A Comparative Introduction to Deep Learning Frameworks: TensorFlow, PyTorch and JAX

JAX

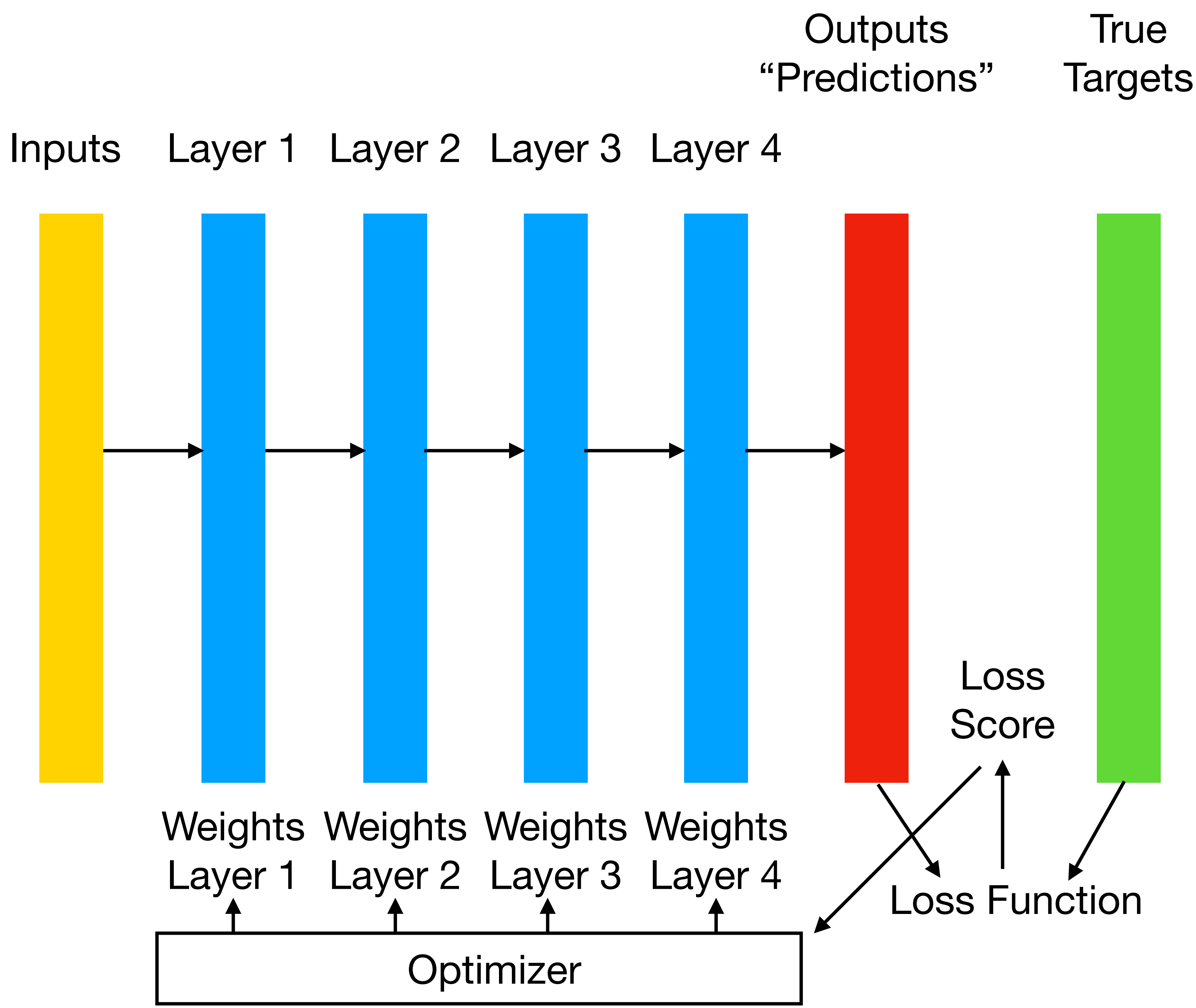
Mirco Musolesi

mircomusolesi@acm.org

JAX

- JAX is a library that enables transformations of array manipulating programs with a NumPy-like array.
- JAX is developed by Google. Current version is 0.3.24.
- One way of seeing it is to consider it as a differentiable NumPy.
- The API itself is the same of NumPy.
- It is designed for being used with accelerators.

