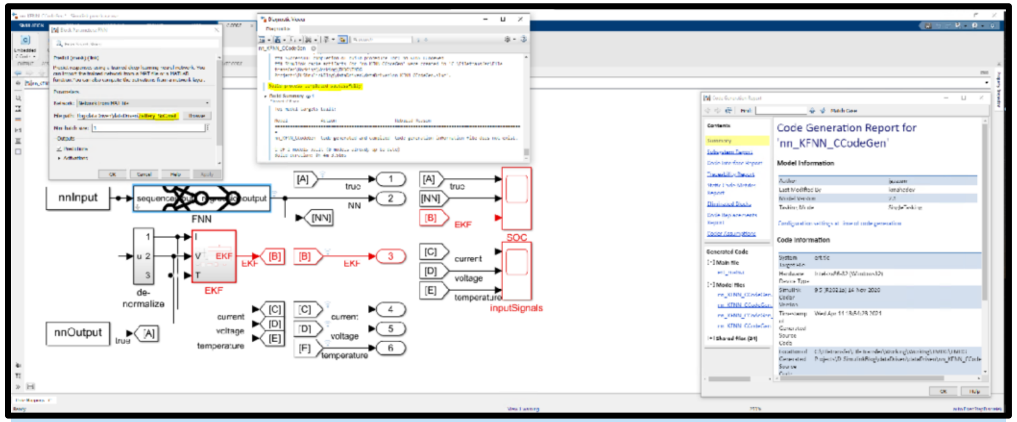
MATLAB EXPO

딥러닝을 위해 MATLAB과 TensorFlow/PyTorch 함께 사용하기

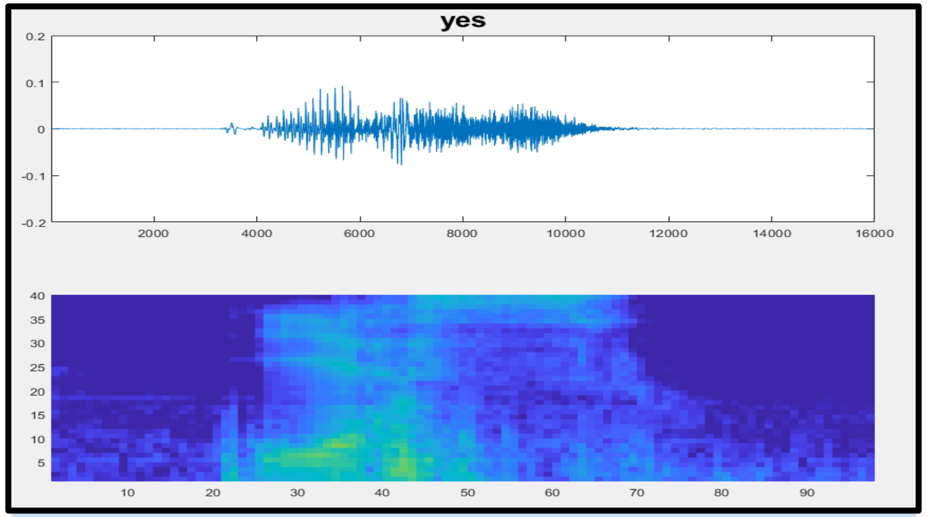
김종남 부장, 매스웍스코리아

MathWorks®

Interoperability has an impact across different vertical applications



Controls/ MBD workflows: models imported from OSS are a part of a bigger system

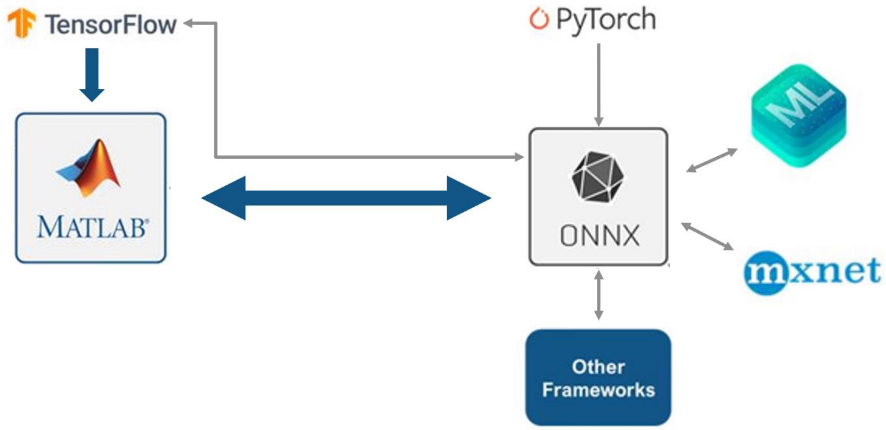


Audio/ Signal Processing: (call dataprocessing in MATLAB from Python)

Ways to Interoperate with TensorFlow and PyTorch

TensorFlow Converter | **ONNX Converter**

Model Exchange



Python Interface | **MATLAB Engine**

MATLAB-Python Coexecution

Model Import Workflow

Model Exchange

ONNX or TensorFlow Model

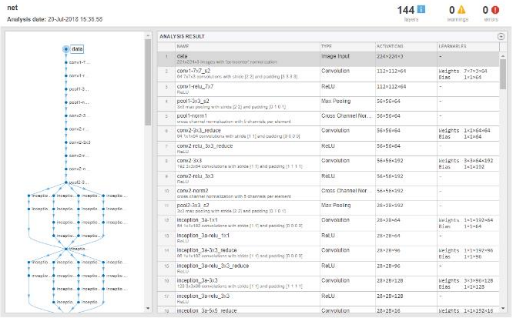
importONNXNetwok/importTensorFlowNetwork

MATLAB Neural Network Model

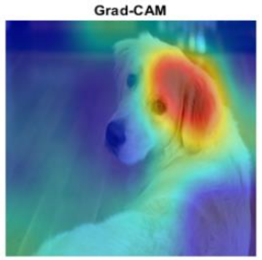
(Automatic) Labeling → **Analyze Network/ Retrain** → **Visualization/ Debugging** → **Code Generation and Compiler**

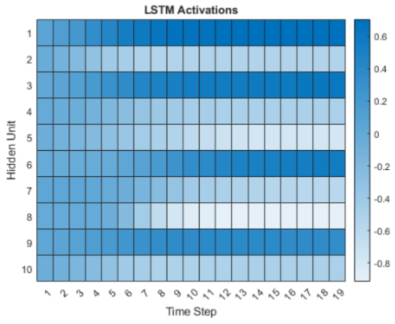
AI System Design

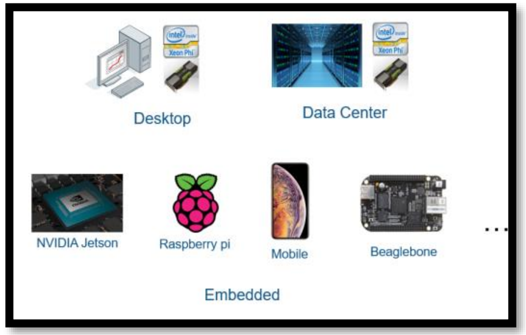
- Image Labeling
- Video Labeling
- Signal/ Audio Labeling
- LiDar Point Cloud Labeling



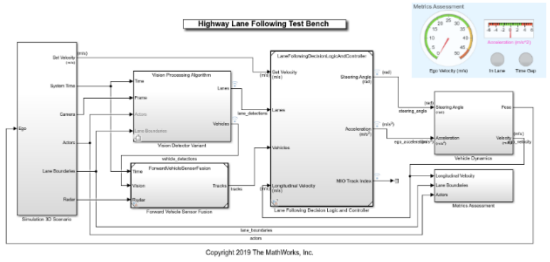
- Automatic Differentiation
- Custom Training Loop
- Weight Sharing







- Standalone Application
- Java
- .NET
- MATLAB Production Server



- Reinforcement Learning
- Automated Driving
- Control Systems

MATLAB-Python Co-execution Workflows

Coexecution

1
- Data Preparation (MATLAB)
- AI Modeling (PyTorch, TensorFlow)
- → Importers → MATLAB
- → Codegen & Compiler

2
- Existing Data Processing Pipelines (PyTorch, TensorFlow)
- AI Modeling (MATLAB)
- → Codegen & Compiler

3
- System Integration (Simulink)
- AI Modeling (PyTorch, TensorFlow)

