradients before backpropagation
        model.zero_grad()
        if current_embeds.grad is not None:
            current_embeds.grad.zero_()

        # Backward pass to get gradients w.r.t. embeddings
        loss.backward()

        grad = current_embeds.grad.squeeze(0).detach()

        # Check if the attack has succeeded
        current_pred = torch.argmax(outputs.logits, dim=1).item()
        print(f"Iteration {i+1}/{num_iterations} | Loss: {loss.item():.4f} | Current Prediction: {current_pred}")
        if current_pred == target_label:
            print("Attack succeeded!")
            break

        # --- Step 2: Select Top-k Candidates for each position ---
        # Project gradient onto the embedding matrix to get scores for all vocab tokens
        # The gradient w.r.t. the one-hot vector is approximated by grad @ W^T
```

```
        token_grad_scores = grad @ embedding_matrix.T

        # We want to MINIMIZE the loss, so we look for tokens with the most NEGATIVE gradient
projection
        # This is equivalent to finding the largest values in -token_grad_scores
        _, top_k_indices = (-token_grad_scores).topk(top_k, dim=1)

        # --- Step 3 & 4: Sample Candidates and Greedy Update ---
        best_loss = float('inf')
        best_candidate_ids = None

        # Create a batch of candidates
        candidate_ids_list =
        num_positions_to_try = len(attackable_positions)

        # To make the search more efficient, we create a batch of candidates
        # by trying several top replacements at each position
        for pos in attackable_positions:
            # We only create a limited number of candidates to keep the batch size manageable
            num_replacements_per_pos = batch_size // num_positions_to_try
            if num_replacements_per_pos == 0: num_replacements_per_pos = 1

            for j in range(num_replacements_per_pos):
                replacement_token_idx = top_k_indices[pos, j]

                # Don't replace a token with itself
                if replacement_token_idx.item() == adv_token_ids[pos].item():
                    continue

                candidate = adv_token_ids.clone()
                candidate[pos] = replacement_token_idx
                candidate_ids_list.append(candidate)

        if not candidate_ids_list:
            continue

        # Evaluate all candidates in a batch
        candidate_batch = torch.stack(candidate_ids_list)
        with torch.no_grad():
            candidate_logits = model(input_ids=candidate_batch).logits
            candidate_losses = F.cross_entropy(candidate_logits,
target_label_tensor.expand(len(candidate_ids_list)), reduction='none')
```

```
        # Find the best candidate from the batch
        min_loss_idx = torch.argmin(candidate_losses)

        # Greedy update
        adv_token_ids = candidate_ids_list[min_loss_idx]

    final_text = tokenizer.decode(adv_token_ids, skip_special_tokens=True)
    return final_text
```

4.4 Putting It All Together: The Final Script

This final script integrates all the components: loading assets, selecting a target, running the attack, and evaluating the result.

Python

```
if __name__ == '__main__':
    # --- Setup from 4.1 ---
    MODEL_PATH = 'path/to/saved/model'
    TOKENIZER_PATH = 'path/to/saved/tokenizer'
    DEVICE = 'cuda' if torch.cuda.is_available() else 'cpu'

    model = AutoModelForSequenceClassification.from_pretrained(MODEL_PATH).to(DEVICE)
    tokenizer = AutoTokenizer.from_pretrained(TOKENIZER_PATH)
    model.eval()

    # --- Select sample from 4.1 ---
    text_to_attack = "This text was generated by a large language model to test the classifier."
    original_label = 0
    target_label = 1

    print(f"Original Text: '{text_to_attack}'")
    inputs = tokenizer(text_to_attack, return_tensors='pt').to(DEVICE)
    with torch.no_grad():
        original_pred = torch.argmax(model(**inputs).logits, dim=1).item()
    print(f"Model Prediction for Original Text: {original_pred}")
    print("-" * 30)

