On more interesting blocks

With discrete parameters in deep learning

Jialin Lu, July 8th 2020
Meeting of Ester Lab

SFU
SIMON FRASER UNIVERSITY
ENGAGING THE WORLD

Why I am talking about discretely-parameterized blocks?

I believe this should be fun and may potentially benefit some lab members in designing and optimizing novel deep learning blocks rather than conventional blocks like fully-connected layers.

The essence of today's talk is about giving technical solutions on how to learn/optimize the parameters. As you might know, optimizing continuous parameters is usually trivial by SGD or its variants, but not for discrete ones.

All the methods discussed today might not have a strong theoretical understanding yet, like convergence/guarantees/bounds, but they should useful. This is how I would like to do.

Outline of This talk

PART 1: Introducing interesting blocks with discrete parameters

I will start with conventional ones and then give two examplar discrete blocks.

PART 2: Learning in the general case

Not fun. Not efficient.

PART 3: Learning in the differentiable case.

The essential part of this talk.

PART 4: The sea of other possibilities

Introduce many other possibilities with discrete parameters.

Conclusion

PART 1

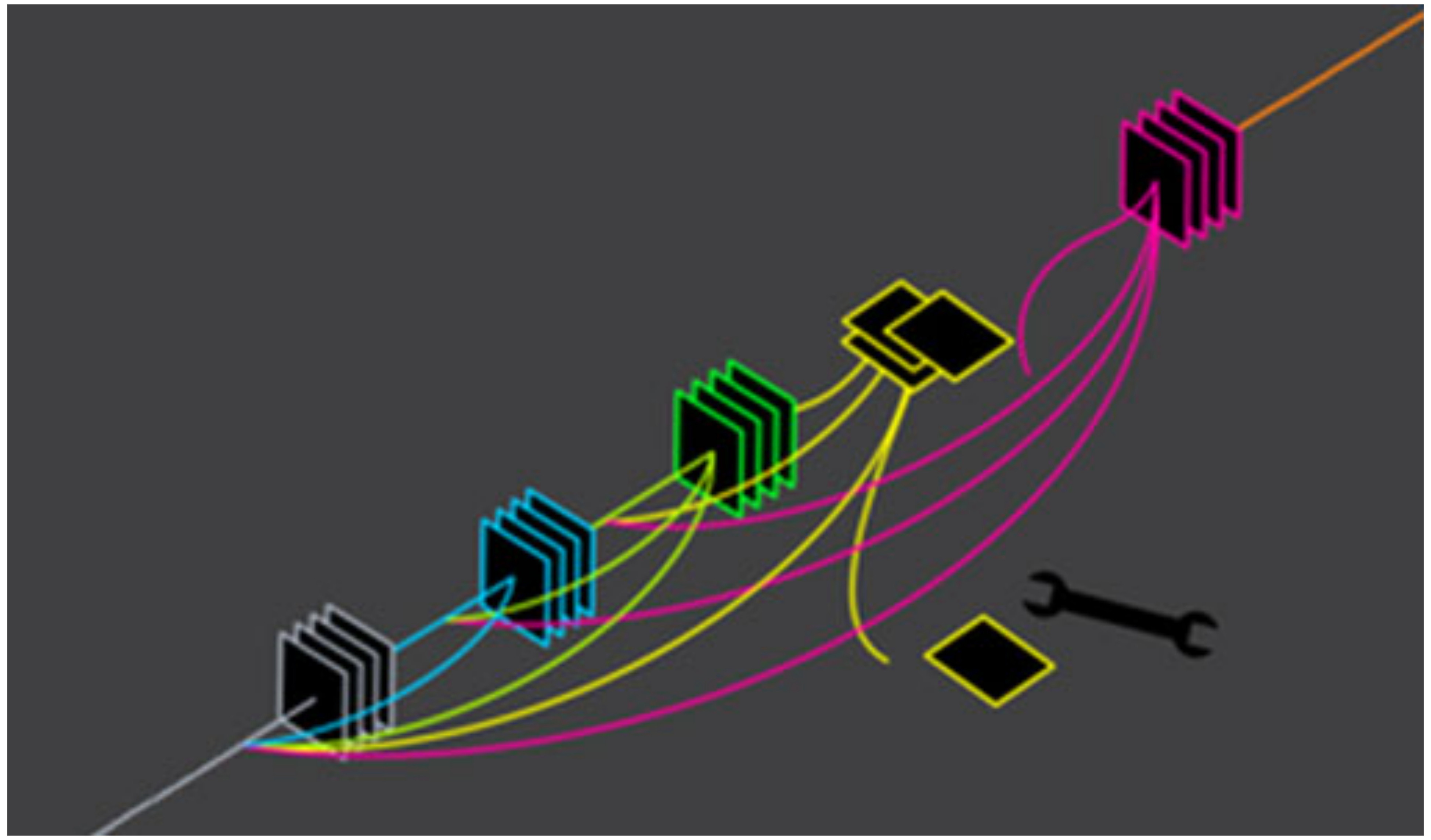
What is a more interesting block

with discrete parameters

But before that, why needs interesting block

It will be fun.

Deep learning is about freely assembling blocks.



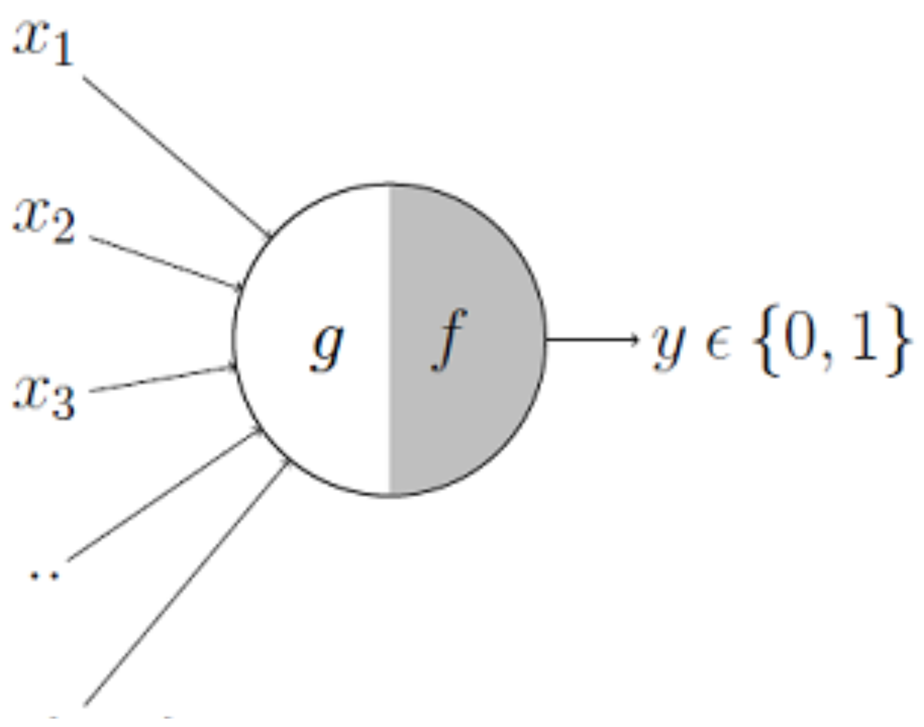
1. We can come up with customized blocks

2. We have efficient ways for optimization by gradient.

Conventional to interesting ones

Conventional block, fully connected layers, convolutional, recurrent etc.

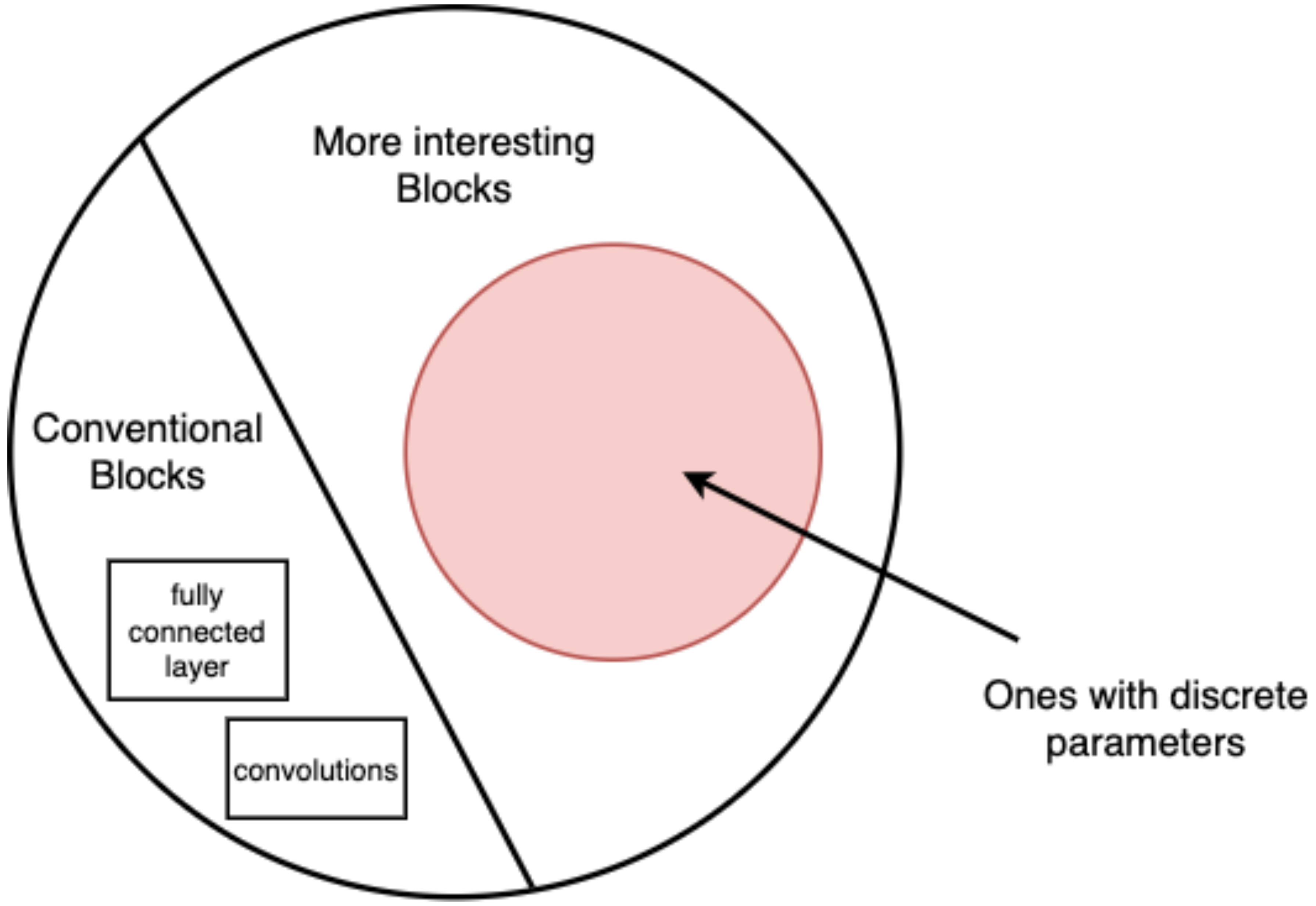
All based on the Pitts model (1943) the first model of the biological neuron



