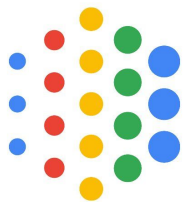
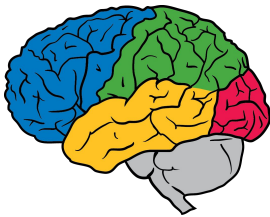
Towards General Purpose Learned Optimizers

Luke Metz (@Luke_Metz)





Outline

1. Motivation
2. Problem Setup
3. Task specific learned optimizers
4. General purpose learned optimizers
5. Future

Outline

1. **Motivation**
2. Problem Setup
3. Task specific learned optimizers
4. General purpose learned optimizers
5. Future

Deep learning

engineering features → learning features

SIFT (Lowe et. al. 1999)
HOG (Dalal et. al. 2005)

LeNet (LeCun et. al. 1998)
AlexNet (Krizhevsky et. al. 2012)

Deep learning

engineering features → learning features

SIFT (Lowe et. al. 1999)
HOG (Dalal et. al. 2005)

LeNet (LeCun et. al. 1998)
AlexNet (Krizhevsky et. al. 2012)

Meta learning

engineering to learn → learning to learn

SGD (Robbins et. al. 1951, Bottou 2010)
Autoencoders (Hinton et. al. 2006)

Learning To Learn (Hochreiter et. al. 2001)
Learned Optimizers (Andrychowicz et. al. 2016, Li et. al. 2016, Wichrowska et. al. 2017, Metz et. al. 2018, 2019)

Design at a higher level of abstraction

Existing methods

Currently:

- Meta-Learning ≈ few-shot learning
- Many do not generalize!
  - Trained and tested on narrow task distributions
  - Not flexible, will not generalize
  - Not general purpose tools

Goal: generalizable, reusable, robust learned learning algorithms

Trained once, applicable everywhere, no tuning necessary

Learned optimizer research

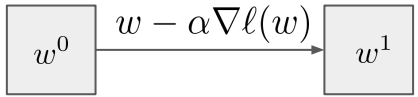
Optimizer

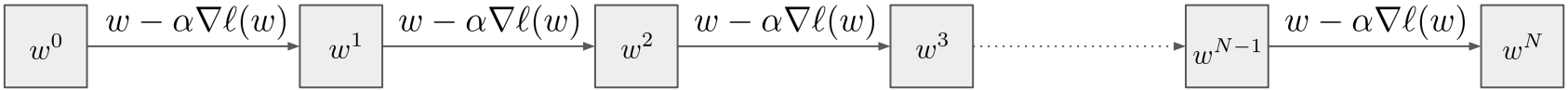
- Among the most widely used methods in ML++
- Current methods are hand designed
- Perfect test beds for meta-learning
    - Simple API -- gradients in, weight updates out

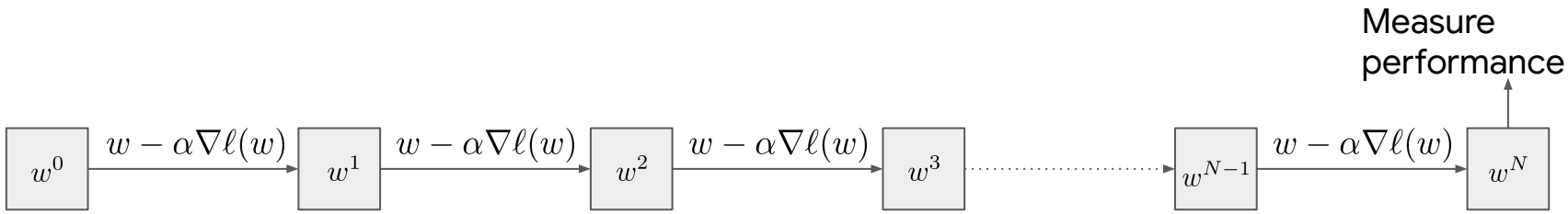
Outline

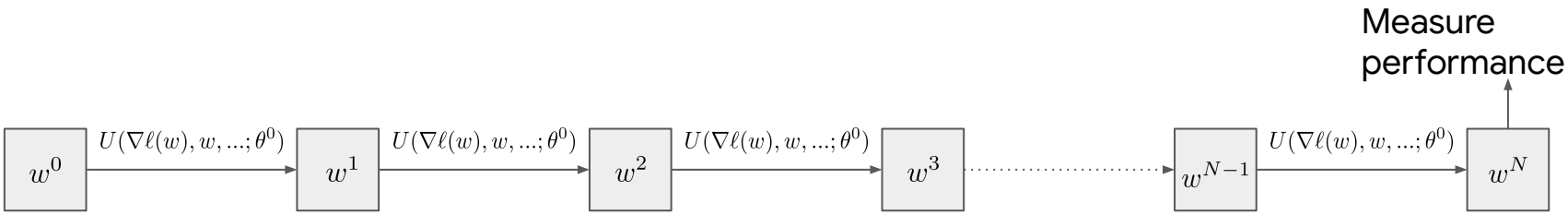
1. Motivation
2. **Problem Setup**
3. Task specific learned optimizers
4. General purpose learned optimizers
5. Future

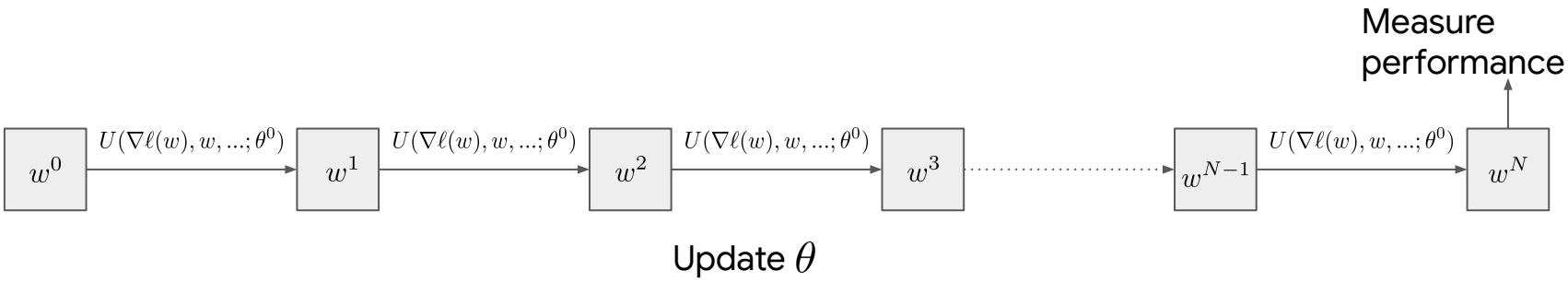

w^0

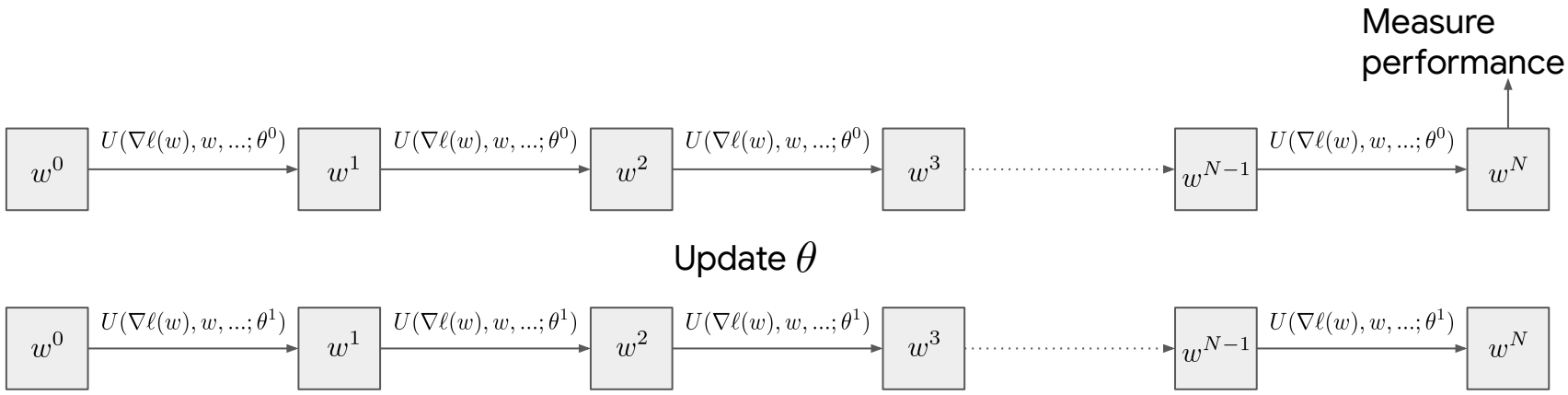

w^0
$w - \alpha\nabla\ell(w)$
w^1

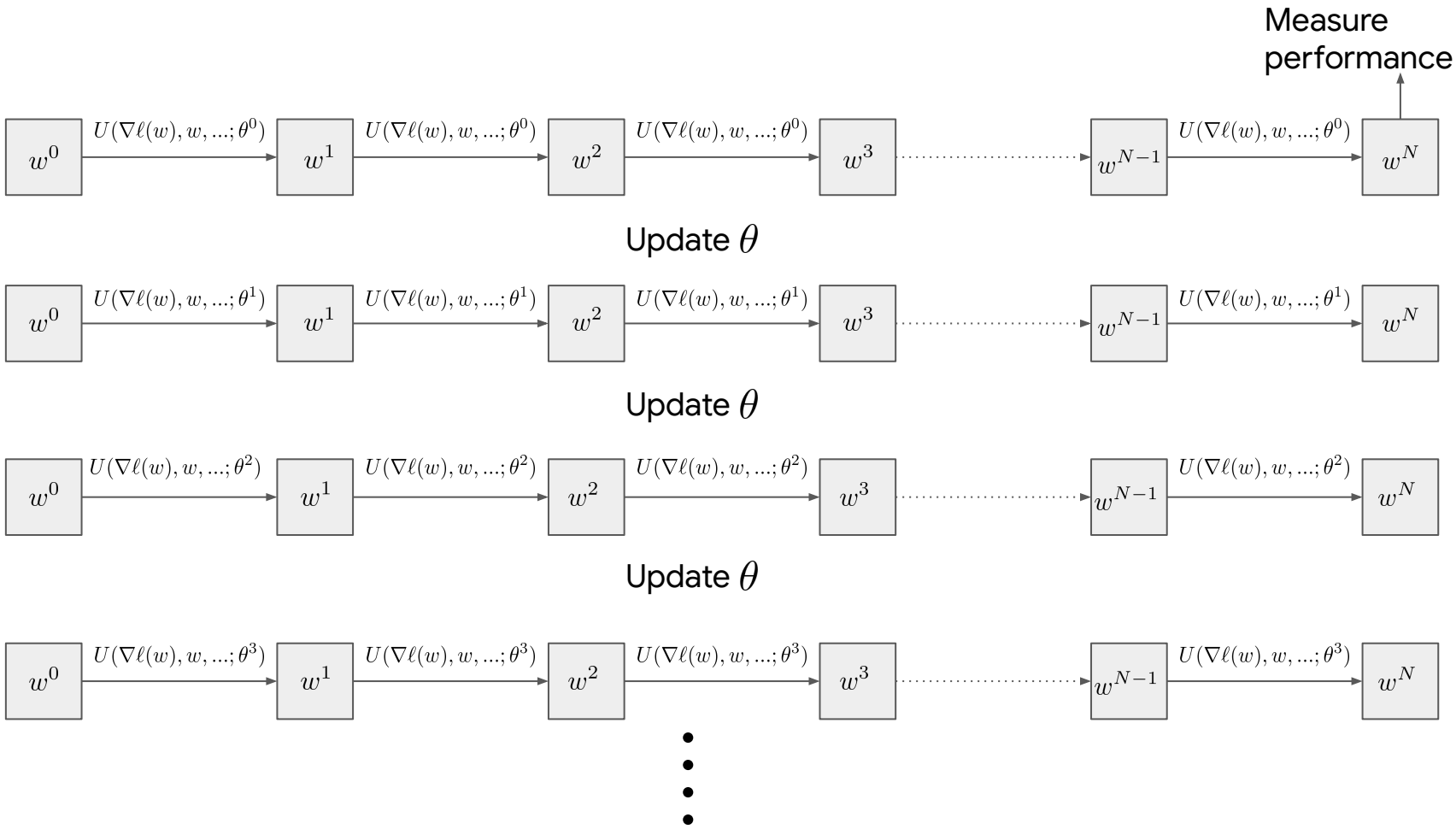

w^0
$w - \alpha\nabla\ell(w)$
w^1
$w - \alpha\nabla\ell(w)$
w^2
$w - \alpha\nabla\ell(w)$
w^3
w^{N-1}
$w - \alpha\nabla\ell(w)$
w^N