Deep Neural Networks



JAX and XLA

- JAX uses XLA to compile and run NumPy code on accelerators, such as GPUs and TPUs.
- Compilation takes place under the hood by default.
  - Libraries are just-in-time compiled and executed.
- JAX also lets to just-in-time compile user-defined functions into XLA-optimised kernels using a predefined function.

XLA

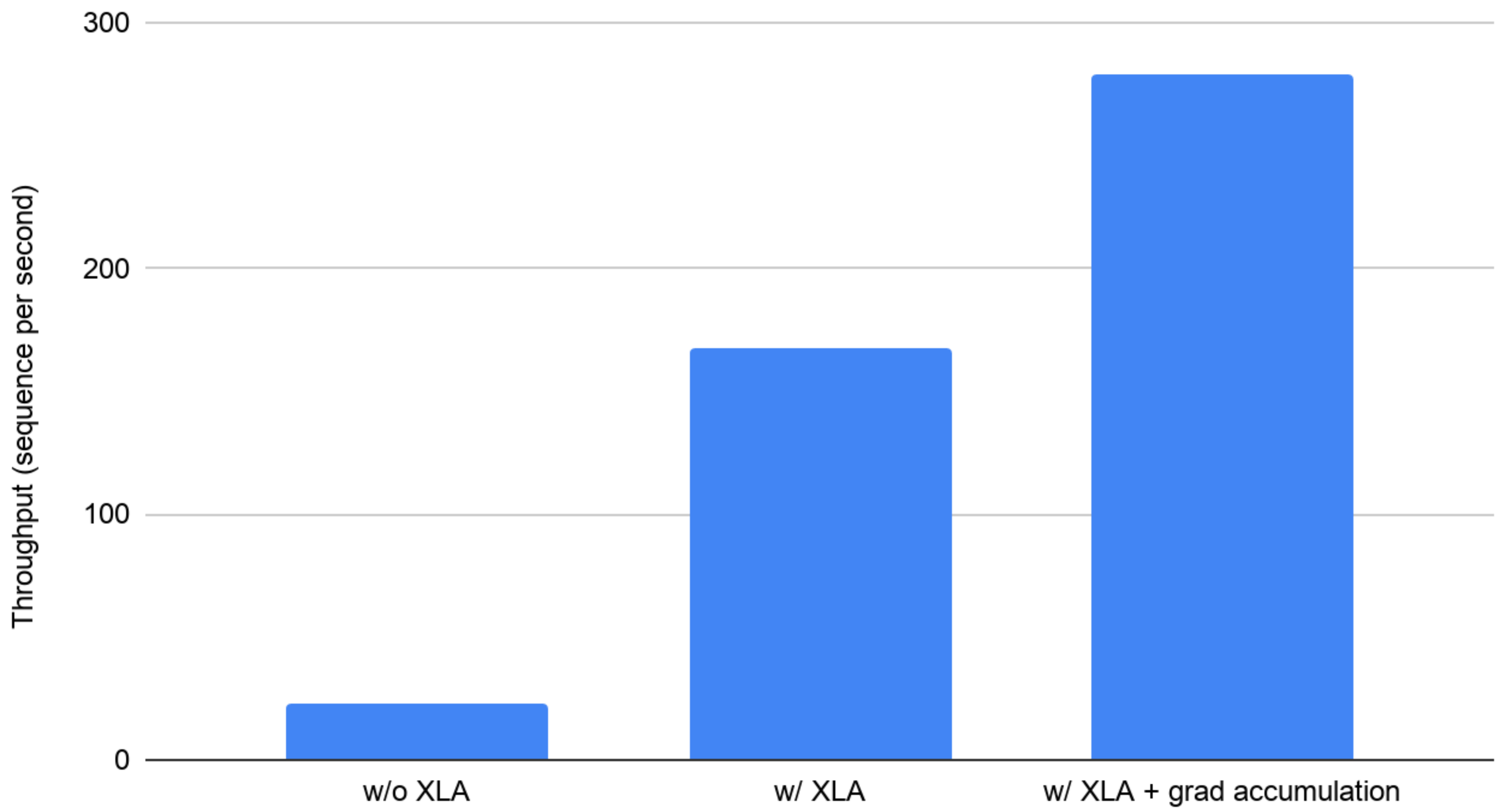
- XLA (Accelerated Linear Algebra) is a domain-specific compiler with linear algebra that can accelerate models with potentially no source changes.

