cool_bonus

September 29, 2021

1 Cool Bonus: Dynamic Mini-batch

This is mainly testing a question I had during class. Which would having different batch sizes during the training help improve the process? Since in class the batch size was always the same during the same training, I wanted to test out whether having three different batch-size(small, medium, large) had any effect.

The specific question I wish to answer in the following experient are: - Would a dynamic batch be faster than a large or slower than a small batch-size? If so, by how much? - Would a dynamic batch have a better than a small or worse than a large batch-size? If so, by how much?

[1]:
```
# Run some setup code for this notebook.
from __future__ import print_function
import random
import numpy as np
from cs682.data_utils import load_CIFAR10
import matplotlib.pyplot as plt


# This is a bit of magic to make matplotlib figures appear inline in the
# notebook rather than in a new window.
%matplotlib inline
plt.rcParams['figure.figsize'] = (10.0, 8.0) # set default size of plots
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'

# Some more magic so that the notebook will reload external python modules;
# see http://stackoverflow.com/questions/1907993/
↪autoreload-of-modules-in-ipython
%load_ext autoreload
%autoreload 2
```

1.1 CIFAR-10 Data Loading and Preprocessing

[2]:
```
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
```

```python
    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 = 'cs682/datasets/cifar-10-batches-py'

    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]
    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


# Cleaning up variables to prevent loading data multiple times (which may cause␣
 ↪memory issue)
try:
```

```python
    del X_train, y_train
    del X_test, y_test
    print('Clear previously loaded data.')
except:
    pass

# 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,)
```

[3]:
```python
from cs682.classifiers import LinearSVM
import time

results = {}
best_val = -1
best_svm = None
batch_sizes = [100, [100, 300, 500], 500]

for batch_size in batch_sizes:

    tic = time.time()

    svm = LinearSVM()
    svm.train(X_train, y_train, batch_size=batch_size, learning_rate=1e-7,␣
 ↪reg=1e4, num_iters=1500)
    y_train_pred = svm.predict(X_train)
    train_accuracy = np.mean(y_train == y_train_pred)
    y_val_pred = svm.predict(X_val)
    val_accuracy = np.mean(y_val == y_val_pred)
```

```
    toc = time.time()
    cost_time = toc-tic

    results[str(batch_size)] = (train_accuracy, val_accuracy, cost_time)

    if val_accuracy > best_val:
        best_svm = svm
        best_val = val_accuracy


# Print out results.
for batch_size in sorted(results):
    train_accuracy, val_accuracy, cost_time = results[str(batch_size)]
    print('batch size %s train accuracy: %f val accuracy: %f took %f seconds' %␣
 ↪(
                batch_size, train_accuracy, val_accuracy, cost_time))

print('best validation accuracy achieved during cross-validation: %f' %␣
 ↪best_val)
```

```
batch size 100 train accuracy: 0.371224 val accuracy: 0.384000 took 1.818013
seconds
batch size 500 train accuracy: 0.385857 val accuracy: 0.399000 took 8.026039
seconds
batch size [100, 300, 500] train accuracy: 0.387633 val accuracy: 0.394000 took
4.874423 seconds
best validation accuracy achieved during cross-validation: 0.399000
```

[4]:
```
from cs682.classifiers import Softmax
results = {}
best_val = -1
best_softmax = None
batch_sizes = [100, [100, 300, 500], 500]

for batch_size in batch_sizes:

    tic = time.time()

    softmax = Softmax()
    softmax.train(X_train, y_train, learning_rate=8e-8, reg=9e3,␣
 ↪batch_size=batch_size, num_iters=1500)
    y_train_pred = softmax.predict(X_train)
    train_accuracy = np.mean(y_train == y_train_pred)
    y_val_pred = softmax.predict(X_val)
    val_accuracy = np.mean(y_val == y_val_pred)

    toc = time.time()
```

```
    cost_time = toc-tic

    results[str(batch_size)] = (train_accuracy, val_accuracy, cost_time)

    if val_accuracy > best_val:
        best_svm = svm
        best_val = val_accuracy


# Print out results.
for batch_size in sorted(results):
    train_accuracy, val_accuracy, cost_time = results[str(batch_size)]
    print('batch size %s train accuracy: %f val accuracy: %f took %f seconds' %␣
 ↪(
                batch_size, train_accuracy, val_accuracy, cost_time))

print('best validation accuracy achieved during cross-validation: %f' %␣
 ↪best_val)
```

```
batch size 100 train accuracy: 0.337184 val accuracy: 0.353000 took 1.148682
seconds
batch size 500 train accuracy: 0.343367 val accuracy: 0.372000 took 8.106762
seconds
batch size [100, 300, 500] train accuracy: 0.339429 val accuracy: 0.352000 took
4.760601 seconds
best validation accuracy achieved during cross-validation: 0.372000
```

The results I got for both SVM and Softmax were interesting.

Dynamic mini-batch was faster in both cases! However the val accuracy for Softmax seems the same as the small batch-size, while for SVM the dynamic mini-batch was much closer to large batch-size and only cost 4.87s when the large batch-size cost 8.02.

I think it would definitely need more experiments to make sure of the benefits for dynamic mini-batch. But the result I got on SVM looks promising!