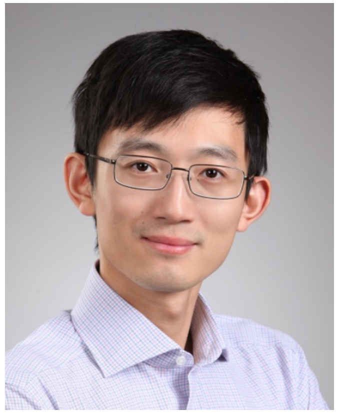
Composed Fine-Tuning: Freezing Pre-trained Denoising Autoencoders for Improved Generalization

Sang Michael Xie, Tengyu Ma, Percy Liang

ICML 2021

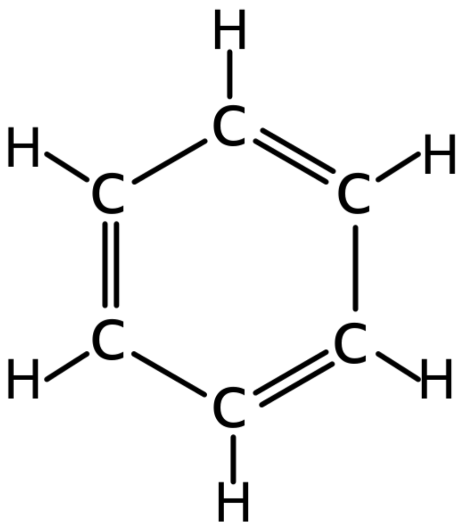

High-dimensional output spaces

```python
def partition(array, start, end):
    pivot = array[start]
    low = start + 1
    high = end
    while True:
        while low <= high and array[high] >= pivot:
            high = high - 1
        while low <= high and array[low] <= pivot:
            low = low + 1
        if low <= high:
            array[low], array[high] = array[high], array[low]
        else:
            break
    array[start], array[high] = array[high], array[start]
    return high

def quick_sort(array, start, end):
    if start >= end:
        return

    p = partition(array, start, end)
    quick_sort(array, start, p-1)
    quick_sort(array, p+1, end)
```

Code



Molecules

Language

Natural Images

High-dimensional output spaces

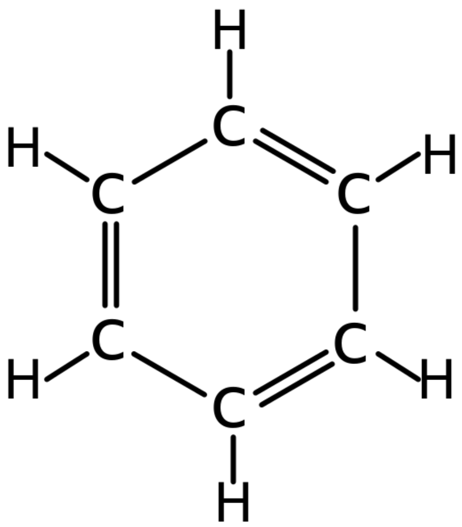
```
def partition(array, start, end):
    pivot = array[start]
    low = start + 1
    high = end
    while True:
        while low <= high and array[high] >= pivot:
            high = high - 1
        while low <= high and array[low] <= pivot:
            low = low + 1
        if low <= high:
            array[low], array[high] = array[high], array[low]
        else:
            break
    array[start], array[high] = array[high], array[start]
    return high

def quick_sort(array, start, end):
    if start >= end:
        return

    p = partition(array, start, end)
    quick_sort(array, start, p-1)
    quick_sort(array, p+1, end)
```

Code

Compiles/Executes



Molecules

Forms a stable molecule

Language

Grammar/syntax

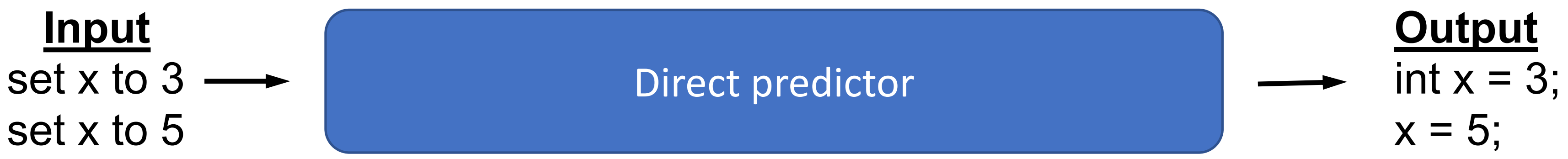
Natural Images

Physical constraints, familiar objects, sharp lines

Output structure: only some outputs are valid

Supervised learning

- Example: Pseudocode-to-code (Kulal et al. 2019)



Supervised learning

- Example: Pseudocode-to-code (Kulal et al. 2019)

