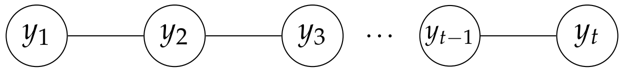
CS281: Advanced ML **October 16, 2017**

Lecture 12: Recurrent Neural Networks

Lecturer: Sasha Rush *Scribes: Alexander Goldberg, Daniel Eaton, George Han, Moritz Graule, Kristo Ment*

12.1 Introduction

Recall the basic structure of a time series model:



In previous lectures we discussed interpreting such a model as a UGM, with log-potentials given by:

$$\theta(y_t) + \theta(y_{t-1}, y_t)$$

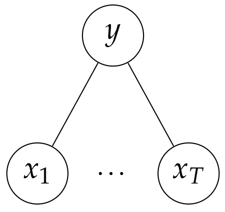
What did we gain from this abstraction? Because all of these models have the same comditional independence structure, we are able to run the same algorithm for inference on all of these parameterizations (sum-product). Thus, conditioned on observed data, we may find the exact marginals: $p(y_s = v)$

Perhaps these structures are not necessary? Is exact inference required in cases where there is alot of data?

12.2 RNNs

12.2.1 What is an RNN?

Recall our discussion of using neural networks for classification. The UGM describing this setting is as follows:



We could then choose to parameterize $p(y|x_{1:T})$ as a neural network:

$$p(y|x_{1:T}) = Softmax(\mathbf{w}^T \phi(x_{1:T}; \theta))$$

where ϕ was just some linear combination of $x_{1:T}$ passed through a link function. If we wish to apply this scheme to cases in which $x_{1:T}$ is a *sequence* we might think to use a ϕ of the following form:

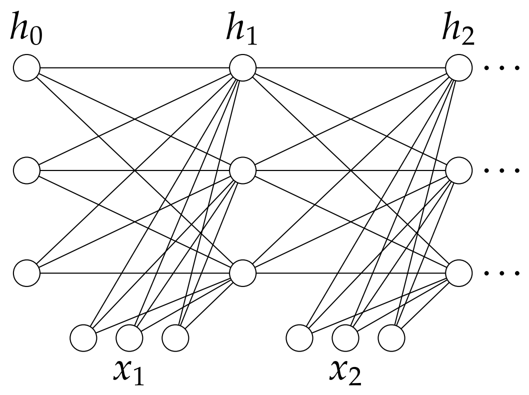
$$\phi(x_{1:T}; \theta) = tanh(\mathbf{wx}) = tanh(\sum_{t=1}^{T} w^{(t)} x_t)$$

However, the problem with this type of approach is that it is "time invarient," that is the same weights are shared by all of the x_t. To see why this is problematic, consider using a bag of words representation for the X_t and encountering the two sentences: "The man ate the hot dog." and "The hot dog ate the man." While these two sentences are saying completely different things, they result in the same value generated by ϕ.

Recurrent neural networks get around this problem by implementing the following choice of ϕ:

$$\phi(x_{1:T};\theta) = tanh(\mathbf{w}^{(1)}x_t + \mathbf{w}^{(2)}\phi(x_{1:t-1};\theta) + b)$$

where $\mathbf{w}^{(1)}x_t$ incorperates the current positional input, $\mathbf{w}^{(2)}\phi(x_{1:t-1};\theta)$ carries information from the previous inputs, b is the bias and $tanh$ is the chosen nonlinear transformation.
Representing this RNN as a computational graph:



where we call h_t the RNN hidden state. As you can see, each h_t is a nonlinear function of $x_{1:t}$.

12.2.2 RNN Training

To understand how backpropagation works in RNNs, consider the functions that the NN is composed of:

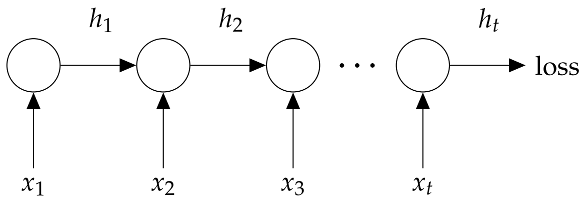
$$h_1 = tanh(\mathbf{w}^{(1)}\mathbf{x}_1 + \mathbf{w}^{(2)}\mathbf{h}_0 + b)$$

$$h_2 = tanh(\mathbf{w}^{(1)}\mathbf{x}_2 + \mathbf{w}^{(2)}\mathbf{h}_1 + b)$$

$$\dots$$

$$h_t = tanh(\mathbf{w}^{(1)}\mathbf{x}_t + \mathbf{w}^{(2)}\mathbf{h}_{t-1} + b)$$

Notice that the only parameters to optimize in these expressions are $\mathbf{w}^{(1)}$ and $\mathbf{w}^{(2)}$, which are shared among all of the equations. In our normal representation of backpropagation we have:



Thus the size of the computational graph and the amount of backpropagation necessary will scale with the length of the input, T. Now, thinking of this situation like a simple feed-forward NN, how many layers does this network have?

