BEGINNER TUTORIALS
Deep Learning with PyTorch: A 60 Minute Blitz
PyTorch for former Torch users
Learning PyTorch with Examples
Transfer Learning tutorial
Data Loading and Processing Tutorial
Deep Learning for NLP with Pytorch
Introduction to PyTorch
Deep Learning with PyTorch
Word Embeddings: Encoding Lexical Semantics
Sequence Models and Long-Short Term Memory Networks
Advanced: Making Dynamic Decisions and the Bi-LSTM CRF
Dynamic versus Static Deep Learning Toolkits
Bi-LSTM Conditional Random Field Discussion
Implementation Notes
Exercise: A new loss function for discriminative tagging
INTERMEDIATE TUTORIALS
Classifying Names with a Character-Level RNN
Generating Names with a Character-Level RNN
Translation: A Character-Level Recurrent Sequence-to-Sequence Model

Advanced: Making Dynamic Decisions and the Bi-LSTM CRF

Dynamic versus Static Deep Learning Toolkits

Pytorch is a dynamic neural network kit. Another example of a dynamic kit is Dynet (I mention this because working with Pytorch and Dynet is similar. If you see an example in Dynet, it will probably help you implement it in Pytorch). The opposite is the static tool kit, which includes Theano, Keras, TensorFlow, etc. The core difference is the following:

- In a static toolkit, you define a computation graph once, compile it, and then stream instances to it.
- In a dynamic toolkit, you define a computation graph *for each instance*. It is never compiled and is executed on-the-fly

Without a lot of experience, it is difficult to appreciate the difference. One example is to suppose we want to build a deep constituent parser. Suppose our model involves roughly the following steps:

- We build the tree bottom-up
- Tag the root nodes (the words of the sentence)
- From there, use a recursive neural network and the embeddings of the words to find combinations that form constituents. Whenever you form a new constituent, use some sort of technique to get an embedding of the constituent. In this case, our network architecture will depend completely on the input sentence. In the sentence "The green cat scratched the wall", at some point in the model, we will want to combine the span $(i, j, r) = (1, 3, NP)$ (that is, an NP constituent spans word 1 to word 3, in this case "The green cat").

However, another sentence might be "Somewhere, the big fat cat scratched the wall". In this sentence, we will want to form the constituent $(2, 4, NP)$ at some point. The constituents we will want to form will depend on the instance. If we just compile the computation graph once, as in a static toolkit, it will be exceptionally difficult or impossible to program this logic. In a dynamic toolkit though, there isn't just 1 pre-defined computation graph. There can be a new computation graph for each instance, so this problem goes away.

Dynamic toolkits also have the advantage of being easier to debug and the code more closely resembling the host language (by that I mean that Pytorch and Dynet look more like actual Python code than Keras or Theano).

Bi-LSTM Conditional Random Field Discussion

For this section, we will see a full, complicated example of a Bi-LSTM Conditional Random Field for named-entity recognition. The LSTM tagger above is typically sufficient for part-of-speech tagging, but a sequence model like the CRF is really essential for strong performance on NER. Familiarity with CRF's is assumed. Although this name sounds scary, all the model is is a CRF but where an LSTM provides the features. This is an advanced model though, for more complicated than any earlier model in this tutorial. If you want to skip it, that is fine. To see if you're ready, see if you can:

- Write the recurrence for the viterbi variable at step i for tag k.
- Modify the above recurrence to compute the forward variables instead.
- Modify again the above recurrence to compute the forward variables in log-space (hint: log-sum-exp)

If you can do those three things, you should be able to understand the code below. Recall that the CRF computes a conditional probability. Let y be a tag sequence and x an input sequence of words. Then we compute

$$P(y|x) = \frac{\exp(\text{Score}(x, y))}{\sum_{y'} \exp(\text{Score}(x, y'))}$$

Where the score is determined by defining some log potentials $\log \psi_i(x, y)$ such that

$$\text{Score}(x, y) = \sum_i \log \psi_{\text{EMIT}}(y_i \rightarrow x_i) + \log \psi_{\text{TRANS}}(y_{i-1} \rightarrow y_i)$$

$$= \sum_i s_i[y_i] + \mathbf{P}_{y_{i-1},\,y_i}$$

where in this second expression, we think of the tags as being assigned unique non-negative indices.

If the above discussion was too brief, you can check out this write up from Michael Collins on CRFs.

Implementation Notes

The example below implements the forward algorithm in log space to compute the partition function, and the viterbi algorithm to decode. Backpropagation will compute the gradients automatically for us. We don't have to do anything by hand.