$$y = ReLU(Wx + b)$$

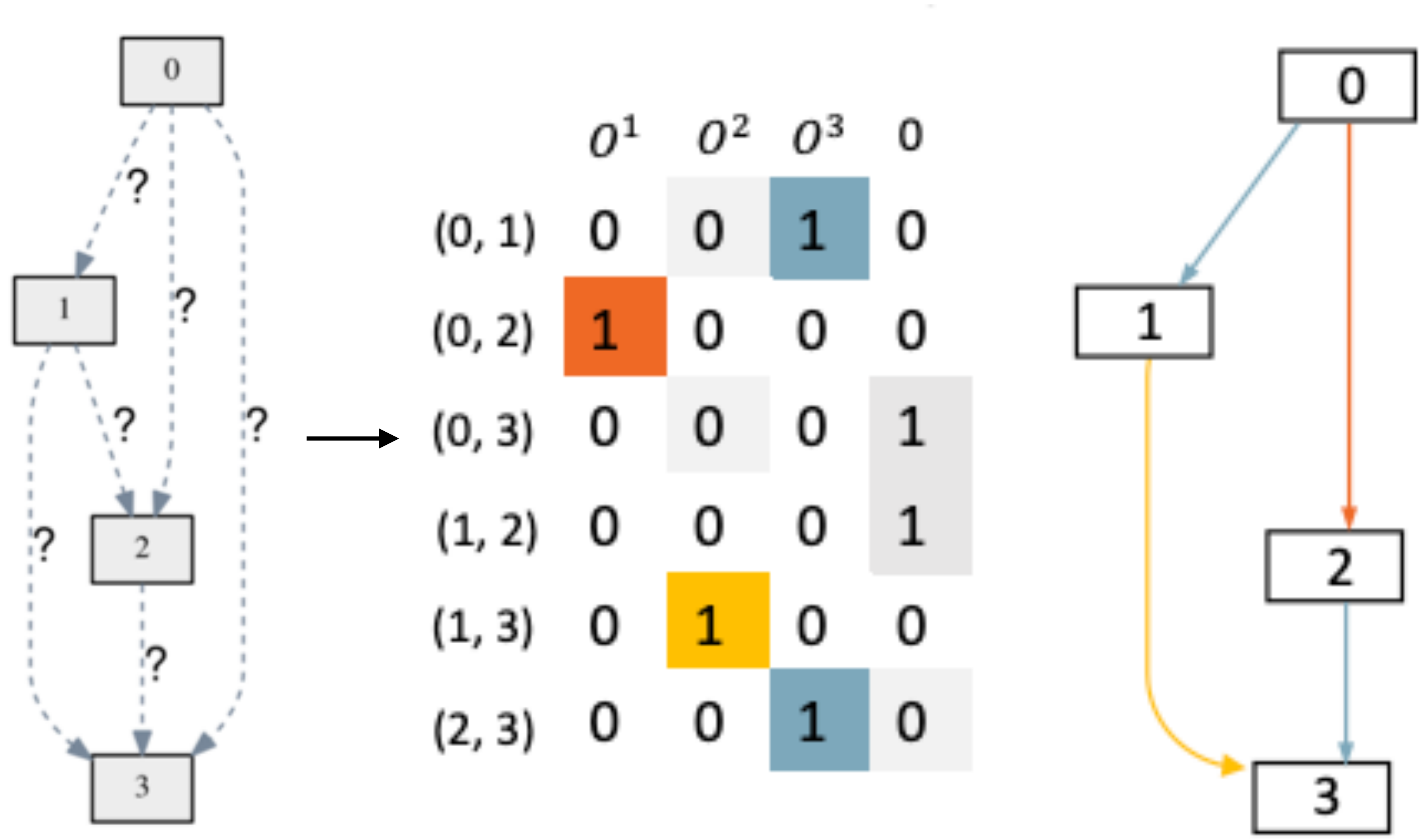
More modern, interesting ones.

We do not care whether these blocks are a good model for the biological neuron. We only need to do computation of our interest.



Two example

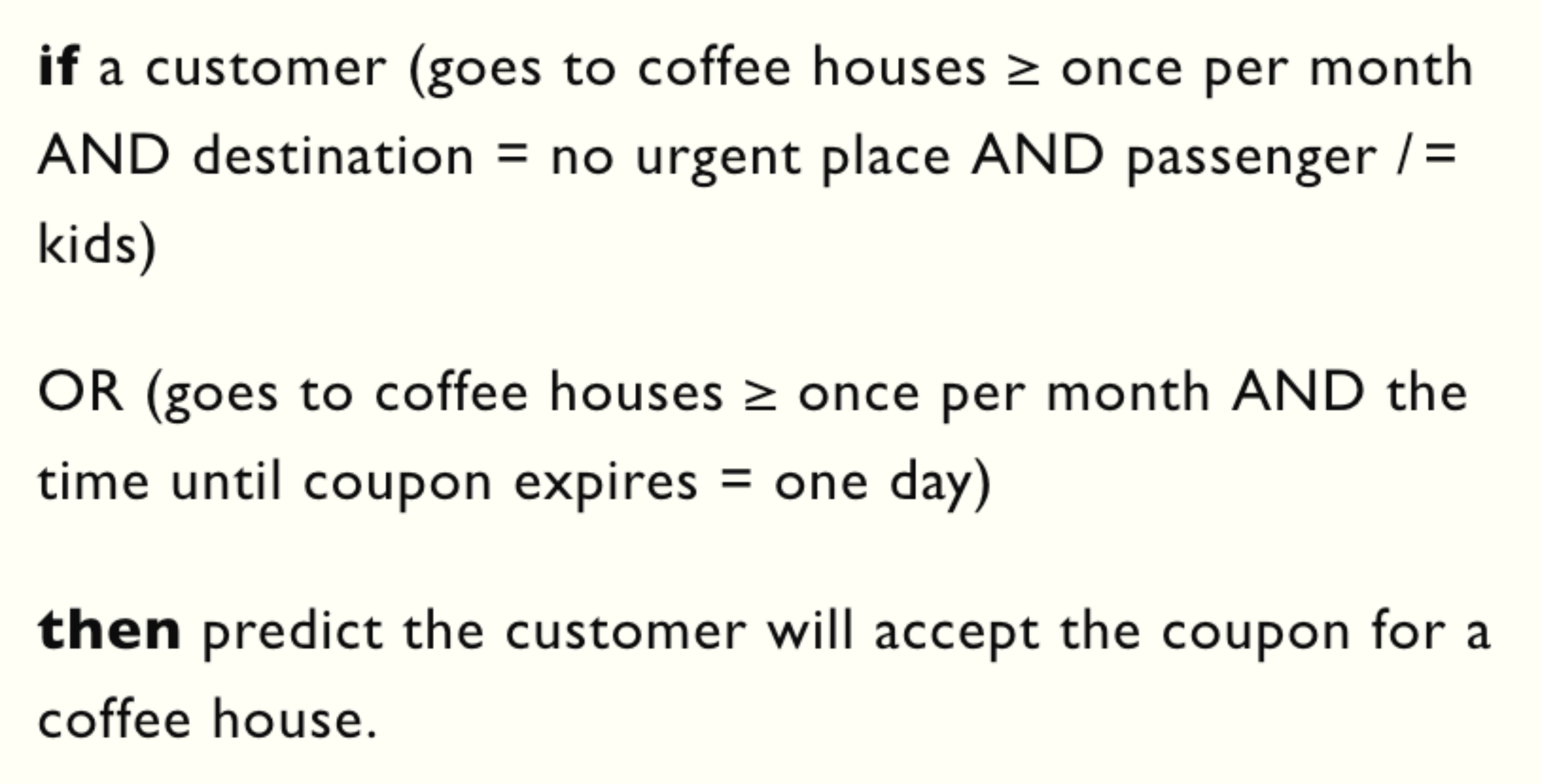
Neural Architecture Search



	o^1	o^2	o^3	0
(0, 1)	0	0	1	0
(0, 2)	1	0	0	0
(0, 3)	0	0	0	1
(1, 2)	0	0	0	1
(1, 3)	0	1	0	0
(2, 3)	0	0	1	0

Learning the routing pattern of some given blocks

Neural Disjunctive Normal Form



if a customer (goes to coffee houses ≥ once per month AND destination = no urgent place AND passenger /= kids)

OR (goes to coffee houses ≥ once per month AND the time until coupon expires = one day)

then predict the customer will accept the coupon for a coffee house.

