SOAR: A Synthesis Approach for Open-Source API Refactoring

Ansong Ni
Yale University

Daniel Ramos
INESC-ID / IST, CMU

Aidan Z. H. Yang
Queen's University

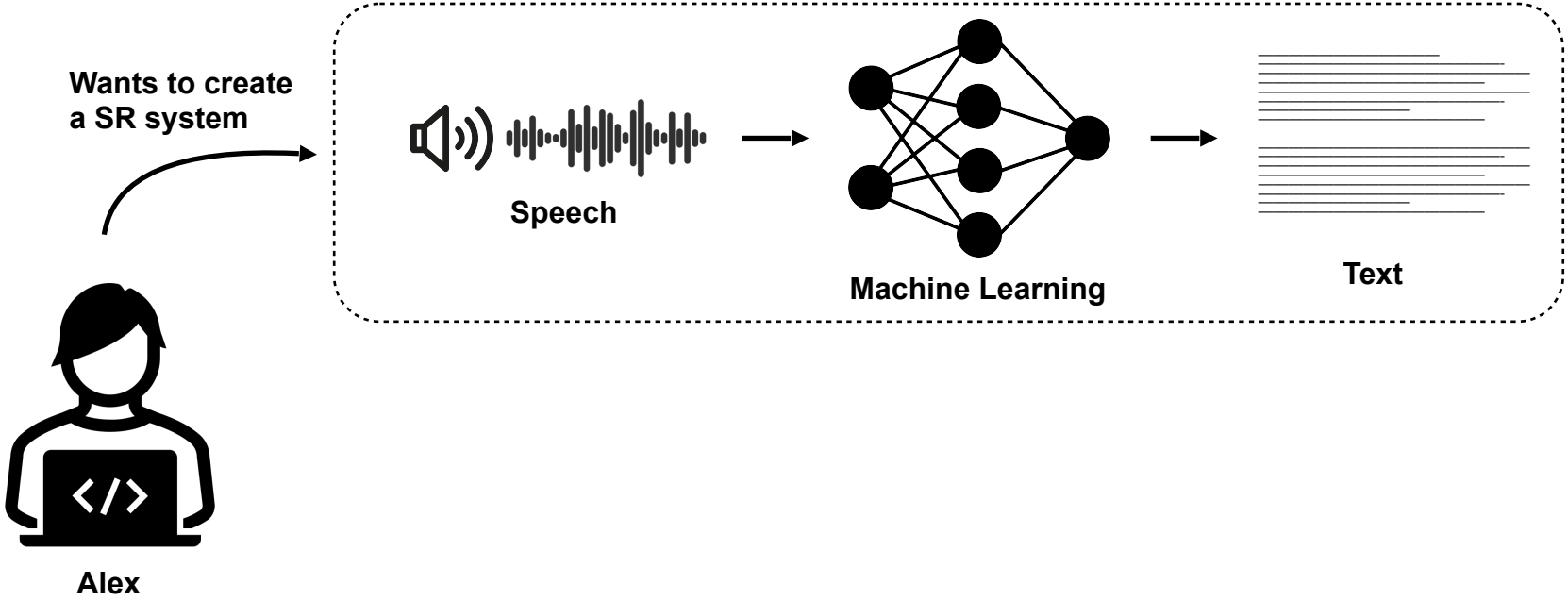
