atharva-pandhare_homework1_submission

February 21, 2025

1 Homework 1

In this homework we try to classify the images form the MNIST dataset, which is a collection of 70000 hand drawn images.

[1]:
```
import numpy as np
from urllib import request
import gzip
import pickle
import numpy as np
import math
import operator
import time
from numba import cuda
import torch
import torch.nn as nn
from torch.utils.data import TensorDataset, DataLoader
```

1.1 Download Dataset

[2]:
```
filename = [
["training_images","train-images-idx3-ubyte.gz"],
["test_images","t10k-images-idx3-ubyte.gz"],
["training_labels","train-labels-idx1-ubyte.gz"],
["test_labels","t10k-labels-idx1-ubyte.gz"]
]

def download_mnist():
    base_url = "https://ossci-datasets.s3.amazonaws.com/mnist/"
    for name in filename:
        print("Downloading "+name[1]+"...")
        request.urlretrieve(base_url+name[1], name[1])
    print("Download complete.")

def save_mnist():
    mnist = {}
    for name in filename[:2]:
        with gzip.open(name[1], 'rb') as f:
```

```
            mnist[name[0]] = np.frombuffer(f.read(), np.uint8, offset=16).
 ↪reshape(-1,28*28)
    for name in filename[-2:]:
        with gzip.open(name[1], 'rb') as f:
            mnist[name[0]] = np.frombuffer(f.read(), np.uint8, offset=8)
    with open("mnist.pkl", 'wb') as f:
        pickle.dump(mnist,f)
    print("Save complete.")

def init():
    download_mnist()
    save_mnist()
#    print ((load()[0]).shape)
def load():
    with open("mnist.pkl",'rb') as f:
        mnist = pickle.load(f)
    return mnist["training_images"], mnist["training_labels"],␣
 ↪mnist["test_images"], mnist["test_labels"]

if __name__ == '__main__':
    init()
```

```
Downloading train-images-idx3-ubyte.gz…
Downloading t10k-images-idx3-ubyte.gz…
Downloading train-labels-idx1-ubyte.gz…
Downloading t10k-labels-idx1-ubyte.gz…
Download complete.
Save complete.
```

1.2 K-Nearest Neighbor (KNN)

We first load and flatten the MNIST dataset. we reshape the data to np arrays

[3]:
```
# classify using kNN
# x_train = np.load('../x_train.npy')
# y_train = np.load('../y_train.npy')
# x_test = np.load('../x_test.npy')
# y_test = np.load('../y_test.npy')
x_train, y_train, x_test, y_test = load()
x_train = x_train.reshape(60000, 28, 28).astype(np.float32)
x_test = x_test.reshape(10000, 28, 28).astype(np.float32)
```

Distance Calculation - L1 or Manhattan Distance is calculated by

$$\sum_{i=1}^{n} |x_i - y_i|$$

- L2 or Euclidean Distance is calculated by

$$\sqrt{\sum_{i=1}^{n}(x_i - y_i)^2}$$

[4]:
```python
def distance(x1, x2, method='l2'):
    # calculate the distance between two images
    match method.lower():
        case 'l1':
            return np.sum(np.abs(x1 - x2))  # L1 norm
        case 'l2':
            return np.sqrt(np.sum((x1 - x2) ** 2))  # L2 norm
```

KNN Classify Function For this function we iterate through all the images in the testing set, then find the distance between the `testImage` and all of the `trainImage` in the `trainSet`. Then we have to find the K nearest for each `testImage` then we get an array `result` which is the output of the knn.

[5]:
```python
def kNNClassify(testSet, trainSet, labels, k, method='l2'):

    result = []

    for testImage in testSet:

        # Calculate distances between test image and all training images
        distances = []

        for trainImage in trainSet:

            #Calculate distance
            distances.append(distance(testImage, trainImage, method))

        # Find k nearest neighbors
        distances = np.array(distances)
        k_nearest_indices = np.argsort(distances)[:k]
        k_nearest_labels = labels[k_nearest_indices]

        # Get most common label among k neighbors
        predicted_label = np.bincount(k_nearest_labels).argmax()
        result.append(predicted_label)

    return result
```

