Turbocharge ML with JAX and TPUs

RA Stringer

2023-06-02

Table of contents

Preface

JAX and TPUs are powering developers and researchers to speed up machine learning training workloads and have more fun in the process.

We hope you enjoy the course and start experimenting!

This is a Quarto book.

To learn more about Quarto books visit https://quarto.org/docs/books.

1 TPUs

1.0.1 Supercomputer for ML

Google designed Cloud TPUs as a matrix processor focused making training and inference of neural networks faster, and more power efficient. The TPU is built for massive matrix processing, and its systolic array architecture assigns thousands of interconnected multiply-accumulators to the task. Cloud TPU v3 contains two systolic arrays of 128 x 128 ALUs, on a single processor. For workloads bound by matmul, TPU can generate significant efficiencies.

1.0.2 Getting started

Prerequisite: a Google Cloud project.

There are several ways to run the commands below.

Vertex AI: If you're running notebooks from within Vertex Workbench, simply open a terminal within the notebook instance and run the commands. Local: Download the gcloud SDK, or open a shell from within Cloud console. * Compute Engine VM: run the commands to set up a TPU VM.

1.0.3 Setting up the VM

First, run the following command to enable the TPU API, and set your user and project configuration:

```
gcloud services enable tpu.googleapis.com
gcloud config set account <your-email-account>
gcloud config set project <your-project>
```

1.0.4 Create the TPU VM

For more information, an extensive guide can be found here.

```
gcloud compute tpus tpu-vm create tpu-name  \
  --zone=zone \
  --accelerator-type=${ACCELERATOR_TYPE}  \
  --version=tpu-vm-v4-base
```

Note: For v2 and v3 configurations use the tpu-vm-base TPU software `version`. For v4 configurations use tpu-vm-v4-base. The correct version of libtpu.so is automatically installed when JAX is installed on the machine.

Since there is no TPU specific JAX software version, we have to manually install JAX on the TPU VM.

To see a list of versions (such as TensorFlow, other PyTorch versions), replace with zone with the zone of yoru project (eg us-central1-b) and run:

```
gcloud compute tpus tpu-vm versions list --zone <ZONE>
```

For all TPU types, the version is followed by the number of TensorCores (e.g., 8, 32, 128). For example, –accelerator-type=v2-8 specifies a TPU v2 with 8 TensorCores and v3-1024 specifies a v3 TPU with 1024 TensorCores (a slice of a v3 Pod).

1.0.5 Connecting to a TPU VM

From one of the options above (Workbench, local terminal, Compute Engine VM etc), adjust the VM name and zone placeholders:

```
gcloud compute tpus tpu-vm ssh <your-tpu-vm-name> --zone <your-zone>
```

1.0.6 Connecting to a TPU VM via local notebook

One of the most popular ways to connect is via a Jupyter Notebook either on another VM or a local machine. This means that rather than develop in notebooks and move .py files to the TPU VM to run them, all experimentation on the notebook can benefit from the TPU.

1.0.6.1 Steps:

Set up the TPU VM as above, and connect from your local machine (or another VM) with a slightly different command (change the parameters within <...>):

```
gcloud compute tpus tpu-vm ssh <tpu_vm_name> --zone <zone>  -- -L 8888:localhost:8888
```

Once connected for the first time, install the JAX library:

```
pip install --upgrade 'jax[tpu]>0.3.0' \
  -f https://storage.googleapis.com/jax-releases/libtpu_releases.html"
```

Let's check JAX is working before going further. From the terminal connected to the VM, try the following lines of code (number of devices will vary depending on configuration):

```
python3
>>> import jax
>>>  num_devices = jax.device_count()
>>> device_type = jax.devices()[0].device_kind
>>> print(f"Using {num_devices} JAX devices of type {device_type}.")
>>> Using 8 JAX devices of type Cloud TPU.
```

It should then be possible to launch a notebook running on the TPU via the usual command:

```
jupyter-lab
```

or

```
jupyter notebook
```

depending on whether you have JupyterLab or classic Jupyter Notebook installed. Now accessing `localhost:8888` in a browser (use the link in the terminal that results from the commands above) should take you to the notebook environment.

1.0.7 Connecting to a TPU VM

From one of the options above (Workbench, local terminal, Compute Engine VM etc), adjust the VM name and zone placeholders:

```
gcloud compute tpus tpu-vm ssh <your-tpu-vm-name> --zone <your-zone>
```

1.0.7.1 Connecting to a TPU VM via local notebook

One of the most popular ways to connect is via a Jupyter Notebook either on another VM or a local machine. This means that rather than develop in notebooks and move .py files to the TPU VM to run them, all experimentation on the notebook can benefit from the TPU.

1.0.7.2 Steps:

Set up the TPU VM as above, and connect from your local machine (or another VM) with a slightly different command (change the parameters within `<...>`):

```
gcloud compute tpus tpu-vm ssh <tpu_vm_name> --zone <zone>  -- -L 8888:localhost:8888
```

It should then be possible to launch a notebook running on the TPU via the usual command:

```
jupyter-lab
```

or

```
jupyter notebook
```

depending on whether you have JupyterLab or classic Jupyter Notebook installed. Now accessing `localhost:8888` in a browser (use the link in the terminal that results from the commands above) should take you to the notebook environment.

2 Introduction to JAX

JAX is a framework that enables high-performance numerical computing by merging Autograd and XLA (Accelerated Linear Algebra). Autograd was originally a library created for automatic differentiation of Python and NumPy code, which has since been added to JAX. XLA is an optimizing compiler for machine learning, which can significantly speed up workloads on both TPU and GPU devices.

2.0.1 Why learn JAX?

2.0.1.1 High performance

JAX has excellent support for accelerators such as GPUs and TPUs and leverages both XLA and Just-In-Time compilation for speedy numerical computation.

2.0.1.2 Automatic differentiation

JAX's grad transform function renders trivial the calculation of gradients of complex functions for gradient descent, backpropagation and other optimization algorithms. Research and experimentation: The framework's flexibility grants low-level programming power to developers for effective and rapid prototyping and research. The ease with which JAX code can be run on any device, functional programming and dynamic computation graphs are ideal for experimentation with different machine learning models.

2.0.1.3 Composable and functional

JAX encourages a functional programming style, enabling clean and modular code. Its functions are pure, which means they produce the same output from the same input every time, limiting side effects and improving safety and resusibility. Plays well with others: JAX features interoperability with NumPy, and can be used in conjunction with TensorFlow and PyTorch. Users often combine the features of other libraries with the performance benefits of JAX.

2.0.1.4 How does JAX compare to other frameworks?

2.0.1.5 TensorFlow

TensorFlow creates static computational graphs that define operations and dependencies before execution, enabling efficient optimization and deployment across devices. The tf.Gradient.Tape API supports automatic differentiation, and extensive support for TPUs and GPUs. The framework has a large and mature ecosystem and strong industry adoption. High-level APIs such as Keras and tf.Module make development easier, and TensorFlow Hub offers a repository of pre-trained models.

2.0.1.6 PyTorch

Uses "eager" execution, dynamically building computation graphs as operations are invoked. This flexibility can allow for more intuitive experimentation and debugging. The torch.autograd module enables automatic differentiation and computing gradients during the backward pass. The framework has good support for TPUs via the XLA compiler, and torch.cuda for GPU memory management. PyTorch has a growing ecosystem focused on research and flexibility. Many state-of-the-art models are implemented and shared first using the framework. Its high-level API simplifies building neural networks and includes modules for optimizztion, data handling and visualization.

2.0.1.7 JAX

JAX combines elements of static and dynamic compilation, providing a hybrid approach in which code can be executed "just-in-time", or traced into static compilation graphs. This enables efficient execution and optimization while retaining flexibility and ease of debugging. The framework offers automatic differentiation through its functional programming model, leveraging function transformations to compute gradients and allow fine-grained control over differentiation.

JAX enjoys excellent GPU and TPU support, integrating tightly with XLA and requiring zero code changes to switch between devices. Spreading data and computation across cores is made simple via its function transformations such as pmap. JAX has a smaller and rapidly-growing community, with popularity among researchers such as Google's DeepMind. It's lower-level API is intuitive to those already familiar with NumPy, and neural network libraries Flax and Haiku provide modules and optimizers for training models.

2.0.2 Key concepts

JAX provides an API similar to NumPy that is intuitive and familar to many researchers and engineers. The framework includes composable function transformations for just-in-time compilation, batching, automatic differentiation and parallelization. JAX can be run on TPU, GPU and CPU, without any code changes.

2.0.3 Accelerated NumPy

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

```
x = jnp.arange(10)
print(x)
```

```
[0 1 2 3 4 5 6 7 8 9]
```

We can move this array from CPU to GPU or TPU.

2.0.4 A word on Colab TPUs

It used to be easy to switch from GPU to TPU in Colabs, however the TPUs set up is now behind JAX version >=0.4, which requires TPU VMs (on GCP).

JAX 0.4 and newer requires TPU VMs, which Colab does not provide at this time. You can still use jax 0.3.25 on Colab TPU, which is the version that comes installed by default on Colab TPU runtimes. If you've already updated JAX, you can choose Runtime->Disconnect and Delete Runtime to get a fresh TPU VM, and then skip the pip install step so that you keep the default jax/jaxlib version 0.3.25.