Inês Lynce
INESC-ID / IST

Vasco Manquinho
INESC-ID / IST

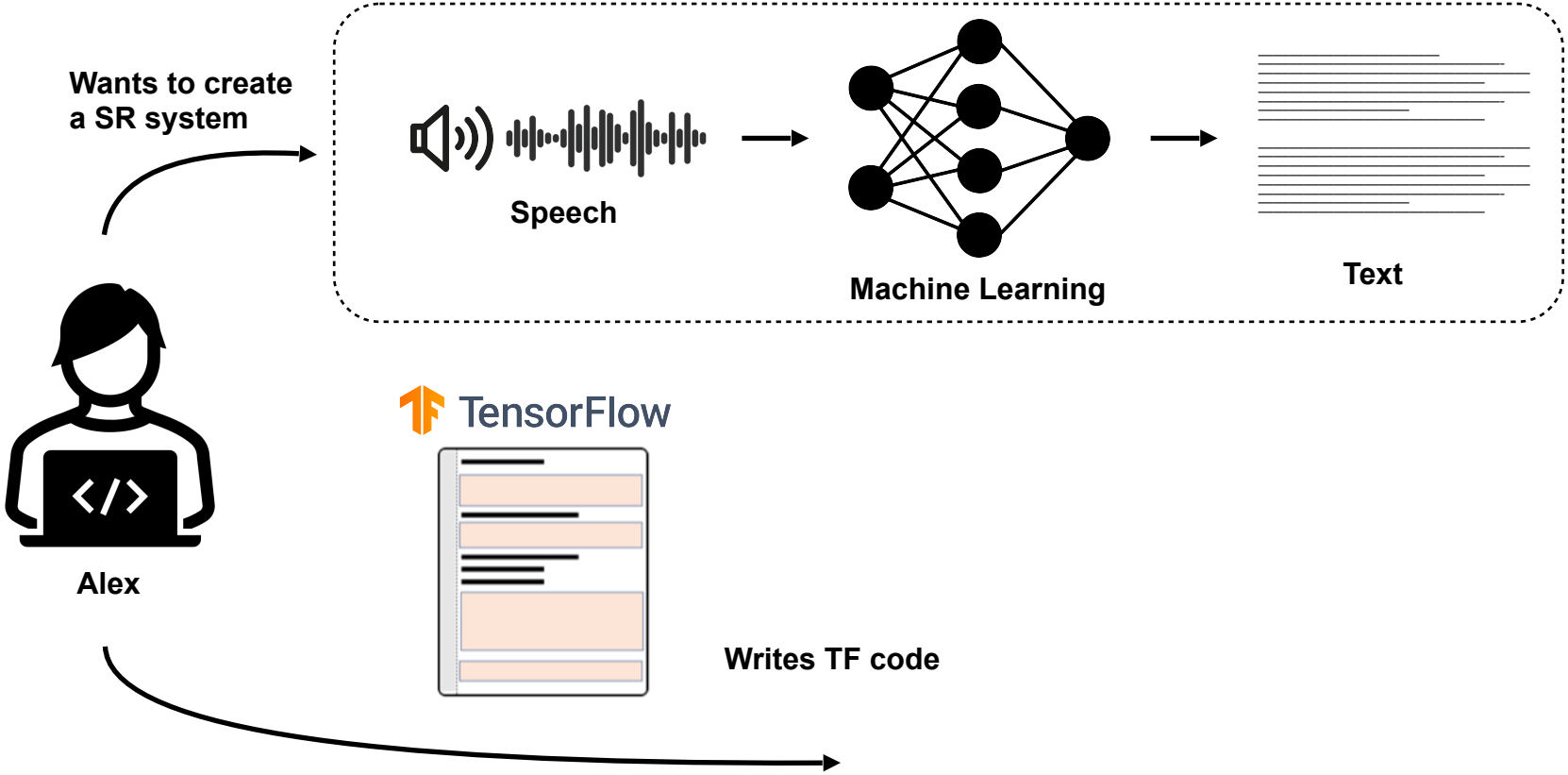
Ruben Martins
CMU

Claire Le Goues
CMU

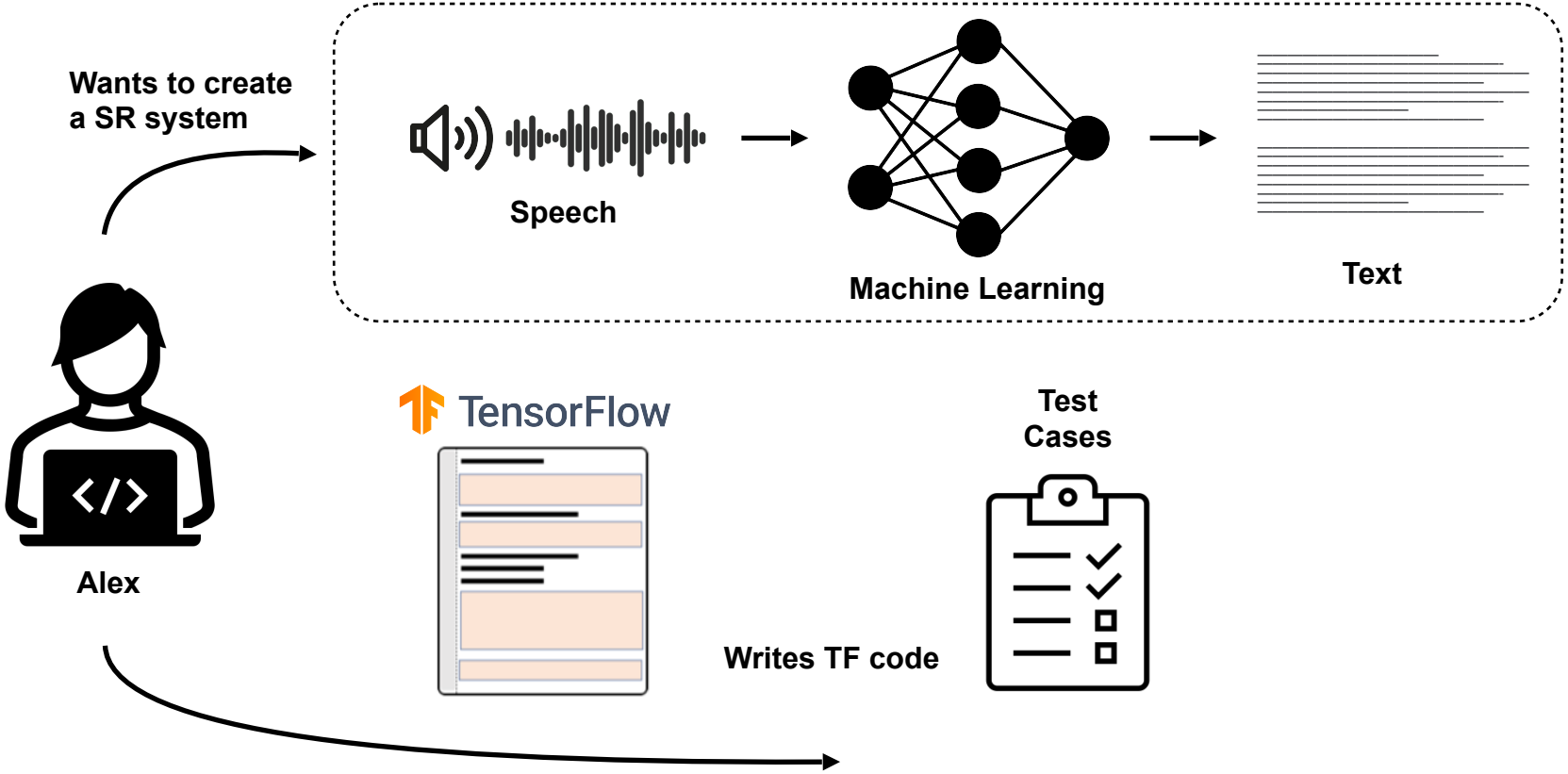
MOTIVATION


