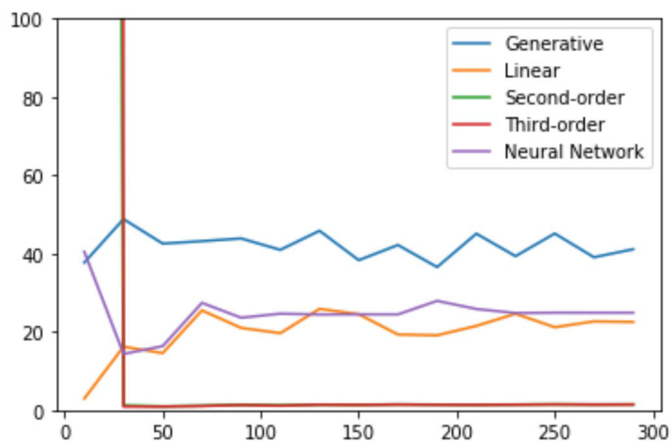
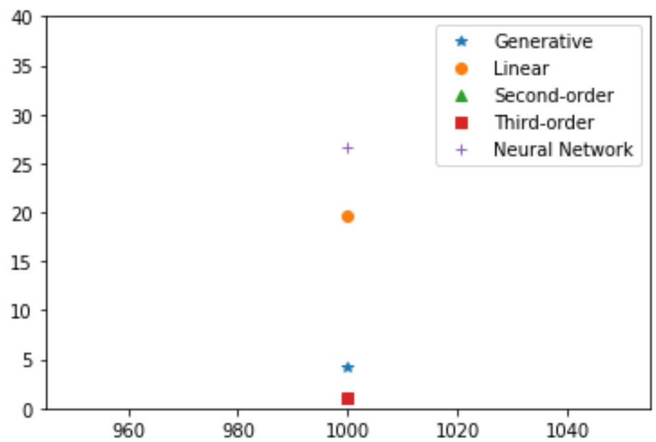
In [107]:
```
import numpy as np
import matplotlib.pyplot as plt
import scipy.spatial
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
import math
import warnings;

warnings.simplefilter('ignore')

# Gradient descent optimization
# The learning rate is specified by eta
class GDOptimizer(object):
    def __init__(self, eta):
        self.eta = eta

    def initialize(self, layers):
        pass

    # This function performs one gradient descent step
    # layers is a list of dense layers in the network
    # g is a list of gradients going into each layer before the nonlinear activa
tion
    # a is a list of of the activations of each node in the previous layer going
    def update(self, layers, g, a):
        m = a[0].shape[1]
        for layer, curGrad, curA in zip(layers, g, a):
            update = np.dot(curGrad,curA.T)
            updateB = np.sum(curGrad,1).reshape(layer.b.shape)
            layer.updateWeights(-self.eta/m * np.dot(curGrad,curA.T))
            layer.updateBias(-self.eta/m * np.sum(curGrad,1).reshape(layer.b.sha
pe))

# Cost function used to compute prediction errors
class QuadraticCost(object):

    # Compute the squared error between the prediction yp and the observation y
    # This method should compute the cost per element such that the output is th
e
    # same shape as y and yp
    @staticmethod
    def fx(y,yp):
        return 0.5 * np.square(yp-y)

    # Derivative of the cost function with respect to yp
    @staticmethod
    def dx(y,yp):
        return y - yp

# Sigmoid function fully implemented as an example
class SigmoidActivation(object):
    @staticmethod
    def fx(z):
        return 1 / (1 + np.exp(-z))

    @staticmethod
    def dx(z):
        return SigmoidActivation.fx(z) * (1 - SigmoidActivation.fx(z))

# Hyperbolic tangent function
```

