Deep Learning
Sequence to Sequence models: Attention Models
Sequence to sequence models

- Sequence goes in, sequence comes out
- No notion of “time synchrony” between input and output
  - May even not even maintain order of symbols
    - E.g. “I ate an apple” $\rightarrow$ “Ich habe einen apfel gegessen”
  - Or even seem related to the input
    - E.g. “My screen is blank” $\rightarrow$ “Please check if your computer is plugged in.”
Modelling the problem

- *Delayed* sequence to sequence
Modelling the problem

• Delayed sequence to sequence

First process the input and generate a hidden representation for it
Modelling the problem

- Delayed sequence to sequence

First process the input and generate a hidden representation for it

Then use it to generate an output

• Delayed sequence to sequence
Modelling the problem

- **Problem**: Each word that is output depends only on current hidden state, and not on previous outputs.
Modelling the problem

- Delayed sequence to sequence
  - Delayed *self-referencing* sequence-to-sequence
The “simple” translation model

- The input sequence feeds into a recurrent structure
- The input sequence is terminated by an explicit <eos> symbol
  - The hidden activation at the <eos> “stores” all information about the sentence

I ate an apple <eos>
The “simple” translation model

- The input sequence feeds into a recurrent structure
- The input sequence is terminated by an explicit <eos> symbol
  - The hidden activation at the <eos> “stores” all information about the sentence
- Subsequently a second RNN uses the hidden activation as initial state, and <sos> as initial symbol, to produce a sequence of outputs
  - The output at each time becomes the input at the next time
  - Output production continues until an <eos> is produced
The “simple” translation model

- The input sequence feeds into a recurrent structure
- The input sequence is terminated by an explicit <eos> symbol
  - The hidden activation at the <eos> “stores” all information about the sentence
- Subsequently a second RNN uses the hidden activation as initial state to produce a sequence of outputs
  - The output at each time becomes the input at the next time
  - Output production continues until an <eos> is produced

Ich

I ate an apple <eos> <sos>
The “simple” translation model

- The input sequence feeds into a recurrent structure
- The input sequence is terminated by an explicit <eos> symbol
  - The hidden activation at the <eos> “stores” all information about the sentence
- Subsequently a second RNN uses the hidden activation as initial state to produce a sequence of outputs
  - The output at each time becomes the input at the next time
  - Output production continues until an <eos> is produced
The “simple” translation model

• The input sequence feeds into a recurrent structure
• The input sequence is terminated by an explicit <eos> symbol
  – The hidden activation at the <eos> “stores” all information about the sentence
• Subsequently a second RNN uses the hidden activation as initial state to produce a sequence of outputs
  – The output at each time becomes the input at the next time
  – Output production continues until an <eos> is produced
The “simple” translation model

- The input sequence feeds into a recurrent structure
- The input sequence is terminated by an explicit <eos> symbol
  - The hidden activation at the <eos> “stores” all information about the sentence
- Subsequently a second RNN uses the hidden activation as initial state to produce a sequence of outputs
  - The output at each time becomes the input at the next time
  - Output production continues until an <eos> is produced
The “simple” translation model

- The recurrent structure that extracts the hidden representation from the input sequence is the *encoder*

- The recurrent structure that utilizes this representation to produce the output sequence is the *decoder*
A problem with this framework

- All the information about the input sequence is embedded into a single vector
  - The “hidden” node layer at the end of the input sequence
  - This one node is “overloaded” with information
    - Particularly if the input is long
A problem with this framework

- In reality: *All* hidden values carry information
  - Some of which may be diluted by the time we get to the final state of the encoder
A problem with this framework

• In reality: *All* hidden values carry information
  – Some of which may be diluted by the time we get to the final state of the encoder

• *Every* output is related to the input directly
  – Not sufficient to have the encoder hidden state to *only* the initial state of the decoder
  – Misses the direct relation of the outputs to the inputs
• Simple solution: Compute the average of all encoder hidden states
• Input this average to every stage of the decoder
• The initial decoder hidden state is now separate from the encoder
  – And may be a learnable parameter
Using all input hidden states

• **Problem:** The average applies the same weight to every input
  • It supplies the same average to every output word
  • In practice, different outputs may be related to different inputs
    – E.g. “Ich” is most related to “I”, and “habe” and “gegessen” are both most related to “ate”
Using all input hidden states

- **Solution:** Use a *different* weighted average for each output word
  - The weighted average provided for the kth output word is:
    \[ c_t = \frac{1}{N} \sum_{i}^{N} w_i(t) h_i \]
Using all input hidden states

- **Solution:** Use a *different* weighted average for each output word
  - The weighted average provided for the kth output word is:

\[
c_0 = \frac{1}{N} \sum_{i=0}^{N} w_i(0) h_i
\]
Using all input hidden states

• **Solution:** Use a *different* weighted average for each output word
  
  – The weighted average provided for the kth output word is:

  \[
c_1 = \frac{1}{N} \sum_{i}^{N} w_i(1)h_i
\]
Using all input hidden states

- **Solution:** Use a *different* weighted average for each output word
  - The weighted average provided for the kth output word is:

\[
c_2 = \frac{1}{N} \sum_{i}^{N} w_i(2) h_i
\]
Using all input hidden states

- **Solution:** Use a *different* weighted average for each output word
  - The weighted average provided for the kth output word is:

\[
c_3 = \frac{1}{N} \sum_{i}^{N} w_i(3)h_i
\]
• **Solution:** Use a *different* weighted average for each output word
  - The weighted average provided for the kth output word is:

\[
c_k = \frac{1}{N} \sum_{i=1}^{N} w_i(k) h_i
\]
Using all input hidden states

- **Solution**: Use a *different* weighted average for each output word
  - The weighted average provided for the $k$th output word is:
    \[
    c_5 = \frac{1}{N} \sum_{i=1}^{N} w_i(5) h_i
    \]
Using all input hidden states

\[ c_t = \frac{1}{N} \sum_{i} w_i(t) h_i \]

- This solution will work if the weights \( w_{ki} \) can somehow be made to “focus” on the right input word
  - E.g., when predicting the word “apfel”, \( w_3(4) \), the weight for “apple” must be high while the rest must be low
- How do we generate such weights??

