In this project, I will be teaching a neural network to translate from English to German.

The Transformer was first introduced in Google's 2017 paper "Attention Is All You Need," in which a new method called the "self-attention mechanism" was proposed. Currently, transformers are used across a wide range of NLP tasks, including language modeling, machine translation, and text generation. A transformer consists of a stack of encoders and decoders to the system, six for each based on the paper, an input of any arbitrary length and some other stack of decoders to output the generated sentence.

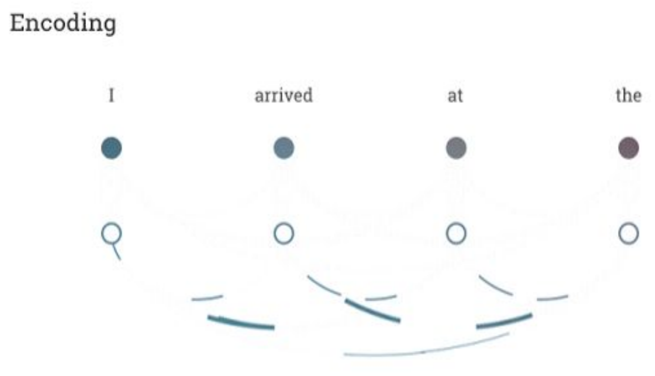


Figure One: Encoding has all of the input words in parallel aggregate to process the sentence to generate a representation for it. Unlike an RNN/LSTM, a transformer can process multiple words in parallel. RNN/LSTM can only process one word at a time through a hidden context vector.

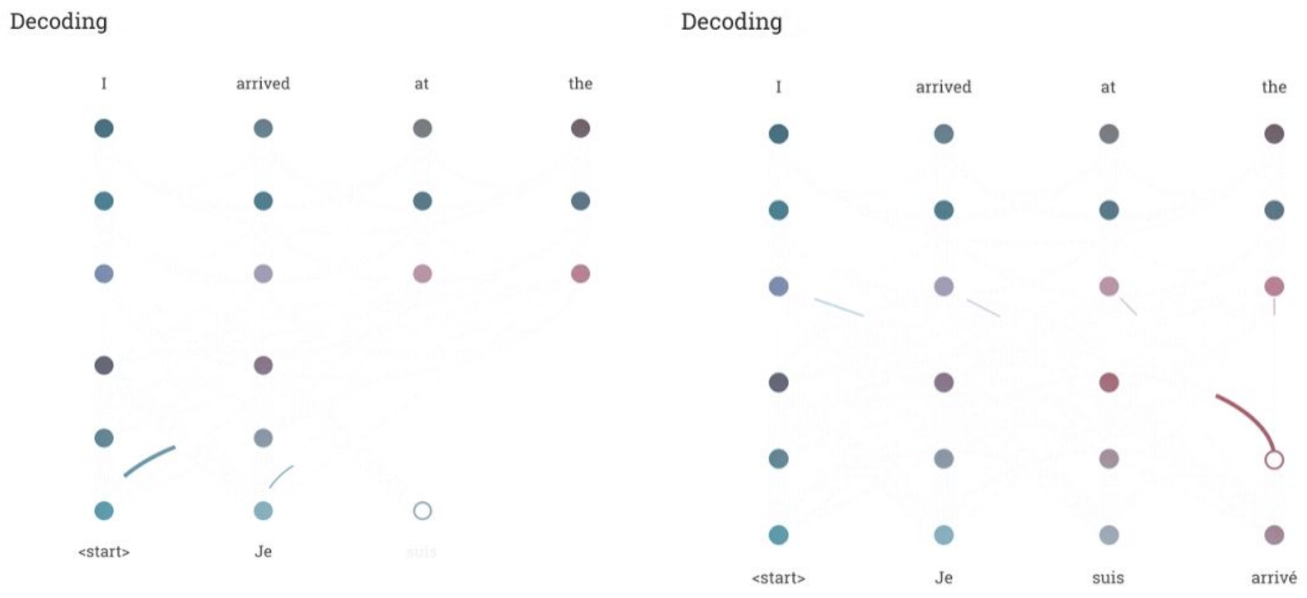


Figure Two: Decoding has two steps: (1) Left image aggregates current target word "suis" from the previous targets ("<start>" "Je") in parallel. (2) Right image aggregates current targeted word with all of the input words from the encoder in parallel. After both steps, we have a prediction for the next target word. The main difference between encoder and decoder, the decoder can only process the current target word through previous time steps whereas an encoder looks at all the words in parallel. The decoder does parallel computation, but only with previous target words that have already been generated; alongside all the input words with representations.

The encoder procedures the input sentence and generates an illustration for it. The decoder makes use of this illustration to create an output with the aid of input words and previously generated (target) words. The initial representation/embedding for every phrase are represented with the aid of the unfilled circles. The algorithm then aggregates facts from all other phrases the usage of self-attention to generate a new illustration per word, represented by the filled balls, knowledgeable via the complete context. This step is then repeated more than one times in parallel for all words, successively generating new representations. Likewise, the decoder generates one word at a time to the end of the sentence. It attends no longer only to the other earlier created words but also to the ultimate representations developed through the encoder.

In distinction to LSTMs, a transformer solely performs a small, regular wide variety of steps whilst making use of a self-attention mechanism which directly fashions relationships between all phrases in a sentence, **regardless** of their respective position. In other words, transformers can use representations of all words in context without compressing all information into a single representation of a fixed length. As a model looks at every phrase in an input sequence, self-attention lets in the model to seem at other applicable components of the entry sequence for higher encoding of the word. It makes use of more than one interest heads which expands the model’s capability to the center of attention on extraordinary positions regardless of their distance in the sequence.

Prelims

Make sure you install the libraries required for this chapter. All of the code will be available on my github.

```
# !pip install http://download.pytorch.org/whl/cu80/torch-0.3.0.post4-cp36-cp36m-linux_x86_64.wh
l numpy matplotlib spacy torchtext seaborn
```

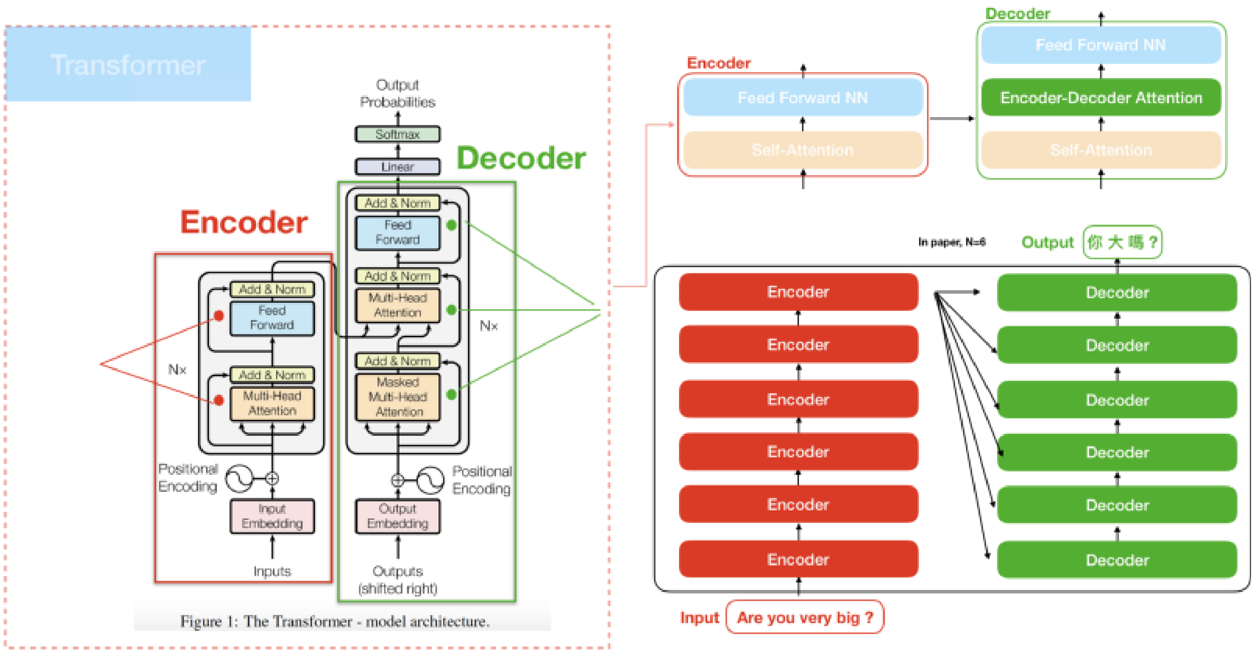
```
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
import math, copy, time
from torch.autograd import Variable
import matplotlib.pyplot as plt
import seaborn
seaborn.set_context(context="talk")
%matplotlib inline
```

The Transformer model

It's important to look at the transformer as a black box before dissecting the intuition. We first type in English letters, later inference german letters down below

INPUT
Je suis étudiant
THE TRANSFORMER
OUTPUT
I am a student

Just like Optimus Prime, we've witnessed an AI transform information but with words instead of cars. The Transformer follows this normal structure using stacked self-attention and point-wise, fully linked layers for both the encoder and decoder



Above we have an encoder (left) and a decoder (right) alongside a generator (linear & softmax) to output the probability of a given input. Before addressing all three modules, I'd thought it is fitting to discuss the attention model architecture.

In previous machine neural translators, we typically have two modules: one encoding higher representation of a given input and decode them onto the translated strings. Now we have a stack of encoders & decoders, 6 stacked each, to enable more attention between the positions and patterns between words. Since both encoder and decoder have self-attention modules, it will be my first discussion.

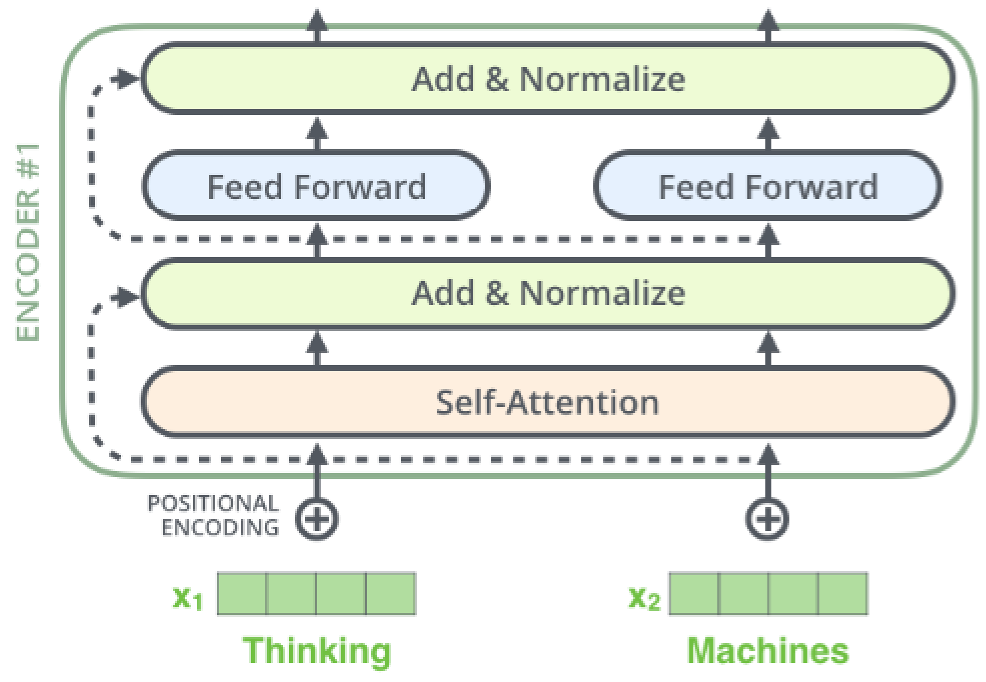
```python
class EncoderDecoder(nn.Module):
    """
    A standard Encoder-Decoder architecture. Base for this and many
    other models.
    """
    def __init__(self, encoder, decoder, src_embed, tgt_embed, generator):
        super(EncoderDecoder, self).__init__()
        self.encoder = encoder
        self.decoder = decoder
        self.src_embed = src_embed
        self.tgt_embed = tgt_embed
        self.generator = generator

    def forward(self, src, tgt, src_mask, tgt_mask):
        "Take in and process masked src and target sequences."
        return self.decode(self.encode(src, src_mask), src_mask,
                            tgt, tgt_mask)

    def encode(self, src, src_mask):
        return self.encoder(self.src_embed(src), src_mask)

    def decode(self, memory, src_mask, tgt, tgt_mask):
        return self.decoder(self.tgt_embed(tgt), memory, src_mask, tgt_mask)
```

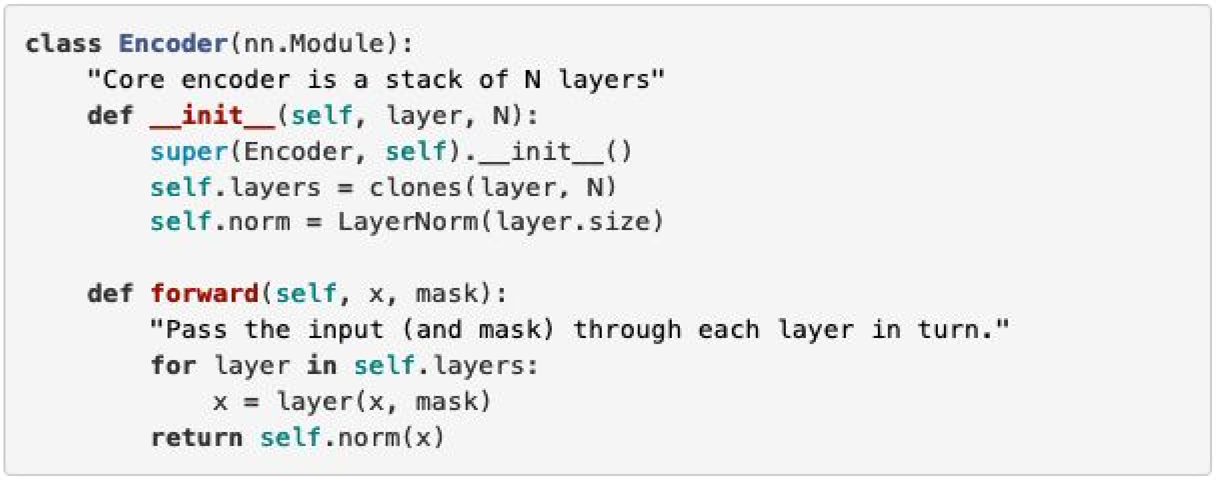
Since both the encoder and decoder both have a stack of six layers, they can be copied using the function clones. In the next two sections you still see both the encoder and decoder class calling the clones function.

```python
def clones(module, N):
    "Produce N identical layers."
    return nn.ModuleList([copy.deepcopy(module) for _ in range(N)])
```

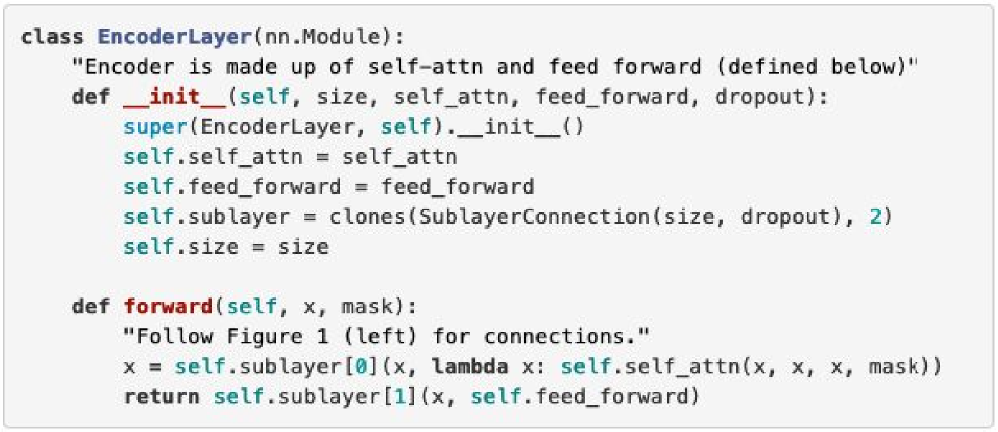
Encoder

The encoder is one of the two modules from the transformer. From the research paper, we're given six each and generate each one using the clone function. Each encoder has two sublayers: multi-headed self-attention and Feed Forward neural net. Its goal is to process input words with deeper representations to later send its context to the decoder. The module "Add & Normalize," then explained, is to help scale the data to train faster with residual connections (dotted arrows).





