ECE 1513 Introduction to Machine Learning

Assignment 4

Zihao Jiao (1002213428)

Date: March 5th, 2020

Problem 1

1. Complete the cell which computes the neural network's prediction through the function predict.

```
[ ]  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)
```

2. Complete the cell which computes the neural network's loss through the function loss.

```
[ ]  def loss(params, batch):
       inputs, targets = batch
       preds = predict(params, inputs)

       return -np.sum(targets*preds)/len(targets)
```

3. Complete the cell which defines mini-batch stochastic gradient descent through the function update.

```
[ ]  learning_rate = 0.1

     @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)]
```

4. Make sure you are able to train the neural network to an accuracy of about 97%.

```
Epoch 0 in 2.00 sec
Training set accuracy 0.9401500225067139
Test set accuracy 0.9377000331878662
Epoch 1 in 0.44 sec
Training set accuracy 0.9592833518981934
Test set accuracy 0.95250004529953
Epoch 2 in 0.43 sec
Training set accuracy 0.9681666493415833
Test set accuracy 0.9607000350952148
Epoch 3 in 0.42 sec
Training set accuracy 0.9759166836738586
Test set accuracy 0.9663000702857971
Epoch 4 in 0.44 sec
Training set accuracy 0.9795500040054321
Test set accuracy 0.9676000475883484
Epoch 5 in 0.43 sec
Training set accuracy 0.982616662979126
Test set accuracy 0.970300018787384
Epoch 6 in 0.41 sec
Training set accuracy 0.9865833520889282
Test set accuracy 0.9716000556945801
Epoch 7 in 0.43 sec
Training set accuracy 0.9892333149909973
Test set accuracy 0.9736000299453735
Epoch 8 in 0.42 sec
Training set accuracy 0.9911666512489319
Test set accuracy 0.9741000533103943
Epoch 9 in 0.44 sec
Training set accuracy 0.992983341217041
Test set accuracy 0.9746000170707703
```

5. Modify the learning rate of your mini-batch SGD and report (a) a value that results in slow convergence, (b) a value that results in oscillations but still converges, and (c) a value that results to instabilities and diverges.
    - Learning rate = 0.0001 results in slow convergence

```
Epoch 0 in 1.97 sec
Training set accuracy 0.163183331489563
Test set accuracy 0.15280000865459442
Epoch 1 in 0.44 sec
Training set accuracy 0.23115000128746033
Test set accuracy 0.22300000488758087
Epoch 2 in 0.47 sec
Training set accuracy 0.30515000224113464
Test set accuracy 0.29920002818107605
Epoch 3 in 0.44 sec
Training set accuracy 0.37263333797454834
Test set accuracy 0.3678000271320343
Epoch 4 in 0.44 sec
Training set accuracy 0.4286166727542877
Test set accuracy 0.42590001225471497
Epoch 5 in 0.48 sec
Training set accuracy 0.46994999051094055
Test set accuracy 0.46970000863075256
Epoch 6 in 0.49 sec
Training set accuracy 0.5043833255767822
Test set accuracy 0.5062000155448914
Epoch 7 in 0.43 sec
Training set accuracy 0.5330166816711426
Test set accuracy 0.5336000323295593
Epoch 8 in 0.45 sec
Training set accuracy 0.5577166676521301
Test set accuracy 0.560200035572052
Epoch 9 in 0.43 sec
Training set accuracy 0.5787833333015442
Test set accuracy 0.5818000435829163
```