Wants to create
a SR system
Speech
Machine Learning
Text
Alex

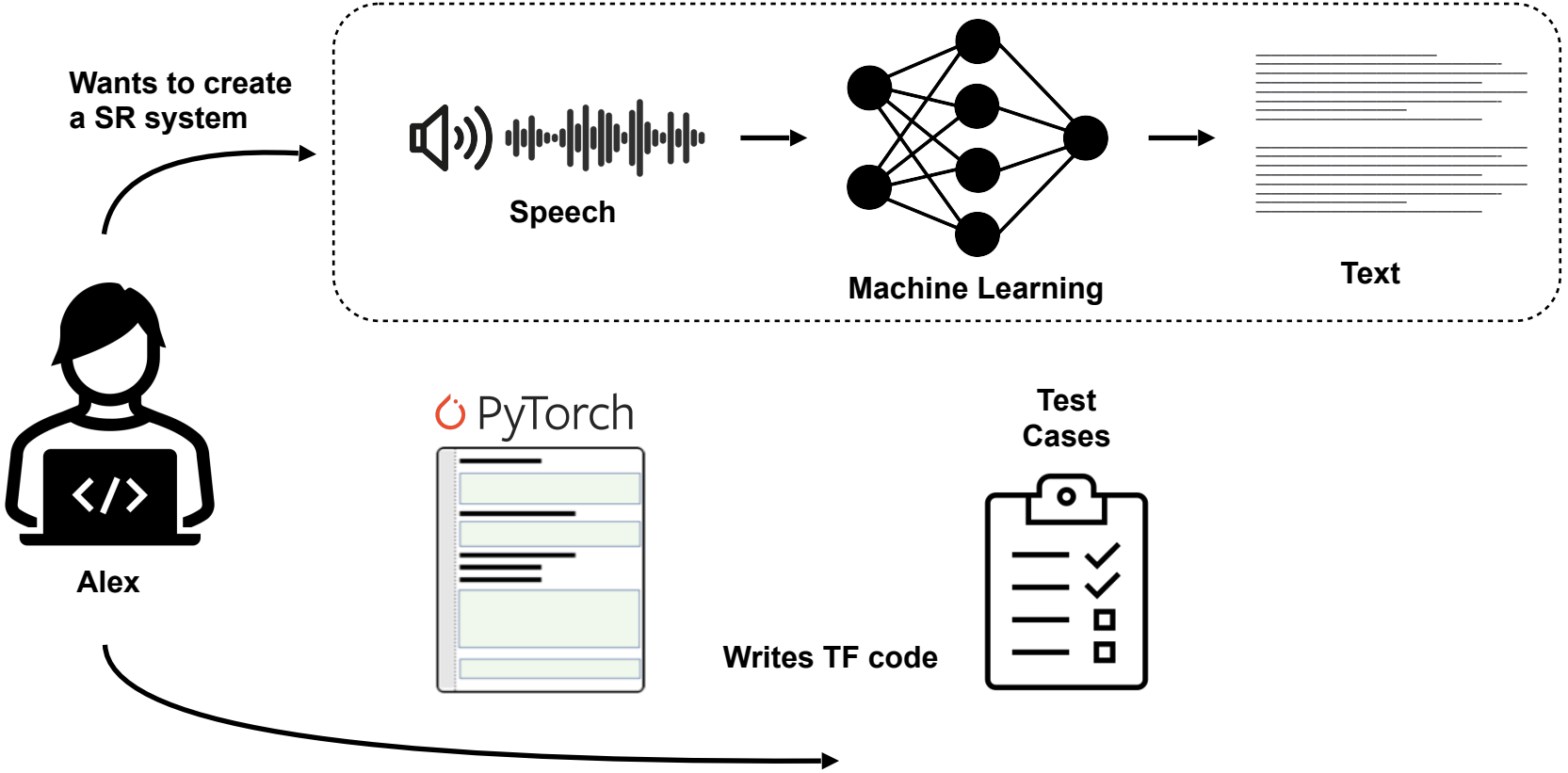
MOTIVATION


Wants to create
a SR system
Speech
Machine Learning
Text
Alex
TensorFlow
Writes TF code

