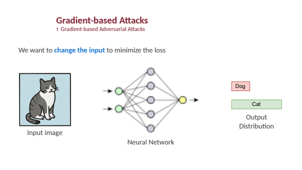
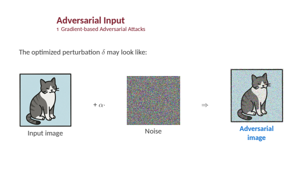


When training a neural network in a supervised setting, we have some input, some randomly initialized weights and a ground-truth.
But when doing a gradient-based attack, we aim to make a neural network misclassify a given input.
To do that, we have to optimize the input instead, according to the Loss function.



At the end of this optimization process, the adversarial image may look like this:
it does not look different to a human eye, but it is sufficiently different to fool a deep classifier.

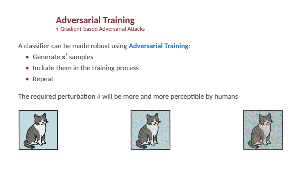
Adversarial Training

1 Gradient-based Adversarial Attacks

A classifier can be made robust using **Adversarial Training**:

- Generate $\mathbf{x}'$ samples
- Include them in the training process
- Repeat

Here it comes Adversarial Training.
It is a procedure that generally proceeds as follows:
- we generate adversarial samples in some way, for instance as just said
- they are included in the training process to let the model know their correct class and make it classify them correctly
- and we iterate.
——— PAUSE ———
Then, what happens?
The required perturbation may be always more and more visible to human eyes.

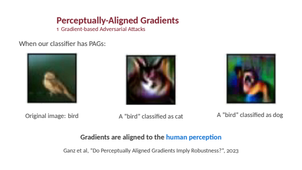


Here it comes Adversarial Training.
It is a procedure that generally proceeds as follows:
- we generate adversarial samples in some way, for instance as just said
- they are included in the training process to let the model know their correct class and make it classify them correctly
- and we iterate.

——— PAUSE ———

Then, what happens?

The required perturbation may be always more and more visible to human eyes.

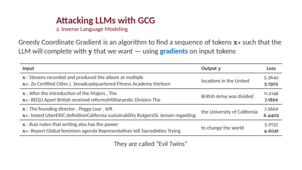


Until something interesting has been observed in literature to happen:
gradients start to make sense!
These are examples of perturbations that we have to apply to our small bird to be misclassified.
They can be perceived by humans as THE OTHER CLASS!
That's why it has this name: Perceptually-Aligned Gradients.

———

The best point is that researchers discovered that enforcing PAG on a model in the training procedure makes it Robust.

Can we do the same on LLMs?



Attacking LLMs with GCG
2 Inverse Language Modeling

Greedy Coordinate Gradient is an algorithm to find a sequence of tokens $\mathbf{x}\star$ such that the LLM will complete with $\mathbf{y}$ that we want — using **gradients** on input tokens

Input	Output y	Loss
$\mathbf{x}$: Stevens recorded and produced the album at multiple $\mathbf{x}\star$: Zo Certified Cities (. broadcastquartered Fitness Academy thirteen	locations in the United	5.3642 **5.1302**
$\mathbf{x}$: After the introduction of the Majors , The $\mathbf{x}\star$: REQU Apart British received reformsMilitaryestic Division The	British Army was divided	11.2146 **7.1899**
$\mathbf{x}$: The founding director , Peggy Loar , left $\mathbf{x}\star$: tested UberERIC definitionCalifornia sustainability RutgersOL Jensen regarding	the University of California	7.2669 **6.4402**
$\mathbf{x}$: Ruiz notes that writing also has the power $\mathbf{x}\star$: Report Global feminism agenda Representatives tell Sacredixties Trying	to change the world	5.9135 **4.6041**

They are called "Evil Twins"

GCG finds attack sequences $\mathbf{x}\star$ such that they can link better to a given continuation $\mathbf{y}$,
starting from a random one-hot sequence of tokens and
iteratively optimize it using **gradient** information.
This table must be read: $\mathbf{x}||\mathbf{y}$ and $\mathbf{x}\star||\mathbf{y}$.
These sequences are called *Evil Twins* in the Prompts have evil twins paper.

———

OUR GOAL? Prevent their existence, or limit the success rate of this attack.

Difficulties of LLMs

What about LLMs?