Ich habe einen apfel gegessen <eos>

I ate an apple <eos>
Attention Models

Attention weights: The weights $w_i(t)$ are dynamically computed as functions of decoder state

- Expectation: if the model is well-trained, this will automatically “highlight” the correct input

But how are these computed?
Attention weights at time \( t \)

- The “attention” weights \( w_i(t) \) at time \( t \) must be computed from available information at time \( t \)

- The primary information is \( s_{t-1} \) (the state at time \( t - 1 \))
  - Also, the input word at time \( t \), but generally not used for simplicity

\[
c_t = \frac{1}{N} \sum_{i}^{N} w_i(t)h_i
\]

\[
w_i(t) = a(h_i, s_{t-1})
\]
Requirement on attention weights

\[ c_i = \frac{1}{N} \sum_{i}^{N} w_i(t) h_i \]

\( w_i(t) \): Sum to 1.0

- The weights \( w_i(t) \) must be positive and sum to 1.0
  - I.e. be a distribution
  - Ideally, they must be high for the most relevant inputs for the ith output and low elsewhere
The weights $w_i(t)$ must be positive and sum to 1.0

- I.e. be a distribution
- Ideally, they must be high for the most relevant inputs for the ith output and low elsewhere

Solution: A two step weight computation

- First compute raw weights (which could be +ve or –ve)
- Then softmax them to convert them to a distribution

\[ e_i(t) = g(h_i, s_{t-1}) \]
\[ w_i(t) = \frac{\exp(e_i(t))}{\sum_j \exp(e_j(t))} \]
The attention framework computes a different “context” vector at each output step (T/F)
- True
- False

The context vector is chosen as the hidden (encoder) representation of the input word that is assigned the highest attention weight (T/F)
- True
- False

The attention weight to any input word is a function of the hidden encoder representation of the word and the most recent decoder state (T/F)
- True
- False
The attention framework computes a different “context” vector at each output step (T/F)

- True
- False

The context vector is chosen as the hidden (encoder) representation of the input word that is assigned the highest attention weight (T/F)

- True
- False

The attention weight to any input word is a function of the hidden encoder representation of the word and the most recent decoder state (T/F)

- True
- False
Requirement on attention weights

- The weights \( w_i(t) \) must be positive and sum to 1.0
  - I.e. be a distribution
  - Ideally, they must be high for the most relevant inputs for the ith output and low elsewhere
- Solution: A two step weight computation
  - First compute raw weights (which could be +ve or -ve)
  - Then softmax them to convert them to a distribution

\[
c_i = \frac{1}{N} \sum_{i=1}^{N} w_i(t) h_i
\]

\[
w_i(t): \text{Sum to 1.0}
\]

\[
e_i(t) = g(h_i, s_{t-1})
\]

\[
w_i(t) = \frac{\exp(e_i(t))}{\sum_j \exp(e_j(t))}
\]

\[
\text{Ich habe einen}
\]

\[
<h_{-1} \rightarrow h_0 \rightarrow h_1 \rightarrow h_2 \rightarrow h_3 \rightarrow h_4>
\]

\[
\text{I ate an apple <eos>}
\]

What is this function?
Attention weights

\[ c_i = \frac{1}{N} \sum_{i}^{N} w_i(t)h_i \]

\( w_i(t): \text{Sum to 1.0} \)

- Typical options for \( g() \) (variables in red must be learned)

\[
\begin{align*}
g(h_i, s_{t-1}) &= h_i^T s_{t-1} \\
g(h_i, s_{t-1}) &= h_i^T W_g s_{t-1} \\
g(h_i, s_{t-1}) &= v_g^T \tanh(W_g \begin{bmatrix} h_i \\ s_{t-1} \end{bmatrix}) \\
g(h_i, s_{t-1}) &= \text{MLP}([h_i, s_{t-1}]) \\
e_i(t) &= g(h_i, s_{t-1}) \\
w_i(t) &= \frac{\exp(e_i(t))}{\sum_j \exp(e_j(t))}
\end{align*}
\]
Attention weights

Let's consider a typical conversion process assuming this model as an example

- Typical options for $g()$ (variables in red must be learned)

\[ c_i = \frac{1}{N} \sum_{i}^{N} w_i(t)h_i \]

\[ w_i(t): \text{Sum to 1.0} \]

\[ g(h_i, s_{t-1}) = h_i^T s_{t-1} \]
\[ g(h_i, s_{t-1}) = h_i^T W_g s_{t-1} \]
\[ g(h_i, s_{t-1}) = \mathbf{v}_g^T \tanh \left( W_g \begin{bmatrix} h_i \\ s_{t-1} \end{bmatrix} \right) \]
\[ g(h_i, s_{t-1}) = \text{MLP}([h_i, s_{t-1}]) \]

\[ e_i(t) = g(h_i, s_{t-1}) \]
\[ w_i(t) = \frac{\exp(e_i(t))}{\sum_j \exp(e_j(t))} \]
Converting an input: Inference

Pass the input through the encoder to produce hidden representations $h_i$. 

Diagram:

- $h_{-1}$
- $h_0$ (I)
- $h_1$ (ate)
- $h_2$ (an)
- $h_3$ (apple <eos>)
- $h_4$
Converting an input: Inference

Pass the input through the encoder to produce hidden representations $h_i$

This may be
- a learned parameter, or
- Or just set to some fixed value, e.g. a vector of 1s or 0s, or
- Or the average of all the encoder embeddings: $mean(h_0, ..., h_4)$
- Or $W_{init} mean(h_0, ..., h_4)$ where $W_{init}$ is a learned parameter
Converting an input: Inference

\[ c_0 = \frac{1}{N} \sum_{i} w_i(0) h_i \]

- Compute the attention weights \( w_i(0) \) for the first output using \( s_{-1} \)
  - Will be a distribution over the input words
- Compute “context” \( c_0 \)
  - Weighted sum of input word hidden states
- Input \( c_0 \) and \(<sos>\) to the decoder at time 0
  - \(<sos>\) because we are starting a new sequence
  - In practice we will enter the embedding of \(<sos>\)
Converting an input: Inference

\[ c_0 = \frac{1}{N} \sum_{i}^{N} w_i(0) h_i \]

- The decoder computes
  - \( s_0 \)
  - A probability distribution over the output vocabulary
    - Output of softmax output layer
Converting an input: Inference

\[ c_0 = \frac{1}{N} \sum_{i} w_i(0) h_i \]

- Sample a word from the output distribution

\[ g(h_i, s_{-1}) = h_i^T W_g s_{-1} \]

\[ e_i(0) = g(h_i, s_{-1}) \]

\[ w_i(0) = \frac{\exp(e_i(0))}{\sum_j \exp(e_j(0))} \]
Converting an input: Inference

- Compute the attention weights $w_i(1)$ over all inputs for the second output using $s_0$
  - Compute raw weights, followed by softmax
- Compute “context” $c_1$
  - Weighted sum of input hidden representations
- Input $c_1$ and first output word to the decoder
  - In practice we enter the embedding of the word

$$g(h_i, s_0) = h_i^T W_g s_0$$
$$e_i(1) = g(h_i, s_0)$$
$$w_i(1) = \frac{\exp(e_i(1))}{\sum_j \exp(e_j(1))}$$
$$c_1 = \frac{1}{N} \sum_i^N w_i(1)h_i$$
Converting an input: Inference