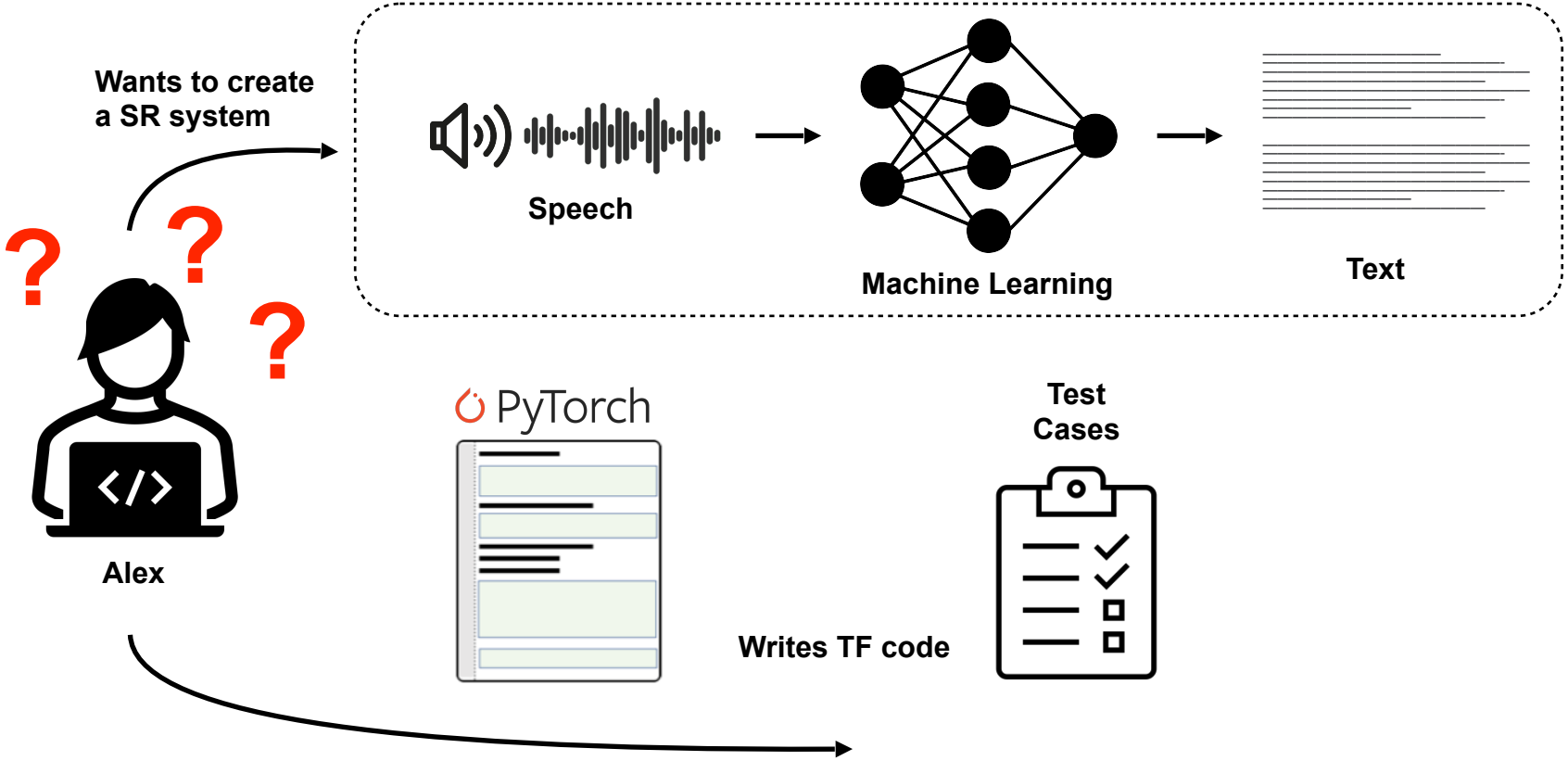
MOTIVATION


Wants to create
a SR system
Speech
Machine Learning
Text
Alex
TensorFlow
Test
Cases
Writes TF code

MOTIVATION

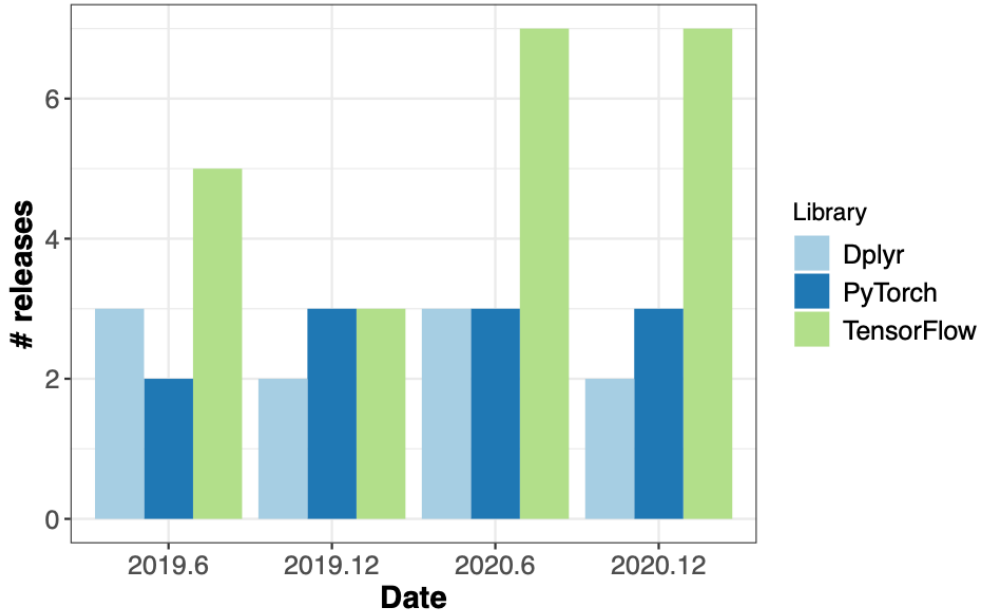

Wants to create
a SR system
Speech
Machine Learning
Text
Alex
PyTorch
Test
Cases
Writes TF code

MOTIVATION


Wants to create
a SR system
Speech
Machine Learning
Text
Alex
PyTorch
Test
Cases
Writes TF code

API REFACTORING PROBLEM

- This is an API refactoring / migration problem
- Libraries no longer supported, new alternatives, updates, etc.