For now, we will proceed with GPUs and run code on Cloud TPU VMs later in the course.

```
import jax

print(jax.device_count())
device_type = jax.devices()[0].device_kind
device_type
```

```
1
```

```
'Tesla T4'
```

More on how JAX creates random numbers later. For now, let's initialize a pseudo random number generator (PRNG) key.

```
import jax.numpy as jnp
from jax import random

key = random.PRNGKey(0)
```

```
key, subkey = random.split(key)
x = random.normal(key, (1000, 1000))

print(f"x is of shape: {x.shape}")
print(f"x has dtype: {x.dtype}")
```

```
x is of shape: (1000, 1000)
x has dtype: float32
```

```
import numpy as np

x = np.array(x)

def x_on_cpu(x):
  return np.dot(x, x)

%timeit -n 1 -r 1 x_on_cpu(x)
```

```
29.3 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
```

```
def x_on_gpu(x):
  return jnp.dot(x, x)

%timeit -n 5 -r 5 x_on_gpu(x).block_until_ready()
```

```
3.03 ms ± 723 µs per loop (mean ± std. dev. of 5 runs, 5 loops each)
```

```
def numpy_random_state():
  print(str(np.random.get_state())[:100], '...')

numpy_random_state()
```

```
('MT19937', array([1483202178, 2954356075, 3069814800,  774374480, 1305506623,
       453414418, 21 ...
```

2.0.5 Random numbers

Generating random numbers can seem complicated at first glance.

Pseudo random number generation (PRNG) creates sequences that aren't truly random because they're determined by their initial value, the `seed`. Each random sampling is a deterministic functino of a `state` carried between examples.

In NumPy, PRNG is based on a global state, using the `numpy.random` module.

```
def numpy_random_state():
  print(str(np.random.get_state())[:100], '...')

numpy_random_state()
```

```
('MT19937', array([         0,          1, 1812433255, 1900727105, 1208447044,
       2481403966, 40 ...
```

This state is then updated by each call to `random`.

```
np.random.seed(0)

numpy_random_state()

_ = np.random.uniform()

numpy_random_state()
```

```
('MT19937', array([         0,          1, 1812433255, 1900727105, 1208447044,
       2481403966, 40 ...
('MT19937', array([2443250962, 1093594115, 1878467924, 2709361018, 1101979660,
       3904844661,  6 ...
```

JAX handles PRNG differently since the framework intends to be easy to reproduce, parallelize and vectorize.

Rather than use a global state, JAX uses a state called a `key`.

```
key = random.PRNGKey(10)

print(key)
```

```
[ 0 10]
```

Random functions consume, and don't alter, the key. This means the same key should always produce the same sample.

```
for i in range(0, 3):
  print(random.normal(key))
```

```
-1.3445405
-1.3445405
-1.3445405
```

One practice to bear in mind, then, is never to resuse keys, unless identical outputs are necessary. We can achieve independent keys by using the `split()` function.

```
key, subkey = random.split(key)
```

2.0.6 Intermediate representations

Introducing the Jaxpr

An *intermediate representation* is an internal interpretation of machine learning code used by underlying frameworks or compilers to optimize the program. When we write code in a framework such as JAX or PyTorch, it is converted from high-level code into a computational graph, or symbolic representation. This is further transformed into an intermediate representation (IR) optimized for efficiency. The IR is then optimized over several passes to conduct operations such as constant folding, operation fusion, model parallelization, and quantization. This is followed by hardware-specific compilation, to convert the IR into low-level code optimized for the required backend (TPU, GPU etc).

The jaxpr

JAX converts functions into an intermediate representation called a jaxpr . Transformations such as grad then work this the jaxpr representation. JAX works by tracing functions. Before we look at what that means, consider this simple Python function:

```
def sum_squares(x):
    return jnp.sum(x**2)
```

What does this function do? It looks easy at first glance, since it adds the result of x**2. x could be a single variable, or an array. However, given Python's dynamism, it could do anything depending on what `x` is…square an ice-cream order, print job or jackpot winnings in a slot machine.

JAX takes advantage of this dynamism by running functions using tracer values. These are experimental inputs to a function, which help JAX understand how it works and what it will accomplish.

We can take a look at the IR, the jaxpr, by using the `jax.make_jaxpr` method.

```
from jax import make_jaxpr

print(make_jaxpr(sum_squares)(3.0))
```

```
{ lambda ; a:f32[]. let
    b:f32[] = integer_pow[y=2] a
    c:f32[] = convert_element_type[new_dtype=float32 weak_type=False] b
    d:f32[] = reduce_sum[axes=()] c
  in (d,) }
```

This result comprises the primitive operations, called lax, that JAX knows how to transform.

Herein lies JAX's power; with no need to make the API complicated, JAX develops a sound idea of what the function is doing, and knows how to vectorize with `vmap`, parallelize with `pmap` , and how to just-in-time compile with `jit` .

2.1 Transformations

Modular, functional programming

JAX can transform functions. This means a numerical function can be returned as a new function that, for example, computes the gradient of, or parallelizes the original function. It could also do both!

2.1.1 grad

One of the most commonly used transformations, `jax.grad` calculates the gradient of a function.

```
from jax import grad

def sum_squares(x):
    return jnp.sum(x**2)
```

Since `jax.grad(f)` computes the gradient of function `f`, `jax.grad(f)(x)` is the gradient of f at x .

```
print(grad(sum_squares)(3.0))
```

```
6.0
```

```
print(grad(grad(sum_squares))(3.0))
```

```
2.0
```

```
import math

def cylinder_volume(r, h):
    vol = jnp.pi * r**2 * h
    return vol

# Compute the volume of a cylinder with radius 3, and height 3
print(cylinder_volume(3, 3))
```

```
84.82300164692441
```

```
print(grad(cylinder_volume)(4.0, 8.0))
print(grad(cylinder_volume)(2.0, 6.0))
```

```
201.06194
75.398224
```

```
print(grad(grad(cylinder_volume))(4.0, 8.0))
print(grad(grad(cylinder_volume))(2.0, 6.0))
```

```
50.265484
37.699112
```

We can use argnums to calculate the gradient with respect to different arguments:

```
def f(x):
  if x > 0:
    return 2 * x ** 3
  else:
    return 3 * x
```

```
key = random.PRNGKey(0)
x = random.normal(key, ())
print(key)
print(x)

print(grad(f)(x))
print(grad(f)(-x))
```

```
[0 0]
-0.20584226
3.0
0.2542262
```

An obvious example to make use of `grad` would be a loss function.

```
def loss(preds, targets):
  return jnp.sum((preds-targets)**2)

x = jnp.asarray([1.0, 2.0, 3.0, 4.0])
targets = jnp.asarray([1.1, 2.1, 3.1, 4.1])

print(grad(loss)(x, y))
```

```
[-4. -2.  0.  2.]
```

2.1.2 Value and grad

We can return both the value and gradient of a function using `value_and_grad`. This is a common pattern in machine learning for logging training loss.

```
from jax import value_and_grad

value_and_grad(loss)(x, y)
```

```
(Array(6., dtype=float32), Array([-4., -2.,  0.,  2.], dtype=float32))
```

2.1.3 jit

The `jax.jit()` transformation performs Just In Time (JIT) compilation of a JAX Python function for efficient execution in XLA.

Let's go back to our `sum_squares()` function and time its original implementation on an array of numbers 1-100.

```
from jax import jit

def sum_squares(x):
    return jnp.sum(x**2)

x = jnp.arange(100)

%timeit sum_squares(x).block_until_ready()
```

```
419 µs ± 41.1 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

Let's jit the function and notice the speed improvement.

```
sum_squares_jit = jit(sum_squares)

# Warm up
sum_squares_jit(x).block_until_ready()

%timeit sum_squares_jit(x).block_until_ready()
```

```
72.6 µs ± 20.4 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

In case this notation isn't familiar, µs denotes a 'microsecond', or a millionth of a second. ns is a 'nanosecond', a billionth of a second. Our jitted function is considerably faster in this simple example. Note: JAX's asynchronous execution model means the Python call might return before the computation ends. This is why we use the block_until_ready() method to make sure we return the end result. a returned array would not be populated as soon as the function returns. Using block_until_ready means we time the actual computation, not just the dispatch.

2.1.4 Sharp edges

It isn't possible or economical to jit everything. jit will throw errors when function inputs spark conditional chains (eg if x < 5: …) and jit itself creates some overhead. jit is best reserved for compiling complex functions that will run several times, such as updating weights in a training loop.

2.1.5 vmap

The `jax.vmap` transformation generates a vectorized implementation of a function.

Reference for this section (thanks to DeepMind).

We can loop over a batch in Python however such operations tend to be costly.

```
from jax import vmap

mat = random.normal(key, (150, 100))
batched_x = random.normal(key, (10, 100))

def apply_matrix(v):
  return jnp.dot(mat, v)
```

```
def naively_batched_apply_matrix(v_batched):
  return jnp.stack([apply_matrix(v) for v in v_batched])

print('Naively batched')
%timeit naively_batched_apply_matrix(batched_x).block_until_ready()
```

```
Naively batched
3.62 ms ± 298 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

```
def vmap_batched_apply_matrix(v_batched):
  return vmap(apply_matrix)(v_batched)

print('Auto-vectorized with vmap')
%timeit vmap_batched_apply_matrix(batched_x).block_until_ready()
```

```
Auto-vectorized with vmap
1.17 ms ± 21.4 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

```
@jit
def jit_vmap_batched_apply_matrix(v_batched):
  return vmap(apply_matrix)(v_batched)

print('jitted and auto-vectorized with vmap')
%timeit jit_vmap_batched_apply_matrix(batched_x).block_until_ready()
```

```
jitted and auto-vectorized with vmap
86.9 µs ± 10 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

Putting it all together

We take a loss function, use it to find gradients with grad, vectorize it for work across batches, then jit compile, all in one line.