- The results are in memory and speed.

- Example: The results are improvements in speed and memory usage: e.g. in BERT MLPerf submission using 8 Volta V100 GPUs using XLA has achieved a ~7x performance improvement and ~5x batch size improvement.

  - MLPerf is a standard benchmark for ML.

XLA



Source: TensorFlow XLA Tutorial (https://www.tensorflow.org/xla)

Decorators

- A decorator is a Python design pattern that allows a user to add new functionality to an existing object without modifying its structure.
- Let us start from underlining the fact that Python allows a nested function to access the outer scope of an enclosing function.

Decorators

- Let’s us consider the following definition of a function

```
def A(a_text):

   def B()

      print(a_text)

   B()
```

- We get the following behaviour:

```
>>> A(“some text”)

some text
```

Decorators

- Let us consider the creation of a simple decorator.
    - We define a wrapper inside an enclosed function.
- Example:

```
def uppercase_decorator(function):

   def wrapper()

       func = function()

       make_uppercase = func.upper()

       return make_uppercase

    return wrapper
```

Decorators

- The decorator function takes a function as an argument.
- We will have to define a function and pass it to the decorator.
- Let's us consider the following function as an example:

```
def say_hello_world():

    return 'hello world!'
```

- We then apply a decorator and we call the decorated function:

```
>>>decorate = uppercase_decorator(say_hello_world)

>>>decorate()

'HELLO WORLD!"
```

Decorators

- Python provides an easier way to apply decorators, i.e., it is sufficient to use the symbol `@` before the function we want to decorate.

- For example we can apply the decorator as in the example before in this way:

```
@uppercase_decorator

def say_hello_world:

   return ‘hello world!’
```

- And then call the function:

```
>>>say_hello_world

‘HELLO WORLD!’
```

jax.jit

- `jax.jit` sets up a function for just-in-time compilation using XLA.
- JAX runs transparently on GPUs and TPUS.
- You can jit-compile a function by using

```
relu_jit = jit(relu)
```

- Or indeed we can use decorators!

`jax.jit`

- Remember that by default, JAX executes operations sequentially.
- Using JIT compilation decorators, sequences of operations can be optimised (and run in parallel).
- Not all the JAX code can be JIT compiled, since it requires array shapes to be static and known at a compile time.

`jax.jit`

- JIT works by *tracing* a function.
- JIT is based on tracer objects that are used to extract the sequence of operations that are specified by the function.
  - Basic traces are sort of stand-ins that encode the shape and the type of the arrays, but they are agnostic to the values.
  - The recorded sequence of computations are applied with XLA to new inputs with the same `shape` and `type`, without re-executing the Python code.
  - When we call the compiled function on matching inputs, no re-compilation is required.

jax.grad()

- **`grad()`** provides the users with automatic differentiation:

```
from jax import grad

print('The value of the derivate of exp for 1 is', jnp.exp(1.0))

grad_exp = grad(jnp.exp)

print('The value of the derivate of exp for 1 is’,grad_exp(1.0))
```

- The output is:

```
The value of the derivate of exp for 1 is 2.7182817

The value of the derivate of exp for 1 is 2.7182817
```

`jax.grad()`

- We can use the grad function with its `argnums` argument to differentiate a function with respect to positional arguments.

