Transformers Learn In-Context by Gradient Descent

ETH zürich · Google Research

Johannes von Oswald, Eyvind Niklasson, Ettore Randazzo, João Sacramento, Alexander Mordvintsev, Andrey Zhmoginov, Max Vladymyrov

ICML International Conference On Machine Learning

Link to the paper and code

Goal & Hypothesis

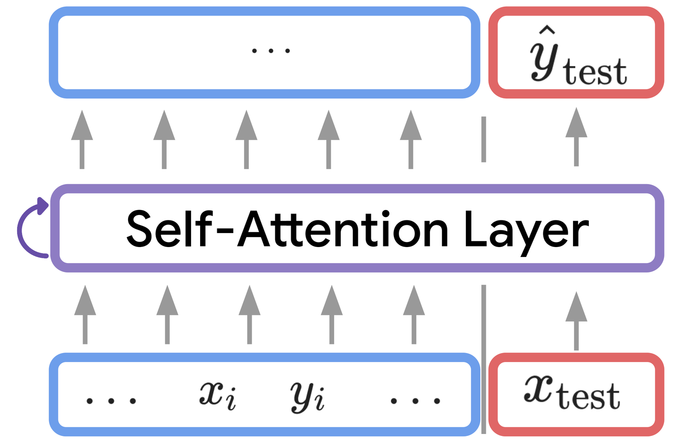
Understand better Transformers and especially their intriguing in-context learning capability.

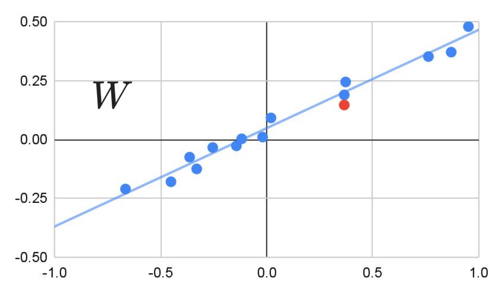
- Garg et al.[2] and Akyürek et al.[3]: trained Transformers on few-shot learning tasks often resemble gradient descent.
- Our goal: to explain this phenomenon by building on the relationship between self-attention and fast weight programming [Schmidhuber, 1992][1].

Contributions:

1) Construction of linear attention weights equivalent to do steps of GD on linear regression.
2) Evidence that this construction is found in practice
3) Show how MLPs in the architecture enables solving non-linear tasks
4) Relax assumption of construction by showing Transformers learn to copy

Setting





$$\hat{y}_{\text{test}} = t_\theta(x_{\text{test}}, \{(x_i, y_i)\}_{i=1}^N)$$
$$\text{where} \quad y_i = W x_i$$

Main Insights & Construction

Linear attention, if presented with correctly pre-processed data, can implement a step of gradient descent on the squared error regression loss.

Compare GD

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: $\Delta W = -\eta \nabla_W L = -\frac{\eta}{N}\sum_{i=1}^{N}(Wx_i - y_i)x_i^T$
3) Trick: Update all y: $L(W + \Delta W, \{(x_i, y_i)\}_i^N) = L(W, \{(x_i, y_i - \Delta W x_i)\}_i^N$
4) Correct test prediction : $\hat{y}_{\text{test}} \leftarrow -1 \cdot \hat{y}_{\text{test}} = -1 \cdot \sum -\Delta W x_{\text{test}}$

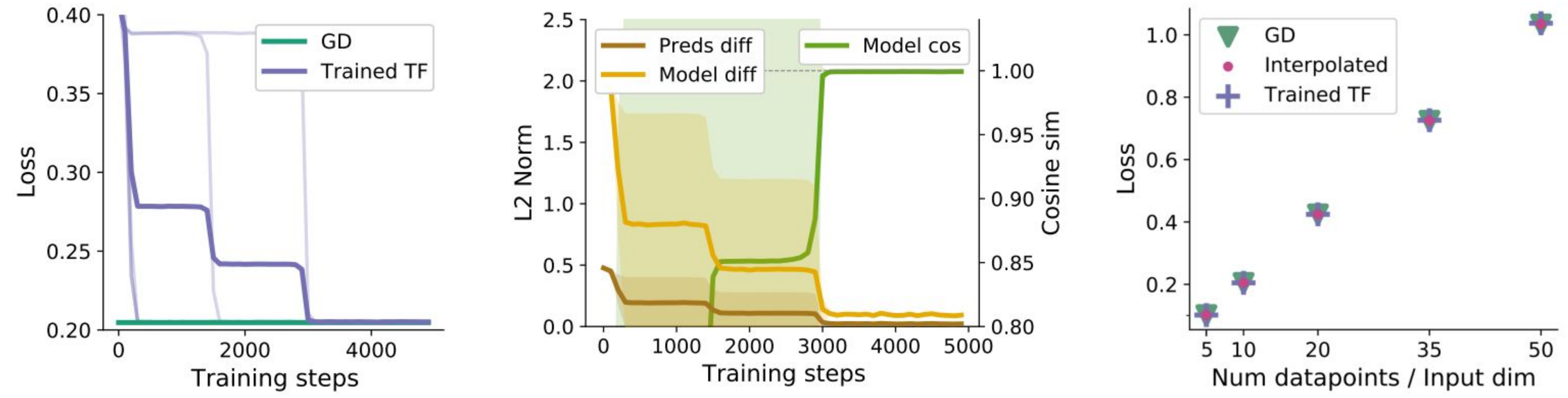
and linear Self-Attention GD

1) Assume token construction of copied data: $e_i = (x_i, y_i)$
2) Update tokens by linear self-attention: $e_i \leftarrow e_i + PVK^T q_i$
3) Can implement GD: $(x_i, y_i) \leftarrow (x_i, y_i) - (0, \frac{\eta}{N}\sum_j (Wx_j - y_j)x_j^T x_i)$

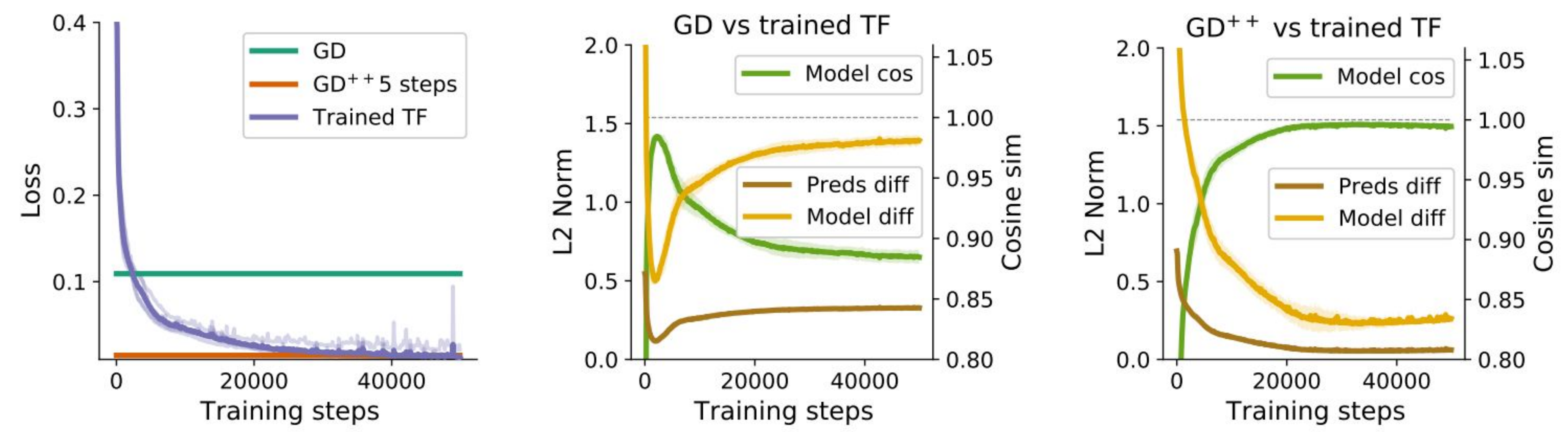
How Transformers can solve *linear* regression tasks

We present several pieces of evidence for the hypothesis that our construction is equivalent to what a trained transformer actually learns.

1) Trained single linear self-attention layer



2) Trained Transformer of 5 linear self-attention layers



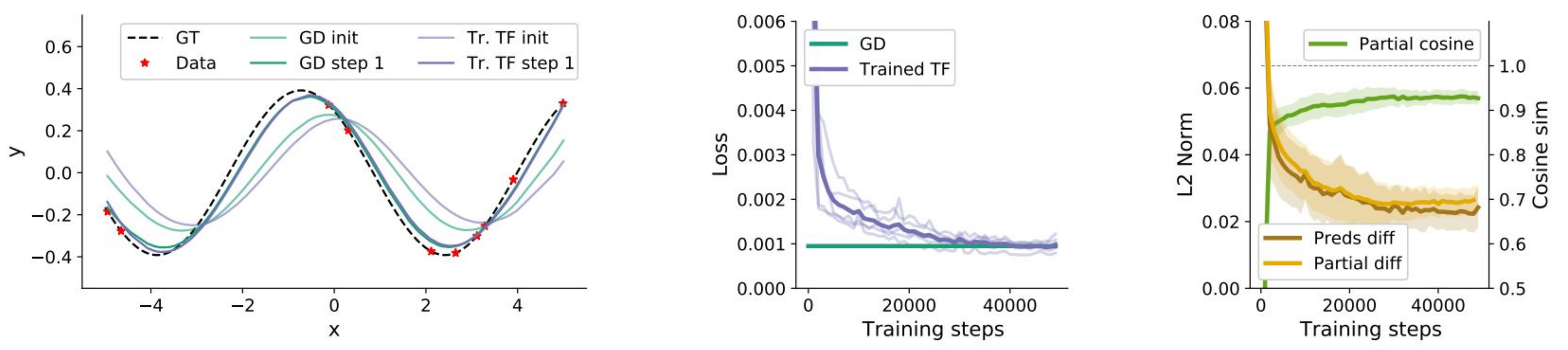
Multi layer Transformer outperform plain gradient descent and iteratively transform the input data X, as well as the targets Y with GD. We term this algorithm GD++

$$(x_i, y_i) \leftarrow (x_i, y_i) - (\gamma X X^T x_i, \Delta W x_i)$$

Key takeaway: When trained on linear regression taks, multi-layer linear self-attention Transformers implement GD, GD++ or *behave* very similarly.

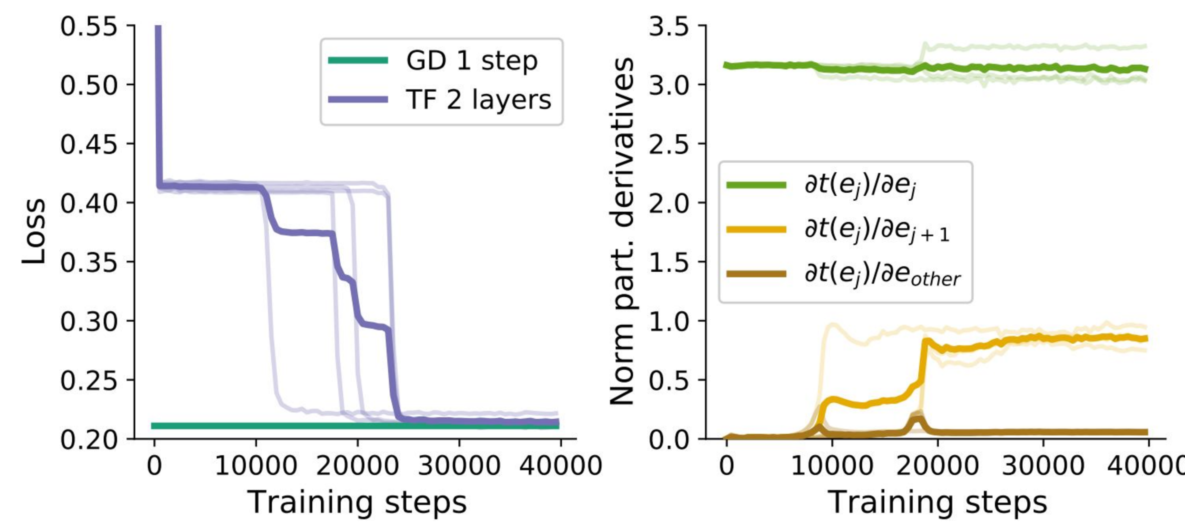
How Transformers can solve *non-linear* regression tasks

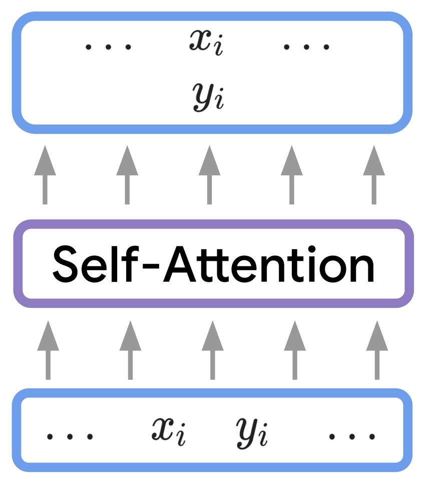
We hypothesize and provide some evidence that Transformers exploit MLPs to non-linearly embed data and solve non-linear regression tasks by gradient descent.



