nature computational science

Article

https://doi.org/10.1038/s43588-024-00593-9

Automated discovery of algorithms from data

In the format provided by the authors and unedited

Table of Contents

Supplementary Section 1: Deep distilling algorithm description and pseudocode

The following two pages contain an overview of the deep distilling algorithm and its various parts. The pseudocode gives an outline of how an ENN is trained, condensed, and then written as code. Brief descriptions of these seven functions are below.

1. **DeepDistilling:** The overall workflow for the deep distilling algorithm
2. **TrainENN:** The method by which a basic ENN is trained, as previously described (*16*)
3. **LearnSubconcepts:** The two new semi-supervised methods for clustering training data into subconcepts
4. **CondenseENN:** The overall workflow to condense an ENN
5. **OrganizeNeurons:** The method by which the neurons in an ENN layer are organized into groups based upon similar connectivity in order to enable the creation of for-loops
6. **InterpretFunction:** The method by which a group's connectivity pattern is analyzed for various types of logical functions with or without condensed variables in order to provide interpretability to the weighted-sum representation of neuron data-processing
7. **WriteCode:** The method by which the condensed ENN (consisting of functions and neuron groups) is turned into computer code.

1. function DEEPDISTILLING(*samples*, *labels*)

```
enn ← TRAINENN(samples, labels)
code ← CONDENSEENN(enn, shape(samples[0]))
print code to an output file
```

2. function TRAINENN(*samples*, *labels*):

```
//Set up concepts
conceptLabels ← unique values in labels
concepts ← [ ]
for each label in conceptLabels
    concept = set with indices of labels that match label
    concepts.push(concept)

//Unsupervised learning of subconcepts
subconcepts ← LEARNSUBCONCEPTS(samples, concepts)

//Supervised learning of differentia neurons
diffLayer ← {weights: [ ], biases: [ ] }
for each sub1 in subconcepts
    for each sub2 in subconcepts such that sub1's concept !=
            sub2's concept
        svm ← trained linear SVM between samples[sub1]
            and samples[sub2]
        diffLayer.weights.concatenate(svm.weights)
        diffLayer.biases.concatenate(svm.biases)

//Supervised learning of subconcept neurons
diff ← sign(samples*diffLayer.weights + diffLayer.biases)
subLayer ← {weights: [ ], biases: [ ] }
for each sub in subconcepts
    subComplement ← all samples not in sub's concept
    svm ← trained linear SVM between diff[sub] and
            diff[subComplement]
    subLayer.weights.concatenate(svm.weights)
    subLayer.biases.concatenate(svm.biases)

//Supervised learning of concept neurons
subc ← sign(diff*subLayer.weights + subLayer.biases)
concLayer ← {weights: [ ], biases: [ ] }
for each conc in concepts
    concComplement ← all samples not in conc
    svm ← trained linear SVM between subc[conc] and
            subc[concComplement]
    concLayer.weights.concatenate(svm.weights)
    concLayer.biases.concatenate(svm.biases)

return [ diffLayer, subLayer, concLayer]
```

3. function LEARNSUBCONCEPTS(*samples*, *concepts*):

```
minNumSubconcepts ← length(concepts)+1

//Ensure familial resemblance of subconcepts
subconcepts ← concepts
if samples are binary then
    subconcepts ← [ ]
    for conc in concepts
        graph ← graph where nodes are samples in conc and
                edge (i,j) exists if samples[i] • samples[j] > 0
        subconcepts.extend(components in graph)

//OPTION 1: Hierarchical clustering
trees ← [ ]
for each subc in subconcepts
    tree ← hierarchical cluster ing of samples in subconc
    trees.push(tree)
cutHeight ← height such that cutting all trees results in at
            least minNumSubconcepts total clusters
while length(subconcepts) < length(samples)
    subconcepts ← result of cutting trees at cutHeight
    linearlySeparable ← TRUE
    for each sub1 in subconcepts
        for each sub2 in subconcepts
            svm ← trained linear SVM between
                samples[sub1] and samples[sub2]
            if svm.error > 0 then
                linearlySeparable ← FALSE
                break out of for-loops
    if linearlySeparable then
        break out of while-loop
    cutHeight ← increase to increment total number of
                subconcepts by 1

return clusters formed by cutting trees at cutHeight

//OPTION 2: Iteratively divide subconcepts
while length(subconcepts) < length(samples)
    maxError ← -1
    split ← ∅
    for each sub1 in subconcepts
        for each sub2 in subconcepts
            svm ← trained linear SVM between
                samples[sub1] and samples[sub2]
            if svm.error > maxError then
                maxError ← svm.error
                split ← sub1 if svm.err1>svm.err2 else sub2
    if length(subconcepts) >= minNumSubconcepts then
        if maxError == 0 then break out of for-loop
    newSub ← misclassified samples from split
    split ← split\newSub
    subconcepts.push(newSub)

return subconcepts
```

4. function CONDENSEENN(*enn,shape*)

```
code ← function header as a string
for each layer in enn
    groups ← ORGANIZENEURONS(layer, shape)
    for each group in groups
        function ← INTERPRETFUNCTION(group)
        code += WRITECODE(function)
code += appropriate return statement as a string
return code
```

5. function ORGANIZENEURONS(*layer,shape*)

```
//Step 1: Create formatted neurons
neurons ← [ ]
for each column in layer
    weights ← column
    maxWeight ← max(abs(weights))
    weights /= maxWeight
    weights ← scale weights such that all values are integers
    neurons.push(new Neuron(weights))

//Step 2: Find intra-neuron patterns
for each neuron in neurons
    uniqueWeights ← unique weights in neuron.weights
    neuron.patterns ← [ ]
    for each u in uniqueWeights
        inGroup, inIndices ← the group(s) and indices from
                input shape weighted by u
        patternType ← check through defined pattern types
                for one that matches inIndices
        neuron.patterns.push(new Pattern(u, inGroup, pattern-
                Type, inIndices)

//Step 3: Put matching neurons in groups
groups ← [ ]
for each neuron in neurons
    matches ← [ ]
    for each n in neurons
        isMatch ← TRUE
        for each pattern in neuron.patterns
            isMatch ← boolean: n.patterns has a pattern with
                the same u, inGroup, and patternType as pat-
                tern
            if NOT isMatch then break out of for-loop
        if isMatch then matches.push(n)
    groups.push(new Group(neuron.patterns, neuron.weights,
            matches))
    neurons.remove(matches)

return groups
```

