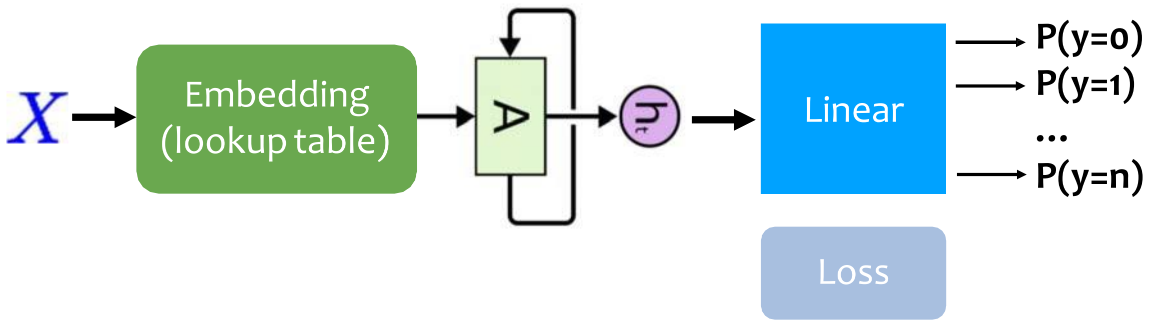
WORD EMBEDDING

Typical RNN Models

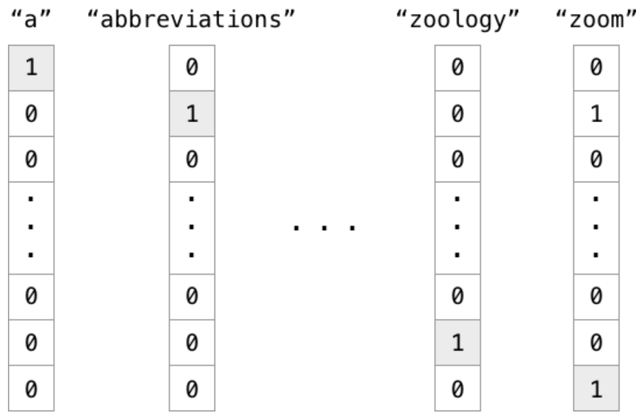
- In Natural Language Processing (NLP), we often map words into vectors that contains numeric values so that machine can understand it.



With CrossEntropy

Traditional Approach

- The traditional Approach to turn text into numbers is one-hot encoding.



- Assume we have a dictionary of 2000 words. When using one-hot encoding, each word will be represented by a vector containing 2000 integers (2000-D).
- And 1999 of these integers are zeros. In a big dataset this approach is not computationally efficient.

One-hot encoding

- There are several issues for one-hot encoding.
  - You cannot infer any relationship between two words given their one-hot representation.
    - For instance, the word "endure" and "tolerate", although have similar meaning, their targets "1" are far from each other.
  - One-hot encoded vectors are high-dimensional and sparse.
    - There are numerous redundant "0" in the vectors, wasting a lot of space.

Word Embedding

5

- The Big Idea of Word Embedding is to turn text into vector of real numbers.
- Word Embedding aims to create a vector representation with a much lower dimensional space. These are called *Word Vectors*.
- This vector representation has two important and advantageous properties:
  - **Dimensionality Reduction** — it is a more efficient representation
  - **Contextual Similarity** — it is a more expressive representation

Word Embedding

6

- Word Vectors are used for semantic parsing, to extract meaning from text to enable natural language understanding.
- For a language model to be able to predict the meaning of text, it needs to be aware of the contextual similarity of words.
- For instance, that we tend to find fruit words (like apple or orange) in sentences where they're grown, picked, eaten and juiced, but wouldn't expect to find those same concepts in such close proximity to, say, the word airplane.
- The vectors created by Word Embedding preserve these similarities, so words that regularly occur nearby in text will also be in close proximity in vector space.

Word2Vec

- What is word embedding?" is: it's a means of **building a low-dimensional vector representation from corpus of text, which preserves the contextual similarity of words.**
- And this is the approach used by one of the best known algorithms for producing word embeddings: word2vec.
- There are actually two ways to implement word2vec
  - CBOW (Continuous Bag-Of-Words) and Skip-gram.

Word2Vec

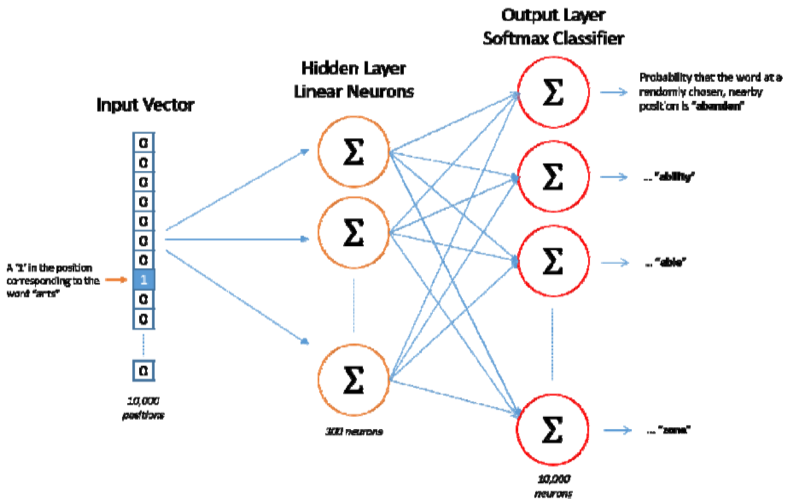
- CBOW
  - In CBOW we have a window around some target word and then consider the words around it (its context).
  - We supply those words as input into our network and then use it to try to predict the target word.
- Skip-gram
  - Skip-gram does the opposite, you have a target word, and you try to predict the words that are in the window around that word, i.e. predict the context around a word.

The Skip-Gram Model

9

- The input words are passed in as one-hot encoded vectors.
- This will go into a hidden layer of linear units, then into a softmax layer to make a prediction.

- The idea here is to train the hidden layer weight matrix to find efficient representations for our words.
- This weight matrix is usually called the **embedding** matrix, and can be queried as a look-up table.



The Skip-Gram Model

10

- The embedding matrix has a size of the number of words by the number of neurons in the hidden layer (the embed size).
- So, if you have 10,000 words and 300 hidden units, the matrix will have size 10,000×300 (as we're using one-hot encoded vectors for our inputs). Once computed, getting the word vector is a speedy O(1) lookup of corresponding row of the results matrix:

$$[0 \quad 0 \quad 0 \quad 1 \quad 0] \times \begin{bmatrix} 17 & 24 & 1 \\ 23 & 5 & 7 \\ 4 & 6 & 13 \\ 10 & 12 & 19 \\ 11 & 18 & 25 \end{bmatrix} = [10 \quad 12 \quad 19]$$

So, for the word that's the 4th entry in the vocabulary, its vector is (10,12,19).

each word has an associated vector, hence the name: word2vec.

Embed size

11

- The embed size, which is the size of the hidden layer and thus the number of features that represent similarities between words, tends to be much smaller than the total number of unique words in the vocabulary, (hundreds rather than tens of thousands).
- The embed size used is a trade-off: more features mean extra computational complexity, and so longer run-times, but also allow more subtle representations, and potentially better models.

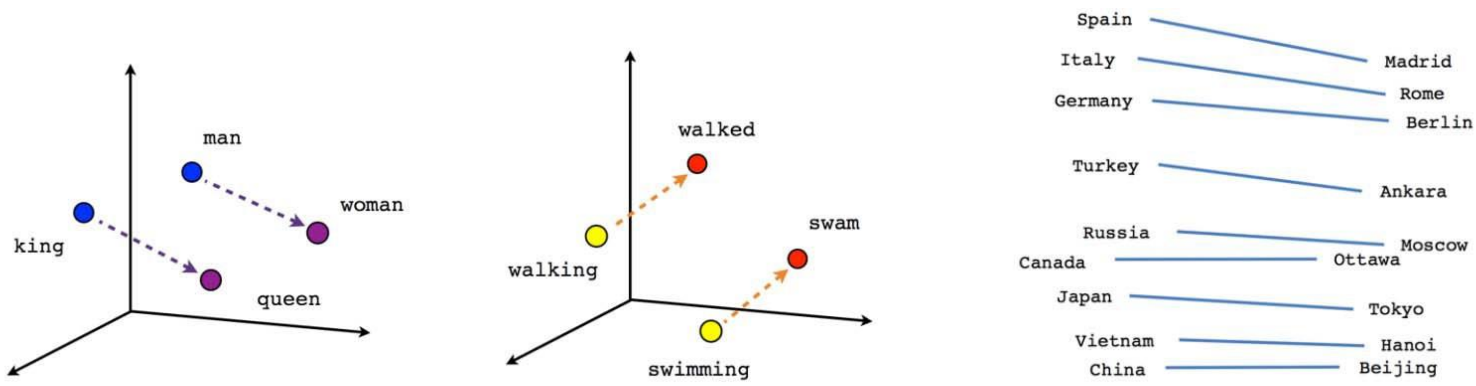
Contextual similarities

12

- Word Embeddings are similarities based on context, which might be gender, tense, geography or something else entirely.
- The classic example is subtracting the ‘notion’ of “King” from “Man” and adding the notion of “Woman”. The answer will depend on your training set, but you’re likely to see one of the top results being the word “Queen”.