- The decoder computes
  - $s_1$
  - A probability distribution over the output vocabulary

\[ g(h_i, s_0) = h_i^T W_g s_0 \]
\[ e_i(1) = g(h_i, s_0) \]
\[ w_i(1) = \frac{\exp(e_i(1))}{\sum_j \exp(e_j(1))} \]
\[ c_1 = \frac{1}{N} \sum_i w_i(1) h_i \]
Converting an input: Inference

\[ c_1 = \frac{1}{N} \sum_{i}^N w_i(1) h_i \]

- Sample the second word from the output distribution
Converting an input: Inference

\[ g(h_i, s_1) = h_i^T W_g s_1 \]

\[ e_i(2) = g(h_i, s_1) \]

\[ w_i(2) = \frac{\exp(e_i(2))}{\sum_j \exp(e_j(2))} \]

\[ c_2 = \frac{1}{N} \sum_i^N w_i(2) h_i \]
Converting an input: Inference

\[ g(h_i, s_1) = h_i^T W_g s_1 \]
\[ e_i(2) = g(h_i, s_1) \]
\[ w_i(2) = \frac{\exp(e_i(2))}{\sum_j \exp(e_j(2))} \]
\[ c_2 = \frac{1}{N} \sum_{i}^{N} w_i(2) h_i \]
Converting an input: Inference

\[
g(h_i, s_1) = h_i^T W_g s_1
\]

\[
e_i(2) = g(h_i, s_1)
\]

\[
w_i(2) = \frac{\exp(e_i(2))}{\sum_j \exp(e_j(2))}
\]

\[
c_2 = \frac{1}{N} \sum_i^N w_i(2) h_i
\]
Converting an input: Inference

\[
e_i(3) = g(h_i, s_2)
\]

\[
w_i(3) = \frac{\exp(e_i(3))}{\sum_j \exp(e_j(3))}
\]

\[
c_3 = \frac{1}{N} \sum_i w_i(3) h_i
\]
Converting an input: Inference

\[ e_i(3) = g(h_i, s_2) \]

\[ w_i(3) = \frac{\exp(e_i(3))}{\sum_j \exp(e_j(3))} \]

\[ c_3 = \frac{1}{N} \sum_i^N w_i(3) h_i \]
Converting an input: Inference

\[ c_3 \]

\[ h_{-1} \rightarrow h_0 \rightarrow h_1 \rightarrow h_2 \rightarrow h_3 \rightarrow h_4 \]

I  ate  an  apple <eos>

\[ e_i(3) = g(h_i, s_2) \]

\[ w_i(3) = \frac{\exp(e_i(3))}{\sum_j \exp(e_j(3))} \]

\[ c_3 = \frac{1}{N} \sum_{i=1}^{N} w_i(3) h_i \]
Converting an input: Inference

\[ e_i(4) = g(h_i, s_3) \]

\[ w_i(4) = \frac{\exp(e_i(4))}{\sum_j \exp(e_j(4))} \]

\[ c_4 = \frac{1}{N} \sum_{i=1}^{N} w_i(4) h_i \]
Converting an input: Inference

\[ e_i(4) = g(h_i, s_3) \]

\[ w_i(4) = \frac{\exp(e_i(4))}{\sum_j \exp(e_j(4))} \]

\[ c_4 = \frac{1}{N} \sum_i w_i(4) h_i \]
Converting an input: Inference

$e_i(4) = g(h_i, s_3)$

$w_i(4) = \frac{\exp(e_i(4))}{\sum_j \exp(e_j(4))}$

$c_4 = \frac{1}{N} \sum_i^N w_i(4)h_i$
Converting an input: Inference

\[ e_i(5) = g(h_i, s_4) \]

\[ w_i(5) = \frac{\exp(e_i(5))}{\sum_j \exp(e_j(5))} \]

\[ c_5 = \frac{1}{N} \sum_i^N w_i(5) h_i \]
Converting an input: Inference

\[ e_i(5) = g(h_i, s_4) \]

\[ w_i(5) = \frac{\exp(e_i(5))}{\sum_j \exp(e_j(5))} \]

\[ c_5 = \frac{1}{N} \sum_i w_i(5) h_i \]
Converting an input: Inference

Continue this process until <eos> is drawn

\[ e_i(5) = g(h_i, s_4) \]
\[ w_i(5) = \frac{\exp(e_i(5))}{\sum_j \exp(e_j(5))} \]
\[ c_5 = \frac{1}{N} \sum_i w_i(5) h_i \]
Attention-based decoding

\[ e_i(t) = g(h_i, s_{t-1}) \]

\[ w_i(t) = \frac{\exp(e_i(t))}{\sum_j \exp(e_j(t))} \]

\[ c_t = \frac{1}{N} \sum_i w_i(t) h_i \]
Modification: Query key value

- Encoder outputs an explicit “key” and “value” at each input time
  - Key is used to evaluate the importance of the input at that time, for a given output
- Decoder outputs an explicit “query” at each output time
  - Query is used to evaluate which inputs to pay attention to
- The weight is a function of key and query
- The actual context is a weighted sum of value
Modification: Query key value

- Encoder outputs an explicit “key” and “value” at each input time
  - Key is used to evaluate the importance of the input at that time, for a given output
- Decoder outputs an explicit “query” at each output time
  - Query is used to evaluate which inputs to pay attention to
- The weight is a function of key and query
- The actual context is a weighted sum of value

\[
e_i(t) = g(k_i, q_t)
\]

\[
w_i(t) = \text{softmax}(e_i(t))
\]
Input to hidden decoder layer: $\sum_i w_i(t)v_i$

$e_i(t) = g(k_i, q_t)$

$w_i(t) = \text{softmax}(e_i(t))$

- Encoder outputs an explicit "key" and "value" at each input time
- Key is used to evaluate the importance of the input at that time, for a given output
- Decoder outputs an explicit "query" at each output time
- Query is used to evaluate which inputs to pay attention to
- The weight is a function of key and query
- The actual context is a weighted sum of value

Special case: $k_i = v_i = h_i$

$q_t = s_{t-1}$

We will continue using this assumption in the following slides but in practice the query-key-value format is used
Pseudocode

# Assuming encoded input
# (K, V) = [k_enc[0]… k_enc[T]], [v_enc[0]… v_enc[T]]
# is available

t = -1
h_out[-1] = 0  # Initial Decoder hidden state
q[0] = 0       # Initial query

# Note: begins with a “start of sentence” symbol
# <sos> and <eos> may be identical
Y_out[0] = <sos>

do
t = t + 1
    C = compute_context_with_attention(q[t], K, V)
    y[t], h_out[t], q[t+1] = RNN_decode_step(h_out[t-1], y_out[t-1], C)
    y_out[t] = generate(y[t])  # Random, or greedy
until y_out[t] == <eos>
# Takes in previous state, encoder states, outputs attention-weighted context

```python
function compute_context_with_attention(q, K, V)
    # First compute attention
    e = []
    for t = 1:T  # Length of input
        e[t] = raw_attention(q, K[t])
    end
    maxe = max(e)  # subtract max(e) from everything to prevent underflow
    a[1..T] = exp(e[1..T] - maxe)  # Component-wise exponentiation
    suma = sum(a)  # Add all elements of a
    a[1..T] = a[1..T]/suma

    C = 0
    for t = 1..T
        C += a[t] * V[t]
    end

    return C
```
As before, the objective of drawing: Produce the most likely output (that ends in an <eos>)

\[
\arg\max_{y_1^{o_1}, y_1^{o_2}, \ldots, y_1^{o_L}} y_1^{o_1} y_1^{o_2} \ldots y_1^{o_L}
\]