The implementation is not optimized. If you understand what is going on, you'll probably quickly see that iterating over the next tag in the forward algorithm could probably be done in one big operation. I wanted to code to be more readable. If you want to make the relevant change, you could probably use this tagger for real tasks.

```
# Author: Robert Guthrie

import torch
import torch.autograd as autograd
import torch.nn as nn
import torch.optim as optim

torch.manual_seed(1)
```

Helper functions to make the code more readable.

```
def argmax(vec):
    # return the argmax as a python int
    _, idx = torch.max(vec, 1)
    return idx.item()

def prepare_sequence(seq, to_ix):
    idxs = [to_ix[w] for w in seq]
    return torch.tensor(idxs, dtype=torch.long)

# Compute log sum exp in a numerically stable way for the forward algorithm
def log_sum_exp(vec):
    max_score = vec[0, argmax(vec)]
    max_score_broadcast = max_score.view(1, -1).expand(1, vec.size()[1])
    return max_score + \
        torch.log(torch.sum(torch.exp(vec - max_score_broadcast)))
```

Create model

```
class BiLSTM_CRF(nn.Module):

    def __init__(self, vocab_size, tag_to_ix, embedding_dim, hidden_dim):
        super(BiLSTM_CRF, self).__init__()
        self.embedding_dim = embedding_dim
        self.hidden_dim = hidden_dim
        self.vocab_size = vocab_size
        self.tag_to_ix = tag_to_ix
        self.target_size = len(tag_to_ix)

        self.word_embeds = nn.Embedding(vocab_size, embedding_dim)
        self.lstm = nn.LSTM(embedding_dim, hidden_dim // 2,
                            num_layers=1, bidirectional=True)

        # Maps the output of the lstm into tag space.
        self.hidden2tag = nn.Linear(hidden_dim, self.target_size)

        # Matrix of transition parameters.  Entry i,j is the score of
        # transitioning *to* i *from* j.
        self.transitions = nn.Parameter(
            torch.randn(self.target_size, self.target_size))

        # These two statements enforce the constraint that we never transfer
        # to the start tag and we never transfer from the stop tag
        self.transitions.data[tag_to_ix[START_TAG], :] = -10000
        self.transitions.data[:, tag_to_ix[STOP_TAG]] = -10000

        self.hidden = self.init_hidden()

    def init_hidden(self):
        return (torch.randn(2, 1, self.hidden_dim // 2),
                ققtorch.randn(2, 1, self.hidden_dim // 2))

    def _forward_alg(self, feats):
        # Do the forward algorithm to compute the partition function
        init_alphas = torch.full((1, self.target_size), -10000.)
        # START_TAG has all of the score.
        init_alphas[0][self.tag_to_ix[START_TAG]] = 0.

        # Wrap in a variable so that we will get automatic backprop
        forward_var = init_alphas

        # Iterate through the sentence
        for feat in feats:
            alpha_t = []  # The forward tensors at this timestep
            for next_tag in range(self.target_size):
                # broadcast the emission score: it is the same regardless of
                # the previous tag
                emit_score = feat[next_tag].view(
                    1, -1).expand(1, self.target_size)
                # the ith entry of trans_score is the score of transitioning to
                # next_tag from i
                trans_score = self.transitions[next_tag].view(1, -1)
                # The ith entry of next_tag_var is the value for the
                # edge (i -> next_tag) before we do log-sum-exp
                next_tag_var = forward_var + trans_score + emit_score
                # The forward variable for this tag is log-sum-exp of all the
                # scores.
                alpha_t.append(log_sum_exp(next_tag_var).view(1))
            forward_var = torch.cat(alpha_t).view(1, -1)
        terminal_var = forward_var + self.transitions[self.tag_to_ix[STOP_TAG]]
        alpha = log_sum_exp(terminal_var)
        return alpha

    def _get_lstm_features(self, sentence):
        self.hidden = self.init_hidden()
        embeds = self.word_embeds(sentence).view(len(sentence), 1, -1)
        lstm_out, self.hidden = self.lstm(embeds, self.hidden)
        lstm_out = lstm_out.view(len(sentence), self.hidden_dim)
        lstm_feats = self.hidden2tag(lstm_out)
        return lstm_feats

    def _score_sentence(self, feats, tags):
        # Gives the score of a provided tag sequence
        score = torch.zeros(1)
        tags = torch.cat([torch.tensor([self.tag_to_ix[START_TAG]], dtype=torch.long), tags])
        for i, feat in enumerate(feats):
            score = score +
                self.transitions[tags[i + 1], tags[i]] + feat[tags[i + 1]]
        score = score + self.transitions[self.tag_to_ix[STOP_TAG], tags[-1]]
        return score

    def _viterbi_decode(self, feats):
        backpointers = []

        # Initialize the viterbi variables in log space
        init_vars = torch.full((1, self.target_size), -10000.)
        init_vars[0][self.tag_to_ix[START_TAG]] = 0

        # Forward_var at step i holds the viterbi variables for step i-1
        forward_var = init_vars
        for feat in feats:
            bptrs_t = []  # holds the backpointers for this step
            viterbivars_t = []  # holds the viterbi variables for this step

            for next_tag in range(self.target_size):
                # next_tag_var[i] holds the viterbi variable for tag i at the
                # previous step, plus the score of transitioning
                # from tag i to next_tag.
                # We don't include the emission scores here because the max
                # does not depend on them (we add them in below)
                next_tag_var = forward_var + self.transitions[next_tag]
                best_tag_id = argmax(next_tag_var)
                bptrs_t.append(best_tag_id)
                viterbivars_t.append(next_tag_var[0][best_tag_id].view(1))
            # Now add in the emission scores, and assign forward_var to the set
            # of viterbi variables we just computed
            forward_var = (torch.cat(viterbivars_t) + feat).view(1, -1)
            backpointers.append(bptrs_t)

        # Transition to STOP_TAG
        terminal_var = forward_var + self.transitions[self.tag_to_ix[STOP_TAG]]
        best_tag_id = argmax(terminal_var)
        path_score = terminal_var[0][best_tag_id]

        # Follow the back pointers to decode the best path.
        best_path = [best_tag_id]
        for bptrs_t in reversed(backpointers):
            best_tag_id = bptrs_t[best_tag_id]
            best_path.append(best_tag_id)
        # Pop off the start tag (we dont want to return that to the caller)
        start = best_path.pop()
        assert start == self.tag_to_ix[START_TAG]  # Sanity check
        best_path.reverse()
        return path_score, best_path

    def neg_log_likelihood(self, sentence, tags):
        feats = self._get_lstm_features(sentence)
        forward_score = self._forward_alg(feats)
        gold_score = self._score_sentence(feats, tags)
        return forward_score - gold_score

    def forward(self, sentence):  # don't confuse this with _forward_alg above.
        # Get the emission scores from the BiLSTM
        lstm_feats = self._get_lstm_features(sentence)

        # Find the best path, given the features.
        score, tag_seq = self._viterbi_decode(lstm_feats)
        return score, tag_seq
```