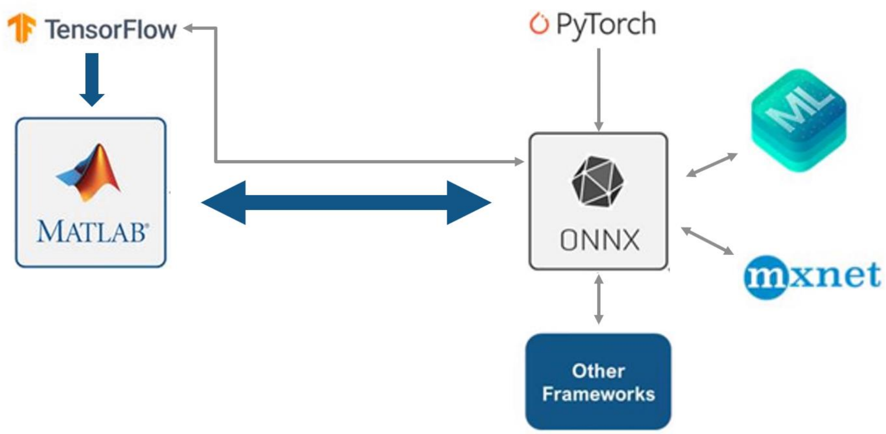
Model Exchange and Complete AI Workflow

Ways to Interoperate with TensorFlow and PyTorch - TensorFlow Converter

TensorFlow Converter

ONNX Converter

Model Exchange

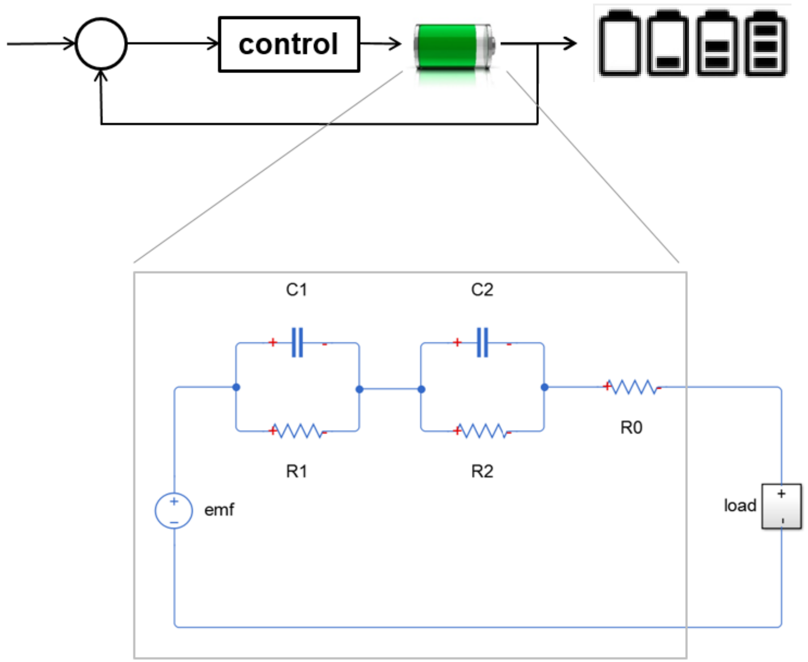


Python Interface

MATLAB Engine

MATLAB-Python Coexecution

Case 1- Example: Battery Management Demo



Predictors

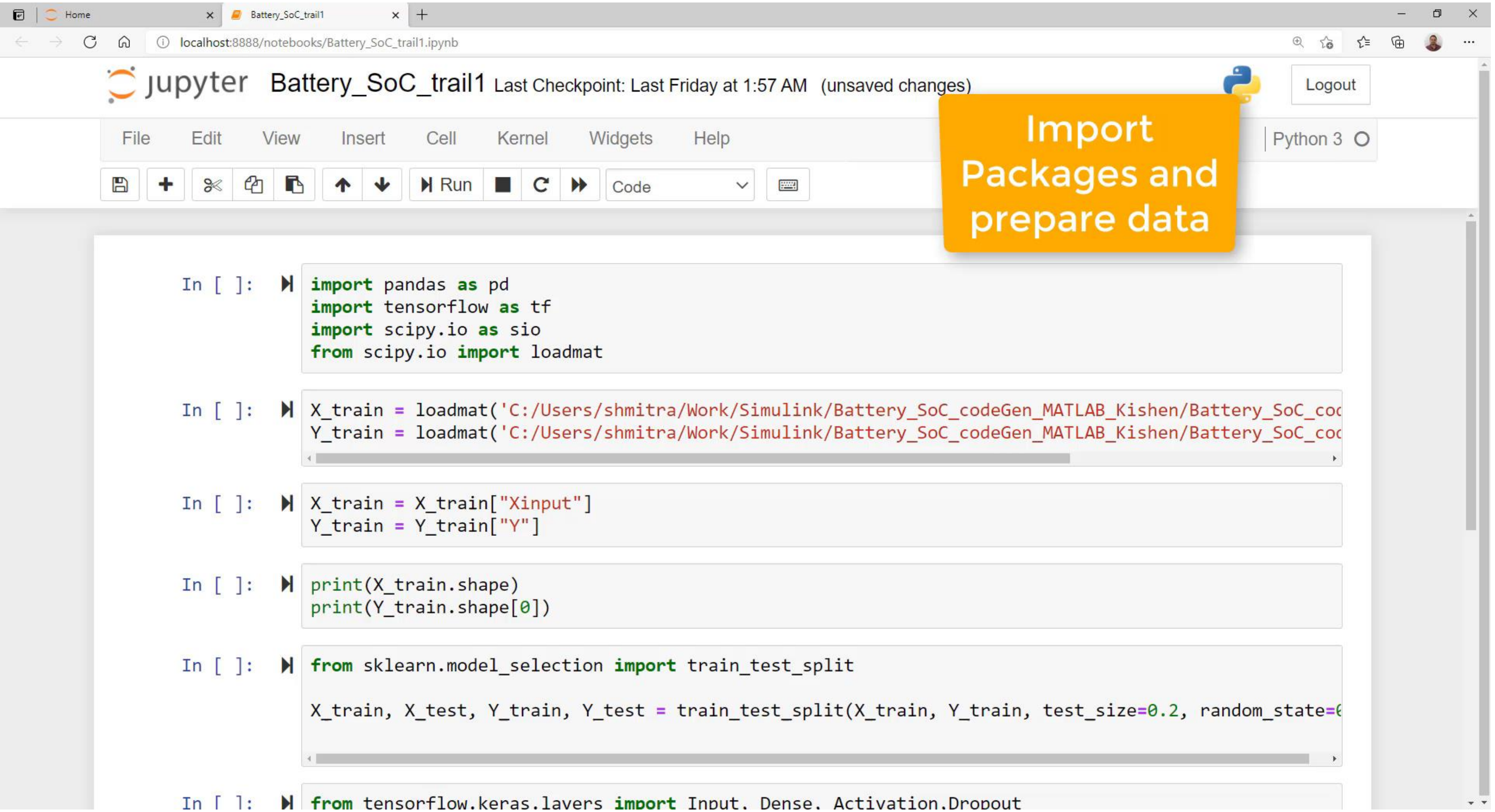
Response

669956x6 table

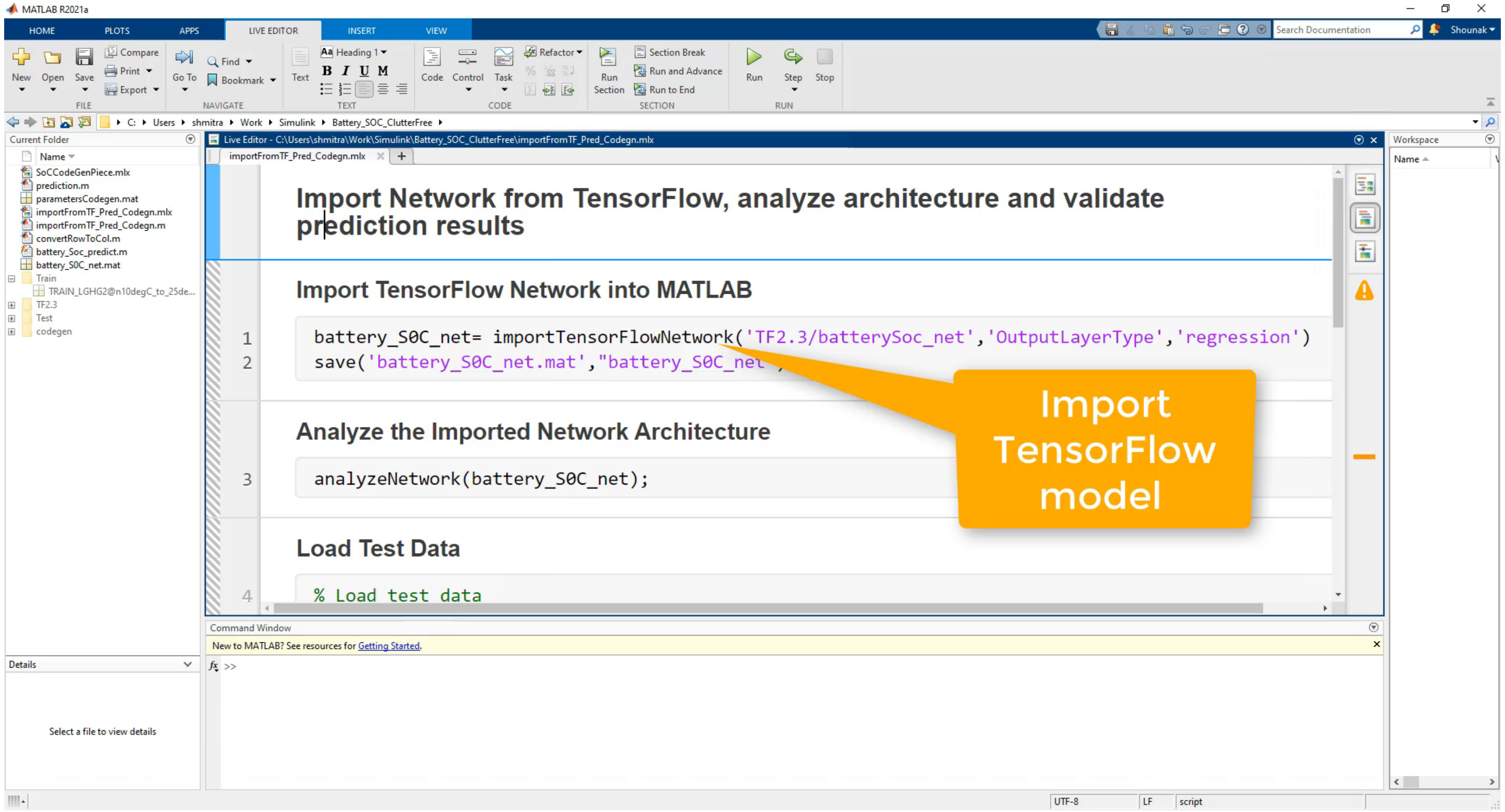
	1 Current	2 Voltage	3 Temperature	4 Moving average Current	5 Moving average temperature	6 battery_SOC
1	0.3851	0.7510	0.3031	0.3851	0.7510	0.2064
2	0.3852	0.7510	0.3046	0.3851	0.7510	0.2064
