In [1]:
```
%matplotlib inline
%matplotlib widget
# from ipywidgets import interact, interactive, fixed, interact_manual
import ipywidgets as widgets
import torch
import torchvision
import torch.nn.functional as F
import torch.nn as nn
from torch.utils.data import DataLoader, TensorDataset
import numpy as np
import matplotlib.pyplot as plt
import matplotlib
import math
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
import pandas

torch.manual_seed(48)
np.random.seed(48)
```

In [2]:
```
##---- Part a) Import and partition data ----##

raw_df = pandas.read_csv('housing.csv')  # read data from csv provided
data   = raw_df.values[:, 1:-1]          # x data: inputs: housing properties/criteria
target = raw_df.values[:, -1]            # y data: output: target/house price, last column

X_train, X_test, Y_train, Y_test = train_test_split(data, target, test_size=0.3, random_state=48) # divide data into test and train
x_vec_numpy_train, x_vec_numpy_valid, y_vec_numpy_train, y_vec_numpy_valid = train_test_split(X_train, Y_train, test_size=0.2, random_state=48) # divide data into test

n_train = np.shape(x_vec_numpy_train)[0] # number of training data points
n_valid = np.shape(x_vec_numpy_valid)[0] # number of validation data points
n_test  = np.shape(Y_test)[0]            # number of test data points

# plot unscaled data
plt.figure()
cri = 6
plt.plot(x_vec_numpy_train[:,cri], x_vec_numpy_train[:,cri+4], 'k.', label='Training Data')
plt.plot(X_test[:,cri], X_test[:,cri+4], 'r.', label='Test Data')
plt.plot(x_vec_numpy_valid[:,cri], x_vec_numpy_valid[:,cri+4], 'c.', label='Validation Data')
plt.xlabel('Age')
plt.ylabel('Tax')
plt.legend()
plt.title('Before Scaling')
plt.show()
```

In [3]:
```
##---- Part b) Scale data ----##

scaler = StandardScaler().fit(x_vec_numpy_train)
x_vec_numpy_train = scaler.transform(x_vec_numpy_train)
x_vec_numpy_valid = scaler.transform(x_vec_numpy_valid)
X_test            = scaler.transform(X_test)

# plot scaled data
plt.figure()
plt.plot(x_vec_numpy_train[:,cri], x_vec_numpy_train[:,cri+4], 'k.', label='Training Data')
plt.plot(X_test[:,cri], X_test[:,cri+4], 'r.', label='Test Data')
plt.plot(x_vec_numpy_valid[:,cri], x_vec_numpy_valid[:,cri+4], 'c.', label='Validation Data')
plt.xlabel('Age')
plt.ylabel('Tax')
plt.title('After Scaling')
plt.legend()
plt.show()
```

In [4]:
```
##---- Part c) Create model ----##

shuffleOpt   = True # choose to shuffle or not
# Convert to torch tensors
train_data   = TensorDataset(torch.tensor(x_vec_numpy_train).float(),torch.tensor(y_vec_numpy_train).float())
valid_data   = TensorDataset(torch.tensor(x_vec_numpy_valid).float(),torch.tensor(y_vec_numpy_valid).float())
test_data    = TensorDataset(torch.tensor(X_test).float(), torch.tensor(Y_test).float())
train_loader = DataLoader(train_data, batch_size = n_train//10, shuffle=shuffleOpt)
valid_loader = DataLoader(valid_data, batch_size = n_valid,     shuffle=shuffleOpt)
test_loader  = DataLoader(test_data,  batch_size = n_test,      shuffle=shuffleOpt)

# define a simple feed-foward neural network with
#  n_hidden hidden layers and n_nodes per hidden layer

class SimpleFNN(nn.Sequential):
    def __init__(self, n_hidden, n_nodes):
        super(SimpleFNN, self).__init__()
        # this is a linear layer (basically matrix multiplication)
        self.add_module('linear0', nn.Linear(13, 1)) # Linear(inputDataSize,targetSize)

# Create a SimpleFNN object and train the model

# ---- Define training parameters --------
epochs = 1000  # iteration number
lr     = 0.01  # learning rate
# ----------------------------------------

nodes  = 1;    # number of nodes, unused here since there are no hidden layers
hidden = 1;    # number of hidden layers, unused here

fnn_model = SimpleFNN(hidden, nodes) # define the model

# fnn_model =torch.nn.Sequential(torch.nn.Linear(13, 1))
# torch.nn.init.normal_(fnn_model[0].weight, mean=0, std=0.1)
# torch.nn.init.constant_(fnn_model[0].bias, val=0)

# define the optimizers (Adam ; lr = learning rate)
fnn_opt = torch.optim.Adam(fnn_model.parameters(), lr)

# for name, param in fnn_model.named_parameters():
#     if param.requires_grad:
#         print(name, param.data)

# keep track of losses
fnn_loss_train = np.zeros((epochs,))
fnn_loss_valid = np.zeros((epochs,))

# tell the model we are training
fnn_model.train()

for epoch in range(epochs):

    # first compute training loss and step optimizer
    for n, (x_train, y_train) in enumerate(train_loader):

        # clear the gradients
        fnn_model.zero_grad()

        # compute the forward pass
        y_fnn = fnn_model.forward(x_train)

        # compute loss function (MSE here)
        fnn_loss = torch.mean(((y_fnn)-(y_train))**2)

        # backward pass
        fnn_loss.backward()

        # optimizer step
        fnn_opt.step()

        # record the losses (convert to numpy)
        fnn_loss_train[epoch] = fnn_loss.detach().numpy()

    # compute validation loss
    for n, (x_valid, y_valid) in enumerate(valid_loader):

        # compute the forward pass
        y_fnn = fnn_model.forward(x_valid)

        # compute loss function (MSE here)
        fnn_loss = torch.mean(((y_fnn)-(y_valid))**2)

        # record the losses (convert to numpy)
        fnn_loss_valid[epoch] = fnn_loss.detach().numpy()

    # periodically print out progress
    if np.mod(epoch, epochs//10)==0:
        print('Epoch: ', epoch, "; FNN Train Loss: ', (fnn_loss_train[epoch]))

# Now plot the training losses for each model
plt.figure()
plt.semilogy((fnn_loss_train) , 'k', label='Training Loss')
plt.semilogy((fnn_loss_valid) , 'r', label='Validation Loss')
plt.xlabel('Epoch')
plt.ylabel('MSE')
plt.title('Training Losses for FNN Model')
plt.legend()
plt.show()
```

```
Epoch:  0 ; FNN Train Loss:  496.1931457519531
Epoch:  100 ; FNN Train Loss:  127.239624023475
Epoch:  200 ; FNN Train Loss:  15.67205333097168
Epoch:  300 ; FNN Train Loss:  283.53125
Epoch:  400 ; FNN Train Loss:  208.83087158203125
Epoch:  500 ; FNN Train Loss:  91.6531982421875
Epoch:  600 ; FNN Train Loss:  36.424831390380886
Epoch:  700 ; FNN Train Loss:  248.63739013671875
Epoch:  800 ; FNN Train Loss:  70.8705825805664
Epoch:  900 ; FNN Train Loss:  37.04237365722656
```

