AI535 Deep Learning – Homework 3

Convolutionl and Recurrent Models in PyTorch

Overview and Objectives. In this homework, we are going to get started in PyTorch with convolutional and recurrent models to better explore these topics and help get you set up for your projects.

How to Do This Assignment. All questions that require a response will be in blue-gray task boxes. We prefer solutions typeset in LaTeX but will accept scanned written work if it is legible. If a TA can't read your work, they can't give you credit. Submit your solutions to Canvas as a PDF and your code as a `.py` file.

Advice. This homework involved using PyTorch so you'll need an environment with it installed. For the things we are doing here, Google Colab should be sufficient for training and evaluation.

1 Convolutional Networks [20pts]

In this section, you'll build a convolutional network in PyTorch for the 3-class CIFAR dataset we worked on in homework 2. The skeleton file `cifar_pytorch.py` has all the infrastructure to train a PyTorch model on this dataset, but the model definition is incomplete:

```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        # define layers

    def forward(self, x):
        # define forward pass
        return x
```

▶ Q1 [13 pt] Implement a simple convolutional neural network with the following structure:

- `conv1` - Conv with 32 3x3 filters and padding 1
- `relu1` - ReLU
- `conv2` - Conv with 32 3x3 filters and padding 1
- `pool1` – 2x2 Max Pool
- `relu2` - ReLU
- `conv3` - Conv with 64 3x3 filters and padding 1
- `relu3` - ReLU
- `conv4` - Conv with 64 3x3 filters and padding 1
- `relu4` - ReLU
- `pool2` – 2x2 Max Pool
- `fc1` – Linear with 512 neurons
- `relu5` - ReLU
- `fc2` – Linear with 3 neurons

This model should easily get validation accuracy in the upper 80s or above when trained. Provide the generated training plot in your report.

Transitioning from the output `out` of `pool2` to the fully-connected layer `fc1` requires flattening the spatial feature for each instance in the batch. This can be accomplished by calling

```
out = out.view(-1, self.num_flat_features(x))
```

where `num_flat_features` is a helper function you should examine carefully.

▶ Q2 [2 pt] Add a batch normalization layer after `fc1` but before `relu5` and repeat the training process. Include your training plot and comment on the differences you observe.

Training a model isn't useful if we just throw it away afterwards. Let's work on saving/loading the model.

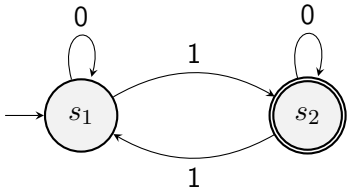
▶ Q3 [5 pt] Modify the training code to save the model weights after an epoch if the validation accuracy is lower than for previously saved weights. Then, write a new file `test.py` that loads these weights and evaluates accuracy on the test set when run.

2 Simple Sequence Models for Parity [30pts]

Consider a univariate LSTM defined given the following equations. We refer to this model as univariate because all inputs, outputs, and weights are scalars. $\sigma(\cdot)$ is a sigmoid activation.

$$
\begin{aligned}
i_t &= \sigma(w_{ix}x_t + w_{ih}h_{t-1} + b_i) && (1)\\
f_t &= \sigma(w_{fx}x_t + w_{fh}h_{t-1} + b_f) && (2)\\
o_t &= \sigma(w_{ox}x_t + w_{oh}h_{t-1} + b_o) && (3)\\
g_t &= \tanh(w_{gx}x_t + w_{gh}h_{t-1} + b_g) && (4)\\
c_t &= f_t c_{t-1} + i_t g_t && (5)\\
h_t &= o_t \tanh(c_t) && (6)
\end{aligned}
$$

To ground this complicated looking model, we will consider a simple sequence classification problem – parity of binary strings. Given an arbitrarily-long binary string $b \in \{0,1\}^*$, classify whether it has an even or odd number of ones. For example all these strings have an even parity – "", `0000`, `101101` – and these an odd parity – `1`, `0011001`, `00011100`. Below is a simple DFA[1] that accepts strings with odd parity at the accepting state s_2. As it only has two states, it seems solving the parity classification problem only requires storing 1 bit of information in memory.



Another way to think about this computation is as a recursive application of `XOR` over time. If the parity up to element $t-1$ is 1 (odd) and we see a 1 for element t, the updated parity is 1 XOR $1 = 0$ (even). Likewise, if at time $t+1$ we observe another 1, 0 XOR $1 = 1$. In other words, PARITY$(X, t) = X[t]$ XOR PARITY$(X, t-1)$ and PARITY$(X, -1) = 0$ by definition. This sounds like the sort of operation we can do with an LSTM!

[1] https://en.wikipedia.org/wiki/Deterministic_finite_automaton

▶ Q4 [5 pt] Manually find weights and biases for the univariate LSTM defined above such that the final hidden state will be greater than or equal to 0.5 for odd parity strings and less than 0.5 for even parity. The parameters you must provide values for are w_{ix}, w_{ih}, b_i, w_{fx}, w_{fh}, b_f, w_{ox}, w_{oh}, b_o, w_{gx}, w_{gh}, b_g and are all scalars. The LSTM will take one bit of the string (0 or 1) as input x at each time step. A tester is set up in `univariate_tester.py` where you can enter your weights and check performance.

Hints: Some of the weights will likely be zero. Others may be large. Scale matters, larger weights will saturate the activation functions. Also note that `A XOR B` can be written as `(A OR B) AND (A NAND B)`. We recommend you work backwards and try to find where this sort of two-term structure could be implemented.

Through this exercise, we've shown that a 1-dimensional LSTM *is sufficient to solve parity of binary strings of arbitrary length.* In the next section, we'll try to actually train an LSTM for this task.

2.1 Learning to Copy Finite State Machines

Now, lets try to train an LSTM to perform the parity task we just did manually. The `driver_parity.py` file provides code for these experiments including a parity dataset shown below. During training, it generates all binary strings of length 0 to `max_length`. At inference, it generates all binary strings of length `max_length`.

```
class Parity(Dataset):

    def __init__(self,split="train", max_length=4):
      if split=="train":
        self.data = []
        for i in range(1,max_length+1):
          self.data += [torch.FloatTensor(seq) for seq in itertools.
    product([0,1], repeat=i)]
      else:
        self.data = [torch.FloatTensor(seq) for seq in itertools.
    product([0,1], repeat=max_length)]

    def __len__(self):
        return len(self.data)

    def __getitem__(self, idx):
        x = self.data[idx]
        y = x.sum() % 2
        return x,y
```

Listing 1: Parity Dataset

To train our network, we can build a DataLoader from our Parity dataset (see below). However, our dataset contains strings of variable length so we need to pass a custom function to our data loader describing how to combine elements in a sampled batch together.

```
    train = Parity(split='train', max_length=max_length)
    train_loader = DataLoader(train, batch_size=B, shuffle=True,
                              collate_fn=pad_collate)
```

Listing 2: Data loader

Our `pad_collate` function puts all the elements of a batch in a $B \times T_{\text{max}}$ tensor for inputs and a B tensor for outputs where T_{max} is the largest length in the batch. Shorter sequences are zero-padded. A batch of inputs looks like:

```
    tensor([[1., 1., 0., 0., 0.],
            [0., 1., 0., 1., 0.],
            [0., 0., 0., 0., 0.],
            [0., 1., 1., 0., 1.],
            [0., 1., 1., 0., 0.]], device='cuda:0')
```

Listing 3: Example input for a batc

▶ Q5 [15 pt] Implement the `ParityLSTM` class in `driver_parity.py`. Your model's `forward` function should process the batch of binary input strings and output a $B \times 2$ tensor y where $y_{b,0}$ is the score for the b^{th} element of the batch having an even parity and $y_{b,1}$ for odd parity. You may use any PyTorch-defined LSTM functions. Larger hidden state sizes will make for easier training in my experiments. Running `driver_parity.py` will train your model and output per-epoch training loss and accuracy. A correctly-implemented model should approach 100% accuracy on the training set. In your write-up for this question, describe any architectural choices you made.

Hint: Efficiently processing batches with variable input lengths requires some bookkeeping or else the LSTM will continue to process the padding for shorter sequences along with the content of longer ones. See pack_padded_sequence and pad_packed_sequence documentation in PyTorch.

▶ Q6 [5 pt] `driver_parity.py` also evaluates your trained model on binary sequences of length 0 to 20 and saves a corresponding plot of accuracy vs. length. Include this plot in your write-up and describe the trend you observe. Why might the model behave this way?

▶ Q7 [5 pt] We know from 1.1 that even a univariate LSTM (one with a scalar hidden state) can theoretically solve this problem. Run a few (3-4) experiments with different hidden state sizes, what is the smallest size for which you can still train to fit this dataset? Feel free to adjust any of the hyper-parameters in the optimization in the `train_model` function if you want. Describe any trends you saw in training or the generalization experiment as you reduced the model capacity.

Note: In developing this assignment, I never successfully got it down to a hidden dimension of 1. Lower dimensional settings tended to be sensitive to batch size and learning rate. Don't spend all night tuning this unless you've finished the rest of the assignment and have nothing better to do.

3 Debriefing (required in your report)

1. Approximately how many hours did you spend on this assignment?
2. Would you rate it as easy, moderate, or difficult?
3. Did you work on it mostly alone or did you discuss the problems with others?
4. How deeply do you feel you understand the material it covers (0%–100%)?
5. Any other comments?