PyTorch for NLP

Jay Urbain, PhD
jay.urbain@gmail.com
https://www.linkedin.com/in/jayurbain/

Credits:

https://github.com/jayurbain/DeepNLPIntro

https://pytorch.org/tutorials/

https://github.com/joosthub/PyTorchNLPBook

https://github.com/rouseguy/DeepLearning-NLP

https://github.com/scoutbee/pytorch-nlp-notebooks

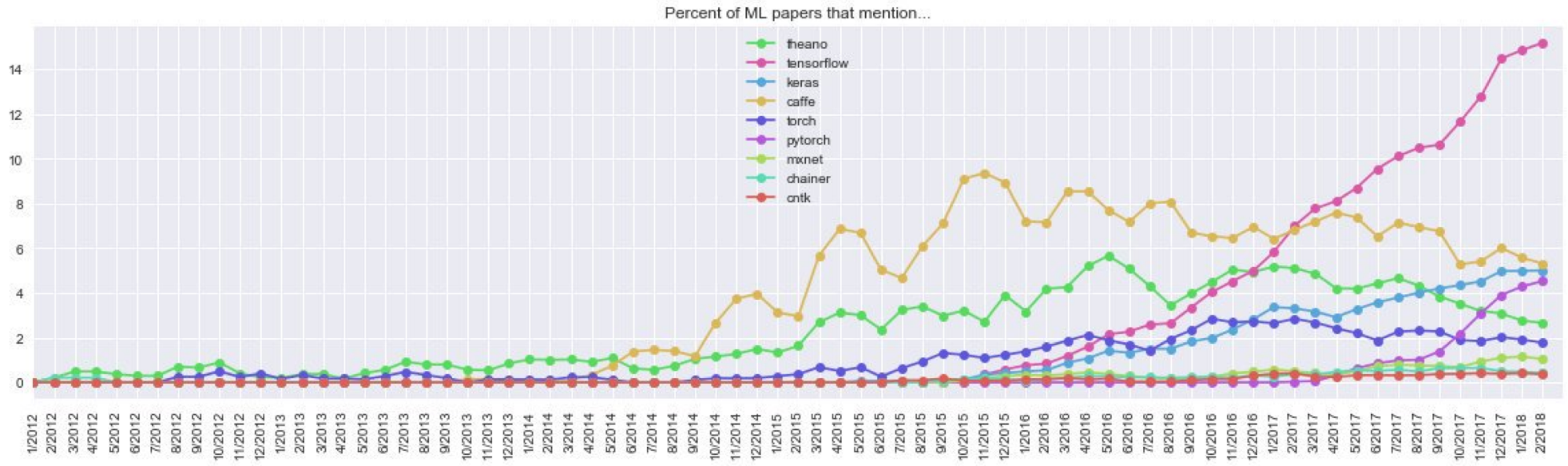
Many more …

Outline

- **Introducing PyTorch**
  - Static vs. dynamic computation graphs
  - PyTorch basic operations
- **Training Networks**
  - Training flow
  - Loss functions
  - Gradient Descent
  - Detailed PyTorch patterns
- **NLP Basics**
  - Examples of DL-NLP
  - Text preprocessing
  - Text representation
- **Embeddings**
  - See previous slides
- **RNNs**
  - How RNNs work
  - Vanishing & Exploding Gradient Problem
  - Types of RNNs
  - Char-RNN text generation example
  - Sentiment classification example
- **Sequence Models**
  - Seq2Seq
  - Attention
  - Transformer
- **Transfer Learning**
  - Why it's useful
  - GPT-2

PyTorch Intro

Academic popularity of DL libraries over time



Source: Karpathy's twitter

Note: NAACL 2019 PyTorch Popularity

Goals of a deep learning library

1) Define a model, loss function, and learning rate optimizer (define the computational graph)
2) Support automatic differentiation (i.e., back-propagation)

Static vs. Dynamic Computational Graphs

Static

- "define-then-run"
- Define the computational graph and then feed data to it
- Easier to distribute over multiple machines
- Ability to ship models independent of code
- More complex to code and debug

Dynamic

- "define-by-run"
- Computational graph is defined on-the-fly
- Ability to use a debugger to view data flow and check matrix shapes
- Can handle variable sequence lengths
- Easier to define some kinds of complex networks
- Easier to code and more *Pythonic*

Find a detailed discussion on the topic here
https://www.tensorflow.org/guide/eager

What is PyTorch?

Deep learning in Python

Numpy ndarrays with GPU support

Automatic differentiation

Optimization based on gradients

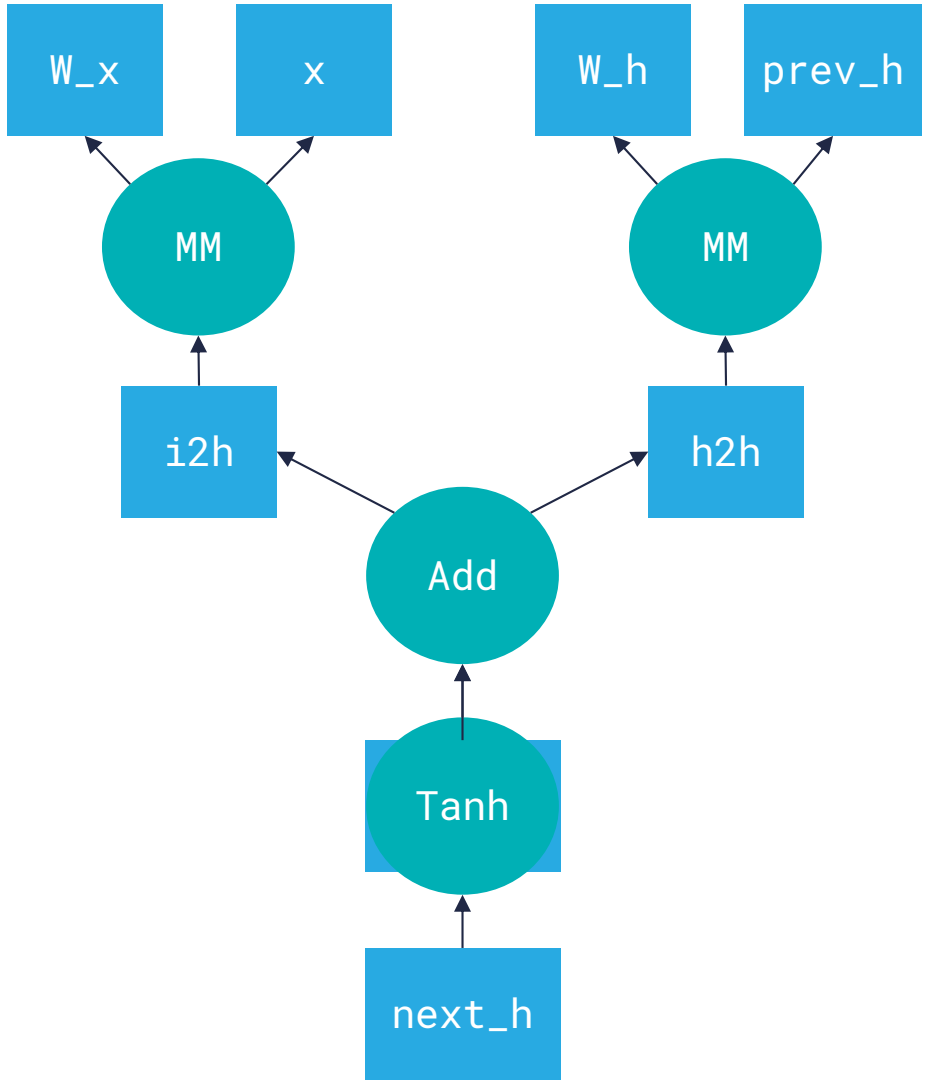
Dynamic computation graphs

```
In [1]: import torch

In [2]: torch.tensor([1, 2, 3])  # Create a tensor
Out[2]: tensor([1, 2, 3])

In [3]: _ * 2  # Broadcasting
Out[3]: tensor([2, 4, 6])

In [4]: _.to('cuda')  # Send the tensor to the GPU
Out[4]: tensor([2, 4, 6])
```

Computation graph is generated on the fly

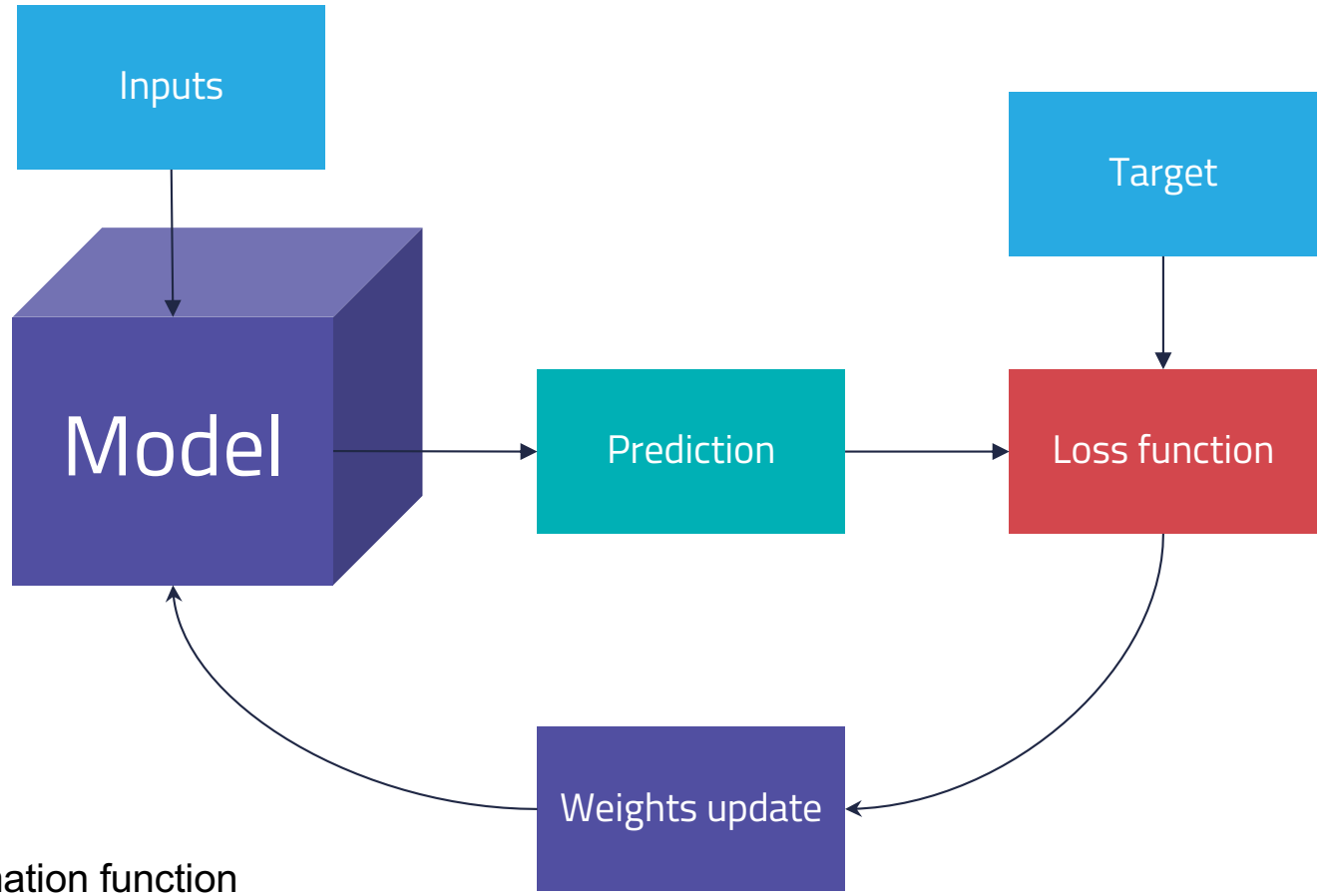
```
x = torch.randn(1, 10)
prev_h = torch.randn(1, 20)

W_h = torch.randn(20, 20)
W_x = torch.randn(20, 10)
W_h.requires_grad = True
W_x.requires_grad = True

i2h = torch.mm(W_x, x.t())
h2h = torch.mm(W_h, prev_h.t())

next_h = i2h + h2h
next_h = next_h.tanh()
```

```
next_h.backward(torch.ones(20, 1))
```



