Transformers learn in-context by gradient descent

Johannes von Oswald

12.12.2022

ETH zürich
Google Research

go/in-context-descent

Many many thanks to all the people involved in this project!

Eyvind Niklasson

Ettore Randazzo

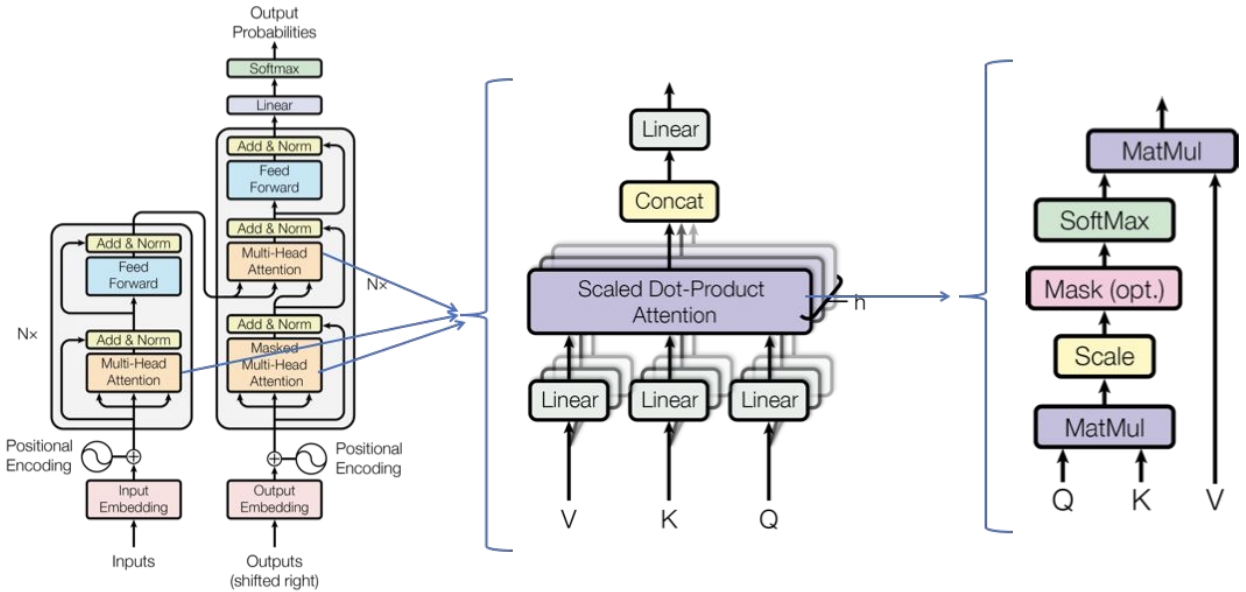
João Sacramento

Alexander Mordvintsev

Andrey Zhmoginov

Max Vladymyrov

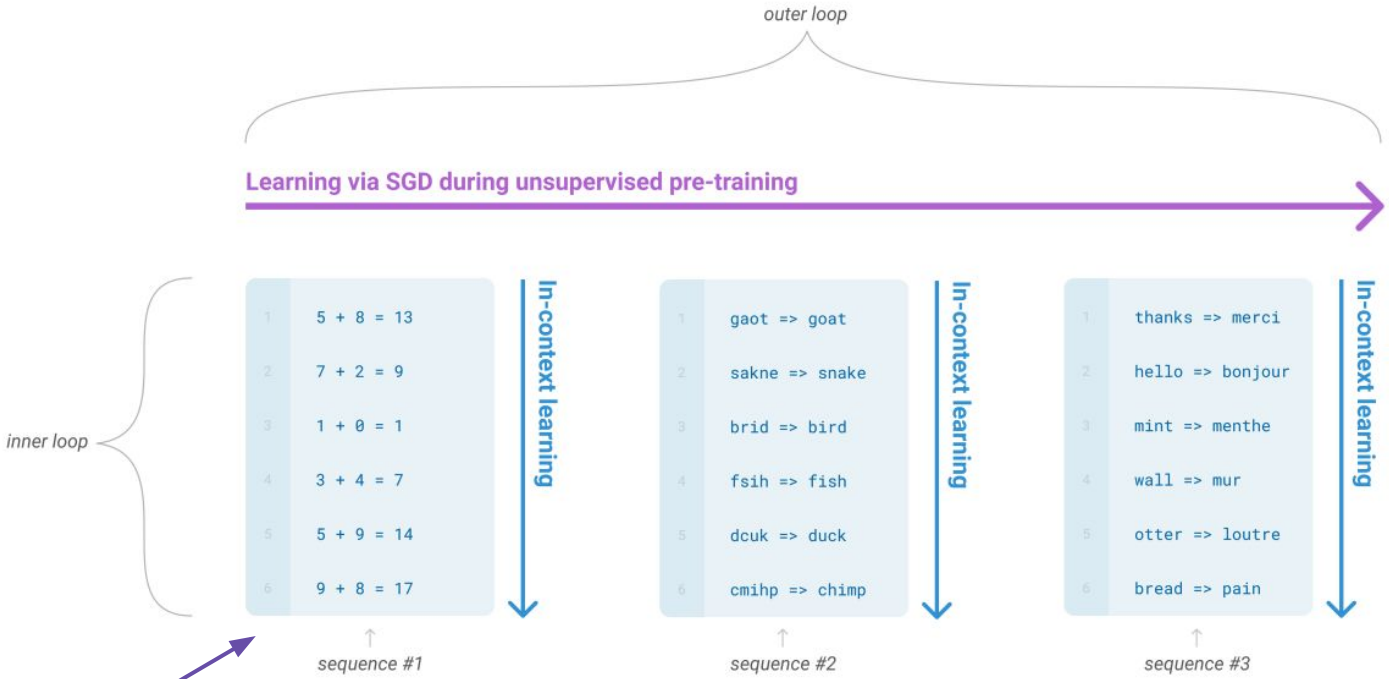
Transformers



… are all you need! ***The*** neural network architecture used in modern AI.

Image Source

GPT-3: Language Models are Few-shot learners



In this talk: What happens here?

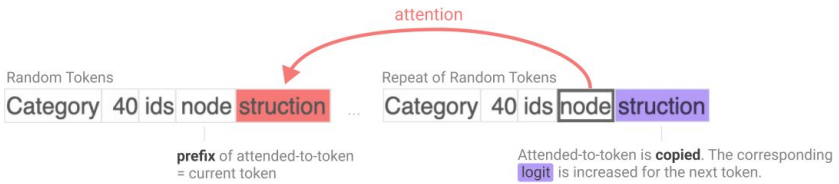
[Language Models are Few-Shot Learners, 2020]

In-context learning by *induction* heads

“The **first attention head** copies information from the previous token into each token.

This makes it possible for the **second attention head** (which we call the "induction head") to attend to tokens based on what happened before them, rather than their own content.

That is, the **two heads working together** and cause the sequence …[A][B]…[A] to be more likely to be completed with [B].”



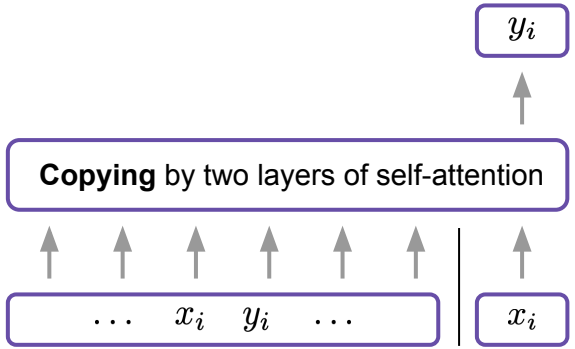
In-context Learning and Induction Heads, 2022

In-context learning by *induction heads*

“The **first attention head** copies information from the previous token into each token.

