AutoML-zero

Genetic Programming meets Neural Networks

Luca Manzoni

The paper

Esteban Real, Chen Liang, David R. So, Quoc V. Le
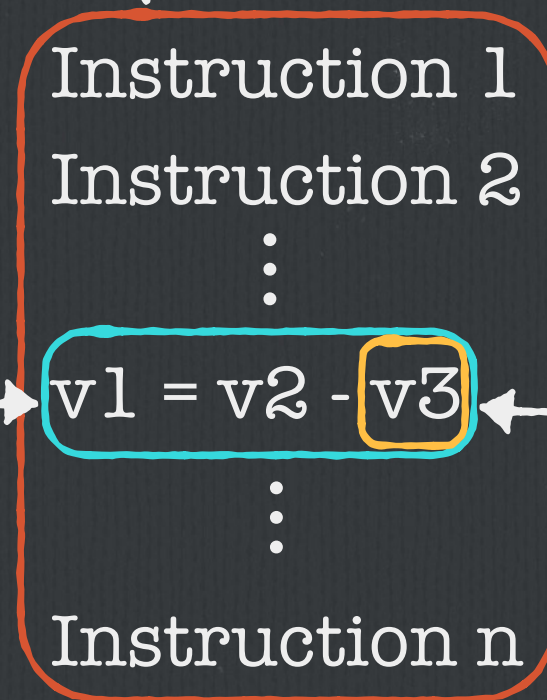
AutoML-zero: Evolving Machine Learning Algorithms From Scratch

Code representation

Programs as linear
lists of instructions

Constants or
memory locations

Instruction 1

Instruction 2

⋮

v1 = v2 - v3

⋮

Instruction n

Each instruction
has one or more
parameters

Instructions are selected among a list of 65

Memories

Target*

s_0

s_1

s_2

s_3

Prediction

Scalar memory

Input

v_0

v_1

v_2

v_3

Vector memory

M_0

M_1

Matrix memory
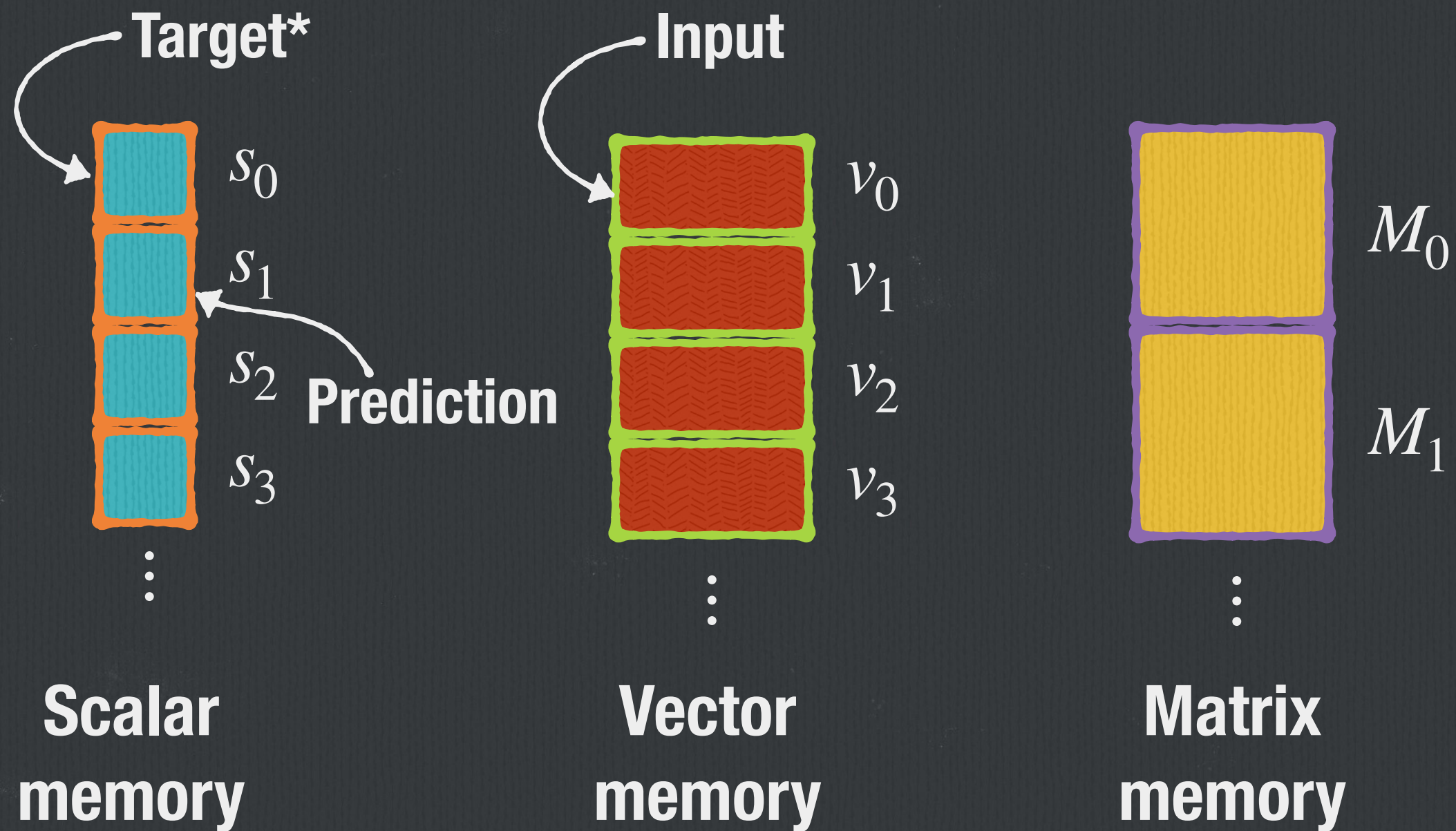
*Available only during the "Learn" phase

