softmax

January 27, 2021

0.1 This is the softmax workbook for ECE C147/C247 Assignment #2

Please follow the notebook linearly to implement a softmax classifier.

Please print out the workbook entirely when completed.

We thank Serena Yeung & Justin Johnson for permission to use code written for the CS 231n class (cs231n.stanford.edu). These are the functions in the cs231n folders and code in the jupyer notebook to preprocess and show the images. The classifiers used are based off of code prepared for CS 231n as well.

The goal of this workbook is to give you experience with training a softmax classifier.

[1]:
```
import random
import numpy as np
from cs231n.data_utils import load_CIFAR10
import matplotlib.pyplot as plt

%matplotlib inline
%load_ext autoreload
%autoreload 2
```

[3]:
```
def get_CIFAR10_data(num_training=49000, num_validation=1000, num_test=1000,␣
 ↪num_dev=500):
    """
    Load the CIFAR-10 dataset from disk and perform preprocessing to prepare
    it for the linear classifier. These are the same steps as we used for the
    SVM, but condensed to a single function.
    """
    # Load the raw CIFAR-10 data
    cifar10_dir = 'C:/Users/Ashwin/Desktop/UCLA/current classes/ece 247/hw2/
 ↪hw2_code/cifar-10-batches-py/' # You need to update this line
    X_train, y_train, X_test, y_test = load_CIFAR10(cifar10_dir)

    # subsample the data
    mask = list(range(num_training, num_training + num_validation))
    X_val = X_train[mask]
    y_val = y_train[mask]
    mask = list(range(num_training))
    X_train = X_train[mask]
```

```python
    y_train = y_train[mask]
    mask = list(range(num_test))
    X_test = X_test[mask]
    y_test = y_test[mask]
    mask = np.random.choice(num_training, num_dev, replace=False)
    X_dev = X_train[mask]
    y_dev = y_train[mask]

    # Preprocessing: reshape the image data into rows
    X_train = np.reshape(X_train, (X_train.shape[0], -1))
    X_val = np.reshape(X_val, (X_val.shape[0], -1))
    X_test = np.reshape(X_test, (X_test.shape[0], -1))
    X_dev = np.reshape(X_dev, (X_dev.shape[0], -1))

    # Normalize the data: subtract the mean image
    mean_image = np.mean(X_train, axis = 0)
    X_train -= mean_image
    X_val -= mean_image
    X_test -= mean_image
    X_dev -= mean_image

    # add bias dimension and transform into columns
    X_train = np.hstack([X_train, np.ones((X_train.shape[0], 1))])
    X_val = np.hstack([X_val, np.ones((X_val.shape[0], 1))])
    X_test = np.hstack([X_test, np.ones((X_test.shape[0], 1))])
    X_dev = np.hstack([X_dev, np.ones((X_dev.shape[0], 1))])

    return X_train, y_train, X_val, y_val, X_test, y_test, X_dev, y_dev


# Invoke the above function to get our data.
X_train, y_train, X_val, y_val, X_test, y_test, X_dev, y_dev =␣
 ↪get_CIFAR10_data()
print('Train data shape: ', X_train.shape)
print('Train labels shape: ', y_train.shape)
print('Validation data shape: ', X_val.shape)
print('Validation labels shape: ', y_val.shape)
print('Test data shape: ', X_test.shape)
print('Test labels shape: ', y_test.shape)
print('dev data shape: ', X_dev.shape)
print('dev labels shape: ', y_dev.shape)
```

```
Train data shape:  (49000, 3073)
Train labels shape:  (49000,)
Validation data shape:  (1000, 3073)
Validation labels shape:  (1000,)
Test data shape:  (1000, 3073)
Test labels shape:  (1000,)
dev data shape:  (500, 3073)
dev labels shape:  (500,)
```

0.2 Training a softmax classifier.

The following cells will take you through building a softmax classifier. You will implement its loss function, then subsequently train it with gradient descent. Finally, you will choose the learning rate of gradient descent to optimize its classification performance.

[4]:
```
from nndl import Softmax
```

[5]:
```
# Declare an instance of the Softmax class.
# Weights are initialized to a random value.
# Note, to keep people's first solutions consistent, we are going to use a␣
 ↪random seed.

np.random.seed(1)

num_classes = len(np.unique(y_train))
num_features = X_train.shape[1]

softmax = Softmax(dims=[num_classes, num_features])
```

Softmax loss

[19]:
```
## Implement the loss function of the softmax using a for loop over
#   the number of examples

loss = softmax.loss(X_train, y_train)
```

[20]:
```
print(loss)
```

```
2.3277607028048966
```

0.3 Question:

You'll notice the loss returned by the softmax is about 2.3 (if implemented correctly). Why does this make sense?

