Problem 1.

1-3: see the code in the back

4. (Training accuracy plot on the right)

learning rate = 0.1

Epoch 0 in 3.29 sec

Training set accuracy 0.9398333430290222

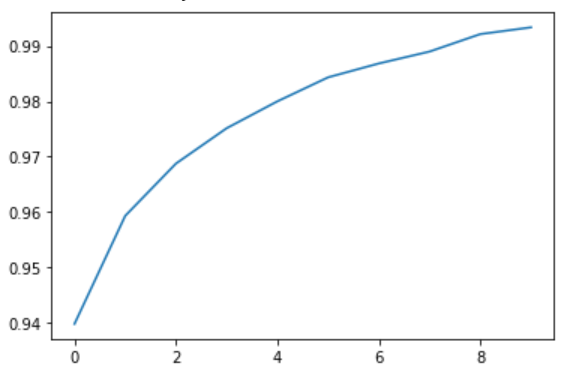
Test set accuracy 0.9395000338554382

Epoch 1 in 0.50 sec

Training set accuracy 0.9593333601951599

Test set accuracy 0.9532000422477722

Epoch 2 in 0.50 sec

Training set accuracy 0.9687666893005371

Test set accuracy 0.9602000713348389

Epoch 3 in 0.50 sec

Training set accuracy 0.9751499891281128

Test set accuracy 0.9661000370979309

Epoch 4 in 0.49 sec

Training set accuracy 0.9800000190734863

Test set accuracy 0.9682000279426575

Epoch 5 in 0.49 sec

Training set accuracy 0.984333336353302

Test set accuracy 0.969200074672699

Epoch 6 in 0.50 sec

Training set accuracy 0.9868333339691162

Test set accuracy 0.9705000519752502

Epoch 7 in 0.51 sec

Training set accuracy 0.9889833331108093

Test set accuracy 0.971000075340271

Epoch 8 in 0.49 sec

Training set accuracy 0.9921333193778992

Test set accuracy 0.9735000729560852

Epoch 9 in 0.50 sec

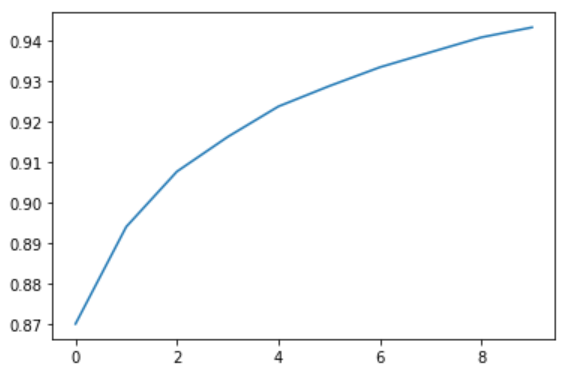
Training set accuracy 0.9933500289916992

Test set accuracy 0.9730000495910645

5. (training accuracy plot on the right)

