Sequence-to-Sequence Translation Demo

Adapted from pytorch tutorial
https://pytorch.org/tutorials/intermediate/seq2seq_translation_tutorial.html

Originally accessed on 03-28-2023

David I. Inouye

NLP From Scratch: Translation with a Sequence to Sequence Network and Attention

Original Author: Sean Robertson

- We will teach a neural network to translate from French to English
- Uses a **sequence-to-sequence (seq2seq)** network with **attention mechanism**
- Two RNNs work together: encoder condenses input, decoder unfolds to output

```
[KEY: > input, = target, < output]

> il est en train de peindre un tableau .
= he is painting a picture .
< he is painting a picture .

> pourquoi ne pas essayer ce vin delicieux ?
= why not try that delicious wine ?
< why not try that delicious wine ?
```

Preliminary: Torch Embedding Layers

- Embedding layers take torch tensors of int type and return vectors for each int
- Essentially, they are "learnable" lookup tables
- Equivalent to creating one-hot vectors and multiplying by a learnable embedding matrix

If $W \in \mathbb{R}^{V \times H}$ represents an embedding matrix where H is the embedding dimension and V is the vocabulary size:

$$\text{embedding}(i) = w_i = W^T \text{OneHot(i)}$$

- **Key property**: The hidden size is appended to the shape of the indices input.
  - Example: Input shape `(N,)` (N indices) → Output shape `(N, H)` (N vectors of size H)

Preliminary: Torch Embedding Layers

```python
import torch
import torch.nn as nn
vocab_size = 5
embedding_size = 2
embedding = nn.Embedding(vocab_size, embedding_size)
input_idx = torch.tensor([1,2,1])
input_embed = embedding(input_idx)
print('Input indices')
print(input_idx.unsqueeze(0).T)
print('After embedding (notice that has grad_fn since embeddings are learnable)')
print(input_embed)

print('\nThis is equivalent to one hot encoding and then matrix mulectures-final/11-demo1-seq2seq_translation_tutorial.qmdltipli
def one_hot(idx, vocab_size):
    out = torch.zeros((*idx.size(), vocab_size))
    for i, j in enumerate(idx):
        out[i, j] = 1
    return out
print('One hot encoding')
print(one_hot(input_idx, vocab_size))
print('W.T times one-hot encodings (batched) and then converted to row vectors')
print((embedding.weight.T @ one_hot(input_idx, vocab_size).T).T)
```

```
Input indices
tensor([[1],
        [2],
        [1]])
After embedding (notice that has grad_fn since embeddings are learnable)
tensor([[-0.1256,  0.8497],
```

Setup: Import Required Libraries

```
%matplotlib inline
from __future__ import unicode_literals, print_function, division
from io import open
import unicodedata
import string
import re
import random

import torch
import torch.nn as nn
from torch import optim
import torch.nn.functional as F

# Check if MPS is available
if torch.cuda.is_available():
    device = torch.device("cuda")
    print("Using CUDA device (GPU)")
# Could use Mac MPS device if available
# elif torch.backends.mps.is_available():
#     device = torch.device("mps")
#     print("Using MPS device (GPU)")
else:
    device = torch.device("cpu")
    print("MPS device not found, using CPU")
```

```
MPS device not found, using CPU
```

Loading Data Files

- The data for this project is many thousands of English to French translation pairs
- Downloaded from tatoeba.org
- Pre-split language pairs available at manythings.org/anki
- The file is a tab-separated list of translation pairs:

```
I am cold.    J'ai froid.
```

Note: Download the data from here and extract it to the current directory.

▶ Code

```
Data already exists, skipping download.
```

Word Representation

- We represent each word as a one-hot vector (giant vector of zeros except for a single one)
- Unlike character-level (dozens of characters), we have many more words
- So the encoding vector is much larger
- We cheat a bit and trim the data to only use a few thousand words per language

The Lang Class

We need a helper class to track word↔index mappings and word counts:

```
SOS_token = 0
EOS_token = 1

class Lang:
    def __init__(self, name):
        self.name = name
        self.word2index = {}
        self.word2count = {}
        self.index2word = {0: "SOS", 1: "EOS"}
        self.n_words = 2  # Count SOS and EOS

    def addSentence(self, sentence):
        for word in sentence.split(' '):
            self.addWord(word)

    def addWord(self, word):
        if word not in self.word2index:
            self.word2index[word] = self.n_words
            self.word2count[word] = 1
            self.index2word[self.n_words] = word
            self.n_words += 1
        else:
            self.word2count[word] += 1
```

Data Preprocessing: Normalization

The files are all in Unicode. To simplify:

- Turn Unicode characters to ASCII
- Make everything lowercase
- Trim most punctuation

```python
# Turn a Unicode string to plain ASCII, thanks to
# https://stackoverflow.com/a/518232/2809427
def unicodeToAscii(s):
    return ''.join(
        c for c in unicodedata.normalize('NFD', s)
        if unicodedata.category(c) != 'Mn'
    )

# Lowercase, trim, and remove non-letter characters


def normalizeString(s):
    s = unicodeToAscii(s.lower().strip())
    s = re.sub(r"([.!?])", r" \1", s)
    s = re.sub(r"[^a-zA-Z.!?]+", r" ", s)
    return s
```