In [108]:
```
##########
## main ##
##########

# 4(b).1

n_train = range(10, 291, 20)

sensor_loc = generate_sensors()
errors = np.zeros( (15, 5), "float" )
np.random.seed(0)
for i, n in enumerate(n_train):
    distance, obj_loc = generate_dataset(sensor_loc, num_data=n)
    # Generative Model
    errors[i, 0] = GM_solver(distance, obj_loc)
    # OLS Linear:
    _, errors[i, 1] = OLS_polynomial_solver(1, distance, obj_loc)
    # OLS Second-order
    _, errors[i, 2] = OLS_polynomial_solver(2, distance, obj_loc)
    # OLS Third-order
    _, errors[i, 3] = OLS_polynomial_solver(3, distance, obj_loc)
    # Neural Network Model
    errors[i, 4] = NN_solver(distance, obj_loc)

plt.figure()
plt.plot(n_train, errors[:, 0], label="Generative")
plt.plot(n_train, errors[:, 1], label="Linear")
plt.plot(n_train, errors[:, 2], label="Second-order")
plt.plot(n_train, errors[:, 3], label="Third-order")
plt.plot(n_train, errors[:, 4], label="Neural Network")
plt.ylim((0, 100))
plt.legend(loc="best")
plt.show()
```

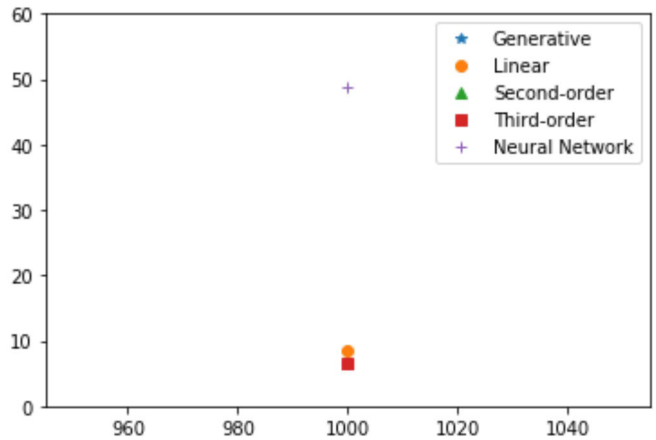


In [116]:

```
##########
## main ##
##########

# 4(b).2

n_test = [1000]

sensor_loc = generate_sensors()
errors = np.zeros( (1, 5), "float" )
np.random.seed(0)
for i, n in enumerate(n_test):
    distance, obj_loc = generate_dataset(sensor_loc, num_data=n)
    # Generative Model
    errors[i, 0] = GM_solver(distance, obj_loc)
    # OLS Linear:
    _, errors[i, 1] = OLS_polynomial_solver(1, distance, obj_loc)
    # OLS Second-order
    _, errors[i, 2] = OLS_polynomial_solver(2, distance, obj_loc)
    # OLS Third-order
    _, errors[i, 3] = OLS_polynomial_solver(3, distance, obj_loc)
    # Neural Network Model
    errors[i, 4] = NN_solver(distance, obj_loc)

plt.figure()
plt.plot(n_test, errors[:, 0], "*", label="Generative")
plt.plot(n_test, errors[:, 1], "o", label="Linear")
plt.plot(n_test, errors[:, 2], "^", label="Second-order")
plt.plot(n_test, errors[:, 3], "s", label="Third-order")
plt.plot(n_test, errors[:, 4], "+", label="Neural Network")
plt.ylim((0, 40))
plt.legend(loc="best")
plt.show()
```