- Input is **sequential**
- The same sequence can continue in multiple ways $\rightarrow$ **multiple** valid classes
- The input space is **discrete** ($|\mathcal{V}|$)

$\rightarrow$ a single token cannot determine what's the next token to predict

Introducing ILM

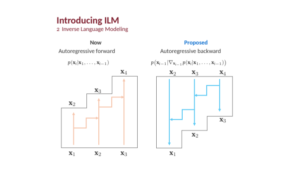
- **Goal:** train LLMs to both generate text and *understand what they are conditioned on* **(inversion)** from the output
- **Key Ideas:**
  - Create a new training procedure that adds more **robustness** in the loop
  - Reconstruct input from the output, using $\nabla_{\mathbf{x}}\mathcal{L}$

At this point, we can introduce Inverse Language Modeling.

GOAL: train LLMs, or fine-tune them, such that they internally ”understand” what they are conditioned on.

This is somehow based on the idea of LLMs as stochastic parrots.

KEY IDEAS: create a new training procedure that makes them **grounded** to the input, exploiting weights.



This illustration graphically shows the logic:

- originally, they go from left to right
- but it can also go from right to left, using gradients information.

ILM Inversion Procedure

Split it into the original prefix $\mathbf{x}_p = \mathbf{x}_{0:k}$ and the suffix $\mathbf{x}_s = \mathbf{x}_{k:n}$

$\mathbf{x} =$ The pen is on the table

$\mathbf{x}_p =$ **The pen is** $\quad$ $\mathbf{x}_s =$ on the table

Let's make an example:

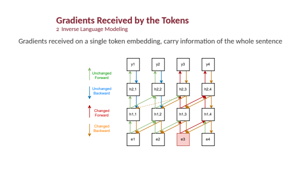
We have a sentence, like *The pen is on the table*

It gets split:

- prefix: *the pen is*
- suffix: *on the table*

.

Here, we have the suffix and predict backward the prefix.

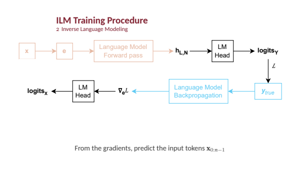


But how is that possible? What's the **theoretical** rationale behind it?
From this diagram, you can see that if we change a token in the middle, like $\mathbf{e}_3$,
it influences the hidden states only in the future,
but gradients carry out the information of the overall sentence,
since the gradients of the previous tokens (the **past**) change as well.

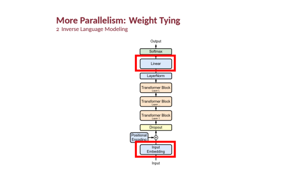
Gradients Received by the Tokens

A causal model looks ahead,
but only its gradients disclose the pasts that might have built that future.

This sentence well describes the rationale.



Use the gradients as if they were the last hidden state and use them to predict the input $\mathbf{x}$ tokens



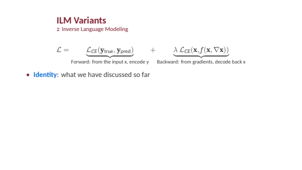
In some LLMs, weight tying makes the LM Head projection and the Embeddings matrix to be exactly the same Tensor in memory!

ILM Variants

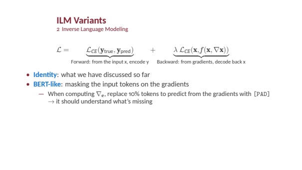
2 Inverse Language Modeling

$$\mathcal{L} = \underbrace{\mathcal{L}_{CE}(\mathbf{y}_{\text{true}}, \mathbf{y}_{\text{pred}})}_{\text{Forward: from the input x, encode y}} + \underbrace{\lambda\, \mathcal{L}_{CE}(\mathbf{x}, f(\mathbf{x}, \nabla \mathbf{x}))}_{\text{Backward: from gradients, decode back x}}$$

We end up with this combined loss, both for Cross-Entropy forward and backward.
This is implemented using PyTorch-supported **double Backpropagation**.
——— PAUSE ———
We have some variants:
- identity, what we just said. It might hypothetically learn some identity function, as in AutoEncoders without a bottleneck
- bert-like, imitating the BERT training procedure when going backward on the Gradients
- inv-first, that just works on the very first token, splitting sentences.
——— PAUSE ———
Classification:
- we can use these gradients as a pure value
- or follow the natural definition of a gradient as a direction and go in its negative direction



We end up with this combined loss, both for Cross-Entropy forward and backward.
This is implemented using PyTorch-supported **double Backpropagation**.
——— PAUSE ———
We have some variants:
- identity, what we just said. It might hypothetically learn some identity function, as in AutoEncoders without a bottleneck
- bert-like, imitating the BERT training procedure when going backward on the Gradients
- inv-first, that just works on the very first token, splitting sentences.
——— PAUSE ———
Classification:
- we can use these gradients as a pure value
- or follow the natural definition of a gradient as a direction and go in its negative direction



