Tasks, stability, architecture, and compute: Training more effective learned optimizers, and using them to train themselves

Google Research

Luke Metz
Google Research, Brain Team
lmetz@google.com

Niru Maheswaranathan
Google Research, Brain Team
nirum@google.com

C. Daniel Freeman
Google Research, Brain Team
cdfreeman@google.com

Ben Poole
Google Research, Brain Team
pooleb@google.com

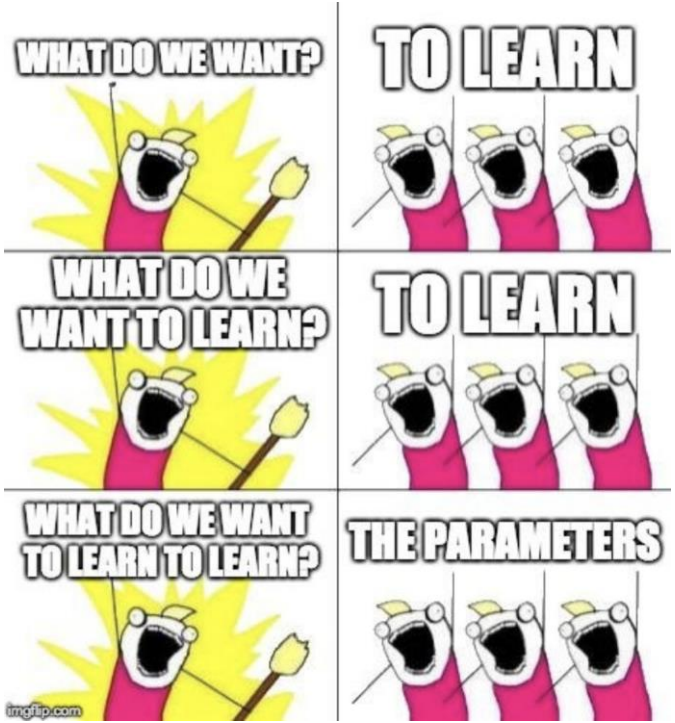
Jascha Sohl-Dickstein
Google Research, Brain Team
jaschasd@google.com

TL;DL (too long didn't listen)

- ➔ Deep learning allowed the replacement of hand-crafted features with learned features.
- ➔ However, we continue to use hand designed optimization algorithms (such as gradient descent, momentum, and so on) for training models.
- ➔ New(ish) field of research: learned optimizers - optimizers that are learned by the network.
- ➔ Google research brain team take this to a new level by using huge computation power and a variety of tasks.

Meta-learning: learning to learn

“Most learning curves plateau. After an initial absorption of statistical regularities, the system saturates and we reach the limits of hand-crafted learning rules and inductive biases. In the worst case, we start to overfit. But what if the learning system could critique its own learning behaviour? In a fully self-referential fashion. **Learning to learn… how to learn how to learn**. Introspection and the recursive bootstrapping of previous learning experiences – is this the key to intelligence?“



(from Robert Lange blog: https://roberttlange.github.io/posts/2020/12/meta-policy-gradients/)

Motivation

"Much as replacing hand-designed features with learned functions has revolutionized how we solve perceptual tasks, we believe learned algorithms will transform how we train models."

The goal: obtain an **optimization function** that can be used to train all kinds of machine learning models.

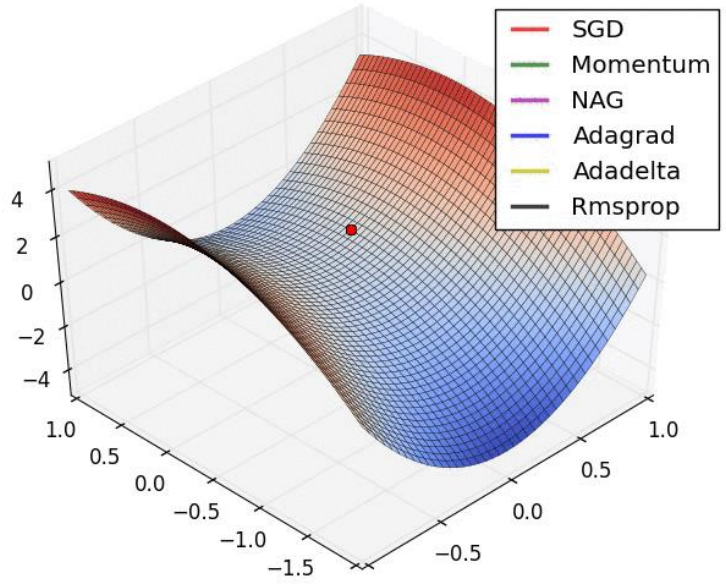
The “Loss Surface”

Optimizers

Algorithms or methods used to change the weights of a neural network in order to reduce the losses.

Cones:

- Require extensive expert supervision in order to be used effectively.
- Fail to flexibly adapt to new problem settings.
- Use information only from the gradients.