    - Learning rate = 1.5 results in oscillation and still converge

```
Epoch 0 in 2.00 sec
Training set accuracy 0.8553666472434998
Test set accuracy 0.8552000522613525
Epoch 1 in 0.44 sec
Training set accuracy 0.9092666506767273
Test set accuracy 0.9081000685691833
Epoch 2 in 0.40 sec
Training set accuracy 0.8664000034332275
Test set accuracy 0.8646000623703003
Epoch 3 in 0.42 sec
Training set accuracy 0.9336333274841309
Test set accuracy 0.9318000674247742
Epoch 4 in 0.42 sec
Training set accuracy 0.9189333319664001
Test set accuracy 0.9159000515937805
Epoch 5 in 0.46 sec
Training set accuracy 0.9212999939918518
Test set accuracy 0.9169000387191772
Epoch 6 in 0.41 sec
Training set accuracy 0.9419000148773193
Test set accuracy 0.9341000318527222
Epoch 7 in 0.41 sec
Training set accuracy 0.9575333595275879
Test set accuracy 0.9494000673294067
Epoch 8 in 0.45 sec
Training set accuracy 0.9422500133514404
Test set accuracy 0.9342000484466553
Epoch 9 in 0.42 sec
Training set accuracy 0.9629499912261963
Test set accuracy 0.9545000195503235
```

- Learning rate = 10 results in instability, diverges

```
Epoch 0 in 2.09 sec
Training set accuracy 0.09736666828393936
Test set accuracy 0.0982000082731247
Epoch 1 in 0.43 sec
Training set accuracy 0.09930000454187393
Test set accuracy 0.10320000350475311
Epoch 2 in 0.42 sec
Training set accuracy 0.09863333404064178
Test set accuracy 0.0958000048995018
Epoch 3 in 0.44 sec
Training set accuracy 0.09930000454187393
Test set accuracy 0.10320000350475311
Epoch 4 in 0.43 sec
Training set accuracy 0.09930000454187393
Test set accuracy 0.10320000350475311
Epoch 5 in 0.43 sec
Training set accuracy 0.09930000454187393
Test set accuracy 0.10320000350475311
Epoch 6 in 0.43 sec
Training set accuracy 0.09736666828393936
Test set accuracy 0.0982000082731247
Epoch 7 in 0.43 sec
Training set accuracy 0.09736666828393936
Test set accuracy 0.0982000082731247
Epoch 8 in 0.44 sec
Training set accuracy 0.09736666828393936
Test set accuracy 0.0982000082731247
Epoch 9 in 0.44 sec
Training set accuracy 0.09871666878461838
Test set accuracy 0.09800000488758087
```

6. Modify the number of neurons and/or number of layers that make up the architecture of your neural network. You can do so by modifying the list layer sizes. Report a list of integers that results in a neural network that underfits the data.

   Solution: underfit conditions

   layer_sizes = [784, 3, 10] results underfitting.

   This is three layers, input layer, hidden layer, output layer, neural network.

```
Epoch 0 in 1.89 sec
Training set accuracy 0.6622833609580994
Test set accuracy 0.6627000570297241
Epoch 1 in 0.40 sec
Training set accuracy 0.7145666480064392
Test set accuracy 0.7187000513076782
Epoch 2 in 0.39 sec
Training set accuracy 0.7367333173751831
Test set accuracy 0.742900013923645
Epoch 3 in 0.38 sec
Training set accuracy 0.74795001745224
Test set accuracy 0.7449000477790833
Epoch 4 in 0.39 sec
Training set accuracy 0.7455666661262512
Test set accuracy 0.7460000514984131
Epoch 5 in 0.36 sec
Training set accuracy 0.7398000359535217
Test set accuracy 0.7325000166893005
Epoch 6 in 0.38 sec
Training set accuracy 0.749916672706604
Test set accuracy 0.7473000288009644
Epoch 7 in 0.39 sec
Training set accuracy 0.7678666710853577
Test set accuracy 0.7676000595092773
Epoch 8 in 0.41 sec
Training set accuracy 0.7804999947547913
Test set accuracy 0.7821000218391418
Epoch 9 in 0.38 sec
Training set accuracy 0.7741833329200745
Test set accuracy 0.7712000608444214
```