Finding K

[6]:
```python
test = 1000
train = 60000
best_k = 0
```

```
best_accuracy = 0

# for k in range(1, 13, 1):  # Test odd values of k from 1 to 13
#     start_time = time.time()
#     outputlabels = kNNClassify(x_test[0:test-1], x_train[0:train-1],␣
 ↪y_train[0:train-1], k)
#     result = y_test[0:test-1] - outputlabels
#     accuracy = (1 - np.count_nonzero(result) / len(outputlabels))
#     print(f"K: {k} with accuracy: {accuracy} ran in: {(time.time() -␣
 ↪start_time)}\n")
#     if accuracy > best_accuracy:
#         best_accuracy = accuracy
#         best_k = k
# print(f"Best K: {best_k} with accuracy: {best_accuracy}")

best_k = 3
```

KNN with L1 Distance

[7]:
```
start_time = time.time()
outputlabels = kNNClassify(x_test[0:test-1], x_train[0:train-1], y_train[0:
 ↪train-1], best_k, method='l1')
result = np.subtract(y_test[0:test-1], outputlabels)
result = 1 - np.count_nonzero(result) / len(outputlabels)
l1_time = time.time() - start_time
print("---classification accuracy for knn on mnist: %s ---" % result)
print("---execution time: %s seconds ---" % (l1_time))
```

```
---classification accuracy for knn on mnist: 0.9529529529529529 ---
---execution time: 275.7464723587036 seconds ---
```

KNN with L2 Distance

[8]:
```
start_time = time.time()
outputlabels = kNNClassify(x_test[0:test-1], x_train[0:train-1], y_train[0:
 ↪train-1], best_k, method='l2')
result = np.subtract(y_test[0:test-1], outputlabels)
result = 1 - np.count_nonzero(result) / len(outputlabels)
l2_time = time.time() - start_time
print("---classification accuracy for knn on mnist: %s ---" % result)
print("---execution time: %s seconds ---" % (l2_time))
```

```
---classification accuracy for knn on mnist: 0.9619619619619619 ---
---execution time: 337.7166998386383 seconds ---
```

1.2.1 Alternative (Parallelized with CUDA)

- **This allows me to run the full testing dataset, rather than just 1000 test images**

Here I need to schedule computation across threads.

[9]:
```
@cuda.jit()  # this is the device funciton
def calculate_l1_cuda(test_image, train_images, distances, n_train, image_size):
    idx = cuda.grid(1)
    if idx < n_train:
        l1_dist = 0.0
        for i in range(image_size):
            for j in range(image_size):
                diff = abs(test_image[i, j] - train_images[idx, i, j])
                l1_dist += diff
        distances[idx] = l1_dist

@cuda.jit()  # this is the device funciton
def calculate_l2_cuda(test_image, train_images, distances, n_train, image_size):
    idx = cuda.grid(1)
    if idx < n_train:
        l2_dist = 0.0
        for i in range(image_size):
            for j in range(image_size):
                diff = test_image[i, j] - train_images[idx, i, j]
                l2_dist += diff * diff
        distances[idx] = math.sqrt(l2_dist)
```

Finding the most frequent Neighbor

[10]:
```
def get_most_frequent(labels):
    values, counts = np.unique(labels, return_counts=True)
    return values[np.argmax(counts)]
```

CUDA Classify Here I also have code to coput all the images to the GPU memory

[11]:
```
# this runs on the host so we need to copy training data to the device and␣
 ↪distances to the device
def kNNClassify_cuda(newInput, dataSet, labels, k, method=2):
    result = []
    n_test = len(newInput)
    n_train = len(dataSet)
    image_size = 28

    d_train_images = cuda.to_device(dataSet)

    distances = np.zeros(n_train, dtype=np.float32)
    d_distances = cuda.to_device(distances)

    threadsperblock = 256
    blockspergrid = (n_train + (threadsperblock - 1)) // threadsperblock

    for test_image in newInput:
        d_test_image = cuda.to_device(test_image)
```

```
        if method == 1:
            calculate_l1_cuda[blockspergrid, threadsperblock](
                d_test_image, d_train_images, d_distances, n_train, image_size
            )
        else:
            calculate_l2_cuda[blockspergrid, threadsperblock](
                d_test_image, d_train_images, d_distances, n_train, image_size
            )

        # Find k nearest neighbors
        distances = np.array(d_distances.copy_to_host())
        k_nearest_indices = np.argsort(distances)[:k]
        k_nearest_labels = labels[k_nearest_indices]

        # Get most common label among k neighbors
        predicted_label = get_most_frequent(k_nearest_labels)
        result.append(predicted_label)
    return result
```

