Introduction to Deep Learning

Outline

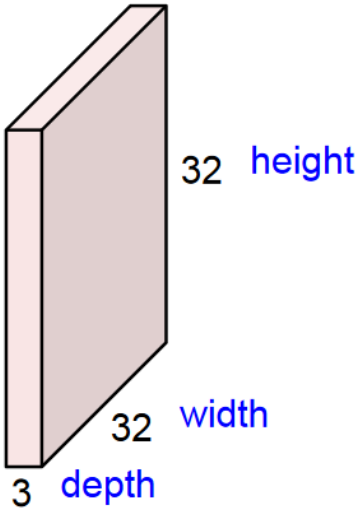
- **Deep Learning**
  - **CNN**
  - **RNN**
  - **Attention**
  - **Transformer**
- Pytorch
  - Introduction
  - Basics
  - Examples

CNNs

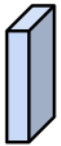
Some slides borrowed from Fei-Fei Li & Justin Johnson & Serena Yeung at Stanford.

Convolutional Layer

Input
32x32x3 image



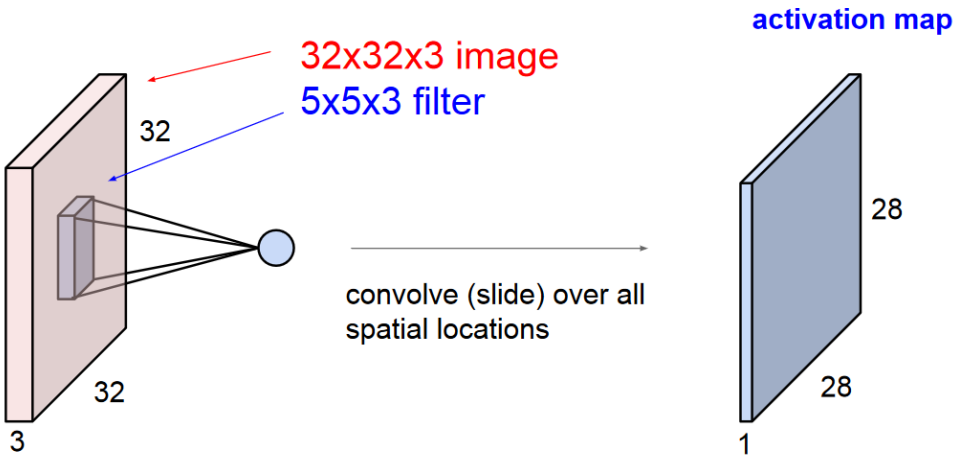
Filter
5x5x3



Convolve the filter with the image i.e. “slide over the image spatially, computing dot products”

Filters always extend the full depth of the input volume.

Convolutional Layer

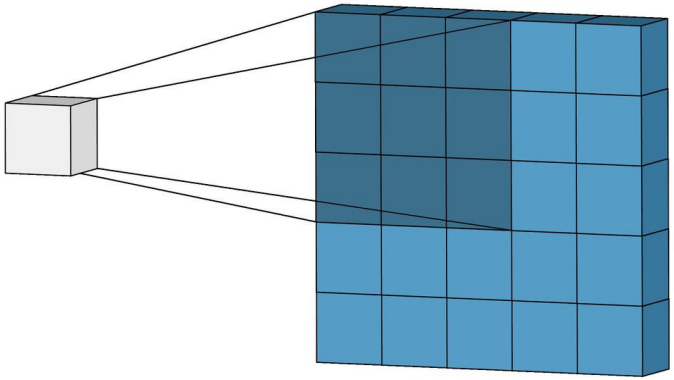


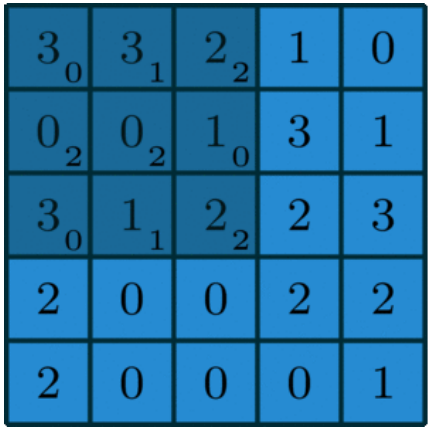
At each step during the convolution, the filter acts on a region in the input image and results in a single number as output.

This number is the result of the dot product between the values in the filter and the values in the 5x5x3 chunk in the image that the filter acts on.

Combining these together for the entire image results in the activation map.

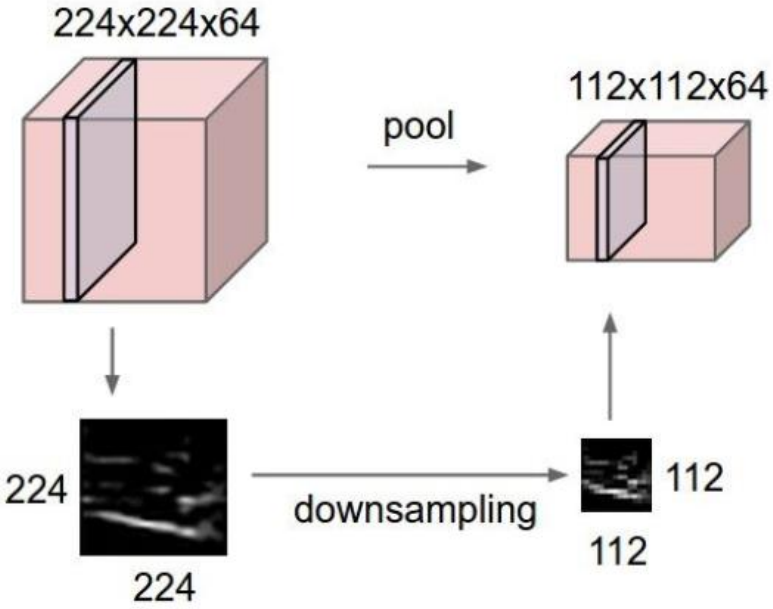
Convolutional Layer





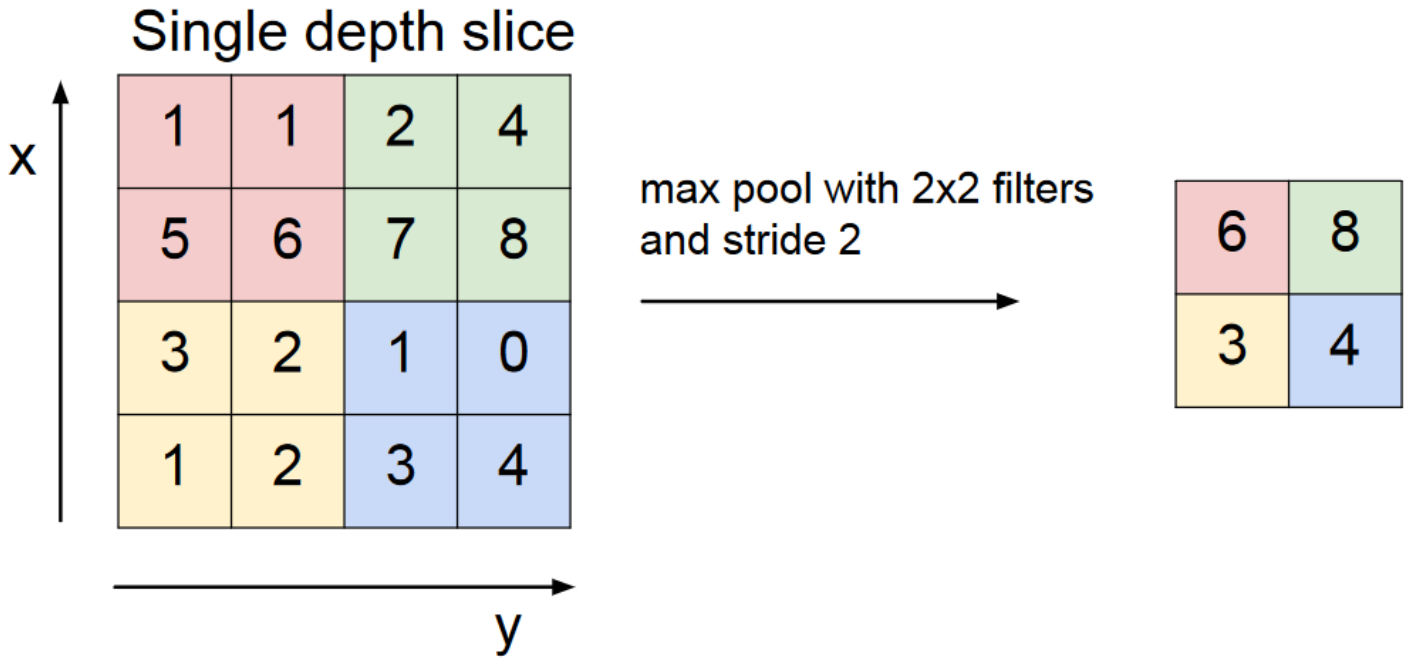
Visualizations borrowed from Irhum Shafkat's blog.

Pooling Layer



- makes the representations smaller and more manageable
- operates over each activation map independently

Max Pooling


Single depth slice
x
y
1 1 2 4
5 6 7 8
3 2 1 0
1 2 3 4
max pool with 2x2 filters
and stride 2
6 8
3 4

ConvNet Layer

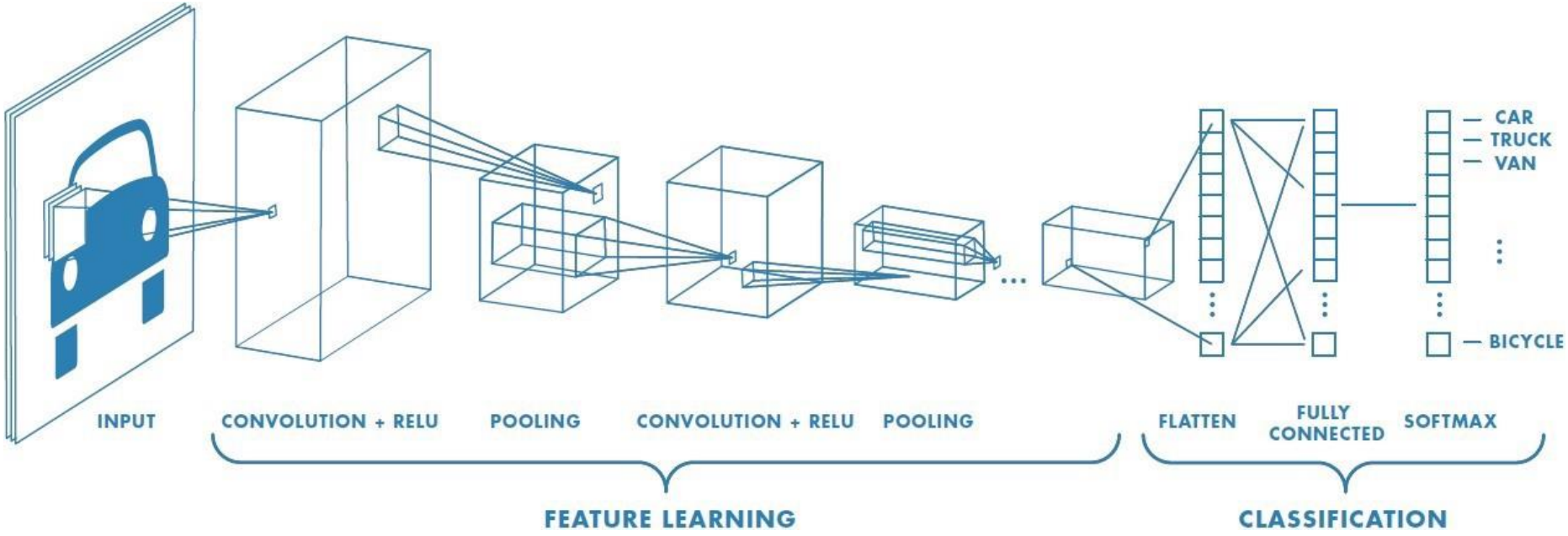
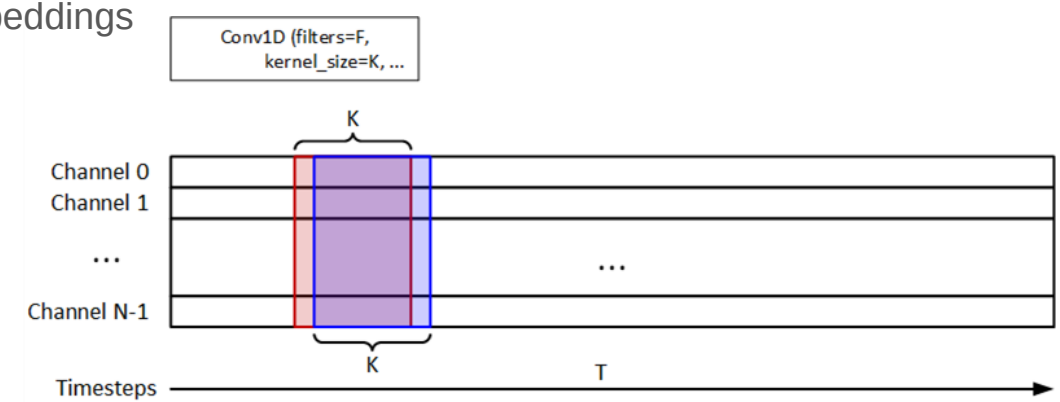


Image credits- Saha's blog.

Application in text

- Not many CNNs in modern NLP
  - Some adjacent applications exist, such as graph convolutions, image captioning, VQA, etc.