6. function INTERPRETFUNCTION(*group*)

```
//Check for conjunction
xMax ← (1+sign(group.weights))/2
if xMax*group.weights + group.bias > 0 then
    if (xMax-1)*group.weights + group.bias <= 0 then
        return new Function("conjunction", group)

//Check for disjunction
xMin ← (1-sign(group.weights))/2
if xMin*group.weights + group.bias < 0 then
    if (xMin+1)*group.weights + group.bias >= 0 then
        return new Function("disjunction", group)

//Check for Boolean formula
if length(group.weights)<5 then
    //Function("Boolean") performs the Quine-
            McCluskey algorithm
    return new Function("Boolean", group)

//Check for nested logic
if length(group.condensedVars) == 2 then
    grid ← grid of all values that group.condensedVars can be
    gridOutput ← sign(grid*group.u + group.biases)
    if number of rows of gridOutput containing different
            values <= 3 then
        return new Function("nested by row", group)
    if number of columns of gridOutput containing differ-
            ent values <= 3 then
        return new Function("nested by col", group)

//If nothing else, just have function print
            u1c1 + u2c2 + ...
return new Function("weighted sum", group)
```

7. function WRITECODE(*function*)

```
code ← initialization of function's output, as a string
if function.forLoop != ∅ then
    code += for-loop line over relevant values
if function.condensedVars != ∅ then
    code += declaring & initializing condensed variables
code += function.toString()

return code
```

Supplementary Section 2: Code produced by deep distilling

On the following pages is the code as produced by deep distilling. For each problem, we have included the code twice. On the left side is the raw code as output by the ENN condenser. This code has certain values hard-coded into it. On the right is the generalized code found as described in the Methods that allows for inputs of arbitrary size. The code is written in Python. Above each we have endeavored to provide descriptions of what each variable is doing to provide an interpretation of what each variable is doing, particularly in relation to the initial model inputs.

In each case the variables that are automatically assigned are fairly nondescript. Variables that start with "D" correspond to differentia neurons in the ENN and are meant to distinguish specific subconcepts from one another. Variables that start with "S" correspond to subconcept neurons in the ENN and are meant to distinguish a specific subconcept from everything else. Variables that start with "C" correspond to the output concept neurons.

The only manual changes to the code are the addition of comment strings and the addition of some blank lines to help align the single-case and generalized code.

Supplementary Section 2a: Distilled code to update a Rule 30 cellular automaton

This algorithm implements the rule 30 cellular automaton exactly as one would expect, albeit with a bit of redundancy due to fitting its logic into the basic ENN framework. The code is below, and above it is a description of the 7 variables created as part of the distilled algorithm. In the description, the logic from the code is re-presented in terms of the original three central cells (denoted by LEFT, CENTER, and RIGHT) in order to see how the rule 30 logic comes about.

- D1 = (not LEFT) or (not (CENTER or RIGHT))
- D2 = LEFT or CENTER or RIGHT

- S1 = LEFT xor (CENTER or RIGHT) # RULE30
- S2 = LEFT and (CENTER or RIGHT)
- S3 = not (LEFT or CENTER or RIGHT)

- C1 = LEFT xnor (CENTER or RIGHT) # not (RULE30)
- C2 = LEFT xor (CENTER or RIGHT) # RULE30

- return → LEFT xor (CENTER or RIGHT) # RULE30

```
def rule30_3(I):
    #I is a 3-cell grid, with cell 1 being the cell to update

    D1 = (not I[0]) or ((not I[1]) and (not I[2]))

    D2 = I[2] or I[1] or I[0]

    S1 = (D1 and D2)

    S2 = (not D1)

    S3 = (not D2)

    C1 = (not S1) or (S2 and S3)

    C2 = (S1 and (not S3)) or (S1 and (not S2) and S3)

    return C2 and not C1
```

```
def rule30(I, n):
    #I is an n-cell grid, with cell (n-1)/2 being the cell
    to update

    D1 = (not I[(n-1)/2 - 1]) or ((not I[(n-1)/2]) and (not
    I[(n-1)/2 + 1]))

    D2 = I[(n-1)/2 + 1] or I[(n-1)/2] or I[(n-1)/2 - 1]

    S1 = (D1 and D2)

    S2 = (not D1)

    S3 = (not D2)

    C1 = (not S1) or (S2 and S3)

    C2 = (S1 and (not S3)) or (S1 and (not S2) and S3)

    return C2 and not C1
```

Supplementary Section 2b: Distilled code to update a Rule 110 cellular automaton

The results here are similar to the Rule 30 cellular automaton above. Notice how the distilled code for rule 110 is the exact same as for rule 30 after the first two differentia variables D1 and D2. Below is a similar description of each of the 7 variables found in the distilled code.

- D1 = CENTER or RIGHT
- D2 = not (LEFT and CENTER and RIGHT)

- S1 = (CENTER or RIGHT) and not (LEFT and CENTER and RIGHT) # RULE110
- S2 = not (CENTER or RIGHT)
- S3 = LEFT and CENTER and RIGHT

- C1 = not (CENTER or RIGHT) or (LEFT and CENTER and RIGHT) # not (RULE110)
- C2 = (CENTER or RIGHT) and not (LEFT and CENTER and RIGHT) # RULE110

- return → (CENTER or RIGHT) and not (LEFT and CENTER and RIGHT) # RULE110

```
def rule110_3(I):
   #I is a 3-cell grid, with cell 1 being the cell to update

   D1 = I[1] or I[2]

   D2 = (not I[0]) or (not I[1]) or (not I[2])

   S1 = (D1 and D2)

   S2 = (not D1)

   S3 = (not D2)

   C1 = (not S1) or (S2 and S3)

   C2 = (S1 and (not S3)) or (S1 and (not S2) and S3)

   return C2 and not C1
```

```
def rule110(I, n):
   #I is an n-cell grid, with cell (n-1)/2 being the cell
   to update

   D1 = I[(n-1)/2] or I[(n-1)/2 + 1]

   D2 = (not I[(n-1)/2 - 1]) or (not I[(n-1)/2]) or (not
   I[(n-1)/2 + 1])

   S1 = (D1 and D2)

   S2 = (not D1)

   S3 = (not D2)

   C1 = (not S1) or (S2 and S3)

   C2 = (S1 and (not S3)) or (S1 and (not S2) and S3)

   return C2 and not C1
```

Supplementary Section 2c: Distilled code to update any elementary cellular automaton

Deep distilling figured out how to basically create a lookup table for the automaton grid and then select the precise update based upon particular bits from the rule vector. As above, LEFT, CENTER, and RIGHT signify the three central cells of the grid, and variables are mostly described in relation to these initial inputs.