SOAR: A Synthesis Approach for API Refactoring

- SOAR's an end-to-end technique for API migration

```python
import tensorflow as tf

class TensorLin(tf.keras.Model):
    def __init__(self, **kwargs):
        super(TensorLin, self).__init__()
        self.fc1 = tf.keras.layers.Dense(128)
        self.fc2 = tf.keras.layers.Dense(256)
        self.out = tf.keras.layers.Dense(10)

    def call(self, x):
        x = self.fc1(x)
        x = self.fc2(x)
        x = self.out(x)
        return x
```

Source (TF)

SOAR: A Synthesis Approach for API Refactoring

- SOAR's an end-to-end technique for API migration

```python
import tensorflow as tf

class TensorLin(tf.keras.Model):
    def __init__(self, **kwargs):
        super(TensorLin, self).__init__()
        self.fc1 = tf.keras.layers.Dense(128)
        self.fc2 = tf.keras.layers.Dense(256)
        self.out = tf.keras.layers.Dense(10)

    def call(self, x):
        x = self.fc1(x)
        x = self.fc2(x)
        x = self.out(x)
        return x
```

Source (TF)

```python
import torch

class TorchLin(torch.nn.Module):
    def __init__(self, **kwargs):
        super(TorchLin, self).__init__()
        self.fc1 = torch.nn.Linear(50,128)
        self.fc2 = torch.nn.Linear(128,256)
        self.out = torch.nn.Linear(256,10)

    def forward(self, x):
        x = self.fc1(x)
        x = self.fc2(x)
        x = self.out(x)
        return x
```

Target

SOAR: A Synthesis Approach for API Refactoring

- SOAR is built upon three key observations:

  1. Data science APIs are well documented

tf.keras.layers.Dense

See Stable | See Nightly

TensorFlow 1 version | View source on GitHub

Just your regular densely-connected NN layer.

Inherits From: `Layer`, `Module`

View aliases

Used in the notebooks

Used in the guide	Used in the tutorials
• Save and load Keras models • The Sequential model • Training and evaluation with the built-in methods • The Functional API • Making new Layers and Models via subclassing	• Overfit and underfit • Time series forecasting • Intro to Autoencoders • Transformer model for language understanding • Load CSV data

`Dense` implements the operation: `output = activation(dot(input, kernel) + bias)` where `activation` is the element-wise activation function passed as the `activation` argument, `kernel` is a weights matrix created by the layer, and `bias` is a bias vector created by the layer (only applicable if `use_bias` is `True`).

★ **Note:** If the input to the layer has a rank greater than 2, then **`Dense`** computes the dot product between the **`inputs`** and the **`kernel`** along the last axis of the **`inputs`** and axis 1 of the **`kernel`** (using **`tf.tensordot`**). For example, if input

Arguments
`units`
Positive integer, dimensionality of the output space.
`activation`
Activation function to use. If you don't specify anything, no activation is applied (ie. "linear" activation: **`a(x) = x`**).
`use_bias`
Boolean, whether the layer uses a bias vector.
`kernel_initializer`
Initializer for the **`kernel`** weights matrix.
`bias_initializer`
Initializer for the bias vector.
`kernel_regularizer`
Regularizer function applied to the **`kernel`** weights matrix.

LINEAR

CLASS `torch.nn.Linear(in_features, out_features, bias=True)` [SOURCE]

Applies a linear transformation to the incoming data: $y = xA^T + b$

This module supports TensorFloat32.

Parameters

- **in_features** – size of each input sample
- **out_features** – size of each output sample
- **bias** – If set to `False`, the layer will not learn an additive bias. Default: `True`

LINEAR

CLASS `torch.nn.Linear(in_features, out_features, bias=True)` [SOURCE]

Applies a linear transformation to the incoming data: $y = xA^T + b$

This module supports TensorFloat32.

Parameters

- **in_features** – size of each input sample
- **out_features** – size of each output sample
- **bias** – If set to `False`, the layer will not learn an additive bias. Default: `True`

Shape:

- Input: $(N, *, H_{in})$ where $*$ means any number of additional dimensions and $H_{in} = \text{in\_features}$
- Output: $(N, *, H_{out})$ where all but the last dimension are the same shape as the input and $H_{out} = \text{out\_features}$.

SOAR: API MAPPINGS

- Using the mappings we can replace the source APIs

```python
import tensorflow as tf

class TensorLin(tf.keras.Model):
    def __init__(self, **kwargs):
        super(TensorLin, self).__init__()
        self.fc1 = tf.keras.layers.Dense(128)
        self.fc2 = tf.keras.layers.Dense(256)
        self.out = tf.keras.layers.Dense(10)

    def call(self, x):
        x = self.fc1(x)
        x = self.fc2(x)
        x = self.out(x)
        return x
```

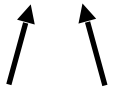
```python
import torch

class TorchLin(torch.nn.Module):
    def __init__(self, **kwargs):
        super(TorchLin, self).__init__()
        self.fc1 = torch.nn.Linear(?, ?)
        self.fc2 = torch.nn.Linear(?, ?)
        self.out = torch.nn.Linear(?, ?)

    def forward(self, x):
        x = self.fc1(x)
        x = self.fc2(x)
        x = self.out(x)
        return x
```