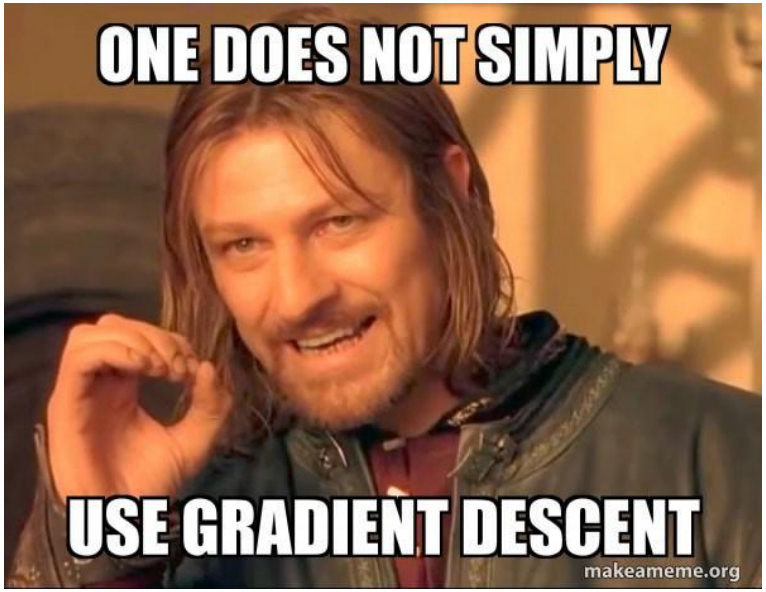
Current Optimization Algorithms

This is a large field but what all these optimizers have in common is humans sitting down coming up with a particular formula.

Repeat until convergence {

$$\theta_j \leftarrow \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta)$$

}

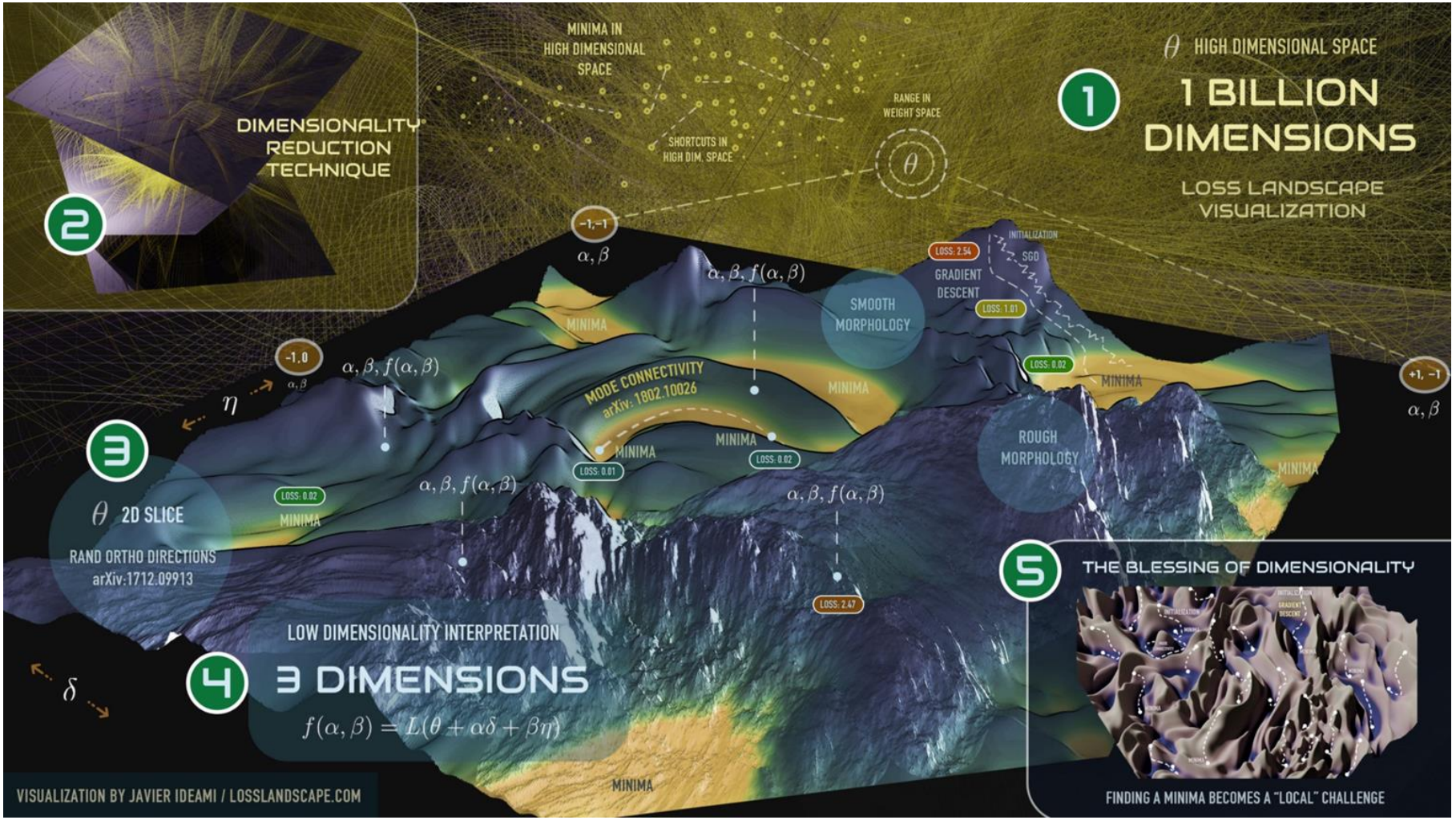


Learned Optimization

- We want to update θ not by a fixed formula but by using a function f which takes θ, the gradient of θ and features that we calculate from them:

$$\theta : f_\psi(\theta, \nabla\theta, \dots)$$

- The goal is to learn f.
- Parameterize f as a neural network that learns to output the next weight for the underlying neural network.
- It is a meta algorithm so tends to be more general and smooth and therefore it could be optimized fairly generally.
- Once we have a good f we can apply it to all sorts of tasks.


