CS187 Lab 1-5: Scaling up: Torchtext and PyTorch

October 12, 2020

[ ]:
```
# Please do not change this cell because some hidden tests might depend on it.
import os

# Otter grader does not handle ! commands well, so we define and use our
# own function to execute shell commands.
def shell(commands, warn=True):
    """Executes the string `commands` as a sequence of shell commands.

       Prints the result to stdout and returns the exit status.
       Provides a printed warning on non-zero exit status unless `warn`
       flag is unset.
    """
    file = os.popen(commands)
    print (file.read().rstrip('\n'))
    exit_status = file.close()
    if warn and exit_status != None:
        print(f"Completed with errors. Exit status: {exit_status}\n")
    return exit_status

shell("""
ls requirements.txt >/dev/null 2>&1
if [ ! $? = 0 ]; then
 rm -rf .tmp
 git clone https://github.com/cs187-2020/lab1-5.git .tmp
 mv .tmp/tests ./
 mv .tmp/requirements.txt ./
 rm -rf .tmp
fi
pip install -q -r requirements.txt
""")
```

[ ]:
```
# Initialize Otter
import otter
grader = otter.Notebook()
```

1 CS187

1.1 Lab 1-5 – Scaling up: Torchtext and PyTorch

The pipeline for NLP applications based on supervised machine learning involves several standard components:

1. Loading of annotated textual corpora.
2. Tokenization and normalization of the text.
3. Distributing instances into subcorpora, for instance, training, development, and test corpora.
4. Training of models on training data, using development data for model selection.
5. Evaluation of the models on test data.

Rather than recapitulate all of these component tasks for each application, standard packages have been developed to facilitate them. In order to facilitate your own experimentation, it's time to make use of some of these packages to scale up your ability to build and test models. That is the subject of this lab.

Torchtext datasets provide a uniform system for establishing dataset objects that contain multiple examples, each of which may have multiple named fields. These fields themselves have specifications that tell whether the data in that field is sequential (like text sequences) or simple (like class labels); whether and how to preprocess, tokenize, or postprocess the data; and many other properties. Dataset objects can be easily split into parts (training and test, for instance), or turned into a sequence of small batches for processing by models.

This lab provides an introduction to using `torchtext` and PyTorch in preparation for its appearance in later labs and project segments.

After this lab, you should be able to

- Read `torchtext` code and understand what it is intending to accomplish.
- Run experiments training and testing simple feed-forward neural networks using PyTorch.

New bits of Python used for the first time in the *solution set* for this lab, and which you may therefore find useful:

- `torch.Tensor.backward`

1.2 Preparation – Loading packages and data

[ ]:
```
import copy
import math
import random
import matplotlib.pyplot as plt
import numpy as np
import os
import re
import sys
import torch
import torch.distributions as ds
```

```
import torch.nn as nn
import torch.nn.functional as F
import torchtext as tt
import warnings

from torch import optim
```

[ ]:
```
%matplotlib inline
plt.style.use('tableau-colorblind10')

# Random seed
random_seed = 1234
random.seed(random_seed)
torch.manual_seed(random_seed)
## GPU check
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(device)
```

[ ]:
```
## Turn off annoying torchtext warnings about pending deprecations
warnings.filterwarnings("ignore", module="torchtext", category=UserWarning)
```

2 Part 1: Manipulating text corpora with `torchtext`

You'll use `torchtext` to load the *Green Eggs and Ham* (GEaH) dataset.

We start with reading in the data and performing some ad hoc cleaning (removing comment lines and blank lines).

[ ]:
```
def strip_file(text):
    """strip #comments and empty lines from `text` string"""
    result = ""
    for line in text.split("\n"):
        line = line.strip()              # trim whitespace
        line = re.sub('#.*$', '', line)  # trim comments
        if line != '':                   # drop blank lines
            result += line + '\n'
    return result

# Read the GEaH data and write out a corresponding TSV file
shell('wget -nv -N -P data "https://github.com/nlp-course/data/raw/master/Seuss/
 ↪seuss - 1960 - green eggs and ham.txt"')
with open('data/seuss - 1960 - green eggs and ham.txt', 'r') as fin:
    with open('data/geah.tsv', 'w') as fout:
        fout.write(strip_file(fin.read()))
```

2.1 Constructing training and test datasets

Take a look at the file `geah.tsv`, which we've just processed and placed into the sibling `data` folder.

[ ]:
```
shell('head "data/geah.tsv"')
```

Notice the structure of this corpus. Each line contains a sentence from the book, preceded by a label that provides the speaker of that sentence. The speaker and sentence are separated by a tab character. The data is thus set up properly for a `torchtext.data.TabularDataset` using its `"TSV"` (tab-separated values) format.