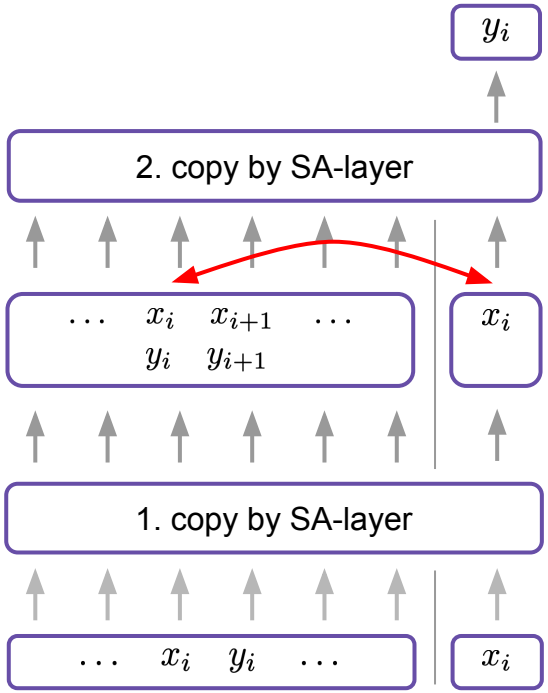
This makes it possible for the **second attention head** (which we call the "induction head") to attend to tokens based on what happened before them, rather than their own content.

That is, the **two heads working together** and cause the sequence …[A][B]…[A] to be more likely to be completed with [B].”



In-context Learning and Induction Heads, 2022

In-context learning by *induction heads*



Summary:

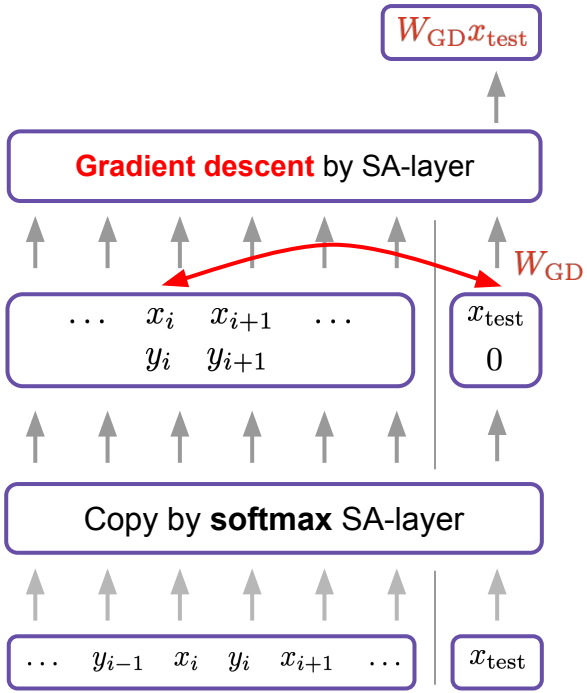
Step 1: First attention layer copies y_i to x_i token

Step 2: Second attention layer x_i query attends to itself by **similarity** and copies over y_i .

Olsson et. al identify induction heads as being important for in-context learning for all model sizes.

In-context Learning and Induction Heads, 2022

This work: In-context learning by *gradient descent*



Summary:

Step 1: First attention layer copy y_i to x_i token

Step 2: Second attention layer **learns linear model** on training data $\{(x_i, y_i)\}_{i=1}^N$ and predict on test (or training) data.

We are not copying but can generalise to unseen data.

Lowering expectations: We will in the following only look at toy datasets and restrict ourselves (mostly) to self-attention only architectures but try to understand them thoroughly.

Take home message

1) **A single layer of self-attention can implement a single step of gradient descent.** We provide a simple weight construction that implements it.

2) **Trained Transformers match our construction** or models generated are remarkably similar to models obtained by GD.

We hypothesize that:

When training Transformers on auto-regressive tasks, in-context learning in the Transformer forward pass is implemented by gradient-based optimization.

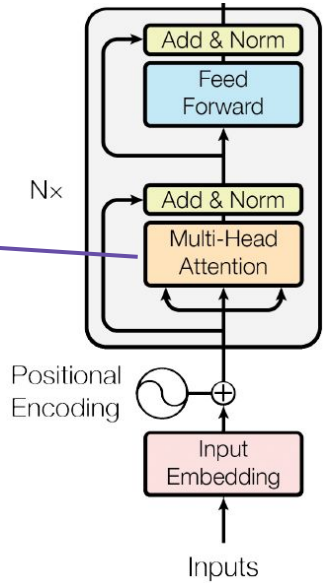
Self-attention recap

Softmax self-attention (SA) layer

$$e_j \leftarrow e_j + \sum_h P_h V_h \text{softmax}(K_h^T q_{h,j})$$

with e.g.

$$V_h = [v_{h,1}, \ldots, v_{h,N}]$$
$$v_{h,i} = W_{h,V} e_i$$



Linear self-attention (LSA) layer

$$e_j \leftarrow e_j + \sum_h P_h V_h K_h^T q_{h,j} = e_j + \sum_h P_h \sum_i v_{h,i} \otimes k_{h,i} q_{h,j}$$
$$= e_j + \sum_h P_h W_{h,V} \sum_i e_{h,i} \otimes e_{h,i} W_{h,K}^T W_{h,Q} e_j$$

This already looks quite similar to gradient descent, no?

Image Source

Recap: Gradient descent on linear regression

Assume: $W_{\text{init}} = 0$ and therefore the initial prediction $\hat{y}_{\text{test}} = 0$

For n training steps do:

1. Compute regression loss:
   $L(W, \{(x_i, y_i)\}_{i=1}^N) = \frac{1}{2N} \sum_{i=1}^{N} (Wx_i - y_i)^2$
2. Gradient descent:
   $\Delta W = -\eta \nabla_W L(W, \{(x_i, y_i)\}_{i=1}^N)$
3. Update weights:
   $W \leftarrow W + \Delta W$

Make predictions:

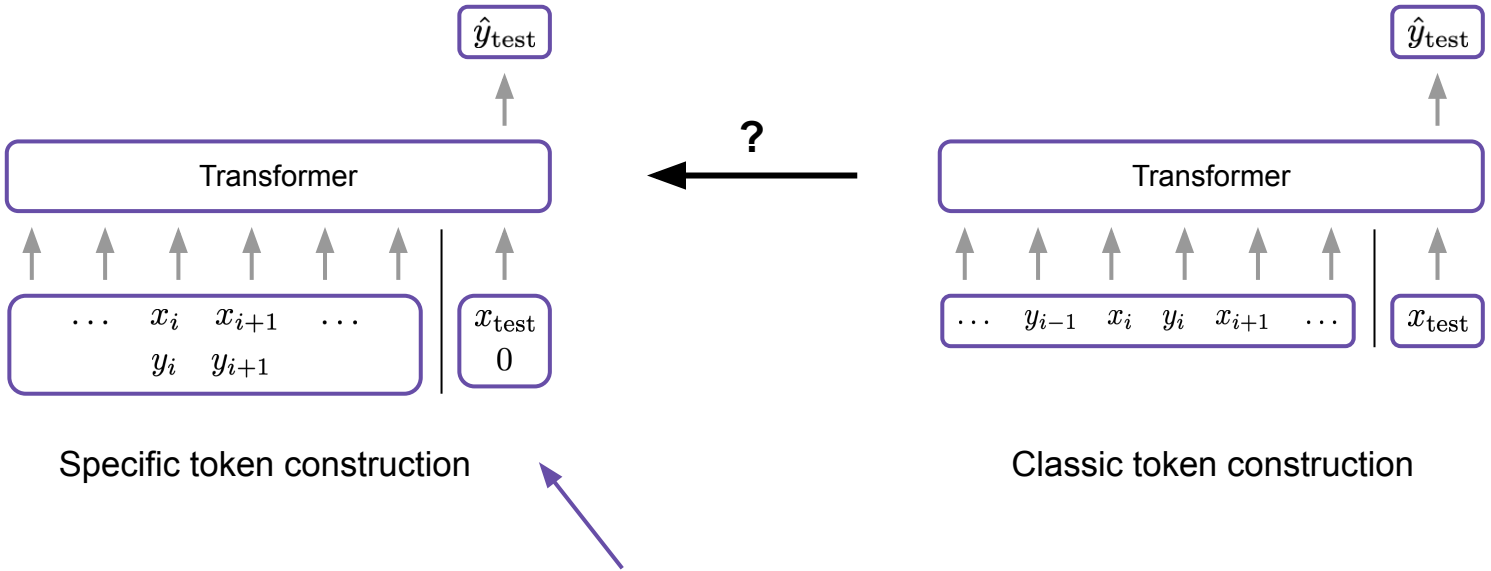
$\hat{y}_{\text{test}} \leftarrow W_{\text{final}} x_{\text{test}} = \sum \Delta W x_{\text{test}}$

1. Compute regression loss:
   $L(W, \{(x_i, y_i)\}_{i=1}^N) = \frac{1}{2N} \sum_{i=1}^{N} (Wx_i - y_i)^2$
2. Gradient descent:
   $\Delta W = -\eta \nabla_W L(W, \{(x_i, y_i)\}_{i=1}^N)$
3. Update targets:
   $L(W + \Delta W, \{(x_i, y_i)\}_i^N) = L(W, \{(x_i, y_i - \Delta W x_i)\}_i^N)$
   and predictions with the same rule:
   $\hat{y}_{\text{test}} \leftarrow \hat{y} - \Delta W x_{\text{test}}$

$\hat{y}_{\text{test}} \leftarrow -1 \cdot \hat{y}_{\text{test}} = -1 \cdot \sum -\Delta W x_{\text{test}}$

Equivalent gradient descent formulation that **transforms the data** instead of the model!

A specific token construction



A single LSA-layer can implement a step of GD

Proposition 1: *Given a 1-head linear attention layer and the tokens* $e_j = (x_j, y_j)$, *one can construct key, query and value matrices* W_K, W_Q, W_V *as well as the projection matrix* P *such that a Transformer step on every token* e_j *is identical to the gradient-induced dynamics* $e_j \leftarrow (x_j, y_j) + (0, -\Delta W x_j) = (x_i, y_i) + PVK^T q_j$ *such that* $e_j = (x_j, \tilde{y}_j)$. *For the test data token* (x_{N+1}, y_{N+1}) *the dynamics are identical.*

We provide the weight matrices in block form: $W_K = W_Q = \begin{pmatrix} 0 & 0 \\ I_x & 0 \end{pmatrix}$ with I_x and I_y the identity matrices of size N_x and N_y respectively. Furthermore, we set $W_V = \begin{pmatrix} 0 & 0 \\ W & -I_y \end{pmatrix}$ with the weight matrix $W \in \mathbb{R}^{N_y \times N_x}$ of the linear model we wish to train and $P = \frac{\eta}{N} I$ with identity matrix of size $N_x + N_y$. With this simple construction we obtain the following dynamics

$$\begin{pmatrix} x_j \\ y_j \end{pmatrix} \leftarrow \begin{pmatrix} x_j \\ y_j \end{pmatrix} + \frac{\eta}{N} I \sum_{i=1}^{N} \left(\begin{pmatrix} 0 & 0 \\ W & -I_y \end{pmatrix} \begin{pmatrix} x_i \\ y_i \end{pmatrix} \right) \otimes \left(\begin{pmatrix} 0 & 0 \\ I_x & 0 \end{pmatrix} \begin{pmatrix} x_i \\ y_i \end{pmatrix} \right) \begin{pmatrix} 0 & 0 \\ I_x & 0 \end{pmatrix} \begin{pmatrix} x_j \\ y_j \end{pmatrix}$$

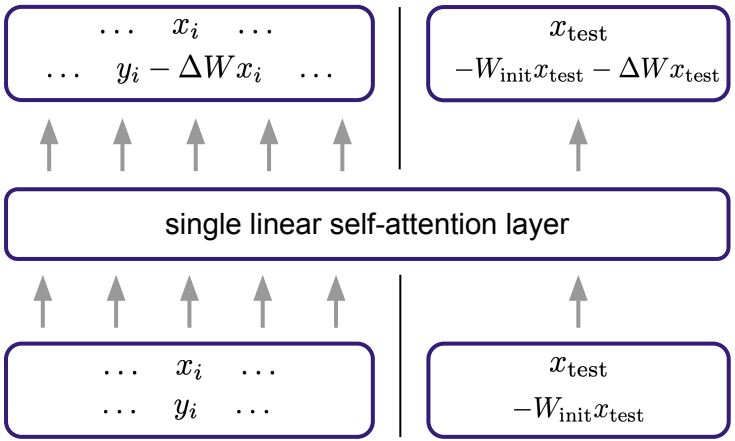
$$= \begin{pmatrix} x_j \\ y_j \end{pmatrix} + \frac{\eta}{N} I \sum_{i=1}^{N} \begin{pmatrix} 0 \\ W x_i - y_i \end{pmatrix} \otimes \begin{pmatrix} 0 \\ x_i \end{pmatrix} \begin{pmatrix} 0 \\ x_j \end{pmatrix} = \begin{pmatrix} x_j \\ y_j \end{pmatrix} + \begin{pmatrix} 0 \\ -\Delta W x_j \end{pmatrix}. \qquad (6)$$

P v_i k_i q_j

A single self-attention can implement a single gradient descent step on linear regression loss!

Visualization of Proposition 1

A single linear self attention layer can compute in its forward pass: $\hat{y}_{\text{test}} = W_{\text{init}} x_{\text{test}} + \Delta W x_{\text{test}}$

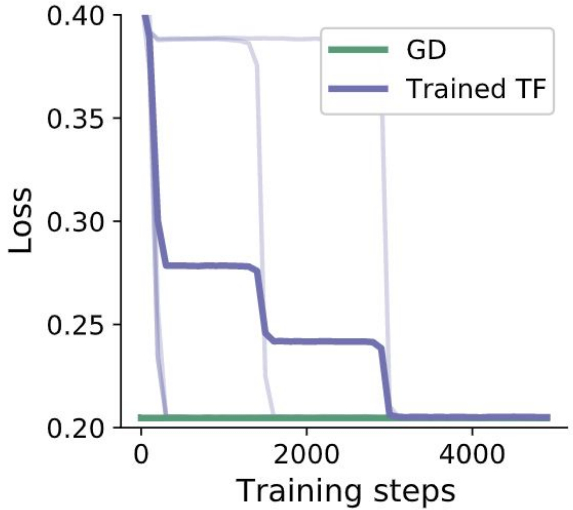


Gradient descent induced transformation of the training data.