- Simply selecting the most likely symbol at each time may result in suboptimal output
Solution: Multiple choices

- Retain all choices and *fork* the network
  - With every possible word as input
To prevent blowup: Prune

- Prune
  - At each time, retain only the top K scoring forks

\[ \text{Top}_K P(O_1|I_1, \ldots, I_N) \]
• At each time, retain only the top K scoring forks
Decoding

- At each time, retain only the top K scoring forks

Note: based on product

\[ \text{Top}_K P(O_2 O_1 | I_1, \ldots, I_N) \]

\[ = \text{Top}_K P(O_2 | O_1, I_1, \ldots, I_N) P(O_1 | I_1, \ldots, I_N) \]
Decoding

Note: based on product

\[ \text{Top}_K P(O_2O_1|I_1, ..., I_N) \]

\[ = \text{Top}_K P(O_2|O_1, I_1, ..., I_N)P(O_1|I_1, ..., I_N) \]

- At each time, retain only the top K scoring forks
Decoding

• At each time, retain only the top $K$ scoring forks

$$= \text{Top}_K P(O_2|O_1, I_1, ..., I_N) \times P(O_2|O_1, I_1, ..., I_N) \times P(O_1|I_1, ..., I_N)$$
• At each time, retain only the top K scoring forks
Decoding

At each time, retain only the top K scoring forks

$$\text{Top}_K \prod_{t=1}^{n} P(O_n|O_1, \ldots, O_{n-1}, I_1, \ldots, I_N)$$
• **Terminate**
  – When the current most likely path overall ends in `<eos>`
    • Or continue producing more outputs (each of which terminates in `<eos>`) to get N-best outputs
**Termination: <eos>**

- **Terminate**
  - Paths cannot continue once the output an <eos>
    - So paths may be different lengths
      - Select the most likely sequence ending in <eos> across all terminating sequences
Pseudocode: Beam search

# Assuming encoder output H = h_{in}[1]... h_{in}[T] is available
path = <sos>
beam = {path}
pathscore = [path] = 1
state[path] = h[0]  # initial state (computed using your favorite method)
do  # Step forward
    nextbeam = {}
    nextpathscore = []
    nextstate = {}
    for path in beam:
        cfin = path[end]
hpath = state[path]
C = compute_context_with_attention(hpath, H)
y, h = RNN_decode_step(hpath, cfin, C)
    for c in Symbolset
        newpath = path + c
        nextstate[newpath] = h
        nextpathscore[newpath] = pathscore[path]*y[c]
        nextbeam += newpath  # Set addition
    end
end
beam, pathscore, state, bestpath = prune(nextstate,nextpathscore,nextbeam)
until bestpath[end] = <eos>
Pseudocode: Beam search

# Assuming encoder output H = h_{in}[1]... h_{in}[T] is available
path = <sos>
beam = {path}
pathscore = [path] = 1
state[path] = h[0]  # computed using your favorite method
context[path] = compute_context_with_attention(h[0], H)
do  # Step forward
  nextbeam = {}
  nextpathscore = []
  nextstate = {}
  nextcontext = {}
  for path in beam:
    cfin = path[end]
    hpath = state[path]
    C = context[path]
    y, h = RNN_decode_step(hpath, cfin, C)
    nextC = compute_context_with_attention(h, H)
    for c in Symbolset
      newpath = path + c
      nextstate[newpath] = h
      nextcontext[newpath] = nextC
      nextpathscore[newpath] = pathscore[path]*y[c]
      nextbeam += newpath # Set addition
  end
  beam, pathscore, state, context, bestpath =
    prune (nextstate, nextpathscore, nextbeam, nextcontext)
until bestpath[end] = <eos>

Slightly more efficient.
Does not perform redundant context computation
What does the attention learn?

- The key component of this model is the attention weight
  - It captures the relative importance of each position in the input to the current output

\[ z_1 = \sum_i w_i(1)h_i \]

\[ g(h_i, s_0) = h_i^T W_g s_0 \]

\[ e_i(1) = g(h_i, s_0) \]

\[ w_i(1) = \frac{\exp(e_i(1))}{\sum_j \exp(e_j(1))} \]
“Alignments” example: Bahdanau et al.

Plot of $w_i(t)$
Color shows value (white is larger)

Note how most important input words for any output word get automatically highlighted.

The general trend is somewhat linear because word order is roughly similar in both languages.
## Translation Examples

<table>
<thead>
<tr>
<th>Source</th>
<th>This kind of experience is part of Disney’s efforts to &quot;extend the lifetime of its series and build new relationships with audiences via digital platforms that are becoming ever more important,&quot; he added.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reference</td>
<td>Ce type d’expérience entre dans le cadre des efforts de Disney pour &quot;étendre la durée de vie de ses séries et construire de nouvelles relations avec son public grâce à des plateformes numériques qui sont de plus en plus importantes&quot;, a-t-il ajouté.</td>
</tr>
<tr>
<td>RNNenc-50</td>
<td>Ce type d’expérience fait partie des initiatives du Disney pour &quot;prolonger la durée de vie de ses nouvelles et de développer des liens avec les lecteurs numériques qui deviennent plus complexes.</td>
</tr>
<tr>
<td>RNNsearch-50</td>
<td>Ce genre d’expérience fait partie des efforts de Disney pour &quot;prolonger la durée de vie de ses séries et créer de nouvelles relations avec des publics via des plateformes numériques de plus en plus importantes&quot;, a-t-il ajouté.</td>
</tr>
<tr>
<td>Google Translate</td>
<td>Ce genre d’expérience fait partie des efforts de Disney à “étendre la durée de vie de sa série et construire de nouvelles relations avec le public par le biais des plate-formes numériques qui deviennent de plus en plus important”, at-il ajouté.</td>
</tr>
</tbody>
</table>

- Bahdanau et al. 2016
Training the network

• We have seen how a trained network can be used to compute outputs
  – Convert one sequence to another

• Let’s consider training..
Given training input (source sequence, target sequence) pairs

**Forward pass**: Pass the actual input sequence through the encoder
  - At each time the output is a probability distribution over words
• **Backward pass**: Compute a divergence between target output and output distributions
  – Backpropagate derivatives through the network
• **Backward pass:** Compute a divergence between target output and output distributions
  – Backpropagate derivatives through the network
• This approach, in fact, performs maximum-likelihood estimation of the model
  – The forward pass of training takes in both inputs and ground-truth output labels for generation
  – This forward pass is not like regular inference
• This approach, in fact, performs maximum-likelihood estimation of the model
  – The forward pass of training takes in both inputs and ground-truth output labels for generation
  – This forward pass is not like regular inference

• But there’s another more traditional mechanism for training
  – Where the model only gets the input during the forward pass,
    • The forward pass is identical to inference
  – and the actual ground truth label is only used to compute loss...
Training Paradigms

- The traditional mechanism for training
  - Why would this not work?