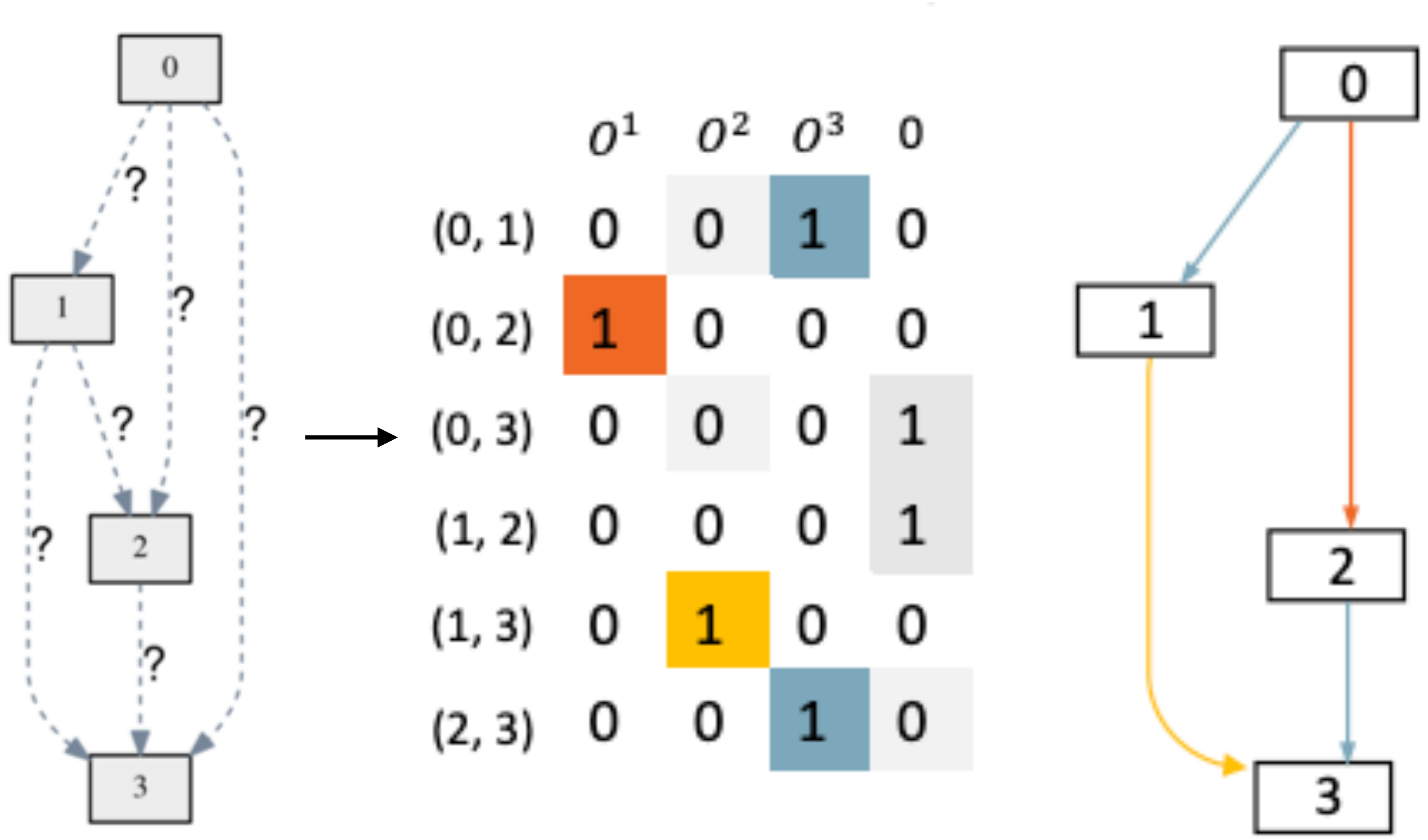
Learning discrete IF-THEN rules.

PART 2

Learning in the general case

Learning discrete blocks in the general case

We do not assume the block to be differentiable, so the discrete and continuous parameters need to be optimized separately. Better think in the problem of neural architecture search.



	o^1	o^2	o^3	0
(0, 1)	0	0	1	0
(0, 2)	1	0	0	0
(0, 3)	0	0	0	1
(1, 2)	0	0	0	1
(1, 3)	0	1	0	0
(2, 3)	0	0	1	0

Learning discrete blocks in the general case

We do not assume the block to be differentiable, so the discrete and continuous parameters need to be optimized separately. Better think in the problem of neural architecture search.

Good solutions include:

- Exhaustive search
- Graduate student local search
- Automatic solutions like some combinatorial algorithms, evolution, reinforment learning.

All the methods need to set a configuration of the discrete parameters, then learn continuous ones and then look at better configurations.

PART 3

Learning in differentiable case

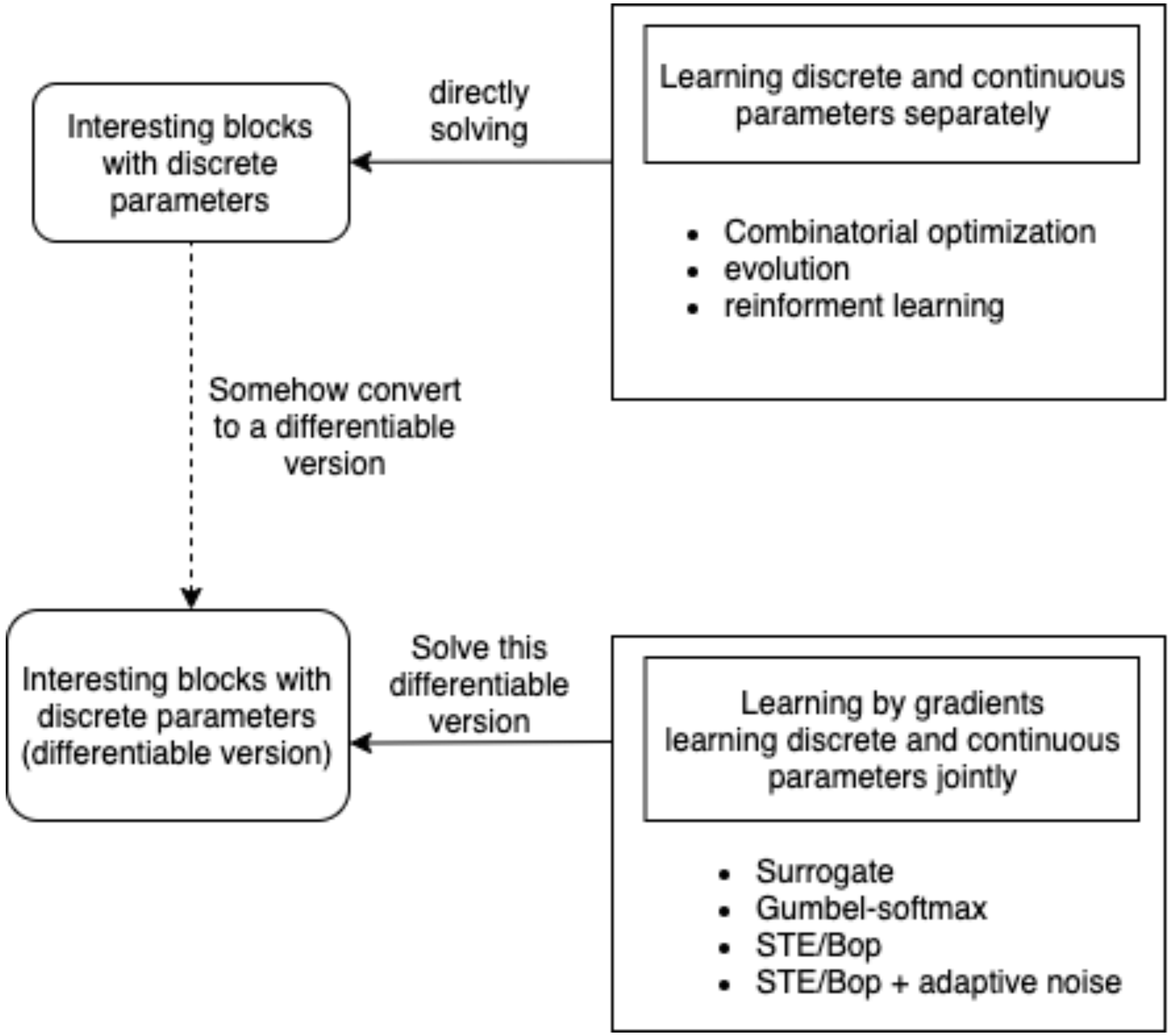
Why general case is bad

Learning in the general case is time-consuming, not efficient.

And in deep learning, non-efficiency often means inferior model performance

But if we can come up with a differentiable version of a block, we can optimize the discrete and continuous parameters at the same time.

If not originally differentiable, figure out one!