Contextual similarities

13



Male-Female Verb tense Country-Capital

The lines shown are just mathematical vectors, so see how you could move 'across' in embedding space from "Man" to "Queen" by subtracting "King" and adding "Woman".

class torch.nn.Embedding

14

- 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.
- Parameters:
  - num_embeddings (int) – size of the dictionary of embeddings
  - embedding_dim (int) – the size of each embedding vector
  - padding_idx (int, optional) – If given, pads the output with the embedding vector at padding_idx (initialized to zeros) whenever it encounters the index.

$$\begin{bmatrix} 17 & 24 & 1 \\ 23 & 5 & 7 \\ 4 & 6 & 13 \\ 10 & 12 & 19 \\ 11 & 18 & 25 \end{bmatrix}$$

class torch.nn.Embedding

- Parameters:
  - max_norm (float, optional) – If given, will renormalize the embedding vectors to have a norm lesser than this before extracting.
  - norm_type (float, optional) – The p of the p-norm to compute for the max_norm option. Default 2.
  - scale_grad_by_freq (boolean, optional) – if given, this will scale gradients by the inverse of frequency of the words in the mini-batch. Default False.
  - sparse (bool, optional) – if True, gradient w.r.t. weight matrix will be a sparse tensor. See Notes for more details regarding sparse gradients.
- Variables:
  - weight (Tensor) – the learnable weights of the module of shape (num_embeddings, embedding_dim)

class torch.nn.Embedding

- The input to the module is a list of indices, and the output is the corresponding word embeddings.
- Shape:
  - Input: LongTensor of arbitrary shape containing the indices to extract
  - Output: (*, embedding_dim), where * is the input shape

class torch.nn.Embedding

- Note
  - Keep in mind that only a limited number of optimizers support sparse gradients: currently it's optim.SGD (CUDA and CPU), optim.SparseAdam (CUDA and CPU) and optim.Adagrad (CPU)
  - With padding_idx set, the embedding vector at padding_idx is initialized to all zeros. However, note that this vector can be modified afterwards, e.g., using a customized initialization method, and thus changing the vector used to pad the output. The gradient for this vector from Embedding is always zero.

Example #1, Embedding


```
>>> # an Embedding module containing 10 tensors of size 3
>>> embedding = nn.Embedding(10, 3)
>>> # a batch of 2 samples of 4 indices each
>>> input = torch.LongTensor([[1,2,4,5],[4,3,2,9]])
>>> embedding(input)
tensor([[[-0.0251, -1.6902,  0.7172],
         [-0.6431,  0.0748,  0.6969],
         [ 1.4970,  1.3448, -0.9685],
         [-0.3677, -2.7265, -0.1685]],

        [[ 1.4970,  1.3448, -0.9685],
         [ 0.4362, -0.4004,  0.9400],
         [-0.6431,  0.0748,  0.6969],
         [ 0.9124, -2.3616,  1.1151]]])
```

- Input: LongTensor of arbitrary shape containing the indices to extract
- Output: (*, embedding_dim), where * is the input shape

Example #2, padding_idx


```
>>> # example with padding_idx
>>> embedding = nn.Embedding(10, 3, padding_idx=0)
>>> input = torch.LongTensor([[0,2,0,5]])
>>> embedding(input)
tensor([[[ 0.0000,  0.0000,  0.0000],
         [ 0.1535, -2.0309,  0.9315],
         [ 0.0000,  0.0000,  0.0000],
         [-0.1655,  0.9897,  0.0635]]])
```

遇到這個index值，
embedding vector 就給0

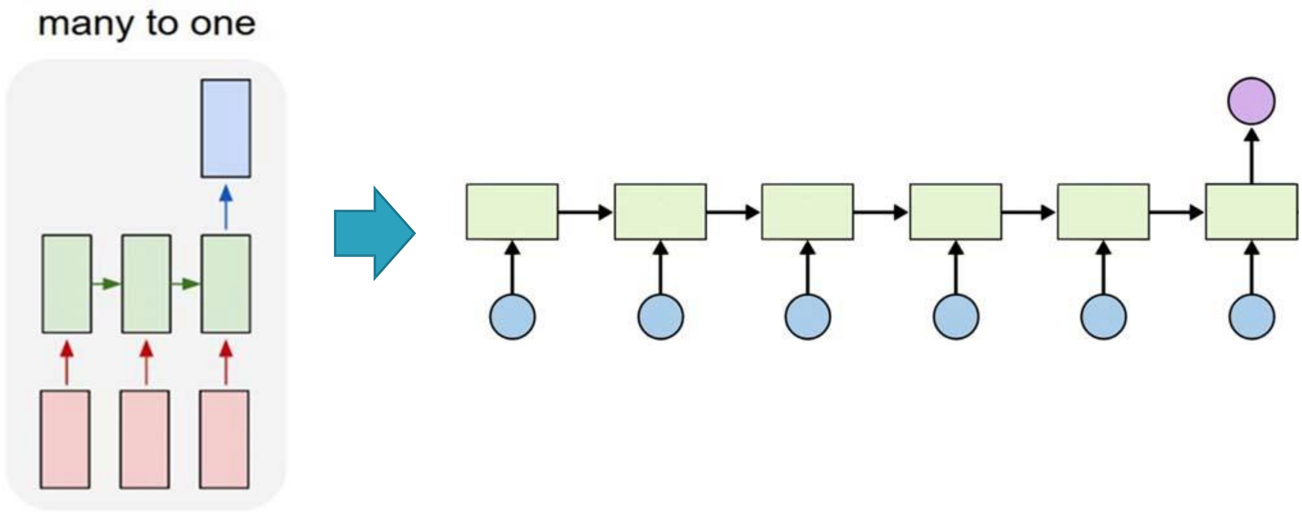
- padding_idx (int, optional) – If given, pads the output with the embedding vector at padding_idx (initialized to zeros) whenever it encounters the index.
- Input: LongTensor of arbitrary shape containing the indices to extract
- Output: (*, embedding_dim), where * is the input shape

Example #3, Embedding ‘hello’


```
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.autograd import Variable

word_to_ix = {'hello': 0, 'world': 1}
embeds = nn.Embedding(2, 5)
hello_idx = torch.LongTensor([word_to_ix['hello']])
hello_embed = embeds(hello_idx)
print(hello_embed)
```

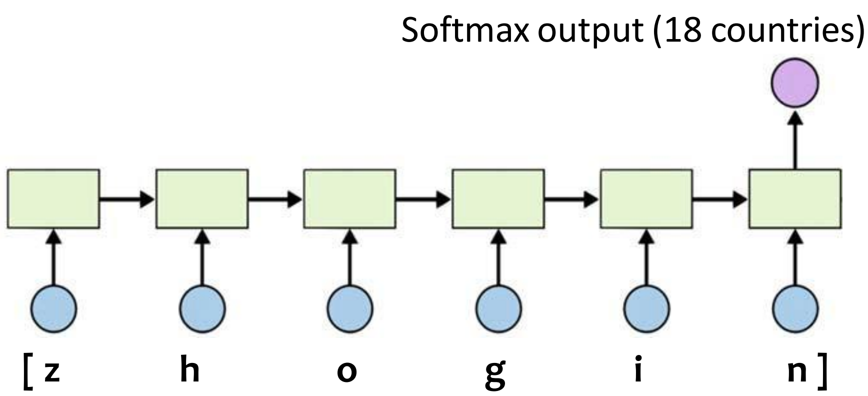
```
Out:
tensor([[ 0.2862, -0.7988, -1.3012, -2.0746, -0.4283]],
grad_fn=<EmbeddingBackward>)
```

RNN Classification



Name Classification: Dataset

0	Nader	Arabic
1	Malouf	Arabic
2	Terajima	Japanese
3	Huie	Chinese
4	Chertushkin	Russian
5	Davletkildeev	Russian
6	Movchun	Russian
7	Pokhvoschev	Russian
8	Zhogin	Russian
9	Hancock	English
10	Tomkins	English



We'll train on a few thousand surnames from 18 languages of origin, and predict which language a name is from based on the spelling.

https://github.com/hunkim/PyTorchZeroToAll
https://github.com/ngarneau/understanding-pytorch-batching-lstm

Input representation

[a d y l o v]