Slow convergence: learning rate = 0.01

Epoch 0 in 0.90 sec

Training set accuracy 0.8627166748046875

Test set accuracy 0.8671000599861145

Epoch 1 in 0.45 sec

Training set accuracy 0.8939833641052246

Test set accuracy 0.8987000584602356

Epoch 2 in 0.48 sec

Training set accuracy 0.9076666831970215

Test set accuracy 0.9103000164031982

Epoch 3 in 0.44 sec

Training set accuracy 0.9162333607673645

Test set accuracy 0.9165000319480896

Epoch 4 in 0.45 sec

Training set accuracy 0.9237666726112366

Test set accuracy 0.9234000444412231

Epoch 5 in 0.44 sec

Training set accuracy 0.9287999868392944

Test set accuracy 0.9283000230789185

Epoch 6 in 0.46 sec

Training set accuracy 0.9334666728973389

Test set accuracy 0.9324000477790833

Epoch 7 in 0.43 sec

Training set accuracy 0.9371833205223083

Test set accuracy 0.9353000521659851

Epoch 8 in 0.47 sec

Training set accuracy 0.9408666491508484

Test set accuracy 0.9381000399589539

Epoch 9 in 0.48 sec

Training set accuracy 0.9433333277702332

Test set accuracy 0.9393000602722168

Oscillation but convergence: learning rate = 1

Epoch 0 in 0.47 sec

Training set accuracy 0.8757833242416382

Test set accuracy 0.8737000226974487

Epoch 1 in 0.50 sec

Training set accuracy 0.9419833421707153

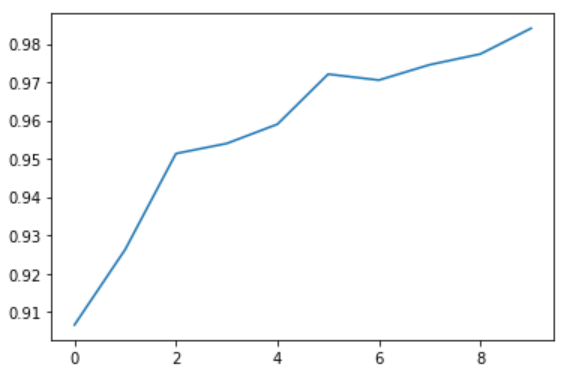
Test set accuracy 0.9349000453948975

Epoch 2 in 0.52 sec

Training set accuracy 0.9506666660308838

Test set accuracy 0.9444000720977783

Epoch 3 in 0.56 sec

Training set accuracy 0.9575666785240173

Test set accuracy 0.9467000365257263

Epoch 4 in 0.58 sec

Training set accuracy 0.9659833312034607

Test set accuracy 0.9533000588417053

Epoch 5 in 0.58 sec

Training set accuracy 0.9727333188056946

Test set accuracy 0.9635000228881836

Epoch 6 in 0.50 sec

Training set accuracy 0.9703166484832764

Test set accuracy 0.9602000713348389

Epoch 7 in 0.52 sec

Training set accuracy 0.9745500087738037

Test set accuracy 0.961400032043457

Epoch 8 in 0.50 sec

Training set accuracy 0.9807167053222656

Test set accuracy 0.9663000702857971

Epoch 9 in 0.52 sec

Training set accuracy 0.9824333190917969

Test set accuracy 0.969700038433075

Oscillation but non-convergence: learning rate = 2

Epoch 0 in 1.02 sec

Training set accuracy 0.09751667082309723

Test set accuracy 0.09740000218153

Epoch 1 in 0.50 sec

Training set accuracy 0.09035000205039978

Test set accuracy 0.08920000493526459

Epoch 2 in 0.47 sec

Training set accuracy 0.11236666887998581

Test set accuracy 0.11350000649690628

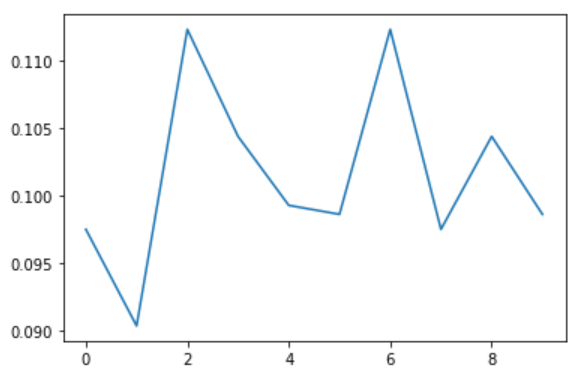
Epoch 3 in 0.46 sec

Training set accuracy 0.10441666841506958

Test set accuracy 0.10280000418424606

Epoch 4 in 0.45 sec

Training set accuracy 0.09930000454187393

Test set accuracy 0.10320000350475311

Epoch 5 in 0.46 sec

Training set accuracy 0.09863333404064178

Test set accuracy 0.0958000048995018

Epoch 6 in 0.46 sec

Training set accuracy 0.11236666887998581

Test set accuracy 0.11350000649690628

Epoch 7 in 0.47 sec

Training set accuracy 0.09751667082309723

Test set accuracy 0.09740000218153

Epoch 8 in 0.45 sec

Training set accuracy 0.10441666841506958

Test set accuracy 0.10280000418424606

Epoch 9 in 0.47 sec

Training set accuracy 0.09863333404064178

Test set accuracy 0.0958000048995018

6. Underfitting: layer\_sizes = [784, 1, 1, 10]

Epoch 0 in 1.22 sec

Training set accuracy 0.21115000545978546