Which code is evolved?

☐ Three main blocks are evolved:

    ☐ **Setup**. Run once.

    ☐ **Predict**. Run to classify an input.

    ☐ **Learn**. Run only after the prediction.

How are the blocks used?

`Setup()` — **Once at the beginning**

`Predict()` `Learn()` — **For each element of the training set**

`Predict()` — **For each element of the validation set**

Code evolution

Best

Setup():	Setup():	Setup():	Setup():	Setup():
Instruction 1	Instruction 1	Instruction 1	Instruction 1	Instruction 1
Instruction 2	Instruction 2	Instruction 2	Instruction 2	Instruction 2
⋮	⋮	⋮	⋮	⋮
Instruction n	Instruction n	Instruction n	Instruction n	Instruction n
Predict():	Predict():	Predict():	Predict():	Predict():
Instruction 1	Instruction 1	Instruction 1	Instruction 1	Instruction 1
Instruction 2	Instruction 2	Instruction 2	Instruction 2	Instruction 2
⋮	⋮	⋮	⋮	⋮
Instruction n	Instruction n	Instruction n	Instruction n	Instruction n
Learn():	Learn():	Learn():	Learn():	Learn():
Instruction 1	Instruction 1	Instruction 1	Instruction 1	Instruction 1
Instruction 2	Instruction 2	Instruction 2	Instruction 2	Instruction 2
⋮	⋮	⋮	⋮	⋮
Instruction n	Instruction n	Instruction n	Instruction n	Instruction n

Mutation

Remove the oldest individual

Find the best individual…

… and copy it

Mutations

Type 1 | Type 2 | Type 3

Setup():
s4 = 0.5

Predict():
v1 = v0 - v9
v5 = v0 + v9
m1 = s2 * m2

Learn():
s4 = s0 - s1
s3 = abs(s1)

Setup():
s4 = 0.5

Predict():
v1 = v0 - v9
v5 = v0 + v9
m1 = s2 * m2

Learn():
s4 = s0 - s1
s2 = sin(v1)
s3 = abs(s1)

Setup():
s4 = 0.5

Predict():
s0 = mean(m1)
s3 = cos(s7)
m5 = m0 + m5

Learn():
s4 = s0 - s1
s3 = abs(s1)

Setup():
s4 = 0.5

Predict():
v1 = **v3** - v9
v5 = v0 + v9
m1 = s2 * m2

Learn():
s4 = s0 - s1
s3 = abs(s1)

Can random search be better?

Comparison with hand-designed reference algorithms

How frequently a randomly generated solution is better?