97 100 121 108 111 118

transpose

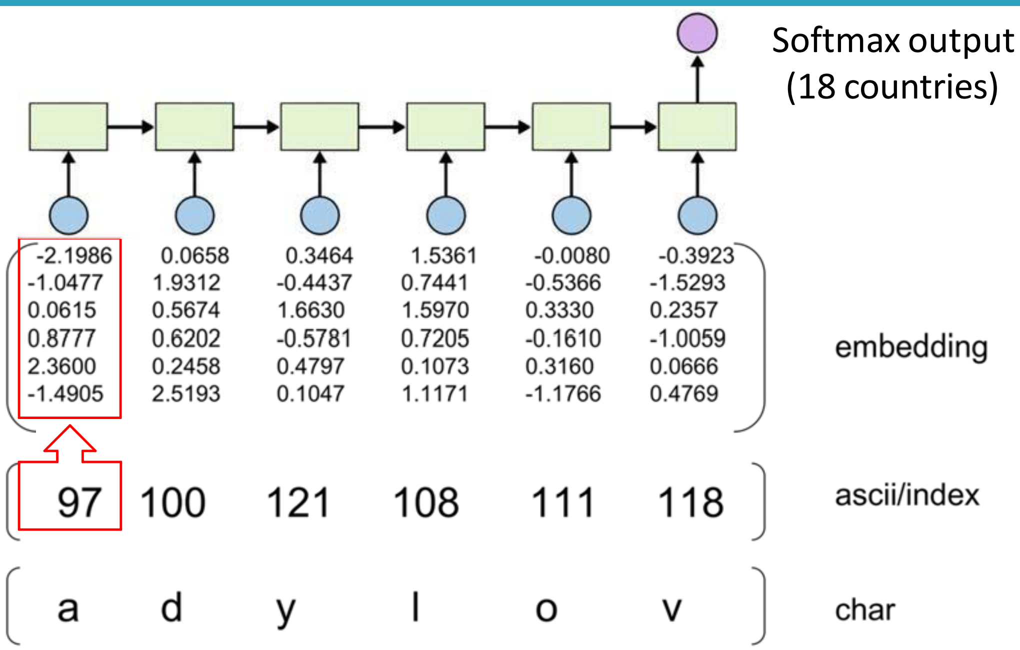
Use an embedding vector to represent a character. Use a matrix to represent the word. embedding_dim =**6**

Convert characters to ASCII code numbers, and regard as indices to retrieve embedding vectors.

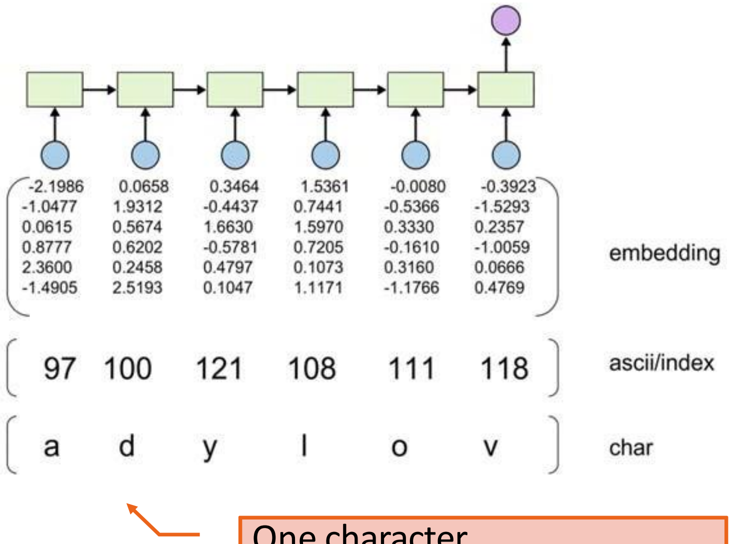
adylov						
97	-2.1986	-1.0477	0.0615	0.8777	2.3600	-1.4905
100	0.0658	1.9312	0.5674	0.6202	0.2458	2.5193
121	0.3464	-0.4437	1.6630	-0.5781	0.4797	0.1047
108	1.5361	0.7441	1.5970	0.7205	0.1073	1.1171
111	-0.0080	-0.5366	0.3330	-0.1610	0.3160	-1.1766
118	-0.3923	-1.5293	0.2357	-1.0059	0.0666	0.4769

adylov Embedding

Input representation



Data preparation



One character,
one word embedding vector

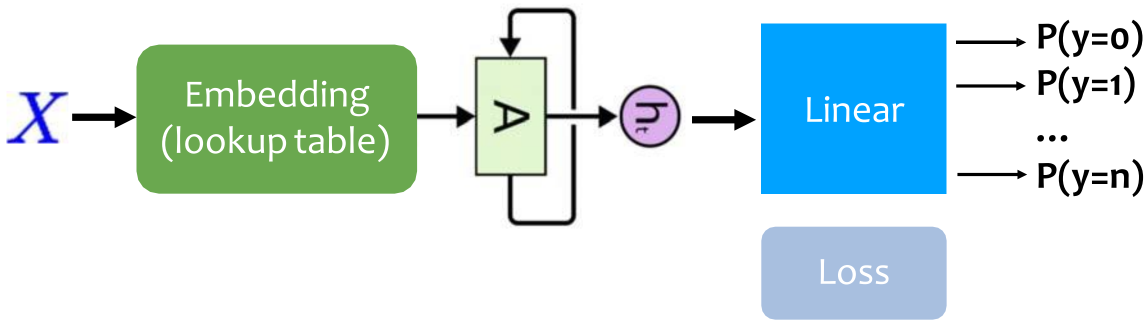
```
self.embedding =
nn.Embedding(input_voc_size,
rnn_input_size)

...
embedded = self.embedding(input)
```

```
def str2ascii_arr(name):
    arr = [ord(c) for c in name]
    return arr, len(arr)
```

`ord()` takes a character and returns its ASCII/UTF-8 integer value

Typical RNN Models

- In Natural Language Processing (NLP), we often map words into vectors that contains numeric values so that machine can understand it.



With CrossEntropy

class torch.nn.Linear

- Applies a linear transformation to the incoming data: $y=xA^T+b$
- Parameters:
  - in_features – size of each input sample
  - out_features – size of each output sample
  - bias – If set to False, the layer will not learn an additive bias. Default: True
- Shape
  - Input: (N,∗,in_features) where ∗ means any number of additional dimensions
  - Output: (N,∗,out_features) where all but the last dimension are the same shape as the input.

class torch.nn.GRU

- Parameters
  - **input_size** – The number of expected features in the input *x*
  - **hidden_size** – The number of features in the hidden state *h*
  - **num_layers** – Number of recurrent layers. E.g., setting num_layers=2 would mean stacking two GRUs together to form a *stacked GRU*, with the second GRU taking in outputs of the first GRU and computing the final results. Default: 1
  - **bias** – If False, then the layer does not use bias weights *b_ih* and *b_hh*. Default: True
  - **batch_first** – If True, then the input and output tensors are provided as (batch, seq, feature). Default: False
  - **dropout** – If non-zero, introduces a *Dropout* layer on the outputs of each GRU layer except the last layer, with dropout probability equal to dropout. Default: 0
  - **bidirectional** – If True, becomes a bidirectional GRU. Default: False

class torch.nn.GRU

- Inputs: input, **h_0**
  - input of shape (seq_len, batch, input_size): tensor containing the features of the input sequence. (seq_len = time_step, input_size = features)
  - The input can also be a packed variable length sequence. See torch.nn.utils.rnn.pack_padded_sequence() for details.
  - h_0 of shape (num_layers * num_directions, batch, hidden_size): tensor containing the initial hidden state for each element in the batch. Defaults to zero if not provided.

class torch.nn.GRU

- Outputs: output, **h_n**
  - output of shape (seq_len, batch, num_directions * hidden_size): tensor containing the output features (h_t) from the last layer of the GRU, for each t. If a torch.nn.utils.rnn.PackedSequence has been given as the input, the output will also be a packed sequence.
  - h_n of shape (num_layers * num_directions, batch, hidden_size): tensor containing the hidden state for t = seq_len.
  - Like output, the layers can be separated using h_n.view(num_layers, num_directions, batch, hidden_size) and similarly for c_n.

class torch.nn.GRU

- Examples:

```
>>> rnn = nn.GRU(10, 20, 2)
>>> input = torch.randn(5, 3, 10)
>>> h0 = torch.randn(2, 3, 20)
>>> output, hn = rnn(input, h0)
```

```
class RNNClassifier(nn.Module):
def__init__(self, input_size, hidden_size, output_size, n_layers=1):
        super(RNNClassifier, self).__init__()
        self.hidden_size = hidden_size # 100
        self.n_layers = n_layers
        self.embedding = nn.Embedding(input_size, hidden_size) # 128x100
        # GRU (input_size, hidden_size, num_layers)
        self.gru = nn.GRU(hidden_size, hidden_size, n_layers)
        self.fc = nn.Linear(hidden_size, output_size) # 100x18

    def forward(self, input):
        # Note: we run this all at once (over the whole input sequence)
        # input = B x S . size(0) = B
        batch_size = input.size(0)
        # input:  B x S  -- (transpose) --> S x B
        input = input.t()
        print(input)
```

embedding_dim=
RNN_input_size= 100

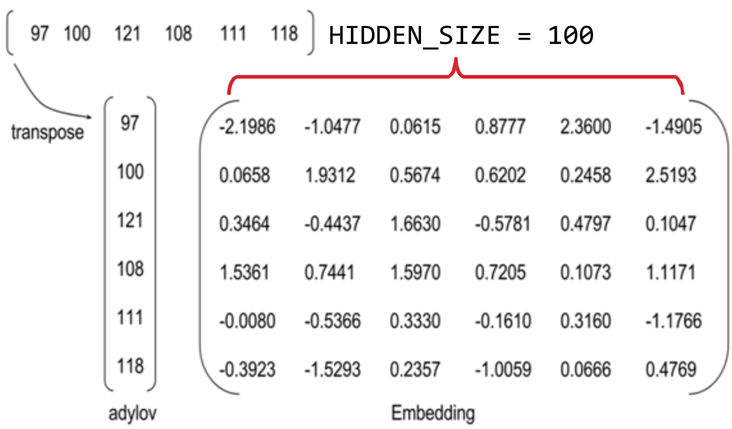
HIDDEN_SIZE = 100
N_CHARS = 128 # ASCII
N_CLASSES = 18

```python
        # Embedding S x B -> S x B x I (embedding size)
        print("  input", input.size())
        embedded = self.embedding(input)
        print("  embedding", embedded.size())

        # Make a hidden
        hidden = self._init_hidden(batch_size)
        # GRU: input of shape (seq_len, batch, input_size)
        output, hidden = self.gru(embedded, hidden)
        print("  gru hidden output", hidden.size())
        # Use the last layer output as FC's input
        # No need to unpack, since we are going to use hidden
        fc_output = self.fc(hidden)
        print("  fc output", fc_output.size())
        return fc_output

    def _init_hidden(self, batch_size):
        hidden = torch.zeros(self.n_layers, batch_size, self.hidden_size)
        return Variable(hidden)
```