In order to establish a `torchtext.data.TabularDataset` object for dealing with the GEaH dataset, you'll first need to establish the two fields (via `torchtext.data.Field`), one for the label (the speaker) and one for the text, which you should call `LABEL` and `TEXT`, respectively. These fields are used for mapping from strings to ids or vice versa, which we'll use for problem set 1.

When setting up a field with `torchtext.data.Field`, you'll want to consider whether you want to further specify the values for the various keyword arguments listed here, or to leave the default values.

> You may see generated messages warning that the Field class or other parts of `torchtext` are deprecated. You can ignore these warnings. We're using the legacy `torchtext` interface rather than the one taking over in the next `torchtext` release.

With respect to the tokenization of the text field, you should use the `torchtext` "basic_english" tokenizer introduced in lab 1-1 for the text field and lowercase all tokens.

[ ]:
```
#TODO: Define one `tt.data.Field` for processing the label
# and another for processing the text using the "basic_english"
# tokenizer and **lowercase** all tokens
LABEL = (
    ...
)

TEXT = (
    ...
)
fields = [("label", LABEL), ("text", TEXT)]
```

[ ]:
```
grader.check("fields_setup")
```

Now, you can set up the dataset using `torchtext.data.TabularDataset`. It should look for the TSV data in the file `data/geah.tsv`, and should use the `fields` defined above.

[ ]:
```
#TODO: Set up the dataset using `tt.data.TabularDataset`
# Note that you need to use `fields` for fields.
geah = ...
```

[ ]:
```
grader.check("dataset_setup")
```

All `torchtext.data.Dataset` objects have a `split` method that splits the dataset into two or three pieces, for instance, to have a separate training and test set. Use the `split` method to generate a 70%/30% split of the GEaH corpus into two subsets called `train` and `test`.

[ ]:
```
#TODO: Split geah into 70% training data and 30% test data
train, test = \
    ...
```

[ ]:
```
grader.check("dataset_split")
```

Fields can have a ssociated with them a *vocabulary* consisiting of all of the possible values that are used in that field in a particular dataset. The vocabulary establishes the kind of indexing scheme between types and indices that we explored in lab 1-1. We will use the training corpus to establish vocabularies for the two fields `LABEL` and `TEXT` using the `build_vocab` method.

[ ]:
```
LABEL.build_vocab(train.label)
TEXT.build_vocab(train.text)
```

The `TEXT` and `LABEL` fields, objects of class `torchtext.data.Field`, now have vocabularies associated with them, accessible in their respective `vocab` fields. How many elements are there in these vocabularies? You can use the `len` function to find out.

[ ]:
```
#TODO: Calculate the sizes of LABEL.vocab and TEXT.vocab
label_vocab_size = ...
text_vocab_size = ...
```

[ ]:
```
grader.check("vocab_sizes")
```

[ ]:
```
print(f"label vocabulary size is {label_vocab_size}\n"
      f"text vocabulary size is {text_vocab_size}")
```

Why are there three elements in the `LABEL` vocabulary, given that there are only two speakers, Guy and Sam? We can find out by examining the `LABEL` vocabulary more closely. `Field` vocabulary objects have an especially useful `stoi` data field, whose value is a dictionary that maps the elements of the vocabulary to integer index representations of the elements. Let's take a look.

[ ]:
```
LABEL.vocab.stoi
```

Question: What is the unexpected third element in the label vocabulary? Why do you think it's there?

Type your answer here, replacing this text.

> As described in more detail in the `torchtext` documentation, there's also an `itos` data field for conversion of vocabulary items from the index representation to the original values and a `freqs` data field that keeps a frequency distribution for the items in the vocabulary as a `collections.Counter` object.

2.2 Operations over datasets

We now have training and test datasets. You can experiment with the kinds of operations you'll need to do to implement models like Naive Bayes or logistic regression.

For instance, you can inspect an example from the dataset.

[ ]:
```
example = train[1] # the second instance
print (f"text: {example.text}\n"
       f"label: {example.label}")
```

You might also need to iterate over the different class labels (the vocabulary of the `LABEL` field) or the word types (the vocabulary of the `TEXT` field). Define a function that iterates over the vocabulary of a field and prints each one out like this:

```
>>> print_vocab(LABEL)
<unk>
GUY
SAM
```

[ ]:
```
#TODO
def print_vocab(field):
    ...
```

[ ]:
```
grader.check("print_vocab")
```

We can use the `print_vocab` function to print out the different class labels in the `LABEL` field.

