PyTorch vs TensorFlow — spotting the difference

Kirill Dubovikov Jun 20, 2017 · 9 min read

In this post I want to explore some of the key similarities and differences between two popular deep learning frameworks: PyTorch and TensorFlow. Why those two and not the others? There are many deep learning frameworks and many of them are viable tools, I chose those two just because I was interested in comparing them specifically.

and production needs. Its closed-source predecessor is called DistBelief.

PyTorch is a cousin of lua-based Torch framework which was developed and used at Facebook. However, PyTorch is not a simple set of wrappers to support popular language, it was rewritten and tailored to be fast and feel native.

The best way to compare two frameworks is to code something up in both of them. I've written a companion jupyter notebook for this post and you can get it here. All code will be provided in the post too.

First, let's code a simple approximator for the following function in both frameworks:

$$f(x) = x^{\phi}$$

We will try to find unknown parameter *phi* given data *x* and function values *f(x)*. Yes, using stochastic gradient descent for this is an overkill and analytical solution may be found easily, but this problem will serve our purpose well as a simple example.

We will solve this with PyTorch first:

```
import torch
from torch.autograd import Variable
import numpy as np

def rmse(y, y_hat):
    """Compute root mean squared error"""
    return torch.sqrt(torch.mean((y - y_hat).pow(2).sum()))

def forward(x, e):
    """Forward pass for our fuction"""
    return x.pow(e.repeat(x.size(0)))

# Let's define some settings
n = 100 # number of examples
```

Get started Open in app

```
# Model definition
x = Variable(torch.rand(n) * 10, requires_grad=False)

# Model parameter and it's true value
exp = Variable(torch.FloatTensor([target_exp]), requires_grad=False)
exp_hat = Variable(torch.FloatTensor([4]), requires_grad=True) # just some starting value, could
y = forward(x, exp)

# a couple of buffers to hold parameter and loss history
loss_history = []
exp_history = []

# Training loop
for i in range(0, 200):
    print("Iteration %d" % i)

    # Compute current estimate
    y_hat = forward(x, exp_hat)

    # Calculate loss function
    loss = rmse(y, y_hat)

    # Do some recordings for plots
    loss_history.append(loss.data[0])
    exp_history.append(y_hat.data[0])

    # Compute gradients
    loss.backward()

    print("loss = %s" % loss.data[0])
    print("exp = %s" % exp_hat.data[0])

    # Update model parameters
    exp_hat.data -= learning_rate * exp_hat.grad.data
    exp_hat.grad.data.zero_()
```

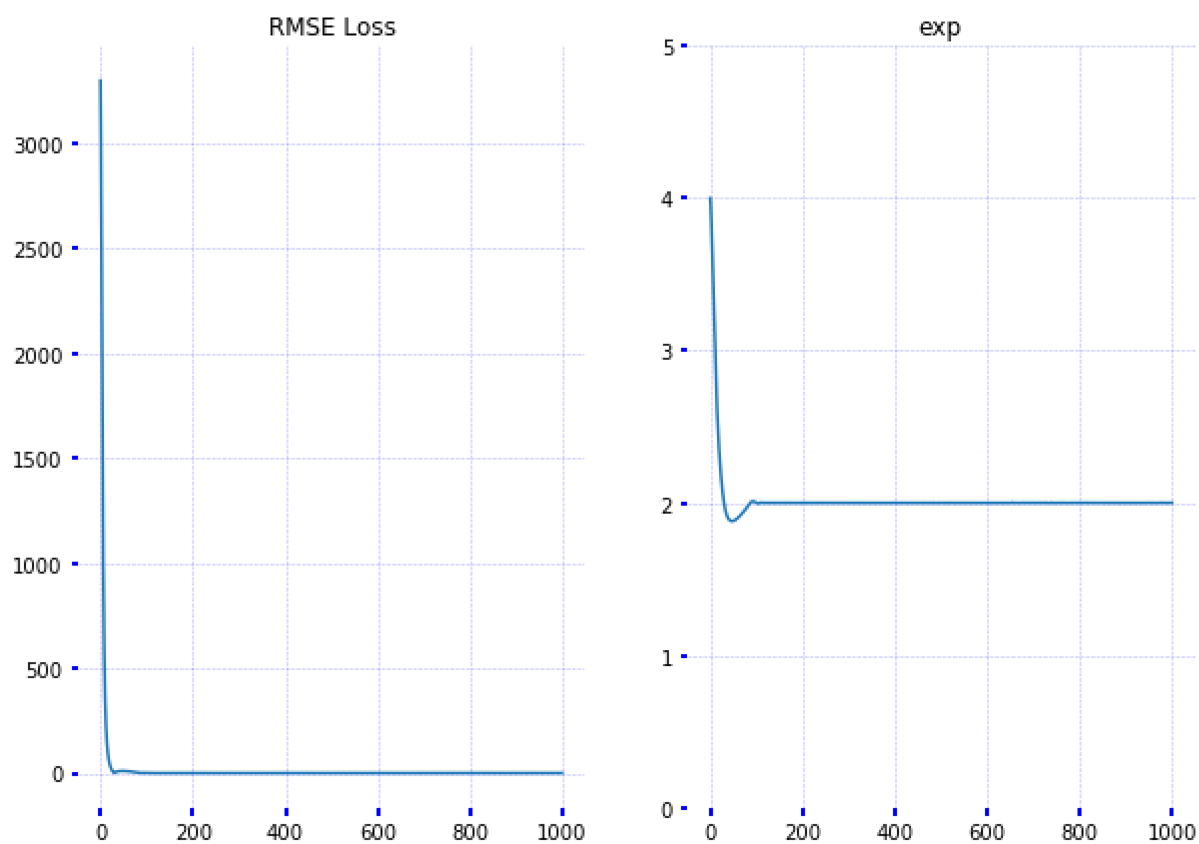
pytorch.py hosted with ❤ by GitHub view raw