3	0.3852	0.7510	0.3061	0.3852	0.7510	0.2064
4	0.3852	0.7510	0.3076	0.3852	0.7510	0.2064
5	0.3852	0.7510	0.3091	0.3852	0.7510	0.2064
6	0.3852	0.7510	0.3106	0.3852	0.7510	0.2064
7	0.3852	0.7510	0.3120	0.3852	0.7510	0.2064
8	0.3852	0.7510	0.3135	0.3852	0.7510	0.2064
9	0.3852	0.7510	0.3150	0.3852	0.7510	0.2064

- Step1: Train model in TensorFlow and save the model
- Step2: Import model into MATLAB and analyze architecture and validate the results
- Step3: Include into a Simulink model for desktop simulation
- Step4: Generate CUDA code from imported TensorFlow Model

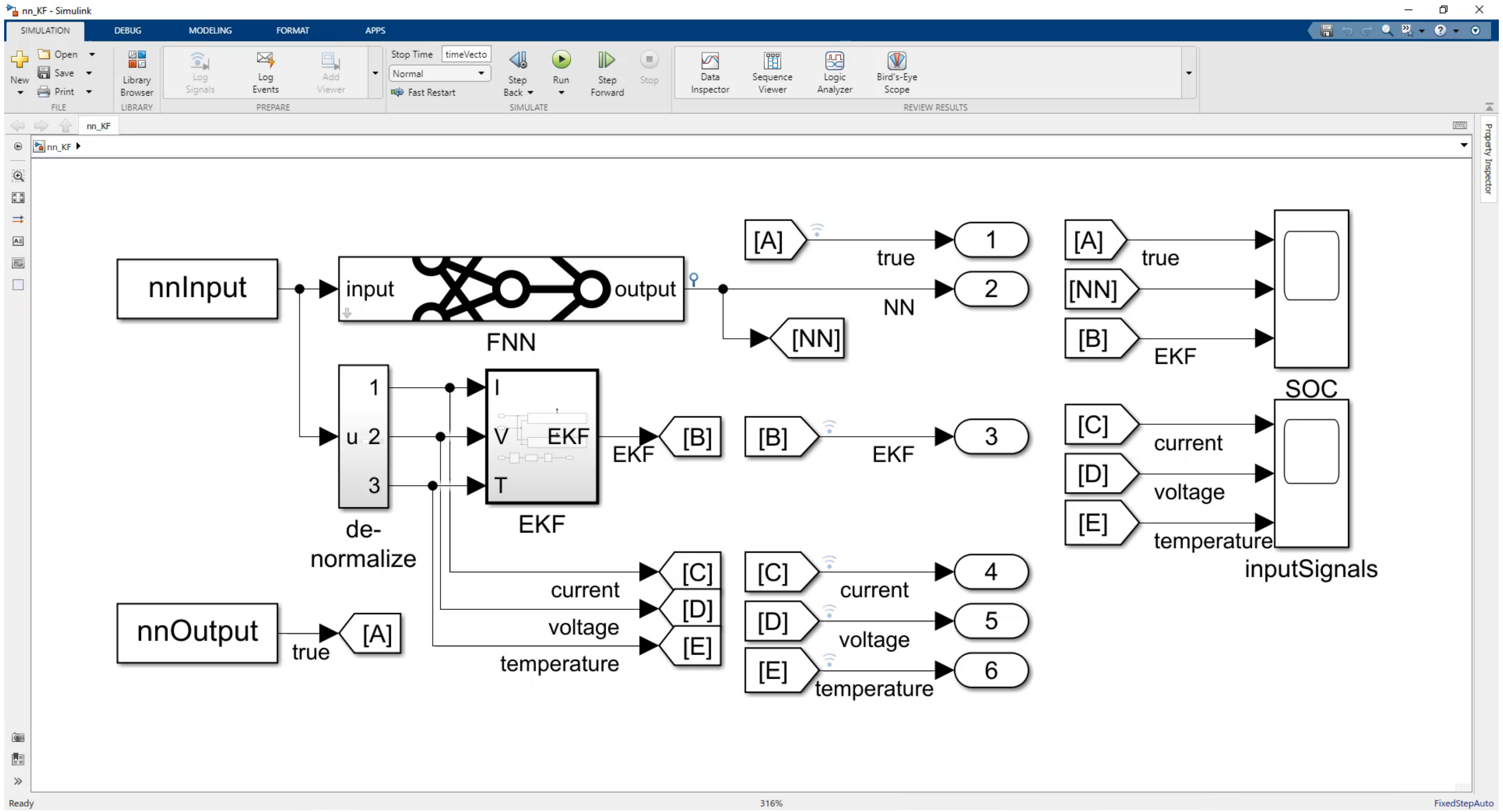
Step1: Train model in TensorFlow and save the model