- D1-8: holds the bitwise negated form of the 8-bit rule vector R
- D9-10: hold CENTER and (not CENTER), respectively
- D11 = not (LEFT) and not (RIGHT)
- D12 = LEFT and not (RIGHT)
- D13 = not (LEFT) and RIGHT
- D14 = LEFT and RIGHT

- S1 = not (R0) and CENTER and (LEFT and RIGHT)
- S2 = R0 and CENTER and (LEFT or RIGHT)
- S3 = not (R1) and CENTER and (LEFT and not (RIGHT))
- S4 = R1 and CENTER and (LEFT or not (RIGHT))
- S5 = not (R2) and not (CENTER) and (LEFT and RIGHT)
- S6 = R2 and not (CENTER) and (LEFT or RIGHT)
- S7 = not (R3) and not (CENTER) and (LEFT and not (RIGHT))
- S8 = R3 and not (CENTER) and (LEFT or not (RIGHT))
- S9 = not (R4) and CENTER and (not (LEFT) and RIGHT)
- S10 = R4 and CENTER and (not (LEFT) or RIGHT)
- S11 = not (R5) and CENTER and (not (LEFT) and not (RIGHT))
- S12 = R5 and CENTER and (not (LEFT) or not (RIGHT))
- S13 = not (R6) and not (CENTER) and (not (LEFT) and RIGHT)
- S14 = R6 and not (CENTER) and (not (LEFT) or RIGHT)
- S15 = not (R7) and not (CENTER) and (not (LEFT) and not (RIGHT))
- S16 = R7 and not (CENTER) and (not (LEFT) or not (RIGHT))

- C1 = any(odd S variables)
- C2 = any(even S variables)

- return → for each unique possible state of the automaton grid, return a specific bit value from the rule vector

```python
def elementary_automata_3(I1, I2):
    #I1 is the 8-bit encoding of the rule number. I2 is a
    3-cell grid, with cell 1 being the cell to update

    D1 = (not I1[0])
    D2 = (not I1[1])
    D3 = (not I1[2])
    D4 = (not I1[3])
    D5 = (not I1[4])
    D6 = (not I1[5])
    D7 = (not I1[6])
    D8 = (not I1[7])

    D9 = (not I2[1])
    D10 = I2[1]

    D11 = 0.5
    if ((not I2[0]) and (not I2[2])):
        D11 = 1
    elif (not I2[2]) or (not I2[0]):
        D11 = 0

    D12 = 0.5
    if (I2[0] and (not I2[2])):
        D12 = 1
    elif (not I2[2]) or I2[0]:
        D12 = 0

    D13 = 0.5
    if (I2[2] and (not I2[0])):
        D13 = 1
    elif (not I2[0]) or I2[2]:
        D13 = 0

    D14 = 0.5
    if (I2[0] and I2[2]):
        D14 = 1
    elif I2[2] or I2[0]:
        D14 = 0

    S1 = (D14 and D1 and D10)
    S2 = ((not D1) and (not D9) and (not D11))
    S3 = (D12 and D2 and D10)
    S4 = ((not D2) and (not D9) and (not D13))
    S5 = (D14 and D3 and D9)
    S6 = ((not D3) and (not D10) and (not D11))
    S7 = (D12 and D4 and D9)
    S8 = ((not D4) and (not D10) and (not D13))
    S9 = (D13 and D5 and D10)
    S10 = ((not D5) and (not D9) and (not D12))
    S11 = (D11 and D6 and D10)
    S12 = ((not D6) and (not D9) and (not D14))
    S13 = (D13 and D7 and D9)
    S14 = ((not D7) and (not D10) and (not D12))
    S15 = (D11 and D8 and D9)
    S16 = ((not D8) and (not D10) and (not D14))

    C1 = S15 or S13 or S11 or S9 or S7 or S5 or S3 or S1
    C2 = S16 or S14 or S12 or S10 or S8 or S6 or S4 or S2

    return C2 and not C1
```

```python
def elementary_automata(I1, I2, n):
    #I1 is the 8-bit encoding of the rule number. I2 is an
    n-cell grid, with cell (n-1)/2 being the cell to update

    D1 = (not I1[0])
    D2 = (not I1[1])
    D3 = (not I1[2])
    D4 = (not I1[3])
    D5 = (not I1[4])
    D6 = (not I1[5])
    D7 = (not I1[6])
    D8 = (not I1[7])

    D9 = (not I2[(n-1)/2])
    D10 = I2[(n-1)/2]

    D11 = 0.5
    if ((not I2[(n-1)/2 - 1]) and (not I2[(n-1)/2 + 1])):
        D11 = 1
    elif (not I2[(n-1)/2 + 1]) nor (not I2[(n-1)/2 - 1]):
        D11 = 0

    D12 = 0.5
    if (I2[(n-1)/2 - 1] and (not I2[(n-1)/2 + 1])):
        D12 = 1
    elif (not I2[(n-1)/2 + 1]) nor I2[(n-1)/2 - 1]:
        D12 = 0

    D13 = 0.5
    if (I2[(n-1)/2 + 1] and (not I2[(n-1)/2 - 1])):
        D13 = 1
    elif (not I2[(n-1)/2 - 1]) nor I2[(n-1)/2 + 1]:
        D13 = 0

    D14 = 0.5
    if (I2[(n-1)/2 - 1] and I2[(n-1)/2 + 1]):
        D14 = 1
    elif I2[(n-1)/2 + 1] nor I2[(n-1)/2 - 1]:
        D14 = 0

    S1 = (D14 and D1 and D10)
    S2 = ((not D1) and (not D9) and (not D11))
    S3 = (D12 and D2 and D10)
    S4 = ((not D2) and (not D9) and (not D13))
    S5 = (D14 and D3 and D9)
    S6 = ((not D3) and (not D10) and (not D11))
    S7 = (D12 and D4 and D9)
    S8 = ((not D4) and (not D10) and (not D13))
    S9 = (D13 and D5 and D10)
    S10 = ((not D5) and (not D9) and (not D12))
    S11 = (D11 and D6 and D10)
    S12 = ((not D6) and (not D9) and (not D14))
    S13 = (D13 and D7 and D9)
    S14 = ((not D7) and (not D10) and (not D12))
    S15 = (D11 and D8 and D9)
    S16 = ((not D8) and (not D10) and (not D14))

    C1 = S15 or S13 or S11 or S9 or S7 or S5 or S3 or S1
    C2 = S16 or S14 or S12 or S10 or S8 or S6 or S4 or S2

    return C2 and not C1
```

Supplementary Section 2d: Distilled code to update a Game of Life cellular automaton

For the Game of Life, the distilled code essentially builds up the different cases leading to death and life as expected per the rules. The nested non-linearities in the rules require the ENN to build up these cases sequentially, even if there are a couple redundancies along the way. Below is a description of each of the variables condensed from the ENN, where CENTER indicates the central cell of the grid.

