Source localization in an ocean waveguide using supervised machine learning

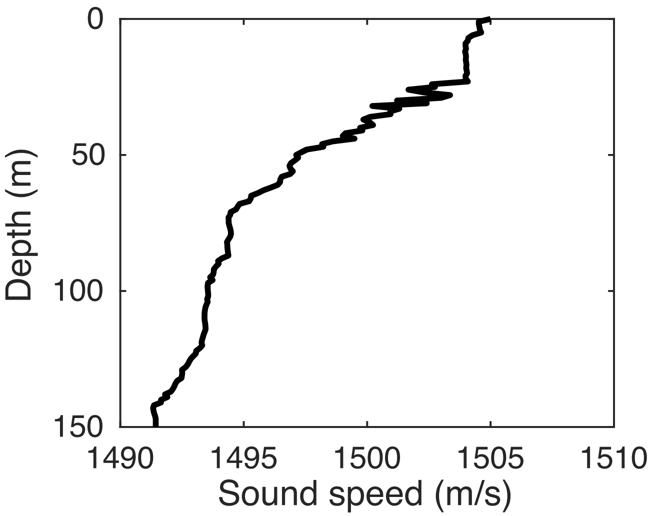
Haiqiang Niu, Emma Reeves, and Peter Gerstoft

Scripps Institution of Oceanography, UC San Diego

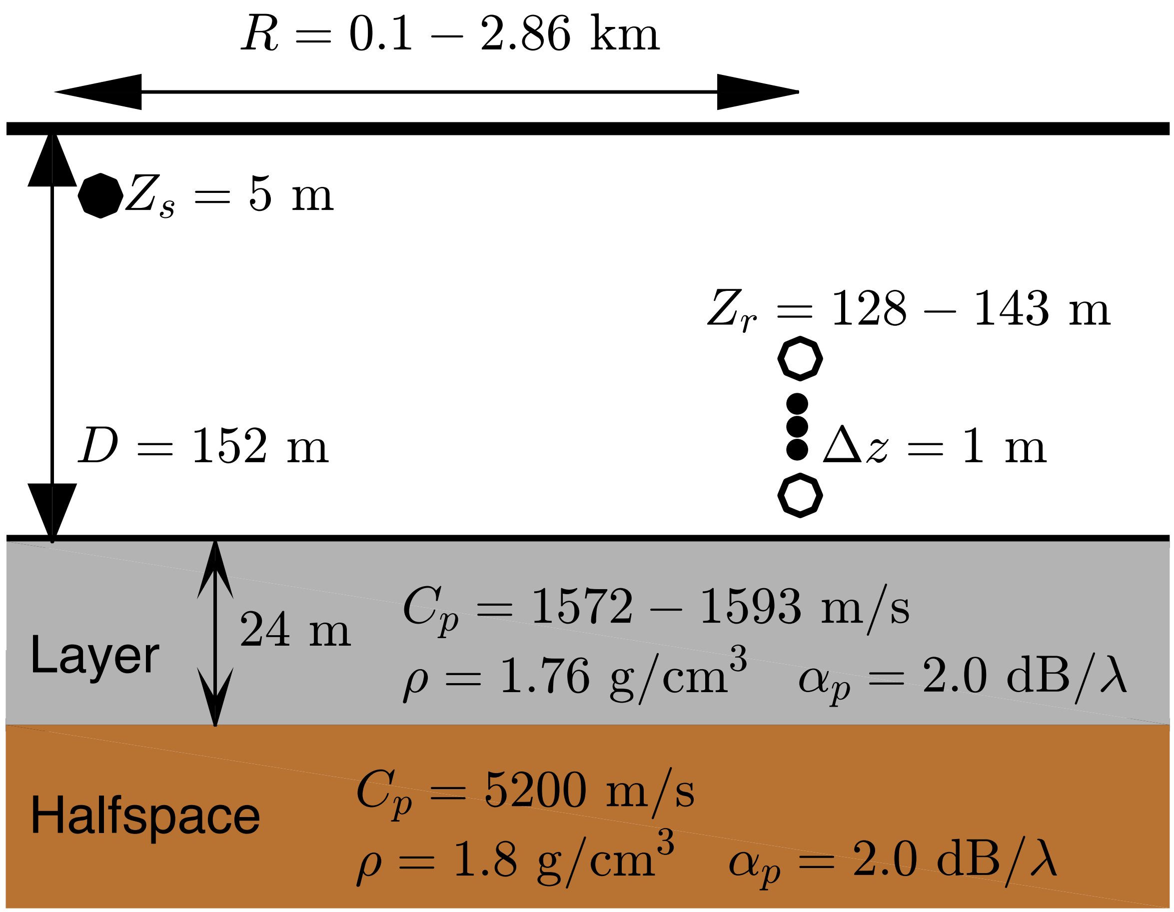
Part I

- Localization on Noise09 data and SBCEx16 data

Noise 09 Experiment

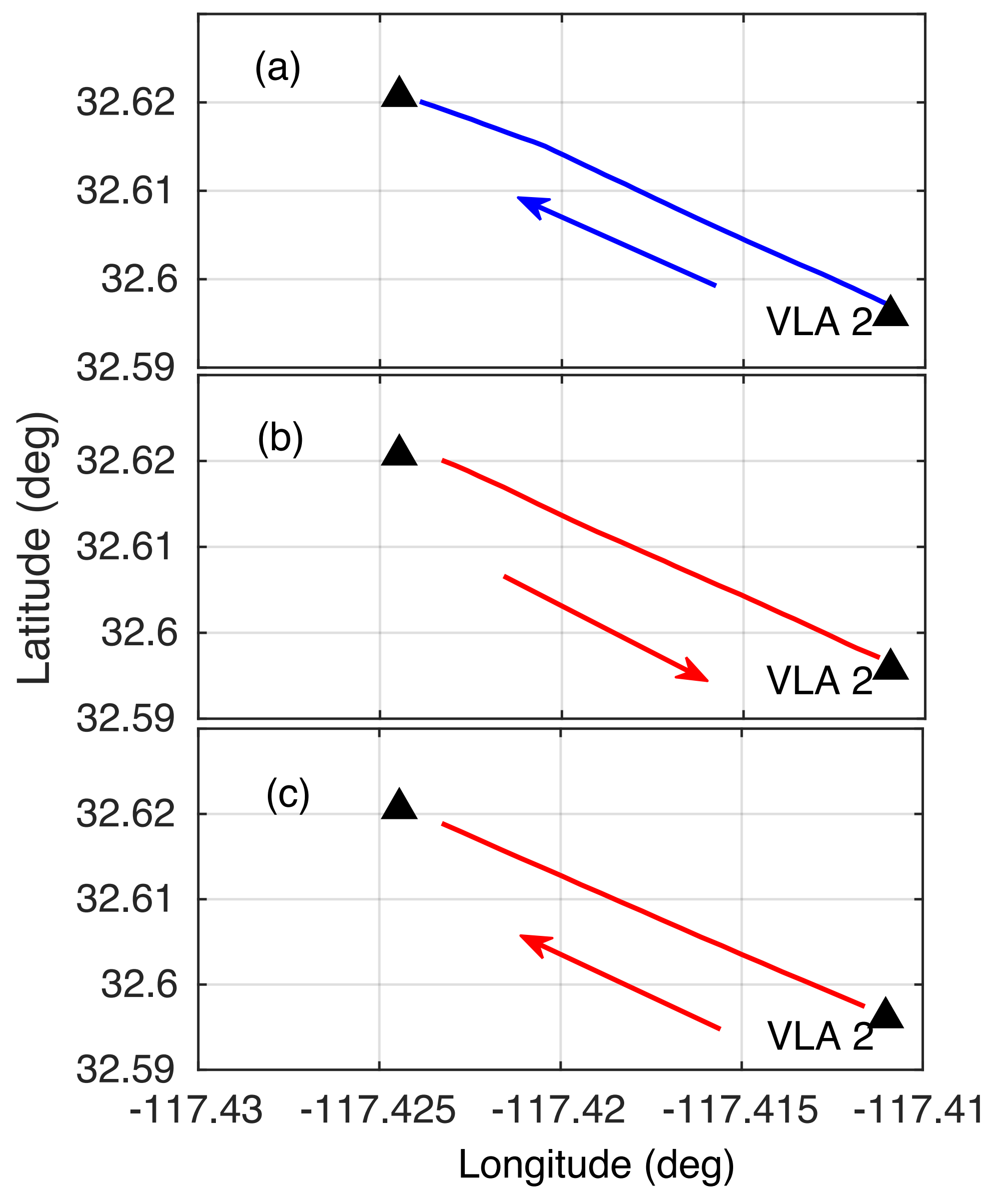


- One
- Two
- Three



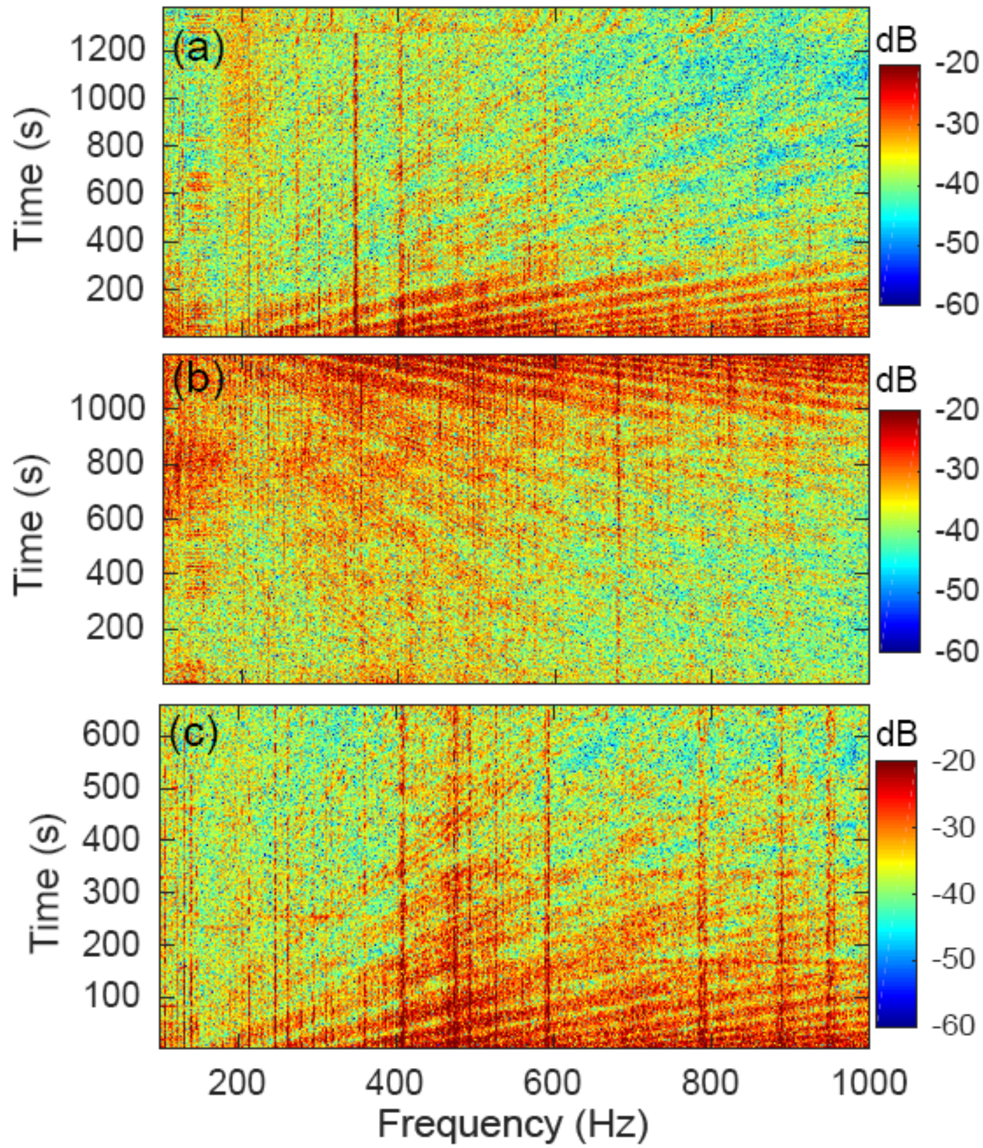
Noise 09 Experiment

- Training data
  - Jan. 31, 2009 01:43-2:05
  - 2 m/s
- Test-Data-1
  - Jan. 31, 2009 01:01-01:24
  - -2 m/s
- Test-Data-2
  - Feb. 4, 2009 13:41-13:51
  - 4 m/s



Noise 09 Experiment

- Training data
  - Jan. 31, 2009 01:43-2:05
  - 2 m/s
- Test-Data-1
  - Jan. 31, 2009 01:01-01:24
  - -2 m/s