Data Preprocessing: Reading Files

To read the data file: - Split the file into lines - Split lines into pairs - Optionally reverse pairs for translating Other Language → English

```
def readLangs(lang1, lang2, reverse=False):
    print("Reading lines...")

    # Read the file and split into lines
    lines = open('data/%s-%s.txt' % (lang1, lang2), encoding=
        read().strip().split('\n')

    # Split every line into pairs and normalize
    pairs = [[normalizeString(s) for s in l.split('\t')] for

    # Reverse pairs, make Lang instances
    if reverse:
        pairs = [list(reversed(p)) for p in pairs]
        input_lang = Lang(lang2)
        output_lang = Lang(lang1)
    else:
        input_lang = Lang(lang1)
        output_lang = Lang(lang2)

    return input_lang, output_lang, pairs
```

Data Preprocessing: Filtering

- Since there are many example sentences, we'll trim the dataset
- We want only relatively short and simple sentences
- Maximum length: 10 words (including ending punctuation)
- Filter to sentences that translate to the form "I am" or "He is" etc.

```python
MAX_LENGTH = 10

eng_prefixes = (
    "i am ", "i m ",
    "he is", "he s ",
    "she is", "she s ",
    "you are", "you re ",
    "we are", "we re ",
    "they are", "they re "
)

def filterPair(p):
    return len(p[0].split(' ')) < MAX_LENGTH and \
        len(p[1].split(' ')) < MAX_LENGTH and \
        p[1].startswith(eng_prefixes)

def filterPairs(pairs):
    return [pair for pair in pairs if filterPair(pair)]
```

Data Preprocessing: Putting It All Together

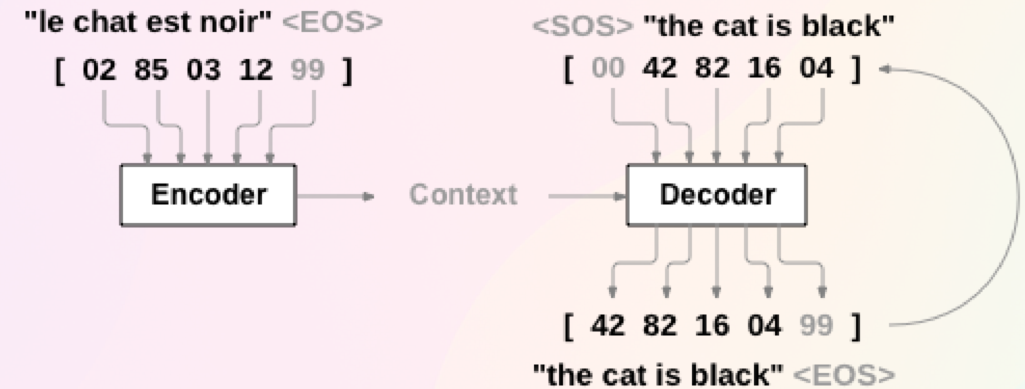
The full process for preparing the data: 1. Read text file and split into lines, split lines into pairs 2. Normalize text, filter by length and content 3. Make word lists from sentences in pairs

```python
def prepareData(lang1, lang2, reverse=False):
    input_lang, output_lang, pairs = readLangs(lang1, lang2,
    print("Read %s sentence pairs" % len(pairs))
    pairs = filterPairs(pairs)
    print("Trimmed to %s sentence pairs" % len(pairs))
    print("Counting words...")
    for pair in pairs:
        input_lang.addSentence(pair[0])
        output_lang.addSentence(pair[1])
    print("Counted words:")
    print(input_lang.name, input_lang.n_words)
    print(output_lang.name, output_lang.n_words)
    return input_lang, output_lang, pairs

input_lang, output_lang, pairs = prepareData('eng', 'fra', Tr
print(random.choice(pairs))
```

```
Reading lines...
Read 135842 sentence pairs
Trimmed to 10599 sentence pairs
Counting words...
Counted words:
fra 4345
eng 2803
['vous etes fort elegant .', 'you re very sophisticated .']
```

The Seq2Seq RNN Model

- A **Recurrent Neural Network (RNN)** operates on a sequence and uses its own output as input for subsequent steps
- A **Sequence to Sequence network** (seq2seq or Encoder-Decoder) consists of two RNNs:
  - **Encoder**: reads input sequence and outputs a single vector
  - **Decoder**: reads that vector to produce output sequence



Why Seq2Seq for Translation?