Out:

```
input torch.Size([6, 1])
embedding torch.Size([6, 1, 100])
gru hidden output torch.Size([1, 1, 100])
fc output torch.Size([1, 1, 18])
```

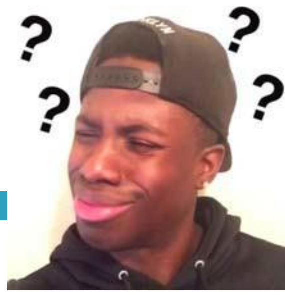
```python
# Parameters and DataLoaders
HIDDEN_SIZE = 100
N_CHARS = 128  # ASCII
N_CLASSES = 18

def str2ascii_arr(msg):
    arr = [ord(c) for c in msg]
    return arr, len(arr)

if __name__ == '__main__':
    classifier = RNNClassifier(N_CHARS,
                               HIDDEN_SIZE, N_CLASSES)

    arr, _ = str2ascii_arr('adylov')
    inp = Variable(torch.LongTensor([arr]))
    out = classifier(inp)
    print("in", inp.size(), "out", out.size())
```



Out:

```
  input torch.Size([6, 1])
  embedding torch.Size([6, 1, 100])
  gru hidden output torch.Size([1, 1, 100])
  fc output torch.Size([1, 1, 18])
in torch.Size([1, 6]) out torch.Size([1, 1, 18])
```

Batch?



```python
if __name__ == '__main__':
    names = ['adylov', 'solan', 'hard', 'san']
    classifier = RNNClassifier(N_CHARS, HIDDEN_SIZE,
N_CLASSES)

    for name in names:
        arr, _ = str2ascii_arr(name)
        inp = Variable(torch.LongTensor([arr]))
        out = classifier(inp)
        print("in", inp.size(), "out", out.size())

    # in torch.Size([1, 6]) out torch.Size([1, 1, 18])
    # in torch.Size([1, 5]) out torch.Size([1, 1, 18])
    # ...
```

adylov	sloan	harb	san
97	115	104	115
100	111	97	97
121	108	114	110
108	97	100	?
111	110	?	
118	?		

Zero padding

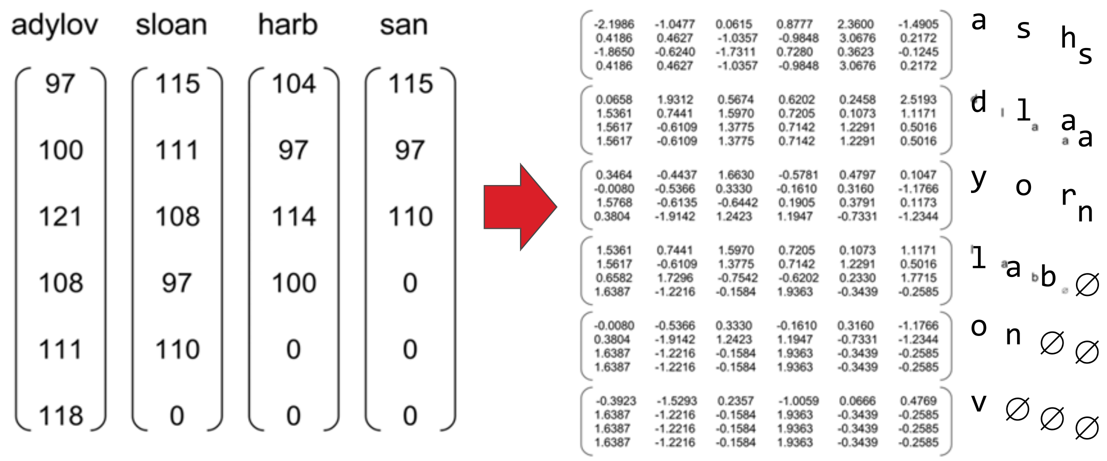
adylov	sloan	harb	san
97	115	104	115
100	111	97	97
121	108	114	110
108	97	100	
111	110		
118			

→

adylov	sloan	harb	san
97	115	104	115
100	111	97	97
121	108	114	110
108	97	100	0
111	110	0	0
118	0	0	0

```python
def pad_sequences(vectorized_seqs, seq_lengths):
    seq_tensor = torch.zeros((len(vectorized_seqs), seq_lengths.max())).long()
    for idx, (seq, seq_len) in enumerate(zip(vectorized_seqs, seq_lengths)):
        seq_tensor[idx, :seq_len] = torch.LongTensor(seq)
    return  seq_tensor
```

Embedding




```
# Embedding S x B -> S x B x I (embedding size)
embedded = self.embedding(input)
```

Use the same class RNNClassifier

```
# pad sequences and sort the tensor
def pad_sequences(vectorized_seqs, seq_lengths):
    seq_tensor = torch.zeros((len(vectorized_seqs), seq_lengths.max())).long()
    for idx, (seq, seq_len) in enumerate(zip(vectorized_seqs, seq_lengths)):
        seq_tensor[idx, :seq_len] = torch.LongTensor(seq)
    return seq_tensor

# Create necessary variables, lengths, and target
def make_variables(names):
    sequence_and_length = [str2ascii_arr(name) for name in names]
    vectorized_seqs = [sl[0] for sl in sequence_and_length]
    seq_lengths = torch.LongTensor([sl[1] for sl in sequence_and_length])
    return pad_sequences(vectorized_seqs, seq_lengths)
```

Full implementation
https://github.com/hunkim/PyTorchZeroToAll/blob/master/13_1_rnn_classification_basics.py

```python
if __name__ == '__main__':
    names = ['adylov', 'solan', 'hard', 'san']
    classifier = RNNClassifier(N_CHARS, HIDDEN_SIZE, N_CLASSES)
    inputs = make_variables(names)
    print(inputs)
    out = classifier(inputs)
    print("batch in", inputs.size(), "batch out", out.size())
```

Full implementation

https://github.com/hunkim/PyTorchZeroToAll/blob/master/13_1_rnn_classification_basics.py

```
Out:
tensor([[ 97, 100, 121, 108, 111, 118],
        [115, 111, 108,  97, 110,   0],
        [104,  97, 114, 100,   0,   0],
        [115,  97, 110,   0,   0,   0]])
tensor([[ 97, 115, 104, 115],
        [100, 111,  97,  97],
        [121, 108, 114, 110],
        [108,  97, 100,   0],
        [111, 110,   0,   0],
        [118,   0,   0,   0]])
  input torch.Size([6, 4])
  embedding torch.Size([6, 4, 100])
  gru hidden output torch.Size([1, 4, 100])
  fc output torch.Size([1, 4, 18])
batch in torch.Size([4, 6]) batch out
torch.Size([1, 4, 18])
```

class torch.nn.utils.rnn.PackedSequence

41

- Holds the data and list of batch_sizes of a packed sequence.
- All RNN modules accept packed sequences as inputs.
- Variables:
  - data (Tensor) – Tensor containing packed sequence
  - batch_sizes (Tensor) – Tensor of integers holding information about the batch size at each sequence step
- Note
  - Instances of this class should never be created manually. They are meant to be instantiated by functions like **pack_padded_sequence().**
  - Batch sizes represent the number elements at each sequence step in the batch, not the varying sequence lengths passed to pack_padded_sequence(). For instance, given data abc and x the PackedSequence would contain data axbc with batch_sizes=[2,1,1].