Interesting blocks with discrete parameters

directly solving

Learning discrete and continuous parameters separately

- Combinatorial optimization
- evolution
- reinforment learning

Somehow convert to a differentiable version

Interesting blocks with discrete parameters (differentiable version)

Solve this differentiable version

Learning by gradients learning discrete and continuous parameters jointly

- Surrogate
- Gumbel-softmax
- STE/Bop
- STE/Bop + adaptive noise

my comment:

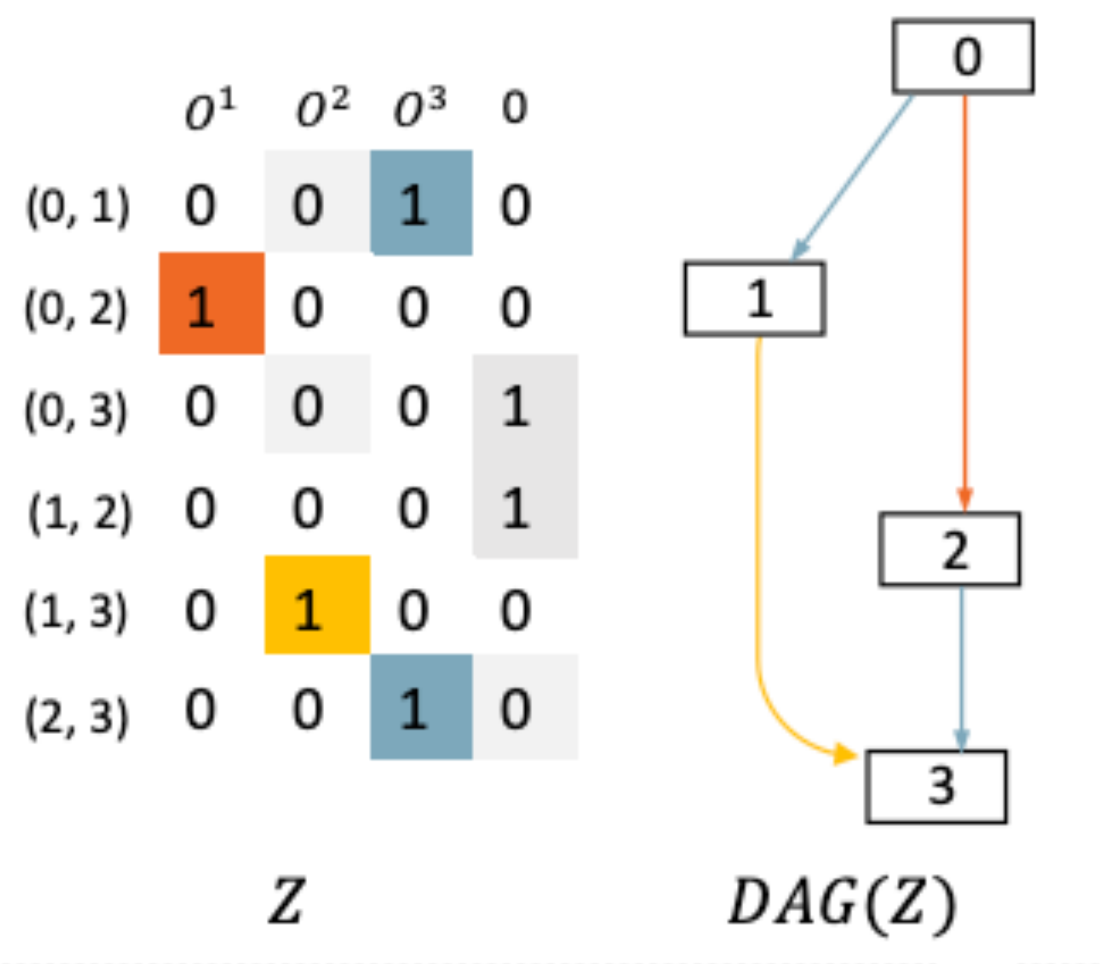
Applys generally
but not efficient.
Thus not always good performance

my comment:

Differentiable version of our two previous example

Neural Architecture Search

As a directed acyclic
computation graph.



$$\begin{array}{c|cccc} & o^1 & o^2 & o^3 & 0 \\ \hline (0,1) & 0 & 0 & 1 & 0 \\ (0,2) & 1 & 0 & 0 & 0 \\ (0,3) & 0 & 0 & 0 & 1 \\ (1,2) & 0 & 0 & 0 & 1 \\ (1,3) & 0 & 1 & 0 & 0 \\ (2,3) & 0 & 0 & 1 & 0 \end{array}$$

Neural Disjunctive Normal Form

The DNF logical operation $\hat{y} = \bigvee_{S_j=1}^{N} \bigwedge_{W_{i,j}=1} x_i$ can be parameterized by binary matrix W and S

$$r_j = \bigwedge_{W_{i,j}=1} x_i \xrightarrow{\text{replaced by}} r_j = \prod_i F_{\text{conj}}(x_i, W_{i,j}), \text{ where } F_{\text{conj}}(x, w) = 1 - w(1 - x)$$

$$\hat{y} = \bigvee_{S_j=1}^{N} r_j \xrightarrow{\text{replaced by}} \hat{y} = 1 - \prod_j^N (1 - F_{\text{disj}}(r_j, S_j)), \text{ where } F_{\text{disj}}(r, s) = r \cdot s$$

Finding a differentiable version will certainly take some effort, but not always impossible

Now the computation becomes differentiable. The blocks now is well defined on continuous values, it is just discrete parameters can only take discrete values.

Techniques for optimizing discrete parameters by gradient

- Continuous Surrogate
- Gumbel-softmax
- Straight-through estimator (STE)
- Binary Optimizer (Bop)
- A slightly improved version: STE/Bop + adaptive noise

By optimizing discrete parameters by gradient.

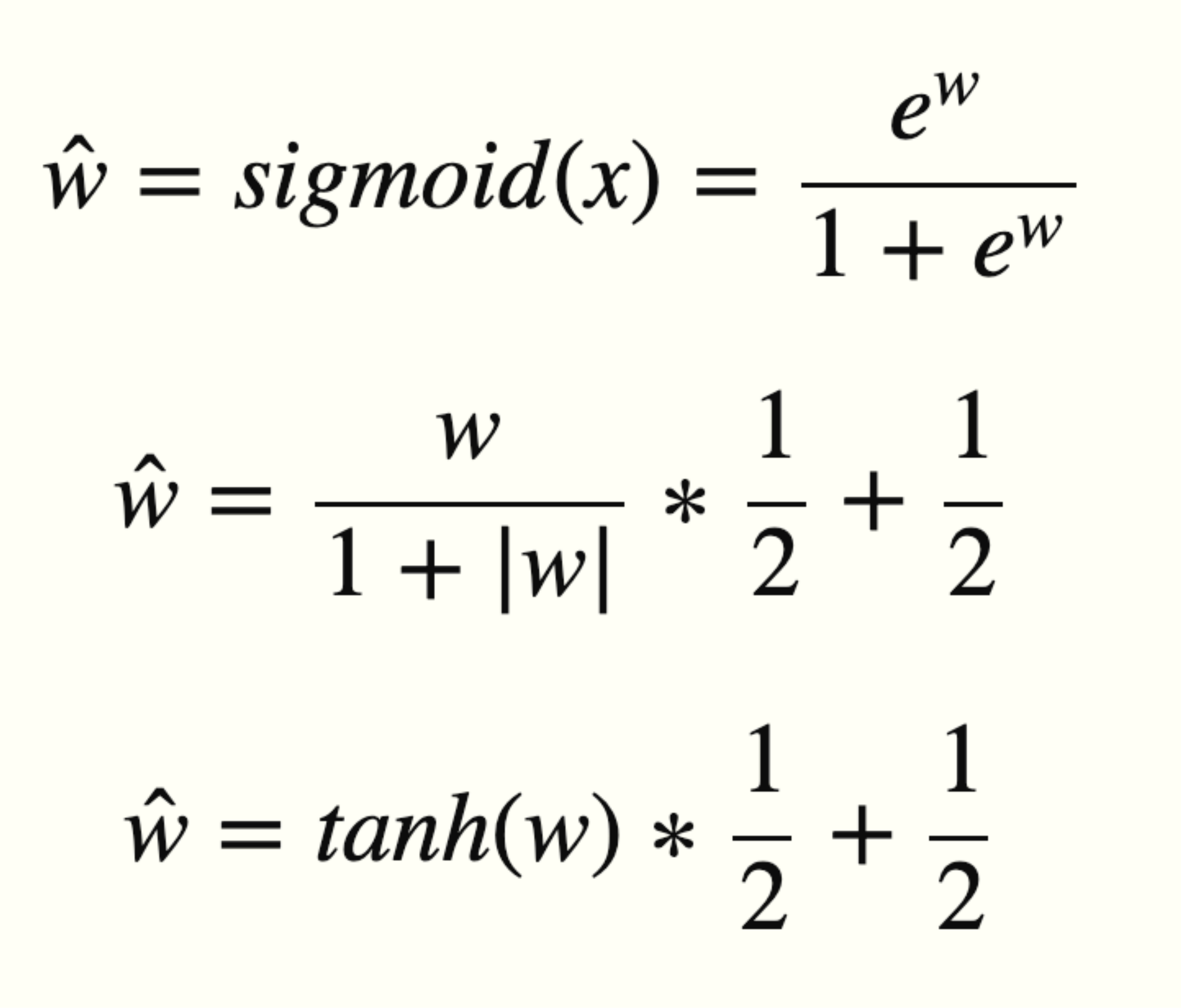
we can do joint optimization of discrete and continuous ones.

