TDLS Code Review

Transformer

(Attention Is All You Need)

Lead by **Xiyang Chen**

Facilitated by **Felipe Pérez** & **Ehsan Amjadian**

Outline

- Motivation
- PyTorch vs. Tensorflow
- Overview
- Individual parts
- Training & experiments
- Discussions

We will be mainly focusing on code.
For more detailed explanation, refer to a previous TDLS session on Transformer
presented by **Joseph Palermo**.

Why code review

Code review enables us to...

- ...look at details that are glossed over
  - Details only to be found on prior work
  - Practical constraints
    - Memory
    - Wall-clock time (a.k.a. paper deadline)
    - Source code availability
- ...use code to aid understanding
- ...use small experiments to test our assumptions
  - Poke, observe, believe
- Be lazy

Code reviews are new and experimental.
Suggestions are welcome!

Why Transformer

- In a way, it's attention to the extreme
- Achieves SotA's in sequence-related tasks
  - BERT
  - TransferTransfo (convo dialog generation)
  - Transformer-XL
- Foundation for many pioneering works
  - Image Transformer
  - Self-attention CycleGAN
  - AlphaStar
  - Cited by over 1,000 works as of early Feb
- Faster, more scalable, more interpretable
  - Unlike RNN, training can be completely parallelized across sequence timesteps

Pytorch vs. Tensorflow

- The official Transformer implementation is in Tensorflow
- Many people prefer PyTorch
- Which framework then?

Andrej Karpathy
@karpathy

Follow

I've been using PyTorch a few months now and I've never felt better. I have more energy. My skin is clearer. My eye sight has improved.

TF vs. PyTorch: practical pros and cons as of Feb 2019

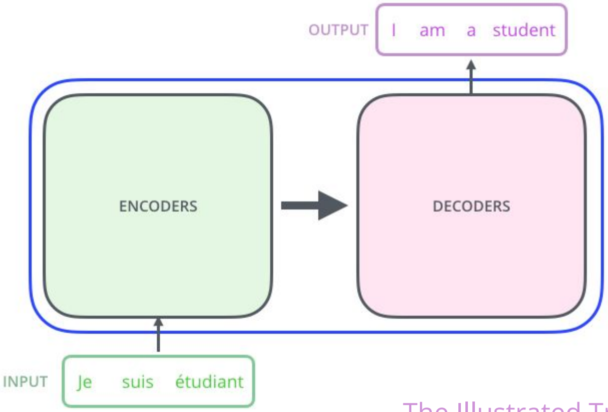
- PyTorch is a very popular choice among researchers
  - Intuitive and flexible
  - Easier to work with CUDA
- TF is production friendly
  - TF Serving gives you both TCP & RESTful APIs
  - TF has more support on most popular cloud platforms (GCP, AWS, etc) in terms of code examples and guides
- TF spans more platforms and types of devices
  - TPU, GPU, CPU, Mobile (TF Lite), Browser/Node.js (TF.js), Edge TPU
- TF's Static Graph mode boosts performance, but is cumbersome to code with, especially for rapid prototyping and experimentation
- TF Eager comes to the rescue
  - API is similar to that of PyTorch and MXNet
  - Can use AutoGraph in Eager Mode to generate graph code
  - Will become the default mode in TF 2.0
  - However, beware: TF Eager is still new. A lot of existing TF code is not compatible with Eager Mode yet

TF vs. PyTorch

- With all considered, we will base our review on the The Annotated Transformer, a PyTorch implementation by Harvard NLP.
- There are other implementations that may be more suitable for your purpose.
  - Check out the reference slide.

Overview

- At high level, an encoder-decoder architecture
- N = 6
- Input size: 512
- Output size: 512
- Output size for most layers: 512



[The Illustrated Transformer]