```
import jax.numpy as jnp
from jax import grad, vmap, jit

def predict(params, inputs):
    for W, b in params:
        outputs = jnp.dot(inputs, W) + b
        inputs = jnp.tahn(outputs)
    return outputs

def mse_loss(params, batch):
    inputs, targets = batch
    preds = predict(params, inputs)
    loss = jnp.sum((preds - targets) ** 2)
    print(loss)
    return loss
```

```
gradients = jit(grad(mse_loss))
vectorized_gradients = jit(vmap(grad(mse_loss), in_axes=(None, 0)))
```

2.1.6 pmap

pmap

```
def pmap_batched_apply_matrix(v_batched):
  return pmap(apply_matrix)(v_batched)
```

```
pmap_batched_apply_matrix(batched_x)
```

NameError: ignored

3 Linear Regression in JAX

```
import jax
import jax.numpy as jnp
import numpy as np
import matplotlib.pyplot as plt
from jax import tree_util, random
```

Reference: JAX for the impatient

3.0.1 Pytrees

Before we jump into linear regression in JAX, let's have a quick look at pytrees. Pytrees are everywhere in JAX and Flax.

JAX treats a pytree as a container of leaf elements. These can include lists, tuples and dicts, so is basically a structure for nested data. Container types do not need to match if nested.

JAX provides the `tree_util` package for working with pytrees.

```
from jax import tree_util

tree = [1, {"k1": 2, "k2": (3, 4)}, 5]
print('tree:', tree)
```

```
tree: [1, {'k1': 2, 'k2': (3, 4)}, 5]
tree_util.tree_map(lambda x: x*2, tree): [2, {'k1': 4, 'k2': (6, 8)}, 10]
```

The `tree_map` function is frequently used for updating a tree and its leaves.

```
tree_util.tree_map(lambda x: x*2, tree)
```

```
[2, {'k1': 4, 'k2': (6, 8)}, 10]
```

We can also provide a tuple of aditional trees of the same shape to the original tree to enable a function to operate on each leaf.

```
transformed_tree = tree_util.tree_map(lambda x: x*2, tree)
tree_util.tree_map(lambda x,y: x+y, tree, transformed_tree)
```

```
[3, {'k1': 6, 'k2': (9, 12)}, 15]
```

```
# Linear feed-forward.
def predict(W, b, x):
  return jnp.dot(x, W) + b

# Loss function: Mean squared error.
def mse(W, b, x_batched, y_batched):
  # Define the squared loss for a single pair (x,y)
  def squared_error(x, y):
    y_pred = predict(W, b, x)
    return jnp.inner(y-y_pred, y-y_pred) / 2.0
  # We vectorize the previous to compute the average of the loss on all samples.
  return jnp.mean(jax.vmap(squared_error)(x_batched, y_batched), axis=0)
```

```
# Set problem dimensions.
n_samples = 20
x_dim = 10
y_dim = 5

# Generate random ground truth W and b.
key = random.PRNGKey(0)
k1, k2 = random.split(key)
W = random.normal(k1, (x_dim, y_dim))
b = random.normal(k2, (y_dim,))

# Generate samples with additional noise.
key_sample, key_noise = random.split(k1)
x_samples = random.normal(key_sample, (n_samples, x_dim))
y_samples = predict(W, b, x_samples) + 0.1 * random.normal(key_noise,(n_samples, y_dim))
print('x shape:', x_samples.shape, '; y shape:', y_samples.shape)
```

```
x shape: (20, 10) ; y shape: (20, 5)
```

In this linear regression, `params` is a pytree which contains `W` and `b`.

```
# Linear feed-forward that takes a params pytree.
def predict_pytree(params, x):
  return jnp.dot(x, params['W']) + params['b']

# Loss function: Mean squared error.
def mse_pytree(params, x_batched,y_batched):
  # Define the squared loss for a single pair (x,y)
  def squared_error(x,y):
    y_pred = predict_pytree(params, x)
    return jnp.inner(y-y_pred, y-y_pred) / 2.0
  # We vectorize the previous to compute the average of the loss on all samples.
  return jnp.mean(jax.vmap(squared_error)(x_batched, y_batched), axis=0)

# Initialize estimated W and b with zeros. Store in a pytree.
params = {'W': jnp.zeros_like(W), 'b': jnp.zeros_like(b)}
```

JAX can differentiate the pytree parameters

```
jax.grad(mse_pytree)(params, x_samples, y_samples)
```

```
{'W': Array([[ 3.02512199e-05,  2.38317996e-04,  5.86672686e-05,
          1.45167112e-04, -1.08840875e-04],
        [-7.21593387e-05, -5.92094846e-04, -1.44919526e-04,
         -3.55741940e-04,  2.12500338e-04],
        [ 6.48805872e-06,  5.42663038e-05,  1.30501576e-05,
          3.26364461e-05, -1.73687004e-05],
        [-1.58965122e-05, -1.67803839e-04, -3.67928296e-05,
         -9.91356210e-05,  1.37500465e-05],
        [-9.83832870e-05, -7.85302604e-04, -1.94110908e-04,
         -4.74306522e-04,  3.20815481e-04],
        [-6.23832457e-05, -5.39575703e-04, -1.28836837e-04,
         -3.22562526e-04,  1.54131034e-04],
        [-8.21873546e-05, -6.98693097e-04, -1.67359249e-04,
         -4.20103315e-04,  2.30040867e-04],
        [-7.44033605e-06, -5.03805932e-05, -1.21158082e-05,
         -3.23755667e-05,  4.07989137e-05],
        [ 2.81375833e-06,  5.13494015e-05,  9.00402665e-06,
          2.99257226e-05,  1.63719524e-05],
        [-2.90344469e-05, -2.14974396e-04, -5.49759716e-05,
         -1.30489469e-04,  1.08879060e-04]], dtype=float32),
```

```
 'b': Array([-3.0185096e-05, -2.2265501e-04, -5.7548052e-05, -1.3502731e-04,
        1.1305924e-04], dtype=float32)}
```

```python
@jax.jit
def update_params_pytree(params, learning_rate, x_samples, y_samples):
  params = jax.tree_util.tree_map(
        lambda p, g: p - learning_rate * g, params,
        jax.grad(mse_pytree)(params, x_samples, y_samples))
  return params

learning_rate = 0.3  # Gradient step size.
print('Loss for "true" W,b: ', mse_pytree({'W': W, 'b': b}, x_samples, y_samples))
for i in range(101):
  # Perform one gradient update.
  params = update_params_pytree(params, learning_rate, x_samples, y_samples)
  if (i % 5 == 0):
    print(f"Loss step {i}: ", mse_pytree(params, x_samples, y_samples))
```

```
Loss for "true" W,b:  0.02363979
Loss step 0:  10.97141
Loss step 5:  1.0798324
Loss step 10:  0.3795825
Loss step 15:  0.17855297
Loss step 20:  0.094415195
Loss step 25:  0.054522194
Loss step 30:  0.03448924
Loss step 35:  0.024058029
Loss step 40:  0.018480862
Loss step 45:  0.015438682
Loss step 50:  0.01375394
Loss step 55:  0.0128103
Loss step 60:  0.012277315
Loss step 65:  0.011974388
Loss step 70:  0.011801446
Loss step 75:  0.011702419
Loss step 80:  0.011645543
Loss step 85:  0.011612838
Loss step 90:  0.011594015
Loss step 95:  0.011583163
Loss step 100:  0.011576912
```

Here we can also use `jax.value_and_grad()` to compute both the return value of the input function and its gradient.

```
# Using jax.value_and_grad instead:
loss_grad_fn = jax.value_and_grad(mse_pytree)
for i in range(101):
  # Note that here the loss is computed before the param update.
    loss_val, grads = loss_grad_fn(params, x_samples, y_samples)
    params = jax.tree_util.tree_map(
        lambda p, g: p - learning_rate * g, params, grads)
    if (i % 5 == 0):
        print(f"Loss step {i}: ", loss_val)
```

```
Loss step 0:  0.011576912
Loss step 5:  0.011573299
Loss step 10:  0.011571216
Loss step 15:  0.011570027
Loss step 20:  0.0115693165
Loss step 25:  0.011568918
Loss step 30:  0.011568695
Loss step 35:  0.01156855
Loss step 40:  0.011568478
Loss step 45:  0.011568436
Loss step 50:  0.011568408
Loss step 55:  0.011568391
Loss step 60:  0.01156838
Loss step 65:  0.011568381
Loss step 70:  0.01156838
Loss step 75:  0.011568385
Loss step 80:  0.011568374
Loss step 85:  0.01156838
Loss step 90:  0.01156837
Loss step 95:  0.0115683675
Loss step 100:  0.01156837
```

4 Exercise 1 Solution

With thanks to DeepMind for code here.

```
# Copyright 2018 The JAX Authors.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     https://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

"""A basic MNIST example using Numpy and JAX.