If you have some experience in deep learning frameworks you may have noticed that we are implementing gradient descent by hand. Not very convenient, huh? Gladly, PyTorch

```
import torch
from torch.autograd import Variable
import numpy as np

def rmse(y, y_hat):
    """Compute root mean squared error"""
    return torch.sqrt(torch.mean((y - y_hat).pow(2)))

def forward(x, e):
    """Forward pass for our fuction"""
    return x.pow(e.repeat(x.size(0)))

# Let's define some settings
n = 1000 # number of examples
learning_rate = 5e-10

# Model definition
x = Variable(torch.rand(n) * 10, requires_grad=False)
y = forward(x, exp)

# Model parameters
exp = Variable(torch.FloatTensor([2.0]), requires_grad=False)
exp_hat = Variable(torch.FloatTensor([4]), requires_grad=True)

# Optimizer (NEW)
opt = torch.optim.SGD([exp_hat], lr=learning_rate, momentum=0.9)

loss_history = []
exp_history = []

# Training loop
for i in range(0, 10000):
    opt.zero_grad()
    print("Iteration %d" % i)

    # Compute current estimate
    y_hat = forward(x, exp_hat)

    # Calculate loss function
    loss = rmse(y, y_hat)
```

```
    exp_history.append(y_hat.data[0])

    # Update model parameters
    loss.backward()
    opt.step()

    print("loss = %s" % loss.data[0])
    print("exp = %s" % exp_hat.data[0])
```

pytorch_optim.py hosted with ❤ by GitHub view raw



Loss function and exponent plots for PyTorch

As you can see, we quickly inferred true exponent from training data. And now let's go on with TensorFlow:

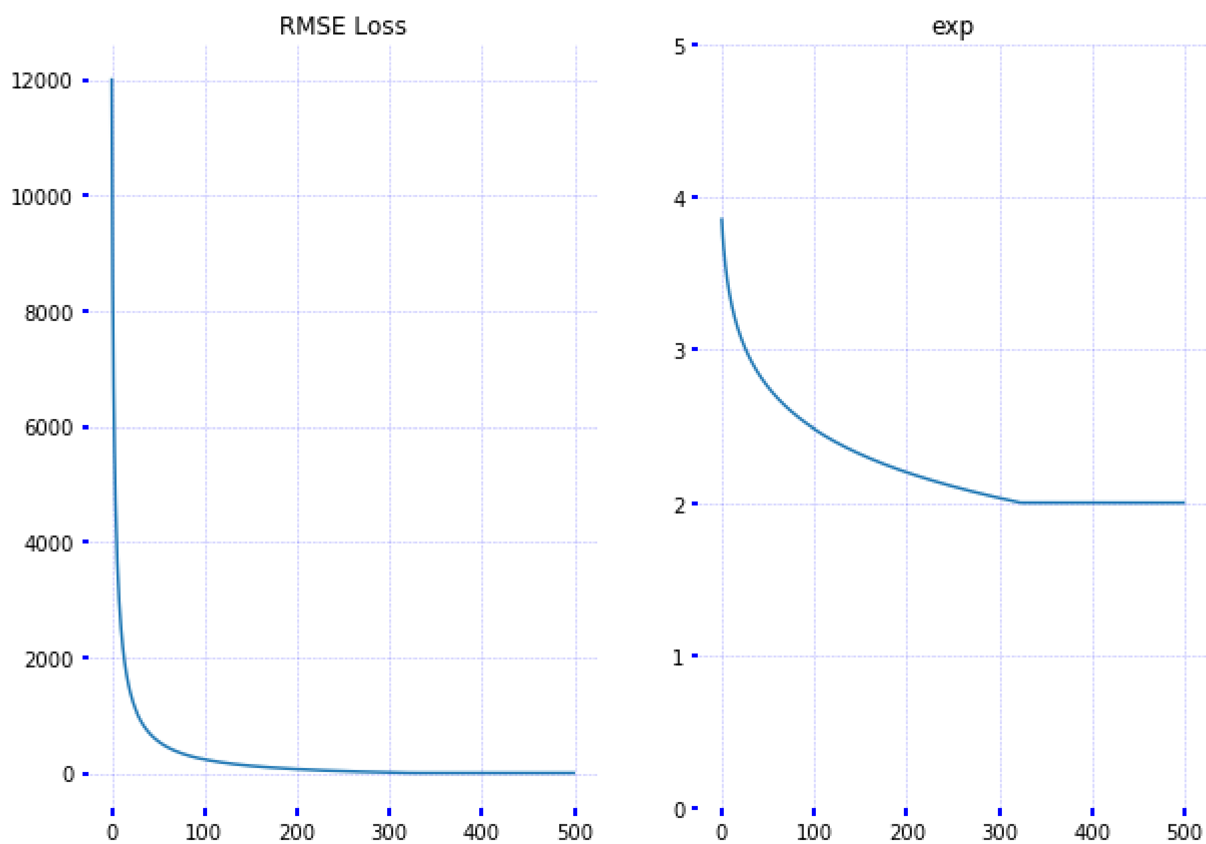
```
import tensorflow as tf

def rmse(y, y_hat):
    """Compute root mean squared error"""
```

Get started Open in app

```
    """Forward pass for our fuction"""
    # tensorflow has automatic broadcasting
    # so we do not need to reshape e manually
    return tf.pow(x, e)

n = 100 # number of examples
learning_rate = 5e-6

# Placeholders for data
x = tf.placeholder(tf.float32)
y = tf.placeholder(tf.float32)

# Model parameters
exp = tf.constant(2.0)
exp_hat = tf.Variable(4.0, name='exp_hat')

# Model definition
y_hat = forward(x, exp_hat)

# Optimizer
loss = rmse(y, y_hat)
opt = tf.train.GradientDescentOptimizer(learning_rate)

# We will run this operation to perform a single training step,
# e.g. opt.step() in Pytorch.
# Execution of this operation will also update model parameters
train_op = opt.minimize(loss)

# Let's generate some training data
x_train = np.random.rand(n) + 10
y_train = x_train ** 2

loss_history = []
exp_history = []

# First, we need to create a Tensorflow session object
with tf.Session() as sess:

    # Initialize all defined variables
    tf.global_variables_initializer().run()

    # Training loop
```

```
        curr_loss, curr_exp, _ = sess.run([loss, exp_hat, train_op], feed_dict={x: x_train, y: y_

        print("loss = %s" % curr_loss)
        print("exp = %s" % curr_exp)

        # Do some recordings for plots
        loss_history.append(curr_loss)
        exp_history.append(curr_exp)
```

tf_example_1.py hosted with ❤ by GitHub view raw



Loss function and exponent plots for TensorFlow

As you can see, implementation in TensorFlow works too (surprisingly 🙃). It took more iterations to recover the exponent, but I am sure that the cause is I did not fiddle with optimiser's parameters enough to reach comparable results.

Now we are ready to explore some differences.

professionals. The framework is well documented and if the documentation will not suffice there are many extremely well-written tutorials on the internet. You can find hundreds of implemented and trained models on github, start here.

PyTorch is relatively new compared to its competitor (and is still in beta), but it is quickly getting its momentum. Documentation and official tutorials are also nice. PyTorch also include several implementations of popular computer vision architectures which are super-easy to use.

Difference #1 — dynamic vs static graph definition

Both frameworks operate on tensors and view any model as a directed acyclic graph (DAG), but they differ drastically on how you can define them.