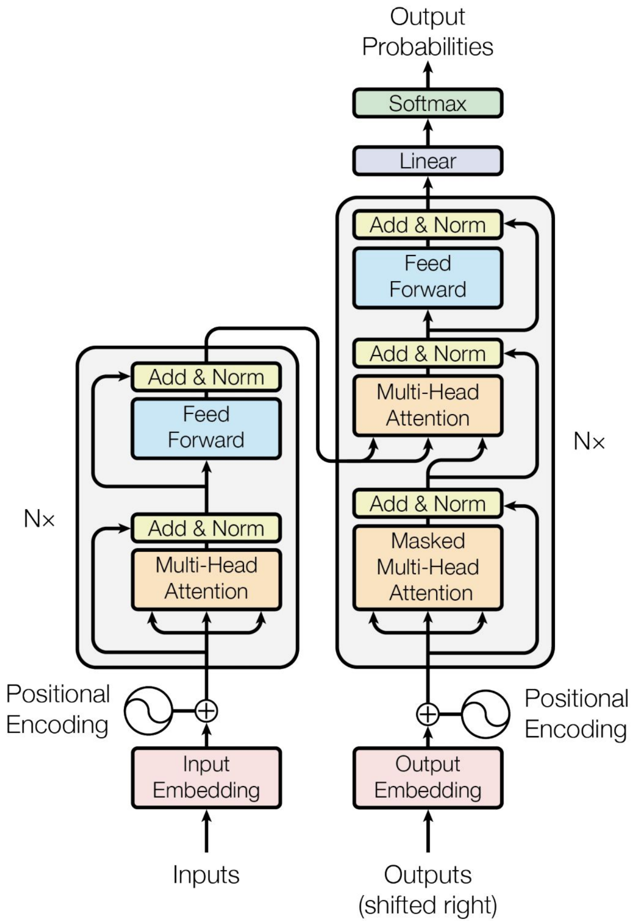
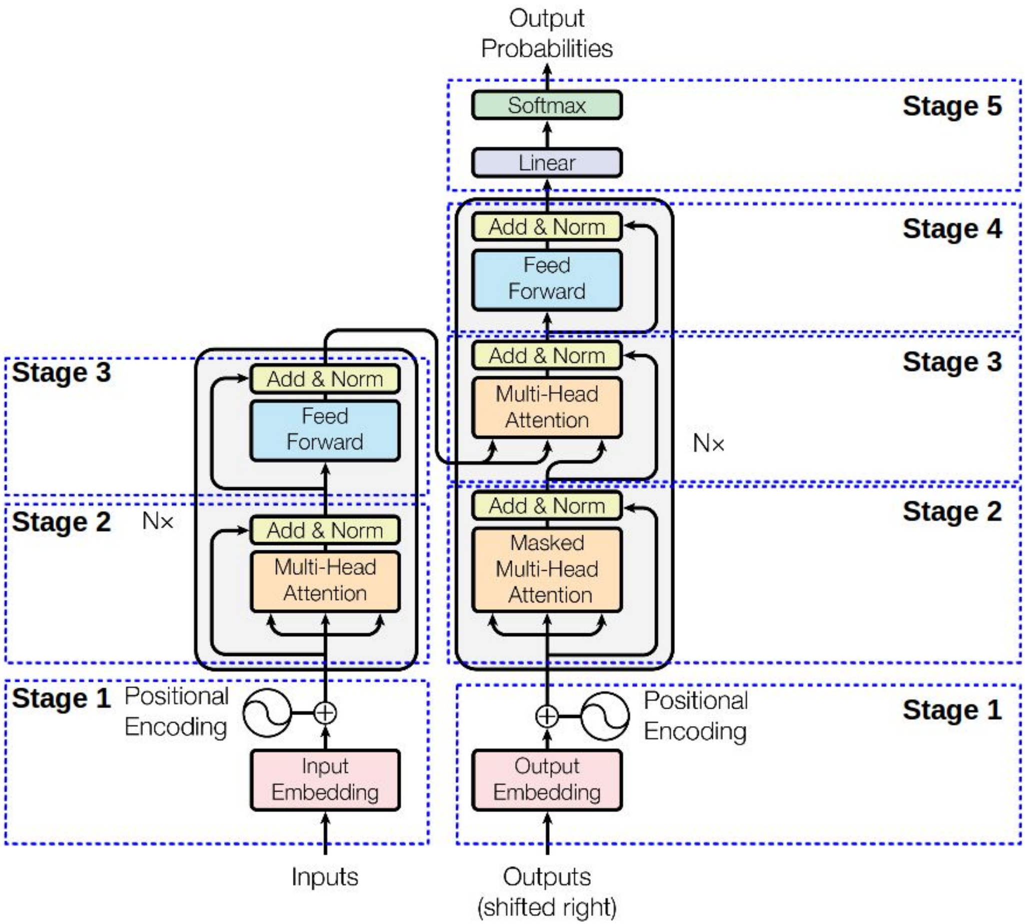


Figure 1: The Transformer - model architecture.