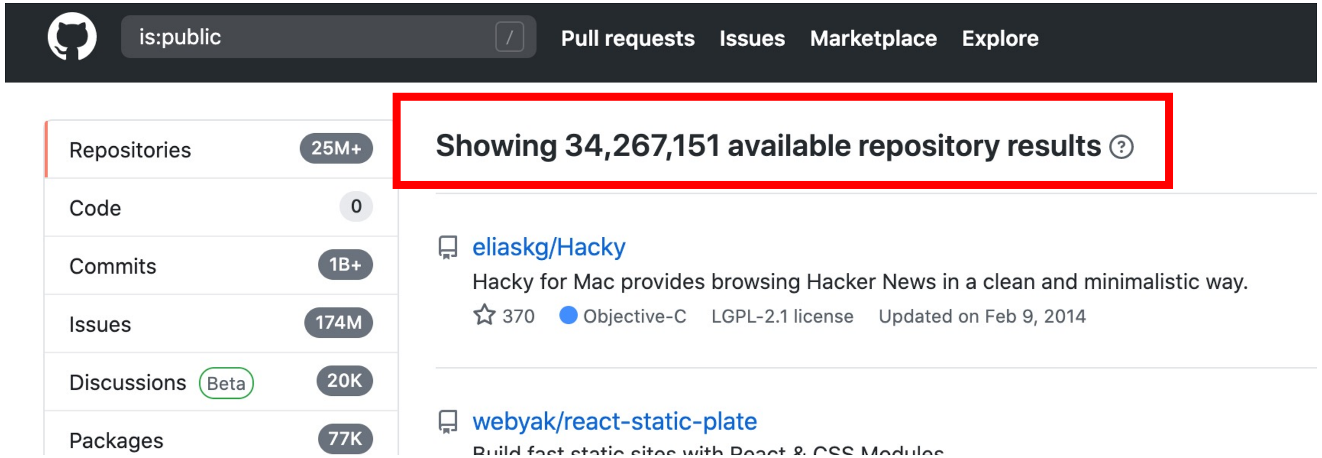
Input
set x to 3
set x to 5

Direct predictor

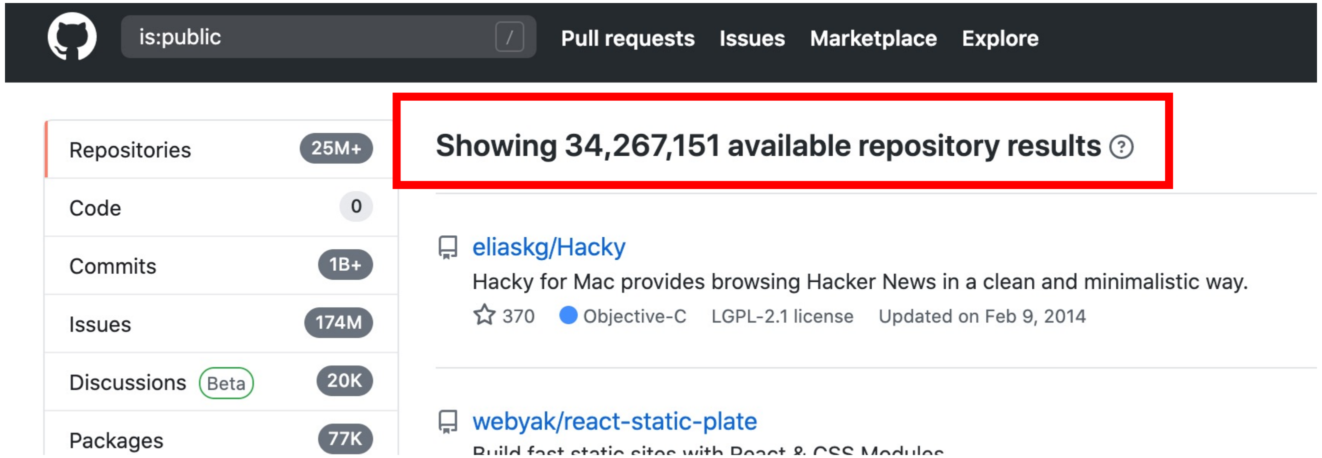
Output
int x = 3;
x = 5;

Predictor handles both **input-output mapping** and **output structure**

”Unlabeled” output data is abundant



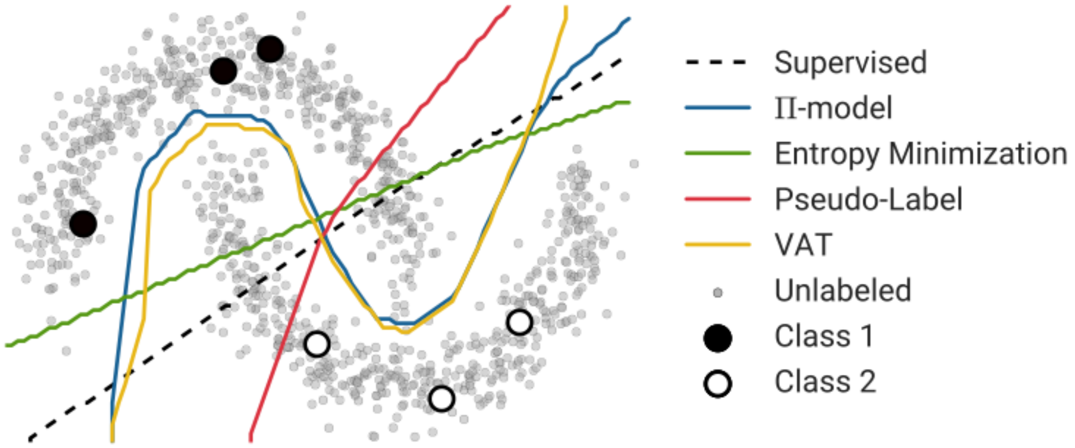
”Unlabeled” output data is abundant



Unlabeled output data can be used for learning the output structure

Unlabeled outputs: not standard SSL

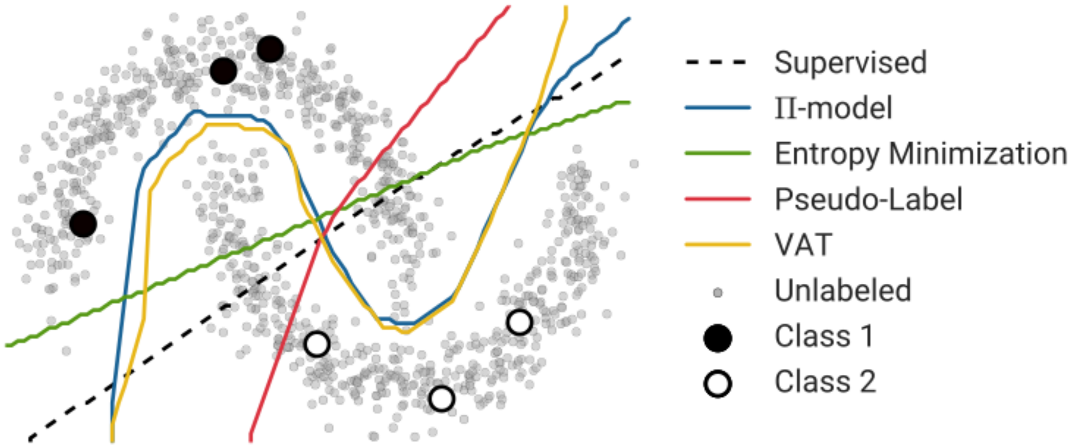
- Standard semi-supervised learning: unlabeled inputs for improving classifier



Oliver et al. 2018

Unlabeled outputs: not standard SSL

- Standard semi-supervised learning: unlabeled inputs for improving classifier

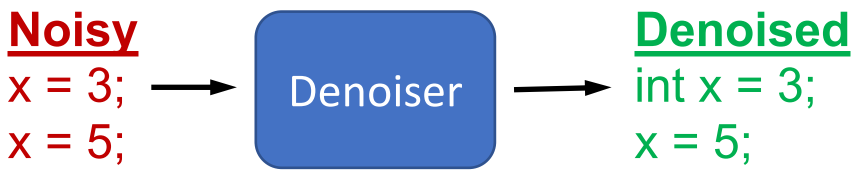


Oliver et al. 2018

- Leveraging unlabeled outputs requires a different way of thinking

Pre-train + Fine-tune paradigm

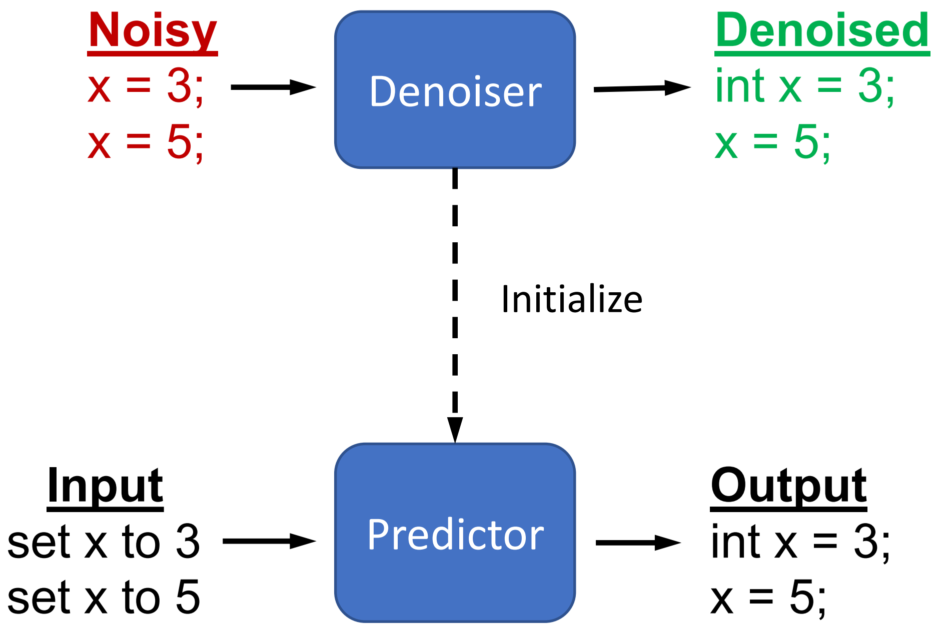
- Output structure: Pre-train a denoising autoencoder (denoiser) on large unlabeled data
  - (BART/T5 Lewis et al. 2020, Raffel et al. 2019)



