Richard Yin

Tips for CSC485 Assignments

I had three assignments for this course. The first two (transition & graph-based parsing, word-sense disambiguation with LESK & BERT) involved using PyTorch and required GPUs in some way. The third required using TRALE for symbolic machine translation. Here's some help on getting started.

PyTorch

Think of layer i in a neural network as a function $h_i\colon \mathbb{R}^{d_{i-1}} \to \mathbb{R}^{d_i}$, where d_{i-1}, d_i are respectively the number of dimensions (ie. number of neurons) in the previous layer and current layer. Thus, a neural network can be written as

$$f(\mathbf{x}) = h_n\Big(h_{n-1}\Big(\cdots\big(h_2\big(h_1(\mathbf{x})\big)\big)\Big)\Big)$$

Typically, a layer function looks like this:

$$h_i(\mathbf{x}) = \phi_i\big(\ell_i(\mathbf{x})\big) = \phi_i(W\mathbf{x} + \mathbf{b})$$

Where $\ell_i\colon \mathbb{R}^{d_{i-1}} \to \mathbb{R}^{d_i}$ is a linear equation (and $W, \mathbf{b}$ contain a bunch of values to be learned, "**parameters**"; don't worry about how the learning works) and $\phi_i\colon \mathbb{R} \to \mathbb{R}$ is a nonlinear function ("**activation function**") that is applied element-wise (ie. on each dimension individually). We need nonlinear functions because they let f approximate a wider range of functions – applying a bunch of linear functions on each other is equivalent to just one linear function.

A neural network implementation might look like this:

```
class neural_net(nn.Module):
   def __init__(self) --> None:
        super().__init__()
        n_inputs = 30
        n_hidden = 50    # the in-between layers are called "hidden layers"
        n_outputs = 2
        self.f = nn.Sequential(
                    nn.Linear(n_inputs, n_hidden),
                    nn.ReLU(),
                    nn.Linear(n_hidden, n_hidden),
                    nn.ReLU(),
                    nn.Linear(n_hidden, n_outputs),
                )
    def forward(self, x: torch.Tensor) -> torch.Tensor:
        return self.f(x)
```

The `nn.Module` is the "base class" of any PyTorch model.

The `self.f` is a `nn.Sequential`, which is a wrapper that allows me to put a bunch of functions inside it without having to create separate variables for each function and apply them one after the other in `self.forward`. It contains:

- A linear function $\ell_1\colon \mathbb{R}^{30} \to \mathbb{R}^{50}$
- A ReLU activation function $\phi_1\colon \mathbb{R} \to \mathbb{R}$
- A linear function $\ell_2\colon \mathbb{R}^{50} \to \mathbb{R}^{50}$
- A ReLU activation function $\phi_2\colon \mathbb{R} \to \mathbb{R}$
- A linear function $\ell_3\colon \mathbb{R}^{50} \to \mathbb{R}^{2}$

So we have $f\colon \mathbb{R}^{30} \to \mathbb{R}^{2}$. I don't apply a ReLU activation function at the end because ReLU's codomain is $[0, \infty)$ and it's not always the case you want your output constrained to that.

Every `nn.something` function acts as a function, where the `forward` method dictates how it affects an input. For instance,

```
model = neural_net()
x = torch.rand(277, 30)
model(x)
```

Since `model` is a `nn.Module`, the last line is interpreted as `model.forward(x)`. The input is multidimensional (a **batch** of 277 data points that are 30-dimensional), so the output is multidimensional (277 predictions that are 2-dimensional).