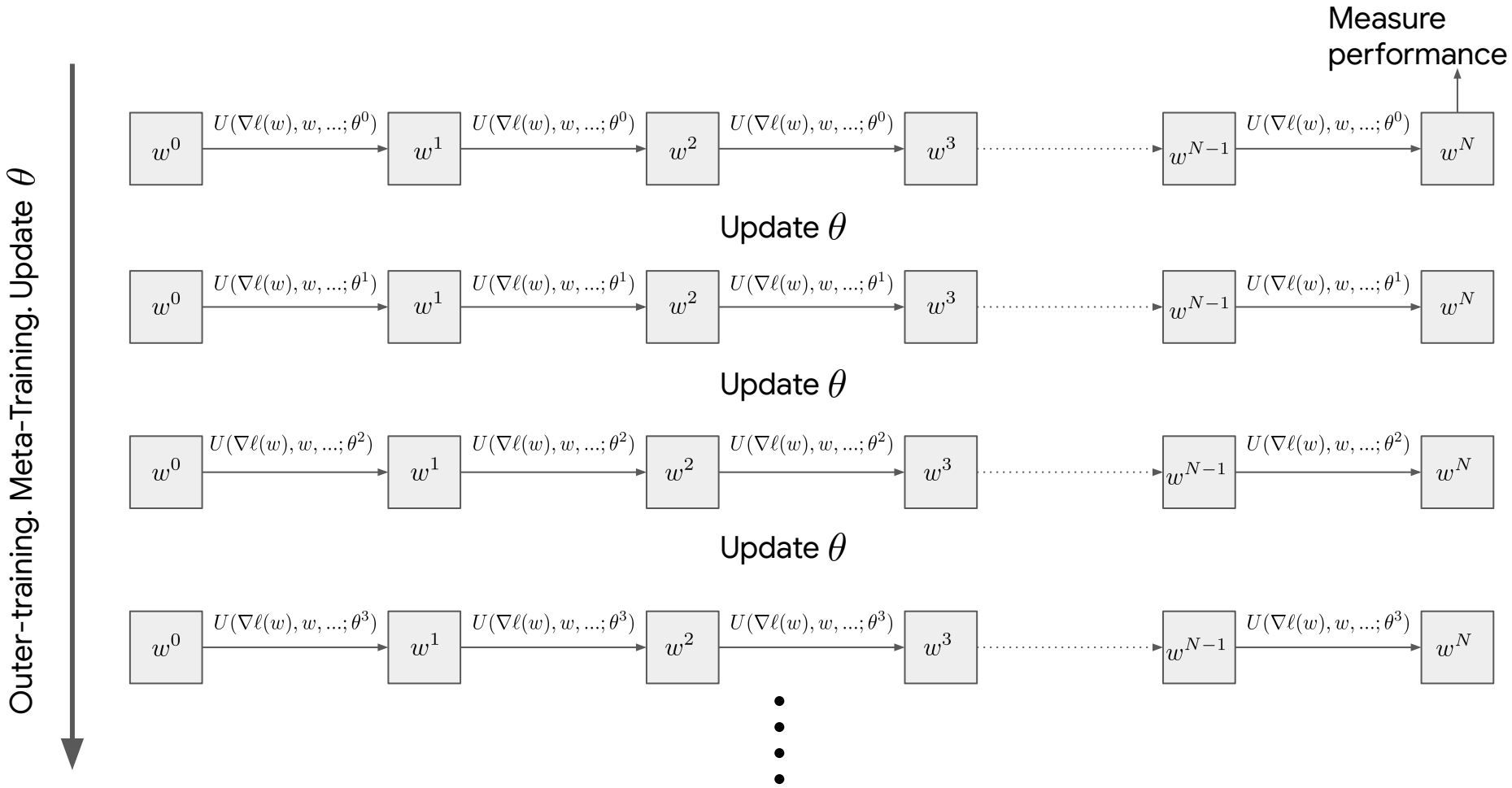

Measure performance
w^0
$w - \alpha \nabla \ell(w)$
w^1
$w - \alpha \nabla \ell(w)$
w^2
$w - \alpha \nabla \ell(w)$
w^3
w^{N-1}
$w - \alpha \nabla \ell(w)$
w^N

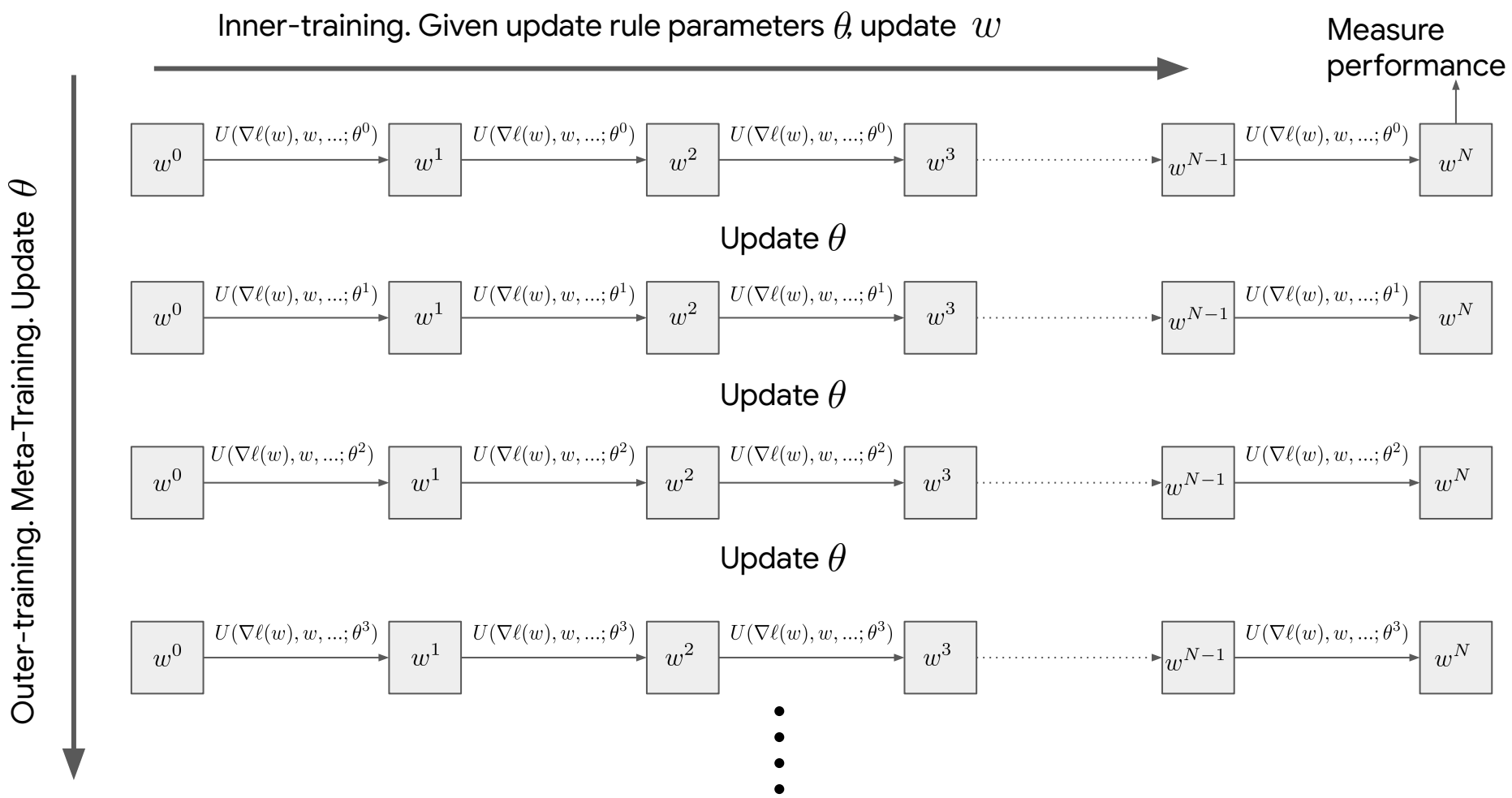

Measure performance
w^0
$U(\nabla\ell(w), w, ...; \theta^0)$
w^1
$U(\nabla\ell(w), w, ...; \theta^0)$
w^2
$U(\nabla\ell(w), w, ...; \theta^0)$
w^3
w^{N-1}
$U(\nabla\ell(w), w, ...; \theta^0)$
w^N

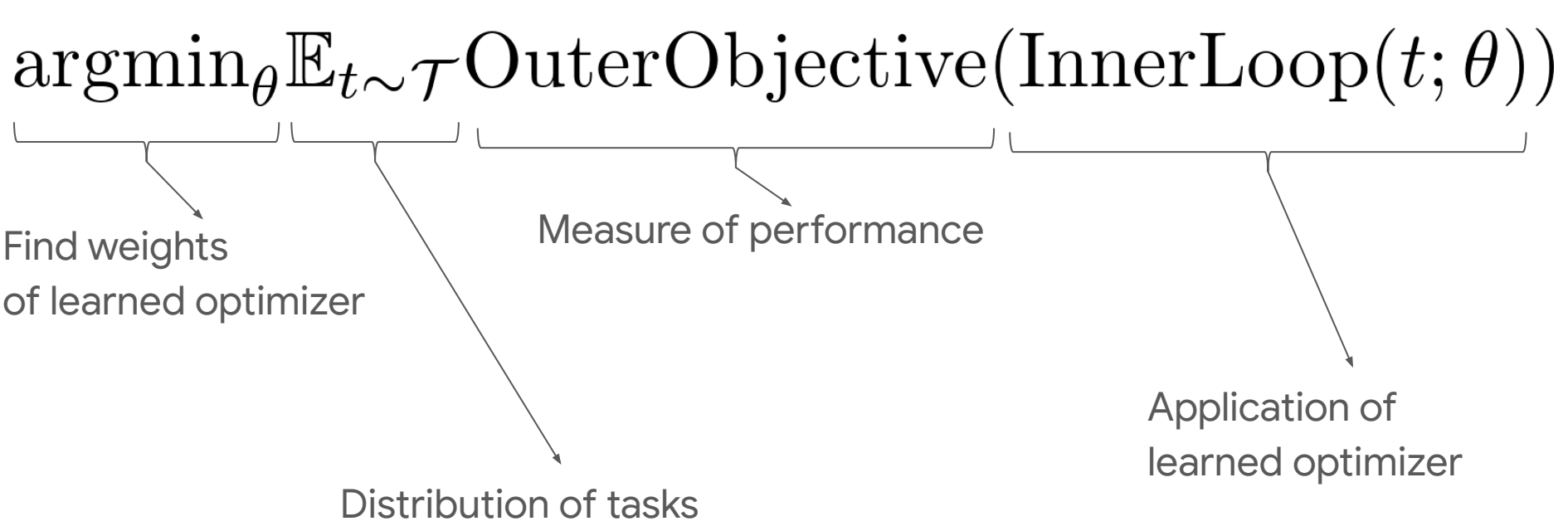

Measure performance
w^0
$U(\nabla\ell(w), w, ...; \theta^0)$
w^1
$U(\nabla\ell(w), w, ...; \theta^0)$
w^2
$U(\nabla\ell(w), w, ...; \theta^0)$
w^3
w^{N-1}
$U(\nabla\ell(w), w, ...; \theta^0)$
w^N
Update θ

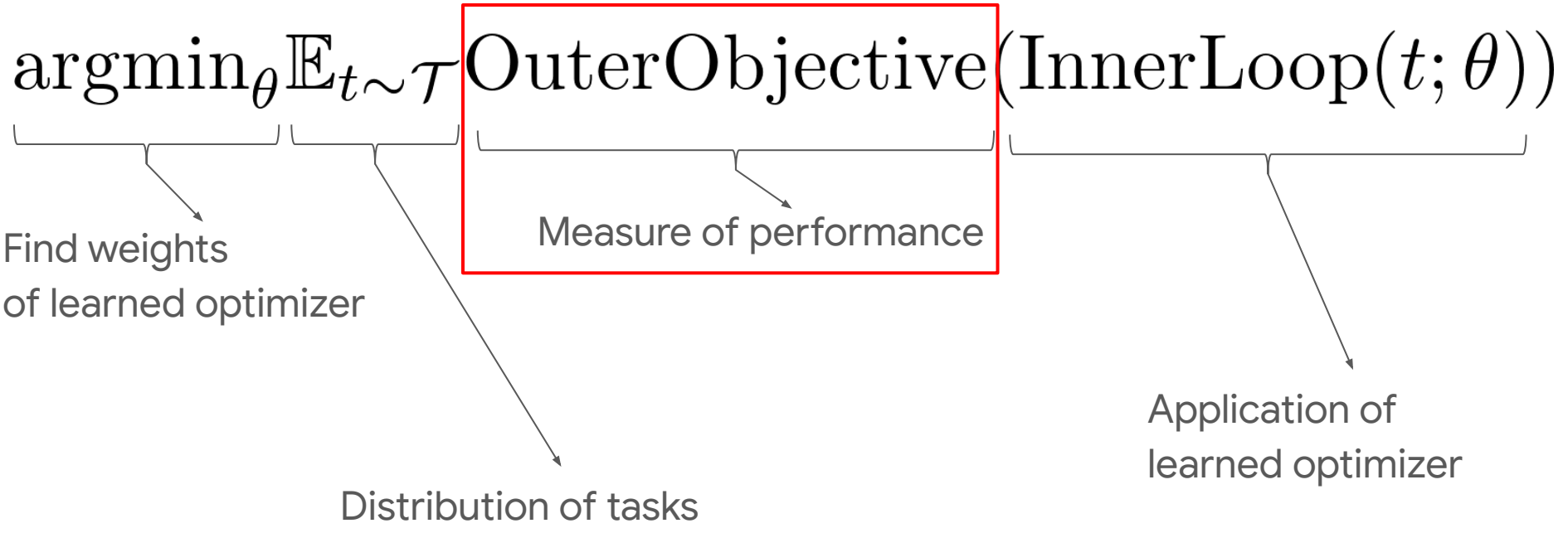

Measure performance
w^0
$U(\nabla\ell(w), w, ...; \theta^0)$
w^1
$U(\nabla\ell(w), w, ...; \theta^0)$
w^2
$U(\nabla\ell(w), w, ...; \theta^0)$
w^3
w^{N-1}
$U(\nabla\ell(w), w, ...; \theta^0)$
w^N
Update θ
w^0
$U(\nabla\ell(w), w, ...; \theta^1)$
w^1
$U(\nabla\ell(w), w, ...; \theta^1)$
w^2
$U(\nabla\ell(w), w, ...; \theta^1)$
w^3
w^{N-1}
$U(\nabla\ell(w), w, ...; \theta^1)$
w^N


Measure performance
w^0
$U(\nabla\ell(w), w, ...; \theta^0)$
w^1
$U(\nabla\ell(w), w, ...; \theta^0)$
w^2
$U(\nabla\ell(w), w, ...; \theta^0)$
w^3
w^{N-1}
$U(\nabla\ell(w), w, ...; \theta^0)$
w^N
Update θ
w^0
$U(\nabla\ell(w), w, ...; \theta^1)$
w^1
$U(\nabla\ell(w), w, ...; \theta^1)$
w^2
$U(\nabla\ell(w), w, ...; \theta^1)$
w^3
w^{N-1}
$U(\nabla\ell(w), w, ...; \theta^1)$
w^N
Update θ
w^0
$U(\nabla\ell(w), w, ...; \theta^2)$
w^1
$U(\nabla\ell(w), w, ...; \theta^2)$
w^2
$U(\nabla\ell(w), w, ...; \theta^2)$
w^3
w^{N-1}
$U(\nabla\ell(w), w, ...; \theta^2)$
w^N
Update θ
w^0
$U(\nabla\ell(w), w, ...; \theta^3)$
w^1
$U(\nabla\ell(w), w, ...; \theta^3)$
w^2
$U(\nabla\ell(w), w, ...; \theta^3)$
w^3
w^{N-1}
$U(\nabla\ell(w), w, ...; \theta^3)$
w^N