Copying data together

Most of the results in the paper assume tokens consist of concatenated inputs and targets. To relax this assumption, we show that Transformers can learn to construct this on their own to implement GD.



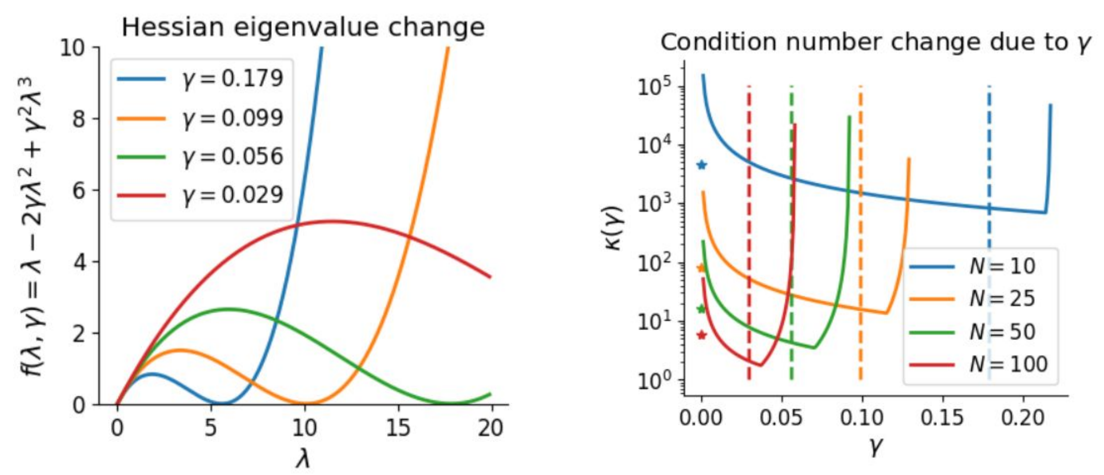


Analyses of GD++

Transformers iteratively transform the input data X while simultaneously doing GD steps. This leads to a change of the loss hessian H and therefore faster learning by better conditioned optimization problems.

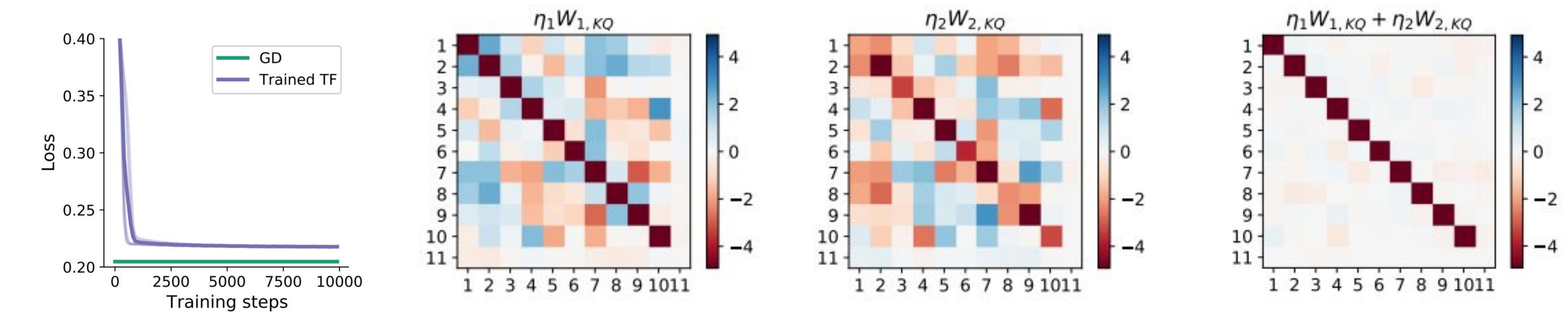
$$x_i \leftarrow x_i - \gamma X X^T x_i$$

$$H = XX^T = U\Sigma U^T \quad \text{vs} \quad H^{++} = U(\Sigma - 2\gamma\Sigma^2 + \gamma^2\Sigma^3)U^T$$



Linearization of softmax self-attention

Multi-head softmax self-attention layers can linearize themselves to approximate a step of GD in a similar fashion.



References

- Schmidhuber, J. Learning to control fast-weight memories: An alternative to dynamic recurrent networks. Neural Computation 1992.
- Garg, S., Tsipras, D., Liang, P., and Valiant, G. What can transformers learn in-context? a case study of simple function classes. In Oh, A. H., Agarwal, A., Belgrave, D., and Cho, K. (eds.), NeurIPS, 2022.
- Akyurek, E., Schuurmans, D., Andreas, J., Ma, T., and ¨ Zhou, D. What learning algorithm is in-context learning? investigations with linear models. ICLR, 2023.

Transformers Learn In-Context by Gradient Descent

ETH zürich · Google Research
Johannes von Oswald, Eyvind Niklasson, Ettore Randazzo, João Sacramento, Alexander Mordvintsev, Andrey Zhmoginov, Max Vladymyrov

ICML International Conference On Machine Learning

Link to the paper and code

Goal & Hypothesis

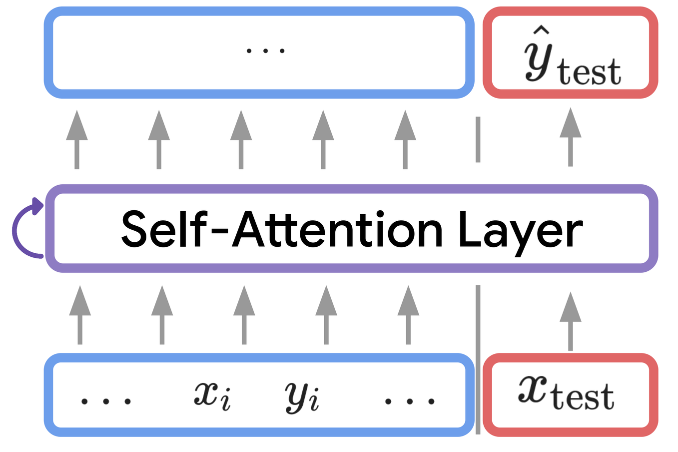
Understand better Transformers and especially their intriguing in-context learning capability.

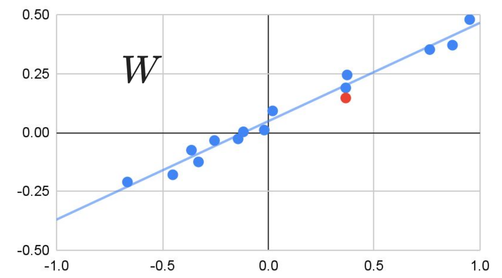
- Garg et al.[2] and Akyürek et al.[3]: trained Transformers on few-shot learning tasks often resemble gradient descent.
- Our goal: to explain this phenomenon by building on the relationship between self-attention and fast weight programming [Schmidhuber, 1992][1].

Contributions:

1) Existence of linear attention weights equivalent to GD on linear regression.
2) Evidence that this construction can be found through training.
3) Show how MLPs in the architecture enables solving non-linear tasks.
4) Relax assumption of construction by showing Transformers learn to copy.

Setting





$$\hat{y}_{test} = t_\theta(x_{test}, \{(x_i, y_i)\}_{i=1}^N\})$$
$$\text{where} \quad y_i = W x_i$$

Main Insights & Construction

Each layer of the linear attention can implement a step of gradient descent on the squared error regression loss.

Compare GD

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: $\Delta W = -\eta \nabla_W L = -\frac{\eta}{N} \sum_{i=1}^{N} (Wx_i - y_i) x_i^T$
3) Trick: Update all y: $L(W + \Delta W, \{(x_i, y_i)\}_i^N) = L(W, \{(x_i, y_i - \Delta W x_i)\}_i^N$
4) Correct test prediction : $\hat{y}_{test} \leftarrow -1 \cdot \hat{y}_{test} = -1 \cdot \sum -\Delta W x_{test}$

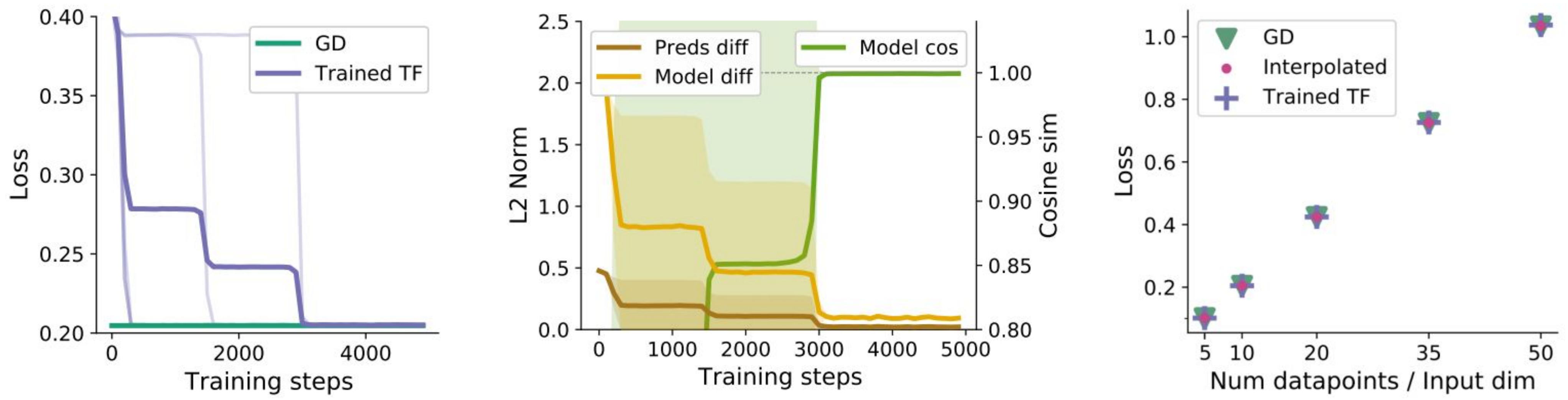
and linear Self-Attention GD

1) Assume token construction of copied data: $e_i = (x_i, y_i)$
2) Update tokens by linear self-attention: $e_i \leftarrow e_i + PVK^T q_i$
3) Can implement GD: $(x_i, y_i) \leftarrow (x_i, y_i) - (0, \frac{\eta}{N} \sum_j (Wx_j - y_j) x_j^T x_i)$

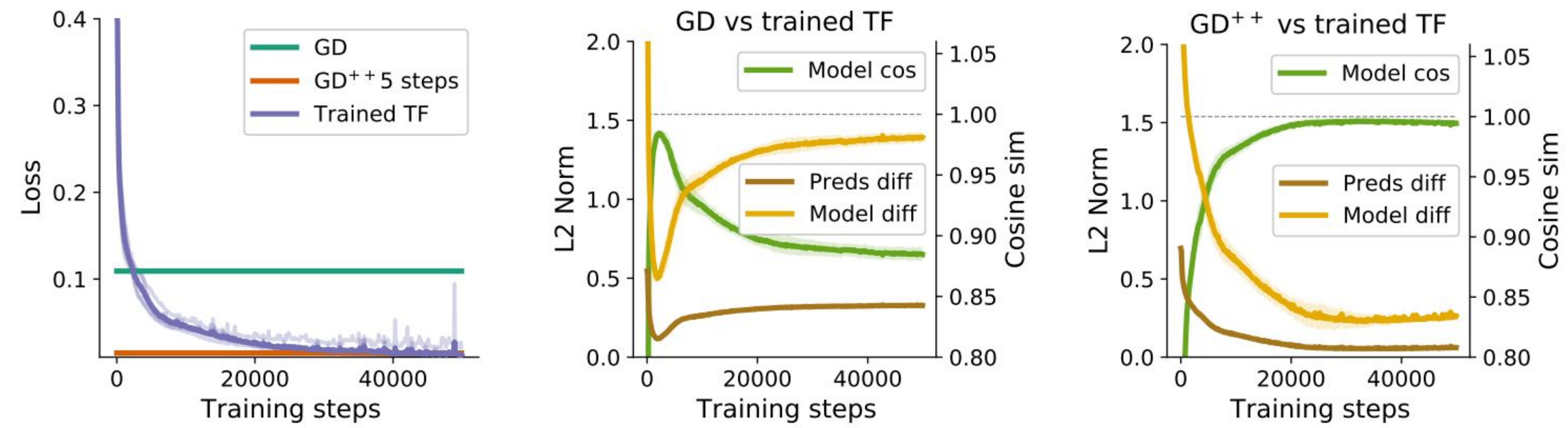
How Transformers can solve *linear* regression tasks

We present several pieces of evidence for the hypothesis that our construction is equivalent to what a trained transformer actually learns.

