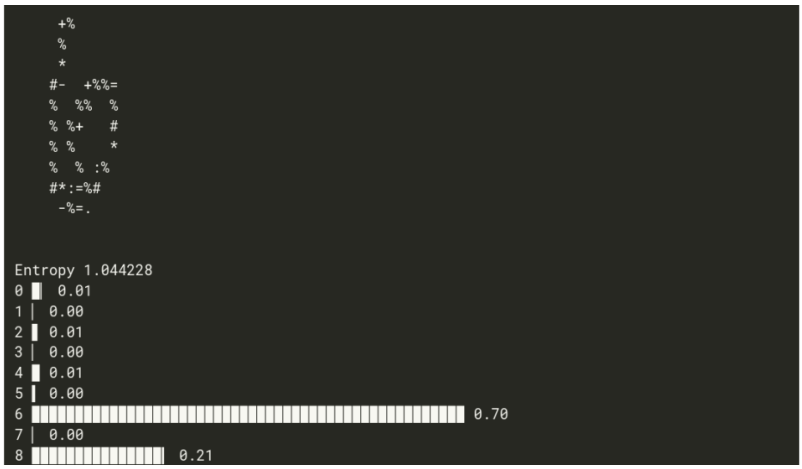
Если вы видите что-то необычное, просто сообщите мне.

Day 8: Model Uncertainty Estimation



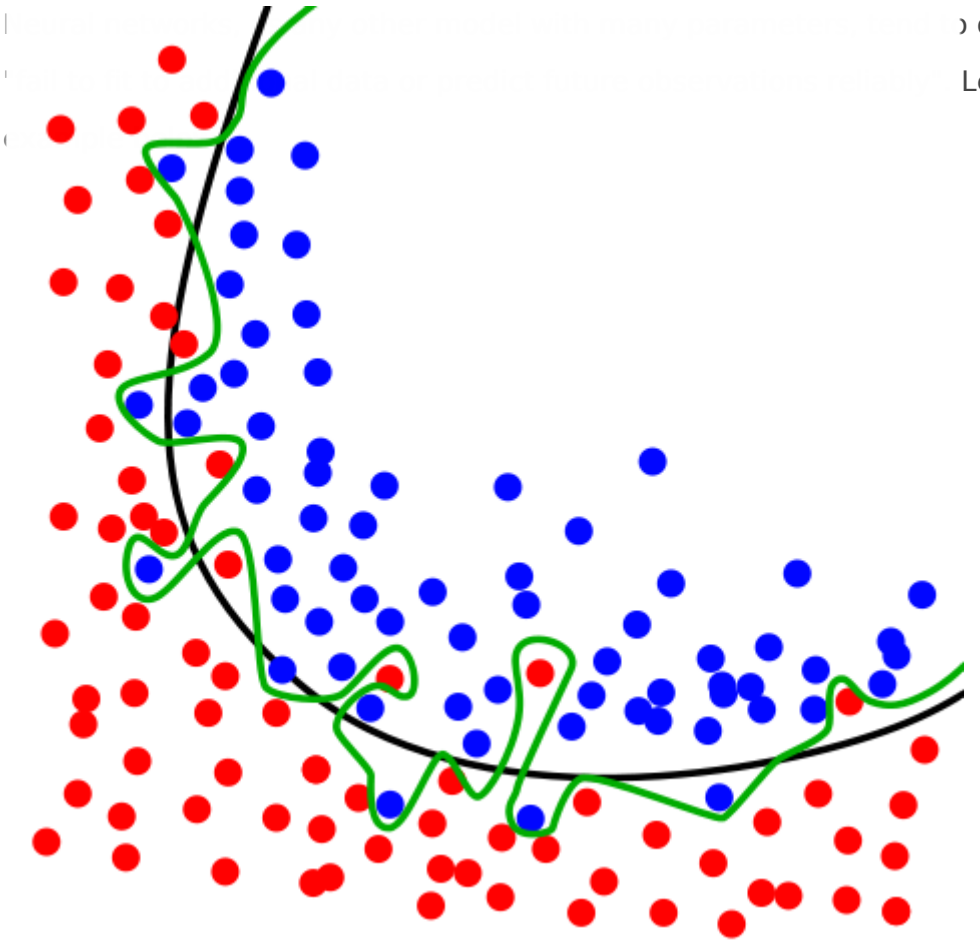
Wouldn't it be nice if the model also told us which predictions are not reliable? Can this be done even on unseen data? The good news is yes, and even on new, completely unseen data. It is also simple to implement in practice. A canonical example is in a medical setting. By measuring model uncertainty, the doctor can learn how reliable is their AI-assisted patient's diagnosis. This allows the doctor to make a better informed decision whether to trust the model or not. And potentially save someone's life.

Today we build upon Day 7 and we continue our journey with Hasktorch:

1. We will introduce a Dropout layer.
2. We will compute on a graphics processing unit (GPU).
3. We will also show how to load and save models.
4. We will train with Adam optimizer.
5. And finally we will talk about model uncertainty estimation.