torch.nn.utils.rnn.pack_padded_sequence

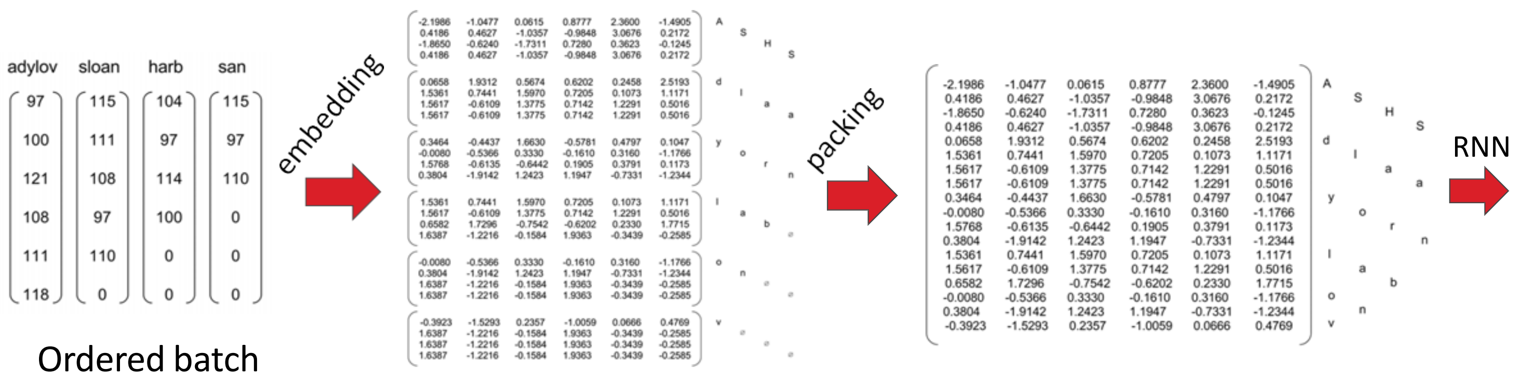
42

- Packs a Tensor containing padded sequences of variable length.
- Input can be of size T x B x * where T is the length of the longest sequence (equal to lengths[0]), B is the batch size, and * is any number of dimensions (including 0). If batch_first is True B x T x * inputs are expected.
- The sequences should be sorted by length in a decreasing order, i.e. input[:,0] should be the longest sequence, and input[:,B-1] the shortest one.

torch.nn.utils.rnn.pack_padded_sequence

- Parameters:
  - input (Tensor) – padded batch of variable length sequences.
  - lengths (Tensor) – list of sequences lengths of each batch element.
  - batch_first (bool, optional) – if True, the input is expected in B x T x * format.
- Returns:
  - a PackedSequence object
- Note
  - This function accepts any input that has at least two dimensions. You can apply it to pack the labels, and use the output of the RNN with them to compute the loss directly.
  - A Tensor can be retrieved from a PackedSequence object by accessing its .data attribute.

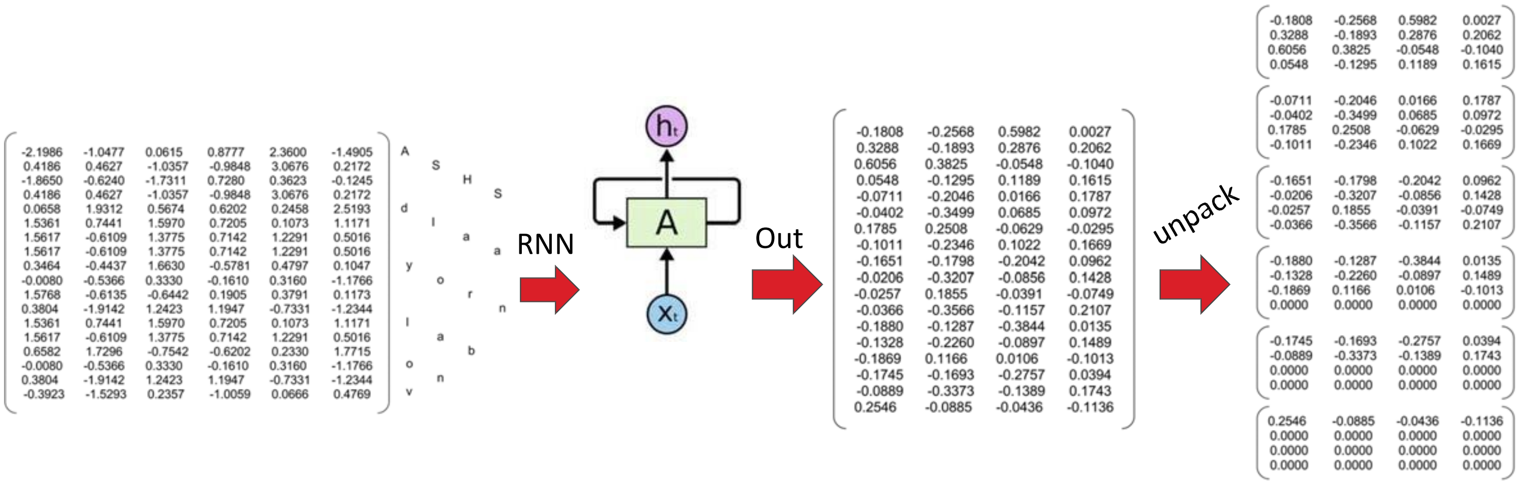
Efficiently handling batched sequences with variable lengths: pack_padded_sequence



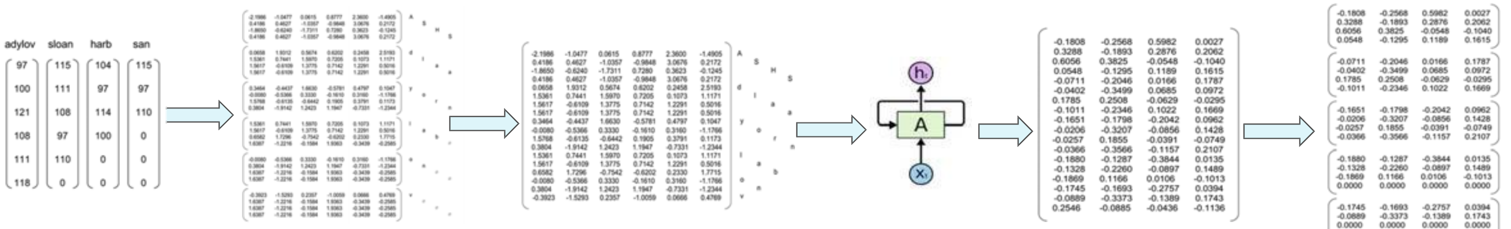
Ordered batch
(by length)

https://github.com/hunkim/PyTorchZeroToAll/blob/master/13_4_pack_pad.py
Matrix visualization from Nicolas, https://github.com/ngarneau

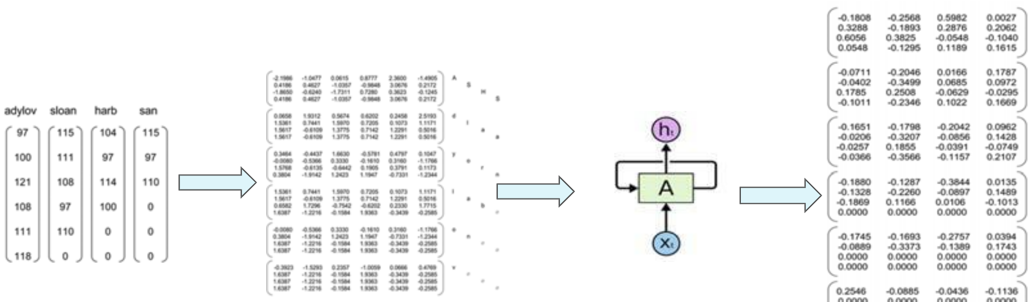
Efficiently handling batched sequences with variable lengths: pack_padded_sequence



Efficiently handling batched sequences with variable lengths: pack_padded_sequence



(a) With packing and unpacking



(b) Without packing and unpacking

```python
def forward(self, input, seq_lengths):
    # Note: we run this all at once (over the whole input sequence)
    # input shape: B x S (input size), transpose to make S x B
    input = input.t()
    batch_size = input.size(1)
    # Make a hidden
    hidden = self._init_hidden(batch_size)
    # Embedding S x B -> S x B x I (embedding size)
    embedded = self.embedding(input)
    # Pack them up nicely
    gru_input = pack_padded_sequence(embedded, seq_lengths.data.cpu().numpy())

    # To compact weights again call flatten_parameters().
    self.gru.flatten_parameters()
    output, hidden = self.gru(gru_input, hidden)

    # Use the last layer output as FC's input
    # No need to unpack, since we are going to use hidden
    fc_output = self.fc(hidden[-1])
    return fc_output
```

https://github.com/hunkim/PyTorchZeroToAll/blob/master/13_2_rnn_classification.py

Homework 14

- Implement the name classification
  - Use PyTorch
  - Use pad-pack
  - Compare RNN, LSTM, GRU, and different HIDDEN_SIZEs
- Reference code
  - https://github.com/hunkim/PyTorchZeroToAll/blob/master/13_2_rnn_classification.py