Test set accuracy 0.2086000144481659

Epoch 1 in 0.44 sec

Training set accuracy 0.21220000088214874

Test set accuracy 0.21090000867843628

Epoch 2 in 0.40 sec

Training set accuracy 0.21213333308696747

Test set accuracy 0.21000000834465027

Epoch 3 in 0.40 sec

Training set accuracy 0.21303333342075348

Test set accuracy 0.2111000120639801

Epoch 4 in 0.40 sec

Training set accuracy 0.21076667308807373

Test set accuracy 0.2079000025987625

Epoch 5 in 0.40 sec

Training set accuracy 0.2117166668176651

Test set accuracy 0.20940001308918

Epoch 6 in 0.44 sec

Training set accuracy 0.21164999902248383

Test set accuracy 0.20900000631809235

Epoch 7 in 0.40 sec

Training set accuracy 0.21295000612735748

Test set accuracy 0.21040001511573792

Epoch 8 in 0.43 sec

Training set accuracy 0.21310000121593475

Test set accuracy 0.20940001308918

Epoch 9 in 0.39 sec

Training set accuracy 0.21338333189487457

Test set accuracy 0.21040001511573792

7. Overfitting: layer\_sizes = [784, 1024, 1024, 10]; num\_epochs = 50; create\_outliers=True

Epoch 0 in 1.00 sec

Training set accuracy 0.4868333339691162

Test set accuracy 0.8271000385284424

Epoch 1 in 0.49 sec

Training set accuracy 0.5063333511352539

Test set accuracy 0.8406000137329102

Epoch 2 in 0.46 sec

Training set accuracy 0.5160666704177856

Test set accuracy 0.8385000228881836

Epoch 3 in 0.50 sec

Training set accuracy 0.5252666473388672

Test set accuracy 0.8416000604629517

Epoch 4 in 0.45 sec

Training set accuracy 0.5402666926383972

Test set accuracy 0.8326000571250916

Epoch 5 in 0.47 sec

Training set accuracy 0.5518666505813599

Test set accuracy 0.7945000529289246

Epoch 6 in 0.44 sec

Training set accuracy 0.5556833148002625

Test set accuracy 0.7870000600814819

Epoch 7 in 0.47 sec

Training set accuracy 0.5856500267982483

Test set accuracy 0.8047000169754028

Epoch 8 in 0.46 sec

Training set accuracy 0.5877333283424377

Test set accuracy 0.7749000191688538

Epoch 9 in 0.48 sec

Training set accuracy 0.6187000274658203

Test set accuracy 0.7550000548362732

Epoch 10 in 0.47 sec

Training set accuracy 0.6121833324432373

Test set accuracy 0.7252000570297241

Epoch 11 in 0.45 sec

Training set accuracy 0.6636000275611877

Test set accuracy 0.7686000466346741

Epoch 12 in 0.46 sec

Training set accuracy 0.6808333396911621

Test set accuracy 0.7438000440597534

Epoch 13 in 0.44 sec

Training set accuracy 0.6863666772842407

Test set accuracy 0.706000030040741

Epoch 14 in 0.46 sec

Training set accuracy 0.7089499831199646

Test set accuracy 0.6793000102043152

Epoch 15 in 0.42 sec

Training set accuracy 0.7196999788284302

Test set accuracy 0.6629000306129456

Epoch 16 in 0.46 sec

Training set accuracy 0.7662667036056519

Test set accuracy 0.6637000441551208

Epoch 17 in 0.42 sec

Training set accuracy 0.785966694355011

Test set accuracy 0.6829000115394592

Epoch 18 in 0.45 sec

Training set accuracy 0.8090500235557556

Test set accuracy 0.6338000297546387

Epoch 19 in 0.43 sec

Training set accuracy 0.824916660785675

Test set accuracy 0.659000039100647

Epoch 20 in 0.52 sec

Training set accuracy 0.8390333652496338

Test set accuracy 0.6429000496864319

Epoch 21 in 0.45 sec

Training set accuracy 0.8325166702270508

Test set accuracy 0.6018000245094299

Epoch 22 in 0.44 sec

Training set accuracy 0.8649333715438843

