The Annotated Transformer

Alexander M. Rush
srush@seas.harvard.edu
Harvard University

Abstract

A major aim of open-source NLP is to quickly and accurately reproduce the results of new work, in a manner that the community can easily use and modify. While most papers publish enough detail for replication, it still may be difficult to achieve good results in practice. This paper is an experiment. In it, I consider a worked exercise with the goal of implementing the results of the recent paper. The replication exercise aims at simple code structure that follows closely with the original work, while achieving an efficient usable system. An implicit premise of this exercise is to encourage researchers to consider this method for new results.

1 Introduction

Replication of published results remains a challenging issue in open-source NLP. When a new paper is published with major improvements, it is common for many members of the community to independently reproduce the numbers experimentally, which is often a struggle. Practically this makes it difficult to improve scores, but also importantly it is a pedagogical issue if students cannot reproduce results from scientific publications.

The recent turn towards deep learning has exerbated this issue. New models require extensive hyperparameter tuning and long training times. Small mistakes can cause major issues. Fortunately though, new toolsets have made it possible to write simpler more mathematically declarative code.

In this experimental paper, I propose an exercise in open-source NLP. The goal is to transcribe a recent paper into a simple and understandable form. The document itself is presented as an annotated paper. That is the main document (in different font) is an excerpt of the recent paper “Attention is All You Need” (Vaswani et al., 2017). I add annotation in the form of italicized comments and include code in PyTorch directly in the paper itself.

Note this document itself is presented as a blog post 1 and is completely executable as a notebook. In the spirit of reproducibility this work itself is distilled from the same source with images inline.

---

1Presented at http://nlp.seas.harvard.edu/2018/04/03/attention.html with source code at https://github.com/harvardnlp/annotated-transformer
2 Background

The goal of reducing sequential computation also forms the foundation of the Extended Neural GPU, ByteNet and ConvS2S, all of which use convolutional neural networks as basic building block, computing hidden representations in parallel for all input and output positions. In these models, the number of operations required to relate signals from two arbitrary input or output positions grows in the distance between positions, linearly for ConvS2S and logarithmically for ByteNet. This makes it more difficult to learn dependencies between distant positions. In the Transformer this is reduced to a constant number of operations, albeit at the cost of reduced effective resolution due to averaging attention-weighted positions, an effect we counteract with Multi-Head Attention.

Self-attention, sometimes called intra-attention is an attention mechanism relating different positions of a single sequence in order to compute a representation of the sequence. Self-attention has been used successfully in a variety of tasks including reading comprehension, abstractive summarization, textual entailment and learning task-independent sentence representations. End-to-end memory networks are based on a recurrent attention mechanism instead of sequence aligned recurrence and have been shown to perform well on simple-language question answering and language modeling tasks.

To the best of our knowledge, however, the Transformer is the first transduction model relying entirely on self-attention to compute representations of its input and output without using sequence aligned RNNs or convolution.

3 Model Architecture

Most competitive neural sequence transduction models have an encoder-decoder structure (Bahdanau et al., 2014). Here, the encoder maps an input sequence of symbol representations \((x_1, ..., x_n)\) to a sequence of continuous representations \(z = (z_1, ..., z_N)\). Given \(z\), the decoder then generates an output sequence \((y_1, ..., y_m)\) of symbols one element at a time. At each step the model is auto-regressive (Graves, 2013), consuming the previously generated symbols as additional input when generating the next.
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):
        "Follow Figure 1 (left) for connections."
        for i in range(N):
            x = self.layers[i](x, mask)
        return self.norm(x)

We employ a residual connection (He et al., 2016) around each of the two sub-layers, followed by layer normalization (Ba et al., 2016).

class LayerNorm(nn.Module):
    "Construct a layernorm module (See citation for details)."
    def __init__(self, size, eps=1e-6):
        super(LayerNorm, self).__init__()
        self.size = size
        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

That is, the output of each sub-layer is LayerNorm(x + Sublayer(x)), where Sublayer(x) is the function implemented by the sub-layer itself. We apply dropout (Srivastava et al., 2014) to the output of each sub-layer, before it is added to the sub-layer input and normalized.

