In [3]:
```
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
import scipy.misc
import glob
import sys
# you shouldn't need to make any more imports
np.seterr(all='raise')
```

Out[3]:
```
{'divide': 'raise', 'invalid': 'raise', 'over': 'raise', 'under': 'rais
e'}
```

In [ ]:

```
class NeuralNetwork(object):
    """
    Abstraction of neural network.
    Stores parameters, activations, cached values.
    Provides necessary functions for training and prediction.
    """

    def __init__(self, layer_dimensions, drop_prob=0.0, reg_lambda=0.0, mome
        """
        Initializes the weights and biases for each layer
        :param layer_dimensions: (list) number of nodes in each layer
        :param drop_prob: drop probability for dropout layers. Only requirec
        :param reg_lambda: regularization parameter. Only required in part 2
        """
        if seed is None:
            seed = np.random.randint(0, 4294967295)
        np.random.seed(seed)
        print("seed = %d" % seed)

        self.parameters = {}
        self.num_layers = len(layer_dimensions)
        self.drop_prob = drop_prob
        self.reg_lambda = reg_lambda

        # init parameters
        self.parameters['w'] = []
        self.parameters['b'] = []
        self.parameters['m_w'] = []
        self.parameters['m_b'] = []

        for li in range(len(layer_dimensions) - 1):
            ls1 = layer_dimensions[li]
            ls2 = layer_dimensions[li + 1]
            init_stddev = 1.0 / np.sqrt(ls1)
            self.parameters['w'].append(init_stddev * np.random.randn(ls2, l
            self.parameters['b'].append(init_stddev * np.random.randn(ls2, 1
            self.parameters['m_w'].append(np.zeros([ls2, ls1]))
            self.parameters['m_b'].append(np.zeros([ls2, 1]))

        # Used for convergence plots
        self.n_iters_base = 0 # So I can call train multiple times and keep
        self.costs = []
        self.cost_times = []
        self.val_costs = []

        self.momentum = momentum

    def affineForward(self, A, W, b):
        """
        Forward pass for the affine layer.
        :param A: input matrix, shape (L, S), where L is the number of hidde
        the number of samples
        :returns: the affine product WA + b, along with the cache required f
        """
        return W @ A + b
```

```
    def activationForward(self, A, activation="relu"):
        """
        Common interface to access all activation functions.
        :param A: input to the activation function
        :param prob: activation funciton to apply to A. Just "relu" for this
        :param activation: this is here for no good reason
        :returns: activation(A)
        """
        assert activation == "relu", "Only relu activation implemented"
        return self.relu(A)

    def relu(self, X):
        return np.maximum(0, X)

    def forwardPropagation(self, X):
        """
        Runs an input X through the neural network to compute activations
        for all layers. Returns the output computed at the last layer along
        with the cache required for backpropagation.
        :returns: (tuple) A_l, cache
            WHERE
            A_l is activation of last layer
            cache is cached values for each layer that
                     are needed in further steps
        """
        A_l = np.array(X)
        cache = {'z': [], 'a': [np.array(X)]}
        for li in range(self.num_layers - 1):
            Z_l = self.affineForward(A_l, self.parameters['w'][li], self.par
            if li == self.num_layers - 2:
                A_l = self.softmax(Z_l)
            else:
                A_l = self.activationForward(Z_l)
                A_l = self.dropout(A_l, self.drop_prob)  # no-op if dropout
            cache['z'].append(np.array(Z_l))
            cache['a'].append(np.array(A_l))
        return A_l, cache

    def softmax(self, AL):
        #return np.exp(AL-np.max(AL)) / np.exp(AL-np.max(AL)).sum(axis=0)
        expAL = np.exp(AL)
        return expAL / expAL.sum(axis=0)

    def softmax_derivative(self, z):
