

# Optimisation Algorithms - Final Assignment

Ernests Kuznecovs - 17332791 - kuznecoe@tcd.ie

2021-2022

## Contents

<b>1</b>	<b>Stochastic Gradient Descent: Constant and Adam Step Sizes</b>	<b>1</b>
<b>2</b>	<b>Report</b>	<b>1</b>
2.1	Model and Dataset 1: MNIST . . . . .	1
2.1.1	Selecting Hyper Parameters . . . . .	2
2.1.2	Comparison . . . . .	2
2.1.3	Generalised vs. Ungeneralised . . . . .	4
2.2	Model and Dataset 2: Twitter Network Wordvectors . . . . .	5
2.2.1	Selecting Hyper Parameters . . . . .	5
2.2.2	Comparison . . . . .	6
2.2.3	Generalised vs. Ungeneralised . . . . .	7
<b>3</b>	<b>Appendix</b>	<b>8</b>
3.1	Code Listing . . . . .	8
<b>4</b>	<b>Bibliography.</b>	<b>36</b>

## 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](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

Ultimately 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. These 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 lighter 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 noisy nature and a lucky score is what caused the global random search to pick the parameters. Perhaps there existed 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 noisy, and there could be parameters that aren't as noisy. 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 noisy 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 noisy. This could be because adam has double the batch size. Also perhaps that the noisy 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.

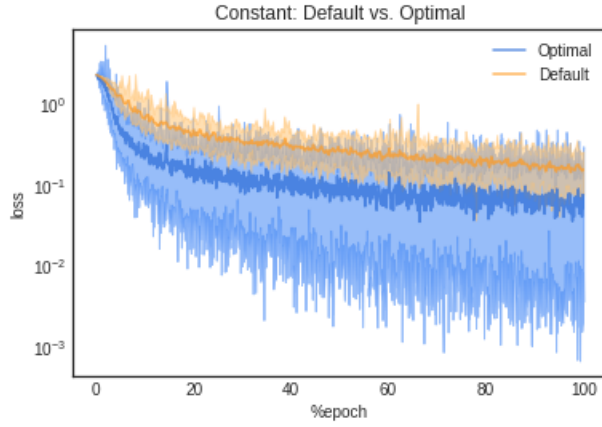


Figure 1

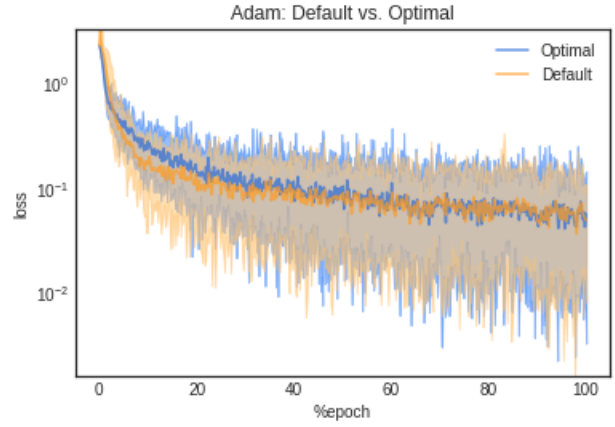


Figure 2

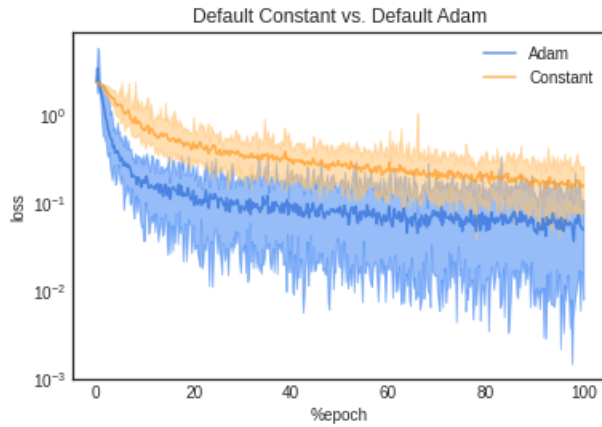


Figure 3

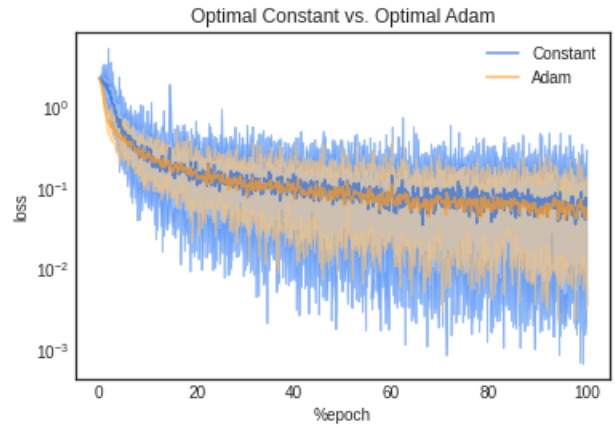


Figure 4

### 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 occurred, and for Adam 612 (due to the different batch sizes and constant 1 epoch).

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

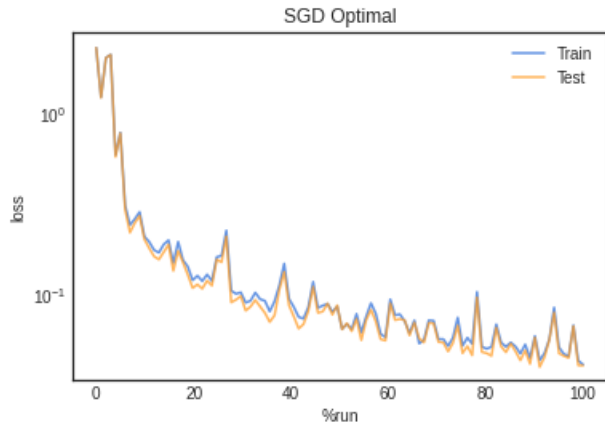


Figure 5

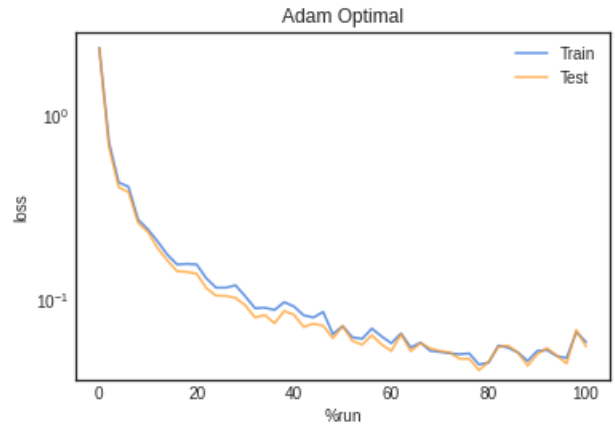


Figure 6

## 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](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 occurrences 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 similarity 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](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 a 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 mistakenly 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 magnitudinal 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 through 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 noisy i.e it is a linear model rather than a non-linear neural network.

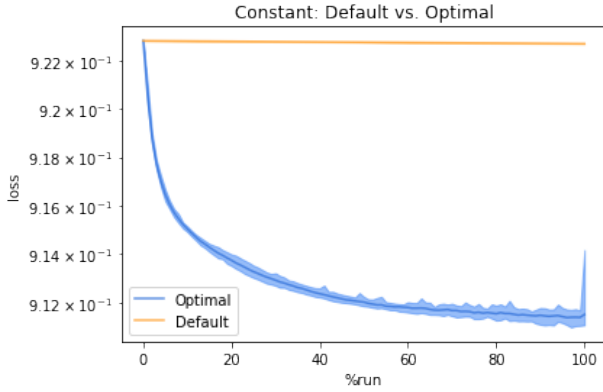


Figure 7

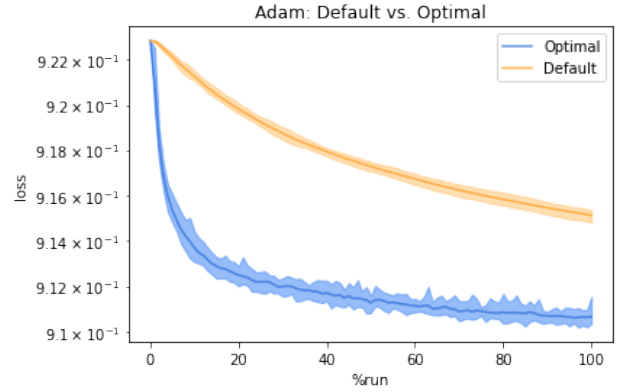


Figure 8

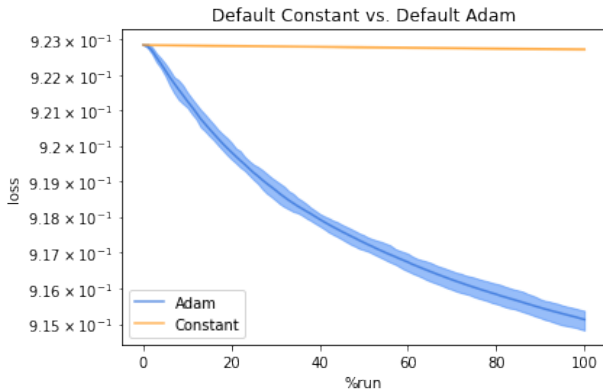


Figure 9

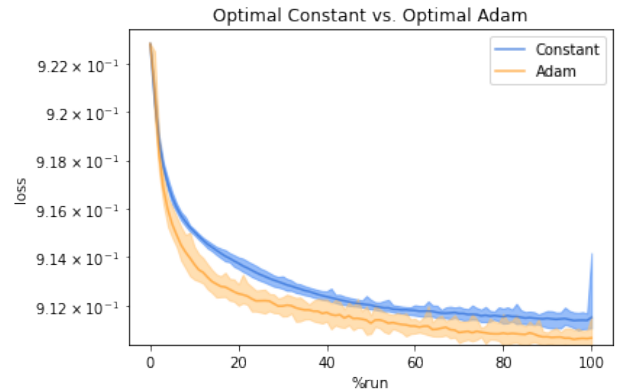


Figure 10

### 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 peculiarities 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 training set.