"Density" of acceptable algorithms

Difficulty as the logarithm of this density

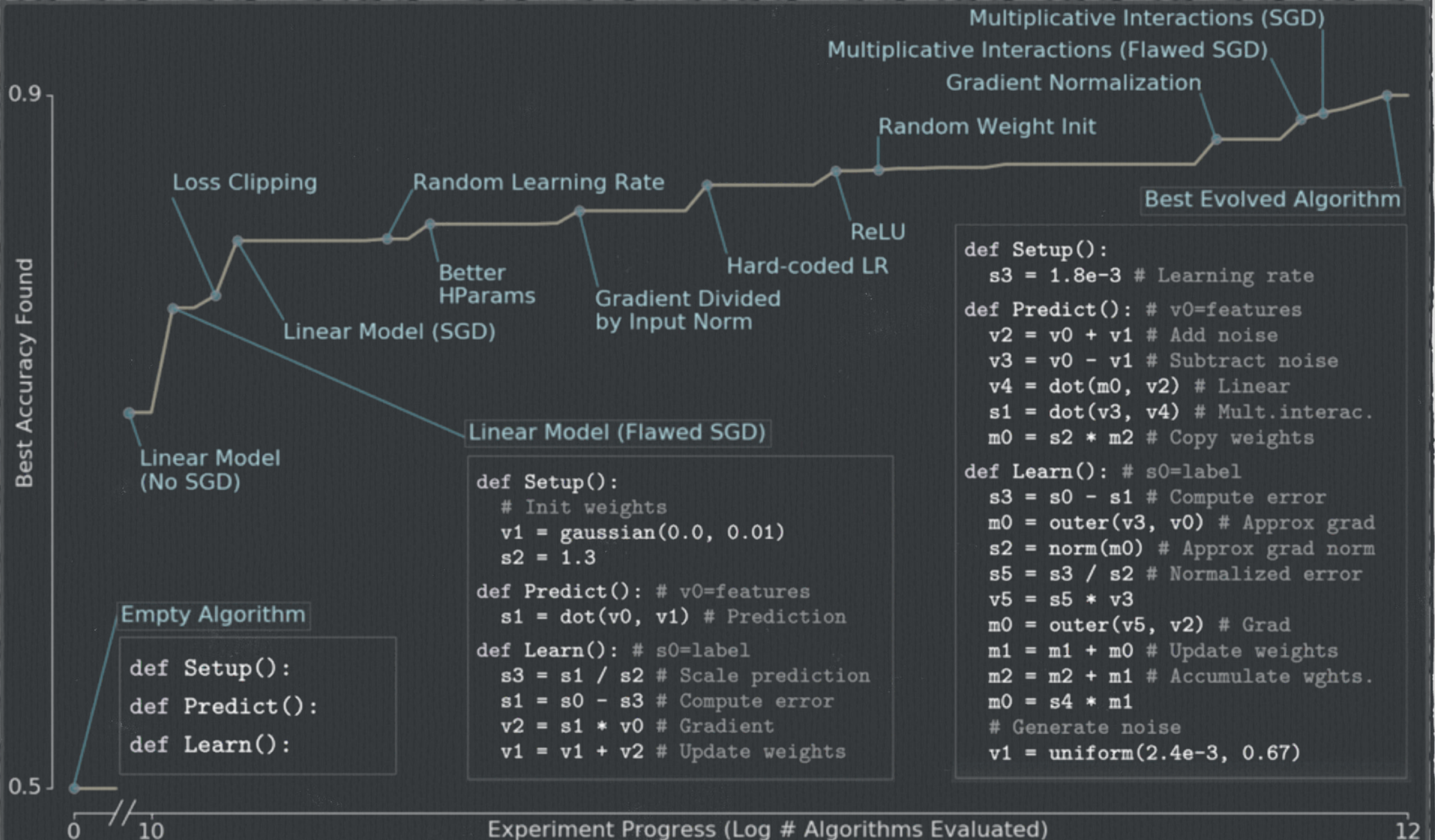| Task	Difficulty
Linear regressor	7.4
Affine regressor	12.1

Experiments

- ☐ **The classification tasks are extracted from MNIST e CIFAR-10**

- ☐ **Classification is only binary**

- ☐ **8000 training and 2000 validation samples each**

- ☐ **During the final evaluation the constants are treated as hyperparameters and tuned**

Results

Best Accuracy Found (y-axis, 0.5 to 0.9)

Experiment Progress (Log # Algorithms Evaluated) (x-axis, 0 to 12)

Labels on chart:
- Multiplicative Interactions (SGD)
- Multiplicative Interactions (Flawed SGD)
- Gradient Normalization
- Random Weight Init
- Loss Clipping
- Random Learning Rate
- ReLU
- Better HParams
- Hard-coded LR
- Linear Model (SGD)
- Gradient Divided by Input Norm
- Best Evolved Algorithm
- Linear Model (Flawed SGD)
- Linear Model (No SGD)
- Empty Algorithm