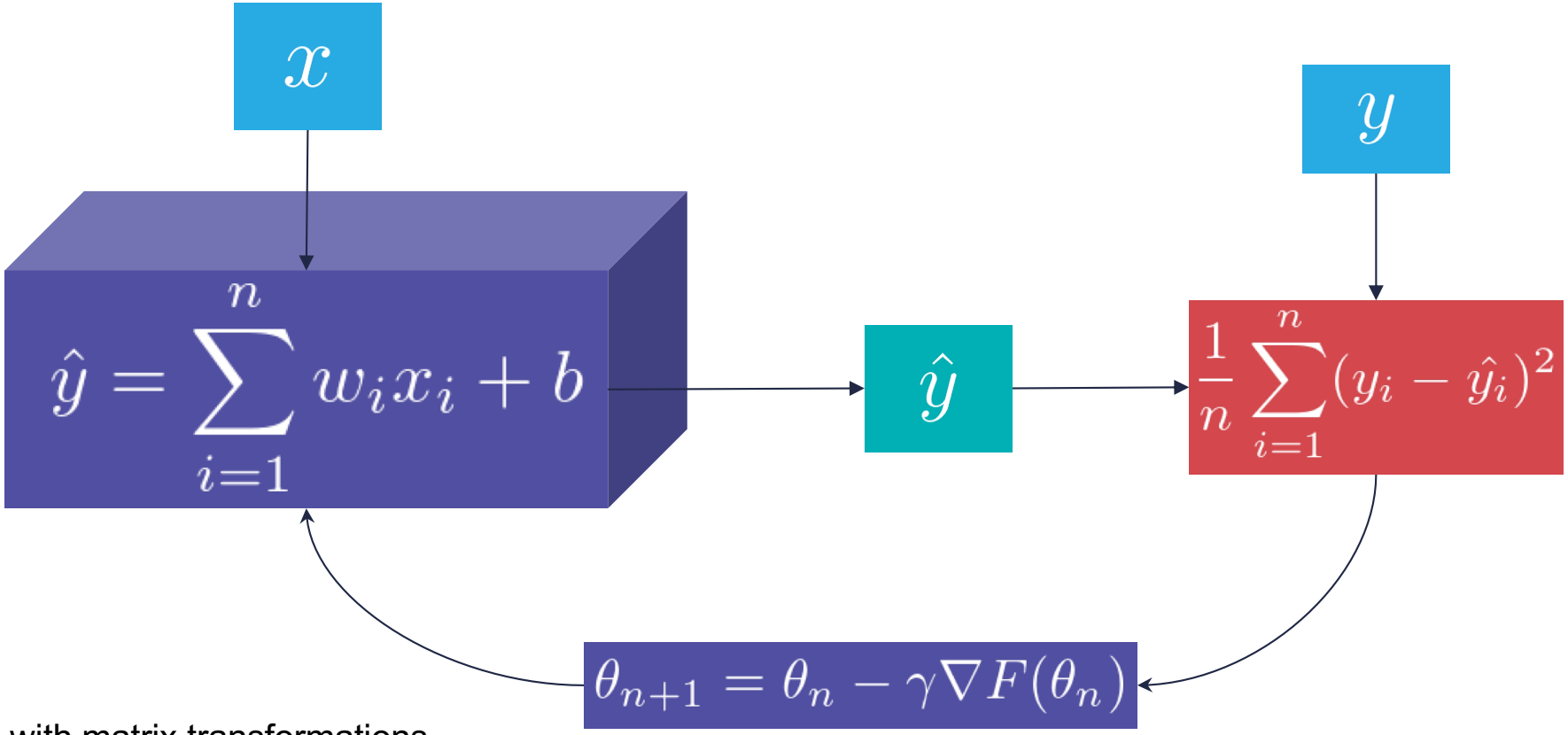
Training

Model training



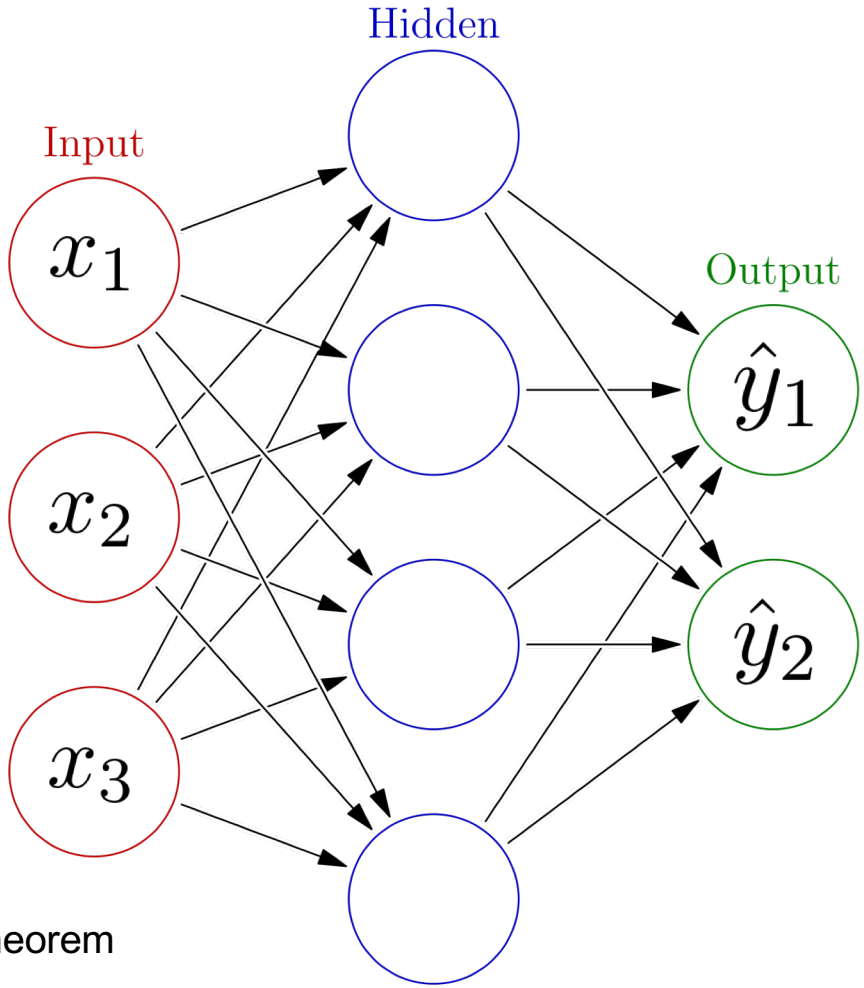
Model is a transformation function

Model training (in math)



Done with matrix transformations

Feed-forward network



Universal Approximation Theorem

Common loss functions

L1 Loss

$$\frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|$$

`nn.L1Loss`

Mean squared error (L2 Loss)

$$\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

`nn.MSELoss`

Negative log likelihood

$$-\sum_{i=1}^{n}\log \hat{y}_i$$

`nn.NLLLoss`

Cross entropy

$$\frac{1}{n}\sum_{i=1}^{n} y_i \cdot \log \hat{y}_i$$

`nn.CrossEntropyLoss`

Using a Loss Function in PyTorch

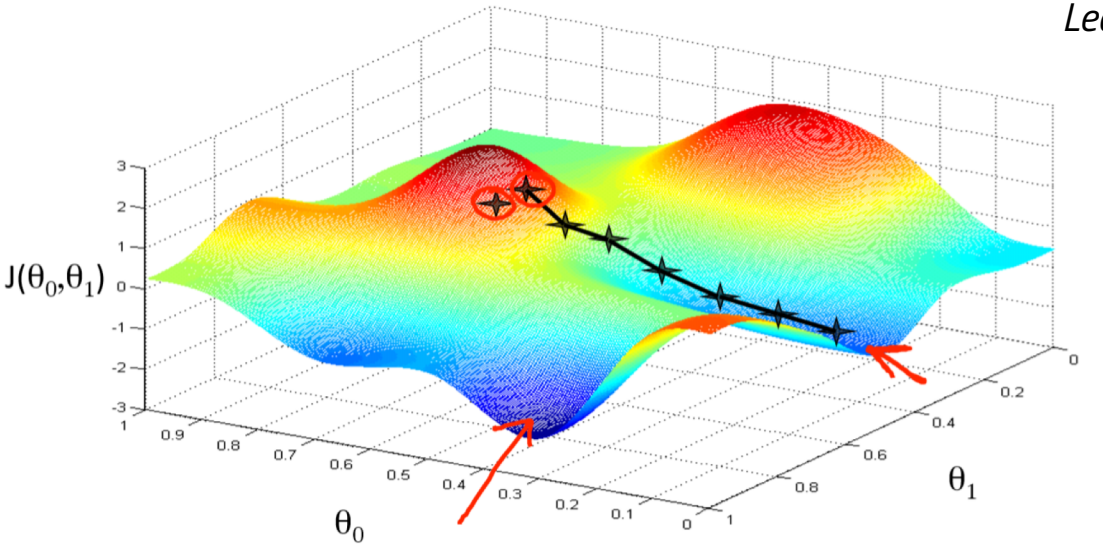
```
criterion = nn.CrossEntropyLoss()

# Calculate how wrong the model is
loss = criterion(prediction, target)
```

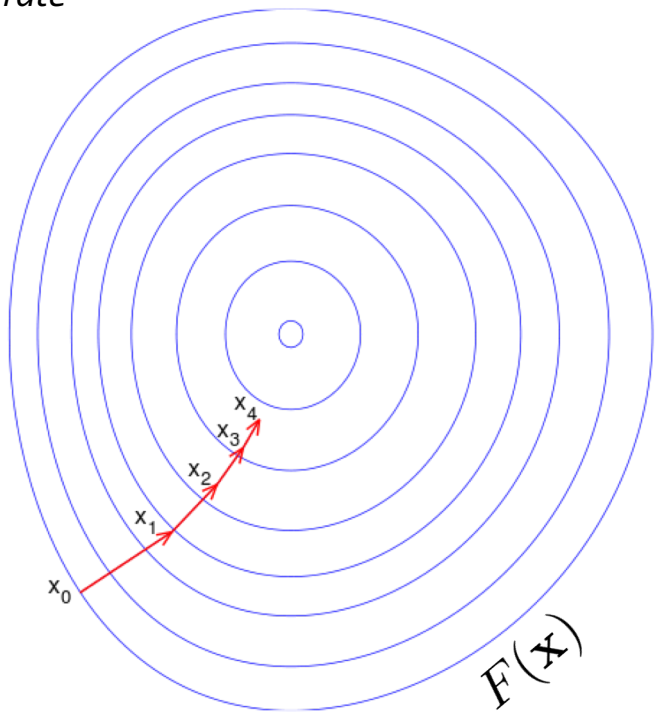
$\hat{y}$ y

Gradient descent

$$\theta_{n+1} = \theta_n - \gamma \Delta F(\theta_n)$$

Learning rate






```
# Perform gradient descent
loss.backward() # Backward pass
optimizer.step() # Update weights
```

PyTorch Dataset Class

```
class MyDataset(Dataset):
    def __init__(self):
        # Read your data file

        # Tokenize and clean text

        # Convert tokens to indices

    def __getitem__(self, i):
        return self.sequences[i], self.targets[i]

    def __len__(self):
        return len(self.sequences)

dataset = MyDataset()
```

Must implement!

PyTorch Model Definition

```
class MyClassifier(nn.Module):
    def __init__(self):
        super(MyClassifier, self).__init__()
        self.fc1 = nn.Linear(128, 32)
        self.fc2 = nn.Linear(32, 16)
        self.fc3 = nn.Linear(16, 1)

    def forward(self, inputs):
        x = F.relu(self.fc1(inputs))
        x = F.relu(self.fc2(x))
        x = F.sigmoid(self.fc3(x))
        return x


model = MyClassifier().to('cuda')
```

PyTorch Training Loop

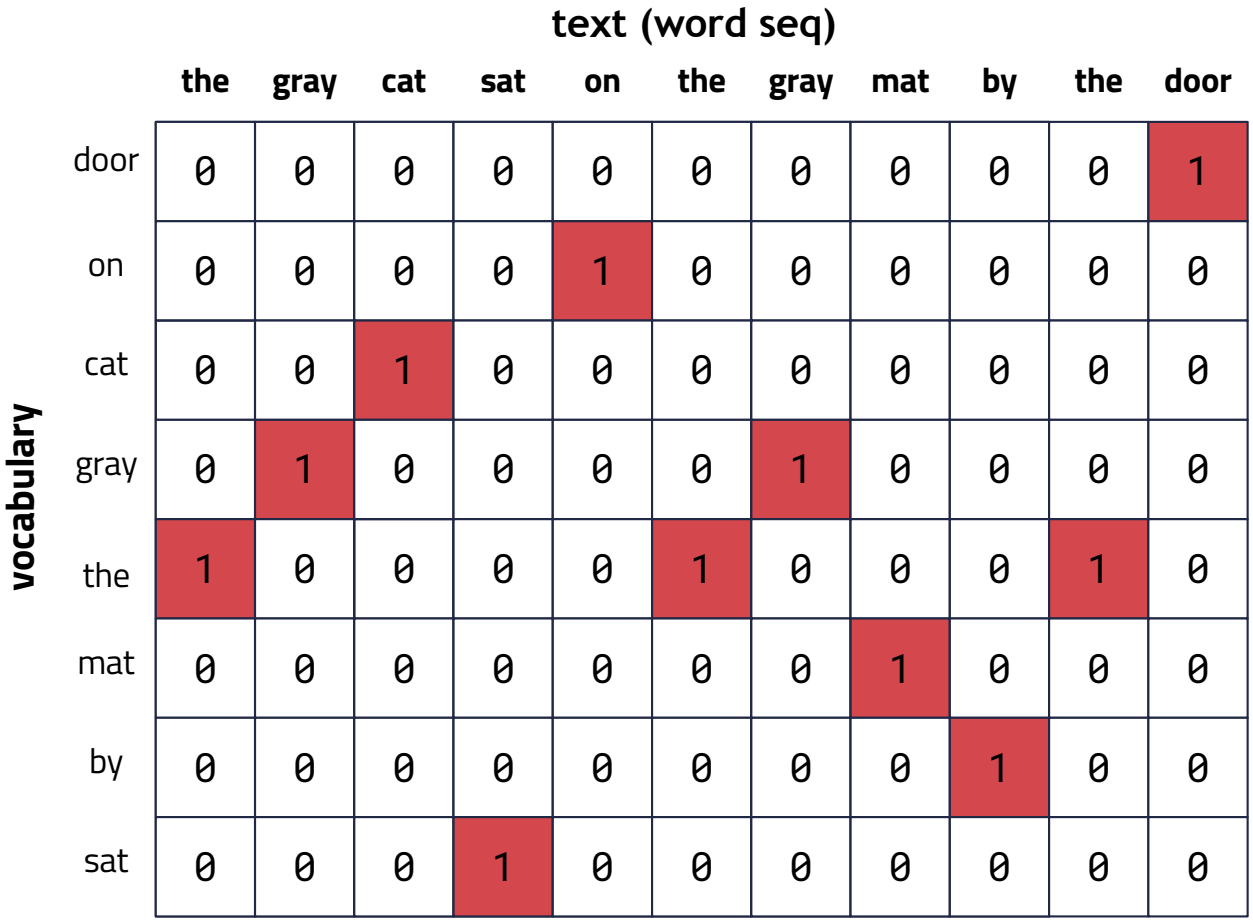
```
for epoch in range(n_epochs):
    for inputs, target in loader:
        # Clean old gradients
        optimizer.zero_grad()

        # Forwards pass
        output = model(inputs)

        # Calculate how wrong the model is
        loss = criterion(output, target)

        # Perform gradient descent, backwards pass
        loss.backward()

        # Take a step in the right direction
        optimizer.step()
```

NLP Basics for PyTorch

Text preprocessing for NLP tasks

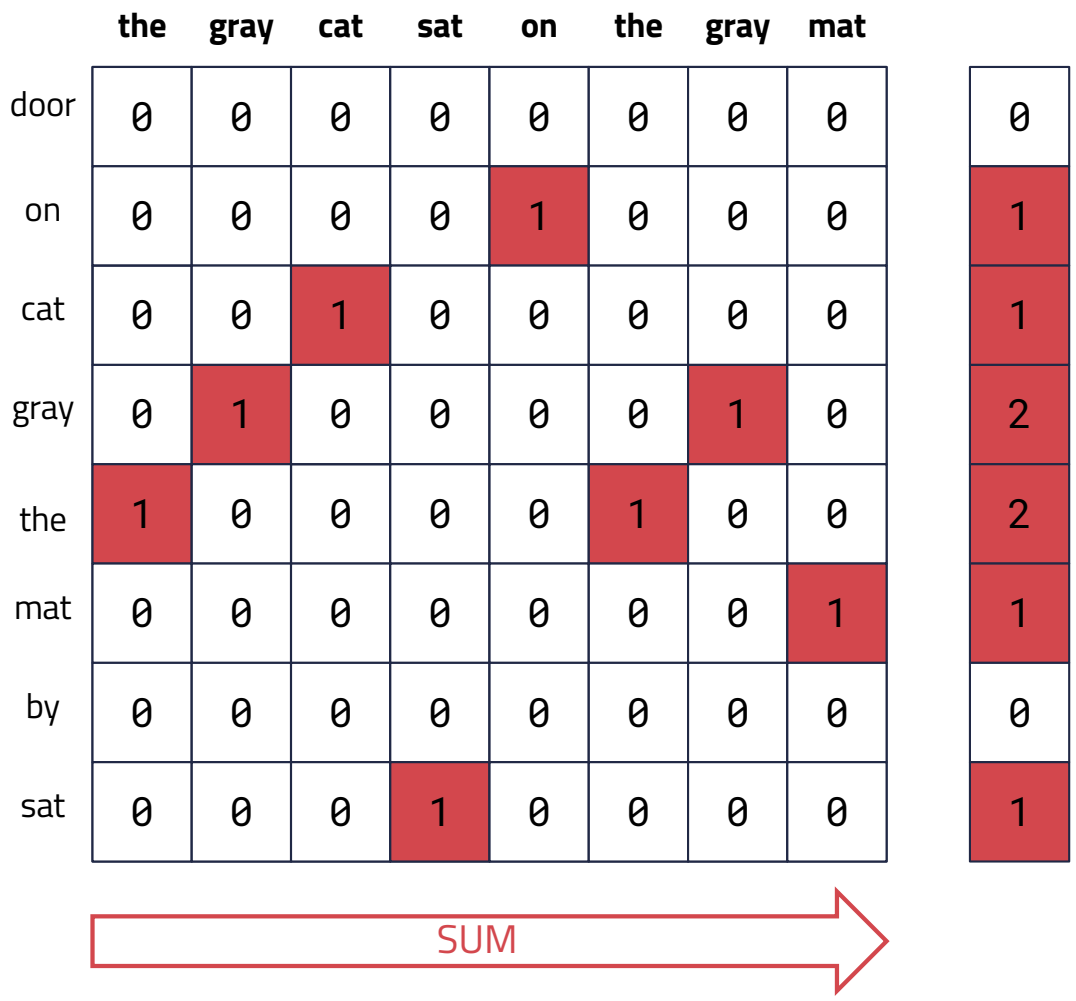
```
In [1]: tokenize('the sneaky fox jumped over the dog')
Out[1]: ['the', 'sneaky', 'fox', 'jumped', 'over', 'the', 'dog']

In [2]: remove_stop_words(_)
Out[2]: ['sneaky', 'fox', 'jumped', 'over', 'dog']

In [3]: lemmatize(_)
Out[3]: ['sneaky', 'fox', 'jump', 'over', 'dog']

In [4]: replace_rare_words(_)  # eg. WordPiece
Out[4]: ['###y', 'fox', 'jump', 'over', 'dog']
```

One-hot encoding



A **numerical representation** of text

Bag-of-words representation




```
from sklearn.feature_extraction.text
import CountVectorizer
```

```
{'on': 1,
'cat': 1,
'gray': 2,
'the': 2,
'mat': 1,
'sat': 1}
```