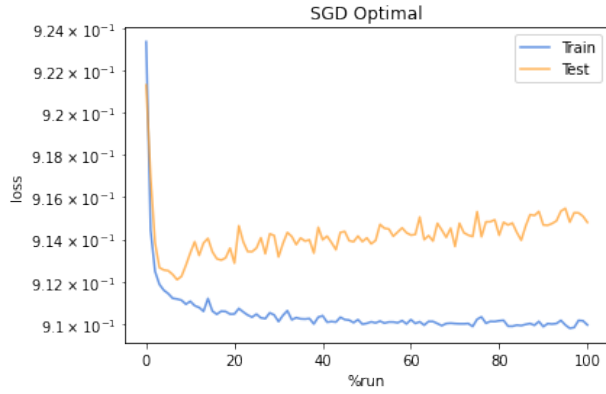


Figure 11

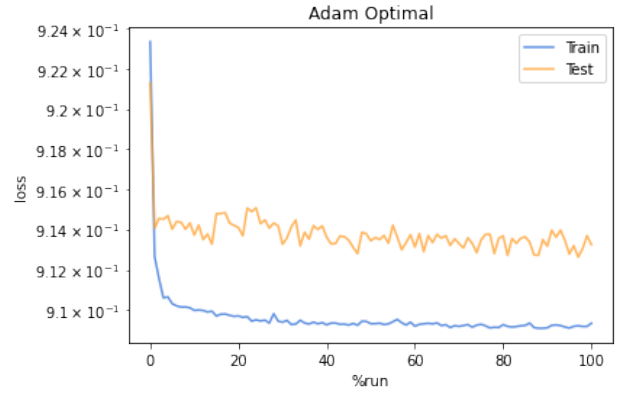


Figure 12

## 3 Appendix

### 3.1 Code Listing

```
1 from typing import Sequence
2
3 import numpy as np
4 import jax
5 import jax.numpy as jnp
6 import flax.linen as nn
7
8 class MLP(nn.Module):
9     features: Sequence[int]
10
11     @nn.compact
12     def __call__(self, x):
13         for feat in self.features[:-1]:
14             x = nn.relu(nn.Dense(feat)(x))
15         x = nn.Dense(self.features[-1])(x)
16         return x
17
18 model = MLP([12, 8, 4])
19 batch = jnp.ones((32, 10))
20 variables = model.init(jax.random.PRNGKey(0), batch)
21 output = model.apply(variables, batch)
22
23 import random
24 from typing import Tuple
25
26 import optax
27 import jax.numpy as jnp
28 import jax
29 import numpy as np
30
31 BATCH_SIZE = 5
32 NUM_TRAIN_STEPS = 1_000
33 RAW_TRAINING_DATA = np.random.randint(255, size=(NUM_TRAIN_STEPS, BATCH_SIZE, 1))
34
35 TRAINING_DATA = np.unpackbits(RAW_TRAINING_DATA.astype(np.uint8), axis=-1)
36 LABELS = jax.nn.one_hot(RAW_TRAINING_DATA % 2, 2).astype(jnp.float32).reshape(
37     NUM_TRAIN_STEPS, BATCH_SIZE, 2)
38
39 initial_params = {
40     'hidden': jax.random.normal(shape=[8, 32], key=jax.random.PRNGKey(0)),
41     'output': jax.random.normal(shape=[32, 2], key=jax.random.PRNGKey(1)),
42 }
43
44 def net(x: jnp.ndarray, params: jnp.ndarray) -> jnp.ndarray:
45     x = jnp.dot(x, params['hidden'])
46     x = jax.nn.relu(x)
47     x = jnp.dot(x, params['output'])
48     return x
49
50
51 def loss(params: optax.Params, batch: jnp.ndarray, labels: jnp.ndarray) -> jnp.
52     ndarray:
53     y_hat = net(batch, params)
54
55     # optax also provides a number of common loss functions.
56     loss_value = optax.sigmoid_binary_cross_entropy(y_hat, labels).sum(axis=-1)
57
58     return loss_value.mean()
59
60 def fit(params: optax.Params, optimizer: optax.GradientTransformation) -> optax.
61     Params:
62     opt_state = optimizer.init(params)
```



```

62 @jax.jit
63 def step(params, opt_state, batch, labels):
64     loss_value, grads = jax.value_and_grad(loss)(params, batch, labels)
65     updates, opt_state = optimizer.update(grads, opt_state, params)
66     params = optax.apply_updates(params, updates)
67     return params, opt_state, loss_value
68
69 for i, (batch, labels) in enumerate(zip(TRAINING_DATA, LABELS)):
70     params, opt_state, loss_value = step(params, opt_state, batch, labels)
71     if i % 100 == 0:
72         print(f'step {i}, loss: {loss_value}')
73
74     return params
75
76 # Finally, we can fit our parametrized function using the Adam optimizer
77 # provided by optax.
78 optimizer = optax.adam(learning_rate=1e-2)
79 optimizer2 = optax.sgd(learning_rate=1e-2)
80 params = fit(initial_params, optimizer)
81 params = fit(initial_params, optimizer2)
82
83 import matplotlib as mpl
84 mpl.rcParams['figure.dpi'] = 200
85 mpl.rcParams['figure.facecolor'] = '1'
86 import matplotlib.pyplot as plt
87 plt.style.use('seaborn-white')
88
89 import copy
90 import numpy as np
91 from sklearn import metrics
92
93 from absl import logging
94 from flax import linen as nn
95 from flax.metrics import tensorboard
96 from flax.training import train_state
97 import jax
98 import jax.numpy as jnp
99 import ml_collections
100 import numpy as np
101 import optax
102 import tensorflow_datasets as tfds
103
104 class CNN(nn.Module):
105     """A simple CNN model."""
106
107     @nn.compact
108     def __call__(self, x):
109         x = nn.Conv(features=32, kernel_size=(3, 3))(x)
110         x = nn.relu(x)
111         x = nn.avg_pool(x, window_shape=(2, 2), strides=(2, 2))
112         x = nn.Conv(features=64, kernel_size=(3, 3))(x)
113         x = nn.relu(x)
114         x = nn.avg_pool(x, window_shape=(2, 2), strides=(2, 2))
115         x = x.reshape((x.shape[0], -1)) # flatten
116         x = nn.Dense(features=256)(x)
117         x = nn.relu(x)
118         x = nn.Dense(features=10)(x)
119         return x
120
121 @jax.jit
122 def apply_model(state, images, labels):
123     """Computes gradients, loss and accuracy for a single batch."""
124     def loss_fn(params):
125         logits = CNN().apply({'params': params}, images)
126         one_hot = jax.nn.one_hot(labels, 10)
127         loss = jnp.mean(optax.softmax_cross_entropy(logits=logits, labels=one_hot))
128         return loss, logits
129

```

```

130 grad_fn = jax.value_and_grad(loss_fn, has_aux=True)
131 (loss, logits), grads = grad_fn(state.params)
132 accuracy = jnp.mean(jnp.argmax(logits, -1) == labels)
133 return grads, loss, accuracy
134
135 @jax.jit
136 def update_model(state, grads):
137     return state.apply_gradients(grads=grads)
138
139 (21 % 20 == 0)
140
141 def train_epoch(state, train_ds, batch_size, rng, loss_history, test_loss_history,
142                 test_ds):
143     """Train for a single epoch."""
144     train_ds_size = len(train_ds['image'])
145     steps_per_epoch = train_ds_size // batch_size
146
147     perms = jax.random.permutation(rng, len(train_ds['image']))
148     perms = perms[:steps_per_epoch * batch_size] # skip incomplete batch
149     perms = perms.reshape((steps_per_epoch, batch_size))
150
151     epoch_loss = []
152     epoch_accuracy = []
153     print("perms:", len(perms))
154     i = 0
155
156     test_images = test_ds['image']
157     test_labels = test_ds['label']
158     train_images = train_ds['image']
159     train_labels = train_ds['label']
160
161     for perm in perms:
162         if (i % 12 == 0):
163             print("iteration", i, "out of", len(perms))
164             grads, loss, accuracy = apply_model(state, test_images, test_labels)
165             test_loss_history.append(loss)
166             grads, loss, accuracy = apply_model(state, train_images, train_labels)
167             loss_history.append(loss)
168
169             i += 1
170             batch_images = train_ds['image'][perm, ...]
171             batch_labels = train_ds['label'][perm, ...]
172
173             grads, loss, accuracy = apply_model(state, batch_images, batch_labels)
174             state = update_model(state, grads)
175             epoch_loss.append(loss)
176             epoch_accuracy.append(accuracy)
177
178     train_loss = np.mean(epoch_loss)
179     train_accuracy = np.mean(epoch_accuracy)
180     return state, train_loss, train_accuracy
181
182 def get_datasets():
183     """Load MNIST train and test datasets into memory."""
184     ds_builder = tfds.builder('mnist')
185     ds_builder.download_and_prepare()
186     train_ds = tfds.as_numpy(ds_builder.as_dataset(split='train', batch_size=-1))
187     test_ds = tfds.as_numpy(ds_builder.as_dataset(split='test', batch_size=-1))
188     train_ds['image'] = jnp.float32(train_ds['image']) / 255.
189     test_ds['image'] = jnp.float32(test_ds['image']) / 255.
190     return train_ds, test_ds
191
192 def create_train_state(rng, config):
193     """Creates initial `TrainState`."""
194     cnn = CNN()
195     params = cnn.init(rng, jnp.ones([1, 28, 28, 1]))['params']
196

```

```

197 tx = config.optimiser
198
199 return train_state.TrainState.create(
200     apply_fn=cnn.apply, params=params, tx=tx)
201
202 def train_and_evaluate(config: ml_collections.ConfigDict,
203     workdir: str,
204     train_ds,
205     test_ds,
206     seed):
207
208     rng, init_rng = jax.random.split(seed)
209     state = create_train_state(init_rng, config)
210
211     _, test_loss, test_accuracy = apply_model(state, test_ds['image'], test_ds['label'
212 ])
213     # print('epoch:% 3d, test_loss: %.4f, test_accuracy: %.2f'
214     #       % (0, test_loss, test_accuracy * 100))
215
216     loss_history = []
217     test_loss_history = []
218
219     for epoch in range(1, config.num_epochs + 1):
220         rng, input_rng = jax.random.split(rng)
221         state, train_loss, train_accuracy = train_epoch(state, train_ds, config.
222 batch_size, input_rng, loss_history, test_loss_history, test_ds)
223         _, test_loss, test_accuracy = apply_model(state, test_ds['image'], test_ds['label
224 '])
225         print('epoch:% 3d, train_loss: %.4f, train_accuracy: %.2f, test_loss: %.4f,
226 test_accuracy: %.2f'
227             % (epoch, train_loss, train_accuracy * 100, test_loss, test_accuracy * 100)
228         )
229     return state, loss_history, test_loss_history
230
231 def get_config(opt, batch_size):
232     """Get the default hyperparameter configuration."""
233     config = ml_collections.ConfigDict()
234     config.optimiser = opt
235     config.batch_size = batch_size
236     config.num_epochs = 1
237     return config
238
239 train_ds, test_ds = get_datasets()
240
241 print(train_ds.keys())
242 print(train_ds['image'].shape)
243 print(train_ds['label'].shape)
244 print(test_ds['label'].shape)
245
246 def f(learning_rate, b1, b2, batch_size):
247     opt = optax.adam(learning_rate=learning_rate, b1=b1, b2=b2)
248     cfg = get_config(opt=opt, batch_size=round(batch_size))
249     _, _, test_loss = train_and_evaluate(cfg, "./mnist/", train_ds, test_ds)
250     return test_loss
251
252 def f2(learning_rate, batch_size):
253     opt = optax.sgd(learning_rate=learning_rate)
254     cfg = get_config(opt=opt, batch_size=round(batch_size))
255     _, _, test_loss = train_and_evaluate(cfg, "./mnist/", train_ds, test_ds)
256     return test_loss
257
258 def global_random_search(intervals, N, f):
259     lowest = None
260     l = [l for l, u in intervals]
261     u = [u for l, u in intervals]

