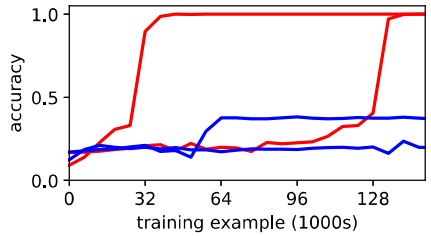

Figure 9: Example learning trajectories on the street sign task from Fig. 1. We show the accuracy on a test set during training for two successful random restarts (red) and two unsuccessful restarts (blue).

A. Street sign mazes

To introduce the NTPT modelling language, we provided an illustrative task in Fig. 1: Given a grid of $W \times H$ directional street signs, follow arrows and measure the length of a path from a randomly chosen start square to a randomly placed stop sign. For completeness, we provide a short description of experiments on this task.

To generate a dataset we selected traffic signs depicting arrows and stop signs from the GTRSB dataset (Stallkamp et al., 2011). These signs were cropped to the bounding box specified in the data set, converted to grayscale and resized to 28×28 pixels. Grids of $W \times H$ signs were constructed, each containing a path of arrows leading from a randomly chosen start square to a single randomly placed stop sign (grid cells not on this path were populated with random arrows). Each maze of signs is labeled with the length of the path between the start and stop cells (following the arrows). Without any direct supervision of what path to take, the system should learn a program that takes a grid of images and returns the path length. We train on 10^4 mazes of $W \times H = 3 \times 2$ and test on 10^3 unseen 3×2 mazes. We keep the set of sign images used to construct the training and test data distinct. After training, we inspect the learned algorithm (see Fig. 1), and see that the system has learned to use the instructions `LOOK`, `MOVE` and `INC` in a loop that generalizes to larger W and H.

Empirically, we find that 2% of random restarts successfully converge to a program that generalizes. Unsuccessful restarts stall in local optimization minima, and we present examples of successful and unsuccessful training trajectories in Fig. 9.

B. Model source code

We provide the NTPT source code for the models described in the main text.

B.1. ADD2X2 scenario

This model learns straight-line code to navigate an accumulate a 2×2 grid of MNIST tiles.

```
# constants
IMAGE_SIZE = 28;              NUM_INSTR = 6
NUM_DIGITS = 10;              T = 5
MAX_INT = 2*NUM_DIGITS - 1;   NUM_NETS = 2
GRID_SIZE = 2

# variables
tiles = InputTensor(IMAGE_SIZE, IMAGE_SIZE)[4]
final_sum = Output(MAX_INT)
instr = Param(NUM_INSTR)[T - 1]
arg1s = Param(T)[T - 1]
arg2s = Param(T)[T - 1]
return_reg = Param(T)
net_choice = Param(NUM_NETS)

pos = Var(4)[T]
registers = Var(MAX_INT)[T, T]

tmp_0 = Var(10)[T]
tmp_1 = Var(4)[T]

# functions
@Learn([Vector(IMAGE_SIZE, IMAGE_SIZE)], 10,
 hid_sizes = [256,256])
def net_0(img): pass

@Learn([Vector(IMAGE_SIZE, IMAGE_SIZE)], 4,
 hid_sizes = [256,256])
def net_1(img): pass

@Runtime([  MAX_INT,   MAX_INT],   MAX_INT)
def Add(a, b): return (a + b)

@Runtime([4], 4)
def move_east(pos):
  x = pos % GRID_SIZE
  y = pos / GRID_SIZE
  x = x+1 if x+1 < GRID_SIZE else (GRID_SIZE - 1)
  new_pos = GRID_SIZE * y + x
  return new_pos
#... similar definitions for move_south/west/north

@Runtime([4], MAX_INT)
def embed_op_code(c): return c

@Runtime([10], MAX_INT)
def embed_digit(c): return c

# initialization
pos[0].set_to(0)

if net_choice == 0:
  tmp_0[0].set_to(net_0(tiles[0]))
  registers[0, 0].set_to(embed_digit(tmp_0[0]))
elif net_choice == 1:
  tmp_1[0].set_to(net_1(tiles[0]))
  registers[0, 0].set_to(embed_op_code(tmp_1[0]))
for r in range(1, T):
  registers[0, r].set_to(0)

# execution model
for t in range(T - 1):
  if instr[t] == 0: # NOOP
    pos[t + 1].set_to(pos[t])
    registers[t + 1, t + 1].set_to(0)
  elif instr[t] == 1: # ADD
    with arg1s[t] as a1:
      with arg2s[t] as a2:
        registers[t + 1, t + 1].set_to(
         Add(registers[t, a1], registers[t, a2]))
    pos[t + 1].set_to(pos[t])
  elif instr[t] == 2: # MOVE-EAST
    pos[t + 1].set_to(move_east(pos[t]))
```

```
    with pos[t + 1] as p:
      if net_choice == 0:
        tmp_0[t+1].set_to(net_0(tiles[p]))
        registers[t+1, t+1].set_to(
         embed_digit(tmp_0[t+1]))
      elif net_choice == 1:
        tmp_1[t+1].set_to(net_1(tiles[p]))
        registers[t+1, t+1].set_to(
         embed_op_code(tmp_1t[t+1]))
  #... similar cases for move_south/west/north

  for r in range(t + 1):
    registers[t + 1, r].set_to(registers[t, r])
  for r in range(t + 2,  T):
    registers[t + 1, r].set_to(registers[t, r])

with return_reg as r:
  final_sum.set_to(registers[T - 1, r])
```