The complete project is also available on Github.

Dropout Layer



Overfitting. Overfitting. Credit

Ignacio Icke, CC BY-SA 4.0

The green line is a decision boundary created by an overfitted model. We see that the model tries to memorize every possible data point. However, it fails to generalize. To ameliorate the situation, we perform a so-called regularization. That is a technique that helps to prevent overfitting. In the image above, the black line is a decision boundary of a regularized model.

One of regularization techniques for artificial neural networks is called dropout or dilution. Its principle of operation is quite simple. During neural network training, we randomly disconnect a fraction of neurons with some probability. It turns out that dropout conditioning results in more reliable neural network models.

A Neural Network with Dropout

The data structures `MLP` (learnable parameters) and `MLPSpec` (number of neurons) remain unchanged. However, we will need to modify the `mlp` function (full network) to include a Dropout layer. If we inspect `dropout :: Double -> Bool -> Tensor -> IO Tensor` type, we see that it accepts three arguments: a `Double` probability of dropout, a `Bool` that turns this layer on or off, and a data `Tensor`. Typically, we turn the dropout on during the training and off during the inference stage.

However, the biggest distinction between e.g. `relu` function and `dropout` is that `relu :: Tensor -> Tensor` is a pure function, i.e. it does not have any 'side-effects'. This means that every time when we call a pure function, the result will be the same. This is not the case with `dropout` that relies on an (external) random number generator, and therefore returns a new result each time. Therefore, its outcome is an `IO Tensor`.

One has to pay a particular attention to those `IO` functions, because they can change the state in the external world. This can be printing text on the screen, deleting a file, or launching missiles. Typically, we prefer to keep functions pure whenever possible, as function purity improves the reasoning about the program: It is a child's play to refactor (reorganize) a program consisting only of pure functions.

I find the so-called do-notation to be the most natural way to combine both pure functions and those with side-effects. The pure equations can be grouped under `let` keyword(s), while the side-effects are summoned with a special `<-` glue. This is how we integrate `dropout` in `mlp`. Note that now the outcome of `mlp` also becomes an `IO Tensor`.

```
mlp :: MLP -> Bool -> Tensor -> IO Tensor
mlp MLP {..} isStochastic x0 = do
  -- This subnetwork encapsulates the composition
  -- of pure functions
  let sub1 =
          linear fc1
          ~> relu
```

```
        ~> linear fc2
        ~> relu

  -- The dropout is applied to the output
  -- of the subnetwork
  x1 <- dropout
          0.1   -- Dropout probability
          isStochastic  -- Activate Dropout when in stochastic mode
          (sub1 x0)  -- Apply dropout to
                     -- the output of `relu` in layer 2

  -- Another linear layer
  let x2 = linear fc3 x1

  -- Finally, logSoftmax, which is numerically more stable
  -- compared to simple log(softmax(x2))
  return $ logSoftmax (Dim 1) x2
```

For model uncertainty estimation, it is empirically recommended to keep the dropout probability anywhere between 0.1 and 0.2.

Computing on a GPU

To transfer data onto a GPU, we use `toDevice :: ... => Device -> a -> a`. Below are helper methods to traverse data structures containing tensors (e.g. MLP) to convert those between devices.

```
toLocalModel :: forall a. HasTypes a Tensor => Device -> DType -> a -> a
toLocalModel device' dtype' = over (types @Tensor @a) (toDevice device')

fromLocalModel :: forall a. HasTypes a Tensor => a -> a
fromLocalModel = over (types @Tensor @a) (toDevice (Device CPU 0))
```

Below is a shortcut to transfer data to cuda:0 device, assuming the Float type.

```
toLocalModel' = toLocalModel (Device CUDA 0) Float
```

The train loop is almost the same as in the previous post, except a few changes. First, we convert training data to GPU with toLocalModel' (assuming that the model itself was already converted to GPU). Second, predic <- mlp model isTrain input is an IO action. Third, we manage optimizer's internal state1.

```
trainLoop
  :: Optimizer o
  => (MLP, o) -> LearningRate -> ListT IO (Tensor, Tensor) -> IO (MLP, o)
trainLoop (model0, opt0) lr = P.foldM step begin done. enumerateData
  where
    isTrain = True
    step :: Optimizer o => (MLP, o) -> ((Tensor, Tensor), Int) -> IO (MLP, o)
    step (model, opt) args = do
      let ((input, label), iter) = toLocalModel' args
      predic <- mlp model isTrain input
      let loss = nllLoss' label predic
      -- Print loss every 100 batches
      when (iter `mod` 100 == 0) $ do
        putStrLn
          $ printf "Batch: %d | Loss: %.2f" iter (asValue loss :: Float)
      runStep model opt loss lr
    done = pure
    begin = pure (model0, opt0)
```

We also modify the `train` function to use Adam optimizer with `mkAdam`:

1. 0 is the initial iteration number (then internally increased by the optimizer).
2. We provide beta1 and beta2 values.
3. flattenParameters net0 are needed to get the shapes of the trained parameters momenta. See also Day 2 for more details.

```
train :: V.MNIST IO -> Int -> MLP -> IO MLP
train trainMnist epochs net0 = do
    (net', _) <- foldLoop (net0, optimizer) epochs $ \(net', optState) _ ->
      runContT (streamFromMap dsetOpt trainMnist)
      $ trainLoop (net', optState) lr. fst
    return net'
  where
    dsetOpt = datasetOpts workers
```

```
    workers = 2
    lr = 1e-4  -- Learning rate
    optimizer = mkAdam 0 beta1 beta2 (flattenParameters net0)
    beta1 = 0.9
    beta2 = 0.999
```

Here is a function to get model accuracy:

```
accuracy :: MLP -> ListT IO (Tensor, Tensor) -> IO Float
accuracy net = P.foldM step begin done. enumerateData
  where
    step :: (Int, Int) -> ((Tensor, Tensor), Int) -> IO (Int, Int)
    step (ac, total) args = do
      let ((input, labels), _) = toLocalModel' args
      -- Compute predictions
      predic <- let stochastic = False
                in argmax (Dim 1) RemoveDim
                     <$> mlp net stochastic input

      let correct = asValue
                      -- Sum those elements
                      $ sumDim (Dim 0) RemoveDim Int64
                      -- Find correct predictions
                      $ predic `eq` labels

      let batchSize = head $ shape predic
      return (ac + correct, total + batchSize)

    -- When done folding, compute the accuracy
    done (ac, total) = pure $ fromIntegral ac / fromIntegral total

    -- Initial errors and totals
    begin = pure (0, 0)

testAccuracy :: V.MNIST IO -> MLP -> IO Float
testAccuracy testStream net = do
    runContT (streamFromMap (datasetOpts 2) testStream) $ accuracy net. fst
```

Below we provide the MLP specification: number of neurons in each layer.

```
spec = MLPSpec 784 300 50 10
```

Saving and Loading the Model

Before we can save the model, we have to make the weight tensors dependent first:

```
save' :: MLP -> FilePath -> IO ()
save' net = save (map toDependent. flattenParameters $ net)
```

The inverse is true for model loading. We also replace parameters in a newly generated model with the one we have just loaded:

```
load' :: FilePath -> IO MLP
load' fpath = do
  params <- mapM makeIndependent <=< load $ fpath
  net0 <- sample spec
  return $ replaceParameters net0 params
```

Load the MNIST data:

```
(trainData, testData) <- initMnist "data"
```

Train a new model:

```
-- A train "loader"
trainMnistStream = V.MNIST { batchSize = 256, mnistData = trainData }
net0 <- toLocalModel' <$> sample spec

epochs = 5
net' <- train trainMnistStream epochs net0
```

Saving the model:

```
save' net' "weights.bin"
```

To load a pretrained model:

```
net <- load' "weights.bin"
```

We can verify the model's accuracy:

```
-- A test "loader"
testMnistStream = V.MNIST { batchSize = 1000, mnistData = testData }

ac <- testAccuracy testMnistStream net
putStrLn $ "Accuracy " ++ show ac
```

```
Accuracy 0.9245
```

The accuracy is not tremendous, but it can be improved by introducing batch norm, convolutional layers, and training longer. We are about to discuss model uncertainty estimation and this accuracy is good enough.

Predictive Entropy

Model uncertainties are obtained as:

$$\mathbb{H}(y|\mathbf{x}) = -\sum_{c} p(y = c|\mathbf{x}) \log p(y = c|\mathbf{x}),$$

where y is label, x – input image, c – class, p – probability. We call H predictive entropy. And it is the very dropout technique that helps us to estimate those uncertainties. All we need to do is to collect several predictions in the stochastic mode (i.e. dropout enabled) and apply the formula from above.

```
predictiveEntropy :: Tensor -> Float
predictiveEntropy predictions =
  let epsilon = 1e-45
      a = meanDim (Dim 0) RemoveDim Float predictions
      b = Torch.log $ a + epsilon
  in asValue $ negate $ sumAll $ a * b
```

Visualizing Softmax Predictions

To get a better feeling what model outputs look like, it would be nice to visualize the softmax output as a histogram or a bar chart. For instance

```
bar ["apples", "oranges", "kiwis"] [50, 100, 25]
```

```
apples  ████████████████████████ 50.00
oranges ████████████████████████████████████████████████ 100.00
kiwis   ████████████ 25.00
```