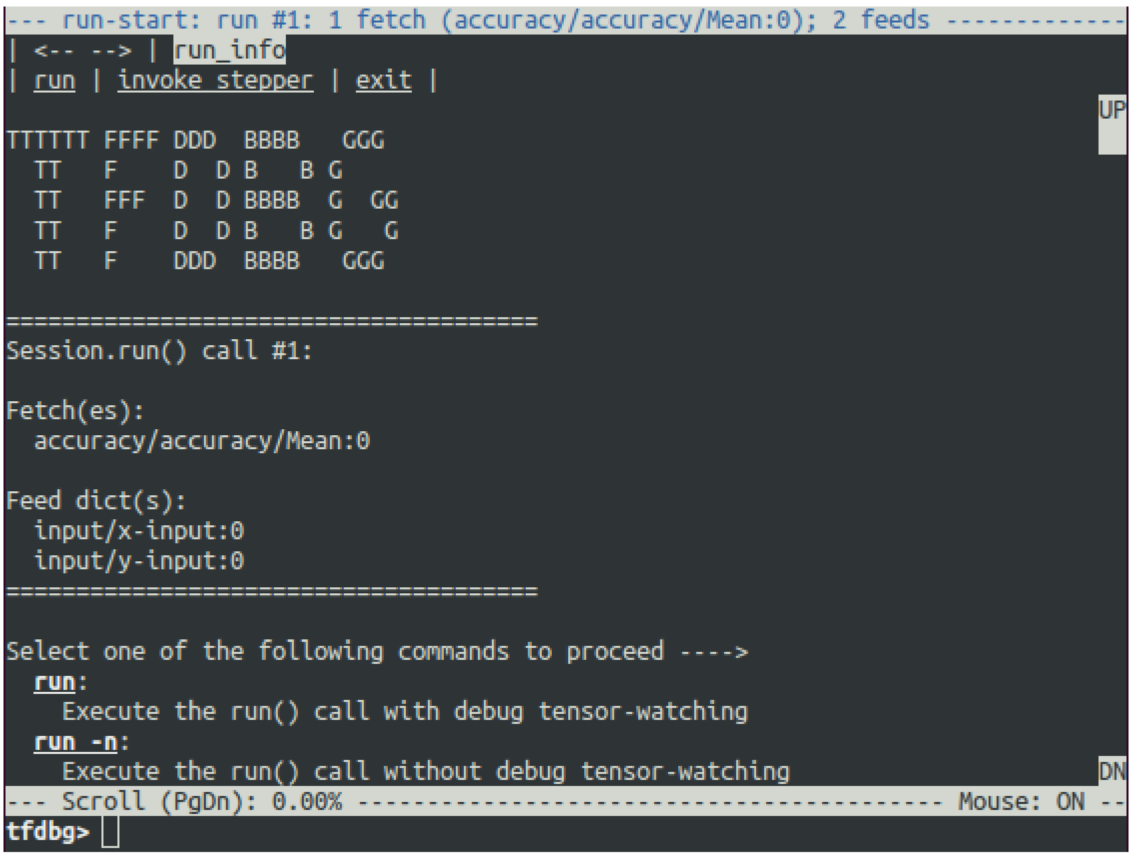
TensorFlow follows 'data as code and code is data' idiom. In TensorFlow you define graph statically before a model can run. All communication with outer world is performed via `tf.Session` object and `tf.Placeholder` which are tensors that will be substituted by external data at runtime.

In PyTorch things are way more imperative and dynamic: you can define, change and execute nodes as you go, no special session interfaces or placeholders. Overall, the framework is more tightly integrated with Python language and feels more native most of the times. When you write in TensorFlow sometimes you feel that your model is behind a brick wall with several tiny holes to communicate over. Anyways, this still sounds like a matter of taste more or less.

However, those approaches differ not only in a software engineering perspective: there are several dynamic neural network architectures that can benefit from the dynamic approach. Recall RNNs: with static graphs, the input sequence length will stay constant. This means that if you develop a sentiment analysis model for English sentences you must fix the sentence length to some maximum value and pad all smaller sequences with zeros. Not too convenient, huh. And you will get more problems in the domain of recursive RNNs and tree-RNNs. Currently Tensorflow has limited support for dynamic inputs via Tensorflow Fold. PyTorch has it by-default.

Python debugging tools such as pdb, ipdb, PyCharm debugger or old trusty print statements.

This is not the case with TensorFlow. You have an option to use a special tool called tfdbg which allows to evaluate tensorflow expressions at runtime and browse all tensors and operations in session scope. Of course, you won't be able to debug any python code with it, so it will be necessary to use pdb separately.