- Test-Data-2
  - Feb. 4, 2009 13:41-13:51
  - 4 m/s

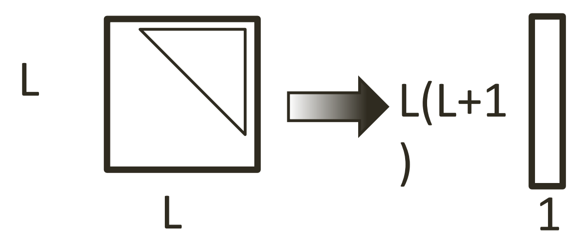


Pre-Processing

1. Convert *p(t)* to *p(f)* (Fast Fourier Transform)
2. Normalize *p(f)*: $\tilde{\mathbf{p}}(f) = \frac{\mathbf{p}(f)}{\sqrt{\sum_{l=1}^{L} |p_l(f)|^2}} = \frac{\mathbf{p}(f)}{\|\mathbf{p}(f)\|_2}$
3. Construct Cross-Spectral Density Matrix (CSDM)
   1. Contains signal coherence information
   2. Improves Signal-to-Noise Ratio (SNR)

$$\mathbf{C}(f) = \frac{1}{N_s} \sum_{s=1}^{N_s} \tilde{\mathbf{p}}_s(f) \tilde{\mathbf{p}}_s^H(f)$$

4. Concatenate upper triangular elements' real and imaginary parts, vectorize to create input **X**
   1. Reduces memory requirements



For *L* sensors, *N* training samples,

X has size *L(L+1)* × *N* .