From now on, we assume the discrete parameters we want to learn is binary {0,1}

Continuous Surrogate

Use a continuous parameter, and apply a transformation function like sigmoid/ softsign/tanh.

Standard optimizer like SGD/Adam can be used.



$$\hat{w} = sigmoid(x) = \frac{e^w}{1 + e^w}$$

$$\hat{w} = \frac{w}{1 + |w|} * \frac{1}{2} + \frac{1}{2}$$

$$\hat{w} = tanh(w) * \frac{1}{2} + \frac{1}{2}$$

Pros: optimization is easy like any other continuous valued blocks

Cons: You do not always get discrete value or near-discrete value in the end. You need thresholding after training.

Continuous Surrogate, temperature-augmented

Use an extra hyper parameter, a temperature to control the closeness to discrete values.

We gradually increase it per epoch.

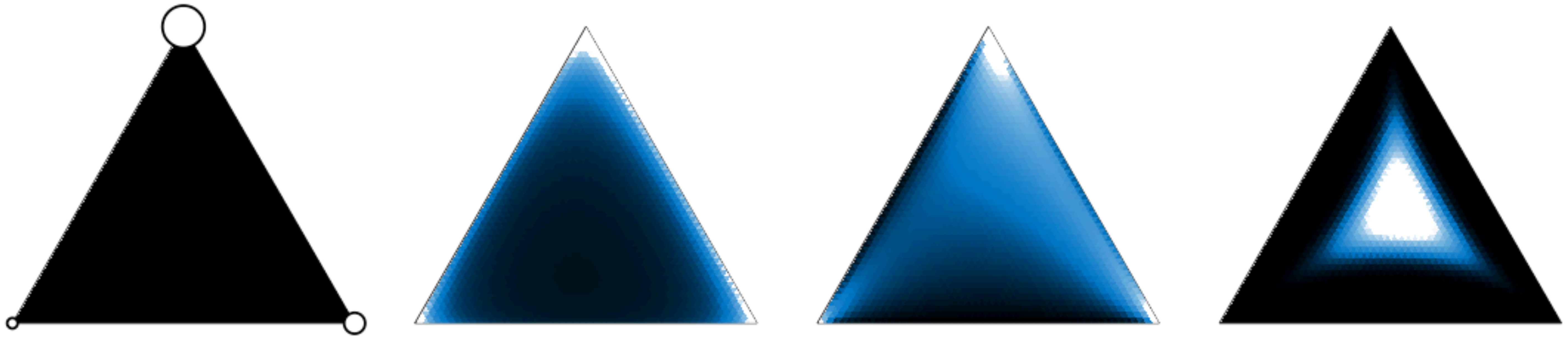
Good: might possibly obtain near-discrete value in the end.

Cons: You introduce a new hyper parameter. How to gradually increase the temperature need tedious tuning. This job might be quite difficult.

$$\hat{w} = sigmoid(w) = \frac{e^{\lambda w}}{1 + e^{\lambda w}}$$

$$\hat{w} = \frac{\lambda w}{1 + |\lambda w|} * \frac{1}{2} + \frac{1}{2}$$

$$\hat{w} = tanh(\lambda w) * \frac{1}{2} + \frac{1}{2}$$



(a) $\lambda = 0$ (b) $\lambda = 1/2$ (c) $\lambda = 1$ (d) $\lambda = 2$

Use a temperature to control the closeness to discrete values

$$\hat{w}_k = \frac{\exp((\log w_k) + g_k)/\lambda}{\sum_i^N \exp((\log w_i) + g_i)/\lambda}$$

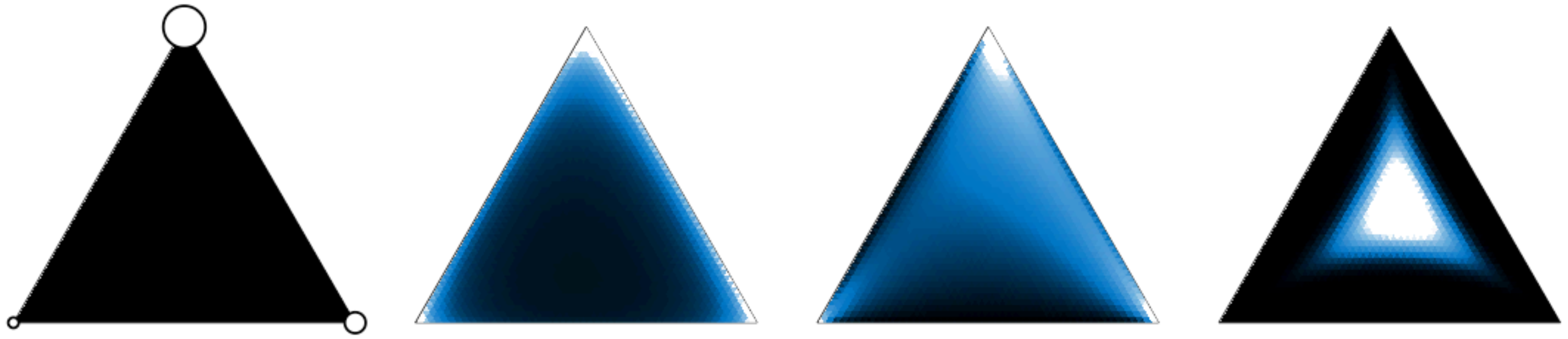
g is a noise drawn from a Gumbel distribution.

$$g_i = -\log(-\log(u)) \text{ where } u \sim Uniform(0, 1)$$

Standard optimizer like SGD/Adam can be used.

Gumbel-softmax trick

$$\hat{w}_k = \frac{\exp((\log w_k) + g_k)/\lambda}{\sum_i^N \exp((\log w_i) + g_i)/\lambda}$$



(a) $\lambda = 0$ (b) $\lambda = 1/2$ (c) $\lambda = 1$ (d) $\lambda = 2$

Pros:

A popular technique, widely used.

Cons:

1. Not so easy to work with as the temperature needs to be gradually decreased by some schedule.
2. What is more, only when the temperature is small, the value will be close to discrete value. But a small temperature also causes numerical instability (division by near-zero).

Straight-through estimator (STE)

First proposed in Hinton's lecture, and then analyzed by Bengio (2014).
The dominating technique used in binary neural network (weight {-1,1})

Conceptually very simple. In the forward pass, a real-valued parameter is thresholded by into discrete values and this discrete value is used for computation of the objective function.

In the backwards, the gradient is updated to the real-valued parameter.

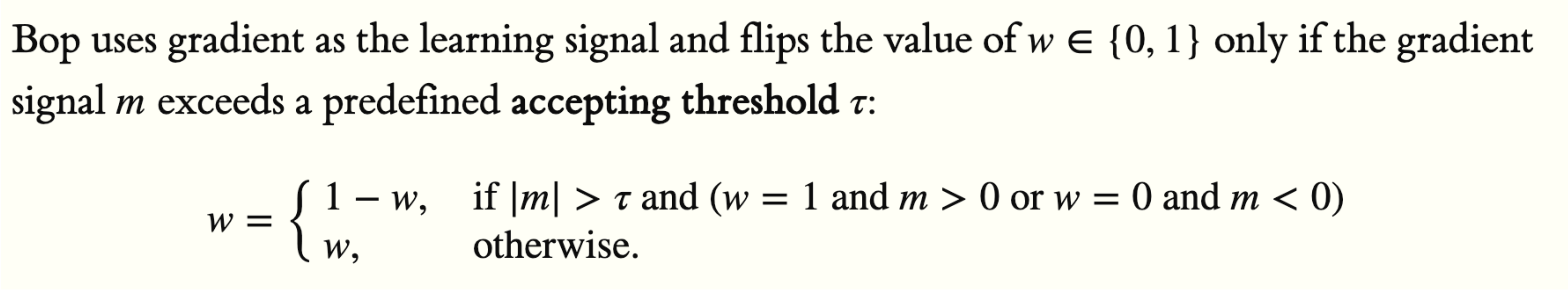
Optimized by standard optimizers like SGD/Adam.

Pros: works quite well for binary neural network, scalable, efficient.

Cons: Might get stuck in local minima and learning will fail for discrete blocks like Neural Disjunctive Normal Form. (reasons will be explained later)

Binary Optimizer (Bop)

First proposed by Helwegen et al (NeurIPS 2019). It argues that STE's real-valued *latent* parameter is not really necessary. We can consider a **new optimizer** that directly optimize discrete values using gradient, instead of SGD/Adam.



Bop uses gradient as the learning signal and flips the value of $w \in \{0, 1\}$ only if the gradient signal m exceeds a predefined **accepting threshold** τ:

$$w = \begin{cases} 1 - w, & \text{if } |m| > \tau \text{ and } (w = 1 \text{ and } m > 0 \text{ or } w = 0 \text{ and } m < 0) \\ w, & \text{otherwise.} \end{cases}$$

m is the gradient learning signal, computed as the exponential average of gradient (momentum)

Pros: Same as STE, works quite well for binary neural network, scalable, efficient. But the accepting threshold avoids rapid noisy flips and is intuitive to understand and tune.

Cons: Same as STE, might get stuck in local minima and learning will fail for discrete blocks like Neural Disjunctive Normal Form, (reasons will be explained later)

But all the mentioned methods is not so great

Continuous Surrogate

Optimization is okay like any continuous valued blocks, but no guarantees on binary values.

