CAS CS 505
Lec 18

Sampling Strategies for Generative Models; Transformer Architecture Continued; Pretrained Models; BERT

1. Some Strategies for Sampling in Generative Models
    a. In Generative Models, "Sampling" refers to the choice of the next token using the probability distribution output by the network.
    b. Let's look at a really simple example:
        i. Suppose the generative model is creating sentences from the vocabulary V = ['<s>', '</s>', 'NLP', 'early', 'fun', 'green', 'hard', 'interesting', 'is']
        ii. We have generated three tokens so far: ['<s>', 'NLP', 'is'] and the softmax output (probability distribution) for the next token is:

Token	Probability
<s>	0.0
</s>	0.01
NLP	0.03
early	0.12
fun	0.12
green	0.0
hard	0.2
interesting	0.51
is	0.01

    c. There are five common strategies for sampling ['<s>', 'NLP', 'is', ???]
        i. Sampling using the unmodified distribution

```
1 print('\nsample: ',sample_choice(outcomes,distribution,option=0))
```

```
<s>             0.0
</s>            0.01
NLP             0.03
early           0.12
fun             0.12
green           0.0
hard            0.2
interesting     0.51
is              0.01

sample:  fun
```

        ii. Greedy Sampling: Just choose the single most probable

```
1 print('\nsample: ',sample_choice(outcomes,distribution,option=1))
```

```
<s>             0.0
</s>            0.0
NLP             0.0
early           0.0
fun             0.0
green           0.0
hard            0.0
interesting     1.0
is              0.0

sample:  interesting
```

iii. Softmax with Temperature

```
print('\nsample: ',sample_choice(outcomes,distribution,option=2,temperature = 0.01))
```

```
<s>             7.095474162284459e-23
</s>            1.928749847963851e-22
NLP             1.4251640827408859e-21
early           1.1548224173015387e-17
fun             1.1548224173015387e-17
green           7.095474162284459e-23
hard            3.4424771084698575e-14
interesting     0.9999999999999655
is              1.928749847963851e-22

sample:  interesting
```

```
print('\nsample: ',sample_choice(outcomes,distribution,option=2,temperature = 0.1))
```

```
<s>             0.005446284988308871
</s>            0.0060190757806309345
NLP             0.0073517157600377
early           0.018082302955730302
fun             0.018082302955730302
green           0.005446284988308871
hard            0.040242905309378116
interesting     0.893310051481244
is              0.0060190757806309345

sample:  interesting
```

```
print('\nsample: ',sample_choice(outcomes,distribution,option=2,temperature = 0.5))
```

```
<s>             0.08396476946896886
</s>            0.08566097032727966
NLP             0.08915686084440715
early           0.10674014184435929
fun             0.10674014184435929
green           0.08396476946896886
hard            0.12526071682556345
interesting     0.23285065904881377
is              0.08566097032727966

sample:  </s>
```

```
print('\nsample: ',sample_choice(outcomes,distribution,option=2,temperature = 10.0))
```

```
<s>             0.10986989484510404
</s>            0.10997981969321279
NLP             0.11019999943895172
early           0.11119627595335972
fun             0.11119627595335972
green           0.10986989484510404
hard            0.11208941394957388
interesting     0.11561860562812123
is              0.10997981969321279

sample:  green
```

iv. Top-K Sampling: From top K most probable, choose proportionally:

```
print('\nsample: ',sample_choice(outcomes,distribution,option=3,K=4))
```

```
interesting     0.5368421052631579
hard            0.2105263157894737
fun             0.12631578947368421
early           0.12631578947368421

sample:  fun
```

Normalize to a probability distribution (sum = 1.0)

```
interesting     0.51
hard            0.2
fun             0.12
early           0.12
NLP             0.03
is              0.01
</s>            0.01
green           0.0
<s>             0.0
```

v. Top-P: From top choices with sum of probability p, choose proportionally:

```
print('\nsample: ',sample_choice(outcomes,distribution,option=4,p=0.75))
```

```
interesting     0.6144578313253012
hard            0.24096385542168675
fun             0.14457831325301204

sample:  interesting
```

Normalize to a probability distribution (sum = 1.0)