Measure performance
Outer-training. Meta-Training. Update θ
w^0
$U(\nabla\ell(w), w, ...; \theta^0)$
w^1
$U(\nabla\ell(w), w, ...; \theta^0)$
w^2
$U(\nabla\ell(w), w, ...; \theta^0)$
w^3
w^{N-1}
$U(\nabla\ell(w), w, ...; \theta^0)$
w^N
Update θ
w^0
$U(\nabla\ell(w), w, ...; \theta^1)$
w^1
$U(\nabla\ell(w), w, ...; \theta^1)$
w^2
$U(\nabla\ell(w), w, ...; \theta^1)$
w^3
w^{N-1}
$U(\nabla\ell(w), w, ...; \theta^1)$
w^N
Update θ
w^0
$U(\nabla\ell(w), w, ...; \theta^2)$
w^1
$U(\nabla\ell(w), w, ...; \theta^2)$
w^2
$U(\nabla\ell(w), w, ...; \theta^2)$
w^3
w^{N-1}
$U(\nabla\ell(w), w, ...; \theta^2)$
w^N
Update θ
w^0
$U(\nabla\ell(w), w, ...; \theta^3)$
w^1
$U(\nabla\ell(w), w, ...; \theta^3)$
w^2
$U(\nabla\ell(w), w, ...; \theta^3)$
w^3
w^{N-1}
$U(\nabla\ell(w), w, ...; \theta^3)$
w^N


Inner-training. Given update rule parameters θ, update w
Measure performance
Outer-training. Meta-Training. Update θ
w^0
w^1
w^2
w^3
w^{N-1}
w^N
$U(\nabla\ell(w), w, ...; \theta^0)$
$U(\nabla\ell(w), w, ...; \theta^1)$
$U(\nabla\ell(w), w, ...; \theta^2)$
$U(\nabla\ell(w), w, ...; \theta^3)$
Update θ
Update θ
Update θ


$\text{argmin}_\theta \mathbb{E}_{t\sim\mathcal{T}} \text{OuterObjective}(\text{InnerLoop}(t;\theta))$
Find weights of learned optimizer
Distribution of tasks
Measure of performance
Application of learned optimizer


$\text{argmin}_\theta \mathbb{E}_{t\sim\mathcal{T}} \text{OuterObjective}(\text{InnerLoop}(t;\theta))$
Find weights of learned optimizer
Distribution of tasks
Measure of performance
Application of learned optimizer

Measurement of optimizer performance

- Target the objective we care about

Measurement of optimizer performance

Optimize our optimizer for:

- Training loss?

Measurement of optimizer performance

Optimize our optimizer for:

- Training loss?
- Validation loss?

Measurement of optimizer performance

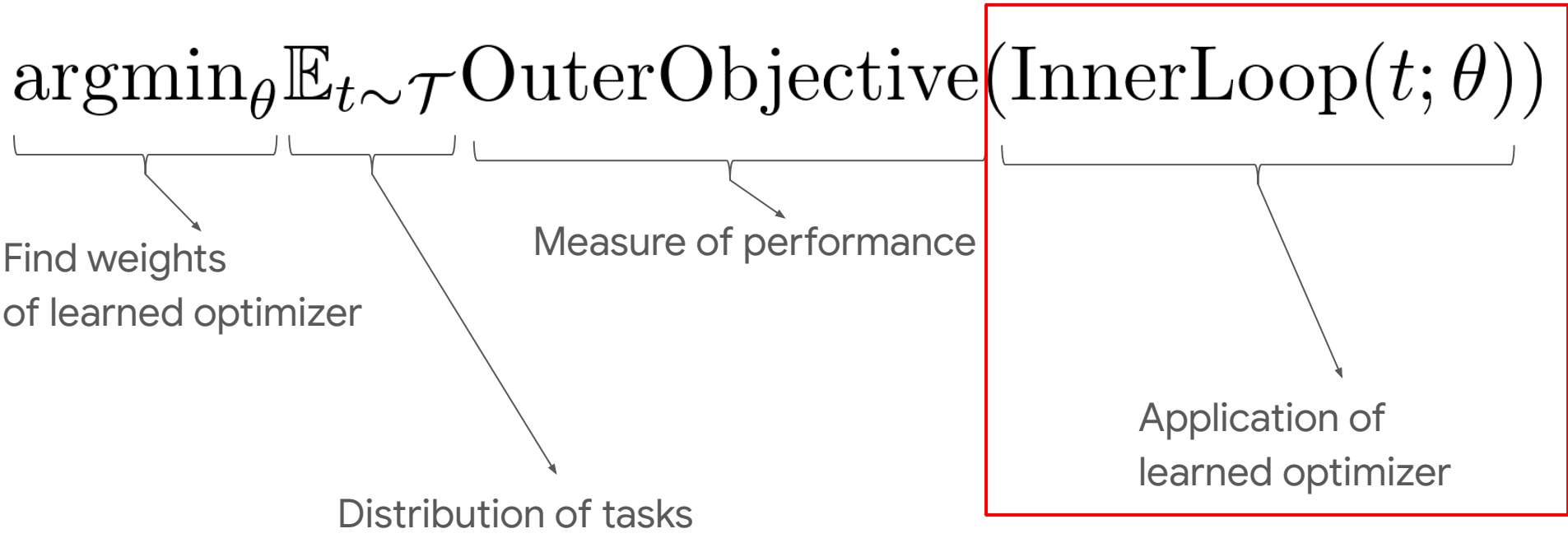
Optimize our optimizer for:

- Training loss?
- Validation loss?
- Performance on different distributions?
  - e.g. for robustness / out of distribution generalization?

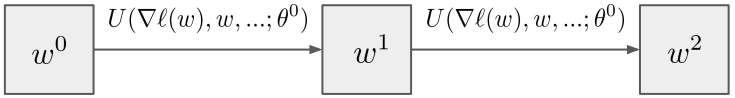
Using learned optimizers to make models robust to input noise

---

Luke Metz [1] **Niru Maheswaranathan** [1] **Jonathon Shlens** [1] **Jascha Sohl-Dickstein** [1] **Ekin D. Cubuk** [1]


$\text{argmin}_\theta \mathbb{E}_{t\sim\mathcal{T}} \text{OuterObjective}(\text{InnerLoop}(t;\theta))$
Find weights of learned optimizer
Distribution of tasks
Measure of performance
Application of learned optimizer

Parametric update (U): Per parameter NN

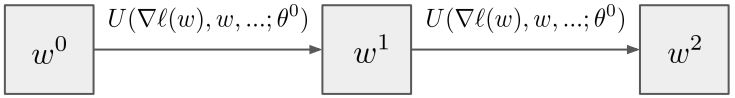


Operates on [n_parameters, 1]

Andrychowicz, Marcin, et al. "Learning to learn by gradient descent by gradient descent." *Neurips*. 2016.

Metz, Luke, et al. "Understanding and correcting pathologies in the training of learned optimizers." *ICML 2019*

Parametric update (U): Per parameter NN



Operates on [n_parameters, 1]

Andrychowicz, Marcin, et al. "Learning to learn by gradient descent by gradient descent." *Neurips*. 2016.

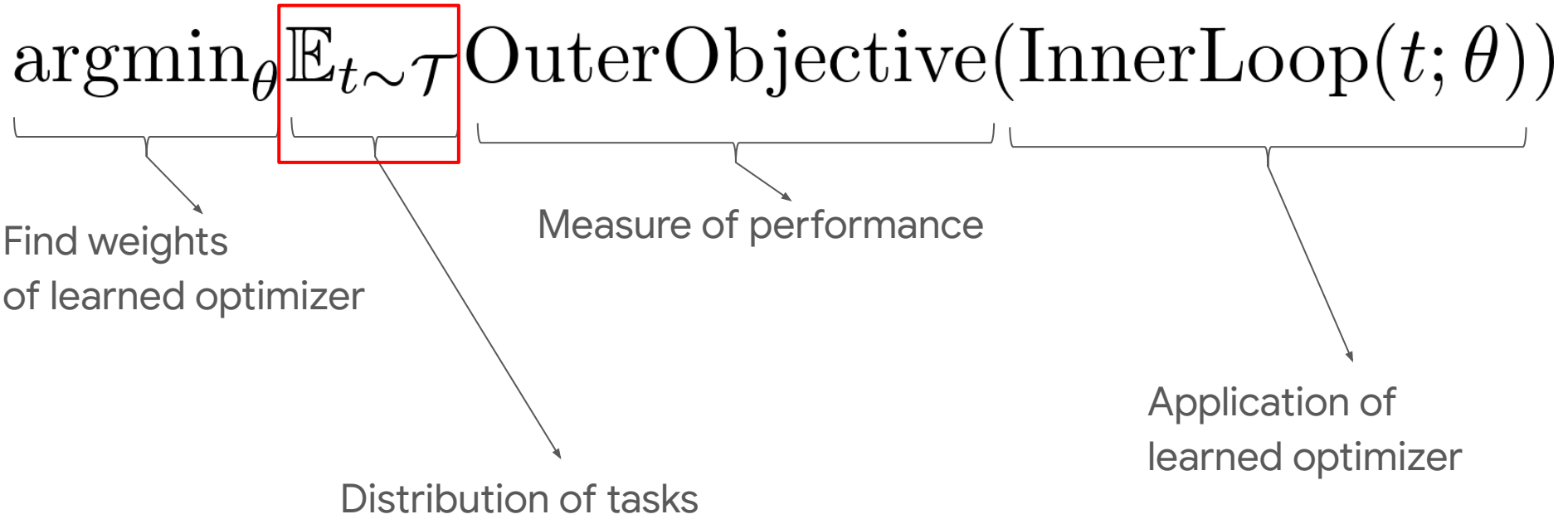
Metz, Luke, et al. "Understanding and correcting pathologies in the training of learned optimizers." *ICML 2019*

For each target model weight matrix.
Shape of gradient: [n, m]

→ Reshape gradient: [n*m, 1]

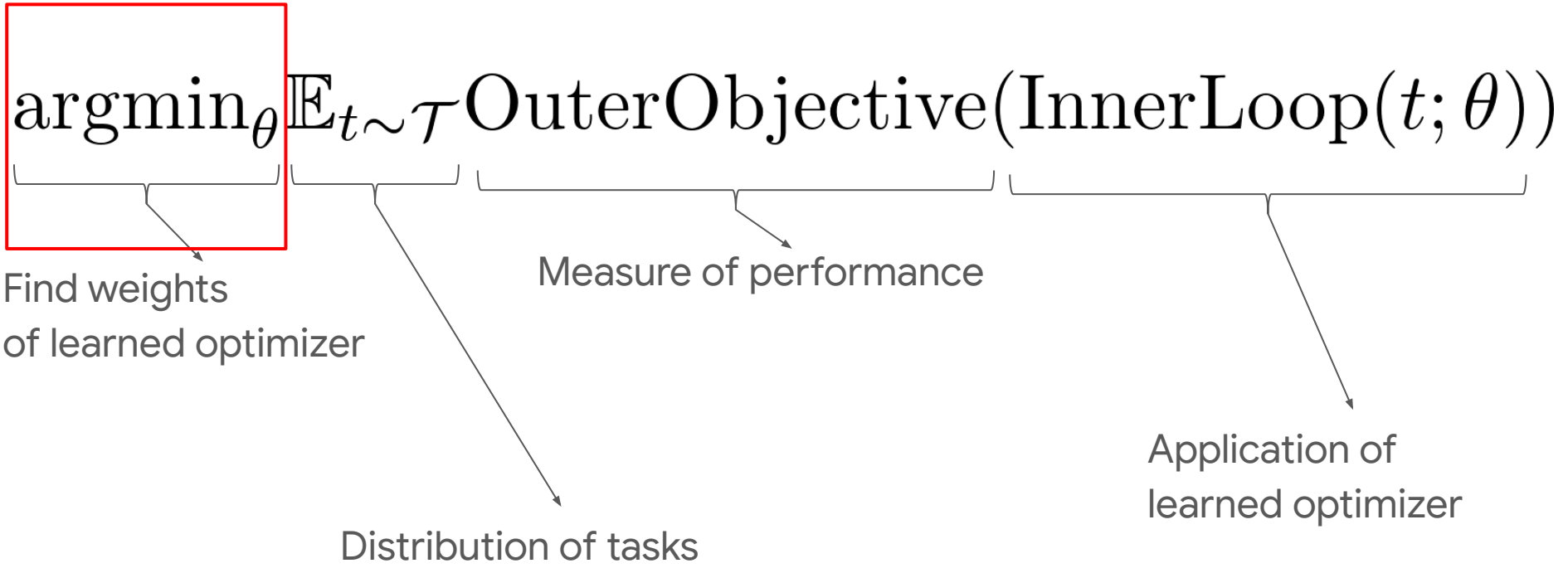
→ Apply the same network on batch:
Input: size 1
Output: size 1

→ Reshape [n, m] and apply weight update


$\text{argmin}_\theta \mathbb{E}_{t\sim\mathcal{T}} \text{OuterObjective}(\text{InnerLoop}(t;\theta))$
Find weights of learned optimizer
Distribution of tasks
Measure of performance
Application of learned optimizer

Distribution of tasks

- Defines the types of tasks learned optimizer performs well on.
- No free lunch for optimization
  - All possible tasks --> no gains
  - Narrow distribution -> more gains


$\text{argmin}_\theta \mathbb{E}_{t\sim\mathcal{T}} \text{OuterObjective}(\text{InnerLoop}(t;\theta))$
Find weights of learned optimizer
Distribution of tasks
Measure of performance
Application of learned optimizer

Outer-optimization is hard

- **Expensive** to compute a single outer-function evaluation
- Outer-function evaluations are **noisy**
- Outer-function is **complex** [U(U(U(U(.....))))]