In [5]:
```
##---- Part d) Learning curves ----##

lr_array = np.array([1e-3, 3e-3, 1e-2, 3e-1, 1e-1])
fig0, sp0 = plt.subplots(5,1,figsize=(5,15))

for ii in range(np.size(lr_array)):

    lr = lr_array[ii]
    fnn_model = SimpleFNN(hidden, nodes) # define the model

    # define the optimizers (Adam ; lr = learning rate)
    fnn_opt = torch.optim.Adam(fnn_model.parameters(), lr)

    # keep track of losses
    fnn_loss_train = np.zeros((epochs,))
    fnn_loss_valid = np.zeros((epochs,))

    # tell the model we are training
    fnn_model.train()

    for epoch in range(epochs):

        # first compute training loss and step optimizer
        for n, (x_train, y_train) in enumerate(train_loader):

            # clear the gradients
            fnn_model.zero_grad()

            # compute the forward pass
            y_fnn = fnn_model.forward(x_train)

            # compute loss function (MSE here)
            fnn_loss = torch.mean(((y_fnn)-(y_train))**2)

            # backward pass
            fnn_loss.backward()

            # optimizer step
            fnn_opt.step()

            # record the losses (convert to numpy)
            fnn_loss_train[epoch] = fnn_loss.detach().numpy()

        # compute validation loss
        for n, (x_valid, y_valid) in enumerate(valid_loader):

            # compute the forward pass
            y_fnn = fnn_model.forward(x_valid)

            # compute loss function (MSE here)
            fnn_loss = torch.mean(((y_fnn)-(y_valid))**2)

            # record the losses (convert to numpy)
            fnn_loss_valid[epoch] = fnn_loss.detach().numpy()

        # periodically print out progress
#         if np.mod(epoch, epochs//10)==0:
#             print('Epoch: ', epoch, '; FNN Train Loss: ', (fnn_loss_train[epoch]))

    sp0[ii].semilogy((fnn_loss_train) , 'k', label='Training Loss')
    sp0[ii].semilogy((fnn_loss_valid) , 'r', label='Validation Loss')
    sp0[ii].set_ylabel('MSE with Learning Rate' +str(lr),fontsize=10)

sp0[ii].set_xlabel('Epoch ',fontsize=10)
fig0.suptitle('Training Losses for FNN Model',fontsize=12)
sp0[ii].legend(bbox_to_anchor=(1,0), loc="lower right",
               bbox_transform=fig0.transFigure)
plt.show()
```

In [9]:
```
##---- Part e) Run the model on the test data ----##

# Now evaluate our model using test data

lr = 0.01
epochs = 500
# tell our model we are evaluating
fnn_model.eval()

for n, (x_test, y_test) in enumerate(test_loader):

    fnn_pred   = fnn_model(x_test)

plt.figure()
plt.plot(y_test.detach().numpy(), 'k.', label='Test Data')
plt.plot(fnn_pred.detach().numpy(), 'r.', label='FNN Prediction')
plt.xlabel('Model Prediction [10000 USD] ')
plt.ylabel('Iterations')
plt.legend()
plt.title('Model Predictions with Test Data')

plt.show()

MSE = np.mean(((y_test.detach().numpy())-(fnn_pred.detach().numpy()))**2)

print(MSE)
```

```
93.433754
```

In [7]:
```
for name, param in fnn_model.named_parameters():
    if param.requires_grad:
        print(name, param.data)
```

```
linear0.weight tensor([[ 0.0181,  0.0515,  0.0746,  0.6944, -0.2683,  0.4724,  0.0152, -0.6295,
         0.0935,  0.0090, -0.0436,  0.3698, -0.5104]])
linear0.bias tensor([22.2038])
```

In [ ]:
```

```