Empty Algorithm

```
def Setup():

def Predict():

def Learn():
```

Linear Model (Flawed SGD)

```
def Setup():
  # Init weights
  v1 = gaussian(0.0, 0.01)
  s2 = 1.3

def Predict(): # v0=features
  s1 = dot(v0, v1) # Prediction

def Learn(): # s0=label
  s3 = s1 / s2 # Scale prediction
  s1 = s0 - s3 # Compute error
  v2 = s1 * v0 # Gradient
  v1 = v1 + v2 # Update weights
```

Best Evolved Algorithm

```
def Setup():
  s3 = 1.8e-3 # Learning rate

def Predict(): # v0=features
  v2 = v0 + v1 # Add noise
  v3 = v0 - v1 # Subtract noise
  v4 = dot(m0, v2) # Linear
  s1 = dot(v3, v4) # Mult.interac.
  m0 = s2 * m2 # Copy weights

def Learn(): # s0=label
  s3 = s0 - s1 # Compute error
  m0 = outer(v3, v0) # Approx grad
  s2 = norm(m0) # Approx grad norm
  s5 = s3 / s2 # Normalized error
  v5 = s5 * v3
  m0 = outer(v5, v2) # Grad
  m1 = m1 + m0 # Update weights
  m2 = m2 + m1 # Accumulate wghts.
  m0 = s4 * m1
  # Generate noise
  v1 = uniform(2.4e-3, 0.67)
```

Evolved structures

☐ **Addition of noise to the input:**
$$a = x + u; b = x - u; u \sim U(\alpha, \beta)$$

☐ **Multiplicative interaction:**
$$o = a^T W b$$

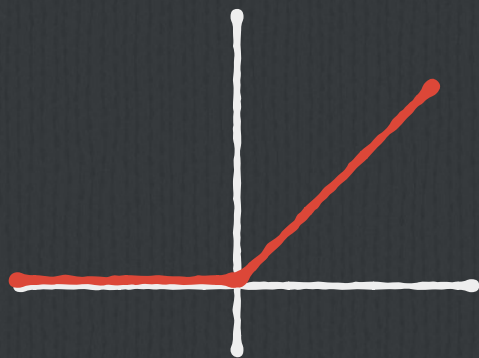☐ **Correct computation of the gradient, normalised to be a unit vector**

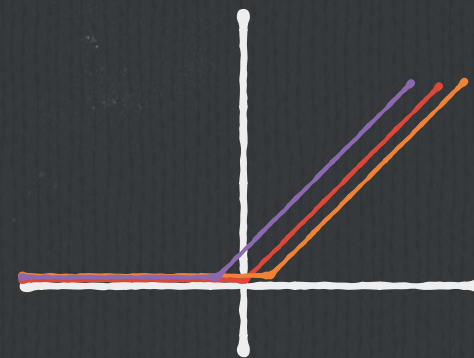☐ **Weight accumulation, work similarly to weight averaging**