0.4 Answer:

No training has been performed, so each class will be classified with a probability of 0.1. If we take the negative log of 0.1 (-ln(0.1)), we get ~2.3, which corresponds to our loss.

Softmax gradient

[23]:
```
## Calculate the gradient of the softmax loss in the Softmax class.
# For convenience, we'll write one function that computes the loss
#   and gradient together, softmax.loss_and_grad(X, y)
# You may copy and paste your loss code from softmax.loss() here, and then
#   use the appropriate intermediate values to calculate the gradient.

loss, grad = softmax.loss_and_grad(X_dev,y_dev)

# Compare your gradient to a gradient check we wrote.
# You should see relative gradient errors on the order of 1e-07 or less if you␣
↪implemented the gradient correctly.
softmax.grad_check_sparse(X_dev, y_dev, grad)
```

```
numerical: -0.722136 analytic: -0.722136, relative error: 3.738540e-09
numerical: 0.819143 analytic: 0.819143, relative error: 2.177321e-08
numerical: 1.817788 analytic: 1.817788, relative error: 1.423580e-09
numerical: -0.422552 analytic: -0.422552, relative error: 5.372484e-09
numerical: -0.161966 analytic: -0.161966, relative error: 3.918769e-07
numerical: 0.868902 analytic: 0.868902, relative error: 3.689266e-08
numerical: 0.286321 analytic: 0.286321, relative error: 8.394643e-08
numerical: 0.270276 analytic: 0.270276, relative error: 1.540718e-08
numerical: -0.302617 analytic: -0.302617, relative error: 9.591102e-09
numerical: -1.046201 analytic: -1.046201, relative error: 5.848762e-08
```

0.5 A vectorized version of Softmax

To speed things up, we will vectorize the loss and gradient calculations. This will be helpful for stochastic gradient descent.

[21]:
```
import time
```

[29]:
```
## Implement softmax.fast_loss_and_grad which calculates the loss and gradient
#    WITHOUT using any for loops.

# Standard loss and gradient
tic = time.time()
loss, grad = softmax.loss_and_grad(X_dev, y_dev)
toc = time.time()
print('Normal loss / grad_norm: {} / {} computed in {}s'.format(loss, np.linalg.
↪norm(grad, 'fro'), toc - tic))

tic = time.time()
loss_vectorized, grad_vectorized = softmax.fast_loss_and_grad(X_dev, y_dev)
toc = time.time()
print('Vectorized loss / grad: {} / {} computed in {}s'.format(loss_vectorized,␣
↪np.linalg.norm(grad_vectorized, 'fro'), toc - tic))

# The losses should match but your vectorized implementation should be much␣
↪faster.
```

```
print('difference in loss / grad: {} /{} '.format(loss - loss_vectorized, np.
 ↪linalg.norm(grad - grad_vectorized)))

# You should notice a speedup with the same output.
```

```
Normal loss / grad_norm: 2.3235766236006388 / 332.5967097296429 computed in
0.24800753593444824s
Vectorized loss / grad: 2.3235766236006383 / 332.5967097296429 computed in
0.008000373840332031s
difference in loss / grad: 4.440892098500626e-16 /2.198458141915654e-13
```

0.6 Stochastic gradient descent

We now implement stochastic gradient descent. This uses the same principles of gradient descent we discussed in class, however, it calculates the gradient by only using examples from a subset of the training set (so each gradient calculation is faster).

0.7 Question:

How should the softmax gradient descent training step differ from the svm training step, if at all?