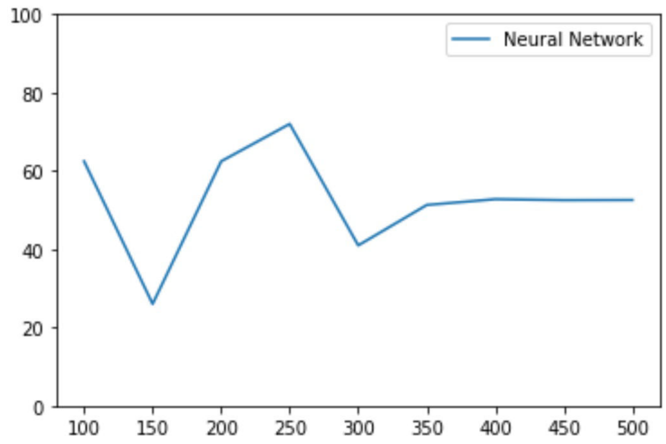


In [118]:
```
##########
## main ##
##########

# 4(b).3

n_test = [1000]

sensor_loc = generate_sensors()
errors = np.zeros( (1, 5), "float" )
np.random.seed(0)
for i, n in enumerate(n_test):
    distance, obj_loc = generate_dataset(sensor_loc, num_data=n, original_dist=
alse)
    # Generative Model
    errors[i, 0] = GM_solver(distance, obj_loc)
    # OLS Linear:
    _, errors[i, 1] = OLS_polynomial_solver(1, distance, obj_loc)
    # OLS Second-order
    _, errors[i, 2] = OLS_polynomial_solver(2, distance, obj_loc)
    # OLS Third-order
    _, errors[i, 3] = OLS_polynomial_solver(3, distance, obj_loc)
    # Neural Network Model
    errors[i, 4] = NN_solver(distance, obj_loc)

plt.figure()
plt.plot(n_test, errors[:, 0], "*", label="Generative")
plt.plot(n_test, errors[:, 1], "o", label="Linear")
plt.plot(n_test, errors[:, 2], "^", label="Second-order")
plt.plot(n_test, errors[:, 3], "s", label="Third-order")
plt.plot(n_test, errors[:, 4], "+", label="Neural Network")
plt.ylim((0, 60))
plt.legend(loc="best")
plt.show()
```

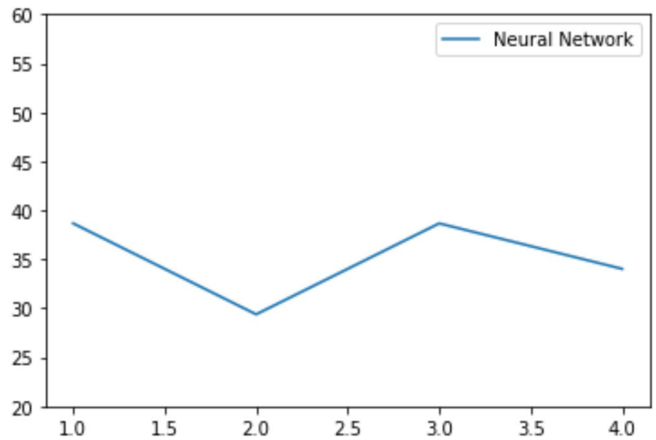


In [134]:

```
##########
## main ##
##########

# 4(c)

def NN_solver_tune_1(l, distance, obj_loc, distance_test, obj_loc_test):
    x=distance
    y=obj_loc
    xtest=distance_test
    ytest=obj_loc

    activations = dict( eL = eL Activation,
                       tanh=TanhActivation,
                       linear=LinearActivation)
    lr = dict( eL =0.02,tanh=0.02,linear=0.005)
    names = [' eL ','linear','tanh']

    for key in [' eL ']:

        # Build the model
        activation = activations[key]
        model = Model(x.shape[1])
        model.addLayer( enseLayer(l,activation()))
        model.addLayer( enseLayer(l,activation()))
        model.addLayer( enseLayer(2,LinearActivation()))
        model.initialize( uadratic ost())

        # Train the model and display the results
        hist = model.train(x,y,500,G Optimizer(eta=lr[key]))
        y at = model.predict(x)
        squared_diff = np.square(y at - y)
        error = np.mean(np.sqrt(squared_diff[:, 0] + squared_diff[:, 1]))

    return error

l = range(100, 501, 50)
n_train = 200
n_test = 1000
errors = np.zeros( (len(l),), "float")

sensor_loc = generate_sensors()
np.random.seed(9001)
for i, ll in enumerate(l):
    distance, obj_loc = generate_dataset(sensor_loc, num_data=n_train)
    distance_test, obj_loc_test = generate_dataset(sensor_loc, num_data=n_test)
    # Neural Network Model
    errors[i] = NN_solver_tune_1(ll, distance, obj_loc, distance_test, obj_loc_t
est)
    mm = np.mean(errors[:i-1])

plt.figure()
plt.plot(l, errors, label="Neural Network")
plt.ylim((0, 100))
plt.legend(loc="best")
plt.show()
```