Meta-learning based approaches: limited success & require specialized "tricks"

Outer Training Methods

Black box

- Random search
- Hyperparameter optimization
- Reinforcement learning
- Evolution

Outer Training Methods

Black box

- Random search
- Hyperparameter optimization
- Reinforcement learning
- Evolution

Gradients

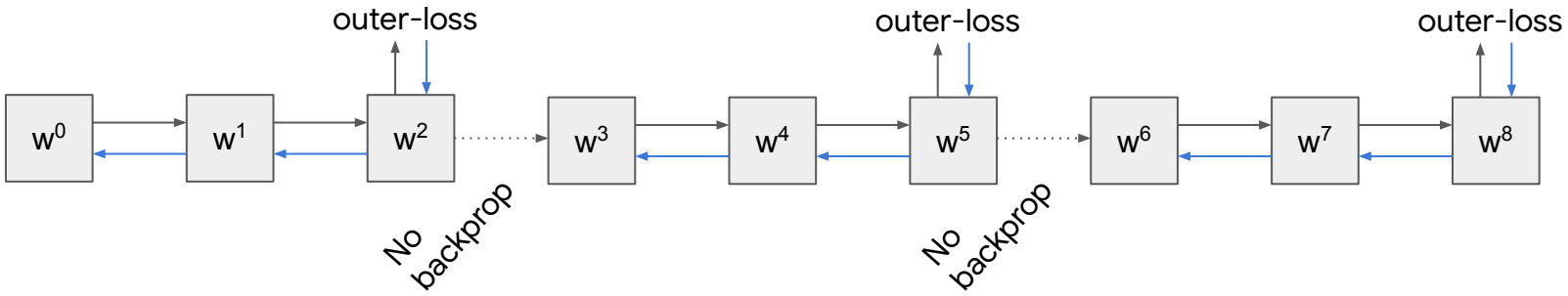
Whole training process is differentiable

- U(U(U...;θ);θ);θ)

"Unroll" optimization -> compute gradients -> SGD

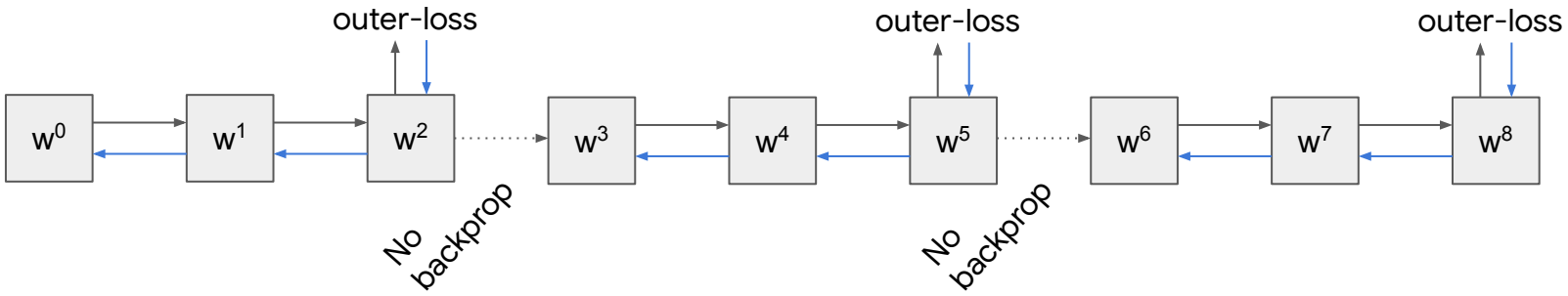
Truncated Backpropagation Through Time

- Split sequence into multiple pieces



Truncated Backpropagation Through Time

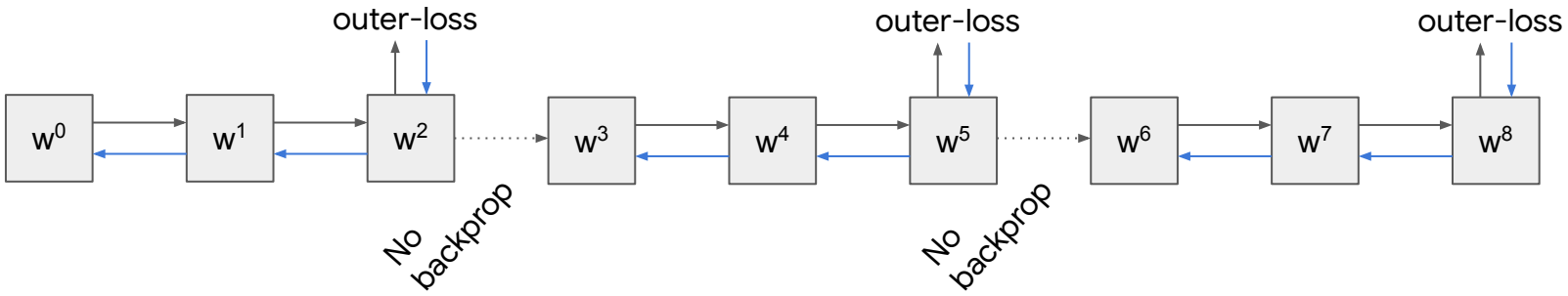
- Split sequence into multiple pieces



- More gradients (one computed per truncation)
- Better behaved loss surface

Truncated Backpropagation Through Time

- Split sequence into multiple pieces



- More gradients (one computed per truncation)
- Better behaved loss surface
- Biased!

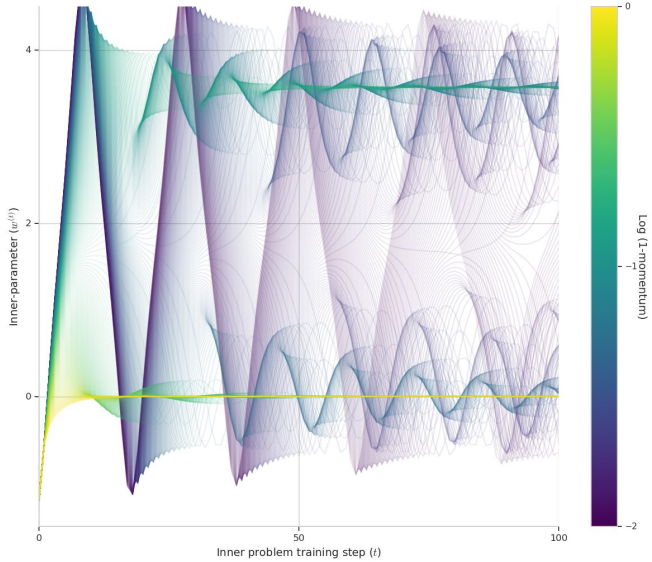
Conflicting Goals in Unroll Lengths

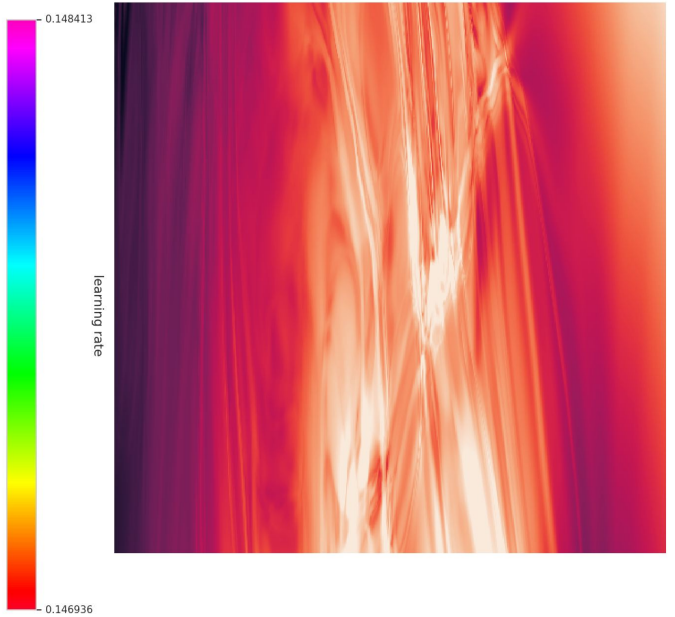
- Long unroll steps per truncation
  - Less biased
  - Each iteration is slower
  - Horrible loss surface -> exploding gradients

- Short unroll steps per truncation
  - Fast to train
  - Well behaved loss surfaces
  - More biased

Understanding and correcting pathologies in the training of learned optimizers

Luke Metz [1] **Niru Maheswaranathan** [1] **Jeremy Nixon** [1] **C. Daniel Freeman** [1] **Jascha Sohl-Dickstein** [1]





Evolutionary Strategies / Variational optimization

- Define a distribution over learned optimizer parameters
- Optimize the parameters of this distribution

$$\mathcal{L}(\theta) = \mathbb{E}_{\tilde{\theta} \sim \mathcal{N}(\theta, \sigma^2 I)} \left[L\left(\tilde{\theta}\right) \right]$$

Sample from distribution

Parameters of the distribution

Original loss

$$\mathcal{L}(\theta) = \mathbb{E}_{\tilde{\theta} \sim \mathcal{N}(\theta, \sigma^2 I)} \left[L\left(\tilde{\theta}\right) \right]$$

Two unbiased gradient estimators

$$\nabla_{\tilde{\theta}} \mathcal{L}(\tilde{\theta}) = ?$$

Evolutionary Strategies (ES)
Score Function

$$g_{\text{es}} = \frac{1}{S} \sum_{s} L\left(\tilde{\theta}_s\right) \nabla_{\theta} \left[log\left(N\left(\tilde{\theta}_s; \theta, \sigma^2 I\right)\right)\right]$$

$$\tilde{\theta}_s \sim N\left(\theta, \sigma^2 I\right)$$

$$\mathcal{L}(\theta) = \mathbb{E}_{\tilde{\theta} \sim \mathcal{N}(\theta, \sigma^2 I)} \left[L\left(\tilde{\theta}\right) \right]$$

Two unbiased gradient estimators

$$\nabla_{\tilde{\theta}} \mathcal{L}(\tilde{\theta}) = ?$$

Evolutionary Strategies (ES)
Score Function

$$g_{\text{es}} = \frac{1}{S} \sum_s L\left(\tilde{\theta}_s\right) \nabla_\theta \left[log\left(N\left(\tilde{\theta}_s; \theta, \sigma^2 I\right)\right)\right]$$

$$\tilde{\theta}_s \sim N\left(\theta, \sigma^2 I\right)$$

Reparameterization Gradient (RP)

$$g_{\text{rp}} = \frac{1}{S} \sum_s \nabla_\theta L\left(\theta + \sigma n_s\right)$$

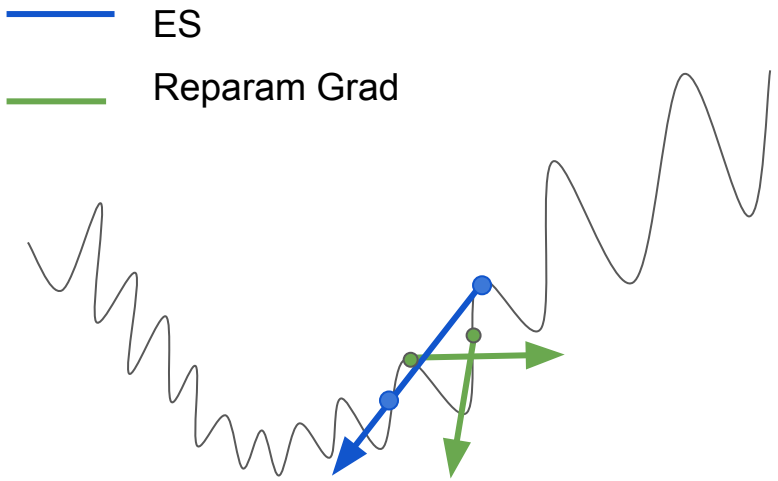
$$n_s \sim N\left(0, I\right)$$

Different gradient variance

- Smooth loss surface -> reparameterization has lower variance
- High curvature loss surface -> ES has lower variance

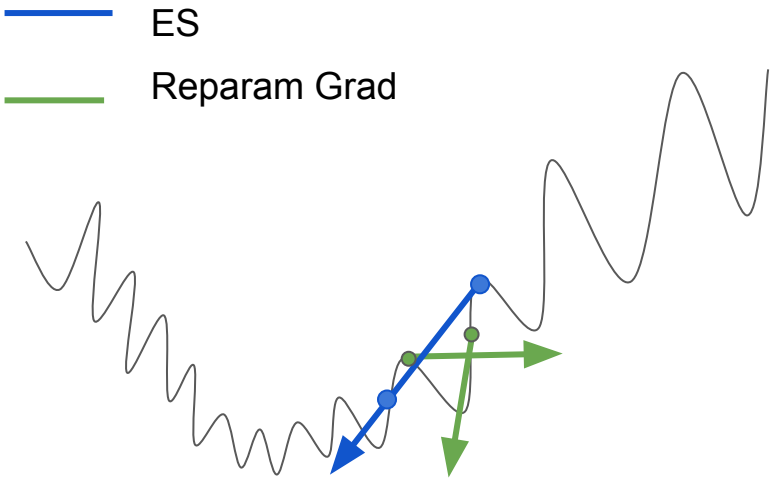
Different gradient variance

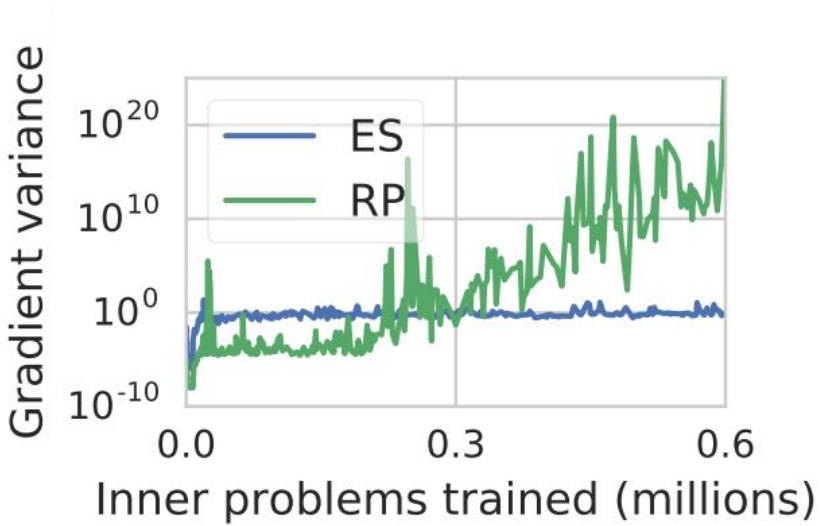
- Smooth loss surface -> reparameterization has lower variance
- High curvature loss surface -> ES has lower variance



Different gradient variance

- Smooth loss surface -> reparameterization has lower variance
- High curvature loss surface -> ES has lower variance





Outline

1. Motivation
2. Problem Setup
3. **Task specific learned optimizers**
4. General purpose learned optimizers
5. Future

Task specific learned optimizers

- Is research into learned optimizers worth it?