Bag-of-words feed forward network

Example #1

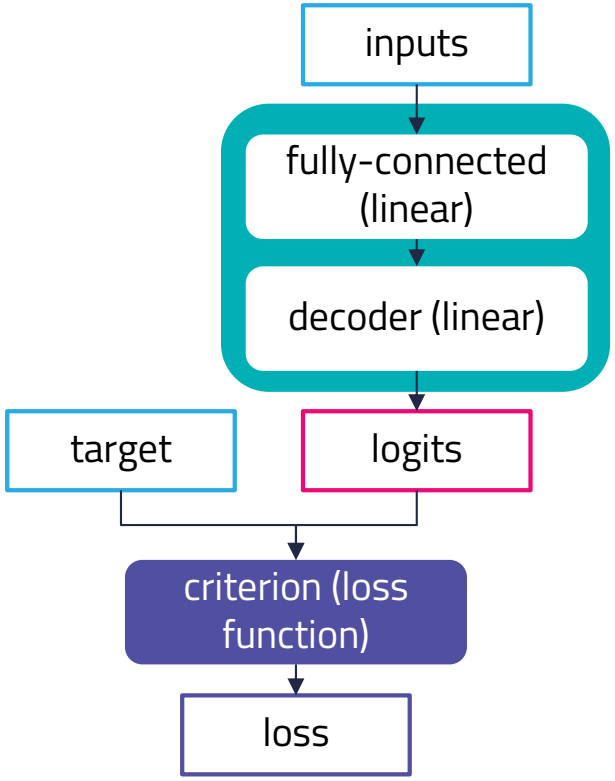
```
inputs:
{“good”:2,”movie”:1,”interesting”:1,”plot”:1}
target: 1
```

Example #2

```
inputs:
{“hated”:1,”experience”:1,”never”:1,”again”:1,”ho
rrible”:2,”film”:1}
target: 0
```

Example #3

```
inputs:
{“dramatic”:1,”twist”:1,”recommend”:3,”experience
”:2}
target: 1
```



Notebook:
Text Classification with Bag of Words

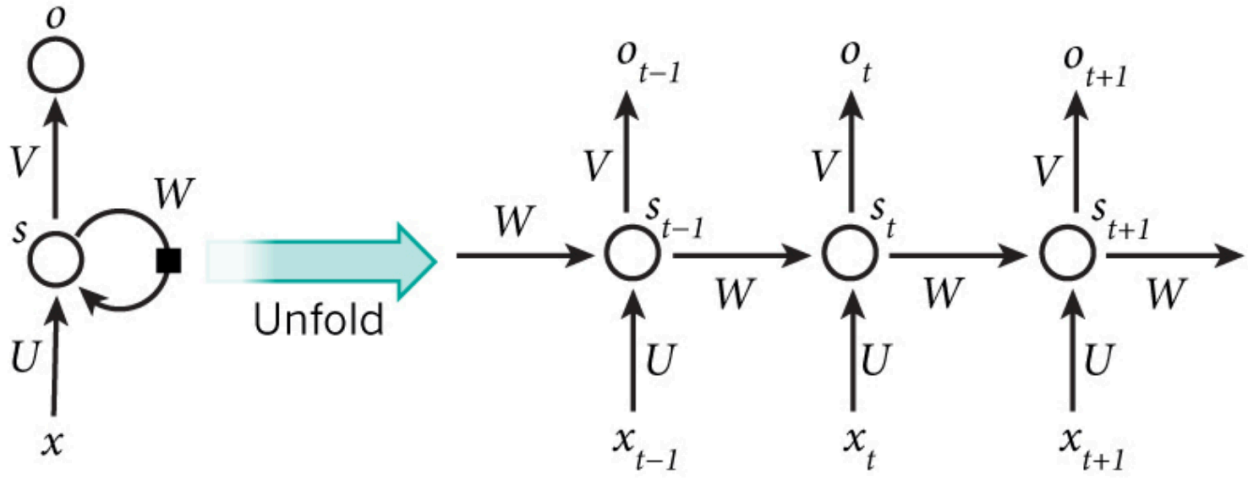
notebooks/PyTorch/1_text_classification_bow.ipynb

Recurrent Neural Networks

Recurrent Neural Network (RNN)

- RNNs have connections between units along a sequence.
- RNN's use the hidden state from the last time step and the input at the current step to make predictions.
- Allow RNN's can capture dynamic temporal behavior for a time sequence.

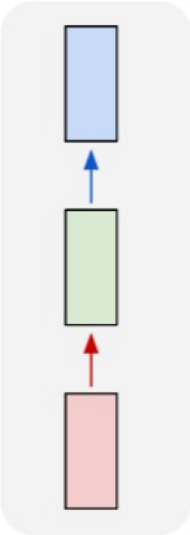
- Must run network for each step of the encoder and decoder – *slow*.
- Context from input words can have a long distance to travel.



$$h_t = f(h_{t-1}, x_t; \theta)$$

Types of RNNs

one to one



No RNN, image classification

one to many

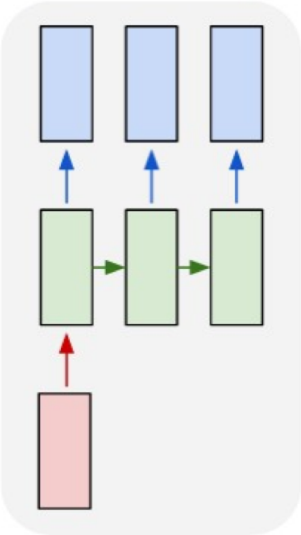
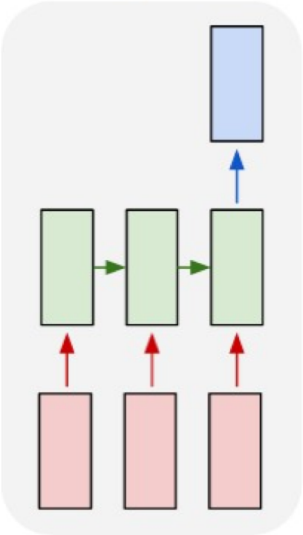


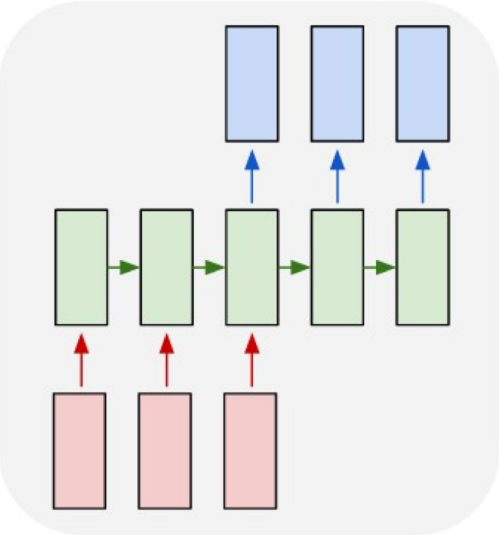
Image captioning

many to one



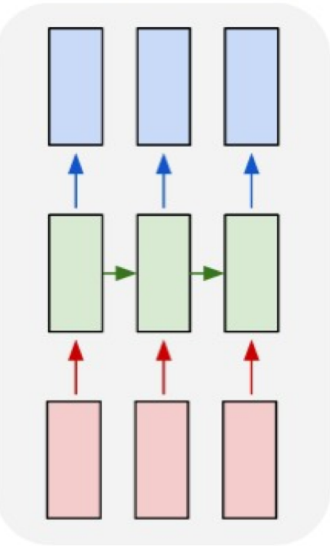
Sentiment classification

many to many



Machine translation, text generation

many to many



Video classification on a frame level

source

Vanishing error gradient

RNNs have difficulties learning long-range dependencies – interactions between words that are several steps apart.

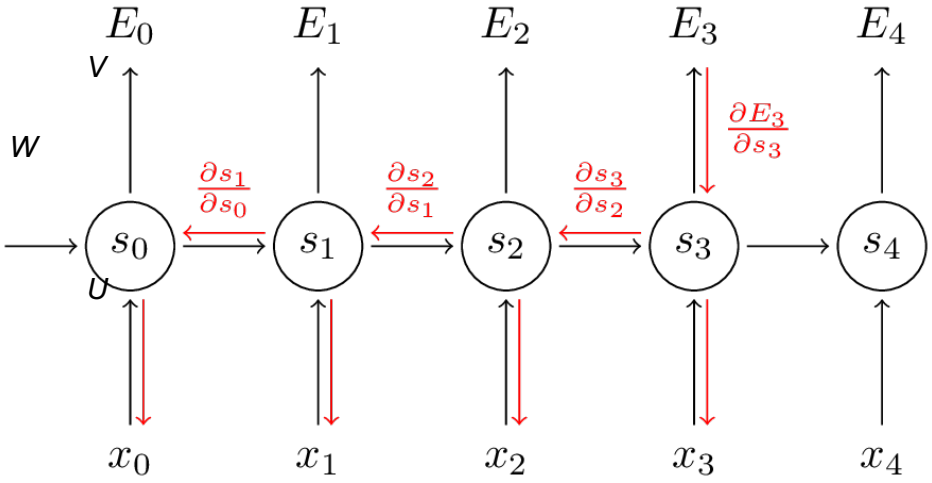
Problem - the meaning of an English sentence is often determined by words that aren't very close: *"The man who wore a wig on his head went inside".*

The sentence is really about a man going inside, not about the wig.

- Learning involves calculating the gradients of the Error *E* with respect to parameters (weights) *U, V,* & *W* using stochastic gradient descent.

$$\frac{\partial E_3}{\partial W} = \sum_{k=0}^{3} \frac{\partial E_3}{\partial \hat{y}_3} \frac{\partial \hat{y}_3}{\partial s_3} \frac{\partial s_3}{\partial s_k} \frac{\partial s_k}{\partial W}$$

$$\frac{\partial E_3}{\partial W} = \sum_{k=0}^{3} \frac{\partial E_3}{\partial \hat{y}_3} \frac{\partial \hat{y}_3}{\partial s_3} \left(\prod_{j=k+1}^{3} \frac{\partial s_j}{\partial s_{j-1}} \right) \frac{\partial s_k}{\partial W}$$



Backpropagation through Time

```
def bptt(self, x, y):
    T = len(y)
    # Perform forward propagation
    o, s = self.forward_propagation(x)
    # We accumulate the gradients in these variables
    dLdU = np.zeros(self.U.shape)
    dLdV = np.zeros(self.V.shape)
    dLdW = np.zeros(self.W.shape)
    delta_o = o
    delta_o[np.arange(len(y)), y] -= 1.
    # For each output backwards...
    for t in np.arange(T)[::-1]:
        dLdV += np.outer(delta_o[t], s[t].T)
        # Initial delta calculation: dL/dz
        delta_t = self.V.T.dot(delta_o[t]) * (1 - (s[t] ** 2))
        # Backpropagation through time (for at most self.bptt_truncate steps)
        for bptt_step in np.arange(max(0, t-self.bptt_truncate), t+1)[::-1]:
            # Add to gradients at each previous step
            dLdW += np.outer(delta_t, s[bptt_step-1])
            dLdU[:,x[bptt_step]] += delta_t
            # Update delta for next step dL/dz at t-1
            delta_t = self.W.T.dot(delta_t) * (1 - s[bptt_step-1] ** 2)
    return [dLdU, dLdV, dLdW]
```

RNN Cell Architectures

Simple RNN

- **Recurrent neural network**
- Simplest
- No gates
- Suffers from the **vanishing gradient problem** and the **exploding gradient problem** for sequences longer than ~4

`nn.RNN`

LSTM

- **Long short-term memory**
- Solves vanishing gradient problem by "remembering"
- Input, forget, update, & output gates
- Expensive to train

`nn.LSTM`

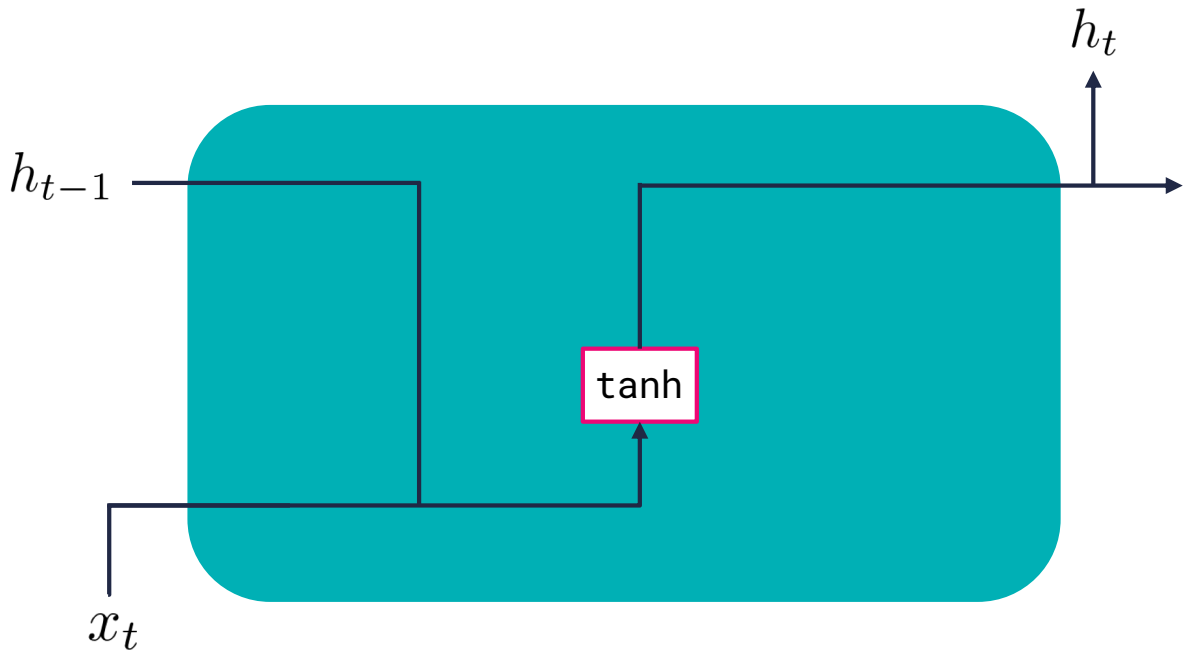
GRU

- **Gated recurrent unit**
- Newest addition to the family
- Typically faster to train than LSTMs without suffering much performance loss (performance ~same)
- Update & reset gates

`nn.GRU`

Simple RNN

Hidden State $h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t)$



Hadamard product (element-wise)

$$(A \circ B)_{ij} = (A)_{ij}(B)_{ij}$$

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \circ \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{bmatrix} = \begin{bmatrix} a_{11}\,b_{11} & a_{12}\,b_{12} & a_{13}\,b_{13} \\ a_{21}\,b_{21} & a_{22}\,b_{22} & a_{23}\,b_{23} \\ a_{31}\,b_{31} & a_{32}\,b_{32} & a_{33}\,b_{33} \end{bmatrix}$$

LSTM

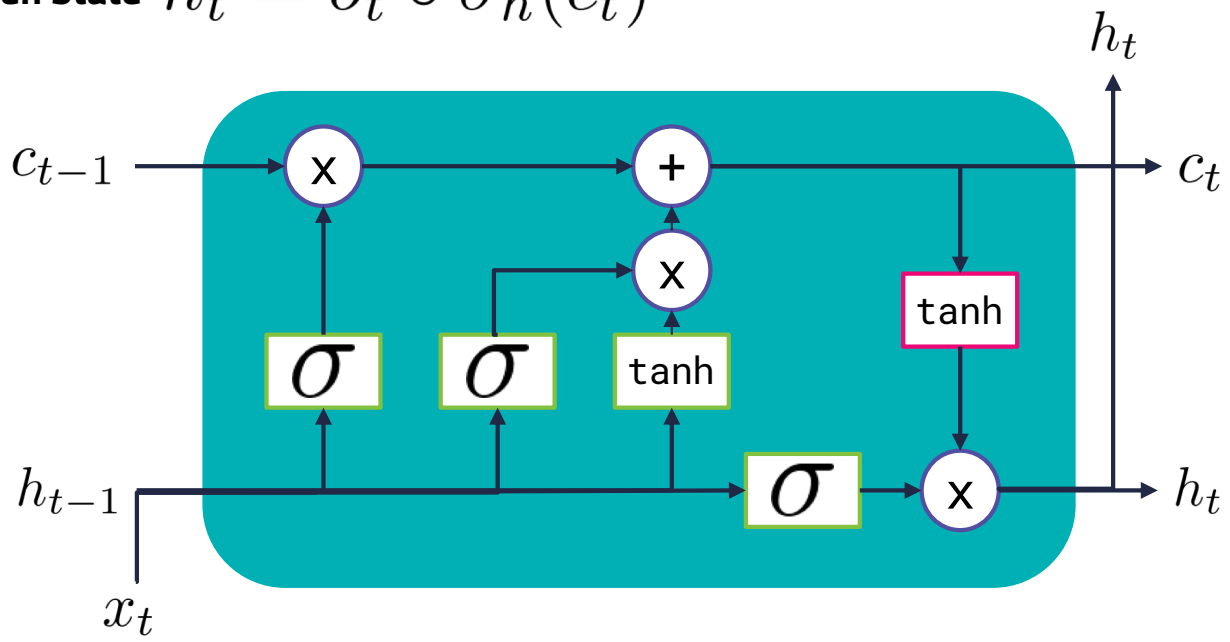
Forget Gate $f_t = \sigma_g(W_f x_t + U_f h_{t-1} + b_f)$