DIMENSIONALITY REDUCTION TECHNIQUE
2
MINIMA IN HIGH DIMENSIONAL SPACE
SHORTCUTS IN HIGH DIM. SPACE
RANGE IN WEIGHT SPACE
θ HIGH DIMENSIONAL SPACE
1
1 BILLION DIMENSIONS
LOSS LANDSCAPE VISUALIZATION
-1,-1
α, β
α, β, f(α, β)
INITIALIZATION
SGD
GRADIENT DESCENT
LOSS: 2.54
LOSS: 1.01
LOSS: 0.02
SMOOTH MORPHOLOGY
MINIMA
MODE CONNECTIVITY
arXiv: 1802.10026
LOSS: 0.01
LOSS: 0.02
ROUGH MORPHOLOGY
+1, -1
-1, 0
η
3
θ 2D SLICE
RAND ORTHO DIRECTIONS
arXiv:1712.09913
LOSS: 0.02
LOSS: 2.47
5
THE BLESSING OF DIMENSIONALITY
FINDING A MINIMA BECOMES A "LOCAL" CHALLENGE
LOW DIMENSIONALITY INTERPRETATION
4
3 DIMENSIONS
$f(\alpha, \beta) = L(\theta + \alpha\delta + \beta\eta)$
δ
VISUALIZATION BY JAVIER IDEAMI / LOSSLANDSCAPE.COM

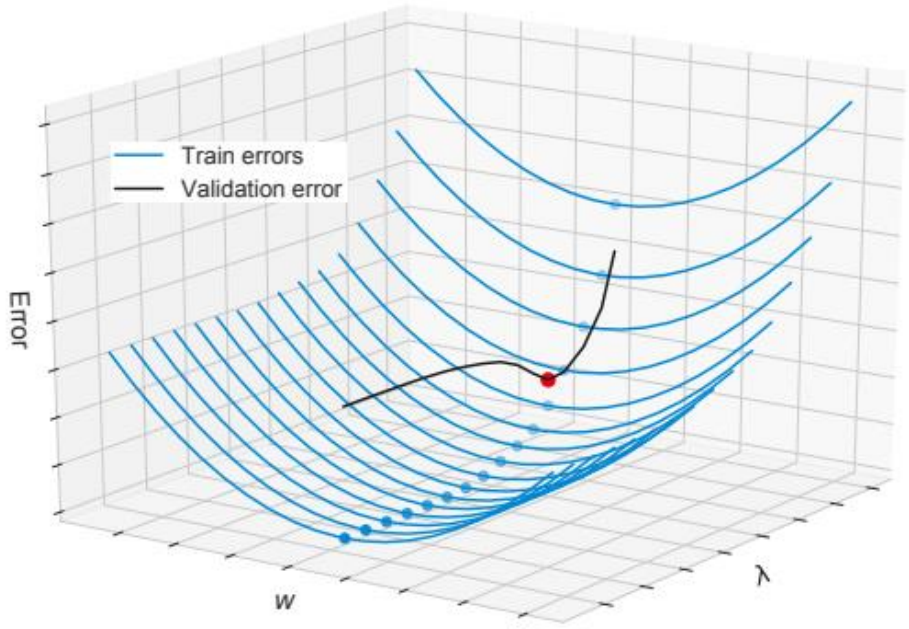
Addressing Problems in Learning Optimizers

1. **Computational scale**: costly.
   ⇨ utilized massive parallel computing.
2. **Training tasks**: need large dataset of optimization tasks for generalization.
   ⇨ created large diverse task set.
3. **Inductive bias optimizer architecture**: The parameterization of the learned optimizer and the task information fed to it both strongly affect performance.
   ⇨ propose new hierarchical learned optimizer architecture that incorporates additional task information (such as validation loss).

Bilevel optimization

contains two loops:

1. an **inner loop** - that applies the optimizer to solve a task
2. an **outer loop -** that iteratively updates the parameters of the learned optimizer



Franceschi et al., 2018

The Dataset - Task Set

6000 tasks (all tasks take less than 100 milliseconds per-training step 🤔)

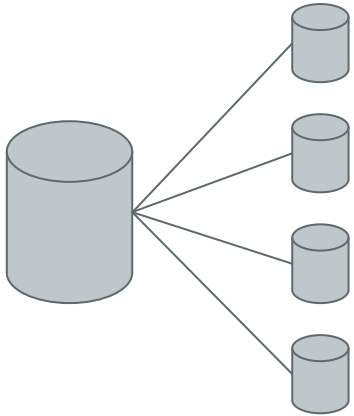
MNIST
5L CNN
BS 32
...

,

CIFAR10
RN 50
BS 64
...

...

Super optimizer



Training set: compute gradients on and use to update the inner-parameters

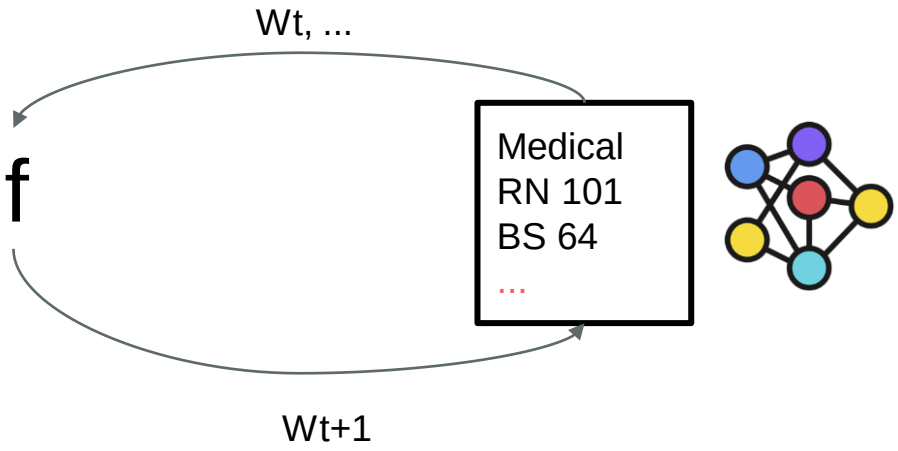
Inner-Validation set: compute validation losses to be used by the learned optimizer