        # sm = self.softmax(z)
        # return sm * (1.0 - sm)
        return z

    def softmax_backwards(self, dLdA, z):
        return dLdA * self.softmax_derivative(z)

    def costFunction(self, AL, y):
        """
        :param AL: Activation of last layer, shape (num_classes, S)
        :param y: labels, shape (S)
        :param alpha: regularization parameter
        :returns cost, dAL: A scalar denoting cost and the gradient of cost
```

```python
        """
        # Cross-entropy cost:
        # compute loss
        cost = 0

        # Used a numerical stability trick here in the softmax to control e
        # softmax_AL = np.exp(AL-np.max(AL))/np.exp(AL-np.max(AL)).sum(axis=

        #Computing the cross entropy:
        #There's probably a much smarter way of doing this vectorized:
        for a, img in enumerate(AL.T): # Get the softmaxed image
            for b, node in enumerate(img): # Loop over elements of image
                if y[a] == b:
                    y_t = 1
                else:
                    y_t = 0
            cost += -y_t*np.log(img[b])

        S = AL.shape[1]
        cost = cost / S

        # L2-regularization:
        if self.reg_lambda > 0:
            l2_reg = self.reg_lambda * np.array([np.sum(i.flatten()**2) for
            cost += 0.5 * l2_reg / S

        # Cross-entropy derivative: subtracting 1 from the correct class
        # There's probably a much smarter way of doing this vectorized:

        #dLdA = np.zeros(softmax_AL.shape) # Just for shape.
        dLdA = AL.copy() # Just for shape.

        # Same deal: can probably vectorize this part:
        for a in range(0, AL.shape[1]): # loop through softmaxed images
            for b in range(0, AL.shape[0]): # loop through elements of soft
                if y[a] == b:
                    y_t = 1
                else:
                    y_t = 0
                dLdA[b,a] -= y_t

        return cost, dLdA

    def affineBackward(self, dLdZ, cache):
        """
        Backward pass for the affine layer.
        :param dA_prev: gradient from the next layer.
        :param cache: The parameters w and a for the layerbackstr
        :returns dA: gradient on the input to this layer
                 dW: gradient on the weights
                 db: gradient on the bias
        """
        w, a = cache  # w is mxl a is lx1
        dLdw = dLdZ @ a.T
        dLdb = dLdZ @ np.ones([a.shape[1], 1])
        dLdA_nxt = w.T @ dLdZ
```

```
        return dLdA_nxt, dLdw, dLdb

    def activationBackward(self, dLdA, z, activation="relu"):
        """
        Interface to call backward on activation functions.
        In this case, it's just relu.
        """

        assert activation == "relu", "Only relu activation implemented...."
        dAdZ = self.relu_derivative(z, None)
        dLdZ = dLdA * dAdZ

        return dLdZ

    def relu_derivative(self, z, cached_x):
        """
        :param z: The value to evaluate the relu derivative at
        :param cached_x: This is here for no reason
        :return: The relu derivative
        """
        return np.array(z > 0, dtype=float)

    def backPropagation(self, dAL, Yasdfdasfsd, cache):
        """
        Run backpropagation to compute gradients on all paramters in the moc
        :param dAL: gradient on the last layer of the network. Returned by t
        :param Y: labels
        :param cache: cached values during forwardprop
        :returns gradients: dW and db for each weight/bias
        """
        gradients = {'w': [], 'b': []}

        dLdA = dAL

        for l in range(self.num_layers - 2, -1, -1):
            if l == self.num_layers-2:
                dLdZ = self.softmax_backwards(dLdA, cache['z'][l])
            else:
                dLdZ = self.activationBackward(dLdA, cache['z'][l])

            dLdA, dLdW, dLdb = self.affineBackward(dLdZ, (self.parameters['v
            gradients['w'].append(np.array(dLdW))
            gradients['b'].append(np.array(dLdb))

            if self.drop_prob > 0:
                # Not necessary.........