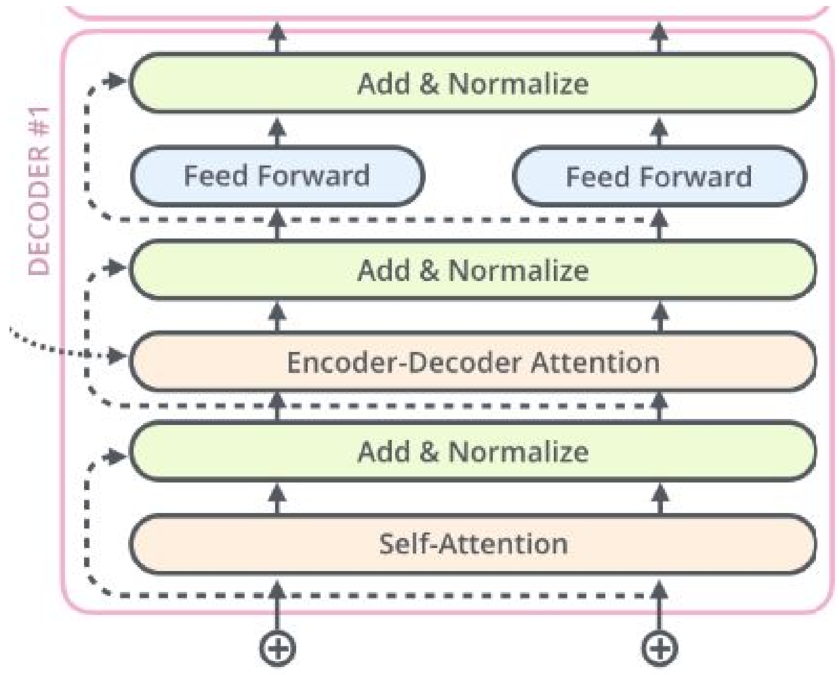
```
class Encoder(nn.Module):
    "Core encoder is a stack of N layers"
    def __init__(self, layer, N):
        super(Encoder, self).__init__()
        self.layers = clones(layer, N)
        self.norm = LayerNorm(layer.size)

    def forward(self, x, mask):
        "Pass the input (and mask) through each layer in turn."
        for layer in self.layers:
            x = layer(x, mask)
        return self.norm(x)
```



```
class EncoderLayer(nn.Module):
    "Encoder is made up of self-attn and feed forward (defined below)"
    def __init__(self, size, self_attn, feed_forward, dropout):
        super(EncoderLayer, self).__init__()
        self.self_attn = self_attn
        self.feed_forward = feed_forward
        self.sublayer = clones(SublayerConnection(size, dropout), 2)
        self.size = size

    def forward(self, x, mask):
        "Follow Figure 1 (left) for connections."
        x = self.sublayer[0](x, lambda x: self.self_attn(x, x, x, mask))
        return self.sublayer[1](x, self.feed_forward)
```

Decoder

The decoder also has six stacks of layers within the transformer; however, there are three sublayers: self-attention, encoder-decoder attention, and a feedforward neural network. The module also has a residual connection to normalize our attentional/context vectors.



```python
class Decoder(nn.Module):
    "Generic N layer decoder with masking."
    def __init__(self, layer, N):
        super(Decoder, self).__init__()
        self.layers = clones(layer, N)
        self.norm = LayerNorm(layer.size)

    def forward(self, x, memory, src_mask, tgt_mask):
        for layer in self.layers:
            x = layer(x, memory, src_mask, tgt_mask)
        return self.norm(x)
```

```python
class DecoderLayer(nn.Module):
    "Decoder is made of self-attn, src-attn, and feed forward (defined bel
ow)"
    def __init__(self, size, self_attn, src_attn, feed_forward, dropout):
        super(DecoderLayer, self).__init__()
        self.size = size
        self.self_attn = self_attn
        self.src_attn = src_attn
        self.feed_forward = feed_forward
        self.sublayer = clones(SublayerConnection(size, dropout), 3)

    def forward(self, x, memory, src_mask, tgt_mask):
        "Follow Figure 1 (right) for connections."
        m = memory
        x = self.sublayer[0](x, lambda x: self.self_attn(x, x, x, tgt_mask
))
        x = self.sublayer[1](x, lambda x: self.src_attn(x, m, m, src_mask
))
        return self.sublayer[2](x, self.feed_forward)
```

Subsequent Mask

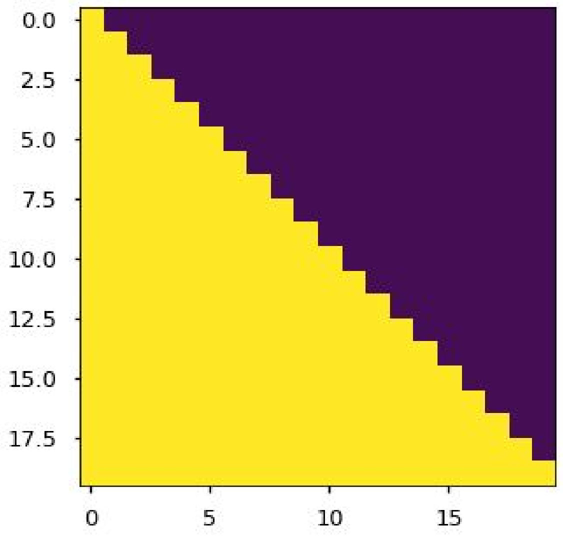
The first self-attention layer inside the decoder can only pay attention to earlier positions in the output sequence: no future content cannot be processed until we generate our current word. subsequent mask type of function, enabling us to hear

```python
def subsequent_mask(size):
    "Mask out subsequent positions."
    attn_shape = (1, size, size)
    subsequent_mask = np.triu(np.ones(attn_shape), k=1).astype('uint8')
    return torch.from_numpy(subsequent_mask) == 0
```

We need to mask future positions using the subsequent_mask function (set to negative infinity) before the softmax classification.

```
plt.figure(figsize=(5,5))
plt.imshow(subsequent_mask(20)[0])
None
```

Below is a graph of attention mask for each word that is allowed to process. The first row @(t=0), there is only one yellow activated signal while the other columns are masked. Once the first word generates, we proceed to the next word which explains why we have these stairs of yellow signals increasing.



Residual & Layer Normalization

Each sub-layer (self-attention, feed-forward networks), for both encoder and decoder, have a residual connection to a layer normalization. If you don’t know what residuals are, they were first introduced by Microsoft for computer vision algorithm to have the option for gradients to skip layers, enabling you to add a large number of layers in a neural network to prevent vanishing gradient. Now they are introduced in NLP to increase the number of stack encoder/decoder for our transformer.

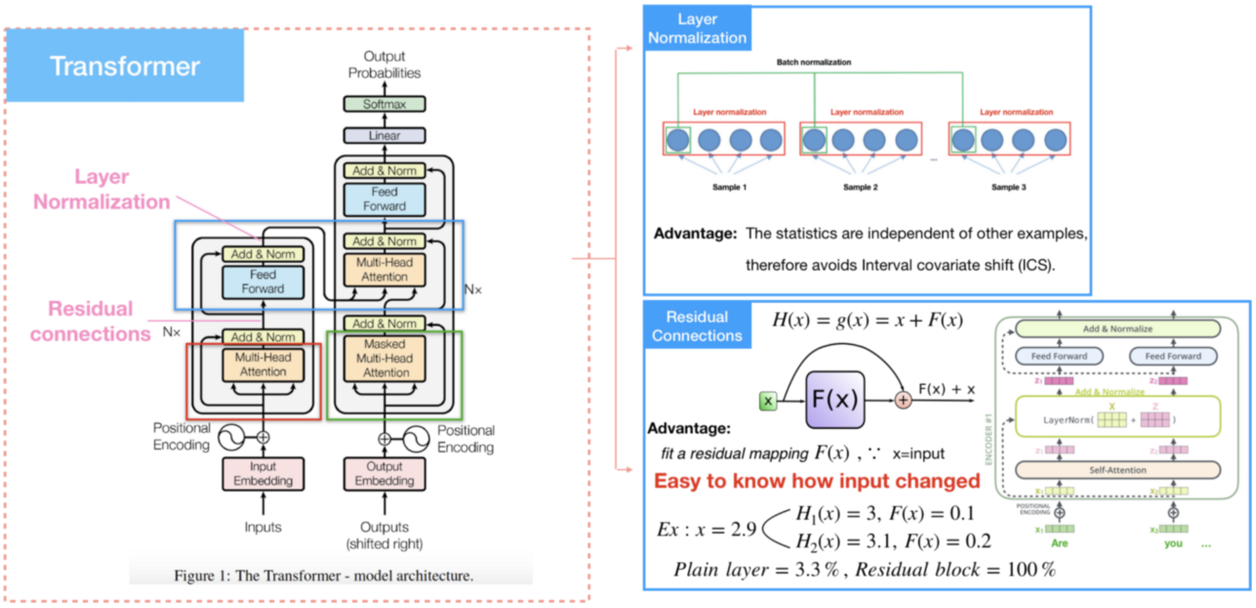
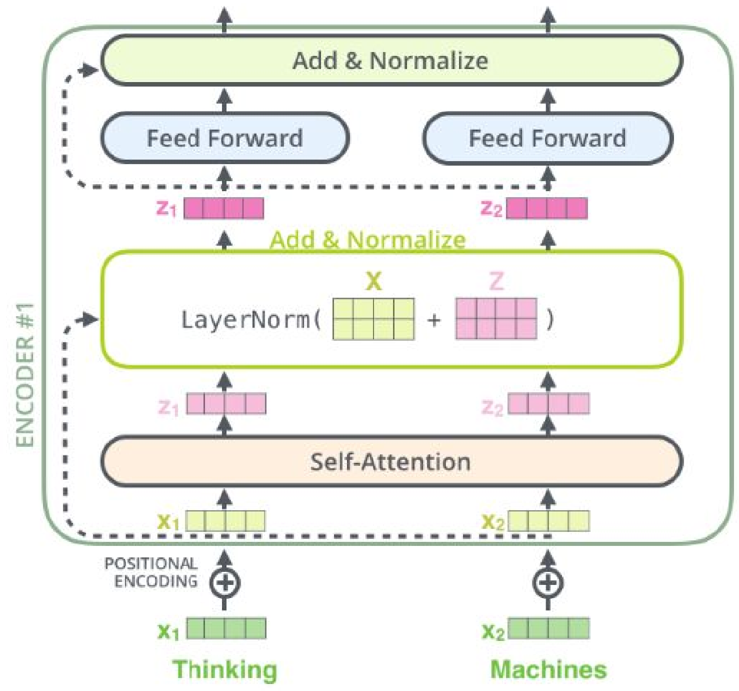


Figure 1: The Transformer - model architecture.

A residual connection is used to form an identity/residual mapping, skip relationships in between sublayers to prevent gradients ceasing through optimization. More info on vanishing gradient is under the chapter “*Machine learning and Deep Learning Fundamentals*.”

```python
class SublayerConnection(nn.Module):
    """
    A residual connection followed by a layer norm.
    Note for code simplicity the norm is first as opposed to last.
    """
    def __init__(self, size, dropout):
        super(SublayerConnection, self).__init__()
        self.norm = LayerNorm(size)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x, sublayer):
        "Apply residual connection to any sublayer with the same size."
        return x + self.dropout(sublayer(self.norm(x)))
```



Above we see a pipeline of words processed from the encoder/decoder. For a quick example, we have two words "Thinking" and "Machines" from the encoder, sent to the same self-attention module where they undergo multi-headed self-attention operations to output (aggregate) their respected context/attention vector. Later normalized with the operation:

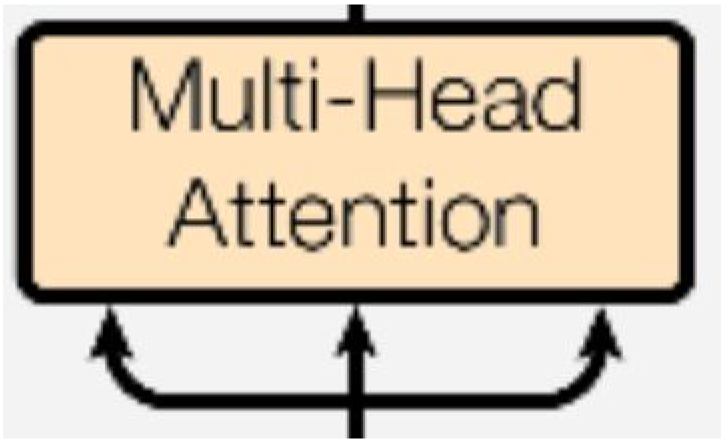
$$LayerNorm(X + Z_i)$$

The (i_{th}) index above are positions for each words either from the input, target, or a combination of both depending on the sub-layer.

```
class LayerNorm(nn.Module):
    "Construct a layernorm module (See citation for details)."
    def __init__(self, features, eps=1e-6):
        super(LayerNorm, self).__init__()
        self.a_2 = nn.Parameter(torch.ones(features))
        self.b_2 = nn.Parameter(torch.zeros(features))
        self.eps = eps

    def forward(self, x):
        mean = x.mean(-1, keepdim=True)
        std = x.std(-1, keepdim=True)
        return self.a_2 * (x - mean) / (std + self.eps) + self.b_2
```

Query, Key, Value

Before talking about attention modules, I need to discuss how they were initially used with RNN & LSTM for NLP. Initially, we had an input of words <x_1, x_2, …, x_n> encoding a representation of hidden vectors <h_1, h_2, …, h_n>. We would have a context vector c_i which is the summation of hidden states, weighted by **attention score** α. Using the dot product between the attention score and hidden state, we get the output sequence decoded <y_1, y_2, …, y_n>. The drawback with this design lacked parallel computation. For (Masked) Multi-Head Attention this was all resolved.