Pre-train + Fine-tune paradigm

- Output structure: Pre-train a denoising autoencoder (denoiser) on large unlabeled data
  - (BART/T5 Lewis et al. 2020, Raffel et al. 2019)

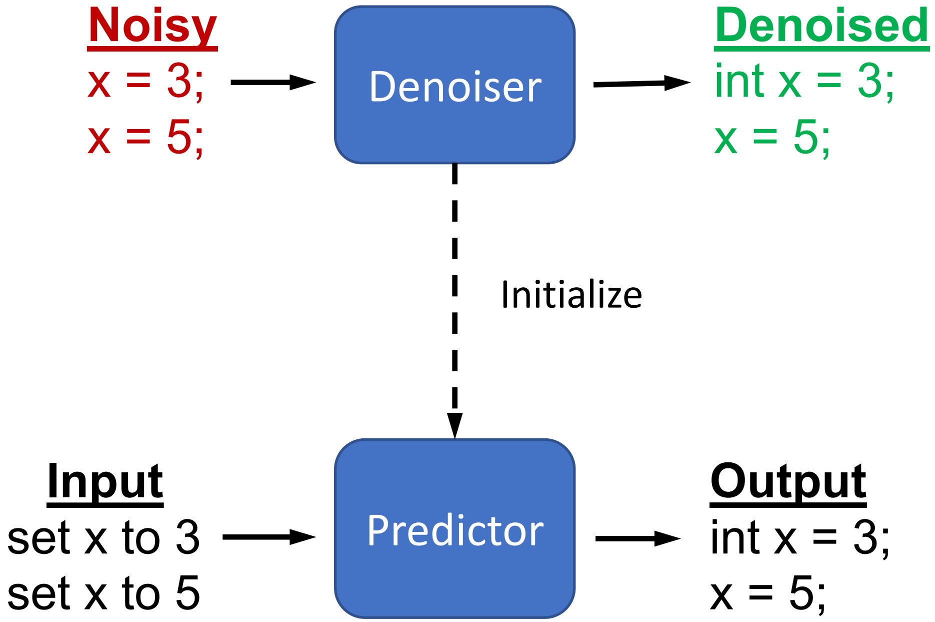
- Input-output mapping: Fine-tune on labeled data



Pre-train + Fine-tune paradigm

- Output structure: Pre-train a denoising autoencoder (denoiser) on large unlabeled data
  - (BART/T5 Lewis et al. 2020, Raffel et al. 2019)

- Input-output mapping: Fine-tune on labeled data



Highly accessible: don’t need unlabeled data during fine-tuning

How well does standard fine-tuning use pre-trained information?

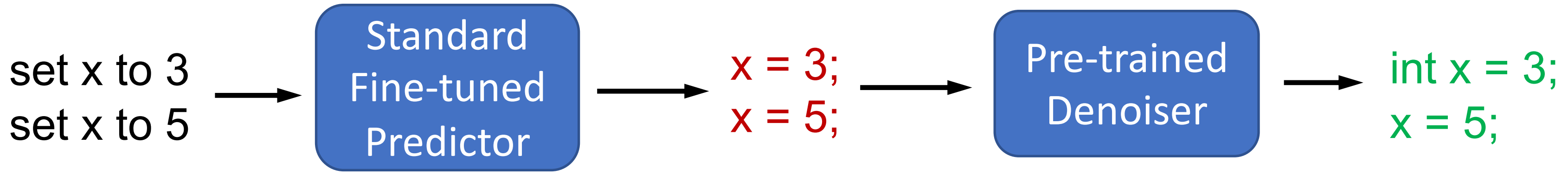
Experiment:

1. Train standard fine-tuned model initialized from pre-trained denoiser

How well does standard fine-tuning use pre-trained information?

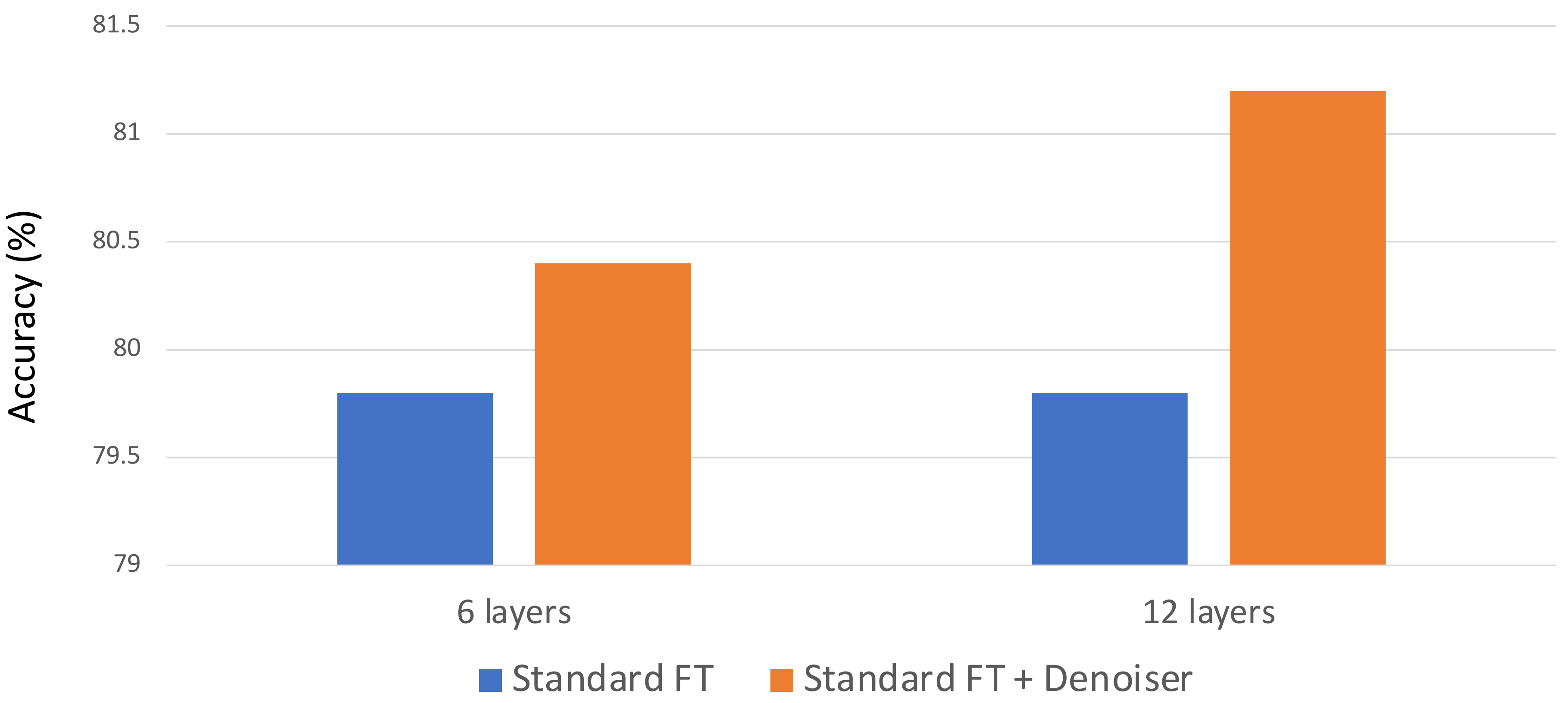
Experiment:

1. Train standard fine-tuned model initialized from pre-trained denoiser
2. Apply the denoiser to the predictions at test-time



Standard fine-tuning destroys some pre-trained output structure

In our SansType pseudocode-to-code dataset, re-applying the denoiser post-hoc improves accuracy by 0.5% to 1.5%