[ ]:
```
print_vocab(LABEL)
```

Other simple calculations that will be useful in implementing the various models:

1. Counting how many instances there are in a dataset.
2. Counting how many instances of a certain class there are in a dataset.
3. Counting how many tokens of a certain type there are in the text of an instance.

Let's write functions for these. They'll come in handy in the first problem set.

[ ]:
```
#TODO - 1. Counting how many instances there are in a dataset.
def count_instances(dataset):
    ...
```

[ ]:
```
grader.check("count_instances")
```

[ ]:
```
#TODO - 2. Counting how many instances of a certain class there are in a dataset.
def count_instances_class(dataset, label):
    ...
```

[ ]:
```
grader.check("count_instances_class")
```

```
[ ]: #TODO - 3. Counting how many tokens of a certain type there are in the text of␣
     ↪an instance.
     # Note: recall that you can access the list of tokens using `instance.text`
     def count_tokens_instance(instance, tokentype):
         ...
```

```
[ ]: grader.check("count_tokens_instances")
```

Recall that the purpose of fields is to map back and forth between strings and word ids. Below provides an example of how to do that.

```
[ ]: example = train[1]
     text = example.text
     word_ids = [TEXT.vocab.stoi[word] for word in text]
     print (f"Mapped to word ids: {word_ids}\n"
            f"Mapped back: {[TEXT.vocab.itos[id] for id in word_ids]}")
     label = example.label
     label_id = LABEL.vocab.stoi[label]
     print (f"Label id: {label_id}\n"
            f"label: {LABEL.vocab.itos[label_id]}")
```

3 Part 2: Training and testing with PyTorch

Past labs have shown that all of the detail about

- establishing models and their parameters,
- using them to calculate the outputs for some inputs,
- training them to optimize the parameters via stochastic gradient descent, and
- evaluating them by testing on held-out data

is tedious to manage. Fortunately, it is also so formulaic, at least for a certain class of models, that general tools can be deployed to manage the process. In the remainder of this lab, you'll use one such tool, PyTorch. For simplicity, rather than a natural-language task, you'll be training a model to fit a curve; it has an especially simple structure: one scalar input and one scalar output.

3.1 Generating training and test data

We start by generating some training and test data. The data is generated as a noisy sine function, calculated by the function `sinusoid`.

```
[ ]: def sinusoid(x, amplitude=1., phase=0., frequency=1., noise=0.):
         """Returns the values on input(s) `x` of a sinusoid determined by␣
     ↪`amplitude`,
            `phase`, and angular `frequency`, with some added normal noise with␣
     ↪variance
            given by `noise`."""
```

```
    normal_noise = ds.normal.Normal(torch.tensor([0.0]), torch.tensor([noise]))
    noise_sample = torch.tensor([normal_noise.sample() for i in range(len(x))])
    y = amplitude * torch.sin(x * frequency + phase) + noise_sample
    return y
```

We can generate data for training and testing by sampling this function.

[ ]:
```
def sample_input(func, count, bound, **kwargs):
    """Returns `count` samples of x-y pairs of function `func`, with the x
       values sampled uniformly between +/-`bound`. The `kwargs` are passed
       on to `func`."""
    input_unif = ds.uniform.Uniform(-bound, +bound)
    x = input_unif.sample(torch.Size([count]))
    y = func(x, **kwargs)
    return x, y
```

To give a sense of what a data sample looks like, we plot a sample of 100 points.

[ ]:
```
def plot_sample(data):
    """Plots `data` given as a single pair of inputs and outputs."""
    X, Y = data
    plt.plot(X.numpy(), Y.numpy(), '.')
    plt.xlabel('Input')
    plt.ylabel('Output')
    # we cannot use plt.show() because otter-grader does not support it
```

[ ]:
```
plot_sample(sample_input(sinusoid, 100, 5, noise=0.1))
```

3.2 Specifying a feed-forward neural network

The model that we will train to predict the output of this function based on a sample will consist of a series of sublayers as depicted in the figure at right. At the bottom of the figure, we start with $\mathbf{x}$, the scalar input (of dimensionality 1 as shown in the "shape" designation). The first layer is a perceptron layer, with a linear sublayer (with weights $\mathbf{U}$) followed by a sigmoid sublayer. Since U is of dimensionality $1 \times D$, the output is a vector of dimensionality D. (We refer to D as the hidden dimension.) Then comes another perceptron layer with output of dimensionality D. Finally, a single linear layer reduces the dimensionality back to the predicted scalar output $\tilde{y}$ of dimensionality 1. The loss is calculated as the mean square error of $\tilde{y}$ relative to y. (In this case, taking the mean for a single example is irrelevant, since y is a scalar, though for batches, the mean will be taken over the batch.)