Limiting the number of training samples

Training on 80 example for 100 epochs made
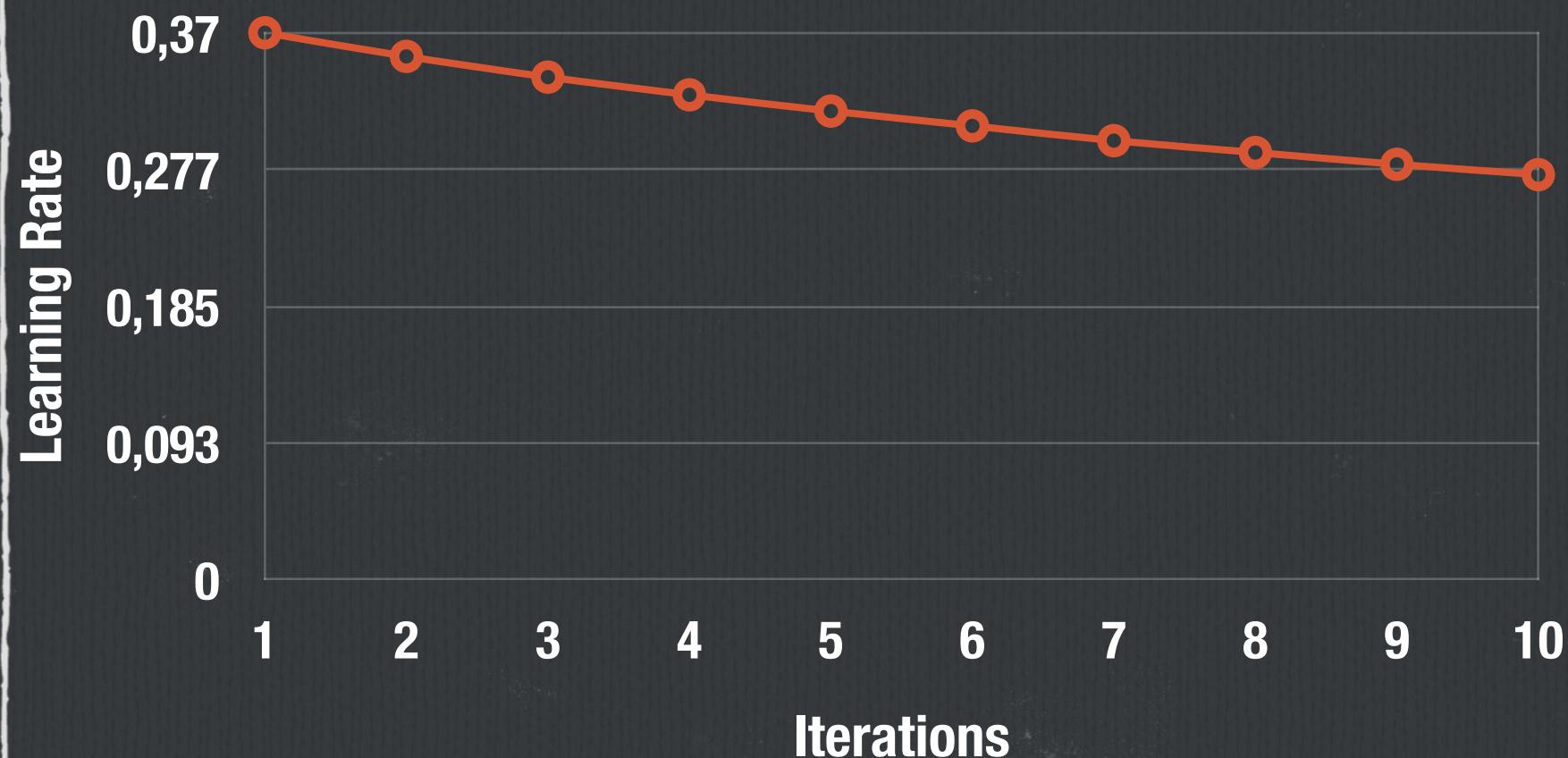the noisy ReLU replace the ReLU

$$f(x) = max(0,x)$$

$$f(x) = \max(0, x + y)$$

$$y \sim N(0, \sigma)$$

Acts similarly to regularisations like dropout

Multiple classes

All 10 CIFAR-10 classes are used.
An unexplained behaviour appears in 24/30 cases

Learn():
```
s3 = mean(m1)
s3 = abs(m1)
s3 = sin(s3)
...
```

Average of the weights

Transformation with $x \mapsto sin(|x|)$

Value now used as learning rate

Chen Liang @crazydonkey200 · Mar 11
Hi, one of the authors here. In section 4.3, we confirmed that it helps
through controlled experiments, but, to be honest, we didn't fully
understand the reason. As you pointed out, it could be a hyperparameter
coupling issue we discussed at the end of Supplementary section S.6.

Limitations

☐ **"Useless code" - bloat in GP - which leads to...**

☐ **Interpretability problems**

☐ **Inability to reuse code, which leads to...**

☐ **Difficulty in evolving deep structures.**

Linear GP

```
void gp(r)
  double r[8];
{  ...
  r[0] = r[5] + 71;
// r[7] = r[0] - 59;
  if (r[1] > 0)
  if (r[5] > 2)
    r[4] = r[2] * r[1];
// r[2] = r[5] + r[4];
  r[6] = r[4] * 13;
  r[1] = r[3] / 2;
// if (r[0] > r[1])
//    r[3] = r[5] * r[5];
  r[7] = r[6] - 2;
// r[5] = r[7] + 15;
  if (r[1] <= r[6])
    r[0] = sin(r[7]);
}
```

An example of Linear GP individual

Commented lines are introns,
with no effect on the output

Possible mutations:
1) macro mutations (instruction level)
2) micro mutations (inside the instruction)