 Decoder does not see ground truth labels during inference

Complicated Neural Network Model

I ate an apple<eos>

\( \mathbf{y}_1 \quad \mathbf{y}_2 \quad \mathbf{y}_3 \quad \mathbf{y}_4 \quad \mathbf{y}_5 \quad \mathbf{y}_6 \)

Loss

Ich habe einen apfel gegessen<eos>
Training Paradigms

- The traditional mechanism for training
  - Why would this not work?

- The output could be *anything*
  - And in the early stages of training, *will* be random
  - Cannot really compute a loss

- Decoder does not see ground truth labels during inference
The maximum-likelihood approach ensures a one-to-one correspondence between outputs and target outputs.

But from the perspective of traditional training, it is “cheating” during the inference of the forward pass, where the ground truth is being passed in instead of drawn samples.

- “Teacher forcing”
- Is there a midway solution?
  - Use drawn words during the forward pass inference while maintaining one-to-one correspondence
**The mid-way solution**

Use the ground truth during forward pass inference, but occasionally pass drawn output instead of ground truth, as input.

**Backward pass:** Compute a divergence between target output and output distributions
  - Backpropagate derivatives through the network
Gumbel Noise trick

- Sampling is not differentiable

- The “Gumbel noise” trick:
  - “Reparametrization”:
    \[
    \text{RandomSample}(Y) = \arg\max_i (G_i + \log(Y))
    \]
  - \(G_i\) is drawn from the standard Gumbel distribution \(\text{Gumbel}(0,1)\)

- The “argmax” can be replaced by a “softmax”, making the process differentiable w.r.t. network outputs
  - \(\text{decoderoutput}(t) = \text{softmax}(G_i + \log(Y(t)))\)

- \(\nabla_{Y(t)} \text{decoderoutput}(t)\) is employed in the chain rule to pass derivatives from \(t+1\) back to \(Y(t)\)
Tricks of the trade...

• Teacher forcing:
  – Ideally we would only use the decoder output during inference
  – This will not be stable
  – Passing in ground truth instead is “teacher forcing”

• Sampling the output:
  – Sample the system output and
  – as input during training for only some of the time

• The “Gumbel noise” trick:
  – Sampling is not differentiable, and gradients cannot be passed through it
  – The “Gumbel noise” approach recasts sampling as computing the argmax of a Gumbel distribution, with the network output as parameters
  – The “argmax” can be replaced by a “softmax”, making the process differentiable w.r.t. network outputs
Various extensions

• Bidirectional processing of input sequence
  – Bidirectional networks in encoder

• Attention: Local attention vs global attention
  – E.g. “Effective Approaches to Attention-based Neural Machine Translation”, Luong et al., 2015
  – Other variants
Extensions: Multihead attention

- Have multiple query/key/value sets.
  - Each attention “head” uses one of these sets
  - The combined contexts from all heads are passed to the decoder
- Each “attender” focuses on a different aspect of the input that’s important for the decode
Which of the following will give you the optimal decode with an attention-based decoder?

- Full tree search
- Beam search

Mark all that are true

- In a query-key-value based attention mechanism, the key and value are used to compute attention weights
- Multi-head attention computes a separate set of keys and values for each head, at each input
- Multi-head attention computes a separate query for each head, at each output
- Training with teacher forcing computes the theoretically correct loss and minimizes it
Which of the following will give you the optimal decode with an attention-based decoder?

- Full tree search
- Beam search

Mark all that are true

- In a query-key-value based attention mechanism, the key and value are used to compute attention weights
- Multi-head attention computes a separate set of keys and values for each head, at each input
- Multi-head attention computes a separate query for each head, at each output
- Training with teacher forcing computes the theoretically correct loss and minimizes it
Some impressive results..

• Attention-based models are currently responsible for the state of the art in many sequence-conversion systems
  – Machine translation
    • Input: Text in source language
    • Output: Text in target language
  – Speech recognition
    • Input: Speech audio feature vector sequence
    • Output: Transcribed word or character sequence
Attention models in image captioning

“Show attend and tell: Neural image caption generation with visual attention”, Xu et al., 2016

- Encoder network is a convolutional neural network
  - Filter outputs at each location are the equivalent of $h_i$ in the regular sequence-to-sequence model
Recap

• Have looked at various forms of sequence-to-sequence models

• Generalizations of recurrent neural network formalisms

• For more details, please refer to papers
  – Post on piazza if you have questions

• Will appear in HW4: **Speech recognition with attention models**
Recap: Seq2Seq models

- The input sequence feeds into a recurrent structure
- The input sequence is terminated by an explicit <eos> symbol
  - The hidden activation at the <eos> “stores” all information about the sentence
- Subsequently a second RNN uses the hidden activation as initial state to produce a sequence of outputs
Recap: Attention Models

- Encoder recurrently produces hidden representations of input word sequence
- Decoder recurrently generates output word sequence
  - For each output word the decoder uses a weighted average of the hidden input representations as input “context”, along with the recurrent hidden state and the previous output word
Recap: Attention Models

Problem: Because of the recurrence, the hidden representation for any word is also influenced by all preceding words

- The decoder is actually paying attention to the sequence, and not just the word

If the decoder is automatically figuring out which words of the input to attend to at each time, is recurrence in the input even necessary?
Non-recurrent encoder

- Modification: Let us eliminate the recurrence in the encoder
Non-recurrent encoder

• But this will eliminate *context-specificity* in the encoder embeddings
  — The embedding for “an” must really depend on the remaining words
    • It could be translated to “ein”, “einer”, or “eines” depending on the context.

• Solution: Use the attention framework itself to introduce context-specificity in embeddings
Recap: Non-recurrent encoder

- The encoder in a sequence-to-sequence model can be composed without recurrence.
- Use the attention framework itself to introduce context-specificity in embeddings
  - “Self” attention
First, for every word in the input sequence we compute an initial representation—E.g. using a single MLP layer
Self attention

\[ q_i = W_q h_i \]
\[ k_i = W_k h_i \]
\[ v_i = W_v h_i \]

- Then, from each of the hidden representations, we compute a query, a key, and a value.
  - Using separate linear transforms
  - The weight matrices \( W_q \), \( W_k \) and \( W_v \) are learnable parameters
Self Attention

For each word, we compute an attention weight between that word and all other words.