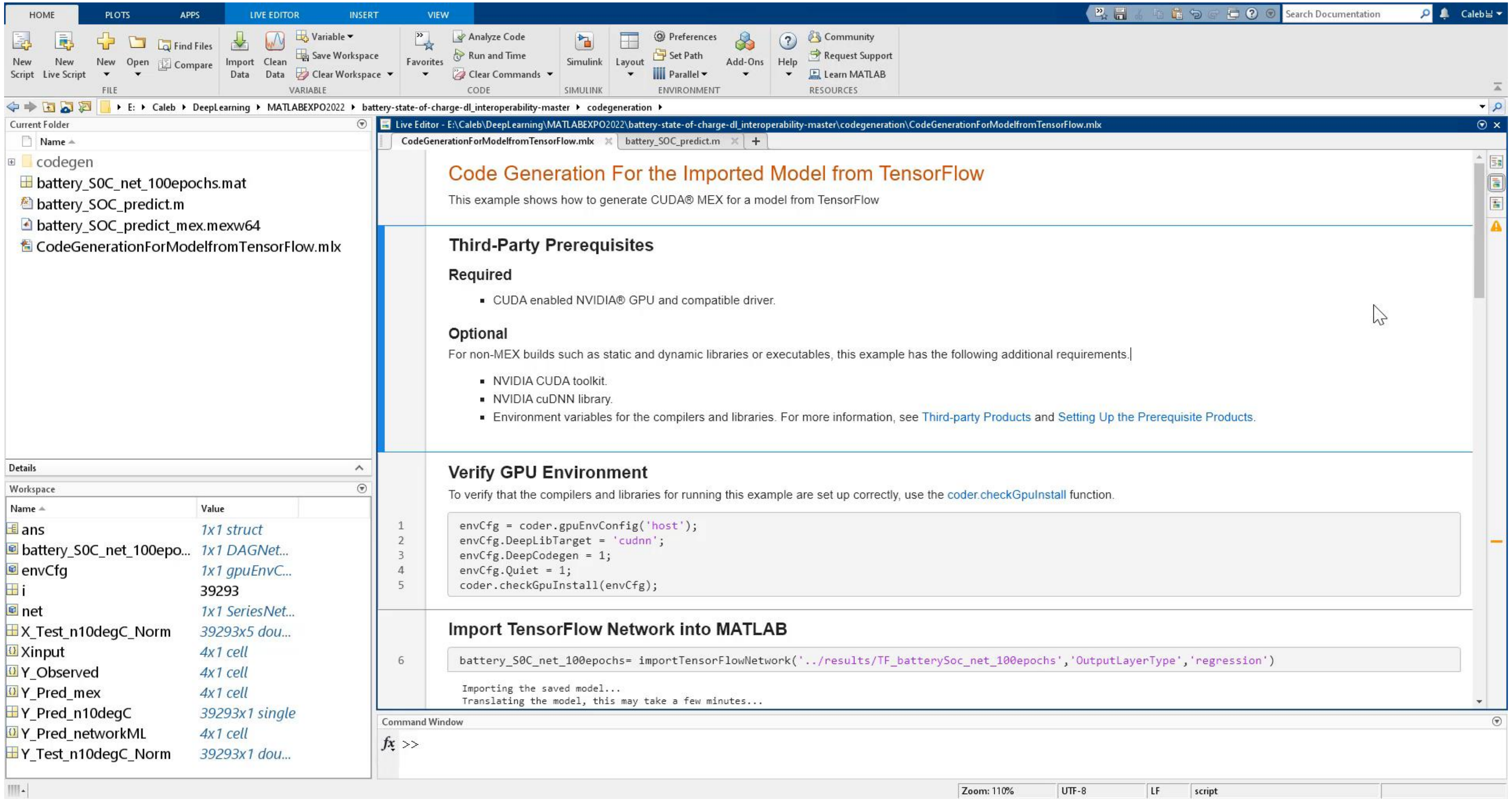
Step 2: Import and analyze architecture, and validate the results



Step3: Include into a Simulink model for desktop simulation



Step4: Generate CUDA code from imported TensorFlow Model



Mitsui Chemicals Deploys AI and Automation Systems with TensorFlow and MATLAB

Challenge

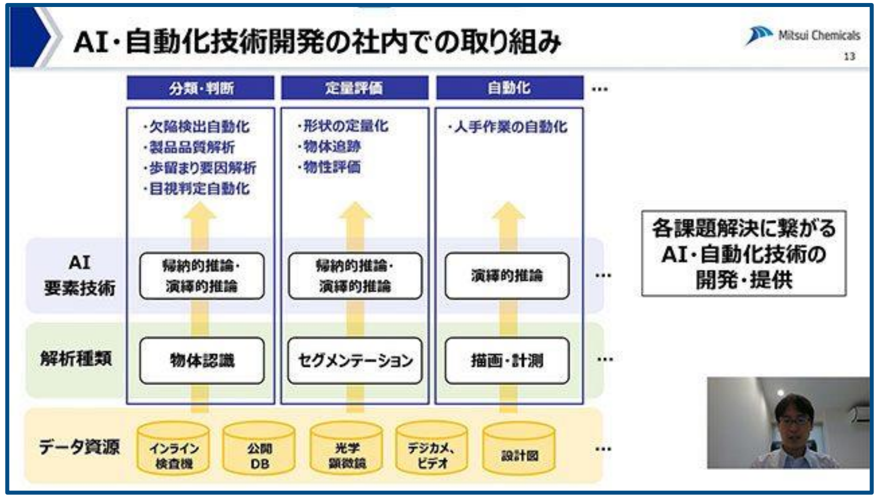
Automate visual inspection of sheet-shaped products and ensure ease of use and maintenance of the deployed model

Solution

Import the trained TensorFlow-Keras model into MATLAB using an importer, create a user interface, and deploy it in the field as an application

Key Outcomes

- Reduced visual inspection time by 80%
- Effectively used models trained in other frameworks
- Deployed application with a user interface that anyone can use



Model development with Python (TensorFlow-Keras) and efficient onsite implementation of models with MATLAB.

> *"MATLAB solved our problems on the field implementation and saved development time. That led to highly accurate development."*
>
> *- Shintaro Maekawa, Mitsui Chemicals, Inc.*

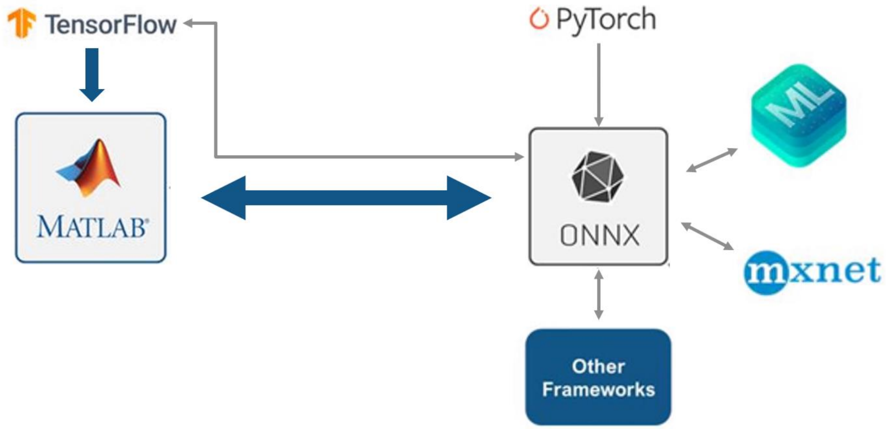
[Link to case study]

