Journal Pytorch

Aritz Bercher

September 8, 2018

Abstract

In this journal, I will try to keep track of my progresses in the mastery of the pytorch library. I will focus on NLP tasks.

1 Useful Resources

- Here there seems to be a **mini-course about NLP with Pytorch**. I screened through it and it seems to end up with Attention:
  `http://dl4nlp.info/en/latest/`
  Edit (30.04.18): It seems that there's been a new version of the course on the 29.04.18 and 30.04.18. The scripts are available on the website (and maybe also the data). The projects seem quite serious.
- I found this page which seems to gather **a lot of Pytorch tutorials**:
  pytorch.org: tutorials
- This pages lists and explains **all the possible operations (implemented with methods) on pytorch tensors**:
  `http://pytorch.org/docs/stable/tensors.html`

2 Journal

20.03.18 ~ Starting tutorials on Pytorch NLP
I finished reading this page about pytorch:
pytorch.org: nlp DL tutorial
It shows a basic NLP model where we use a one-layer neural network. The only part which wasn't clear to me was

```
loss.backward()
optimizer.step()
```

It is not so clear how the optimizer knows that it is this loss function which has to be minimized.

23.03.18 ~ Installing pytorch in my mypy36 conda environment
Following the instructions given here I installed pytorch in my personal environment (mypy36) with the following command:

```
conda install pytorch torchvision -c pytorch
```

In this tutorial:
pytorch.org: word embeddings tutorial
the **continuous bag of word** model is used which differs from the skip-gram model (presented in this page that I had already read) as explained in this page from Quora:
Quora: What are the continous bag of words and skip-gram architectures?

24.03.18 ~ First exercise with Pytorch
I did my first exercise with Pytorch, the one presented there:
pytorch.org: word embedding tutorial
It was a very simple model and I copied (but not copy-pasted!) most of it from the previous example. But I feel like I'm getting familiar with the syntax.

25.03.18 ~ LSTM for part-of-speech tagging
I read the part-of-speech tagging example of the following page:
pytorch.org: sequence models tutorial

26.03.18 ~ part-of-speech tagging with character-level features
I tried to do the exercise on this page:
pytorch.org: sequence models tutorial
but I couldn't do it. I understood what the idea was but there is a problem with the implementation, one cannot give any kind of input to a model. It has to be tensor. But if one gives for a word like "the", the index associated to this word (by `word_to_ix`) as well as a vector containing the indices associated to each one of the letters containing this word, the size of the input will vary. I found this solution online:
`https://github.com/AdenosHermes/pytorch-NLPTutorial`
which consists in first using a model to create the embedding for the characters, and then use it as input (together with the index of the word) for a second model. The first model does the backward propagation (adjustment of the weights) after each word, whereas the second after each sentence (like the original one).

27.03.18 ~ Using zero padding for inputs with varying length
Nikos explained me that one could use zero padding when the length of the inputs is varying. There should even be some already implemented lstm model which take this in account.

It seems that there are two functions which help going from an input with zero padding (the kind we have to provide to the model) to a transformed input without zero padding (better in order to use an lstm). I found them here:
`http://pytorch.org/docs/master/nn.html#torch.nn.utils.rnn.pack_padded_sequence`
They are called
`torch.nn.utils.rnn.pack_padded_sequence`
and
`torch.nn.utils.rnn.pad_packed_sequence`.
But maybe I could implement my own since it doesn't look to be very difficult. I only fear that it may break the differentiation graph structure.
I was able to do a little toy example of transformation of the input using zero padding. I should implement a function which does this transformation to transform the input before passing it to the model and then using the functions mentioned above inside the model.

28.03.18 ~ Trying to use padding
I built a little function to do padding but I'm facing new difficulties now. First, some entries of my dictionary lead to 0. I fear that their will be a conflict when I will try to unpad the entries of the tensor to feed them into an LSTM. Then before using the LSTM I have to do the embedding meaning each index is going to be change into a vector. I guess that I should do the unpadding first. I'm also wondering what is allowed to use as an input. If I add a digit to every row, giving the actual length of the word and then use it as an index, am I not going to run into trouble for the optimization par with the gradient descent?

