Haskell-Torch

Sep 15, 2020

Contents

Coming soon!

CHAPTER 1

A walkthrough of the basics

We're going to walk through end-to-end examples of the Haskell-Torch API. You can get many more examples of full networks in the repository.

First, we need some extensions. A lot of extensions.

```
{-# LANGUAGE AllowAmbiguousTypes, ConstraintKinds, ExtendedDefaultRules,␣
↪FlexibleContexts, FlexibleInstances, GADTs, OverloadedStrings #-}
{-# LANGUAGE PolyKinds, QuasiQuotes, RankNTypes, ScopedTypeVariables, TemplateHaskell,
↪ TypeApplications, TypeFamilies #-}
{-# LANGUAGE TypeFamilyDependencies, TypeInType, TypeOperators, UndecidableInstances
↪#-}
```

This example is somewhat faithful translation of the yunjey basics of pytorch tutorial. You should be able to follow the Haskell and Python versions side by side, or just read through this example on its own.

```
module Basics where
import           Control.Monad
import           Data.Default
import           Data.Kind
import           Data.Maybe
import           Data.Singletons
import           Data.String.InterpolateIO
import qualified Data.Vector                as V'
import           Data.Vector.Storable       (Vector)
import qualified Data.Vector.Storable       as V
import           Foreign.C.Types
import           Pipes
import qualified Pipes.Prelude              as P
import           Torch
import qualified Torch.C.Variable           as C
import           Torch.Datasets.Vision.CIFAR
```

1.1 Basic autograd

We start with a very low-level example of how to compute a gradient for some mathematical operations. You'll notice that all of our operations are in `IO`. Right now, we don't provide a pure interface. We may in the future. At the end we print the AD graph to give you a visual feel of what is going on.

```
ex1 = do
```

Always a good idea to set the seed so that your results are reproducible.

```
  setSeed 0
```

If we don't annotate the type parameters here we get into trouble, GHC won't know where these values should be stored. You could write `KCuda` or `KBest` instead of `KCpu`. If your machine doesn't have a CUDA device, `KCuda` will not typecheck. You will notice that all parameters required to determine the shape, size, type, and device of networks are passed as type parameters, not runtime values. This is integral to our goal of trying to eliminate runtime errors.

```
  s <- stored @KCpu <$> toScalar (float 1)
```

We need to mark which values need gradients.

```
  x <- stored @KCpu <$> (needGrad =<< toScalar (float 1))
  w <- needGrad =<< toScalar (float 2)
  b <- needGrad =<< toScalar (float 3)
```

This package uses a customized version of `Interpolate` called `InterpolateIO`. You will often want to print tensors and this can't be done through any pure operations as their storage can change unexpectedly. Ensuring that tensors can share storage is key to maintaining good performance and one of the major reasons why we have no pure API: better to be impure and fast than pure and useless.

```
  putStrLn =<< [c|X: #{x}
W: #{w}
B: #{b}|]
```

Now we get to run our actual computation. Notice the funky infix operators. `.*` is multiplication between two pure tensors and results in an tensor wrapped in `IO`, while `..*` takes arguments that are already in `IO`.

```
  y <- pure w ..* pure x ..+ pure b
```

Usually, you want to compute the gradient with respect to a single tensor and don't want to keep the AD graph around for higher order derivatives.

```
  backward1 y False False
```

Again we use `InterpolateIO` but this time in a more interesting way. Note that `gradient` is an impure operation, its result is in `IO`, but string interpolation takes care of these values.

```
  putStrLn =<< [c|dX: #{gradient x} expected 2
dW: #{gradient w} expected 1
dB: #{gradient b} expected 1|]
```

Finally, a small bonus that doesn't appear in the original version, we print the AD graph. This should show it on screen if you have graphviz installed.

```
  debuggingPrintADGraph y
```