The primary aim here is simplicity and minimal dependencies.
"""

import array
import gzip
import os
from os import path
import struct
import urllib.request

import numpy as np


_DATA = "/tmp/jax_example_data/"
```

```
def _download(url, filename):
  """Download a url to a file in the JAX data temp directory."""
  if not path.exists(_DATA):
    os.makedirs(_DATA)
  out_file = path.join(_DATA, filename)
  if not path.isfile(out_file):
    urllib.request.urlretrieve(url, out_file)
    print(f"downloaded {url} to {_DATA}")


def _partial_flatten(x):
  """Flatten all but the first dimension of an ndarray."""
  return np.reshape(x, (x.shape[0], -1))


def _one_hot(x, k, dtype=np.float32):
  """Create a one-hot encoding of x of size k."""
  return np.array(x[:, None] == np.arange(k), dtype)


def mnist_raw():
  """Download and parse the raw MNIST dataset."""
  # CVDF mirror of http://yann.lecun.com/exdb/mnist/
  base_url = "https://storage.googleapis.com/cvdf-datasets/mnist/"

  def parse_labels(filename):
    with gzip.open(filename, "rb") as fh:
      _ = struct.unpack(">II", fh.read(8))
      return np.array(array.array("B", fh.read()), dtype=np.uint8)

  def parse_images(filename):
    with gzip.open(filename, "rb") as fh:
      _, num_data, rows, cols = struct.unpack(">IIII", fh.read(16))
      return np.array(array.array("B", fh.read()),
                      dtype=np.uint8).reshape(num_data, rows, cols)

  for filename in ["train-images-idx3-ubyte.gz", "train-labels-idx1-ubyte.gz",
                   "t10k-images-idx3-ubyte.gz", "t10k-labels-idx1-ubyte.gz"]:
    _download(base_url + filename, filename)

  train_images = parse_images(path.join(_DATA, "train-images-idx3-ubyte.gz"))
```

```
  train_labels = parse_labels(path.join(_DATA, "train-labels-idx1-ubyte.gz"))
  test_images = parse_images(path.join(_DATA, "t10k-images-idx3-ubyte.gz"))
  test_labels = parse_labels(path.join(_DATA, "t10k-labels-idx1-ubyte.gz"))

  return train_images, train_labels, test_images, test_labels


def mnist(permute_train=False):
  """Download, parse and process MNIST data to unit scale and one-hot labels."""
  train_images, train_labels, test_images, test_labels = mnist_raw()

  train_images = _partial_flatten(train_images) / np.float32(255.)
  test_images = _partial_flatten(test_images) / np.float32(255.)
  train_labels = _one_hot(train_labels, 10)
  test_labels = _one_hot(test_labels, 10)

  if permute_train:
    perm = np.random.RandomState(0).permutation(train_images.shape[0])
    train_images = train_images[perm]
    train_labels = train_labels[perm]

  return train_images, train_labels, test_images, test_labels
```

```
# Copyright 2018 The JAX Authors.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     https://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

"""A basic MNIST example using Numpy and JAX.

The primary aim here is simplicity and minimal dependencies.
"""
```

```
import time

import numpy.random as npr

from jax import jit, grad
from jax.scipy.special import logsumexp
import jax.numpy as jnp
from examples import datasets

def init_random_params(scale, layer_sizes):
    # Solution
    key = random.PRNGKey(0)
    # Split the PRNGKey into two new keys
    key1, key2 = random.split(key, 2)
    params = [(scale * random.normal(key1, (m, n)), scale * random.normal(key2, (n,)))
              for m, n in zip(layer_sizes[:-1], layer_sizes[1:])]
    return params


def predict(params, inputs):
  activations = inputs
  for w, b in params[:-1]:
    outputs = jnp.dot(activations, w) + b
    activations = jnp.tanh(outputs)

  final_w, final_b = params[-1]
  logits = jnp.dot(activations, final_w) + final_b
  return logits - logsumexp(logits, axis=1, keepdims=True)

def loss(params, batch):
  inputs, targets = batch
  preds = predict(params, inputs)
  return -jnp.mean(jnp.sum(preds * targets, axis=1))

def accuracy(params, batch):
  inputs, targets = batch
  target_class = jnp.argmax(targets, axis=1)
  predicted_class = jnp.argmax(predict(params, inputs), axis=1)
  return jnp.mean(predicted_class == target_class)
```

```
if __name__ == "__main__":
  layer_sizes = [784, 1024, 1024, 10]
  param_scale = 0.1
  step_size = 0.001
  num_epochs = 10
  batch_size = 128

  train_images, train_labels, test_images, test_labels = mnist()
  num_train = train_images.shape[0]
  num_complete_batches, leftover = divmod(num_train, batch_size)
  num_batches = num_complete_batches + bool(leftover)

  def data_stream():
    rng = npr.RandomState(0)
    while True:
      perm = rng.permutation(num_train)
      for i in range(num_batches):
        batch_idx = perm[i * batch_size:(i + 1) * batch_size]
        yield train_images[batch_idx], train_labels[batch_idx]
  batches = data_stream()

  # Solution
  # jit compiling the update function brings the most benefits;
  # it does the heavy lifting for the training loop and runs
  # many times
  @jit
  def update(params, batch):
    # Solution
    grads = grad(loss)(params, batch)
    return [(w - step_size * dw, b - step_size * db)
            for (w, b), (dw, db) in zip(params, grads)]

  params = init_random_params(param_scale, layer_sizes)
  for epoch in range(num_epochs):
    start_time = time.time()
    for _ in range(num_batches):
      params = update(params, next(batches))
    epoch_time = time.time() - start_time

    train_acc = accuracy(params, (train_images, train_labels))
    test_acc = accuracy(params, (test_images, test_labels))
```

```
    print(f"Epoch {epoch} in {epoch_time:0.2f} sec")
    print(f"Training set accuracy {train_acc}")
    print(f"Test set accuracy {test_acc}")
```

5 Flax Foundations

Efficient and flexible model development

By combining JAX's auto-differentiation and Flax's modular design, developers can easily construct and train state-of-the-art deep learning models. JAX/Flax traces pure functions and compiles for GPU and TPU accelerators.

```
import jax
from typing import Any, Callable, Sequence
from jax import lax, random, numpy as jnp
from flax.core import freeze, unfreeze
from flax import linen as nn
```

```
# Here's a single dense layer that takes a number of features as input
model = nn.Dense(features=5)

key1, key2 = random.split(random.PRNGKey(0))
# Dummy input data
x = random.normal(key1, (10,))
# Initialize the model
params = model.init(key2, x)
# Forward pass
model.apply(params, x)
```

```
WARNING:jax._src.xla_bridge:No GPU/TPU found, falling back to CPU. (Set TF_CPP_MIN_LOG_LEVEL=
```

```
Array([-1.3721193 ,  0.61131495,  0.6442836 ,  2.2192965 , -1.1271116 ],      dtype=float32)
```

Note we only mention to Flax the number of features for the output of the model, rather than specifying the size of the input. Flax works out the correct kernel size for us!

Let's take a look at the pytree:

```
# Check output shapes
jax.tree_util.tree_map(lambda x: x.shape, params)
```

```
FrozenDict({
    params: {
        bias: (5,),
        kernel: (10, 5),
    },
})
```

Notice the parameters are stored in a `FrozenDict`, which prevents any mutation of the values.

```python
import jax
import jax.numpy as jnp
import flax.linen as nn

# Dummy data
inputs = jnp.array([[0.2, 0.3, 0.4], [0.1, 0.2, 0.3]])
targets = jnp.array([[0.5], [0.8]])

# Simple feedforward neural network
class SimpleNetwork(nn.Module):
    hidden_size: int
    output_size: int

    def setup(self):
        self.dense1 = nn.Dense(self.hidden_size)
        self.dense2 = nn.Dense(self.output_size)

    def __call__(self, x):
        x = self.dense1(x)
        x = nn.relu(x)
        x = self.dense2(x)
        return x

# Initialization
hidden_size = 16
output_size = 1
rng = jax.random.PRNGKey(0)
model = SimpleNetwork(hidden_size, output_size)
params = model.init(rng, inputs)
tree = jax.tree_util.tree_map(lambda inputs: inputs.shape, params) # Checking output shape
print(tree)
```

```
  # Forward pass
  predictions = model.apply(params, inputs)

  print(f"Inputs: \n{inputs}")
  print(f"\nPredictions: \n{predictions}")
  print(f"\nTarget data: \n{targets}")
```

```
FrozenDict({
    params: {
        dense1: {
            bias: (16,),
            kernel: (3, 16),
        },
        dense2: {
            bias: (1,),
            kernel: (16, 1),
        },
    },
})
Inputs:
[[0.2 0.3 0.4]
 [0.1 0.2 0.3]]

Predictions:
[[-0.01026188]
 [-0.01458298]]

Target data:
[[0.5]
 [0.8]]
```

In this example, we defined our model explicitly using `setup`. We can also define architecrures using `nn.compact`, which allows us to define a modulea s a single method. This can lead to cleaner code if you are writing custom layers.

Here's our SimpleNetwork again, using `setup`.

```
  class SimpleNetwork(nn.Module):
      hidden_size: int
      output_size: int

      def setup(self):
```

```
        self.dense1 = nn.Dense(self.hidden_size)
        self.dense2 = nn.Dense(self.output_size)

    def __call__(self, x):
        x = self.dense1(x)
        x = nn.relu(x)
        x = self.dense2(x)
        return x
```

And using `nn.compact`:

```
class SimpleNetwork(nn.Module):
  hidden_size: int
  output_size: int

  @nn.compact
  def __call__(self, x):
    x = nn.Dense(hidden_size, name="dense1")(x)
    x = nn.relu(x)
    x = nn.Dense(output_size, name="dense2")(x)
    return x
```

If you are porting models from PyTorch, or prefer explicit definition and separation of sub-modules, `setup` may suit. `nn.compact` may be best for reducing duplication, writing code that looks closer to mathematical notation, or if you are using shape inference (parameters dependant on shapes of inputs unknown at initialization).

5.0.1 Flax modules

Flax it easy to incorporate training techniques such as batch normalization and learning rate scheduling via the `flax.linen.Module`.

Here's our simple multi-layer perceptron again:

```
class SimpleNetwork(nn.Module):
  hidden_size: int
  output_size: int

  @nn.compact
  def __call__(self, x):
    x = nn.Dense(hidden_size, name="dense1")(x)
    x = nn.relu(x)
```

```
    x = nn.Dense(output_size, name="dense2")(x)
    return x
```

Batch normalization is a regularization technique which computes running averages over feature dimensions. This speeds up training cycles and improves convergence. To apply batch normalization, we call upon `flax.linen.BatchNorm`.

