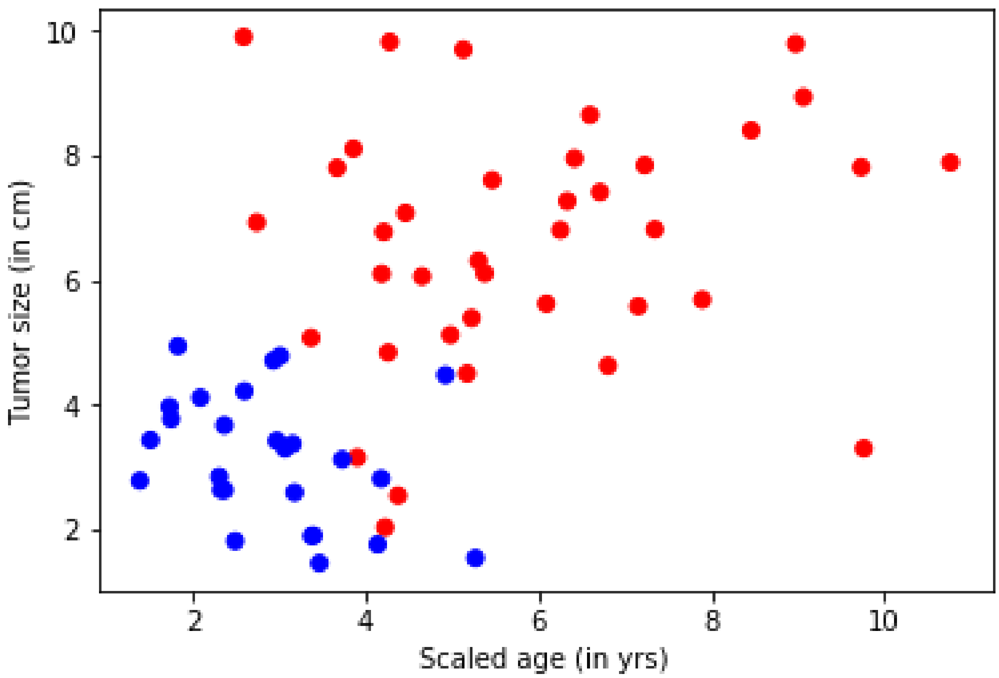
FeedForward Network

August 22, 2017

```
In [1]: from IPython.display import Image

In [2]: # Figure 1
        Image(url="https://www.cntk.ai/jup/cancer_data_plot.jpg", width=400, height=400)

Out[2]: <IPython.core.display.Image object>

In [3]: # Figure 2
        Image(url="https://upload.wikimedia.org/wikipedia/en/5/54/Feed_forward_neural_net.gif",

Out[3]: <IPython.core.display.Image object>

In [4]: from __future__ import print_function # Use a function definition from future version (s
        import matplotlib.pyplot as plt
        %matplotlib inline

        import numpy as np
        import sys
        import os

        import cntk as C
        import cntk.tests.test_utils

In [5]: np.random.seed(0)

        # Define the data dimensions
        input_dim = 2
        num_output_classes = 2

In [6]: def generate_random_data_sample(sample_size, feature_dim, num_classes):
            # Create synthetic data using NumPy.
            Y = np.random.randint(size=(sample_size, 1), low=0, high=num_classes)

            # Make sure that the data is separable
            X = (np.random.randn(sample_size, feature_dim)+3) * (Y+1)
            X = X.astype(np.float32)
            # converting class 0 into the vector "1 0 0",
            # class 1 into vector "0 1 0", ...
            class_ind = [Y==class_number for class_number in range(num_classes)]
            Y = np.asarray(np.hstack(class_ind), dtype=np.float32)
            return X, Y
```

In [7]:
```
mysamplesize = 64
features, labels = generate_random_data_sample(mysamplesize, input_dim, num_output_class
```

In [8]:
```
import matplotlib.pyplot as plt
%matplotlib inline

# given this is a 2 class
colors = ['r' if l == 0 else 'b' for l in labels[:,0]]

plt.scatter(features[:,0], features[:,1], c=colors)
plt.xlabel("Scaled age (in yrs)")
plt.ylabel("Tumor size (in cm)")
plt.show()
```



In [9]:
```
Image(url="http://cntk.ai/jup/feedforward_network.jpg", width=200, height=200)
```