- $\mathrm{D1} = \mathrm{CENTER}$
- part_sum (aka NEIGHBORHOOD) = sum of the 8 cells surrounding the center
- $\mathrm{D2} = \mathrm{NEIGHBORHOOD} \leq 3$
- $\mathrm{D3} = \mathrm{NEIGHBORHOOD} > 1$
- $\mathrm{D4} = \mathrm{NEIGHBORHOOD} > 2$

- S1 = CENTER and (NEIGHBORHOOD=1 or NEIGHBORHOOD=2)
- $\mathrm{S2} = \mathrm{NEIGHBORHOOD} = 3$
- $\mathrm{S3} = \mathrm{NEIGHBORHOOD} > 3$
- S4 = (not CENTER) and ($\mathrm{NEIGHBORHOOD} \leq 2$)
- $\mathrm{S5} = \mathrm{NEIGHBORHOOD} \leq 1$

- C1 = ($\mathrm{NEIGHBORHOOD} \leq 1$) or ($\mathrm{NEIGHBORHOOD} > 3$) or
  ((not CENTER) and NEIGHBORHOOD=2)
- C2 = (NEIGHBORHOOD=3) or (CENTER and (NEIGHBORHOOD=1 or NEIGHBORHOOD=2))

- return (NEIGHBORHOOD=3) or (CENTER and (NEIGHBORHOOD=1 or NEIGHBORHOOD=2))

```
def game_of_life_3(I):
   #I is a 3x3 grid, with the center cell being the
   cell to update

   D1 = I[1, 1]

   D2 = 0
   part_sum = (I[0,0] + I[0,1] + I[0,2] + I[1,0] +
    I[1,2] + I[2,0] + I[2,1] + I[2,2])



   if part_sum <= 3:
      D2 = 1
   elif part_sum > 3:
      D2 = -1

   D3 = 0
   part_sum = (I[0,0] + I[0,1] + I[0,2] + I[1,0] +
    I[1,2] + I[2,0] + I[2,1] + I[2,2])



   if part_sum > 1:
      D3 = 1
   elif part_sum <= 1:
      D3 = -1

   D4 = 0
   part_sum = (I[0,0] + I[0,1] + I[0,2] + I[1,0] +
    I[1,2] + I[2,0] + I[2,1] + I[2,2])



   if part_sum > 2:
      D3 = 1
   elif part_sum <= 2:
      D3 = -1


   S1 = (D1>0 and D2>0 and D3>0)

   S2 = (D2>0 and D4>0)

   S3 = (not D2>0)

   S4 = ((not D1>0) and (not D4>0))

   S5 = (not D3>0)

   C1 = (S3 or S4 or S5)

   C2 = (S1 or S2)

   return C2 and not C1
```

```
def game_of_life(I, n):
   #I is an nxn grid, with the center cell being
   the cell to update

   D1 = I[(n-1)/2, (n-1)/2]

   D2 = 0
   part_sum = (I[(n-1)/2-1, (n-1)/2-1] + I[(n-1)/2-
    1, (n-1)/2] + I[(n-1)/2-1, (n-1)/2+1] + I[(n-
    1)/2, (n-1)/2-1] + I[(n-1)/2, (n-1)/2+1] +
    I[(n-1)/2+1, (n-1)/2-1] + I[(n-1)/2+1, (n-1)/2]
    + I[(n-1)/2+1, (n-1)/2+1])
   if part_sum > 3:
      D2 = 1
   elif part_sum <= 3:
      D2 = -1

   D3 = 0
   part_sum = (I[(n-1)/2-1, (n-1)/2-1] + I[(n-1)/2-
    1, (n-1)/2] + I[(n-1)/2-1, (n-1)/2+1] + I[(n-
    1)/2, (n-1)/2-1] + I[(n-1)/2, (n-1)/2+1] + I[2,
    (n-1)/2-1] + I[(n-1)/2+1, (n-1)/2] + I[(n-
    1)/2+1, (n-1)/2+1])
   if part_sum > 1:
      D3 = 1
   elif part_sum <= 1:
      D3 = -1

   D4 = 0
   part_sum = (I[(n-1)/2-1, (n-1)/2-1] + I[(n-1)/2-
    1, (n-1)/2] + I[(n-1)/2-1, (n-1)/2+1] + I[(n-
    1)/2, (n-1)/2-1] + I[(n-1)/2, (n-1)/2+1] + I[2,
    (n-1)/2-1] + I[(n-1)/2+1, (n-1)/2] + I[(n-
    1)/2+1, (n-1)/2+1])
   if part_sum > 2:
      D3 = 1
   elif part_sum <= 2:
      D3 = -1


   S1 = (D1>0 and D2>0 and D3>0)

   S2 = (D2>0 and D4>0)

   S3 = (not D2>0)

   S4 = ((not D1>0) and (not D4>0))

   S5 = (not D3>0)

   C1 = (S3 and S5) or (S3 and S4 and (not S5))

   C2 = (S1 and S2)

   return C2 and not C1
```

Supplementary Section 2e: Distilled code to find the maximum absolute value

Because basic ENNs do not have any recurrent connections, it is not possible for them to iterate over the array of numbers and store the running maximum magnitude. Instead, it compares each number with all other numbers and with the negative of those numbers as well. In order for a number to have the maximum magnitude, it has to either win all of these comparisons or lose all of them. The distilled code returns whichever index won all of these comparisons. A description of the variables created in the distilled code is below.

