Recurrent Neural Networks

11-785 / Spring 2019/ Recitation 7

Daanish and David

Slides by Raphael Olivier

Recap : RNNs are Magic

- They have infinite memory
- They handle all kinds of series
- They have resulted in significant advancements in NLP: Machine Translation, Speech Recognition, Speech Synthesis, etc.
- Have you noticed how Google Translate suddenly became good in November 2016 ?

Recap : RNNs are hard to train

They suffer from :

- Saturation
- Vanishing/exploding gradients
- Complex loss surfaces with tons of bad local minima
- They don’t usually like dropout
- ...

LSTMs/GRUs address some of these issues, but they’re not perfect.

When you use RNNs, you will spend most of your time tuning hyper-parameters (or looking for hacks in papers).

News : RNNs are hard to implement

Or at least a bit harder than other networks.

That's what this recitation is for.

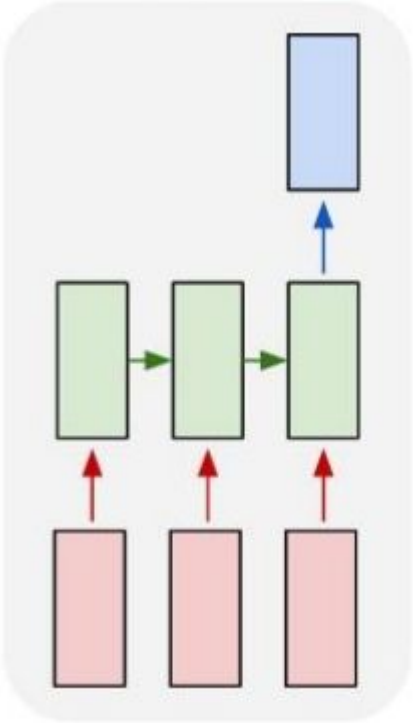
Today we'll create a language model, a simple example of task that uses RNNs.

Plan for today

- Language models
- RNNs in Pytorch
- Train an RNN
- Generation with an RNN
- Variable length inputs