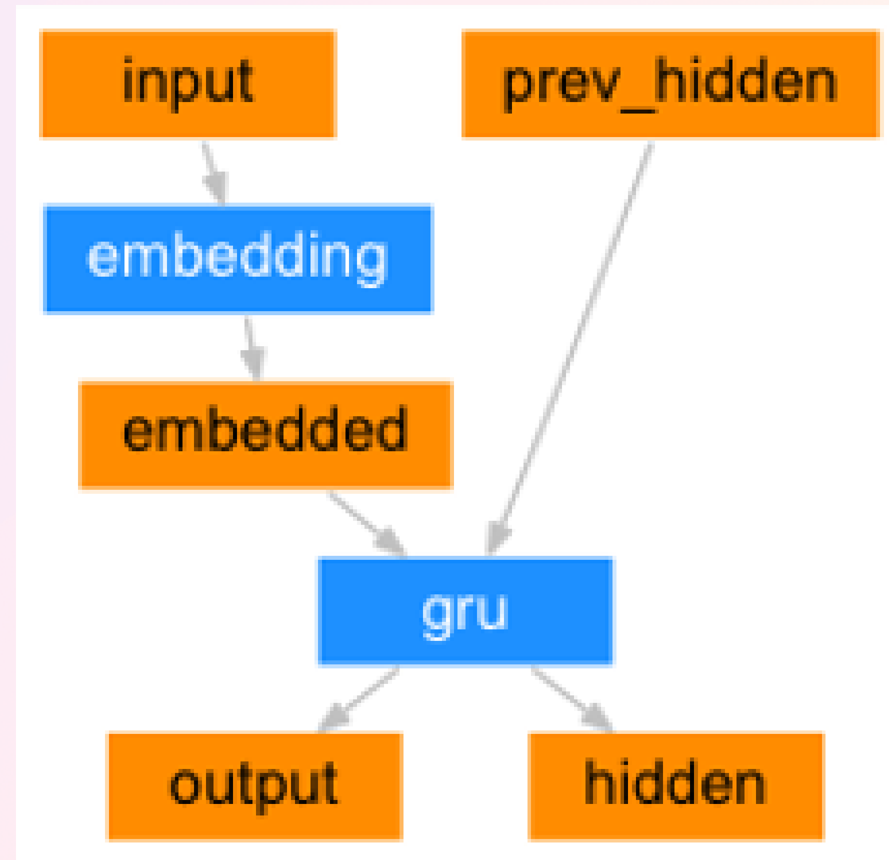
- Unlike sequence prediction with a single RNN (where every input corresponds to an output), seq2seq frees us from sequence length and order
- Ideal for translation between two languages

Example: “Je ne suis pas le chat noir” → “I am not the black cat”

- Most words have direct translation but in different orders (“chat noir” vs “black cat”)
- The “ne/pas” construction adds complexity (one more word in input)
- The encoder creates a single vector encoding the “meaning” of the input sequence

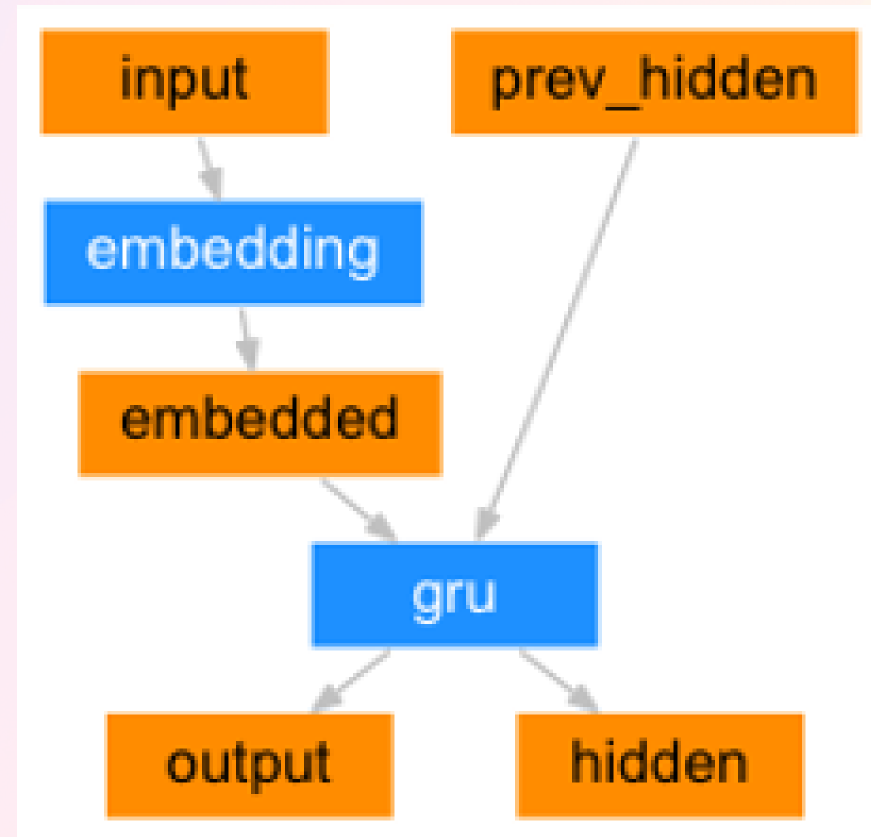
The Encoder

- The encoder is an RNN that outputs some value for every word from the input sentence
- For every input word:
  - Outputs a vector and a hidden state
  - Uses the hidden state for the next input word



The Encoder

```
1  class EncoderRNN(nn.Module):
2      def __init__(self, input_size, hidden_size):
3          super(EncoderRNN, self).__init__()
4          self.hidden_size = hidden_size
5
6          self.embedding = nn.Embedding(input_size, hidden_size
7          self.gru = nn.GRU(hidden_size, hidden_size)
8
9      def forward(self, input, hidden):
10         embedded = self.embedding(input).view(1, 1, -1)
11         output = embedded
12         output, hidden = self.gru(output, hidden)
13         return output, hidden
14
15     def initHidden(self):
16         return torch.zeros(1, 1, self.hidden_size, device=dev
```