1) Simultaneously predicting on test data
2) Correct prediction after multiplying with -1

A single linear self-attention layer can compute the gradient descent update on training as well as update the model prediction simultaneously.

Training a single *linear* self-attention layer



Training setup:

- **1 step** of gradient descent
- 10 datapoints with input dimension 10
- Each datapoint is sampled from x ~ U[-1, 1]
- Teacher is sampled from W ~ N(0,I)
- Train and test distributions are the same.
- Gradient descent learning rate is tuned on 10000 tasks

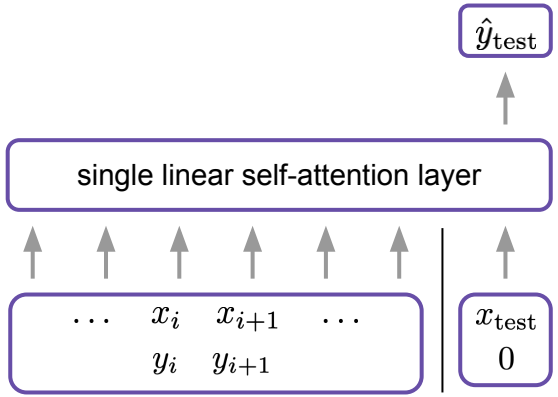
LSA-layer and GD reach the exact same loss.
But are 1) the models and/or 2) the learning algorithms the same?

Models generated by TFs vs GD

$$\hat{y}_{\mathrm{GD}} = W_{\mathrm{init}} x_{\mathrm{test}} + \Delta W x_{\mathrm{test}} \overset{W_{\mathrm{init}}=0}{=} \Delta W x_{\mathrm{test}}$$

vs

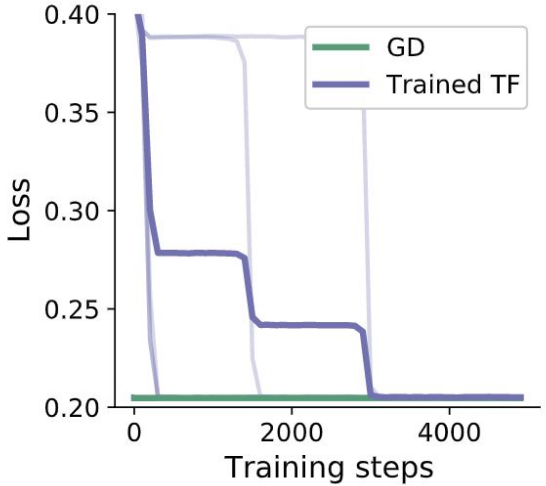
$$\hat{y}_{\mathrm{TF}} = \ ? \ + PVK^T W_q x_{\mathrm{test}} \overset{W_{\mathrm{init}}\approx 0}{=} \Delta W_{\mathrm{TF}} x_{\mathrm{test}}$$

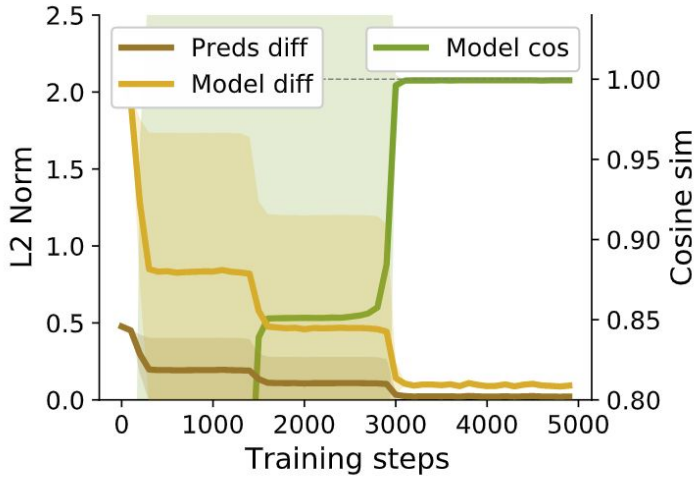


We compare the models by

- Differences in predictions: $||\hat{y}_{\mathrm{TF}} - \hat{y}_{\mathrm{GD}}||$
- Differences of models: $\left\| \frac{\partial \hat{y}_{\mathrm{TF}}}{\partial x_{\mathrm{test}}} - \frac{\partial \hat{y}_{\mathrm{GD}}}{\partial x_{\mathrm{test}}} \right\|$
- Cosine similarities of models: $\cos(\frac{\partial \hat{y}_{\mathrm{TF}}}{\partial x_{\mathrm{test}}}, \frac{\partial \hat{y}_{\mathrm{GD}}}{\partial x_{\mathrm{test}}})$

Trained Transformer vs GD





Models are remarkably similar!

Interpolating between trained TFs and our construction / GD

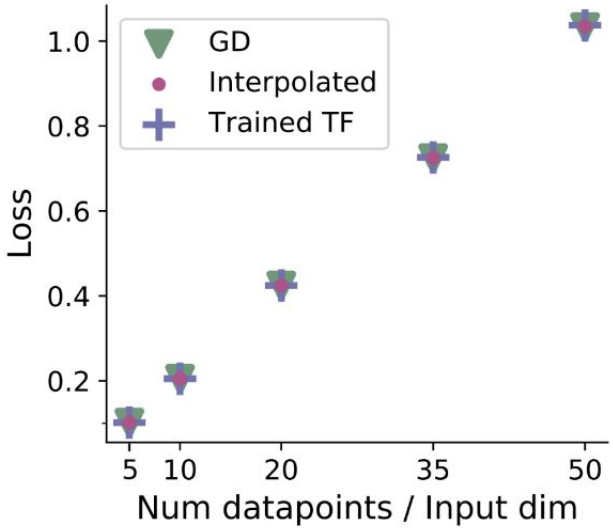
Linear self-attention update:

$$e_j \leftarrow e_j + PVK^T q_j = e_j + PW_V \sum_i e_i \otimes e_i W_K^T W_Q e_j$$

$$= e_j + W_{PV} \sum_i e_i \otimes e_i W_{KQ} e_j$$

GD update:

$$e_j \leftarrow e_j + PVK^T q_j = e_j + W_{PV} \sum_i e_i \otimes e_i W_{KQ} e_j$$

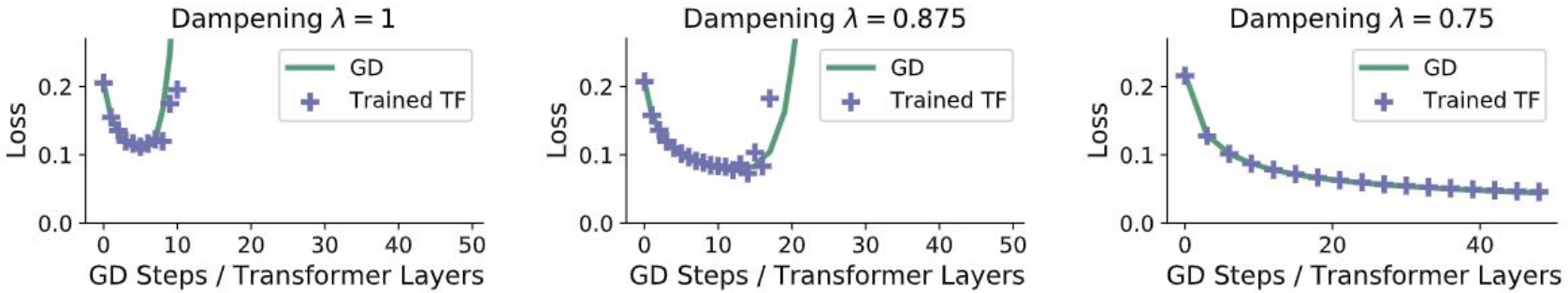


We can **interpolate** between our construction and the weights found!