Run training

```
START_TAG = "<START>"
STOP_TAG = "<STOP>"
EMBEDDING_DIM = 5
HIDDEN_DIM = 4

# Make up some training data
training_data = [(
    "the wall street journal reported today that apple corporation made money".split(),
    "B I I I O O O B I O O O".split()
),(
    "georgia tech is a university in georgia".split(),
    "B I O O O O B".split()
)]

word_to_ix = {}
for sentence, tags in training_data:
    for word in sentence:
        if word not in word_to_ix:
            word_to_ix[word] = len(word_to_ix)

tag_to_ix = {"B": 0, "I": 1, "O": 2, START_TAG: 3, STOP_TAG: 4}

model = BiLSTM_CRF(len(word_to_ix), tag_to_ix, EMBEDDING_DIM, HIDDEN_DIM)
optimizer = optim.SGD(model.parameters(), lr=0.01, weight_decay=1e-4)

# Check predictions before training
with torch.no_grad():
    precheck_sent = prepare_sequence(training_data[0][0], word_to_ix)
    precheck_tags = torch.tensor([tag_to_ix[t] for t in training_data[0][1]], dtype=torch.long)
    print(model(precheck_sent))

# Make sure prepare_sequence from earlier in the LSTM section is loaded
for epoch in range(
    300):  # again, normally you would NOT do 300 epochs, it is toy data
    for sentence, tags in training_data:
        # Step 1. Remember that Pytorch accumulates gradients.
        # We need to clear them out before each instance
        model.zero_grad()

        # Step 2. Get our inputs ready for the network, that is,
        # turn them into Tensors of word indices.
        sentence_in = prepare_sequence(sentence, word_to_ix)
        targets = torch.tensor([tag_to_ix[t] for t in tags], dtype=torch.long)

        # Step 3. Run our forward pass.
        loss = model.neg_log_likelihood(sentence_in, targets)

        # Step 4. Compute the losses, gradients, and update the parameters by
        # calling optimizer.step()
        loss.backward()
        optimizer.step()

# Check predictions after training
with torch.no_grad():
    precheck_sent = prepare_sequence(training_data[0][0], word_to_ix)
    print(model(precheck_sent))
# We got it!
```

Out:

```
tensor(2.6907)
tensor([0, 1, 2, 2, 2, 2, 2, 2, 2, 2, 1])
tensor([0.0000]
tensor([0, 1, 1, 1, 1, 2, 2, 0, 1, 2, 2])
```

Exercise: A new loss function for discriminative tagging

It wasn't really necessary for us to create a computation graph when doing decoding, since we do not backpropagate from the viterbi path score. Since we have it anyway, try training the tagger where the loss function is the difference between the Viterbi path score and the score of the gold-standard path. It should be clear that this function is non-negative and 0 when the predicted tag sequence is the correct tag sequence. This is essentially structured perceptron.

This modification should be short, since Viterbi and score_sentence are already implemented. This is an example of the shape of the computation graph *depending on the training instance*. Although I haven't tried implementing this in a static toolkit, I imagine that it is possible but much less straightforward.

Pick up some real data and do a comparison!

Total running time of the script: (0 minutes 6.056 seconds)

Download Python source code: advanced_tutorial.py
Download Jupyter notebook: advanced_tutorial.ipynb
Gallery generated by Sphinx-Gallery

© Copyright 2017, PyTorch.
Built with Sphinx using a theme provided by Read the Docs.