                dLdA = self.dropout_backward(dLdA, cache)

        return gradients

    def updateParameters(self, gradients, alpha):
        """
        :param gradients: gradients for each weight/bias
        :param alpha: step size for gradient descent
        """
        for i in range(len(self.parameters['w'])):
            Jw = gradients['w'][-1 - i]
```

```
            Jb = gradients['b'][-i - 1]

            w = self.parameters['w'][i]
            b = self.parameters['b'][i]
            m_w = self.parameters['m_w'][i]
            m_b = self.parameters['m_b'][i]

            self.parameters['m_w'][i] = self.momentum * m_w - alpha * (Jw +
            self.parameters['m_b'][i] = self.momentum * m_b - alpha * (Jb +
            self.parameters['w'][i] += self.parameters['m_w'][i]
            self.parameters['b'][i] += self.parameters['m_b'][i]

    def dropout_backward(self, dA, cache):
        # Why?
        return dA

    def dropout(self, A, p):
        """
        :param A: Activation
        :param prob: drop prob
        :returns: tuple (A, M)
            WHERE
            A is matrix after applying dropout
            M is dropout mask, used in the backward pass
        """
        if p == 0:
            return A
        mask = np.random.rand(*A.shape)
        mask = (mask > p).astype(float)
        mask /= (1-p)
        A *= mask
        return A

    def train(self, X, y, iters=1000, alpha=0.0001, batch_size=100, print_ev
        """
        :param X: input samples, each column is a sample
        :param y: labels for input samples, y.shape[0] must equal X.shape[1
        :param iters: number of training iterations
        :param alpha: step size for gradient descent
        :param batch_size: number of samples in a minibatch
        :param print_every: no. of iterations to print debug info after
        """


        # A hack because of the interface of this function
        x_val = X[:, 45000:50000]
        y_val = y[45000:50000]
        X_train = X[:, 0:45000]
        y_train = y[0:45000]

        for i in range(0, iters):
            X_batch, Y_batch = self.get_batch(X_train, y_train, batch_size)

            # forward prop
            A_l, cache = self.forwardPropagation(X_batch)

            # compute loss
```

```
            cost, dLdA = self.costFunction(A_l, Y_batch)

            # compute gradients
            gradients = self.backPropagation(dLdA, y_train, cache)

            # update weights and biases based on gradient
            self.updateParameters(gradients, alpha)

            if i % print_every == 0:
                # print cost, train and validation set accuracies

                self.cost_times.append(self.n_iters_base + i)
                pred_class_batch = np.argmax(A_l, axis=0)
                n_correct_batch = np.sum(np.array((pred_class_batch - Y_batc
                Aval = self.predict(x_val)
                n_correct_val = np.sum(np.array((Aval - y_val) == 0, dtype=i
                self.val_costs.append(n_correct_val/len(y_val))
                self.costs.append(n_correct_batch/batch_size)
                #n_correct = np.sum(np.array(pred_class - y_class == 0, dtyp
                print("(%d/%d) cost = %f, batch_correct = %0.3f, val correct
                                                                      self.n_i
                                                                      cost, n_
                                                                      n_correc

        self.n_iters_base += iters

    def predict(self, X):
        """
        Make predictions for each sample
        """
        y, _ = self.forwardPropagation(X)
        return np.argmax(y, axis=0)

    def get_batch(self, X, y, batch_size):
        """
        Return minibatch of samples and labels

        :param X, y: samples and corresponding labels
        :parma batch_size: minibatch size
        :returns: (tuple) X_batch, y_batch
        """

        idx = np.random.choice(X.shape[1], batch_size, replace=False)
        #return X[:, idx], y[:, idx]
        return X[:, idx], y[idx]
```