Outer-Validation set: used for updating the weights of the learned optimizer

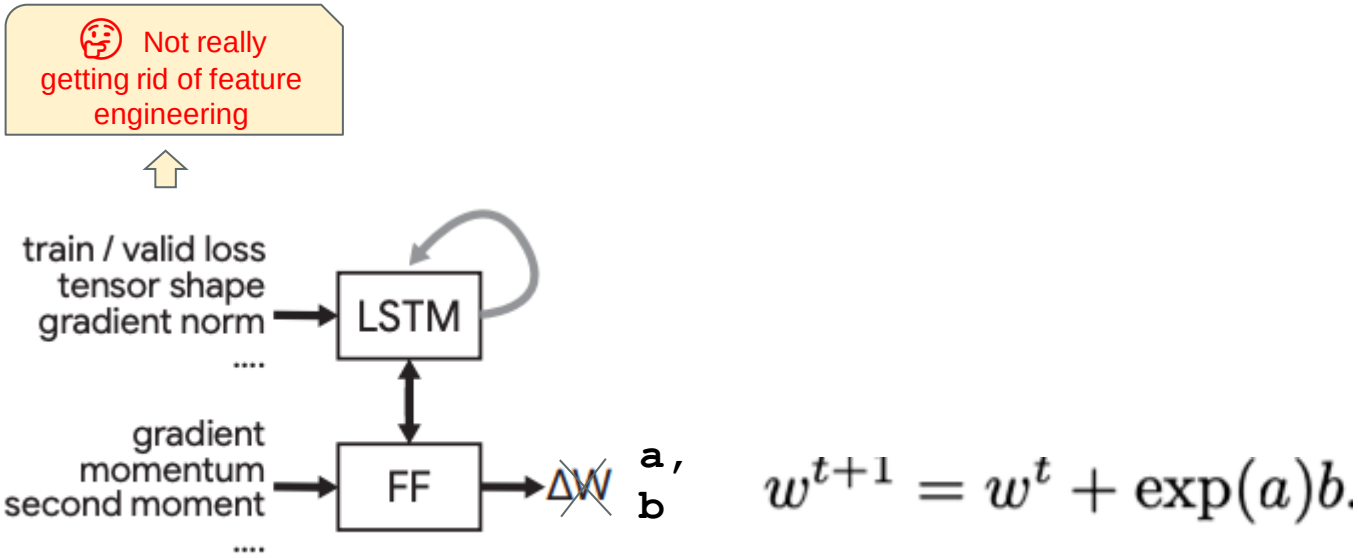
Test set: used for testing an already trained learned optimizer

Goal

After finding f which is an optimizer that works for all of these dataset samples we can give any sort of new sample and the optimizer will output good parameters for that task.