- The raw attention of the $i$th word to the $j$th word is a function of query $q_i$ and key $k_j$.
- The raw attention values are put through a softmax to get the final attention weights.

$q_i = W_q h_i$

$k_i = W_k h_i$

$v_i = W_v h_i$

$w_{ij} = \text{attn}(q_i, k_{0:N})$

$e_{ij} = q_i^T k_j$

$w_{i0}, ..., w_{iN} = \text{softmax}(e_{i0}, ..., e_{iN})$
The updated representation for the word is the attention-weighted sum of the values for all words

– Including itself
• Compute query-key-value sets for every word

• For each word
  – Using the query for that word, compute attention weights for all words using their keys
  – Compute updated representation for the word as attention-weighted sum of values of all words
• Compute query-key-value sets for every word
• For each word
  – Using the query for that word, compute attention weights for all words using their keys
  – Compute updated representation for the word as attention-weighted sum of values of all words
\[ q_i = W_q h_i \]
\[ k_i = W_k h_i \]
\[ v_i = W_v h_i \]

- Compute query-key-value sets for every word
- For each word
  - Using the query for that word, compute attention weights for all words using their keys
  - Compute updated representation for the word as attention-weighted sum of values of all words

\[ w_{2j} = \text{attn}(q_2, k_{0:N}) \]

\[ o_2 = \sum_j w_{2j} v_j \]
Compute query-key-value sets for every word

- For each word
  - Using the query for that word, compute attention weights for all words using their keys
  - Compute updated representation for the word as attention-weighted sum of values of all words
• Compute query-key-value sets for every word
• For each word
  – Using the query for that word, compute attention weights for all words using their keys
  – Compute updated representation for the word as attention-weighted sum of values of all words
\[ w_{ij} = \text{attn}(q_i, k_{0:N}) \]

\[ o_i = \sum_j w_{ij} v_j \]

- Compute query-key-value sets for every word
- For each word
  - Using the query for that word, compute attention weights for all words using their keys
  - Compute updated representation for the word as attention-weighted sum of values of all words
This is a “single-head” self-attention block

\[ w_{ij} = \text{attn}(q_i, k_{0:N}) \]

\[ o_i = \sum_j w_{ij} v_j \]

\[ q_i = W_q h_i \]
\[ k_i = W_k h_i \]
\[ v_i = W_v h_i \]
We can have multiple such attention "heads" – Each will have an independent set of queries, keys and values – Each will obtain an independent set of attention weights • Potentially focusing on a different aspect of the input than other heads – Each computes an independent output

The final output is the concatenation of the outputs of these attention heads • "MULTI-HEAD ATTENTION" (actually Multi-head self attention)
• Multi-head self attention
  – Multiple self-attention modules in parallel
Typically, the output of the multi-head self attention is passed through one or more regular feedforward layers

- Affine layer followed by a non-linear activation such as ReLU
The entire unit, including multi-head self-attention module followed by MLP is a **multi-head self-attention block**.
The entire unit, including multi-head self-attention module followed by MLP is a **multi-head self-attention block**
• The encoder can include many layers of such blocks
• No need for recurrence...
Recap: The encoder in a sequence-to-sequence model can replace recurrence through a series of “multi-head self attention” blocks.
Recap: The encoder in a sequence-to-sequence model can replace recurrence through a series of “multi-head self attention” blocks.

But this still ignores relative position:
- A context word one word away is different from one 10 words away.
- The attention framework does not take distance into consideration.
• Note that the inputs are actually word embeddings
Note that the inputs are actually word embeddings. We add a “positional” encoding to them to capture the relative distance from one another.
• **Positional Encoding:** A sequence of vectors $P_0, ..., P_N$, to encode position
  
  – Every vector is unique (and uniquely represents time)
  – Relationship between $P_t$ and $P_{t+\tau}$ only depends on the distance between them

  \[ P_{t+\tau} = M_\tau P_t \]

  • The linear relationship between $P_t$ and $P_{t+\tau}$ enables the net to learn shift-invariant “gap” dependent relationships
Positional Encoding

A vector of sines and cosines of a harmonic series of frequencies

- Every $2l$-th component of $P_t$ is $\sin \omega_l t$
- Every $2l + 1$-th component of $P_t$ is $\cos \omega_l t$

- Never repeats
- Has the linearity property required

$P_t = \begin{bmatrix} \sin \omega_1 t \\ \cos \omega_1 t \\ \sin \omega_2 t \\ \cos \omega_2 t \\ \vdots \\ \sin \omega_{d/2} t \\ \sin \omega_{d/2} t \end{bmatrix}$

$\omega_l = \frac{1}{10000^{2l/d}}$

$P_{t+\tau} = M_\tau P_t$

$M_\tau = \text{diag} \left( \begin{bmatrix} \cos \omega_l \tau & \sin \omega_l \tau \\ -\sin \omega_l \tau & \cos \omega_l \tau \end{bmatrix} , l = 1 \ldots d/2 \right)$
• The linear relationship between $P_t$ and $P_{t+\tau}$ enables the net to learn shift-invariant “gap” dependent relationships.
The self-attending encoder!!
The self-attending encoder!!

Can we use self attention to replace recurrence in the decoder?
Self attention and masked self attention

- **Self attention in encoder**: Can use input embedding at time $t+1$ and further to compute output at time $t$, because all inputs are available.
Self attention and masked self attention

- **Self attention in decoder:** Decoder is sequential
  - Each word is produced using the previous word as input
  - Only embeddings until time t are available to compute the output at time t
- The attention will have to be “masked”, forcing attention weights for t+1 and later to 0
The “masked self attention block” includes an MLP after the masked self attention

- Like in the encoder
The “masked self attention block” sequentially computes outputs begin to end.

- Sequential nature of decoding prevents outputs from being computed in parallel.
- Unlike in an encoder.
Masked multi-head self-attention

$$q_i^a = W_q^a h_i$$
$$k_i^a = W_k^a h_i$$
$$v_i^a = W_v^a h_i$$

$$w_{ij}^a = \text{attn}(q_i^a, k_{0:i-1}^a)$$

$$o_i^a = \sum_j w_{ij}^a v_j^a$$

- The “masked multi-head self attention block” includes multiple masked attention heads
  - Like in the encoder
Masked multi-head self-attention block

The “masked multi-head self attention block” includes multiple masked attention heads
- Like in the encoder
The “masked multi-head self attention block” includes multiple masked attention heads, followed by an MLP:

- Like in the encoder
I ate an apple <eos>
Mark all that are true

- Self attention computed for an N-length input requires the computation of an N x N attention weight matrix for each head
- Masked self attention is only required in the first layer of the decoder. Subsequent layers see the entire output of the previous layers and can use full self attention
- We cannot combine recurrent layers with self attention layers
- Positional encodings are different in the encoder and decoder because the self attention in the decoder is masked.
Mark all that are true

• **Self attention computed for an N-length input requires the computation of an N x N attention weight matrix for each head**
• Masked self attention is only required in the first layer of the decoder. Subsequent layers see the entire output of the previous layers and can use full self attention
• We cannot combine recurrent layers with self attention layers
• Positional encodings are different in the encoder and decoder because the self attention in the decoder is masked.
Transformer: Attention is all you need

Transformer: A sequence-to-sequence model that replaces recurrence with positional encoding and multi-head self attention
– “Attention is all you need”

**Transformer**

- Transformer: tremendous decrease in model computation for similar performance as state-of-art translation models
- The last row in the table shows transformer performance
- The final two columns show computational cost.

