

# Sequence-to-Sequence Translation Demo

Adapted from pytorch tutorial

[https://pytorch.org/tutorials/intermediate/seq2seq\\_translation\\_tutorial.html](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 délicieux ?  
= 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)  $\rightarrow$  Output shape  $(N, H)$  (N vectors of size H)



# Preliminary: Torch Embedding Layers

```
1 import torch
2 import torch.nn as nn
3 vocab_size = 5
4 embedding_size = 2
5 embedding = nn.Embedding(vocab_size, embedding_size)
6 input_idx = torch.tensor([1,2,1])
7 input_embed = embedding(input_idx)
8 print('Input indices')
9 print(input_idx.unsqueeze(0).T)
10 print('After embedding (notice that has grad_fn since embeddings are learnable)')
11 print(input_embed)
12
13 print('\nThis is equivalent to one hot encoding and then matrix multiplication')
14 def one_hot(idx, vocab_size):
15     out = torch.zeros((*idx.size(), vocab_size))
16     for i, j in enumerate(idx):
17         out[i, j] = 1
18     return out
19 print('One hot encoding')
20 print(one_hot(input_idx, vocab_size))
21 print('W.T times one-hot encodings (batched) and then converted to row vectors')
22 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([[ 2.1795, -0.4219],
```

# Setup: Import Required Libraries

```
1 %matplotlib inline
2 from __future__ import unicode_literals, print_function, division
3 from io import open
4 import unicodedata
5 import string
6 import re
7 import random
8
9 import torch
10 import torch.nn as nn
11 from torch import optim
12 import torch.nn.functional as F
13
14 # Check if MPS is available
15 if torch.cuda.is_available():
16     device = torch.device("cuda")
17     print("Using CUDA device (GPU)")
18 # Could use Mac MPS device if available
19 # elif torch.backends.mps.is_available():
20 #     device = torch.device("mps")
21 #     print("Using MPS device (GPU)")
22 else:
23     device = torch.device("cpu")
24     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](https://tatoeba.org)
- Pre-split language pairs available at [manythings.org/anki](https://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:

```
1 SOS_token = 0
2 EOS_token = 1
3
4 class Lang:
5     def __init__(self, name):
6         self.name = name
7         self.word2index = {}
8         self.word2count = {}
9         self.index2word = {0: "SOS", 1: "EOS"}
10        self.n_words = 2 # Count SOS and EOS
11
12    def addSentence(self, sentence):
13        for word in sentence.split(' '):
14            self.addWord(word)
15
16    def addWord(self, word):
17        if word not in self.word2index:
18            self.word2index[word] = self.n_words
19            self.word2count[word] = 1
20            self.index2word[self.n_words] = word
21            self.n_words += 1
22        else:
23            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

```
1 # Turn a Unicode string to plain ASCII, thanks to
2 # https://stackoverflow.com/a/518232/2809427
3 def unicodeToAscii(s):
4     return ''.join(
5         c for c in unicodedata.normalize('NFD', s)
6         if unicodedata.category(c) != 'Mn'
7     )
8
9 # Lowercase, trim, and remove non-letter characters
10
11
12 def normalizeString(s):
13     s = unicodeToAscii(s.lower().strip())
14     s = re.sub(r"([.!?])", r" \1", s)
15     s = re.sub(r"^[a-zA-Z.!?]+", r" ", s)
16     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

```
1 def readLangs(lang1, lang2, reverse=False):
2     print("Reading lines...")
3
4     # Read the file and split into lines
5     lines = open('data/%s-%s.txt' % (lang1, lang2), encoding=
6                 read().strip().split('\n')
7
8     # Split every line into pairs and normalize
9     pairs = [[normalizeString(s) for s in l.split('\t')] for
10
11 # Reverse pairs, make Lang instances
12 if reverse:
13     pairs = [list(reversed(p)) for p in pairs]
14     input_lang = Lang(lang2)
15     output_lang = Lang(lang1)
16 else:
17     input_lang = Lang(lang1)
18     output_lang = Lang(lang2)
19
20 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.

```
1 MAX_LENGTH = 10
2
3 eng_prefixes = (
4     "i am ", "i m ",
5     "he is", "he s ",
6     "she is", "she s ",
7     "you are", "you re ",
8     "we are", "we re ",
9     "they are", "they re "
10 )
11
12 def filterPair(p):
13     return len(p[0].split(' ')) < MAX_LENGTH and \
14           len(p[1].split(' ')) < MAX_LENGTH and \
15           p[1].startswith(eng_prefixes)
16
17 def filterPairs(pairs):
18     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