To facilitate these residual connections, all sub-layers in the model, as well as the embedding layers, produce outputs of dimension $d_{model} = 512$.

class SublayerConnection(nn.Module):
    "A layer norm followed by a residual connection. Note norm is not applied to residual x."
    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 sublayer fn."
        return x + self.dropout(sublayer(self.norm(x)))

Each layer has two sub-layers. The first is a multi-head self-attention mechanism, and the second is a simple, position-wise fully connected feed-forward network.

class EncoderLayer(nn.Module):
    "Encoder calls self-attn and feed forward."
    def __init__(self, size, self_attn, src_attn, feed_forward, dropout):
        super(EncoderLayer, self).__init__()
        self.self_attn = self_attn
        self.src_attn = src_attn
        self.feed_forward = feed_forward
        self.sublayer = clones(SublayerConnection(size, dropout), 2)

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

class DecoderLayer(nn.Module):
    "Decoder calls self-attn, src-attn, and feed forward."
    def __init__(self, size, self_attn, src_attn, feed_forward, dropout):
        super(DecoderLayer, self).__init__()
        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 (left) for connections."
        m = memory
        x = self.sublayer[0](x, lambda s: self.self_attn(x, x, x, t_mask))
        x = self.sublayer[1](x, lambda s: self.src_attn(x, m, m, s_mask))
        x = self.sublayer[2](x, self.feed_forward)

3.1 Decoder

The decoder is also composed of a stack of $N = 6$ identical layers.

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 i in range(N):
            x = self.layers[i](x, memory, src_mask, tgt_mask)
        return self.norm(x)

In addition to the two sub-layers in each encoder layer, the decoder inserts a third sub-layer, which performs multi-head attention over the output of the encoder stack. Similar to the encoder, we employ residual connections around each of the sub-layers, followed by layer normalization.

class DecoderLayer(nn.Module):
    "Decoder calls self-attn, src-attn, and feed forward."
    def __init__(self, layer, src_attn, feed_forward, dropout):
        super(DecoderLayer, self).__init__()
        self.self_attn = self_attn
        self.src_attn = src_attn
        self.feed_forward = feed_forward
        self.sublayer = clones(SublayerConnection(layer.size, dropout), 3)

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

We also modify the self-attention sub-layer in the decoder stack to prevent positions from attending to subsequent positions. This masking, combined with fact that the output embeddings are offset by one position, ensures that the predictions for position $i$ can depend only on the known outputs at positions less than $i$.

class EncoderLayer(nn.Module):
    "Encoder calls self-attn and feed forward."
    def __init__(self, self_attn, src_attn, feed_forward, dropout):
        super(EncoderLayer, self).__init__()
        self.self_attn = self_attn
        self.src_attn = src_attn
        self.feed_forward = feed_forward
        self.sublayer = clones(SublayerConnection(layer.size, dropout), 2)

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

An attention function can be described as mapping a query and a set of key-value pairs to an output, where the query, keys, values, and output are all vectors. The output is computed as a weighted sum of the values, where the weight assigned to each value is computed by a compatibility function of the query with the corresponding key.

We call our particular attention “Scaled Dot-Product Attention”. The input consists of queries and keys of dimension $d_k$, and values of dimension $d_v$. We compute the dot products of the query with all keys, divide each by $\sqrt{d_k}$, and apply a softmax function to obtain the weights on the values.

In practice, we compute the attention function on a set of queries simultaneously, packed together into a matrix $Q$. The keys and values are also packed together into matrices $K$ and $V$. We compute the matrix of outputs as:

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

```python
def attention(query, key, value, mask=None, dropout=None):
    d_k = query.size(-1)
    key_t = key.transpose(-2, -1)
    scores = torch.matmul(query, key_t) / 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
```

The two most commonly used attention functions are additive attention (Bahdanau et al., 2014), and dot-product (multiplicative) attention. Dot-product attention is identical to our algorithm, except for the scaling factor of $\frac{1}{\sqrt{d_k}}$. Additive attention computes the compatibility function using a feed-forward network with a single hidden layer. While the two are similar in theoretical complexity, dot-product attention is much faster and more space-efficient in practice, since it can be implemented using highly optimized matrix multiplication code.