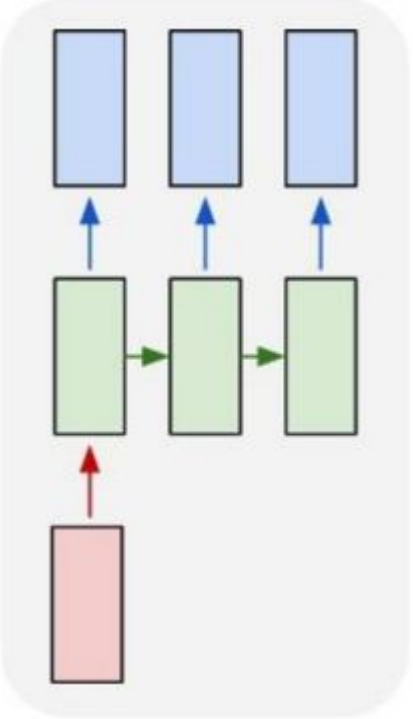
Sequence-based tasks

Many to one



Ex : text classification

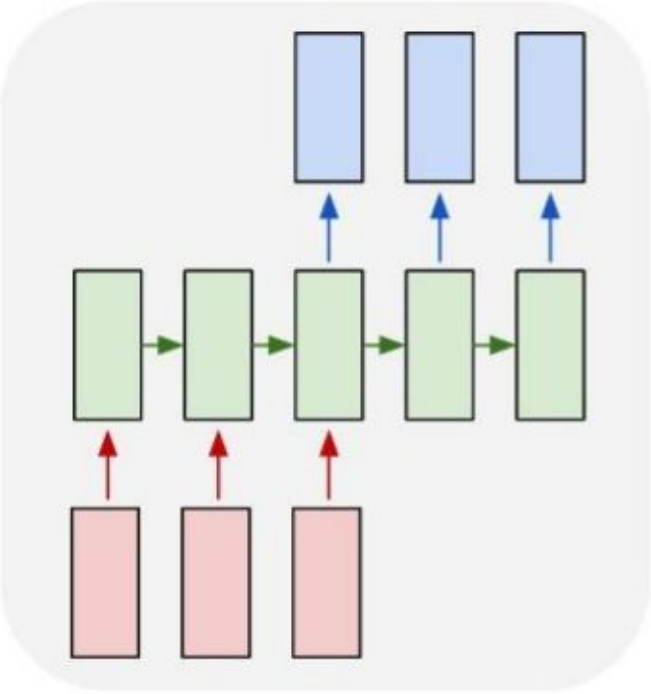
One to many



Ex : sentence generation

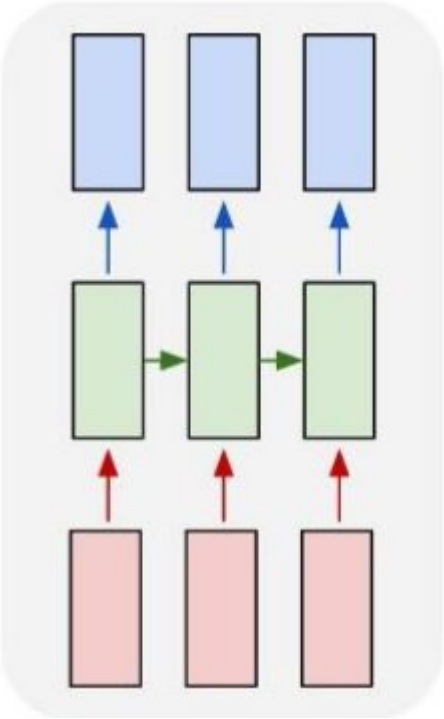
Sequence-based tasks

Many to many



Ex : machine translation

Many to many



Ex : POS tagging

Language models

Goal : predict the “probability of a sentence” P(E)
i.e. how likely it is to be an actual sentence.

Useful as a sub-task in many contexts. Ex : fluency assessment in machine translation

Building a language model is an unsupervised task…. and also a many-to-many one.

Language models

$$P(E) = P(e_1, e_2, ..., e_M)$$
$$= \prod_{m=1}^{M} P(e_m | e_1, ..., e_{m-1})$$

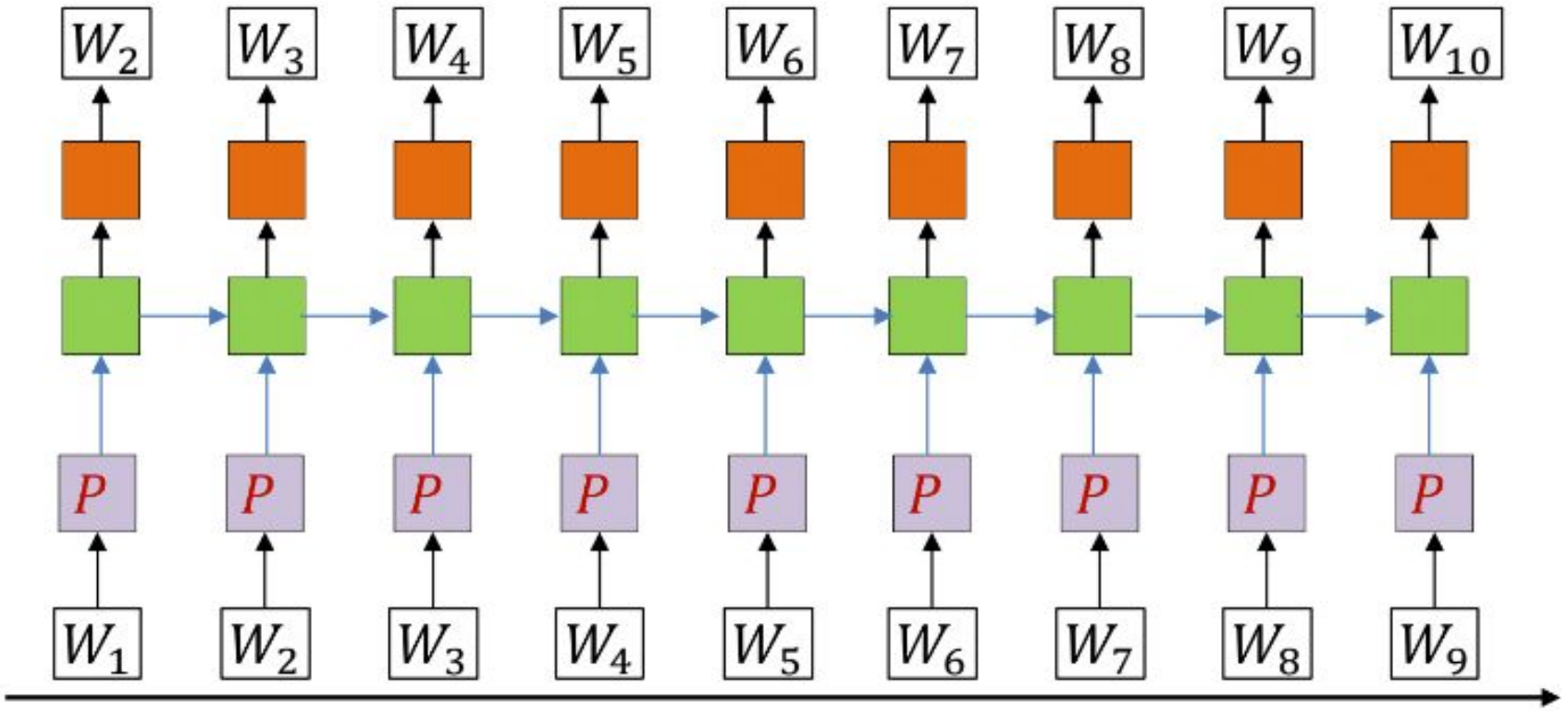
We now have a sequence to predict :