- For example:

```
w_grad = grad(loss, argnums=0)(w, b)
```

returns the gradient with respect to `w` (note that `argnums=0`, so in theory in this case, specifying it in the function is actually redundant).

- And:

```
b_grad = grad(loss, 1)(w, b)
```

returns the gradient with respect to `b`.

`jax.vmap`

- `vmap` is a vectorising map. It creates a function which maps a function in input over the argument axes.

- Vectorising means that it allows to compute the output of a function in parallel over some axis of the input.

- It has two key arguments. `in_axes`, which is a tuple that indicates which axes of the function's arguments should be parallelised and `out_axes`, which specifies which axes of the function's output we need to parallelise over.

A Full MNIST Example

- We are going to use one of the example that is provided by the JAX documentation and we will comment about the key parts.

A Full Example using MNIST

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

Helper Functions

```
# A helper function to randomly initialize weights and biases
# for a dense neural network layer
def random_layer_params(m, n, key, scale=1e-2):
  w_key, b_key = random.split(key)
  return scale * random.normal(w_key, (n, m)), scale * random.normal(b_key,
(n,))

# Initialize all layers for a fully-connected neural network with sizes
"sizes"
def init_network_params(sizes, key):
  keys = random.split(key, len(sizes))
  return [random_layer_params(m, n, k) for m, n, k in zip(sizes[:-1],
sizes[1:], keys)]

layer_sizes = [784, 512, 512, 10]
step_size = 0.01
num_epochs = 10
batch_size = 128
n_targets = 10
params = init_network_params(layer_sizes, random.PRNGKey(0))
```

Predict Function

```
from jax.scipy.special import logsumexp

def relu(x):
  return jnp.maximum(0, x)

def predict(params, image):
  # per-example predictions
  activations = image
  for w, b in params[:-1]:
    outputs = jnp.dot(w, activations) + b
    activations = relu(outputs)

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

Log-probabilities and Cross-entropy Loss

- Let's recall the definition of cross-entropy loss:

$$\mathscr{L} = -\sum_{c=1}^{M} y_i log(p_i)$$

- Let's now consider the logits (before the softmax). We indicate them with x_j.

- The logits will go through a softmax function:

$$Softmax(x_j) = \frac{e^{x_j}}{\sum_{i=1}^{M} e^{x_i}} = p_j$$

- Let's know consider the log values of the probabilities:

$$logp_j = log \frac{e^{x_j}}{\sum_{i=1}^{M} e^{x_i}} = loge^{x_j} - log \sum_{i=1}^{M} e^{x_i} = x_j - \sum_{i=1}^{M} e^{x_i}$$

- $\sum_{i=1}^{M} e^{x_i}$ is implemented in Python by `scipy.special.logsumexp`.

Code

```
# This works on single examples
random_flattened_image = random.normal(random.PRNGKey(1), (28 * 28,))
preds = predict(params, random_flattened_image)
print(preds.shape)

random_flattened_images = random.normal(random.PRNGKey(1), (10, 28 * 28))
try:
  preds = predict(params, random_flattened_images)
except TypeError:
  print('Invalid shapes!')

