Optimisation Algorithms - Final Assignment

Ernests Kuznecovs - 17332791 - kuznecoe@tcd.ie

2021-2022

Contents

1 Stochastic Gradient Descent: Constant and Adam Step Sizes

Constant	Alpha	Batch Size
Default	0.1	128
Optimal MNIST	0.489	49
Optimal TwNet	1000	2000

Adam	Alpha	Batch Size	Beta1	Beta2
Default	0.01	128	0.9	0.999
Optimal MNIST	0.015	98	0.898	0.9575
Optimal TwNet	1.9	960	0.869	0.662

2 Report

2.1 Model and Dataset 1: MNIST

MNIST model and code obtained from `https://github.com/google/flax/tree/main/examples/mnist`. 60,000 28x28 images and labels used for training, and 10,000 used for testing. The model classifies which of the 10 digits is handwritten in the image.

It is a neural net with convolutions using "softmax cross entropy" loss `https://optax.readthedocs.io/en/latest/api.html#optax.softmax_cross_entropy` with the following configuration:

```
x = nn.Conv(features=32, kernel_size=(3, 3))(x)
x = nn.relu(x)
x = nn.avg_pool(x, window_shape=(2, 2), strides=(2, 2))
x = nn.Conv(features=64, kernel_size=(3, 3))(x)
x = nn.relu(x)
x = nn.avg_pool(x, window_shape=(2, 2), strides=(2, 2))
x = x.reshape((x.shape[0], -1))  # flatten
```

```
    x = nn.Dense(features=256)(x)
    x = nn.relu(x)
    x = nn.Dense(features=10)(x)
```

2.1.1 Selecting Hyper Parameters

Ulitmately global random search was used to find the optimal hyperparameters. However to get a feel for how different hyperparameters effected the performance and a suitable range for the global random search, the algorithms were manually run for a small number of iterations and the loss was observed.

A fixed iteration number was picked across all the runs of global random search as we can be quite sure that larger iterations will yield better results, this way random search runs will not be wasted on low iteration choices by the random search.

The training dataset was used to calculate the loss at the end of a run to try and optimise for generalised performance.

This model was run on a CPU, so only one epoch was ran. 20 runs for each Constant step and Adam.

The ranges were picked as small as thought was appropriate depending on short manual runs of the training.

Hyperparameter ranges for:

- Constant step size:
  - Alpha: 0.4 to 0.8
  - BatchSize: 40 to 90
- Adam:
  - Alpha: 0.001 to 0.1
  - Beta1: 0.5 to 0.99
  - Beta2: 0.5 to 0.99
  - BatchSize: 1 to 128

The resulting best hyperparameters returned by global random search are given in the tables of 1.

2.1.2 Comparison

The algorithms were run with optimal and default parameters 20 times. During the runs the loss of the training batches were recorded of each iteration (ungeneralised). On a CPU, unbatched loss calculations would be quite computationally expensive (since using all of the data to calculate the loss), even if done every 1% of iterations. Theese runs are plotted and compared with each other. The mean loss for an iteration of the runs is given by the darker, opaque line, and the min and max value of the loss for the iteration across runs is the ligher transparent colour coloured across the y axis:

- Fig 1: Constant Default vs Constant Optimal. We see that default has a tighter spread, this is due to both the smaller step size and higher batch size. The smaller step size causes more cautious movements and leaves more iterations to average out its downhill direction while not changing the magnitude of the loss too much. The higher batch size reduces the noise in a single iteration because it uses more of the data to construct the gradients. The opposite we can see in the Optimal parameters for the opposite reasons, it has higher step and lower batch, which performs better on average. It could be that the noisey nature and a lucky score is what caused the global random search to pick the parameters. Perhaps there exited a more consistent param combination but did not get picked because of the high variance one.
- Fig 2: Adam Default vs Optimal. Even though default and optimal params (Beta2 most varied) are not exactly identical, the performance is quite the same. Though we can see that both are quite noisey, and there could be parameters that aren't as noisey. This could be because the alpha and batch size aren't too different.

- Fig 3: Default Constant vs Default Adam. We can see the default adam is better than the default constant, though it is a little bit more noisey than the constant.
- Fig 4: Optimal Constant vs Optimal Adam. We can see that on average the optimal constant and adam perform identical, though the constant one is a bit more noisey. This could be because adam has double the batch size. Also perhaps that the noisey ones are more likely to be picked, and perhaps adams noise is uniform over a larger range of parameters, due to the averaging effects of Beta1 and Beta2.

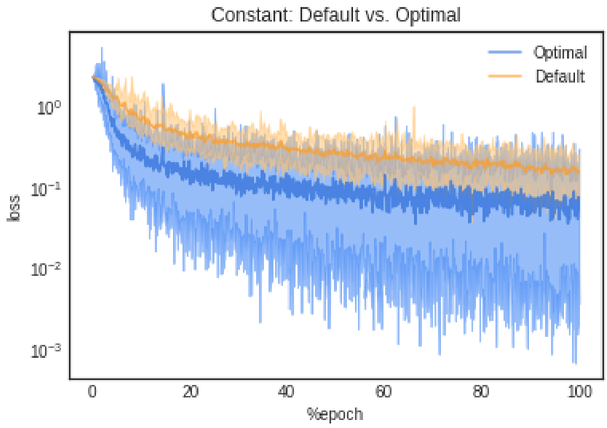

Figure 1

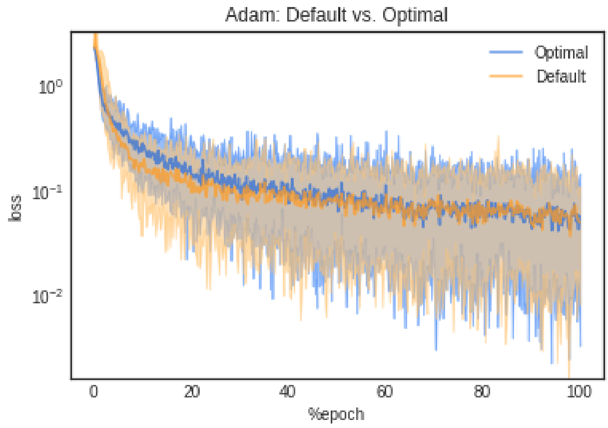

Figure 2

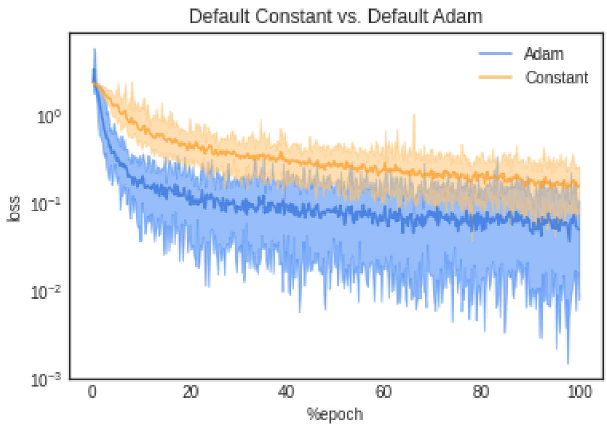

Figure 3

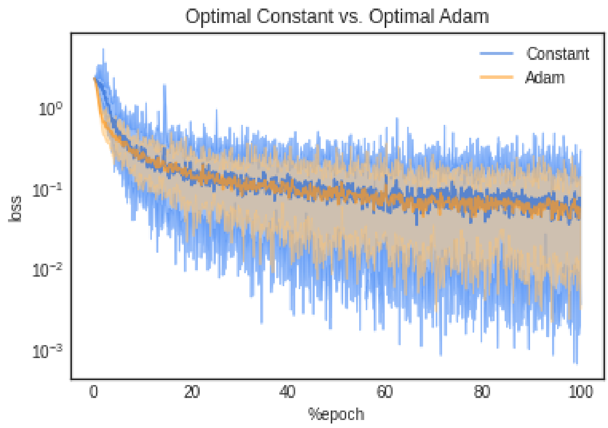

Figure 4

2.1.3 Generalised vs. Ungeneralised

Each algorithm was run once with optimal parameters, after every 12 iterations, the non-batched train and test losses were calculated and recorded. For Constant step size 1224 occured, and for Adam 612 (due to the different batch sizes and constant 1 epoch).

Train and test losses are plotted 5, 6, we can can see that for both, the difference between train and test performance is quite identical throughout the training.

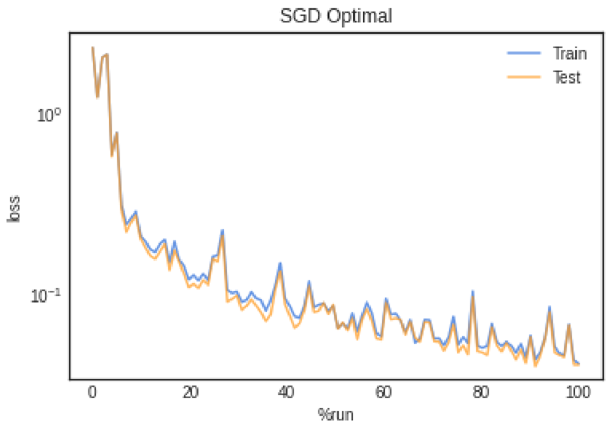

Figure 5

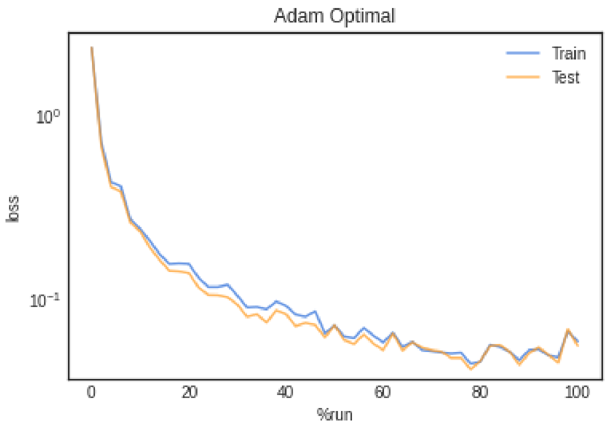

Figure 6

2.2 Model and Dataset 2: Twitter Network Wordvectors

The second dataset is a model and data adapted from my Text Analytics group paper, available at `https://github.com/ErnestKz/TextAnalyticsReport/blob/main/Text_Analytics_Final_Paper%20(1).pdf`. Input data to model is all possible combinations of links between users and corresponding pair of wordvectors counting the word occurances of their posts, output data is a value that represents the strength of the connection, the labelled data comes from whether they are following each other or not. Weights are added on wordvectors such that wordvector similairty between users corpus can maximally overlap with whether they are following each other. The model is a linear regression model. Rather than batching both wordvector indices and edges as done in the paper, only edges are batched to resemble standard stochastic gradient descent.