Finding K

[12]:
```
best_k = 0
best_accuracy = 0

# for k in range(1, 13, 2):
#     start_time = time.time()
#     outputlabels = kNNClassify(x_test, x_train, y_train, k)
#     result = y_test - outputlabels
#     accuracy = (1 - np.count_nonzero(result) / len(outputlabels))
#     print(f"K: {k} with accuracy: {accuracy} ran in: {(time.time() -␣
 ↪start_time)}\n")
#     if accuracy > best_accuracy:
#         best_accuracy = accuracy
#         best_k = k
# print(f"Best K: {best_k} with accuracy: {best_accuracy}")

k = 3
```

KNN with L1 Distance

[13]:
```
start_time = time.time()
outputlabels = kNNClassify_cuda(x_test, x_train, y_train, k, method=1)
l1_cuda_result = np.subtract(y_test, outputlabels)
l1_cuda_result = 1 - np.count_nonzero(l1_cuda_result) / len(outputlabels)
l1_time = time.time() - start_time
print("---classification accuracy for knn with L1 Distance on mnist: %s ---" %␣
 ↪l1_cuda_result)
print("---execution time: %s seconds ---" % (time.time() - start_time))
```

```
---classification accuracy for knn with L1 Distance on mnist: 0.9633 ---
---execution time: 46.68482303619385 seconds ---
```

KNN with L2 Distance

[14]:
```
start_time = time.time()
outputlabels = kNNClassify_cuda(x_test, x_train, y_train, k, method=2)
l2_cuda_result = np.subtract(y_test, outputlabels)
l2_cuda_result = 1 - np.count_nonzero(l2_cuda_result) / len(outputlabels)
l2_time = time.time() - start_time
print("---classification accuracy for knn with L2 Distance on mnist: %s ---" %␣
 ↪l2_cuda_result)
print("---execution time: %s seconds ---" % (l2_time))
```

```
---classification accuracy for knn with L2 Distance on mnist: 0.9705 ---
---execution time: 47.02558994293213 seconds ---
```

1.3 Linear Classifier

Here I need to load the data as Tensors

[15]:
```
x_train, y_train, x_test, y_test = load() ## reload the data to convert to␣
 ↪tensor, because the previous data is in flattened
X_train = torch.FloatTensor(x_train)
y_train = torch.LongTensor(y_train)
X_test = torch.FloatTensor(x_test)
y_test = torch.LongTensor(y_test)
```

Here I create the Linear Classifier as a NN module

[16]:
```
class LinearClassifier(nn.Module):
    def __init__(self):
        super(LinearClassifier, self).__init__()
        self.linear = nn.Linear(784, 10)

    def forward(self, x):
        return self.linear(x)
```

I define the model as the the previously defined Linear Classifier
I also define the criterion as Cross Entropy Loss

[17]:
```
model = LinearClassifier()
criterion = nn.CrossEntropyLoss()
```

Random Search

I do This by creating random tensors teh size of the weight and bias matix, then randomly iterating those tensors by adding random tensors of similar dimensions.

I also use a paitience so that if the criterion doesnt improve in that paitence it stops itereating and uses the current best.

[18]:
```
def random_search(model, X_train, y_train, num_iterations):
    W = torch.randn_like(model.linear.weight) * 0.001
    b = torch.zeros_like(model.linear.bias)
    bestloss = float("inf")
    patience = 20
    no_improve = 0

    # Create DataLoader for batch processing
    dataset = TensorDataset(X_train, y_train)
    loader = DataLoader(dataset, batch_size=256, shuffle=True)

    for i in range(num_iterations):
        step_size = 0.0001
        # Randomly iterate weights and biases
        Wtry = W + torch.randn_like(W) * step_size
        btry = b + torch.randn_like(b) * step_size

        # Evaluate on batches
        total_loss = 0
        with torch.no_grad():
            model.linear.weight.data = Wtry
            model.linear.bias.data = btry

            for X_batch, y_batch in loader:
                outputs = model(X_batch)
                loss = criterion(outputs, y_batch)
                total_loss += loss.item()

        # Update if better
        if total_loss < bestloss:
            W = Wtry
            b = btry
            bestloss = total_loss
            no_improve = 0
            print(f"iter {i} loss is {bestloss}")
        else:
            no_improve += 1

        # Early stopping
        if no_improve >= patience:
            print(f"Early stopping at iteration {i}")
            break

    with torch.no_grad():
        model.linear.weight.data = W
        model.linear.bias.data = b
```

```
    return bestloss
```