```
1 def prepareData(lang1, lang2, reverse=False):
2     input_lang, output_lang, pairs = readLangs(lang1, lang2,
3     print("Read %s sentence pairs" % len(pairs))
4     pairs = filterPairs(pairs)
5     print("Trimmed to %s sentence pairs" % len(pairs))
6     print("Counting words...")
7     for pair in pairs:
8         input_lang.addSentence(pair[0])
9         output_lang.addSentence(pair[1])
10    print("Counted words:")
11    print(input_lang.name, input_lang.n_words)
12    print(output_lang.name, output_lang.n_words)
13    return input_lang, output_lang, pairs
14
15 input_lang, output_lang, pairs = prepareData('eng', 'fra', Tr
16 print(random.choice(pairs))
```

Reading lines...

Read 135842 sentence pairs

Trimmed to 10599 sentence pairs

Counting words...

Counted words:

fra 4345

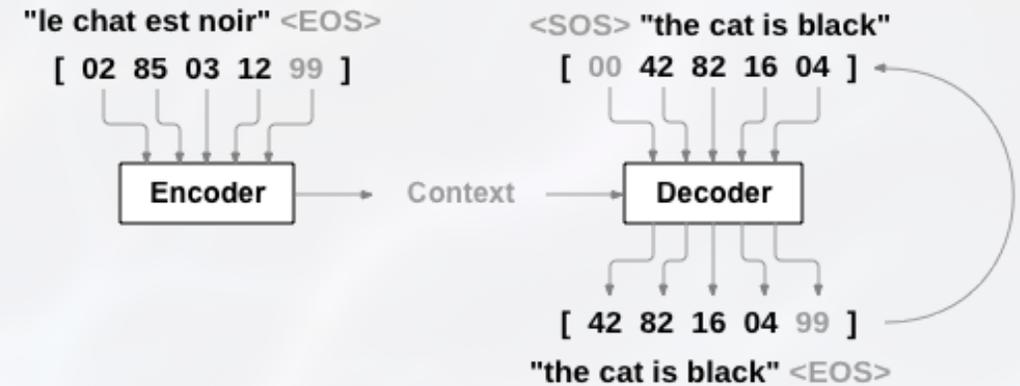
eng 2803

['tu n es pas l unique femme ici .', 'you re not the only woman here .']



# 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

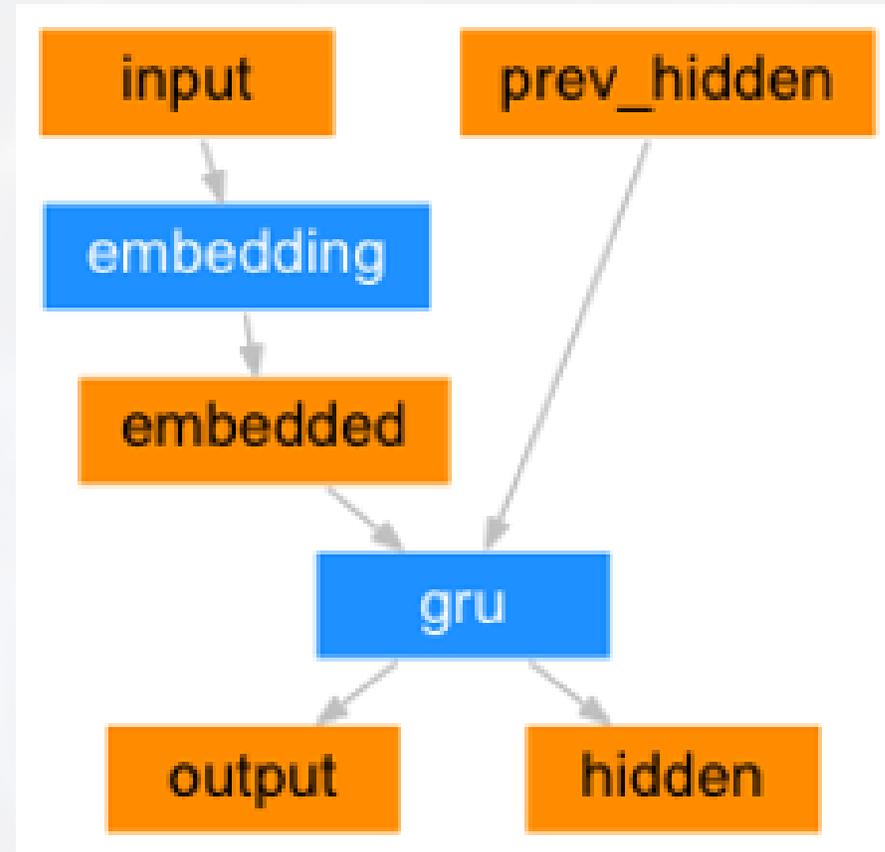
**Example:** “Je ne suis pas le chat noir” → “I am not the black cat”

- 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