Feed-Forward Neural Network

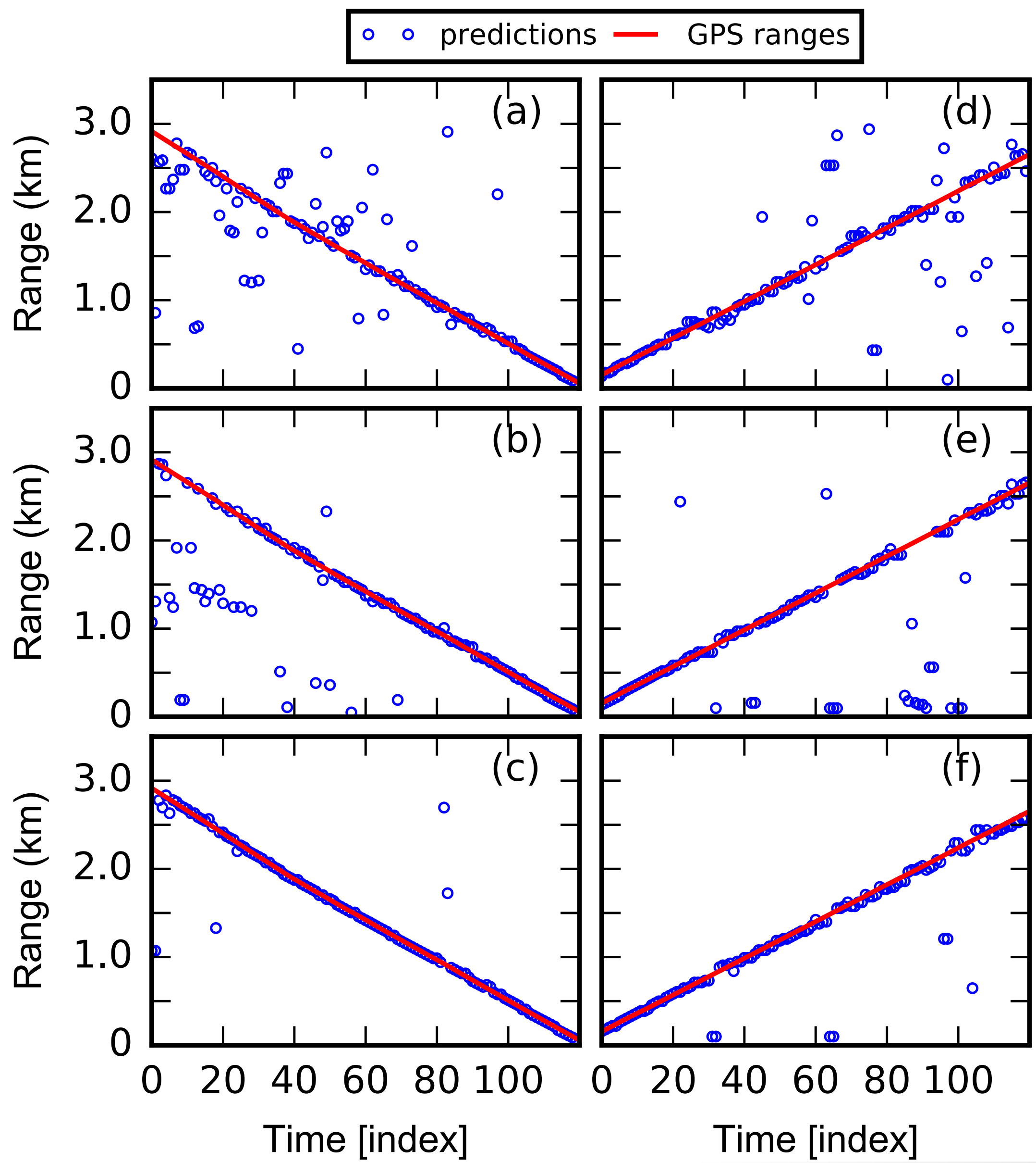
- 2-layer network
- Classification with classes r_k, $k = 1,\ldots, K$
- Activation Functions:
  - Layer 2: Sigmoid ($\sigma(\mathbf{X})$)
  - Output Layer: Softmax
- Multiple-frequency inputs to increase SNR

- Best error rate:
  Test-Data-1: $\boxed{3\%}$ **
  Freq. = 300:10:950 Hz, Hidden Nodes = 1024, # Outputs = 690, # Snapshots = 10
- Test-Data-2: $\boxed{3\%}$ **
  Freq. = 300:10:950 Hz, Hidden Nodes = 1024, # Outputs = 138, # Snapshots = 5 or 20

- **MAPE error, R_{pi} = predicted range, R_{gi} = ground truth range: $\frac{100}{N}\sum_{i=1}^{N}\left|\frac{R_{p_i}-R_{g_i}}{R_{g_i}}\right|$

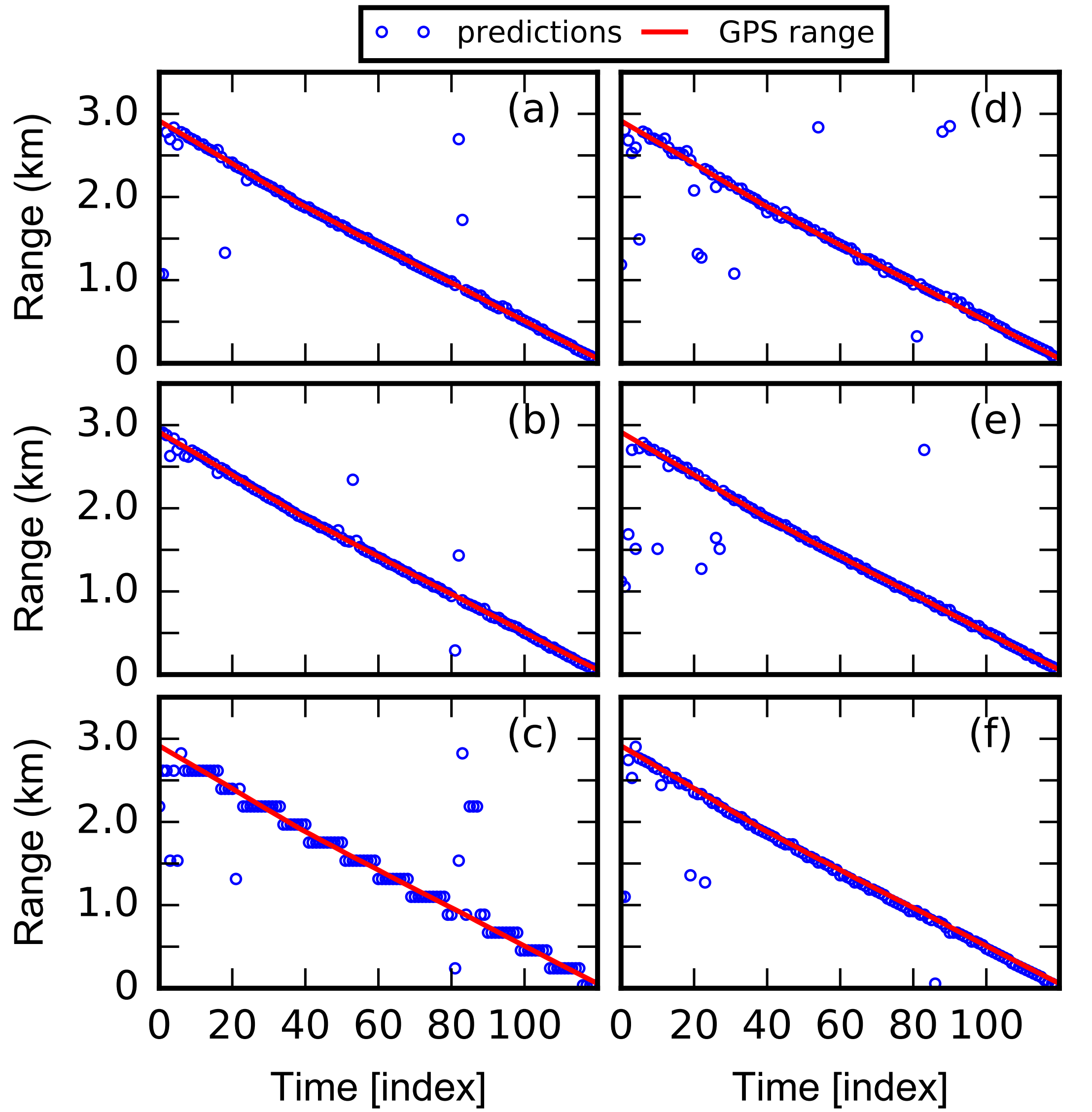
Multiple Frequencies: FNN

- (a)-(c) Test-Data-1
- (d)-(f) Test-Data-2
- *From top to bottom*: 550 Hz, 950 Hz, and 300-950 Hz with 10 Hz increments