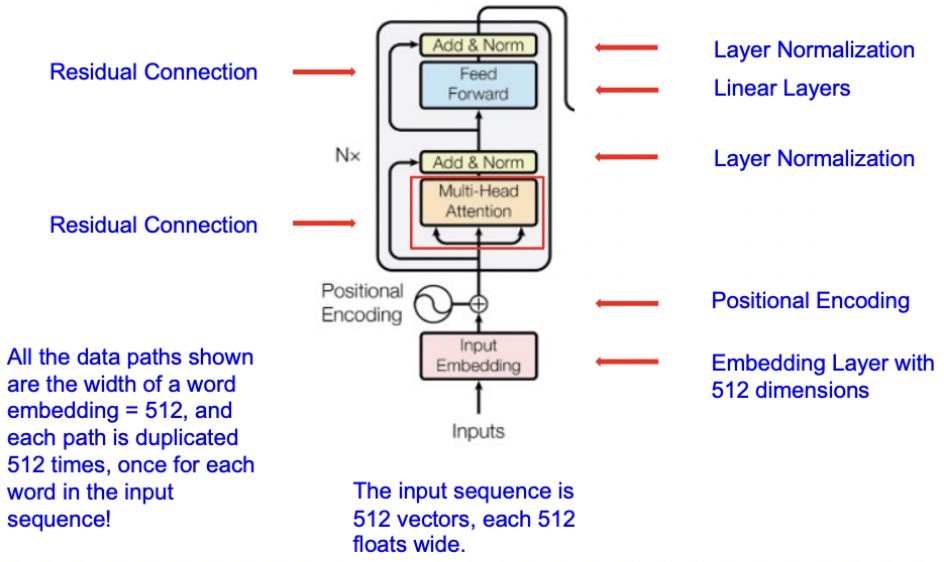
		Sum
interesting	0.51	0.51
hard	0.2	0.71
fun	0.12	0.83
early	0.12	0.95
NLP	0.03	0.98
is	0.01	0.99
</s>	0.01	1.0
green	0.0	
<s>	0.0	

2. Some Refinements for Sampling in Generative Models

a.

```python
# Options

#    0 = No modification  (default)
#    1 = Greedy: return one-hot of maximum prob, equivalent to softmax with 0 temperature
#    2 = Softmax with temperature (temperature of 1.0 is ordinary softmax)
#    3 = Top-K: Choose from top K most probable options (no other modification)
#    4 = Nucleus (Top-p): p is a cutoff, choose from the top choices whose cumulative prob just
#            exceeds p (so in general, cumulative prob is slightly more than p)


import math
import numpy as np
from numpy.random import choice

def sample_choice(outcomes,distribution,option=3,temperature=0.3,K=5,p=0.25):

    if type(outcomes) != np.ndarray:
        outcomes = np.array(outcomes)
    if type(distribution) != np.ndarray:
        distribution = np.array(distribution)


    if option == 0:
        return choice(a=outcomes,p=distribution/sum(distribution))
    elif option == 1:
        oneHot = np.array([0.0]*len(distribution))
        oneHot[np.argmax(distribution)] = 1.0
        return outcomes[np.argmax(distribution)]
    elif option == 2:
        sum_exp = sum(math.exp(x/temperature) for x in distribution)
        lst = [math.exp(x/temperature)/sum_exp for x in distribution]
        return choice(a=outcomes,p=[math.exp(x/temperature)/sum_exp for x in distribution])
    elif option == 3:
        indices = np.argsort(distribution)[-K:]
        return choice(a=outcomes[indices],p=distribution[indices]/sum(distribution[indices]))
    elif option == 4:
        indices = np.argsort(distribution)
        total = 0
        for C in range(len(distribution)-1,-1,-1):
            total += distribution[indices[C]]
            if total > p:
                break
        indices = indices[C:]
        return choice(a=outcomes[indices],p=distribution[indices]/sum(distribution[indices]))

```

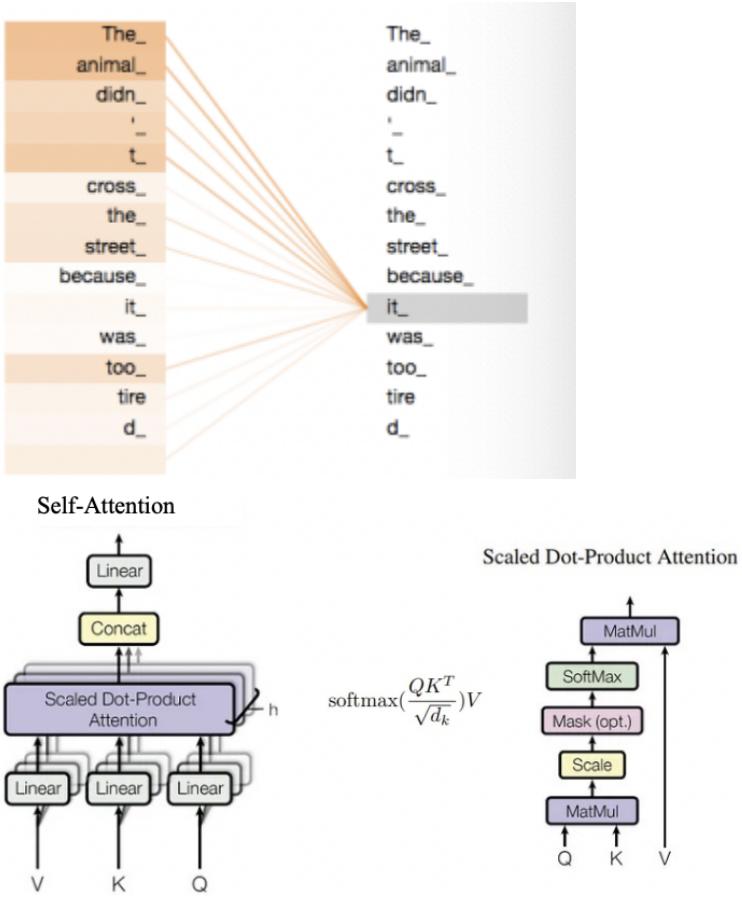
3. Transformer: Multi-Head Attention
    a. The center of the design is the Attention mechanism, here Multi-Head Attention….



