Implementing a linear-chain Conditional Random Field (CRF) in PyTorch

towardsdatascience.com/implementing-a-linear-chain-conditional-random-field-crf-in-pytorch-16b0b9c4b4ea

October 2, 2021

During the last days I've been implementing a CRF model from scratch using PyTorch. My idea by doing this was to understand better how a CRF model works. I've found a lot of online content about CRFs, including blog posts, tutorials and books. I've also watched the series of video lectures from Hugo Larochelle's class on CRF models and found them very intuitive.

After watching them, I decided to implement those math equations from scratch using PyTorch (no worries about gradients whatsoever!). So, this is the purpose of this post! To share with you an easy-to-understand guide on how to implement a (*linear-chain*) CRF model!

Disclaimer: CRFs are a generalization of any undirected graph structure, such as sequences, trees, or graphs. In this post, I'll focus on sequential structures, which means that our model will condition only on previous transitions. This parameterization is known as Linear Chain CRF. For the rest of this post I'll use the acronym CRF to denominate a general CRF and its linear chain counterpart interchangeably.

In this post, I will not describe the reasons for using CRFs nor its applications. I think if you are reading this post is because you already know all that, and you are only interested in the technical part. That said, I'll cover some the basic theory behind it and introduce the notation that will be used for writing the code.

In summary, in this post you will see:

1. The basic theory behind CRFs;
2. How to find the most probable sequence of labels given a sequence of observations;
3. How to compute the score of a sequence of observations given their labels;
4. How to compute the partition function to normalize this score;
5. How to implement all of them in log-space (numerically stable).

And in the next post: how to vectorize the *for-loops* to make calculations easier for parallel computer units (GPUs say 🙌). Take a sneak peek of the vectorized code here.

The basic theory behind the scenes

The original Conditional Random Fields paper was published at the beginning of this century [1]. Since then, the machine learning community has been applying CRFs everywhere, from computational biology and computer vision to natural language

processing. A quick search on google scholar with keywords like “using CRF” and “using Conditional Random Fields” returns ~20k responses.

Over the last few years, CRFs models were combined with LSTMs to get state-of-the-art results. In the NLP community, stacking a CRF layer on top of a BiLSTM was considered a rule of thumb for achieving a higher accuracy on sequence tagging problems. You can see some examples here and here.

In a sequence classification problem, our main goal is to find the probability of a sequence of labels () given a sequence vectors (**X**) as input. This is denoted as the conditional probability *P*(**y**| **X**).

First, let’s define some notation adapted from Larochelle’s class:

- Training set: input and target sequence pairs
- The th input sequence of vectors: ℓ
- The th target sequence of labels: ℓ]
- ℓ is the sequence length.

Assuming time-wise independence, for a sample (,), in a regular classification problem we compute *P*(**y**|) by multiplying the probability of each item at the *k-t*h position:

$$
\begin{aligned}
P(\mathbf{y} \mid \mathbf{X}) &= \prod_{k=1}^{\ell} P(y_k \mid \mathbf{x}_k) \\
&= \prod_{k=1}^{\ell} \frac{\exp(U(\mathbf{x}_k, y_k))}{Z(\mathbf{x}_k)} \\
&= \frac{\exp\left(\sum_{k=1}^{\ell} U(\mathbf{x}_k, y_k)\right)}{\prod_{k=1}^{\ell} Z(\mathbf{x}_k)}
\end{aligned}
$$

Note that we are modelling *P(yk* | **x***k)* using a normalized exponential. This is analogous to the *softmax* transformation widely used in neural networks. These are some intuitions for the reason of using the *exp* function*:*

1. when we multiply very small numbers, we get a smaller number which may suffer .
2. all values are mapped between 0 and +inf.
3. it pushes high values up and low values down. This has a similar effect with an operation. More .

We took two symbols out of the hat: *U and Z*. Let’s see what they are.