The loss function is a cosine distance `https://optax.readthedocs.io/en/latest/api.html#optax.cosine_distance` applied to the pair of word vectors for each edge, and then another cosine distance of the resulting edge vector with the edge vector that corresponds to the followage of each edge.

The wordvectors have a large dimension and hence the word counts have to be scaled to a smaller value such that the cosine distance calculation does not cause an floating point overflow.

32,000 edges were used for training. 11,000 edges were used for testing. The size of a wordvector is 3933 dimensions.

2.2.1 Selecting Hyper Parameters

Similarly, the model was run manually with small number of iterations varying the parameters to get a feel for them.

Global Random Search of 20 runs each was performed, though this time the final training loss was mistakingly used to compare performance, though it offers some variance in the analysis.

Another accidental difference in this global random search was that the iteration count (3000) was kept constant rather than the epoch, this lead to varied batch sizes causing different amount of data used across runs, which may not have been ideal and fair across runs.

Hyperparameter ranges for:

- Constant step size:
  - Alpha: 1 to 1000
  - BatchSize: 12 to 2048
- Adam:
  - Alpha: 0.1 to 100
  - Beta1: 0.5 to 0.99
  - Beta2: 0.5 to 0.99
  - BatchSize: 12 to 2049

This time the values of the global random search were collected and plotted. Though it was hard to observe obvious trends in the data due to sparsity of data and it's high dimensionality, but it did give enough intuition on how to iterate on new ranges for the global random search, though this was not done for this report, due to it's time consuming nature and parameters seemed good enough for comparison.

The resulting best hyperparameters returned by global random search are given in the tables of 1.

Step size and batch on alpha needs to be large perhaps because the slope is small due to sparse, uniform nature of the problem, i.e word vectors are sparse, and the ground truth edges dont contain many connections.

Where beta1 in adam is more keeping track of directional information (heavyball-like), beta2 is more keeping track of magnitudional information (RMSProp-like). We see that the optimal beta2 parameter is quite low, causing to forget the previous gradients faster, this makes sense as the word vectors are quite sparse, and might only need a slight adjustment once the algorithm encounters it and not to keep updating after it is gone. It could be that the heavy ball component allows for a continued extra push whenever it eventually encounters a gradient in one the batches that it had seen before.

2.2.2 Comparison

The algorithms were run 20 times with 3000 iterations each (again is quite unfair as the different algorithms have different batch sizes). Unlike the previous, since a GPU was available, nonbatched test loss was recorded every 1% of the total iteration count. The results are plotted in the same fashion:

- Fig 7: Constant Default vs Constant Optimal. Constant default has a tiny alpha compared to the optimal one, so the default one is not making any progress.
- Fig 8: Adam Default vs Adam Optimal. We also see a slow convergence on default adam, though it is not as bad. The optimal alpha is only 2 magnitudes bigger than the default, whereas for constant it was 4. Though we see more jitter on the optimal one, even though batch size is large. The batch size is 8 times larger than the default, which most likely makes it an unfair comparison. It is perhaps going though more of the data, causing more chance to make overfits, and these graphs can show it as they are test losses.
- Fig 9: Default Constant vs Default Adam. Again, default constant alpha is too tiny to be comparable.
- Fig 10: Optimal Constant vs Optimal Adam. Both perform quite similarly. Even though the parameters are quite varied between them in terms of batch size and step size. Though Adam still pulls ahead of Constant.

We see less noise overall than the previous model, this can be because there are not as many degrees of freedom to be noisey i.e it is a linear model rather than a non-linear neural network.

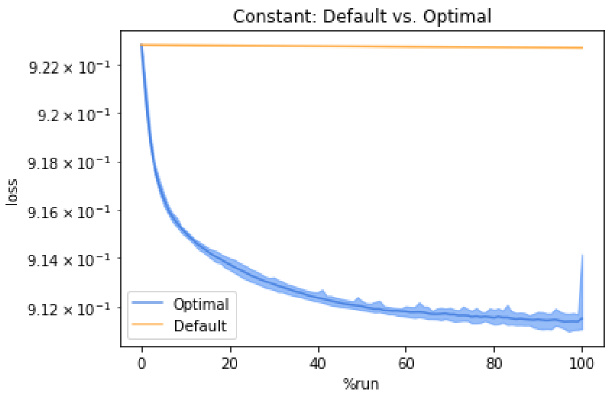

Figure 7

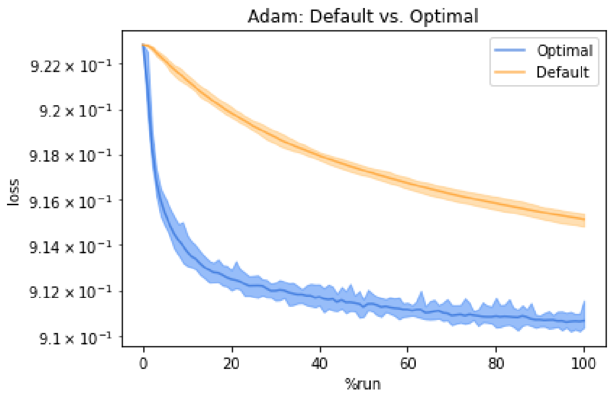

Figure 8

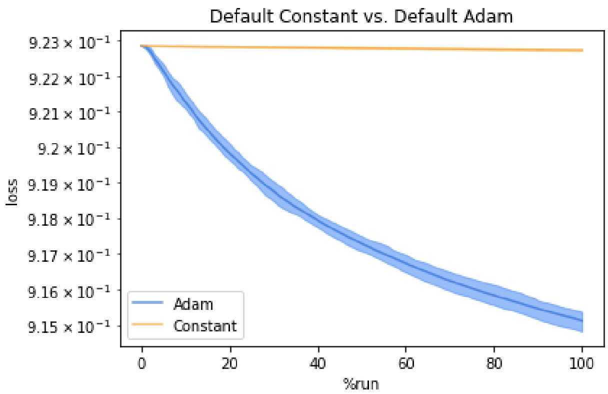

Figure 9

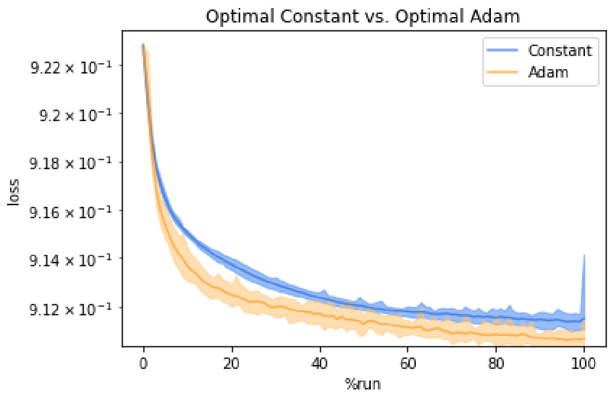

Figure 10

2.2.3 Generalised vs. Ungeneralised

The algorithms were run 100,000 times with optimal parameters evaluating and recording non-batched loss on test and train datasets at every 1% of iterations.

11, 12 we see that both suffer the same problem, the test dataset is indicating that the convergence stopped at early on, and furthermore for Constant step the loss is increasing. The constant step loss increase could be that having the batch size high allows it to fit to the non-generalised peculiarties of the test dataset more. The separation of train and test could also be a symptom of picking the hyperparameter based on the train value, causing the algorithm to behave such that it overfits to the trianing set.

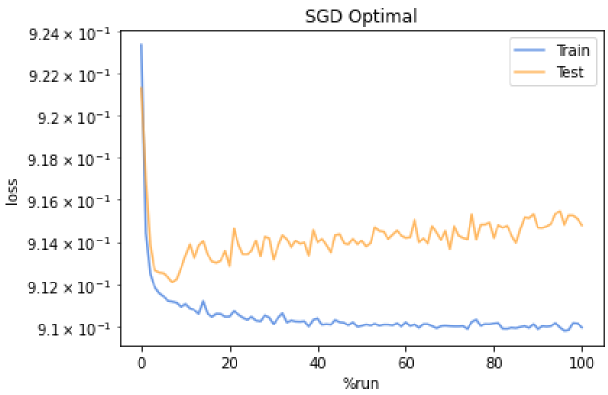

Figure 11

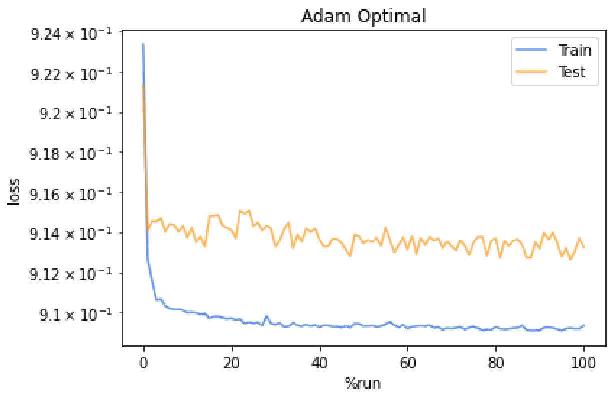

Figure 12

3 Appendix

3.1 Code Listing

```
from typing import Sequence

import numpy as np
import jax
import jax.numpy as jnp
import flax.linen as nn

class MLP(nn.Module):
  features: Sequence[int]

  @nn.compact
  def __call__(self, x):
    for feat in self.features[:-1]:
      x = nn.relu(nn.Dense(feat)(x))
    x = nn.Dense(self.features[-1])(x)
    return x

model = MLP([12, 8, 4])
batch = jnp.ones((32, 10))
variables = model.init(jax.random.PRNGKey(0), batch)
output = model.apply(variables, batch)

import random
from typing import Tuple

import optax
import jax.numpy as jnp
import jax
import numpy as np

BATCH_SIZE = 5
NUM_TRAIN_STEPS = 1_000
RAW_TRAINING_DATA = np.random.randint(255, size=(NUM_TRAIN_STEPS, BATCH_SIZE, 1))

TRAINING_DATA = np.unpackbits(RAW_TRAINING_DATA.astype(np.uint8), axis=-1)
LABELS = jax.nn.one_hot(RAW_TRAINING_DATA % 2, 2).astype(jnp.float32).reshape(NUM_TRAIN_STEPS, BATCH_SIZE, 2)

initial_params = {
    'hidden': jax.random.normal(shape=[8, 32], key=jax.random.PRNGKey(0)),
    'output': jax.random.normal(shape=[32, 2], key=jax.random.PRNGKey(1)),
}


def net(x: jnp.ndarray, params: jnp.ndarray) -> jnp.ndarray:
  x = jnp.dot(x, params['hidden'])
  x = jax.nn.relu(x)
  x = jnp.dot(x, params['output'])
  return x


def loss(params: optax.Params, batch: jnp.ndarray, labels: jnp.ndarray) -> jnp.ndarray:
  y_hat = net(batch, params)

  # optax also provides a number of common loss functions.
  loss_value = optax.sigmoid_binary_cross_entropy(y_hat, labels).sum(axis=-1)

  return loss_value.mean()

def fit(params: optax.Params, optimizer: optax.GradientTransformation) -> optax.Params:
  opt_state = optimizer.init(params)