Update Gate $i_t = \sigma_g(W_i x_t + U_i h_{t-1} + b_i)$

Output Gate $o_t = \sigma_g(W_o x_t + U_o h_{t-1} + b_o)$

Cell State $c_t = f_t \circ c_{t-1} + i_t \circ \sigma_c(W_c x_t + U_c h_{t-1} + b_c)$

Hidden State $h_t = o_t \circ \sigma_h(c_t)$

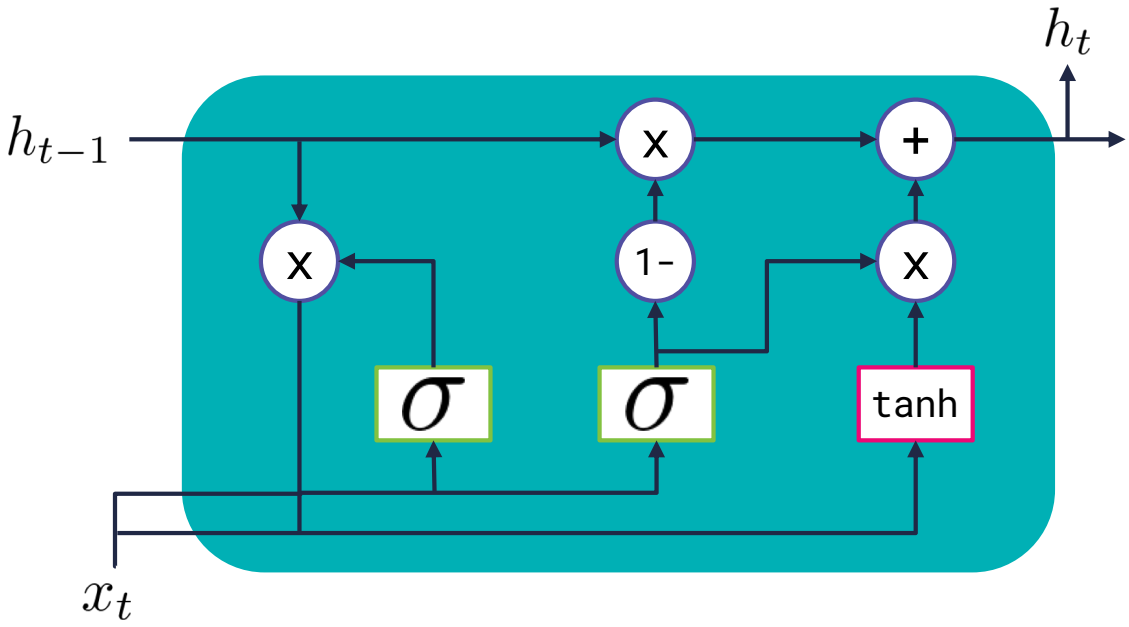


GRU

Update Gate $z_t = \sigma_g(W_z x_t + U_z h_{t-1} + b_z)$

Reset Gate $r_t = \sigma_g(W_r x_t + U_r h_{t-1} + b_r)$

Hidden State $h_t = (1 - z_t) \circ h_{t-1} + z_t \circ \sigma_h(W_h x_t + U_h(r_t \circ h_{t-1}) + b_h)$



PyTorch RNN Model Definition

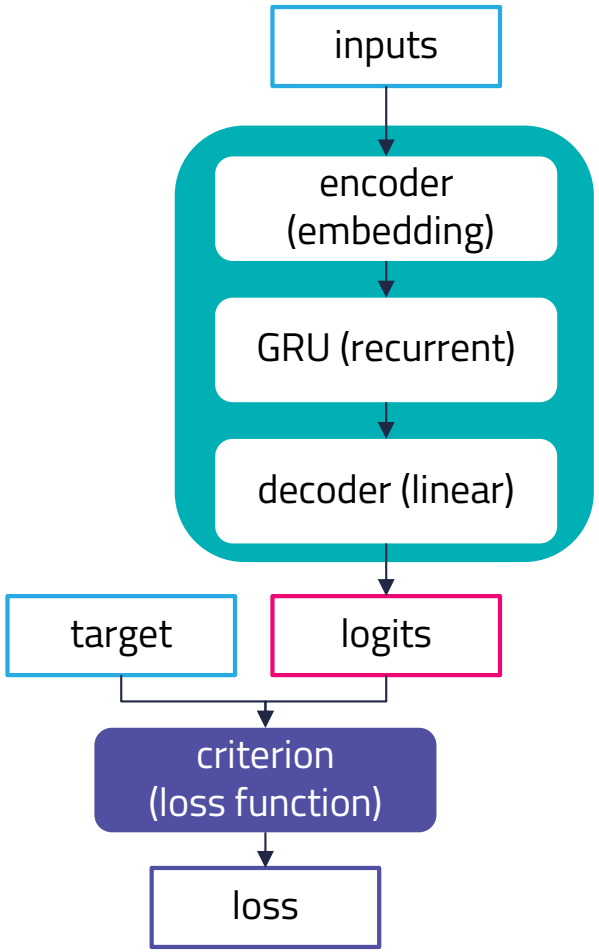
```
class MyModel(nn.Module):
    def __init__(self):
        super(MyModel, self).__init__()
        self.encoder = nn.Embedding(10, 50) # vocab size, emb. size
        self.rnn = nn.GRU(50, 128) # num inputer, num neurons
        self.decoder = nn.Linear(128, 1) # single output

    def init_hidden(self):
        return torch.randn(1, 1, 128)

    def forward(self, input_, hidden):
        encoded = self.encoder(input_)
        output, hidden = self.rnn(encoded.unsqueeze(1), hidden)
        output = self.decoder(output.squeeze(1))
        return output, hidden
```

Sentiment Text Classification

Example #1
inputs: “it was amazing 100%”
target: 1

Example #2
inputs: “that movie sucked!”
target: 0

Example #3
inputs: “i thought it was pretty decent”
target: 1



Notebook: RNN Classification

notebooks/PyTorch/3_rnn_text_classification.ipynb

Language Models for Text Generation

Char-RNN Text Generation (Language modeling)

Example #1

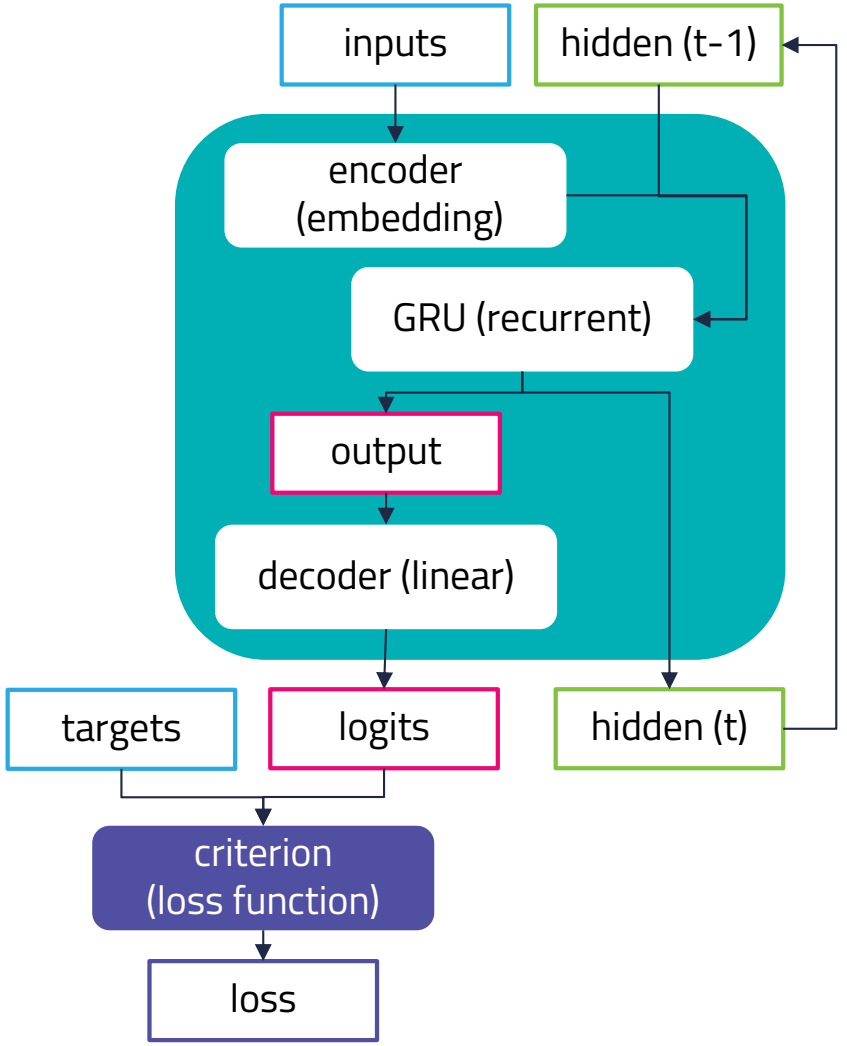
```
inputs:  “quick brown fox”
targets: “uick brown fox ”
```

Example #2

```
inputs:  “uick brown fox ”
targets: “ick brown fox j”
```

Example #3

```
inputs:  “ick brown fox j”
targets: “ck brown fox ju”
```

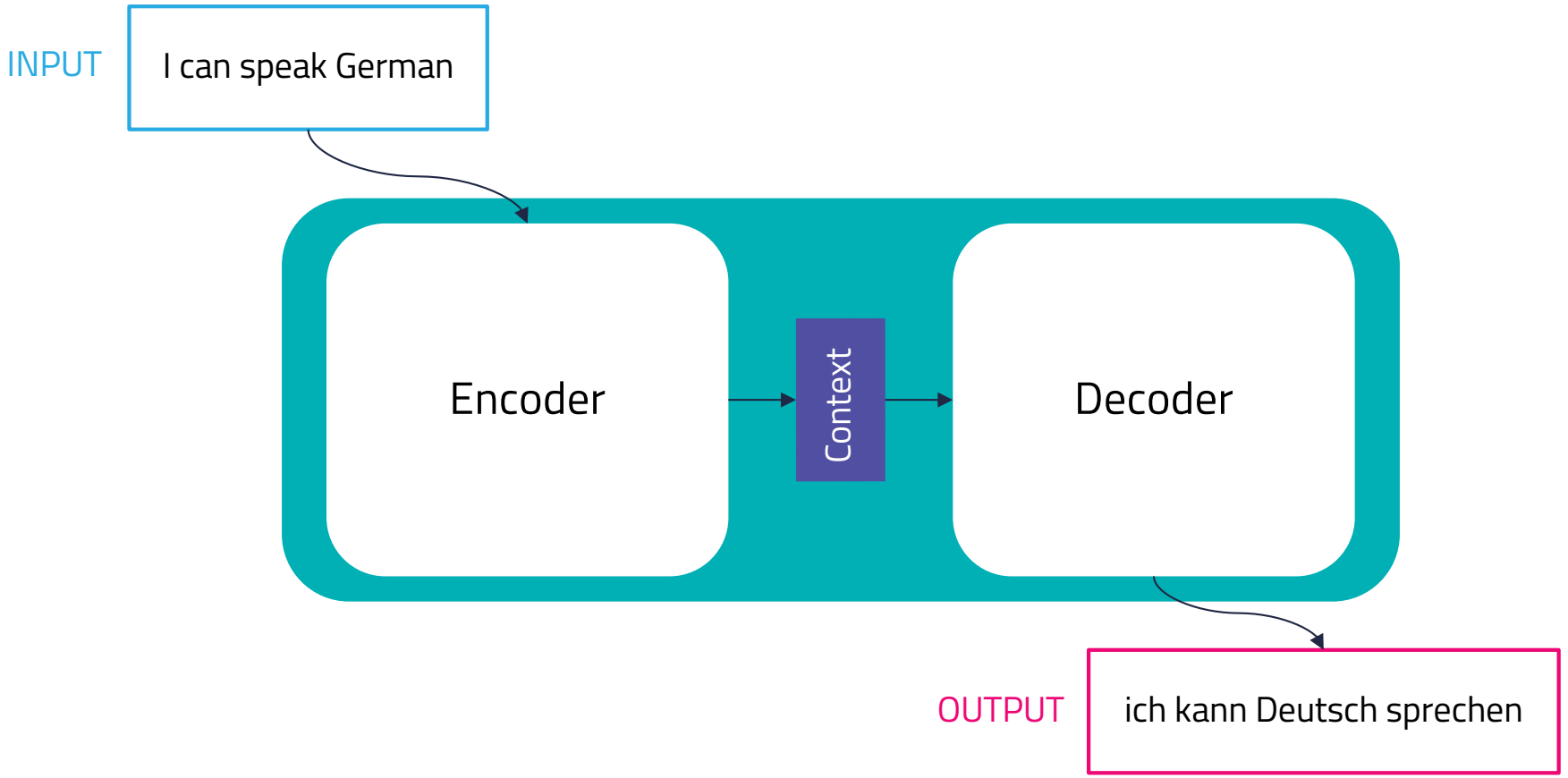


Notebook: Text Generation (skip?)

notebooks/PyTorch/4_character_text_generation.ipynb

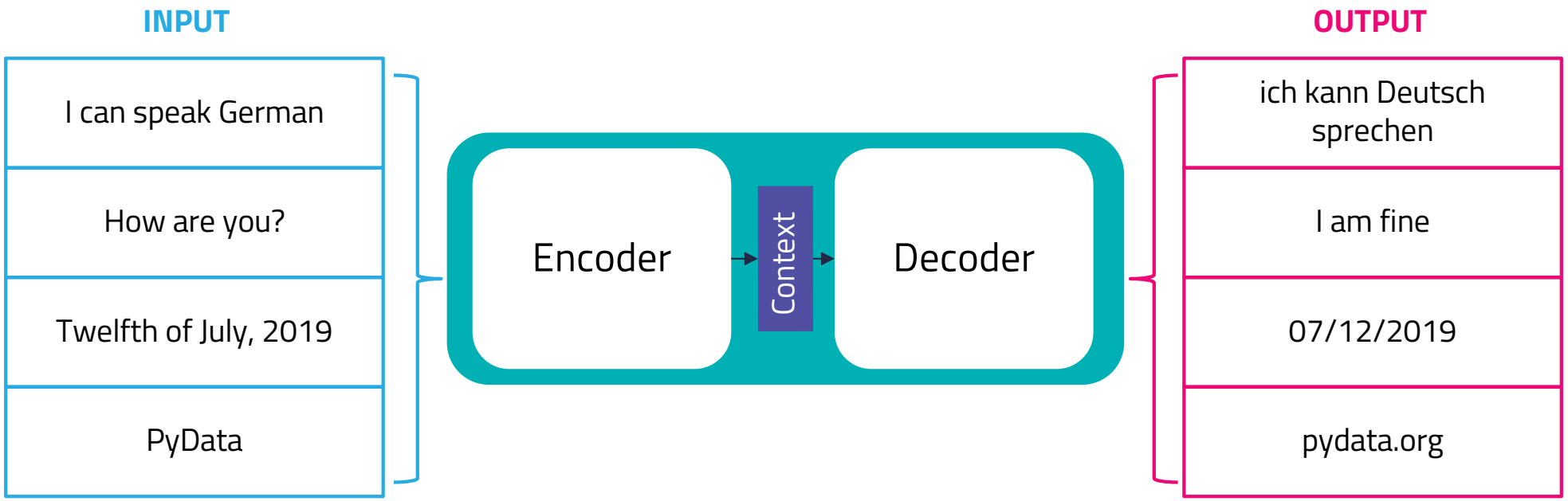
Sequence Models

Sequence to Sequence (Seq2Seq)