$(e_1, e_2, ..., e_{M-1})$ are the inputs

$(e_2, e_3, ..., e_M)$ are the outputs

And we can train with cross-entropy !

Recurrent language models



Without infinite memory ?

Non-recurrent architectures make the n-gram assumption to be able to use finite memory :

Ex n=3 : $P(e_m|e_{m-1},\ldots,e_1) = P(e_m|e_{m-1},e_{m-2})$

$\rightarrow$ Non-neural count-based models, MLP with finite window, etc.

Recurrent networks have infinite memory so they don't need that assumption.

RNN modules in pytorch

```
rnn = nn.RNN(input_size = 32,
             hidden_size = 64,
             num_layers = 1,
             batch_first = False,
             dropout = 0,
             bidirectional = False)
```

- num_layers is the number of **stacked** (vertical) layers
- dropout is the dropout **between stacked layers**

The .forward() method takes an input of size
seq_length x *batch_size* x *input_size*
and an optional initial hidden state (defaults to 0) of size *num_layers* x *batch_size* x *hidden_size*. It returns an output of same size and the final hidden state.

RNN modules in pytorch

Important : the outputs are exactly the hidden states of the final layer. Hence if the model returns y,h :
y[-1] == h[-1]

Similar classes for LSTM and GRU, **but** for LSTM, h is a tuple (hidden_state, memory_cell)

RNN modules in pytorch

Sometimes you want to have more control between the stacked layers and the time steps (for example to access the intermediate hidden states).

For that you have the RNNCell module.

```
rncell = nn.RNNCell(input_size = 32,
                    hidden_size = 64)
```

It takes an input of size *batch_size* x *input_size* and a hidden state, and returns the next hidden state.

Other layers in a LM

After the RNN module, you stack a linear layer of size

hidden_size x *vocabulary_size*

Before it, you need a *word projection* aka an embedding.

```
embed = nn.Embedding(num_embeddings = VOCAB_SIZE,
                     embedding_dim = 32)
```

Takes a LongTensor of arbitrary shape.

Training a LM

Now you need batches to feed your model. Initially, you only have one big text.

The simplest way :

- Fix a sequence length L
- Concatenate all your words into one big (long) tensor of size N
- Divide it into N // L tensors of size L
- These are your elements.

Even if you train on a fixed size, the network should learn to generate text of arbitrary length.

Evaluate your model

To evaluate how good your model is, you usually feed it with actual text from the (validation) set and look at :

- The loss per word : *l = loss/n_words*
- The **perplexity :** p = exp(l)

It quantifies how well your model predicts that sentence.

A perplexity of 100 (loosely) means that your model performed as if it had to choose uniformly and independently among 100 possibilities for each word

Let’s try all that out !

Prediction

Language models are usually used in other downstream tasks. But you can use them to generate some text given a beginning.