In general, if you give any input shape as input, the model will only affect the last dimension. You must make sure the dimensions of the input fit with the dimensions of your model. If you ever need to modify the shape of a tensor:

```
v = torch.rand(10, 20)
v.shape                          # prints (10, 20)

# reshape can change the shape and even add/remove dimensions!
v = v.reshape(5, 2, 20)
v = v.reshape(200, 1)

# torch.swapaxes switches two dimensions of a tensor.
torch.swapaxes(v, 0, 1).shape    # prints (1, 200)

# torch.stack creates a new 1st dimension, given a list of inputs of the same shape
torch.stack([v, v]).shape        # prints (2, 200, 1)

# torch.concat can combine two tensors in a dimension - in the other dimensions, they must match in shape!
u = torch.stack(200, 2)
torch.concat([u, v], dim=1).shape       # prints (200, 3)
```

Note that a shape of (200, 1) is different from a shape of (200,)! Some math operations are really stingy about making sure the dimensions of two input tensors match exactly. Be sure to check out the documentation for more.

It can be annoying to train and use a GPU on the teach.cs servers. If you have a Google account, you can use **Google Colab**, which is built for Python. It grants you access to a free GPU (only for a limited time – a few hours per day. Stick to the regular CPU for most of your bugtesting).

If you use Google Colab, you may run into difficulties if you want to use any test files that the assignment gives you. Luckily, the Google Colab instance's OS is Linux, so you can reconfigure stuff like filepaths with "**line magic**" – type ! or % followed by a Linux command, and the line is treated like shell script. Each ! line will be treated as being run on a new terminal each time, so commands like `cd` only "apply" for one line. If you want to the change to persist, chain multiple commands in one line, using ; between each shell command. I think % lines use the same terminal every time.

TRALE

TRALE is based on ALE, which is based on Prolog, which is a logic programming language. Since I myself did not take CSC324, and the in-class + online resources were inadequate, I struggled. I have compiled my trial-and-error knowledge here so that you don't suffer like I did. This might not be 100% accurate though, so be warned.

For testing, I recommend loading TRALE without the GUI, like `/h/u2/csc485h/fall/pub/trale/trale -fsu`.
You will however need the GUI loaded to run the assignment's test code.
To compile your grammar `your_file.pl`, use `compile_gram(your_file).` with the period at the end.

TRALE seems to use `pothole_case` like Python. It is pretty flexible with white space.
Periods define the end of sentences; they are the equivalent of ; of Java and C.

TRALE has **types** and **features**, which are like classes and fields in OOP but you only define the type/feature's name.
Types can have **subtypes** (like subclasses) and features are set to types. Subtypes inherit the supertype's features.
The **supertype** (parent type) of all of types must be called `bot`, for reasons beyond my understanding.
There can be **recursion** where a subtype takes on a feature of type supertype.

```
% Defining a type with subtypes
type sub [subtype1, subtype2, subtype3, …].

% Defining a type's features
type intro [feature1:feature1type, …].

% Example type feature structure
bot sub [a, b] intro [f1:c].
    a sub [a1, a2] intro [f2:c, f3:d].
        d sub [d1, d2, d3].
    b intro [f4:e, f5:a].
```

In the example above:

- The types are a, a1, a2, b, c, d, d1, d2, d3, e
- Types a, a1, a2, b have feature f1 that takes on the type c
- Types a, a1, a2 have feature f2 that takes on the type c
- Types a, a1, a2 have feature f3 that takes on types d, d1, d2, d3
- Type b has features f4 and f5 that respectively take on types e and a
- Types c and e have been implicitly defined even though `sub` has not been used to define them.

The **lexicon** is the set of allowed words ("**tokens**") in the grammar. Each token must take on a type.
Think of the lexicon like a set of pre-instantiated versions of classes that you are allowed to use.

```
% Defining tokens
token ---> (type, feature1:feature1type, …).

% Continuing the example from above
hello ---> (a2, f1:c, f2:c, f3:d3).
world ---> (b, f1:c, f4:e, f5:a).
world ---> (d1, f1:c, f2:c, f3:d2).    % Two possible readings of the token "world"
```

You can verify your lexicon works by compiling, then typing `rec[token_name]` in the TRALE console, like so:

```
?- rec[hello].

STRING:
0 hello 1

CATEGORY:   % This is really useful for debugging, as you can make sure all features have the types you expect.
a2
F1  c
F2  c
F3  d3

ANOTHER? …
```

I think `rec` finds every possible interpretation/reading of a list of input tokens.
The `ANOTHER?` asks if you want to see another reading.