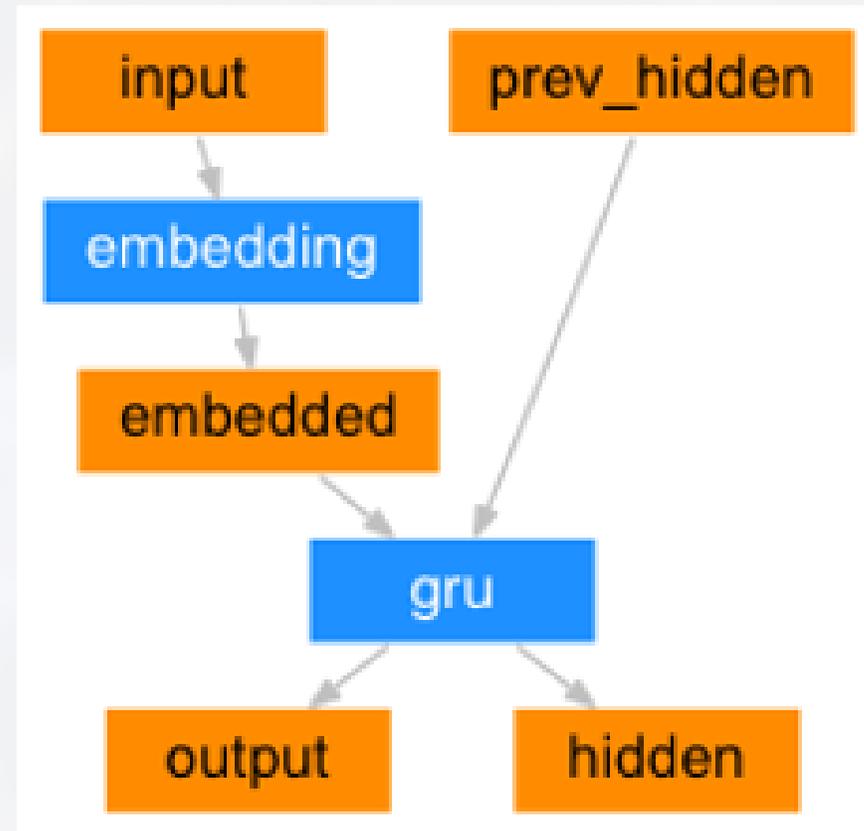
# The Encoder

- 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.

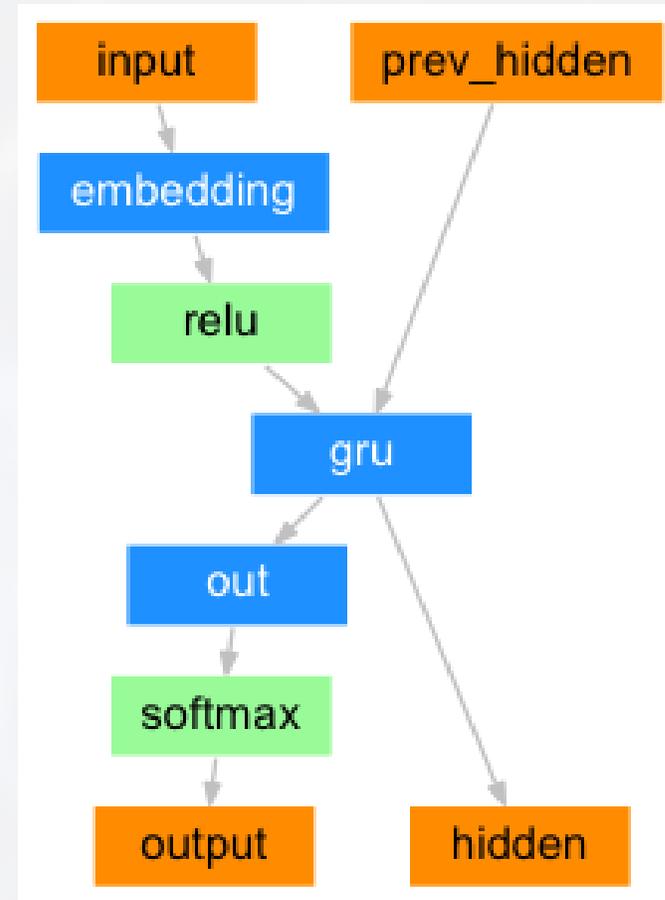
## Two approaches:

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!)



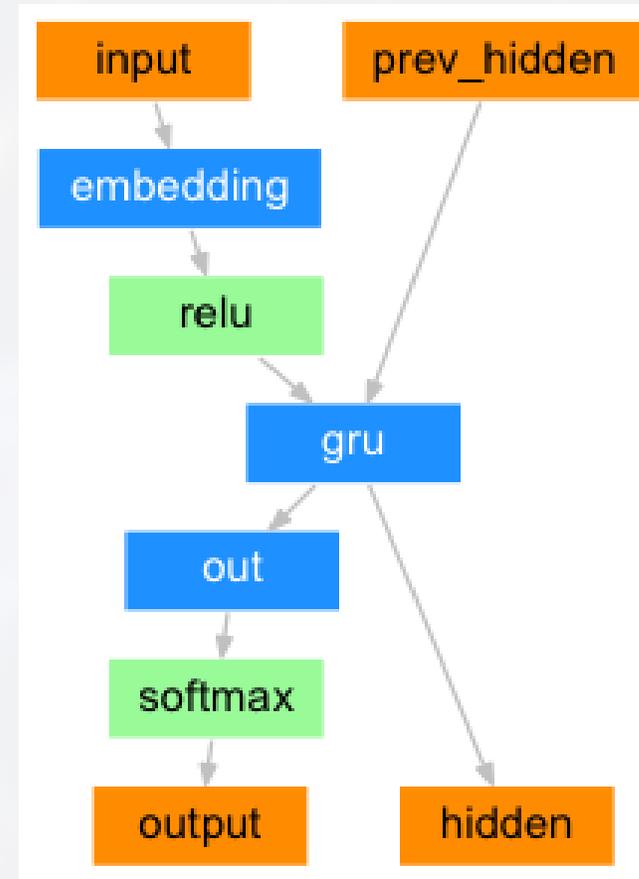
# Simple Decoder

- 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  $\langle \text{SOS} \rangle$
  - 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_size)
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
```

