Outline

● Extended Kalman Filter

● A brief Introduction to Tensorflow
Outline

● Extended Kalman Filter

● A brief Introduction to Tensorflow
Dynamical System

Control Input

State

Observation

\( u(t - 1) \)

\( x(t - 1) \)

\( u(t) \)

\( x(t) \)

\( u(t + 1) \)

\( x(t + 1) \)

\( z(t - 1) \)

\( z(t) \)

\( z(t + 1) \)
Kalman Filter

An algorithm that uses a series of measurements observed over time, containing statistical noise and other inaccuracies, and produces estimates of unknown variables that tend to be more accurate than those based on a single measurement alone, by estimating a joint probability distribution over the variables for each timeframe.

Kalman Filter

An algorithm that uses a series of *measurements observed over time*, containing *statistical noise* and other inaccuracies, and produces *estimates of unknown variables* that tend to be more accurate than those based on a single measurement alone, by estimating a joint probability distribution over the variables for each timeframe.

*Source: Wikipedia*

To make it even more illustrative ->
What does Kalman Filter do?

Control Input:
- $u(t - 1)$
- $u(t)$
- $u(t + 1)$

State:
- $x(t - 1)$
- $x(t)$
- $x(t + 1)$

Observation:
- $z(t - 1)$
- $z(t)$
- $z(t + 1)$
What does Kalman Filter do?
What does Kalman Filter do?

Control Input

State

Observation

Known Distributions

Unknown Distributions

Known Values
Extended Kalman Filter

- Extended Kalman filter (EKF) is heuristic for nonlinear filtering problem.
- Often works well (when tuned properly), but sometimes not.
- Widely used in practice.

Based on
- Linearizing dynamics and output functions at current estimate.
- Propagating an approximation of the conditional expectation and covariance.

Source: EE363
Extended Kalman Filter

- Extended Kalman filter (EKF) is heuristic for **nonlinear** filtering problem.
- Often works well (when tuned properly), but sometimes not.
- Widely used in practice.

Based on
- Linearizing dynamics and output functions at current estimate.
- Propagating an approximation of the conditional expectation and covariance.

Source: EE363
Implementing Extended Kalman Filter

- Define the state, the control, and the noise
- Derive the system and the observation
- Compute the current Jacobian matrix (*linearizing dynamics*)
- Compute the distribution of the current state
- Iterate this process across time
Define the State

State: 6-dimensional vector (position, velocity)
Define the System Matrix

Control Input
- $u(t-1)$
- $u(t)$
- $u(t+1)$

State
- $x(t-1)$
- $x(t)$
- $x(t+1)$

Observation
- $z(t-1)$
- $z(t)$
- $z(t+1)$

Known Distributions
Unknown Distributions
Define the System Matrix

\[ x_t = \begin{bmatrix} p_t^x \\ p_t^y \\ p_t^z \\ v_t^x \\ v_t^y \\ v_t^z \end{bmatrix} \]

\[ x_{t+1} = Ax_t + \epsilon_t \]
Define the Observation

Control Input

\[ u(t - 1) \]

\[ u(t) \]

\[ u(t + 1) \]

State

\[ x(t - 1) \]

\[ x(t) \]

\[ x(t + 1) \]

Observation

\[ z(t - 1) \]

\[ z(t) \]

\[ z(t + 1) \]
Define the Observation

\[ z_t = h(x_t) + v_t \]

Observation in Q1: 2-dimensional vector (pixel location)
Observation in Q2: 3-dimensional vector (pixel location, disparity)

\( h(x_t) \) can be derived using the camera model we learned from previous lectures.
Computing the Jacobian

\[ J = \begin{bmatrix} \frac{\partial f}{\partial x_1} & \cdots & \frac{\partial f}{\partial x_n} \\ \nabla^T f_1 \\ \vdots \\ \nabla^T f_m \end{bmatrix} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix} \]