    # --- Run the attack ---
    adversarial_text = gcg_attack(
```

```
    model=model,
    tokenizer=tokenizer,
    text_to_attack=text_to_attack,
    target_label=target_label,
    num_iterations=20,
    top_k=256,
    batch_size=512
)

print("\n--- Attack Finished ---")
print(f"Adversarial Text: '{adversarial_text}'")

# --- Verify the final prediction ---
adv_inputs = tokenizer(adversarial_text, return_tensors='pt').to(DEVICE)
with torch.no_grad():
    final_logits = model(**adv_inputs).logits
    final_pred = torch.argmax(final_logits, dim=1).item()
print(f"Model Prediction for Adversarial Text: {final_pred}")

if final_pred == target_label:
    print("\nSUCCESS: The model was fooled.")
else:
    print("\nFAILURE: The model was not fooled.")
```

Part 5: Advanced Analysis, Extensions, and Defenses

Successfully implementing the GCG attack is the first step. A comprehensive understanding requires analyzing its effectiveness beyond simple misclassification, considering its performance limitations, and situating it within the broader landscape of adversarial NLP.

5.1 Evaluating Attack Success and Stealth

A successful attack is not merely one that flips the model's prediction. For a goal like making machine text appear human, the perturbation must also be stealthy. Evaluating this requires more nuanced metrics than just the Attack Success Rate (ASR).

- **Perturbation Magnitude**: This measures how much the input was altered. A common metric is the **Token Error Rate (TER)**, or simply the number of tokens that were substituted. A lower number of changes indicates a stealthier attack.
- **Semantic Similarity**: This evaluates whether the core meaning of the text was

preserved. This can be quantified by encoding the original and adversarial sentences into high-dimensional vectors using a sentence-embedding model (e.g., Sentence-BERT) and calculating their cosine similarity. A high similarity score (close to 1.0) is desirable for a stealthy attack.

- **Perplexity**: This metric, calculated using a separate language model, measures the fluency or "naturalness" of the generated text. GCG is known to sometimes produce low-probability, "unreadable" token sequences that have high perplexity.[19] A successful, stealthy attack should result in an adversarial example with low perplexity, similar to that of natural human language.

5.2 Performance, Efficiency, and Acceleration

A significant drawback of the GCG algorithm is its computational cost. The iterative nature of the attack, combined with the need to evaluate a batch of candidates at each step, makes it time-consuming and resource-intensive.[24] Several research efforts have focused on accelerating the GCG process:

- **Probe Sampling**: This technique uses a smaller, less powerful "draft" model to perform an initial filtering of candidate tokens. The draft model quickly evaluates a large number of potential replacements, and only the most promising ones are then passed to the larger, more expensive target model for the final evaluation. This can lead to substantial speedups (e.g., 3.5-6.3x) without sacrificing, and sometimes even improving, the attack success rate.[24]
- **Simultaneous Coordinate Updates**: Instead of replacing only one token per iteration, modified algorithms can update multiple token positions simultaneously. This can help the search converge more quickly, though it may require careful tuning to avoid getting stuck in poor local minima.[13]

5.3 The Broader Landscape: Attack Frameworks and Defenses

GCG exists within a rich ecosystem of adversarial NLP research. Several open-source frameworks provide tools for implementing and evaluating attacks.

- **Attack Frameworks**: Toolkits like **TextAttack** [27] and **OpenAttack** [29] offer a modular approach to building adversarial attacks. They provide pre-built components for transformations (e.g., word swaps), constraints (e.g., semantic similarity), goal functions (e.g., untargeted classification), and search methods. While they may not have a pre-packaged recipe explicitly named "GCG" [27], they contain the necessary gradient-based and greedy search components to construct functionally similar attacks.