In [5]:
```
# Helper functions, DO NOT modify this

def get_img_array(path):
    """
    Given path of image, returns it's numpy array
    """
    return scipy.misc.imread(path)

def get_files(folder):
    """
    Given path to folder, returns list of files in it
    """
    filenames = [file for file in glob.glob(folder+'*/*')]
    filenames.sort()
    return filenames

def get_label(filepath, label2id):
    """
    Files are assumed to be labeled as: /path/to/file/999_frog.png
    Returns label for a filepath
    """
    tokens = filepath.split('/')
    label = tokens[-1].split('_')[1][:-4]
    if label in label2id:
        return label2id[label]
    else:
        sys.exit("Invalid label: " + label)
```

In [7]:
```python
# Functions to load data, DO NOT change these

def get_labels(folder, label2id):
    """
    Returns vector of labels extracted from filenames of all files in folder
    :param folder: path to data folder
    :param label2id: mapping of text labels to numeric ids. (Eg: automobile
    """
    files = get_files(folder)
    y = []
    for f in files:
        y.append(get_label(f,label2id))
    return np.array(y)

def one_hot(y, num_classes=10):
    """
    Converts each label index in y to vector with one_hot encoding
    """
    y_one_hot = np.zeros((num_classes, y.shape[0]))
    y_one_hot[y, range(y.shape[0])] = 1
    return y_one_hot

def get_label_mapping(label_file):
    """
    Returns mappings of label to index and index to label
    The input file has list of labels, each on a separate line.
    """
    with open(label_file, 'r') as f:
        id2label = f.readlines()
        id2label = [l.strip() for l in id2label]
    label2id = {}
    count = 0
    for label in id2label:
        label2id[label] = count
        count += 1
    return id2label, label2id

def get_images(folder):
    """
    returns numpy array of all samples in folder
    each column is a sample resized to 30x30 and flattened
    """
    files = get_files(folder)
    images = []
    count = 0

    for f in files:
        count += 1
        if count % 10000 == 0:
            print("Loaded {}/{}".format(count,len(files)))
        img_arr = get_img_array(f)
        img_arr = img_arr.flatten() / 255.0
        images.append(img_arr)
    X = np.column_stack(images)

    return X
```

```python

def get_train_data(data_root_path):
    """
    Return X and y
    """
    train_data_path = data_root_path + 'train'
    id2label, label2id = get_label_mapping(data_root_path+'labels.txt')
    print(label2id)
    X = get_images(train_data_path)
    y = get_labels(train_data_path, label2id)
    return X, y

def save_predictions(filename, y):
    """
    Dumps y into .npy file
    """
    np.save(filename, y)
```

In [8]:
```python
# Load the data
data_root_path = './cifar10-hw1/'
X_train, y_train = get_train_data(data_root_path) # this may take a few minu
X_test = get_images(data_root_path + 'test')
print('Data loading done')
```

```
{'airplane': 0, 'automobile': 1, 'bird': 2, 'cat': 3, 'deer': 4, 'dog':
5, 'frog': 6, 'horse': 7, 'ship': 8, 'truck': 9}

/home/francis/miniconda3/envs/csgy_9223_hw1/lib/python3.6/site-packages/i
pykernel_launcher.py:7: DeprecationWarning: `imread` is deprecated!
`imread` is deprecated in SciPy 1.0.0, and will be removed in 1.2.0.
Use ``imageio.imread`` instead.
  import sys

Loaded 10000/50000
Loaded 20000/50000
Loaded 30000/50000
Loaded 40000/50000
Loaded 50000/50000
Loaded 10000/10000
Data loading done
```

Part 1

Simple fully-connected deep neural network

In [100]:

```
layer_dimensions = [X_train.shape[0], 512, 256, 128, 10]
NN = NeuralNetwork(layer_dimensions, reg_lambda=0.0, drop_prob=0.0)
NN.train(X_train, y_train, iters=18000, alpha=1e-4, batch_size=500, print_ev
```

```
(5150/18000) cost = 0.112301, batch_correct = 0.420, val correct = 0.4398
000000
(5200/18000) cost = 0.077834, batch_correct = 0.432, val correct = 0.4302
000000
(5250/18000) cost = 0.240752, batch_correct = 0.426, val correct = 0.4166
000000
(5300/18000) cost = 0.126523, batch_correct = 0.412, val correct = 0.4260
000000
(5350/18000) cost = 0.130249, batch_correct = 0.472, val correct = 0.4496
000000
(5400/18000) cost = 0.123636, batch_correct = 0.420, val correct = 0.4428
000000
(5450/18000) cost = 0.123545, batch_correct = 0.494, val correct = 0.4330
000000
(5500/18000) cost = 0.123790, batch_correct = 0.478, val correct = 0.4282
000000
(5550/18000) cost = 0.119662, batch_correct = 0.494, val correct = 0.4500
```