- For text sequences, it sometimes helps to use **1-dimensional** convolutions (because embedding dimension ordering has no intrinsic meaning)
- Extensions to different filter sizes can result in various **n-gram** features
  - Assuming good underlying word embeddings

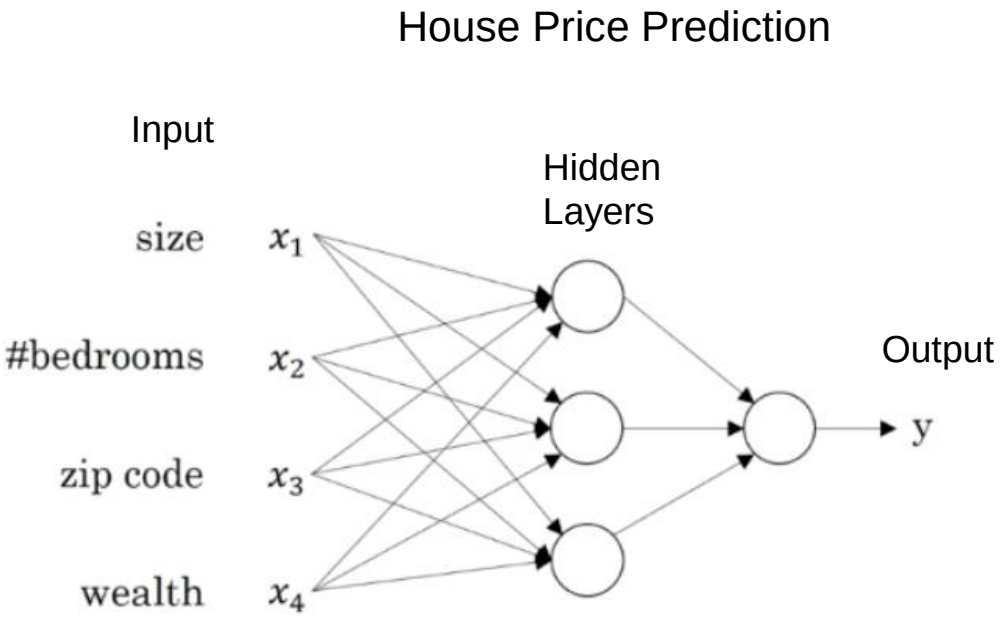


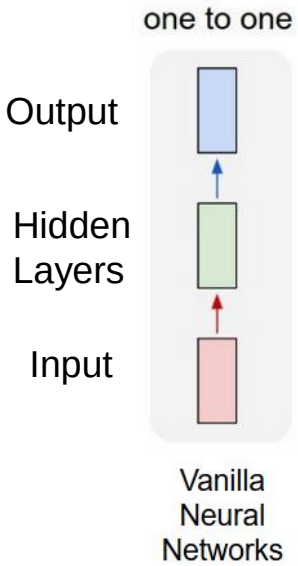
https://gdcoder.com/what-is-an-n-gram-multichannel-convolutional-neural-network-for-text-classification/

RNNs

Some slides borrowed from Fei-Fei Li & Justin Johnson & Serena Yeung at Stanford.

Vanilla Neural Networks

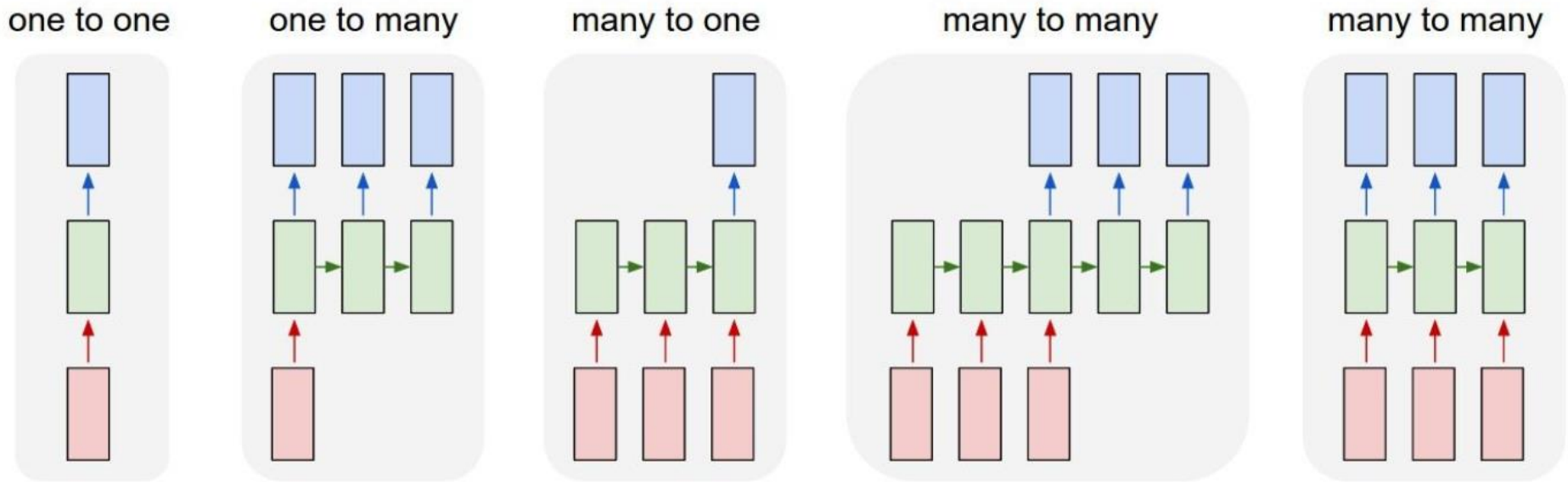




How to model sequences?

- Text Classification: Input Sequence → Output label
- Translation: Input Sequence → Output Sequence
- Image Captioning: Input image → Output Sequence

RNN - Recurrent Neural Networks



Vanilla Neural Networks

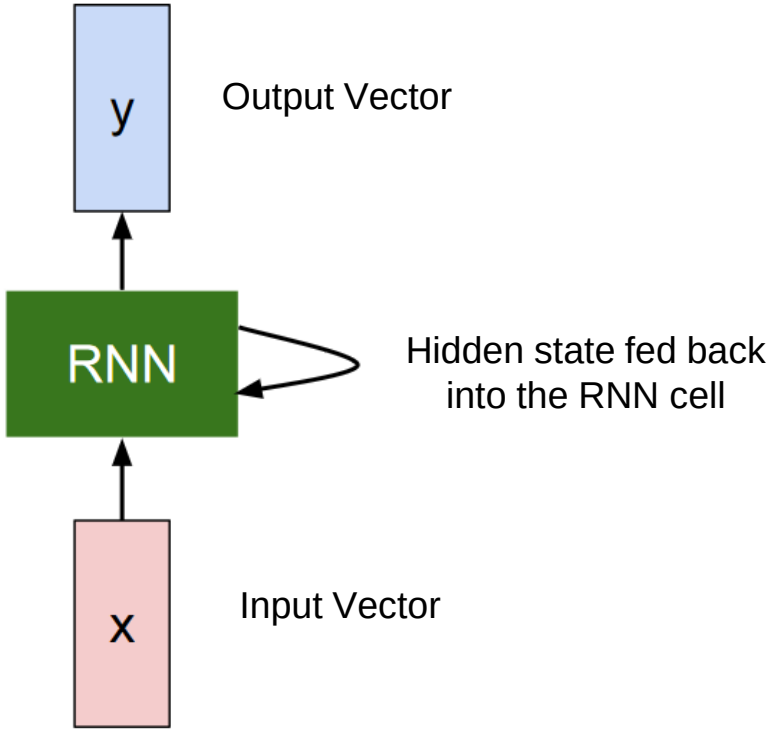
e.g. Image captioning

e.g. Text classification

e.g. Translation

e.g. POS tagging

RNN - Representation



RNN - Recurrence Relation

Recurrent Neural Networks (RNNs) are a solution to sequential data.
Each token of the input is processed in series, and the network considers both the current token as well as all previous information.

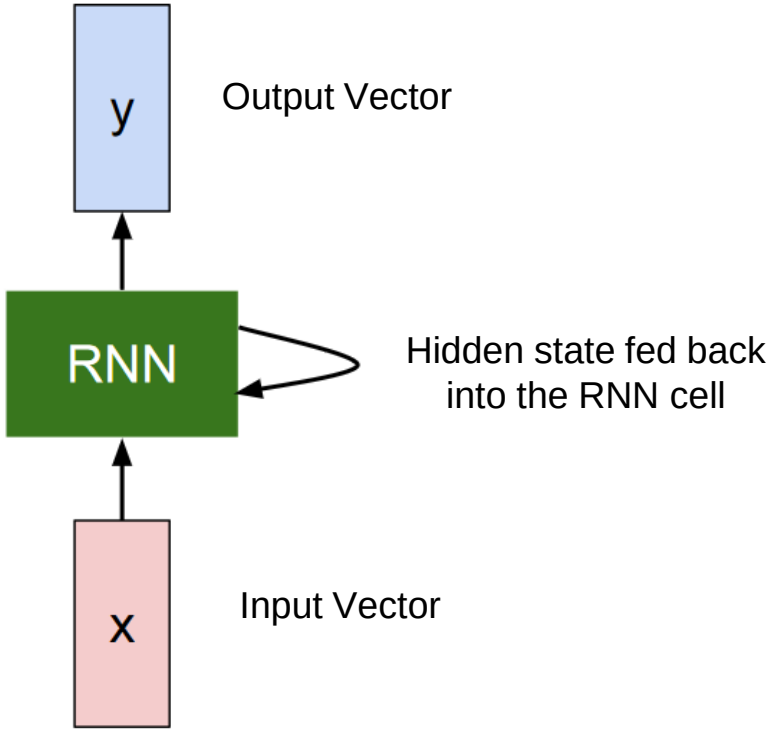
$$h_t = f_W(h_{t-1}, x_t)$$

new state

old state

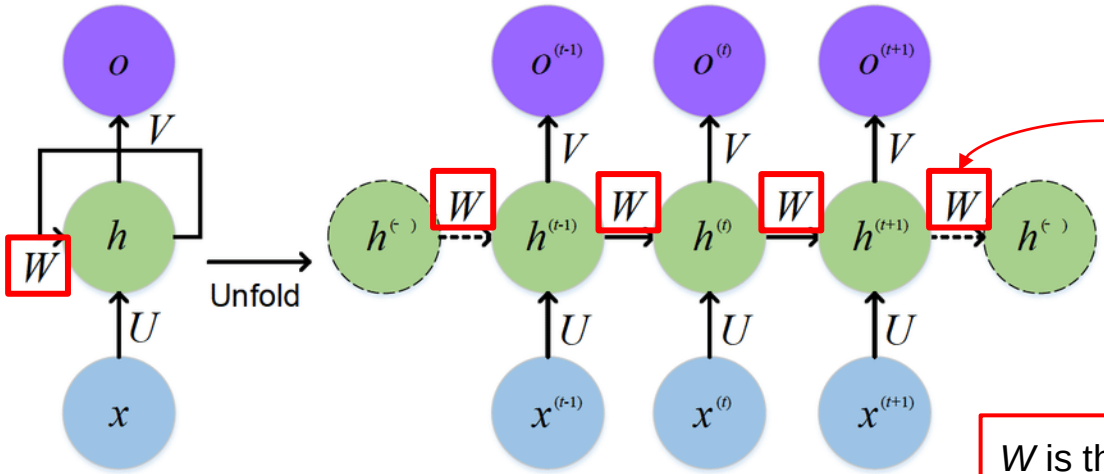
input vector at some time step

some function with parameters W



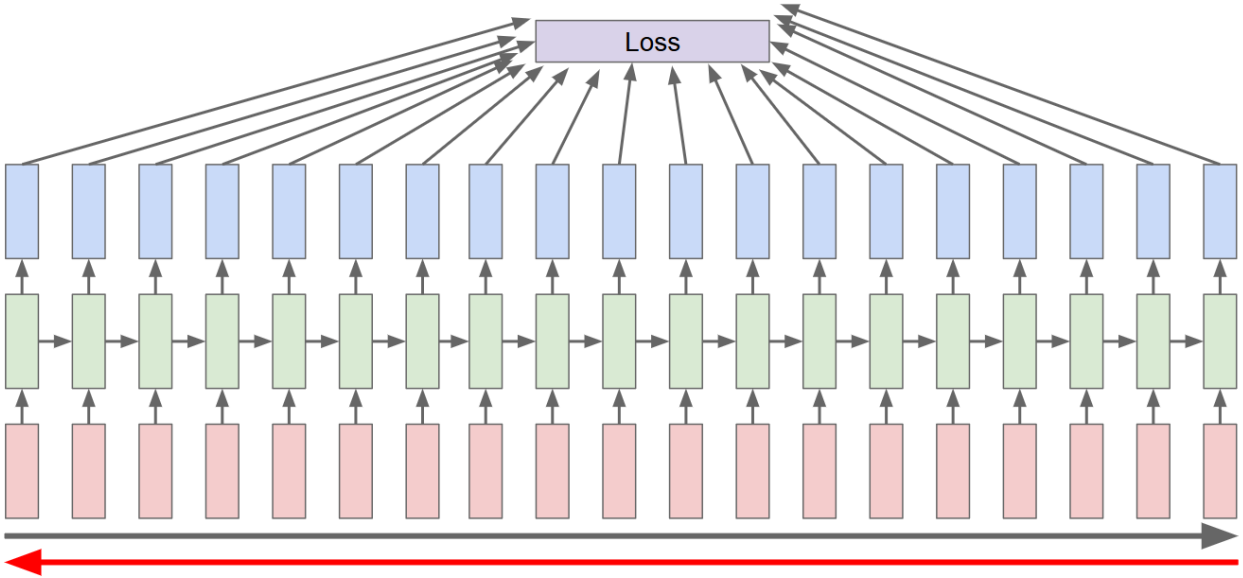
RNN - Rolled out representation

After considering the entire sequence, the final hidden state of the network will hold all relevant information from the entire sequence.



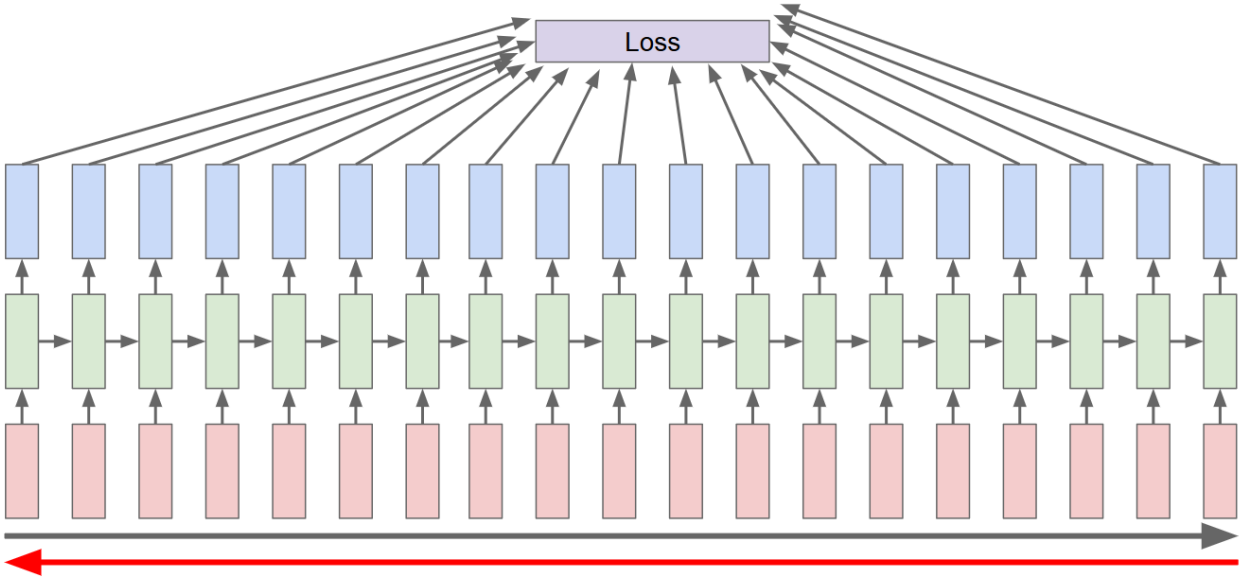
https://www.researchgate.net/figure/The-standard-RNN-and-unfolded-RNN_fig1_318332317

RNN - Backpropagation Through Time



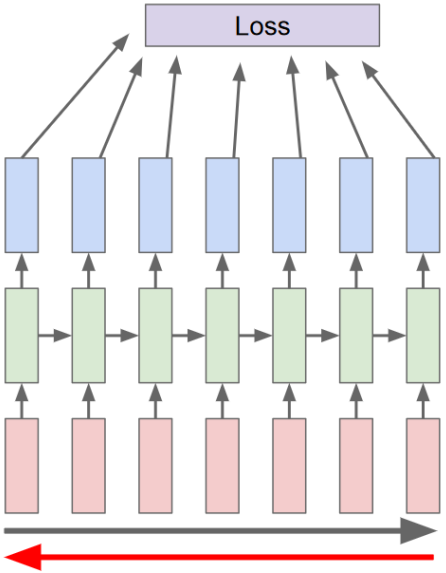
Forward pass through entire sequence to produce intermediate hidden states, output sequence and finally the loss. Backward pass through the entire sequence to compute gradient.