$U(x, y)$is known as our **emissions** or **unary scores**. It's simply a score for the label y given our **x**vector at the kth timestep. You can see it as the kth output of a BiLSTM. Actually, in theory, our **x** vector can be anything you want. In practice, our **x** vector is usually the concatenation of surrounding elements, like word embeddings from a sliding window. Each *unary* factor is weighted by a learnable weight in our model. This is easy to understand if we consider them as LSTM outputs.

$Z(x)$is commonly referred as the **partition function**. We can think of it as a normalization factor since we'd like to get probabilities in the end. This is analogous to the denominator of the *softmax* function.

So far, we described a regular classification model with a "final softmax activation" in order to get probabilities. Now we are going to add new learnable weights to model the chance of a label *yk* being followed by *yk+1*. By modelling this, we are creating a dependency between successive labels. Thus, the name *linear-chain CRF!* In order to do so, we multiply our previous probability by *P(yk+1 | yk),* for which we can use exponential properties to rewrite it as unary scores $U(, y)$ plus learnable **transition scores** $T(y, y)$*:*

$$P(\mathbf{y} \mid \mathbf{X}) = \frac{\exp\left(\sum_{k=1}^{\ell} U(\mathbf{x}_k, y_k) + \sum_{k=1}^{\ell-1} T(y_k, y_{k+1})\right)}{Z(\mathbf{X})}$$

In code, $T(y, y)$can be seen as a matrix with shape *(nb_labels, nb_labels),* where each entry is a learnable parameter representing the transition of going from the *i-th* label to the *j-th* label. Let's review all our new variables:

- scores representing how likely is given the input
- scores representing how likely is followed by
- normalization factor in order to get a probability distribution over sequences.

The only thing left to be properly defined is the partition function *Z:*

$$Z(\mathbf{X}) = \sum_{y'_1} \sum_{y'_2} \cdots \sum_{y'_k} \cdots \sum_{y'_\ell} \exp\left(\sum_{k=1}^{\ell} U(\mathbf{x}_k, y'_k) + \sum_{k=1}^{\ell-1} T(y'_k, y'_{k+1})\right)$$

Turns out it's not trivial to compute $Z(\mathrm{X})$ because we have too many nested loops 😖. It's a sum over all possible combinations over the label set at each timestep. To be more precise, we have ℓ computations over the label set. This give us a time complexity of $O(|y|^{\ell})$.

Luckily, we can exploit the recurrent dependencies and use dynamic programming to compute it efficiently! The algorithm that does this is called **forward algorithm** or **backward algorithm —** depending on the order that you iterate over the sequence. Not

to be confused with forward and backward propagation used in neural networks.

And that's all we need to know to start our implementation journey! If you got lost during my explanation, there are very good resources out there explaining CRFs in more detail. Like this excellent tutorial by Edwin Chen, or this extensive tutorial by Sutton and McCallum, or this right-to-the-point notes by Michael Collins.

Code

Let's start our code by creating a class called CRF that inherits from PyTorch's *nn.Module* in order to keep track of our gradients automatically. Also, I added special tokens for the beginning/end of the sentence, and a special flag to inform whether we are passing tensors with batch-first dimension or not.

We could add a special token for a pad id as well. If we do this, we have to make sure to enforce constraints to prevent transitions *from* padding and transitions *to* padding — except if we already are in a pad position. We can see that our transitions scores *T* are denoted as the matrix *self.transitions* and encoded using *torch.parameter*. In this way,PyTorch will learn these weights with autodiff! Great!

Defining the loss function

In a supervised classification problem, our goal is to minimize the expected error during training. We can do this by defining a loss function *L* which takes as input our predictions and our true labels and returns a zero score if they are equal or a positive score if they are different — indicating an error.