Outline

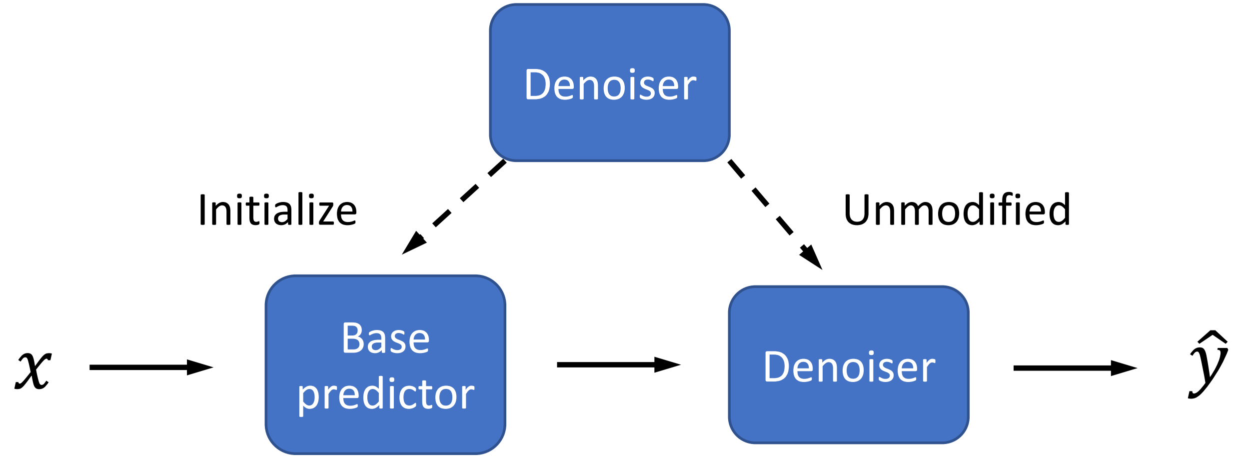
- Algorithm: Composed fine-tuning
- Analysis: Composing can reduce complexity
- Experiments: pseudocode-to-code and image generation

Outline

- Algorithm: Composed fine-tuning
- Analysis: Composing can reduce complexity
- Experiments: pseudocode-to-code and image generation

Composed Fine-Tuning

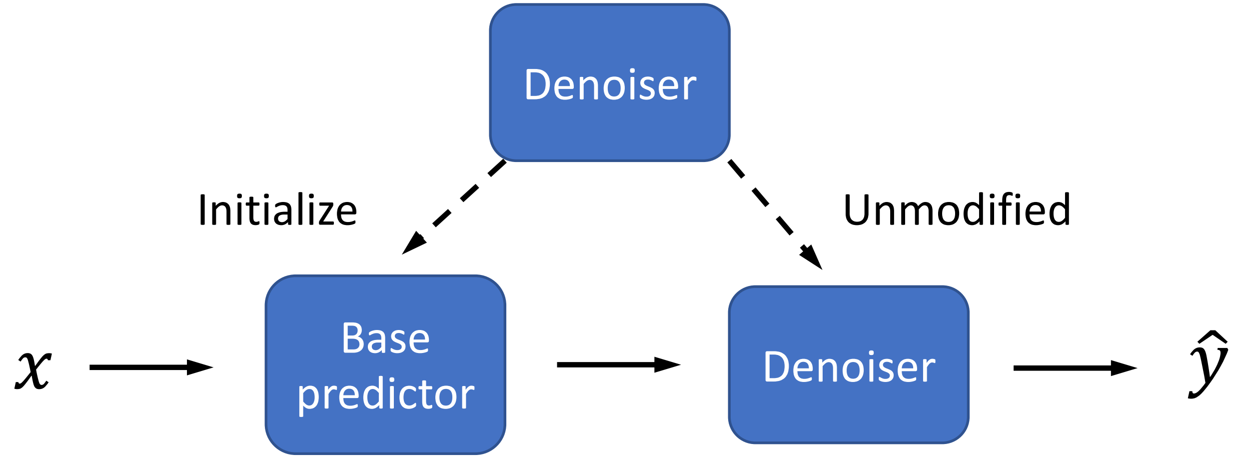
Given pre-trained denoiser Π, learn the base predictor f_θ composed with denoiser on labeled data:



$$Loss(x, y, \theta) = \ell(\Pi \circ f_\theta(x), y) + \lambda \ell(f_\theta(x), y)$$

Composed Fine-Tuning

Given pre-trained denoiser Π, learn the base predictor f_θ composed with denoiser on labeled data:

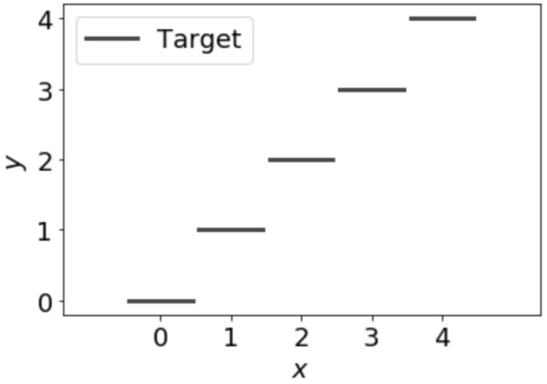


$$Loss(x, y, \theta) = \ell(\Pi \circ f_\theta(x), y) + \lambda \ell(f_\theta(x), y)$$