- D1 = 2D array containing all comparisons of $x_i > x_j$
- D2 = 2D array containing all comparisons of $x_i > -x_j$

- S1 = 1D array containing whether an x_i won all comparisons in D1 and D2
  i.e. S1[i] = all(D1[i,:]) and all(D2[i,:])
- S2 = 1D array containing whether an x_i won no comparisons in D1 and D2
  i.e. S2[i] = not (any(D1[i,:]) or any(D2[i,:]))
    - row_sum_1 and row_sum_2 = the sum of all values in either D1 or D2, respectively

- C = 1D array containing whether an x_i was the winner in either S1 or S2
  i.e. C[i] = S1[i] or S2[i]
- return → the index of C that won all comparisons

```python
import numpy as np
import random

def absmax_20(I):
    #I is an array of 20 numbers

    D1 = np.zeros((20, 20))
    for i in range(20):
        for j in range(20):
            if i == j:
                continue
            value_1 = I[i]
            value_2 = I[j]
            if value_1 > value_2:
                D1[i,j] = 1
            elif value_1 < value_2:
                D1[i,j] = -1

    D2 = np.zeros((20, 20))
    for i in range(20):
        for j in range(20):
            if i == j:
                continue
            value_1 = I[i]
            value_2 = I[j]
            if value_1 > -value_2:
                D2[i,j] = 1
            elif value_1 < -value_2:
                D2[i,j] = -1

    S1 = np.zeros(20)
    for i in range(20):
        row_sum_1 = np.sum(D1[i, :])
        row_sum_2 = np.sum(D2[i, :])
        if row_sum_1 < 18:
            S1[i] = -1
        elif row_sum_2 < 18:
            S1[i] = -1
        elif row_sum_1 + row_sum_2 > -37:
            S1[i] = 1
        else:
            S1[i] = -1

    S2 = np.zeros(20)
    for i in range(20):
        row_sum_1 = np.sum(D1[i, :])
        row_sum_2 = np.sum(D2[i, :])
        if row_sum_1 > -18:
            S2[i] = -1
        elif row_sum_2 > -18:
            S2[i] = -1
        elif -row_sum_1 - row_sum_2 > -37:
            S2[i] = 1
        else:
            S2[i] = -1

    C = np.zeros(20)
    for i in range(20):
        C[i] = 20*S2[i] + 20*S1[i] - np.sum(S2) - np.sum(S1)

    results = np.where(C==max(C))[0]
    return random.choice(results)
```

```python
import numpy as np
import random

def absmax(I, n):
    #I is an array of n numbers

    D1 = np.zeros((n, n))
    for i in range(n):
        for j in range(n):
            if i == j:
                continue
            value_1 = I[i]
            value_2 = I[j]
            if value_1 > value_2:
                D1[i,j] = 1
            elif value_1 < value_2:
                D1[i,j] = -1

    D2 = np.zeros((n, n))
    for i in range(n):
        for j in range(n):
            if i == j:
                continue
            value_1 = I[i]
            value_2 = I[j]
            if value_1 > -value_2:
                D2[i,j] = 1
            elif value_1 < -value_2:
                D2[i,j] = -1

    S1 = np.zeros(n)
    for i in range(n):
        row_sum_1 = np.sum(D1[i, :])
        row_sum_2 = np.sum(D2[i, :])
        if row_sum_1 < n-2:
            S1[i] = -1
        elif row_sum_2 < n-2:
            S1[i] = -1
        elif row_sum_1 + row_sum_2 > 3-2*n:
            S1[i] = 1
        else:
            S1[i] = -1

    S2 = np.zeros(n)
    for i in range(n):
        row_sum_1 = np.sum(D1[i, :])
        row_sum_2 = np.sum(D2[i, :])
        if row_sum_1 > 2-n:
            S2[i] = -1
        elif row_sum_2 > 2-n:
            S2[i] = -1
        elif -row_sum_1 - row_sum_2 > 3-2*n:
            S2[i] = 1
        else:
            S2[i] = -1

    C = np.zeros(n)
    for i in range(n):
        C[i] = n*S2[i] + n*S1[i] - np.sum(S2) - np.sum(S1)

    results = np.where(C==max(C))[0]
    return random.choice(results)
```

Supplementary Section 2f: Distilled code to find the best assignment for MAX-SAT

The distilled code for this problem goes through each clause of the Boolean formula individually to determine whether there are any other variables present in the formula besides the first. Then it determines for either case what the difference is in the number of clauses that have the first variable present as a positive—rather than a negative—literal. It weights the two cases differently (by 10 and 2.298, respectively) and returns the sigmoid output of this.

- D1 = 1D array indicating for each clause if any of the other variables are present
- D2 = 1D array indicating for each clause if all of the other variables are absent
- D3 = 1D array indicating for each clause the negation of the first variable, and if it is absent then indicating if any other variables are present
    - col_mean = half the percentage of other literals present in the clause
- D4 = 1D array indicating for each clause the value of the first variable, and if it is absent then indicating if any other variables are present
    - col_mean = half the percentage of other literals present in the clause
- D5 = 1D array indicating for each clause if the first variable is present and POSITIVE
- D6 = 1D array indicating for each clause if the first variable is present and NEGATIVE

- S1 = 1D array indicating for each clause if the first variable is NEGATIVE and there are other variables present (aka NEG-OTHERS)
- S2 = 1D array indicating for each clause if the first variable is NEGATIVE and there are no other variables present (aka NEG-ALONE)
- S3 = 1D array indicating for each clause if the first variable is POSITIVE and there are other variables present (aka POS-OTHERS)
- S4 = 1D array indicating for each clause if the first variable is POSITIVE and there are no other variables present (aka POS-ALONE)