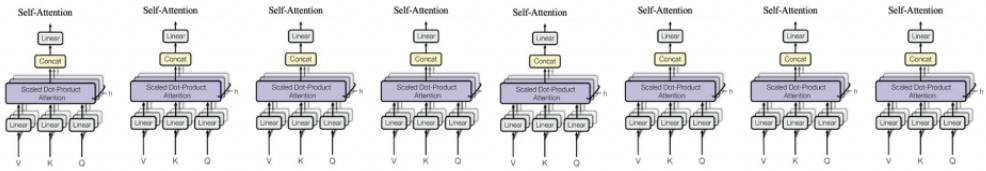
b.

c. The basic mechanism at work here is self-attention, where the input is processed to determine the dependencies between different words.

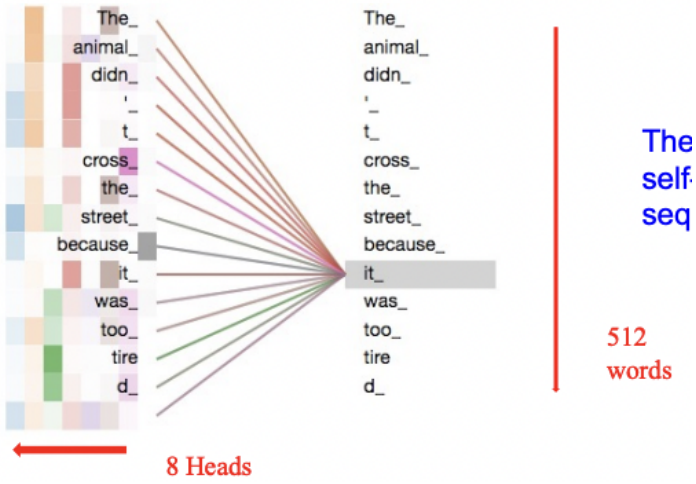
d. Self-attention is implemented by a series of linear transformations, scaled, and then softmaxed to produce the probability distribution which tells us how much each word depends on other words in the same sequence.



e. Multi-Head Attention applies this self-attention mechanism as 8 (or more) self-attention Heads:

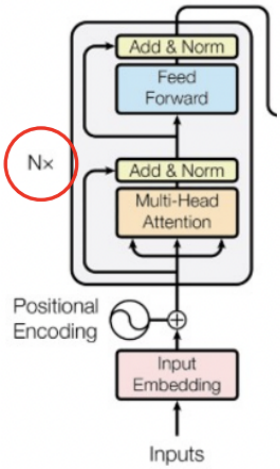


f. This allows the encoder to try to understand 8 different kinds of dependencies among the words in the input.



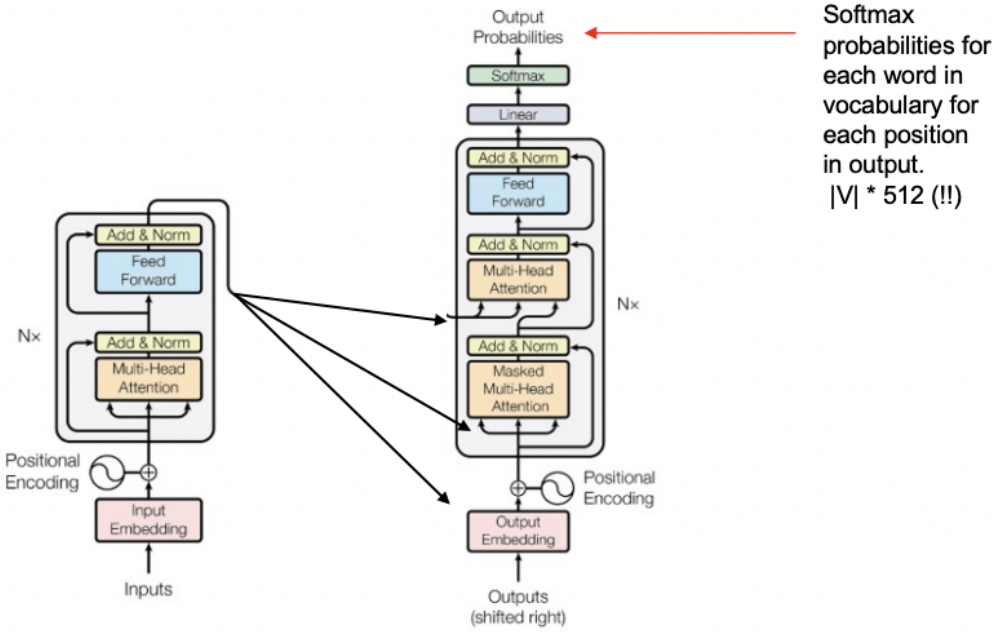
i. The original design applied 8 self-attention heads to sequences of length 512.