Overview

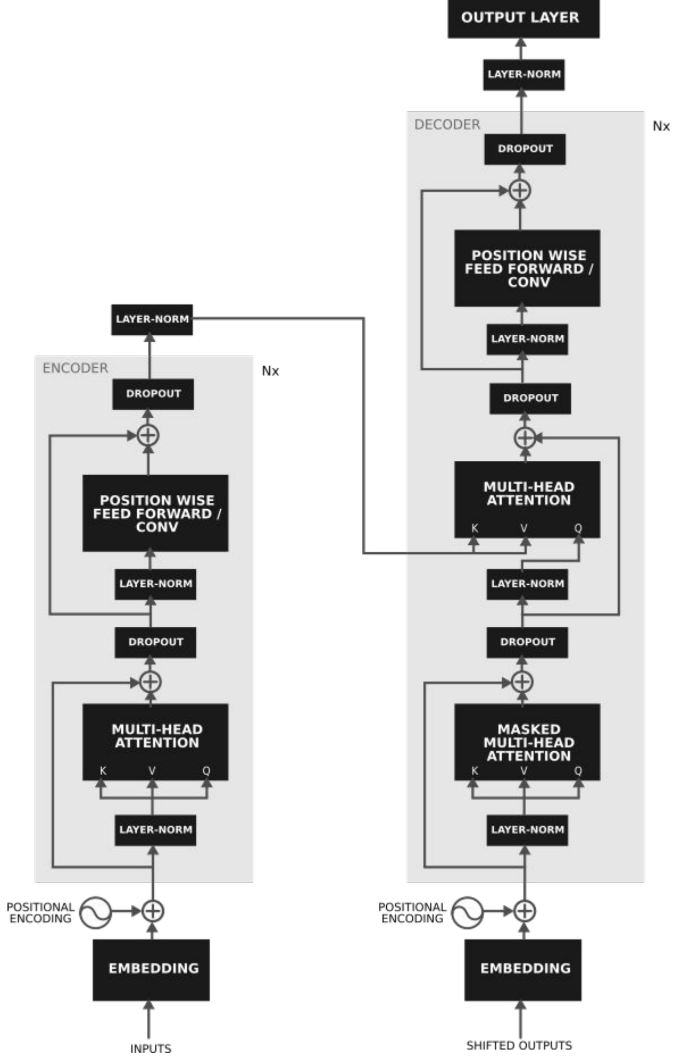
- Source data ⇒ encoder
- Target data ⇒ decoder
  - Target data is required for learning the context (e.g. what words have been translated so far)
  - Therefore, masked
- Output is compared against target data
  - Loss is KL_Div(x, Targ)



[Michał Chromiak's blog post on Transformer]

Overview

- Two unique parts:
  - The Multiheaded Attention layer
  - Positionwise Feed-Forward layer
- Other parts:
  - Positional Encodings
  - Masks
  - Embeddings
- Loss & training
  - Single GPU
  - Multiple GPU



PyTorch Preliminaries

- Modules inherit from `nn.Module`
  - We supplies two functions: `__init__()` and `forward()`
- Function →... → Function → Module/model definition (`class Net:...`) → instantiation `net = Net(x)`
  - Instantiation wraps the internal `__call__()` method, so that we can do `y = net(x)`
- Any object which is of type “Variable” and is attached to the class definition will be automatically added for gradient computation
  - Unless we explicitly disable gradient computation

Generic enc-dec

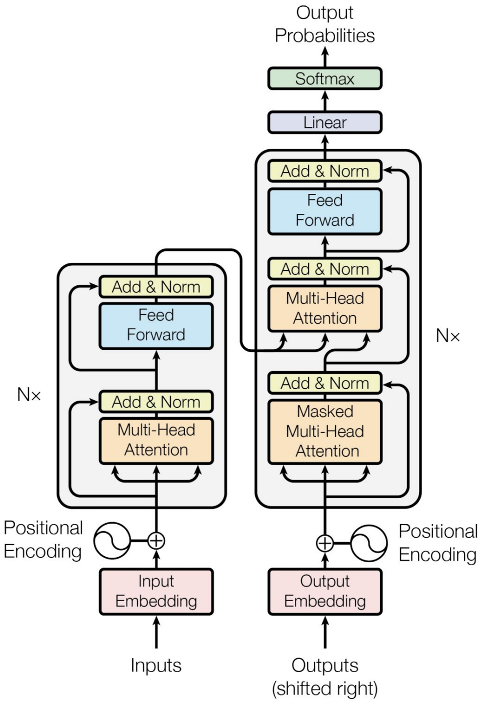


Figure 1: The Transformer - model architecture.

```
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):
    ...

    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)
```

Encoder

```python
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)
```

```python
class EncoderLayer(nn.Module):
    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)
```

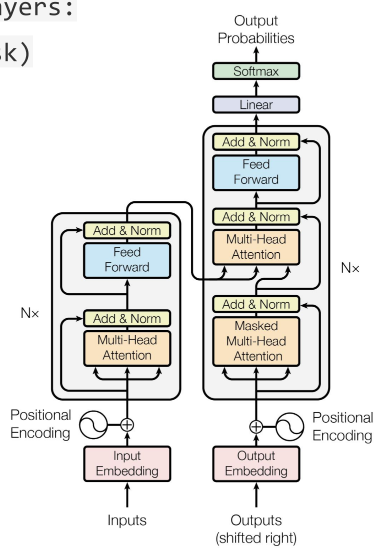


Figure 1: The Transformer - model architecture.

Encoder

```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):
        return x + self.dropout(sublayer(self.norm(x)))
```

```python
class EncoderLayer(nn.Module):
    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)
```

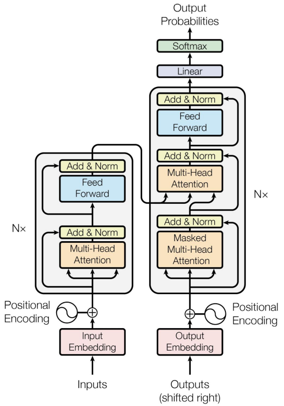


Figure 1: The Transformer - model architecture.