```

```

260     for s in range(N):
261         r = np.random.uniform(1, u)
262         print("iteration:", s, "trying out:", r)
263         v = f(*r)
264         if (not lowest) or lowest[0] > v:
265             lowest = (v.copy(), r.copy())
266     return lowest
267
268 v = global_random_search([(0.001, 0.1), (0.5,0.99), (0.5,0.99), (1, 128)], 20, f)
269
270 v
271
272 learning_rate = 0.0015
273 beta1 = 0.898
274 beta2 = 0.9575
275 batch_size = 98
276
277 v2 = global_random_search([(0.4, 0.8), (40, 90)], 20, f2)
278
279 print(v2)
280
281 # (array(0.05145077, dtype=float32), array([ 0.49315356, 58.39919518]))
282 # (array(0.05257225, dtype=float32), array([ 0.75313327, 93.05358694]))
283 # (array(0.04633828, dtype=float32), array([ 0.48917857, 48.61637121]))
284 learning_rate = 0.489
285 batch_size = 49
286
287 # opt = optax.sgd(learning_rate=0.1)
288 opt = optax.adam(learning_rate=0.001, b1=0.9, b2=0.999)
289 cfg = get_config(opt=opt, batch_size=128)
290 # state, loss_history, test_loss = train_and_evaluate(cfg, "./mnist/", train_ds,
291               test_ds)
292
293 print(len(loss_history))
294 print(len(train_ds['label']/128))
295
296 plt.plot(range(len(loss_history)), loss_history)
297
298 def sgdf(learning_rate, batch_size, seed):
299     opt = optax.sgd(learning_rate=learning_rate)
300     cfg = get_config(opt=opt, batch_size=round(batch_size))
301     _, loss_history, test_loss_history = train_and_evaluate(cfg, "./mnist/", train_ds,
302                                                            test_ds, seed)
303     return loss_history, test_loss_history
304
305 def adamf(learning_rate, b1, b2, batch_size, seed):
306     opt = optax.adam(learning_rate=learning_rate, b1=b1, b2=b2)
307     cfg = get_config(opt=opt, batch_size=round(batch_size))
308     _, loss_history, test_loss_history = train_and_evaluate(cfg, "./mnist/", train_ds,
309                                                            test_ds, seed)
310     return loss_history, test_loss_history
311
312 def run_multiple(runs, f):
313     # need to thread random seed
314
315     loss_histories = []
316     test_losses = []
317
318     seed = jax.random.PRNGKey(0)
319     seed, subseed = jax.random.split(seed)
320
321     for r in range(runs):
322         print("Run number:", r)
323         loss_history, test_loss_history = f(subseed)
324         seed, subseed = jax.random.split(seed)
325         loss_histories += [loss_history]
326         test_losses += [test_loss_history]
327     return loss_histories, test_losses

```

```

325
326 sgd_default_alpha = 0.1
327 sgd_default_batch = 128
328 sgd_default = lambda seed: sgdf(sgd_default_alpha, sgd_default_batch, seed=seed)
329
330 sgd_optimal_alpha = 0.489
331 sgd_optimal_batch = 49
332 sgd_optimal = lambda seed: sgdf(sgd_optimal_alpha, sgd_optimal_batch, seed=seed)
333
334 adam_default_alpha = 0.01
335 adam_default_b1 = 0.9
336 adam_default_b2 = 0.999
337 adam_default_batch = 128
338 adam_default = lambda seed: adamf(adam_default_alpha, adam_default_b1,
    adam_default_b2, adam_default_batch, seed=seed)
339
340 adam_optimal_alpha = 0.0015
341 adam_optimal_b1 = 0.898
342 adam_optimal_b2 = 0.9575
343 adam_optimal_batch = 98
344 adam_optimal = lambda seed: adamf(adam_optimal_alpha, adam_optimal_b1,
    adam_optimal_b2, adam_optimal_batch, seed=seed)
345
346 rng = jax.random.PRNGKey(0)
347
348 sgd_train_loss, sgd_test_loss = sgd_optimal(rng)
349
350 adam_train_loss, adam_test_loss = adam_optimal(rng)
351
352 sgd_train_loss
353
354 sgd_test_loss
355
356 r1 = np.array(sgd_train_loss)
357 r1.shape[0]
358
359 compare_tt(sgd_train_loss, sgd_test_loss, "SGD Optimal", "Train", "Test")
360
361 compare_tt(adam_train_loss, adam_test_loss, "Adam Optimal", "Train", "Test")
362
363 runs = 2
364 sgd_default_loss_histories, sgd_default_test_losses = run_multiple(runs, sgd_default)
365
366 print(sgd_default_test_losses)
367
368 runs = 20
369 print("SGD Default")
370 sgd_default_loss_histories, sgd_default_test_losses = run_multiple(runs, sgd_default)
371
372 print("SGD Optimal")
373 sgd_optimal_loss_histories, sgd_optimal_test_losses = run_multiple(runs, sgd_optimal)
374
375 print("Adam Default")
376 adam_default_loss_histories, adam_default_test_losses = run_multiple(runs,
    adam_default)
377
378 print("Adam Optimal")
379 adam_optimal_loss_histories, adam_optimal_test_losses = run_multiple(runs,
    adam_optimal)
380
381 import pickle
382
383 mlruns = {
384     "sgd_default_loss_histories": sgd_default_loss_histories,
385     "sgd_default_test_losses": sgd_default_test_losses,
386     "sgd_optimal_loss_histories": sgd_optimal_loss_histories,
387     "sgd_optimal_test_losses": sgd_optimal_test_losses,
388

```

```

389     "adam_default_loss_histories": adam_default_loss_histories,
390     "adam_default_test_losses": adam_default_test_losses,
391     "adam_optimal_loss_histories": adam_optimal_loss_histories,
392     "adam_optimal_test_losses": adam_optimal_test_losses
393 }
394
395 pickle.dump(mlruns, open("mlruns.p", "wb"))
396
397 import pickle
398 mlruns_1 = pickle.load(open("mlruns.p", "rb"))
399
400 mlruns_1.keys()
401
402 def plot_history(losses):
403     'losses :: [[float]], ith element is loss vs iteration of ith run of the SGD'
404     losses = np.array(losses)
405     average_on_iter_i = np.mean(losses, axis=0)
406     min_on_iter_i = np.minimum.reduce(losses)
407     max_on_iter_i = np.maximum.reduce(losses)
408     x = range(len(average_on_iter_i))
409     plt.plot(x, average_on_iter_i, 'k-')
410     plt.fill_between(x, min_on_iter_i, max_on_iter_i)
411
412 def avg_max_min(loss_histories):
413     average_on_iter_i = np.mean(loss_histories, axis=0)
414     min_on_iter_i = np.minimum.reduce(loss_histories)
415     max_on_iter_i = np.maximum.reduce(loss_histories)
416     return average_on_iter_i, min_on_iter_i, max_on_iter_i
417
418 plot_history(mlruns_1['sgd_default_loss_histories'])
419
420 plot_history(mlruns_1['sgd_optimal_loss_histories'])
421
422 plot_history(mlruns_1['adam_default_loss_histories'])
423
424 plot_history(mlruns_1['adam_optimal_loss_histories'])
425
426 np.array(mlruns_1['sgd_default_loss_histories']).shape
427
428 np.array(mlruns_1['sgd_optimal_loss_histories']).shape
429
430 def compare_sgd(r1, r2, title="Title", r1l="r1", r2l="r2"):
431     r1 = np.array(r1) ; r2 = np.array(r2)
432     xr = r1.shape[1] ; xr2 = r2.shape[1]
433     x1 = np.linspace(0, 100, xr)
434     x2 = np.linspace(0, 100, xr2)
435     a1, l1, h1 = avg_max_min(r1)
436     a2, l2, h2 = avg_max_min(r2)
437
438     plt.semilogy(x1, a1, color='#2e6fd9bb', label=r1l)
439     plt.fill_between(x1, l1, h1, color="#3d84f588")
440     xlim = plt.xlim()
441     ylim = plt.ylim()
442     plt.semilogy(x2, a2, color='#ff9c24bb', label=r2l)
443     plt.fill_between(x2, l2, h2, color='#ffc37088')
444     plt.xlim(xlim)
445     plt.ylim(ylim)
446     plt.title(title)
447     plt.legend()
448
449     plt.xlabel(r'%epoch')
450     plt.ylabel(r'loss')
451     # plt.title("default vs optimal")
452
453 r1 = np.array(mlruns_1['sgd_default_loss_histories'])
454 r2 = np.array(mlruns_1['sgd_optimal_loss_histories'])
455
456 compare_sgd(r2, r1, title="Constant: Default vs. Optimal", r1l="Optimal", r2l="