- C1 = 10 * Σ (POS-OTHERS – NEG-OTHERS) + 2.298 * Σ (POS-ALONE – NEG-ALONE)
- C2 = –C1

- return → sigmoid(2*C1)

```python
import numpy as np

def maxsat_10_50(I):
    #I is an input of size 10x50 (5 one-hot-encoded Boolean
    variables, 50 clauses)

    D1 = np.zeros(50)
    for i in range(50):
        if np.any(I[2:, i]!=0):
            D1[i] = -1
        else:
            D1[i] = 1

    D2 = np.zeros(50)
    for i in range(50):
        if np.any(I[2:, i]!=0):
            D2[i] = 1
        else:
            D2[i] = -1

    D3 = np.zeros(50)
    for i in range(50):
        col_mean = np.mean(I[2:, i])
        if I[1, i] + col_mean - I[0, i] > 0:
            D3[i] = 1
        else:
            D3[i] = -1

    D4 = np.zeros(50)
    for i in range(50):
        col_mean = np.mean(I[2:, i])
        if I[0, i] + col_mean - I[1, i] > 0:
            D4[i] = 1
        else:
            D4[i] = -1

    D5 = np.zeros(50)
    for i in range(50):
        if (I[0, i] and (not I[1, 0])):
            D5[i] = 1
        elif (not I[1, 0]) or I[0, i]:
            D5[i] = -1

    D6 = np.zeros(50)
    for i in range(50):
        if (I[1, i] and (not I[0, 0])):
            D6[i] = 1
        elif (not I[0, 0]) or I[1, i]:
            D6[i] = -1

    S1 = np.zeros(50)
    for i in range(50):
        S1[i] = (D6[i]>0 and D1[i]>0)

    S2 = np.zeros(50)
    for i in range(50):
        S2[i] = (D2[i]>0 and D3[i]>0)

    S3 = np.zeros(50)
    for i in range(50):
        S3[i] = (D5[i]>0 and D1[i]>0)

    S4 = np.zeros(50)
    for i in range(50):
        S4[i] = (D2[i]>0 and D4[i]>0)

    C1 = 10.0*np.sum(S3) + 2.298*np.sum(S4) -
     2.298*np.sum(S2) - 10.0*np.sum(S1)
    C2 = 10.0*np.sum(S1) + 2.298*np.sum(S2) -
     2.298*np.sum(S4) - 10.0*np.sum(S3)
    C = [C1, C2]

    return np.exp(C)/np.sum(np.exp(C))
```

```python
import numpy as np

def maxsat(I, n, m):
    #I is an input of size mxn (m one-hot-encoded Boolean
    variables, n clauses)

    D1 = np.zeros(n)
    for i in range(n):
        if np.any(I[2:, i]!=0):
            D1[i] = -1
        else:
            D1[i] = 1

    D2 = np.zeros(n)
    for i in range(n):
        if np.any(I[2:, i]!=0):
            D2[i] = 1
        else:
            D2[i] = -1

    D3 = np.zeros(n)
    for i in range(n):
        col_mean = np.mean(I[2:, i])
        if I[1, i] + col_mean - I[0, i] > 0:
            D3[i] = 1
        else:
            D3[i] = -1

    D4 = np.zeros(n)
    for i in range(n):
        col_mean = np.mean(I[2:, i])
        if I[0, i] + col_mean - I[1, i] > 0:
            D4[i] = 1
        else:
            D4[i] = -1

    D5 = np.zeros(n)
    for i in range(n):
        if (I[0, i] and (not I[1, 0])):
            D5[i] = 1
        elif (not I[1, 0]) or I[0, i]:
            D5[i] = -1

    D6 = np.zeros(n)
    for i in range(n):
        if (I[1, i] and (not I[0, 0])):
            D6[i] = 1
        elif (not I[0, 0]) or I[1, i]:
            D6[i] = -1

    S1 = np.zeros(n)
    for i in range(n):
        S1[i] = (D6[i]>0 and D1[i]>0)

    S2 = np.zeros(n)
    for i in range(n):
        S2[i] = (D2[i]>0 and D3[i]>0)

    S3 = np.zeros(n)
    for i in range(n):
        S3[i] = (D5[i]>0 and D1[i]>0)

    S4 = np.zeros(n)
    for i in range(n):
        S4[i] = (D2[i]>0 and D4[i]>0)

    C1 = 10.0*np.sum(S3) + 2.298*np.sum(S4) -
     2.298*np.sum(S2) - 10.0*np.sum(S1)
    C2 = 10.0*np.sum(S1) + 2.298*np.sum(S2) -
     2.298*np.sum(S4) - 10.0*np.sum(S3)
    C = [C1, C2]

    return np.exp(C)/np.sum(np.exp(C))
```

Supplementary Section 2g: Distilled code to find a shape's orientation

The distilled code learns a nonintuitive algorithm. In most cases it essentially determines whether overall there are more columns that have greater total brightness than rows (i.e. a vertical orientation). However, there are interesting edge cases handled where the margin of difference in the column-row comparisons outside of the column (or row) in question is very close, in which case it has a series of tiebreakers that account for the given column (or row). A description of the variables appearing in the distilled code is below.

- D = a 2D matrix the same size as the image containing, pixel-by-pixel, whether the sum total brightness of each column is greater than that of each row
- col_sum = the sum total brightness of a column
- row_sum = the sum total brightness of a row

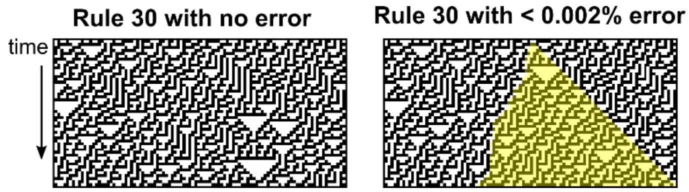
- S1 = 1D array; if the total margin of victory for columns over rows is great enough, all values will be TRUE; if the total margin of victory is very close, there are a couple of tiebreakers (for example, whether a column won any comparisons at all)
  - row_sum = the margin of victory for the pixel-by-pixel comparisons won by a given column in the image
  - offrow_sum = the margin of victory for the pixel-by-pixel comparisons won by all other columns in the image
- S2 = same S1 above but flipped for rows and columns
  - col_sum = the same as row_sum above, but for rows in the image
  - offcol_sum = the same as offrow_sum above, but for rows in the image

- C1 = whether columns won more than rows did
- C2 = whether rows won more than columns did