As in MLPs for classification, you want to predict the most likely element from your output distribution

Generate one word is straightforward : you feed your text, get the last output (probability distribution on the vocabulary) and predict its argmax.

Generation

To generate N words, you have N*vocabulary_size possible sequences. Recall that

$$P(E) = P(e_1, e_2, ..., e_M)$$
$$= \prod_{m=1}^{M} P(e_m | e_1, ..., e_{m-1})$$

To know each sentence's probability you'd need to feed all (N-1)-length beginnings → (N-1) * vocabulary_size forward passes ! Unfeasible.

→ Need another way to get the most likely sequence, or at least a very likely one.

Greedy search

Idea : if at each step you take the most likely word, the overall sentence should be likely too.
It's called **greedy search** :

- At step t, select the most likely word from your distribution over the vocabulary
- Use it as input at step t+1

It's the most simple inference method (and also not the best).

Let's try it out !

Random search

- At step t, sample a word *from* your distribution over the vocabulary
- Use it as input at step t+1

Not obviously better than greedy search **but** you can apply it several times, get different results, and take the likeliest one.

Beam search

Same as greedy search **but** you keep the *n best words* at each step rather than the one best. n is the *beam size*.

You store and update a list of hypothesis.

When increasing n, you get rather close from the most likely sentence.

Most research papers' favorite, but a bit trickier to implement.

And that’s all with LMs

Are we done ?

And that’s all with RNNs

Are we done ?

.

.

.

.

.

.

… Not exactly.

And that's all with RNNs

Are we done ?

.

.

.

.

.

.

… Not exactly.

Let's go back a few slides.

Training a LM

Now you need batches to feed your model. Initially, you only have one big text.

The simplest way :

- Fix a sequence length L
- Concatenate all your words into one big (long) tensor of size N
- Divide it into N // L tensors of size L
- These are your elements.

Even if you train on a fixed size, the network should learn to generate text of arbitrary length.

Training a LM

Now you need batches to feed your model. Initially, **you only have one big text.**

The simplest way :

- Fix a sequence length L
- Concatenate all your words into one big (long) tensor of size N
- Divide it into N // L tensors of size L
- These are your elements.

Even if you train on a fixed size, the network should learn to generate text of arbitrary length.

Limits of fixed-length inputs

For language models trained on one large set, creating batches is rather easy.

For language models trained on several large texts (ex: WSJ articles), you can get fixed-length sequences from separate texts too.

But for supervised NLP tasks, that's never how it works.

Limits of fixed-length inputs

In Machine Translation, Speech recognition,etc. you have pairs of sequences.

ex : I like apples → J'aime les pommes

You need to keep these sequences as is to learn something.

We're not dealing with any of these specific applications today but to learn RNNs you need to learn how to deal with **variable length inputs**.

You will need to do this in the upcoming HWs. Pay attention.

Variable-length inputs

Your dataset is now a list of N sequences of different lengths.

A tensor has fixed dimensions.

How do you feed that in batches to your RNN ?

Variable-length inputs

Idea #1
Use batches of size 1.

Advantages : the simplest, and you can still do minibatch optimization by accumulating the gradients over several examples

Problems : It's really, *really,* ***REALLY*** slow.

Conclusion : you may start with that for your prototype, but do better when you begin actual training.

Variable-length inputs

Idea #2
Look for sequences of same size and do batches with them

Advantages : Normally fast, not too complex

Problems : Usually the data won't permit it : in many applications pairs of sequences have very different lengths, and finding many several pairs with same length elements is unreasonable.

Conclusion : if you can, consider doing it.
*Hint: This is probably **not** going to work for the HW datasets*

Variable-length inputs

Idea #3
Look for sequences of close sizes and pad to the same size

Advantages : Easier to find sequences to batch, padding is easy

Problems : Usually no “natural” way to pad, so your loss will be noisy

Conclusion : don’t do it

Variable-length inputs

Idea #4
Pad and remove noisy elements before computing the loss (ex: with a mask)

What does that mean?
Compute the loss element-wise, and zero out all the loss terms that correspond to the padded input.