Now, we would like to display an image, the predictive entropy, and the softmax output, followed by predic[illegible] use the following identity: $$e^{\ln(\text{softmax}(x))} = \text{softmax}(x),$$ that is softmax = exp. logSoftmax.

```
displayImage :: MLP -> (Tensor, Tensor) -> IO ()
displayImage model (testImg, testLabel) = do
  let repeatN = 20
      stochastic = True
  preds <- forM [1..repeatN] $ \_ -> exp  -- logSoftmax -> softmax
                                     <$> mlp model stochastic testImg
  pred0 <- mlp model (not stochastic) testImg
  let entropy = predictiveEntropy $ Torch.cat (Dim 0) preds

  -- Select only the images with high entropy
  when (entropy > 0.9) $ do
      V.dispImage testImg
      putStr "Entropy "
      print entropy
      -- exp. logSoftmax = softmax
```

```
      bar (map show [0..9]) (asValue $ flattenAll $ exp pred0 :: [Float])
      putStrLn $ "Model        : " ++ (show. argmax (Dim 1) RemoveDim. exp $ pred0)
      putStrLn $ "Ground Truth : " ++ show testLabel
```

Note that below we show only some of those images the model is uncertain about (entropy > 0.9)

```
testMnistStream = V.MNIST {batchSize = 1, mnistData = testData}
forM_ [0 .. 200] $ displayImage (fromLocalModel net) <=< getItem testMnistStream
```

```
     +%
     %
     *
    #-  +%%=
    %  %%  %
    % %+   #
    % %    *
    %  % :%
    #*:=%#
     -%=.



Entropy 1.044228
0 ▉  0.01
1 ▏ 0.00
2 ▌ 0.01
3 ▏ 0.00
4 ▉ 0.01
5 ▌ 0.00
6 ▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉ 0.70
7 ▏ 0.00
8 ▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉ 0.21
9 ▉▉▉▉ 0.05
Model        : Tensor Int64 [1] [ 6]
Ground Truth : Tensor Int64 [1] [ 6]



      .#%#.
    %%+:
     %
     %..
```

Entropy 1.2909155

0 | 0.00

1 | 0.00

2 ▌ 0.00

3 ▉▉▉▉▉▉▉▉ 0.07

4 | 0.00

5 ▉▉▉▌ 0.44

6 | 0.00

7 ▌ 0.00

8 ▉▉ 0.47

9 ▉ 0.01

Model : Tensor Int64 [1] [8]

Ground Truth : Tensor Int64 [1] [5]



Entropy 1.3325933

0 ▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▌ 0.19

1 | 0.00

2 | 0.00

3 | 0.00

```
4 ▉▉▉
```

```
     =
     %  =
     %  %
     -= @
      = %
        %
        %
        %


Entropy 0.9308149
0 |  0.00
1 |  0.00
2 |  0.00
3 ▏ 0.01
4 ████████████████████▏  0.29
5 ▎ 0.00
6 |  0.00
7 ▎ 0.00
8 ▉  0.02
9 ████████████████████████████████████████████████ 0.67
Model        : Tensor Int64 [1] [ 9]
Ground Truth : Tensor Int64 [1] [ 9]




        #
      % #
      % *
      % =
     %%@%
     *  %
        %
        %
        %
        =


Entropy 1.39582
0 |  0.00
1 ▉  0.01
2 |  0.00
```

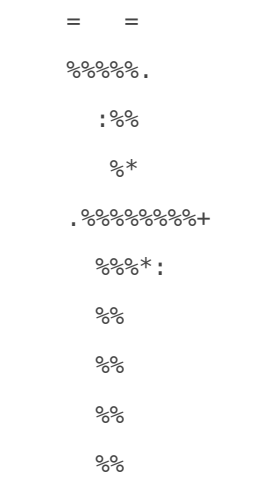
```
3 ██████ 0.06
4 ████████████████████████████████████████████████ 0.48
5 ████████████████▉ 0.17
6 ████ 0.04
7▏ 0.00
8 ▉ 0.02
9 ████████████████████▉ 0.22
Model        : Tensor Int64 [1] [ 4]
Ground Truth : Tensor Int64 [1] [ 4]



       .#%@
       %%%%=
      +%. %#
       %%%%:
        %%%
       -%%
      -%%
     .%%
     %%-
     %*


Entropy 1.0009595
0▏ 0.00
1▏ 0.00
2▏ 0.00
3 ▉ 0.02
4▏ 0.00
5▎ 0.00
6▏ 0.00
7 ████████████████████████▉ 0.35
8 ▊ 0.01
9 ████████████████████████████████████████████████ 0.62
Model        : Tensor Int64 [1] [ 9]
Ground Truth : Tensor Int64 [1] [ 9]
```

```
     %##%
    :%+%%.
    -%  %:
    -%  %+
     +  %+
        %+
        %+
        %#
        %%
        .+
Entropy 1.0057298
0 |  0.00
1 |  0.00
2 |  0.00
3 |  0.00
4 |  0.00
5 ▉▉  0.03
6 |  0.00
7 ▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉  0.33
8 |  0.00
9 ▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉ 0.63
Model        : Tensor Int64 [1] [ 9]
Ground Truth : Tensor Int64 [1] [ 7]



   %%%%%
      .%
      %.
    =%%%+
    %   %# -
         %%.
        *%-
       %:%
      %-%=
      %%-

Entropy 1.0500848
0 ▉▉▉▉  0.07
1 ▏ 0.00
```

```
2 ▏ 0.00
3 ████████████████████████████████████████████████ 0.79
4 ███ 0.04
5 ████▏ 0.05
6 ▏ 0.00
7 ▌ 0.01
8 ▏ 0.00
9 ██ 0.03
Model        : Tensor Int64 [1] [ 3]
Ground Truth : Tensor Int64 [1] [ 3]
```



```
Entropy 1.590256
0 ▏ 0.00
1 ████████████████████████████████████████████████ 0.36
2 ▏ 0.00
3 ████████████ 0.10
4 ████████████████████████████████████████████▏ 0.32
5 ████▏ 0.02
6 ▏ 0.00
7 ▌ 0.00
8 ████████████████▏ 0.12
9 ██████████▏ 0.07
Model        : Tensor Int64 [1] [ 1]
Ground Truth : Tensor Int64 [1] [ 1]
```