[19]:
```
# Run random search
num_iterations = 1000
best_loss = random_search(model, X_train, y_train, num_iterations)
```

```
iter 0 loss is 972.9965524673462
iter 3 loss is 971.1871347427368
iter 5 loss is 970.7853734493256
iter 7 loss is 953.9483580589294
iter 8 loss is 948.4549331665039
iter 10 loss is 938.0834476947784
iter 11 loss is 921.4819538593292
iter 12 loss is 912.2363419532776
iter 13 loss is 891.3741278648376
iter 25 loss is 887.4673926830292
iter 27 loss is 885.2469449043274
iter 28 loss is 881.9186415672302
iter 31 loss is 873.1291332244873
iter 32 loss is 865.4674007892609
iter 37 loss is 860.9611802101135
iter 41 loss is 858.6912062168121
iter 42 loss is 852.3809564113617
iter 49 loss is 852.3537600040436
iter 51 loss is 851.4858276844025
iter 52 loss is 843.6579508781433
iter 62 loss is 843.0981078147888
iter 66 loss is 838.7063837051392
iter 70 loss is 833.1891491413116
iter 73 loss is 817.394223690033
iter 91 loss is 807.0933244228363
iter 94 loss is 804.0488550662994
iter 96 loss is 799.809755563736
iter 97 loss is 799.7234072685242
iter 100 loss is 792.5128192901611
iter 104 loss is 789.1803686618805
iter 105 loss is 789.1078298091888
iter 106 loss is 786.9032552242279
iter 109 loss is 776.5120255947113
iter 112 loss is 769.8213701248169
iter 116 loss is 768.878169298172
iter 119 loss is 768.7650716304779
iter 123 loss is 765.5589973926544
iter 127 loss is 756.1874144077301
iter 129 loss is 751.1587266921997
iter 130 loss is 742.4206235408783
iter 135 loss is 731.8277344703674
iter 139 loss is 729.2146506309509
```

```
iter 141 loss is 712.3126292228699
iter 142 loss is 711.9655363559723
iter 143 loss is 711.7835221290588
iter 144 loss is 709.7122802734375
iter 158 loss is 706.3389110565186
iter 159 loss is 703.022952079773
iter 165 loss is 696.4874119758606
iter 166 loss is 690.8733220100403
iter 168 loss is 689.5666244029999
iter 172 loss is 679.9857468605042
iter 177 loss is 677.1229717731476
iter 178 loss is 675.4020781517029
iter 181 loss is 668.9591097831726
iter 185 loss is 656.3398959636688
iter 189 loss is 652.8328516483307
iter 191 loss is 651.8487560749054
iter 194 loss is 645.1855752468109
iter 195 loss is 644.1110932826996
iter 196 loss is 640.193487405777
iter 198 loss is 638.8000872135162
iter 199 loss is 631.491781949997
iter 206 loss is 629.029390335083
iter 208 loss is 626.6619520187378
iter 210 loss is 624.8346736431122
iter 217 loss is 623.2165613174438
iter 219 loss is 618.9623892307281
iter 221 loss is 614.773638010025
iter 224 loss is 613.8875856399536
iter 225 loss is 610.6387121677399
iter 233 loss is 610.4425909519196
iter 234 loss is 609.6140298843384
iter 238 loss is 605.3239204883575
iter 241 loss is 601.6308274269104
iter 244 loss is 593.9261500835419
iter 253 loss is 588.1044182777405
iter 259 loss is 584.8424837589264
iter 263 loss is 578.1553497314453
iter 265 loss is 574.5319938659668
iter 269 loss is 572.1391398906708
iter 270 loss is 570.4906339645386
iter 271 loss is 569.0766112804413
iter 280 loss is 568.3956248760223
iter 282 loss is 566.5225369930267
iter 288 loss is 562.7425298690796
iter 296 loss is 562.3144805431366
iter 301 loss is 561.9298346042633
iter 303 loss is 556.2889078855515
iter 305 loss is 553.7046575546265
```

```
iter 311 loss is 553.3110675811768
iter 316 loss is 548.9126040935516