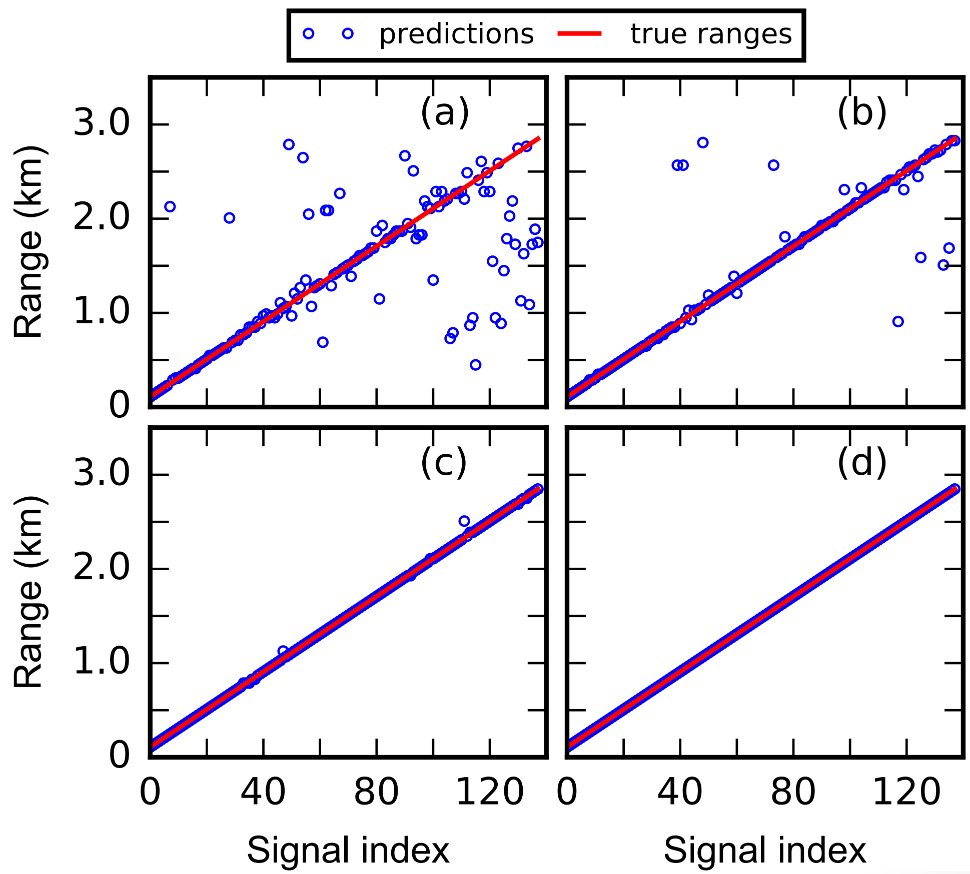
Other parameters: FNN

- Test-Data-1
- (a)-(c) varying # of classes (output nodes) 138, 690, 14 outputs
- (d)-(f) varying # of snapshots (stacked CSDMs) 1, 5, 20 snapshots



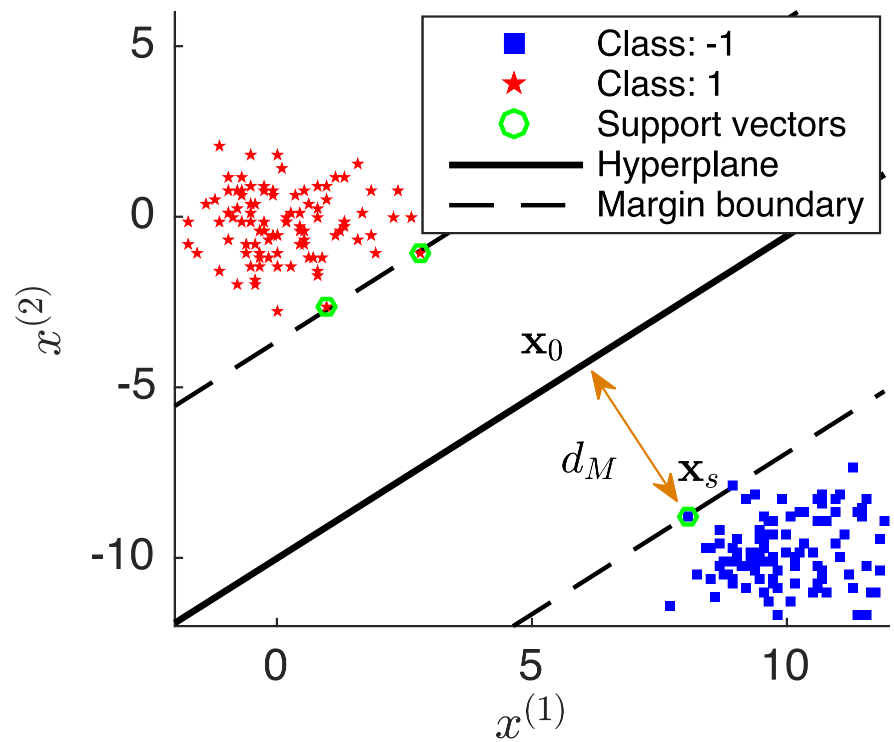
Signal-to-Noise Ratio: FNN

- SNR affects any algorithms ability to localize a source
- Source localization on simulated data with added white noise at SNR:
  - (a) -10 dB
  - (b) -5 dB
  - (c) 0 dB
  - (d) 5 dB
- Multiple frequencies, more snapshots, also increase SNR indirectly



Support Vector Machine (SVM)

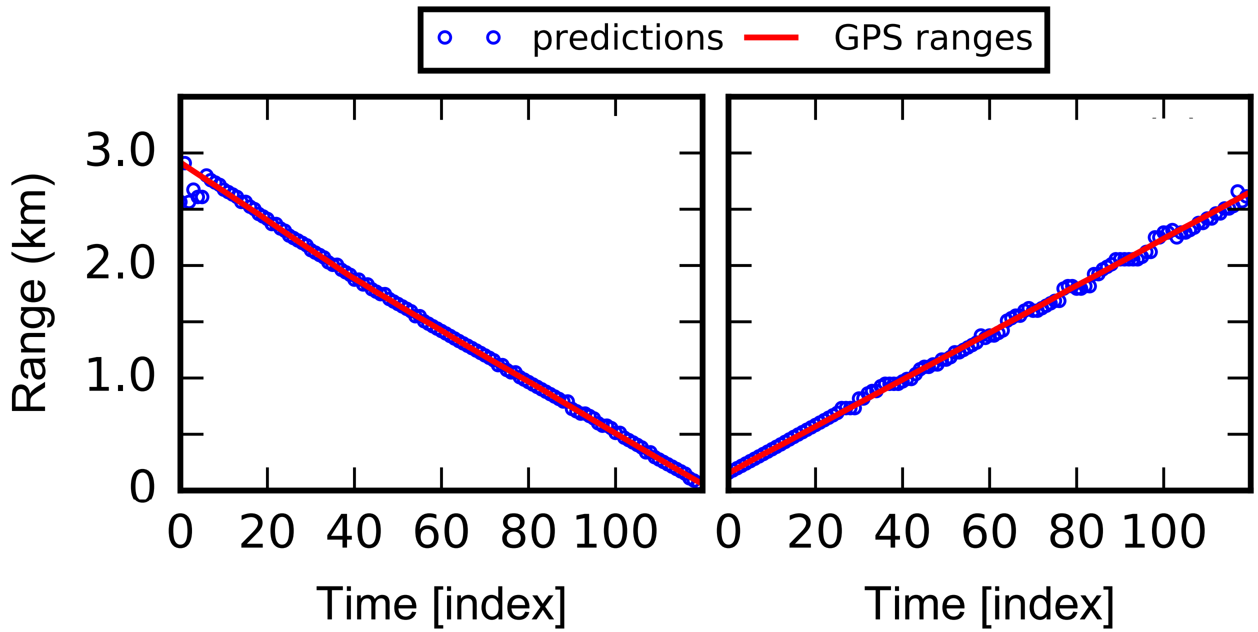
- Hyperplane maximally separates (overlapping) classes
- Shown: 2-class, 2-D example with no overlap
- Acoustic source localization has ~138 classes and 17,952 dimensions!