Entropy 0.9592192

0 | 0.00

1 ▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉ 0.28

2 ▌ 0.01

3 ▌ 0.00

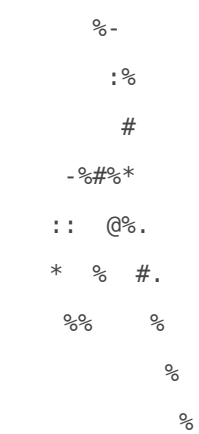
4 | 0.00

5 | 0.00

6 ▌ 0.01

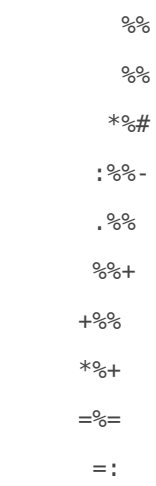
7 ▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉

```
1 |  0.00
2 ████████████████████████████████████████████████ 0.48
3 |  0.00
4 |  0.00
5 |  0.00
6 |  0.00
7 ███████████████████████████████████████████████▏ 0.47
8 ████ 0.03
9 ▉  0.01
Model        : Tensor Int64 [1] [ 2]
Ground Truth : Tensor Int64 [1] [ 2]



      -
    :%%%-
   :%   %
   +:   :%-
  -%     *%
  *:      %*
  ==      *%
   *      :%
   #::..:*%%
    :%*%%-:



Entropy 1.3647958
0 ████████████████████████████████████████████████ 0.50
1 |  0.00
2 ██████████████████████▏  0.23
3 ██████████████████████▉ 0.23
4 |  0.00
5 ███▏  0.03
6 |  0.00
7 |  0.00
8 |  0.00
9 ▉  0.01
Model        : Tensor Int64 [1] [ 0]
Ground Truth : Tensor Int64 [1] [ 0]
```

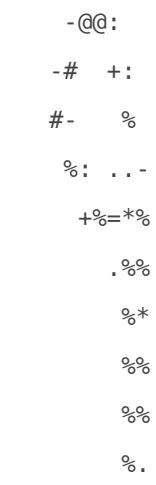


```
Entropy 1.1518966
0 ▉▉▉  0.06
1 ▌ 0.01
2 ▊ 0.01
3 ▏ 0.00
4 ▉▉▉ 0.05
5 ▏ 0.00
6 ▏ 0.00
7 ▏ 0.00
8 ▌ 0.01
9 ▉▉▉
```

```
0 ▉▉ 0.02
1 ▏ 0.00
2 ▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉ 0.29
3 ▍ 0.00
4 ▏ 0.00
5 ▏ 0.00
6 ▏ 0.00
7 ▏ 0.00
8 ▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉ 0.59
9 ▉▉▉▉▉▉▉▉ 0.10
Model        : Tensor Int64 [1] [ 8]
Ground Truth : Tensor Int64 [1] [ 9]



      --%:
     .   %
         %:
     ** .%
      *%%.
      %%*%
     %*  %
     %   %
     %  %:
     %%%:



Entropy 1.0862491
0 ▏ 0.00
1 ▉▉▉ 0.03
2 ▉▉▉▉▉ 0.05
3 ▏ 0.00
4 ▏ 0.00
5 ▉ 0.01
6 ▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉ 0.42
7 ▏ 0.00
8 ▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉ 0.50
9 ▏ 0.00
Model        : Tensor Int64 [1] [ 8]
Ground Truth : Tensor Int64 [1] [ 8]
```



```
Entropy 1.0085171
0 |  0.00
1 ████████████████████████████████████████████████ 0.81
2 ▏ 0.00
3 ▏ 0.01
4 ▏ 0.00
5 |  0.00
6 |  0.00
7 ██████████  0.16
8 ▏ 0.01
9 ▏ 0.01
Model        : Tensor Int64 [1] [ 1]
Ground Truth : Tensor Int64 [1] [ 1]
```



```
Entropy 1.5438546
0 |  0.00
1 |  0.00
2 ████ 0.03
3 ████████████████▏ 0.14
4 ██████ 0.05
5 ▊ 0.01
6 |  0.00
7 █████ 0.04
8 ████████████████████████████████▏ 0.31
9 ████████████████████████████████████████████ 0.42
Model        : Tensor Int64 [1] [ 9]
Ground Truth : Tensor Int64 [1] [ 9]
```