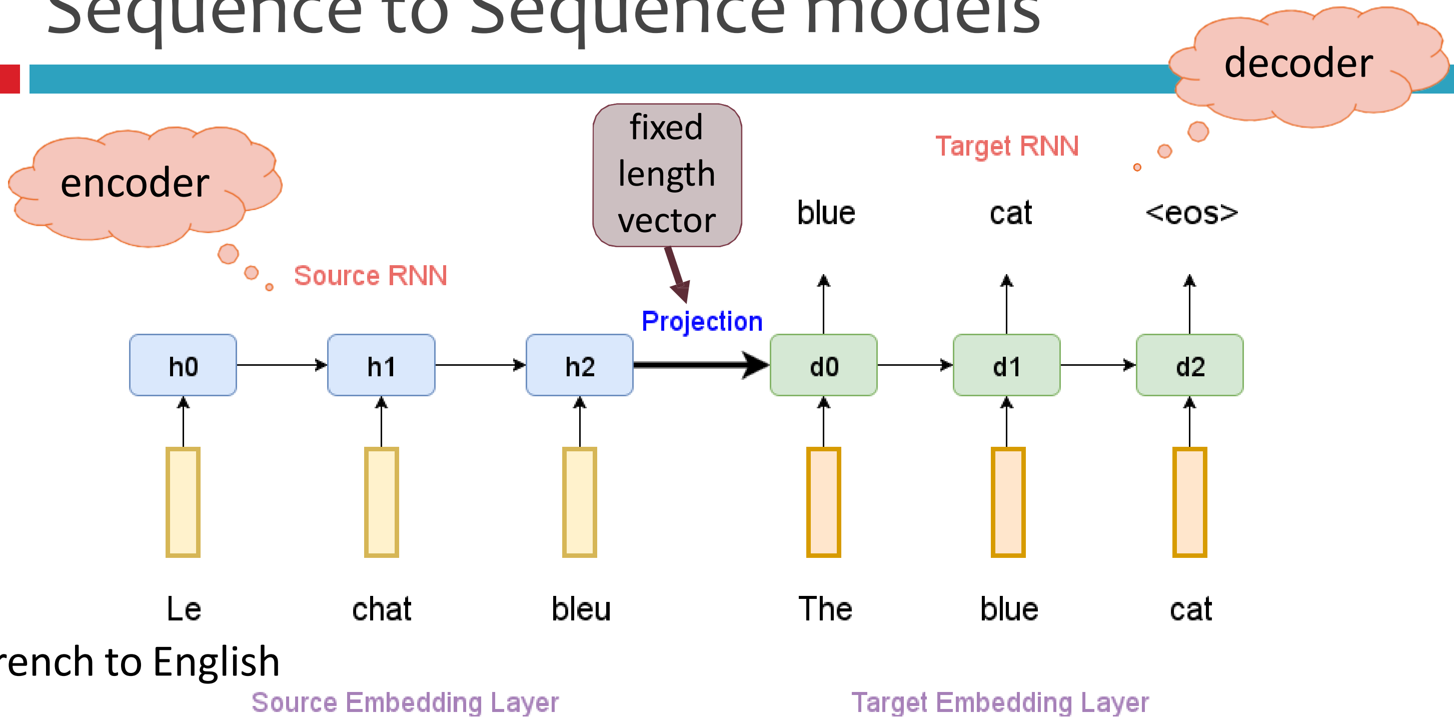
TRANSLATION WITH A SEQUENCE TO SEQUENCE NETWORK AND ATTENTION

Sequence to Sequence models

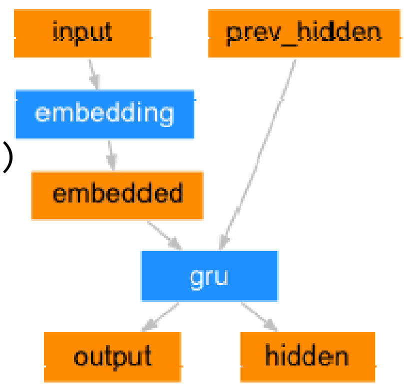
- A vanilla sequence to sequence model presented in https://arxiv.org/abs/1409.3215, https://arxiv.org/abs/1406.1078 consists of using a RNN such as an LSTM or GRU to encode a sequence of words or characters in a *source* language into a fixed length vector representation and then decoding from that representation using another RNN in the *target* language.

- Sequence to Sequence Learning with Neural Networks: https://arxiv.org/abs/1409.3215
- Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation: https://arxiv.org/abs/1406.1078

Sequence to Sequence models

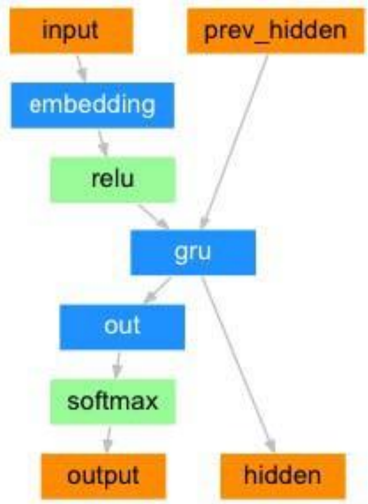



```python
class EncoderRNN(nn.Module):
    def___init__(self, input_size, hidden_size):
        super(EncoderRNN, self).__init__()
        self.hidden_size = hidden_size
        self.embedding = nn.Embedding(input_size, hidden_size)
        self.gru = nn.GRU(hidden_size, hidden_size)

    def forward(self, input, hidden):
        embedded = self.embedding(input).view(1, 1, -1)
        output = embedded
        output, hidden = self.gru(output, hidden)
        return output, hidden

    def initHidden(self):
        return torch.zeros(1, 1, self.hidden_size, device=device)
```



https://pytorch.org/tutorials/intermediate/seq2seq_translation_tutorial.html

```python
class DecoderRNN(nn.Module):
    def___init__(self, hidden_size, output_size):
        super(DecoderRNN, self).__init__()
        self.hidden_size = hidden_size

        self.embedding = nn.Embedding(output_size, hidden_size)
        self.gru = nn.GRU(hidden_size, hidden_size)
        self.out = nn.Linear(hidden_size, output_size)
        self.softmax = nn.LogSoftmax(dim=1)

    def forward(self, input, hidden):
        output = self.embedding(input).view(1, 1, -1)
        output = F.relu(output)
        output, hidden = self.gru(output, hidden)
        output = self.softmax(self.out(output[0]))
        return output, hidden

    def initHidden(self):
        return torch.zeros(1, 1, self.hidden_size, device=device)
```




```python
def train(src, target):
    …
    encoder_hidden = encoder.init_hidden()
    encoder_outputs, encoder_hidden = encoder(src_var, encoder_hidden)
    hidden = encoder_hidden
    loss = 0
    for c in range(len(target_var)):
        token = target_var[c - 1] if c else str2tensor(SOS_token)
        output, hidden = decoder(token, hidden)
        loss += criterion(output, target_var[c])
encoder.zero_grad()
    decoder.zero_grad()
    loss.backward()
    optimizer.step()
    return loss.data[0] / len(target_var)

encoder = sm.EncoderRNN(N_CHARS, HIDDEN_SIZE, N_LAYERS)
decoder = sm.DecoderRNN(HIDDEN_SIZE, N_CHARS, N_LAYERS)
for epoch in range(1, N_EPOCH + 1):
    for i, (srcs, targets) in enumerate(train_loader):
        train_loss = train(srcs[0], targets[0])  # Batch is 1
```

Full implementation:
https://github.com/hunkim/PyTorchZeroToAll/blob/master/14_1_seq2seq.py

Sequence to Sequence models

- In the picture above, “Le”, “chat” and “bleu” words are fed into an encoder, and after a special signal (not shown) the decoder starts producing a translated sentence.
- The decoder keeps generating words until a special end of sentence token is produced. Here, the h vectors represent the internal state of the encoder.
- If you look closely, you can see that the decoder is supposed to generate a translation solely based on the last hidden state (h2 above) from the encoder.
- This h2 vector must encode everything we need to know about the source sentence. It must fully capture its meaning.

Sequence to Sequence models

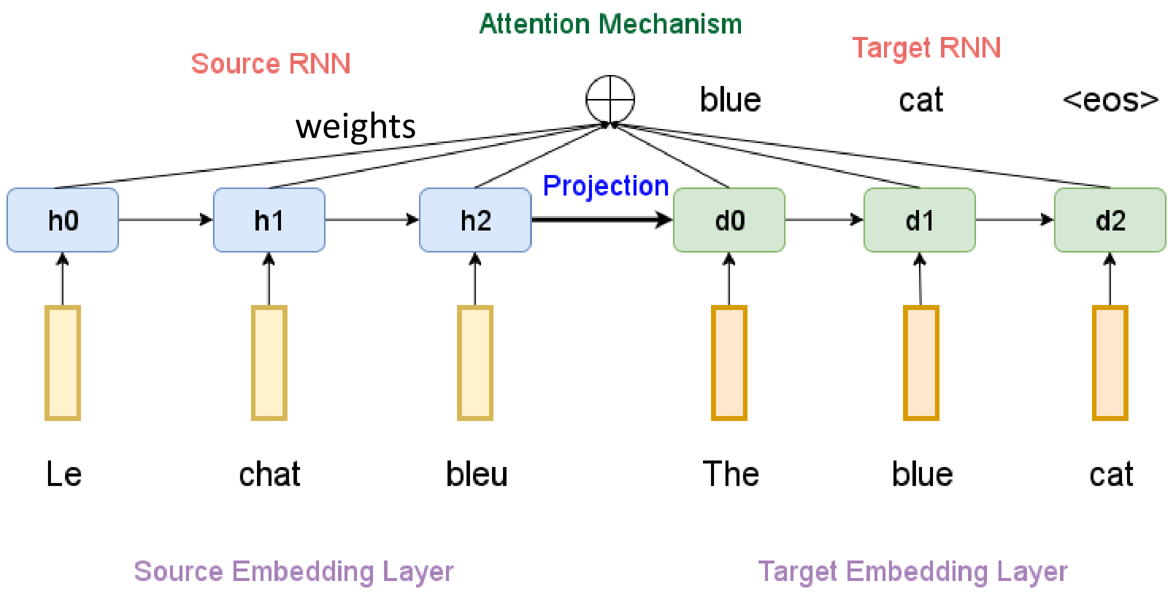
- It seems unreasonable to assume that we can encode all information about a potentially very long sentence into a single vector and then have the decoder produce a good translation based on only that.
- With an attention mechanism we no longer try encode the full source sentence into a fixed-length vector. Rather, we allow the decoder to “attend” to different parts of the source sentence at each step of the output generation.
- Importantly, we let the model **learn** what to attend to based on the input sentence and what it has produced so far. So, in languages that are pretty well aligned (like English and German) the decoder would probably choose to attend to things sequentially.
- Attending to the first word when producing the first English word, and so on.