iter 320 loss is 548.4304649829865
iter 321 loss is 547.7561362981796
iter 326 loss is 545.4244694709778
iter 333 loss is 542.9867570400238
iter 335 loss is 540.2695699930191
iter 343 loss is 540.1204762458801
iter 344 loss is 538.5413639545441
iter 346 loss is 533.8516862392426
iter 362 loss is 530.4737641811371
iter 374 loss is 525.5853114128113
iter 376 loss is 524.6620662212372
iter 381 loss is 523.2887961864471
iter 383 loss is 520.5795179605484
iter 385 loss is 518.2775642871857
iter 390 loss is 515.551144361496
iter 391 loss is 510.74187767505646
iter 395 loss is 504.51985573768616
iter 397 loss is 501.6357589960098
iter 399 loss is 497.73357629776
iter 403 loss is 497.65443181991577
iter 406 loss is 497.6275441646576
iter 409 loss is 495.30923891067505
iter 418 loss is 495.12645173072815
iter 435 loss is 493.0992373228073
iter 439 loss is 490.2086853981018
iter 445 loss is 484.4991374015808
iter 452 loss is 481.06915187835693
iter 456 loss is 479.9139897823334
iter 466 loss is 477.37454903125763
iter 478 loss is 477.1031663417816
iter 480 loss is 473.21530425548553
iter 485 loss is 472.72734010219574
iter 487 loss is 472.34826827049255
iter 490 loss is 471.5476804971695
iter 493 loss is 469.43462109565735
iter 495 loss is 468.73708939552307
iter 498 loss is 464.43986761569977
iter 500 loss is 456.5877398252487
iter 501 loss is 452.9402903318405
iter 507 loss is 451.63563573360443
iter 508 loss is 449.42149209976196
iter 513 loss is 449.30492627620697
iter 514 loss is 448.1386889219284
iter 517 loss is 447.5015195608139
iter 519 loss is 444.71901881694794
iter 527 loss is 441.0835210084915
```

```
iter 537 loss is 440.78677928447723
iter 538 loss is 439.96349024772644
iter 539 loss is 439.44799995422363
iter 544 loss is 439.18560540676117
iter 559 loss is 438.9425200223923
iter 563 loss is 437.8042559623718
iter 571 loss is 436.71906316280365
iter 575 loss is 436.51673781871796
iter 576 loss is 431.48624312877655
iter 580 loss is 431.1464195251465
iter 584 loss is 431.04766070842743
iter 586 loss is 427.5263440608978
iter 587 loss is 427.23269963264465
iter 591 loss is 424.09798848629
iter 606 loss is 422.65196192264557
iter 608 loss is 418.65102314949036
iter 611 loss is 416.73304307460785
iter 616 loss is 412.46741580963135
iter 618 loss is 410.88068103790283
iter 621 loss is 406.4417848587036
iter 623 loss is 406.1593087911606
iter 625 loss is 404.69087970256805
iter 626 loss is 404.5505175590515
iter 640 loss is 401.39259934425354
iter 644 loss is 400.3712797164917
iter 645 loss is 400.05269384384155
iter 649 loss is 398.5798667669296
iter 650 loss is 397.8381208181381
iter 651 loss is 397.21473145484924
iter 670 loss is 394.4460771083832
iter 687 loss is 393.0947366952896
iter 694 loss is 392.2623220682144
iter 701 loss is 390.69711470603943
iter 702 loss is 387.9499179124832
iter 704 loss is 385.8403527736664
iter 708 loss is 383.4819859266281
Early stopping at iteration 728
```

I then run Random Search over 1000 Iterations

Then I can evaluate the Model

[20]:
```python
# Evaluate the model
model.eval()
with torch.no_grad():
    outputs = model(X_test)
    _, predicted = torch.max(outputs, 1)
    accuracy = (predicted == y_test).sum().item() / y_test.size(0)
```

```
    print(f"Test Accuracy: {accuracy * 100:.2f}%")
```

```
Test Accuracy: 52.55%
```