Decoder

```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):
    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)
```

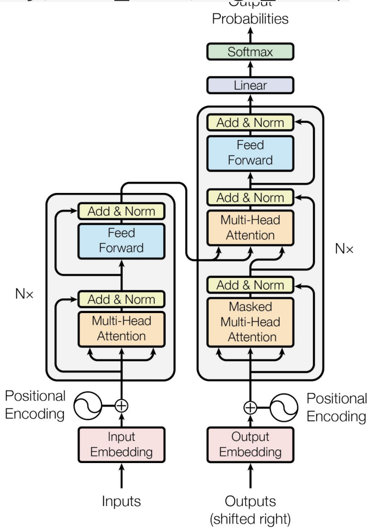


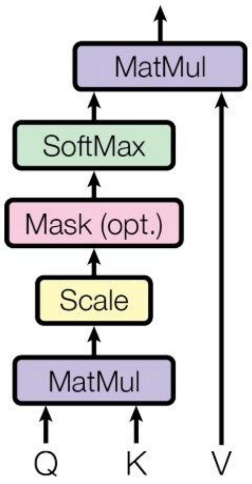
Figure 1: The Transformer - model architecture.

Attention overview

- ***Keys***: A sequence of vectors also known as the memory
- ***Values***: A sequence of vectors from which we aggregate the output through a weighted linear combination. Often Keys serve as Values.
- ***Query***: A single vector that we use to probe the Keys
- ***Output:*** A single vector which is derived from a linear combination of the *Values* using the probabilities from the previous step as weights

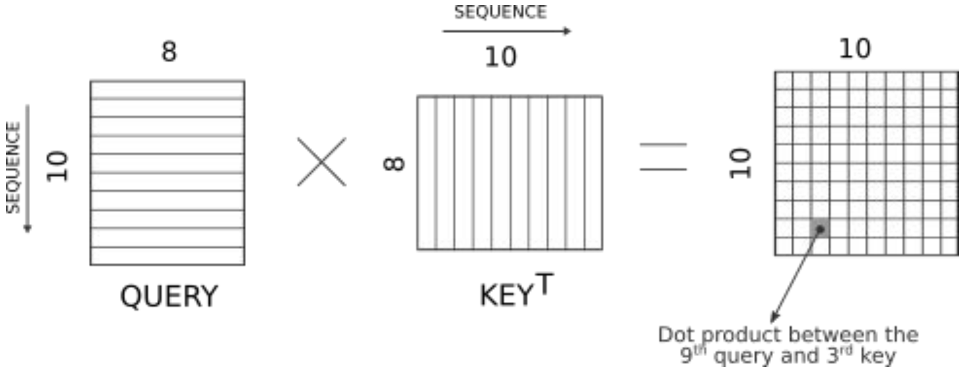
Building the Mighty Transformer for Sequence Tagging in PyTorch

Scaled Dot-Product Attention



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

$$\sigma(\mathbf{z})_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} \quad \text{for } j = 1, \ldots, K.$$



Attention, attention, attention

[Tensor2Tensor Transformers (slides by Łukasz Kaiser)]

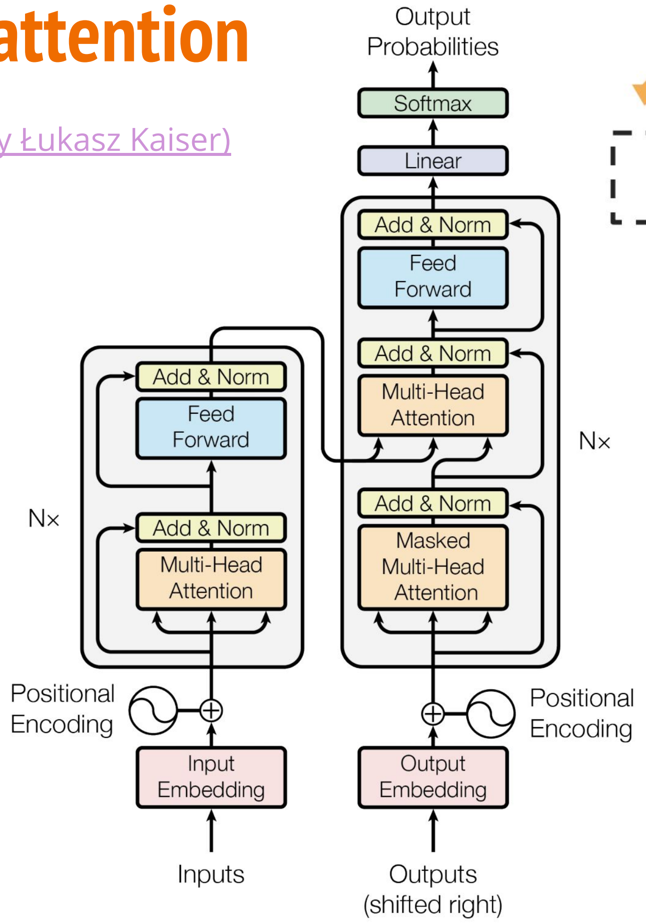
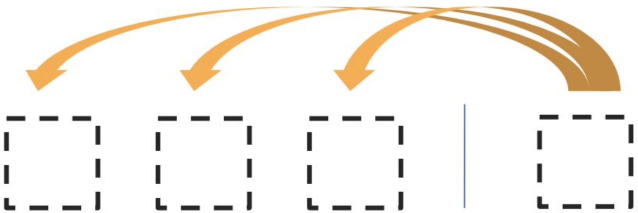
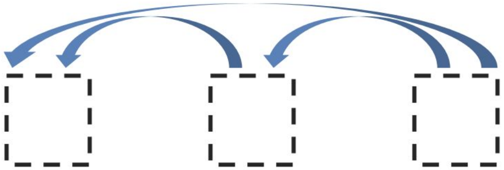