PROBLEM

- How can we compute the new arguments?

```tensorflow
import tensorflow as tf

class TensorLin(tf.keras.Model):
    def __init__(self, **kwargs):
        super(TensorLin, self).__init__()
        self.fc1 = tf.keras.layers.Dense(128)
        self.fc2 = tf.keras.layers.Dense(256)
        self.out = tf.keras.layers.Dense(10)

    def call(self, x):
        x = self.fc1(x)
        x = self.fc2(x)
        x = self.out(x)
        return x
```

```pytorch
import torch

class TorchLin(torch.nn.Module):
    def __init__(self, **kwargs):
        super(TorchLin, self).__init__()
        self.fc1 = torch.nn.Linear(?, ?)
        self.fc2 = torch.nn.Linear(?, ?)
        self.out = torch.nn.Linear(?, ?)

    def forward(self, x):
        x = self.fc1(x)
        x = self.fc2(x)
        x = self.out(x)
        return x
```



SOAR: A Synthesis Approach for API Refactoring

- SOAR is built upon three key observations:

  1. Data science APIs are really well documented
  2. **Computing arguments is a great fit for program synthesis**

SOAR: Program Synthesis

- Documentation has information about argument types, constraints
- Enumerate possible values and validate

```
import torch

class TorchLin(torch.nn.Module):
    def __init__(self, **kwargs):
        super(TorchLin, self).__init__()
        self.fc1 = torch.nn.Linear(int: x1, int: x2)
        self.fc2 = torch.nn.Linear(int: x3, int: x4)
        self.out = torch.nn.Linear(int: x5, int: x5)

    def forward(self, x):
        x = self.fc1(x)
        x = self.fc2(x)
        x = self.out(x)
        return x
```

SOAR: Program Synthesis

- Documentation has information about argument types, constraints
- Enumerate possible values and validate

```
import torch

class TorchLin(torch.nn.Module):
    def __init__(self, **kwargs):
        super(TorchLin, self).__init__()
        self.fc1 = torch.nn.Linear(int: x1, int: x2)
        self.fc2 = torch.nn.Linear(int: x3, int: x4)
        self.out = torch.nn.Linear(int: x5, int: x5)

    def forward(self, x):
        x = self.fc1(x)
        x = self.fc2(x)
        x = self.out(x)
        return x
```

`x1 = -1 & x2 = 256` ✗

`x1 = 50 & x2 = -1` ✗

`x1 = 50 & x2 = 128` ✓

SOAR: A Synthesis Approach for API Refactoring

- SOAR is built upon three key observations:
  1. Data science APIs are really well documented
  2. Computing arguments is a great fit for program synthesis

SOAR uses documentation to guide the synthesis

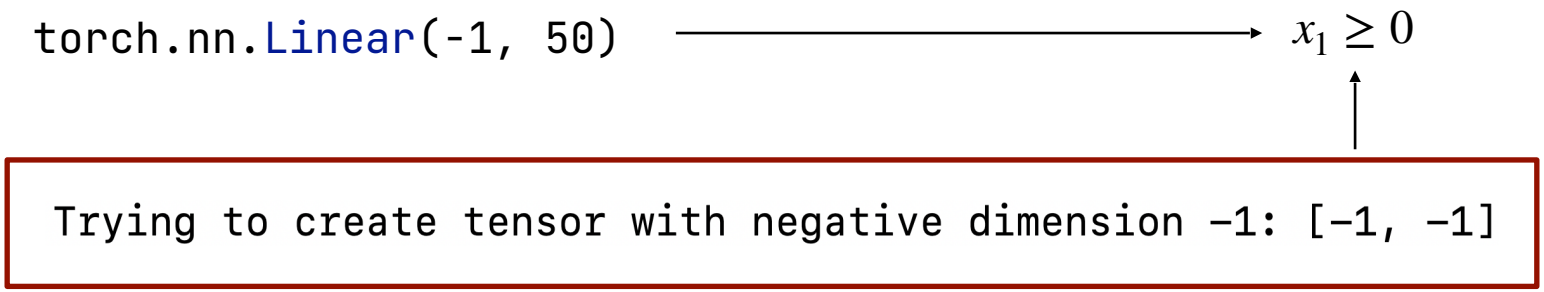
SOAR: A Synthesis Approach for API Refactoring

- SOAR is built upon three key observations:

  1. Data science APIs are really well documented
  2. Computing arguments is a great fit for program synthesis
  3. **Well-documented APIs have informative error messages**

SOAR: Error Message Understanding

- SOAR uses error messages to speed-up the synthesis process
- Generates constraints from the errors to prune the search space