```

```
  @jax.jit
  def step(params, opt_state, batch, labels):
    loss_value, grads = jax.value_and_grad(loss)(params, batch, labels)
    updates, opt_state = optimizer.update(grads, opt_state, params)
    params = optax.apply_updates(params, updates)
    return params, opt_state, loss_value

  for i, (batch, labels) in enumerate(zip(TRAINING_DATA, LABELS)):
    params, opt_state, loss_value = step(params, opt_state, batch, labels)
    if i % 100 == 0:
      print(f'step {i}, loss: {loss_value}')

  return params

# Finally, we can fit our parametrized function using the Adam optimizer
# provided by optax.
optimizer = optax.adam(learning_rate=1e-2)
optimizer2 = optax.sgd(learning_rate=1e-2)
params = fit(initial_params, optimizer)
params = fit(initial_params, optimizer2)

import matplotlib as mpl
mpl.rcParams['figure.dpi'] = 200
mpl.rcParams['figure.facecolor'] = '1'
import matplotlib.pyplot as plt
plt.style.use('seaborn-white')

import copy
import numpy as np
from sklearn import metrics

from absl import logging
from flax import linen as nn
from flax.metrics import tensorboard
from flax.training import train_state
import jax
import jax.numpy as jnp
import ml_collections
import numpy as np
import optax
import tensorflow_datasets as tfds

class CNN(nn.Module):
  """A simple CNN model."""

  @nn.compact
  def __call__(self, x):
    x = nn.Conv(features=32, kernel_size=(3, 3))(x)
    x = nn.relu(x)
    x = nn.avg_pool(x, window_shape=(2, 2), strides=(2, 2))
    x = nn.Conv(features=64, kernel_size=(3, 3))(x)
    x = nn.relu(x)
    x = nn.avg_pool(x, window_shape=(2, 2), strides=(2, 2))
    x = x.reshape((x.shape[0], -1))  # flatten
    x = nn.Dense(features=256)(x)
    x = nn.relu(x)
    x = nn.Dense(features=10)(x)
    return x

@jax.jit
def apply_model(state, images, labels):
  """Computes gradients, loss and accuracy for a single batch."""
  def loss_fn(params):
    logits = CNN().apply({'params': params}, images)
    one_hot = jax.nn.one_hot(labels, 10)
    loss = jnp.mean(optax.softmax_cross_entropy(logits=logits, labels=one_hot))
    return loss, logits

```

```
  grad_fn = jax.value_and_grad(loss_fn, has_aux=True)
  (loss, logits), grads = grad_fn(state.params)
  accuracy = jnp.mean(jnp.argmax(logits, -1) == labels)
  return grads, loss, accuracy

@jax.jit
def update_model(state, grads):
  return state.apply_gradients(grads=grads)

(21 % 20 == 0)

def train_epoch(state, train_ds, batch_size, rng, loss_history, test_loss_history,
    test_ds):
  """Train for a single epoch."""
  train_ds_size = len(train_ds['image'])
  steps_per_epoch = train_ds_size // batch_size

  perms = jax.random.permutation(rng, len(train_ds['image']))
  perms = perms[:steps_per_epoch * batch_size]  # skip incomplete batch
  perms = perms.reshape((steps_per_epoch, batch_size))

  epoch_loss = []
  epoch_accuracy = []
  print("perms:", len(perms))
  i = 0


  test_images = test_ds['image']
  test_labels = test_ds['label']
  train_images = train_ds['image']
  train_labels = train_ds['label']

  for perm in perms:
    if (i % 12 == 0):
        print("iteration", i, "out of", len(perms))
        grads, loss, accuracy = apply_model(state, test_images, test_labels)
        test_loss_history.append(loss)
        grads, loss, accuracy = apply_model(state, train_images, train_labels)
        loss_history.append(loss)

    i += 1
    batch_images = train_ds['image'][perm, ...]
    batch_labels = train_ds['label'][perm, ...]

    grads, loss, accuracy = apply_model(state, batch_images, batch_labels)
    state = update_model(state, grads)
    epoch_loss.append(loss)
    epoch_accuracy.append(accuracy)

  train_loss = np.mean(epoch_loss)
  train_accuracy = np.mean(epoch_accuracy)
  return state, train_loss, train_accuracy

def get_datasets():
  """Load MNIST train and test datasets into memory."""
  ds_builder = tfds.builder('mnist')
  ds_builder.download_and_prepare()
  train_ds = tfds.as_numpy(ds_builder.as_dataset(split='train', batch_size=-1))
  test_ds = tfds.as_numpy(ds_builder.as_dataset(split='test', batch_size=-1))
  train_ds['image'] = jnp.float32(train_ds['image']) / 255.
  test_ds['image'] = jnp.float32(test_ds['image']) / 255.
  return train_ds, test_ds

def create_train_state(rng, config):
  """Creates initial `TrainState`."""
  cnn = CNN()
  params = cnn.init(rng, jnp.ones([1, 28, 28, 1]))['params']

```

```
  tx = config.optimiser

  return train_state.TrainState.create(
      apply_fn=cnn.apply, params=params, tx=tx)

def train_and_evaluate(config: ml_collections.ConfigDict,
                        workdir: str,
                        train_ds,
                        test_ds,
                        seed):

  rng, init_rng = jax.random.split(seed)
  state = create_train_state(init_rng, config)

  _, test_loss, test_accuracy = apply_model(state, test_ds['image'], test_ds['label'])
  # print('epoch:% 3d, test_loss: %.4f, test_accuracy: %.2f'
  #         % (0, test_loss, test_accuracy * 100))


  loss_history = []
  test_loss_history = []

  for epoch in range(1, config.num_epochs + 1):
    rng, input_rng = jax.random.split(rng)
    state, train_loss, train_accuracy = train_epoch(state, train_ds, config.batch_size, input_rng, loss_history, test_loss_history, test_ds)
    _, test_loss, test_accuracy = apply_model(state, test_ds['image'], test_ds['label'])

    print('epoch:% 3d, train_loss: %.4f, train_accuracy: %.2f, test_loss: %.4f, test_accuracy: %.2f'
          % (epoch, train_loss, train_accuracy * 100, test_loss, test_accuracy * 100))
  return state, loss_history, test_loss_history

def get_config(opt, batch_size):
  """Get the default hyperparameter configuration."""
  config = ml_collections.ConfigDict()
  config.optimiser = opt
  config.batch_size = batch_size
  config.num_epochs = 1
  return config

train_ds, test_ds = get_datasets()

print(train_ds.keys())
print(train_ds['image'].shape)
print(train_ds['label'].shape)
print(test_ds['label'].shape)

def f(learning_rate, b1, b2, batch_size):
    opt = optax.adam(learning_rate=learning_rate, b1=b1, b2=b2)
    cfg = get_config(opt=opt, batch_size=round(batch_size))
    _, _, test_loss = train_and_evaluate(cfg, "./mnist/", train_ds, test_ds)
    return test_loss

def f2(learning_rate, batch_size):
    opt = optax.sgd(learning_rate=learning_rate)
    cfg = get_config(opt=opt, batch_size=round(batch_size))
    _, _, test_loss = train_and_evaluate(cfg, "./mnist/", train_ds, test_ds)
    return test_loss

def global_random_search(intervals, N, f):
    lowest = None
    l = [l for l, u in intervals]
    u = [u for l, u in intervals]

```

```
    for s in range(N):
        r = np.random.uniform(l, u)
        print("iteration:", s, "trying out:", r)
        v = f(*r)
        if (not lowest) or lowest[0] > v:
            lowest = (v.copy(), r.copy())
    return lowest

v = global_random_search([(0.001, 0.1), (0.5,0.99), (0.5,0.99), (1, 128)], 20, f)

v

  learning_rate = 0.0015
  beta1 = 0.898
  beta2 = 0.9575
  batch_size = 98

v2 = global_random_search([(0.4, 0.8), (40, 90)], 20, f2)

print(v2)

# (array(0.05145077, dtype=float32), array([ 0.49315356, 58.39919518]))
 # (array(0.05257225, dtype=float32), array([ 0.75313327, 93.05358694]))
# (array(0.04633828, dtype=float32), array([ 0.48917857, 48.61637121]))
learning_rate = 0.489
batch_size = 49

# opt = optax.sgd(learning_rate=0.1)
opt = optax.adam(learning_rate=0.001, b1=0.9, b2=0.999)
cfg = get_config(opt=opt, batch_size=128)
# state, loss_history, test_loss = train_and_evaluate(cfg, "./mnist/", train_ds, test_ds)

print(len(loss_history))
print(len(train_ds['label'])/128)

plt.plot(range(len(loss_history)), loss_history)

def sgdf(learning_rate, batch_size, seed):
    opt = optax.sgd(learning_rate=learning_rate)
    cfg = get_config(opt=opt, batch_size=round(batch_size))
    _, loss_history, test_loss_history = train_and_evaluate(cfg, "./mnist/", train_ds, test_ds, seed)
    return loss_history, test_loss_history

def adamf(learning_rate, b1, b2, batch_size, seed):
    opt = optax.adam(learning_rate=learning_rate, b1=b1, b2=b2)
    cfg = get_config(opt=opt, batch_size=round(batch_size))
    _, loss_history, test_loss_history = train_and_evaluate(cfg, "./mnist/", train_ds, test_ds, seed)
    return loss_history, test_loss_history

def run_multiple(runs, f):
    # need to thread random seed

    loss_histories = []
    test_losses = []

    seed = jax.random.PRNGKey(0)
    seed, subseed = jax.random.split(seed)

    for r in range(runs):
        print("Run number:", r)
        loss_history, test_loss_history = f(subseed)
        seed, subseed = jax.random.split(seed)
        loss_histories += [loss_history]
        test_losses += [test_loss_history]
    return loss_histories, test_losses
```

```
sgd_default_alpha = 0.1
sgd_default_batch = 128
sgd_default = lambda seed: sgdf(sgd_default_alpha, sgd_default_batch,seed=seed)

sgd_optimal_alpha = 0.489
sgd_optimal_batch = 49
sgd_optimal = lambda seed: sgdf(sgd_optimal_alpha, sgd_optimal_batch, seed=seed)

adam_default_alpha = 0.01
adam_default_b1 = 0.9
adam_default_b2 = 0.999
adam_default_batch = 128
adam_default = lambda seed: adamf(adam_default_alpha, adam_default_b1,
    adam_default_b2, adam_default_batch, seed=seed)

adam_optimal_alpha = 0.0015
adam_optimal_b1 = 0.898
adam_optimal_b2 = 0.9575
adam_optimal_batch = 98
adam_optimal = lambda seed: adamf(adam_optimal_alpha, adam_optimal_b1,
    adam_optimal_b2, adam_optimal_batch, seed=seed)

rng = jax.random.PRNGKey(0)

sgd_train_loss, sgd_test_loss = sgd_optimal(rng)

adam_train_loss, adam_test_loss = adam_optimal(rng)

sgd_train_loss

sgd_test_loss

r1 = np.array(sgd_train_loss)
r1.shape[0]

compare_tt(sgd_train_loss, sgd_test_loss, "SGD Optimal", "Train", "Test")

compare_tt(adam_train_loss, adam_test_loss, "Adam Optimal", "Train", "Test")

runs = 2
sgd_default_loss_histories, sgd_default_test_losses = run_multiple(runs, sgd_default)

print(sgd_default_test_losses)

runs = 20
print("SGD Default")
sgd_default_loss_histories, sgd_default_test_losses = run_multiple(runs, sgd_default)

print("SGD Optimal")
sgd_optimal_loss_histories, sgd_optimal_test_losses = run_multiple(runs, sgd_optimal)

print("Adam Default")
adam_default_loss_histories, adam_default_test_losses = run_multiple(runs,
    adam_default)

print("Adam Optimal")
adam_optimal_loss_histories, adam_optimal_test_losses = run_multiple(runs,
    adam_optimal)

import pickle

mlruns = {
    "sgd_default_loss_histories": sgd_default_loss_histories,
    "sgd_default_test_losses": sgd_default_test_losses,
    "sgd_optimal_loss_histories": sgd_optimal_loss_histories,
    "sgd_optimal_test_losses": sgd_optimal_test_losses,