Outline

- Extended Kalman Filter
- A brief Introduction to Tensorflow
Tensorflow v.s. PyTorch
```python
input = tf.placeholder(shape=[None, 480, 640, 3], dtype=tf.float32)
label = tf.placeholder(shape=[None, 3], dtype=tf.float32)
predict = make_model(input)
loss, optimizer = make_optimizer(predict, label)
saver = tf.train.Saver()

batch_size = 16
num_epochs = 15

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    train_losses = []
    test_losses = []

    for i in range(num_epochs), desc='Training':
        train_index = np.random.permutation(train.shape[0])
        current = 0

        losses = []
        while current < train.shape[0]:
            batch_image_train = image_train[train_index[current:current+batch_size, label_train.shape[0]]]
            batch_label_train = label_train[train_index[current:current+batch_size, label_train.shape[0]]]
            loss_val, _ = sess.run([loss, optimizer], feed_dict={input: batch_image_train, label: batch_label_train})
            losses.append(loss_val)
            current = current + batch_size

        train_losses.append(np.mean(losses))

        test_index = np.random.permutation(test.shape[0])
        current = 0

        losses = []
        while current < test.shape[0]:
            batch_image_test = image_test[test_index[current:current+batch_size, label_test.shape[0]]]
            batch_label_test = label_test[test_index[current:current+batch_size, label_test.shape[0]]]
            loss_val = sess.run(loss, feed_dict={input: batch_image_test, label: batch_label_test})
            losses.append(loss_val)
            current = current + batch_size

        test_losses.append(np.mean(losses))

    saver.save(sess, 'trained_model')

    if I = 1:
        clear_output()
        fig, (ax1, ax2) = plt.subplots(1, 2)
        ax1.plot(train_losses)
        ax2.plot(test_losses)
        ax1.set_xlabel('Epoch')
        ax1.set_ylabel('Train Loss')
        ax2.set_xlabel('Epoch')
        ax2.set_ylabel('Test Loss')
        plt.show()
        print('Final training loss: ', train_losses[-1])
        print('Final testing loss: ', test_losses[-1])
```
Define the graph

Initialize the session

Run the session
Defining the Graph

```python
input = tf.placeholder(shape=[None, 480, 640, 3], dtype=tf.float32)
label = tf.placeholder(shape=[None, 3], dtype = tf.float32)
predict = make_model(input)
loss, optimizer = make_optimizer(predict, label)
```
Running the Session

```python
input = tf.placeholder(shape=[None, 480, 640, 3], dtype=tf.float32)
label = tf.placeholder(shape=[None, 3], dtype=tf.float32)
predict = make_model(input)
loss, optimizer = make_optimizer(predict, label)

sess = tf.Session()
sess.run(tf.global_variables_initializer())

loss_val = sess.run(loss, feed_dict={input: batch_image_test, label: batch_label_test})
```
Running the Session

```python
input = tf.placeholder(shape=[None, 480, 640, 3], dtype=tf.float32)
label = tf.placeholder(shape=[None, 3], dtype = tf.float32)
predict = make_model(input)
loss, optimizer = make_optimizer(predict, label)

loss_val = sess.run(loss, feed_dict={input: batch_image_test, label: batch_label_test})
```
Neural Network Architecture

```python
def make_model(input):
    conv1_1 = tf.layers.conv2d(input, 32, 3, padding='same')
    conv1_1 = tf.layers.batch_normalization(conv1_1)
    conv1_1 = tf.nn.relu(conv1_1)
    conv1_2 = tf.layers.conv2d(conv1_1, 32, 3, padding='same')
    conv1_2 = tf.layers.batch_normalization(conv1_2)
    conv1_2 = tf.nn.relu(conv1_2)
    pool_1 = tf.nn.max_pool(conv1_2, [1, 2, 2, 1], [1, 2, 2, 1], padding='SAME')

    conv2_1 = tf.layers.conv2d(pool_1, 64, 3, padding='same')
    conv2_1 = tf.layers.batch_normalization(conv2_1)
    conv2_1 = tf.nn.relu(conv2_1)
    conv2_2 = tf.layers.conv2d(conv2_1, 64, 3, padding='same')
    conv2_2 = tf.layers.batch_normalization(conv2_2)
    conv2_2 = tf.nn.relu(conv2_2)
    pool_2 = tf.nn.max_pool(conv2_2, [1, 2, 2, 1], [1, 2, 2, 1], padding='SAME')

    conv3_1 = tf.layers.conv2d(pool_2, 128, 3, padding='same')
    conv3_1 = tf.layers.batch_normalization(conv3_1)
    conv3_1 = tf.nn.relu(conv3_1)
    conv3_2 = tf.layers.conv2d(conv3_1, 128, 3, padding='same')
    conv3_2 = tf.layers.batch_normalization(conv3_2)
    conv3_2 = tf.nn.relu(conv3_2)
    conv3_3 = tf.layers.conv2d(conv3_2, 128, 3, padding='same')

    feature_points = tf.contrib.layers.spatial_softmax(conv3_3)

    fc1 = tf.layers.dense(feature_points, 64)
    fc1 = tf.layers.batch_normalization(fc1)
    fc1 = tf.nn.relu(fc1)
    fc2 = tf.layers.dense(fc1, 64)
    fc2 = tf.layers.batch_normalization(fc2)
    fc2 = tf.nn.relu(fc2)
    fc3 = tf.layers.dense(fc2, 3)
    return fc3
```
def make_optimizer(pred, label):
    loss = tf.reduce_mean(tf.reduce_sum((pred - label) ** 2, axis=-1))
    optimizer = tf.train.AdamOptimizer(learning_rate=3e-4).minimize(loss)
    return loss, optimizer
Outline

- Extended Kalman Filter
- A brief Introduction to Tensorflow