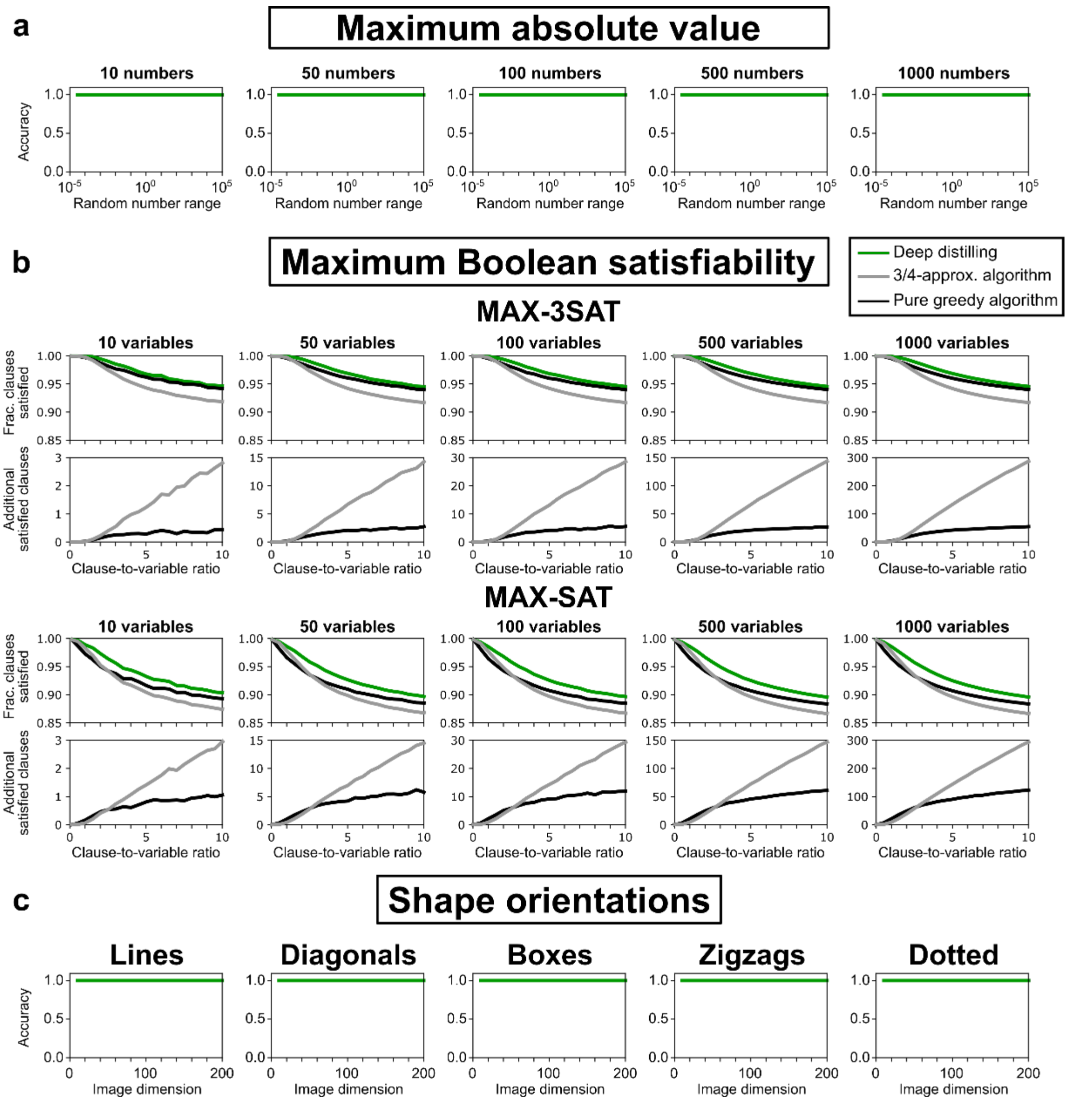
- return → VERTICAL if columns won more than rows, otherwise HORIZONTAL

```
import numpy as np
import random

def orientation_28(I, n):
#I is an input image that is 28x28
#I calculate pixel score for each pixel depending on if its
row or column is brighter
    D = np.zeros((28, 28))
    for i in range(28):
        for j in range(28):
            col_sum = np.sum(I[:, i])
            row_sum = np.sum(I[j, :])
            if col_sum > row_sum:
                D[i,j] = 1
            elif col_sum < row_sum:
                D[i,j] = -1

#for each row, calculate sum of pixel scores outside of the
row and compare with the image width to determine if that
row is significant. Use the sum of pixel scores in the row
to break ties
    S1 = np.zeros(28)
    for i in range(28):
        row_sum = np.sum(D[i, :])
        offrow_sum = (np.sum(D) - np.sum(D[i, :]))
        if offrow_sum < -29:
            S1[i] = 1
        elif offrow_sum > -27:
            S1[i] = -1
        elif offrow_sum == -27:
            if np.all(D[i, :]==1):
                S1[i] = 1
            elif not np.all(D[i, :]==1):
                S1[i] = -1
        elif offrow_sum == -28:
            if row_sum > 0:
                S1[i] = 1
            elif row_sum < 0:
                S1[i] = -1
        elif offrow_sum == -29:
            if not np.all(D[i, :]==-1):
                S1[i] = 1
            elif np.all(D[i, :]==-1):
                S1[i] = -1

#do the same for each column
    S2 = np.zeros(28)
    for i in range(28):
        offcol_sum = (np.sum(D) - np.sum(D[:, i]))
        col_sum = np.sum(D[:, i])
        if offcol_sum < 27:
            S2[i] = -1
        elif offcol_sum > 29:
            S2[i] = 1
        elif offcol_sum == 29:
            if np.all(D[:, i]==1):
                S2[i] = -1
            elif not np.all(D[:, i]==1):
                S2[i] = 1
        elif offcol_sum == 28:
            if col_sum > 0:
                S2[i] = -1
            elif col_sum < 0:
                S2[i] = 1
        elif offcol_sum == 27:
            if not np.all(D[:, i]==-1):
                S2[i] = -1
            elif np.all(D[:, i]==-1):
                S2[i] = 1

    C1 = np.sum(S1) - np.sum(S2)
    C2 = np.sum(S2) - np.sum(S1)
    C = [C1, C2]
#compare the number of significant rows versus columns
    results = np.where(C==max(C))[0]
    return random.choice(results)
```