Out[9]: <IPython.core.display.Image object>

In [10]:
```
num_hidden_layers = 2
hidden_layers_dim = 50
```

In [11]:
```
# The input variable (representing 1 observation, in our example of age and size) x, wh
# in this case has a dimension of 2.
#
# The label variable has a dimensionality equal to the number of output classes in our
```

```
input = C.input_variable(input_dim)
label = C.input_variable(num_output_classes)
```

In [12]:
```
def linear_layer(input_var, output_dim):
    input_dim = input_var.shape[0]

    weight = C.parameter(shape=(input_dim, output_dim))
    bias = C.parameter(shape=(output_dim))

    return bias + C.times(input_var, weight)
```

In [13]:
```
def dense_layer(input_var, output_dim, nonlinearity):
    l = linear_layer(input_var, output_dim)

    return nonlinearity(l)
```

In [14]:
```
# Define a multilayer feedforward classification model
def fully_connected_classifier_net(input_var, num_output_classes, hidden_layer_dim,
                                   num_hidden_layers, nonlinearity):

    h = dense_layer(input_var, hidden_layer_dim, nonlinearity)
    for i in range(1, num_hidden_layers):
        h = dense_layer(h, hidden_layer_dim, nonlinearity)

    return linear_layer(h, num_output_classes)
```

In [15]:
```
# Create the fully connected classfier
z = fully_connected_classifier_net(input, num_output_classes, hidden_layers_dim,
                                   num_hidden_layers, C.sigmoid)
```

In [16]:
```
def create_model(features):
    with C.layers.default_options(init=C.layers.glorot_uniform(), activation=C.sigmoid)
        h = features
        for _ in range(num_hidden_layers):
            h = C.layers.Dense(hidden_layers_dim)(h)
        last_layer = C.layers.Dense(num_output_classes, activation = None)

        return last_layer(h)

z = create_model(input)
```

In [17]:
```
loss = C.cross_entropy_with_softmax(z, label)
```

In [18]:
```
eval_error = C.classification_error(z, label)
```

In [19]:
```
# Instantiate the trainer object to drive the model training
learning_rate = 0.5
lr_schedule = C.learning_rate_schedule(learning_rate, C.UnitType.minibatch)
learner = C.sgd(z.parameters, lr_schedule)
trainer = C.Trainer(z, (loss, eval_error), [learner])
```

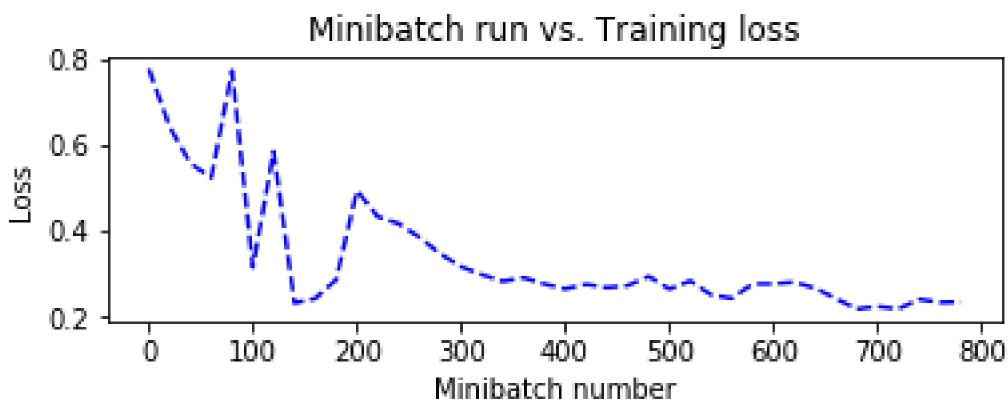
In [20]:
```
# Define a utility function to compute the moving average sum.
# A more efficient implementation is possible with np.cumsum() function
def moving_average(a, w=10):
    if len(a) < w:
        return a[:]    # Need to send a copy of the array
    return [val if idx < w else sum(a[(idx-w):idx])/w for idx, val in enumerate(a)]


# Defines a utility that prints the training progress
def print_training_progress(trainer, mb, frequency, verbose=1):
    training_loss = "NA"
    eval_error = "NA"

    if mb%frequency == 0:
        training_loss = trainer.previous_minibatch_loss_average
        eval_error = trainer.previous_minibatch_evaluation_average
        if verbose:
            print ("Minibatch: {}, Train Loss: {}, Train Error: {}".format(mb, training

    return mb, training_loss, eval_error
```