Temperature-augmented surrogate/Gumbel-softmax

Tuning temperature is so difficult. The convergence is more or less determined by how we do the temperature schedule, which is hard

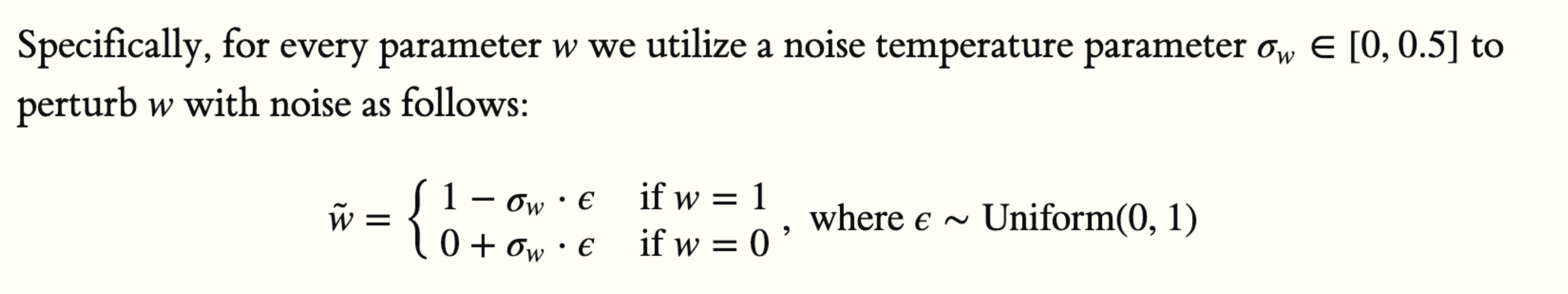
STE/Bop

Learning can stuck in local minima. Unlike binary fully-connected layer, for Neural DNF it simply does not work. Because when stuck in a local minima, all gradient w.r.t to the parameter be zero, thus learning fails. This is because for the discrete block computation, despite differentiable, the loss function is highly non-smooth.

A slight improvement: STE/Bop with adaptive noise

This improved is developed in our work for Neural DNF which can be applied to both STE/Bop.

We simply add noise to perturb the discrete value during forward computation of the objective function.

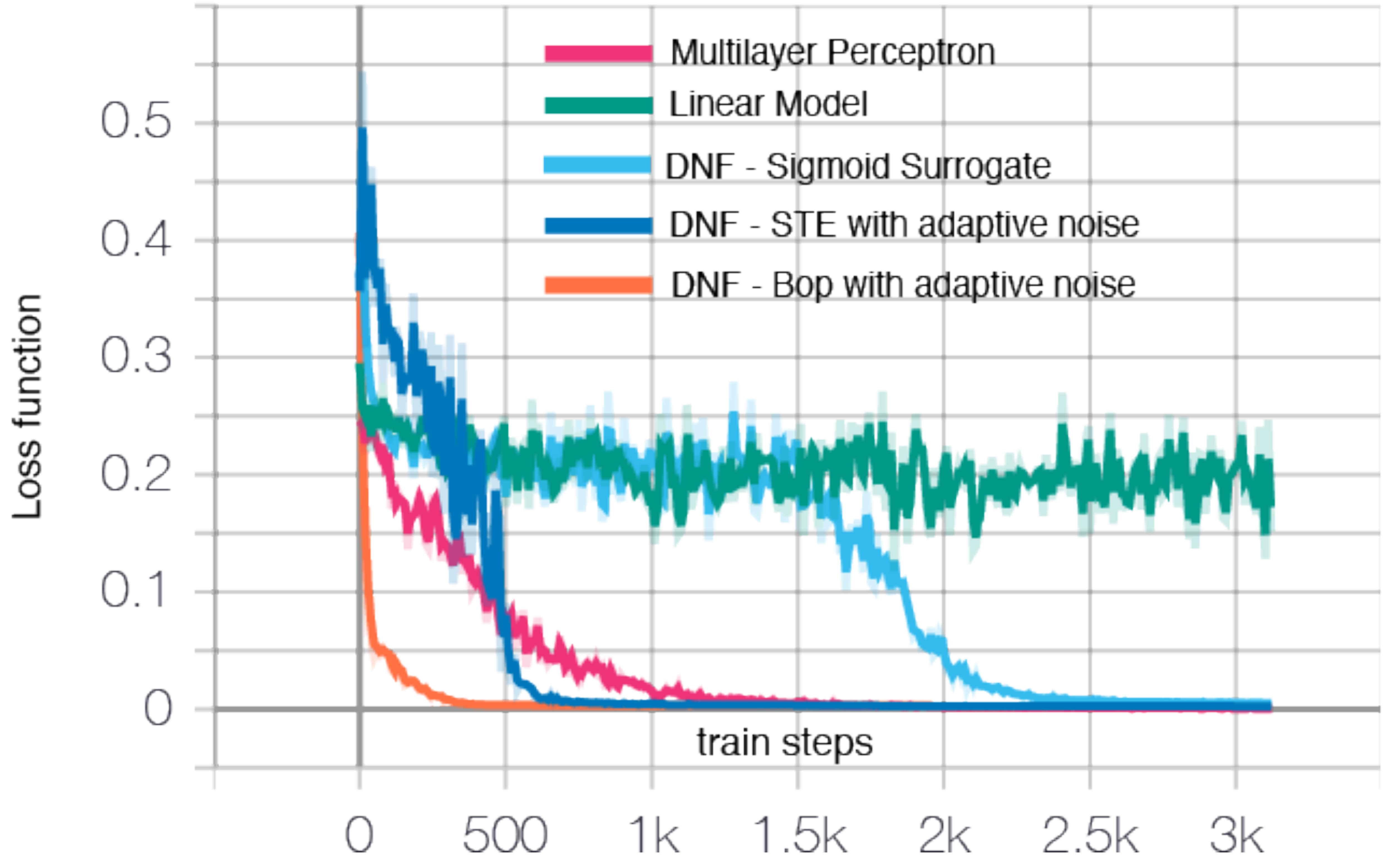


Specifically, for every parameter w we utilize a noise temperature parameter $\sigma_w \in [0, 0.5]$ to perturb w with noise as follows:

$$\tilde{w} = \begin{cases} 1 - \sigma_w \cdot \epsilon & \text{if } w = 1 \\ 0 + \sigma_w \cdot \epsilon & \text{if } w = 0 \end{cases}, \text{ where } \epsilon \sim \text{Uniform}(0, 1)$$

The new introduced parameter, the noise temperature, can also be optimized by standard continuous optimizer like SGD/Adam. So we do not need to do the tuning of temperature schedule.

A slight improvement: STE/Bop with adaptive noise



We apply Neural DNF on a synthetic dataset.

Both STE and Bop with adaptive noise give decent convergence speed. We did not show it, but STE or Bop alone do not converge. (learning always fails)

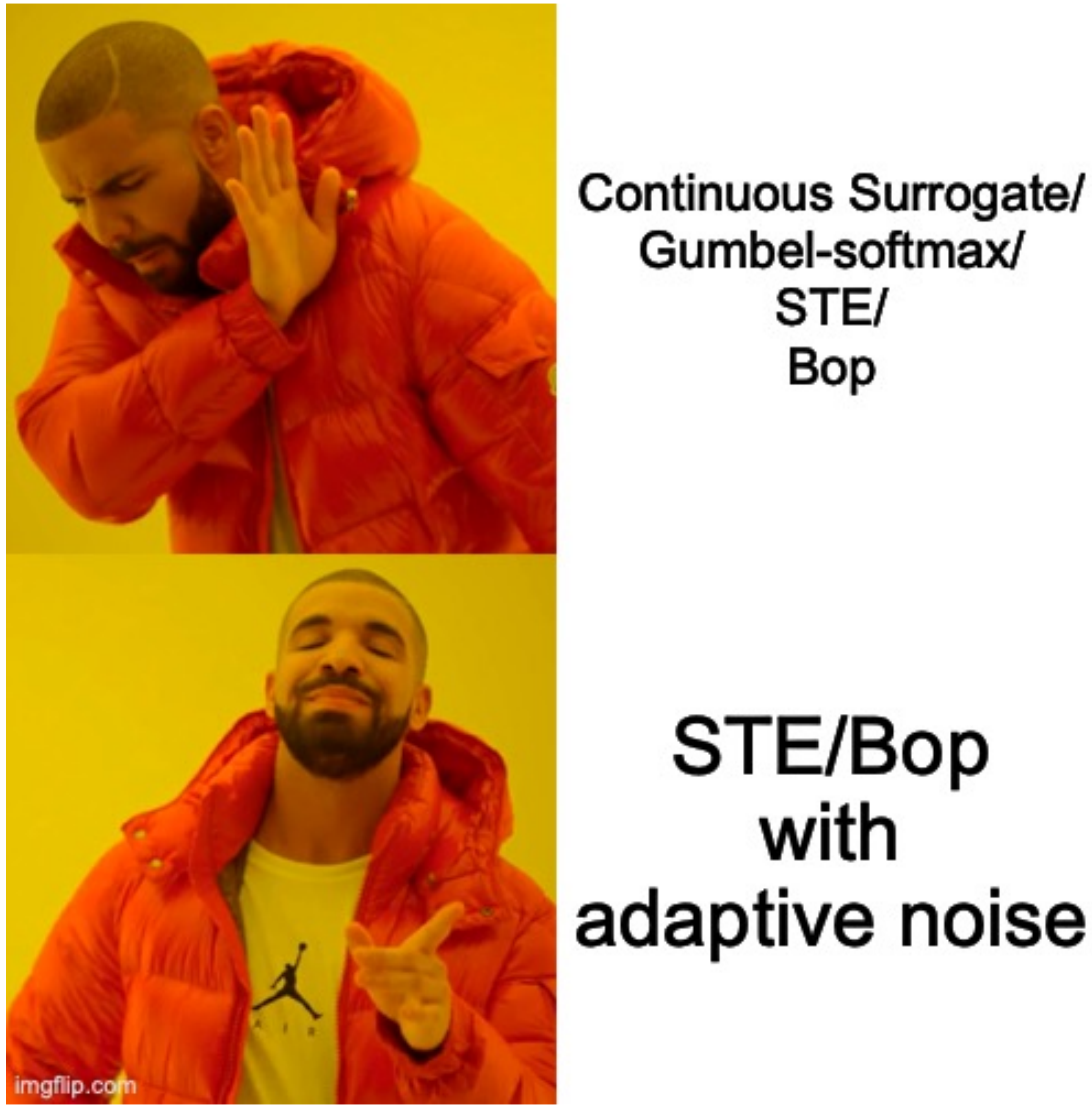
We do not show temeperatured surrogate and gumbel-softmax because for these two, convergence is determined by the temperature schedule. With tuning, we can fake any curve.