- If you enter `y`, it displays another reading. If no more exist, the result is `no` (unsuccessful termination).
- If you don't enter `y`, I believe this is interpreted as no, so the result is `yes` (successful termination).

Since we've only defined single tokens, `rec` only works for single tokens; rules will allow `rec` to process multiple tokens.
Note that `rec[world]` will have two possible readings because it has been defined twice.

The RHS of a token definition uses a notation whose name I don't know, so I'll call it **bracket matching**.

```
% Generic form
(type1, type2, …, feature1:feature1type, feature2:feature2type, …, Var1, Var2, …).

% "(…;…)" is a logical OR that allows a1 or a2. The bracket is important here.
% This is equivalent to (a, f1:c, f2:c, f3:d3) or defining "like" twice for a1 and a2.
like ---> ((a1;a2), f1:c, f2:c, f3:d3).

% If you don't specify any details about the feature values, they will be automatically filled by blanks.
and ---> (b).

% You can nest bracket matching if needed - note the use of "a" as a generic that allows "a1" and "a2".
subscribe ---> (b, f5:(a, f1:c, f2:c, f3:d1)).
```

Types can be substituted for bracket matchers. Think of them as selectors that accept anything that satisfies all of:

- It is of type `type1` and `type2`
- It has `feature1` and `feature2`, which are respectively equal to `feature1type` and `feature2type`.
- `Var1` and `Var2` are **global variables**, as indicated by beginning with a capital letter.
    - During `rec`, if some reading of a token satisfies this selector, the global variables inside the selector, `Var1` and `Var2`, will set to that reading of the token. This can be used in rules to enforce equalities.
    - TBH I'm not sure how global variables work when defined in a lexicon.

In TRALE, **rules** are used to create complex types from a combination of simpler types.

```
% Defining a rule
rule_name rule
(type1, …) ===>
cat> (type2, …),
cat> (type3, …).
```

The rule above says that a `(type1, …)` is made from the concatenation (`cat>`) of two tokens in a specific order:

- The 1st token satisfies `(type2, …)`
- The 2nd token satisfies `(type3, …)`.

Your assignment might feature `sem_head>`, which is the same as `cat>` but also marks which token is the "semantic head", usually of a phrase (eg. the noun (N) is the semantic head of the noun phrase (NP)). I think this marking needs to be made for the assignment's testing code to work? There's also `goal>` and `sem_goal>` (see below).

Consider this modified grammar from the [TRALE tutorial on Github]:

```
bot sub [cat, agr, person].
    cat sub [agreeable, s].
        s intro [agr:agr].
        agreeable sub [pn, v] intro [agr:agr].
    agr intro [person:person].
    person sub [first, second, third].

i ---> (pn, agr:person:first).       % Equivalent to (pn, agr:(person:first))
you ---> (pn, agr:person:second).    % The long form is (pn, agr:(agr, person:first))
he ---> (pn, agr:person:third).
she ---> (pn, agr:person:third).
sleep ---> (v, agr:person:first).    % Alternatively, (v, agr:(agr, person:(first;second))) works too
sleep ---> (v, agr:person:second).
sleeps ---> (v, agr:person:third).
```

```
intransitive rule
(s, agr:Agr) --->
cat> (pn, agr:Agr),
cat> (v, agr:Agr).
```

The rule enforces agreement: only `rec[i,sleep].`, `rec[you,sleep].`, `rec[he,sleeps].`, and `rec[she,sleeps].` work. It:

- Finds a pronoun (`pn`) and sets the global variable `Agr` to whatever the pronoun's agreement is.
- Finds a verb (`v`) and since `Agr` is already set, checks if the verb's agreement matches `Agr`.
    - If it doesn't match, the rule does not hold.
    - If it matches, the rule holds: create a sentence (`s`) with agreement equal to `Agr`.