```

```

    Default")
457
458 r1 = np.array(mlruns_1['adam_default_loss_histories'])
459 r2 = np.array(mlruns_1['adam_optimal_loss_histories'])
460
461 compare_sgd(r1=r2, r2=r1, title="Adam: Default vs. Optimal", r1l="Optimal", r2l="
    Default")
462
463 r1 = np.array(mlruns_1['sgd_optimal_loss_histories'])
464 r2 = np.array(mlruns_1['adam_optimal_loss_histories'])
465 compare_sgd(r1=r1, r2=r2, title="Optimal Constant vs. Optimal Adam", r1l="Constant",
    r2l="Adam")
466
467 r1 = np.array(mlruns_1['sgd_default_loss_histories'])
468 r2 = np.array(mlruns_1['adam_default_loss_histories'])
469 compare_sgd(r1=r2, r2=r1, title="Default Constant vs. Default Adam", r1l="Adam", r2l="
    Constant")
470
471 def compare_tt(F, F2, title="Title", F1="r1", F2l="r2"):
472
473     r1 = np.array(F) ; r2 = np.array(F2)
474     xr = r1.shape[0] ; xr2 = r2.shape[0]
475
476     x1 = np.linspace(0, 100, xr)
477     x2 = np.linspace(0, 100, xr2)
478
479     plt.semilogy(x1, F, color='#2e6fd9bb', label=F1)
480
481     xlim = plt.xlim()
482     ylim = plt.ylim()
483     plt.semilogy(x2, F2, color='#ff9c24bb', label=F2l)
484
485     plt.xlim(xlim)
486     plt.ylim(ylim)
487
488     plt.title(title)
489     plt.legend()
490
491     plt.xlabel(r'%run')
492     plt.ylabel(r'loss')
493
494 print(mlruns_1['sgd_optimal_test_losses'][:10])
495 print(np.array(mlruns_1['sgd_optimal_test_losses']).shape)
496 print(np.array(mlruns_1['sgd_optimal_loss_histories']).shape)
497 print(mlruns_1['sgd_optimal_test_losses'])
498 print([x[-1] for x in mlruns_1['sgd_optimal_loss_histories']])
499
500
501
502 """Trains an SST2 text classifier."""
503 from typing import Any, Callable, Dict, Iterable, Optional, Sequence, Tuple, Union
504
505 from absl import logging
506 from flax import struct
507 from flax.metrics import tensorboard
508 from flax.training import train_state
509 import jax
510 import jax.numpy as jnp
511 import ml_collections
512 import numpy as np
513 import optax
514 import tensorflow as tf
515
516 import input_pipeline
517 import models
518
519
520 Array = jnp.ndarray

```

```

521 Example = Dict[str, Array]
522 TrainState = train_state.TrainState
523
524
525 class Metrics(struct.PyTreeNode):
526     """Computed metrics."""
527     loss: float
528     accuracy: float
529     count: Optional[int] = None
530
531
532 @jax.vmap
533 def sigmoid_cross_entropy_with_logits(*, labels: Array, logits: Array) -> Array:
534     """Sigmoid cross entropy loss."""
535     zeros = jnp.zeros_like(logits, dtype=logits.dtype)
536     condition = (logits >= zeros)
537     relu_logits = jnp.where(condition, logits, zeros)
538     neg_abs_logits = jnp.where(condition, -logits, logits)
539     return relu_logits - logits * labels + jnp.log1p(jnp.exp(neg_abs_logits))
540
541
542 def get_initial_params(rng, model):
543     """Returns randomly initialized parameters."""
544     token_ids = jnp.ones((2, 3), jnp.int32)
545     lengths = jnp.ones((2,), dtype=jnp.int32)
546     variables = model.init(rng, token_ids, lengths, deterministic=True)
547     return variables['params']
548
549
550 def create_train_state(rng, config: ml_collections.ConfigDict, model):
551     """Create initial training state."""
552     params = get_initial_params(rng, model)
553     tx = optax.chain(
554         optax.sgd(learning_rate=config.learning_rate, momentum=config.momentum),
555         optax.additive_weight_decay(weight_decay=config.weight_decay))
556     state = TrainState.create(apply_fn=model.apply, params=params, tx=tx)
557     return state
558
559
560 def compute_metrics(*, labels: Array, logits: Array) -> Metrics:
561     """Computes the metrics, summed across the batch if a batch is provided."""
562     if labels.ndim == 1: # Prevent the labels from broadcasting over the logits.
563         labels = jnp.expand_dims(labels, axis=1)
564     loss = sigmoid_cross_entropy_with_logits(labels=labels, logits=logits)
565     binary_predictions = (logits >= 0.)
566     binary_accuracy = jnp.equal(binary_predictions, labels)
567     return Metrics(
568         loss=jnp.sum(loss),
569         accuracy=jnp.sum(binary_accuracy),
570         count=logits.shape[0])
571
572
573 def model_from_config(config: ml_collections.ConfigDict):
574     """Builds a text classification model from a config."""
575     model = models.TextClassifier(
576         embedding_size=config.embedding_size,
577         hidden_size=config.hidden_size,
578         vocab_size=config.vocab_size,
579         output_size=config.output_size,
580         dropout_rate=config.dropout_rate,
581         word_dropout_rate=config.word_dropout_rate,
582         unk_idx=config.unk_idx)
583     return model
584
585
586 def train_step(
587     state: TrainState,
588     batch: Dict[str, Array],

```



```

589     rngs: Dict[str, Any],
590 ) -> Tuple[TrainState, Metrics]:
591     """Train for a single step."""
592     # Make sure to get a new RNG at every step.
593     step = state.step
594     rngs = {name: jax.random.fold_in(rng, step) for name, rng in rngs.items()}
595
596     def loss_fn(params):
597         variables = {'params': params}
598         logits = state.apply_fn(
599             variables, batch['token_ids'], batch['length'],
600             deterministic=False,
601             rngs=rngs)
602
603         labels = batch['label']
604         if labels.ndim == 1:
605             labels = jnp.expand_dims(labels, 1)
606         loss = jnp.mean(
607             sigmoid_cross_entropy_with_logits(labels=labels, logits=logits))
608         return loss, logits
609
610     grad_fn = jax.value_and_grad(loss_fn, has_aux=True)
611     value, grads = grad_fn(state.params)
612     (_, logits) = value
613
614     new_state = state.apply_gradients(grads=grads)
615     metrics = compute_metrics(labels=batch['label'], logits=logits)
616     return new_state, metrics
617
618 def eval_step(state: TrainState, batch: Dict[str, Array],
619             rngs: Dict[str, Any]) -> Metrics:
620     """Evaluate for a single step. Model should be in deterministic mode."""
621     variables = {'params': state.params}
622     logits = state.apply_fn(
623         variables, batch['token_ids'], batch['length'],
624         deterministic=True,
625         rngs=rngs)
626     metrics = compute_metrics(labels=batch['label'], logits=logits)
627     return metrics
628
629
630 def normalize_batch_metrics(
631     batch_metrics: Sequence[Metrics]) -> Metrics:
632     """Consolidates and normalizes a list of per-batch metrics dicts."""
633     # Here we sum the metrics that were already summed per batch.
634     total_loss = np.sum([metrics.loss for metrics in batch_metrics])
635     total_accuracy = np.sum([metrics.accuracy for metrics in batch_metrics])
636     total = np.sum([metrics.count for metrics in batch_metrics])
637     # Divide each metric by the total number of items in the data set.
638     return Metrics(
639         loss=total_loss.item() / total, accuracy=total_accuracy.item() / total)
640
641
642 def batch_to_numpy(batch: Dict[str, tf.Tensor]) -> Dict[str, Array]:
643     """Converts a batch with TF tensors to a batch of NumPy arrays."""
644     # _numpy() reuses memory, does not make a copy.
645     # pylint: disable=protected-access
646     return jax.tree_map(lambda x: x._numpy(), batch)
647
648
649 def evaluate_model(
650     eval_step_fn: Callable[..., Any],
651     state: TrainState,
652     batches: Union[Iterable[Example], tf.data.Dataset],
653     epoch: int,
654     rngs: Optional[Dict[str, Any]] = None
655 ) -> Metrics:

```

```

657 """Evaluate a model on a dataset."""
658 batch_metrics = []
659 for i, batch in enumerate(batches):
660     batch = batch_to_numpy(batch)
661     if rngs is not None: # New RNG for each step.
662         rngs = {name: jax.random.fold_in(rng, i) for name, rng in rngs.items()}
663
664     metrics = eval_step_fn(state, batch, rngs)
665     batch_metrics.append(metrics)
666
667 batch_metrics = jax.device_get(batch_metrics)
668 metrics = normalize_batch_metrics(batch_metrics)
669 logging.info('eval epoch %03d loss %.4f accuracy %.2f', epoch,
670             metrics.loss, metrics.accuracy * 100)
671 return metrics
672
673
674 def train_epoch(train_step_fn: Callable[..., Tuple[TrainState, Metrics]],
675                state: TrainState,
676                train_batches: tf.data.Dataset,
677                epoch: int,
678                rngs: Optional[Dict[str, Any]] = None
679                ) -> Tuple[TrainState, Metrics]:
680     """Train for a single epoch."""
681     batch_metrics = []
682     for batch in train_batches:
683         batch = batch_to_numpy(batch)
684         state, metrics = train_step_fn(state, batch, rngs)
685         batch_metrics.append(metrics)
686
687     # Compute the metrics for this epoch.
688     batch_metrics = jax.device_get(batch_metrics)
689     metrics = normalize_batch_metrics(batch_metrics)
690
691     logging.info('train epoch %03d loss %.4f accuracy %.2f', epoch,
692                 metrics.loss, metrics.accuracy * 100)
693
694     return state, metrics
695
696
697 def train_and_evaluate(config: ml_collections.ConfigDict,
698                       workdir: str) -> TrainState:
699     """Execute model training and evaluation loop.
700     Args:
701         config: Hyperparameter configuration for training and evaluation.
702         workdir: Directory where the tensorboard summaries are written to.
703     Returns:
704         The final train state that includes the trained parameters.
705     """
706     # Prepare datasets.
707     train_dataset = input_pipeline.TextDataset(
708         tfds_name='glue/sst2', split='train')
709     eval_dataset = input_pipeline.TextDataset(
710         tfds_name='glue/sst2', split='validation')
711     train_batches = train_dataset.get_bucketed_batches(
712         config.batch_size,
713         config.bucket_size,
714         max_input_length=config.max_input_length,
715         drop_remainder=True,
716         shuffle=True,
717         shuffle_seed=config.seed)
718     eval_batches = eval_dataset.get_batches(batch_size=config.batch_size)
719
720     # Keep track of vocab size in the config so that the embedder knows it.
721     config.vocab_size = len(train_dataset.vocab)
722
723     # Compile step functions.
724     train_step_fn = jax.jit(train_step)