30.03.18 ~ Zero padding solved but subsetting unclear
I managed to do the zero padding (as part of the preprocessing of the input) with a home-made function. I was stuck for a while because I wanted to put all the inputs of the model in a same tensor which was obviously stupid. Nikos made me realize that it is much better to give several arguments.
But I don't know how to use the first LSTM on sequences of (embedded) characters, of different length. For now I apply the LSTM on a tensor containing all the padded sequences of characters for all the words in the sentence (so 3 dimension). An LSTM object takes a 3 dimensional object:

1. `seq_len` the index of the object in the sequence given to the LSTM
2. `batch` the index of the sequence (inside a batch of sequences)
3. `input_size)` which just gives the index of the dimension of the entry of object (if a character is embedded in $\mathbb{R}^3$ this will be 3).

The problem is for me that I don't have a batch a priori. But since each sentence contains several word, I can use it as a batch of words. But I don't know if it is a good idea (especially since my understanding of the batch is that it contains data points with

independent distribution, where in the case of a sentence it might not be the case).
A bigger issue is how to retrieve the output that I am interested in inside the output of the first LSTM. As a toy example, if my sentence consists of two words ("a girl") and if the dimension of the hidden layer outputted by the LSTM is 3, the total output of the LSTM is going to be a tensor of dimension $4 \times 2 \times 3$ (4 because the longest word has 4 letters, 2 because there are two words (so batch size is 2), and 3 because the dimension of the hidden layer outputted by the LSTM is 3). But I'm only interested in the entries [0, 0, .] and [3, 1, .]. How do I get them?
But maybe I could solve this problem using the `gather` function as explained here:
`https://discuss.pytorch.org/t/select-specific-columns-of-each-row-in-a-torch-tensor/497`
and transforming my `list_len` into a LongTensor of indices (substracting 1 to each entry).

31.03.18 ~ Retrieving some entries of a tensor
I used the `gather` function and it worked very well.

02.04.18 ~ Final page of the tutorial on NLP with Pytorch, Question about inputs of pytorch models
I tried to go through this page but it was a bit too advanced (I don't know these **CRFs** well enough):
pytorch.org: Advanced: Making Dynamic Decisions and the Bi-LSTM CRF
I solved most of the problems (zero padding, retrieving output of the first lstm) that I was facing at the beginning but my model is still not working and producing errors. I feel I learned enough with this exercise. I posted a question on the official pytorch forum to know what kind of input a pytorch model is receiving (for now it receives a python list and two tensors wrapped in Variable):
pytorch.org: what kinds of input take a pytorch model?
Someone answered me on the pytorch forum (here). To summarize, **the forward method of a model can take any kind of arguments** and there was a small typo in my code (that I corrected and since then it compiles). The model with character-level features seems to work better than the original one (with only word-level features)

03.04.18 ~ torchtext.data
I came back to the lesson4-imdb notebook. I discovered the `torchtext.data` module which seems very important in NLP:
`http://torchtext.readthedocs.io/en/latest/data.html`
Then I discovered some preprocessed data sets for pytorch here:
`http://torchtext.readthedocs.io/en/latest/datasets.html`
Edit (15.06.18): See entry of the 08.05.18 (and in particular the Edit, for more details on torchtext).

10.04.18 ~ problems with cuda and pytorch 0.3.1, Explanation of Auto-

grad
It seems that with the last version of pytorch that I just installed in my conda environment mypy36 (pytorch 0.3.1) my GPU is not supported anymore. But it was in the previous one (pytorch 0.3.0) which is the one used by fastai. More precisely the error is

```
Found GPU0 GeForce 940MX which is of cuda capability 5.0.
    PyTorch no longer supports this GPU because it is too old.

  warnings.warn(old_gpu_warn % (d, name, major, capability[1]))
```