Figure 1: The Transformer - model architecture.



Encoder-Decoder Attention

Encoder Self-Attention

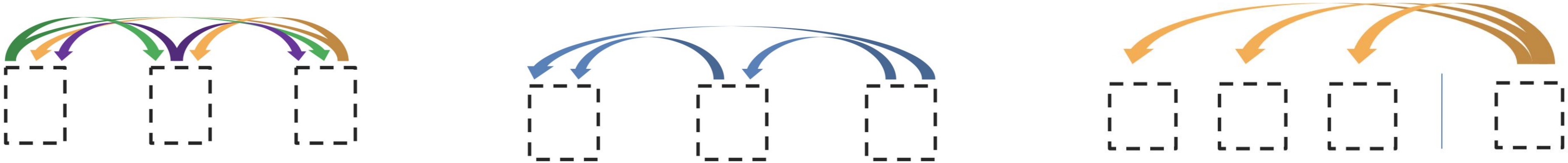


MaskedDecoder Self-Attention

Self Attention vs enc-dec attention

$\texttt{self\_attn\_map = attention(x}^{=bn*8*64*512}\texttt{, x}^{=ditto}\texttt{, x}^{=ditto}\texttt{, mask)}$

$\texttt{enc\_dec\_attn\_map = attention(Q\_split}^{=bn*8*64*512}\texttt{, K\_split, V\_split, mask)}$



Encoder Self-Attention

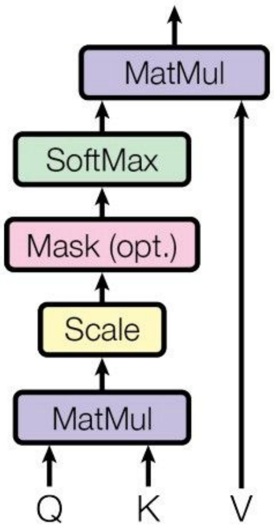
MaskedDecoder Self-Attention

Encoder-Decoder Attention

[Tensor2Tensor Transformers (slides by Łukasz Kaiser)]

Generic Attention

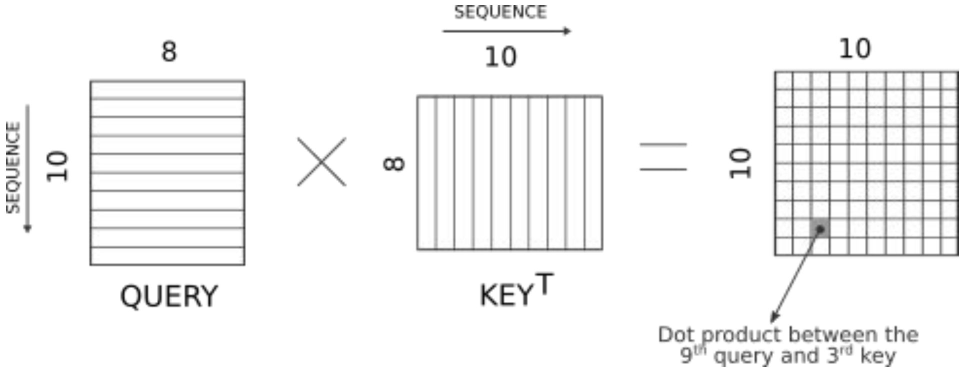
Scaled Dot-Product Attention




```
def attention(query^bn*H*S*D, key^bn*H*S*D, value^bn*H*S*D, mask = None):
    d_k^=D = 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^=bn*H*S*S = F.softmax(scores, dim = -1)
    return torch.matmul(p_attn^=bn*H*S*S, value^=bn*H*S*D)^=bn*H*S*D, p_attn
```

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

$$\sigma(\mathbf{z})_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} \quad \text{for } j = 1, \ldots, K.$$