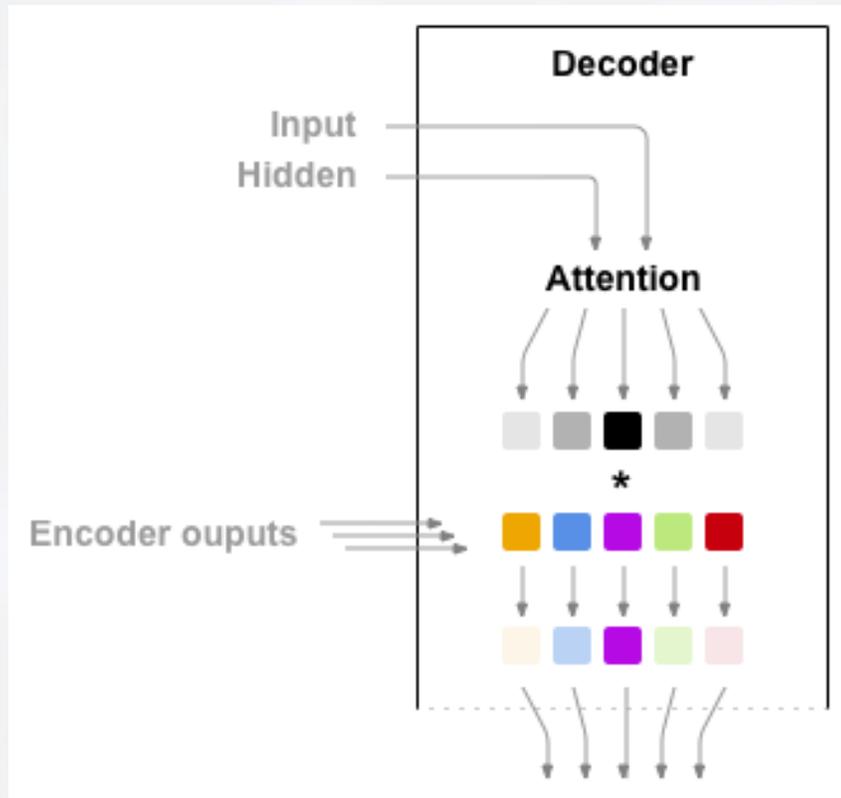


# Attention Decoder: Motivation

- 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



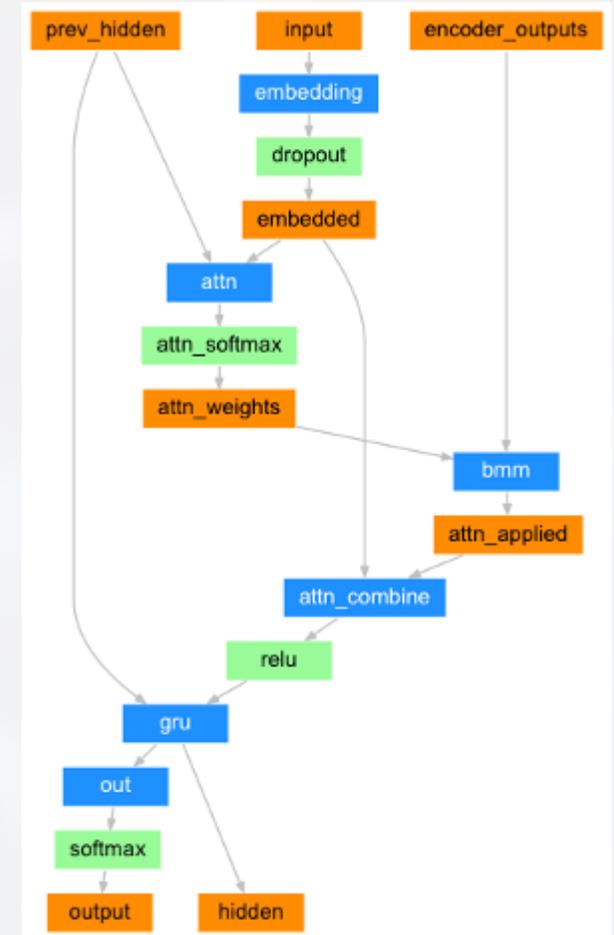
# Attention Decoder: How It Works



- 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

```
1 class AttnDecoderRNN(nn.Module):
2     def __init__(self, hidden_size, output_size, dropout_p=0.1, max_length=MAX_LENGTH):
3         super(AttnDecoderRNN, self).__init__()
4         self.hidden_size = hidden_size
5         self.output_size = output_size
6         self.dropout_p = dropout_p
7         self.max_length = max_length
8
9         self.embedding = nn.Embedding(self.output_size, self.hidden_size)
10        self.attn = nn.Linear(self.hidden_size * 2, self.max_length)
11        self.attn_combine = nn.Linear(self.hidden_size * 2, self.hidden_size)
12        self.dropout = nn.Dropout(self.dropout_p)
13        self.gru = nn.GRU(self.hidden_size, self.hidden_size)
14        self.out = nn.Linear(self.hidden_size, self.output_size)
15
16    def forward(self, input, hidden, encoder_outputs):
17        # `input` is a single long (int64) value (representing a single word)
18        # `hidden` is (1, 1, 256) (L, N, H) (1 word, 1 sample in batch, hidden size)
19        # `encoder_outputs` (10, 256) (L_in, H) (total length of input, hidden size)