Note that we are calculating $P(\mathbf{y}|\ )$ and this is something we want to maximize. In order to frame this as a minimization problem, we take the negative log of this probability. This is also known as the *negative log-likelihood loss (NLL-Loss)*. In our case, we get: $L = -log(P(\mathbf{y}|\ ))$. And applying log-properties, like $log(a/b) = log(a) - log(b)$, we get:

$$
\begin{aligned}
-\log(P(\mathbf{y} \mid \mathbf{X})) &= -\log\left(\frac{\exp\left(\sum_{k=1}^{\ell} U(\mathbf{x}_k, y_k) + \sum_{k=1}^{\ell-1} T(y_k, y_{k+1})\right)}{Z(\mathbf{X})}\right) \\
&= \log\left(Z(\mathbf{X})\right) - \log\left(\exp\left(\sum_{k=1}^{\ell} U(\mathbf{x}_k, y_k) + \sum_{k=1}^{\ell-1} T(y_k, y_{k+1})\right)\right) \\
&= \log\left(Z(\mathbf{X})\right) - \left(\sum_{k=1}^{\ell} U(\mathbf{x}_k, y_k) + \sum_{k=1}^{\ell-1} T(y_k, y_{k+1})\right) \\
&= Z_{\log}(\mathbf{X}) - \left(\sum_{k=1}^{\ell} U(\mathbf{x}_k, y_k) + \sum_{k=1}^{\ell-1} T(y_k, y_{k+1})\right)
\end{aligned}
$$

Where *Z_log* means we take the *log* during the computation of the partition function. This is going to make things easier for us later when we apply the forward-algorithm. So, let's check how the code looks for this part:

Our *forward* pass is simply the *NLL loss* (not to be confused with the forward-algorithm for computing Z(X)), in which we inserted the minus symbol in front of the regular *log_likelihood* method. The *log_likelihood* is computed by first computing the scores and the log partition methods, and lately subtracting each other. Furthermore, we pass a mask matrix to those methods so they can ignore computations associated with pad symbols. For completeness, the mask matrix looks like this:

```
input = [['lorem', 'ipsum 'dolor', 'sit', 'amet'],          ['another', 'sentence',
'here', '<pad>', '<pad>']]mask = [[1, 1, 1, 1, 1],        [1, 1, 1, 0, 0]]
```

Computing the numerator: scores

Since we applied *log* to the *exp* function, the numerator is just the summation of our emission and transition scores at each timestep. In code:

In order to understand this code, you have to think that all operations are the same for each sentence in the batch. So, we start by taking the tags for the first word in each batch by calling `tags[:, 0]`. Similarly, we sum the mask over the timestep dimension to get a list of lengths, which would be `[5, 3]` for the previous example. In practice, it's returned a *torch.LongTensor* with shape `(batch_size,)`. For instance, let's look at the line 28:

```
emissions[:, 0].gather(1, first_tags.unsqueeze(1)).squeeze(1)
```

1. First, we select all batches from the first timestep: `emissions[:, 0]`, which returns a tensor with shape `(batch_size, nb_labels)`.
2. Then we want to select only the values from the columns () that are in the `first_tags`, which has shape `(batch_size,)`. Since `emissions` is a 2D matrix, we the last dimension of `first_tags` to get the shape `(batch_size, 1)`: `first_tags.unsqueeze(1)`
3. Now that they have the same shape, we can use the function to select the values inside `first_tags` in the specified dimension of `emissions`: `emissions[:, 0].gather(1, first_tags.unsqueeze(1))`
4. Finally, this will result in a matrix with shape `(batch_size, 1)`, so we it back to get a 1D

This simple procedure is used throughout the code to select a group of labels within a specified dimension.

The last thing I want to talk about this code is how we ignore scores associated with a padding symbol. And the idea to solve this is very simple: we perform an element-wise multiplication between the two vectors (score and mask vectors), to zero-out the scores in timesteps associated with a pad position.

Computing the partition function: forward algorithm

Now that we have computed our scores, let's focus on our denominator. In order to compute the partition function efficiently, we use the forward algorithm. I will describe it briefly and show how we can compute it in log-space.

