from \_\_future\_\_ import print\_function, division

from future import standard\_library

standard\_library.install\_aliases()

from builtins import range

from builtins import object

import os

import pickle as pickle

import numpy as np

from cs231n import optim

class Solver(object):

"""

A Solver encapsulates all the logic necessary for training classification

models. The Solver performs stochastic gradient descent using different

update rules defined in optim.py.

The solver accepts both training and validataion data and labels so it can

periodically check classification accuracy on both training and validation

data to watch out for overfitting.

To train a model, you will first construct a Solver instance, passing the

model, dataset, and various optoins (learning rate, batch size, etc) to the

constructor. You will then call the train() method to run the optimization

procedure and train the model.

After the train() method returns, model.params will contain the parameters

that performed best on the validation set over the course of training.

In addition, the instance variable solver.loss\_history will contain a list

of all losses encountered during training and the instance variables

solver.train\_acc\_history and solver.val\_acc\_history will be lists of the

accuracies of the model on the training and validation set at each epoch.

Example usage might look something like this:

data = {

'X\_train': # training data

'y\_train': # training labels

'X\_val': # validation data

'y\_val': # validation labels

}

model = MyAwesomeModel(hidden\_size=100, reg=10)

solver = Solver(model, data,

update\_rule='sgd',

optim\_config={

'learning\_rate': 1e-3,

},

lr\_decay=0.95,

num\_epochs=10, batch\_size=100,

print\_every=100)

solver.train()

A Solver works on a model object that must conform to the following API:

- model.params must be a dictionary mapping string parameter names to numpy

arrays containing parameter values.

- model.loss(X, y) must be a function that computes training-time loss and

gradients, and test-time classification scores, with the following inputs

and outputs:

Inputs:

- X: Array giving a minibatch of input data of shape (N, d\_1, ..., d\_k)

- y: Array of labels, of shape (N,) giving labels for X where y[i] is the

label for X[i].

Returns:

If y is None, run a test-time forward pass and return:

- scores: Array of shape (N, C) giving classification scores for X where

scores[i, c] gives the score of class c for X[i].

If y is not None, run a training time forward and backward pass and

return a tuple of:

- loss: Scalar giving the loss

- grads: Dictionary with the same keys as self.params mapping parameter

names to gradients of the loss with respect to those parameters.

"""

def \_\_init\_\_(self, model, data, \*\*kwargs):

"""

Construct a new Solver instance.

Required arguments:

- model: A model object conforming to the API described above

- data: A dictionary of training and validation data containing:

'X\_train': Array, shape (N\_train, d\_1, ..., d\_k) of training images

'X\_val': Array, shape (N\_val, d\_1, ..., d\_k) of validation images

'y\_train': Array, shape (N\_train,) of labels for training images

'y\_val': Array, shape (N\_val,) of labels for validation images

Optional arguments:

- update\_rule: A string giving the name of an update rule in optim.py.

Default is 'sgd'.

- optim\_config: A dictionary containing hyperparameters that will be

passed to the chosen update rule. Each update rule requires different

hyperparameters (see optim.py) but all update rules require a

'learning\_rate' parameter so that should always be present.

- lr\_decay: A scalar for learning rate decay; after each epoch the

learning rate is multiplied by this value.

- batch\_size: Size of minibatches used to compute loss and gradient

during training.

- num\_epochs: The number of epochs to run for during training.

- print\_every: Integer; training losses will be printed every

print\_every iterations.

- verbose: Boolean; if set to false then no output will be printed

during training.

- num\_train\_samples: Number of training samples used to check training

accuracy; default is 1000; set to None to use entire training set.

- num\_val\_samples: Number of validation samples to use to check val

accuracy; default is None, which uses the entire validation set.

- checkpoint\_name: If not None, then save model checkpoints here every

epoch.

"""

self.model = model

self.X\_train = data['X\_train']

self.y\_train = data['y\_train']

self.X\_val = data['X\_val']

self.y\_val = data['y\_val']

# Unpack keyword arguments

self.update\_rule = kwargs.pop('update\_rule', 'sgd')

self.optim\_config = kwargs.pop('optim\_config', {})

self.lr\_decay = kwargs.pop('lr\_decay', 1.0)

self.batch\_size = kwargs.pop('batch\_size', 100)

self.num\_epochs = kwargs.pop('num\_epochs', 10)

self.num\_train\_samples = kwargs.pop('num\_train\_samples', 1000)

self.num\_val\_samples = kwargs.pop('num\_val\_samples', None)

self.checkpoint\_name = kwargs.pop('checkpoint\_name', None)

self.print\_every = kwargs.pop('print\_every', 10)

self.verbose = kwargs.pop('verbose', True)

# Throw an error if there are extra keyword arguments

if len(kwargs) > 0:

extra = ', '.join('"%s"' % k for k in list(kwargs.keys()))

raise ValueError('Unrecognized arguments %s' % extra)

# Make sure the update rule exists, then replace the string

# name with the actual function

if not hasattr(optim, self.update\_rule):

raise ValueError('Invalid update\_rule "%s"' % self.update\_rule)

self.update\_rule = getattr(optim, self.update\_rule)

self.\_reset()

def \_reset(self):

"""

Set up some book-keeping variables for optimization. Don't call this

manually.