Ways to Interoperate with TensorFlow and PyTorch

TensorFlow Converter

ONNX Converter

Model Exchange

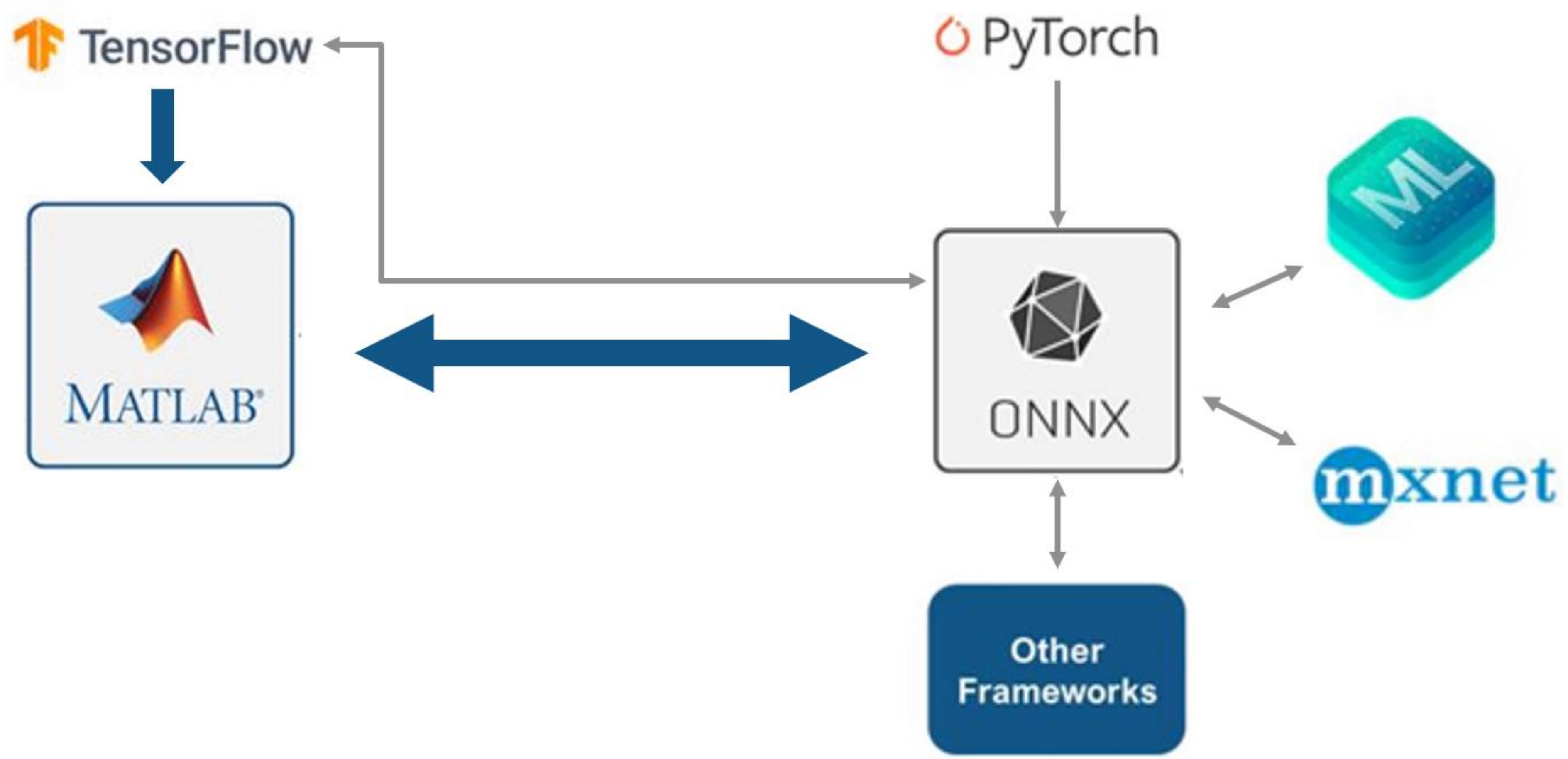


Python Interface

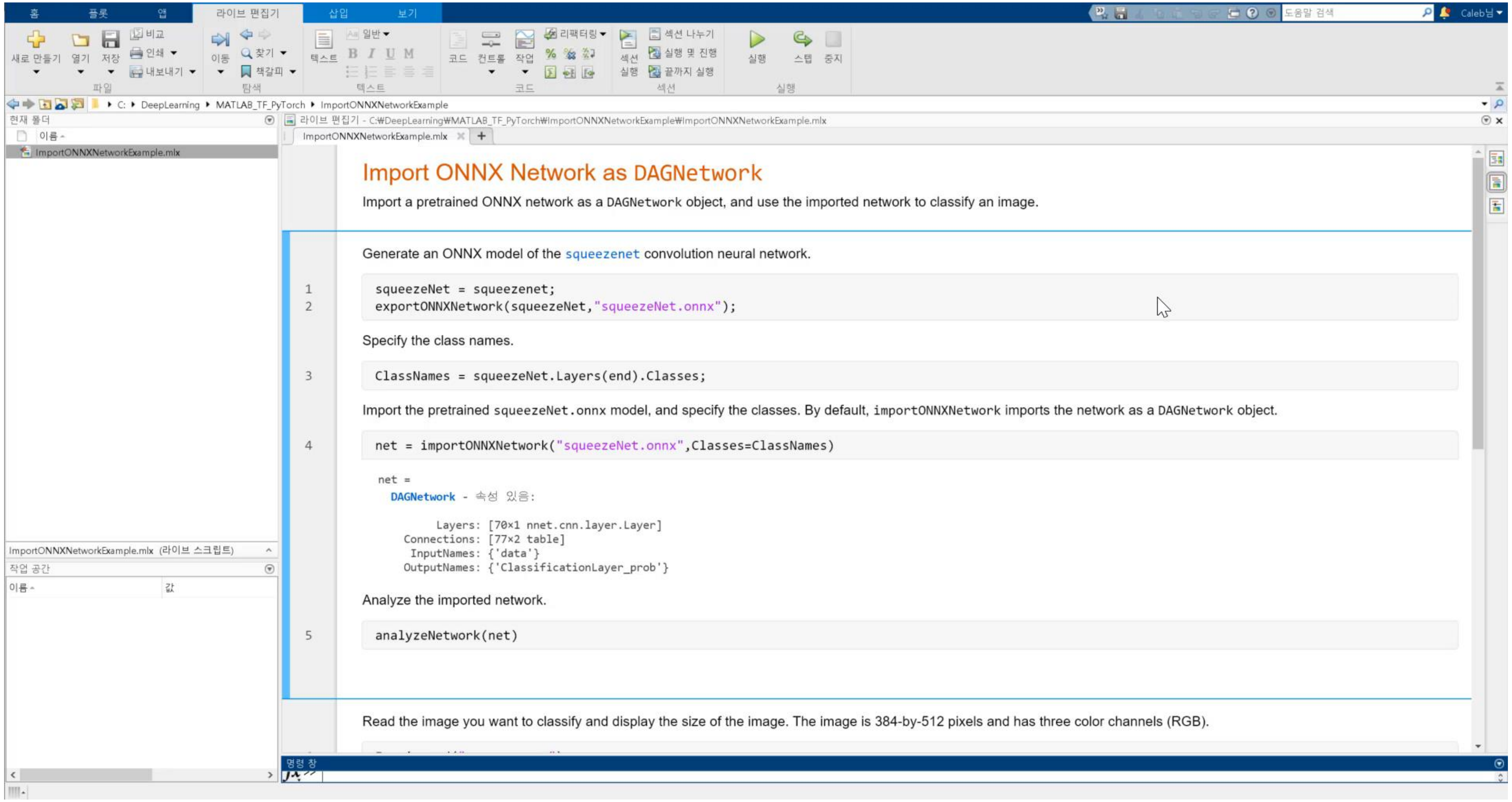
MATLAB Engine

MATLAB-Python Coexecution

Ways to Interoperate with TensorFlow and PyTorch



ONNX Exporting and Importing



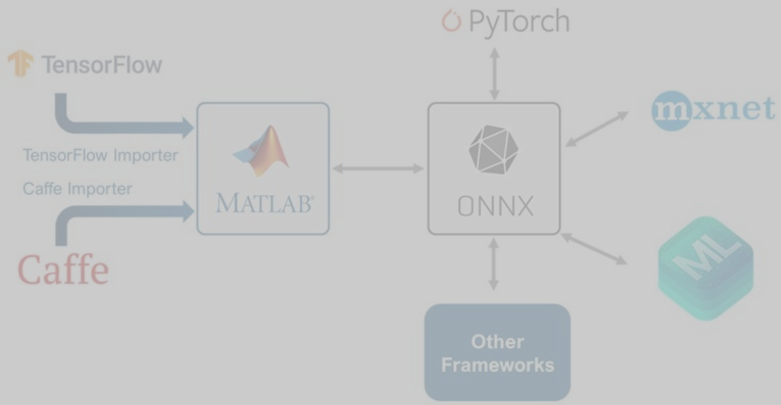
Use MATLAB and Python in model training

Ways to Interoperate with TensorFlow and PyTorch

TensorFlow Converter

ONNX Converter

Model Exchange



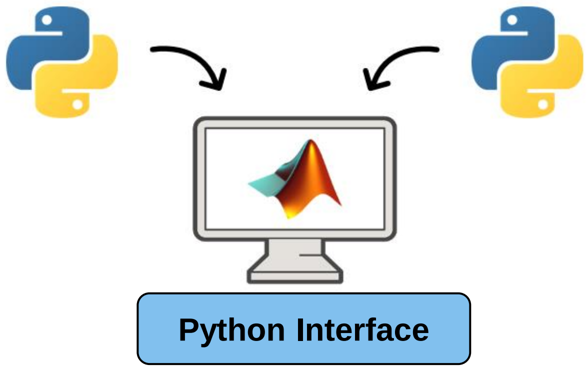
Python Interface

MATLAB Engine

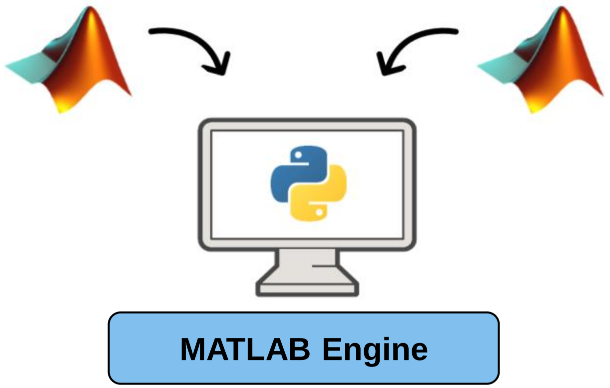
MATLAB-Python Coexecution

Why Co-execution?