While for small values of $d_k$ the two mechanisms perform similarly, additive attention outperforms dot product attention without scaling for larger values of $d_k$ (Britz et al., 2017). We suspect that for large values of $d_k$, the dot products grow large in magnitude, pushing the softmax function into regions where it has extremely small gradients (To illustrate why the dot products get large, assume that the components of $q$ and $k$ are independent random variables with mean 0 and variance 1. Then their dot product, $q \cdot k = \sum_{i=1}^{d_k} q_i k_i$, has mean 0 and variance $d_k$). To counteract this effect, we scale the dot products by $\frac{1}{\sqrt{d_k}}$.

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

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, ..., \text{head}_h) W^O$$

where $\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$

Where the projections are parameter matrices $W_i^Q \in \mathbb{R}^{d_{model} \times d_k}$, $W_i^K \in \mathbb{R}^{d_{model} \times d_k}$, $W_i^V \in \mathbb{R}^{d_{model} \times d_v}$ and $W^O \in \mathbb{R}^{hd_v \times d_{model}}$. In this work we employ $h = 8$ parallel attention layers, or heads. For each of these we use $d_k = d_v = d_{model} / h = 64$. Due to the reduced
dimension of each head, the total computational cost is similar to that of single-head attention with full dimensionality.

class MultiHeadedAttention(nn.Module):
    def __init__(self, head, d_model, dropout=0.1):
        """Take in model size and number of heads."""
        super(MultiHeadedAttention, self).__init__()
        self.d_k = d_model // head
        self.h = head
        self.linears = clones(nn.Linear(d_model, d_model), head)
        self.attn = None
        self.dropout = nn.Dropout(p=dropout)

    def forward(self, query, key, value, mask=None):
        """Implements Figure 2"""
        nb = query.size(0)
        if mask is not None:
            mask = mask.unsqueeze(1)
        # 1) Do all the linear projections in batch from d_model => h x d_k
        query, key, value = [l(x).view(nb, -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(nb, -1, self.h * self.d_k)
        return self.linear(x)

    def __repr__(self):
        return self.__class__.__name__ + '(' + str(self.linears[0]) + ')'

class Embeddings(nn.Module):
    def __init__(self, vocab, d_model):
        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(d_model)

3.2 Position-wise Feed-Forward Networks

In addition to attention sub-layers, each of the layers in our encoder and decoder contains a fully connected feed-forward network, which is applied to each position separately and identically. This consists of two linear transformations with a ReLU activation in between.

\[
FFN(x) = \max(0, xW_1 + b_1)W_2 + b_2
\]

While the linear transformations are the same across different positions, they use different parameters from layer to layer. Another way of describing this is as two convolutions with kernel size 1. The dimensionality of input and output is \(d_{\text{model}} = 512\), and the inner-layer has dimensionality \(d_{\text{ff}} = 2048\).

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))))

3.3 Embeddings and Softmax

Similarly to other sequence transduction models, we use learned embeddings to convert the input tokens and output tokens to vectors of dimension \(d_{\text{model}}\). We also use the usual learned linear transformation and softmax function to convert the decoder output to predicted next-token probabilities. In our model, we share the same weight matrix between the two embedding layers and the pre-softmax linear transformation, similar to (Press and Wolf, 2016). In the embedding layers, we multiply those weights by \(\sqrt{d_{\text{model}}}\).

class Embeddings(nn.Module):
    def __init__(self, vocab, d_model):
        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(d_model)

3.4 Positional Encoding

Since our model contains no recurrence and no convolution, in order for the model to make use of the order of the sequence, we must inject some information about the relative or absolute position of the tokens in the sequence. To this end, we add "positional encodings" to the input embeddings at the bottoms of the encoder and decoder stacks. The positional encodings have the same dimension \(d_{\text{model}}\) as the embeddings, so that the two can be summed. There are many choices of positional encodings, learned and fixed (Gehring et al., 2017).

