MLP MNIST

This notebook trains a simple Multilayer Perceptron (MLP) classifier for hand-written digit recognition (MNIST dataset).

from typing import Sequence

from flax import linen as nn

import jax

import jax.numpy as jnp

import optax

import numpy as np

import tensorflow as tf

import tensorflow\_datasets as tfds

# @markdown The learning rate for the optimizer:

LEARNING\_RATE = 0.002 # @param{type:"number"}

# @markdown Number of samples in each batch:

BATCH\_SIZE = 128 # @param{type:"integer"}

# @markdown Total number of epochs to train for:

N\_EPOCHS = 1 # @param{type:"integer"}

MNIST is a dataset of 28x28 images with 1 channel. We now load the dataset using tensorflow\_datasets, apply min-max normalization to the images, shuffle the data in the train set and create batches of size BATCH\_SIZE.

(train\_loader, test\_loader), info = tfds.load(

"mnist", split=["train", "test"], as\_supervised=True, with\_info=True

)

min\_max\_rgb = lambda image, label: (tf.cast(image, tf.float32) / 255., label)

train\_loader = train\_loader.map(min\_max\_rgb)

test\_loader = test\_loader.map(min\_max\_rgb)

NUM\_CLASSES = info.features["label"].num\_classes

IMG\_SIZE = info.features["image"].shape

train\_loader\_batched = train\_loader.shuffle(

buffer\_size=10\_000, reshuffle\_each\_iteration=True

).batch(BATCH\_SIZE, drop\_remainder=True)

test\_loader\_batched = test\_loader.batch(BATCH\_SIZE, drop\_remainder=True)

The data is ready! Next let's define a model. Optax is agnostic to which (if any) neural network library is used. Here we use Flax to implement a simple MLP.

class MLP(nn.Module):

"""A simple multilayer perceptron model for image classification."""

hidden\_sizes: Sequence[int] = (1000, 1000)

@nn.compact

def \_\_call\_\_(self, x):

# Flattens images in the batch.

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

x = nn.Dense(features=self.hidden\_sizes[0])(x)

x = nn.relu(x)

x = nn.Dense(features=self.hidden\_sizes[1])(x)

x = nn.relu(x)

x = nn.Dense(features=NUM\_CLASSES)(x)

return x

net = MLP()

@jax.jit

def predict(params, inputs):

return net.apply({"params": params}, inputs)

@jax.jit

def loss\_accuracy(params, data):

"""Computes loss and accuracy over a mini-batch.

Args:

params: parameters of the model.

bn\_params: state of the model.

data: tuple of (inputs, labels).

is\_training: if true, uses train mode, otherwise uses eval mode.

Returns:

loss: float

"""

inputs, labels = data

logits = predict(params, inputs)

loss = optax.softmax\_cross\_entropy\_with\_integer\_labels(

logits=logits, labels=labels

).mean()

accuracy = jnp.mean(jnp.argmax(logits, axis=-1) == labels)

return loss, {"accuracy": accuracy}

@jax.jit

def update\_model(state, grads):

return state.apply\_gradients(grads=grads)

Next we need to initialize network parameters and solver state. We also define a convenience function dataset\_stats that we'll call once per epoch to collect the loss and accuracy of our solver over the test set.

solver = optax.adam(LEARNING\_RATE)

rng = jax.random.PRNGKey(0)

dummy\_data = jnp.ones((1,) + IMG\_SIZE, dtype=jnp.float32)

params = net.init({"params": rng}, dummy\_data)["params"]

solver\_state = solver.init(params)

def dataset\_stats(params, data\_loader):

"""Computes loss and accuracy over the dataset `data\_loader`."""

all\_accuracy = []

all\_loss = []

for batch in data\_loader.as\_numpy\_iterator():

batch\_loss, batch\_aux = loss\_accuracy(params, batch)

all\_loss.append(batch\_loss)

all\_accuracy.append(batch\_aux["accuracy"])

return {"loss": np.mean(all\_loss), "accuracy": np.mean(all\_accuracy)}

Finally, we do the actual training. The next cell train the model for N\_EPOCHS. Within each epoch we iterate over the batched loader train\_loader\_batched, and once per epoch we also compute the test set accuracy and loss.

train\_accuracy = []

train\_losses = []

# Computes test set accuracy at initialization.

test\_stats = dataset\_stats(params, test\_loader\_batched)

test\_accuracy = [test\_stats["accuracy"]]

test\_losses = [test\_stats["loss"]]

@jax.jit

def train\_step(params, solver\_state, batch):

# Performs a one step update.

(loss, aux), grad = jax.value\_and\_grad(loss\_accuracy, has\_aux=True)(

params, batch

)

updates, solver\_state = solver.update(grad, solver\_state, params)

params = optax.apply\_updates(params, updates)

return params, solver\_state, loss, aux

for epoch in range(N\_EPOCHS):

train\_accuracy\_epoch = []

train\_losses\_epoch = []

for step, train\_batch in enumerate(train\_loader\_batched.as\_numpy\_iterator()):

params, solver\_state, train\_loss, train\_aux = train\_step(

params, solver\_state, train\_batch

)

train\_accuracy\_epoch.append(train\_aux["accuracy"])

train\_losses\_epoch.append(train\_loss)

if step % 20 == 0:

print(

f"step {step}, train loss: {train\_loss:.2e}, train accuracy:"

f" {train\_aux['accuracy']:.2f}"

)

test\_stats = dataset\_stats(params, test\_loader\_batched)

test\_accuracy.append(test\_stats["accuracy"])

test\_losses.append(test\_stats["loss"])

train\_accuracy.append(np.mean(train\_accuracy\_epoch))

train\_losses.append(np.mean(train\_losses\_epoch))

f"Improved accuracy on test DS from {test\_accuracy[0]} to {test\_accuracy[-1]}"