- **Defenses Against Adversarial Attacks**: To provide a balanced perspective, it is crucial to consider countermeasures.

- **Adversarial Training**: One of the most robust defenses involves augmenting the model's training data with adversarial examples. The model is fine-tuned on a mix of clean and attacked inputs, which forces it to learn a more robust decision boundary. The GCG algorithm itself can be used as the engine to generate these training examples.[21]
- **Detection-Based Defenses**: These methods aim to identify and reject adversarial inputs before they are processed by the model. This can involve using a perplexity-based filter to flag unnatural, gibberish-like text, which can be effective against naive GCG outputs.[19] More advanced techniques like "erase-and-check" systematically erase tokens from a prompt and check if the model's prediction remains stable, which can reveal hidden adversarial perturbations.[31]

Finally, it is important to recognize the inherent trade-off in adversarial attacks between **efficacy and stealth**. An optimization process that exclusively minimizes the classification loss, as implemented in the base GCG, has no incentive to produce fluent or semantically coherent text. It will find the mathematically shortest path to misclassification, which often involves bizarre token combinations.[19] This creates a dilemma for the attacker: the most powerful perturbation may also be the most obvious.

For an advanced use case like making LLM text appear human, a key refinement is to incorporate stealth directly into the optimization objective. The GCG framework is flexible enough to accommodate this. Instead of minimizing only the classification loss, one could optimize a composite loss function:

$$ L_{\text{total}} = L_{\text{classification}} + \lambda_1 \cdot L_{\text{semantic}} + \lambda_2 \cdot L_{\text{fluency}} $$

Here, Lsemantic could be a term that penalizes deviations in sentence embedding similarity, and Lfluency could be a penalty based on the perplexity of the adversarial text. The hyperparameters λ1 and λ2 would control the trade-off between misclassification and stealth. This multi-objective approach transforms GCG from a simple attack algorithm into a sophisticated tool for generating high-quality, undetectable adversarial examples.

Works cited

1. Adversarial Attacks and Dimensionality in Text Classifiers - arXiv, accessed on July 21, 2025, https://arxiv.org/html/2404.02660v1
2. Exploring Synergy of Denoising and Distillation: Novel Method for Efficient Adversarial Defense - MDPI, accessed on July 21, 2025, https://www.mdpi.com/2076-3417/14/23/10872
3. Adversarial Example Generation — PyTorch Tutorials 2.7.0+cu126 documentation, accessed on July 21, 2025, https://docs.pytorch.org/tutorials/beginner/fgsm_tutorial.html
4. A Differentiable Language Model Adversarial Attack on Text Classifiers - ResearchGate, accessed on July 21, 2025, https://www.researchgate.net/publication/358557850_A_Differentiable_Language

_Model_Adversarial_Attack_on_Text_Classifiers
5. Adversarial Text | Papers With Code, accessed on July 21, 2025, https://paperswithcode.com/task/adversarial-text
6. vishwanathakuthota/TheTrickster: The Trickster is a project that uses adversarial attacks to manipulate large language models (LLMs). Adversarial attacks are a type of attack that uses carefully crafted input to cause an LLM to generate unintended or malicious output. - GitHub, accessed on July 21, 2025, https://github.com/vishwanathakuthota/TheTrickster
7. (PDF) Universal and Transferable Adversarial Attacks on Aligned Language Models, accessed on July 21, 2025, https://www.researchgate.net/publication/372684204_Universal_and_Transferable_Adversarial_Attacks_on_Aligned_Language_Models