RNN - Backpropagation Through Time



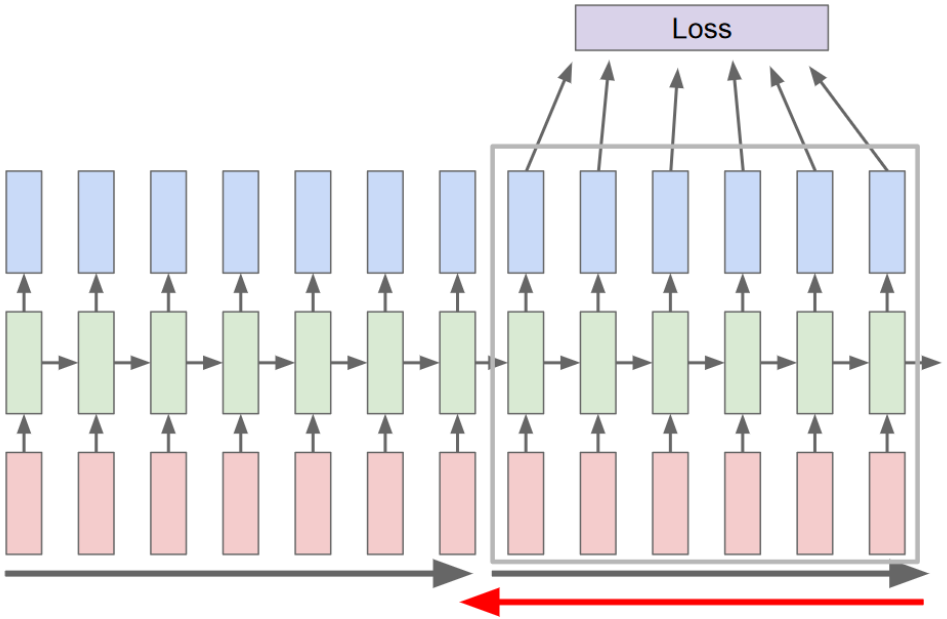
Running Backpropagation through time for the entire text would be very slow.

RNN - Truncated Backpropagation Through Time



Run forward and backward through chunks of the sequence instead of whole sequence

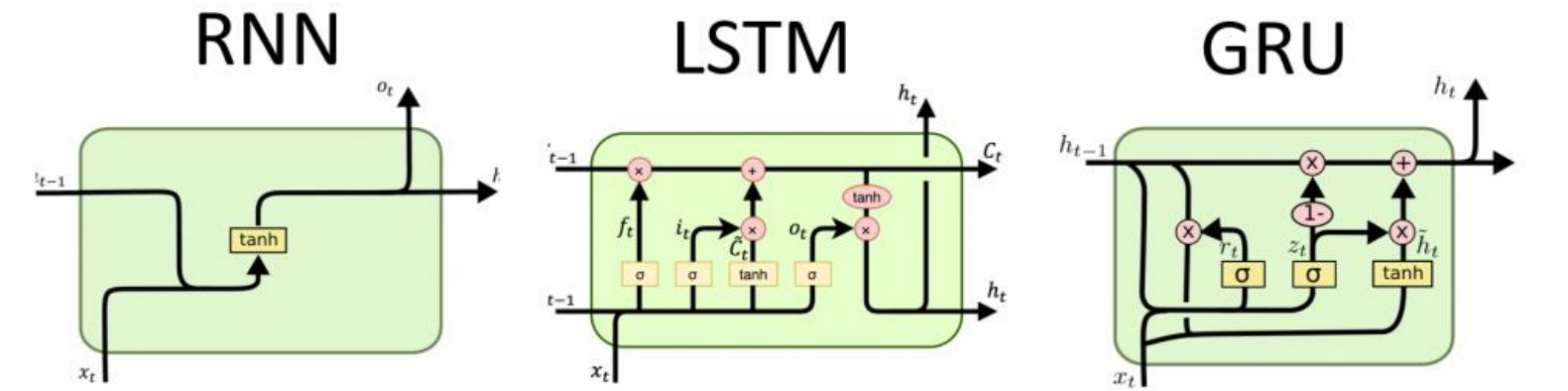
RNN - Truncated Backpropagation Through Time



Switch to an approximation-
Truncated Backpropagation Through Time

RNN Variants

The standard RNN has several variants, among which the base RNN, LSTM (long short term memory) and GRU (gated recurrent unit) are the most popular.



http://dprogrammer.org/rnn-lstm-gru

RNN Further Reading

Some good resources:

Understanding LSTM Networks

An Empirical Exploration of Recurrent Network Architectures

Stanford CS231n: Lecture 10 | Recurrent Neural Networks

http://karpathy.github.io/2015/05/21/rnn-effectiveness/

https://arxiv.org/pdf/1412.3555v1.pdf LSTMs vs GRUs

In PyTorch

```
class LSTMNet(nn.Module):
    def __init__(self,
                 vocab_size: int,
                 output_dim: int,
                 embed_dim: int = 128):
        super(LSTMNet, self).__init__()
        self.rnn = nn.LSTM(input_size=embed_dim,
                           hidden_size=embed_dim,
                           num_layers=1,
                           batch_first=True)
        self.embedding = nn.Embedding(vocab_size, embed_dim)
        self.activ = nn.ReLU()
        self.lin = nn.Linear(embed_dim, output_dim)

    def forward(self, x):
        x = self.embedding(x)
        x, (hidden_state, cell_states) = self.rnn(x)
        x = self.lin(self.activ(hidden_state[-1]))
        return x
```

How many words will be in the vocabulary for this task?

How many output units do we need (i.e. classes)?

How big should word embedding vectors be?

Create an LSTM that expects word-embeddings to be “embed_dim” size, has a hidden state as big as “embed_dim”, has a single layer of hidden units, and expects data in (batch, sequence, token) form, as opposed to (sequence, batch, token).

Create an embedding matrix for “vocab_size” words with “embed_dim” sized embeddings

We’ll use a ReLU activation after the LSTM

We’ll pass the LSTM output through a linear layer to get it to “output_dim” size.

Given word indices, embed them with our word embedding matrix

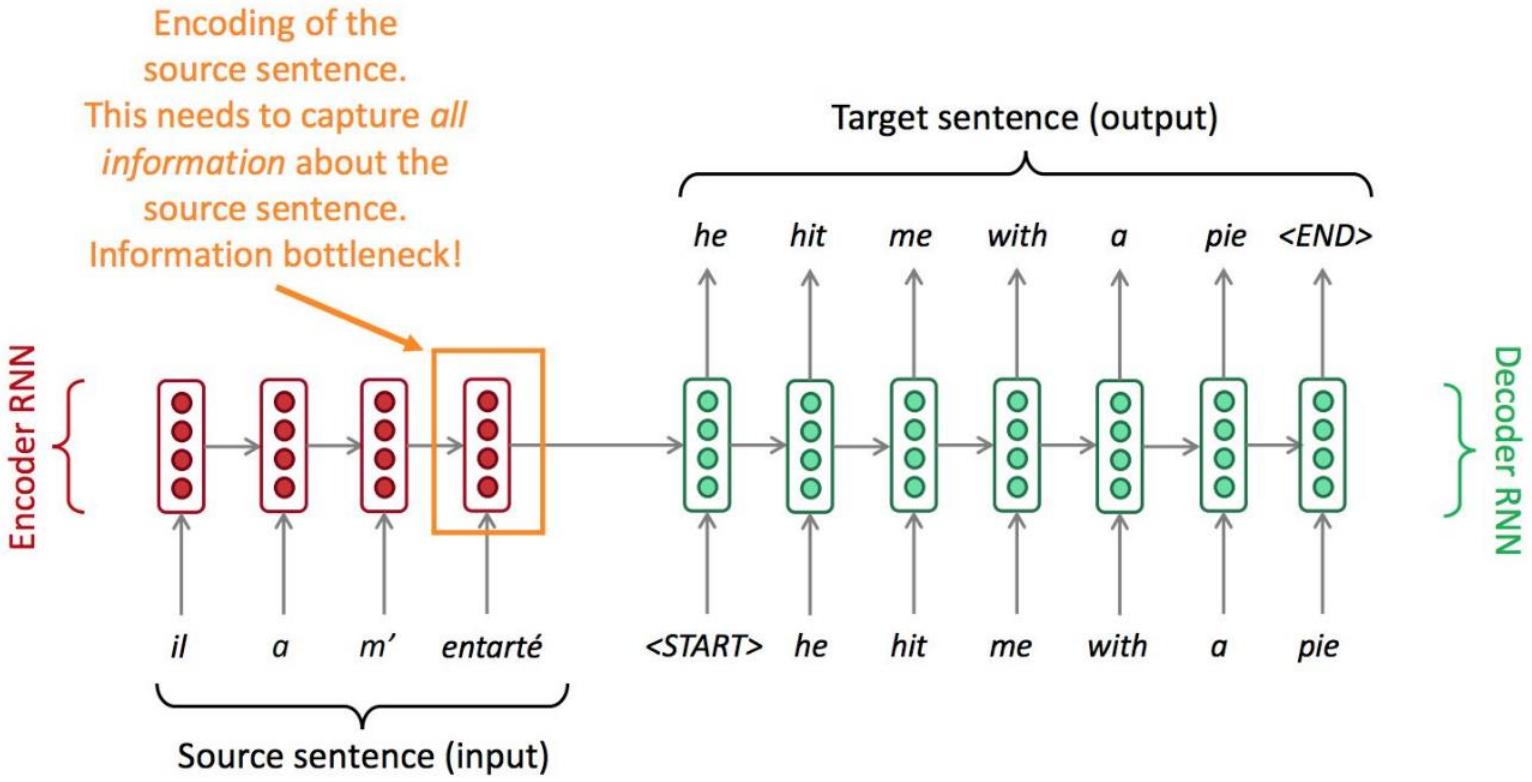
Pass the sequence through the LSTM. We could pass a hidden state in here, if we had any prior knowledge, but we don’t here. We get output states for each token (x) as well as the network’s hidden states and cell states at each point step of the sequence (hidden_state, cell_state)

Use the final hidden state to represent the whole sequence, pass through the ReLU (self.activ()) then the final linear layer (self.lin())

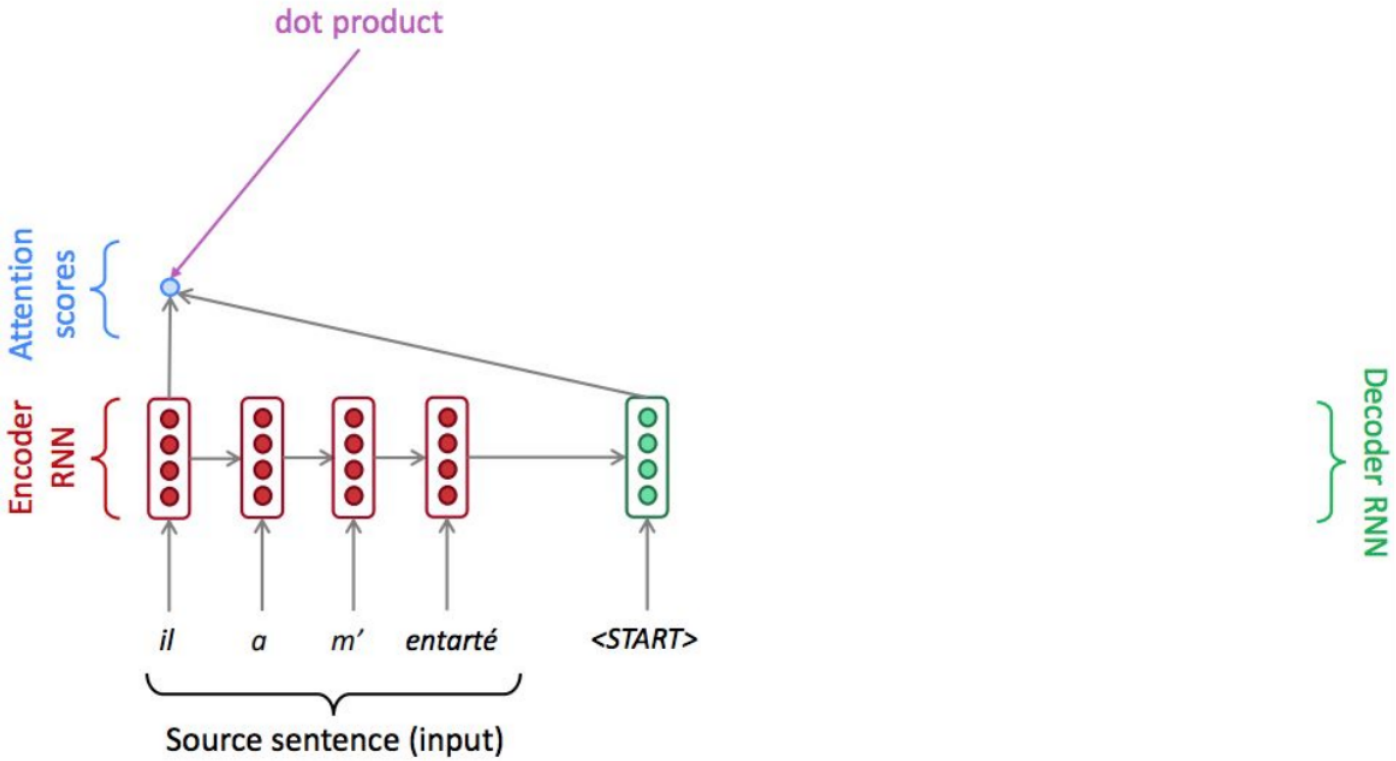
Attention

Some slides borrowed from Sarah Wiegreffe
at Georgia Tech and Abigail See, Stanford CS224n.