The pseudo-code for the forward-algorithm goes as follows:

1) Initialize, for all values of *y'2:*

$$\alpha_1(y'_2) = \sum_{y'_1} \exp\left(U(\mathbf{x}_1, y'_1) + T(y'_1, y'_2)\right)$$

2) For *k=2 to* ℓ-1, for all values of *y'k+1* (log-space)*:*

$$\log(\alpha_k(y'_{k+1})) = \log \sum_{y'_k} \exp\left(U(\mathbf{x}_k, y'_k) + T(y'_k, y_{k+1}) + \log(\alpha_{k-1}(y'_k))\right)$$

3) Finally:

$$Z(\mathbf{X}) = \sum_{y'_\ell} \exp\left(U(\mathbf{x}_\ell, y'_\ell) + \log(\alpha_{k-1}(y'_\ell))\right)$$

Observe that in the 2nd step we take a log of a summation of exps. This can be problematic: if a score for a given label *y'k* is too large, then the exponential will grow very fast to a **very** large number. So, before we take the log in the end we might find an *overflow*. Luckily, there is a trick for making this operation stable:

$$\log \sum_k \exp(z_k) = \max(\mathbf{z}) + \log \sum_k \exp(z_k - \max(\mathbf{z}))$$

The proof that the left side is equal to the right side looks like this:

```
_ = log sum(exp(zk))  = log sum(exp(zk-c) * exp(c))  = log exp(c) + log
sum(exp(zk-c))  = c + log sum(exp(zk-c))
```

Set *c* as *max*(*z*) and we are done. In addition, PyTorch already have this stable implementation available for us in `torch.logsumexp`. Let's now code the algorithm above using PyTorch:

The code above is very similar to the way we have computed the scores in the numerator. In fact, we are computing scores! But now we are accumulating them by looking at previous iterations. I added comments about shapes so you can understand what's going on.

There is just one thing here that we didn't see before (*kind of*):

> `alphas = is_valid * new_alphas + (1 – is_valid) * alphas` : In this line, we are changing the current values of alpha by new ones if we didn't reach a pad position, and keeping the same values otherwise. To see how this works, take a look at this example when we are at the timestep :

```
>>> mask
tensor([[1., 0., 0.],
        [1., 1., 0.],
        [1., 1., 1.]])

>>> alphastensor([[-0.7389, -0.6433, -0.0571, -0.3587, -2.1117],        [ 1.0372,
1.8366, -0.9350, -1.2656, -0.5815],        [ 0.1011,  0.7373,  0.0929, -0.8695,
0.7016]])>>> new_alphastensor([[11.1889, 10.6471, 11.0028, 11.0248, 11.0909],
[10.3975, 11.0104,  8.5674, 10.2359, 13.9150],        [10.1440,  9.9298, 11.3141,
10.1534, 10.3397]])>>> is_valid = mask[:, 1].unsqueeze(-1)>>>
is_validtensor([[0.],        [1.],        [1.]])>>> is_valid * new_alphas + (1 –
is_valid) * alphastensor([[-0.7389, -0.6433, -0.0571, -0.3587, -2.1117],
[10.3975, 11.0104,  8.5674, 10.2359, 13.9150],        [10.1440,  9.9298, 11.3141,
10.1534, 10.3397]])
```

We have updated the 2nd and 3rd sequences but not the 1st because at timestep *i=1* we reached a pad position.

As a good observation, you can see that once we take the *logsumexp* we are already in log-space! So, we can just add the values of alpha to our scores. And finally, we take one more *logsumexp* operation at the final timestep to return the final values of reaching the end of the sentence — so we are still in log-space.

The time complexity of this algorithm is $O(\mathcal{N}|y|^2)$, which is much lower than the exponential bound we had with the naive approach.