1) Trained single linear self-attention layer



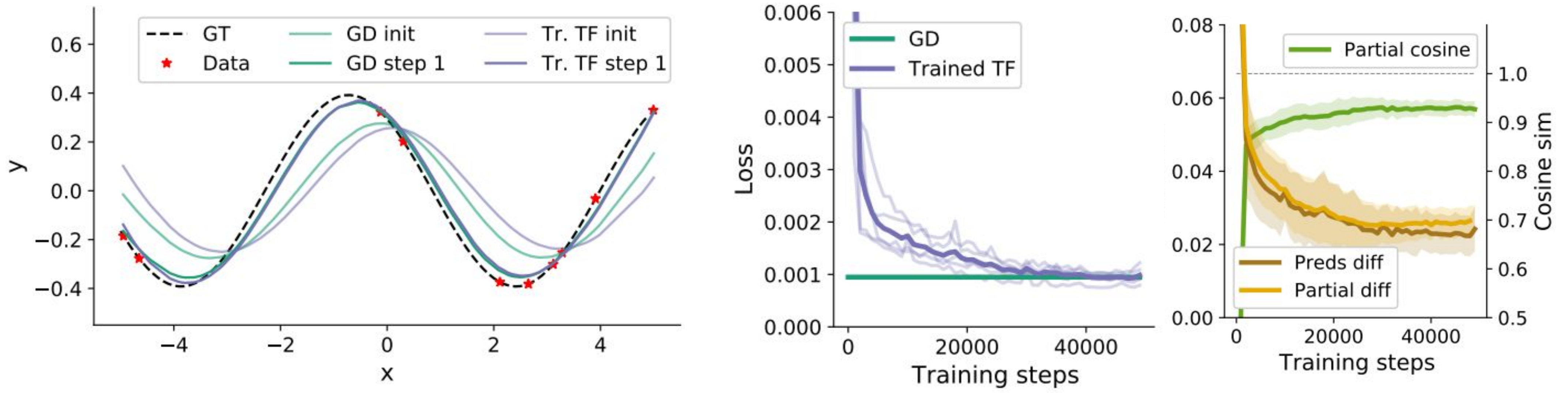
2) Trained Transformer of 5 linear self-attention layers



Key takeaway: *When trained on linear regression taks, multi-layer linear self-attention Transformers implement GD, GD++ or behave very similarly.*

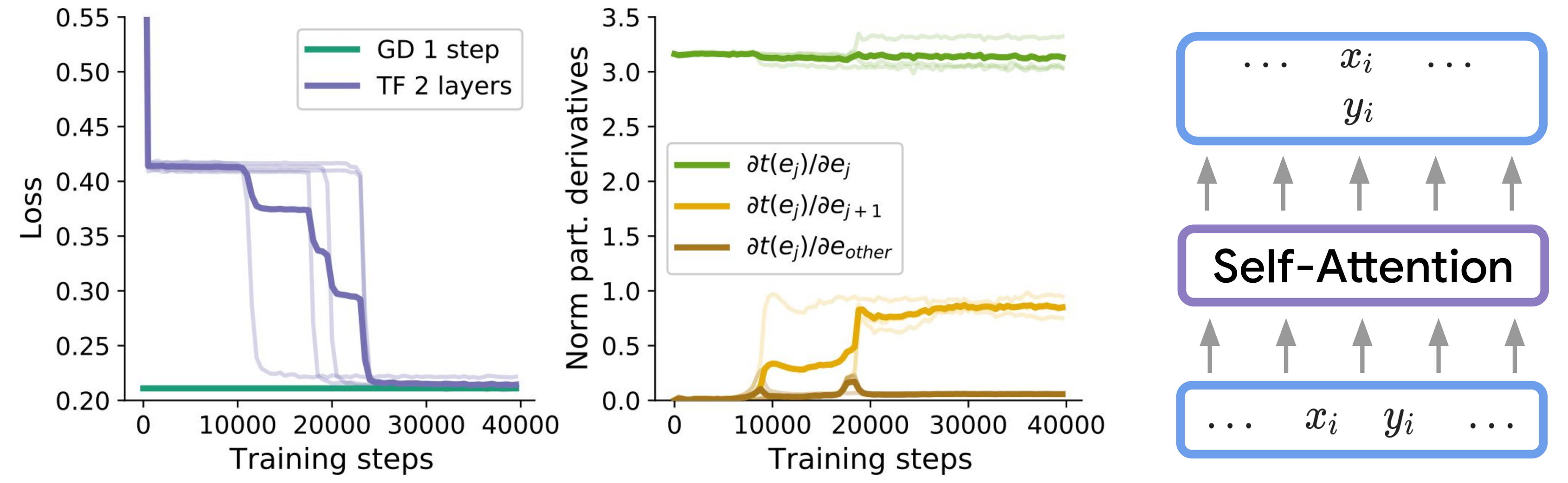
How Transformers can solve *non-linear* regression tasks

We hypothesize and provide some evidence that Transformers exploit MLPs to non-linearly embed data and solve non-linear regression tasks by gradient descent.



Copying data together

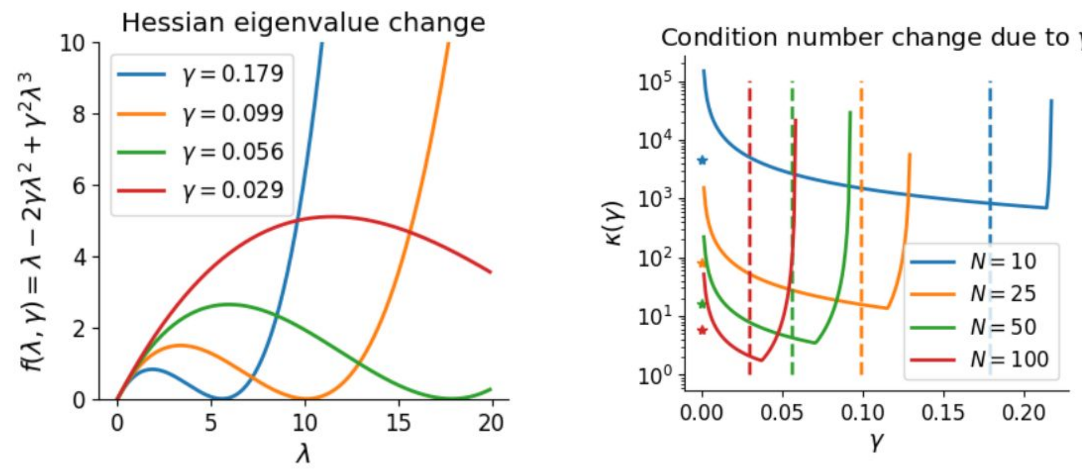
Most of the results in the paper assume tokens consist of concatenated inputs and targets. To relax this assumption, we show that Transformers can learn to construct this on their own to implement GD.



Analyses of GD++

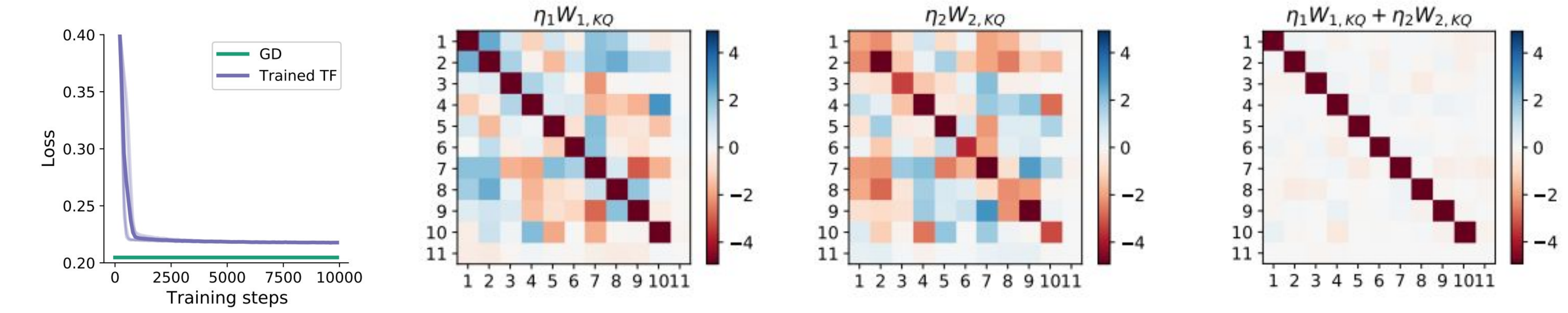
Transformers iteratively transform the input data X while simultaneously doing GD steps. This leads to a change of the loss hessian H and therefore faster learning by better conditioned optimization problems.

$$x_i \leftarrow x_i + \gamma X X^T x_i$$
$$H = XX^T = U\Sigma U^T \quad \text{vs} \quad H^{++} = U(\Sigma - 2\gamma\Sigma^2 + \gamma^2\Sigma^3)U^T$$



Linearization of softmax self-attention

Multi-head softmax self-attention layers can linearize themselves to approximate a step of GD in a similar fashion.



References

- Schmidhuber, J. Learning to control fast-weight memories: An alternative to dynamic recurrent networks. Neural Computation 1992.
- Garg, S., Tsipras, D., Liang, P., and Valiant, G. What can transformers learn in-context? a case study of simple function classes. In Oh, A. H., Agarwal, A., Belgrave, D., and Cho, K. (eds.), NeurIPS, 2022.
- Akyurek, E., Schuurmans, D., Andreas, J., Ma, T., and ¨ Zhou, D. What learning algorithm is in-context learning? investigations with linear models. ICLR, 2023.

Transformers Learn In-Context by Gradient Descent

ETH zürich · Google Research

Johannes von Oswald, Eyvind Niklasson, Ettore Randazzo, João Sacramento, Alexander Mordvintsev, Andrey Zhmoginov, Max Vladymyrov

ICML International Conference On Machine Learning

Goal

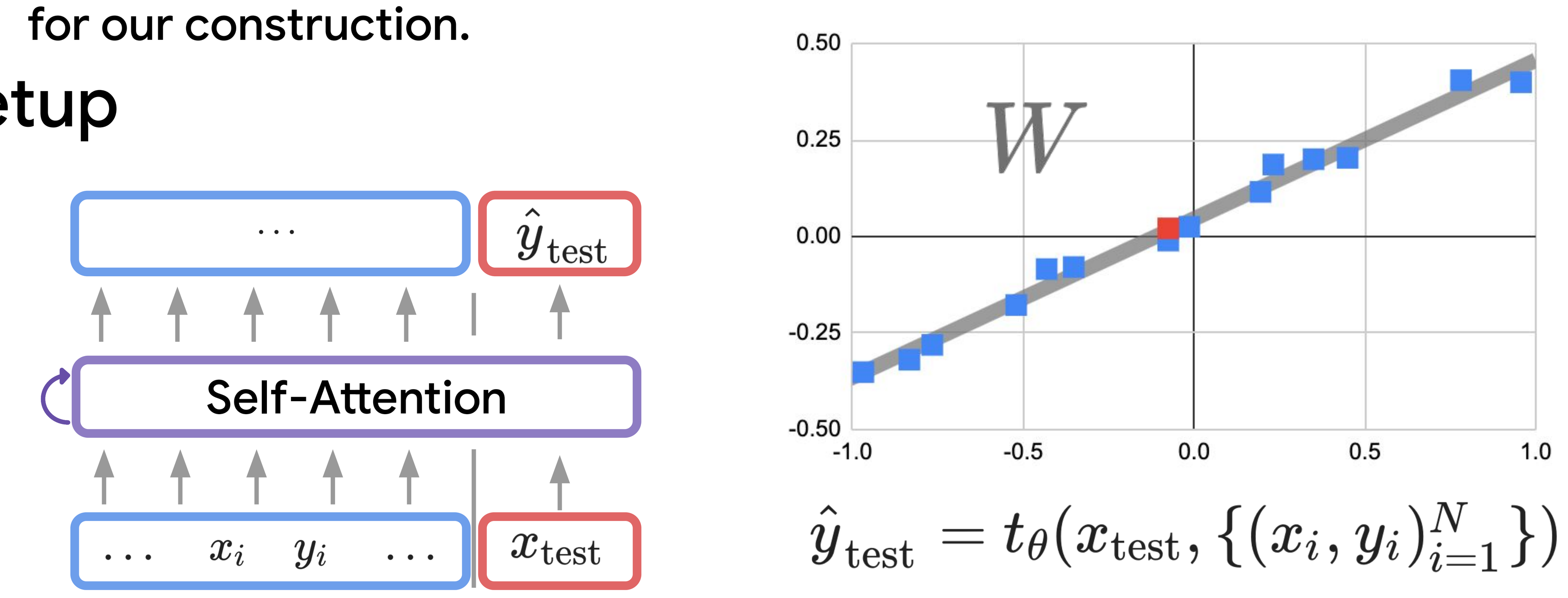
To better understand Transformers and especially their intriguing in-context learning capability. Garg et al.[2] and Akyürek et al.[3] trained Transformers on few-shot learning tasks often resembling gradient descent (GD).