```

```

725 eval_step_fn = jax.jit(eval_step)
726
727 # Create model and a state that contains the parameters.
728 rng = jax.random.PRNGKey(config.seed)
729 model = model_from_config(config)
730 state = create_train_state(rng, config, model)
731
732 summary_writer = tensorboard.SummaryWriter(workdir)
733 summary_writer.hparams(dict(config))
734
735 # Main training loop.
736 logging.info('Starting training...')
737 for epoch in range(1, config.num_epochs + 1):
738
739     # Train for one epoch.
740     rng, epoch_rng = jax.random.split(rng)
741     rngs = {'dropout': epoch_rng}
742     state, train_metrics = train_epoch(
743         train_step_fn, state, train_batches, epoch, rngs)
744
745     # Evaluate current model on the validation data.
746     eval_metrics = evaluate_model(eval_step_fn, state, eval_batches, epoch)
747
748     # Write metrics to TensorBoard.
749     summary_writer.scalar('train_loss', train_metrics.loss, epoch)
750     summary_writer.scalar(
751         'train_accuracy',
752         train_metrics.accuracy * 100,
753         epoch)
754     summary_writer.scalar('eval_loss', eval_metrics.loss, epoch)
755     summary_writer.scalar(
756         'eval_accuracy',
757         eval_metrics.accuracy * 100,
758         epoch)
759
760     summary_writer.flush()
761     return state
762
763     x = nn.Conv(features=32, kernel_size=(3, 3))(x)
764     x = nn.relu(x)
765     x = nn.avg_pool(x, window_shape=(2, 2), strides=(2, 2))
766     x = nn.Conv(features=64, kernel_size=(3, 3))(x)
767     x = nn.relu(x)
768     x = nn.avg_pool(x, window_shape=(2, 2), strides=(2, 2))
769     x = x.reshape((x.shape[0], -1)) # flatten
770     x = nn.Dense(features=256)(x)
771     x = nn.relu(x)
772     x = nn.Dense(features=10)(x)

```

```

1 # -*- coding: utf-8 -*-
2 """Text Analytics - Twitter Network - Stochastic Gradient Descent - Jax.ipynb
3
4 Automatically generated by Colaboratory.
5
6 Original file is located at
7     https://colab.research.google.com/drive/10J0bGhZzah8hBwv8cYwnbaENeagwOXgz
8 """
9
10 import jax.numpy as jnp
11 import numpy as np
12 from jax import grad, jit, vmap, pmap, random
13
14 """# Datatypes and Naming Conventions
15
16 ### Type Synonyms
17 """
18
19 from typing import Any
20 from typing import FrozenSet

```

```

21 from typing import List
22 import numpy.typing as npt
23
24 UserId = str
25 UserIdSet = FrozenSet[UserId]
26
27 Corpus = str
28 Token = str
29 Tokens = List[Token]
30 UniqueTokens = FrozenSet[Token]
31
32 Edge = Any
33 Edges = FrozenSet[Edge]
34
35 Graph = Any
36 EdgeWeight = int
37
38 Vec = any
39 VecInt = npt.NDArray[np.int_]
40 WordVec = VecInt
41
42 TwitterData = Any
43 TwitterDataP = Any
44
45 TokenCount = Any
46
47
48 UserCorpusMap = Any
49 UserTokensMap = Any
50 UserTokenCountMap = Any
51 UserWordVecMap = Any
52 Set = Any
53 IndexMap = Any
54 WordVecIndex = IndexMap
55 EdgeVecIndex = IndexMap
56
57 """
58 TwitterDataP =
59     [
60         {
61             user: UserId,
62             corpus: Tokens,
63             followers: [ UserId ],
64             followings: [ UserId ]
65         }
66     ]
67
68 Edge = frozenset(UserId, UserId)
69 Graph = Map<Edge, EdgeWeight>
70
71 TokenCount = Map<Token, Int>
72
73 UserCorpusMap = Map<UserId, Corpus>
74 UserTokensMap = Map<UserId, Tokens>
75 UserTokenCountMap = Map<UserId, TokenCount>
76 UserWordVecMap = Map<UserId, WordVec>
77
78 WordVecIndex = Map<Token, Integer>
79 EdgeVecIndex = Map<Edge, Integer>
80
81 IndexMap = Map<T, Integer>
82 """
83
84 # Functions
85
86 ## User Extraction from Data
87 """
88

```

```

89 def createUserTokensMap(data : TwitterDataP) -> UserTokensMap:
90     userTokensMap = {}
91     for user in data:
92         userTokensMap[user['user']] = user['corpus']
93     return userTokensMap
94
95 def getUserIdSet(userCorpusMap : UserCorpusMap) -> UserIdSet:
96     return frozenset(userCorpusMap.keys())
97
98 """## Graph Construction"""
99
100 from itertools import combinations
101
102 def mkEdge(u1 : UserId, u2 : UserId) -> Edge:
103     return frozenset({u1,u2})
104
105 def createEdges(userIdSet : UserIdSet) -> Edges:
106     edges = set({})
107     for (a,b) in combinations(userIdSet, 2):
108         edges.add(mkEdge(a, b))
109     return edges
110
111 def graphSet(edges : Edges, val : int) -> Graph:
112     graph = {}
113     for e in edges:
114         graph[e] = val
115     return graph
116
117 def graphZeroed(edges : Edges) -> Graph:
118     return graphSet(edges, 0)
119
120 def graphMergeGroundTruth(graph: Graph, data : TwitterDataP) -> Graph:
121     mergedGraph = graph.copy()
122     for d in data:
123         followed = set(d['followers'])
124         following = set(d['followings'])
125         u1 = d['user']
126
127         for u2 in followed:
128             mergedGraph[mkEdge(u1, u2)] = 0.5
129         for u2 in following:
130             mergedGraph[mkEdge(u1, u2)] = 0.5
131
132         twoWay = followed.intersection(following)
133         for u2 in twoWay:
134             mergedGraph[mkEdge(u1, u2)] = 1.0
135
136     return mergedGraph
137
138 def graphCos(edges: Edges, userWordVecMap : UserWordVecMap) -> Graph:
139     graph = {}
140     for u1, u2 in edges:
141         v = cos(userWordVecMap[u1], userWordVecMap[u2])
142         graph[mkEdge(u1,u2)] = v
143     return graph
144
145 def createEdgeVecIndex(edges: Edges) -> EdgeVecIndex:
146     return createWordVecIndex(edges)
147
148 def graphEdgeVec(graph : Graph, edgeVecIndex : EdgeVecIndex):
149     return tokenCountWordVec(graph, edgeVecIndex)
150
151 """## Corpus Processing and Transformations"""
152
153 from nltk import word_tokenize
154
155 def countTokens(tokens : Tokens) -> TokenCount:
156     tokenCount = {}

```

```

157     for token in tokens:
158         tokenCount[token] = tokenCount.get(token, 0) + 1
159     return tokenCount
160
161 def combineTokensList(tokens : List[Tokens]) -> Tokens:
162     return sum(tokens, [])
163
164 def tokenise(corpus : Corpus) -> Tokens:
165     return word_tokenize(corpus)
166
167 def leastOccuringTokens(countUpperBound : int, tokenCount : TokenCount) -> Tokens:
168     leastOccuring = []
169     for token, count in tokenCount.items():
170         if count <= countUpperBound:
171             leastOccuring.append(token)
172     return leastOccuring
173
174 def excludeTokens(excludedTokens : Tokens, tokenCount : TokenCount) -> TokenCount:
175     excludedCount = tokenCount.copy()
176     for token in excludedTokens:
177         if token in excludedCount:
178             del excludedCount[token]
179     return excludedCount
180
181 """## WordVector
182
183 """
184
185 def createWordVecIndex(allUniqueTokens : UniqueTokens) -> WordVecIndex:
186     wvIndex = {}
187     for wordIndex, word in enumerate(allUniqueTokens):
188         wvIndex[word] = wordIndex
189     return wvIndex
190
191 def createUserWordVecMap(userTokenCountMap : UserTokenCountMap, wvIndex :
192     WordVecIndex) -> UserWordVecMap:
193     userWvMap = userTokenCountMap.copy()
194     for userId, tokenCount in userTokenCountMap.items():
195         userWvMap[userId] = tokenCountWordVec(tokenCount, wvIndex)
196     return userWvMap
197
198 def tokenCountWordVec(tokenCount : TokenCount, wvIndex : WordVecIndex) -> WordVec:
199     wordVec = np.zeros(len(wvIndex))
200     for token, count in tokenCount.items():
201         if token in wvIndex:
202             wordIndex = wvIndex[token]
203             wordVec[wordIndex] = count
204     return wordVec
205
206 """## Similarity Measures"""
207
208 @jit
209 def cos(v1 : Vec, v2 : Vec) -> float:
210     return jnp.sum(v1 * v2) / (jnp.sqrt(jnp.sum(v1**2)) * (jnp.sqrt(jnp.sum(v2**2))) +
211         0.000001)
212
213 @jit
214 def cos2(v : Vec) -> float:
215     #print(v.shape)
216     #print(v)
217     v1 = v[0]
218     v2 = v[1]
219     r = jnp.sum(v1 * v2) / (jnp.sqrt(jnp.sum(v1**2)) * (jnp.sqrt(jnp.sum(v2**2))) +
220         0.000001)
221     #print(r)
222     return r
223
224 def npcoss2(v : Vec) -> float:

```

```

222     #print(v.shape)
223     #print(v)
224     v1 = v[0]
225     v2 = v[1]
226     r = np.sum(v1 * v2) / (np.sqrt(np.sum(v1**2)) * (np.sqrt(np.sum(v2**2))) +
227         0.000001)
228     #print(r)
229     return r
230
231 def euc2(v1 : Vec, v2 : Vec) -> float:
232     return jnp.sqrt(jnp.sum((v1 - v2)**2))
233
234 npcos2(np.array([[1, 1, 0],[0, 1, 0]]))
235
236 """# Loading Data & Processing Data
237
238 ## Loading and Counting
239
240 from google.colab import drive
241 drive.mount('/content/drive')
242
243 from google.colab import drive
244 drive.mount('/content/drive')
245
246 import json
247 with open('./drive/MyDrive/TA/data_pre.json', 'r') as f:
248     dataset = json.load(f)['dataset']
249 dataset[0].keys()
250
251 userTokensMap = createUserTokensMap(dataset)
252 print("Users in dataset: \t", len(userTokensMap))
253
254 combinedTokens = combineTokensList(list(userTokensMap.values()))
255 print("Total token count: \t", len(combinedTokens))
256
257 totalTokenCount = countTokens(combinedTokens)
258 print("Unique tokens: \t \t", len(totalTokenCount.keys()))
259
260 mapToTuples = lambda x: [(k, v) for k, v in x.items()]
261 sortByVal = lambda x: sorted(mapToTuples(x), reverse=True, key=lambda x: x[1])
262
263 print("Top 10 tokens: \t \t", sortByVal(totalTokenCount)[:10])
264 print("Bottom 10 tokens: \t", sortByVal(totalTokenCount)[::-1][:10])
265
266 """## Pruning Least Occuring Tokens"""
267
268 leastOccuring3 = leastOccuringTokens(3, totalTokenCount)
269 print("Unique tokens that occur 3 or less times: \t", len(leastOccuring3))
270 print(leastOccuring3[:5])
271
272 excludedCount3 = excludeTokens(leastOccuring3, totalTokenCount)
273 print("Unique tokens that occur more than 3 times: \t", len(excludedCount3))
274 print(mapToTuples(excludedCount3)[:5])
275
276 userTokenCountMap = userTokensMap.copy()
277 for user, tokens in userTokenCountMap.items():
278     userTokenCountMap[user] = excludeTokens(leastOccuring3, countTokens(tokens))
279
280 mapToTuples(userTokenCountMap)[0]
281
282 """# Constructing Graphs
283
284 ### Ground Truth Graph
285 """
286
287 userIdSet = getUserIdSet(userTokenCountMap)
288 edges = createEdges(userIdSet)