Difference #3 — Visualization

Tensorboard is awesome when it comes to visualization 😎. This tool comes with TensorFlow and it is very useful for debugging and comparison of different training runs. For example, consider you trained a model, then tuned some hyperparameters and trained it again. Both runs can be displayed at Tensorboard simultaneously to indicate possible differences. Tensorboard can:

- Display model graph

- Visualize distributions and histograms
- Visualize images
- Visualize embeddings
- Play audio

Tensorboard can display various summaries which can be collected via `tf.summary` module. We will define summary operations for our toy exponent example and use `tf.summary.FileWriter` to save them to disk.

```
import tensorflow as tf
import numpy as np

def rmse(y, y_hat):
    """Compute root mean squared error"""
    return tf.sqrt(tf.reduce_mean(tf.square((y - y_hat))))

def forward(x, e):
    """Forward pass for our fuction"""
    # tensorflow has automatic broadcasting
    # so we do not need to reshape e manually
    return tf.pow(x, e)

n = 100 # number of examples
learning_rate = 5e-6

# Placeholders for data
x = tf.placeholder(tf.float32)
y = tf.placeholder(tf.float32)

# Model parameters
exp = tf.constant(2.0)
exp_hat = tf.Variable(4.0, name='exp_hat')

# Model definition
y_hat = forward(x, exp_hat)

# Optimizer
loss = rmse(y, y_hat)
```

```python
loss_summary = tf.summary.scalar("loss", loss)
exp_summary = tf.summary.scalar("exp", exp_hat)
all_summaries = tf.summary.merge_all()

# We will run this operation to perform a single training step,
# e.g. opt.step() in Pytorch.
# Execution of this operation will also update model parameters
train_op = opt.minimize(loss)

# Let's generate some training data
x_train = np.random.rand(n) + 10
y_train = x_train ** 2

loss_history = []
exp_history = []

# First, we need to create a Tensorflow session object
with tf.Session() as sess:

    # Initialize all defined variables
    tf.global_variables_initializer().run()

    summary_writer = tf.summary.FileWriter('./tensorboard', sess.graph)

    # Training loop
    for i in range(0, 500):
        print("Iteration %d" % i)
        # Run a single trainig step
        summaries, curr_loss, curr_exp, _ = sess.run([all_summaries, loss, exp_hat, train_op], f

        print("loss = %s" % curr_loss)
        print("exp = %s" % curr_exp)

        # Do some recordings for plots
        loss_history.append(curr_loss)
        exp_history.append(curr_exp)

        summary_writer.add_summary(summaries, i)
```

tensorboard.py hosted with ❤ by GitHub view raw

Tensorboard competitor from the PyTorch side is visdom. It is not as feature-complete, but a bit more convenient to use. Also, integrations with Tensorboard do exist. Also, you are free to use standard plotting tools — matplotlib and seaborn.

Difference #4 — Deployment

If we start talking about deployment TensorFlow is a clear winner for now: is has TensorFlow Serving which is a framework to deploy your models on a specialized gRPC server. Mobile is also supported.

When we switch back to PyTorch we may use Flask or another alternative to code up a REST API on top of the model. This could be done with TensorFlow models as well if gRPC is not a good match for your usecase. However, TensorFlow Serving may be a better option if performance is a concern.

Tensorflow also supports distributed training which PyTorch lacks for now.

Difference #5 — Data Parallelism

One of the biggest features that distinguish PyTorch from TensorFlow is declarative data parallelism: you can use `torch.nn.DataParallel` to wrap any module and it will be (almost magically) parallelized over batch dimension. This way you can leverage multiple GPUs with almost no effort.

On the other hand, TensorFlow allows you to fine tune every operation to be run on specific device. Nonetheless, defining parallelism is way more manual and requires careful thought. Consider the code that implements something like `DataParallel` in TensorFlow:

```
def make_parallel(fn, num_gpus, **kwargs):
    in_splits = {}
    for k, v in kwargs.items():
        in_splits[k] = tf.split(v, num_gpus)

out_split = []
    for i in range(num_gpus):
        with tf.device(tf.DeviceSpec(device_type="GPU",
```

```
                out_split.append(fn(**{k : v[i] for k, v in
in_splits.items()}))

return tf.concat(out_split, axis=0)

def model(a, b):
    return a + b

c = make_parallel(model, 2, a=a, b=b)
```

That being said, when using TensorFlow you can achive everything you can do in PyTorch, but with more effort (you have more control as a bonus).

Also it is worth noting that both frameworks support distributed execution and provide high level interfaces for defining clusters.

Difference #6 — A Framework or a library

Let's build a CNN classifier for handwritten digits. Now PyTorch will really start to look like a *framework*. Recall that a programming framework gives us useful abstractions in certain domain and a convenient way to use them to solve concrete problems. That is the essence that separates a framework from a library.