Support Vector Machine (SVM)

- Gaussian radial basis function (RBF): $$k = \exp(-\frac{1}{K}\|\mathbf{x} - \mathbf{x}'\|^2)$$
- Best error rate--

  Test-Data-1 (left): $\boxed{\textbf{\textit{2\%}}}$**, *Test-Data-2 (right)*: $\boxed{\textbf{\textit{2\%}}}$**

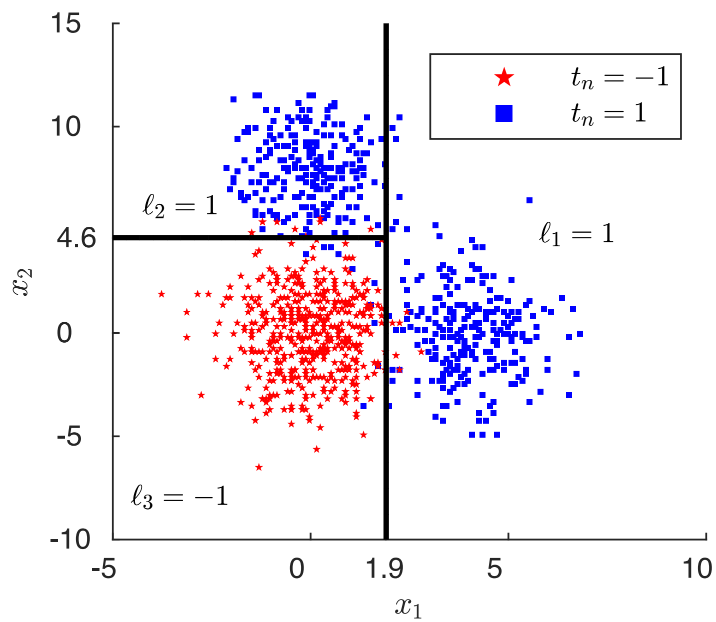


** MAPE error

Random Forest (RF)

- Gini index used to find optimal partition:

$$H = \frac{1}{n_m} \sum_{x_n \in \mathbf{x}_m} I(t_n, \ell_m) \left[1 - \frac{1}{n_m} I(t_n - \ell_m) \right]$$

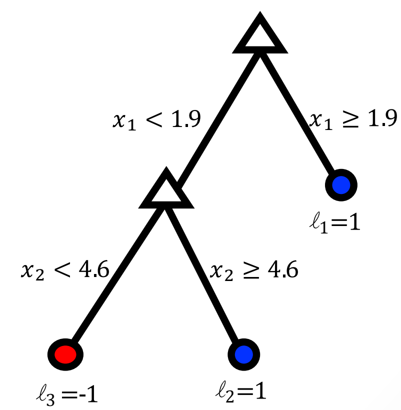


I(t$_n$ – ℓ$_m$): identity function

ℓ$_m$: estimated class for region *m*

t$_n$: true label of a point

n$_m$: number of points in region *m*



Gini index : equivalently, the percent of correctly estimated labels multiplied by the percent of incorrectly estimated labels.

Random Forest (RF)

- Bagging is used to avoid learning noise in the data

1. Learn a tree model until any new region contains less than 50 points (then stop)

2. Randomly initialize the model and run again. Since it is a greedy model, the trees likely won't match

3. Average the label for each point across all trees:

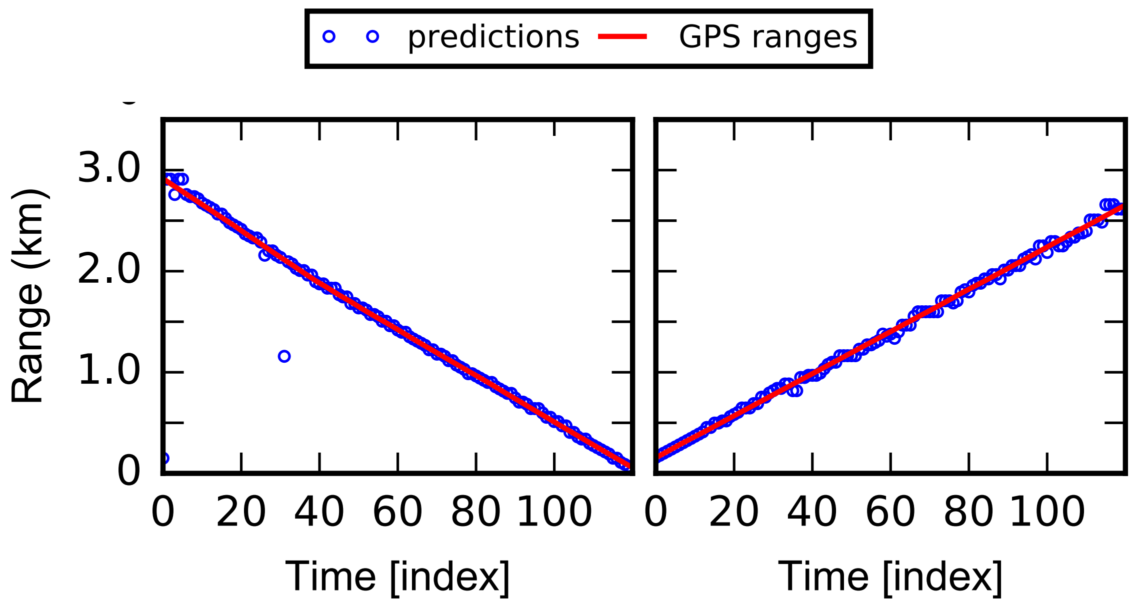
$$\hat{f}^{bag}(\mathbf{x}_n) = \frac{1}{B}\sum_{b=1}^{B}\hat{f}^{tree,b}(\mathbf{x}_n)$$

where $\hat{f}^{tree,b}(\mathbf{x}_i)$ is the estimated class of $\boldsymbol{x}_i$ for the b^{th} tree

Random Forest (RF)