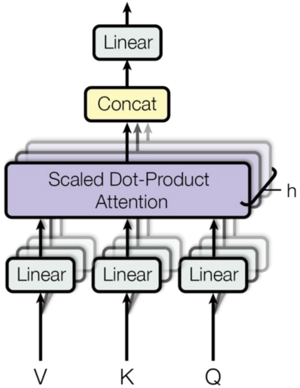


Q, K & V:4D Tensors
[batch_size, num_heads, seq_len, depth/num_heads]

Multi-Head Attention

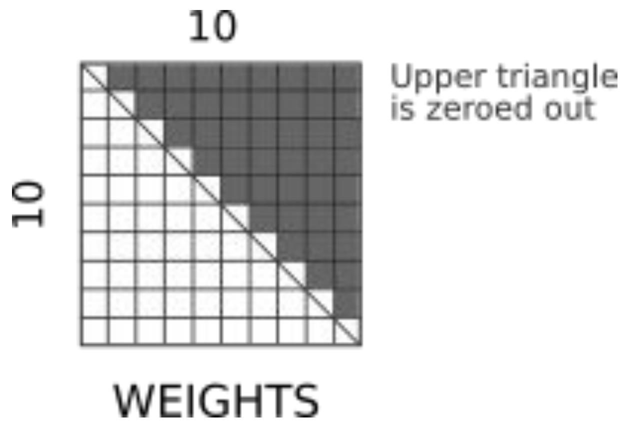
```
class MultiHeadedAttention(nn.Module):
def __init__(self, h=8, d_model=512, dropout=0.1):
    "Take in model size and number of heads."
    super(MultiHeadedAttention, self).__init__()
    assert d_model % h == 0
    # We assume d_v always equals d_k
    self.d_k=64 = d_model // h
    self.h = h=8
    self.linears = clones(
                nn.Linear(d_model, d_model)=512*512, 4
                    )
    self.attn = None
    self.dropout = nn.Dropout(p=dropout)
```

```
def forward(self, query=bs*512*512, key=ditto, value=ditto, 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=bs*512*8*64, key=ditto, value=ditto = \
        [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)
```



Masking

```python
def _gen_bias_mask(max_length):

    """

    Generates bias values (-Inf) to mask future timesteps during attention

    """

    np_mask = np.triu(np.full([max_length, max_length], -np.inf), 1)

    torch_mask = torch.from_numpy(np_mask).type(torch.FloatTensor)


    # Reshape to 4D Tensor to handle multiple heads

    return torch_mask.unsqueeze(0).unsqueeze(1)
```




```
>>> np.triu([[1,2,3],[4,5,6],[7,8,9],[10,11,12]], 1)
array([[ 0,  2,  3],
       [ 0,  0,  6],
       [ 0,  0,  9],
       [ 0,  0,  0]])
```