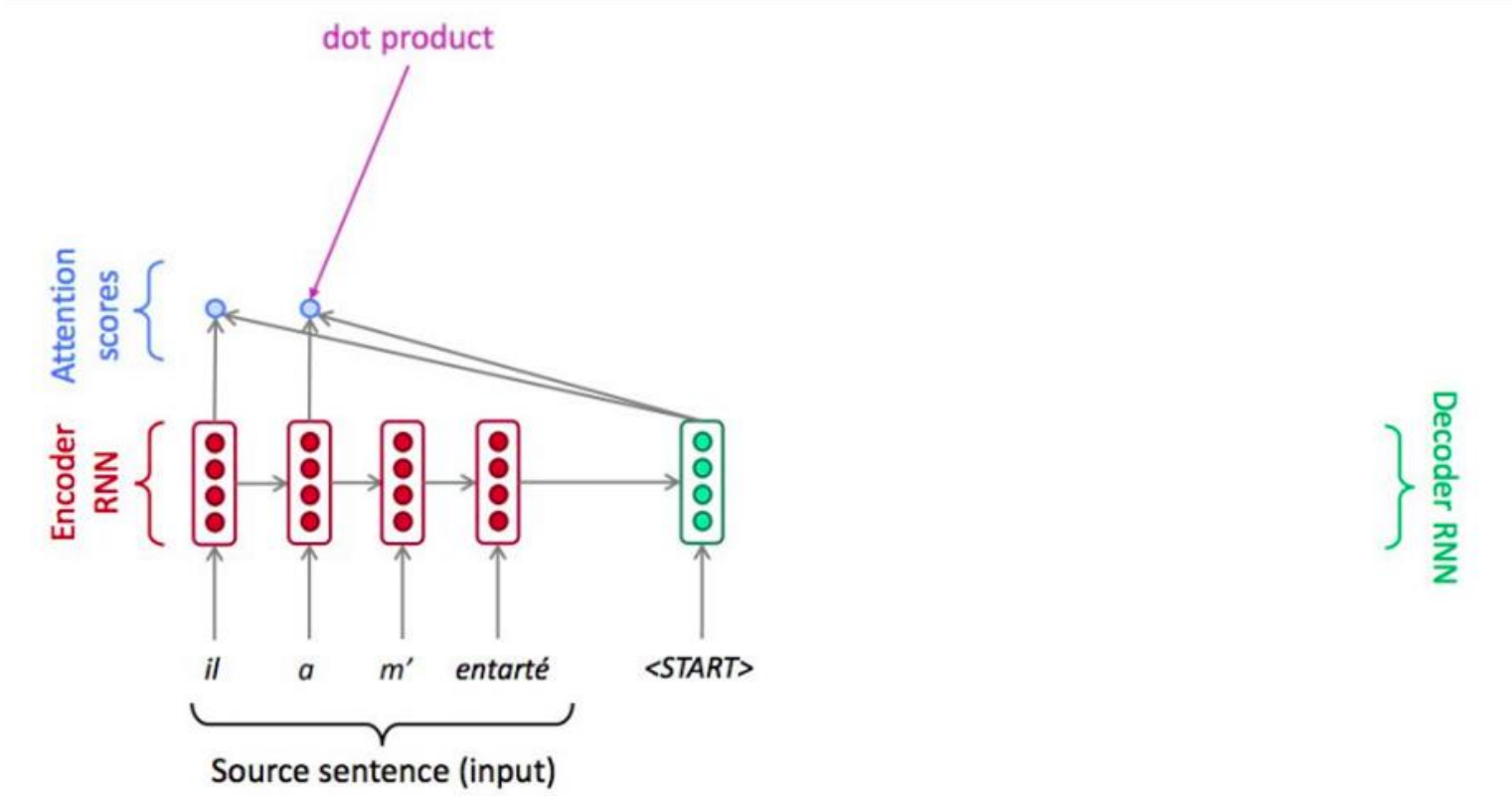
RNN


Encoding of the source sentence. This needs to capture *all information* about the source sentence. Information bottleneck!
Target sentence (output)
he hit me with a pie <END>
Encoder RNN
Decoder RNN
il a m' entarté
<START> he hit me with a pie
Source sentence (input)