Pros: STE/Bop with adaptive noise requires minimal modification and give good results. The noise temperature is optimized as well so needs no tuning

Cons: We lack of theoretical understanding. We might be able to find a probabilistic interpretation, linking this adaptive noise to approximate variational inference.

So consider using it!

Simple solution, solve all the drawbacks of these alternative methods.



PART 4

The sea of other possibilities

Discrete not just in computation, but regularization

Sometimes, it is not the forward computation, but the objective function that has a discrete component.

We can use to train sparse neural network with L-0 regularizations.!

Simply add a “gate” binary parameter to each of the edges in a fully-connected layer or any other blocks. And L-0 regularization will minimize the non-zero edges!

You can find the reference in the reading web version

More other blocks: differentiable programs

The Disjunctive Normal Form is only the simplest program (a basic version of propositional logic) which means it only suits for binary classification.

But what about other tasks?

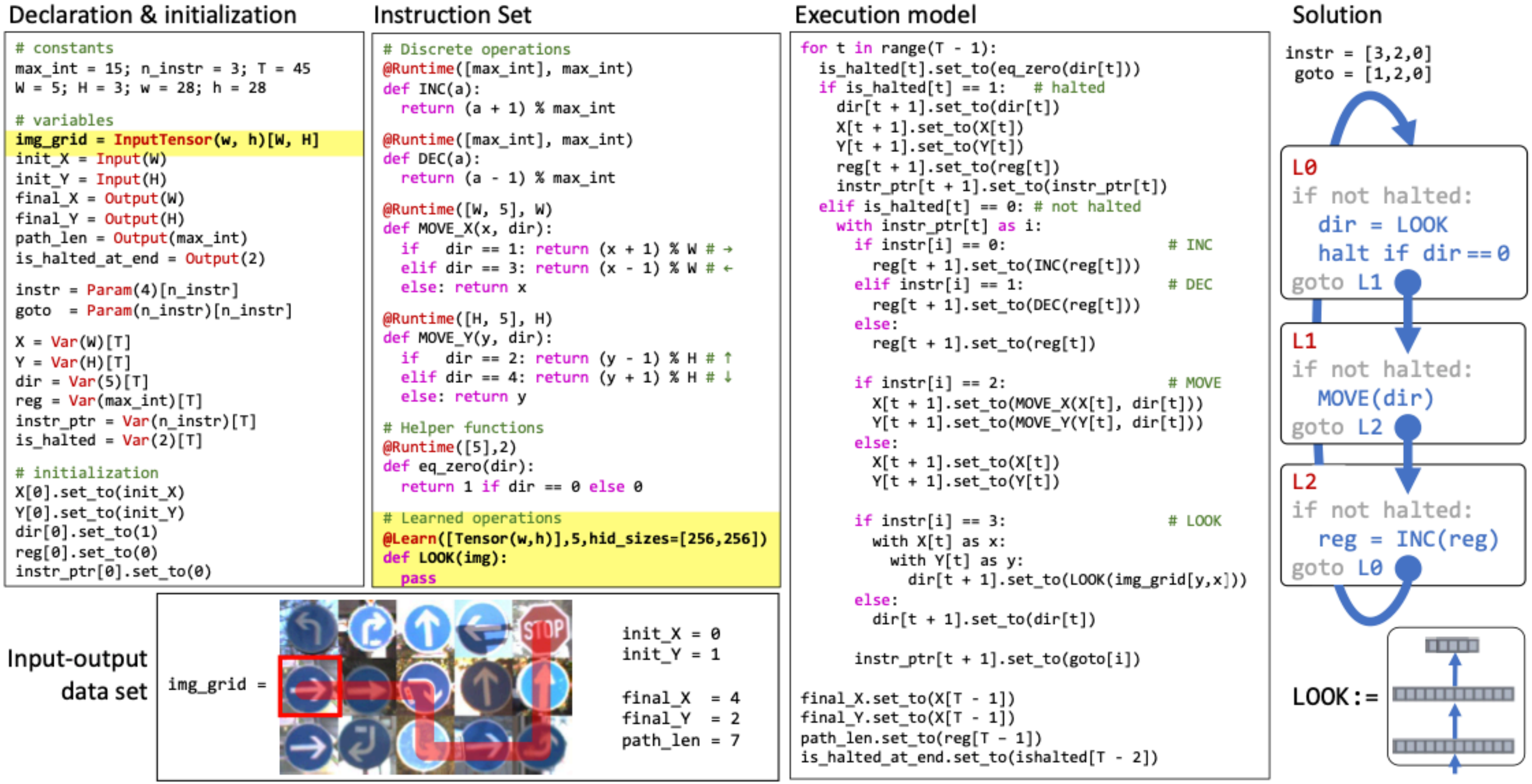
Differentiable program induction seems to be a promising direction!

(Program induction is to learn a program given input and output pairs.)

You write down a program template, leaving some learnable component, make a differentiable version of it. And then simply apply gradient-based learning!

An example, learn to measure the maze length from raw input

Using a hybrid model consists of a neural network and a program



Declaration & initialization

```
# constants
max_int = 15; n_instr = 3; T = 45
W = 5; H = 3; w = 28; h = 28

# variables
img_grid = InputTensor(w, h)[W, H]
init_X = Input(W)
init_Y = Input(H)
final_X = Output(W)
final_Y = Output(H)
path_len = Output(max_int)
is_halted_at_end = Output(2)

instr = Param(4)[n_instr]
goto  = Param(n_instr)[n_instr]

X = Var(W)[T]
Y = Var(H)[T]
dir = Var(5)[T]
reg = Var(max_int)[T]
instr_ptr = Var(n_instr)[T]
is_halted = Var(2)[T]

# initialization
X[0].set_to(init_X)
Y[0].set_to(init_Y)
dir[0].set_to(1)
reg[0].set_to(0)
instr_ptr[0].set_to(0)
```

Instruction Set

```
# Discrete operations
@Runtime([max_int], max_int)
def INC(a):
  return (a + 1) % max_int

@Runtime([max_int], max_int)
def DEC(a):
  return (a - 1) % max_int

@Runtime([W, 5], W)
def MOVE_X(x, dir):
  if   dir == 1: return (x + 1) % W # →
  elif dir == 3: return (x - 1) % W # ←
  else: return x

@Runtime([H, 5], H)
def MOVE_Y(y, dir):
  if   dir == 2: return (y - 1) % H # ↑
  elif dir == 4: return (y + 1) % H # ↓
  else: return y

# Helper functions
@Runtime([5],2)
def eq_zero(dir):
  return 1 if dir == 0 else 0

# Learned operations
@Learn([Tensor(w,h)],5,hid_sizes=[256,256])
def LOOK(img):
  pass
```

Execution model

```
for t in range(T - 1):
  is_halted[t].set_to(eq_zero(dir[t]))
  if is_halted[t] == 1:   # halted
    dir[t + 1].set_to(dir[t])
    X[t + 1].set_to(X[t])
    Y[t + 1].set_to(Y[t])
    reg[t + 1].set_to(reg[t])
    instr_ptr[t + 1].set_to(instr_ptr[t])
  elif is_halted[t] == 0: # not halted
    with instr_ptr[t] as i:
      if instr[i] == 0:                   # INC
        reg[t + 1].set_to(INC(reg[t]))
      elif instr[i] == 1:                 # DEC
        reg[t + 1].set_to(DEC(reg[t]))
      else:
        reg[t + 1].set_to(reg[t])

      if instr[i] == 2:                   # MOVE
        X[t + 1].set_to(MOVE_X(X[t], dir[t]))
        Y[t + 1].set_to(MOVE_Y(Y[t], dir[t]))
      else:
        X[t + 1].set_to(X[t])
        Y[t + 1].set_to(Y[t])

      if instr[i] == 3:                   # LOOK
        with X[t] as x:
          with Y[t] as y:
            dir[t + 1].set_to(LOOK(img_grid[y,x]))
      else:
        dir[t + 1].set_to(dir[t])

      instr_ptr[t + 1].set_to(goto[i])

final_X.set_to(X[T - 1])
final_Y.set_to(X[T - 1])
path_len.set_to(reg[T - 1])
is_halted_at_end.set_to(ishalted[T - 2])
```

Solution

```
instr = [3,2,0]
 goto = [1,2,0]
```

```
L0
if not halted:
  dir = LOOK
  halt if dir == 0
goto L1

L1
if not halted:
  MOVE(dir)
goto L2

L2
if not halted:
  reg = INC(reg)
goto L0
```

LOOK :=

Input-output data set

```
img_grid =

init_X = 0
init_Y = 1

final_X   = 4
final_Y   = 2
path_len  = 7
```

The merits of differentiable programs

You get some sense of control by incorporating task specification and human knowledge into the form of program

The resulted program is certainly transparent and interpretable.