I found this page which discusses the problem:
`https://discuss.pytorch.org/t/pytorch-no-longer-supports-this-gpu-because-it-is-too-old/13803`
and they say it works "if you compile from source". But I guess it doesn't in Jupyter notebook.
This seems very tedious to change. Rather than trying to hack the thing, I will try to avoid using cuda in my tutorials (since I just want to learn) and if I need to run something on my own machine, I will use my fastai environment (having pytorch 0.3.0 on it). In the discussion above, they mention bugs fixed in pytorch 0.3.1 but not in 0.3.0. I hope that they are not too important.
I started reading this **tutorial introducing the fundamentals of pytorch**:
`http://pytorch.org/tutorials/beginner/deep_learning_60min_blitz.html`
This page in particular explains very well **what automatic differentiation is** and how **Autograd** takes care of it:
pytorch.org: blitz tutorial, autograd automatic differentiation
The next page:
pytorch.org: blitz tutorial, neural networks
explains with details how the following points are dealt with in pytorch:

1. Define the neural network that has some learnable parameters (or weights)
2. Iterate over a dataset of inputs
3. Process input through the network
4. Compute the loss (how far is the output from being correct)
5. Propagate gradients back into the network's parameters
6. Update the weights of the network

In particular this page and the previous one explains the following APIs:

- `autograd.Variable`
- `backward()`

- `nn.Module`
- `nn.Parameter`
- `autograd.Function`
- `.grad_fn`
- `.grad`
- `.step()`

Still on this tutorial:
pytorch.org: blitz tutorial, neural networks
they explain that every neural network expects batch of data as input. "f you have **a single sample**, just use `input.unsqueeze(0)` to add a fake batch dimension".

11.04.18 ~ CIFAR10 dataset, loading a dataset of image
I started reading this page:
pytorch.org: blitz tutorial, training a classifier with CIFAR10 dataset
It explains how to **load a dataset of images**. It uses "data loaders". If I understand it well, it is an object over which one can iterate to obtain batches of data. Here is a short example taken from the notebook of the tutorial:

```
trainset = torchvision.datasets.CIFAR10(root='./data', train=True,
                                        download=True, transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=4,
                                          shuffle=True, num_workers=2)
```

and then latter (this not a complete code of the training) we use:

```
for i, data in enumerate(trainloader, 0):
    # get the inputs
    inputs, labels = data

    # wrap them in Variable
    inputs, labels = Variable(inputs), Variable(labels)

    # zero the parameter gradients
    optimizer.zero_grad()

    # forward + backward + optimize
    outputs = net(inputs)
```

12.04.18 ~ Progressing with the 60 Blitz tutorial (CIFAR10 dataset), num_workers, two APIs for zero_grad, how to use GPU with cuda

I downloaded the **CIFAR10 dataset**, which is quite light (60K images of 32×32 resolution). It can be used to quickly test an algorithm. One can find information about it here:
`http://www.cs.toronto.edu/~kriz/cifar.html`
In order to make a **(real) copy of a torch tensor**, one has to do it like this:

```
my_tensor = torch.FloatTensor([[1,2],[3,4]])
my_tensor2 = my_tensor.clone()
```

I learned a bit about the `num_workers` argument of a `DataLoader` here:
`https://discuss.pytorch.org/t/guidelines-for-assigning-num-workers-to-dataloader/813/6`
I don't really understand the details, but I have the impression that it is a way to parallelize how the data is brought to the model. The RAM memory seems to be the limitation to this process. I fail to see the exact link with the number of core of the CPU and what people call "num_GPU" on the forum. I guess the "num_GPU" is the number of Graphical cards (I imagine that we can have several ones).
It seems there are **two API's to empty the gradients** (between two batches during the training) as explained here:
`https://discuss.pytorch.org/t/zero-grad-optimizer-or-net/1887`
The two possible cases are displayed in this tutorial:
`http://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html`
and in this one:
`http://pytorch.org/tutorials/beginner/nlp/sequence_models_tutorial.html`
I realized that `torch.max` (when given the argument `dim`) has two outputs: the maximum value, and the index (along the given dimension) at which the maximum is reached, as explained here:
`http://pytorch.org/docs/master/torch.html`
Finally this tutorial explains **how to use cuda in order to use the GPU**.