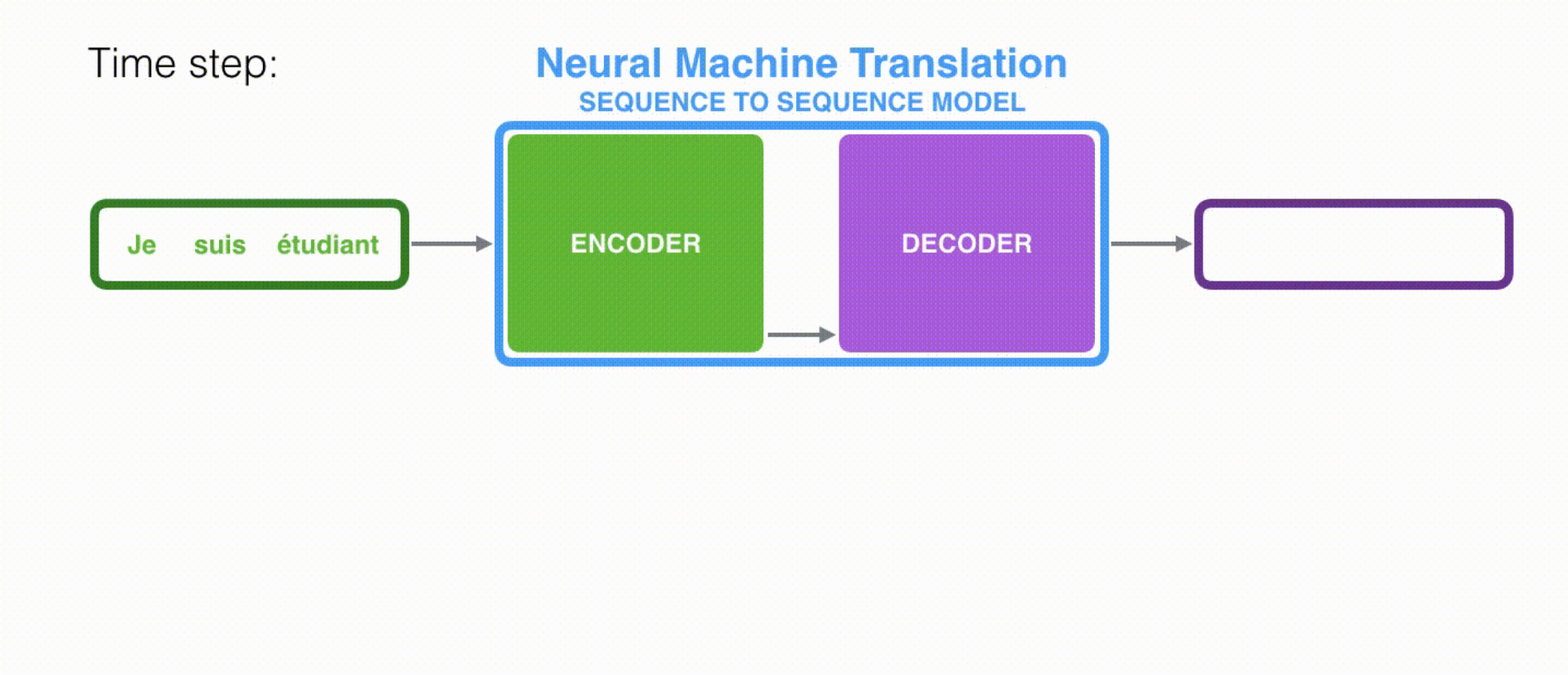


Applications of Seq2Seq - Encoder-Decoder

- **Machine Language Translation**
- **Question Answering / Chatbot**
- **Date Formatting**
- **Speech to Text**
- **Name to Domain**
- **Website Summary**

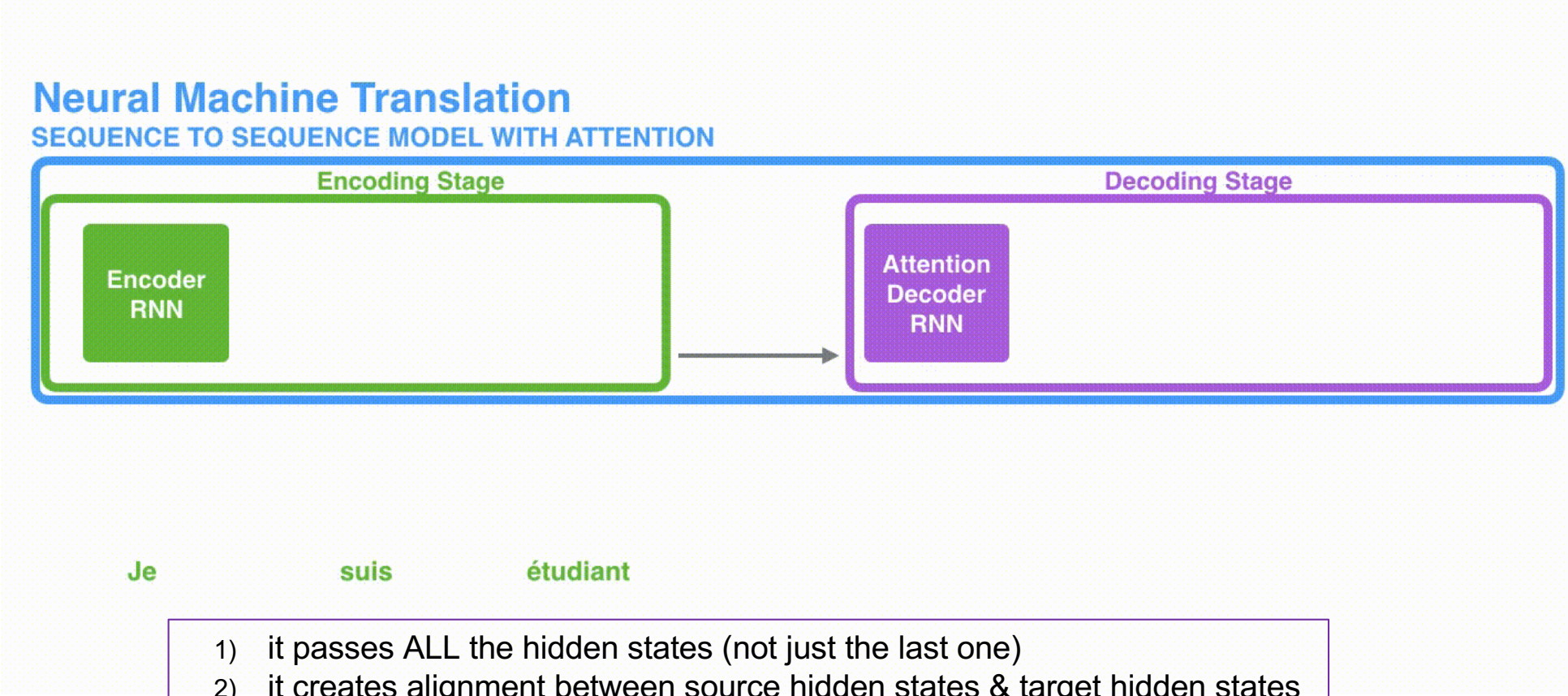


Seq2Seq animated



source

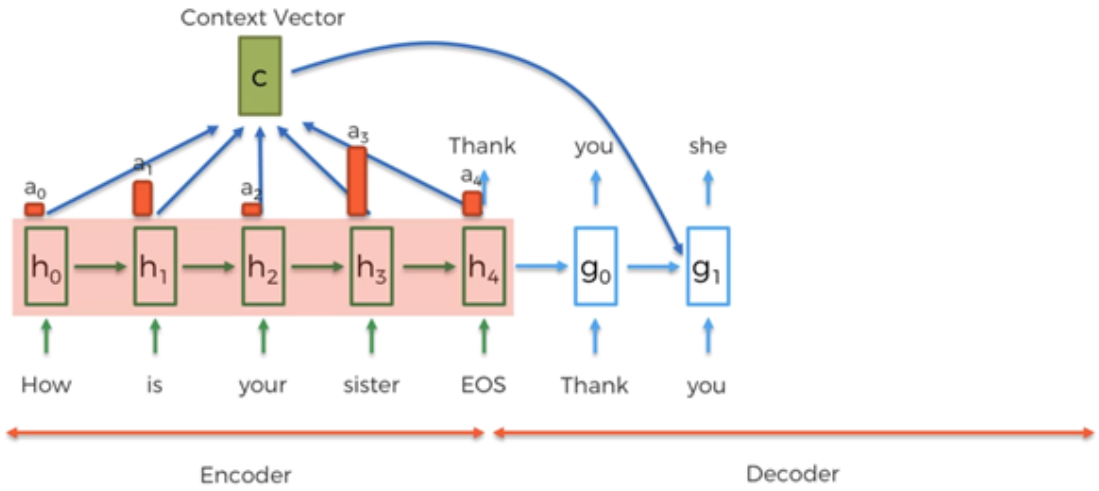
Seq2Seq with attention



1) it passes ALL the hidden states (not just the last one)
2) it creates alignment between source hidden states & target hidden states

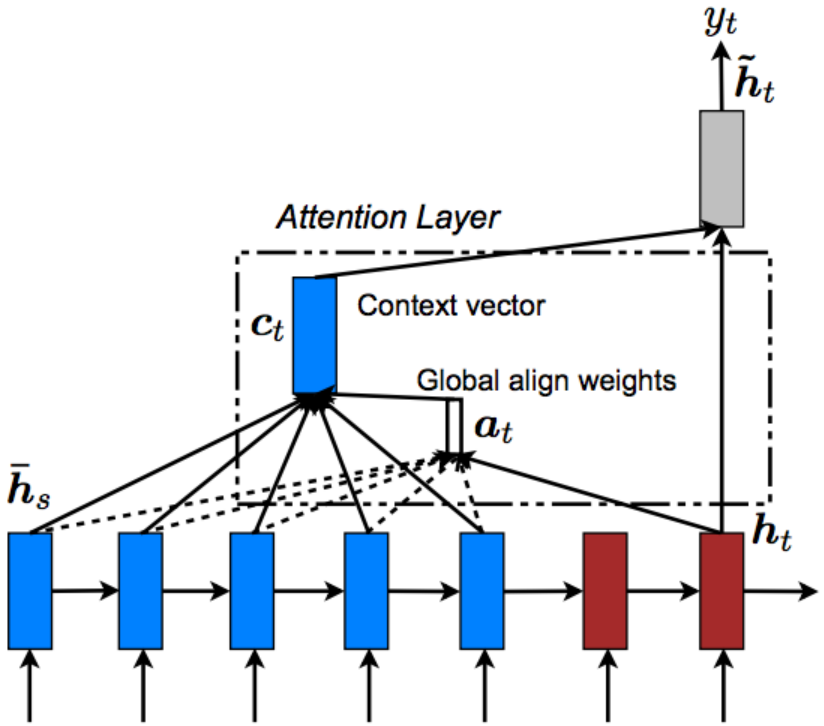
source

Attention

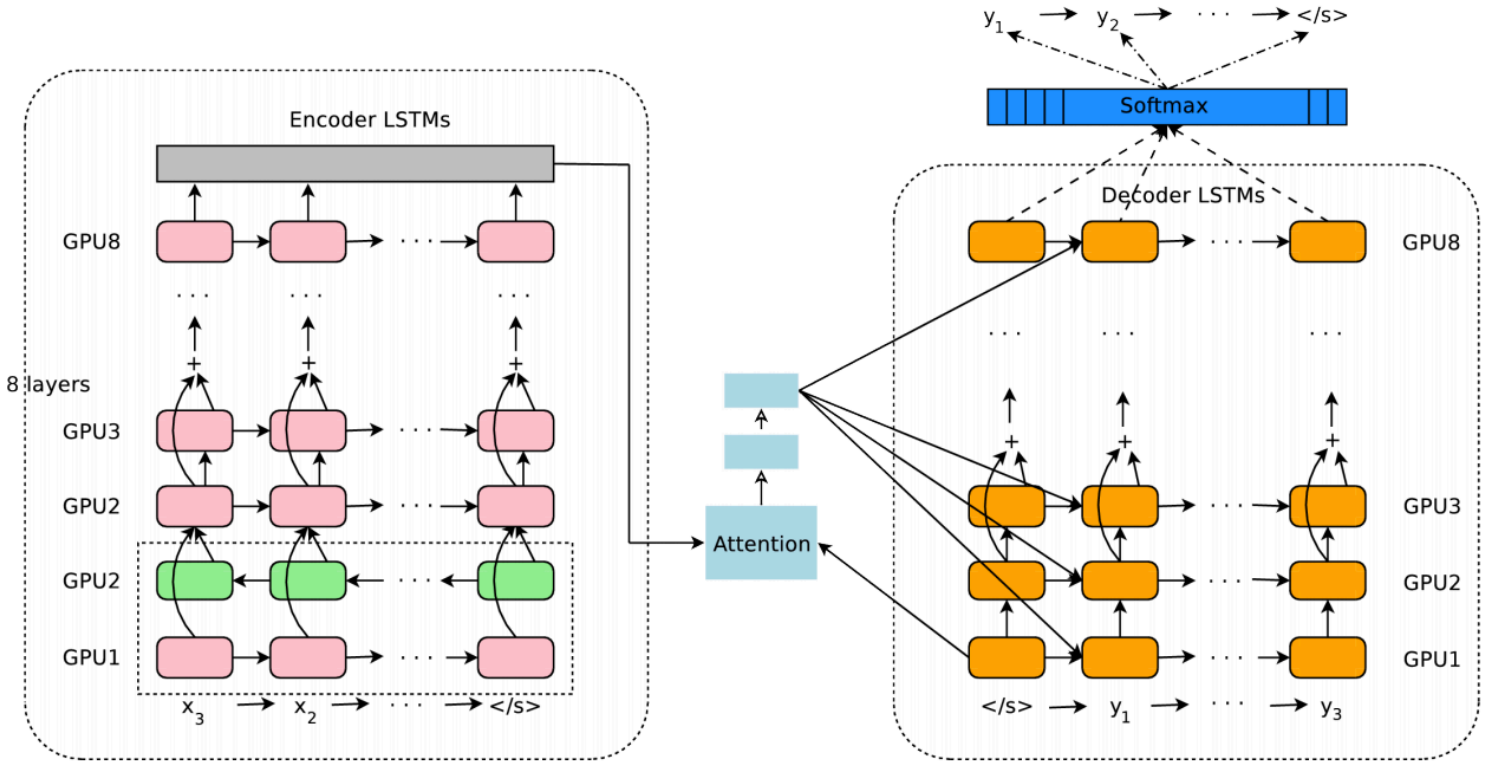

Context Vector
c
a_0
a_1
a_2
a_3
a_4
Thank
you
she
h_0
h_1
h_2
h_3
h_4
g_0
g_1
How
is
your
sister
EOS
Thank
you
Encoder
Decoder

Attention Mechanism

Implementation



Encoder Decoder



Encoder-Decoder Attention

Attention determines which parts of an input sequence are important for the decoder at a certain timestep

target hidden state

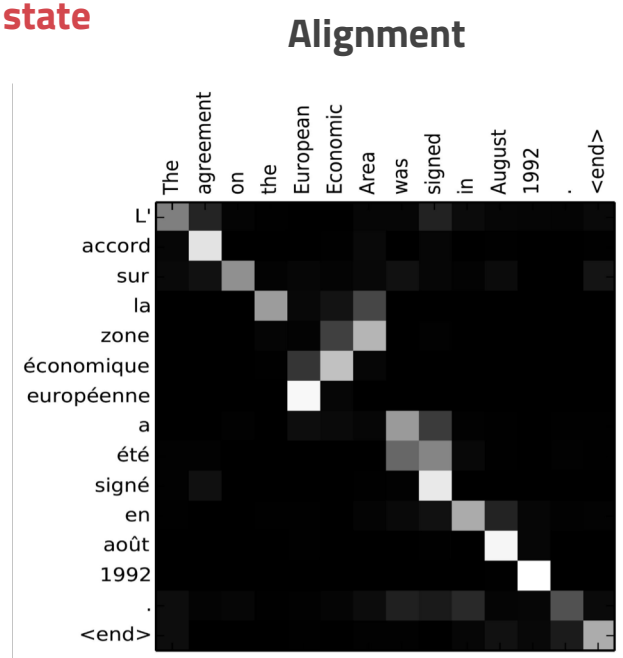
source hidden state

Attention weights (Alignment) $\alpha_{ts} = \frac{\exp(\text{score}(\mathbf{h}_t, \bar{\mathbf{h}}_s))}{\sum_{s'=1}^{S} \exp(\text{score}(\mathbf{h}_t, \bar{\mathbf{h}}_{s'}))}$

Context vector $\mathbf{c}_t = \sum_s \alpha_{ts} \bar{\mathbf{h}}_s$

Attention vector $\mathbf{a}_t = f(\mathbf{c}_t, \mathbf{h}_t) = \tanh(\mathbf{W}_c[\mathbf{c}_t; \mathbf{h}_t])$

Prediction $p(y_t|y_{<t}, x) = softmax(\boldsymbol{W_s a_t})$



Encoder-Decoder Attention in PyTorch

```
# Attention score (Bahdanaus 2015)
score = torch.tanh(
    self.W1(encoder_output) + self.W2(hidden_with_time_axis)
)

# Attention weights
attention_weights = torch.softmax(self.V(score), dim=1)

# Find the context vectors
context_vector = attention_weights * encoder_output
context_vector = torch.sum(context_vector, dim=1)
```

Find the family of attention score functions here

Transformer Model

parallelizable!

faster to train!