4. Transformer: Encoder
   a. Then, this encoder layer is stacked 6, 8, or more layers deep!
   b.

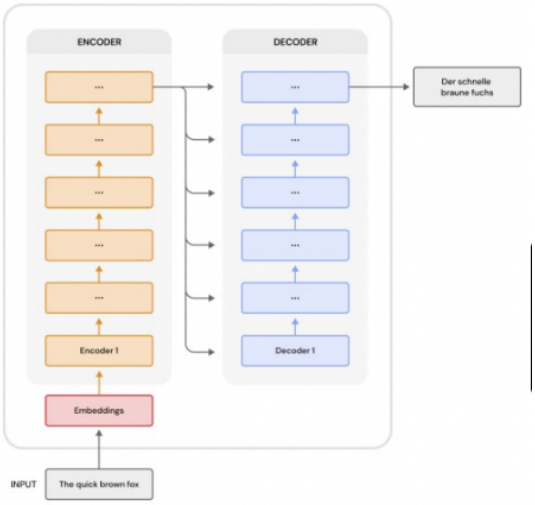


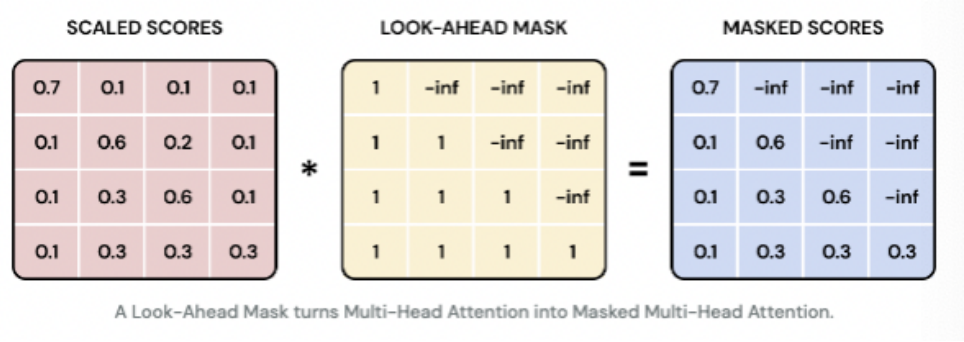
c.

5. Transformer: Decoder
    a. The Decoder stage has essentially the same features as the Encoder, except for a stage called Masked Multi-Head Attention.
    b. Apparently, the classic diagram (from the original paper) is a bit misleading, and there are additional connections between the encoder and decoder:



c. The Decoder stage has essentially the same features as the Encoder, except for Multi-Headed Attention is masked to simulate left to right generation of output; Each stage of Decoder takes the context vector (output of encoder) as an additional input:

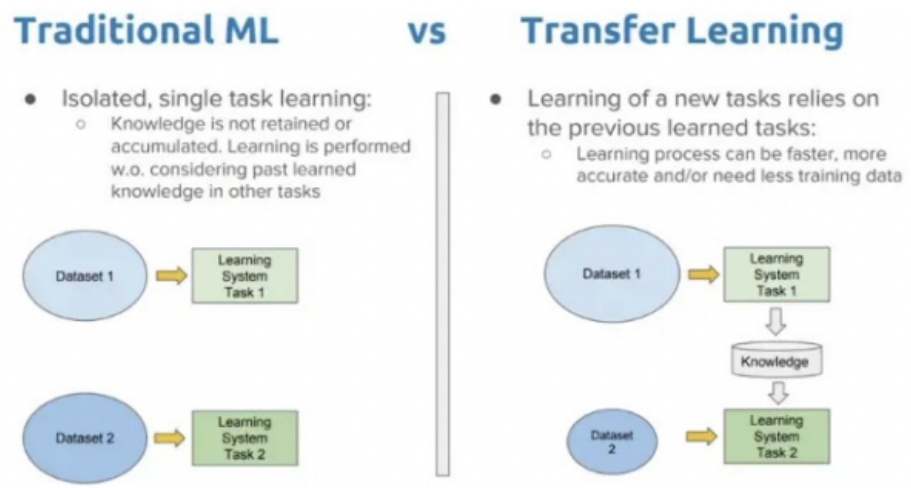




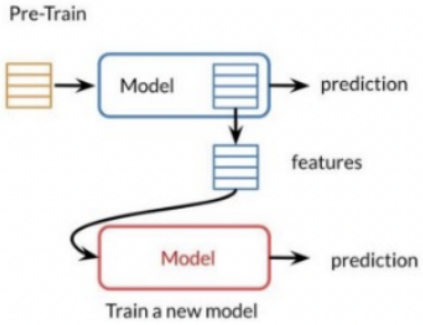
A Look-Ahead Mask turns Multi-Head Attention into Masked Multi-Head Attention.

6. Transfer Learning: Leveraging pre-trained models
    a. A hugely important development in ML in the last decade is the development of transfer learning systems, consisting of two stages:
        i. Pre-Training: A (large) model is trained on a (large) corpus to produce generic outputs;
        ii. Downstream: A NN is added on top of the pre-trained model to build models for a different task (called a "downstream task")

b.