18.04.18 $\sim$ Data parallelism, From a Numpy implementation to a Pytorch implementation (step by step)
At the end of the 60min blitz tutorial there is a page explaining **how to use multiple GPUs using** `DataParallel`:
pytorch.org: blitz tutorial, data parallelism
but I will skip it for now, since I don't really need it now.
I started a new tutorial which introduces Pytorch using examples:
pytorch.org: pytorch with examples
It is very nice as it starts from a basic, explicit Numpy implementation, and adds one by one, the tools used in Pytorch, explaining the difference at each step. It seems that it repeats what was presented in the previous one so I won't spend too much time on it. I read the first part which shows an example of a simple model implemented using Numpy. It illustrates well what the **forward and backward passes** are.
Then it explains

- what pytorch tensors (`torch`) are,
- how automatic differentiation with `torch.autograd` and `torch.autograd.Variable` works, and how to do the backward pass with `loss.backward()`,
- how to define functions which are compatible with the pytorch automatic differentiation tools.

I had already seen a similar example previously. I stopped after this part. There are a few sections remaining which seem very interesting.

23.04.18 $\sim$ `nn` module

I continued reading this tutorial:
`http://pytorch.org/tutorials/beginner/pytorch_with_examples.html#pytorch-variables-and-a`
It presents the `nn` module as a set of high-level API's for Pytorch. It introduce in particular

- `torch.nn.Sequential`
- `torch.nn.Linear`
- `torch.nn.ReLu`
- `torch.nn.MSELoss`

When one uses these wrappers the weights become somehow "hidden" and we cannot update them explicitly (as in the previous more elementary codes). So instead of using

```
w1.data -= learning_rate * w1.grad.data
w2.data -= learning_rate * w2.grad.data
```

we use

```
for param in model.parameters():
    param.data -= learning_rate * param.grad.data
```

Then the tutorial introduces the `torch.optim` module to wrap-up the optimization algorithm used in the **update of the different parameters of the model** (in this case with `torch.optim.Adam`). The important API's are

- `optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)`
  for the initialization of the optimizer
- `optimizer.step()` to use it to update the weights/parameters

Then it moves on to **custom nn Modules** which are useful when you "want to specify models that are more complex than a sequence of existing Modules". Then one has to create a subclass of `torch.nn.Module` and implement the methods `__init__` and `forward`. An example of such custom nn Module for Convolutional neural networks can

be found in another tutorial here:
`http://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html#define-a-convolution-n`
Finally it shows an example of model where having a dynamic computational graph is useful (where the number of intermediary hidden layer is random).

I scrolled through this **tutorial about Data Loading and Processing**: `http://pytorch.org/tutorials/beginner/data_loading_tutorial.html`
It seems to be focused on Image processing which is not my goal so I will skip it for now.

25.04.18 ~ Transfer learning, Learning Rate Scheduler, Pytorch 4.0 disappearance of "Variable"
I started this tutorial about **transfer learning**:
`http://pytorch.org/tutorials/beginner/transfer_learning_tutorial.html`
I went back to this tutorial to remember how the dataloader were working:
`http://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html#what-about-data`
And I realized that it has changed. It doesn't seem to be needed anymore to use "Variables" at all. It is explained here:
`http://pytorch.org/2018/04/22/0_4_0-migration-guide.html`
Now the classes `Tensor` and `Variable` have merged. `Tensor` objects have a new attribute `requires_grad` which is by default `False` but if assigned to `True` makes the Tensor behave like a Variable. It is not exactly clear to me how this works with the `DataLoader` objects (objects over which we can iterate to get the batches (c.f. entry of the 11.04.18). I guess that the output they produce when we use

```
enumerate(trainloader, 0)
```

has this parameter `requires_grad` set to `True`.
After checking this issue I came back to the tutorial. They explain that there are **two ways to perform Transfer Learning**:

- Either initialize all the weights on the value of the weights of another model (instead of random weights) and then train the models like usual.
- Initialize all the weights on the value of the weights of another model (instead of random weights) and train only an extra layer on the top of this (the others being frozen).