In [101]:
```
NN.train(X_train, y_train, iters=3000, alpha=1e-4, batch_size=500, print_eve
```

```
(18000/21000) cost = 0.104659, batch_correct = 0.552, val correct = 0.497
6000000
(18050/21000) cost = 0.114603, batch_correct = 0.568, val correct = 0.516
6000000
(18100/21000) cost = 0.146467, batch_correct = 0.550, val correct = 0.512
6000000
(18150/21000) cost = 0.086795, batch_correct = 0.578, val correct = 0.489
2000000
(18200/21000) cost = 0.107734, batch_correct = 0.584, val correct = 0.506
4000000
(18250/21000) cost = 0.101764, batch_correct = 0.574, val correct = 0.493
8000000
(18300/21000) cost = 0.098303, batch_correct = 0.526, val correct = 0.482
0000000
(18350/21000) cost = 0.109158, batch_correct = 0.572, val correct = 0.500
8000000
(18400/21000) cost = 0.103981, batch_correct = 0.604, val correct = 0.512
6000000
(18450/21000) cost = 0.097944, batch_correct = 0.642, val correct = 0.517
0000000
(18500/21000) cost = 0.126720, batch_correct = 0.576, val correct = 0.499
6000000
(18550/21000) cost = 0.049877, batch_correct = 0.576, val correct = 0.517
2000000
(18600/21000) cost = 0.113175, batch_correct = 0.562, val correct = 0.514
8000000
(18650/21000) cost = 0.108529, batch_correct = 0.610, val correct = 0.499
8000000
(18700/21000) cost = 0.102782, batch_correct = 0.556, val correct = 0.511
0000000
(18750/21000) cost = 0.130602, batch_correct = 0.586, val correct = 0.509
2000000
(18800/21000) cost = 0.176462, batch_correct = 0.556, val correct = 0.481
0000000
(18850/21000) cost = 0.094050, batch_correct = 0.614, val correct = 0.504
6000000
(18900/21000) cost = 0.179748, batch_correct = 0.534, val correct = 0.483
8000000
(18950/21000) cost = 0.079439, batch_correct = 0.596, val correct = 0.511
8000000
(19000/21000) cost = 0.124904, batch_correct = 0.560, val correct = 0.522
8000000
(19050/21000) cost = 0.081473, batch_correct = 0.548, val correct = 0.510
6000000
(19100/21000) cost = 0.177195, batch_correct = 0.516, val correct = 0.507
2000000
(19150/21000) cost = 0.122345, batch_correct = 0.578, val correct = 0.511
2000000
(19200/21000) cost = 0.087160, batch_correct = 0.612, val correct = 0.519
0000000
(19250/21000) cost = 0.129303, batch_correct = 0.580, val correct = 0.512
6000000
(19300/21000) cost = 0.110915, batch_correct = 0.572, val correct = 0.503
4000000
(19350/21000) cost = 0.117323, batch_correct = 0.564, val correct = 0.512
```

```
4000000
(19400/21000) cost = 0.148524, batch_correct = 0.554, val correct = 0.510
0000000
(19450/21000) cost = 0.094645, batch_correct = 0.598, val correct = 0.517
8000000
(19500/21000) cost = 0.097556, batch_correct = 0.580, val correct = 0.503
4000000
(19550/21000) cost = 0.135069, batch_correct = 0.556, val correct = 0.508
0000000
(19600/21000) cost = 0.091380, batch_correct = 0.564, val correct = 0.507
0000000
(19650/21000) cost = 0.136606, batch_correct = 0.562, val correct = 0.517
8000000
(19700/21000) cost = 0.172717, batch_correct = 0.596, val correct = 0.508
0000000
(19750/21000) cost = 0.163416, batch_correct = 0.606, val correct = 0.511
6000000
(19800/21000) cost = 0.164275, batch_correct = 0.554, val correct = 0.517
2000000
(19850/21000) cost = 0.093963, batch_correct = 0.592, val correct = 0.519
8000000
(19900/21000) cost = 0.073295, batch_correct = 0.598, val correct = 0.481
8000000
(19950/21000) cost = 0.095730, batch_correct = 0.548, val correct = 0.505
2000000
(20000/21000) cost = 0.076113, batch_correct = 0.568, val correct = 0.514
4000000
(20050/21000) cost = 0.146062, batch_correct = 0.536, val correct = 0.506
6000000
(20100/21000) cost = 0.109685, batch_correct = 0.616, val correct = 0.517
8000000
(20150/21000) cost = 0.129260, batch_correct = 0.570, val correct = 0.514
4000000
(20200/21000) cost = 0.093769, batch_correct = 0.578, val correct = 0.504
0000000
(20250/21000) cost = 0.061682, batch_correct = 0.560, val correct = 0.498
4000000
(20300/21000) cost = 0.073562, batch_correct = 0.594, val correct = 0.517
2000000
(20350/21000) cost = 0.093059, batch_correct = 0.636, val correct = 0.510
0000000
(20400/21000) cost = 0.168235, batch_correct = 0.536, val correct = 0.513
8000000
(20450/21000) cost = 0.195782, batch_correct = 0.582, val correct = 0.498
8000000
(20500/21000) cost = 0.124246, batch_correct = 0.558, val correct = 0.494
6000000
(20550/21000) cost = 0.113496, batch_correct = 0.598, val correct = 0.511
6000000
(20600/21000) cost = 0.109927, batch_correct = 0.608, val correct = 0.503
8000000
(20650/21000) cost = 0.129113, batch_correct = 0.492, val correct = 0.497
6000000
(20700/21000) cost = 0.119117, batch_correct = 0.608, val correct = 0.513
6000000
(20750/21000) cost = 0.105020, batch_correct = 0.586, val correct = 0.517
0000000
```

```
(20800/21000) cost = 0.112805, batch_correct = 0.550, val correct = 0.501
0000000
(20850/21000) cost = 0.093397, batch_correct = 0.586, val correct = 0.518
6000000
(20900/21000) cost = 0.099290, batch_correct = 0.576, val correct = 0.516
6000000
(20950/21000) cost = 0.147827, batch_correct = 0.546, val correct = 0.511
6000000
```