We have **three** attention modules: one inside encoder and the other two inside decoder, one of them called "masked". First I need to discuss what attention modules are from a mathematical standpoint (query, key, value) and explain the location for each one within the architecture. Each module serves a unique purpose.
There are three inputs for (Masked) Multi-Head Attentions: key (address), value (element), and Query (current word). They're used to approximate what words we need to pay the most attention to represent. Embedding each word from the input or target creates the three attention vectors. Each sentence fed into the module will have multiple attention scores, one for each word, representing all information instead of just remembering a single hidden fixed-length vector like RNN & LSTM. There will be times the model pays attention to input words (encoder), target words (decoder), and a combination of both (decoder): depending on which module located on the architecture. All cases perform parallel computation.

To acquire a context/attention vector for the current word we are observing, output vector of attention, we need to find the score. The higher the score, the more important the current word is for the encoder/decoder. Calculating context requires three steps.

We **first** need to find the similarity between the word we are current observing (query), against multiple words (keys) that are being compared. This is known as the similarity dot product score:

$$Similarity(Query, Key_i) = Query * Key_i$$

The (i_{th}) index above represents an array of key words being compared against the current word we are observing (query).

The **second** step is to find the attention score by computing the similarity inside a softmax function, the probability distribution between 0 and 1, to determine which key is most relatable to the query. We need to make sure the similarity dot product doesn't grow large in magnitude, this will push the softmax function to lower the gradients (vanishing gradient). Google thought the best solution is to scale the similarity dot product is diving the equation by $\sqrt{d_k}$. d_k is the square root of the key dimension. Now we have the following formula to find attention score:

$$\text{Attention Score} : \alpha_i$$

$$\alpha_i = Softmax(\frac{Similarity(Query, Key_i)}{\sqrt{d_k}})$$

$$\alpha_i = Softmax(\frac{QK_i^T}{\sqrt{d_k}})$$

To understand why the dot product can potentially get large between query and key, assume their independent random variables with mean 0 and variance 1. The equation below indicated the dot product the mean 0 with a variance of d_k:

$$q \cdot k = \sum_{i=1}^{d_k} q_i k_i$$

Last step is to use the attention score α, also known as the weights, in conjunction with the corresponding (i_{th}) index to value (hidden state to a word) to find the **attention/context vector**.

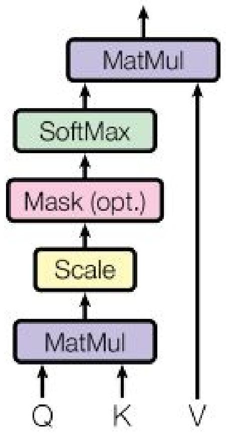
$$\text{Hidden State: Value}_i \leftrightarrow (h)$$

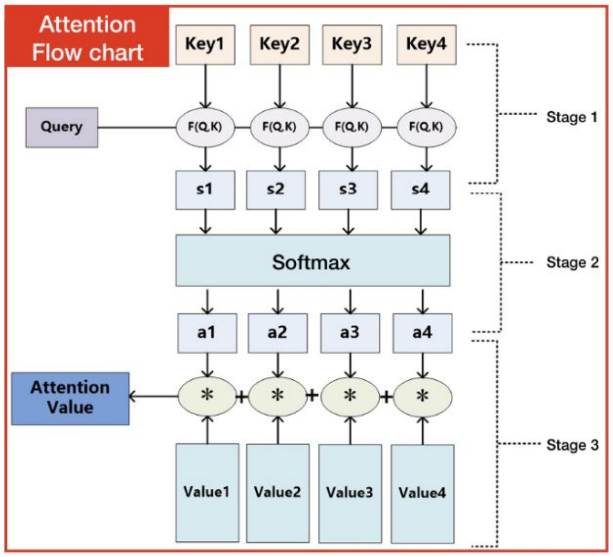
$$\text{Attention Score: } \sum_{i=1}^{L_x} \alpha_i$$

$$\text{Attention/Context Vector: } c_i = \sum_{j=1}^{T_x} \alpha_{ij} h_j \leftrightarrow Attention(Query, Source)$$

Below is the final **attention/context** vector after observing one word in the query: also known as the output of a self-attention layer.

$$Attention(Query, Source) = \sum_{i=1}^{L_x} \alpha_i \cdot \text{Value}_i$$



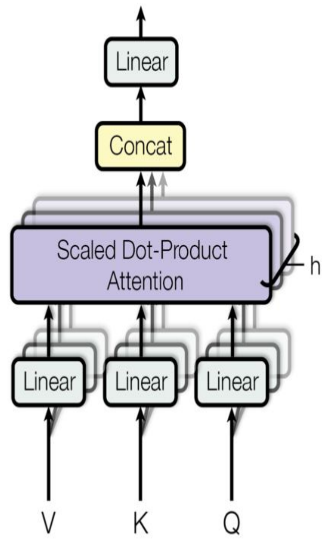


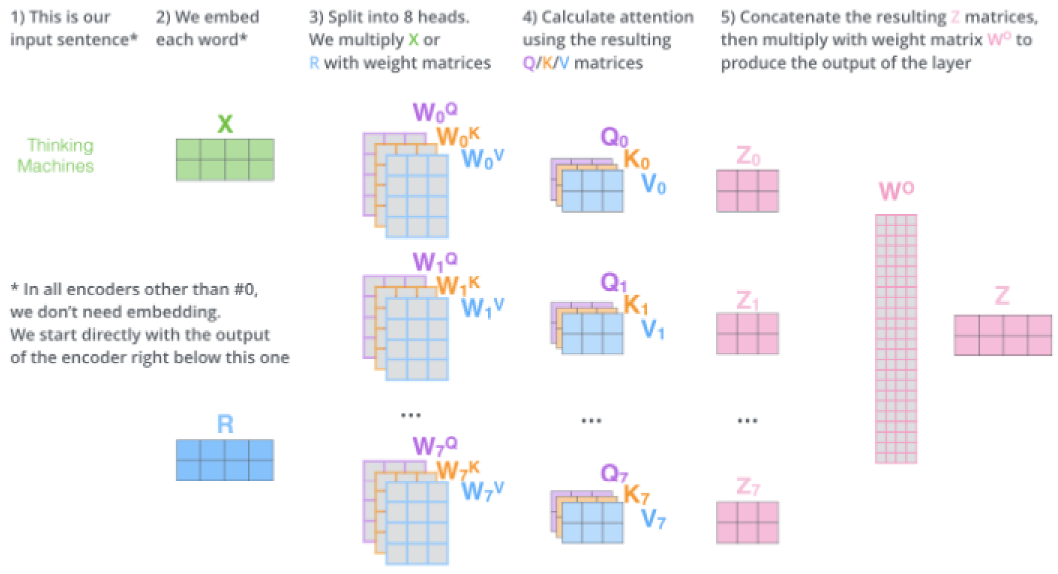
Above is a flow diagram how the algorithm creates a attention/context vector. If you remember how the key and value have an (i_{th}) index, The images above show they represent an element within arrays.

```python
def attention(query, key, value, mask=None, dropout=None):
    "Compute 'Scaled Dot Product Attention'"
    d_k = query.size(-1)
    scores = torch.matmul(query, key.transpose(-2, -1)) \
             / math.sqrt(d_k)
    if mask is not None:
        scores = scores.masked_fill(mask == 0, -1e9)
    p_attn = F.softmax(scores, dim = -1)
    if dropout is not None:
        p_attn = dropout(p_attn)
    return torch.matmul(p_attn, value), p_attn
```

Multi-Headed Attention

With self-attention alone, finding relationships between words won't aggregate computation in parallel unless we project multi-headed attention. In Google's research paper, they used an "eight-headed" self-attention function. In the previous section, I've only demonstrated "one-headed" attention model: one key, one value, and one query to compute each word. The benefit having multiple heads aggregating each word not only performs parallel computation compared to an RNN & LSTM but can find an extensive pattern in a sentence.



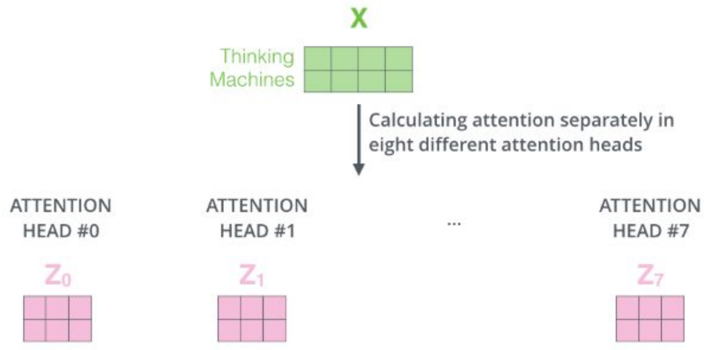

1) This is our input sentence*
2) We embed each word*
3) Split into 8 heads. We multiply X or R with weight matrices
4) Calculate attention using the resulting Q/K/V matrices
5) Concatenate the resulting Z matrices, then multiply with weight matrix W^O to produce the output of the layer
Thinking Machines
X
R
* In all encoders other than #0, we don't need embedding. We start directly with the output of the encoder right below this one
W0Q W0K W0V
W1Q W1K W1V
W7Q W7K W7V
Q0 K0 V0
Q1 K1 V1
Q7 K7 V7
Z0
Z1
Z7
WO
Z

We keep separate $Q/K/V$ weight matrices for each head with multi-headed attention resulting in different $Q/K/V$ matrices. As we have done before, we multiply the input (X / R), coming from the encoder or decoder, by matrices of $W^Q/W^K/W^V$ to produce matrices of $Q/K/V$. Using the same self-attention formula from the previous section:

$$Attention(Query, Source) = \sum_{i=1}^{L_x} \alpha_i \cdot \text{Value}_i$$

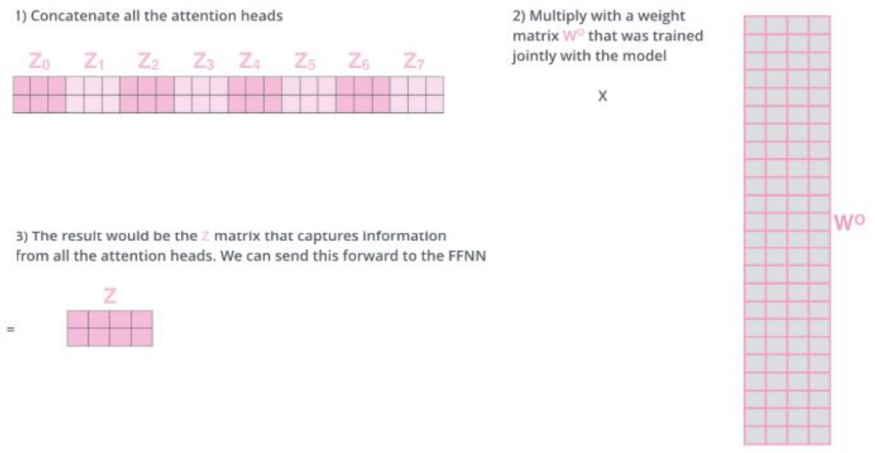
We now repeat this process eight times with head (i) = 8:

$$head_i = Attention(QW_i^Q, KW_i^k, VW_i^V)$$



The feed forward neural net, the next sub-layer after multi-head attention, can only receive one matrix (a vector row for each word) while we have eight from the attention head. The best solution is to concatenate all of the attention heads, increasing the column dimension, and perform matrix multiplication with an initialized weight matrix W^O, to shorten the attention head column dimension to the size of one word for each row.

$$MultiHead(Q, K, V) = Concat(head_1, ..., head_h)W^O$$


1) Concatenate all the attention heads
Z0 Z1 Z2 Z3 Z4 Z5 Z6 Z7
2) Multiply with a weight matrix W^O that was trained jointly with the model
x
W^O
3) The result would be the Z matrix that captures information from all the attention heads. We can send this forward to the FFNN
=
Z

Where the projections are parameter matrices

$W_i^Q \epsilon \mathbb{R}^{d_{model} \times d_q}$, $W_i^K \epsilon \mathbb{R}^{d_{model} \times d_k}$, $W_i^V \epsilon \mathbb{R}^{d_{model} \times d_v}$ and $W_i^O \epsilon \mathbb{R}^{d_{model} \times hd_o}$

We use $h = 8$ parallel layers of attention, or heads, in this work. We use $dk = dv = d_{model}/h = 64$ for each one of these. The performance for this algorithm is equal to one-headed attention since we cancel out the concatenation of eight heads.

```python
class MultiHeadedAttention(nn.Module):
    def __init__(self, h, d_model, dropout=0.1):
        "Take in model size and number of heads."
        super(MultiHeadedAttention, self).__init__()
        assert d_model % h == 0
        # We assume d_v always equals d_k
        self.d_k = d_model // h
        self.h = h
        self.linears = clones(nn.Linear(d_model, d_model), 4)
        self.attn = None
        self.dropout = nn.Dropout(p=dropout)

    def forward(self, query, key, value, mask=None):
        "Implements Figure 2"
        if mask is not None:
            # Same mask applied to all h heads.
            mask = mask.unsqueeze(1)
        nbatches = query.size(0)

        # 1) Do all the linear projections in batch from d_model => h x d_k
        query, key, value = \
            [l(x).view(nbatches, -1, self.h, self.d_k).transpose(1, 2)
             for l, x in zip(self.linears, (query, key, value))]

        # 2) Apply attention on all the projected vectors in batch.
        x, self.attn = attention(query, key, value, mask=mask,
                                 dropout=self.dropout)

        # 3) "Concat" using a view and apply a final linear.
        x = x.transpose(1, 2).contiguous() \
             .view(nbatches, -1, self.h * self.d_k)
        return self.linears[-1](x)
```

Three Attentions: (Masked) Multi-Head Attention

Now we’re going explain the location for each attention module within the transformer: one is inside the encoder, two more in the decoder. If you remember the three images in the beginning of this chapter, this is the transformation process if you haven’t figured it out by now. Google’s AI website has the images done in real-time animation so I recommend checking out their blog post before reading any further.