Understanding and correcting pathologies in the training of learned optimizers

Luke Metz [1] **Niru Maheswaranathan** [1] **Jeremy Nixon** [1] **C. Daniel Freeman** [1] **Jascha Sohl-Dickstein** [1]

Task specific learned optimizers

- Is research into learned optimizers worth it?

$$\underbrace{\text{argmin}_\theta}\underbrace{\mathbb{E}_{t\sim\mathcal{T}}}\underbrace{\text{OuterObjective}}\underbrace{(\text{InnerLoop}(t;\theta))}$$

Evolutionary strategies + backoprop gradients

10 way classification problems sampled from imagenet
small 3 hidden layer convnet

Training and Validation loss

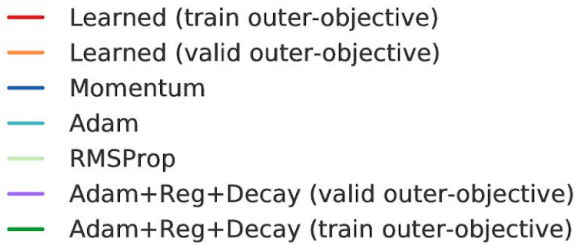
Per Param MLP optimizer for 10k iterations

Understanding and correctness pathologies in the training of learned optimizers

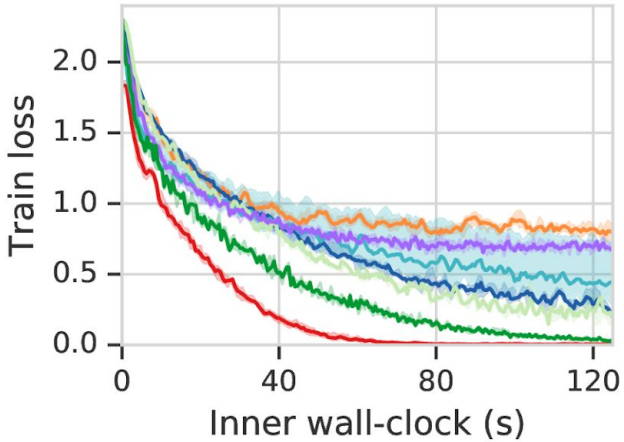
---

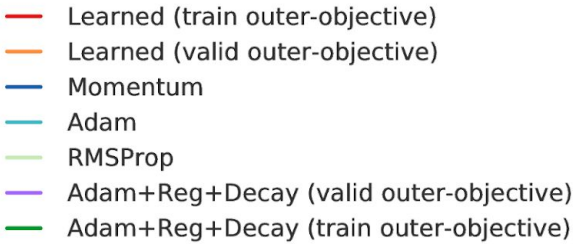
Luke Metz [1] **Niru Maheswaranathan** [1] **Jeremy Nixon** [1] **C. Daniel Freeman** [1] **Jascha Sohl-Dickstein** [1]

Outer-test example problem

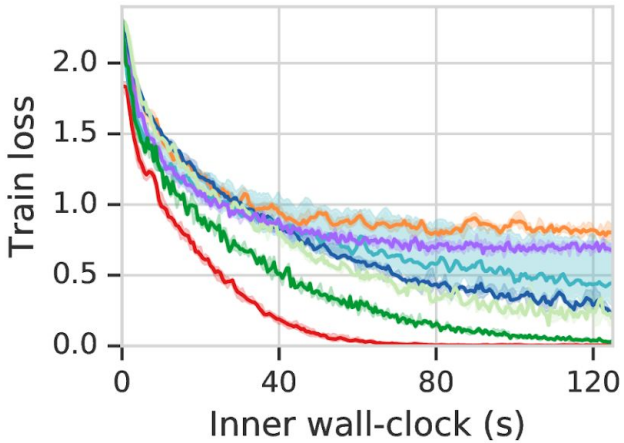


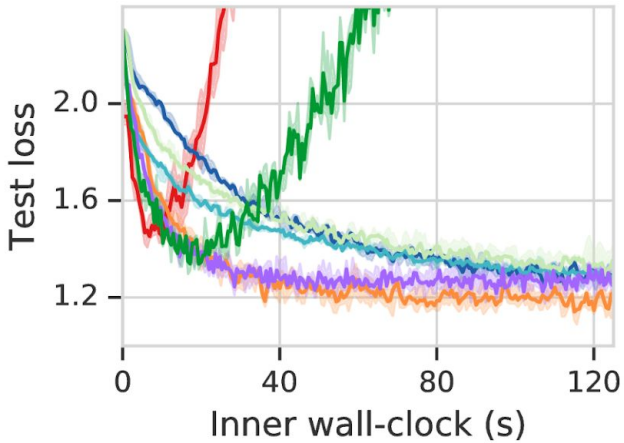
Outer-test example problem

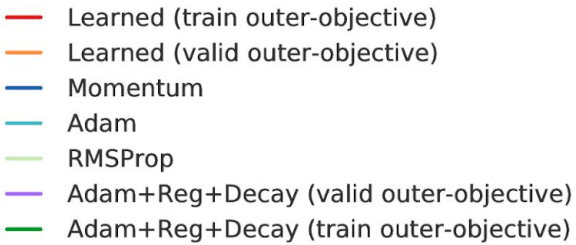




Outer-test example problem







Outline

1. Motivation
2. Problem Setup
3. Task specific learned optimizers
4. **General purpose learned optimizers**
5. Future

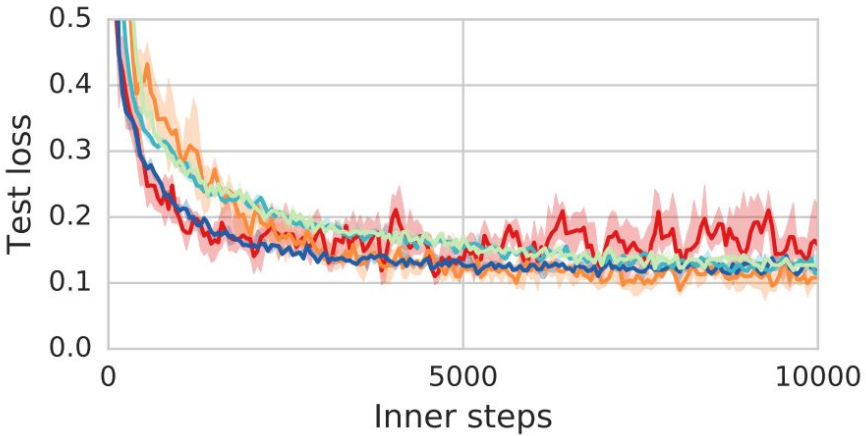
General purpose == Out of distribution generalization

- Works out-the-box on new tasks
- No tuning necessary

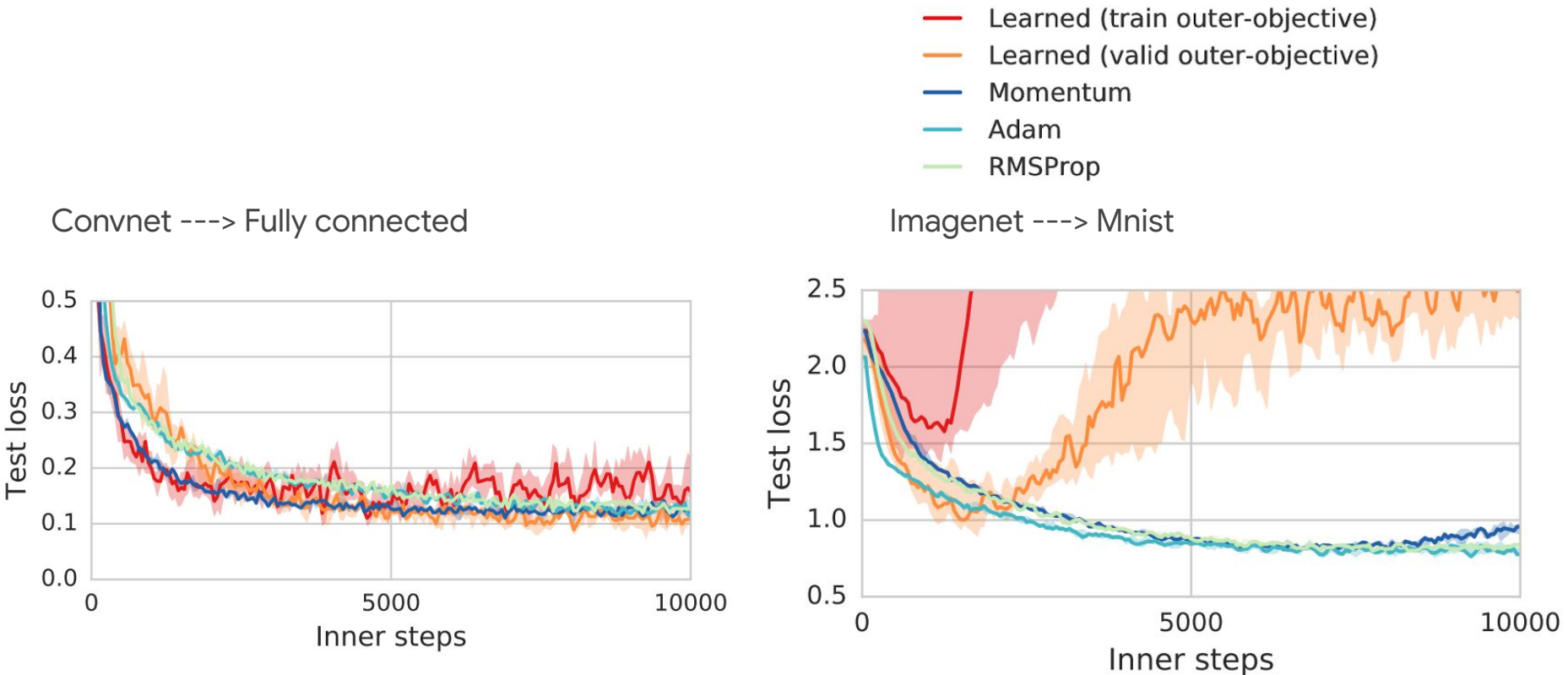
Out of distribution generalization

Learned (train outer-objective)
Learned (valid outer-objective)
Momentum
Adam
RMSProp

Convnet ---> Fully connected



Out of distribution generalization



Training on 1 task....

Training on 1 task....

Let's 5,000x that

TaskSet: A dataset of tasks

Using a thousand optimization tasks to learn hyperparameter search strategies

Luke Metz [1] Niru Maheswaranathan [1] Ruoxi Sun [1] C. Daniel Freeman [1] Ben Poole [1] Jascha Sohl-Dickstein [1]

Samplers

```python
@registry.task_registry.register_sampler("conv_pooling_family")
def sample_conv_pooling_family_cfg(seed: int) -> ConvPoolingConfig:
  """Sample a task config for a conv net with pooling model on image da

  These configs are nested python structures that provide enough inform
  to create an instance of the problem.

  Args:
    seed: int Random seed to generate task from.

  Returns:
    A nested dictionary containing a configuration.
  """
  rng = np.random.RandomState(seed)
  cfg = {}
  layer_choices = [1, 2, 3, 4, 5]
  max_layer = np.max(layer_choices)
  n_layers = rng.choice(layer_choices)
  # pattern for how strides are chosen. Either all 2s, repeated 1,2
  # or repeated 2,1.
  stride_pattern = rng.choice(["all_two", "one_two", "two_one"])
  if stride_pattern == "all_two":
    cfg["strides"] = ([2] * max_layer)[0:n_layers]
  elif stride_pattern == "one_two":
    cfg["strides"] = ([1, 2] * max_layer)[0:n_layers]
  elif stride_pattern == "two_one":
    cfg["strides"] = ([2, 1] * max_layer)[0:n_layers]
  cfg["strides"] = list(zip(cfg["strides"], cfg["strides"]))
  cfg["hidden_units"] = [
      utils.sample_log_int(rng, 8, 64) for _ in range(n_layers)
  ]

  cfg["activation"] = utils.sample_activation(rng)
  cfg["w_init"] = utils.sample_initializer(rng)
  cfg["padding"] = [
      str(rng.choice([snt.SAME, snt.VALID])) for _ in range(n_layers)
  ]
  cfg["pool_type"] = str(rng.choice(["mean", "max", "squared_mean"]))
  cfg["use_bias"] = bool(rng.choice([True, False]))
  cfg["dataset"] = utils.sample_image_dataset(rng)
  cfg["center_data"] = bool(rng.choice([True, False]))
  return cfg
```

Config

```
{
  "strides": [
    [2, 2]
    [2, 2],
    [2, 2],
  ],
  "hidden_units": [15, 47, 22, 28],
  "activation": "tanh",
  "w_init": [
    "he_uniform",
    null
  ],
  "padding": ["SAME", "SAME", "VALID"],
  "pool_type": "max",
  "use_bias": true,
  "dataset": [
    "food101_32x32",
    {
      "bs": 8,
      "just_train": false,
      "num_train": null
    },
    null
  ],
  "center_data": true
}
```

Code

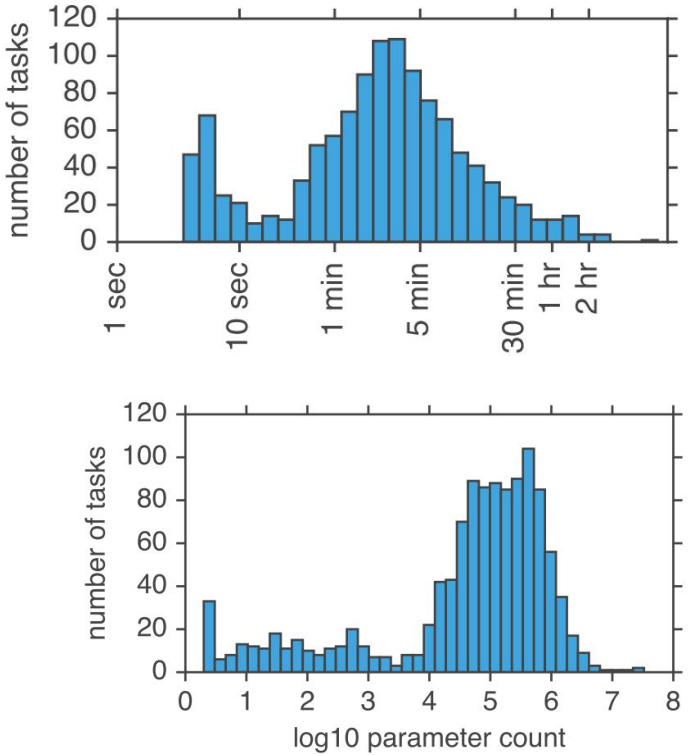
```python
def _build(batch):
  """Builds the sonnet module."""
  image = utils.maybe_center(cfg["center_dat

  net = snt.nets.ConvNet2D(
      hidden_units,
      kernel_shapes=[(3, 3)],
      strides=cfg["strides"],
      paddings=cfg["padding"],
      activation=act_fn,
      use_bias=cfg["use_bias"],
      initializers=init,
      activate_final=True)(
          image)
  if cfg["pool_type"] == "mean":
    net = tf.reduce_mean(net, axis=[1, 2])
  elif cfg["pool_type"] == "max":
    net = tf.reduce_max(net, axis=[1, 2])
  elif cfg["pool_type"] == "squared_mean":
    net = tf.reduce_mean(net**2, axis=[1, 2]

  logits = snt.Linear(batch["label_onehot"].

  loss_vec = tf.nn.softmax_cross_entropy_wit
      labels=batch["label_onehot"], logits=l

  return tf.reduce_mean(loss_vec)
```

Sampling tasks

- Image supervised
  - MLP classification
  - Conv with FC last
  - Conv pooling
- Image unsupervised
  - MLP autoencoders
  - MLP variational autoencoders
  - Non volume preserving flows
  - Masked autoregressive
- Language supervised
  - RNN text classification
- Language unsupervised
  - Character language modeling RNN
  - Word language modeling RNN
- Synthetic
  - Set of synthetic tasks from Wichrowska 2017 et. al.
  - Quadratic tasks possibly with nonlinearity

Outer-generalization to new tasks is key

- Ideally we train on the problems we care about (e.g. large ResNets).
  - This is too expensive! Have to make trade offs.