Our goal is to explain this phenomenon, building on the relationship between self-attention and fast weight programming [Schmidhuber, 1992][1].

Our contributions are

- Constructing linear attention weights such that inference is equivalent to performing gradient descent on a linear regression loss.
- Evidence that this construction is learned in practice.
- Evidence that Transformers' MLPs extend the above to non-linear tasks.
- Evidence that Transformers learn to copy and merge tokens as assumed for our construction.

Setup



$$\hat{y}_{\text{test}} = t_\theta(x_{\text{test}}, \{(x_i, y_i)_{i=1}^N\})$$
$$\text{where} \quad y_i = Wx_i$$

Main insights and construction

A linear attention layer, when presented with the data pairwise concatenated, can implement a step of gradient descent on the squared error regression loss. Below we demonstrate the fundamental similarity.

Gradient Descent

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: $\Delta W = -\eta \nabla_W L = -\frac{\eta}{N}\sum_{i=1}^N (Wx_i - y_i)x_i^T$
3) Trick - update all y: $L(W + \Delta W, \{(x_i, y_i)\}_i^N) = L(W, \{(x_i, y_i - \Delta W x_i)\}_i^N)$
4) Correct test prediction : $\hat{y}_{\text{test}} \leftarrow -1 \cdot \hat{y}_{\text{test}} = -1 \cdot \sum -\Delta W x_{\text{test}}$

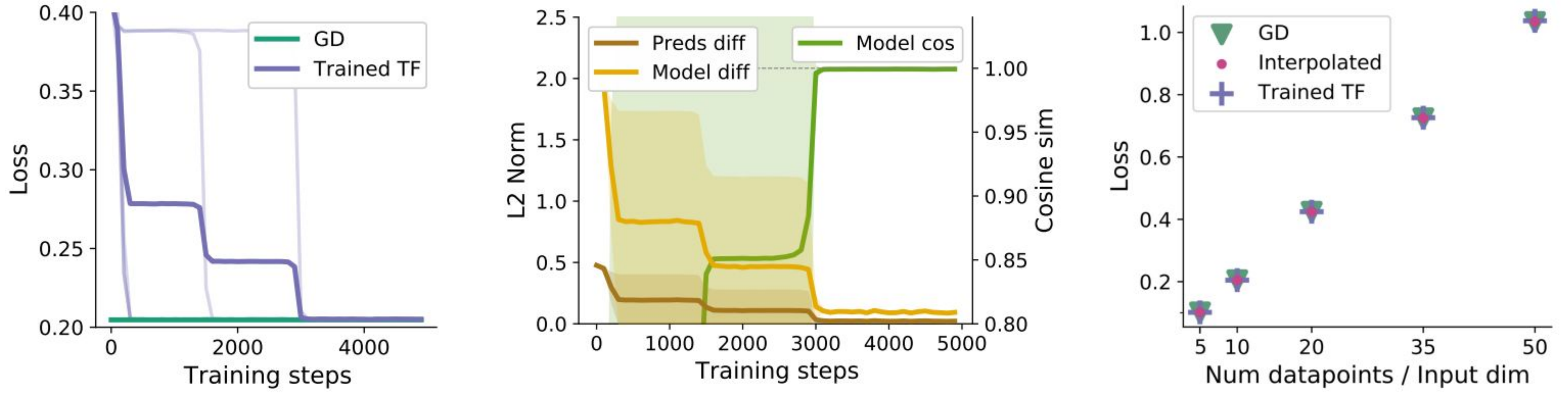
Linear Self Attention implementing Gradient Descent

1) Assume tokens constructed as follows: $e_i = (x_i, y_i)$
2) Update tokens by linear self-attention: $e_i \leftarrow e_i + PVK^T q_i$
3) Can implement GD: $(x_i, y_i) \leftarrow (x_i, y_i) - (0, \frac{\eta}{N}\sum_j (Wx_j - y_j)x_j^T x_i)$

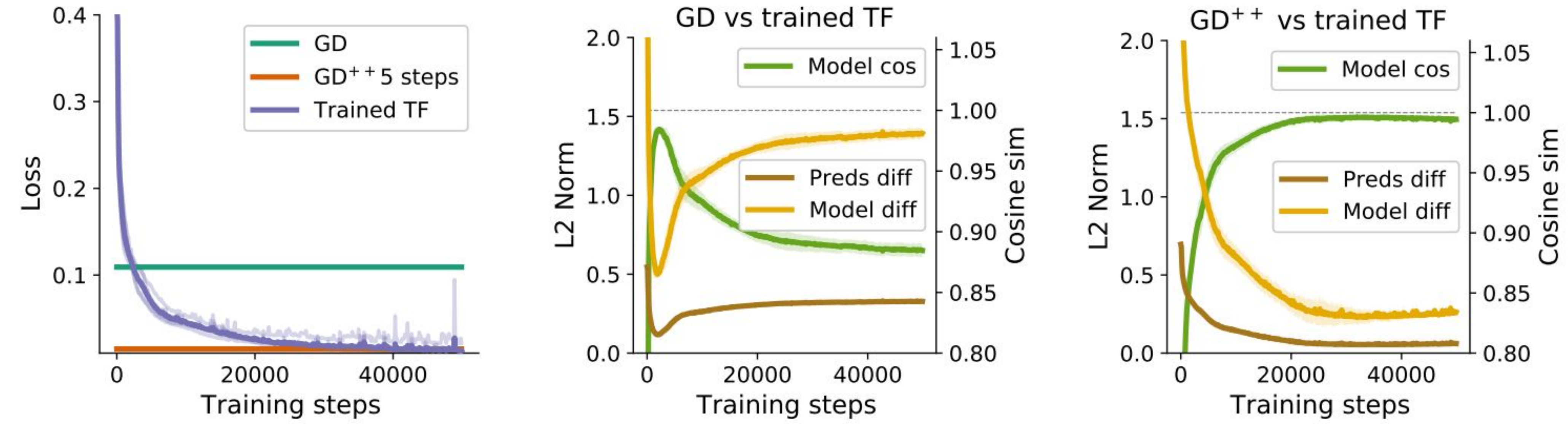
Transformers **solve** *linear* regression tasks using GD

We present several pieces of evidence for the hypothesis that our construction is equivalent to what a trained transformer learns given data of this form during training.

- Trained linear self-attention layer (Trained TF) versus gradient descent (GD)



- Trained five-layer linear Transformer (Trained TF) versus gradient descent (GD, GD++)



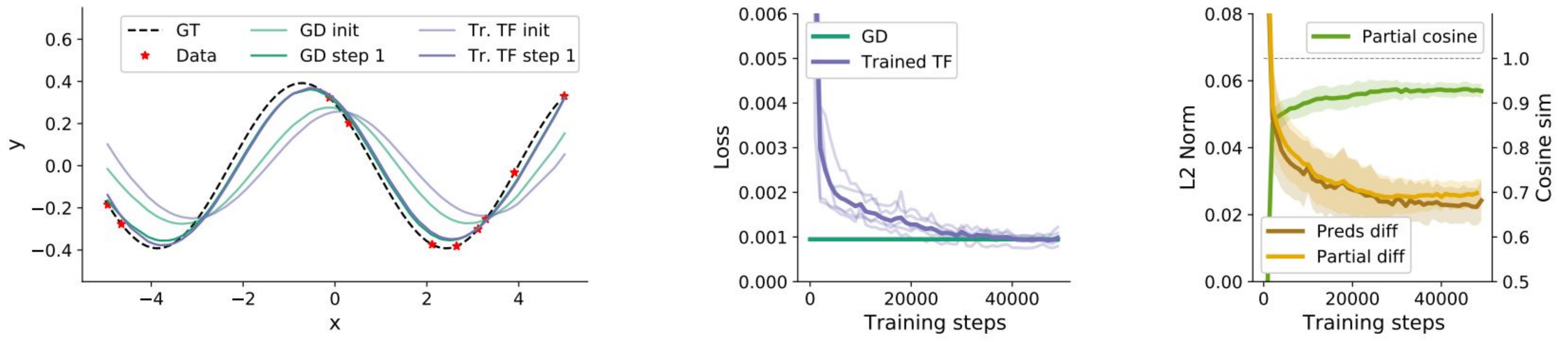
Multi-layer Transformers **outperform** plain gradient descent and iteratively precondition the input data X, while simultaneously updating Y using GD. We term this algorithm **GD++.**

$$(x_i, y_i) \leftarrow (x_i, y_i) - (\gamma XX^T x_i, \Delta W x_i)$$

Key takeaway is that when trained on linear regression tasks, multi-layer linear self-attention Transformers implement GD, GD++, or *behave* very similarly.

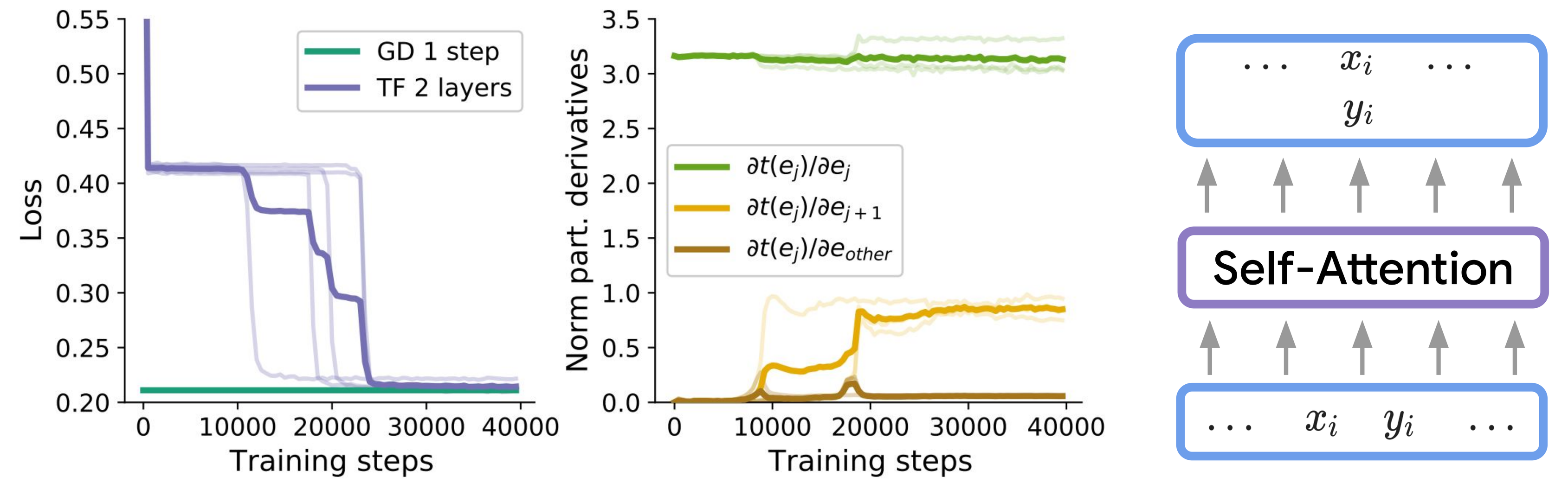
Transformers **solve** *non-linear* regression tasks using GD

We hypothesize and provide some evidence that Transformers exploit MLPs to non-linearly embed data and solve non-linear regression tasks by gradient descent.



Aligning data

Most of the results in the paper assume tokens consist of inputs and targets concatenated together. To relax this assumption, we show that Transformers learn to concatenate consecutive tokens, as demonstrated by a sensitivity analysis to the previous token and an identical final loss.

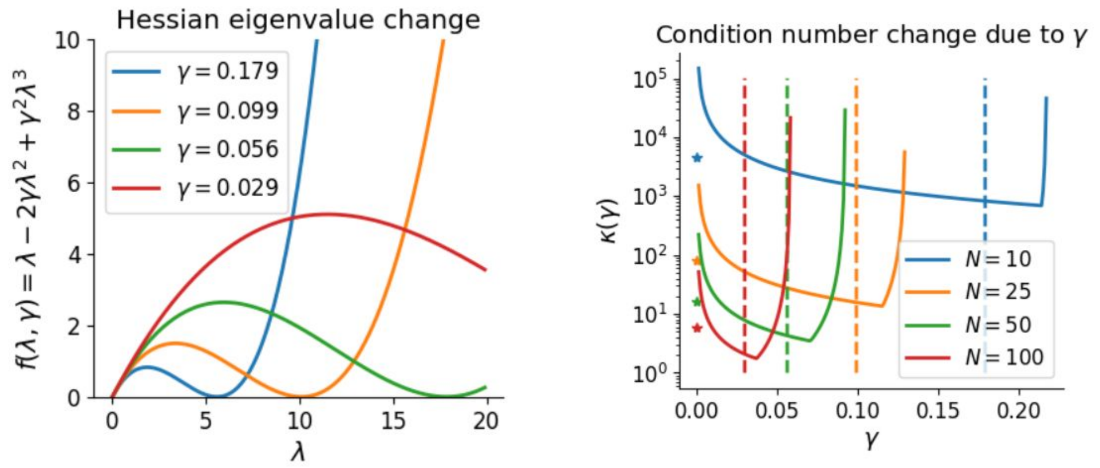


Analyses of GD++

Transformers iteratively transform the input data X while simultaneously doing GD steps. This leads to changes in the hessian of the loss (H) which allow for faster learning, (as a better conditioned optimization problem).