In this tutorial they used `models.resnet18` as the pretrained model. I learned that these methods "use modules which have different training and evaluation behavior, such as batch normalization. To switch between these modes, use `model.train()` or `model.eval()` as appropriate", here:
`http://pytorch.org/docs/master/torchvision/models.html`
I finished the tutorial. One interesting API that I learn is about the **Learning Rate Scheduling**. One initialize the scheduler like this

```
scheduler = lr_scheduler.StepLR(optimizer_ft, step_size=7, gamma=0.1)
```

and then during the training, one starts each epoch by:

```
scheduler.step()
```

03.05.18 ∼ Tutorial about Bi-LSTM with CRF

I went through this tutorial which is using a **model combining Bi-LSTM with Conditional Random Field** in order to do tag words of sentences:
`https://pytorch.org/tutorials/beginner/nlp/advanced_tutorial.html`
The code was quite advanced and not so easy to read but I went through it. So I will gather here, what would have helped me to understand the code in the first place.
The **general idea of Conditional Random Field** (CRF) goes as follow:
An input is a sequence of items (words for instance) and we would like to assign a label/tag (belonging to a fixed set T) to each word of the sequence, taking in account that the items are part of a sentence. For instance one might want to determine automatically which word are verbs, nouns, etc... in an English sentence, and it a word is assigned to the tag "verb", it would be unlikely that the next one is again a verb. The items can also be seen as some noisy observed signal from an underlying latent variables which take value in the label/tag set T. In CRF we use a score function to determine how likely a given sequence of labels y is to be for a sequence of items x. The scoring function can be decomposed as a sum of logs of "feature functions" ψ_i:

$$Score(x, y) = \sum_i \log(\psi_i(x, y)). \tag{1}$$

Then one can find the best sequence of labels for a fixed input seuqence x by picking the sequence of label y which maximises the score. One approach might be to do an exhaustive computation of the score of each possible sequence y. In the case of **linear-chain CRF**, the feature functions are only functions of the input sequence, and of two consecutive labels. In this case, we can use the **Viterbi algorithm** in order to find the best sequence of labels. The idea is that if we know for an n fixed, what are the each of the $|T|$ sequences of length n which maximize the score while ending by each one of the $|T|$ possible labels/tag, then it is easy to find which are the $|T|$ sequences of length $n + 1$ which maximize the score while ending by each one of the $|T|$ possible labels/tag by looking only at the feature functions depending on the labels n and $n + 1$.
In the tutorial we are precisely in this case and we have only two types of feature functions: the first are "transition functions" which depend solely on the value of two consecutive labels (in the tutorial it is written as $\psi_{TRANS}(y_{i-1} \to y_i)$). Implicitly they decide that these functions are space independent (meaning for instance that it is the transition function for step 1 to 2 gives the same weight to "verb to noun", than the transition function for step 4 to 5 to "verb to noun"), and hence one can encode them all together in a single matrix of dimensions $|T| \times |T|$ (more precisely it is the log of

their value which is encoded in a matrix). The second are "emission functions" which depend only on one item (the third for instance) and its corresponding label.
The trick to **incorporate the LSTMs into a CRF model** is the following: we look at the output of the LSTM (or more precisely Bi-LSTM) for, lets say, the 3rd item. It is a vector of length $|T|$ containing the (unnormalized) probabilities of belonging to each one of the $|T|$ possible tags. This is used by the CRF model as value of the log of the emission functions on the third input/word with the $|T|$ possible tags for the value of y_3.
Now some notes concerning the implementation:

1. the methods `_forward_alg` and `_viterbi_decode` do more or less the same thing. They both use the Viterbi algorithm to find the sequence with the best score using the output of the Bi-LSTM as emission function. The difference is that `_forward_alg` doesn't keep track of the best sequence of labels and returns only the score of the best sequence. It is used only during the training phase since the loss function depends (in their implementation) only on the score of the predicted sequence of tags (and not on the sequence itself). The `_viterbi_decode` is used to do predictions (through the `forward` method) and return both the optimal sequence and its score.

2. `feats` is a matrix of dimension (length of the sentence) $\times|T|$ which contains the output of the LSTM, i.e. for each word of the sentence the probabilities to have each one of the $\times|T|$ tags.

3. `backpointers` is a list of length equal to the length of the sentence. Each entry contains a list of length $|T|$. The entry j of the sublist i contains the index of the label for word i which gives the highest score when we assign the next word to the label with index j. The entries of the sublists areare tensors.