20
21        # Get input embedding
22        # (1, 1, 256) (L, N, H)
23        embedded = self.embedding(input).view(1, 1, -1)
24        embedded = self.dropout(embedded) # Add random 0s
25
26        # 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:

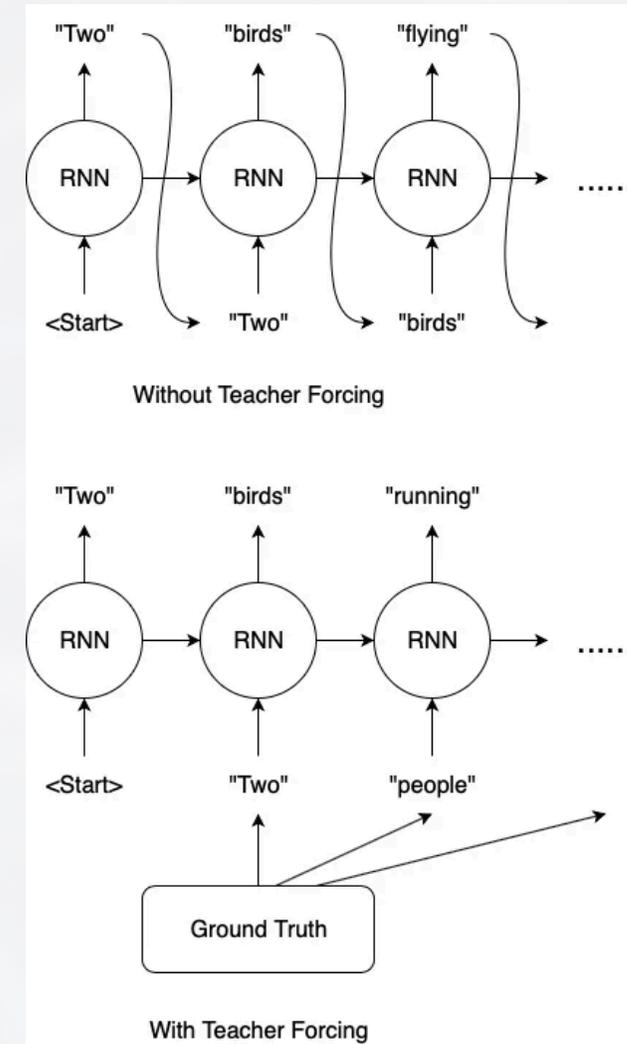
- **Input tensor:** indexes of words in the input sentence
- **Target tensor:** indexes of words in the target sentence
- Append the EOS token to both sequences

```
1 def indexesFromSentence(lang, sentence):
2     return [lang.word2index[word] for word in sentence.split(' ')]
3
4 def tensorFromSentence(lang, sentence):
5     indexes = indexesFromSentence(lang, sentence)
6     indexes.append(EOS_token)
7     return torch.tensor(indexes, dtype=torch.long, device=device).view(-1, 1)
8
9 def tensorsFromPair(pair):
10    input_tensor = tensorFromSentence(input_lang, pair[0])
11    target_tensor = tensorFromSentence(output_lang, pair[1])
12    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

```
1 import time
2 import math
3
4 def asMinutes(s):
5     m = math.floor(s / 60)
6     s -= m * 60
7     return '%dm %ds' % (m, s)
8
9 def timeSince(since, percent):
10    now = time.time()
11    s = now - since
12    es = s / (percent)
13    rs = es - s
14    return '%s (- %s)' % (asMinutes(s), asMinutes(rs))
```