Finding the *best* sequence of labels

Now that we computed the partition function, we have almost everything done. If we compute the backward algorithm as well — which is just traversing the sequence backwards — we could find the label that maximizes $P(\mathbf{y}k \mid \mathbf{X})$ at each timestep k. Interestingly, thiswould be the optimal solution if we assume the CRF is the true distribution. And it can be formulated like this:

$$P(y_k \mid \mathbf{X}) = \frac{\exp\left(U(\mathbf{x}_k, y_k) + log(\alpha_{k-1}(y_k)) + log(\beta_{k+1}(y_k))\right)}{\sum_{y'_k} \exp\left(U(\mathbf{x}_k, y'_k) + log(\alpha_{k-1}(y'_k)) + log(\beta_{k+1}(y'_k))\right)}$$

Where the α scores come from the forward algorithm and the β scores come from the backward algorithm. In order to find the best sequence of label $\mathbf{y}^*$ we could take the *argmax* at each timestep:

$$\mathbf{y}^\star = \underset{y_1, y_2, \ldots, y_\ell}{\arg\max}\, P(y_k \mid \mathbf{X})$$

Viterbi algorithm

But, turns out we don't need to compute the backward algorithm in order to find the most probable sequence of labels. Instead we can simply keep track of the maximum scores at each timestep during the forward algorithm. And once we are done, we can follow the backward trace of the max operations (*argmax*) in order to **decode** the sequence that maximizes the scores. This is exactly what the code below does:

This algorithm is known as Viterbi algorithm. It is almost the same as the forward-algorithm we have used in the log_partition function, but instead of having regular scores for the whole sequence, we have maximum scores and the tags which maximize these scores. In other words, we have replaced the *torch.logsumexp* operation by *torch.max*, which returns the *max* and the *argmax* together.

So, everything we need to do now is to pick these final tags and follow the backward trace to find the whole sequence of "*argmax*" tags. The continuation of the above code goes as follows:

See that for each sample we are iterating over the backtrace of that sample, and in each timestep we are inserting the tag which maximizes the scores at the beginning of `best_path`. So, at the end, we have a list where the first element corresponds to the first tag and the last element corresponds to the last tag valid tag (*see line 15*) of the sequence. Note also that .insert(0,) in python is O(n), so a faster approach would be to use .append() and then reverse the list later.

That is it! When we compute the *find_best_path* operation for all samples in the batch we are done!

Putting all together

The full code is available here: https://github.com/mtreviso/linear-chain-crf. Take a look at `main.py` and `bilstm_crf.py` to see the CRF in practice!

Conclusion

Well, this post got longer then I intended 😛. Feel free to make this code more efficient and leave a comment to show us how you managed to do it 🙂.

Finally, I think it is worth to mention that If you'd like to use a CRF model in production, I strongly suggest you to use a well-tested and efficient implementation, like this great pytorch package, or the one provided by the allennlp library.

In the next post we'll see how we can vectorize our *for-loops* to calculate the partition function and the Viterbi algorithm.

More info

CRFs are only one among many sequential models available. I highly recommend you to take a look at this Noah Smith presentation on sequential models or at this lecture by André Martins to see some visual examples of the algorithms presented in this post. Or even better, you can attend the next Lisbon Machine Learning Summer School, which will cover sequential models and many more interesting topics about Machine Learning and Natural Language Processing!

References

- Hugo Larochelle's lectures on CRFs:
- Pytorch tutorial — Advanced: Making Dynamic Decisions and the Bi-LSTM CRF:
- Michael Collins's notes on Log-Linear Models, MEMMs, and CRFs:
- Michael Collins's notes on the Forward-Backward Algorithm:
- Tutorial by Sutton and McCallum — An Introduction to Conditional Random Fields:
- Edwin Chen blog post:
- An Overview of Conditional Random Fields by Ravish Chawla:

🔍 *Special thanks to @flassantos31, @erickrfonseca and @thales.bertaglia for reviewing this post.*

🔮 Post-review thanks: