50.039 – Theory and Practice of Deep Learning

Alex

Week 8: Recurrent Neural Networks 1

1 Inclass / Homework

Due: Sat 30th of March
Take the pytorch tutorial `https://pytorch.org/tutorials/intermediate/char_rnn_classification_tutorial.html`, but

- in the first step make the code run with an LSTM instead of the custom RNN in there.
- in the second step adapt it for use with a batchsize > 1, that is at least of 2
- learning goals:
  - using RNN for scoring of a whole sentence
  - using an LSTM in code
  - modifying code for larger batchsizes

for the LSTM: `https://pytorch.org/docs/stable/nn.html` read the documentation on input and output. it can process the whole sequence in one.

unlike in the pytorch code example, since you know the inputted sequence in advance, you can input the sequence as a whole in the `def forward(self,x):` of your model class, not using a slow python for loop!

```
output, (hn, cn)= self.lstm(x,(h0,c0))
```

use an optimizer, unlike what it is done in the tutorial

```
optimizer = optim.SGD(net.parameters(), lr=0.1)
```

feel free to add weight decay or momentum here ... above initialization is just an example.

1.1 Rough structure

- prepare a class which can return pair of (name,label) from the train set and from the test set.
- you will need a code to measure prediction accuracy from the outputs of your network (can do after you have defined your network and when you are able to print your outputs) – since the output is a softmax classifier on top of the LSTM hidden state, argmax will give you the predicted class.

in a training for-loop of an epoch for batch size 1:

- fetch a pair of (name,label) from something similar to a dataset class
- `optimizer.zero_grad()`
- init memory cell and hidden state of the LSTM to zero ⟵ this is different from cnns!
- run forward pass of the model, for the lstm see above, but you need to transform the hidden state by an fc layer (what is the output dimensionality?)
  - read on inputs and outputd of the LSTM to understand how to program the forward pass of your model
- compute loss between prediction and label
- `loss.backward()`
- `optimizer.step()` to update parameters

in a validation for-loop of an epoch for batch size 1:

- fetch a pair of (name,label) from something similar to a dataset class
- set torch.no_grad() as with ... : environment
- init memory cell and hidden state of the LSTM to zero
- run forward pass of the model, for the lstm see above, but you need to transform the hidden state by an fc layer (what is the output dimensionality?)
- compute accuracy between prediction and label (and NLL loss, but that only for comparing train - test learning curves)

With that take the example and modify it.

1.2 Some steps from tutorial to runing code

1. it can be useful for clarity to at first take the code, and put all dangling commands and variables that are globally defined in the file into functions for the sake of encapsulation, and just call one function from

```
if __name__ == '__main__':
  run()
```

This may increase readability of the code a lot.

2. Data access. You can (but not must) write your own python iterator. Below is an example that generates batchsize 1 samples of feature-label-pairs. If you want to construct a batch of 2, then you can overwrite the `return` statement, or call the iterator twice. `def __next__(self):` defines what happens if you call once

```
for element in youriterator:
  #whatever
  pass
```

Here is an example iterator implementation. No perfection guaranteed.

```
class iteratefromdict():
    def __init__(self, adict,all_categories):
      #whatever here

    def num(self):
      return len(self.namlab)

    def __iter__(self):
        return self

    def __next__(self):

        if self.ct==len(self.namlab):
            # reset before raising iteration for reusal
            np.random.shuffle(self.namlab)
            self.ct=0
            #raise
            raise StopIteration()
        else:
            self.ct+=1
            #return feature-label pair here
            return self.namlab[self.ct-1][0],self.namlab[self.ct-1][1]
```

Such an iterator class allows you to iterate in training and in testing loops:

```
for counter,(feature,label) in enumerate(youriterator):
  #whatever
  pass
```

It functions like a dataloader (not dataset) which provides you batchsize-1-pairs of features and labels. This stuff should train reasonably fast on a notebook, AWS should be not necessary unless your notebook is very old or a toy.

- Task1 Report performance on the test set in terms of accuracy for 1 and 2 layer LSTMs with at least 3 different sizes of the hidden layer, and submit code of course.

- Task2 - part A: do it after task1 in case that you cannot finish it, not together. Take your result from task1 and adapt the code to run with a batchsize of at least 2. For this small network it does not matter but for your deep learning tasks after SUTD it is necessary.

  The problem about batchsizes here is: sequences have varying length! You cannot just clutch them together. You can use `torch.nn.pack_padded_sequence` or `torch.nn.utils.rnn.pack_sequence` (which I have used in my implementation).

  If you are going to use *output* from the LSTM, and not (hn, cn), then you will also need to use `pad_packed_sequence` on it, to make the output into some readable tensor again.

  You can create a larger batch, by sampling n times from your iterator, and constructing a batch from it, or you create a specialized iterator for that.

- Task2 - part B: compare training and test error and test accuracy as a function of epoch for one choice of LSTM (e.g. 1 layer, 200 hidden dimensions) for batchsize 1,10,30. Plot it.