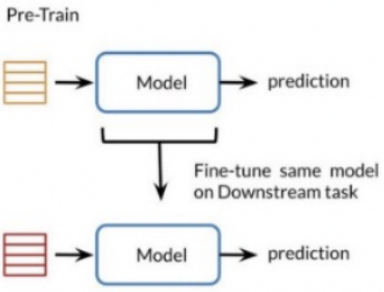


c. There are two flavors of transfer learning:
   i. Feature-based: Use outputs of pre-trained model as "features" input to downstream model; further training only occurs in downstream model:



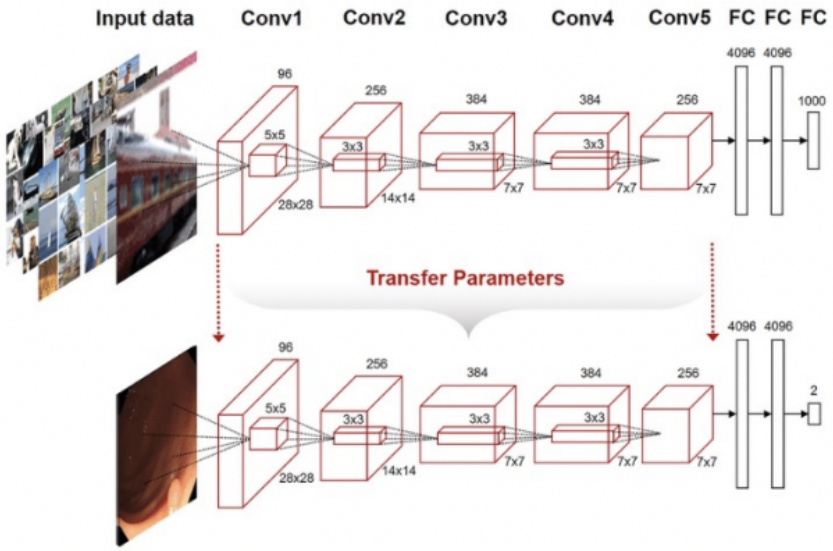
      1. Example: ELMO

   ii. Fine-Tuning: Outputs of pre-trained model are inputs to downstream model, but BOTH models are trained; typically, changes to pre-trained model are minimal.



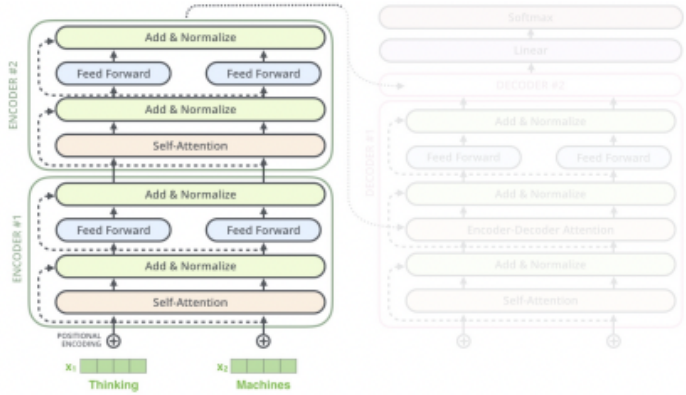
      1. Example: GPT, BERT

d. This was first used in image processing (but is now used everywhere):



AlexNet structure of transfer learning from ImageNet database. The parameters are transferred in all layers from Conv1 to Conv5 except FC.

7. Transfer Learning for NLP: GPT, ELMO, and BERT
    a. Pre-trained models are typically very large and trained for a long time on huge amounts of data (so they have a lot of knowledge you can leverage for your specific task.
    b. The most successful systems in NLP are
        i. GPT (Generative Pre-trained Transformer)
        ii. ELMO (Embeddings from Language Models)
        iii. BERT (Bidirectional Encoder Representations from Transformers)
    c. These all
        i. Are based on transformers, but use ONLY the encoder phase;
        ii. Produce contextual embeddings;
        iii. GPT is autoregressive (outputs feed back into inputs)



d. The most significant difference in the models is how they process the input sequence:
    i. GPT uses a simplified "causal" transformer model which only uses left-context:

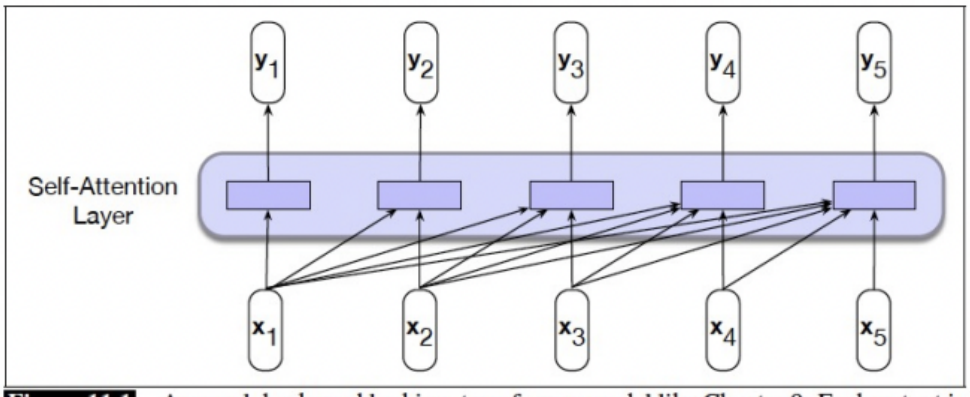


Figure 11.1 A causal, backward looking, transformer model like Chapter 9. Each output is computed independently of the others using only information seen earlier in the context.