What happens if we test the model on data it was not trained on?

Trained Transformer vs GD

Transformer was trained on one step - what if we roll it out for longer?

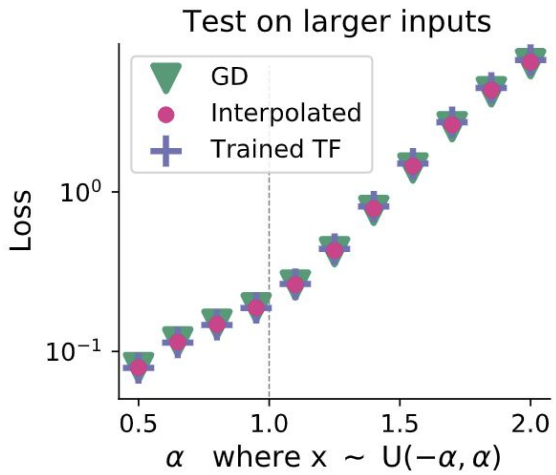


Dampening: $\hat{y}_{\mathrm{GD}} \leftarrow \hat{y}_{\mathrm{GD}} + \lambda \Delta W x_{\mathrm{test}}$ or $\hat{y}_{\mathrm{TF}} \leftarrow \hat{y}_{\mathrm{TF}} + \lambda \Delta P V K^T W_Q x_{\mathrm{test}}$

Transformer and GD show remarkable similarity on this out-of-training setup!

Trained Transformer vs GD

LSA-layer was trained on specific data distribution.
What if we change that during evaluation?



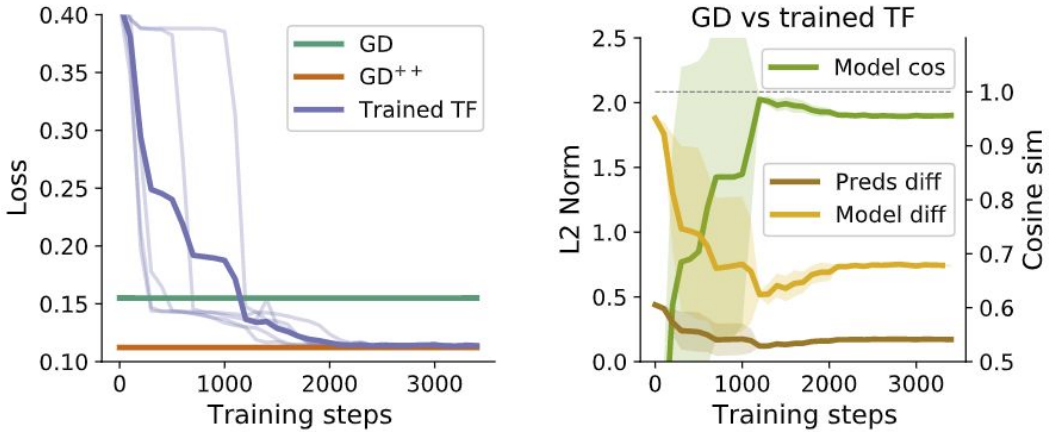
Transformer and GD behave remarkably similar when tested on different data distributions!

Recap

1) We can construct simple self-attention weights to mimic a single step of gradient descent on a linear regression loss

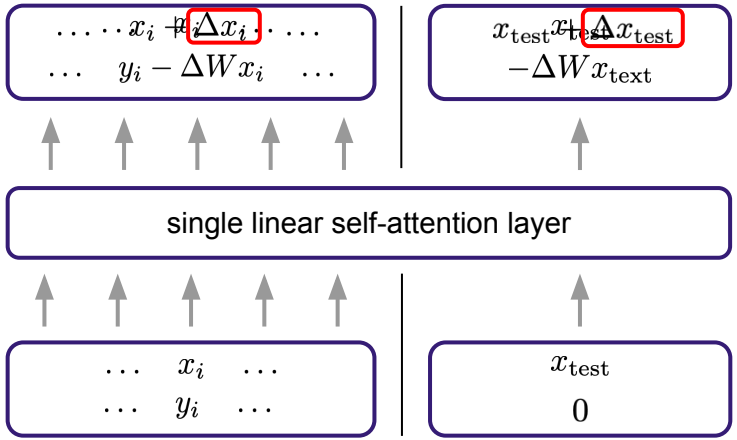
2) Optimizing a single self-attention layer to solve linear regression problems finds Transformer weights that match our construction and therefore implements GD.

Two layer LSA *recurrent* Transformer



Transformer surpasses GD at which point our alignment metrics start decreasing.
But what is GD++?

How can we improve upon plain gradient descent?



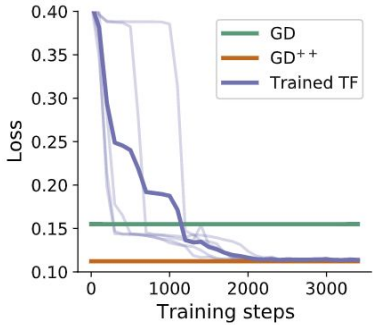
Can we find input transformations to improve upon plain gradient descent?

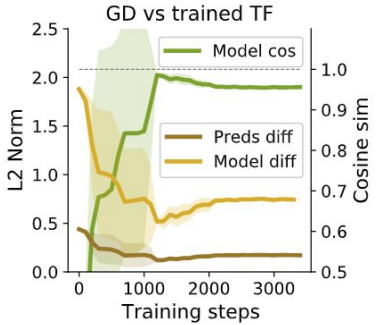
$$x_j \leftarrow x_j + \Delta x_j = (I + PV(X)K(X)^T W_q)x_j = H(X)x_j$$

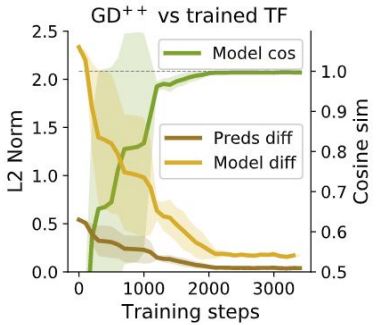
We found that the following transformation is enough to match and realign the to Transformer. We call this variant GD++: $H(X) = (I - \gamma XX^T)$