"""

# Set up some variables for book-keeping

self.epoch = 0

self.best\_val\_acc = 0

self.best\_params = {}

self.loss\_history = []

self.train\_acc\_history = []

self.val\_acc\_history = []

# Make a deep copy of the optim\_config for each parameter

self.optim\_configs = {}

for p in self.model.params:

d = {k: v for k, v in self.optim\_config.items()}

self.optim\_configs[p] = d

def \_step(self):

"""

Make a single gradient update. This is called by train() and should not

be called manually.

"""

# Make a minibatch of training data

num\_train = self.X\_train.shape[0]

batch\_mask = np.random.choice(num\_train, self.batch\_size)

X\_batch = self.X\_train[batch\_mask]

y\_batch = self.y\_train[batch\_mask]

# Compute loss and gradient

loss, grads = self.model.loss(X\_batch, y\_batch)

self.loss\_history.append(loss)

# Perform a parameter update

for p, w in self.model.params.items():

dw = grads[p]

config = self.optim\_configs[p]

next\_w, next\_config = self.update\_rule(w, dw, config)

self.model.params[p] = next\_w

self.optim\_configs[p] = next\_config

def \_save\_checkpoint(self):

if self.checkpoint\_name is None: return

checkpoint = {

'model': self.model,

'update\_rule': self.update\_rule,

'lr\_decay': self.lr\_decay,

'optim\_config': self.optim\_config,

'batch\_size': self.batch\_size,

'num\_train\_samples': self.num\_train\_samples,

'num\_val\_samples': self.num\_val\_samples,

'epoch': self.epoch,

'loss\_history': self.loss\_history,

'train\_acc\_history': self.train\_acc\_history,

'val\_acc\_history': self.val\_acc\_history,

}

filename = '%s\_epoch\_%d.pkl' % (self.checkpoint\_name, self.epoch)

if self.verbose:

print('Saving checkpoint to "%s"' % filename)

with open(filename, 'wb') as f:

pickle.dump(checkpoint, f)

def check\_accuracy(self, X, y, num\_samples=None, batch\_size=100):

"""

Check accuracy of the model on the provided data.

Inputs:

- X: Array of data, of shape (N, d\_1, ..., d\_k)

- y: Array of labels, of shape (N,)

- num\_samples: If not None, subsample the data and only test the model

on num\_samples datapoints.

- batch\_size: Split X and y into batches of this size to avoid using

too much memory.

Returns:

- acc: Scalar giving the fraction of instances that were correctly

classified by the model.

"""

# Maybe subsample the data

N = X.shape[0]

if num\_samples is not None and N > num\_samples:

mask = np.random.choice(N, num\_samples)

N = num\_samples

X = X[mask]

y = y[mask]

# Compute predictions in batches

num\_batches = N // batch\_size

if N % batch\_size != 0:

num\_batches += 1

y\_pred = []

for i in range(num\_batches):

start = i \* batch\_size

end = (i + 1) \* batch\_size

scores = self.model.loss(X[start:end])

y\_pred.append(np.argmax(scores, axis=1))

y\_pred = np.hstack(y\_pred)

acc = np.mean(y\_pred == y)

return acc

def train(self):

"""

Run optimization to train the model.

"""

num\_train = self.X\_train.shape[0]

iterations\_per\_epoch = max(num\_train // self.batch\_size, 1)

num\_iterations = self.num\_epochs \* iterations\_per\_epoch

for t in range(num\_iterations):

self.\_step()

# Maybe print training loss

if self.verbose and t % self.print\_every == 0:

print('(Iteration %d / %d) loss: %f' % (

t + 1, num\_iterations, self.loss\_history[-1]))

# At the end of every epoch, increment the epoch counter and decay

# the learning rate.

epoch\_end = (t + 1) % iterations\_per\_epoch == 0

if epoch\_end:

self.epoch += 1

for k in self.optim\_configs:

self.optim\_configs[k]['learning\_rate'] \*= self.lr\_decay

# Check train and val accuracy on the first iteration, the last

# iteration, and at the end of each epoch.

first\_it = (t == 0)

last\_it = (t == num\_iterations - 1)

if first\_it or last\_it or epoch\_end:

train\_acc = self.check\_accuracy(self.X\_train, self.y\_train,

num\_samples=self.num\_train\_samples)

val\_acc = self.check\_accuracy(self.X\_val, self.y\_val,

num\_samples=self.num\_val\_samples)

self.train\_acc\_history.append(train\_acc)

self.val\_acc\_history.append(val\_acc)

self.\_save\_checkpoint()

if self.verbose:

print('(Epoch %d / %d) train acc: %f; val\_acc: %f' % (

self.epoch, self.num\_epochs, train\_acc, val\_acc))

# Keep track of the best model

if val\_acc > self.best\_val\_acc:

self.best\_val\_acc = val\_acc

self.best\_params = {}

for k, v in self.model.params.items():

self.best\_params[k] = v.copy()

# At the end of training swap the best params into the model

self.model.params = self.best\_params