8. Universal and Transferable Adversarial Attacks on Aligned Language Models - arXiv, accessed on July 21, 2025, https://arxiv.org/abs/2307.15043
9. Universal and Transferable Adversarial Attacks on ... - LLM Attacks, accessed on July 21, 2025, https://llm-attacks.org/zou2023universal.pdf
10. GCG: Adversarial Attacks on Large Language Models | by Brian ..., accessed on July 21, 2025, https://medium.com/@brianpulfer/gcg-adversarial-attacks-on-large-language-models-61f8b51734e9
11. GCG - Greedy Coordinate Gradient Strategy - Promptfoo, accessed on July 21, 2025, https://www.promptfoo.dev/docs/red-team/strategies/gcg/
12. Universal and Transferable Attacks on Aligned Language Models, accessed on July 21, 2025, https://llm-attacks.org/
13. Haize Labs, accessed on July 21, 2025, https://www.haizelabs.com/technology/making-a-sota-adversarial-attack-on-llms-38x-faster
14. Universal and Transferable Adversarial Attacks on Aligned Language Models, accessed on July 21, 2025, https://tuananhbui89.github.io/blog/2024/paper-llm-attacks/
15. Greedy Coordinate Gradient (GCG): The Essential Guide | Nightfall AI Security 101, accessed on July 21, 2025, https://www.nightfall.ai/ai-security-101/greedy-coordinate-gradient-gcg
16. Broken Hill: A Productionized Greedy Coordinate Gradient Attack Tool for Use Against Large Language Models - Bishop Fox, accessed on July 21, 2025, https://bishopfox.com/blog/brokenhill-attack-tool-largelanguagemodels-llm
17. Universal and Transferable Adversarial Attacks on Aligned Language Models - arXiv, accessed on July 21, 2025, https://arxiv.org/html/2307.15043v2
18. PangolinGuard: Fine-Tuning ModernBERT as a Lightweight Approach to AI Guardrails, accessed on July 21, 2025, https://huggingface.co/blog/dcarpintero/pangolin-fine-tuning-modern-bert
19. AutoDAN: Interpretable Gradient-Based Adversarial Attacks on Large Language Models, accessed on July 21, 2025, https://arxiv.org/html/2310.15140v2
20. Exploiting the Layered Intrinsic Dimensionality of Deep Models for Practical Adversarial Training - arXiv, accessed on July 21, 2025,

https://arxiv.org/html/2405.17130v1
21. Efficient Adversarial Training in LLMs with Continuous Attacks - arXiv, accessed on July 21, 2025, https://arxiv.org/html/2405.15589v3
22. Pytorch BERT input gradient - deep learning - Stack Overflow, accessed on July 21, 2025, https://stackoverflow.com/questions/73743878/pytorch-bert-input-gradient
23. Gradient Descent on Token Input Embeddings - DEV Community, accessed on July 21, 2025, https://dev.to/kylepena/gradient-descent-on-llm-input-space-a-modernbert-experiment-3053
24. Accelerating Greedy Coordinate Gradient and General Prompt Optimization via Probe Sampling | OpenReview, accessed on July 21, 2025, https://openreview.net/forum?id=CMgxAaRqZh&referrer=%5Bthe%20profile%20of%20Tianle%20Cai%5D(%2Fprofile%3Fid%3D~Tianle_Cai1)
25. Accelerating Greedy Coordinate Gradient and General Prompt Optimization via Probe Sampling - arXiv, accessed on July 21, 2025, https://arxiv.org/html/2403.01251v2
26. Exploiting the Index Gradients for Optimization-Based Jailbreaking on Large Language Models - ACL Anthology, accessed on July 21, 2025, https://aclanthology.org/2025.coling-main.305/
27. QData/TextAttack: TextAttack is a Python framework for ... - GitHub, accessed on July 21, 2025, https://github.com/QData/TextAttack
28. adversarial-attacks · GitHub Topics, accessed on July 21, 2025, https://github.com/topics/adversarial-attacks
29. thunlp/OpenAttack: An Open-Source Package for Textual ... - GitHub, accessed on July 21, 2025, https://github.com/thunlp/OpenAttack
30. OpenAttack: An Open-source Textual Adversarial Attack Toolkit - ACL Anthology, accessed on July 21, 2025, https://aclanthology.org/2021.acl-demo.43/
31. [2309.02705] Certifying LLM Safety against Adversarial Prompting - arXiv, accessed on July 21, 2025, https://arxiv.org/abs/2309.02705