Reflecting on softmax outputs above we can state that

1. Softmax output alone is not enough to estimate the model uncertainty. We can observe wrong predictions even when the margin between the top and second-best guess is large.
2. Sometimes prediction and ground truth coincide. So why the entropy is high? We actually need to inspect such cases in more details. The first point is well illustrated by this example:



```
Entropy 1.1518966
0 ███▏ 0.06
1 ▌ 0.01
```

```
2 ▌ 0.01
3 │  0.00
4 ███ 0.05
5 │  0.00
6 │  0.00
7 │  0.00
8 ▌ 0.01
9 ████████████████████████████████████████████████ 0.86
Model        : Tensor Int64 [1] [ 9]
Ground Truth : Tensor Int64 [1] [ 2]
```

To illustrate the last point, let us take a closer look at cases with high entropy. By running several realizations of the stochatic model, we can verify if the model has any "doubt" by selecting different answers.

```
displayImage' :: MLP -> (Tensor, Tensor) -> IO ()
displayImage' model (testImg, testLabel) = do
  let repeatN = 10
  pred' <- forM [1..repeatN] $ \_ -> exp  -- logSoftmax -> softMax
                                     <$> mlp model True testImg
  pred0 <- mlp model False testImg
  let entropy = predictiveEntropy $ Torch.cat (Dim 0) pred'

  V.dispImage testImg
  putStr "Entropy "
  print entropy
  forM_ pred' ( \pred ->
      putStrLn ""
      >> bar (map show [0..9]) (asValue $ flattenAll pred :: [Float]) )
  putStrLn $ "Model        : " ++ (show. argmax (Dim 1) RemoveDim. exp $ pred0)
  putStrLn $ "Ground Truth : " ++ show testLabel
```

The first example from above (dataset index 11) gives this:

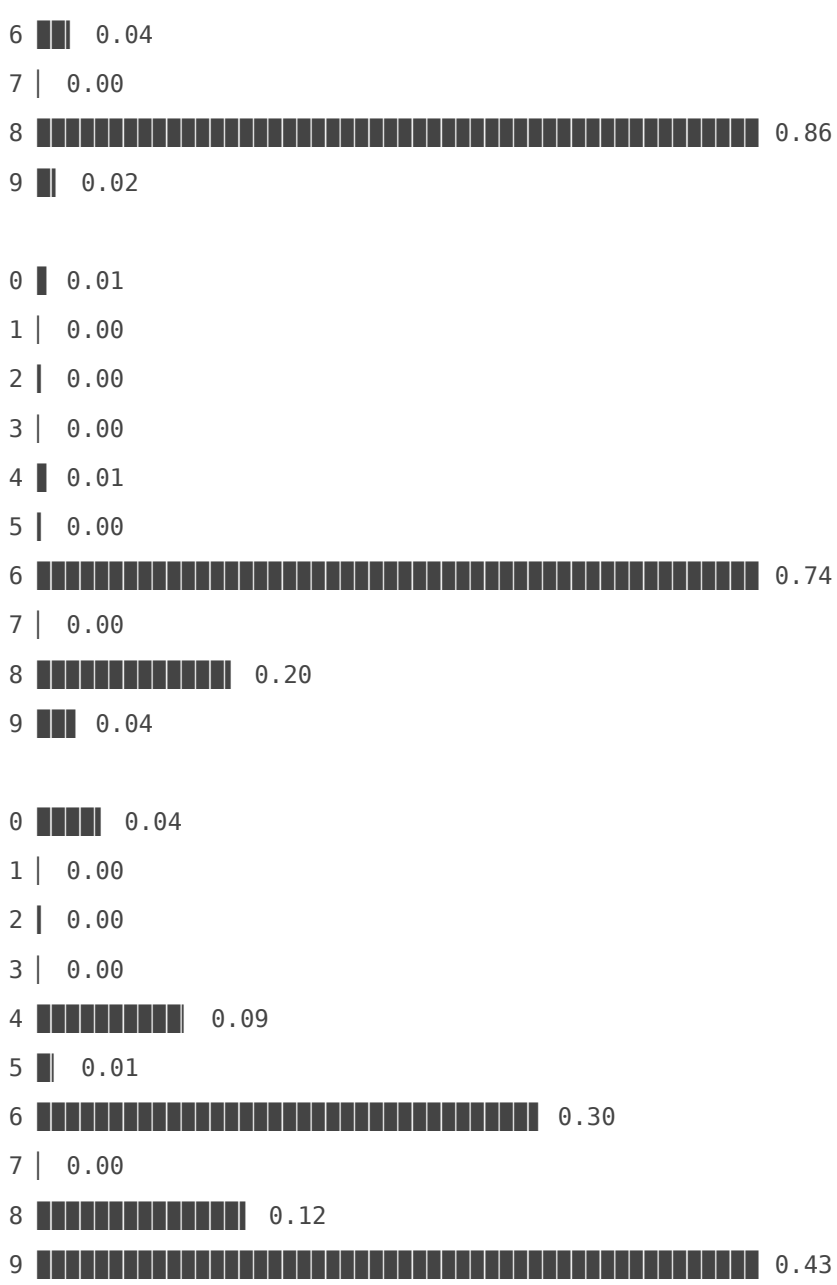
```
(displayImage' (fromLocalModel net) <=< getItem testMnistStream) 11
```