The Decoder: Overview

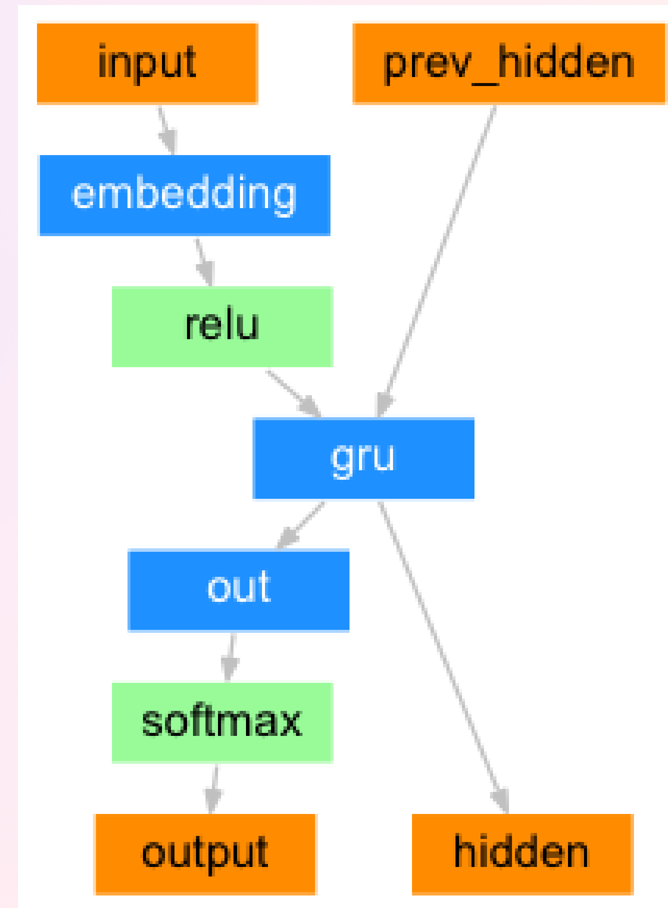
The decoder is another RNN that takes the encoder output vector(s) and outputs a sequence of words to create the translation.

Two approaches:

1. **Simple Decoder**: Uses only the last output of the encoder
2. **Attention Decoder**: Can focus on different parts of the encoder outputs (we'll use this!)

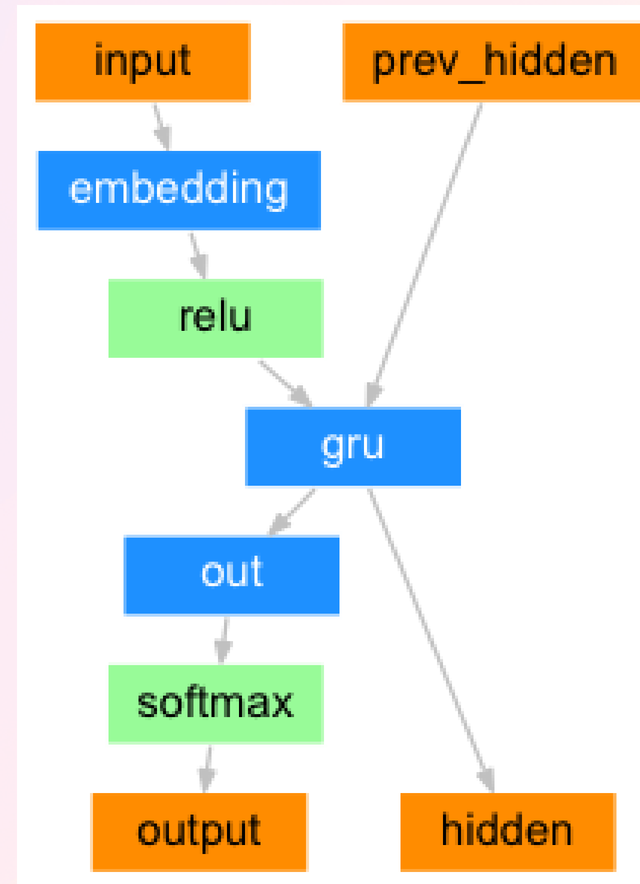
Simple Decoder

- Uses only last output of the encoder as the **context vector**
- This context vector is the initial hidden state of the decoder
- At every decoding step:
  - Given an input token and hidden state
  - Initial input token is `<SOS>`
  - First hidden state is the context vector