- Best error rate--

  Test-Data-1 (left): **3%** **, *Test-Data-2 (right)*: **2%** **



** MAPE error

Regression v Classification

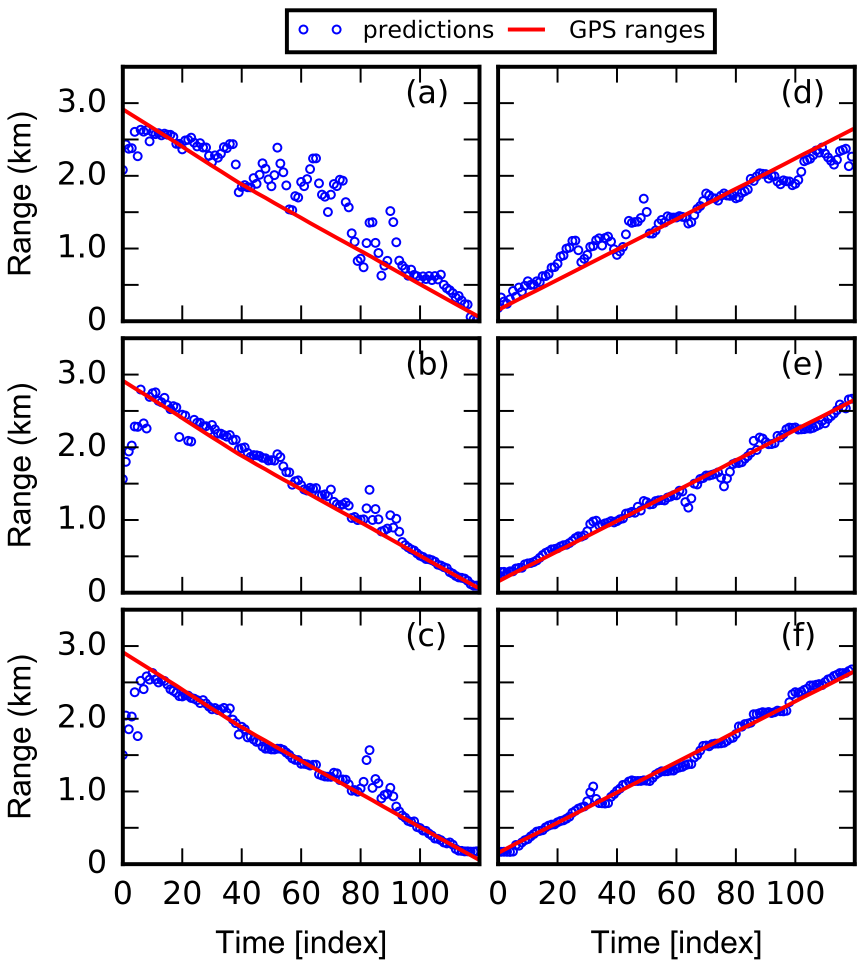
- Replace error cost function with mean squared (or absolute) error:

  - FNN: $E(\mathbf{w}) = -\frac{1}{2}\sum_{n=1}^{N}\left|y(\mathbf{x}_n, \mathbf{w}) - r_n\right|^2$
  - SVM: $E(y_n - r_n) = \begin{cases} 0, & \left|y_n - r_n\right| < \varepsilon \\ \left|y_n - r_n\right| - \varepsilon, & otherwise \end{cases}$
  - RF: $H = \sum_{x_n \in \mathbf{x}_m} (\ell_m - r_n)^2$

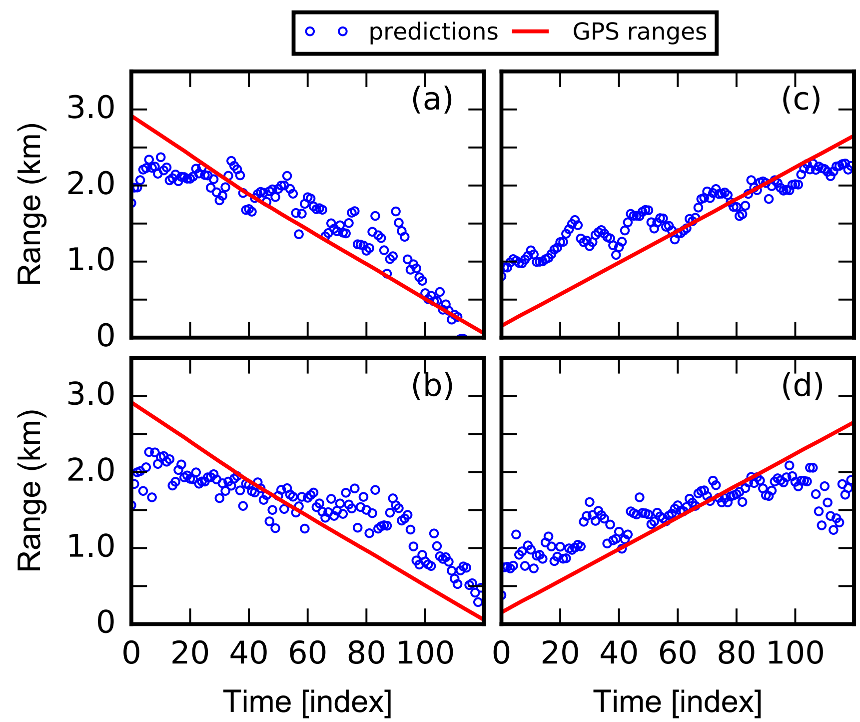
    $\ell_m = \frac{1}{n_m} \sum_{x_n \in \mathbf{x}_m} r_n$

Regression

- *Left:* FNN results for (a)-(c) Test-Data-1 and (b)-(d) Test-Data-2. *Top to Bottom:* 1, 2, and 3 hidden layers



- *Right:* SVM (top) and RF (bottom) results for (a)-(b) Test-Data-1 and (c)-(d) Test-Data-2