Attention Is All You Need

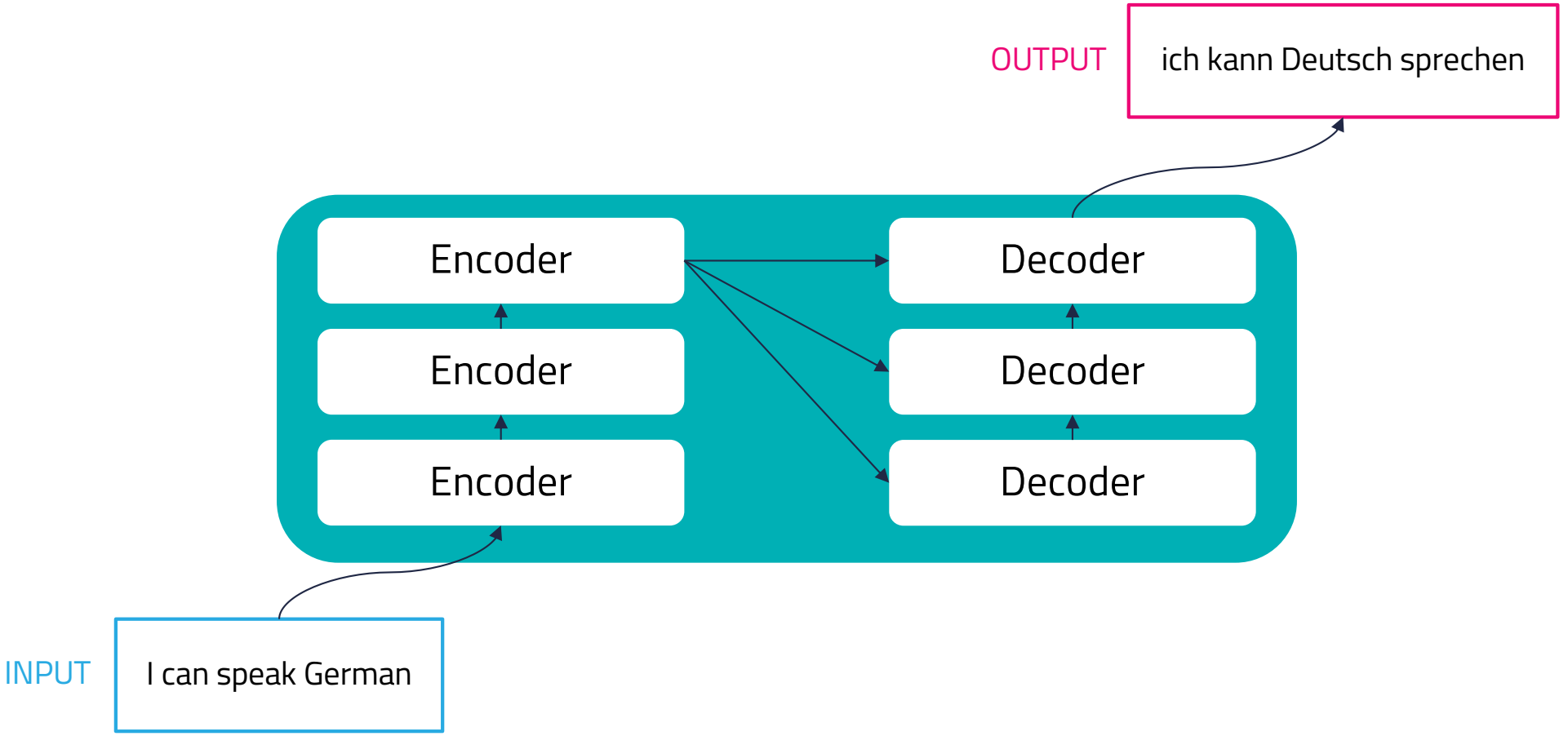
Ashish Vaswani* Google Brain avaswani@google.com
Noam Shazeer* Google Brain noam@google.com
Niki Parmar* Google Research nikip@google.com
Jakob Uszkoreit* Google Research usz@google.com
Llion Jones* Google Research llion@google.com
Aidan N. Gomez* † University of Toronto aidan@cs.toronto.edu
Łukasz Kaiser* Google Brain lukaszkaiser@google.com
Illia Polosukhin* ‡ illia.polosukhin@gmail.com

Abstract

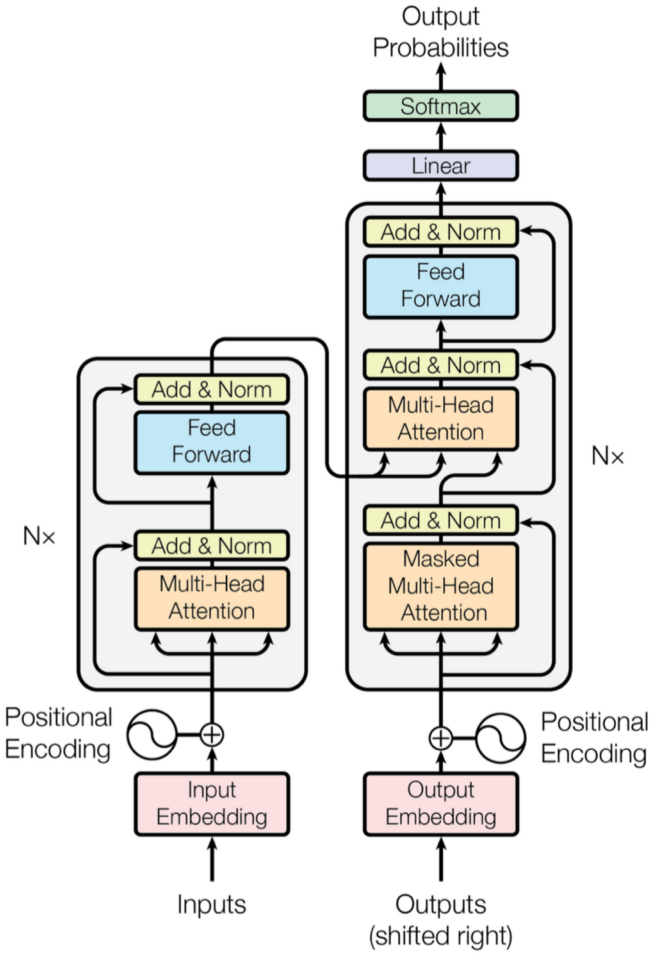
The dominant sequence transduction models are based on complex recurrent or convolutional neural networks that include an encoder and a decoder. The best performing models also connect the encoder and decoder through an attention mechanism. We propose a new simple network architecture, the Transformer, based solely on attention mechanisms, dispensing with recurrence and convolutions entirely. Experiments on two machine translation tasks show these models to be superior in quality while being more parallelizable and requiring significantly less time to train. Our model achieves 28.4 BLEU on the WMT 2014 English-to-German translation task, improving over the existing best results, including ensembles, by over 2 BLEU. On the WMT 2014 English-to-French translation task, our model establishes a new single-model state-of-the-art BLEU score of 41.8 after training for 3.5 days on eight GPUs, a small fraction of the training costs of the best models from the literature. We show that the Transformer generalizes well to other tasks by applying it successfully to English constituency parsing both with large and limited training data.

1 Introduction

Transformer: A stack of encoders and decoders



Transformer Model Architecture



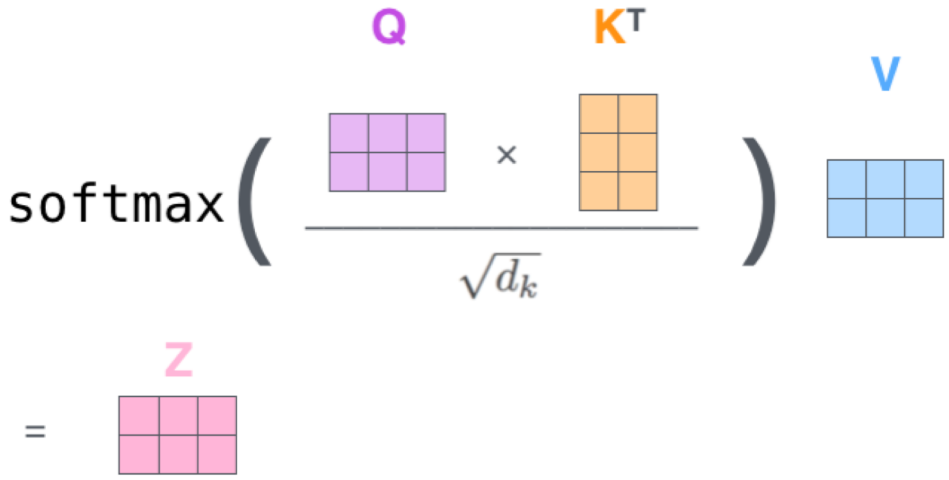
Self-Attention

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

seq length x embedding size

embedding size x vector size

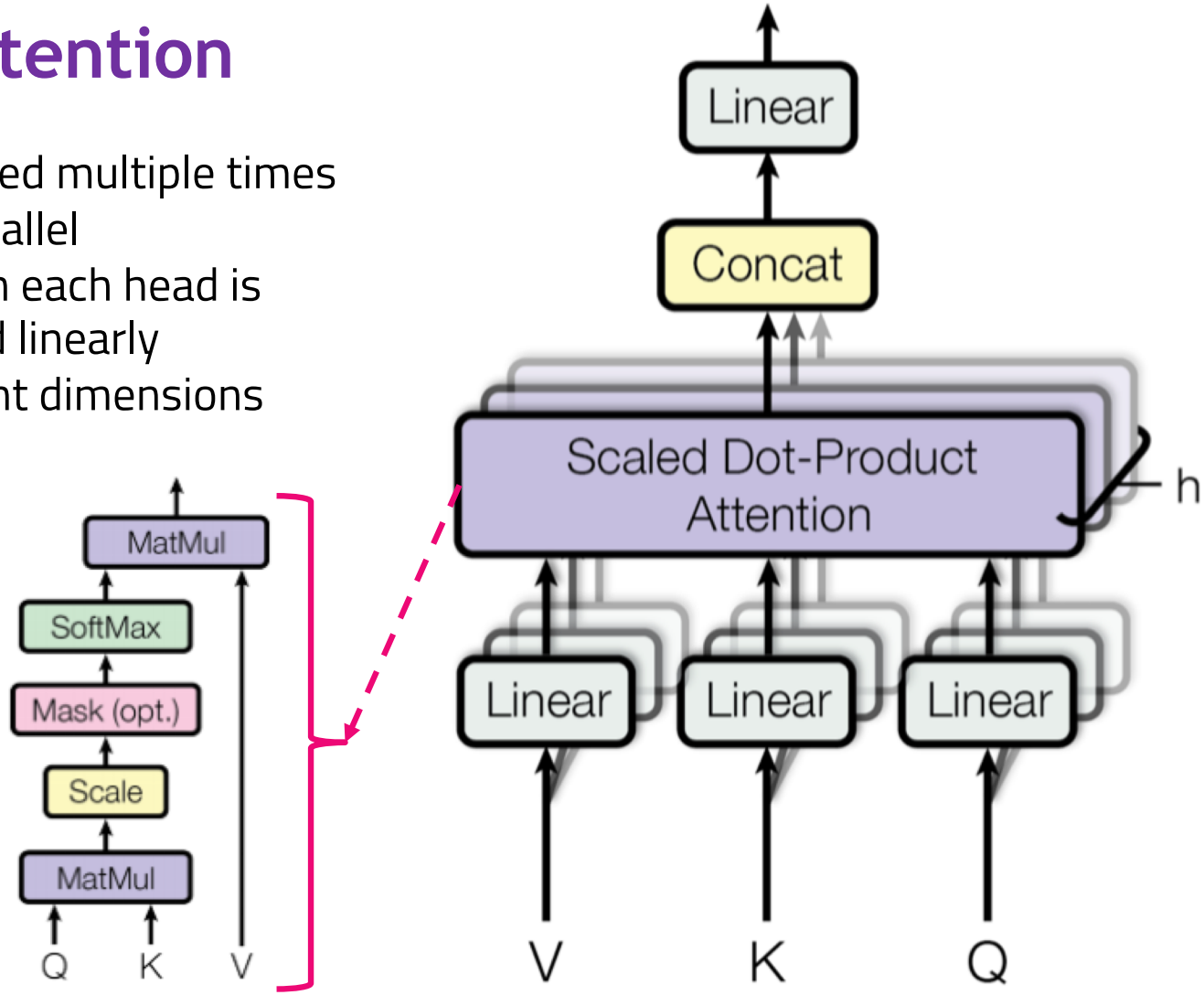
$$Q = XW^Q$$
$$K = XW^K$$
$$V = XW^V$$



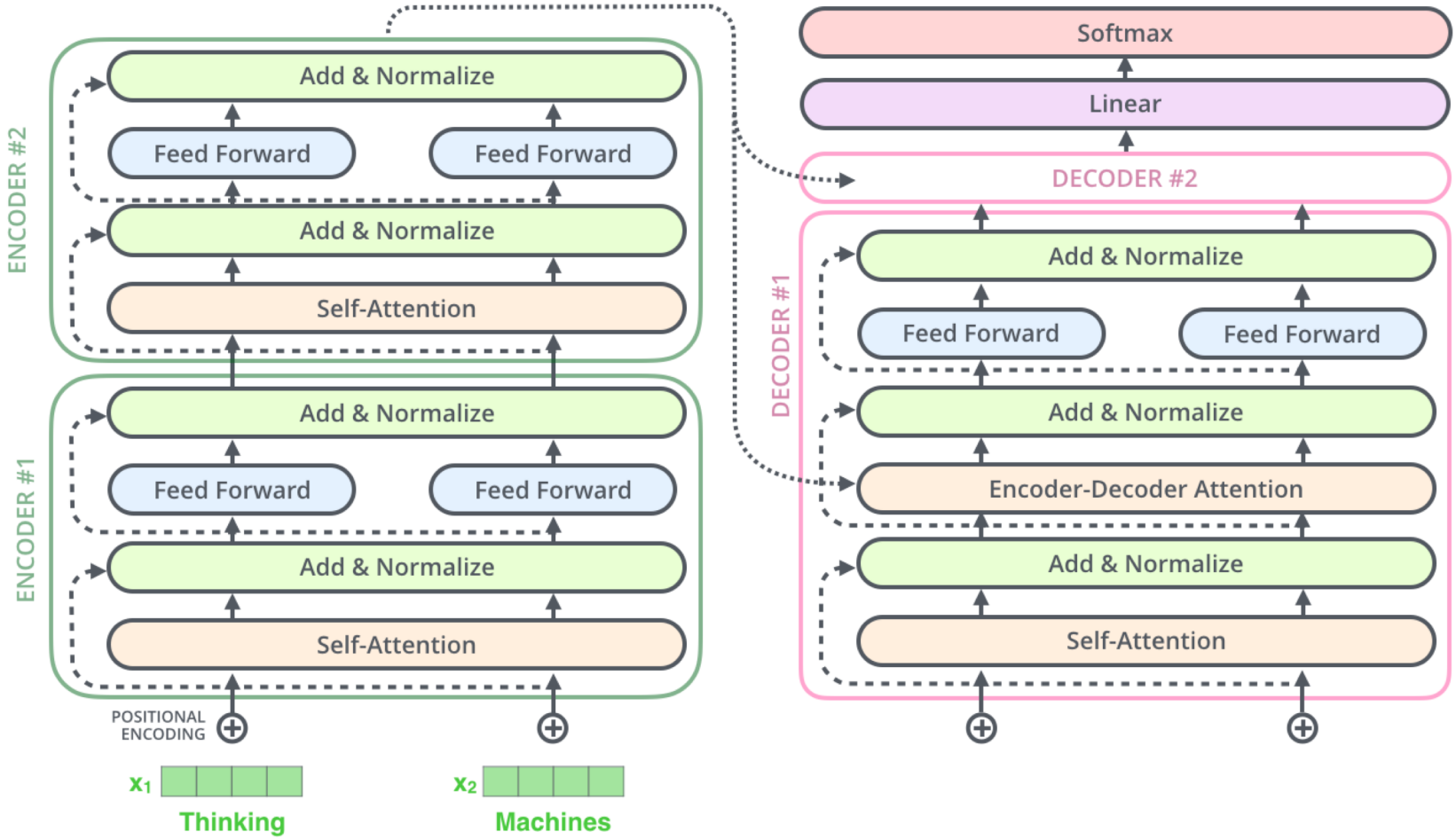
Multi-Head Self-Attention

- Self-attention is calculated multiple times independently and in parallel
- The context vectors from each head is simply concatenated and linearly transformed into the right dimensions

"multi-head attention allows the model to jointly attend to information from different representation subspaces at different positions. With a single attention head, averaging inhibits this."