Test set accuracy 0.6093000173568726

Epoch 23 in 0.44 sec

Training set accuracy 0.8950833678245544

Test set accuracy 0.6377000212669373

Epoch 24 in 0.43 sec

Training set accuracy 0.9064666628837585

Test set accuracy 0.5840000510215759

Epoch 25 in 0.43 sec

Training set accuracy 0.9083666801452637

Test set accuracy 0.6249000430107117

Epoch 26 in 0.46 sec

Training set accuracy 0.9273000359535217

Test set accuracy 0.6517000198364258

Epoch 27 in 0.44 sec

Training set accuracy 0.9330166578292847

Test set accuracy 0.6289000511169434

Epoch 28 in 0.43 sec

Training set accuracy 0.9493499994277954

Test set accuracy 0.6568000316619873

Epoch 29 in 0.44 sec

Training set accuracy 0.956516683101654

Test set accuracy 0.6318000555038452

Epoch 30 in 0.42 sec

Training set accuracy 0.9604499936103821

Test set accuracy 0.6577000021934509

Epoch 31 in 0.42 sec

Training set accuracy 0.9635666608810425

Test set accuracy 0.659500002861023

Epoch 32 in 0.43 sec

Training set accuracy 0.9565500020980835

Test set accuracy 0.6285000443458557

Epoch 33 in 0.44 sec

Training set accuracy 0.9696000218391418

Test set accuracy 0.629800021648407

Epoch 34 in 0.44 sec

Training set accuracy 0.9702000021934509

Test set accuracy 0.6482000350952148

Epoch 35 in 0.45 sec

Training set accuracy 0.9759666919708252

Test set accuracy 0.6378000378608704

Epoch 36 in 0.43 sec

Training set accuracy 0.972516655921936

Test set accuracy 0.6288000345230103

Epoch 37 in 0.42 sec

Training set accuracy 0.9734833240509033

Test set accuracy 0.6488000154495239

Epoch 38 in 0.42 sec

Training set accuracy 0.9805166721343994

Test set accuracy 0.6380000114440918

Epoch 39 in 0.42 sec

Training set accuracy 0.9778500199317932

Test set accuracy 0.6522000432014465

Epoch 40 in 0.42 sec

Training set accuracy 0.978783369064331

Test set accuracy 0.6229000091552734

Epoch 41 in 0.42 sec

Training set accuracy 0.9770500063896179

Test set accuracy 0.6455000042915344

Epoch 42 in 0.54 sec

Training set accuracy 0.9789833426475525

Test set accuracy 0.6625000238418579

Epoch 43 in 0.43 sec

Training set accuracy 0.9837333559989929

Test set accuracy 0.6265000104904175

Epoch 44 in 0.44 sec

Training set accuracy 0.9850000143051147

Test set accuracy 0.6150000095367432

Epoch 45 in 0.45 sec

Training set accuracy 0.9859499931335449

Test set accuracy 0.6467000246047974

Epoch 46 in 0.45 sec

Training set accuracy 0.985883355140686

Test set accuracy 0.6465000510215759

Epoch 47 in 0.44 sec

Training set accuracy 0.9857833385467529

Test set accuracy 0.6454000473022461

Epoch 48 in 0.45 sec

Training set accuracy 0.9869833588600159

Test set accuracy 0.6300000548362732

Epoch 49 in 0.45 sec

Training set accuracy 0.9899166822433472

Test set accuracy 0.6450000405311584

Problem 2.

The following hyperparameters are used: learning rate = 0.01, batch size = 32, epoch = 10, momentum = 0.9 architecture of the network:

init\_random\_params, predict = stax.serial(

    stax.Conv(32, (3, 3), strides=(1, 1)),

    stax.Relu,

    stax.MaxPool((2, 2), strides=(2, 2)),

    stax.Conv(64, (3, 3), strides=(1, 1)),

    stax.Relu,

    stax.Conv(64, (3, 3), strides=(1, 1)),

    stax.Relu,

    stax.MaxPool((2, 2), strides=(2, 2)),

    stax.Flatten,

    stax.Dense(100),

    stax.Relu,

    stax.Dense(10),

)

The highest accuracy with this setup is 98.26%.