In this work, we use sine and cosine functions of different frequencies:

\[
PE_{\text{pos}, 2i} = \sin(\text{pos}/10000^{2i/d_{\text{model}}})
\]

\[
PE_{\text{pos}, 2i+1} = \cos(\text{pos}/10000^{2i/d_{\text{model}}})
\]

where \(\text{pos}\) is the position and \(i\) is the dimension. That is, each dimension of the positional encoding corresponds to a sinusoid. The wavelengths form a geometric progression from \(2\pi\) to \(10000 \cdot 2\pi\). We chose this function because we hypothesized it would allow the model to easily learn to attend by relative positions, since for any fixed offset \(k\), \(PE_{\text{pos} + k}\) can be represented as a linear function of \(PE_{\text{pos}}\).

In addition, we apply dropout to the sums of the embeddings and the positional encodings in both the encoder and decoder stacks. For the base model, we use a rate of \(P_{\text{drop}} = 0.1\).
We also experimented with using learned positional embeddings (Gehring et al., 2017) instead, and found that the two versions produced nearly identical results. We chose the sinusoidal version because it may allow the model to extrapolate to sequence lengths longer than the ones encountered during training.

```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
attm = MultiHeadedAttention(h, d_model)
ff = PositionwiseFeedForward(d_model, d_ff, dropout)
position = PositionalEncoding(d_model, dropout)
d = d_model
model = EncoderDecoder(d_model, d_model, position, w=position, d=d_model)
model = EncoderDecoder(d_model, d_model, position, w=position, d=d_model)
model = EncoderDecoder(d_model, d_model, position, w=position, d=d_model)
model = Generator(d_model, tgt_vocab)
model = Generator(d_model, tgt_vocab)
model = 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
```

4 Training

This section describes the training regime for our models.

4.1 Batches and Masking

```python
class Batch:
    "Batch of data with mask for training."

def __init__(self, src, trg=None, pad=0):
```

4.2 Training Loop

```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:
            print("Epoch Step: %d Loss: %f Tokens / Sec: %f" % (i, loss / batch.ntokens, tokens / elapsed))
        elapsed = time.time() - start
        tokens = 0
        return total_loss / total_tokens
```

4.3 Training Data and Batching

We trained on the standard WMT 2014 English-German dataset consisting of about 4.5 million sentence pairs. Sentences were encoded using byte-pair encoding, which has a shared source-target vocabulary of about 37000 tokens. For English-French, we used the significantly larger WMT 2014 English-French dataset consisting of 36M sentences and split tokens into a 32000 word-piece vocabulary.