12.2.3 Issues: Network Layers

This network will have T layers, where T is the length of the input, which may be very long. Thus, when performing back-propagation it is very likely that there will be problems of gradient instability – very high (exploding) or low (vanishing) values of the gradient somewhere in the back propagation, making it hard to learn the parameters for low layers.

Consider using tanh as the activation function at each layer. Then, the gradient is close to 0 for very large or very negative values, which is quite likely to happen somewhere in a network with many layers, so multiplying these small gradients together in back-propagation will make the contributions of the beginning of the sequence to the loss very small. Thus, it could take prohibitively long to learn weights for the beginning of the sequence. This problem is known as the problem of *vanishing gradients*.

12.2.4 Main idea/trick/hack for vanishing gradients

In order to deal with vanishing gradients, we want to try to pass on more info from low layers while taking gradients, so we add connections variously called:

- residual connections
- gated connections
- highway connections
- adaptive connections

The idea of residual connections is that we let

$$\mathbf{h_t} = \mathbf{h_{t-1}} + \tanh\left(\mathbf{w}^{(1)}\mathbf{x_t} + \mathbf{w}^{(2)}\mathbf{h_{t-1}} + b\right)$$

so that taking the gradient at layer t we get more information passed on from the the linear term $\mathbf{h}_{t-1}$ outside of the tanh.
In fact, we can adaptively learn how much the gradient at each time step should be taken from the previous time step directly from the cata. Thus, we weight the contributions of the $\mathbf{h_t}$ and $\tanh\left(\mathbf{w}^{(2)}\mathbf{h_{t-1}} + ...\right)$ by a factor λ that is also learned from the data:

$$\mathbf{h_t} = \lambda \odot \mathbf{h}_{t-1} + (1-\lambda) \odot \tanh\left(\mathbf{w}^{(1)}\mathbf{x_t} + \mathbf{w}^{(2)}\mathbf{h}_{t-1} + b\right)$$

$$\lambda = \sigma\left(\mathbf{w}^{(4)}h_{t-1} + \mathbf{w}^{(3)}\mathbf{x}_t + b\right)$$

By passing on information directly from previous timesteps, we can prevent vanishing gradients, since the linear terms pass on more information from previous timesteps. In this sense, the λs function as "memory" of the previous timesteps. Important RNN variants using this idea are:

- LSTSM (Long short-term memory networks)
- GRU
- ResNet

12.3 Using RNNs

12.3.1 Classification

Our classification algorithm has three stages:

1. Run LSTM
2. Compute Softmax
3. Find maximizing class

This means that in order to make a prediction, we only need to compute

$$p(y_i|x_{1:T}) = \text{Softmax}(\mathbf{w}^{(2)}\mathbf{h}_i)$$

and multiply across each y_i to obtain

$$p(y_{1:T}|x_{1:T}) = \prod_{i=1}^{T} p(y_i|x_{1:T}) = \prod_{i=1}^{T} \text{Softmax}(\mathbf{w}^{(2)}\mathbf{h}_i)$$

Critically, this means that we do not attempt to model the relationship between the y_i at all, and thus we do not need to assume any distribution over y!

Lets compare this to our alternative approach, which requires full generation. Imagine that any y_i is conditional on all of $y_{1:i-1}$. Then our DGM (for five nodes) is a fully connected K_5:

$$\begin{tikzcd}
\mathbf{y}_0 \arrow[r] \arrow[rr, bend left] \arrow[rrr, bend left=40] \arrow[rrrr, bend left=50] & \mathbf{y}_1 \arrow[r] \arrow[rr, bend left] \arrow[rrr, bend left=40] & \mathbf{y}_2 \arrow[r] \arrow[rr, bend left] & \mathbf{y}_3 \arrow[r] & \mathbf{y}_4
\end{tikzcd}$$

How can we then compute $p(y_s = v)$? A naive approach would be to literally enumerate all possible sequences and sum across all possibilities. However, this is very computationally expensive (exponential in T). Instead, we can speed up this approach by employing a greedy search. Let

$$\hat{y}_1 = \text{argmax}_v \, p(y_1 = v)$$

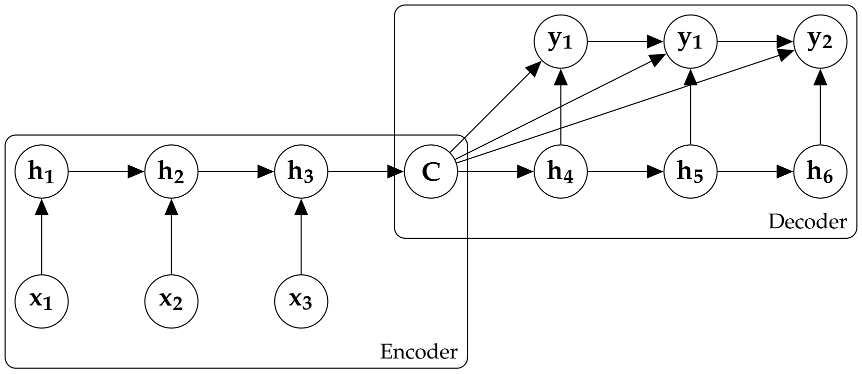
Now for each subsequent point y_i, we can compute

$$\hat{y}_i = \text{argmax}_v \, p(y_1 = v|\hat{y}_{i-1})$$

which is now linear in T, instead of exponential in T.