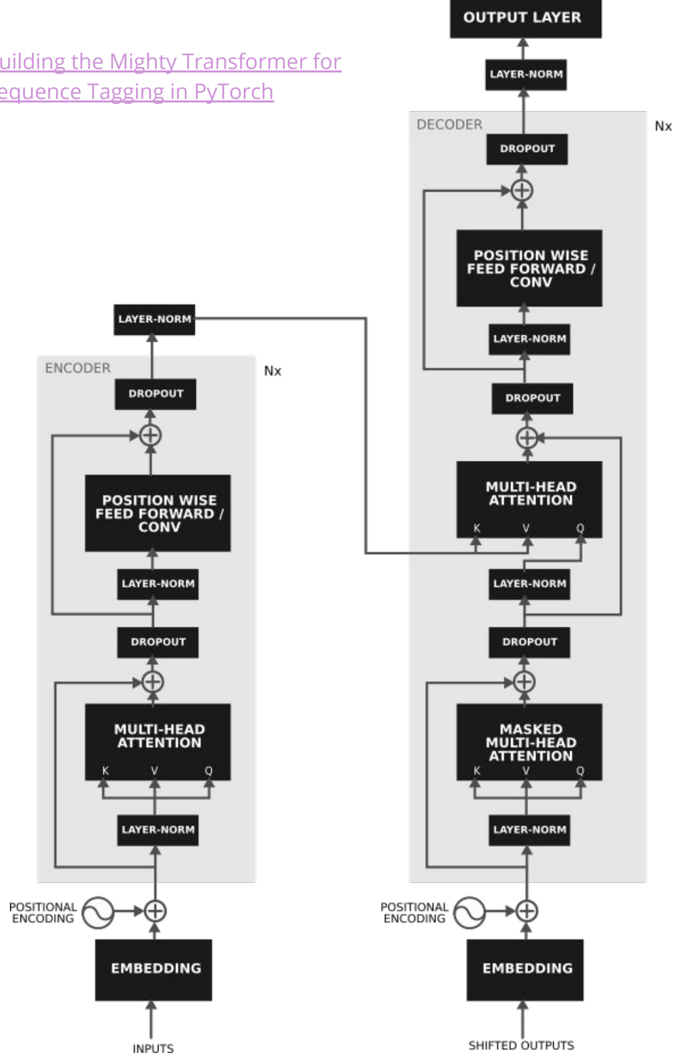
Positionwise Feed Forward

- applied to each position *separately* and *identically*

```
class PositionwiseFeedForward(nn.Module):
    "Implements FFN equation."
    def __init__(self, d_model=512, d_ff=2048, dropout=0.1):
        super(PositionwiseFeedForward, self).__init__()
        self.w_1=512x2048 = nn.Linear(d_model, d_ff)
        self.w_2=2048x512 = nn.Linear(d_ff, d_model)
        self.dropout = nn.Dropout(dropout)

    def forward=bn*S*512(self, x=bn*S*512):
        return self.w_2(self.dropout(F.relu(self.w_1(x))))
```

$$\mathrm{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2$$

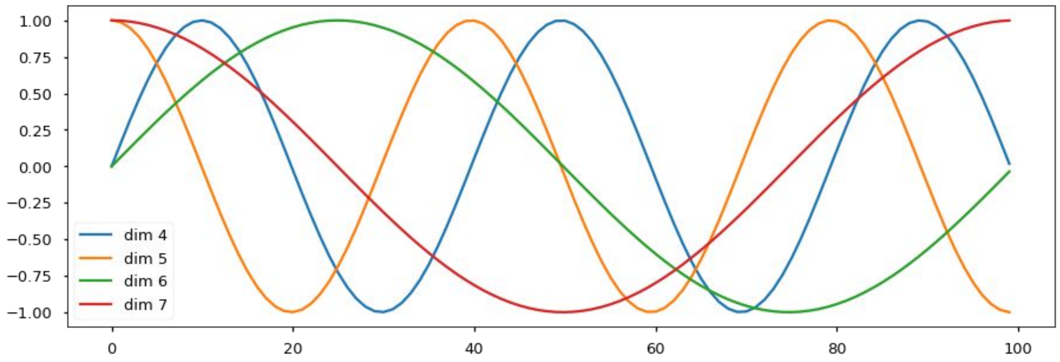
[Building the Mighty Transformer for Sequence Tagging in PyTorch]



Positional Encoding

Gehring, Jonas, et al. "Convolutional sequence to sequence learning." *arXiv preprint arXiv:1705.03122* (2017).

```
class PositionalEncoding(nn.Module):
    def __init__(self, d_model=512, dropout, max_len=5000):
        ...
        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)
```



LayerNorm

```python
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
```

Put Together

```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
```

Train it

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

    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
```

```python
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
```

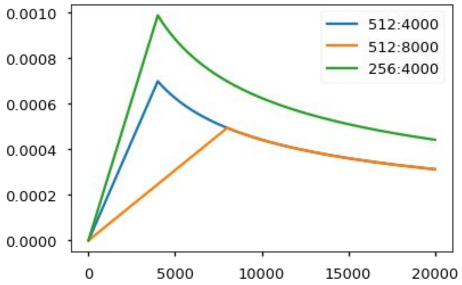
Train it