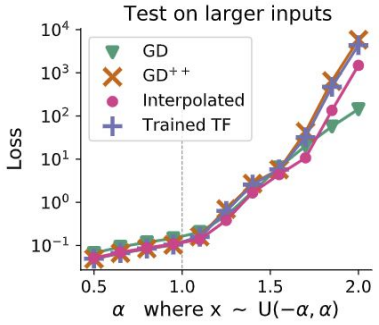
Crude approx of loss curvature inverse

Two layer *recurrent* LSA-Transformer



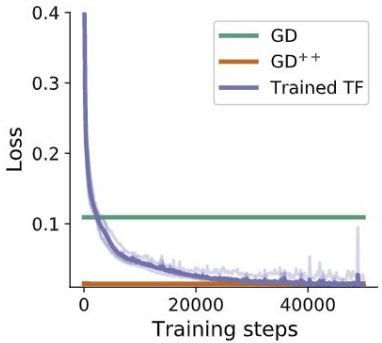


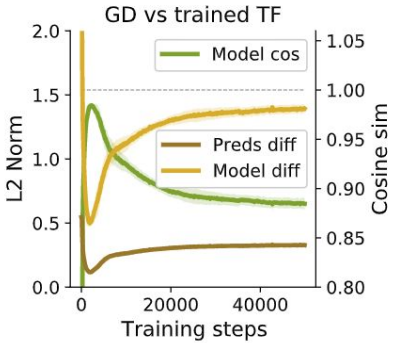


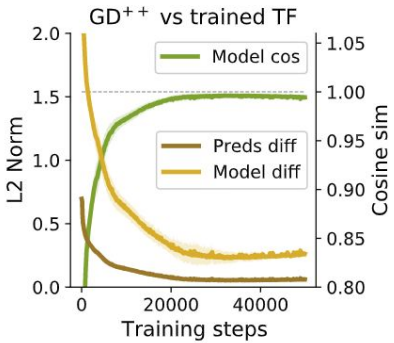


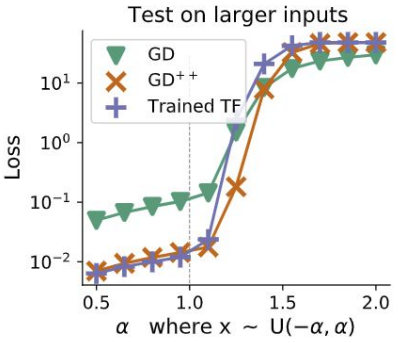
Recurrent Transformer and GD++ realign in all metrics and interpolate!

Five layer *non*-recurrent LSA Transformer









Non-recurrent Transformer and GD++ realign in all metrics, no interpolation (yet)!

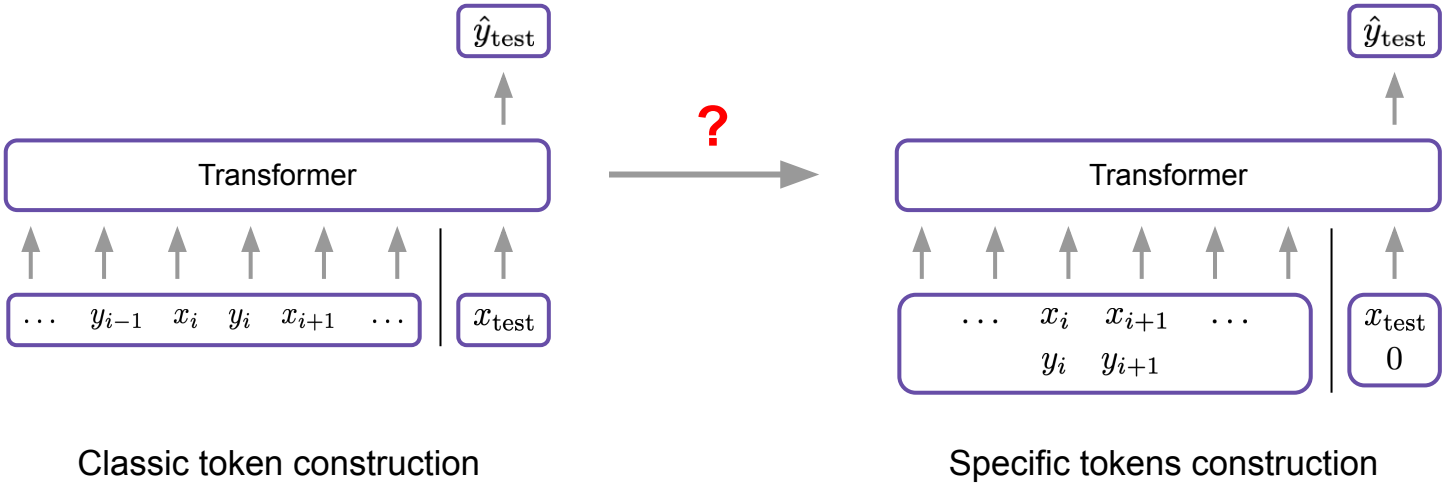
Recap

1) We can construct simple self-attention weights to mimic a single step of gradient descent on a linear regression loss

2) Optimizing a single self-attention layer to solve linear regression problems finds Transformer weights that match our construction and therefore implements GD.

3) Optimizing multiple layers of self-attention to solve a linear regression problem finds Transformer weights that behave remarkable similar to GD++, an accelerated GD version.

Do Transformers *build datasets* to do GD?



Classic token construction

Specific tokens construction

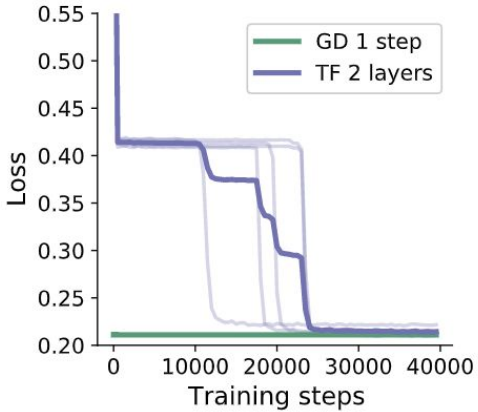
Do Transformers learn to create the specific token construction necessary for Proposition 1?

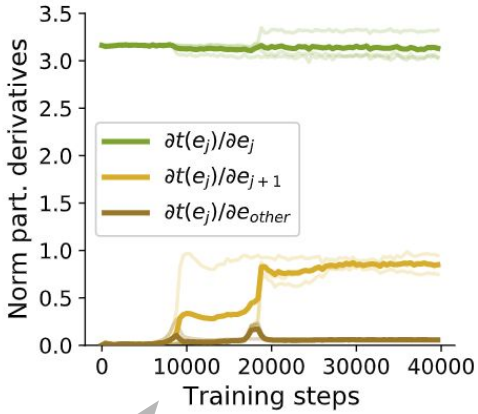
Do Transformers *build datasets* to do GD?

Proposition 2: *Given a 1-head linear or softmax attention layer and the token construction* $e_{2j} = (x_j), e_{2j+1} = (0, y_j)$ *with a zero vector* 0 *of dim* $N_x - N_y$ *and concatenated positional encodings, one can construct key, query and value matrix* W_K, W_Q, W_V *as well as the projection matrix* P *such that all tokens* e_j *are transformed into tokens equivalent to the ones required in Proposition 1.*

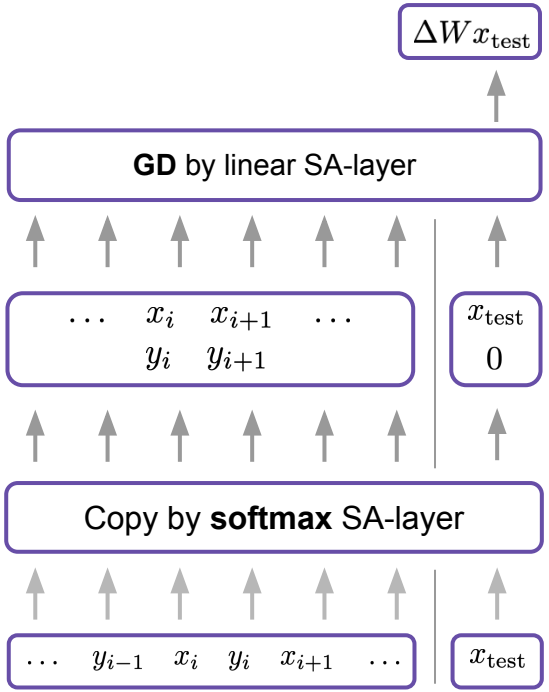
… one self-attention layer can copy over data and construct the token construction required for Proposition 1.