0.8 Answer:

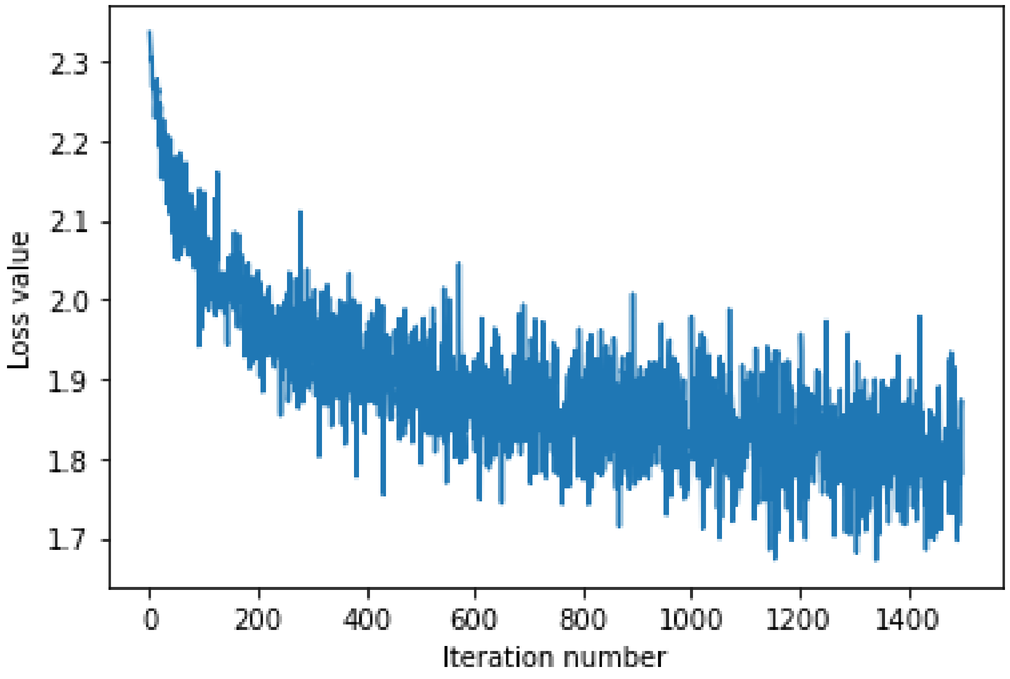
Since the only difference between the 2 classifiers are the loss functions, the training steps will be the same.

[30]:
```
# Implement softmax.train() by filling in the code to extract a batch of data
# and perform the gradient step.
import time


tic = time.time()
loss_hist = softmax.train(X_train, y_train, learning_rate=1e-7,
                      num_iters=1500, verbose=True)
toc = time.time()
print('That took {}s'.format(toc - tic))

plt.plot(loss_hist)
plt.xlabel('Iteration number')
plt.ylabel('Loss value')
plt.show()
```

```
iteration 0 / 1500: loss 2.336592660663754
iteration 100 / 1500: loss 2.0557222613850827
iteration 200 / 1500: loss 2.035774512066282
iteration 300 / 1500: loss 1.9813348165609885
iteration 400 / 1500: loss 1.9583142443981612
iteration 500 / 1500: loss 1.8622653073541355
iteration 600 / 1500: loss 1.8532611454359382
iteration 700 / 1500: loss 1.8353062223725825
iteration 800 / 1500: loss 1.829389246882764
iteration 900 / 1500: loss 1.899215853035748
iteration 1000 / 1500: loss 1.9783503540252303
iteration 1100 / 1500: loss 1.8470797913532635
iteration 1200 / 1500: loss 1.8411450268664085
iteration 1300 / 1500: loss 1.79104024957921
iteration 1400 / 1500: loss 1.870580302938226
That took 17.281003713607788s
```



0.8.1 Evaluate the performance of the trained softmax classifier on the validation data.

[31]:
```python
## Implement softmax.predict() and use it to compute the training and testing␣
↪error.

y_train_pred = softmax.predict(X_train)
print('training accuracy: {}'.format(np.mean(np.equal(y_train,y_train_pred), )))
y_val_pred = softmax.predict(X_val)
print('validation accuracy: {}'.format(np.mean(np.equal(y_val, y_val_pred)), ))
```

```
training accuracy: 0.3811428571428571
validation accuracy: 0.398
```

0.9 Optimize the softmax classifier

You may copy and paste your optimization code from the SVM here.

[33]:
```
np.finfo(float).eps
```

[33]:
```
2.220446049250313e-16
```

[37]:
```
# ================================================================ #
# YOUR CODE HERE:
#   Train the Softmax classifier with different learning rates and
#     evaluate on the validation data.
#   Report:
#     - The best learning rate of the ones you tested.
#     - The best validation accuracy corresponding to the best validation error.
#
#   Select the SVM that achieved the best validation error and report
#     its error rate on the test set.
# ================================================================ #
learning_rates = [0.000001, 0.00001, 0.001, 0.0025, 0.01, 0.025, 0.1, 0.5, 1]

#generates error based on learning rate on validation test
def generateError (rate):
    softmax.train(X_train, y_train, rate)
    y_test_pred = softmax.predict(X_test)
    y_val_pred = softmax.predict(X_val)

    test_error_rate = 1 - np.mean(np.equal(y_test,y_test_pred))
    validation_accuracy = np.mean(np.equal(y_val, y_val_pred))
    print("learning rate:", rate, "validation_accuracy", validation_accuracy,␣
 ↪"test_error_rate", test_error_rate)
    return (rate, validation_accuracy, test_error_rate)

tuples = [generateError(x) for x in learning_rates]

print(max(tuples, key = lambda x : x[1]))
# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #
```

```
learning rate: 1e-06 validation_accuracy 0.374 test_error_rate 0.632
learning rate: 1e-05 validation_accuracy 0.327 test_error_rate
0.6970000000000001
learning rate: 0.001 validation_accuracy 0.087 test_error_rate 0.897
learning rate: 0.0025 validation_accuracy 0.087 test_error_rate 0.897
learning rate: 0.01 validation_accuracy 0.087 test_error_rate 0.897
learning rate: 0.025 validation_accuracy 0.087 test_error_rate 0.897
learning rate: 0.1 validation_accuracy 0.087 test_error_rate 0.897
learning rate: 0.5 validation_accuracy 0.087 test_error_rate 0.897
learning rate: 1 validation_accuracy 0.087 test_error_rate 0.897
(1e-06, 0.374, 0.632)
```