Attention mechanism

- An extension of sequence to sequence models that incorporate an attention mechanism was presented in https://arxiv.org/abs/1409.0473 that uses information from the RNN hidden states in the source language at each time step in the decoder RNN.
- This attention mechanism significantly improves performance on tasks like machine translation.
- A few variants of the attention model for the task of machine translation have been presented in https://arxiv.org/abs/1508.04025.

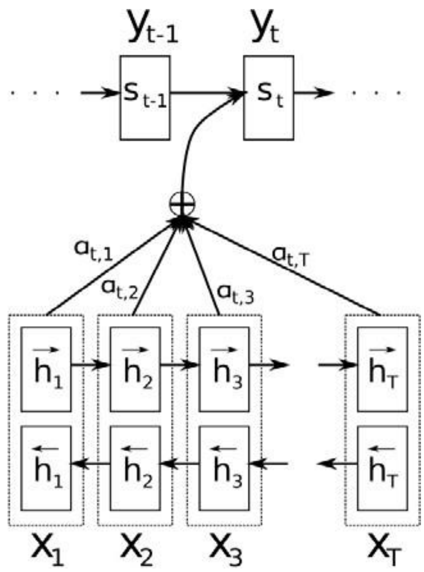
- Neural Machine Translation by Jointly Learning to Align and Translate: https://arxiv.org/abs/1409.0473
- Effective Approaches to Attention-based Neural Machine Translation: https://arxiv.org/abs/1508.04025

Attention mechanism



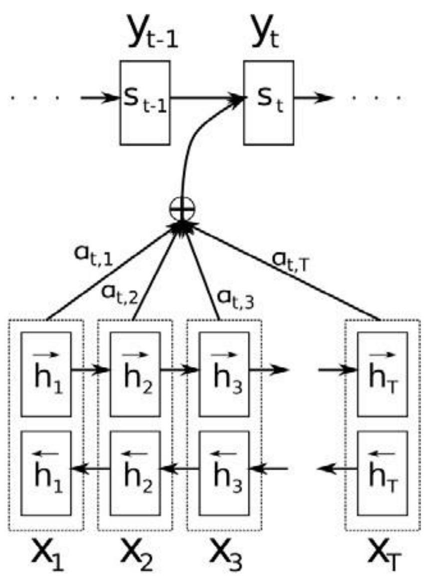
Attention mechanism

- The y's are our translated words produced by the decoder, and the x's are our source sentence words.
- The illustration uses a bidirectional RNN, but that's not important and you can just ignore the inverse direction.
- The important part is that each decoder output word y_t now depends on a weighted combination of all the input states, not just the last state.



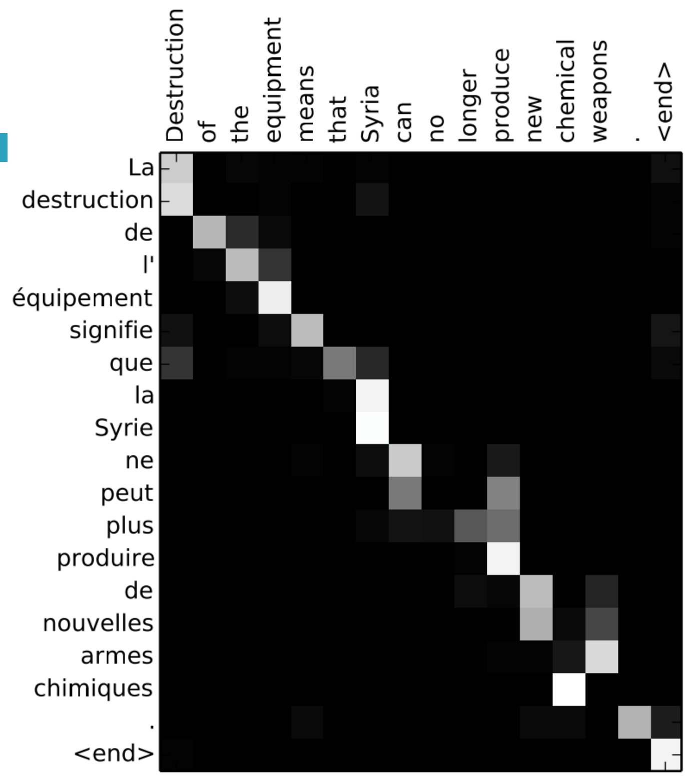
Attention mechanism

- The a's are weights that define in how much of each input state should be considered for each output.
- So, if $a_{3,2}$ is a large number, this would mean that the decoder pays a lot of attention to the second state in the source sentence while producing the third word of the target sentence.
- The a's are typically normalized to sum to 1.



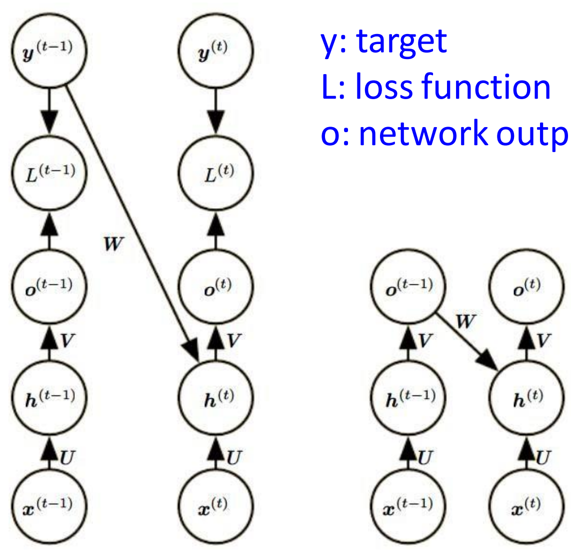
Attention mechanism

- A big advantage of attention is that it gives us the ability to interpret and visualize what the model is doing.
- For example, by visualizing the attention weight matrix ***a*** when a sentence is translated, we can understand how the model is translating.



Teacher Forcing

- Teacher forcing is a training technique that is applicable to RNNs that have connections from their output to their hidden states at the next time step.
- (Left)At train time, we feed the correct (target) output y(t) drawn from the train set as input to h(t+1).
- (Right)When the model is deployed, the true output is generally not known. In this case, we approximate the correct output y(t) with the model's output o(t), and feed the output back into the model.



Train time

Test time

y: target
L: loss function
o: network output

```python
class AttnDecoderRNN(nn.Module):
    def __init__(self, hidden_size, output_size, dropout_p=0.1, max_length=MAX_LENGTH):
        super(AttnDecoderRNN, self).__init__()
        self.hidden_size = hidden_size
        self.output_size = output_size
        self.dropout_p = dropout_p
        self.max_length = max_length

        self.embedding = nn.Embedding(self.output_size, self.hidden_size)
        self.attn = nn.Linear(self.hidden_size * 2, self.max_length)
        self.attn_combine = nn.Linear(self.hidden_size * 2, self.hidden_size)
        self.dropout = nn.Dropout(self.dropout_p)
        self.gru = nn.GRU(self.hidden_size, self.hidden_size)
        self.out = nn.Linear(self.hidden_size, self.output_size)
```

```python
    def forward(self, input, hidden, encoder_outputs):
        embedded = self.embedding(input).view(1, 1, -1)
        embedded = self.dropout(embedded)

        attn_weights = F.softmax(
            self.attn(torch.cat((embedded[0], hidden[0]), 1)), dim=1)
        attn_applied = torch.bmm(attn_weights.unsqueeze(0),
                                 encoder_outputs.unsqueeze(0))

        output = torch.cat((embedded[0], attn_applied[0]), 1)
        output = self.attn_combine(output).unsqueeze(0)

        output = F.relu(output)
        output, hidden = self.gru(output, hidden)

        output = F.log_softmax(self.out(output[0]), dim=1)
        return output, hidden, attn_weights
```