Offload complexity of learning output structure to the pre-trained denoiser

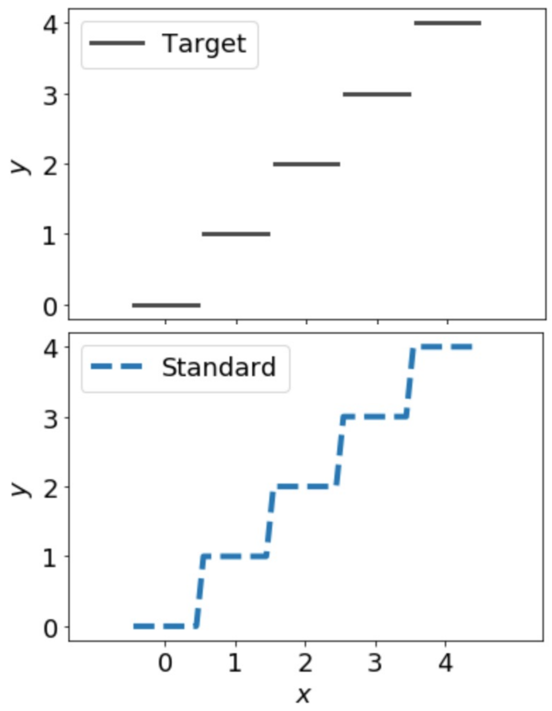
Composed fine-tuning simplifies models

- Target function: staircase function
  - valid outputs are integers



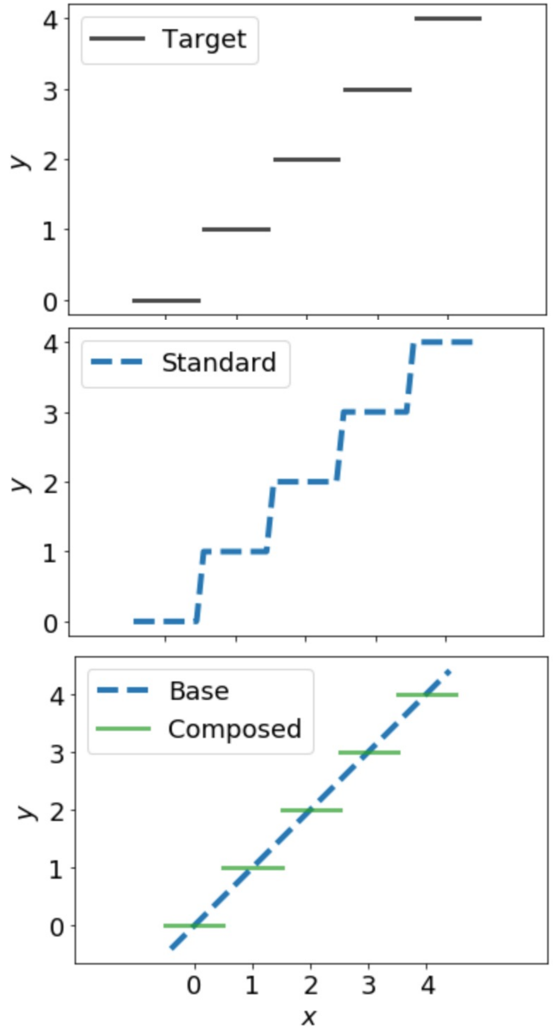
Composed fine-tuning simplifies models

- Target function: staircase function
  - valid outputs are integers
- Standard fine-tuning fits target directly:
  - requires a complex function with many slope changes



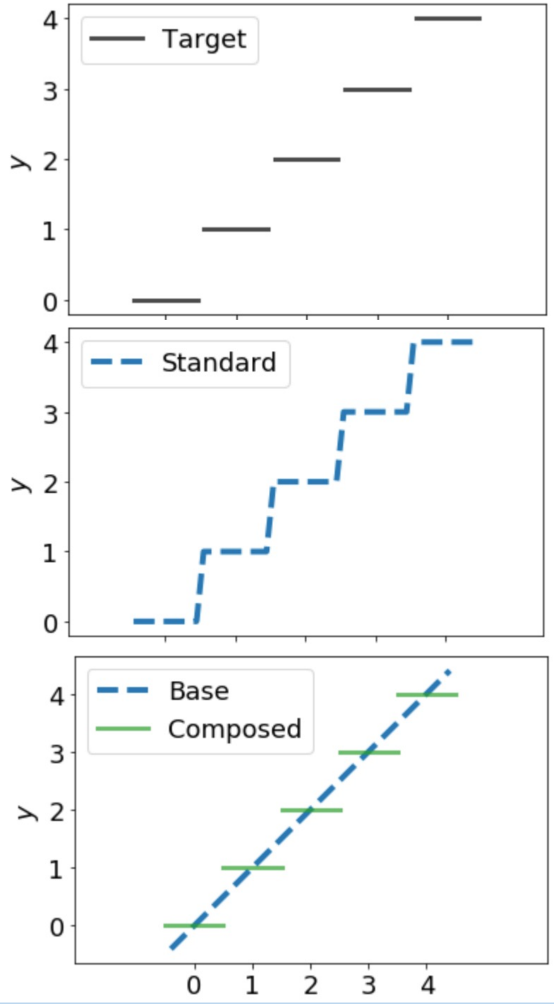
Composed fine-tuning simplifies models

- Target function: staircase function
  - valid outputs are integers
- Standard fine-tuning fits target directly:
  - requires a complex function with many slope changes
- Composed:
  - If denoiser rounds to nearest integer, base predictor is simple (linear)



Composed fine-tuning simplifies models

- Target function: staircase function
  - valid outputs are integers
- Standard fine-tuning fits target directly:
  - requires a complex function with many slope changes
- Composed:
  - If denoiser rounds to nearest integer, base predictor is simple (linear)



Base predictor is simple (linear) and extrapolates perfectly

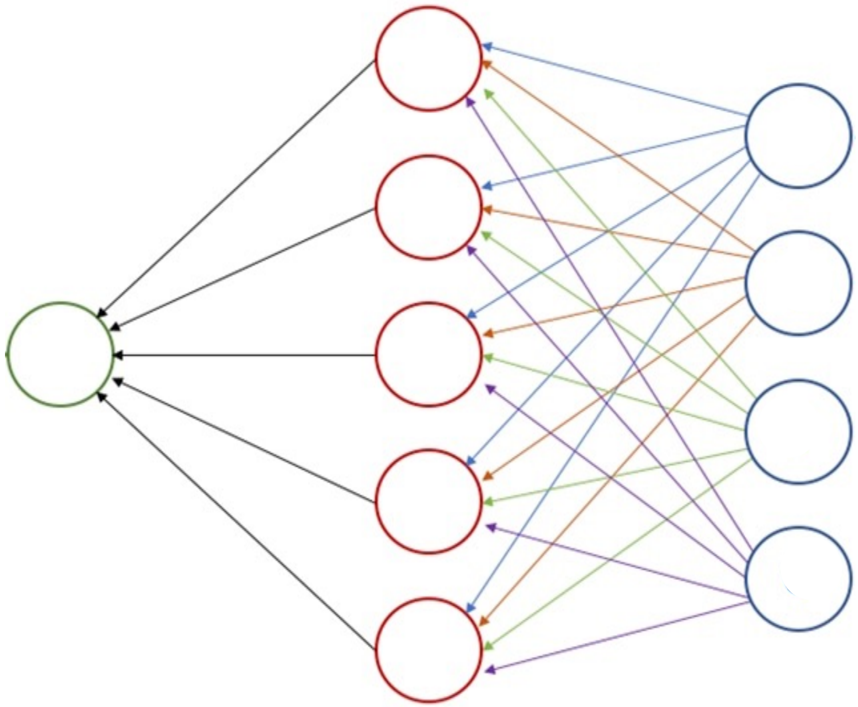
Outline

- Algorithm: Composed fine-tuning
- Analysis: Composing can reduce complexity
- Experiments: pseudocode-to-code and image generation

Analysis of composed fine-tuning

Setup

- Base model f_θ: 2 layer ReLU net with high-dim output
- Denoiser Π: projects to the nearest valid output
- Complexity measure $C(\theta)$: L2 norm of weights θ



Lower complexity -> better generalization

Analysis of composed fine-tuning

Can composed learning increase complexity?

Analysis of composed fine-tuning

Can composed learning increase complexity?

$$C(\theta_{std}) = \min_\theta\{C(\theta): f_\theta = f^*\}$$
$$C(\theta_{composed}) = \min_\theta\{C(\theta): \Pi \circ f_\theta = f^*\}$$

then

$$C(\theta_{composed}) \leq C(\theta_{std})$$

since $f_{\theta_{std}}$ is a feasible solution of the composed problem.

Analysis of composed fine-tuning

Can composed learning increase complexity?

$$C(\theta_{std}) = \min_\theta\{C(\theta): f_\theta = f^*\}$$
$$C(\theta_{composed}) = \min_\theta\{C(\theta): \Pi \circ f_\theta = f^*\}$$

then

$$C(\theta_{composed}) \leq C(\theta_{std})$$

since $f_{\theta_{std}}$ is a feasible solution of the composed problem.

Composing with a denoiser never increases the complexity

Composing can reduce model complexity

How much does composing decrease complexity?

Composing can reduce model complexity

How much does composing decrease complexity?

- We prove that composing can arbitrarily reduce complexity, depending on the stability of the target function

$$\frac{C(\theta_{std})}{C(\theta_{composed})} \rightarrow \infty \text{ as target function is more stable}$$

Composing can reduce model complexity

How much does composing decrease complexity?