Normalization of tasks

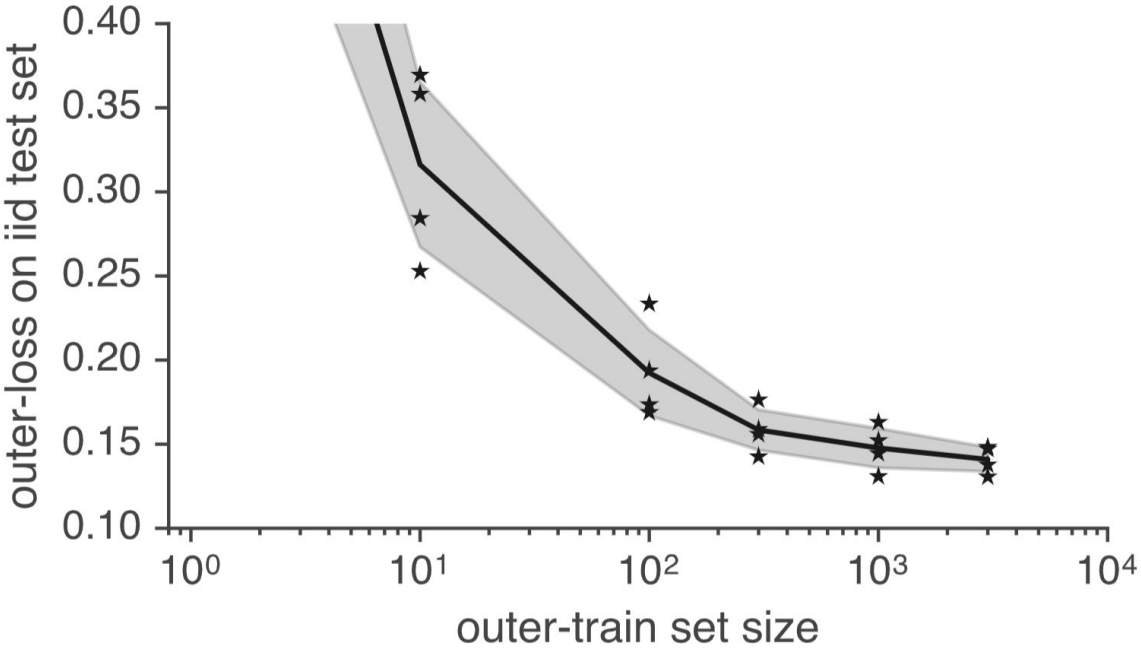
- Wildly different loss scales across problem
- Need to normalize so each task has same "weight"

Normalization of tasks

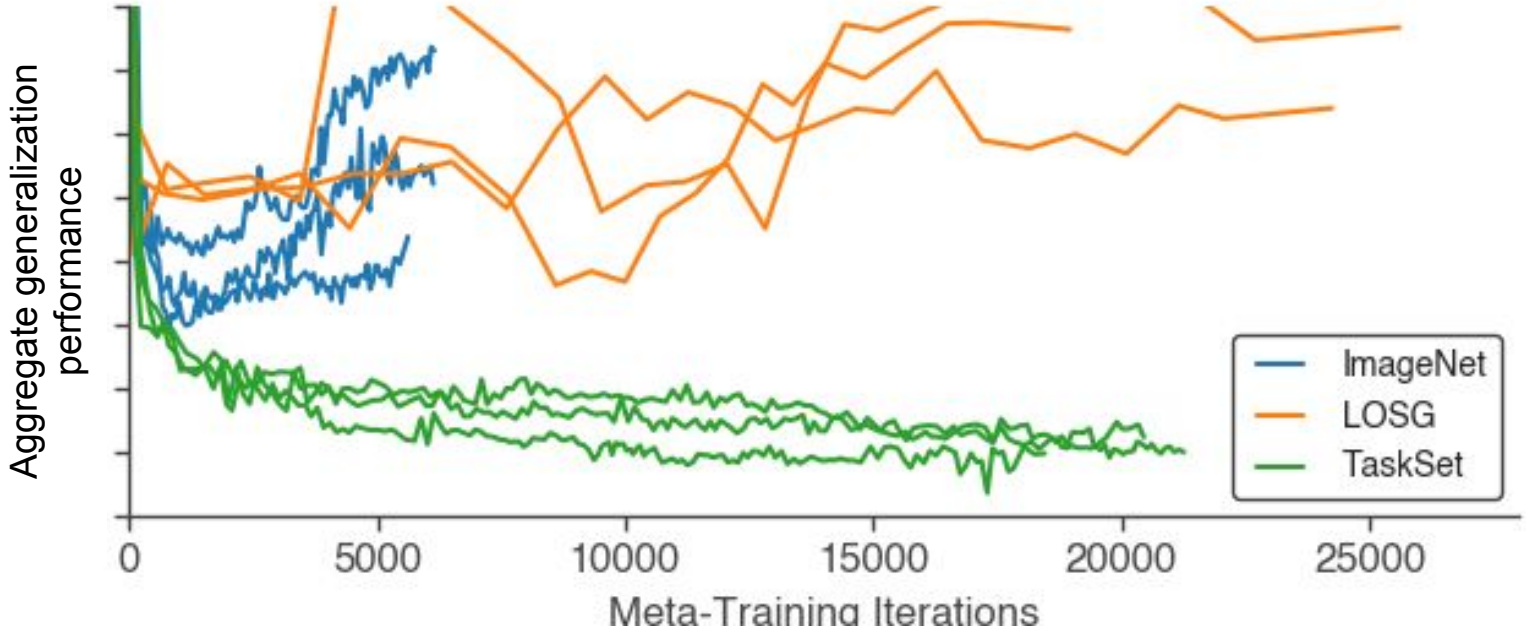
- Wildly different loss scales across problem
- Need to normalize so each task has same "weight"

- Train task with Adam in different configurations
- Linearly interpolate between initial loss (1.0) and the lowest loss (0.0)
- Clip between (-2, 2)

Hypothesis: More tasks leads to better training



Hypothesis: Outer-training on a diverse set of tasks will allow our learned optimizers to generalize to larger problems.

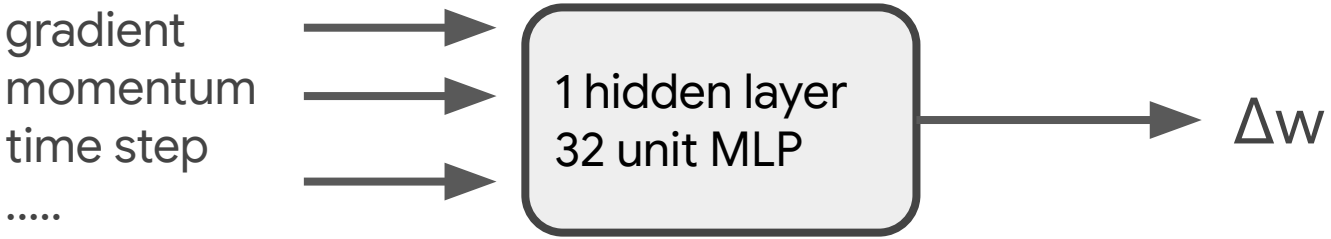


Improved Learned Optimizer Architectures

Architectures: Per parameter NN based update rules

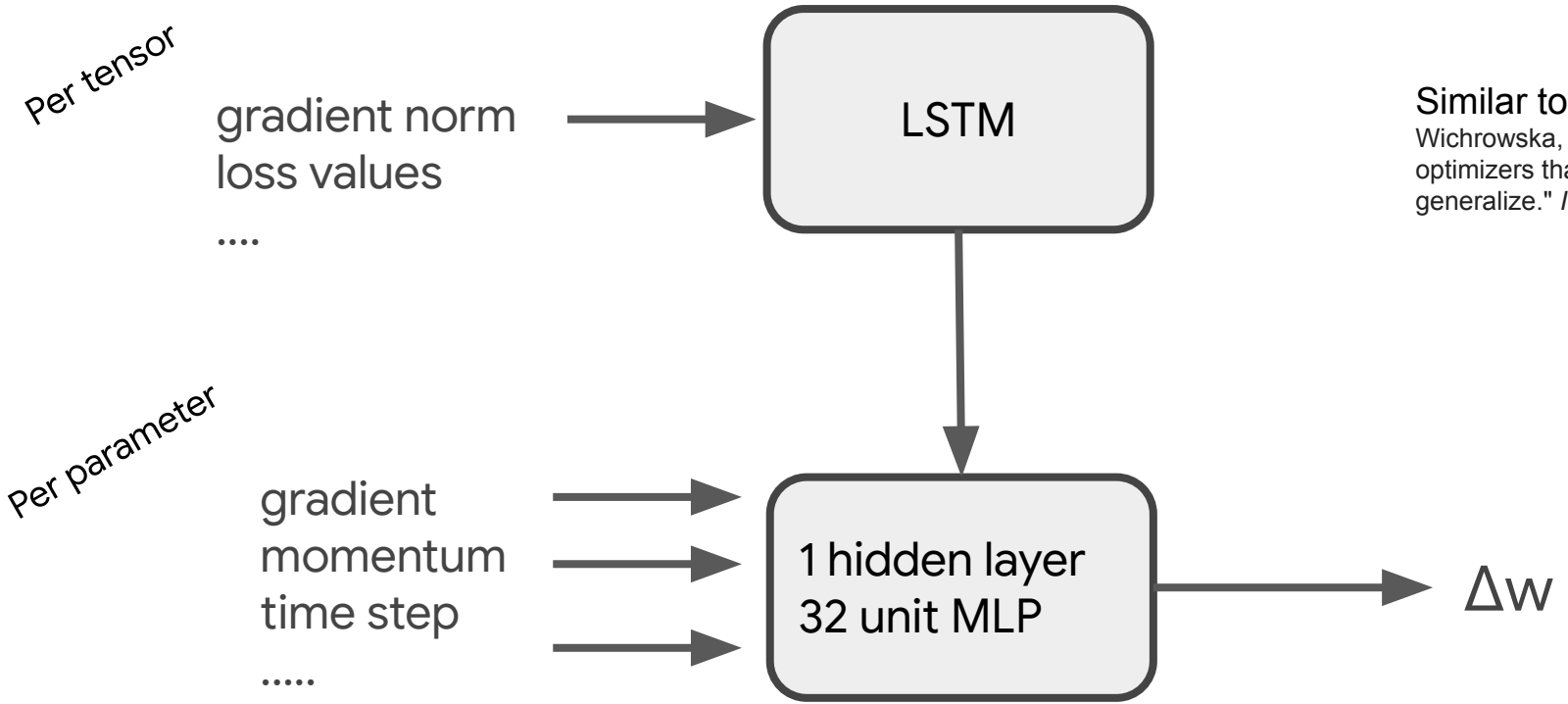
Operates on [n_parameters, 1]

Andrychowicz, Marcin, et al. "Learning to learn by gradient descent by gradient descent." *Neurips*. 2016.



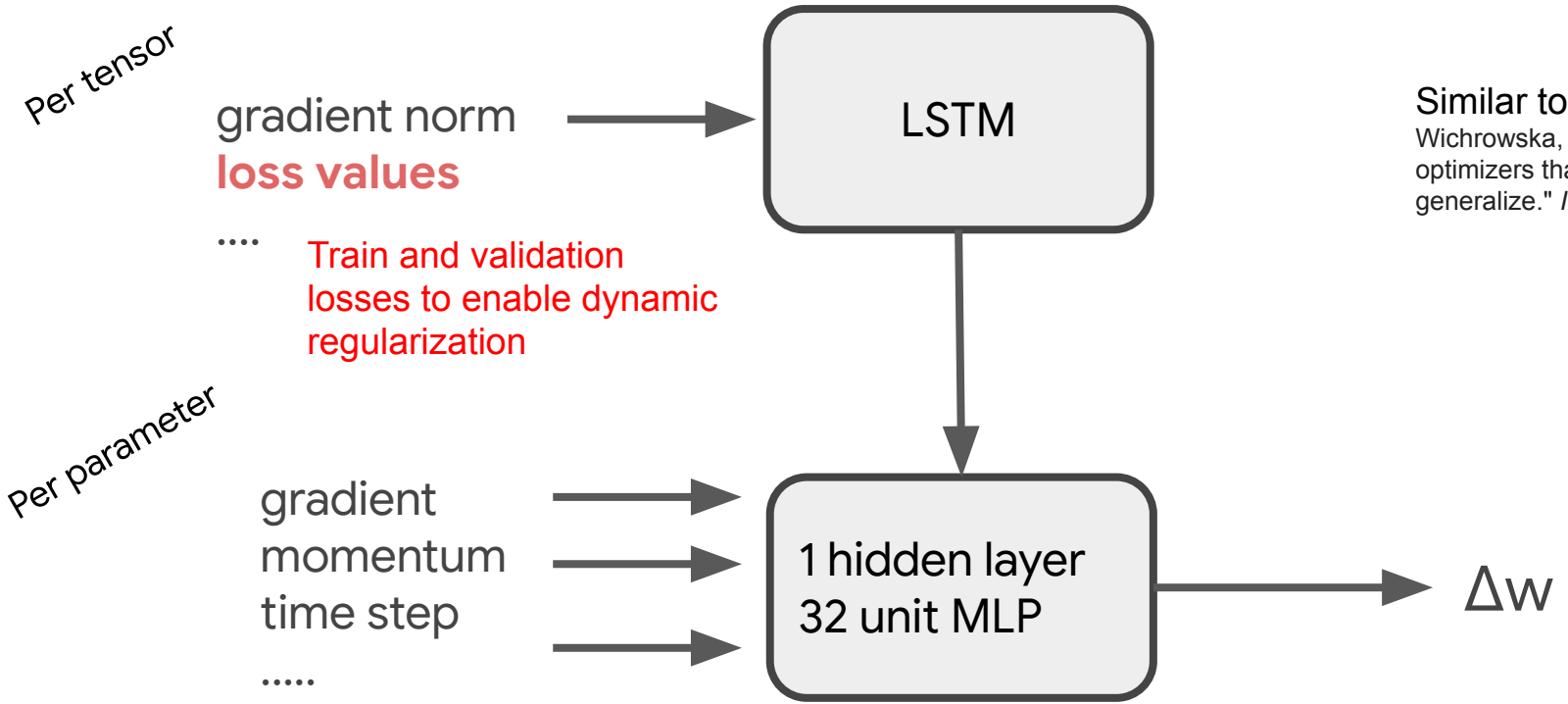
Metz, Luke, et al. "Understanding and correcting pathologies in the training of learned optimizers." *ICML 2019*