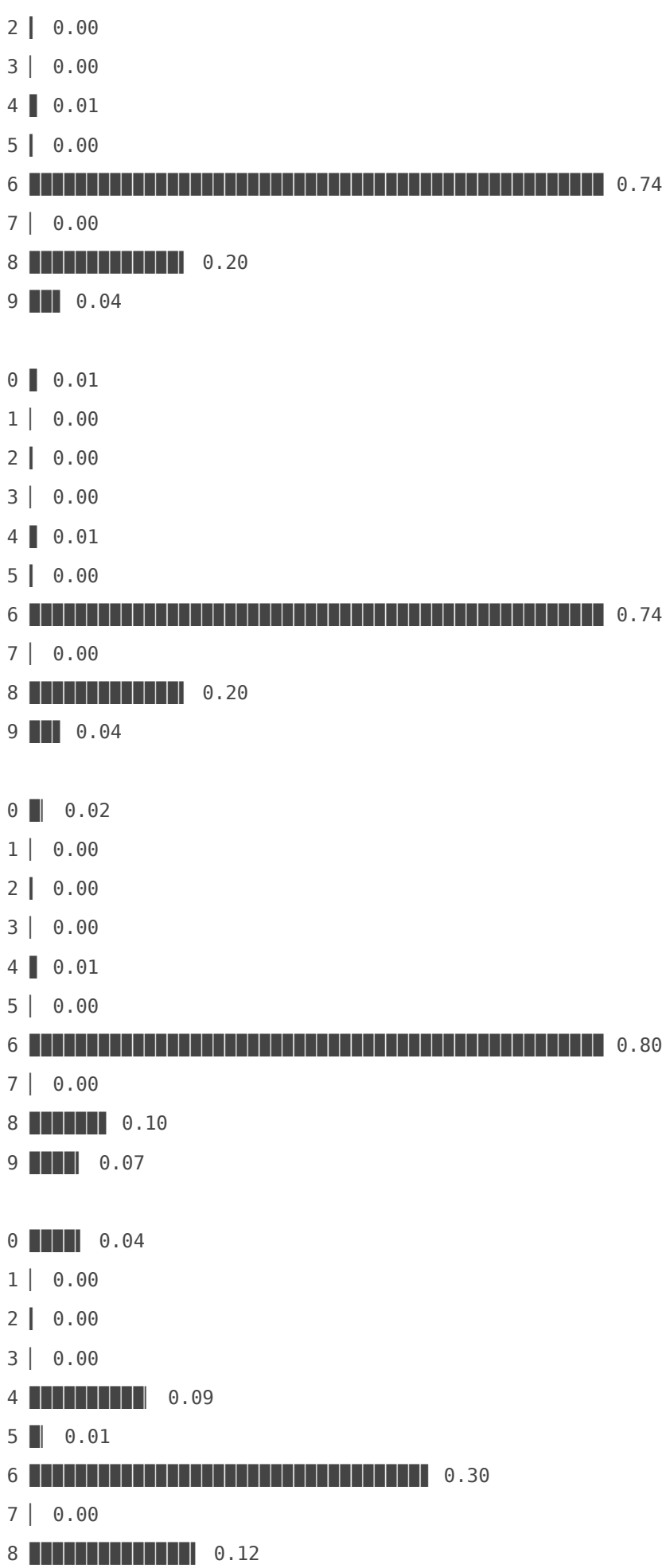
```


    +%
    %
```



Entropy 1.1085687

0 ▏ 0.00
1 ▏ 0.00
2 ▏ 0.00
3 ▏ 0.00
4 ▏ 0.00
5 ▏ 0.00
6 ▉▉▉


6 ▉▉▏ 0.04
7 ▏ 0.00
8 ▉▉▉


2 ▏ 0.00
3 ▏ 0.00
4 ▌ 0.01
5 ▏ 0.00
6 ▉▉▉

```
9 ████████████████████████████████████████████████ 0.43
Model        : Tensor Int64 [1] [ 6]
Ground Truth : Tensor Int64 [1] [ 6]
```