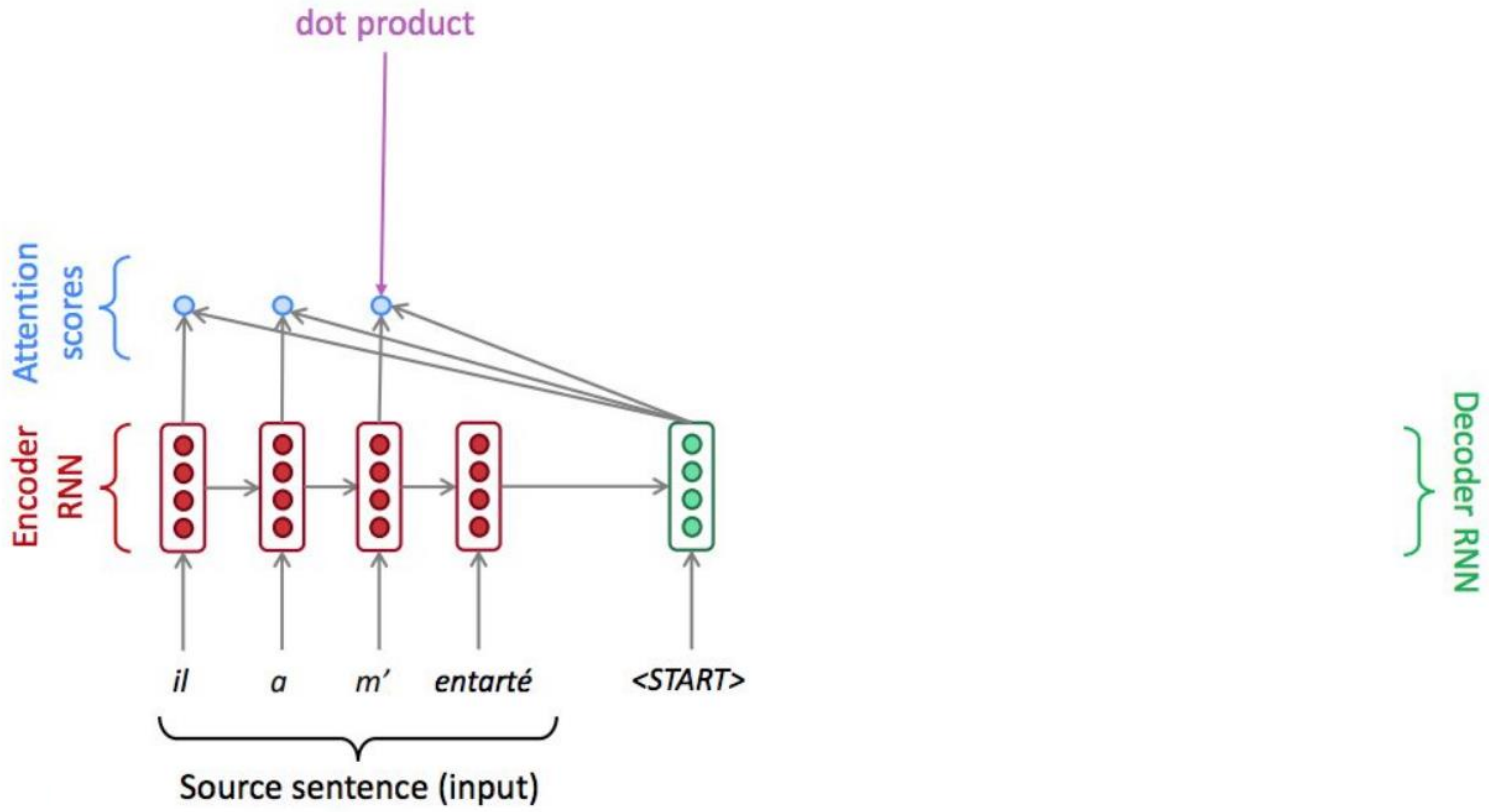
RNN - Attention



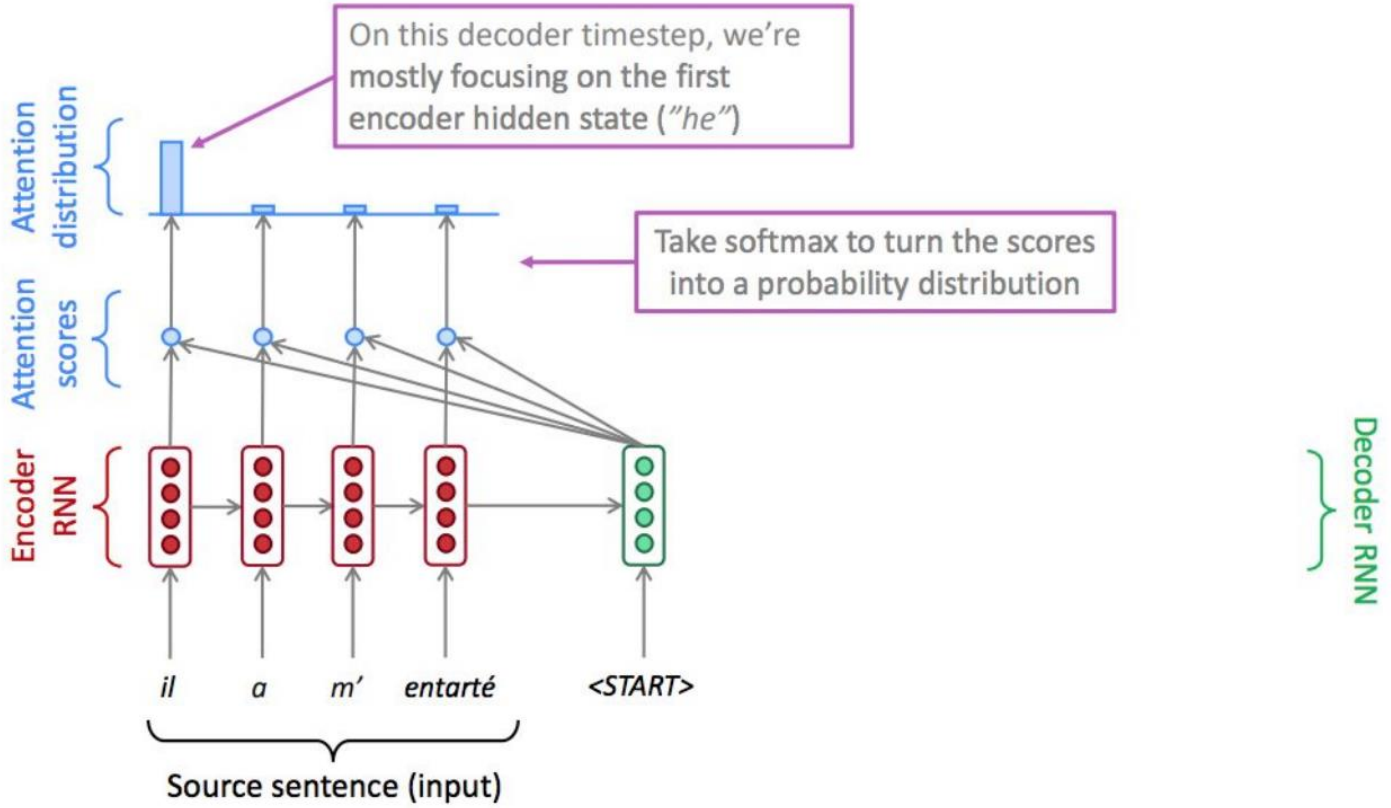
RNN - Attention



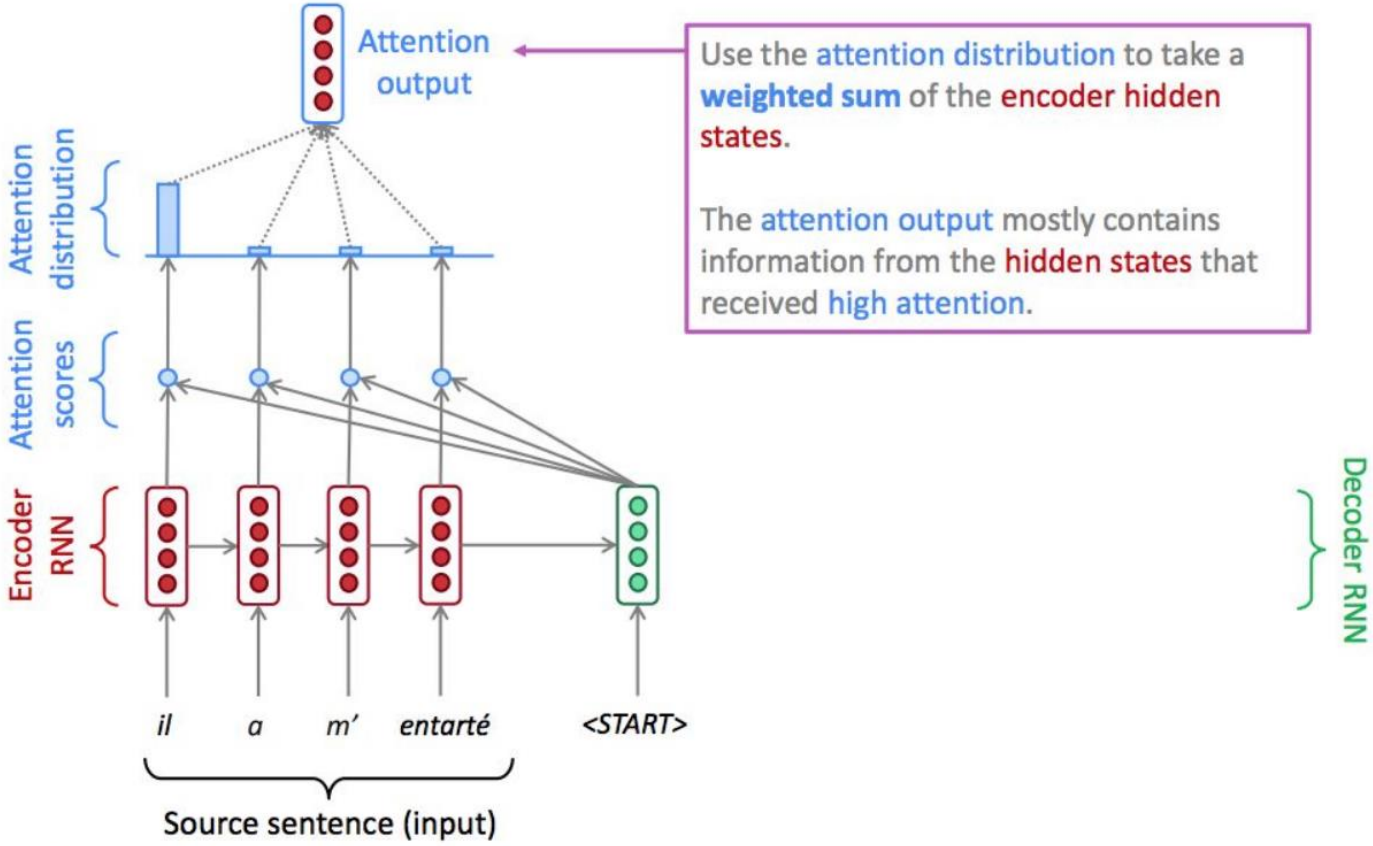
RNN - Attention



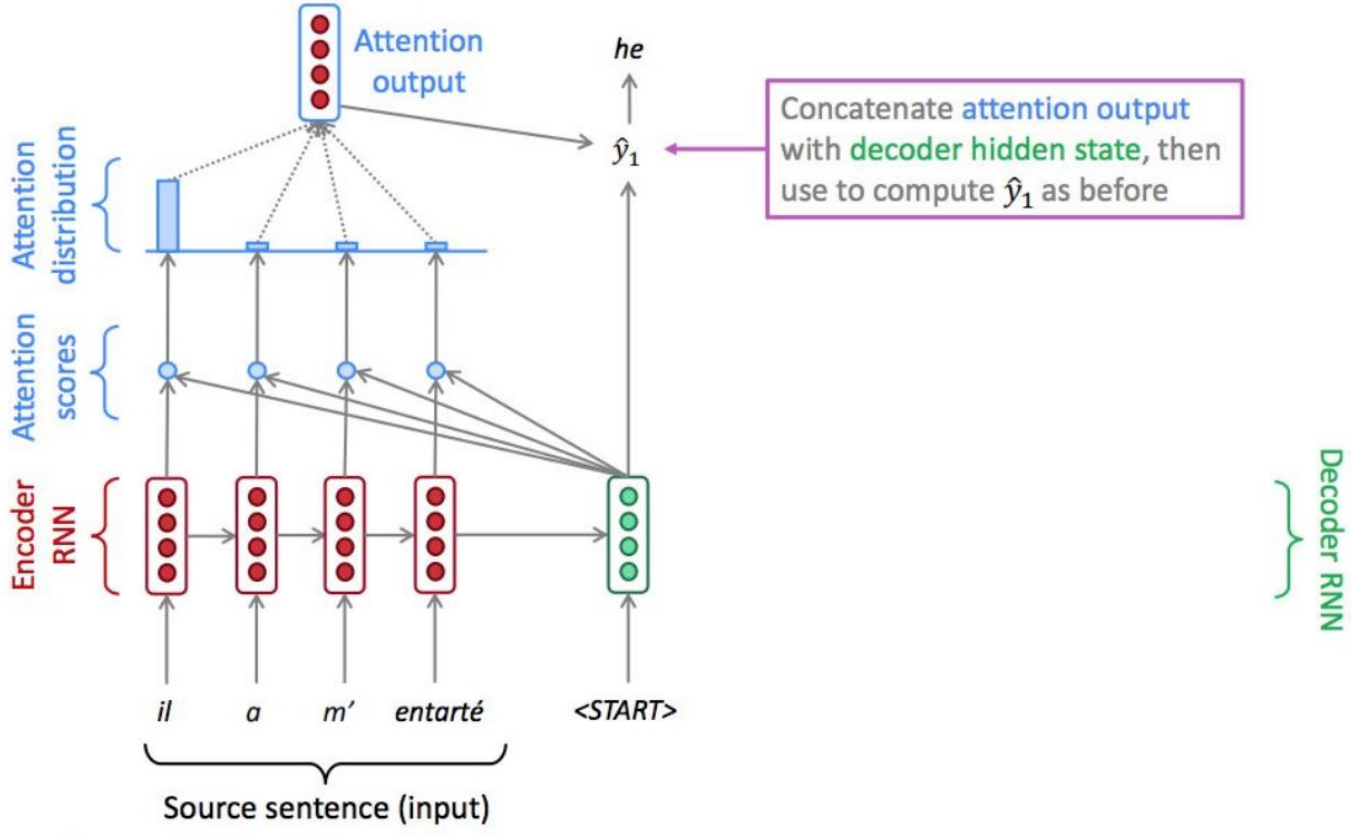
RNN - Attention



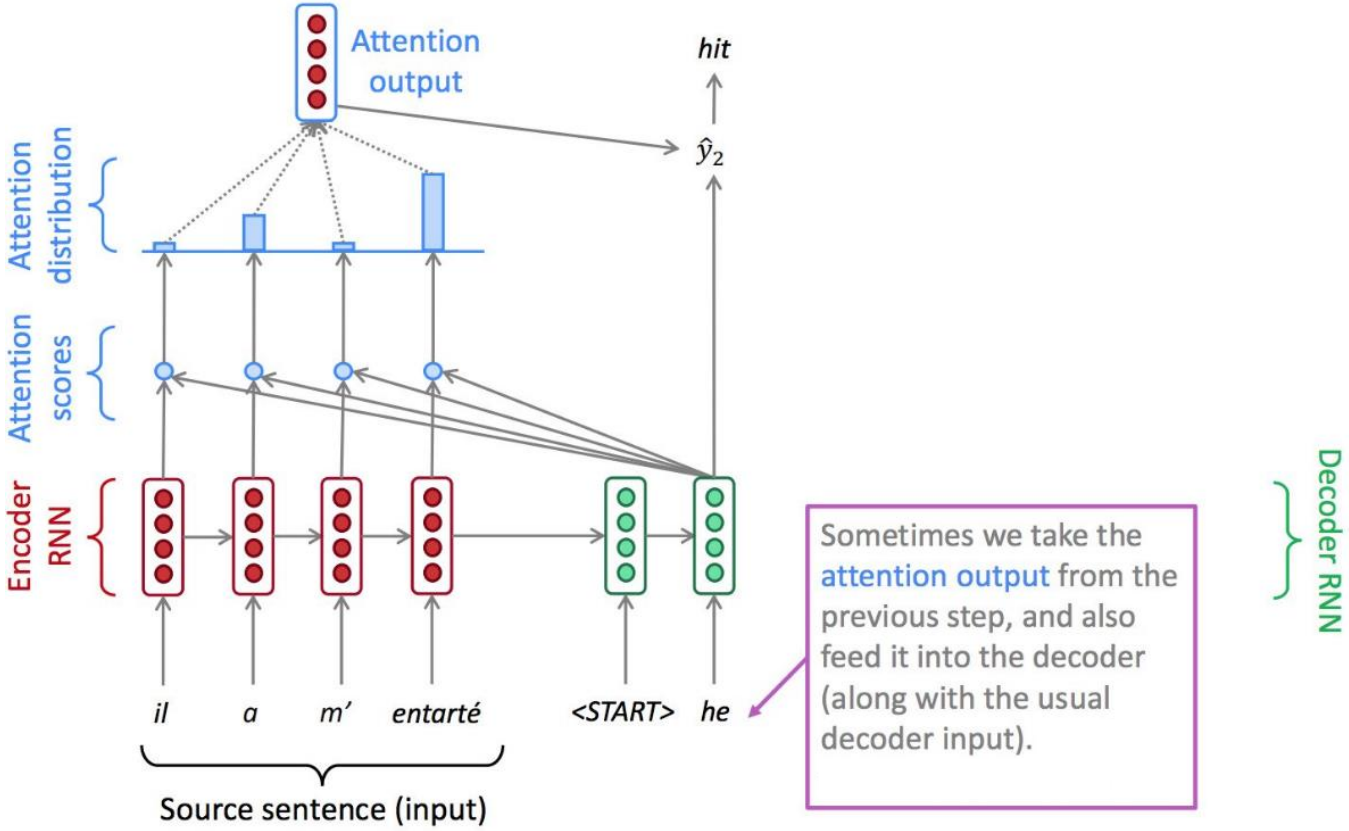
RNN - Attention



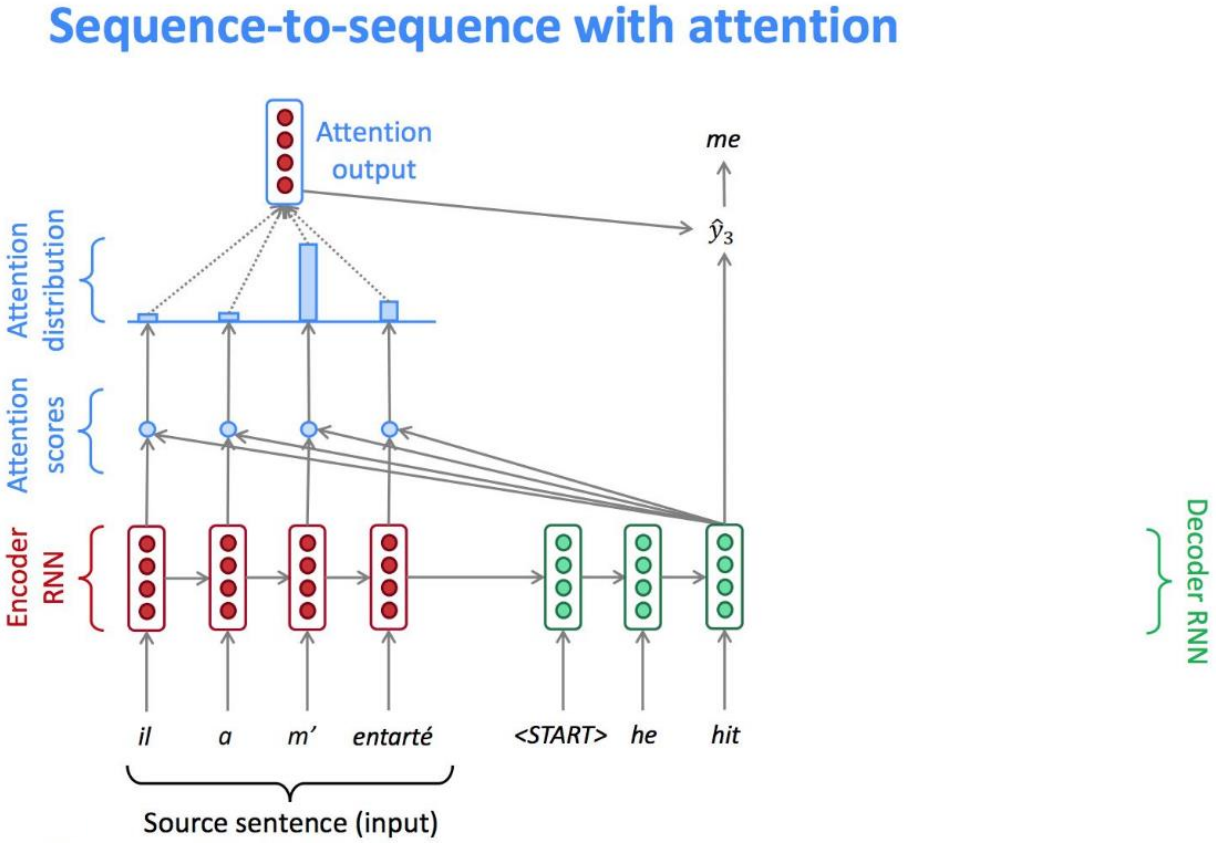
RNN - Attention



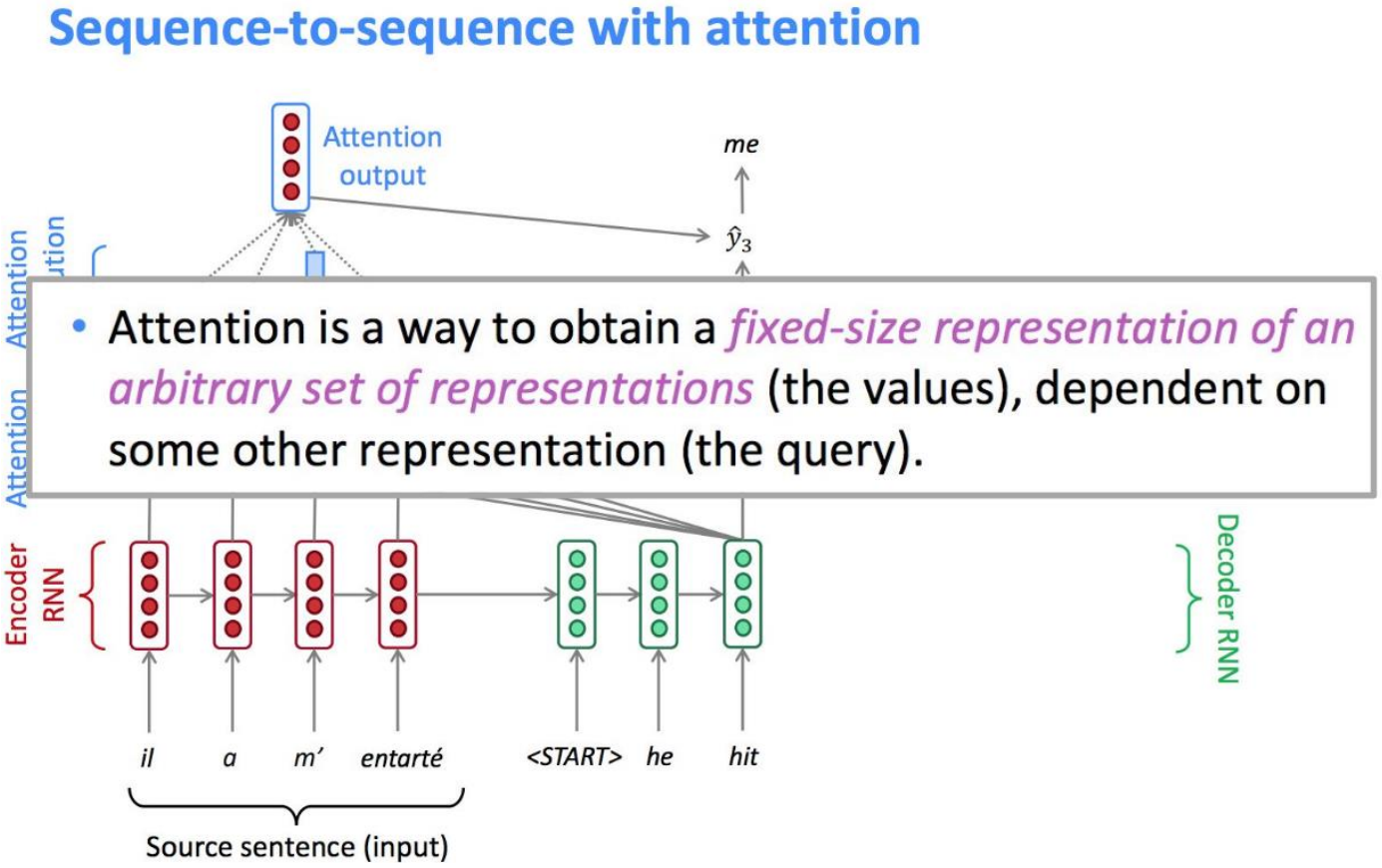
RNN - Attention



RNN - Attention



RNN - Attention



What does attention do for us?

