Q1_IFT3165

February 17, 2019

In [ ]:
```
# Importing dataset
import urllib.request
import _pickle as pickle
import gzip
import os
import numpy as np
%matplotlib inline
import matplotlib.pyplot as plt

path = 'http://deeplearning.net/data/mnist'
mnist_filename_all = 'mnist.pkl'
local_filename = os.path.join("./", mnist_filename_all)
urllib.request.urlretrieve(
    "{}/{}.gz".format(path,mnist_filename_all), local_filename+'.gz')
tr,va,te = pickle.load(gzip.open(local_filename+'.gz','r'), encoding='latin1')
np.save(open(local_filename+'.npy','wb'), (tr,va,te))
```

In [ ]:
```
# Split dataset

validRatio = 0.1
mnist_npy = np.load("mnist.pkl.npy")
train_images = mnist_npy[0,0]
train_labels = mnist_npy[0,1]
valid_images = mnist_npy[1,0]
valid_labels = mnist_npy[1,1]
test_images = mnist_npy[2,0]
test_labels = mnist_npy[2,1]

def one_hot_vectorize(vector):
  one_hot_matrix = np.zeros((vector.shape[0], np.max(vector) + 1))
  one_hot_matrix[np.arange(vector.shape[0]), vector] = 1
  return one_hot_matrix

train_labels = one_hot_vectorize(train_labels)
valid_labels = one_hot_vectorize(valid_labels)
test_labels = one_hot_vectorize(test_labels)
```

In [ ]:
```
# Building basic modules
import numpy as np
```

```python
from abc import ABC, abstractmethod
from enum import Enum

class Function(ABC):
  @abstractmethod
  def fprop(self, input):
    pass

  @abstractmethod
  def bprop(self, grads_ouput):
    pass

class InitMethod(Enum):
  ZERO = 1
  NORMAL = 2
  GLOROT = 3
  XAVIER = 4

class LinearLayer(Function):
  def __init__(self, in_n_nodes, out_n_nodes, init_method=InitMethod.GLOROT):
    self.in_n_nodes = in_n_nodes
    self.out_n_nodes = out_n_nodes
    self.init_method = init_method

  def reset(self):
    if self.init_method == InitMethod.ZERO:
      self.weights = np.zeros((self.in_n_nodes, self.out_n_nodes))
    elif self.init_method == InitMethod.NORMAL:
      self.weights = np.random.normal(size=(self.in_n_nodes, self.out_n_nodes))
    elif self.init_method == InitMethod.GLOROT:
      bound = np.sqrt(6.0/(1.0*
                          (self.in_n_nodes
                           + self.out_n_nodes)))
      self.weights = np.random.uniform(-bound,
                                       bound,
                                       size=(self.in_n_nodes,
                                             self.out_n_nodes))
    elif self.init_method == InitMethod.XAVIER:
      bound = 1/np.sqrt(self.in_n_nodes)
      self.weights = np.random.uniform(-bound,
                                       bound,
                                       size=(self.in_n_nodes,
                                             self.out_n_nodes))
    self.bias = np.zeros(self.out_n_nodes)
    self.iterations = 0
    self.zero_grads()

  def zero_grads(self):
```

```
    self.grad_weights = np.zeros_like(self.weights)
    self.grad_bias = np.zeros_like(self.bias)
    self.grad_input = np.zeros((1, self.in_n_nodes))

  def fprop(self, input):
    self.input = input
    output = self.input.dot(self.weights) + self.bias
    assert(output.shape == (self.input.shape[0], self.out_n_nodes))
    return output

  def bprop(self, grad_output):
    self.grad_weights = (self.input.T.dot(grad_output)) / self.input.shape[0]
    assert(self.grad_weights.shape == self.weights.shape)

    self.grad_bias = np.mean(grad_output, axis=0)
    assert(self.grad_bias.shape == self.bias.shape)

    self.grad_input = grad_output.dot(self.weights.T)
    assert(self.grad_input.shape == self.input.shape)

    return self.grad_input

  def update(self, learning_rate):
    self.weights -= learning_rate * self.grad_weights
    self.bias -= learning_rate * self.grad_bias

class ReLU(Function):
  def fprop(self, input):
    self.input = input
    output = np.maximum(self.input, 0)
    assert(output.shape == self.input.shape)
    return output

  def bprop(self, grad_output):
    grad_input = (self.input > 0) * grad_output
    return grad_input

class Softmax():
  @staticmethod
  def fprop(input):
    exp = np.exp(input - input.max(axis=1, keepdims=True))
    output = exp / exp.sum(axis=1, keepdims=True)
    return output

class CrossEntropyLoss(Function):
  def fprop(self, input, target):
    self.epsilon = 1e-10
    self.target = target
```

```
        self.input = Softmax.fprop(input)
        assert(np.allclose(self.input.sum(axis=1, keepdims=True),
                           np.ones_like(self.target.shape[0])))
        logs = -np.log(self.input[np.arange(input.shape[0]),
                                  self.target.argmax(axis=1)]+self.epsilon)
        self.losses = logs.sum()
        return (self.losses) / input.shape[0]

    def bprop(self):
        self.grad_input = copy.deepcopy(self.input)
        self.grad_input[np.arange(self.input.shape[0]), self.target.argmax(axis=1)] -= 1
        return self.grad_input
```