```
class SimpleNetwork(nn.Module):
  hidden_size: int
  output_size: int

  @nn.compact
  def __call__(self, x, train: bool):
    x = nn.Dense(hidden_size, name="dense1")(x)
    x = nn.BatchNorn(use_running_average=not train)(x)
    x = nn.relu(x)
    x = nn.Dense(output_size, name="dense2")(x)
    return x
```

5.0.2 Dropout

Dropout is another (stochastic) regularization technique that randomly removes units in a network to improve reduce overfitting and improve generalization.

Dropout requires our PRNG skills to endure it is a random operation.

When splitting a key, we can simply split into three keys, granting the third for `flax.linen.dropout`.

```
key = jax.random.PRNGKey(seed=0)
main_key, params_key, dropout_key = jax.random.split(key=key, num=3)
```

Then add the module to our model:

```
class SimpleNetwork(nn.Module):
  hidden_size: int
  output_size: int

  @nn.compact
  def __call__(self, x, train: bool):
    x = nn.Dense(hidden_size, name="dense1")(x)
    x = nn.Dropout(rate=0.5, deterministic=not train)(x)
```

```
    x = nn.BatchNorm(use_running_average=not train)(x)
    x = nn.relu(x)
    x = nn.Dense(output_size, name="dense2")(x)
    return x
```

We can then initialize the model:

```
simple_net = SimpleNetwork(hidden_size=5, output_size=1)
x = jnp.empty((3, 4, 4, 5, 5))
# Dropout is enabled via `deterministic=True`.
variables = simple_net.init(params_key, x, train=False)
params = variables['params']
```

5.0.3 Train states

A "train state" is the mutable state of a model during training, including properties such as its parameters (weights) and optimizer state.

The train state is typically represented as an instance of the `flax.training.TrainState` class, which encapsulates and provides methods to update the state.

One of the features of JAX/Flax is its functional programming characteristic of immutability. Models are updates are purely functional, enabling model parallelism and efficient training.

```
# Example, will not run

def create_train_state(rng, learning_rate, momentum):
  """Creates initial `TrainState`."""
  cnn = CNN()
  params = cnn.init(rng, jnp.ones([1, 28, 28, 1]))['params']
  tx = optax.sgd(learning_rate, momentum)
  return train_state.TrainState.create(
      apply_fn=cnn.apply, params=params, tx=tx)
```

5.0.4 Optax

Optax is a gradient processing and optimization package. It is generally used with Flax as follows:

Create an optimizer state from parameters using any optimization method (eg `optax.rmsprop`). Compute loss gradients using `value_and_grad()`. Call the Optax `update` function to update

the internal optimizer state to work out how to tweak the parameters. Use `apply_updates` to apply update the to the parameters.

For example (will not run):

```
import optax

optimizer = optax.adam(learning_rate=learning_rate)
optimizer_state = optimizer.init(params)
loss_grad_func = jax.value_and_grad(mse)

for i in range(10):
  loss, grads = loss_grad_func(params, x_samples, y_samples)
  updates, optimizer_state = optimizer.update(grads, optimizer_state)
  params = optax.apply_updates(params, updates)
  if i % 10 == 0:
    print('Loss step {}: '.format(i), loss)
```

MNIST Example

```
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 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 = state.apply_fn({'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)


def train_epoch(state, train_ds, batch_size, rng):
  """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 = []

  for perm in perms:
    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, learning_rate, momentum):
  """Creates initial `TrainState`."""
  cnn = CNN()
  params = cnn.init(rng, jnp.ones([1, 28, 28, 1]))['params']
  tx = optax.sgd(learning_rate, momentum)
  return train_state.TrainState.create(
      apply_fn=cnn.apply, params=params, tx=tx)


def train_and_evaluate(learning_rate, momentum,
                       batch_size, num_epochs) -> train_state.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 train state (which includes the `.params`).
  """
```

```
  train_ds, test_ds = get_datasets()
  rng = jax.random.PRNGKey(0)

  rng, init_rng = jax.random.split(rng)
  state = create_train_state(init_rng, 0.01, 0.9 )

  for epoch in range(1, num_epochs + 1):
    rng, input_rng = jax.random.split(rng)
    state, train_loss, train_accuracy = train_epoch(state, train_ds,
                                                    64,
                                                    input_rng)
    _, 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_accurac
        % (epoch, train_loss, train_accuracy * 100, test_loss,
           test_accuracy * 100))

    print('train_loss', train_loss, epoch)
    print('train_accuracy', train_accuracy, epoch)
    print('test_loss', test_loss, epoch)
    print('test_accuracy', test_accuracy, epoch)

  return state
```

```
train_and_evaluate(0.01, 0.9, 128, 1)
```

6 Exercise 2: Linear Regression in Flax

```
import jax
from jax import numpy as jnp, random, lax, jit
from flax import linen as nn

# TODO: Data preparation
# Create variables X and Y:
# X is a 1 x 10 matrix
# Y is a 1-dimensional array of size 5
# Some references on generating matrices here:
# https://flax.readthedocs.io/en/latest/guides/jax_for_the_impatient.html

X = pass
Y = pass

# TODO: create a model of one Dense layer with 5 features
# For help:
# https://flax.readthedocs.io/en/latest/guides/flax_basics.html
model = nn.Dense(features=5)

@jit
def predict(params):
  return model.apply({'params': params}, X)

@jit
def loss_fn(params):
  return jnp.mean(jnp.abs(Y - predict(params)))

# Initialize the model with random values
# use random number generator ('rng') as input
# to initialize params based on input shape of 'X'.
@jit
def init_params(rng):
  mlp_variables = model.init({'params': rng}, X)
  return mlp_variables['params']
```

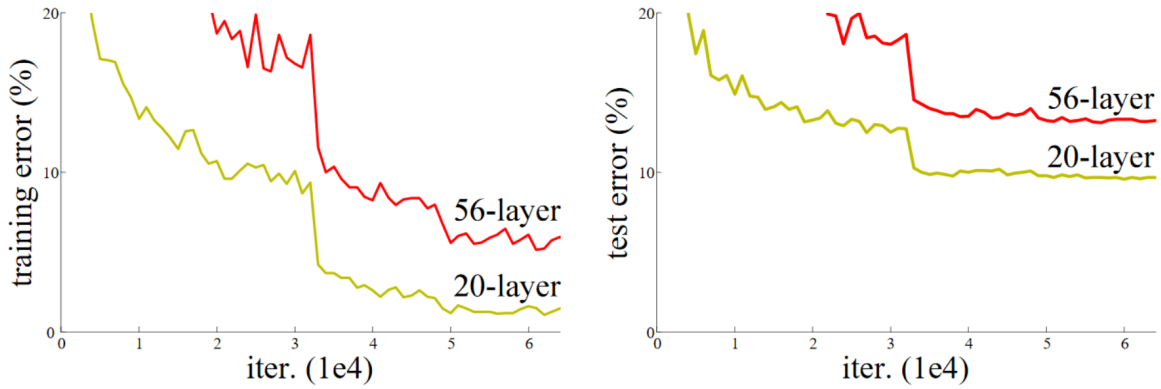
```
# TODO
# use the init_params function and
# jax.random to initalize random params
# using PRNGKey
params = None
print("initial params", params)