12.3.2 Applications

RNNs are commonly used for machine language translation and speech recognition. A simple machine translation model is that of the Encoder-Decoder model.



In this model, a sentence from the first language is fed in word by word (each x_i) into the Encoder. This then runs through a normal RNN setup before being fed into C, which stores the final result of the encoder.

Now in the decoder, another RNN is run in reverse that spits out words in the second, translated, language. Each translated word y_i depends on the current layer in the decoder RNN, h_i, C and the last translated word (to prevent the same word from being generated multiple times).

Remark. We will talk about *Information Theory* in the next lecture.

12.4 Practical Exercise: Speech Generator

RNNs have been used successfully as speech generators, taking in a sequence of letters (or words) and predicting subsequent letters (words). Implement and train a character-level RNN with PyTorch, and use it to sample a sentence.

12.4.1 Solution

The architecture of the RNN can be conveniently implemented with torch.nn. As our training data, we downloaded all of Donald Trump's 2016 campaign speeches from http://www.presidency.ucsb.edu/2016_election.php. The concatenated data is available in `concat_speeches.txt` (size: 1.3 MB). To facilitate the training process, we only included lowercase letters and a few additional symbols ` .:?[]` from the raw data, for a total of `n_letters=32` characters. We then assembled a training set by randomly drawing 10,000 30-character sequences from the data, and used each of these to predict the next (31st) characters. We also generated a validation set by similarly drawing 500 additional character sequences. All of the input data must be converted to one-hot vectors: the entire training set was stored as an 30x10000x32 array, as required by `torch.nn.LSTM` below.
We then proceeded to set up an RNN with long-short term memory (LSTM), 2 layers, and `n_hidden=200` as follows:

```
model = torch.nn.Sequential()
model.add_module("lstm", torch.nn.LSTM(n_letters, n_hidden, 2))
model.add_module("encoder", torch.nn.Linear(n_hidden, n_letters))
```

The linear "encoder" layer is necessary to convert the output of the final LSTM layer to 32 probabilities, each for a single character. A `dropout` parameter can be added to the LSTM call in order to prevent overfitting (this will randomly overwrite some input nodes with zeros). We then implement the loss function via a cross-entropy layer which includes a `LogSoftmax()` function:

```
loss = torch.nn.CrossEntropyLoss()
```

Finally, we choose Adam from `torch.optim` as our optimizer:

```
optimizer = torch.optim.Adam(model.parameters(), lr=0.005)
```

The model is then trained over a significant number of epochs: it took us a few hours to get through 300 epochs on a single CPU. This performance can likely be improved by utilizing the `torch.cuda` library to run the training on a GPU instead. The training function for a single epoch can be implemented as follows:

```
# Train the RNN and return the loss
def train(model, optimizer, data, target):

    optimizer.zero_grad()
    output, hc = model.lstm(data)
    new_loss = loss(model.encoder(output[-1]), target)
    new_loss.backward()
    optimizer.step()
    return new_loss.data[0]
```

Note that we are only using the last step of the output to compare against the target: we are interested in predicting the 31st character after seeing the entire 30-character sequence. We can monitor the training and validation losses to check for convergence and overfitting.
Finally, we can feed the trained model any input sequence and let it predict the next (say, 140) characters. For example, after 175 epochs of training, simply feeding the network "Trump:" yields the following sentence:

```
Trump: i an presisent of the sime and i wat the worlica tore and the spaising our
    componitien and i wat of the spared the worled the worle and the spaiting our
    communitien
```

This already begins to sound like real speech, considering the fact that the RNN had to learn the language from scratch! In any case, humans take years to learn to speak whereas we have only been training for a few hours. At 300 epochs, we obtained samples like this:

```
Trump:ey southon fac cort the whale getton. buoller campouse tilathad sumplee ours
    dousting to beokew athing the semerica ince forette. [applause]
```

However, both training and validation losses were still, at that point, declining rapidly. Thus, further training will likely result in an improved performance.
This solution is merely an outline to a plausible more successful algorithm. In particular, we can expand the size of the model, the number of training sequences, or the sequence length, or tweak other relevant model parameters to train a smarter speech generator. The Python code used for this write-up is included as `trump_generator.py`.