Architecture: Hierarchical



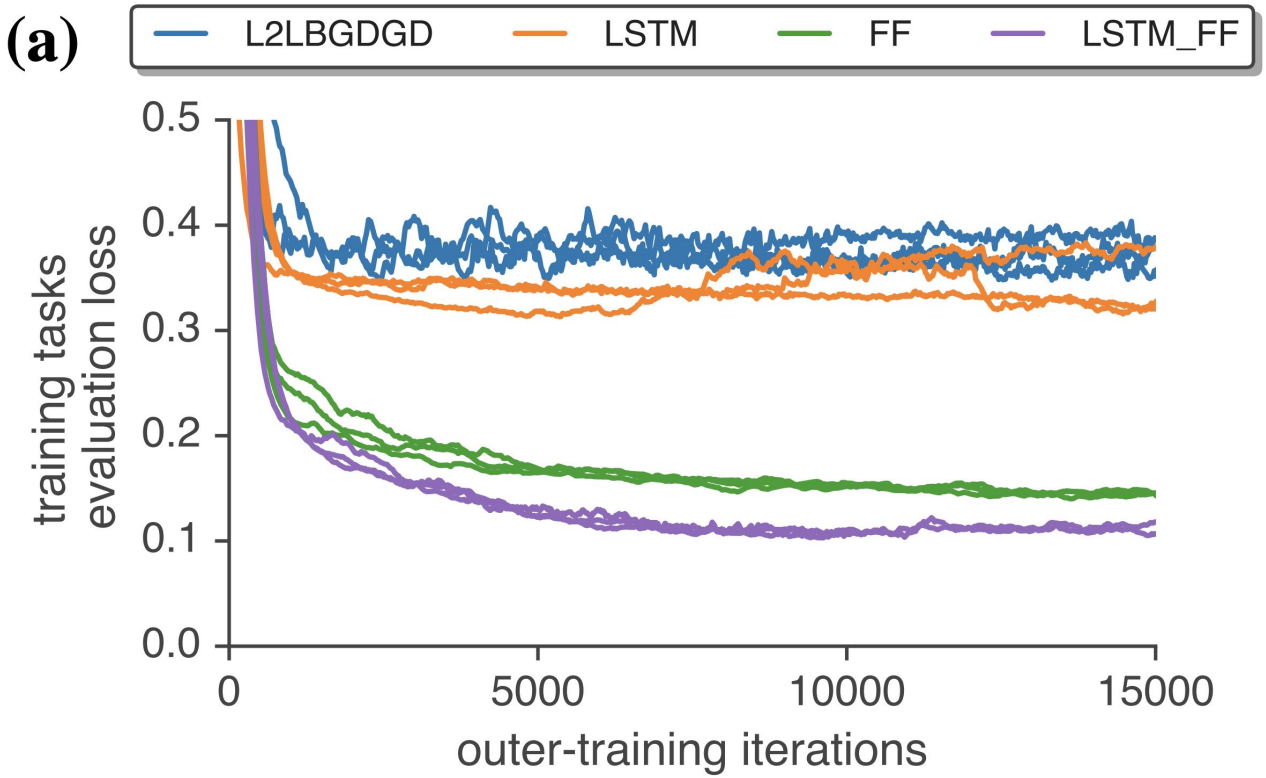
Similar to:
Wichrowska, Olga, et al. "Learned optimizers that scale and generalize." *ICML* 2017.

Architecture: Hierarchical



Similar to:
Wichrowska, Olga, et al. "Learned optimizers that scale and generalize." *ICML* 2017.

Improved architectures train faster



Compute / Outer Optimization

Learning to learn by gradient descent by gradient descent

1 GPU
< 1 day (?)

Task specific learned optimizers

~1k CPU cores for
~5 days

General purpose learned optimizers

~20k-40k CPU cores for
~4 weeks

Things are starting to get interesting!

How does it do?

- Comparing optimizers: hard
- Comparing learned optimizers: EXTRA HARD

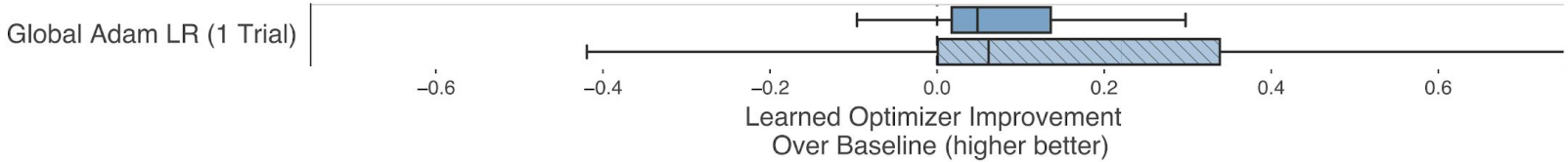
How does it do?

- Comparing optimizers: hard
- Comparing learned optimizers: EXTRA HARD

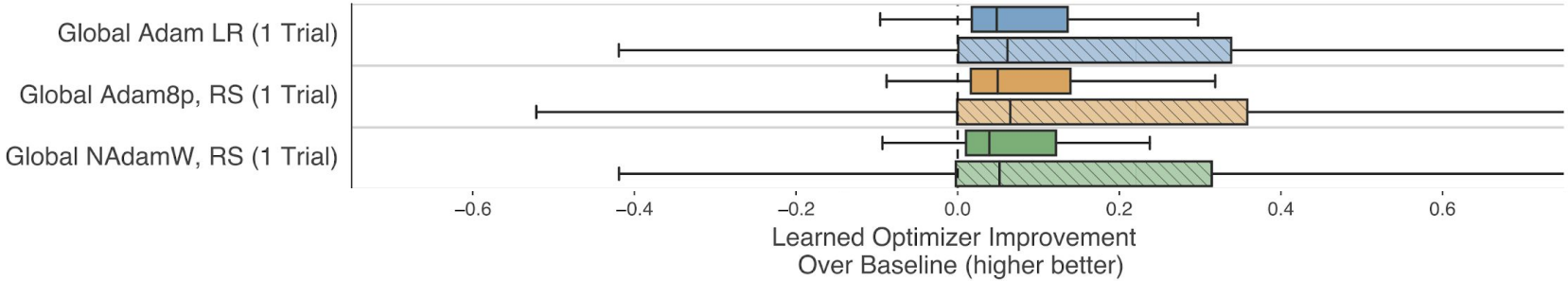
Distribution of IID problems (usual train / test performance)