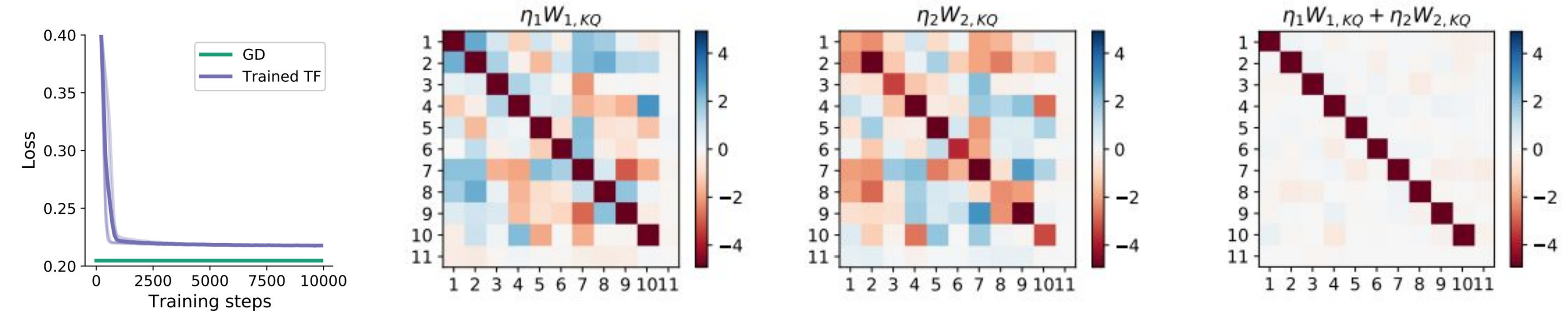
$$x_i \leftarrow x_i - \gamma XX^T x_i$$

$$H = XX^T = U\Sigma U^T \quad \text{vs} \quad H^{++} = U(\Sigma - 2\gamma\Sigma^2 + \gamma^2\Sigma^3)U^T$$



Linearization of softmax self-attention

Multi-head softmax self-attention layers can linearize themselves to approximate a step of GD in a similar fashion.



References

- Schmidhuber, J. Learning to control fast-weight memories: An alternative to dynamic recurrent networks. Neural Computation 1992.
- Garg, S., Tsipras, D., Liang, P., and Valiant, G. What can transformers learn in-context? a case study of simple function classes. In Oh, A. H., Agarwal, A., Belgrave, D., and Cho, K. (eds.), NeurIPS, 2022.
- Akyurek, E., Schuurmans, D., Andreas, J., Ma, T., and ¨ Zhou, D. What learning algorithm is in-context learning? investigations with linear models. ICLR, 2023.

Paper + Code

ETH zürich
Google Research

Transformers Learn In-Context by Gradient Descent

Johannes von Oswald, Eyvind Niklasson, Ettore Randazzo, João Sacramento, Alexander Mordvintsev, Andrey Zhmoginov, Max Vladymyrov

ICML International Conference On Machine Learning

Link to the paper and code

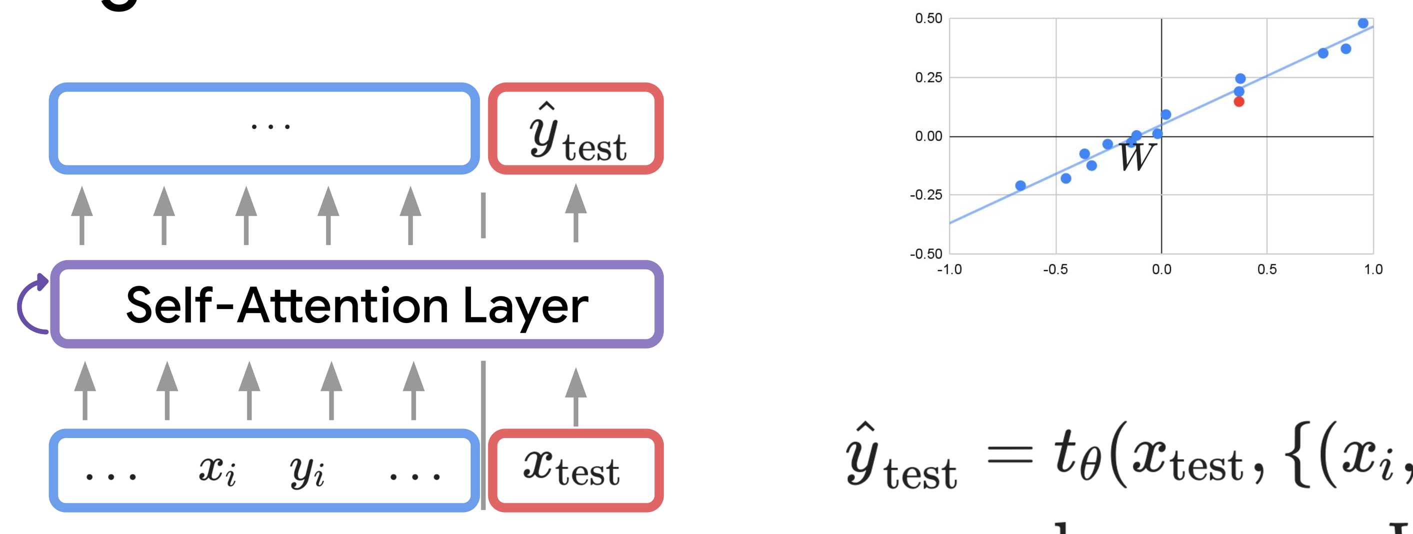
Goal & Hypothesis

To better understand Transformers and especially their intriguing in-context learning capability. Garg et al.[2] and Akyürek et al.[3] trained Transformers on few-shot learning tasks often resembling gradient descent (GD). **Our goal** is to explain this phenomenon, building on the relationship between self-attention and fast weight programming [Schmidhuber, 1992][1].

Contributions

- Existence of linear attention weights equivalent to GD on linear regression.
- Evidence that training a transformer converges to this construction.
- Show how MLPs in the architecture enables solving non-linear tasks.
- Relax assumption of construction by showing Transformers learn to copy.

Setting



$$\hat{y}_{test} = t_\theta(x_{test}, \{(x_i, y_i)\}_{i=1}^N) \quad \text{where} \quad y_i = W x_i$$

Main Insights & Construction

Each layer of the linear attention can implement a step of gradient descent on the squared error regression loss.

Compare GD

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: $\Delta W = -\eta \nabla_W L = -\frac{\eta}{N}\sum_{i=1}^N (Wx_i - y_i)x_i^T$
3) Trick: Update all y: $L(W + \Delta W, \{(x_i, y_i)\}_i^N) = L(W, \{(x_i, y_i - \Delta W x_i)\}_i^N)$
4) Correct test prediction : $\hat{y}_{test} \leftarrow -1 \cdot \hat{y}_{test} = -1 \cdot \sum -\Delta W x_{test}$

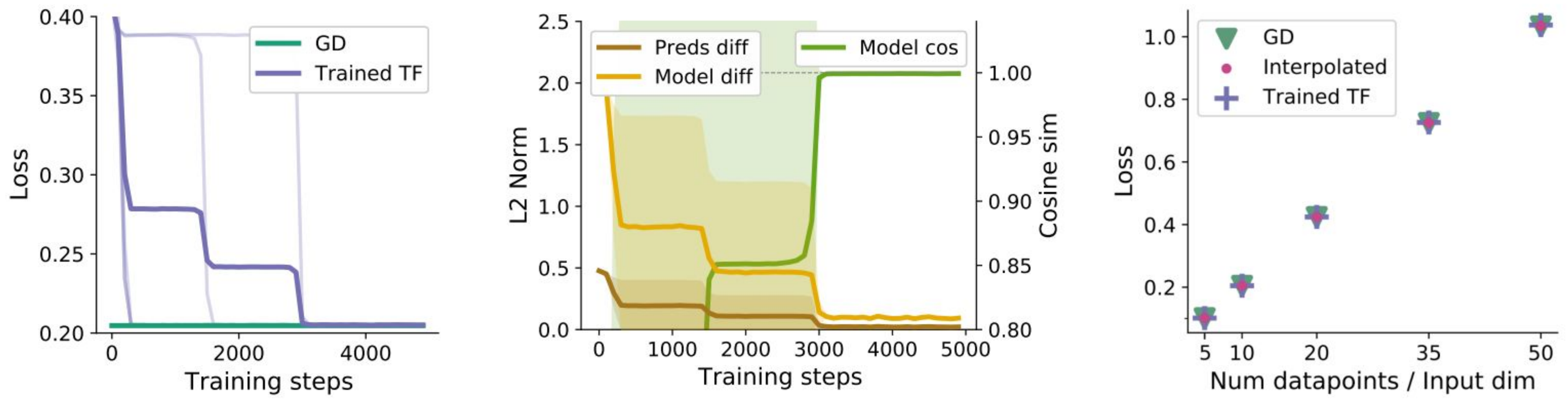
and linear Self-Attention GD

1) Assume token construction of copied data: $e_i = (x_i, y_i)$
2) Update tokens by linear self-attention: $e_i \leftarrow e_i + PVK^T q_i$
3) Can implement GD: $(x_i, y_i) \leftarrow (x_i, y_i) - (0, \frac{\eta}{N}\sum_j (Wx_j - y_j)x_j^T x_i)$

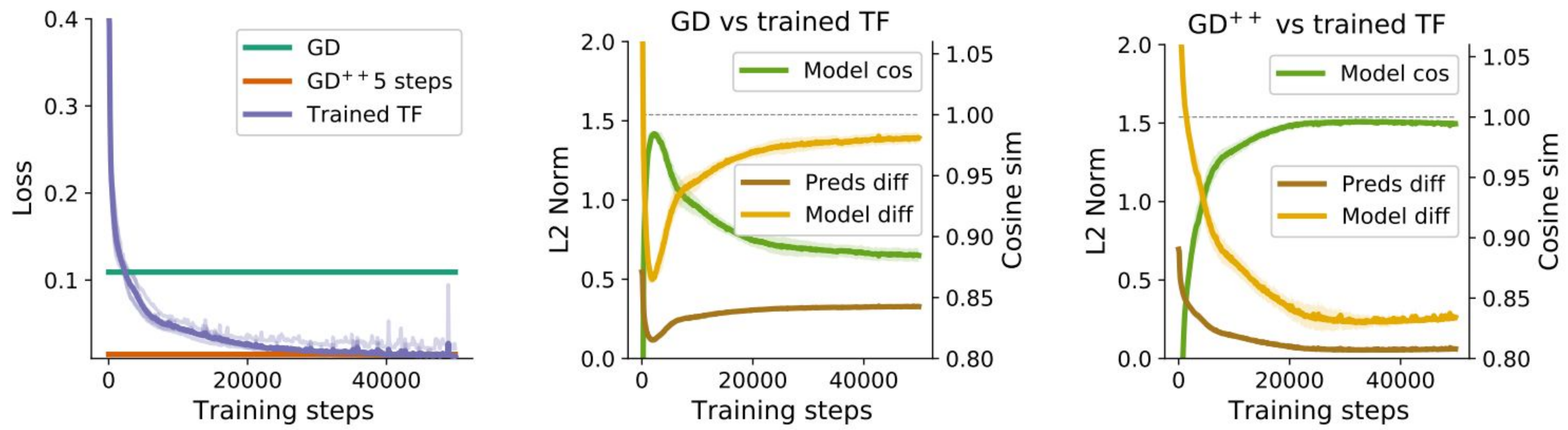
How Transformers can solve *linear* regression tasks

We present several pieces of evidence for the hypothesis that our construction is equivalent to what a trained transformer actually learns.

1) Trained single linear self-attention layer



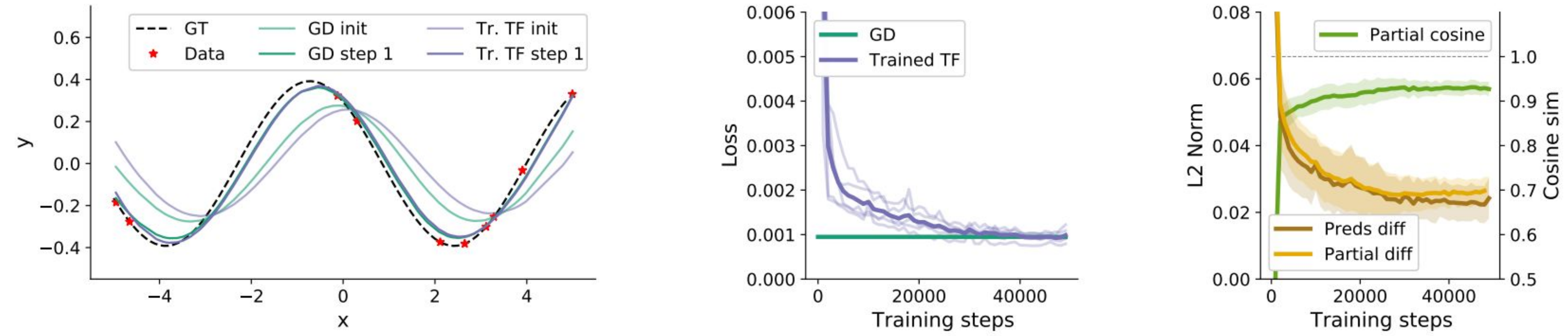
2) Trained Transformer of 5 linear self-attention layers



Key takeaway: When trained on linear regression taks, multi-layer linear self-attention Transformers implement GD, GD++ or *behave* very similarly.