1.2 Basic autograd with SGD

Now it's time to do some real optimization!

```
ex2 = do
```

Enabling gradients isn't technically required upfront because they are enbled by default. If you have code that disables them, this is a good time to ensure they are on. You might wonder why we even bother providing such an usafe operation instead of something like `bracket` for enabling gradients. We do provide `withGrad` and `withoutGrad` which enable and disable gradients in a region, but it is helpful to have low level access as well. For example, some existing code may not be factored out into neat regions.

```
  unsafeEnableGrad
  setSeed 0
```

As before, we create some tensors, this time they are random. Note that we use `typed`, `stored`, and `sized`. These and many similar operations let you pin down at compile time one or more features of whatever tensor you are operating on. It's a great idea to use these whenever you can. In this case, they are necessary, otherwise the compiler will not know what size to make our tensors.

```
  x <- typed @TFloat <$> stored @KCpu <$> sized (size_ @'[10,3]) <$> randn
```

Note that we don't need to specify these properties separately, `randn` gives us the type parameters to do this in one go. In this case, we are overspecifying the types as the fact that `x` is `TFloat` and `KCpu` would propagate to `y` even without these type annotations. The size of `y` is constrained by the size of the linear layer we define below, it too could be omitted in which case the code would simply be `y <- randn`.

```
  y <- randn @TFloat @KCpu @'[10,2]
```

You will see `gradP` in many places. It's how we create parameters that are differentiable. The size, type, device, etc. of the parameters is inferred. A collection of weights can be converted to parameters, which is really an opaque heterogenous container that can store tensors of any size. We then create an optimizer, in this can just a simple SGD with constant learning rate. Note that optimizers are created for parameters, you may create as many of them as you wish and even bind the same parameters to multiple optimizers.

```
  w <- gradP
  params <- toParameters w
  let optimizer = sgd (def { sgdLearningRate = 0.01 }) params
```

Our first layer and loss. When a function has many type parameters, or their meaning is ambiguous, we tag them as below. Pure functions which end in underscores always tag parameter types. Code like `linear @3 @2` would simply be unreadable. Many values also have sensible defaults, you can use `def` to fill them in.

```
  let model = linear (inFeatures_ @3) (outFeatures_ @2) w
  pred <- model x
  let criterion = mseLoss y def
  loss <- criterion pred
```

This has captured an AD graph, which we can walk to compute gradients and print the state of the model.

```
  backward1 loss False False
  putStrLn =<< [c|weights & biases:\n#{w}
Loss: #{loss}|]
```

Now that we have a gradient it's time to take a step. Notice how `step_` also ends in an underscore, this is to indicate that the operation is inplace. Functions which end in underscore and are impure, are inplace operations. These is generally going to be be a non-destructive version without the underscore, but that is not guaranteed.

You should see the loss go down!

```
  step_ optimizer
  pred <- model x
  loss <- criterion pred
  putStrLn =<< [c|Loss after 1 SGD step #{loss}|]
  pure ()
```

1.3 Loading and storing model weights

We plan to have a more comprehensive checkpointing and restarting solution down the road, but for now, you can save and load model weights to disk. Whenever we say "loading a model" it always means that we load the weights of the model, networks are always defined in Haskell, so there is no way to load them from disk.

```
ex3 = do
  let v = V.fromList [1,2,3,4]
  -- The resulting tensor is always on the CPU, does not have gradients enabled
  -- and is marked as a leaf. Its type depends on the type of the Vector.  Only
  -- Vectors with Foreign.C types are allowed (so CDouble instead of Double,
  -- etc.).
  --
  -- For the result to be useful you must somehow constrain the types.  Here we
  -- do so locally using type application but if some downstream consumer of t
  -- constrained its shape we would not need to do so.
  --
  -- This is one of the few interfaces between runtime values and types. It will
  -- error out if the size of the vector is not exactly equal to size of the
  -- tensor.
  t <- fromVector @'TDouble @'[4] v
  -- Alternatively we can use the functions found under the Constraints heading
  -- in Torch.Tensor. These have no runtime component, they just allow you to
  -- constrain the types of tensors easily.
  t' <- typed @'TDouble <$> sized (size_ @'[4]) <$> fromVector v
  -- Or we can say that the new tensor should inherit its properties, aside from
  -- AD status like if gradients are required, from another tensor.
  t'' <- like t <$> fromVector v
  -- A few other ways to create vectors exist, see the "Tensor creation" section
  -- in Tensor.Torch, for example we can make the vector of all 1s that's just
  -- like t.
  t''' <- like t <$> ones
  -- We can also convert tenstors back into vectors.
  v' <- toVector t
  writeModelToFile t "/tmp/woof"
  tl <- like t <$> readModelFromFile "/tmp/woof"
  out t
  out tl
  out =<< t .== tl
```