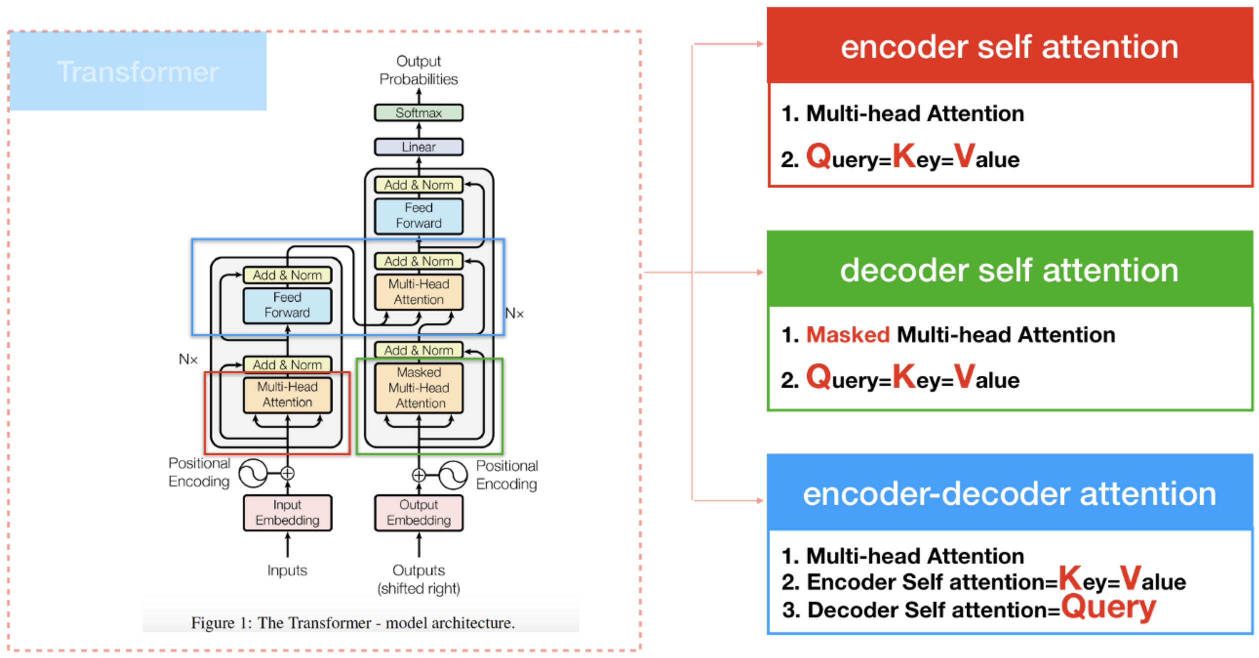


Figure 1: The Transformer - model architecture.

encoder self attention

1. **Multi-head Attention**
2. **Query=Key=Value**

decoder self attention

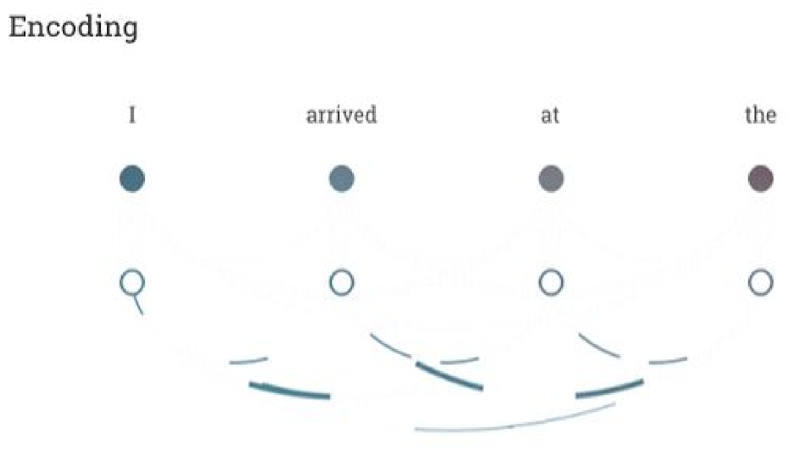
1. **Masked Multi-head Attention**
2. **Query=Key=Value**

encoder-decoder attention

1. **Multi-head Attention**
2. **Encoder Self attention=Key=Value**
3. **Decoder Self attention=Query**

The transformer has multi-headed attention models used in three unique ways:

1) **Encoder Self Attention:**
The keys, values, and queries will all be processed from the input words, thergo inside the encoder. This is the first out of the three transformers to process words. Every word at the end will have representation to send meaning into the decoder.



Above is a screenshot of the encoder transformation (currently aggregating 2_{nd} stack). We see unfilled circles that are the initial representation for each word. Using multi-head self-attention, we can aggregate words in parallel which explains multiples signals (lines) jumping between each unfilled circle. The attention model later fill the circles by generating new representation per word to repeat the process for the next encoder stack. All the keys, values, and query are all from the input data: non from the target word.

If we had the following sentence fed into the self-attention model, assume the algorithm is not trained.

“The animal didn’t cross the street because it was too tired.”

The word “it” at position #8 is a pronoun to refer to that one previously mentioned. It’s common sense for us humans to know the word “it” is an animal at position #2 but how does the computer know?

The **First** step is to have each word embedded with three vectors (key/value/query) to helps us approximate an attention score: determining how much focus to place on other parts of the input sentence. We are still ignoring the decoder until we're done processing all 6 stacks for the encoder.

The **second** step to aggregate input words with each other. To continue the consistency for the reader, if we have the word “it” as the query, we would have the following dot product similarities:

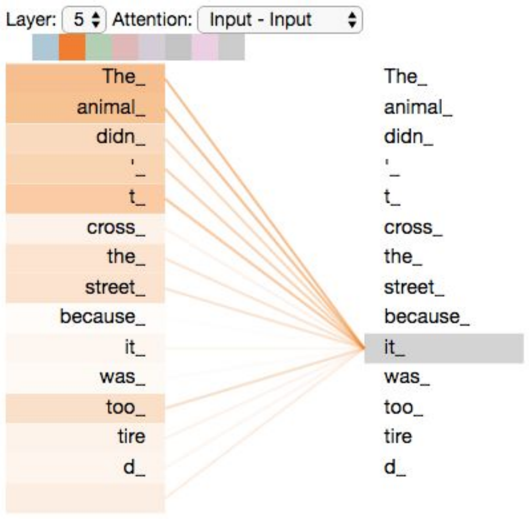
$Similarity(Query_8, Key_1) \leftrightarrow Q_8 * K_1$: (“it” vs “The”)
$Similarity(Query_8, Key_2) \leftrightarrow Q_8 * K_2$: (“it” vs “animal”)
$Similarity(Query_8, Key_3) \leftrightarrow Q_8 * K_3$: (“it” vs “didn’t”)
$Similarity(Query_8, Key_4) \leftrightarrow Q_8 * K_4$: (“it” vs “cross”)
$Similarity(Query_8, Key_5) \leftrightarrow Q_8 * K_5$: (“it” vs “the”)
$Similarity(Query_8, Key_6) \leftrightarrow Q_8 * K_6$: (“it” vs “street”)
$Similarity(Query_8, Key_7) \leftrightarrow Q_8 * K_7$: (“it” vs “because”)
$Similarity(Query_8, Key_8) \leftrightarrow Q_8 * K_8$: (“it” vs “it”)
$Similarity(Query_8, Key_9) \leftrightarrow Q_8 * K_9$: (“it” vs “was”)
$Similarity(Query_8, Key_{10}) \leftrightarrow Q_8 * K_{10}$: (“it” vs “too”)
$Similarity(Query_8, Key_{11}) \leftrightarrow Q_8 * K_{11}$: (“it” vs “tired”)

With the image to the right, we see a high correlation with the word “The” and “animal” to “it.” Based on the given English corpus database, there's a lot of sentences with the word “it” correlated with the two: resulting in massive distribution. Sometimes context between languages requires more than one word to understand the meaning. In this case “The” and “animal” is the example given in the sentence.

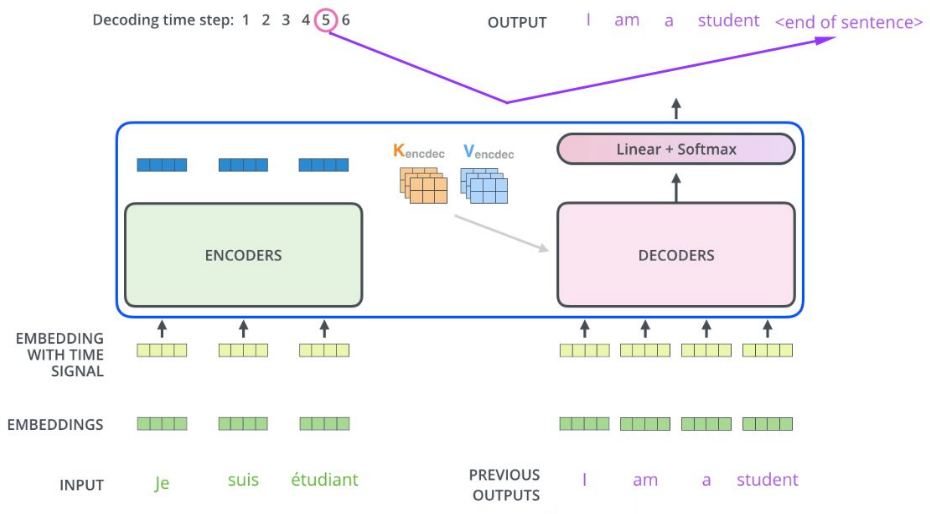
We still need to scale down the dot product similarities to prevent vanishing gradient, the **third** step is to scale the similarity score by diving by $\sqrt{d_k}$. Later apply the softmax function to get your attention score. Look at the formulas give from self-attention.

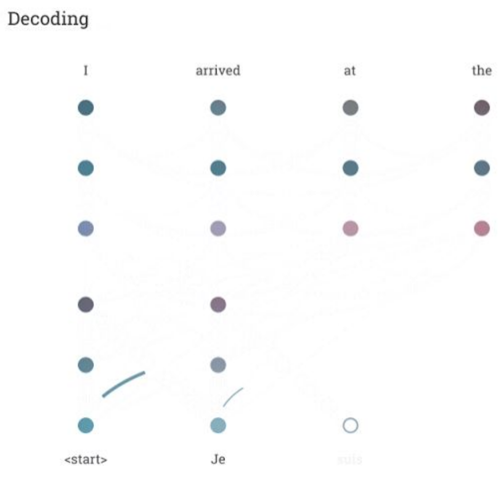
The last step is to multiply the attention score with each value vector corresponding to their respected key, later sum up the weighted value vectors.

$$Attention(Query, Source) = \sum_{i=1}^{L_x} \alpha_i \cdot \text{Value}_i$$



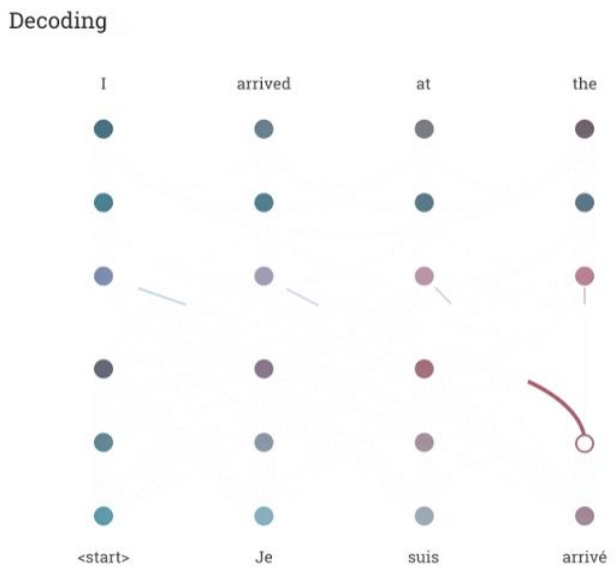
2) **Decoder Masked Multi Self Attention:**
While the encoder self-attention can pay attention to the entire input sequence in parallel, decoder self-attention can do parallel computation as well but only aggregate earlier positions in the output sequence. So we need to mask future positions (initialize to -inf) in the similarity dot product attention



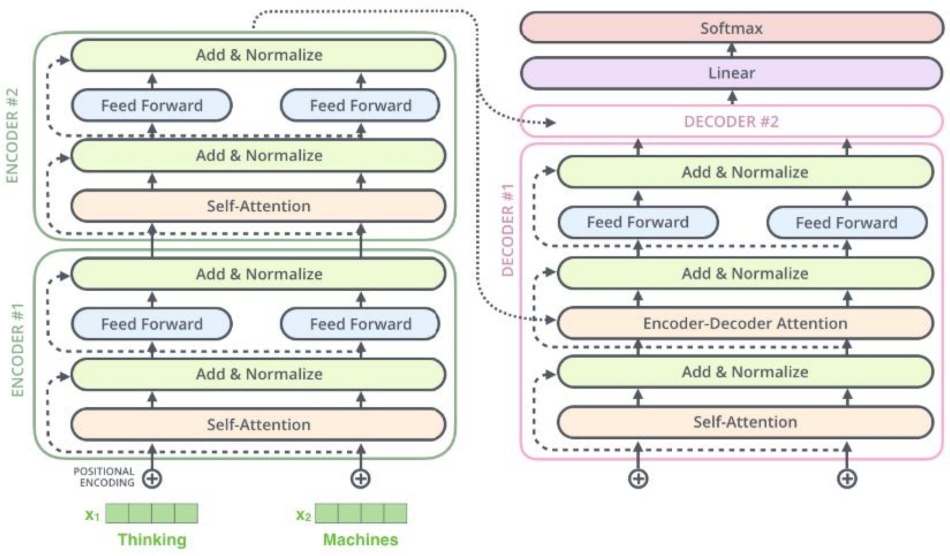


If you look at both images, we have the output sequence gathering representations from the previous targets in order to make a prediction on what word to generate next.

3) **Encoder-Decoder Self-Attention**
The output for both the encoder and decoder self-attention happen to be the input for the encoder-decoder self-attention module: the last attention in the transformer. This is where all the words representative from the input encoder, and the current and previous words that have been generated in the decoder, process information together to generate the output.



Each word at a time is generated from the decoder represents a time step.



Here we see the last stack of the encoder feeding it's representation into the encoder-decoder attention module. While the decoder is looking previous target words to generate a new one, the encoder send its representation

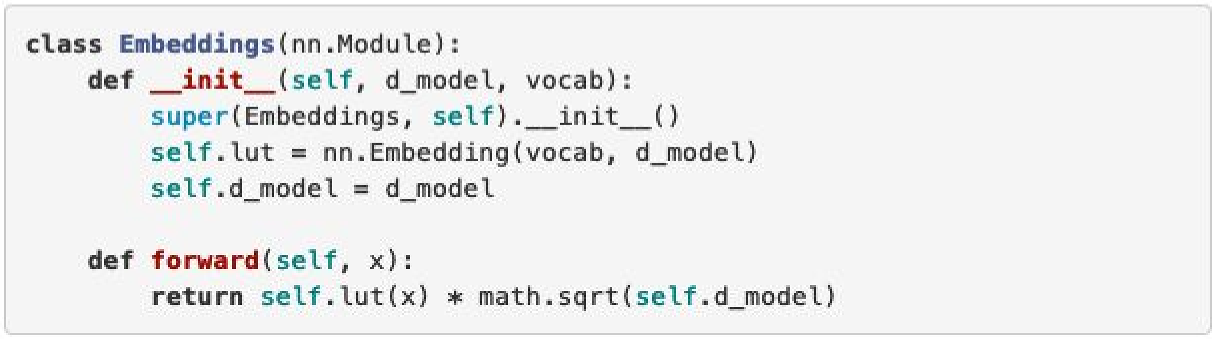
Word Embeddings

Word embeddings are used to convert a set of strings into vectors with consecutive numbers to represent meaningful context between words. Each word embedded is mapped into one vector that are learned like to a neural network.