# Training: Main Training Loop

```
1 def trainIters(encoder, decoder, n_iters, print_every=1000, plot_every=100, learning_rate=0.01):
2     start = time.time()
3     plot_losses = []
4     print_loss_total = 0 # Reset every print_every
5     plot_loss_total = 0 # Reset every plot_every
6
7     encoder_optimizer = optim.SGD(encoder.parameters(), lr=learning_rate)
8     decoder_optimizer = optim.SGD(decoder.parameters(), lr=learning_rate)
9     training_pairs = [tensorsFromPair(random.choice(pairs))
10                       for i in range(n_iters)]
11     criterion = nn.NLLLoss()
12
13     for iter in range(1, n_iters + 1):
14         training_pair = training_pairs[iter - 1]
15         input_tensor = training_pair[0]
16         target_tensor = training_pair[1]
17
18         loss = train(input_tensor, target_tensor, encoder,
19                     decoder, encoder_optimizer, decoder_optimizer, criterion)
20         print_loss_total += loss
21         plot_loss_total += loss
22
23         if iter % print_every == 0:
24             print_loss_avg = print_loss_total / print_every
25             print_loss_total = 0
26             print('%s (%d %d%%) %.4f' % (timeSince(start, iter / n_iters),
27                                           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.

```
1 import matplotlib.pyplot as plt
2 plt.switch_backend('agg')
3 import matplotlib.ticker as ticker
4 import numpy as np
5 %matplotlib inline
6
7
8 def showPlot(points):
9     plt.figure()
10    fig, ax = plt.subplots()
11    # this locator puts ticks at regular intervals
12    loc = ticker.MultipleLocator(base=0.2)
13    ax.yaxis.set_major_locator(loc)
14    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

```
1 def evaluate(encoder, decoder, sentence, max_length=MAX_LENGTH):
2     with torch.no_grad():
3         input_tensor = tensorFromSentence(input_lang, sentence)
4         input_length = input_tensor.size()[0]
5         encoder_hidden = encoder.initHidden()
6
7         encoder_outputs = torch.zeros(max_length, encoder.hidden_size, device=device)
8
9         for ei in range(input_length):
10            encoder_output, encoder_hidden = encoder(input_tensor[ei],
11                                                    encoder_hidden)
12            encoder_outputs[ei] += encoder_output[0, 0]
13
14            decoder_input = torch.tensor([[SOS_token]], device=device) # SOS
15
16            decoder_hidden = encoder_hidden
17
18            decoded_words = []
19            decoder_attentions = torch.zeros(max_length, max_length)
20
21            for di in range(max_length):
22                decoder_output, decoder_hidden, decoder_attention = decoder(
23                    decoder_input, decoder_hidden, encoder_outputs)
24                decoder_attentions[di] = decoder_attention.data
25                topv, topi = decoder_output.data.topk(1)
26                if topi.item() == EOS_token:
27                    decoded_words.append('EOS')
```

# 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:

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



# Training and Evaluating

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

```
1 hidden_size = 256
2 encoder1 = EncoderRNN(input_lang.n_words, hidden_size).to(dev
3 attn_decoder1 = AttnDecoderRNN(hidden_size,
4     output_lang.n_words, dropout_p=0.1).to(device)
5
6 if os.path.exists('encoder1.pth') and os.path.exists('attn_de
7     # Load the models if not training
8     print("Loading pretrained models rather than training")
9     encoder1.load_state_dict(torch.load('encoder1.pth'))
10    attn_decoder1.load_state_dict(torch.load('attn_decoder1.p
11 else:
12    trainIters(encoder1, attn_decoder1, 75000, print_every=10
13    # Save the models after training
14    torch.save(encoder1.state_dict(), 'encoder1.pth')
15    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)
```

```
> vous etes fascinant .  
= you re fascinating .  
< you re fascinating . <EOS>
```

```
> vous etes tres genereuse .  
= you re very generous .  
< you re very generous . <EOS>
```

```
> je le fais pour toi .  
= i m doing this for you .  
< i m doing this for you . <EOS>
```

```
> nous sommes toutes les memes .  
= we re all the same .  
< we re all the same . <EOS>
```

```
> je suis desolee si je vous ai derangee .  
= i m sorry if i disturbed you .  
< i m sorry if i disturbed you . <EOS>
```

```
> je vais chez le dentiste demain .  
= i am going to see the dentist tomorrow .  
< i am going to the the the . . <EOS>
```

```
> j en ai marre de perdre .  
= i m tired of losing .  
< i m tired of losing . <EOS>
```