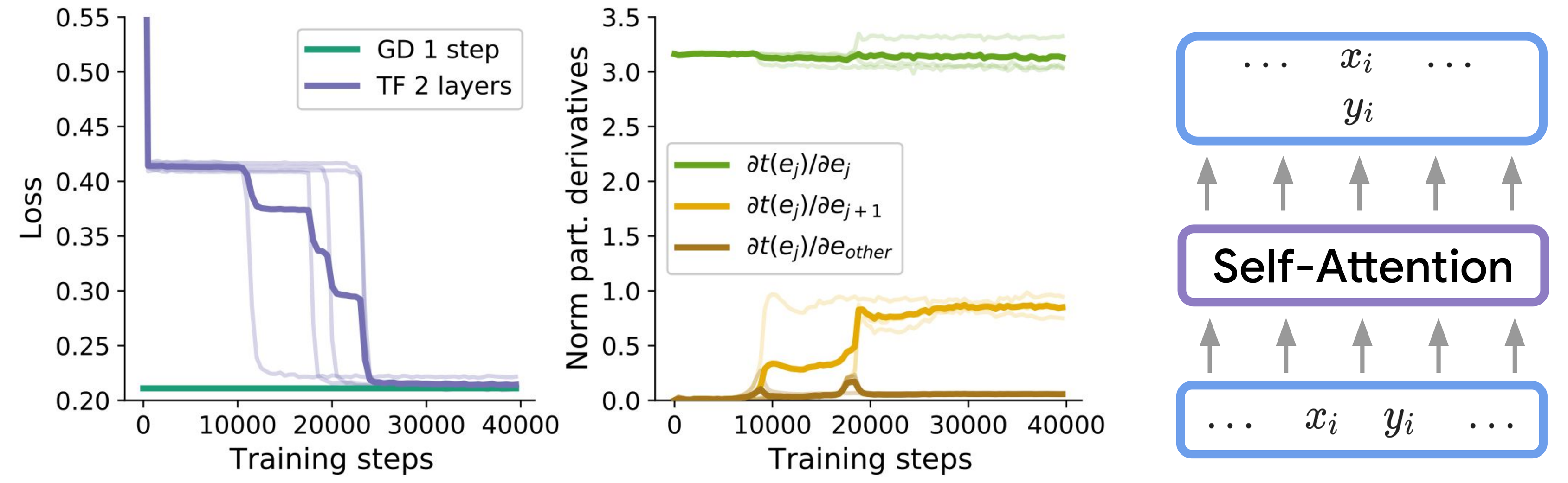
How Transformers can solve *non-linear* regression tasks

We hypothesize and provide some evidence that Transformers exploit MLPs to non-linearly embed data and solve non-linear regression tasks by gradient descent.



Copying data together

Most of the results in the paper assume tokens consist of concatenated inputs and targets. To relax this assumption, we show that Transformers can learn to construct this on their own to implement GD.

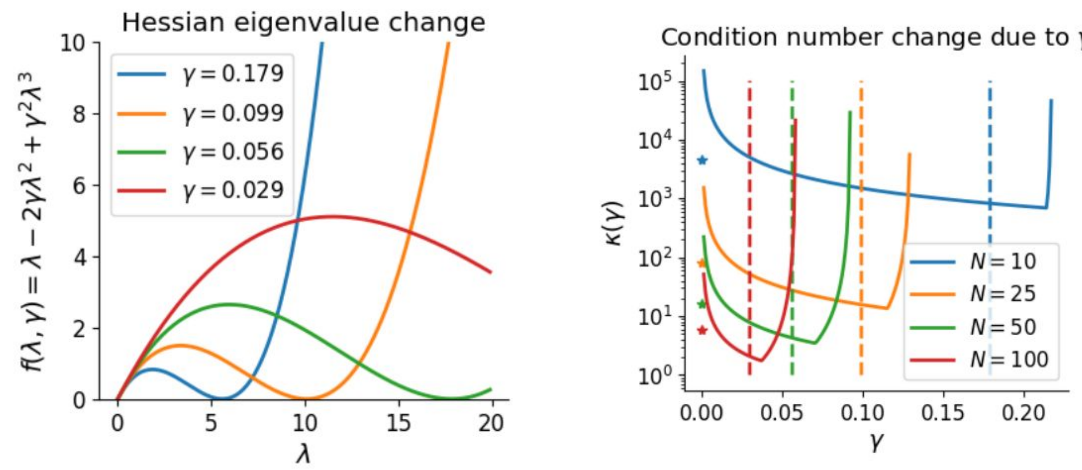


Analyses of GD++

Transformers iteratively transform the input data X while simultaneously doing GD steps. This leads to a change of the loss hessian H and therefore faster learning by better conditioned optimization problems.

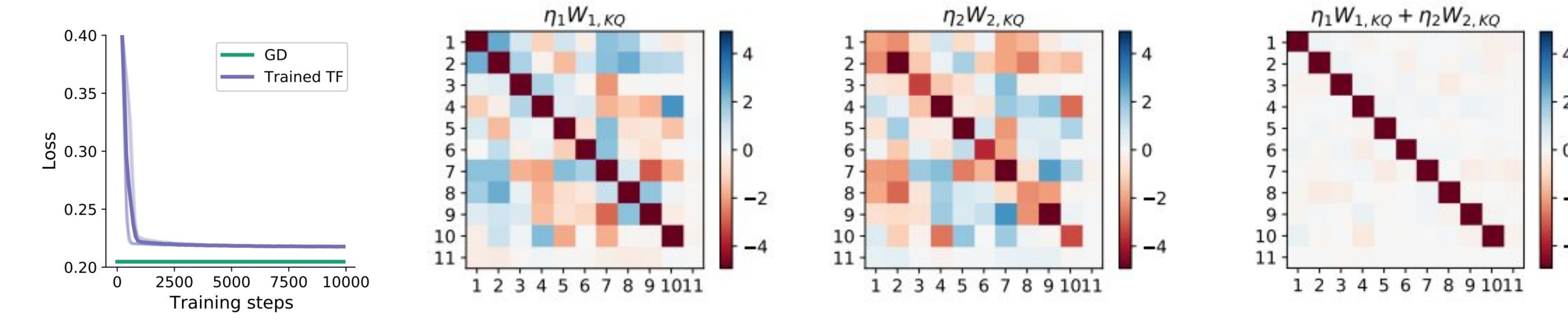
$$x_i \leftarrow x_i + \gamma X X^T x_i$$

$$H = XX^T = U\Sigma U^T \quad \text{vs} \quad H^{++} = U(\Sigma - 2\gamma\Sigma^2 + \gamma^2\Sigma^3)U^T$$



Linearization of softmax self-attention

Multi-head softmax self-attention layers can linearize themselves to approximate a step of GD in a similar fashion.



References

- Schmidhuber, J. Learning to control fast-weight memories: An alternative to dynamic recurrent networks. Neural Computation 1992.
- Garg, S., Tsipras, D., Liang, P., and Valiant, G. What can transformers learn in-context? a case study of simple function classes. In Oh, A. H., Agarwal, A., Belgrave, D., and Cho, K. (eds.), NeurIPS, 2022.
- Akyurek, E., Schuurmans, D., Andreas, J., Ma, T., and ¨ Zhou, D. What learning algorithm is in-context learning? investigations with linear models. ICLR, 2023.

Transformers Learn In-Context by Gradient Descent

Johannes von Oswald, Eyvind Niklasson, Ettore Randazzo, João Sacramento, Alexander Mordvintsev, Andrey Zhmoginov, Max Vladymyrov

Google Research

ICML International Conference On Machine Learning

Link to the paper and code

Goal & Hypothesis

Understand better Transformers and especially their intriguing in-context learning capability. Garg et al., 2022 and others showed that when training Transformers on few-shot learning tasks, the functions obtained often resemble solutions obtained with gradient descent.

Our aim is to explain this phenomenon by building on the relationship between self-attention and fast weight programming from Schmidhuber.

Setting

Main Insights

Linear attention, if presented with correctly pre-processed data, can implement a step of gradient descent on the squared error regression loss.

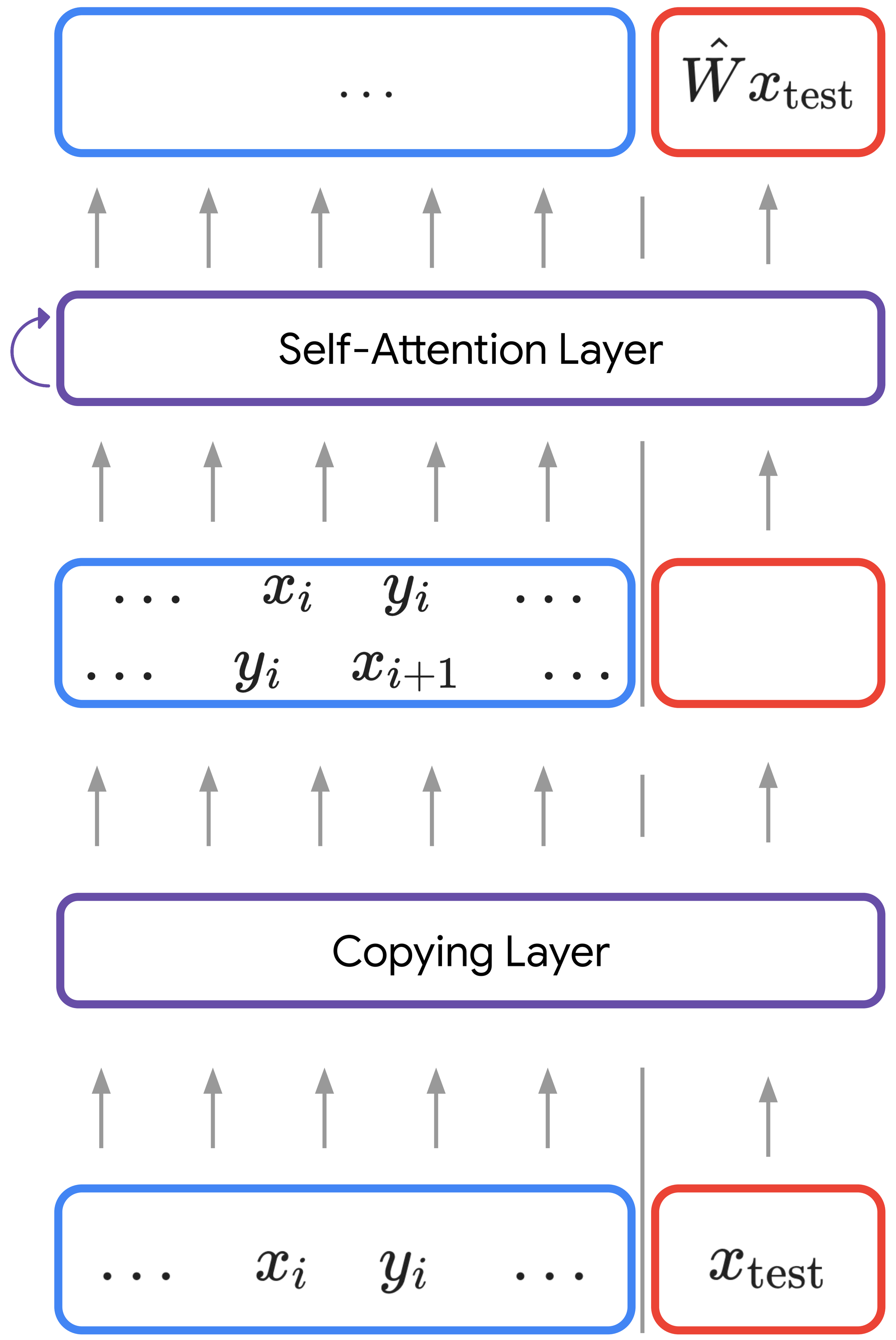
Compare GD

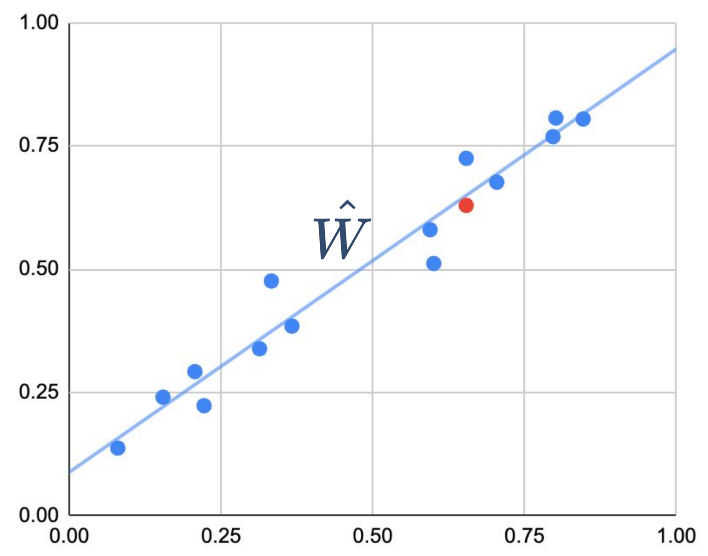
1) Compute regression loss: $L(W, \{(x_i, y_i)\}_{i=1}^N) = \frac{1}{2N}\sum_{i=1}^{N}(Wx_i - y_i)^2$
2) Gradient descent: $\Delta W = -\eta \nabla_W L(W, \{(x_i, y_i)\}_{i=1}^N)$
3) Update y's: $L(W + \Delta W, \{(x_i, y_i)\}_i^N) = L(W, \{(x_i, y_i - \Delta W x_i)\}_i^N)$
4) Correct test prediction : $\hat{y}_{\text{test}} \leftarrow -1 \cdot \hat{y}_{\text{test}} = -1 \cdot \sum -\Delta W x_{\text{test}}$