### Table 2: The Transformer achieves better BLEU scores than previous state-of-the-art models on the English-to-German and English-to-French newstest2014 tests at a fraction of the training cost.

<table>
<thead>
<tr>
<th>Model</th>
<th>BLEU</th>
<th>Training Cost (FLOPs)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>EN-DE</td>
<td>EN-FR</td>
</tr>
<tr>
<td>ByteNet [18]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Deep-Att + PosUnk [39]</td>
<td>24.6</td>
<td>39.2</td>
</tr>
<tr>
<td>GNMT + RL [38]</td>
<td>25.16</td>
<td>40.46</td>
</tr>
<tr>
<td>ConvS2S [9]</td>
<td>26.03</td>
<td>40.56</td>
</tr>
<tr>
<td>MoE [32]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Deep-Att + PosUnk Ensemble [39]</td>
<td></td>
<td>40.4</td>
</tr>
<tr>
<td>GNMT + RL Ensemble [38]</td>
<td>26.30</td>
<td>41.16</td>
</tr>
<tr>
<td>ConvS2S Ensemble [9]</td>
<td>26.36</td>
<td><strong>41.29</strong></td>
</tr>
<tr>
<td>Transformer (base model)</td>
<td>27.3</td>
<td>38.1</td>
</tr>
<tr>
<td>Transformer (big)</td>
<td><strong>28.4</strong></td>
<td><strong>41.8</strong></td>
</tr>
</tbody>
</table>
Transformer

From “Attention is all you need”

<table>
<thead>
<tr>
<th>Model</th>
<th>BLEU EN-DE</th>
<th>BLEU EN-FR</th>
<th>Training Cost (FLOPs) EN-DE</th>
<th>Training Cost (FLOPs) EN-FR</th>
</tr>
</thead>
<tbody>
<tr>
<td>ByteNet [18]</td>
<td>23.75</td>
<td>39.2</td>
<td>1.0 ( \cdot ) 10^{20}</td>
<td></td>
</tr>
<tr>
<td>Deep-Att + PosUnk [39]</td>
<td>24.6</td>
<td>39.92</td>
<td>2.3 ( \cdot ) 10^{19}</td>
<td>1.4 ( \cdot ) 10^{20}</td>
</tr>
<tr>
<td>GNMT + RL [38]</td>
<td>25.16</td>
<td>40.46</td>
<td>9.6 ( \cdot ) 10^{18}</td>
<td>1.5 ( \cdot ) 10^{20}</td>
</tr>
<tr>
<td>ConvS2S [9]</td>
<td>26.03</td>
<td>40.56</td>
<td>2.0 ( \cdot ) 10^{19}</td>
<td>1.2 ( \cdot ) 10^{20}</td>
</tr>
<tr>
<td>MoE [32]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Deep-Att + PosUnk Ensemble [39]</td>
<td></td>
<td>40.4</td>
<td></td>
<td>8.0 ( \cdot ) 10^{20}</td>
</tr>
<tr>
<td>GNMT + RL Ensemble [38]</td>
<td>26.30</td>
<td>41.16</td>
<td>1.8 ( \cdot ) 10^{20}</td>
<td>1.1 ( \cdot ) 10^{21}</td>
</tr>
<tr>
<td>ConvS2S Ensemble [9]</td>
<td>26.36</td>
<td>41.29</td>
<td>7.7 ( \cdot ) 10^{19}</td>
<td>1.2 ( \cdot ) 10^{21}</td>
</tr>
<tr>
<td>Transformer (base model)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Transformer (big)</td>
<td><strong>28.4</strong></td>
<td><strong>41.8</strong></td>
<td><strong>3.3 ( \cdot ) 10^{18}</strong></td>
<td><strong>2.3 ( \cdot ) 10^{19}</strong></td>
</tr>
</tbody>
</table>

- Transformer: tremendous decrease in model computation for similar performance as state-of-art translation models
- The last row in the table shows transformer performance
- The final two columns show computational cost.
Recap: Vanishing/exploding gradients

\[ \nabla_{f_k} D_{iv} = \nabla D \cdot \nabla f_N \cdot W_N \cdot \nabla f_{N-1} \cdot W_{N-1} \ldots \nabla f_{k+1} W_{k+1} \]

- RNNs are just very deep networks
- LSTMs mitigate the problem at the cost of 3x more matrix multiplications
- Transformers get rid of it! To encode a full sentence, they have way fewer layers than an unrolled RNN.
- The same goes with the vanishing memory issue to an extent.
• Computing $Y(T)$ requires $Y(T - 1)$...
• Which requires $Y(T - 2)$, etc...
• RNN inputs must be processed in order $\Rightarrow$ slow implementation
• $q_n, k_n, v_n$ can be computed separately.
• $n^2 < q_n, k_n >$ dot products to compute.
• Self attention is easy to compute in parallel $\rightarrow$ Faster implementations
Transformer

- Transformer: tremendous decrease in model computation for similar performance as state-of-art translation models
- The last row in the table shows transformer performance
- The final two columns show computational cost.

Table 2: The Transformer achieves better BLEU scores than previous state-of-the-art models on the English-to-German and English-to-French newstest2014 tests at a fraction of the training cost.