- We prove that composing can arbitrarily reduce complexity, depending on the stability of the target function

$$\frac{C(\theta_{std})}{C(\theta_{composed})} \to \infty \text{ as target function is more stable}$$

More stability in input-output mapping -> composing helps more

Outline

- Algorithm: Composed fine-tuning
- Analysis: Composing can reduce complexity
- Experiments: pseudocode-to-code and image generation

Pseudocode-to-code

Example: from our generated SansType dataset, where pseudocode specifies all but the types

```
set var_8 to "str_2";
instantiate var_2;
read var_2 from stdin;
add "str_4" to the beginning of var_8;
set var_2 to false;
set var_5 to true;
if var_2 is true, ↩
        swap the values of var_2 and var_5;
if var_2 is true↩
        , set var_2 to the value of var_5↩
         and var_5 to the value of var_2;
output var_8 to stdout;
print var_2;
output var_5 to stdout;
```

Input pseudocode

```
int main () {
  string var_8 = "str_2";
  bool var_2;
  cin >> var_2;
  var_8 = "str_4" + var_8;
  var_2 = false;
  var_5 = true;
  if ( var_2 ) {
    string temp = var_2;
    var_2 = var_5;
    var_5 = temp; }
  if ( var_2 ) {
    bool temp = var_2;
    var_2 = var_5;
    var_5 = temp; }
  cout << var_8;
  cout << var_2;
  cout << var_5;
  return 0; }
```

Base predictor

```
int main () {
  string var_8 = "str_2";
  bool var_2;
  cin >> var_2;
  var_8 = "str_4" + var_8;
  var_2 = false;
  bool var_5 = true;
  if ( var_2 ) {
    bool temp = var_2;
    var_2 = var_5;
    var_5 = temp; }
  if ( var_2 ) {
    bool temp = var_2;
    var_2 = var_5;
    var_5 = temp; }
  cout << var_8;
  cout << var_2;
  cout << var_5;
  return 0; }
```

Composed

Pseudocode-to-code

- **Task**: *full-program* pseudocode-to-code translation with Transformers (Vaswani et al. 2017) – correct code passes test cases
  - Previous works consider line-by-line translation and compiler side information (Kulal et al 2019, Yasunaga and Liang 2020)

Pseudocode-to-code

- **Task**: *full-program* pseudocode-to-code translation with Transformers (Vaswani et al. 2017) – correct code passes test cases
  - Previous works consider line-by-line translation and compiler side information (Kulal et al 2019, Yasunaga and Liang 2020)
- **Validity**: code must compile & execute

SansType: synthetic pseudocode-to-code

- Dataset generation: generate pseudocode and code from templates
- Test sets
  - In-distribution (ID): same templates as training
  - Out-of-distribution (OOD): mix-and-matched ID pseudocode templates

SansType: synthetic pseudocode-to-code

- Dataset generation: generate pseudocode and code from templates
- Test sets
  - In-distribution (ID): same templates as training
  - Out-of-distribution (OOD): mix-and-matched ID pseudocode templates
- Example:
  - ID templates: `print <var>`, `output <var> to stdout`
  - OOD templates: `print <var> to stdout`, `output <var>`, `stdout <var>`

SPoC dataset for pseudocode-to-code

- Crowdsourced pseudocode for programming competition code from codeforces.com

SPoC dataset for pseudocode-to-code

- Crowdsourced pseudocode for programming competition code from codeforces.com
- Two test sets:
  - ID: test generalization to new pseudocode for previously seen programs
  - OOD: test generalization to new programs

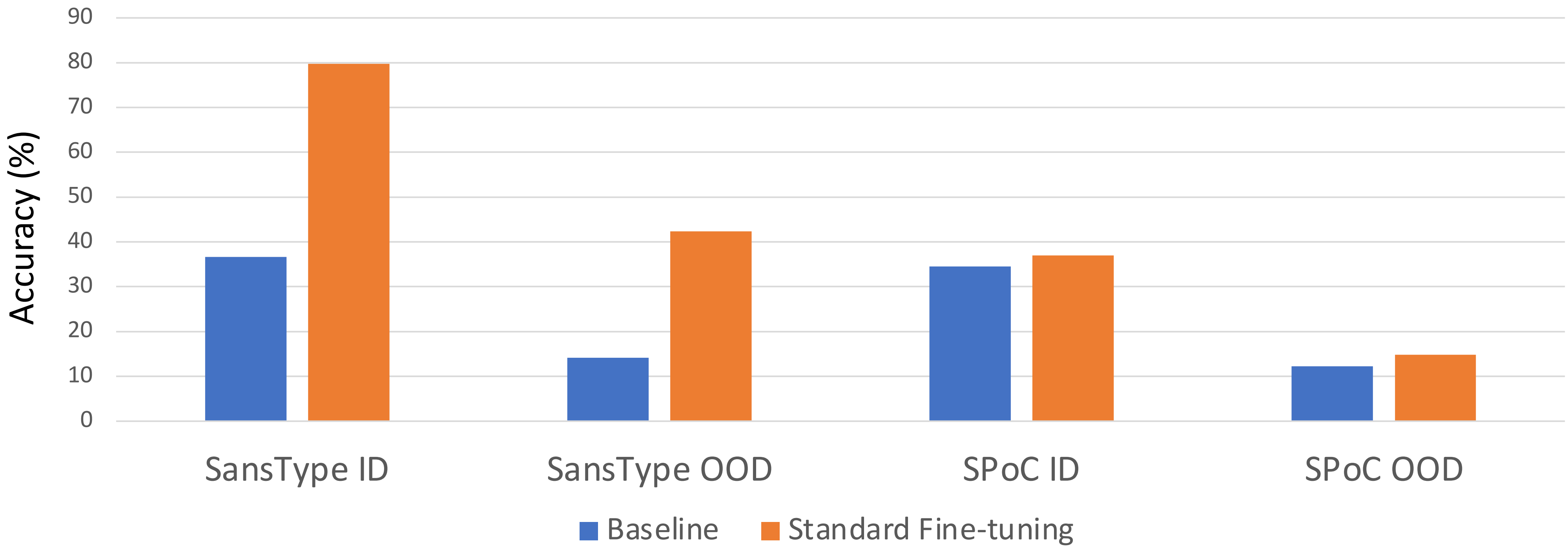
```
in function main
  let n be integer
  read n
  let A be vector of integers
  set size of A = n
  read n elements into A
  for all elements in A
    set min_i to i
    for j = i + 1 to size of A exclusive
      set min_i to j if A[min_i] > A[j]
    swap A[i], A[min_i]
  print all elements of A
```

```
int main() {
  int n;
  cin >> n;
  vector<int> A;
  A.resize(n);
  for(int i = 0; i < A.size(); i++) cin >> A[i];
  for(int i = 0; i < A.size(); i++) {
    int min_i = i;
    for(int j = i+1; j < A.size(); j++) {
      if(A[min_i] > A[j]) { min_i = j; }
    swap(A[i], A[min_i]);
  for(int i=0; i<A.size(); i++) cout<<A[i]<<" ";
}
```