Sentence pairs were batched together by approximate sequence length. Each training batch contained a set of sentence pairs containing approximately 25000 source tokens and 25000 target tokens.

```python
global max_src_in_batch, max_tgt_in_batch

def batch_size_fn(new, count, sofar):
    return max_src_in_batch = 0
    if count == 1:
        max_src_in_batch = len(new.src)
        max_tgt_in_batch = len(new.trg)
        max_tgt_in_batch = len(new.trg)
        max_tgt_in_batch = len(new.trg)
        max_tgt_in_batch = len(new.trg)
        max_tgt_in_batch = len(new.trg)
        max_tgt_in_batch = len(new.trg)
        max_tgt_in_batch = len(new.trg)
        max_tgt_in_batch = len(new.trg)
        max_tgt_in_batch = len(new.trg)
        max_tgt_in_batch = len(new.trg)
        max_tgt_in_batch = len(new.trg)
        max_tgt_in_batch = len(new.trg)
        max_tgt_in_batch = len(new.trg)
        max_tgt_in_batch = len(new.trg)
        max_tgt_in_batch = len(new.trg)
        max_tgt_in_batch = len(new.trg)
        max_tgt_in_batch = len(new.trg)
        max_tgt_in_batch = len(new.trg)
        max_tgt_in_batch = len(new.trg)
        max_tgt_in_batch = len(new.trg)
        max_tgt_in_batch = len(new.trg)
        max_tgt_in_batch = len(new.trg)
        max_tgt_in_batch = len(new.trg)
        max_tgt_in_batch = len(new.trg)
        max_tgt_in_batch = len(new.trg)
        max_tgt_in_batch = len(new.trg)
        max_tgt_in_batch = len(new.trg)
        max_tgt_in_batch = len(new.trg)
        max_tgt_in_batch = len(new.trg)
        max_tgt_in_batch = len(new.trg)
        max_tgt_in_batch = len(new.trg)
        max_tgt_in_batch = len(new.trg)
        max_tgt_in_batch = len(new.trg)
        max_tgt_in_batch = len(new.trg)
        max_tgt_in_batch = len(new.trg)
        max_tgt_in_batch = len(new.trg)
        max_tgt_in_batch = len(new.trg)
        max_tgt_in_batch = len(new.trg)
        max_tgt_in_batch = len(new.trg)
        max_tgt_in_batch = len(new.trg)
        max_tgt_in_batch = len(new.trg)
        max_tgt_in_batch = len(new.trg)
        max_tgt_in_batch = len(new.trg)
        max_tgt_in_batch = len(new.trg)
        max_tgt_in_batch = len(new.trg)
        max_tgt_in_batch = len(new.trg)
        max_tgt_in_batch = len(new.trg)
        max_tgt_in_batch = len(new.trg)
```
out the paper, each training step took about 0.4 seconds. We trained the base models for a total of 100,000 steps or 12 hours. For our big models, step time was 1.0 seconds. The big models were trained for 300,000 steps (3.5 days).

4.5 Optimizer

We used the Adam optimizer (Kingma and Ba, 2014) 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:

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

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.

4.6 Regularization

4.6.1 Label Smoothing

During training, we employed label smoothing of value $\epsilon_{ls} = 0.1$ (Szegedy et al., 2015). This hurts perplexity, as the model learns to be more unsure, but improves accuracy and BLEU score.

```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))
```

# 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)
```
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]])
    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)])

5.1 Greedy Decoding

def greedy_decode(model, src, src_mask,
                   max_len, start_sym):
    memory = model.encode(src, src_mask)
    ys = torch.ones(1, 1).fill_(start_sym).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.argmax(prob, dim=-1)
        ys = torch.cat((ys, next_word.unsqueeze(-1)), dim=1)

    return ys

model.eval()
sent = "@@@The log file can be sent secretly with email or FTP to a specified receiver.".split()
sent = torch.LongTensor([[SRC.stoi[w] for w in sent]])
src = torch.LongTensor([[SRC.stoi[w]] for w in sent])
src_mask = src.ne(0)
out = greedy_decode(model, src, src_mask)
References

Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. 2016. Layer normalization. *arXiv preprint arXiv:1607.06450*.

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2014. Neural machine translation by jointly learning to align and translate. *CoRR*, abs/1409.0473.

Denny Britz, Anna Goldie, Minh-Thang Luong, and Quoc V. Le. 2017. Massive exploration of neural machine translation architectures. *CoRR*, abs/1703.03906.

Jonas Gehring, Michael Auli, David Grangier, Denis Yarats, and Yann N. Dauphin. 2017. Convolutional sequence to sequence learning. *CoRR*, abs/1705.03122.

Alex Graves. 2013. Generating sequences with recurrent neural networks. *CoRR*, abs/1308.0850.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778.

Diederik P. Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980.

Ofir Press and Lior Wolf. 2016. Using the output embedding to improve language models. *CoRR*, abs/1608.05859.

Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: A simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958.

Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens, and Zbigniew Wojna. 2015. Rethinking the inception architecture for computer vision. *CoRR*, abs/1512.00567.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *CoRR*, abs/1706.03762.