<table>
<thead>
<tr>
<th>Model</th>
<th>BLEU</th>
<th>Training Cost (FLOPs)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>EN-DE</td>
<td>EN-FR</td>
</tr>
<tr>
<td>ByteNet [18]</td>
<td>23.75</td>
<td></td>
</tr>
<tr>
<td>Deep-Att + PosUnk [39]</td>
<td>24.6</td>
<td>39.2</td>
</tr>
<tr>
<td>GNMT + RL [38]</td>
<td>25.16</td>
<td>40.46</td>
</tr>
<tr>
<td>ConvS2S [9]</td>
<td>26.03</td>
<td>40.56</td>
</tr>
<tr>
<td>Deep-Att + PosUnk Ensemble [39]</td>
<td></td>
<td>40.4</td>
</tr>
<tr>
<td>GNMT + RL Ensemble [38]</td>
<td>26.30</td>
<td>41.16</td>
</tr>
<tr>
<td>ConvS2S Ensemble [9]</td>
<td>26.36</td>
<td><strong>41.29</strong></td>
</tr>
<tr>
<td>Transformer (base model)</td>
<td>27.3</td>
<td>38.1</td>
</tr>
<tr>
<td>Transformer (big)</td>
<td><strong>28.4</strong></td>
<td><strong>41.8</strong></td>
</tr>
</tbody>
</table>
GPT

Alec Radford et. al., Improving Language Understanding by Generative Pre-Training

Table 5: Analysis of various model ablations on different tasks. Avg. score is a unweighted average of all the results. \((mc=\text{Mathews correlation}, acc=\text{Accuracy}, pc=\text{Pearson correlation})\)

<table>
<thead>
<tr>
<th>Method</th>
<th>Avg. Score</th>
<th>CoLA (mc)</th>
<th>SST2 (acc)</th>
<th>MRPC (F1)</th>
<th>STSB (pc)</th>
<th>QQP (F1)</th>
<th>MNLI (acc)</th>
<th>QNLI (acc)</th>
<th>RTE (acc)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transformer w/ aux LM (full)</td>
<td>74.7</td>
<td>45.4</td>
<td>91.3</td>
<td>82.3</td>
<td>82.0</td>
<td>70.3</td>
<td>81.8</td>
<td>88.1</td>
<td>56.0</td>
</tr>
<tr>
<td>Transformer w/o pre-training</td>
<td>59.9</td>
<td>18.9</td>
<td>84.0</td>
<td>79.4</td>
<td>30.9</td>
<td>65.5</td>
<td>75.7</td>
<td>71.2</td>
<td>53.8</td>
</tr>
<tr>
<td>Transformer w/o aux LM</td>
<td>75.0</td>
<td>47.9</td>
<td>92.0</td>
<td>84.9</td>
<td>83.2</td>
<td>69.8</td>
<td>81.1</td>
<td>86.9</td>
<td>54.4</td>
</tr>
<tr>
<td>LSTM w/ aux LM</td>
<td>69.1</td>
<td>30.3</td>
<td>90.5</td>
<td>83.2</td>
<td>71.8</td>
<td>68.1</td>
<td>73.7</td>
<td>81.1</td>
<td>54.6</td>
</tr>
</tbody>
</table>

- GPT uses only the decoder of the transformer as an LM — “Transformer w/o aux LM”
- Large performance improvement in many tasks
• Add **Task conditioning**: put the nature of your task in the input (not just LM)
• Parameters x1000

→ **GPT-3**: Generalizes to more tasks, not just more inputs!
BERT

- Bert: Only uses encoder of transformer to derive word and sentence embeddings
- Trained to “fill in the blanks”
- This is representation learning (more next lecture)

<table>
<thead>
<tr>
<th>System</th>
<th>MNLI-(m/mm)</th>
<th>QQP</th>
<th>QNLI</th>
<th>SST-2</th>
<th>CoLA</th>
<th>STS-B</th>
<th>MRPC</th>
<th>RTE</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pre-OpenAI SOTA</td>
<td>80.6/80.1</td>
<td>66.1</td>
<td>82.3</td>
<td>93.2</td>
<td>35.0</td>
<td>81.0</td>
<td>86.0</td>
<td>61.7</td>
<td>74.0</td>
</tr>
<tr>
<td>BiLSTM+ELMo+Attn</td>
<td>76.4/76.1</td>
<td>64.8</td>
<td>79.8</td>
<td>90.4</td>
<td>36.0</td>
<td>73.3</td>
<td>84.9</td>
<td>56.8</td>
<td>71.0</td>
</tr>
<tr>
<td>OpenAI GPT</td>
<td>82.1/81.4</td>
<td>70.3</td>
<td>87.4</td>
<td>91.3</td>
<td>45.4</td>
<td>80.0</td>
<td>82.3</td>
<td>56.0</td>
<td>75.1</td>
</tr>
<tr>
<td>BERT BASE</td>
<td>84.6/83.4</td>
<td>71.2</td>
<td>90.5</td>
<td>93.5</td>
<td>52.1</td>
<td>85.8</td>
<td>88.9</td>
<td>66.4</td>
<td>79.6</td>
</tr>
<tr>
<td>BERT LARGE</td>
<td><strong>86.7/85.9</strong></td>
<td><strong>72.1</strong></td>
<td><strong>92.7</strong></td>
<td><strong>94.9</strong></td>
<td><strong>60.5</strong></td>
<td><strong>86.5</strong></td>
<td><strong>89.3</strong></td>
<td><strong>70.1</strong></td>
<td><strong>82.1</strong></td>
</tr>
</tbody>
</table>

Table 1: GLUE Test results, scored by the evaluation server (https://gluebenchmark.com/leaderboard). The number below each task denotes the number of training examples. The “Average” column is slightly different than the official GLUE score, since we exclude the problematic WNLI set. BERT and OpenAI GPT are single-model, single task. F1 scores are reported for QQP and MRPC, Spearman correlations are reported for STS-B, and accuracy scores are reported for the other tasks. We exclude entries that use BERT as one of their components.
Attention is all you need

• Self-attention can effectively replace recurrence in sequence-to-sequence models
  – “Transformers”
  – Requires “positional encoding” to capture positional information

• Can also be used in regular sequence analysis settings as a substitute for recurrence

• Currently the state of the art in most sequence analysis/prediction...
Attention is all you need

- Self-attention can effectively replace recurrence in sequence-to-sequence models
  - “Transformers”
  - Requires “positional encoding” to capture positional information

- Can also be used in regular sequence analysis settings as a substitute for recurrence

- Currently the state of the art in most sequence analysis/prediction... and even computer vision problems!
Poll 4 (@1217)

Mark all that are true

- BERT is essentially the encoder of a transformer model
- GPT is essentially the encoder of a transformer model
- BERT is essentially the decoder of a transformer model
- GPT is essentially the decoder of a transformer model
Mark all that are true

- BERT is essentially the encoder of a transformer model
- GPT is essentially the encoder of a transformer model
- BERT is essentially the decoder of a transformer model
- GPT is essentially the decoder of a transformer model
Vision Transformers

- Divide your image in patches with pos. encodings
- Apply Self-Attention!
  → Sequential and image problems are similar when using transformers

Dosovitskiy et al, An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale, 2020
Impact of Transformers

• Transformers have played a major role in the “uniformization” of DL-based tasks:
  – Find a pretrained “BERT-like” transformer (Text, Image, Speech)
  – Fine-tune on your task – or not! (Prompting…)
• This has helped democratize Deep Learning considerably

• But…
Caveat 1

• Not all transformers are the same: Big/small, fast/slow, mono-/multilingual, contrastive/generative, regressive/autoencoding...

• Pick the right one!
Caveat 2

• Transformers are not always the right choice.
• They often require more parameters than LSTMs at equal performance
  ➔ Tricky on small hardware (phones, IoT, etc)