# Visualizing Attention

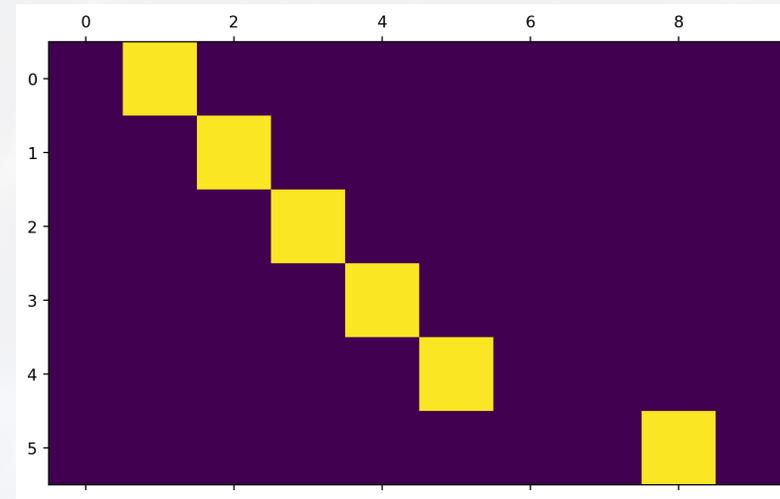
A useful property of the attention mechanism is its highly interpretable outputs.

- 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(attention)` to see attention output displayed as a matrix
  - Columns are input steps
  - Rows are output steps



# Visualizing Attention

```
1 %matplotlib inline
2 output_words, attentions = evaluate(
3     encoder1, attn_decoder1, "je suis trop froid .")
4 plt.matshow(attendances.numpy())
5 plt.show()
```



# Visualizing Attention: Better Display

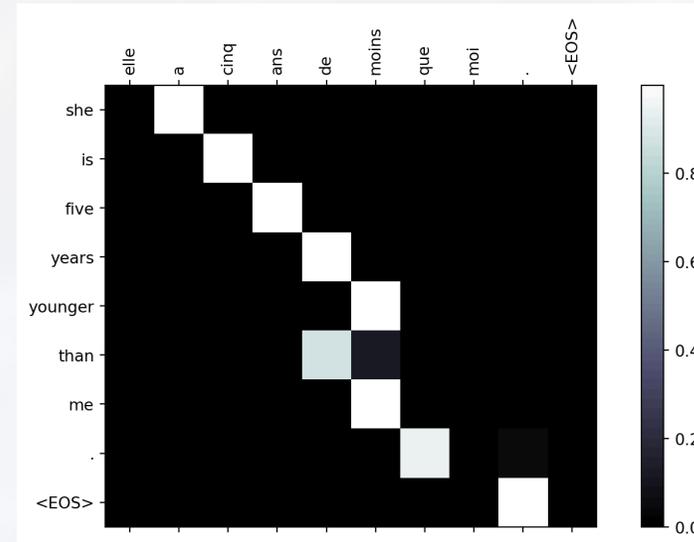
For a better viewing experience we add axes and labels:

```
1 def showAttention(input_sentence, output_words, attentions):
2     # Set up figure with colorbar
3     fig = plt.figure()
4     ax = fig.add_subplot(111)
5     cax = ax.matshow(attentions.numpy(), cmap='bone')
6     fig.colorbar(cax)
7
8     # Set up axes
9     ax.set_xticklabels([''] + input_sentence.split(' ') +
10                        ['<EOS>'], rotation=90)
11     ax.set_yticklabels([''] + output_words)
12
13     # Show label at every tick
14     ax.xaxis.set_major_locator(ticker.MultipleLocator(1))
15     ax.yaxis.set_major_locator(ticker.MultipleLocator(1))
16
17     plt.show()
18
19 def evaluateAndShowAttention(input_sentence):
20     output_words, attentions = evaluate(
21         encoder1, attn_decoder1, input_sentence)
22     print('input =', input_sentence)
23     print('output =', ' '.join(output_words))
24     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 younger than me . <EOS>
```



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