and linear Self-Attention GD

1) Assume token construction of copied data: $e_i = (x_i, y_i)$
2) Update tokens by linear self-attention: $e_i \leftarrow e_i + PVK^T q_i$
3) Goal: $(x_i, y_i) \leftarrow (x_i, y_i) - (0, \frac{\eta}{N}\sum_j (Wx_j - y_j)x_j^T x_i)$

Empirical results





$$\hat{y} = t_\theta(x_{\text{test}}, \{(x_i, y_i)_{i=1}^N\})$$
$$\text{where} \quad y_i = Wx_i$$

References

- Schmidhuber, J. Learning to control fast-weight memories: An alternative to dynamic recurrent networks. Neural Computation 1992.
- Garg, S., Tsipras, D., Liang, P., and Valiant, G. What can transformers learn in-context? a case study of simple function classes. In Oh, A. H., Agarwal, A., Belgrave, D., and Cho, K. (eds.), NeurIPS, 2022.
- Akyurek, E., Schuurmans, D., Andreas, J., Ma, T., and ¨ Zhou, D. What learning algorithm is in-context learning? investigations with linear models. ICLR, 2023.

Meta-Learning Bidirectional Update Rules

Mark Sandler, Max Vladymyrov, Andrey Zhmoginov, Nolan Miller, Andrew Jackson, Tom Madams, Blaise Agüera y Arcas

Goal

Meta-learn synapse update rules with very mild assumptions on the inner-loop (no loss functions, no gradients) that learns faster than traditional methods.

Motivation

SGD optimization via Backpropagation:

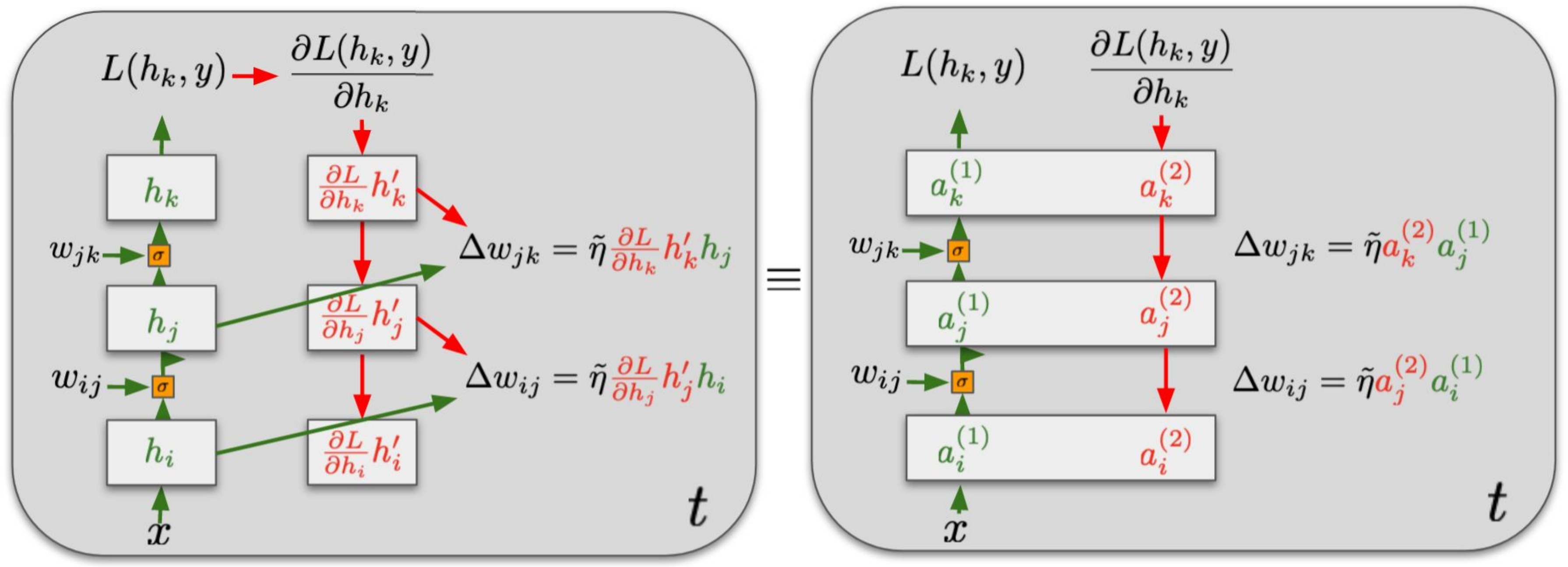
- Uses predefined loss function computed at every iteration.
- The loss is minimized via gradient descent (steepest direction of the current loss).
    - Optimization can use previous iterations (e.g. momentum), but (mostly) can't see forward.
- Optimization procedure is independent from the dataset.

Bidirectional Learning Update Rules (BLUR):

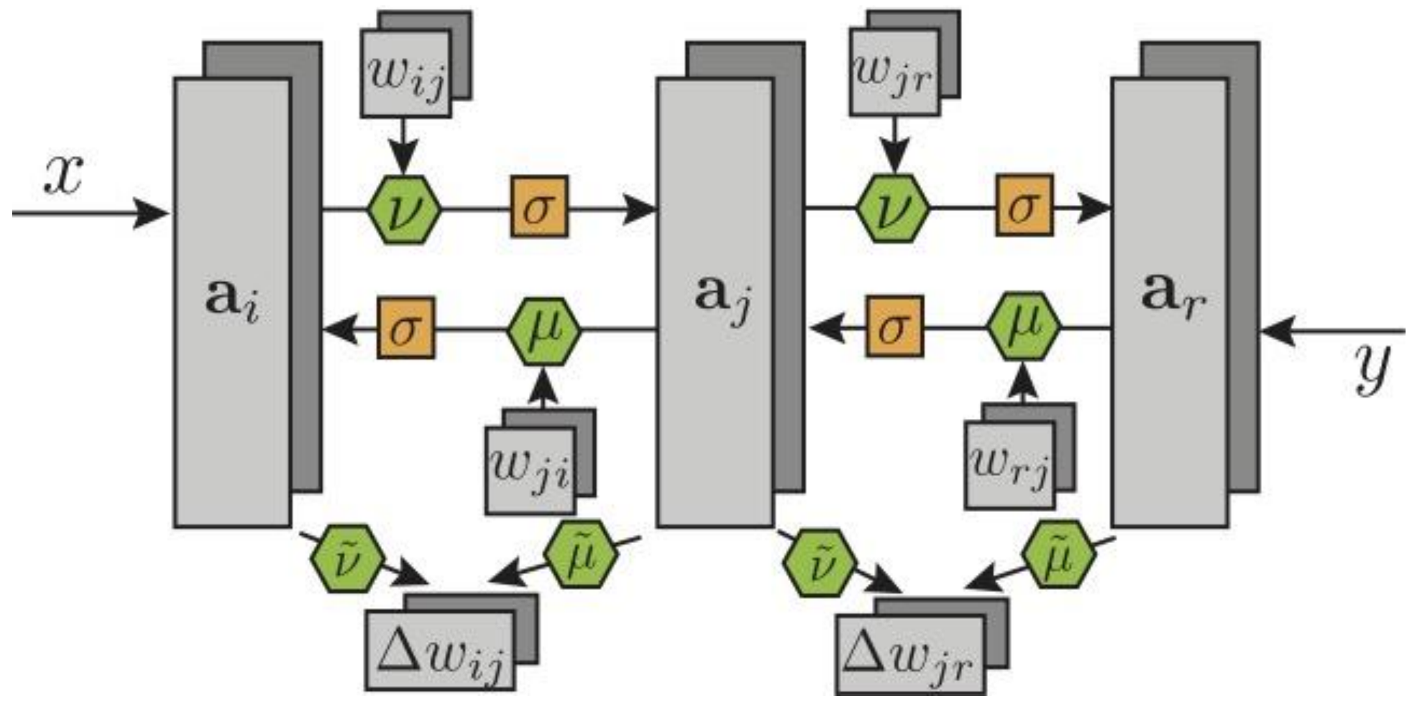
- Synapse updated rules are parametrized and meta-learned via a low-dimensional genome matrix.
- No predefined per-iteration loss function, no explicit gradients.
- Keep bidirectionality of the updates:
    - Input is passed at the forward pass,
    - Labels are passed at the backward pass.
- Metatrain to a given iteration (unroll).

SGD is a special case of two-state neurons

Backpropagation can be equivantly reformulated with generalized two-state neurons a_j^c, where j is a layer and $c \in \{0,1\}$ is a state.



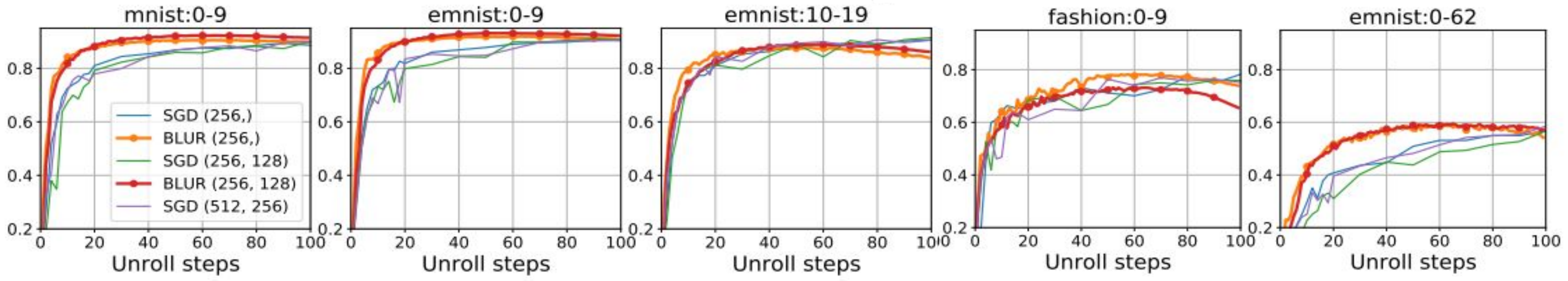
Bidirectional Learning Update Rules (BLUR)