Neural Network
100
80
60
40
20
0
100
150
200
250
300
350
400
450
500

In [141]:
```
##########
## main ##
##########

# 4(d)

def NN_solver_tune_2(k, l, distance, obj_loc, distance_test, obj_loc_test):
    x=distance
    y=obj_loc
    xtest=distance_test
    ytest=obj_loc

    activations = dict( eL = eL Activation,
                       tanh=TanhActivation,
                       linear=LinearActivation)
    lr = dict( eL =0.02,tanh=0.02,linear=0.005)
    names = [' eL ','linear','tanh']

    for key in [' eL ']:

        # Build the model
        activation = activations[key]
        model = Model(x.shape[1])
        for _ in range(k):
            model.addLayer( enseLayer(l,activation()))
        model.addLayer( enseLayer(2,LinearActivation()))
        model.initialize( uadratic ost())

        # Train the model and display the results
        hist = model.train(x,y,500,G Optimizer(eta=lr[key]))
        y at = model.predict(x)
        squared_diff = np.square(y at - y)
        error = np.mean(np.sqrt(squared_diff[:, 0] + squared_diff[:, 1]))

    return error

k = range(1, 5)
n_train = 200
n_test = 200
errors = np.zeros( (len(k),), "float")

sensor_loc = generate_sensors()
np.random.seed(9001)
for i, kk in enumerate(k):
    if kk == 1:
        ll = 1000
    elif kk == 2:
        ll = 95
    elif kk == 3:
        ll = 67
    else:
        ll = 55
    distance, obj_loc = generate_dataset(sensor_loc, num_data=n_train)
    distance_test, obj_loc_test = generate_dataset(sensor_loc, num_data=n_test)
    # Neural Network Model
    errors[i] = NN_solver_tune_2(kk, ll, distance, obj_loc, distance_test, obj_l
oc_test)
    mm = np.mean(errors[:i-1])

plt.figure()
plt.plot(k, errors, label="Neural Network")
plt.ylim((20, 60))
```



In [147]:

```
##########
## main ##
##########

k = 2
l = 150

# 4(e)
sensor_loc = generate_sensors()
np.random.seed(9001)
n_train = 200
n_test = 1000
distance, obj_loc = generate_dataset(sensor_loc, num_data=n_train)
error = NN_solver_tune_2(k, l, distance, obj_loc, distance_test, obj_loc_test)
print(error)
```

```
15.201287385073272
```

In [146]:
```
## Problem 5

# Copyright 2015 The TensorFlow Authors. All Rights Reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
# ==============================================================================

"""A very simple MNIST classifier.

See extensive documentation at
https://www.tensorflow.org/get_started/mnist/beginners
"""
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

import argparse
import sys

from tensorflow.examples.tutorials.mnist import input_data

import tensorflow as tf

 LAGS = None


def main(_):
  # Import data
  mnist = input_data.read_data_sets( LAGS.data_dir, one_hot=True)

  # Create the model
  x = tf.placeholder(tf.float32, [None, 784])
  W = tf. ariable(tf.zeros([784, 10]))
  b = tf. ariable(tf.zeros([10]))
  y = tf.matmul(x, W) + b

  # Define loss and optimizer
  y_ = tf.placeholder(tf.float32, [None, 10])

  # The raw formulation of cross-entropy,
  #
  #   tf.reduce_mean(-tf.reduce_sum(y_ * tf.log(tf.nn.softmax(y)),
  #                                 reduction_indices=[1]))
  #
  # can be numerically unstable.
  #
  # So here we use tf.nn.softmax_cross_entropy_with_logits on the raw
  # outputs of 'y', and then average across the batch.
  cross_entropy = tf.reduce_mean(
      tf.nn.softmax_cross_entropy_with_logits(labels=y_, logits=y))
  train_step = tf.train.GradientDescentOptimizer(0.5).minimize(cross_entropy)
```

```
 xtracting /tmp/tensorflow/mnist/input_data train-images-idx3-ubyte.gz
 xtracting /tmp/tensorflow/mnist/input_data train-labels-idx1-ubyte.gz
 xtracting /tmp/tensorflow/mnist/input_data t10k-images-idx3-ubyte.gz
 xtracting /tmp/tensorflow/mnist/input_data t10k-labels-idx1-ubyte.gz
0.9218
```

```
An exception has occurred, use  tb to see the full traceback.
```

System xit