Out of distribution

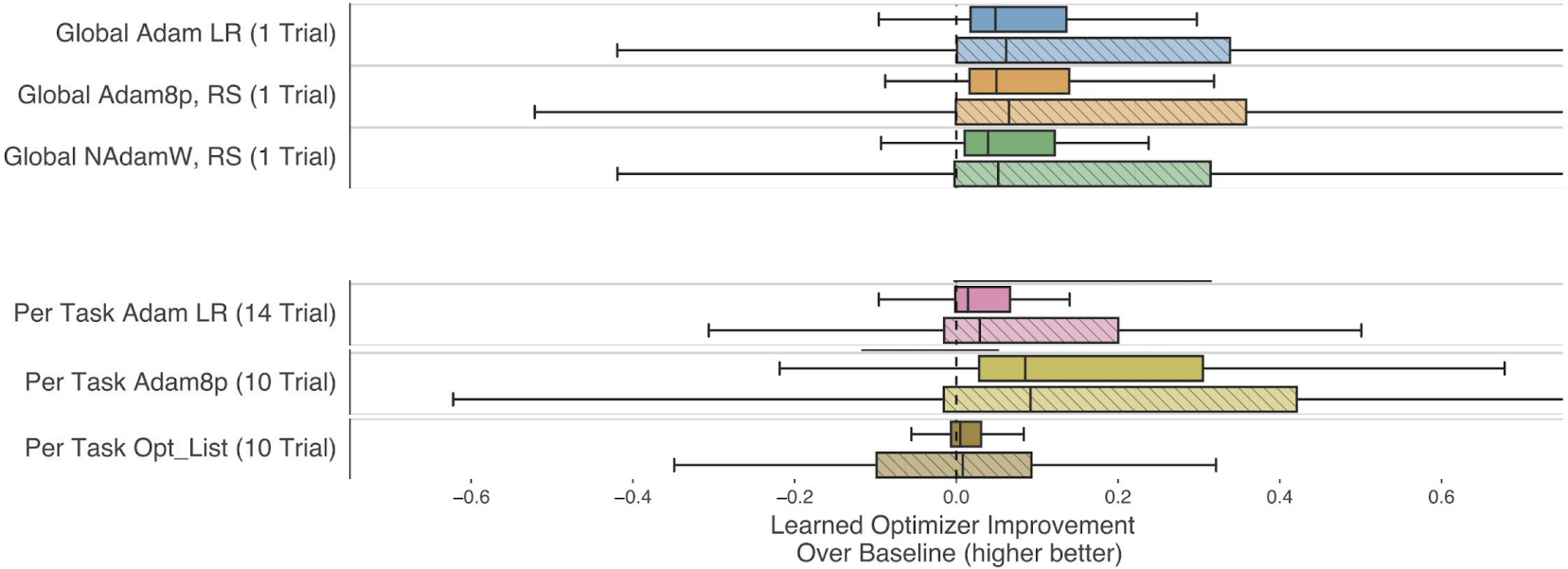
Performance on IID train / test



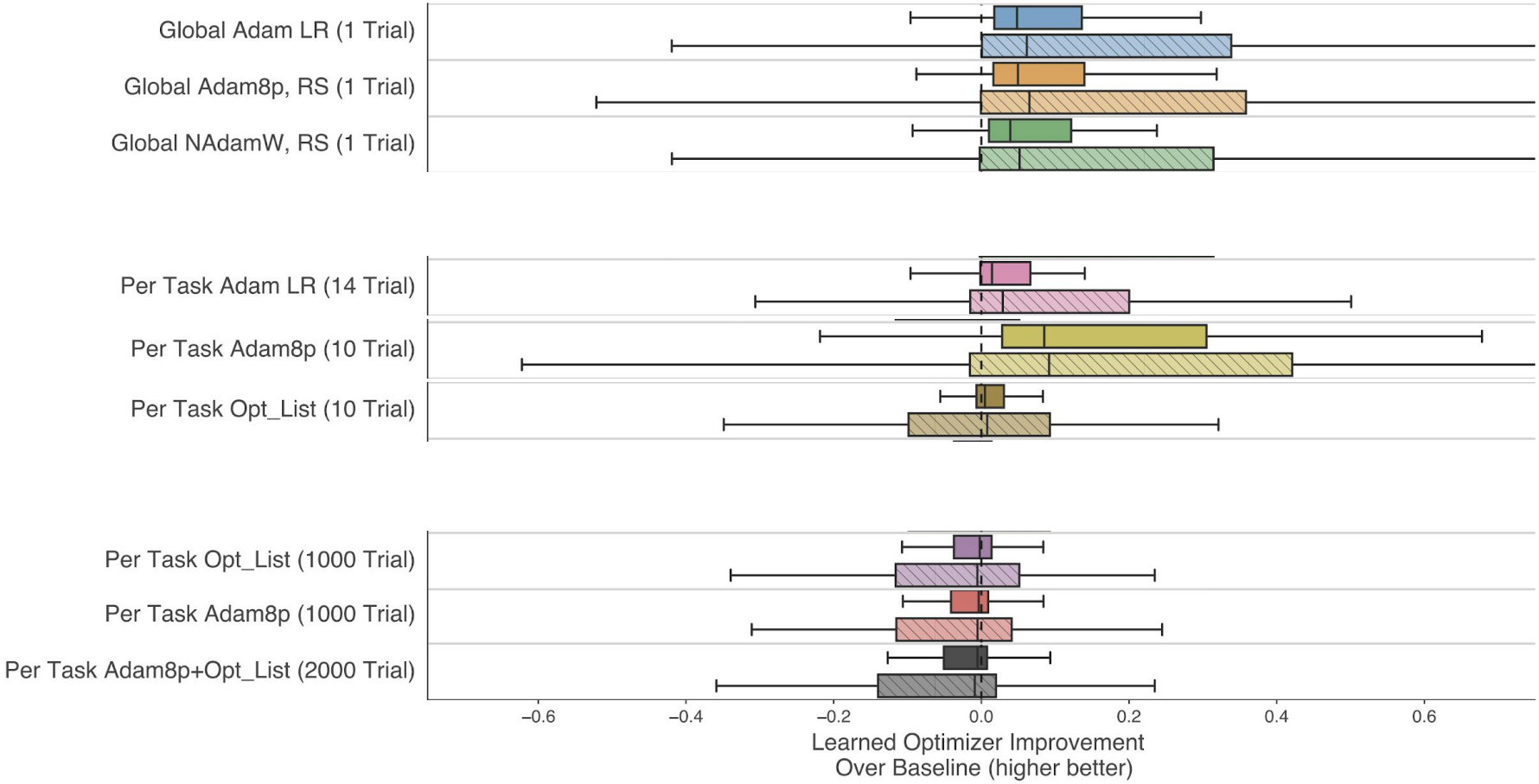
Performance on IID train / test



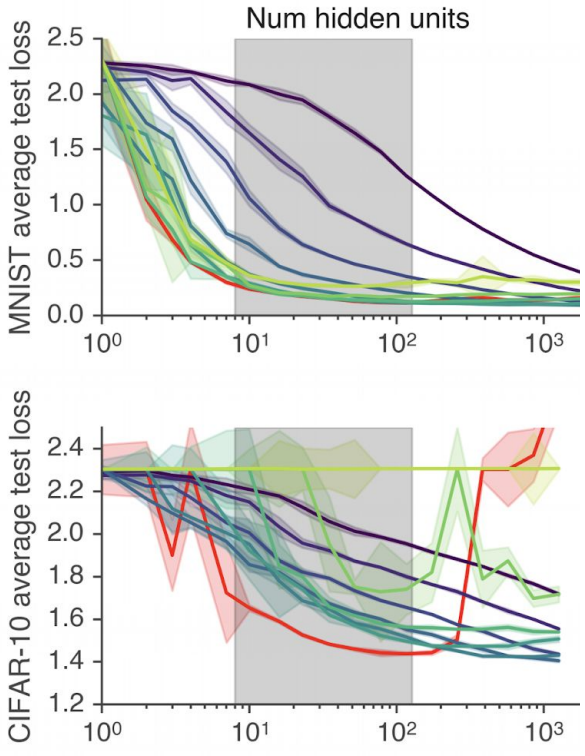
Performance on IID train / test



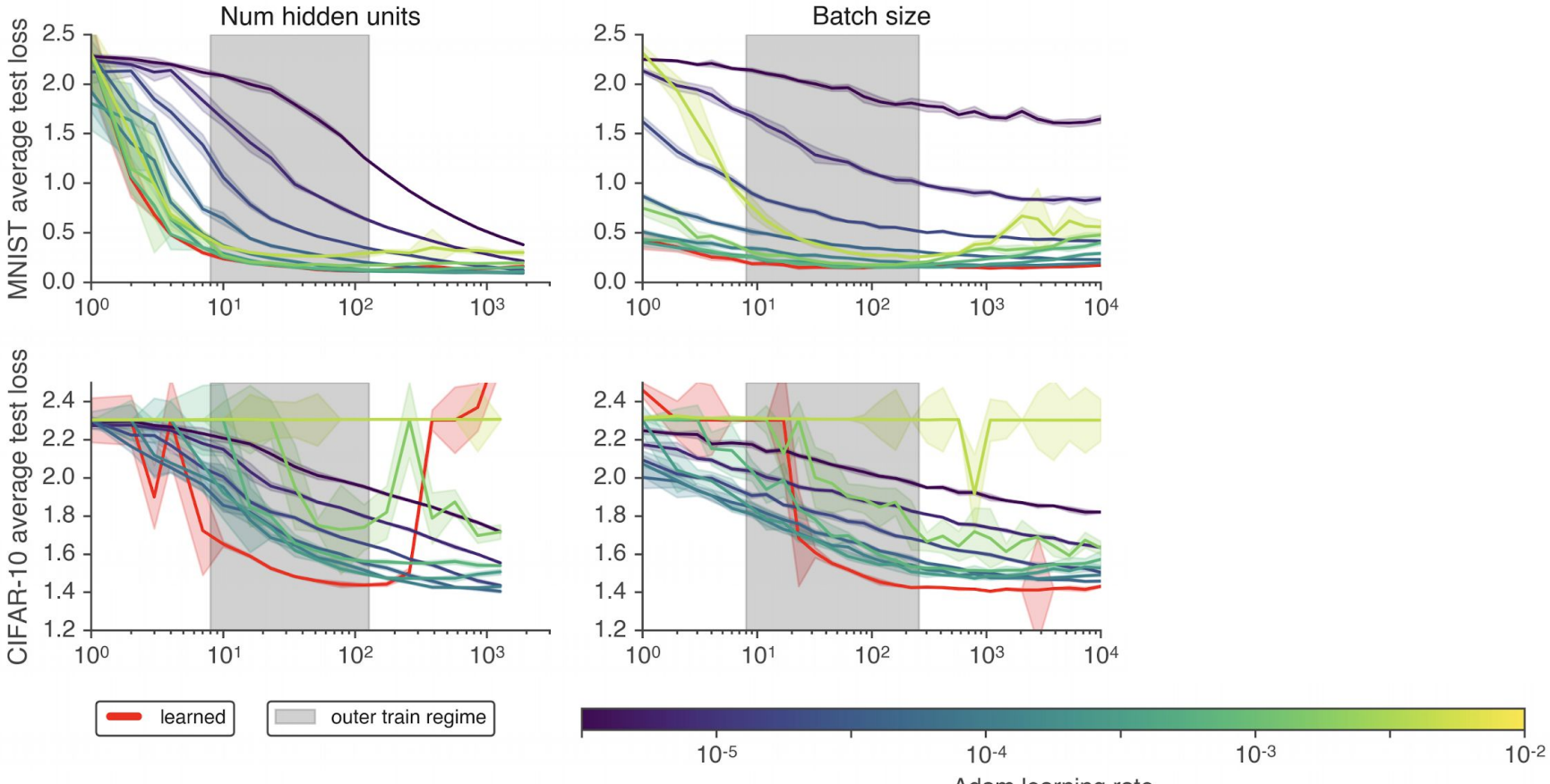
Performance on IID train / test



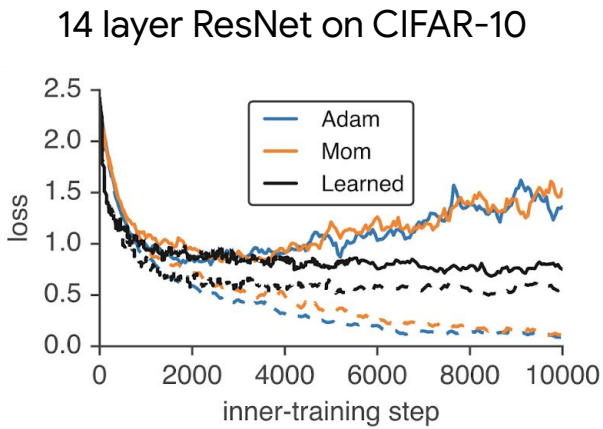
Controlled out of distribution



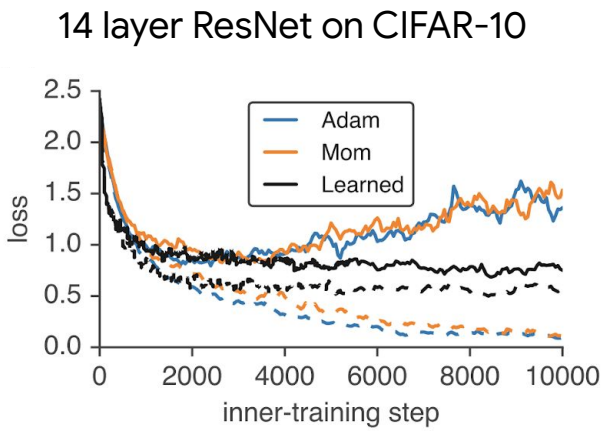
Controlled out of distribution

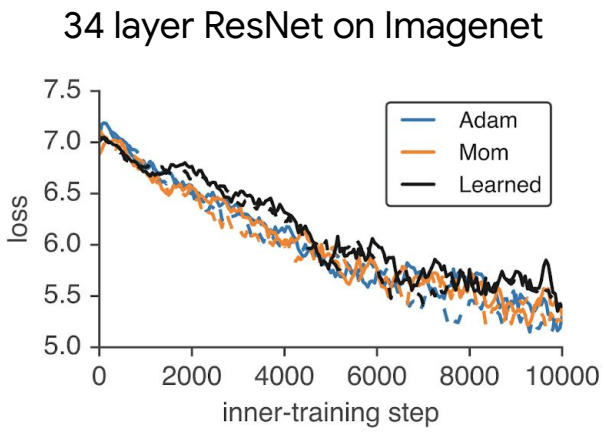


Large Image model transfer

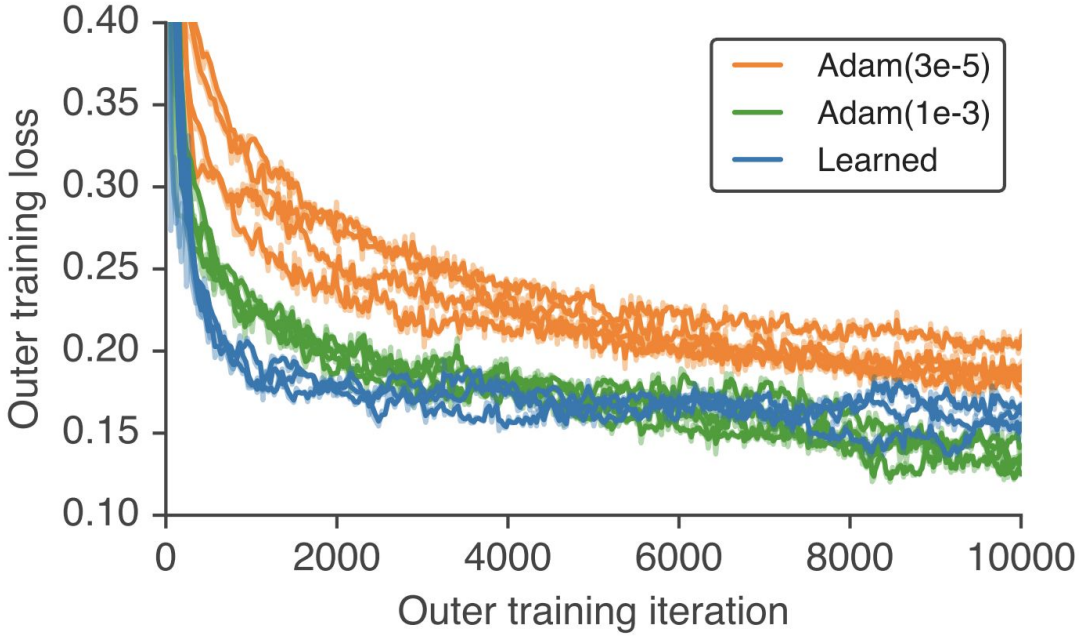


Large Image model transfer





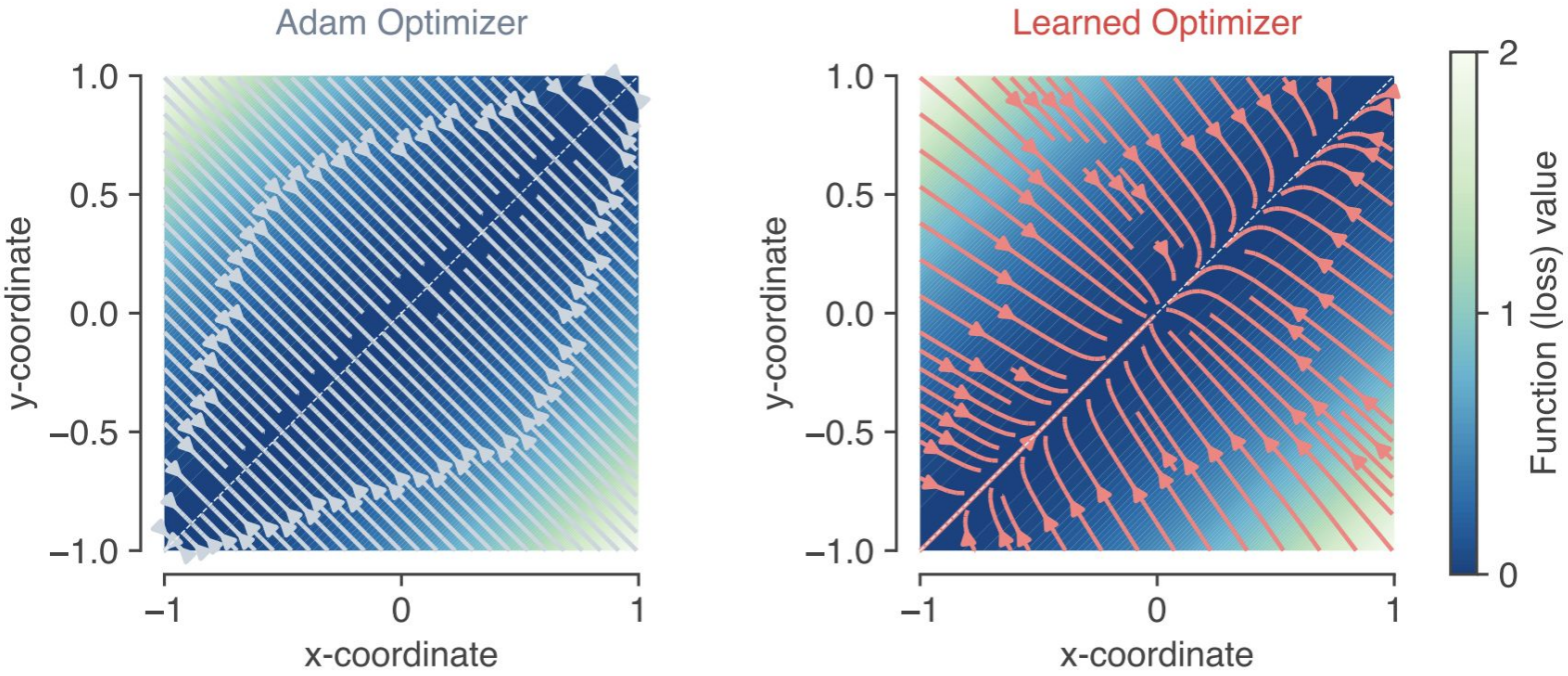
Training new learned optimizers



This is *almost* to the point where I would want to use this for my research.

Learned, implicit regularization

$$f(x, y) = \tfrac{1}{2}(x - y)^2$$



Outline

1. Motivation
2. Problem Setup
3. Task specific learned optimizers
4. General purpose learned optimizers
5. **Future**

Make learned optimizers more accessible

- Too expensive
  - prototype model that runs on TPU promising
- More reliable, better generalization
- More power -- get more performance out of optimizers
  - With better features? Second order?
  - Better models?

Future of learned learning algorithms

- I believe we are at the brink of an "AlexNet" like moment
- Learned optimizers are almost usable to me in my own research
- Optimizers are just the beginning

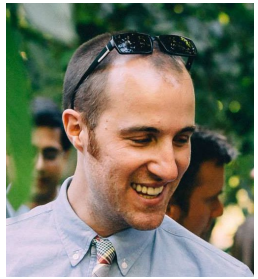
My Awesome Collaborators

Niru
Maheswaranathan

Ruoxi
Sun

Daniel
Freeman

Ben
Poole



Jascha
Sohl-Dickstein

Jeremy
Nixon

Thanks!

Questions + Discussion?