B.2. APPLY2X2 scenario

This model is a small variation on the above to navigate a 2×2 grid of operator tiles and apply the operators to `aux_ints`.

```
# constants
IMAGE_SIZE = 28;              NUM_INSTR = 6
NUM_DIGITS = 10;              T = 5
MAX_INT = 2*NUM_DIGITS - 1;   NUM_NETS = 2

# variables
tiles = InputTensor(IMAGE_SIZE, IMAGE_SIZE)[4]
aux_ints = Input(MAX_INT)[3]
final_sum = Output(MAX_INT)
instr = Param(NUM_INSTR)[T - 1]
arg1s = Param(T + 3)[T - 1]
arg2s = Param(T + 3)[T - 1]
arg3s = Param(T + 3)[T - 1]
return_reg = Param(T)
net_choice = Param(NUM_NETS)

pos = Var(4)[T]
registers = Var(MAX_INT)[T, T+3]

tmp_0 = Var(10)[T]
tmp_1 = Var(4)[T]
tmp_opcode = Var(4)[T - 1]

# functions
@Learn([Vector(IMAGE_SIZE, IMAGE_SIZE)], 10,
 hid_sizes = [256,256])
def net_0(img): pass

@Learn([Vector(IMAGE_SIZE, IMAGE_SIZE)], 4,
 hid_sizes = [256,256])
def net_1(img): pass

@Runtime([MAX_INT, MAX_INT, 4], MAX_INT)
def Apply(a, b, op):
  if op == 0:   return (a + b) % MAX_INT
  elif op == 1: return (a - b) % MAX_INT
  elif op == 2: return (a * b) % MAX_INT
  elif op == 3: return (MAX_INT-1 if b == 0
                  else int(a / b) % MAX_INT)
  else: return a

@Runtime([4], 4)
def move_east(pos):
  x = pos % GRID_SIZE
  y = pos / GRID_SIZE
  x = x+1 if x+1 < GRID_SIZE else (GRID_SIZE - 1)
  new_pos = GRID_SIZE * y + x
  return new_pos
#... similar definitions for move_south/west/north

@Runtime([MAX_INT], 4)
def extract_op_code(c): return c if c < 4 else 0

@Runtime([4], MAX_INT)
def embed_op_code(c): return c

@Runtime([10], MAX_INT)
def embed_digit(c): return c
```

```
# initialization
pos[0].set_to(0)
registers[0,0].set_to(aux_ints[0])
registers[0,1].set_to(aux_ints[1])
registers[0,2].set_to(aux_ints[2])

if net_choice == 0:
  tmp_0[0].set_to(net_0(tiles[0]))
  registers[0, 3].set_to(embed_digit(tmp_0[0]))
elif net_choice == 1:
  tmp_1[0].set_to(net_1(tiles[0]))
  registers[0, 3].set_to(embed_op_code(tmp_1[0]))
for r in range(4, T+3):
  registers[0, r].set_to(0)

# execution model
for t in range(T - 1):
  if instr[t] == 0: # NOOP
    pos[t + 1].set_to(pos[t])
    registers[t + 1, t + 4].set_to(0)
  elif instr[t] == 1: # APPLY
    with arg3s[t] as a3:
      tmp_opcode[t].set_to(
       extract_op_code(registers[t, a3]))
    with arg1s[t] as a1:
      with arg2s[t] as a2:
        registers[t + 1, t + 4].set_to(
         Apply(registers[t, a1],
               registers[t, a2],
               tmp_opcode[t]))
    pos[t + 1].set_to(pos[t])
  elif instr[t] == 2: # MOVE-E
    pos[t + 1].set_to(move_east(pos[t]))
    with pos[t + 1] as p:
      if net_choice == 0:
        tmp_0[t+1].set_to(net_0(tiles[p]))
        registers[t + 1, t + 4].set_to(
         embed_digit(tmp_0[t+1]))
      elif net_choice == 1:
        tmp_1[t+1].set_to(net_1(tiles[p]))
        registers[t + 1, t + 4].set_to(
         embed_op_code(tmp_1[t+1]))
  #... similar cases for move_south/west/north

  for r in range(t + 4):
    registers[t + 1, r].set_to(registers[t, r])
  for r in range(t + 5,T+3):
    registers[t + 1, r].set_to(registers[t, r])

with return_reg as r:
  final_sum.set_to(registers[T-1, r])
```

