# A comparison of neural network frameworks

Erik Spence

SciNet HPC Consortium

18 January 2023

SciNet compute • calcul CANADA

# Today's slides

You can get the code and slides for today's class at the SciNet Education site.

`https://scinet.courses/1274`

SciNet compute • calcul CANADA

# Today's talk

Today we will review the implementation of a simple convolutional neural network, using two neural network frameworks:

- PyTorch Lightning,
- Keras/TensorFlow.

The goal of this lecture is to examine the ease of implementation, the amount of code needed to build the model, the amount of time needed to train the model, and the ease of parallelization.

Please note that we won't be discussing neural networks in general in this class. Nor will we be focusing on Python programming in general.

All results in this talk were performed on Mist, SciNet's GPU cluster (Power9 architecture, V100 GPUs).

# Neural network frameworks

Nobody codes neural networks by hand, but rather neural network 'frameworks' are used. Why would we do that?

- Coding your own networks from scratch can be tricky, and is error-prone.
- Neural network (NN) frameworks have been specifically designed to solve NN problems.
- Python, of course, is not a high-performance language.
- The neural networks which are built using frameworks are compiled before being used, thus being much faster than Python.
- The NN frameworks are also designed to use GPUs, which make things significantly faster than just using CPUs.

The training of neural networks is particularly well suited to GPUs. We will only consider the training of networks on GPUs in this talk.

SciNet compute • calcul CANADA

# Keras

Keras is a very user-friendly neural network framework.

- More accurately, it's an API standard for creating neural networks.
- Designed for the fast development of networks.
- Initially released in 2015.
- Up until version 2.3 Keras could be run on top of any number of 'back ends': Theano, CNTK, MXNet, TypeScript, JavaScript, PlaidML, Scala, CoreML, and others.
- This allowed Keras to simply be a high-level interface to any number of neural network frameworks.
- As of version 2.4 TensorFlow is the only back end that is supported (current release is 2.11). TensorFlow now ships with Keras.

The easiest way to get Keras is to just install TensorFlow.

SciNet compute • calcul CANADA

# PyTorch Lightning

PyTorch Lightning is a wrapper around PyTorch.

- Initially released in 2019.
- Simplifies the construction and training of PyTorch neural networks, which can be quite verbose.
- Contains all manner of builtin goodies:
  - half-precision (16-bit floats),
  - suggestions and diagnostics,
  - hooks to use Tensorboard,
- Very easy to parallelize,
- As well as all kinds of other goodies.

PyTorch Lightning is now a very commonly-used interface to PyTorch, and PyTorch is the dominant academic neural network framework.

# CIFAR-10 data set

A commonly-used data set for testing image classification networks is the CIFAR-10 data set.

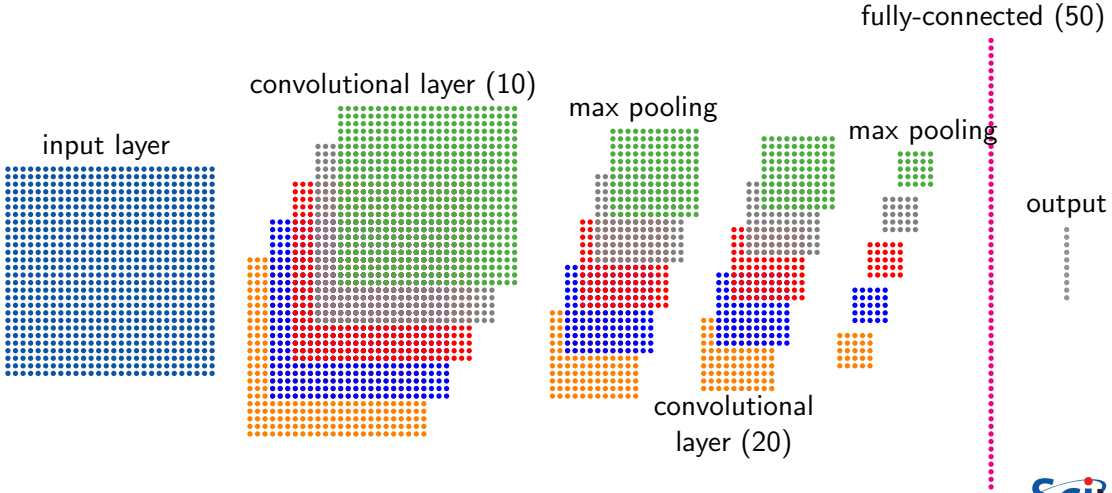
- A data set of images, in 10 different classes (airplane, automobile, bird, cat, deer, dog, frog, horse, ship, truck).
- Contains 60000 images, with 6000 images in each class.
- Each image is colour, 32 x 32 pixels, 3 colours.
- This data set is so common that it is built into both Keras and PyTorch.