Calling Python from MATLAB



Calling MATLAB from Python



Already working in MATLAB, and:
- Want to reuse existing Python code
- Need functionality that is only available in Python

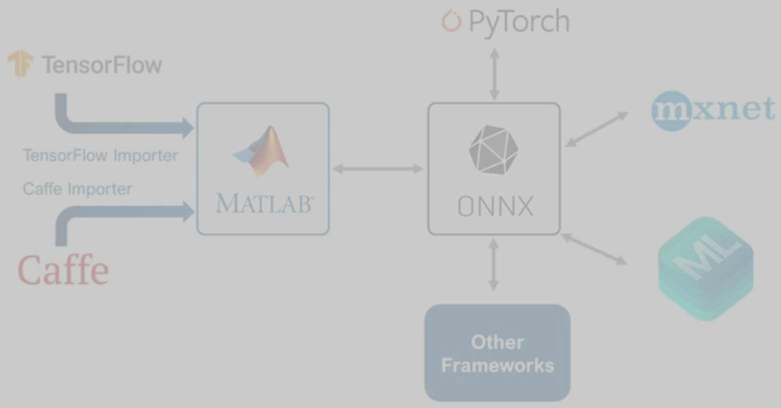
Already working in Python, and:
- Want to reuse existing MATLAB code
- Need functionality available in MATLAB
- Want to collaborate with MATLAB users

Ways to Interoperate with TensorFlow and PyTorch

TensorFlow Converter

ONNX Converter

Model Exchange

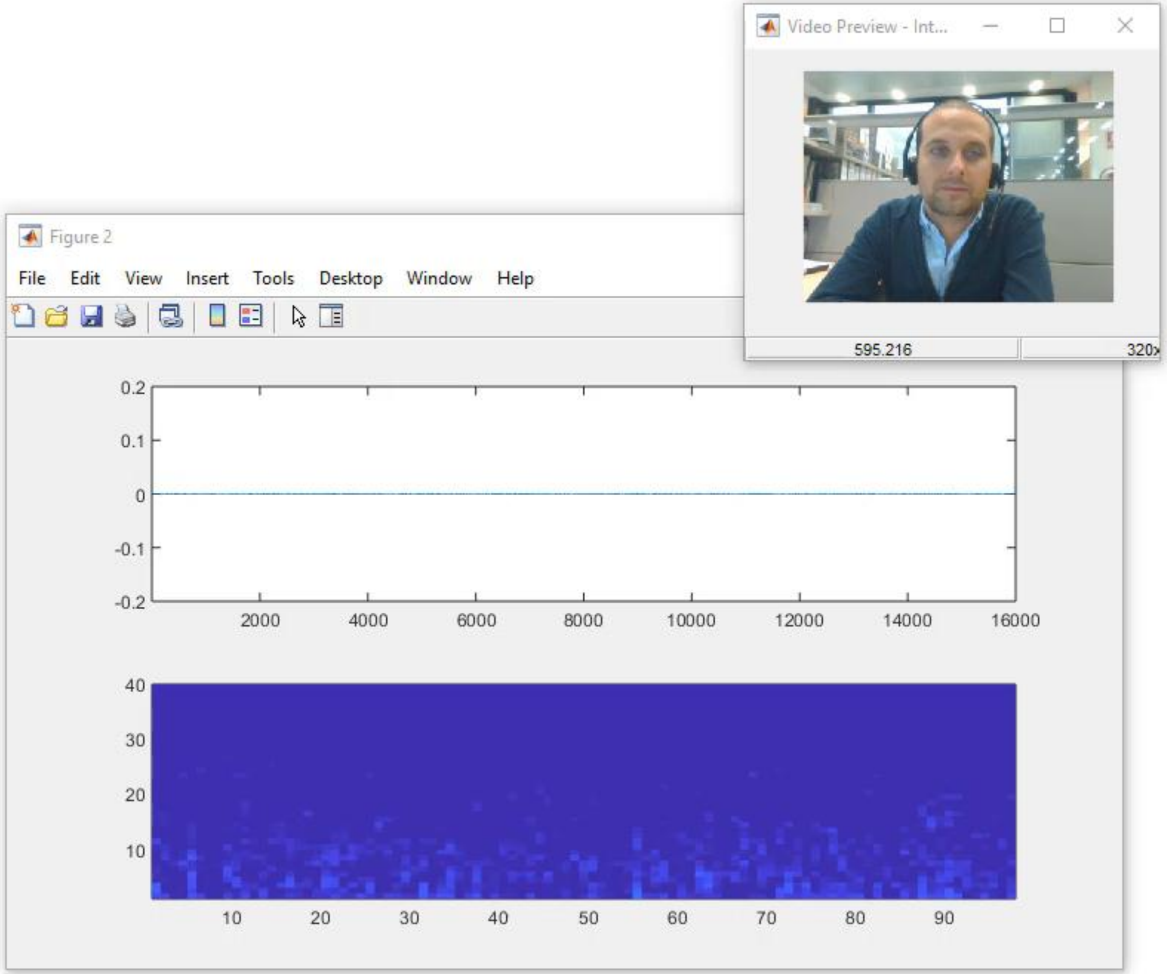


Python Interface

MATLAB Engine

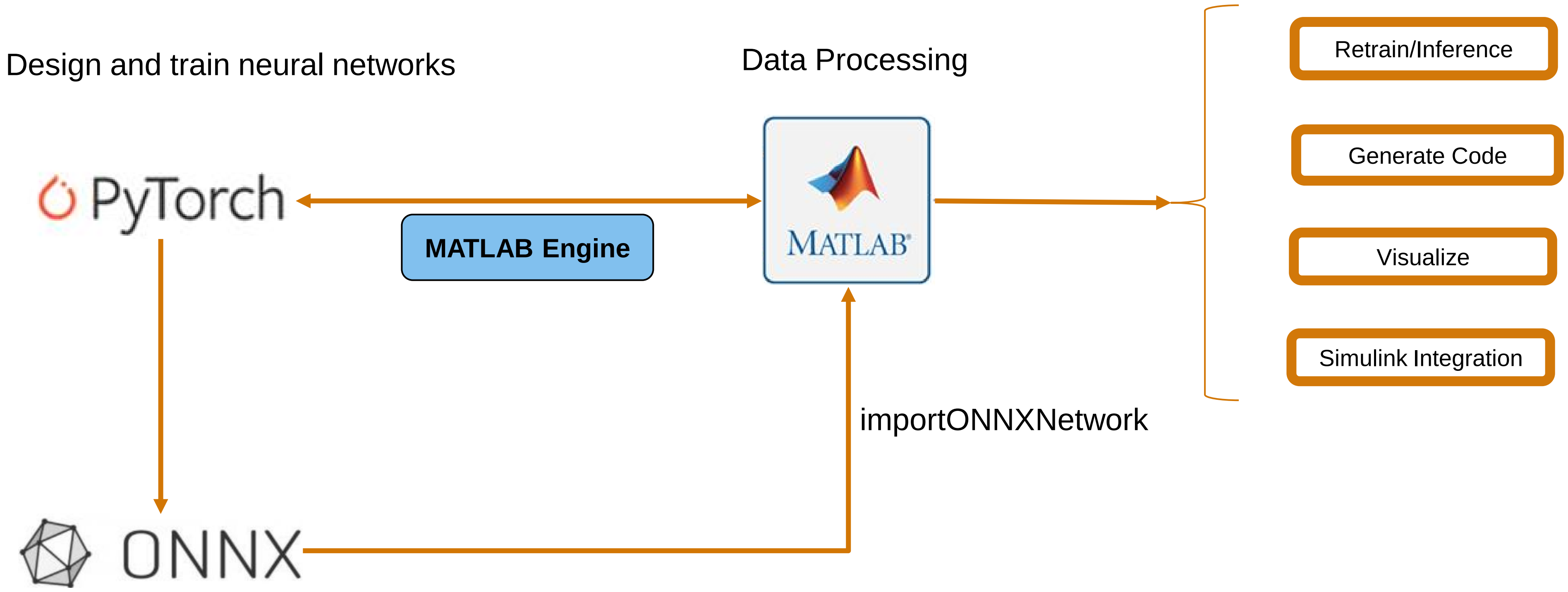
MATLAB-Python Coexecution

Example: Speech Command Recognition – Train in PyTorch, call data processing in MATLAB

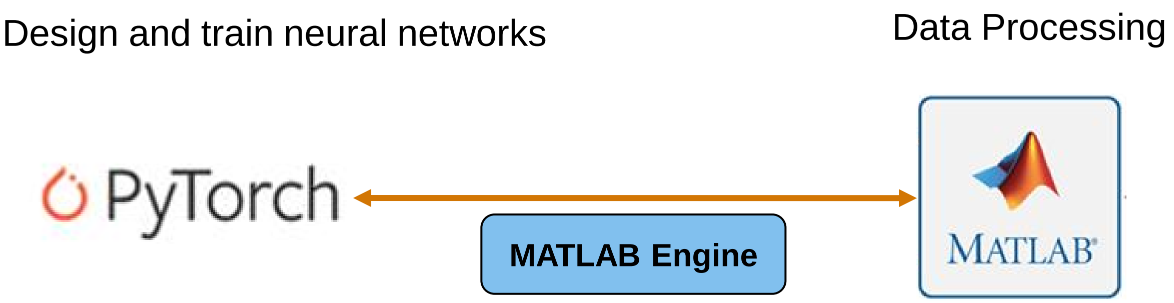


- Step 1: Setting up MATLAB engine in Python
- Step 2: Setting up functions to call MATLAB from PyTorch
- Step 3: Preparing data and designing network in PyTorch
- Step 4: Calling MATLAB preprocessing functions from PyTorch training loop
- Step 5: Exporting trained network to ONNX and import ONNX model in MATLAB