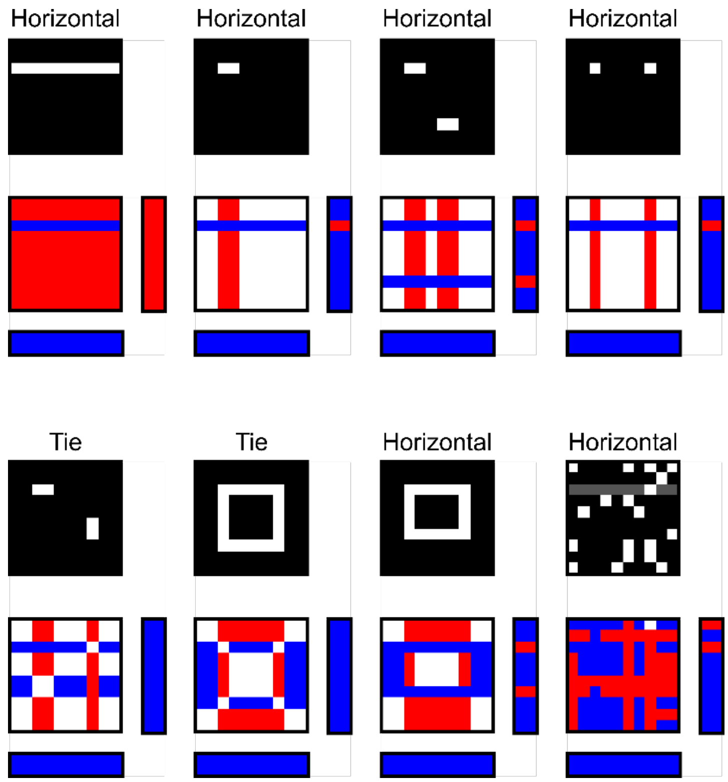
```
import numpy as np
import random

def orientation(I, n):
#I is an input image that is nxn
#I calculate score for each pixel depending on if its row
or column is brighter
    D = np.zeros((n, n))
    for i in range(n):
        for j in range(n):
            col_sum = np.sum(I[:, i])
            row_sum = np.sum(I[j, :])
            if col_sum > row_sum:
                D[i,j] = 1
            elif col_sum < row_sum:
                D[i,j] = -1

#for each row, calculate sum of pixel scores outside of the
row and compare with the image width to determine if that
row is significant. Use the sum of pixel scores in the row
to break ties
    S1 = np.zeros(n)
    for i in range(n):
        row_sum = np.sum(D[i, :])
        offrow_sum = (np.sum(D) - np.sum(D[i, :]))
        if offrow_sum < -1-n:
            S1[i] = 1
        elif offrow_sum > 1-n:
            S1[i] = -1
        elif offrow_sum == 1-n:
            if np.all(D[i, :]==1):
                S1[i] = 1
            elif not np.all(D[i, :]==1):
                S1[i] = -1
        elif offrow_sum == -n:
            if row_sum > 0:
                S1[i] = 1
            elif row_sum < 0:
                S1[i] = -1
        elif offrow_sum == -1-n:
            if not np.all(D[i, :]==-1):
                S1[i] = 1
            elif np.all(D[i, :]==-1):
                S1[i] = -1

    #do the same for each column
    S2 = np.zeros(n)
    for i in range(n):
        offcol_sum = (np.sum(D) - np.sum(D[:, i]))
        col_sum = np.sum(D[:, i])
        if offcol_sum < n-1:
            S2[i] = -1
        elif offcol_sum > n+1:
            S2[i] = 1
        elif offcol_sum == n+1:
            if np.all(D[:, i]==1):
                S2[i] = -1
            elif not np.all(D[:, i]==1):
                S2[i] = 1
        elif offcol_sum == n:
            if col_sum > 0:
                S2[i] = -1
            elif col_sum < 0:
                S2[i] = 1
        elif offcol_sum == n-1:
            if not np.all(D[:, i]==-1):
                S2[i] = -1
            elif np.all(D[:, i]==-1):
                S2[i] = 1

    C1 = np.sum(S1) - np.sum(S2)
    C2 = np.sum(S2) - np.sum(S1)
    C = [C1, C2]
#compare the number of significant rows versus columns
    results = np.where(C==max(C))[0]
    return random.choice(results)
```



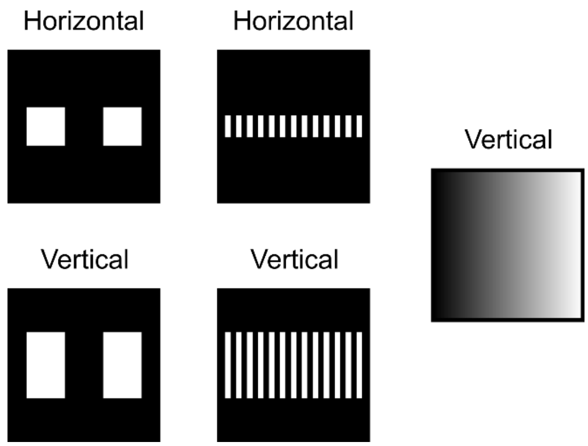
Supplementary Fig. 1. Deep learning lacks performance guarantees. Even though this deep learning model had almost perfect performance (i.e. 0.002% error on test images), the occurrence of a rare error is able to propagate and grow over time. The image on the right is a simple example of what this can look like, when a single error can grow and produce different behavior than it should (highlighted in yellow). This demonstrates the importance of having performance guarantees.



Supplementary Fig. 2. Deep distilled code generalizes across input sizes and complexities. The distilled algorithms were able to generalize to arbitrary input sizes for the (a) maximum absolute value, (b) MAX-SAT, and (c) shape orientation problems. **a,** Training occurred on input sizes of 18, 19, and 20 numbers, all in the set $\{-1,0,1\}$, but perfect accuracy was measured with the distilled code for sizes 10-1000 and with values in the range $[-10^{-5}, 10^{-5}]$ through $[-10^{5}, 10^{5}]$. **b,** Training data for MAX-SAT used only 8, 9, and 10 variables and 98, 99, and 100 clauses. The distilled code was able to perform well on Boolean formulae of much larger sizes, even to 1000 variables and 10,000 clauses, for both MAX-3SAT and MAX-SAT. For each, the upper plots show the percentage of clauses that were satisfied as a function of the number of clauses by the distilled code, by the pure greedy algorithm, and by the 3/4-approximation algorithm. The lower plots show the absolute difference in clauses satisfied by the two human-designed algorithm compared to the distilled code (a positive difference indicates the distilled code satisfied more clauses). **c,** Training data for shape orientations included 26x26, 27x27, and 28x28 pixel images of black images with a single white row or white column. Perfect accuracy was found on test sets of images sizes from 10x10 through 200x200, and with shapes that included variable-length lines, diagonal lines, boxes, zigzags, and dotted lines.



Supplementary Fig. 3. Examples of orientation image processing. Eight different example images are shown here along with how the distilled orientation algorithm processes them. The square matrix under each image shows the pixelwise row-versus-column orientation scores, with positive results (i.e., column brighter than row) in red, negative results (i.e., row brighter than column) in blue, and tied results in white. The results of the line scores compared to the overall image are shown to the right and below this matrix, with red bands indicating where there is a significant row or column. The final output label is denoted above each image.



Supplementary Fig. 4. Deep distilling assigns meaning in ambiguous cases. Each of these images are ambiguous in terms of how they could be classified (i.e., horizontal or vertical orientation). The labels provided are what the distilled code returned for each. This illustrates how a distilled algorithm is able to provide a consistent and unambiguous standard to provide meaning in ambiguous cases.