Simple Decoder

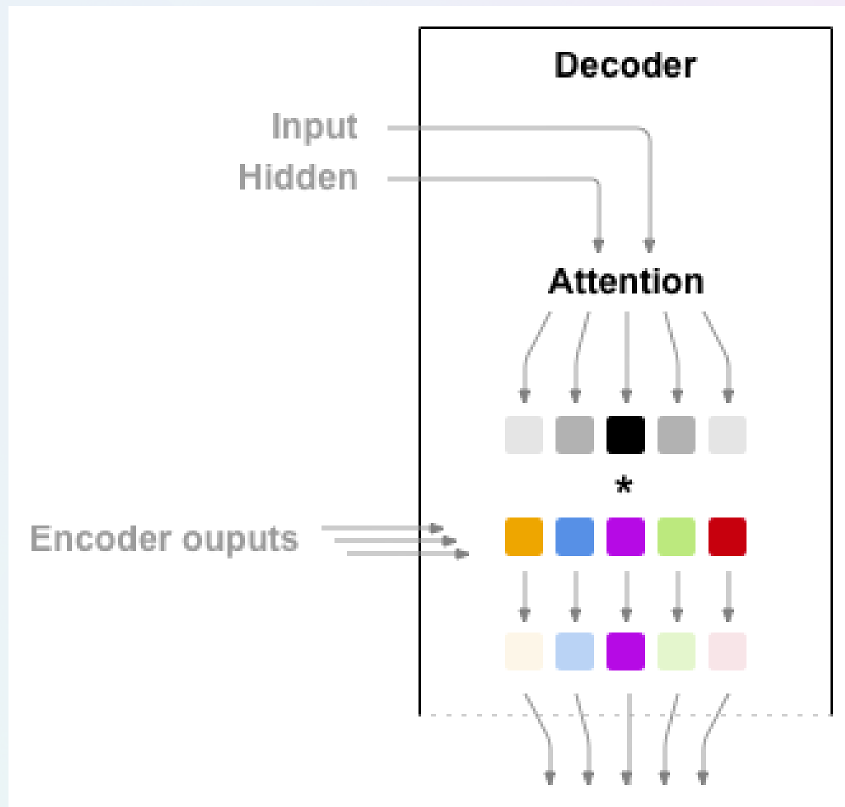
```
1  class DecoderRNN(nn.Module):
2      def __init__(self, hidden_size, output_size):
3          super(DecoderRNN, self).__init__()
4          self.hidden_size = hidden_size
5
6          self.embedding = nn.Embedding(output_size, hidden_siz
7          self.gru = nn.GRU(hidden_size, hidden_size)
8          self.out = nn.Linear(hidden_size, output_size)
9          self.softmax = nn.LogSoftmax(dim=1)
10
11     def forward(self, input, hidden):
12         # `input` is a single long (int64) value (representin
13         # `hidden` is (1, 1, 256) (L, N, H) (1 word, 1 sample
14         output = self.embedding(input).view(1, 1, -1)
15         output = F.relu(output)
16         output, hidden = self.gru(output, hidden)
17
18         # Convert to output log probability (of different siz
19         #  (1, 4345) (N, V) where V is vocab size of output
20         output = self.softmax(self.out(output[0]))
21         return output, hidden
22
23     def initHidden(self):
24         return torch.zeros(1, 1, self.hidden_size, device=dev
```



Attention Decoder: Motivation

- If only the context vector is passed between encoder and decoder, that single vector carries the burden of encoding the entire sentence
- **Attention** allows the decoder to “focus” on different parts of the encoder’s outputs for every step
- First, calculate a set of **attention weights**
- Multiply these by the encoder output vectors to create a weighted combination
- Result contains information about that specific part of the input sequence

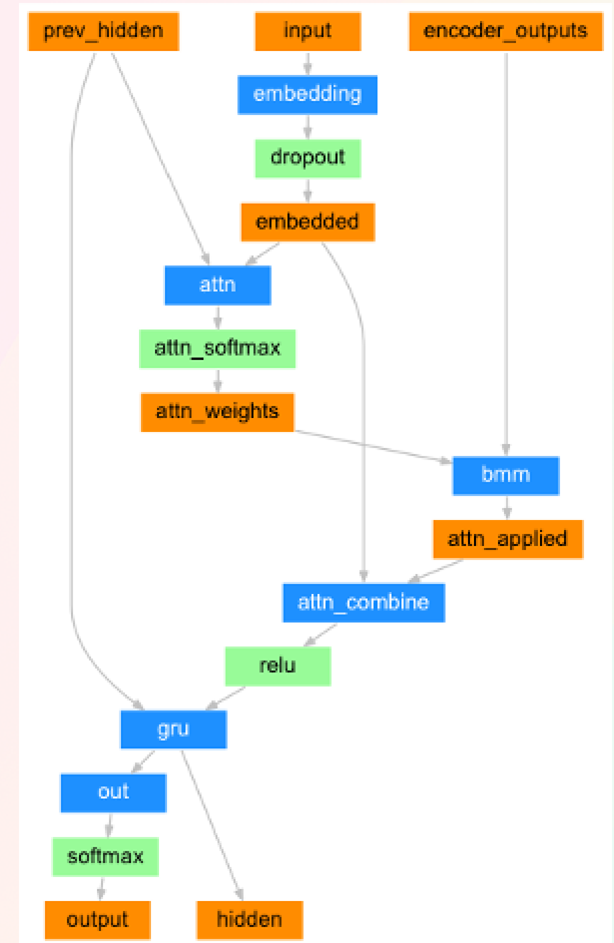
Attention Decoder: How It Works



- Attention weights calculated with feed-forward layer `attn`
- Uses decoder's input and hidden state as inputs
- Must choose a maximum sentence length for the layer
- Sentences of max length use all attention weights
- Shorter sentences use only the first few