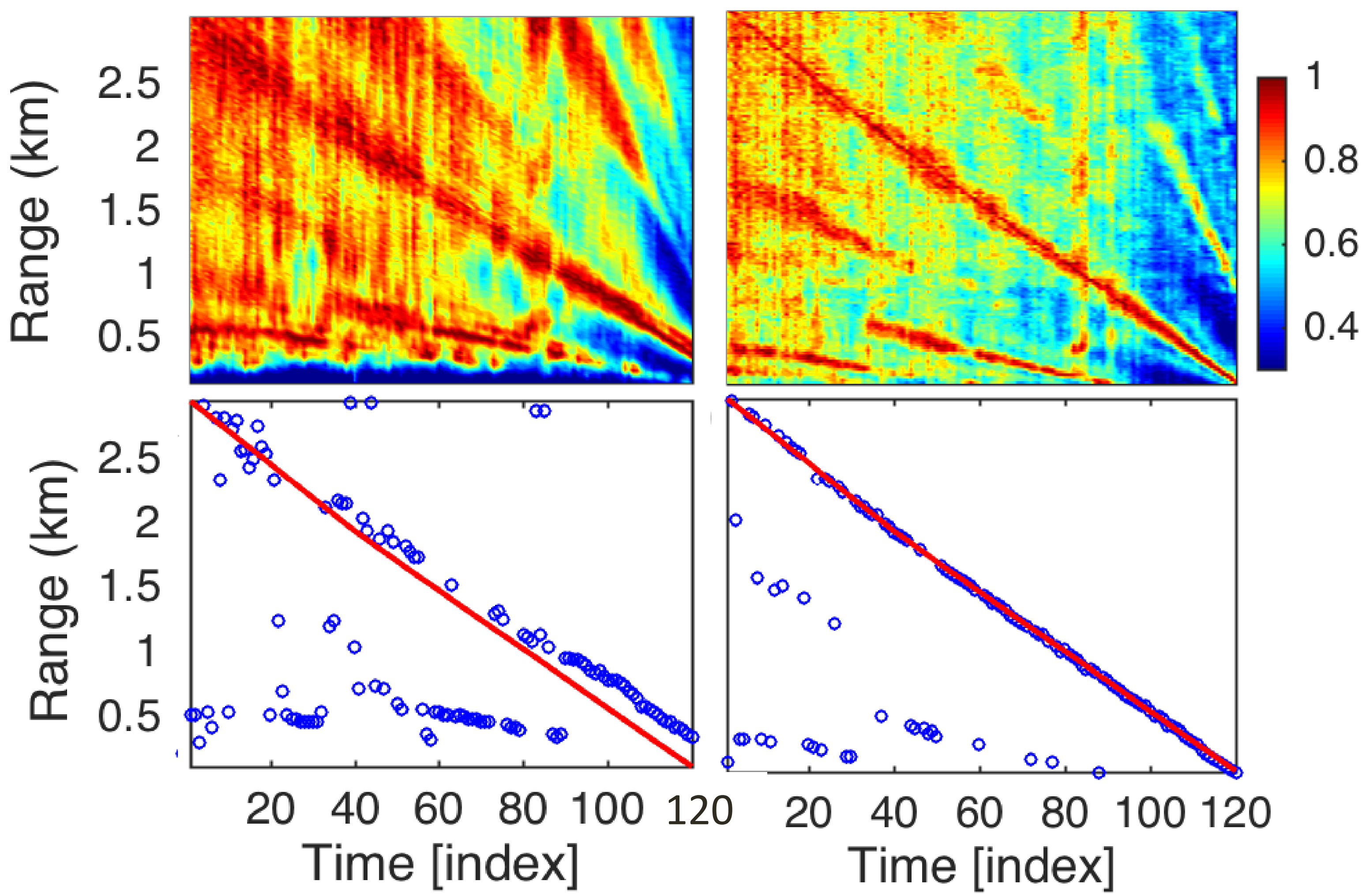


Matched-Field Processing

- Matched-field processing (MFP) is a popular method in underwater acoustics
- Maximize $|\mathbf{a}_i \mathbf{x}_i|^2$, where $\mathbf{a}_i$ is a replica and $\mathbf{x}_i$ is the data, both at the i^{th} receiver, over all *i*
- $\mathbf{a}_i$ is generated by a realistic physical model (e.g. using the wave equation)
  - Requires us to know the environment pretty well
- Add L_2 or L_1 penalties to promote sparsity
- Adaptive solutions make assumptions on the noise to suppress it

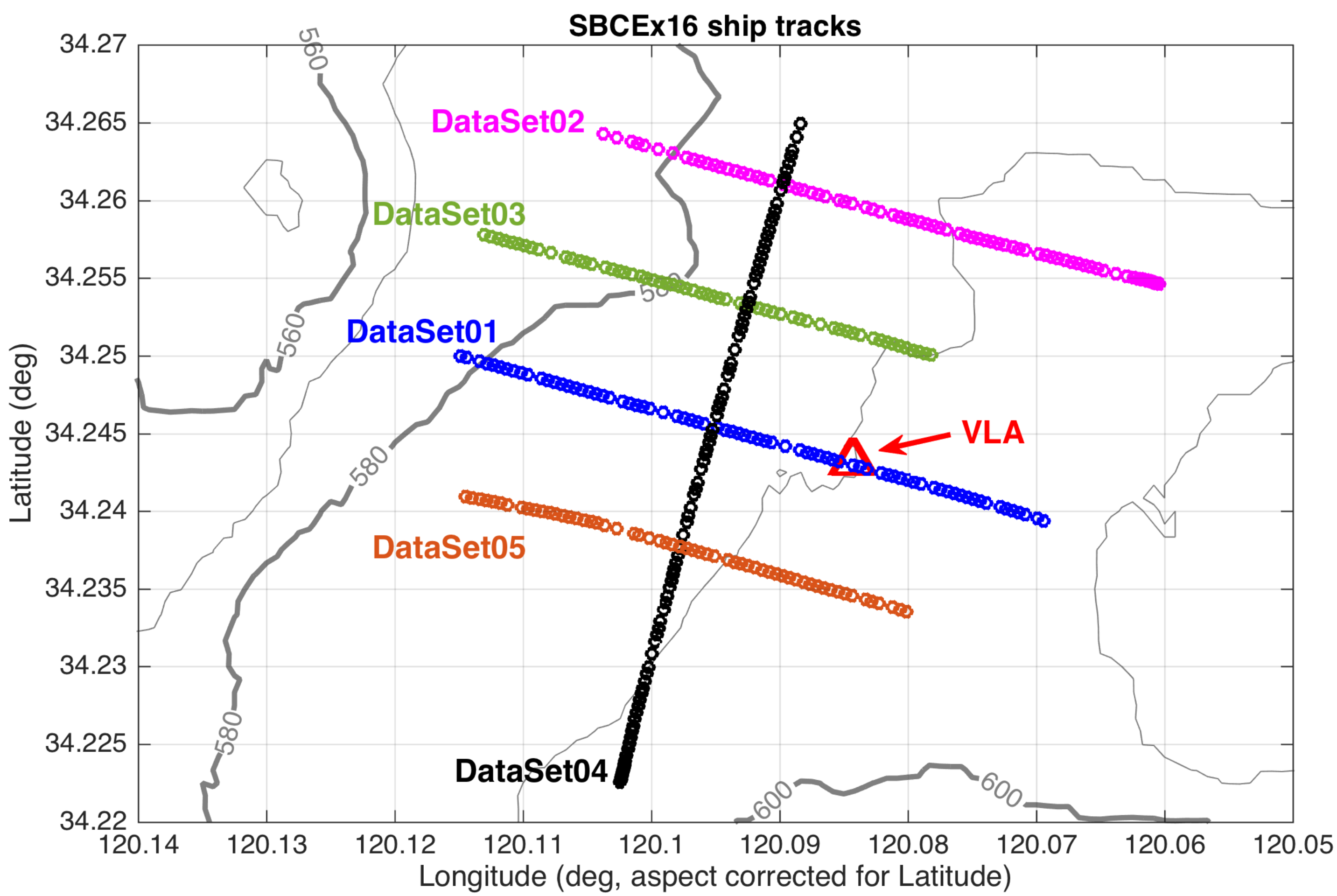
Matched-Field Processing

- The properties of sound in the ocean leads to peak ambiguities or “sidelobes”, that degrades performance



Preliminary results from SBCEx16

- tex



Part II

- How to use Python for machine learning codes

FNN in TensorFlow

1. Read in datasets

```
def read_data_sets(train_dir, FileName,fake_data=False):
    class DataSets(object):
        pass
    data_sets = DataSets()
    if fake_data:
        data_sets.train = DataSet([], [], fake_data=True)
        data_sets.validation = DataSet([], [], fake_data=True)
        data_sets.test = DataSet([], [], fake_data=True)
        return data_sets

    train_images = numpy.loadtxt(train_dir + '/train_input/'+ FileName)
    train_labels = numpy.loadtxt(train_dir + '/train_label/'+FileName)
    test_images = numpy.loadtxt(train_dir + '/test_input/'+FileName)
    test_labels = numpy.loadtxt(train_dir + '/test_label/'+FileName)
#     train_images = numpy.loadtxt(train_dir + '/Noise09_traindata_x_450Hz.txt')
#     train_labels = numpy.loadtxt(train_dir + '/Noise09_traindata_y_450Hz.txt')
#     test_images = numpy.loadtxt(train_dir + '/Noise09_testdata_x_450Hz.txt')
#     test_labels = numpy.loadtxt(train_dir + '/Noise09_testdata_y_450Hz.txt')

    if train_labels.ndim == 1:
        train_labels = numpy.reshape(train_labels,(train_labels.size,1))
        test_labels = numpy.reshape(test_labels,(test_labels.size,1))

    print train_images.shape,train_labels.shape,test_images.shape,test_labels.shape
    data_sets.train = DataSet(train_images, train_labels)
    data_sets.test = DataSet(test_images, test_labels)
    return data_sets
```

define a function

not used

load training and test sets

fix dimensions to match TensorFlow input

make sure dimensions look right

assign to *data_sets* object

FNN in TensorFlow

- Properties of *data_sets* object

```
class DataSet(object):
    def __init__(self, images, labels, fake_data=False):
        if fake_data:
            self._num_examples = 10000
        else:
            assert images.shape[0] == labels.shape[0], (
                "images.shape: %s labels.shape: %s" % (images.shape,
                                                       labels.shape))
            self._num_examples = images.shape[0]
            # Convert shape from [num examples, rows, columns, depth]
            # to [num examples, rows*columns] (assuming depth == 1)
            #assert images.shape[3] == 1
            #images = images.reshape(images.shape[0],
            #                        images.shape[1] * images.shape[2])
            # Convert from [0, 255] -> [0.0, 1.0].
            #             images = images.astype(numpy.float32)
            #images = numpy.multiply(images, 1.0 / 255.0)
        self._images = images
        self._labels = labels
        self._epochs_completed = 0
        self._index_in_epoch = 0
```