```

```
    "adam_default_loss_histories": adam_default_loss_histories,
    "adam_default_test_losses": adam_default_test_losses,
    "adam_optimal_loss_histories": adam_optimal_loss_histories,
    "adam_optimal_test_losses": adam_optimal_test_losses
}

pickle.dump(mlruns, open("mlruns.p", "wb"))

import pickle
mlruns_l = pickle.load(open( "mlruns.p", "rb" ))

mlruns_l.keys()

def plot_history(losses):
    'losses :: [[float]], ith element is loss vs iteration of ith run of the SGD'
    losses = np.array(losses)
    average_on_iter_i = np.mean(losses, axis=0)
    min_on_iter_i = np.minimum.reduce(losses)
    max_on_iter_i = np.maximum.reduce(losses)
    x = range(len(average_on_iter_i))
    plt.plot(x, average_on_iter_i , 'k-')
    plt.fill_between(x, min_on_iter_i, max_on_iter_i)

def avg_max_min(loss_histories):
    average_on_iter_i = np.mean(loss_histories, axis=0)
    min_on_iter_i = np.minimum.reduce(loss_histories)
    max_on_iter_i = np.maximum.reduce(loss_histories)
    return average_on_iter_i, min_on_iter_i, max_on_iter_i

plot_history(mlruns_l['sgd_default_loss_histories'])

plot_history(mlruns_l['sgd_optimal_loss_histories'])

plot_history(mlruns_l['adam_default_loss_histories'])

plot_history(mlruns_l['adam_optimal_loss_histories'])

np.array(mlruns_l['sgd_default_loss_histories']).shape

np.array(mlruns_l['sgd_optimal_loss_histories']).shape

def compare_sgd(r1, r2, title="Title", r1l="r1", r2l="r2"):
    r1 = np.array(r1) ; r2 = np.array(r2)
    xr = r1.shape[1]  ; xr2 = r2.shape[1]
    x1 = np.linspace(0, 100, xr)
    x2 = np.linspace(0, 100, xr2)
    a1, l1, h1 = avg_max_min(r1)
    a2, l2, h2 = avg_max_min(r2)

    plt.semilogy(x1, a1, color='#2e6fd9bb', label=r1l)
    plt.fill_between(x1, l1, h1, color="#3d84f588")
    xlim = plt.xlim()
    ylim = plt.ylim()
    plt.semilogy(x2, a2, color='#ff9c24bb', label=r2l)
    plt.fill_between(x2, l2, h2, color='#ffc37088')
    plt.xlim(xlim)
    plt.ylim(ylim)
    plt.title(title)
    plt.legend()

    plt.xlabel(r'%epoch')
    plt.ylabel(r'loss')
    # plt.title("default vs optimal")

r1 = np.array(mlruns_l['sgd_default_loss_histories'])
r2 = np.array(mlruns_l['sgd_optimal_loss_histories'])

compare_sgd(r2, r1, title="Constant: Default vs. Optimal", r1l="Optimal", r2l="
```

```
    Default")

r1 = np.array(mlruns_l['adam_default_loss_histories'])
r2 = np.array(mlruns_l['adam_optimal_loss_histories'])

compare_sgd(r1=r2, r2=r1, title="Adam: Default vs. Optimal", r1l="Optimal", r2l="Default")

r1 = np.array(mlruns_l['sgd_optimal_loss_histories'])
r2 = np.array(mlruns_l['adam_optimal_loss_histories'])
compare_sgd(r1=r1, r2=r2, title="Optimal Constant vs. Optimal Adam", r1l="Constant", r2l="Adam")

r1 = np.array(mlruns_l['sgd_default_loss_histories'])
r2 = np.array(mlruns_l['adam_default_loss_histories'])
compare_sgd(r1=r2, r2=r1, title="Default Constant vs. Default Adam", r1l="Adam", r2l="Constant")

def compare_tt(F, F2, title="Title", Fl="r1", F2l="r2"):

    r1 = np.array(F) ; r2 = np.array(F2)
    xr = r1.shape[0]  ; xr2 = r2.shape[0]

    x1 = np.linspace(0, 100, xr)
    x2 = np.linspace(0, 100, xr2)

    plt.semilogy(x1, F, color='#2e6fd9bb', label=Fl)

    xlim = plt.xlim()
    ylim = plt.ylim()
    plt.semilogy(x2, F2, color='#ff9c24bb', label=F2l)

    plt.xlim(xlim)
    plt.ylim(ylim)

    plt.title(title)
    plt.legend()

    plt.xlabel(r'%run')
    plt.ylabel(r'loss')

print(mlruns_l['sgd_optimal_test_losses'][:10])
print(np.array(mlruns_l['sgd_optimal_test_losses']).shape)
print(np.array(mlruns_l['sgd_optimal_loss_histories']).shape)
print(mlruns_l['sgd_optimal_test_losses'])
print([x[-1] for x in mlruns_l['sgd_optimal_loss_histories']])



"""Trains an SST2 text classifier."""
from typing import Any, Callable, Dict, Iterable, Optional, Sequence, Tuple, Union

from absl import logging
from flax import struct
from flax.metrics import tensorboard
from flax.training import train_state
import jax
import jax.numpy as jnp
import ml_collections
import numpy as np
import optax
import tensorflow as tf

import input_pipeline
import models


Array = jnp.ndarray
```

```python
Example = Dict[str, Array]
TrainState = train_state.TrainState


class Metrics(struct.PyTreeNode):
  """Computed metrics."""
  loss: float
  accuracy: float
  count: Optional[int] = None


@jax.vmap
def sigmoid_cross_entropy_with_logits(*, labels: Array, logits: Array) -> Array:
  """Sigmoid cross entropy loss."""
  zeros = jnp.zeros_like(logits, dtype=logits.dtype)
  condition = (logits >= zeros)
  relu_logits = jnp.where(condition, logits, zeros)
  neg_abs_logits = jnp.where(condition, -logits, logits)
  return relu_logits - logits * labels + jnp.log1p(jnp.exp(neg_abs_logits))


def get_initial_params(rng, model):
  """Returns randomly initialized parameters."""
  token_ids = jnp.ones((2, 3), jnp.int32)
  lengths = jnp.ones((2,), dtype=jnp.int32)
  variables = model.init(rng, token_ids, lengths, deterministic=True)
  return variables['params']


def create_train_state(rng, config: ml_collections.ConfigDict, model):
  """Create initial training state."""
  params = get_initial_params(rng, model)
  tx = optax.chain(
      optax.sgd(learning_rate=config.learning_rate, momentum=config.momentum),
      optax.additive_weight_decay(weight_decay=config.weight_decay))
  state = TrainState.create(apply_fn=model.apply, params=params, tx=tx)
  return state


def compute_metrics(*, labels: Array, logits: Array) -> Metrics:
  """Computes the metrics, summed across the batch if a batch is provided."""
  if labels.ndim == 1:  # Prevent the labels from broadcasting over the logits.
    labels = jnp.expand_dims(labels, axis=1)
  loss = sigmoid_cross_entropy_with_logits(labels=labels, logits=logits)
  binary_predictions = (logits >= 0.)
  binary_accuracy = jnp.equal(binary_predictions, labels)
  return Metrics(
      loss=jnp.sum(loss),
      accuracy=jnp.sum(binary_accuracy),
      count=logits.shape[0])


def model_from_config(config: ml_collections.ConfigDict):
  """Builds a text classification model from a config."""
  model = models.TextClassifier(
      embedding_size=config.embedding_size,
      hidden_size=config.hidden_size,
      vocab_size=config.vocab_size,
      output_size=config.output_size,
      dropout_rate=config.dropout_rate,
      word_dropout_rate=config.word_dropout_rate,
      unk_idx=config.unk_idx)
  return model


def train_step(
    state: TrainState,
    batch: Dict[str, Array],
```

```
    rngs: Dict[str, Any],
) -> Tuple[TrainState, Metrics]:
  """Train for a single step."""
  # Make sure to get a new RNG at every step.
  step = state.step
  rngs = {name: jax.random.fold_in(rng, step) for name, rng in rngs.items()}

  def loss_fn(params):
    variables = {'params': params}
    logits = state.apply_fn(
        variables, batch['token_ids'], batch['length'],
        deterministic=False,
        rngs=rngs)

    labels = batch['label']
    if labels.ndim == 1:
      labels = jnp.expand_dims(labels, 1)
    loss = jnp.mean(
        sigmoid_cross_entropy_with_logits(labels=labels, logits=logits))
    return loss, logits

  grad_fn = jax.value_and_grad(loss_fn, has_aux=True)
  value, grads = grad_fn(state.params)
  (_, logits) = value