Kulal et al. 2019

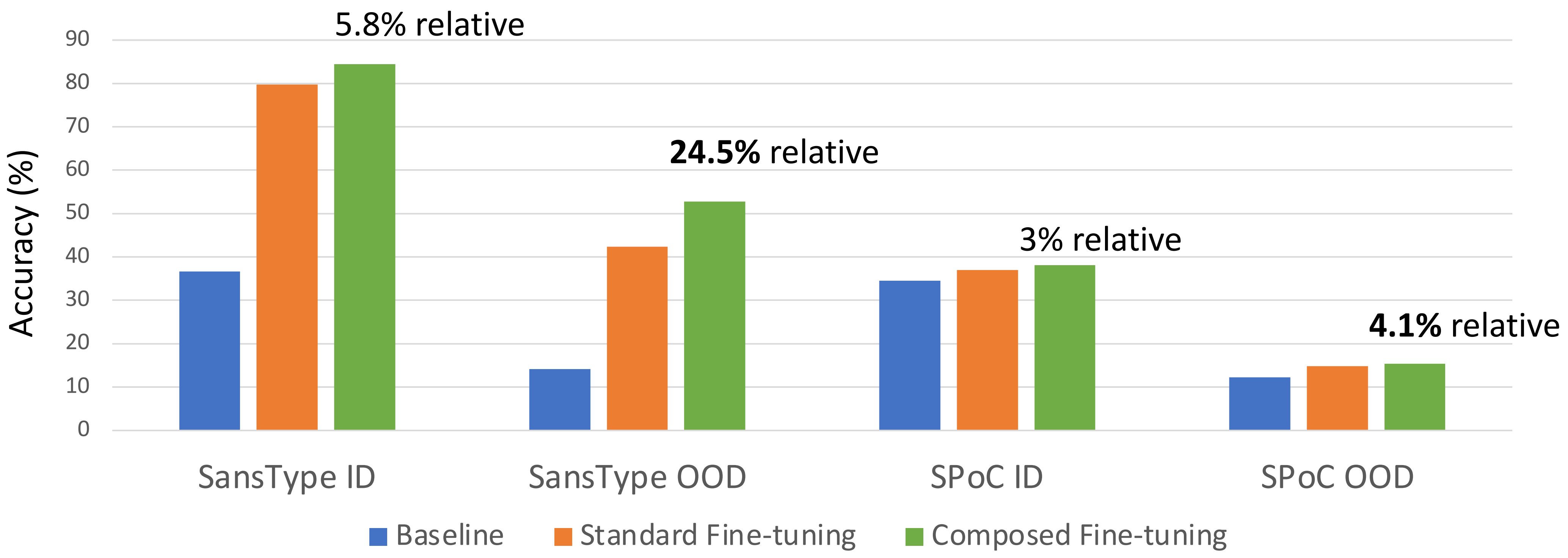
Pseudocode-to-code results

- Standard fine-tuning improves over baseline (no pretraining)



Pseudocode-to-code results

- Standard fine-tuning improves over baseline (no pretraining)
- Composed fine-tuning improves both ID and OOD, but especially OOD

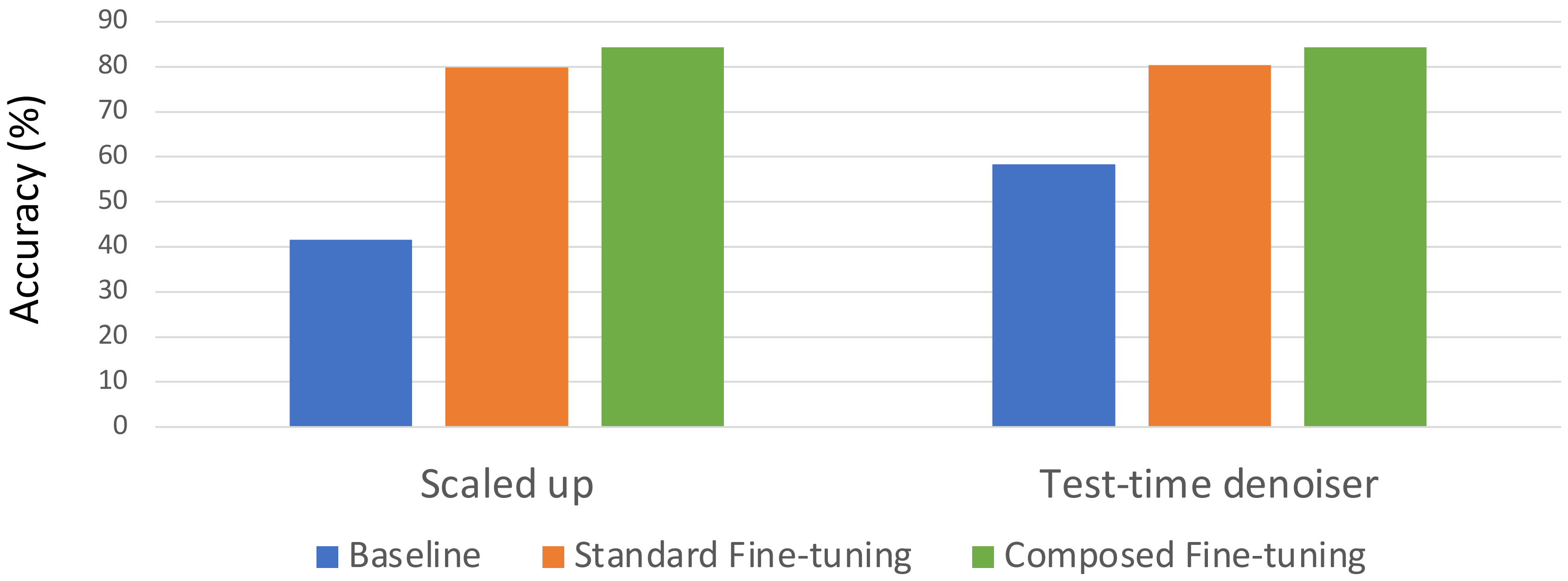


Stronger baselines

- **Scaled-up baseline**: double the number of layers
  - Note: pre-trained denoiser is also doubled for these models
- **Test-time denoiser**: apply denoiser post-hoc to baseline/standard fine-tuning

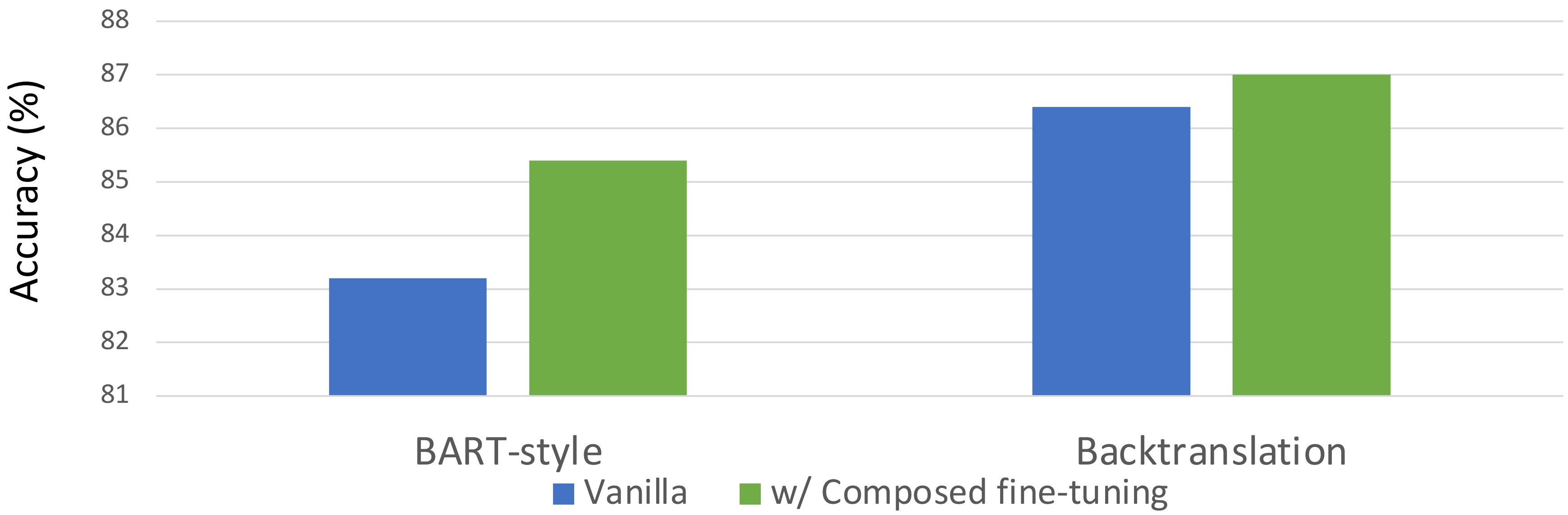
Stronger baselines

- **Scaled-up baseline**: double the number of layers
  - Note: pre-trained denoiser is also doubled for these models
- **Test-time denoiser**: apply denoiser post-hoc to baseline/standard fine-tuning
- Composed fine-tuning still improves over these (SansType)