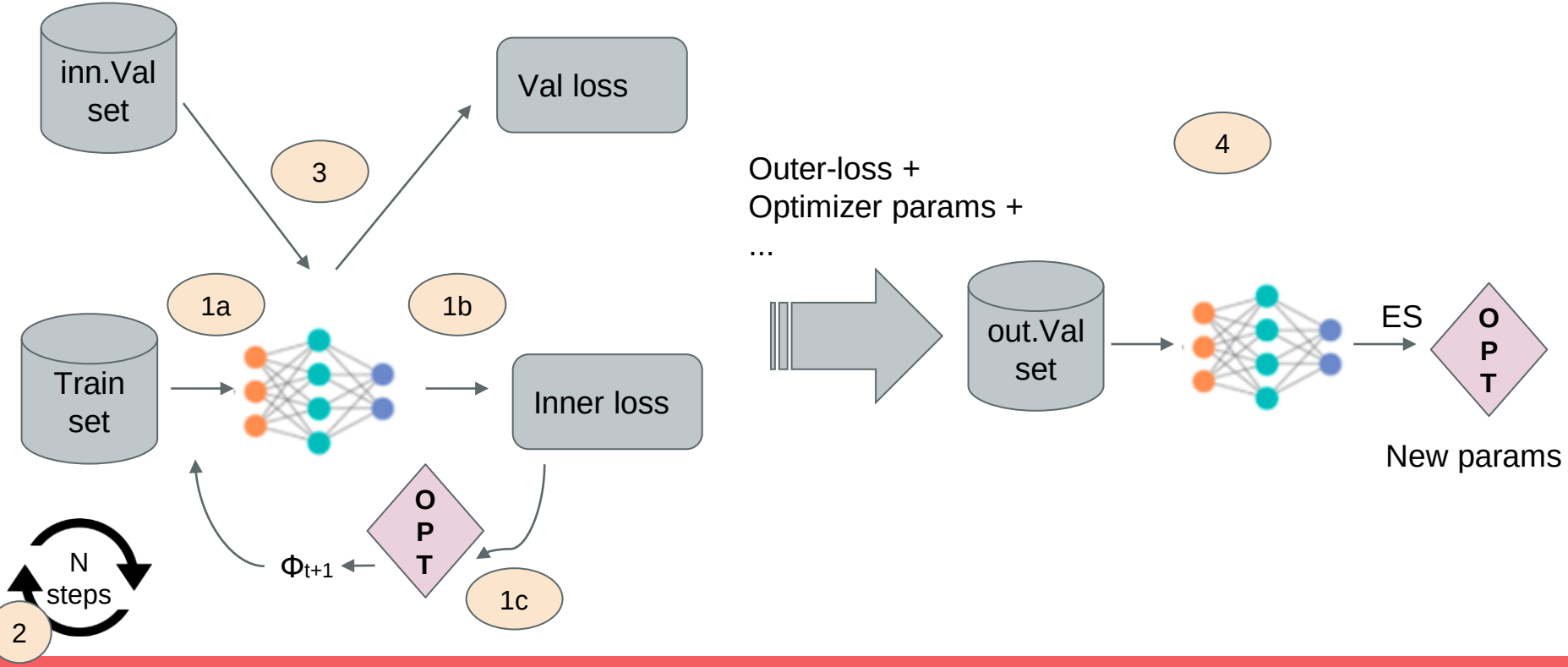
Optimizer Architecture



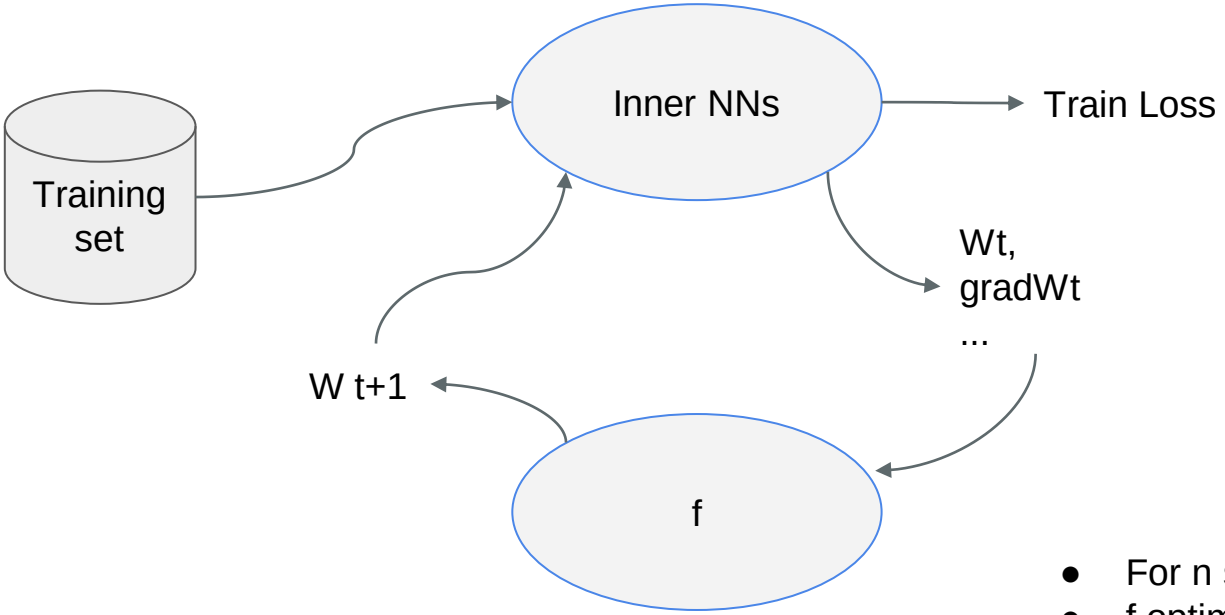
$$w^{t+1} = w^t + \exp(a)b.$$

- For example, for simple gradient descent it will output, a=0 and b=gradient. Of course, we add more information for better decisions.
- The output is not a symbolic expression but 2 numbers/vectors.
- We want to train f such that the validation loss of the inner task is as small as possible

How the optimizer is being learned



Training Step - Part 1



- For n steps
- f optimizes the inner train loss
- Not optimizing f here

Pseudocode - Inner Training

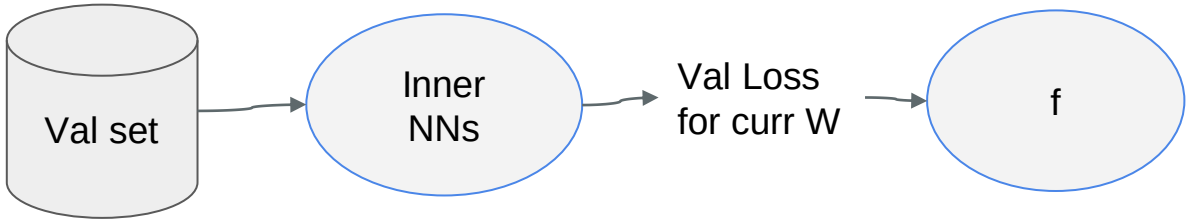
```
task = SampleTask()
optimizer = Optimizer()

params = task.initialParams()
state = optimizer.initial_state(state)

for i in range(N): # Length of inner training
    inputs = GetInputFromTask(params, with_valid_data=(i % 10 == 0))
    params, state = optimizer.next_state(inputs, params, state)
```

Training Step - Part 2

We want to optimize f using the mean inner validation loss:



But…

- All of part 1 needs to be differentiable.
- The loss is computed only after n steps so we need to packprop through all of them. These are thousands of steps so it is not possible

The Solution: Evolution Strategies

How should we optimize the model to make the validation loss as small as possible a given an n step rollout of the inner network while we can't back propagate through the entire rollout?

"We deal with these issues by using derivative-free optimization – specifically, evolutionary strategies – to minimize the outer-loss, obviating the need to compute derivatives through the unrolled optimization process. Previous work has used unrolled derivatives, and was thus limited to short numbers of unrolled steps. Using ES, we are able to use considerably longer unrolls."