With the input tokens given from the english-german text corpse, we embed the content into dimensional d_{model} vector.

x_1
x_2
x_3
Je
suis
étudiant

We multiply our weights by $\sqrt{d_{model}}$ in the embedding layers in order to



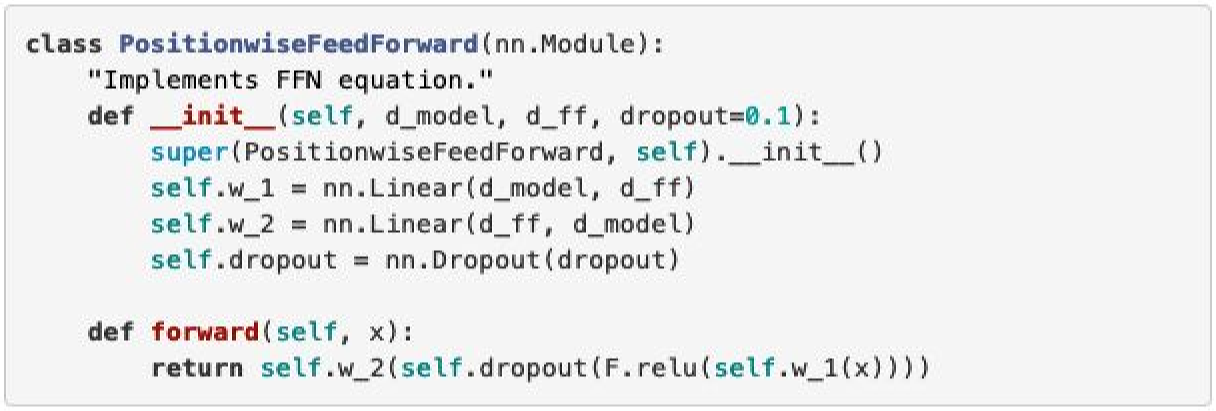
```
class Embeddings(nn.Module):
    def __init__(self, d_model, vocab):
        super(Embeddings, self).__init__()
        self.lut = nn.Embedding(vocab, d_model)
        self.d_model = d_model

    def forward(self, x):
        return self.lut(x) * math.sqrt(self.d_model)
```

Position-wise Feed Forward Neural Net

For both encoder and decoder self-attention sub-layers, lies a feed-forward neural network (ffnn) to approximate each word: output to each position separately and identically. We're given two linear transformations of a relu rectifier from each word processed inside ffnn: separately and identically. The number of ffnn for each sub-layer is dependent on the number of words process from the input sequence.

$$FFN(x) = max(0, xW_1 + b_1)W2 + b2$$

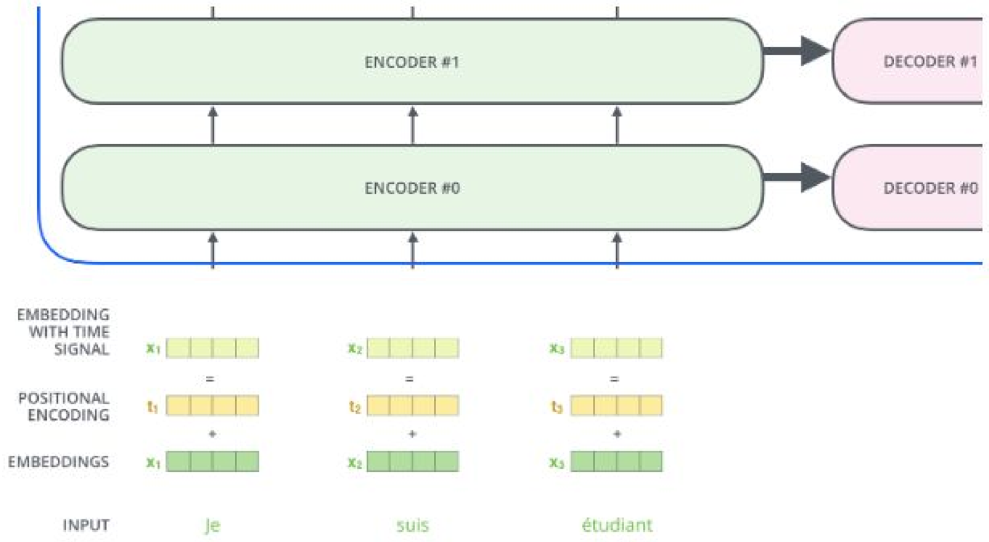
A linear transformation is a dependant for each position fed into the transformer.



```
class PositionwiseFeedForward(nn.Module):
    "Implements FFN equation."
    def __init__(self, d_model, d_ff, dropout=0.1):
        super(PositionwiseFeedForward, self).__init__()
        self.w_1 = nn.Linear(d_model, d_ff)
        self.w_2 = nn.Linear(d_ff, d_model)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x):
        return self.w_2(self.dropout(F.relu(self.w_1(x))))
```

Positional Encoding

While it looks like Multi-headed attention alone looks superior to its predecessor RNN, it cannot make use of the position of the words in the input sequence. If the input sentence "I love NLP" was sent to the transformer, the order of the sequence would be unordered and possibly return a different sequential pattern "Love I NLP." We need to apply some information to the word embeddings to have relative and or absolute positions of the token in the sequence. Thus enter **position encoding**.



To form an embedded vector with a “tracker” for each position, we use trigonometric functions to learn the representation between relative and absolute positions:
$PE_{(pos,2i)} = sin(\frac{pos}{10000^{2i/d_{model}}})$, $PE_{(pos,2i+1)} = cos(\frac{pos}{10000^{2i/d_{model}}})$
Above see the the variable pos representing the position and (i_{th}) is the dimension with a wave of different frequency. That is, a sinusoid corresponds to each dimension of the positional encoding. The wavelengths constitute a geometric progression:

$PE[pos + k]$ is a linear function to $PE[pos]$

With the given functions we can now find the relative positions between different embeddings that can be easily understood for each word.

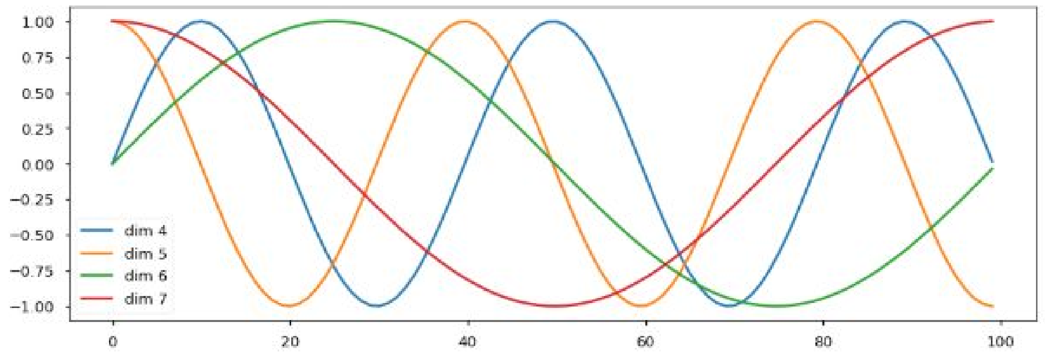
$$\text{If shape of encoder, decoder } = [T, d_{model}] \text{ pos: position of the word}$$
$$\rightarrow \text{then pos } \epsilon[0, T), \text{ i } \epsilon[0, d_{model}) \text{ i: Element i in } d_{model}$$

There are a huge amount of weight parameters inside the transformer to keep track all of the word positions so we add a dropout regularization with $P_{drop} = 0.1$ inside the encoder and decoder.

```
class PositionalEncoding(nn.Module):
    "Implement the PE function."
    def __init__(self, d_model, dropout, max_len=5000):
        super(PositionalEncoding, self).__init__()
        self.dropout = nn.Dropout(p=dropout)

        # Compute the positional encodings once in log space.
        pe = torch.zeros(max_len, d_model)
        position = torch.arange(0, max_len).unsqueeze(1)
        div_term = torch.exp(torch.arange(0, d_model, 2) *
                             -(math.log(10000.0) / d_model))
        pe[:, 0::2] = torch.sin(position * div_term)
        pe[:, 1::2] = torch.cos(position * div_term)
        pe = pe.unsqueeze(0)
        self.register_buffer('pe', pe)

    def forward(self, x):
        x = x + Variable(self.pe[:, :x.size(1)],
                         requires_grad=False)
        return self.dropout(x)
```

Below the positional encoding, a position-based sine wave will be added. The wave's frequency and offset for each dimension is different.

```
plt.figure(figsize=(15, 5))
pe = PositionalEncoding(20, 0)
y = pe.forward(Variable(torch.zeros(1, 100, 20)))
plt.plot(np.arange(100), y[0, :, 4:8].data.numpy())
plt.legend(["dim %d"%p for p in [4,5,6,7]])
None
```



For the example above we have graphs with multiple crossover high signals @time=18, same goes for low signals in the same location @time=65, this means we have words with similar context meaning at both positions.

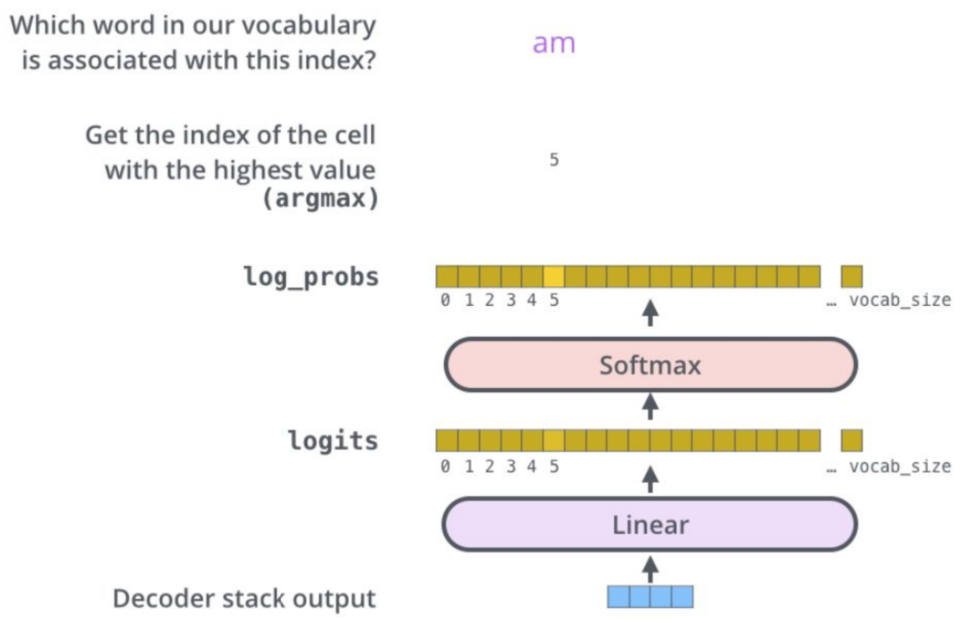
The Final Linear and Softmax Layer (Generator)

After we're done generating content with the last decoder stack, we need to convert out floating vector (word) into a linear module to project our content and perform a softmax approximation to find the highest value closest to one (argmax).

Linear module: a feed forward network projects a new larger vector called logits

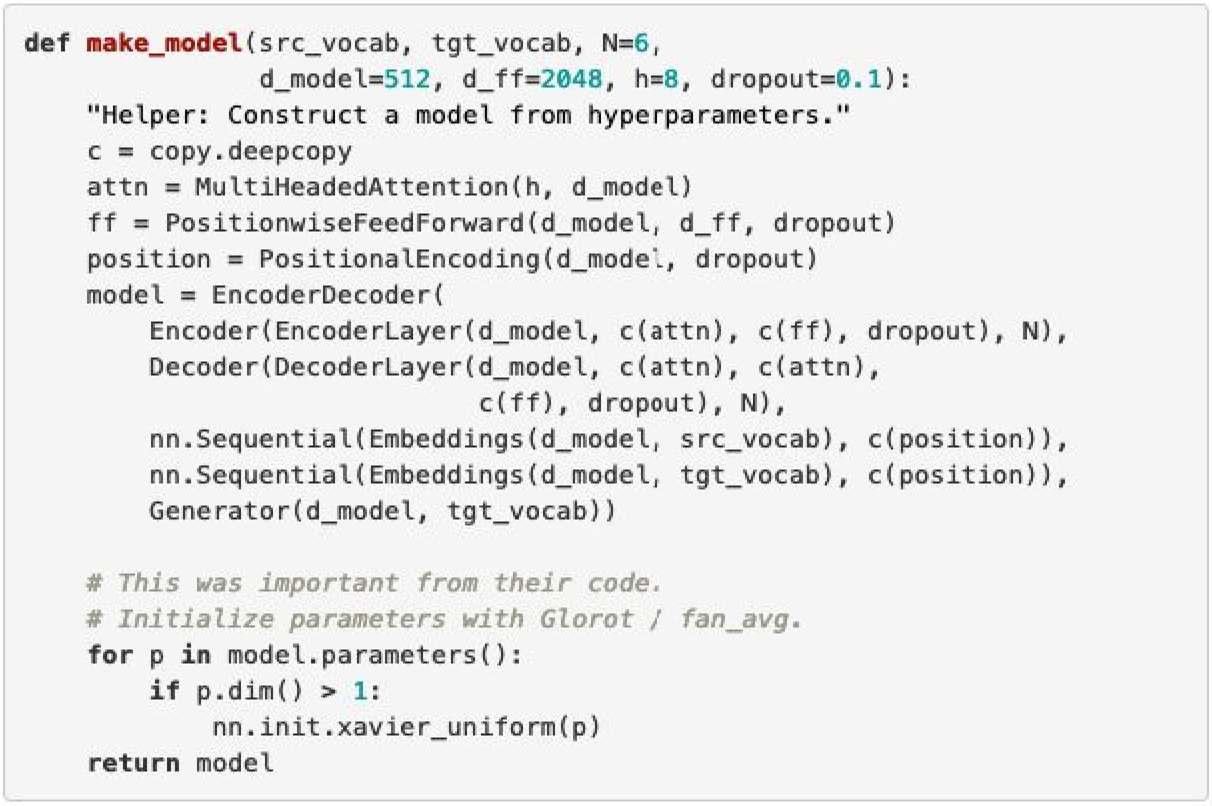
Softmax: approximate the logits vector to find the element with the value closest to one (argmax).

```python
class Generator(nn.Module):
    "Define standard linear + softmax generation step."
    def __init__(self, d_model, vocab):
        super(Generator, self).__init__()
        self.proj = nn.Linear(d_model, vocab)

    def forward(self, x):
        return F.log_softmax(self.proj(x), dim=-1)
```


Which word in our vocabulary is associated with this index?
am
Get the index of the cell with the highest value (argmax)
5
log_probs
0 1 2 3 4 5 … vocab_size
Softmax
logits
0 1 2 3 4 5 … vocab_size
Linear
Decoder stack output