Attention Decoder

```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)

    def forward(self, input, hidden, encoder_outputs):
        # `input` is a single long (int64) value (representing a single word)
        # `hidden` is (1, 1, 256) (L, N, H) (1 word, 1 sample in batch, hidden size)
        # `encoder_outputs` (10, 256) (L_in, H) (total length of input, hidden size)

        # Get input embedding
        # (1, 1, 256) (L, N, H)
        embedded = self.embedding(input).view(1, 1, -1)
        embedded = self.dropout(embedded) # Add random 0s

        # Compute attention weights
```



Note: There are other forms of attention that work around the length limitation by using a relative

Training: Preparing Training Data

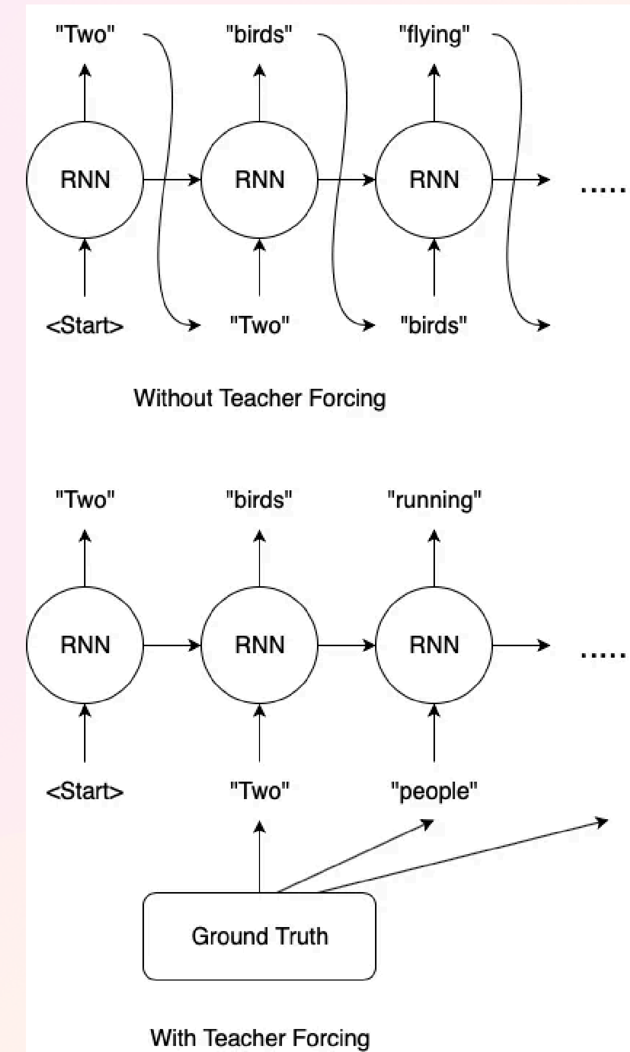
To train, for each pair we need:

- **Input tensor**: indexes of words in the input sentence
- **Target tensor**: indexes of words in the target sentence
- Append the EOS token to both sequences

```
def indexesFromSentence(lang, sentence):
    return [lang.word2index[word] for word in sentence.split(' ')]

def tensorFromSentence(lang, sentence):
    indexes = indexesFromSentence(lang, sentence)
    indexes.append(EOS_token)
    return torch.tensor(indexes, dtype=torch.long, device=device).view(-1, 1)

def tensorsFromPair(pair):
    input_tensor = tensorFromSentence(input_lang, pair[0])
    target_tensor = tensorFromSentence(output_lang, pair[1])
    return (input_tensor, target_tensor)
```

Training: Teacher Forcing

- **Teacher forcing**: using the real target outputs as each next input (instead of the decoder's guess)
- Causes faster convergence but may exhibit instability when deployed
- Teacher-forced networks may read with coherent grammar but wander from correct translation
- It learns to represent output grammar but may not properly learn how to create the sentence from the translation
- We randomly choose to use teacher forcing with probability `teacher_forcing_ratio`



Source: https://medium.com/data-science/what-is-teacher-forcing-3da6217fed1c, Accessed on 10-27-25

Training: The Training Function

```
1  teacher_forcing_ratio = 0.5
2  def train(input_tensor, target_tensor, encoder, decoder, encoder_optimizer, decoder_optimizer, criterion, max_length=MAX_LENGTH)
3      encoder_hidden = encoder.initHidden()
4  
5      encoder_optimizer.zero_grad()
6      decoder_optimizer.zero_grad()
7  
8      input_length = input_tensor.size(0)
9      target_length = target_tensor.size(0)
10 
11     # Need to initialize up to max_length for attention
12     encoder_outputs = torch.zeros(max_length, encoder.hidden_size, device=device)
13 
14     loss = 0
15 
16     # Loop through input with encoder saving outputs
17     #  for use in decoder step
18     for ei in range(input_length):
19         encoder_output, encoder_hidden = encoder(
20             input_tensor[ei], encoder_hidden)
21         encoder_outputs[ei] = encoder_output[0, 0]
22 
23     # Initialize decoder input with start of sentence SOS token
24     decoder_input = torch.tensor([[SOS_token]], device=device)
25 
26     # Copy the last encoder_hidden value to initialize the decoder_hidden
27     decoder_hidden = encoder_hidden
```