# Run SGD.
for i in range(50):
  # TODO use jax transformations to extract the loss value
  # and gradients of the loss with respect to the params
  loss, grad = pass
  print(i, "loss = ", loss, "Yhat = ", predict(params))
  lr = 0.03
  params = jax.tree_util.tree_map(lambda x, d: x - lr * d, params, grad)
```

7 Let's build a ResNet!

Now that we have an understanding of JAX, and how to build neural network layers and optimizations in Flax, let's put our skills to implementing one of the most cited academic publications in machine learning: Deep Residual Learning for Image Recognition.

7.0.1 The idea

The paper authors found that after batchnorm, networks with more layers often performed worse that those with fewer.



(Image from paper linked above by Kaimimg He and others).

The researchers experimented with the idea of adding extra layers as an 'identity mapping', which means they have parameters and are trainable, but which return inputs without changing them.

This idea resulted in 'skip connections', which leap over convolutions as in this diagram:

The skip connections make the network make the larger architecture easier to train, and prevent overfitting.

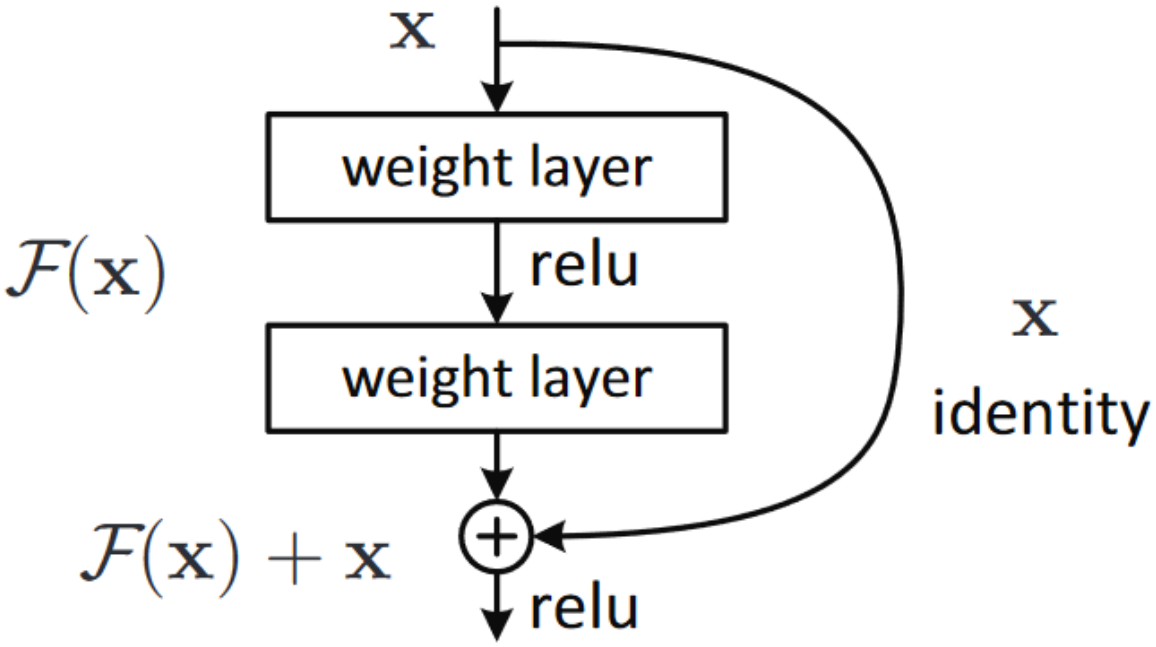


Figure 7.1: Skip connections

7.0.1.1 Residuals

A 'residual' is basically a prediction minus the target. ResNet blocks beat most earlier benchmarks because rather than asking them to predict the target, they predict the *difference* between the target and the prediction. This architecture proved very strong in detecting slight differences in images (is it a wolf or an off-leash Husky running through a dark forest?).

```
from functools import partial

from flax import linen as nn
import jax.numpy as jnp
from typing import Any, Callable, Sequence, Tuple
```

```
class ResNetBlock(nn.Module):
  filters: int
  strides: Tuple[int, int] = (1, 1)

  @nn.compact
  def __call__(self, x):
```

```
residual = x
y = nn.Conv(self.filters, (3, 3), self.strides)(x)
y = nn.BatchNorm()(y)
y = nn.relu(y)
y = nn.Conv(self.filters, (3, 3))(y)
return x + y

if residual.shape != y.shape:
      residual = nn.conv(self.filters, (1, 1),
                         self.strides, name='conv_proj')(residual)
      residual = self.norm(name='norm_proj')(residual)

return self.act(residual + y)
```

7.0.2 Bottleneck layers

To enable training deeper models without spiking memory and computation use, we can use bottleneck layers. These were also introduced in the original paper as suitable for ResNets with a depth of 50 or more layers.

In our original ResNet layer, we have two convolutions with kernel size 3. Bottleneck layers use a 1 x 1 convolution at the start and end, and a 3 x 3 layer in between.

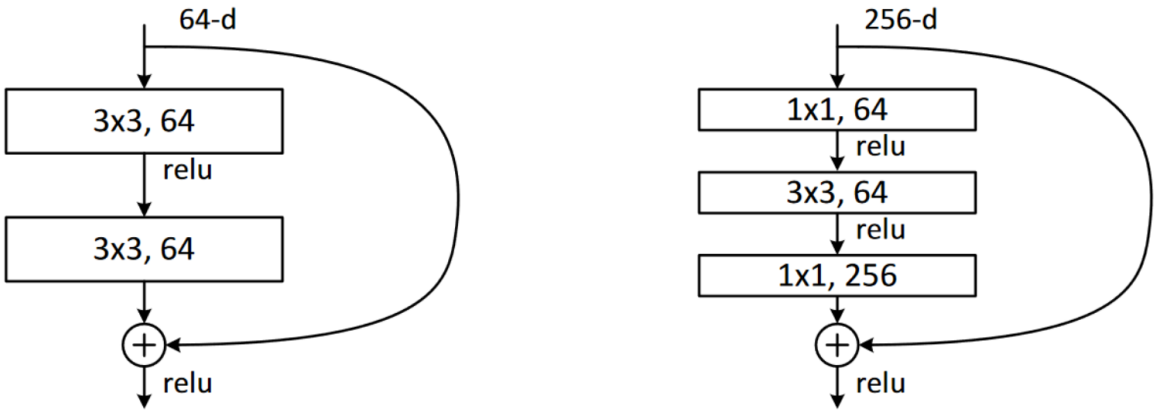


Figure 7.2: Bottleneck layers

These improve training when used with deeper models since they allow us to add more filters. Filters mean we can reduce the color channels of images, then restore them – hence their name.

```
class BottleneckResNetBlock(nn.Module):
  filters: int
  strides: Tuple[int, int] = (1, 1)

  @nn.compact
  def __call__(self, x):
    residual = x
    y = nn.Conv(self.filters, (1, 1))(x)
    y = nn.BatchNorm()(y)
    y = nn.relu(y)
    y = nn.Conv(self.filters, (3, 3), self.strides)(y)
    y = nn.BatchNorm()(y)
    y = nn.relu(y)
    y = nn.Conv(self.filters * 4, (1, 1), self.strides)(y)
    y = nn.BatchNorm(scale_init=nn.initializers.zeros_init())(y)

    if residual.shape != y.shape:
          residual = nn.Conv(self.filters * 4, (1, 1),
                             strides=(1, 1), name='conv_proj')(residual)
          residual = self.BatchNorm(name='norm_proj')(residual)

    return nn.relu(residual + y)
```

7.0.3 Creating the ResNet

Now we have our blocks, we can stack them together to create a fully-fledged ResNet.

Blocks are generally grouped by shape, so if our model has [2,2,2,2] blocks, it means we have four groups of 2 blocks.

We will use the original `ResNetBlock` for those < 50 layers, and the `BottleneckResNetBlock` for larger architectures.

The various ResNet sizes (ResNet18, ResNet50 etc) simply denote the number of layers.

A ResNet 18's blocks are [2,2,2,2], so how do we get to 18 layers?

We have 1 initial conv layer, 8 conv layers in residual blocks, a final conv layer, which gives us 10 layers. The remaining 8 are fully connected layers that follow the last conv layer, which typically serve as the classifier head and output predictions.

```
from functools import partial

class ResNet(nn.Module):
  num_classes: int
  block_class: nn.Module
  num_blocks: Sequence[int]
  filters: int = 64
  dtype: Any = jnp.float32


  @nn.compact
  def __call__(self, x, train: bool = True):
    x = nn.Conv(self.filters, (7, 7), (2, 2),
              padding=[(3, 3), (3, 3)],
              use_bias=False,
              name='conv_init')(x)
    x = nn.BatchNorm(name='bn_init')(x, use_running_average=not train)
    x = nn.relu(x)
    x = nn.max_pool(x, (3, 3), (2, 2), padding='SAME')

    for i, block_size in enumerate(self.num_blocks):
      for j in range(block_size):
        strides = (2, 2) if i > 0 and j == 0 else (1, 1)
        x = self.block_class(self.filters * (2**i),
                             strides=strides)(x)

    x = jnp.mean(x, axis=(1, 2))
    x = nn.Dense(self.num_classes, dtype=self.dtype)(x)
    x = jnp.asarray(x, self.dtype)

    return x

ResNet18 = partial(ResNet, num_blocks=[2, 2, 2, 2],
                   block_class=ResNetBlock)
ResNet34 = partial(ResNet, num_blocks=[3, 4, 6, 3],
                   block_class=ResNetBlock)
ResNet50 = partial(ResNet, num_blocks=[3, 4, 6, 3],
                   block_class=BottleneckResNetBlock)
ResNet101 = partial(ResNet, num_blocks=[3, 4, 23, 3],
                    block_class=BottleneckResNetBlock)
ResNet152 = partial(ResNet, num_blocks=[3, 8, 36, 3],
```

```
                    block_class=BottleneckResNetBlock)
ResNet200 = partial(ResNet, num_blocks=[3, 24, 36, 3],
                    block_class=BottleneckResNetBlock)
```

8 Stable Diffusion in JAX / Flax !

Grateful to share this notebook from Hugging Face. Related blog post here.

Hugging Face Diffusers supports Flax since version `0.5.1`! This allows for super fast inference on Google TPUs, such as those available in Colab, Kaggle or Google Cloud Platform.

This notebook shows how to run inference using JAX / Flax. If you want more details about how Stable Diffusion works or want to run it in GPU, please refer to this Colab notebook.

First, make sure you are using a TPU backend. If you are running this notebook in Colab, select `Runtime` in the menu above, then select the option "Change runtime type" and then select `TPU` under the `Hardware accelerator` setting.

Note that JAX is not exclusive to TPUs, but it shines on that hardware because each TPU server has 8 TPU accelerators working in parallel.

8.1 Setup

```
!pip install flax transformers ftfy
!pip install diffusers==0.9.0
```

```
import jax
```

```
num_devices = jax.device_count()
device_type = jax.devices()[0].device_kind

print(f"Found {num_devices} JAX devices of type {device_type}.")
assert "TPU" in device_type, "Available device is not a TPU, please select TPU from Edit >
```

```
Found 8 JAX devices of type Cloud TPU.
```

Then we import all the dependencies.