We end up with this combined loss, both for Cross-Entropy forward and backward.
This is implemented using PyTorch-supported **double Backpropagation**.
——— PAUSE ———
We have some variants:
- identity, what we just said. It might hypothetically learn some identity function, as in AutoEncoders without a bottleneck
- bert-like, imitating the BERT training procedure when going backward on the Gradients
- inv-first, that just works on the very first token, splitting sentences.
——— PAUSE ———
Classification:
- we can use these gradients as a pure value
- or follow the natural definition of a gradient as a direction and go in its negative direction

ILM Variants

$$\mathcal{L} = \underbrace{\mathcal{L}_{CE}(\mathbf{y}_{\text{true}}, \mathbf{y}_{\text{pred}})}_{\text{Forward: from the input x, encode y}} + \underbrace{\lambda\, \mathcal{L}_{CE}(\mathbf{x}, f(\mathbf{x}, \nabla\mathbf{x}))}_{\text{Backward: from gradients, decode back x}}$$

- **Identity**: what we have discussed so far
- **BERT-like**: masking the input tokens on the gradients
  - When computing $\nabla_{\mathbf{e}}$, replace 10% tokens to predict from the gradients with `[PAD]` → it should understand what's missing
- **Inv-First**: assign the first token to `[PAD]` and invert it

We end up with this combined loss, both for Cross-Entropy forward and backward.
This is implemented using PyTorch-supported **double Backpropagation**.
——— PAUSE ———
We have some variants:
- identity, what we just said. It might hypothetically learn some identity function, as in AutoEncoders without a bottleneck
- bert-like, imitating the BERT training procedure when going backward on the Gradients
- inv-first, that just works on the very first token, splitting sentences.
——— PAUSE ———
Classification:
- we can use these gradients as a pure value
- or follow the natural definition of a gradient as a direction and go in its negative direction

ILM Variants

$$\mathcal{L} = \underbrace{\mathcal{L}_{CE}(\mathbf{y}_{\text{true}}, \mathbf{y}_{\text{pred}})}_{\text{Forward: from the input x, encode y}} + \underbrace{\lambda\, \mathcal{L}_{CE}(\mathbf{x}, f(\mathbf{x}, \nabla \mathbf{x}))}_{\text{Backward: from gradients, decode back x}}$$

- **Identity**: what we have discussed so far
- **BERT-like**: masking the input tokens on the gradients
  - When computing $\nabla_{\mathbf{e}}$, replace 10% tokens to predict from the gradients with `[PAD]` $\rightarrow$ it should understand what's missing
- **Inv-First**: assign the first token to `[PAD]` and invert it

Classification Stategies:

We end up with this combined loss, both for Cross-Entropy forward and backward.
This is implemented using PyTorch-supported **double Backpropagation**.
——— PAUSE ———
We have some variants:
- identity, what we just said. It might hypothetically learn some identity function, as in AutoEncoders without a bottleneck
- bert-like, imitating the BERT training procedure when going backward on the Gradients
- inv-first, that just works on the very first token, splitting sentences.
——— PAUSE ———
Classification:
- we can use these gradients as a pure value
- or follow the natural definition of a gradient as a direction and go in its negative direction

ILM Variants

2 Inverse Language Modeling

$$\mathcal{L} = \underbrace{\mathcal{L}_{CE}(\mathbf{y}_{\text{true}}, \mathbf{y}_{\text{pred}})}_{\text{Forward: from the input x, encode y}} + \underbrace{\lambda\, \mathcal{L}_{CE}(\mathbf{x}, f(\mathbf{x}, \nabla \mathbf{x}))}_{\text{Backward: from gradients, decode back x}}$$

- **Identity**: what we have discussed so far
- **BERT-like**: masking the input tokens on the gradients
  - When computing $\nabla_{\mathbf{e}}$, replace 10% tokens to predict from the gradients with `[PAD]` $\rightarrow$ it should understand what's missing
- **Inv-First**: assign the first token to `[PAD]` and invert it

Classification Stategies:

- Use gradient as **value** $- f(\nabla_{\mathbf{x}_i} \mathcal{L}_{CE})$

We end up with this combined loss, both for Cross-Entropy forward and backward.
This is implemented using PyTorch-supported **double Backpropagation**.
——— PAUSE ———
We have some variants:
- identity, what we just said. It might hypothetically learn some identity function, as in AutoEncoders without a bottleneck
- bert-like, imitating the BERT training procedure when going backward on the Gradients
- inv-first, that just works on the very first token, splitting sentences.
——— PAUSE ———
Classification:
- we can use these gradients as a pure value
- or follow the natural definition of a gradient as a direction and go in its negative direction