```

```

289 graphZero = graphZeroed(edges)
290 graphGroundTruth = graphMergeGroundTruth(graphZero, dataset)
291
292 print("Total connections strength: \t", sum(graphGroundTruth.values()))
293 print("Number of edges: \t \t", len(graphGroundTruth.keys()))
294
295 """WordVec Cosine Similarity Graph"""
296
297 wvIndex = createWordVecIndex(excludedCount3.keys())
298 userWordVecMap = createUserWordVecMap(userTokenCountMap, wvIndex)
299
300 print("Unique tokens a.k.a Vector length: \t", len(wvIndex))
301 print("Number of users: \t \t \t", len(userWordVecMap))
302 print("User wv length: \t \t \t", len(list(userWordVecMap.values())[0]))
303
304 graphCosine = graphCos(edges, userWordVecMap)
305
306 mapToTuples(graphCosine)[:10]
307
308 """# Analysing Data
309
310 We have 2 datasets:
311 - Dataset 1 : Retrieved by traversing followers on twitter.
312 - Dataset 2 : Retrived by querying a topic and gathering the users.
313
314 We have already pre-processed dataset, now preprocess dataset 2.
315 We also want to learn about each of the datasets.
316 We can first investigate the links:
317 - Looking at total number of users in each dataset.
318 - Counting the percentage of edges which are 0.5 strength on the ground truth graph.
319 - Counting the percentage of edges which are 1 strength on the ground truth graph.
320
321 We can also investigate the text content of the whole corpus:
322 - How many unique tokens are there in each dataset?
323 - What is the mean and variance of the number of unique tokens for a single user in
  each dataset?
324 - What is the distribution of the tokens for the each dataset?
325
326 Yash has plotted the plot of token occurance for dataset 1, do the same for dataset
  2.
327
328 ### Links in Each Dataset
329 """
330
331 numberOfUsersDataset1 = len(userTokenCountMap)
332 print(numberOfUsersDataset1)
333 # repeat for dataset 2
334
335 print(mapToTuples(graphGroundTruth)[:5])
336
337 def countValOccurance(vals, val):
338     count = 0
339     for v in vals:
340         if val == v:
341             count += 1
342     return count
343
344 totalEdges = len(graphGroundTruth)
345 print(totalEdges)
346 halfLinkCount = countValOccurance(graphGroundTruth.values(), 0.5)
347 fullLinkCount = countValOccurance(graphGroundTruth.values(), 1)
348 print(halfLinkCount/totalEdges)
349 print(fullLinkCount/totalEdges)
350
351 # repeat for dataset 2
352
353 """### Text Content of Dataset"""
354

```



```

355 # can use userTokenCountMap and excludedCount3/
356 print(mapToTuples(userTokenCountMap)[0]) # is a map from user to token count for that
      user
357 # excludedCount3 has all the tokens counted, except maybe do it for full token list (
      see the code above that constructs this variable)
358
359
360
361 """"# Comparing Graphs with Similarity Measures""""
362
363 edgeVecIndex = createEdgeVecIndex(edges)
364 cosineEv = graphEdgeVec(graphCosine, edgeVecIndex)
365 groundTruthEv = graphEdgeVec(graphGroundTruth, edgeVecIndex)
366 zeroedEv = graphEdgeVec(graphZero, edgeVecIndex)
367 meanEv = graphEdgeVec(graphSet(edges, np.mean(groundTruthEv)), edgeVecIndex)
368
369 print("Ground Truth vs Cosine Similarity")
370 print("Cosine Similarity: \t", cos(cosineEv, groundTruthEv))
371 print("Euclidean Distance: \t", euc2(cosineEv, groundTruthEv))
372
373 print("\nIdentities")
374 print("Cosine Similarity: \t", cos(cosineEv, cosineEv))
375 print("Euclidean Distance: \t", euc2(cosineEv, cosineEv))
376
377 print("\nZeroed")
378 print("Cosine Similarity: \t", cos(groundTruthEv, zeroedEv))
379 print("Euclidean Distance: \t", euc2(groundTruthEv, zeroedEv))
380
381 print("\nMean")
382 print("Cosine Similarity: \t", cos(groundTruthEv, meanEv))
383 print("Euclidean Distance: \t", euc2(groundTruthEv, meanEv))
384
385 """"# Linear Regression
386
387 2 sets of indices on data.
388 $m$ = Subset of $M$
389 $k$ = subset of $K$
390
391 $ $ = Set of all edges
392
393
394 ...
395 edges = [(u1, u2), (u1, u3), ...]
396 ...
397
398
399
400 $K$ = Set of all word vector indices
401
402 $N_{mk}$ = Edges and associated word vector subset similarity.
403
404 $x$ = weights on wordvector indices
405
406 $N_{mk}$ =
407
408 ...
409 [
410     (u1, u2) = cosine(u1_kx, u2_kx)
411     (u1, u3) = cosine(u1_kx, u3_kx)
412     .
413     .
414     .
415 ]
416 u1_kx = [ (k_i * x_i * u1_i) .... ]
417 ...
418 if $i$ in set $k$ then $k_i=1$ otherwise $k_i = 0$
419
420 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.
421
422 Exponential nature of edges.
423 Exponential nature of ngrams.
424
425 ### Preparing the Data
426 """
427
428 def constructSetIndex(setToIndex : Set) -> IndexMap:
429     return createWordVecIndex(setToIndex)
430
431 def userIdxToWvVector(userWordVecMap, userIndexMap):
432     wordVecLength = len(list(userWordVecMap.values())[0])
433     dataWV = np.zeros((len(userIndexMap), wordVecLength))
434     for user, wv in userWordVecMap.items():
435         idx = userIndexMap[user]
436         dataWV[idx] = wv
437     return dataWV
438
439 def constructM(userIndexMap, edgeIndexMap):
440     dataM = np.zeros((len(edgeIndexMap), 2))
441     for (u1, u2), idx in edgeIndexMap.items():
442         dataM[idx] = np.array([ np.array(userIndexMap[u1]), np.array(userIndexMap[u2]) ])
443     return dataM
444
445 edgeIndexMap = constructSetIndex(edges)
446 userIndexMap = constructSetIndex(list(userWordVecMap.keys()))
447
448 npGT = graphEdgeVec(graphGroundTruth, edgeIndexMap)
449 npWV = userIdxToWvVector(userWordVecMap, userIndexMap).astype(int)
450 npM = constructM(userIndexMap, edgeIndexMap).astype(int)
451
452 npR = npWV[npM]
453
454 dataGT = jnp.array(npGT)
455 dataWV = jnp.array(npWV)
456 dataM = jnp.array(npM)
457 dataR = jnp.array(npR)
458
459 print("dataM", dataM.shape, dataM[:4])
460 print("dataWV", dataWV.shape, dataWV[:4])
461 print("dataGT", dataGT.shape, dataGT[:4])
462 print("dataR", dataR.shape, dataR[:4])
463
464 a = np.array([
465     [[1,2,3],[4,5,6]],
466     [[7,8,9],[10,11,12]]
467 ])
468 b = np.array([2, 0, 1])
469 print(a.shape)
470 a*b
471
472 """### Indexing and Batching the Data"""
473
474 K = jnp.arange(len(wvIndex))
475 M = jnp.arange(len(edgeVecIndex))
476
477 def indices(key, bm, bk):
478     m = random.choice(key, M, [bm])
479     k = random.choice(key, K, [bk])
480     return m, k
481
482 """### Loss Function"""
483
484 @jit
485 def loss(x, m, k):
486     a1 = dataM.at[m].get()
487     a2 = dataWV.at[a1].get()

```

```

488
489     a3 = a2.T.at[k].get().T
490     x1 = x.at[k].get()
491
492     a4 = a3 * x1
493     a5 = vcos2(a4)
494     return cos(a5, dataGT[m])
495
496 @jit
497 def loss_mse(x, m, k):
498     a1 = dataM.at[m].get()
499     a2 = dataWV.at[a1].get()
500
501     a3 = a2.T.at[k].get().T
502     x1 = x.at[k].get()
503
504     a4 = a3 * x1
505     a5 = vmse(a4)
506     return mse(a5, dataGT[m])
507
508 @jit
509 def loss_unbatched(x):
510     a1 = dataR * x
511     a2 = vmap(cos2)(a1)
512     return cos(a2, dataGT)
513
514 @jit
515 def cos(v1: Vec, v2: Vec) -> float:
516     return jnp.sum(v1 * v2) / (jnp.sqrt(jnp.sum(v1**2)) * (jnp.sqrt(jnp.sum(v2**2))) +
517                                0.000001)
518
519 @jit
520 def cos2(v: Vec) -> float:
521     v1 = v[0]
522     v2 = v[1]
523     r = jnp.sum(v1 * v2) / (jnp.sqrt(jnp.sum(v1**2)) * (jnp.sqrt(jnp.sum(v2**2))) +
524                             0.000001)
525     return r
526
527 @jit
528 def euc2(v1: Vec, v2: Vec) -> float:
529     return jnp.sqrt(jnp.sum((v1 - v2)**2))
530
531 @jit
532 def euc2(v1: Vec, v2: Vec) -> float:
533     return jnp.sqrt(jnp.sum((v1 - v2)**2))
534
535 @jit
536 def mse2(v) -> float:
537     v1 = v[0]
538     v2 = v[1]
539     return jnp.sum((v1 - v2)**2) / jnp.size(v1)
540
541 @jit
542 def mse(v1, v2) -> float:
543     return jnp.sum((v1 - v2)**2) / jnp.size(v1)
544
545 vcos2 = vmap(cos2)
546 vmse = vmap(mse2)
547
548 loss_unbatched(x)
549
550 # Commented out IPython magic to ensure Python compatibility.
551 key = random.PRNGKey(0)
552 x = jnp.ones(len(K))
553 m, k = indices(key, len(M), len(K))
554 m2, k2 = indices(key, 128, len(K))