Test set loss, accuracy (%): (0.84, 96.11)

Test set loss, accuracy (%): (0.75, 97.28)

Test set loss, accuracy (%): (0.70, 97.47)

Test set loss, accuracy (%): (0.71, 97.99)

Test set loss, accuracy (%): (0.66, 98.04)

Test set loss, accuracy (%): (0.65, 98.12)

Test set loss, accuracy (%): (0.71, 98.26)

Test set loss, accuracy (%): (0.71, 97.95)

Test set loss, accuracy (%): (0.67, 98.16)

Test set loss, accuracy (%): (0.72, 98.07)

It is just taking too long to blindly searching for a better hyperparameter set that reaches 99% accuracy rate. However, with the same architecture, in TensorFlow it manages to reach above 99% (99.05%) accuracy rate…

The code in TF is below (<https://machinelearningmastery.com/how-to-develop-a-convolutional-neural-network-from-scratch-for-mnist-handwritten-digit-classification/>):

# deeper cnn model for mnist

from numpy import mean

from numpy import std

from matplotlib import pyplot

from sklearn.model\_selection import KFold

from keras.datasets import mnist

from keras.utils import to\_categorical

from keras.models import Sequential

from keras.layers import Conv2D

from keras.layers import MaxPooling2D

from keras.layers import Dense

from keras.layers import Flatten

from keras.optimizers import SGD

# load train and test dataset

def load\_dataset():

  # load dataset

  (trainX, trainY), (testX, testY) = mnist.load\_data()

  # reshape dataset to have a single channel

  trainX = trainX.reshape((trainX.shape[0], 28, 28, 1))

  testX = testX.reshape((testX.shape[0], 28, 28, 1))

  # one hot encode target values

  trainY = to\_categorical(trainY)

  testY = to\_categorical(testY)

  return trainX, trainY, testX, testY

# scale pixels

def prep\_pixels(train, test):

  # convert from integers to floats

  train\_norm = train.astype('float32')

  test\_norm = test.astype('float32')

  # normalize to range 0-1

  train\_norm = train\_norm / 255.0

  test\_norm = test\_norm / 255.0

  # return normalized images

  return train\_norm, test\_norm

# define cnn model

def define\_model():

  model = Sequential()

  model.add(Conv2D(32, (3, 3), activation='relu', kernel\_initializer='he\_uniform', input\_shape=(28, 28, 1)))

  model.add(MaxPooling2D((2, 2)))

  model.add(Conv2D(64, (3, 3), activation='relu', kernel\_initializer='he\_uniform'))

  model.add(Conv2D(64, (3, 3), activation='relu', kernel\_initializer='he\_uniform'))

  model.add(MaxPooling2D((2, 2)))

  model.add(Flatten())

  model.add(Dense(100, activation='relu', kernel\_initializer='he\_uniform'))

  model.add(Dense(10, activation='softmax'))

  # compile model

  opt = SGD(lr=0.01, momentum=0.9)

  model.compile(optimizer=opt, loss='categorical\_crossentropy', metrics=['accuracy'])

  return model

# evaluate a model using k-fold cross-validation

def evaluate\_model(dataX, dataY, n\_folds=5):

  scores, histories = list(), list()

  # prepare cross validation

  kfold = KFold(n\_folds, shuffle=True, random\_state=1)

  # enumerate splits

  for train\_ix, test\_ix in kfold.split(dataX):

    # define model

    model = define\_model()

    # select rows for train and test

    trainX, trainY, testX, testY = dataX[train\_ix], dataY[train\_ix], dataX[test\_ix], dataY[test\_ix]

    # fit model

    history = model.fit(trainX, trainY, epochs=10, batch\_size=32, validation\_data=(testX, testY), verbose=0)

    # evaluate model

    \_, acc = model.evaluate(testX, testY, verbose=0)

    print('> %.3f' % (acc \* 100.0))

    # stores scores

    scores.append(acc)

    histories.append(history)

  return scores, histories

# plot diagnostic learning curves

def summarize\_diagnostics(histories):

  for i in range(len(histories)):

    # plot loss

    pyplot.subplot(2, 1, 1)

    pyplot.title('Cross Entropy Loss')

    pyplot.plot(histories[i].history['loss'], color='blue', label='train')

    pyplot.plot(histories[i].history['val\_loss'], color='orange', label='test')