    ii. ELMO and BERT use a bidirectional transformer architecture:

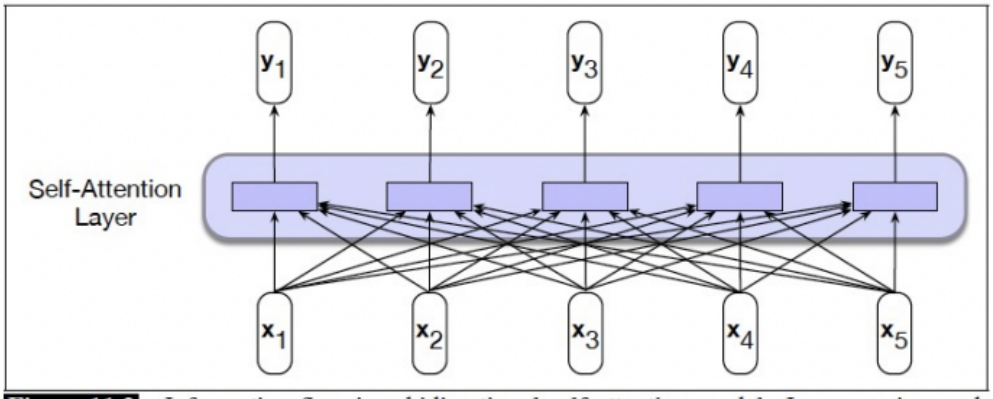
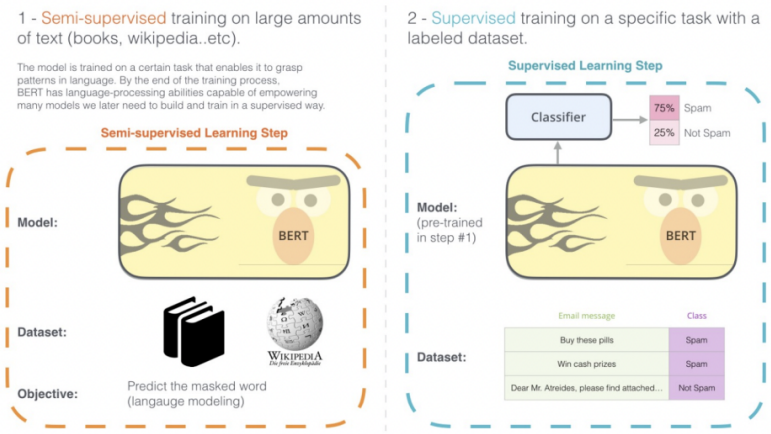
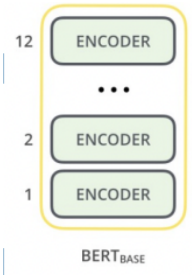


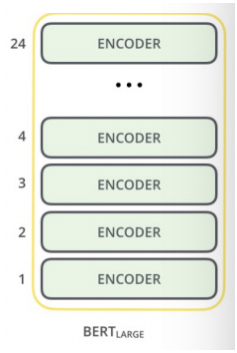
Figure 11.2 Information flow in a bidirectional self-attention model. In processing each element of the sequence, the model attends to all inputs, both before and after the current one.

8. BERT

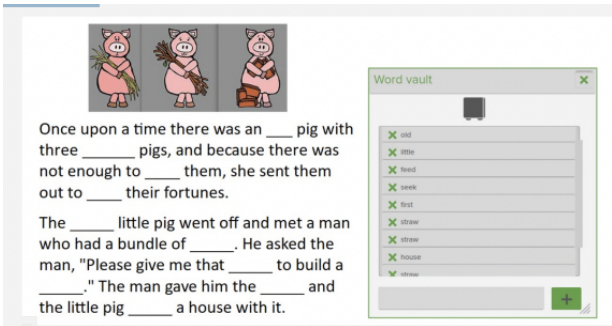
a.

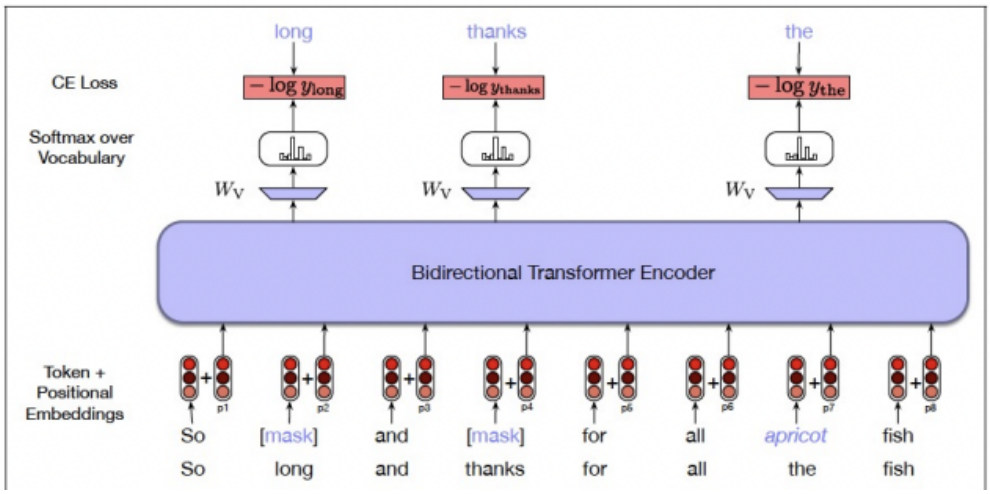


b. Bert has two implementations:
    i. Bert base:
        1. 12 layers; 12 attention heads per layer;
        2. 768 hidden units; 110 M parameters
        3. 