```
import numpy as np
import jax
import jax.numpy as jnp

from pathlib import Path
from jax import pmap
from flax.jax_utils import replicate
from flax.training.common_utils import shard
from PIL import Image

from huggingface_hub import notebook_login
from diffusers import FlaxStableDiffusionPipeline
```

8.2 Model Loading

Before using the model, you need to accept the model license in order to download and use the weights.

The license is designed to mitigate the potential harmful effects of such a powerful machine learning system. We request users to **read the license entirely and carefully**. Here we offer a summary:

1. You can't use the model to deliberately produce nor share illegal or harmful outputs or content,
2. We claim no rights on the outputs you generate, you are free to use them and are accountable for their use which should not go against the provisions set in the license, and
3. You may re-distribute the weights and use the model commercially and/or as a service. If you do, please be aware you have to include the same use restrictions as the ones in the license and share a copy of the CreativeML OpenRAIL-M to all your users.

Flax weights are available in Hugging Face Hub as part of the Stable Diffusion repo. To use them, you need to be a registered user in Hugging Face Hub and use an access token for the code to work. You have two options to provide your access token:

- Use the `huggingface-cli login` command-line tool in your terminal and paste your token when prompted. It will be saved in a file in your computer.
- Or use `notebook_login()` in a notebook, which does the same thing.

The following cell will present a login interface unless you've already authenticated before in this computer. You'll need to paste your access token.

```
if not (Path.home()/'.huggingface'/'token').exists(): notebook_login()
```

```
Login successful
Your token has been saved to /root/.huggingface/token
```

TPU devices support `bfloat16`, an efficient half-float type. We'll use it for our tests, but you can also use `float32` to use full precision instead.

```
dtype = jnp.bfloat16
```

Flax is a functional framework, so models are stateless and parameters are stored outside them. Loading the pre-trained Flax pipeline will return both the pipeline itself and the model weights (or parameters). We are using a `bf16` version of the weights, which leads to type warnings that you can safely ignore.

```
pipeline, params = FlaxStableDiffusionPipeline.from_pretrained(
    "CompVis/stable-diffusion-v1-4",
    revision="bf16",
    dtype=dtype,
)
```

```
Downloading:   0%|          | 0.00/563 [00:00<?, ?B/s]
```

```
Fetching 16 files:   0%|          | 0/16 [00:00<?, ?it/s]
```

```
Downloading:   0%|          | 0.00/342 [00:00<?, ?B/s]
```

```
Downloading:   0%|          | 0.00/4.78k [00:00<?, ?B/s]
```

```
Downloading:   0%|          | 0.00/608M [00:00<?, ?B/s]
```

```
Downloading:   0%|          | 0.00/209 [00:00<?, ?B/s]
```

```
Downloading:   0%|          | 0.00/230 [00:00<?, ?B/s]
```

```
Downloading:   0%|          | 0.00/587 [00:00<?, ?B/s]
```

```
Downloading:   0%|          | 0.00/246M [00:00<?, ?B/s]
```

```
Downloading:   0%|          | 0.00/525k [00:00<?, ?B/s]

Downloading:   0%|          | 0.00/472 [00:00<?, ?B/s]

Downloading:   0%|          | 0.00/806 [00:00<?, ?B/s]

Downloading:   0%|          | 0.00/1.06M [00:00<?, ?B/s]

Downloading:   0%|          | 0.00/587 [00:00<?, ?B/s]

Downloading:   0%|          | 0.00/1.72G [00:00<?, ?B/s]

Downloading:   0%|          | 0.00/556 [00:00<?, ?B/s]

Downloading:   0%|          | 0.00/167M [00:00<?, ?B/s]

Some of the weights of FlaxStableDiffusionSafetyChecker were initialized in bfloat16 precisio
[('concept_embeds',), ('concept_embeds_weights',), ('special_care_embeds',), ('special_care_e
You should probably UPCAST the model weights to float32 if this was not intended. See [`~Flax
Some of the weights of FlaxCLIPTextModel were initialized in bfloat16 precision from the mode
[('text_model', 'embeddings', 'position_embedding', 'embedding'), ('text_model', 'embeddings
You should probably UPCAST the model weights to float32 if this was not intended. See [`~Flax
Some of the weights of FlaxAutoencoderKL were initialized in bfloat16 precision from the mode
[('decoder', 'conv_in', 'bias'), ('decoder', 'conv_in', 'kernel'), ('decoder', 'conv_norm_out
You should probably UPCAST the model weights to float32 if this was not intended. See [`~Mode
Some of the weights of FlaxUNet2DConditionModel were initialized in bfloat16 precision from t
[('conv_in', 'bias'), ('conv_in', 'kernel'), ('conv_norm_out', 'bias'), ('conv_norm_out', 'sc
You should probably UPCAST the model weights to float32 if this was not intended. See [`~Mode
```

8.3 Inference

Since TPUs usually have 8 devices working in parallel, we'll replicate our prompt as many times as devices we have. Then we'll perform inference on the 8 devices at once, each responsible for generating one image. Thus, we'll get 8 images in the same amount of time it takes for one chip to generate a single one.

After replicating the prompt, we obtain the tokenized text ids by invoking the `prepare_inputs` function of the pipeline. The length of the tokenized text is set to 77 tokens, as required by the configuration of the underlying CLIP Text model.

```
prompt = "A cinematic film still of Morgan Freeman starring as Jimi Hendrix, portrait, 40m
prompt = [prompt] * jax.device_count()
prompt_ids = pipeline.prepare_inputs(prompt)
prompt_ids.shape
```

```
(8, 77)
```

8.3.1 Replication and parallelization

Model parameters and inputs have to be replicated across the 8 parallel devices we have. The parameters dictionary is replicated using `flax.jax_utils.replicate`, which traverses the dictionary and changes the shape of the weights so they are repeated 8 times. Arrays are replicated using `shard`.

```
p_params = replicate(params)
```

```
prompt_ids = shard(prompt_ids)
prompt_ids.shape
```

```
(8, 1, 77)
```

That shape means that each one of the `8` devices will receive as an input a `jnp` array with shape `(1, 77)`. `1` is therefore the batch size per device. In TPUs with sufficient memory, it could be larger than `1` if we wanted to generate multiple images (per chip) at once.

We are almost ready to generate images! We just need to create a random number generator to pass to the generation function. This is the standard procedure in Flax, which is very serious and opinionated about random numbers – all functions that deal with random numbers are expected to receive a generator. This ensures reproducibility, even when we are training across multiple distributed devices.

The helper function below uses a seed to initialize a random number generator. As long as we use the same seed, we'll get the exact same results. Feel free to use different seeds when exploring results later in the notebook.

```
def create_key(seed=0):
    return jax.random.PRNGKey(seed)
```

We obtain a rng and then "split" it 8 times so each device receives a different generator. Therefore, each device will create a different image, and the full process is reproducible.

```
rng = create_key(0)
rng = jax.random.split(rng, jax.device_count())
```

JAX code can be compiled to an efficient representation that runs very fast. However, we need to ensure that all inputs have the same shape in subsequent calls; otherwise, JAX will have to recompile the code, and we wouldn't be able to take advantage of the optimized speed.

The Flax pipeline can compile the code for us if we pass `jit = True` as an argument. It will also ensure that the model runs in parallel in the 8 available devices.

The first time we run the following cell it will take a long time to compile, but subequent calls (even with different inputs) will be much faster. For example, it took more than a minute to compile in a TPU v2-8 when I tested, but then it takes about **`7s`** for future inference runs.

```
%%time

images = pipeline(prompt_ids, p_params, rng, jit=True)[0]
```

```
CPU times: user 56.2 s, sys: 42.5 s, total: 1min 38s
Wall time: 1min 29s
```

The returned array has shape `(8, 1, 512, 512, 3)`. We reshape it to get rid of the second dimension and obtain 8 images of `512 × 512 × 3` and then convert them to PIL.

```
images = images.reshape((images.shape[0] * images.shape[1], ) + images.shape[-3:])
images = pipeline.numpy_to_pil(images)
```

8.3.2 Visualization

Let's create a helper function to display images in a grid.