    # plot accuracy

    pyplot.subplot(2, 1, 2)

    pyplot.title('Classification Accuracy')

    pyplot.plot(histories[i].history['accuracy'], color='blue', label='train')

    pyplot.plot(histories[i].history['val\_accuracy'], color='orange', label='test')

  pyplot.show()

# summarize model performance

def summarize\_performance(scores):

  # print summary

  print('Accuracy: mean=%.3f std=%.3f, n=%d' % (mean(scores)\*100, std(scores)\*100, len(scores)))

  # box and whisker plots of results

  pyplot.boxplot(scores)

  pyplot.show()

# run the test harness for evaluating a model

def run\_test\_harness():

  # load dataset

  trainX, trainY, testX, testY = load\_dataset()

  # prepare pixel data

  trainX, testX = prep\_pixels(trainX, testX)

  # evaluate model

  scores, histories = evaluate\_model(trainX, trainY)

  # learning curves

  summarize\_diagnostics(histories)

  # summarize estimated performance

  summarize\_performance(scores)

# entry point, run the test harness

run\_test\_harness()

Results:

> 99.033

> 99.050

> 98.742

> 99.183

> 98.700

Python Code:

# -\*- coding: utf-8 -\*-

"""Copy of ECE1513H - Assignment 4 boilerplate

Automatically generated by Colaboratory.

Original file is located at

https://colab.research.google.com/drive/1lmsYRDtum0ot25W11w8UI6g0SmkGJ0ei

Let's first get the imports out of the way.

"""

import array

import gzip

import itertools

import numpy

import numpy.random as npr

import os

import struct

import time

from os import path

import urllib.request

import matplotlib.pyplot as plt

import jax.numpy as np

from jax.api import jit, grad

from jax.config import config

from jax.scipy.special import logsumexp

from jax import random

"""The following cell contains boilerplate code to download and load MNIST data."""

\_DATA = "/tmp/"

def \_download(url, filename):

"""Download a url to a file in the JAX data temp directory."""

if not path.exists(\_DATA):

os.makedirs(\_DATA)

out\_file = path.join(\_DATA, filename)

if not path.isfile(out\_file):

urllib.request.urlretrieve(url, out\_file)

print("downloaded {} to {}".format(url, \_DATA))

def \_partial\_flatten(x):

"""Flatten all but the first dimension of an ndarray."""

return numpy.reshape(x, (x.shape[0], -1))

def \_one\_hot(x, k, dtype=numpy.float32):

"""Create a one-hot encoding of x of size k."""

return numpy.array(x[:, None] == numpy.arange(k), dtype)

def mnist\_raw():

"""Download and parse the raw MNIST dataset."""

# CVDF mirror of http://yann.lecun.com/exdb/mnist/

base\_url = "https://storage.googleapis.com/cvdf-datasets/mnist/"

def parse\_labels(filename):

with gzip.open(filename, "rb") as fh:

\_ = struct.unpack(">II", fh.read(8))

return numpy.array(array.array("B", fh.read()), dtype=numpy.uint8)

def parse\_images(filename):

with gzip.open(filename, "rb") as fh:

\_, num\_data, rows, cols = struct.unpack(">IIII", fh.read(16))

return numpy.array(array.array("B", fh.read()),

dtype=numpy.uint8).reshape(num\_data, rows, cols)

for filename in ["train-images-idx3-ubyte.gz", "train-labels-idx1-ubyte.gz",

"t10k-images-idx3-ubyte.gz", "t10k-labels-idx1-ubyte.gz"]:

\_download(base\_url + filename, filename)

train\_images = parse\_images(path.join(\_DATA, "train-images-idx3-ubyte.gz"))

train\_labels = parse\_labels(path.join(\_DATA, "train-labels-idx1-ubyte.gz"))

test\_images = parse\_images(path.join(\_DATA, "t10k-images-idx3-ubyte.gz"))

test\_labels = parse\_labels(path.join(\_DATA, "t10k-labels-idx1-ubyte.gz"))

return train\_images, train\_labels, test\_images, test\_labels

#def mnist(create\_outliers=False):

def mnist(create\_outliers=True):