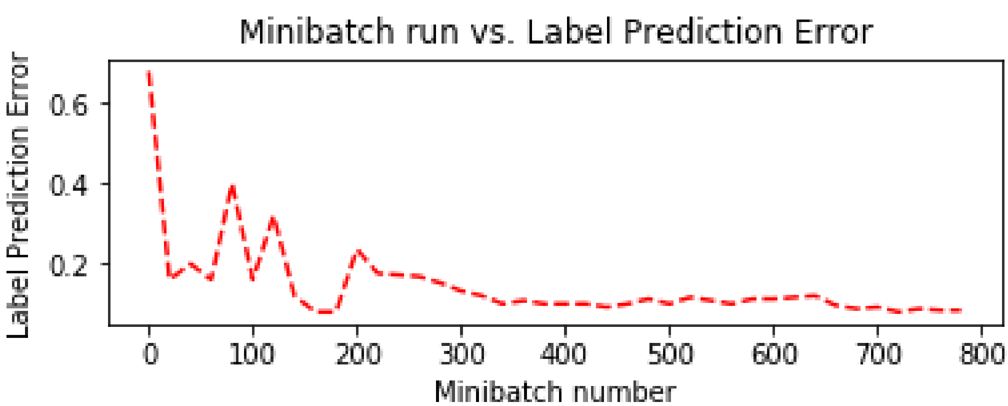
In [21]:
```
# Initialize the parameters for the trainer
minibatch_size = 25
num_samples = 20000
num_minibatches_to_train = num_samples / minibatch_size
```

In [22]:
```
# Run the trainer and perform model training
training_progress_output_freq = 20

plotdata = {"batchsize":[], "loss":[], "error":[]}

for i in range(0, int(num_minibatches_to_train)):
    features, labels = generate_random_data_sample(minibatch_size, input_dim, num_outpu

    # Specify the input variables mapping in the model to actual minibatch data for tra
    trainer.train_minibatch({input : features, label : labels})
    batchsize, loss, error = print_training_progress(trainer, i,
                                                     training_progress_output_freq, ver

    if not (loss == "NA" or error =="NA"):
        plotdata["batchsize"].append(batchsize)
        plotdata["loss"].append(loss)
        plotdata["error"].append(error)
```

In [23]:
```
# Compute the moving average loss to smooth out the noise in SGD
plotdata["avgloss"] = moving_average(plotdata["loss"])
plotdata["avgerror"] = moving_average(plotdata["error"])
```

```python
# Plot the training loss and the training error
import matplotlib.pyplot as plt

plt.figure(1)
plt.subplot(211)
plt.plot(plotdata["batchsize"], plotdata["avgloss"], 'b--')
plt.xlabel('Minibatch number')
plt.ylabel('Loss')
plt.title('Minibatch run vs. Training loss')

plt.show()

plt.subplot(212)
plt.plot(plotdata["batchsize"], plotdata["avgerror"], 'r--')
plt.xlabel('Minibatch number')
plt.ylabel('Label Prediction Error')
plt.title('Minibatch run vs. Label Prediction Error')
plt.show()
```





```
In [24]: # Generate new data
         test_minibatch_size = 25
         features, labels = generate_random_data_sample(test_minibatch_size, input_dim, num_outp

         trainer.test_minibatch({input : features, label : labels})
```

```
Out[24]: 0.12
```

```
In [25]: Image(url="http://cntk.ai/jup/feedforward_network.jpg", width=200, height=200)
```

```
Out[25]: <IPython.core.display.Image object>
```

```
In [26]: out = C.softmax(z)
```

```
In [27]: predicted_label_probs = out.eval({input : features})
```

```
In [28]: print("Label    :", [np.argmax(label) for label in labels])
         print("Predicted:", [np.argmax(row) for row in predicted_label_probs])
```

```
Label    : [1, 0, 0, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 0, 1, 0, 1, 1, 0, 0, 1, 0, 0, 0, 1]
Predicted: [1, 0, 0, 0, 0, 0, 0, 1, 1, 0, 1, 1, 1, 0, 1, 0, 1, 1, 0, 0, 1, 0, 0, 0, 1]
```

```
In [ ]:
```