    ii. Bert large:
        1. 16 layers; 16 attention heads per layer;
        2. 1024 hidden units; 340 M parameters
        3. 

    iii. BERT added two significant ideas for training which allowed it to achieve SOTA performance on significant tasks:
        1. Training using a Masked Language Model (MLM);
        2. Focus on a "next sentence prediction" task with two sentences as input.
    iv. Note: The first versions of BERT used the 800 M word BooksCorpus and a 2.5 B word English Wikipedia corpus.
c. The Masked Language Model is based on an educational theory/testing paradigm known as the Cloze Task, where students learn a language by filling in blanks in a story or piece of text:



9. Training BERT
    a. Masked Language Modeling uses unannotated text from a large corpus. 15% of the words in the corpus are selected for the training phase:
        i. of these, 80% of replaced with the token [MASK]
        ii. 10% are replaced with randomly-selected tokens
        iii. 10% are left unchanged.
    b. The model is trained to predict the missing tokens
    c. 

    d. A variation of MLM uses spans (subsequences of the input sequence); all of the words in the span are replaced as before:

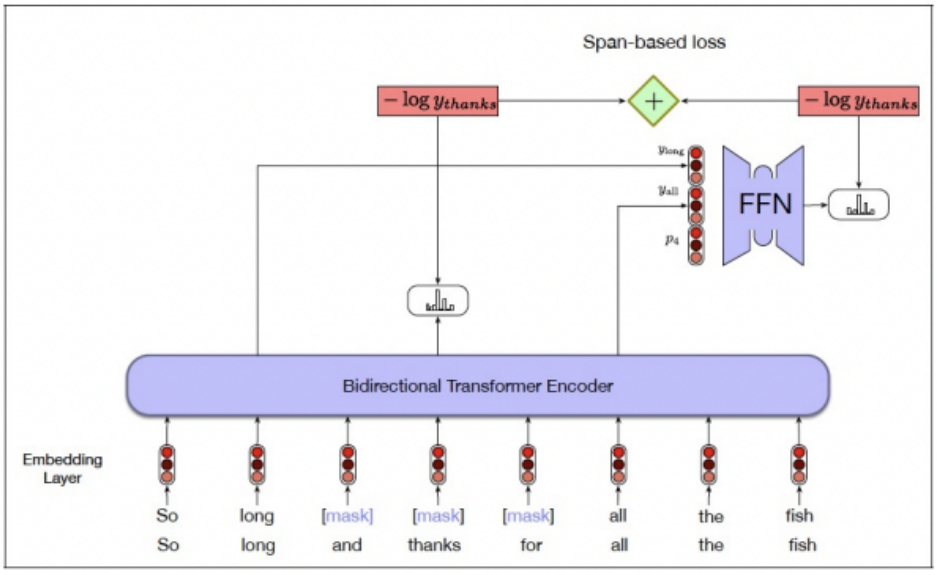


Figure 11.6 Span-based language model training. In this example, a span of length 3 is selected for training and all of the words in the span are masked. The figure illustrates the loss computed for word *thanks*; the loss for the entire span is based on the loss for all three of the words in the span.

e. The Next Sentence Prediction task is to input TWO sentences starting with the token [CLS] and separated by the token [SEP]. The training set has 50% sentences that are next to each other in the corpus, and 50% random sentences.

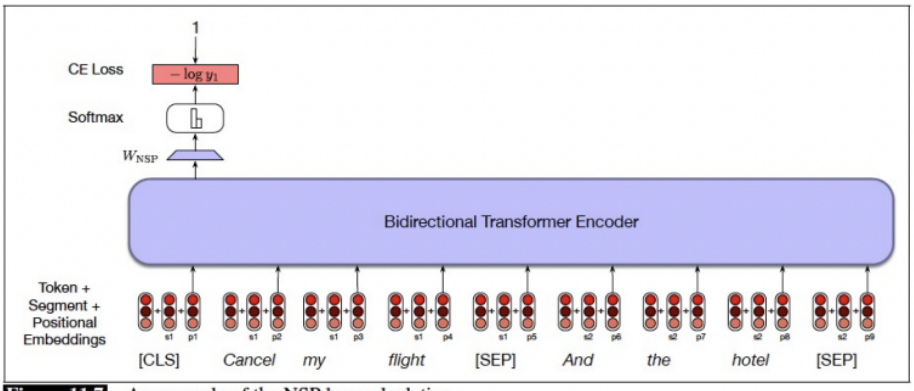


Figure 11.7 An example of the NSP loss calculation.

