The dictionary is case-sensitive!

Test: I love paris in the spring time I love paris in the fall I love paris in the I love franch songs and to hear them in the afternoon
Train: I went to Paris last year becuase I wanted to have some fun
Valid: yesterday I went home to see my daughter which is called ayalla and also my wife which is called tanya

```
>>> corpus.dictionary.word2idx
{'and': 21, 'love': 25, 'Paris': 3, 'spring': 29, 'is': 18, 'in': 27, 'some': 9, 'see': 14, 'have': 8, 'year': 5, 'home': 13, 'franch': 32, 'also': 22, 'paris': 26, 'to': 2,
'tanya': 24, 'which': 17, 'becuase': 6, 'them': 35, 'I': 0, 'ayalla': 20, 'yesterday': 12, '<eos>': 11, 'hear': 34, 'fall': 31, 'last': 4, 'wanted': 7, 'the': 28,
'daughter': 16, 'wife': 23, 'afternoon': 36, 'time': 30, 'fun': 10, 'went': 1, 'my': 15, 'called': 19, 'songs': 33}

>>> corpus.dictionary.idx2word
['I', 'went', 'to', 'Paris', 'last', 'year', 'becuase', 'wanted', 'have', 'some', 'fun', '<eos>', 'yesterday', 'home', 'see', 'my', 'daughter', 'which', 'is', 'called',
'ayalla', 'and', 'also', 'wife', 'tanya', 'love', 'paris', 'in', 'the', 'spring', 'time', 'fall', 'franch', 'songs', 'hear', 'them', 'afternoon']
```

The Corpus is divided to p-batches and is parallel analyzed by the <p-batches> nets. A loss is calculated on all seq_len on the <p-batches> nets.

The words in train.txt were divided to parralel-batches; every column is a p-batch. All p-batches are entered together to the net. (batches are later made from rows @ get_batch) A loss and an update is per batch (not per p-batch).

```
>>> train_data

  0    0
  1    7
  2    2
  3    8
  4    9
  5   10
  6   11
[torch.LongTensor of size 7x2]
```

args

Data.corpus(args.data)

Args.
batch_size=2
(p-batch)

corpus

.train .valid .test

batchify | batchify | batchify

train_data | val_data | test_data

```
>>> corpus.valid

 12
  0
  1
 13
  2
 14
 15
 16
 17
 18
 19
 20
 21
 22
 15
 23
 17
 18
 19
 24
 11
[torch.LongTensor of size 21]
```

```
>>> corpus.train

  0
  1
  2
  3
  4
  5
  6
  0
  7
  2
  8
  9
 10
 11
[torch.LongTensor of size 14]
```

```
>>> corpus.test

  0
 25
 26
 27
 28
 29
 30
  0
 25
 26
 27
 28
 31
  0
 25
 26
 27
 28
  0
 25
 32
 33
 21
  2
 34
 35
 27
 28
 36
 11
[torch.LongTensor of size 30]
```

seq_len = 3
Takes the data raw-wise!

batch=1:
>>> data
Variable containing:
 3 8
 4 9
 5 10
[torch.LongTensor of size 3x2]

batch=0: **(I: m-batch idx)**
>>> data
Variable containing:
 0 0 => [b0:v0 b1:v0]
 1 7 => [b0:v1 b1:v1]
 2 2 => [b0:v2 b1:v2]
[torch.LongTensor of size 3x2]

data, targets = get_batch(train_data, i)

data | targets

batch=1:
>>> targets
Variable containing:
 4
 9
 5
10
 6
11
[torch.LongTensor of size 6]

batch=0:
>>> targets
Variable containing:
 1 => [b0:v0]
 7 => [b1:v0]
 2 => [b0:v1]
 2 => [b1:v1]
 3 => [b0:v2]
 8 => [b1:v2]
[torch.LongTensor of size 6]

RNNModel (
  (drop): Dropout (p = 0.2)
  (encoder): Embedding(37, 200)
  (rnn): LSTM(200, 200, num_layers=2, dropout=0.2)
  (decoder): Linear (200 -> 37))

Criterion:
CrossEntropyLoss ()

Single init @ train

hidden = model.init_hidden(args.batch_size)

Feeding the model with input of size N (I.e data has N rows) is like feeding it with data of size 1 N times (and off-course looping the hidden from output to input N-1 times)

2xp-batchSizex200]

output, hidden = model(data, hidden) | hidden | hidden = repackage_hidden(hidden)

```
>>>
output.data.shape
(3L, 2L, 37L):

[0:,0,0:] :
Batch00 – vec00
Batch00 – vec01
Batch00 – vec02

[0:,1,0:] :
Batch01 – vec00
Batch01 – vec01
Batch01 – vec02
```

output

Matrix with lines:

Batch00 – vec00
Batch01 – vec00
Batch00 – vec01
Batch01 – vec01
Batch00 – vec02
Batch01 – vec02

view

criterion

loss

```
loss.backward()

torch.nn.utils.clip_grad_norm(model.parameters(), args.clip)

for p in model.parameters():
        p.data.add_(-lr, p.grad.data)
```

INPUTS

OUTPUTS

I=0:
>>> data
Variable containing:
 0 0
 1 7
 2 2
[torch.LongTensor of size 3x2]

data

A simple lookup table that stores embeddings of a fixed dictionary and size.

This module is often used to store word embeddings and retrieve them using indices. The input to the module is a list of indices, and the output is the corresponding word embeddings.

This is also trained:
>>> self.encoder.weight.requires_grad
True

Our dim is 200

Self.encoder = nn.embedding

Word Embeddings

>>> emb.data.shape
(3L, 2L, 200L)

Self.drop = dropout(0.2)

emb

2xp-batchSizex200]

LSTM

hidden

output

>>> output.data.shape
(3L, 2L, 200L) = (3 x batchSize x 200)

Self.drop = dropout(0.2)

output

>>>
output.data.shape
(3L, 2L, 37L):

[0:,0,0:] :
Batch00 – vec00
Batch00 – vec01
Batch00 – vec02

[0:,1,0:] :
Batch01 – vec00
Batch01 – vec01
Batch01 – vec02

[torch.FloatTensor of size 6x200]

A matrix whose lines are:

Batch00 – vec00
Batch01 – vec00
Batch00 – vec01
Batch01 – vec01
Batch00 – vec02
Batch01 – vec02

view

view

Decoder = linear(200,ntokens)

decoded

>>> decoded.data.shape
(6L, 37L)

Output[n] = decoded = LSTM(emb,hidden) = LSTM(v[n],hidden) **=>**
We can gradient output[n] w.r.t v[n]