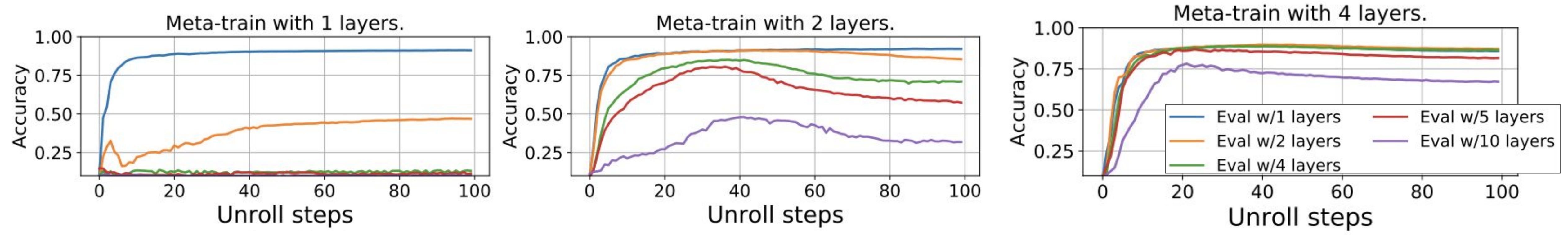
	Backpropagation/SGD	BLUR (Multi-state)
Forward	$a_j^c \leftarrow \phi^c\left(\sum_{i\in I(j),d} w_{ij} a_i^d \nu^{cd}\right)$	$a_j^c \leftarrow \sigma\left(f a_j^c + \eta \sum_{i,d} w_{ij}^c \nu^{cd} a_i^d\right)$
Backward	$a_i^{(2)} \leftarrow a_i^{(2)} \sum_{j\in J(i),d} w_{ij} a_j^d \mu^d$	$a_i^c \leftarrow \sigma\left(f a_i^c + \eta \sum_{j,d} w_{ji}^c \mu^{cd} a_j^d\right)$
Weight update	$w_{ij} \leftarrow w_{ij} - \tilde\eta \sum_{c,d} a_j^c \tilde\mu^c a_i^d \tilde\nu^d$	$w_{ij}^c \leftarrow \tilde f w_{ij}^c + \tilde\eta \sum_{e,d} a_i^e \tilde\nu^{ec} \cdot \tilde\mu^{cd} a_j^d$
States	- Two states neuron: $c, d \in \{1,2\}$ - Single state synapse.	- k neuron states. - k synapse states (possibly asymmetric).
Feedback	- Derivative of the loss function.	- Passed directly to the final layer.
Forward pass	- Both updates computes from the first state. - Different activation functions for each state.	- All states are updated via transform matrix ν^{cd}. - Same activation function for each state. - Forget f and update η are learned parameters.
Backward pass	- Second state update only multiplicatively. - Linear activation.	- All states are updated via transform matrix μ^{cd}. - Same activation for each state. - Forget f and update η are learned parameters.
Synapse update	- Second state of postsynaptic and first state of presynaptic. - Learning rate is a user parameter.	- All states from presynaptic and postsynaptic are mixed together via transform matrices $\tilde\nu^{cd}$ and $\tilde\mu^{cd}$. - Forget $\tilde f$ and update $\tilde\eta$ are learned parameters.

Generalization of a genome

- **Trained** on 10x10 MNIST using 2-layer 4-state architecture. **Validated** on 28x28 digits.



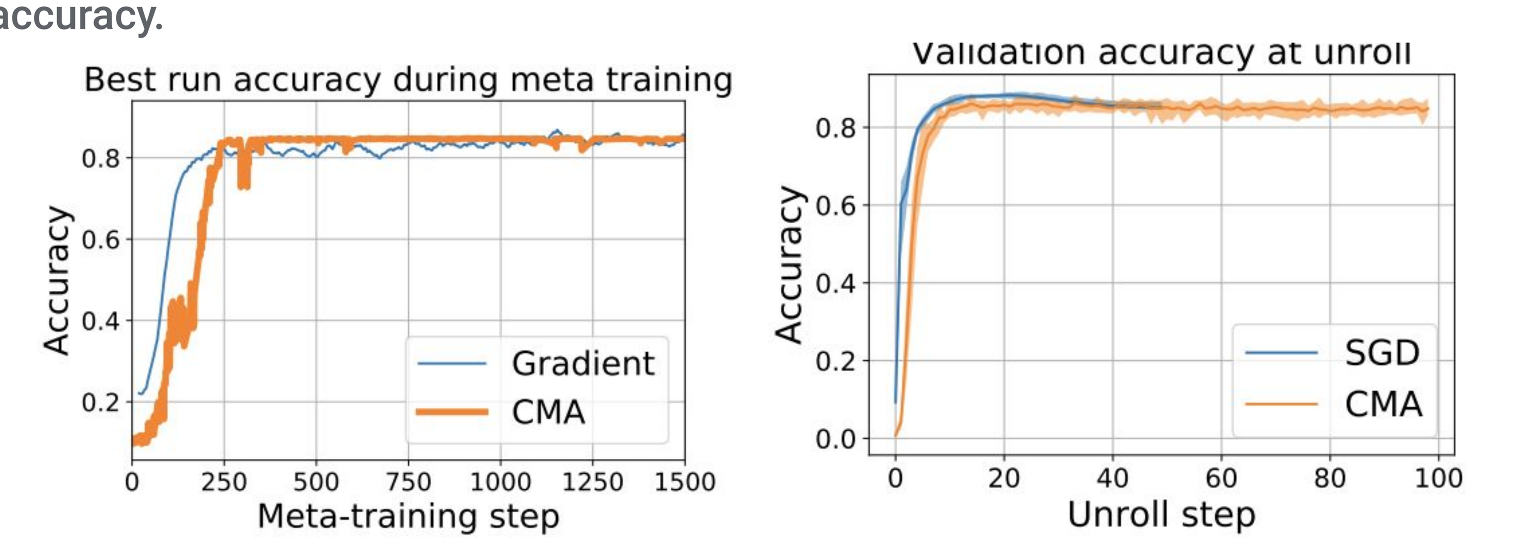
- Train networks with 1,2,4 layers to 10 untrolls and evaluated to 1,2,4,5,10 layers.



Meta-learning the genome

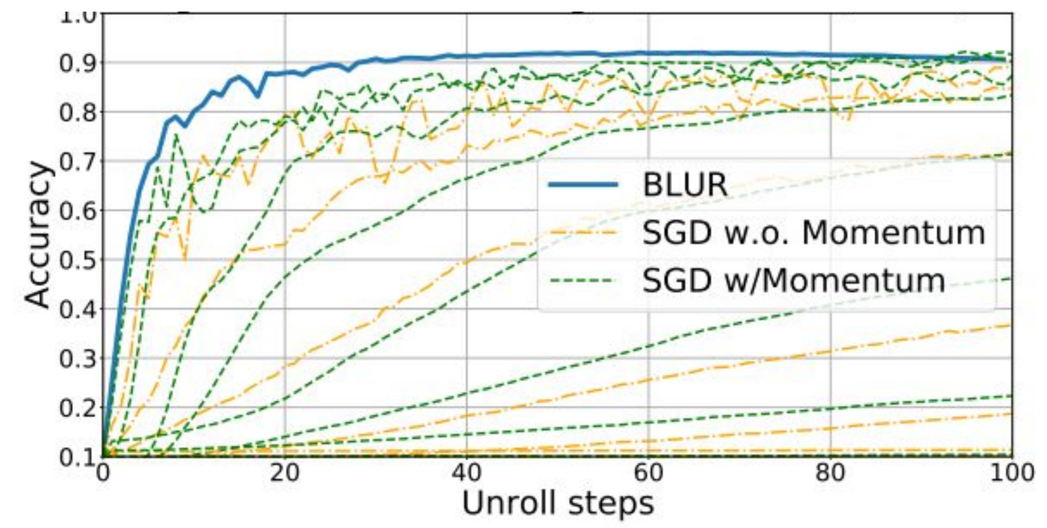
1. Start with a random genome
2. Repeat until meta-convergence:
    a. Apply forward/backward/synapse update for t unroll steps
    b. Measure the quality(*) of the learned synapses
    c. Meta-step: Update genome using ES or SGD

(*) quality can be any fitness functions, e.g. cross-entropy loss or validation accuracy.



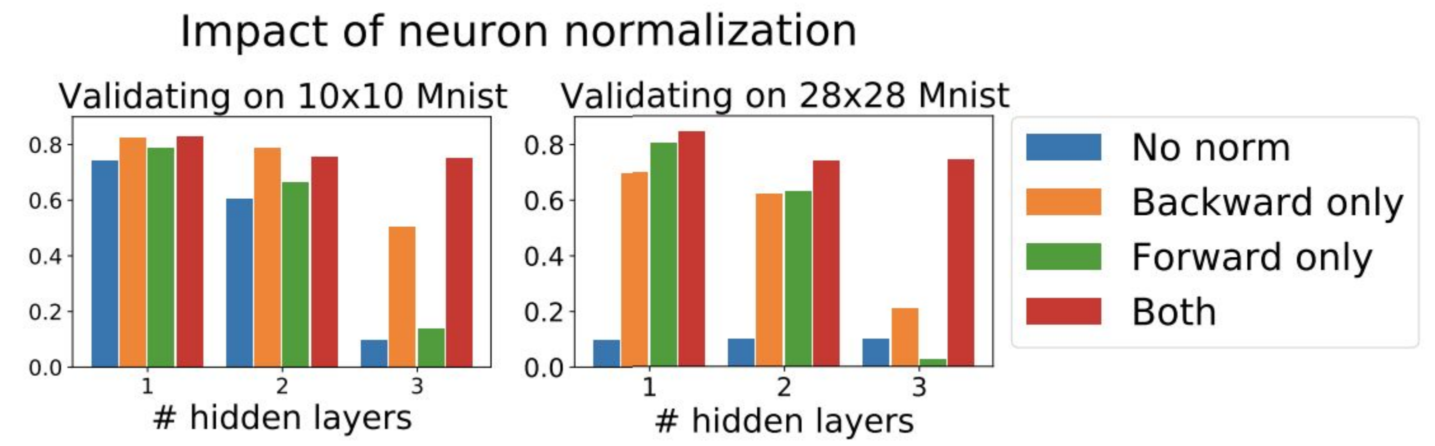
SGD w/ different parameters vs BLUR

Genome learns faster than SGD with any learning rate/momentum.



Role of normalization

Forward and backward (!!) activation normalization is important for good generalization.

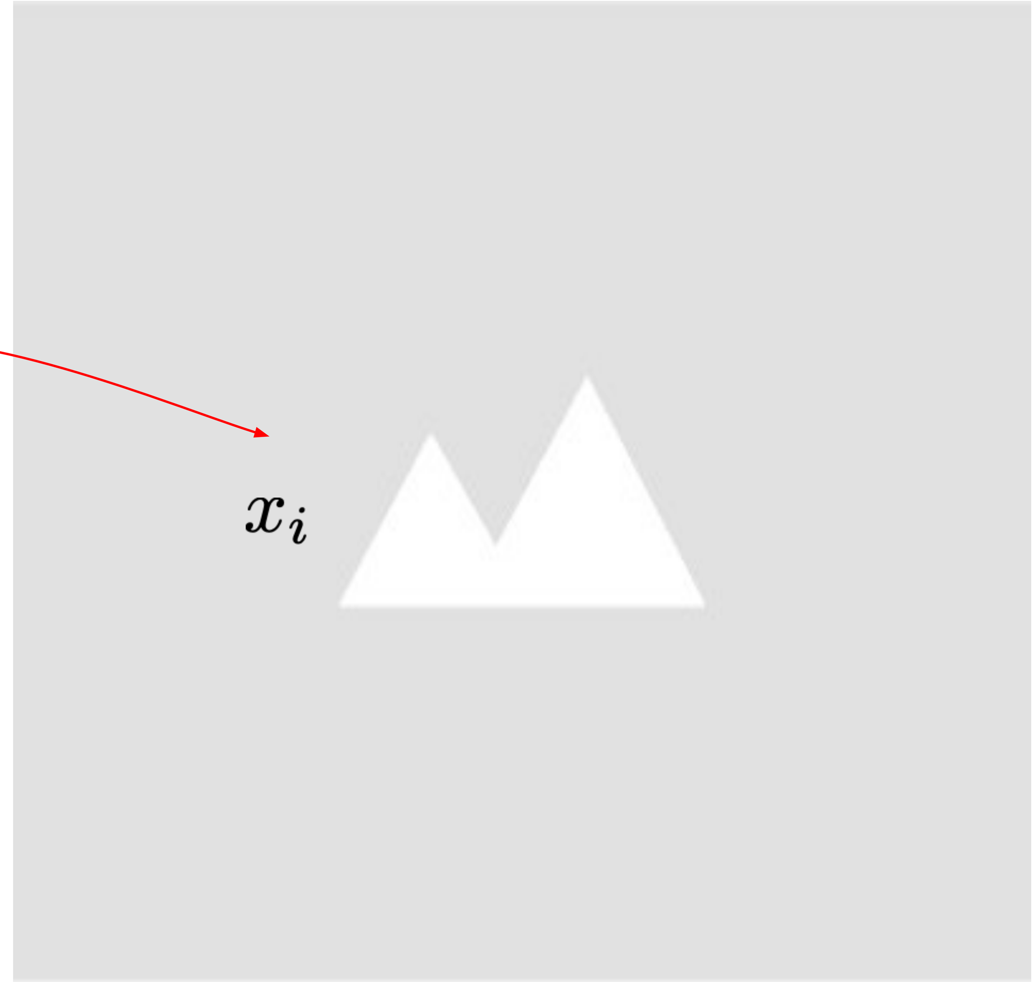


Paper: **Code:**

Playground

$\ldots \quad x_i \quad x_{i+1} \quad \ldots$

y_i



Every sequence presented to the Transformer consists of a dataset obtained by new random inputs and *teacher* and to compute the targets.

$\ldots \quad x_i \quad x_{i+1} \quad \ldots$

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:
$$W \leftarrow W + \Delta W$$
$$((W + \Delta W)x_i - y_i) = (W + (\Delta W x_i - y_i))$$

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!