FNN in TensorFlow

- How to define the mini-batch update
- Do this within the *data_sets* object declaration

```
    def next_batch(self, batch_size, fake_data=False):
        """Return the next `batch_size` examples from this data set."""
        if fake_data:
            fake_image = [1.0 for _ in xrange(34)]
            fake_label = 0
            return [fake_image for _ in xrange(batch_size)], [
                fake_label for _ in xrange(batch_size)]
        start = self._index_in_epoch
        self._index_in_epoch += batch_size
        if self._index_in_epoch > self._num_examples:
            # Finished epoch
            self._epochs_completed += 1
            # Shuffle the data
            #numpy.random.seed(42)
            perm = numpy.arange(self._num_examples)
            numpy.random.shuffle(perm)
            self._images = self._images[perm]
            self._labels = self._labels[perm]
            # Start next epoch
            start = 0
            self._index_in_epoch = batch_size
            assert batch_size <= self._num_examples
        end = self._index_in_epoch
        #print(start,end)
        return self._images[start:end], self._labels[start:end]
```

FNN in TensorFlow

- Import libraries

```
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

import tensorflow as tf
import numpy as np
import load_data_nhq_si
import matplotlib.pyplot as plt
import matplotlib.cm as cm
from matplotlib.patches import Polygon
from matplotlib.collections import PatchCollection
from os import listdir
from os.path import isfile, join
```

prevents Python from getting confused

import separate file to load data

here we use MatPlotLib

when looping over files in a directory

FNN in TensorFlow

- TensorFlow uses “flags” to keep track of model parameters

```
flags = tf.app.flags
FLAGS = flags.FLAGS
flags.DEFINE_boolean('fake_data', False, 'If true, uses fake data '
                     'for unit testing.')
#flags.DEFINE_boolean('train', True, 'If true, training')
flags.DEFINE_integer('max_steps', 2000, 'Number of steps to run trainer.')
flags.DEFINE_float('learning_rate', 0.01, 'Initial learning rate.')
flags.DEFINE_float('dropout', 0.5, 'Keep probability for training dropout.')
flags.DEFINE_string('data_dir', './data/'+Data_set, 'Directory for storing data')
```

- It also uses “Interactive Sessions” that update variables dynamically when run

```
sess = tf.InteractiveSession()
```

FNN in TensorFlow

- First, load data and define neural network architecture
- (We put this inside the function *train_and_prediction()* for processing multiple files)
- (You could define the function with more inputs, like *n_hidden*, to try different architectures)

```
source_data = load_data_nhq_si.read_data_sets(FLAGS.data_dir, FileName, fake_data=FLAGS.fake_data)
n_input = 17952  # 66 frequencies
# n_input = 272  # single frequency
n_output = 138
n_hidden = 1024
n_mini_batch = 128
sess = tf.InteractiveSession()
```

FNN in TensorFlow

- Second, define your variables
- These include: *weights, biases, weight and bias random initialization, cost function, optimization method, performance metrics (i.e. mean square error)*

- Example: define weight variable

```
def weight_variable(shape):
  """Create a weight variable with appropriate initialization."""
  initial = tf.truncated_normal(shape, stddev=0.02)
  return tf.Variable(initial)
```

other types:
constant, *weight_variable,* and
bias_variable.
See TensorFlow documentation for more.

FNN in TensorFlow

- Third, define a neural network layer

- As we saw in previous lectures, we usually use **sigmoid** activation function for hidden layers and **softmax** or sigmoid for output layers

- You can always find more in online TensorFlow documents

```
def nn_layer(input_tensor, input_dim, output_dim, layer_name, act=tf.nn.relu):
  """Reusable code for making a simple neural net layer.

  It does a matrix multiply, bias add, and then uses relu to nonlinearize.
  It also sets up name scoping so that the resultant graph is easy to read, and
  adds a number of summary ops.
  """
  # Adding a name scope ensures logical grouping of the layers in the graph.
  with tf.name_scope(layer_name):
    # This Variable will hold the state of the weights for the layer
    with tf.name_scope('weights'):
      weights = weight_variable([input_dim, output_dim])
    with tf.name_scope('biases'):
      biases = bias_variable([output_dim])
    with tf.name_scope('Wx_plus_b'):
      preactivate = tf.matmul(input_tensor, weights) + biases
    if act:
      activations = act(preactivate, 'activation')
    else:
      activations = preactivate
    return activations
```

preallocate weights & biases

compute activation

apply activation function (if any)

FNN in TensorFlow

- Fourth, define neural network layer architecture

```
hidden1 = nn_layer(x, n_input, n_hidden, 'layer1', act=tf.nn.sigmoid)
dropped = tf.nn.dropout(hidden1, keep_prob)
y = nn_layer(dropped, n_hidden, n_output, 'layer2', act=False)
```

special TensorFlow feature:
Softmax is built into the error function (*softmax_cross_entropy_with_logits*)

A dropout layer removes hidden nodes with probability 50% to prevent overfitting.

FNN in TensorFlow

- Finally, *train* and *predict*!

- All variables must be initialized first:

```
tf.initialize_all_variables().run()
```

```
print('------------training process------------')
for i in range(FLAGS.max_steps):
  if i % 100 == 0:  # Record summaries and training or test-set accuracy
    acc = sess.run(accuracy, feed_dict=feed_dict(True))
    print('Accuracy at step %s: %s' % (i, acc))
  else: # Record train set summarieis, and train
    _ = sess.run(train_step, feed_dict=feed_dict(True))

print('---------------predicting---------------')
predict_out = sess.run(tf.nn.softmax(y), feed_dict=feed_dict(False))
predictions = sess.run(y, feed_dict=feed_dict(False))
```

- Run the Interactive Session we defined earlier

Next step, deep learning?.....

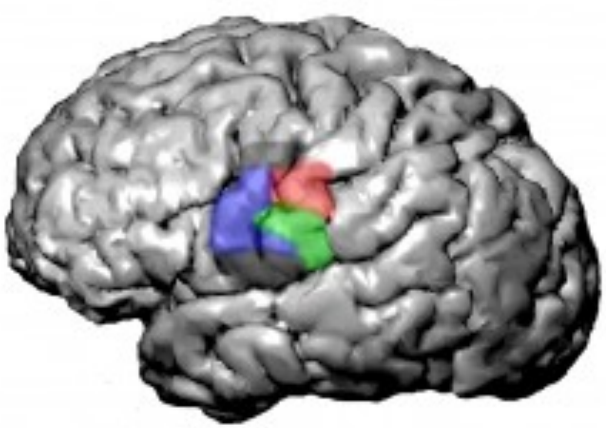


photo credit: Kris Bouchard

photo credit: Roy Kaltschmidt

Berkeley Lab Computing Sciences