We define a class `FFNN` (**f**eed-**f**orward **n**eural **n**etwork), which inherits from the `nn.Module` class, PyTorch's class for neural network models. It takes an argument `hidden_dim` which is the size of the hidden layers, D in the figure.

The parameters of this model – the values that will be adjusted to minimize the loss – are the elements of the tensors $\mathbf{U}$, $\mathbf{V}$, and $\mathbf{W}$. Those parameters are created and tracked when the corre-

sponding sublayers are created using `nn.Linear`. That's the wonder of using PyTorch – so much happens under the hood.

[ ]:
```
class FFNN(nn.Module):
    def __init__(self, hidden_dim, init_low=-2, init_high=2):
        super().__init__()
        # dimensionality of hidden layers
        self.hidden_dim = hidden_dim
        # establishing the sublayers -- two perceptrons and a final
        # linear layer -- and their parameters
        self.sublayer1 = nn.Linear(1, hidden_dim)          # U: 1 X D
        self.sublayer2 = nn.Sigmoid()
        self.sublayer3 = nn.Linear(hidden_dim, hidden_dim) # V: D X D
        self.sublayer4 = nn.Sigmoid()
        self.sublayer5 = nn.Linear(hidden_dim, 1)          # W: D X 1

        # initialize parameters randomly
        torch.manual_seed(random_seed)
        for p in self.parameters():
            p.data.uniform_(init_low, init_high)
        # save a copy of the parameters to allow resetting
        self.init_state = copy.deepcopy(self.state_dict())

    # Resetting state: If you want to rerun a model, say, with a different
    # training regime, you can reset the model's parameter state using
    #    model.reset_state()
    # before retraining, e.g.,
    #    train_model(model, criterion, optim, train_data, n_epochs=50)
    def reset_state(self):
        self.load_state_dict(copy.deepcopy(self.init_state))

    def forward(self, x):
        # first perceptron layer
        z = self.sublayer2(self.sublayer1(x))
        # second perceptron layer
        z_prime = self.sublayer4(self.sublayer3(z))
        # final linear layer
        return self.sublayer5(z_prime)
```

We can build a model by instantiating the `FFNN` class. We'll do so with a hidden dimension of 4, being careful to move the model with its parameters to the device we're using for calculations (a GPU if one is available, as on Google Colab).

[ ]:
```
HIDDEN_DIMENSION = 4
model = FFNN(HIDDEN_DIMENSION).to(device)
```

We specify the criterion to be optimized as the mean square error loss function provided by PyTorch.

[ ]:
```
criterion = nn.MSELoss(reduction='mean')
```

3.3 Evaluating data according to a model

To evaluate how well the model performs on some test data, we run the model forward on the x values and compute the loss relative to the y values. We define a function `eval_model` to carry out this calculation.

[ ]:
```
def eval_model(model, criterion, data):
    """Applies the `model` to the x values in the `data` and returns the
       loss relative to the y values in the `data` along with the predicted
       y values."""
    model.eval()                            # turn on evaluation mode
    with torch.no_grad():                   # turn off propagating gradients
        X, Y = data                         # extract x and y values
        X = X.view(-1, 1).to(device)        # convert x and y to column vectors
        Y = Y.view(-1, 1).to(device)        # ...and move them to the device
        predictions = model(X)              # calculate the predicted y values
        loss = criterion(predictions, Y)    # see how far off they are
    return loss.item(), predictions
```

All that remains is training the model. We'll use one of PyTorch's built in optimizers, the `Adam` optimizer. We set a few parameters for the training process: the learning rate, the number of "epochs" (passes through the training data) to perform, and the number of examples to train on at a time (the "batch size").

[ ]:
```
## Parameters of the training regimen
LEARNING_RATE = 0.003
NUMBER_EPOCHS = 25
BATCH_SIZE = 20

## Choices for optimizers:

# Stochastic Gradient Descent (SGD) optimizer
# optim = torch.optim.SGD(model.parameters(), lr = learning_rate)

# The Adam optimizer, as described in the paper:
# Kingma and Ba. 2014. Adam: A Method for Stochastic Optimization.
# [https://arxiv.org/abs/1412.6980]
optim = torch.optim.Adam(model.parameters(), lr = LEARNING_RATE)
```

3.4 Training the parameters of a model

Finally, we get to the function to train the parameters of the model so as to best fit the predictions to the actual values. We've provided the code, except for a few lines that you'll need to provide (marked `#TODO`), making use of some of the tools defined above. Those lines, which form the heart

of the computation, calculate "forwards" to get the output predictions for the inputs, calculate the loss for those predictions, and calculate "backwards" the gradients of the loss for each of the parameters of the model. This sets up the optimizer to take a step of updating the parameters, making use of the calculated gradients to determine the direction to step. The saved gradients can then be zeroed and the process repeated.