  new_state = state.apply_gradients(grads=grads)
  metrics = compute_metrics(labels=batch['label'], logits=logits)
  return new_state, metrics


def eval_step(state: TrainState, batch: Dict[str, Array],
              rngs: Dict[str, Any]) -> Metrics:
  """Evaluate for a single step. Model should be in deterministic mode."""
  variables = {'params': state.params}
  logits = state.apply_fn(
      variables, batch['token_ids'], batch['length'],
      deterministic=True,
      rngs=rngs)
  metrics = compute_metrics(labels=batch['label'], logits=logits)
  return metrics


def normalize_batch_metrics(
        batch_metrics: Sequence[Metrics]) -> Metrics:
  """Consolidates and normalizes a list of per-batch metrics dicts."""
  # Here we sum the metrics that were already summed per batch.
  total_loss = np.sum([metrics.loss for metrics in batch_metrics])
  total_accuracy = np.sum([metrics.accuracy for metrics in batch_metrics])
  total = np.sum([metrics.count for metrics in batch_metrics])
  # Divide each metric by the total number of items in the data set.
  return Metrics(
      loss=total_loss.item() / total, accuracy=total_accuracy.item() / total)


def batch_to_numpy(batch: Dict[str, tf.Tensor]) -> Dict[str, Array]:
  """Converts a batch with TF tensors to a batch of NumPy arrays."""
  # _numpy() reuses memory, does not make a copy.
  # pylint: disable=protected-access
  return jax.tree_map(lambda x: x._numpy(), batch)


def evaluate_model(
        eval_step_fn: Callable[..., Any],
        state: TrainState,
        batches: Union[Iterable[Example], tf.data.Dataset],
        epoch: int,
        rngs: Optional[Dict[str, Any]] = None
) -> Metrics:
```

```python
  """Evaluate a model on a dataset."""
  batch_metrics = []
  for i, batch in enumerate(batches):
    batch = batch_to_numpy(batch)
    if rngs is not None:  # New RNG for each step.
      rngs = {name: jax.random.fold_in(rng, i) for name, rng in rngs.items()}

    metrics = eval_step_fn(state, batch, rngs)
    batch_metrics.append(metrics)

  batch_metrics = jax.device_get(batch_metrics)
  metrics = normalize_batch_metrics(batch_metrics)
  logging.info('eval  epoch %03d loss %.4f accuracy %.2f', epoch,
               metrics.loss, metrics.accuracy * 100)
  return metrics


def train_epoch(train_step_fn: Callable[..., Tuple[TrainState, Metrics]],
                state: TrainState,
                train_batches: tf.data.Dataset,
                epoch: int,
                rngs: Optional[Dict[str, Any]] = None
                ) -> Tuple[TrainState, Metrics]:
  """Train for a single epoch."""
  batch_metrics = []
  for batch in train_batches:
    batch = batch_to_numpy(batch)
    state, metrics = train_step_fn(state, batch, rngs)
    batch_metrics.append(metrics)

  # Compute the metrics for this epoch.
  batch_metrics = jax.device_get(batch_metrics)
  metrics = normalize_batch_metrics(batch_metrics)

  logging.info('train epoch %03d loss %.4f accuracy %.2f', epoch,
               metrics.loss, metrics.accuracy * 100)

  return state, metrics


def train_and_evaluate(config: ml_collections.ConfigDict,
                       workdir: str) -> TrainState:
  """Execute model training and evaluation loop.
  Args:
    config: Hyperparameter configuration for training and evaluation.
    workdir: Directory where the tensorboard summaries are written to.
  Returns:
    The final train state that includes the trained parameters.
  """
  # Prepare datasets.
  train_dataset = input_pipeline.TextDataset(
      tfds_name='glue/sst2', split='train')
  eval_dataset = input_pipeline.TextDataset(
      tfds_name='glue/sst2', split='validation')
  train_batches = train_dataset.get_bucketed_batches(
      config.batch_size,
      config.bucket_size,
      max_input_length=config.max_input_length,
      drop_remainder=True,
      shuffle=True,
      shuffle_seed=config.seed)
  eval_batches = eval_dataset.get_batches(batch_size=config.batch_size)

  # Keep track of vocab size in the config so that the embedder knows it.
  config.vocab_size = len(train_dataset.vocab)

  # Compile step functions.
  train_step_fn = jax.jit(train_step)
```

```
  eval_step_fn = jax.jit(eval_step)

  # Create model and a state that contains the parameters.
  rng = jax.random.PRNGKey(config.seed)
  model = model_from_config(config)
  state = create_train_state(rng, config, model)

  summary_writer = tensorboard.SummaryWriter(workdir)
  summary_writer.hparams(dict(config))

  # Main training loop.
  logging.info('Starting training...')
  for epoch in range(1, config.num_epochs + 1):

    # Train for one epoch.
    rng, epoch_rng = jax.random.split(rng)
    rngs = {'dropout': epoch_rng}
    state, train_metrics = train_epoch(
        train_step_fn, state, train_batches, epoch, rngs)

    # Evaluate current model on the validation data.
    eval_metrics = evaluate_model(eval_step_fn, state, eval_batches, epoch)

    # Write metrics to TensorBoard.
    summary_writer.scalar('train_loss', train_metrics.loss, epoch)
    summary_writer.scalar(
        'train_accuracy',
        train_metrics.accuracy * 100,
        epoch)
    summary_writer.scalar('eval_loss', eval_metrics.loss, epoch)
    summary_writer.scalar(
        'eval_accuracy',
        eval_metrics.accuracy * 100,
        epoch)

  summary_writer.flush()
  return state

    x = nn.Conv(features=32, kernel_size=(3, 3))(x)
    x = nn.relu(x)
    x = nn.avg_pool(x, window_shape=(2, 2), strides=(2, 2))
    x = nn.Conv(features=64, kernel_size=(3, 3))(x)
    x = nn.relu(x)
    x = nn.avg_pool(x, window_shape=(2, 2), strides=(2, 2))
    x = x.reshape((x.shape[0], -1))  # flatten
    x = nn.Dense(features=256)(x)
    x = nn.relu(x)
    x = nn.Dense(features=10)(x)
```

```
# -*- coding: utf-8 -*-
"""Text Analytics - Twitter Network - Stochastic Gradient Descent - Jax.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1OJ0bGhZzah8hBwv8cYwnbaENeagw0Xgz
"""

import jax.numpy as jnp
import numpy as np
from jax import grad, jit, vmap, pmap, random

"""# Datatypes and Naming Conventions

### Type Synonyms
"""

from typing import Any
from typing import FrozenSet
```

```
from typing import List
import numpy.typing as npt

UserId = str
UserIdSet = FrozenSet[UserId]

Corpus = str
Token = str
Tokens = List[Token]
UniqueTokens = FrozenSet[Token]

Edge = Any
Edges = FrozenSet[Edge]

Graph = Any
EdgeWeight = int

Vec = any
VecInt = npt.NDArray[np.int_]
WordVec = VecInt

TwitterData = Any
TwitterDataP = Any

TokenCount = Any


UserCorpusMap = Any
UserTokensMap = Any
UserTokenCountMap = Any
UserWordVecMap = Any
Set = Any
IndexMap = Any
WordVecIndex   = IndexMap
EdgeVecIndex = IndexMap

"""```
TwitterDataP =
            [
              {
                user: UserId,
                corpus: Tokens,
                followers: [ UserId ],
                followings: [ UserId ]
              }
            ]

Edge = frozenset(UserId, UserId)
Graph = Map<Edge, EdgeWeight>

TokenCount = Map<Token, Int>

UserCorpusMap = Map<UserId, Corpus>
UserTokensMap = Map<UserId, Tokens>
UserTokenCountMap = Map<UserId, TokenCount>
UserWordVecMap = Map<UserId, WordVec>

WordVecIndex = Map<Token, Integer>
EdgeVecIndex = Map<Edge, Integer>

IndexMap = Map<T, Integer>
```