I originally intended to use the MNIST data set for this talk, but switched to CIFAR-10 as the data is larger, slightly more challenging, but just as easy to implement.

That being said, this is a simple data set. The hope for this talk is to keep things relatively simple, so that the key differences in the frameworks can bee seen.

SciNet compute • calcul CANADA

# The network


input layer
convolutional layer (10)
max pooling
convolutional
layer (20)
max pooling
fully-connected (50)
output

# PyTorch Lightning, data loading

```
# torch_settings.py
import torch, os
import torch.utils.data as tud, torchvision.datasets as tvd, torchvision.transforms as tvt

n_epochs = 300;    learning_rate = 0.01

PATH_DATASETS = os.environ.get("PATH_DATASETS", ".")
BATCH_SIZE = 256 if torch.cuda.is_available() else 64

# Device configuration.
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

train_data = tvd.CIFAR10(PATH_DATASETS, train = True, download = True, transform = tvt.ToTensor())
train_loader = tud.DataLoader(train_data, batch_size = BATCH_SIZE, num_workers = 2)

test_data = tvd.CIFAR10(PATH_DATASETS, train = False, download = True, transform = tvt.ToTensor())
test_loader = tud.DataLoader(test_data, batch_size = BATCH_SIZE, num_workers = 2)
```

# PyTorch Lightning

Here we begin by importing some packages. These packages are the same as needed when using PyTorch, with the except of the pytorch_lightning package.

```
# run_lightning_cifar10.py

import torch

import torch.nn as nn
import torch.nn.functional as F
import torch.optim as to

import torchmetrics as tm

from torch_settings import *

import pytorch_lightning as pl
```

# PyTorch Lightning, continued

```
class CIFAR10Model(pl.LightningModule):
  def __init__(self):
    super(CIFAR10Model, self).__init__()
    self.conv1 = nn.Conv2d(3, 10, kernel_size = 5)
    self.pool1 = nn.MaxPool2d(2, 2)
    self.conv2 = nn.Conv2d(10, 20, kernel_size = 5)
    self.pool2 = nn.MaxPool2d(2, 2)
    self.fc1 = nn.Linear(5 * 5 * 20, 50)
    self.fc2 = nn.Linear(50, 10)

    self.accuracy = tm.Accuracy('multiclass', 10)
    self.cost = nn.CrossEntropyLoss()

  def forward(self, x):
    x = self.pool1(F.relu(self.conv1(x)))
    x = self.pool2(F.relu(self.conv2(x)))
    x = x.view(-1, 500)
    return self.fc2(F.relu(self.fc1(x)))
```

```
  def configure_optimizers(self):
    return to.SGD(self.parameters(),
      lr = learning_rate)

  def training_step(self, batch, batch_nb):
    x, y = batch;   y_hat = self(x)
    loss = self.cost(y_hat, y)
    acc = self.accuracy(y_hat, y)
    return {'loss': loss, 'accuracy': acc}

  def training_epoch_end(self, outputs):
    acc = sum(output['accuracy']
      for output in outputs) / len(outputs)
    loss = sum(output['loss']
      for output in outputs)
    print('Epoch: {} Loss: {:.3f} Acc: {:.3f}'
      .format(self.current_epoch, loss, acc))
    self.log_dict({'loss': loss, 'acc': acc})
```