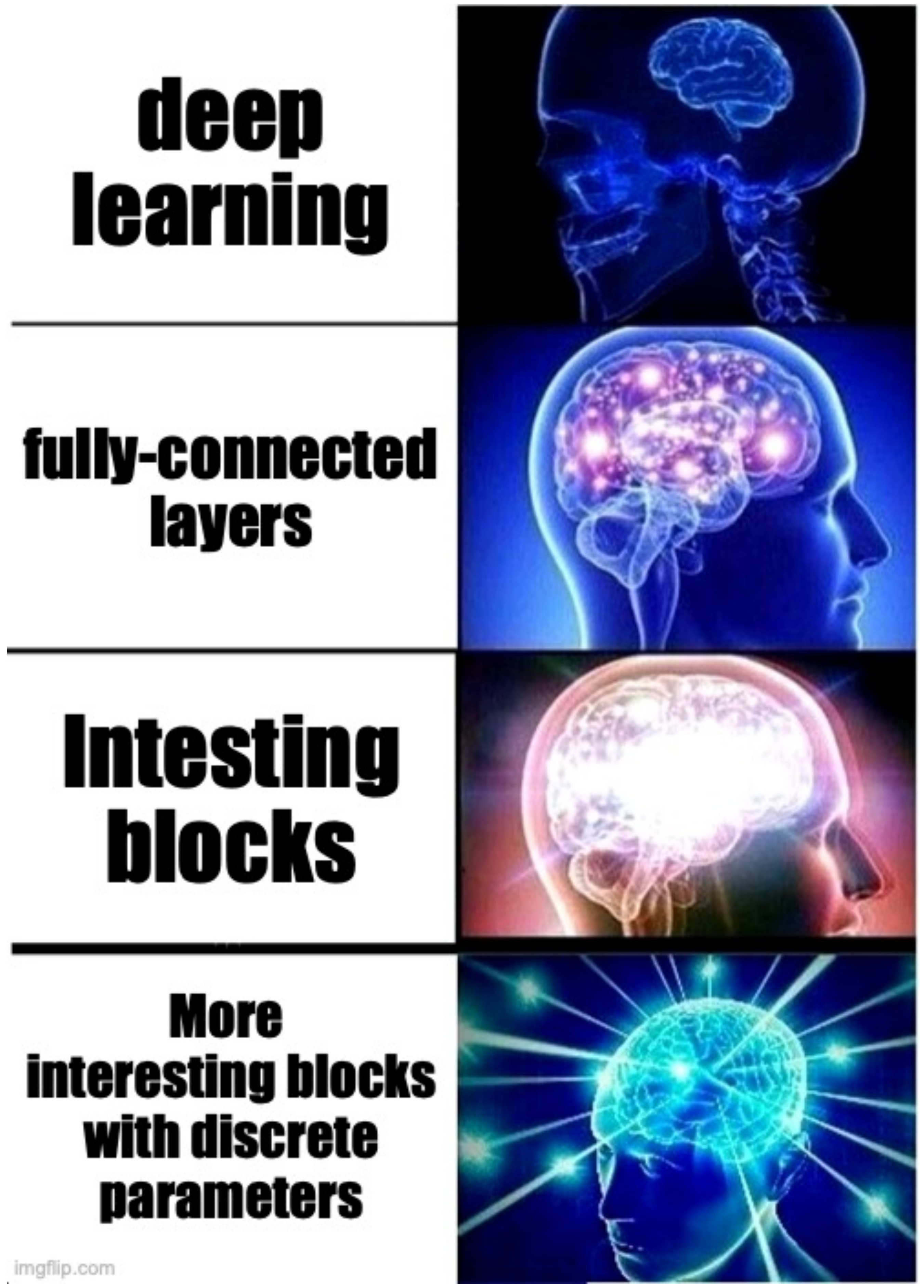
The program can be trained jointly with any other neural networks. You can let the neural network to do perception of patterns from raw data and let the program to do some high-level computations.

Customize task-specific blocks/programs!

Conclusion

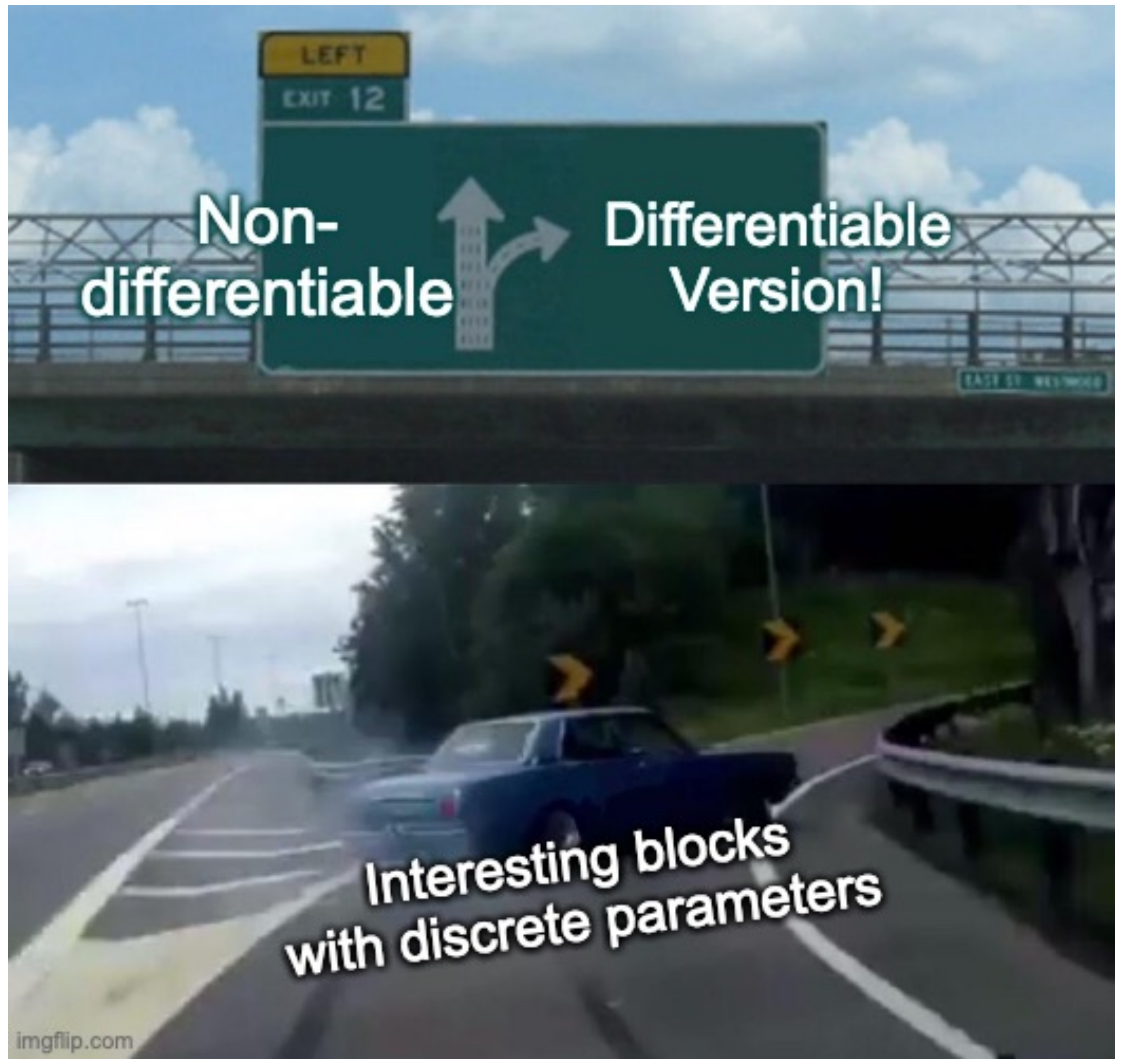
Message 1: think in more interesting blocks

Thinking in more interesting blocks, instead of conventional blocks such as fully-connected, convolutional layers, should open some possibilities on building advanced deep learning models.



Message 2: make differentiable!

When you come up with a block with discrete parameters, it is better to think of a differentiable version. Because this way we can get more efficient learning that we can jointly learn discrete and continuous parameters at the same time.



Message 3: optimize using gradient!

We introduced several alternative methods like continuous surrogate, gumbel-softmax, straight-through estimator (STE), Binary Optimizer (Bop), and a slight improvement STE/Bop with adaptive noise.

Continuous Surrogate/
Gumbel-softmax/
STE/
Bop

STE/Bop
with
adaptive noise

Summary

- Thinking in more interesting blocks, instead of conventional blocks such as fully-connected, convolutional layers, should open some possibilities on building advanced deep learning models.
- when you come up with a block with discrete parameters, it is better to think of a differentiable version. Because this way we can get more efficient learning that we can jointly learn discrete and continuous parameters at the same time.
- We introduced several alternative methods like continuous surrogate, gumbel-softmax, straight-through estimator (STE), Binary Optimizer (Bop), and a slight improvement STE/Bop with adaptive noise.

Thank you.