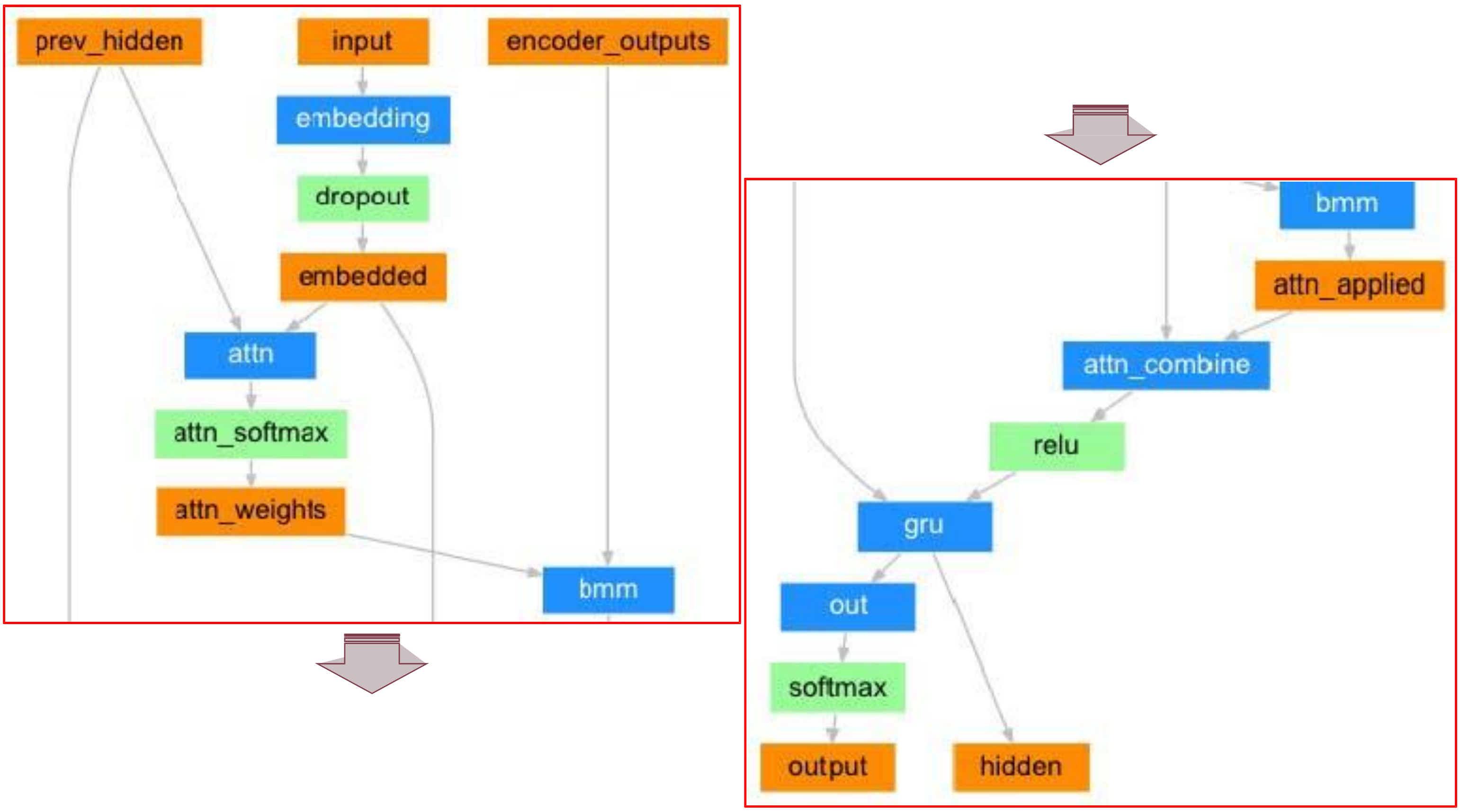
```
    def initHidden(self):
        return torch.zeros(1, 1, self.hidden_size, device=device)

hidden_size = 256
encoder1 = EncoderRNN(input_lang.n_words, hidden_size).to(device)
attn_decoder1 = AttnDecoderRNN(hidden_size, output_lang.n_words,
   dropout_p=0.1).to(device)

trainIters(encoder1, attn_decoder1, 75000, print_every=5000)
```

bmm performs a batch matrix-matrix product

If batch1 is a (b×n×m), batch2 is a (b×m×p), out will be a (b×n×p) tensor.

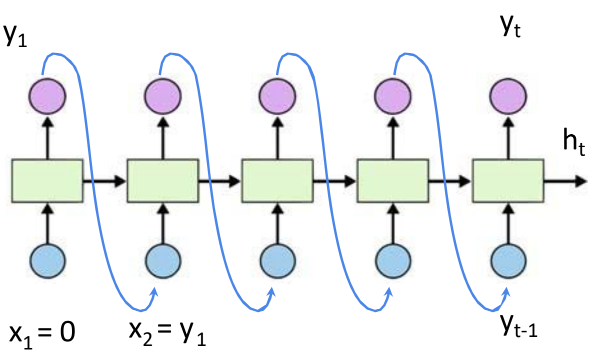
$B1_{nxm}$ x $B2_{mxp}$ = Out_{nxp}

Full implementation:
https://pytorch.org/tutorials/intermediate/seq2seq_translation_tutorial.html



```
def train(...)
    ...
    use_teacher_forcing = True if random.random() < teacher_forcing_ratio else False
    if use_teacher_forcing:
        # Teacher forcing: Feed the target as the next input
        for di in range(target_length):
            decoder_output, decoder_hidden, decoder_attention = decoder(
                decoder_input, decoder_hidden, encoder_outputs)
            loss += criterion(decoder_output, target_tensor[di])
            decoder_input = target_tensor[di]  # Teacher forcing
    else:
        # Without teacher forcing: use its own predictions as the next input
        for di in range(target_length):
            decoder_output, decoder_hidden, decoder_attention = decoder(
                decoder_input, decoder_hidden, encoder_outputs)
            topv, topi = decoder_output.topk(1)
            decoder_input = topi.squeeze().detach()  # detach from history as input

            loss += criterion(decoder_output, target_tensor[di])
            if decoder_input.item() == EOS_token:
                break
    ...
```

Without teacher forcing

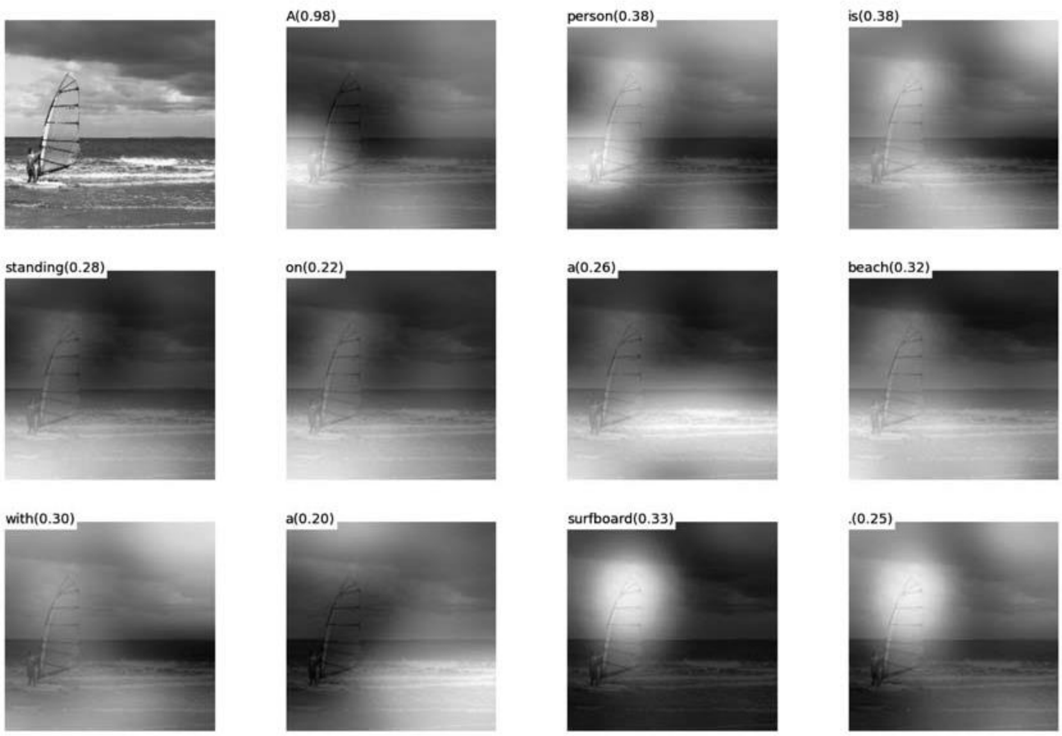


No Teacher Forcing
(more natural)

```
def train(line):
    input = str2tensor(line[:-1])
    target = str2tensor(line[1:])

    hidden = decoder.init_hidden()
    decoder_in = input[0]
    loss = 0

    for c in range(len(input)):
        output, hidden = decoder(decoder_in, hidden)
        loss += criterion(output, target[c])
        decoder_in = output.max(1)[1]

    decoder.zero_grad()
    loss.backward()
    decoder_optimizer.step()

    return loss.data[0] / len(input)
```

Attention beyond Machine Translation

- The same attention mechanism from above can be applied to any recurrent model.
  - Show, Attend and Tell: Neural Image Caption Generation with Visual Attention, https://arxiv.org/abs/1502.03044
  - The authors apply attention mechanisms to the problem of generating image descriptions. They use a Convolutional Neural Network to “encode” the image, and a Recurrent Neural Network with attention mechanisms to generate a description.
  - By visualizing the attention weights (just like in the translation example), we interpret what the model is looking at while generating a word:



(b) A person is standing on a beach with a surfboard.

Reference

- Attention and Memory in Deep Learning and NLP
  - http://www.wildml.com/2016/01/attention-and-memory-in-deep-learning-and-nlp/
- Code
  - Translation with a Sequence to Sequence Network and Attention
    - https://pytorch.org/tutorials/intermediate/seq2seq_translation_tutorial.html
  - Sequence to Sequence models:
    - https://github.com/MaximumEntropy/Seq2Seq-PyTorch