In [ ]:
```
# Building NN
import copy

class NN(object):
  def __init__(self,hidden_dims=(2,2), init_method=InitMethod.GLOROT):
    dims = (784, *hidden_dims)
    self.fcs = []
    self.layers = []
    for i_dim in range(len(dims) - 1):
      fc = LinearLayer(dims[i_dim], dims[i_dim + 1], init_method)
      relu = ReLU()
      self.fcs.append(fc)
      self.layers.append(fc)
      self.layers.append(relu)

    last_linear = LinearLayer(dims[-1], 10, init_method)
    self.fcs.append(last_linear)
    self.layers.append(last_linear)

    self.lossFunc = CrossEntropyLoss()

    self.initialize_weights()

  def initialize_weights(self):
    for fc in self.fcs:
      fc.reset()

  def zero_grads(self):
    for fc in self.fcs:
      fc.zero_grads()

  def forward(self, input):
    x = input
    for layer in self.layers:
      x = layer.fprop(x)
```

```
        return x

      def loss(self, prediction, labels):
        return self.lossFunc.fprop(prediction, labels)

      def softmax(self,input):
        output = Softmax.fprop(input)
        return output

      def backward(self):
        grad = self.lossFunc.bprop()
        for layer in reversed(self.layers):
          grad = layer.bprop(grad)

      def update(self, learning_rate):
        for fc in self.fcs:
          fc.update(learning_rate)
```

```
In [ ]: def finite_gradient_check(mlp, x, y, k=None, i=None):
          if not k:
            k = np.random.randint(1, 5)
          if not i:
            i = np.random.randint(0, 5)
          eps = 1/float(k*10**i)

          weights = copy.deepcopy(mlp.fcs[1].weights)
          n_weights = min(10, weights.shape[1])
          estimated_grads = np.zeros_like(weights[:n_weights])

          for i_idx in range(n_weights):
            for j_idx in range(weights.shape[0]):
              mlp.fcs[1].weights[i_idx,j_idx] += eps

              output = mlp.forward(x)
              loss_peps = mlp.loss(output, y)

              mlp.fcs[1].weights[i_idx,j_idx] -= 2*eps

              output = mlp.forward(x)
              loss_neps = mlp.loss(output, y)

              estimated_grads[i_idx, j_idx] = (loss_peps - loss_neps)/(2.0*eps)

          mlp.fcs[1].weights = weights
          output = mlp.forward(x)
          loss = mlp.loss(output, y)
          mlp.backward()
        #   print("Gradient estimé")
```

```
    #   print(estimated_grads)

    #   print("Gradient par backpropagation")
    #   print(mlp.fcs[1].grad_weights)
      return estimated_grads,mlp.fcs[1].grad_weights
```

In [ ]:
```
import matplotlib.axes as ax
np.random.seed(6)   # Adjust seed if no straight line is shown in the graph
mlp = NN((2,2),init_method=InitMethod.GLOROT)
maxDiff = []
xaxis = []

for i in range(3,6):
  for k in range(1,6):
    estimatedGrads, bpropGrads = finite_gradient_check(mlp,
                                                      train_images[0].reshape(1,-1),
                                                      train_labels[0].reshape(1,-1),
                                                      k, i)
    maxDiff.append(np.amax(np.absolute(estimatedGrads-bpropGrads)))
    xaxis.append(1/float(k*10**i))

xaxis.reverse()
maxDiff.reverse()

plt.plot(xaxis,maxDiff)

plt.xlabel('$\epsilon$ value')
plt.ylabel('Maxium gradient difference')
```

In [ ]:
```
def batchLoader(dataset, batchSize):
  images, labels = dataset
  batches = []
  eof = False
  i = 0
  while eof == False:
    if (i+batchSize)>= len(images):
      batches.append((images[i:],labels[i:]))
      eof = True
    else:
      batches.append((images[i:i+batchSize],labels[i:i+batchSize]))
    i += batchSize
  return batches
```

In [ ]:
```
def trainEpoch(mlp, dataset, batchSize, learningRate):
  batches = batchLoader(dataset,batchSize)
  loss_avg = 0
  acc_avg = 0
  for i, (images, labels) in enumerate(batches):
```

```
            output = mlp.forward(images.reshape(len(images), -1))
            loss = mlp.loss(output,labels)
            loss_avg += loss
            mlp.backward()
            mlp.update(learningRate)
            mlp.zero_grads()
            correct = (np.argmax(output, axis=1) == np.argmax(labels, axis=1)).sum()
            acc_avg += correct
            if (i+1) % 100 == 0:
              print('Step [{}/{}], Loss: {:.4f}({:.4f}), Accuracy: {:.3f}({:.3f})'
                           .format(i+1, len(batches), loss, loss_avg / (i + 1),
                                   correct * 100 / batchSize,
                                   acc_avg * 100 / ((i + 1) * batchSize)))
          return loss_avg / (i + 1), acc_avg * 100 / ((i + 1) * batchSize)
```