# Let's upgrade it to handle batches using `vmap`

# Make a batched version of the `predict` function
batched_predict = vmap(predict, in_axes=(None, 0))

# `batched_predict` has the same call signature as `predict`
batched_preds = batched_predict(params, random_flattened_images)
print(batched_preds.shape)
```

Code

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

def accuracy(params, images, targets):
  target_class = jnp.argmax(targets, axis=1)
  predicted_class = jnp.argmax(batched_predict(params, images), axis=1)
  return jnp.mean(predicted_class == target_class)

def loss(params, images, targets):
  preds = batched_predict(params, images)
  return -jnp.mean(preds * targets)

@jit
def update(params, x, y):
  grads = grad(loss)(params, x, y)
  return [(w - step_size * dw, b - step_size * db)
          for (w, b), (dw, db) in zip(params, grads)]
```

Code

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

def accuracy(params, images, targets):
  target_class = jnp.argmax(targets, axis=1)
  predicted_class = jnp.argmax(batched_predict(params, images), axis=1)
  return jnp.mean(predicted_class == target_class)

def loss(params, images, targets):
  preds = batched_predict(params, images)
  return -jnp.mean(preds * targets)

@jit
def update(params, x, y):
  grads = grad(loss)(params, x, y)
  return [(w - step_size * dw, b - step_size * db)
          for (w, b), (dw, db) in zip(params, grads)]
```

Code

```
import tensorflow as tf
# Ensure TF does not see GPU and grab all GPU memory.
tf.config.set_visible_devices([], device_type='GPU')

import tensorflow_datasets as tfds

data_dir = '/tmp/tfds'

# Fetch full datasets for evaluation
# tfds.load returns tf.Tensors (or tf.data.Datasets if batch_size != -1)
# You can convert them to NumPy arrays (or iterables of NumPy arrays) with tfds.dataset_as_numpy
mnist_data, info = tfds.load(name="mnist", batch_size=-1, data_dir=data_dir, with_info=True)
mnist_data = tfds.as_numpy(mnist_data)
train_data, test_data = mnist_data['train'], mnist_data['test']
num_labels = info.features['label'].num_classes
h, w, c = info.features['image'].shape
num_pixels = h * w * c

# Full train set
train_images, train_labels = train_data['image'], train_data['label']
train_images = jnp.reshape(train_images, (len(train_images), num_pixels))
train_labels = one_hot(train_labels, num_labels)

# Full test set
test_images, test_labels = test_data['image'], test_data['label']
test_images = jnp.reshape(test_images, (len(test_images), num_pixels))
test_labels = one_hot(test_labels, num_labels)
```

Code

```
import time

def get_train_batches():
  # as_supervised=True gives us the (image, label) as a tuple instead of a dict
  ds = tfds.load(name='mnist', split='train', as_supervised=True, data_dir=data_dir)
  # You can build up an arbitrary tf.data input pipeline
  ds = ds.batch(batch_size).prefetch(1)
  # tfds.dataset_as_numpy converts the tf.data.Dataset into an iterable of NumPy arrays
  return tfds.as_numpy(ds)

for epoch in range(num_epochs):
  start_time = time.time()
  for x, y in get_train_batches():
    x = jnp.reshape(x, (len(x), num_pixels))
    y = one_hot(y, num_labels)
    params = update(params, x, y)
  epoch_time = time.time() - start_time

  train_acc = accuracy(params, train_images, train_labels)
  test_acc = accuracy(params, test_images, test_labels)
  print("Epoch {} in {:0.2f} sec".format(epoch, epoch_time))
  print("Training set accuracy {}".format(train_acc))
  print("Test set accuracy {}".format(test_acc))
```

Code

```
Epoch 0 in 10.44 sec
Training set accuracy 0.9252499938011169
Test set accuracy 0.9271000027656555
Epoch 1 in 5.16 sec
Training set accuracy 0.9428166747093201
Test set accuracy 0.9409999847412109
Epoch 2 in 4.65 sec
Training set accuracy 0.9532666802406311
Test set accuracy 0.9512999653816223
Epoch 3 in 4.57 sec
Training set accuracy 0.9598667025566101
Test set accuracy 0.9557999968528748
Epoch 4 in 4.52 sec
Training set accuracy 0.9650833606719971
Test set accuracy 0.960099995136261
Epoch 5 in 8.04 sec
Training set accuracy 0.9691833257675171
Test set accuracy 0.9629999995231628
Epoch 6 in 10.43 sec
Training set accuracy 0.9726333618164062
Test set accuracy 0.9651999473571777
Epoch 7 in 10.43 sec
Training set accuracy 0.9754000306129456
Test set accuracy 0.9666999578475952
Epoch 8 in 6.69 sec
Training set accuracy 0.9779166579246521
Test set accuracy 0.9679999947547913
Epoch 9 in 6.79 sec
Training set accuracy 0.9804666638374329
Test set accuracy 0.9691999554634094
```

A Note on Pseudo Random Number Generators (PRNGs) in JAX