```python
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
```

```
Epoch Step: 1 Loss: 3.023465 Tokens per Sec: 403.074173
Epoch Step: 1 Loss: 1.920030 Tokens per Sec: 641.689380
1.9274832487106324
Epoch Step: 1 Loss: 1.940011 Tokens per Sec: 432.003378
Epoch Step: 1 Loss: 1.699767 Tokens per Sec: 641.979665
1.657595729827881
Epoch Step: 1 Loss: 1.860276 Tokens per Sec: 433.320240
Epoch Step: 1 Loss: 1.546011 Tokens per Sec: 640.537198
1.4888023376464843
...
Epoch Step: 1 Loss: 0.459483 Tokens per Sec: 434.594030
Epoch Step: 1 Loss: 0.290385 Tokens per Sec: 642.519464
0.2612409472465515
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
```

Optimizer

We used the Adam optimizer (cite) with $\beta_1 = 0.9$, $\beta_2 = 0.98$ and $\epsilon = 10^{-9}$. We varied the learning rate over the course of training, according to the formula: $lrate = d_{\text{model}}^{-0.5} \cdot \min(step\_num^{-0.5}, step\_num \cdot warmup\_steps^{-1.5})$ This corresponds to increasing the learning rate linearly for the first $warmup_steps$ training steps, and decreasing it thereafter proportionally to the inverse square root of the step number. We used $warmup_steps = 4000$.



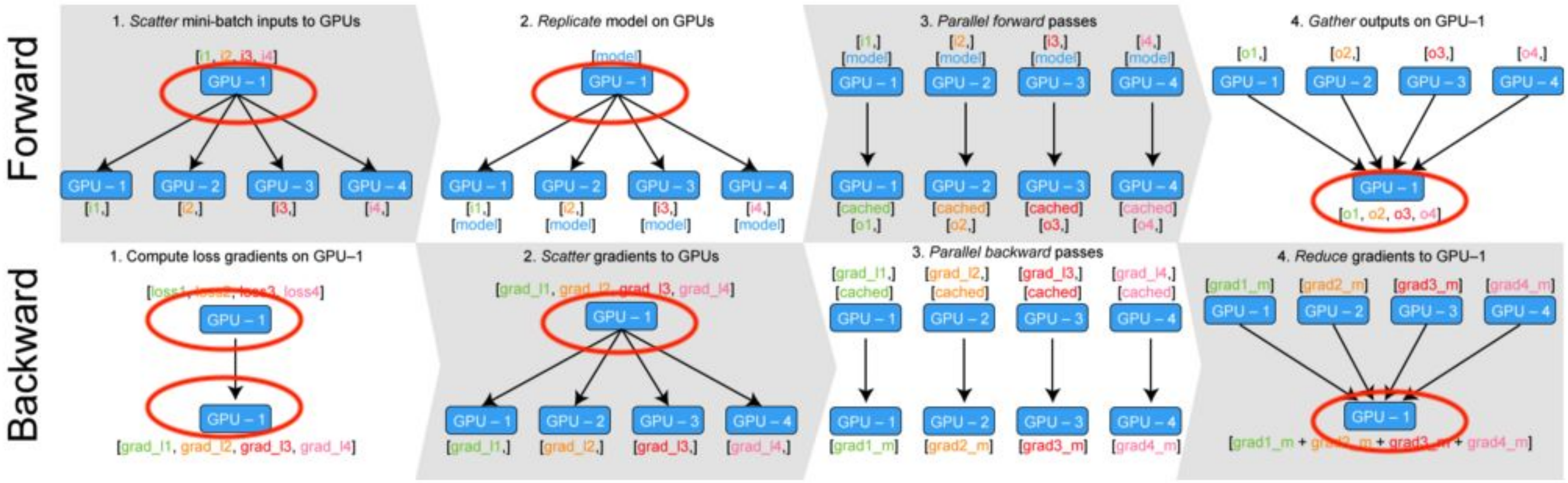

```python
class NoamOpt:
    "Optim wrapper that implements rate."
    def __init__(self, model_size, factor, warmup, optimizer):
        ...
        self._step = 0
        self._rate = 0

    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()
```

```python
    "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))
```

$$lrate = d_{\text{model}}^{-0.5} \cdot \min(step\_num^{-0.5}, step\_num \cdot warmup\_steps^{-1.5})$$

Multi-GPU training

- 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.



Forward and Backward passes with torch.nn.DataParallel

```
class MultiGPULossCompute:
...
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
```

Test it out: greedy decoding

```
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.item()
        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))
```

```
    1     2     3     4     5     6     7     8     9    10
[torch.LongTensor of size 1x10]
```

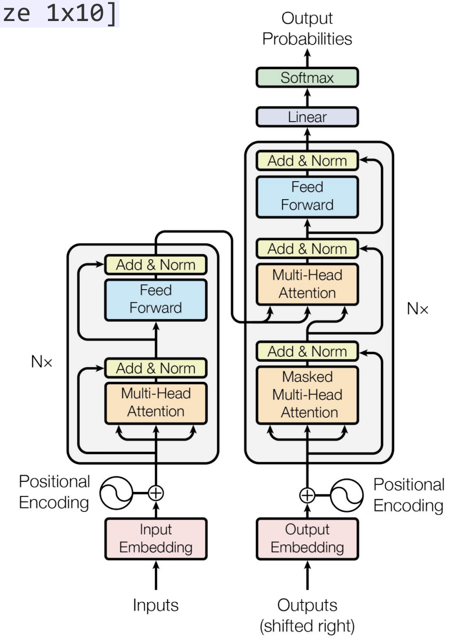


Figure 1: The Transformer - model architecture.

The Authors' Implementation

- https://github.com/tensorflow/tensor2tensor/blob/master/tensor2tensor/models/transformer.py
- TF Estimator
- Supports TPU, GPU & CPU
- Attention caching

Thanks

- Thanks to Renyu Li and another friend for helping with the preparation
- Thanks to Rouzbeh Afrasiabi and Dave Fernandes for suggesions

References

- Vaswani, Ashish, et al. "Attention is all you need." Advances in Neural Information Processing Systems. 2017
- The Annotated Transformer
- The Illustrated Transformer
- How to code The Transformer in Pytorch
- Michał Chromiak's blog post on Transformer
- Tensor2Tensor Transformers (slides by Łukasz Kaiser)
- Building the Mighty Transformer for Sequence Tagging in PyTorch
- Transformer from NLP Tutorial by Tae Hwan Jung(Jeff Jung) & Derek Miller

Model averaging

The paper averages the last k checkpoints to create an ensembling effect.

```
def average(model, models):
    "Average models into model"
    for ps in zip(*[m.params() for m in [model] + models]):
        p[0].copy_(torch.sum(*ps[1:]) / len(ps[1:]))
```