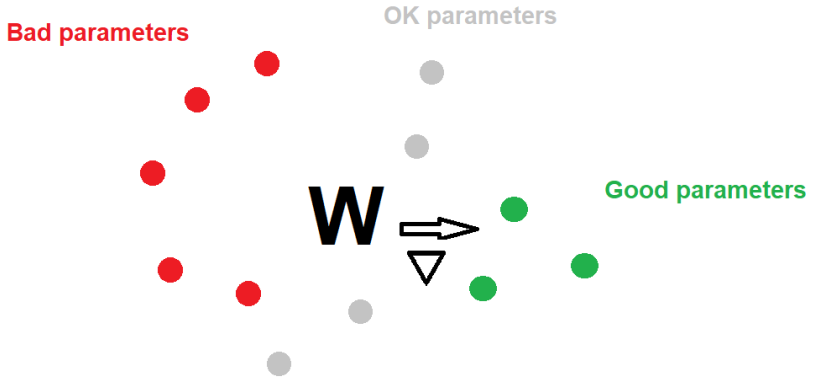
Evolution Strategies

- An optimization technique based on ideas of evolution.
- Use natural problem-dependent representations, and primarily mutation and selection, as search operators.
- The sequence of generations is continued until a termination criterion is met.
- The selection in is deterministic and only based on the fitness rankings.

Training Step - Part 2 - Using Evolution

The details are not reported in the paper. However, the intuition is:

- Taking the optimizer parameters and perturb them by a little bit in multiple directions.
- Shift our guess of the best parameters into the direction of the good ones and away from the direction of the bad ones.
- From this shift we obtain a pseudo gradient.
- The gradient is feeded into Adam which optimizes the weights.



Pseudocode - Outer Training

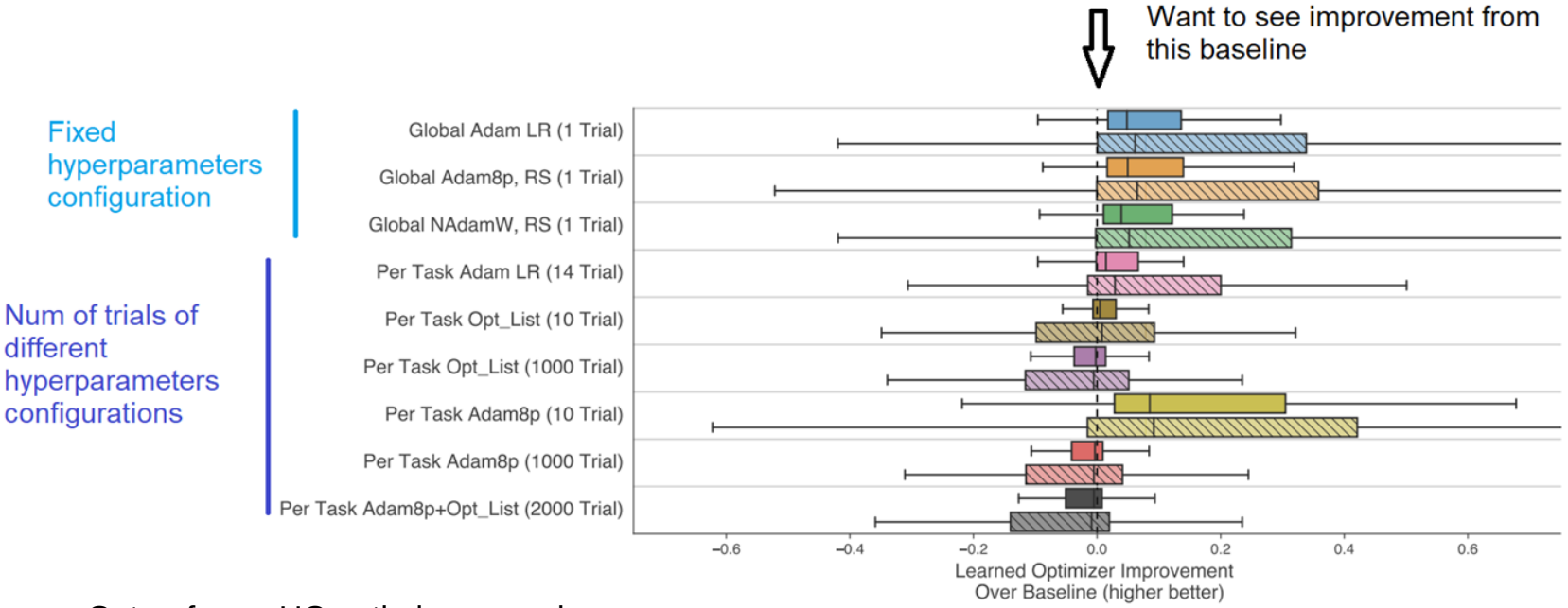
```
task = SampleTask()
optimizer = Optimizer()
# When outer-training with ES, perturb optimizer weights.

params = task.initialParams()
state = optimizer.initial_state(state)

outer_valid_losses = []
for i in range(N): # Length of a trunction
    inputs = GetInputFromTask(params, with_valid_data=(i % 10 == 0))
    params, state = optimizer.next_state(inputs, params, state)
    for i in range(10):
        outer_valid_losses.append(task.valid_outer_loss(params))

outer_loss = mean(outer_valid_losses)
# When outer-training with ES, one must perturb the optimizer's
                                parameters before computing
                                outer_loss.
```