In [103]:

```
y_predicted = NN.predict(X_test)
save_predictions('ans1-fw710', y_predicted)
```

In [104]:

```
# test if your numpy file has been saved correctly
loaded_y = np.load('ans1-fw710.npy')
print(loaded_y.shape)
loaded_y[:10]
```

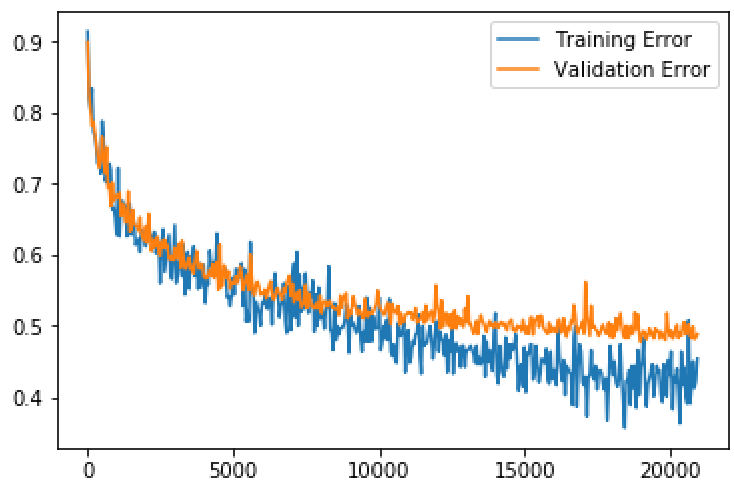
```
(10000,)
```

Out[104]:

```
array([3, 9, 0, 5, 5, 0, 6, 7, 8, 1])
```

In [105]:

```
plt.plot(NN.cost_times, 1.0 - np.array(NN.costs), label="Training Error")
plt.plot(NN.cost_times, 1.0 - np.array(NN.val_costs), label="Validation Erro
plt.legend(loc="best")
plt.show()
```