In your assignment, you probably want to fill every feature of `s` with stuff that is carried over from component parts; this can be done with rules and global variables!

TRALE defines special syntax for **lists** when given the exact list definition below.

```
list sub [e_list, ne_list].
    ne_list intro [hd:bot, tl:list].
```

The list is similar to a linked list, with either an empty node (`e_list`) or non-empty node (`ne_list`).

- Non-empty nodes have a head (`hd`), the item it is storing (which can be any type, so `bot`).
- Non-empty nodes have a tail, which is what it points to. It can be any list node item.

I think the TRALE tutorial has a decent example on using lists, by extracting list elements node-by-node per rule.

TRALE also has special behavior for **emptiness**. The line of code below treats emptiness as a noun phrase (`np`):

```
empty (np, sem:Sem, gap:(sem:Sem, gap:none)).

% Example rule; see below
verb_phrase rule
vp ===>
cat> (v),
cat> (np).
```

If `rec[some_verb].` is called with the above grammar, `some_verb` has multiple readings:

- A verb (`v`), since `some_verb` would be defined as a verb
- A verb phrase (`vp`), as emptiness is a noun phrase (`np`) and it interprets `some_verb` as (`some_verb` + nothing)

TRALE will also parse `some_verb` by as (nothing + `some_verb`), but since there is no rule for `np` followed by `v`, it fails. The tutorial also has some good examples for using emptiness when parsing gaps.

The TRALE tutorial does a good job explaining **macros**, so I'll just explain about **goals**. Goals are like functions that only return true/false values, and can be enforce many things (eg. a list is empty, a global variable is not some type).

```
# In Python, you have:
def func(a, b):
   return a + b             # "Given the inputs, you return the output."

% In Prolog, you have:
func(a, b, a + b) if true    % "Given a input/output tuple, return true."
% The "if" can be followed by more functions on the goal's arguments, or by "true" if there's nothing else.

% Define is_empty to be true if the input is an empty list, [].
is_empty([]) if true.
```

Here is a list appending example. It's not important to understand it, but know which arguments are the inputs/outputs.

```
% This is a Prolog thing. You can find more info at
% https://www.cs.toronto.edu/~iq/csc324h/content/prolog_2_handout.pdf

append([], Xs, Xs) if true.                          % Base case
append([H|T1], L2,[H|T2]) if append(T1, L2, T2).     % Recursive case
% So the function can be read as append(l1, l2, output)
```

```
def append(l1, l2):
    if l1 == []:            # Base case
        return l2
    h, t1 = l1[0], l1[1:]   # Recursive case
    t2 = append(t1, l2)
    return [h] + t2
```

To use a goal inside a rule to enforce something, add `goal⟩` or `sem_goal⟩` (I think they behave the same), where the goal arguments are filled by global variables.

```
is_proper_vp(vp, []) if true.    % assume vp is a type
is_stupid(me) if true.           % assume me is a type
is_np(np) if true.               % assume np is a type

cool_rule rule
vp ===⟩
cat⟩ (v, subcat:List, Verb),
cat⟩ (np, NP),
goal⟩ \+(is_np(Verb)),                        % \+(…) is a logical NOT
goal⟩ (is_proper_vp(Verb, List);is_stupid(NP)).  % (…;…) is a logical OR
```

The above rule:

- Finds a verb (`v`) with a subcategorization feature (`subcat`), sets `List` to its type, then sets `Verb` to the verb itself
- Finds a noun phrase (`np`) and sets `NP` to the noun phrase itself.
- Checks that `Verb` is not of type `np`
- Checks that `Verb` is of type `vp` and `List` is `[]`, or `NP` is of type `me`.
- Produces a `vp` if all conditions are met.

That's all, folks! Hope this helped, and good luck on your assignment.