Do early self-attention layers build regression tasks?



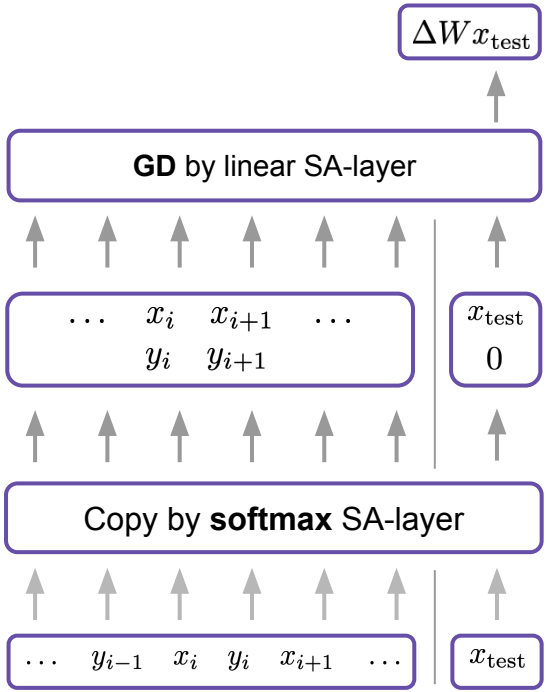


We compute the sensitivity of the output token wrt. the input tokens of the first layer.



The first attention layer copies token data to build linear regression loss and solves it.

Do early self-attention layers build regression tasks?



What I find fascinating: The Transformer creates its own dataset and solves it!

Does this also happen in larger language models?e

Recap

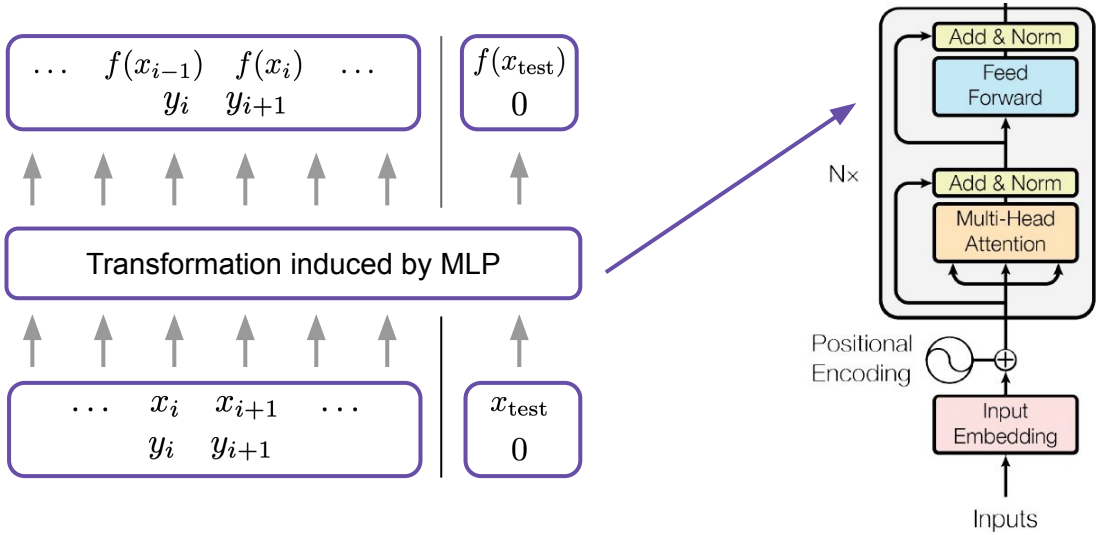
1) We can **construct** simple self-attention weights to mimic a single step of gradient descent on a linear regression loss

2) **Optimizing** multiple layers of self-attention to solve a linear regression problem finds Transformer weights that behave remarkable similar to GD++, an accelerated GD version.

3) As already shown in related work, in-context learning is linked to self-attention layers copying “in - and output data” into one token. This enables a second attention layer to predict by transformations induced by GD.

Thank you!

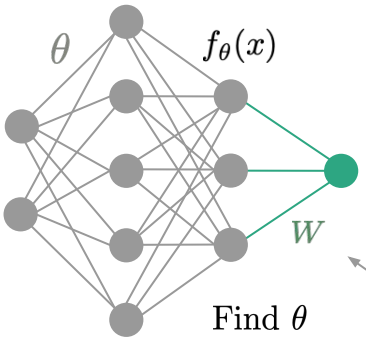
What about non-linear regression tasks?



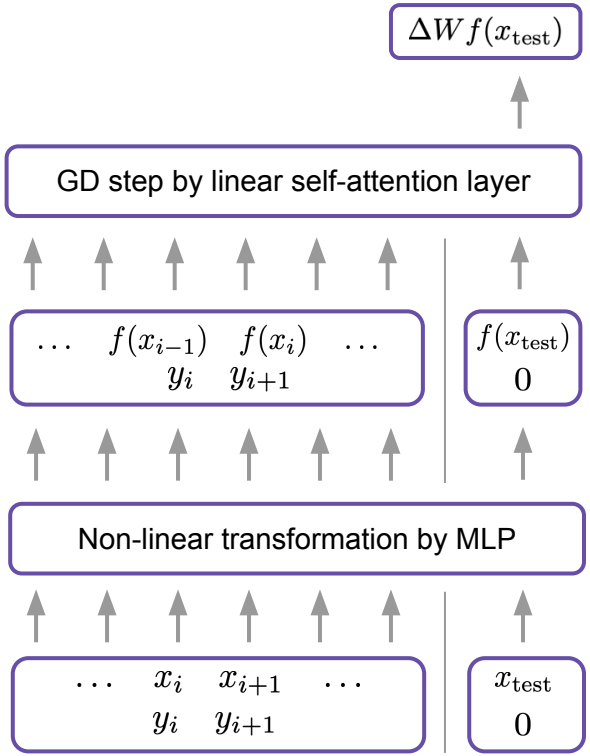
We are neglecting potentially useful transformations to the target.

Image Source

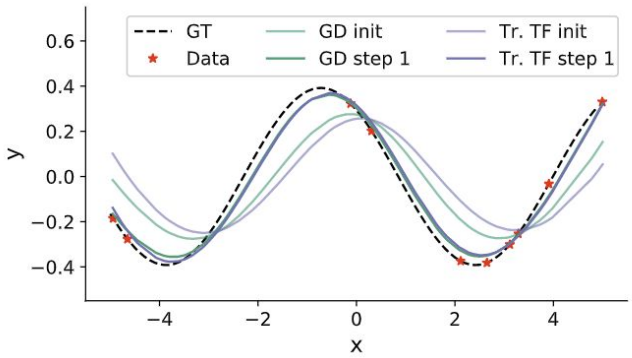
What about non-linear regression tasks?