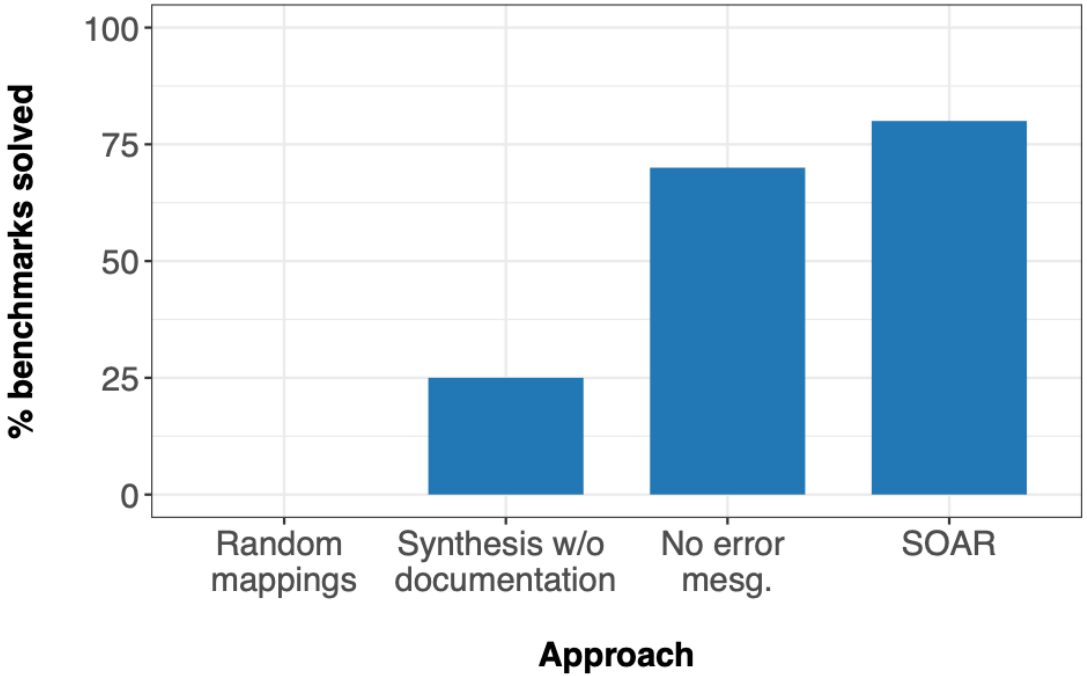
Benchmarks

- Evaluated SOAR on 40 benchmarks from two domains:
  - 20 tasks from TensorFlow to PyTorch
  - 20 tasks from dplyr (R) to pandas (Python)

- Migrations tasks from TensorFlow to PyTorch correspond to migrating a network with 3 to 44 layers, including:
  - Auto-encoders
  - VGG
  - AlexNet
  - LeNet

SOAR TRANSLATES 80% OF THE BENCHMARKS

- Benchmarks solved for the TensorFlow to PyTorch task (1h timeout)



SOAR: Migration Example

```python
from tensorflow import keras

class AutoEncoder(keras.Model):
  def __init__(self, **kwargs):
    super(AutoEncoder, self).__init__()
    self.conv1 = keras.layers.Conv2D(filters=32,
                                     kernel_size=3,
                                     strides=(2,2))

    self.relu1 = keras.layers.ReLU()
    self.conv2 = keras.layers.Conv2D(filters=64,
                                     kernel_size=3,
                                     strides=(2,2))

    self.relu2 = keras.layers.ReLU()
    self.flatt = keras.layers.Flatten()
    self.dense = keras.layers.Dense(10)
    self.dense = keras.layers.Dense(units=1568)
    self.relu3 = keras.layers.ReLU()
    self.resha = keras.layers.Reshape(
                            target_shape=(7, 7, 32))
    self.conv3 = keras.layers.Conv2DTranspose(
                                     filters=64,
                                     kernel_size=3,
```

```python
import torch

class AutoEncoder(torch.nn.Module):
  def __init__(self, **kwargs):
    super(AutoEncoder, self).__init__()
    self.many_0 = lambda x: x.permute(0, 3, 1, 2)
    self.var816 = torch.nn.Conv2d(32, 32, (3, 3),
                                  stride=(2, 2),
                                  padding=(0, 0))
    self.var817 = torch.nn.ReLU()
    self.many_1 = lambda x: x.permute(0, 1, 3, 2)
    self.var1528 = torch.nn.Conv2d(32, 64, (3, 3),
                                   stride=(2, 2),
                                   padding=(0, 0))
    self.var1529 = torch.nn.ReLU()
    self.var1530 = lambda x: torch.flatten(x, 1)
    self.var1710 = torch.nn.Linear(64, 10)
    self.var1801 = torch.nn.Linear(10, 1568)
    self.var1802 = torch.nn.ReLU()
    self.many_2 = lambda x: x.view(10, 7, 7, 32)

    self.many_3 = lambda x: x.permute(0, 3, 1, 2)
    self.var5089 = torch.nn.ConvTranspose2d(32, 64,
                                            (3,3),
```

SOAR: Migration Example

```python
from tensorflow import keras

class AutoEncoder(keras.Model):
  def __init__(self, **kwargs):
    super(AutoEncoder, self).__init__()
    self.conv1 = keras.layers.Conv2D(filters=32,
                                     kernel_size=3,
                                     strides=(2,2))

    self.relu1 = keras.layers.ReLU()
    self.conv2 = keras.layers.Conv2D(filters=64,
                                     kernel_size=3,
                                     strides=(2,2))

    self.relu2 = keras.layers.ReLU()
    self.flatt = keras.layers.Flatten()
    self.dense = keras.layers.Dense(10)
    self.dense = keras.layers.Dense(units=1568)
    self.relu3 = keras.layers.ReLU()
    self.resha = keras.layers.Reshape(
                            target_shape=(7, 7, 32))
    self.conv3 = keras.layers.Conv2DTranspose(
                                     filters=64,
                                     kernel_size=3,
```

```python
import torch

class AutoEncoder(torch.nn.Module):
  def __init__(self, **kwargs):
    super(AutoEncoder, self).__init__()
    self.many_0 = lambda x: x.permute(0, 3, 1, 2)
    self.var816 = torch.nn.Conv2d(32, 32, (3, 3),
                                  stride=(2, 2),
                                  padding=(0, 0))
    self.var817 = torch.nn.ReLU()
    self.many_1 = lambda x: x.permute(0, 1, 3, 2)
    self.var1528 = torch.nn.Conv2d(32, 64, (3, 3),
                                   stride=(2, 2),
                                   padding=(0, 0))
    self.var1529 = torch.nn.ReLU()
    self.var1530 = lambda x: torch.flatten(x, 1)
    self.var1710 = torch.nn.Linear(64, 10)
    self.var1801 = torch.nn.Linear(10, 1568)
    self.var1802 = torch.nn.ReLU()
    self.many_2 = lambda x: x.view(10, 7, 7, 32)

    self.many_3 = lambda x: x.permute(0, 3, 1, 2)
    self.var5089 = torch.nn.ConvTranspose2d(32, 64,
                                            (3,3),
```