As we can see, the learning rate that performs the best is 1e-06, with a validation_accuracy of 0.374, and hence a validation error rate of 1 - 0.374 = 0.626, and a test error rate of 0.632 (which is also the lowest in all of the samples, showing that our hyperparameter tuning worked as expected).

0.10 Softmax Notebook:

[ ]:
```
import numpy as np
import math

class Softmax(object):

  def __init__(self, dims=[10, 3073]):
    self.init_weights(dims=dims)

  def init_weights(self, dims):
    """
        Initializes the weight matrix of the Softmax classifier.
        Note that it has shape (C, D) where C is the number of
        classes and D is the feature size.
        """
    self.W = np.random.normal(size=dims) * 0.0001

  def loss(self, X, y):
    """
    Calculates the softmax loss.

    Inputs have dimension D, there are C classes, and we operate on minibatches
    of N examples.

    Inputs:
    - X: A numpy array of shape (N, D) containing a minibatch of data.
    - y: A numpy array of shape (N,) containing training labels; y[i] = c means
      that X[i] has label c, where 0 <= c < C.

    Returns a tuple of:
    - loss as single float
    """

    # Initialize the loss to zero.
    loss = 0.0

    # ================================================================= #
    # YOUR CODE HERE:
        #   Calculate the normalized softmax loss.  Store it as the variable␣
↪loss.
    #   (That is, calculate the sum of the losses of all the training
    #   set margins, and then normalize the loss by the number of
        #        training examples.)
    # ================================================================ #

    product = np.dot(X, self.W.T)

    for i, row in enumerate(product):
        row -= np.max(row)
        inner_sum = np.exp(row[y[i]]) / np.sum(np.exp(row)) #the inner sum of␣
↪softmax loss
        logsum = -np.log(inner_sum) #applying negative log to the sum
        loss += logsum

    #normalize
    loss = loss / len(X)

    # ================================================================ #
    # END YOUR CODE HERE
    # ================================================================ #

    return loss

  def loss_and_grad(self, X, y):
    """
        Same as self.loss(X, y), except that it also returns the gradient.

        Output: grad -- a matrix of the same dimensions as W containing
                the gradient of the loss with respect to W.
        """

    # Initialize the loss and gradient to zero.
    loss = 0.0
    grad = np.zeros_like(self.W)

    # ================================================================ #
    # YOUR CODE HERE:
        #   Calculate the softmax loss and the gradient. Store the gradient
        #   as the variable grad.
    # ================================================================ #
    product = np.dot(X, self.W.T)

    for i, row in enumerate(product):
        row -= np.max(row)
        inner_sum = np.exp(row[y[i]]) / np.sum(np.exp(row)) #the inner sum of␣
↪softmax loss
        logsum = -np.log(inner_sum) #applying negative log to the sum
        loss += logsum

        #for each class, calculating the gradient using the Softmax Derivative␣
↪hint on CCLE
        for j in range(10):
            j_softmax = np.exp(row[j]) / np.sum(np.exp(row))
            if j == y[i]:
                grad[j] += X[i] * (j_softmax - 1)
            else:
                grad[j] += X[i] * j_softmax

    #normalize
    loss = loss / len(X)
    grad = grad / len(X)

    # ================================================================ #
    # END YOUR CODE HERE
    # ================================================================ #

    return loss, grad

  def grad_check_sparse(self, X, y, your_grad, num_checks=10, h=1e-5):
    """
    sample a few random elements and only return numerical
    in these dimensions.
    """

    for i in np.arange(num_checks):
      ix = tuple([np.random.randint(m) for m in self.W.shape])

      oldval = self.W[ix]
      self.W[ix] = oldval + h # increment by h
      fxph = self.loss(X, y)
      self.W[ix] = oldval - h # decrement by h
      fxmh = self.loss(X,y) # evaluate f(x - h)
      self.W[ix] = oldval # reset

      grad_numerical = (fxph - fxmh) / (2 * h)
      grad_analytic = your_grad[ix]
      rel_error = abs(grad_numerical - grad_analytic) / (abs(grad_numerical) +␣
↪abs(grad_analytic))
      print('numerical: %f analytic: %f, relative error: %e' % (grad_numerical,␣
↪grad_analytic, rel_error))
```

```
def fast_loss_and_grad(self, X, y):
  """
  A vectorized implementation of loss_and_grad. It shares the same