Composed fine-tuning is complementary with fancier methods

- BART-style fine-tuning
  - Two-stage process: freeze later layers first, then fully finetune
- Backtranslation (Sennrich et al. 2015)
  - Use unlabeled output data during fine-tuning to create synthetic inputs



OOD conditional image generation

Train

[D, Disney font] → 

Test

[i, Disney font] → 

- **Task**: given font family and character, output a font image
- **Validity**: font images have sharp lines, adhere to font styling

OOD conditional image generation

Train

[D, Disney font] → 

Test

[i, Disney font] → 

- **Task**: given font family and character, output a font image
- **Validity**: font images have sharp lines, adhere to font styling

- Useful for prototyping new fonts: supply a few characters and the model fills in the rest
- Extrapolate to new character-font pairs at test time

OOD conditional image generation

- Image generations for some random fonts (MLP base model)

(a) Direct

OOD conditional image generation

- Image generations for some random fonts (MLP base model)

(a) Direct

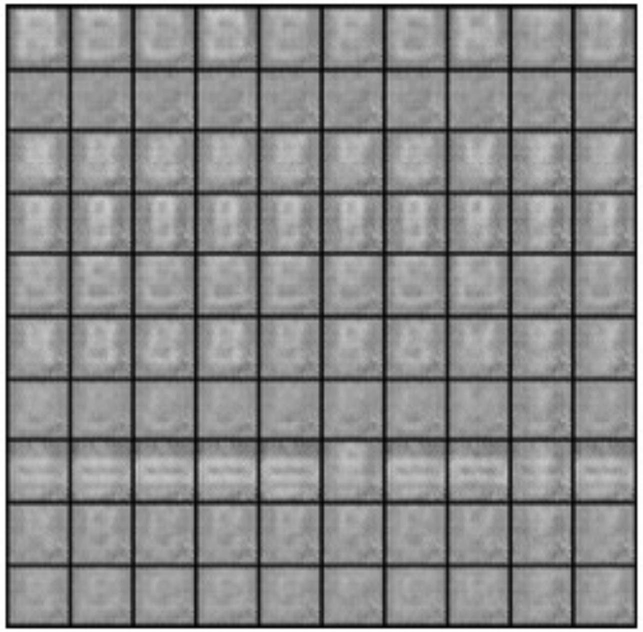
(b) Composed

OOD conditional image generation

- Image generations for some random fonts (MLP base model)

(a) Direct

(b) Composed



(c) Base

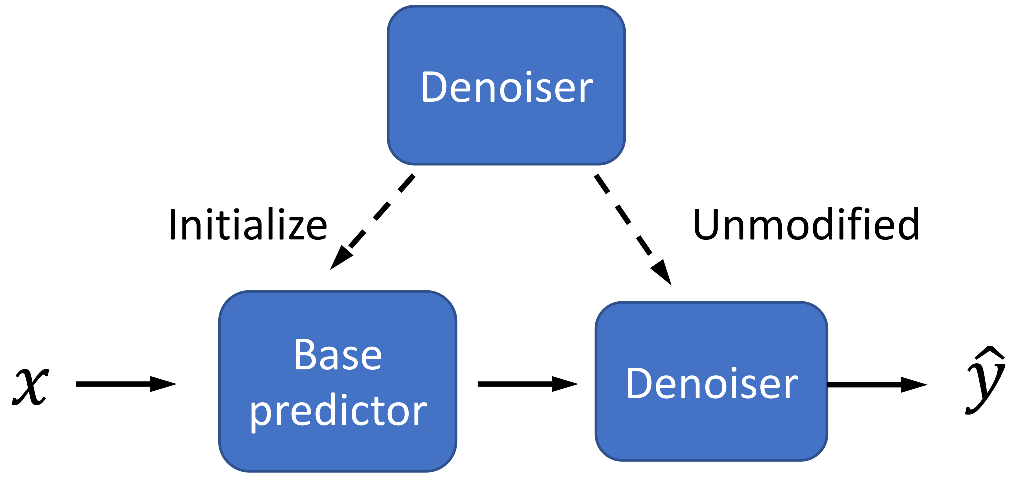
Base predictor is simpler – gray and blobby outputs

Takeaways

- Standard fine-tuning can destroy some output structure pre-trained by denoising **unlabeled outputs**

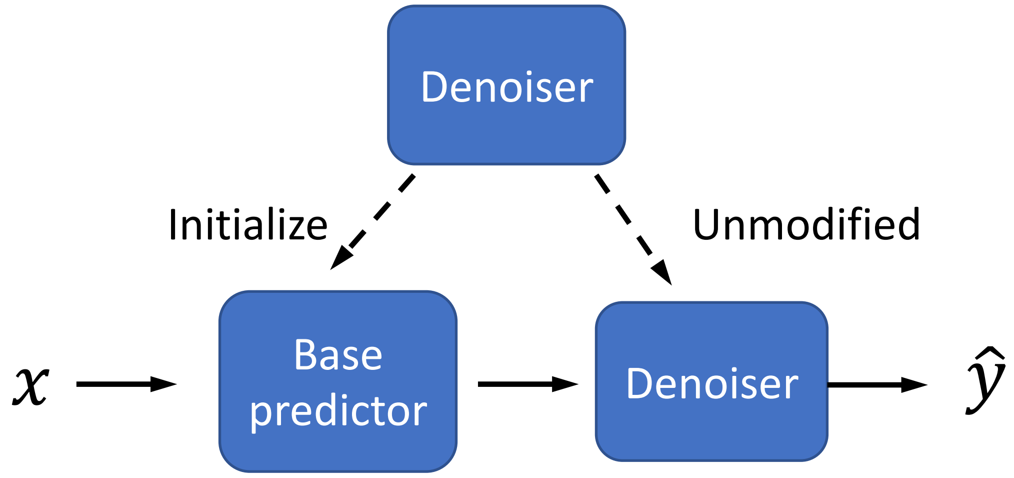
Takeaways

- Standard fine-tuning can destroy some output structure pre-trained by denoising **unlabeled outputs**
- **Composed fine-tuning** preserves it by freezing the denoiser. Base predictor only needs to learn the input-output mapping



Takeaways

- Standard fine-tuning can destroy some output structure pre-trained by denoising **unlabeled outputs**
- **Composed fine-tuning** preserves it by freezing the denoiser. Base predictor only needs to learn the input-output mapping
- Composed fine-tuning leads to **reduced complexity** and **better generalization**, especially **OOD**!



Thanks!

We thank Michi Yasunaga, Robin Jia, Albert Gu, Karan Goel, Rohan Taori, and reviewers for helpful discussions and comments. SMX is supported by an NDSEG Graduate Fellowship. The work is partially supported by a PECASE award, SDSI, and SAIL at Stanford University.