# PyTorch Lightning, continued more

```
if __name__ == "__main__":

  # Initialize the model.
  model = CIFAR10Model()

  # Initialize a trainer.
  trainer = pl.Trainer(
    accelerator = 'auto',
    gpus = 1,
    max_epochs = n_epochs,
    enable_progress_bar = False)

  trainer.fit(model, train_loader)
  trainer.test(model, test_loader)
```

```
ejspence@mist-login01 ~>
ejspence@mist-login01 ~> python run_lightning_cifar10.py
GPU available: True, used: True
TPU available: False, using: 0 TPU cores
IPU available: False, using: 0 IPUs
⋮
```

PyTorch Lightning will automatically detect not only GPUs, but TPUs and other accelerators.

# PyTorch Lightning, continued some more

PyTorch Lightning automatically prints out the summary of the model.

```
  ⋮
| Name | Type | Params
------------------------------------------------
0 | conv1 | Conv2d | 760
1 | pool1 | MaxPool2d | 0
2 | conv2 | Conv2d | 5.0 K
3 | pool2 | MaxPool2d | 0
4 | fc1 | Linear | 25.1 K
5 | fc2 | Linear | 510
6 | accuracy | MulticlassAccuracy | 0
7 | cost | CrossEntropyLoss | 0
------------------------------------------------
31.3 K Trainable params
0 Non-trainable params
31.3 K Total params
  ⋮
```

# PyTorch Lightning, continued even more

```
⋮
Epoch:  0, Training Loss:  451.516, Accuracy:  0.114
Epoch:  1, Training Loss:  450.179, Accuracy:  0.134
Epoch:  2, Training Loss:  447.250, Accuracy:  0.157
Epoch:  3, Training Loss:  435.601, Accuracy:  0.185
Epoch:  4, Training Loss:  409.612, Accuracy:  0.239
⋮
Epoch:  295, Training Loss:  149.082, Accuracy:  0.736
Epoch:  296, Training Loss:  148.983, Accuracy:  0.737
Epoch:  297, Training Loss:  148.739, Accuracy:  0.738
Epoch:  298, Training Loss:  148.539, Accuracy:  0.737
Epoch:  299, Training Loss:  148.367, Accuracy:  0.738
Testing Accuracy:  0.624
ejspence@mist-login01 ~>
```

# Keras

```
import tensorflow.keras.models as km
import tensorflow.keras.layers as kl
import tensorflow.keras.optimizers as ko
import tensorflow.keras.utils as ku
import tensorflow.keras.datasets as kd

import tensorflow as tf
import tensorflow.config as tfc

def build_cifar10_data(data, labels, batch_size):
  def gen():
    for image, label in zip(data, labels):
      yield image.astype('float32') / 255, ku.to_categorical(label, 10)
  ds = tf.data.Dataset.from_generator(gen, output_types = (tf.float32, tf.int32),
    output_shapes = ((28, 28), (10)))
  return ds.batch(batch_size)
```

Here we build a generator to pass the data to the model.

# Keras, continued

Building the model does not require the use of objects in Keras, though you can certainly go that route.

Here we simply create the model within a function.

```
def get_model(numfm, numnodes, input_shape = (32, 32, 3),
  output_size = 10):

  model = km.Sequential()

  model.add(kl.Conv2D(numfm, kernel_size = (5, 5),
                      activation = 'relu', input_shape = input_shape)
  model.add(kl.MaxPooling2D(pool_size = (2, 2), strides = (2, 2)))

  model.add(kl.Conv2D(2 * numfm, kernel_size = (5, 5),
                      activation = 'relu'))
  model.add(kl.MaxPooling2D(pool_size = (2, 2), strides = (2, 2)))

  model.add(kl.Flatten())
  model.add(kl.Dense(numnodes, activation = 'relu'))
  model.add(kl.Dense(output_size, activation = 'softmax'))
  return model
```

# Keras, continued more

The model compilation step is used to specify the loss and optimization functions.