Training: Helper Functions for Timing

```python
import time
import math

def asMinutes(s):
    m = math.floor(s / 60)
    s -= m * 60
    return '%dm %ds' % (m, s)

def timeSince(since, percent):
    now = time.time()
    s = now - since
    es = s / (percent)
    rs = es - s
    return '%s (- %s)' % (asMinutes(s), asMinutes(rs))
```

Training: Main Training Loop

```python
def trainIters(encoder, decoder, n_iters, print_every=1000, plot_every=100, learning_rate=0.01):
    start = time.time()
    plot_losses = []
    print_loss_total = 0  # Reset every print_every
    plot_loss_total = 0  # Reset every plot_every

    encoder_optimizer = optim.SGD(encoder.parameters(), lr=learning_rate)
    decoder_optimizer = optim.SGD(decoder.parameters(), lr=learning_rate)
    training_pairs = [tensorsFromPair(random.choice(pairs))
                      for i in range(n_iters)]
    criterion = nn.NLLLoss()

    for iter in range(1, n_iters + 1):
        training_pair = training_pairs[iter - 1]
        input_tensor = training_pair[0]
        target_tensor = training_pair[1]

        loss = train(input_tensor, target_tensor, encoder,
                     decoder, encoder_optimizer, decoder_optimizer, criterion)
        print_loss_total += loss
        plot_loss_total += loss

        if iter % print_every == 0:
            print_loss_avg = print_loss_total / print_every
            print_loss_total = 0
            print('%s (%d %d%%) %.4f' % (timeSince(start, iter / n_iters),
                                         iter, iter / n_iters * 100, print_loss_avg))
```

Training: Plotting Results

Plotting is done with matplotlib, using the array of loss values `plot_losses` saved while training.

```
import matplotlib.pyplot as plt
plt.switch_backend('agg')
import matplotlib.ticker as ticker
import numpy as np
%matplotlib inline


def showPlot(points):
    plt.figure()
    fig, ax = plt.subplots()
    # this locator puts ticks at regular intervals
    loc = ticker.MultipleLocator(base=0.2)
    ax.yaxis.set_major_locator(loc)
    plt.plot(points)
```

Evaluation

Evaluation is mostly the same as training, but:

- No targets, so we feed the decoder's predictions back to itself for each step
- Every time it predicts a word we add it to the output string
- If it predicts the EOS token we stop there
- We also store the decoder's attention outputs for display later

Evaluation Function

```python
def evaluate(encoder, decoder, sentence, max_length=MAX_LENGTH):
    with torch.no_grad():
        input_tensor = tensorFromSentence(input_lang, sentence)
        input_length = input_tensor.size()[0]
        encoder_hidden = encoder.initHidden()

        encoder_outputs = torch.zeros(max_length, encoder.hidden_size, device=device)

        for ei in range(input_length):
            encoder_output, encoder_hidden = encoder(input_tensor[ei],
                                                     encoder_hidden)
            encoder_outputs[ei] += encoder_output[0, 0]

        decoder_input = torch.tensor([[SOS_token]], device=device)  # SOS

        decoder_hidden = encoder_hidden

        decoded_words = []
        decoder_attentions = torch.zeros(max_length, max_length)

        for di in range(max_length):
            decoder_output, decoder_hidden, decoder_attention = decoder(
                decoder_input, decoder_hidden, encoder_outputs)
            decoder_attentions[di] = decoder_attention.data
            topv, topi = decoder_output.data.topk(1)
            if topi.item() == EOS_token:
```

Evaluating Random Sentences

We can evaluate random sentences from the training set and print out the input, target, and output to make some subjective quality judgements:

```
def evaluateRandomly(encoder, decoder, n=10):
    for i in range(n):
        pair = random.choice(pairs)
        print('>', pair[0])
        print('=', pair[1])
        output_words, attentions = evaluate(encoder, decoder, pair[0])
        output_sentence = ' '.join(output_words)
        print('<', output_sentence)
        print('')
```

Training and Evaluating

- After about 40 minutes on a MacBook CPU we'll get some reasonable results

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

if os.path.exists('encoder1.pth') and os.path.exists('attn_de
    # Load the models if not training
    print("Loading pretrained models rather than training")
    encoder1.load_state_dict(torch.load('encoder1.pth'))
    attn_decoder1.load_state_dict(torch.load('attn_decoder1.p
else:
    trainIters(encoder1, attn_decoder1, 75000, print_every=10
    # Save the models after training
    torch.save(encoder1.state_dict(), 'encoder1.pth')
    torch.save(attn_decoder1.state_dict(), 'attn_decoder1.pth
```

```
Loading pretrained models rather than training
```

Deployment: Running on Random Examples

```
1 evaluateRandomly(encoder1, attn_decoder1)
```

```
> je ne te crains plus .
= i m not scared of you anymore .
< i m not scared of you anymore . <EOS>