Functions

User Extraction from Data
"""

```
```

```python
def createUserTokensMap(data : TwitterDataP) -> UserTokensMap:
  userTokensMap = {}
  for user in data:
    userTokensMap[user['user']] = user['corpus']
  return userTokensMap

def getUserIdSet(userCorpusMap : UserCorpusMap) -> UserIdSet:
  return frozenset(userCorpusMap.keys())

"""## Graph Construction"""

from itertools import combinations

def mkEdge(u1 : UserId, u2 : UserId) -> Edge:
  return frozenset({u1,u2})

def createEdges(userIdSet : UserIdSet) -> Edges:
  edges = set({})
  for (a,b) in combinations(userIdSet, 2):
    edges.add(mkEdge(a, b))
  return edges

def graphSet(edges : Edges, val : int) -> Graph:
  graph = {}
  for e in edges:
    graph[e] = val
  return graph

def graphZeroed(edges : Edges) -> Graph:
  return graphSet(edges, 0)

def graphMergeGroundTruth(graph: Graph, data : TwitterDataP) -> Graph:
  mergedGraph = graph.copy()
  for d in data:
    followed = set(d['followers'])
    following = set(d['followings'])
    u1 = d['user']

    for u2 in followed:
      mergedGraph[mkEdge(u1, u2)] = 0.5
    for u2 in following:
      mergedGraph[mkEdge(u1, u2)] = 0.5

    twoWay = followed.intersection(following)
    for u2 in twoWay:
      mergedGraph[mkEdge(u1, u2)] = 1.0

  return mergedGraph

def graphCos(edges: Edges, userWordVecMap : UserWordVecMap) -> Graph:
  graph = {}
  for u1, u2 in edges:
    v = cos(userWordVecMap[u1], userWordVecMap[u2])
    graph[mkEdge(u1,u2)] = v
  return graph

def createEdgeVecIndex(edges: Edges) -> EdgeVecIndex:
  return createWordVecIndex(edges)

def graphEdgeVec(graph : Graph, edgeVecIndex : EdgeVecIndex):
  return tokenCountWordVec(graph, edgeVecIndex)

"""## Corpus Processing and Transformations"""

from nltk import word_tokenize

def countTokens(tokens : Tokens) -> TokenCount:
  tokenCount = {}
```

```python
  for token in tokens:
    tokenCount[token] = tokenCount.get(token, 0) + 1
  return tokenCount

def combineTokensList(tokens : List[Tokens]) -> Tokens:
  return sum(tokens, [])

def tokenise(corpus : Corpus) -> Tokens:
  return word_tokenize(corpus)

def leastOccuringTokens(countUpperBound : int, tokenCount : TokenCount) -> Tokens:
  leastOccuring = []
  for token, count in tokenCount.items():
    if count <= countUpperBound:
      leastOccuring.append(token)
  return leastOccuring

def excludeTokens(excludedTokens : Tokens, tokenCount : TokenCount) -> TokenCount:
  excludedCount = tokenCount.copy()
  for token in excludedTokens:
    if token in excludedCount:
      del excludedCount[token]
  return excludedCount

"""## WordVector

"""

def createWordVecIndex(allUniqueTokens : UniqueTokens) -> WordVecIndex:
  wvIndex = {}
  for wordIndex, word in enumerate(allUniqueTokens):
    wvIndex[word] = wordIndex
  return wvIndex

def createUserWordVecMap(userTokenCountMap : UserTokenCountMap, wvIndex :
    WordVecIndex) -> UserWordVecMap:
    userWvMap = userTokenCountMap.copy()
    for userId, tokenCount in userTokenCountMap.items():
        userWvMap[userId] = tokenCountWordVec(tokenCount, wvIndex)
    return userWvMap

def tokenCountWordVec(tokenCount : TokenCount, wvIndex : WordVecIndex) -> WordVec:
    wordVec = np.zeros(len(wvIndex))
    for token, count in tokenCount.items():
      if token in wvIndex:
          wordIndex = wvIndex[token]
          wordVec[wordIndex] = count
    return wordVec

"""## Similarity Measures"""

@jit
def cos(v1 : Vec, v2 : Vec) -> float:
  return jnp.sum(v1 * v2) / (jnp.sqrt(jnp.sum(v1**2)) * (jnp.sqrt(jnp.sum(v2**2))) +
    0.000001)

@jit
def cos2(v : Vec) -> float:
  #print(v.shape)
  #print(v)
  v1 = v[0]
  v2 = v[1]
  r = jnp.sum(v1 * v2) / (jnp.sqrt(jnp.sum(v1**2)) * (jnp.sqrt(jnp.sum(v2**2))) +
    0.000001)
  #print(r)
  return r

def npcos2(v : Vec) -> float:
```

```
  #print(v.shape)
  #print(v)
  v1 = v[0]
  v2 = v[1]
  r = np.sum(v1 * v2) / (np.sqrt(np.sum(v1**2)) * (np.sqrt(np.sum(v2**2))) +
    0.000001)
  #print(r)
  return r

def euc2(v1 : Vec, v2 : Vec) -> float:
  return jnp.sqrt(jnp.sum((v1 - v2)**2))

npcos2(np.array([[1, 1, 0],[0, 1 ,0]]))

"""# Loading Data & Processing Data

## Loading and Counting

from google.colab import drive
drive.mount('/content/drive')
"""

from google.colab import drive
drive.mount('/content/drive')

import json
with open('./drive/MyDrive/TA/data_pre.json', 'r') as f:
  dataset = json.load(f)['dataset']
dataset[0].keys()

userTokensMap = createUserTokensMap(dataset)
print("Users in dataset: \t", len(userTokensMap))

combinedTokens = combineTokensList(list(userTokensMap.values()))
print("Total token count: \t",len(combinedTokens))

totalTokenCount = countTokens(combinedTokens)
print("Unique tokens: \t \t", len(totalTokenCount.keys()))

mapToTuples = lambda x: [(k, v) for k, v in x.items()]
sortByVal   = lambda x: sorted(mapToTuples(x), reverse=True, key=lambda x: x[1])

print("Top 10 tokens: \t \t", sortByVal(totalTokenCount)[:10])
print("Bottom 10 tokens: \t", sortByVal(totalTokenCount)[::-1][:10])

"""## Pruning Least Occuring Tokens"""

leastOccuring3 = leastOccuringTokens(3, totalTokenCount)
print("Unique tokens that occur 3 or less times: \t", len(leastOccuring3))
print(leastOccuring3[:5])

excludedCount3 = excludeTokens(leastOccuring3, totalTokenCount)
print("Unique tokens that occur more than 3 times: \t", len(excludedCount3))
print(mapToTuples(excludedCount3)[:5])

userTokenCountMap = userTokensMap.copy()
for user, tokens in userTokenCountMap.items():
  userTokenCountMap[user] = excludeTokens(leastOccuring3, countTokens(tokens))

mapToTuples(userTokenCountMap)[0]

"""# Constructing Graphs

### Ground Truth Graph
"""

userIdSet = getUserIdSet(userTokenCountMap)
edges = createEdges(userIdSet)
```

```
graphZero = graphZeroed(edges)
graphGroundTruth = graphMergeGroundTruth(graphZero, dataset)

print("Total connections strength: \t", sum(graphGroundTruth.values()))
print("Number of edges: \t \t",len(graphGroundTruth.keys()))

"""WordVec Cosine Similarity Graph"""

wvIndex = createWordVecIndex(excludedCount3.keys())
userWordVecMap = createUserWordVecMap(userTokenCountMap, wvIndex)

print("Unique tokens a.k.a Vector length: \t", len(wvIndex))
print("Number of users: \t \t \t", len(userWordVecMap))
print("User wv length: \t \t \t", len(list(userWordVecMap.values())[0]))

graphCosine = graphCos(edges, userWordVecMap)

mapToTuples(graphCosine)[:10]

"""# Analyising Data

We have 2 datasets:
- Dataset 1 : Retrieved by traversing followers on twitter.
- Dataset 2 : Retrived by querying a topic and gathering the users.

We have already pre-processed dataset, now preprocess dataset 2.
We also want to learn about each of the datasets.
We can first investigate the links:
- Looking at total number of users in each dataset.
- Counting the percentage of edges which are 0.5 strength on the ground truth graph.
- Counting the percentage of edges which are 1 strength on the ground truth graph.

We can also investigate the text content of the whole corpus:
- How many unique tokens are there in each dataset?
- What is the mean and variance of the number of unique tokens for a single user in each dataset?
- What is the distribution of the tokens for the each dataset?

Yash has plotted the plot of token occurance for dataset 1, do the same for dataset 2.

### Links in Each Dataset
"""

numberOfUsersDataset1 = len(userTokenCountMap)
print(numberOfUsersDataset1)
# repeat for dataset 2

print(mapToTuples(graphGroundTruth)[:5])

def countValOccurance(vals, val):
  count = 0
  for v in vals:
    if val == v:
      count += 1
  return count

totalEdges = len(graphGroundTruth)
print(totalEdges)
halfLinkCount = countValOccurance(graphGroundTruth.values(), 0.5)
fullLinkCount = countValOccurance(graphGroundTruth.values(), 1)
print(halfLinkCount/totalEdges)
print(fullLinkCount/totalEdges)

# repeat for dataset 2

"""### Text Content of Dataset"""

```

```
# can use userTokenCountMap and excludedCount3/
print(mapToTuples(userTokenCountMap)[0]) # is a map from user to token count for that user
# excludedCount3 has all the tokens counted, except maybe do it for full token list (see the code above that constructs this variable)



"""# Comparing Graphs with Similarity Measures"""

edgeVecIndex = createEdgeVecIndex(edges)
cosineEv = graphEdgeVec(graphCosine, edgeVecIndex)
groundTruthEv = graphEdgeVec(graphGroundTruth, edgeVecIndex)
zeroedEv = graphEdgeVec(graphZero, edgeVecIndex)
meanEv = graphEdgeVec(graphSet(edges, np.mean(groundTruthEv)), edgeVecIndex)

print("Ground Truth vs Cosine Similarity")
print("Cosine Similarity: \t", cos(cosineEv, groundTruthEv))
print("Euclidean Distance: \t", euc2(cosineEv, groundTruthEv))

print("\nIdenities")
print("Cosine Similarity: \t", cos(cosineEv, cosineEv))
print("Euclidean Distance: \t", euc2(cosineEv, cosineEv))

print("\nZeroed")
print("Cosine Similarity: \t", cos(groundTruthEv, zeroedEv))
print("Euclidean Distance: \t", euc2(groundTruthEv, zeroedEv))

print("\nMean")
print("Cosine Similarity: \t", cos(groundTruthEv, meanEv))
print("Euclidean Distance: \t", euc2(groundTruthEv, meanEv))

"""# Linear Regression

2 sets of indices on data.
$m$ = Subset of $M$
$k$ = subset of $K$

 $  $ = Set of all edges


```
edges = [(u1, u2), (u1, u3), ...]
```


K = Set of all word vector indices

N_{mk} = Edges and associated word vector subset similarity.

x = weights on wordvector indices

N_{mk} =