      inputs and ouptuts as loss_and_grad.
  """
  loss = 0.0
  grad = np.zeros(self.W.shape) # initialize the gradient as zero

  # ================================================================ #
  # YOUR CODE HERE:
      #   Calculate the softmax loss and gradient WITHOUT any for loops.
  # ================================================================ #

  #used this as a resource: https://mlxai.github.io/2017/01/09/
↪implementing-softmax-classifier-with-vectorized-operations.html

  N = X.shape[0]

  product = np.dot(X, self.W.T)
  sum_j = np.sum(np.exp(product), axis=1, keepdims=True)
  coefs = np.exp(product) / sum_j
  coefs[np.arange(N), y] -= 1
  loss = np.log(sum_j) - product[np.arange(N), y]
  loss = np.mean(loss)
  grad = coefs.T.dot(X)
  grad /= len(X)

  # ================================================================ #
  # END YOUR CODE HERE
  # ================================================================ #

  return loss, grad

def train(self, X, y, learning_rate=1e-3, num_iters=100,
          batch_size=200, verbose=False):
  """
  Train this linear classifier using stochastic gradient descent.

  Inputs:
  - X: A numpy array of shape (N, D) containing training data; there are N
    training samples each of dimension D.
  - y: A numpy array of shape (N,) containing training labels; y[i] = c
    means that X[i] has label 0 <= c < C for C classes.
  - learning_rate: (float) learning rate for optimization.
  - num_iters: (integer) number of steps to take when optimizing
  - batch_size: (integer) number of training examples to use at each step.
  - verbose: (boolean) If true, print progress during optimization.
```

```
    Outputs:
    A list containing the value of the loss function at each training iteration.
    """
    num_train, dim = X.shape
    num_classes = np.max(y) + 1 # assume y takes values 0...K-1 where K is␣
↪number of classes

    self.init_weights(dims=[np.max(y) + 1, X.shape[1]])        # initializes␣
↪the weights of self.W

    # Run stochastic gradient descent to optimize W
    loss_history = []

    for it in np.arange(num_iters):
      X_batch = None
      y_batch = None

      # ================================================================ #
      # YOUR CODE HERE:
      #   Sample batch_size elements from the training data for use in
      #          gradient descent.  After sampling,
      #     - X_batch should have shape: (dim, batch_size)
          #     - y_batch should have shape: (batch_size,)
          #   The indices should be randomly generated to reduce correlations
          #   in the dataset.  Use np.random.choice.  It's okay to sample with
          #   replacement.
      # ================================================================ #
      indices = np.random.choice(np.arange(num_train), batch_size)
      X_batch = X[indices]
      y_batch = y[indices]
      # ================================================================ #
      # END YOUR CODE HERE
      # ================================================================ #

      # evaluate loss and gradient
      loss, grad = self.fast_loss_and_grad(X_batch, y_batch)
      loss_history.append(loss)

      # ================================================================ #
      # YOUR CODE HERE:
      #   Update the parameters, self.W, with a gradient step
      # ================================================================ #
      self.W = self.W - learning_rate * grad

          # ================================================================ #
      # END YOUR CODE HERE
      # ================================================================ #

      if verbose and it % 100 == 0:
        print('iteration {} / {}: loss {}'.format(it, num_iters, loss))

    return loss_history

  def predict(self, X):
    """
    Inputs:
    - X: N x D array of training data. Each row is a D-dimensional point.

    Returns:
    - y_pred: Predicted labels for the data in X. y_pred is a 1-dimensional
      array of length N, and each element is an integer giving the predicted
      class.
    """
    y_pred = np.zeros(X.shape[1])
    # ================================================================ #
    # YOUR CODE HERE:
    #   Predict the labels given the training data.
    # ================================================================ #
    product = np.dot(self.W, X.T)
    y_pred = np.argmax(product, axis=0)
    # ================================================================ #
    # END YOUR CODE HERE
    # ================================================================ #

    return y_pred
```