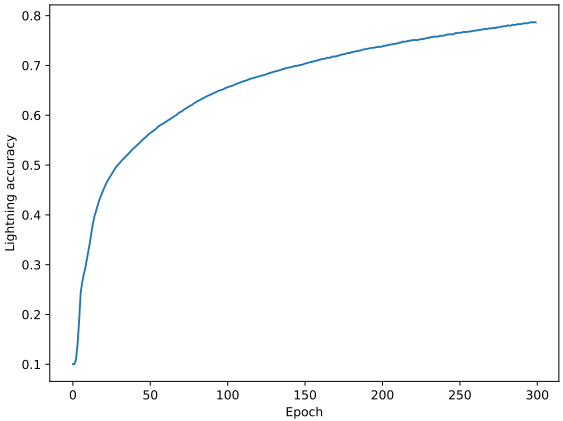
The fitting of the model is done in a single command.

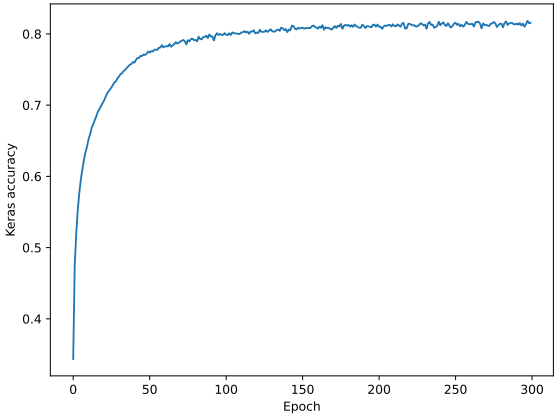
```
if __name__ == "__main__":
  n_epochs = 300
  learning_rate = 0.01
  BATCH_SIZE = 256 if len(tfc.list_physical_devices('GPU')) == 0
                   else 64
  (x_train, y_train), (x_test, y_test) = kd.cifar10.load_data()

  training_data = build_cifar10_data(x_train, y_train, 10)
  testing_data = build_cifar10_data(x_test, y_test, 20)

  model = get_model(10, 50)

  model.compile(optimizer = ko.SGD(learning_rate = learning_rate),
                metrics = ['accuracy'],
                loss = 'categorical_crossentropy')
  model.fit(training_data, epochs = n_epochs, verbose = 2,
            batch_size = BATCH_SIZE)
  print(model.evaluate(testing_data))
```

# Keras, continued more

```
ejspence@mist-login01 ~>
ejspence@mist-login01 ~> python run_keras_cifar10.py
Epoch 1/300
5000/5000 - 57s - loss:  1.8057 - accuracy:  0.3437 - 57s/epoch - 11ms/step
Epoch 2/300
5000/5000 - 20s - loss:  1.4611 - accuracy:  0.4759 - 20s/epoch - 4ms/step
⋮
Epoch 298/300
5000/5000 - 19s - loss:  0.5180 - accuracy:  0.8184 - 19s/epoch - 4ms/step
Epoch 299/300
5000/5000 - 19s - loss:  0.5310 - accuracy:  0.8150 - 19s/epoch - 4ms/step
Epoch 300/300
5000/5000 - 19s - loss:  0.5258 - accuracy:  0.8156 - 19s/epoch - 4ms/step
500/500 - 3s - loss:  3.5750 - accuracy:  0.5329 - 3s/epoch - 5ms/step
[3.575030565261841, 0.5328999757766724]
ejspence@mist-login01 ~>
```

# Training speed




Single-GPU training accuracy. The rate of training improvement for Lightning is lower than Keras.