```

```

554 m3, k3 = indices(key, 128, 128)
555 # %timeit loss(x, m, k)
556 # %timeit loss(x, m2, k2)
557 # %timeit loss(x, m3, k3)
558
559 print(len(M))
560 print(len(K))
561
562 """#### On CPU
563 ##### 43660 Edges, 3933 Tokens
564 - 1 loop, best of 5: 12.5 s per loop
565
566 ##### 128 Edges, 3933 Tokens
567 - 1 loop, best of 5: 12 ms per loop
568
569 ##### 128 Edges, 128 Tokens
570 - 1 loop, best of 5: 196 s per loop
571
572 #### On GPU
573 ##### 43660 Edges, 3933 Tokens
574 - 100 loops, best of 5: 19.5 ms per loop
575
576 ##### 128 Edges, 3933 Tokens
577 - 100 loops, best of 5: 124 s per loop
578
579 ##### 128 Edges, 128 Tokens
580 - 10000 loops, best of 5: 44.7 s per loop
581
582 Looks like gradients don't work with cosine similarity.
583 Gradient immediately returns NaN
584
585 ### Stochastic Gradient Descent
586 """
587
588 def monitor(i, iters, p, x, F, F2):
589     p2 = round((i/iters) * 100)
590     if p2 > p:
591         p = p2; print(p, "%")
592         F += [loss(x, M, K)]
593         F2 += [loss_mse(x, M, K)]
594     return p, F, F2
595
596 def sgd(x0, alpha, iters, bm, bk, rngKey):
597     x = x0
598
599     F = []; F2 = []; p=0
600
601     for i in range(iters):
602         p, F, F2 = monitor(i, iters, p, x, F, F2)
603
604         key, rngKey = random.split(rngKey)
605         m, k = indices(key, bm, bk)
606
607         g = (grad(loss_mse)(x, m, k))
608         x = x - alpha * g
609     return x, F, F2
610
611 def adam(x0, alpha, iters, bm, bk, b1, b2, rngKey):
612     F = []; F2 = []; p=0
613
614     x = x0
615     am = jnp.zeros(len(x0)) ; av = jnp.zeros(len(x0)) ; ak = 1
616     for i in range(iters):
617         p, F, F2 = monitor(i, iters, p, x, F, F2)
618
619         key, rngKey = random.split(rngKey)
620         m, k = indices(key, bm, bk)
621

```

```

622     # might need to skip the weight updates and history record when k decides to turn
        off some of the weights
623     # below part of adam can be jit'ed, need to extract it into separate function and
        carry over the context
624     g = (grad(loss_mse)(x, m, k))
625     am = b1 * am + (1 - b1) * g
626     av = b2 * av + (1 - b2) * g**2
627     mhat = (am / (1 - b1**ak))
628     vhat = (av / (1 - b2**ak))
629     x = x - alpha * (mhat / (jnp.sqrt(vhat) + 0.00001))
630     ak = ak + 1
631
632     return x, F, F2
633
634     """### Running Code"""
635
636     key = random.PRNGKey(0)
637     x = jnp.ones(len(K))
638     # r, F, F2 = sgd(x, 0.1, 100_000, 1024, 2048, key)
639
640     # ar, aF, aF2 = adam(x, 0.001, 100_000, 1024, 2048, b1=0.9, b2=0.999, rngKey=key)
641
642     print("\n Cosine Similarity")
643     print("Word Count: \t", loss(x, m, k))
644     print("SGD: \t \t", loss(r, m, k))
645     print("Adam: \t \t", loss(ar, m, k))
646     print("\n Mean Squared Error")
647     print("Word Count: \t", loss_mse(x, m, k))
648     print("SGD: \t \t", loss_mse(r, m, k))
649     print("Adam: \t \t", loss_mse(ar, m, k))
650
651     import matplotlib.pyplot as plt
652     plt.plot(range(len(F)), F, label="SGD Constant")
653     plt.plot(range(len(aF)), aF, label="SGD Adam")
654     plt.title("Cosine Similarity vs. %Iters \n 100,000 iterations, 1024 Edge Batch, 2048
        Word Batch")
655     plt.legend()
656
657     plt.semilogy(range(len(F2)), F2, label="SGD Constant")
658     plt.semilogy(range(len(aF2)), aF2, label="SGD Adam")
659     plt.title("MSE vs. %Iters \n 100,000 iterations, 1024 Edge Batch, 2048 Word Batch")
660     plt.legend()
661
662     r[:20]
663
664     def invertIndexMap(m):
665         m2 = {}
666         for v1, v2 in m.items():
667             m2[v2] = v1
668         return m2
669
670     print(mapToTuples(wvIndex)[:10])
671     print(mapToTuples(invertIndexMap(wvIndex))[:10])
672
673     def weightsToWordWeightMap(indexToWord, weights):
674         wordWeightMap = {}
675         for idx, weight in enumerate(weights):
676             wordWeightMap[indexToWord[idx]] = weight
677         return wordWeightMap
678
679     indexToWord = invertIndexMap(wvIndex)
680     wordWeightMap = weightsToWordWeightMap(indexToWord, np.array(r))
681     wordWeightMapAdam = weightsToWordWeightMap(indexToWord, np.array(ar))
682
683     sortByVal(wordWeightMap)[:15]
684
685     sortByVal(wordWeightMap)[::-1][:10]
686

```

```

687 sortByVal(wordWeightMapAdam)[:15]
688
689 sortByVal(wordWeightMapAdam)[:,-1][:10]
690
691 """## With Optax
692 - SGD (Batching?)
693   - Looks like SGD is actually just GD
694   - Batching implemented by ourselves
695 - Loss Function?
696   - https://optax.readthedocs.io/en/latest/api.html#common-losses
697     - l2 aka means squared error
698     - cosine distance
699
700
701 - So perhaps can just use the loss functions
702
703
704 - Though will still need to use Adam
705   - So perhaps can use SGD since will need to fit into the framework anyway
706 """
707
708 !pip install optax
709
710
711
712 from optax import cosine_distance, cosine_similarity, l2_loss
713 from jax import value_and_grad
714
715 M = jnp.arange(len(edgeVecIndex))
716 from sklearn.model_selection import train_test_split
717 M_train, M_test = train_test_split(M)
718
719 print(M_train.shape)
720
721 print(M_test.shape)
722 print(len(K))
723
724 vcos = vmap(jit(lambda x: cosine_distance(x[0], x[1], 1e-9)))
725 vl2 = vmap(jit(lambda x: jnp.mean(l2_loss(x.at[0].get(), x.at[1].get()))))
726
727 @jit
728 def loss(x, m):
729     a1 = dataM.at[m].get()
730     a2 = dataWV.at[a1].get()
731
732     a4 = a2 * x
733     a5 = vl2(a4)
734     return jnp.mean(l2_loss(a5, dataGT[m]))
735
736 WV = dataWV.at[dataM.at[M].get()].get()
737
738 @jit
739 def loss_cos2(x, m):
740     return cosine_distance(vcos(WV[m] * x), dataGT[m] * 0.001, 1e-9)
741
742 @jit
743 def loss_cos(x, m):
744     a1 = dataM.at[m].get()
745     a2 = dataWV.at[a1].get()
746     a4 = a2 * x * 0.001
747     a5 = vcos(a4)
748     return cosine_distance(a5, dataGT[m] * 0.001, 1e-9)
749
750
751 # prob better way to do this using optax/jax ecosystem
752 def mkMonitor(iters, ltrain, ltest):
753     def fn(state, x):
754         p = state["p"]

```

```

755     i = state["i"]
756     Ftrain = state["Ftrain"]
757     Ftest = state["Ftest"]
758
759     p2 = round((i/iters) * 100)
760     if p2 > p:
761         state["p"] = p2;
762         print(p2, "%")
763         state["Ftrain"] = Ftrain + [ltrain(x)]
764         state["Ftest"] = Ftest + [ltest(x)]
765
766     state["i"] = i + 1
767     return state
768
769     state = {}
770     state["p"] = -1
771     state["Ftrain"] = []
772     state["Ftest"] = []
773     state["i"] = 0
774
775     return state, fn
776
777 def sgd(x0, alpha, iters, b, rngKey):
778     x = x0
779
780     state, monitorIter = mkMonitor(iters,
781                                     lambda x: loss_cos(x, M_train),
782                                     lambda x: loss_cos(x, M_test))
783
784     for i in range(iters):
785         state = monitorIter(state, x)
786
787         key, rngKey = random.split(rngKey)
788         m = random.choice(key, M_train, [b])
789
790         g = (grad(loss_cos)(x, m))
791         x = x - alpha * g
792     return x, state["Ftrain"], state["Ftest"]
793
794
795 def adam(x0, alpha, iters, b1, b2, b, rngKey):
796     x = x0
797
798     state, monitorIter = mkMonitor(iters,
799                                     lambda x: loss_cos(x, M_train),
800                                     lambda x: loss_cos(x, M_test))
801     am = jnp.zeros(len(x0)) ; av = jnp.zeros(len(x0)) ; ak = 1
802     for i in range(iters):
803         state = monitorIter(state, x)
804         key, rngKey = random.split(rngKey)
805         m = random.choice(key, M_train, [b])
806
807         g = (grad(loss_cos)(x, m))
808         am = b1 * am + (1 - b1) * g
809         av = b2 * av + (1 - b2) * g**2
810         mhat = (am / (1 - b1**ak))
811         vhat = (av / (1 - b2**ak))
812         x = x - alpha * (mhat / (jnp.sqrt(vhat) + 0.00001))
813         ak = ak + 1
814
815     return x, state["Ftrain"], state["Ftest"]
816
817 @jit
818 def cos(x):
819     a1 = dataM.at[M].get()
820     a2 = dataWV.at[a1].get()
821     a4 = a2 * x * 0.001
822     a5 = vcos(a4)