7. Find a setting in which your neural network overfits. To this end, modify the set of hyperparameters discussed so far, i.e., learning rate and architecture, as well as the number of epochs if that's necessary. You may also find it useful to set create outliers to True and reload the MNIST data by executing mnist() again. This will mislabel half of the training data, which makes it easier to find a setting in which the model overfits. Report the set of hyperparameters that result in a neural network that overfits the data.
Solution: overfit conditions
   - Outliers = True
   - layer_sizes = [784, 2056, 918, 221, 128, 10]
   - learning rate = 0.01
   - num_epoche = 50

```
Epoch 36 in 1.12 sec
Training set accuracy 0.9616000056266785
Test set accuracy 0.588200032711029
Epoch 37 in 1.12 sec
Training set accuracy 0.9650999903678894
Test set accuracy 0.612000048160553
Epoch 38 in 1.11 sec
Training set accuracy 0.9706166982650757
Test set accuracy 0.6010000109672546
Epoch 39 in 1.11 sec
Training set accuracy 0.9770333170890808
Test set accuracy 0.5872000455856323
Epoch 40 in 1.11 sec
Training set accuracy 0.9766833186149597
Test set accuracy 0.5885000228881836
Epoch 41 in 1.11 sec
Training set accuracy 0.9729166626930237
Test set accuracy 0.5866000056266785
Epoch 42 in 1.14 sec
Training set accuracy 0.9796000123023987
Test set accuracy 0.6060000061988831
Epoch 43 in 1.11 sec
Training set accuracy 0.9804999828338623
Test set accuracy 0.593000054359436
Epoch 44 in 1.10 sec
Training set accuracy 0.9809166789054871
Test set accuracy 0.5785000324249268
Epoch 45 in 1.09 sec
Training set accuracy 0.9833333492279053
Test set accuracy 0.594700038433075
Epoch 46 in 1.10 sec
Training set accuracy 0.9869000315666199
Test set accuracy 0.5871000289916992
Epoch 47 in 1.11 sec
Training set accuracy 0.9851666688919067
Test set accuracy 0.5907000303268433
Epoch 48 in 1.11 sec
Training set accuracy 0.9873499870300293
Test set accuracy 0.5817000269889832
Epoch 49 in 1.11 sec
Training set accuracy 0.9898999929428101
Test set accuracy 0.5874000191688538
```

As we can see the testing accuracy significantly smaller than training accuracy, which indicates overfitting.

Problem 2

The architecture is below:

```
init_random_params, predict = stax.serial(
    stax.Conv(256, (5,5),strides = (2,2)),
    stax.Relu,
    stax.Conv(128, (3,3)),
    stax.Relu,
    stax.Conv(32, (3,3)),
    stax.Relu,
    stax.MaxPool((2,2)),
    stax.Flatten,
    stax.Dense(1024),
    stax.Relu,
    stax.Dense(128),
    stax.Relu,
    stax.Dense(10),
)
```

Hyperparameters setting:

- batch size = 50
- learning rate = 0.08
- number of epochs = 20

Accuracy result:

```
Test set loss, accuracy (%): (0.06, 98.36)
Test set loss, accuracy (%): (0.03, 98.94)
Test set loss, accuracy (%): (0.04, 98.69)
Test set loss, accuracy (%): (0.04, 98.87)
Test set loss, accuracy (%): (0.03, 98.99)
Test set loss, accuracy (%): (0.03, 99.13)
Test set loss, accuracy (%): (0.03, 99.12)
Test set loss, accuracy (%): (0.04, 99.04)
Test set loss, accuracy (%): (0.03, 99.11)
Test set loss, accuracy (%): (0.03, 99.15)
Test set loss, accuracy (%): (0.04, 99.02)
Test set loss, accuracy (%): (0.04, 99.08)
Test set loss, accuracy (%): (0.03, 99.22)
Test set loss, accuracy (%): (0.03, 99.23)
Test set loss, accuracy (%): (0.04, 99.21)
Test set loss, accuracy (%): (0.04, 99.23)
Test set loss, accuracy (%): (0.04, 99.06)
Test set loss, accuracy (%): (0.04, 99.07)
Test set loss, accuracy (%): (0.04, 99.27)
Test set loss, accuracy (%): (0.04, 99.27)
```

Statistical results for the 5 runs:

Run	Average mean	Standard deviation
1	99.05302%	0.21178113
2	99.02002%	0.24829412
3	99.06051%	0.24150474
4	99.0715%	0.3134694
5	99.091%	0.27937233

Overall average = 99.05921%

Std: 0.25167

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 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):
  """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)
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.