Part 2: Regularizing the neural network

Add dropout and L2 regularization

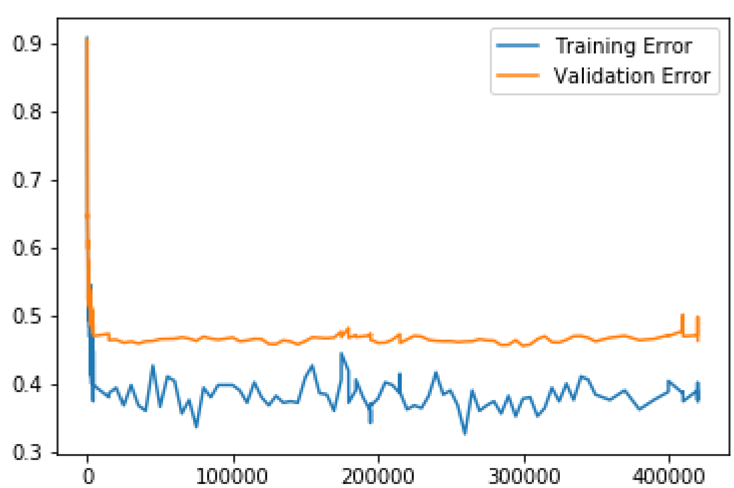
In [11]:
```
layer_dimensions = [X_train.shape[0], 256, 256, 128, 10]
NN2 = NeuralNetwork(layer_dimensions, drop_prob=0.0, reg_lambda=0.1, momentu
NN2.train(X_train, y_train, iters=15000, alpha=0.0001, batch_size=500, print
```

```
seed = 3353606490
(0/15000) cost = 0.325139, batch_correct = 0.092, val correct = 0.0944000
000
(50/15000) cost = 0.245757, batch_correct = 0.234, val correct = 0.222600
0000
(100/15000) cost = 0.224272, batch_correct = 0.250, val correct = 0.25940
00000
(150/15000) cost = 0.253618, batch_correct = 0.312, val correct = 0.33600
00000
(200/15000) cost = 0.221191, batch_correct = 0.356, val correct = 0.35680
00000
(250/15000) cost = 0.225849, batch_correct = 0.366, val correct = 0.35080
00000
(300/15000) cost = 0.175539, batch_correct = 0.414, val correct = 0.38360
00000
(350/15000) cost = 0.184628, batch_correct = 0.372, val correct = 0.39020
00000
(400/15000) cost = 0.204121, batch_correct = 0.412, val correct = 0.40220
00000
```

In [89]:
```
# I ran this a bunch of times with a smaller learning rate (I was too lazy t
NN2.train(X_train, y_train, iters=10000, alpha=5e-5, batch_size=500, print_e
```

```
(420000/430000) cost = 0.134398, batch_correct = 0.610, val correct = 0.5
296000000
(420100/430000) cost = 0.151494, batch_correct = 0.624, val correct = 0.5
284000000
(420200/430000) cost = 0.111439, batch_correct = 0.598, val correct = 0.5
020000000
(420300/430000) cost = 0.139377, batch_correct = 0.626, val correct = 0.5
372000000
```

In [91]:
```
plt.plot(NN2.cost_times, 1.0 - np.array(NN2.costs), label="Training Error")
plt.plot(NN2.cost_times, 1.0 - np.array(NN2.val_costs), label="Validation Er
plt.legend(loc="best")
plt.show()
```



In [108]:
```
y_predicted2 = NN4.predict(X_test)
save_predictions('ans2-fw710', y_predicted2)
```