```

```

823     return cosine_distance(a5, dataGT[M] * 0.001, 1e-9)
824
825 key = random.PRNGKey(0)
826 x = jnp.ones(len(K))
827 r, F = sgd(x, 500, 1000, 128, key)
828
829 F[-1]
830
831 print(cos(x))
832 print(cos(r))
833
834 import matplotlib.pyplot as plt
835 plt.semilogy(range(len(F)), F, label="SGD Constant")
836
837 r2, F2 = adam(x, 1, 1_000, 0.99, 0.9, 128, key)
838
839 plt.semilogy(range(len(F)), F, label="Constant")
840 plt.semilogy(range(len(F2)), F2, label="Adam ")
841
842 def global_random_search(intervals, N, f):
843     lowest = None
844     l = [l for l, u in intervals]
845     u = [u for l, u in intervals]
846
847     tries = []
848
849     for s in range(N):
850         r = np.random.uniform(l, u)
851         print("\niteration:", s, "trying out:", r)
852         v = f(*r)
853         print("got", v)
854         if (not lowest) or lowest[0] > v:
855             lowest = (v.copy(), r.copy())
856         tries += [(v, r.copy())]
857     return tries, lowest
858
859 def run_multiple(runs, f):
860     loss_histories = []
861     test_losses = []
862
863     seed = random.PRNGKey(0)
864     seed, subseed = random.split(seed)
865
866     for r in range(runs):
867         print("Run number:", r)
868         loss_history = f(subseed)
869         seed, subseed = random.split(seed)
870         loss_histories += [loss_history]
871     return loss_histories
872
873 iters = 3_000
874
875 def f_sgd(learning_rate, batch_size, rngkey):
876     r, F = sgd(x, learning_rate, iters, round(batch_size), rngkey)
877     return F
878
879 def f_sgd_key(learning_rate, batch_size):
880     rngkey = random.PRNGKey(0)
881     return f_sgd(learning_rate, round(batch_size), rngkey)[-1]
882
883 sgd_optimal_alpha = 1000
884 sgd_optimal_batch = 2000
885
886 sgd_default_alpha = 0.1
887 sgd_default_batch = 128
888
889 adam_optimal_alpha = 1.9
890 adam_optimal_b1 = 0.869

```



```

891 adam_optimal_b2 = 0.662
892 adam_optimal_batch = 960
893
894 adam_default_alpha = 0.01
895 adam_default_b1 = 0.9
896 adam_default_b2 = 0.999
897 adam_default_batch = 128
898
899 def f_sgd_optimal(seed):
900     return f_sgd(sgd_optimal_alpha, sgd_optimal_batch, seed)
901
902 def f_sgd_default(seed):
903     return f_sgd(sgd_default_alpha, sgd_default_batch, seed)
904
905 def f_adam(learning_rate, b1, b2, batch_size, rngkey):
906     r, F = adam(x, learning_rate, iters, b1, b2, round(batch_size), rngkey)
907     return F
908
909 def f_adam_key(learning_rate, b1, b2, batch_size):
910     rngkey = random.PRNGKey(0)
911     return f_adam(learning_rate, b1, b2, round(batch_size), rngkey)[-1]
912
913 def f_adam_optimal(seed):
914     return f_adam(adam_optimal_alpha, adam_optimal_b1, adam_optimal_b2,
915                   adam_optimal_batch, seed)
916
917 def f_adam_default(seed):
918     return f_adam(adam_default_alpha, adam_default_b1, adam_default_b2,
919                   adam_default_batch, seed)
920
921 seed = random.PRNGKey(0)
922 xar, Faaa, Faaatest = adam(x, adam_optimal_alpha, 100_000, adam_optimal_b1,
923                             adam_optimal_b2, adam_optimal_batch, seed)
924
925 xsr, Fsss, Fsssatest = sgd(x, sgd_optimal_alpha, 100_000, sgd_optimal_batch, seed)
926
927 print("Baseline")
928 print("Test:\t", loss_cos(x, M_test))
929 print("Train:\t", loss_cos(x, M_train))
930 print("Adam")
931 print("Test:\t", loss_cos(xar, M_test))
932 print("Train:\t", loss_cos(xar, M_train))
933 print("Constant")
934 print("Test:\t", loss_cos(xsr, M_test))
935 print("Train:\t", loss_cos(xsr, M_train))
936
937 def compare_tt(F, F2, title="Title", F1="r1", F2l="r2"):
938     xs = range(len(F))
939     plt.semilogy(xs, F, color='#2e6fd9bb', label=F1)
940
941     xlim = plt.xlim()
942     ylim = plt.ylim()
943     plt.semilogy(xs, F2, color='#ff9c24bb', label=F2l)
944
945     plt.xlim(xlim)
946     plt.ylim(ylim)
947
948     plt.title(title)
949     plt.legend()
950
951     plt.xlabel(r'%run')
952     plt.ylabel(r'loss')
953
954 compare_tt(Fsss, Fsssatest, "SGD Optimal", "Train", "Test")
955 # optimal parameters were picked with train results, perhaps would expect something
956     different with test evaluation

```

```

955 compare_tt(Faaa, Faaatest, "Adam Optimal", "Train", "Test")
956
957 print(M.shape)
958 print(M_test.shape)
959 print(M_train.shape)
960
961 psgd = [(1, 1000), (12, 2048)]
962 lsgd, lowsgd = global_random_search(psgd, 20, f_sgd_key)
963
964 lowsgd
965
966 (array(0.91140574, dtype=float32), array([ 922.32970583, 2032.0561759 ]))
967
968 ys1 = 1 - np.array([x for x, (_, _) in lsgd])
969 xs2 = [y for _, (x, y) in lsgd]
970 xs1 = [x for _, (x, y) in lsgd]
971 import matplotlib.pyplot as plt
972
973 #plt.scatter(xs, ys1)
974 ax = plt.axes(projection='3d')
975 ax.scatter3D(xs1, xs2, ys1, c=ys1, cmap='Greens');
976 ax.view_init(30, 20)
977
978 plt.scatter(xs1, ys1)
979
980 plt.scatter(xs2, ys1)
981
982 padam = [(0.1, 100), (0.5,0.99), (0.5,0.99), (12, 2049)]
983 ladam, lowadam = global_random_search(padam, 20, f_adam_key)
984
985 lowadam
986
987 (array(0.9106287, dtype=float32),
988  array([1.92316220e+00, 8.68884077e-01, 6.62402454e-01, 9.57793168e+02]))
989
990 lha= run_multiple(20, f_adam_optimal)
991
992 lhad= run_multiple(20, f_adam_default)
993
994 lho = run_multiple(20, f_sgd_optimal)
995
996 lhod = run_multiple(20, f_sgd_default)
997
998 """- global random search and plotting parameters
999 - optimal and default
1000 - split data set into test and train
1001 """
1002
1003 sgd_default_loss_histories = lhod
1004 sgd_optimal_loss_histories = lho
1005 adam_default_loss_histories = lhad
1006 adam_optimal_loss_histories = lha
1007
1008 import pickle
1009
1010 mlruns = {
1011     "sgd_default_loss_histories": sgd_default_loss_histories,
1012     # "sgd_default_test_losses": sgd_default_test_losses,
1013     "sgd_optimal_loss_histories": sgd_optimal_loss_histories,
1014     # "sgd_optimal_test_losses": sgd_optimal_test_losses,
1015
1016     "adam_default_loss_histories": adam_default_loss_histories,
1017     # "adam_default_test_losses": adam_default_test_losses,
1018     "adam_optimal_loss_histories": adam_optimal_loss_histories,
1019     # "adam_optimal_test_losses": adam_optimal_test_losses
1020 }
1021
1022 pickle.dump(mlruns, open("./drive/MyDrive/TA/mlruns.p", "wb"))

```

```

1023
1024 import pickle
1025 mlruns_l = pickle.load(open( "./drive/MyDrive/TA/mlruns.p", "rb" ))
1026
1027 def plot_history(losses):
1028     'losses :: [[float]], ith element is loss vs iteration of ith run of the SGD'
1029     losses = np.array(losses)
1030     average_on_iter_i = np.mean(losses, axis=0)
1031     min_on_iter_i = np.minimum.reduce(losses)
1032     max_on_iter_i = np.maximum.reduce(losses)
1033     x = range(len(average_on_iter_i))
1034     plt.plot(x, average_on_iter_i, 'k-')
1035     plt.fill_between(x, min_on_iter_i, max_on_iter_i)
1036
1037 def avg_max_min(loss_histories):
1038     average_on_iter_i = np.mean(loss_histories, axis=0)
1039     min_on_iter_i = np.minimum.reduce(loss_histories)
1040     max_on_iter_i = np.maximum.reduce(loss_histories)
1041     return average_on_iter_i, min_on_iter_i, max_on_iter_i
1042
1043 def compare_sgd(r1, r2, title="Title", r1l="r1", r2l="r2"):
1044     r1 = np.array(r1) ; r2 = np.array(r2)
1045     xr = r1.shape[1] ; xr2 = r2.shape[1]
1046     x1 = np.linspace(0, 100, xr)
1047     x2 = np.linspace(0, 100, xr2)
1048     a1, l1, h1 = avg_max_min(r1)
1049     a2, l2, h2 = avg_max_min(r2)
1050
1051     plt.semilogy(x1, a1, color='#2e6fd9bb', label=r1l)
1052     plt.fill_between(x1, l1, h1, color="#3d84f588")
1053     xlim = plt.xlim()
1054     ylim = plt.ylim()
1055     plt.semilogy(x2, a2, color='#ff9c24bb', label=r2l)
1056     plt.fill_between(x2, l2, h2, color='#ffc37088')
1057     plt.xlim(xlim)
1058     plt.ylim(ylim)
1059     plt.title(title)
1060     plt.legend()
1061
1062     plt.xlabel(r'%run')
1063     plt.ylabel(r'loss')
1064
1065 import matplotlib.pyplot as plt
1066
1067 r1 = np.array(mlruns_l['sgd_default_loss_histories'])
1068 r2 = np.array(mlruns_l['sgd_optimal_loss_histories'])
1069
1070 compare_sgd(r2, r1, title="Constant: Default vs. Optimal", r1l="Optimal", r2l="
    Default")
1071
1072 r1 = np.array(mlruns_l['adam_default_loss_histories'])
1073 r2 = np.array(mlruns_l['adam_optimal_loss_histories'])
1074
1075 compare_sgd(r1=r2, r2=r1, title="Adam: Default vs. Optimal", r1l="Optimal", r2l="
    Default")
1076
1077 r1 = np.array(mlruns_l['sgd_optimal_loss_histories'])
1078 r2 = np.array(mlruns_l['adam_optimal_loss_histories'])
1079 compare_sgd(r1=r1, r2=r2, title="Optimal Constant vs. Optimal Adam", r1l="Constant",
    r2l="Adam")
1080
1081 r1 = np.array(mlruns_l['sgd_default_loss_histories'])
1082 r2 = np.array(mlruns_l['adam_default_loss_histories'])
1083 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://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/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://optax.readthedocs.io/en/latest/api.html#optax.softmax_cross_entropy)
- <https://jax.readthedocs.io/en/latest/>