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)

  return -np.sum(targets*preds)/len(targets)
```

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, 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 = 50
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

@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 = 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))
```

```
Epoch 0 in 2.66 sec
Training set accuracy 0.9605833292007446
Test set accuracy 0.9535000324249268
Epoch 1 in 1.08 sec
Training set accuracy 0.9768000245094299
Test set accuracy 0.9665000438690186
Epoch 2 in 1.09 sec
Training set accuracy 0.9838666915893555
Test set accuracy 0.9708000421524048
Epoch 3 in 1.10 sec
Training set accuracy 0.9891666769981384
Test set accuracy 0.9743000268936157
Epoch 4 in 1.09 sec
Training set accuracy 0.9914166927337646
Test set accuracy 0.9727000594139099
Epoch 5 in 1.09 sec
Training set accuracy 0.9946333169937134
Test set accuracy 0.9759000539779663
Epoch 6 in 1.09 sec
Training set accuracy 0.9973666667938232
Test set accuracy 0.9770000576972961
Epoch 7 in 1.09 sec
Training set accuracy 0.9978833198547363
Test set accuracy 0.9777000546455383
Epoch 8 in 1.08 sec
Training set accuracy 0.9990333318710327
Test set accuracy 0.9782000184059143
Epoch 9 in 1.07 sec
Training set accuracy 0.9993166923522949
Test set accuracy 0.9795000553131104
```

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(256, (5,5),strides = (2,2)),
    stax.Relu,
    stax.Conv(128, (3,3)),
    stax.Relu,
    stax.Conv(32, (3,3)),
    stax.Relu,
    stax.MaxPool((2,2)),
    stax.Flatten,
    stax.Dense(1024),
    stax.Relu,
    stax.Dense(128),
    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.sum(targets*preds)/len(targets)
```

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

```
learning_rate = 0.08
opt_init, opt_update, get_params = optimizers.sgd(learning_rate)

@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.

learning rate, batch size, number of epochs

```
num_epochs = 20

key = random.PRNGKey(123)
_, init_params = init_random_params(key, (-1, 28, 28, 1))
opt_state = opt_init(init_params)
itercount = itertools.count()

acc_list=[]
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
  acc_list.append(test_acc)
```

```
Test set loss, accuracy (%): (0.05, 98.35)
Test set loss, accuracy (%): (0.04, 98.84)
Test set loss, accuracy (%): (0.05, 98.43)
Test set loss, accuracy (%): (0.03, 98.91)
Test set loss, accuracy (%): (0.03, 99.16)
Test set loss, accuracy (%): (0.03, 99.26)
Test set loss, accuracy (%): (0.04, 98.87)
Test set loss, accuracy (%): (0.03, 99.08)
Test set loss, accuracy (%): (0.03, 99.01)
Test set loss, accuracy (%): (0.03, 99.11)
Test set loss, accuracy (%): (0.03, 99.24)
Test set loss, accuracy (%): (0.03, 99.14)
Test set loss, accuracy (%): (0.03, 99.25)
Test set loss, accuracy (%): (0.03, 99.26)
Test set loss, accuracy (%): (0.03, 99.30)
Test set loss, accuracy (%): (0.03, 99.33)
Test set loss, accuracy (%): (0.04, 99.34)
Test set loss, accuracy (%): (0.04, 99.33)
Test set loss, accuracy (%): (0.04, 99.30)
Test set loss, accuracy (%): (0.04, 99.31)
```

```
result = np.array(acc_list)
average_score = result.mean()*100
score_std = result.std()*100

print("Average Accuracy: " + str(average_score)+"%")
print("Variance: " + str(score_std))
```

```
Average Accuracy: 99.091%
Variance: 0.27937233
```