Example Workflow



Example Workflow



Step 1: Setting up MATLAB engine in Python

Install the Engine API

At the MATLAB command prompt —

```
cd (fullfile(matlabroot,'extern','engines','python'))
system('python setup.py install')
```

- [Calling MATLAB from Python]

Start MATLAB Engine

Start Python, import the module, and start the MATLAB engine:

```
import matlab.engine
eng = matlab.engine.start_matlab()
```

Step 2: Setting up functions to call MATLAB from PyTorch

MATLAB Engine

```python
import torch
from torch.utils.data import Dataset, DataLoader
import torch.nn as nn
import torch.onnx

import time
import os

cuda = torch.device('cuda')

# start a MATLAB engine
import matlab.engine
MLEngine = matlab.engine.start_matlab()

miniBatchSize = 128.0

# Prepare training dataset
class TrainData(Dataset):
    def __init__(self):
        # Create persistent training dataset in MATLAB
        MLEngine.setupDatasets(miniBatchSize)
        # Set the dataset length to the number of minibatches
        # in the training dataset
        self.len = int(MLEngine.getNumIterationsPerEpoch())

    def __getitem__(self, index):
        # Call MATLAB to get a minibatch of features + labels
        minibatch = MLEngine.extractTrainingFeatures()
        x = torch.FloatTensor(minibatch.get('features'))
        y = torch.FloatTensor(minibatch.get('labels'))
        return x, y
```

```matlab
function [ads, batchSize] = setupDatasets(varargin)

persistent adsTrain  miniBatchSize
if isempty(adsTrain)
    adsTrain = audioDatastore(datafolder, ...
        'IncludeSubfolders',true, ...
        'LabelSource','foldernames');

    if nargin == 0
        miniBatchSize = 128;
    else
        miniBatchSize = varargin{1};
    end
end

ads = adsTrain;
batchSize = miniBatchSize;
```

Step 2: Setting up functions to call MATLAB from PyTorch

MATLAB Engine

```python
import torch
from torch.utils.data import Dataset, DataLoader
import torch.nn as nn
import torch.onnx

import time
import os

cuda = torch.device('cuda')

# start a MATLAB engine
import matlab.engine
MLEngine = matlab.engine.start_matlab()

miniBatchSize = 128.0

# Prepare training dataset
class TrainData(Dataset):
    def __init__(self):
        # Create persistent training dataset in MATLAB
        MLEngine.setupDatasets(miniBatchSize)
        # Set the dataset length to the number of minibatches
        # in the training dataset
        self.len = int(MLEngine.getNumIterationsPerEpoch())

    def __getitem__(self, index):
        # Call MATLAB to get a minibatch of features + labels
        minibatch = MLEngine.extractTrainingFeatures()
        x = torch.FloatTensor(minibatch.get('features'))
        y = torch.FloatTensor(minibatch.get('labels'))
        return x, y
```

```matlab
function numIterations = getNumIterationsPerEpoch

[trainingDatastore, batchSize] = setupDatasets;
numIterations = numel(trainingDatastore.Files)/batchSize;

end
```

Step 3: Preparing data and designing network in PyTorch

MATLAB Engine

Initiate a handle to prepare the data that will be read in the training loop

```
trainDataset = TrainData()

trainLoader = DataLoader(dataset=trainDataset, batch_size=1)
```

Similar to Datastores in MATLAB

Design the neural network architecture

```
class CNN(nn.Module):

    # Contructor
    def __init__(self, out_1=NumF):
        super(CNN, self).__init__()
        self.cnn1 = nn.Conv2d(in_channels=1, out_channels=out_1, kernel_size=3, padding=1)
        self.batch1 = nn.BatchNorm2d(out_1)
        self.relu1 = nn.ReLU()

        self.maxpool1 = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)

        self.cnn2 = nn.Conv2d(in_channels=out_1, out_channels=2*out_1, kernel_size=3, padding=1)
        self.batch2 = nn.BatchNorm2d(2*out_1)
        self.relu2 = nn.ReLU()

        self.maxpool2 = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)

        self.cnn3 = nn.Conv2d(in_channels=2*out_1, out_channels=4 * out_1, kernel_size=3, padding=1)
        self.batch3 = nn.BatchNorm2d(4 * out_1)
        self.relu3 = nn.ReLU()

        self.maxpool3 = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)

        self.cnn4 = nn.Conv2d(in_channels=4 * out_1, out_channels=4 * out_1, kernel_size=3, padding=1)
        self.batch4 = nn.BatchNorm2d(4 * out_1)
        self.relu4 = nn.ReLU()
        self.cnn5 = nn.Conv2d(in_channels=4 * out_1, out_channels=4 * out_1, kernel_size=3, padding=1)
        self.batch5 = nn.BatchNorm2d(4 * out_1)
        self.relu5 = nn.ReLU()

        self.maxpool4 = nn.MaxPool2d(kernel_size=(timePoolSize, 1))
```

Step 4: Calling MATLAB preprocessing functions from PyTorch training loop

MATLAB Engine

```python
miniBatchSize = 128.0

# Prepare training dataset
class TrainData(Dataset):
    def __init__(self):
        # Create persistent training dataset in MATLAB
        MLEngine.setupDatasets(miniBatchSize)
        # Set the dataset length to the number of minibatches
        # in the training dataset
        self.len = int(MLEngine.getNumIterationsPerEpoch())

    def __getitem__(self, index):
        # Call MATLAB to get a minibatch of features + labels
        minibatch = MLEngine.extractTrainingFeatures()
        x = torch.FloatTensor(minibatch.get('features'))
        y = torch.FloatTensor(minibatch.get('labels'))
        return x, y
```

```matlab
function [ads, batchSize] = setupDatasets(varargin)

persistent adsTrain  miniBatchSize
if isempty(adsTrain)
    adsTrain = audioDatastore(datafolder, ...
        'IncludeSubfolders',true, ...
        'LabelSource','foldernames');

    if nargin == 0
        miniBatchSize = 128;
    else
        miniBatchSize = varargin{1};
    end
end

ads = adsTrain;
batchSize = miniBatchSize;
```

Custom training loop in PyTorch

```python
for epoch in range(n_epochs):

    if epoch == 20:
        for g in optimizer.param_groups:
            g['lr'] = 3e-5

    count = 0
    for batch in trainLoader:
        count += 1
        print('Epoch ', epoch+1, ' Iteration', count, ' of ', trainDataset.len)
        x = batch[0].cuda()
        y = batch[1].cuda()
        optimizer.zero_grad()
        z = model(torch.squeeze(x.float(), 0))
        loss = criterion(z, torch.squeeze(y).long())
        loss.backward()
        optimizer.step()
```

Step 5: Exporting trained network to ONNX and import ONNX model in MATLAB

MATLAB Engine

- In PyTorch, export model to onnx

```
# Export the trained model to ONXX format
torch.onnx.export(model,
                  torch.empty(1, 98, 50).cuda(),
                  "cmdRecognitionPyTorch.onnx",
                  export_params=True,
                  opset_version=9,
                  do_constant_folding=True,
                  input_names=['input'],
                  output_names=['output'])
```

- Import the model to MATLAB with importONNXNetwork

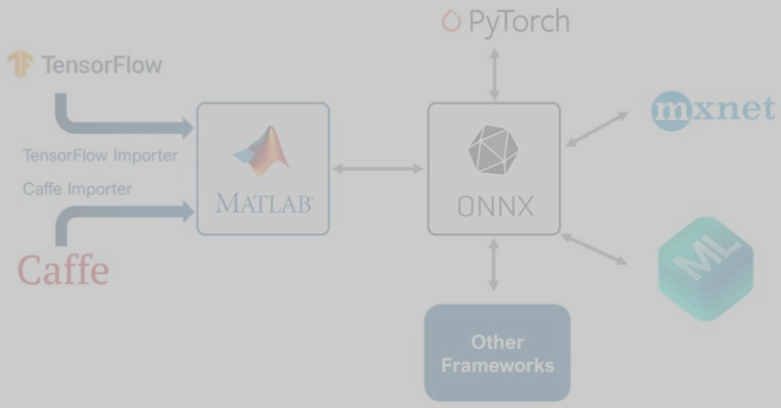
```
>> cmdRecognitionONNX = importONNXNetwork('cmdRecognition.onnx','OutputLayerType','classification')
Warning: Adding a Softmax layer to the imported network. The ONNX network does not include
a Softmax, which is required for classification networks.
> In nnet.internal.cnn.onnx.translateONNX>insertSoftmaxBeforeClassificationLayer (line 351)
In nnet.internal.cnn.onnx.translateONNX>postprocessImportedLayers (line 291)
In nnet.internal.cnn.onnx.translateONNX (line 218)
In nnet.internal.cnn.onnx.importONNXNetwork (line 11)
In importONNXNetwork (line 52)
```

Ways to Interoperate with TensorFlow and PyTorch

TensorFlow Converter

ONNX Converter

Model Exchange



Python Interface

MATLAB Engine

MATLAB-Python Coexecution

Using TensorFlow Network Design Inside the MATLAB Script

Calling Python from MATLAB

In this example, you invoke TensorFlow training from MATLAB. The training loop is in MATLAB. The neural network model and the gradient/loss computations happen in TensorFlow.