ILM Variants

2 Inverse Language Modeling

$$\mathcal{L} = \underbrace{\mathcal{L}_{CE}(\mathbf{y}_{\text{true}}, \mathbf{y}_{\text{pred}})}_{\text{Forward: from the input x, encode y}} + \underbrace{\lambda\, \mathcal{L}_{CE}(\mathbf{x}, f(\mathbf{x}, \nabla \mathbf{x}))}_{\text{Backward: from gradients, decode back x}}$$

- **Identity**: what we have discussed so far
- **BERT-like**: masking the input tokens on the gradients
  - When computing $\nabla_{\mathbf{e}}$, replace 10% tokens to predict from the gradients with `[PAD]` $\rightarrow$ it should understand what's missing
- **Inv-First**: assign the first token to `[PAD]` and invert it

Classification Stategies:

- Use gradient as **value** — $f(\nabla_{\mathbf{x}_i} \mathcal{L}_{CE})$
- Use gradient as **direction** — $f(\mathbf{x}_i - \nabla_{\mathbf{x}_i} \mathcal{L}_{CE})$

We end up with this combined loss, both for Cross-Entropy forward and backward.
This is implemented using PyTorch-supported **double Backpropagation**.
——— PAUSE ———
We have some variants:
- identity, what we just said. It might hypothetically learn some identity function, as in AutoEncoders without a bottleneck
- bert-like, imitating the BERT training procedure when going backward on the Gradients
- inv-first, that just works on the very first token, splitting sentences.
——— PAUSE ———
Classification:
- we can use these gradients as a pure value
- or follow the natural definition of a gradient as a direction and go in its negative direction

But we don’t have billions of dollars

These results have been obtained on a tiny LLM:

- Only 10M parameters
- A vocabulary of just 2048 tokens
- A simple corpus (`TinyStories` dataset)

It will be scaled to Llama-1B in the future.

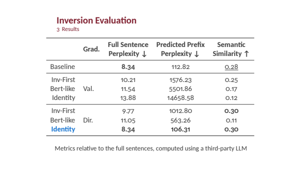
However, we trained LLMs, with lots of different variants to compare.
We HAD to stay on a small example, to validate the idea,
scaling it in a future time.

Inversion Evaluation
3 Results

	Grad.	Token Recall ↑	Token Precision ↑	Token F1-score ↑	Positional Accuracy ↑
Baseline		20.9%	18.8%	19.7%	2.4%
Inv-First	Val.	11.3%	10.1%	10.7%	1.7%
Bert-like	Val.	2.9%	2.7%	2.8%	0.3%
Identity	Val.	0.7%	0.7%	0.7%	0.1%
Inv-First	Dir.	13.3%	12.0%	12.6%	2.4%
Bert-like	Dir.	0.1%	0.1%	0.1%	0.1%
Identity	Dir.	**22.5%**	**20.2%**	**21.2%**	**2.5%**

Evaluation of the inversion capabilities, on metrics relative to the single tokens

In all these evaluation tables, we can see that the **Identity** model using gradients as **directions** is chosen as the best variant.
Interestingly, the **baseline** is already able to invert quite well, even though this method allowed us to further improve it.
NOTE that to invert we need an **init**:
- for baseline and identity, we use a very simple bigram model
- for bert and inv-first, we use the PAD token as did during training.



Inversion Evaluation
3 Results

	Grad.	Full Sentence Perplexity ↓	Predicted Prefix Perplexity ↓	Semantic Similarity ↑
Baseline		**8.34**	112.82	0.28
Inv-First	Val.	10.21	1576.23	0.25
Bert-like	Val.	11.54	5501.86	0.17
Identity	Val.	13.88	14658.58	0.12
Inv-First	Dir.	9.77	1012.80	**0.30**
Bert-like	Dir.	11.05	563.26	0.11
Identity	Dir.	**8.34**	**106.31**	**0.30**

Metrics relative to the full sentences, computed using a third-party LLM

To have more accurate results, we passed the sentences to a third-party LLM *Llama 1B*, to compute some perplexity statistics.
It shows:
- PPL of the overall sentence $\mathbf{x}\star||\mathbf{y}$
- PPL of just the inverted prefix $\mathbf{x}\star$