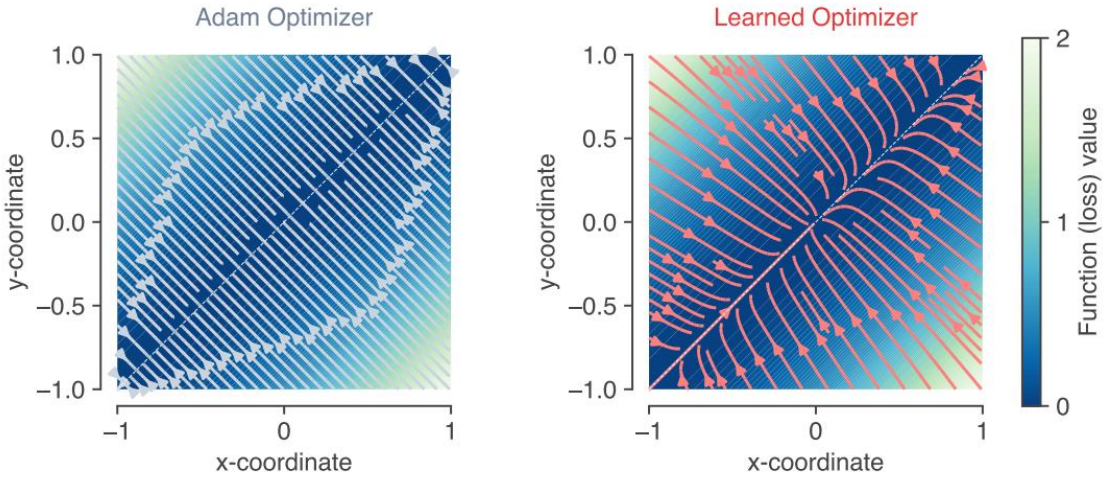
Main Results - Comparing to Hand-Crafted Optimizers



- Outperforms HC optimizers used without grid-search
- Use more memory

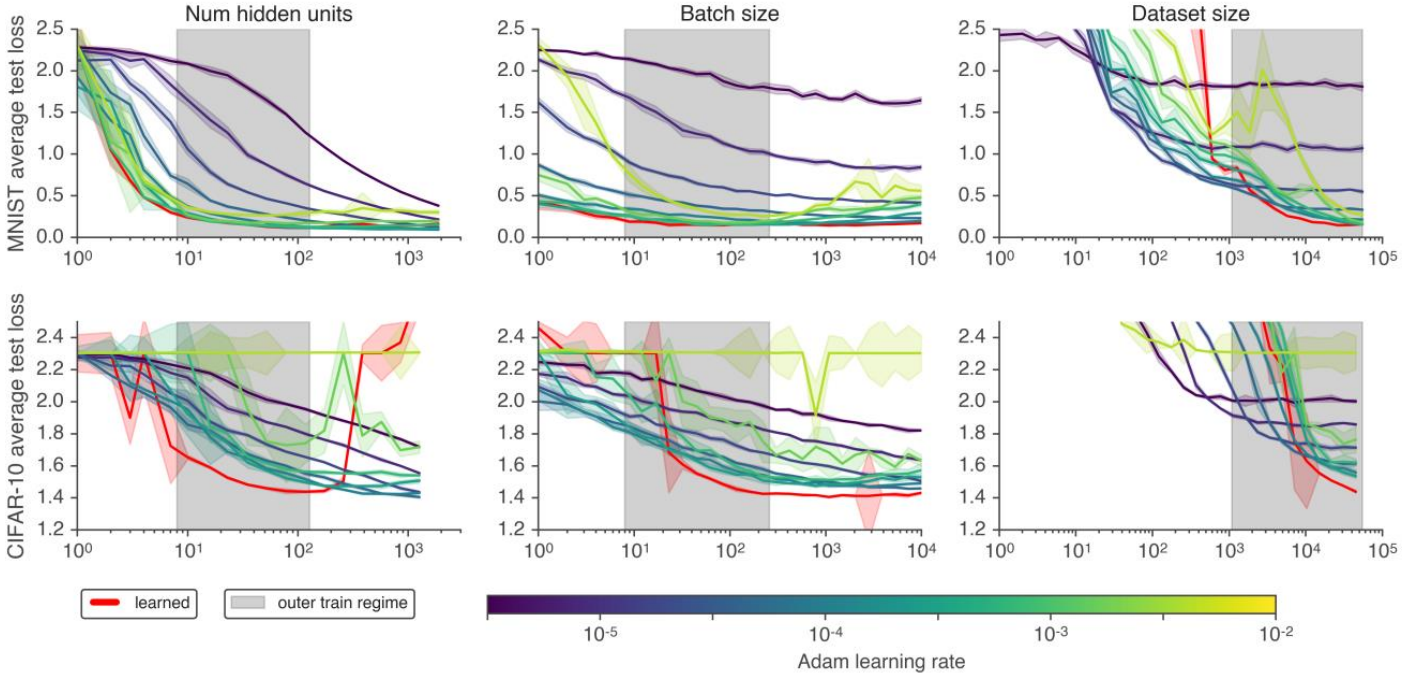
Our learned optimizer - only 1 trial (it doesn't have any hyperparameters)

Implicit Regularization in the Learned Optimizer



- The loss function: $f(x, y) = \frac{1}{2}(x - y)^2$
- The trajectories of Adam go in the direction of the closest solution while the learned optimizer pulls towards (0,0).
- As if there were an added regularization penalty on the magnitude of each parameter.