Full Model

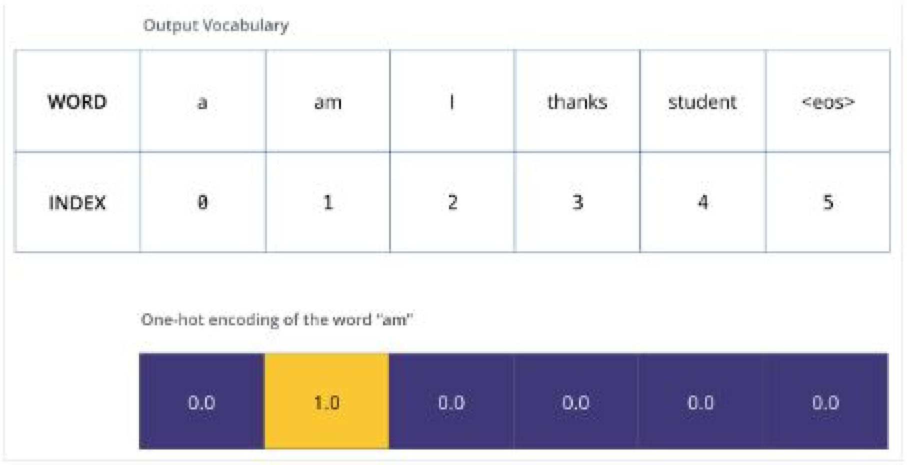
This is the full model of the algorithm.



```python
def make_model(src_vocab, tgt_vocab, N=6,
               d_model=512, d_ff=2048, h=8, dropout=0.1):
    "Helper: Construct a model from hyperparameters."
    c = copy.deepcopy
    attn = MultiHeadedAttention(h, d_model)
    ff = PositionwiseFeedForward(d_model, d_ff, dropout)
    position = PositionalEncoding(d_model, dropout)
    model = EncoderDecoder(
        Encoder(EncoderLayer(d_model, c(attn), c(ff), dropout), N),
        Decoder(DecoderLayer(d_model, c(attn), c(attn),
                             c(ff), dropout), N),
        nn.Sequential(Embeddings(d_model, src_vocab), c(position)),
        nn.Sequential(Embeddings(d_model, tgt_vocab), c(position)),
        Generator(d_model, tgt_vocab))

    # This was important from their code.
    # Initialize parameters with Glorot / fan_avg.
    for p in model.parameters():
        if p.dim() > 1:
            nn.init.xavier_uniform(p)
    return model
```

Training

Now that the architecture is completed to pass in data, we still need to train the parameters within the model to process words to generate content close to previous examples. If given the following sentence, “a am I thanks student <eos>,” we know at first glance the sentence is awkward and unordered.


Output Vocabulary

WORD	a	am	I	thanks	student	<eos>
INDEX	0	1	2	3	4	5

One-hot encoding of the word "am"

0.0	1.0	0.0	0.0	0.0	0.0

One-hot encoding of our output vocabulary

Above we see an untrained probability distribution. Most cases in a trained model we would have each word assigned a value between zero to one, approximating a legitimate distribution. The output vocabulary needs more training.

Batches and Masking

```python
class Batch:
    "Object for holding a batch of data with mask during training."
    def __init__(self, src, trg=None, pad=0):
        self.src = src
        self.src_mask = (src != pad).unsqueeze(-2)
        if trg is not None:
            self.trg = trg[:, :-1]
            self.trg_y = trg[:, 1:]
            self.trg_mask = \
                self.make_std_mask(self.trg, pad)
            self.ntokens = (self.trg_y != pad).data.sum()

    @staticmethod
    def make_std_mask(tgt, pad):
        "Create a mask to hide padding and future words."
        tgt_mask = (tgt != pad).unsqueeze(-2)
        tgt_mask = tgt_mask & Variable(
            subsequent_mask(tgt.size(-1)).type_as(tgt_mask.data))
        return tgt_mask
```

Training Loop

Now we need to keep track of our lost results using a generic loss function, in this case adam optimizer with the learning rate: β1=0.9, β2=0.98 and ϵ=10−e9.

```
def run_epoch(data_iter, model, loss_compute):
    "Standard Training and Logging Function"
    start = time.time()
    total_tokens = 0
    total_loss = 0
    tokens = 0
    for i, batch in enumerate(data_iter):
        out = model.forward(batch.src, batch.trg,
                            batch.src_mask, batch.trg_mask)
        loss = loss_compute(out, batch.trg_y, batch.ntokens)
        total_loss += loss
        total_tokens += batch.ntokens
        tokens += batch.ntokens
        if i % 50 == 1:
            elapsed = time.time() - start
            print("Epoch Step: %d Loss: %f Tokens per Sec: %f" %
                    (i, loss / batch.ntokens, tokens / elapsed))
            start = time.time()
            tokens = 0
    return total_loss / total_tokens
```

Training Data and Batching

The data used for his chapter is trained on the WMT 2004 English-German with 4.5 million of sentence pairs.

```
global max_src_in_batch, max_tgt_in_batch
def batch_size_fn(new, count, sofar):
    "Keep augmenting batch and calculate total number of tokens + padding."
    global max_src_in_batch, max_tgt_in_batch
    if count == 1:
        max_src_in_batch = 0
        max_tgt_in_batch = 0
    max_src_in_batch = max(max_src_in_batch,  len(new.src))
    max_tgt_in_batch = max(max_tgt_in_batch,  len(new.trg) + 2)
    src_elements = count * max_src_in_batch
    tgt_elements = count * max_tgt_in_batch
    return max(src_elements, tgt_elements)
```

Hardware and Schedule

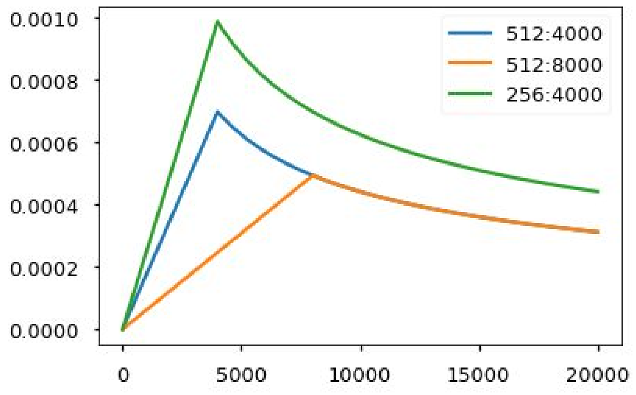
Make sure you have a GPU to run this algorithm. At least for me, I utilized 8 NVIDIA's P100 GPU's. Setting aside the number of training steps (100,000) for 12 hours, each epoch took about 0.4 seconds. You can configure the optimization configurations depending on your specs. The entire training process took about 3.5 days.

Optimizer

An adam optimizer is used with the following settings: $\beta 1=0.9$, $\beta 2=0.98$ and $\epsilon=10-9$, with the following learning rate:
$lrate = d_{model}^{-0.5} \cdot min(step\_num^{-0.5}, step\_num \cdot warmup\_steps^{-1.5})$
The learning rate resembles the first warm up training steps, we won't expect a decline after a find a relative inverse square root of the step number. The $\mathrm{warmup\_steps}$ is equal to 4000.

```python
class NoamOpt:
    "Optim wrapper that implements rate."
    def __init__(self, model_size, factor, warmup, optimizer):
        self.optimizer = optimizer
        self._step = 0
        self.warmup = warmup
        self.factor = factor
        self.model_size = model_size
        self._rate = 0

    def step(self):
        "Update parameters and rate"
        self._step += 1
        rate = self.rate()
        for p in self.optimizer.param_groups:
            p['lr'] = rate
        self._rate = rate
        self.optimizer.step()

    def rate(self, step = None):
        "Implement `lrate` above"
        if step is None:
            step = self._step
        return self.factor * \
            (self.model_size ** (-0.5) *
            min(step ** (-0.5), step * self.warmup ** (-1.5)))

def get_std_opt(model):
    return NoamOpt(model.src_embed[0].d_model, 2, 4000,
            torch.optim.Adam(model.parameters(), lr=0, betas=(0.9, 0.98), eps=1e-9))
```

```python
# Three settings of the lrate hyperparameters.
opts = [NoamOpt(512, 1, 4000, None),
        NoamOpt(512, 1, 8000, None),
        NoamOpt(256, 1, 4000, None)]
plt.plot(np.arange(1, 20000), [[opt.rate(i) for opt in opts] for i in range(1, 20000)])
plt.legend(["512:4000", "512:8000", "256:4000"])
None
```



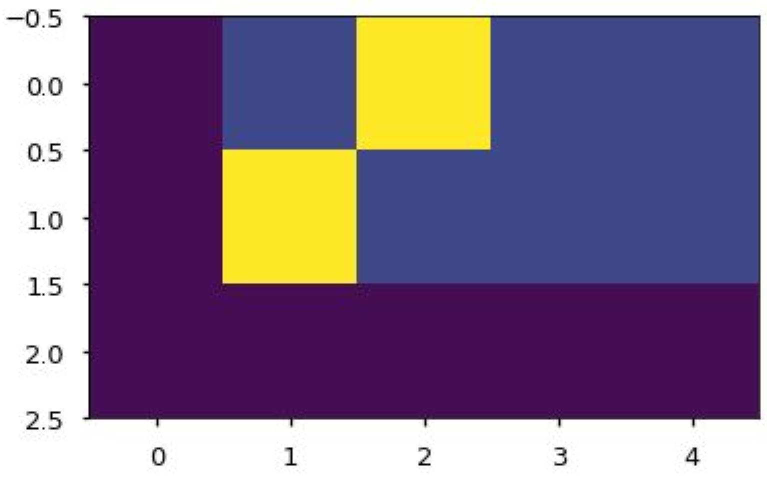
Regularization (Label Smoothing)

There are a tremendous amount of weight parameters used to train this architecture. To have excellent performance in the BLEU score, we need to have a label smoothing value of (An amount using the KL div loss). We generate the correct word using the smoothing mass distributed throughout the vocabulary text corpse. The only tradeoff implementing smoothing parameters is the perplexity.

```python
class LabelSmoothing(nn.Module):
    "Implement label smoothing."
    def __init__(self, size, padding_idx, smoothing=0.0):
        super(LabelSmoothing, self).__init__()
        self.criterion = nn.KLDivLoss(size_average=False)
        self.padding_idx = padding_idx
        self.confidence = 1.0 - smoothing
        self.smoothing = smoothing
        self.size = size
        self.true_dist = None

    def forward(self, x, target):
        assert x.size(1) == self.size
        true_dist = x.data.clone()
        true_dist.fill_(self.smoothing / (self.size - 2))
        true_dist.scatter_(1, target.data.unsqueeze(1), self.confidence)
        true_dist[:, self.padding_idx] = 0
        mask = torch.nonzero(target.data == self.padding_idx)
        if mask.dim() > 0:
            true_dist.index_fill_(0, mask.squeeze(), 0.0)
        self.true_dist = true_dist
        return self.criterion(x, Variable(true_dist, requires_grad=False))
```

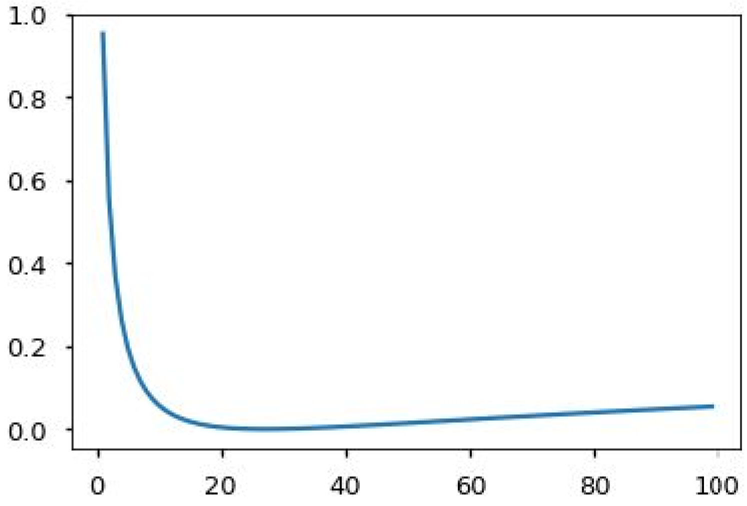
```python
# Example of label smoothing.
crit = LabelSmoothing(5, 0, 0.4)
predict = torch.FloatTensor([[0, 0.2, 0.7, 0.1, 0],
                             [0, 0.2, 0.7, 0.1, 0],
                             [0, 0.2, 0.7, 0.1, 0]])
v = crit(Variable(predict.log()),
         Variable(torch.LongTensor([2, 1, 0])))

# Show the target distributions expected by the system.
plt.imshow(crit.true_dist)
None
```

Here we the the mass distribution of words based on confidence.



Using th loss function, we can penalize the label smoothing if we're having an overfitting case, also known as over confident from our distribution.

```
crit = LabelSmoothing(5, 0, 0.1)
def loss(x):
    d = x + 3 * 1
    predict = torch.FloatTensor([[0, x / d, 1 / d, 1 / d, 1 / d],
                                 ])
    #print(predict)
    return crit(Variable(predict.log()),
                 Variable(torch.LongTensor([1]))).data[0]
plt.plot(np.arange(1, 100), [loss(x) for x in range(1, 100)])
None
```



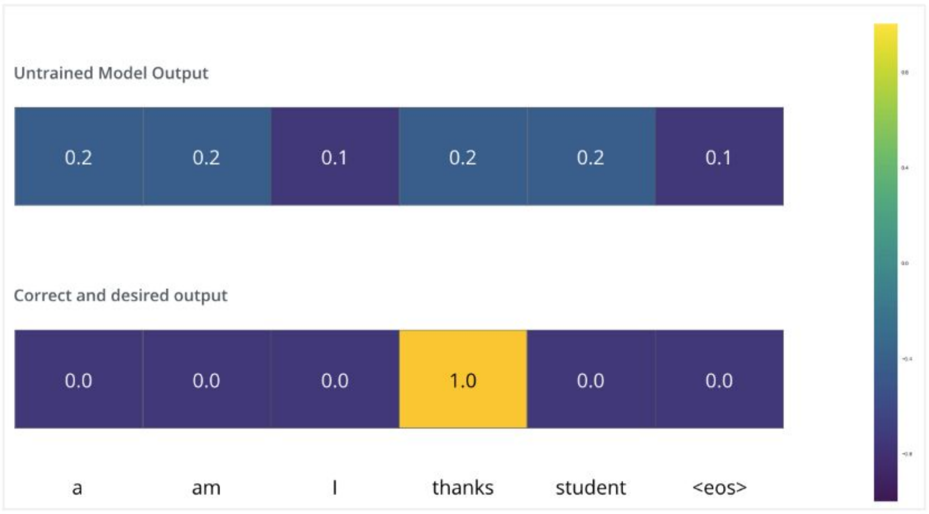
Synthetic Data

What we can do is to generate a random set of vocabulary words, we use a copy-task method.

```
def data_gen(V, batch, nbatches):
    "Generate random data for a src-tgt copy task."
    for i in range(nbatches):
        data = torch.from_numpy(np.random.randint(1, V, size=(batch, 10)))
        data[:, 0] = 1
        src = Variable(data, requires_grad=False)
        tgt = Variable(data, requires_grad=False)
        yield Batch(src, tgt, 0)
```

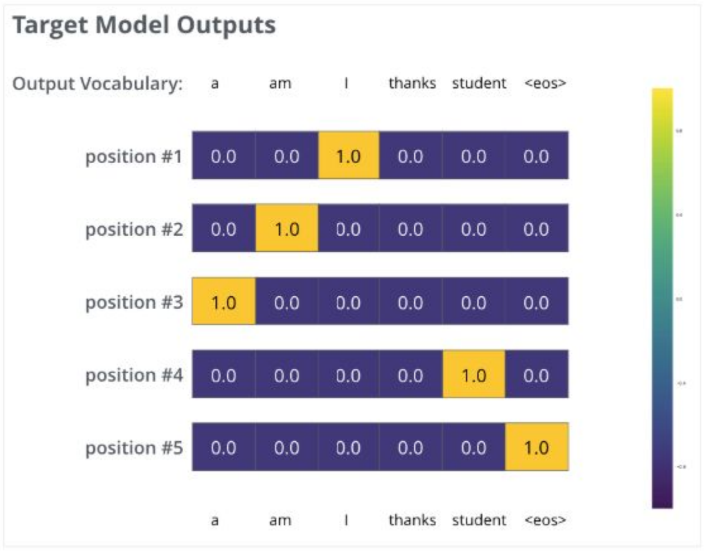
The Loss Function

Let's say for example we’re translating the word “merci” to “thanks”, so we want the probability distribution to output “thanks” so then start the training process.