"""Download, parse and process MNIST data to unit scale and one-hot labels."""

train\_images, train\_labels, test\_images, test\_labels = mnist\_raw()

train\_images = \_partial\_flatten(train\_images) / numpy.float32(255.)

test\_images = \_partial\_flatten(test\_images) / numpy.float32(255.)

train\_labels = \_one\_hot(train\_labels, 10)

test\_labels = \_one\_hot(test\_labels, 10)

if create\_outliers:

mum\_outliers = 30000

perm = numpy.random.RandomState(0).permutation(mum\_outliers)

train\_images[:mum\_outliers] = train\_images[:mum\_outliers][perm]

return train\_images, train\_labels, test\_images, test\_labels

def shape\_as\_image(images, labels, dummy\_dim=False):

target\_shape = (-1, 1, 28, 28, 1) if dummy\_dim else (-1, 28, 28, 1)

return np.reshape(images, target\_shape), labels

#train\_images, train\_labels, test\_images, test\_labels = mnist(create\_outliers=False)

train\_images, train\_labels, test\_images, test\_labels = mnist(create\_outliers=True)

num\_train = train\_images.shape[0]

"""# \*\*Problem 1\*\*

This function computes the output of a fully-connected neural network (i.e., multilayer perceptron) by iterating over all of its layers and:

1. taking the `activations` of the previous layer (or the input itself for the first hidden layer) to compute the `outputs` of a linear classifier. Recall the lectures: `outputs` is what we wrote $z=w\cdot x + b$ where $x$ is the input to the linear classifier.

2. applying a non-linear activation. Here we will use $tanh$.

Complete the following cell to compute `outputs` and `activations`.

"""

def predict(params, inputs):

activations = inputs

for w, b in params[:-1]:

outputs = np.dot(activations, w) + b

activations = np.tanh(outputs)

final\_w, final\_b = params[-1]

logits = np.dot(activations, final\_w) + final\_b

return logits - logsumexp(logits, axis=1, keepdims=True)

"""The following cell computes the loss of our model. Here we are using cross-entropy combined with a softmax but the implementation uses the `LogSumExp` trick for numerical stability. This is why our previous function `predict` returns the logits to which we substract the `logsumexp` of logits. We discussed this in class but you can read more about it [here](https://blog.feedly.com/tricks-of-the-trade-logsumexp/).

Complete the return line. Recall that the loss is defined as :

$$ l(X, Y) = -\frac{1}{n} \sum\_{i\in 1..n} \sum\_{j\in 1.. K}y\_j^{(i)} \log(f\_j(x^{(i)})) = -\frac{1}{n} \sum\_{i\in 1..n} \sum\_{j\in 1.. K}y\_j^{(i)} \log\left(\frac{z\_j^{(i)}}{\sum\_{k\in 1..K}z\_k^{(i)}}\right) $$

where $X$ is a matrix containing a batch of $n$ training inputs, and $Y$ a matrix containing a batch of one-hot encoded labels defined over $K$ labels. Here $z\_j^{(i)}$ is the logits (i.e., input to the softmax) of the model on the example $i$ of our batch of training examples $X$.

"""

def loss(params, batch):

inputs, targets = batch

preds = predict(params, inputs)

ce = -np.mean(np.sum(targets\*preds, axis=1))

print(ce)

return ce

"""The following cell defines the accuracy of our model and how to initialize its parameters."""

def accuracy(params, batch):

inputs, targets = batch

target\_class = np.argmax(targets, axis=1)

predicted\_class = np.argmax(predict(params, inputs), axis=1)

return np.mean(predicted\_class == target\_class)

def init\_random\_params(layer\_sizes, rng=npr.RandomState(0)):

scale = 0.1

return [(scale \* rng.randn(m, n), scale \* rng.randn(n))

for m, n, in zip(layer\_sizes[:-1], layer\_sizes[1:])]

"""The following line defines our architecture with the number of neurons contained in each fully-connected layer (the first layer has 784 neurons because MNIST images are 28\*28=784 pixels and the last layer has 10 neurons because MNIST has 10 classes)"""

layer\_sizes = [784, 1024, 1024, 10]