10. Using BERT

a. Bert can be used for sentence classification if a single sentence is input:

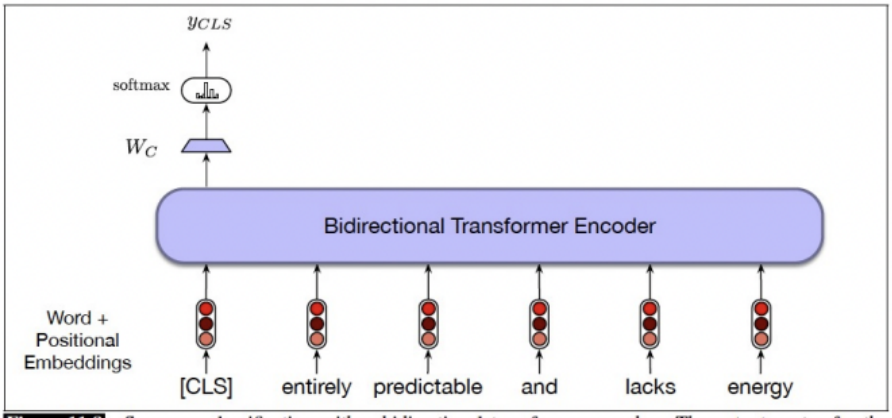


Figure 11.8 Sequence classification with a bidirectional transformer encoder. The output vector for the `[CLS]` token serves as input to a simple classifier.

b. BERT can classify the relationship between two sentences:

- `Neutral`
  - a: Jon walked back to the town to the smithy.
  - b: Jon traveled back to his hometown.
- `Contradicts`
  - a: Tourist Information offices can be very helpful.
  - b: Tourist Information offices are never of any help.
- `Entails`
  - a: I'm confused.
  - b: Not all of it is very clear to me.

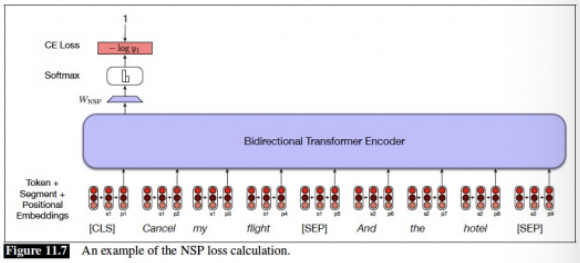


Figure 11.7 An example of the NSP loss calculation.

c. BERT can generate the most likely sentence to follow a given sentence:

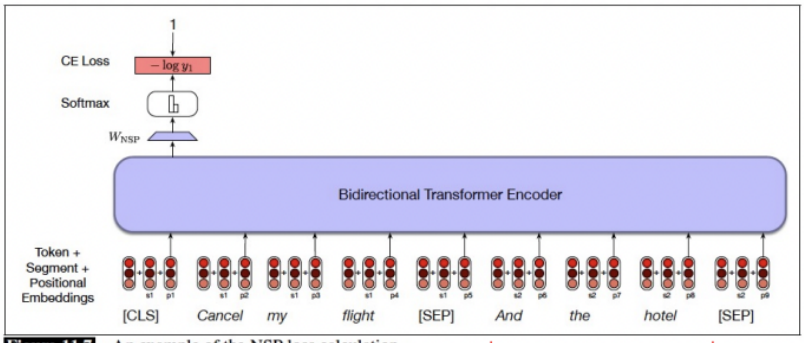


Figure 11.7 An example of the NSP loss calculation.

Use Beam Search to find most likely sentence to follow.

d. Bert can be used for sequence labelling if all of the outputs are used:

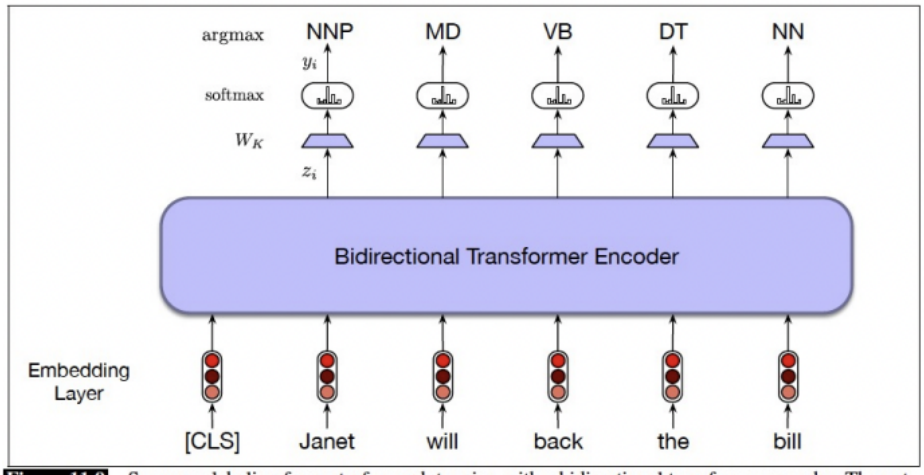


Figure 11.9 Sequence labeling for part-of-speech tagging with a bidirectional transformer encoder. The output vector for each input token is passed to a simple k-way classifier.