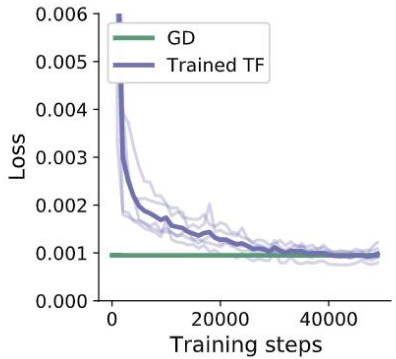


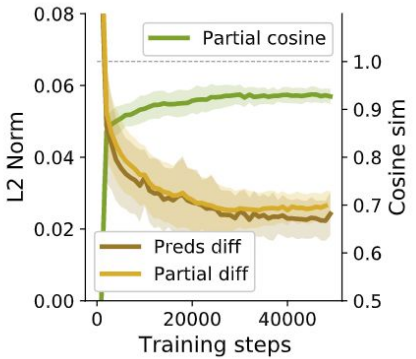
Find θ
s.t. $(W - \nabla_W L(\mathrm{D}^{\mathrm{train}}))f_\theta(x^{\mathrm{test}}) \approx y^{\mathrm{test}}$



The sine wave regression task





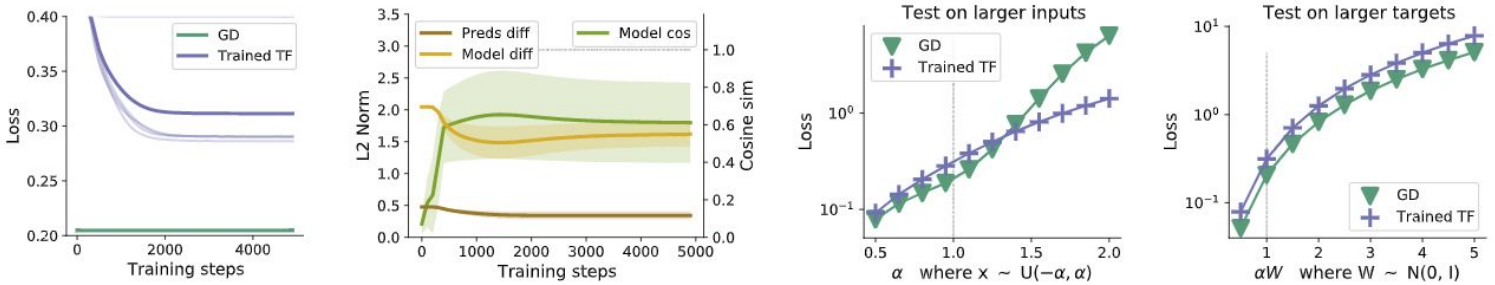


MLPs in the Transformer architecture enable

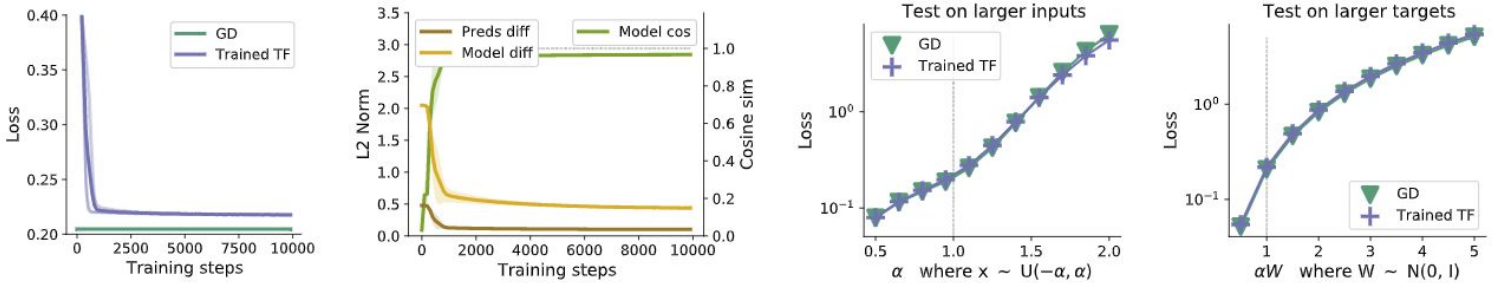
1) learning linear models on data representations and therefore
2) learning non-linear regression tasks.

What about softmax?

Comparing 1 step of gradient descent with a trained *softmax single* head self-attention layer.



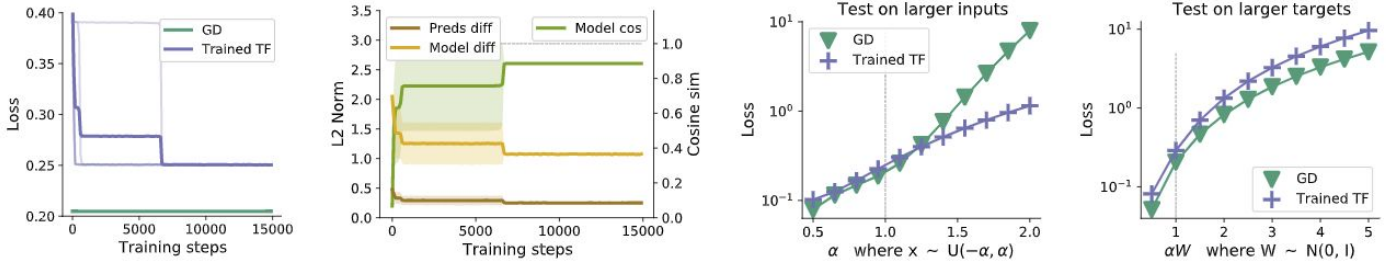
Comparing 1 step of gradient descent with a trained *softmax two* head self-attention layer.



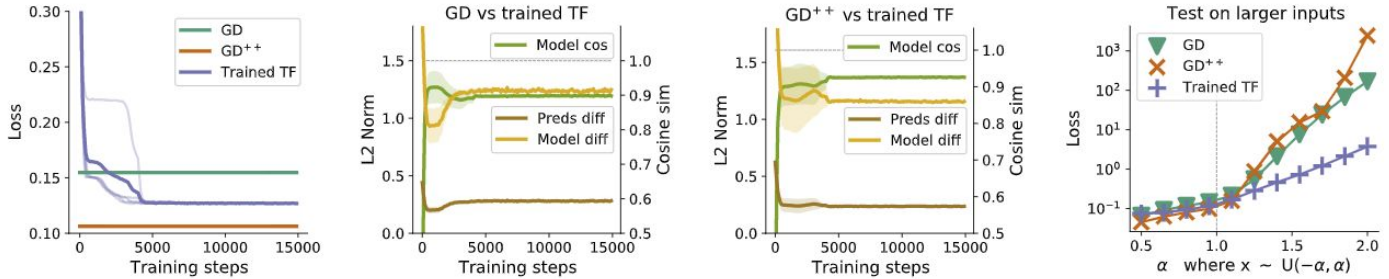
Softmax self-attention layer underperform on gradient descent but! are needed for copying.

What about LayerNorm?

Comparing 1 step of gradient descent with a single layer LSA Transformer with LayerNorm.



Comparing 2 steps of gradient descent with a two layer LSA Transformer with LayerNorm.



Self-attention + LayerNorm layer underperform on gradient descent but stabilize training.