Wow! The model sometimes "sees" digit 6, sometimes digit 8, and sometimes digit 9! For the contrast, here is how predictions with low entropy typically look like.

```
(displayImage' (fromLocalModel net) <=< getItem testMnistStream) 0
```

```
   #%%*****
      ::: %
         %:
        :%
        #:
       :%
       %.
      #=
     :%.
     =#
Entropy 4.8037423e-4

0 | 0.00
1 | 0.00
2 | 0.00
3 | 0.00
4 | 0.00
5 | 0.00
6 | 0.00
7 ████████████████████████████████████████████████ 1.00
8 | 0.00
9 | 0.00

0 | 0.00
1 | 0.00
2 | 0.00
3 | 0.00
4 | 0.00
5 | 0.00
6 | 0.00
7 ████████████████████████████████████████████████ 1.00
```

```
8 |  0.00
9 |  0.00

0 |  0.00
1 |  0.00
2 |  0.00
3 |  0.00
4 |  0.00
5 |  0.00
6 |  0.00
7 ████████████████████████████████████████████████ 1.00
8 |  0.00
9 |  0.00

0 |  0.00
1 |  0.00
2 |  0.00
3 |  0.00
4 |  0.00
5 |  0.00
6 |  0.00
7 ████████████████████████████████████████████████ 1.00
8 |  0.00
9 |  0.00

0 |  0.00
1 |  0.00
2 |  0.00
3 |  0.00
4 |  0.00
5 |  0.00
6 |  0.00
7 ████████████████████████████████████████████████ 1.00
8 |  0.00
9 |  0.00

0 |  0.00
1 |  0.00
2 |  0.00
3 |  0.00
```

```
4 |  0.00
5 |  0.00
6 |  0.00
7 ████████████████████████████████████████████████ 1.00
8 |  0.00
9 |  0.00

0 |  0.00
1 |  0.00
2 |  0.00
3 |  0.00
4 |  0.00
5 |  0.00
6 |  0.00
7 ████████████████████████████████████████████████ 1.00
8 |  0.00
9 |  0.00

0 |  0.00
1 |  0.00
2 |  0.00
3 |  0.00
4 |  0.00
5 |  0.00
6 |  0.00
7 ████████████████████████████████████████████████ 1.00
8 |  0.00
9 |  0.00

0 |  0.00
1 |  0.00
2 |  0.00
3 |  0.00
4 |  0.00
5 |  0.00
6 |  0.00
7 ████████████████████████████████████████████████ 1.00
8 |  0.00
9 |  0.00
```

```
0 |  0.00
1 |  0.00
2 |  0.00
3 |  0.00
4 |  0.00
5 |  0.00
6 |  0.00
7 ▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇ 1.00
8 |  0.00
9 |  0.00
Model        : Tensor Int64 [1] [ 7]
Ground Truth : Tensor Int64 [1] [ 7]
```

The model always "sees" digit 7. That is why the predictive entropy is low. Note that the results are model-dependent. Therefore we also share the weights for reproducibility. However, every realization of the stochastic model might still be different, especially in those cases where the entropy is high.

Find the complete project on Github. For suggestions about the content feel free to open a new issue.

Summary

I hope you are now convinced that model's uncertainty estimation is an invaluable tool. This simple technique is essential when applying deep learning for real-life decision making. This post also develops on how to use Hasktorch library in practice. Notably, it is very straightforward to run computations on a GPU. Overall, Hasktorch can be used for real-world deep learning. The code is well-structured and relies on a mature Torch library. On the other hand, it would be desirable to capture high-level patterns so that the user does not need to think about low-level concepts such as dependent and independent tensors, for example. The end user should be able to simply apply `save net "weights.bin"` and `mynet <- load "weights.bin"` without any indirections. The same reasoning applies to the `trainLoop`, i.e. the user does not need to reinvent it every time. Eventually, a higher-level package on top of Hasktorch should capture the best practices, similar to PyTorch Lightning or fast.ai.

Now your turn: explore image recognition with AlexNet convolutional network and have fun!

Edit 27/04/2022: The original version from 23/04 did not correctly handle optimizer's internal state. Therefore, `train` and `trainLoop` were fixed. You will find the updated notebook on Github.

Learn More

- Improving neural networks by preventing co-adaptation of feature detectors https://arxiv.org/pdf/1207.0580.pdf
- Dropout: A Simple Way to Prevent Neural Networks from Overfitting https://www.jmlr.org/papers/volume15/srivastava14a/srivastava14a.pdf
- Tutorial: Dropout as Regularization and Bayesian Approximation https://xuwd11.github.io/Dropout_Tutorial_in_PyTorch/
- Two Simple Ways To Measure Your Model’s Uncertainty https://towardsdatascience.com/2-easy-ways-to-measure-your-image-classification-models-uncertainty-1c489fefaec8
- Uncertainty in Deep Learning, Yarin Gal http://mlg.eng.cam.ac.uk/yarin/thesis/thesis.pdf
- AlexNet example in Hasktorch https://github.com/hasktorch/hasktorch/tree/master/examples/alexNet

---

Revision #5
Created 2022-09-03 08:53:45 UTC by gasick
Updated 2026-03-02 19:51:46 UTC by gasick