This can be done with a simple 1-0 mask

Reduce the ‘masked’ loss to a scalar term (average) and backpropagate

Variable-length inputs

Idea #4
Pad and remove noisy elements before computing the loss (ex: with a mask)

Advantages : Simultaneously quite fast, doable, and Pytorch provides a *pad_sequence* method to help you.

Problems : You still lose time applying the RNN on zeros. With just one big sequence you may get an unexpected CUDA memory error. Implementation prone to bugs, hard to track and debug.

Conclusion : one of the recommended methods, but be careful.

Variable-length inputs

Idea #5
Build something up with several tensors for different time steps

Advantages : Faster than all of the previous ones

Problems : Implementing this is going to be difficult, time-consuming, and a bug-filled nightmare.
God-like engineering skills are needed for a fast **and** correct implementation.

Conclusion : just don't..

Variable-length inputs

Idea #5b
Use Pytorch's **packed sequences** (it's idea #5 but some gods at facebook did most of the godly work for you)

Advantages : Doable and the fastest. RNN modules are optimized for it. Pytorch provides methods to help you.

Problems : At some point you pad, so unexpected CUDA errors can still happen. Harder to implement and debug than #4 (pad and mask) because of sorting. Network has to be changed.

Conclusion : recommended, but be extra careful.

Pad and pack sequences

```
import torch.nn.utils.rnn as rnn
x1 = torch.rand(1,2)
x2 = torch.rand(4,2)
x3 = torch.rand(3,2)
padded = rnn.pad_sequence([x1,x2,x3], batch_first=False)
padded
```

```
tensor([[[0.3097, 0.1797],
         [0.6480, 0.8418],
         [0.6333, 0.9508]],

        [[0.0000, 0.0000],
         [0.7329, 0.6177],
         [0.4506, 0.2960]],

        [[0.0000, 0.0000],
         [0.4260, 0.0718],
         [0.4125, 0.8680]],

        [[0.0000, 0.0000],
         [0.3733, 0.6269],
         [0.0000, 0.0000]]])
```

Pad and pack sequences

```
x1 = torch.rand(1,2)
x2 = torch.rand(4,2)
x3 = torch.rand(3,2)
packed = rnn.pack_sequence([x1,x2,x3])
type(packed)
```

```
ValueError: 'lengths' array has to be sorted in decreasing order
```

```
x1 = torch.rand(1,2)
x2 = torch.rand(4,2)
x3 = torch.rand(3,2)
packed = rnn.pack_sequence([x2,x3,x1])
type(packed)
```

```
torch.nn.utils.rnn.PackedSequence
```

Pad and pack sequences

You can go from padded to packed and packed to padded, but need to track the lengths

```
padded2 = rnn.pad_sequence([x2,x3,x1])
lens = [len(x) for x in [x2,x3,x1]]
packed2 = rnn.pack_padded_sequence(padded2,lens)
print(type(packed2))
padded3,lens2 = rnn.pad_packed_sequence(packed2)
print(padded3.equal(padded2))
```

```
<class 'torch.nn.utils.rnn.PackedSequence'>
True
```

Packed sequences and RNNs

A PackedSequence can be fed into RNN modules, but nothing else.

```
model = nn.RNN(input_size=2,hidden_size=3)
packed_output, hidden_state = model(packed)
padded_output, lens = rnn.pad_packed_sequence(packed_output)
output_list = [padded_output[:lens[i],i] for i in range(3)]
output_list
```

```
[tensor([[-0.3548,  0.5536, -0.2994],
         [-0.3248,  0.5481, -0.1642],
         [-0.4053,  0.6779, -0.1642],
         [-0.3501,  0.6754, -0.2385]], grad_fn=<SelectBackward>),
 tensor([[-0.3761,  0.5722, -0.4507],
         [-0.5109,  0.6319, -0.3484],
         [-0.4333,  0.6192, -0.1229]], grad_fn=<SelectBackward>),
 tensor([[-0.3209,  0.5255, -0.4195]], grad_fn=<SelectBackward>)]
```

Packed sequences are on the same device as the padded sequence.

Let’s try this out in a language model !

Next

Sequence-to-sequence with RNNs :

CTC (HW3) (Rec. 8)
Encoder-decoder and attention (HW4) (Rec. 9)

Check out this paper for Regularization and Optimization tricks:
https://arxiv.org/pdf/1708.02182.pdf

Questions ?