```
# This works on single examples
random_flattened_image = random.normal(random.PRNGKey(1), (28 * 28,))
preds = predict(params, random_flattened_image)
print(preds.shape)

random_flattened_images = random.normal(random.PRNGKey(1), (10, 28 * 28))
try:
  preds = predict(params, random_flattened_images)
except TypeError:
  print('Invalid shapes!')

# Let's upgrade it to handle batches using `vmap`

# Make a batched version of the `predict` function
batched_predict = vmap(predict, in_axes=(None, 0))

# `batched_predict` has the same call signature as `predict`
batched_preds = batched_predict(params, random_flattened_images)
print(batched_preds.shape)
```

A Note on Pseudo Random Number Generators (PRNGs) in JAX

- There are potential problems related to the use of the standard Pseudo Random Number Generators (PRNGs) offered by NumPy (`numpy.random`).
- Example:

```
import numpy as np

    np.random.seed(0)

    random_numbers = np.random.uniform(size = 2)

    print random_numbers
```

- This returns:

```
[0.7135691, 0.5401991]
```

- If you re-run the program again (with the same seed), you get the same result.
    - This is essential for reproducibility.

A Note on Pseudo Random Number Generators (PRNGs) in JAX

- However, if you a program that is running on a parallel architecture, such as a GPU, this is not the case anymore.
- In fact, PRNGs in NumPy are based on a global state. It is not possible to guarantee the usual *sequential equivalent guarantee* (same numbers generated by the same seed)

A Note on Pseudo Random Number Generators (PRNGs) in JAX

- Foe example, if we use JIT for parallelising the execution, the order for this is not guaranteed:

```
import numpy as np

np.random.seed(0)

def f1():

   return np.random.uniform()

def f2()

   return np.random.uniform()

def f3():

   return f1()+f2()
```

A Note on Pseudo Random Number Generators (PRNGs) in JAX

- To avoid this issue, JAX does not use a global state.
- Instead, random functions "consume" the state, which is referred as a key:

```
from jax import random

key = random.PRNGKey(42)
```

which returns:

```
[0, 42]
```

- Random keys in JAX corresponds essentially to random seeds. However, instead of setting once as in NumPy, any call of a random function in JAX requires a key to be specified.

A Note on Pseudo Random Number Generators (PRNGs) in JAX

- In order to generate different and independent samples, we must “split” the key ourselves whenever we call a random function as follows:

```
new_key, sub_key = random.split(key)

del key

sample = random.uniform(sub_key)

del sub_key

key = new_key
```

- `split()` is a deterministic function that converts one key into several keys. We keep one of the outputs as the `new_key`.
- We use a unique extra key (`sub_key`) as input once and then discard it.
- If we ant to get another sample from the normal distribution we will split key again, etc.

JAX Ecosystem

- I presented JAX from a close-to-the-metal point of view.
- However, there are many libraries that are based on it and they simplify the development of complex deep learning architectures.
- Examples include Flax, Haiku, RLax (for Reinforcement Learning) and Jraph (for graph neural neural networks), etc.
- Huggingface also maintains a JAX library, which also provides support for transformers.

References

- JAX Documentation.
- Autograd Documentation.
- DeepMind JAX pages.

- Some of the material in these slides has been taken from the official JAX documentation.