4. `forward_var` (in `_viterbi_decode`) is a tensor of dimension $1 \times |T|$ which (at step n of the algorithm) contains in entry i the score of the best sequence of length $n-1$ finishing by the ith label. This variable is updated at each step.

5. `alphas` in `_forward_alg` seem to serve the same purpose as `forward_var` in `_viterbi_decode`.

6. The loss function is kind of strange. It is the difference between the score obtained by the best sequence of label (at this step, for this scoring function depending on weights (transition matrix and LSTM parameters) which change at each step) minus the score of the actual true sequence. As the Viterbi algorithm guarantees that the sequence picked gives the highest score, this difference is always positive. By minimizing this loss function using the usual SGD-type methods we change the scoring function at each step.

I saved the code in a file named pytorch_CRF-BiLSTM_speech_tagger.py

04.05.18 ~ Sentiment analysis with Pytorch
I started a project where I will try to do sentiment analysis with Pytorch for IMDB reviews (like in the notebook of fastai).

05.05.18 ~ A tutorial on how to build your own Dataset and DataLoader
This tutorial explains **how to load data in the python model**. More precisely it explains **how to build a Dataset** and **how to build a DataLoader** in detail:
`https://pytorch.org/tutorials/beginner/data_loading_tutorial.html#sphx-glr-beginner-data`
The main points are the following:

- The role of the Dataset object is to allow one to load the objects in python. They are accessible one by one using usual indexing and for loops as in

```
face_dataset = FaceLandmarksDataset(csv_file='faces/face_landmarks.csv',
                                    root_dir='faces/')

fig = plt.figure()

for i in range(len(face_dataset)):
    sample = face_dataset[i]

    print(i, sample['image'].shape, sample['landmarks'].shape)

    ax = plt.subplot(1, 4, i + 1)
    plt.tight_layout()
    ax.set_title('Sample #{}'.format(i))
    ax.axis('off')
    show_landmarks(**sample)

    if i == 3:
        plt.show()
        break
```

- One creates its own subclass of `Dataset`, and define the methods specific to the data that we are using.
- Upon instantiation of this subclass one can give some transformation objects (which have to be defined as classes with specific methods) as argument. The transformations implemented will be applied to each sample which is accessed via the instance.
- In order to iterate in a better way on the dataset (batching the data, shuffling the data, load the data in parallel using `multiprocessing` workers) one has to

instantiate an object of the class `DataLoader` which takes a Dataset object as argument upon instantiation as in

```
dataloader = DataLoader(transformed_dataset, batch_size=4,
                        shuffle=True, num_workers=4)
```

and can then be used as in

```
for i_batch, sample_batched in enumerate(dataloader):
    print(i_batch, sample_batched['image'].size(),
          sample_batched['landmarks'].size())
```

08.05.18 ∼ Pytorch's version not up to date in Anaconda, More on Torchtext

When I use `conda list` I see that the Pytorch version I have in my mypy36 environment is only 0.3.1, even if the version 0.4.0 is already online. I tried to update it via conda but it seems that the packages of the official Anaconda channel are not up-to-date:

```
(mypy36) aritz@aritz-ThinkPad-T460p:~$ conda update pytorch
Solving environment: done

# All requested packages already installed.
```

This page seems to indicate that there is often some delay between the appearance of a new version and its integration in the Anaconda channel:
`https://github.com/ContinuumIO/anaconda-issues/issues/7410`

Edit (15.06.18): The library **torchtext** seems to be the tool to **create Datasets from data stored in text**, similar to what was done in the Pytorch tutorial about data loading which was using pictures as data. The `torchtext.data.Dataset` is actually a subclass of `torch.utils.data.Dataset`. It **can use spaCy** as a tool to do the tokenization. If I remember well, one of the advantages of torchtext over the normal Dataset and DataLoader, is that it creates batches with text of the same length which is good because it avoid doing too much padding (texts have generally different lengths). I read this tutorial:
`http://mlexplained.com/2018/02/08/a-comprehensive-tutorial-to-torchtext/`