Transformer Model Architecture

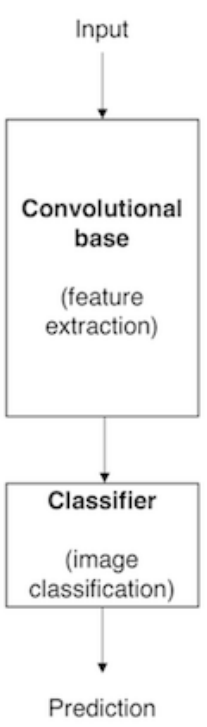


Notebook: Sequence to Sequence

notebooks/PyTorch/5_seq2seq_attention_translation.ipynb

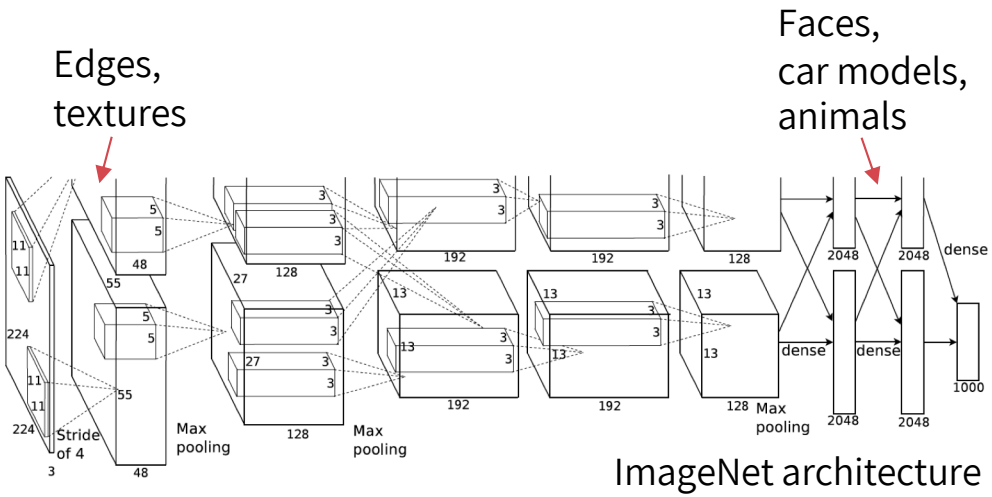
Transfer Learning + Fine Tuning – NLP?

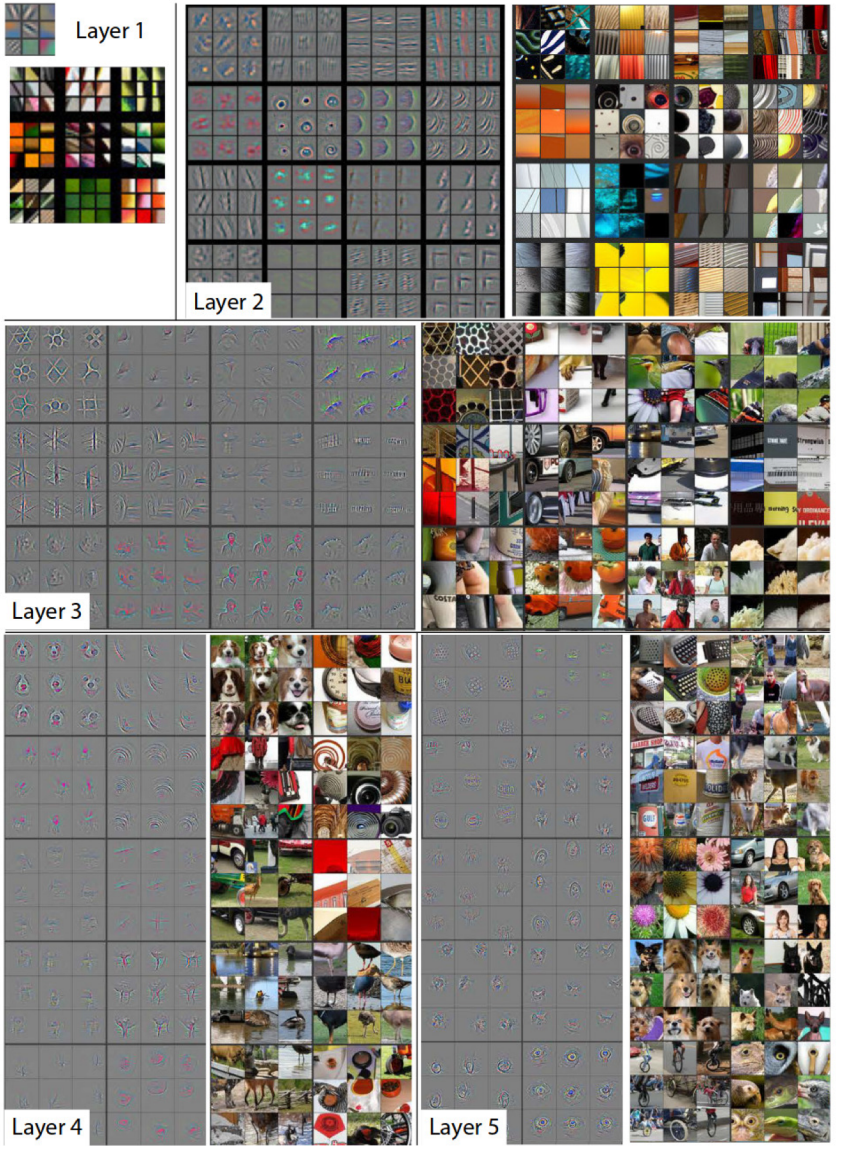
- Norm for computer vision tasks, work of Razavian et al (2014).
- Reduces the training time and data needed to achieve custom ML task.
- Replace final layers of pre-trained CNN model (ImageNet), with one or more custom layers.
- Continue training with custom data set:
  - Freeze original weights, fine-tune, etc.



Transfer Learning for Images

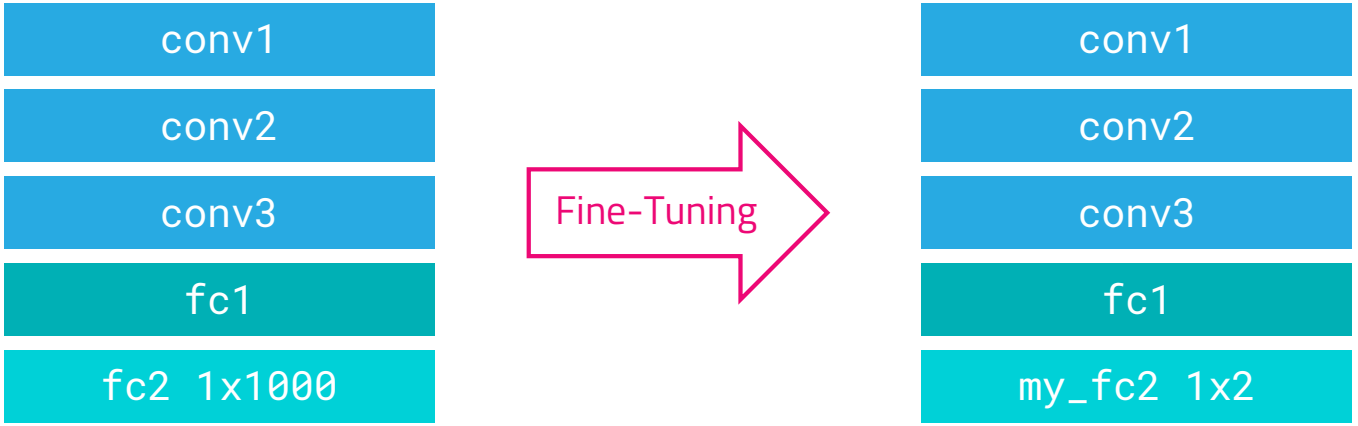
ImageNet: The first widely-used successful application of transfer learning





Fine-tuning

Fine-tuning is typically taking a large pretrained model, replacing the last layer with your own, and training it with new examples on a specific task



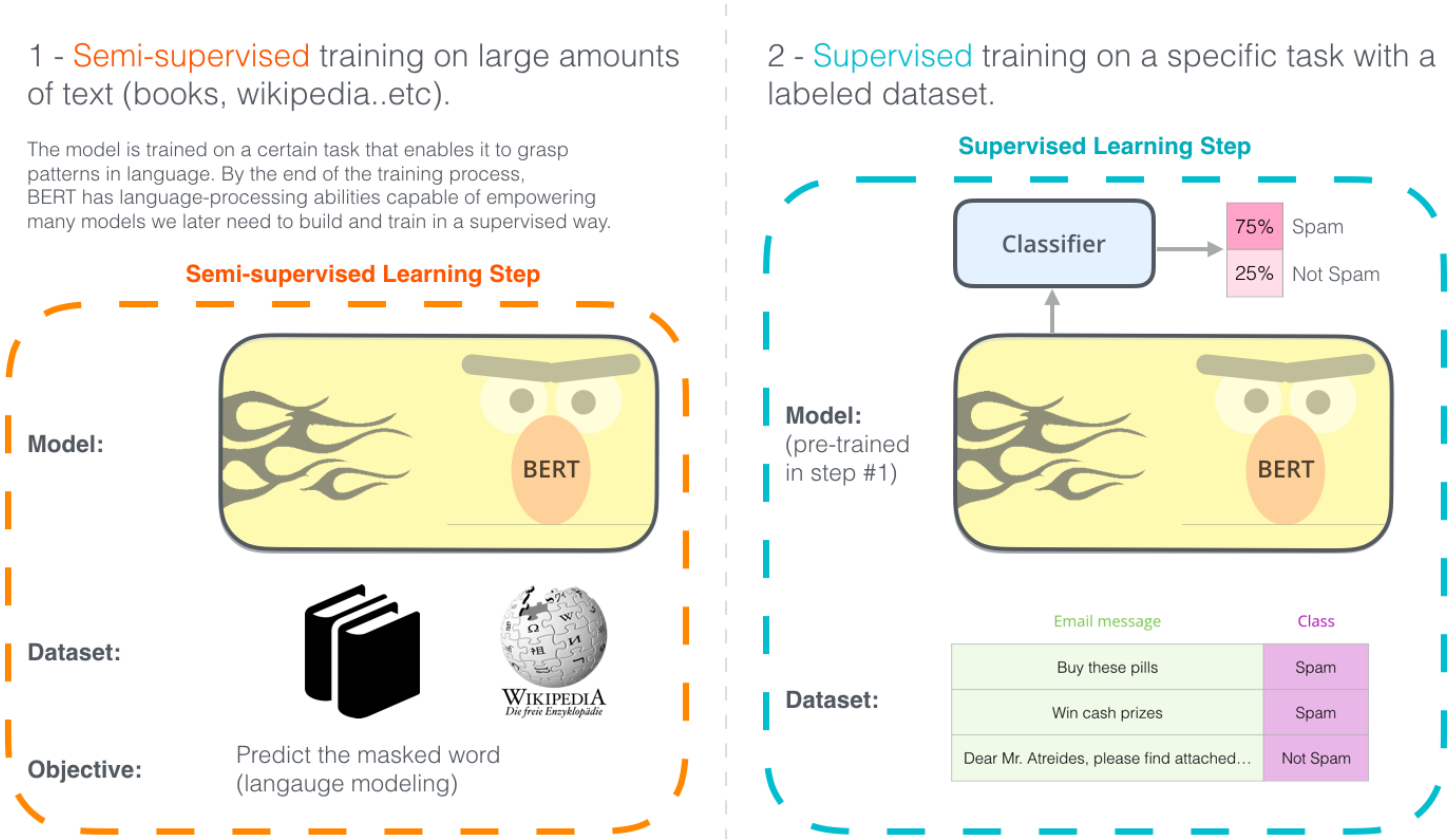

```
{
    …,
    862: 'torch',
    863: 'totem pole',
    864: 'tow truck, tow car, wrecker',
    865: 'toyshop',
    866: 'tractor',
    …,
}
```

```
{
    0: 'cat',
    1: 'dog',
}
```

Breakthrough NLP Developments in 2017 & 2018

- Attention, Self-Attention
- Transformer – Universal Encoder

Breakthrough NLP Developments in 2018-2019



https://jalammar.github.io/illustrated-bert/

Self-supervised pre-training

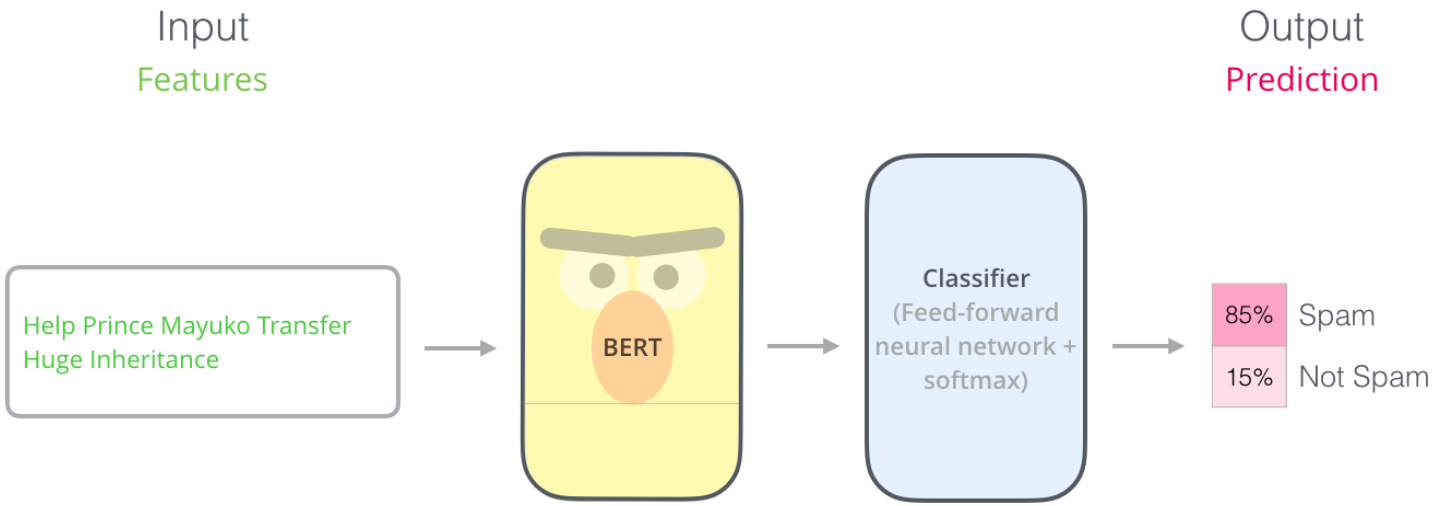
BERT builds upon the ideas of several models:

- Transformer
- ULMFit
- ElMo
- OpenAI Transformer (GPT, GPT-2)

A couple of models claim to now outperform BERT

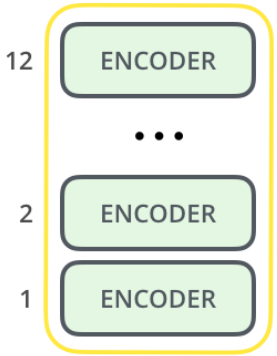
- XLNet

Example: classify a piece of text



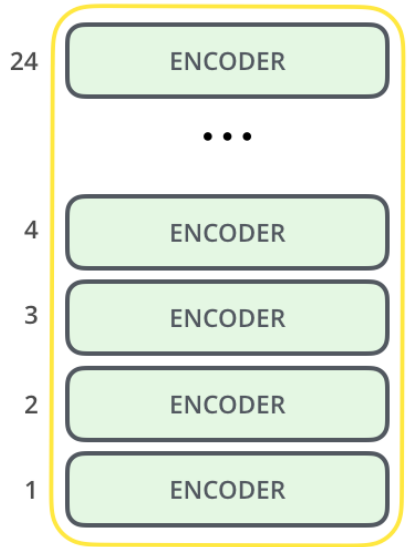
Email message	Class
Buy these pills	Spam
Win cash prizes	Spam
Dear Mr. Atreides, please find attached…	Not Spam

BERT is basically a trained Transformer Encoder stack.