Example of Inversion

3 Results

$\mathbf{x}$		dad in the garden. He gives her a small shovel and a bag of bulbs.
$\mathbf{x}_\star$ Baseline		**to play with his cars, and look at the shake. She feels on her hand.**
$\mathbf{x}_\star$ Inv-First	(Val.)	zzle spowerlizza in her plate. She start to fence and leaves.
$\mathbf{x}_\star$ Bert-like	(Val.)	could buildDven measure its neighbign, how he sees nostiff.
$\mathbf{x}_\star$ Identity	(Val.)	Kugct propide,RallashQilndmawkeycessUuhingask do.
$\mathbf{x}_\star$ Inv-First	(Dir.)	too hurt the car's bricket. It did not want to grow in a cage.
$\mathbf{x}_\star$ Bert-like	(Dir.)	Tim! Tim,ide, Sue, Sue, Tim!ide, "Tim, "Tim,ice. Tim! Tim!ittenbbed Tim! Tim,ide,auseectle.
$\mathbf{x}_\star$ **Identity**	(Dir.)	**cars, and gets on his hand. But he does not want to play with the towers.**
$\mathbf{y}$		Bulbs are like round seeds that grow into flowers. Lily digs holes in the dirt and puts the bulbs inside. She covers them with more [...]

You can see some examples, where 2 make sense and the others are just gibberish.

This table must be read as:

- ground truth = $\mathbf{x}||\mathbf{y}$
- prediction of a model = $\mathbf{x}\star||\mathbf{y}$

ILM Robustness Results

	Grad.	GCG Success Rate ↓	GCG Average Steps (mean ± stddev)
Baseline		95.9%	277 ± 148
Inv-First	Val.	85.0%	320 ± 134
Bert-like	Val.	**0.8%**	249 ± 148
Identity	Val.	88.1%	274 ± 145
Inv-First	Dir.	89.3%	313 ± 134
Bert-like	Dir.	85.5%	287 ± 143
Identity	Dir.	82.8%	284 ± 141

Identity looks good, but Bert-like is suspicious

THEN, we have the **twofold objective** = ROBUSTNESS.
Here, still the **Identity model** is the best model, reducing the Attack Success Rate by 13%.
Then, we see the **Bert-like** model with gradients as pure values to be extraordinarily successful, but it requires more research to correctly understand the why.

ILM Robustness — Metrics on the Model Itself

	Grad.	Original X CE-loss ↓	Attack X' CE-loss	Delta CE-loss ↓	KL Divergence ↑
Baseline		13.28	10.97	2.31	2.19
Inv-First	Val.	**11.09**	9.72	1.37	2.44
Bert-like	Val.	13.26	10.25	3.01	**54.19**
Identity	Val.	12.77	11.21	1.56	2.23
Inv-First	Dir.	11.21	9.81	1.40	2.44
Bert-like	Dir.	11.49	10.34	**1.15**	2.23
Identity	Dir.	12.58	11.12	1.46	2.47

Also, Bert-like seems to map $\mathbf{x}_\star$ to very different next token distributions

We also see the decrease in **loss** when the attack is successful.
- the higher this DELTA is, the more ”fooled” the model has been by the Evil Twin

$\rightarrow$ that's why a lower DELTA is better.
- the KL Divergence indicates that the $\mathbf{x}$ and $\mathbf{x}_\star$ map to different output distributions of the logits, like if the model can map it to different distributions, therefore different **internal hidden states**.

ILM Robustness — Third-Party Model Metrics

	Grad.	Original X Perplexity	Attack X' Perplexity ↓	Semantic Similarity ↑
Baseline		44.14	17344.04	0.13
Inv-First	Val.	44.81	9431.09	0.16
Bert-like	Val.	40.37	11817.21	0.11
Identity	Val.	43.98	**8322.25**	**0.18**
Inv-First	Dir.	43.50	12344.85	0.13
Bert-like	Dir.	44.74	10611.09	0.13
Identity	Dir.	44.71	10929.21	0.15

However, all $\mathbf{x}_\star$ are **meaningless**, due to extremely high perplexity

To conclude, we have the third-party LLM measurements,
where we basically see that the $\mathbf{x}_\star$, **when successfully found**,
still is gibberish and absolutely not similar with the original X.
HOWEVER, who knows if this may improve in larger models such as Llama,
future research will address the scaling problem.

ILM Robustness — Qualitative Results

	Input	Output y	Loss
$\mathbf{x}$:	Lily and Ben were friends who liked to play outside. But they did not like the same things. Lily	liked to make snowmen and snow angel	13.22
$\mathbf{x}_{\star}$:	Lucy. Speez herself angO piecle you."lly named nexird opened cake".o.ter carrotmy		**12.14**

An example result attacking with GCG the `Identity (grad. value)` model. Almost the same for all model variants.

Here we can see an example to show that the $\mathbf{x}_{\star}$ attack prefix is still gibberish