- Allows network to focus on specific components of the sequence as they are needed
  - Results in improved performance for decoders
- Bypasses network bottleneck
  - Means that sequence information isn’t as likely to get lost or washed out
- Helps with vanishing gradients
  - Even with long sequences, attention can pass gradient information directly from one end of the sequence to another
- Helps with debugging or feature engineering
  - What does the network seem to be focusing on, why might that be?

Drawbacks of RNN

- RNNs involve sequential computation
  - can’t parallelize = time-consuming
- RNNs “forget” past information
- No explicit modeling of long and short range dependencies

Transformer

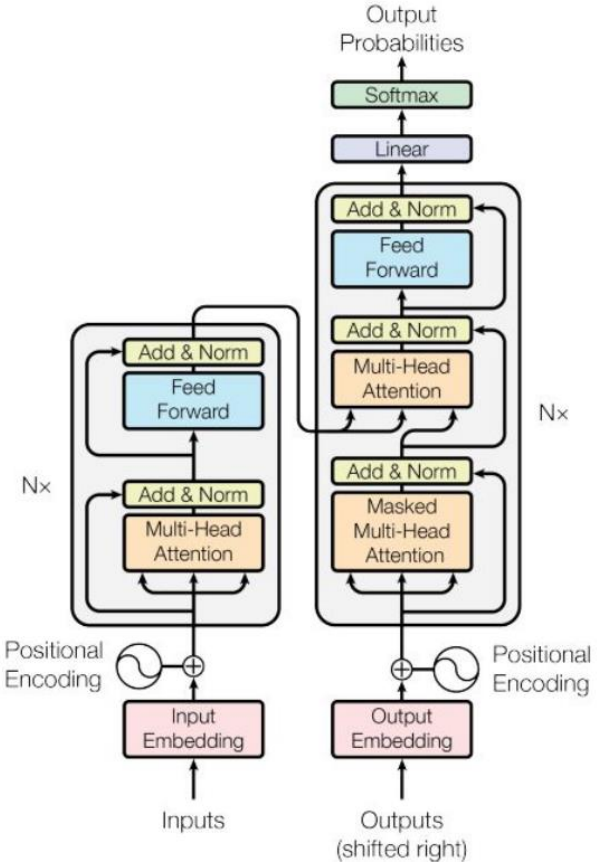
Some slides borrowed from Sarah Wiegreffe
at Georgia Tech and "The Illustrated Transformer"
https://jalammar.github.io/illustrated-transformer/

Transformer

“Attention is All You Need”

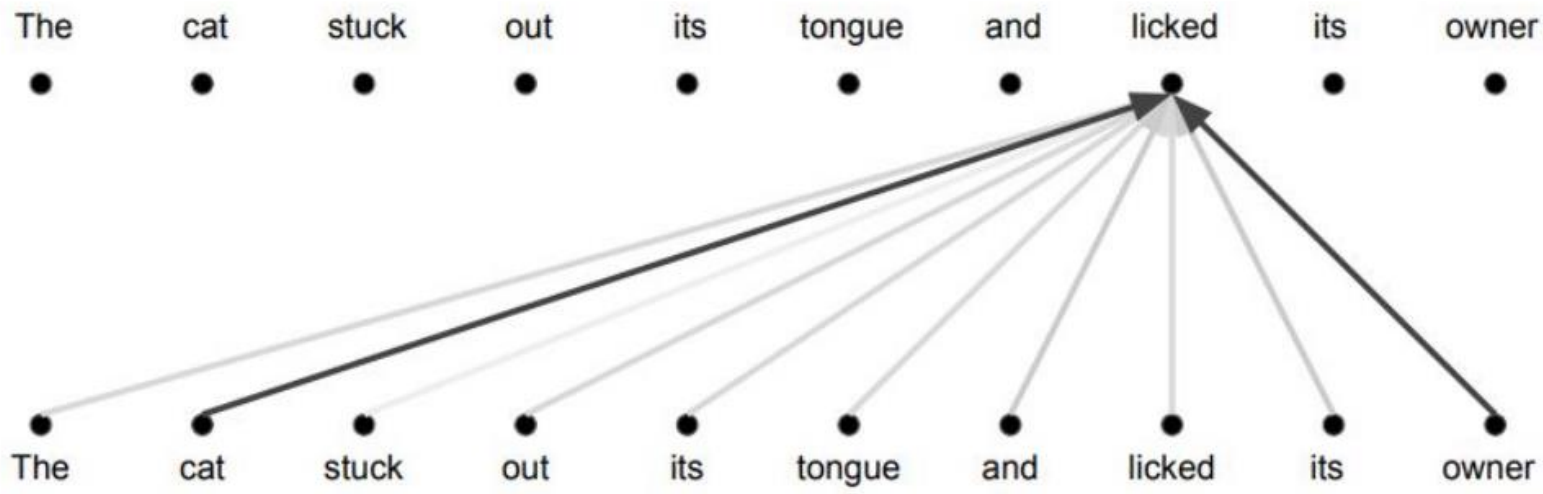
(Vaswani et. al 2017)