SOAR: Migration Example

```python
from tensorflow import keras

class AutoEncoder(keras.Model):
  def __init__(self, **kwargs):
    super(AutoEncoder, self).__init__()
    self.conv1 = keras.layers.Conv2D(filters=32,
                                     kernel_size=3,
                                     strides=(2,2))

    self.relu1 = keras.layers.ReLU()
    self.conv2 = keras.layers.Conv2D(filters=64,
                                     kernel_size=3,
                                     strides=(2,2))

    self.relu2 = keras.layers.ReLU()
    self.flatt = keras.layers.Flatten()
    self.dense = keras.layers.Dense(10)
    self.dense = keras.layers.Dense(units=1568)
    self.relu3 = keras.layers.ReLU()
    self.resha = keras.layers.Reshape(
                            target_shape=(7, 7, 32))
    self.conv3 = keras.layers.Conv2DTranspose(
                                     filters=64,
                                     kernel_size=3,
```

```python
import torch

class AutoEncoder(torch.nn.Module):
  def __init__(self, **kwargs):
    super(AutoEncoder, self).__init__()
    self.many_0 = lambda x: x.permute(0, 3, 1, 2)
    self.var816 = torch.nn.Conv2d(32, 32, (3, 3),
                                  stride=(2, 2),
                                  padding=(0, 0))
    self.var817 = torch.nn.ReLU()
    self.many_1 = lambda x: x.permute(0, 1, 3, 2)
    self.var1528 = torch.nn.Conv2d(32, 64, (3, 3),
                                   stride=(2, 2),
                                   padding=(0, 0))
    self.var1529 = torch.nn.ReLU()
    self.var1530 = lambda x: torch.flatten(x, 1)
    self.var1710 = torch.nn.Linear(64, 10)
    self.var1801 = torch.nn.Linear(10, 1568)
    self.var1802 = torch.nn.ReLU()
    self.many_2 = lambda x: x.view(10, 7, 7, 32)

    self.many_3 = lambda x: x.permute(0, 3, 1, 2)
    self.var5089 = torch.nn.ConvTranspose2d(32, 64,
                                            (3,3),
```

SOAR: Migration Example

```
19      self.relu3 = keras.layers.ReLU()
20      self.resha = keras.layers.Reshape(
21                               target_shape=(7, 7, 32))
22      self.conv3 = keras.layers.Conv2DTranspose(
23                                       filters=64,
24                                       kernel_size=3,
25                                       strides=2)
26
27      self.relu4 = keras.layers.ReLU()
28      self.conv4 = keras.layers.Conv2DTranspose(
29                                       filters=32,
30                                       kernel_size=3,
31                                       strides=2)
32
33      self.relu5 = keras.layers.ReLU()
34      self.conv5 = keras.layers.Conv2DTranspose(
35                                       filters=1,
36                                       kernel_size=3,
37                                       strides=1)
38
39      self.relu6 = keras.layers.ReLU()
```

```
19      self.var1802 = torch.nn.ReLU()
20      self.many_2 = lambda x: x.view(10, 7, 7, 32)
21
22      self.many_3 = lambda x: x.permute(0, 3, 1, 2)
23      self.var5089 = torch.nn.ConvTranspose2d(32, 64,
24                                              (3,3),
25                                              stride=(2, 2),
26                                              padding=(0, 0))
27      self.var5090 = torch.nn.ReLU()
28      self.many_4 = lambda x: x.permute(0, 1, 3, 2)
29      self.var7498 = torch.nn.ConvTranspose2d(64, 32,
30                                              (3, 3),
31                                              stride=(2, 2),
32                                              padding=(0, 0))
33      self.var7499 = torch.nn.ReLU()
34      self.many_5 = lambda x: x.permute(0, 1, 3, 2)
35      self.var8396 = torch.nn.ConvTranspose2d(32, 1,
36                                              (3, 3),
37                                              stride=(1, 1),
38                                              padding=(0, 0))
39      self.var8397 = torch.nn.ReLU()
```

LIMITATIONS

- Automatic Testing - Decomposability
  Intermediate outputs must be tested independently

- Error message understanding model is library specific
  Different libraries have different stylings for error presentation

- Synthesis doesn’t support many-to-many mappings
  SOAR only supports one-to-one and one-to-many mappings

Final Remarks

- Novel approach for automatic API refactoring:
  - Requires no training data
  - Leverages API documentation and error messages
  - Guarantees that migrated code passes all test cases
  - Language agnostic

- All code is open-source: https://github.com/danieltrt/SOAR/
- Replication package: https://zenodo.org/record/4452730

KEY TAKEAWAY

Documentation can help automating API refactoring tasks

- All code is open-source: https://github.com/danieltrt/SOAR/
- Replication package: https://zenodo.org/record/4452730

KEY TAKEAWAY

Documentation can help automating API refactoring tasks

- All code is open-source: https://github.com/danieltrt/SOAR/
- Replication package: https://zenodo.org/record/4452730