```
[
  (u1, u2) = cosine(u1_kx, u2_kx)
  (u1, u3) = cosine(u1_kx, u3_kx)
  .
  .
  .
]
u1_kx = [ (k_i * x_i * u1_i) .... ]
```
if i in set k then $k_i=1$ otherwise $k_i = 0$

Can try running this 2-level batching, but in this case not sure if going to be
```

```
    necessary, but actually maybe for ngrams and stuff.

Exponential nature of edges.
Exponential nature of ngrams.

### Preparing the Data
"""

def constructSetIndex(setToIndex : Set) -> IndexMap:
  return createWordVecIndex(setToIndex)

def userIdxToWvVector(userWordVecMap, userIndexMap):
  wordVecLength = len(list(userWordVecMap.values())[0])
  dataWV = np.zeros((len(userIndexMap), wordVecLength))
  for user, wv in userWordVecMap.items():
    idx = userIndexMap[user]
    dataWV[idx] = wv
  return dataWV

def constructM(userIndexMap, edgeIndexMap):
  dataM = np.zeros((len(edgeIndexMap), 2))
  for (u1, u2), idx in edgeIndexMap.items():
    dataM[idx] = np.array([ np.array(userIndexMap[u1]), np.array(userIndexMap[u2]) ])
  return dataM

edgeIndexMap = constructSetIndex(edges)
userIndexMap = constructSetIndex(list(userWordVecMap.keys()))

npGT = graphEdgeVec(graphGroundTruth, edgeIndexMap)
npWV = userIdxToWvVector(userWordVecMap, userIndexMap).astype(int)
npM = constructM(userIndexMap, edgeIndexMap).astype(int)

npR = npWV[npM]

dataGT = jnp.array(npGT)
dataWV = jnp.array(npWV)
dataM = jnp.array(npM)
dataR = jnp.array(npR)

print("dataM", dataM.shape, dataM[:4])
print("dataWV", dataWV.shape, dataWV[:4])
print("dataGT", dataGT.shape, dataGT[:4])
print("dataR", dataR.shape, dataR[:4])

a = np.array([
  [[1,2,3],[4,5,6]],
  [[7,8,9],[10,11,12]]
 ])
b = np.array([2, 0, 1])
print(a.shape)
a*b

"""### Indexing and Batching the Data"""

K = jnp.arange(len(wvIndex))
M = jnp.arange(len(edgeVecIndex))

def indices(key, bm, bk):
  m = random.choice(key, M, [bm])
  k = random.choice(key, K, [bk])
  return m, k

"""### Loss Function"""

@jit
def loss(x, m, k):
  a1 = dataM.at[m].get()
  a2 = dataWV.at[a1].get()
```

```
  a3 = a2.T.at[k].get().T
  x1 = x.at[k].get()

  a4 = a3 * x1
  a5 = vcos2(a4)
  return cos(a5, dataGT[m])

@jit
def loss_mse(x, m, k):
  a1 = dataM.at[m].get()
  a2 = dataWV.at[a1].get()

  a3 = a2.T.at[k].get().T
  x1 = x.at[k].get()

  a4 = a3 * x1
  a5 = vmse(a4)
  return mse(a5, dataGT[m])

@jit
def loss_unbatched(x):
  a1 = dataR * x
  a2 = vmap(cos2)(a1)
  return cos(a2, dataGT)

@jit
def cos(v1 : Vec, v2 : Vec) -> float:
  return jnp.sum(v1 * v2) / (jnp.sqrt(jnp.sum(v1**2)) * (jnp.sqrt(jnp.sum(v2**2))) + 0.000001)

@jit
def cos2(v : Vec) -> float:
  v1 = v[0]
  v2 = v[1]
  r = jnp.sum(v1 * v2) / (jnp.sqrt(jnp.sum(v1**2)) * (jnp.sqrt(jnp.sum(v2**2))) + 0.000001)
  return r

@jit
def euc2(v1 : Vec, v2 : Vec) -> float:
  return jnp.sqrt(jnp.sum((v1 - v2)**2))

@jit
def euc2(v1 : Vec, v2 : Vec) -> float:
  return jnp.sqrt(jnp.sum((v1 - v2)**2))


@jit
def mse2(v) -> float:
  v1 = v[0]
  v2 = v[1]
  return jnp.sum((v1 - v2)**2) / jnp.size(v1)

@jit
def mse(v1, v2) -> float:
  return jnp.sum((v1 - v2)**2) / jnp.size(v1)

vcos2 = vmap(cos2)
vmse = vmap(mse2)

loss_unbatched(x)

# Commented out IPython magic to ensure Python compatibility.
key = random.PRNGKey(0)
x = jnp.ones(len(K))
m, k = indices(key, len(M), len(K))
m2, k2 = indices(key, 128, len(K))
```

```
m3, k3 = indices(key, 128, 128)
# %timeit loss(x, m, k)
# %timeit loss(x, m2, k2)
# %timeit loss(x, m3, k3)

print(len(M))
print(len(K))

"""#### On CPU
##### 43660 Edges, 3933 Tokens
- 1 loop, best of 5: 12.5 s per loop

##### 128 Edges, 3933 Tokens
- 1 loop, best of 5: 12 ms per loop

##### 128 Edges, 128 Tokens
- 1 loop, best of 5: 196  s  per loop

#### On GPU
##### 43660 Edges, 3933 Tokens
- 100 loops, best of 5: 19.5 ms per loop

##### 128 Edges, 3933 Tokens
- 100 loops, best of 5: 124  s  per loop

##### 128 Edges, 128 Tokens
- 10000 loops, best of 5: 44.7  s  per loop

Looks like gradients don't work with cosine similarity.
Gradient immediately returns NaN

### Stochastic Gradient Descent
"""

def monitor(i, iters, p, x, F, F2):
  p2 = round((i/iters) * 100)
  if p2 > p:
    p = p2; print(p, "%")
    F += [loss(x, M, K)]
    F2 += [loss_mse(x, M, K)]
  return p, F, F2

def sgd(x0, alpha, iters, bm, bk, rngKey):
  x = x0

  F = []; F2 = []; p=0

  for i in range(iters):
    p, F, F2 = monitor(i, iters, p, x, F, F2)

    key, rngKey = random.split(rngKey)
    m, k = indices(key, bm, bk)

    g = (grad(loss_mse)(x, m, k))
    x = x - alpha * g
  return x, F, F2

def adam(x0, alpha, iters, bm, bk, b1, b2, rngKey):
  F = []; F2 = []; p=0

  x = x0
  am = jnp.zeros(len(x0)) ; av = jnp.zeros(len(x0)) ; ak = 1
  for i in range(iters):
    p, F, F2 = monitor(i, iters, p, x, F, F2)

    key, rngKey = random.split(rngKey)
    m, k = indices(key, bm, bk)

```

```
    # might need to skip the weight updates and history record when k decides to turn
     off some of the weigths
    # below part of adam can be jit'ed, need to extract it into separate function and
     carry over the context
    g = (grad(loss_mse)(x, m, k))
    am = b1 * am + (1 - b1) * g
    av = b2 * av + (1 - b2) * g**2
    mhat = (am / (1 - b1**ak))
    vhat = (av / (1 - b2**ak))
    x = x - alpha * (mhat / (jnp.sqrt(vhat) + 0.00001))
    ak = ak + 1

  return x, F, F2

"""### Running Code"""

key = random.PRNGKey(0)
x = jnp.ones(len(K))
# r, F, F2 = sgd(x, 0.1, 100_000, 1024, 2048, key)

# ar, aF, aF2 = adam(x, 0.001, 100_000, 1024, 2048, b1=0.9, b2=0.999, rngKey=key)

print("\n Cosine Similarity")
print("Word Count: \t", loss(x, m, k))
print("SGD: \t \t", loss(r, m, k))
print("Adam: \t \t",loss(ar, m, k))
print("\n Mean Squared Error")
print("Word Count: \t", loss_mse(x, m, k))
print("SGD: \t \t",loss_mse(r, m, k))
print("Adam: \t \t", loss_mse(ar, m, k))

import matplotlib.pyplot as plt
plt.plot(range(len(F)), F, label="SGD Constant")
plt.plot(range(len(aF)), aF, label="SGD Adam")
plt.title("Cosine Similarity vs. %Iters \n 100,000 iterations, 1024 Edge Batch, 2048
    Word Batch")
plt.legend()

plt.semilogy(range(len(F2)), F2, label="SGD Constant")
plt.semilogy(range(len(aF2)), aF2, label="SGD Adam")
plt.title("MSE vs. %Iters \n 100,000 iterations, 1024 Edge Batch, 2048 Word Batch")
plt.legend()

r[:20]

def invertIndexMap(m):
  m2 = {}
  for v1, v2 in m.items():
    m2[v2] = v1
  return m2

print(mapToTuples(wvIndex)[:10])
print(mapToTuples(invertIndexMap(wvIndex))[:10])

def weightsToWordWeightMap(indexToWord, weights):
  wordWeightMap = {}
  for idx, weight in enumerate(weights):
    wordWeightMap[indexToWord[idx]] = weight
  return wordWeightMap

indexToWord = invertIndexMap(wvIndex)
wordWeightMap = weightsToWordWeightMap(indexToWord, np.array(r))
wordWeightMapAdam = weightsToWordWeightMap(indexToWord, np.array(ar))

sortByVal(wordWeightMap)[:15]

sortByVal(wordWeightMap)[::-1][:10]