# PyTorch Lightning, multi-GPU

```
def training_step(self, batch, batch_nb):
  x, y = batch
  y_hat = self(x)
  loss = self.cost(y_hat, y)
  acc = self.accuracy(y_hat, y)
  return {'loss': loss, 'accuracy': acc,
    'y': y, 'y_hat': y_hat}
```

The metrics calculated at the end of each epoch need to be reduced to the head GPU before being logged or printed.

The 'self.trainer.is_global_zero' flag indicates the head GPU.

```
def training_epoch_end(self, outputs):
  results = torch.zeros((10, 10)).to(self.device)
  for out in outputs:
    for label, pred in zip(out['y'], out['y_hat']):
      results[int(label), int(pred)] += 1
  torch.distributed.reduce(results, 0,
    torch.distributed.ReduceOp.SUM)
  acc = results.diag().sum() / results.sum()

  loss = sum(output['loss'] for output in outputs)
  torch.distributed.reduce(loss, 0,
    torch.distributed.ReduceOp.SUM)

  if self.trainer.is_global_zero:
    print('Epoch: {} Loss: {:.3f} Acc: {:.3f}'
      .format(pyself.current_epoch, loss, acc))
    self.log_dict({'loss': loss, 'acc': acc},
                  rank_zero_only = True)
```

# PyTorch Lightning, multi-GPU, continued

Implementing a data-parallelized model across multiple GPUs only requires two more pieces:

- The 'ddp' for the accelerator indicates Distributed Data-Parallel. This is for the data parallelism.
- Specifying "gpus = 2". Either the number of GPUs, or the indices of the GPUs.

```
if __name__ == "__main__":

  # Initialize the model.
  model = CIFAR10Model()

  # Initialize a trainer.
  trainer = pl.Trainer(
    accelerator = 'ddp',
    gpus = 2,
    max_epochs = n_epochs,
    enable_progress_bar = False)

  trainer.fit(model, train_loader)
  test_results = trainer.test(model, test_loader)
  if trainer.is_global_zero:
    print(trainer.results)
    print(test_results)
```

# Keras, multi-GPU

To enable multi-GPU support using Keras, one must create a parallel 'strategy', and then create and compile the model under that strategy.

In this case we again use data parallelism.

After that, the model is trained in the usual way.

```
if __name__ == "__main__":
⋮
  strategy = tf.distribute.MirroredStrategy()

  with strategy.scope():
    model = get_model(10, 50)
    model.compile(optimizer = ko.SGD(learning_rate = learning_rate),
                  metrics = ['accuracy'],
                  loss = 'categorical_crossentropy')

  model.fit(training_data, epochs = n_epochs, verbose = 2,
            batch_size = BATCH_SIZE)
  print(model.evaluate(testing_data))
```

# Average results

| | 1 GPU | | | 2 GPUs | | |
|---|---|---|---|---|---|---|
| | Time | Training Acc | Test Acc | Time | Training Acc | Test Acc |
| Lightning | 0:18:32 | 76.8 | 64.1 | 0:31:14 | 69.1 | 63.2 |
| Keras | 1:42:32 | 82.2 | 54.5 | 1:49:54 | 81.0 | 53.7 |

Single GPU times have been averaged over 5 runs; two-GPU over 6 runs.

Some notes:

- Though Lightning is faster to train per epoch, the rate of training improvement is slower.
- The neural network being considered over-fits the data. The testing accuracies are included here for completeness.
- The problem being considered is too small to accurately reflect the utility of using multiple GPUs. Again, the multi-GPU results are only included for completeness.

SciNet compute • calcul CANADA

# Final comparison

| | Lightning | Keras |
|---|---|---|
| Amount of code | More | Less |
| Ease of parallelization | Easy | Easy |
| Speed of training | Fast | Slow |

At the end of the day, the training-speed advantages of PyTorch Lightning arguably outweigh the coding convenience of Keras, at least for this problem.

SciNet compute • calcul CANADA