There are two things I didn't really get:

1. The wrapper thing. What is the type of the non-wrapped data coming out of the data loader?

2. Why does the author normalize his probabilities (predictions) outside of the model? Why isn't it a problem for the computation of the loss:

```
loss = loss_func(y, preds)
```

13.05.18 ~ Break and list of links towards tutorials

Due to my job search, I haven't been working much on pytorch recently. I will put here the links towards the the tutorials to be able to find them back later and close them now:

https://pytorch.org/tutorials/
https://pytorch.org/tutorials/beginner/data_loading_tutorial.html#sphx-glr-beginner-data
http://mlexplained.com/2018/02/08/a-comprehensive-tutorial-to-torchtext/
http://mlexplained.com/2018/02/15/language-modeling-tutorial-in-torchtext-practical-torc
https://towardsdatascience.com/use-torchtext-to-load-nlp-datasets-part-i-5da6f1c89d84

19.06.18 ~ Updating Pytorch to have Pytorch 0.4.0

I updated Pytorch in my mypy36 conda environment.

```
The following NEW packages will be INSTALLED:

    ninja:   1.8.2-py36h6bb024c_1

The following packages will be UPDATED:

    pytorch: 0.3.1-py36had95abb_0 --> 0.4.0-py36hd73e86b_0

Proceed ([y]/n)? y


Downloading and Extracting Packages
ninja-1.8.2          |  1.3 MB | ############################################################
pytorch-0.4.0        | 225.2 MB | ############################################################
Preparing transaction: done
Verifying transaction: done
Executing transaction: done
```

19.07.18 ~ type of input/output of a layer

I discovered that some layer (like `nn.Embedding`) expect to have LongTensor (aka torch.int64) as input type, but some others (like `nn.LSTM`) expect (by default) Float (aka torch.float32)

and it creates of course problems. But as I was explained on the Pytorch forum, one can change the input and output type of a layer (upon initialization) with for instance:

```
lstm_toy = nn.LSTM(input_size=emb_dim_toy,
                           hidden_size=n_lstm_units_toy,
                           dropout=1-keep_prob_toy).double()
```

(here it is the `.double()` at the end which changes that). But then I also had to change the type of the hidden and state variable.

20.08.18 ~ Special page for sentiment analysis of fast.ai with pytorch and transfer learning
`http://nlp.fast.ai/classification/2018/05/15/introducting-ulmfit.html`

04.09.18 ~ Model implemented by Aurélien
`https://github.com/coetaur0/ESIM`

07.09.18 ~ `children()` of a nn.module subclass instance, `nn.ModuleList`
When we define a model/estimator by creating a subclass of nn.module and instantiate it, it has a method (probably inherited from nn.module called `children()`. From what I understood it allows to access all the attributes of the newly defined class which are themselves instances of subclasses of nn.module objects (typically predefined layers). This seems to be useful to initialize or change all the parameters of a model/estimator.

`nn.ModuleList` seems to be a class to put several layers in a list (typically if one wants to create new class of model having one attribute being a list of dense layers or lstm layers). This link gives a nice explanation of the difference between this and `nn.Sequential`:
`https://stackoverflow.com/questions/47544051/when-should-i-use-nn-modulelist-and-when-sh`

08.09.18 ~ `param_groups` attribute from `Optimizer` instances
Inside the layer_optimizer.py from fastai, I learned a couple of things about the attribute `param_groups` of an object of the class `Optimizer` (typically an object created by a line like

```
optimizer = torch.optim.Adagrad(parameters, lr=0.1)
```

in my sentiment analysis project). It is a list containing dictionaries. Each dictionary contains

- a `'params'` entry, containing a list of trainable weights
- an `'lr'` entry containing a learning rate (optional)
- a `'weight_decay'` entry containing a weight decay parameter (optional)

The idea is that each item of the list contains weights of a separate group of layers, and the parameters (learning rate, etc...) to tune them. It can be useful when we want to tune different layers differently.

3 To Do

1. When I will have read a bit about Tensor flow, I should read this page:
   `https://towardsdatascience.com/pytorch-vs-tensorflow-spotting-the-difference-25c757`