> Note: The code we're asking you to write is *tiny*. If you find yourself writing more than a short line of code per `#TODO`, you're missing something.

[ ]:
```
def train_model(model, criterion, optimizer, data,
                n_epochs=NUMBER_EPOCHS, batch_size=BATCH_SIZE):
    """Optimizes the parameters of the `model` by minimizing the `criterion`
       on the training `data`, using the `optimizer` algorithm for updates."""
    model.train()                       # Turn on training mode

    X, Y = data
    trainX_len = len(X)

    for epoch in range(n_epochs):
        loss_per_epoch = 0.
        for batch_i in range(int(trainX_len/batch_size)):
            optimizer.zero_grad()      # new batch; zero the gradients of the␣
 ↪parameters

            # Input tensors and their corresponding output values for this batch
            batch_X = (X[batch_i * batch_size
                         : (batch_i+1) * batch_size] # extract examples in batch
                       .view(-1, 1)                  # reshape to column vector
                       .to(device)                   # move to device
                      )
            batch_Y = (Y[batch_i * batch_size
                         : (batch_i+1) * batch_size]
                       .view(-1, 1)
                       .to(device)
                      )

            #TODO: Calculate predictions for the x values in this batch
            predictions = ...

            #TODO: Calculate the loss for the predictions
            loss = ...

            #TODO: Perform backpropagation to calculate gradients
            ...

            # Update all parameters
            optimizer.step()
```

```
            loss_per_epoch += loss.item()

        print(f"Epoch: {epoch+1:3d}  Total loss: {loss_per_epoch:8.3f}")
```

3.5 Putting it all together

Let's try it out. We start by generating some training and test data. The training data will be 10,000 samples of a noisy sinusoid. The test data, 100 samples from the same sinusoid, will be noise-free, so we can see how close the predictions are to noise-free outputs.

[ ]:
```
train_data = sample_input(sinusoid, 10000, 5., frequency=1.5, noise=0.05)
test_data = sample_input(sinusoid, 100, 5., frequency=1.5)

plot_sample(train_data)
```

We train the model.

[ ]:
```
model.reset_state()
train_model(model, criterion, optim, train_data)
```

...and test the trained model by evaluating it on the the test data.

[ ]:
```
loss, predictions = eval_model(model, criterion, test_data)
```

[ ]:
```
grader.check("model_reduces_loss")
```

We can see how well the model works by plotting the test data (circles) along with the predicted values (crosses).

[ ]:
```
def visualize_predictions(data, predictions):
    X, Y = data

    # Plot the actual output values
    plt.plot(X.cpu().numpy(), Y.cpu().numpy(), '.', label = 'Target Values')

    # Plot the predicted output values
    predictions = predictions.view(-1, 1)
    plt.plot(X.cpu().numpy(), predictions.cpu().numpy(), 'x', label =␣
 ↪'Predictions')

    plt.xlabel('Input')
    plt.ylabel('Output')
    plt.legend()
    # we cannot use plt.show() because otter-grader does not support it
```

[ ]:
```
# Visualize the predictions
visualize_predictions(test_data, predictions)
```

3.6 Trying different models

Now that we have the infrastructure, try experimenting with different models. Here are a few things you might play with. (No need to try them all.) What happens if you change the hidden dimension, increasing it to 8 or decreasing it to 2? What happens if you drop the middle layer? What about no middle layer but a much higher hidden dimension size? Does running for more epochs improve performance? Does the SGD optimizer work better or worse than the Adam optimizer?

Perform any experimentation in cells below this point, so you don't modify the cells above that are being unit tested.

Question: What conclusions have you drawn from your experimentation?

Type your answer here, replacing this text.

3.7 Lab debrief – for consensus submission only

Question: We're interested in any thoughts your group has about this lab so that we can improve this lab for later years, and to inform later labs for this year. Please list any issues that arose or comments you have to improve the lab. Useful things to comment on include the following:

- Was the lab too long or too short?
- Were the readings appropriate for the lab?
- Was it clear (at least after you completed the lab) what the points of the exercises were?
- Are there additions or changes you think would make the lab better?

Type your answer here, replacing this text.

4 End of lab 1-5

To double-check your work, the cell below will rerun all of the autograder tests.

[ ]:
```
grader.check_all()
```