Encoder →

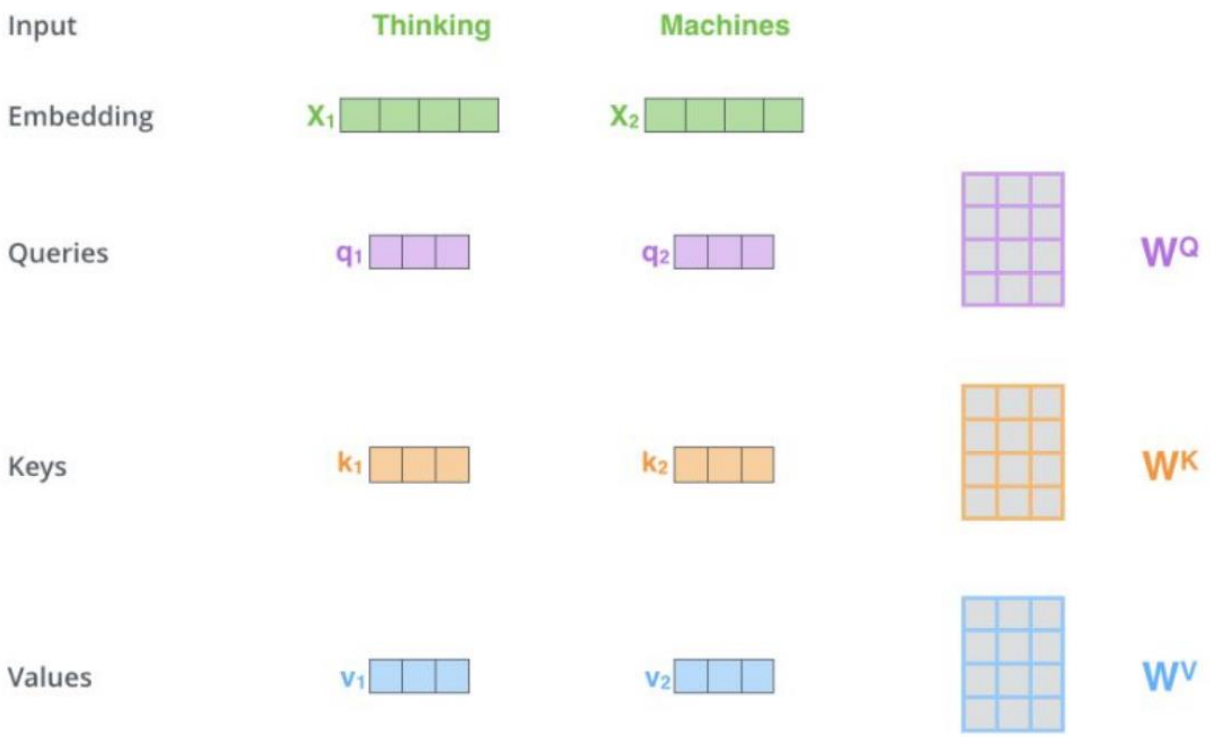


← Decoder

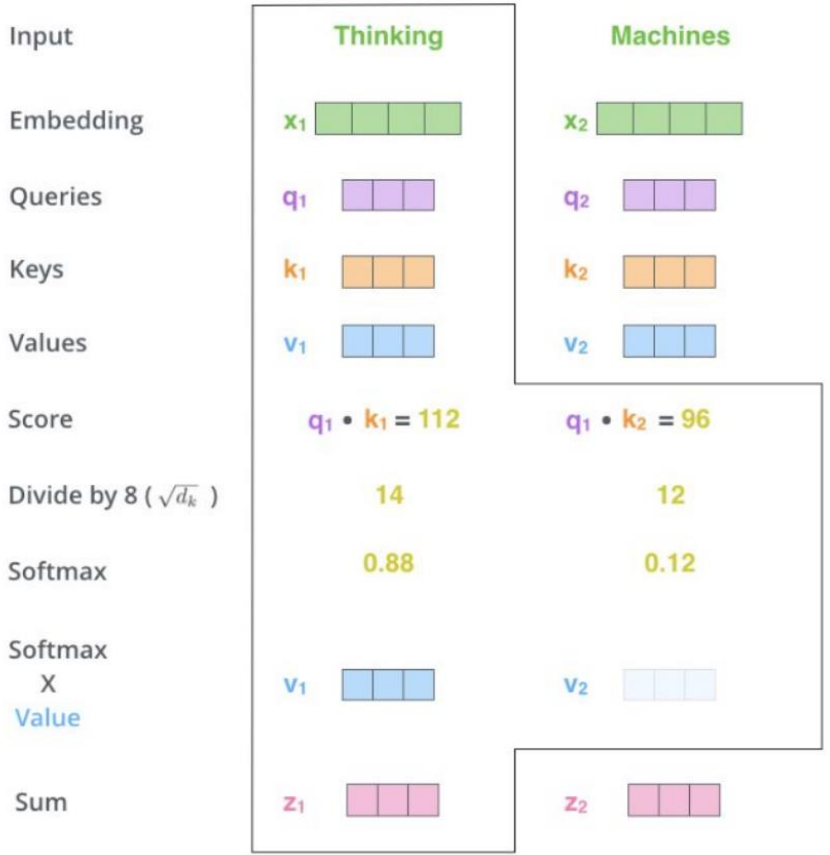
Self-Attention



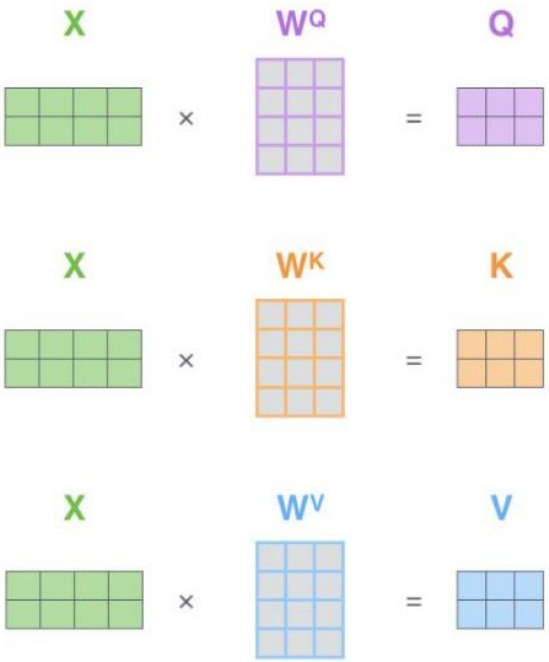
Self-Attention



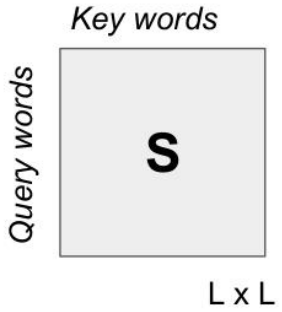
Self-Attention



Self-Attention

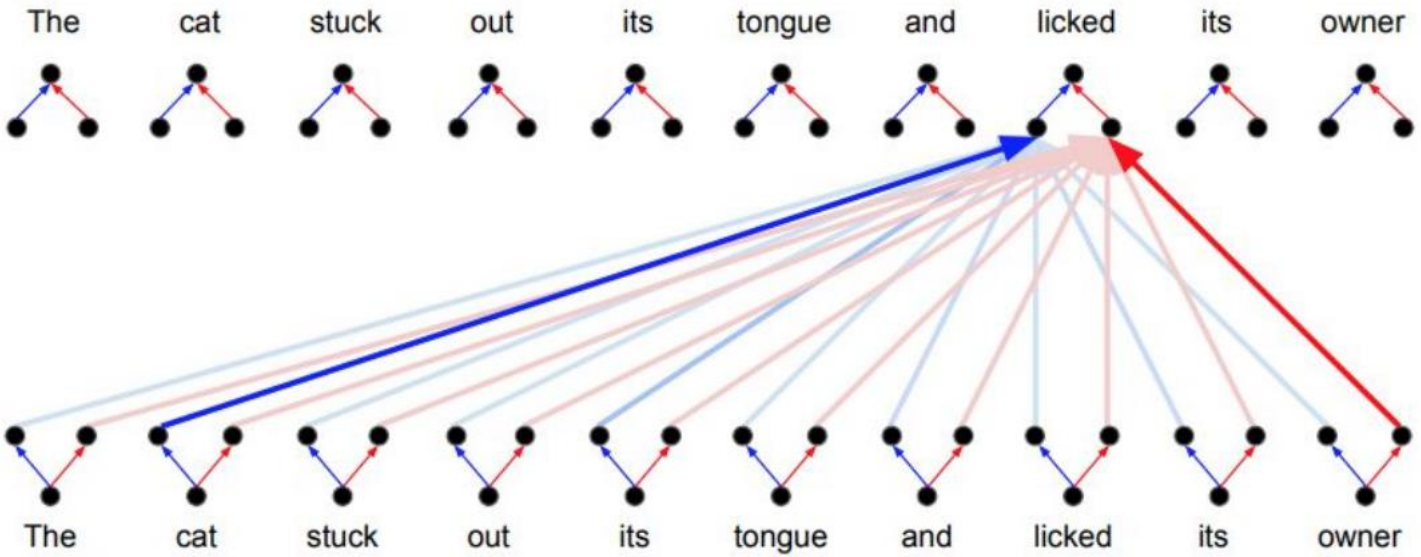


$$A(Q,K,V) = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

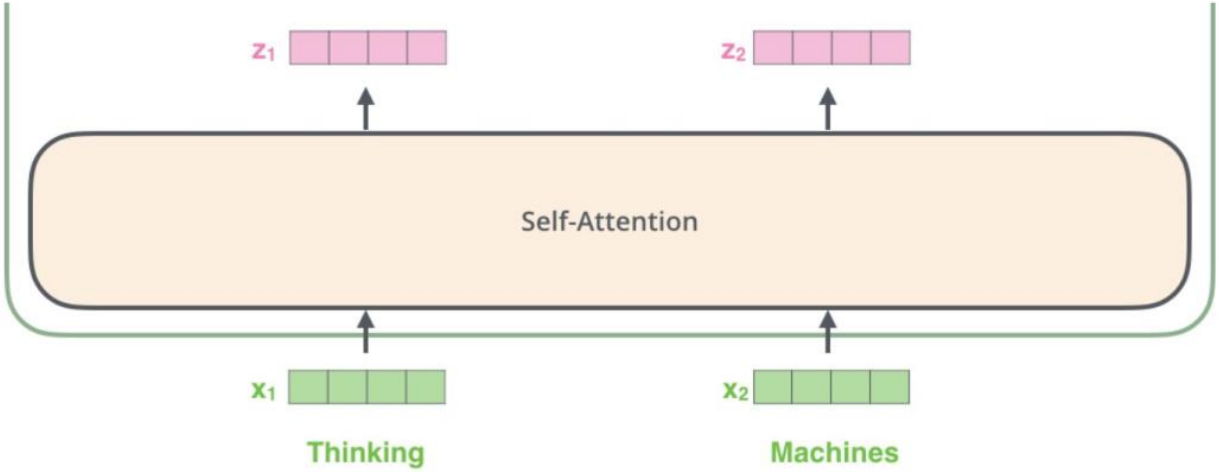


Multi-Head Self-Attention

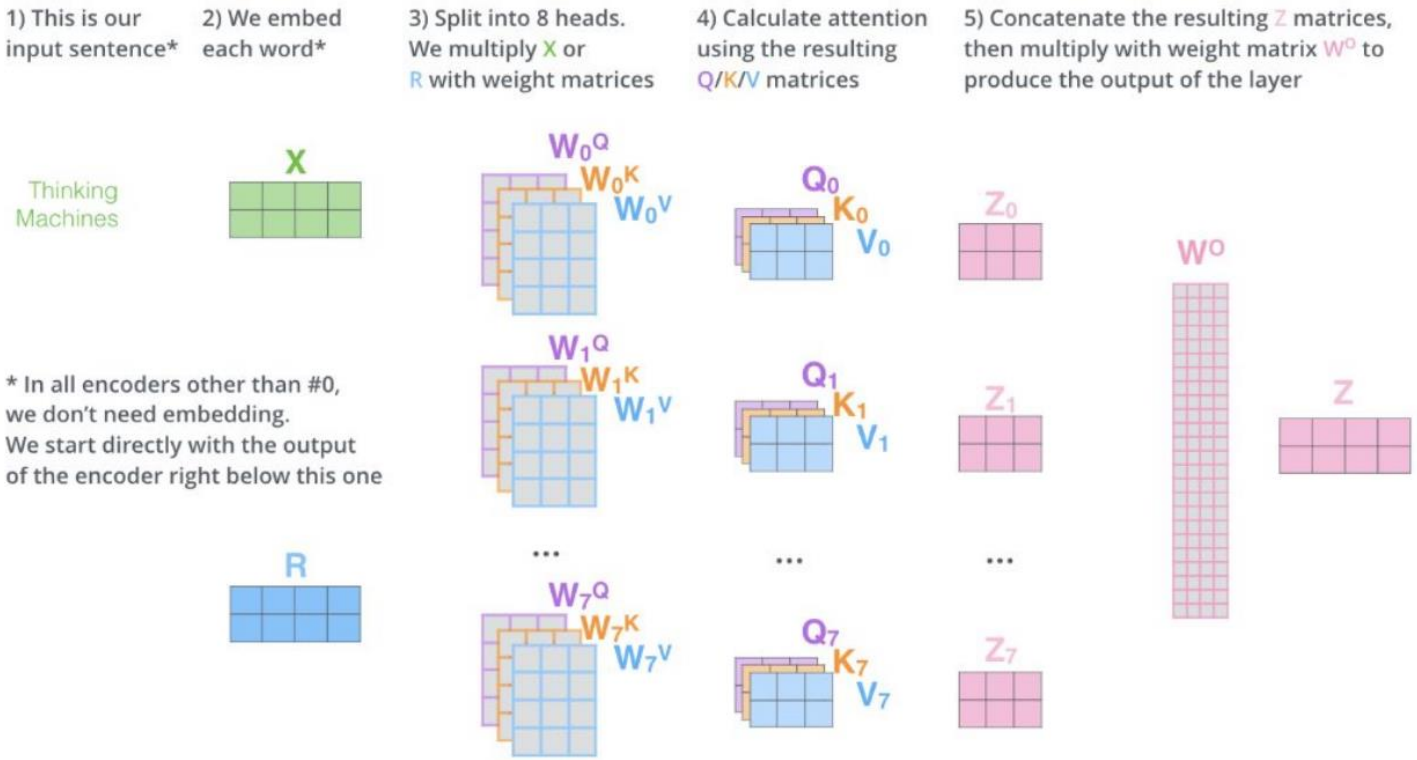
Parallel attention layers with different linear transformations on input and output.



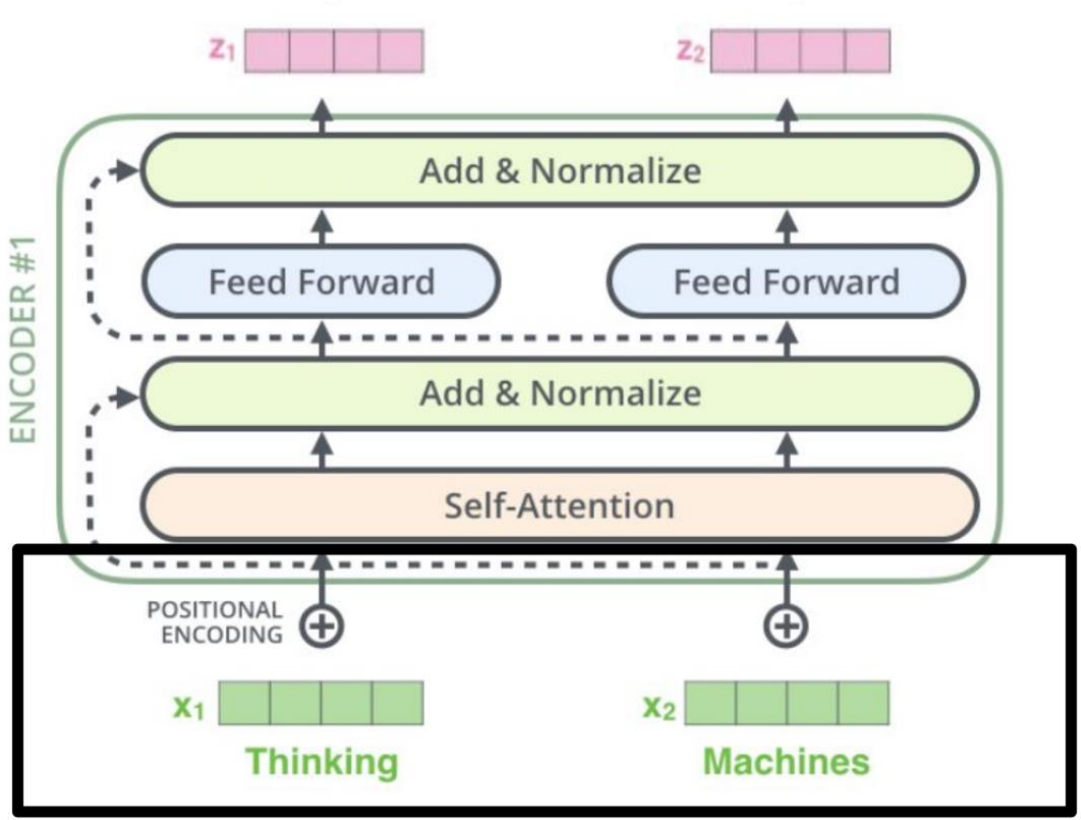
Retaining Hidden State Size



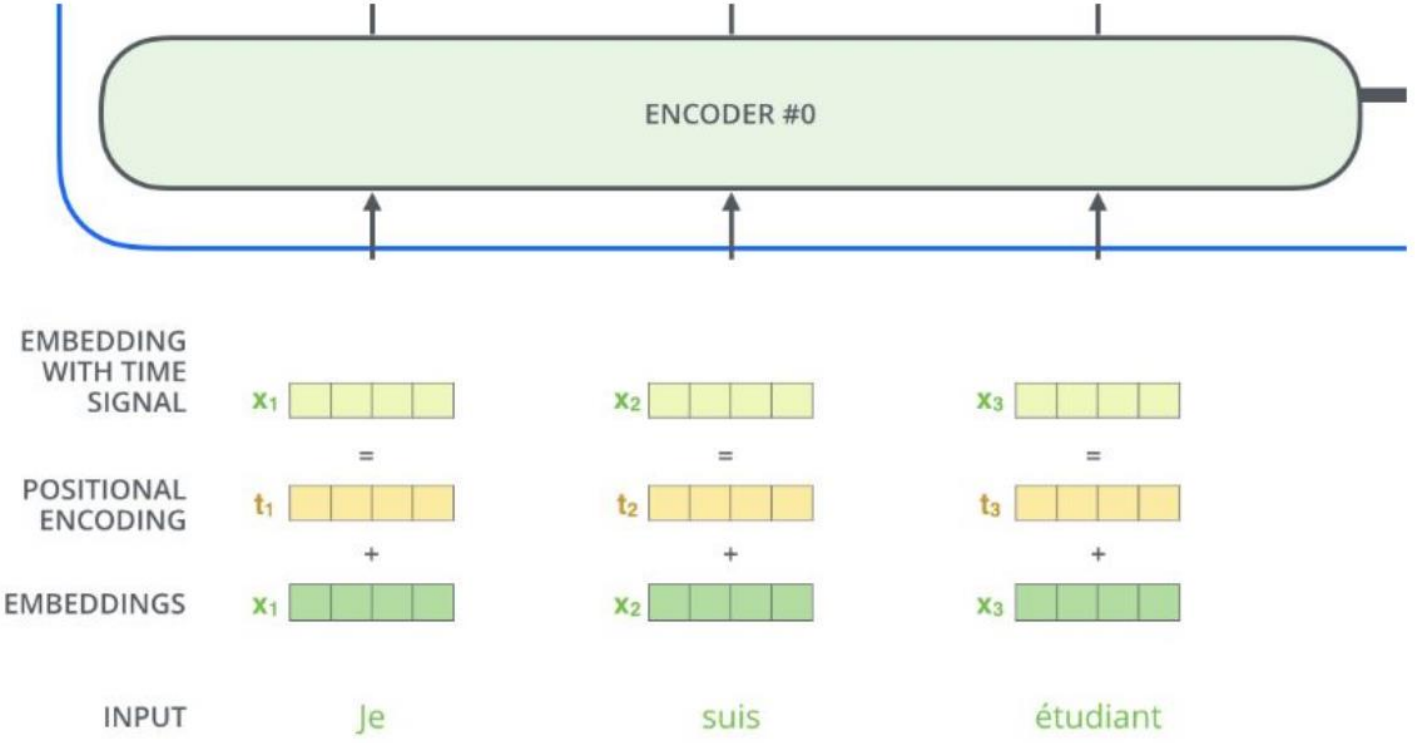
Details of Each Attention Sub-Layer of Transformer Encoder