In [ ]:
```
def modelEval(mlp, dataset):
  images, labels = dataset
  output = mlp.forward(images.reshape(len(images),-1))
  loss = np.mean(mlp.loss(output,labels))
  rightPred = 0
  total = 0
  for i,example in enumerate(output):
    if np.argmax(example)==np.argmax(labels[i]):
      rightPred += 1
    total += 1
  print('Validation Loss: {:.4f}, Accuracy: {:.4f} '
        .format(loss, rightPred * 100/total))
  return loss, rightPred/total
```

In [ ]:
```
# ZERO INIT
mlp = NN((512,512), init_method=InitMethod.ZERO)
losses_zero_init = []
for epoch in range(10):
  print("Epoch [{}/10]".format(epoch))
  loss = trainEpoch(mlp, (train_images, train_labels), 20, 0.0005)
  losses_zero_init.append(loss)
```

In [ ]:
```
# NORMAL INIT
mlp = NN((512,512), init_method=InitMethod.NORMAL)
losses_normal_init = []
for epoch in range(10):
  print("Epoch [{}/10]".format(epoch))
  loss = trainEpoch(mlp, (train_images, train_labels), 20, 0.0005)
  losses_normal_init.append(loss)
```

In [ ]:
```
# GLOROT INIT
mlp = NN((512,512), init_method=InitMethod.GLOROT)
losses_glorot_init = []
for epoch in range(10):
```

```
    print("Epoch [{}/10]".format(epoch))
    loss = trainEpoch(mlp, (train_images, train_labels), 20, 0.0005)
    losses_glorot_init.append(loss)
```

In [ ]:
```
# XAVIER INIT
mlp = NN((512,512), init_method=InitMethod.XAVIER)
losses_xavier_init = []
for epoch in range(10):
  print("Epoch [{}/10]".format(epoch))
  loss = trainEpoch(mlp, (train_images, train_labels), 20, 0.0005)
  losses_xavier_init.append(loss)
```

In [ ]:
```
#Plotting losses
x = np.arange(1, len(losses_glorot_init)+1)
plt.subplot(1,2,1)
plt.plot(x,[losses_zero_init[i][0] for i in range(len(losses_zero_init))])
plt.plot(x,[losses_normal_init[i][0] for i in range(len(losses_normal_init))])
plt.plot(x,[losses_glorot_init[i][0] for i in range(len(losses_glorot_init))])

plt.title("Loss curve as a function of the epoch number")
plt.xlabel("Epoch number")
plt.ylabel("Loss")

plt.subplot(1,2,2)
plt.plot(x,[losses_zero_init[i][1] for i in range(len(losses_zero_init))])
plt.plot(x,[losses_normal_init[i][1] for i in range(len(losses_normal_init))])
plt.plot(x,[losses_glorot_init[i][1] for i in range(len(losses_glorot_init))])

plt.title("Accuracy curve as a function of the epoch number")
plt.xlabel("Epoch number")
plt.ylabel("Accuracy")
plt.legend(["Zero","Normal","Glorot"], title = "Initialization method")

fig = plt.gcf()
fig.set_size_inches(12, 6)
```

In [ ]:
```
# test main
model = NN((512,512), init_method=InitMethod.GLOROT)
trainDataset = [train_images,train_labels]
validDataset = [valid_images,valid_labels]
testDataset = [test_images,test_labels]
nbEpoch = 30
batchSize = 128
learningRate = 0.05

train_loss = []
train_acc = []
valid_loss = []
```

```
valid_accuracy = []

for epoch in range(1,nbEpoch+1):
  print("Epoch [{}/{}]".format(epoch, nbEpoch))
  # train
  loss, acc = trainEpoch(model, trainDataset, batchSize, learningRate)
  train_loss.append(loss)
  train_acc.append(acc)

  # valid
  loss, acc = modelEval(model, validDataset)
  valid_loss.append(loss)
  if epoch % 17 == 0:#(epoch>1 and np.abs(acc-valid_accuracy[-1])<0.01):
    learningRate /=10
    print('New learning rate : {}'.format(learningRate))
  valid_accuracy.append(acc)

# test
print('Test performances: \n')
loss, acc = modelEval(model, testDataset)

x = np.arange(len(train_loss))
plt.plot(x,train_loss)
plt.plot(x,valid_loss)
```

```
In [ ]: x = np.arange(len(train_loss))
plt.subplot(1,2,1)
plt.plot(x,train_loss)
plt.plot(x,valid_loss)
plt.xlabel("Epoch number")
plt.ylabel("Loss")

plt.subplot(1,2,2)
plt.plot(x,train_acc)
plt.plot(x,100*np.asarray(valid_accuracy))
plt.xlabel("Epoch number")
plt.ylabel("Accuracy")

plt.legend(["Training","Validation"])

fig = plt.gcf()
fig.set_size_inches(12, 6)
```