> je suis tres frustree .
= i m very frustrated .
< i m very frustrated . <EOS>

> je suis trop somnolent pour conduire .
= i m too sleepy to drive .
< i m too sleepy to drive . <EOS>

> vous etes tres religieuses n est ce pas ?
= you re very religious aren t you ?
< you re very religious aren t you ? <EOS>

> vous me faites marrer .
= you re amusing .
< you re amusing . <EOS>

> je suis une nouvelle etudiante .
= i m a new student .
< i m a new student . <EOS>

> il craint la mort .
= he is afraid of death .
< he is afraid of death . <EOS>
```

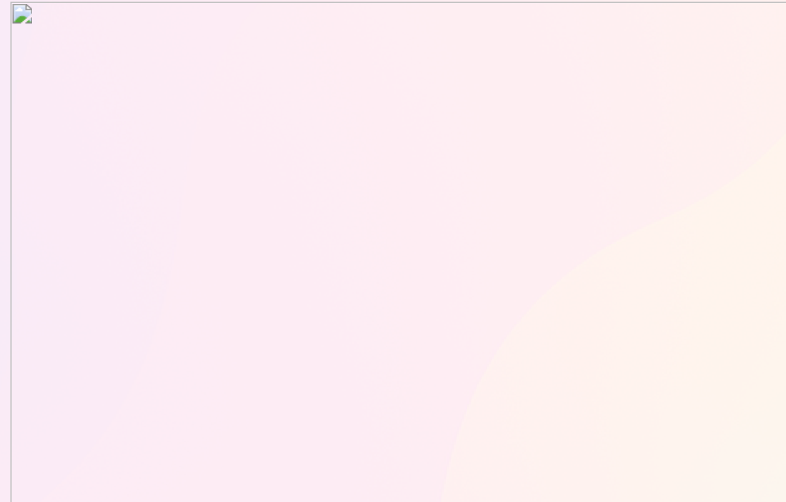
Visualizing Attention

A useful property of the attention mechanism is its highly interpretable outputs.

- Because it is used to weight specific encoder outputs of the input sequence, we can imagine looking where the network is focused most at each time step
- You could simply run `plt.matshow(attentions)` to see attention output displayed as a matrix
  - Columns are input steps
  - Rows are output steps

Visualizing Attention

```
%matplotlib inline
output_words, attentions = evaluate(
    encoder1, attn_decoder1, "je suis trop froid .")
plt.matshow(attentions.numpy())
plt.show()
```



Visualizing Attention: Better Display

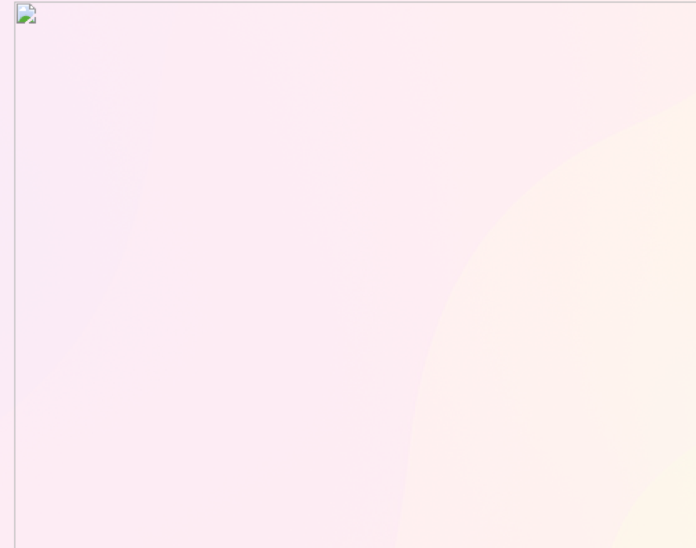
For a better viewing experience we add axes and labels:

```python
def showAttention(input_sentence, output_words, attentions):
    # Set up figure with colorbar
    fig = plt.figure()
    ax = fig.add_subplot(111)
    cax = ax.matshow(attentions.numpy(), cmap='bone')
    fig.colorbar(cax)

    # Set up axes
    ax.set_xticklabels([''] + input_sentence.split(' ') +
                       ['<EOS>'], rotation=90)
    ax.set_yticklabels([''] + output_words)

    # Show label at every tick
    ax.xaxis.set_major_locator(ticker.MultipleLocator(1))
    ax.yaxis.set_major_locator(ticker.MultipleLocator(1))

    plt.show()

def evaluateAndShowAttention(input_sentence):
    output_words, attentions = evaluate(
        encoder1, attn_decoder1, input_sentence)
    print('input =', input_sentence)
    print('output =', ' '.join(output_words))
    showAttention(input_sentence, output_words, attentions)
```

Visualizing Attention: Examples

```
1  %matplotlib inline
2  evaluateAndShowAttention("elle a cinq ans de moins que moi ."
3
4  evaluateAndShowAttention("elle est trop petit .")
5
6  evaluateAndShowAttention("je ne crains pas de mourir .")
7
8  evaluateAndShowAttention("c est un jeune directeur plein de t
```

```
input = elle a cinq ans de moins que moi .
output = she is five years older and younger . <EOS>
```



```
input = elle est trop petit .
output = she is too short . <EOS>
```