B.3. MATH scenario

This model can learn a loopy program to evaluate a simple handwritten mathematical expression.

```
# constants
N_TILES = 4;     T = 6;     N_INSTR = 2
N_BLOCK = 3;     N_REG = N_BLOCK
MAX_INT = 19;    IMAGE_SIZE = 28

# variables
tiles = InputTensor(IMAGE_SIZE, IMAGE_SIZE)[N_TILES]
final_sum = Output(MAX_INT)
halt_at_end = Output(2)
instr = Param(N_INSTR)[N_BLOCK]
arg1s = Param(N_REG)[N_BLOCK]
arg2s = Param(N_REG)[N_BLOCK]
arg3s = Param(N_REG)[N_BLOCK]
net_choice = Param(2)[N_BLOCK]
goto_block = Param(N_BLOCK)[N_BLOCK]
return_reg = Param(N_REG)

block_ptr = Var(N_BLOCK)[T + 1]
registers = Var(MAX_INT)[T + 1, N_REG]
pos = Var(N_TILES)[T + 1]
ishalted = Var(2)[T + 1]

tmp_0 = Var(10)[T]
tmp_1 = Var(4)[T]
tmp_opcode = Var(4)[T]
```

```
# functions
@Learn([Vector(IMAGE_SIZE, IMAGE_SIZE)], 10,
 hid_sizes = [256,256])
def net_0(img): pass

@Learn([Vector(IMAGE_SIZE, IMAGE_SIZE)], 4,
 hid_sizes = [256,256])
def classify_operation(img): pass

@Runtime([MAX_INT, MAX_INT, 4], MAX_INT)
def Apply(a, b, op):
  if op == 0:   return (a + b) % MAX_INT
  elif op == 1: return (a - b) % MAX_INT
  elif op == 2: return (a * b) % MAX_INT
  elif op == 3: return (MAX_INT-1 if b == 0
                 else int(a / b) % MAX_INT)
  else: return a

@Runtime([N_TILES], N_TILES)
def move_east(pos): return (pos + 1) % N_TILES

@Runtime([N_BLOCK], N_BLOCK)
def inc_block_ptr(bp): return (bp + 1) % N_BLOCK

@Runtime([4], MAX_INT)
def embed_op_code(c): return c

@Runtime([10], MAX_INT)
def embed_digit(c): return c

@Runtime([N_TILES], 2)
def pos_eq_zero(pos): return 1 if pos == 0 else 0

@Runtime([MAX_INT], 4)
def extract_op_code(c): return c if c < 4 else 0

@Runtime([N_REG, N_REG], 2)
def register_equality(r1, r2):
  return 1 if r1 == r2 else 0

# initialization
pos[0].set_to(0)
block_ptr[0].set_to(0)
for r in range(N_REG):
  registers[0,r].set_to(0)
ishalted[0].set_to(0)

# execution model
for t in range(T):
  if ishalted[t] == 1:
    ishalted[t+1].set_to(ishalted[t])
    block_ptr[t + 1].set_to(block_ptr[t])
    pos[t + 1].set_to(pos[t])
    for r in range(N_REG):
      registers[t + 1, r].set_to(registers[t, r])
  elif ishalted[t] == 0:
    with block_ptr[t] as bp:
      if instr[bp] == 0: # APPLY
        with arg3s[bp] as a3:
           tmp_opcode[t].set_to(
            extract_op_code(registers[t, a3]))
        with arg1s[bp] as a1:
          with arg2s[bp] as a2:
            registers[t + 1, bp].set_to(
             Apply(registers[t, a1],
                   registers[t, a2],
                   tmp_opcode[t]))
        pos[t + 1].set_to(pos[t])
      elif instr[bp] == 1: # MOVE_E
        pos[t + 1].set_to(move_east(pos[t]))
        with pos[t + 1] as p:
          if net_choice[bp] == 0:
            tmp_0[t].set_to(net_0(tiles[p]))
            registers[t + 1, bp].set_to(
             embed_digit(tmp_0[t]))
          elif net_choice[bp] == 1:
            tmp_1[t].set_to(net_1(tiles[p]))
            registers[t + 1, bp].set_to(
             embed_op_code(tmp_1[t]))

      block_ptr[t + 1].set_to(goto_block[bp])
      ishalted[t+1].set_to(pos_eq_zero(pos[t + 1]))

      for r in range(bp):
        registers[t + 1, r].set_to(registers[t, r])
      for r in range(bp+1,N_REG):
        registers[t + 1, r].set_to(registers[t, r])

with return_reg as r:
    final_sum.set_to(registers[T, r])
halt_at_end.set_to(ishalted[T])
```