# [784, 1024, 128, 10]

"""The following cell creates a Python generator for our dataset. It outputs one batch of $n$ training examples at a time."""

batch\_size = 32

num\_complete\_batches, leftover = divmod(num\_train, batch\_size)

num\_batches = num\_complete\_batches + bool(leftover)

def data\_stream():

rng = npr.RandomState(0)

while True:

perm = rng.permutation(num\_train)

for i in range(num\_batches):

batch\_idx = perm[i \* batch\_size:(i + 1) \* batch\_size]

yield train\_images[batch\_idx], train\_labels[batch\_idx]

batches = data\_stream()

"""We are now ready to define our optimizer. Here we use mini-batch stochastic gradient descent. Complete `<w UPDATE RULE>` and `<b UPDATE RULE>` using the update rule we saw in class. Recall that `dw` is the partial derivative of the `loss` with respect to `w` and `learning\_rate` is the learning rate of gradient descent."""

learning\_rate = 0.1

# 0.01: slow

# 1: oscillate but converge

# 2: oscillate but non-converge

@jit

def update(params, batch):

grads = grad(loss)(params, batch)

return [(w - learning\_rate \* dw, b - learning\_rate \* db)

for (w, b), (dw, db) in zip(params, grads)]

"""This is now the proper training loop for our fully-connected neural network."""

num\_epochs = 50

#num\_epochs = 10

params = init\_random\_params(layer\_sizes)

for epoch in range(num\_epochs):

start\_time = time.time()

for \_ in range(num\_batches):

params = update(params, next(batches))

epoch\_time = time.time() - start\_time

train\_acc = accuracy(params, (train\_images, train\_labels))

test\_acc = accuracy(params, (test\_images, test\_labels))

print("Epoch {} in {:0.2f} sec".format(epoch, epoch\_time))

print("Training set accuracy {}".format(train\_acc))

print("Test set accuracy {}".format(test\_acc))

"""# \*\*Problem 2\*\*

Before we get started, we need to import two small libraries that contain boilerplate code for common neural network layer types and for optimizers like mini-batch SGD.

"""

from jax.experimental import optimizers

from jax.experimental import stax

"""Here is a fully-connected neural network architecture, like the one of Problem 1, but this time defined with `stax`"""

init\_random\_params, predict = stax.serial(

stax.Conv(32, (3, 3), strides=(1, 1)),

stax.Relu,

stax.MaxPool((2, 2), strides=(2, 2)),

stax.Conv(64, (3, 3), strides=(1, 1)),

stax.Relu,

stax.Conv(64, (3, 3), strides=(1, 1)),

stax.Relu,

stax.MaxPool((2, 2), strides=(2, 2)),

stax.Flatten,

stax.Dense(100),

stax.Relu,

stax.Dense(10),

)

"""We redefine the cross-entropy loss for this model. As done in Problem 1, complete the return line below (it's identical)."""

def loss(params, batch):

inputs, targets = batch

logits = predict(params, inputs)

preds = stax.logsoftmax(logits)

return -np.mean(np.sum(targets\*preds, axis=1))

"""Next, we define the mini-batch SGD optimizer, this time with the optimizers library in JAX."""

learning\_rate = 0.01

opt\_init, opt\_update, get\_params = optimizers.momentum(learning\_rate, 0.9)

@jit

def update(\_, i, opt\_state, batch):

params = get\_params(opt\_state)

return opt\_update(i, grad(loss)(params, batch), opt\_state)

"""The next cell contains our training loop, very similar to Problem 1."""

num\_epochs = 10

key = random.PRNGKey(123)

\_, init\_params = init\_random\_params(key, (-1, 28, 28, 1))

opt\_state = opt\_init(init\_params)

itercount = itertools.count()

for epoch in range(1, num\_epochs + 1):

for \_ in range(num\_batches):

opt\_state = update(key, next(itercount), opt\_state, shape\_as\_image(\*next(batches)))

params = get\_params(opt\_state)

test\_acc = accuracy(params, shape\_as\_image(test\_images, test\_labels))

test\_loss = loss(params, shape\_as\_image(test\_images, test\_labels))

print('Test set loss, accuracy (%): ({:.2f}, {:.2f})'.format(test\_loss, 100 \* test\_acc))