Setup Training Datastore

Set up a tranining datastore with the desired minibatch size. This is the same function used in the original demo version (call MATLAB from Python).

```
miniBatchSize = 128;
[trainingDatastore, validationDatastore] = setupDatasets(miniBatchSize);
numIterationPerEpoch = numel(trainingDatastore.Files)/miniBatchSize;
```

Compute Validation Data

Get the validation data (similar to original example).

```
validationData = extractValidationFeatures;
validationData.features = permute(validationData.features, [1 3 4 2]);
```

Instantiate the deep learning model. This is a class defined in Python. You will call methods on this object in the training loop.

```
model = py.SpeechCommandRecognition.SpeechCommandRecognition();
```

Using TensorFlow Network Design Inside the MATLAB Script

Instantiate the deep learning model. This is a class defined in Python. You will call methods on this object in the training loop.

```
model = py.SpeechCommandRecognition.SpeechCommandRecognition();
```

```
class SpeechCommandRecognition(tf.Module):

    def make_model(self):
        # Define the model
        inputs = keras.Input(shape=(98, 50, 1))

        x = layers.Conv2D(12, 3, strides=1, padding='same')(inputs)
        x = layers.BatchNormalization(axis=3)(x)
        x = layers.Activation('relu')(x)

        x = layers.MaxPool2D(pool_size=3, strides=2, padding='same')(x)

        x = layers.Conv2D(2 * 12, 3, strides=1, padding='same')(x)
        x = layers.BatchNormalization(axis=3)(x)
        x = layers.Activation('relu')(x)

        x = layers.MaxPool2D(pool_size=3, strides=2, padding='same')(x)

        x = layers.Conv2D(4 * 12, 3, strides=1, padding='same')(x)
        x = layers.BatchNormalization(axis=3)(x)
        x = layers.Activation('relu')(x)

        x = layers.MaxPool2D(pool_size=3, strides=2, padding='same')(x)

        x = layers.Conv2D(4 * 12, 3, strides=1, padding='same')(x)
        x = layers.BatchNormalization(axis=3)(x)
        x = layers.Activation('relu')(x)
        x = layers.Conv2D(4 * 12, 3, strides=1, padding='same')(x)
        x = layers.BatchNormalization(axis=3)(x)
        x = layers.Activation('relu')(x)

        x = layers.MaxPool2D(pool_size=(13, 1), strides=(1, 1), padding='valid')(x)
```

Training Loop

In the training loop, call the method forward to update the weights.

```
numEpochs = 1;
for epoch = 1:numEpochs
    model.initializeAcc;
    for i = 1:numIterationPerEpoch
        if mod(i,10)==1
            fprintf('Epoch %d - Iteration %d of %d\n',epoch,i,numIterationPerEpoch);
        end
        values = extractTrainingFeatures;
        features = permute(values.features, [1 3 4 2]);
        labels = values.labels.';
        model.forward(features, labels, pyargs('training', true));
    end
    model.printAcc;
    z = model.forward(validationData.features, 0);
    z = double(z);
    [~,m] = max(z,[],2);
    acc = sum((validationData.labels ==   (m-1)))/numel(m);
    fprintf('Validation accuracy: %f percent\n',100 * acc);
end
```

Using TensorFlow Network Design Inside the MATLAB Script

Instantiate the deep learning model. This is a class defined in Python. You will call methods on this object in the training loop.

```
model = py.SpeechCommandRecognition.SpeechCommandRecognition();
```

Training Loop

In the training loop, call the method forward to update the weights.

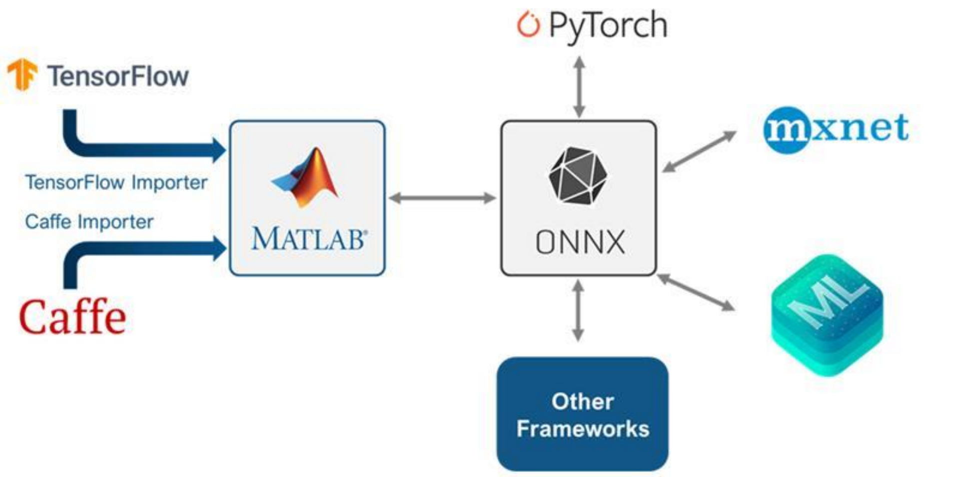
```
numEpochs = 1;
for epoch = 1:numEpochs
    model.initializeAcc;
    for i = 1:numIterationPerEpoch
        if mod(i,10)==1
            fprintf('Epoch %d - Iteration %d of %d\n',epoch,i,numIterationPerEpoch);
        end
        values = extractTrainingFeatures;
        features = permute(values.features, [1 3 4 2]);
        labels = values.labels.';
        model.forward(features, labels, pyargs('training', true));
    end
    model.printAcc;
    z = model.forward(validationData.features, 0);
    z = double(z);
    [~,m] = max(z,[],2);
    acc = sum((validationData.labels ==   (m-1)))/numel(m);
    fprintf('Validation accuracy: %f percent\n',100 * acc);
end
```

```
def initializeAcc(self):
    self.epoch_loss_avg = tf.keras.metrics.Mean()
    self.epoch_accuracy = tf.keras.metrics.SparseCategoricalAccuracy()

def __init__(self):
    super(SpeechCommandRecognition, self).__init__()
    self.model = self.make_model()
    lr = tf.Variable(.0003, trainable=False, dtype=tf.float32)
    self.optimizer = tf.keras.optimizers.Adam(learning_rate=lr)
    self.initializeAcc()

def forward(self, x, y, training=False):

    x = np.expand_dims(x, 3)
    if training:
        with tf.GradientTape() as tape:
            z = self.model(x)
            loss_value = self.loss(y, z)
            grads = tape.gradient(loss_value, self.model.trainable_variables)

            self.optimizer.apply_gradients(zip(grads, self.model.trainable_variables))

            # Track progress
            self.epoch_loss_avg(loss_value)  # Add current batch loss
            # Compare predicted label to actual label
            self.epoch_accuracy(y, self.model(x))
    else:
        return self.model(x)
```

Summary: MATLAB with TensorFlow & PyTorch

Model Exchange

Used when working mainly with Deep Learning models
(R2017b or later)



Best for AI model evolution, codegen, and system integration

- Import model from third-party framework to Deep Learning Toolbox
- Use MATLAB's data labeling/ processing/ code generation and compiler pipelines
- Integrate model into Simulink using Deep Learning Toolbox blocks or MATLAB Function block
- Export modified model to third-party framework if needed

Co-execution

Used when

- working with Deep Learning models or other Matlab/ Python code
- pretrained models cannot be directly imported into MATLAB

Best for encapsulation and reuse of Python code in MATLAB/ Simulink

- Use existing data pipelines in Python and train and perform experiment management in MATLAB using apps
- Use TensorFlow/ PyTorch for training with MATLAB's data labeling/ processing pipelines
- Create Python API in separate MATLAB function in Simulink
- Use a MATLAB Function block in Simulink to call Python subroutines and models

MATLAB EXPO

Thank you

MathWorks®