$BERT_{BASE}$

12



$BERT_{LARGE}$

24

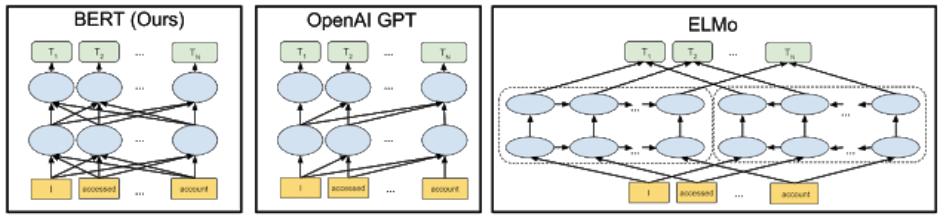
BERT - Bidirectional Encoder Representations from Transformers

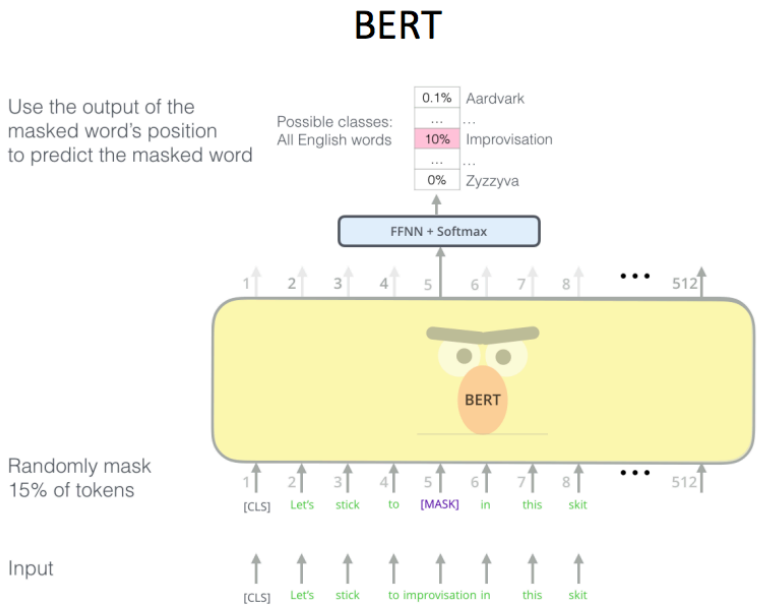
BERT builds upon recent work in pre-training contextual representations — including Semi-supervised Sequence Learning.

Unlike ELMo and ULMFit, BERT is the first deeply bidirectional, unsupervised language representation, pre-trained using only a plain text corpus (Wikipedia).

Pre-trained representations can either be context-free or contextual, and contextual representations.
Bert uses innovative training procedure for bidirectional context.

.





Devlin, Jacob, et al. "Bert: Pre-training of deep bidirectional transformers for language understanding." (2018).

BERT Pre-Training

- It's not possible to train bidirectional models by simply conditioning each word on its previous *and* next words, since this would allow the word that's being predicted to indirectly "see itself" in a multi-layer model.

  To solve this problem, BERT masks out some of the words in the input and then conditions each word bidirectionally to predict the masked words. E.g.,:

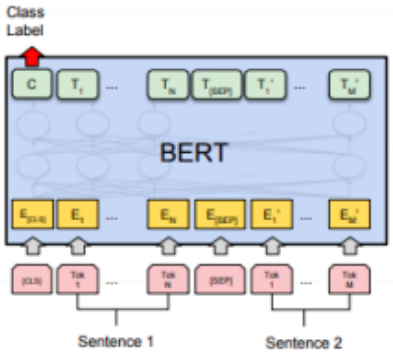
  Input: The man went to the $[MASK]_1$. He bought a $[MASK]_2$ of milk .
  Labels: $[MASK]_1$ = store; $[MASK]_2$ = gallon

- BERT also learns to model relationships between sentences by pre-training on a very simple task that can be generated from any text corpus:
  - Given two sentences A and B, is B the actual next sentence that comes after A in the corpus, or just a random sentence? For example:

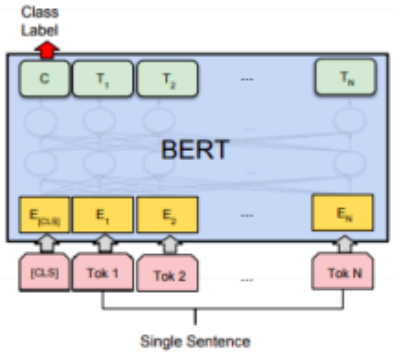
  Sentence A = The man went to the store.
  Sentence B = He bought a gallon of milk.
  Label = IsNextSentence

  Sentence A = The man went to the store.
  Sentence B = Penguins are flightless.
  Label = NotNextSentence

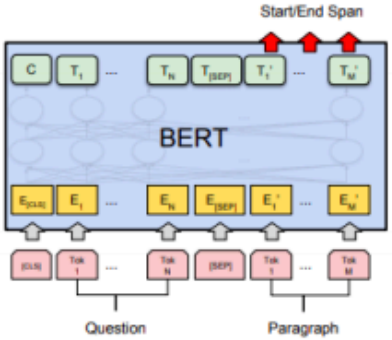
BERT Applications



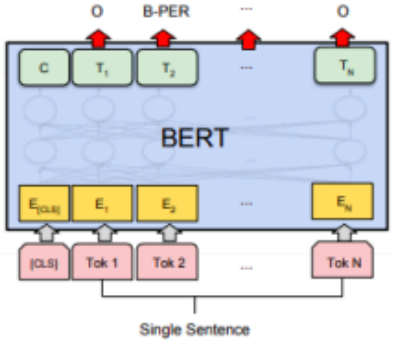
(a) Sentence Pair Classification Tasks: MNLI, QQP, QNLI, STS-B, MRPC, RTE, SWAG



(b) Single Sentence Classification Tasks: SST-2, CoLA



(c) Question Answering Tasks: SQuAD v1.1



(d) Single Sentence Tagging Tasks: CoNLL-2003 NER

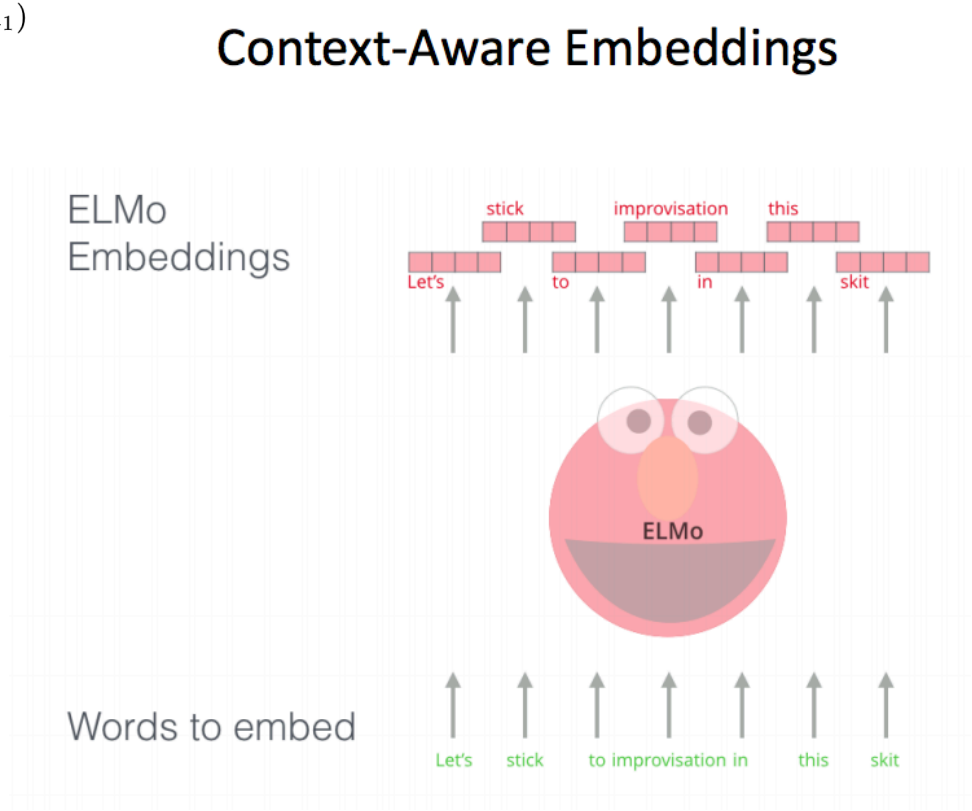
Now you can use BERT:

- Create contextualized word embeddings (like ELMo)
- Sentence classification
- Sentence pair classification
- Sentence pair similarity
- Multiple choice
- Sentence tagging
- Question answering

ELMo

Language model: $$P(w_i \mid w_{i-(n-1)}, \ldots, w_{i-1}) = \frac{\text{count}(w_{i-(n-1)}, \ldots, w_{i-1}, w_i)}{\text{count}(w_{i-(n-1)}, \ldots, w_{i-1})}$$

- ELMo - deep contextualized word representation
- Word vectors are learned functions of the internal states of a deep bidirectional language model (biLM), which is pre-trained on a large text corpus.
- Can be added to existing models.
- Significantly improves the state of the art across a broad range of challenging NLP problems: question answering, textual entailment and sentiment analysis

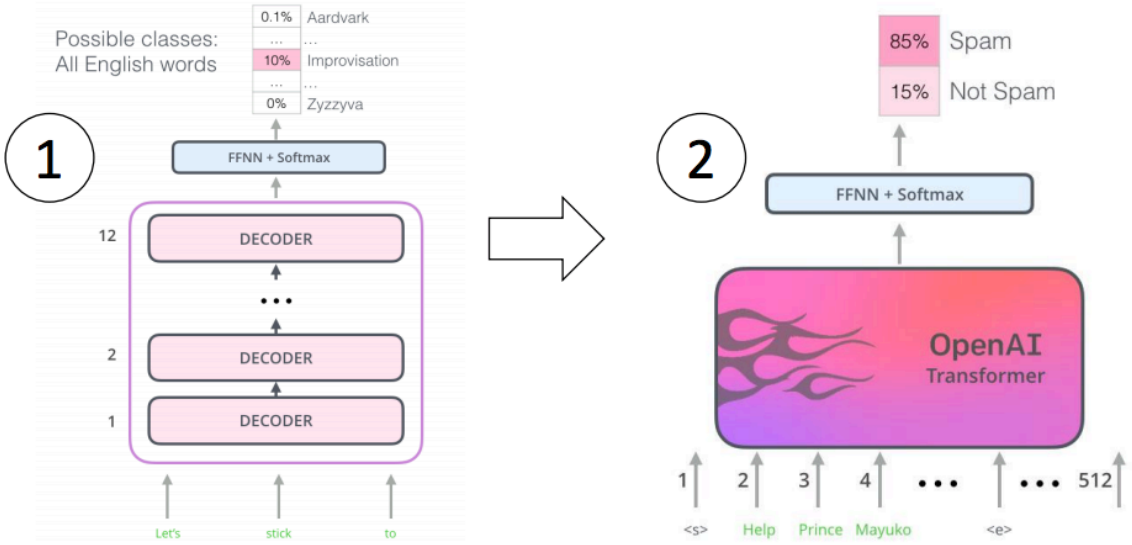


Deep contextualized word representations
Matthew E. Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, Luke Zettlemoyer.
NAACL 2018.

Key insight: pairing supervised learning methods with unsupervised pre-training works very well

GPT, GPT-2

OpenAI Transformer



Works in two stages:
1) Train a transformer model on a very large amount of data in an unsupervised manner — using language modeling as a training signal.
2) Fine-tune this model on much smaller supervised datasets to help it solve specific tasks.

Improving Language Understanding by Generative Pre-Training Alec Radford OpenAI alec@openai.com Karthik Narasimhan OpenAI karthikn@openai.com Tim Salimans OpenAI tim@openai.com Ilya Sutskever OpenAI ilyasu@openai.com

Key insight: unsupervised learning techniques can yield surprisingly discriminative features when trained on enough data.

Notebook: GPT-2 Language Generation

notebooks/PyTorch/6_gpt2_finetuned_text_generation.ipynb

Extra credit:
notebooks/PyTorch/2_embeddings.ipynb

Hands-on Exercise

Generating Toxic Comments

```
# 1. open up GPT-2 Notebook in Google Colab
# 2. download the file
!wget https://raw.githubusercontent.com/ipcplusplus/toxic-comments-classification/master/train.csv


# 3. load into dataframe
df = pd.read_csv('train.csv')
```

Additional References

- Neural Responding Machine for Short-Text Conversation (2015-03)
- A Neural Conversational Model (2015-06)
- A Neural Network Approach to Context-Sensitive Generation of Conversational Responses (2015-06)
- The Ubuntu Dialogue Corpus: A Large Dataset for Research in Unstructured Multi-Turn Dialogue Systems (2015-06)
- Building End-To-End Dialogue Systems Using Generative Hierarchical Neural Network Models (2015-07)
- A Diversity-Promoting Objective Function for Neural Conversation Models (2015-10)
- Attention with Intention for a Neural Network Conversation Model (2015-10)
- Improved Deep Learning Baselines for Ubuntu Corpus Dialogs (2015-10)
- A Survey of Available Corpora for Building Data-Driven Dialogue Systems (2015-12)
- Incorporating Copying Mechanism in Sequence-to-Sequence Learning (2016-03)
- A Persona-Based Neural Conversation Model (2016-03)
- How NOT To Evaluate Your Dialogue System: An Empirical Study of Unsupervised Evaluation Metrics for Dialogue Response Generation (2016-03)

Learning resources

Stanford class on deep learning for NLP.http://cs224d.stanford.edu/syllabus.html

Hinton's Coursera Course. Get it right from the horse's mouth. He explains things well.
https://www.coursera.org/course/neuralnets

Online textbook in preparation for deep learning from Yoshua Bengio and friends. Clear and understandable.
http://www.iro.umontreal.ca/~bengioy/dlbook/

TensorFlow tutorials.
https://www.tensorflow.org/versions/r0.8/tutorials/index.html

Further resources

- Great visualizations from Jay Alammar's blog (the gifs we use are from here)
- New fast.ai course: A Code-First Introduction to Natural Language Processing
- Upcoming book by SpaCy creator: Deep Learning with Text: Natural Language Processing (Almost) from Scratch with Python and spaCy by Patrick Harrison and Matthew Honnibal
- Great blog post on RNNs from colah's blog
- Large curated list of resources: brianspiering/awesome-dl4nlp