Generalization across Tasks

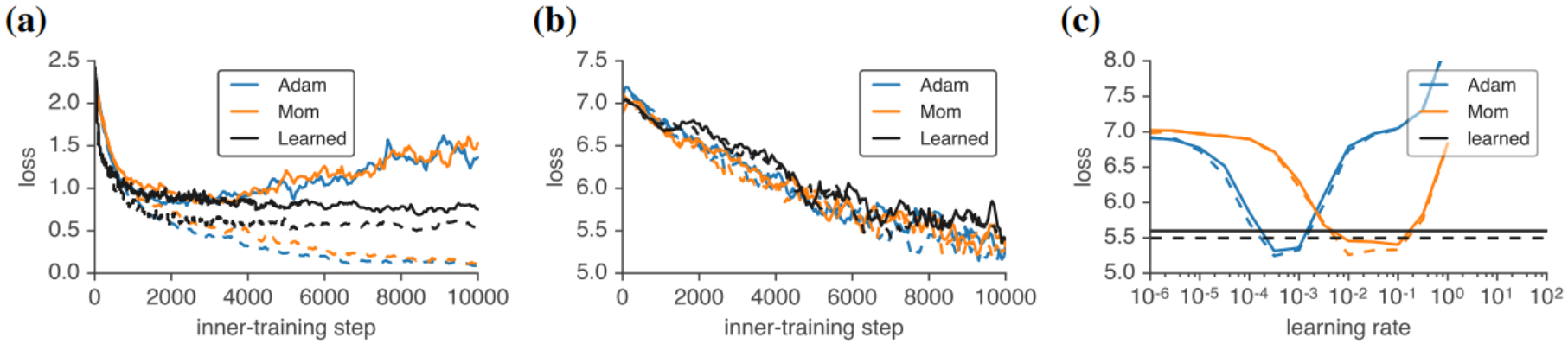


"We find the learned optimizer is able to generalize outside of the outer-training distribution in some cases."

Scaling Up

small ResNet on CIFAR-10

small ResNet on 64x64 resized ImageNet

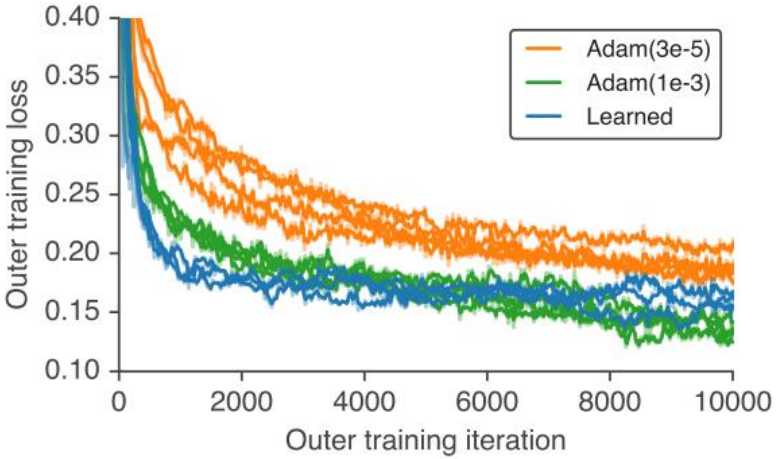


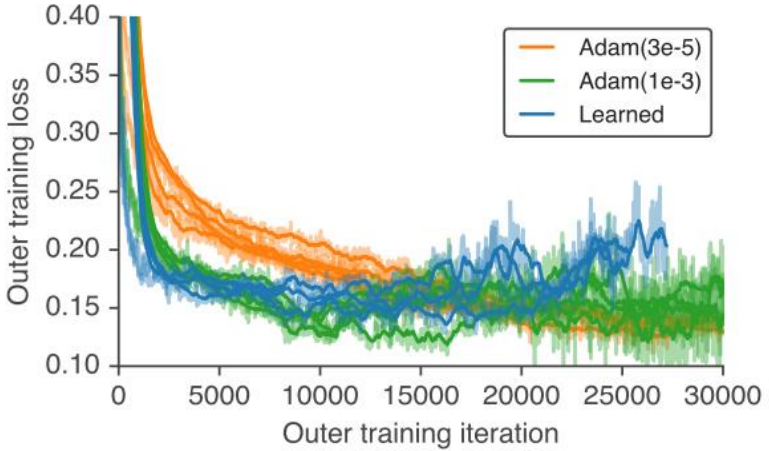
Doesn't overfit

Larger task - outperformed by a tuned optimizer

The Learned Optimizer Trains Itself

- Train the learned optimizer once
- Use this trained optimizer for training the learned optimizer
- Faster at the beginning
- Tends to flatten after 4k iterations





Pseudocode

```
        inp = tf.concat(sorted_values(inps), 1)
        inp = inp * tf.rsqrt(1e-8 +
                             tf.reduce_mean(tf.square(inp), axis=0,
                                                            keep_dims
                                                            =True))
        inp = tf.clip_by_value(inp * 0.5, -1, 1)
```

Next, we embed the current inner-training step with sinusoids of different frequencies.

```
        def sin_embedding(x):
          mix_proj = []
          for i in [1, 3, 10, 30, 100, 300, 1000, 3000, 10000, 30000,
                                                 100000]:
            s = tf.to_float(tf.to_float(i) / float(np.pi))
            mix_proj.append(tf.sin(s * tf.to_float(x)))
          return tf.stack(mix_proj)

        step = utils.sin_embedding(training_step)
        stack_step = tf.tile(
            tf.reshape(step, [1, -1]),
            tf.stack([tf.shape(flat_g)[0], 1]))
```

We compute a features based on the number of tensors:

```
        log_num_tensors = tf.log(float(len(grads_and_vars))) - 1.

        stack_num_tensors = tf.tile(
            tf.reshape(log_num_tensors, [1, 1]),
            tf.stack([tf.shape(flat_g)[0], 1]))
```

And that's it!