Here we introduce `datasets` module which contains wrappers for popular datasets used to benchmark deep learning architectures. Also `nn.Module` is used to build a custom convolutional neural network classifier. `nn.Module` is a building block PyTorch gives us to create complex deep learning architectures. There are large amounts of ready to use modules in `torch.nn` package that we can use as a base for our model. Notice how PyTorch uses object oriented approach to define basic building blocks and give us some 'rails' to move on while providing ability to extend functionality via subclassing.

Here goes a slightly modified version of https://github.com/pytorch/examples/blob/master/mnist/main.py:

```
import numpy as np
import tensorflow as tf

from tensorflow.contrib import learn
```

```


def cnn_model_fn(features, labels, mode):
  """Model function for CNN."""
  # Input Layer
  # Reshape X to 4-D tensor: [batch_size, width, height, channels]
  # MNIST images are 28x28 pixels, and have one color channel
  input_layer = tf.reshape(features, [-1, 28, 28, 1])

  # Convolutional Layer #1
  # Computes 32 features using a 5x5 filter with ReLU activation.
  # Padding is added to preserve width and height.
  # Input Tensor Shape: [batch_size, 28, 28, 1]
  # Output Tensor Shape: [batch_size, 28, 28, 32]
  conv1 = tf.layers.conv2d(
      inputs=input_layer,
      filters=32,
      kernel_size=[5, 5],
      padding="same",
      activation=tf.nn.relu)

  # Pooling Layer #1
  # First max pooling layer with a 2x2 filter and stride of 2
  # Input Tensor Shape: [batch_size, 28, 28, 32]
  # Output Tensor Shape: [batch_size, 14, 14, 32]
  pool1 = tf.layers.max_pooling2d(inputs=conv1, pool_size=[2, 2], strides=2)

  # Convolutional Layer #2
  # Computes 64 features using a 5x5 filter.
  # Padding is added to preserve width and height.
  # Input Tensor Shape: [batch_size, 14, 14, 32]
  # Output Tensor Shape: [batch_size, 14, 14, 64]
  conv2 = tf.layers.conv2d(
      inputs=pool1,
      filters=64,
      kernel_size=[5, 5],
      padding="same",
      activation=tf.nn.relu)

  # Pooling Layer #2
  # Second max pooling layer with a 2x2 filter and stride of 2
  # Input Tensor Shape: [batch_size, 14, 14, 64]
```

Get started Open in app

```

# Flatten tensor into a batch of vectors
# Input Tensor Shape: [batch_size, 7, 7, 64]
# Output Tensor Shape: [batch_size, 7 * 7 * 64]
pool2_flat = tf.reshape(pool2, [-1, 7 * 7 * 64])

# Dense Layer
# Densely connected layer with 1024 neurons
# Input Tensor Shape: [batch_size, 7 * 7 * 64]
# Output Tensor Shape: [batch_size, 1024]
dense = tf.layers.dense(inputs=pool2_flat, units=1024, activation=tf.nn.relu)

# Add dropout operation; 0.6 probability that element will be kept
dropout = tf.layers.dropout(
    inputs=dense, rate=0.4, training=mode == learn.ModeKeys.TRAIN)

# Logits layer
# Input Tensor Shape: [batch_size, 1024]
# Output Tensor Shape: [batch_size, 10]
logits = tf.layers.dense(inputs=dropout, units=10)

loss = None
train_op = None

# Calculate Loss (for both TRAIN and EVAL modes)
if mode != learn.ModeKeys.INFER:
  onehot_labels = tf.one_hot(indices=tf.cast(labels, tf.int32), depth=10)
  loss = tf.losses.softmax_cross_entropy(
      onehot_labels=onehot_labels, logits=logits)

# Configure the Training Op (for TRAIN mode)
if mode == learn.ModeKeys.TRAIN:
  train_op = tf.contrib.layers.optimize_loss(
      loss=loss,
      global_step=tf.contrib.framework.get_global_step(),
      learning_rate=0.001,
      optimizer="SGD")

# Generate Predictions
predictions = {
    "classes": tf.argmax(
        input=logits, axis=1),
```

```
  # Return a ModelFnOps object
  return model_fn_lib.ModelFnOps(
      mode=mode, predictions=predictions, loss=loss, train_op=train_op)


# Load training and eval data
mnist = learn.datasets.load_dataset("mnist")
train_data = mnist.train.images  # Returns np.array
train_labels = np.asarray(mnist.train.labels, dtype=np.int32)
eval_data = mnist.test.images  # Returns np.array
eval_labels = np.asarray(mnist.test.labels, dtype=np.int32)

# Create the Estimator
mnist_classifier = learn.Estimator(
  model_fn=cnn_model_fn, model_dir="/tmp/mnist_convnet_model")

# Set up logging for predictions
# Log the values in the "Softmax" tensor with label "probabilities"
tensors_to_log = {"probabilities": "softmax_tensor"}
logging_hook = tf.train.LoggingTensorHook(
  tensors=tensors_to_log, every_n_iter=50)

# Train the model
mnist_classifier.fit(
  x=train_data,
  y=train_labels,
  batch_size=100,
  steps=20000,
  monitors=[logging_hook])

# Configure the accuracy metric for evaluation
metrics = {
  "accuracy":
      learn.MetricSpec(
          metric_fn=tf.metrics.accuracy, prediction_key="classes"),
}

# Evaluate the model and print results
eval_results = mnist_classifier.evaluate(
  x=eval_data, y=eval_labels, metrics=metrics)
print(eval_results)
```