```
def image_grid(imgs, rows, cols):
    w,h = imgs[0].size
    grid = Image.new('RGB', size=(cols*w, rows*h))
    for i, img in enumerate(imgs): grid.paste(img, box=(i%cols*w, i//cols*h))
    return grid
```

```
image_grid(images, 2, 4)
```

8.4 Using different prompts

We don't have to replicate the *same* prompt in all the devices. We can do whatever we want: generate 2 prompts 4 times each, or even generate 8 different prompts at once. Let's do that!

First, we'll refactor the input preparation code into a handy function:

```
prompts = [
    "Labrador in the style of Hokusai",
    "Painting of a squirrel skating in New York",
    "HAL-9000 in the style of Van Gogh",
    "Times Square under water, with fish and a dolphin swimming around",
    "Ancient Roman fresco showing a man working on his laptop",
    "Close-up photograph of young black woman against urban background, high quality, boke
    "Armchair in the shape of an avocado",
    "Clown astronaut in space, with Earth in the background"
]
```

```
prompt_ids = pipeline.prepare_inputs(prompts)
prompt_ids = shard(prompt_ids)

images = pipeline(prompt_ids, p_params, rng, jit=True).images
images = images.reshape((images.shape[0] * images.shape[1], ) + images.shape[-3:])
```

```
images = pipeline.numpy_to_pil(images)

image_grid(images, 2, 4)
```

8.5 How does parallelization work?

We said before that the `diffusers` Flax pipeline automatically compiles the model and runs it in parallel on all available devices. We'll now briefly look inside that process to show how it works.

JAX parallelization can be done in multiple ways. The easiest one revolves around using the `jax.pmap` function to achieve single-program, multiple-data (SPMD) parallelization. It means we'll run several copies of the same code, each on different data inputs. More sophisticated approaches are possible, we invite you to go over the JAX documentation and the `pjit` pages to explore this topic if you are interested!

`jax.pmap` does two things for us: - Compiles (or `jits`) the code, as if we had invoked `jax.jit()`. This does not happen when we call `pmap`, but the first time the pmapped function is invoked. - Ensures the compiled code runs in parallel in all the available devices.

To show how it works we `pmap` the `_generate` method of the pipeline, which is the private method that runs generates images. Please, note that this method may be renamed or removed in future releases of `diffusers`.

```
p_generate = pmap(pipeline._generate)
```

After we use `pmap`, the prepared function `p_generate` will conceptually do the following: * Invoke a copy of the underlying function `pipeline._generate` in each device. * Send each device a different portion of the input arguments. That's what sharding is used for. In our case, `prompt_ids` has shape `(8, 1, 77, 768)`. This array will be split in `8` and each copy of `_generate` will receive an input with shape `(1, 77, 768)`.

We can code `_generate` completely ignoring the fact that it will be invoked in parallel. We just care about our batch size (`1` in this example) and the dimensions that make sense for our code, and don't have to change anything to make it work in parallel.

The same way as when we used the pipeline call, the first time we run the following cell it will take a while, but then it will be much faster.

```
%%time

images = p_generate(prompt_ids, p_params, rng)
images = images.block_until_ready()
images.shape
```

```
CPU times: user 1min 15s, sys: 18.2 s, total: 1min 34s
Wall time: 1min 15s
```

```
(8, 1, 512, 512, 3)
```

```
images.shape
```

```
(8, 1, 512, 512, 3)
```

We use `block_until_ready()` to correctly measure inference time, because JAX uses asynchronous dispatch and returns control to the Python loop as soon as it can. You don't need to use that in your code; blocking will occur automatically when you want to use the result of a computation that has not yet been materialized.

9 Exercise 1 Solution

With thanks to DeepMind for code here.

```
# Copyright 2018 The JAX Authors.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     https://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

"""A basic MNIST example using Numpy and JAX.

The primary aim here is simplicity and minimal dependencies.
"""

import array
import gzip
import os
from os import path
import struct
import urllib.request

import numpy as np


_DATA = "/tmp/jax_example_data/"
```

```
def _download(url, filename):
  """Download a url to a file in the JAX data temp directory."""
  if not path.exists(_DATA):
    os.makedirs(_DATA)
  out_file = path.join(_DATA, filename)
  if not path.isfile(out_file):
    urllib.request.urlretrieve(url, out_file)
    print(f"downloaded {url} to {_DATA}")


def _partial_flatten(x):
  """Flatten all but the first dimension of an ndarray."""
  return np.reshape(x, (x.shape[0], -1))


def _one_hot(x, k, dtype=np.float32):
  """Create a one-hot encoding of x of size k."""
  return np.array(x[:, None] == np.arange(k), dtype)


def mnist_raw():
  """Download and parse the raw MNIST dataset."""
  # CVDF mirror of http://yann.lecun.com/exdb/mnist/
  base_url = "https://storage.googleapis.com/cvdf-datasets/mnist/"

  def parse_labels(filename):
    with gzip.open(filename, "rb") as fh:
      _ = struct.unpack(">II", fh.read(8))
      return np.array(array.array("B", fh.read()), dtype=np.uint8)

  def parse_images(filename):
    with gzip.open(filename, "rb") as fh:
      _, num_data, rows, cols = struct.unpack(">IIII", fh.read(16))
      return np.array(array.array("B", fh.read()),
                      dtype=np.uint8).reshape(num_data, rows, cols)

  for filename in ["train-images-idx3-ubyte.gz", "train-labels-idx1-ubyte.gz",
                   "t10k-images-idx3-ubyte.gz", "t10k-labels-idx1-ubyte.gz"]:
    _download(base_url + filename, filename)

  train_images = parse_images(path.join(_DATA, "train-images-idx3-ubyte.gz"))
```

```
  train_labels = parse_labels(path.join(_DATA, "train-labels-idx1-ubyte.gz"))
  test_images = parse_images(path.join(_DATA, "t10k-images-idx3-ubyte.gz"))
  test_labels = parse_labels(path.join(_DATA, "t10k-labels-idx1-ubyte.gz"))

  return train_images, train_labels, test_images, test_labels


def mnist(permute_train=False):
  """Download, parse and process MNIST data to unit scale and one-hot labels."""
  train_images, train_labels, test_images, test_labels = mnist_raw()

  train_images = _partial_flatten(train_images) / np.float32(255.)
  test_images = _partial_flatten(test_images) / np.float32(255.)
  train_labels = _one_hot(train_labels, 10)
  test_labels = _one_hot(test_labels, 10)

  if permute_train:
    perm = np.random.RandomState(0).permutation(train_images.shape[0])
    train_images = train_images[perm]
    train_labels = train_labels[perm]

  return train_images, train_labels, test_images, test_labels
```

```
# Copyright 2018 The JAX Authors.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     https://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

"""A basic MNIST example using Numpy and JAX.

The primary aim here is simplicity and minimal dependencies.
"""
```

```
import time

import numpy.random as npr

from jax import jit, grad
from jax.scipy.special import logsumexp
import jax.numpy as jnp
from examples import datasets

def init_random_params(scale, layer_sizes):
    # Solution
    key = random.PRNGKey(0)
    # Split the PRNGKey into two new keys
    key1, key2 = random.split(key, 2)
    params = [(scale * random.normal(key1, (m, n)), scale * random.normal(key2, (n,)))
              for m, n in zip(layer_sizes[:-1], layer_sizes[1:])]
    return params


def predict(params, inputs):
  activations = inputs
  for w, b in params[:-1]:
    outputs = jnp.dot(activations, w) + b
    activations = jnp.tanh(outputs)

  final_w, final_b = params[-1]
  logits = jnp.dot(activations, final_w) + final_b
  return logits - logsumexp(logits, axis=1, keepdims=True)

def loss(params, batch):
  inputs, targets = batch
  preds = predict(params, inputs)
  return -jnp.mean(jnp.sum(preds * targets, axis=1))

def accuracy(params, batch):
  inputs, targets = batch
  target_class = jnp.argmax(targets, axis=1)
  predicted_class = jnp.argmax(predict(params, inputs), axis=1)
  return jnp.mean(predicted_class == target_class)
```

```
if __name__ == "__main__":
  layer_sizes = [784, 1024, 1024, 10]
  param_scale = 0.1
  step_size = 0.001
  num_epochs = 10
  batch_size = 128

  train_images, train_labels, test_images, test_labels = mnist()
  num_train = train_images.shape[0]
  num_complete_batches, leftover = divmod(num_train, batch_size)
  num_batches = num_complete_batches + bool(leftover)

  def data_stream():
    rng = npr.RandomState(0)
    while True:
      perm = rng.permutation(num_train)
      for i in range(num_batches):
        batch_idx = perm[i * batch_size:(i + 1) * batch_size]
        yield train_images[batch_idx], train_labels[batch_idx]
  batches = data_stream()

  # Solution
  # jit compiling the update function brings the most benefits;
  # it does the heavy lifting for the training loop and runs
  # many times
  @jit
  def update(params, batch):
    # Solution
    grads = grad(loss)(params, batch)
    return [(w - step_size * dw, b - step_size * db)
            for (w, b), (dw, db) in zip(params, grads)]

  params = init_random_params(param_scale, layer_sizes)
  for epoch in range(num_epochs):
    start_time = time.time()
    for _ in range(num_batches):
      params = update(params, next(batches))
    epoch_time = time.time() - start_time

    train_acc = accuracy(params, (train_images, train_labels))
    test_acc = accuracy(params, (test_images, test_labels))
```

```
    print(f"Epoch {epoch} in {epoch_time:0.2f} sec")
    print(f"Training set accuracy {train_acc}")
    print(f"Test set accuracy {test_acc}")
```

10 Exercise 2 Solution

```
import jax
from jax import numpy as jnp, random, lax, jit
from flax import linen as nn

X = jnp.ones((1, 10))
Y = jnp.ones((5,))

model = nn.Dense(features=5)

@jit
def predict(params):
  return model.apply({'params': params}, X)

@jit
def loss_fn(params):
  return jnp.mean(jnp.abs(Y - predict(params)))

@jit
def init_params(rng):
  mlp_variables = model.init({'params': rng}, X)
  return mlp_variables['params']

# Get initial parameters
params = init_params(jax.random.PRNGKey(42))
print("initial params", params)

# Run SGD.
for i in range(50):
  loss, grad = jax.value_and_grad(loss_fn)(params)
  print(i, "loss = ", loss, "Yhat = ", predict(params))
  lr = 0.03
  params = jax.tree_util.tree_map(lambda x, d: x - lr * d, params, grad)
```

11 Summary

In summary, this book has no content whatsoever.

References