(continues on next page)

(continued from previous page)

```
  print v'
  print $ v == v'

-- | Input datasets
ex4 = do
  -- Datasets get downloaded and then streamed using Pipes
  (train,test) <- cifar10 "datasets/image/"
  -- Unpack the training set
  (Right trainStream) <- liftM (transformObjectStream rgbImageToTensor) <$>␣
↪fetchDataset train
  -- Data is loaded on demand, here we read the first data point
  (Just d) <- P.head trainStream
  image <- typed @TByte <$> dataObject d
  label <- dataLabel d
  print $ dataProperties d
  print $ size image
  out label
  -- All datasets can define any custom metadata that they want. CIFAR gives you
  -- a map between indices and text labels so you can interpret the classes.
  metadata <- metadataDataset train
  print metadata
  -- Lets iterate one by one over the first 10 data points shuffling with a
  -- horizon of 1000
  forEachData
    (\d -> do
        print "One data point at a time"
        -- Training code goes here
        putStrLn =<< [c|n: #{dataProperties d} lab: #{dataLabel d}|])
    (shuffle 1000 trainStream >-> P.take 10)
  -- Same if we batch by 64. True means give us a partial batch at the end if our
  -- data isn't divisble by 64.
  forEachData
    (\ds -> do
        print "Batches of 64"
        print $ V'.length ds
        mapM_ (\d ->
                        -- Training code goes here
                        putStrLn =<< [c|n: #{dataProperties d} lab: #{dataLabel d}
↪|]) ds)
     (batch 64 True (shuffle 1000 trainStream) >-> P.take 3)
  pure ()

-- ex5 does not exist. We don't need anything like custom loaders, you just
-- create pipes. Look at how the datasets are constructed in Torch.Datasets

-- ex6 does not exist. Have a look at Torch.Models.Vision.AlexNet how to load
-- pretrained models.

-- ex7 does not exist. TODO We do not yet have integrated checkpointing
-- support. You can save and load a model but we cannot yet do this with
-- optimizers and cannot do it all for you in one go.

-- Viewing the trace of a computation
--------------------------------------------------------------------------------

ex8 = do
  unsafeEnableGrad
```

(continues on next page)

(continued from previous page)

```
  setSeed 0
  -- Create tensors of shape (10, 3) and (10, 2).
  x <- typed @TFloat <$> stored @KCpu <$> sized (size_ @'[10,3]) <$> randn
  y <- sized (size_ @'[10,2]) <$> randn
  -- Weights and biases for a fully connected layer
  w <- noGradP
  let model = linear (inFeatures_ @3) (outFeatures_ @2) w
  let criterion = mseLoss y def
  params <- toParameters w
  (loss, trace) <- withTracing [AnyTensor x, AnyTensor y] $ do
    pred <- model x
    criterion pred
  putStrLn =<< [c|weights & biases:\n#{w}
Loss: #{loss}|]
  printTrace trace
  printTraceONNX trace [AnyTensor x, AnyTensor y] False 11
  trace' <- parseTrace trace
  rawTrace trace
  summarizeTrace trace'
  showTraceGraph trace False
  -- 1 step of gradient descent
  p <- toParameters w
  step_ (sgd (def { sgdLearningRate = 0.01 }) p)
  --
  pred <- model x
  loss <- criterion pred
  putStrLn =<< [c|Loss after 1 SGD step #{loss}|]
  pure ()
```

CHAPTER 2

Indices and tables

- genindex
- modindex
- search