Plain TensorFlow feels a lot more like a library rather than a framework: all operations are pretty low-level and you will need to write lots of boilerplate code even when you might not want to (let's define those biases and weights again and again and …).

As the time as passed a whole ecosystem of high-level wrappers started to emerge around TensorFlow. Each of those aims to simplify the way you work with the library. Many of them are currently located at `tensorflow.contrib` module (which is not considered a stable API) and some started to migrate to the main repository (see `tf.layers`).

So, you have a lot of freedom on how to use TensorFlow and what framework will suit the task best: TFLearn, tf.contrib.learn, Sonnet, Keras, plain `tf.layers`, etc. To be honest, Keras deserves another post but is currently out of the scope of this comparison.

Here we will use `tf.layers` and `tf.contrib.learn` to build our CNN classifier. The code follows the official tutorial on tf.layers:

```
import numpy as np
import tensorflow as tf

from tensorflow.contrib import learn
from tensorflow.contrib.learn.python.learn.estimators import model_fn as model_fn_lib

tf.logging.set_verbosity(tf.logging.INFO)


def cnn_model_fn(features, labels, mode):
  """Model function for CNN."""
  # Input Layer
  # Reshape X to 4-D tensor: [batch_size, width, height, channels]
  # MNIST images are 28x28 pixels, and have one color channel
  input_layer = tf.reshape(features, [-1, 28, 28, 1])

  # Convolutional Layer #1
  # Computes 32 features using a 5x5 filter with ReLU activation.
  # Padding is added to preserve width and height.
  # Input Tensor Shape: [batch_size, 28, 28, 1]
```

```
        filters=32,
        kernel_size=[5, 5],
        padding="same",
        activation=tf.nn.relu)

    # Pooling Layer #1
    # First max pooling layer with a 2x2 filter and stride of 2
    # Input Tensor Shape: [batch_size, 28, 28, 32]
    # Output Tensor Shape: [batch_size, 14, 14, 32]
    pool1 = tf.layers.max_pooling2d(inputs=conv1, pool_size=[2, 2], strides=2)

    # Convolutional Layer #2
    # Computes 64 features using a 5x5 filter.
    # Padding is added to preserve width and height.
    # Input Tensor Shape: [batch_size, 14, 14, 32]
    # Output Tensor Shape: [batch_size, 14, 14, 64]
    conv2 = tf.layers.conv2d(
        inputs=pool1,
        filters=64,
        kernel_size=[5, 5],
        padding="same",
        activation=tf.nn.relu)

    # Pooling Layer #2
    # Second max pooling layer with a 2x2 filter and stride of 2
    # Input Tensor Shape: [batch_size, 14, 14, 64]
    # Output Tensor Shape: [batch_size, 7, 7, 64]
    pool2 = tf.layers.max_pooling2d(inputs=conv2, pool_size=[2, 2], strides=2)

    # Flatten tensor into a batch of vectors
    # Input Tensor Shape: [batch_size, 7, 7, 64]
    # Output Tensor Shape: [batch_size, 7 * 7 * 64]
    pool2_flat = tf.reshape(pool2, [-1, 7 * 7 * 64])

    # Dense Layer
    # Densely connected layer with 1024 neurons
    # Input Tensor Shape: [batch_size, 7 * 7 * 64]
    # Output Tensor Shape: [batch_size, 1024]
    dense = tf.layers.dense(inputs=pool2_flat, units=1024, activation=tf.nn.relu)

    # Add dropout operation; 0.6 probability that element will be kept
    dropout = tf.layers.dropout(
```

```
  # Input Tensor Shape: [batch_size, 1024]
  # Output Tensor Shape: [batch_size, 10]
  logits = tf.layers.dense(inputs=dropout, units=10)

  loss = None
  train_op = None

  # Calculate Loss (for both TRAIN and EVAL modes)
  if mode != learn.ModeKeys.INFER:
    onehot_labels = tf.one_hot(indices=tf.cast(labels, tf.int32), depth=10)
    loss = tf.losses.softmax_cross_entropy(
        onehot_labels=onehot_labels, logits=logits)

  # Configure the Training Op (for TRAIN mode)
  if mode == learn.ModeKeys.TRAIN:
    train_op = tf.contrib.layers.optimize_loss(
        loss=loss,
        global_step=tf.contrib.framework.get_global_step(),
        learning_rate=0.001,
        optimizer="SGD")

  # Generate Predictions
  predictions = {
      "classes": tf.argmax(
          input=logits, axis=1),
      "probabilities": tf.nn.softmax(
          logits, name="softmax_tensor")
  }

  # Return a ModelFnOps object
  return model_fn_lib.ModelFnOps(
      mode=mode, predictions=predictions, loss=loss, train_op=train_op)


# Load training and eval data
mnist = learn.datasets.load_dataset("mnist")
train_data = mnist.train.images  # Returns np.array
train_labels = np.asarray(mnist.train.labels, dtype=np.int32)
eval_data = mnist.test.images  # Returns np.array
eval_labels = np.asarray(mnist.test.labels, dtype=np.int32)