```

```
sortByVal(wordWeightMapAdam)[:15]

sortByVal(wordWeightMapAdam)[::-1][:10]

"""## With Optax
- SGD (Batching?)
  - Looks like SGD is actually just GD
  - Batching implemented by ourselves
- Loss Function?
  - https://optax.readthedocs.io/en/latest/api.html#common-losses
    - l2 aka means squared error
    - cosine distance


- So perhaps can just use the loss functions


- Though will still need to use Adam
  - So perhaps can use SGD since will need to fit into the framework anyway
"""

!pip install optax



from optax import cosine_distance, cosine_similarity, l2_loss
from jax import value_and_grad

M = jnp.arange(len(edgeVecIndex))
from sklearn.model_selection import train_test_split
M_train, M_test = train_test_split(M)

print(M_train.shape)

print(M_test.shape)
print(len(K))

vcos = vmap(jit(lambda x: cosine_distance(x[0], x[1], 1e-9)))
vl2 = vmap(jit(lambda x: jnp.mean(l2_loss(x.at[0].get(), x.at[1].get()))))

@jit
def loss(x, m):
  a1 = dataM.at[m].get()
  a2 = dataWV.at[a1].get()

  a4 = a2 * x
  a5 = vl2(a4)
  return jnp.mean(l2_loss(a5, dataGT[m]))

WV = dataWV.at[dataM.at[M].get()].get()

@jit
def loss_cos2(x, m):
  return cosine_distance(vcos(WV[m] * x), dataGT[m] * 0.001, 1e-9)

@jit
def loss_cos(x, m):
  a1 = dataM.at[m].get()
  a2 = dataWV.at[a1].get()
  a4 = a2 * x * 0.001
  a5 = vcos(a4)
  return cosine_distance(a5, dataGT[m] * 0.001, 1e-9)


# prob better way to do this using optax/jax ecosystem
def mkMonitor(iters, ltrain, ltest):
  def fn(state, x):
    p = state["p"]
```

```
    i = state["i"]
    Ftrain = state["Ftrain"]
    Ftest = state["Ftest"]

    p2 = round((i/iters) * 100)
    if p2 > p:
      state["p"] = p2;
      print(p2, "%")
      state["Ftrain"] = Ftrain + [ltrain(x)]
      state["Ftest"] = Ftest + [ltest(x)]

    state["i"] = i + 1
    return state

  state = {}
  state["p"] = -1
  state["Ftrain"] = []
  state["Ftest"]= []
  state["i"] = 0

  return state, fn

def sgd(x0, alpha, iters, b, rngKey):
  x = x0

  state, monitorIter = mkMonitor(iters,
                                 lambda x: loss_cos(x, M_train),
                                 lambda x: loss_cos(x, M_test))

  for i in range(iters):
    state = monitorIter(state, x)

    key, rngKey = random.split(rngKey)
    m = random.choice(key, M_train, [b])

    g = (grad(loss_cos)(x, m))
    x = x - alpha * g
  return x, state["Ftrain"], state["Ftest"]


def adam(x0, alpha, iters,b1, b2, b, rngKey):
  x = x0

  state, monitorIter = mkMonitor(iters,
                                lambda x: loss_cos(x, M_train),
                                lambda x: loss_cos(x, M_test))
  am = jnp.zeros(len(x0)) ; av = jnp.zeros(len(x0)) ; ak = 1
  for i in range(iters):
    state = monitorIter(state, x)
    key, rngKey = random.split(rngKey)
    m = random.choice(key, M_train, [b])

    g = (grad(loss_cos)(x, m))
    am = b1 * am + (1 - b1) * g
    av = b2 * av + (1 - b2) * g**2
    mhat = (am / (1 - b1**ak))
    vhat = (av / (1 - b2**ak))
    x = x - alpha * (mhat / (jnp.sqrt(vhat) + 0.00001))
    ak = ak + 1

  return x, state["Ftrain"], state["Ftest"]

@jit
def cos(x):
  a1 = dataM.at[M].get()
  a2 = dataWV.at[a1].get()
  a4 = a2 * x * 0.001
  a5 = vcos(a4)
```

```python
  return cosine_distance(a5, dataGT[M] * 0.001, 1e-9)

key = random.PRNGKey(0)
x = jnp.ones(len(K))
r, F = sgd(x, 500, 1000, 128, key)

F[-1]

print(cos(x))
print(cos(r))

import matplotlib.pyplot as plt
plt.semilogy(range(len(F)), F, label="SGD Constant")

r2, F2 = adam(x, 1, 1_000, 0.99, 0.9, 128, key)

plt.semilogy(range(len(F)), F, label="Constant")
plt.semilogy(range(len(F2)), F2, label="Adam ")

def global_random_search(intervals, N, f):
    lowest = None
    l = [l for l, u in intervals]
    u = [u for l, u in intervals]

    tries = []

    for s in range(N):
        r = np.random.uniform(l, u)
        print("\niteration:", s, "trying out:", r)
        v = f(*r)
        print("got", v)
        if (not lowest) or lowest[0] > v:
            lowest = (v.copy(), r.copy())
        tries += [(v, r.copy())]
    return tries, lowest

def run_multiple(runs, f):
    loss_histories = []
    test_losses = []

    seed = random.PRNGKey(0)
    seed, subseed = random.split(seed)

    for r in range(runs):
        print("Run number:", r)
        loss_history = f(subseed)
        seed, subseed = random.split(seed)
        loss_histories += [loss_history]
    return loss_histories

iters = 3_000

def f_sgd(learning_rate, batch_size, rngkey):
    r, F = sgd(x, learning_rate, iters, round(batch_size), rngkey)
    return F

def f_sgd_key(learning_rate, batch_size):
  rngkey = random.PRNGKey(0)
  return f_sgd(learning_rate, round(batch_size), rngkey)[-1]

sgd_optimal_alpha = 1000
sgd_optimal_batch = 2000

sgd_default_alpha = 0.1
sgd_default_batch = 128

adam_optimal_alpha = 1.9
adam_optimal_b1 = 0.869
```

```
adam_optimal_b2 = 0.662
adam_optimal_batch = 960

adam_default_alpha = 0.01
adam_default_b1 = 0.9
adam_default_b2 = 0.999
adam_default_batch = 128

def f_sgd_optimal(seed):
  return f_sgd(sgd_optimal_alpha, sgd_optimal_batch, seed)

def f_sgd_default(seed):
  return f_sgd(sgd_default_alpha, sgd_default_batch, seed)

def f_adam(learning_rate, b1, b2, batch_size, rngkey):
    r, F = adam(x, learning_rate, iters, b1, b2, round(batch_size), rngkey)
    return F

def f_adam_key(learning_rate, b1, b2, batch_size):
  rngkey = random.PRNGKey(0)
  return f_adam(learning_rate, b1, b2, round(batch_size), rngkey)[-1]

def f_adam_optimal(seed):
  return f_adam(adam_optimal_alpha, adam_optimal_b1, adam_optimal_b2,
    adam_optimal_batch, seed)


def f_adam_default(seed):
  return f_adam(adam_default_alpha, adam_default_b1, adam_default_b2,
    adam_default_batch, seed)

seed = random.PRNGKey(0)
xar, Faaa, Faaatest = adam(x, adam_optimal_alpha, 100_000, adam_optimal_b1,
    adam_optimal_b2, adam_optimal_batch, seed)

xsr, Fsss, Fssstest = sgd(x, sgd_optimal_alpha, 100_000, sgd_optimal_batch, seed)

print("Baseline")
print("Test:\t", loss_cos(x, M_test))
print("Train:\t",loss_cos(x, M_train))
print("Adam")
print("Test:\t", loss_cos(xar, M_test))
print("Train:\t",loss_cos(xar, M_train))
print("Constant")
print("Test:\t",loss_cos(xsr, M_test))
print("Train:\t",loss_cos(xsr, M_train))

def compare_tt(F, F2, title="Title", Fl="r1", F2l="r2"):
    xs = range(len(F))
    plt.semilogy(xs, F, color='#2e6fd9bb', label=Fl)

    xlim = plt.xlim()
    ylim = plt.ylim()
    plt.semilogy(xs, F2, color='#ff9c24bb', label=F2l)

    plt.xlim(xlim)
    plt.ylim(ylim)

    plt.title(title)
    plt.legend()

    plt.xlabel(r'%run')
    plt.ylabel(r'loss')

compare_tt(Fsss, Fssstest, "SGD Optimal", "Train", "Test")
# optimal parameters were picked with train results, perhaps would expect something
    different with test evaluation

```

```
compare_tt(Faaa, Faaatest, "Adam Optimal", "Train", "Test")

print(M.shape)
print(M_test.shape)
print(M_train.shape)

psgd = [(1, 1000), (12, 2048)]
lsgd, lowsgd = global_random_search(psgd, 20, f_sgd_key)

lowsgd

(array(0.91140574, dtype=float32), array([ 922.32970583, 2032.0561759 ]))

ys1 = 1 - np.array([x for x, (_, _) in lsgd])
xs2 = [y for _, (x, y) in lsgd]
xs1 = [x for _, (x, y) in lsgd]
import matplotlib.pyplot as plt

#plt.scatter(xs, ys1)
ax = plt.axes(projection='3d')
ax.scatter3D(xs1, xs2, ys1, c=ys1, cmap='Greens');
ax.view_init(30, 20)

plt.scatter(xs1, ys1)

plt.scatter(xs2, ys1)

padam = [(0.1, 100), (0.5,0.99), (0.5,0.99), (12, 2049)]
ladam, lowadam = global_random_search(padam, 20, f_adam_key)

lowadam

(array(0.9106287, dtype=float32),
 array([1.92316220e+00, 8.68884077e-01, 6.62402454e-01, 9.57793168e+02]))

lha= run_multiple(20, f_adam_optimal)

lhad= run_multiple(20, f_adam_default)

lho = run_multiple(20, f_sgd_optimal)

lhod = run_multiple(20, f_sgd_default)

"""- global random search and plotting parameters
- optimal and default
- split data set into test and train
"""

sgd_default_loss_histories = lhod
sgd_optimal_loss_histories = lho
adam_default_loss_histories = lhad
adam_optimal_loss_histories = lha

import pickle

mlruns = {
    "sgd_default_loss_histories": sgd_default_loss_histories,
   # "sgd_default_test_losses": sgd_default_test_losses,
    "sgd_optimal_loss_histories": sgd_optimal_loss_histories,
   # "sgd_optimal_test_losses": sgd_optimal_test_losses,

    "adam_default_loss_histories": adam_default_loss_histories,
  #  "adam_default_test_losses": adam_default_test_losses,
    "adam_optimal_loss_histories": adam_optimal_loss_histories,
  #  "adam_optimal_test_losses": adam_optimal_test_losses
}

pickle.dump(mlruns, open("./drive/MyDrive/TA/mlruns.p", "wb"))
```

```python

import pickle
mlruns_l = pickle.load(open( "./drive/MyDrive/TA/mlruns.p", "rb" ))

def plot_history(losses):
    'losses :: [[float]], ith element is loss vs iteration of ith run of the SGD'
    losses = np.array(losses)
    average_on_iter_i = np.mean(losses, axis=0)
    min_on_iter_i = np.minimum.reduce(losses)
    max_on_iter_i = np.maximum.reduce(losses)
    x = range(len(average_on_iter_i))
    plt.plot(x, average_on_iter_i , 'k-')
    plt.fill_between(x, min_on_iter_i, max_on_iter_i)

def avg_max_min(loss_histories):
    average_on_iter_i = np.mean(loss_histories, axis=0)
    min_on_iter_i = np.minimum.reduce(loss_histories)
    max_on_iter_i = np.maximum.reduce(loss_histories)
    return average_on_iter_i, min_on_iter_i, max_on_iter_i

def compare_sgd(r1, r2, title="Title", r1l="r1", r2l="r2"):
    r1 = np.array(r1) ; r2 = np.array(r2)
    xr = r1.shape[1]  ; xr2 = r2.shape[1]
    x1 = np.linspace(0, 100, xr)
    x2 = np.linspace(0, 100, xr2)
    a1, l1, h1 = avg_max_min(r1)
    a2, l2, h2 = avg_max_min(r2)

    plt.semilogy(x1, a1, color='#2e6fd9bb', label=r1l)
    plt.fill_between(x1, l1, h1, color="#3d84f588")
    xlim = plt.xlim()
    ylim = plt.ylim()
    plt.semilogy(x2, a2, color='#ff9c24bb', label=r2l)
    plt.fill_between(x2, l2, h2, color='#ffc37088')
    plt.xlim(xlim)
    plt.ylim(ylim)
    plt.title(title)
    plt.legend()

    plt.xlabel(r'%run')
    plt.ylabel(r'loss')

import matplotlib.pyplot as plt

r1 = np.array(mlruns_l['sgd_default_loss_histories'])
r2 = np.array(mlruns_l['sgd_optimal_loss_histories'])

compare_sgd(r2, r1, title="Constant: Default vs. Optimal", r1l="Optimal", r2l="
    Default")

r1 = np.array(mlruns_l['adam_default_loss_histories'])
r2 = np.array(mlruns_l['adam_optimal_loss_histories'])

compare_sgd(r1=r2, r2=r1, title="Adam: Default vs. Optimal", r1l="Optimal", r2l="
    Default")

r1 = np.array(mlruns_l['sgd_optimal_loss_histories'])
r2 = np.array(mlruns_l['adam_optimal_loss_histories'])
compare_sgd(r1=r1, r2=r2, title="Optimal Constant vs. Optimal Adam", r1l="Constant",
    r2l="Adam")

r1 = np.array(mlruns_l['sgd_default_loss_histories'])
r2 = np.array(mlruns_l['adam_default_loss_histories'])
compare_sgd(r1=r2, r2=r1, title="Default Constant vs. Default Adam", r1l="Adam", r2l=
    "Constant")
```

4 Bibliography.

- https://optax.readthedocs.io/en/latest/api.html#optax.cosine_distance
- https://github.com/ErnestKz/TextAnalyticsReport/blob/main/Text_Analytics_Final_Paper%20(1).pdf
- https://github.com/google/flax/tree/main/examples/mnist
- https://optax.readthedocs.io/en/latest/api.html#optax.softmax_cross_entropy
- https://jax.readthedocs.io/en/latest/