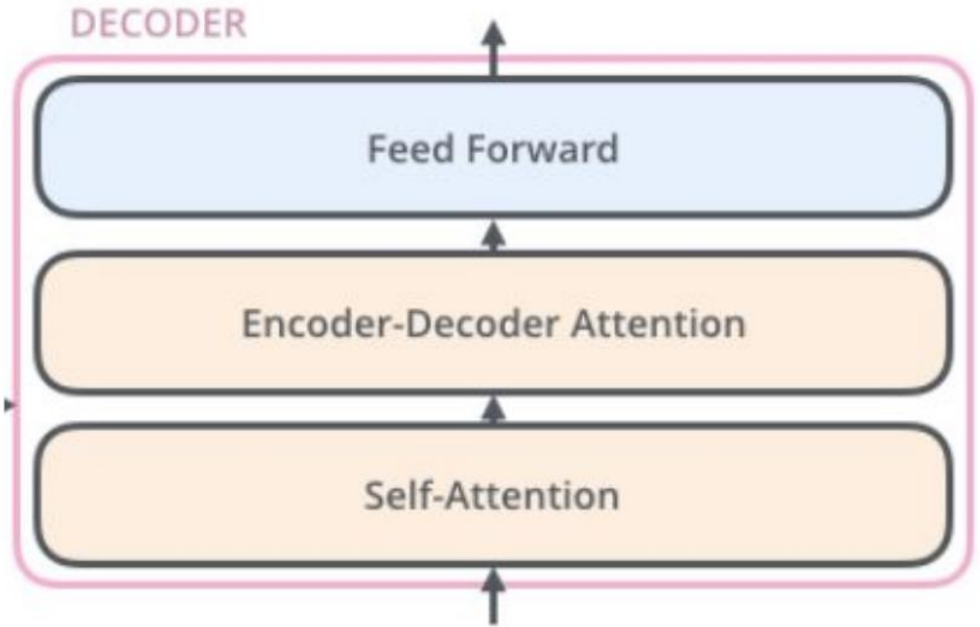
Each Layer of Transformer Encoder



Positional Encoding

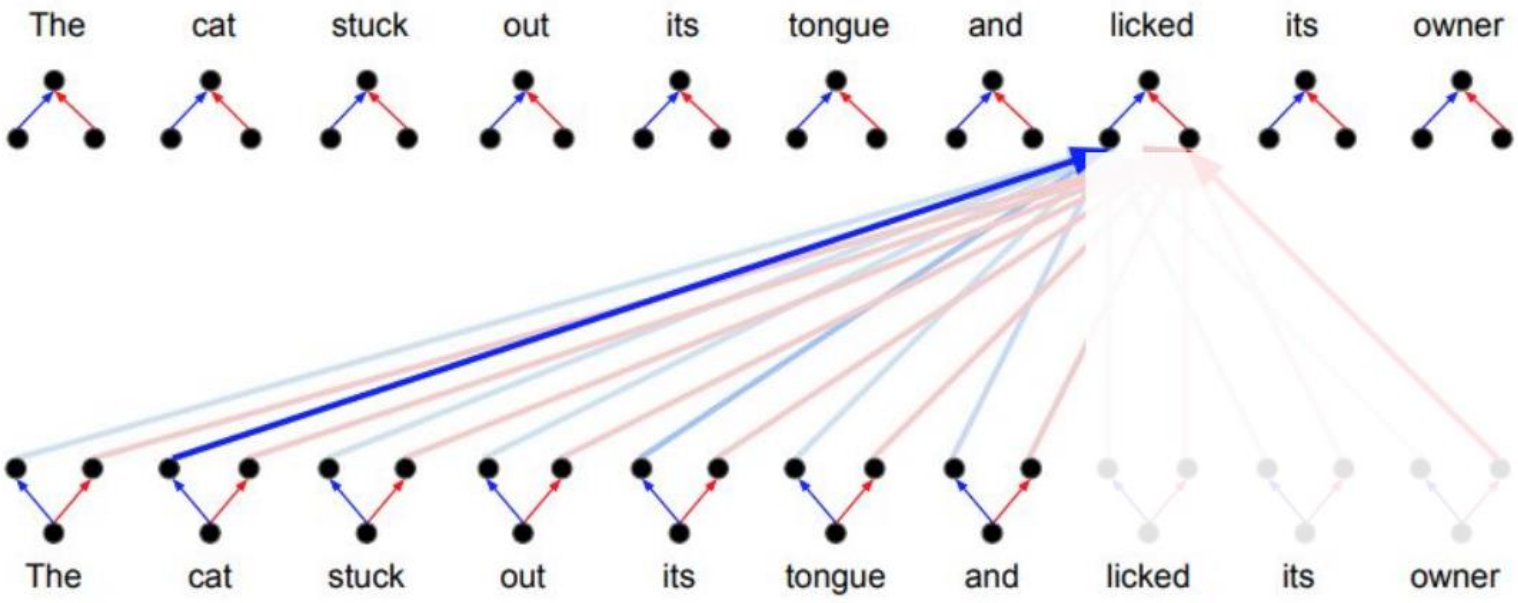


Each Layer of Transformer Decoder



Transformer Decoder - Masked Multi-Head Attention

Problem of Encoder self-attention: we can’t see the future !

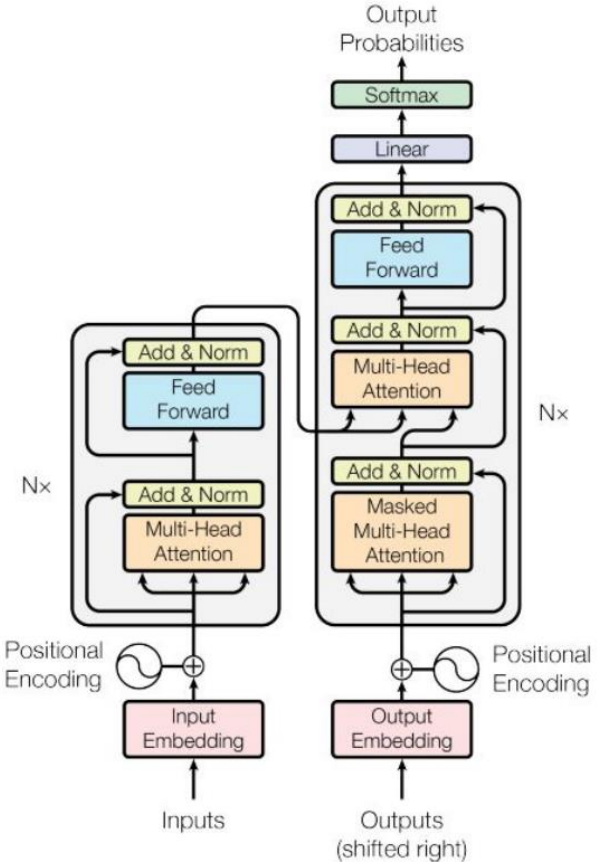


Transformer

“Attention is All You Need”

(Vaswani et. al 2017)

Encoder →



← Decoder

Outline

- Deep Learning
  - CNN
  - RNN
  - Attention
  - Transformer
- **Pytorch**
  - **Introduction**
  - **Basics**
  - **Examples**

What is PyTorch

- Open source ML Library developed by Facebook AI Research
- Enables us to easily leverage GPUs
- Provides automatic differentiation (autograd)
- Lets us easily and quickly build/test neural networks
    - Or other gradient-based ML ideas

Why PyTorch?

- It is pythonic-- concise, close to Python conventions
- Strong GPU support
- Autograd-- automatic differentiation
- Many algorithms and components are already implemented
- Similar to NumPy

Installing PyTorch

Installation

Via Anaconda/Miniconda:

```
condainstall pytorch-c pytorch
```

Via pip:

```
pip3 install torch
```

iPython Notebook Tutorial

http://bit.ly/pytorchbasics

Tensors

- Tensors are similar to NumPy’s ndarrays
  - But-- Tensors can also be used on a GPU to accelerate computing.
- Common operations for creation and manipulation of these Tensors are similar to those for ndarrays in NumPy.
  - `torch.rand`
  - `torch.ones`
  - `torch.zeros`
  - `sample = torch.tensor([1, 2, 3])`
    - sample[2] == 3
    - sample[:1] == torch.tensor([1, 2])
  - reshaping with `torch.view`
  - multiplication, addition, scalar/matrix math all similar to NumPy

Tensor data & gradients

- Attributes of a tensor 't':
  - `t= torch.randn(1)`
- requires_grad - making a trainable parameter
  - By default False
  - Turn on with:
    - `t.requires_grad_()`
    - `t = torch.randn(1, requires_grad=True)`
- Accessing tensor value:
  - `t.data`
- Accessing tensor gradient
  - `t.grad`
- grad_fn- history of operations for autograd
  - `t.grad_fn`

Loading Data, Devices and CUDA

- NumPy to PyTorch:
  - `sample = torch.from_numpy(numpy_data)`
    - Warning-- CPU by default!
- PyTorch to NumPy:
  - numpy_data = sample.numpy()
- Moving to GPU or CPU:
  - `sample.to(device='cuda')`
  - `sample.to(device='cpu')` or `sample.cpu()`
- Is the GPU available and is cuda installed?
  - `torch.cuda.is_available()`
- Check a tensor's device:
  - `sample.device`

Autograd

- Automatic Differentiation Package
- Don’t need to worry about partial differentiation, chain rule etc.
  - `backward()` does that
- Gradients are accumulated for each step by default:
  - Need to zero out gradients after each update
  - `tensor.grad_zero()`

Optimizer and Loss

Optimizer

- Adam, SGD etc.
- An optimizer takes:
  - the parameters we want to update,
  - the learning rate we want to use
  - other optimizer-specific hyper-parameters
- and performs the updates

```
output = model(input_data)
loss_value = loss_function(output, labels)
optimizer.zero_grad()
loss_value.backward()
optimizer.step()
```

Loss

- Various predefined loss functions to choose from
  - L1, MSE, Cross Entropy

Model

In PyTorch, a model is represented by a regular Python class that inherits from the Module class.

- Two components
  - `__init__(self):`
    - This defines the parts that make up the model- in our case, two parameters, a and b
  - `forward(self, x):`
    - This performs the actual computation, that is, it outputs a prediction, given the input x

```
class ManualRegression(nn.Module):
    def __init__(self):
        super().__init__()
        self.a = nn.Parameter(torch.randn(1, requires_grad=True, dtype=torch.float))
        self.b = nn.Parameter(torch.randn(1, requires_grad=True, dtype=torch.float))

    def forward(self, x):
        prediction = self.a + self.b * x
        return prediction
```

Design Model

- Initilaize modules.
- Use linear layer here.
  - Can change it to RNN, CNN, Transformer etc.
- Randomly initilaize parameters
  - Not strictly necessary
- Foward pass

```
import torch.nn as nn
import torch.nn.functional as F
class TextSentiment(nn.Module):
    def __init__(self, vocab_size, embed_dim, num_class): #Initilaize modules.
        super().__init__()
        self.embedding = nn.EmbeddingBag(vocab_size, embed_dim, sparse=True)
        # This is equivalent to nn.Embedding followed by torch.mean(dim=0)
        self.fc = nn.Linear(embed_dim, num_class) #Use linear layer here.
        self.init_weights()

    def init_weights(self):  # Randomly initilaize parameters
        initrange = 0.5
        self.embedding.weight.data.uniform_(-initrange, initrange) # Uniform distribution
        self.fc.weight.data.uniform_(-initrange, initrange)
        self.fc.bias.data.zero_()  # Make bias zeros

    def forward(self, text, offsets): # Foward pass
        embedded = self.embedding(text, offsets)
        # input (N,)  it will be treated as a concatenation of multiple bags (sequences).
        # offsets is required to be a 1D tensor containing the starting index positions of
        # Therefore, for offsets of shape (B), input will be viewed as having B bags.
        # ouput (B, embed_dim)
        return self.fc(embedded) # ouput (B, num_class)
```

Pre-Process Data

- Load in the dataset

- Build vocabulary

```
import torch
import torchtext
from torchtext.datasets import text_classification
NGRAMS = 1
import os
if not os.path.isdir('./.data'):
 os.mkdir('./.data')
train_dataset, test_dataset = text_classification.DATASETS['AG_NEWS'](
   root='./.data', ngrams=NGRAMS, vocab=None)
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

```
VOCAB_SIZE = len(train_dataset.get_vocab())
EMBED_DIM = 32
NUM_CLASS = len(train_dataset.get_labels())
model = TextSentiment(VOCAB_SIZE, EMBED_DIM, NUM_CLASS).to(device)
```

Batch the data

- Create a batch of data
  - Input sample + label
- Padding
  - Is it necessary?
  - How to do it?

```
def generate_batch(batch):
    # Input: a iterator of items with length of batch_size. For example:[(1,(tensor
    # Generate a batch used in SGD
    label = torch.tensor([entry[0] for entry in batch]) #tensor of shape (batch_siz
    text = [entry[1] for entry in batch]
    offsets = [0] + [len(entry) for entry in text]
    # torch.Tensor.cumsum returns the cumulative sum
    # of elements in the dimension dim.
    # torch.Tensor([1.0, 2.0, 3.0]).cumsum(dim=0)

    offsets = torch.tensor(offsets[:-1]).cumsum(dim=0) # For example tensor([0,3])
    text = torch.cat(text) # a list of tensor -> tensor of shape (sum([len(i) for i
    return text, offsets, label
```

Epochs of Training

- DataLoader creates our batches
- Zero optimizer gradients
- Forward pass

- Backprop gradients
- Update parameters

- Learning rate decay
  - Optional

```
from torch.utils.data import DataLoader
BATCH_SIZE = 16

def train_func(sub_train_):

    # Train the model
    train_loss = 0
    train_acc = 0
    data = DataLoader(sub_train_, batch_size=BATCH_SIZE, shuffle=True,
                      collate_fn=generate_batch)  # Iterable batches
    for i, (text, offsets, cls) in enumerate(data):
        optimizer.zero_grad() # Before each optimization, make previous gradients z
        text, offsets, cls = text.to(device), offsets.to(device), cls.to(device)
        output = model(text, offsets)
        loss = criterion(output, cls) # Forward pass to compute loss
        train_loss += loss.item()
        # Extract the number from a tensor containing only one item, this number wi
        loss.backward() # Backforward propagation to compute gradients of each vari
        optimizer.step() # Update parameters according to gradients
        #choose the class with the highest score as current prediction and compare
        train_acc += (output.argmax(1) == cls).sum().item()

    # Adjust the learning rate. After each epoch, do learning rate decay ( optional
    scheduler.step()

    return train_loss / len(sub_train_), train_acc / len(sub_train_) #return averag
```

Testing on new data

We do not backprop loss or calculate gradients on test data!!!

Training on the test set is cheating, and you should take every precaution to prevent training data from “leaking” into test sets

```
def test(data_):
    #Similar to train_func but do not need back propagation or parameter update
    loss = 0
    acc = 0
    data = DataLoader(data_, batch_size=BATCH_SIZE, collate_fn=generate_batch)
    for text, offsets, cls in data:
        text, offsets, cls = text.to(device), offsets.to(device), cls.to(device
        with torch.no_grad(): # prevent computing gradient, could not use backw
            output = model(text, offsets)
            loss = criterion(output, cls)
            loss += loss.item()
            acc += (output.argmax(1) == cls).sum().item()

    return loss / len(data_), acc / len(data_)
```

Training Loop

- CrossEntropyLoss
- SGD Optimizer
- Exponential Decay on the learning rate
- Split dataset into train/dev sets
- Train over training data
- Test on validation data

```
import time
from torch.utils.data.dataset import random_split
N_EPOCHS = 5
min_valid_loss = float('inf')
criterion = torch.nn.CrossEntropyLoss().to(device)
optimizer = torch.optim.SGD(model.parameters(), lr=4.0)
scheduler = torch.optim.lr_scheduler.StepLR(optimizer, 1, gamma=0.9)

train_len = int(len(train_dataset) * 0.95)
#Split whole training dataset to create validation (hold-out datset)
sub_train_, sub_valid_ = \
   random_split(train_dataset, [train_len, len(train_dataset) - train_len])
for epoch in range(N_EPOCHS):

   start_time = time.time()
   train_loss, train_acc = train_func(sub_train_)
   valid_loss, valid_acc = test(sub_valid_)

   secs = int(time.time() - start_time)
   mins = secs / 60
   secs = secs % 60

   #Print information to monitor the training process
   print('Epoch: %d' %(epoch + 1), " | time in %d minutes, %d seconds" %(mins,
   print(f'\tLoss: {train_loss:.4f}(train)\t|\tAcc: {train_acc * 100:.1f}%(tra
   print(f'\tLoss: {valid_loss:.4f}(valid)\t|\tAcc: {valid_acc * 100:.1f}%(val
```