```

Get started Open in app

```

# Set up logging for predictions
# Log the values in the "Softmax" tensor with label "probabilities"
tensors_to_log = {"probabilities": "softmax_tensor"}
logging_hook = tf.train.LoggingTensorHook(
  tensors=tensors_to_log, every_n_iter=50)

# Train the model
mnist_classifier.fit(
  x=train_data,
  y=train_labels,
  batch_size=100,
  steps=20000,
  monitors=[logging_hook])

# Configure the accuracy metric for evaluation
metrics = {
  "accuracy":
      learn.MetricSpec(
          metric_fn=tf.metrics.accuracy, prediction_key="classes"),
}

# Evaluate the model and print results
eval_results = mnist_classifier.evaluate(
  x=eval_data, y=eval_labels, metrics=metrics)
print(eval_results)
```

tensorflow_mnist.py hosted with ❤ by GitHub view raw

So, both TensorFlow and PyTorch provide useful abstractions to reduce amounts of boilerplate code and speed up model development. The main difference between them is that PyTorch may feel more “pythonic” and has an object-oriented approach while TensorFlow has several options from which you may choose.

Personally, I consider PyTorch to be more clear and developer-friendly. It’s `torch.nn.Module` gives you the ability to define reusable modules in an OOP manner and I find this approach very flexible and powerful. Later you can compose all kind of modules via `torch.nn.Sequential` (hi Keras ✋🏻). Also, you have all built-in modules in a

Of course, you can write very clean code in plain TensorFlow but it just takes more skill and trial-and-error before you get it. When it goes to higher-level frameworks such as Keras or TFLearn get ready to lose at least some of the flexibility TensorFlow has to offer.

Conclusion

TensorFlow is very powerful and mature deep learning library with strong visualization capabilities and several options to use for high-level model development. It has production-ready deployment options and support for mobile platforms. TensorFlow is a good option if you:

- Develop models for production
- Develop models which need to be deployed on mobile platforms
- Want good community support and comprehensive documentation
- Want rich learning resources in various forms (TensorFlow has an an entire MOOC)
- Want or need to use Tensorboard
- Need to use large-scale distributed model training

PyTorch is still a young framework which is getting momentum fast. You may find it a good fit if you:

- Do research or your production non-functional requirements are not very demanding
- Want better development and debugging experience
- *Love* all things Pythonic

If you have the time the best advice would be to try both and see what fits your needs best.

If you liked this article, please leave a few 👏. It lets me know that I am helping.

native C++ API, JIT compilation and ONNX integration. This means that you will be able to write production-ready services and do what TensorFlow Serving does. This is a big step to PyTorch and surely will empower its position as a fully featured framework for both research and production purposes.

Contact us

Need help with TensorFlow or PyTorch? Contact us at **datalab@cinimex.ru**

Want to get regular stream of interesting resources on Data Science and Entrepreneurship?

Follow me on 🐦 Twitter and 📝Medium.

Machine Learning Deep Learning TensorFlow Python Pytorch