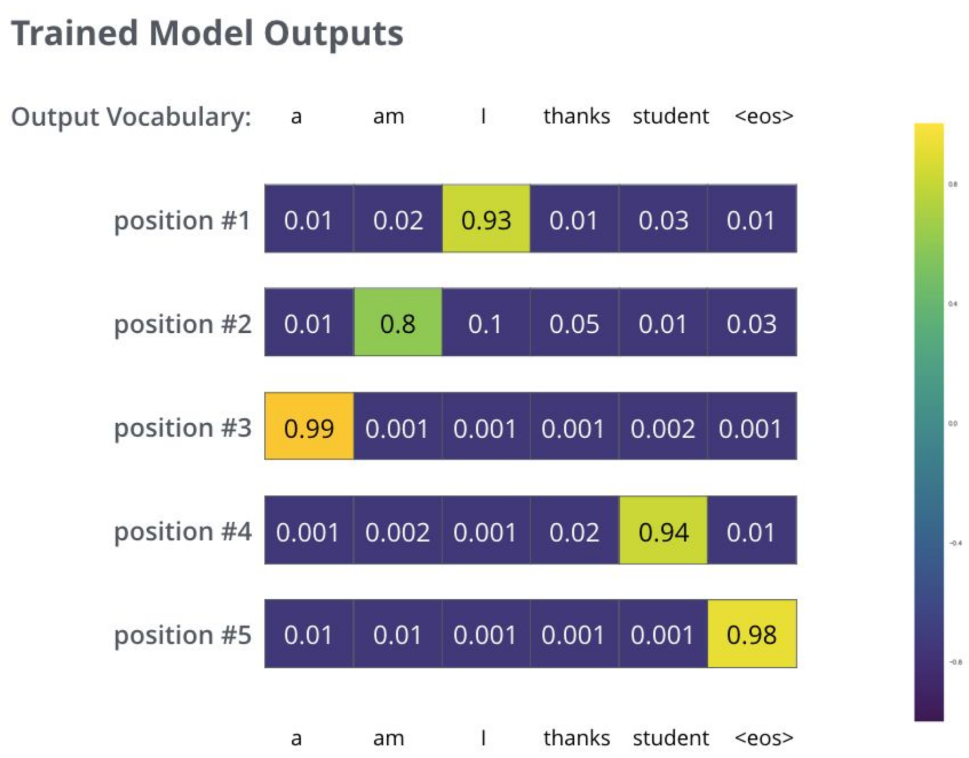


Before training, the weight parameters are randomly initialized (untrained) so the probability distribution generates us a random word. We can compare the created word with the expected output (lost function) to optimize our parameters using backpropagation to map our expected targeted word closely.

There are a number of techniques used to compare two probability distribution (aka loss function): cross-entropy & Kullback-Leribar divergence



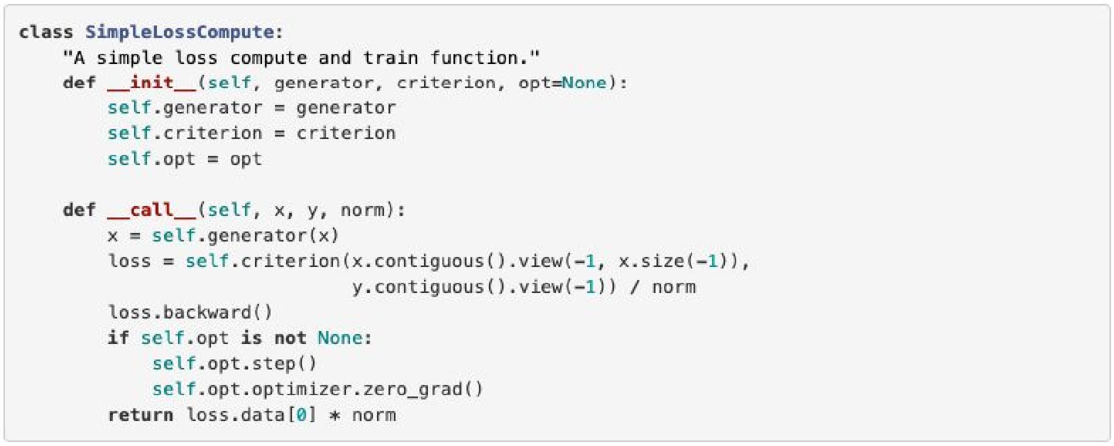
After a series of training, we could expect a probability distribution that looks like the following:



After training, we have a probability to determine which among the words had the highest score for their respected position. It looks like the word “I” is the first position which is common sense for a human. In order we have generated the sentence in the following sequence:

“I am a student <eos>”

There were no high probability distribution for the word “thanks”, so we disregard it’s context for a lack of relation between other words.



```
class SimpleLossCompute:
    "A simple loss compute and train function."
    def __init__(self, generator, criterion, opt=None):
        self.generator = generator
        self.criterion = criterion
        self.opt = opt

    def __call__(self, x, y, norm):
        x = self.generator(x)
        loss = self.criterion(x.contiguous().view(-1, x.size(-1)),
                              y.contiguous().view(-1)) / norm
        loss.backward()
        if self.opt is not None:
            self.opt.step()
            self.opt.optimizer.zero_grad()
        return loss.data[0] * norm
```

Greedy Decoding

```
# Train the simple copy task.
V = 11
criterion = LabelSmoothing(size=V, padding_idx=0, smoothing=0.0)
model = make_model(V, V, N=2)
model_opt = NoamOpt(model.src_embed[0].d_model, 1, 400,
        torch.optim.Adam(model.parameters(), lr=0, betas=(0.9, 0.98), eps=1e-9))

for epoch in range(10):
    model.train()
    run_epoch(data_gen(V, 30, 20), model,
              SimpleLossCompute(model.generator, criterion, model_opt))
    model.eval()
    print(run_epoch(data_gen(V, 30, 5), model,
                    SimpleLossCompute(model.generator, criterion, None)))
```

```
Epoch Step: 1 Loss: 1.031042 Tokens per Sec: 434.557008
Epoch Step: 1 Loss: 0.437069 Tokens per Sec: 643.630322
0.4323212027549744
Epoch Step: 1 Loss: 0.617165 Tokens per Sec: 436.652626
Epoch Step: 1 Loss: 0.258793 Tokens per Sec: 644.372296
0.27331129014492034
```

```
def greedy_decode(model, src, src_mask, max_len, start_symbol):
    memory = model.encode(src, src_mask)
    ys = torch.ones(1, 1).fill_(start_symbol).type_as(src.data)
    for i in range(max_len-1):
        out = model.decode(memory, src_mask,
                           Variable(ys),
                           Variable(subsequent_mask(ys.size(1))
                                    .type_as(src.data)))
        prob = model.generator(out[:, -1])
        _, next_word = torch.max(prob, dim = 1)
        next_word = next_word.data[0]
        ys = torch.cat([ys,
                        torch.ones(1, 1).type_as(src.data).fill_(next_word)], dim=1)
    return ys

model.eval()
src = Variable(torch.LongTensor([[1,2,3,4,5,6,7,8,9,10]]) )
src_mask = Variable(torch.ones(1, 1, 10) )
print(greedy_decode(model, src, src_mask, max_len=10, start_symbol=1))
```

‘

A Real World Example

Using the English-German text corpse for this example, we will utilize a multi-gpu processing to make it efficient.

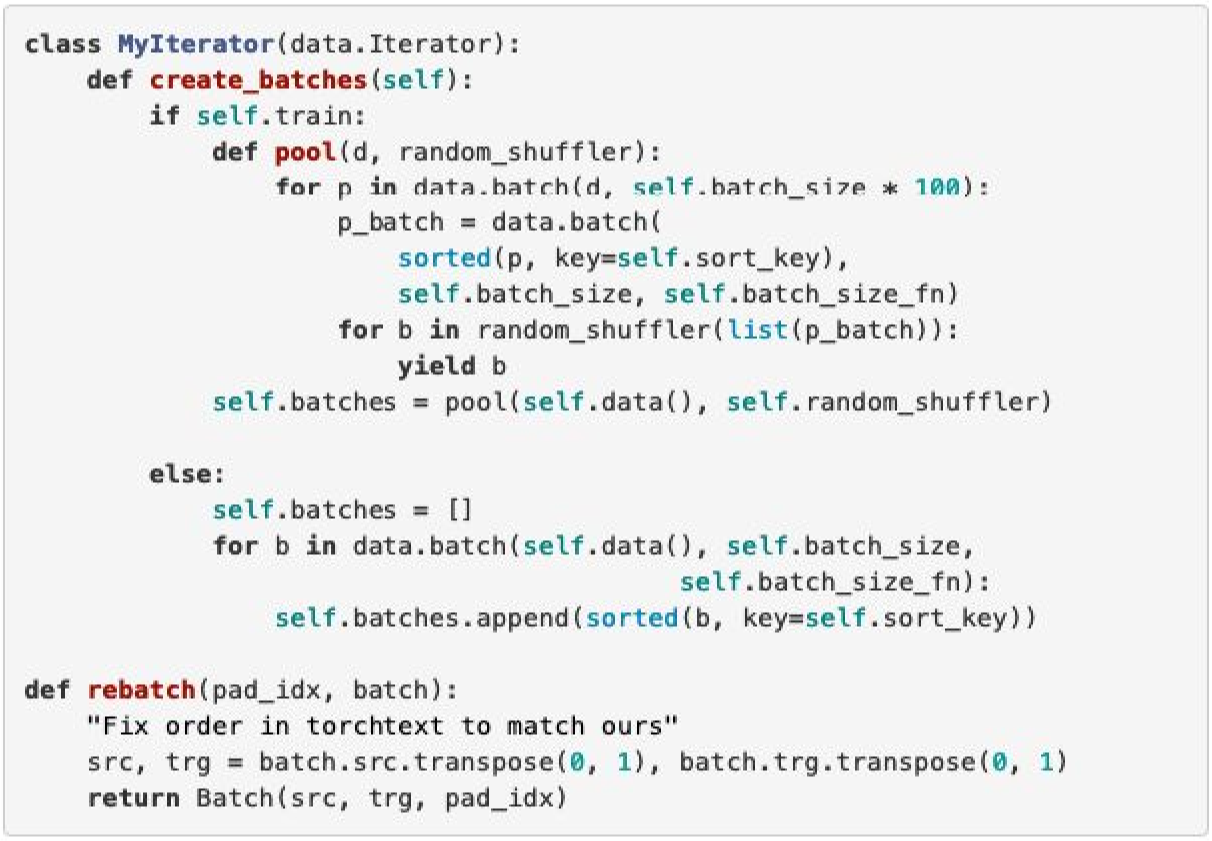
```
#!pip install torchtext spacy
#!python -m spacy download en
#!python -m spacy download de
```

Data Loading

Load dataset using torchtext & spacy using tokenization.

```
# For data loading.
from torchtext import data, datasets

if True:
    import spacy
    spacy_de = spacy.load('de')
    spacy_en = spacy.load('en')

    def tokenize_de(text):
        return [tok.text for tok in spacy_de.tokenizer(text)]

    def tokenize_en(text):
        return [tok.text for tok in spacy_en.tokenizer(text)]

    BOS_WORD = '<s>'
    EOS_WORD = '</s>'
    BLANK_WORD = "<blank>"
    SRC = data.Field(tokenize=tokenize_de, pad_token=BLANK_WORD)
    TGT = data.Field(tokenize=tokenize_en, init_token = BOS_WORD,
                     eos_token = EOS_WORD, pad_token=BLANK_WORD)

    MAX_LEN = 100
    train, val, test = datasets.IWSLT.splits(
        exts=('.de', '.en'), fields=(SRC, TGT),
        filter_pred=lambda x: len(vars(x)['src']) <= MAX_LEN and
            len(vars(x)['trg']) <= MAX_LEN)
    MIN_FREQ = 2
    SRC.build_vocab(train.src, min_freq=MIN_FREQ)
    TGT.build_vocab(train.trg, min_freq=MIN_FREQ)
```

Iterators



```
class MyIterator(data.Iterator):
    def create_batches(self):
        if self.train:
            def pool(d, random_shuffler):
                for p in data.batch(d, self.batch_size * 100):
                    p_batch = data.batch(
                        sorted(p, key=self.sort_key),
                        self.batch_size, self.batch_size_fn)
                    for b in random_shuffler(list(p_batch)):
                        yield b
            self.batches = pool(self.data(), self.random_shuffler)

        else:
            self.batches = []
            for b in data.batch(self.data(), self.batch_size,
                                          self.batch_size_fn):
                self.batches.append(sorted(b, key=self.sort_key))

def rebatch(pad_idx, batch):
    "Fix order in torchtext to match ours"
    src, trg = batch.src.transpose(0, 1), batch.trg.transpose(0, 1)
    return Batch(src, trg, pad_idx)
```

Multi-GPU Training

If you ever need to train your algorithm multiple GPU's, assuming your model takes days to optimize, then look no further! Using PyTorch parallel GPU primitives, we can do the following after splitting words at training time into chunks:

- replicate - split modules onto different gpus.
- scatter - split batches onto different gpus
- parallel_apply - apply module to batches on different gpus
- gather - pull scattered data back onto one gpu.
- nn.DataParallel - a special module wrapper that calls these all before evaluating.

```
# Skip if not interested in multigpu.
class MultiGPULossCompute:
    "A multi-gpu loss compute and train function."
    def __init__(self, generator, criterion, devices, opt=None, chunk_size=5):
        # Send out to different gpus.
        self.generator = generator
        self.criterion = nn.parallel.replicate(criterion,
                                               devices=devices)
        self.opt = opt
        self.devices = devices
        self.chunk_size = chunk_size

    def __call__(self, out, targets, normalize):
        total = 0.0
        generator = nn.parallel.replicate(self.generator,
                                          devices=self.devices)
        out_scatter = nn.parallel.scatter(out,
                                          target_gpus=self.devices)
        out_grad = [[] for _ in out_scatter]
        targets = nn.parallel.scatter(targets,
                                      target_gpus=self.devices)

        # Divide generating into chunks.
        chunk_size = self.chunk_size
        for i in range(0, out_scatter[0].size(1), chunk_size):
            # Predict distributions
            out_column = [[Variable(o[:, i:i+chunk_size].data,
                                    requires_grad=self.opt is not None)]
                           for o in out_scatter]
            gen = nn.parallel.parallel_apply(generator, out_column)

            # Compute loss.
            y = [(g.contiguous().view(-1, g.size(-1)),
                  t[:, i:i+chunk_size].contiguous().view(-1))
                 for g, t in zip(gen, targets)]
            loss = nn.parallel.parallel_apply(self.criterion, y)

            # Sum and normalize loss
            l = nn.parallel.gather(loss,
                                   target_device=self.devices[0])
            l = l.sum()[0] / normalize
            total += l.data[0]

            # Backprop loss to output of transformer
            if self.opt is not None:
                l.backward()
                for j, l in enumerate(loss):
                    out_grad[j].append(out_column[j][0].grad.data.clone())

        # Backprop all loss through transformer.
        if self.opt is not None:
            out_grad = [Variable(torch.cat(og, dim=1)) for og in out_grad]
            o1 = out
            o2 = nn.parallel.gather(out_grad,
                                    target_device=self.devices[0])
            o1.backward(gradient=o2)
            self.opt.step()
            self.opt.optimizer.zero_grad()
        return total * normalize
```

Initialized: model, criterion, optimizer, iterators (data), and hardware parallel computation.

```
# GPUs to use
devices = [0, 1, 2, 3]
if True:
    pad_idx = TGT.vocab.stoi["<blank>"]
    model = make_model(len(SRC.vocab), len(TGT.vocab), N=6)
    model.cuda()
    criterion = LabelSmoothing(size=len(TGT.vocab), padding_idx=pad_idx, smoothing=0.1)
    criterion.cuda()
    BATCH_SIZE = 12000
    train_iter = MyIterator(train, batch_size=BATCH_SIZE, device=0,
                            repeat=False, sort_key=lambda x: (len(x.src), len(x.trg)),
                            batch_size_fn=batch_size_fn, train=True)
    valid_iter = MyIterator(val, batch_size=BATCH_SIZE, device=0,
                            repeat=False, sort_key=lambda x: (len(x.src), len(x.trg)),
                            batch_size_fn=batch_size_fn, train=False)
    model_par = nn.DataParallel(model, device_ids=devices)
None
```

Training the System

All is left is to training our algorithm with multiple GPU's

```
#!wget https://s3.amazonaws.com/opennmt-models/iwslt.pt
```

```
if False:
    model_opt = NoamOpt(model.src_embed[0].d_model, 1, 2000,
            torch.optim.Adam(model.parameters(), lr=0, betas=(0.9, 0.98), eps=1e-9))
    for epoch in range(10):
        model_par.train()
        run_epoch((rebatch(pad_idx, b) for b in train_iter),
                  model_par,
                  MultiGPULossCompute(model.generator, criterion,
                                      devices=devices, opt=model_opt))
        model_par.eval()
        loss = run_epoch((rebatch(pad_idx, b) for b in valid_iter),
                          model_par,
                          MultiGPULossCompute(model.generator, criterion,
                          devices=devices, opt=None))
        print(loss)
else:
    model = torch.load("iwslt.pt")
```

Once the decoder is able to generate new words (translate) after training, we test the performance through validation set containing the first sentence. We need to utilize the greedy search algorithm for the lack of data.

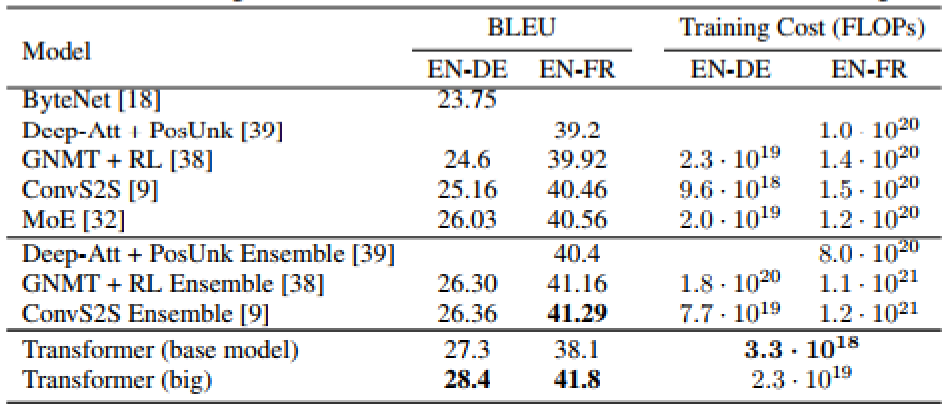
```
for i, batch in enumerate(valid_iter):
    src = batch.src.transpose(0, 1)[:1]
    src_mask = (src != SRC.vocab.stoi["<blank>"]).unsqueeze(-2)
    out = greedy_decode(model, src, src_mask,
                        max_len=60, start_symbol=TGT.vocab.stoi["<s>"])
    print("Translation:", end="\t")
    for i in range(1, out.size(1)):
        sym = TGT.vocab.itos[out[0, i]]
        if sym == "</s>": break
        print(sym, end =" ")
    print()
    print("Target:", end="\t")
    for i in range(1, batch.trg.size(0)):
        sym = TGT.vocab.itos[batch.trg.data[i, 0]]
        if sym == "</s>": break
        print(sym, end =" ")
    print()
    break
```

```
Translation:    <unk> <unk> . In my language , that means , thank you very much .
Gold:   <unk> <unk> . It means in my language , thank you very much .
```

Results

With the WMT English-to-German dataset, Transformers is the biggest trend in the NLP community outperforming previous models by more than 2.0 BLEU score of 28.4.

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.



Model	BLEU		Training Cost (FLOPs)	
	EN-DE	EN-FR	EN-DE	EN-FR
ByteNet [18]	23.75			
Deep-Att + PosUnk [39]		39.2		$1.0 \cdot 10^{20}$
GNMT + RL [38]	24.6	39.92	$2.3 \cdot 10^{19}$	$1.4 \cdot 10^{20}$
ConvS2S [9]	25.16	40.46	$9.6 \cdot 10^{18}$	$1.5 \cdot 10^{20}$
MoE [32]	26.03	40.56	$2.0 \cdot 10^{19}$	$1.2 \cdot 10^{20}$
Deep-Att + PosUnk Ensemble [39]		40.4		$8.0 \cdot 10^{20}$
GNMT + RL Ensemble [38]	26.30	41.16	$1.8 \cdot 10^{20}$	$1.1 \cdot 10^{21}$
ConvS2S Ensemble [9]	26.36	**41.29**	$7.7 \cdot 10^{19}$	$1.2 \cdot 10^{21}$
Transformer (base model)	27.3	38.1	$\mathbf{3.3 \cdot 10^{18}}$	
Transformer (big)	**28.4**	**41.8**	$2.3 \cdot 10^{19}$	

Base model code for OpenNMT

```
!wget https://s3.amazonaws.com/opennmt-models/en-de-model.pt
```

```
model, SRC, TGT = torch.load("en-de-model.pt")
```

```
model.eval()
sent = "▁The ▁log ▁file ▁can ▁be ▁sent ▁secret ly ▁with ▁email ▁or ▁FTP ▁to ▁a ▁specified ▁recei
ver".split()
src = torch.LongTensor([[SRC.stoi[w] for w in sent]])
src = Variable(src)
src_mask = (src != SRC.stoi["<blank>"]).unsqueeze(-2)
out = greedy_decode(model, src, src_mask,
                    max_len=60, start_symbol=TGT.stoi["<s>"])
print("Translation:", end="\t")
trans = "<s> "
for i in range(1, out.size(1)):
    sym = TGT.itos[out[0, i]]
    if sym == "</s>": break
    trans += sym + " "
print(trans)
```

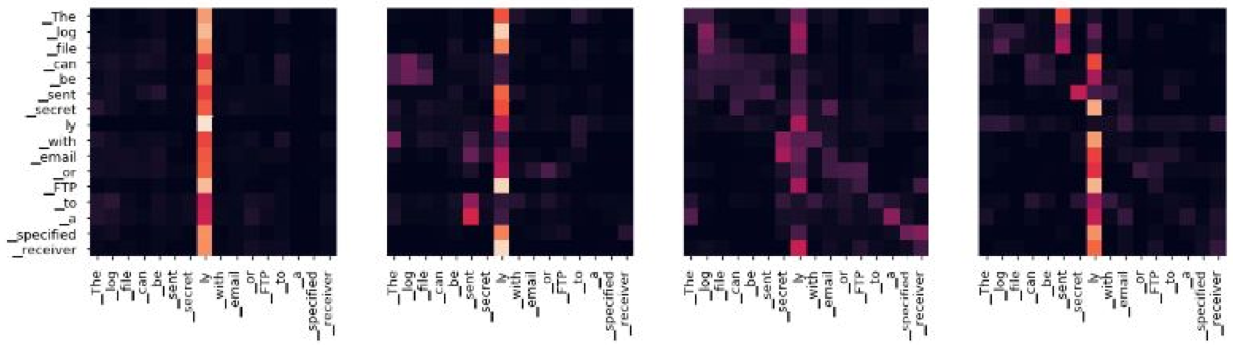
```
Translation:    <s> ▁Die ▁Protokoll datei ▁kann ▁ heimlich ▁per ▁E - Mail ▁oder ▁FTP ▁an ▁einen
▁bestimmte n ▁Empfänger ▁gesendet ▁werden .
```

Attention Visualization

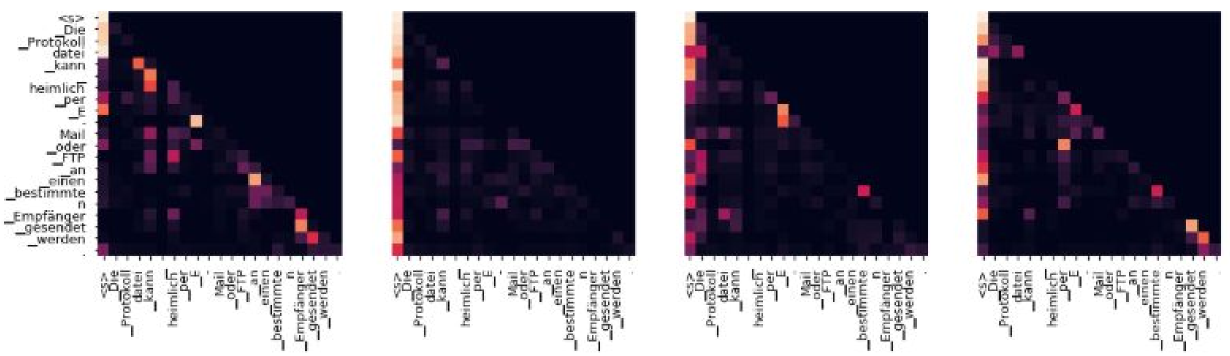
If you wish to see the visual effects of each layer of the attention model

```
tgt_sent = trans.split()
def draw(data, x, y, ax):
    seaborn.heatmap(data,
                    xticklabels=x, square=True, yticklabels=y, vmin=0.0, vmax=1.0,
                    cbar=False, ax=ax)

for layer in range(1, 6, 2):
    fig, axs = plt.subplots(1,4, figsize=(20, 10))
    print("Encoder Layer", layer+1)
    for h in range(4):
        draw(model.encoder.layers[layer].self_attn.attn[0, h].data,
            sent, sent if h ==0 else [], ax=axs[h])
    plt.show()

for layer in range(1, 6, 2):
    fig, axs = plt.subplots(1,4, figsize=(20, 10))
    print("Decoder Self Layer", layer+1)
    for h in range(4):
        draw(model.decoder.layers[layer].self_attn.attn[0, h].data[:len(tgt_sent), :len(tgt_sent
)],
            tgt_sent, tgt_sent if h ==0 else [], ax=axs[h])
    plt.show()
    print("Decoder Src Layer", layer+1)
    fig, axs = plt.subplots(1,4, figsize=(20, 10))
    for h in range(4):
        draw(model.decoder.layers[layer].self_attn.attn[0, h].data[:len(tgt_sent), :len(sent)],
            sent, tgt_sent if h ==0 else [], ax=axs[h])
    plt.show()
```

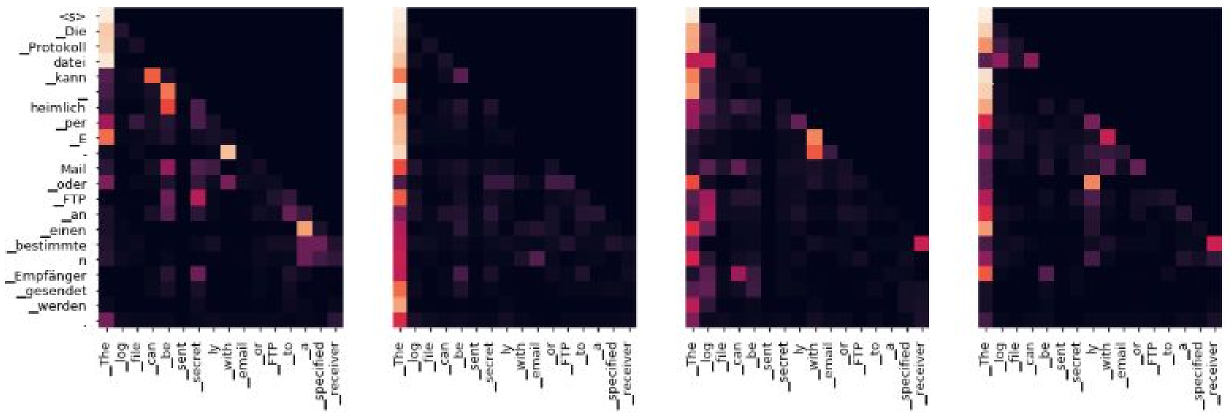
```
Encoder Layer 6
```



Decoder Self Layer 6



Decoder Src Layer 6


<s>
_Die
_Protokoll
datei
_kann
heimlich
_per
_E
Mail
_oder
_FTP
_an
_einen
_bestimmte
n
_Empfänger
_gesendet
_werden
_The
_log
_file
_can
_be
_sent
_secret
ly
_with
_email
_or
_FTP
_to
_a
_specified
_receiver